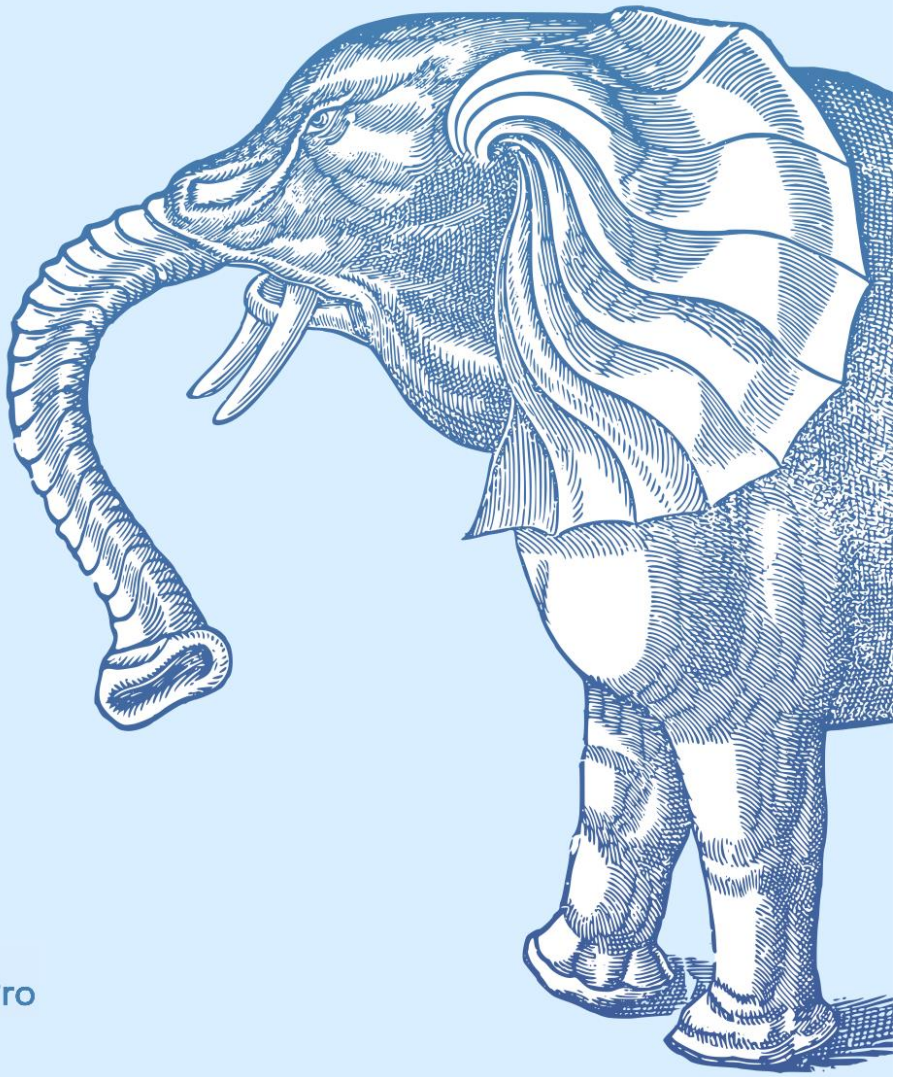
Egor Rogov

PostgreSQL 14 Internals



PostgresPro

Egor Rogov

PostgreSQL 14 Internals

Postgres Professional
Moscow, 2023

표지의 코끼리는 1658년 런던에서 출간된 에드워드 탑셀Edward Topsell의 『네 발 달린 짐승과 뱀의 역사』에 나오는 삽화의 일부입니다.

PostgreSQL 14 Internals
by Egor Rogov
Translated from English by MyoungSig,Youn

ISBN 978-5-6045970-4-0

This book in PDF is available at postgrespro.com/community/books/internals

번역: 윤명식 (jazzlian@gmail.com)

목차

이 책에 관해서

> 책은 믿기 위해 만들어지는 게 아니라
> 탐구하도록 만들어졌습니다.
> - 움베르토 에코Umberto Eco, 장미의 이름The Name of the Rose

이 책은 누구를 위한 책인가요?

이 책은 데이터베이스 작업에 있어 블랙박스 접근 방식에 만족하지 않는 분들을 위한 책입니다. 배우고자 하는 열정이 있고, 전문가의 조언을 그대로 받아들이기보다는 스스로 모든 것을 파악하고 싶어 하는 분이라면 이 책을 따라오시기 바랍니다.

저는 독자가 이미 `PostgreSQL`을 사용해 본 경험이 있고, 그 작동 방식에 대해 어느 정도 기본적인 이해가 있다고 가정합니다. 초보자는 이 책의 내용을 다소 어렵게 느낄 수 있습니다. 예를 들어, 서버 설치 방법, psql 명령 입력 방법, 설정 매개변수 지정 방법 등에 대해서는 설명하지 않을 것입니다.

또한 이 책은 다른 데이터베이스 시스템에 익숙하지만 `PostgreSQL`로 전환하면서 그 차이점을 이해하고자 하는 분들에게도 유용할 것이라 생각합니다. 몇 년 전에 이런 책이 있었다면 저 역시 많은 시간을 절약할 수 있었을 것입니다. 그래서 저는 마침내 이 책을 집필하게 되었습니다.

이 책에서 다루지 않는 내용

이 책은 단순한 레시피 모음집이나 튜토리얼이 아닙니다. 모든 상황에 대한 해결책을 제공하지는 않지만, 복잡한 시스템의 원리를 이해함으로써 다른 사람들의 경험을 분석하고 비판적으로 평가하여 자신만의 결론을 도출할 수 있도록 돕습니다. 이를 위해 처음에는 실용적인 가치가 불분명해 보이는 세부 사항들도 다룰 것입니다.

이 책은 제가 더 관심 있는 몇 가지 분야에 대해 깊이 있게 다루지만, 다른 분야는 전혀 다루지 않을 수도 있습니다. 또한, 이 책은 레퍼런스가 아닙니다. 정확성을 최대한 유지하려 노력했지만, 문서를 대체하는 것이 목적이 아니므로 중요하지 않다고 생각되는 일부 세부 사항은 생략했을 수 있습니다. 불확실한 상황이 있다면 항상 공식 문서를 참고하시기를 바랍니다.

이 책은 `PostgreSQL`의 코어 개발 방법을 가르치지 않으며, `C` 언어에 대한 지식을 요구하지 않습니다. 이 책의 주요 대상은 데이터베이스 관리자와 애플리케이션 개발자입니다. 그러나 소스 코드에 대한 참조를 제공함으로써 원하는 만큼의 세부 정보를 얻을 수 있도록 도울 것입니다.

이 책에서 제공하는 내용

서문에서는 이 책의 후속 내용을 이해하는 데 필요한 주요 데이터베이스 개념을 간단히 소개합니다. 물론, 이 장에서는 많은 새로운 정보를 얻지 못할 수도 있지만, 전체적인 이해를 위해 필요합니다. 특히 다른 데이터베이스 시스템에서 넘어오는 분들에게 유용한 정보를 제공합니다.

1부에서는 데이터 일관성과 격리에 대해 중점적으로 다룹니다. 먼저 사용자 관점에서 이를 탐구하고 (사용할 수 있는 격리 수준과 그 효과를 배울 것입니다), 그 후 내부 동작을 자세히 설명합니다. 이를 위해 다중 버전

동시성 제어와 스냅숏 격리의 구현 세부 사항, 그리고 오래된 행 버전 정리에 관한 주의 깊은 설명을 제공합니다.

2부에서는 버퍼 캐시와 `WAL`(장애 발생 후 데이터 일관성 복구에 사용됨)에 관해 설명합니다.

3부에서는 다양한 종류의 잠금lock 구조와 사용 방법에 관해 주로 다룹니다. 메모리RAM에 대한 경량lightweight 잠금, 관계relations에 관한 무거운heavyweight 잠금 및 행 수준row-level 잠금을 위해 사용됩니다.

4부에서는 서버가 어떻게 `SQL` 쿼리를 계획하고 실행하는지에 대해 설명합니다. 사용할 수 있는 데이터 접근 방법, 가능한 조인join 방법, 그리고 수집된 통계가 어떻게 적용되는지 알려드릴 것입니다.

5부에서는 `B`-트리 이외의 다른 인덱스 접근 방법에 관해 논의를 확장합니다. 인덱싱 시스템의 핵심, 인덱스 접근 방법, 그리고 데이터 유형 간 경계를 정의하는 일반적인 원칙 등을 설명합니다. 그 후 각 사용 가능한 방법에 관해 자세히 다룰 것입니다.

`PostgreSQL`에는 루틴 작업에서 사용되지 않지만, 서버의 내부 동작을 살펴볼 수 있는 "인트로스펙티브introspective"한 확장 기능이 많이 포함되어 있습니다. 이 책에서는 이런 확장 기능을 활용합니다. 이러한 확장 기능은 서버 내부를 이해하는 데 도움을 주며, 복잡한 사용 시나리오에서 문제 해결을 쉽게 만들어 줍니다.

규칙

이 책은 페이지 순서대로 읽으실 수 있도록 구성되었습니다. 그렇지만 모든 내용을 한 번에 설명하기는 어려워, 같은 주제를 여러 번 다루게 되었습니다. "이후에 다룰 예정"이라는 말을 반복적으로 쓰는 것은 텍스트를 길게 만들 수 있어, 대신 페이지 번호를 표기하여 추가 설명이 필요한 부분을 참조하실 수 있게 하였습니다. 또한, 이전에 언급된 주제를 참조할 때도 페이지 번호를 표기하였습니다.

이 책의 내용과 모든 코드 예제는 `PostgreSQL`의 버전 14에 적용됩니다. 일부 내용 옆에는 버전 번호를 표기하였는데, 이는 해당 정보가 해당 버전의 `PostgreSQL`부터 적용된다는 것을 의미합니다. 이전 버전에서는 해당 기능이 없거나 다르게 구현되었습니다. 이런 참조는 아직 최신 버전으로 업그레이드하지 않은 분들에게 도움이 될 것입니다.

이 책에서는 여백을 활용해 매개변수의 기본값을 표시합니다. 일반 매개변수와 저장 매개변수의 이름은 이탤릭체로 표시됩니다, 예를 들어, *work_mem*.

각주에서는 다양한 정보의 출처로 이어지는 링크를 제공합니다. 이런 출처 중 가장 먼저 언급되는 것은 `PostgreSQL` 문서[1]입니다. 이 문서는 지식의 근원이며, `PostgreSQL` 개발자들이 항상 최신 정보를 유지하므로 프로젝트에 꼭 필요한 자료입니다. 그러나 가장 중요한 참고 자료는 소스 코드[2]입니다. 소스 코드를 읽고 주석을 보면, `C` 언어에 익숙하지 않아도 많은 정보를 얻을 수 있습니다. 때때로 `commitfest` 항목[3]도 참조합니다. 이메일 내용을 정독하기는 어려울 수 있지만, `psql-hackers` 메일링 리스트에서 관련 토론을 읽으면 모든 변경 사항의 배경과 개발자들의 결정 과정을 이해할 수 있습니다.

---

[1] `postgresql.org/docs/14/index.html`

[2] `git.postgresql.org/gitweb/?p=postgresql.git;a=summary`

[3] `commitfest.postgresql.org`

> 이 책에는 주제와 조금 벗어난 부가적인 내용도 포함되어 있습니다. 이런 내용은 다르게 표시하여 빠르게 건너뛸 수 있도록 하였습니다.

당연히, 이 책에는 주로 PostgreSQL에서 사용되는 다양한 코드 예제들이 포함되어 있습니다. 코드는 '=>' 프롬프트와 함께 제공되며, 필요한 경우 서버의 응답이 이어서 표시됩니다.

```
=> SELECT now();
now
──────────────────────────────────
2023-03-06 14:00:08.008545+03
(1 행)
```

만약 이 책에 나온 모든 명령을 PostgreSQL 14 에서 주의 깊게 따라 한다면, 정확히 같은 결과를 얻을 수 있어야 합니다. 이는 트랜잭션의 세부 사항 같은 불필요한 부분까지도 포함됩니다. 실제로, 이 책의 모든 코드 예제는 이와 같은 명령을 사용한 스크립트로 생성되었습니다.

한편, 여러 트랜잭션을 동시에 실행하는 부분을 설명하기 위해서는 다른 세션에서 실행되는 코드 부분을 들여쓰기하고 수직선으로 구분하였습니다.

```
=> SHOW server_version;
server_version
──────────────────
14.7
(1 행)
```

이런 명령들을 직접 시도해보는 것은 자습이나 실험에 유용합니다. 이를 위해서는 두 개의 psql 터미널을 열어 놓는 것이 좋습니다.

명령어와 데이터베이스 객체의 이름(예: 테이블 및 열, 함수, 확장 등)은 D2Coding체를 사용하여 강조 표시하였습니다. 예를 들어, UPDATE, pg_class 등입니다. 운영 체제에서 유틸리티를 호출하는 경우는 '$'로 끝나는 프롬프트로 표시합니다. 예를 들어,

```
postgres$ whoami
postgres
```

저는 리눅스를 사용하였지만, 특별한 기술은 요구하지 않습니다. 해당 운영 체제에 관한 기본적인 이해만 있다면 충분합니다.

감사 인사

혼자서 책을 쓰는 것은 불가능합니다. 이제는 이 책을 함께 만들어준 훌륭한 사람들에게 감사의 말씀을 전할 기회입니다.

파벨 루자노프Pavel Luzanov에게는 이 가치 있는 일을 시작하도록 제안해 준 데 대해 진심으로 감사드립니다. 또한 제가 이 책 작업에 많은 시간을 투자할 수 있도록 해준 포스트그레스 프로페셔널Postgres Professional에도 감사의 말씀을 전합니다. 하지만 회사 뒤에는 항상 사람들이 있습니다. 올레그 바르투노프Oleg Bartunov는 아이디어

와 에너지를 무한히 공유해주었고, 이반 판첸코Ivan Panchenko는 지원을 아끼지 않고 라텍스LATEX를 제공해 주었습니다. 이 두 분에게 깊은 감사의 말씀을 전합니다.

교육팀 동료들에게는 창의적인 분위기와 토론에 감사드리며, 우리 교육 과정의 형태를 만드는 데 크게 이바지 해준 것에 감사드립니다. 초안을 세심하게 검토해 준 파벨 톨마체프Pavel Tolmachev에게는 특별히 감사의 말씀을 전합니다.

이 책의 많은 장들은 먼저 `Habr` 블로그[4]에서 기사로 게시되었습니다. 독자들로부터 받은 의견과 피드백에 감사드립니다. 이 덕분에 이 작업의 중요성을 깨닫고, 제 지식의 빈틈을 찾아내어 내용을 개선하는 데 도움이 되었습니다.

이 책의 언어를 다듬는 데 많은 노력을 기울인 리우드밀라 만트로바Liudmila Mantrova에게도 감사의 말씀을 전합니다. 그녀 덕분에 각 문장이 매끄럽게 읽힙니다. 또한, 리우드밀라는 이 책을 영어로 번역하는 데도 큰 역할을 해주었습니다. 그녀의 수고에 진심으로 감사드립니다.

이 책에서 언급된 각 함수나 기능은 수많은 사람들이 수년 동안 노력한 결과입니다. 저는 `PostgreSQL` 개발자들을 존경하며, 그들 중 많은 분을 동료로 불러, 그 영광을 누리게 되어 매우 기쁩니다.

[4] `habr.com/en/company/postgrespro/blog`

1. 소개

1.1. 데이터 구성

데이터베이스

`PostgreSQL`은 데이터베이스를 관리하는 프로그램의 한 종류입니다. 이 프로그램이 작동하면, 이를 'PostgreSQL 서버' 또는 인스턴스instance라고 합니다.

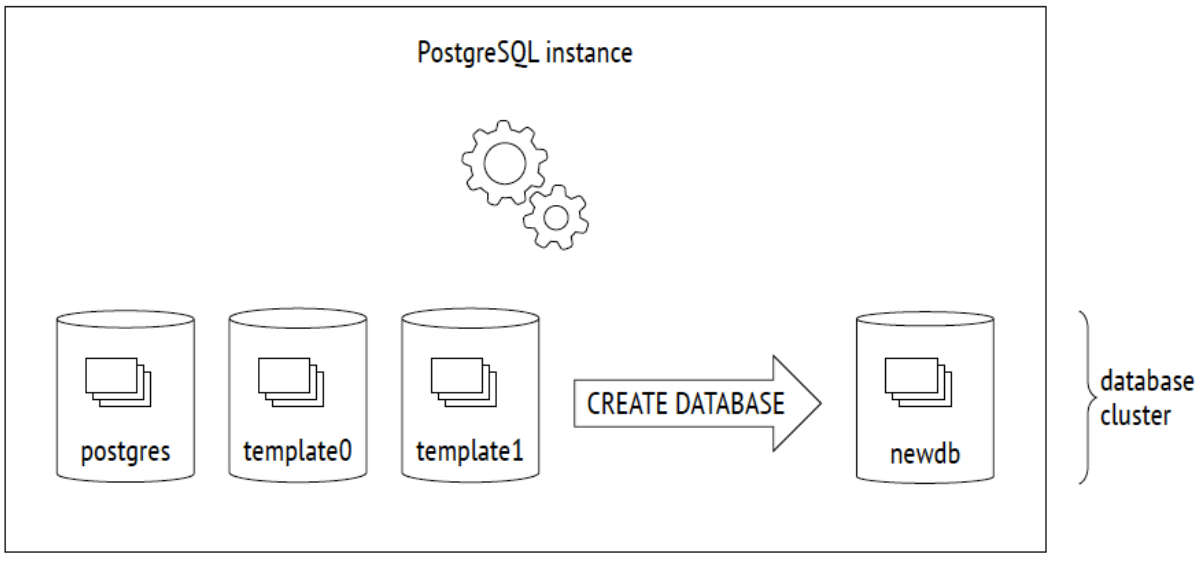
`PostgreSQL`이 관리하는 모든 데이터는 데이터베이스에 저장되며[5], 한 번에 여러 개의 데이터베이스를 관리할 수 있습니다. 이렇게 여러 데이터베이스를 한데 묶은 것을 '데이터베이스 클러스터'라고 부릅니다.

클러스터를 사용하려면 먼저 만들어야 합니다.[6] 클러스터와 관련된 모든 파일이 저장된 폴더는 보통 '데이터 디렉터리'라고 불립니다. 이 이름은, 이 폴더를 가리키는 `PGDATA` 환경 변수의 이름에서 유래된 것입니다.

> 미리 만들어진 패키지로 설치할 때는, 필요한 모든 설정값을 직접 지정해서 `PostgreSQL`의 일반적인 기능 위에 별도의 '추상화 계층abstraction layers'을 만들 수 있습니다. 이럴 때, 데이터베이스 서버는 운영체제의 서비스로 실행되며, `PGDATA` 변수를 직접 설정할 필요는 없을 수 있습니다. 하지만 이 용어는 널리 알려져 있으므로 사용하겠습니다.

클러스터를 초기화하면, `PGDATA`에는 세 가지 데이터베이스가 들어 있습니다:

- **`template0`**은 논리적 백업에서 데이터를 복구하거나, 인코딩이 다른 데이터베이스를 만들 때 등의 경우에 사용합니다. 이것은 절대로 변경하면 안 됩니다.
- **`template1`**은 사용자가 클러스터에서 새로운 데이터베이스를 만들 때 사용하는 템플릿입니다.
- **`postgres`**는 자유롭게 사용할 수 있는 일반적인 데이터베이스입니다.



[5] `postgresql.org/docs/14/managing-databases.html`

[6] `postgresql.org/docs/14/app-initdb.html`

시스템 카탈로그

클러스터의 객체들(테이블, 인덱스, 데이터 타입 또는 함수 등)의 메타데이터는 시스템 카탈로그system catalog라는 테이블에 저장됩니다.[7] 각 데이터베이스는 자신의 객체를 설명하는 고유한 테이블과 뷰의 집합을 하고 있습니다. 몇몇 시스템 카탈로그 테이블은 전체 클러스터에서 공용으로 사용되고, 특정 데이터베이스에 속하지 않습니다. 이 테이블들은 모든 데이터베이스에서 접근할 수 있습니다.

시스템 카탈로그는 일반적인 SQL 질의를 통해 확인할 수 있고, 수정은 관리자 명령으로 이루어집니다. psql 클라이언트는 시스템 카탈로그의 내용을 보여주는 다양한 명령어를 제공합니다.

시스템 카탈로그 테이블의 이름은 'pg_'로 시작합니다. 예를 들어 pg_database와 같은 형태입니다. 칼럼 이름은 테이블 이름에 해당하는 세 글자 접두사로 시작하는 경우가 일반적입니다. 예를 들면 datname이 있습니다.

시스템 카탈로그 테이블에서 기본 키로 선언된 칼럼은 oid(객체 식별자)라고 부르며, 이 칼럼의 데이터 타입인 oid도 32비트 정수입니다.

> oid 라는 객체 식별자의 작동 방식은 사실상 시퀀스 sequences와 매우 비슷합니다. 그러나 이런 방식은 PostgreSQL 에 훨씬 먼저 존재했습니다. 이런 방식이 특별한 이유는, 공통 카운터에서 발급된 고유 식별자가 시스템 카탈로그의 다른 테이블에서도 사용되기 때문입니다. 만약 할당된 식별자가 최댓값을 넘어가면, 카운터는 다시 시작합니다. 특정 테이블의 모든 값이 고유하게 유지되도록, 다음에 발급될 oid 는 고유 unique 인덱스를 통해 확인됩니다. 만약 이미 해당 테이블에서 사용 중이라면, 카운터를 증가시키고 다시 확인하는 과정을 반복합니다.[8]

스키마

스키마Schemas[9]는 데이터베이스의 모든 객체를 저장하는 공간입니다. 사용자 정의 스키마 외에도, PostgreSQL은 몇 가지 미리 정의된 스키마를 제공합니다:

- **public**은 사용자 객체의 기본 스키마로, 다른 설정이 없을 때 사용됩니다.
- **pg_catalog**는 시스템 카탈로그 테이블을 위해 사용됩니다.
- **information_schema**는 SQL 표준에서 정의하는 시스템 카탈로그의 대체 뷰를 제공하는 데 사용됩니다.
- **pg_toast**는 TOAST 기능과 관련된 객체를 위해 사용됩니다.
- **pg_temp**는 임시 테이블을 만드는 데 사용됩니다. 여러 사용자가 pg_temp_N이라는 서로 다른 스키마에서 임시 테이블을 생성하지만, 모두 pg_temp 별칭을 사용하여 객체에 접근합니다.

각 스키마는 특정 데이터베이스에 한정되며, 모든 데이터베이스 객체는 이 스키마에 속해 있습니다.

객체에 접근할 때 스키마가 명시적으로 지정되지 않으면, PostgreSQL은 검색 경로에서 가장 먼저 적합한 스키마를 선택합니다. 검색 경로는 search_path 매개변숫값에 기반하며, 묵시적으로 pg_catalog와 필요한 경

---

[7] postgresql.org/docs/14/catalogs.html

[8] backend/catalog/catalog.c, GetNewOidWithIndex function

[9] postgresql.org/docs/14/ddl-schemas.html

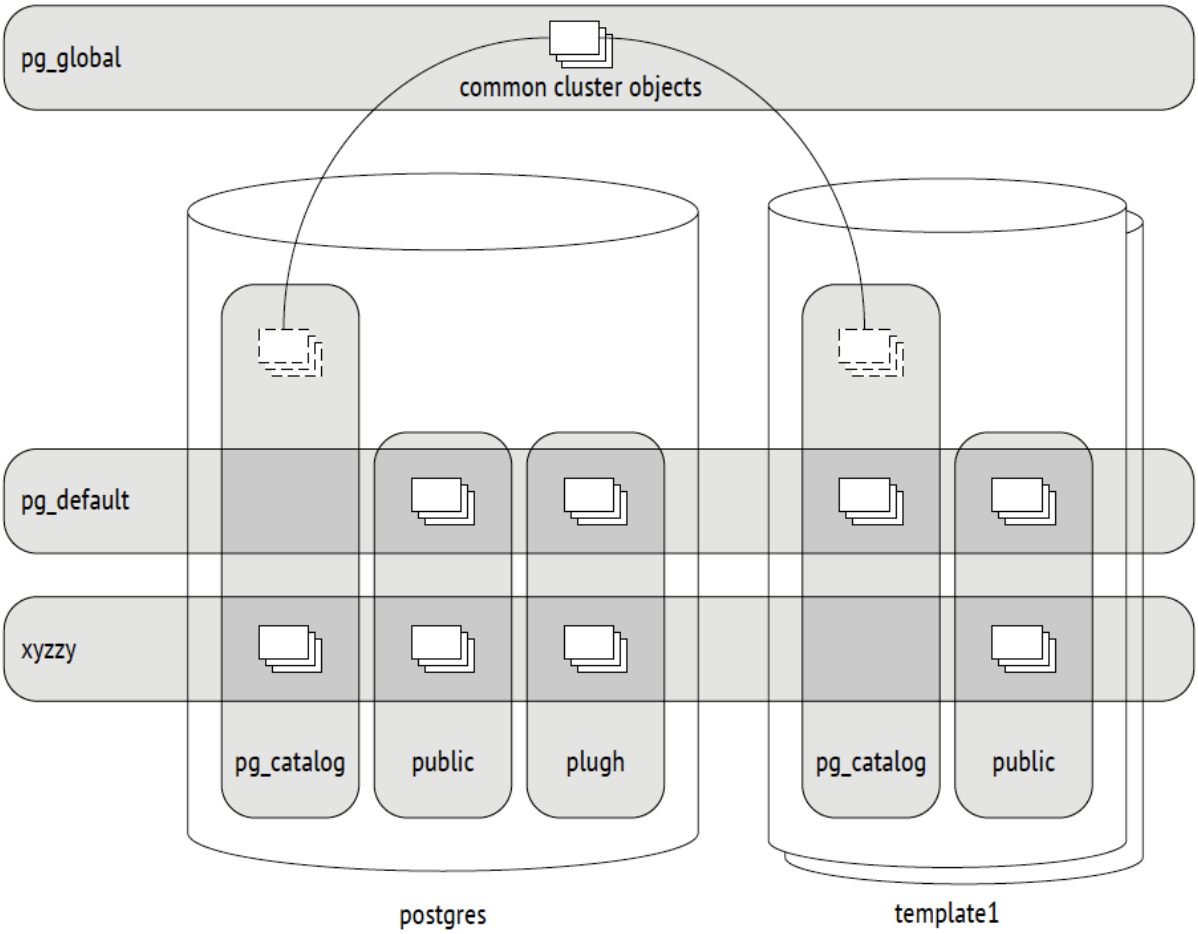
우 `pg_temp`스키마와 함께 확장됩니다. 따라서, 서로 다른 스키마에 같은 이름의 객체가 존재할 수 있습니다.

테이블 스페이스

데이터베이스와 스키마는 객체가 어떻게 논리적으로 분포되는지를 결정하는 반면, 테이블 스페이스Tablespaces는 데이터가 물리적으로 어떻게 배열되는지를 정의합니다. 테이블 스페이스는 본질적으로 파일 시스템의 디렉터리입니다. 테이블 스페이스를 활용하면 데이터를 분산시키는 것이 가능하며, 이를 통해 자주 접근되거나 수정되는 데이터는 빠른 디스크에, 비교적 접근이 적은 데이터는 느린 디스크에 저장할 수 있습니다.

같은 테이블 스페이스는 여러 데이터베이스에서 공유하여 사용할 수 있고, 각 데이터베이스는 여러 테이블 스페이스에 데이터를 저장할 수 있습니다. 이는 논리적인 구조와 물리적인 데이터 배치가 서로 독립적임을 의미합니다.

모든 데이터베이스는 자체 기본 테이블 스페이스를 가지고 있습니다. 특별한 위치가 지정되지 않는다면, 모든 데이터베이스 객체는 이 기본 테이블 스페이스에 생성됩니다. 또한, 이 데이터베이스와 관련된 시스템 카탈로그 객체도 해당 테이블 스페이스에 저장됩니다.



클러스터를 초기화할 때, 두 개의 테이블 스페이스가 생성됩니다:

- `pg_default`는 `$PGDATA/base` 디렉터리에 위치하며, 특정 테이블 스페이스가 명시되지 않는 경우 기본적으로 사용되는 테이블 스페이스입니다.

- `pg_global`은 `$PGDATA/global` 디렉터리에 위치하고, 클러스터 전체에서 공용으로 사용되는 시스템 카탈로그 객체를 저장합니다.

사용자가 직접 테이블 스페이스를 생성할 때, 원하는 디렉터리를 지정할 수 있습니다. `PostgreSQL`은 이 위치에 관한 심볼릭 링크를 `$PGDATA/pg_tblspc` 디렉터리에 생성합니다. 실제로 `PostgreSQL`이 사용하는 모든 경로는 `$PGDATA/base` 디렉터리를 기준으로, 상대적으로 지정되므로, 서버를 중지한 상태에서 다른 위치로 이동할 수 있습니다.

이전 페이지의 그림에서는 데이터베이스, 스키마, 테이블 스페이스가 함께 표현되어 있습니다. 그림에서 `postgres` 데이터베이스는 `xyzzy`를 기본 테이블 스페이스로 사용하고, `template1` 데이터베이스는 `pg_default`를 사용하고 있습니다. 다양한 데이터베이스 객체는 테이블 스페이스와 스키마의 교차점에 있습니다.

릴레이션

데이터베이스에서 가장 중요한 객체인 테이블과 인덱스는 서로 다른 특성이 있지만, 한 가지 공통점을 가지고 있습니다. 바로 행으로 구성되어 있다는 점입니다. 테이블은 이해하기 쉽지만, 이 개념은 `B`-트리 노드에도 마찬가지로 적용됩니다. B-트리 노드는 인덱스값과 다른 노드 또는 테이블 행에 관한 참조를 포함하고 있습니다.

다른 몇몇 객체들도 같은 구조로 되어 있습니다. 예를 들어, 시퀀스는 사실상 한 행만 있는 테이블이며, 실체화된 뷰materialized views는 해당 질의의 결과를 저장하는 테이블로 볼 수 있습니다. 또한 데이터를 저장하지 않지만, 테이블과 매우 유사한 구조를 가진 뷰도 있습니다.

`PostgreSQL`에서는 이러한 모든 객체를 '릴레이션Relations'이라는 일반적인 용어로 표현합니다.

> 제 개인적인 생각으로는, '릴레이션'이라는 용어는 데이터베이스 테이블과 관계 이론에서 정의된 '진정한' 릴레이션을 혼동하게 만들어, 좋은 용어라고 보기 어렵습니다. 여기서 느껴지는 것은 PostgreSQL 프로젝트의 학문적 배경과 창시자 마이클 스톤브레이커 Michael Stonebraker 의 성향이, 즉 모든 것을 '릴레이션'으로 보려는 경향이 드러나는 것입니다. 그의 다른 작업에서는 '정렬된 릴레이션'이라는 개념을 도입하여, 행의 순서가 인덱스에 의해 정의되는 테이블에 관해 설명하였습니다.
>
> 시스템 카탈로그 테이블에 관한 '릴레이션'이라는 용어는 원래 `pg_relation`으로 불렸습니다. 하지만 객체 지향적인 추세를 따라 `pg_class`로 이름이 바뀌었습니다. 이제 우리는 이 이름에 익숙해졌습니다. 하지만 해당 테이블의 칼럼들은 여전히 `rel` 접두사를 가지고 있습니다.

파일과 포크

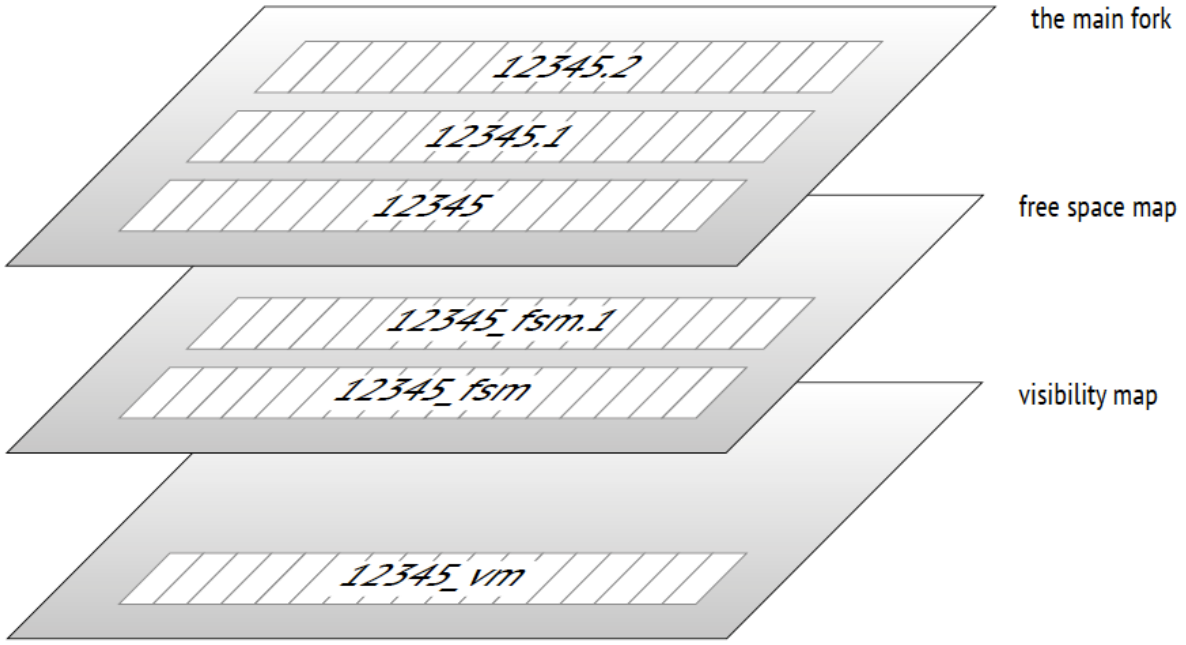
릴레이션과 관련된 모든 정보는 여러 개의 다른 포크forks라는 곳에 저장되며[10], 각 포크는 특정한 유형의 데이터를 담고 있습니다.

[10] `postgresql.org/docs/14/storage-file-layout.html`

초기에는 포크가 하나의 파일로 표현되며, 이 파일의 이름은 숫자(`oid`)로 이루어져 있고, 포크 유형을 나타내는 접미사가 붙습니다.

이 파일은 시간이 흐르면서 점차 커지고, 크기가 `1GB`에 이르면 해당 포크에 새로운 파일이 생성됩니다. 이런 파일들을 세그먼트 segments라고도 부릅니다. 세그먼트의 순서 번호는 파일 이름의 뒷부분에 추가됩니다.

`1GB`의 파일 크기 제한은 이전에는 대용량 파일을 처리하지 못하는 몇몇 파일 시스템을 지원하기 위해 설정되었습니다. `PostgreSQL`을 빌드할 때 이 제한을 변경할 수 있습니다(`./configure --with-segsize` 옵션을 사용하여).



그래서, 한 릴레이션은 디스크에 여러 개의 파일로 나타나게 됩니다. 인덱스가 없는 작은 테이블이라도 필요한 포크의 수에 따라 최소 세 개의 파일을 가지게 됩니다.

각 테이블스페이스 디렉토리는(`pg_global` 제외) 특정 데이터베이스를 위한 별도의 하위 디렉터리를 포함하고 있습니다. 같은 테이블 스페이스와 데이터베이스에 속한 모든 객체의 파일은 같은 하위 디렉터리에 자리 잡고 있습니다. 하지만 파일 시스템이 한 디렉터리에 너무 많은 파일을 처리하지 못하는 경우가 있으므로, 이 점을 고려해야 합니다.

포크에는 여러 가지 표준 유형들이 있습니다.

주 포크는 테이블 행이나 인덱스 행과 같은 실제 데이터를 나타냅니다. 이 포크는 뷰를 제외한 모든 릴레이션에서 사용됩니다(뷰에는 데이터가 들어있지 않습니다).

주 포크의 파일들은 숫자 아이디로 이름이 지어지며, 이 값은 `pg_class` 테이블의 `relfilenode` 값에 저장됩니다.

`pg_default` 테이블 스페이스에서 만들어진 테이블의 파일 경로를 살펴봅시다.

```
=> CREATE UNLOGGED TABLE t(
a integer,
b numeric,
c text,
d json
);

=> INSERT INTO t VALUES (1, 2.0, 'foo', '{}');
=> SELECT pg_relation_filepath('t');

pg_relation_filepath
----------------------
base/16384/16385
(1 행)
```

기본 디렉터리는 `pg_default` 테이블 스페이스를 의미하며, 그 아래에 있는 하위 디렉터리는 데이터베이스에 사용됩니다. 이곳에서 우리가 찾는 파일을 찾을 수 있습니다.

```
=> SELECT oid FROM pg_database WHERE datname = 'internals';
oid
-------
16384
(1 행)

=> SELECT relfilenode FROM pg_class WHERE relname = 't';
relfilenode
-------------
16385
(1 행)
```

다음은 파일 시스템에 있는 해당 파일입니다:

```
=> SELECT size
FROM pg_stat_file('/usr/local/pgsql/data/base/16384/16385');
size
------
8192
(1 행)
```

초기화 포크[11]는 로그되지 않은 테이블(`UNLOGGED` 절로 생성된)과 그 테이블의 인덱스에만 사용됩니다. 이런 객체들은 일반 객체와 같지만, 그들에 관한 모든 작업은 사전에 기록write-ahead 로그에 남기지 않습니다. 이 때문에 이런 작업은 매우 빠르지만, 장애가 발생하면 일관된 데이터를 복구할 수 없습니다. 그래서 `PostgreSQL`은 복구 과정에서 이런 객체의 모든 포크를 간단히 삭제하고, 초기화 포크로 주 포크를 덮어쓰면서 더미 파일

[11] `postgresql.org/docs/14/storage-init.html`

을 생성합니다.

t 테이블은 로그되지 않은 상태로 생성되었기 때문에 초기화 포크가 있습니다. 이 포크는 주 포크와 같은 이름을 가지지만 뒤에 _init 접미사가 붙습니다.

```
=> SELECT size
FROM pg_stat_file('/usr/local/pgsql/data/base/16384/16385_init');
size
------
0
(1 행)
```

빈 공간 맵free space map [12]은 페이지 내의 사용 가능한 공간을 추적합니다. 맵의 크기는 항상 변하며, 백업 작업 후에는 커지고 새로운 행 버전이 생길 때마다 줄어듭니다. 빈 공간 맵은 새로 삽입되는 데이터를 수용할 수 있는 페이지를 빠르게 찾는 데 사용됩니다.

빈 공간 맵과 관련된 모든 파일은 _fsm 접미사를 가집니다. 처음에는 이런 파일들이 생성되지 않고, 필요할 때만 만들어집니다. 이런 파일을 가장 쉽게 얻는 방법은 테이블을 베큠vacuum하는 것입니다.

```
=> VACUUM t;

=> SELECT size
FROM pg_stat_file('/usr/local/pgsql/data/base/16384/16385_fsm');
size
-------
24576
(1 행)
```

검색 속도를 향상하기 위해, 빈 공간 맵은 트리 구조로 만들어집니다. 최소 세 개의 페이지가 필요하며, 이는 거의 비어 있는 테이블의 파일 크기입니다.

빈 공간 맵은 테이블과 인덱스 모두에 적용됩니다. 하지만 인덱스 행은 임의의 페이지에 추가될 수 없습니다 (예를 들어, B-트리는 삽입 위치를 정렬 순서에 따라 정합니다). 그래서 PostgreSQL은 인덱스 구조에서 재사용할 수 있는 완전히 빈 페이지만을 추적합니다.

가시성 맵visibility map[13]은 페이지가 베큠이나 프로즌frozen이 필요한지를 빠르게 확인할 수 있게 해줍니다. 이를 위해 각 테이블 페이지마다 두 개의 비트를 제공합니다.

첫 번째 비트는 최신 행 버전만 포함하는 페이지에 설정됩니다. 베큠 작업은 이런 페이지를 건너뛰므로, 정리할 내용이 없습니다. 또한, 트랜잭션이 이런 페이지에서 행을 읽을 때 가시성을 확인하는 것은 불필요하므로, 인덱스 전용 스캔을 사용할 수 있습니다.

---

[12] postgresql.org/docs/14/storage-fsm.html
backend/storage/freespace/README

[13] postgresql.org/docs/14/storage-vm.html

두 번째 비트는 프로즌된 행 버전만 포함하는 페이지에 설정됩니다. 이 포크의 이 부분을 프로즈 freeze 맵이라고 합니다.

가시성 맵 파일은 _vm 접미사를 가집니다. 일반적으로 이 파일은 가장 작은 크기를 가집니다.

```
=> SELECT size
FROM pg_stat_file('/usr/local/pgsql/data/base/16384/16385_vm');
size
------
8192
(1 행)
```

테이블에 관한 가시성 맵은 제공되지만 인덱스에 관한 가시성 맵은 제공되지 않습니다.

페이지

모든 파일은 입출력(I/O)을 쉽게 하기 위해 논리적으로 페이지Pages (또는 블록)로 분할됩니다. 이는 읽거나 쓸 수 있는 최소한의 데이터 양을 나타냅니다. 그래서 PostgreSQL의 많은 내부 알고리즘은 페이지 처리에 최적화되어 있습니다.

페이지 크기는 보통 8kb인데, 조정이 가능하지만(최대 32kb까지), 빌드 시간에만 설정할 수 있습니다 (./configure --with-blocksize). 대부분의 경우 이런 설정을 하지 않으며, 한 번 빌드하고 실행한 인스턴스는 동일한 크기의 페이지만 사용할 수 있습니다. 서로 다른 페이지 크기를 지원하는 테이블 스페이스를 만드는 것은 불가능합니다.

어떤 포크에 속하든, 모든 파일은 서버에서 거의 동일한 방식으로 처리됩니다. 페이지는 먼저 버퍼 캐시로 이동하여 프로세스에 의해 읽고 수정될 수 있고, 필요할 때 디스크로 플러시됩니다.

TOAST

각 행은 하나의 페이지에 들어가야 하며, 행을 다음 페이지로 넘겨서 이어나가는 방법은 없습니다. 긴 행을 저장하기 위해서 PostgreSQL은 TOAST[14](오버사이즈된 속성 저장 기술The Oversized Attributes Storage Technique)라는 특별한 방식을 사용합니다.

TOAST는 다양한 전략을 포함합니다. 긴 속성 값을 작은 TOAST로 분할하여 별도의 서비스 테이블로 옮길 수 있습니다. 또 다른 방법은 행이 페이지에 들어갈 수 있도록 긴 값을 압축하는 것입니다. 또는 값을 먼저 압축한 후에 분할하여 이동시키는 방법도 있습니다.

주 테이블에 긴 속성이 있을 가능성이 있다면, 그런 속성들을 위해 별도의 TOAST 테이블이 바로 만들어집니다. 예를 들어, 숫자나 텍스트 유형의 열이 있는 테이블에서는, 이 열이 긴 값을 저장하지 않아도 TOAST 테이블이 생성됩니다.

인덱스의 경우, TOAST 방식은 압축 기능만 제공하며, 긴 속성을 별도의 테이블로 이동시키는 것은 지원하지

[14] postgresql.org/docs/14/storage-toast.html
include/access/heaptoast.h

않습니다. 이는 인덱스 키의 크기를 제한하고, 인덱싱할 수 있는 키의 크기를 제한합니다(실제 구현은 특정 연산자 클래스에 따라 다릅니다).

기본적으로 `TOAST` 전략은 열의 데이터 유형에 따라 결정됩니다. 사용된 전략을 확인하는 가장 간단한 방법은 `psql`에서 `\d+` 명령을 실행하는 것입니다. 하지만, 저는 시스템 카탈로그를 조회하여 더 깔끔한 결과를 얻는 방법을 선호합니다.

```
=> SELECT attname, atttypid::regtype,
  CASE attstorage
   WHEN 'p' THEN 'plain'
   WHEN 'e' THEN 'external'
   WHEN 'm' THEN 'main'
   WHEN 'x' THEN 'extended'
 END AS storage
FROM pg_attribute
WHERE attrelid = 't'::regclass AND attnum > 0;

 attname ¦ atttypid ¦ storage
---------+----------+----------
a        ¦ integer  ¦ plain
b        ¦ numeric  ¦ main
c        ¦ text     ¦ extended
d        ¦ json     ¦ extended
(4 행s)
```

`PostgreSQL`은 다음과 같은 전략을 지원합니다:

- **plain**은 `TOAST`를 사용하지 않는 것을 의미하며, 이는 주로 '짧다'라고 알려진 데이터 유형에 적용됩니다. 예를 들어, 정수 유형이 이에 해당합니다.
- **extended**는 속성을 압축하고, 필요한 경우 별도의 `TOAST` 테이블에 저장할 수 있게 해줍니다.
- **external**은 긴 속성을 압축하지 않은 상태로 별도의 `TOAST` 테이블에 저장합니다.
- **main**은 긴 속성을 먼저 압축하고, 압축이 효과적이지 않을 때만 별도의 `TOAST` 테이블로 이동합니다.

일반적으로 알고리즘은 다음과 같이 작동합니다.[15] `PostgreSQL`은 페이지에 최소한 4개의 행이 들어가도록 설계되었습니다. 그래서 행의 크기가 페이지의 1/4(표준 크기 페이지의 경우 약 2,000바이트)을 넘을 경우(헤더는 제외), 일부 값을 `TOAST` 메커니즘을 적용해야 합니다. 아래에 설명된 과정을 따라, 행의 길이가 더 이상 임곗값을 초과하지 않을 때까지 진행합니다:

1. 먼저, 가장 긴 속성부터 시작하여 `external`과 `extended` 전략을 사용하는 속성을 확인합니다. `extended`된 속성은 압축되며, 결괏값(다른 속성을 고려하지 않고 독립적으로)이 페이지의 1/4을 초과하면 바로 `TOAST` 테이블로 이동됩니다. `external` 속성도 같은 방식으로 처리되지만, 압축 단계는 생략합니다.

[15] `backend/access/heap/heaptoast.c`

2. 첫 번째 패스 후에도 행이 페이지에 맞지 않으면, `external`과 `extended` 전략을 사용하는 나머지 속성을 하나씩 `TOAST` 테이블로 이동시킵니다.
3. 이렇게 해도 도움이 되지 않으면, `main` 전략을 사용하는 속성을 압축하여 테이블 페이지에 그대로 둡니다.
4. 행이 여전히 아주 짧지 않다면, `main` 속성을 `TOAST` 테이블로 이동시킵니다.

임곗값은 2,000`byte`이지만, `toast_tuple_target` 저장 매개변수를 사용해 테이블 단위로 이 값을 재정의할 수 있습니다.

가끔은 일부 열의 기본 전략을 바꾸는 것이 도움이 될 수 있습니다. 예를 들어, 특정 열의 데이터가 압축될 수 없다는 것을 미리 알고 있다면(예: 열이 `JPEG` 이미지를 저장하는 경우), 그 열에 대해 `external` 전략을 설정할 수 있습니다. 이를 통해 불필요한 압축 시도를 피할 수 있습니다. 전략 변경은 다음과 같이 할 수 있습니다:

```
=> ALTER TABLE t ALTER COLUMN d SET STORAGE external;
```

질의를 반복하면 다음과 같은 결과가 표시됩니다:

```
 attname ¦ atttypid ¦ storage
---------+----------+----------
 a       ¦ integer  ¦ plain
 b       ¦ numeric  ¦ main
 c       ¦ text     ¦ extended
 d       ¦ json     ¦ external
(4 행s)
```

`TOAST` 테이블은 보통 숨겨져 있으며, 별도의 `pg_toast` 스키마에 위치하고 있습니다. 이는 `TOAST` 테이블이 검색 경로에 포함되지 않기 때문입니다. 임시 테이블의 경우에는 `pg_temp_N`과 같은 방식으로 `pg_toast_temp_N` 스키마가 사용됩니다.

이제 알고리즘의 내부 동작을 살펴보겠습니다. `t` 라는 테이블에는 긴 속성이 포함되어 있을 수 있습니다. 그래서 이에 상응하는 `TOAST` 테이블이 있어야 합니다. 다음은 그 예시입니다:

```
=> SELECT relnamespace::regnamespace, relname
FROM pg_class
WHERE oid = (
SELECT reltoastrelid
FROM pg_class WHERE relname = 't'
);

 relnamespace ¦ relname
--------------+----------------
 pg_toast     ¦ pg_toast_16385
(1 행)

=> \d+ pg_toast.pg_toast_16385
```

```
TOAST table "pg_toast.pg_toast_16385"
     Column ¦ Type    ¦ Storage
------------+---------+---------
  chunk_id  ¦ oid     ¦ plain
 chunk_seq  ¦ integer ¦ plain
 chunk_data ¦ bytea   ¦ plain
Owning table: "public.t"
Indexes:
"pg_toast_16385_index" PRIMARY KEY, btree (chunk_id, chunk_seq)
Access method: heap
```

결국 TOAST로 처리된 행의 청크는 plain 전략을 사용하는 것이 합리적입니다. 두 번째 단계의 TOAST는 존재하지 않습니다.

TOAST 테이블 자체뿐만 아니라 PostgreSQL은 동일한 스키마에 해당하는 인덱스도 생성합니다. 이 인덱스는 항상 TOAST 청크에 접근하는 데 사용됩니다. 인덱스의 이름은 결과에 표시되지만, 다음과 같은 질의를 실행하여 확인할 수도 있습니다:

```
=> SELECT indexrelid::regclass FROM pg_index
WHERE indrelid = (
SELECT oid
FROM pg_class WHERE relname = 'pg_toast_16385'
);
         indexrelid
-----------------------------
 pg_toast.pg_toast_16385_index
(1 행)

=> \d pg_toast.pg_toast_16385_index
Unlogged index "pg_toast.pg_toast_16385_index"
    Column ¦  Type   ¦ Key? ¦ Definition
-----------+---------+------+------------
  chunk_id ¦     oid ¦ yes  ¦ chunk_id
 chunk_seq ¦ integer ¦ yes  ¦ chunk_seq
primary key, btree, for table "pg_toast.pg_toast_16385"
```

그러므로, TOAST 테이블은 테이블이 사용하는 최소 포크 파일 수를 최대 8개로 늘립니다. 주 테이블은 3개, TOAST 테이블은 3개, TOAST 인덱스는 2개의 포크 파일이 필요합니다.

열 c는 extended 전략을 사용하므로 해당 값은 압축될 것입니다:

```
=> UPDATE t SET c = repeat('A',5000);
=> SELECT * FROM pg_toast.pg_toast_16385;

 chunk_id ¦ chunk_seq ¦ chunk_data
```

```
----------+-----------+------------
(0 행s)
```

TOAST 테이블은 비어있습니다. 이는 반복되는 기호가 LZ 알고리즘을 통해 압축되어, 값이 테이블 페이지에 맞게 되기 때문입니다.

이제 이 값을 무작위 기호로 구성해 보겠습니다:

```
=> UPDATE t SET c = (
      SELECT string_agg( chr(trunc(65+random()*26)::integer), '')
      FROM generate_series(1,5000)
  )
  RETURNING left(c,10) || '...' || right(c,10);

       ?column?
-------------------------
YEYNNDTSZR...JPKYUGMLDX
(1 행)
UPDATE 1
```

이 시퀀스는 압축할 수 없으므로 TOAST 테이블에 들어갑니다:

```
=> SELECT chunk_id,
          chunk_seq,
          length(chunk_data),
          left(encode(chunk_data,'escape')::text, 10) || '...' ||
          right(encode(chunk_data,'escape')::text, 10)
FROM pg_toast.pg_toast_16385;

chunk_id  | chunk_seq | length | ?column?
----------+-----------+--------+-------------------------
16390     | 0         | 1996   | YEYNNDTSZR...TXLNDZOXMY
16390     | 1         | 1996   | EWEACUJGZD...GDBWMUWTJY
16390     | 2         | 1008   | GSGDYSWTKF...JPKYUGMLDX
(3 행s)
```

문자들이 청크로 나뉘는 것을 확인할 수 있습니다. 청크의 크기는 TOAST 테이블의 페이지에 4개의 행이 들어갈 수 있도록 설정되어 있습니다. 이 값은 페이지 헤더의 크기에 따라 버전마다 약간씩 다릅니다.

긴 속성에 접근할 때, PostgreSQL은 이를 자동으로 원래의 값으로 복원하여 클라이언트에 반환합니다. 이 과정은 응용 프로그램에게 완전히 투명하게 진행됩니다. 만약 긴 속성이 질의에 포함되지 않는다면, TOAST 테이블은 전혀 읽히지 않습니다. 이것이 제품 설루션에서 별표(*)를 사용하는 것을 피해야 하는 이유 중 하나입니다.

클라이언트가 긴 값의 첫 번째 청크 중 하나를 질의할 경우, PostgreSQL은 필요한 청크만 읽어옵니다. 값이

압축되었더라도 이는 같이 적용됩니다.

그러나 데이터를 압축하고 청크로 나누는 작업은 상당한 자원를 필요로 하며, 원래의 값을 복원하는 것 역시 마찬가지입니다. 그래서 PostgreSQL에 큰 데이터를 저장하는 것은 좋은 방법이 아닙니다. 이는 특히 데이터가 활발하게 사용되며, 트랜잭션 로직(예: 스캔된 회계 문서)이 필요하지 않은 경우에 더욱 그렇습니다. 더 나은 대안은 데이터를 파일 시스템에 저장하고, 데이터베이스에는 해당 파일의 이름만 저장하는 것입니다. 그러나 이 경우, 데이터 일관성을 데이터베이스 시스템이 보장할 수 없습니다.

1.2 프로세스와 메모리

PostgreSQL 서버 인스턴스는 여러 프로세스가 상호작용하며 구성되어 있습니다.

서버가 시작될 때 가장 먼저 실행되는 프로세스는 postgres인데, 이는 전통적으로 postmaster라고 부릅니다. 이 프로세스는 모든 다른 프로세스를 생성하며(유닉스 계열 시스템에서는 포크fork 시스템 호출을 사용하여 생성합니다), 이들을 관리합니다. 만약 어떤 프로세스가 실패하면, postmaster는 해당 프로세스를 다시 시작합니다. 그리고 만약 공유 데이터에 손상의 가능성이 있다면, 전체 서버를 재시작합니다.

> PostgreSQL 에서는 원래부터 프로세스 모델이 단순했기 때문에, 스레드로 전환하는 것에 관한 끊임없는 논의가 있었습니다.
>
> 현재 모델에는 몇몇 단점이 있습니다. 정적static인 공유 메모리 할당으로 인해 버퍼 캐시 등의 구조를 동적으로 크기를 조정하는 것이 불가능합니다. 병렬 알고리즘을 구현하는 것이 어렵고, 그 효율성도 떨어질 수 있습니다. 세션은 프로세스에 한정되어 있습니다. 스레드를 사용하는 것은 좋아 보이지만, 격리, 운영 체제 호환성, 자원 관리 등에 관련된 여러 도전이 있습니다. 스레드를 도입하려면 코드를 철저히 개편하고 몇 년 동안 작업해야 하므로 현재는 보수적인 태도가 우세합니다. 가까운 미래에 이런 변화가 일어날 것으로 생각되지 않습니다.

서버 작업은 백그라운드 프로세스에 의해 지속적으로 수행됩니다. 주요 프로세스들은 다음과 같습니다:

- **startup**: 장애 발생 후 시스템을 복구하는 작업을 담당합니다.
- **autovacuum**: 테이블과 인덱스에서 오래된 데이터를 제거하는 작업을 합니다.
- **wal writer**: WAL 항목을 디스크에 기록하는 역할을 합니다.
- checkpointer: 체크포인트를 실행하는 작업을 담당합니다.
- **writer**: 변경된 페이지를 디스크에 플러시 하는 역할을 합니다.
- stats collector: 인스턴스의 사용 통계를 수집하는 작업을 담당합니다.
- wal sender: WAL 항목을 복제본에 전송하는 역할을 합니다.
- wal receiver: 복제본에서 WAL 항목을 받는 작업을 합니다.

이러한 프로세스들 중 일부는 작업이 완료되면 종료되며, 일부는 계속해서 백그라운드에서 실행되고, 또 일부는 종료할 수 있습니다.

> 각 프로세스는 설정 매개변수에 의해 관리되며, 때때로 수십 개의 매개변수에 의해 조정됩니다. 서버를 전반적으로 설정하려면 내부 동작에 관한 이해가 필요합니다. 그러나 초기 설정에 관한

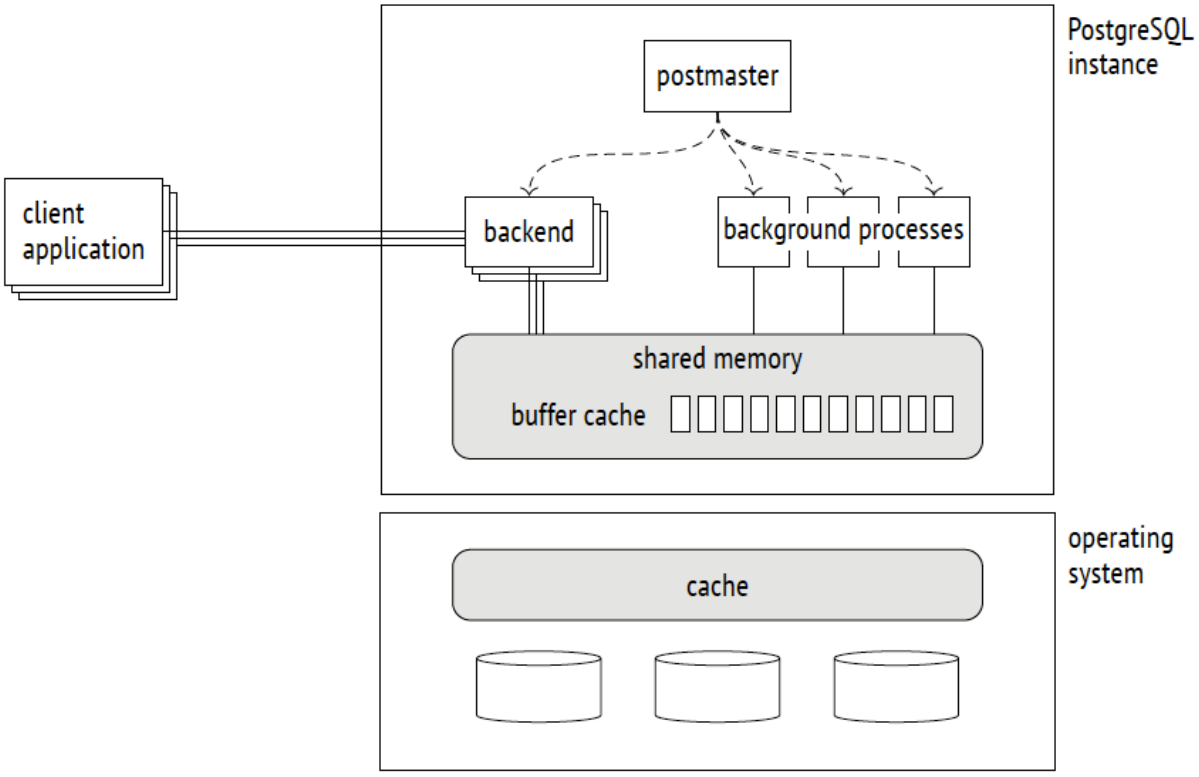
> 일반적인 고려 사항은 초깃값에 더 적절한 값을 선택하는 데 도움이 될 뿐입니다. 이후에는 모니터링 데이터를 기반으로 이러한 설정을 세밀하게 조정해야 합니다.

프로세스 간의 상호작용을 가능케 하기 위해, `postmaster`는 모든 프로세스에서 사용할 수 있는 공유 메모리를 할당합니다.

디스크(특히 `HDD`, 그리고 `SSD`도 마찬가지)는 `RAM`보다 많이 느리기 때문에, `PostgreSQL`은 캐싱을 활용합니다. 공유 `RAM`의 일부는 최근에 읽은 페이지를 위해 예약되어 있어, 이 페이지들이 여러 번 필요할 때 디스크 액세스의 부하를 줄일 수 있습니다. 변경된 데이터는 바로가 아니라 일정 시간 이후에 디스크에 갱신됩니다.

버퍼 캐시는 공유 메모리의 대부분을 차지하고, 서버가 디스크 액세스를 가속화하기 위해 사용하는 다른 버퍼도 포함되어 있습니다.

운영 체제 자체에도 캐시가 있는데, `PostgreSQL`은 운영 체제의 메커니즘을 우회하지 않고 거의 직접 `I/O`를 사용하므로 이중 캐싱이 발생합니다.



정전이나 운영 체제 충돌 등 예기치 않은 상황이 발생하면, `RAM`에 저장된 데이터(버퍼 캐시 포함)는 사라집니다. 디스크에 저장된 파일들의 페이지 기록 시간이 각각 다르므로 데이터 일관성 문제가 생길 수 있습니다. 이를 해결하기 위해 `PostgreSQL`은 `Write-Ahead Log`(`WAL`)를 사용합니다. `WAL`은 `PostgreSQL` 운영 중에 계속 기록되며, 필요할 경우 손실된 작업을 `WAL`을 통해 복구할 수 있습니다.
-- 여기까지

1.3 클라이언트와 클라이언트-서버 프로토콜

포스트마스터 프로세스는 들어오는 연결을 받는 역할도 합니다. 새로운 클라이언트가 접속하면, 포스트마스

터는 별도의 백엔드 프로세스를 만들어 줍니다.[16] 그런 다음 클라이언트는 이 백엔드와 연결을 맺어 세션을 시작합니다. 이 세션은 클라이언트가 연결을 끊거나, 연결이 끊어질 때까지 계속됩니다.

서버는 클라이언트마다 별도의 백엔드를 생성해야 하는데, 이는 많은 클라이언트가 접속을 시도하면 문제가 될 수 있습니다.

- 각 프로세스는 카탈로그 테이블, 준비된 문장, 중간 질의 결과 등을 캐시 하기 위해 RAM이 필요합니다. 따라서 연결이 많아질수록 메모리 사용량도 증가합니다.
- 연결이 짧고 자주 일어나는 경우(클라이언트가 작은 질의를 실행하고 연결을 끊는 경우)에는 연결 설정 비용, 새 프로세스 생성 비용, 불필요한 로컬 캐싱 비용 등이 높아질 수 있습니다.
- 프로세스가 많아질수록 그 목록을 확인하는 데 걸리는 시간이 늘어나며, 이 작업은 매우 자주 수행됩니다. 그 결과, 클라이언트 수가 증가할수록 성능 저하가 발생할 수 있습니다.

이 문제는 연결 풀링을 통해 해결할 수 있습니다. 연결 풀링은 생성되는 백엔드 수를 제한하는 방법입니다. PostgreSQL은 내장된 연결 풀링 기능이 없으므로 애플리케이션 서버에 내장된 풀링 매니저나 외부 도구(`PgBouncer`[17], `Odyssey`[18] 등)를 사용해야 합니다. 이 방법을 사용하면 각 서버 백엔드가 연속적으로 다른 클라이언트의 트랜잭션을 처리할 수 있습니다. 그러나 이 방식은 전체 세션 대신 트랜잭션에만 로컬 자원을 사용하도록 애플리케이션 개발에 제한을 가합니다.

클라이언트와 서버 간의 원활한 상호작용을 위해 동일한 인터페이스 프로토콜을 사용해야 합니다.[19] 대체로 표준 `libpq` 라이브러리를 기반으로 하지만, 사용자가 직접 만든 구현체도 사용할 수 있습니다.

간단히 말하면, 이 프로토콜은 클라이언트가 서버에 연결하고 `SQL` 질의를 실행할 수 있게 합니다.

연결은 특정 데이터베이스와 특정 사용자를 대표하며 설정됩니다. 서버는 데이터베이스 클러스터를 지원하지만, 애플리케이션에서 사용할 데이터베이스마다 별도의 연결을 설정해야 합니다. 이 단계에서 인증 authentication이 이루어집니다(예를 들면 비밀번호를 물어본다든지). 백엔드 프로세스는 사용자의 신원을 확인하고, 해당 사용자가 서버와 지정된 데이터베이스에 연결할 권한이 있는지 검사합니다.

`SQL` 질의는 텍스트 문자열 형태로 백엔드 프로세스에 전달됩니다. 프로세스는 이 텍스트를 분석하고 질의를 최적화한 후 실행하고, 그 결과를 클라이언트에 반환합니다.

---

16 backend/tcop/postgres.c, PostgresMain function

17 pgbouncer.org

18 github.com/yandex/odyssey

19 postgresql.org/docs/14/protocol.html

Part I

격리와 MVCC

2 장 격리Isolation

2.1 일관성Consistency

관계형 데이터베이스의 핵심 기능은 데이터 일관성, 즉 데이터의 정확성을 유지하는 능력입니다.

데이터베이스 수준에서 `NOT NULL`이나 `UNIQUE`와 같은 무결성 제약 조건을 설정할 수 있습니다. 데이터베이스 시스템은 이런 제약 조건이 항상 지켜지도록 보장함으로써 데이터 무결성을 유지합니다.

모든 필요한 제약 조건이 데이터베이스 수준에서 설정될 수 있다면, 데이터 일관성이 보장될 것입니다. 그러나, 일부 조건은 복잡해서 여러 테이블에 걸쳐 적용되거나, 데이터베이스에서 설정할 수 있지만 어떤 이유로 설정되지 않은 예도 있습니다.

따라서, 데이터 일관성은 무결성보다 엄격한 개념이지만, 데이터베이스 시스템은 일관성이 무엇인지 정확히 알 수 없습니다. 만약 애플리케이션에서 일관성은 깨지지만, 무결성은 유지된다면, 데이터베이스 시스템은 이를 확인할 방법이 없습니다. 그래서 데이터 일관성에 관한 기준은 애플리케이션에서 설정해야 하며, 이 기준이 정확하게 작성되어 있어야 합니다.

그런데 애플리케이션이 항상 올바르게 동작한다면, 데이터베이스 시스템은 어떤 역할을 하는 걸까요?

이는 올바른 연산 순서가 때때로 데이터 일관성을 일시적으로 깨뜨릴 수 있기 때문입니다. 이는 사실 매우 정상적인 현상입니다.

계좌 간의 자금 이체를 예로 들어보겠습니다. 일관성 규칙은 '자금 이체는 영향받는 계좌의 총잔액을 변동시켜서는 안 된다'로 정의할 수 있습니다. 이 규칙을 `SQL`에서 무결성 제약 조건으로 설정하기는 어렵습니다(가능하긴 하지만). 그래서 이 규칙을 애플리케이션 수준에서 설정하고, 데이터베이스 시스템에는 알리지 않는다고 가정해 봅시다. 이체 작업은 두 단계로 이루어집니다: 한 계좌에서 돈을 빼는 것과, 그 돈을 다른 계좌에 넣는 것입니다. 첫 번째 단계에서 데이터 일관성이 깨지고, 두 번째 단계에서 복구됩니다.

만약 첫 번째 단계는 성공했지만, 두 번째 단계가 어떤 이유로 실패하면 데이터 일관성이 깨집니다. 이런 상황은 허용되지 않지만, 애플리케이션 수준에서 이를 감지하고 처리하기는 어렵습니다. 하지만, 데이터베이스 시스템이 이 두 단계를 하나의 트랜잭션으로 인식한다면, 이 문제는 데이터베이스 시스템이 해결할 수 있습니다.

그런데 여기서 더 섬세한 문제가 있습니다. 트랜잭션은 개별적으로는 완벽하게 동작하지만, 병렬로 실행될 때는 문제가 발생할 수 있습니다. 이는 서로 다른 트랜잭션의 동작이 섞일 수 있기 때문입니다. 데이터베이스 시스템이 한 트랜잭션의 모든 동작을 끝내고 다음 트랜잭션으로 넘어간다면 이런 문제는 발생하지 않을 것입니다. 하지만 이렇게 차례대로 실행하면 성능이 매우 떨어질 것입니다.

> 실제로 동시에 실행되는 트랜잭션은 멀티코어 프로세서나 디스크 어레이 같은 적절한 하드웨어를 갖춘 시스템에서만 가능합니다. 하지만, 같은 논리는 시간을 공유하며 명령을 순차적으로 실행하는 서버에도 적용됩니다. 이런 상황을 쉽게 이해하기 위해, 때때로 이를 '동시 실행'이라고 부르기도 합니다.

동시에 실행되는 트랜잭션이 잘못 동작하면, 이를 '동시성 이상 anomalies'이라고 부릅니다.

예를 들어, 애플리케이션은 데이터베이스에서 일관된 데이터를 얻기 위해, 아직 확정되지 않은 다른 트랜잭션의 변화를 보지 않아야 합니다. 그렇지 않으면, 일부 트랜잭션이 취소되어 실제로는 존재하지 않는 데이터베이스 상태를 볼 수 있게 됩니다. 이를 '더티 리드dirty read'라고 부릅니다. 더 복잡한 이상 현상들도 있습니다.

트랜잭션을 동시에 실행할 때, 데이터베이스는 그 결과가 순차적 실행의 결과와 같게 되도록 보장해야 합니다. 즉, 트랜잭션들을 서로 격리해 데이터 일관성을 유지해야 합니다.

요약하자면, 트랜잭션은 데이터베이스 상태를 올바르게 변경하는 일련의 작업입니다. 이는 트랜잭션이 완전히 실행되는 동안 다른 트랜잭션에 영향을 받지 않아야 한다는 가정하에 이루어집니다. 이는 원자성과 격리성의 개념을 포함합니다.

하지만 완전한 격리성은 구현하기 어렵고 성능에 부정적인 영향을 미칠 수 있습니다. 대부분의 실제 시스템에서는 약한 격리 수준을 사용하여 일부 이상 현상은 방지하지만, 모든 이상 현상을 방지하지는 않습니다. 그래서 데이터 일관성을 유지하는 것은 애플리케이션에도 일부 의존됩니다. 따라서 시스템이 어떤 격리 수준을 사용하고, 그 수준에서 무엇을 보장하며, 무엇을 보장하지 않는지, 그리고 코드가 해당 환경에서 어떻게 제대로 작동하게 할 수 있는지 알아야 합니다.

2.2 SQL 표준의 격리 수준 및 이상 징후

SQL 표준은 격리 수준을 네 가지로 구분하며[20], 이는 동시에 처리되는 트랜잭션에서 발생할 수 있는 예외 현상들을 기준으로 정의됩니다. 그래서 격리 수준에 관한 이해를 시작할 때는 이런 예외 현상들부터 알아봐야 합니다.

표준은 이론적인 구조이기 때문에 실제 운영 환경과는 다양한 방식으로 다를 수 있습니다. 따라서 여기서 제시한 모든 예시는 대부분 가상의 상황을 나타내므로 이를 인지하고 있어야 합니다. 은행 계좌 트랜잭션에 관한 예시는 이해를 돕기 위한 것이며, 실제 은행 업무와는 무관함을 인정해야 합니다.

또한, 실제 데이터베이스 이론은 표준과는 다르게 발전했습니다. 이는 표준이 도입된 이후에 이론이 개발되었고, 실무가 이미 표준을 앞서 나아갔기 때문입니다.

잃어버린 수정Lost Update

잃어버린 수정 이상 현상이란, 두 트랜잭션이 같은 테이블 행을 읽고, 한 트랜잭션이 행을 수정한 후에, 다른 트랜잭션이 첫 번째 트랜잭션이 만든 변경 사항을 무시하고 같은 행을 수정할 때 발생합니다. 예를 들어, 한 계정의 잔액을 100달러 올리는 두 트랜잭션이 있다고 가정해 보겠습니다. 첫 번째 트랜잭션은 현재 잔액(1,000달러)을 읽고, 두 번째 트랜잭션도 같은 값을 읽습니다. 첫 번째 트랜잭션은 잔액을 증가시켜 1,100달러로 데이터베이스에 기록합니다. 그런데 두 번째 트랜잭션도 마찬가지로 잔액을 증가시켜 1,100달러로 기록합니다. 결과적으로, 고객이 100달러를 잃게 됩니다. 이러한 잃어버린 수정현상은 모든 격리 수준에서 표준으로 금지되어 있습니다.

[20] postgresql.org/docs/14/transaction-iso.html

더티 읽기 및 커밋되지 않은 읽기Dirty Reads and Read Uncommitted

더티 리드 이상 현상은 한 트랜잭션이 다른 트랜잭션의 아직 확정되지 않은 변경 사항을 읽게 되 때 발생합니다. 예를 들어, 첫 번째 트랜잭션은 빈 계정에 100달러를 입금하지만, 이 변경 사항이 아직 확정되지 않은 상태입니다. 그런데 다른 트랜잭션이 이 계정의 상태를 읽어 들입니다. 이 계정은 이미 수정은 되었지만, 확정은 되지 않았습니다. 그 결과, 고객이 돈을 찾을 수 있게 됩니다. 그러나 첫 번째 트랜잭션이 중단되어 변경 사항이 되돌려지면, 계정은 다시 빈 상태가 됩니다. 표준에서는 커밋되지 않은 읽기 수준에서 이러한 더티 리드 현상을 허용하고 있습니다.

반복할 수 없는 읽기와 커밋된 읽기Non-Repeatable Reads and Read Committed

반복할 수 없는 읽기 이상 현상은 한 트랜잭션이 동일한 행을 두 번 읽는 동안, 다른 트랜잭션이 해당 행을 수정하거나 삭제하고 그 변경 사항을 확정할 때 발생합니다. 이에 따라 첫 번째 트랜잭션은 두 번의 읽기 사이에 결과가 달라집니다.

예를 들어, 은행 계좌에서는 잔액이 음수가 되지 않도록 하는 규칙이 있다고 가정해 봅시다. 첫 번째 트랜잭션은 계정 잔액을 100달러 줄이려고 합니다. 현재 잔액(1,000달러)을 확인하고, 이 작업이 가능하다고 판단합니다. 그런데 다른 트랜잭션이 동시에 해당 계정에서 모든 금액을 인출하고 그 변경 사항을 확정합니다. 이 시점에서 첫 번째 트랜잭션이 잔액을 다시 확인하면 0달러를 확인하게 될 것입니다. 그런데 이미 돈을 찾기로 했기 때문에, 이 작업은 초과 인출을 발생시킵니다.

표준에서는 커밋되지 않은 읽기 및 커밋된 읽기 수준에서 반복할 수 없는 읽기 현상을 허용하고 있습니다.

팬텀 읽기 및 반복 읽기Phantom Reads and Repeatable Read

팬텀 읽기 이상 현상은 한 트랜잭션이 특정 조건에 부합하는 행 집합을 반환하는 동일한 질의를 두 번 실행하는 동안, 다른 트랜잭션이 이 조건에 맞는 새로운 행을 추가하고 그 변경 사항을 확정하는 시간 사이에 발생합니다. 그 결과, 첫 번째 트랜잭션은 두 번의 질의에서 다른 행 집합을 반환받게 됩니다.

예를 들어, 한 고객이 세 개 이상의 계정을 가지는 것을 제한하는 규칙이 있다고 가정해봅시다. 첫 번째 트랜잭션은 새로운 계정을 개설하려고 합니다. 그래서 현재 고객이 가진 계정 수를 확인하고, 이 작업이 가능하다고 판단합니다(예를 들어, 현재 두 개의 계정이 있다고 가정합니다). 그런데 이때, 두 번째 트랜잭션이 같은 고객에 대해 새로운 계정을 개설하고 그 변경 사항을 확정합니다. 첫 번째 트랜잭션이 다시 계정 수를 확인하면 세 개의 계정을 확인하게 될 것입니다. 그런데 이미 새 계정 개설 작업이 진행 중이므로, 이 고객은 결국 네 개의 계정을 가지게 됩니다.

표준에서는 커밋되지 않은 읽기, 커밋된 읽기, 반복 읽기의 격리 수준에서 팬텀 리드 현상을 허용하고 있습니다.

이상 없음 및 직렬화 가능No Anomalies and Serializable

표준은 직렬화 수준을 정의하고 있으며, 이 수준에서는 어떠한 이상 현상도 허용되지 않습니다. 이는 잃어버린 수정, 더티 읽기, 반복할 수 없는 읽기, 팬텀 읽기와 같은 알려진 이상 현상을 금지하는 것을 넘어서, 알려지지 않은 이상 현상까지도 금지하게 됩니다. 사실, 표준에서 언급하는 것보다 훨씬 많은 이상 현상이 존재하며, 그중 일부는 아직 알려지지 않았습니다.

직렬화 수준은 모든 종류의 이상 현상을 방지해야 합니다. 이는 응용 프로그램 개발자가 격리 수준에 대해 고려할 필요가 없음을 의미합니다. 트랜잭션이 개별적으로 실행될 때와 동일한 올바른 연산 순서로 실행된다면, 동시에 실행되는 트랜잭션은 데이터 일관성을 깨뜨리지 않습니다.

이 개념을 설명하기 위해, 표준에서 제공하는 잘 알려진 테이블을 사용하겠습니다. 여기에는 명확한 설명을 위해 마지막 열이 추가되었습니다:

	lost update	dirty read	non-repeatable read	phantom read	other anomalies
Read Uncommitted	–	yes	yes	yes	yes
Read Committed	–	–	yes	yes	yes
Repeatable Read	–	–	–	yes	yes
Serializable	–	–	–	–	–

왜 이런 이상 현상이 발생했을까요?

표준에서 왜 특정 이상 현상만을 언급하며,그 중에서도 왜 바로 이런 이상 현상들을 선택했는지는 명확히 알 수 없습니다. 하지만, 초기 표준이 도입될 때 이론이 실무에 비해 많이 뒤떨어져 있었던 점을 고려하면, 일부 다른 이상 현상들이 빠졌을 수 있습니다.

또한, 격리 수준이 잠금을 기반으로 해야 한다는 가정이 있었습니다. 널리 쓰이는 이차 잠금two-phase locking 프로토콜(2PL)은 트랜잭션이 실행되는 동안 영향을 받는 행을 잠그고, 트랜잭션이 끝나면 잠금을 해제해야 합니다. 간단히 말해서, 트랜잭션이 획득하는 잠금이 많아질수록 다른 트랜잭션으로부터 더욱 격리되지만, 이런 경우에는 시스템의 성능이 저하되고, 트랜잭션들이 동시에 실행되는 대신 같은 행에 접근하기 위해 대기 상태에 놓이게 됩니다.

저는 표준 격리 수준 간의 차이가 주로 각 수준의 구현에 필요한 잠금의 개수에 따라 결정된다고 생각합니다.

만약 수정할 행들이 쓰기 작업에 대해서만 잠금이 걸려 있고, 읽기 작업에 대해서는 잠금이 없다면, 이는 커밋되지 않은 읽기 격리 수준을 의미합니다. 이 수준에서는 아직 커밋되지 않은 데이터를 읽을 수 있게 됩니다.

수정할 행들이 읽기와 쓰기 모두에 대해 잠금이 걸려 있다면, 이는 커밋된 읽기 수준을 의미합니다. 이 수준에서는 아직 커밋되지 않은 데이터를 읽는 것은 금지되지만, 같은 질의를 여러 번 실행하면 다른 값을 반환받을 수 있습니다(반복할 수 없는 읽기).

모든 작업에 대해 읽을 행과 수정할 행이 잠겨 있다면, 이는 반복 읽기 수준을 의미합니다. 이 경우, 반복해서 실행하는 질의는 항상 같은 결과를 반환합니다.

하지만 직렬화 수준에서는 문제가 생깁니다. 아직 존재하지 않는 행에 대해 잠금을 걸 수는 없습니다. 이에 따라 팬텀 읽기 현상이 발생할 여지가 생깁니다. 트랜잭션은 이전 질의의 조건에 부합하는 새로운 행을 추가할 수 있고, 이 새로운 행은 다음 질의의 결과에 포함될 수 있습니다.

그러므로 일반적인 잠금만으로는 완전한 격리를 보장할 수 없습니다. 이를 해결하기 위해서는 행이 아니라 조건(술어부predicates)에 잠금을 걸어야 합니다. 이런 술어부 잠금은 1976년 시스템 R이 개발될 때 처음 도입

되었지만, 실제로 적용할 수 있는 경우는 두 개의 술어부가 충돌할 수 있는지가 명확한 간단한 조건에 한정되었습니다. 제가 알기로는 술어부 잠금이 그 원래 의도대로 어떤 시스템에도 구현된 적은 없습니다.

2.3 PostgreSQL의 격리 수준

시간이 흐르면서 트랜잭션 관리에 관한 잠금 기반 프로토콜은 스냅숏 격리Snapshot Isolation(SI) 프로토콜로 대체되었습니다. 이 방식의 핵심 개념은 각 트랜잭션이 특정 시점의 데이터 스냅숏에 접근하게 된다는 것입니다. 이 스냅숏에는 스냅숏이 찍힌 시점 이전에 커밋된 모든 변경 내용이 포함됩니다.

스냅숏 격리는 필요한 잠금의 수를 최소화합니다. 실제로, 행이 잠기는 경우는 동시에 수정을 시도할 때뿐입니다. 그 외의 모든 경우에는 작업이 동시에 실행될 수 있습니다. 쓰기 작업이 읽기를 잠글 필요가 없으며, 읽기 작업이 다른 것을 잠글 필요도 없습니다.

PostgreSQL은 스냅숏 격리 프로토콜의 다중 버전 형태를 사용합니다. 다중 버전 동시성 제어Multiversion concurrency control를 통해 데이터베이스 시스템이 한 행의 여러 버전을 관리할 수 있게 되므로, PostgreSQL은 오래된 데이터를 읽으려는 트랜잭션을 중단시키는 대신 적절한 버전을 스냅숏에 포함할 수 있습니다.

PostgreSQL의 스냅숏 기반 격리는 표준에서 정한 요구사항과는 다릅니다. 실제로, 약간 더 엄격한 편입니다. 더티 읽기는 원칙적으로 허용되지 않습니다. 커밋되지 않은 읽기 수준을 지정할 수는 있지만, 그 동작은 커밋된 읽기와 같으므로 이 수준은 별도로 언급하지 않겠습니다. 반복 읽기는 반복할 수 없는 읽기와 팬텀 읽기를 허용하지 않습니다(전체적인 격리는 보장되지 않습니다). 그러나 커믹된 읽기 수준에서는 변경 사항을 잃을 위험이 있습니다.

	lost updates	dirty reads	non-repeatable reads	phantom reads	other anomalies
Read Committed	yes	–	yes	yes	yes
Repeatable Read	–	–	–	–	yes
Serializable	–	–	–	–	–

각 격리 수준의 내부 메커니즘을 살펴보기 전에, 사용자 관점에서의 각 격리 수준을 먼저 살펴보겠습니다.

이를 위해서는 계정 테이블을 만들어야 합니다. 엘리스Alice와 밥Bob이라는 두 사람이 각각 100달러를 가지고 있고, 밥이 두 개의 계정을 가지고 있다고 가정해봅시다:

```
=> CREATE TABLE accounts(
id integer PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
client text,
amount numeric
);
=> INSERT INTO accounts VALUES
(1, 'alice', 1000.00), (2, 'bob', 100.00), (3, 'bob', 900.00);
```

커밋된 읽기

더티 읽기는 허용되지 않습니다. 더티 데이터를 읽는 것이 허용되지 않는지 직접 확인해볼 수 있습니다. 그러기 위해 트랜잭션을 시작해봅시다. 기본적으로 커밋된 읽기[21]이라는 격리 수준을 사용하도록 하겠습니다:

```
=> BEGIN;
=> SHOW transaction_isolation;
transaction_isolation
———————————————————————
read committed
(1 행)
```

더 정확하게 말하자면, 기본 수준은 다음과 같은 세팅값에 의해 설정되며, 필요에 따라 이 값을 변경할 수 있습니다:

```
=> SHOW default_transaction_isolation;
default_transaction_isolation
———————————————————————————————
read committed
(1 행)
```

진행 중인 트랜잭션에서는 고객 계정에서 일부 돈을 빼내지만 아직 이런 변경 내용을 확정(커밋)하지 않았습니다. 그러나 항상 자신이 한 변경 내용은 볼 수 있습니다:

```
=> UPDATE accounts SET amount = amount - 200 WHERE id = 1;
=> SELECT * FROM accounts WHERE client = 'alice';
id  ¦ client  ¦ amount
————+————————+————————
1   ¦ alice   ¦ 800.00
(1 행)
```

두 번째 세션에서는 커밋된 읽기 수준으로 또 다른 트랜잭션을 시작하겠습니다:

```
=> BEGIN;
=> SELECT * FROM accounts WHERE client = 'alice';
id  ¦ client  ¦ amount
————+————————+————————
1   ¦ alice   ¦ 1000.00
(1 행)
```

[21] postgresql.org/docs/14/transaction-iso.html#XACT-READ-COMMITTED

우리의 예상대로, 두 번째 트랜잭션에서는 아직 확정되지 않은 변경 내용을 볼 수 없습니다. 더티 읽기는 허용되지 않습니다.

반복할 수 없는 읽기에 관해 살펴보겠습니다. 이제 첫 번째 트랜잭션에서 변경 내용을 확정(커밋)해보도록 하겠습니다. 그 후 두 번째 트랜잭션에서는 같은 질의를 다시 실행해보겠습니다:

```
=> COMMIT;


 => SELECT * FROM accounts WHERE client = 'alice';
 id  ¦ client  ¦ amount
----+---------+---------
 1   ¦ alice   ¦ 800.00
 (1 행)
 => COMMIT;
```

이 질의는 데이터의 최신 버전을 받게 되는데, 이는 커밋된 읽기 수준에서 허용되는 반복할 수 없는 읽기 현상을 정확히 보여줍니다.

실용적인 통찰력: 트랜잭션 내에서 이전 작업에 의해 읽힌 데이터를 기반으로 결정을 내릴 수 없습니다. 왜냐하면 그사이에 모든 것이 변할 수 있기 때문입니다. 다음은 응용 프로그램 코드에서 자주 발생하는 변형으로, 이는 고전적인 안티 패턴으로 간주할 수 있는 예입니다:

```
IF (SELECT amount FROM accounts WHERE id = 1) >= 1000 THEN
   UPDATE accounts SET amount = amount - 1000 WHERE id = 1;
END IF;
```

조건 확인과 데이터의 수정 사이의 시간 동안 다른 트랜잭션들이 계정의 상태를 자유롭게 변경할 수 있기 때문에 이런 조건 확인은 전혀 의미가 없습니다. 이해를 돕기 위해, 다른 트랜잭션의 임의의 작업들이 현재 트랜잭션의 작업들 사이에 끼어들어wedged 있다고 생각해 볼 수 있습니다. 예를 들면, 다음과 같이 될 수 있습니다:

```
IF (SELECT amount FROM accounts WHERE id = 1) >= 1000 THEN

    UPDATE accounts SET amount = amount - 200 WHERE id = 1;
    COMMIT;

  UPDATE accounts SET amount = amount - 1000 WHERE id = 1;
END IF;
```

연산자를 재배열하면서 문제가 발생하는 코드는 올바르지 않다는 것을 명심하세요. 절대로 자신이 이런 문제에 빠지지 않을 것으로 생각하지 마세요. 가능한 모든 문제는 반드시 발생할 수 있습니다. 이런 오류들은 재

현하기가 매우 어려워, 수정하는 것이 큰 도전이 될 수 있습니다.

- 이 코드를 수정하는 방법에는 다음과 같은 여러 가지 방법이 있습니다:

  절차적인 코드를 선언적인 코드로 바꿔보세요. 예를 들어, `IF` 문을 `CHECK` 제약 조건으로 바꿀 수 있습니다:

  ```
  ALTER TABLE accounts
     ADD CHECK amount >= 0;
  ```

  이렇게 하면 코드에서 별도의 검사가 필요 없게 됩니다. 명령을 실행하고 무결성 제약 조건이 위반될 때 발생하는 예외만 처리하면 됩니다.

- 단일 `SQL` 연산자를 사용해 보세요.

  다른 트랜잭션의 연산 사이에서 트랜잭션이 커밋되면 데이터 일관성이 손상될 수 있습니다. 단일 연산자를 사용하면 이런 문제를 방지할 수 있습니다.

  PostgreSQL은 복잡한 작업을 단일 문장으로 처리할 수 있는 기능을 제공합니다. 특히, `INSERT`, `UPDATE`, `DELETE`와 `ON CONFLICT` 연산자를 사용할 수 있는 공통 테이블 표현식(`CTE`)을 제공합니다. `ON CONFLICT` 연산자는 '행이 없으면 삽입, 있으면 수정'이라는 로직을 구현합니다.

- 명시적으로 잠금을 설정하세요.

  마지막으로, 필요한 모든 행에 대해 직접 독점적인 잠금을 설정하는 방법이 있습니다(`SELECT FOR UPDATE`). 심지어 전체 테이블에 대해서도 가능합니다(`LOCK TABLE`). 이 방법은 항상 작동하지만 `MVCC`의 모든 장점을 잃게 됩니다. 동시에 실행할 수 있는 일부 작업이 차례대로 실행될 것입니다.

읽기 왜곡 Read skew. 하지만 모든 상황이 이렇게 단순하지는 않습니다. PostgreSQL 구현은 표준으로 규제되지 않은 다른 알려지지 않은 이상 현상을 허용합니다. 예를 들어, 첫 번째 트랜잭션이 밥의 두 계좌 간에 돈을 이체한다고 가정해봅시다:

첫 번째 트랜잭션을 시작하여 돈을 이체합니다:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount - 100 WHERE id = 2;
```

동시에, 두 번째 트랜잭션이 밥의 계좌를 검사하여 총 잔액을 계산합니다. 이때 첫 번째 계좌의 잔액을 확인합니다(이전 상태를 보게 됩니다):

```
=> BEGIN;
=> SELECT amount FROM accounts WHERE id = 2;
amount
```

```
--------
100.00
(1 행)
```

이때 첫 번째 트랜잭션이 성공적으로 완료됩니다:

```
=> UPDATE accounts SET amount = amount + 100 WHERE id = 3;
=> COMMIT;
```

그리고 두 번째 트랜잭션은 두 번째 계좌의 상태를 확인합니다(이미 수정된 값을 볼 수 있습니다):

```
=> SELECT amount FROM accounts WHERE id = 3;
amount
-----------
1000.00
(1 행)
=> COMMIT;
```

이렇게 되면, 두 번째 트랜잭션은 잘못된 데이터를 읽어 $1,000이라는 잘못된 결과를 얻게 됩니다. 이런 현상을 읽기 왜곡이라고 부릅니다.

읽기 왜곡 현상을 방지하는 방법은 무엇일까요? 커밋된 읽기 수준에서는 단일 연산자를 사용하는 것이 효과적입니다. 예를 들면 다음과 같이 질의를 작성할 수 있습니다:

```
SELECT sum(amount) FROM accounts WHERE client = 'bob';
```

지금까지 말했듯이, 데이터의 가시성은 연산자 사이에서만 바뀔 수 있다고 했습니다. 그런데 이게 정말일까요? 만약 질의가 오래 걸린다면 어떨까요? 이런 상황에서는 다른 상태의 데이터를 볼 수 있을까요?

이를 확인해 보기 위해, `pg_sleep` 함수를 호출하여 연산자에 지연을 추가해 보겠습니다. 첫 번째 행은 바로 읽히지만, 두 번째 행은 2초 동안 기다려야 합니다:

```
=> SELECT amount, pg_sleep(2) -- two seconds
FROM accounts WHERE client = 'bob';
```

이 명령어가 실행되는 동안, 다른 트랜잭션을 시작하여 돈을 다시 이체해 보겠습니다:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount + 100 WHERE id = 2;
=> UPDATE accounts SET amount = amount - 100 WHERE id = 3;
=> COMMIT;
```

그 결과, 연산자는 실행 시작 시점의 모든 데이터 상태를 보게 됩니다:

```
amount        ¦ pg_sleep
──────────────+────────────
0.00          ¦
1000.00       ¦
(2 행s)
```

하지만 모든 상황이 이렇게 단순하지는 않습니다. 질의에 `VOLATILE`로 선언된 함수가 포함되어 있고, 이 함수가 다른 질의를 실행한다면, 중첩된 질의에서 보는 데이터는 메인 질의의 결과와 일치하지 않을 수 있습니다.

다음과 같은 함수를 사용해 Bob의 계좌 잔액을 확인해봅시다:

```
=> CREATE FUNCTION get_amount(id integer) RETURNS numeric
AS $$
SELECT amount FROM accounts a WHERE a.id = get_amount.id;
$$ VOLATILE LANGUAGE sql;

=> SELECT get_amount(id), pg_sleep(2)
FROM accounts WHERE client = 'bob';
```

이 때, 지연된 질의가 실행되는 동안에 다시 돈을 계좌 간에 이체해보겠습니다:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount + 100 WHERE id = 2;
=> UPDATE accounts SET amount = amount - 100 WHERE id = 3;
=> COMMIT;
```

이런 경우에는 일관성이 없는 데이터를 얻게 됩니다. 즉, $100이 손실되었습니다:

```
get_amount        ¦ pg_sleep
──────────────────+────────────
100.00            ¦
800.00            ¦
(2 행s)
```

여기서 주목해야 할 것은 이런 현상이 커밋된 읽기 격리 수준에서만 발생하며, 함수가 `VOLATILE`로 선언되었을 때만 가능하다는 점입니다. 문제는 PostgreSQL이 기본적으로 이 격리 수준과 변동성 범주를 사용한다는 것입니다. 그래서 이런 함정이 굉장히 교묘하게 설정되어 있다는 사실을 인지해야 합니다.

잃어버린 수정 대신 읽기 왜곡 현상이 일어납니다. 읽기 왜곡 현상은 수정작업 중에도 단일 작업 내에서 예상치 못한 방식으로 발생할 수 있습니다. 두 개의 트랜잭션이 동일한 행을 수정하려고 시도할 때 어떤 일이 일어나는지 살펴봅시다. 현재 밥은 두 개의 계좌에 총 $1,000을 가지고 있습니다:

```
=> SELECT * FROM accounts WHERE client = 'bob';
id    ¦ client     ¦ amount
————+————————+————————
2     ¦ bob        ¦ 200.00
3     ¦ bob        ¦ 800.00
(2 행s)
```

밥의 잔액을 줄이는 트랜잭션을 시작해 봅시다:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount - 100 WHERE id = 3;
```

동시에 다른 트랜잭션은 총잔액이 $1,000 이상인 모든 고객 계좌에 관해 이자를 계산하려 합니다:

```
=> UPDATE accounts SET amount = amount * 1.01
WHERE client IN (
SELECT client
FROM accounts
GROUP BY client
HAVING sum(amount) >= 1000
);
```

수정 작업은 크게 두 단계로 이루어집니다. 먼저, 제공된 조건에 따라 수정할 행이 선택됩니다. 첫 번째 트랜잭션이 아직 커밋되지 않았기 때문에 두 번째 트랜잭션은 그 결과를 볼 수 없어 이자 계산에 영향을 주지 않습니다. 그래서 밥의 계좌는 조건을 만족하며, 수정 작업이 완료되면 잔액이 $10 증가해야 합니다.

두 번째 단계에서는 선택된 각 행이 수정됩니다. 두 번째 트랜잭션은 id = 3인 행이 잠겨 있으므로 기다려야 합니다. 첫 번째 트랜잭션이 해당 행을 수정하고 있기 때문입니다.

첫 번째 트랜잭션이 변경 사항을 커밋하면, 다음과 같은 결과를 얻습니다:

```
=> COMMIT;
=> SELECT * FROM accounts WHERE client = 'bob';
id    ¦ client     ¦ amount
————+————————+————————————
2     ¦ bob        ¦ 202.0000
3     ¦ bob.       ¦ 707.0000
(2 행s)
```

수정 작업은 첫 번째 트랜잭션이 수행한 변경 사항을 보지 못해야 하지만, 커밋된 변경 사항을 잃어버리면 안 됩니다.

잠금이 해제되면 수정 작업은 수정할 행을 다시 읽습니다(하지만 이 행만!). 그 결과, 밥은 총 $900에 기반한 $9의 이자를 받게 됩니다. 그런데 만약 그의 잔액이 $900이라면, 처음부터 그의 계좌는 질의 결과에 포함되지 않아야 했습니다.

결국, 트랜잭션은 잘못된 데이터를 반환했습니다. 서로 다른 스냅숏에서 서로 다른 행이 읽혔습니다. 이는 잃어버린 수정 대신에 발생한 읽기 왜곡 현상입니다.

잃어버린 수정 Lost updates **현상**. 그러나 잠긴 행을 다시 읽는 방법은, 데이터가 다른 `SQL` 작업에 의해 변경될 경우 잃어버린 수정을 방지하는 데에는 도움이 되지 않습니다.

이미 본 예시가 있습니다. 애플리케이션은 엘리스의 계좌 잔액을 읽고 (데이터베이스 외부에서) 기록합니다:

```
=> BEGIN;
=> SELECT amount FROM accounts WHERE id = 1;
amount
─────────
800.00
(1 행)
```

그와 동시에, 다른 트랜잭션도 같은 작업을 수행합니다:

```
=> BEGIN;
=> SELECT amount FROM accounts WHERE id = 1;
amount
─────────
800.00
(1 행)
```

첫 번째 트랜잭션은 이전에 기록한 값을 $100 상승시키고 이 변경 사항을 커밋합니다:

```
=> UPDATE accounts SET amount = 800.00 + 100 WHERE id = 1
RETURNING amount;
amount
─────────
900.00
(1 행)

UPDATE 1
=> COMMIT;
```

두 번째 트랜잭션도 같은 작업을 수행합니다:

```
=> UPDATE accounts SET amount = 800.00 + 100 WHERE id = 1
RETURNING amount;
amount
————————
900.00
(1 행)
UPDATE 1

=> COMMIT;
```

불행하게도, 엘리스는 $100을 잃어버렸습니다. 데이터베이스 시스템은 기록된 $800의 값이 어떤 방식으로든 `accounts.amount`와 연결되어 있다는 것을 인지하지 못하므로, 잃어버린 수정 현상을 방지할 수 없습니다. 커밋된 읽기 격리 수준에서 이 코드는 잘못된 것입니다.

반복 읽기Repeatable Read

반복 읽기와 유령 읽기phantom reads. 반복 읽기[22] 격리 수준은, 이름에서 알 수 있듯이, 반복 읽기를 보장해야 합니다. 이제 이를 확인하고, 유령 읽기도 발생하지 않는지 확인해봅시다. 이를 위해 밥의 계좌를 이전 상태로 되돌리고, 찰리를 위한 새 계좌를 생성하는 트랜잭션을 시작해보겠습니다:

```
=> BEGIN;
=> UPDATE accounts SET amount = 200.00 WHERE id = 2;
=> UPDATE accounts SET amount = 800.00 WHERE id = 3;
=> INSERT INTO accounts VALUES
(4, 'charlie', 100.00);

=> SELECT * FROM accounts ORDER BY id;
id  ¦ client   ¦ amount
————+————————+————————
1   ¦ alice    ¦ 900.00
2   ¦ bob      ¦ 200.00
3   ¦ bob      ¦ 800.00
4   ¦ charlie  ¦ 100.00
(4 행s)
```

두 번째 세션에서는 반복 읽기 수준의 트랜잭션을 시작해보겠습니다. 이때 `BEGIN` 명령어에서 격리 수준을 명시적으로 지정하였습니다 (첫 번째 트랜잭션의 수준은 중요하지 않습니다):

[22] postgresql.org/docs/14/transaction-iso.html#XACT-REPEATABLE-READ

```
  => BEGIN ISOLATION LEVEL REPEATABLE READ;
  => SELECT * FROM accounts ORDER BY id;
  id  ¦ client   ¦ amount
  ————+————————+————————————
  1   ¦ alice    ¦ 900.00
  2   ¦ bob      ¦ 202.0000
  3   ¦ bob      ¦ 707.0000
  (3 행s)
```

이제 첫 번째 트랜잭션에서 변경 사항을 커밋하고, 두 번째 트랜잭션에서 동일한 질의를 다시 실행해봅니다:

```
=> COMMIT;

  => SELECT * FROM accounts ORDER BY id;
  id  ¦ client   ¦ amount
  ————+————————+————————————
  1   ¦ alice.   ¦ 900.00
  2   ¦ bob      ¦ 202.0000
  3   ¦ bob      ¦ 707.0000
  (3 행s)
  => COMMIT;
```

두 번째 트랜잭션은 여전히 이전과 동일한 데이터를 확인할 수 있습니다. 새로운 행이나 행 수정도 보이지 않습니다. 이 격리 수준에서는, 연산자 간에 변화가 일어날 것을 걱정할 필요가 없습니다.

잃어버린 수정 대신 직렬화 Serialization **실패**. 앞서 확인했듯이, 커밋된 읽기 수준에서 두 트랜잭션이 같은 행을 수정하면, 읽기 왜곡 현상이 발생할 가능성이 있습니다. 대기 중인 트랜잭션은 잠겨 있는 행을 다시 읽어야 하므로, 다른 행들과 비교해 해당 행의 상태를 다른 시점에서 볼 수 있습니다.

이런 현상은 반복 읽기 격리 수준에서는 허용되지 않으며, 만약 발생하면 트랜잭션은 직렬화 실패serialization failure로 인해 중단될 수밖에 없습니다. 이제 이런 상황을 이자 발생 예를 통해 확인해 봅시다:

```
=> SELECT * FROM accounts WHERE client = 'bob';
   id ¦ client   ¦ amount
————+————————+————————
    2 ¦ bob      ¦ 200.00
    3 ¦ bob      ¦ 800.00
(2 행s)
=> BEGIN;
=> UPDATE accounts SET amount = amount - 100.00 WHERE id = 3;

  => BEGIN ISOLATION LEVEL REPEATABLE READ;
  => UPDATE accounts SET amount = amount * 1.01
```

```
  WHERE client IN (
  SELECT client
  FROM accounts
  GROUP BY client
  HAVING sum(amount) >= 1000
  );

=> COMMIT;

  ERROR: could not serialize access due to concurrent update
  => ROLLBACK;
```

데이터는 여전히 일관성을 유지합니다:

```
=> SELECT * FROM accounts WHERE client = 'bob';
id  ¦ client ¦ amount
----+--------+--------
2   ¦ bob    ¦ 200.00
3   ¦ bob    ¦ 700.00
(2 행s)
```

다른 열에 영향을 주는 동시 행 수정이라도 동일한 오류가 발생합니다.

또한, 이전에 저장된 값에 기반하여 잔액을 수정하려는 시도도 동일한 오류를 발생시킵니다:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT amount FROM accounts WHERE id = 1;
amount
---------
900.00
(1 행)

  => BEGIN ISOLATION LEVEL REPEATABLE READ;
  => SELECT amount FROM accounts WHERE id = 1;
  amount
 ---------
  900.00
  (1 행)

=> UPDATE accounts SET amount = 900.00 + 100.00 WHERE id = 1
RETURNING amount;
amount
----------
```

```
1000.00
(1 행)
UPDATE 1
=> COMMIT;

  => UPDATE accounts SET amount = 900.00 + 100.00 WHERE id = 1
     RETURNING amount;
     ERROR: could not serialize access due to concurrent update
  => ROLLBACK;
```

실용적인 통찰력으로, 응용 프로그램이 쓰기 트랜잭션에 반목 읽기 격리 수준을 사용하는 경우, 직렬화 실패로 인해 완료된 트랜잭션을 재시도할 준비가 되어야 합니다. 읽기 전용 트랜잭션의 경우 이런 결과는 발생하지 않습니다.

쓰기 왜곡이라는 이상 현상에 관해 설명하겠습니다. 앞서 확인했듯이, PostgreSQL의 반복 읽기 격리 수준은 표준에서 설명한 모든 이상 현상을 방지하는 데 있어서 효과적입니다. 그러나 모든 가능한 이상 현상을 방지하는 것은 아닙니다. 어떤 혹은 얼마나 많은 이상 현상이 존재하는지는 아무도 확실하게 알 수 없습니다. 하지만 한 가지 확실한 사실은 스냅숏 격리가 어떤 다른 이상 현상이 발생하더라도 두 가지 이상 현상만은 방지하지 못한다는 것입니다.

첫 번째 이상 현상은 쓰기 왜곡입니다. 다음과 같은 일관성 규칙을 정의해봅시다: 고객의 계좌 중 일부는 잔액이 음수일 수 있지만, 전체 잔액은 항상 음수가 아니어야 합니다. 첫 번째 트랜잭션은 밥의 계좌의 총잔액을 조회합니다:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT sum(amount) FROM accounts WHERE client = 'bob';
sum
---------
900.00
(1 행)
```

두 번째 트랜잭션도 동일한 합계를 조회합니다:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT sum(amount) FROM accounts WHERE client = 'bob';
sum
---------
900.00
(1 행)
```

첫 번째 트랜잭션은 하나의 계좌에서 $600.00을 출금 가능하다고 판단합니다:

```
=> UPDATE accounts SET amount = amount - 600.00 WHERE id = 2;
```

두 번째 트랜잭션도 같은 결론에 도달하여 다른 계좌에서 출금합니다:

```
  => UPDATE accounts SET amount = amount - 600.00 WHERE id = 3;
  => COMMIT;

=> COMMIT;
=> SELECT * FROM accounts WHERE client = 'bob';
id  ¦ client  ¦ amount
----+---------+----------
2   ¦ bob     ¦ -400.00
3   ¦ bob     ¦ 100.00
(2 행s)
```

밥의 총 잔액은 이제 음수가 되었습니다. 이는 각 트랜잭션이 개별적으로 실행되었다면 올바른 결과였을 것입니다.

읽기 전용 트랜잭션 이상 현상. 읽기 전용 트랜잭션 이상은 반복 읽기 격리 수준에서 나타날 수 있는 두 번째이자 마지막 이상 현상입니다. 이 현상을 확인하려면 총 세 개의 트랜잭션을 수행해야 합니다. 이 중 두 개는 데이터를 수정하고, 마지막 트랜잭션은 읽기 전용입니다.

먼저, 밥의 계좌 잔액을 복원해 봅시다:

```
=> UPDATE accounts SET amount = 900.00 WHERE id = 2;
=> SELECT * FROM accounts WHERE client = 'bob';
 id ¦ client  ¦ amount
----+---------+---------
  3 ¦ bob     ¦ 100.00
  2 ¦ bob     ¦ 900.00
 (2 행s)
```

첫 번째 트랜잭션은 밥의 총 잔액에 이자를 계산하여 그의 한 계좌에 더합니다:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ; -- 1
=> UPDATE accounts SET amount = amount + (
SELECT sum(amount) FROM accounts WHERE client = 'bob'
) * 0.01
WHERE id = 2;
```

그 다음, 두 번째 트랜잭션은 밥의 다른 계좌에서 일정 금액을 인출하고 그 변경 사항을 반영합니다:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ; -- 2
```

```
=> UPDATE accounts SET amount = amount - 100.00 WHERE id = 3;
=> COMMIT;
```

이 시점에서 첫 번째 트랜잭션이 커밋되면, 이상 현상은 발생하지 않습니다. 첫 번째 트랜잭션이 두 번째 트랜잭션보다 먼저 커밋되었기 때문입니다(하지만 반대의 경우는 성립하지 않습니다 - 첫 번째 트랜잭션은 두 번째 트랜잭션에 의한 수정가 이루어지기 전에 id = 3인 계좌의 상태를 확인했습니다).

그런데 이 시점에서 첫 번째와 두 번째 트랜잭션에 영향을 받지 않는 계좌를 확인하려는 읽기 전용 트랜잭션을 시작한다고 가정해 봅시다:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ; -- 3
=> SELECT * FROM accounts WHERE client = 'alice';
   id ¦ client     ¦ amount
————+————————+——————————
    1 ¦ alice      ¦ 1000.00
(1 행)
```

그리고 이제 첫 번째 트랜잭션이 커밋됩니다:

```
=> COMMIT;
```

이제 세 번째 트랜잭션은 어떤 상태를 볼까요? 시작된 후, 이미 커밋된 두 번째 트랜잭션의 변경 사항은 볼 수 있겠지만, 첫 번째 트랜잭션의 변경 사항은 아직 커밋되지 않았으므로 보지 못할 것입니다. 하지만 앞서 확인했듯이, 두 번째 트랜잭션은 첫 번째 트랜잭션이 시작된 후에 시작되었어야 합니다. 따라서, 세 번째 트랜잭션이 볼 수 있는 어떤 상태도 일관성이 없을 것입니다. 이것이 바로 읽기 전용 트랜잭션 이상입니다:

```
=> SELECT * FROM accounts WHERE client = 'bob';
   id ¦ client.    ¦ amount
————+————————+————————
    2 ¦ bob        ¦ 900.00
    3 ¦ bob        ¦ 0.00
(2 행s)
=> COMMIT;
```

직렬화Serializable

직렬화[23] 격리 수준은 데이터베이스에서 가장 엄격한 격리 수준입니다. 이 수준에서는 모든 가능한 이상 현상을 방지하며, 이는 사실상 스냅숏 격리 위에 구축되어 있습니다. 직렬화 격리 수준에서는 반복 불가능한 읽기, 더티 읽기, 유령 읽기와 같은 이상 현상이 발생하지 않습니다. 이들 현상은 데이터 일관성을 해칠 수 있으므로 반복할 수 있는 읽기 격리 수준에서도 방지됩니다. 직렬화 격리 수준에서는 또한 쓰기 왜곡과 읽기 전용 트랜잭션 이상이라는 두 가지 추가적인 이상 현상을 감지하고 방지합니다. 이런 이상 현상이 발생하면, 시스

[23] postgresql.org/docs/14/transaction-iso.html#XACT-SERIALIZABLE

템은 트랜잭션을 중단시키고 직렬화 실패라는 오류를 발생시킵니다. 이는 다른 트랜잭션이 동시에 같은 데이터에 접근하려고 할 때 발생하며, 이를 통해 데이터의 일관성을 유지하게 됩니다. 따라서 직렬화 격리 수준은 데이터베이스에서 가장 안정적인 격리 수준이지만, 이에 따라 동시성이 감소하고 성능이 저하될 수 있습니다. 따라서 애플리케이션의 요구 사항과 성능을 고려하여 적절한 격리 수준을 선택해야 합니다.

이상 현상 없음. 쓰기 왜곡 예제는 결국 직렬화 실패로 끝나게 됩니다:

```
=> BEGIN ISOLATION LEVEL SERIALIZABLE;
=> SELECT sum(amount) FROM accounts WHERE client = 'bob';
sum
----------
910.0000
(1 행)

  => BEGIN ISOLATION LEVEL SERIALIZABLE;
  => SELECT sum(amount) FROM accounts WHERE client = 'bob';
  sum
  ----------
  910.0000
  (1 행)

=> UPDATE accounts SET amount = amount - 600.00 WHERE id = 2;

  => UPDATE accounts SET amount = amount - 600.00 WHERE id = 3;
  => COMMIT;
  COMMIT

=> COMMIT;

ERROR: could not serialize access due to read/write dependencies
among transactions
DETAIL: Reason code: Canceled on identification as a pivot, during
commit attempt.
HINT: The transaction might succeed if retried.
The scenario with the read-only transaction anomaly will lead to the same error.
```

읽기 전용 트랜잭션 이상을 가진 예제도 동일한 오류로 이어집니다.

읽기 전용 트랜잭션 지연. PostgreSQL에서는 읽기 전용 트랜잭션 이상 현상을 피하기 위해 트랜잭션의 실행을 안전해질 때까지 지연시키는 방법을 제공합니다. 이는 데이터의 일관성을 유지하기 위한 중요한 방법입니다. 이 경우, 행 수정로 인해 `SELECT` 문이 차단되는 유일한 경우가 됩니다.

읽기 전용 트랜잭션 이상 현상이 어떻게 작동하는지 확인하기 위해, 간단한 예제를 반복해보겠습니다

```
=> UPDATE accounts SET amount = 900.00 WHERE id = 2;
=> UPDATE accounts SET amount = 100.00 WHERE id = 3;
=> SELECT * FROM accounts WHERE client = 'bob' ORDER BY id;
 id ¦ client ¦ amount
----+--------+--------
  2 ¦ bob    ¦ 900.00
  3 ¦ bob    ¦ 100.00
(2 행s)

=> BEGIN ISOLATION LEVEL SERIALIZABLE; -- 1
=> UPDATE accounts SET amount = amount + (
SELECT sum(amount) FROM accounts WHERE client = 'bob'
) * 0.01
WHERE id = 2;

  => BEGIN ISOLATION LEVEL SERIALIZABLE; -- 2
  => UPDATE accounts SET amount = amount - 100.00 WHERE id = 3;
  => COMMIT;
```

세 번째 트랜잭션을 명시적으로 `READ ONLY` 및 `DEFERRABLE`로 선언해보겠습니다:

```
  => BEGIN ISOLATION LEVEL SERIALIZABLE READ ONLY DEFERRABLE; -- 3
  => SELECT * FROM accounts WHERE client = 'alice';
```

질의를 실행하려고 하면 해당 트랜잭션이 차단됩니다. 그렇지 않으면 이상 현상이 발생할 수 있습니다.

그리고 첫 번째 트랜잭션이 커밋되면, 세 번째 트랜잭션은 실행을 계속할 수 있습니다.

```
=> COMMIT;

   id ¦ client ¦ amount
  ----+--------+---------
    1 ¦ alice  ¦ 1000.00
   (1 행)
   => SELECT * FROM accounts WHERE client = 'bob';
   id ¦ client ¦ amount
  ----+--------+----------
    2 ¦ bob    ¦ 910.0000
    3 ¦ bob    ¦ 0.00
  (2 행s)
  => COMMIT;
```

따라서, 애플리케이션에서 직렬화 격리 수준을 사용하는 경우, 직렬화 실패로 끝난 트랜잭션을 재시도할 수 있는 구조가 필요합니다. 이는 반복 가능한 읽기 수준에서도 같게 적용되며, 이는 읽기 전용 트랜잭션에 국한되지 않습니다.

직렬화 격리 수준은 프로그래밍의 편의성을 제공하지만, 이상 현상 감지와 일부 트랜잭션의 강제 종료로 인한 부하가 따르게 됩니다. 읽기 전용 트랜잭션을 선언할 때 명시적으로 `READ ONLY` 절을 사용하면 이런 부하를 줄일 수 있습니다. 그러나 중요한 점은 중단된 트랜잭션의 비율이 얼마나 되는지입니다. 이 비율이 높을수록 재시도해야 하는 트랜잭션의 수가 많아집니다. PostgreSQL이 실제로 호환되지 않는 트랜잭션만을 중단시키고, 데이터 충돌을 일으키는 것이라면 그렇게 나쁜 상황은 아닐 수 있습니다. 하지만 이는 각 행의 작업을 추적하는 등의 많은 자원을 소모하게 되므로, 불가피하게 부하가 발생하게 됩니다.

현재의 구현은 잘못된 양성 결과를 허용합니다. 즉, PostgreSQL은 단순히 운[luck]이 없어서 안전한 트랜잭션을 중단시킬 수 있습니다. 이 운은 적절한 인덱스의 존재 여부, 사용할 수 있는 잠금의 양 등 여러 요인에 따라 달라지므로, 실제 동작은 사전에 예측하기 어렵습니다.

직렬화 수준을 사용하는 경우, 애플리케이션의 모든 트랜잭션에서 이를 준수해야 합니다. 다른 격리 수준과 결합하면 직렬화는 알림 없이 반복할 수 있는 읽기와 동일하게 동작합니다. 따라서 직렬화 수준을 사용하기로 한 경우, `default_transaction_isolation` 매개변숫값을 적절하게 수정하는 것이 좋습니다.

그리고 직렬화 수준에서 실행되는 질의는 복제본에서 실행할 수 없습니다. 또한, 현재의 제한 사항과 부하 때문에 이 격리 수준의 사용은 덜 매력적일 수 있습니다. 이러한 제약 사항들을 고려하여 애플리케이션의 요구에 가장 적합한 격리 수준을 선택하는 것이 중요합니다.

2.4 어떤 격리 수준을 사용해야 하나요?

PostgreSQL 에서 커밋된 읽기가 기본 격리 수준으로 설정되어 있으며, 많은 애플리케이션에서 이 수준이 사용됩니다. 커밋된 읽기 수준에서는 트랜잭션 실패 시에만 트랜잭션이 중단되며, 이는 데이터 일관성을 유지하면서 직렬화 실패로 인한 트랜잭션 재시도를 걱정하지 않아도 될 수 있습니다.

이 수준의 단점은 가능한 이상 현상이 많다는 것인데, 이에 대해 상세히 설명되었습니다. 개발자는 항상 이를 염두에 두고 코드를 작성하여 이러한 현상이 발생하지 않도록 해야 합니다. 단일 `SQL` 문으로 필요한 모든 작업을 정의하기가 불가능한 경우 명시적인 잠금을 사용해야 합니다. 가장 어려운 부분은 데이터 일관성과 관련된 오류를 테스트하기 어렵다는 점입니다. 이러한 오류는 예측할 수 없고 거의 재현할 수 없는 방식으로 나타나므로 수정하기도 매우 어렵습니다.

반면에 반복 읽기 격리 수준은 일부 일관성 문제를 해결하지만, 모든 문제를 해결하지는 않습니다. 따라서 이 수준에서는 여전히 이상 현상을 고려하고 이를 처리할 수 있는 코드 작성이 필요하며, 직렬화 실패로 인한 트랜잭션 재시도도 고려해야 합니다. 하지만 이 수준은 읽기 전용 트랜잭션에 대해 커밋된 읽기 수준과 잘 보완되며, 복잡한 보고서 작성 등에 유용합니다.

마지막으로, 직렬화 격리 수준은 데이터 일관성에 관한 걱정을 줄여주며, 개발자에게 코드 작성을 더욱 간편하게 만들어줍니다. 이 수준에서는 직렬화 실패로 인한 트랜잭션 재시도를 처리할 수 있는 능력이 요구되지만, 이로 인한 부하가 처리량에 영향을 줄 수 있으며, 복제본에서 지원되지 않는다는 점과 다른 격리 수준과 결합할 수 없다는 점을 염두에 두어야 합니다. 이처럼 각 격리 수준은 그 자체의 장단점을 가지고 있습니다.

따라서 어떤 격리 수준을 선택할 것인가는 애플리케이션의 요구 사항, 개발자의 역량, 시스템의 성능 등 여러 요소를 고려하여 결정해야 합니다.

3 장 페이지와 튜플 Pages and Tuples

3.1 페이지 구조

각 페이지는 일반적으로 다음과 같은 부분으로 구성된 특수한 내부 구성을 가지고 있습니다:[24]

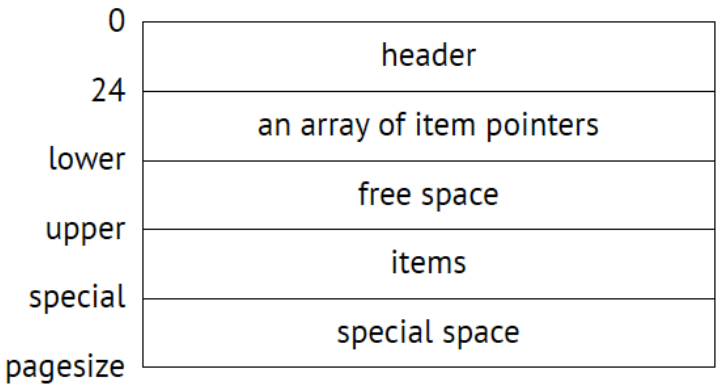
- 페이지 헤더
- 아이템 포인터 배열
- 빈 공간
- 아이템 (행 버전)
- 특수 공간

페이지 헤더

페이지 헤더는 가장 낮은 포인터에 자리 잡고 있으며, 그 크기는 고정되어 있습니다. 페이지 헤더에는 페이지의 체크섬과 페이지의 각 부분의 크기 등, 페이지에 관련된 다양한 정보가 저장되어 있습니다.

`pageinspect`라는 확장 기능을 이용하면 이러한 정보를 쉽게 확인할 수 있습니다.[25] 예를 들어, 테이블의 첫 번째 페이지(페이지 번호는 0부터 시작)를 보고 싶다면 `pageinspect` 확장 기능을 사용하면 됩니다.

```
=> CREATE EXTENSION pageinspect;
=> SELECT lower, upper, special, pagesize
FROM page_header(get_raw_page('accounts',0));

 lower ¦ upper ¦ special ¦ pagesize
-------+-------+---------+----------
   152 ¦ 6904  ¦ 8192    ¦ 8192
(1 행)
```



[24] postgresql.org/docs/14/storage-page-layout.html
include/storage/bufpage.h

[25] postgresql.org/docs/14/pageinspect.html

특수 공간

페이지의 반대쪽 끝, 즉 가장 높은 포인터에 있는 공간을 특수 공간이라고 부릅니다. 이 특수 공간은 일부 인덱스에서 부가 정보를 저장하는 데 이용됩니다. 그러나 모든 인덱스나 테이블 페이지에서 이 공간이 사용되는 것은 아니며, 사용하지 않는 경우 이 공간의 크기는 0입니다.

인덱스 페이지의 구성은 그 종류에 따라 크게 차이 납니다. 페이지 내부의 내용은 주로 해당 인덱스의 유형에 따라 결정됩니다. 실제로 하나의 인덱스 내에서도 페이지의 종류는 다양할 수 있습니다. 예를 들어, `B`-트리 인덱스는 특별한 구조를 가진 메타데이터 페이지(페이지 0)와 일반 테이블 페이지와 유사한 구조를 가진 일반 페이지를 포함하고 있습니다.

튜플

행은 데이터베이스에서 실제로 저장되는 데이터와 일부 추가 정보를 포함하고 있습니다. 이러한 행은 특수 공간 바로 앞에 위치하게 됩니다.

테이블에서는 단순히 '행'이라는 개념보다는 '행 버전'이라는 개념을 다루게 됩니다. 이는 PostgreSQL에서 다중 버전 동시성 제어 방법을 사용하기 때문인데, 이 방법은 하나의 행을 여러 버전으로 관리합니다. 반면, 인덱스는 이 다중 버전 동시성 제어 방법을 사용하지 않으며, 대신 모든 가능한 행 버전에 관한 참조를 가지고 있어, 가시성 규칙에 따라 적절한 버전을 선택하게 됩니다.

> 테이블의 행 버전과 인덱스 항목은 종종 '튜플'이라는 용어로 참조됩니다. 이 용어는 관계 이론에서 유래된 것으로, PostgreSQL 의 학문적 배경을 반영하는 부분입니다.

아이템 포인터

튜플에 관한 포인터 배열은 페이지 내에서 튜플의 위치를 가리키는 역할을 하며, 페이지 헤더 바로 다음에 있게 됩니다.

인덱스 항목은 특정 힙 튜플을 참조해야 하는데, PostgreSQL에서는 이를 위해 6바이트의 튜플 식별자(`TID`)를 사용합니다. 각 튜플 식별자는 메인 포크의 페이지 번호와 그 페이지 내의 특정 행 버전을 가리키게 됩니다.

이론적으로, 튜플은 페이지 시작점으로부터의 오프셋을 통해 참조될 수 있습니다. 하지만 이렇게 되면 튜플이 페이지 내에서 이동할 수 없게 되어, 페이지 단편화와 같은 문제를 초래할 수 있습니다. 이런 문제를 해결하기 위해 PostgreSQL은 간접 포인터 지정 방식을 사용합니다. 튜플 식별자는 해당 포인터 번호를 참조하고, 이 포인터는 튜플의 현재 오프셋을 지정합니다. 이렇게 하면 튜플이 페이지 내에서 이동하더라도 튜플 식별자는 같이 유지되며, 페이지 내의 포인터만 수정하면 됩니다.

각 포인터는 정확히 4바이트로

- 페이지 시작점으로부터의 튜플 오프셋
- 튜플의 길이
- 튜플 상태를 정의하는 여러 비트

포함하고 있습니다.

빈 공간

페이지는 포인터와 튜플 사이에 여유 공간을 가질 수 있으며, 이는 빈 공간 맵에 표시됩니다. 그러나 PostgreSQL에서 페이지 단편화는 발생하지 않습니다. 사용할 수 있는 모든 여유 공간은 항상 하나의 연속된 청크로 집계되어 관리됩니다.[26] 이렇게 함으로써 효율적인 공간 사용과 데이터 관리를 가능하게 합니다.

3.2 행 버전 구조

각 행 버전에는 헤더와 실제 데이터가 포함되어 있습니다. 헤더 부분은 다음과 같은 여러 필드로 구성됩니다:

xmin, xmax 트랜잭션 ID를 나타내며 같은 행의 서로 다른 버전을 구분하는 데 사용됩니다.
infomask 버전 속성을 정의하는 정보 비트의 집합을 제공합니다.
ctid 동일한 행의 다음 수정된 버전을 가리키는 포인터입니다.
null 비트맵bitmap `NULL` 값을 포함할 수 있는 열을 표시하는 비트bit 배열입니다.

따라서 헤더의 크기는 상당히 크며, 튜플마다 최소 23바이트가 필요합니다. 이 수치는 `NULL` 비트맵과 데이터 정렬을 위해 필요한 패딩padding 때문에 종종 증가하기도 합니다. 좁은narrow 테이블에서는 이러한 다양한 메타데이터의 크기가 실제로 저장된 데이터의 크기를 쉽게 초과할 수 있습니다.

디스크 상의 데이터 구조는 메모리 내의 데이터 표현과 완전히 일치하며, 페이지와 그 안의 튜플은 어떠한 변환 없이 그대로 버퍼 캐시로 읽힙니다. 이 때문에 데이터 파일은 다른 플랫폼 간에 호환되지 않습니다.[27]

이러한 호환성 문제의 한 가지 원인은 바이트byte 순서, 즉 엔디안Endian입니다. 예를 들어, `x86` 구조는 리틀 엔디안을 사용하고, `z/Architecture`는 빅 엔디안을 사용하며, `ARM`은 바이트 순서를 설정할 수 있습니다.

또 다른 이유는 기계 워드 경계에 따른 데이터 정렬입니다. 많은 구조에서 이는 필수적인 요구사항입니다. 예를 들어, 32비트 `x86` 시스템에서는 정수형 데이터(4바이트)가 4바이트 워드 경계에 맞춰 정렬되고, 더블 프리시즌 부동 소수점 숫자(8바이트) 역시 4바이트 워드 경계에 맞춰 정렬됩니다. 반면 64비트 시스템에서는 더블 프리시즌 부동 소수점 숫자가 8바이트 워드 경계에 맞춰 정렬됩니다.

데이터 정렬은 테이블의 필드 순서에 따라 튜플의 크기가 달라지게 만듭니다. 이 효과는 일반적으로 무시할 만하지만, 일부 경우에는 크기가 상당히 증가할 수 있습니다. 다음은 예시입니다:

```
=> CREATE TABLE padding(
b1 boolean,
i1 integer,
b2 boolean,
i2 integer
);
=> INSERT INTO padding VALUES (true,1,false,2);
=> SELECT lp_len FROM heap_page_items(get_raw_page('padding', 0));
lp_len
```

[26] backend/storage/page/bufpage.c, PageRepairFragmentation function

[27] include/access/htup_details.h

```
--------
40
(1 행)
```

`pageinspect` 확장 기능의 `heap_page_items 함수를` 통해 포인터와 튜플에 관한 일부 세부 정보를 확인해 보았습니다.

> PostgreSQL 에서 테이블을 힙 heap 이라고 종종 부릅니다. 이 용어는 튜플의 공간 할당과 동적 메모리 할당 사이에 어느 정도 유사성이 있음을 시사하나, 실제로 테이블은 전혀 다른 알고리즘으로 관리됩니다. 힙 이라는 용어는 "모든 것이 힙에 쌓여 있다"는 의미로 이해할 수 있으며, 이는 정렬된 인덱스와는 반대되는 개념입니다.

행의 크기가 40바이트인 경우, 헤더는 24바이트를 차지하고, 정수 타입의 열은 4바이트를 차지하며, 불리언 열은 각각 1바이트를 차지합니다. 이렇게 계산하면 총 34바이트가 사용되는 것이지만, 정수 열의 4바트 정렬을 위해 6바이트가 낭비되는 상황입니다.

테이블을 재구축한다면, 이 공간을 더 효율적으로 사용할 수 있습니다. 이는 데이터의 물리적인 배치를 최적화하여 저장 공간을 절약하고, 데이터 액세스 성능을 향상시키는 좋은 방법입니다.

```
=> DROP TABLE padding;
=> CREATE TABLE padding(
i1 integer,
i2 integer,
b1 boolean,
b2 boolean
);

=> INSERT INTO padding VALUES (1,2,true,false);
=> SELECT lp_len FROM heap_page_items(get_raw_page('padding', 0));
lp_len
--------
34
(1 행)
```

성능 최적화의 다른 가능한 방법의 하나는 `NULL` 값을 포함하지 않는 고정 길이 열로 테이블을 시작하는 것입니다. 이렇게 하면 튜플 내에서의 오프셋을 캐시할 수 있어, 이러한 열에 관한 접근이 더 효율적으로 될 수 있습니다. 즉, 데이터의 위치를 빠르게 찾아낼 수 있어 처리 속도가 향상될 수 있습니다.[28]

3.3 튜플에 관한 작업들

PostgreSQL에서는 동일한 행의 서로 다른 버전을 구분하기 위해 각 버전에 `xmin`과 `xmax`라는 두 가지 값을 부여합니다. 이 값들은 각 행 버전의 '유효 기간validity time'을 정의하는데 사용되지만, 실제 시간이 아니라 증가

[28] backend/access/common/heaptuple.c, heap_deform_tuple function

하는 트랜잭션 ID에 기반합니다.

행이 생성될 때, 해당 행의 xmin 값은 그 행을 생성한 INSERT 질의의 트랜잭션의 ID로 설정됩니다. 반면에 행이 삭제될 때는, 현재 버전의 xmax 값이 DELETE 질의의 트랜잭션의 ID로 설정됩니다.

추상적인 수준에서 보면, UPDATE 질의는 두 단계의 작업으로 이해될 수 있습니다: DELETE와 INSERT입니다. 첫 번째 단계에서는, 현재 행 버전의 xmax 값이 UPDATE 질의를 수행한 트랜잭션 ID로 설정됩니다. 그리고 두 번째 단계에서는, 해당 행의 새로운 버전이 생성되고, 이 새 버전의 xmin 값은 이전 버전의 xmax 값과 같게 설정됩니다.

이제 튜플에 관한 다양한 작업의 저수준 세부사항에 관해 알아보겠습니다.[29]

이 실험을 위해 한 열column에 인덱스가 생성된 두 개의 열로 구성된 테이블이 필요합니다.

```
=> CREATE TABLE t(
id integer GENERATED ALWAYS AS IDENTITY,
s text
);
=> CREATE INDEX ON t(s);
```

Insert

트랜잭션을 시작하고 한 행을 삽입하세요:

```
=> BEGIN;
=> INSERT INTO t(s) VALUES ('FOO');
```

현재 트랜잭션은 다음과 같습니다:

```
=> -- v.13 이전에는 txid_current()를 사용하세요
SELECT pg_current_xact_id();
pg_current_xact_id
--------------------
776
(1 행)
```

> PostgreSQL 에서는 xact 라는 단어를 트랜잭션을 나타내는 용어로 사용합니다. 이 단어는 함수 이름이나 소스 코드에서도 찾아볼 수 있습니다. 그래서 트랜잭션을 xact, txn 또는 간단히 tx 로도 표현할 수 있습니다. 이러한 약어들은 자주 반복해서 나오기 때문에 익숙하게 될 것입니다.

[29] backend/access/transam/README

`heap_page_items` 함수를 통해 모든 필요한 정보를 얻을 수 있지만, 데이터가 원시 형태로 제공되기 때문에 출력 형식을 이해하는 데 약간 어려움이 있을 수 있습니다.

```
=> SELECT *
FROM heap_page_items(get_raw_page('t',0)) \gx
-[ RECORD 1 ]------------------
lp          ¦ 1
lp_off      ¦ 8160
lp_flags    ¦ 1
lp_len      ¦ 32
t_xmin      ¦ 776
t_xmax      ¦ 0
t_field3    ¦ 0
t_ctid      ¦ (0,1)
t_infomask2 ¦ 2
t_infomask  ¦ 2050
t_hoff      ¦ 24
t_bits      ¦
t_oid       ¦
t_data      ¦ \x0100000009464f4f
```

더 읽기 쉽게 만들기 위해서는 일부 정보를 생략하고, 필요한 열을 확장하는 것이 가능합니다.

```
=> SELECT '(0,'¦¦lp¦¦')' AS ctid,
        CASE lp_flags
            WHEN 0 THEN 'unused'
            WHEN 1 THEN 'normal'
            WHEN 2 THEN 'redirect to '¦¦lp_off
            WHEN 3 THEN 'dead'
        END AS state,
    t_xmin as xmin,
    t_xmax as xmax,
   (t_infomask & 256) > 0 AS xmin_committed,
   (t_infomask & 512) > 0 AS xmin_aborted,
   (t_infomask & 1024) > 0 AS xmax_committed,
   (t_infomask & 2048) > 0 AS xmax_aborted
FROM heap_page_items(get_raw_page('t',0)) ₩gx

-[ RECORD 1 ]--+--------
ctid           ¦ (0,1)
```

```
state            ¦ normal
xmin             ¦ 776
xmax             ¦ 0
xmin_committed   ¦ f
xmin_aborted     ¦ f
xmax_committed   ¦ f
xmax_aborted     ¦ t
```

다음과 같은 수정이 이루어졌습니다:

- '`lp`' 포인터는 튜플 포인터의 표준 형식인 (페이지 번호, 포인터 번호)로 변환되었습니다.
- '`lp_flags`' 상태는 상세하게 표시되었습니다. 여기서, 일반 값으로 설정된 것은 실제로 튜플을 가리키고 있음을 의미합니다.
- 모든 정보 비트 중에서, 우리는 지금까지 두 개의 쌍만을 따로 추출하였습니다. '`xmin_committed`'와 '`xmin_aborted`' 비트는 `xmin` 트랜잭션이 커밋되었는지 아니면 중단되었는지를 나타내며, '`xmax_committed`'와 '`xmax_aborted`' 비트는 `xmax` 트랜잭션에 관한 유사한 정보를 제공합니다.

> `pageinspect` 확장 기능은 모든 정보 비트를 설명하는 `heap_tuple_infomask_flags` 함수를 제공하지만, 현재 필요한 정보만을 선택적으로 검색하고 그것을 더 간결하게 표현하는 것이 좋을 것입니다.

실험을 계속해 보겠습니다. `INSERT` 명령은 힙 페이지에 1번 포인터를 추가했는데, 이는 현재 유일하게 존재하는 첫 번째 튜플을 가리킵니다.

튜플의 `xmin` 필드는 현재 트랜잭션 `ID`로 설정되어 있습니다. 이 트랜잭션은 아직 실행 중이므로, `xmin_committed` 및 `xmin_aborted` 비트는 아직 설정되지 않았습니다.

`xmax` 필드에는 0이 들어 있습니다. 이는 튜플이 삭제되지 않았음을, 그리고 이 행이 현재 버전임을 나타내는 더미 숫자입니다. 트랜잭션은 이 숫자를 무시할 것입니다, 왜냐하면 `xmax_aborted` 비트가 설정되어 있기 때문입니다.

> 아직 발생하지 않은 트랜잭션에 대해 중단된 트랜잭션을 나타내는 비트가 설정되는 것은 이상하게 보일 수 있습니다. 그러나, 격리성의 관점에서 보면 이러한 트랜잭션들 사이에는 차이가 없습니다. 중단된 트랜잭션은 어떠한 흔적도 남기지 않기 때문에, 사실상 존재하지 않는 것과 같다고 볼 수 있습니다.

이 질의를 여러 번 사용할 예정이므로, 함수로 만드 것이 좋겠습니다. 동시에, 정보 비트 열을 숨기고 트랜잭션의 상태와 `ID`를 함께 표시함으로써 출력을 더 간결하게 만들 수 있습니다.

```
=> CREATE FUNCTION heap_page(relname text, pageno integer)
    RETURNS TABLE
            (
                ctid  tid,
```

```
                state text,
                xmin  text,
                xmax  text
            ) AS $$
    SELECT (pageno, lp)::text::tid AS ctid, CASE lp_flags
                                              WHEN 0 THEN 'unused'
                                              WHEN 1 THEN 'normal'
                                              WHEN 2 THEN 'redirect to ' || lp_off
                                              WHEN 3 THEN 'dead'
           END AS state,
       t_xmin || CASE
                   WHEN (t_infomask & 256) > 0 THEN ' c'
                   WHEN (t_infomask & 512) > 0 THEN ' a'
                   ELSE ''
           END AS xmin,
       t_xmax || CASE
                   WHEN (t_infomask & 1024) > 0 THEN ' c'
                   WHEN (t_infomask & 2048) > 0 THEN ' a'
                   ELSE ''
           END AS xmax
FROM heap_page_items(get_raw_page(relname, pageno))
ORDER BY lp;
$$
LANGUAGE sql;
```

이제 튜플 헤더에서 어떤 일이 발생하는지 훨씬 더 명확하게 이해할 수 있을 것입니다.

```
=> SELECT * FROM heap_page('t',0);
      ctid ¦       state ¦     xmin ¦     xmax
-----------+-------------+----------+----------
     (0,1) ¦      normal ¦      776 ¦     0 a
(1 행)
```

테이블 자체에서 `xmin`과 `xmax` 의사 열을 질의함으로써, 비슷하지만 상세도가 약간 떨어지는 정보를 얻을 수 있습니다.

```
=> SELECT xmin, xmax, * FROM t;
xmin  ¦ xmax ¦ id ¦ s
------+------+----+-----
  776 ¦ 0    ¦  1 ¦ FOO
(1 행)
```

Commit

트랜잭션이 성공적으로 완료되면, 해당 트랜잭션의 상태를 어떠한 방식으로든 저장해야 합니다. 트랜잭션이 커밋되었다는 사실을 기록해야 하는데, 이를 위해 PostgreSQL은 특별한 `CLOG`(커밋 로그) 구조를 사용합니다.[30] 이 구조는 시스템 카탈로그 테이블이 아니라 `$PGHOME/PGDATA/pg_xact` 디렉터리에 파일 형태로 저장됩니다.

> 이전에는 이러한 파일들이 $PGHOME/PGDATA/pg_clog 디렉터리에 위치했었지만, 포스트그레스규엘 10 버전에서 이 디렉터리의 이름이 변경되었습니다.[31] 포스트그레스규엘에 익숙하지 않은 데이터베이스 관리자들은 "로그"를 필요 없는 것으로 생각하고, 여유 디스크 공간을 확보하기 위해 이 디렉터리를 삭제하는 경우가 많았습니다.

`CLOG`는 접근 편의성을 위해 여러 파일로 분할되며, 이 파일들은 서버의 공유 메모리를 통해 페이지 단위로 접근됩니다.[32]

튜플 헤더와 같이, `CLOG`에도 트랜잭션마다 커밋된 상태와 중단된 상태를 나타내는 두 개의 비트가 있습니다. 트랜잭션이 커밋되면, `CLOG`에 커밋된 비트로 표시됩니다.

다른 트랜잭션이 힙 페이지에 액세스할 때는 다음 두 가지 질문에 답해야 합니다:

- `xmin` 트랜잭션이 이미 종료되었는가?
  아니라면, 생성된 튜플은 보이지 않아야 합니다. PostgreSQL은 트랜잭션이 아직 활성 상태인지 확인하기 위해 인스턴스의 공유 메모리에 있는 또 다른 구조체인 `ProcArray`를 사용합니다. 이 구조체에는 모든 활성 프로세스의 목록이 포함되며, 각 프로세스에 대해 해당하는 현재 활성 트랜잭션이 지정됩니다.
- 그렇다면, 커밋되었는지 아니면 중단되었는지?
  중단되었다면, 해당 튜플 역시 보이지 않아야 합니다. 이를 확인하기 위해 `CLOG`가 필요하며, 이 확인을 수행하는 것은 매번 비용이 많이 듭니다, 심지어 최근의 `CLOG` 페이지들이 메모리 버퍼에 저장되어 있더라도 말이죠.
- 트랜잭션 상태가 결정되면, 해당 상태는 튜플 헤더에 기록됩니다. 구체적으로는 `xmin_committed` 및 `xmin_aborted` 정보 비트, 즉 힌트 비트에 기록됩니다. 이 비트 중 하나가 설정되어 있다면, `xmin` 트랜잭션 상태는 이미 알려진 상태로 간주하며, 다음 트랜잭션은 `CLOG`나 `ProcArray`에 액세스할 필요가 없습니다.

행을 삽입하는 트랜잭션이 이러한 비트를 설정하지 않는 이유는 해당 시점에서 트랜잭션이 성공적으로 완료될지 아직 알 수 없기 때문입니다. 또한 커밋 시점에서 어떤 튜플이나 페이지가 변경되었는지 추적하는 것은 매우 비용이 많이 드는 작업이 될 수 있습니다. 하나의 트랜잭션이 많은 페이지에 영향을 미치는 경우, 이를 추적하는 것은 상당한 비용이 들 수 있습니다. 더욱이, 일부 페이지는 더 이상 캐시에 없을 수 있으므로, 힌트 비트를 간단히 수정하기 위해 다시 읽는 것은 커밋의 속도를 심각하게 늦출 수 있습니다.

---

[30] include/access/clog.h
backend/access/transam/clog.c

[31] commitfest.postgresql.org/13/750

[32] backend/access/transam/clog.c

이러한 비용 절감의 반대편에는, 어떤 트랜잭션(읽기 전용인 `SELECT` 명령 포함)도 힌트 비트를 설정할 수 있으며, 이로 인해 버퍼 캐시에 '더러워진dirtied' 페이지의 흔적이 남게 될 수 있다는 점이 있습니다.

그럼 이제, `INSERT` 문으로 시작된 트랜잭션을 커밋해보도록 하겠습니다.

```
=> COMMIT;
```

페이지에는 아무런 변화가 없었습니다. (그러나 우리는 이미 트랜잭션 상태가 `CLOG`에 기록되었다는 것을 알고 있습니다.)

```
=> SELECT * FROM heap_page('t',0);
  ctid ¦  state ¦ xmin ¦ xmax
-------+--------+------+------
 (0,1) ¦ normal ¦  776 ¦ 0 a
(1 행)
```

이제 "표준standard" 방식으로 페이지에 접근하는 첫 번째 트랜잭션(`pageinspect`를 사용하지 않는 방식)은 `xmin` 트랜잭션의 상태를 결정하고 힌트 비트를 수정해야 합니다.

```
=> SELECT * FROM t;
 id ¦ s
----+---
  1 ¦ FOO
(1 행)

=> SELECT * FROM heap_page('t',0);
  ctid ¦  state ¦  xmin ¦ xmax
-------+--------+-------+------
 (0,1) ¦ normal ¦ 776 c ¦ 0 a
(1 행)
```

Delete

행이 삭제될 경우, 해당 행의 현재 버전의 `xmax` 필드는 삭제를 수행한 트랜잭션 `ID`로 설정되며, `xmax_aborted` 비트는 해제됩니다.

> 이 트랜잭션이 활성 상태인 동안 `xmax` 값은 행 잠금의 역할을 합니다. 다른 트랜잭션이 이 행을 수정하거나 삭제하려면, `xmax` 트랜잭션이 완료될 때까지 대기해야 합니다.

한 행을 삭제해보겠습니다.

```
=> BEGIN;
=> DELETE FROM t;
=> SELECT pg_current_xact_id();
pg_current_xact_id
--------------------
777
(1 행)
```

트랜잭션 `ID`는 이미 `xmax` 필드에 기록되었지만, 정보 비트는 아직 설정되지 않았습니다.

```
=> SELECT * FROM heap_page('t',0);
  ctid ¦  state ¦  xmin ¦ xmax
-------+--------+-------+------
 (0,1) ¦ normal ¦ 776 c ¦ 777
(1 행)
```

Abort

트랜잭션을 중단하는 메커니즘은 커밋하는 것과 유사하며, 매우 빠르게 일어나지만, `CLOG`에서 커밋된 비트 대신 중단된 비트를 설정합니다. 이 동작은 롤백ROLLBACK이라고 불리지만, 실제로 데이터 롤백은 발생하지 않습니다. 중단된 트랜잭션에 의해 데이터 페이지에서 수행된 모든 변경 사항은 그대로 유지됩니다.

```
=> ROLLBACK;
=> SELECT * FROM heap_page('t',0);
  ctid ¦  state ¦  xmin ¦ xmax
-------+--------+-------+------
 (0,1) ¦ normal ¦ 776 c ¦ 777
(1 행)
```

페이지에 접근할 때, 트랜잭션 상태가 확인되고, 튜플은 `xmax_aborted` 힌트 비트를 받게 됩니다. `xmax` 번호 자체는 여전히 페이지에 남아 있지만, 이제는 더 이상 그것에 대해 신경을 쓸 필요가 없게 됩니다.

```
=> SELECT * FROM t;
 id ¦ s
----+-----
  1 ¦ FOO
(1 행)

=> SELECT * FROM heap_page('t',0);
  ctid ¦ state  ¦  xmin ¦ xmax
-------+--------+-------+-------
 (0,1) ¦ normal ¦ 776 c ¦ 777 a
(1 행)
```

Update

수정은 현재 튜플이 삭제된 것처럼 처리되며, 그 이후에 새로운 튜플이 삽입되는 방식으로 진행됩니다.

```
=> BEGIN;
=> UPDATE t SET s = 'BAR';
=> SELECT pg_current_xact_id();
pg_current_xact_id
--------------------
778
(1 행)
```

이 질의는 단일 행(새로운 버전)을 반환합니다.

```
=> SELECT * FROM t;
 id ¦ s
----+-----
  1 ¦ BAR
(1 행)
```

그러나 페이지는 두 버전의 행을 모두 유지합니다.

```
=> SELECT * FROM heap_page('t',0);
  ctid ¦ state   ¦ xmin ¦ xmax
-------+--------+-------+------
 (0,1) ¦ normal ¦ 776 c ¦ 778
 (0,2) ¦ normal ¦ 778   ¦ 0 a
(2 행s)
```

이전에 삭제된 버전의 `xmax` 필드에는 현재 트랜잭션 ID가 포함되어 있습니다. 이 값은 이전 트랜잭션이 중단되었기 때문에 이전 값을 덮어씁니다. `xmax_aborted` 비트는 현재 트랜잭션의 상태가 아직 알려지지 않았기 때문에 해제됩니다.

이 실험을 완료하기 위해 트랜잭션을 커밋해보도록 하겠습니다

```
=> COMMIT;
```

3.4 인덱스

인덱스는 행 버전 관리를 사용하지 않으며 이는 인덱스의 종류에 상관없이 동일합니다. 각 행은 정확히 하나의 튜플로 표현되며, 인덱스 행 헤더에는 `xmin`과 `xmax` 필드가 포함되지 않습니다. 인덱스 항목은 해당 테이블 행의 모든 버전을 가리킵니다. 어떤 행 버전이 보이는지 확인하려면 트랜잭션은 테이블에 접근해야 합니다(필요한 페이지가 가시성visibility 맵에 나타나지 않은 경우).

편의를 위해, 페이지에 있는 모든 인덱스 항목을 표시하기 위해 pageinspect를 사용하는 간단한 함수를 생성해보도록 하겠습니다. (B-트리 인덱스 페이지는 이를 평면 리스트로 저장합니다.)

```
=> CREATE FUNCTION index_page(relname text, pageno integer)
RETURNS TABLE(itemoffset smallint, htid tid)
AS $$
SELECT itemoffset,
       htid -- ctid before v.13
FROM bt_page_items(relname,pageno);
$$ LANGUAGE sql;
```

해당 페이지는 현재와 이전의 두 개의 힙 튜플을 모두 참조합니다.

```
=> SELECT * FROM index_page('t_s_idx',1);
 itemoffset ¦ htid
------------+-------
          1 ¦ (0,2)
          2 ¦ (0,1)
(2 행s)
```

BAR < FOO이므로, 두 번째 튜플에 관한 포인터가 인덱스에서 먼저 나타납니다.

3.5 토스트TOAST

토스트 테이블은 사실상 일반 테이블이며, 본체 테이블의 행 버전과는 독립적으로 자체 버전 관리를 수행합니다. 그러나 토스트 테이블의 행은 수정되지 않고 추가되거나 삭제되는 방식으로 처리되므로, 그들의 버전 관리는 약간 인위적입니다.

데이터 변경이 있을 때마다 주 테이블에는 새로운 튜플이 생깁니다. 그러나 토스트에 저장된 긴 값에 영향을 주지 않는 수정의 경우, 새로운 튜플은 기존의 토스트값을 참조합니다. 긴 값이 수정되는 경우에만, PostgreSQL은 주 테이블에 새로운 튜플과 새로운 토스트를 생성합니다.

3.6 가상Virtual 트랜잭션

PostgreSQL은 트랜잭션 ID를 절약하기 위해 특별한 최적화 기능을 제공합니다. 트랜잭션이 읽기 전용인 경우, 해당 트랜잭션은 행의 가시성에 아무런 영향을 미치지 않습니다. 따라서 이러한 트랜잭션에는 처음에 가상 XID[33]가 부여됩니다. 가상 XID는 백엔드 프로세스 ID와 순차적인 번호로 구성됩니다. 가상 XID를 할당하는 것은 다른 프로세스 간의 동기화를 필요로하지 않으므로 매우 빠르게 수행됩니다. 이 시점에서 트랜잭션은 아직 실제 ID가 없습니다.

```
=> BEGIN;
=> -- txid_current_if_assigned() before v.13
```

[33] backend/access/transam/xact.c

```
SELECT pg_current_xact_id_if_assigned();
pg_current_xact_id_if_assigned
--------------------------------
(1 행)
```

다른 시점에서 시스템에는 이미 사용된 가상 `XID`가 포함될 수 있습니다. 이는 완전히 정상적인 현상입니다. 가상 `XID`는 RAM에서만 존재하며, 해당 트랜잭션이 활성 상태인 동안에만 유지됩니다. 가상 `XID`는 데이터 페이지에 기록되지 않으며 디스크에 저장되지 않습니다.

트랜잭션이 데이터를 수정하기 시작하면, 실제로 고유한 `ID`가 부여됩니다.

```
=> UPDATE accounts
SET amount = amount - 1.00;
=> SELECT pg_current_xact_id_if_assigned();
pg_current_xact_id_if_assigned
--------------------------------
                            780
(1 행)

=> COMMIT;
```

3.7 하위 트랜잭션Subtransactions

저장점Savepoints

SQL은 저장점을 지원하여 트랜잭션 내에서 일부 작업을 전체 트랜잭션을 중단하지 않고 취소할 수 있도록 합니다. 그러나 이러한 예시는 위에서 설명한 방식과 일치하지 않습니다. 트랜잭션의 상태는 모든 작업에 적용되며, 물리적인 데이터 롤백은 수행되지 않습니다.

이러한 기능을 구현하기 위해, 저장점을 포함하는 트랜잭션은 여러 개의 하위 트랜잭션[34]으로 분할되어 각각의 상태를 별도로 관리할 수 있습니다.

하위 트랜잭션은 본 트랜잭션의 `ID`보다 큰 고유한 `ID`를 가지고 있습니다. 하위 트랜잭션의 상태는 일반적인 방식으로 `CLOG`에 기록됩니다. 그러나 커밋된 하위 트랜잭션은 커밋된 상태와 중단된 상태를 동시에 가질 수 있습니다. 최종 결정은 본 트랜잭션의 상태에 따라 달라집니다. 본 트랜잭션이 중단된 경우, 모든 하위 트랜잭션은 중단된 것으로 간주됩니다.

하위 트랜잭션에 관한 정보는 `$PGHOME/PGDATA/pg_subtrans` 디렉토리에 저장됩니다. 파일 접근은 인스턴스의 공유 메모리에 위치한 `CLOG` 버퍼와 동일한 구조를 가진 버퍼를 통해 처리됩니다.[35]

[34] backend/access/transam/subtrans.c

[35] backend/access/transam/slru.c

하위 트랜잭션과 자율 트랜잭션 autonomous 을 혼동하지 않는 것이 중요합니다. 하위 트랜잭션과는 달리 자율 트랜잭션은 어떠한 방식으로도 서로 의존하지 않습니다. 기본 PostgreSQL 은 자율 트랜잭션을 지원하지 않으며, 이것은 아마도 가장 좋은 선택일 것입니다. 자율 트랜잭션은 매우 드문 경우에 필요하지만, 다른 데이터베이스 시스템에서 이용 가능하다면 남용이 유발되어 많은 문제를 야기할 수 있습니다.

오류 및 원자성 Errors and Atomicity

만약 문장 실행 중에 오류가 발생한다면 어떻게 될까요?

```
=> BEGIN;
=> SELECT * FROM t;
 id ¦ s
----+-----
  2 ¦ FOO
  4 ¦ BAR
(2 행s)
=> UPDATE t SET s = repeat('X', 1/(id-4));
ERROR: division by zero
```

실패한 후, 전체 트랜잭션은 중단되며 추가적인 작업을 수행할 수 없습니다.

```
=> SELECT * FROM t;
ERROR: current transaction is aborted, commands ignored until end
of transaction block
```

커밋을 시도하더라도, PostgreSQL은 트랜잭션이 롤백된 것으로 알려줍니다.

```
=> COMMIT;
ROLLBACK
```

실패한 트랜잭션을 계속 진행하는 것이 왜 금지되는지 궁금하나요? 이미 실행된 작업들은 롤백되지 않기 때문에 오류 발생 전에 이루어진 변경 사항에 접근할 수 있게 됩니다. 이는 문장의 원자성을, 그리고 따라서 트랜잭션의 원자성을 깨뜨리게 됩니다.

예를 들어 실험 상황에서, 운영자는 오류 발생 전에 두 행 중 한 행의 수정에 성공했다고 생각해봅시다.

```
=> SELECT * FROM heap_page('t',0);
  ctid ¦  state ¦  xmin ¦ xmax
-------+--------+-------+------
 (0,1) ¦ normal ¦ 782 c ¦ 785
 (0,2) ¦ normal ¦ 783 a ¦ 0 a
```

```
 (0,3) ¦ normal ¦ 784 c ¦ 0 a
 (0,4) ¦ normal ¦ 785   ¦ 0 a
(4 행s)
```

더불어, `psql`은 특별한 모드를 제공해 오류가 발생한 문장이 롤백된 것처럼 보이게 하면서도 오류 발생 후에도 트랜잭션을 계속 진행할 수 있게 해줍니다.

```
=> \set ON_ERROR_ROLLBACK on

=> BEGIN;
=> UPDATE t SET s = repeat('X', 1/(id-4));
ERROR: division by zero

=> SELECT * FROM t;
 id ¦ s
----+-----
  2 ¦ FOO
  4 ¦ BAR
(2 행s)

=> COMMIT;
COMMIT
```

보시다시피, `psql`이 이 모드에서 실행될 때는 각 명령어 앞에 암묵적인 저장점을 추가합니다. 만약 실패하면 롤백이 시작됩니다. 이 모드가 기본적으로 사용되지 않는 이유는, 저장점을 생성하는 것이 (실제로 롤백이 이뤄지지 않더라도) 상당한 오버헤드를 야기하기 때문입니다.

4 장 스냅숏Snapshots

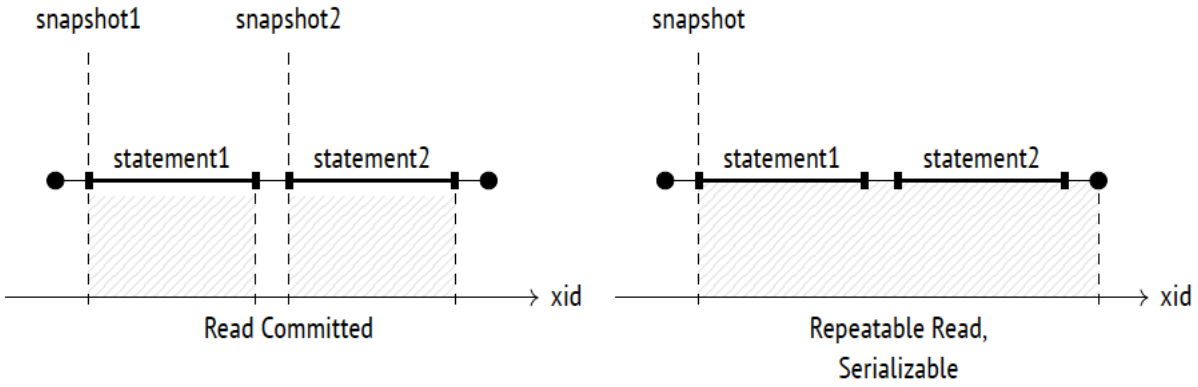
4.1 스냅숏이란 무엇인가요?

하나의 데이터 페이지에는 동일한 행의 여러 버전이 존재할 수 있지만, 각 트랜잭션은 그 중 최대 하나의 버전만 보게 됩니다. 이처럼 다른 행의 보이는 버전들이 모여 스냅숏을 만들게 됩니다. 스냅숏은 해당 시점에 커밋된 최신 데이터를 포함하므로, 특정 시점에 관한 일관된 데이터 뷰(ACID에서 말하는 의미의 일관성)를 제공합니다.

격리성을 보장하기 위해서, 각 트랜잭션은 자신만의 스냅숏을 사용합니다. 이는 서로 다른 트랜잭션이 서로 다른 시간에 촬영된 스냅숏을 볼 수 있음을 의미하지만, 그런데도 데이터의 일관성이 유지됩니다.

커밋된 읽기 격리 수준에서는 각 문장이 시작될 때마다 스냅숏이 촬영되고, 이 스냅숏은 해당 문장이 실행되는 동안만 유지됩니다.

반면, 반복 읽기와 직렬화 수준에서는 트랜잭션의 첫 번째 문장이 시작될 때 스냅숏이 촬영되고, 이 스냅숏은 전체 트랜잭션이 완료될 때까지 유지됩니다.



4.2 행 버전 가시성

스냅숏은 필요한 튜플들의 물리적 복사본이 아닙니다. 대신, 여러 숫자들로 이루어져 있으며, 튜플의 가시성은 특정한 규칙에 따라 결정됩니다.

튜플의 가시성은 튜플 헤더의 xmin과 xmax 필드(즉, 삽입 및 삭제를 수행하는 트랜잭션의 ID)와 이에 관련된 힌트 비트에 의해 정의됩니다. xmin과 xmax 간격이 겹치지 않기 때문에, 각 행은 스냅숏에서 한 버전으로만 나타납니다.

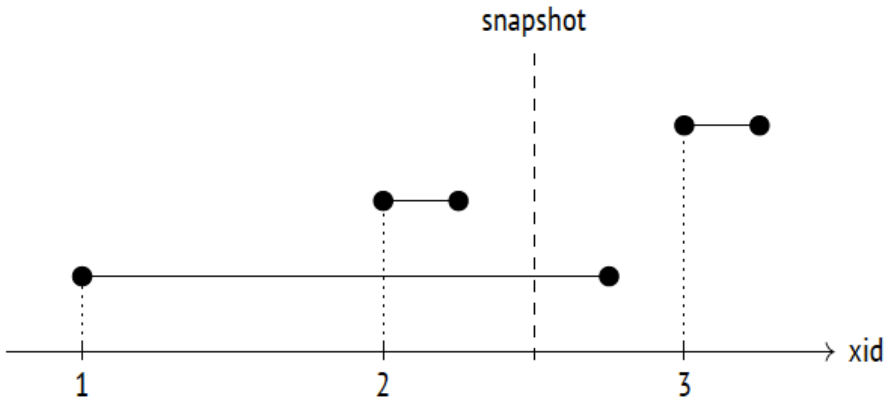
가시성 규칙은 다양한 시나리오와 예외 상황을 고려하기 때문에 상당히 복잡합니다.[36] 매우 간략하게 설명하면, 튜플은 이미 존재하며 아직 삭제되지 않았을 때 스냅숏에서 보이게 됩니다. 이 스냅샷에는 xmin 트랜잭션의 변경사항은 포함되지만 xmax 트랜잭션의 변경사항은 포함되지 않습니다.

마찬가지로, 트랜잭션의 변경 사항은 해당 트랜잭션이 스냅숏 생성 이전에 커밋된 경우에만 스냅숏에서 볼

[36] backend/access/heap/heapam_visibility.c

수 있습니다. 예외적으로, 트랜잭션은 자신이 만든 커밋되지 않은 변경 사항을 볼 수 있습니다. 만약 트랜잭션이 중단된 경우, 해당 변경 사항은 어떤 스냅숏에서도 볼 수 없게 됩니다.

간단한 예시를 통해 살펴봅시다. 이 그림에서 선분은 트랜잭션을 나타내며 (시작 시간부터 커밋 시간까지), 각 트랜잭션의 변경 사항이 스냅숏에서 보여집니다:



여기서 가시성 규칙이 트랜잭션에 적용됩니다:

- 트랜잭션 2는 스냅숏 생성 전에 커밋되었으므로 해당 변경 사항이 가시적입니다.
- 트랜잭션 1은 스냅숏 생성 시간에 활성화되어 있었기 때문에 해당 변경 사항은 가시적이지 않습니다.
- 트랜잭션 3은 스냅숏 생성 이후에 시작되었기 때문에 해당 변경 사항도 가시적이지 않습니다 (이 트랜잭션이 완료되었는지는 상관없습니다).

4.3 스냅숏 구조

그러나 불행히도, 이전의 그림은 PostgreSQL이 실제로 이를 인식하는 방식과는 전혀 관련이 없습니다.[37] 문제는 시스템이 트랜잭션이 언제 커밋되었는지 알 수 없다는 점입니다. 트랜잭션이 시작된 시점은 알려져 있습니다(이는 트랜잭션 ID에 의해 정의됩니다) 그러나 완료된 시점은 어디에도 기록되어 있지 않습니다.

> 커밋 시간은 `track_commit_timestamp` 매개변수를 활성화함으로써 추적할 수 있지만[38](기본값: `off`), 이는 가시성 검사에는 전혀 참여하지 않습니다. (그러나 외부 복제 솔루션에 적용하기 위해 다른 목적으로 추적하는 데는 여전히 유용할 수 있습니다.)
>
> 또한, 포스트그레스규엘은 항상 커밋 및 롤백 시간을 해당 `WAL` 항목에 기록하지만, 이 정보는 시점 복구 Point-In-Time Recovery 에만 사용됩니다.

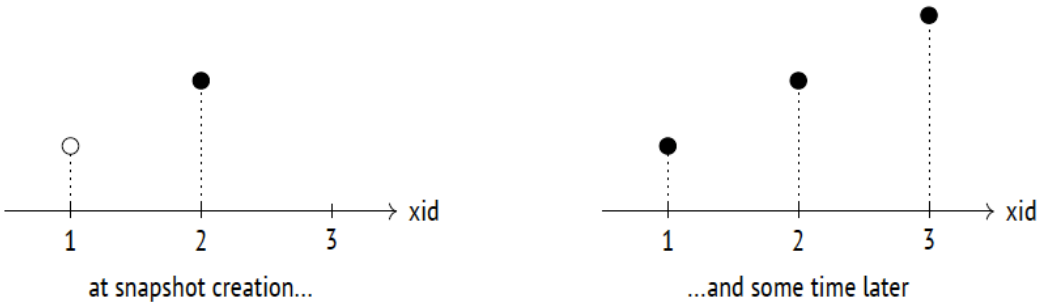
우리가 알 수 있는 것은 오직 트랜잭션의 현재 상태뿐입니다. 이 정보는 서버의 공유 메모리에 저장되어 있습니다. `ProcArray` 구조체에는 모든 활성 세션과 그들의 트랜잭션 목록이 포함되어 있습니다. 트랜잭션이 완료되면 해당 트랜잭션이 스냅숏 생성 시점에 활성화되어 있었는지를 알아낼 수 없게 됩니다. 따라서 스냅숏을

[37] include/utils/snapshot.h
backend/utils/time/snapmgr.c

[38] backend/access/transam/commit_ts.c

생성하기 위해선, 촬영된 시점을 등록하는 것만큼이나, 해당 시점에서의 모든 트랜잭션의 상태를 수집하는 것이 필요합니다. 그렇지 않으면 나중에 어떤 튜플이 스냅숏에서 보여야 하는지, 어떤 튜플이 제외되어야 하는지를 파악하기 어렵습니다.

스냅숏이 촬영된 시점과 이후에 시스템에서 사용할 수 있는 정보를 살펴봅시다. (흰색 원은 활성 트랜잭션을, 검은색 원은 완료된 트랜잭션을 나타냅니다):



가정해 봅시다. 스냅숏이 촬영된 시점에서 첫 번째 트랜잭션이 아직 실행 중이고, 세 번째 트랜잭션이 아직 시작되지 않았다는 사실을 알지 못했다고 생각해 봅시다. 그럼 첫 번째와 세 번째 트랜잭션은 두 번째 트랜잭션과 마찬가지로 보일 것이며, 이들을 필터링하는 것이 불가능해 보일 것입니다.

이러한 이유로 PostgreSQL은 임의의 과거 시점에서 데이터의 일관된 상태를 보여주는 스냅숏을 생성할 수 없습니다. 실제로 모든 필요한 튜플이 힙 페이지에 존재한다 하더라도 그렇습니다. 따라서 회고적인 질의(가끔은 시간적이거나 플래시백 질의라고도 불리는)를 구현하는 것은 불가능합니다.

> 흥미롭게도, 이러한 기능은 원래 PostgreSQL 의 목표 중 하나였으며 처음부터 구현되었습니다. 그러나 이 프로젝트가 커뮤니티로 이관되면서 이 기능은 데이터베이스 시스템에서 제거되었습니다.[39]

따라서, 스냅숏은 생성 시점에 저장된 여러 값으로 구성됩니다:[40]

xmin 스냅숏의 하한이며, 가장 오래된 활성 트랜잭션의 `ID`를 가리킵니다. 이보다 작은 `ID`를 가진 모든 트랜잭션은 이미 커밋되었거나 중단되었으므로, 해당 변경 사항은 스냅쇼에 포함되거나 무시됩니다.

xmax 스냅숏의 상한이며, 최신 커밋 트랜잭션의 `ID`보다 1 큰 값을 가집니다. 이 상한은 스냅숏이 촬영된 시점을 정의합니다. `xmax` 이상의 `ID`를 가진 모든 트랜잭션은 아직 실행 중이거나 존재하지 않으므로, 해당 변경 사항은 가시적이지 않습니다.

xip_list 가상 트랜잭션을 제외한 모든 활성 트랜잭션의 `ID` 목록입니다. 가상 트랜잭션은 가시성에 영향을 미치지 않습니다.

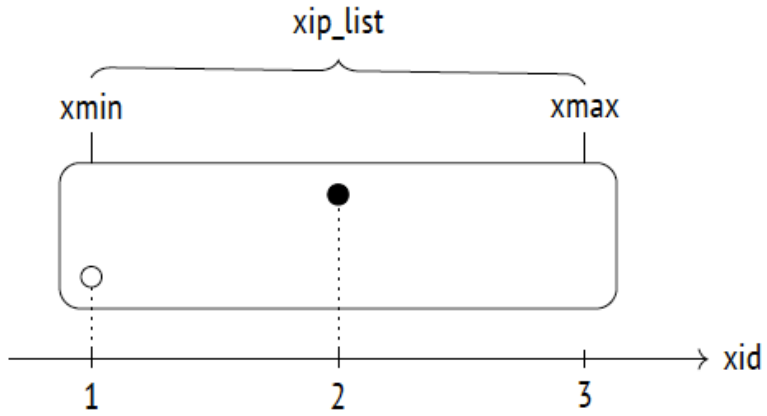
스냅숏에는 몇 가지 다른 매개변수도 포함되지만, 이번 설명에서는 생략하겠습니다.

---

[39] Joseph M. Hellerstein, Looking Back at Postgres. https://arxiv.org/pdf/1901.01973.pdf

[40] backend/storage/ipc/procarray.c, GetSnapshotData function

그래픽으로 표현하면, 스냅숏은 `xmin`부터 `xmax`까지의 트랜잭션을 포함하는 직사각형으로 나타낼 수 있습니다.



스냅숏에 의해 가시성 규칙이 어떻게 정의되는지 이해하기 위해, `accounts` 테이블에서 위의 예시를 재현해 보겠습니다.

```
=> TRUNCATE TABLE accounts;
```

첫 번째 트랜잭션은 테이블에 첫 번째 행을 삽입하고 여전히 열려 있습니다:

```
=> BEGIN;
=> INSERT INTO accounts VALUES (1, 'alice', 1000.00);
=> SELECT pg_current_xact_id();
  pg_current_xact_id
--------------------
                 790
(1 행)
```

두 번째 트랜잭션은 두 번째 행을 삽입하고 이 변경 사항을 즉시 커밋합니다:

```
  => BEGIN;
  => INSERT INTO accounts VALUES (2, 'bob', 100.00);
  => SELECT pg_current_xact_id();
    pg_current_xact_id
  --------------------
                   791
  (1 행)
  => COMMIT;
```

이 시점에서 다른 세션에서 새로운 스냅숏을 생성해 보겠습니다. 이를 위해 단순히 질의를 실행할 수 있지만, 이 스냅숏을 즉시 확인하기 위해 특별한 함수를 사용하겠습니다:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> -- txid_current_snapshot() before v.13
SELECT pg_current_snapshot();
  pg_current_snapshot
---------------------
          790:792:790
(1 행)
```

이 함수는 콜론(:)으로 구분된 다음 스냅샷 구성 요소를 표시합니다: `xmin`, `xmax`, 그리고 `xip_list`(활성 트랜잭션의 목록; 이 특정 경우에는 하나의 항목으로 구성됩니다).

스냅숏을 촬영한 후, 첫 번째 트랜잭션을 커밋하겠습니다:

```
=> COMMIT;
```

세 번째 트랜잭션은 스냅숏 생성 이후에 시작됩니다. 이 트랜잭션은 두 번째 행을 수정하므로 새로운 튜플이 생성됩니다:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount + 100 WHERE id = 2;
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                792
(1 행)
=> COMMIT;
```

우리의 스냅숏은 오직 하나의 튜플만을 볼 수 있습니다.

```
=> SELECT ctid, * FROM accounts;
  ctid | id | client | amount
-------+----+--------+--------
 (0,2) |  2 |    bob | 100.00
(1 행)
```

하지만 테이블에는 세 개의 튜플이 있습니다.

```
=> SELECT * FROM heap_page('accounts',0);
   ctid |      state |      xmin | xmax
--------+------------+-----------+-------
  (0,1) |     normal |     790 c | 0 a
  (0,2) |     normal |     791 c | 792 c
  (0,3) |     normal |     792 c | 0 a
(3 행s)
```

PostgreSQL은 어떻게 버전을 선택하여 보여줄까요? 위의 규칙에 따라, 변경 사항은 다음 기준을 충족하는 커밋된 트랜잭션에 의해서만 스냅숏에 포함됩니다:

- `xid < xmin`: 이 경우 변경 사항은 무조건 보여집니다. 예를 들어, `accounts` 테이블을 생성한 트랜잭션의 경우가 이에 해당합니다.
- `xmin ⩽ xid < xmax`: 이 경우 해당 트랜잭션 `ID`가 `xip_list`에 없는 경우에만 변경 사항이 보여집니다. 첫 번째 행 (0,1)은 `xip_list`에 나타나는 트랜잭션에 의해 추가되었기 때문에 보이지 않습니다. 비록 이 트랜잭션은 스냅숏 범위에 포함되지만 말이죠.

두 번째 행의 가장 최신 버전 (0,3)은 해당 트랜잭션 ID가 스냅숏의 상한을 초과하기 때문에 보이지 않습니다.

그러나 두 번째 행의 첫 번째 버전 (0,2)는 보입니다. 왜냐하면 행의 삽입은 스냅숏 범위 내에 있고 `xip_list`에 나타나지 않는 트랜잭션에 의해 수행되었기 때문입니다(따라서 추가가 보입니다). 반면에 행의 삭제는 스냅숏의 상한을 초과하는 트랜잭션에 의해 이루어졌기 때문에 삭제는 보이지 않습니다.

```
    => COMMIT;
```

4.4 트랜잭션 자체의 변경 사항 가시화

트랜잭션의 자체 변경에 관한 가시성 규칙을 정의하는 것은 약간 더 복잡해집니다. 일부 경우에는 해당 변경 사항의 일부만 보여야 합니다. 예를 들어, 특정 시점에서 열린 커서는 격리 수준과 관계없이 후에 발생한 변경 사항을 보지 않아야 합니다.

이러한 상황을 해결하기 위해 튜플 헤더는 트랜잭션 내에서 작업의 순서 번호를 나타내는 특수 필드를 제공합니다. 이 특수 필드는 `cmin` 및 `cmax` 가상 열로 표시됩니다. `cmin` 열은 추가를 식별하는 데 사용되고, `cmax`는 삭제 작업에 사용됩니다. 공간을 절약하기 위해 이 값은 두 개의 별도 필드가 아닌 튜플 헤더의 단일 필드에 저장됩니다. PostgreSQL은 동일한 행이 한 번의 트랜잭션 내에서 추가와 삭제를 모두 수행하지 않는다고 가정하지만, 만약 그런 경우가 발생하면 PostgreSQL은 이 필드에 특수한 조합 식별자를 작성하고, 실제 `cmin` 및 `cmax` 값은 이 경우에는 백엔드에 의해 저장됩니다.[41]

예시로, 트랜잭션을 시작하고 테이블에 행을 추가해 보겠습니다:

```
=> BEGIN;
=> INSERT INTO accounts VALUES (3, 'charlie', 100.00);
=> SELECT pg_current_xact_id();
   pg_current_xact_id
--------------------
                  793
(1 행)
```

이 테이블의 행 수를 반환하는 질의를 실행하기 위해 커서를 열어보겠습니다:

```
=> DECLARE c CURSOR FOR SELECT count(*) FROM accounts;
```

[41] backend/utils/time/combocid.c

하나의 더 행을 추가해 보세요:

```
=> INSERT INTO accounts VALUES (4, 'charlie', 200.00);
```

이제 출력을 다른 열로 확장하여 우리의 트랜잭션에 의해 삽입된 행들의 `cmin` 값을 표시해 보세요 (다른 행들에 대해서는 의미가 없습니다):

```
=> SELECT xmin, CASE WHEN xmin = 793 THEN cmin END cmin, *
FROM accounts;
 xmin ┆ cmin ┆ id ┆  client ┆ amount
──────┆──────┆────┆─────────┆─────────
  790 ┆      ┆  1 ┆   alice ┆ 1000.00
  792 ┆      ┆  2 ┆     bob ┆ 200.00
  793 ┆    0 ┆  3 ┆ charlie ┆ 100.00
  793 ┆    1 ┆  4 ┆ charlie ┆ 200.00
(4 행s)
```

커서 질의는 세 개의 행만을 가져옵니다. 커서가 이미 열려있을 때 추가된 행은 `cmin < 1` 조건을 만족시키지 않기 때문에 스냅숏에 포함되지 않습니다:

```
=> FETCH c;
 count
───────
     3
(1 행)
```

당연히 이 `cmin` 숫자도 스냅숏에 저장되어 있지만, 어떤 `SQL` 수단을 사용하여도 표시할 수는 없습니다.

4.5 트랜잭션의 범위

앞서 말씀드렸듯이, 스냅숏의 최솟값은 `xmin`이라고 표현되고, 이는 스냅숏이 만들어질 때 가장 오래된 활성 트랜잭션의 `ID`를 나타냅니다. 이 `xmin` 값은 스냅숏을 활용하는 트랜잭션의 범위를 결정하는 매우 중요한 요소입니다.

만약 트랜잭션이 활성화된 스냅숏이 없다면, 예를 들어 커밋된 읽기 격리 수준에서 질의를 수행하는 경우, 그 트랜잭션의 범위는 부여받은 고유 `ID`로 결정됩니다.

그리고 `xmin` 값보다 작은 모든 트랜잭션(`xid < xmin`)은 반드시 커밋되었다고 볼 수 있습니다. 이는 트랜잭션이 자신의 범위를 넘는 현재의 행 버전만을 볼 수 있다는 것을 의미합니다.

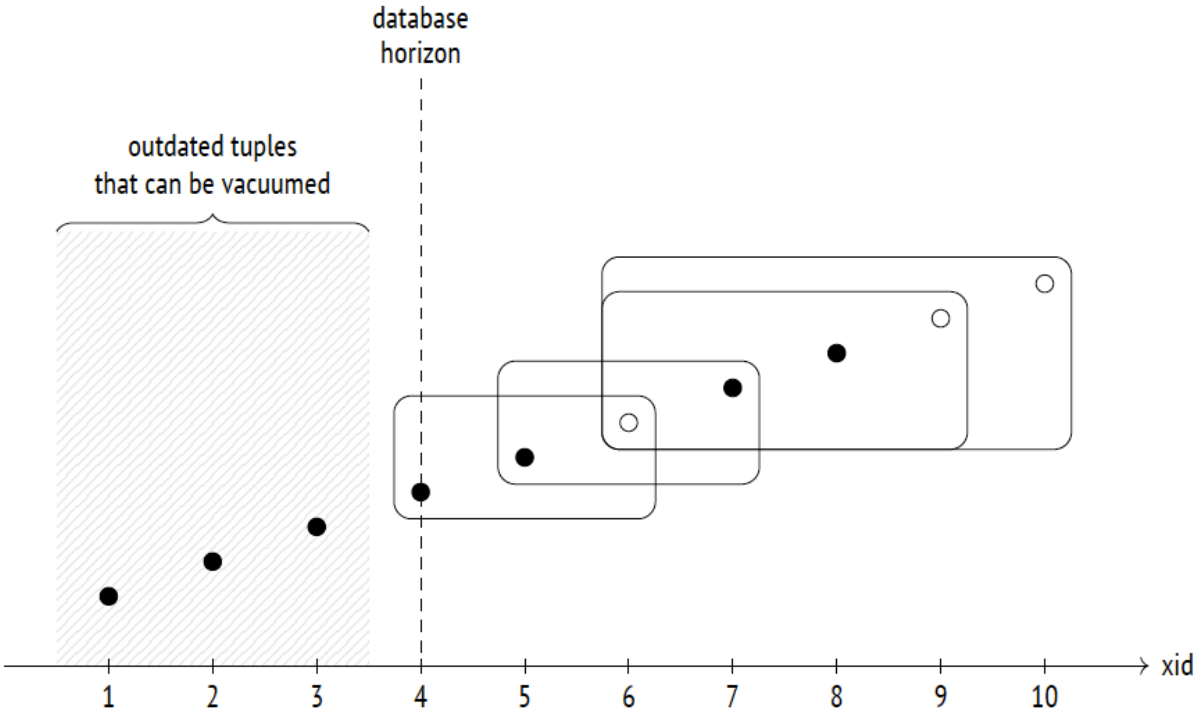
> 짐작할 수 있듯이 이 용어는 물리학의 사건의 **지평선 개념**에서 영감을 받았습니다.

PostgreSQL은 현재 실행 중인 모든 프로세스의 범위를 추적하고 있습니다. 트랜잭션은 `pg_stat_activity` 테이블을 통해 자신의 범위를 확인할 수 있습니다. 아래는 그 예시입니다:

```
=> BEGIN;
=> SELECT backend_xmin FROM pg_stat_activity
WHERE pid = pg_backend_pid();
 backend_xmin
--------------
          793
(1 행)
```

가상 트랜잭션은 실제로는 `ID`를 갖지 않지만, 일반 트랜잭션처럼 스냅숏을 활용하여 자신만의 범위를 설정합니다. 하지만, 스냅숏이 활성화되지 않은 가상 트랜잭션의 경우에는 범위라는 개념이 중요하지 않습니다. 이런 트랜잭션은 스냅숏과 가시성에 대해 시스템에 완전히 "투명transparent"하게 작동합니다. 그렇지만 `pg_stat_activity.backend_xmin`에서는 여전히 이전 스냅숏의 `xmin` 값이 포함될 수 있습니다.

또한, 데이터베이스의 범위(호라이즌)도 비슷한 방식으로 설정됩니다. 이를 위해 데이터베이스의 모든 트랜잭션의 범위를 확인하고, 그중 가장 오래된 범위를 가진 트랜잭션을 선택합니다. 이 가장 오래된 `xmin`[42]을 기준으로, 이 범위를 넘어선, 오래된 힙 튜플은 이 데이터베이스의 어떤 트랜잭션에도 보이지 않게 됩니다. 이런 튜플은 베큠VACUUM을 통해 안전하게 정리될 수 있습니다. 이렇게 범위 개념을 이해하고 활용하는 것이 왜 중요한지를 실질적으로 알 수 있게 됩니다.



자 정리해 보겠습니다:

- 반복 읽기 또는 직렬화 격리 수준에서 실행되는 트랜잭션(실제 트랜잭션, 가상 트랜잭션 구분 없이)

[42] backend/storage/ipc/procarray.c, ComputeXidHorizons function

이 오랫동안 실행 중일 경우, 해당 트랜잭션은 데이터베이스의 범위를 유지하고, 이에 따라 베큠 작업이 지연될 수 있습니다.

- 커밋된 읽기 격리 수준에서 실행되는 실제 트랜잭션도, "`idle in transaction`" 상태에 있더라도 데이터베이스의 범위를 유지합니다.
- 반면 커밋된 읽기 격리 수준에서 실행되는 가상 트랜잭션은 연산을 수행하는 동안에만 범위를 유지합니다.

데이터베이스 전체에는 단 한 개의 범위만 존재합니다. 따라서 트랜잭션이 범위를 유지하는 동안에는, 실제로 해당 트랜잭션이 특정 데이터에 접근하지 않았더라도, 그 범위 내의 데이터를 베큠 작업으로 정리할 수 없게 됩니다. 이러한 이유로 트랜잭션의 범위 관리는 매우 중요합니다.

> 시스템 카탈로그의 클러스터 전체 테이블은 모든 데이터베이스에 걸친 모든 트랜잭션을 고려한 독립적인 범위를 가지고 있습니다. 그러나 임시 테이블은 그럴 필요가 없습니다. 임시 테이블은 현재 프로세스에서 실행 중인 트랜잭션에만 초점을 맞추며, 그 외의 다른 트랜잭션에 대해선 고려할 필요가 없습니다.

저희 현재 실험으로 돌아가보겠습니다. 첫 번째 세션의 활성 트랜잭션은 여전히 데이터베이스의 범위를 유지하고 있습니다. 이를 트랜잭션 카운터를 증가시켜 확인할 수 있습니다:

```
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                794
(1 행)

=> SELECT backend_xmin FROM pg_stat_activity
WHERE pid = pg_backend_pid();
 backend_xmin
--------------
          793
(1 행)
```

그리고 이 트랜잭션이 완료되면, 범위가 앞으로 이동하고 오래된 튜플이 베큠 될 수 있습니다:

```
=> COMMIT;
=> SELECT backend_xmin FROM pg_stat_activity
WHERE pid = pg_backend_pid();
  backend_xmin
--------------
          795
(1 행)
```

이상적인 상황에서는, 장기간 실행되는 트랜잭션과 빈번한 페이지 수정(새로운 행 버전을 생성하는 것)을 조

합하는 것을 피해야 합니다. 그렇지 않으면 테이블과 인덱스가 부풀어bloating 오르게 됩니다.

4.6 시스템 카탈로그 스냅숏

시스템 카탈로그는 일반적인 테이블로 구성되어 있지만, 트랜잭션 또는 연산자가 사용하는 스냅숏으로는 접근이 불가능합니다. 스냅숏은 최신의 변동 사항을 모두 포함하도록 최신 상태를 유지해야 합니다. 그렇지 않다면, 트랜잭션은 테이블 열의 오래된 정의를 확인하거나, 새롭게 추가된 무결성 제약 조건을 놓칠 수 있습니다.

이에 관한 간단한 예시를 들어보겠습니다.

```
=> BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;

=> SELECT 1; -- 트랜잭션에 관한 스냅숏이 생성됩니다.

  => ALTER TABLE accounts
   ALTER amount SET NOT NULL;

=> INSERT INTO accounts(client, amount)
   VALUES ('alice', NULL);
ERROR: null value in column "amount" of relation "accounts"
violates not-null constraint
DETAIL: Failing 행 contains (1, alice, null).

=> ROLLBACK;
```

스냅숏이 생성된 이후에 적용된 무결성 제약 조건은 `INSERT` 명령을 통해 확인할 수 있었습니다. 이런 동작이 격리성을 위반하는 것처럼 보일 수 있지만, 만약 `INSERT`를 수행하는 트랜잭션이 `ALTER TABLE` 명령을 내리기 전에 이미 `accounts` 테이블에 접근했다면, `ALTER TABLE` 명령은 해당 트랜잭션이 완료될 때까지 기다렸을 것입니다.

일반적으로, 서버는 각 시스템 카탈로그 질의가 실행될 때마다 별도의 스냅숏이 생성된 것처럼 동작합니다. 그러나 실제 구현은 이보다 훨씬 복잡하며[43], 자주 스냅숏을 생성하는 것은 성능에 부정적인 영향을 미칠 수 있습니다. 또한 많은 시스템 카탈로그 객체들이 캐시 되므로, 이 부분도 고려해야 합니다.

4.7 스냅숏 내보내기

일부 상황에서는 여러 트랜잭션이 동시에 실행될 때 반드시 같은 스냅숏을 보여야 합니다. 예를 들어, `pg_dump` 유틸리티가 병렬 모드로 작동할 때, 모든 프로세스가 일관된 백업을 생성하기 위해 동일한 데이터베이스 상태를 보여야 합니다.

그러나, 트랜잭션이 '동시에simultaneously' 시작된다고 해서 같은 스냅숏을 가질 것이라고 단정할 수는 없습니다. 모든 트랜잭션이 동일한 데이터를 보기 위해서는 스냅숏 내보내기 메커니즘이 필요합니다.

---

[43] backend/utils/time/snapmgr.c, GetCatalogSnapshot function

`pg_export_snapshot` 함수는 스냅숏의 `ID`를 반환하는데, 이 `ID`는 데이터베이스 시스템 외부의 다른 트랜잭션에 전달될 수 있습니다. 이렇게 함으로써 동일한 스냅숏을 공유하는 트랜잭션들이 생성될 수 있습니다.

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT count(*) FROM accounts;
 count
-------
     4
(1 행)

=> SELECT pg_export_snapshot();
  pg_export_snapshot
---------------------
  00000004-0000006E-1
(1 행)
```

첫 번째 명령을 실행하기 전에, 다른 트랜잭션들은 `SET TRANSACTION SNAPSHOT` 명령을 통해 해당 스냅숏을 가져올 수 있습니다. 이때 트랜잭션의 격리 수준은 반드시 반복 읽기 또는 직렬화로 설정되어야 합니다. 이는 커밋된 읽기 수준의 연산에서는 자체 스냅숏을 사용하기 때문입니다.

```
=> DELETE FROM accounts;
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SET TRANSACTION SNAPSHOT '00000004-0000006E-1';
```

이제 두 번째 트랜잭션은 첫 번째 트랜잭션의 스냅숏을 사용하고, 따라서 0개 대신 4개의 행을 볼 수 있을 것입니다.

```
=> SELECT count(*) FROM accounts;
 count
-------
     4
(1 행)
```

언급한 대로, 두 번째 트랜잭션은 스냅숏이 내보내게 된 이후 첫 번째 트랜잭션이 수행한 변경 사항을 볼 수 없습니다. 반대의 경우도 마찬가지입니다. 이는 일반적인 데이터 가시성 규칙이 여전히 적용되기 때문입니다.

내보내진 스냅숏의 생명주기는 스냅숏을 내보내는 트랜잭션의 생명주기와 동일합니다.

```
  => COMMIT;

=> COMMIT;
```

5 장 페이지 정리와 HOT[Heap-Only Tuple] 수정

5.1 페이지 정리

PostgreSQL 은 힙 페이지를 읽거나 수정하는 동안 빠른 페이지 정리[44] 작업을 수행할 수 있습니다. 이러한 작업은 다음과 같은 경우에 발생합니다:

- 이전의 `UPDATE` 작업에서 동일한 페이지에 새로운 튜플을 저장할 충분한 공간을 찾지 못한 경우, 이는 페이지 헤더에 반영됩니다.
- 힙 페이지에 저장된 데이터가 `fillfactor` 저장 매개변수(기본값: 100)에서 허용되는 양을 초과한 경우입니다.

> `INSERT` 작업은 `fillfactor` 백분율보다 적게 채워진 페이지에만 새로운 행을 추가할 수 있습니다. 나머지 공간은 `UPDATE` 작업을 위해 보존됩니다. 기본적으로는 이러한 공간이 예약되지 않습니다

페이지 정리[pruning]는 더 이상 어떤 스냅숏에서도 볼 수 없는 튜플(즉, 데이터베이스 범위를 벗어난 튜플)을 제거합니다. 이 작업은 단일 힙 페이지를 벗어나지 않지만 매우 빠르게 수행됩니다. 정리된 튜플에 관한 포인터는 그대로 남아 있으며, 이미 다른 페이지인 인덱스에서 참조될 수 있습니다.

동일한 이유로 가시성 맵이나 빈 공간 맵은 갱신되지 않습니다. 따라서 회복된 공간은 삽입이 아닌 수정을 위해 확보됩니다.

페이지가 읽기 중에 정리될 수 있기 때문에, 어떤 `SELECT` 문도 페이지 수정을 일으킬 수 있습니다. 이는 정보 비트의 지연 설정 외에도 또 다른 경우입니다.

페이지 정리가 실제로 어떻게 작동하는지 살펴보겠습니다. 두 개의 열로 구성된 테이블을 생성하고 각 열에 대해 인덱스를 작성할 것입니다:

```
=> CREATE TABLE hot(id integer, s char(2000)) WITH (fillfactor = 75);
=> CREATE INDEX hot_id ON hot(id);
=> CREATE INDEX hot_s ON hot(s);
```

`s` 열이 라틴 문자만을 포함한다면, 각 힙 튜플은 2,004 바이트의 고정 크기를 가지며, 헤더는 24 바이트입니다. `fillfactor` 저장 매개변수는 75%로 설정되어 있습니다. 이는 페이지에 네 개의 튜플에 관한 충분한 여유 공간이 있지만, 우리는 세 개만 삽입할 수 있습니다.

다음과 같이 새로운 행을 삽입하고 여러 번 수정해보겠습니다:

```
=> INSERT INTO hot VALUES (1, 'A');
=> UPDATE hot SET s = 'B';
=> UPDATE hot SET s = 'C';
=> UPDATE hot SET s = 'D';
```

[44] backend/access/heap/pruneheap.c, heap_page_prune_opt function

지금 페이지에는 네 개의 튜플이 포함되어 있습니다.

```
=> SELECT * FROM heap_page('hot',0);
  ctid ¦  state ¦  xmin ¦ xmax
-------+--------+-------+-------
 (0,1) ¦ normal ¦ 801 c ¦ 802 c
 (0,2) ¦ normal ¦ 802 c ¦ 803 c
 (0,3) ¦ normal ¦ 803 c ¦ 804
 (0,4) ¦ normal ¦ 804.  ¦ 0 a
 (4 행s)
```

예상대로, 우리는 `fillfactor` 임계값을 초과했습니다. 페이지 크기와 상한 값의 차이를 통해 확인할 수 있습니다. 이 값은 페이지 크기인 6,144 바이트의 75%보다 큽니다:

```
=> SELECT upper, pagesize FROM page_header(get_raw_page('hot',0));
    upper ¦ pagesize
--------+------------
       64 ¦ 8192
 (1 행)
```

다음 페이지 접근 시, 페이지 정리가 수행되어 모든 오래된 튜플이 제거됩니다. 그 후, 해제된 공간에 새로운 튜플 (0,5)이 추가됩니다.

```
=> UPDATE hot SET s = 'E';
=> SELECT * FROM heap_page('hot',0);
  ctid ¦  state ¦ xmin  ¦ xmax
-------+--------+-------+------
 (0,1) ¦ dead   ¦       ¦
 (0,2) ¦ dead   ¦       ¦
 (0,3) ¦ dead   ¦       ¦
 (0,4) ¦ normal ¦ 804 c ¦ 805
 (0,5) ¦ normal ¦ 805   ¦ 0 a
 (5 행s)
```

남아 있는 힙 튜플은 물리적으로 가장 높은 포인터로 이동되어 모든 빈 공간이 하나의 연속된 청크로 집약됩니다. 또한 튜플 포인터도 이에 맞게 수정됩니다. 이로 인해 페이지에는 빈 공간의 단편화가 없습니다.

정리된 튜플에 관한 포인터는 아직 제거할 수 없습니다. 이는 인덱스에서 여전히 참조되기 때문입니다. PostgreSQL은 이러한 튜플의 상태를 정상`normal`에서 더미`dead`로 변경합니다. 이제 `hot_s` 인덱스의 첫 번째 페이지를 살펴보겠습니다. (메타데이터에는 0 페이지가 사용됩니다).

```
=> SELECT * FROM index_page('hot_s',1);
 itemoffset ¦ htid
------------+-------
```

```
          1 ¦ (0,1)
          2 ¦ (0,2)
          3 ¦ (0,3)
          4 ¦ (0,4)
          5 ¦ (0,5)
 (5 행s)
```

다른 인덱스에서도 동일한 상황을 볼 수 있습니다.

```
=> SELECT * FROM index_page('hot_id',1);
 itemoffset ¦ htid
------------+-------
          1 ¦ (0,1)
          2 ¦ (0,2)
          3 ¦ (0,3)
          4 ¦ (0,4)
          5 ¦ (0,5)
 (5 행s)
```

인덱스 스캔은 (0, 1), (0, 2), 그리고 (0, 3)과 같은 튜플 식별자를 반환할 수 있습니다. 그러나 서버가 해당 힙 튜플을 읽으려고 할 때, 포인터가 더미 상태임을 인식합니다. 이는 해당 튜플이 더 이상 존재하지 않고 무시되어야 함을 의미합니다. 또한, 서버는 반복적인 힙 페이지 접근을 피하기 위해 인덱스 페이지에서 포인터 상태를 수정합니다.[45]

인덱스 페이지를 표시하는 함수를 확장하여, 포인터가 더미인지 여부를 표시하도록 해보겠습니다.

```
=> DROP FUNCTION index_page(text, integer);

=> CREATE FUNCTION index_page(relname text, pageno integer)
RETURNS TABLE(itemoffset smallint, htid tid, dead boolean)
AS $$
SELECT itemoffset,
       htid,
       dead -- starting from v.13
FROM bt_page_items(relname,pageno);
$$ LANGUAGE sql;

=> SELECT * FROM index_page('hot_id',1);
 itemoffset ¦ htid  ¦ dead
------------+-------+------
          1 ¦ (0,1) ¦ f
          2 ¦ (0,2) ¦ f
          3 ¦ (0,3) ¦ f
```

[45] backend/access/index/indexam.c, index_fetch_heap function

```
          4 ¦ (0,4) ¦ f
          5 ¦ (0,5) ¦ f
  (5 행s)
```

현재 인덱스 페이지의 모든 포인터는 활성 상태입니다. 그러나 처음으로 인덱스 스캔이 발생하면 포인터의 상태가 변경됩니다.

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM hot WHERE id = 1;
QUERY PLAN
-----------------------------------------------------------
Index Scan using hot_id on hot (actual 행s=1 loops=1)
  Index Cond: (id = 1)
(2 행s)

=> SELECT * FROM index_page('hot_id',1);
 itemoffset ¦  htid ¦ dead
------------+-------+------
          1 ¦ (0,1) ¦ t
          2 ¦ (0,2) ¦ t
          3 ¦ (0,3) ¦ t
          4 ¦ (0,4) ¦ t
          5 ¦ (0,5) ¦ f
  (5 행s)
```

네 번째 포인터가 가리키는 힙 튜플은 아직 정리되지 않아 정상 상태로 남아 있습니다. 하지만 이미 데이터베이스 범위를 넘어섰습니다. 이 때문에 해당 포인터는 인덱스에서 더미로 표시는 것입니다.

5.2 HOT 수정

모든 힙 튜플에 관한 참조를 인덱스에서 관리하는 것은 매우 비효율적인 방법입니다. 첫째, 행의 각 수정 작업은 테이블에 생성된 모든 인덱스를 수정해야 하기 때문입니다. 새로운 힙 튜플이 생겨나면, 그 튜플에 관한 참조는 모든 인덱스에 포함되어야 합니다. 이는 수정된 필드가 인덱스에 포함되지 않았더라도 해당합니다.

둘째, 인덱스는 힙 튜플에 관한 역사적인 참조를 쌓아가는데, 이 참조들도 튜플이 정리됨에 따라 함께 정리되어야 합니다.

마지막으로, 테이블에 인덱스를 더 많이 생성하면, 이런 문제들은 더욱 심해집니다.

그러나, 수정되는 열이 어떤 인덱스에도 속하지 않는다면, 동일한 키값을 가진 새로운 인덱스 항목을 만드는 것은 불필요합니다. 이런 중복을 방지하기 위해 PostgreSQL은 `Heap-Only Tuple` 수정이라는 최적화 기능을 제공합니다.[46]

[46] backend/access/heap/README.HOT

HOT 수정을 사용하면, 인덱스 페이지는 각 행에 대해 오직 한 개의 항목만을 포함하게 됩니다. 이 항목은 해당 행의 맨 처음 버전을 가리키게 됩니다. 그리고 같은 페이지에 있는 모든 후속 버전은 튜플 헤더의 ctid 포인터를 통해 연결된 사슬 형태를 이룹니다.

인덱스에서 참조되지 않는 행 버전은 HOT 비트로 표시되어, HOT 사슬에 포함된 버전은 Heap Hot Updated 비트로 표시됩니다. 이렇게 함으로써 효율적이며 중복을 최소화한 데이터 관리가 가능해집니다.

만약 인덱스 스캔이 힙 페이지에 접근해 Heap Hot Updated로 표시된 행 버전을 찾게 된다면, 이것은 스캔이 계속 진행되어야 한다는 신호입니다. 따라서 스캔은 HOT 수정 사슬을 따라 계속 진행합니다. 물론, 결과를 클라이언트에 반환하기 전에 모든 행 버전의 가시성을 확인하는 과정은 필수적입니다.

HOT 수정이 실제로 어떻게 작동하는지 살펴보기 위해, 한 번 인덱스를 하나 제거하고 테이블을 비워 보는 것은 어떨까요? 이렇게 해서 HOT 수정의 동작 방식을 직접 확인해 볼 수 있습니다.

```
=> DROP INDEX hot_s;
=> TRUNCATE TABLE hot;
```

편의를 위해, 우리는 heap_page 함수를 다시 정의해서 출력에 ctid와 HOT 수정에 관련된 두 가지 비트를 포함시키는 것이 좋겠습니다. 이렇게 하면 더욱 자세한 정보를 확인하며 분석할 수 있을 것입니다.

```
=> DROP FUNCTION heap_page(text,integer);
=> CREATE FUNCTION heap_page(relname text, pageno integer)
RETURNS TABLE(
ctid tid, state text,
xmin text, xmax text,
hhu text, hot text, t_ctid tid
) AS $$
SELECT (pageno,lp)::text::tid AS ctid,
  CASE lp_flags
    WHEN 0 THEN 'unused'
    WHEN 1 THEN 'normal'
    WHEN 2 THEN 'redirect to '||lp_off
    WHEN 3 THEN 'dead'
  END AS state,
  t_xmin || CASE
    WHEN (t_infomask & 256) > 0 THEN ' c'
    WHEN (t_infomask & 512) > 0 THEN ' a'
    ELSE ''
  END AS xmin,
  t_xmax || CASE
    WHEN (t_infomask & 1024) > 0 THEN ' c'
    WHEN (t_infomask & 2048) > 0 THEN ' a'
    ELSE ''
  END AS xmax,
  CASE WHEN (t_infomask2 & 16384) > 0 THEN 't' END AS hhu,
```

```
  CASE WHEN (t_infomask2 & 32768) > 0 THEN 't' END AS hot,
  t_ctid
FROM heap_page_items(get_raw_page(relname,pageno))
ORDER BY lp;
$$ LANGUAGE sql;
```

삽입 및 수정작업을 반복해 보겠습니다.

```
=> INSERT INTO hot VALUES (1, 'A');
=> UPDATE hot SET s = 'B';
```

현재 페이지에는 `HOT` 수정의 연속이 담겨 있습니다:

- 힙의 `Hot Updated` 비트는 `CTID` 사슬을 따라 이동해야 한다는 것을 표시합니다.
- `Heap Only Tuple` 비트는 이 튜플이 어떤 인덱스에서도 참조되지 않는다는 것을 표시합니다

```
=> SELECT * FROM heap_page('hot',0);
  ctid ¦  state ¦  xmin ¦ xmax ¦ hhu ¦ hot ¦ t_ctid
-------+--------+-------+------+-----+-----+--------
 (0,1) ¦ normal ¦ 812 c ¦ 813  ¦ t   ¦     ¦ (0,2)
 (0,2) ¦ normal ¦ 813   ¦ 0 a  ¦     ¦  t  ¦ (0,2)
(2 행s)
```

추가적인 수정이 이루어질수록 연쇄는 더욱 커지게 될 것입니다. 하지만 이 증가는 페이지의 제한 범위 내에서만 발생하게 될 것입니다.

```
=> UPDATE hot SET s = 'C';
=> UPDATE hot SET s = 'D';

=> SELECT * FROM heap_page('hot',0);
  ctid ¦ state  ¦ xmin  ¦ xmax  ¦ hhu ¦ hot ¦ t_ctid
-------+--------+-------+-------+-----+-----+--------
 (0,1) ¦ normal ¦ 812 c ¦ 813 c ¦ t   ¦     ¦ (0,2)
 (0,2) ¦ normal ¦ 813 c ¦ 814 c ¦ t   ¦ t   ¦ (0,3)
 (0,3) ¦ normal ¦ 814 c ¦ 815   ¦ t   ¦ t   ¦ (0,4)
 (0,4) ¦ normal ¦ 815   ¦ 0 a   ¦     ¦ t   ¦ (0,4)
(4 행s)
```

인덱스는 여전히 하나의 참조를 가지고 있는데, 그 참조는 이 사슬의 첫 부분을 가리키고 있습니다.

```
=> SELECT * FROM index_page('hot_id',1);
 itemoffset ¦  htid ¦ dead
------------+-------+------
          1 ¦ (0,1) ¦ f
 (1 행)
```

만약 수정된 필드가 특정 인덱스의 부분이 아니라면, `HOT` 수정을 실행할 수 있습니다. 그렇지 않을 경우, 일부 인덱스는 체인 중간에 있는 힙 튜플을 가리키게 되는데, 이는 최적화 아이디어와 상충합니다. `HOT` 사슬은 단 한 페이지 내에서만 확장될 수 있으므로, 전체 체인을 살펴볼 때 다른 페이지에 관한 접근이 필요 없으므로, 성능에는 부정적인 영향을 주지 않습니다.

5.3 HOT 수정을 위한 페이지 정리

`HOT` 수정 사슬의 정리는 페이지 정리의 특별한 경우로, 중요한 부분입니다.

위의 예에서 `fillfactor` 임곗값이 이미 초과한 상태이므로, 다음 수정은 페이지 정리를 유발합니다. 그러나 이번에는 페이지에 `HOT` 수정 사슬이 포함되어 있습니다. 이 사슬의 첫 부분은 인덱스에서 항상 위치를 유지해야 하므로 참조되지만, 다른 포인터들은 외부 참조가 없다는 확신이 있으므로 제거될 수 있습니다.

헤드를 움직이지 않기 위해 PostgreSQL은 이중 주소 지정dual addressing을 사용합니다. 이 경우, 인덱스에서 참조되는 포인터(여기서는 (0, 1))는 현재 사슬의 시작 부분을 가리키기 때문에 리디렉션 상태를 받게 됩니다.

```
=> UPDATE hot SET s = 'E';

=> SELECT * FROM heap_page('hot',0);
  ctid ¦ state         ¦ xmin  ¦ xmax ¦ hhu ¦ hot ¦ t_ctid
-------+---------------+-------+------+-----+-----+--------
 (0,1) ¦ redirect to 4 ¦       ¦      ¦     ¦     ¦
 (0,2) ¦ normal        ¦   816 ¦ 0 a  ¦     ¦ t   ¦ (0,2)
 (0,3) ¦ unused        ¦       ¦      ¦     ¦     ¦
 (0,4) ¦ normal        ¦ 815 c ¦ 816  ¦   t ¦ t   ¦ (0,2)
(4 행s)
```

튜플 (0,1), (0,2), (0,3)은 정리 과정을 거쳤습니다. 리디렉션 목적으로 첫 번째 포인터 1은 그대로 두었지만, 포인터 2와 3은 해제되었습니다(사용되지 않는 상태로 변경됨). 이는 인덱스에서 참조되지 않을 것이 확실하기 때문입니다. 새로운 튜플은 해제된 공간에 (0, 2) 튜플로 기록됩니다.

더 많은 수정을 수행해 보겠습니다.

```
=> UPDATE hot SET s = 'F';
=> UPDATE hot SET s = 'G';

=> SELECT * FROM heap_page('hot',0);
  ctid ¦         state ¦  xmin ¦  xmax ¦ hhu ¦ hot ¦ t_ctid
-------+---------------+-------+-------+-----+-----+--------
 (0,1) ¦ redirect to 4 ¦       ¦       ¦     ¦     ¦
 (0,2) ¦ normal        ¦ 816 c ¦ 817 c ¦ t   ¦ t   ¦ (0,3)
 (0,3) ¦ normal        ¦ 817 c ¦ 818   ¦ t   ¦ t   ¦ (0,5)
 (0,4) ¦ normal        ¦ 815 c ¦ 816 c ¦ t   ¦ t   ¦ (0,2)
 (0,5) ¦ normal        ¦ 818   ¦ 0 a   ¦     ¦ t   ¦ (0,5)
(5 행s)
```

다음 수정에서는 페이지 정리가 시작 될 것입니다.

```
=> UPDATE hot SET s = 'H';

=> SELECT * FROM heap_page('hot',0);
  ctid ¦       state    ¦  xmin ¦ xmax ¦ hhu ¦ hot ¦ t_ctid
-------+----------------+-------+------+-----+-----+--------
 (0,1) ¦ redirect to 5 ¦       ¦      ¦     ¦     ¦
 (0,2) ¦ normal        ¦   819 ¦ 0 a  ¦     ¦   t ¦ (0,2)
 (0,3) ¦ unused        ¦       ¦      ¦     ¦     ¦
 (0,4) ¦ unused        ¦       ¦      ¦     ¦     ¦
 (0,5) ¦ normal        ¦ 818 c ¦  819 ¦   t ¦   t ¦ (0,2)
(5 행s)
```

다시 말하자면, 일부 튜플들이 잘려나갈 수 있고, 그로 인해 체인의 머리 부분을 가리키는 포인터가 이동하게 됩니다.

인덱스가 없는 열이 자주 변경된다면, `fillfactor` 값을 줄이는 것이 의미있을 수 있습니다. 이런 방식으로 페이지에 수정를 위한 공간을 사전에 예약할 수 있습니다. 하지만, `fillfactor` 값이 낮아질수록 페이지에 남는 여유 공간이 많아지므로 테이블의 물리적 크기가 커질 수 있다는 점을 잊지 마세요.

5.4 HOT 사슬 분할

페이지에 새로운 튜플을 담을 공간이 없어진다면, 사슬은 분리됩니다. 이럴 때 PostgreSQL은 다른 페이지에 위치한 튜플을 참조하기 위해 별도의 인덱스 항목을 추가해야 합니다.

이런 상황을 직접 확인해보기 위해, 페이지 정리를 막는 스냅숏을 이용하면서 동시에 트랜잭션을 시작해보는 것이 좋을 것입니다.

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT 1;
```

이제 첫 번째 세션에서 일부 수정을 수행하려고 합니다.

```
=> UPDATE hot SET s = 'I';
=> UPDATE hot SET s = 'J';
=> UPDATE hot SET s = 'K';

=> SELECT * FROM heap_page('hot',0);
  ctid ¦         state ¦ xmin  ¦ xmax  ¦ hhu ¦ hot ¦ t_ctid
-------+---------------+-------+-------+-----+-----+--------
 (0,1) ¦ redirect to 2 ¦       ¦       ¦     ¦     ¦
 (0,2) ¦ normal        ¦ 819 c ¦ 820 c ¦ t   ¦ t   ¦ (0,3)
 (0,3) ¦ normal        ¦ 820 c ¦ 821 c ¦ t   ¦ t   ¦ (0,4)
 (0,4) ¦ normal        ¦ 821 c ¦ 822   ¦ t   ¦ t   ¦ (0,5)
 (0,5) ¦ normal        ¦ 822   ¦ 0 a   ¦     ¦ t   ¦ (0,5)
```

```
(5 행s)
```

다음 수정이 일어날 때, 이 페이지는 더 이상 다른 튜플을 받아들일 수 없게 되며, 페이지 정리로도 공간을 확보할 수 없게 될 것입니다.

```
=> UPDATE hot SET s = 'L';


  => COMMIT; -- 스냅숏은 더 이상 필요하지 않습니다.


=> SELECT * FROM heap_page('hot',0);
  ctid ¦         state ¦  xmin ¦  xmax ¦ hhu ¦ hot ¦ t_ctid
-------+---------------+-------+-------+-----+-----+--------
 (0,1) ¦ redirect to 2 ¦       ¦       ¦     ¦     ¦
 (0,2) ¦ normal        ¦ 819 c ¦ 820 c ¦ t   ¦ t   ¦ (0,3)
 (0,3) ¦ normal        ¦ 820 c ¦ 821 c ¦ t   ¦ t   ¦ (0,4)
 (0,4) ¦ normal        ¦ 821 c ¦ 822 c ¦ t   ¦ t   ¦ (0,5)
 (0,5) ¦ normal        ¦ 822 c ¦ 823   ¦     ¦ t   ¦ (1,1)
(5 행s)
```

튜플 (0,5)에는 페이지 1로 이동하는 (1,1) 참조가 포함되어 있습니다.

```
=> SELECT * FROM heap_page('hot',1);
```

```
  ctid ¦  state ¦ xmin ¦ xmax ¦ hhu ¦ hot ¦ t_ctid
-------+--------+------+------+-----+-----+--------
 (1,1) ¦ normal ¦  823 ¦  0 a ¦     ¦     ¦ (1,1)
(1 행)
```

그러나 이 참조는 실제로 사용되지 않습니다. 튜플 (0,5)는 힙 핫 수정 비트가 설정되어 있지 않습니다. 튜플 (1,1)의 경우, 이제 인덱스에 두 개의 항목을 통해 접근할 수 있습니다. 각 항목은 자신만의 핫 사슬의 시작점을 가리키고 있습니다.

```
=> SELECT * FROM index_page('hot_id',1);
 itemoffset ¦  htid ¦ dead
------------+-------+------
          1 ¦ (0,1) ¦ f
          2 ¦ (1,1) ¦ f
(2 행s)
```

5.5 인덱스 페이지 정리

제가 말했듯이, 페이지 정리는 단일 힙 페이지에 한정되어 인덱스에는 영향을 미치지 않습니다. 하지만 인덱스에도 별도의 정리 과정이 있으며[47], 이 경우 인덱스 페이지 하나를 정리하는 작업이 이루어집니다.

[47] postgresql.org/docs/14/btree-implementation.html#BTREE-DELETION

인덱스 정리는 B-트리에 삽입하려는 데이터가 페이지를 두 개로 나누려 할 때 발생합니다. 이는 원래 페이지에 충분한 공간이 없기 때문입니다. 문제는 일부 인덱스 항목이 삭제되어도, 분할된 두 개의 인덱스 페이지가 다시 하나로 병합되지 않는다는 점입니다. 이에 따라 인덱스가 팽창하게 되며, 일단 팽창하면 데이터의 대부분이 삭제되더라도 인덱스 크기를 줄일 수 없게 됩니다. 그러나 정리를 통해 일부 튜플을 제거하면, 페이지 분할은 연기될 수 있습니다.

인덱스에서 정리가 가능한 튜플은 두 가지 유형이 있습니다.

첫 번째로, PostgreSQL은 삭제된 튜플을 정리합니다.[48] 이전에 언급했듯이, PostgreSQL은 인덱스 스캔 중에 스냅숏에서 더 이상 볼 수 없거나 실제로 존재하지 않는 튜플을 가리키는 인덱스 항목을 감지하면 해당 항목을 삭제 표시합니다.

삭제된 튜플에 관한 정보가 없는 경우, PostgreSQL은 한 테이블 행의 다른 버전을 참조하는 인덱스 항목을 확인합니다.[49] `MVCC`로 인해 수정작업은 다수의 행 버전을 생성할 수 있고, 그중 많은 버전은 곧 데이터베이스 범위 외에 사라지게 됩니다. 핫 수정은 이런 효과를 완화하지만, 항상 적용되는 것은 아닙니다. 수정할 열이 인덱스의 일부인 경우, 해당 참조는 모든 인덱스에 전파됩니다. 페이지를 나누기 전에, 아직 삭제 표시가 되지 않았지만 이미 정리할 수 있는 행을 찾는 것이 유의미합니다. 이를 위해 PostgreSQL은 힙 튜플의 가시성을 확인해야 합니다. 이 확인 과정은 테이블 접근이 필요하며, 따라서 `MVCC` 목적으로 기존 튜플의 복사본으로 생성된 가능성 있는 인덱스 튜플에 대해서만 수행됩니다. 페이지를 추가로 나누는 것보다 이런 확인 과정을 거치는 것이 비용 효율적입니다.

[48] backend/access/nbtree/README, Simple deletion section

[49] backend/access/nbtree/README, Bottom-Up deletion section
include/access/tableam.h

6 장 베큠과 자동베큠Vacuum and Autovacuum

6.1 베큠

페이지 정리는 속도가 매우 빠르며, 재활용할 수 있는 공간 중 일부만을 해제하는 작업입니다. 이 작업은 하나의 힙 페이지 내에서 이루어지고 인덱스에는 영향을 미치지 않습니다. 또한, 반대로 인덱스 페이지를 정리할 때는 테이블에 영향을 미치지 않습니다.

`VACUUM` 명령[50]으로 수행되는 주기적인 정리는 주요한 베큠 작업[51]으로, 전체 테이블을 대상으로 합니다. 이 작업은 오래된 힙 튜플과 그에 해당하는 모든 인덱스 항목을 제거합니다.

이 백업 작업은 데이터베이스 시스템의 다른 프로세스와 동시에 병렬로 진행됩니다. 베큠 작업이 진행되는 동안, 테이블과 인덱스는 일반적으로 읽고 쓰는 작업에 사용될 수 있습니다. 단, `CREATE INDEX`, `ALTER INDEX` 등 일부 명령어의 동시 수행은 허용되지 않습니다.

PostgreSQL은 불필요한 페이지 스캔을 방지하기 위해 가시성 맵을 활용합니다. 이 지도에 표시된 페이지는 현재 튜플만을 포함하므로, 이러한 페이지는 베큠 작업에서 무시 됩니다. 만약 페이지가 가시성 맵에 표시되지 않았다면, 그 페이지만이 베큠됩니다. 베큠 작업 후에 페이지에 남는 모든 튜플이 데이터베이스 범위를 초과하면, 가시성 맵은 갱신되어 해당 페이지를 포함하게 됩니다. 또한, 해제된 공간을 반영하여 빈 공간 맵도 수정됩니다.

이제, 인덱스가 포함된 테이블을 생성해 볼 시간입니다.

```
=> CREATE TABLE vac(
  id integer,
  s char(100)
  )
  WITH (autovacuum_enabled = off);

=> CREATE INDEX vac_s ON vac(s);
```

`autovacuum_enabled`는 자동 베큠 기능을 비활성화하는 저장 매개변수입니다. 이는 주로 베큠 시작 시간을 정확히 제어하기 위한 실험적인 용도로 사용됩니다. 이제 한 행을 추가하고 몇 가지 수정을 진행해보도록 하겠습니다.

```
=> INSERT INTO vac(id,s) VALUES (1,'A');

=> UPDATE vac SET s = 'B';

=> UPDATE vac SET s = 'C';
```

[50] postgresql.org/docs/14/sql-vacuum.html
backend/commands/vacuum.c

[51] postgresql.org/docs/14/routine-vacuuming.html

이제 테이블에는 세 개의 튜플이 포함되어 있습니다.

```
=> SELECT * FROM heap_page('vac',0);
  ctid ¦ state  ¦ xmin  ¦ xmax  ¦ hhu ¦ hot ¦ t_ctid
-------+--------+-------+-------+-----+-----+--------
 (0,1) ¦ normal ¦ 826 c ¦ 827 c ¦     ¦     ¦ (0,2)
 (0,2) ¦ normal ¦ 827 c ¦ 828   ¦     ¦     ¦ (0,3)
 (0,3) ¦ normal ¦ 828   ¦ 0 a   ¦     ¦     ¦ (0,3)
 (3 행s)
```

각 튜플은 인덱스에서 참조됩니다.

```
=> SELECT * FROM index_page('vac_s',1);
 itemoffset ¦  htid ¦ dead
------------+-------+------
          1 ¦ (0,1) ¦ f
          2 ¦ (0,2) ¦ f
          3 ¦ (0,3) ¦ f
 (3 행s)
```

베큠 작업을 통해 모든 불필요한 튜플이 제거되어 현재 튜플만 남았습니다:

```
=> VACUUM vac;

=> SELECT * FROM heap_page('vac',0);
  ctid ¦ state  ¦ xmin  ¦ xmax ¦ hhu ¦ hot ¦ t_ctid
-------+--------+-------+------+-----+-----+--------
 (0,1) ¦ unused ¦       ¦      ¦     ¦     ¦
 (0,2) ¦ unused ¦       ¦      ¦     ¦     ¦
 (0,3) ¦ normal ¦ 828 c ¦ 0 a  ¦     ¦     ¦ (0,3)
 (3 행s)
```

페이지 정리에 관해 말하자면, 첫 두 개의 포인터는 더 이상 인덱스 항목에 의해 참조되지 않아 필요 없는 것으로 간주됩니다. 하지만 현재 상황에서는 `unused` 상태 입니다.

```
=> SELECT * FROM index_page('vac_s',1);
 itemoffset ¦  htid ¦ dead
------------+-------+------
          1 ¦ (0,3) ¦ f
(1 행)
```

`unused` 상태의 포인터들은 빈공간으로 처리되어 새로운 행 버전에서 재사용될 수 있습니다. 이제 힙 페이지가 가시성 맵에 표시됩니다. 이를 확인하기 위해 `pg_visibility` 확장 기능을 사용하면 됩니다.

```
=> CREATE EXTENSION pg_visibility;
```

```
=> SELECT all_visible
   FROM pg_visibility_map('vac',0);
 all_visible
-------------
           t
(1 행)
```

페이지 헤더에는 모든 스냅숏에서 해당 페이지의 모든 튜플이 보일 수 있음을 나타내는 속성이 추가되었습니다.

```
=> SELECT flags & 4 > 0 AS all_visible
FROM page_header(get_raw_page('vac',0));
 all_visible
-------------
  t
(1 행)
```

6.2 데이터베이스 수평선 재방문

데이터베이스의 관점에서 보면, 베큠 작업은 사용되지 않는 튜플을 찾아내는 역할을 합니다. 이 개념은 매우 기본적이지만, 그 중요성 때문에 다시 한 번 살펴보는 것이 필요합니다.

그래서 우리는 처음부터 실험을 다시 시작해 보려 합니다.

```
=> TRUNCATE vac;

=> INSERT INTO vac(id,s) VALUES (1,'A');

=> UPDATE vac SET s = 'B';
```

이번에는 데이터베이스의 시점을 유지하기 위해 트랜잭션을 새롭게 열어 행을 수정하려 합니다. 이는 커밋된 읽기 격리 수준에서 실행되는 가상 트랜잭션을 제외하고, 대부분 트랜잭션에 해당합니다. 예를 들어, 이 트랜잭션을 통해 **다른** 테이블의 일부 행들을 수정할 수 있습니다.

```
  => BEGIN;
  => UPDATE accounts SET amount = 0;

=> UPDATE vac SET s = 'C';
```

이제 테이블에는 세 개의 튜플이 있고, 인덱스에도 세 개의 참조가 있습니다. 테이블을 베큠한 후에 변경된 내용을 살펴보려 합니다:

```
=> VACUUM vac;

=> SELECT * FROM heap_page('vac',0);
```

```
 ctid ¦ state  ¦ xmin  ¦ xmax  ¦ hhu ¦ hot ¦ t_ctid
------+--------+-------+-------+-----+-----+--------
 (0,1) ¦ unused ¦       ¦       ¦     ¦     ¦
 (0,2) ¦ normal ¦ 833 c ¦ 835 c ¦     ¦     ¦ (0,3)
 (0,3) ¦ normal ¦ 835 c ¦ 0 a   ¦     ¦     ¦ (0,3)
(3 행s)

=> SELECT * FROM index_page('vac_s',1);
 itemoffset ¦  htid ¦ dead
------------+-------+------
          1 ¦ (0,2) ¦ f
          2 ¦ (0,3) ¦ f
 (2 행s)
```

이전 실행에서는 페이지에 하나의 튜플만 남았었지만, 이번에는 두 개의 튜플이 남아 있습니다. 베큠은 (0,2) 버전을 아직 제거하면 안 된다고 판단했습니다. 이 결정의 근거는 아직 완료되지 않은 트랜잭션에 의해 정의된 데이터베이스의 시점입니다.

```
 => SELECT backend_xmin FROM pg_stat_activity
 WHERE pid = pg_backend_pid();
  backend_xmin
 --------------
           834
 (1 행)
```

우리는 `VERBOSE` 절을 사용해 `VACUUM`을 호출함으로써, 어떤 일이 발생하고 있는지 관찰할 수 있습니다.

```
=> VACUUM VERBOSE vac;
INFO: vacuuming "public.vac"
INFO: table "vac": found 0 removable, 2 nonremovable 행 versions
in 1 out of 1 pages
DETAIL: 1 dead 행 versions cannot be removed yet, oldest xmin: 834
Skipped 0 pages due to buffer pins, 0 frozen pages.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```

출력 결과에는 다음과 같은 정보가 나타나 있습니다:

- 제거 가능한 튜플이 없음을 나타냅니다 (0 `REMOVABLE`).
- 제거하면 안 되는 튜플이 두 개 있음을 보여줍니다 (2 `NONREMOVABLE`).
- 제거하면 안 되는 튜플 중 하나는 더 이상 사용되지 않는 튜플(1 `DEAD`), 다른 하나는 현재 사용 중인 튜플을 나타냅니다.
- `VACUUM`이 참조하는 현재의 시점 (`OLDEST XMIN`)은 활성화된 트랜잭션의 시점을 나타냅니다.

활성 트랜잭션이 완료되면, 데이터베이스의 시점이 앞으로 이동하며, 이에 따라 베큠 작업이 계속 진행될 수

있습니다.

```
  => COMMIT;

=> VACUUM VERBOSE vac;
INFO: vacuuming "public.vac"
INFO: scanned index "vac_s" to remove 1 행 versions
DETAIL: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO: table "vac": removed 1 dead item identifiers in 1 pages
DETAIL: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO: index "vac_s" now contains 1 행 versions in 2 pages
DETAIL: 1 index 행 versions were removed.
0 index pages were newly deleted.
0 index pages are currently deleted, of which 0 are currently
reusable.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO: table "vac": found 1 removable, 1 nonremovable 행 versions
in 1 out of 1 pages
DETAIL: 0 dead 행 versions cannot be removed yet, oldest xmin: 836
Skipped 0 pages due to buffer pins, 0 frozen pages.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```

VACUUM은 새로운 데이터베이스 시점을 넘어서는 사용되지 않는 튜플을 감지하고 제거했습니다. 이제 페이지에는 이전의 행 버전이 없고, 남아 있는 유일한 버전은 현재 버전입니다.

```
=> SELECT * FROM heap_page('vac',0);
  ctid ¦ state  ¦ xmin  ¦ xmax ¦ hhu ¦ hot ¦ t_ctid
-------+--------+-------+------+-----+-----+--------
 (0,1) ¦ unused ¦       ¦      ¦     ¦     ¦
 (0,2) ¦ unused ¦       ¦      ¦     ¦     ¦
 (0,3) ¦ normal ¦ 835 c ¦ 0 a  ¦     ¦     ¦ (0,3)
 (3 행s)
```

인덱스 역시 단 한 개의 항목만을 포함하고 있습니다.

```
=> SELECT * FROM index_page('vac_s',1);
 itemoffset ¦  htid ¦ dead
------------+-------+------
          1 ¦ (0,3) ¦ f
 (1 행)
```

6.3 베큠 단계

베큠 작업의 구조은 매우 단순해 보일 수 있지만, 실제로는 그렇지 않습니다. 결국, 테이블과 인덱스를 동시

에 처리하면서 다른 프로세스를 방해하지 않아야 합니다. 이를 가능하게 하기 위해 각 테이블의 베큠 작업은 여러 단계로 이루어집니다. 베큠 작업은 다음과 같은 과정으로 시작됩니다.[52]

첫 단계에서는 테이블을 스캔하여 사용되지 않는 튜플을 찾습니다. 이런 튜플이 발견되면, 먼저 인덱스에서 제거한 후에 테이블 자체에서 제거합니다. 많은 수의 사용되지 않는 튜플을 한 번에 베큠을 해야 하는 상황이라면, 이 과정이 반복될 수 있습니다. 마지막으로, 테이블의 축소truncation 작업이 진행될 수 있습니다.

테이블Heap 스캔

첫 번째 단계는 테이블 스캔이 진행됩니다.[53] 이 스캔 과정에서 가시성 맵이 고려되는데, 이 맵에 의해 트래킹되는 모든 페이지는 오래된 튜플이 포함되어 있지 않으므로 스캔에서 제외됩니다. 만약 시점을 벗어난 튜플이 더 이상 필요하지 않다면, 해당 튜플의 `ID`는 특별한 `tid` 배열에 추가됩니다. 이 튜플들은 아직 인덱스에서 참조될 수 있으므로 즉시 제거할 수는 없습니다.

이 `tid` 배열은 `VACUUM` 프로세스의 로컬 메모리에 저장됩니다. 할당된 메모리 청크의 크기는 `maintenance_work_mem` 매개변수에 따라 결정됩니다(기본값: 64MB). 이 메모리는 한 번에 전체 청크로 할당되는 것이 아니라 필요에 따라 할당됩니다. 하지만 최악의 경우에도 할당된 메모리는 필요한 양을 초과하지 않으므로, 테이블 크기가 작은 경우 베큠 작업은 이 매개변수에 지정된 메모리보다 더 적은 양의 메모리를 사용할 수 있습니다.

인덱스 베큠

첫 번째 단계는 두 가지 결과를 가져올 수 있습니다: 테이블이 완전히 스캔되는 것이거나, 작업이 완료되기 전에 `tid` 배열에 할당된 메모리가 가득 차는 것입니다. 이러한 상황에 상관없이 인덱스 베큠 작업이 시작됩니다.[54] 이 단계에서는 테이블에 생성된 모든 인덱스를 완전히 스캔하여, `tid` 배열에 등록된 튜플을 참조하는 모든 항목을 찾습니다. 이렇게 찾아낸 항목들은 인덱스 페이지에서 제거됩니다.

> 인덱스는 인덱스 키를 통해 힙 튜플에 빠르게 접근할 수 있지만, 특정 튜플 `ID`에 대응하는 인덱스 항목을 빠르게 찾는 기능은 아직 제공되지 않습니다. 이 기능은 현재 `B`-트리[55]에 관해 구현 중이지만, 아직 완료되지 않았습니다.

`min_parallel_index_scan_size` 값(기본값: 512kB)보다 큰 여러 개의 인덱스가 있으면, 베큠 작업은 백그라운드 워커들이 병렬로 실행되는 방식으로 수행될 수 있습니다. `parallel N` 절을 통해 병렬 처리 수준을 명시적으로 정의하지 않는 한, `VACUUM`은 적절한 인덱스마다 하나의 워커를 실행합니다(백그라운드 워커의 수에 관한 일반적인 제한 안에서[56]). 한 인덱스는 여러 워커들이 동시에 처리하는 것은 불가능합니다.

인덱스 베큠 단계에서 PostgreSQL은 빈 공간 맵을 수정하며, 베큠 작업에 관한 통계를 계산합니다. 그러나 해당 테이블에 사용되지 않는 튜플이 없고, 행이 오직 삽입만 되었으며 삭제나 수정이 이루어지지 않으면 이

---

52 backend/access/heap/vacuumlazy.c, heap_vacuum_rel function

53 backend/access/heap/vacuumlazy.c, lazy_scan_heap function

54 backend/access/heap/vacuumlazy.c, lazy_vacuum_all_indexes function

55 commitfest.postgresql.org/21/1802

56 postgresql.org/docs/14/bgworker.html

단계를 건너뜁니다. 이후 인덱스 스캔은 별도의 **인덱스 정리**[57] 단계의 일부로서, 마지막으로 한 번만 강제적으로 수행됩니다.

인덱스 베큠 단계에서는 인덱스에서 오래된 테이블 튜플에 관한 참조를 제거하지만, 튜플 자체는 테이블에 여전히 남아 있습니다. 이는 완전히 정상적인 상황입니다. 인덱스 스캔은 사용되지 않는 튜플을 찾지 못하며, 테이블의 순차적인 스캔은 가시성 규칙에 따라 필터링하여 사용되지 않는 튜플을 제외합니다.

테이블 베큠

다음으로 테이블 베큠 단계가 시작됩니다.[58] 이 단계에서는 테이블이 다시 스캔 되고, `tid` 배열에 등록된 튜플이 제거되며, 해당 포인터를 해제합니다. 이 시점에서는 모든 관련 인덱스 참조가 이미 제거되었기 때문에, 이 과정을 안전하게 진행할 수 있습니다.

`VACUUM`에 의해 회수된 공간은 빈 공간 맵에 반영됩니다. 그리고 현재 스냅숏에서 볼 수 있는 현재 튜플만을 포함하는 페이지는 가시성 맵에서 태그가 지정됩니다.

테이블 스캔 단계에서 테이블이 완전히 스캔되지 않은 경우, `tid` 배열은 초기화되고 테이블 스캔은 이전에 중단된 위치부터 다시 시작됩니다.

테이블 축소

베큠된 테이블 페이지는 일부 여유 공간을 가지게 되며, 때때로 전체 페이지를 비울 수도 있습니다. 만약 파일의 끝 부분에 여러 개의 빈 페이지가 있다면, 베큠 작업은 이 끝 부분을 제거하고 회수된 공간을 운영 체제에 반환할 수 있습니다. 이 과정은 테이블 축소Heap Truncation라는 마지막 베큠 단계에서 이루어집니다.[59]

테이블 축소는 테이블에 관한 짧은 배타적 잠금Exclusive Lock을 필요로 합니다. 다른 프로세스가 너무 오래 대기하지 않도록 하기 위해, 잠금 획득 시도는 5초를 넘지 않습니다.

테이블이 잠겨야 하기 때문에, 비어있는 끝 부분이 테이블의 최소 `1/16`을 차지하거나 1,000 페이지의 길이에 이를 때만 절단이 수행됩니다. 이 임계값은 하드코딩되어 있어서 사용자가 설정을 변경할 수 없습니다.

이런 모든 예방 조치에도 불구하고 테이블 잠금이 문제를 일으키는 경우, `vacuum_truncate` 및 `toast.vacuum_truncate` 설정 매개변수를 사용하여 축소 기능을 완전히 비활성화할 수 있습니다.

6.4 분석

베큠 작업에 관해 논의하는 동안, 형식적으로는 연결되어 있지 않지만 밀접하게 관련된 또 다른 작업인 **분석**Analysis을 이야기 하겠습니다.[60] 이는 질의 플래너를 위한 통계 정보 수집 또는 집계 작업을 의미합니다. 수집된 통계 정보에는 행 수(`pg_class.reltuples`) 및 페이지 수(`pg_class.relpages`)와 같은 관계relations 내의 정보, 열의 데이터 분포 등 다양한 정보가 포함됩니다.

---

[57] backend/access/heap/vacuumlazy.c, lazy_cleanup_all_indexes function
backend/access/nbtree/nbtree.c, btvacuumcleanup function

[58] backend/access/heap/vacuumlazy.c, lazy_vacuum_heap function

[59] backend/access/heap/vacuumlazy.c, lazy_truncate_heap function

[60] postgresql.org/docs/14/routine-vacuuming.html#VACUUM-FOR-STATISTICS

분석 작업은 `ANALYZE` 명령[61]을 사용하여 수동으로 실행할 수 있으며, `VACUUM ANALYZE`를 호출하여 베큠 작업과 결합할 수도 있습니다. 그러나 이 두 작업은 여전히 차례대로 수행되므로 성능 면에서는 차이가 없습니다.

> 역사적으로, `VACUUM ANALYZE`는 6.1 버전에서 처음 도입되었고, 별도의 `ANALYZE` 명령은 7.2 버전에서 구현되기 전까지는 존재하지 않았습니다. 그 이전 버전에서는 `TCL` 스크립트를 통해 통계 정보가 수집되었습니다.

자동 베큠과 분석은 설정 방식이 비슷하므로, 이 두 가지를 함께 논의하는 것이 맞습니다.

6.5 자동 베큠과 분석

데이터베이스 수평선이 지속해서 늦춰지지 않는다면, 일반적인 베큠 작업으로 충분히 관리할 수 있을 것입니다. 하지만 `VACUUM` 명령은 얼마나 자주 호출해야 할까요?

자주 수정되는 테이블이 너무 드물게 베큠 되면, 원하는 것보다 아주 커질 가능성이 있습니다. 또한, 너무 많은 변경 사항이 누적될 수 있고, 다음 `VACUUM` 실행 때에는 인덱스를 여러 번 통과해야 할 수도 있습니다.

반면에, 테이블이 너무 자주 청소되면 서버가 유용한 작업 대신 유지 보수 작업과 관련한 시간을 더 많이 할애하게 될 수 있습니다.

또한, 일반적인 작업 부하는 시간이 지남에 따라 변화할 수 있으므로, 고정된 베큠 일정으로는 이를 해결하기 어렵습니다. 테이블 수정이 빈번하게 발생하면, 베큠 또한 더 자주 이루어져야 합니다.

이 문제는 자동 베큠autovacuum을 통해 해결할 수 있습니다.[62] 자동 베큠은 테이블 수정의 빈도에 기반하여 베큠과 분석 작업을 실행합니다.

자동 베큠 방법에 관하여

자동 베큠 기능이 활성화되어 있으면(즉, `autovacuum` 설정 매개변수가 켜져 있으면), 자동 베큠 런처가 시스템에서 항상 실행되게 됩니다. 이 프로세스는 자동 베큠 일정을 관리하며, 사용 통계에 기반하여 "활성" 데이터베이스 목록을 유지합니다. 이러한 통계는 `track_counts` 매개변수가 활성화된 경우에만 수집됩니다. 이 매개변수를 꺼두면 자동 베큠 기능이 제대로 작동하지 않을 수 있으므로, 꺼두지 않는 것이 좋습니다.

자동 베큠 런처autovacuum launcher는 `autovacuum_naptime`(기본값: 1분) 간격으로 활성 데이터베이스 목록에 관해 자동 베큠 워커autovacuum worker[63]를 시작합니다. 이러한 워커들은 일반적으로 `postmaster`에 의해 생성됩니다. 따라서 클러스터에 `N`개의 활성 데이터베이스가 있다면, `autovacuum_naptime` 간격 내에 `N`개의 워커가 생성될 것입니다. 그러나 병렬로 실행되는 총 자동 베큠 워커 수는 `autovacuum_max_workers`(기본값: 3) 설정 매개변수로 정의된 임곗값을 넘을 수 없습니다.

---

[61] backend/commands/analyze.c

[62] postgresql.org/docs/14/routine-vacuuming.html#AUTOVACUUM

[63] backend/postmaster/autovacuum.c

> 자동 베큠 워커는 일반적인 백그라운드 워커와 매우 유사하나, 작업 관리의 일반적인 방법보다 훨씬 이전에 등장했습니다. 자동 베큠 구현을 변경하지 않기로 한 결정으로 인해, 자동 베큠 워커는 `max_worker_processes` 슬롯을 사용하지 않게 되었습니다.

백그라운드 워커가 시작되면 지정된 데이터베이스에 연결하고, 다음 두 가지 목록을 작성합니다:

- 베큠할 모든 테이블, 실체화된 뷰 및 `TOAST` 테이블의 목록
- 분석할 모든 테이블과 실체화된 뷰의 목록 (`TOAST` 테이블은 항상 인덱스를 통해 접근되기 때문에 분석되지 않습니다.)

선택된 객체는 하나씩 베큠되거나 분석되며(또는 두 가지 작업을 모두 수행합니다), 그 작업이 완료되면 워커는 종료됩니다.

자동 베큠은 `VACUUM` 명령으로 시작되는 수동 베큠과 유사하게 작동하지만, 몇 가지 미묘한 차이가 있습니다:

- 수동 베큠은 `maintenance_work_mem` 크기의 메모리 청크에 튜플 `ID`를 누적합니다. 그러나 자동 베큠에서 동일한 제한을 사용하는 것은 바람직하지 않습니다. 메모리 소비가 과도하게 될 수 있기 때문입니다. 여러 자동 베큠 워커가 병렬로 실행될 수 있으며, 각각이 한 번에 `maintenance_work_mem`의 메모리를 소비할 것입니다. 그 대신, PostgreSQL은 베큠 프로세스에 관해 별도의 메모리 제한을 제공하며, 이는 `autovacuum_work_mem` 매개변수로 정의됩니다. 기본적 `autovacuum_work_mem`(기본값: -1) 매개변수는 일반적인 `maintenance_work_mem` 제한을 따릅니다. 따라서 `autovacuum_max_workers` 값이 크다면, `autovacuum_work_mem` 값을 해당 값에 맞게 조정해야 할 수 있습니다.
- 한 테이블에 생성된 여러 인덱스의 동시 처리는 수동 베큠으로만 수행할 수 있습니다. 이를 위해 자동 베큠을 사용하면 많은 수의 병렬 프로세스가 생성되어서는 안 됩니다.

자동 베큠 워커가 `autovacuum_naptime` 간격 내에 예정된 모든 작업을 완료하지 못한 경우, 자동 베큠 런처는 해당 데이터베이스에서 병렬로 실행될 추가 워커를 생성합니다. 이 두 번째 워커는 독립적으로 베큠과 분석 대상 객체 목록을 작성하고 처리를 시작합니다. 단, 테이블 수준에서는 병렬 처리가 이루어지지 않으며, 서로 다른 테이블만 동시에 처리될 수 있습니다.

어떤 테이블을 베큠 해야 하나요?

테이블 수준에서 자동 베큠을 비활성화하는 것이 가능합니다. 그러나 이를 필요로 하는 경우는 매우 드뭅니다. 이를 위해 두 가지 저장소 매개변수가 제공되는데, 하나는 일반 테이블에 관한 것이고 다른 하나는 `TOAST` 테이블에 관한 것입니다:

- `autovacuum_enabled`
- `toast.autovacuum_enabled`

일반적으로 자동 베큠은 죽은 튜플의 누적이나 새로운 행의 삽입으로 살행됩니다.

죽은 튜플 누적. 죽은 튜플은 통계 수집기에 의해 지속해서 세어지며, 현재의 수는 시스템 카탈로그 테이블인 `pg_stat_all_tables`에서 확인할 수 있습니다.

죽은 튜플을 청소해야 하는지 여부는 다음 두 가지 매개변수에서 정의한 임계값을 초과하면 판단됩니다:

- `autovacuum_vacuum_threshold`(기본값: 50)는 죽은 튜플의 수를 나타냅니다(절대값).
- `autovacuum_vacuum_scale_factor`(기본값: 0.2)는 테이블의 죽은 튜플 비율을 설정합니다.

다음 조건이 충족되면 청소가 필요하다고 판단됩니다:

```
pg_stat_all_tables.n_dead_tup > autovacuum_vacuum_threshold +
(autovacuum_vacuum_scale_factor × pg_class.reltuples)
```

주의 깊게 봐야 할 주요 매개변수는 `autovacuum_vacuum_scale_factor`입니다. 이 값은 대형 테이블에 중요하며(대형 테이블이 대부분의 문제를 일으킬 가능성이 높음), 기본값인 20%는 너무 큰 값일 수 있어서, 필요에 따라 크게 줄일 수도 있습니다.

다른 테이블에 대해서는 최적의 매개변수 값이 달라질 수 있습니다. 이는 주로 테이블의 크기와 작업 부하 유형에 따라 변하게 됩니다. 적절한 초기값을 설정한 후, 저장소 매개변수를 사용하여 특정 테이블에 관한 설정을 재정의하는 것이 바람직합니다.

- `autovacuum_vacuum_threshold`와 `toast.autovacuum_vacuum_threshold`
- `autovacuum_vacuum_scale_factor`와 `toast.autovacuum_vacuum_scale_factor`

이런 식으로 특정 테이블에 관한 매개변수 값을 조정할 수 있습니다.

행 삽입. 만약 어떤 테이블에서 행이 오직 삽입되며, 삭제나 수정이 일어나지 않는다면, 해당 테이블에는 '죽은 튜플'이 존재하지 않을 것입니다. 그러나 이런 테이블에서도 힙 튜플을 미리 고정하고 가시성 맵을 tnwjd 하기 위해 베큠 연산을 실행해야 합니다. 이 과정을 통해 인덱스만 스캔할 수 있는 상태를 유지합니다.

이전의 베큠 작업 이후 삽입된 행의 수가 아래 두 매개변수로 정의된 임계값을 넘으면, 테이블에 대해 메큠이 수행됩니다:

- `autovacuum_vacuum_insert_threshold` (기본값: 1000)
- `autovacuum_vacuum_insert_scale_factor` (기본값: 0.2)

이는 다음 공식으로 표현됩니다:

```
pg_stat_all_tables.n_ins_since_vacuum > autovacuum_vacuum_insert_threshold +
autovacuum_vacuum_insert_scale_factor × pg_class.reltuples
```

이전과 같이, 테이블 수준에서 이 매개변수들을 재정의할 수 있습니다. 저장 매개변수를 사용하는 경우, `autovacuum_vacuum_insert_threshold`와 그에 해당하는 `TOAST` 매개변수, 그리고 `autovacuum_vacuum_insert_scale_factor`와 그에 해당하는 `TOAST` 매개변수를 사용하게 됩니다.

어떤 테이블을 분석 해야 하나요?

자동 분석은 변경된 행만 다루기 때문에, 자동 베큠 보다 계산이 약간 더 단순합니다.

다음 두 가지 설정 매개변수에 의해 정의된 임계값을 넘는 경우, 이전 분석 이후에 수정된 행의 수가 테이블을 분석해야 한다고 가정합니다:

- `autovacuum_analyze_threshold` (기본값: 50)
- `autovacuum_analyze_scale_factor` (기본값: 0.1)

자동 분석은 아래의 조건을 만족할 때 실행됩니다:

`pg_stat_all_tables.n_mod_since_analyze > autovacuum_analyze_threshold + autovacuum_analyze_scale_factor × pg_class.reltuples`

특정 테이블에 대해 자동 분석 설정을 재정의하려면, `autovacuum_analyze_threshold`와 `autovacuum_analyze_scale_factor`라는 동일한 이름의 저장 매개변수를 사용할 수 있습니다.
`TOAST` 테이블은 분석되지 않기 때문에, 이에 해당하는 매개변수는 존재하지 않습니다.

자동 베큠 작동

이 절에서 설명한 내용을 정리하고, 현재 베큠과 분석이 필요한 테이블을 보여주는 두 개의 뷰를 만들어 보겠습니다.[64] 이 뷰들에서 사용되는 함수는 전달된 매개변수의 현재 값을 반환합니다. 이때, 이 값은 테이블 수준에서 재정의될 수 있다는 점을 고려합니다.

```
=> CREATE FUNCTION p(param text, c pg_class) RETURNS float
AS $$
  SELECT coalesce(
    -- use storage parameter if set
    (SELECT option_value
       FROM pg_options_to_table(c.reloptions)
      WHERE option_name = CASE
               -- for TOAST tables the parameter name is different
               WHEN c.relkind = 't' THEN 'toast.' ELSE ''
              END || param
  ),
  -- else take the configuration parameter value
  current_setting(param)
)::float;
$$ LANGUAGE sql;
```

베큠관련 뷰는 아래와 같습니다:

```
=> CREATE VIEW need_vacuum AS
WITH c AS (
    SELECT c.oid,
           greatest(c.reltuples, 0) reltuples,
           p('autovacuum_vacuum_threshold', c) threshold,
           p('autovacuum_vacuum_scale_factor', c) scale_factor,
```

[64] backend/postmaster/autovacuum.c, relation_needs_vacanalyze function

```
            p('autovacuum_vacuum_insert_threshold', c) ins_threshold,
            p('autovacuum_vacuum_insert_scale_factor', c) ins_scale_factor
    FROM pg_class c
    WHERE c.relkind IN ('r','m','t')
)
SELECT st.schemaname ¦¦ '.' ¦¦ st.relname AS tablename,
        st.n_dead_tup AS dead_tup,
        c.threshold + c.scale_factor * c.reltuples AS max_dead_tup,
        st.n_ins_since_vacuum AS ins_tup,
        c.ins_threshold + c.ins_scale_factor * c.reltuples AS max_ins_tup,
        st.last_autovacuum
FROM pg_stat_all_tables st
JOIN c ON c.oid = st.relid;
```

`max_dead_tup` 열은 자동 베큠을 실행할 때의 '죽은 튜플'의 최대 허용 수를 나타냅니다. 반면, `max_ins_tup` 열은 삽입과 관련된 임곗값을 표현합니다.

이어서, 분석에 관한 유사한 뷰에 대해 설명하겠습니다:

```
=> CREATE VIEW need_analyze AS
WITH c AS (
    SELECT c.oid,
            greatest(c.reltuples, 0) reltuples,
            p('autovacuum_analyze_threshold', c) threshold,
            p('autovacuum_analyze_scale_factor', c) scale_factor
    FROM pg_class c
    WHERE c.relkind IN ('r','m')
)
SELECT st.schemaname ¦¦ '.' ¦¦ st.relname AS tablename,
        st.n_mod_since_analyze AS mod_tup,
        c.threshold + c.scale_factor * c.reltuples AS max_mod_tup,
        st.last_autoanalyze
FROM pg_stat_all_tables st
JOIN c ON c.oid = st.relid;
```

`max_mod_tup` 열은 자동 분석을 수행할 때의 임곗값을 나타냅니다.

실험을 빠르게 하려고 초마다 자동 바큠 작업을 시작하도록 설정하겠습니다.

```
=> ALTER SYSTEM SET autovacuum_naptime = '1s';
=> SELECT pg_reload_conf();
```

`vac` 테이블을 초기화하고, 그 다음에 1,000개의 행을 삽입해 보겠습니다. 이 과정에서 테이블 수준에서 자동 베큠 기능이 비활성화되어 있음을 유의해 주세요.

```
=> TRUNCATE TABLE vac;
```

```
=> INSERT INTO vac(id,s)
SELECT id, 'A' FROM generate_series(1,1000) id;
```

베큠 관련 뷰는 다음과 같습니다:

```
=> SELECT * FROM need_vacuum WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]---+-----------
tablename       ¦ public.vac
dead_tup        ¦ 0
max_dead_tup    ¦ 50
ins_tup         ¦ 1000
max_ins_tup     ¦ 1000
last_autovacuum ¦
```

실제로 `max_dead_tup`의 임곗값은 50입니다. 위에서 언급한 공식에 따르면 50 + 0.2 × 1000 = 250이 되어야 하지만, `INSERT` 명령은 통계 정보를 수정하지 않아, 이 테이블에 관한 통계 정보는 아직 사용할 수 없습니다.

```
=> SELECT reltuples FROM pg_class WHERE relname = 'vac';
 reltuples
-----------
        -1
```

`pg_class.reltuples` 값은 -1로 설정되어 있습니다. 이 특별한 값은 통계 정보가 없는 테이블과 실제로 비어 있지만 이미 분석이 완료된 테이블을 구분하기 위해 0 대신 사용됩니다. 계산에 사용될 때, 이 음수 값은 0으로 간주되어 50 + 0.2 × 0 = 50이 됩니다.

또한, 예상되는 값이 1,200임에도 불구하고 `max_ins_tup` 값은 1,000입니다. 이 역시 동일한 이유로 발생하는 현상입니다.

이제 분석 뷰에 대해 살펴보겠습니다.

```
=> SELECT * FROM need_analyze WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]----+-----------
       tablename ¦ public.vac
         mod_tup ¦ 1006
     max_mod_tup ¦ 50
last_autoanalyze ¦
```

우리는 1,000개의 행을 수정했고, 이는 사실상 삽입을 의미합니다. 이에 따라 임곗값이 초과하였습니다: 테이블의 크기는 아직 알려지지 않았기 때문에 현재 임곗값은 50으로 설정되어 있습니다. 이는 만약 아래 처럼 설정을 하면 자동 분석이 곧바로 실행될 것임을 의미합니다.

```
=> ALTER TABLE vac SET (autovacuum_enabled = on);
```

테이블 분석이 완료되면, 임곗값은 적절한 값인 150으로 재조정됩니다.

```
=> SELECT reltuples FROM pg_class WHERE relname = 'vac';
 reltuples
-----------
       1000
(1 행)

=> SELECT * FROM need_analyze WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]----+------------------------------
tablename        ¦ public.vac
mod_tup          ¦ 0
max_mod_tup      ¦ 150
last_autoanalyze ¦ 2023-03-06 14:00:45.533464+03
```

그럼 다시 자동 베큠에 관해 이야기해 보겠습니다:

```
=> SELECT * FROM need_vacuum WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]---+-----------
tablename       ¦ public.vac
dead_tup        ¦ 0
max_dead_tup    ¦ 250
ins_tup         ¦ 1000
max_ins_tup     ¦ 1200
last_autovacuum ¦
```

실제 분석을 통해 파악된 테이블 크기에 기반하여 `max_dead_tup`과 `max_ins_tup` 값도 수정되었습니다.

베큠은 다음의 어느 한 조건이라도 만족할 경우 시작됩니다:

- 죽은 튜플이 250개 이상 쌓일 경우
- 테이블에 행이 200개 이상 삽입될 경우

다시 한번 자동 베큠을 비활성화하고, 임계값을 1개 초과하는 총 251개의 행을 수정해 보겠습니다.

```
=> ALTER TABLE vac SET (autovacuum_enabled = off);
=> UPDATE vac SET s = 'B' WHERE id <= 251;
=> SELECT * FROM need_vacuum WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]---+-----------
tablename       ¦ public.vac
dead_tup        ¦ 251
max_dead_tup    ¦ 250
ins_tup         ¦ 1000
max_ins_tup     ¦ 1200
last_autovacuum ¦
```

이제 트리거 조건이 충족되었습니다. 자동 베큠을 활성화해 보겠습니다. 잠시 후에 테이블이 처리되고 사용 통계가 재설정된 상태를 확인하실 수 있을 것입니다:

```
=> ALTER TABLE vac SET (autovacuum_enabled = on);
=> SELECT * FROM need_vacuum WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]---+------------------------------
      tablename ¦ public.vac
       dead_tup ¦ 0
   max_dead_tup ¦ 250
        ins_tup ¦ 0
    max_ins_tup ¦ 1200
last_autovacuum ¦ 2023-03-06 14:00:51.736815+03
```

6.6 부하 관리

페이지 수준에서 작동하는 베큠은 다른 프로세스를 차단하진 않지만, 시스템 부하를 증가시키고 성능에 상당한 영향을 미칠 수 있습니다.

베큠 조절

베큠의 강도를 조절하기 위해 PostgreSQL은 테이블 처리 도중에 정기적으로 일시 중단합니다. 작업 단위로 대략 `vacuum_cost_limit`(기본값: 200)만큼 작업을 완료한 후, 프로세스는 `vacuum_cost_delay`(기본값: 0) 시간 동안 잠자기 상태로 전환됩니다.

`vacuum_cost_delay`의 기본값인 0은 일반적인 베큠 작업이 실질적으로 절대로 잠들지 않는다는 것을 의미합니다. 따라서, `vacuum_cost_limit`의 정확한 값은 큰 차이를 만들지 않습니다. 수동 베큠을 수행해야 할 경우, 관리자는 가능한 빠르게 완료를 원할 것으로 가정됩니다.

만약 sleep 시간이 설정되어 있다면, 프로세스는 버퍼 캐시에서 페이지를 처리하는데 `vacuum_cost_limit` 단위의 작업을 소비한 후마다 일시 중단됩니다. 각 페이지 읽기 비용은 버퍼 캐시에서 페이지를 찾은 경우 `vacuum_cost_page_hit`(기본값: 1) 단위로 계산되며, 그렇지 않은 경우에는 `vacuum_cost_page_miss` 단위(기본값: 2)로 계산됩니다.[65] 만약 깨끗한 페이지가 베큠 작업에 의해 더러워진다면, 추가로 `vacuum_cost_page_dirty`(기본값: 20) 단위가 더해집니다.[66]

`vacuum_cost_limit` 매개변수의 기본값을 유지한다면, 베큠은 최선의 경우(모든 페이지가 캐시되고 베큠에 의해 페이지가 변경되지 않는 경우)에는 한 번의 주기에서 최대 200개 페이지를 처리할 수 있습니다. 반면, 최악의 경우(모든 페이지가 디스크에서 읽혀지고 변경되는 경우)에는 한 번의 주기에서 최대 9개 페이지만 처리할 수 있습니다.

자동 베큠 조절

자동 베큠[67]의 조절은 베큠의 조절과 매우 유사하지만, 자동 베큠은 자체적인 매개변수 집합을 가지고 있어 다른 강도로 실행될 수 있습니다:

- `autovacuum_vacuum_cost_limit(`기본값: -1)

[65] backend/storage/buffer/bufmgr.c, ReadBuffer_common function

[66] backend/storage/buffer/bufmgr.c, MarkBufferDirty function

[67] backend/postmaster/autovacuum.c, autovac_balance_cost function

- `autovacuum_vacuum_cost_delay`(기본값: 2ms)

이 매개변수들 중 하나가 -1로 설정되면, 해당하는 일반 베큠의 매개변수로 대체됩니다. 따라서 `autovacuum_vacuum_cost_limit` 매개변수는 기본적으로 `vacuum_cost_limit` 값에 의존하게 됩니다.

> 버전 12 이전에는 `autovacuum_vacuum_cost_delay`의 기본값이 20ms 였는데, 이는 현대 하드웨어에서 매우 낮은 성능을 초래했습니다.

자동 베큠의 작업 단위는 주기당 `autovacuum_vacuum_cost_limit`로 제한되며, 이 값은 모든 워커 간에 공유되므로, 워커의 수와 상관없이 시스템 전체에 대해 거의 동일한 영향을 미칩니다. 따라서 자동 베큠을 가속하려면, `autovacuum_max_workers`와 `autovacuum_vacuum_cost_limit` 값을 비례적으로 증가시켜야 합니다.

필요한 경우, 다음의 저장 매개변수를 설정하여 특정 테이블에 대해 이러한 설정을 재정의할 수 있습니다:
- `autovacuum_vacuum_cost_delay`와 `toast.autovacuum_vacuum_cost_delay`
- `autovacuum_vacuum_cost_limit`와 `toast.autovacuum_vacuum_cost_limit`

6.7 모니터링

베큠이 모니터링되는 경우, 참조가 `maintenance_work_mem` 메모리 청크에 맞지 않아 죽은 튜플을 한 번에 제거할 수 없는 상황을 감지할 수 있습니다. 이 경우 모든 인덱스를 여러 번에 걸쳐 완전히 스캔해야 합니다. 큰 테이블의 경우 이 과정은 상당한 시간이 소요되며, 시스템에 부하를 일으킬 수 있습니다. 질의는 차단되지 않지만, 추가적인 `I/O` 작업은 시스템 처리량을 심각하게 제한할 수 있습니다.

이러한 문제는 테이블을 더 자주 베큠하거나(각 실행에서 적은 튜플을 정리하는 방식), 더 많은 메모리를 할당하는 방법으로 해결할 수 있습니다.

베큠 모니터링

`VERBOSE` 절과 함께 실행되는 베큠 명령은 정리 작업을 수행하고 상태 보고서를 표시합니다. 또한, `pg_stat_progress_vacuum` 뷰는 시작된 프로세스의 현재 상태를 보여줍니다.

분석에 관한 유사한 뷰인 `pg_stat_progress_analyze`도 있지만, 일반적으로 이 작업은 매우 빠르게 수행되며 문제를 일으키는 경우는 거의 없습니다.

테이블에 더 많은 행을 삽입하고 모두 수정하여 베큠이 상당한 시간 동안 실행되도록 해보겠습니다.

```
=> TRUNCATE vac;
=> INSERT INTO vac(id,s)
SELECT id, 'A' FROM generate_series(1,500000) id;
=> UPDATE vac SET s = 'B';
```

이 데모를 위해 `tid` 배열에 할당되는 메모리 양을 `1MB`로 제한하겠습니다.

```
=> ALTER SYSTEM SET maintenance_work_mem = '1MB';
=> SELECT pg_reload_conf();
```

베큠 명령을 실행하고, 실행 중에 여러 번 `pg_stat_progress_vacuum` 뷰를 조회해 보겠습니다.

```
=> VACUUM VERBOSE vac;
```

```
=> SELECT * FROM pg_stat_progress_vacuum \gx
-[ RECORD 1 ]------+------------------
               pid ¦ 14531
             datid ¦ 16391
           datname ¦ internals
             relid ¦ 16479
             phase ¦ vacuuming indexes
   heap_blks_total ¦ 17242
 heap_blks_scanned ¦ 3009
heap_blks_vacuumed ¦ 0
index_vacuum_count ¦ 0
   max_dead_tuples ¦ 174761
   num_dead_tuples ¦ 174522
=> SELECT * FROM pg_stat_progress_vacuum \gx
-[ RECORD 1 ]------+------------------
               pid ¦ 14531
             datid ¦ 16391
           datname ¦ internals
             relid ¦ 16479
             phase ¦ vacuuming indexes
   heap_blks_total ¦ 17242
 heap_blks_scanned ¦ 17242
heap_blks_vacuumed ¦ 6017
index_vacuum_count ¦ 2
   max_dead_tuples ¦ 174761
   num_dead_tuples ¦ 150956
```

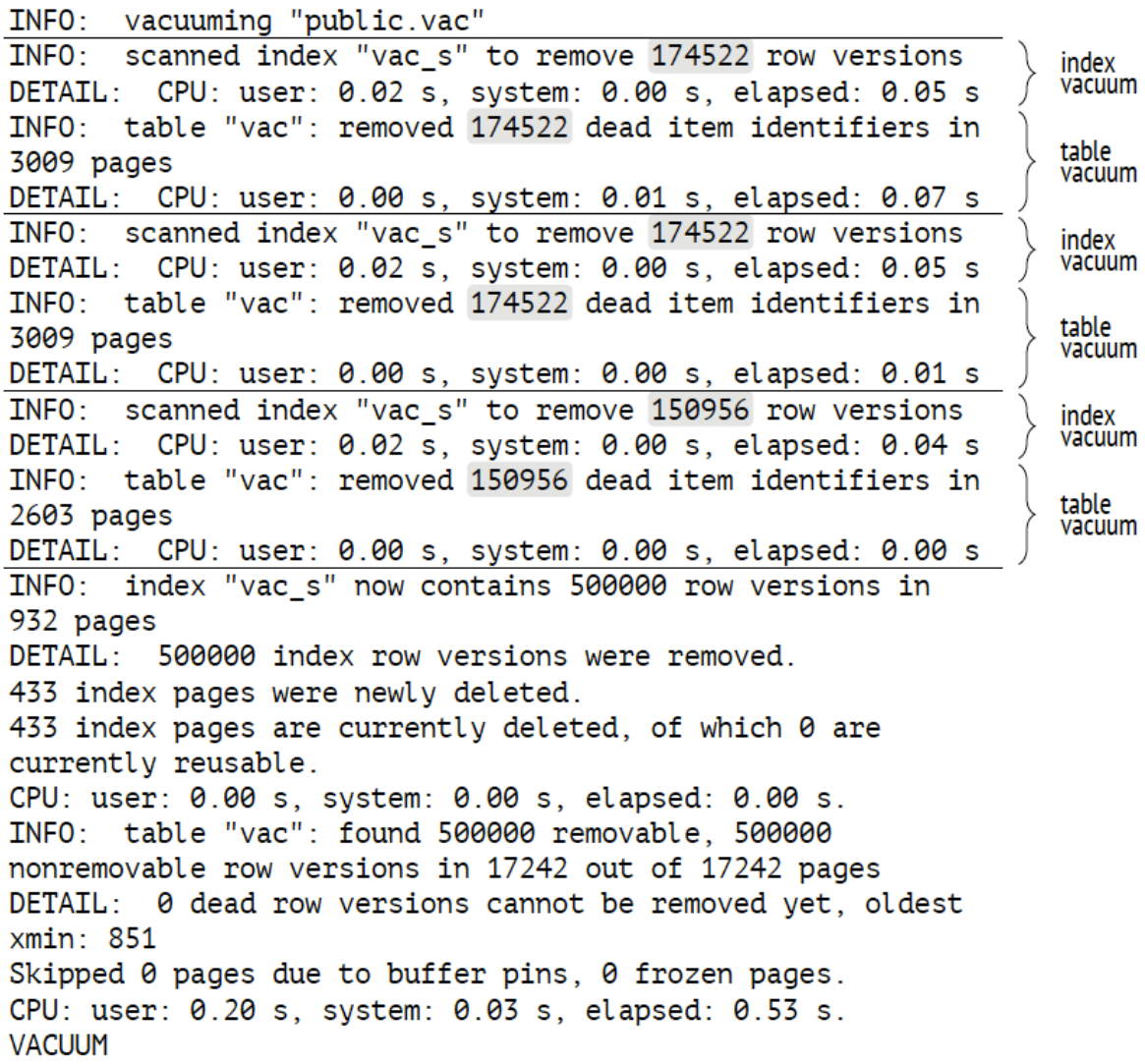
특히, 이 뷰는 다음을 보여줍니다:

- phase - 현재 베큠 단계의 이름 (주요 단계를 설명했지만 실제로는 더 많은 단계가 있습니다.[68])
- heap_blks_total - 테이블의 전체 페이지 수
- heap_blks_scanned - 스캔된 페이지 수
- heap_blks_vacuumed - 베큠된 페이지 수
- index_vacuum_count - 인덱스 스캔의 수

[68] postgresql.org/docs/14/progress-reporting.html#VACUUM-PHASES

전체 베큠 진행 상황은 `heap_blks_vacuumed` 값을 `heap_blks_total` 값으로 나눈 비율로 정의됩니다. 그러나 이 비율이 인덱스 스캔으로 인해 갑작스럽게 변할 수 있으므로 이 점을 염두에 두어야 합니다. 사실, 한 번에 충분한 메모리가 베큠 완료를 위해 할당되지 않았다면, 베큠 주기의 수에 주목하는 것이 더 중요합니다. 주기의 값이 1보다 크다면, 한 번에 베큠을 완료하는 데 충분한 메모리가 없었음을 의미합니다.

이미 완료된 `VACUUM VERBOSE` 명령의 출력에서는 전체적인 상황을 볼 수 있습니다.



총 3번의 인덱스 스캔이 있었습니다. 각 스캔은 최대 174,522개의 죽은 튜플 포인터를 제거했습니다. 이 값은 `maintenance_work_mem` 크기의 배열에 들어갈 수 있는 tid 포인터(각각은 6바이트를 차지)의 수에 의해 정의됩니다. 최대 크기는 `pg_stat_progress_vacuum.max_dead_tuples`에 표시되지만, 실제로 사용되는 공간은 항상 약간 작습니다. 이는 다음 페이지가 읽힐 때, 그 페이지에 있는 죽은 튜플의 모든 포인터가 남은 메모리에 맞게 보장됨을 의미합니다. 이 페이지에 있는 죽은 튜플의 수가 얼마나 많든지 상관없습니다.

자동 베큠 모니터링

자동 VACUUM을 모니터링하는 주요 방법 중 하나는 상태 정보를 서버 로그에 기록하여 추가 분석을 위해 사용하는 것입니다. 이 상태 정보는 `VACUUM VERBOSE` 명령의 출력과 유사합니다.

log_autovacuum_min_duration(기본값: -1) 매개변수가 0으로 설정된 경우, 모든 자동 베큠 작업이 로그에 기록됩니다.

```
=> ALTER SYSTEM SET log_autovacuum_min_duration = 0;
=> SELECT pg_reload_conf();
=> UPDATE vac SET s = 'C';
UPDATE 500000

postgres$ tail -n 13 /home/postgres/logfile
2023-03-06 14:01:13.727 MSK [17351] LOG: automatic vacuum of table
"internals.public.vac": index scans: 3
pages: 0 removed, 17242 remain, 0 skipped due to pins, 0 skipped frozen
tuples: 500000 removed, 500000 remain, 0 are dead but not
yet removable, oldest xmin: 853
index scan needed: 8622 pages from table (50.01% of total)
had 500000 dead item identifiers removed
index "vac_s": pages: 1428 in total, 496 newly deleted, 929
currently deleted, 433 reusable
avg read rate: 12.404 MB/s, avg write rate: 14.810 MB/s
buffer usage: 46038 hits, 5670 misses, 6770 dirtied
WAL usage: 40390 records, 15062 full page images, 89188595 bytes
system usage: CPU: user: 0.31 s, system: 0.33 s, elapsed: 3.57 s
2023-03-06 14:01:14.117 MSK [17351] LOG: automatic analyze of table
"internals.public.vac"
avg read rate: 41.081 MB/s, avg write rate: 0.020 MB/s
buffer usage: 15355 hits, 2035 misses, 1 dirtied
system usage: CPU: user: 0.14 s, system: 0.00 s, elapsed: 0.38 s
```

베큠 및 분석을 수행해야 하는 테이블 목록을 추적하기 위해 이전에 검토했던 need_vacuum과 need_analyze 뷰를 사용할 수 있습니다. 이 목록이 증가한다면, 자동 베큐이 부하를 처리하지 못하고 있음을 의미하며, 이럴 경우 간격(autovacuum_vacuum_cost_delay)을 줄이거나 간격 사이에서 수행하는 작업량(autovacuum_vacuum_cost_limit)을 증가시켜 속도를 높여야 할 수 있습니다. 또한 병렬 처리의 정도도 증가시켜야 할 수 있습니다(autovacuum_max_workers).

7 장 프리징Freezing

7.1 트랜잭션 ID 랩어라운드Wraparound

PostgreSQL에서는 트랜잭션 ID가 32비트를 사용합니다. 40억이라는 숫자는 꽤 크게 느껴질 수 있지만, 시스템이 활발하게 동작하면 이 숫자도 금방 소진될 수 있습니다. 예시로, 초당 평균 1,000개의 트랜잭션가 발생한다고 가정하면, 연속적인 작업으로 이어지면 대략 6주 후에 모두 소진됩니다.

일련번호가 다 사용된 경우, 다음 순서를 위해 카운터를 초기화해야 합니다. 이를 랩어라운드라고 부릅니다. 하지만, 작은 ID를 가진 트랜잭션는 항상 증가하는 숫자로 배정되어야만, 큰 ID를 가진 트랜잭션보다 오래된 트랜잭션로 인식될 수 있습니다. 그래서 카운터는 초기화된 후에도 같은 번호를 재사용할 수 없습니다.

PostgreSQL이 트랜잭션 ID에 64비트를 할당하지 않은 이유는 각 튜플 헤더가 두 개의 트랜잭션 ID(xmin과 xmax)를 저장해야 하기 때문입니다. 이미 헤더의 크기가 상당히 크며(데이터 정렬을 고려하면 최소 24바이트), 64비트를 추가하면 추가로 8바이트가 더 필요해집니다. 이렇게 하면 메모리 사용량이 증가하고, 그 결과 전체 시스템의 효율성이 떨어질 수 있기 때문입니다.

> PostgreSQL 은 일반적인 ID 를 확장하기 위해 32 비트 epoch 와 결합된 64 비트 트랜잭션 ID[69]를 사용하지만, 이는 내부적으로만 활용하며 데이터 페이지에는 절대로 포함되지 않습니다. 이는 메모리 사용량을 최소화하면서도 트랜잭션 ID 의 고갈 문제를 해결하는 방법입니다.

랩어라운드를 적절히 처리하기 위해 PostgreSQL은 트랜잭션 ID 자체보다는 트랜잭션의 나이를 비교합니다. 여기서 트랜잭션의 나이란, 해당 트랜잭션 시작 이후에 발생한 후속 트랜잭션의 수를 의미합니다. 따라서 작다 또는 크다는 개념 대신 오래된(먼저 시작된) 트랜잭션과 젊은(나중에 시작된) 트랜잭션이라는 개념을 사용합니다. 이렇게 함으로써 트랜잭션의 순서를 더 정확하게 파악하고 관리할 수 있습니다.

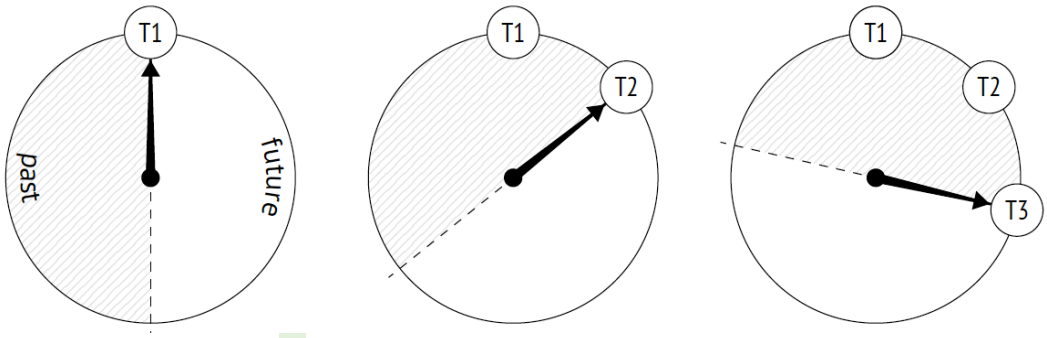
이 비교는 코드 내에서 32비트 산술을 통해 실행됩니다. 먼저, 두 32비트 트랜잭션 ID 사이의 차이를 계산하고, 그 결과를 0과 비교합니다.[70]

이 개념을 더 잘 이해하기 위해 트랜잭션 ID 순서를 시계 바늘로 생각해보실 수 있습니다. 각 트랜잭션마다 시계 방향으로 절반의 원은 미래에 속하고, 반대쪽 절반은 과거에 속한다고 상상해보세요. 이렇게 생각하면 트랜잭션의 나이에 관한 개념을 더 쉽게 이해할 수 있습니다.

---

[69] include/access/transam.h, FullTransactionId type

[70] backend/access/transam/transam.c, TransactionIdPrecedes function



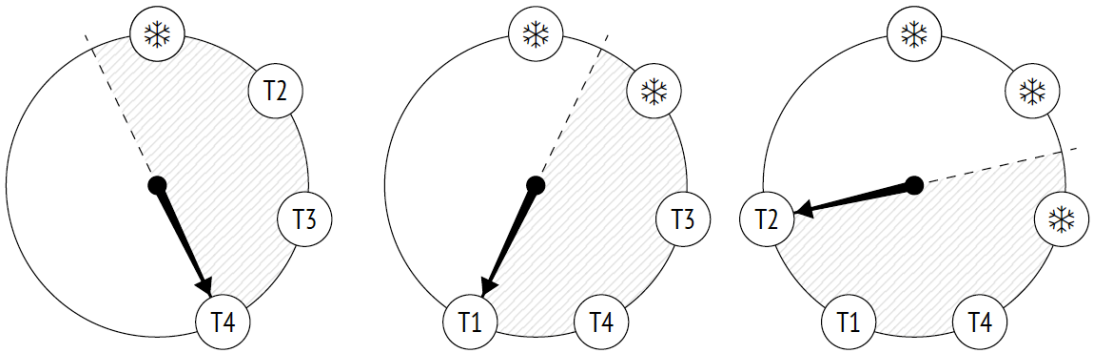
이 시계 바늘로 트랜잭션 `ID`를 상상하는 방법에는 주의해야 할 점이 있습니다. 오래된 트랜잭션(`T1`)는 최근 트랜잭션에 비해 원의 과거 부분에 위치합니다. 그런데 만약 빠르게 새로운 트랜잭션이 발생하여 원의 미래 부분에서 `T1`을 바라본다면, 이는 문제를 일으킬 수 있습니다. 왜냐하면, 그런 상황에서 발생하는 모든 새로운 거래는 `T1`에 의해 이루어진 변경 사항을 인식할 수 없게 될 것이기 때문입니다. 이는 시스템에 파괴적인 영향을 끼칠 수 있습니다.

7.2 튜플 프리징 및 가시성 규칙

시간 여행 문제를 방지하기 위해, 베큠은 페이지 정리 작업 외에도 추가적인 작업을 수행합니다.[71] 이 작업은 데이터베이스 영역을 벗어나도록 설정된 튜플을 찾아 특별한 방식으로 표시하거나 프리징하는 것입니다.

프리징된 튜플에 대해서는 가시성 규칙이 `xmin`을 고려할 필요가 없습니다. 왜냐하면 이런 튜플은 모든 스냅숏에서 볼 수 있다고 알려져 있기 때문입니다. 따라서 해당 트랜잭션 `ID`를 안전하게 재사용할 수 있게 됩니다.

프리징된 튜플에서는, `xmin` 트랜잭션 `ID`를 가상의 "음의 무한대"로 대체된다고 생각할 수 있습니다(이는 아래의 눈송이로 표현됩니다). 이는 해당 튜플이 실제 `ID`가 더 이상 중요하지 않은 과거의 트랜잭션에 의해 생성되었다는 것을 나타냅니다. 그러나 실제로 `xmin`은 변경되지 않습니다. 프리징 상태는 '커밋`committed`'과 '중단`aborted`' 두 가지 힌트 비트의 조합으로 정의됩니다. 이 두 가지 힌트 비트는 트랜잭션의 상태를 나타내는 데 사용되며, 이를 통해 튜플이 프리징되었는지 여부를 판단할 수 있습니다.



> 많은 자료들(공식 문서 포함)에서는 `FrozenTransactionId`의 값이 2 라고 언급하곤 합니다. 이것은 이미 설명한 "음의 무한대"를 의미합니다. 이 값은 PostgreSQL 9.4 버전 이전에서 `xmin`을 대체하는 데 사용되었지만, 현재는 힌트 비트를 사용합니다. 결과적으로 원래의 트랜잭션 `ID`는 튜플에 그대로 유지되어 디버깅이나 지원을 위한 작업에 도움이 됩니다. 그리고

[71] postgresql.org/docs/14/routine-vacuuming.html#VACUUM-FOR-WRAPAROUND

> 오래된 시스템이 더 높은 버전으로 업그레이드되더라도, 오래된 `FrozenTransactionId`를 포함하고 있을 수 있습니다.

`xmax` 트랜잭션 `ID`는 프리징 과정에 직접적으로 관여하지 않습니다. `xmax`는 오래된 튜플에만 존재하며, 해당 튜플이 모든 스냅숏에서 보이지 않게 되면(`xmax` `ID`가 데이터베이스 경계를 벗어난 경우), 베큠 작업으로 해당 튜플이 제거됩니다.

예제를 위해 새 테이블을 만들어 보는 것이 좋을 것 같습니다. `fillfactor` 매개변수를 가장 낮게 설정하여 각 페이지에 두 개의 튜플만 들어갈 수 있게 합니다. 이렇게 하면 진행 상황을 확인하는 것이 더 쉬워질 것입니다. 또한 테이블이 필요에 따라 자동으로 정리되지 않도록 `autovacuum` 기능을 비활성화해야 합니다.

```
=> CREATE TABLE tfreeze(
id integer,
s char(300)
)
WITH (fillfactor = 10, autovacuum_enabled = off);
```

우리는 `pageinspect`를 사용해 힙 페이지를 보여주는 함수를 만들 계획입니다. 이 함수는 페이지 범위를 처리하며, 각 튜플에 관해 프리징 속성(`f`)과 `xmin` 거래의 나이값을 표시할 것입니다. 당연히 나이 자체는 힙 페이지에 저장되지 않습니다. 이를 위해 시스템 함수인 `age`를 호출해야 합니다. 이 함수는 트랜잭션의 나이를 계산해주는 역할을 합니다.

```
=> CREATE FUNCTION heap_page(
relname text, pageno_from integer, pageno_to integer
)
RETURNS TABLE(
ctid tid, state text,
xmin text, xmin_age integer, xmax text
) AS $$
SELECT (pageno,lp)::text::tid AS ctid,
   CASE lp_flags
      WHEN 0 THEN 'unused'
      WHEN 1 THEN 'normal'
      WHEN 2 THEN 'redirect to '||lp_off
      WHEN 3 THEN 'dead'
  END AS state,
  t_xmin || CASE
              WHEN (t_infomask & 256+512) = 256+512 THEN ' f'
              WHEN (t_infomask & 256) > 0 THEN ' c'
              WHEN (t_infomask & 512) > 0 THEN ' a'
           ELSE ''
  END AS xmin,
  age(t_xmin) AS xmin_age,
  t_xmax || CASE
              WHEN (t_infomask & 1024) > 0 THEN ' c'
```

```
               WHEN (t_infomask & 2048) > 0 THEN ' a'
            ELSE ''
  END AS xmax
FROM generate_series(pageno_from, pageno_to) p(pageno),
     heap_page_items(get_raw_page(relname, pageno))
ORDER BY pageno, lp;
$$ LANGUAGE sql;
```

그럼 이제 테이블에 몇 개의 행을 추가하고, 즉시 가시성 맵을 생성하기 위해 베큠 명령을 실행해보겠습니다.

```
=> CREATE EXTENSION IF NOT EXISTS pg_visibility;

=> INSERT INTO tfreeze(id, s)
SELECT id, 'FOO'||id FROM generate_series(1,100) id;
INSERT 0 100
```

`pg_visibility` 확장을 사용하여 첫 번째 두 개의 힙 페이지를 살펴보겠습니다. 베큠 작업이 완료된 후에는 두 페이지 모두 가시성 맵에 '전체 가시성'(`all_visible`) 태그가 붙지만, 프리징 맵에는 '전체 동결'(`all_frozen`) 태그가 붙지 않습니다. 이는 아직 프리징되지 않은 튜플이 페이지 내에 일부 포함되어 있기 때문입니다. 이렇게 관찰하면서 직접 프리징 과정을 이해하는 데 도움이 될 것입니다.

```
=> VACUUM tfreeze;
=> SELECT *
FROM generate_series(0,1) g(blkno),
     pg_visibility_map('tfreeze',g.blkno)
ORDER BY g.blkno;
 blkno | all_visible | all_frozen
-------+-------------+------------
     0 |           t |          f
     1 |           t |          f
(2 행s)
```

생성한 행의 거래의 `xmin_age` 값은 1입니다. 이는 이 행이 시스템에서 가장 최근에 수행된 트랜잭션에 의해 생성되었음을 나타냅니다.

```
=> SELECT * FROM heap_page('tfreeze',0,1);
  ctid |  state |  xmin | xmin_age | xmax
-------+--------+-------+----------+------
 (0,1) | normal | 856 c | 1        | 0 a
 (0,2) | normal | 856 c | 1        | 0 a
 (1,1) | normal | 856 c | 1        | 0 a
 (1,2) | normal | 856 c | 1        | 0 a
(4 행s)
```

7.3 프리징 관리

데이터베이스에서 프리징을 제어하는 주요 매개변수는 네 가지로, 모두 트랜잭션의 나이를 나타내고 특정 이벤트가 발생하는 시기를 정의합니다.

- `vacuum_freeze_min_age`: 프리징이 시작되는 시점을 정의합니다.
- `vacuum_freeze_table_age`: 공격적인 프리징이 시작되는 시점을 나타냅니다.
- `autovacuum_freeze_max_age`: 시스템이 자동으로 프리징을 강제적으로 실행하는 시점을 설정합니다.
- `vacuum_failsafe_age`: 프리징이 우선 순위를 부여받아 실행되는 시점을 나타냅니다.

최소 프리징 나이

`vacuum_freeze_min_age` 매개변수(기본값: 5,000만)는 데이터베이스에서 `xmin` 트랜잭션의 최소 프리징 나이를 설정합니다. 이 값을 낮게 설정하면 데이터베이스의 부하가 증가합니다. 왜냐하면 자주 변경되는 `hot`한 행에 대해 모든 새로운 버전을 프리징하는 것은 불필요한 작업이 될 수 있기 때문입니다. 반면에 이 값을 높게 설정하면 프리징 작업을 지연시켜 일정 기간 동안은 프리징 작업을 수행하지 않을 수 있습니다.

프리징 작업을 직접 관찰하기 위해 `vacuum_freeze_min_age` 매개변수를 1로 설정하는 방법은 다음과 같습니다:

```
ALTER SYSTEM SET vacuum_freeze_min_age = 1;
SELECT pg_reload_conf();
```

이 설정을 적용한 후에는 0 페이지에 있는 특정 행을 수정해 보세요. 예를 들어:

```
UPDATE tfreeze SET s = 'BAR' WHERE id = 1;
```

이렇게 하면 `fillfactor` 값이 낮아 새로운 행 버전이 같은 페이지에 위치하게 됩니다. 이러한 설정 변경을 통해 어떻게 프리징 동작이 바뀌는지 실험적으로 확인할 수 있습니다.

모든 트랜잭션의 연령이 1씩 증가하고, 힙 페이지는 다음과 같이 보입니다:

```
=> SELECT * FROM heap_page('tfreeze',0,1);
      ctid ¦       state ¦      xmin ¦        xmin_age ¦ xmax
-----------+-------------+-----------+-----------------+-------
     (0,1) ¦      normal ¦     856 c ¦               2 ¦ 857
     (0,2) ¦      normal ¦     856 c ¦               2 ¦ 0 a
     (0,3) ¦      normal ¦     857   ¦               1 ¦ 0 a
     (1,1) ¦      normal ¦     856 c ¦               2 ¦ 0 a
     (1,2) ¦      normal ¦     856 c ¦               2 ¦ 0 a
(5 행s)
```

이 시점에서, `vacuum_freeze_min_age = 1`보다 오래된 튜플은 프리징 대상입니다. 그러나 가시성 맵에 표시된 페이지는 베큠이 처리하지 않을 것입니다:

```
=> SELECT * FROM generate_series(0,1) g(blkno),
                 pg_visibility_map('tfreeze',g.blkno)
    ORDER BY g.blkno;

 blkno | all_visible | all_frozen
-------+-------------+------------
     0 | f           | f
     1 | t           | f
(2 행s)
```

이전의 `UPDATE` 명령은 0 페이지의 가시성 비트를 제거했기 때문에 이 페이지에서 적절한 `xmin` 나이를 가진 튜플이 프리징 될 것입니다. 그러나 첫 번째 페이지는 완전히 건너뛰게 됩니다:

```
=> VACUUM tfreeze;

=> SELECT * FROM heap_page('tfreeze',0,1);
  ctid |       state | xmin | xmin_age | xmax
-------+-------------+------+----------+------
 (0,1) | redirect to 3 |      |          |
 (0,2) | normal      | 856 f |        2 | 0 a
 (0,3) | normal      | 857 c |        1 | 0 a
 (1,1) | normal      | 856 c |        2 | 0 a
 (1,2) | normal      | 856 c |        2 | 0 a
(5 행s)
```

이제 0 페이지는 가시성 맵에 다시 나타나고, 이 페이지에 변경 사항이 없다면 베큠 작업은 더 이상 이 페이지로 돌아가지 않을 것입니다:

```
=> SELECT * FROM generate_series(0,1) g(blkno),
                 pg_visibility_map('tfreeze',g.blkno)
   ORDER BY g.blkno;

 blkno | all_visible | all_frozen
-------+-------------+------------
     0 | t           | f
     1 | t           | f
(2 행s)
```

공격적인 프리징 나이

방금 보았듯이, 페이지에 현재 스냅숏에서 가시성이 있는 튜플만 포함되어 있다면, 베큠 작업은 그들을 프리징하지 않습니다. 이 제약을 극복하기 위해 PostgreSQL은 `vacuum_freeze_table_age`(기본값: 1억 5천만) 매개변수를 제공합니다. 이는 베큠 작업이 가시성 맵을 무시하고 모든 힙 페이지를 프리징할 수 있는 트랜잭션 나이를 정의합니다.

각 테이블에 대해 시스템 카탈로그는 모든 이전 트랜잭션이 프리징된 것으로 알려진 트랜잭션 나이(`relfrozenxid`)를 유지합니다:

```
=> SELECT relfrozenxid, age(relfrozenxid)
   FROM pg_class
   WHERE relname = 'tfreeze';

         relfrozenxid ¦ age
──────────────────────+─────
                  854 ¦ 4
(1 행)
```

이 트랜잭션의 나이는 `vacuum_freeze_table_age` 값과 비교되어 공격적인 프리징을 수행할 시기를 결정합니다.

프리징 맵은 베큠 작업 중에 전체 테이블을 스캔할 필요가 없게 해주는 중요한 최적화 도구입니다. 이 맵을 사용하면 나타나지 않는 페이지만 확인하면 되므로, 처리 시간을 크게 절약할 수 있습니다. 또한, 프리징 맵은 장애 허용 기능도 제공합니다. 만약 베큠 작업이 어떤 이유로 중단되더라도, 다음번 실행 때 이미 처리된 페이지를 다시 처리할 필요가 없습니다. 이미 프리징 맵에 태그가 되어있는 페이지들은 건너뛸 수 있기 때문입니다.

PostgreSQL은 시스템에서 수행된 트랜잭션 수가 `vacuum_freeze_table_age`에서 `vacuum_freeze_min_age`를 뺀 값에 도달하면, 테이블의 모든 페이지에 대해 공격적인 프리징을 수행합니다. 기본 설정을 사용하는 경우, 이는 1억 번의 트랜잭션마다 한 번씩 발생합니다. 만약 `vacuum_freeze_min_age` 값을 너무 크게 설정하면, 과도한 프리징이 발생하며 이는 시스템에 불필요한 부하를 초래할 수 있습니다. 따라서 이 값을 적절히 설정하는 것이 시스템 성능에 중요합니다.

전체 테이블을 프리징하기 위해 `vacuum_freeze_table_age` 값을 4로 줄이고, 이제 공격적인 프리징 조건이 충족됩니다:

```
=> ALTER SYSTEM SET vacuum_freeze_table_age = 4;
=> SELECT pg_reload_conf();
```

`VACUUM` 명령을 실행합니다:

```
=> VACUUM VERBOSE tfreeze;
INFO: aggressively vacuuming "public.tfreeze"
INFO: table "tfreeze": found 0 removable, 100 nonremovable 행
versions in 50 out of 50 pages
DETAIL: 0 dead 행 versions cannot be removed yet, oldest xmin: 858
Skipped 0 pages due to buffer pins, 0 frozen pages.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```

이제 전체 테이블이 분석되었으므로 `relfrozenxid` 값이 진행될 수 있습니다. 힙 페이지에는 이제 더 이전의 프리징 되지 않은 `xmin` 트랜잭션이 없음이 보장됩니다:

```
=> SELECT relfrozenxid, age(relfrozenxid)
FROM pg_class
WHERE relname = 'tfreeze';

          relfrozenxid ¦ age
———————————————————————+——————
                   857 ¦ 1
(1 행)
```

첫 번째 페이지에는 이제 프리징 된 튜플만 포함되어 있습니다:

```
=> SELECT * FROM heap_page('tfreeze',0,1);

      ctid ¦                   state ¦      xmin ¦       xmin_age ¦ xmax
———————————+———————————————————————+———————————+————————————————+———————
     (0,1) ¦          redirect to 3 ¦           ¦                ¦
     (0,2) ¦                 normal ¦      856 f ¦              2 ¦ 0 a
     (0,3) ¦                 normal ¦      857 c ¦              1 ¦ 0 a
     (1,1) ¦                 normal ¦      856 f ¦              2 ¦ 0 a
     (1,2) ¦                 normal ¦      856 f ¦              2 ¦ 0 a
(5 행s)
```

또한, 이 페이지는 프리징 맵에 태그가 지정되어 있습니다:

```
=> SELECT * FROM generate_series(0,1) g(blkno),
                 pg_visibility_map('tfreeze',g.blkno)
   ORDER BY g.blkno;

    blkno ¦         all_visible ¦ all_frozen
——————————+———————————————————————+——————————————
        0 ¦                   t ¦                f
        1 ¦                   t ¦                t
(2 행s)
```

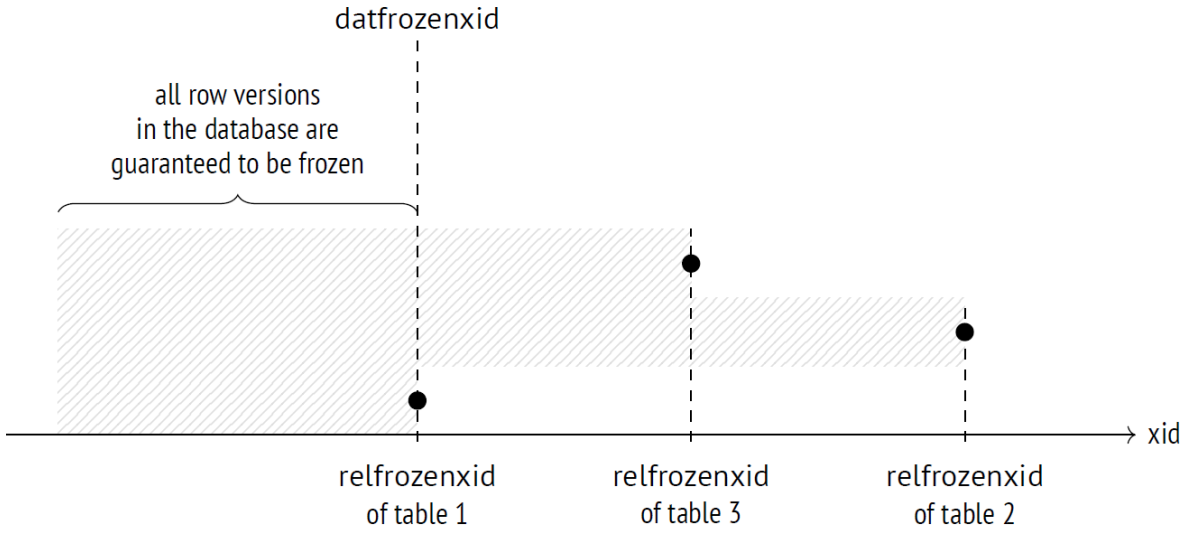
강제 자동 베큠 나이

때로는 튜플을 적시에 프리징하기 위해 위에서 설명한 두 가지 매개변수만으로는 충분하지 않을 수 있습니다. 자동 베큠이 꺼져 있을 수 있으며, 정기적인 베큠이 전혀 호출되지 않을 수도 있습니다(이는 매우 좋지 않은 아이디어이지만 기술적으로 가능합니다). 게다가 `template0`과 같은 일부 비활성 데이터베이스는 베큠을 수행하지 않을 수도 있습니다. PostgreSQL은 이러한 상황을 처리하기 위해 자동 베큠을 강제로 실행할 수 있습니다.

자동 베큠은 데이터베이스의 일부 트랜잭션이 아직 처리되지 않은 상태(`IS`)에서 그 나이가

autovacuum_freeze_max_age 값(기본값: 2억)을 넘을 가능성이 있을 때, 비록 사용자가 이를 중지시키더라도 강제로 작동하게 됩니다. 이 판단은 모든 테이블에서 가장 오래된 pg_class.relfrozenxid 트랜잭션의 나이를 기준으로 이루어집니다. 이렇게 처리된 이전 트랜잭션들은 모두 프리징 상태가 보장되며, 이들 트랜잭션의 ID는 시스템 카탈로그에 기록됩니다.

```
=> SELECT datname, datfrozenxid, age(datfrozenxid) FROM pg_database;

  datname  | datfrozenxid | age
-----------+--------------+-----
 postgres  |          726 | 132
 template1 |          726 | 132
 template0 |          726 | 132
 internals |          726 | 132
(4 행s)
```



autovacuum_freeze_max_age의 제한은 20억 개의 트랜잭션으로 설정되어 있고, 이는 약 원의 절반 크기보다 약간 작은 값입니다. 기본 설정값은 이보다 10배 작게 설정되어 있습니다. 이는 트랜잭션 ID wraparound의 위험을 증가시키는 큰 값 때문에 합리적인 조치로 취해진 것입니다. PostgreSQL은 필요한 튜플을 적절한 시기에 동결하지 못하는 경우가 있을 수 있습니다. 이런 상황에서는 서버가 가능한 문제를 미리 방지하기 위해 즉각 중단되어야 하며, 이후에는 관리자에 의해 다시 시작되어야 합니다.

autovacuum_freeze_max_age 값은 CLOG의 크기에도 직접적인 영향을 미칩니다. 프리징된 트랜잭션의 상태를 유지할 필요가 없기 때문에, 클러스터에서 가장 오래된 datfrozenxid 값을 가진 트랜잭션 이전의 모든 트랜잭션은 반드시 프리징 상태에 있습니다. 이후 필요하지 않게 된 CLOG 파일들은 자동 제큠 과정에 의해 자동으로 제거됩니다.[72]

autovacuum_freeze_max_age 매개변수를 수정하려면 서버를 재시작해야 합니다. 하지만, 이전에 언급한 모든 동결 설정들은 해당 스토리지 매개변수를 통해 테이블 수준에서도 조절할 수 있습니다. 이런 매개변수들

[72] backend/commands/vacuum.c, vac_truncate_clog function

의 이름은 모두 "auto"로 시작하며, 다음과 같습니다:

- autovacuum_freeze_min_age와 toast.autovacuum_freeze_min_age
- autovacuum_freeze_table_age와 toast.autovacuum_freeze_table_age
- autovacuum_freeze_max_age와 toast.autovacuum_freeze_max_age

실패 방지 프리징 나이

자동 배큠이 트랜잭션 ID 랩어라운드를 방지하기 위해 노력 중이지만, 시간이 절박한 상황이라면, 안전 스위치가 작동하게 됩니다. 이때 자동 배큠은 autovacuum_vacuum_cost_delay (vacuum_cost_delay) 설정을 무시하고, 가능한 빠르게 힙 튜플을 고정하기 위해 인덱스를 배큠하지 않습니다.

데이터베이스 내에서 해제되지 않은 트랜잭션의 나이가 vacuum_failsafe_age 값보다 클 위험이 있을 경우, 보호를 위한 실패 방지 모드가 활성화됩니다.[73] 이 값은 autovacuum_freeze_max_age (기본값: 16억)보다 크다고 가정하게 됩니다.

7.4 수동 프리징

가끔은 자동 배큠에 의존하는 대신 수동으로 프리징을 관리하는 것이 더 편리할 때도 있습니다.

베큠을 통한 프리징

FREEZE 옵션을 사용하여 VACUUM 명령을 실행하면 프리징을 시작할 수 있습니다. 이렇게 하면 vacuum_freeze_min_age = 0인 것처럼 트랜잭션의 나이와 상관없이 모든 힙 튜플을 프리징 할 수 있습니다.

만약 이러한 호출이 힙 튜플을 가능한 빠르게 프리징 하는 것이 목적이라면, 실패 방지 모드처럼 인덱스 배큠을 비활성화하는 것이 합리적일 수 있습니다. 이를 위해 VACUUM (freeze, index_cleanup false) 명령을 명시적으로 실행하거나 vacuum_index_cleanup 설정 매개변수를 통해 수행할 수 있습니다. 하지만, 이는 VACUUM이 주요 작업인 페이지 정리를 주로 수행하지 못하게 만드므로, 이를 정기적으로 실행하는 것은 바람직하지 않습니다.

초기 적재 시 데이터를 프리징

변경이 예상되지 않는 데이터는 데이터베이스에 로딩될 때 한 번에 프리징 될 수 있습니다. 이는 FREEZE 옵션을 사용하여 COPY 명령을 실행함으로써 이루어집니다.

초기 적재 과정에서만 튜플을 프리징 할 수 있으며, 이는 결과 테이블이 동일한 트랜잭션 내에서 생성되거나 삭제된 경우에만 가능합니다. 이런 작업은 테이블에 관한 배타적인 잠금을 필요하므로 이 제한이 필요합니다. 프리징 된 튜플은 격리 수준에 상관없이 모든 스냅숏에서 볼 수 있어야 하기 때문입니다. 그렇지 않다면, 트랜잭션이 업로드되는 동안 갑작스럽게 새롭게 고정된 튜플을 보게 될 수 있습니다. 그러나 잠금이 설정된 경우, 다른 트랜잭션은 해당 테이블에 접근할 수 없게 됩니다.

그럼에도 불구하고 격리성을 깨는 것은 기술적으로 가능합니다. 별도의 세션에서 반복적인 읽기 격리 수준으로 새로운 트랜잭션을 시작해 보겠습니다:

---

[73] backend/access/heap/vacuumlazy.c, lazy_check_wraparound_failsafe function

```
 => BEGIN ISOLATION LEVEL REPEATABLE READ;
 => SELECT 1; -- 스냅숏이 구축됩니다
```

같은 트랜잭션 내에서 `tfreeze` 테이블을 비우고 이 테이블에 새로운 행을 삽입합니다. (읽기 전용 트랜잭션이 이미 `tfreeze` 테이블에 액세스한 경우, `TRUNCATE` 명령이 차단됩니다.)

```
=> BEGIN;
=> TRUNCATE tfreeze;
=> COPY tfreeze FROM stdin WITH FREEZE;
1 FOO
2 BAR
3 BAZ
\.
=> COMMIT;
```

이제 읽기 트랜잭션이 새로운 데이터를 볼 수 있습니다:

```
 => SELECT count(*) FROM tfreeze;
     count
 ────────
           3
 (1 행)
 => COMMIT;
```

이는 격리성을 깨지만, 데이터 적재가 정기적으로 발생하지 않는 한 대부분의 경우에는 문제를 일으키지 않을 것입니다. 데이터를 프리징으로 적재 하는 경우, 가시성 맵이 한 번에 생성되며 페이지 헤더는 가시성 속성을 받습니다:

```
=> SELECT * FROM pg_visibility_map('tfreeze',0);
       all_visible  ¦ all_frozen
────────────────+─────────────
                t ¦ t
 (1 행)

=> SELECT flags & 4 > 0 AS all_visible
   FROM page_header(get_raw_page('tfreeze',0));
    all_visible
──────────────
                t
(1 행)
```

따라서 데이터가 프리징으로 적재 된 경우, 데이터가 변경되지 않는 한 테이블은 `VACUUM`에 의해 처리되지 않을 것입니다. 하지만 `TOAST` 테이블에 대해서는 아직 이 기능이 지원되지 않습니다. 크기가 너무 큰 값이 적재되면 `VACUUM`은 모든 페이지 헤더에서 가시성 속성을 설정하기 위해 전체 `TOAST` 테이블을 다시 작성해야 합니다.

8 장 테이블과 인덱스 재구성

8.1 전체 베큠작업

왜 정기적인 베큠만으로는 충분하지 않을까요?

정기적인 베큠은 페이지 가지치기보다 더 많은 공간을 해제할 수 있지만, 가끔씩은 여전히 부족한 경우가 있습니다.

테이블 또는 인덱스 파일의 크기가 커진 경우, 베큠은 페이지 내부의 일부 공간을 정리할 수 있지만, 페이지 수 자체를 줄이는 것은 흔치 않은 일입니다. 회수된 공간은 파일의 맨 끝에 여러 개의 빈 페이지가 나타날 때만 운영 체제에 반환될 수 있으며, 이런 경우는 그다지 자주 발생하지 않습니다.

과도한 크기는 불편한 결과를 초래할 수 있습니다:

- 전체 테이블(또는 인덱스) 스캔 시간이 길어집니다.
- 더 큰 버퍼 캐시가 필요할 수 있습니다(페이지는 전체로 캐시되므로 데이터 밀도가 감소합니다).
- B-트리가 추가적인 레벨을 가지게 되어 인덱스 접근 속도가 느려집니다.
- 파일이 디스크와 백업에서 추가적인 공간을 차지합니다.

파일 내의 유용한 데이터 비율이 합리적인 수준 아래로 떨어진 경우, 관리자는 `VACUUM FULL`[74] 명령을 실행하여 전체 베큠을 수행할 수 있습니다. 이 경우, 테이블과 모든 인덱스는 처음부터 다시 구축되며, 데이터는 가능한 한 밀집하게(패킹 인자를 고려하여) 재구성 됩니다.

전체 베큠작업이 진행될 때, PostgreSQL은 우선 테이블 전체를 완벽하게 재구성하고, 그 후 각각의 인덱스를 재조정합니다. 이러한 객체가 재조정되는 동안, 이전 파일과 새로운 파일이 디스크[75]에 동시에 저장되어야 하므로, 상당한 여유 공간이 필요할 수 있습니다.

더불어, 이 과정은 테이블의 읽기 및 쓰기 접근을 완전히 차단하므로, 이 부분을 반드시 고려해야 합니다.

데이터 밀도 예측

예시를 위해 몇 개의 행을 테이블에 삽입해 보겠습니다:

```
=> TRUNCATE vac;
=> INSERT INTO vac(id,s)
   SELECT id, id::text FROM generate_series(1,500000) id;
```

`pgstattuple` 확장 기능을 사용하여 저장 공간 밀도를 추정할 수 있습니다:

```
=> CREATE EXTENSION pgstattuple;
```

[74] postgresql.org/docs/14/routine-vacuuming.html#VACUUM-FOR-SPACE-RECOVERY

[75] backend/commands/cluster.c

```
=> SELECT * FROM pgstattuple('vac') \gx
-[ RECORD 1 ]------+---------
table_len          | 70623232
tuple_count        | 500000
tuple_len          | 64500000
tuple_percent      | 91.33
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 381844
free_percent       | 0.54
```

이 함수는 전체 테이블을 읽고 파일 내 공간 분포에 관한 통계를 표시합니다. `tuple_percent` 필드는 유용한 데이터(힙 튜플)가 차지하는 공간의 백분율을 나타냅니다. 이 값은 페이지 내의 다양한 메타데이터 때문에 반드시 100%보다 작습니다. 그러나 이 예시에서는 여전히 상당히 높은 값입니다.

인덱스의 경우, 표시된 정보는 약간 다르지만, `avg_leaf_density` 필드는 동일한 의미를 가지고 있습니다. 이는 유용한 데이터(B-트리 리프 페이지)의 백분율을 나타냅니다.

```
=> SELECT * FROM pgstatindex('vac_s') \gx
-[ RECORD 1 ]-----+------------
version           | 4
tree_level        | 3
index_size        | 114302976
root_block_no     | 2825
internal_pages    | 376
leaf_pages        | 13576
empty_pages       | 0
deleted_pages     | 0
avg_leaf_density  | 53.88
leaf_fragmentation | 10.59
```

이전에 쓰였던 `pgstattuple` 함수는 테이블이나 인덱스를 전체적으로 읽어 정확한 통계치를 얻습니다. 하지만 큰 객체의 경우 비용이 많이 소모되기 때문에, 이 확장 기능은 가시성 맵에서 추적된 페이지를 무시하는 `pgstattuple_approx`라는 새로운 함수를 제공합니다.

데이터 볼륨과 파일 크기의 비율을 시스템 카탈로그를 이용해 대략적으로 예측하는 방법은 더욱 빠르지만, 역시 정확성은 더욱 떨어집니다.[76]

테이블과 해당 인덱스의 현재 크기는 다음과 같습니다:

```
=> SELECT pg_size_pretty(pg_table_size('vac')) AS table_size,
          pg_size_pretty(pg_indexes_size('vac')) AS index_size;
```

[76] wiki.postgresql.org/wiki/Show_database_bloat

```
       table_size ¦ index_size
──────────────+─────────────
            67 MB ¦ 109 MB
(1 행)
```

이제 모든 행의 90%를 삭제해 보겠습니다:

```
=> DELETE FROM vac WHERE id % 10 != 0;
   DELETE 450000
```

베큠 작업은 파일 크기에 영향을 주지 않습니다. 왜냐하면 파일 끝에 빈 페이지가 없기 때문입니다:

```
=> VACUUM vac;
=> SELECT pg_size_pretty(pg_table_size('vac')) AS table_size,
          pg_size_pretty(pg_indexes_size('vac')) AS index_size;
       table_size ¦ index_size
──────────────+─────────────
            67 MB ¦ 109 MB
(1 행)
```

그러나 데이터 밀도가 약 10배 정도 감소했습니다:

```
=> SELECT vac.tuple_percent, vac_s.avg_leaf_density
   FROM pgstattuple('vac') vac, pgstatindex('vac_s') vac_s;
        tuple_percent ¦ avg_leaf_density
──────────────────+──────────────────
                9.13 ¦ 6.71
(1 행)
```

테이블과 인덱스는 현재 다음 파일에 위치해 있습니다:

```
=> SELECT pg_relation_filepath('vac') AS vac_filepath,
          pg_relation_filepath('vac_s') AS vac_s_filepath \gx
─[ RECORD 1 ]──+──────────────────
vac_filepath     ¦ base/16391/16514
vac_s_filepath   ¦ base/16391/16515
```

`VACUUM FULL` 이 실행된 후의 결과를 확인해 보겠습니다. 명령이 수행되는 동안 진행 상황은 `pg_stat_progress_cluster` 뷰에서 확인할 수 있습니다(이는 `VACUUM`을 위해 제공되는 `pg_stat_progress_vacuum` 뷰와 유사합니다):

```
=> VACUUM FULL vac;

  => SELECT * FROM pg_stat_progress_cluster \gx
  -[ RECORD 1 ]-----+-----------------
  pid               ¦ 19488
```

```
  datid             ¦ 16391
  datname           ¦ internals
  relid             ¦ 16479
  command             ¦ VACUUM FULL
  phase               ¦ rebuilding index
  cluster_index_relid ¦ 0
  heap_tuples_scanned ¦ 50000
  heap_tuples_written ¦ 50000
  heap_blks_total     ¦ 8621
  heap_blks_scanned   ¦ 8621
  index_rebuild_count ¦ 0
```

예상대로, `VACUUM FULL` 단계[77]는 일반적인 베큠 작업과 다릅니다.

전체 베큠작업이 이전 파일을 새 파일로 대체했습니다:

```
=> SELECT pg_relation_filepath('vac') AS vac_filepath,
          pg_relation_filepath('vac_s') AS vac_s_filepath \gx
-[ RECORD 1 ]--+------------------
vac_filepath    ¦ base/16391/16526
vac_s_filepath  ¦ base/16391/16529
```

테이블과 인덱스의 크기가 현저히 작아졌습니다:

```
=> SELECT pg_size_pretty(pg_table_size('vac')) AS table_size,
          pg_size_pretty(pg_indexes_size('vac')) AS index_size;
       table_size ¦ index_size
------------------+------------
          6904 kB ¦ 6504 kB
(1 행)
```

결과적으로 데이터 밀도가 증가했습니다. 인덱스의 경우, 원래 것보다 더 높습니다. 사용 가능한 데이터를 기반으로 처음부터 B-트리를 생성하는 것이 이미 존재하는 인덱스에 행 단위로 항목을 삽입하는 것보다 효율적입니다:

```
=> SELECT vac.tuple_percent,
          vac_s.avg_leaf_density
FROM pgstattuple('vac') vac,
     pgstatindex('vac_s') vac_s;
         tuple_percent ¦ avg_leaf_density
-----------------------+------------------
                 91.23 ¦ 91.08
(1 행)
```

[77] postgresql.org/docs/14/progress-reporting.html#CLUSTER-PHASES

프리징

테이블이 재구성되는 동안 PostgreSQL은 튜플을 프리징시키며, 이 과정은 다른 작업에 비해 거의 비용이 소모되지 않습니다:

```
=> SELECT * FROM heap_page('vac',0,0) LIMIT 5;
  ctid ¦  state ¦  xmin ¦ xmin_age ¦ xmax
-------+--------+-------+----------+------
 (0,1) ¦ normal ¦ 861 f ¦        5 ¦ 0 a
 (0,2) ¦ normal ¦ 861 f ¦        5 ¦ 0 a
 (0,3) ¦ normal ¦ 861 f ¦        5 ¦ 0 a
 (0,4) ¦ normal ¦ 861 f ¦        5 ¦ 0 a
 (0,5) ¦ normal ¦ 861 f ¦        5 ¦ 0 a
(5 행s)
```

하지만 페이지는 가시성 맵이나 프리징 맵에 등록되지 않으며, 페이지 헤더에 가시성 속성이 지정되지 않습니다 (`COPY` 명령을 `FREEZE` 옵션과 함께 실행할 때 발생하는 것과 달리):

```
=> SELECT * FROM pg_visibility_map('vac',0);
        all_visible ¦ all_frozen
--------------------+-------------
                  f ¦ f
(1 행)

=> SELECT flags & 4 > 0 as all_visible
    FROM page_header(get_raw_page('vac',0));
   all_visible
---------------
                  f
(1 행)
```

상황은 `VACUUM`이 호출되거나 (또는 자동 베큠이 트리거될 때) 개선됩니다:

```
=> VACUUM vac;
=> SELECT * FROM pg_visibility_map('vac',0);
        all_visible ¦ all_frozen
--------------------+-------------
                  t ¦ t
(1 행)

=> SELECT flags & 4 > 0 AS all_visible
   FROM page_header(get_raw_page('vac',0));
 all_visible
--------------
```

```
 t
(1 행)
```

이것은 페이지에 있는 모든 튜플이 데이터베이스의 가시성 범위를 벗어나도, 해당 페이지는 재작성되어야 함을 뜻합니다.

8.2 다른 재구성 방법

전체 베큠작업의 대안 방법들

`VACUUM FULL` 외에도 테이블과 인덱스를 완전히 재구성하는 다양한 명령어들이 있습니다. 이들 모두 테이블을 독점적으로 잠그고, 기존의 데이터 파일을 삭제한 후 새로 생성합니다.

`CLUSTER` 명령어는 `VACUUM FULL`과 완전히 같은 기능을 하며, 사용 가능한 인덱스 중 하나를 기반으로 파일 내의 튜플을 재정렬합니다. 이는 특정 경우에 플래너가 인덱스 스캔을 더 효율적으로 이용할 수 있게 도와줍니다. 하지만 클러스터링은 지속되지 않으며, 이후의 테이블 수정은 튜플의 물리적 순서를 깨뜨릴 수 있습니다.

프로그래밍 측면에서 보면, `VACUUM FULL`은 튜플 재정렬이 필요 없는 `CLUSTER` 명령의 특별한 경우입니다.[78]

`REINDEX` 명령어는 하나 이상의 인덱스를 재구성합니다.[79] 사실, `VACUUM FULL`과 `CLUSTER`는 인덱스를 재구성할 때 이 `REINDEX` 명령어를 내부적으로 사용합니다.

`TRUNCATE` 명령어[80]는 테이블의 모든 행을 삭제합니다. `WHERE` 절 없이 `DELETE`를 수행하는 것과 논리적으로 동일합니다. 그러나 `DELETE`는 힙 튜플을 삭제로 표시만 하므로(따라서 여전히 `VACUUM` 작업이 필요합니다.), `TRUNCATE`는 새로운 빈 파일을 생성하여 일반적으로 더 빠른 속도를 보여줍니다.

재구성 중 다운타임 감소시키기

`VACUUM FULL`은 작업 동안 테이블을 (질의에 관해) 완전히 독점적으로 잠그기 때문에, 이를 정기적으로 실행하는 것은 바람직하지 않습니다. 이는 보통 고가용성 시스템에 부적합합니다.

`pg_repack`[81]과 같은 여러 확장 기능을 사용하면 테이블과 인덱스를 거의 다운타임 없이 재구성할 수 있습니다. 독점적인 잠금은 여전히 필요하지만, 이 과정은 시작과 종료 시에만 짧은 시간 동안 필요합니다. 이를 위해 변경 사항은 트리거에 의해 원본 테이블에서 추적되고, 새로운 테이블에 적용됩니다. 작업을 완료하기 위해, 유틸리티는 시스템 카탈로그에서 한 테이블을 다른 테이블로 교체합니다.

`pgcompacttable` 도구[82]는 또 다른 해결책을 제공합니다. 이는 현재 행 버전이 파일의 시작 부분으로 점진적으로 이동하도록 여러 가짜 행 수정(데이터를 변경하지 않음)을 수행합니다.

수정 작업 사이에서는 `VACUUM` 작업이 오래된 튜플을 제거하고 파일 크기를 점진적으로 줄입니다. 이 방법은

---

[78] backend/commands/cluster.c

[79] backend/commands/indexcmds.c

[80] backend/commands/tablecmds.c, ExecuteTruncate function

[81] github.com/reorg/pg_repack

[82] github.com/dataegret/pgcompacttable

훨씬 더 많은 시간과 자원이 필요로 하지만, 테이블 재구성을 위한 추가적인 공간이 필요 없으며, 부하 증가도 발생시키지 않습니다. 테이블이 점진적으로 줄어들면서 짧은 시간 동안 독점적인 잠금이 필요하게 되지만, `VACUUM` 작업이 이를 상당히 원활하게 처리합니다.

8.3 주의사항

읽기 전용 질의

파일이 부풀어 오르는 이유 중 하나는 데이터 업데이트와 함께 데이터베이스 범위를 유지하는 장기 실행 트랜잭션입니다.

이에 따라 장기 실행(읽기 전용) 트랜잭션은 문제를 일으키지 않습니다. 그래서 일반적인 접근 방식은 부하를 다른 시스템으로 분산하는 것입니다. 빠른 `OLTP`(`On-Line Transaction Processing`) 쿼리는 주 서버에 유지하고 모든 `OLAP`(`On-Line Analytical Processing`) 트랜잭션은 복제본으로 보냅니다. 비록 이는 설루션을 더 비싸고 복잡하게 만들지만, 이러한 조치는 필수적일 수 있습니다.

장기 실행 트랜잭션은 필요성보다는 애플리케이션 또는 드라이버 버그의 결과일 수도 있습니다. 문제를 문명적인 방식으로 해결할 수 없는 경우, 관리자는 다음 두 개의 매개변수를 사용할 수 있습니다:

- `old_snapshot_threshold` 매개변수는 스냅숏의 최대 수명을 정의합니다. 이 시간이 지나면 서버는 오래된 튜플을 제거할 권한을 갖습니다. 장기 실행 트랜잭션이 아직 이러한 튜플이 필요하다면 "스냅숏이 너무 오래되었습니다"라는 오류가 발생합니다.
- `idle_in_transaction_session_timeout` 매개변수는 유휴 트랜잭션의 수명을 제한합니다. 이 임곗값에 도달하면 트랜잭션이 중단됩니다.

데이터 수정

파일 부풀림bloating의 주요 원인 중 하나는 대량의 튜플을 동시에 수정하는 행위입니다. 테이블의 모든 행이 수정될 경우, 튜플의 수는 두 배로 증가하며, 이로 인해 백업 작업에 필요한 충분한 시간을 확보할 수 없게 됩니다. 페이지 가지치기로 이 문제를 어느 정도 해결할 수 있지만, 완전히 문제를 없애지는 못합니다.

프로세스된 행을 추적하기 위해서, 출력을 다른 열로 확장하는 예시를 들어보겠습니다. `vac`이라는 테이블에 `processed`라는 `boolean` 타입의 열을 추가하겠습니다:

```
=> ALTER TABLE vac ADD processed boolean DEFAULT false;
=> SELECT pg_size_pretty(pg_table_size('vac'));
pg_size_pretty
----------------
6936 kB
(1 행)
```

이제 모든 행을 수정하면 테이블의 크기가 거의 두 배로 늘어납니다:

```
=> UPDATE vac SET processed = true;
UPDATE 50000
=> SELECT pg_size_pretty(pg_table_size('vac'));
pg_size_pretty
----------------
```

```
14 MB
(1 행)
```

이러한 상황을 해결하기 위해 단일 트랜잭션에서 수행되는 변경 사항의 수를 줄여 시간을 분산시킬 수 있습니다. 그런 다음 백업 작업을 통해 오래된 튜플을 삭제하고 이미 존재하는 페이지 내에 새로운 공간을 확보할 수 있습니다.

각 행 수정을 개별적으로 커밋할 수 있다고 가정하면, 다음과 같은 쿼리를 사용하여 지정된 크기의 배치 행을 선택할 수 있습니다:

```
SELECT ID
FROM table
WHERE filtering the already processed 행s
LIMIT batch size
FOR UPDATE SKIP LOCKED
```

이 코드 조각은 지정된 크기를 초과하지 않는 배치 행 집합을 선택하고 즉시 잠그는 기능을 제공합니다. 다른 트랜잭션에 의해 이미 잠긴 행은 건너뛰며, 다음번에 다른 배치로 들어갑니다. 이는 배치 크기를 쉽게 변경하고 오류 발생 시 작업을 다시 시작할 수 있는 유연하고 편리한 설루션입니다. `processed` 속성을 해제하고 전체 백업 작업을 수행하여 테이블의 원래 크기를 복원해 봅시다:

```
=> UPDATE vac SET processed = false;
=> VACUUM FULL vac;
```

첫 번째 배치가 수정되면 테이블 크기가 약간 증가합니다:

```
=> WITH batch AS (
SELECT id FROM vac WHERE NOT processed LIMIT 1000
FOR UPDATE SKIP LOCKED
)
UPDATE vac SET processed = true
WHERE id IN (SELECT id FROM batch);
UPDATE 1000
=> SELECT pg_size_pretty(pg_table_size('vac'));
pg_size_pretty
————————————————
7064 kB
(1 행)
```

하지만 이제부터는 크기가 거의 동일하게 유지되며, 새로운 튜플이 제거된 튜플을 대체합니다:

```
=> VACUUM vac;
=> WITH batch AS (
SELECT id FROM vac WHERE NOT processed LIMIT 1000
FOR UPDATE SKIP LOCKED
)
UPDATE vac SET processed = true
```

```
WHERE id IN (SELECT id FROM batch);
UPDATE 1000

=> SELECT pg_size_pretty(pg_table_size('vac'));
pg_size_pretty
----------------
7072 kB
(1 행)
```

Part II

버퍼 캐시와 WAL

9 장 버퍼 캐쉬Buffer Cache

9.1 캐싱Caching

현대의 컴퓨팅 시스템에서는 하드웨어와 소프트웨어 모두에서 캐싱이 사용됩니다. 프로세서에는 일반적으로 세 개 또는 네 개의 캐시 레벨이 있을 수 있으며, `RAID` 컨트롤러와 디스크에도 캐시가 추가로 존재합니다.

캐싱의 주요 목적은 빠른 메모리와 느린 메모리 사이의 성능 차이를 보완하는 것입니다. 빠른 메모리는 비용이 많이 들고 용량이 작으며, 느린 메모리는 용량이 크고 비용이 저렴합니다. 이에 따라, 빠른 메모리가 느린 메모리에 저장된 모든 데이터를 수용할 수 없습니다. 그러나 대체로, 특정 시점에는 전체 데이터 중 **일부분**만 활발하게 사용됩니다. 이러한 상황에서 일부 빠른 메모리를 **캐시**로 활용하여 자주 사용되는 핫 데이터를 유지하면, 느린 메모리 접근에 따른 부하를 크게 줄일 수 있습니다.

PostgreSQL에서는 버퍼 캐시[83]가 관계 페이지를 보유하여 디스크(밀리초)와 `RAM`(나노초)의 액세스 시간을 균형있게 조절합니다.

운영 체제에는 동일한 목적을 위한 자체 캐시가 있습니다. 이러한 이유로 데이터베이스 시스템은 일반적으로 중복 캐싱을 피하기 위해 디스크에 저장된 데이터를 일반적으로 직접 쿼리하여 `OS` 캐시를 우회합니다. 그러나 PostgreSQL은 다른 접근 방식을 사용합니다. PostgreSQL은 모든 데이터를 버퍼링된 파일 작업을 통해 읽고 씁니다.

> 이중 캐싱은 직접 direct I/O 를 적용하면 피할 수 있습니다. 직접 I/O 를 사용하면 버퍼링된 페이지를 OS 주소 공간으로 복사하는 대신 PostgreSQL 이 직접 메모리에 접근(DMA)하여 오버헤드를 줄일 수 있습니다. 또한 디스크에 관한 물리적인 쓰기에 관한 즉각적인 제어를 얻을 수 있습니다. 그러나 버퍼화에 의해 가능한 데이터 프리페치 prefetching 를 지원하지 않는다는 단점이 있으므로 비동기 asynchronous I/O 를 통해 따로 구현해야 합니다. 이는 PostgreSQL 코어에서 대규모 코드 수정 및 직접 및 비동기 I/O 지원과 관련된 OS 호환성 처리를 필요로 합니다. 그러나 비동기 통신이 설정되면 대기 없는 디스크 액세스의 추가적인 이점을 누릴 수 있습니다.
>
> PostgreSQL 커뮤니티는 이미 이러한 주요 작업[84]을 시작했지만 실제 결과가 나타나기까지는 시간이 걸릴 것입니다.

9.2 버퍼 캐시 설계

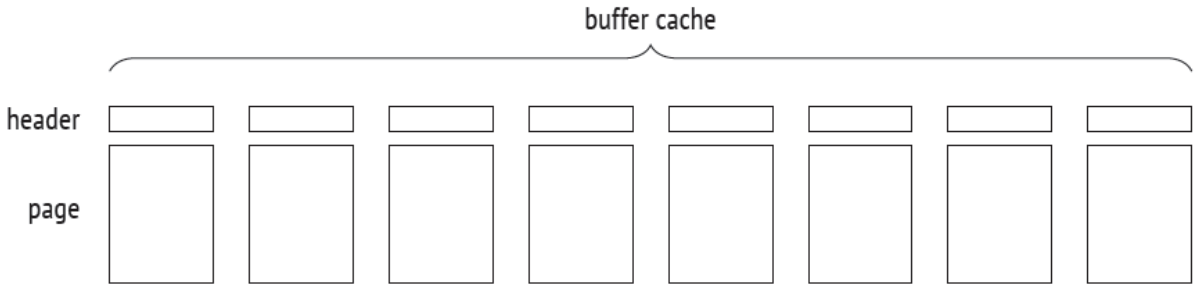
버퍼 캐시는 서버의 공유 메모리 영역에 위치해 있어 모든 프로세스가 접근 가능합니다. 이는 공유 메모리의 대부분을 차지하며, PostgreSQL에서 가장 중요하고 복잡한 데이터 구조 중 하나입니다. 캐시의 작동 방식을 이해하는 것은 매우 중요하며, 하위 트랜잭션, `CLOG` 트랜잭션 상태, 그리고 `Write-Ahead Logging(WAL)` 항목 등 많은 다른 구조들이 유사한 캐싱 메커니즘을 사용하기 때문에 더욱 중요합니다. 이들은 상대적으로 더 간단한 형태를 가지고 있습니다.

---

[83] backend/storage/buffer/README

[84] www.postgresql.org/message-id/flat/20210223100344.llw5an2aklengrmn%40alap3.anarazel.de

이 캐시의 이름은 그 내부 구조에서 따왔으며, 버퍼의 배열로 구성되어 있습니다. 각 버퍼는 데이터 페이지와 관련 헤더를 포함할 수 있는 메모리 청크를 예약해 둡니다.[85]



헤더에는 버퍼와 관련된 페이지에 관한 다양한 정보가 포함되어 있습니다. 예를 들면 다음과 같습니다:

- 페이지의 물리적 위치: 이는 파일 `ID`, 포크, 그리고 포크 내의 블록 번호를 포함합니다.
- 더티 페이지: 페이지의 데이터가 변경되어, 디스크에 빠르게 다시 쓰여져야 함을 나타내는 속성입니다.
- 버퍼 사용 횟수: 버퍼가 얼마나 자주 사용되었는지를 나타냅니다.
- 핀 카운트 또는 참조 카운트: 이는 버퍼가 현재 얼마나 많이 참조되고 있는지를 나타냅니다.

프로세스가 관계 데이터 페이지에 접근하려면 먼저 버퍼 매니저[86]에게 요청을 보내야 합니다. 그러면 버퍼 매니저는 해당 페이지를 가지고 있는 버퍼의 `ID`를 프로세스에 주게 됩니다. 이후 프로세스는 필요에 따라 캐시된 데이터를 읽거나 캐시 내에서 데이터를 직접 수정하게 됩니다. 이때, 페이지가 사용 중일 때는 해당 버퍼가 **고정**pinned 상태가 됩니다. 고정이란, 캐시된 페이지가 제거되지 못하게 하는 상태를 말하며, 다른 잠금과 함께 적용될 수 있습니다. 페이지가 한 번 고정될 때마다 그 사용 횟수가 증가하게 됩니다. 주의할 점은, 페이지가 캐시에 있는 상태에서는 파일 작업 없이 사용할 수 있다는 것입니다. 이렇게 되면 파일 접근 시간을 줄일 수 있어 효율적입니다.

`pg_buffercache` 확장 기능을 사용하여 버퍼 캐시를 탐색할 수 있습니다:

```
=> CREATE EXTENSION pg_buffercache;
```

테이블을 생성하고 행을 삽입해 봅시다:

```
=> CREATE TABLE cacheme(
   id integer
) WITH (autovacuum_enabled = off);
=> INSERT INTO cacheme VALUES (1);
```

이제 버퍼 캐시에는 새로 삽입된 행이 포함된 힙 페이지가 있습니다. 특정 테이블과 관련된 모든 버퍼를 선택하여 직접 확인할 수 있습니다. 이와 같은 쿼리가 다시 필요하므로 함수로 래핑해 봅시다:

```
=> CREATE FUNCTION buffercache(rel regclass)
```

[85] include/storage/buf_internals.h

[86] backend/storage/buffer/bufmgr.c

```
RETURNS TABLE(
   bufferid integer, relfork text, relblk bigint,
   isdirty boolean, usagecount smallint, pins integer
) AS $$
SELECT bufferid,
       CASE relforknumber
        WHEN 0 THEN 'main'
        WHEN 1 THEN 'fsm'
        WHEN 2 THEN 'vm'
      END,
      relblocknumber,
      isdirty,
      usagecount,
      pinning_backends
FROM pg_buffercache
WHERE relfilenode = pg_relation_filenode(rel)
ORDER BY relforknumber, relblocknumber;
$$ LANGUAGE sql;

=> SELECT * FROM buffercache('cacheme');
 bufferid ¦ relfork ¦ relblk ¦ isdirty ¦ usagecount ¦ pins
----------+---------+--------+---------+------------+------
      268 ¦    main ¦      0 ¦       t ¦          1 ¦ 0
(1 행)
```

9.3 캐시 히트

버퍼 매니저가 페이지를 읽어야 할 때[87], 먼저 버퍼 캐시를 확인합니다. 이는 캐시에서 먼저 원하는 데이터를 찾아보는 과정이며, 이를 통해 파일 접근 시간을 줄일 수 있습니다. 또한, 모든 버퍼 `ID`는 해시 테이블[88]에 저장되어 있습니다. 해시 테이블은 키-값 쌍을 저장하는 데이터 구조로, 버퍼 `ID`를 키로 사용하여 값을 빠르게 검색할 수 있게 해줍니다. 이런 접근 방식은 검색 속도를 높이는 데 큰 역할을 합니다.

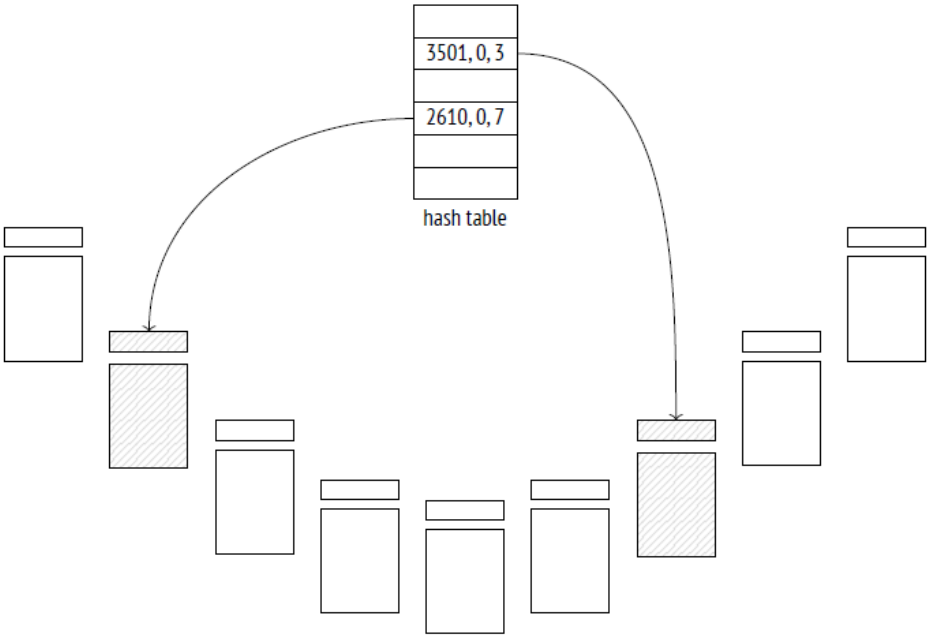
> 대부분의 현대 프로그래밍 언어들은 해시 테이블을 기본 데이터 유형 중 하나로 사용합니다. 해시 테이블은 '연관 배열'이라고도 불리며, 사용자가 보기에는 일반 배열과 비슷한 모습을 보입니다. 하지만, 해시 테이블의 인덱스인 **해시 키**는 정수뿐만 아니라 텍스트 문자열과 같은 모든 종류의 데이터 유형이 될 수 있습니다.
>
> 가능한 키 값의 범위는 매우 클 수 있지만, 해시 테이블이 한 번에 너무 많은 다른 값을 포함하지는 않습니다. 해싱의 핵심 아이디어는 **해시 함수**를 사용해 키 값을 정수로 변환하는 것입니다. 이 변환된 숫자(또는 그 일부분의 비트)는 일반 배열의 인덱스로 사용됩니다. 이 배열의 각 요소를 '해시 테이블 버킷'이라고 부릅니다.

[87] backend/storage/buffer/bufmgr.c, ReadBuffer_common function

[88] backend/storage/buffer/buf_table.c

좋은 해시 함수는 해시 키를 버킷 사이에 균일하게 분배하는 역할을 합니다. 그러나 동일한 숫자를 다른 키에 할당하면서 같은 버킷에 배치할 수도 있습니다. 이런 현상을 **충돌** collision 이라고 합니다. 이 때문에 값은 해시 키와 함께 버킷에 저장되어야 하고, PostgreSQL 은 해시 키를 사용하여 해시 된 값을 찾기 위해 버킷의 모든 키를 확인해야 합니다.

해시 테이블에는 여러 가지 구현 방법이 있습니다. 그 중 버퍼 캐시는 체이닝을 통해 해시 충돌을 해결하는 확장 가능한 테이블을 사용합니다.[89] 해시 키는 관계 파일의 `ID`, 포크의 유형 및 해당 포크 파일 내 페이지의 ID로 구성됩니다. 따라서 페이지를 알면 PostgreSQL은 해당 페이지를 포함하는 버퍼를 빠르게 찾거나 페이지가 현재 캐시되어 있는지 확인할 수 있습니다.



버퍼 캐시의 구현이 해시 테이블에 의존하는 점은 오랫동안 여러 비판을 받아왔습니다. 이 구조는 특정 관계의 페이지를 사용하는 모든 버퍼를 찾는 데 사용되지 않습니다. 이런 구조는 `DROP` 이나 `TRUNCATE` 명령을 실행하거나, `VACUUM` 작업 중에 테이블을 잘라내야 하거나, 캐시에서 페이지를 제거해야 하는 상황에서 필요하게 됩니다. 그러나 아직까지 이러한 문제에 관한 적절한 해결책이 제시되지 않았습니다. 그래서 현재로서는 해시 테이블에 의존하는 구조를 계속 사용하고 있습니다.

해시 테이블에 필요한 버퍼 `ID`가 들어있다면, 버퍼 매니저는 해당 버퍼를 고정하고 그 `ID`를 프로세스에 반환합니다. 그러면 이 프로세스는 `I/O` 트래픽을 발생시키지 않고도 캐시된 페이지를 사용할 수 있게 됩니다.

버퍼를 고정하기 위해 PostgreSQL은 해당 버퍼 헤더의 고정 카운터를 증가시킵니다. 한 번에 여러 프로세스가 버퍼를 고정할 수 있습니다. 고정 카운터가 0보다 크다면 그 버퍼는 사용 중이라고 간주되며, 그 내용은 크게 변경될 수 없습니다. 예를 들어, 새로운 튜플은 나타날 수 있지만(가시성 규칙에 따라 보이지 않을 수 있음), 페이지 자체는 교체되지 않습니다.

[89] backend/utils/hash/dynahash.c

`analyze`와 `buffers` 옵션을 사용하여 실행하는 `EXPLAIN` 명령은, 표시된 쿼리 계획을 실행하고 사용된 버퍼의 수를 보여줍니다:

예를 들어, 다음과 같이 실행할 수 있습니다:

```
=> EXPLAIN (analyze, buffers, costs off, timing off, summary off)
SELECT * FROM cacheme;
QUERY PLAN
──────────────────────────────────────────────────
Seq Scan on cacheme (actual 행s=1 loops=1)
  Buffers: shared hit=1
Planning:
  Buffers: shared hit=12 read=7
(4 행s)
```

여기서 hit=1은 읽어야 할 유일한 페이지가 캐시에서 찾아졌다는 것을 의미합니다.
버퍼 고정은 사용 횟수를 하나 증가시킵니다:

```
=> SELECT * FROM buffercache('cacheme');
      bufferid  ¦       relfork ¦      relblk ¦      isdirty ¦         usagecount ¦ pins
────────────+───────────+─────────+───────────+──────────────+────
            268 ¦          main ¦           0 ¦            t  ¦                  2 ¦ 0
(1 행)
```

쿼리 실행 중에 버퍼 고정을 확인해보려면, 커서를 열어보세요 - 결과 세트에서 다음 행에 빠른 접근을 제공해야 하므로 버퍼 핀을 유지합니다:

```
=> BEGIN;
=> DECLARE c CURSOR FOR SELECT * FROM cacheme;
=> FETCH c;
  id
────
  1
(1 행)
=> SELECT * FROM buffercache('cacheme');
       bufferid ¦       relfork ¦      relblk ¦      isdirty ¦         usagecount ¦ pins
────────────+───────────+─────────+───────────+──────────────+────
            268 ¦          main ¦           0 ¦            t ¦                  3 ¦ 1
(1 행)
```

프로세스가 고정된 버퍼를 사용할 수 없는 경우, 보통 그것을 건너뛰고 다른 것을 선택합니다. 테이블 베큠작업 동안 이를 확인할 수 있습니다:

```
=> VACUUM VERBOSE cacheme;
INFO: vacuuming "public.cacheme"
INFO: table "cacheme": found 0 removable, 0 nonremovable 행
versions in 1 out of 1 pages
DETAIL: 0 dead 행 versions cannot be removed yet, oldest xmin:
877
Skipped 1 page due to buffer pins, 0 frozen pages.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```

페이지의 튜플이 고정된 버퍼에서 물리적으로 제거될 수 없기 때문에, 해당 페이지는 건너뛰어집니다. 하지만 특정 버퍼가 반드시 필요한 경우, 프로세스는 대기열에 들어가고 해당 버퍼에 관한 독점적인 접근을 기다립니다. 이런 작업의 예시로는 프리징이 포함된 베큠 작업이 있습니다.[90]

커서가 닫히거나 다른 페이지로 이동하면 버퍼는 고정 해제됩니다. 이 예시에서는 트랜잭션 종료 시점에서 이루어집니다:

```
=> COMMIT;
=> SELECT * FROM buffercache('cacheme');
       bufferid ¦       relfork ¦      relblk ¦      isdirty ¦         usagecount ¦ pins
————————————+———————————+———————————+———————————+————————————————+————
            268 ¦          main ¦           0 ¦           t ¦                  3 ¦ 0
            310 ¦            vm ¦           0 ¦           f ¦                  2 ¦ 0
(2 행s)
```

페이지 수정도 같은 고정 메커니즘으로 보호됩니다. 예를 들어, 테이블에 새 행을 삽입해 보겠습니다(같은 페이지에 들어갈 것입니다):

```
=> INSERT INTO cacheme VALUES (2);
=> SELECT * FROM buffercache('cacheme');
       bufferid¦       relfork ¦      relblk ¦      isdirty ¦         usagecount ¦ pins
————————————+———————————+———————————+———————————+————————————————+————
            268 ¦          main ¦           0 ¦           t ¦                  4 ¦ 0
            310 ¦            vm ¦           0 ¦           f ¦                  2 ¦ 0
(2 행s)
```

PostgreSQL은 디스크에 즉시 기록하지 않습니다. 페이지는 일정 시간 동안 버퍼 캐시에서 더티dirty 상태로 유지되어, 읽기와 쓰기 모두에 관한 성능 향상을 제공합니다.

9.4 캐시 미스

해시 테이블에 쿼리된 페이지와 관련된 항목이 없으면 해당 페이지가 캐시되지 않았다는 의미입니다. 이 경우 새 버퍼가 할당(및 즉시 고정)되고 페이지가 이 버퍼로 읽혀지고 해시 테이블 참조가 그에 따라 수정됩니

[90] backend/storage/buffer/bufmgr.c, LockBufferForCleanup function

다.

캐시를 확인하기 위해 인스턴스를 다시 시작하겠습니다:

```
postgres$ pg_ctl restart -l /home/postgres/logfile
```

페이지를 읽으려 하면 캐시 미스가 발생하고 페이지가 새 버퍼로 로드됩니다:

```
=> EXPLAIN (analyze, buffers, costs off, timing off, summary off)
SELECT * FROM cacheme;
QUERY PLAN
-------------------------------------------
Seq Scan on cacheme (actual 행s=2 loops=1)
Buffers: shared read=1 dirtied=1
Planning:
Buffers: shared hit=15 read=7
(4 행s)
```

쿼리 계획은 이제 히트 대신 읽기 상태를 표시하는데, 이는 캐시 미스를 나타냅니다. 또한 쿼리가 일부 힌트 비트를 수정했기 때문에 이 페이지가 더티 상태가 되었습니다.

버퍼 캐시 쿼리는 새로 추가된 페이지의 사용 횟수가 1로 설정되었음을 보여줍니다:

```
=> SELECT * FROM buffercache('cacheme');
 bufferid ¦ relfork ¦ relblk ¦ isdirty ¦ usagecount ¦ pins
----------+---------+--------+---------+------------+------
       98 ¦    main ¦      0 ¦       t ¦          1 ¦ 0
(1 행)
```

`pg_statio_all_tables` 뷰에는 테이블별 버퍼 캐시 사용에 관한 전체 통계가 포함됩니다:

```
=> SELECT heap_blks_read, heap_blks_hit
FROM pg_statio_all_tables
WHERE relname = 'cacheme';
 heap_blks_read ¦ heap_blks_hit
----------------+---------------
              2 ¦ 5
(1 행)
```

PostgreSQL은 인덱스와 시퀀스에 대해 유사한 뷰를 제공합니다. `I/O` 작업에 관한 통계도 표시할 수 있지만 `track_io_timing`(기본값: `OFF`)이 활성화된 경우에만 표시됩니다.

버퍼 검색과 제거

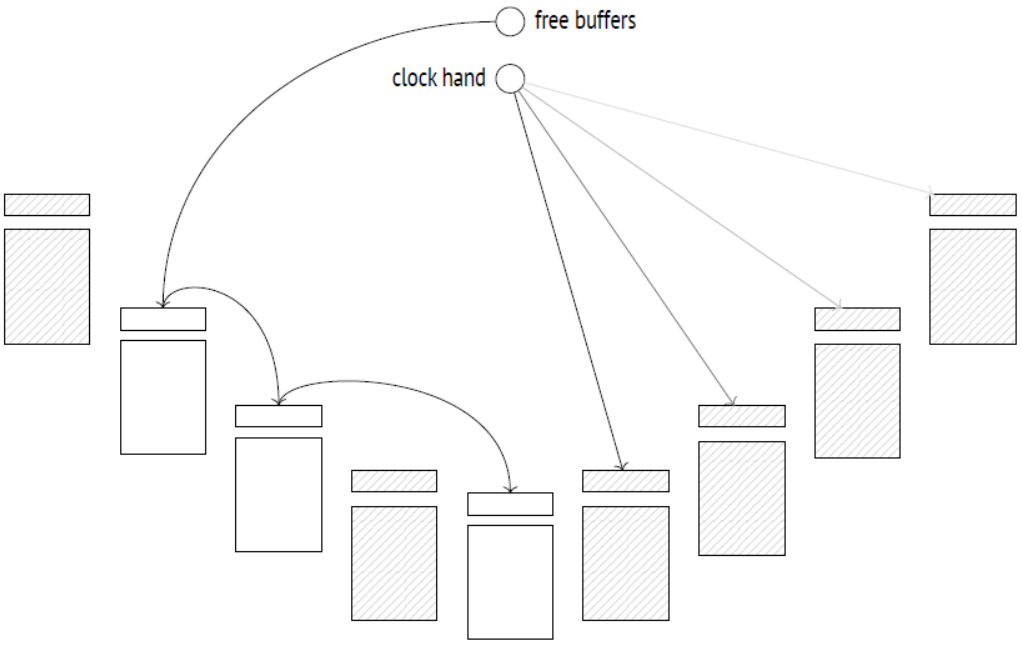
페이지에 관한 버퍼 선택은 간단하지 않은 작업입니다.[91] 이에는 두 가지 주요한 시나리오가 있습니다:

1. 서버가 시작된 직후에는 모든 버퍼가 비어 있고 리스트에 바인딩되어 있습니다. 이후에 디스크로부터 읽은 페이지는 여분의 버퍼가 아직 남아있는 동안 첫 번째 버퍼를 차지하고 리스트에서 제거됩니다. 버퍼가 다른 페이지로 대체되지 않고 사라지면 다시 리스트[92]에 포함될 수 있습니다. 이런 상황은 `DROP`이나 `TRUNCATE` 명령을 호출하거나, 테이블이 베큠 과정에서 잘려나갈 때 발생할 수 있습니다.
2. 언젠가는 여분의 버퍼가 더 이상 남아있지 않게 됩니다. 이는 주로 데이터베이스의 크기가 캐시에 할당된 메모리 크기보다 클 때 발생합니다. 이런 경우에는 버퍼 매니저가 이미 사용 중인 버퍼 하나를 선택하고, 해당 버퍼에서 캐시된 페이지를 제거해야 합니다. 이 작업은 `clock sweep` 알고리즘을 사용하여 수행됩니다. `clock sweep` 알고리즘은 버퍼 캐시를 돌며 시계 손을 이동시키고, 각 캐시된 페이지의 사용 횟수를 하나씩 줄입니다. 시계 손이 찾은 첫 번째 고정되지 않은 사용 횟수가 0인 버퍼가 클리어됩니다.

버퍼가 접근(고정)될 때마다 사용 횟수가 증가하고, 버퍼 매니저가 페이지를 제거하기 위해 검색할 때마다 사용 횟수가 감소하는 방식으로 작동합니다. 이로 인해 가장 최근에 사용되지 않은 페이지가 가장 먼저 제거되고, 더 자주 접근된 페이지는 캐시에 더 오래 유지됩니다.

모든 버퍼의 사용 횟수가 0이 아닌 경우, 시계 손은 최종적으로 0 값을 찾기 전에 한 바퀴 이상 돌아야 합니다. 그러나 여러 라운드를 실행하는 것을 피하기 위해 포스트그래스큐엘은 사용 횟수를 5로 제한합니다.

제거할 버퍼를 찾게 되면, 그 버퍼에 있는 페이지에 관한 해시 테이블의 참조를 제거해야 합니다. 그러나 해당 버퍼가 더티 상태인 경우, 즉 수정된 데이터가 포함되어 있는 경우, 기존 페이지를 단순히 버릴 수는 없습니다. 그럴 경우 버퍼 매니저는 해당 페이지를 먼저 디스크에 기록해야 합니다.



[91] backend/storage/buffer/freelist.c, StrategyGetBuffer function

[92] backend/storage/buffer/freelist.c, StrategyFreeBuffer function

그 다음, 버퍼 매니저는 새로운 페이지를 찾은 버퍼로 읽어들입니다. 이 때, 버퍼가 클리어되어야 하는지 아니면 여전히 비어 있는지 여부는 중요하지 않습니다. 이 과정에서는 버퍼드buffered `I/O`를 사용하여 페이지를 읽어들입니다. 이는 운영 체제의 캐시에서 페이지를 찾을 수 없는 경우에만 디스크에서 읽어들이게 됩니다.

> 직접 `I/O`를 사용하고 운영 체제 캐시에 의존하지 않는 데이터베이스 시스템은 논리적인 읽기(즉, `RAM`에서 또는 버퍼 캐시에서의 읽기)와 물리적인 읽기(즉, 디스크에서 읽기)를 구분합니다. PostgreSQL 의 관점에서 보면, 페이지는 버퍼 캐시에서 읽힐 수도 있고 운영 체제에 의해 요청될 수도 있습니다. 그러나 후자의 경우에는 해당 페이지가 `RAM`에서 찾아진 것인지, 아니면 디스크에서 읽어진 것인지를 확인할 방법은 없습니다.

해시 테이블은 새로운 페이지를 참조하도록 수정되고, 해당 버퍼는 고정됩니다. 사용 횟수는 증가하여 현재 1로 설정되며, 이는 시계 손이 버퍼 캐시를 통과하는 동안 이 값이 증가할 수 있는 기회를 제공합니다.

9.5 대량 제거

대량의 읽기나 쓰기 작업이 수행될 때, 일시적인 데이터가 버퍼 캐시에서 유용한 페이지를 빠르게 대체하게 되는 위험이 존재합니다.

이러한 상황을 방지하기 위해, 대량 작업들은 상대적으로 **작은 버퍼 링**을 사용합니다. 그리고 제거 작업은 이 작은 범위 내에서만 수행되므로, 다른 버퍼에는 영향을 끼치지 않습니다. 이런 방식으로, 버퍼 캐시 내의 중요한 데이터를 보호하면서도 대량의 데이터 처리를 가능하게 합니다.

> "버퍼 링"과 함께 코드에서는 "링 버퍼"라는 용어도 사용됩니다. 그러나 이 동의어는 상당히 모호합니다. 왜냐하면 링 버퍼 자체가 여러 개의 버퍼(버퍼 캐시에 속하는)로 구성되기 때문입니다. 이런 면에서 "버퍼 링"이 더 정확한 용어입니다.

특정 크기의 버퍼 링은 연속적으로 사용되는 버퍼 배열로 구성됩니다. 초기에는 이 버퍼 링은 비어 있으며, 개별 버퍼는 일반적인 방식으로 버퍼 캐시에서 선택되어 하나씩 버퍼 링에 추가됩니다. 그런 다음, 제거 작업이 시작되지만, 이 작업은 링의 범위 내에서만 수행됩니다.[93]

버퍼 링에 추가된 버퍼는 버퍼 캐시에서 제외되지 않으며, 다른 작업에서 계속 사용할 수 있습니다. 따라서 재사용할 버퍼가 고정되어 있거나 사용 횟수가 1보다 높은 경우, 해당 버퍼는 단순히 링에서 분리되고 다른 버퍼로 교체됩니다.

PostgreSQL은 세 가지 제거 전략을 지원합니다.

대용량 테이블의 순차 스캔에 대해선, '**대량 읽기 전략**'이 사용됩니다. 이는 이전 버퍼 캐시의 1/4를 초과하는 경우에 적용되며, 링 버퍼는 256k (32개의 일반 페이지)를 사용합니다. 이 전략에서는 버퍼를 해제하여 더티 페이지를 디스크에 기록하는 것이 허용되지 않습니다. 대신, 버퍼는 링에서 제외되고 다른 버퍼로 교체됩니다. 결과적으로, 쓰기 작업이 완료될 때까지 읽기 작업을 기다릴 필요가 없어, 읽기 속도가 향상됩니다. 또한, 테이블이 이미 스캔 중인 경우, 다른 스캔을 시작하는 프로세스는 기존의 버퍼 링에 참여하며 현재 사

[93] backend/storage/buffer/freelist.c, GetBufferFromRing function

용 가능한 데이터에 접근할 수 있습니다. 이로써 추가적인 `I/O` 작업이 발생하지 않습니다.[94]

'**대량 쓰기 전략**'은 `COPY FROM`, `CREATE TABLE AS SELECT`, `CREATE MATERIALIZED VIEW` 명령뿐 아니라, `ALTER TABLE` 명령 중 테이블 재작성을 일으키는 명령에 적용됩니다. 이 전략에서 할당된 링은 16MB (2048개의 일반 페이지)로 상당히 큰데, 이 크기는 버퍼 캐시의 총 크기의 1/8을 초과하지 않습니다.

`VACUUM` 프로세스가 가시성 맵을 고려하지 않고 전체 테이블 스캔을 수행하는 경우, '**베큠 전략**'이 사용됩니다. 이 전략에서 링 버퍼는 256k (32개의 일반 페이지)의 `RAM`을 할당받습니다.

버퍼 링은 항상 원치 않는 제거를 방지하는 것은 아닙니다. `UPDATE`나 `DELETE` 명령이 많은 행에 영향을 미치는 경우, 수행된 테이블 스캔은 대량 읽기 전략이 적용되지만, 페이지가 지속적으로 수정되기 때문에 버퍼 링은 실질적으로 효용을 잃게 됩니다.

또 다른 예로는 `TOAST` 테이블에 대용량 데이터를 저장하는 경우를 들 수 있습니다. 읽어야 할 데이터의 양이 많을 수 있지만, `TOAST`된 값을 접근할 때는 항상 인덱스를 통해 이루어지므로 버퍼 링은 우회됩니다.

이제 '대량 읽기 전략'에 대해 좀 더 자세히 살펴보겠습니다. 간단하게 하기 위해, 삽입된 행이 전체 페이지를 차지하도록 테이블을 생성합니다. 기본적으로 버퍼 캐시의 크기는 16,384 페이지이며, 각 페이지의 크기는 8k입니다. 따라서 스캔에서 버퍼 링을 사용하려면 테이블이 4,096 페이지 이상을 차지해야 합니다.

```
=> CREATE TABLE big(
id integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
s char(1000)
) WITH (fillfactor = 10);

=> INSERT INTO big(s)
SELECT 'FOO' FROM generate_series(1,4096+1);
```

테이블을 분석해 보겠습니다.

```
=> ANALYZE big;
=> SELECT relname, relfilenode, relpages
FROM pg_class
WHERE relname IN ('big', 'big_pkey');
  relname ¦ relfilenode ¦ relpages
----------+-------------+----------
      big ¦       16546 ¦ 4097
 big_pkey ¦       16551 ¦ 14
(2 행s)
```

서버를 재시작하여 캐시를 비우는 것은 분석 중인 힙 페이지가 캐시에 남아있어서 기존 데이터에 의해 결과가 왜곡될 가능성을 제거하는 좋은 방법입니다. 이렇게 함으로써, 새로운 분석이 이전 작업의 영향을 받지 않

[94] backend/access/common/syncscan.c

고, 더 정확한 결과를 얻을 수 있습니다.

```
postgres$ pg_ctl restart -l /home/postgres/logfile
```

서버를 재시작한 후 전체 테이블을 읽어보겠습니다.

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT id FROM big;
QUERY PLAN
------------------------------------------
Seq Scan on big (actual 행s=4097 loops=1)
(1 행)
```

힙 페이지는 총 32개의 버퍼를 차지하며, 이는 이 작업을 위한 버퍼 링을 구성합니다.

```
=> SELECT count(*)
FROM pg_buffercache
WHERE relfilenode = pg_relation_filenode('big'::regclass);
count
-------
32
(1 행)
```

하지만 인덱스 스캔의 경우에는 버퍼 링이 사용되지 않습니다.

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM big ORDER BY id;
QUERY PLAN
-------------------------------------------------------------
Index Scan using big_pkey on big (actual 행s=4097 loops=1)
(1 행)
```

결과적으로, 버퍼 캐시에는 전체 테이블과 전체 인덱스가 포함되게 됩니다.

```
=> SELECT relfilenode, count(*)
FROM pg_buffercache
WHERE relfilenode IN (
pg_relation_filenode('big'),
pg_relation_filenode('big_pkey')
)
GROUP BY relfilenode;
 relfilenode ¦ count
-------------+-------
       16546 ¦ 4097
       16551 ¦ 14
(2 행s)
```

9.6 버퍼 캐시 크기 선택하기

버퍼 캐시의 크기는 **shared_buffers** 매개변수 (기본값: 128MB)를 통해 지정됩니다. 기본값이 낮다는 것이 잘 알려져 있으므로, PostgreSQL 설치 후에 즉시 증가시키는 것이 좋습니다. 이 경우에는 서버를 다시 로드해야 합니다. 왜냐하면 공유 메모리는 서버 시작 시에 캐시에 할당되기 때문입니다.

그렇다면 적절한 값을 어떻게 결정할까요?

매우 큰 데이터베이스라도 동시에 활용되는 '핫 데이터' 집합은 제한적입니다. 이 집합은 버퍼 캐시에 들어갈 수 있어야 합니다(일회성 데이터를 위한 공간도 예약하는 것이 좋습니다). 캐시 크기가 너무 작으면 활발하게 사용되는 페이지들이 계속해서 제거되며, 이로 인해 과도한 `I/O` 작업이 발생할 수 있습니다. 그러나 캐시 크기를 무턱대고 늘리는 것은 좋은 전략이 아닙니다. `RAM`은 소중한 자원이며, 더 큰 캐시는 더 높은 유지보수 비용을 야기할 수 있습니다. 따라서, 버퍼 캐시 크기를 설정할 때는 데이터베이스의 크기와 활동성, 그리고 사용 가능한 `RAM` 등을 고려해야 합니다.

최적의 버퍼 캐시 크기는 시스템마다 다르며, 사용 가능한 메모리의 총 크기, 데이터 프로파일 및 작업 부하 유형과 같은 요소에 따라 달라집니다. 불행히도, 모두에게 동일하게 적합한 마법같은 값이나 공식은 없습니다.

또한 PostgreSQL에서 캐시 미스는 반드시 물리적인 `I/O` 작업을 발생시키지 않습니다. 버퍼 캐시가 매우 작은 경우, `OS` 캐시는 남은 빈 메모리를 사용하여 일부 정도로 문제를 완화할 수 있습니다. 그러나 데이터베이스와 달리 운영 체제는 읽은 데이터에 대해 아무것도 알지 못하므로 다른 제거 전략을 적용합니다.

일반적인 권장 사항은 `RAM`의 1/4로 시작한 다음 필요에 따라이 설정을 조정하는 것입니다.

가장 좋은 접근 방식은 실험입니다. 캐시 크기를 늘리거나 줄여서 시스템 성능을 비교할 수 있습니다. 당연히 이는 프로덕션 환경과 완전히 유사한 테스트 시스템이 있어야 하며, 일반적인 작업 부하를 재현할 수 있어야 합니다.

`pg_buffercache` 확장 기능을 사용하여 몇 가지 분석을 실행할 수도 있습니다. 예를 들어, 사용에 따른 버퍼 분포를 탐색할 수 있습니다:

```
=> SELECT usagecount, count(*)
FROM pg_buffercache
GROUP BY usagecount
ORDER BY usagecount;
        usagecount ¦ count
———————————————+———————
                1 ¦ 4128
                2 ¦ 50
                3 ¦ 4
                4 ¦ 4
                5 ¦ 73
                  ¦ 12125
(6 행s)
```

NULL 사용 횟수 값은 빈 버퍼에 해당합니다. 서버가 재시작되고 대부분의 시간이 유휴 상태로 유지되었기 때문에 이는 예상된 결과입니다. 사용된 버퍼의 대부분은 백엔드가 시스템 카탈로그 캐시를 채우고 쿼리를 수행하기 위해 읽은 시스템 카탈로그 테이블의 페이지를 포함합니다.

각 관계의 어느 정도가 캐시되었는지, 그리고 이 데이터가 핫 한지(사용 횟수가 1보다 큰 경우)를 확인할 수 있습니다:

```
=> SELECT c.relname,
          count(*) blocks,
          round( 100.0 * 8192 * count(*) / pg_table_size(c.oid) ) AS "% of rel",
          round( 100.0 * 8192 * count(*) FILTER (WHERE b.usagecount > 1) /
                                                    pg_table_size(c.oid) ) AS "% hot"
   FROM pg_buffercache b
   JOIN pg_class c ON pg_relation_filenode(c.oid) = b.relfilenode
   WHERE b.reldatabase IN ( 0, -- cluster-wide objects
                    (SELECT oid FROM pg_database WHERE datname = current_database())
                    )
     AND b.usagecount IS NOT NULL
   GROUP BY c.relname, c.oid
   ORDER BY 2 DESC
   LIMIT 10;
                            relname ¦      blocks ¦       % of rel ¦ % hot
————————————————————————————————————+————————————+————————————+————————
                                big ¦        4097 ¦             100 ¦ 1
                       pg_attribute ¦          30 ¦              48 ¦ 47
                           big_pkey ¦          14 ¦             100 ¦ 0
                            pg_proc ¦          13 ¦              12 ¦ 6
                        pg_operator ¦          11 ¦              61 ¦ 50
                           pg_class ¦          10 ¦              59 ¦ 59
                  pg_proc_oid_index ¦           9 ¦              82 ¦ 45
    pg_attribute_relid_attnum_index ¦           8 ¦              73 ¦ 64
     pg_proc_proname_args_nsp_index ¦           6 ¦              18 ¦ 6
                          pg_amproc ¦           5 ¦              56 ¦ 56
(10 행s)
```

이 예제에서는 큰 테이블과 해당 인덱스가 완전히 캐시되었지만, 해당 페이지들이 활발하게 사용되지는 않는다는 것을 보여줍니다.

다양한 각도로 데이터를 분석하면 유용한 통찰력을 얻을 수 있습니다. 그러나 pg_buffercache 쿼리를 실행할 때 다음과 같은 간단한 규칙을 따라야 합니다:

- 반환된 숫자는 어느 정도 변동할 수 있으므로 해당 쿼리를 여러 번 반복하세요.
- pg_buffercache 확장은 보는 버퍼를 잠그므로 지속적으로 실행하지 마세요.

9.7 캐시 워밍

서버를 재시작한 후에는 캐시가 활발하게 사용되는 데이터를 축적하는 데 시간이 필요합니다. 특정 테이블을 즉시 캐시하면 도움이 될 수 있고, 이를 위해 `pg_prewarm`확장 기능을 사용할 수 있습니다:

```
=> CREATE EXTENSION pg_prewarm;
```

이 확장 기능은 테이블을 버퍼 캐시(또는 `OS` 캐시만)로 불러오는 것뿐만 아니라, 현재 캐시 상태를 디스크에 저장하고 서버 재시작 후에 복원하는 기능도 제공합니다. 이 기능을 사용하려면, `shared_preload_libraries`에 이 확장 기능의 라이브러리를 추가하고 서버를 재시작해야 합니다:

```
=> ALTER SYSTEM SET shared_preload_libraries = 'pg_prewarm';
```

```
postgres$ pg_ctl restart -l /home/postgres/logfile
```

`pg_prewarm.autoprewarm`(기본값: ON) 설정이 변경되지 않은 경우, 서버가 다시 로드된 후에는 자동으로 `autoprewarm leader`라는 프로세스가 시작됩니다. `pg_prewarm.autoprewarm_interval`(기본값: 300초) 간격으로 이 프로세스는 캐시된 페이지 목록을 디스크로 플러시합니다(`max_parallel_processes` 슬롯 중 하나를 사용하여).

```
postgres$ ps -o pid,command --ppid `head -n 1 /usr/local/pgsql/data/postmaster.pid` |
grep prewarm
23124 postgres: autoprewarm leader
```

서버가 재시작되었으므로 큰 테이블은 더 이상 캐시되지 않습니다:

```
=> SELECT count(*)
FROM pg_buffercache
WHERE relfilenode = pg_relation_filenode('big'::regclass);
  count
────────
   0
(1 행)
```

이렇게 하면, 재시작 후에도 중요한 테이블들이 즉시 캐시될 수 있으며, 성능 저하를 최소화할 수 있습니다.

활발하게 사용되는 전체 테이블을 디스크 액세스로 인해 응답 시간이 받아들일 수 없을 것이라 예상될 경우, 해당 테이블을 미리 버퍼 캐시에 로드할 수 있습니다:

```
=> SELECT pg_prewarm('big');
  pg_prewarm
──────────────
        4097
(1 행)

=> SELECT count(*)
FROM pg_buffercache
WHERE relfilenode = pg_relation_filenode('big'::regclass);
```

```
  count
────────
    4097
(1 행)
```

페이지 목록은 PGDATA/autoprewarm.blocks 파일에 덤프됩니다. autoprewarm leader가 처음으로 완료될 때까지 기다릴 수 있지만, 우리는 수동으로 덤프를 시작할 것입니다:

```
=> SELECT autoprewarm_dump_now();
  autoprewarm_dump_now
──────────────────────
                  4224
(1 행)
```

덤프된 페이지 수가 4,097보다 큽니다. 이는 모든 사용된 버퍼가 고려되기 때문입니다. 이 파일은 텍스트 형식으로 작성되며, 데이터베이스, 테이블스페이스, 파일의 노드 및 fork, 세그먼트 번호를 포함합니다:

```
postgres$ head -n 10 /usr/local/pgsql/data/autoprewarm.blocks
<<4224>>
0,1664,1262,0,0
0,1664,1260,0,0
16391,1663,1259,0,0
16391,1663,1259,0,1
16391,1663,1259,0,2
16391,1663,1259,0,3
16391,1663,1249,0,0
16391,1663,1249,0,1
16391,1663,1249,0,2
```

서버를 다시 재시작하겠습니다.

```
postgres$ pg_ctl restart -l /home/postgres/logfile
```

테이블이 즉시 캐시에 표시됩니다:

```
=> SELECT count(*)
FROM pg_buffercache
WHERE relfilenode = pg_relation_filenode('big'::regclass);
  count
────────
   4097
(1 행)
```

다시 한 번 autoprewarm leader가 모든 예비 작업을 수행합니다. 이는 파일을 읽어, 페이지를 데이터베이스별로 정렬하고, 가능한 경우 디스크 읽기를 순차적으로 하기 위해 페이지를 재정렬한 뒤, autoprewarm worker에 전달하여 처리하는 작업을 포함합니다.

9.8 로컬 캐시

임시 테이블은 위에서 설명한 워크플로우를 따르지 않습니다. 임시 데이터는 단일 프로세스에게만 보여지기 때문에, 공유 버퍼 캐시에 로드하는 것은 큰 의미가 없습니다. 따라서 임시 데이터는 해당 테이블을 소유한 프로세스의 로컬 캐시를 사용합니다.[95]

일반적으로 로컬 버퍼 캐시는 공유 버퍼 캐시와 유사한 방식으로 작동합니다:

- 페이지 검색은 해시 테이블을 통해 수행됩니다.
- 제거는 표준 알고리즘을 따릅니다(버퍼 링은 사용되지 않음).
- 페이지는 제거를 방지하기 위해 고정될 수 있습니다.

그러나 로컬 캐시의 구현은 훨씬 단순합니다. 이는 버퍼에 관한 메모리 구조의 잠금(버퍼에는 단일 프로세스만 액세스 가능)이나 내결함성(임시 데이터는 세션 종료 시까지만 존재)을 처리할 필요가 없기 때문입니다. 이러한 특징 때문에 로컬 캐시는 특정 상황에서 더 효율적인 처리를 가능하게 합니다.

일반적으로 임시 테이블을 사용하는 세션은 몇 개 없으므로 로컬 캐시 메모리는 필요에 따라 할당됩니다. 세션에 할당된 로컬 캐시의 최대 크기는 `temp_buffers` 매개변수로 제한되며 기본값은 8`MB`입니다.
`temp_file_limit` 매개변수는 임시 테이블과는 관련이 없으며, 쿼리 실행 중에 생성될 수 있는 중간 데이터를 일시적으로 저장하는 데 사용되는 파일과 관련이 있습니다.

`EXPLAIN` 명령의 출력에서 로컬 버퍼 캐시에 관한 모든 호출은 공유 대신 로컬로 태그가 지정됩니다:

```
=> CREATE TEMPORARY TABLE tmp AS SELECT 1;
=> EXPLAIN (analyze, buffers, costs off, timing off, summary off)
SELECT * FROM tmp;
QUERY PLAN
──────────────────────────────────────────────
Seq Scan on tmp (actual 행s=1 loops=1)
  Buffers: local hit=1
Planning:
  Buffers: shared hit=12 read=7
(4 행s)
```

[95] backend/storage/buffer/localbuf.c

10 장 미리 쓰기 로그Write-Ahead Log

10.1 로깅

장애가 발생할 경우(예를 들어 전력 문제, 운영 체제 오류, 데이터베이스 서버 충돌 등), RAM에 있는 모든 정보는 사라지고, 디스크에 저장된 데이터만 남게 됩니다. 서버를 재시작하려면 데이터의 일관성을 복구해야 합니다. 만약 디스크 자체가 손상되었다면, 백업을 통해 문제를 해결해야 합니다.

이론적으로는 디스크에서 데이터의 일관성을 항상 유지할 수 있습니다. 하지만 실제로 서버는 디스크에 계속해서 무작위 페이지를 기록해야 하고(순차적인 쓰기가 더 저렴하더라도), 이 기록의 순서를 관리하여 특정 시점에서 데이터의 일관성이 손상되지 않도록 해야 합니다. 이는 특히 복잡한 인덱스 구조를 다룰 때 어려울 수 있습니다.

PostgreSQL과 같은 대부분의 데이터베이스 시스템은 다른 방식을 사용합니다. 서버가 작동하는 동안 현재 데이터의 일부는 RAM에서만 사용 가능하고, 영구 저장소로의 기록은 지연됩니다. 그래서 서버 작동 중에 디스크에 저장된 데이터는 항상 일관성을 가지지 않으며, 페이지가 한 번에 모두 저장되지 않습니다. 그러나 RAM에서 발생하는 각 변화, 예를 들어 버퍼 캐시에서 이루어진 페이지 수정은 로그로 기록됩니다. PostgreSQL은 필요한 경우에 반복할 수 있는 모든 중요한 정보를 포함하는 로그 항목을 생성합니다.[96]

페이지 수정과 관련된 로그 항목은 수정된 페이지 자체보다 먼저 디스크에 기록되어야 합니다. 그래서 이 로그의 이름은 미리. 쓰기 로그 또는 줄여서 WAL입니다. 이 요구 사항은. PostgreSQL이 장애가 발생했을 때 디스크에서 WAL 항목을 읽어 이미 완료된 작업을 다시 수행할 수 있도록 해줍니다. 이 작업은 아직 RAM에 있고, 충돌 전에 디스크로 전송되지 않은 완료된 작업의 결과입니다.

미리 쓰기 로그를 유지하는 것은 일반적으로 무작위 페이지를 디스크에 기록하는 것보다 효율적이라고 할 수 있습니다. WAL 항목들은 HDD에서도 처리 가능한 연속적인 데이터 스트림으로 구성되어 있습니다. 추가로, WAL 항목들은 페이지 크기보다 작은 경우가 많습니다.

장애가 발생했을 때 데이터 일관성을 깨뜨릴 수 있는 모든 작업은 로그로 기록되어야 합니다. 특히, 아래와 같은 작업이 WAL에 기록됩니다:

- 버퍼 캐시에서 이루어진 페이지 수정 - 기록이 지연되기 때문입니다.
- 트랜잭션 커밋과 롤백 - 상태 변경이 CLOG 버퍼에서 발생하고 즉시 디스크로 전송되지 않기 때문입니다.
- 파일 작업(테이블 추가/제거 시 파일 및 디렉터리 생성/삭제 등) - 이러한 작업은 데이터 변경과 동기화되어야 합니다.

다음과 같은 작업은 로그로 기록되지 않습니다:

- UNLOGGED 테이블에서의 작업
- 임시 테이블에서의 작업 - 수명이 생성한 세션에 의해 제한되기 때문입니다.

[96] postgresql.org/docs/14/wal-intro.html

> PostgreSQL 10 이전에는 해시 인덱스도 로그로 기록되지 않았습니다. 해시 인덱스의 주요 목적은 해시 함수를 다른 데이터 유형에 맞추는 것이었습니다.

장애 복구 외에도, WAL은 백업과 복제를 통해 특정 시점으로 복구하는데 사용될 수 있습니다.

10.2 WAL 구조

논리적 구조

WAL[97]은 가변적인 길이의 로그 항목 스트림으로, 논리적인 구조를 가지고 있습니다. 각 항목은 표준 헤더로 시작하며, 특정 작업과 관련된 **데이터**를 포함합니다. 헤더[98]는 다음과 같은 정보를 제공합니다:

- 해당 항목과 관련된 트랜잭션 ID
- 항목을 해석하는 자원 매니저[99]
- 데이터 손상을 감지하기 위한 체크섬
- 항목의 길이
- 이전 WAL 항목에 관한 참조

> 일반적으로 WAL 은 순방향으로 읽히지만, pg_rewind 같은 일부 유틸리티는 역방향으로 스캔할 수 있습니다.

WAL 데이터 자체는 다양한 형식과 의미를 가질 수 있습니다. 예를 들어, 특정 오프셋의 페이지 일부를 대체하는 페이지 조각일 수 있습니다. 해당 자원 매니저는 특정 항목을 어떻게 해석하고 재실행할지 알아야 합니다. 테이블, 다양한 인덱스 유형, 트랜잭션 상태 등 각각에 대해 별도의 자원 매니저가 있습니다.

WAL 파일은 서버의 공유 메모리에서 특별한 버퍼를 사용합니다. WAL이 사용하는 캐시의 크기는 wal_buffers(기본값: -1) 매개변수로 정의됩니다. 기본적으로 이 크기는 전체 버퍼 캐시 크기의 1/32로 자동으로 선택됩니다.

WAL 캐시는 버퍼 캐시와 매우 유사하나, 주로 링 버퍼 모드로 작동합니다. 새로운 항목은 앞쪽에 추가되고, 이전 항목은 꼬리에서 시작하여 디스크에 저장됩니다. 만약 WAL 캐시가 너무 작다면, 디스크 동기화가 필요한 것보다 더 자주 수행될 수 있습니다.

낮은 부하 상태에서는 삽입 위치(버퍼의 머리)가 대부분 이미 디스크에 저장된 항목의 위치(버퍼의 꼬리)와 동일합니다:

```
=> SELECT pg_current_wal_lsn(), pg_current_wal_insert_lsn();
 pg_current_wal_lsn ¦ pg_current_wal_insert_lsn
--------------------+----------------------------
         0/3DF56000 ¦ 0/3DF57968
(1 행)
```

---

97 postgresql.org/docs/14/wal-internals.html
backend/access/transam/README

98 include/access/xlogrecord.h

99 include/access/rmgrlist.h

> PostgreSQL 10 이전에는 모든 함수 이름에 `WAL` 대신 `XLOG` 약어가 포함되었습니다.

특정 항목을 참조하기 위해 PostgreSQL은 특별한 데이터 유형인 `pg_lsn`을 사용합니다. 이는 `WAL`의 시작부터 항목까지의 바이트 단위의 64비트 오프셋을 나타냅니다. `LSN`은 슬래시로 구분된 두 개의 32비트 숫자로 16진법으로 표시됩니다.

테이블을 생성해 봅시다:

```
=> CREATE TABLE wal(id integer);
=> INSERT INTO wal VALUES (1);
```

트랜잭션을 시작하고 `WAL` 삽입 위치의 `LSN`을 기록합니다:

```
=> BEGIN;
=> SELECT pg_current_wal_insert_lsn();
pg_current_wal_insert_lsn
---------------------------
0/3DF708D8
(1 행)
```

이제 임의의 명령을 실행해 봅시다. 예를 들어, 행을 수정 합니다:

```
=> UPDATE wal SET id = id + 1;
```

페이지 수정은 `RAM`의 버퍼 캐시에서 수행됩니다. 이 변경사항은 `RAM`의 `WAL` 페이지에 로그됩니다. 결과적으로, 삽입 `LSN`은 앞으로 이동합니다:

```
=> SELECT pg_current_wal_insert_lsn();
pg_current_wal_insert_lsn
---------------------------
0/3DF70920
(1 행)
```

수정된 데이터 페이지가 해당 `WAL` 항목 이후에 엄격하게 디스크에 플러시되도록 보장하기 위해, 페이지 헤더는 이 페이지와 관련된 최신 `WAL` 항목의 `LSN`을 저장합니다. 이 `LSN`은 `pageinspect`를 사용하여 확인할 수 있습니다:

```
=> SELECT lsn FROM page_header(get_raw_page('wal',0));
   lsn
-------------
0/3DF70920
(1 행)
```

전체 데이터베이스 클러스터에는 하나의 `WAL`만 존재하며, 새로운 항목들이 계속 추가됩니다. 이로 인해 페이지에 저장된 `LSN`이 이전에 `pg_current_wal_insert_lsn` 함수로 반환된 `LSN`보다 작을 수 있습니다. 그러나

시스템에서 아무런 변화가 없다면 이러한 숫자는 동일하게 될 것입니다.

이제 트랜잭션을 커밋해봅시다:

```
COMMIT;
```

커밋 작업도 로그로 기록되며, 삽입 LSN이 다시 변경됩니다:

```
SELECT pg_current_wal_insert_lsn();
pg_current_wal_insert_lsn
---------------------------
0/3DF70948
(1 행)
```

커밋은 CLOG 페이지에서 트랜잭션 상태를 업데이트하며, CLOG 페이지는 자체 캐시에 유지됩니다.[100] CLOG 캐시는 일반적으로 공유 메모리에서 128개의 페이지를 사용합니다.[101] CLOG 페이지가 해당 WAL 항목보다 먼저 디스크에 플러시되지 않도록 하기 위해, 최신 WAL 항목의 LSN을 CLOG 페이지에 대해서도 추적해야 합니다. 그러나 이 정보는 페이지 자체가 아닌 RAM에 저장됩니다.

특정 시점에 WAL 항목이 디스크에 기록될 것이며, 그 후에 CLOG와 데이터 페이지를 캐시에서 제거할 수 있습니다. 이들이 이전에 제거되어야 하는 경우 감지되었을 것이며, WAL 항목이 먼저 디스크로 강제로 기록되었을 것입니다.[102]

두 개의 LSN 위치를 알고 있다면, 간단히 한 위치에서 다른 위치를 빼면 (바이트 단위로) 두 LSN 사이의 WAL 항목 크기를 계산할 수 있습니다. 단순히 pg_lsn 유형으로 캐스팅하면 됩니다:

```
SELECT '0/3DF70948'::pg_lsn - '0/3DF708D8'::pg_lsn;
?column?
----------
112
(1 행)
```

이 경우 UPDATE 및 COMMIT 작업과 관련된 WAL 항목은 약 100바이트 정도였습니다.

이와 같은 방법을 통해, 특정 워크로드에서 생성되는 단위 시간당 WAL 항목의 양을 추정할 수 있습니다. 이 정보는 체크포인트 설정에 필요합니다.

물리적 구조

디스크에서 WAL은 별도의 파일, 또는 세그먼트로 PGDATA/pg_wal 디렉토리에 저장됩니다. 그들의 크기는 읽기 전용 wal_segment_size(기본값: 16MB) 매개변수로 확인할 수 있습니다.
높은 부하 시스템의 경우 세그먼트 크기를 늘리는 것이 좋을 수 있습니다. 이는 오버헤드를 줄일 수 있기 때

[100] backend/access/transam/slru.c

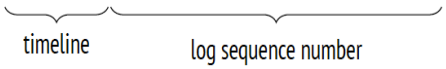
[101] backend/access/transam/clog.c, CLOGShmemBuffers function

[102] backend/storage/buffer/bufmgr.c, FlushBuffer function

문입니다. 하지만 이 설정은 클러스터 초기화시에만 변경할 수 있습니다(`initdb --wal-segsize`).
`WAL` 항목은 현재 파일에 공간이 없을 때까지 추가됩니다. 그 후 PostgreSQL은 새 파일을 시작합니다.

특정 항목이 어느 파일에 위치해 있으며, 파일 시작점에서 어떤 오프셋에 있는지 알아볼 수 있습니다:

```
=> SELECT file_name, upper(to_hex(file_offset)) file_offset
FROM pg_walfile_name_offset('0/3DF708D8');
                         file_name | file_offset
---------------------------------+-------------
              000000010000000000000003D | F708D8
```



파일 이름은 두 부분으로 구성됩니다. 가장 높은 8개의 16진수는 백업에서 복구를 위해 사용되는 타임라인을 정의하며, 나머지는 가장 높은 `LSN` 비트를 나타냅니다(가장 낮은 `LSN` 비트는 `file_offset` 필드에 표시됩니다).

현재의 `WAL` 파일을 보려면, 다음 함수를 호출할 수 있습니다:

```
=> SELECT *
FROM pg_ls_waldir()
WHERE name = '000000010000000000000003D';
                               name |          size | modification
---------------------------------+---------------+---------------------
              000000010000000000000003D |      16777216 | 2023-03-06 14:01:48+03
(1 행)
```

이제 `pg_waldump` 유틸리티를 사용하여 새롭게 생성된 `WAL` 항목의 헤더를 살펴보겠습니다. 이 유틸리티는 `LSN` 범위(예제처럼)나 특정 트랜잭션 `ID`로 `WAL` 항목을 필터링할 수 있습니다.
`pg_waldump` 유틸리티는 디스크의 `WAL` 파일에 접근해야 하므로, `postgres` `OS` 사용자로 실행되어야 합니다.

```
postgres$ /usr/local/pgsql/bin/pg_waldump \
-p /usr/local/pgsql/data/pg_wal -s 0/3DF708D8 -e 0/3DF70948#
rmgr: Heap len (rec/tot): 69/ 69, tx: 886, lsn:
0/3DF708D8, prev 0/3DF708B0, desc: HOT_UPDATE off 1 xmax 886 flags
0x40 ; new off 2 xmax 0, blkref #0: rel 1663/16391/16562 blk 0
rmgr: Transaction len (rec/tot): 34/ 34, tx: 886, lsn:
0/3DF70920, prev 0/3DF708D8, desc: COMMIT 2023-03-06 14:01:48.875861
MSK
```

여기서 두 개의 항목의 헤더를 확인할 수 있습니다.

첫 번째 항목은 `Heap` 자원 매니저가 처리하는 `HOT_UPDATE` 작업입니다. `blkref` 필드는 수정된 힙 페이지의 파일 이름과 페이지 번호를 보여줍니다:

```
SELECT pg_relation_filepath('wal');
```

```
pg_relation_filepath
------------------------
base/16391/16562
(1 행)
```

두 번째 항목은 `Transaction` 자원 매니저가 감독하는 `COMMIT` 작업입니다.

10.3 체크포인트Checkpoint

데이터의 일관성을 복구하기 위해, 즉 복구 작업을 수행하기 위해 PostgreSQL은 `WAL`을 순차적으로 재생하면서 손실된 변경 사항을 해당 페이지에 적용해야 합니다. 이를 위해 디스크에 저장된 페이지의 `LSN`과 `WAL` 항목의 `LSN`을 비교하여 손실된 내용을 파악합니다. 그런데 복구 작업은 어디서 시작해야 할까요? 너무 늦게 시작하면, 그 이전의 디스크에 기록된 페이지들은 모든 변경 사항을 반영받지 못하게 되어 데이터가 손상될 수 있습니다. 반면에 처음부터 시작하는 것은 현실적으로 불가능합니다. 이처럼 방관한 양의 데이터를 저장할 수 없을 뿐더러, 오랜 복구 시간을 허용할 수도 없기 때문입니다. 우리는 체크포인트를 점진적으로 전진시키며 안전하게 복구를 시작하고, 이전의 모든 `WAL` 항목을 제거할 수 있는 시점을 필요로 합니다.

가장 직접적인 체크포인트 생성 방법은 주기적으로 모든 시스템 작업을 일시 중지하고, 모든 변경된 페이지를 디스크에 강제로 저장하는 것입니다. 하지만 이 방식은 시스템이 무기한 중지되어 상당한 시간이 소요되므로, 실질적으로 적용하기 어렵습니다.

체크포인트는 시간적으로 분산되어 일종의 구간을 형성합니다. 이 작업은 `checkpointer`[103]라는 특별한 백그라운드 프로세스에 의해 수행됩니다.

체크포인트 시작. `checkpointer` 프로세스는 모든 것을 디스크에 즉시 기록합니다. 이때 `CLOG` 트랜잭션 상태, 하위 트랜잭션의 메타데이터, 그 외 몇 가지 구조 등이 포함됩니다.

체크포인트 실행. 대부분의 시간이 변경된 페이지를 디스크에 기록하는 데 사용됩니다.[104] 체크포인트 시작 시점에서 변경된 모든 버퍼의 헤더에는 특별한 태그가 설정되는데, 이 과정은 `I/O` 작업이 없어 매우 빠르게 진행됩니다.

그 후 `checkpointer`는 모든 버퍼를 검토하며, 태그가 설정된 버퍼를 디스크에 기록합니다. 이 페이지들은 캐시에서 제거되지 않고, 사용량과 핀 카운트는 무시되며, 단순히 기록됩니다.

가능한 한 페이지는 `ID` 순서대로 처리되어 임의적인 기록을 피합니다. 로드 밸런싱을 향상시키기 위해, PostgreSQL은 서로 다른 테이블스페이스 사이에서 작업을 번갈아 수행합니다(물리적 장치가 다를 수 있기 때문입니다).

백엔드가 백그라운드에서 작업 중인 경우, 태그가 지정된 버퍼를 디스크에 기록할 수도 있습니다. 어떤 경우든, 버퍼 태그는 이 단계에서 제거되어, 각 버퍼는 체크포인트를 위해 한 번만 기록됩니다.

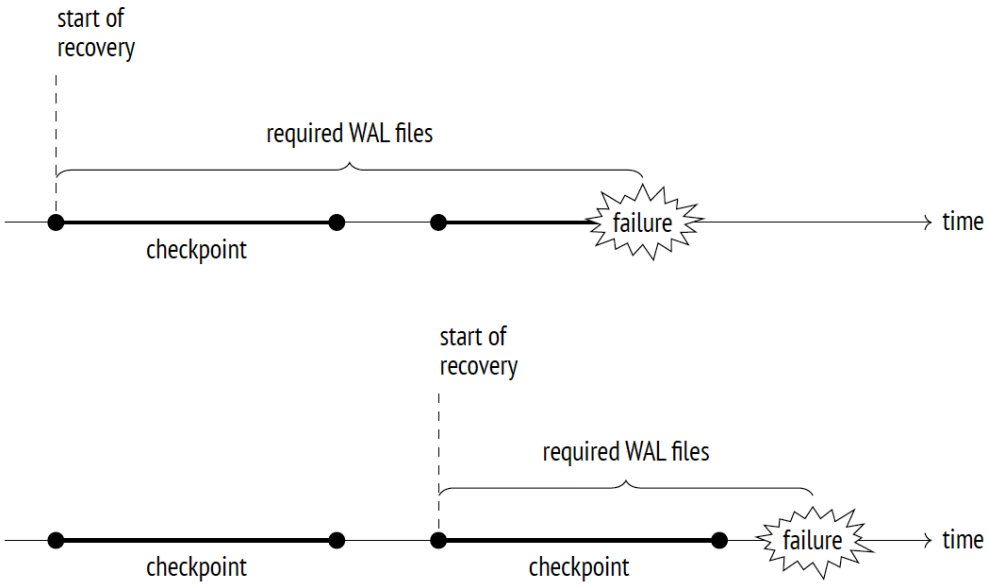
---

[103] backend/postmaster/checkpointer.c
backend/access/transam/xlog.c, CreateCheckPoint function

[104] backend/storage/buffer/bufmgr.c, BufferSync function

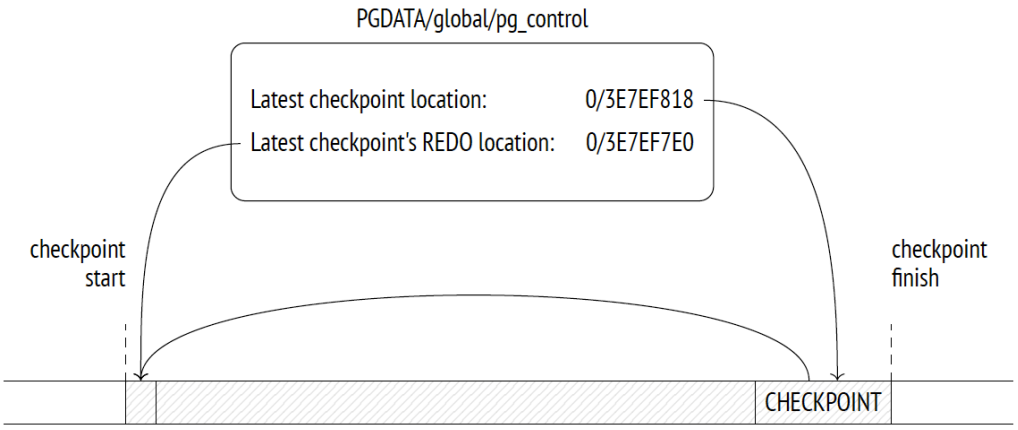
당연히, 체크포인트가 진행되는 동안에도 버퍼 캐시에서 페이지는 계속 수정될 수 있습니다. 그러나 새로 변경된 버퍼에는 태그가 지정되지 않으므로, `checkpointer`는 이를 무시합니다.

체크포인트 완료. 체크포인트 시작 시점에서 변경된 모든 버퍼가 디스크에 기록되면, 체크포인트는 완료된 것으로 간주됩니다. 이제부터 체크포트 시작점이 복구의 새로운 시작점으로 사용됩니다. 이 지점 이전에 기록된 모든 항목은 더 이상 필요하지 않습니다.



마지막 단계에서는, `checkpointer`가 체크포인트 완료를 나타내는 `WAL` 항목을 생성하고, 그 체크포인트의 시작 `LSN`을 지정합니다. 체크포인트가 시작될 때 로그에는 어떠한 것도 기록되지 않기 때문에, 이 `LSN`은 어떤 유형의 `WAL` 항목에도 해당할 수 있습니다.

또한, `PGDATA/global/pg_control` 파일도 최신 완료된 체크포인트를 참조하도록 업데이트됩니다. 이 과정이 완료될 때까지, `pg_control`은 이전 체크포인트를 계속 유지합니다.



한 번에 모든 것을 명확히 하기 위해 간단한 예제를 살펴보겠습니다. 여러 개의 캐시된 페이지를 변경 상태로 만들겠습니다:

```
=> UPDATE big SET s = 'FOO';
```

```
=> SELECT count(*) FROM pg_buffercache WHERE isdirty;
count
-------
4119
(1 행)
```

현재 WAL 위치를 확인해 보겠습니다:

```
=> SELECT pg_current_wal_insert_lsn();
pg_current_wal_insert_lsn
---------------------------
0/3E7EF7E0
(1 행)
```

이제 체크포인트를 수동으로 완료해 보겠습니다. 변경된 페이지는 모두 디스크에 플러시될 것입니다. 시스템에서 아무런 동작이 없으므로 새로운 변경된 페이지는 나타나지 않습니다:

```
=> CHECKPOINT;
=> SELECT count(*) FROM pg_buffercache WHERE isdirty;
count
--------
0
(1 행)
```

체크포인트가 WAL에 어떻게 반영되었는지 살펴보겠습니다:

```
=> SELECT pg_current_wal_insert_lsn();
pg_current_wal_insert_lsn
---------------------------
0/3E7EF890
(1 행)
```

```
postgres$ /usr/local/pgsql/bin/pg_waldump \
-p /usr/local/pgsql/data/pg_wal -s 0/3E7EF7E0 -e 0/3E7EF890
rmgr: Standby len (rec/tot): 50/ 50, tx: 0, lsn:
0/3E7EF7E0, prev 0/3E7EF7B8, desc: RUNNING_XACTS nextXid 888
latestCompletedXid 887 oldestRunningXid 888
rmgr: XLOG len (rec/tot): 114/ 114, tx: 0, lsn:
0/3E7EF818, prev 0/3E7EF7E0, desc: CHECKPOINT_ONLINE redo
0/3E7EF7E0; tli 1; prev tli 1; fpw true; xid 0:888; oid 24754; multi
1; offset 0; oldest xid 726 in DB 1; oldest multi 1 in DB 1;
oldest/newest commit timestamp xid: 0/0; oldest running xid 888;
online
```

최신 WAL 항목은 체크포인트 완료와 관련이 있습니다(CHECKPOINT_ONLINE).

이 체크포인트의 시작 LSN은 redo라는 단어 뒤에 지정되어 있으며, 이 위치는 체크포인트 시작 시점의 가장 최신으로 삽입된 WAL 항목에 해당합니다.

동일한 정보는 pg_control 파일에서도 찾을 수 있습니다:

```
postgres$ /usr/local/pgsql/bin/pg_controldata \
-D /usr/local/pgsql/data ¦ egrep 'Latest.*location'
Latest checkpoint location:        0/3E7EF818
Latest checkpoint's REDO location: 0/3E7EF7E0
```

10.4 복구Recovery

서버가 시작될 때, 가장 먼저 실행되는 프로세스는 postmaster입니다. postmaster는 장애 발생 시 데이터 복구를 담당하는 시작 프로세스를 생성합니다.[105]

복구 필요 여부를 확인하기 위해, 시작 프로세스는 pg_control 파일을 읽어 클러스터의 상태를 확인합니다. pg_controldata 유틸리티를 사용하면 이 파일의 내용을 확인할 수 있습니다:

```
postgres$ /usr/local/pgsql/bin/pg_controldata \
-D /usr/local/pgsql/data ¦ grep state
Database cluster state: in production
```

서버가 정상적으로 종료된 경우 "shut down" 상태를 가집니다. "in production" 상태는 실행 중이지 않은 서버에서 장애를 나타냅니다. 이 경우, 시작 프로세스는 동일한 pg_control 파일에서 찾은 최신 완료된 체크포인트의 시작 LSN부터 자동으로 복구를 시작합니다.

> 만약 PGDATA 디렉토리에 백업과 관련된 backup_label 파일이 있다면, 시작 LSN 위치는 그 파일에서 가져옵니다.

시작 프로세스는 지정된 위치부터 하나씩 WAL 항목을 읽고, 데이터 페이지의 LSN이 WAL 항목의 LSN보다 작을 경우 해당 항목을 데이터 페이지에 적용합니다. 페이지에 더 큰 LSN이 있는 경우 WAL을 적용해서는 **안됩니다**. 왜냐하면 각 항목은 순차적으로 재생되도록 설계되었기 때문에 엄격히 순차적으로 재생되어야 합니다.

그러나 일부 WAL 항목은 전체 페이지 이미지(FPI)로 구성됩니다. 이 유형의 항목은 페이지의 상태에 상관없이 적용될 수 있습니다. 왜냐하면 모든 페이지 내용이 어차피 지워지기 때문입니다. 이러한 수정 작업은 **멱등**idempotent하다고 합니다. 멱등한 작업의 또 다른 예는 트랜잭션 상태 변경을 등록하는 것입니다. 각 트랜잭션 상태는 이전 값과 상관없이 설정되는 특정 비트에 의해 CLOG에서 정의됩니다. 따라서 CLOG 페이지에서 최신 변경의 LSN을 유지할 필요가 없습니다.

WAL 항목은 일반적인 페이지 업데이트와 마찬가지로 버퍼 캐시의 페이지에 적용됩니다. 예를 들어, WAL 항목이 파일이 존재해야 한다고 나타내지만 어떤 이유로 인해 파일이 없는 경우, 새로 생성됩니다.

복구 과정에서도 파일은 유사한 방식으로 WAL에서 복원됩니다. WAL 항목이 파일이 존재해야 한다고 보여주

[105] backend/postmaster/startup.c
backend/access/transam/xlog.c, StartupXLOG function

지만 특정 이유로 인해 파일이 누잠금된 경우, 파일은 다시 생성됩니다.

복구가 완료된 후에는 모든 로그되지 않은 관계가 해당 초기화 포크로 덮어씌워집니다. 마지막으로, 체크포인트가 실행되어 디스크에 복구된 상태를 안전하게 보장합니다. 이로써 시작 프로세스의 작업은 완료됩니다.

> 전통적인 형태의 복구 과정은 두 단계로 구성됩니다. 롤-포워드 단계에서는 잃어버린 작업을 반복하여 `WAL` 항목을 재생합니다. 롤-백 단계에서는 서버의 실패 시점에서 아직 커밋되지 않은 트랜잭션을 중단시킵니다.
> PostgreSQL 에서는 두 번째 단계인 롤-백 단계가 필요하지 않습니다. 복구 후에는 `CLOG` 에 미완료된 트랜잭션에 관한 커밋 또는 중단 비트가 포함되지 않을 것입니다(기술적으로는 활성 트랜잭션을 나타냅니다). 그러나 해당 트랜잭션이 더 이상 실행 중이지 않다는 것을 확실히 알고 있으므로, 이는 중단된 것으로 간주됩니다.[106]

우리는 서버를 즉시 중지 모드로 강제로 중지하여 장애를 시뮬레이션할 수 있습니다:

```
postgres$ pg_ctl stop -m immediate
```

새로운 클러스터 상태는 다음과 같습니다:

```
postgres$ /usr/local/pgsql/bin/pg_controldata \
-D /usr/local/pgsql/data ¦ grep 'state'
Database cluster state: in production
```

서버를 시작하면 시작 프로세스는 장애가 발생했음을 감지하고 복구 모드로 진입합니다:

```
postgres$ pg_ctl start -l /home/postgres/logfile
postgres$ tail -n 6 /home/postgres/logfile
LOG: database system was interrupted; last known up at 2023-03-06
14:01:49 MSK
LOG: database system was not properly shut down; automatic recovery
in progress
LOG: redo starts at 0/3E7EF7E0
LOG: invalid record length at 0/3E7EF890: wanted 24, got 0
LOG: redo done at 0/3E7EF818 system usage: CPU: user: 0.00 s,
system: 0.00 s, elapsed: 0.00 s
LOG: database system is ready to accept connections
```

서버가 정상적으로 중지되는 경우 `postmaster`는 모든 클라이언트와의 연결을 해제한 다음 최종 체크포인트를 실행하여 모든 변경된 페이지를 디스크에 기록합니다.

현재 WAL 위치를 확인하세요:

```
=> SELECT pg_current_wal_insert_lsn();
pg_current_wal_insert_lsn
```

[106] backend/access/heap/heapam_visibility.c, HeapTupleSatisfiesMVCC function

```
-------------------------
0/3E7EF908
(1 행)
```

먼저 서버를 올바르게 중지해 봅니다:

```
postgres$ pg_ctl stop
```

새로운 클러스터 상태는 다음과 같습니다:

```
postgres$ /usr/local/pgsql/bin/pg_controldata \
-D /usr/local/pgsql/data ¦ grep state
Database cluster state: shut down
```

WAL의 끝에서 CHECKPOINT_SHUTDOWN 항목을 볼 수 있습니다. 이는 최종 체크포인트를 나타냅니다:

```
postgres$ /usr/local/pgsql/bin/pg_waldump \
-p /usr/local/pgsql/data/pg_wal -s 0/3E7EF908
rmgr: XLOG len (rec/tot): 114/ 114, tx: 0, lsn:
0/3E7EF908, prev 0/3E7EF890, desc: CHECKPOINT_SHUTDOWN redo
0/3E7EF908; tli 1; prev tli 1; fpw true; xid 0:888; oid 24754; multi
1; offset 0; oldest xid 726 in DB 1; oldest multi 1 in DB 1;
oldest/newest commit timestamp xid: 0/0; oldest running xid 0;
shutdown
pg_wump: fatal: error in WAL record at 0/3E7EF908: record
length at 0/3E7EF980: wanted 24, got 0
```

가장 최신의 pg_waldump 메시지는 해당 유틸리티가 WAL을 끝까지 읽었다는 것을 나타냅니다.

이제 인스턴스를 다시 시작해 봅니다:

```
postgres$ pg_ctl start -l /home/postgres/logfile
```

10.5 백그라운드 쓰기

만약 백엔드가 변경된 페이지를 버퍼에서 제거해야 하는 상황이라면, 해당 페이지를 디스크에 기록해야 합니다. 이는 원치 않는 상황이므로 대기 시간이 발생하게 됩니다. 따라서 이런 작업을 백그라운드에서 비동기적으로 처리하는 것이 훨씬 효율적입니다.

이런 작업은 checkpointer에 의해 부분적으로 처리되지만, 여전히 완벽하게 처리되지 않습니다. 따라서 vhPostgreSQL은 백그라운드에서의 기록을 위해 특별히 설계된 bgwriter[107]라는 추가 프로세스를 제공합니다. 이 프로세스는 제거와 같은 버퍼 탐색 알고리즘을 사용하지만, 다음 두 가지 주요 차이점이 있습니다:

- bgwriter 프로세스는 제거 보다 뒤처지지 않는 자체 클록 핸드를 사용하며, 일반적으로 더 빠르게 진행됩니다.

[107] backend/postmaster/bgwriter.c

- 버퍼를 순회하는 동안 사용량 카운트는 감소하지 않습니다.

만약 버퍼가 고정되지 않고 사용량 카운트가 0이라면, 변경된 페이지는 디스크에 기록됩니다. 따라서 `bgwriter`는 제거 전에 실행되며, 제거될 가능성이 높은 페이지를 미리 디스크에 기록합니다.

이를 통해 제거를 위해 선택된 버퍼가 깨끗할 확률을 높일 수 있습니다.

10.6 WAL 설정

체크포인트 구성

체크포인트 지속 시간(더욱 정확히는 변경된 버퍼를 디스크에 기록하는 시간)은 `checkpoint_completion_target`(기본값: 0.9) 매개변수에 의해 정의됩니다. 이 값은 두 인접한 체크포인트 시작 사이에 할당된 기록 시간의 비율을 지정합니다. 이 매개변수를 1로 설정하지 않아야 하는데, 그렇게 하면 이전 체크포인트가 완료되기 전에 다음 체크포인트가 예정될 수 있습니다. 재앙은 발생하지 않을 것이지만 한 번에 여러 체크포인트를 실행할 수 없으므로 정상적인 작동이 중단될 수 있습니다.

다른 매개변수를 설정할 때는 다음과 같은 접근 방식을 사용할 수 있습니다. 먼저, 인접한 두 체크포인트 사이에 저장될 `WAL` 파일의 적절한 용량을 정의합니다. 용량이 클수록 부하는 줄어들지만, 이 값은 여전히 사용할 수 있는 여유 공간과 허용되는 복구 시간에 의해 제한될 것입니다.

일반적인 로드로 이 볼륨을 생성하는 데 필요한 시간을 추정하려면 초기 삽입 `LSN`을 기록하고 때때로 현재 삽입 위치와의 차이를 확인해야 합니다. 이렇게 하면 시스템의 성능을 최적화하는 데 도움이 될 것입니다.

받은 숫자는 일반적인 체크포인트 간격으로 가정되므로 `checkpoint_timeout`(기본값: 5분) 매개변숫값으로 사용될 것입니다. 기본 설정은 너무 작을 수 있으므로 일반적으로 30분으로 증가시킵니다.

그러나 로드가 때때로 높을 수 있으므로, 이 간격 동안 생성되는 `WAL` 파일의 크기가 너무 커질 수 있습니다. 이런 경우 체크포인트는 더 자주 실행되어야 합니다. 이런 트리거를 설정하기 위해 복구에 필요한 `WAL` 파일의 크기를 `max_wal_size`(기본값: `1GB`) 매개변수로 제한합니다. 이 임곗값을 초과하면 서버는 추가적인 체크포인트를 실행합니다.[108]

복구에 필요한 `WAL` 파일에는 최신 완료된 체크포인트 및 아직 완료되지 않은 현재 체크포인트의 모든 항목이 포함됩니다. 따라서 예상되는 총용량은 체크포인트 간의 계산된 `WAL` 크기에 `checkpoint_completion_target`를 곱한 값입니다.

> PostgreSQL 버전 11 이전에는 두 개의 완료된 체크포인트를 위해 `WAL` 파일을 유지했으므로 곱하는 값은 `2 + checkpoint_completion_target` 였습니다.

이 접근 방식을 따르면 대부분의 체크포인트는 `checkpoint_timeout` 간격마다 예정대로 실행되지만, 로드가 증가하는 경우 `WAL` 크기가 `max_wal_size` 값을 초과할 때 체크포인트가 발생됩니다.

[108] backend/access/transam/xlog.c, LogCheckpointNeeded & CalculateCheckpointSegments functions

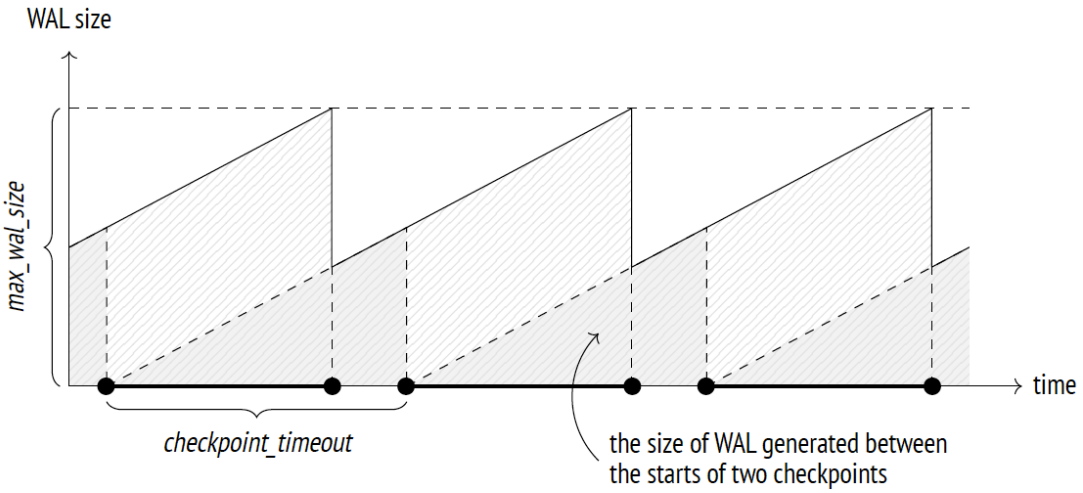
실제 진행 상황은 예상치와 비교하여 주기적으로 확인됩니다:[109]

- **실제 진행 상황**은 이미 처리된 캐시된 페이지의 비율로 정의됩니다.
- **예상 진행 상황(시간 기준)**은 이미 지난 시간의 비율로 정의됩니다. 이는 체크포인트가 `checkpoint_timeout × checkpoint_completion_target` 간격 내에 완료되어야 한다고 가정합니다.
- **예상 진행 상황(크기 기준)**은 이미 채워진 `WAL` 파일의 비율로 정의됩니다. 여기서 예상되는 파일 수는 `max_wal_size × checkpoint_completion_target` 값을 기반으로 추정됩니다.

만약 변경된 페이지가 예정보다 빨리 디스크에 기록된다면, `checkpointer`는 잠시 중지됩니다. 매개변수 중 하나라도 지연이 발생하면 가능한 한 빨리 따라잡습니다.[110] 시간과 데이터 크기 모두 고려되므로 PostgreSQL은 예정된 체크포인트와 요청에 따른 체크포인트를 동일한 방식으로 관리할 수 있습니다.

체크포인트가 완료된 후에는 더 이상 복구에 필요하지 않은 `WAL` 파일이 삭제됩니다.[111] 그러나 일부 파일은 `min_wal_size`(기본값: 80MB)까지 재사용을 위해 유지되고 단순히 이름이 변경됩니다. 이러한 이름 변경은 지속적인 파일 생성 및 삭제로 인한 오버헤드를 줄이지만, 사용하지 않는 경우 `wal_recycle`(기본값: `on`) 매개변수를 사용하여 이 기능을 비활성화할 수 있습니다.

다음 그림은 일반적인 상황에서 디스크에 저장된 `WAL` 파일의 크기가 어떻게 변하는지를 보여줍니다.



디스크에 저장된 `WAL` 파일의 실제 `크기는 max_wal_size` 값보다 큰 경우가 있을 수 있다는 점을 염두에 두는 것이 중요합니다:

- `max_wal_size` 매개변수는 하드 제한이 아닌 원하는 대상 값으로 지정됩니다. 부하가 급증하는 경우 기록이 예정보다 늦어질 수 있습니다.
- 서버는 아직 복제되지 않은 또는 계속되는 아카이빙 처리가 필요한 `WAL` 파일을 삭제할 권한이 없습니다. 이 기능이 활성화된 경우에는 지속적으로 모니터링되어야 하며, 디스크 오버플로우를 쉽게 발생시킬 수 있습니다.
- `wal_keep_size`(기본값: `0MB`) 매개변수를 구성하여 `WAL` 파일에 특정 공간을 예약할 수 있습니다.

---

[109] backend/postmaster/checkpointer.c, IsCheckpointOnSchedule function

[110] backend/postmaster/checkpointer.c, CheckpointWriteDelay function

[111] backend/access/transam/xlog.c, RemoveOldXlogFiles function

백그라운드 쓰기 구성

checkpointer를 구성한 후에는 bgwriter도 설정해야 합니다. 이 두 프로세스는 백엔드가 재사용하기 전에 변경된 버퍼를 디스크에 기록할 수 있어야 합니다.

bgwriter는 작동 중에 주기적으로 일시 정지하며, bgwriter_delay(기본값: 200ms) 시간 단위로 대기합니다.

두 번의 일시 정지 사이에 기록된 페이지 수는 이전 실행 이후 백엔드에 의해 액세스된 버퍼의 평균 수에 따라 달라집니다 (PostgreSQL은 이동 평균을 사용하여 가능한 급증을 완화하고 동시에 매우 오래된 데이터에 의존하지 않도록 합니다). 계산된 수는 bgwriter_lru_multiplier(기본값: 2)로 곱해집니다. 그러나 어떤 경우에도 단일 실행에서 기록된 페이지 수는 bgwriter_lru_maxpages(기본값: 100) 값을 초과할 수 없습니다.

변경된 버퍼가 감지되지 않는 경우 (즉, 시스템에서 아무런 동작이 없는 경우), bgwriter는 백엔드가 버퍼에 액세스할 때까지 대기합니다. 그런 다음 깨어나서 정상적인 작동을 계속합니다.

모니터링

체크포인트 설정은 모니터링 데이터에 따라 조절될 수 있습니다.
만일 설정된 checkpoint_warning(기본 설정값: 30초) 매개변수보다 자주 사이즈 기반 체크포인트가 실행된다면, PostgreSQL은 경고를 발생시키게 됩니다. 이 설정값은 최대 부하가 예상되는 수준에 맞춰 조절해야 합니다.

log_checkpoints(기본 설정값: off) 매개변수는 체크포인트에 관한 정보를 서버 로그에 출력하는 기능을 합니다. 이 기능을 활성화해보도록 하겠습니다:

```
=> ALTER SYSTEM SET log_checkpoints = on;
=> SELECT pg_reload_conf();
```

이제 데이터를 수정하고 체크포인트를 실행해보겠습니다:

```
=> UPDATE big SET s = 'BAR';
=> CHECKPOINT;
```

서버 로그에는 체크포인트 후에 작성된 버퍼의 수, 체크포인트 이후 WAL 파일 변경에 관한 통계, 체크포인트의 지속 시간 및 두 인접 체크포인트 시작지점 간의 거리(바이트)가 표시됩니다:

```
postgres$ tail -n 2 /home/postgres/logfile
LOG: checkpoint starting: immediate force wait
LOG: checkpoint complete: wrote 4100 buffers (25.0%); 0 WAL file(s)
added, 1 removed, 0 recycled; write=0.076 s, sync=0.009 s,
total=0.099 s; sync files=3, longest=0.007 s, average=0.003 s;
distance=9213 kB, estimate=9213 kB
```

구성 결정에 영향을 미칠 수 있는 가장 유용한 데이터는 pg_stat_bgwriter 뷰에서 제공되는 백그라운드 쓰기와 체크포인트 실행에 관한 통계입니다.

버전 9.2 이전에는 두 작업 모두 `bgwriter`에 의해 수행되었으며, 그 후에는 별도의 `checkpointer` 프로세스가 도입되었지만 공통된 뷰는 변경되지 않았습니다.

```
=> SELECT * FROM pg_stat_bgwriter \gx
-[ RECORD 1 ]----------+------------------------------
checkpoints_timed      | 0
checkpoints_req        | 14
checkpoint_write_time  | 33111
checkpoint_sync_time   | 221
buffers_checkpoint     | 14253
buffers_clean          | 13066
maxwritten_clean       | 122
buffers_backend        | 84226
buffers_backend_fsync  | 0
buffers_alloc          | 86700
stats_reset            | 2023-03-06 14:00:07.369124+03
```

이 뷰는 완료된 체크포인트의 수를 보여줍니다:

- `checkpoints_timed` 필드는 예약된 체크포인트를 나타냅니다(체크포인트 타임아웃 간격에 도달할 때 트리거됨).
- `checkpoints_req` 필드는 요청에 의한 체크포인트를 나타냅니다(`max_wal_size` 크기에 도달할 때 트리거되는 것을 포함).

체크포인트 실행 횟수가 예상치를 초과할 경우, 이는 `checkpoints_req` 값이 `checkpoints_timed` 값보다 클 때를 말합니다. 또한, 다음 페이지들에 관한 쓰기 통계도 중요하게 여겨집니다:

- `checkpointer`가 작성한 `buffers_checkpoint` 페이지 수
- 백엔드가 작성한 `buffers_backend` 페이지 수
- `bgwriter`가 작성한 `buffers_clean` 페이지 수

잘 조정된 시스템에서는 `buffers_backend` 값이 `buffers_checkpoint` 값과 `buffers_clean` 값의 합계보다 아주 낮아야 합니다.

백그라운드 쓰기 설정 시에는 `maxwritten_clean` 값을 주의 깊게 봐야 합니다. 이 값은 `bgwriter_lru_maxpages`로 설정된 임곗값을 초과하여 `bgwriter`가 중단된 횟수를 나타냅니다.

다음의 호출은 수집된 통계를 초기화할 것입니다:

```
=> SELECT pg_stat_reset_shared('bgwriter');
```

11 장 WAL 모드

11.1 성능

서버가 정상 작동하는 동안에는 WAL 파일이 지속적으로 디스크에 작성됩니다. 그러나 이 쓰기 작업들은 순차적으로 진행되며, 거의 무작위로 접근하지 않아서 HDD로도 처리 가능합니다. 이와 같은 부하 형태는 일반적인 데이터 파일 접근과 많이 다르기 때문에, WAL 파일을 위해 별도의 물리적 저장소를 설정하고 PGDATA/pg_wal 카탈로그를 마운트된 파일 시스템의 디렉터리로 대체하는 심볼릭 링크로 바꾸는 것이 가치 있는 선택일 수 있습니다.

> WAL 파일을 읽고 쓸 필요가 있는 경우는 주로 두 가지입니다. 하나는 충돌 복구라는 명확한 경우이며, 또 다른 하나는 스트리밍 복제입니다. walsender[112] 프로세스는 WAL 항목을 파일에서 직접 읽어옵니다.[113] 따라서 복제본이 필요한 페이지가 아직 기본 서버의 OS 버퍼에 있을 때 복제본이 WAL 항목을 받지 못한다면, 데이터를 디스크에서 읽어와야 합니다. 그러나 이러한 접근 방식도 무작위가 아닌 순차적으로 이루어집니다.

WAL 항목은 아래 두 가지 모드 중 하나로 작성될 수 있습니다:

- 동기 모드는 트랜잭션 커밋이 관련된 모든 WAL 항목을 디스크에 저장할 때까지 추가 작업을 중단시킵니다.
- 비동기 모드는 트랜잭션 커밋이 즉시 이루어지며, WAL 항목은 이후에 백그라운드에서 디스크에 작성됩니다.

현재 모드는 synchronous_commit(기본값: on) 매개변수에 의해 결정됩니다.

동기 모드에서는 커밋 사실을 신뢰성 있게 기록하기 위해 WAL 항목을 운영 체제에 전달하는 것만으로는 충분하지 않습니다. 디스크 동기화가 성공적으로 완료되었는지 확인해야 합니다. 동기화는 실제 I/O 작업을 의미하며(매우 느릴 수 있습니다), 가능하다면 이 작업을 최소화하는 것이 좋습니다.

이를 위해 트랜잭션을 완료하고 WAL 항목을 디스크에 작성하는 백엔드는 commit_delay(기본값: 0초) 매개변수로 정의된 짧은 일시 정지를 수행할 수 있습니다. 그러나 이 일시 정지는 시스템에 최소한 commit_siblings(기본값: 5) 개의 활성 트랜잭션이 있을 때만 발생합니다.[114] 이 일시 정지 동안 일부 트랜잭션들이 완료될 수 있고, 서버는 모든 WAL 항목을 한 번에 동기화할 수 있습니다. 이는 마치 누군가가 엘리베이터 문을 열어두고 다른 사람이 빠르게 들어오도록 하는 것과 비슷합니다.

기본적으로는 이런 일시 정지는 없습니다. commit_delay 매개변수는 주로 많은 양의 짧은 OLTP 트랜잭션을 처리하는 시스템에서 조정하는 것이 유의미합니다.

일시 정지 후, 트랜잭션을 완료하는 프로세스는 모든 누적된 WAL 항목을 디스크에 플러시하고 동기화를 수행

[112] backend/replication/walsender.c

[113] backend/access/transam/xlogreader.c

[114] backend/access/transam/xlog.c, XLogFlush function

합니다(커밋 항목과 관련된 이전 항목 모두 저장하는 것이 중요합니다. 그 외의 항목은 추가 비용 없이 작성됩니다).

이 시점부터 ACID의 내구성 요구사항이 보장되며, 트랜잭션은 신뢰성 있게 커밋된 것으로 간주됩니다.[115] 이것이 동기 모드가 기본 모드인 이유입니다.

그러나 동기 커밋의 단점은 더 긴 대기 시간(COMMIT 명령이 동기화가 완료될 때까지 제어를 반환하지 않음)과 특히 OLTP 부하에 관한 시스템 처리량의 감소입니다.

비동기 모드에서는 WAL 항목을 비동기 커밋[116]으로 활성화하려면 synchronous_commit 매개변수를 꺼야 합니다.

이 모드에서 WAL 항목은 walwriter[117] 프로세스가 디스크에 작성합니다. 이 프로세스는 작업과 휴식을 번갈아 가며 수행하며, 휴식 시간은 wal_writer_delay(기본값: 200ms) 값으로 정의됩니다.

휴식 후, 프로세스는 캐시에서 새로 채워진 WAL 페이지를 확인합니다. 만약 새로 채워진 페이지가 있다면 현재 페이지를 건너뛰고 해당 페이지들을 디스크에 작성합니다. 만약 없다면, 현재 반쯤 채워진 페이지를 작성합니다.[118]

이 알고리즘의 목적은 동일한 페이지를 여러 번 플러시하지 않도록 하여, 데이터 변경이 많은 작업 부하에 관한 성능 향상을 가져오는 것입니다.

WAL 캐시는 링 버퍼로 사용되지만, walwriter는 캐시의 마지막 페이지에 도달하면 중지됩니다. 휴식 후, 다음 작성 주기는 첫 번째 페이지부터 시작됩니다. 따라서 가장 안 좋은 경우에는 walwriter가 특정 WAL 항목에 도달하기 위해 세 번의 실행이 필요할 수 있습니다: 먼저 캐시의 끝에 있는 모든 완전히 채워진 페이지를 작성하고, 시작점으로 돌아가 마지막으로 해당 항목이 포함된 반쯤 비어 있는 페이지를 처리합니다. 그러나 대부분의 경우에는 한 두 개의 주기로 충분합니다.

동기화는 wal_writer_flush_after(기본값: 1MB)로 설정된 데이터 양이 작성될 때마다 수행되며, 작성 주기의 끝에서도 한 번 더 수행됩니다.

비동기 커밋은 물리적인 디스크 쓰기를 기다릴 필요가 없기 때문에 동기 커밋보다 빠릅니다. 그러나 신뢰성이 저하됩니다: 장애 발생 전에 커밋된 데이터는 기본적으로 최대 3 × wal_writer_delay 시간(0.6초) 동안 손실될 수 있습니다.

실제 세계에서는 동기 모드와 비동기 모드가 서로 보완적으로 사용되곤 합니다. 동기 모드에서도 긴 트랜잭션과 관련된 WAL 항목은 비동기적으로 작성될 수 있어 WAL 버퍼를 해제할 수 있습니다. 반대로, 버퍼 캐시에서 제거될 페이지와 관련된 WAL 항목은 비동기 모드에서도 즉시 디스크에 기록됩니다. 그렇지 않으면 작업을 계속할 수 없습니다.

---

[115] backend/access/transam/xlog.c, RecordTransactionCommit function

[116] postgresql.org/docs/14/wal-async-commit.html

[117] backend/postmaster/walwriter.c

[118] backend/access/transam/xlog.c, XLogBackgroundFlush function

성능과 내구성 사이에서 대부분의 경우 시스템 설계자는 어려운 선택을 해야 합니다.

`synchronous_commit` 매개변수는 특정 트랜잭션에 대해 설정할 수 있습니다. 모든 트랜잭션을 절대적으로 중요한 것으로 봐야 하거나(예: 금융 데이터 처리) 중요하지 않은 트랜잭션으로 응용 프로그램 수준에서 분류할 수 있다면, 중요하지 않은 트랜잭션을 잃을 위험을 감수하면서 성능을 향상시킬 수 있습니다.

비동기 커밋의 잠재적인 성능 향상을 알아보기 위해 `pgbench` 테스트를 사용하여 동기 모드와 비동기 모드에서의 지연 시간과 처리량을 비교해 보겠습니다.[119]

먼저 필요한 테이블을 초기화합니다:

```
postgres$ /usr/local/pgsql/bin/pgbench -i internals
```

동기 모드에서 30초 동안 테스트를 시작합니다:

```
postgres$ /usr/local/pgsql/bin/pgbench -T 30 internals
pgbench (14.7)
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
duration: 30 s
number of transactions actually processed: 20123
latency average = 1.491 ms
initial connection time = 2.507 ms
tps = 670.809688 (initial connection time 을 제외한 값)
```

이제 비동기 모드에서 동일한 테스트를 실행합니다:

```
=> ALTER SYSTEM SET synchronous_commit = off;
=> SELECT pg_reload_conf();
```

```
postgres$ /usr/local/pgsql/bin/pgbench -T 30 internals
pgbench (14.7)
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
duration: 30 s
number of transactions actually processed: 61809
latency average = 0.485 ms
```

[119] postgresql.org/docs/14/pgbench.html

```
initial connection time = 1.915 ms
tps = 2060.399861 (initial connection time을 제외한 값)
```

비동기 모드에서 이 간단한 벤치마크는 지연 시간이 상당히 낮고 처리량(TPS)이 높음을 보여줍니다. 물론 각 시스템은 현재 부하에 따라 다른 결과를 보일 것이지만, 짧은 OLTP 트랜잭션에 미치는 영향이 크게 느껴질 수 있습니다.

기본 설정으로 복원하겠습니다:

```
=> ALTER SYSTEM RESET synchronous_commit;
=> SELECT pg_reload_conf();
```

11.2 장애 허용성Fault Toerance

사전 기록 로깅은 어떤 상황에서라도 (영구 저장소 자체가 파손되지 않는 이상) 충돌 복구를 반드시 보장해야 한다는 것은 분명하다. 데이터 일관성에 영향을 줄 수 있는 많은 요소들이 있겠지만, 여기서는 캐싱, 데이터의 손상 그리고 비 원자적 쓰기에 대해서만 다루어 볼 예정이다.[120]

캐싱

비휘발성 저장소(예: 하드 디스크)에 도달하기 전에 데이터는 여러 캐시를 거칠 수 있습니다.

디스크 쓰기는 실제로는 운영 체제에게 데이터를 캐시에 넣도록 지시하는 것일 뿐입니다(이 캐시 역시 RAM의 다른 부분과 마찬가지로 충돌 위험이 있습니다). 실제 쓰기는 운영 체제의 I/O 스케줄러 설정에 따라 비동기적으로 이루어집니다.

스케줄러가 데이터를 플러시해야 한다고 결정하면, 이 데이터는 스토리지 장치의 캐시(예: HDD)로 이동하게 됩니다. 스토리지 장치는 인접한 페이지들을 그룹화하는 등의 방법으로 쓰기를 연기할 수 있습니다. RAID 컨트롤러는 디스크와 운영 체제 사이에 추가적인 캐싱 레벨을 제공합니다.

특별한 조치를 취하지 않는 한, 데이터가 신뢰성 있게 디스크에 저장되는 시점은 알 수 없습니다. 하지만 일반적으로 이는 사전 기록 로깅(WAL)이 있기 때문에 그리 중요하지 않습니다.[121] 그러나 WAL 항목 자체는 신뢰성 있게 즉시 디스크에 저장되어야 합니다. 비동기 모드에 대해서도 마찬가지입니다. 그렇지 않으면 WAL 항목이 수정된 데이터보다 먼저 디스크에 기록되는 것을 보장할 수 없습니다.

체크포인터 프로세스는 '더티 페이지'를 운영 체제의 캐시에서 디스크로 안전하게 옮겨야 합니다. 더불어, 다른 프로세스에 의해 수행된 모든 파일 작업(예: 페이지 쓰기 또는 파일 삭제)을 동기화해야 합니다. 체크포인트 작업이 완료되면, 이들 모든 작업의 결과는 이미 디스크에 저장되어 있어야 합니다.[122]

또한, 로깅되지 않은 동작을 최소한의 사전 기록 로깅(WAL) 수준에서 실행하는 등, 안전한 쓰기가 필요한 다른 상황도 있습니다. 이를 위해, 운영 체제는 비휘발성 스토리지로 데이터를 즉시 기록하기 위한 다양한 방법

[120] postgresql.org/docs/14/wal-reliability.html

[121] backend/access/transam/xlog.c, issue_xlog_fsync function

[122] backend/storage/sync/sync.c

을 제공합니다. 이들은 크게 두 가지 주요 방식으로 요약할 수 있습니다: 쓰기 후 별도의 동기화 명령(`fsync` 또는 `fdatasync` 등)을 호출하거나, 파일을 열거나 쓸 때 동기화를 수행하도록 지정하는 것입니다(심지어 운영 체제 캐시를 우회하는 직접 쓰기도 가능합니다).

`pg_test_fsync` 유틸리티는 운영 체제와 파일 시스템에 따라 `WAL`을 동기화하는 가장 좋은 방법을 결정하는 데 도움이 될 수 있습니다. 선호하는 방법은 `wal_sync_method` 매개변수로 지정할 수 있습니다. 다른 작업에 대해서는 적절한 동기화 방법이 자동으로 선택되며, 이는 사용자가 구성할 수 없습니다.[123]

여기서 주의해야 할 점은 각각의 경우에 가장 적합한 방법이 하드웨어에 따라 다를 수 있다는 것입니다. 예를 들어, 백업 배터리가 탑재된 컨트롤러를 사용하는 경우, 배터리가 전원 장애 시 데이터를 보호하기 때문에 캐시를 활용할 수 있습니다.

하지만, 비동기 커밋과 동기화를 끄는 것은 완전히 다른 문제입니다. 동기화를 끄는 것(`fsync` 매개변수, 기본값: on)은 시스템 성능을 향상시키지만, 어떤 장애가 발생하면 치명적인 데이터 손실로 이어질 수 있습니다. 반면, 비동기 모드는 일관된 상태까지의 충돌 복구를 보장하지만, 가장 최근의 데이터 업데이트가 누잠금될 수 있습니다.

데이터 손상Corruption

기술 장비는 완벽하지 않아서, 데이터는 메모리나 디스크에서, 또는 인터페이스 케이블을 통해 전송되는 동안 손상될 수 있습니다. 이러한 오류는 대부분 하드웨어 수준에서 처리되지만, 일부는 빠질 수 있습니다.

이러한 문제를 적시에 발견하기 위해 PostgreSQL은 항상 `WAL` 항목을 체크섬으로 보호합니다. 또한 데이터 페이지에 대해서도 체크섬을 계산할 수 있습니다.[124] 이 작업은 클러스터 초기화 중이나 서버가 중지된 상태에서 `pg_checksums`[125] 유틸리티를 실행하여 수행됩니다.[126]

운영 시스템에서는 어떤 경우에도 체크섬을 항상 활성화해야 합니다. 약간의 계산 및 확인 부담이 있더라도, 이는 손상을 빠르게 발견할 수 있는 기회를 높여줍니다. 그러나 아래와 같은 몇 가지 예외 상황이 존재합니다:

- 체크섬 검증은 페이지에 접근할 때만 수행되므로, 데이터 손상이 오랜 시간 동안 감지되지 않을 수 있습니다. 이는 모든 백업에 포함되고 올바른 데이터의 원본이 없을 때까지 계속될 수 있습니다.
- 영점으로 초기화된 페이지는 올바른 것으로 간주하므로, 파일 시스템이 실수로 페이지를 영점으로 초기화하는 경우 이 문제는 발견되지 않을 것입니다.
- 체크섬은 관계의 주요 포크에 대해서만 계산됩니다. 다른 포크 및 파일(예: `CLOG`의 트랜잭션 상태)은 보호되지 않습니다.

체크섬이 활성화되어 있는지 확인하기 위해 읽기 전용 `data_checksums` 매개변수를 확인하면 됩니다:

```
=> SHOW data_checksums;
data_checksums
————————————————————
```

[123] backend/storage/file/fd.c, pg_fsync function

[124] backend/storage/page/README

[125] postgresql.org/docs/14/app-pgchecksums.html

[126] commitfest.postgresql.org/27/2260

```
on
(1 행)
```

이제 서버를 중지하고 테이블의 영점 페이지의 일부 바이트를 영점으로 초기화해봅시다:

```
=> SELECT pg_relation_filepath('wal');
pg_relation_filepath
──────────────────────
base/16391/16562
(1 행)
```

```
postgres$ pg_ctl stop
postgres$ dd if=/dev/zero of=/usr/local/pgsql/data/base/16391/16562 \
oflag=dsync conv=notrunc bs=1 count=8
8+0 records in
8+0 records out
8 bytes copied, 0,00776573 s, 1,0 kB/s
```

이후에, 서버를 다시 시작합니다:

```
postgres$ pg_ctl start -l /home/postgres/logfile
```

사실 서버를 계속 실행해도 괜찮습니다. 페이지를 디스크에 기록하고 캐시에서 제거하기만 하면 됩니다(그렇지 않으면 서버는 캐시된 버전을 계속 사용합니다). 그러나 이러한 작업 흐름은 재현하기가 더 어렵습니다.

이제 테이블을 읽어보려고 시도해 봅시다:

```
=> SELECT * FROM wal LIMIT 1;
WARNING: page verification failed, calculated checksum 20397 but
expected 28733
ERROR: invalid page in block 0 of relation base/16391/16562
```

데이터를 백업에서 복원할 수 없는 경우, 손상된 페이지를 읽어보는 것이 최소한의 의미가 있습니다(손상된 출력을 얻을 위해 위험을 감수해야 함). 이를 위해 `ignore_checksum_failure(default: off)` 매개변수를 활성화해야 합니다:

```
=> SET ignore_checksum_failure = on;
=> SELECT * FROM wal LIMIT 1;
WARNING: page verification failed, calculated checksum 20397 but
expected 28733
id
────
2
(1 행)
```

이 경우 모든 것이 문제없이 진행되었습니다. 왜냐하면 페이지 헤더의 비중요한 부분(최신 `WAL` 항목의 `LSN`)

을 손상시켰기 때문입니다. 데이터 자체는 손상되지 않았습니다.

비원자적 쓰기

데이터베이스 페이지는 일반적으로 8KB를 차지하지만, 낮은 수준에서는 블록 단위로 쓰기가 수행되며, 이 블록은 보통 512byte 또는 4KB와 같이 작을 수 있습니다. 따라서 장애가 발생하면 페이지가 부분적으로만 기록될 수 있습니다. 이러한 페이지에 일반적인 WAL 항목을 복구 중에 적용하는 것은 의미가 없습니다.

PostgreSQL은 부분적인 쓰기를 피하기 위해 WAL에서 첫 번째 변경 시 전체 페이지 이미지(FPI)를 저장합니다. 이 동작은 full_page_writes(기본값: on) 매개변수에 의해 제어되지만, 이를 끄는 것은 치명적인 데이터 손상을 초래할 수 있습니다.

복구 프로세스가 WAL에서 FPI를 만나면, 해당 FPI를 LSN을 확인하지 않고 무조건 디스크에 기록합니다. WAL 항목과 마찬가지로 FPI는 체크섬으로 보호되므로 손상이 눈치채지 못할 수 없습니다. 그런 다음 정확한 상태로 일반적인 WAL 항목이 적용됩니다.

힌트 비트를 설정하는 별도의 WAL 항목 유형은 없습니다. 이 작업은 비중요한 작업으로 간주되며, 페이지에 액세스하는 모든 쿼리는 필요한 비트를 새로 설정합니다. 그러나 힌트 비트 변경은 페이지의 체크섬에 영향을 줍니다. 따라서 체크섬이 활성화된 경우(wal_log_hints(기본값: off) 매개변수가 켜져 있는 경우 포함), 힌트 비트 수정은 FPI로 로깅됩니다.[127]

FPI에는 빈 공간이 제외되기 때문에 로깅 메커니즘[128]은 생성된 WAL 파일의 크기를 상당히 증가시킵니다. wal_compression(기본값: off) 매개변수를 통해 FPI 압축을 활성화하면 상황을 크게 개선할 수 있습니다.

pgbench 유틸리티를 사용하여 간단한 실험을 실행해 보겠습니다. 체크포인트를 수행하고 바로 일정한 트랜잭션 수를 가진 벤치마크 테스트를 시작합니다:

```
=> CHECKPOINT;
=> SELECT pg_current_wal_insert_lsn();
pg_current_wal_insert_lsn
---------------------------
0/42CE5DA8
(1 행)

postgres$ /usr/local/pgsql/bin/pgbench -t 20000 internals
=> SELECT pg_current_wal_insert_lsn();
pg_current_wal_insert_lsn
---------------------------
0/449113E0
(1 행)
```

생성된 WAL 항목의 크기는 다음과 같습니다:

---

[127] backend/storage/buffer/bufmgr.c, MarkBufferDirtyHint function

[128] backend/access/transam/xloginsert.c, XLogRecordAssemble function

```
=> SELECT pg_size_pretty('0/449755C0'::pg_lsn - '0/42CE5DA8'::pg_lsn);
pg_size_pretty
----------------
29 MB
(1 행)
```

이 예에서 FPI는 총 WAL 크기의 절반 이상을 차지합니다. WAL 항목의 수(N), 일반 항목의 크기(Record size), 그리고 각 자원 유형(Type)에 관한 FPI 크기를 보여주는 수집된 통계를 통해 직접 확인할 수 있습니다:

```
postgres$ /usr/local/pgsql/bin/pg_waldump --stats \
-p /usr/local/pgsql/data/pg_wal -s 0/42CE5DA8 -e 0/449755C0
Type            N    (%)       Record size      (%)      FPI size    (%)
----            -  ---- -------------   ---- --------- ----
XLOG         4294 ( 3,31)            210406 ( 2,50)     19820068 (93,78)
Transaction 20004 (15,41)            680536 ( 8,10)            0 ( 0,00)
Storage         1 ( 0,00)                42 ( 0,00)            0 ( 0,00)
CLOG            1 ( 0,00)                30 ( 0,00)            0 ( 0,00)
Standby         6 ( 0,00)               416 ( 0,00)            0 ( 0,00)
Heap2       24774 (19,09)           1536253 (18,27)        24576 ( 0,12)
Heap        80234 (61,81)           5946242 (70,73)       295664 ( 1,40)
Btree         494 ( 0,38)             32747 ( 0,39)       993860 ( 4,70)
        ------                ---------        ---------
Total      129808                   8406672 [28,46%]    21134168 [71,54%]
```

이 비율은 데이터 페이지가 체크포인트 간에 여러 번 수정되는 경우 더 작아질 수 있습니다. 이는 체크포인트를 덜 자주 수행해야 하는 또 다른 이유입니다.

압축을 사용할 수 있는지 확인하기 위해 같은 실험을 반복해 보겠습니다.

```
=> ALTER SYSTEM SET wal_compression = on;
=> SELECT pg_reload_conf();
=> CHECKPOINT;
=> SELECT pg_current_wal_insert_lsn();
pg_current_wal_insert_lsn
---------------------------
0/44D4C228
(1 행)
```

```
postgres$ /usr/local/pgsql/bin/pgbench -t 20000 internals
```

```
=> SELECT pg_current_wal_insert_lsn();
pg_current_wal_insert_lsn
---------------------------
0/457653B0
(1 행)
```

WAL 크기는 압축이 활성화된 상태에서 다음과 같습니다:

```
=> SELECT pg_size_pretty('0/457653B0'::pg_lsn - '0/44D4C228'::pg_lsn);
pg_size_pretty
----------------
10 MB
(1 행)
```

```
postgres$ /usr/local/pgsql/bin/pg_waldump --stats \
-p /usr/local/pgsql/data/pg_wal -s 0/44D4C228 -e 0/457653B0
Type            N     (%) Record size    (%)   FPI size     (%)
----            -     --- -----------    ---   --------    ---
XLOG          344 (  0,29)      17530 ( 0,22)    435492 (17,75)
Transaction 20001 ( 16,73)     680114 ( 8,68)         0 ( 0,00)
Storage         1 (  0,00)         42 ( 0,00)         0 ( 0,00)
Standby         5 (  0,00)        330 ( 0,00)         0 ( 0,00)
Heap2       18946 ( 15,84)    1207425 (15,42)    101601 ( 4,14)
Heap        80141 ( 67,02)    5918020 (75,56)   1627008 (66,31)
Btree         143 (  0,12)       8443 ( 0,11)    289654 (11,80)
            ------            --------          --------
Total       119581             7831904 [76,14%]  2453755 [23,86%]
```

결국, 체크섬이나 full_page_writes로 인해 많은 FPI가 발생하는 경우 (즉, 대부분의 경우), CPU 부하가 약간 발생하더라도 압축을 사용하는 것이 유의미합니다.

11.3 WAL 수준

사전 기록 로그의 핵심 목적은 충돌 복구를 가능하게 하는 것입니다. 그러나 로그에 기록되는 정보의 범위를 넓히면 WAL은 다른 용도로도 활용할 수 있습니다. 포스트그래스큐엘은 최소, 복제, 그리고 논리 로깅 수준을 제공합니다. 각 수준은 이전 수준에서 로그된 모든 내용을 포함하며, 추가 정보를 더합니다. 현재 사용 중인 수준은 wal_level(기본값: replica) 파라미터로 정의되며, 이를 변경하려면 서버를 재시작해야 합니다.

최소Minimal

최소 수준은 충돌 복구만을 보장합니다. 현재 트랜잭션에서 생성되거나 비워진 테이블에 관한 작업은 공간을 절약하기 위해 로그에 기록되지 않습니다. 이는 큰 양의 데이터를 삽입하는 경우(예: CREATE TABLE AS SELECT, CREATE TABLE 등)에 해당합니다. 대신, 필요한 모든 데이터는 즉시 디스크에 기록되고, 시스템 카탈로그의 변경 사항은 트랜잭션 커밋 직후에 바로 반영됩니다.

만약 이러한 작업이 중간에 실패하더라도, 이미 디스크에 저장된 데이터는 표시되지 않으므로 일관성에 영향을 주지 않습니다. 작업이 완료된 상태에서 실패가 발생하면, 다음 WAL 항목을 적용하기 위해 필요한 모든 데이터는 이미 디스크에 저장되어 있습니다.

이 최적화를 위해 새로 생성된 테이블에 쓰여질 데이터의 양은 wal_skip_threshold(기본값: 2MB) 파라미터로 정의됩니다.

`minimal` 수준에서 로깅되는 내용을 살펴보겠습니다.

기본적으로 더 높은 `replica` 수준이 사용되며, 이는 데이터 복제를 지원합니다. `minimal` 수준을 선택하려면 `max_wal_senders`(기본값: 10) 파라미터에서 허용되는 `walsender` 프로세스의 수를 0으로 설정해야 합니다:

```
=> ALTER SYSTEM SET wal_level = minimal;
=> ALTER SYSTEM SET max_wal_senders = 0;
```

이 변경 사항을 적용하려면 서버를 재시작해야 합니다:

```
postgres$ pg_ctl restart -l /home/postgres/logfile
```

현재 `WAL` 위치를 확인하세요:

```
=> SELECT pg_current_wal_insert_lsn();
pg_current_wal_insert_lsn
---------------------------
0/45767698
(1 행)
```

테이블을 비우고 동일한 트랜잭션 내에서 `wal_skip_threshold`가 초과될 때까지 새로운 행을 계속해서 삽입합니다:

```
=> BEGIN;
=> TRUNCATE TABLE wal;
=> INSERT INTO wal
SELECT id FROM generate_series(1,100000) id;
=> COMMIT;
=> SELECT pg_current_wal_insert_lsn();
pg_current_wal_insert_lsn
---------------------------
0/45767840
(1 행)
```

> 새로운 테이블을 생성하는 대신 `TRUNCATE` 명령을 실행하면 `WAL` 항목이 더 적게 생성됩니다.

익숙한 `pg_waldump` 유틸리티를 사용하여 생성된 `WAL`을 살펴보겠습니다.

```
postgres$ /usr/local/pgsql/bin/pg_waldump \
-p /usr/local/pgsql/data/pg_wal -s 0/45767698 -e 0/45767840#
rmgr: Storage len (rec/tot): 42/ 42, tx: 0, lsn:
0/45767698, prev 0/45767660, desc: CREATE base/16391/24784
rmgr: Heap len (rec/tot): 123/ 123, tx: 122844, lsn:
0/457676C8, prev 0/45767698, desc: UPDATE off 45 xmax 122844 flags
0x60 ; new off 48 xmax 0, blkref #0: rel 1663/16391/1259 blk 0
rmgr: Btree len (rec/tot): 64/ 64, tx: 122844, lsn:
0/45767748, prev 0/457676C8, desc: INSERT_LEAF off 176, blkref #0:
```

```
rel 1663/16391/2662 blk 2
rmgr: Btree len (rec/tot): 64/ 64, tx: 122844, lsn:
0/45767788, prev 0/45767748, desc: INSERT_LEAF off 147, blkref #0:
rel 1663/16391/2663 blk 2
rmgr: Btree len (rec/tot): 64/ 64, tx: 122844, lsn:
0/457677C8, prev 0/45767788, desc: INSERT_LEAF off 254, blkref #0:
rel 1663/16391/3455 blk 4
rmgr: Transaction len (rec/tot): 54/ 54, tx: 122844, lsn:
0/45767808, prev 0/457677C8, desc: COMMIT 2023-03-06 14:03:58.395214
MSK; rels: base/16391/24783
```

첫 번째 항목은 테이블에 관한 새 파일 생성을 기록합니다(`TRUNCATE`는 사실상 테이블을 다시 작성하기 때문입니다).

다음 네 항목은 시스템 카탈로그 작업과 관련이 있습니다. 이는 `pg_class` 테이블과 해당하는 세 개의 인덱스의 변경 사항을 반영합니다.

마지막으로 커밋과 관련된 항목이 있습니다. 데이터 삽입은 로그로 기록되지 않습니다.

복제Replica

크래시 복구 중에는 `WAL` 항목이 재생되어 디스크의 데이터가 일관된 상태로 복원됩니다. 백업 복구도 비슷한 방식으로 작동하지만 `WAL` 아카이브를 사용하여 지정된 복구 대상 지점까지 데이터베이스 상태를 복원할 수도 있습니다. 아카이브된 `WAL` 항목의 수는 매우 많을 수 있으며 (예: 몇 일에 걸칠 수 있음), 복구 기간에는 여러 개의 체크포인트가 포함됩니다. 따라서 최소한의 `WAL` 레벨만으로는 충분하지 않습니다. 로깅되지 않은 작업은 반복할 수 없습니다. 백업 복구를 위해 `WAL` 파일에는 모든 작업이 포함되어야 합니다.

복제에도 동일한 사실이 적용됩니다. 로깅되지 않은 명령은 복제본에 전송되지 않으며 다시 재생되지 않습니다.

복제본에서 쿼리를 실행하는 경우 더 복잡해집니다. 우선, 복제본은 기본 서버에서 획득한 배타적 잠금 정보를 가져와야 합니다. 왜냐하면 이러한 잠금들은 복제본에서의 쿼리와 충돌할 수 있기 때문입니다. 두 번째로, 복제본은 스냅샷을 캡처할 수 있어야 하는데, 이는 활성 트랜잭션에 관한 정보가 필요합니다. 복제본을 다룰 때에는 로컬 트랜잭션과 기본 서버에서 실행 중인 트랜잭션 모두 고려되어야 합니다.

이러한 데이터를 복제본에 전송하는 유일한 방법은 주기적으로 `WAL` 파일에 기록하는 것입니다.[129] 이 작업은 `bgwriter`[130] 프로세스에 의해 15초에 한 번씩 수행됩니다 (이 간격은 하드코딩되어 있습니다).

백업에서 데이터 복구와 물리적 복제를 수행할 수 있는 능력은 `replica` 수준에서 보장됩니다.

앞에서 구성한 매개변수를 단순히 재설정하고 서버를 다시 시작하여 복제본 수준을 사용할 수 있습니다:

```
=> ALTER SYSTEM RESET wal_level;
```

[129] backend/storage/ipc/standby, LogStandbySnapshot function

[130] backend/postmaster/bgwriter.c

```
=> ALTER SYSTEM RESET max_wal_senders;
```

```
postgres$ pg_ctl restart -l /home/postgres/logfile
```

이전과 동일한 작업 흐름을 반복해 보겠습니다. (하지만 이번에는 더 깔끔한 출력을 위해 한 행만 삽입합니다):

```
=> SELECT pg_current_wal_insert_lsn();
pg_current_wal_insert_lsn
---------------------------
0/45D88E48
(1 행)

=> BEGIN;
=> TRUNCATE TABLE wal;
=> INSERT INTO wal VALUES (42);
=> COMMIT;
=> SELECT pg_current_wal_insert_lsn();
pg_current_wal_insert_lsn
---------------------------
0/45D89108
(1 행)
```

생성된 `WAL` 항목을 확인해 보겠습니다.

최소한의 레벨에서 본 내용 이외에도 다음과 같은 항목들이 있습니다:

- 스탠바이 자원 매니저의 복제 관련 항목: `RUNNING_XACTS` (활성 트랜잭션) 및 `LOCK`
- `INSERT+INIT` 작업을 기록하는 항목으로, 새 페이지를 초기화하고 이 페이지에 새로운 행을 삽입합니다.

```
postgres$ /usr/local/pgsql/bin/pg_waldump \
-p /usr/local/pgsql/data/pg_wal -s 0/45D88E48 -e 0/45D89108
rmgr: Standby len (rec/tot): 42/ 42, tx: 122846, lsn:
0/45D88E48, prev 0/45D88DD0, desc: LOCK xid 122846 db 16391 rel 16562
rmgr: Storage len (rec/tot): 42/ 42, tx: 122846, lsn:
0/45D88E78, prev 0/45D88E48, desc: CREATE base/16391/24786
rmgr: Heap len (rec/tot): 123/ 123, tx: 122846, lsn:
0/45D88EA8, prev 0/45D88E78, desc: UPDATE off 49 xmax 122846 flags
0x60 ; new off 50 xmax 0, blkref #0: rel 1663/16391/1259 blk 0
rmgr: Btree len (rec/tot): 64/ 64, tx: 122846, lsn:
0/45D88F28, prev 0/45D88EA8, desc: INSERT_LEAF off 178, blkref #0:
rel 1663/16391/2662 blk 2
rmgr: Btree len (rec/tot): 64/ 64, tx: 122846, lsn:
0/45D88F68, prev 0/45D88F28, desc: INSERT_LEAF off 149, blkref #0:
rel 1663/16391/2663 blk 2
```

```
rmgr: Btree len (rec/tot): 64/ 64, tx: 122846, lsn:
0/45D88FA8, prev 0/45D88F68, desc: INSERT_LEAF off 256, blkref #0:
rel 1663/16391/3455 blk 4
rmgr: Heap len (rec/tot): 59/ 59, tx: 122846, lsn:
0/45D88FE8, prev 0/45D88FA8, desc: INSERT+INIT off 1 flags 0x00,
blkref #0: rel 1663/16391/24786 blk 0
rmgr: Standby len (rec/tot): 42/ 42, tx: 0, lsn:
0/45D89028, prev 0/45D88FE8, desc: LOCK xid 122846 db 16391 rel 16562
rmgr: Standby len (rec/tot): 54/ 54, tx: 0, lsn:
0/45D89058, prev 0/45D89028, desc: RUNNING_XACTS nextXid 122847
latestCompletedXid 122845 oldestRunningXid 122846; 1 xacts: 122846
rmgr: Transaction len (rec/tot): 114/ 114, tx: 122846, lsn:
0/45D89090, prev 0/45D89058, desc: COMMIT 2023-03-06 14:04:14.538399
MSK; rels: base/16391/24785; inval msgs: catcache 51 catcache 50
relcache 16562
```

논리Logical

마지막으로, `logical` 수준은 논리 디코딩과 논리 복제를 가능하게 합니다. 이는 게시 서버에서 활성화해야 합니다.

`WAL` 항목을 살펴보면 이 수준이 복제본과 거의 동일하다는 것을 알 수 있습니다. 이는 복제 소스와 애플리케이션에 의해 생성될 수 있는 임의의 논리 항목을 추가합니다. 논리 디코딩은 주로 활성 트랜잭션 정보(`RUNNING_XACTS`)에 의존합니다. 왜냐하면 시스템 카탈로그 변경 사항을 추적하기 위해 스냅샷을 캡처해야 하기 때문입니다.

Part III

Locks

12 장 테이블 수준의 잠금

12.1 잠금에 관해서

잠금Locks은 공유 자원에 관한 동시 액세스를 제어합니다.

동시 액세스는 여러 프로세스가 동시에 동일한 자원를 가져오려고 시도하는 것을 의미합니다.
이러한 프로세스가 병렬로 실행되는지 (하드웨어가 허용하는 경우) 또는 시분할 모드로 순차적으로 실행되는지는 중요하지 않습니다. 만약 동시 액세스가 없으면 잠금을 얻을 필요가 없습니다 (예: 공유 버퍼 캐시는 잠금이 필요하지만 로컬 캐시는 필요하지 않을 수 있음).

자원에 액세스하기 전에 프로세스는 해당 자원에 관한 잠금을 **획득**해야 합니다. 작업이 완료되면 이 잠금을 해제하여 다른 프로세스가 해당 자원를 사용할 수 있도록 합니다. 잠금이 데이터베이스 시스템에 의해 관리되는 경우 작업의 순서가 자동으로 유지됩니다. 잠금이 응용 프로그램으로 제어되는 경우 프로토콜은 응용 프로그램에서 강제해야 합니다.

낮은 수준에서 잠금은 단순히 잠금 상태(획득됨 또는 획득되지 않음)를 정의하는 공유 메모리 청크입니다. 또한 프로세스 번호나 획득 시간과 같은 추가 정보를 제공할 수도 있습니다.

> 공유 메모리 세그먼트 자체도 자원입니다. 이러한 자원에 관한 동시 액세스는 운영 체제에서 제공하는 동기화 기본 요소(예: 세마포어 또는 뮤텍스)에 의해 규제됩니다. 이러한 기본 요소는 공유 자원에 액세스하는 코드의 엄격한 연속 실행을 보장합니다. 최하위 수준에서는 이러한 기본 요소가 원자적인 `CPU` 명령(예: 테스트-앤-셋 또는 비교-앤-스왑)을 기반으로 합니다.

일반적으로 잠금은 특정 잠금 주소를 명확하게 식별하고 할당할 수 있는 모든 자원를 보호하는 데 사용할 수 있습니다.

예를 들어, 테이블(시스템 카탈로그에서 `oid`로 식별되는)이나 데이터 페이지(파일 이름과 해당 파일 내 위치로 식별되는), 행 버전(페이지와 페이지 내 오프셋으로 식별되는)과 같은 데이터베이스 객체를 잠그는 것이 가능합니다. 해시 테이블이나 버퍼(할당된 `ID`로 식별되는)와 같은 메모리 구조도 잠금할 수 있습니다. 물리적인 표현이 없는 추상적인 리소스도 잠그는 것이 가능합니다.

하지만 잠금을 즉시 획득하는 것이 항상 가능한 것은 아닙니다. 다른 사용자에 의해 이미 잠겨있을 수도 있습니다. 그런 경우 프로세스는 대기열에 참여하거나 나중에 다시 시도해야 합니다. 어떤 방식이든 잠금이 해제될 때까지 기다려야 합니다.

잠금 효율에 큰 영향을 미칠 수 있는 두 가지 요소를 강조하고 싶습니다.

잠금의 "곡선 크기"인 **적정성**granularity입니다. 적정성은 리소스가 계층 구조를 형성하는 경우 중요합니다.

> 예를 들어, 테이블은 페이지로 구성되고, 페이지는 튜플로 구성됩니다. 이러한 객체들은 모두 잠금으로 보호될 수 있습니다. 테이블 수준의 잠금은 거칠게coarsegrained작동하며, 프로세스가 서로 다른 페이지나 행에 접근해야 할 경우에도 동시 액세스를 허용하지 않습니다.

행 수준의 잠금은 미세하게fine-grained 작동하므로 이러한 단점은 없습니다. 그러나 잠금의 수가 증가합니다. 잠금 관련 메타데이터에 너무 많은 메모리를 사용하지 않기 위해 PostgreSQL은 다양한 방법을 적용할 수 있으며, 그 중 하나는 **잠금 승격**lock escalation입니다. 미세한 잠금의 수가 일정 임곗값을 초과하면 더 거친 적정성의 단일 잠금으로 대체됩니다.

잠금을 획득할 수 있는 여러 **모드의 집합**도 있습니다.

일반적으로 두 가지 모드만 적용됩니다. 배타적 모드는 자체를 포함한 다른 모드와 호환되지 않습니다. 공유 모드는 여러 프로세스에 의해 동시에 리소스를 잠그는 것을 허용합니다. 공유 모드는 읽기에 사용되고, 배타적 모드는 쓰기에 사용됩니다.

일반적으로 다른 모드도 존재할 수 있습니다. 모드의 이름은 중요하지 않으며, 중요한 것은 호환성 행렬입니다.

더 미세한 적정성과 다중 호환 모드 지원은 동시 실행에 관한 더 많은 기회를 제공합니다.

모든 잠금은 지속 기간에 따라 분류할 수 있습니다.

장기적 잠금은 잠재적으로 오랜 시간(대부분 트랜잭션의 끝까지) 동안 획득되며, 일반적으로 관계와 행과 같은 리소스를 보호합니다. 이러한 잠금은 일반적으로 PostgreSQL이 자동으로 관리하지만, 사용자는 이 과정에 일부 제어권을 가지고 있습니다.

장기적인 잠금은 다양한 동시 작업을 가능하게 하는 여러 모드를 제공합니다. 일반적으로 대기열, 교착 상태 감지, 계측 등과 같은 다양한 인프라를 갖추고 있으며, 이는 보호된 데이터에 관한 작업보다 유지 관리 비용이 훨씬 저렴합니다.

단기적 잠금은 일부 초 단위로 획득되며, 일반적으로 여러 `CPU` 명령보다 오래 지속되지 않습니다. 일반적으로 공유 메모리의 데이터 구조를 보호합니다. PostgreSQL은 이러한 잠금을 완전히 자동화된 방식으로 관리합니다.

단기적인 잠금은 일반적으로 매우 적은 모드와 기본적인 인프라만 제공하며, 계측조차 없을 수도 있습니다.

PostgreSQL은 다양한 유형의 잠금을 지원합니다.[131] 관계 및 기타 객체에서 획득되는 **중량있는**Heavyweight **잠금**과 **행 수준 잠금**은 장기적인 잠금으로 간주됩니다. 단기적인 잠금은 공유 메모리의 다양한 잠금과 같은 **메모리 구조에 관한 잠금**을 포함합니다. 또한 "`predicate locks`"라는 별도의 그룹도 있으며, 이름과는 달리 실제로는 잠금이 아닙니다.

12.2 중량적 잠금

중량적 잠금은 장기적인 잠금입니다. 객체 수준에서 획득되며, 주로 관계에 사용되지만 다른 유형의 객체에도 적용될 수 있습니다. 중량있는 잠금은 일반적으로 동시 업데이트로부터 객체를 보호하거나 재구성 중에 사용을 금지하는 데 사용되지만, 다른 목적으로도 사용될 수 있습니다. 이러한 유형의 잠금은 다양한 목적으로 사용되므로 모호한 정의입니다. 그들이 공통적으로 갖는 것은 내부 구조뿐입니다.

[131] backend/storage/lmgr/README

명시적으로 다른 방식으로 지정하지 않는 한, 잠금이라는 용어는 일반적으로 중량있는 잠금을 의미합니다.

중량있는 잠금은 서버의 공유 메모리[132]에 위치하며, `pg_locks` 뷰에서 확인할 수 있습니다. 이들의 총 수는 `max_locks_per_transaction`(기본값: 64) 값에 `max_connections`(기본값: 100)를 곱한 값으로 제한됩니다. 모든 트랜잭션은 공통의 잠금 풀을 사용하므로 한 트랜잭션은 `max_locks_per_transaction` 잠금보다 많이 획득할 수 있습니다. 실제로 중요한 것은 시스템의 총 잠금 수가 정의된 제한을 초과하지 않는 것입니다. 풀은 서버가 시작될 때 초기화되므로 이러한 두 매개변수 중 하나를 변경하려면 서버를 다시 시작해야 합니다.

호환되지 않는 모드로 이미 리소스가 잠겨 있는 경우, 다른 잠금을 획득하려는 프로세스는 대기열에 참여합니다. 대기 프로세스는 `CPU` 시간을 낭비하지 않으며, 잠금이 해제되고 운영 체제가 깨우면까지 잠들어 있습니다.

첫 번째 트랜잭션이 다른 트랜잭션이 잠그고 있는 리소스를 얻을 때까지 작업을 진행할 수 없고, 이를 위해 첫 번째 트랜잭션이 잠그고 있는 리소스가 필요한 경우, 두 트랜잭션은 **교착 상태**deadlock에 빠질 수 있습니다. 이 경우는 상대적으로 간단한 경우이며, 교착 상태는 두 개 이상의 트랜잭션을 포함할 수도 있습니다. 교착 상태는 무한한 대기를 유발하므로, PostgreSQL은 자동으로 교착 상태를 감지하고 영향을 받는 트랜잭션 중 하나를 중단하여 정상적인 작업을 계속할 수 있도록 합니다.

다른 유형의 중량있는 잠금은 서로 다른 목적을 가지며, 다른 리소스를 보호하며, 다른 모드를 지원합니다. 따라서 이들을 개별적으로 고려하겠습니다.

다음 목록은 `pg_locks` 뷰의 `locktype` 열에 표시되는 잠금 유형의 이름을 제공합니다:

- `transactionid` 및 `virtualxid`: 트랜잭션에 관한 잠금
- `relation`: 테이블 수준의 잠금
- `tuple`: 튜플에 획득된 잠금
- `object`: 테이블이 아닌 개체에 관한 잠금
- `extend`: 테이블 확장 잠금
- `page`: 일부 인덱스 유형에서 사용되는 페이지 수준의 잠금
- `advisory`: 자문 잠금

거의 모든 중량있는 잠금은 필요할 때 자동으로 획득되며, 해당 트랜잭션이 완료될 때 자동으로 해제됩니다. 그러나 몇 가지 예외가 있습니다. 예를 들어, 관계 수준의 잠금은 명시적으로 설정할 수 있으며, 자문 잠금은 항상 사용자에 의해 관리됩니다.

12.3 트랜잭션 `ID` 잠금

각 트랜잭션은 항상 자체 `ID`(가상 및 실제 `ID`, 가능한 경우)에 관한 배타적인 잠금을 보유합니다.

PostgreSQL은 이를 위해 배타적 모드와 공유 모드 두 가지 잠금 모드를 제공합니다. 이들의 호환성 행렬은 매우 간단합니다: 공유 모드는 자체와 호환되지만, 배타적 모드는 어떤 모드와도 결합할 수 없습니다.

[132] backend/storage/lmgr/lock.c

	Shared	Exclusive
Shared		×
Exclusive	×	×

특정 트랜잭션이 완료되었는지 추적하기 위해 프로세스는 해당 트랜잭션의 ID에 관한 잠금을 어떤 모드로든 요청할 수 있습니다. 트랜잭션 자체가 이미 자체 ID에 관한 배타적인 잠금을 보유하고 있기 때문에 다른 잠금을 획득하는 것은 불가능합니다. 이러한 잠금을 요청하는 프로세스는 대기열에 참여하고 잠들게 됩니다. 트랜잭션이 완료되면 잠금이 해제되고 대기 중인 프로세스가 깨어납니다. 분명히 해당 리소스가 이미 사라져 있기 때문에 해당 잠금을 획득하는 것은 불가능하지만, 이 잠금은 실제로 필요한 것이 아닙니다.

별도의 세션에서 트랜잭션을 시작하고 백엔드의 프로세스 ID(PID)를 가져와 봅시다:

```
    => BEGIN;
    => SELECT pg_backend_pid();
    pg_backend_pid
   ----------------
    28980
   (1 행)
```

시작된 트랜잭션은 자체 가상 ID에 관한 배타적인 잠금을 보유하고 있습니다:

```
=> SELECT locktype, virtualxid, mode, granted
FROM pg_locks WHERE pid = 28980;
         locktype ¦       virtualxid ¦                mode ¦ granted
------------------+------------------+--------------------+---------
       virtualxid ¦              5/2 ¦        ExclusiveLock ¦ t
(1 행)
```

여기서 locktype은 잠금의 유형을, virtualxid는 잠긴 리소스를 식별하는 가상 트랜잭션 ID를, mode는 잠금 모드를 (이 경우에는 배타적 잠금) 나타냅니다. granted 플래그는 요청한 잠금이 획득되었는지 여부를 보여줍니다.

트랜잭션이 실제 ID를 받으면 해당 잠금이 이 목록에 추가됩니다:

```
    => SELECT pg_current_xact_id();
    pg_current_xact_id
    --------------------
    122849
    (1 행)

=> SELECT locktype, virtualxid, transactionid AS xid, mode, granted
FROM pg_locks WHERE pid = 28980;
      locktype ¦ virtualxid ¦    xid ¦          mode ¦ granted
---------------+------------+--------+---------------+---------
    virtualxid ¦        5/2 ¦        ¦ ExclusiveLock ¦ t
 transactionid ¦            ¦ 122849 ¦ ExclusiveLock ¦ t
```

```
(2 행s)
```

이제 이 트랜잭션은 자체 `ID`에 관한 배타적 잠금을 보유하고 있습니다.

12.4 테이블 수준 잠금

포그트그레스큐엘은 관계(테이블, 인덱스 또는 기타 개체)를 잠글 수 있는 8가지 모드를 제공합니다.[133] 이러한 다양성을 통해 관계에 대해 실행할 수 있는 동시 명령의 수를 극대화할 수 있습니다.

다음 페이지에는 해당 잠금 모드를 필요로 하는 명령의 예제를 포함하여 확장된 호환성 행렬이 표시됩니다. 이러한 모든 모드를 외우거나 그들의 네이밍에 관한 논리를 찾으려고 하는 것은 의미가 없지만, 이 데이터를 검토하고 일반적인 결론을 도출하며 필요에 따라 이 테이블을 참조하는 것은 분명히 유용합니다.

	AS	RS	RE	SUE	S	SRE	E	AE	
Access Share								×	SELECT
Row Share							×	×	SELECT FOR UPDATE/SHARE
Row Exclusive					×	×	×	×	INSERT, UPDATE, DELETE
Share Update Exclusive				×	×	×	×	×	VACUUM, CREATE INDEX CONCURRENTLY
Share			×	×		×	×	×	CREATE INDEX
Share Row Exclusive			×	×	×	×	×	×	CREATE TRIGGER
Exclusive		×	×	×	×	×	×	×	REFRESH MAT. VIEW CONCURRENTLY
Access Exclusive	×	×	×	×	×	×	×	×	DROP, TRUNCATE, VACUUM FULL, LOCK TABLE, REFRESH MAT. VIEW

`Access Share` 모드는 가장 약한 모드입니다. `Access Exclusive`와는 호환되지 않으며, 다른 모든 모드와 함께 사용할 수 있습니다. 따라서 `SELECT` 명령은 거의 모든 작업과 병렬로 실행될 수 있지만, 질의 중인 테이블을 삭제할 수는 없습니다.

첫 번째 네 가지 모드는 동시 힙 수정을 허용하며, 나머지 네 가지 모드는 허용하지 않습니다. 예를 들어, `CREATE INDEX` 명령은 `Share` 모드를 사용하며, 자체와 호환되므로 테이블에 여러 인덱스를 동시에 생성할 수 있으며, 읽기 전용 작업에 사용되는 모드와도 호환됩니다. 결과적으로 `SELECT` 명령은 인덱스 생성과 병렬로 실행될 수 있지만, `INSERT`, `UPDATE` 및 `DELETE` 명령은 차단됩니다.

반대로, 힙 데이터를 수정하는 미완료 트랜잭션은 `VACUUM` 명령을 차단합니다. 대신 `CREATE INDEX CONCURRENTLY`를 호출할 수 있으며, 이는 더 약한 `Share Update Exclusive` 모드를 사용합니다. 인덱스 생성에 더 많은 시간이 걸리고 (실패할 수도 있음), 그 대신 동시 데이터 업데이트가 허용됩니다.

`ALTER TABLE` 명령에는 다양한 잠금 모드를 사용하는 다양한 옵션이 있습니다 (`Share Update Exclusive`,

[133] postgresql.org/docs/14/explicit-locking#LOCKING-TABLES.html

Share 행 Exclusive, Access Exclusive). 이에 관한 자세한 내용은 문서에서 설명되어 있습니다.[134]

이 책의 이 부분에서 제공된 예제는 다시 accounts 테이블을 기반으로 합니다:

```
=> TRUNCATE accounts;
=> INSERT INTO accounts(id, client, amount)
VALUES
(1, 'alice', 100.00),
(2, 'bob', 200.00),
(3, 'charlie', 300.00);
```

pg_locks 테이블에 여러 번 액세스해야 하므로 단일 열에 모든 잠금을 표시하는 뷰를 생성하여 출력을 더 간결하게 만들어 봅시다:

```
=> CREATE VIEW locks AS
SELECT pid,
  locktype,
  CASE locktype
      WHEN 'relation' THEN relation::regclass::text
      WHEN 'transactionid' THEN transactionid::text
      WHEN 'virtualxid' THEN virtualxid
  END AS lockid,
  mode,
  granted
FROM pg_locks
ORDER BY 1, 2, 3;
```

첫 번째 세션에서 여전히 실행 중인 트랜잭션은 한 행을 업데이트합니다. 이 작업은 accounts 테이블과 해당 인덱스를 잠그며, 이로 인해 행 Exclusive 모드로 획득된 관계 유형의 두 개의 새로운 잠금이 발생합니다:

```
    => UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;

=> SELECT locktype, lockid, mode, granted
FROM locks WHERE pid = 28980;
            locktype ¦               lockid ¦                     mode ¦ granted
─────────────────────+──────────────────────+──────────────────────────+─────────
            relation ¦             accounts ¦          행ExclusiveLock ¦ t
            relation ¦         accounts_pkey ¦          행ExclusiveLock ¦ t
       transactionid ¦               122849 ¦             ExclusiveLock ¦ t
          virtualxid ¦                  5/2 ¦             ExclusiveLock ¦ t
(4 행s)
```

[134] postgresql.org/docs/14/sql-altertable.html

12.5 대기열

중량있는 잠금은 공정한 대기열Wait Queue을 형성합니다.[135] 프로세스는 현재 잠금 또는 이미 대기열에 있는 다른 프로세스가 요청한 잠금과 호환되지 않는 잠금을 획득하려고 시도하면 대기열에 참여합니다.

첫 번째 세션에서 업데이트 작업을 수행하는 동안 다른 세션에서 이 테이블에 인덱스를 생성해 보겠습니다:

```
   => SELECT pg_backend_pid();
           pg_backend_pid
  ──────────────────
                    29459
   (1 행)
   => CREATE INDEX ON accounts(client);
```

이 명령은 리소스가 해제될 때까지 대기하며 멈춰 있습니다. 트랜잭션은 테이블을 `Share` 모드로 잠그려고 시도하지만 할 수 없습니다:

```
=> SELECT locktype, lockid, mode, granted
FROM locks WHERE pid = 29459;
         locktype ¦        lockid ¦                 mode ¦ granted
──────────────+────────────+──────────────────+──────────
         relation ¦      accounts ¦            ShareLock ¦ f
       virtualxid ¦           6/3 ¦.       ExclusiveLock ¦ t
(2 행s)
```

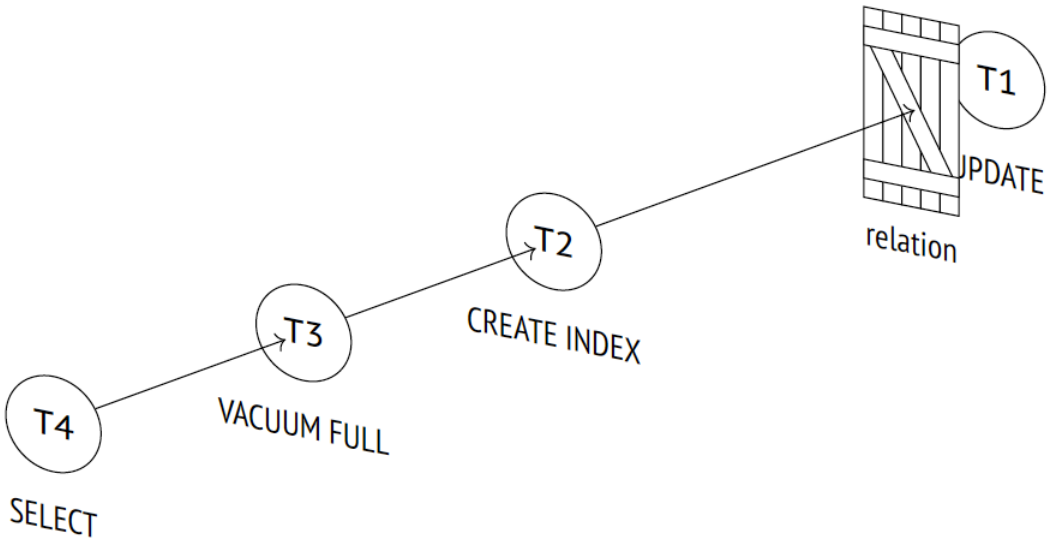
이제 세 번째 세션에서 `VACUUM FULL` 명령을 시작해 보겠습니다. 이 명령은 `Access Exclusive` 모드를 필요로 하며, 다른 모든 모드와 충돌합니다. 따라서 대기열에 참여하게 됩니다:

```
   => SELECT pg_backend_pid();
     pg_backend_pid
   ────────────────
              29662
   (1 행)
   => VACUUM FULL accounts;

=> SELECT locktype, lockid, mode, granted
FROM locks WHERE pid = 29662;
      locktype ¦   lockid ¦                mode ¦ granted
───────────────+──────────+────────────────────+─────────
      relation ¦ accounts ¦ AccessExclusiveLock ¦ f
 transactionid ¦   122853 ¦       ExclusiveLock ¦ t
   virtualxid ¦.     7/4 ¦       ExclusiveLock ¦ t
(3 행)
```

[135] backend/storage/lmgr/lock.c, LockAcquire function

이제 이후의 모든 경쟁자들은 자신들의 잠금 모드와 관계없이 대기열에 참여해야 합니다. 심지어 간단한 `SELECT` 쿼리도 `VACUUM FULL` 명령어를 따라야 합니다. 실제로는 첫 번째 세션에서 업데이트 작업을 수행하는 동안 보유하고 있는 `행 Exclusive` 잠금과 호환될 수 있지만, 그들은 정직하게 대기열에 참여하게 됩니다.

```
   => SELECT pg_backend_pid();
   pg_backend_pid
  ----------------
            29872
  (1 행)
=> SELECT * FROM accounts;

=> SELECT locktype, lockid, mode, granted
FROM locks WHERE pid = 29872;
      locktype ¦   lockid ¦            mode ¦ granted
---------------+----------+-----------------+---------
      relation ¦ accounts ¦ AccessShareLock ¦ f
    virtualxid ¦      8/3 ¦   ExclusiveLock ¦ t
(2 행s)
```



`pg_blocking_pids` 함수는 모든 대기 상태를 간략하게 보여줍니다. 지정된 프로세스 앞에 대기 중인 모든 프로세스의 `ID`를 표시하며, 이미 호환되지 않는 잠금을 보유하고 있는지 또는 획득하려는지를 나타냅니다.

```
=> SELECT pid,
          pg_blocking_pids(pid),
          wait_event_type,
          state,
          left(query,50) AS query
FROM pg_stat_activity
WHERE pid IN (28980,29459,29662,29872) \gx
```

```
-[ RECORD 1 ]----+--------------------------------------------------
            pid ¦ 28980
pg_blocking_pids ¦ {}
 wait_event_type ¦ Client
           state ¦ idle in transaction
           query ¦ UPDATE accounts SET amount = amount + 100.00 WHERE
-[ RECORD 2 ]----+--------------------------------------------------
            pid ¦ 29459
pg_blocking_pids ¦ {28980}
 wait_event_type ¦ Lock
           state ¦ active
           query ¦ CREATE INDEX ON accounts(client);
-[ RECORD 3 ]----+--------------------------------------------------
            pid ¦ 29662
pg_blocking_pids ¦ {28980,29459}
 wait_event_type ¦ Lock
           state ¦ active
           query ¦ VACUUM FULL accounts;
-[ RECORD 4 ]----+--------------------------------------------------
            pid ¦ 29872
pg_blocking_pids ¦ {29662}
 wait_event_type ¦ Lock
           state ¦ active
           query ¦ SELECT * FROM accounts;
```

이 테이블에서 제공되는 정보를 검토하여 자세한 내용을 확인할 수 있습니다.[136]

트랜잭션이 완료되면(커밋되거나 롤백되면) 해당 트랜잭션의 모든 잠금이 해제됩니다.[137] 대기열의 첫 번째 프로세스는 요청한 잠금을 획득하고 깨어납니다.

여기서 첫 번째 세션에서의 트랜잭션 커밋으로 인해 대기 중인 모든 프로세스가 순차적으로 실행됩니다:

```
    => ROLLBACK;
    ROLLBACK

    CREATE INDEX

    VACUUM

    id ¦      client ¦ amount
----+----------+---------
```

[136] wiki.postgresql.org/wiki/Lock_dependency_information

[137] backend/storage/lmgr/lock.c, LockReleaseAll & LockRelease functions

```
  1 ¦   alice ¦ 100.00
  2 ¦     bob ¦ 200.00
  3 ¦ charlie ¦ 300.00
(3 행s)
```

13 장 행 수준 잠금

13.1 잠금 설계

스냅숏 격리snapshot isolation을 통해, 힙 튜플 행들은 읽기를 위해 잠금을 걸 필요가 없습니다. 하지만, 두 개의 쓰기 트랜잭션이 동시에 동일한 행을 수정하는 것을 허용해서는 안 됩니다. 이 경우에는 행을 잠그지만, 중량잠금은 이를 위한 좋은 선택이 아닙니다: 각각의 중량잠금은 서버의 공유 메모리에 공간을 차지하며 (수백 바이트를 차지하며, 모든 지원 인프라를 고려하지 않았을 때), PostgreSQL의 내부 메커니즘은 대량의 동시 중량잠금을 처리하기 위해 설계되지 않았습니다.

일부 데이터베이스 시스템에서는 잠금 승격lock escalation을 사용하여 이 문제를 해결합니다: 만약 행 수준 잠금이 너무 많다면, 보다 세부적인 단위(예: 페이지 수준 또는 테이블 수준)의 단일 잠금으로 대체됩니다. 이는 구현을 단순화하지만, 시스템 처리량을 크게 제한할 수 있습니다.

PostgreSQL에서는 특정 행이 잠겨있는지 여부에 관한 정보는 현재 힙 튜플의 헤더에만 저장됩니다.
행 수준 잠금은 실제 잠금이 아닌 힙 페이지에서 가상의 속성으로 존재하며, `RAM`에는 어떤 방식으로도 반영되지 않습니다.

행은 일반적으로 수정 또는 삭제가 진행될 때 잠금을 걸게 됩니다. 이 두 경우 모두, 현재 버전의 행은 삭제되었다고 표시됩니다. 이를 위해 사용하는 속성은 `xmax` 필드에 지정된 현재 트랜잭션의 `ID`이며, 이 `ID`(추가적인 힌트 비트와 결합된 형태)는 행이 잠겨있음을 나타냅니다. 만약 트랜잭션이 행을 수정하려고 할 때 현재 버전의 `xmax` 필드에 활성 트랜잭션 `ID`가 있는 경우, 이 트랜잭션이 완료될 때까지 기다려야 합니다. 한 번 완료되면 모든 잠금이 해제되고, 기다리던 트랜잭션이 진행할 수 있습니다.

이 메커니즘은 추가 비용 없이 필요한 만큼 많은 행을 잠글 수 있게 합니다. 이 설루션의 단점은 다른 프로세스가 대기열을 형성할 수 없다는 점입니다. `RAM`은 이러한 잠금에 관한 정보를 포함하지 않기 때문에, 중량잠금은 여전히 필요합니다: 행이 해제되기를 기다리는 프로세스는 현재 해당 행과 작업 중인 트랜잭션의 `ID`에 관한 잠금을 요청합니다. 트랜잭션이 완료되면 행은 다시 사용 가능해집니다. 따라서 중량잠금의 수는 행이 수정되는 것보다는 동시 프로세스의 수에 비례합니다.

13.2 행 수준 잠금 모드

행 수준 잠금은 네 가지 모드를 지원합니다.[138] 이 중 두 가지는 동시에 한 번에 하나의 트랜잭션만이 획득할 수 있는 배타적 잠금exclusive lock을 구현하며, 나머지 두 가지는 여러 트랜잭션이 동시에 보유할 수 있는 공유 잠금shared lock을 제공합니다.

다음은 이러한 모드의 호환성 매트릭스입니다:

[138] postgresql.org/docs/14/explicit-locking#LOCKING-행S.html

	Key Share	Share	No Key Update	Update
Key Share				×
Share			×	×
No Key Update		×	×	×
Update	×	×	×	×

베타적 모드

수정 모드는 어떤 튜플 필드도 수정하고 전체 튜플을 삭제하는 것을 허용합니다. 반면 비 키 수정 모드는 고유한 인덱스와 관련된 필드를 변경하지 않는 변경만 허용합니다 (다시 말해, 외래 키는 영향을 받지 않아야 합니다).

`UPDATE` 명령은 가능한 가장 약한 잠금 모드를 자동으로 선택합니다. 보통 키는 변경되지 않기 때문에 행은 일반적으로 비 키 수정 모드로 잠깁니다.

다음은 우리가 관심 있는 몇 가지 튜플 메타데이터, 즉 `xmax` 필드와 여러 힌트 비트를 표시하기 위해 `pageinspect`를 사용하는 함수를 생성하는 예입니다:

```
CREATE FUNCTION 행_locks(relname text, pageno integer)
RETURNS TABLE(
ctid tid, xmax text,
lock_only text, is_multi text,
keys_upd text, keyshr text,
shr text
)
AS $$
SELECT (pageno,lp)::text::tid,
   t_xmax,
   CASE WHEN t_infomask & 128 = 128 THEN 't' END,
   CASE WHEN t_infomask & 4096 = 4096 THEN 't' END,
   CASE WHEN t_infomask2 & 8192 = 8192 THEN 't' END,
   CASE WHEN t_infomask & 16 = 16 THEN 't' END,
   CASE WHEN t_infomask & 16+64 = 16+64 THEN 't' END
FROM heap_page_items(get_raw_page(relname,pageno))
ORDER BY lp;
$$ LANGUAGE sql;
```

이제 `accounts` 테이블에서 첫 번째 계정의 잔액을 업데이트하고 (키는 동일한 상태로 유지됨) 두 번째 계정의 `ID`를 업데이트하는 트랜잭션을 시작해보겠습니다:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
=> UPDATE accounts SET id = 20 WHERE id = 2;
```

이제 페이지에는 다음과 같은 메타데이터가 포함되어 있습니다:

```
=> SELECT * FROM 행_locks('accounts',0) LIMIT 2;
  ctid ¦   xmax ¦ lock_only ¦ is_multi ¦ keys_upd ¦ keyshr ¦ shr
-------+--------+-----------+----------+----------+--------+-----
 (0,1) ¦ 122858 ¦           ¦          ¦          ¦        ¦
 (0,2) ¦ 122858 ¦           ¦          ¦        t ¦        ¦
(2 행s)
```

잠금 모드는 `keys_updated` 힌트 비트에 의해 정의됩니다.

```
=> ROLLBACK;
```

`SELECT FOR` 명령은 잠금 속성으로서 `xmax` 필드를 사용하지만, 이 경우에는 `xmax_lock_only` 힌트 비트도 설정되어야 합니다. 이 비트는 튜플이 잠겨있지만 삭제되지 않았음을 나타내며, 즉 여전히 현재 상태인 것을 의미합니다.

```
=> BEGIN;
=> SELECT * FROM accounts WHERE id = 1 FOR NO KEY UPDATE;
=> SELECT * FROM accounts WHERE id = 2 FOR UPDATE;

=> SELECT * FROM 행_locks('accounts',0) LIMIT 2;
  ctid ¦   xmax ¦ lock_only ¦ is_multi ¦ keys_upd ¦ keyshr ¦ shr
-------+--------+-----------+----------+----------+--------+-----
 (0,1) ¦ 122859 ¦         t ¦          ¦          ¦        ¦
 (0,2) ¦ 122859 ¦         t ¦          ¦        t ¦        ¦
(2 행s)
=> ROLLBACK;
```

공유 모드

공유 모드는 행을 읽어야 하지만 다른 트랜잭션에 의한 수정이 금지되어야 할 때 적용할 수 있습니다. 키 공유 모드는 키 속성을 제외한 모든 튜플 필드를 수정할 수 있도록 허용합니다.

PostgreSQL 코어는 모든 공유 모드 중에서 키 공유만 사용하며, 이는 외래 키를 확인할 때 적용됩니다. 외래 키 검사는 비 키 수정 배타적 모드와 호환되기 때문에, 외래 키 검사는 비 키 속성의 동시 수정에 영향을 주지 않습니다. 응용 프로그램은 원하는 공유 모드를 사용할 수 있습니다. 한 가지 강조하고 싶은 점은 단순한 `SELECT` 명령은 절대로 행 수준 잠금을 사용하지 않는다는 것입니다.

이제 다음과 같이 힙 튜플에서 확인할 수 있는 내용을 살펴보겠습니다:

```
=> BEGIN;
=> SELECT * FROM accounts WHERE id = 1 FOR KEY SHARE;
=> SELECT * FROM accounts WHERE id = 2 FOR SHARE;
```

다음은 힙 튜플에서 확인할 수 있는 내용입니다:

```
=> SELECT * FROM 행_locks('accounts',0) LIMIT 2;
  ctid ¦   xmax ¦ lock_only ¦ is_multi ¦ keys_upd ¦ keyshr ¦ shr
-------+--------+-----------+----------+----------+--------+-----
 (0,1) ¦ 122860 ¦         t ¦          ¦          ¦      t ¦
 (0,2) ¦ 122860 ¦         t ¦          ¦          ¦      t ¦ t
(2 행s)
```

두 작업에 대해 `xmax_keyshr_lock` 비트가 설정되어 있지만, 다른 힌트 비트로 공유 모드를 인식할 수 있습니다.[139]

13.3 다중트랜젝션

우리가 확인한 대로, 잠금 속성은 잠금을 획득한 트랜잭션의 `ID`인 `xmax` 필드로 표현됩니다. 그렇다면 여러 트랜잭션이 동시에 보유한 공유 잠금의 경우 이 속성은 어떻게 설정될까요?

PostgreSQL은 공유 잠금을 다룰 때, `so-called` **다중트랜젝션** (`multixacts`)[140]라고 불리는 것을 적용합니다. 다중트렌젝션은 별도의 `ID`가 할당된 트랜잭션 그룹입니다. 그룹 멤버 및 그들의 잠금 모드에 관한 자세한 정보는 `PGDATA/pg_multixact` 디렉토리의 파일에 저장됩니다. 빠른 액세스를 위해 잠긴 페이지는 서버의 공유 메모리[141]에 캐시되며, 모든 변경 사항은 장애 허용성을 보장하기 위해 로그에 기록됩니다.

다중트렌젝션 `ID는` 일반적인 트랜잭션 `ID`와 동일한 32비트 길이를 가지지만, 독립적으로 발급됩니다. 즉, 트랜잭션과 다중트렌젝션은 동일한 `ID`를 가질 수 있습니다. 둘을 구분하기 위해 PostgreSQL은 추가적인 힌트 비트인 `xmax_is_multi`를 사용합니다.

이제 다른 트랜잭션이 획득한 하나의 배타적 잠금(키 공유와 비 키 수정 모드는 호환됨)을 추가해보겠습니다:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;

=> SELECT * FROM 행_locks('accounts',0) LIMIT 2;
  ctid ¦   xmax ¦ lock_only ¦ is_multi ¦ keys_upd ¦ keyshr ¦ shr
-------+--------+-----------+----------+----------+--------+-----
 (0,1) ¦      1 ¦           ¦        t ¦          ¦        ¦
 (0,2) ¦ 122860 ¦         t ¦          ¦          ¦      t ¦ t
(2 행s)
```

`xmax_is_multi` 비트는 첫 번째 행이 일반적인 `ID` 대신에 다중 트랜젝션 `ID`를 사용한다는 것을 보여줍니다. 더 구체적인 구현 세부 사항에 관해 들어가지 않고, `pg행locks` 확장을 사용하여 가능한 모든 행 수준 잠금에 관한 정보를 표시해보겠습니다:

---

[139] include/access/htup_details.h

[140] backend/access/transam/multixact.c

[141] backend/access/transam/slru.c

```
=> CREATE EXTENSION pg행locks;
=> SELECT * FROM pg행locks('accounts') \gx
-[ RECORD 1 ]--------------------------
locked_행 ¦ (0,1)
    locker ¦ 1
     multi ¦ t
      xids ¦ {122860,122861}
     modes ¦ {"Key Share","No Key Update"}
      pids ¦ {30423,30723}
-[ RECORD 2 ]--------------------------
locked_행 ¦ (0,2)
    locker ¦ 122860
     multi ¦ f
      xids ¦ {122860}
     modes ¦ {"For Share"}
      pids ¦ {30423}
```

`pg행locks` 함수는 `pg_locks` 뷰를 조회하는 것과 많이 유사하지만, `RAM`에는 행 수준 잠금에 관한 정보가 없으므로 `pg행locks` 함수는 힙 페이지에 액세스해야 합니다.

```
=> COMMIT;

    => ROLLBACK;
```

`multixact ID`는 32비트이므로 일반적인 트랜잭션 `ID`와 마찬가지로 카운터 제한으로 인해 `wraparound`의 영향을 받습니다. 따라서 PostgreSQL은 `multixact ID`를 동결하는 방식으로 처리해야 합니다. 오래된 `multixact ID`는 새로운 `ID`로 대체되며 (또는 해당 시점에 잠금을 보유하는 트랜잭션이 하나뿐인 경우에는 일반적인 트랜잭션 `ID`로 대체됩니다).[142]

일반적인 트랜잭션 `ID`는 `xmin` 필드에서만 동결되지만 (비어 있지 않은 `xmax`가 튜플이 오래되었고 곧 제거될 것을 나타냄), 다중 트랜젝션의 경우 동결해야 할 필드는 `xmax`입니다: 현재 행 버전은 공유 모드로 새로운 트랜잭션에 의해 반복적으로 잠금을 받을 수 있습니다.

다중 트랜젝션의 동결은 일반적인 동결과 유사한 방식으로 관리할 수 있습니다. 이를 위해 서버 파라미터가 제공됩니다: `vacuum_multixact_freeze_min_age`, `vacuum_multixact_freeze_table_age`, `autovacuum_multixact_freeze_max_age`, `vacuum_multixact_failsafe_age` 등이 있습니다.

13.4 대기열

베타적 모드

행 수준 잠금은 그 자체로 하나의 속성이므로, 대기열은 그다지 간단한 방식으로 정렬됩니다. 트랜잭션이 행

[142] backend/access/heap/heapam.c, FreezeMultiXactId function

을 수정하려고 할 때 다음 단계를 따라야 합니다:[143]

1. `xmax` 필드와 힌트 비트가 행이 호환되지 않는 모드로 잠겨있음을 나타내면, 수정 중인 튜플에 배타적인 중량잠금을 획득합니다.
2. 필요한 경우, 모든 호환되지 않는 잠금이 해제될 때까지 기다리기 위해 `xmax` 트랜잭션(또는 `xmax`에 `mutixact ID`가 포함된 경우 여러 트랜잭션)의 `ID`에 관한 잠금을 요청합니다.
3. 튜플 헤더의 `xmax`에 자신의 `ID`를 기록하고 필요한 힌트 비트를 설정합니다.
4. 첫 번째 단계에서 획득한 경우에는 튜플 잠금을 해제합니다.

튜플 잠금은 일반적인 행 수준 잠금과 구분하기 위해 튜플 유형을 가진 다른 종류의 중량잠금입니다.

1단계와 4단계는 중복된 것처럼 보일 수 있으며, 단순히 모든 잠금 트랜잭션이 완료될 때까지 기다리면 충분할 것으로 생각될 수 있습니다. 그러나 여러 트랜잭션이 동일한 행을 수정하려고 시도하는 경우, 모두 해당 행을 현재 처리 중인 트랜잭션에 대해 대기하게 됩니다. 해당 트랜잭션이 완료되면, 행을 잠그기 위한 권한을 얻기 위해 경쟁 상태에 놓이게 되며, 일부 불행한 트랜잭션은 무기한으로 대기해야 할 수도 있습니다. 이러한 상황을 **자원 굶주림**resource starvation이라고 합니다. 튜플 잠금은 대기열에서 첫 번째 트랜잭션을 식별하고, 그 트랜잭션이 다음으로 잠금을 얻을 것임을 보장합니다.

하지만 직접 확인해보실 수 있습니다. PostgreSQL은 운영 중에 여러 종류의 다른 잠금을 획득하며, 각각은 `pg_locks` 테이블의 별도의 행에 반영됩니다. 그러므로 `pg_locks` 위에 또 다른 뷰를 생성하겠습니다. 이 뷰는 우리가 현재 관심을 갖는 잠금(계정 테이블과 트랜잭션 자체와 관련된 잠금)만 보여줄 것입니다. (가상 `ID`에 관한 잠금은 제외합니다):

```
=> CREATE VIEW locks_accounts AS
SELECT pid,
       locktype,
       CASE locktype
            WHEN 'relation' THEN relation::regclass::text
            WHEN 'transactionid' THEN transactionid::text
            WHEN 'tuple' THEN relation::regclass||'('||page||','||tuple||')'
       END AS lockid,
       mode,
       granted
FROM pg_locks
WHERE locktype in ('relation','transactionid','tuple')
AND (locktype != 'relation' OR relation = 'accounts'::regclass)
ORDER BY 1, 2, 3;
```

첫 번째 트랜잭션을 시작하고 행을 업데이트해보겠습니다:

```
    => BEGIN;
    => SELECT txid_current(), pg_backend_pid();
    txid_current | pg_backend_pid
   --------------+----------------
```

[143] backend/access/heap/README.tuplock

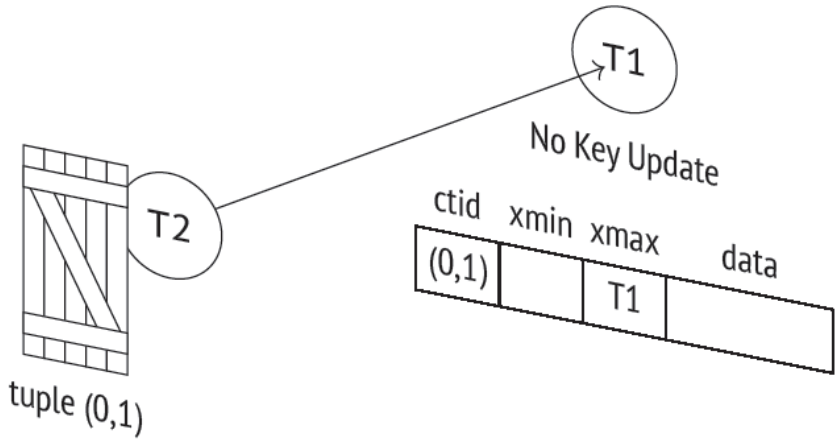
```
          122863 ¦ 30723
    (1 행)
    => UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
```

이제 트랜잭션은 워크플로우의 네 가지 단계를 모두 완료하고 테이블에 잠금을 보유하고 있습니다:

```
=> SELECT * FROM locks_accounts WHERE pid = 30723;
   pid ¦      locktype ¦   lockid ¦             mode ¦ granted
-------+---------------+----------+------------------+---------
 30723 ¦      relation ¦ accounts ¦ 행ExclusiveLock ¦ t
 30723 ¦ transactionid ¦   122863 ¦    ExclusiveLock ¦ t
(2 행s)
```

두 번째 트랜잭션을 시작하고 동일한 행을 업데이트하려고 시도합니다. 이 트랜잭션은 잠금을 기다리면서 대기 상태에 빠지게 됩니다:

```
    => BEGIN;
    => SELECT txid_current(), pg_backend_pid();
    t xid_current ¦ pg_backend_pid
    --------------+----------------
           122864 ¦ 30794
    (1 행)
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
```



두 번째 트랜잭션은 두 번째 단계까지만 진행됩니다. 이 때문에 테이블과 해당 트랜잭션의 잠금 외에도 첫 번째 단계에서 획득한 튜플 잠금과 두 번째 단계에서 요청한 두 개의 트랜잭션의 `ID` 잠금이 추가로 설정됩니다. 이는 `pg_locks` 뷰에도 반영됩니다:

```
=> SELECT * FROM locks_accounts WHERE pid = 30794;
   pid ¦      locktype ¦        lockid ¦             mode ¦ granted
-------+---------------+---------------+------------------+---------
 30794 ¦      relation ¦      accounts ¦ RowExclusiveLock ¦ t
 30794 ¦ transactionid ¦        122863 ¦        ShareLock ¦ f
 30794 ¦ transactionid ¦        122864 ¦    ExclusiveLock ¦ t
```

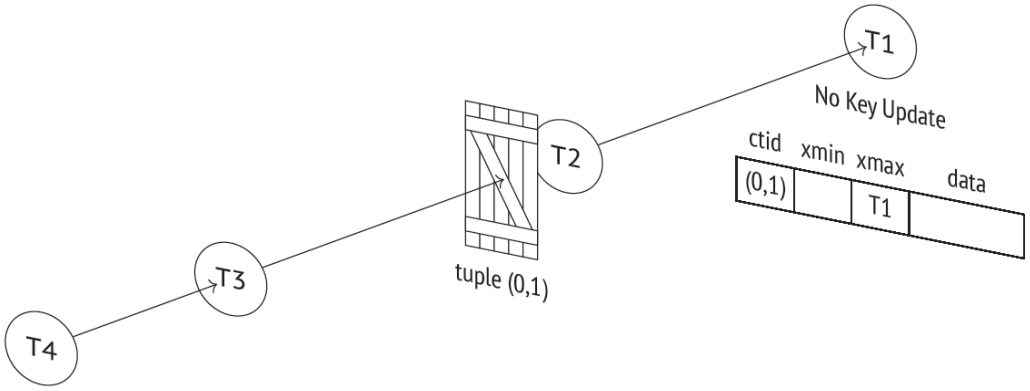
```
 30794 ¦         tuple ¦ accounts(0,1) ¦    ExclusiveLock ¦ t
(4 rows)
```

세 번째 트랜잭션은 첫 번째 단계에서 멈추게 됩니다. 이 트랜잭션은 튜플에 관한 잠금을 획득하려고 시도하고 이 지점에서 멈추게 됩니다:

```
    => BEGIN;
    => SELECT txid_current(), pg_backend_pid();
      txid_current ¦ pg_backend_pid
     --------------+----------------
            122865 ¦ 30865
     (1 row)
    => UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;

=> SELECT * FROM locks_accounts WHERE pid = 30865;
   pid ¦      locktype ¦        lockid ¦             mode ¦ granted
-------+---------------+---------------+------------------+---------
 30865 ¦      relation ¦      accounts ¦ RowExclusiveLock ¦ t
 30865 ¦ transactionid ¦        122865 ¦    ExclusiveLock ¦ t
 30865 ¦         tuple ¦ accounts(0,1) ¦    ExclusiveLock ¦ f
(3 rows)
```

네 번째와 이후의 행을 수정하려는 모든 트랜잭션은 이와 마찬가지로 세 번째 트랜잭션과 동일한 상태입니다. 모두 동일한 튜플 잠금을 기다리게 됩니다.

```
    => BEGIN;
    => SELECT txid_current(), pg_backend_pid();
    txid_current ¦ pg_backend_pid
   --------------+----------------
          122866 ¦ 30936
     (1 row)
     => UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;

=> SELECT * FROM locks_accounts WHERE pid = 30865;
   pid ¦      locktype ¦        lockid ¦             mode ¦ granted
-------+---------------+---------------+------------------+---------
 30865 ¦      relation ¦      accounts ¦ RowExclusiveLock ¦ t
 30865 ¦ transactionid ¦        122865 ¦    ExclusiveLock ¦ t
 30865 ¦         tuple ¦ accounts(0,1) ¦    ExclusiveLock ¦ f
(3 rows)
```



현재 대기 상태의 전체 상황을 파악하기 위해 `pg_stat_activity` 뷰를 잠금 프로세스 정보와 확장할 수 있습니다:

```
=> SELECT pid,
          wait_event_type,
          wait_event,
          pg_blocking_pids(pid)
FROM pg_stat_activity
WHERE pid IN (30723,30794,30865,30936);
   pid ¦ wait_event_type ¦    wait_event ¦ pg_blocking_pids
-------+----------------+---------------+------------------
 30723 ¦          Client ¦    ClientRead ¦ {}
 30794 ¦            Lock ¦ transactionid ¦ {30723}
 30865 ¦            Lock ¦         tuple ¦ {30794}
 30936 ¦            Lock ¦         tuple ¦ {30794,30865}
(4 rows)
```

만약 첫 번째 트랜잭션이 중단된다면, 모든 후속 트랜잭션은 대기열을 건너뛰지 않고 한 단계씩 진행될 것입니다.

하지만 첫 번째 트랜잭션이 커밋될 가능성이 더 높습니다. 반복적 읽기 또는 직렬화 격리 수준에서는 직렬화 실패가 발생하여 두 번째 트랜잭션을 중단시켜야 합니다[144] (대기열의 모든 후속 트랜잭션도 중단됩니다). 그러나 커밋된 읽기 격리 수준에서는 수정된 행이 다시 읽히고 수정이 다시 시도됩니다.

따라서 첫 번째 트랜잭션이 커밋됩니다:

```
    => COMMIT;
```

두 번째 트랜잭션이 깨어나고 워크플로우의 세 번째 단계와 네 번째 단계를 성공적으로 완료합니다:

```
    UPDATE 1

=> SELECT * FROM locks_accounts WHERE pid = 30794;
```

[144] backend/executor/nodeModifyTable.c, ExecUpdate function

```
  pid ¦      locktype ¦   lockid ¦            mode ¦ granted
------+---------------+----------+-----------------+---------
 30794 ¦      relation ¦ accounts ¦ RowExclusiveLock ¦ t
 30794 ¦ transactionid ¦   122864 ¦    ExclusiveLock ¦ t
(2 rows)
```

두 번째 트랜잭션이 튜플 잠금을 해제하자마자, 세 번째 트랜잭션도 깨어납니다. 그러나 새로운 튜플의 `xmax` 필드가 이미 다른 `ID`를 포함하고 있는 것을 알 수 있습니다.

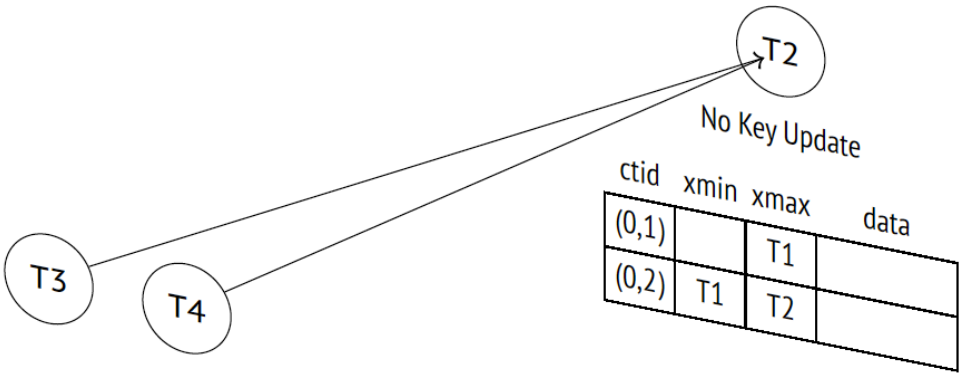
이 시점에서 위에서 설명한 워크플로우는 끝났습니다. 커밋된 읽기 격리 수준에서는 행을 잠그기 위해 한 번 더 시도하지만[145], 위에서 설명한 단계를 따르지 않습니다. 세 번째 트랜잭션은 이제 두 번째 트랜잭션이 완료되기를 기다리면서 튜플 잠금을 획득하려고 시도하지 않습니다:

```
=> SELECT * FROM locks_accounts WHERE pid = 30865;
  pid ¦      locktype ¦   lockid ¦            mode ¦ granted
------+---------------+----------+-----------------+---------
 30865 ¦      relation ¦ accounts ¦ RowExclusiveLock ¦ t
 30865 ¦ transactionid ¦   122864 ¦        ShareLock ¦ f
 30865 ¦ transactionid ¦   122865 ¦    ExclusiveLock ¦ t
(3 rows)
```

네 번째 트랜잭션도 동일한 상태입니다:

```
=> SELECT * FROM locks_accounts WHERE pid = 30936;
  pid ¦      locktype ¦   lockid ¦            mode ¦ granted
------+---------------+----------+-----------------+---------
 30936 ¦      relation ¦ accounts ¦ RowExclusiveLock ¦ t
 30936 ¦ transactionid ¦   122864 ¦        ShareLock ¦ f
 30936 ¦ transactionid ¦   122866 ¦    ExclusiveLock ¦ t
(3 rows)
```

이제 세 번째와 네 번째 트랜잭션은 두 번째 트랜잭션이 완료되기를 기다리면서 서로 경쟁 상태에 빠지게 됩니다. 대기열은 사실상 붕괴되었습니다.



[145] backend/access/heap/heapam_handler.c, heapam_tuple_lock function

대기열이 존재하는 동안 다른 트랜잭션이 참여했다면, 모두 이 경쟁 상태에 끌려들었을 것입니다.

결론: 동시에 여러 프로세스에서 동일한 테이블 행을 업데이트하는 것은 좋은 아이디어가 아닙니다. 높은 부하 하에서 이러한 핫스팟은 성능 문제를 일으킬 수 있는 병목 현상으로 빠르게 변할 수 있습니다.
시작된 모든 트랜잭션을 커밋해보겠습니다.

```
    => COMMIT;

     UPDATE 1
     => COMMIT;

      UPDATE 1
      => COMMIT;
```

공유 모드

PostgreSQL은 공유 잠금을 참조 무결성 검사에만 사용합니다. 고부하 애플리케이션에서 이를 사용하면 자원 굶주림 문제가 발생할 수 있으며, 두 단계 잠금 모델은 이러한 결과를 방지할 수 없습니다.

다시 행을 잠그기 위해 트랜잭션이 수행해야 할 단계를 상기해보겠습니다:

1. `xmax` 필드와 힌트 비트가 행이 배타적 모드로 잠겨있음을 나타내면, 배타적인 중량 튜플 잠금을 획득합니다.
2. 필요한 경우, 호환되지 않는 잠금이 모두 해제될 때까지 기다리기 위해 `xmax` 트랜잭션(또는 `xmax`에 `multixact ID`가 포함된 경우 여러 트랜잭션)의 `ID`에 관한 잠금을 요청합니다.
3. 튜플 헤더의 `xmax`에 자신의 `ID`를 기록하고 필요한 힌트 비트를 설정합니다.
4. 첫 번째 단계에서 획득한 경우에는 튜플 잠금을 해제합니다.

첫 두 단계는 잠금 모드가 호환되는 경우, 트랜잭션이 대기열을 건너뛰게 됨을 의미합니다.

이제 처음부터 실험을 다시 진행해보겠습니다.

```
=> TRUNCATE accounts;
=> INSERT INTO accounts(id, client, amount)
VALUES
(1,'alice',100.00),
(2,'bob',200.00),
(3,'charlie',300.00);
```

첫 번째 트랜잭션을 시작합니다:

```
    => BEGIN;
    => SELECT txid_current(), pg_backend_pid();
    txid_current ¦ pg_backend_pid
   --------------+---------------
```

```
    122869 ¦ 30723
(1 row)
```

이제 행이 공유 모드로 잠겨있습니다:

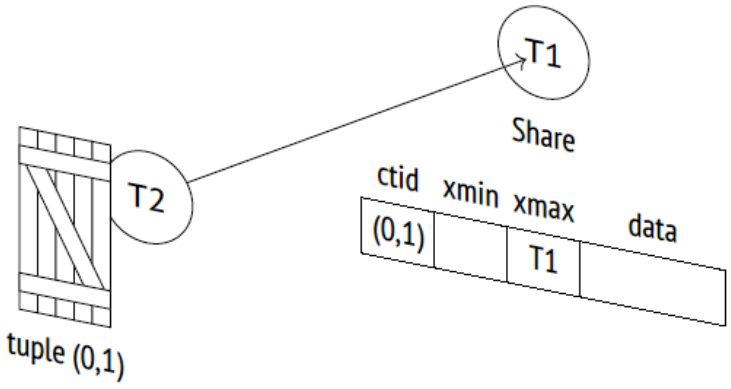
```
    => SELECT * FROM accounts WHERE id = 1 FOR SHARE;
```

두 번째 트랜잭션은 동일한 행을 수정하려고 시도하지만 허용되지 않습니다. 공유 모드와 비 키 수정 모드는 호환되지 않기 때문입니다:

```
    => BEGIN;
    => SELECT txid_current(), pg_backend_pid();
    txid_current ¦ pg_backend_pid
   --------------+----------------
          122870 ¦ 30794
    (1 row)
    => UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
```

첫 번째 트랜잭션이 완료될 때까지 기다리고 있는 동안, 두 번째 트랜잭션은 이전 예제와 마찬가지로 튜플 잠금을 보유하고 있습니다:

```
=> SELECT * FROM locks_accounts WHERE pid = 30794;
   pid ¦      locktype ¦        lockid ¦             mode ¦ granted
-------+---------------+---------------+------------------+---------
 30794 ¦      relation ¦      accounts ¦ RowExclusiveLock ¦ t
 30794 ¦ transactionid ¦        122869 ¦        ShareLock ¦ f
 30794 ¦ transactionid ¦        122870 ¦    ExclusiveLock ¦ t
 30794 ¦         tuple ¦ accounts(0,1) ¦    ExclusiveLock ¦ t
(4 rows)
```



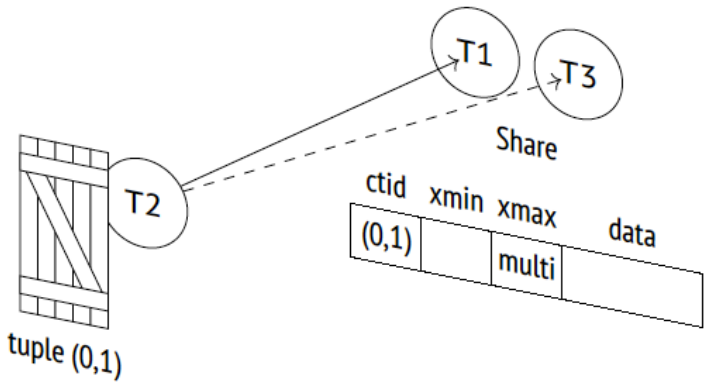
이제 세 번째 트랜잭션을 공유 모드로 행을 잠글 수 있습니다. 이러한 잠금은 이미 획득한 잠금과 호환되기 때문에 이 트랜잭션은 대기열을 건너뛰게 됩니다:

```
    => BEGIN;
    => SELECT txid_current(), pg_backend_pid();
```

```
 txid_current ¦ pg_backend_pid
--------------+----------------
       122871 ¦ 30865
  (1 row)
=> SELECT * FROM accounts WHERE id = 1 FOR SHARE;
```

이제 같은 행을 잠그고 있는 두 개의 트랜잭션이 있습니다:

```
=> SELECT * FROM pgrowlocks('accounts') \gx
-[ RECORD 1 ]---------------
locked_row ¦ (0,1)
    locker ¦ 2
     multi ¦ t
      xids ¦ {122869,122871}
     modes ¦ {Share,Share}
      pids ¦ {30723,30865}
```



이 시점에서 첫 번째 트랜잭션이 완료되면, 두 번째 트랜잭션은 여전히 행이 잠겨있는 것을 알고 대기열로 돌아갑니다. 그러나 이번에는 세 번째 트랜잭션 뒤에 위치하게 됩니다:

```
    => COMMIT;

=> SELECT * FROM locks_accounts WHERE pid = 30794;
   pid ¦      locktype ¦        lockid ¦.            mode ¦ granted
-------+---------------+---------------+------------------+---------
 30794 ¦      relation ¦      accounts ¦ RowExclusiveLock ¦ t
 30794 ¦ transactionid ¦        122870 ¦    ExclusiveLock ¦ t
 30794 ¦ transactionid ¦        122871 ¦        ShareLock ¦ f
 30794 ¦         tuple ¦ accounts(0,1) ¦    ExclusiveLock ¦ t
(4 rows)
```

세 번째 트랜잭션이 완료되면, 두 번째 트랜잭션이 업데이트를 수행할 수 있게 될 것입니다 (이 시간 동안 다른 공유 잠금이 발생하지 않는 한).

```
  => COMMIT;

  UPDATE 1
  => COMMIT;
```

외래 키 검사는 대개 키 속성이 변경되지 않으며 키 공유를 비 키 수정과 함께 사용할 수 있기 때문에 문제를 일으키지 않을 것입니다. 그러나 대부분의 경우, 응용 프로그램에서는 공유 행 수준 잠금을 피하는 것이 좋습니다.

13.5 비 대기 잠금

`SQL` 명령은 일반적으로 요청한 자원이 해제될 때까지 대기합니다. 그러나 때로는 잠금을 즉시 획득할 수 없는 경우 작업을 취소하는 것이 합리적일 수 있습니다. 이를 위해 `SELECT`, `LOCK`, `ALTER`와 같은 명령에는 `NOWAIT` 절이 제공됩니다.

한 행을 잠가보겠습니다:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 1;
```

요청한 자원이 잠금되어 있는 경우 `NOWAIT` 절을 사용한 명령은 즉시 오류로 완료됩니다:

```
   => SELECT * FROM accounts
      FOR UPDATE NOWAIT;
      ERROR: could not obtain lock on row in relation "accounts"
```

이러한 오류는 응용 프로그램 코드에서 잡아내고 처리할 수 있습니다.

`UPDATE`와 `DELETE` 명령에는 `NOWAIT` 절이 없습니다. 대신, `SELECT FOR UPDATE` 명령을 사용하여 행을 잠금하고 시도에 성공한 경우에만 업데이트하거나 삭제할 수 있습니다.

일부 특수한 경우에는 이미 잠금된 행을 건너뛰고 사용 가능한 행을 즉시 처리하는 것이 편리할 수 있습니다. 이것이 바로 `SKIP LOCKED` 절을 사용하여 실행되는 `SELECT FOR`의 역할입니다:

```
   => SELECT * FROM accounts
   ORDER BY id
   FOR UPDATE SKIP LOCKED
   LIMIT 1;
    id ¦     client ¦ amount
  ————+————————+————————
     2 ¦        bob ¦ 200.00
   (1 row)
```

이 예제에서는 첫 번째(이미 잠금된) 행이 건너뛰어지고, 쿼리가 두 번째 행을 잠금하고 반환되었습니다.

이 접근 방식을 사용하면 일괄로 행을 처리하거나 이벤트 큐의 병렬 처리를 설정할 수 있습니다. 그러나 이

명령에 대해 다른 사용 사례를 고안하는 것은 피해야 합니다. 대부분의 작업은 훨씬 간단한 방법을 사용하여 처리할 수 있습니다.

마지막으로, 타임아웃을 설정하여 긴 대기 시간을 피할 수 있습니다:

```
=> SET lock_timeout = '1s';
=> ALTER TABLE accounts DROP COLUMN amount;
ERROR: canceling statement due to lock timeout
```

이 명령은 1초 내에 잠금을 획득하지 못하여 오류로 완료됩니다. 타임아웃은 세션 수준뿐만 아니라 하위 수준에서도 설정할 수 있습니다. 예를 들어 특정 트랜잭션을 위해 타임아웃을 설정할 수 있습니다. 이 방법을 사용하면 부하가 있는 상황에서 배타적인 잠금을 필요로 하는 명령을 실행할 때 테이블 처리 중에 긴 대기 시간을 방지할 수 있습니다. 오류가 발생한 경우 잠시 후에 이 명령을 다시 시도할 수 있습니다.

> `statement_timeout`은 연산자 실행의 총 시간을 제한하는 반면, `lock_timeout` 매개변수는 잠금 대기 시간의 최대 시간을 정의합니다.

```
=> ROLLBACK;
```

13.6 교착 상태Deadlocks

교착 상태(deadlock)는 한 트랜잭션이 현재 다른 트랜잭션이 사용 중인 자원을 필요로 할 때 발생할 수 있습니다. 이 다른 트랜잭션은 또한 세 번째 트랜잭션에 의해 잠금이 걸려 대기 중인 경우가 있습니다. 이와 같은 트랜잭션들은 중량잠금을 사용하여 대기열에 들어갑니다.

그러나 가끔씩 대기열에 이미 있는 트랜잭션이 또 다른 자원을 필요로 할 경우, 동일한 대기열에 다시 참여하고 이 자원이 해제될 때까지 기다려야 합니다. 이러한 경우 **교착 상태**[146]가 발생합니다. 이제 대기열에는 순환 의존성이 있어 스스로 해결할 수 없는 상태가 되는 것입니다.

시각화를 위해 대기 그래프wait-for graph를 그려봅시다. 그래프의 노드는 활성 프로세스를 나타내고, 화살표로 표시된 간선은 잠금을 대기하는 프로세스에서 이 잠금을 보유한 프로세스로 향합니다. 그래프에 순환이 있는 경우, 즉 노드가 화살표를 따라 자기 자신에게 도달할 수 있는 경우 교착 상태가 발생한 것입니다.

> 이 예시에서는 프로세스 대신 트랜잭션을 보여줍니다. 이 대체는 일반적으로 허용됩니다. 왜냐하면 하나의 트랜잭션은 하나의 프로세스에 의해 실행되고, 잠금은 트랜잭션 내에서만 획득할 수 있기 때문입니다. 그러나 일반적으로는 프로세스에 관해 이야기하는 것이 더 정확합니다. 왜냐하면 트랜잭션이 완료되었을 때 일부 잠금이 즉시 해제되지 않을 수 있기 때문입니다.

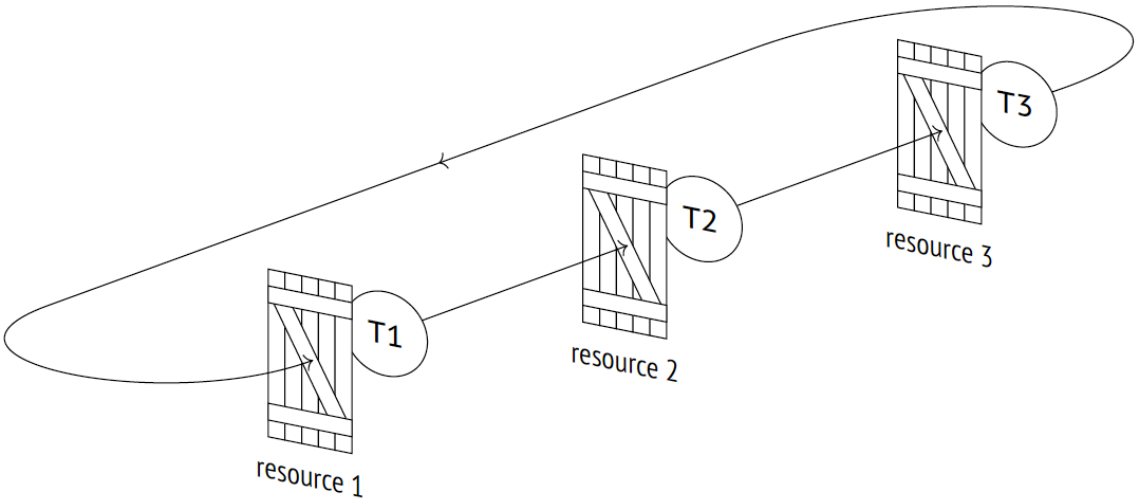
교착 상태가 발생하고 참가자 중 아무도 타임아웃을 설정하지 않은 경우, 트랜잭션들은 영원히 서로를 대기하게 됩니다. 이것이 잠금 관리자[147]가 자동으로 교착 상태를 감지하는 이유입니다.

---

[146] postgresql.org/docs/14/explicit-locking#LOCKING-DEADLOCKS.html

[147] backend/storage/lmgr/README

그러나 이러한 확인은 일정한 노력을 필요로 합니다. 매번 잠금이 요청될 때마다 이 노력을 낭비해서는 안 됩니다(왜냐하면 교착 상태는 자주 발생하지 않기 때문입니다). 따라서 프로세스가 락을 요청할 때마다 이러한 확인을 수행하는 것은 효율적이지 않습니다.



잠금을 획득하려는 시도가 실패하고 대기열에 참여한 후 프로세스가 잠들면, PostgreSQL은 `dealock_timeout`(기본값: 1초) 매개변수[148]로 정의된 시간까지 자동으로 타임아웃을 설정합니다. 리소스가 더 빨리 사용 가능해진다면 추가적인 확인 비용을 피할 수 있습니다. 그러나 교착 상태 타임아웃 시간이 지나도록 대기가 계속된다면, 대기 중인 프로세스는 깨어나서 확인을 시작합니다.[149]

이 확인은 실제로는 대기 그래프를 구성하고 사이클을 검색하는 것입니다.[150] 현재 그래프의 상태를 "고정"하기 위해 PostgreSQL은 확인하는 동안 중량잠금의 처리를 전체적으로 중지합니다.

교착 상태가 감지되지 않으면, 프로세스는 다시 잠들게 됩니다. 언젠가는 차례가 올 것입니다.

교착 상태가 감지되면, 트랜잭션 중 하나가 강제로 종료되어 잠금이 해제되고 다른 트랜잭션이 실행을 계속할 수 있게 됩니다. 대부분의 경우, 확인을 시작한 트랜잭션이 중단되지만, 사이클에 현재 튜플 블록을 방지하기 위해 튜플을 동결하지 않는 `autovacuum` 프로세스가 포함된 경우 서버는 우선 순위가 낮은 `autovacuum`을 종료합니다.

교착 상태는 일반적으로 잘못된 애플리케이션 설계를 나타냅니다. 이러한 상황을 발견하기 위해 서버 로그의 해당 메시지와 `pg_stat_database` 테이블에서 증가하는 교착 상태 값에 주의해야 합니다.

행 수정에 따른 교착 상태

교착 상태는 궁극적으로 중량잠금에 의해 발생하지만, 주로 서로 다른 순서로 획득된 행 수준 잠금이 교착 상태를 야기합니다.

---

148 backend/storage/lmgr/proc.c, ProcSleep function

149 backend/storage/lmgr/proc.c, CheckDeadLock function

150 backend/storage/lmgr/deadlock.c

예를 들어, 한 트랜잭션이 두 개의 계정 간에 $100을 이체하는 경우를 가정해봅시다. 첫 번째 계정에서 이 금액을 인출하여 시작합니다:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount - 100.00 WHERE id = 1;
UPDATE 1
```

동시에 다른 트랜잭션이 두 번째 계정에서 첫 번째 계정으로 $10을 이체하려고 합니다. 먼저 두 번째 계정에서 이 금액을 인출합니다:

```
    => BEGIN;
    => UPDATE accounts SET amount = amount - 10.00 WHERE id = 2;
    UPDATE 1
```

이제 첫 번째 트랜잭션이 두 번째 계정의 금액을 증가시키려고 시도하지만 해당하는 행이 잠금이되어 있음을 알 수 있습니다:

```
=> UPDATE accounts SET amount = amount + 100.00 WHERE id = 2;
```

그리고 두 번째 트랜잭션은 첫 번째 계정을 수정하려고 시도하지만 잠금을 받지 못합니다:

```
    => UPDATE accounts SET amount = amount + 10.00 WHERE id = 1;
```

이러한 순환 대기 상태는 스스로 해결되지 않습니다. 첫 번째 트랜잭션은 1초 내에 리소스를 획득할 수 없으므로 교착 상태 확인을 시작하고 서버에 의해 중단됩니다:

```
ERROR: deadlock detected
DETAIL: Process 30423 waits for ShareLock on transaction 122877;
blocked by process 30723.
Process 30723 waits for ShareLock on transaction 122876; blocked by
process 30423.
HINT: See server log for query details.
CONTEXT: while updating tuple (0,2) in relation "accounts"
```

이제 두 번째 트랜잭션이 계속할 수 있습니다. 그것은 깨어나서 수정을 수행합니다:

```
UPDATE 1
```

트랜잭션을 완료해봅시다.

```
    => ROLLBACK;

=> ROLLBACK;
```

이와 같은 작업을 수행하는 올바른 방법은 동일한 순서로 리소스를 잠금을 하는 것입니다. 예를 들어, 이 경우에는 계정 번호를 기준으로 오름차순으로 계정을 잠금 할 수 있었습니다.

두개의 UPDATE 구문 사이의 교착 상태

일부 경우에는 교착 상태가 불가능한 것처럼 보이지만 실제로 발생할 수 있습니다.

일반적으로 SQL 명령은 원자적atomic이라고 가정하지만, 정말 그럴까요? UPDATE 명령을 더 자세히 살펴보겠습니다. 이 명령은 한 번에 모든 행을 잠금하는 대신 수정되는 행을 잠금하며, 동시에 발생하지 않습니다. 따라서 한 UPDATE 명령이 특정한 순서로 여러 행을 수정하고, 다른 UPDATE 명령이 다른 순서로 동일한 작업을 수행하는 경우 교착 상태가 발생할 수 있습니다.

이러한 시나리오를 재현해보겠습니다. 먼저, amount 열에 내림차순으로 인덱스를 작성합니다:

```
=> CREATE INDEX ON accounts(amount DESC);
```

과정을 관찰할 수 있도록 작업을 늦추는 함수를 작성할 수 있습니다:

```
=> CREATE FUNCTION inc_slow(n numeric)
RETURNS numeric
AS $$
  SELECT pg_sleep(1);
  SELECT n + 100.00;
$$ LANGUAGE sql;
```

첫 번째 UPDATE 명령은 모든 튜플을 수정할 것입니다. 실행 계획은 전체 테이블에 관한 순차 검색을 기반으로 합니다.

```
=> EXPLAIN (costs off)
UPDATE accounts SET amount = inc_slow(amount);
QUERY PLAN
----------------------------
Update on accounts
-> Seq Scan on accounts
(2 rows)
```

amount 열을 기준으로 힙 페이지가 행을 오름차순으로 저장하도록 하기 위해, 테이블을 잘라내고 새로운 행을 삽입해야 합니다.

```
=> TRUNCATE accounts;
=> INSERT INTO accounts(id, client, amount)
VALUES
(1,'alice',100.00),
(2,'bob',200.00),
(3,'charlie',300.00);

=> ANALYZE accounts;
=> SELECT ctid, * FROM accounts;
  ctid ¦ id ¦  client ¦ amount
```

```
-------+----+---------+--------
  (0,1) ¦ 1 ¦   alice ¦ 100.00
  (0,2) ¦ 2 ¦     bob ¦ 200.00
  (0,3) ¦ 3 ¦ charlie ¦ 300.00
(3 rows)
```

순차 검색은 행을 동일한 순서로 수정할 것입니다 (하지만 대규모 테이블의 경우 항상 그렇지는 않습니다).

수정을 시작해봅시다:

```
    => UPDATE accounts SET amount = inc_slow(amount);
```

한편, 다른 세션에서 순차 검색을 금지해보겠습니다:

```
=> SET enable_seqscan = off;
```

결과적으로, 플래너는 다음 `UPDATE` 명령에 관해 인덱스 스캔을 선택합니다.

```
=> EXPLAIN (costs off)
UPDATE accounts SET amount = inc_slow(amount)
WHERE amount > 100.00;

                   QUERY PLAN
-----------------------------------------------------
Update on accounts
-> Index Scan using accounts_amount_idx on accounts
    Index Cond: (amount > 100.00)
(3 rows)
```

두 번째와 세 번째 행이 조건을 만족합니다. 인덱스가 내림차순이므로 행은 역순으로 수정될 것입니다.

다음 수정을 시작해봅시다:

```
    => UPDATE accounts SET amount = inc_slow(amount)
    WHERE amount > 100.00;
```

`pgrowlocks` 확장 기능을 사용하면 첫 번째 연산자가 이미 첫 번째 행 (0,1)을 수정하였고, 두 번째 연산자가 마지막 행 (0,3)을 수정한 것을 확인할 수 있습니다.

```
=> SELECT locked_row, locker, modes FROM pgrowlocks('accounts');
 locked_row ¦ locker ¦ modes
------------+--------+------------------
      (0,1) ¦ 122883 ¦ {"No Key Update"}. <- first
      (0,3) ¦ 122884 ¦ {"No Key Update"}. <- second
(2 rows)
```

1초가 더 흐릅니다. 첫 번째 연산자가 두 번째 행을 수정하였고, 다른 연산자도 동일한 작업을 수행하려고 하지만 허용되지 않습니다.

```
=> SELECT locked_row, locker, modes FROM pgrowlocks('accounts');
 locked_row ¦ locker ¦ modes
------------+--------+------------------
      (0,1) ¦ 122883 ¦ {"No Key Update"}
      (0,2) ¦ 122883 ¦ {"No Key Update"} <- the first one wins
      (0,3) ¦ 122884 ¦ {"No Key Update"}
(3 rows)
```

이제 첫 번째 연산자는 마지막 테이블 행을 수정하려고 합니다. 그러나 해당 행은 이미 두 번째 연산자에 의해 잠금이 걸려 있습니다. 교착 상태가 발생했습니다.

두 트랜잭션 중 하나가 중단됩니다:

```
ERROR: deadlock detected
DETAIL: Process 30794 waits for ShareLock on transaction 122883;
blocked by process 30723.
Process 30723 waits for ShareLock on transaction 122884; blocked by
process 30794.
HINT: See server log for query details.
CONTEXT: while updating tuple (0,2) in relation "accounts"
```

그리고 다른 트랜잭션은 실행을 완료합니다:

```
UPDATE 3
```

고립된 상황처럼 보이지만, 대량의 행 수정이 수행되는 고부하 시스템에서는 이러한 상황이 발생할 수 있습니다.

14 장 기타 잠금

14.1 비객체 잠금

관계로 간주되지 않는 리소스를 잠그기 위해 PostgreSQL은 객체 유형[151]의 중량있는 잠금을 사용합니다. 시스템 카탈로그에 저장된 거의 모든 것을 잠글 수 있습니다. 테이블스페이스, 구독, 스키마, 역할, 정책, 열거형 데이터 유형 등과 같은 것들이 포함됩니다.

테이블을 생성하는 트랜잭션을 시작해 보겠습니다:

```
=> BEGIN;
=> CREATE TABLE example(n integer);
```

이제 `pg_locks` 테이블에서 관계가 아닌 잠금을 살펴보세요:

```
=> SELECT database,
         (
           SELECT datname FROM pg_database WHERE oid = database
         ) AS dbname,
         classid,
        (
          SELECT relname FROM pg_class WHERE oid = classid
        ) AS classname,
        objid,
        mode,
        granted
 FROM pg_locks
WHERE locktype = 'object'
  AND pid = pg_backend_pid() \gx

-[ RECORD 1 ]--------------
 database ¦ 16391
   dbname ¦ internals
  classid ¦ 2615
classname ¦ pg_namespace
    objid ¦ 2200
     mode ¦ AccessShareLock
  granted ¦ t
```

잠긴 리소스는 다음 세 가지 값에 의해 정의됩니다:

database - 잠금 대상 객체가 포함된 데이터베이스의 `oid` (또는 이 객체가 전체 클러스터에 공통인 경우 0)

[151] backend/storage/lmgr/lmgr.c, LockDatabaseObject & LockSharedObject functions

classid - 리소스 유형을 정의하는 시스템 카탈로그 테이블 이름에 해당하는 `pg_class`에 나열된 `oid`
objid - `classid`가 참조하는 시스템 카탈로그 테이블에 나열된 `oid`

`database` 값은 내부 데이터베이스를 가리킵니다. 현재 세션이 연결된 데이터베이스입니다. `classid` 열은 스키마를 나열하는 `pg_namespace` 테이블을 가리킵니다.

이제 `objid`를 해석할 수 있습니다:

```
=> SELECT nspname FROM pg_namespace WHERE oid = 2200;
nspname
─────────
public
(1 row)
```

따라서 PostgreSQL은 트랜잭션이 실행 중인 동안 누구도 삭제하지 못하도록 `public` 스키마를 잠갔습니다.

마찬가지로, 객체 삭제에는 객체 자체와 해당 객체가 의존하는 모든 리소스에 관한 배타적 잠금이 필요합니다.[152]

```
=> ROLLBACK;
```

14.2 관계 확장 잠금

관계의 튜플 수가 증가함에 따라 PostgreSQL은 가능한 경우 이미 사용 가능한 페이지의 빈 공간에 새로운 튜플을 삽입합니다. 그러나 어느 시점에서는 관계를 확장하기 위해 새로운 페이지를 추가해야 할 것입니다. 물리적 레이아웃 관점에서 새로운 페이지는 해당 파일의 끝에 추가됩니다 (이로 인해 새로운 파일이 생성될 수도 있음).

새로운 페이지가 한 번에 한 프로세스에 의해서만 추가되기 위해 이 작업은 특별한 중량있는 확장 유형[153]의 잠금으로 보호됩니다. 이러한 잠금은 인덱스 진공 작업에서도 인덱스 스캔 중에 새로운 페이지 추가를 금지하는 데 사용됩니다.

관계 확장 잠금은 지금까지 본 것과 약간 다르게 동작합니다:

- 확장이 생성되면 트랜잭션이 완료될 때까지 기다리지 않고 즉시 해제됩니다.
- 데드락을 유발할 수 없으므로 대기 그래프에 포함되지 않습니다.

> 그러나, 관계를 확장하는 절차가 `deadlock_timeout` 보다 오랜 시간이 걸리는 경우에는 여전히 데드락 검사가 수행됩니다. 이는 일반적인 상황은 아니지만 여러 프로세스가 동시에 다중 삽입을 수행하는 경우에 발생할 수 있습니다. 이 경우, 검사가 여러 번 호출되어 정상적인 시스템 작동이 사실상 마비될 수 있습니다.

[152] backend/catalog/dependency.c, performDeletion function

[153] backend/storage/lmgr/lmgr.c, LockRelationForExtension function

이러한 위험을 최소화하기 위해 힙 파일은 한 번에 여러 페이지씩 확장됩니다 (잠금을 기다리는 프로세스 수에 비례하되 한 번에 512 페이지를 초과하지 않도록)[154]. 이 규칙의 예외는 B-트리 인덱스 파일로, 한 번에 한 페이지씩 확장됩니다.[155]

14.3 페이지 잠금

페이지 수준[156]의 중량있는 잠금인 페이지 유형은 `GIN` 인덱스에만 적용되며, 다음과 같은 경우에만 적용됩니다.

`GIN` 인덱스는 텍스트 문서의 단어와 같은 복합 값의 요소를 검색하는 속도를 높일 수 있습니다. `GIN` 인덱스는 문서 전체가 아닌 개별 단어를 저장하는 B-트리와 비슷한 구조로 대략적으로 설명할 수 있습니다. 새로운 문서가 추가되면 해당 문서에 나타나는 각 단어를 포함하기 위해 인덱스를 철저하게 수정해야 합니다.

성능을 향상시키기 위해 `GIN` 인덱스는 지연된 삽입을 허용합니다. 이는 **`fastupdate`** (기본값: `on`) 저장 매개변수에 의해 제어됩니다. 새로운 단어는 먼저 정렬되지 않은 대기 목록에 빠르게 추가되고, 어느 정도 시간이 지난 후에 모든 누적된 항목이 주 인덱스 구조로 이동됩니다. 서로 다른 문서에 중복된 단어가 포함될 가능성이 높기 때문에 이 접근 방식은 비용 효율적입니다.

여러 프로세스에 의한 단어의 동시 전송을 피하기 위해 인덱스 메타페이지는 주 인덱스로 단어가 모두 이동될 때까지 배타적인 모드로 잠깁니다. 이 잠금은 일반적인 인덱스 사용에는 영향을 주지 않습니다.

관계 확장 잠금과 마찬가지로 페이지 잠금은 작업이 완료되면 트랜잭션의 종료를 기다리지 않고 즉시 해제되므로 데드락을 발생시키지 않습니다.

14.4 조언적 잠금

다른 중량있는 잠금 (관계 잠금과 같은)과 달리, 조언적 잠금[157]은 자동으로 획득되지 않습니다. 이 잠금은 응용 프로그램 개발자에 의해 제어됩니다. 이러한 잠금은 응용 프로그램이 특정 목적을 위해 전용 잠금 로직이 필요한 경우 편리하게 사용할 수 있습니다.

예를 들어, `SELECT FOR` 또는 `LOCK TABLE` 명령을 사용하여 잠글 수 있는 데이터베이스 객체와 대응하지 않는 리소스를 잠그려면 리소스에 숫자 `ID`를 할당해야 합니다. 리소스에 고유한 이름이 있다면 해당 이름에 관한 해시 코드를 생성하는 것이 가장 쉬운 방법입니다.

```
=> select hashtext('resource1');
hashtext
------------
991601810
(1 row)
```

[154] backend/access/heap/hio.c, RelationAddExtraBlocks function

[155] backend/access/nbtree/nbtpage.c, _bt_getbuf function

[156] backend/storage/lmgr/lmgr.c, LockPage function

[157] postgresql.org/docs/14/explicit-locking#ADVISORY-LOCKS.html

PostgreSQL은 조언적 잠금[158]을 관리하기 위한 전체 함수 클래스를 제공합니다. 함수 이름은`pg_advisory` 접두사로 시작하며 다음과 같은 목적을 힌트로 하는 단어를 포함할 수 있습니다.

- `lock`: 잠금 획득
- `try`: 대기 없이 잠금 획득을 시도
- `unlock`: 잠금 해제
- `share`: 공유 잠금 모드 사용 (기본적으로 배타적 모드 사용)
- `xact`: 트랜잭션의 끝까지 잠금 획득 및 유지 (기본적으로 세션의 끝까지 잠금 유지)

세션의 끝까지 배타적 잠금을 획득해 보겠습니다:

```
=> BEGIN;
=> SELECT pg_advisory_lock(hashtext('resource1'));
=> SELECT locktype, objid, mode, granted
   FROM pg_locks WHERE locktype = 'advisory' AND pid = pg_backend_pid();

 locktype ¦     objid ¦          mode ¦ granted
----------+-----------+---------------+---------
 advisory ¦ 991601810 ¦ ExclusiveLock ¦ t
(1 row)
```

조언적 잠금이 실제로 작동하려면 다른 프로세스도 리소스에 액세스할 때 설정된 순서를 준수해야 합니다. 이는 응용 프로그램에 의해 보장되어야 합니다.

획득한 잠금은 트랜잭션이 완료된 후에도 유지됩니다:

```
=> COMMIT;
=> SELECT locktype, objid, mode, granted
FROM pg_locks WHERE locktype = 'advisory' AND pid = pg_backend_pid();
 locktype ¦     objid ¦          mode ¦ granted
----------+-----------+---------------+---------
 advisory ¦ 991601810 ¦ ExclusiveLock ¦ t
(1 row)
```

리소스에 관한 작업이 완료되면 잠금을 명시적으로 해제해야 합니다:

```
=> SELECT pg_advisory_unlock(hashtext('resource1'));
```

14.5 조건부 잠금

조건부 잠금이라는 용어는 잠금을 기반으로 한 완전한 격리를 구현하려는 최초의 시도에서 등장했습니다.[159] 당시 직면한 문제는 읽고 업데이트할 모든 행을 잠그더라도 완전한 격리를 보장할 수 없었다는 것입니다. 실제로, 테이블에 조건을 만족하는 새로운 행이 삽입되면 그 행은 유령 행이 될 수 있습니다.

---

[158] postgresql.org/docs/14/functions-admin#FUNCTIONS-ADVISORY-LOCKS.html

[159] K. P. Eswaran, J. N. Gray, R. A. Lorie, I. L. Traiger. The notions of consistency and predicate locks in a database system

이러한 이유로 행이 아닌 조건(술어)을 잠그는 것을 제안했습니다. 예를 들어 a > 10 조건을 가진 쿼리를 실행한다면, 이 술어를 잠그면 이 조건을 만족하는 새로운 행을 테이블에 추가할 수 없으므로 유령 행이 발생하지 않습니다. 문제는 a < 20과 같은 다른 술어가 나타나면 이러한 술어들이 겹치는지 여부를 확인해야 한다는 것입니다. 이론적으로 이 문제는 알고리즘적으로 해결할 수 없습니다. 실제로는 이 예시와 같이 매우 간단한 클래스의 술어에 관해서만 해결할 수 있습니다.

PostgreSQL에서 직렬화 격리 수준은 다른 방식으로 구현됩니다. 직렬화 격리 수준은 `Serializable Snapshot Isolation` (`SSI`) 프로토콜[160]을 사용합니다. 조건부 잠금이라는 용어는 여전히 사용되지만, 그 의미는 근본적으로 바뀌었습니다. 사실, 이러한 잠금은 아무 것도 잠그지 않습니다. 대신, 이들은 서로 다른 트랜잭션 간의 데이터 종속성을 추적하는 데 사용됩니다.

반복 가능한 읽기 수준에서 스냅숏 격리는 쓰기 기울기(`write skew`) 및 읽기 전용 트랜잭션 이상 현상을 제외한 모든 이상 현상을 허용하지 않음이 증명되었습니다. 이 두 가지 이상 현상은 상대적으로 적은 비용으로 데이터 종속성 그래프에서 특정 패턴을 발견할 수 있습니다.

문제는 두 가지 종류의 종속성을 구별해야 한다는 것입니다:

- 첫 번째 트랜잭션이 두 번째 트랜잭션에 의해 나중에 수정되는 행을 읽음 (`RW` 종속성).
- 첫 번째 트랜잭션이 두 번째 트랜잭션에 의해 나중에 읽히는 행을 수정함 (`WR` 종속성).

WR 종속성은 일반적인 락을 사용하여 감지할 수 있지만, RW 종속성은 조건부 잠금을 통해 추적해야 합니다. 이러한 추적은 직렬화 격리 수준에서 자동으로 활성화되며, 이것이 모든 트랜잭션 (또는 적어도 상호 연결된 트랜잭션)에 대해 이 수준을 사용하는 것이 중요한 이유입니다. 다른 수준에서 실행 중인 트랜잭션이 있는 경우 조건부 잠금을 설정하지 않으므로 직렬화 수준은 반복적인 읽기 수준으로 내려가게 됩니다.

다시 한 번 강조하고 싶은 것은 조건부 잠금이라는 이름에도 불구하고, 조건부 잠금은 실제로 아무 것도 잠그지 않습니다. 대신, 트랜잭션이 커밋될 때 "위험한" 종속성을 확인하고, PostgreSQL이 이상 현상을 의심하면 해당 트랜잭션은 중단됩니다.

다음은 여러 페이지에 걸친 인덱스를 가진 테이블을 생성하는 예시입니다 (낮은 `fillfactor` 값을 사용하여 구현할 수 있습니다):

```
CREATE TABLE pred(n numeric, s text);

INSERT INTO pred(n) SELECT n FROM generate_series(1,10000) n;

CREATE INDEX ON pred(n) WITH (fillfactor = 10);

ANALYZE pred;
```

만약 쿼리가 순차 스캔을 수행하는 경우, 제공된 필터 조건을 만족시키지 않는 일부 행이 있더라도 전체 테이

[160] backend/storage/lmgr/README-SSI
backend/storage/lmgr/predicate.c

블에 대해 조건부 잠금이 획득됩니다.

```
=> SELECT pg_backend_pid();
   pg_backend_pid
 ----------------
            34753
(1 row)
=> BEGIN ISOLATION LEVEL SERIALIZABLE;
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM pred WHERE n > 100;
QUERY PLAN
-------------------------------------------
Seq Scan on pred (actual rows=9900 loops=1)
Filter: (n > '100'::numeric)
Rows Removed by Filter: 100
```

위의 예시에서는 n > 100 조건을 만족하는 행을 조회하는 쿼리입니다. 이 쿼리가 순차 스캔을 수행하므로 전체 테이블에 대한 조건부 잠금이 획득됩니다.

조건부 잠금은 자체 인프라를 가지고 있지만, `pg_locks` 뷰는 무겁게 잠긴 잠금과 함께 이들을 함께 표시합니다. 모든 조건부 잠금은 항상 직렬화 격리 수준의 읽기(`SIRead`) 모드로 획득됩니다.

```
=> SELECT relation::regclass, locktype, page, tuple
FROM pg_locks WHERE mode = 'SIReadLock' AND pid = 34753
ORDER BY 1, 2, 3, 4;
 relation ¦ locktype ¦ page ¦ tuple
----------+----------+------+-------
     pred ¦ relation ¦      ¦
(1 row)

=> ROLLBACK;
```

위의 예시는 `SIReadLock` 모드로 획득된 조건부 잠금을 조회하는 것입니다. 조건부 잠금은 트랜잭션 간의 종속성을 추적하는 데 사용되므로, 트랜잭션의 지속 시간보다 오래 유지될 수 있습니다. 그러나 이들은 자동으로 관리됩니다.

인덱스 스캔이 수행되는 경우 상황이 개선됩니다. B-트리 인덱스의 경우, 읽은 힙 튜플과 스캔된 리프 페이지에 대한 조건부 잠금을 설정하는 것만으로도 충분합니다. 이렇게 함으로써 정확한 값뿐만 아니라 읽은 전체 범위를 "잠그게" 됩니다.

```
=> BEGIN ISOLATION LEVEL SERIALIZABLE;
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM pred WHERE n BETWEEN 1000 AND 1001;
QUERY PLAN
-------------------------------------------------------------------
Index Scan using pred_n_idx on pred (actual rows=2 loops=1)
```

```
  Index Cond: ((n >= '1000'::numeric) AND (n <= '1001'::numeric))
  (2 rows)

=> SELECT relation::regclass, locktype, page, tuple
FROM pg_locks WHERE mode = 'SIReadLock' AND pid = 34753
ORDER BY 1, 2, 3, 4;
   relation | locktype | page | tuple
------------+----------+------+-------
       pred |    tuple |    4 | 96
       pred |    tuple |    4 | 97
 pred_n_idx |     page |   28 |
(3 rows)
```

이미 스캔된 튜플에 해당하는 리프 페이지의 수는 변경될 수 있습니다. 예를 들어, 새로운 행이 테이블에 삽입되면 인덱스 페이지가 분할될 수 있습니다. 그러나 PostgreSQL은 이를 고려하여 새로 생성된 페이지도 잠그게 됩니다.

```
=> INSERT INTO pred
SELECT 1000+(n/1000.0) FROM generate_series(1,999) n;

=> SELECT relation::regclass, locktype, page, tuple
FROM pg_locks WHERE mode = 'SIReadLock' AND pid = 34753
ORDER BY 1, 2, 3, 4;
   relation | locktype | page | tuple
------------+----------+------+-------
       pred |    tuple |    4 | 96
       pred |    tuple |    4 | 97
 pred_n_idx |     page |   28 |
 pred_n_idx |     page |  266 |
 pred_n_idx |     page |  267 |
 pred_n_idx |     page |  268 |
 pred_n_idx |     page |  269 |
(7 rows)
```

위의 예시에서는 새로운 행을 삽입하여 페이지가 추가되는 것을 확인할 수 있습니다. `SIReadLock` 모드로 획득된 조건부 잠금이 페이지에 대해 설정되는 것을 확인할 수 있습니다.

각 읽은 튜플은 개별적으로 잠겨지며, 이러한 튜플은 상당히 많을 수 있습니다. 조건부 잠금은 서버 시작 시 할당된 자체 풀을 사용합니다. 조건부 잠금의 총 수는 `max_pred_locks_per_transaction` (기본값: 64) 값에 `max_connections` (기본값: 100)을 곱한 것으로 제한됩니다 (매개변수 이름에도 불구하고, 조건부 잠금은 개별 트랜잭션 당으로 계산되지 않습니다).

여기서는 행 수준 잠금과 동일한 문제가 발생하지만, 다른 방식으로 해결됩니다: 잠금 에스컬레이션이 적용

됩니다.[161]

하나의 페이지에 관련된 튜플 잠금 수가 max_pred_locks_per_page(기본값: 2) 매개변수의 값보다 크면, 이들은 단일한 페이지 수준 잠금으로 대체됩니다.

```
  => EXPLAIN (analyze, costs off, timing off, summary off)
  SELECT * FROM pred WHERE n BETWEEN 1000 AND 1002;
  QUERY PLAN
  ------------------------------------------------------------------------
  Index Scan using pred_n_idx on pred (actual rows=3 loops=1)
      Index Cond: ((n >= '1000'::numeric) AND (n <= '1002'::numeric))
  (2 rows)
```

튜플 타입의 세 개의 잠금 대신에 이제 페이지 타입의 하나의 잠금이 있습니다:

```
=> SELECT relation::regclass, locktype, page, tuple
FROM pg_locks WHERE mode = 'SIReadLock' AND pid = 34753
ORDER BY 1, 2, 3, 4;
   relation ¦ locktype ¦ page ¦ tuple
------------+----------+------+-------
       pred ¦     page ¦    4 ¦
 pred_n_idx ¦     page ¦   28 ¦
 pred_n_idx ¦     page ¦  266 ¦
 pred_n_idx ¦     page ¦  267 ¦
 pred_n_idx ¦     page ¦  268 ¦
 pred_n_idx ¦     page ¦  269 ¦
 (6 rows)

=> ROLLBACK;
```

페이지 수준의 잠금의 확대는 동일한 원칙을 따릅니다. 특정 관계에 대한 이러한 잠금의 수가 **max_pred_locks_per_relation**(기본값: -2) 값보다 크면, 하나의 관계 수준 잠금으로 대체됩니다. (이 매개변수가 음수로 설정되면 임계값은 **max_pred_locks_per_transaction**(기본값: 64)을 **max_pred_locks_per_relation**의 절대값으로 나눈 것으로 계산됩니다. 따라서 기본 임곗값은 32입니다). 잠금 확대는 여러 가지 잘못된 직렬화 오류를 유발하여 시스템 처리량에 부정적인 영향을 미칩니다. 따라서 성능과 사용 가능한 RAM에 대한 적절한 균형을 찾아야 합니다.

조건부 잠금은 다음과 같은 인덱스 유형을 지원합니다:

- B-트리
- 해시 인덱스, GiST 및 GIN

인덱스 스캔이 수행되지만 인덱스가 조건부 잠금을 지원하지 않는 경우 전체 인덱스가 잠깁니다. 이 경우에는 무의미한 이유로 중단된 트랜잭션의 수도 증가할 것으로 예상됩니다.

---

[161] backend/storage/lmgr/predicate.c, PredicateLockAcquire function

직렬화 수준에서 보다 효율적인 작업을 위해 `READ ONLY` 절을 사용하여 명시적으로 읽기 전용 트랜잭션을 선언하는 것이 좋습니다.[162] 잠금 관리자가 읽기 전용 트랜잭션이 다른 트랜잭션과 충돌하지 않을 것을 확인하면 이미 설정된 조건부 잠금을 해제하고 새로운 잠금을 획득하지 않을 수 있습니다. 그리고 이러한 트랜잭션을 `DEFERABLE`로 선언하면 읽기 전용 트랜잭션 이상현상도 피할 수 있습니다.

[162] backend/storage/lmgr/predicate.c, SxactIsROSafe macro

15 장 메모리 구조의 잠금

15.1 스핀잠금Spinlocks

PostgreSQL은 일반적인 무거운 잠금 대신에 가벼우면서도 비용이 적은 여러 유형의 잠금을 사용하여 공유 메모리의 데이터 구조를 보호합니다.

가장 간단한 잠금은 스핀잠금(`spinlock`)입니다. 스핀락은 보통 매우 짧은 시간 동안(여러 `CPU` 사이클 이내) 특정 메모리 셀을 동시 업데이트로부터 보호하기 위해 획득됩니다.

스핀잠금은 비교과 교체(`compare-and-swap`)[163]와 같은 원자적인 `CPU` 명령을 기반으로 합니다. 스핀잠금은 배타적인 잠금 모드만 지원합니다. 필요한 리소스가 이미 잠겨 있는 경우 프로세스는 명령을 반복하여 바쁜 대기(`busy-wait`)를 수행합니다(따라서 이름이 "스핀"인 것입니다). 지정된 시간 내에 잠금을 획득할 수 없는 경우 프로세스는 잠시 일시 정지한 후 다른 루프를 시작합니다.

이 전략은 충돌 확률이 매우 낮다고 추정되는 경우에 의미가 있습니다. 따라서 실패한 시도 이후에는 잠금이 몇 개의 명령 내에서 획득될 가능성이 높습니다.

스핀잠금은 교착 상태 감지나 계측 기능이 없습니다. 실용적인 관점에서는 스핀잠금의 존재를 알고 있으면 되며, 올바른 구현에 대한 전체 책임은 PostgreSQL 개발자에게 있습니다.

15.2 경량 잠금

그 다음으로, 경량 잠금 또는 `lwlock`[164]이라고 불리는 것이 있습니다. 가벼운 잠금은 데이터 구조(예: 해시 테이블 또는 포인터 목록)을 처리하는 데 필요한 시간 동안 획득되며, 일반적으로 짧은 시간이 소요됩니다. 그러나 `I/O` 작업을 보호하는 데 사용될 때는 더 오랜 시간이 걸릴 수도 있습니다.

가벼운 잠금은 배타적 모드(데이터 수정용)와 공유 모드(읽기 전용 작업용)를 지원합니다. 큐는 따로 존재하지 않습니다. 잠금을 기다리는 여러 프로세스가 있는 경우, 어느 한 프로세스가 비교적 무작위로 자원에 접근할 수 있습니다. 다중 동시 프로세스가 있는 고부하 시스템에서는 일부 불편한 효과를 초래할 수 있습니다.

교착 상태 검사는 제공되지 않습니다. 가벼운 잠금이 올바르게 구현되었다는 점에 대해 PostgreSQL 개발자들을 신뢰해야 합니다. 그러나 이러한 잠금은 계측 기능을 갖고 있으므로 스핀잠금과 달리 관찰할 수 있습니다.

15.3 예제

스핀잠금과 경량 잠금이 어떻게 사용되고 어디에서 사용될 수 있는지 알아보기 위해 버퍼 캐시와 `WAL` 버퍼라는 두 가지 공유 메모리 구조를 살펴보겠습니다. 모든 잠금을 다 나열하기는 너무 복잡하고 PostgreSQL 핵심 개발자들에게만 흥미로울 것입니다.

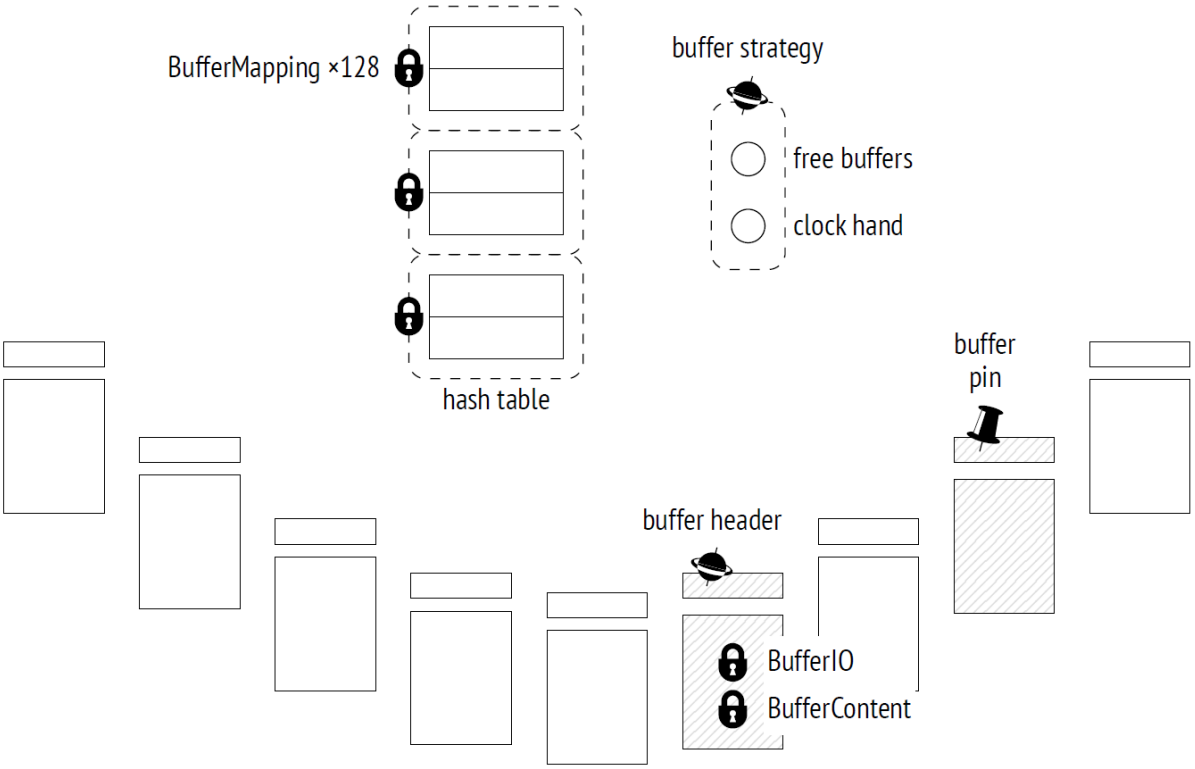
---

[163] backend/storage/lmgr/s_lock.c

[164] backend/storage/lmgr/lwlock.c

버퍼 캐시

버퍼 캐시에서 특정 버퍼를 찾기 위해 사용되는 해시 테이블에 접근하려면 프로세스는 **버퍼 매핑**BufferMapping 경량 잠금을 획득해야 합니다. 읽기를 위해서는 공유 모드로, 수정이 예상되는 경우에는 배타적 모드로 획득해야 합니다.



해시 테이블은 매우 빈번하게 접근되기 때문에 이 잠금은 종종 병목 현상이 됩니다. 더 세분화하기 위해 해시 테이블은 128개의 개별 경량 잠금으로 구성된 트랜치로 구성됩니다.[165]

> 해시 테이블 잠금은 PostgreSQL 8.2 에서 16 개의 잠금으로 변환되었으며, 10 년 후인 버전 9.5 가 출시될 때 트랜치의 크기가 128 로 증가되었지만, 현대의 멀티코어 시스템에는 여전히 충분하지 않을 수 있습니다.

버퍼 헤더에 액세스하려면 프로세스는 버퍼 헤더 스핀잠금[166]을 획득합니다(스핀잠금은 사용자가 볼 수 있는 이름이 없으므로 이름은 임의로 지정됩니다). 사용량 카운터를 증가시키는 등의 일부 작업은 명시적인 잠금이 필요하지 않으며 원자적인 `CPU` 명령을 사용하여 수행할 수 있습니다.

버퍼에서 페이지를 읽기 위해 프로세스는 해당 버퍼[167]의 헤더에 있는 `BufferContent` 잠금을 획득합니다. 이 잠금은 일반적으로 튜플 포인터를 읽는 동안에만 보유되며, 이후에는 버퍼 피닝(`buffer pinning`)에 의해 제공되는 보호가 충분합니다. 버퍼 내용을 수정해야 하는 경우에는 `BufferContent` 잠금을 배타적 모드로 획

---

[165] backend/storage/buffer/bufmgr.c
include/storage/buf_internals.h, BufMappingPartitionLock function

[166] backend/storage/buffer/bufmgr.c, LockBufHdr function

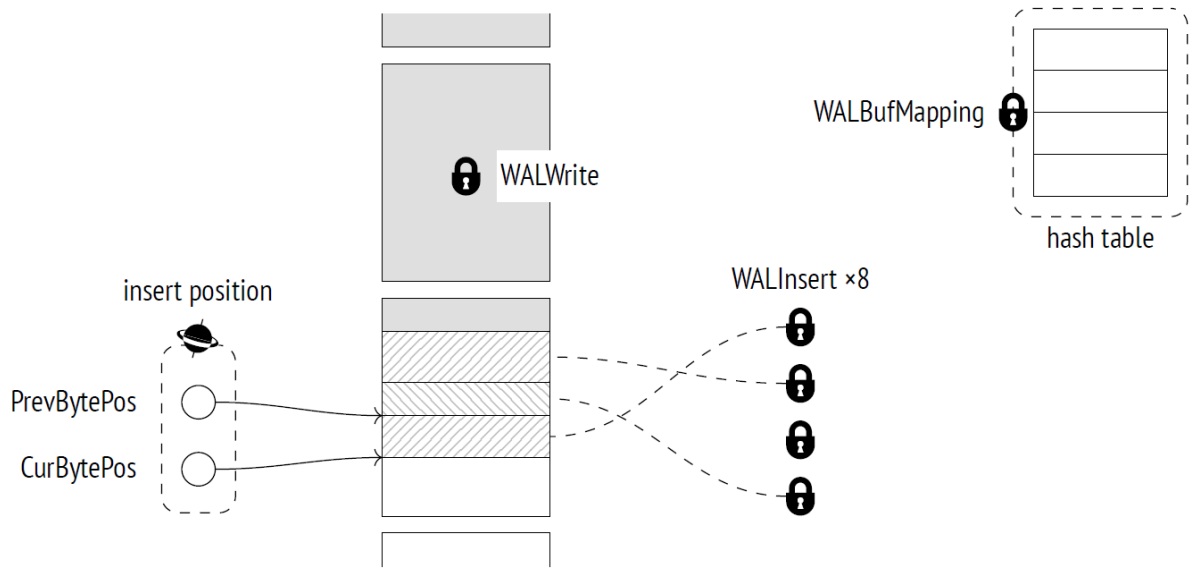
[167] include/storage/buf_internals.h

득해야 합니다.

버퍼가 디스크에서 읽혀지거나 디스크로 기록될 때, PostgreSQL은 버퍼 헤더에 `BufferIO` 잠금을 획득합니다. 이는 실제로는 잠금[168]이 아닌 잠금으로 사용되는 속성입니다. 이는 이 페이지에 대한 액세스를 요청하는 다른 프로세스에게 `I/O` 작업이 완료될 때까지 기다려야 함을 알리는 역할을 합니다.

빈 버퍼의 포인터와 제거 메커니즘의 클락 핸드는 하나의 공통 버퍼 전략strategy 스핀잠금[169]에 의해 보호됩니다.

WAL 버퍼



`WAL` 캐시도 페이지를 버퍼에 매핑하기 위해 해시 테이블을 사용합니다. 버퍼 캐시 해시 테이블과는 달리, `WAL` 캐시는 작은 크기입니다(일반적으로 버퍼 캐시 크기의 1/32). 그리고 버퍼 액세스가 더 순서적입니다. 따라서 `WAL` 캐시의 해시 테이블은 단일 **`WALBufMapping`** 경량 잠금으로 보호됩니다.[170]

`WAL` 페이지를 디스크에 기록하는 작업은 **`WALWrite`** 경량 잠금으로 보호됩니다. 이는 한 번에 한 프로세스에 의해 수행되도록 보장합니다.

`WAL` 항목을 생성하기 위해 프로세스는 먼저 `WAL` 페이지 내에서 일부 공간을 예약한 다음 데이터로 채웁니다. 공간 예약은 엄격하게 순서가 지정되어 있으며, 프로세스는 삽입 포인터를 보호하는 **`insert position`** 스핀잠금을 획득해야 합니다.[171] 그러나 한 번 공간이 예약되면 여러 동시 프로세스가 채울 수 있습니다. 이를 위해 각 프로세스는 **`WALInsert`** 트랜치를 구성하는 여덟 개의 경량 잠금 중 하나를 획득해야 합니다.[172]

---

[168] backend/storage/buffer/bufmgr.c, StartBufferIO function

[169] backend/storage/buffer/freelist.c

[170] backend/access/transam/xlog.c, AdvanceXLInsertBuffer function

[171] backend/access/transam/xlog.c, ReserveXLogInsertLocation function

[172] backend/access/transam/xlog.c, WALInsertLockAcquire function

15.4 대기 모니터링

의심의 여지 없이 잠금은 PostgreSQL의 올바른 동작에 필수적이지만, 원하지 않는 대기 상태로 이어질 수 있습니다. 이러한 대기 상태를 추적하여 원인을 파악하는 것이 유용합니다.

장기적인 잠금의 개요를 얻는 가장 쉬운 방법은 `log_lock_waits`(기본값: `off`) 매개변수를 활성화하는 것입니다. 이 매개변수는 트랜잭션이 `deadlock_timeout`(기본값: 1s)보다 오랜 시간 동안 대기하는 모든 잠금에 대해 상세한 로깅을 가능하게 합니다. 이 데이터는 교착 상태 검사가 완료될 때 표시되므로 매개변수 이름이 지어졌습니다.

그러나 `pg_stat_activity` 뷰는 훨씬 유용하고 완전한 정보를 제공합니다. 시스템 프로세스나 백엔드와 같은 프로세스가 무언가를 기다리기 때문에 작업을 진행할 수 없을 때, 해당 대기는 `wait_event_type` 및 `wait_event` 필드에 반영됩니다. 이는 각각 대기의 유형과 이름을 표시합니다.

모든 대기는 다음과 같이 분류될 수 있습니다.[173]

다양한 잠금에 대한 대기는 상당히 큰 그룹을 형성합니다:

`Lock` - 무거운 잠금
`LWLock` - 가벼운 잠금
`BufferPin` - 고정된 버퍼

그러나 프로세스는 다른 이벤트를 기다릴 수도 있습니다:

`IO` - 데이터를 읽거나 쓰기 위해 필요한 입출력
`Client` - 클라이언트가 보낸 데이터 (`psql`은 대부분 이 상태에 있음)
`IPC` - 다른 프로세스가 보낸 데이터
`Extension` - 확장 기능에 의해 등록된 특정 이벤트

때로는 프로세스가 유용한 작업을 수행하지 않을 수도 있습니다. 이러한 대기는 일반적으로 "정상"이라고 할 수 있으며, 이는 문제를 나타내지 않습니다. 이 그룹에는 다음과 같은 대기가 포함됩니다:

`Activity` - 백그라운드 프로세스의 주 기동
`Timeout` - 타이머

각 대기 유형의 잠금은 대기 이름별로 추가로 분류됩니다. 예를 들어, 가벼운 잠금에 대한 대기는 해당 잠금이나 해당 트랜치의 이름을 가져옵니다.[174]

`pg_stat_activity` 뷰는 소스 코드에서 적절한 방식으로 처리되는 대기만 표시합니다.[175] 이 뷰에 대기의 이름이 나타나지 않는 경우, 프로세스는 알려진 유형의 대기 상태에 있지 않습니다. 이러한 시간은 계산되지 않은 것으로 간주되어야 하며, 프로세스가 아무것도 기다리지 않는다는 것을 의미하는 것은 아닙니다. 우리는

[173] postgresql.org/docs/14/monitoring-stats#WAIT-EVENT-TABLE.html

[174] postgresql.org/docs/14/monitoring-stats#WAIT-EVENT-LWLOCK-TABLE.html

[175] include/utils/wait_event.h

그 순간에 무슨 일이 일어나고 있는지 알지 못합니다.

```
=> SELECT backend_type, wait_event_type AS event_type, wait_event
FROM pg_stat_activity;
                backend_type ¦ event_type ¦ wait_event
-----------------------------+------------+---------------------
 logical replication launcher ¦   Activity ¦ LogicalLauncherMain
          autovacuum launcher ¦   Activity ¦ AutoVacuumMain
               client backend ¦            ¦
            background writer ¦   Activity ¦ BgWriterMain
                 checkpointer ¦   Activity ¦ CheckpointerMain
                    walwriter ¦   Activity ¦ WalWriterMain
(6 rows)
```

여기에서 모든 백그라운드 프로세스는 뷰가 샘플링될 때 유휴 상태이며, **client backend**는 쿼리를 실행하고 어떤 대기 상태에도 있지 않았습니다.

15.5 샘플링

안타깝게도, pg_stat_activity 뷰는 대기 상태에 대한 현재 정보만 표시하며 통계는 누적되지 않습니다. 대기 데이터를 시간에 따라 수집하려면 일정한 간격으로 뷰를 샘플링해야 합니다.

샘플링의 확률적 특성을 고려해야 합니다. 대기 시간이 샘플링 간격보다 짧을수록 해당 대기를 감지할 확률은 낮아집니다. 따라서 샘플링 간격이 길어질수록 실제 상태를 반영하기 위해 더 많은 샘플이 필요합니다(하지만 샘플링 속도를 높이면 부하도 증가합니다). 마찬가지로, 짧은 지속 세션을 분석하는 데는 샘플링이 거의 쓸모가 없습니다.

PostgreSQL은 샘플링을 위한 내장 도구를 제공하지 않지만, pg_wait_sampling[176] 확장을 사용하여 시도해 볼 수 있습니다. 이를 위해 shared_preload_libraries 매개변수에서 해당 라이브러리를 지정하고 서버를 재시작해야 합니다:

```
=> ALTER SYSTEM SET shared_preload_libraries = 'pg_wait_sampling';
```

```
postgres$ pg_ctl restart -l /home/postgres/logfile
```

이제 데이터베이스에 확장을 설치해 보겠습니다:

```
=> CREATE EXTENSION pg_wait_sampling;
```

이 확장은 대기 이력을 저장하는 링 버퍼에 저장된 대기를 표시할 수 있습니다. 그러나 세션 전체 기간 동안 누적된 통계인 대기 프로파일을 얻는 것이 훨씬 흥미로울 것입니다.

예를 들어 벤치마킹 중 대기 상태를 살펴보겠습니다. pgbench 유틸리티를 시작하고 실행 중인 동안 해당 프로세스 ID를 확인해야 합니다:

[176] github.com/postgrespro/pg_wait_sampling

```
postgres$ /usr/local/pgsql/bin/pgbench -T 60 internals
```

```
=> SELECT pid FROM pg_stat_activity
WHERE application_name = 'pgbench';
pid
--------
36367
(1 row)
```

테스트가 완료되면 대기 프로파일은 다음과 같이 보일 것입니다:

```
=> SELECT pid, event_type, event, count
FROM pg_wait_sampling_profile WHERE pid = 36367
ORDER BY count DESC LIMIT 4;
   pid | event_type |        event | count
-------+------------+--------------+-------
 36367 |         IO |      WALSync | 3478
 36367 |         IO |     WALWrite | 52
 36367 |     Client |   ClientRead | 30
 36367 |         IO | DataFileRead | 2
(4 rows)
```

기본적으로 (`pg_wait_sampling.profile_period` 매개변수에 의해 설정됨) 10ms마다 100번 샘플이 수집됩니다. 따라서 대기의 지속 시간을 초 단위로 추정하려면 count 값을 100으로 나눠야 합니다.

이 경우 대부분의 대기는 `WAL` 항목을 디스크에 플러시하는 것과 관련이 있습니다. 이는 계산되지 않은 대기 시간을 잘 보여주는 좋은 예입니다. **`WALSync`** 이벤트는 PostgreSQL12 이전에는 계측되지 않았으며, 낮은 버전에서는 대기 프로필에 첫 번째 행이 포함되지 않을 수 있지만, 대기 자체는 여전히 존재합니다.

또한 다음과 같이 파일 시스템의 `I/O` 작업을 인위적으로 0.1초로 지연시키는 경우 프로필은 다음과 같이 보일 것입니다(`slowfs`[177]를 사용하여 이 작업을 수행합니다):

```
postgres$ /usr/local/pgsql/bin/pgbench -T 60 internals
```

```
=> SELECT pid FROM pg_stat_activity
WHERE application_name = 'pgbench';
pid
-------
36747
(1 row)

=> SELECT pid, event_type, event, count
FROM pg_wait_sampling_profile WHERE pid = 36747
ORDER BY count DESC LIMIT 4;
```

[177] github.com/nirs/slowfs

```
   pid ¦ event_type ¦          event ¦ count
-------+------------+----------------+-------
 36747 ¦         IO ¦       WALWrite ¦ 3603
 36747 ¦     LWLock ¦       WALWrite ¦ 2095
 36747 ¦         IO ¦        WALSync ¦ 22
 36747 ¦         IO ¦ DataFileExtend ¦ 19
(4 rows)
```

이제 `I/O` 작업이 가장 느린 작업입니다. 주로 동기 모드에서 `WAL` 파일을 디스크에 쓰는 작업과 관련이 있습니다. `WALWrite` 경량 잠금으로 보호되므로 해당 행도 프로파일에 나타납니다.

분명히 이전 예제와 동일한 잠금이 여기에서도 획득됩니다. 그러나 대기 시간이 샘플링 간격보다 짧기 때문에 샘플링 횟수가 매우 적거나 프로필에 포함되지 않을 수 있습니다. 이는 짧은 대기 시간을 분석하려면 상당한 시간 동안 샘플링해야 한다는 것을 다시 한 번 보여줍니다.

Part IV

쿼리 실행

16 장 쿼리 실행 단계

16.1 데모 데이터베이스

이 책의 이전 부분의 예제는 소수의 행만 있는 간단한 테이블을 기반으로 했습니다. 이와 이후의 부분은 쿼리 실행을 다루며, 이 측면에서는 더 많은 요구사항이 있습니다. 훨씬 더 많은 행을 가진 관련 테이블이 필요합니다. 각 예제마다 새로운 데이터 세트를 만드는 대신에, 우리는 러시아의 여객 항공 트래픽[178]을 보여주는 기존의 데모 데이터베이스를 사용했습니다. 이 데이터베이스에는 여러 버전이 있으며, 우리는 2017년 8월 15일에 생성된 더 큰 버전을 사용할 것입니다. 이 버전을 설치하려면 데이터베이스 사본[179]을 포함하는 파일을 압축 해제하고 이 파일을 `psql`에서 실행해야 합니다.

이 데모 데이터베이스를 개발할 때, 우리는 스키마가 추가적인 설명 없이도 이해하기 충분하도록 간단하게 만들려고 노력했습니다. 동시에 의미있는 쿼리 작성을 가능하게하기 위해 충분히 복잡하게 만들고자 했습니다. 데이터베이스는 현실적인 데이터로 채워져 있어 예제가 더 포괄적이고 작업하기 흥미로울 것입니다. 여기에서는 주요한 데이터베이스 개체에 대해서만 간단히 다룰 것입니다. 전체 스키마를 검토하려면 각주에서 참조된 전체 설명을 확인하실 수 있습니다.

주요 개체는 **예약**(`booking` 테이블에 매핑됨)입니다. 하나의 예약에는 여러 승객이 포함될 수 있으며, 각각은 별도의 전자 **티켓**을 가지고 있습니다(`tickets`). 승객은 별도의 개체를 구성하지 않습니다. 실험을 위해 모든 승객이 고유하다고 가정합니다.

각 티켓은 하나 이상의 **항공편 세그먼트**를 포함합니다(`ticket_flights` 테이블에 매핑됨). 하나의 티켓은 두 가지 경우에 여러 항공편 세그먼트를 가질 수 있습니다. 왕복 티켓이거나 연결 항공편을 위해 발행된 경우입니다. 스키마에는 해당 제약 조건이 없지만, 예약 내의 모든 티켓은 동일한 항공편 세그먼트를 가지고 있다고 가정합니다.

각 **항공편**(`flights`)은 한 **공항**(`airports`)에서 다른 공항으로 이동합니다. 같은 항공편 번호의 항공편은 출발지와 도착지가 동일하지만 출발 날짜가 다릅니다.

`routes` 뷰는 `flights` 테이블을 기반으로 합니다. 이 뷰는 특정 항공편 날짜에 의존하지 않는 **루트**에 관한 정보를 표시합니다.

체크인 시, 각 승객은 좌석 번호가 있는 **탑승권**(`boarding_passes`)을 발급받습니다. 승객은 해당 항공편이 티켓에 포함되어 있는 경우에만 해당 항공편에 체크인할 수 있습니다. 항공편-좌석 조합은 고유해야 하므로 동일한 좌석에 대해 두 개의 탑승권을 발급하는 것은 불가능합니다.

항공기의 **좌석**(`seats`) 수와 다른 여행 클래스 간의 분포는 해당 항공편을 수행하는 **항공기 모델**(`aircrafts`)에 따라 다릅니다. 각 항공기 모델은 하나의 캐빈 구성만 가질 수 있다고 가정합니다.

일부 테이블은 대리키를 사용하고, 다른 테이블은 자연 키를 사용합니다(일부는 복합 키입니다). 이는 단순히 예제로서의 목적으로 수행되었으며, 따라야 할 예시는 아닙니다.

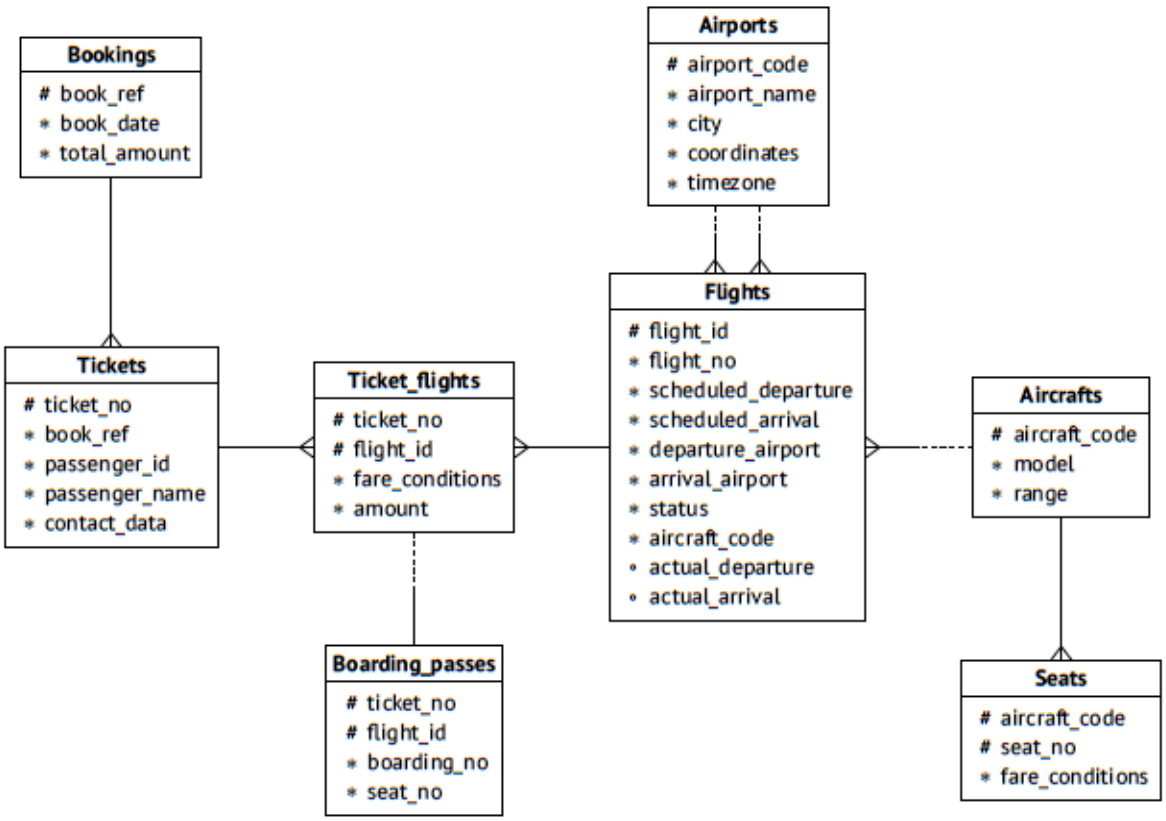
---

[178] postgrespro.com/community/demodb

[179] edu.postgrespro.com/demo-big-en-20170815.zip

데모 데이터베이스는 실제 시스템의 덤프로 생각할 수 있습니다. 과거 특정 시간에 촬영된 데이터 스냅샷을 포함하고 있습니다. 이 시간을 표시하기 위해 `bookings.now()` 함수를 호출할 수 있습니다. 데모 쿼리에서 실제로 `now()` 함수를 요구하는 경우에 이 함수를 사용하십시오.

공항, 도시 및 항공기 모델의 이름은 `airports_data`와 `aircrafts_data` 테이블에 저장되어 있습니다. 이러한 이름은 영어와 러시아어로 제공됩니다. 이 장에 대한 예시를 구성하기 위해 일반적으로 엔티티-관계 다이어그램에 표시된 `airports` 및 `aircrafts` 뷰를 쿼리합니다. 이 뷰는 `bookings.lang`(기본값: en) 매개변수 값을 기반으로 출력 언어를 선택합니다. 그러나 일부 기본 테이블의 이름은 여전히 쿼리 계획에 나타날 수 있습니다.



16.2 간단한 쿼리 프로토콜

클라이언트-서버 프로토콜[180]의 간단한 버전은 `SQL` 쿼리 실행을 가능하게 합니다. 이 프로토콜은 쿼리 텍스트를 서버로 보내고, 어떤 행의 수에 상관없이 완전한 실행 결과를 받습니다.[181] 서버로 보내진 쿼리는 파싱, 변환, 계획 및 실행 등 여러 단계를 거칩니다.

파싱

먼저, PostgreSQL은 실행해야 할 내용을 이해하기 위해 쿼리 텍스트를 파싱[182]해야 합니다.

어휘 및 구문 분석. 렉서는 쿼리 텍스트를 키워드, 문자열 리터럴 및 숫자 리터럴과 같은 렉서[183]로 분할하고, 파서는 이러한 렉서를 `SQL` 언어 문법에 대해 유효성을 검사합니다.[184] PostgreSQL은 주로 `Flex` 및 `Bison` 도구를 사용하여 파싱을 수행합니다.

파싱된 쿼리는 백엔드의 메모리에 추상 구문 트리로 표현됩니다.

예를 들어, 다음과 같은 쿼리를 살펴보겠습니다.

```
SELECT schemaname, tablename
FROM pg_tables
WHERE tableowner = 'postgres'
ORDER BY tablename;
```

렉서는 다섯 개의 키워드, 다섯 개의 식별자, 하나의 문자열 리터럴 및 세 개의 단일 문자 렉서(쉼표, 등호, 세미콜론)를 분리합니다. 파서는 이러한 렉서를 사용하여 파싱 트리를 구축하며, 아래 그림은 매우 단순화된 형태로 파싱 트리를 보여줍니다. 트리 노드 옆의 캡션은 쿼리의 해당 부분을 지정합니다.

---

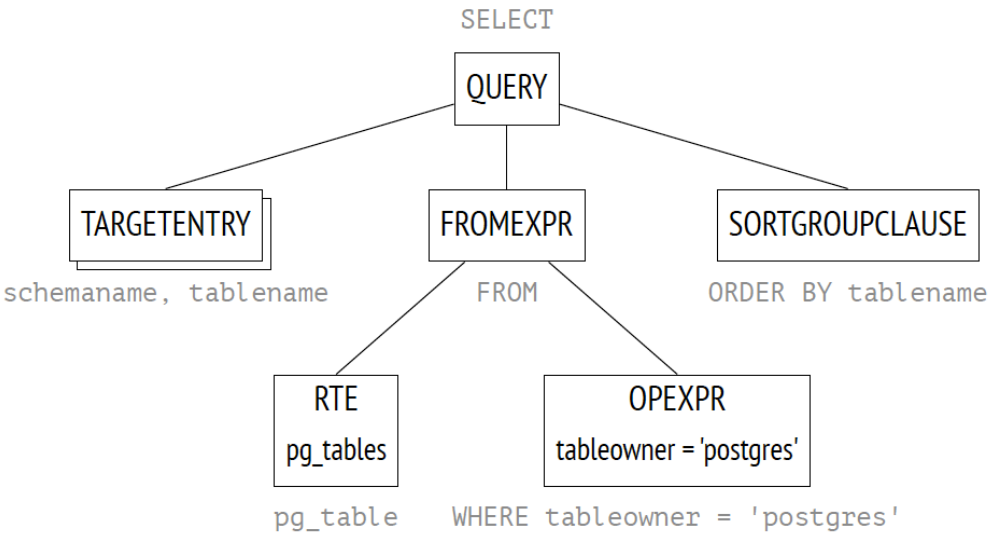
180 postgresql.org/docs/14/protocol.html

181 backend/tcop/postgres.c, exec_simple_query function

182 postgresql.org/docs/14/parser-stage.html
backend/parser/README

183 backend/parser/scan.l

184 backend/parser/gram.y



RTE(범위 테이블 엔트리)라는 상당히 난해한 약어는 PostgreSQL 소스 코드에서 사용됩니다. PostgreSQL은 테이블, 서브쿼리, 조인 결과와 같은 행 집합을 의미하는 범위 테이블이라는 용어를 사용합니다. 다시 말해, `SQL` 연산자로 처리할 수 있는 모든 행 집합을 가리킵니다.[185]

의미 분석. 의미 분석[186]의 목적은 이 쿼리가 이름으로 참조하는 테이블이나 다른 객체가 데이터베이스에 있는지, 사용자가 이러한 객체에 대한 액세스 권한이 있는지를 판단하는 것입니다. 의미 분석에 필요한 모든 정보는 시스템 카탈로그에 저장됩니다.

파싱 트리를 받은 후, 의미 분석기는 추가적인 재구성 작업을 수행합니다. 이 작업에는 특정 데이터베이스 객체, 데이터 타입 및 기타 정보에 대한 참조를 추가하는 것이 포함됩니다.

`debug_print_parse` 매개변수를 활성화하면 서버 로그에서 전체 파싱 트리를 볼 수 있지만, 실제로는 큰 의미가 없습니다.

변환

다음 단계에서는 쿼리를 변환(재작성)할 수 있습니다.[187]

PostgreSQL 코어는 여러 목적으로 변환을 사용합니다. 그 중 하나는 뷰의 이름을 파싱 트리에서 해당 뷰의 기본 쿼리에 해당하는 서브트리로 대체하는 것입니다.

변환을 사용하는 또 다른 경우는 행 수준 보안 구현입니다.[188]

재귀 쿼리의 `SEARCH` 및 `CYCLE` 절도 이 단계에서 변환됩니다.[189]

---

185 include/nodes/parsenodes.h

186 backend/parser/analyze.c

187 postgresql.org/docs/14/rule-system.html

188 backend/rewrite/rowsecurity.c

189 backend/rewrite/rewriteSearchCycle.c

위의 예시에서 `pg_tables`은 뷰입니다. 이 뷰의 정의를 쿼리 텍스트에 넣으면 다음과 같이 보일 것입니다.

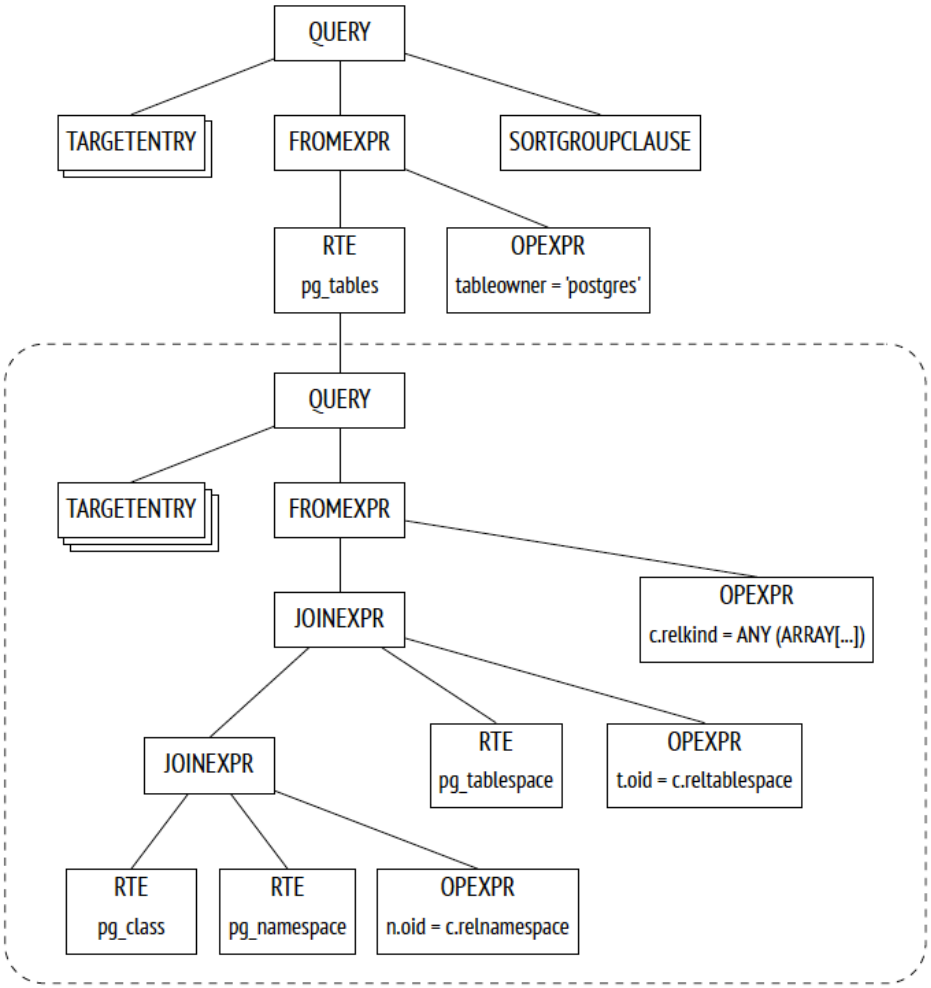
```
SELECT schemaname, tablename
FROM (
  -- pg_tables
  SELECT n.nspname AS schemaname,
      c.relname AS tablename,
      pg_get_userbyid(c.relowner) AS tableowner,
      ...
  FROM pg_class c
      LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
      LEFT JOIN pg_tablespace t ON t.oid = c.reltablespace
  WHERE c.relkind = ANY (ARRAY['r'::char, 'p'::char])
)
WHERE tableowner = 'postgres'
ORDER BY tablename;
```

하지만 서버는 쿼리의 텍스트 표현을 직접 처리하지 않으며, 모든 조작은 파싱 트리에서 수행됩니다. 그림은 변환된 트리의 축소 버전을 보여줍니다(만약 `debug_print_rewritten` 매개변수를 활성화하면 전체 버전을 서버 로그에서 볼 수 있습니다).

파싱 트리는 쿼리의 구문 구조를 반영하지만, 작업을 수행하는서에 대한 정보는 제공하지 않습니다.

PostgreSQL은 사용자가 리라이트 규칙 시스템을 통해 구현할 수 있는 사용자 정의 변환도 지원합니다.[190]

[190] postgresql.org/docs/14/rules.html



포스트그레스 개발의 주요 목표 중 하나로 규칙 시스템 지원이 선언되었습니다.[191] 규칙 시스템은 처음 구현되었을 때 아직 학문적인 프로젝트였지만, 그 이후로 여러 차례 재설계되었습니다. 규칙 시스템은 매우 강력한 메커니즘입니다. 그러나 이를 이해하고 디버깅하는 것은 상당히 어렵습니다. 규칙을 PostgreSQL 에서 완전히 제거하는 제안도 있었지만, 이 아이디어는 일치하는 지지를 받지 못했습니다. 대부분의 경우, 규칙 대신 트리거를 사용하는 것이 안전하고 쉽습니다.

플래닝

SQL은 선언적인 언어입니다. 쿼리는 데이터를 가져올 대상을 지정하지만, 가져오는 방법은 지정하지 않습니다.

어떤 쿼리에는 여러 실행 경로가 있습니다. 파싱 트리에 표시된 각 작업은 여러 가지 방법으로 완료될 수 있습니다. 예를 들어, 결과는 전체 테이블을 읽어서 (중복을 필터링하여) 가져올 수도 있고, 인덱스 스캔을 통해 필요한 행을 찾아올 수도 있습니다. 데이터 세트는 항상 쌍으로 조인되므로 조인 순서가 다른 많은 옵션이 있습니다. 또한 다양한 조인 알고리즘이 있습니다. 예를 들어, 실행자는 첫 번째 데이터 세트의 행을 스캔하고

[191] M. Stonebraker, L. A. Rowe. The Design of Postgres

다른 세트에서 일치하는 행을 검색할 수도 있고, 두 데이터 세트를 먼저 정렬한 다음 병합할 수도 있습니다. 각 알고리즘에는 다른 알고리즘보다 더 나은 성능을 발휘하는 사용 사례를 찾을 수 있습니다.

최적과 비최적 계획의 실행 시간은 수십 배 이상 차이가 날 수 있으므로, 파싱된 쿼리를 최적화하는 플래너[192]는 시스템의 가장 복잡한 구성 요소 중 하나입니다.

플랜 트리. 실행 계획도 트리로 표현되지만, 논리적인 작업이 아닌 데이터에 관한 물리적인 작업을 다룹니다.

전체 계획 트리를 탐색하고 싶다면 `debug_print_plan` 매개변수를 활성화하여 서버 로그에 덤프할 수 있습니다. 그러나 실제로는 `EXPLAIN` 명령으로 표시되는 계획의 텍스트 표현을 보는 것으로 충분합니다.[193]

다음 그림은 트리의 주요 노드를 강조합니다. 이 노드들이 아래에 제공된 `EXPLAIN` 명령의 출력에서 보여지는 노드입니다.

지금은 다음 두 가지에 주목해 봅시다.

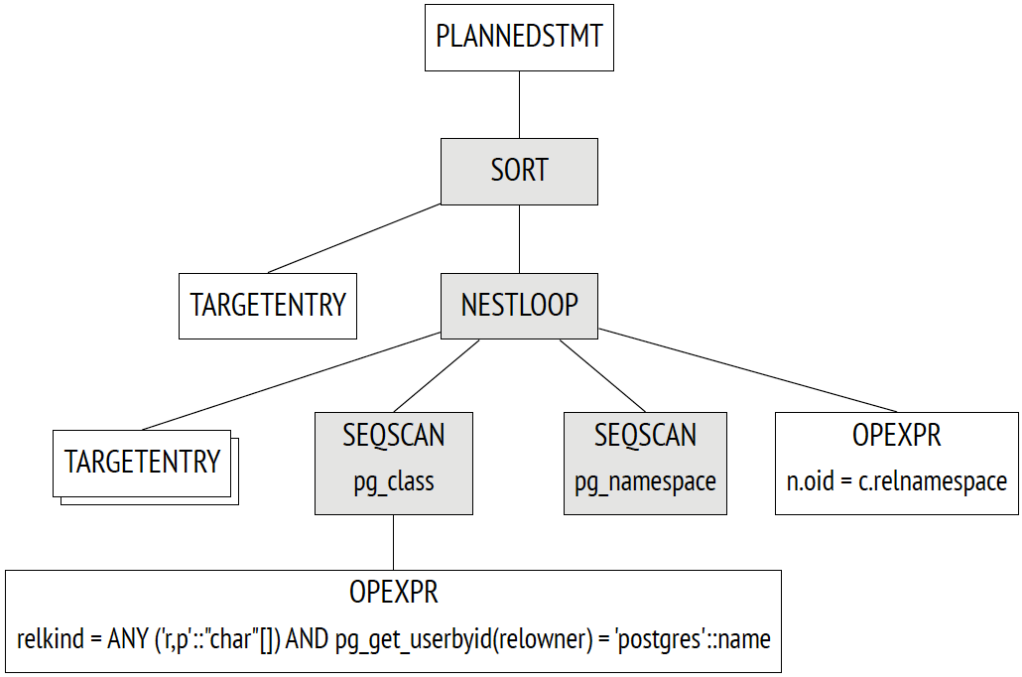
- 트리에는 세 개의 테이블 중 두 개의 테이블만 있습니다. 플래너는 결과를 검색하는 데 필요하지 않은 테이블을 계획 트리에서 제거했습니다.
- 트리의 각 노드에 대해 플래너는 예상 비용과 처리할 예상 행 수를 제공합니다.

```
=> EXPLAIN SELECT schemaname, tablename
FROM pg_tables
WHERE tableowner = 'postgres'
ORDER BY tablename;
QUERY PLAN
------------------------------------------------------------------------
Sort (cost=21.03..21.04 rows=1 width=128)
  Sort Key: c.relname
      -> Nested Loop Left Join (cost=0.00..21.02 rows=1 width=128)
             Join Filter: (n.oid = c.relnamespace)
             -> Seq Scan on pg_class c (cost=0.00..19.93 rows=1 width=72)
                    Filter: ((relkind = ANY ('{r,p}'::"char"[])) AND (pg_g...
             -> Seq Scan on pg_namespace n (cost=0.00..1.04 rows=4 wid...
(7 rows)
```

쿼리 계획에 표시된 `Seq Scan` 노드는 테이블을 읽는 것을 나타내고, `Nested Loop` 노드는 조인 작업을 나타냅니다.

192 postgresql.org/docs/14/planner-optimizer.html

193 postgresql.org/docs/14/using-explain.html



플랜 탐색. PostgreSQL은 비용 기반 옵티마이저[194]를 사용합니다. 옵티마이저는 잠재적인 계획을 검토하고 실행에 필요한 리소스(예: `I/O` 작업 또는 `CPU` 사이클)를 추정합니다. 숫자 값으로 정규화된 이 추정은 계획의 비용이라고 합니다. 고려된 모든 계획 중에서 비용이 가장 낮은 계획이 선택됩니다.

문제는 조인된 테이블의 수에 따라 잠재적으로 가능한 계획의 수가 기하급수적으로 증가하기 때문에, 비교적 간단한 쿼리에도 모든 계획을 고려하는 것은 불가능합니다. 탐색은 일반적으로 동적 프로그래밍 알고리즘과 일부 휴리스틱을 결합하여 좁혀집니다. 이를 통해 플래너는 테이블 수가 많은 쿼리에 대해 수학적으로 정확한 솔루션을 적당한 시간 내에 찾을 수 있습니다.

> 정확한 솔루션은 선택된 계획이 실제로 최적인 것을 보장하지 않습니다. 플래너는 단순화된 수학적 모델을 사용하고 신뢰할만한 입력 데이터가 부족할 수 있기 때문입니다.

조인 순서 관리. 쿼리는 최적화를 위해 어느 정도로 검색 범위를 제한할 수 있습니다(최적 계획을 놓칠 위험도 있음).

- 공통 테이블 표현식과 주 쿼리는 별도로 최적화될 수 있습니다. 이러한 동작을 보장하려면 `MATERIALIZED` 절을 지정할 수 있습니다.[195]
- `SQL` 함수 내에서 실행되는 서브쿼리는 항상 별도로 최적화됩니다. (`SQL` 함수는 때로는 주 쿼리에 인라인될 수 있습니다.[196])
- 쿼리에서 명시적인 `JOIN` 절을 사용하고 `join_collapse_limit` 매개변수를 설정하면 일부 조인의 순서는 쿼리 구문 구조에 의해 정의됩니다. `from_collapse_limit` 매개변수는 서브쿼리에 동일한 효과를 줍니다.[197]

---

[194] backend/optimizer/README

[195] postgresql.org/docs/14/queries-with.html

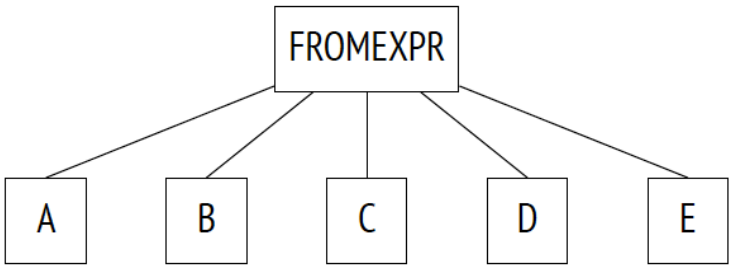
[196] wiki.postgresql.org/wiki/Inlining_of_SQL_functions

[197] postgresql.org/docs/14/explicit-joins.html

마지막으로 언급한 부분을 설명해 보겠습니다. 다음과 같이 `FROM` 절에 명시적인 조인을 지정하지 않는 쿼리를 살펴보겠습니다.

```
SELECT ...
FROM a, b, c, d, e
WHERE ...
```

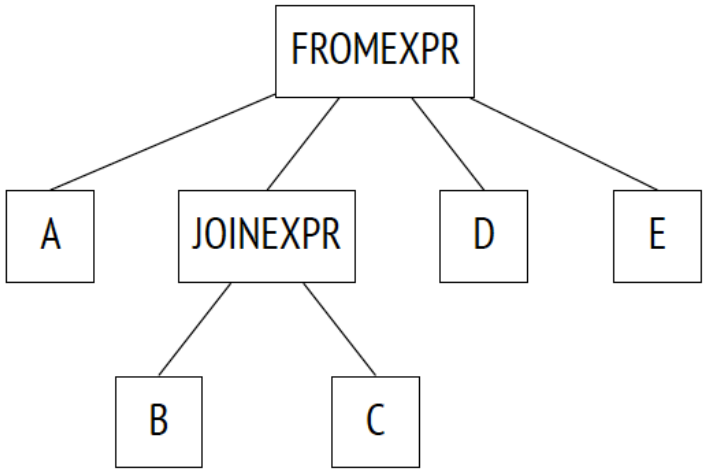
여기서 플래너는 모든 가능한 조인 쌍을 고려해야 합니다. 쿼리는 다음과 같이 파싱 트리의 일부로 표시됩니다(그림으로 간략히 표시됨).



다음 예제에서는 조인이 `JOIN` 절에 의해 정의된 특정 구조를 갖습니다:

```
SELECT ...
FROM a, b JOIN c ON ..., d, e
WHERE ...
```

파싱 트리는 이 구조를 반영합니다.



플래너는 일반적으로 조인 트리를 평탄화하여 첫 번째 예시와 같은 모양으로 만듭니다. 이 알고리즘은 트리를 재귀적으로 탐색하고, 각 `JOINEXPR` 노드를 해당 요소의 평탄한 목록으로 대체합니다.[198]

[198] backend/optimizer/plan/initsplan.c, deconstruct_jointree function

그러나 이러한 축소는 결과적으로 생성된 평탄한 목록이 `join_collapse_limit`(기본값: 8) 요소 이하인 경우에만 수행됩니다. 이 특정 경우에서는 `join_collapse_limit` 값이 5보다 작다면 `JOINEXPR` 노드가 축소되지 않을 것입니다.

플래너에게 다음을 의미합니다:

- 테이블 B는 테이블 C와 조인되어야 합니다(또는 그 반대로, C가 B와 조인되어야 합니다; 쌍 내에서 조인 순서는 제한되지 않습니다).
- 테이블 A, D, E 및 B와 C를 조인한 결과는 어떤 순서로든 조인할 수 있습니다.

`join_collapse_limit` 매개변수를 1로 설정하면 명시적인 `JOIN` 절에 의해 정의된 순서가 유지됩니다.

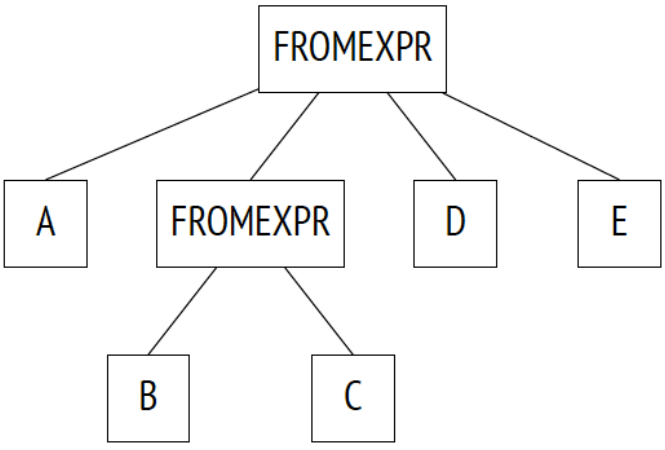
`FULL OUTER JOIN` 연산자의 경우, `join_collapse_limit` 매개변수의 값에 관계없이 축소되지 않습니다.

`from_collapse_limit`(기본값: 8) 매개변수는 서브쿼리 평탄화를 유사한 방식으로 제어합니다. 서브쿼리는 `JOIN` 절과 같지 않지만, 파싱 트리 수준에서 유사성이 드러납니다.

다음은 샘플 쿼리입니다:

```
SELECT ...
FROM a,
  (
      SELECT ... FROM b, c WHERE ...
  ) bc,
  d, e
WHERE ...
```

해당하는 조인 트리는 아래에 표시됩니다. 여기서 유일한 차이점은 이 트리가 `JOINEXPR` 대신 `FROMEXPR` 노드를 포함한다는 것입니다(따라서 매개변수 이름이 이와 같습니다).



유전적 쿼리 최적화. 평탄화된 트리는 한 레벨에 너무 많은 요소(테이블 또는 조인 결과)를 포함할 수 있으며, 이들은 별도로 최적화되어야 합니다. 계획 시간은 조인해야 할 데이터 세트의 수에 지수적으로 의존하기 때문에 합리적인 한계를 초과할 수 있습니다.

`geqo`(기본값: on) 매개변수가 활성화되어 있고, 한 레벨의 요소 수가 `geqo_threshold`(기본값: 12) 값보다 크면 플래너는 유전 알고리즘을 사용하여 쿼리를 최적화합니다.[199] 이 알고리즘은 동적 프로그래밍 대비 훨씬 빠르지만, 찾은 계획이 최적인지 보장할 수는 없습니다. 따라서 일반적으로 최적화할 요소의 수를 줄여서 유전 알고리즘을 사용하지 않도록 하는 것이 좋습니다.

유전 알고리즘에는 여러 가지 구성 가능한 매개변수[200]가 있지만, 여기에서는 다루지 않겠습니다.

최적의 계획 선택. 계획이 최적인지 여부는 특정 클라이언트가 쿼리 결과를 어떻게 사용할지에 따라 달라집니다. 클라이언트가 전체 결과를 한 번에 필요로 하는 경우(예: 보고서 생성), 계획은 모든 행을 검색하는 최적화를 해야 합니다. 그러나 첫 번째 행을 가능한 빨리 반환하는 것이 우선순위인 경우(예: 화면에 표시), 최적의 계획은 완전히 다를 수 있습니다.

이 선택을 위해 포스트그레스튜엘은 비용의 두 구성 요소를 계산합니다:

```
=> EXPLAIN
SELECT schemaname, tablename
FROM pg_tables
WHERE tableowner = 'postgres'
ORDER BY tablename;
                      QUERY PLAN
------------------------------------------------------------------------
Sort (cost=21.03..21.04 rows=1 width=128)
  Sort Key: c.relname
      -> Nested Loop Left Join (cost=0.00..21.02 rows=1 width=128)
              Join Filter: (n.oid = c.relnamespace)
              -> Seq Scan on pg_class c (cost=0.00..19.93 rows=1 width=72)
                      Filter: ((relkind = ANY ('{r,p}'::"char"[])) AND (pg_g...
              -> Seq Scan on pg_namespace n (cost=0.00..1.04 rows=4 wid...
(7 rows)
```

첫 번째 구성 요소(시작 비용)는 노드 실행을 준비하기 위해 지불하는 비용을 나타내며, 두 번째 구성 요소(총 비용)는 결과를 가져오는 데 소요되는 모든 비용을 포함합니다.

> 가끔 시작 비용은 결과 집합의 첫 번째 행을 검색하는 비용이라고 언급되지만, 이는 완전히 정확하지 않습니다.

선호하는 계획을 구분하기 위해 옵티마이저는 쿼리가 커서를 사용하는지(예: `SQL`에서 제공하는 `DECLARE` 명령어 또는 `PL/pgSQL`에서 명시적으로 선언) 확인합니다.[201] 커서를 사용하지 않는 경우, 클라이언트가 한 번에 전체 결과를 필요로 한다고 가정하고, 옵티마이저는 총 비용이 가장 적은 계획을 선택합니다.

---

199 postgresql.org/docs/14/geqo.html
backend/optimizer/geqo/geqo_main.c

200 postgresql.org/docs/14/runtime-config-query#RUNTIME-CONFIG-QUERY-GEQO.html

201 backend/optimizer/plan/planner.c, standard_planner function

커서를 사용하여 쿼리를 실행하는 경우, 선택된 계획은 `cursor_tuple_fraction`(기본값: 0.1)의 비율로만 결과를 검색하는 최적화를 수행해야 합니다. 좀 더 정확히 말하면, PostgreSQL은 다음 식의 가장 작은 값을 가지는 계획을 선택합니다:[202]

$$startup\ cost + cursor\_tuple\_fraction\ (total\ cost - startup\ cost)$$

비용 추정 개요. 계획의 총 비용을 추정하려면 모든 노드의 비용 추정 값을 가져와야 합니다. 노드의 비용은 해당 유형에 따라 다르며 (힙 데이터 읽기 비용이 정렬 비용과 동일하지 않음을 명백히 알 수 있음) 이 노드에서 처리하는 데이터 양에 따라 비용이 달라집니다 (데이터 양이 크면 일반적으로 더 높은 비용 발생). 노드 유형은 알려져 있지만, 데이터 양은 입력 세트의 예상 카디널리티(노드가 입력으로 사용하는 행 수) 및 노드의 선택도(출력에 남아 있는 행의 비율)를 기반으로 예측될 수 있습니다. 이러한 계산은 테이블 크기와 테이블 열의 데이터 분포와 같은 수집된 통계에 의존합니다.

따라서 수행된 최적화는 자동 진공에 의해 수집과 수정되는 통계 데이터의 정확성에 따라 달라집니다.

각 노드의 카디널리티 추정이 정확한 경우, 계산된 비용은 실제 비용을 잘 반영할 것입니다. 주요한 계획 결함은 일반적으로 카디널리티와 선택도의 부정확한 추정으로 인한 것으로, 이는 부정확하거나 오래된 통계 데이터, 통계 데이터 사용의 불가능성, 또는 상대적으로 불완전한 계획 모델에 의해 발생할 수 있습니다.

카디널리티 추정. 노드의 카디널리티를 계산하기 위해 플래너는 다음 단계를 재귀적으로 완료해야 합니다:

1. 각 하위 노드의 카디널리티를 추정하고, 노드가 해당 하위 노드로부터 받을 입력 행 수를 평가합니다.
2. 노드의 선택도를 추정합니다. 즉, 입력 행 중 출력에 남을 행의 비율을 의미합니다.
3.

노드의 카디널리티는 이 두 값의 곱으로 표현됩니다.

> 선택도는 0 부터 1 까지의 숫자로 나타납니다. 숫자가 작을수록 선택도가 높으며, 그 반대로 1 에 가까운 숫자는 선택도가 낮음을 나타냅니다. 이는 역설적으로 보일 수 있지만, 아이디어는 매우 선택적인 조건은 거의 모든 행을 제거하고, 몇 개의 행만 제거하는 조건은 선택도가 낮다는 것입니다.

먼저, 플래너는 데이터 액세스 방법을 정의하는 리프 노드의 카디널리티를 추정합니다. 이러한 계산은 테이블의 총 크기와 같은 수집된 통계에 의존합니다.

필터 조건의 선택도는 조건의 유형에 따라 달라집니다. 가장 단순한 경우에는 상수 값을 가정할 수 있지만, 플래너는 추정을 더 정교하게 하기 위해 가능한 모든 정보를 사용하려고 합니다. 일반적으로 간단한 필터 조건의 추정에 대한 정보만 알면 충분합니다. 조건에 논리 연산이 포함된 경우에는 다음 공식을 사용하여 선택도를 계산합니다:[203]

$$sel_{x\ and\ y} = sel_x\ sel_y$$
$$sel_{x\ or\ y} = 1 - (1 - sel_x)(1 - sel_y) = sel_x + sel_y - sel_x\ sel_y$$

---

[202] backend/optimizer/util/pathnode.c, compare_fractional_path_costs function

[203] backend/optimizer/path/clausesel.c, clauselist_selectivity_ext & clauselist_selectivity_or functions

하지만 이러한 공식은 조건 x와 y가 서로 독립적이라고 가정합니다. 상관된predicate 조건의 경우, 이러한 추정은 정확하지 않을 수 있습니다.

조인의 카디널리티를 추정하기 위해서는 카테시안 곱(두 데이터 세트의 카디널리티의 곱)의 카디널리티를 계산하고, 조인 조건의 선택도를 추정해야 합니다. 이는 다시 조건의 유형에 따라 달라집니다.

정렬이나 집계와 같은 다른 노드의 카디널리티도 유사한 방식으로 추정됩니다.

중요한 점은 낮은 계획 노드에 대한 잘못된 카디널리티 추정이 모든 후속 계산에 영향을 미치며, 부정확한 총 비용 추정과 나쁜 계획 선택으로 이어질 수 있다는 것입니다. 더 심각한 문제는 플래너가 조인 결과에 관한 통계를 가지고 있지 않으며, 테이블에 대한 통계만 가지고 있다는 것입니다.

비용 추정. 비용을 추정하는 과정도 재귀적입니다. 서브트리의 비용을 계산하기 위해서는 모든 자식 노드의 비용을 계산하고 합산한 다음, 부모 노드 자체의 비용을 추가해야 합니다.

PostgreSQL은 노드가 수행하는 작업의 수학적 모델을 적용하여 노드의 비용을 추정합니다. 이미 추정된 노드의 카디널리티를 입력으로 사용합니다. 각 노드에 대해 시작 비용과 총 비용이 모두 계산됩니다.

일부 작업은 사전 조건이 없으므로 즉시 실행이 시작됩니다. 이러한 노드는 시작 비용이 0입니다.

반대로 다른 작업은 일부 준비 작업이 완료될 때까지 기다려야 합니다. 예를 들어, 정렬 노드는 일반적으로 자식 노드로부터 모든 데이터가 도착할 때까지 기다려야 자체 작업을 진행할 수 있습니다. 이러한 노드의 시작 비용은 일반적으로 0보다 높습니다. 즉, 위의 노드(또는 클라이언트)가 전체 출력 중 하나의 행만 필요한 경우에도 이 비용을 지불해야 합니다.

플래너가 수행하는 모든 계산은 단순히 추정치입니다. 이러한 추정치는 실제 실행 시간과는 아무런 관련이 없을 수 있습니다. 그들의 목적은 동일한 조건에서 동일한 쿼리에 대한 다른 계획을 비교할 수 있도록 하는 것입니다. 다른 쿼리(특히 서로 다른 쿼리)의 비용을 비교하는 것은 의미가 없습니다. 예를 들어, 통계가 오래되어 비용이 과소 추정될 수 있습니다. 통계가 갱신되면 계산된 숫자가 증가할 수 있지만, 추정이 더 정확해지므로 서버는 더 나은 계획을 선택할 것입니다.

실행

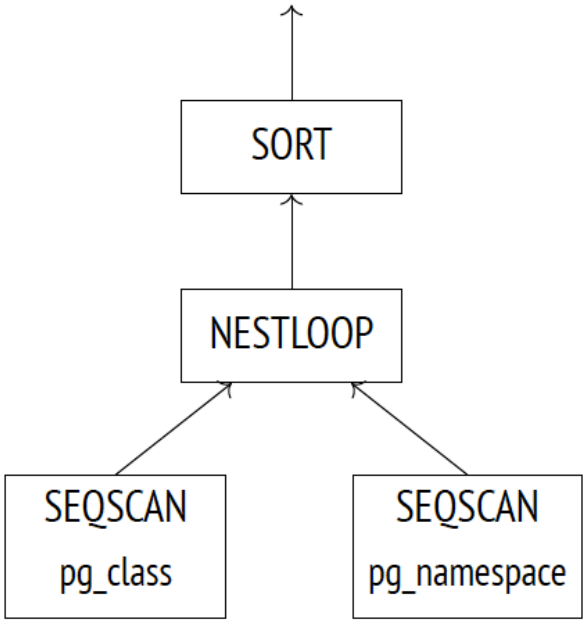
쿼리 최적화 중에 구성된 계획은 이제 실행되어야 합니다.[204]

실행자는 백엔드의 메모리[205]에 포털을 엽니다. 포털은 현재 실행 중인 쿼리의 상태를 유지하는 객체입니다. 이 상태는 트리로 표현됩니다.

---

[204] postgresql.org/docs/14/executor.html
backend/executor/README

[205] backend/utils/mmgr/portalmem.c



쿼리 실행은 계획 트리의 구조를 따르는 트리로 표현된 상태에서 진행됩니다. 이 트리의 노드는 서로 행을 요청하고 보내는 방식으로 동작합니다.

쿼리 실행은 루트에서 시작됩니다. 루트 노드(이 예시에서는 `SORT` 작업을 나타냄)는 자식 노드로부터 데이터를 가져옵니다. 모든 행을 받은 후에는 행을 정렬하고 클라이언트로 전달합니다.

일부 노드(예: 이 그림에서 보여지는 `NESTLOOP` 노드)는 서로 다른 소스로부터 받은 데이터 세트를 결합합니다. 이러한 노드는 두 개의 자식 노드로부터 데이터를 가져와 조인 조건을 만족하는 행 쌍을 받은 후에 결과 행을 즉시 상위로 전달합니다(정렬과 달리 먼저 모든 행을 가져와야 하는 것은 아닙니다).

이 시점에서 노드의 실행은 부모가 다음 행을 요청할 때까지 중단됩니다. 부분 결과만 필요한 경우(예: 쿼리에 `LIMIT` 절이 있는 경우), 작업은 완전히 수행되지 않을 수 있습니다.

트리의 두 개의 `SEQSCAN` 리프 노드는 테이블 스캔을 담당합니다. 부모 노드가 이러한 노드로부터 데이터를 요청하면, 해당 테이블에서 다음 행을 가져옵니다.

따라서 일부 노드는 행을 저장하지 않고 즉시 상위로 전달하지만, 다른 노드(예: `SORT`)는 잠재적으로 큰 데이터 양을 유지해야 합니다. 이를 위해 백엔드의 메모리에 `work_mem`(기본값: 4MB) 청크가 할당됩니다. 이 청크가 충분하지 않은 경우, 나머지 데이터는 디스크의 임시 파일로 쓰여집니다.[206]

계획에는 데이터 저장소가 필요한 여러 노드가 있을 수 있으므로 PostgreSQL은 `work_mem` 크기의 여러 메모리 청크를 할당할 수 있습니다. 쿼리가 사용할 수 있는 `RAM`의 총 크기에는 제한이 없습니다.

16.3 확장 쿼리 프로토콜

단순한 쿼리 프로토콜을 사용할 때는, 모든 명령어(여러 번 반복되는 경우도 포함하여)가 다음과 같은 단계를

[206] backend/utils/sort/tuplestore.c

거쳐야 합니다:

1. 파싱
2. 변환
3. 계획 수립
4. 실행

하지만, 동일한 쿼리를 반복해서 파싱하는 것은 의미가 없습니다. 상수만 다른 쿼리를 반복해서 파싱하는 것도 큰 의미가 없습니다. 파싱 트리 구조는 여전히 동일하기 때문입니다.

단순한 쿼리 프로토콜의 또 다른 단점은 클라이언트가 결과를 한 번에 모두 받는다는 것입니다. 행의 수와 관계없이 전체 결과를 한 번에 받게 됩니다.

일반적으로 `SQL` 명령어를 사용하여 이러한 제한을 극복할 수 있습니다. 첫 번째 제한을 해결하기 위해 실행하기 전에 쿼리를 준비할 수 있으며(`EXECUTE` 명령어 실행 전에), 두 번째 관련 문제는 `DECLARE`로 커서를 생성하고 `FETCH`를 통해 행을 반환함으로써 해결할 수 있습니다. 그러나 이 경우에는 새로 생성된 객체의 이름을 클라이언트가 처리해야 하며, 서버는 추가적으로 추가 명령어를 파싱하는 부하가 발생합니다.

확장된 클라이언트-서버 프로토콜은 명령어 수준에서 개별 연산자 실행 단계를 정밀하게 제어할 수 있는 대안적인 솔루션을 제공합니다.

준비

준비 단계에서는 쿼리가 일반적으로 파싱되고 변환되지만, 결과로 나온 파싱 트리는 백엔드의 메모리에 유지됩니다.

포스트에스큐엘은 쿼리에 대한 전역 캐시를 갖고 있지 않습니다. 이 구조의 단점은 명백합니다: 동일한 쿼리가 이미 다른 백엔드에서 파싱되었음에도 불구하고, 각 백엔드는 모든 들어오는 쿼리를 파싱해야 합니다. 하지만 이에도 일부 이점이 있습니다. 전역 캐시는 잠금 때문에 병목 현상이 발생할 수 있습니다. 상수만 다른 작은 쿼리를 여러 개 실행하는 클라이언트는 많은 트래픽을 생성하며 전체 인스턴스의 성능에 부정적인 영향을 줄 수 있습니다.
PostgreSQL에서는 쿼리가 로컬에서 파싱되므로 다른 프로세스에 영향을 미치지 않습니다.

준비된 쿼리는 매개변수화될 수 있습니다. 다음은 `SQL` 명령어를 사용한 간단한 예제입니다. (프로토콜 수준의 준비와는 다르지만, 최종적인 효과는 동일합니다):

```
=> PREPARE plane(text) AS
SELECT * FROM aircrafts WHERE aircraft_code = $1;
```

모든 명명된 준비된 문은 `pg_prepared_statements` 뷰에서 확인할 수 있습니다:

```
=> SELECT name, statement, parameter_types
FROM pg_prepared_statements \gx

[RECORD 1]
     name       ¦ plane
statement       ¦ PREPARE plane(text) AS +
```

```
                ¦ SELECT * FROM aircrafts WHERE aircraft_code = $1;
parameter_types ¦ {text}
```

여기에는 이름이 지정되지 않은 문(확장된 쿼리 프로토콜이나 PL/pgSQL을 사용하는 문)은 없습니다. 다른 백엔드에서 준비된 문도 표시되지 않습니다. 다른 세션의 메모리에 액세스할 수 없기 때문입니다.

매개변수 바인딩

준비된 문이 실행되기 전에 실제 매개변수 값을 바인딩해야 합니다.

```
=> EXECUTE plane('733');
 aircraft_code ¦          model ¦ range
---------------+----------------+-------
           733 ¦ Boeing 737-300 ¦ 4200
(1 row)
```

준비된 문에서 매개변수를 바인딩하는 것은 쿼리 문자열과 리터럴을 연결하는 것보다 SQL 인젝션을 완전히 불가능하게 만듭니다. 바인딩된 매개변수 값은 이미 구축된 파싱 트리를 어떤 방식으로든 수정할 수 없습니다. 준비된 문 없이 동일한 보안 수준을 달성하려면 신뢰할 수 없는 소스에서 받은 각 값을 주의해서 이스케이프 처리해야 합니다.

계획 수립과 실행

준비된 문 실행에는 실제 매개변수 값을 기반으로 쿼리 계획이 수립되고, 그 계획이 실행자에게 전달됩니다. 다른 매개변수 값은 서로 다른 최적의 계획을 의미할 수 있으므로 정확한 값을 고려하는 것이 중요합니다. 예를 들어, 비싼 예약을 찾을 때, 계획 수립자는 일치하는 행이 그리 많지 않을 것으로 가정하고 인덱스 스캔을 사용합니다.

```
=> CREATE INDEX ON bookings(total_amount);
=> EXPLAIN SELECT * FROM bookings
WHERE total_amount > 1000000;
QUERY PLAN
------------------------------------------------------------------------
Bitmap Heap Scan on bookings (cost=86.49..9245.82 rows=4395 wid...
  Recheck Cond: (total_amount > '1000000'::numeric)
      -> Bitmap Index Scan on bookings_total_amount_idx (cost=0.00....
  Index Cond: (total_amount > '1000000'::numeric)
(4 rows)
```

하지만 모든 예약에서 조건을 만족하는 경우, 전체 테이블을 스캔해야 하므로 인덱스를 사용하는 것은 의미가 없습니다.

```
=> EXPLAIN SELECT * FROM bookings WHERE total_amount > 100;
QUERY PLAN
------------------------------------------------------------------------
Seq Scan on bookings (cost=0.00..39835.88 rows=2111110 width=21)
```

```
  Filter: (total_amount > '100'::numeric)
(2 rows)
```

일부 경우에는 계획 수립자가 반복된 계획을 피하기 위해 파싱 트리와 쿼리 계획을 모두 유지할 수 있습니다. 이러한 계획은 매개변수 값에 대한 고려를 하지 않으므로 실제 값에 기반한 사용자 정의 계획과 구별하여 일반 계획이라고 합니다.[207]

서버가 성능을 저하시키지 않고 일반 계획을 사용할 수 있는 명확한 경우는 매개변수가 없는 쿼리입니다. 매개변수화된 준비된 문의 처음 다섯 가지 최적화는 항상 실제 매개변수 값에 의존합니다. 계획 수립자는 이러한 값에 기반한 사용자 정의 계획의 평균 비용을 계산합니다. 여섯 번째 실행부터는 일반 계획이 사용자 정의 계획보다 평균적으로 더 효율적일 경우 (사용자 정의 계획은 매번 다시 만들어야 함을 고려하여)[208], 계획 수립자는 일반 계획을 유지하고 최적화 단계를 건너뛰며 계속 사용합니다.

plane 준비된 문은 이미 한 번 실행되었습니다. 다음 세 번의 실행 후에도 서버는 여전히 사용자 정의 계획을 사용합니다. 이는 쿼리 계획의 매개변수 값에 의해 확인할 수 있습니다.

```
=> EXECUTE plane('763');
=> EXECUTE plane('773');
=> EXPLAIN EXECUTE plane('319');
QUERY PLAN
------------------------------------------------------------------
Seq Scan on aircrafts_data ml (cost=0.00..1.39 rows=1 width=52)
  Filter: ((aircraft_code)::text = '319'::text)
(2 rows)
```

쿼리 계획에서 매개변수 값에 대한 참조가 나타납니다.

다섯 번째 실행 이후에 계획 수립자는 일반 계획으로 전환합니다. 이 계획은 사용자 정의 계획과 동일하며 비용도 동일하지만, 백엔드는 한 번 빌드하고 최적화 단계를 건너뛸 수 있으므로 계획 수립 부하를 줄일 수 있습니다. `EXPLAIN` 명령은 이제 매개변수가 값이 아닌 위치에 의해 참조된다는 것을 보여줍니다.

```
=> EXECUTE plane('320');
=> EXPLAIN EXECUTE plane('321');
QUERY PLAN
------------------------------------------------------------------
Seq Scan on aircrafts_data ml (cost=0.00..1.39 rows=1 width=52)
  Filter: ((aircraft_code)::text = $1)
(2 rows)
```

우리는 처음 몇 가지 사용자 정의 계획이 일반 계획보다 비용이 더 비싸지만, 그 이후의 계획이 더 효율적일 수 있는 불행한 상황을 쉽게 상상할 수 있습니다. 그러나 계획 수립자는 이러한 계획을 고려하지 않을 것입니다. 게다가, 계산 시 실제 비용이 아닌 추정치를 비교하므로 잘못된 계산으로 이어질 수도 있습니다.

---

[207] backend/utils/cache/plancache.c, choose_custom_plan function

[208] backend/utils/cache/plancache.c, cached_plan_cost function

그러나 계획 수립자가 실수한 경우, 자동 결정을 무시하고 `plan_cache_mode`(기본값: `auto`) 매개변수를 `force_custom_plan`으로 설정하여 일반 계획이나 사용자 정의 계획 중 하나를 선택할 수 있습니다.

```
=> SET plan_cache_mode = 'force_custom_plan';
=> EXPLAIN EXECUTE plane('CN1');
QUERY PLAN
------------------------------------------------------------------
Seq Scan on aircrafts_data ml (cost=0.00..1.39 rows=1 width=52)
  Filter: ((aircraft_code)::text = 'CN1'::text)
(2 rows)
```

`pg_prepared_statements` 뷰는 선택된 계획에 대한 통계도 보여줍니다.

```
=> SELECT name, generic_plans, custom_plans
FROM pg_prepared_statements;
  name | generic_plans | custom_plans
-------+---------------+--------------
 plane |             1 | 6
(1 row)
```

확장된 쿼리 프로토콜을 사용하면 한 번에 모든 데이터를 가져오는 대신 일괄적으로 데이터를 검색할 수 있습니다. `SQL` 커서도 거의 동일한 효과를 갖지만 (서버에 약간의 추가 작업이 필요하며, 플래너는 첫 번째 `cursor_tuple_fraction` 행을 최적화하여 가져옵니다), 전체 결과 세트가 아닌 일부를 검색할 수 있습니다.

```
=> BEGIN;
=> DECLARE cur CURSOR FOR
SELECT *
FROM aircrafts
ORDER BY aircraft_code;

=> FETCH 3 FROM cur;
 aircraft_code |           model | range
---------------+-----------------+-------
           319 | Airbus A319-100 | 6700
           320 | Airbus A320-200 | 5700
           321 | Airbus A321-200 | 5600
(3 rows)

=> FETCH 2 FROM cur;
 aircraft_code |          model | range
---------------+----------------+-------
           733 | Boeing 737-300 | 4200
           763 | Boeing 767-300 | 7900
(2 rows)
=> COMMIT;
```

쿼리가 많은 행을 반환하고 클라이언트가 모두 필요한 경우, 시스템 처리량은 일괄 처리 크기에 많이 의존합니다. 일괄 처리 크기가 크면 클라이언트가 서버에 액세스하고 응답을 받는 데 드는 통신 오버헤드가 적어집니다. 그러나 일괄 처리 크기가 커질수록 이러한 이점은 덜 감지됩니다. 행을 하나씩 가져오는 것과 10개 행씩 일괄 처리하는 것의 차이는 엄청나지만, 100개와 1,000개 행의 일괄 처리를 비교하면 훨씬 덜 눈에 띄게 됩니다.

17 장 통계

관계 수준의 기본 통계[209]는 시스템 카탈로그의 `pg_class` 테이블에 저장되어 있으며 다음과 같은 데이터를 포함합니다:

- 관계에 있는 튜플의 수(`reltuples`)
- 관계의 크기, 페이지 단위(`relpages`)
- 가시성 맵에 태그가 지정된 페이지의 수(`relallvisible`)

다음은 `flights` 테이블의 이러한 값입니다:

```
=> SELECT reltuples, relpages, relallvisible
FROM pg_class WHERE relname = 'flights';

 reltuples ¦ relpages ¦ relallvisible
-----------+----------+---------------
    214867 ¦     2624 ¦ 2624
(1 row)
```

쿼리에 필터 조건이 포함되지 않은 경우, `reltuples` 값이 기본 카디널리티 추정값으로 사용됩니다:

```
=> EXPLAIN SELECT * FROM flights;
QUERY PLAN
------------------------------------------------------------------
Seq Scan on flights (cost=0.00..4772.67 rows=214867 width=63)
(1 row)
```

통계는 테이블 분석 중에 수집되며, 수동 및 자동 분석이 모두 해당됩니다.[210] 또한 기본 통계가 매우 중요하므로 이 데이터는 일부 다른 작업(`VACUUM FULL` 및 `CLUSTER`[211], `CREATE INDEX` 및 `REINDEX`[212]) 중에도 계산되며 베큠중에 정교화됩니다.[213]

분석을 위해, `default_statistics_target`(기본값: 100)의 300배에 해당하는 임의의 행들이 샘플링됩니다. 특정 정확도의 통계를 구축하는 데 필요한 샘플 크기는 분석 대상 데이터의 크기에 대한 의존성이 낮기 때문에, 테이블의 크기는 고려되지 않습니다.[214]

샘플링된 행은 동일한 페이지 수(기본 통계 대상의 300배)에서 임의로 선택됩니다.[215] 분명히, 테이블 자체

---

209 postgresql.org/docs/14/planner-stats.html

210 backend/commands/analyze.c, do_analyze_rel function

211 backend/commands/cluster.c, copy_table_data function

212 backend/catalog/heap.c, index_update_stats function

213 backend/access/heap/vacuumlazy.c, heap_vacuum_rel function

214 backend/commands/analyze.c, std_typanalyze function

215 backend/commands/analyze.c, acquire_sample_rows function

가 더 작다면, 읽히는 페이지가 적어질 수 있고, 분석을 위해 선택되는 행도 적어집니다.

대형 테이블의 경우, 통계 수집에 모든 행이 포함되지 않으므로, 추정치가 실제 값과 다를 수 있습니다. 이는 완전히 정상입니다. 데이터가 변경되면, 통계가 항상 정확할 수는 없기 때문입니다. 대략 한 자릿수 정도의 정확도면 적절한 계획을 선택하는 데 충분합니다.

자동 베큠을 비활성화한 `flights` 테이블의 복사본을 만들어보겠습니다. 그래서 자동 분석 시작 시간을 제어할 수 있습니다.

```
=> CREATE TABLE flights_copy(LIKE flights)
WITH (autovacuum_enabled = false);
```

새 테이블에 대한 통계가 아직 없습니다.

```
=> SELECT reltuples, relpages, relallvisible
FROM pg_class WHERE relname = 'flights_copy';
 reltuples ¦ relpages ¦ relallvisible
-----------+----------+---------------
        -1 ¦        0 ¦ 0
(1 row)
```

`reltuples = -1` 값은 아직 분석되지 않은 테이블과 행이 전혀 없는 정말 빈 테이블을 구별하기 위해 사용됩니다.

테이블이 생성된 직후에 일부 행이 테이블에 삽입될 가능성이 매우 높습니다.
따라서 현재 상황을 모르는 상태에서 플래너는 테이블에 10페이지가 포함되어 있다고 가정합니다:

```
=> EXPLAIN SELECT * FROM flights_copy;
QUERY PLAN
------------------------------------------------------------------
Seq Scan on flights_copy (cost=0.00..14.10 rows=410 width=170)
(1 row)
```

행 수는 단일 행의 크기를 기반으로 추정되는데, 이는 계획에서 `width`로 표시됩니다. 행 너비는 일반적으로 분석 중에 계산된 평균 값이지만, 아직 통계가 수집되지 않았기 때문에 여기에서는 단순히 열 데이터 유형에 대한 근사 값입니다.[216]

이제 데이터를 `flights` 테이블에서 복사하고 분석을 수행합니다:

```
=> INSERT INTO flights_copy SELECT * FROM flights;
INSERT 0 214867
=> ANALYZE flights_copy;
```

---

backend/utils/misc/sampling.c

[216] backend/access/table/tableam.c, table_block_relation_estimate_size function

수집된 통계는 실제 행 수(테이블 크기가 분석기가 모든 데이터에 대한 통계를 수집할 만큼 충분히 작음)를 반영합니다:

```
=> SELECT reltuples, relpages, relallvisible
FROM pg_class WHERE relname = 'flights_copy';
 reltuples ¦ relpages ¦ relallvisible
-----------+----------+---------------
    214867 ¦     2624 ¦ 0
(1 row)
```

`relallvisible` 값은 인덱스 전용 스캔의 비용을 추정하는 데 사용됩니다. 이 값은 `VACUUM`에 의해 업데이트 됩니다:

```
=> VACUUM flights_copy;

=> SELECT relallvisible FROM pg_class WHERE relname = 'flights_copy';
relallvisible
---------------
2624
(1 row)
```

이제 통계를 업데이트하지 않고 행 수를 두 배로 늘리고 쿼리 계획의 기본값 추정을 확인합니다:

```
=> INSERT INTO flights_copy SELECT * FROM flights;
=> SELECT count(*) FROM flights_copy;
count
--------
429734
(1 row)
=> EXPLAIN SELECT * FROM flights_copy;
QUERY PLAN
------------------------------------------------------------------------
Seq Scan on flights_copy (cost=0.00..9545.34 rows=429734 width=63)
(1 row)
```

구식 `pg_class` 데이터에도 불구하고 추정이 정확한 것으로 판명됩니다:

```
=> SELECT reltuples, relpages
FROM pg_class WHERE relname = 'flights_copy';
 reltuples ¦ relpages
-----------+----------
    214867 ¦ 2624
(1 row)
```

플래너가 `relpages`와 실제 파일 크기 사이의 차이를 보면 `reltuples` 값을 조정하여 추정 정확도를 향상시킬

수 있습니다.[217] 파일 크기가 `relpages`와 비교하여 두 배로 증가했기 때문에, 플래너는 데이터 밀도가 동일하다고 가정하에 추정된 행 수를 조정합니다.

```
=> SELECT reltuples *
(pg_relation_size('flights_copy') / 8192) / relpages AS tuples
FROM pg_class WHERE relname = 'flights_copy';
tuples
--------
429734
(1 row)
```

물론 이러한 조정이 항상 작동하는 것은 아닙니다(예: 행을 삭제하면 추정이 동일하게 유지됨). 하지만 경우에 따라 플래너가 중요한 변경 사항이 다음 분석 실행을 트리거할 때까지 버틸 수 있도록 해줍니다.

17.2 NULL 값

`NULL` 값은 이론가들에게 비난을 받지만[218], 관계형 데이터베이스에서 여전히 중요한 역할을 합니다. `NULL` 값은 값이 알려지지 않았거나 존재하지 않음을 나타내는 편리한 방법을 제공합니다.

하지만 특별한 값은 특별한 처리를 요구합니다. 이론적인 모순 외에도 고려해야 할 여러 실제적인 문제가 있습니다. 일반적인 부울 논리 대신에 세 가지 값 논리로 대체되므로 `NOT IN`이 예상치 못하게 동작합니다. `NULL` 값은 일반 값보다 큰지 작은지에 대해 명확하지 않습니다(따라서 정렬을 위한 `NULLS FIRST` 및 `NULLS LAST` 절이 있습니다). 집계 함수에서 `NULL` 값이 고려되어야 할지 여부도 명확하지 않습니다. 엄밀히 말하면 `NULL` 값은 전혀 값이 아니므로 플래너는 이를 처리하기 위해 추가 정보가 필요합니다.

릴레이션 수준에서 수집되는 가장 간단한 기본 통계 외에도 분석기는 릴레이션의 각 열에 대한 통계도 수집합니다. 이 데이터는 시스템 카탈로그의 `pg_statistic` 테이블에 저장되지만, `pg_stats` 뷰를 통해 더 편리한 형식으로 이에 접근할 수도 있습니다.

`NULL` 값의 **비율**은 열 수준의 통계에 속하며, 분석 과정에서 계산되며 `null_frac` 속성으로 표시됩니다.

예를 들어, 아직 출발하지 않은 항공편을 검색할 때는 출발 시간이 정의되지 않았다는 것을 기대할 수 있습니다:

```
=> EXPLAIN SELECT * FROM flights WHERE actual_departure IS NULL;
QUERY PLAN
------------------------------------------------------------------
Seq Scan on flights (cost=0.00..4772.67 rows=16702 width=63)
  Filter: (actual_departure IS NULL)
(2 rows)
```

결과를 추정하기 위해 플래너는 전체 행 수에 `NULL` 값의 비율을 곱합니다:

---

[217] backend/access/table/tableam.c, table_block_relation_estimate_size function

[218] sigmodrecord.org/publications/sigmodRecord/0809/p20.date.pdf

```
=> SELECT round(reltuples * s.null_frac) AS rows
FROM pg_class
JOIN pg_stats s ON s.tablename = relname
WHERE s.tablename = 'flights'
AND s.attname = 'actual_departure';
rows
-------
16702
(1 row)
```

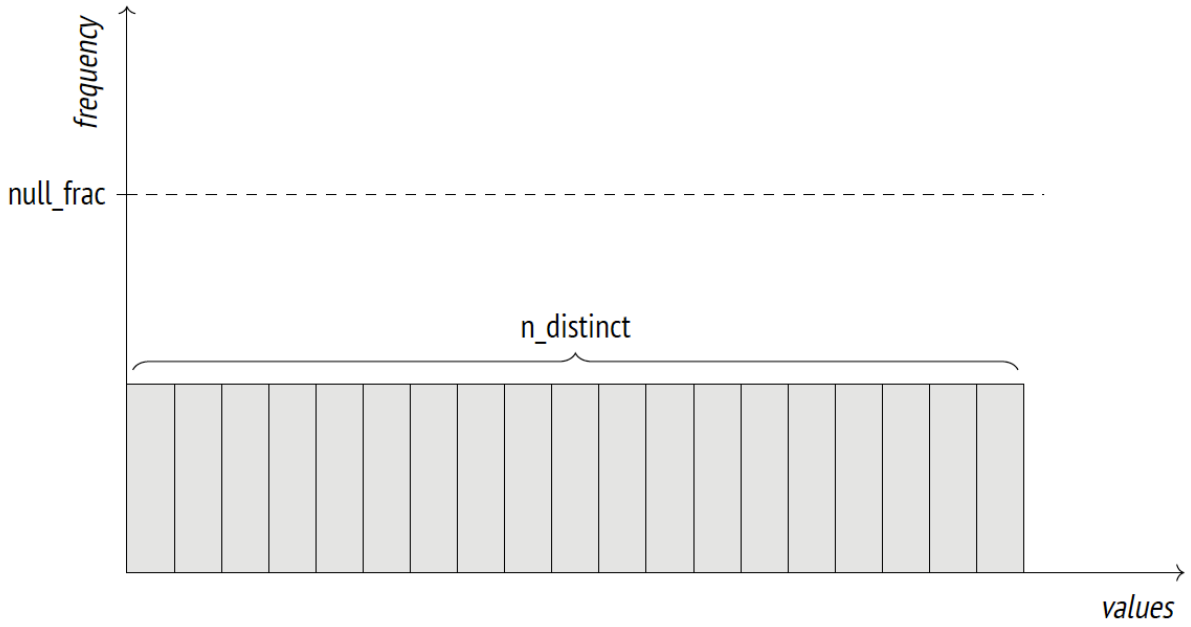
그리고 여기에 실제 행 수가 있습니다:

```
=> SELECT count(*) FROM flights WHERE actual_departure IS NULL;
count
-------
16348
(1 row)
```

17.3 고윳값

`pg_stats` 뷰의 `n_distinct` 필드는 열의 고유한 값의 개수를 나타냅니다.

`n_distinct`가 음수인 경우, 그 절댓값은 실제 개수가 아닌 열의 고유한 값의 비율을 나타냅니다. 예를 들어, -1은 모든 열 값이 고유하다는 것을 나타내고, -3은 각 값이 평균적으로 세 개의 행에 나타난다는 것을 의미합니다. 분석기는 추정된 고유한 값의 개수가 전체 행 수의 10%를 초과하는 경우 분수를 사용합니다. 이 경우, 추가적인 데이터 수정으로 이 비율을 변경하는 것은 불가능할 것입니다.[219]



[219] backend/commands/analyze.c, compute_distinct_stats function

균일한 데이터 분포가 예상되는 경우 고유한 값의 개수가 사용됩니다. 예를 들어, "column = expression" 조건의 기수를 추정할 때, 플래너는 계획 단계에서 정확한 값이 알려지지 않은 경우 표현식이 어떤 열 값이든 동일한 확률로 취할 수 있다고 가정합니다:[220]

```
=> EXPLAIN SELECT *
FROM flights
WHERE departure_airport = (
  SELECT airport_code FROM airports WHERE city = 'Saint Petersburg'
);
QUERY PLAN
------------------------------------------------------------------------
Seq Scan on flights (cost=30.56..5340.40 rows=2066 width=63)
  Filter: (departure_airport = $0)
  InitPlan 1 (returns $0)
      -> Seq Scan on airports_data ml (cost=0.00..30.56 rows=1 wi...
              Filter: ((city ->> lang()) = 'Saint Petersburg'::text)
(5 rows)
```

여기서 InitPlan 노드는 한 번만 실행되며, 계산된 값이 주요 계획에서 사용됩니다.

```
=> SELECT round(reltuples / s.n_distinct) AS rows
FROM pg_class
JOIN pg_stats s ON s.tablename = relname
WHERE s.tablename = 'flights'
AND s.attname = 'departure_airport';
     rows
-------
     2066
(1 row)
```

추정된 고유한 값의 개수가 잘못된 경우(분석된 행의 수가 제한적인 경우) 열 수준에서 재정의할 수 있습니다:

```
ALTER TABLE ...
  ALTER COLUMN ...
  SET (n_distinct = ...);
```

모든 데이터가 항상 균일한 분포를 가진다면, 이 정보(최소값과 최대값과 함께)는 충분합니다. 그러나 균일하지 않은 분포(실제로 훨씬 더 일반적인 경우)의 경우, 이러한 추정은 부정확합니다:

```
=> SELECT min(cnt), round(avg(cnt)) avg, max(cnt)
FROM (
SELECT departure_airport, count(*) cnt
FROM flights
```

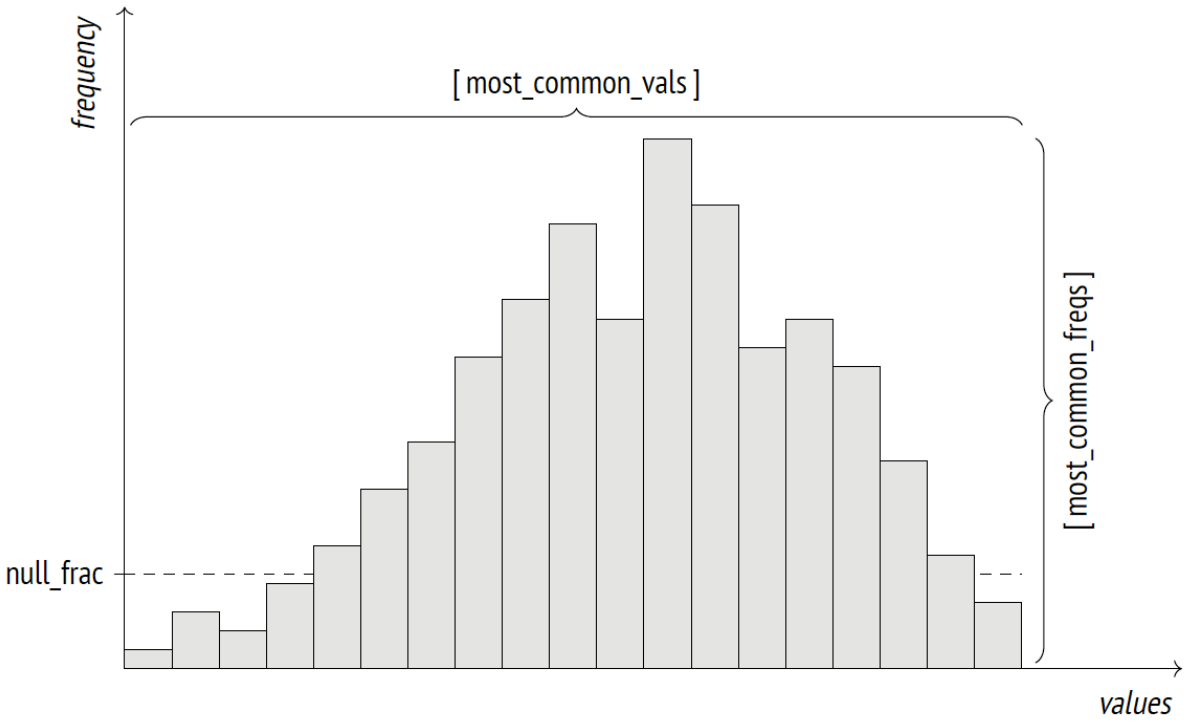
[220] backend/utils/adt/selfuncs.c, var_eq_non_const function

```
GROUP BY departure_airport
) t;
 min ¦  avg ¦ max
-----+------+-------
 113 ¦ 2066 ¦ 20875
(1 row)
```

17.4 가장 일반적인 값

데이터 분포가 균일하지 않은 경우, 추정은 가장 일반적인 값(MCV)과 그들의 빈도에 대한 통계를 기반으로 세밀하게 조정됩니다. pg_stats 뷰는 이러한 배열을 각각 most_common_vals와 most_common_freqs 필드에 표시합니다.

다음은 다양한 항공기 유형에 대한 이러한 통계의 예입니다.




```
=> SELECT most_common_vals AS mcv,
left(most_common_freqs::text,60) ¦¦ '...' AS mcf
FROM pg_stats
WHERE tablename = 'flights' AND attname = 'aircraft_code' \gx
-[ RECORD 1 ]-----------------------------------------------------------
mcv ¦ {CN1,CR2,SU9,321,733,763,319,773}
mcf ¦ {0.27886668,0.27266666,0.26176667,0.057166666,0.037666667,0....
```

"column = value" 조건의 선택도를 추정하기 위해서는 most_common_vals 배열에서 해당 값이 있는지 찾고,

인덱스가 동일한 most_common_freqs 배열 요소에서 빈도를 가져오면 됩니다.[221]

```
=> EXPLAIN SELECT * FROM flights WHERE aircraft_code = '733';
                    QUERY PLAN
------------------------------------------------------------
Seq Scan on flights (cost=0.00..5309.84 rows=8093 width=63)
  Filter: (aircraft_code = '733'::bpchar)
(2 rows)

=> SELECT round(reltuples * s.most_common_freqs[
  array_position((s.most_common_vals::text::text[]),'733')
  ])
FROM pg_class
JOIN pg_stats s ON s.tablename = relname
WHERE s.tablename = 'flights'
AND s.attname = 'aircraft_code';
round
-------
8093
(1 row)
```

이러한 추정은 실제 값과 근접할 것이 분명합니다.

```
=> SELECT count(*) FROM flights WHERE aircraft_code = '733';
count
-------
8263
(1 row)
```

MCV 목록은 부등식 조건의 선택도를 추정하는 데에도 사용됩니다. 예를 들어, "column < value"와 같은 조건은 분석기가 most_common_vals에서 대상 값보다 작은 모든 값을 찾고 most_common_freqs에 나열된 해당 빈도를 합산해야 합니다.[222]

MCV 통계는 고유한 값이 너무 많지 않을 때 가장 잘 작동합니다. 배열의 최대 크기는 default_statistics_target(기본값: 100) 매개변수에 의해 정의되며, 또한 분석 목적으로 무작위로 샘플링할 행 수를 100으로 제한합니다.

일부 경우에는 기본 매개변수 값을 늘리는 것이 의미가 있으며, 이로 인해 MCV 목록이 확장되고 추정의 정확도가 향상됩니다. 이를 열 수준에서 수행할 수 있습니다:

```
ALTER TABLE ...
  ALTER COLUMN ...
  SET STATISTICS ...;
```

[221] backend/utils/adt/selfuncs.c, var_eq_const function

[222] backend/utils/adt/selfuncs.c, scalarineqsel function

샘플 크기도 증가하지만, 지정된 테이블에만 해당됩니다.

MCV 배열은 실제 값들을 저장하기 때문에 상당한 공간을 차지할 수 있습니다. pg_statistic의 크기를 통제하고 플래너에 불필요한 작업을 부과하지 않기 위해 1KB보다 큰 값은 분석 및 통계에서 제외됩니다. 그러나 이러한 큰 값은 고유할 가능성이 높기 때문에 대부분 most_common_vals에 포함되지 않을 것입니다.

17.5 히스토그램

만약 고유한 값들이 배열에 저장하기에 너무 많다면, PostgreSQL은 히스토그램을 사용합니다. 이 경우 값들은 히스토그램의 여러 버킷들 사이에 분포됩니다. 버킷의 개수는 default_statistics_target 매개변수에 의해 제한됩니다.

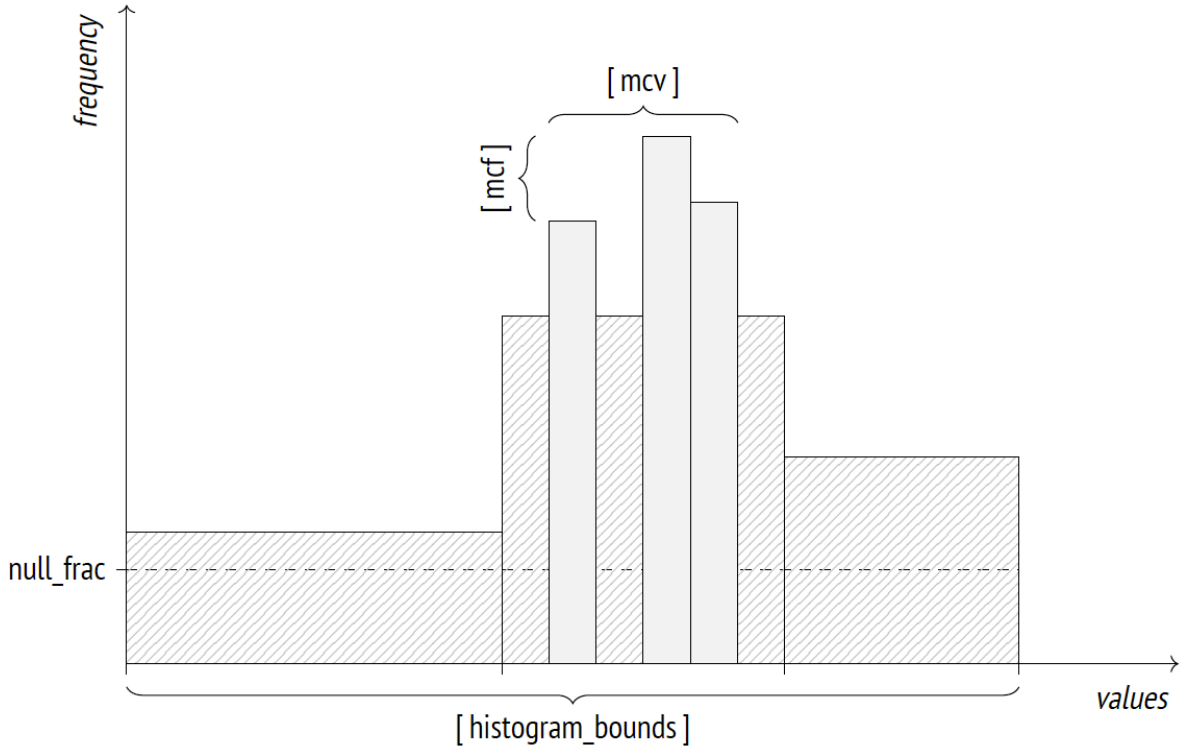
각 버킷의 폭은 대략적으로 동일한 수의 값들이 들어가도록 선택됩니다 (이 특성은 큰 무늬로 칠해진 사각형들의 면적이 동일함으로써 그림으로 나타납니다). MCV 리스트에 포함된 값들은 고려되지 않습니다. 결과적으로, 각 버킷에 포함된 값들의 누적 빈도는 **1/버킷의 개수**와 동일합니다.

히스토그램은 pg_stats 뷰의 histogram_bounds 필드에 버킷의 경계값들의 배열로 저장됩니다:

```
=> SELECT left(histogram_bounds::text,60) ¦¦ '...' AS hist_bounds
FROM pg_stats s
WHERE s.tablename = 'boarding_passes' AND s.attname = 'seat_no';
                     hist_bounds
--------------------------------------------------------------
{10B,10E,10F,10F,11H,12B,13B,14B,14H,15G,16B,17B,17H,19B,19B...
(1 row)
```

MCV 리스트와 결합하여, 히스토그램은 '보다 크다'나 '보다 작다' 조건의 선택도를 추정하는 등의 작업에 사용됩니다.[223] 예를 들어, 맨 뒷열에 발급된 탑승권 수를 살펴보겠습니다.

[223] backend/utils/adt/selfuncs.c, ineq_histogram_selectivity function




```
=> EXPLAIN SELECT * FROM boarding_passes WHERE seat_no > '30B';
                          QUERY PLAN
---------------------------------------------------------------------------
Seq Scan on boarding_passes (cost=0.00..157350.10 rows=2983242 ...
  Filter: ((seat_no)::text > '30B'::text)
(2 rows)
```

저는 의도적으로 히스토그램 버킷 사이의 경계에 위치한 좌석 번호를 선택했습니다.

이 조건의 선택도는 **N/버킷의 개수**로 추정됩니다. 여기서 N은 해당 조건을 만족하는 값들을 가지고 있는 버킷의 개수를 의미합니다 (즉, 지정된 값의 오른쪽에 위치한 값들을 말합니다). 또한 `MCV`는 히스토그램에 포함되지 않는다는 점도 고려해야 합니다.

덧붙여 말하자면, `NULL` 값은 히스토그램에 나타나지 않지만 `seat_no` 열에는 그런 값들이 없습니다.

```
=> SELECT s.null_frac FROM pg_stats s
WHERE s.tablename = 'boarding_passes' AND s.attname = 'seat_no';
 null_frac
-----------
         0
(1 row)
```

먼저, 조건을 만족하는 `MCV`의 비율을 찾아봅시다:

```
=> SELECT sum(s.most_common_freqs[
```

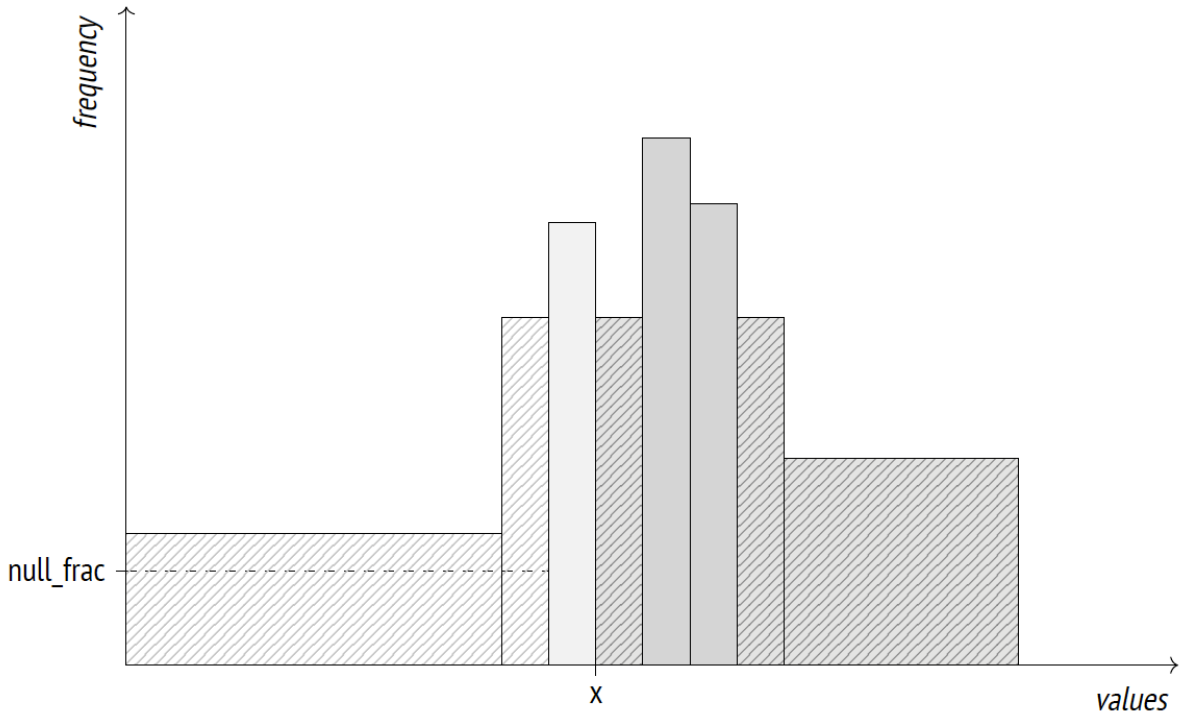
```
array_position((s.most_common_vals::text::text[]),v)
])
FROM pg_stats s, unnest(s.most_common_vals::text::text[]) v
WHERE s.tablename = 'boarding_passes' AND s.attname = 'seat_no'
AND v > '30B';
   sum
------------
0.21226665
(1 row)
```

히스토그램에 의해 무시되는 전체 `MCV`의 비율은 다음과 같습니다:

```
=> SELECT sum(s.most_common_freqs[
array_position((s.most_common_vals::text::text[]),v)
])
FROM pg_stats s, unnest(s.most_common_vals::text::text[]) v
WHERE s.tablename = 'boarding_passes' AND s.attname = 'seat_no';
   sum
------------
0.67816657
(1 row)
```

지정된 조건을 충족하는 값들은 정확히 `N`개의 버킷을 차지하므로 (총 100개의 버킷 중), 다음과 같은 추정값을 얻을 수 있습니다:

```
=> SELECT round( reltuples * (
      0.21226665 -- MCV share
      + (1 - 0.67816657 - 0) * (51 / 100.0) -- histogram share
  ))
FROM pg_class
WHERE relname = 'boarding_passes';
  round
---------
2983242
(1 row)
```



일반적인 경우에는 경계값이 아닌 값들에 대해서는 플래너가 선형 보간을 적용하여 대상 값이 포함된 버킷의 비율을 고려합니다.

다음은 실제로 맨 뒷좌석인 좌석 수입니다:

```
=> SELECT count(*) FROM boarding_passes WHERE seat_no > '30B';
  count
----------
2993735
(1 행)
```

`default_statistics_target` 값을 증가시키면 추정 정확도가 향상될 수 있지만, 우리의 예시에서 보듯이 히스토그램과 `MCV` 리스트를 결합한 경우에는 대부분 좋은 결과를 얻을 수 있습니다. 심지어 열에 많은 고유한 값이 포함되어 있더라도입니다:

```
=> SELECT n_distinct FROM pg_stats
WHERE tablename = 'boarding_passes' AND attname = 'seat_no';
  n_distinct
--------------
  461
(1 행)
```

추정 정확도를 향상시키는 것은 계획 수립에 더 나은 결과를 가져올 때에만 의미가 있습니다. 고려 없이 `default_statistics_target` 값을 증가시키는 것은 계획 수립과 분석을 느리게 할 뿐 아무 이점이 없을 수 있습니다. 그렇지만, 이 매개변수 값을 (0까지) 줄이는 것은 계획 수립과 분석을 빠르게 할 수 있지만 잘못된 계획 선택으로 이어질 수 있습니다. 이러한 절약은 보통 정당화되지 않습니다.

17.6 스칼라가 아닌 데이터 유형에 관한 통계

PostgreSQL은 스칼라가 아닌 데이터 유형에 대해서는 값의 분포뿐만 아니라 이러한 값을 구성하는 요소들의 분포에 대한 통계를 수집할 수 있습니다. 이는 제1 정규형을 따르지 않는 열을 쿼리할 때 계획 정확도를 향상시킵니다.

- `most_common_elems` 및 `most_common_elem_freqs` 배열은 가장 일반적인 요소의 목록과 그 사용 빈도를 보여줍니다. 이러한 통계는 배열[224] 및 `tsvector`[225] 데이터 유형에 대한 연산의 선택도를 추정하는 데 사용됩니다.
- `elem_count_histogram` 배열은 값 내의 고유한 요소 수에 대한 히스토그램을 보여줍니다. 이 데이터는 배열에 대한 연산의 선택도를 추정하는 데 사용됩니다.
- 범위 유형의 경우, PostgreSQL은 범위의 길이 및 하한 및 상한 경계에 관한 분포 히스토그램을 작성합니다. 이러한 히스토그램은 이러한 유형[226]에 대한 다양한 연산의 선택도를 추정하는 데 사용됩니다. 그러나 `pg_stats` 뷰에는 표시되지 않습니다.

다중 범위 데이터 유형[227]에 대해서도 유사한 통계가 수집됩니다.

17.7 평균 필드 길이

`pg_stats` 뷰의 `avg_width` 필드는 열에 저장된 값들의 평균 크기를 나타냅니다. 당연히 정수나 char(3)와 같은 유형의 경우에는 항상 동일한 크기입니다. 그러나 텍스트와 같은 가변 길이의 데이터 유형의 경우, 열마다 크기가 매우 다를 수 있습니다:

```
=> SELECT attname, avg_width FROM pg_stats
WHERE (tablename, attname) IN ( VALUES
('tickets', 'passenger_name'), ('ticket_flights','fare_conditions')
);
         attname ¦ avg_width
-----------------+-----------
 fare_conditions ¦ 8
 passenger_name ¦ 16
 (2 rows)
```

이 통계는 정렬이나 해싱과 같은 작업에 필요한 메모리 양을 추정하는 데 사용됩니다.

---

224 postgresql.org/docs/14/arrays.html
backend/utils/adt/array_typanalyze.c
backend/utils/adt/array_selfuncs.c

225 postgresql.org/docs/14/datatype-textsearch.html
backend/tsearch/ts_typanalyze.c
backend/tsearch/ts_selfuncs.c

226 postgresql.org/docs/14/rangetypes.html
backend/utils/adt/rangetypes_typanalyze.c
backend/utils/adt/rangetypes_selfuncs.c

227 backend/utils/adt/multirangetypes_selfuncs.c

17.8 연관성

`pg_stats` 뷰의 연관성 필드는 데이터의 물리적인 순서와 비교 연산에 의해 정의된 논리적인 순서 간의 상관관계를 나타냅니다. 값들이 엄격히 오름차순으로 저장되어 있다면, 그들의 연관성은 1에 가깝게 될 것입니다. 만약 값들이 내림차순으로 정렬되어 있다면, 연관성은 -1에 가깝게 될 것입니다. 디스크 상의 데이터 분포가 더욱 혼돈스러울수록 연관성은 0에 가까워집니다.

```
=> SELECT attname, correlation
FROM pg_stats WHERE tablename = 'airports_data'
ORDER BY abs(correlation) DESC;
      attname ¦ correlation
--------------+-------------
  coordinates ¦
 airport_code ¦ -0.21120238
         city ¦ -0.1970127
 airport_name ¦ -0.18223621
     timezone ¦ 0.17961165
(5 rows)
```

참고로, 좌표(`coordinates`) 열에 대해서는 이 통계가 수집되지 않습니다: 좌표 데이터 유형에 대해서는 작은 값과 큰 값 연산자가 정의되어 있지 않기 때문입니다.

연관성은 인덱스 스캔의 비용 추정에 사용됩니다.

17.9 표현식 통계

열 수준의 통계는 비교 연산의 왼쪽 또는 오른쪽 부분이 해당 열 자체를 참조하고 어떠한 표현식도 포함하지 않을 경우에만 사용할 수 있습니다. 예를 들어, 플래너는 열의 함수 계산이 통계에 어떤 영향을 미칠지 예측할 수 없으므로 "함수 호출 = 상수"와 같은 조건에 대해서는 선택도가 항상 0.5%로 추정됩니다:[228]

```
=> EXPLAIN SELECT * FROM flights
WHERE extract(
month FROM scheduled_departure AT TIME ZONE 'Europe/Moscow'
) = 1;
                        QUERY PLAN
-----------------------------------------------------------------------
Seq Scan on flights (cost=0.00..6384.17 rows=1074 width=63)
  Filter: (EXTRACT(month FROM (scheduled_departure AT TIME ZONE ...
(2 rows)
=> SELECT round(reltuples * 0.005)
FROM pg_class WHERE relname = 'flights';
  round
-------
  1074
(1 row)
```

[228] backend/utils/adt/selfuncs.c, eqsel function

플래너는 표준 함수를 포함한 함수의 의미에 대해 아무 것도 알지 못합니다. 일반적인 지식에 따르면 1월에 수행되는 항공편은 전체 항공편 수의 약 1/12를 차지할 것으로 예상되는데, 이 값은 예측 값보다 한 단계 크게 나타납니다.

추정을 개선하기 위해서는 열 수준의 통계에 의존하는 대신 표현식 통계를 수집해야 합니다. 이를 위해 두 가지 방법이 있습니다.

확장 표현식 통계

첫 번째 옵션은 확장 표현 통계를 사용하는 것입니다.[229] 이러한 통계는 기본적으로 수집되지 않습니다. 해당 데이터베이스 객체를 수동으로 생성하기 위해 `CREATE STATISTICS` 명령을 실행해야 합니다.

```
> CREATE STATISTICS flights_expr ON (extract(
month FROM scheduled_departure AT TIME ZONE 'Europe/Moscow'
))
FROM flights;
Once the data is gathered, the estimation accuracy improves:
=> ANALYZE flights;
=> EXPLAIN SELECT * FROM flights
WHERE extract(
month FROM scheduled_departure AT TIME ZONE 'Europe/Moscow'
) = 1;
                       QUERY PLAN
-------------------------------------------------------------------
Seq Scan on flights (cost=0.00..6384.17 rows=16667 width=63)
  Filter: (EXTRACT(month FROM (scheduled_departure AT TIME ZONE ...
(2 rows)
```

수집된 통계가 적용되려면 쿼리가 `CREATE STATISTICS` 명령에서 사용된 것과 정확히 동일한 형식으로 표현식을 지정해야 합니다.

`ALTER STATISTICS` 명령을 실행하여 별도로 확장 통계의 크기 제한을 조정할 수 있습니다. 예:

```
=> ALTER STATISTICS flights_expr SET STATISTICS 42;
```

확장 통계와 관련된 모든 메타데이터는 시스템 카탈로그의 `pg_statistic_ext` 테이블에 저장되며 수집된 데이터 자체는 별도의 `pg_statistic_ext_data`라는 테이블에 상주합니다. 이러한 분리는 민감한 정보에 대한 액세스 제어를 구현하는 데 사용됩니다.

특정 사용자가 사용할 수 있는 확장 식 통계는 별도의 보기에서 더 편리한 형식으로 표시할 수 있습니다.

```
=> SELECT left(expr,50) || '...' AS expr,
null_frac, avg_width, n_distinct,
```

[229] postgresql.org/docs/14/planner-stats#PLANNER-STATS-EXTENDED.html
backend/statistics/README

```
most_common_vals AS mcv,
left(most_common_freqs::text,50) ¦¦ '...' AS mcf,
correlation
FROM pg_stats_ext_exprs
WHERE statistics_name = 'flights_expr' \gx
-[ RECORD 1 ]----------------------------------------------------
       expr ¦ EXTRACT(month FROM (scheduled_departure AT TIME ZO...
  null_frac ¦ 0
  avg_width ¦ 8
 n_distinct ¦ 12
        mcv ¦ {8,9,12,3,1,5,6,7,11,10,4,2}
        mcf ¦ {0.12053333,0.11326667,0.0802,0.07976667,0.0775666...
correlation ¦ 0.08355749
```

표현식 인덱스 통계

표현식 인덱스에 대해 수집된 특수 통계를 사용하여 기본 키 추정을 향상시키는 또 다른 방법이 있습니다. 이 통계는 인덱스가 생성될 때 자동으로 수집되므로 테이블에 대해 수행된 것처럼 수행됩니다. 인덱스가 정말 필요한 경우, 이 접근 방식은 매우 편리하다는 것이 밝혀집니다.

```
=> DROP STATISTICS flights_expr;
=> CREATE INDEX ON flights(extract(
month FROM scheduled_departure AT TIME ZONE 'Europe/Moscow'
));
=> ANALYZE flights;
=> EXPLAIN SELECT * FROM flights
WHERE extract(
month FROM scheduled_departure AT TIME ZONE 'Europe/Moscow'
) = 1;
                        QUERY PLAN
-----------------------------------------------------------------------
Bitmap Heap Scan on flights (cost=324.86..3247.92 rows=17089 wi...
  Recheck Cond: (EXTRACT(month FROM (scheduled_departure AT TIME...
      -> Bitmap Index Scan on flights_extract_idx (cost=0.00..320.5...
              Index Cond: (EXTRACT(month FROM (scheduled_departure AT TI...
(4 rows)
```

표현식 인덱스에 대한 통계는 테이블의 통계와 동일한 방식으로 저장됩니다. 예를 들어, `pg_stats` 조회 시 테이블 이름으로 인덱스 이름을 지정하여 고유 값 수를 얻을 수 있습니다.

```
=> SELECT n_distinct FROM pg_stats
WHERE tablename = 'flights_extract_idx';
  n_distinct
-------------
          12
(1 row)
```

`ALTER INDEX` 명령을 사용하여 인덱스 관련 통계의 정확도를 조정할 수 있습니다. 인덱싱된 표현식에 해당하는 열 이름을 모르는 경우 먼저 찾아야 합니다. 예:

```
=> SELECT attname FROM pg_attribute
WHERE attrelid = 'flights_extract_idx'::regclass;
attname
---------
extract
(1 row)

=> ALTER INDEX flights_extract_idx
      ALTER COLUMN extract SET STATISTICS 42;
```

17.10 다변량 통계

다변량 통계를 수집하는 것도 가능하며, 이는 여러 테이블 열에 걸쳐 있을 수 있습니다. 사전 조건으로, `CREATE STATISTICS` 명령을 사용하여 해당하는 확장 통계를 수동으로 생성해야 합니다. PostgreSQL은 세 종류의 다변량 통계를 구현하고 있습니다.

열 간의 기능 종속성

한 열의 값이 다른 열의 값에 완전히 또는 부분적으로 의존하는 경우, 필터 조건에 이러한 열들이 모두 포함되면 카디널리티가 과소평가될 수 있습니다.

두 개의 필터 조건을 포함한 쿼리를 고려해봅시다:

```
=> SELECT count(*) FROM flights
WHERE flight_no = 'PG0007' AND departure_airport = 'VKO';
    count
--------
      396
(1 row)
```

이 값은 크게 과소평가되었습니다:

```
=> EXPLAIN SELECT * FROM flights
WHERE flight_no = 'PG0007' AND departure_airport = 'VKO';
                      QUERY PLAN
-------------------------------------------------------------------------
Bitmap Heap Scan on flights (cost=10.49..816.84 rows=15 width=63)
  Recheck Cond: (flight_no = 'PG0007'::bpchar)
  Filter: (departure_airport = 'VKO'::bpchar)
  -> Bitmap Index Scan on flights_flight_no_scheduled_departure_key
      (cost=0.00..10.49 rows=276 width=0)
      Index Cond: (flight_no = 'PG0007'::bpchar)
(6 rows)
```

이는 상관된 술어들의 잘 알려진 문제입니다. 플래너는 술어들이 서로 의존하지 않는다고 가정하므로, 전체 선택도는 논리적 `AND`로 결합된 필터 조건들의 선택도의 곱으로 추정됩니다. 위의 계획은 이 문제를 명확하게 보여줍니다: `flight_no` 열에 대한 조건에 대한 `Bitmap Index Scan` 노드가, `departure_airport` 열에 대한 조건에 의해 `Bitmap Heap Scan` 노드가 결과를 필터링함에 따라 값이 크게 줄어들었습니다.

하지만 우리는 공항이 항공편 번호에 의해 명확하게 정의된다는 것을 알고 있습니다. 두 번째 조건은 사실상 중복된 조건입니다 (공항 이름에 오류가 있는 경우를 제외하고). 이러한 경우에는 함수적 종속성에 대한 확장 통계를 적용하여 추정을 개선할 수 있습니다.

두 열 사이의 함수적 종속성에 대한 확장 통계를 생성해봅시다:

```
=> CREATE STATISTICS flights_dep(dependencies)
ON flight_no, departure_airport FROM flights;
```

다음 분석 실행에서 이 통계가 수집되며, 추정이 개선됩니다:

```
=> ANALYZE flights;
=> EXPLAIN SELECT * FROM flights
WHERE flight_no = 'PG0007'
AND departure_airport = 'VKO';
                        QUERY PLAN
-----------------------------------------------------------------------
Bitmap Heap Scan on flights (cost=10.57..819.51 rows=277 width=63)
  Recheck Cond: (flight_no = 'PG0007'::bpchar)
  Filter: (departure_airport = 'VKO'::bpchar)
  -> Bitmap Index Scan on flights_flight_no_scheduled_departure_key
      (cost=0.00..10.50 rows=277 width=0)
      Index Cond: (flight_no = 'PG0007'::bpchar)
(6 rows)
```

수집된 통계는 시스템 카탈로그에 저장되며, 다음과 같이 액세스할 수 있습니다:

```
=> SELECT dependencies
FROM pg_stats_ext WHERE statistics_name = 'flights_dep';
dependencies
----------------------------------------
{"2 => 5": 1.000000, "5 => 2": 0.010200}
(1 row)
```

여기서 2와 5는 `pg_attribute` 테이블에 저장된 열 번호입니다. 해당 값은 함수적 종속성의 정도를 정의합니다: 0은 의존성이 없음을 나타내며, 1은 두 번째 열의 값이 첫 번째 열의 값에 완전히 의존함을 나타냅니다.

다변량 고윳값 수

다른 열에 저장된 값들의 고유한 조합의 수에 대한 통계는 여러 열에 대해 수행되는 `GROUP BY` 작업의 카디널

리티 추정을 개선합니다.

예를 들어, 출발 공항과 도착 공항의 가능한 쌍의 추정 수는 공항의 총 수의 제곱이지만, 실제 값은 훨씬 작습니다. 모든 쌍이 직항으로 연결되지 않기 때문입니다:

```
=> SELECT count(*)
FROM (
SELECT DISTINCT departure_airport, arrival_airport FROM flights
) t;
 count
-------
   618
(1 row)
=> EXPLAIN SELECT DISTINCT departure_airport, arrival_airport
FROM flights;
                              QUERY PLAN
-----------------------------------------------------------------------
HashAggregate (cost=5847.01..5955.16 rows=10816 width=8)
  Group Key: departure_airport, arrival_airport
  -> Seq Scan on flights (cost=0.00..4772.67 rows=214867 width=8)
(3 rows)
```

고유한 값에 대한 확장 통계를 정의하고 수집해봅시다:

```
=> CREATE STATISTICS flights_nd(ndistinct)
ON departure_airport, arrival_airport FROM flights;
=> ANALYZE flights;
```

카디널리티 추정이 개선되었습니다:

```
=> EXPLAIN SELECT DISTINCT departure_airport, arrival_airport
FROM flights;
QUERY PLAN
-----------------------------------------------------------------------
HashAggregate (cost=5847.01..5853.19 rows=618 width=8)
  Group Key: departure_airport, arrival_airport
  -> Seq Scan on flights (cost=0.00..4772.67 rows=214867 width=8)
(3 rows)
```

시스템 카탈로그에서 수집된 통계를 확인할 수 있습니다:

```
=> SELECT n_distinct
FROM pg_stats_ext WHERE statistics_name = 'flights_nd';
  n_distinct
---------------
{"5, 6": 618}
```

```
(1 row)
```

다변량 MCV 목록

값의 분포가 균일하지 않은 경우, 추정 정확도는 특정 값 쌍에 크게 의존하기 때문에 기능 종속성에만 의존하는 것으로는 충분하지 않을 수 있습니다. 예를 들어, 계획가는 셰레메티예보 공항에서 운항된 보잉 737 항공편의 수를 과소평가합니다:

```
=> SELECT count(*) FROM flights
WHERE departure_airport = 'SVO' AND aircraft_code = '733';
 count
-------
  2037
(1 row)
=> EXPLAIN SELECT * FROM flights
WHERE departure_airport = 'SVO' AND aircraft_code = '733';
      QUERY PLAN
------------------------------------------------------------------------
Seq Scan on flights (cost=0.00..5847.00 rows=736 width=63)
  Filter: ((departure_airport = 'SVO'::bpchar) AND (aircraft_cod...
(2 rows)
```

이 경우 다변수 `MCV` 목록에 대한 통계를 수집하여 추정을 개선할 수 있습니다:[230]

```
=> CREATE STATISTICS flights_mcv(mcv)
      ON departure_airport, aircraft_code FROM flights;
=> ANALYZE flights;
```

새 카디널리티 추정은 훨씬 더 정확합니다:

```
=> EXPLAIN SELECT * FROM flights
WHERE departure_airport = 'SVO' AND aircraft_code = '733';
                      QUERY PLAN
------------------------------------------------------------------------
Seq Scan on flights (cost=0.00..5847.00 rows=1927 width=63)
  Filter: ((departure_airport = 'SVO'::bpchar) AND (aircraft_cod...
(2 rows)
```

이러한 추정을 얻기 위해 계획가는 시스템 카탈로그에 저장된 빈도 값에 의존합니다:

```
=> SELECT values, frequency
FROM pg_statistic_ext stx
JOIN pg_statistic_ext_data stxd ON stx.oid = stxd.stxoid,
```

[230] backend/statistics/README.mcv
backend/statistics/mcv.c

```
pg_mcv_list_items(stxdmcv) m
WHERE stxname = 'flights_mcv'
AND values = '{SVO,773}';
    values ¦ frequency
-----------+--------------------
 {SVO,773} ¦ 0.005266666666666667
(1 row)
```

일반적인 `MCV` 목록과 마찬가지로 다변수 목록에는 **`default_statistics_target`**(기본값: 100) 값이 저장됩니다(이 매개변수가 열 수준에서도 설정된 경우, 해당 값 중 가장 큰 값이 사용됩니다).
필요한 경우 확장된 표현식 통계와 마찬가지로 목록의 크기를 변경할 수도 있습니다:

```
ALTER STATISTICS ... SET STATISTICS ...;
```

이 모든 예제에서는 두 개의 열만 사용했지만, 더 많은 열의 다변수 통계를 수집할 수도 있습니다.

여러 유형의 통계를 하나의 객체에 결합하려면 해당 정의에 쉼표로 구분된 목록을 제공할 수 있습니다. 유형을 지정하지 않으면 PostgreSQL은 지정된 열에 대해 가능한 모든 유형의 통계를 수집합니다.

실제 열 이름 외에도 다변수 통계는 표현식 통계와 마찬가지로 임의의 표현식을 사용할 수도 있습니다.

18 장 테이블 액세스 방법

18.1 플러그형 스토리지 엔진

PostgreSQL는 여러 가지 테이블 액세스 방법(플러그 가능한 스토리지 엔진)을 생성하고 플러그인으로 사용할 수 있도록 허용합니다. 그러나 현재는 기본 제공되는 액세스 방법이 하나뿐입니다. 테이블을 생성할 때 사용할 엔진을 지정할 수 있으며(`CREATE TABLE ... USING`), 그렇지 않으면 `default_table_access_method` 매개변수에 나열된 기본 엔진인 '`heap`'이 적용됩니다.

PostgreSQL 코어는 여러 엔진과 동일한 방식으로 작동하기 위해 테이블 액세스 방법이 특수 인터페이스를 구현해야 합니다.[231] `amhandler` 열에 지정된 함수는 코어가 필요로 하는 모든 정보를 포함하는 인터페이스 구조체를 반환합니다.[232]

모든 테이블 액세스 방법에서 다음 코어 구성 요소를 사용할 수 있습니다:

- 트랜잭션 관리자(`ACID` 및 스냅샷 격리 지원)
- 버퍼 관리자
- I/O 시스템
- `TOAST`
- 옵티마이저 및 실행기
- 인덱스 지원

이러한 구성 요소는 엔진에 항상 제공되며, 엔진이 모두 사용하지 않더라도 유지됩니다.

한편, 엔진은 다음을 정의합니다:

- 튜플 형식 및 데이터 구조
- 테이블 스캔 구현 및 비용 추정
- 삽입, 삭제, 수정 및 잠금 작업의 구현
- 가시성 규칙
- 백업 및 분석 절차

엔진은 다른 방법들과의 간섭 없이 표준 엔진의 특성을 모두 고려하는 좋은 디자인을 만드는 것이 매우 어렵기 때문에 PostgreSQL은 싱글 내장 데이터 저장소를 사용했고, 적절한 프로그래밍 인터페이스 없이 이를 처리했습니다.

> 예를 들어, `WAL` 을 처리하는 방법은 여전히 불분명합니다. 새로운 액세스 방법은 코어가 인식하지 못하는 자체 작업을 로그로 기록해야 할 수도 있습니다. 기존의 일반적인 `WAL` 메커니즘[233]은 오버헤드가 너무 크기 때문에 적합하지 않습니다. 새로운 `WAL` 항목 유형을 처리하기 위한 또 다른 인터페이스를 추가할 수도 있지만, 이 경우 크래시 복구가 외부 코드에

[231] postgresql.org/docs/14/tableam.html

[232] include/access/tableam.h

[233] postgresql.org/docs/14/generic-wal.html

> 의존하게 되어 매우 바람직하지 않습니다. 지금까지 가장 타당한 해결책은 특정 엔진마다 코어를 패치하는 것입니다.

이러한 이유로 테이블 액세스 방법과 코어 사이에 엄격한 구분을 제공하지 않았습니다. 이 책의 이전 부분에서 설명한 많은 기능은 코어 자체보다는 '`heap`' 액세스 방법에 속합니다. 이 방법이 의 최종 표준 엔진으로 남을 가능성이 높으며, 다른 방법들은 특정 로드 유형의 도전에 대응하기 위해 별도의 역할을 수행할 것입니다.

현재 개발 중인 새로운 엔진 중에서 다음을 언급하고 싶습니다:

Zheap은 테이블의 팽창을 해결하기 위해 개발되었습니다.[234] 이 엔진은 인플레이스 행 업데이트를 구현하고 `MVCC`와 관련된 과거 데이터를 별도의 `undo` 스토리지로 이동시킵니다. 이러한 엔진은 데이터 업데이트가 빈번한 로드에 유용합니다.

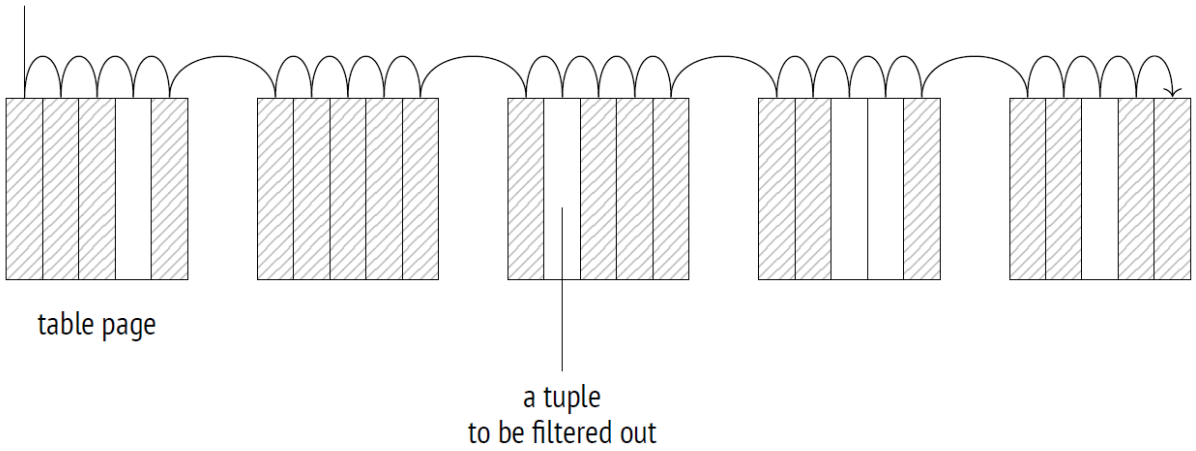
Zheap 아키텍처는 오라클 사용자에게 친숙해 보일 것이지만 몇 가지 뉘앙스가 있습니다(예: 인덱스 액세스 방법의 인터페이스에서 자체 버전 관리 인덱스를 생성할 수 없습니다).

Zedstore는 열 지향 저장소를 구현하며[235], OLAP(Online Analytical Processing) 쿼리와 가장 효율적일 것으로 예상됩니다.

저장된 데이터는 튜플 ID의 B-트리로 구성되며, 각 열은 주 B-트리와 연관된 자체 B-트리에 저장됩니다. 앞으로는 여러 열을 하나의 B-트리에 저장하여 하이브리드 저장소를 얻을 수도 있을 것입니다.

18.2 순차 스캔

저장 엔진은 테이블 데이터의 물리적 레이아웃을 정의하고 액세스 방법을 제공합니다. 지원되는 방법은 순차 스캔으로, 테이블의 주 포크 파일(또는 파일들)을 전체적으로 읽습니다. 각 읽은 페이지에서 각 튜플의 가시성을 확인하며, 쿼리를 만족시키지 못하는 튜플은 필터링됩니다.



스캔 프로세스는 버퍼 캐시를 통해 진행되며, 대용량 테이블이 유용한 데이터를 제거하지 않도록 작은 크기

[234] github.com/EnterpriseDB/zheap

[235] github.com/greenplum-db/postgres/tree/zedstore

의 버퍼 링을 사용합니다. 동일한 테이블을 스캔하는 다른 프로세스는 이 버퍼 링에 참여하여 추가 디스크 읽기를 피합니다. 이러한 스캔을 동기화된 스캔이라고 합니다. 따라서 스캔은 항상 파일의 시작에서 시작할 필요가 없습니다.

순차 스캔은 전체 테이블 또는 그 중 가장 좋은 부분을 읽는 가장 효율적인 방법입니다. 다시 말해, 선택성이 낮을 때 순차 스캔이 가장 가치가 있습니다. (선택성이 높다는 것은 쿼리가 몇 개의 행만 선택해야 하는 경우로, 인덱스를 사용하는 것이 선호됩니다.)

비용 추정

쿼리 실행 계획에서 순차 스캔은 Seq Scan 노드로 표시됩니다:

```
=> EXPLAIN SELECT *
FROM flights;
                       QUERY PLAN
---------------------------------------------------------------
Seq Scan on flights (cost=0.00..4772.67 rows=214867 width=63)
(1 row)
```

기본 통계의 일부로 예상 행 수가 제공됩니다:

```
=> SELECT reltuples FROM pg_class WHERE relname = 'flights';
  reltuples
-----------
     214867
(1 row)
```

비용을 추정할 때 옵티마이저는 디스크 I/O 및 CPU 리소스 두 가지 구성 요소를 고려합니다.[236]

I/O 비용은 테이블의 페이지 수와 페이지를 순차적으로 읽는 가정하에 한 페이지를 읽는 비용을 곱하여 계산됩니다. 버퍼 매니저가 페이지를 요청할 때 운영 체제는 실제로 디스크에서 더 많은 데이터를 읽으므로, 연속된 여러 페이지가 운영 체제 캐시에 이미 존재할 가능성이 매우 높습니다. 따라서 순차 스캔을 사용하여 단일 페이지를 읽는 비용(옵티마이저가 `seq_page_cost`(기본값: 1)로 추정)은 임의 접근 비용(임의 접근 비용은 `random_page_cost`(기본값: 4) 값으로 정의됨)보다 낮습니다.

기본 설정은 `HDD`에 대해 잘 작동합니다. 그러나 `SSD`를 사용하는 경우 `random_page_cost` 값을 상당히 낮추는 것이 좋습니다. (`seq_page_cost` 매개변수는 참고 값으로 보통 그대로 두는 것이 일반적입니다). 이러한 매개변수 사이의 최적 비율은 하드웨어에 따라 다르기 때문에 일반적으로 테이블스페이스 수준에서 설정됩니다. (`ALTER TABLESPACE ... SET`을 사용하여 설정합니다).

```
=> SELECT relpages,
    current_setting('seq_page_cost') AS seq_page_cost,
    relpages * current_setting('seq_page_cost')::real AS total
FROM pg_class WHERE relname = 'flights';
```

[236] backend/optimizer/path/costsize.c, cost_seqscan function

```
 relpages ¦ seq_page_cost ¦ total
----------+---------------+-------
     2624 ¦             1 ¦ 2624
(1 row)
```

이러한 계산은 적시에 수행되지 않은 VACUUM으로 인해 발생하는 테이블 확장의 결과를 명확히 보여줍니다: 테이블의 주된 포크가 클수록 페이지를 스캔해야 하는 양이 증가하며, 해당 페이지에 포함된 실제 행의 수와는 관계없이 이렇게 됩니다.

CPU 리소스 추정은 각 튜플을 처리하는 데 필요한 비용을 포함합니다. 이 비용은 옵티마이저가 `cpu_tuple_cost`(기본값: 0.01)로 추정합니다.

```
=> SELECT reltuples,
current_setting('cpu_tuple_cost') AS cpu_tuple_cost,
reltuples * current_setting('cpu_tuple_cost')::real AS total
FROM pg_class WHERE relname = 'flights';
 reltuples ¦ cpu_tuple_cost ¦ total
-----------+----------------+---------
    214867 ¦           0.01 ¦ 2148.67
(1 row)
```

이 두 추정치의 합은 실행 계획의 총 비용을 나타냅니다. 순차 스캔은 사전 조건이 없으므로 시작 비용은 0입니다.

스캔된 테이블이 필터링되어야 하는 경우, 적용된 필터 조건은 Seq Scan 노드의 Filter 섹션에 표시됩니다. 추정 행 수는 이러한 조건의 선택도에 따라 달라지며, 비용 추정에는 관련된 계산 비용이 포함됩니다.

`EXPLAIN ANALYZE` 명령은 실제로 반환된 행 수와 필터링된 행 수를 모두 표시합니다. 이를 통해 쿼리의 실행 결과와 필터링 결과를 확인할 수 있습니다.

```
=> EXPLAIN (analyze, timing off, summary off)
SELECT * FROM flights
WHERE status = 'Scheduled';
                QUERY PLAN
-----------------------------------------------
Seq Scan on flights
  (cost=0.00..5309.84 rows=15383 width=63)
  (actual rows=15383 loops=1)
  Filter: ((status)::text = 'Scheduled'::text)
  Rows Removed by Filter: 199484
(5 rows)
```

보다 복잡한 집계를 사용하는 실행 계획을 살펴보겠습니다.

```
=> EXPLAIN SELECT count(*) FROM seats;
                        QUERY PLAN
```

```
------------------------------------------------------------
Aggregate (cost=24.74..24.75 rows=1 width=8)
-> Seq Scan on seats (cost=0.00..21.39 rows=1339 width=0)
(2 rows)
```

이 실행 계획은 두 개의 노드로 구성됩니다. 상위 노드(`Aggregate`)는 `count` 함수를 계산하고, 데이터를 스캔하는 하위 노드(`Seq Scan`)에서 데이터를 가져옵니다.

상위 노드인 `Aggregate`의 시작 비용에는 집계 작업 자체가 포함됩니다. 첫 번째 행(이 경우에는 유일한 행)을 가져오기 위해서는 하위 노드에서 모든 행을 가져와야 합니다. 집계 비용은 각 입력 행에 대한 조건 연산의 실행 비용(estimated at `cpu_operator_cost`(default: 0.0025)로 추정)을 기반으로 추정됩니다.[237]

```
=> SELECT reltuples,
  current_setting('cpu_operator_cost') AS cpu_operator_cost,
  round((
      reltuples * current_setting('cpu_operator_cost')::real
  )::numeric, 2) AS cpu_cost
FROM pg_class WHERE relname = 'seats';
 reltuples ¦ cpu_operator_cost ¦ cpu_cost
-----------+-------------------+----------
      1339 ¦            0.0025 ¦ 3.35
(1 row)
```

받은 추정치는 `Seq Scan` 노드의 총 비용에 추가됩니다.

`Aggregate` 노드의 총 비용에는 반환될 행의 처리 비용도 포함됩니다. 이 비용은 `cpu_tuple_cost`(기본값: 0.01)로 추정된 각 행의 처리 비용을 의미합니다:

```
=> WITH t(cpu_cost) AS (
  SELECT round((
      reltuples * current_setting('cpu_operator_cost')::real
      )::numeric, 2)
  FROM pg_class WHERE relname = 'seats'
  )
SELECT 21.39 + t.cpu_cost AS startup_cost,
  round((
      21.39 + t.cpu_cost + 1 * current_setting('cpu_tuple_cost')::real
      )::numeric, 2) AS total_cost
FROM t;
 startup_cost ¦ total_cost
--------------+------------
        24.74 ¦ 24.75
(1 row)
```

[237] backend/optimizer/path/costsize.c, cost_agg function

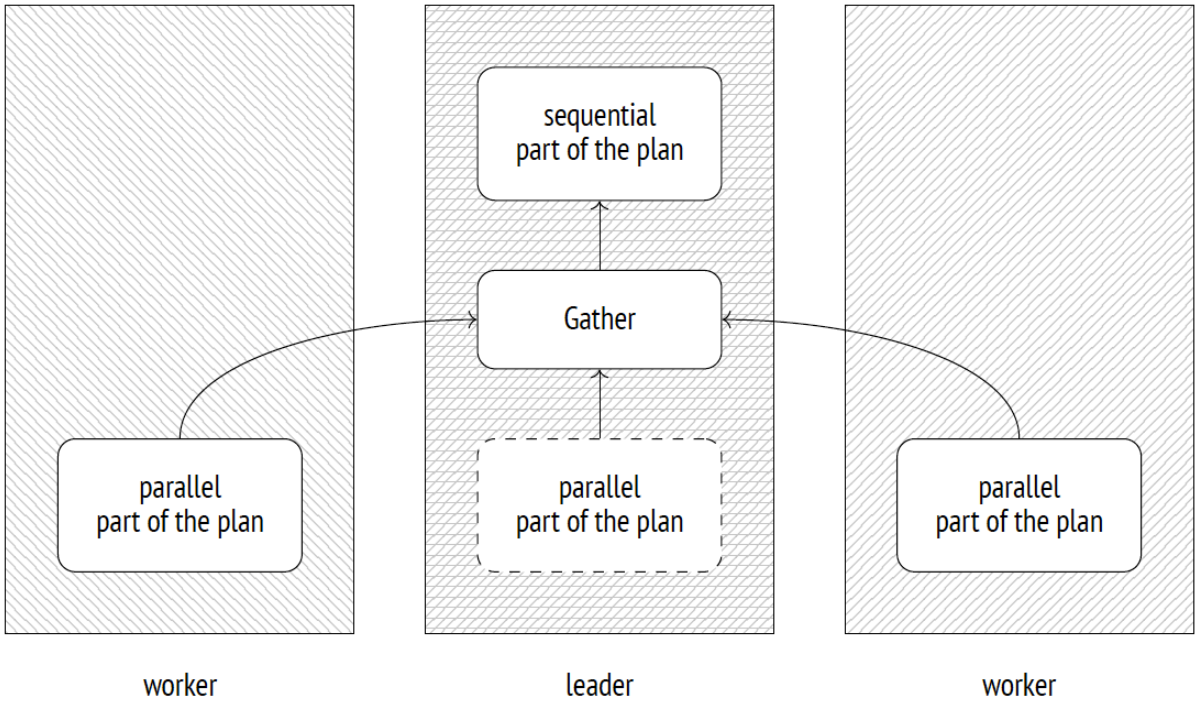
따라서 비용 추정의 종속성을 다음과 같이 그릴 수 있습니다:

```
                QUERY PLAN
-------------------------------------------
Aggregate
  (cost=24.74..24.75 rows=1 width=8)
  -> Seq Scan on seats
       (cost=0.00..21.39 rows=1339 width=0)
(4 rows)
```

18.3 병렬 플랜

PostgreSQL은 병렬 쿼리 실행을 지원합니다. [238]쿼리를 수행하는 주요 프로세스는 (`postmaster`를 통해) 여러 개의 워커 프로세스를 생성하여 동시에 동일한 병렬 계획 부분을 실행합니다. 결과는 `Gather`[239] 노드에서 모아지며 리더에게 전달됩니다. 데이터를 수집하지 않을 경우, 리더는 병렬 계획의 실행에도 참여할 수 있습니다.

필요한 경우, `parallel_leader_participation`(기본값: `on`) 매개변수를 사용하여 리더의 병렬 계획 실행 참여를 금지할 수 있습니다. 이 매개변수를 끄면 리더는 병렬 계획 실행에 참여하지 않습니다.



당연히 이러한 프로세스를 시작하고 데이터를 전송하는 것은 무료가 아니기 때문에 모든 쿼리를 병렬화해야 하는 것은 아닙니다.

---

[238] postgresql.org/docs/14/parallel-query.html
backend/access/transam/README.parallel

[239] backend/executor/nodeGather.c

또한, 병렬 실행이 허용되더라도 계획의 모든 부분을 동시에 처리할 수 있는 것은 아닙니다. 일부 작업은 리더만 단독으로 순차적으로 수행됩니다.

> PostgreSQL 은 다른 접근 방식인 여러 워커가 가상으로 어셈블리 라인을 형성하여 데이터 처리를 수행하는 방식을 지원하지 않습니다. (말 그대로 각 계획 노드를 별도의 프로세스에서 수행하는 방식) 이러한 메커니즘은 PostgreSQL 개발자들에게 효율적이지 않다고 판단되었습니다.

18.4 병렬 순차 스캔

`Parallel Seq Scan` 노드는 병렬 처리를 위해 설계된 노드 중 하나로, **병렬 순차 스캔**을 수행합니다.

이름이 약간 모순적으로 들릴 수 있지만 (스캔이 순차인지 병렬인지), 이 작업의 본질을 반영합니다. 파일 액세스를 살펴보면 테이블 페이지는 순차적으로 읽히며, 단순한 순차 스캔이 수행되었을 때의 순서를 따릅니다. 그러나 이 작업은 여러 개의 동시 프로세스에 의해 수행됩니다. 동일한 페이지를 두 번 스캔하는 것을 피하기 위해 실행자는 공유 메모리를 통해 이러한 프로세스들을 동기화합니다.

여기에서 주목할 점은 운영 체제가 일반적인 순차 스캔의 전체 그림을 파악하지 못하고 대신 여러 프로세스가 무작위 읽기를 수행하는 것을 본다는 것입니다. 따라서 일반적으로 순차 스캔을 가속화하는 데이터 프리페칭은 사실상 쓸모가 없어집니다. 이러한 불필요한 효과를 최소화하기 위해 PostgreSQL은 각 프로세스에게 연속적인 여러 페이지를 할당합니다.[240]

따라서 병렬 스캔은 일반적으로 읽기 비용이 프로세스 간 데이터 전송으로 인해 추가로 증가하기 때문에 큰 의미가 없습니다. 그러나 워커가 검색된 행에 대해 후속 처리(예: 집계)를 수행하는 경우 전체 실행 시간이 훨씬 짧아질 수 있습니다.

비용 추정

커다란 테이블에서 집계를 수행하는 간단한 쿼리의 실행 계획을 살펴보겠습니다. 실행 계획은 병렬화되었습니다.

```
=> EXPLAIN SELECT count(*) FROM bookings;
QUERY PLAN
------------------------------------------------------------------
Finalize Aggregate (cost=25442.58..25442.59 rows=1 width=8)
  -> Gather (cost=25442.36..25442.57 rows=2 width=8)
      Workers Planned: 2
      -> Partial Aggregate
            (cost=24442.36..24442.37 rows=1 width=8)
            -> Parallel Seq Scan on bookings
                  (cost=0.00..22243.29 rows=879629 width=0)
(7 rows)
```

[240] backend/access/heap/heapam.c, table_block_parallelscan_startblock_init & table_block_parallelscan_nextpage functions

`Gather` 노드 아래에 있는 모든 노드는 실행 계획의 병렬 부분에 속합니다. 이들은 각각의 워커 프로세스(여기서는 두 개의 프로세스가 계획되었습니다)에 의해 실행되며, 리더 프로세스에 의해서도 실행될 수 있습니다(단, `parallel_leader_participation` 매개변수로 이 기능을 끌 수 있습니다). `Gather` 노드 자체와 그 위의 모든 노드는 순차적인 부분으로, 리더 프로세스에 의해서만 실행됩니다.

`Parallel Seq Scan` 노드는 병렬 힙 스캔을 나타냅니다. `rows` 필드는 단일 프로세스가 처리할 예상 평균 행 수를 나타냅니다. 전체 실행은 세 개의 프로세스(하나의 리더와 두 개의 워커)에 의해 수행되지만, 리더 프로세스는 더 적은 행을 처리합니다. 워커의 수가 증가함에 따라 리더의 할당량은 작아지는데,[241] 이 경우에는 2.4의 비율로 할당됩니다.

```
=> SELECT reltuples::numeric, round(reltuples / 2.4) AS per_process
FROM pg_class WHERE relname = 'bookings';
 reltuples ¦ per_process
-----------+-------------
   2111110 ¦ 879629
(1 row)
```

`Parallel Seq Scan`의 비용은 순차 스캔과 유사하게 계산됩니다. 각 프로세스가 처리하는 행의 수가 적기 때문에 받는 값은 더 작습니다. 여전히 전체 테이블을 페이지 단위로 읽어야 하므로 `I/O` 부분은 전체 비용에 포함됩니다:

```
=> SELECT round((
       relpages * current_setting('seq_page_cost')::real +
reltuples / 2.4 * current_setting('cpu_tuple_cost')::real
)::numeric, 2)
FROM pg_class WHERE relname = 'bookings';
      round
----------
   22243.29
(1 row)
```

다음으로, `Partial Aggregate` 노드는 검색된 데이터의 집계를 수행합니다. 이 특정 예에서는 행의 수를 계산합니다.

집계 비용은 일반적인 방식으로 추정되며, 테이블 스캔의 비용 추정에 추가됩니다.

```
=> WITH t(startup_cost)
AS (
  SELECT 22243.29 + round((
      reltuples / 2.4 * current_setting('cpu_operator_cost')::real
      )::numeric, 2)
  FROM pg_class
  WHERE relname = 'bookings'
```

[241] backend/optimizer/path/costsize.c, get_parallel_divisor function

```
)
SELECT startup_cost,
  startup_cost + round((1 * current_setting('cpu_tuple_cost')::real
      )::numeric, 2) AS total_cost
FROM t;
 startup_cost ¦ total_cost
--------------+------------
     24442.36 ¦ 24442.37
(1 row)
```

다음 노드인 `Gather`는 리더 프로세스에 의해 실행됩니다. 이 노드는 워커를 시작하고 그들이 반환한 데이터를 수집하는 역할을 담당합니다.

계획을 위해 프로세스 시작 비용(수의 제한 없이)은 `parallel_setup_cost` 매개변수(기본값: 1000)로 정의되며, 프로세스 간의 각 행 전송 비용은 `parallel_tuple_cost` 매개변수(기본값: 0.1)로 추정됩니다.

이 예시에서는 프로세스 시작 비용(프로세스 시작에 소비되는 비용)이 우선합니다. 이 값은 `Partial Aggregate` 노드의 시작 비용에 추가됩니다. 전체 비용에는 두 개의 행 전송 비용도 포함되는데, 이 값은 `Partial Aggregate` 노드의 총 비용에 추가됩니다.[242]

```
=> SELECT
  24442.36 + round(
              current_setting('parallel_setup_cost')::numeric,2) AS setup_cost,
  24442.37 + round(current_setting('parallel_setup_cost')::numeric +
              2 * current_setting('parallel_tuple_cost')::numeric,2) AS total_cost;
 setup_cost ¦ total_cost
------------+------------
   25442.36 ¦ 25442.57
(1 row)
```

마지막으로, `Finalize Aggregate` 노드는 `Gather` 노드로부터 병렬 프로세스로부터 받은 모든 부분 결과를 집계합니다.

최종 집계는 다른 집계와 마찬가지로 추정됩니다. 시작 비용은 세 개의 행을 집계하는 비용을 기반으로 합니다. 이 값은 `Gather`의 총 비용에 추가됩니다(결과를 계산하기 위해 모든 행이 필요하기 때문). `Finalize Aggregate`의 총 비용에는 또한 한 개의 행을 반환하는 비용도 포함됩니다.

```
=> WITH t(startup_cost) AS (
    SELECT 25442.57 + round((3 * current_setting('cpu_operator_cost')::real
            )::numeric, 2)
    FROM pg_class WHERE relname = 'bookings'
)
SELECT startup_cost,
```

[242] backend/optimizer/path/costsize.c, cost_gather function

```
    startup_cost + round((
        1 * current_setting('cpu_tuple_cost')::real)::numeric, 2) AS total_cost
FROM t;
 startup_cost ¦ total_cost
--------------+------------
     25442.58 ¦ 25442.59
(1 row)
```

비용 추정 사이의 종속성은 노드가 결과를 상위 노드에 전달하기 전에 데이터를 누적해야 하는지 여부에 따라 결정됩니다. 집계는 모든 입력 행을 받을 때까지 결과를 반환할 수 없기 때문에, 시작 비용은 하위 노드의 총 비용을 기반으로 합니다. 그러나 `Gather` 노드는 데이터를 가져오자마자 상위로 행을 전송하기 시작합니다. 따라서 이 작업의 시작 비용은 하위 노드의 시작 비용에 의존하며, 총 비용은 하위 노드의 총 비용을 기반으로 합니다.

다음은 종속성 그래프입니다:

```
                        QUERY PLAN
-----------------------------------------------------------
Finalize Aggregate
  (cost=25442.58..25442.59 rows=1 width=8)
  -> Gather
      (cost=25442.36..25442.57 rows=2 width=8)
      Workers Planned: 2
      -> Partial Aggregate
              (cost=24442.36..24442.37 rows=1 width=8)
              -> Parallel Seq Scan on bookings
                      (cost=0.00..22243.29 rows=879629 width=0)
(9 rows)
```

18.5 병렬 실행 제한

백그라운드 워커 수

프로세스의 수는 세 개의 매개변수 계층에 의해 제어됩니다. 동시에 실행되는 백그라운드 워커의 최대 수는 `max_worker_processes`(기본값: 8) 값에 의해 정의됩니다.

그러나 병렬 쿼리 실행은 백그라운드 워커가 필요한 유일한 작업은 아닙니다. 예를 들어, 논리 복제에도 참여하며 확장 프로그램에서도 사용될 수 있습니다. 병렬 계획 실행을 위해 할당된 프로세스 수는 `max_parallel_workers`(기본값: 8) 값으로 제한됩니다.

이 중 최대 `max_parallel_workers_per_gather`(기본값: 2) 개의 프로세스가 하나의 리더에 대해 서비스를 할 수 있습니다.

이러한 매개변수 값의 선택은 다음 요소에 따라 달라집니다:

- 하드웨어 성능: 시스템은 병렬 실행에 전용된 여분의 코어가 있어야 합니다.

- 테이블 크기: 데이터베이스에는 큰 테이블이 있어야 합니다.
- 일반적인 작업 부하: 병렬 실행에서 잠재적으로 이점을 얻을 수 있는 쿼리가 있어야 합니다.

이러한 기준은 `OLTP` 시스템보다는 `OLAP` 시스템에서 일반적으로 충족됩니다.

만약 읽어야 할 힙 데이터의 예상 크기가 `min_parallel_table_scan_size`(기본값: 8MB) 값보다 작다면, 플래너는 병렬 실행을 고려하지 않습니다.

`parallel_workers` 저장 매개변수를 통해 특정 테이블에 대한 프로세스 수를 명시적으로 지정하지 않는 한, 다음 공식에 따라 계산됩니다:

$$1 + \left\lfloor \log_3 \left(\frac{\text{table size}}{min\_parallel\_table\_scan\_size} \right) \right\rfloor$$

이는 테이블이 세 배로 성장할 때마다 PostgreSQL은 처리를 위해 하나의 추가 병렬 워커를 할당한다는 의미입니다. 기본 설정에서는 다음과 같은 숫자가 제공됩니다:

테이블 MB	프로세스 수
8	1
24	2
72	3
216	4
648	5
1944	6

어떤 경우에도 병렬 워커의 수는 `max_parallel_workers_per_gather` 매개변수로 정의된 제한을 초과할 수 없습니다.

만약 19MB 크기의 작은 테이블을 쿼리한다면, 하나의 워커만 계획되고 실행될 것입니다.

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT count(*) FROM flights;
                      QUERY PLAN
------------------------------------------------------------------------
Finalize Aggregate (actual rows=1 loops=1)
  -> Gather (actual rows=2 loops=1)
      Workers Planned: 1
      Workers Launched: 1
      -> Partial Aggregate (actual rows=1 loops=2)
              -> Parallel Seq Scan on flights (actual rows=107434 lo...
(6 rows)
```

105MB 크기의 테이블에 대한 쿼리는 `max_parallel_workers_per_gather`(기본값: 2) 워커의 제한에 도달

하여 두 개의 프로세스만 사용됩니다:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT count(*) FROM bookings;
                      QUERY PLAN
------------------------------------------------------------------
Finalize Aggregate (actual rows=1 loops=1)
  -> Gather (actual rows=3 loops=1)
      Workers Planned: 2
      Workers Launched: 2
      -> Partial Aggregate (actual rows=1 loops=3)
              -> Parallel Seq Scan on bookings (actual rows=703703 l...
(6 rows)
```

만약 이 제한을 제거한다면, 예상되는 세 개의 프로세스를 얻을 수 있을 것입니다.

```
=> ALTER SYSTEM SET max_parallel_workers_per_gather = 4;
=> SELECT pg_reload_conf();
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT count(*) FROM bookings;
                      QUERY PLAN
------------------------------------------------------------------
Finalize Aggregate (actual rows=1 loops=1)
  -> Gather (actual rows=4 loops=1)
      Workers Planned: 3
      Workers Launched: 3
      -> Partial Aggregate (actual rows=1 loops=4)
              -> Parallel Seq Scan on bookings (actual rows=527778 l...
(6 rows)
```

쿼리 실행 중에 사용 가능한 슬롯의 수가 계획된 값보다 작은 경우, 사용 가능한 워커 수만 실행됩니다.

총 병렬 프로세스 수를 다섯 개로 제한하고 두 개의 쿼리를 동시에 실행해 봅시다.

```
=> ALTER SYSTEM SET max_parallel_workers = 5;
=> SELECT pg_reload_conf();
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT count(*) FROM bookings;

  => EXPLAIN (analyze, costs off, timing off, summary off)
  SELECT count(*) FROM bookings;
                             QUERY PLAN
  ------------------------------------------------------------------
  Finalize Aggregate (actual rows=1 loops=1)
  -> Gather (actual rows=3 loops=1)
      Workers Planned: 3
```

```
      Workers Launched: 2
      -> Partial Aggregate (actual rows=1 loops=3)
      -> Parallel Seq Scan on bookings (actual rows=7037...
  (6 rows)

                        QUERY PLAN
------------------------------------------------------------------------
Finalize Aggregate (actual rows=1 loops=1)
  -> Gather (actual rows=4 loops=1)
      Workers Planned: 3
      Workers Launched: 3
      -> Partial Aggregate (actual rows=1 loops=4)
              -> Parallel Seq Scan on bookings (actual rows=527778 l...
(6 rows)
```

두 경우 모두 세 개의 프로세스가 예상되었지만, 한 쿼리는 슬롯 두 개만 얻을 수 있었습니다.

기본 설정으로 복원해 봅시다.

```
=> ALTER SYSTEM RESET ALL;
=> SELECT pg_reload_conf();
```

비 병렬 쿼리

모든 쿼리를 병렬화할 수 있는 것은 아닙니다.[243] 특히 다음과 같은 쿼리 유형에는 병렬 계획을 사용할 수 없습니다:

- 데이터를 수정하거나 잠그는 쿼리(UPDATE, DELETE, SELECT FOR UPDATE 등)는 병렬 계획을 사용할 수 없습니다. 하지만 이 제한은 다음 명령문 내의 서브쿼리에는 적용되지 않습니다:

  ```
  - CREATE TABLE AS, SELECT INTO, CREATE MATERIALZED VIEW 등
  - REFRESH MATERIALIZED VIEW
  ```

  그러나 모든 경우에 여전히 행 삽입은 순차적으로 수행됩니다.

- 일시 중지할 수 있는 쿼리입니다. 이는 커서 내에서 실행되는 쿼리에 적용되며, `PL/pgSQL`의 `FOR` 루프를 포함합니다.

- `PARALLEL UNSAFE` 함수를 호출하는 쿼리입니다. 기본적으로 이는 사용자 정의 함수와 일부 표준 함수입니다. 안전하지 않은 함수의 전체 목록은 시스템 카탈로그를 조회하여 확인할 수 있습니다:

  ```
  SELECT * FROM pg_proc WHERE proparallel = 'u';
  ```

[243] postgresql.org/docs/14/when-can-parallel-query-be-used.html

- 병렬화된 쿼리에서 호출되는 함수 내의 쿼리는 작업자 수의 재귀적인 증가를 피하기 위해 병렬 처리되지 않습니다.

PostgreSQL의 향후 버전에서는 이러한 제한 중 일부가 제거될 수 있습니다. 예를 들어, 직렬화 격리 수준에서 쿼리를 병렬화할 수 있는 기능은 이미 제공되고 있습니다.

> 현재 `INSERT` 및 `COPY`와 같은 명령을 사용하여 행을 병렬로 삽입하는 기능은 개발 중에 있습니다[244].

쿼리가 병렬화되지 않을 수 있는 여러 가지 이유가 있을 수 있습니다:

- 해당 유형의 쿼리는 전혀 병렬화를 지원하지 않습니다.
- 서버 설정으로 인해 병렬 계획 사용이 금지되었습니다(예: 테이블 크기 제한으로 인해).
- 병렬 계획이 순차 계획보다 더 비용이 많이 듭니다.

쿼리가 병렬화될 수 있는지 확인하기 위해 `force_parallel_mode`(기본값: off) 매개변수를 일시적으로 활성화할 수 있습니다. 그러면 플래너는 가능한 경우에 병렬 계획을 생성할 것입니다.

```
=> EXPLAIN SELECT * FROM flights;
                       QUERY PLAN
------------------------------------------------------------
Seq Scan on flights (cost=0.00..4772.67 rows=214867 width=63)
(1 row)

=> SET force_parallel_mode = on;
=> EXPLAIN SELECT * FROM flights;
                       QUERY PLAN
------------------------------------------------------------------------
  Gather (cost=1000.00..27259.37 rows=214867 width=63)
      Workers Planned: 1
      Single Copy: true
      -> Seq Scan on flights (cost=0.00..4772.67 rows=214867 width=63)
(4 rows)
```

병렬 제한된 쿼리

병렬 계획의 병렬 부분이 클수록 성능 향상이 가능합니다. 그러나 특정 작업은 병렬화와는 관련이 없지만 엄격하게 순차적으로 리더 프로세스에 의해 실행됩니다.[245] 다시 말해, 이러한 작업은 `Gather` 노드 아래의 계획 트리에 나타날 수 없습니다. 이러한 작업은 일반적으로 병렬화를 방해하지 않는 작업이지만, 현재 구현에서는 순차적으로 실행됩니다.

---

[244] commitfest.postgresql.org/32/2844
commitfest.postgresql.org/32/2841
commitfest.postgresql.org/32/2610

[245] postgresql.org/docs/14/parallel-safety.html

확장할 수 없는 서브쿼리의 가장 명확한 예는 CTE 결과를 스캔하는 것입니다.[246] 이는 실행 계획에서 CTE Scan 노드로 표시됩니다.

```
=> EXPLAIN (costs off)
WITH t AS MATERIALIZED (
  SELECT * FROM flights
)
SELECT count(*) FROM t;
  QUERY PLAN
----------------------------
Aggregate
  CTE t
      -> Seq Scan on flights
  -> CTE Scan on t
(4 rows)
```

`CTE`가 자체적으로 계산되는 경우에는 `CTE Scan` 노드가 실행 계획에 포함되지 않으므로 이 제한은 적용되지 않습니다.

그러나 `CTE` 자체는 병렬 모드에서 계산될 수 있습니다. 이는 비용이 더 저렴한 경우에 해당합니다. 따라서 `CTE`를 병렬화된 방식으로 계산할 수 있습니다.

```
=> EXPLAIN (costs off)
WITH t AS MATERIALIZED (
  SELECT count(*) FROM flights
)
SELECT * FROM t;
              QUERY PLAN
-----------------------------------------------
CTE Scan on t
  CTE t
      -> Finalize Aggregate
              -> Gather
                      Workers Planned: 1
                      -> Partial Aggregate
                              -> Parallel Seq Scan on flights
(7 rows)
```

다른 예로는 실행 계획에서 SubPlan 노드 아래에 표시된 서브쿼리가 있습니다.

```
=> EXPLAIN (costs off)
SELECT * FROM flights f
WHERE f.scheduled_departure > ( -- SubPlan
SELECT min(f2.scheduled_departure)
```

[246] backend/optimizer/plan/subselect.c

```
FROM flights f2
WHERE f2.aircraft_code = f.aircraft_code
);
                        QUERY PLAN
-----------------------------------------------------------
Seq Scan on flights f
  Filter: (scheduled_departure > (SubPlan 1))
  SubPlan 1
      -> Aggregate
              -> Seq Scan on flights f2
                      Filter: (aircraft_code = f.aircraft_code)
(6 rows)
```

첫 두 개의 행은 주 쿼리의 실행 계획을 나타냅니다. `flights` 테이블은 순차적으로 스캔되고, 각 행은 제공된 필터와 비교됩니다. 필터 조건에는 서브쿼리가 포함되어 있으며, 이 서브쿼리의 실행 계획은 세 번째 행에서 시작됩니다. 따라서 `SubPlan` 노드는 여러 번 실행되며, 이 경우에는 순차 스캔으로 가져온 각 행마다 한 번씩 실행됩니다.

이 계획의 상단 `Seq Scan` 노드는 `SubPlan` 노드가 반환한 데이터에 의존하기 때문에 병렬 실행에 참여할 수 없습니다.

마지막으로, `InitPlan` 노드에 의해 나타나는 또 다른 확장할 수 없는 서브쿼리가 있습니다.

```
=> EXPLAIN (costs off)
SELECT * FROM flights f
WHERE f.scheduled_departure > ( -- SubPlan
  SELECT min(f2.scheduled_departure)
  FROM flights f2
  WHERE EXISTS ( -- InitPlan
      SELECT *
      FROM ticket_flights tf
      WHERE tf.flight_id = f.flight_id
      )
  );
                        QUERY PLAN
-----------------------------------------------------------
Seq Scan on flights f
  Filter: (scheduled_departure > (SubPlan 2))
  SubPlan 2
  -> Finalize Aggregate
      InitPlan 1 (returns $1)
              -> Seq Scan on ticket_flights tf
                      Filter: (flight_id = f.flight_id)
      -> Gather
              Workers Planned: 1
```

```
              Params Evaluated: $1
              -> Partial Aggregate
                      -> Result
                              One-Time Filter: $1
                              -> Parallel Seq Scan on flights f2
(14 rows)
```

`SubPlan` 노드와 달리, `InitPlan`은 한 번만 평가됩니다(이 경우 `SubPlan 2` 노드의 각 실행마다 한 번).

`InitPlan`의 상위 노드는 병렬 실행에 참여할 수 없지만(하지만 `InitPlan` 평가 결과를 받는 노드는 이 예시와 같이 병렬 실행에 참여할 수 있습니다).

임시 테이블은 병렬 스캔을 지원하지 않습니다. 임시 테이블은 생성한 프로세스에 의해서만 접근할 수 있기 때문입니다. 임시 테이블의 페이지는 로컬 버퍼 캐시에서 처리됩니다. 로컬 캐시를 여러 프로세스에서 접근할 수 있게하려면 공유 캐시와 같은 잠금 메커니즘이 필요하며, 이는 다른 장점을 상쇄시킬 수 있습니다.

```
=> CREATE TEMPORARY TABLE flights_tmp AS SELECT * FROM flights;
=> EXPLAIN (costs off)
SELECT count(*) FROM flights_tmp;
      QUERY PLAN
------------------------------
Aggregate
-> Seq Scan on flights_tmp
(2 rows)
```

`PARALLEL RESTRICTED`로 정의된 함수는 계획의 순차적인 부분에서만 허용됩니다. 이러한 함수의 목록은 다음 쿼리를 실행하여 시스템 카탈로그에서 확인할 수 있습니다:

```
SELECT * FROM pg_proc WHERE proparallel = 'r';
```

`PARALLEL RESTRICTED` (물론 `PARALLEL SAFE`도 마찬가지)로 함수를 지정하기 전에 모든 영향과 부과된 제한 사항을 완전히 이해하고 주의 깊게 연구한 경우에만 해당 함수를 사용하십시오.[247]

[247] postgresql.org/docs/14/parallel-safety#PARALLEL-LABELING.html

19 장 인덱스 액세스 방법

19.1 색인 및 확장성

인덱스는 주로 데이터 접근을 가속화하는 목적으로 사용되는 데이터베이스 객체입니다. 이들은 보조 구조물로, 어떤 인덱스든 힙 데이터를 기반으로 삭제하고 다시 생성할 수 있습니다. 데이터 접근 속도 향상 외에도, 인덱스는 일부 무결성 제약 조건을 강제하는 데에도 사용됩니다.

PostgreSQL 코어는 여섯 가지 내장 인덱스 액세스 메소드(인덱스 타입)를 제공합니다:

```
=> SELECT amname FROM pg_am WHERE amtype = 'i';
amname
─────────
btree
hash
gist
gin
spgist
brin
(6 rows)
```

PostgreSQL의 확장성은 코어를 수정하지 않고 새로운 액세스 메소드를 추가할 수 있다는 것을 의미합니다. 이러한 확장(블룸 메소드)은 표준 모듈 집합에 포함되어 있습니다.

다양한 인덱스 타입 간의 모든 차이점에도 불구하고, 모든 인덱스는 결국 키(인덱스화된 열의 값)를 포함하는 힙 튜플과 키를 일치시킵니다. 튜플은 여섯 바이트의 튜플 `ID` 또는 `TID`로 참조됩니다. 키 또는 키에 대한 일부 정보를 알고 있다면, 전체 테이블을 스캔하지 않고도 필요한 데이터를 포함할 가능성이 있는 튜플을 빠르게 읽을 수 있습니다.

PostgreSQL은 새로운 액세스 메소드를 확장으로 추가할 수 있도록 공통 인덱싱 엔진을 구현하여 보장합니다. 이 엔진의 주요 목표는 특정 액세스 메소드에 의해 반환된 TID를 검색하고 처리하는 것입니다:

- 해당 힙 튜플에서 데이터를 읽기
- 특정 스냅샷에 대한 튜플 가시성 확인
- 메소드에 의한 평가가 결정적이지 않은 경우 조건 재확인

인덱싱 엔진은 최적화 단계에서 구축된 계획의 실행에도 참여합니다. 다양한 실행 경로를 평가할 때, 옵티마이저는 모든 가능한 액세스 메소드의 특성을 알아야 합니다: 메소드가 필요한 순서로 데이터를 반환할 수 있는지, 별도의 정렬 단계가 필요한지 등등.

액세스 메소드의 특수성을 알아야 하는 것은 옵티마이저뿐만이 아닙니다. 인덱스 생성은 더 많은 질문을 던지게 됩니다: 액세스 메소드가 다중 열 인덱스를 지원하는지, 이 인덱스가 고유성을 보장할 수 있는지 등.

인덱싱 엔진은 다양한 액세스 메소드를 사용할 수 있도록 허용하며, 지원되기 위해서는 액세스 메소드가 특

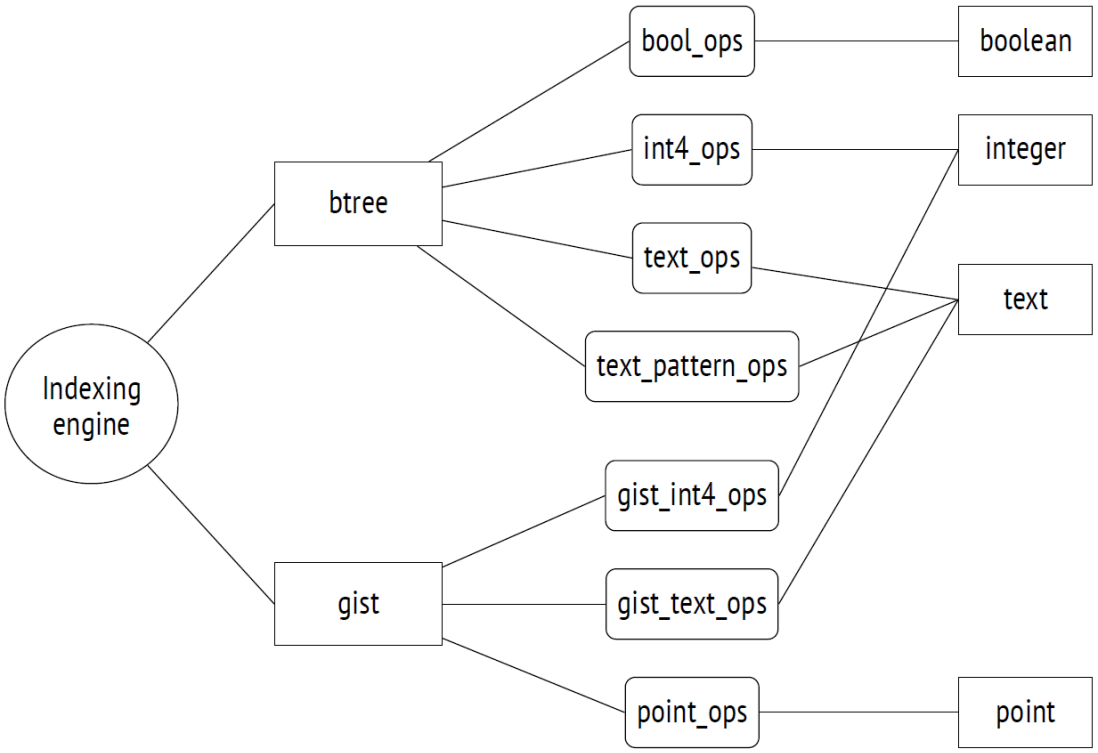
정 인터페이스를 구현하여 기능과 특성을 선언해야 합니다.
액세스 메소드는 다음과 같은 작업을 처리하는 데 사용됩니다:

- 인덱스를 구축하는 알고리즘 구현 및 인덱스 항목 삽입 및 삭제
- 페이지 간에 인덱스 항목 분배 (이후 버퍼 캐시 매니저에서 처리)
- VACUUM 알고리즘 구현
- 올바른 동시 작업을 보장하기 위해 잠금 획득
- WAL 항목 생성
- 키를 사용하여 인덱싱된 데이터 검색
- 인덱스 스캔 비용 추정

확장성은 액세스 메소드가 미리 알지 못하는 새로운 데이터 유형을 추가할 수 있는 능력으로도 나타납니다. 따라서 액세스 메소드는 임의의 데이터 유형을 플러그인하기 위해 고유한 인터페이스를 정의해야 합니다. 특정 액세스 메소드와 함께 새로운 데이터 유형을 사용하려면 해당 인터페이스를 구현해야 합니다. 즉, 인덱스에서 사용할 수 있는 연산자와 필요한 경우 일부 보조 지원 함수를 제공해야 합니다. 이러한 연산자와 함수의 집합을 연산자 클래스라고 합니다.

인덱싱 로직은 액세스 메소드 자체에 부분적으로 구현되지만, 그 중 일부는 연산자 클래스에 외부 위임됩니다. 이 분배는 상당히 임의적입니다. B-트리는 모든 로직이 액세스 메소드에 내장되어 있지만, 다른 메소드는 주로 주요 프레임워크만 제공하고 모든 구현 세부사항은 특정 연산자 클래스의 판단에 따라 달려 있을 수 있습니다. 하나의 데이터 유형은 종종 여러 연산자 클래스에서 지원되며, 사용자는 가장 적합한 동작을 가진 클래스를 선택할 수 있습니다.

전체적인 그림의 일부분을 예로 들면 다음과 같습니다:



19.2 오퍼레이터 객체와 패밀리

오퍼레이터 객체

액세스 메소드 인터페이스[248]는 연산자 클래스[249]에 의해 구현되며, 이는 액세스 메소드가 특정 데이터 유형에 적용하는 연산자와 지원 함수의 집합입니다.

연산자 클래스는 시스템 카탈로그의 pg_opclass 테이블에 저장됩니다. 다음 쿼리는 위의 예시에 대한 완전한 데이터를 반환합니다:

```
=> SELECT amname, opcname, opcintype::regtype
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid;
      amname ┆                opcname ┆ opcintype
─────────────+──────────────────────+──────────────────────────────
       btree ┆              array_ops ┆ anyarray
        hash ┆              array_ops ┆ anyarray
       btree ┆                bit_ops ┆ bit
       btree ┆               bool_ops ┆ boolean
       ...
        brin ┆ pg_lsn_minmax_multi_ops ┆ pg_lsn
        brin ┆        pg_lsn_bloom_ops ┆ pg_lsn
        brin ┆       box_inclusion_ops ┆ box
(177 rows)
```

대부분의 경우, 우리는 연산자 클래스에 대해 알 필요가 없습니다. 우리는 단순히 기본적으로 어떤 연산자 클래스를 사용하는 인덱스를 생성합니다.

예를 들어, 다음은 텍스트 유형을 지원하는 B-트리 연산자 클래스입니다. 클래스 중 하나는 항상 기본값으로 표시됩니다:

```
=> SELECT opcname, opcdefault
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
WHERE amname = 'btree'
AND opcintype = 'text'::regtype;

              opcname ┆ opcdefault
──────────────────────+─────────────
             text_ops ┆ t
          varchar_ops ┆ f
      text_pattern_ops ┆ f
   varchar_pattern_ops ┆ f
(4 rows)
```

[248] postgresql.org/docs/14/xindex.html

[249] postgresql.org/docs/14/indexes-opclass.html

인덱스 생성을 위한 일반적인 명령문은 다음과 같습니다:

```
CREATE INDEX ON aircrafts(model, range);
```

하지만 이것은 단축 표기법으로, 다음 구문으로 확장됩니다:

```
CREATE INDEX ON aircrafts
USING btree -- 기본 액세스 메소드
(
  model text_ops, -- 텍스트에 대한 기본 연산자 클래스
  range int4_ops -- 정수에 대한 기본 연산자 클래스
);
```

다른 유형의 인덱스를 사용하거나 사용자 정의 동작을 달성하려면 원하는 액세스 메소드나 연산자 클래스를 명시적으로 지정해야 합니다.

특정 액세스 메소드와 데이터 유형에 대해 정의된 각 연산자 클래스는 이 유형의 매개변수를 사용하고 이 액세스 메소드의 의미론을 구현하는 연산자 집합을 포함해야 합니다.

예를 들어, btree 액세스 메소드는 다섯 가지 필수 비교 연산자를 정의합니다. 어떤 btree 연산자 클래스든 이 다섯 가지 연산자를 모두 포함해야 합니다:

```
=> SELECT opcname, amopstrategy, amopopr::regoperator
FROM pg_am am
JOIN pg_opfamily opf ON opfmethod = am.oid
JOIN pg_opclass opc ON opcfamily = opf.oid
JOIN pg_amop amop ON amopfamily = opcfamily
WHERE amname = 'btree'
AND opcname IN ('text_ops', 'text_pattern_ops')
AND amoplefttype = 'text'::regtype
AND amoprighttype = 'text'::regtype
ORDER BY opcname, amopstrategy;

         opcname ¦ amopstrategy ¦ amopopr
-------------+------ ---+-----------------
        text_ops ¦           1 ¦ <(text,text)
        text_ops ¦           2 ¦ <=(text,text)
        text_ops ¦           3 ¦ =(text,text)
        text_ops ¦           4 ¦ >=(text,text)
        text_ops ¦           5 ¦ >(text,text)
text_pattern_ops ¦           1 ¦ ~<~(text,text)
text_pattern_ops ¦           2 ¦ ~<=~(text,text)
text_pattern_ops ¦           3 ¦ =(text,text)
text_pattern_ops ¦           4 ¦ ~>=~(text,text)
text_pattern_ops ¦           5 ¦ ~>~(text,text)
```

```
(10 rows)
```

액세스 메소드에 의해 암시되는 연산자의 의미는 `amopstrategy`[250]로 표시되는 전략 번호에 반영됩니다. 예를 들어, `btree`의 전략 1은 작다(`less than`)를 의미하고, 2는 작거나 같다(`less than or equal to`)를 의미합니다. 연산자 자체는 임의의 이름을 가질 수 있습니다.

위의 예시는 두 가지 종류의 연산자를 보여줍니다. 일반 연산자와 물결(~)이 있는 연산자의 차이점은 후자가 `collation`[251]을 고려하지 않고 문자열의 비트별 비교를 수행한다는 것입니다. 그러나 두 가지 버전은 동일한 비교 연산의 논리적인 동작을 구현합니다

.
`text_pattern_ops` 연산자 클래스는 `~~` 연산자(`LIKE` 연산자에 해당)의 지원 제한을 해결하기 위해 설계되었습니다. `C` 이외의 어떤 `collation`을 사용하는 데이터베이스에서는 이 연산자가 텍스트 필드에 일반 인덱스를 사용할 수 없습니다:

```
=> SHOW lc_collate;
lc_collate
─────────────
en_US.UTF-8
(1 row)

=> CREATE INDEX ON tickets(passenger_name);

=> EXPLAIN (costs off)
SELECT * FROM tickets WHERE passenger_name LIKE 'ELENA%';
              QUERY PLAN
──────────────────────────────────────────
Seq Scan on tickets
  Filter: (passenger_name ~~ 'ELENA%'::text)
(2 rows)
```

`text_pattern_ops` 연산자 클래스를 사용하는 인덱스는 다르게 동작합니다:

```
=> CREATE INDEX tickets_passenger_name_pattern_idx
ON tickets(passenger_name text_pattern_ops);
=> EXPLAIN (costs off)
SELECT * FROM tickets WHERE passenger_name LIKE 'ELENA%';
                      QUERY PLAN
──────────────────────────────────────────────────────────
Bitmap Heap Scan on tickets
Filter: (passenger_name ~~ 'ELENA%'::text)
-> Bitmap Index Scan on tickets_passenger_name_pattern_idx
```

---

[250] postgresql.org/docs/14/xindex#XINDEX-STRATEGIES.html

[251] postgresql.org/docs/14/collation.html
postgresql.org/docs/14/indexes-collations.html

```
  Index Cond: ((passenger_name ~>=~ 'ELENA'::text) AND
  (passenger_name ~<~ 'ELENB'::text))
(5 rows)
```

`Index Cond` 조건에서 필터 표현식이 변경된 것에 주목하세요. 검색은 이제 % 이전의 템플릿 접두사만 사용하며, 거짓 양성 결과는 Filter 조건을 기반으로 재확인 과정에서 필터링됩니다. `btree` 액세스 메소드를 위한 연산자 클래스에는 템플릿 비교를 위한 연산자가 제공되지 않으며, 여기서 B-트리를 적용하는 유일한 방법은 비교 연산자를 사용하여 이 조건을 다시 작성하는 것입니다. `text_pattern_ops` 클래스의 연산자는 `collation`을 고려하지 않으므로, 대신 동등한 조건을 사용할 수 있는 기회가 제공됩니다.[252]

인덱스는 다음 두 가지 사전 조건을 충족할 경우 필터 조건에 의한 액세스 속도를 향상시키는 데 사용될 수 있습니다:

1. 조건이 "**인덱스된-열 연산자 표현식**"으로 작성되어 있어야 합니다. (연산자에 대응하는 교환 연산자가 지정된 경우[253], 조건은 "**표현식 연산자 인덱스된-열**"의 형태일 수도 있습니다.)[254]
2. 그리고 연산자가 인덱스 선언에서 인덱스된 열에 지정된 연산자 클래스에 속해야 합니다.

예를 들어, 다음 조회는 인덱스를 사용할 수 있습니다:

```
=> EXPLAIN (costs off)
SELECT * FROM tickets WHERE 'ELENA BELOVA' = passenger_name;
                        QUERY PLAN
-----------------------------------------------------------
Index Scan using tickets_passenger_name_idx on tickets
  Index Cond: (passenger_name = 'ELENA BELOVA'::text)
(2 rows)
```

`Index Cond` 조건에서 인자의 위치에 주목하세요: 실행 단계에서, 인덱스가 적용된 필드는 왼쪽에 있어야 합니다. 인자들이 순서가 바뀌면, 연산자는 교환 가능한 다른 것으로 대체됩니다; 이 특정 경우에는, 동등 관계가 교환 가능하기 때문에 같은 연산자로 대체됩니다.

다음 조회에서는, 조건에서 컬럼 이름이 함수 호출로 대체되기 때문에 일반 인덱스를 사용하는 것이 기술적으로 불가능합니다:

```
=> EXPLAIN (costs off)
SELECT * FROM tickets WHERE initcap(passenger_name) = 'Elena Belova';
                        QUERY PLAN
-----------------------------------------------------------------
Seq Scan on tickets
  Filter: (initcap(passenger_name) = 'Elena Belova'::text)
(2 rows)
```

---

[252] backend/utils/adt/like_support.c

[253] postgresql.org/docs/14/xoper-optimization#id-1.8.3.18.6.html

[254] backend/optimizer/path/indxpath.c, match_clause_to_indexcol function

여기서는 선언 시 컬럼 대신 임의의 표현식이 지정된 **표현식 인덱스**[255]를 사용할 수 있습니다:

```
=> CREATE INDEX ON tickets( (initcap(passenger_name)) );
=> EXPLAIN (costs off)
SELECT * FROM tickets WHERE initcap(passenger_name) = 'Elena Belova';
                      QUERY PLAN
------------------------------------------------------------------
Bitmap Heap Scan on tickets
  Recheck Cond: (initcap(passenger_name) = 'Elena Belova'::text)
  -> Bitmap Index Scan on tickets_initcap_idx
       Index Cond: (initcap(passenger_name) = 'Elena Belova'::text)
(4 rows)
```

인덱스 표현식은 힙 튜플 값에만 의존해야 하며, 데이터베이스에 저장된 다른 데이터나 구성 매개변수(로캘 설정과 같은)에 영향을 받지 않아야 합니다. 다시 말해, 표현식에 함수 호출이 포함되어 있는 경우, 해당 함수는 `IMMUTABLE`[256]이어야 하며 이 변동성 범주를 준수해야 합니다. 그렇지 않으면 인덱스 스캔과 힙 스캔은 동일한 쿼리에 대해 다른 결과를 반환할 수 있습니다.

일반 연산자 외에도 연산자 클래스는 액세스 메소드에서 필요로 하는 지원 함수[257]를 제공할 수 있습니다. 예를 들어, btree 액세스 메소드는 다섯 가지 지원 함수[258]를 정의합니다. 첫 번째 함수(두 값 비교)는 필수이며, 나머지 함수들은 없을 수 있습니다:

```
=> SELECT amprocnum, amproc::regproc
  FROM pg_am am
       JOIN pg_opfamily opf ON opfmethod = am.oid
       JOIN pg_opclass opc ON opcfamily = opf.oid
       JOIN pg_amproc amproc ON amprocfamily = opcfamily
WHERE amname = 'btree'
AND opcname = 'text_ops'
AND amproclefttype = 'text'::regtype
AND amprocrighttype = 'text'::regtype
ORDER BY amprocnum;
 amprocnum ¦ amproc
-----------+--------------------
         1 ¦ bttextcmp
         2 ¦ bttextsortsupport
         4 ¦ btvarstrequalimage
(3 rows)
```

255 postgresql.org/docs/14/indexes-expressional.html

256 postgresql.org/docs/14/xfunc-volatility.html

257 postgresql.org/docs/14/xindex#XINDEX-SUPPORT.html

258 postgresql.org/docs/14/btree-support-funcs.html

연산자 패밀리

각 연산자 클래스는 항상 어떤 **연산자 패밀리**[259]에 속합니다 (`pg_opfamily` 테이블에서 시스템 카탈로그에 나열됨). 패밀리는 동일한 방식으로 유사한 데이터 유형을 처리하는 여러 클래스로 구성될 수 있습니다.

예를 들어, `integer_ops` 패밀리에는 크기가 다른 정수 데이터 유형에 대해 동일한 의미론을 가지지만 크기가 다른 여러 클래스가 포함되어 있습니다:

```
=> SELECT opcname, opcintype::regtype
  FROM pg_am am
      JOIN pg_opfamily opf ON opfmethod = am.oid
      JOIN pg_opclass opc ON opcfamily = opf.oid
WHERE amname = 'btree'
AND opfname = 'integer_ops';
  opcname ¦ opcintype
----------+-----------
 int2_ops ¦ smallint
 int4_ops ¦ integer
 int8_ops ¦ bigint
(3 rows)
```

`datetime_ops` 패밀리는 날짜를 처리하는 연산자 클래스를 포함합니다:

```
=> SELECT opcname, opcintype::regtype
  FROM pg_am am
      JOIN pg_opfamily opf ON opfmethod = am.oid
      JOIN pg_opclass opc ON opcfamily = opf.oid
WHERE amname = 'btree'
AND opfname = 'datetime_ops';
         opcname ¦ opcintype
-----------------+----------------------------
        date_ops ¦ date
 timestamptz_ops ¦ timestamp with time zone
   timestamp_ops ¦ timestamp without time zone
(3 rows)
```

각 연산자 클래스는 하나의 데이터 유형을 지원하지만, 패밀리는 서로 다른 데이터 유형을 위한 연산자 클래스를 포함할 수 있습니다:

```
=> SELECT opcname, amopopr::regoperator
  FROM pg_am am
      JOIN pg_opfamily opf ON opfmethod = am.oid
      JOIN pg_opclass opc ON opcfamily = opf.oid
      JOIN pg_amop amop ON amopfamily = opcfamily
```

[259] postgresql.org/docs/14/xindex#XINDEX-OPFAMILY.html

```
WHERE amname = 'btree'
AND opfname = 'integer_ops'
AND amoplefttype = 'integer'::regtype
AND amopstrategy = 1
ORDER BY opcname;

  opcname | amopopr
----------+---------------------
 int2_ops | <(integer,bigint)
 int2_ops | <(integer,smallint)
 int2_ops | <(integer,integer)
 int4_ops | <(integer,bigint)
 int4_ops | <(integer,smallint)
 int4_ops | <(integer,integer)
 int8_ops | <(integer,bigint)
 int8_ops | <(integer,smallint)
 int8_ops | <(integer,integer)
(9 rows)
```

여러 연산자를 단일 패밀리로 그룹화함으로써, 플래너는 서로 다른 유형의 값이 포함된 조건에 대해 인덱스를 사용할 때 형 변환 없이 작업할 수 있습니다.

19.3 인덱싱 엔진 인터페이스

테이블 액세스 메소드와 마찬가지로, `pg_am` 테이블의 `amhandler` 열에는 인터페이스를 구현하는 함수의 이름이 포함됩니다.[260]

```
=> SELECT amname, amhandler FROM pg_am WHERE amtype = 'i';
 amname | amhandler
--------+-------------
  btree | bthandler
   hash | hashhandler
   gist | gisthandler
    gin | ginhandler
 spgist | spghandler
   brin | brinhandler
(6 rows)
```

이 함수는 인터페이스 구조[261]의 플레이스홀더를 실제 값으로 채웁니다. 일부는 인덱스 액세스와 관련된 개별 작업을 처리하는 함수입니다 (예를 들어, 인덱스 스캔을 수행하고 힙 튜플 `ID`를 반환할 수 있습니다). 다른 일부는 인덱싱 엔진이 알아야 하는 인덱스 메소드 속성입니다.

---

[260] postgresql.org/docs/14/indexam.html

[261] include/access/amapi.h

모든 속성은 다음 세 가지 범주로 구분됩니다:[262]

- 액세스 메소드 속성
- 특정 인덱스의 속성
- 인덱스의 열 수준 속성

액세스 메소드와 인덱스 수준 속성 간의 구분은 미래를 고려하여 제공됩니다: 현재 특정 액세스 메소드를 기반으로 하는 모든 인덱스는 이 두 수준에서 항상 동일한 속성을 갖습니다.

액세스 메서드 속성

다음 다섯 가지 속성은 액세스 메소드 수준에서 정의됩니다 (여기서는 B-트리 메소드를 기준으로 표시됨):

```
=> SELECT a.amname, p.name, pg_indexam_has_property(a.oid, p.name)
FROM pg_am a, unnest(array[
'can_order', 'can_unique', 'can_multi_col',
'can_exclude', 'can_include'
]) p(name)
WHERE a.amname = 'btree';
 amname |          name | pg_indexam_has_property
--------+---------------+-------------------------
  btree |     can_order | t
  btree |    can_unique | t
  btree | can_multi_col | t
  btree |   can_exclude | t
  btree |   can_include | t
(5 rows)
```

정렬된 데이터를 받을 수 있는 능력입니다.[263] 이 속성은 현재 B-트리에서만 지원됩니다.

원하는 순서로 결과를 얻기 위해 테이블을 스캔한 다음 검색된 데이터를 정렬할 수 있습니다:

```
=> EXPLAIN (costs off)
SELECT * FROM seats ORDER BY seat_no;
      QUERY PLAN
-----------------------
Sort
  Sort Key: seat_no
  -> Seq Scan on seats
(3 rows)
```

하지만 이 속성을 지원하는 인덱스가 있다면, 데이터를 한 번에 원하는 순서로 반환할 수 있습니다:

---

[262] backend/utils/adt/amutils.c, indexam_property function

[263] postgresql.org/docs/14/indexes-ordering.html

```
=> EXPLAIN (costs off)
SELECT * FROM seats ORDER BY aircraft_code;
              QUERY PLAN
--------------------------------------
Index Scan using seats_pkey on seats
(1 row)
```

고유 및 기본 키 제약 조건을 지원하는 능력입니다.[264] 이 속성은 B-트리에만 적용됩니다.

고유 또는 기본 키 제약 조건이 선언될 때마다 PostgreSQL은 자동으로 이 제약 조건을 지원하기 위해 고유한 인덱스를 생성합니다.

```
=> INSERT INTO bookings(book_ref, book_date, total_amount)
VALUES ('000004', now(), 100.00);
ERROR: duplicate key value violates unique constraint
"bookings_pkey"
DETAIL: Key (book_ref)=(000004) already exists.
```

그렇지만 무결성 제약 조건을 명시적으로 선언하지 않고 고유 인덱스만 생성한다면, 효과는 동일해 보일 것입니다. 인덱싱된 열은 중복을 허용하지 않을 것입니다. 그렇다면 차이점은 무엇일까요?

무결성 제약 조건은 결코 위반되어서는 안 되는 속성을 정의하는 반면, 인덱스는 그것을 보장하기 위한 메커니즘에 불과합니다. 이론적으로는 다른 수단을 사용하여 제약 조건을 부과할 수 있습니다.

예를 들어, PostgreSQL은 파티션된 테이블에 대한 전역 인덱스를 지원하지 않지만, 해당 테이블에 고유 제약 조건을 생성할 수 있습니다 (파티션 키가 포함된 경우). 이 경우, 각 파티션의 로컬 고유 인덱스에 의해 전역 고유성이 보장됩니다. 각 파티션은 동일한 파티션 키를 가질 수 없기 때문입니다.

다중 열 인덱스를 구축하는 능력입니다.[265]

다중 열 인덱스는 서로 다른 테이블 열에 적용된 여러 조건에 의해 검색 속도를 향상시킬 수 있습니다. 예를 들어, `ticket_flights` 테이블은 복합 기본 키를 가지므로 해당 인덱스는 하나 이상의 열에 구축됩니다:

```
=> \d ticket_flights_pkey
Index "bookings.ticket_flights_pkey"
    Column |          Type | Key? | Definition
-----------+---------------+------+------------
 ticket_no | character(13) |  yes | ticket_no
 flight_id |       integer |  yes | flight_id
primary key, btree, for table "bookings.ticket_flights"
```

항공편 번호와 항공편 `ID`를 사용하여 항공편을 검색하는 경우 인덱스를 사용하여 수행됩니다:

---

[264] postgresql.org/docs/14/indexes-unique.html

[265] postgresql.org/docs/14/indexes-multicolumn.html

```
=> EXPLAIN (costs off)
SELECT * FROM ticket_flights
WHERE ticket_no = '0005432001355'
AND flight_id = 51618;
                        QUERY PLAN
-----------------------------------------------------------
Index Scan using ticket_flights_pkey on ticket_flights
  Index Cond: ((ticket_no = '0005432001355'::bpchar) AND
  (flight_id = 51618))
(3 rows)
```

일반적으로, 다중 열 인덱스는 필터 조건이 일부 열에만 관련된 경우에도 검색 속도를 향상시킬 수 있습니다. B-트리의 경우, 검색이 효율적으로 수행될 것입니다만, 필터 조건이 인덱스 선언에서 처음으로 나타나는 열의 범위를 포함하는 경우입니다:

```
=> EXPLAIN (costs off)
SELECT *
FROM ticket_flights
WHERE ticket_no = '0005432001355';
               QUERY PLAN
---------------------------------------------------------
Index Scan using ticket_flights_pkey on ticket_flights
  Index Cond: (ticket_no = '0005432001355'::bpchar)
(2 rows)
```

다른 모든 경우에는 (예를 들어, 조건이 `flights_id`만 포함하는 경우) 검색은 사실상 초기 열로 제한될 것이며 (쿼리에 해당 조건이 포함된 경우), 다른 조건은 반환된 결과를 필터링하는 데에만 사용됩니다. 다른 유형의 인덱스는 다르게 동작할 수 있습니다.

`EXCLUDE` 제약 조건을 지원하는 능력입니다.[266]

`EXCLUDE` 제약 조건은 테이블 행의 어떤 쌍에 대해 연산자로 정의된 조건이 만족되지 않도록 보장합니다. 이 제약 조건을 부과하기 위해 PostgreSQL은 자동으로 인덱스를 생성합니다. 이 조건에 사용된 연산자를 포함하는 연산자 클래스가 있어야 합니다.

일반적으로 이를 위해 교차 연산자 &&가 사용됩니다. 예를 들어, 동일한 시간에 회의실을 두 번 예약할 수 없다고 명시적으로 선언하거나, 지도에서 건물이 서로 겹치지 않도록 할 수 있습니다. 동등 연산자를 사용하면, 제외 제약 조건은 고유성의 의미를 갖습니다. 즉, 테이블에서 동일한 키 값을 가진 두 개의 행을 가질 수 없습니다.

그럼에도 불구하고, 이는 `UNIQUE` 제약 조건과 동일하지 않습니다. 특히, 제외 제약 조건 키는 외래 키에서 참조할 수 없으며, `ON CONFLICT` 절에서 사용할 수도 없습니다.

---

[266] postgresql.org/docs/14/ddl-constraints#DDL-CONSTRAINTS-EXCLUSION.html

비-키 열을 인덱스에 추가하여 해당 인덱스를 커버링 인덱스로 만드는 능력입니다.

이 속성을 사용하면 고유 인덱스에 추가 열을 확장할 수 있습니다. 이러한 인덱스는 여전히 모든 키 열 값이 고유함을 보장하며, 포함된 열에서의 데이터 검색은 힙 액세스를 수반하지 않습니다.

```
=> CREATE UNIQUE INDEX ON flights(flight_id) INCLUDE (status);
=> EXPLAIN (costs off)
SELECT status FROM flights
WHERE flight_id = 51618;
                        QUERY PLAN
-----------------------------------------------------------
Index Only Scan using flights_flight_id_status_idx on flights
  Index Cond: (flight_id = 51618)
(2 rows)
```

인덱스 레벨 속성

다음은 인덱스와 관련된 속성들입니다 (기존 인덱스에 대한 것으로 표시됨):

```
=> SELECT p.name, pg_index_has_property('seats_pkey', p.name)
FROM unnest(array[
'clusterable', 'index_scan', 'bitmap_scan', 'backward_scan'
]) p(name);
          name | pg_index_has_property
---------------+-----------------------
   clusterable | t
    index_scan | t
   bitmap_scan | t
 backward_scan | t
(4 rows)
```

클러스터링: 인덱스 스캔에 의해 반환된 `ID`의 순서에 따라 힙 튜플을 물리적으로 이동시킬 수 있는 능력을 가리킵니다. 이 속성은 `CLUSTER` 명령이 지원되는지 여부를 나타냅니다.

인덱스 스캔: 인덱스 스캔 지원을 나타냅니다. 이 속성은 액세스 방법이 `TID`를 하나씩 반환할 수 있는지를 의미합니다. 일부 인덱스는 이 기능을 제공하지 않을 수 있습니다.

비트맵 스캔: 비트맵 스캔 지원을 나타냅니다. 이 속성은 액세스 방법이 모든 `TID`의 비트맵을 한 번에 빌드하고 반환할 수 있는지 여부를 정의합니다.

역순 스캔: 인덱스 생성시 지정된 순서와는 반대로 결과를 반환할 수 있는 능력을 가리킵니다. 이 속성은 액세스 방법이 인덱스 스캔을 지원하는 경우에만 의미가 있습니다.

열 레벨 속성

마지막으로, 열 속성을 살펴보겠습니다:

```
=> SELECT p.name,
pg_index_column_has_property('seats_pkey', 1, p.name)
FROM unnest(array[
'asc', 'desc', 'nulls_first', 'nulls_last', 'orderable',
'distance_orderable', 'returnable', 'search_array', 'search_nulls'
]) p(name);
               name | pg_index_column_has_property
--------------------+------------------------------
                asc | t
               desc | f
         nulls_first | f
          nulls_last | t
           orderable | t
  distance_orderable | f
          returnable | t
        search_array | t
        search_nulls | t
(9 rows)
```

ASC, DESC, NULLS FIRST, NULLS LAST: 열 값의 정렬 순서를 지정하는 속성입니다. 이 속성은 열 값이 오름차순 또는 내림차순으로 저장되어야 하는지, NULL 값이 일반 값보다 앞에 나와야 하는지 뒤에 나와야 하는지를 정의합니다. 이러한 속성은 B-트리에만 적용됩니다.

ORDERABLE: ORDER BY 절을 사용하여 열 값을 정렬할 수 있는 능력입니다. 이 속성은 B-트리에만 적용됩니다.

DISTANCE ORDERABLE: 정렬 연산자를 지원하는 능력입니다.[267] 일반적인 인덱싱 연산자가 논리값을 반환하는 반면, 정렬 연산자는 한 인수에서 다른 인수까지의 "거리"를 나타내는 실수값을 반환합니다. 인덱스는 쿼리의 ORDER BY 절에서 지정된 이러한 연산자를 지원합니다.

예를 들어, 정렬 연산자 <->는 지정된 지점에 가장 가까운 공항을 찾을 수 있습니다.

```
=> CREATE INDEX ON airports_data USING gist(coordinates);
=> EXPLAIN (costs off)
SELECT * FROM airports
ORDER BY coordinates <-> point (43.578,57.593)
LIMIT 3;
                      QUERY PLAN
------------------------------------------------------------------
Limit
  -> Index Scan using airports_data_coordinates_idx on airpo...
       Order By: (coordinates <-> '(43.578,57.593)'::point)
(3 rows)
```

[267] postgresql.org/docs/14/xindex#XINDEX-ORDERING-OPS.html

`RETURNABLE`: 데이터를 테이블에 접근하지 않고 반환할 수 있는 능력 (*인덱스 전용 스캔 지원*).

이 속성은 인덱스 구조가 인덱싱된 값을 검색할 수 있는지 여부를 정의합니다. 항상 가능한 것은 아닙니다. 예를 들어, 일부 인덱스는 실제 값 대신 해시 코드를 저장할 수도 있습니다. 이 경우 `CAN INCLUDE` 속성도 사용할 수 없습니다.

`SEARCH ARRAY`: 배열에서 여러 요소를 검색하는 기능.

명시적인 배열 사용이 필요한 유일한 경우는 아닙니다. 예를 들어, 플래너는 `IN` (리스트) 식을 배열 스캔으로 변환합니다:

```
=> EXPLAIN (costs off)
SELECT * FROM bookings
WHERE book_ref IN ('C7C821', 'A5D060', 'DDE1BB');
                QUERY PLAN
------------------------------------------
Index Scan using bookings_pkey on bookings
  Index Cond: (book_ref = ANY
  ('{C7C821,A5D060,DDE1BB}'::bpchar[]))
(3 rows)
```

인덱스 방법이 이러한 연산자를 지원하지 않는 경우, 실행자는 특정 값들을 찾기 위해 여러 번의 반복을 수행해야 할 수 있습니다 (이는 인덱스 스캔을 덜 효율적으로 만들 수 있습니다).

`SEARCH NULLS`: `IS NULL` 및 `IS NOT NULL` 조건을 검색합니다.

`NULL` 값을 인덱싱해야 할까요? 한편으로는 `IS [NOT] NULL`과 같은 조건에 대한 인덱스 스캔을 수행하고, 필터 조건이 제공되지 않은 경우 인덱스를 커버링 인덱스로 사용할 수 있습니다(이 경우 인덱스는 `NULL` 값을 포함하는 모든 힙 튜플의 데이터를 반환해야 합니다). 그러나 다른 한편으로는 NULL 값을 건너뛰면 인덱스 크기를 줄일 수 있습니다.

결정은 접근 방법 개발자의 재량에 달려 있지만, 대부분의 경우 `NULL` 값은 인덱싱됩니다.

인덱스에서 `NULL` 값을 필요로하지 않는다면 필요한 행만 커버하는 부분 인덱스[268]를 구축하여 제외할 수 있습니다. 예를 들어:

```
=> CREATE INDEX ON flights(actual_arrival)
WHERE actual_arrival IS NOT NULL;
=> EXPLAIN (costs off)
SELECT * FROM flights
WHERE actual_arrival = '2017-06-13 10:33:00+03';
                        QUERY PLAN
-----------------------------------------------------------------
```

[268] postgresql.org/docs/14/indexes-partial.html

```
Index Scan using flights_actual_arrival_idx on flights
  Index Cond: (actual_arrival = '2017-06-13 10:33:00+03'::ti...
(2 rows)
```

부분 인덱스는 전체 인덱스보다 작으며, 수정된 행이 인덱싱되는 경우에만 업데이트되므로 때로는 실질적인 성능 향상을 이끌 수 있습니다. 당연히 `NULL` 확인 외에도 `WHERE` 절은 모든 조건(불변 함수와 함께 사용할 수 있는 조건)을 제공할 수 있습니다.

부분 인덱스를 구축하는 능력은 인덱싱 엔진에서 제공되므로 접근 방법에 의존하지 않습니다.

당연히, 인터페이스에는 올바른 결정을 위해 사전에 알아야 하는 인덱스 방법의 속성만 포함됩니다. 예를 들어, 예측 불가능한 잠금 또는 비차단 인덱스 생성(`CONCURRENTLY`)과 같은 기능을 지원하는 속성은 인터페이스를 구현하는 함수의 코드에서 정의됩니다.

20 장 인덱스 스캔

20.1 정기 인덱스 스캔

인덱스가 제공하는 TIDs에 접근하는 기본적인 방법은 두 가지가 있습니다. 첫 번째 방법은 인덱스 스캔을 수행하는 것입니다. 대부분의 인덱스 접근 방식(하지만 모든 것은 아님)은 이 작업을 지원하기 위해 `INDEX SCAN` 속성을 가지고 있습니다. 인덱스 스캔은 계획에서 `Index Scan`[269] 노드로 표현됩니다:

```
=> EXPLAIN SELECT * FROM bookings
WHERE book_ref = '9AC0C6' AND total_amount = 48500.00;
                    QUERY PLAN
-----------------------------------------------------
Index Scan using bookings_pkey on bookings
   (cost=0.43..8.45 rows=1 width=21)
   Index Cond: (book_ref = '9AC0C6'::bpchar)
   Filter: (total_amount = 48500.00)
(4 rows)
```

인덱스 스캔 동안, 접근 방식은 `TID`를 하나씩 반환합니다.[270] `TID`를 받은 후, 인덱싱 엔진은 이 `TID`가 참조하는 힙 페이지에 접근하여 해당 튜플을 가져오고, 가시성 규칙이 충족되면, 이 튜플의 요청된 필드 세트를 반환합니다. 이 과정은 접근 방식이 쿼리와 일치하는 `TID`를 모두 소진할 때까지 계속됩니다.

`Index Cond` 줄에는 인덱스를 사용하여 확인할 수 있는 필터 조건만 포함됩니다. 힙에 대해 다시 확인해야 하는 다른 조건들은 `Filter` 줄에 별도로 나열됩니다.

이 예시에서 보듯이, 인덱스와 힙 접근 작업 모두 공통의 `Index Scan` 노드에 의해 처리되며, 서로 다른 두 노드에 의해 처리되는 것이 아닙니다. 하지만, ID가 사전에 알려진 경우 힙에서 튜플을 가져오는 별도의 `Tid Scan`[271] 노드도 있습니다:

```
=> EXPLAIN SELECT * FROM bookings WHERE ctid = '(0,1)'::tid;
                      QUERY PLAN
-------------------------------------------------------
  Tid Scan on bookings (cost=0.00..4.01 rows=1 width=21)
      TID Cond: (ctid = '(0,1)'::tid)
(2 rows)
```

비용 추정

인덱스 스캔의 비용 추정은 인덱스 접근 작업과 힙 페이지 읽기의 예상 비용을 포함합니다. 분명히, 추정의

[269] backend/executor/nodeIndexscan.c

[270] backend/access/index/indexam.c, index_getnext_tid function

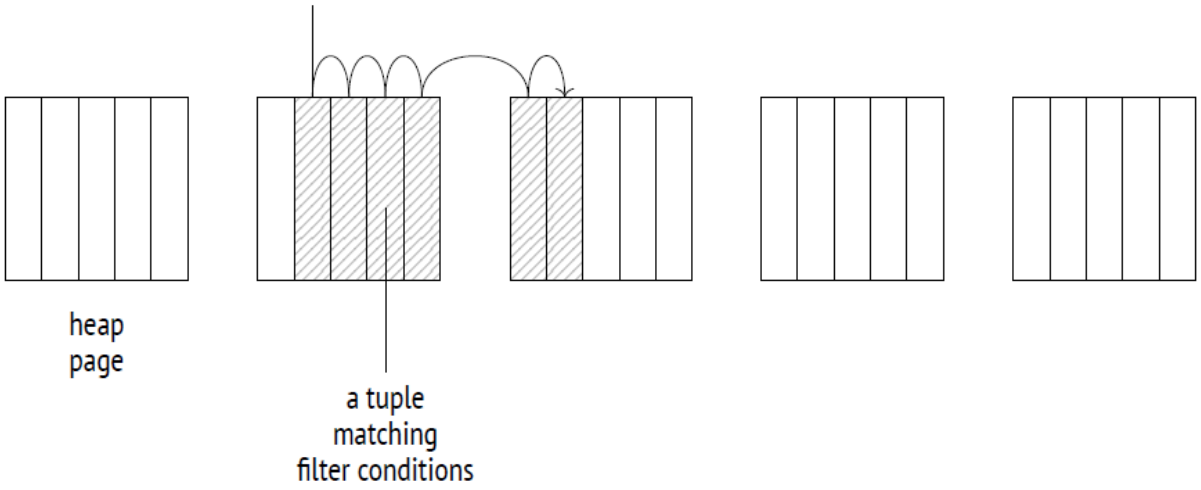
[271] backend/executor/nodeTidscan.c

인덱스 관련 부분은 특정 접근 방식에 전적으로 의존합니다. B-트리의 경우, 비용은 주로 인덱스 페이지를 가져오고 그 항목들을 처리하는 데에 발생합니다. 읽을 페이지와 행의 수는 데이터의 총량과 적용된 필터의 선택성에 의해 결정될 수 있습니다. 인덱스 페이지는 무작위로 접근됩니다(논리적 구조에서 서로 뒤따르는 페이지가 디스크상에서 물리적으로 흩어져 있습니다). 추정은 루트에서 리프 노드로 가는 데에 소요되는 `CPU` 자원과 필요한 모든 표현식을 계산하는 데에 의해 더 증가됩니다.[272]

힙 관련 부분의 추정에는 힙 페이지 접근 비용과 가져온 모든 튜플을 처리하는 데 필요한 `CPU` 시간이 포함됩니다. 중요한 것은 `I/O` 추정이 인덱스 스캔의 선택성과 디스크상의 튜플의 물리적 순서와 접근 방식이 그들의 `ID`를 반환하는 순서 사이의 상관관계에 모두 의존한다는 점입니다.

좋은 시나리오: 높은 상관관계

디스크 상의 튜플의 물리적 순서가 인덱스 내 `TID`의 논리적 순서와 완벽한 상관관계를 가진다면, 각 페이지는 단 한 번만 접근될 것입니다: `Index Scan` 노드는 한 페이지에서 다른 페이지로 **순차적으로** 이동하며, 튜플을 하나씩 읽어갑니다.



PostgreSQL은 상관관계에 대한 통계를 수집합니다:

```
=> SELECT attname, correlation
FROM pg_stats WHERE tablename = 'bookings'
ORDER BY abs(correlation) DESC;

      attname ¦ correlation
--------------+--------------
     book_ref ¦ 1
 total_amount ¦ 0.0026738467
    book_date ¦ 8.02188e-05
(3 rows)
```

[272] backend/utils/adt/selfuncs.c, btcostestimate function
postgresql.org/docs/14/index-cost-estimation.html

상관관계는 해당 절대값이 1에 가까울 경우 높다고 할 수 있습니다(예를 들어, book_ref의 경우와 같이); 0에 가까운 값들은 데이터 분포가 혼돈적임을 나타냅니다.

> 구체적인 이 경우, book_ref 열의 높은 상관관계는 물론이거니와 이 열을 기준으로 상승 순서대로 데이터가 테이블에 로드되었고, 아직 업데이트가 없었기 때문입니다. 이 열에 생성된 인덱스에 대해 CLUSTER 명령을 실행했다면 우리는 같은 상황을 볼 수 있었을 것입니다.
>
> 그러나 완벽한 상관관계가 모든 쿼리가 book_ref 값의 오름차순으로 결과를 반환한다는 것을 보장하지는 않습니다. 먼저, 어떤 행의 업데이트는 결과 튜플을 테이블의 끝으로 이동시킵니다. 둘째, 다른 열을 기반으로 한 인덱스 스캔에 의존하는 계획은 결과를 다른 순서로 반환합니다. 심지어 순차 스캔도 테이블의 시작 부분에서 시작하지 않을 수 있습니다. 그러므로 특정 순서가 필요하다면, ORDER BY 절에서 명시적으로 정의해야 합니다.

```
=> EXPLAIN SELECT * FROM bookings WHERE book_ref < '100000';
                                QUERY PLAN
──────────────────────────────────────────────────────────────────────
  Index Scan using bookings_pkey on bookings
      (cost=0.43..4638.91 rows=132999 width=21)
      Index Cond: (book_ref < '100000'::bpchar)
(3 rows)
```

조건의 선택성은 다음과 같이 추정됩니다:

```
=> SELECT round(132999::numeric/reltuples::numeric, 4)
FROM pg_class WHERE relname = 'bookings';
  round
─────────
  0.0630
(1 row)
```

이 값은 1/16에 가까운데, 이는 book_ref 값이 000000부터 FFFFFF까지 범위를 가지고 있다는 것을 알고 있었다면 추측할 수 있었을 것입니다.

B-트리의 경우, I/O 비용 추정의 인덱스 관련 부분은 필요한 모든 페이지를 읽는 비용을 포함합니다. B-트리에 의해 지원되는 어떤 조건을 만족하는 인덱스 항목은 순서가 있는 리스트로 묶인 페이지에 저장되므로, 읽혀져야 할 인덱스 페이지의 수는 인덱스 크기에 선택성을 곱한 값으로 추정됩니다. 하지만 이러한 페이지들이 물리적으로 정렬되어 있지 않기 때문에, 읽기는 무작위 방식으로 발생합니다.

CPU 자원은 읽혀진 모든 인덱스 항목을 처리하는 데에 소모되며(단일 항목을 처리하는 비용은 cpu_index_tuple_cost(기본값: 0.005) 값으로 추정됩니다) 그리고 각 항목에 대해 조건을 계산하는 데에 소모됩니다(이 경우, 조건은 단일 연산자를 포함하며 그 비용은 cpu_operator_cost(기본값: 0.0025) 값으로 추정됩니다).

테이블 접근은 필요한 페이지 수의 순차적 읽기로 간주됩니다. 완벽한 상관관계의 경우, 힙 튜플은 디스크 상에서 서로를 따라갈 것이므로, 페이지 수는 테이블 크기에 선택성을 곱한 값으로 추정됩니다.

`I/O` 비용은 튜플 처리에 의해 발생하는 비용에 의해 추가로 확장됩니다; 이들은 튜플 당 `cpu_tuple_cost`(기본값: 0.01) 값으로 추정됩니다.

```
=> WITH costs(idx_cost, tbl_cost) AS (
SELECT
(
  SELECT round(
      current_setting('random_page_cost')::real * pages +
      current_setting('cpu_index_tuple_cost')::real * tuples +
      current_setting('cpu_operator_cost')::real * tuples
      )
  FROM (
      SELECT relpages * 0.0630 AS pages, reltuples * 0.0630 AS tuples
      FROM pg_class WHERE relname = 'bookings_pkey'
      ) c
  ),
  (
  SELECT round(
      current_setting('seq_page_cost')::real * pages +
      current_setting('cpu_tuple_cost')::real * tuples
      )
  FROM (
      SELECT relpages * 0.0630 AS pages, reltuples * 0.0630 AS tuples
      FROM pg_class WHERE relname = 'bookings'
      ) c
  )
)
SELECT idx_cost, tbl_cost, idx_cost + tbl_cost AS total
FROM costs;

 idx_cost ¦ tbl_cost ¦ total
----------+----------+-------
     2457 ¦     2177 ¦ 4634
(1 row)
```

이 계산들은 비용 추정 뒤에 있는 논리를 설명해주므로, 결과는 계획자가 제공한 추정치와 일치하게 됩니다. 비록 그것이 근사값일지라도 말이죠. 정확한 값을 얻기 위해서는 다른 세부 사항들을 고려해야 하지만, 우리는 여기서 그것에 대해 논의하지 않을 것입니다.

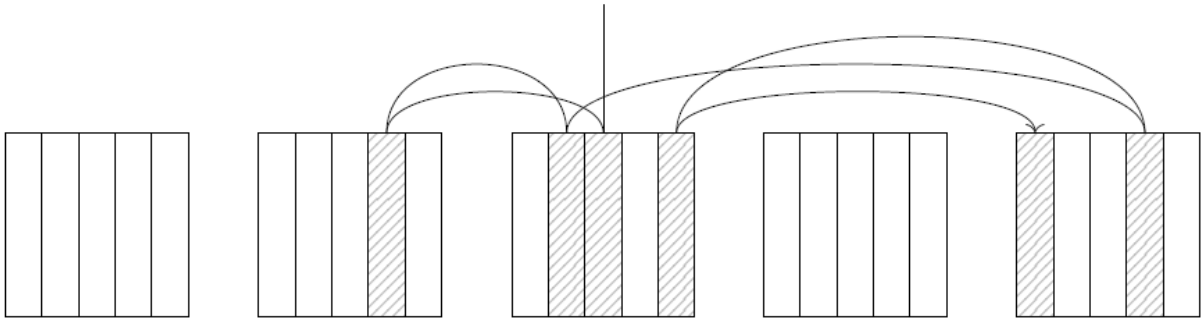
나쁜 시나리오: 낮은 상관관계

상관관계가 낮은 경우 모든 것이 달라집니다. `book_date` 열에 인덱스를 생성합시다. 이 열은 이 인덱스와 거

의 0에 가까운 상관관계를 가지고 있습니다. 그리고 나서 이전 예제와 거의 같은 비율의 행을 선택하는 쿼리를 살펴봅시다. 인덱스 접근은 너무 비싸서, 계획자는 다른 모든 대안들이 명시적으로 금지되었을 때만 이를 선택합니다:

```
=> CREATE INDEX ON bookings(book_date);
=> SET enable_seqscan = off;
=> SET enable_bitmapscan = off;
=> EXPLAIN SELECT * FROM bookings
WHERE book_date < '2016-08-23 12:00:00+03';
                              QUERY PLAN
──────────────────────────────────────────────────────────────────────
  Index Scan using bookings_book_date_idx on bookings
      (cost=0.43..56957.48 rows=132403 width=21)
      Index Cond: (book_date < '2016-08-23 12:00:00+03'::timestamp w...
(3 rows)
```

낮은 상관관계는 접근 방식에 의해 반환된 다음 튜플이 다른 페이지에 위치할 가능성을 높입니다. 따라서, `Index Scan` 노드는 페이지들을 순차적으로 읽는 대신 페이지들 사이를 이동해야 합니다; 최악의 시나리오에서, 페이지 접근 횟수는 가져온 튜플의 수에 도달할 수 있습니다.



그러나 우리는 좋은 시나리오 계산에서 `seq_page_cost`를 `random_page_cost`로, `relpages`를 `reltuples`로 단순히 교체할 수 없습니다. 계획에서 볼 수 있는 비용은 이 방식으로 추정했을 때의 값보다 훨씬 낮습니다.

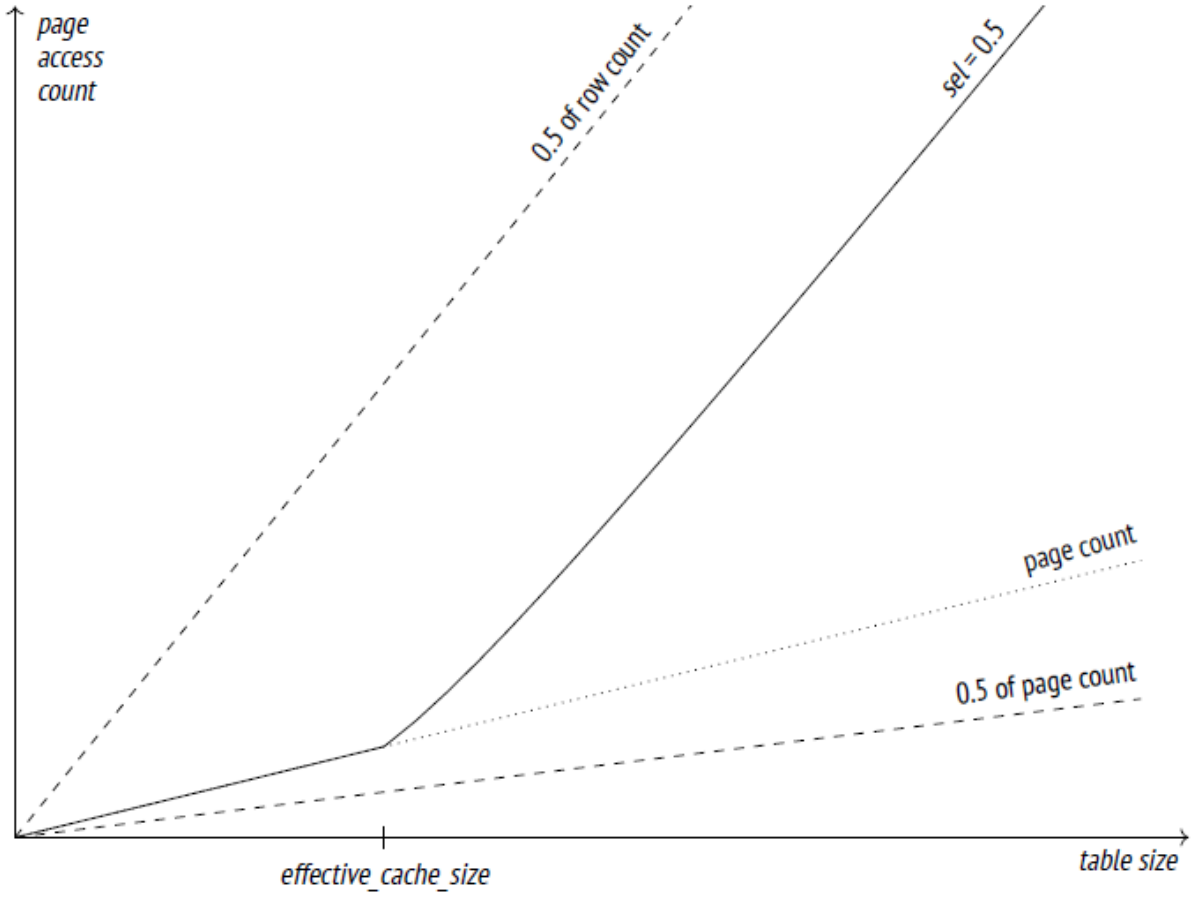
```
=> WITH costs(idx_cost, tbl_cost) AS (
SELECT
( SELECT round(
      current_setting('random_page_cost')::real * pages +
      current_setting('cpu_index_tuple_cost')::real * tuples +
      current_setting('cpu_operator_cost')::real * tuples
      )
  FROM (
      SELECT relpages * 0.0630 AS pages, reltuples * 0.0630 AS tuples
      FROM pg_class WHERE relname = 'bookings_pkey'
      ) c
  ),
```

```
  ( SELECT round(
      current_setting('random_page_cost')::real * tuples +
      current_setting('cpu_tuple_cost')::real * tuples
      )
  FROM (
      SELECT relpages * 0.0630 AS pages, reltuples * 0.0630 AS tuples
      FROM pg_class WHERE relname = 'bookings'
      ) c
  )
)
SELECT idx_cost, tbl_cost, idx_cost + tbl_cost AS total FROM costs;
 idx_cost ¦ tbl_cost ¦ total
----------+----------+--------
     2457 ¦   533330 ¦ 535787
(1 row)
```

이유는 모델이 캐싱을 고려하기 때문입니다. 자주 사용되는 페이지들은 버퍼 캐시(그리고 OS 캐시)에 보관되므로, 캐시 크기가 클수록 필요한 페이지를 캐시에서 찾을 가능성이 더 높아져, 추가 디스크 접근 작업을 피할 수 있습니다. 계획을 위해 캐시 크기는 effective_cache_size(기본값: 4GB) 매개변수에 의해 정의됩니다. 그 값이 작을수록 더 많은 페이지들이 읽혀질 것으로 예상됩니다.

이어지는 그래프는 읽혀야 할 페이지 수의 추정치와 테이블 크기 사이의 의존성을 보여줍니다(선택성이 1/2이고 페이지가 10개의 행을 포함하는 경우).[273] 점선은 가능한 최선의 시나리오에서의 접근 횟수(상관관계가 완벽한 경우 페이지 수의 절반)와 최악의 시나리오에서의 접근 횟수(상관관계가 제로이고 캐시가 없는 경우 행 수의 절반)를 보여줍니다.

[273] backend/optimizer/path/costsize.c, index_pages_fetched function



effective_cache_size 값은 캐싱에 사용될 수 있는 메모리의 총량(PostgreSQL 버퍼 캐시와 OS 캐시를 모두 포함)을 나타낸다고 가정합니다. 그러나 이 매개변수는 오로지 추정 목적으로 사용되며 메모리 할당 자체에는 영향을 주지 않기 때문에, 이 설정을 변경할 때 실제 수치를 고려할 필요는 없습니다.

만약 effective_cache_size를 최소한으로 줄이면, 계획 추정치는 위에서 설명한 캐싱이 없는 경우의 저단 값에 가까워질 것입니다:

```
=> SET effective_cache_size = '8kB';
=> EXPLAIN SELECT * FROM bookings
WHERE book_date < '2016-08-23 12:00:00+03';
                        QUERY PLAN
-----------------------------------------------------------------------------
  Index Scan using bookings_book_date_idx on bookings
      (cost=0.43..532745.48 rows=132403 width=21)
      Index Cond: (book_date < '2016-08-23 12:00:00+03'::timestamp w...
(3 rows)

=> RESET effective_cache_size;
=> RESET enable_seqscan;
=> RESET enable_bitmapscan;
```

계획자는 최악의 경우와 최선의 경우 모두에 대해 테이블 `I/O` 비용을 계산한 후, 실제 상관관계에 기반한 중간값을 취합니다.[274]

따라서, 읽어야 할 행의 일부만 해당한다면 인덱스 스캔은 좋은 선택이 될 수 있습니다. 힙 튜플이 접근 방식이 `ID`를 반환하는 순서와 상관관계가 있다면, 이러한 비율은 상당히 큰 것일 수 있습니다. 그러나 상관관계가 낮은 경우, 선택성이 낮은 쿼리에 대해 인덱스 스캐닝은 훨씬 덜 매력적이 됩니다.

20.2 인덱스 전용 스캔

쿼리에 필요한 모든 힙 데이터를 포함하는 인덱스를 이 특정 쿼리에 대한 커버링 인덱스라고 합니다. 이러한 인덱스가 사용 가능한 경우, 추가적인 테이블 접근을 피할 수 있습니다: TID 대신, 접근 방식이 실제 데이터를 직접 반환할 수 있습니다. 이러한 유형의 인덱스 스캔을 **인덱스 전용 스캔**[275]이라고 합니다. 이는 RETURNABLE 속성을 지원하는 접근 방식에 의해 사용될 수 있습니다.

계획에서 이 작업은 Index Only Scan[276] 노드로 표현됩니다:

```
=> EXPLAIN SELECT book_ref FROM bookings WHERE book_ref < '100000';
                        QUERY PLAN
──────────────────────────────────────────────────────────────────────
  Index Only Scan using bookings_pkey on bookings
      (cost=0.43..3791.91 rows=132999 width=7)
      Index Cond: (book_ref < '100000'::bpchar)
(3 rows)
```

이 이름은 이 노드가 힙에 접근할 필요가 없음을 암시하지만, 실제로는 그렇지 않습니다. PostgreSQL에서 인덱스는 튜플 가시성에 대한 정보를 포함하고 있지 않으므로, 접근 방식은 현재 트랜잭션이 볼 수 없더라도 필터 조건을 만족하는 모든 힙 튜플의 데이터를 반환합니다. 그 후에 인덱싱 엔진에 의해 그들의 가시성이 확인됩니다.

그러나, 이 방법이 각 튜플의 가시성을 확인하기 위해 테이블에 접근해야 한다면, 이는 일반적인 인덱스 스캔과 다를 바가 없을 것입니다. 대신, 이는 베큠 프로세스에 의해 모든 가시 튜플만(즉, 사용된 스냅숏에 관계없이 모든 트랜잭션에 접근 가능한 튜플) 포함된 페이지를 표시하는 테이블을 위한 **가시성 맵**을 사용합니다. 인덱스 접근 방식에 의해 반환된 TID가 그러한 페이지에 속한다면, 그 가시성을 확인할 필요가 없습니다.

인덱스 전용 스캔의 비용 추정은 힙 내의 모든 가시 페이지의 비율에 따라 달라집니다. PostgreSQL은 이러한 통계를 수집합니다:

```
=> SELECT relpages, relallvisible
FROM pg_class WHERE relname = 'bookings';
 relpages ¦ relallvisible
──────────+───────────────
    13447 ¦ 13446
```

---

[274] backend/optimizer/path/costsize.c, cost_index function

[275] postgresql.org/docs/14/indexes-index-only-scans.html

[276] backend/executor/nodeIndexonlyscan.c

```
(1 row)
```

인덱스 전용 스캔의 비용 추정은 일반적인 인덱스 스캔과 다릅니다: 테이블 접근과 관련된 그것의 `I/O` 비용은 가시성 맵에 나타나지 않는 페이지의 비율에 비례하여 취급됩니다. (튜플 처리의 비용 추정은 동일합니다.)

이 특정 예제에서 모든 페이지가 오직 모든 가시 튜플만 포함하기 때문에, 실제로 힙 `I/O`의 비용은 비용 추정에서 제외됩니다:

```
=> WITH costs(idx_cost, tbl_cost) AS (
SELECT
(
  SELECT round(
      current_setting('random_page_cost')::real * pages +
      current_setting('cpu_index_tuple_cost')::real * tuples +
      current_setting('cpu_operator_cost')::real * tuples
  )
  FROM (
      SELECT relpages * 0.0630 AS pages,
             reltuples * 0.0630 AS tuples
      FROM pg_class WHERE relname = 'bookings_pkey'
      ) c
  ) AS idx_cost,
(
  SELECT round(
      (1 - frac_visible) * -- fraction of non-all-visible pages
      current_setting('seq_page_cost')::real * pages +
      current_setting('cpu_tuple_cost')::real * tuples
      )
  FROM (
      SELECT relpages * 0.0630 AS pages,
             reltuples * 0.0630 AS tuples,
             relallvisible::real/relpages::real AS frac_visible
      FROM pg_class WHERE relname = 'bookings'
      ) c
  ) AS tbl_cost
)
SELECT idx_cost, tbl_cost, idx_cost + tbl_cost AS total
FROM costs;
 idx_cost ¦ tbl_cost ¦ total
----------+----------+-------
     2457 ¦     1330 ¦ 3787
(1 row)
```

아직 데이터베이스의 수평선 뒤로 사라지지 않은 어떠한 미청소 변경 사항들도 계획의 추정 비용을 증가시킵니다(그리고 따라서 이 계획을 최적화기에게 덜 매력적으로 만듭니다). `EXPLAIN ANALYZE` 명령어는 실제 힙

접근 횟수를 보여줄 수 있습니다.

새로 생성된 테이블에서 PostgreSQL은 모든 튜플의 가시성을 확인해야 합니다:

```
=> CREATE TEMP TABLE bookings_tmp
WITH (autovacuum_enabled = off) AS
SELECT * FROM bookings
ORDER BY book_ref;

=> ALTER TABLE bookings_tmp ADD PRIMARY KEY(book_ref);
=> ANALYZE bookings_tmp;
=> EXPLAIN (analyze, timing off, summary off)
SELECT book_ref FROM bookings_tmp WHERE book_ref < '100000';
                      QUERY PLAN
-------------------------------------------------------------------
  Index Only Scan using bookings_tmp_pkey on bookings_tmp
      (cost=0.43..4638.91 rows=132999 width=7) (actual rows=132109 l...
      Index Cond: (book_ref < '100000'::bpchar)
      Heap Fetches: 132109
(4 rows)
```

하지만 테이블이 베큠 처리되면, 이러한 확인은 불필요해지며 모든 페이지가 모두 가시적인 상태로 남아 있는 한 수행되지 않습니다.

```
=> VACUUM bookings_tmp;
=> EXPLAIN (analyze, timing off, summary off)
SELECT book_ref FROM bookings_tmp WHERE book_ref < '100000';
                      QUERY PLAN
-------------------------------------------------------------------
  Index Only Scan using bookings_tmp_pkey on bookings_tmp
      (cost=0.43..3787.91 rows=132999 width=7) (actual rows=132109 l...
      Index Cond: (book_ref < '100000'::bpchar)
      Heap Fetches: 0
(4 rows)
```

포함 절이 있는 인덱스

쿼리에서 요구하는 모든 컬럼을 인덱스에 추가하는 것이 항상 가능한 것은 아닙니다:

- 고유 인덱스의 경우, 새로운 컬럼을 추가하면 원래 키 컬럼의 고유성이 손상될 수 있습니다.
- 인덱스 접근 방식이 추가될 컬럼의 데이터 타입에 대한 연산자 클래스를 제공하지 않을 수 있습니다.

이 경우에도, 컬럼을 인덱스 키의 일부로 만들지 않고 인덱스에 포함시킬 수는 있습니다. 물론 포함된 컬럼을 기반으로 한 인덱스 스캔을 수행하는 것은 불가능할 것입니다만, 쿼리가 이러한 컬럼을 참조한다면 인덱스는

커버링 인덱스로 기능할 것입니다.

다음 예제는 자동으로 생성된 기본 키 인덱스를 포함된 컬럼이 있는 다른 인덱스로 대체하는 방법을 보여줍니다:

```
=> CREATE UNIQUE INDEX ON bookings(book_ref) INCLUDE (book_date);

=> BEGIN;
=> ALTER TABLE bookings
DROP CONSTRAINT bookings_pkey CASCADE;
NOTICE: drop cascades to constraint tickets_book_ref_fkey on table
tickets
ALTER TABLE

=> ALTER TABLE bookings ADD CONSTRAINT bookings_pkey PRIMARY KEY
USING INDEX bookings_book_ref_book_date_idx; -- a new index
NOTICE: ALTER TABLE / ADD CONSTRAINT USING INDEX will rename index
"bookings_book_ref_book_date_idx" to "bookings_pkey"
ALTER TABLE
=> ALTER TABLE tickets
ADD FOREIGN KEY (book_ref) REFERENCES bookings(book_ref);
=> COMMIT;

=> EXPLAIN SELECT book_ref, book_date
FROM bookings WHERE book_ref < '100000';
                      QUERY PLAN
------------------------------------------------------------------------
  Index Only Scan using bookings_pkey on bookings (cost=0.43..437...
      Index Cond: (book_ref < '100000'::bpchar)
(2 rows)
```

> 이러한 인덱스들은 종종 커버링이라고 불리지만, 이는 정확한 표현은 아닙니다. 인덱스가 특정 쿼리에 의해 요구되는 모든 컬럼들을 포함하는 컬럼 집합을 가지고 있을 때, 그 인덱스는 커버링으로 간주됩니다. 여기에는 `INCLUDE` 절에 의해 추가된 컬럼이 포함되어 있든, 오직 키 컬럼만 사용되고 있든 상관없습니다. 더욱이, 동일한 인덱스가 한 쿼리에 대해서는 커버링이 될 수 있지만 다른 쿼리에 대해서는 그렇지 않을 수 있습니다.

20.3 비트맵 스캔

인덱스 스캔의 효율성은 한계가 있습니다: 상관관계가 감소함에 따라 힙 페이지에 대한 접근 횟수가 증가하며, 스캔은 순차적이기보다는 랜덤하게 됩니다. 이러한 한계를 극복하기 위해, PostgreSQL은 테이블에 접근하기 전에 모든 `TID`를 가져와서 페이지 번호[277]에 기반하여 오름차순으로 정렬할 수 있습니다. 이것이 바로 비트맵 스캔이 작동하는 방식이며, `TID`를 처리하는 또 다른 일반적인 접근 방식입니다. 이는 **비트맵 스캔** 속

[277] backend/access/index/indexam.c, index_getbitmap function

성을 지원하는 접근 방식에 의해 사용될 수 있습니다.

일반 인덱스 스캔과 달리, 이 작업은 쿼리 계획에서 두 노드로 표현됩니다:

```
=> CREATE INDEX ON bookings(total_amount);
=> EXPLAIN
SELECT * FROM bookings WHERE total_amount = 48500.00;
                      QUERY PLAN
──────────────────────────────────────────────────────────────────────
  Bitmap Heap Scan on bookings (cost=54.63..7040.42 rows=2865 wid...
      Recheck Cond: (total_amount = 48500.00)
      -> Bitmap Index Scan on bookings_total_amount_idx
              (cost=0.00..53.92 rows=2865 width=0)
              Index Cond: (total_amount = 48500.00)
(5 rows)
```

비트맵 인덱스 스캔[278] 노드는 접근 방식으로부터 모든 TID[279]의 비트맵을 가져옵니다.

비트맵은 각각 단일 힙 페이지에 해당하는 별도의 세그먼트로 구성됩니다. 이러한 모든 세그먼트는 페이지의 모든 튜플에 충분한 동일한 크기를 가지고 있으며, 그 안에 존재하는 튜플의 수와 관계없습니다. 이 숫자는 튜플 헤더가 상당히 크기 때문에 제한되며, 표준 크기 페이지는 최대 256개의 튜플을 수용할 수 있으며, 이는 32바이트에 해당합니다.[280]

그런 다음 **비트맵 힙 스캔**[281]이 비트맵 세그먼트를 세그먼트별로 순회하며, 해당 페이지를 읽고, 모두 가시적으로 표시된 모든 튜플을 확인합니다. 따라서, 페이지는 그들의 번호에 기반하여 오름차순으로 읽히며, 각각은 정확히 한 번씩 읽힙니다.

그렇다고 해서 이 과정이 순차 스캔과 같은 것은 아닙니다. 접근된 페이지들이 서로 이어지는 경우는 드뭅니다. 운영 체제에 의해 수행되는 정규 프리페칭은 이 경우 도움이 되지 않으므로, 비트맵 힙 스캔 노드는 `effective_io_concurrency`(기본값: 1) 페이지를 비동기적으로 읽음으로써 자체 프리페칭을 구현하며, 이는 이 작업을 수행하는 유일한 노드입니다. 이 메커니즘은 일부 운영 체제에 의해 구현된 `posix_fadvise` 함수에 의존합니다. 시스템이 이 함수를 지원한다면, 하드웨어 능력에 따라 테이블스페이스 레벨에서 `effective_io_concurrency` 파라미터를 구성하는 것이 의미가 있습니다.

비동기 프리페칭은 다른 일부 내부 프로세스에 의해서도 사용됩니다:

- 힙 행이 삭제될 때 인덱스 페이지에 대해[282]
- 분석(ANALYZE) 동안 힙 페이지에 대해[283]

---

[278] backend/executor/nodeBitmapIndexscan.c

[279] backend/access/index/indexam.c, index_getbitmap function

[280] backend/nodes/tidbitmap.c

[281] backend/executor/nodeBitmapHeapscan.c

[282] backend/access/heap/heapam.c, index_delete_prefetch_buffer function

[283] backend/commands/analyze.c, acquire_sample_rows function

프리페칭 깊이는 `maintenance_io_concurrency`(기본값: 10)에 의해 정의됩니다.

비트맵 정확도

쿼리의 필터 조건을 만족하는 튜플을 포함하는 페이지가 더 많을수록 비트맵의 크기는 커집니다. 이 비트맵은 백엔드의 로컬 메모리에서 구축되며, 그 크기는 `work_mem`(기본값: 4MB) 파라미터에 의해 제한됩니다. 허용되는 최대 크기에 도달하면, 일부 비트맵 세그먼트는 손실이 발생합니다: 손실 세그먼트의 각 비트는 전체 페이지에 해당하며, 세그먼트 자체는 페이지 범위를 포함합니다.[284] 결과적으로, 비트맵의 크기는 그 정확도를 희생하여 작아집니다.

`EXPLAIN ANALYZE` 명령은 구축된 비트맵의 정확도를 보여줍니다:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM bookings WHERE total_amount > 150000.00;
                      QUERY PLAN
-------------------------------------------------------------------
  Bitmap Heap Scan on bookings (actual rows=242691 loops=1)
      Recheck Cond: (total_amount > 150000.00)
      Heap Blocks: exact=13447
      -> Bitmap Index Scan on bookings_total_amount_idx (actual rows...
              Index Cond: (total_amount > 150000.00)
(5 rows)
```

여기에서는 정확한 비트맵을 위한 충분한 메모리가 있습니다.

만약 `work_mem` 값을 줄인다면, 일부 비트맵 세그먼트가 손실되게 됩니다:

```
=> SET work_mem = '512kB';

=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM bookings WHERE total_amount > 150000.00;
                      QUERY PLAN
-------------------------------------------------------------------
  Bitmap Heap Scan on bookings (actual rows=242691 loops=1)
      Recheck Cond: (total_amount > 150000.00)
      Rows Removed by Index Recheck: 1145721
      Heap Blocks: exact=5178 lossy=8269
              -> Bitmap Index Scan on bookings_total_amount_idx (actual rows...
                      Index Cond: (total_amount > 150000.00)
(6 rows)

=> RESET work_mem;
```

[284] backend/nodes/tidbitmap.c, tbm_lossify function

손실 비트맵 세그먼트에 해당하는 힙 페이지를 읽을 때, PostgreSQL은 페이지 내 각 튜플에 대해 필터 조건을 다시 확인해야 합니다. 다시 확인해야 하는 조건은 계획에서 항상 `Recheck Cond`로 표시되며, 이 다시 확인 작업이 수행되지 않더라도 마찬가지입니다. 다시 확인하는 동안 필터링된 튜플의 수는 별도로 표시됩니다(인덱스 재확인에 의해 제거된 행으로).

> 결과 집합의 크기가 너무 크면, 모든 세그먼트가 손실되더라도 비트맵이 `work_mem` 메모리 청크에 맞지 않을 수 있습니다. 그런 경우에는 이 제한이 무시되고, 비트맵은 필요한 만큼의 공간을 차지합니다. PostgreSQL 은 비트맵의 정확도를 더 줄이지 않으며, 그 어떤 세그먼트도 디스크로 플러시하지 않습니다.

비트맵에 관한 연산

만약 쿼리가 각각 별도의 인덱스가 생성된 여러 테이블 컬럼에 조건을 적용한다면, 비트맵 스캔은 이러한 여러 인덱스를 함께 사용할 수 있습니다.[285] 이 모든 인덱스는 실시간으로 자체 비트맵을 구축하며, 이 비트맵들은 논리적 연결(AND로 연결된 표현식의 경우 논리적 곱, OR로 연결된 표현식의 경우 논리적 합)을 사용하여 비트 단위로 결합됩니다. 예를 들어:

```
=> EXPLAIN (costs off)
SELECT * FROM bookings
WHERE book_date < '2016-08-28'
AND total_amount > 250000;

                        QUERY PLAN
-----------------------------------------------------------------------
  Bitmap Heap Scan on bookings
      Recheck Cond: ((total_amount > '250000'::numeric) AND (book_da...
      -> BitmapAnd
              -> Bitmap Index Scan on bookings_total_amount_idx
                      Index Cond: (total_amount > '250000'::numeric)
              -> Bitmap Index Scan on bookings_book_date_idx
                      Index Cond: (book_date < '2016-08-28 00:00:00+03'::tim...
(7 rows)
```

여기서 `BitmapAnd` 노드는 비트단위 `AND` 연산을 사용하여 두 비트맵을 결합합니다.

두 비트맵이 하나로 병합될 때[286], 새 비트맵이 `work_mem` 메모리 청크에 맞는 경우 정확한 세그먼트는 병합되어도 정확하게 유지되지만, 한 쌍의 세그먼트 중 어느 하나라도 손실이 있는 경우 결과 세그먼트도 손실될 것입니다.

---

[285] postgresql.org/docs/14/indexes-ordering.html

[286] backend/nodes/tidbitmap.c, tbm_union & tbm_intersect functions

비용 추정

비트맵 스캔을 사용하는 쿼리를 살펴보겠습니다:

```
=> EXPLAIN
SELECT * FROM bookings WHERE total_amount = 28000.00;
                        QUERY PLAN
-----------------------------------------------------------------------
  Bitmap Heap Scan on bookings (cost=599.48..14444.96 rows=31878 ...
      Recheck Cond: (total_amount = 28000.00)
      -> Bitmap Index Scan on bookings_total_amount_idx
              (cost=0.00..591.51 rows=31878 width=0)
              Index Cond: (total_amount = 28000.00)
(5 rows)
```

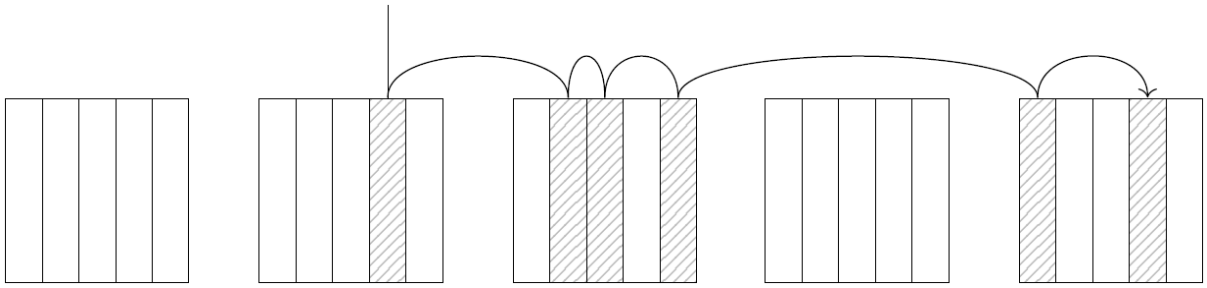
계획자가 사용하는 조건의 대략적인 선택도는:

```
=> SELECT round(31878::numeric/reltuples::numeric, 4)
FROM pg_class WHERE relname = 'bookings';
round
--------
0.0151
(1 row)
```

비트맵 인덱스 스캔 노드의 총 비용 추정 방식은 힙(heap) 접근을 고려하지 않는 일반 인덱스 스캔의 비용 추정 방식과 동일합니다:

```
=> SELECT round(
  current_setting('random_page_cost')::real * pages +
  current_setting('cpu_index_tuple_cost')::real * tuples +
  current_setting('cpu_operator_cost')::real * tuples
)
FROM (
  SELECT relpages * 0.0151 AS pages, reltuples * 0.0151 AS tuples
  FROM pg_class WHERE relname = 'bookings_total_amount_idx'
) c;
 round
-------
  589
(1 row)
```

비트맵 힙 스캔 노드의 `I/O` 비용 추정은 완벽한 상관관계를 가진 일반 인덱스 스캔의 경우와 다릅니다. 비트맵을 사용하면, 같은 페이지로 다시 돌아가지 않고 페이지 번호에 따라 오름차순으로 힙 페이지를 읽을 수 있지만, 필터 조건을 만족하는 튜플들이 더 이상 연속적으로 나타나지 않습니다. 따라서 매우 컴팩트한 순차적 페이지 범위를 읽는 대신, PostgreSQL은 훨씬 더 많은 페이지에 접근할 가능성이 높습니다.



읽어야 할 페이지 수는 다음 공식으로 추정됩니다:[287]

$$\min\left(\frac{2\ \text{relpages}\cdot\text{reltuples}\cdot sel}{2\ \text{relpages}+\text{reltuples}\cdot sel},\text{relpages}\right)$$

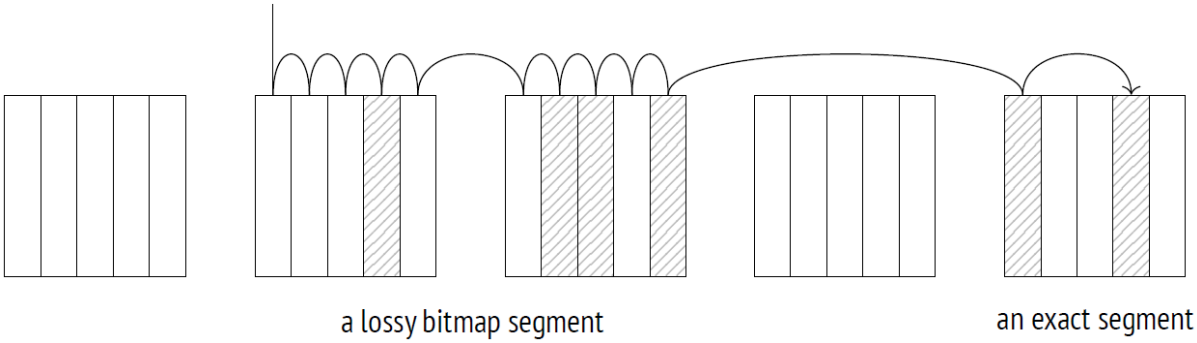
단일 페이지를 읽는 데 드는 추정 비용은 테이블의 총 페이지 수 대비 가져온 페이지의 비율에 따라 `seq_page_cost`와 `random_page_cost` 사이에 있습니다:

```
=> WITH t AS (
SELECT relpages,
least(
  (2 * relpages * reltuples * 0.0151) /
  (2 * relpages + reltuples * 0.0151),
  relpages
) AS pages_fetched,
  round(reltuples * 0.0151) AS tuples_fetched,
  current_setting('random_page_cost')::real AS rnd_cost,
  current_setting('seq_page_cost')::real AS seq_cost
FROM pg_class WHERE relname = 'bookings'
)
SELECT pages_fetched,
  rnd_cost - (rnd_cost - seq_cost) *
  sqrt(pages_fetched / relpages) AS cost_per_page,
  tuples_fetched
FROM t;

 pages_fetched ¦ cost_per_page ¦ tuples_fetched
---------------+---------------+----------------
         13447 ¦             1 ¦ 31878
(1 row)
```

일반적으로, `I/O` 추정값은 가져온 튜플 각각을 처리하는 데 드는 비용만큼 증가합니다. 정확한 비트맵이 사용되는 경우, 튜플의 수는 테이블의 총 튜플 수에 필터 조건의 선택도를 곱한 값으로 추정합니다. 하지만 비트맵 세그먼트 중 손실이 있는 경우, PostgreSQL은 해당 페이지에 접근하여 그 안의 모든 튜플을 다시 확인

[287] backend/optimizer/path/costsize.c, compute_bitmap_pages function

해야 합니다.



따라서, 이 추정값은 손실이 있는 비트맵 세그먼트의 예상 비율(`work_mem`으로 정의된 비트맵 크기 제한과 선택된 행의 총 수를 기반으로 계산할 수 있음)을 고려합니다.[288]

조건 재확인의 총 비용도 추정값을 늘립니다(비트맵의 정확도에 관계없이).

Bitmap Heap Scan 노드의 시작 비용 추정은 **Bitmap Index Scan** 노드의 총 비용을 기반으로 하며, 이는 비트맵 처리 비용으로 확장됩니다.

```
                QUERY PLAN
-----------------------------------------------------
  Bitmap Heap Scan on bookings
      (cost=599.48..14444.96 rows=31878 width=21)
      Recheck Cond: (total_amount = 28000.00)
      -> Bitmap Index Scan on bookings_total_amount_idx
              (cost=0.00..591.51 rows=31878 width=0)
              Index Cond: (total_amount = 28000.00)
(6 rows)
```

여기서 비트맵은 정확하며, 비용은 대략 다음과 같이 추정됩니다:[289]

```
=> WITH t AS (
  SELECT 1 AS cost_per_page,
  13447 AS pages_fetched,
  31878 AS tuples_fetched
),
costs(startup_cost, run_cost) AS (
  SELECT
  ( SELECT round(
              589 /* cost estimation for the child node */ +
              0.1 * current_setting('cpu_operator_cost')::real *
              reltuples * 0.0151
              )
```

[288] backend/optimizer/path/costsize.c, compute_bitmap_pages function

[289] backend/optimizer/path/costsize.c, cost_bitmap_heap_scan function

```
  FROM pg_class WHERE relname = 'bookings_total_amount_idx'
  ),
( SELECT round(
      cost_per_page * pages_fetched +
      current_setting('cpu_tuple_cost')::real * tuples_fetched +
      current_setting('cpu_operator_cost')::real * tuples_fetched
      )
  FROM t
  )
)
SELECT startup_cost, run_cost,
startup_cost + run_cost AS total_cost
FROM costs;

 startup_cost ¦ run_cost ¦ total_cost
--------------+----------+------------
          597 ¦    13845 ¦ 14442
(1 row)
```

쿼리 계획이 여러 비트맵을 결합하는 경우, 별도 인덱스 스캔의 비용 합계에 작은 비용을 더하여(그 비용은 비트맵들을 합치는 데 드는 비용입니다) 증가합니다.[290]

20.4 병렬 인덱스 스캔

모든 인덱스 스캔 모드(일반 인덱스 스캔, 인덱스 전용 스캔, 비트맵 스캔)에는 병렬 계획을 위한 고유한 형태가 있습니다.

병렬 실행의 비용은 순차 실행의 비용과 동일한 방식으로 추정되지만, (병렬 순차 스캔의 경우와 마찬가지로) `CPU` 자원이 모든 병렬 프로세스에 분산되므로 총 비용이 줄어듭니다. `I/O` 비용 구성요소는 페이지 액세스를 순차적으로 수행하도록 프로세스가 동기화되기 때문에 분산되지 않습니다.

이제 병렬 계획의 몇 가지 예를 비용 추정의 상세 내용 없이 보여드리겠습니다.

병렬 인덱스 스캔:

```
=> EXPLAIN SELECT sum(total_amount)
FROM bookings WHERE book_ref < '400000';
                QUERY PLAN
------------------------------------------------------------------------
  Finalize Aggregate (cost=19192.81..19192.82 rows=1 width=32)
      -> Gather (cost=19192.59..19192.80 rows=2 width=32)
              Workers Planned: 2
                      -> Partial Aggregate (cost=18192.59..18192.60 rows=1 widt...
```

[290] backend/optimizer/path/costsize.c, cost_bitmap_and_node & cost_bitmap_or_node functions

```
                              -> Parallel Index Scan using bookings_pkey on bookings
                                    (cost=0.43..17642.82 rows=219907 width=6)
                                    Index Cond: (book_ref < '400000'::bpchar)
(7 rows)
```

B-트리의 병렬 스캔이 진행 중일 때, 현재 인덱스 페이지의 ID는 서버의 공유 메모리에 저장됩니다. 초기 값은 스캔을 시작하는 프로세스에 의해 설정됩니다. 이 프로세스는 루트에서 시작하여 첫 번째 적합한 리프 페이지까지 트리를 통과하고 그 페이지의 ID를 저장합니다. 작업자들은 필요에 따라 후속 인덱스 페이지에 접근하며, 저장된 ID를 교체합니다. 페이지를 가져오면, 작업자는 그 안의 모든 적합한 엔트리를 반복하고 해당 힙 튜플을 읽습니다. 작업자가 쿼리 필터를 충족하는 값의 전체 범위를 읽으면 스캔이 완료됩니다.

병렬 인덱스 전용 스캔:

```
=> EXPLAIN SELECT sum(total_amount)
FROM bookings WHERE total_amount < 50000.00;
                     QUERY PLAN
-----------------------------------------------------------------------
  Finalize Aggregate (cost=23370.60..23370.61 rows=1 width=32)
      -> Gather (cost=23370.38..23370.59 rows=2 width=32)
              Workers Planned: 2
              -> Partial Aggregate (cost=22370.38..22370.39 rows=1 widt...
                      -> Parallel Index Only Scan using bookings_total_amoun...
                            (cost=0.43..21387.27 rows=393244 width=6)
                            Index Cond: (total_amount < 50000.00)
(7 rows)
```

병렬 인덱스 전용 스캔은 모든 가시 페이지의 힙 액세스를 건너뜁니다. 이는 병렬 인덱스 스캔과의 유일한 차이점입니다.

병렬 비트맵 스캔:

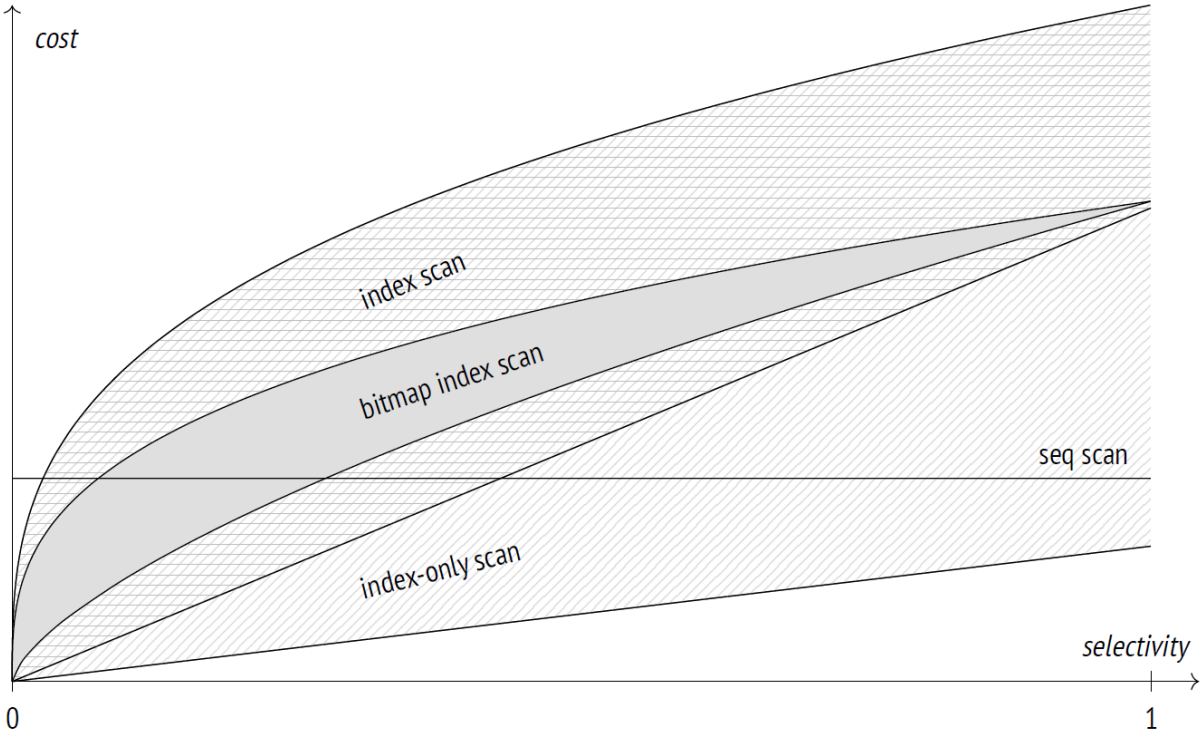
```
=> EXPLAIN SELECT sum(total_amount)
FROM bookings WHERE book_date < '2016-10-01';
                     QUERY PLAN
-----------------------------------------------------------------------
  Finalize Aggregate (cost=21492.21..21492.22 rows=1 width=32)
      -> Gather (cost=21491.99..21492.20 rows=2 width=32)
              Workers Planned: 2
              -> Partial Aggregate (cost=20491.99..20492.00 rows=1 widt...
                      -> Parallel Bitmap Heap Scan on bookings
                            (cost=4891.17..20133.01 rows=143588 width=6)
                            Recheck Cond: (book_date < '2016-10-01 00:00:00+03...
                            -> Bitmap Index Scan on bookings_book_date_idx
                                  (cost=0.00..4805.01 rows=344611 width=0)
                                  Index Cond: (book_date < '2016-10-01 00:00:00+...
```

```
(10 rows)
```

비트맵 스캔은 비트맵이 항상 단일 리더 프로세스에 의해 순차적으로 작성된다는 것을 의미합니다. 이러한 이유로 `Bitmap Index Scan` 노드의 이름에 병렬(Parallel)이라는 단어가 포함되지 않습니다. 비트맵이 준비되면 `Parallel Bitmap Heap Scan` 노드가 병렬 힙 스캔을 시작합니다. 작업자들은 후속 힙 페이지에 접근하고 이를 동시에 처리합니다.

20.5 다양한 액세스 방법 비교

다음 그림은 다양한 액세스 방법의 비용이 필터 조건의 선택도에 어떻게 의존하는지 보여줍니다:



이는 정성적인 다이어그램입니다. 실제 숫자는 특정 테이블과 서버 구성에 따라 달라집니다.

순차 스캔은 선택도에 영향을 받지 않으며, 선택된 행의 일정 비율부터는 일반적으로 다른 방법보다 더 효율적입니다.

인덱스 스캔의 비용은 물리적 튜플 순서와 액세스 방법에 의해 반환되는 튜플의 순서 간의 상관관계에 영향을 받습니다. 상관관계가 완벽하면 인덱스 스캔이 선택된 행의 비율이 다소 높더라도 매우 효율적일 수 있습니다. 하지만, 낮은 상관관계(훨씬 더 일반적)의 경우 순차 스캔보다 비용이 더 많이 들 수 있습니다. 그럼에도 불구하고, 인덱스를 사용하여 단일 행을 선택할 때 인덱스 스캔은 여전히 절대적인 선두주자입니다(일반적으로 고유 인덱스).

적용 가능하다면, 인덱스 전용 스캔은 훌륭한 성능을 보여주고 모든 행이 선택되더라도 순차 스캔을 능가할 수 있습니다. 하지만, 성능은 가시성 맵에 크게 의존하며, 최악의 시나리오에서 인덱스 전용 스캔은 일반 인덱스 스캔으로 저하될 수 있습니다.

비트맵 스캔의 비용은 사용 가능한 메모리 크기에 영향을 받지만, 인덱스 스캔 비용이 상관관계에 의존하는 정도보다 훨씬 적습니다. 상관관계가 낮으면 비트맵 스캔이 훨씬 더 저렴하다는 것이 입증됩니다.

각 액세스 방법에는 완벽한 사용 시나리오가 있습니다. 다른 방법보다 항상 뛰어난 성능을 보이는 방법은 없습니다. 플래너는 각 방법의 효율성을 각 개별 경우에 대해 정확하게 추정하기 위해 광범위한 계산을 수행해야 합니다. 분명히, 이러한 추정의 정확도는 수집된 통계의 정확도에 크게 의존합니다.

21. 중첩 루프

21.1 조인 유형 및 방법

조인은 SQL 언어의 핵심 기능입니다. 이것은 그것의 힘과 유연성의 기초로 작용합니다. 행의 집합(직접 테이블에서 검색되거나 다른 작업의 결과로 받은 것)은 항상 짝지어 조인됩니다.

여러 종류의 조인이 있습니다:

내부Inner 조인. 내부 조인(`INNER JOIN` 또는 단순히 `JOIN`으로 지정됨)은 특정 조인 조건을 만족하는 두 세트의 행 쌍으로 구성됩니다. 조인 조건은 한 세트의 행의 일부 열과 다른 세트의 일부 열을 결합하며, 관련된 모든 열은 **조인 키**를 구성합니다.

조인 조건이 두 세트의 조인 키가 같아야 한다고 요구하는 경우, 이러한 조인은 **동등equi 조인**이라고 하며, 이것이 가장 흔한 조인 유형입니다.

두 세트의 **카테시안 곱**(`CROSS JOIN`)은 이러한 세트의 모든 가능한 행 쌍을 포함합니다. 이것은 참 조건을 가진 내부 조인의 특별한 경우입니다.

외부Outer 조인. 왼쪽 외부 조인(`LEFT OUTER JOIN` 또는 단순히 `LEFT OUTER`으로 지정됨)은 오른쪽 세트에서 일치하는 항목이 없는 왼쪽 세트의 행으로 내부 조인의 결과를 확장합니다(해당 오른쪽 열은 `NULL` 값으로 채워집니다).

오른쪽 외부 조인(`RIGHT JOIN`)도 세트의 순열에 따라 동일하게 적용됩니다.

전체 외부 조인(`FULL JOIN`으로 지정됨)은 왼쪽 및 오른쪽 외부 조인을 포함하며, 일치하는 항목이 발견되지 않은 오른쪽 및 왼쪽 행을 모두 추가합니다.

안티Anti 조인과 세미Semi 조인. 세미 조인은 내부 조인과 많이 닮았지만, 오른쪽 세트에 일치하는 항목이 있는 왼쪽 세트의 행만 포함합니다(한 행은 여러 번 일치해도 한 번만 포함됩니다).

안티 조인은 다른 세트에서 일치하는 항목이 없는 세트의 행을 포함합니다.

`SQL` 언어는 명시적인 세미 조인과 안티 조인을 가지고 있지 않지만, `EXISTS`와 `NOT EXISTS` 같은 술어를 사용하여 같은 결과를 달성할 수 있습니다.

이 모든 조인은 논리적 작업입니다. 예를 들어, 내부 조인은 종종 조인 조건을 만족하지 않는 행이 제거된 카테시안 곱으로 설명됩니다. 그러나 물리적 수준에서 내부 조인은 일반적으로 덜 비싼 수단을 통해 달성됩니다.

PostgreSQL은 여러 조인 방법을 제공합니다:

- 중첩 루프 조인
- 해시 조인

- 병합 조인

조인 방법은 SQL 조인의 논리적 작업을 구현하는 알고리즘입니다. 이 기본 알고리즘은 종종 특정한 조인 유형에 맞춰진 특별한 형태를 가지고 있으며, 그 중 일부만 지원할 수도 있습니다. 예를 들어, 중첩 루프는 내부 조인(계획에서 중첩 루프 노드로 표시됨)과 왼쪽 외부 조인(중첩 루프 왼쪽 조인 노드로 표시됨)을 지원하지만, 전체 조인에는 사용할 수 없습니다.

같은 알고리즘의 일부 변형은 집계와 같은 다른 작업에도 사용될 수 있습니다.

다른 조인 방법은 다른 조건에서 가장 잘 수행됩니다; 가장 비용 효율적인 것을 선택하는 것은 플래너의 임무입니다.

21.2 중첩 루프 조인

중첩 루프 조인의 기본 알고리즘은 다음과 같이 작동합니다. 바깥쪽 루프는 첫 번째 세트(바깥쪽 세트라고 함)의 모든 행을 순회합니다. 이러한 각 행에 대해, 중첩된 루프는 두 번째 세트(내부 세트라고 함)의 행을 순회하며 조인 조건을 만족하는 행을 찾습니다. 각각 찾아진 쌍은 즉시 쿼리 결과의 일부로 반환됩니다.[291]

알고리즘은 바깥쪽 세트의 행이 있는 만큼 많은 번 내부 세트에 접근합니다. 따라서, 중첩 루프 조인의 효율성은 여러 요소에 따라 달라집니다:

- 바깥쪽 행 세트의 카디널리티(요소의 수)
- 내부 세트의 필요한 행을 효율적으로 가져올 수 있는 접근 방법의 가용성
- 내부 세트의 동일한 행에 대한 반복적 접근

카테시안 곱

중첩 루프 조인이 세트의 행 수에 관계없이 카테시안 곱을 찾는 가장 효율적인 방법이라고 말하는 것은 정확하지 않습니다:

```
=> EXPLAIN SELECT * FROM aircrafts_data a1
CROSS JOIN aircrafts_data a2
WHERE a2.range > 5000;

                QUERY PLAN
-----------------------------------------------------
  Nested Loop (cost=0.00..2.78 rows=45 width=144)
    -> Seq Scan on aircrafts_data a1              outer set
            (cost=0.00..1.09 rows=9 width=72)
    -> Materialize (cost=0.00..1.14 rows=5 width=72)
            -> Seq Scan on aircrafts_data a2          inner set
                    (cost=0.00..1.11 rows=5 width=72)
                    Filter: (range > 5000)
(7 rows)
```

[291] backend/executor/nodeNestloop.c

중첩 루프 노드는 위에서 설명한 알고리즘을 사용하여 조인을 수행합니다. 항상 두 개의 자식 노드를 가지며, 계획에서 더 높은 위치에 표시된 노드가 바깥쪽 행 세트에 해당하고, 낮은 쪽 노드가 내부 세트를 나타냅니다.

이 예시에서, 내부 세트는 `Materialize` 노드에 의해 나타납니다.[292] 이 노드는 자식 노드로부터 받은 행들을 미래의 사용을 위해 저장하면서 반환합니다(행들은 총 크기가 `work_mem`(기본값: 4MB)에 도달할 때까지 메모리에 누적됩니다; 그 후 PostgreSQL은 이들을 디스크의 임시 파일로 옮기기 시작합니다). 다시 접근하면, 노드는 자식 노드를 호출하지 않고 누적된 행들을 읽습니다. 따라서, 실행자는 전체 테이블을 다시 스캔하는 것을 피하고 조건을 만족하는 행들만을 읽을 수 있습니다.

유사한 계획은 일반적인 동등 조인을 사용하는 쿼리에 대해서도 구축될 수 있습니다.

```
=> EXPLAIN SELECT *
FROM tickets t
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
WHERE t.ticket_no = '0005432000284';
                        QUERY PLAN
------------------------------------------------------------------
  Nested Loop (cost=0.99..25.05 rows=3 width=136)
      -> Index Scan using tickets_pkey on tickets t
              (cost=0.43..8.45 rows=1 width=104)
              Index Cond: (ticket_no = '0005432000284'::bpchar)
      -> Index Scan using ticket_flights_pkey on ticket_flights tf
              (cost=0.56..16.58 rows=3 width=32)
              Index Cond: (ticket_no = '0005432000284'::bpchar)
(7 rows)
```

두 값의 동등성을 인식한 플래너는 조인 조건 tf.ticket_no = t.ticket_no를 tf.ticket_no = 상수 조건으로 대체하여, 사실상 등가 조인을 카테시안 곱으로 축소합니다.[293]

카디널리티 추정. 카테시안 곱의 카디널리티는 조인된 데이터 세트의 카디널리티의 곱으로 추정됩니다: 3 = 1 × 3.

비용 추정. 조인 작업의 시작 비용은 모든 자식 노드의 시작 비용을 결합합니다.

조인의 전체 비용은 다음 구성 요소를 포함합니다:

- 바깥쪽 세트의 모든 행을 가져오는 비용
- 바깥쪽 세트의 카디널리티 추정이 하나로 동일하므로 내부 세트의 모든 행을 단일 검색하는 비용
- 반환될 각 행을 처리하는 비용

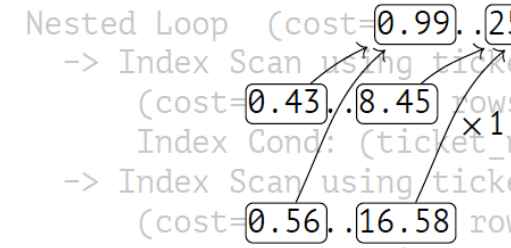
비용 추정을 위한 의존성 그래프가 여기 있습니다:

---

[292] backend/executor/nodeMaterial.c

[293] backend/optimizer/path/equivclass.c

```
                                QUERY PLAN
-----------------------------------------------------------------------
 Nested Loop  (cost=0.99..25.05 rows=3 width=136)
   -> Index Scan using tickets_pkey on tickets t
        (cost=0.43..8.45 rows=1 width=104)
        Index Cond: (ticket_no = '0005432000284'::bpchar)
   -> Index Scan using ticket_flights_pkey on ticket_flights tf
        (cost=0.56..16.58 rows=3 width=32)
        Index Cond: (ticket_no = '0005432000284'::bpchar)
(7 rows)
```



조인의 비용은 다음과 같이 계산됩니다:

```
=> SELECT 0.43 + 0.56 AS startup_cost,
  round((
      8.45 + 16.57 + 3 * current_setting('cpu_tuple_cost')::real
      )::numeric, 2) AS total_cost;

 startup_cost | total_cost
--------------+------------
         0.99 |      25.05
(1 row)
```

앞서 언급한 예시로 돌아가 보겠습니다:

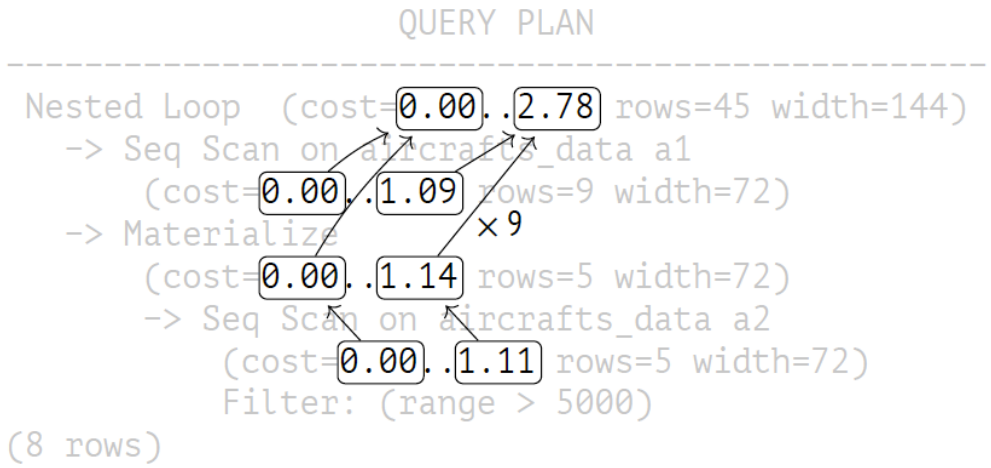
```
=> EXPLAIN SELECT *
FROM aircrafts_data a1
CROSS JOIN aircrafts_data a2
WHERE a2.range > 5000;
                QUERY PLAN
-------------------------------------------------
  Nested Loop (cost=0.00..2.78 rows=45 width=144)
      -> Seq Scan on aircrafts_data a1
              (cost=0.00..1.09 rows=9 width=72)
      -> Materialize (cost=0.00..1.14 rows=5 width=72)
              -> Seq Scan on aircrafts_data a2
                      (cost=0.00..1.11 rows=5 width=72)
                      Filter: (range > 5000)
(7 rows)
```

이제 계획에는 `Materialize` 노드가 포함되어 있습니다. `Materialize`는 자식 노드로부터 받은 행들을 한 번 누적한 후, 이후의 모든 호출에 대해 훨씬 더 빠르게 반환합니다.

일반적으로 조인의 총 비용에는 다음과 같은 비용이 포함됩니다:[294]

- 바깥쪽 세트의 모든 행을 가져오는 비용
- 내부 세트의 모든 행을 처음 가져오는 비용 (이 과정에서 물리화가 수행됩니다)
- (N-1)배의 비용이 드는 내부 세트의 행들을 반복해서 가져오는 비용 (여기서 N은 바깥쪽 세트의 행 수입니다)
- 반환될 각 행을 처리하는 비용

여기의 의존성 그래프는 다음과 같습니다:



이 예제에서는 물리화(materialization)가 반복적인 데이터 가져오기의 비용을 줄여줍니다. 계획에서는 첫 번째 Materialize 호출의 비용이 나타나지만, 그 이후의 모든 호출은 나열되지 않습니다. 여기서 구체적인 계산은 제공하지 않지만,[295] 이 특정 사례에서의 추정치는 0.0125입니다.

따라서, 이 예제에서 수행된 조인의 비용은 다음과 같이 계산됩니다:

```
=> SELECT 0.00 + 0.00 AS startup_cost,
  round((
      1.09 + (1.14 + 8 * 0.0125) + 45 * current_setting('cpu_tuple_cost')::real
      )::numeric, 2) AS total_cost;

 startup_cost ¦ total_cost
--------------+------------
         0.00 ¦ 2.78
(1 row)
```

매개 변수화된 조인

카테시안 곱(Cartesian product)으로 단순화되지 않는 더 일반적인 예를 고려해 보겠습니다:

---

[294] backend/optimizer/path/costsize.c, initial_cost_nestloop andfinal_cost_nestloop function

[295] backend/optimizer/path/costsize.c, cost_rescan function

```
=> CREATE INDEX ON tickets(book_ref);
=> EXPLAIN SELECT *
FROM tickets t
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
WHERE t.book_ref = '03A76D';
                        QUERY PLAN
-----------------------------------------------------------------
  Nested Loop (cost=0.99..45.68 rows=6 width=136)
      -> Index Scan using tickets_book_ref_idx on tickets t
              (cost=0.43..12.46 rows=2 width=104)
              Index Cond: (book_ref = '03A76D'::bpchar)
      -> Index Scan using ticket_flights_pkey on ticket_flights tf
              (cost=0.56..16.58 rows=3 width=32)
              Index Cond: (ticket_no = t.ticket_no)
(7 rows)
```

여기서 중첩 루프(Nested Loop) 노드는 외부 집합(티켓)의 행을 순회하며, 이러한 각 행에 대해 내부 집합(항공편)의 해당 행을 찾습니다. 이때 티켓 번호(t.ticket_no)를 조건에 파라미터로 전달합니다. 내부 노드(인덱스 스캔)가 호출될 때, 이는 ticket_no = 상수 조건을 처리해야 합니다.

카디널리티 추정은 쿼리 플래너가 조인을 수행하기 전에 필터링 조건, 특히 이 경우에는 예약 번호에 의해 만족되는 행의 수를 예측하는 과정입니다. 외부 집합에서 예약 번호에 의해 필터링 되는 행이 2행이고(즉, `rows=2`), 이 각각의 행이 내부 집합에서 평균적으로 3행과 일치한다고 예상한다면(`rows=3`), 이는 조인의 선택성을 계산하는 데 사용됩니다.

조인 선택성은 두 집합의 카테시안 곱에서 조인 후에 남는 부분의 비율을 의미합니다. 조인 키에서 **NULL** 값을 포함하는 두 집합의 행을 제외해야 한다는 것은 명백합니다. 왜냐하면 동등 조건은 이들에 대해 결코 만족될 수 없기 때문입니다.

추정된 카디널리티는 두 집합의 카디널리티(즉, 두 집합의 카디널리티의 곱)에 선택성을 곱한 값과 같습니다.[296]

여기서 첫 번째(외부) 집합의 추정된 카디널리티는 2행입니다. 조인 조건 자체를 제외하고 두 번째(내부) 집합에 적용되는 조건이 없으므로, 두 번째 집합의 카디널리티는 `ticket_flights` 테이블의 카디널리티로 간주됩니다.

조인된 테이블이 외래 키로 연결되어 있기 때문에, 선택성 추정은 자식 테이블의 각 행이 부모 테이블에서 정확히 하나의 일치하는 행을 가진다는 사실에 의존합니다. 따라서 선택성은 외래 키로 참조되는 테이블의 크기의 역수로 취해집니다.[297]

따라서 `ticket_no` 열에 NULL 값이 없는 경우 추정은 다음과 같습니다:

---

[296] backend/optimizer/path/costsize.c, calc_joinrel_size_estimate function

[297] backend/optimizer/path/costsize.c, get_foreign_key_join_selectivity function

```
=> SELECT round(2 * tf.reltuples * (1.0 / t.reltuples)) AS rows
FROM pg_class t, pg_class tf
WHERE t.relname = 'tickets'
AND tf.relname = 'ticket_flights';
  rows
------
  6
(1 row)
```

분명히, 테이블은 외래 키를 사용하지 않고도 조인될 수 있습니다. 그럼 선택성은 특정 조인 조건의 추정된 선택성으로 취해질 것입니다.[298]

이 예제에서 동등 조인(equi-join)의 경우, 값의 균일 분포를 가정하는 선택성 추정의 일반적인 공식은 다음과 같습니다: $\min(1/nd_1, 1/nd_2)$, 여기서 nd_1과 nd_2는 각각 첫 번째 및 두 번째 집합에서 조인 키의 고유 값 수를 나타냅니다.[299]

고유 값에 대한 통계는 티켓 테이블의 티켓 번호가 고유함을 보여줍니다(티켓 번호 열이 기본 키이므로 당연한 결과입니다), 그리고 `ticket_flights`는 각 티켓에 대해 대략 세 개의 일치하는 행을 가지고 있습니다.

```
=> SELECT t.n_distinct, tf.n_distinct
FROM pg_stats t, pg_stats tf
WHERE t.tablename = 'tickets' AND t.attname = 'ticket_no'
AND tf.tablename = 'ticket_flights' AND tf.attname = 'ticket_no';
 n_distinct ¦ n_distinct
------------+-------------
         -1 ¦ -0.30362356
(1 row)
```

결과는 외래 키와의 조인에 대한 추정치와 일치할 것입니다:

```
=> SELECT round(2 * tf.reltuples *
      least(1.0/t.reltuples, 1.0/tf.reltuples/0.30362356)
      ) AS rows
FROM pg_class t, pg_class tf
WHERE t.relname = 'tickets' AND tf.relname = 'ticket_flights';
 rows
------
    6
(1 row)
```

플래너는 가능한 경우 이 기본 추정치를 정제하려고 시도합니다. 현재 히스토그램을 사용할 수 없지만, 두 테이블 모두에서 조인 키에 대해 해당 통계가 수집된 경우 MCV(Most Common Values, 가장 흔한 값) 목록

[298] backend/optimizer/path/clausesel.c, clauselist_selectivity function

[299] backend/utils/adt/selfuncs.c, eqjoinsel function

을 고려합니다.[300] 목록에 나타나는 행의 선택성은 더 정확하게 추정될 수 있으며, 나머지 행들은 균일 분포를 기반으로 한 계산에 의존해야만 합니다.

일반적으로, 외래 키가 정의되어 있을 경우 조인 선택성 추정은 더 정확할 가능성이 높습니다. 이는 특히 복합 조인 키의 경우에 해당되며, 이 경우 선택성이 종종 크게 과소평가됩니다. `EXPLAIN ANALYZE` 명령어를 사용하면, 실제 행 수뿐만 아니라 내부 루프가 실행된 횟수도 볼 수 있습니다.

```
=> EXPLAIN (analyze, timing off, summary off) SELECT *
FROM tickets t
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
WHERE t.book_ref = '03A76D';
                        QUERY PLAN
-----------------------------------------------------------------------
  Nested Loop (cost=0.99..45.68 rows=6 width=136)
      (actual rows=8 loops=1)
      -> Index Scan using tickets_book_ref_idx on tickets t
              (cost=0.43..12.46 rows=2 width=104) (actual rows=2 loops=1)
              Index Cond: (book_ref = '03A76D'::bpchar)
      -> Index Scan using ticket_flights_pkey on ticket_flights tf
              (cost=0.56..16.58 rows=3 width=32) (actual rows=4 loops=2)
              Index Cond: (ticket_no = t.ticket_no)
(8 rows)
```

외부 집합은 두 행을 포함하고 있으며(실제 행 수=2), 추정이 정확했습니다. 따라서 인덱스 스캔 노드는 두 번 실행되었으며(loops=2), 각각 평균 네 행을 선택했습니다(실제 행 수=4). 그러므로 찾은 총 행 수는 실제 행 수=8입니다.

> 각 계획 단계의 실행 시간을 표시하지 않습니다(`TIMING OFF`). 이는 출력이 페이지의 제한된 너비에 맞추기 위함이며, 또한 일부 플랫폼에서는 타이밍이 활성화된 출력이 쿼리 실행을 상당히 느리게 할 수 있기 때문입니다. 하지만 실행 시간을 포함시켰다면, PostgreSQL 은 행 수와 마찬가지로 평균 값을 표시할 것입니다. 총 실행 시간을 얻으려면, 이 값을 반복 횟수(루프)로 곱해야 합니다.

비용 추정. 여기서의 비용 추정 공식은 이전 예제들과 동일합니다.

우리의 쿼리 계획을 다시 생각해 봅시다:

```
=> EXPLAIN SELECT *
FROM tickets t
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
WHERE t.book_ref = '03A76D';
                        QUERY PLAN
-----------------------------------------------------------------------
```

[300] backend/utils/adt/selfuncs.c, eqjoinsel function

```
  Nested Loop (cost=0.99..45.68 rows=6 width=136)
      -> Index Scan using tickets_book_ref_idx on tickets t
              (cost=0.43..12.46 rows=2 width=104)
              Index Cond: (book_ref = '03A76D'::bpchar)
      -> Index Scan using ticket_flights_pkey on ticket_flights tf
              (cost=0.56..16.58 rows=3 width=32)
              Index Cond: (ticket_no = t.ticket_no)
(7 rows)
```

이 경우, 내부 집합의 각 후속 스캔 비용은 첫 번째 스캔의 비용과 동일합니다. 그래서 최종적으로 다음과 같은 수치를 얻게 됩니다:

```
=> SELECT 0.43 + 0.56 AS startup_cost,
  round((
      12.46 + 2 * 16.57 + 6 * current_setting('cpu_tuple_cost')::real
      )::numeric, 2) AS total_cost;
 startup_cost ¦ total_cost
--------------+------------
         0.99 ¦ 45.66
(1 row)
```

행 캐싱(메모화)

내부 집합이 동일한 매개변수 값으로 반복해서 스캔될 경우(따라서 동일한 결과를 제공함), 이 집합의 행을 캐시하는 것이 유리할 수 있습니다.

이러한 캐싱은 `Memoize`[301] 노드에 의해 수행됩니다. `Materialize` 노드와 유사하게, 이는 매개변수화된 조인을 처리하도록 설계되었으며 훨씬 더 복잡한 구현을 가지고 있습니다:

- `Materialize` 노드는 자식 노드에 의해 반환된 모든 행을 단순히 실체화하는 반면, `Memoize`는 다른 매개변수 값에 대해 반환된 행이 별도로 유지되도록 보장합니다.
- 오버플로우가 발생하는 경우, `Materialize` 저장소는 행을 디스크로 옮기기 시작하지만, `Memoize`는 모든 행을 메모리에 유지합니다(그렇지 않으면 캐싱의 의미가 없습니다).

`Memoize`를 사용하는 쿼리의 예는 다음과 같습니다:

```
=> EXPLAIN SELECT *
FROM flights f
JOIN aircrafts_data a ON f.aircraft_code = a.aircraft_code
WHERE f.flight_no = 'PG0003';
                        QUERY PLAN
------------------------------------------------------------------------
  Nested Loop (cost=5.44..387.10 rows=113 width=135)
      -> Bitmap Heap Scan on flights f
```

[301] backend/executor/nodeMemoize.c

```
              (cost=5.30..382.22 rows=113 width=63)
              Recheck Cond: (flight_no = 'PG0003'::bpchar)
                      -> Bitmap Index Scan on flights_flight_no_scheduled_depart...
                              (cost=0.00..5.27 rows=113 width=0)
                              Index Cond: (flight_no = 'PG0003'::bpchar)
  -> Memoize (cost=0.15..0.27 rows=1 width=72)
      Cache Key: f.aircraft_code
      Cache Mode: logical
      -> Index Scan using aircrafts_pkey on aircrafts_data a
              (cost=0.14..0.26 rows=1 width=72)
              Index Cond: (aircraft_code = f.aircraft_code)
(13 rows)
```

캐시된 행을 저장하는 데 사용되는 메모리 청크의 크기는 `work_mem`(기본값: 4MB) × `hash_mem_multiplier`(기본값:1.0)입니다. 두 번째 매개변수의 이름에서 암시하듯, 캐시된 행은 해시 테이블에 저장됩니다(오픈 어드레싱 방식).[302] 해시 키(계획에서는 캐시 키로 표시됨)는 매개변수 값입니다(매개변수가 여러 개인 경우 여러 값일 수 있음).

모든 해시 키는 리스트로 묶여 있으며, 그 끝 중 하나는 차가운 것으로 간주됩니다(오랫동안 사용되지 않은 키를 포함하기 때문에), 반면 다른 하나는 뜨거운 것으로 간주됩니다(최근에 사용된 키를 저장합니다).

`Memoize` 노드에 대한 호출이 전달된 매개변수 값이 이미 캐시된 행에 해당하는 것으로 나타나면, 이 행들은 자식 노드를 확인하지 않고 부모 노드(중첩 루프)로 전달됩니다. 사용된 해시 키는 그 후 리스트의 뜨거운 끝으로 이동됩니다.

캐시에 필요한 행이 없는 경우, `Memoize` 노드는 자식 노드로부터 행을 가져와 캐시하고, 그 위의 노드로 전달합니다. 해당 해시 키도 뜨거워집니다.

새로운 데이터가 캐시되면서, 모든 사용 가능한 메모리를 채울 수 있습니다. 일부 공간을 확보하기 위해, 차가운 키에 해당하는 행은 제거됩니다. 이 제거 알고리즘은 버퍼 캐시에서 사용되는 것과 다르지만 같은 목적을 제공합니다.

일부 매개변수 값은 너무 많은 일치하는 행을 가지고 있어 할당된 메모리 청크에 맞지 않을 수 있으며, 이미 다른 행들이 제거된 상태라도 마찬가지입니다. 이러한 매개변수는 건너뛰어집니다—다음 호출이 여전히 자식 노드로부터 모든 행을 가져와야 하기 때문에 일부 행만 캐시하는 것은 의미가 없습니다.

비용 및 카디널리티 추정. 이러한 계산은 우리가 위에서 이미 본 것과 매우 유사합니다. 우리가 기억해야 할 것은 계획에 표시된 `Memoize` 노드의 비용이 실제 비용과는 아무 관련이 없다는 것입니다: 그것은 단지 자식 노드의 비용에 `cpu_tuple_cost`(기본값: 0.01) 값을 더한 것일 뿐입니다.[303]

우리는 이미 `Materialize` 노드에 대해 비슷한 상황을 마주쳤습니다: 그 비용은 후속 스캔에만 계산되며[304]

[302] include/lib/simplehash.h

[303] backend/optimizer/util/pathnode.c, create_memoize_path function

[304] backend/optimizer/path/costsize.c, cost_memoize_rescan function

계획에는 반영되지 않습니다.

분명히, `Memoize`를 사용하는 것은 그것이 자식 노드보다 저렴할 경우에만 의미가 있습니다. 각 후속 `Memoize` 스캔의 비용은 예상 캐시 액세스 프로필과 캐싱에 사용될 수 있는 메모리 청크의 크기에 따라 달라집니다. 계산된 값은 내부 행 집합의 스캔에 사용될 서로 다른 매개변수 값의 수의 정확한 추정에 크게 의존합니다.[305] 이 숫자에 기반하여, 캐시에 저장될 행의 확률과 캐시에서 제거될 행의 확률을 가중할 수 있습니다. 예상 히트는 추정 비용을 줄이는 반면, 잠재적인 제거는 그것을 증가시킵니다. 여기서는 이러한 계산의 세부 사항을 생략하겠습니다.

쿼리 실행 중에 실제로 무슨 일이 일어나고 있는지 파악하기 위해, 우리는 평소처럼 `EXPLAIN ANALYZE` 명령어를 사용할 것입니다:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM flights f
JOIN aircrafts_data a ON f.aircraft_code = a.aircraft_code
WHERE f.flight_no = 'PG0003';
                      QUERY PLAN
------------------------------------------------------------------------
  Nested Loop (actual rows=113 loops=1)
      -> Bitmap Heap Scan on flights f
              (actual rows=113 loops=1)
              Recheck Cond: (flight_no = 'PG0003'::bpchar)
              Heap Blocks: exact=2
              -> Bitmap Index Scan on flights_flight_no_scheduled_depart...
                      (actual rows=113 loops=1)
                      Index Cond: (flight_no = 'PG0003'::bpchar)
      -> Memoize (actual rows=1 loops=113)
              Cache Key: f.aircraft_code
              Cache Mode: logical
              Hits: 112 Misses: 1 Evictions: 0 Overflows: 0 Memory
              Usage: 1kB
              -> Index Scan using aircrafts_pkey on aircrafts_data a
                      (actual rows=1 loops=1)
                      Index Cond: (aircraft_code = f.aircraft_code)
(16 rows)
```

이 쿼리는 특정 유형의 항공기로 수행되는 같은 경로를 따르는 항공편을 선택하므로, `Memoize` 노드에 대한 모든 호출은 동일한 해시 키를 사용합니다. 첫 번째 행은 테이블에서 가져와야 합니다(미스: 1), 하지만 그 후의 모든 행은 캐시에서 찾을 수 있습니다(히트: 112). 전체 작업은 메모리 1K만을 사용합니다.

다른 두 표시된 값은 0입니다: 이들은 특정 매개변수 세트와 관련된 모든 행을 캐시할 수 없었던 때의 제거 횟수와 캐시 오버플로의 수를 나타냅니다. 큰 수치는 할당된 캐시가 너무 작다는 것을 나타내며, 이는 서로 다른 매개변수 값의 수를 부정확하게 추정한 결과일 수 있습니다. 그러면 `Memoize` 노드의 사용이 상당히 비

[305] backend/utils/adt/selfuncs.c, estimate_num_groups function

싸게 될 수 있습니다. 극단적인 경우에는 `enable_memoize`(기본값: on) 매개변수를 끄는 것으로 계획자가 캐싱을 사용하지 못하게 할 수 있습니다.

외부 조인

중첩 루프 조인(Nested Loop Join)은 왼쪽 외부 조인(Left Outer Join)을 수행하는 데 사용될 수 있습니다:

```
=> EXPLAIN SELECT *
FROM ticket_flights tf
LEFT JOIN boarding_passes bp ON bp.ticket_no = tf.ticket_no
                            AND bp.flight_id = tf.flight_id
WHERE tf.ticket_no = '0005434026720';
                        QUERY PLAN
-----------------------------------------------------------------------
  Nested Loop Left Join (cost=1.12..33.35 rows=3 width=57)
      Join Filter: ((bp.ticket_no = tf.ticket_no) AND (bp.flight_id = tf.flight_id))
      -> Index Scan using ticket_flights_pkey on ticket_flights tf
              (cost=0.56..16.58 rows=3 width=32)
              Index Cond: (ticket_no = '0005434026720'::bpchar)
      -> Materialize (cost=0.56..16.62 rows=3 width=25)
              -> Index Scan using boarding_passes_pkey on boarding_passe...
                      (cost=0.56..16.61 rows=3 width=25)
                      Index Cond: (ticket_no = '0005434026720'::bpchar)
(10 rows)
```

여기서 조인 연산은 `Nested Loop Left Join` 노드로 표현됩니다. 계획자는 필터가 있는 비파라미터화된 조인을 선택했습니다: 이는 내부 행 집합의 동일한 스캔을 수행합니다(그래서 이 집합은 `Materialize` 노드 뒤에 숨겨져 있습니다) 및 필터 조건을 만족하는 행을 반환합니다(`Join Filter`).

외부 조인의 카디널리티는 내부 조인의 것과 마찬가지로 추정되지만, 계산된 추정치는 외부 행 집합의 카디널리티와 비교되며, 더 큰 값이 최종 결과로 취해집니다.[306] 다시 말해, 외부 조인은 행의 수를 줄이지는 않지만 늘릴 수는 있습니다.

비용 추정은 내부 조인의 것과 유사합니다.

또한 계획자가 내부 조인과 외부 조인에 대해 다른 계획을 선택할 수 있다는 점도 기억해야 합니다. 심지어 이 간단한 예에서도, 계획자가 중첩 루프 조인을 사용하도록 강제된다면 다른 `Join Filter`를 가질 것입니다:

```
=> SET enable_mergejoin = off;
=> EXPLAIN SELECT *
FROM ticket_flights tf
JOIN boarding_passes bp ON bp.ticket_no = tf.ticket_no
                       AND bp.flight_id = tf.flight_id
WHERE tf.ticket_no = '0005434026720';
```

[306] backend/optimizer/path/costsize.c, calc_joinrel_size_estimate function

```
                        QUERY PLAN
------------------------------------------------------------------------
  Nested Loop (cost=1.12..33.33 rows=3 width=57)
      Join Filter: (tf.flight_id = bp.flight_id)
      -> Index Scan using ticket_flights_pkey on ticket_flights tf
              (cost=0.56..16.58 rows=3 width=32)
              Index Cond: (ticket_no = '0005434026720'::bpchar)
      -> Materialize (cost=0.56..16.62 rows=3 width=25)
              -> Index Scan using boarding_passes_pkey on boarding_passe...
                      (cost=0.56..16.61 rows=3 width=25)
                      Index Cond: (ticket_no = '0005434026720'::bpchar)
(9 rows)
=> RESET enable_mergejoin;
```

전체 비용에서 약간의 차이가 발생하는 것은 외부 조인이 외부 행 집합에서 일치하는 것이 없는 경우 올바른 결과를 얻기 위해 티켓 번호도 확인해야 하기 때문입니다.

오른쪽 조인은 지원되지 않습니다.[307] 중첩 루프 알고리즘은 내부 및 외부 집합을 다르게 처리하기 때문입니다. 외부 집합은 전체가 스캔되며, 내부 집합의 경우, 인덱스 접근을 통해 조인 조건을 만족하는 행만 읽을 수 있으므로 일부 행은 전혀 건너뛰어질 수 있습니다.

동일한 이유로 전체 조인도 지원되지 않습니다.

안티 조인과 세미 조인

안티 조인(Anti-joins)과 세미 조인(Semi-joins)은 첫 번째(외부) 집합의 각 행에 대해 두 번째(내부) 집합에서 하나의 일치하는 행만 찾으면 충분하다는 점에서 유사합니다.

안티 조인은 첫 번째 집합의 행이 두 번째 집합과 일치하지 않을 경우에만 해당 행을 반환합니다: 실행자가 두 번째 집합에서 첫 번째 일치하는 행을 찾는 즉시 현재 루프를 종료할 수 있습니다: 첫 번째 집합의 해당 행은 결과에서 제외되어야 합니다.

안티 조인은 `NOT EXISTS` 조건을 계산하는 데 사용될 수 있습니다.

예를 들어, 정의되지 않은 기내 구성을 가진 항공기 모델을 찾아봅시다. 해당 계획에는 Nested Loop Anti Join 노드가 포함됩니다:

```
=> EXPLAIN SELECT *
FROM aircrafts a
WHERE NOT EXISTS (
  SELECT * FROM seats s WHERE s.aircraft_code = a.aircraft_code
);
```

[307] backend/optimizer/path/joinpath.c, match_unsorted_outer function

```
                        QUERY PLAN
------------------------------------------------------------------------
  Nested Loop Anti Join (cost=0.28..4.65 rows=1 width=40)
      -> Seq Scan on aircrafts_data ml (cost=0.00..1.09 rows=9 widt...
      -> Index Only Scan using seats_pkey on seats s
              (cost=0.28..5.55 rows=149 width=4)
              Index Cond: (aircraft_code = ml.aircraft_code)
(5 rows)
```

`NOT EXISTS` 조건 없이 다른 쿼리를 사용해도 같은 계획을 가질 것입니다:

```
=> EXPLAIN SELECT a.*
FROM aircrafts a
LEFT JOIN seats s ON a.aircraft_code = s.aircraft_code
WHERE s.aircraft_code IS NULL;
                        QUERY PLAN
------------------------------------------------------------------------
  Nested Loop Anti Join (cost=0.28..4.65 rows=1 width=40)
      -> Seq Scan on aircrafts_data ml (cost=0.00..1.09 rows=9 widt...
      -> Index Only Scan using seats_pkey on seats s
              (cost=0.28..5.55 rows=149 width=4)
              Index Cond: (aircraft_code = ml.aircraft_code)
(5 rows)
```

세미 조인(Semi-join)은 첫 번째 집합의 행 중에서 두 번째 집합과 적어도 하나의 일치하는 행이 있는 경우 해당 행을 반환합니다(다시 말하지만, 다른 일치 항목을 확인할 필요가 없습니다—결과는 이미 알려져 있습니다).

세미 조인은 `EXISTS` 조건을 계산하는 데 사용될 수 있습니다. 기내에 좌석이 설치된 항공기 모델을 찾아봅시다:

```
=> EXPLAIN SELECT *
FROM aircrafts a
WHERE EXISTS (
  SELECT * FROM seats s
      WHERE s.aircraft_code = a.aircraft_code
);
                        QUERY PLAN
------------------------------------------------------------------------
  Nested Loop Semi Join (cost=0.28..6.67 rows=9 width=40)
      -> Seq Scan on aircrafts_data ml (cost=0.00..1.09 rows=9 widt...
      -> Index Only Scan using seats_pkey on seats s
              (cost=0.28..5.55 rows=149 width=4)
              Index Cond: (aircraft_code = ml.aircraft_code)
(5 rows)
```

`Nested Loop Semi Join` 노드는 동일한 이름의 조인 방법을 나타냅니다. 이 계획은 (위의 안티 조인 계획과 마찬가지로) 좌석 테이블의 행 수에 대한 기본적인 추정치(행=149)를 제공하지만, 실제로는 그 중 하나만 검색하는 것으로 충분합니다. 실제 쿼리 실행은 물론 첫 번째 행을 가져온 후에 중단됩니다.

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM aircrafts a
WHERE EXISTS (
  SELECT * FROM seats s
  WHERE s.aircraft_code = a.aircraft_code
);
                      QUERY PLAN
------------------------------------------------------------
  Nested Loop Semi Join (actual rows=9 loops=1)
      -> Seq Scan on aircrafts_data ml (actual rows=9 loops=1)
      -> Index Only Scan using seats_pkey on seats s
              (actual rows=1 loops=9)
              Index Cond: (aircraft_code = ml.aircraft_code)
              Heap Fetches: 0
(6 rows)
```

카디널리티 추정. 세미 조인의 선택도는 일반적인 방식으로 추정되지만, 내부 집합의 카디널리티는 하나로 간주됩니다. 안티 조인의 경우, 추정된 선택도는 부정과 마찬가지로 하나에서 빼집니다.[308]

비용 추정. 안티 조인과 세미 조인의 경우, 비용 추정은 두 번째 집합의 스캔이 첫 번째 일치하는 행을 찾는 즉시 중단된다는 사실을 반영합니다.[309]

비동등 조인

중첩 루프 알고리즘은 어떤 조인 조건을 기반으로 행 집합을 조인할 수 있습니다.

분명히, 내부 집합이 인덱스가 생성된 기본 테이블이고, 조인 조건이 이 인덱스의 연산자 클래스에 속하는 연산자를 사용한다면, 내부 집합에 대한 접근은 상당히 효율적일 수 있습니다. 하지만 어떤 조건에 의해 필터링된 행의 카테시안 곱을 계산하여 조인을 수행할 수도 있습니다—이 경우에는 완전히 임의적일 수 있습니다. 다음과 같은 쿼리에서처럼, 서로 가까이 위치한 공항의 쌍을 선택합니다:

```
=> CREATE EXTENSION earthdistance CASCADE;
=> EXPLAIN (costs off) SELECT *
FROM airports a1
JOIN airports a2 ON a1.airport_code != a2.airport_code
                AND a1.coordinates <@> a2.coordinates < 100;
                      QUERY PLAN
------------------------------------------------------------------------
```

[308] backend/optimizer/path/costsize.c, calc_joinrel_size_estimate function

[309] backend/optimizer/path/costsize.c, final_cost_nestloop function

```
  Nested Loop
      Join Filter: ((ml.airport_code <> ml_1.airport_code) AND
      ((ml.coordinates <@> ml_1.coordinates) < '100'::double precisi...
      -> Seq Scan on airports_data ml
      -> Materialize
              -> Seq Scan on airports_data ml_1
(6 rows)
```

병렬 모드

중첩 루프 조인은 병렬 계획 실행에 참여할 수 있습니다.[310]

외부 집합만 병렬로 처리될 수 있는데, 이는 여러 작업자가 동시에 스캔할 수 있기 때문입니다. 외부 행을 가져온 각 작업자는 그 다음 내부 집합에서 일치하는 행을 찾아야 하는데, 이 과정은 순차적으로 이루어집니다.

아래에 표시된 쿼리는 여러 조인을 포함하고 있으며, 특정 항공편에 대한 티켓을 가진 승객을 검색합니다:

```
=> EXPLAIN (costs off) SELECT t.passenger_name
FROM tickets t
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
JOIN flights f ON f.flight_id = tf.flight_id
WHERE f.flight_id = 12345;
                              QUERY PLAN
----------------------------------------------------------------------
  Nested Loop
      -> Index Only Scan using flights_flight_id_status_idx on fligh...
              Index Cond: (flight_id = 12345)
      -> Gather
              Workers Planned: 2
              -> Nested Loop
                      -> Parallel Seq Scan on ticket_flights tf
                              Filter: (flight_id = 12345)
                      -> Index Scan using tickets_pkey on tickets t
                              Index Cond: (ticket_no = tf.ticket_no)
(10 rows)
```

상위 수준에서 중첩 루프 조인은 순차적으로 수행됩니다. 외부 집합은 고유 키로 가져온 `flights` 테이블의 단일 행으로 구성되므로, 내부 행의 수가 많더라도 중첩 루프의 사용이 정당화됩니다.

내부 집합은 병렬 계획을 사용하여 검색됩니다. 각 작업자는 `ticket_flights` 테이블의 자신의 몫의 행을 스캔하고 중첩 루프 알고리즘을 사용하여 이를 티켓과 조인합니다.

[310] backend/optimizer/path/joinpath.c, consider_parallel_nestloop function

22. 해싱Hashing

22.1 해쉬 조인

1-패스 해쉬 조인

해시 조인은 사전에 구축된 해시 테이블을 사용하여 일치하는 행을 검색합니다. 다음은 그러한 조인이 있는 계획의 예입니다:

```
=> EXPLAIN (costs off) SELECT *
FROM tickets t
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no;
              QUERY PLAN
----------------------------------------
  Hash Join
      Hash Cond: (tf.ticket_no = t.ticket_no)
      -> Seq Scan on ticket_flights tf
      -> Hash
              -> Seq Scan on tickets t
(5 rows)
```

첫 번째 단계에서, 해시 조인 노드[311]는 해시 노드[312]를 호출하는데, 이는 자식 노드로부터 내부 행의 전체 집합을 가져와 해시 테이블에 배치합니다.

해시 키와 값의 쌍을 저장함으로써, 해시 테이블은 키에 의한 값에 대한 빠른 접근을 가능하게 합니다; 해시 키가 제한된 수의 버킷 사이에 어느 정도 균등하게 분포되어 있기 때문에, 검색 시간은 해시 테이블의 크기에 따라 달라지지 않습니다. 주어진 키가 가는 버킷은 해시 키의 해시 함수에 의해 결정됩니다; 버킷의 수가 항상 2의 거듭제곱이기 때문에, 계산된 값의 필요한 수의 비트를 취하는 것으로 충분합니다.

버퍼 캐시와 마찬가지로, 이 구현은 동적으로 확장 가능한 해시 테이블을 사용하여 체이닝을 통해 해시 충돌을 해결합니다.[313]

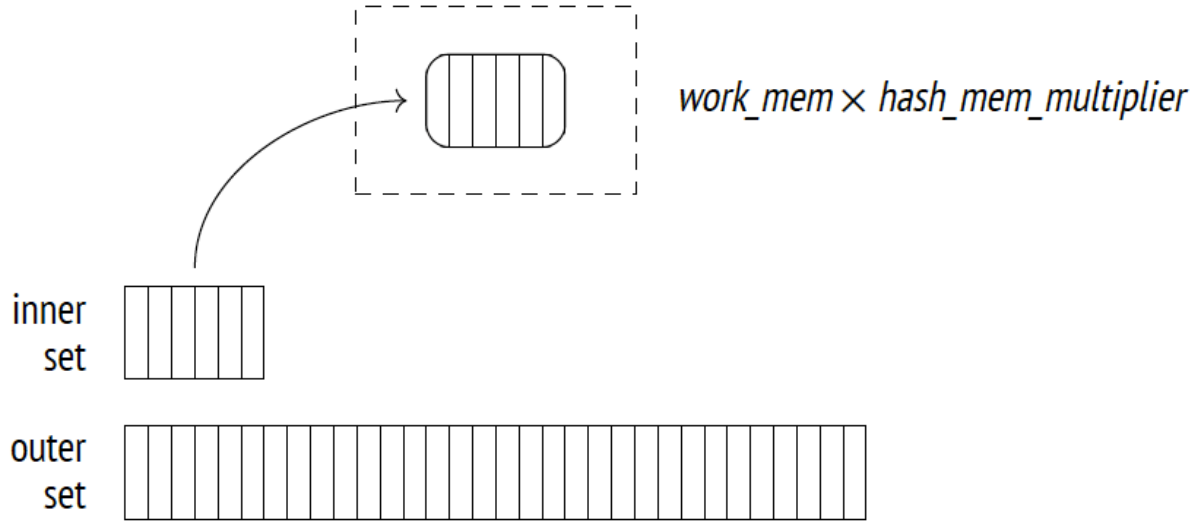
조인 연산의 **첫 단계**에서 내부 집합이 스캔되고, 각 행에 대해 해시 함수가 계산됩니다. 조인 조건(해시 조건)에서 참조된 열은 해시 키로 사용되며, 해시 테이블 자체는 내부 집합의 모든 조회된 필드를 저장합니다.

해시 조인은 전체 해시 테이블을 RAM에 수용할 수 있을 때 가장 효율적이며, 이 경우 실행기는 데이터를 한 번에 처리할 수 있습니다. 이 목적으로 할당된 메모리 덩어리의 크기는 work_mem(기본값: 4MB) × hash_mem_multiplier(기본값: 1.0) 값으로 제한됩니다.

311 backend/executor/nodeHashjoin.c

312 backend/executor/nodeHash.c

313 backend/utils/hash/dynahash.c



쿼리의 메모리 사용에 대한 통계를 보기 위해 `EXPLAIN ANALYZE`를 실행해 봅시다:

```
=> SET work_mem = '256MB';
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM bookings b
JOIN tickets t ON b.book_ref = t.book_ref;
                    QUERY PLAN
------------------------------------------------------------------
  Hash Join (actual rows=2949857 loops=1)
      Hash Cond: (t.book_ref = b.book_ref)
      -> Seq Scan on tickets t (actual rows=2949857 loops=1)
      -> Hash (actual rows=2111110 loops=1)
              Buckets: 4194304 Batches: 1 Memory Usage: 145986kB
              -> Seq Scan on bookings b (actual rows=2111110 loops=1)
(6 rows)
```

중첩 루프 조인이 내부 집합과 외부 집합을 다르게 취급하는 것과 달리, 해시 조인은 이들을 서로 바꿀 수 있습니다. 일반적으로 더 작은 집합이 내부 집합으로 사용되며, 이는 더 작은 해시 테이블을 결과로 합니다.

이 예에서, 전체 테이블이 할당된 캐시에 맞습니다: 약 143MB(메모리 사용량)를 차지하며 4M = 2^{22} 버킷을 포함합니다. 그래서 조인은 한 번의 패스(배치)에서 수행됩니다.

그러나 쿼리가 하나의 열만을 참조한다면, 해시 테이블은 111MB에 맞을 것입니다:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT b.book_ref
FROM bookings b
JOIN tickets t ON b.book_ref = t.book_ref;
              QUERY PLAN
------------------------------------------------------------------
  Hash Join (actual rows=2949857 loops=1)
```

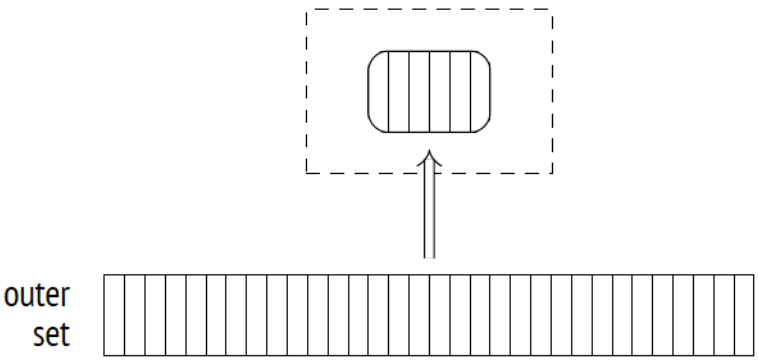
```
      Hash Cond: (t.book_ref = b.book_ref)
      -> Index Only Scan using tickets_book_ref_idx on tickets t
              (actual rows=2949857 loops=1)
              Heap Fetches: 0
      -> Hash (actual rows=2111110 loops=1)
              Buckets: 4194304 Batches: 1 Memory Usage: 113172kB
              -> Seq Scan on bookings b (actual rows=2111110 loops=1)
(8 rows)
=> RESET work_mem;
```

이것은 쿼리에서 불필요한 필드를 참조하는 것을 피해야 하는 또 다른 이유입니다(예를 들어, 별표를 사용하는 경우 발생할 수 있습니다).

선택된 버킷의 수는 해시 테이블이 데이터로 완전히 채워졌을 때 평균적으로 각 버킷이 하나의 행만을 가지도록 보장해야 합니다. 높은 밀도는 해시 충돌률을 증가시켜 검색 효율을 떨어뜨리며, 덜 조밀한 해시 테이블은 너무 많은 메모리를 차지할 것입니다. 버킷의 예상 수는 가장 가까운 2의 거듭제곱까지 증가됩니다.[314]
(단일 행의 평균 너비를 기반으로 한 메모리 한도를 해시 테이블 크기가 초과하는 경우, 2단계 해싱이 적용됩니다.)

해시 조인은 해시 테이블이 완전히 구축될 때까지 결과를 반환하기 시작할 수 없습니다.

두 번째 단계에서(이 시점에 해시 테이블은 이미 구축되어 있습니다), 해시 조인 노드는 그의 두 번째 자식 노드를 호출하여 외부 행 집합을 가져옵니다. 스캔된 각 행에 대해, 해시 테이블은 일치하는 항목을 찾기 위해 검색됩니다. 이는 조인 조건에 포함된 외부 집합의 열에 대한 해시 키를 계산하는 것을 요구합니다.



찾아진 일치 항목들은 부모 노드로 반환됩니다.

비용 추정. 우리는 이미 카디널리티 추정에 대해 다루었습니다; 이것은 조인 방법에 의존하지 않기 때문에, 이제 비용 추정에 초점을 맞출 것입니다.

해시 노드의 비용은 그 자식 노드의 총 비용으로 표현됩니다. 이것은 단순히 계획[315]에서 슬롯을 채우는 더미

[314] backend/executor/nodeHash.c, ExecChooseHashTableSize function

[315] backend/optimizer/plan/createplan.c, create_hashjoin_plan function

숫자입니다. 모든 실제 추정치는 해시 조인 노드[316]의 비용에 포함됩니다.

다음은 그 예입니다:

```
=> EXPLAIN (analyze, timing off, summary off)
SELECT * FROM flights f
JOIN seats s ON s.aircraft_code = f.aircraft_code;
                      QUERY PLAN
-----------------------------------------------------------------------
  Hash Join (cost=38.13..278507.28 rows=16518865 width=78)
      (actual rows=16518865 loops=1)
      Hash Cond: (f.aircraft_code = s.aircraft_code)
      -> Seq Scan on flights f (cost=0.00..4772.67 rows=214867 widt...
              (actual rows=214867 loops=1)
      -> Hash (cost=21.39..21.39 rows=1339 width=15)
              (actual rows=1339 loops=1)
              Buckets: 2048 Batches: 1 Memory Usage: 79kB
              -> Seq Scan on seats s (cost=0.00..21.39 rows=1339 width=15)
                      (actual rows=1339 loops=1)
(10 rows)
```

조인의 시작 비용은 주로 해시 테이블 생성의 비용을 반영하며 다음 구성 요소를 포함합니다:

- 해시 테이블을 구축하는 데 필요한 내부 집합의 총 획득 비용
- 내부 집합의 각 행에 대해 조인 키에 포함된 모든 열의 해시 함수 계산 비용 (cpu_operator_cost(기본값: 0.0025) 당 연산으로 추정)
- 모든 내부 행을 해시 테이블에 삽입하는 비용(cpu_tuple_cost(기본값: 0.01) 당 삽입된 행으로 추정)
- 조인 작업을 시작하는 데 필요한 외부 행 집합의 획득 시작 비용

총 비용은 시작 비용과 조인 자체의 비용으로 구성되며, 다음을 포함합니다:

- 외부 집합의 각 행에 대해 조인 키에 포함된 모든 열의 해시 함수 계산 비용(cpu_operator_cost)
- 가능한 해시 충돌을 해결하기 위해 필요한 조인 조건 재검사 비용(cpu_operator_cost 당 각 검사된 연산자로 추정)
- 각 결과 행에 대한 처리 비용(cpu_tuple_cost)

필요한 재검사 횟수를 추정하는 것이 가장 어렵습니다. 이것은 외부 집합의 행 수에 내부 집합의 일부 분수(해시 테이블에 저장됨)를 곱하여 계산됩니다. 이 분수를 추정하기 위해, 계획자는 데이터 분포가 균일하지 않을 수 있다는 점을 고려해야 합니다. 이 계산의 세부 사항은 여러분에게 생략하겠습니다[317]; 이 특정 경우에, 이 분수는 **0.150112**로 추정됩니다.

---

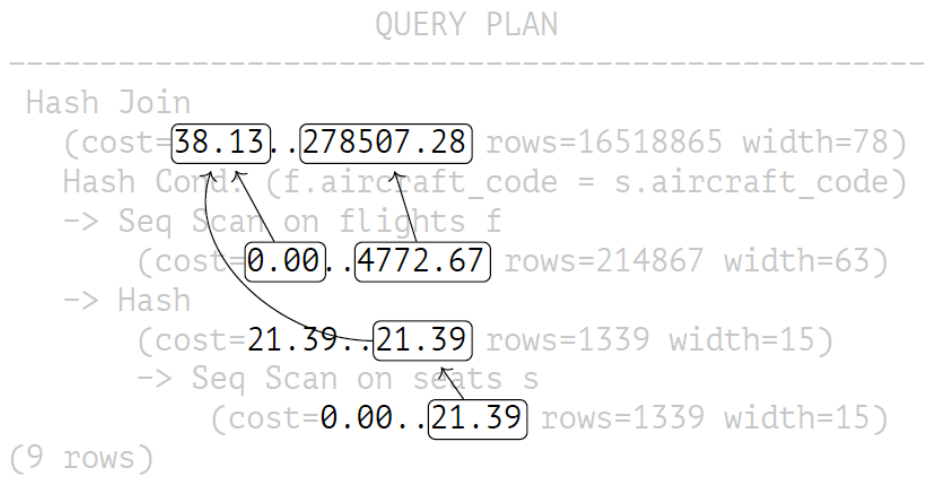
316 backend/optimizer/path/costsize.c, initial_cost_hashjoin and final_cost_hashjoin functions

317 backend/utils/adt/selfuncs.c, estimate_hash_bucket_stats function

따라서, 우리 쿼리의 비용은 다음과 같이 추정됩니다:

```
=> WITH cost(startup) AS (
  SELECT round((
              21.39 + current_setting('cpu_operator_cost')::real * 1339 +
                      current_setting('cpu_tuple_cost')::real * 1339 + 0.00
              )::numeric, 2)
)
SELECT startup,
  startup + round(( 4772.67 + current_setting('cpu_operator_cost')::real * 214867 +
                      current_setting('cpu_operator_cost')::real * 214867 * 1339 *
                      0.150112 + current_setting('cpu_tuple_cost')::real * 16518865
              )::numeric, 2) AS total
FROM cost;
 startup ¦ total
---------+-----------
   38.13 ¦ 278507.26
(1 row)
```

그리고 여기 의존성 그래프가 있습니다:



2-패스 해쉬 조인

계획자의 추정에 따르면 해시 테이블이 할당된 메모리에 맞지 않을 경우, 내부 행 집합은 별도로 처리될 배치로 분할됩니다. 배치 수(버킷 수와 마찬가지로)는 항상 2의 거듭제곱이며, 사용할 배치는 해시 키[318]의 해당 비트 수에 의해 결정됩니다.

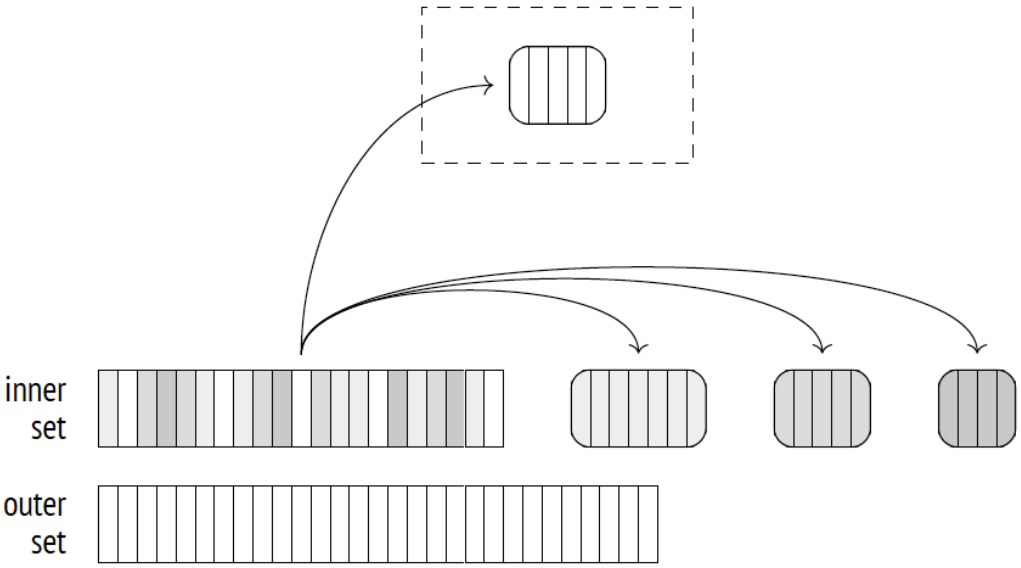
어떤 두 일치하는 행도 하나의 동일한 배치에 속합니다: 다른 배치에 배치된 행들은 동일한 해시 코드를 가질 수 없습니다.

[318] backend/executor/nodeHash.c, ExecHashGetBucketAndBatch function

모든 배치는 동일한 수의 해시 키를 보유합니다. 데이터가 균일하게 분포되어 있다면, 배치 크기도 대략적으로 같을 것입니다. 계획자는 적절한 수의 배치를 선택함으로써 메모리 소비를 제어할 수 있습니다.[319]

첫 번째 단계에서, 실행자는 해시 테이블을 구축하기 위해 내부 행 집합을 스캔합니다. 스캔된 행이 첫 번째 배치에 속하는 경우, 그것은 해시 테이블에 추가되고 RAM에 유지됩니다. 그렇지 않으면, 그것은 임시 파일에 기록됩니다(각 배치마다 별도의 파일이 있습니다).[320]

> 세션에서 디스크에 저장할 수 있는 임시 파일의 총 용량은 `temp_file_limit`(기본값: -1) 매개변수에 의해 제한됩니다(임시 테이블은 이 한도에 포함되지 않습니다). 세션이 이 값에 도달하면 쿼리는 중단됩니다.



두 번째 단계에서, 외부 집합이 스캔됩니다. 행이 첫 번째 배치에 속하는 경우, 그것은 내부 집합의 첫 번째 배치 행들을 포함하는 해시 테이블과 대조됩니다(어차피 다른 배치에서는 일치하는 항목이 없을 것입니다).

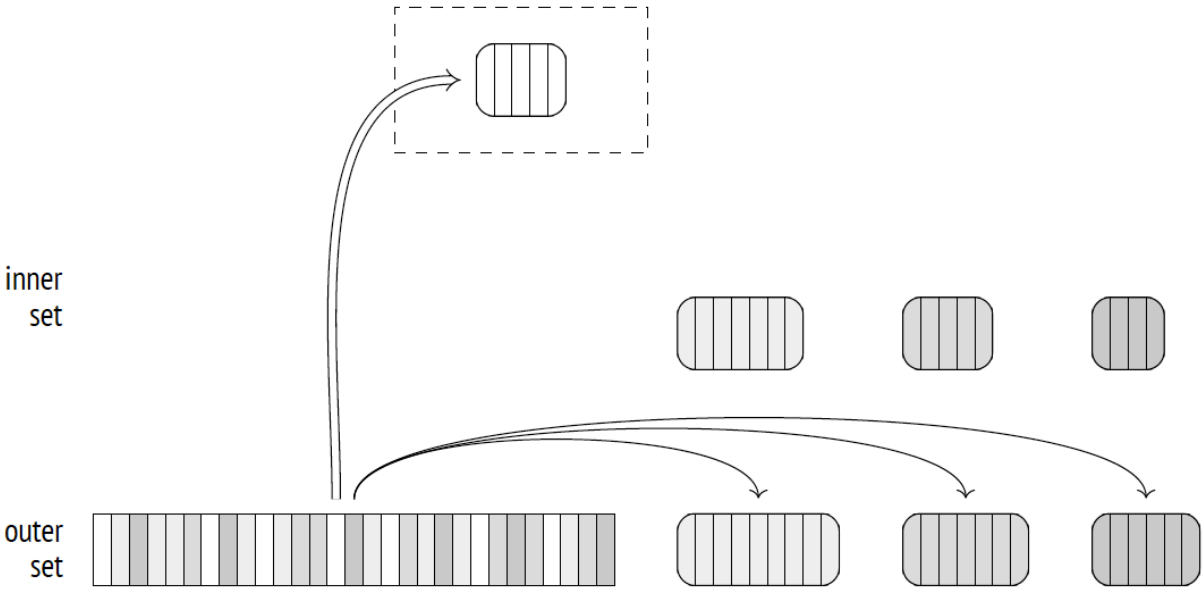
행이 다른 배치에 속하는 경우, 그것은 임시 파일에 저장되며, 이 파일은 다시 각 배치마다 별도로 생성됩니다. 따라서, N개의 배치는 2(N – 1)개의 파일(일부 배치가 비어 있을 경우 더 적을 수 있음)을 사용할 수 있습니다.

두 번째 단계가 완료되면, 해시 테이블에 할당된 메모리는 해제됩니다. 이 시점에서, 우리는 이미 배치 중 하나에 대한 조인 결과를 가지고 있습니다.

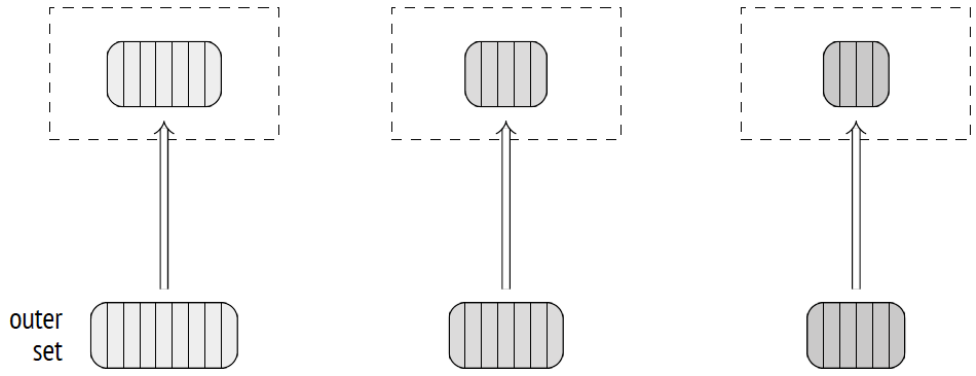
---

[319] backend/executor/nodeHash.c, ExecChooseHashTableSize function

[320] backend/executor/nodeHash.c, ExecHashTableInsert function



디스크에 저장된 각 배치에 대해 두 단계가 반복됩니다: 내부 집합의 행들이 임시 파일에서 해시 테이블로 전송되고, 그 다음 동일한 배치에 관련된 외부 집합의 행들이 다른 임시 파일에서 읽혀지고 이 해시 테이블과 대조됩니다. 처리가 완료되면, 임시 파일들은 삭제됩니다.



원패스 조인에 대한 유사한 출력과 달리, 두 패스 조인에 대한 `EXPLAIN` 명령어의 출력은 한 개 이상의 배치를 포함합니다. `BUFFERS` 옵션과 함께 실행되면, 이 명령어는 디스크 접근에 대한 통계도 표시합니다:

```
=> EXPLAIN (analyze, buffers, costs off, timing off, summary off)
SELECT *
FROM bookings b
JOIN tickets t ON b.book_ref = t.book_ref;
                QUERY PLAN
-------------------------------------------------------------------
  Hash Join (actual rows=2949857 loops=1)
      Hash Cond: (t.book_ref = b.book_ref)
      Buffers: shared hit=7236 read=55626, temp read=55126 written=55126
      -> Seq Scan on tickets t (actual rows=2949857 loops=1) Buffers: shared read=49415
      -> Hash (actual rows=2111110 loops=1)
              Buckets: 65536 Batches: 64 Memory Usage: 2277kB
              Buffers: shared hit=7236 read=6211, temp written=10858
```

```
               -> Seq Scan on bookings b (actual rows=2111110 loops=1)
                     Buffers: shared hit=7236 read=6211
(11 rows)
```

이미 `work_mem` 설정을 증가시킨 쿼리를 위에서 보여드렸습니다. 기본값인 4MB는 전체 해시 테이블이 `RAM`에 맞기에는 너무 작습니다; 이 예제에서, 데이터는 64개의 배치로 분할되고, 해시 테이블은 64K = 2^{16} 버킷을 사용합니다. 해시 테이블이 구축되는 동안(`Hash node`), 데이터는 임시 파일들에 기록됩니다(`temp written`); 조인 단계(`Hash Join node`)에서, 임시 파일들은 읽고 쓰기 모두 수행됩니다(`temp read, written`).

임시 파일에 대한 더 많은 통계를 수집하려면, `log_temp_files`(기본값: -1) 매개변수를 0으로 설정할 수 있습니다. 그러면 서버 로그가 모든 임시 파일과 그 크기를 삭제 시점의 상태로 나열할 것입니다.

동적 조정

계획된 사건의 진행은 두 가지 문제에 의해 방해받을 수 있습니다: 부정확한 통계와 비균일한 데이터 분포입니다.

조인 키 열에서 값의 분포가 비균일한 경우, 다른 배치는 다른 크기를 가질 것입니다.

어떤 배치(맨 처음 배치를 제외하고)가 너무 크다고 판명되면, 그 배치의 모든 행들은 디스크에 기록되어야 하고, 그 후에 디스크에서 읽혀져야 합니다. 외부 집합이 대부분의 문제를 일으키는데, 이는 일반적으로 더 크기 때문입니다. 그러므로 외부 집합의 `MCV`에 대한 정규적이고 비다변량 통계가 있다면(즉, 외부 집합이 테이블로 표현되고 조인이 단일 열로 수행된다면), `MCV`에 해당하는 해시 코드를 가진 행들은 첫 번째 배치[321]의 일부로 간주됩니다. 이 기술(왜곡 최적화라고 함)은 2-패스 조인의 `I/O` 오버헤드를 어느 정도 줄일 수 있습니다.

이 두 요인 때문에, 일부(또는 모든) 배치의 크기가 추정치를 초과할 수 있습니다. 그런 다음 해당 해시 테이블은 할당된 메모리 덩어리에 맞지 않고 정의된 한계를 초과할 것입니다.

그래서 해시 테이블을 구축하는 과정에서 너무 크게 되면, 배치의 수는 즉석에서 증가(두 배로 됨)됩니다. 각 배치는 가상으로 두 개의 새로운 것으로 분할됩니다: 행의 대략 절반(분포가 균일하다고 가정할 때)은 해시 테이블에 남아 있고, 다른 절반은 새 임시 파일에 저장됩니다.[322]

이러한 분할은 원래 1-패스 조인이 계획되었더라도 발생할 수 있습니다. 사실, 1-패스 조인과 2-패스 조인은 동일한 코드에 의해 구현된 하나의 같은 알고리즘을 사용합니다; 여기서 나는 단지 이야기를 더 부드럽게 전달하기 위해 그들을 별도로 구분합니다.

배치의 수를 줄일 수는 없습니다. 플래너가 데이터 크기를 과대평가한 것으로 밝혀지면, 배치들은 함께 병합되지 않을 것입니다.

비균일 분포의 경우, 배치 수를 늘리는 것이 도움이 되지 않을 수 있습니다. 예를 들어, 키 열이 모든 행에서

---

[321] backend/executor/nodeHash.c, ExecHashBuildSkewHash function

[322] backend/executor/nodeHash.c, ExecHashIncreaseNumBatches function

동일한 값을 가지고 있다면, 해시 함수가 계속해서 동일한 값을 반환하기 때문에 그 행들은 같은 배치에 배치될 것입니다. 불행히도, 이 경우에 해시 테이블은 부과된 제한에 관계없이 계속 성장할 것입니다.

> 이론적으로, 이 문제는 부분적으로 배치를 스캔하는 멀티-패스 조인을 통해 해결될 수 있지만, 이는 지원되지 않습니다.

배치 수의 동적 증가를 보여주기 위해서, 우리는 먼저 몇 가지 조작을 수행해야 합니다:

```
=> CREATE TABLE bookings_copy (LIKE bookings INCLUDING INDEXES)
WITH (autovacuum_enabled = off);
=> INSERT INTO bookings_copy SELECT * FROM bookings;
INSERT 0 2111110
=> DELETE FROM bookings_copy WHERE random() < 0.9;
DELETE 1899232
=> ANALYZE bookings_copy;
=> INSERT INTO bookings_copy SELECT * FROM bookings
ON CONFLICT DO NOTHING;
INSERT 0 1899232
=> SELECT reltuples FROM pg_class WHERE relname = 'bookings_copy';
reltuples
-----------
211878
(1 row)
```

결과적으로, 우리는 bookings_copy라고 불리는 새로운 테이블을 얻게 됩니다. 이것은 bookings 테이블의 정확한 복사본이지만, 플래너가 그 안의 행 수를 열 배로 과소평가합니다. 유사한 상황은 다른 조인 연산에 의해 생성된 행 집합에 대해 해시 테이블이 생성될 때 발생할 수 있으므로, 신뢰할 수 있는 통계가 없습니다.

이 계산 오류로 인해 플래너는 8개의 버킷이 충분하다고 생각하지만, 조인이 수행되는 동안 이 숫자는 32개로 증가합니다:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT *
FROM bookings_copy b
JOIN tickets t ON b.book_ref = t.book_ref;
                QUERY PLAN
-------------------------------------------------------------------------
  Hash Join (actual rows=2949857 loops=1)
      Hash Cond: (t.book_ref = b.book_ref)
      -> Seq Scan on tickets t (actual rows=2949857 loops=1)
      -> Hash (actual rows=2111110 loops=1)
              Buckets: 65536 (originally 65536) Batches: 32 (originally 8)
                      Memory Usage: 4040kB
              -> Seq Scan on bookings_copy b (actual rows=2111110 loops=1)
(7 rows)
```

비용 추정. 이미 이 예시를 사용하여 원패스 조인에 대한 비용 추정을 보여주었지만, 이제 사용 가능한 메모리의 크기를 최소로 줄여서 플래너가 두 배치를 사용해야 할 것입니다. 이것은 조인의 비용을 증가시킵니다:

```
=> SET work_mem = '64kB';
=> EXPLAIN (analyze, timing off, summary off)
SELECT * FROM flights f
JOIN seats s ON s.aircraft_code = f.aircraft_code;
                QUERY PLAN
-------------------------------------------------------------------------
  Hash Join (cost=45.13..283139.28 rows=16518865 width=78)
      (actual rows=16518865 loops=1)
      Hash Cond: (f.aircraft_code = s.aircraft_code)
      -> Seq Scan on flights f (cost=0.00..4772.67 rows=214867 widt...
              (actual rows=214867 loops=1)
      -> Hash (cost=21.39..21.39 rows=1339 width=15)
              (actual rows=1339 loops=1)
              Buckets: 2048 Batches: 2 Memory Usage: 55kB
              -> Seq Scan on seats s (cost=0.00..21.39 rows=1339 width=15)
                      (actual rows=1339 loops=1)
(10 rows)
=> RESET work_mem;
```

두 번째 패스의 비용은 행을 임시 파일로 옮기고 이 파일들로부터 읽는 데에 발생합니다.

2-패스 조인의 시작 비용은 1-패스 조인의 비용에 기반을 두고 있으며, 내부 집합[323]의 모든 행의 필요한 필드를 저장하기 위해 필요한 만큼의 페이지를 쓰는 추정 비용에 의해 증가됩니다. 해시 테이블이 구축될 때 첫 번째 배치가 디스크에 쓰이지 않더라도, 추정은 이를 고려하지 않으며 따라서 배치 수에 의존하지 않습니다.

차례로, 전체 비용은 1-패스 조인의 전체 비용과 이전에 디스크에 저장된 내부 집합의 행들을 읽는 추정 비용, 그리고 외부 집합의 행들을 읽고 쓰는 추정 비용을 포함합니다.

쓰기와 읽기 모두 페이지 당 `seq_page_cost`로 추정되며, `I/O` 작업은 순차적으로 가정됩니다.

이 특정 경우에, 내부 집합에 필요한 페이지 수는 7로 추정되며, 외부 집합의 데이터는 2309 페이지에 맞을 것으로 예상됩니다. 이 추정치를 위에서 계산한 1-패스 조인 비용에 추가하면, 쿼리 계획에서 보여진 것과 같은 수치를 얻게 됩니다:

```
=> SELECT 38.13 + -- startup cost of a one-pass join
      current_setting('seq_page_cost')::real * 7
         AS startup,
         278507.28 + -- total cost of a one-pass join
      current_setting('seq_page_cost')::real * 2 * (7 + 2309)
AS total;
```

[323] backend/optimizer/path/costsize.c, page_size function

```
 startup ¦ total
---------+-----------
   45.13 ¦ 283139.28
(1 row)
```

따라서, 충분한 메모리가 없는 경우, 조인은 두 번의 패스로 수행되며 효율이 떨어집니다. 그러므로 다음 사항을 준수하는 것이 중요합니다:

- 쿼리는 해시 테이블에서 불필요한 필드를 제외하도록 구성되어야 합니다.
- 플래너는 해시 테이블을 구축할 때 두 행 집합 중 더 작은 것을 선택해야 합니다.

병렬 계획에서 해시 조인 사용

위에서 설명한 해시 조인 알고리즘은 병렬 계획에서도 사용될 수 있습니다. 먼저, 여러 병렬 프로세스가 서로 독립적으로 내부 집합에 대해 자신만의 (완전히 동일한) 해시 테이블을 구축한 다음, 동시에 외부 집합을 처리하기 시작합니다. 여기서 성능 향상은 각 프로세스가 외부 행의 자신의 몫만을 스캔하기 때문에 발생합니다.

다음 계획은 일반적인 1-패스 해시 조인을 사용합니다:

```
=> SET work_mem = '128MB';
=> SET enable_parallel_hash = off;
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT count(*)
FROM bookings b
JOIN tickets t ON t.book_ref = b.book_ref;
                      QUERY PLAN
------------------------------------------------------------------------
Finalize Aggregate (actual rows=1 loops=1)
  -> Gather (actual rows=3 loops=1)
      Workers Planned: 2
      Workers Launched: 2
      -> Partial Aggregate (actual rows=1 loops=3)
              -> Hash Join (actual rows=983286 loops=3)
                      Hash Cond: (t.book_ref = b.book_ref)
                      -> Parallel Index Only Scan using tickets_book_ref...
                              Heap Fetches: 0
                      -> Hash (actual rows=2111110 loops=3)
                              Buckets: 4194304 Batches: 1 Memory Usage: 113172kB
                              -> Seq Scan on bookings b (actual rows=2111110...
(13 rows)
=> RESET enable_parallel_hash;
```

여기서 각 프로세스는 `bookings` 테이블을 해싱한 다음, `Parallel Index Only Scan` 노드를 통해 자신의 외부 행 몫을 검색하고 이 행들을 결과 해시 테이블과 매칭합니다.

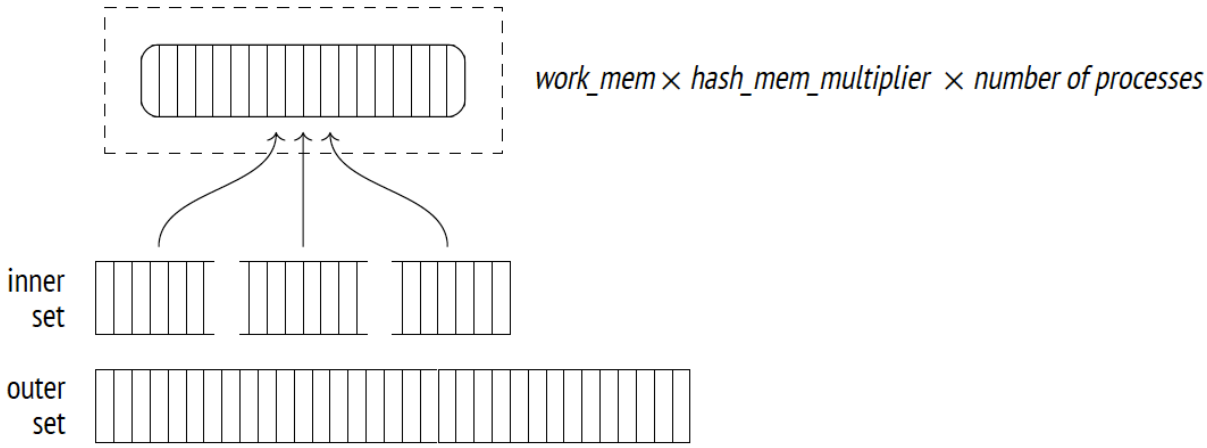
해시 테이블 메모리 제한은 각 병렬 프로세스에 별도로 적용되므로, 이 목적으로 할당된 총 메모리 크기는 계획에서 지정된 것보다 세 배 더 클 것입니다(메모리 사용량).

병렬 1-패스 해쉬 조인

일반 해시 조인은 병렬 계획에서 상당히 효율적일 수 있습니다(특히 병렬 처리가 크게 의미가 없는 작은 내부 집합에 대해), 그러나 더 큰 데이터 세트는 특별한 병렬 해시 조인 알고리즘으로 더 잘 처리됩니다.

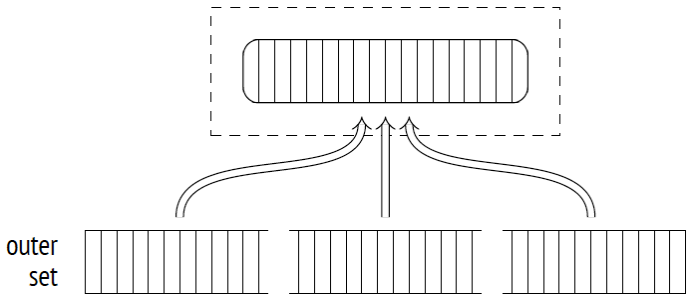
이 알고리즘의 병렬 버전의 중요한 차이점은 해시 테이블이 공유 메모리에 생성된다는 것입니다. 이 메모리는 동적으로 할당되며 조인 작업에 기여하는 모든 병렬 프로세스가 접근할 수 있습니다. 여러 개의 별도 해시 테이블 대신, 참여하는 모든 프로세스에 할당된 총 메모리 양을 사용하는 단일 공통 테이블이 구축됩니다. 이는 한 번의 패스로 조인을 완료할 확률을 증가시킵니다.

첫 번째 단계에서(계획에서 `Parallel Hash` 노드로 표시됨), 모든 병렬 프로세스들은 내부 집합에 대한 병렬 접근을 활용하여 공통 해시 테이블을 구축합니다.[324]



여기서 진행하려면, 각 병렬 프로세스는 첫 번째 단계의 처리 분담을 완료해야 합니다.[325]

두 번째 단계(`Parallel Hash Join` 노드)에서, 프로세스들은 이미 구축된 해시 테이블에 대해 외부 집합의 행들의 분담을 매칭하기 위해 다시 병렬로 실행됩니다.[326]



[324] backend/executor/nodeHash.c, MultiExecParallelHash function

[325] backend/storage/ipc/barrier.c

[326] backend/executor/nodeHashjoin.c, ExecParallelHashJoin function

이러한 계획의 예시는 다음과 같습니다:

```
=> SET work_mem = '64MB';
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT count(*)
FROM bookings b
JOIN tickets t ON t.book_ref = b.book_ref;
                      QUERY PLAN
-----------------------------------------------------------------------
  Finalize Aggregate (actual rows=1 loops=1)
      -> Gather (actual rows=3 loops=1)
              Workers Planned: 2
              Workers Launched: 2
              -> Partial Aggregate (actual rows=1 loops=3)
                      -> Parallel Hash Join (actual rows=983286 loops=3)
                            Hash Cond: (t.book_ref = b.book_ref)
                            -> Parallel Index Only Scan using tickets_book_ref...
                                  Heap Fetches: 0
                            -> Parallel Hash (actual rows=703703 loops=3)
                                  Buckets: 4194304 Batches: 1 Memory Usage:
                                  115392kB
                            -> Parallel Seq Scan on bookings b (actual row...
(13 rows)
=> RESET work_mem;
```

이전 섹션에서 보여드린 것과 동일한 쿼리이지만, 그때는 `enable_parallel_hash`(기본값: on) 매개변수를 사용하여 병렬 해시 조인 기능이 꺼져 있었습니다.

앞서 보여준 일반 해시 조인에 비해 사용 가능한 메모리가 절반으로 줄어들었음에도 불구하고, 이 작업은 모든 병렬 프로세스에 할당된 메모리를 사용하기 때문에 한 번의 패스로 여전히 완료됩니다(메모리 사용량). 해시 테이블은 조금 더 커지지만, 현재 우리가 가진 것이 이것뿐이므로 전체 메모리 사용량이 감소했습니다.

병렬 2-패스 해쉬 조인

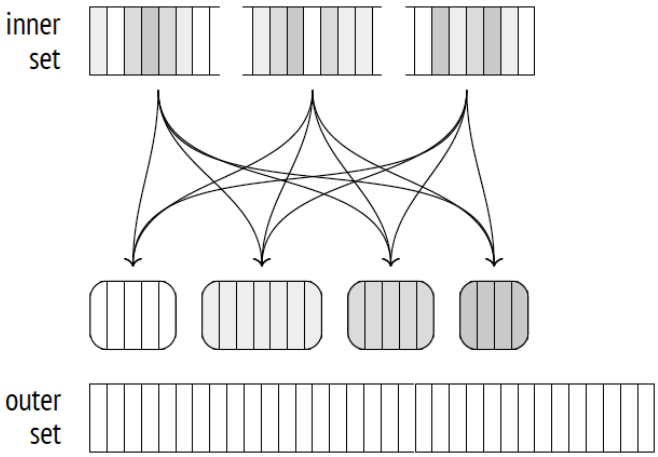
모든 병렬 프로세스의 통합 메모리가 여전히 전체 해시 테이블을 수용하기에 충분하지 않을 수 있습니다. 이는 계획 단계에서나 나중에, 쿼리 실행 중에 명확해질 수 있습니다. 이 경우에 적용된 2- 패스 알고리즘은 지금까지 본 것과 상당히 다릅니다.

이 알고리즘의 핵심 차이점은 단일 큰 테이블 대신 여러 개의 작은 해시 테이블을 생성한다는 것입니다. 각 프로세스는 자신만의 테이블을 받아 독립적으로 자신의 배치를 처리합니다. (하지만 별도의 해시 테이블이 여전히 공유 메모리에 위치해 있기 때문에, 어떤 프로세스든 이 테이블들 중 어느 하나에 접근할 수 있습니다.) 계획에서 여러 배치가 필요하다는 것이 나타나면,[327] 각 프로세스를 위한 별도의 해시 테이블이 바로 구

[327] backend/executor/nodeHash.c, ExecChooseHashTableSize function

축됩니다. 실행 단계에서 결정이 내려지면, 해시 테이블이 재구축됩니다.[328]

따라서 **첫 번째 단계**에서 프로세스들은 내부 집합을 병렬로 스캔하며, 이를 배치로 나누어 임시 파일에 씁니다.[329] 각 프로세스가 내부 집합의 자신의 몫만 읽기 때문에, 그 어떤 배치(첫 번째 배치조차도)에 대해서도 전체 해시 테이블을 구축하지 않습니다. 어떤 배치의 전체 행 세트는 동기화된 방식으로 모든 병렬 프로세스에 의해 작성된 파일에만 누적됩니다.[330] 따라서 비병렬 및 일회성 병렬 버전의 알고리즘과 달리, 병렬 두 번째 패스 해시 조인은 첫 번째 배치를 포함하여 모든 배치를 디스크에 씁니다.



내부 집합에 대한 해싱이 모든 프로세스에 의해 완료되면, **두 번째 단계**가 시작됩니다.[331]

알고리즘의 비병렬 버전을 사용했다면, 첫 번째 배치에 속하는 외부 집합의 행들이 바로 해시 테이블과 매치될 것입니다. 하지만 병렬 버전의 경우, 메모리에는 아직 해시 테이블이 존재하지 않으므로, 작업자들이 배치를 독립적으로 처리합니다. 따라서, 두 번째 단계는 외부 집합의 병렬 스캔으로 시작하여 그 행들을 배치로 분배하고, 각 배치를 별도의 임시 파일에 씁니다.[332] 스캔된 행들은 해시 테이블에 삽입되지 않습니다(첫 번째 단계에서 발생하는 것처럼), 그래서 배치의 수는 절대 증가하지 않습니다.

모든 프로세스가 외부 집합의 스캔을 완료하면, 디스크에는 2N개의 임시 파일이 생성됩니다; 이들은 내부 및 외부 집합의 배치를 포함합니다.

---

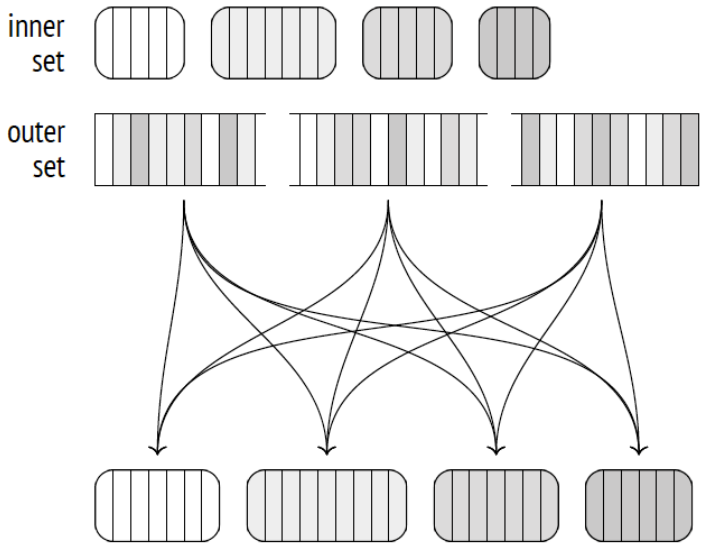
328 *back*end/executor/nodeHash.c, ExecParallelHashIncreaseNumBatches function

329 backend/executor/nodeHash.c, MultiExecParallelHash function

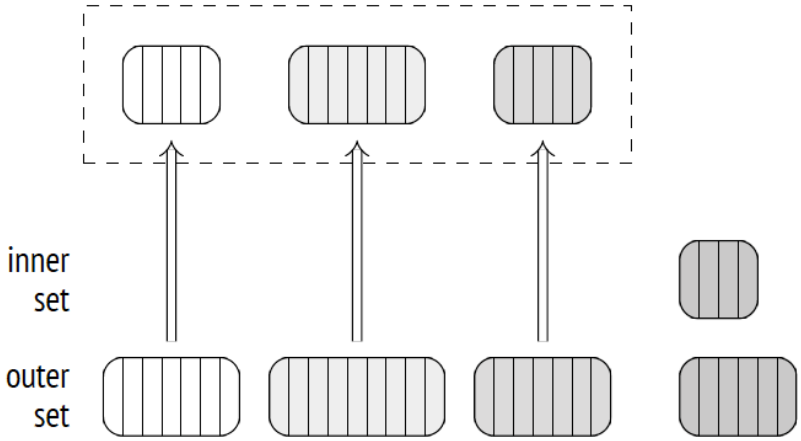
330 backend/utils/sort/sharedtuplestore.c

331 backend/executor/nodeHashjoin.c, ExecParallelHashJoin function

332 backend/executor/nodeHashjoin.c, ExecParallelHashJoinPartitionOuter function

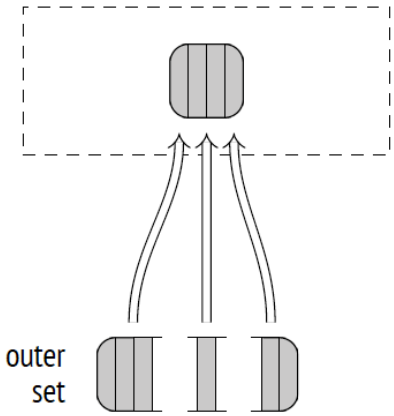


그런 다음 각 프로세스는 배치 중 하나를 선택하고 조인을 수행합니다: 메모리에 있는 해시 테이블에 내부 집합의 행들을 로드하고, 외부 집합의 행들을 스캔한 후, 그것들을 해시 테이블과 매칭합니다. 배치 조인이 완료되면, 프로세스는 아직 처리되지 않은 다음 배치를 선택합니다.[333]



처리되지 않은 배치가 더 이상 남아 있지 않다면, 자신의 배치를 완료한 프로세스는 다른 프로세스가 현재 처리 중인 배치 중 하나를 처리하기 시작합니다; 모든 해시 테이블이 공유 메모리에 위치해 있기 때문에 이러한 동시 처리가 가능합니다.

[333] backend/executor/nodeHashjoin.c, ExecParallelHashJoinNewBatch function



이 접근 방식은 모든 프로세스에 대해 단일 큰 해시 테이블을 사용하는 것보다 더 효율적입니다: 병렬 처리를 설정하기가 더 쉽고, 동기화 비용이 저렴합니다.

수정 사항

해시 조인 알고리즘은 내부 조인뿐만 아니라 왼쪽, 오른쪽, 전체 외부 조인, 그리고 반(세미) 조인과 반대(안티) 조인과 같은 모든 유형의 조인을 지원합니다. 하지만 이미 언급했듯이, 조인 조건은 동등 연산자로 제한됩니다.

중첩 루프 조인을 다루면서 이러한 작업 중 일부를 이미 관찰했습니다. 여기 오른쪽 외부 조인의 예시가 있습니다:

```
=> EXPLAIN (costs off) SELECT *
FROM bookings b
LEFT OUTER JOIN tickets t ON t.book_ref = b.book_ref;
              QUERY PLAN
---------------------------------------
  Hash Right Join
      Hash Cond: (t.book_ref = b.book_ref)
      -> Seq Scan on tickets t
      -> Hash
              -> Seq Scan on bookings b
(5 rows)
```

SQL 쿼리에서 지정된 논리적 왼쪽 조인이 실행 계획에서 오른쪽 조인의 물리적 작업으로 변환되었다는 점을 기억하는 것이 중요합니다.

논리적 수준에서, `bookings`는 외부 테이블(조인 작업의 왼쪽을 구성)이며, `tickets` 테이블은 내부 테이블입니다. 따라서, `tickets`가 없는 `bookings`도 조인 결과에 포함되어야 합니다.

물리적 수준에서, 내부 및 외부 집합은 쿼리 텍스트에서의 위치가 아닌 조인의 비용을 기반으로 할당됩니다. 이는 일반적으로 더 작은 해시 테이블을 가진 집합이 내부 집합으로 사용될 것임을 의미합니다. 여기서 발생하는 것이 바로 이러한 경우입니다: `bookings` 테이블이 내부 집합으로 사용되며, 왼쪽 조인은 오른쪽 조인으로 변경됩니다.

그리고 그 반대의 경우도 마찬가지입니다. 쿼리에서 오른쪽 외부 조인을 지정하여(어떤 bookings에도 연관되지 않은 티켓을 표시하기 위해), 실행 계획은 왼쪽 조인을 사용합니다:

```
=> EXPLAIN (costs off) SELECT *
FROM bookings b
RIGHT OUTER JOIN tickets t ON t.book_ref = b.book_ref;
              QUERY PLAN
---------------------------------------
  Hash Left Join
      Hash Cond: (t.book_ref = b.book_ref)
      -> Seq Scan on tickets t
      -> Hash
              -> Seq Scan on bookings b
(5 rows)
```

전체 그림을 완성하기 위해, 전체 외부 조인을 사용하는 쿼리 계획의 예시를 제공하겠습니다:

```
=> EXPLAIN (costs off) SELECT *
FROM bookings b
FULL OUTER JOIN tickets t ON t.book_ref = b.book_ref;
              QUERY PLAN
---------------------------------------
  Hash Full Join
      Hash Cond: (t.book_ref = b.book_ref)
      -> Seq Scan on tickets t
      -> Hash
              -> Seq Scan on bookings b
(5 rows)
```

병렬 해시 조인은 현재 오른쪽 조인과 전체 조인에 대해서 지원되지 않습니다.[334]

다음 예시에서는 bookings 테이블을 외부 집합으로 사용하지만, 만약 지원된다면 계획자는 오른쪽 조인을 선호했을 것입니다:

```
=> EXPLAIN (costs off) SELECT sum(b.total_amount)
FROM bookings b
LEFT OUTER JOIN tickets t ON t.book_ref = b.book_ref;
                     QUERY PLAN
-----------------------------------------------------------------------
  Finalize Aggregate
      -> Gather
              Workers Planned: 2
              -> Partial Aggregate
                      -> Parallel Hash Left Join
```

[334] commitfest.postgresql.org/33/2903

```
                              Hash Cond: (b.book_ref = t.book_ref)
                              -> Parallel Seq Scan on bookings b
                              -> Parallel Hash
                                      -> Parallel Index Only Scan using tickets_book...
(9 rows)
```

22.2 고유한 값 및 그룹화

집계를 위해 값을 그룹화하고 중복을 제거하는 알고리즘은 조인 알고리즘과 매우 유사합니다. 이들이 사용할 수 있는 접근 방법 중 하나는 필요한 열에 대해 해시 테이블을 구축하는 것입니다. 해당 값이 아직 해시 테이블에 없을 경우에만 값이 해시 테이블에 포함됩니다. 결과적으로, 해시 테이블은 모든 고유한 값을 축적하게 됩니다.

해시 집계를 수행하는 노드는 `HashAggregate`라고 불립니다.[335]

이 노드가 필요할 수 있는 몇 가지 상황을 고려해 보겠습니다.

각 여행 클래스별 좌석 수 (`GROUP BY`):

```
=> EXPLAIN (costs off) SELECT fare_conditions, count(*)
FROM seats
GROUP BY fare_conditions;
      QUERY PLAN
------------------------------
  HashAggregate
      Group Key: fare_conditions
      -> Seq Scan on seats
(3 rows)
```

여행 클래스 목록 (DISTINCT):

```
=> EXPLAIN (costs off) SELECT DISTINCT fare_conditions
FROM seats;
      QUERY PLAN
------------------------------
  HashAggregate
      Group Key: fare_conditions
      -> Seq Scan on seats
(3 rows)
```

하나 이상의 값과 결합된 여행 클래스 (UNION):

```
=> EXPLAIN (costs off) SELECT fare_conditions
FROM seats
```

[335] backend/executor/nodeAgg.c

```
UNION
SELECT NULL;
               QUERY PLAN
----------------------------------
  HashAggregate
      Group Key: seats.fare_conditions
      -> Append
              -> Seq Scan on seats
              -> Result
(5 rows)
```

`Append` 노드는 두 세트를 결합하지만, UNION 결과에 나타나지 않아야 할 어떠한 중복도 제거하지 않습니다. 중복은 `HashAggregate` 노드에 의해 별도로 제거되어야 합니다.

해시 테이블에 할당된 메모리 청크는 해시 조인의 경우와 마찬가지로 `work_mem`(기본값: 4MB) × `hash_mem_multiplier`(기본값: 1.0) 값에 의해 제한됩니다.

해시 테이블이 할당된 메모리에 맞는 경우, 집계는 단일 배치를 사용합니다:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT DISTINCT amount FROM ticket_flights;
               QUERY PLAN
-----------------------------------------------------------------
  HashAggregate (actual rows=338 loops=1)
      Group Key: amount
      Batches: 1 Memory Usage: 61kB
      -> Seq Scan on ticket_flights (actual rows=8391852 loops=1)
(4 rows)
```

`amounts` 필드에서 고유한 값이 그리 많지 않기 때문에, 해시 테이블은 오직 61k(메모리 사용량)만을 차지합니다.

할당된 메모리를 해시 테이블이 채우는 즉시, 모든 추가 값들은 임시 파일로 옮겨지고 그들의 해시 값의 여러 비트를 기반으로 파티션으로 그룹화됩니다. 파티션의 수는 2의 거듭제곱이며, 각각의 해시 테이블이 할당된 메모리에 맞도록 선택됩니다. 추정의 정확성은 물론 수집된 통계의 질에 달려 있으므로, 받은 숫자는 파티션 크기를 더 줄이고 한 번의 패스로 각 파티션을 처리할 확률을 높이기 위해 1.5배로 곱해집니다.[336]

전체 세트가 스캔되면, 노드는 해시 테이블에 들어간 값들에 대한 집계 결과를 반환합니다.

그런 다음 해시 테이블은 지워지고, 이전 단계에서 임시 파일에 저장된 각 파티션은 다른 행 세트처럼 스캔되고 처리됩니다. 만약 해시 테이블이 여전히 할당된 메모리를 초과한다면, 오버플로우 대상 행들은 다시 파티션되어 추가 처리를 위해 디스크에 쓰여집니다.

[336] backend/executor/nodeAgg.c, hash_choose_num_partitions function

과도한 I/O를 피하기 위해, 두 번째 패스 해시 조인 알고리즘은 MCV를 첫 번째 배치로 이동시킵니다. 그러나, 집계는 이러한 최적화를 요구하지 않습니다: 할당된 메모리에 맞는 행들은 파티션으로 나뉘지 않을 것이며, MCV는 RAM에 들어갈 만큼 충분히 일찍 발생할 가능성이 큽니다.

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT DISTINCT flight_id FROM ticket_flights;
                       QUERY PLAN
-------------------------------------------------------------
  HashAggregate (actual rows=150588 loops=1)
      Group Key: flight_id
      Batches: 5 Memory Usage: 4145kB Disk Usage: 98184kB
      -> Seq Scan on ticket_flights (actual rows=8391852 loops=1)
(4 rows)
```

이 예시에서, 고유 ID의 수가 상대적으로 높기 때문에 해시 테이블이 할당된 메모리에 맞지 않습니다. 쿼리를 수행하는 데는 다섯 번의 배치가 필요합니다: 초기 데이터 세트를 위한 하나와 디스크에 쓰여진 파티션들을 위한 네 개입니다.

23. 정렬과 병합

23.1 병합 조인

병합 조인은 조인 키로 정렬된 데이터 세트를 처리하고 유사한 방식으로 정렬된 결과를 반환합니다. 입력 세트는 인덱스 스캔을 따라 사전에 정렬될 수 있으며, 그렇지 않은 경우 실행자는 실제 병합이 시작되기 전에 세트를 정렬해야 합니다.[337]

정렬된 세트 병합

병합 조인의 예시를 살펴보겠습니다. 실행 계획에서는 "Merge Join" 노드로 표현됩니다:[338]

```
=> EXPLAIN (costs off) SELECT *
FROM tickets t
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
ORDER BY t.ticket_no;
                      QUERY PLAN
------------------------------------------------------------------
  Merge Join
      Merge Cond: (t.ticket_no = tf.ticket_no)
      -> Index Scan using tickets_pkey on tickets t
      -> Index Scan using ticket_flights_pkey on ticket_flights tf
(4 rows)
```

최적화 프로그램은 `ORDER BY` 절에 정의된 대로 정렬된 결과를 반환하기 때문에 이 조인 방법을 선호합니다. 계획을 선택할 때, 최적화 프로그램은 데이터 세트의 정렬 순서를 주목하고 정말 필요한 경우가 아니면 정렬을 수행하지 않습니다. 예를 들어, 병합 조인에 의해 생성된 데이터 세트가 이미 적절한 정렬 순서를 가지고 있다면, 그대로 이후의 병합 조인에서 사용될 수 있습니다.

```
=> EXPLAIN (costs off) SELECT *
FROM tickets t
JOIN ticket_flights tf ON t.ticket_no = tf.ticket_no
JOIN boarding_passes bp ON bp.ticket_no = tf.ticket_no
AND bp.flight_id = tf.flight_id
ORDER BY t.ticket_no;
                              QUERY PLAN
----------------------------------------------------------------------
  Merge Join
      Merge Cond: (tf.ticket_no = t.ticket_no)
      -> Merge Join
              Merge Cond: ((tf.ticket_no = bp.ticket_no) AND (tf.flight_...
              -> Index Scan using ticket_flights_pkey on ticket_flights tf
```

[337] backend/optimizer/path/joinpath.c, generate_mergejoin_paths function

[338] backend/executor/nodeMergejoin.c

```
              -> Index Scan using boarding_passes_pkey on boarding_passe...
      -> Index Scan using tickets_pkey on tickets t
(7 rows)
```

첫 번째로 조인되는 테이블은 `ticket_flights`와 `boarding_passes`이며, 두 테이블 모두 복합 기본 키(`ticket_no, flight_id`)를 가지고 있고, 결과는 이 두 열에 의해 정렬됩니다. 생성된 행 집합은 그 후 `ticket_no` 열로 정렬된 `tickets` 테이블과 조인됩니다.

이 조인은 두 데이터 세트 모두에 대해 단 한 번의 패스만 요구하며 추가 메모리를 사용하지 않습니다. 이는 내부 및 외부 세트의 현재 행(원래 첫 번째 행임)에 대한 두 포인터를 사용합니다.

현재 행의 키가 일치하지 않으면, 더 작은 키를 가진 행을 참조하는 포인터 중 하나가 다음 행으로 이동될 때까지 진행되어 일치하는 행을 찾습니다. 조인된 행은 상위 노드로 반환되고, 내부 세트의 포인터는 한 위치 앞으로 이동됩니다. 한 세트가 끝날 때까지 작업이 계속됩니다.

이 알고리즘은 내부 세트의 중복을 처리할 수 있지만, 외부 세트도 중복을 포함할 수 있습니다. 따라서, 알고리즘은 개선되어야 합니다: 외부 포인터가 진행된 후 키가 동일하게 유지되면, 내부 포인터는 첫 번째 일치하는 행으로 되돌아갑니다. 이렇게 하여 외부 세트의 각 행은 동일한 키를 가진 내부 세트의 모든 행과 매칭될 것입니다.[339]

외부 조인의 경우, 알고리즘은 약간 조정되지만 여전히 동일한 원칙에 기반합니다.

병합 조인 조건은 동등 연산자만 사용할 수 있으므로, 오직 등가 조인(equi-joins)만 지원됩니다(비록 다른 조건 유형에 대한 지원도 현재 진행 중입니다).[340]

비용 추정. 이전 예제를 좀 더 자세히 살펴보겠습니다:

```
=> EXPLAIN SELECT *
FROM tickets t
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
ORDER BY t.ticket_no;
                        QUERY PLAN
------------------------------------------------------------------
  Merge Join (cost=0.99..822355.54 rows=8391852 width=136)
      Merge Cond: (t.ticket_no = tf.ticket_no)
      -> Index Scan using tickets_pkey on tickets t
              (cost=0.43..139110.29 rows=2949857 width=104)
      -> Index Scan using ticket_flights_pkey on ticket_flights tf
              (cost=0.56..570972.46 rows=8391852 width=32)
(6 rows)
```

조인의 시작 비용에는 모든 자식 노드의 시작 비용이 최소한 포함됩니다. 일반적으로, 첫 번째 매치를 찾기

[339] backend/executor/nodeMergejoin.c, ExecMergeJoin function

[340] For example, see commitfest.postgresql.org/33/3160

전에 외부 세트나 내부 세트의 일부를 스캔해야 할 수도 있습니다. 이 비율은 두 세트에서 가장 작은 조인 키를 비교함으로써(히스토그램을 기반으로) 추정할 수 있습니다.[341] 그러나 이 특별한 경우에는, 두 테이블에서 티켓 번호의 범위가 동일합니다.

총 비용에는 자식 노드로부터 데이터를 가져오는 비용과 계산 비용이 포함됩니다. 조인 알고리즘은 한 세트가 끝나는 즉시 중지되므로(물론 외부 조인을 수행하는 경우를 제외하고), 다른 세트는 부분적으로만 스캔될 수 있습니다. 스캔된 부분의 크기를 추정하기 위해, 두 세트에서 최대 키 값들을 비교할 수 있습니다. 이 예제에서는 두 세트 모두 전체적으로 읽히므로, 조인의 총 비용에는 두 자식 노드의 총 비용 합이 포함됩니다.

게다가, 중복이 있는 경우 내부 세트의 일부 행은 여러 번 스캔될 수 있습니다. 반복 스캔의 추정 횟수는 조인 결과의 카디널리티와 내부 세트의 카디널리티 차이와 같습니다. 이 쿼리에서, 이 카디널리티는 같은데, 이는 세트에 중복이 없다는 것을 의미합니다.

알고리즘은 두 세트의 조인 키를 비교합니다. 한 번의 비교 비용은 cpu_operator_cost(기본값: 0.0025) 값으로 추정되며, 추정되는 비교 횟수는 두 세트의 행 합계로 취할 수 있습니다(중복에 의한 반복 읽기 횟수 증가). 결과에 포함된 각 행의 처리 비용은 평소와 같이 cpu_tuple_cost(기본값: 0.01) 값으로 추정됩니다.

따라서, 이 예제에서 조인의 비용은 다음과 같이 추정됩니다:[342]

```
=> SELECT 0.43 + 0.56 AS startup,
      round((
              139110.29 + 570972.46 +
              current_setting('cpu_tuple_cost')::real * 8391852 +
              current_setting('cpu_operator_cost')::real * (2949857 + 8391852)
      )::numeric, 2) AS total;
 startup ¦ total
---------+-----------
    0.99 ¦ 822355.54
(1 row)
```

병렬 모드

병합 조인은 병렬적인 특성이 없음에도 불구하고, 병렬 계획에서 여전히 사용될 수 있습니다.[343] 외부 세트는 여러 작업자에 의해 병렬로 스캔될 수 있지만, 내부 세트는 항상 각 작업자에 의해 전체적으로 스캔됩니다.

병렬 해시 조인이 거의 항상 더 저렴하기 때문에, 잠시 동안 해시 조인을 비활성화하겠습니다:

```
=> SET enable_hashjoin = off;
```

병합 조인을 사용하는 병렬 계획의 예는 다음과 같습니다:

```
=> EXPLAIN (costs off)
```

---

[341] backend/utils/adt/selfuncs.c, mergejoinscansel function

[342] backend/optimizer/path/costsize.c, initial_cost_mergejoin & final_cost_mergejoin functions

[343] backend/optimizer/path/joinpath.c, consider_parallel_mergejoin function

```
SELECT count(*), sum(tf.amount)
FROM tickets t
JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no;
                      QUERY PLAN
------------------------------------------------------------------------
  Finalize Aggregate
      -> Gather
              Workers Planned: 2
              -> Partial Aggregate
                      -> Merge Join
                              Merge Cond: (tf.ticket_no = t.ticket_no)
                              -> Parallel Index Scan using ticket_flights_pkey o...
                              -> Index Only Scan using tickets_pkey on tickets t
(8 rows)
```

병렬 계획에서는 전체 외부 병합 조인(Full Outer Merge Join)과 오른쪽 외부 병합 조인(Right Outer Merge Join)이 허용되지 않습니다.

수정 사항

병합 조인 알고리즘은 모든 유형의 조인에 사용될 수 있습니다. 유일한 제한은 전체 외부 조인(Full Outer Join)과 오른쪽 외부 조인(Right Outer Join)의 조인 조건이 병합 호환 표현식("외부 열이 내부 열과 동일" 또는 "열이 상수와 동일")[344]을 포함해야 한다는 것입니다. 내부 조인(Inner Join)과 왼쪽 외부 조인(Left Outer Join)은 관련 없는 조건에 의해 조인 결과를 필터링하지만, 전체 조인과 오른쪽 조인에 대해서는 이러한 필터링이 적용되지 않습니다.

다음은 병합 알고리즘을 사용하는 전체 조인의 예입니다:

```
=> EXPLAIN (costs off) SELECT *
FROM tickets t
FULL JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
ORDER BY t.ticket_no;
                      QUERY PLAN
------------------------------------------------------------------------
  Sort
      Sort Key: t.ticket_no
      -> Merge Full Join
              Merge Cond: (t.ticket_no = tf.ticket_no)
              -> Index Scan using tickets_pkey on tickets t
              -> Index Scan using ticket_flights_pkey on ticket_flights tf
(6 rows)
```

내부 조인(Inner Join)과 왼쪽 조인(Left Join)은 정렬 순서를 유지합니다. 그러나 전체 외부 조인(Full Outer

[344] backend/optimizer/path/joinpath.c, select_mergejoin_clauses function

Join)과 오른쪽 외부 조인(Right Outer Join)은 정렬된 외부 세트의 값 사이에 `NULL` 값이 삽입될 수 있기 때문에 정렬 순서를 보장할 수 없습니다.[345] 필요한 순서를 복원하기 위해, 플래너는 여기에 Sort 노드를 도입합니다. 당연히, 이는 계획의 비용을 증가시켜 해시 조인을 더 매력적으로 만들므로, 플래너는 해시 조인이 현재 비활성화된 상태에서만 이 계획을 선택했습니다.

그러나 다음 예제는 해시 조인 없이는 수행할 수 없습니다: 중첩 루프는 전체 조인을 전혀 허용하지 않으며, 지원되지 않는 조인 조건 때문에 병합을 사용할 수 없습니다. 따라서 해시 조인은 `enable_hashjoin` 매개변수 값에 관계없이 사용됩니다:

```
=> EXPLAIN (costs off) SELECT *
FROM tickets t
FULL JOIN ticket_flights tf ON tf.ticket_no = t.ticket_no
AND tf.amount > 0
ORDER BY t.ticket_no;
              QUERY PLAN
-----------------------------------------------
  Sort
      Sort Key: t.ticket_no
      -> Hash Full Join
              Hash Cond: (tf.ticket_no = t.ticket_no)
              Join Filter: (tf.amount > '0'::numeric)
              -> Seq Scan on ticket_flights tf
              -> Hash
                      -> Seq Scan on tickets t
(8 rows)
```

이전에 비활성화했던 해시 조인 사용 기능을 복원합시다:

```
=> RESET enable_hashjoin;
```

23.2 정렬

조인 키로 정렬되지 않은 세트(또는 두 세트 모두 가능)가 있다면, 조인 작업이 시작되기 전에 재정렬되어야 합니다. 이 정렬 작업은 계획에서 Sort 노드로 표시됩니다:[346]

```
=> EXPLAIN (costs off)
SELECT * FROM flights f
JOIN airports_data dep ON f.departure_airport = dep.airport_code
ORDER BY dep.airport_code;
                      QUERY PLAN
------------------------------------------------------
  Merge Join
      Merge Cond: (f.departure_airport = dep.airport_code)
```

[345] backend/optimizer/path/pathkeys.c, build_join_pathkeys function

[346] backend/executor/nodeSort.c

```
      -> Sort
              Sort Key: f.departure_airport
              -> Seq Scan on flights f
      -> Sort
              Sort Key: dep.airport_code
              -> Seq Scan on airports_data dep
(8 rows)
```

이러한 정렬은 일반 쿼리와 윈도우 함수 내에서 `ORDER BY` 절이 지정되는 경우, 조인의 맥락 외부에서도 적용될 수 있습니다:

```
=> EXPLAIN (costs off)
SELECT flight_id,
row_number() OVER (PARTITION BY flight_no ORDER BY flight_id)
FROM flights f;
              QUERY PLAN
-------------------------------------
  WindowAgg
      -> Sort
              Sort Key: flight_no, flight_id
              -> Seq Scan on flights f
(4 rows)
```

여기서 `WindowAgg` 노드[347]는 `Sort` 노드에 의해 사전에 정렬된 데이터 세트에 대해 윈도우 함수를 계산합니다.
플래너는 여러 가지 정렬 방법을 도구 상자에 가지고 있습니다. 이미 보여드린 예제는 그 중 두 가지(Sort Method)를 사용합니다. 이러한 세부 사항은 평소와 같이 `EXPLAIN ANALYZE` 명령어로 표시될 수 있습니다:

```
=> EXPLAIN (analyze,costs off,timing off,summary off)
SELECT * FROM flights f
JOIN airports_data dep ON f.departure_airport = dep.airport_code
ORDER BY dep.airport_code;
                      QUERY PLAN
------------------------------------------------------------------
  Merge Join (actual rows=214867 loops=1)
      Merge Cond: (f.departure_airport = dep.airport_code)
      -> Sort (actual rows=214867 loops=1)
              Sort Key: f.departure_airport
              Sort Method: external merge Disk: 17136kB
              -> Seq Scan on flights f (actual rows=214867 loops=1)
      -> Sort (actual rows=104 loops=1)
              Sort Key: dep.airport_code
              Sort Method: quicksort Memory: 52kB
```

[347] backend/executor/nodeWindowAgg.c

```
              -> Seq Scan on airports_data dep (actual rows=104 loops=1)
(10 rows)
```

퀵정렬

정렬할 데이터 세트가 work_mem(기본값: 4MB) 청크에 맞는 경우, 클래식 퀵정렬 방법이 적용됩니다. 이 알고리즘은 모든 교과서에 설명되어 있으므로 여기서는 설명하지 않겠습니다.

구현과 관련하여, 정렬은 사용 가능한 메모리의 양[348]과 일부 다른 요인에 따라 가장 적합한 알고리즘을 선택하는 전용 컴포넌트에 의해 수행됩니다.

비용 추정. 작은 테이블이 어떻게 정렬되는지 살펴보겠습니다. 이 경우, 정렬은 메모리에서 퀵정렬 알고리즘을 사용하여 수행됩니다:

```
=> EXPLAIN SELECT *
FROM airports_data
ORDER BY airport_code;
                      QUERY PLAN
-------------------------------------------------------------------
  Sort (cost=7.52..7.78 rows=104 width=145)
      Sort Key: airport_code
      -> Seq Scan on airports_data (cost=0.00..4.04 rows=104 width=...
(3 rows)
```

n개의 값을 정렬하는 계산 복잡도는 $O(n\log_2 n)$으로 알려져 있습니다. 단일 비교 연산은 `cpu_operator_cost`(기본값: 0.0025) 값의 두 배로 추정됩니다. 전체 데이터 세트는 결과를 검색하기 전에 스캔되고 정렬되어야 하므로, 정렬의 시작 비용에는 자식 노드의 전체 비용과 비교 연산에 의해 발생하는 모든 비용이 포함됩니다.

정렬의 총 비용에는 반환될 각 행을 처리하는 비용도 포함되며, 이는 `cpu_operator_cost`에서 추정되며(통상적인 `cpu_tuple_cost` 값이 아닌, `Sort` 노드에 의해 발생하는 오버헤드가 미미하기 때문입니다).[349]

이 예제의 비용은 다음과 같이 계산됩니다:

```
=> WITH costs(startup) AS (
 SELECT 4.04 + round((current_setting('cpu_operator_cost')::real * 2 * 104 * log(2, 104)
                      )::numeric, 2)
)
SELECT startup,
  startup + round(( current_setting('cpu_operator_cost')::real * 104 )::numeric, 2) AS
      total
FROM costs;
   startup ¦ total
```

[348] backend/utils/sort/tuplesort.c

[349] backend/optimizer/path/costsize.c, cost_sort function

```
---------+-------
    7.52 ¦ 7.78
(1 row)
```

Top-N 힙정렬

데이터 세트가 부분적으로만 정렬되어야 하는 경우(LIMIT 절에 의해 정의됨), 힙 정렬(heapsort) 방법이 적용될 수 있습니다(계획에서는 top-N heapsort로 표현됩니다). 더 정확히 말하자면, 이 알고리즘은 정렬이 행의 수를 적어도 절반으로 줄이거나 할당된 메모리가 전체 입력 세트를 수용할 수 없는 경우(반면 출력 세트는 수용 가능한 경우)에 사용됩니다.

```
=> EXPLAIN (analyze, timing off, summary off)
SELECT * FROM seats
ORDER BY seat_no LIMIT 100;
                QUERY PLAN
------------------------------------------------------------------------
  Limit (cost=72.57..72.82 rows=100 width=15)
      (actual rows=100 loops=1)
      -> Sort (cost=72.57..75.91 rows=1339 width=15)
              (actual rows=100 loops=1)
              Sort Key: seat_no
              Sort Method: top-N heapsort Memory: 33kB
              -> Seq Scan on seats (cost=0.00..21.39 rows=1339 width=15)
                      (actual rows=1339 loops=1)
(8 rows)
```

n개 중에서 k개의 가장 높은(또는 낮은) 값을 찾기 위해, 실행자는 처음 k개의 행을 힙이라고 불리는 데이터 구조에 추가합니다. 그런 다음 나머지 행들이 하나씩 추가되고, 각 반복 후에는 힙에서 가장 작은(또는 가장 큰) 값이 제거됩니다. 모든 행이 처리되면, 힙에는 찾고자 하는 k개의 값이 포함됩니다.

> 여기서 '힙(heap)'이라는 용어는 잘 알려진 데이터 구조를 지칭하며, 종종 같은 이름으로 언급되는 데이터베이스 테이블과는 관련이 없습니다.

비용 추정. 이 알고리즘의 계산 복잡도는 $O(n \log_2 k)$로 추정되지만, 각각의 특정 연산은 퀵소트 알고리즘에 비해 더 비쌉니다. 따라서 공식은 $n \log_2 2k$를 사용합니다.[350]

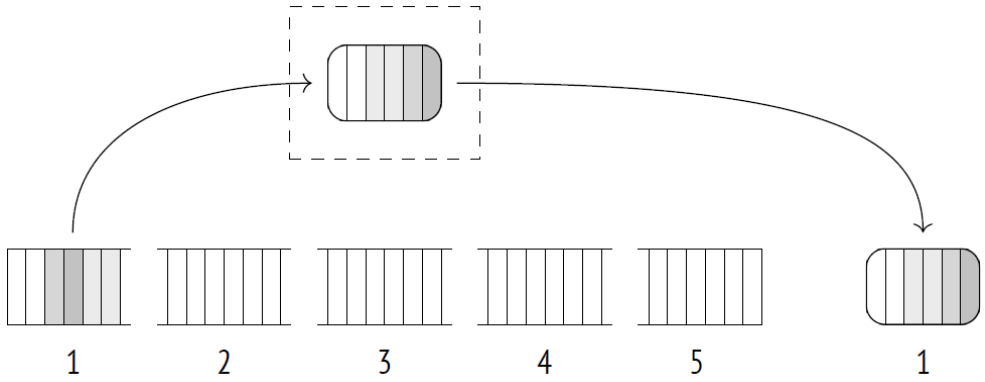
```
=> WITH costs(startup)
AS (
   SELECT 21.39 + round((
      current_setting('cpu_operator_cost')::real * 2 * 1339 * log(2, 2 * 100)
      )::numeric, 2)
)
SELECT startup,
```

[350] backend/optimizer/path/costsize.c, cost_sort function

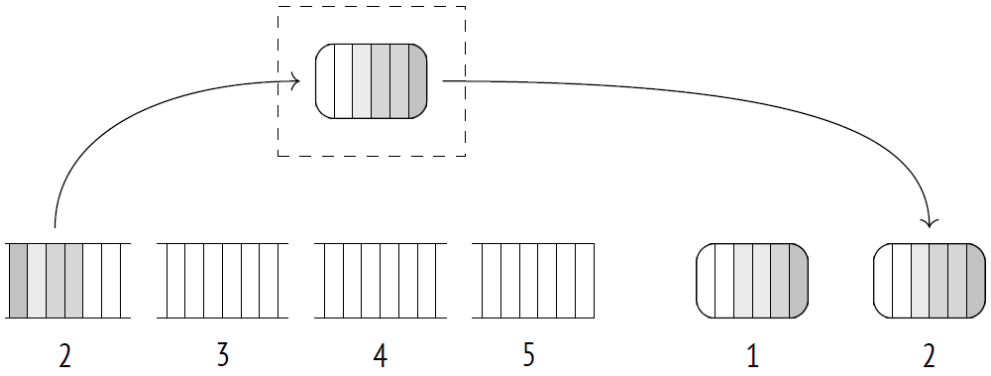
```
startup + round(( current_setting('cpu_operator_cost')::real * 100 )::numeric, 2) AS
  total
FROM costs;
 startup ¦ total
---------+-------
   72.57 ¦ 72.82
(1 row)
```

외부 정렬

스캔이 데이터 세트가 메모리 내에서 정렬하기에 너무 크다는 것을 나타내면, 정렬 노드는 외부 병합 정렬(계획에서는 external merge로 표시됨)로 전환됩니다. 이미 스캔된 행들은 퀵소트 알고리즘으로 메모리 내에서 정렬되며 임시 파일로 기록됩니다.



그 다음에는 남은 메모리로 추가 행들을 읽어들이고, 이 절차를 모든 데이터가 여러 개의 사전 정렬된 파일로 기록될 때까지 반복합니다.



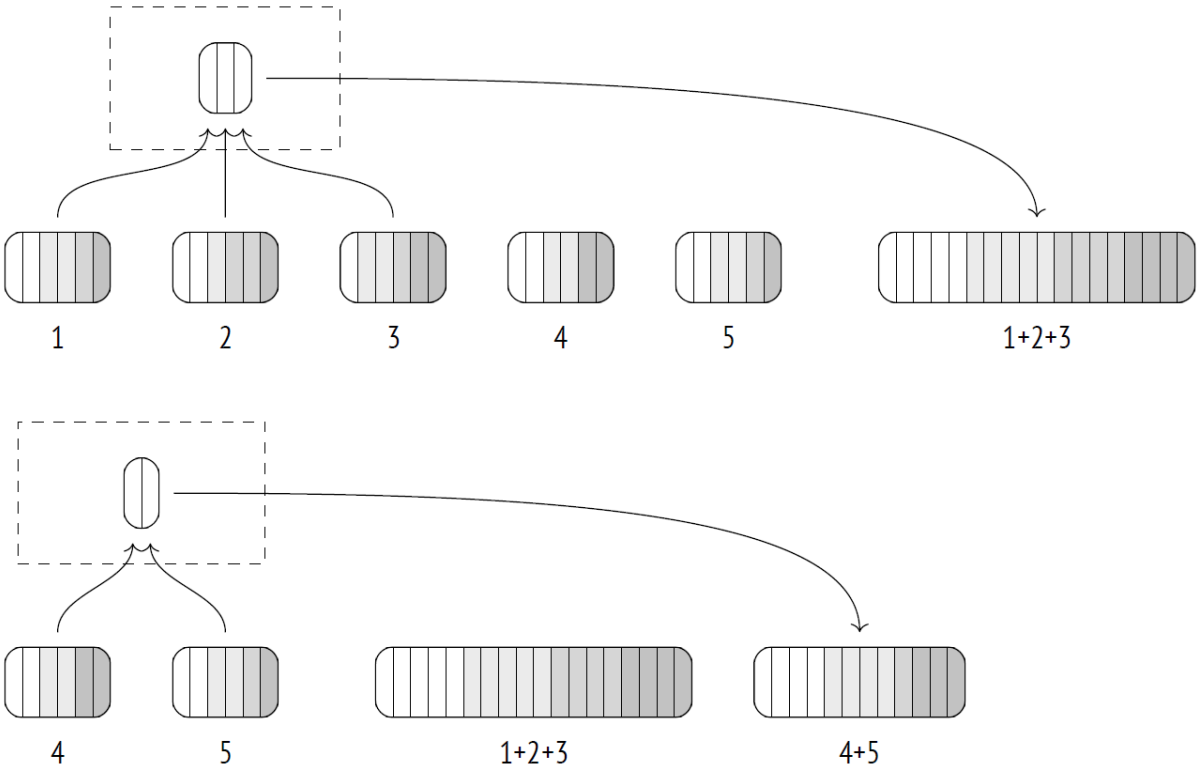
다음으로, 이 파일들은 하나로 병합됩니다. 이 작업은 병합 조인에 사용되는 알고리즘과 대략적으로 같은 알고리즘으로 수행되지만, 한 번에 두 개 이상의 파일을 처리할 수 있다는 주요 차이점이 있습니다.

병합 연산은 많은 메모리를 필요로 하지 않습니다. 실제로, 파일 당 한 행의 공간만 있으면 충분합니다. 각 파일에서 첫 번째 행이 읽혀지고, 가장 낮은 값(또는 정렬 순서에 따라 가장 높은 값)을 가진 행이 부분 결과로 반환되며, 해방된 메모리는 같은 파일에서 가져온 다음 행으로 채워집니다.

실제로는 행들이 하나씩이 아닌 32페이지의 배치로 읽혀져, I/O 작업의 횟수를 줄입니다. 단일 반복에서 병합되는 파일의 수는 사용 가능한 메모리에 따라 다르지만, 절대로 여섯 개 미만이 되지는 않습니다. 효율성이 너무 많은 파일이 있을 때 고통받기 때문에, 상한선은 또한 제한되어 있습니다(500으로).[351]

> 정렬 알고리즘은 오래전부터 확립된 용어를 가지고 있습니다. 외부 정렬은 원래 자기 테이프를 사용하여 수행되었고, PostgreSQL 은 임시 파일[352]을 제어하는 컴포넌트에 비슷한 이름을 유지하고 있습니다. 부분적으로 정렬된 데이터 세트는 "run"[353]이라고 불립니다. 병합에 참여하는 run 의 수는 "병합 차수(merge order)"라고 불립니다. 이 용어들을 사용하지 않았지만, PostgreSQL 의 코드와 주석을 이해하고 싶다면 알아두면 좋습니다.

정렬된 임시 파일들을 한 번에 모두 병합할 수 없는 경우, 여러 번의 패스를 통해 처리되어야 하며, 그 부분적인 결과들은 새로운 임시 파일들로 기록됩니다. 각 반복은 읽고 쓰여야 할 데이터의 양을 증가시키므로, 사용 가능한 `RAM`이 더 많을수록 외부 정렬이 더 빨리 완료됩니다.



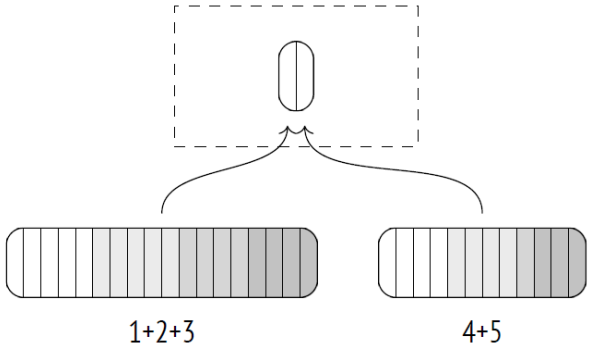
다음 반복에서는 새로 생성된 임시 파일들을 병합합니다.

---

[351] backend/utils/sort/tuplesort.c, tuplesort_merge_order function

[352] backend/utils/sort/logtape.c

[353] *Donald E. Knuth*. The Art of Computer Programming. Volume III. Sorting and Searching



최종 병합은 일반적으로 연기되며 상위 노드가 데이터를 가져올 때 실시간으로 수행됩니다.

외부 정렬에 의해 사용된 디스크 공간이 얼마나 되는지 보려면, `EXPLAIN ANALYZE` 명령어를 실행해 봅시다. `BUFFERS` 옵션은 임시 파일(임시로 읽고 쓴 것)에 대한 버퍼 사용 통계를 보여줍니다. 쓰인 버퍼의 수는 (대략적으로) 읽힌 버퍼의 수와 같을 것이며; 이 값은 킬로바이트로 변환되어 계획에서 '디스크'로 표시됩니다:

```
=> EXPLAIN (analyze, buffers, costs off, timing off, summary off)
SELECT * FROM flights
ORDER BY scheduled_departure;
                      QUERY PLAN
------------------------------------------------------------
  Sort (actual rows=214867 loops=1)
      Sort Key: scheduled_departure
      Sort Method: external merge Disk: 17136kB
      Buffers: shared hit=2627, temp read=2142 written=2150
      -> Seq Scan on flights (actual rows=214867 loops=1)
              Buffers: shared hit=2624
(6 rows)
```

서버 로그에 임시 파일 사용에 대한 자세한 내용을 출력하려면, `log_temp_files` 매개변수를 활성화할 수 있습니다.

비용 추정. 예를 들어, 외부 정렬을 포함한 같은 계획을 살펴봅시다:

```
=> EXPLAIN SELECT *
FROM flights
ORDER BY scheduled_departure;
                      QUERY PLAN
------------------------------------------------------------------------
  Sort (cost=31883.96..32421.12 rows=214867 width=63)
      Sort Key: scheduled_departure
      -> Seq Scan on flights (cost=0.00..4772.67 rows=214867 width=63)
(3 rows)
```

여기서 비교의 정규 비용(메모리 내 퀵소트 작업의 경우와 동일한 수)은 `I/O` 비용으로 확장됩니다.[354] 모든 입력 데이터는 먼저 디스크의 임시 파일로 쓰여져야 하며, 그 다음 병합 작업 동안(만약 생성된 모든 파일이 한 번의 반복으로 병합될 수 없다면 여러 번 가능성이 있음) 디스크에서 읽혀져야 합니다.

디스크 작업(읽기 및 쓰기 모두)의 3/4이 순차적이라고 가정하며, 1/4은 랜덤입니다.

디스크에 쓰여진 데이터의 양은 정렬되어야 하는 행의 수와 쿼리에서 사용된 컬럼의 수에 따라 달라집니다.[355] 이 예제에서, 쿼리는 flights 테이블의 모든 컬럼을 표시하므로, 튜플과 페이지 메타데이터를 고려하지 않는다면 디스크에 쏟아진 데이터의 크기는 거의 전체 테이블의 크기와 같습니다(2624 페이지 대신 2309 페이지).

여기서 정렬은 한 번의 반복으로 완료됩니다.

따라서, 이 계획에서 정렬 비용은 다음과 같이 추정됩니다:

```
=> WITH costs(startup) AS (
  SELECT 4772.67 + round((
      current_setting('cpu_operator_cost')::real * 2 * 214867 * log(2, 214867) +
      (current_setting('seq_page_cost')::real * 0.75 +
      current_setting('random_page_cost')::real * 0.25) * 2 * 2309 * 1 -- one iteration
      )::numeric, 2)
)
SELECT startup,
      startup + round((current_setting('cpu_operator_cost')::real * 214867
                      )::numeric, 2) AS total
FROM costs;
  startup ¦ total
----------+----------
 31883.96 ¦ 32421.13
(1 row)
```

증분 정렬

데이터 세트가 키 $K_1 \cdots K_m \cdots K_n$에 의해 정렬되어야 하고, 이 데이터 세트가 첫 번째 $_m$개의 키에 의해 이미 정렬되어 있는 것으로 알려져 있다면, 처음부터 다시 정렬할 필요가 없습니다. 대신, 이 세트를 같은 첫 번째 키 $K_1 \cdots K_m$에 의해 그룹으로 나눌 수 있습니다(이 그룹들에서는 이미 정의된 순서를 따르고 있음), 그리고 나서 이러한 각 그룹을 나머지 $K_{m+1} \cdots K_n$ 키에 의해 별도로 정렬할 수 있습니다. 이 방법을 증분 정렬이라고 합니다.

증분 정렬은 다른 정렬 알고리즘보다 메모리 사용량이 적으며, 세트를 여러 개의 작은 그룹으로 나누기 때문에, 또한 전체 세트가 정렬될 때까지 기다리지 않고 첫 번째 그룹이 처리된 후에 실행자가 결과를 반환하기 시작할 수 있게 합니다.

---

[354] backend/optimizer/path/costsize.c, cost_sort function

[355] backend/optimizer/path/costsize.c, relation_byte_size function

PostgreSQL에서는 구현이 조금 더 미묘합니다:[356] 상대적으로 큰 행 그룹은 별도로 처리되지만, 더 작은 그룹들은 함께 결합되어 전체적으로 정렬됩니다. 이는 정렬 절차를 호출하는 데 드는 오버헤드를 줄입니다. [357]

실행 계획에서는 증분 정렬을 `Incremental Sort` 노드로 표시합니다.

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM bookings
ORDER BY total_amount, book_date;
                      QUERY PLAN
------------------------------------------------------------------
  Incremental Sort (actual rows=2111110 loops=1)
      Sort Key: total_amount, book_date
      Presorted Key: total_amount
      Full-sort Groups: 2823 Sort Method: quicksort Average
      Memory: 30kB Peak Memory: 30kB
      Pre-sorted Groups: 2624 Sort Method: quicksort Average
      Memory: 3152kB Peak Memory: 3259kB
      -> Index Scan using bookings_total_amount_idx on bookings (ac...
(8 rows)
```

계획에서 보여주듯이, 데이터 세트는 `total_amount` 필드에 의해 사전 정렬되어 있습니다. 이는 이 컬럼에서 실행된 인덱스 스캔의 결과입니다(Presorted Key). `EXPLAIN ANALYZE` 명령어는 또한 실행 시간 통계를 표시합니다. Full-sort Groups 행은 전체적으로 정렬되기 위해 합쳐진 작은 그룹들과 관련이 있으며, Presorted Groups 행은 부분적으로 정렬된 데이터를 가진 큰 그룹들에 대한 데이터를 표시하며, 이는 book_date 컬럼에 대해서만 증분 정렬이 필요했습니다. 두 경우 모두, 메모리 내 퀵정렬 방법이 적용되었습니다. 그룹 크기의 차이는 예약 비용의 균일하지 않은 분포 때문입니다.

증분 정렬은 윈도우 함수를 계산하는 데에도 사용될 수 있습니다:

```
=> EXPLAIN (costs off)
SELECT row_number() OVER (ORDER BY total_amount, book_date)
FROM bookings;
                      QUERY PLAN
------------------------------------------------------------------
  WindowAgg
      -> Incremental Sort
              Sort Key: total_amount, book_date
              Presorted Key: total_amount
              -> Index Scan using bookings_total_amount_idx on bookings
(5 rows)
```

[356] backend/executor/nodeIncrementalSort.c

[357] backend/utils/sort/tuplesort.c

비용 추정. 증분 정렬에 대한 비용 계산[358]은 예상되는 그룹 수와 평균 크기의 그룹을 정렬하는 예상 비용을 기반[359]으로 합니다(이미 검토한 내용입니다).

시작 비용은 첫 번째 그룹을 정렬하는 비용 추정을 반영하며, 이를 통해 노드가 정렬된 행을 반환하기 시작할 수 있습니다; 총 비용은 모든 그룹의 정렬 비용을 포함합니다.

여기서 이러한 계산을 더 탐구하지는 않을 것입니다.

병렬 모드

정렬은 동시에 수행될 수도 있습니다. 하지만 병렬 작업자가 데이터 분할을 사전에 정렬하더라도 Gather 노드는 그들의 정렬 순서에 대해 전혀 알지 못하고, 오직 먼저 도착한 순서대로 그들을 쌓아갈 수만 있습니다. 정렬 순서를 유지하기 위해서는 실행자가 `Gather Merge` 노드를 적용해야 합니다.[360]

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT *
FROM flights
ORDER BY scheduled_departure
LIMIT 10;
                      QUERY PLAN
------------------------------------------------------------------------
  Limit (actual rows=10 loops=1)
      -> Gather Merge (actual rows=10 loops=1)
              Workers Planned: 1
              Workers Launched: 1
              -> Sort (actual rows=7 loops=2)
                      Sort Key: scheduled_departure
                      Sort Method: top-N heapsort Memory: 27kB
                      Worker 0: Sort Method: top-N heapsort Memory: 27kB
                      -> Parallel Seq Scan on flights (actual rows=107434 lo...
(9 rows)
```

`Gather Merge` 노드는 여러 작업자에 의해 가져온 행의 순서를 조정하기 위해 이진 힙[361]을 사용합니다. 이는 외부 정렬이 수행하는 것처럼 여러 개의 정렬된 행 집합을 가상으로 병합하지만, 다른 사용 사례를 위해 설계되었습니다: `Gather Merge`는 일반적으로 소수의 고정된 데이터 소스를 다루며, 블록 단위가 아닌 한 번에 한 행씩 가져옵니다.

비용 추정. `Gather Merge` 노드의 시작 비용은 자식 노드의 시작 비용을 기반으로 합니다. Gather 노드와 마찬가지로, 이 값은 병렬 프로세스를 시작하는 비용(`parallel_setup_cost`(기본값: 1000))에 의해 증가됩니다.

---

[358] backend/optimizer/path/costsize.c, cost_incremental_sort function

[359] backend/utils/adt/selfuncs.c, estimate_num_groups function

[360] backend/executor/nodeGatherMerge.c

[361] backend/lib/binaryheap.c

받은 값은 이진 힙을 구성하는 비용으로 더 확장되며, 이는 n개의 값 정렬을 요구합니다. 여기서 n은 병렬 작업자의 수입니다(즉, $n\log_2 n$). 단일 비교 연산은 `cpu_operator_cost`(기본값: 0.0025)의 두 배로 추정됩니다. 그러나 n이 상당히 작기 때문에 이러한 연산의 전체 비율은 일반적으로 무시할 수 있습니다.

총 비용은 병렬 부분 계획을 수행하는 여러 프로세스에 의해 모든 데이터를 가져오는 데 발생하는 비용과 이 데이터를 리더에게 전송하는 비용을 포함합니다. 단일 행 전송은 `parallel_tuple_cost`(기본값: 0.1)에 5%를 추가하여 다음 값들을 가져오는 동안의 대기를 보상하기 위해 증가된 것으로 추정됩니다.

이진 힙 업데이트에 의해 발생하는 비용도 총 비용 계산에 포함해야 합니다: 각 입력 행은 $\log_2 n$ 비교 연산과 일정한 추가 행동을 요구합니다(이는 `cpu_operator_cost`로 추정됩니다).[362]

이제 `Gather Merge` 노드를 사용하는 또 다른 계획을 살펴보겠습니다. 여기서 작업자들은 먼저 해싱을 통해 부분 집계를 수행한 다음, Sort 노드가 받은 결과를 정렬합니다(집계 후 남은 행이 적기 때문에 비용이 저렴합니다)하여 리더 프로세스에게 전달되고, 이 리더 프로세스는 `Gather Merge` 노드에서 전체 결과를 모읍니다. 최종 집계는 정렬된 값 리스트에서 수행됩니다:

```
=> EXPLAIN SELECT amount, count(*)
FROM ticket_flights
GROUP BY amount;
                        QUERY PLAN
------------------------------------------------------------------------
  Finalize GroupAggregate (cost=123399.62..123485.00 rows=337 wid...
      Group Key: amount
      -> Gather Merge (cost=123399.62..123478.26 rows=674 width=14)
              Workers Planned: 2
              -> Sort (cost=122399.59..122400.44 rows=337 width=14)
                      Sort Key: amount
                      -> Partial HashAggregate (cost=122382.07..122385.44 r...
                              Group Key: amount
                              -> Parallel Seq Scan on ticket_flights (cost=0.00...
(9 rows)
```

여기서 우리는 리더를 포함하여 세 개의 병렬 프로세스를 가지고 있으며, `Gather Merge` 노드의 비용은 다음과 같이 계산됩니다:

```
=> WITH costs(startup, run) AS (
  SELECT round((
  -- launching processes
      current_setting('parallel_setup_cost')::real +
  -- building the heap
      current_setting('cpu_operator_cost')::real * 2 * 3 * log(2, 3)
  )::numeric, 2),
  round((
```

[362] backend/optimizer/path/costsize.c, cost_gather_merge function

```
  -- passing rows
      current_setting('parallel_tuple_cost')::real * 1.05 * 674 +
  -- updating the heap
      current_setting('cpu_operator_cost')::real * 2 * 674 * log(2, 3) +
      current_setting('cpu_operator_cost')::real * 674
  )::numeric, 2)
)
SELECT 122399.59 + startup AS startup,
       122400.44 + startup + run AS total
FROM costs;
   startup ¦ total
-----------+-----------
 123399.61 ¦ 123478.26
(1 row)
```

23.3 고유한 값 및 그룹화

우리가 방금 보았듯이, 집계를 수행하고 중복을 제거하기 위해 값들을 그룹화하는 작업은 해싱뿐만 아니라 정렬을 통해서도 수행될 수 있습니다. 정렬된 리스트에서는 중복되는 값들의 그룹을 한 번의 패스로 식별할 수 있습니다.

정렬된 리스트에서 고유한 값을 검색하는 것은 계획에서 매우 간단한 노드인 `'Unique'`에 의해 나타납니다:[363]

```
=> EXPLAIN (costs off) SELECT DISTINCT book_ref
FROM bookings
ORDER BY book_ref;
                      QUERY PLAN
------------------------------------------------------------
  Result
  -> Unique
      -> Index Only Scan using bookings_pkey on bookings
(3 rows)
```

집계는 '`GroupAggregate`' 노드에서 수행됩니다:[364]

```
=> EXPLAIN (costs off) SELECT book_ref, count(*)
FROM bookings
GROUP BY book_ref
ORDER BY book_ref;
                      QUERY PLAN
-----------------------------------------------------
  GroupAggregate
```

[363] backend/executor/nodeUnique.c

[364] backend/executor/nodeAgg.c, agg_retrieve_direct function

```
      Group Key: book_ref
      -> Index Only Scan using bookings_pkey on bookings
(3 rows)
```

병렬 계획에서, 이 노드는 'Partial GroupAggregate'로 불리며, 집계를 완료하는 노드는 'Finalize GroupAggregate'로 불립니다.

해싱 및 정렬 전략은 GROUPING SETS, CUBE 또는 ROLLUP 절에서 지정된 여러 열 집합을 통해 그룹화가 수행될 경우 단일 노드에서 결합될 수 있습니다. 이 알고리즘의 다소 복잡한 세부 사항에 들어가지 않고, 메모리가 부족한 조건에서 세 가지 다른 열로 그룹화를 수행하는 예를 간단히 제공하겠습니다:

```
=> SET work_mem = '64kB';
=> EXPLAIN (costs off) SELECT count(*)
FROM flights
GROUP BY GROUPING SETS (aircraft_code, flight_no, departure_airport);
      QUERY PLAN
------------------------------
  MixedAggregate
      Hash Key: departure_airport
      Group Key: aircraft_code
      Sort Key: flight_no
              Group Key: flight_no
      -> Sort
              Sort Key: aircraft_code
              -> Seq Scan on flights
(8 rows)
=> RESET work_mem;
```

이 쿼리가 실행되는 동안 일어나는 일입니다. 계획에서 MixedAggregate로 표시된 집계 노드는 aircraft_code 열에 의해 정렬된 데이터 세트를 받습니다.

먼저, 이 데이터 세트는 스캔되며, 값들은 aircraft_code 열에 의해 그룹화됩니다(그룹 키). 스캔이 진행됨에 따라, 행들은 flight_no 열에 따라 재정렬됩니다(메모리가 충분하면 퀵소트 방법을 통해, 또는 디스크 상의 외부 정렬을 사용하여 일반 Sort 노드가 하는 것처럼); 동시에, 실행자는 이 행들을 departure_airport를 키로 사용하는 해시 테이블에 넣습니다(메모리 내에서, 또는 임시 파일을 사용하여 해시 집계가 하는 것처럼).

두 번째 단계에서, 실행자는 방금 flight_no 열에 의해 정렬된 데이터 세트를 스캔하고 동일한 열에 의해 값들을 그룹화합니다(정렬 키와 중첩된 그룹 키 노드). 만약 행들이 또 다른 열에 의해 그룹화되어야 한다면, 필요에 따라 다시 정렬될 것입니다.

마지막으로, 첫 번째 단계에서 준비된 해시 테이블이 스캔되고, 값들은 departure_airport 열에 의해 그룹화됩니다(해시 키).

23.4 조인 방법 비교

우리가 보았듯이, 두 데이터 세트는 세 가지 다른 방법을 사용하여 조인될 수 있으며, 각 방법은 자신만의 장단점을 가지고 있습니다.

중첩 루프 조인(nested loop join)은 사전 준비가 필요 없으며, 결과 세트의 첫 번째 행을 바로 반환할 수 있습니다. 이는 내부 세트를 완전히 스캔할 필요가 없는 유일한 조인 방법입니다(인덱스 접근이 가능한 경우). 이러한 특성은 상대적으로 작은 행 세트를 다루는 짧은 OLTP 쿼리에 이상적인 선택으로 중첩 루프 알고리즘(인덱스와 결합됨)을 만듭니다.

데이터 볼륨이 커짐에 따라 중첩 루프의 약점이 분명해집니다. 카테시안 곱(Cartesian product)의 경우, 이 알고리즘은 이차 복잡도(quadratic complexity)를 가집니다—비용은 조인되는 데이터 세트 크기의 곱에 비례합니다. 그러나, 실제로 카테시안 곱은 그리 흔하지 않습니다; 외부 세트의 각 행에 대해, 실행자는 일반적으로 인덱스를 사용하여 내부 세트의 특정 수의 행에 접근하며, 이 평균 숫자는 데이터 세트의 전체 크기에 의존하지 않습니다(예를 들어, 예약에 있는 평균 티켓 수는 예약 및 구매된 티켓 수가 증가함에 따라 변하지 않습니다). 따라서, 중첩 루프 알고리즘의 복잡도는 종종 이차적인 성장보다는 선형적인 성장을 보여주곤 합니다, 비록 높은 선형 계수로 말이죠.

중첩 루프 알고리즘의 중요한 차별점은 그것의 보편적 적용 가능성입니다: 모든 조인 조건을 지원하는 반면, 다른 방법들은 등가 조인(equijoins)만을 다룰 수 있습니다. 이는 모든 유형의 조건(전체 조인(full join)은 중첩 루프와 함께 사용할 수 없음을 제외하고)으로 쿼리를 실행할 수 있게 하지만, 대규모 데이터 세트의 비등가 조인(non-equi-join)은 원하는 것보다 훨씬 느리게 수행될 가능성이 매우 높다는 것을 염두에 두어야 합니다.

해시 조인(hash join)은 대규모 데이터 세트에서 가장 잘 작동합니다. `RAM`이 충분하다면, 두 데이터 세트에 대한 한 번의 패스만 필요하므로 그 복잡도는 선형입니다. 순차적 테이블 스캔과 결합될 때, 이 알고리즘은 대량의 데이터를 기반으로 결과를 계산하는 `OLAP` 쿼리에 일반적으로 사용됩니다.

그러나, 응답 시간이 처리량보다 더 중요한 경우, 해시 조인은 최선의 선택이 아닙니다: 전체 해시 테이블이 구축될 때까지 결과 행의 반환을 시작하지 않을 것입니다.

해시 조인 알고리즘은 등가 조인에만 적용 가능합니다. 다른 제한은 조인 키의 데이터 유형이 해싱을 지원해야 한다는 것입니다(그러나 거의 모든 데이터 유형이 그렇습니다).

중첩 루프 조인은 때때로 메모이즈(Memoize) 노드(해시 테이블을 기반으로 함)에서 내부 세트의 행을 캐싱하는 것을 활용하여 해시 조인을 이길 수 있습니다. 해시 조인이 항상 내부 세트를 전체적으로 스캔하는 반면, 중첩 루프 알고리즘은 그럴 필요가 없어 일부 비용 절감을 가져올 수 있습니다.

병합 조인(merge join)은 짧은 `OLTP` 쿼리와 긴 `OLAP` 쿼리 모두를 완벽하게 처리할 수 있습니다. 이는 선형 복잡도를 가지며(조인될 세트는 한 번만 스캔되어야 함), 많은 메모리를 요구하지 않으며, 사전 처리 없이 결과를 반환합니다; 하지만, 데이터 세트는 이미 필요한 정렬 순서를 가지고 있어야 합니다. 가장 비용 효율적인 방법은 인덱스 스캔을 통해 데이터를 가져오는 것입니다. 행 수가 적을 경우 자연스러운 선택이며, 더 큰 데이터 세트의 경우, 인덱스 스캔이 여전히 효율적일 수 있지만, 힙(heap) 접근이 최소화되거나 전혀 발생하지 않을 때만 그렇습니다.

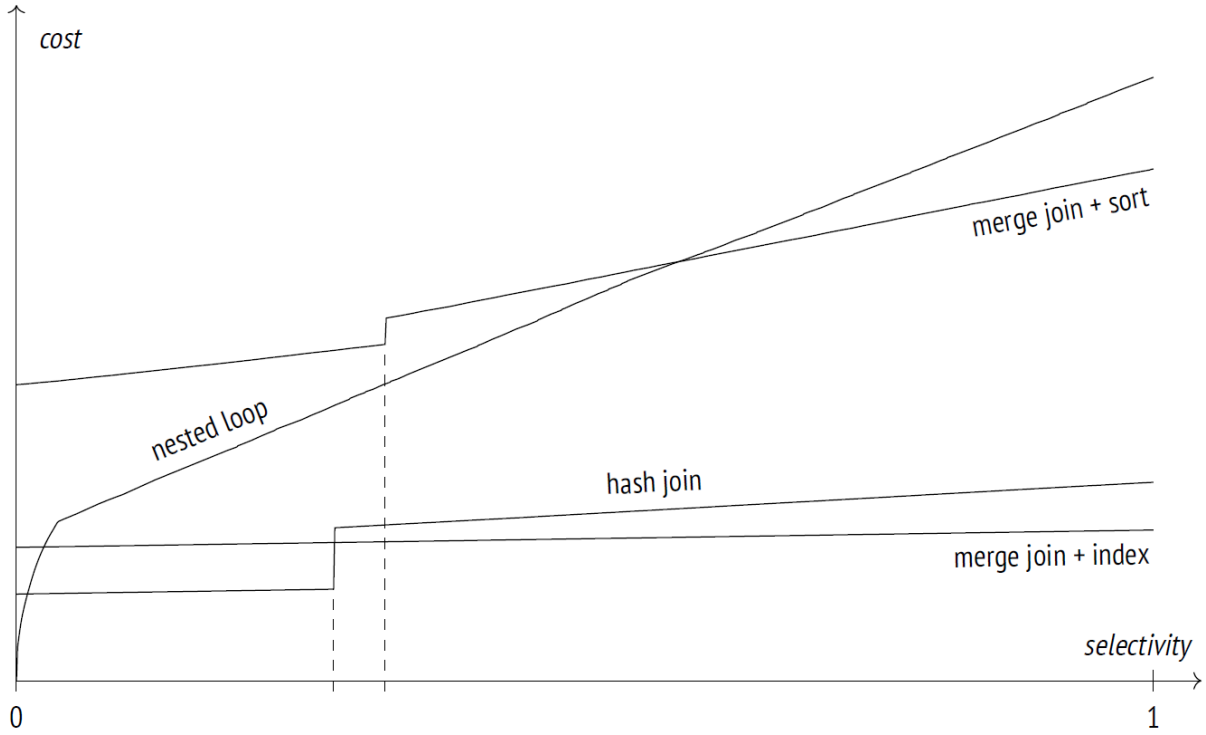
적합한 인덱스가 없는 경우, 세트는 정렬되어야 하지만, 이 작업은 메모리 집약적이며 복잡도가 선형보다 높

습니다: $O(n \log_2 n)$. 이러한 경우, 해시 조인이 병합 조인보다 거의 항상 더 저렴합니다—결과가 정렬되어야 하는 경우를 제외하고.

병합 조인의 추가적인 장점은 내부 및 외부 세트의 동등성입니다. 중첩 루프 및 해시 조인의 효율성은 플래너가 내부 및 외부 세트를 올바르게 할당할 수 있는지 여부에 크게 의존합니다.

병합 조인은 등가 조인(equi-joins)으로 제한됩니다. 또한, 데이터 유형은 B-트리 연산자 클래스를 가져야 합니다.

다음 그래프는 조인되어야 할 행의 비율과 다양한 조인 방법의 비용 사이의 대략적인 의존성을 보여줍니다.



선택성이 높은 경우, 중첩 루프 조인은 두 테이블 모두에 대해 인덱스 접근을 사용합니다; 그 후 플래너는 외부 테이블의 전체 스캔으로 전환하는데, 이는 그래프의 선형 부분에 반영됩니다. 여기서 해시 조인은 두 테이블 모두에 대해 전체 스캔을 사용합니다. 그래프 상의 "계단"은 해시 테이블이 전체 메모리를 채우고 배치가 디스크로 넘쳐흐르기 시작하는 순간에 해당합니다.

인덱스 스캔이 사용되는 경우, 병합 조인의 비용은 작은 선형 성장을 보입니다. `work_mem` 크기가 충분히 큰 경우, 해시 조인이 일반적으로 더 효율적이지만, 임시 파일이 관련될 때 병합 조인이 그것을 이깁니다.

정렬-병합 조인의 상단 그래프는 인덱스가 사용 불가능하고 데이터를 정렬해야 할 때 비용이 증가하는 것을 보여줍니다. 해시 조인의 경우와 마찬가지로, 그래프 상의 "계단"은 메모리가 부족하여 정렬을 위해 임시 파일을 사용해야 함으로 인해 발생합니다.

이는 단지 한 예일 뿐이며, 각각의 특정 사례에서 비용 간의 비율은 다를 것입니다.

Part V

인덱스 종류

24 장. 해쉬

24.1 개요

해시 인덱스[365]는 특정 인덱스 키에 의해 튜플 `ID`(`TID`)를 빠르게 찾을 수 있는 기능을 제공합니다. 대략적으로 말하자면, 그것은 단순히 디스크에 저장된 해시 테이블입니다. 해시 인덱스가 지원하는 유일한 연산은 동등 조건에 의한 검색입니다.

값이 인덱스에 삽입될 때[366], 인덱스 키의 해시 함수가 계산됩니다. PostgreSQL에서 해시 함수는 32비트 또는 64비트 정수를 반환합니다; 이 값들의 여러 가장 낮은 비트들이 해당 버킷의 번호로 사용됩니다.
`TID`와 키의 해시 코드가 선택된 버킷에 추가됩니다. 키 자체는 인덱스에 저장되지 않습니다. 왜냐하면 작고 고정 길이의 값을 다루는 것이 더 편리하기 때문입니다.

인덱스의 해시 테이블은 동적으로 확장됩니다.[367] 버킷의 최소 수는 두 개입니다. 인덱스된 튜플의 수가 증가함에 따라, 버킷 중 하나가 두 개로 분할됩니다. 이 연산은 해시 코드의 한 비트를 더 사용하기 때문에, 요소들은 분할로 인해 생긴 두 버킷 사이에서만 재분배되며, 해시 테이블의 다른 버킷들의 구성은 그대로 유지됩니다.[368]

인덱스 검색 연산[369]은 인덱스 키의 해시 함수와 해당 버킷 번호를 계산합니다. 버킷 내용 중에서, 검색은 키의 해시 코드에 해당하는 `TID`들만 반환할 것입니다. 버킷 요소들이 키의 해시 코드에 따라 정렬되어 있기 때문에, 이진 검색은 일치하는 `TID`들을 효율적으로 반환할 수 있습니다.

키가 해시 테이블에 저장되지 않기 때문에, 인덱스 접근 방식은 해시 충돌로 인해 중복된 `TID`를 반환할 수 있습니다. 따라서, 인덱싱 엔진은 접근 방식으로 가져온 모든 결과를 다시 확인해야 합니다. 같은 이유로 인덱스-만 스캔은 지원되지 않습니다.

24.2 페이지 레이아웃

일반 해시 테이블과 달리, 해시 인덱스는 디스크에 저장됩니다. 따라서, 모든 데이터는 페이지로 정리되어야 하며, 인덱스 연산(검색, 삽입, 삭제)이 가능한 한 적은 페이지에 접근을 요구하도록 하는 방식이 바람직합니다. 해시 인덱스는 네 가지 유형의 페이지를 사용합니다:

- 메타페이지: 인덱스의 "목차"를 제공하는 페이지 제로
- 버킷 페이지: 인덱스의 주요 페이지, 버킷당 하나
- 오버플로 페이지: 주요 버킷 페이지가 모든 요소를 수용할 수 없을 때 사용되는 추가 페이지
- 비트맵 페이지: 해제되어 재사용될 수 있는 오버플로 페이지를 추적하는 데 사용되는 비트 배열이 포함된 페이지

---

365 postgresql.org/docs/14/hash-index.html
backend/access/hash/README

366 backend/access/hash/hashinsert.c

367 backend/access/hash/hashpage.c, _hash_expandtable function

368 backend/access/hash/hashpage.c, _hash_getbucketbuf_from_hashkey function

369 backend/access/hash/hashsearch.c

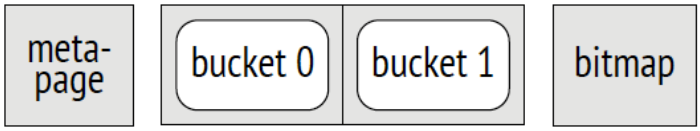
'pageinspect' 확장을 사용하면 인덱스 페이지 내부를 살펴볼 수 있습니다. 빈 테이블로 시작해 보겠습니다:

```
=> CREATE EXTENSION pageinspect;
=> CREATE TABLE t(n integer);
=> ANALYZE t;
=> CREATE INDEX ON t USING hash(n)
```

테이블을 분석했으므로, 생성된 인덱스는 가능한 최소 크기를 가질 것입니다. 그렇지 않았다면, 버킷의 수는 테이블이 10 페이지를 포함하고 있다는 가정에 기반하여 선택되었을 것입니다.[370]

인덱스에는 네 개의 페이지가 포함되어 있습니다: 메타페이지, 두 개의 버킷 페이지, 그리고 하나의 비트맵 페이지(미래 사용을 위해 한 번에 생성됨):

```
=> SELECT page, hash_page_type(get_raw_page('t_n_idx', page))
FROM generate_series(0,3) page;
    page ¦ hash_page_type
─────────+─────────────────
       0 ¦ metapage
       1 ¦ bucket
       2 ¦ bucket
       3 ¦ bitmap
(4 rows)
```



메타페이지에는 인덱스에 관한 모든 제어 정보가 포함되어 있습니다. 현재 우리가 관심 있는 몇 가지 값만 확인해 보겠습니다:

```
=> SELECT ntuples, ffactor, maxbucket
FROM hash_metapage_info(get_raw_page('t_n_idx', 0));
     ntuples ¦      ffactor ¦ maxbucket
─────────────+──────────────+────────────
           0 ¦          307 ¦         1
(1 row)
```

버킷당 예상 행 수는 ffactor 필드에 표시됩니다. 이 값은 블록 크기와 fillfactor(기본값: 75) 저장 매개변수 값에 기반하여 계산됩니다. 데이터 분포가 완전히 균일하고 해시 충돌이 없는 경우 더 높은 fillfactor 값을 사용할 수 있지만, 실제 데이터베이스에서는 페이지 오버플로의 위험을 증가시킵니다.

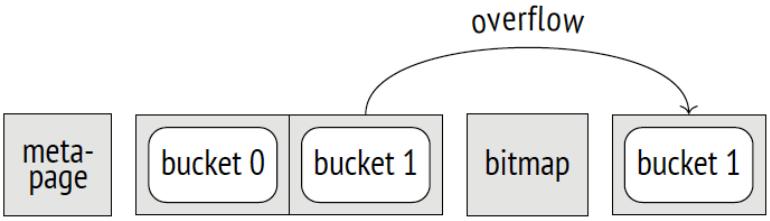
해시 인덱스에 대한 최악의 시나리오는 데이터 분포에 큰 편향이 있을 때입니다. 키가 여러 번 반복되면 해시 함수는 같은 값을 반환하고, 모든 데이터가 동일한 버킷에 배치되므로 버킷 수를 늘려도 도움이 되지 않습니

[370] backend/access/table/tableam.c, table_block_relation_estimate_size function

다.

현재 인덱스는 `ntuples` 필드에 의해 표시된 바와 같이 비어 있습니다. 동일한 인덱스 키 값을 가진 여러 행을 삽입하여 버킷 페이지 오버플로를 일으켜 보겠습니다. 인덱스에 오버플로 페이지가 나타납니다:

```
=> INSERT INTO t(n)
SELECT 0 FROM generate_series(1,500); -- the same value

=> SELECT page, hash_page_type(get_raw_page('t_n_idx', page))
FROM generate_series(0,4) page;
 page | hash_page_type
------+----------------
    0 | metapage
    1 | bucket
    2 | bucket
    3 | bitmap
    4 | overflow
(5 rows)
```



모든 페이지에 대한 종합적인 통계를 보면, 버킷 0은 비어 있고, 모든 값이 버킷 1에 배치되었다는 것을 알 수 있습니다. 일부는 메인 페이지에 위치하고, 나머지는 오버플로 페이지에서 찾을 수 있습니다.

```
=> SELECT page, live_items, free_size, hasho_bucket
FROM (VALUES (1), (2), (4)) p(page),
hash_page_stats(get_raw_page('t_n_idx', page));
    page |       live_items |      free_size | hasho_bucket
-------+--------------+------------+---------------
       1 |                0 |           8148 | 0
       2 |              407 |              8 | 1
       4 |               93 |           6288 | 1
(3 rows)
```

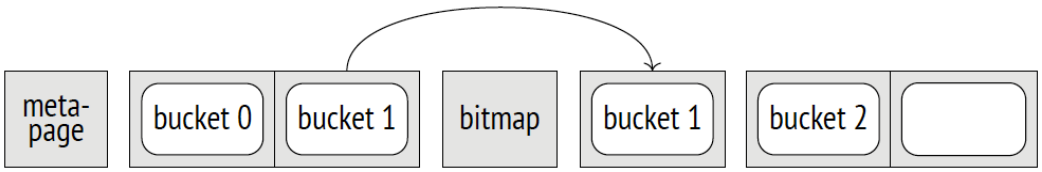
하나의 버킷에 속한 요소들이 여러 페이지에 걸쳐 분포되어 있으면 성능이 저하될 것이 명확합니다. 해시 인덱스는 데이터 분포가 균일할 때 최상의 결과를 보입니다.

이제 버킷이 어떻게 분할되는지 살펴보겠습니다. 이는 인덱스의 행 수가 사용 가능한 버킷에 대해 예상된 `ffactor` 값을 초과할 때 발생합니다. 여기에는 두 개의 버킷이 있고, `ffactor`는 307이므로, 615번째 행이 인덱스에 삽입될 때 발생할 것입니다:

```
=> SELECT ntuples, ffactor, maxbucket, ovflpoint
FROM hash_metapage_info(get_raw_page('t_n_idx', 0));
 ntuples | ffactor | maxbucket | ovflpoint
---------+---------+-----------+-----------
     500 |     307 |         1 | 1
(1 row)

=> INSERT INTO t(n)
SELECT n FROM generate_series(1,115) n; -- now values are different

=> SELECT ntuples, ffactor, maxbucket, ovflpoint
FROM hash_metapage_info(get_raw_page('t_n_idx', 0));
 ntuples | ffactor | maxbucket | ovflpoint
---------+---------+-----------+-----------
     615 |     307 |         2 | 2
(1 row)
```

`maxbucket` 값이 2로 증가되었으므로, 이제 0에서 2까지 번호가 매겨진 세 개의 버킷이 있습니다. 하지만 한 개의 버킷만 추가했음에도 불구하고, 페이지 수는 두 배가 되었습니다:

```
=> SELECT page, hash_page_type(get_raw_page('t_n_idx', page))
FROM generate_series(0,6) page;
    page | hash_page_type
-------+----------------
       0 | metapage
       1 | bucket
       2 | bucket
       3 | bitmap
       4 | overflow
       5 | bucket
       6 | unused
(7 rows)
```



새로운 페이지 중 하나는 버킷 2에 의해 사용되며, 다른 하나는 여전히 비어있고 버킷 3이 나타나는 대로 사용될 것입니다.

```
=> SELECT page, live_items, free_size, hasho_bucket
FROM (VALUES (1), (2), (4), (5)) p(page),
hash_page_stats(get_raw_page('t_n_idx', page));
    page |       live_items |      free_size | hasho_bucket
```

```
-------+-------------+------------+---------------
      1 |          27 |       7608 | 0
      2 |         407 |          8 | 1
      4 |         158 |       4988 | 1
      5 |          23 |       7688 | 2
(4 행)
```

따라서, 운영 체제의 관점에서 해시 인덱스는 돌발적으로 성장하지만, 논리적 관점에서는 해시 테이블이 점진적으로 성장하는 것으로 나타납니다.

이 성장을 어느 정도 평준화하고 한 번에 너무 많은 페이지를 할당하는 것을 피하기 위해, 열 번째 증가부터는 페이지가 모두 한꺼번에 할당되는 대신 네 개의 동등한 배치로 할당됩니다.

메타페이지의 두 가지 추가 필드, 즉 버킷 주소에 대한 세부 정보를 제공하는 가상 비트 마스크는 다음과 같습니다:

```
=> SELECT maxbucket, highmask::bit(4), lowmask::bit(4)
FROM hash_metapage_info(get_raw_page('t_n_idx', 0));
     maxbucket |      highmask | lowmask
-------------+-----------+----------
             2 |          0011 | 0001
(1 행)
```

버킷 번호는 `highmask`에 해당하는 해시 코드 비트로 정의됩니다. 그러나 받은 버킷 번호가 존재하지 않는 경우(즉, `maxbucket`을 초과하는 경우) `lowmask` 비트가 사용됩니다.[371] 이 특정 경우에는 가장 낮은 두 비트를 취하여 0에서 3까지의 값을 얻습니다; 그러나 만약 우리가 3을 얻었다면, 가장 낮은 한 비트만을 취하여 버킷 3 대신 버킷 1을 사용합니다.

크기가 두 배가 될 때마다 새로운 버킷 페이지들은 단일 연속 청크로 할당되며, 오버플로 및 비트맵 페이지는 필요에 따라 이러한 조각들 사이에 삽입됩니다. 메타페이지는 `spares` 배열에 각 청크에 삽입된 페이지 수를 유지하여, 단순한 산술을 사용하여 버킷 번호를 기반으로 주 페이지의 수를 계산할 수 있는 기회를 제공합니다.[372]

이 특정 경우에는 첫 번째 증가 후에 두 페이지(비트맵 페이지와 오버플로 페이지)가 삽입되었지만, 두 번째 증가 후에는 아직 새로운 추가가 이루어지지 않았습니다:

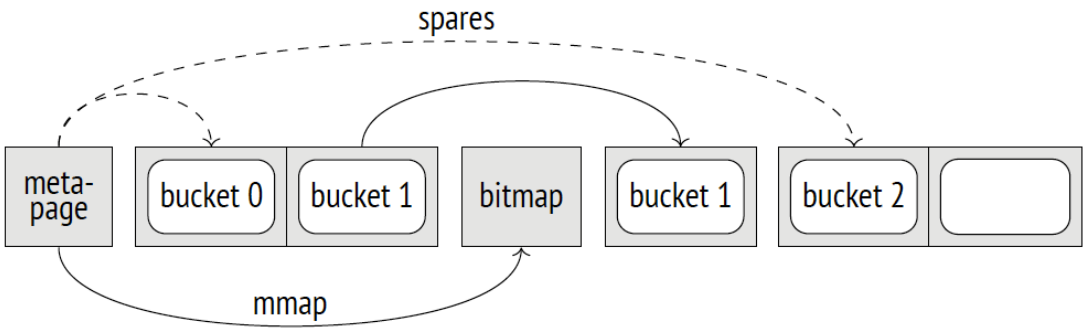
```
=> SELECT spares[2], spares[3]
FROM hash_metapage_info(get_raw_page('t_n_idx', 0));
 spares | spares
--------+--------
      2 | 2
(1 row)
```

371 backend/access/hash/hashutil.c, _hash_hashkey2bucket function

372 include/access/hash.h, BUCKET_TO_BLKNO macro

메타페이지는 또한 비트맵 페이지들로의 포인터 배열을 저장합니다:

```
=> SELECT mapp[1]
FROM hash_metapage_info(get_raw_page('t_n_idx', 0));
  mapp
------
   3
(1 row)
```



죽은 튜플을 가리키는 포인터가 제거될 때 인덱스 페이지 내의 공간이 해제됩니다. 페이지 가지치기(완전히 채워진 페이지에 요소를 삽입하려는 시도로 트리거됨)[373] 동안이나 일상적인 베큠 작업이 수행될 때 일어납니다.

그러나 해시 인덱스는 축소할 수 없습니다. 한번 할당되면 인덱스 페이지는 운영 체제에 반환되지 않습니다. 주 페이지는 어떤 요소도 포함하지 않더라도 자신의 버킷에 영구적으로 할당됩니다. 지워진 오버플로 페이지는 비트맵에 추적되고 재사용될 수 있습니다(다른 버킷에서도 가능). 인덱스의 물리적 크기를 줄이는 유일한 방법은 `REINDEX`나 `VACUUM FULL` 명령을 사용하여 다시 빌드하는 것입니다.

쿼리 계획에는 인덱스 유형에 대한 표시가 없습니다:

```
=> CREATE INDEX ON flights USING hash(flight_no);

=> EXPLAIN (costs off)
SELECT *
FROM flights
WHERE flight_no = 'PG0001';
              QUERY PLAN
------------------------------------------------
  Bitmap Heap Scan on flights
      Recheck Cond: (flight_no = 'PG0001'::bpchar)
      -> Bitmap Index Scan on flights_flight_no_idx
              Index Cond: (flight_no = 'PG0001'::bpchar)
(4 rows)
```

373 backend/access/hash/hashinsert.c, _hash_vacuum_one_page function

24.3 연산자Operator 클래스

PostgreSQL 10 이전에는 해시 인덱스가 로깅되지 않았습니다. 즉, 실패에 대해 보호되지 않았으며 복제되지 않았기 때문에 사용하는 것이 권장되지 않았습니다. 그러나 그럼에도 불구하고 해시 인덱스는 자체적인 가치를 가졌습니다. 해싱 알고리즘은 널리 사용되고 있으며(특히 해시 조인과 그룹화 수행 시), 시스템은 특정 데이터 유형에 대해 어떤 해시 함수를 사용할 수 있는지 알아야 합니다. 그러나 이 대응 관계는 정적이지 않습니다. 즉, 한 번 정의되어 영구적인 것이 아니며, PostgreSQL은 새로운 데이터 유형을 즉석에서 추가할 수 있기 때문에 유지 관리되어야 합니다. 따라서, 이는 해시 인덱스의 연산자 클래스와 특정 데이터 유형에 의해 유지됩니다. 해시 함수 자체는 클래스의 지원 함수로 표현됩니다.

```
=> SELECT opfname AS opfamily_name,
amproc::regproc AS opfamily_procedure
FROM pg_am am
JOIN pg_opfamily opf ON opfmethod = am.oid
JOIN pg_amproc amproc ON amprocfamily = opf.oid
WHERE amname = 'hash'
AND amprocnum = 1
ORDER BY opfamily_name, opfamily_procedure;
      opfamily_name | opfamily_procedure
--------------------+--------------------
        aclitem_ops | hash_aclitem
          array_ops | hash_array
           bool_ops | hashchar
         bpchar_ops | hashbpchar
 bpchar_pattern_ops | hashbpchar
                    ...
         timetz_ops | timetz_hash
           uuid_ops | uuid_hash
           xid8_ops | hashint8
            xid_ops | hashint4
(38 rows)
```

이러한 함수들은 32비트 정수를 반환합니다. 문서화되어 있지 않지만, 해당 유형의 값을 위한 해시 코드를 계산하는 데 사용될 수 있습니다.

예를 들어, text_ops 패밀리는 hashtext 함수를 사용합니다:

```
=> SELECT hashtext('one'), hashtext('two');
   hashtext | hashtext
------------+------------
 1793019229 | 1590507854
(1 row)
```

해시 인덱스의 연산자 클래스는 '동등' 연산자만을 제공합니다:

```
=> SELECT opfname AS opfamily_name,
left(amopopr::regoperator::text, 20) AS opfamily_operator
FROM pg_am am
JOIN pg_opfamily opf ON opfmethod = am.oid
JOIN pg_amop amop ON amopfamily = opf.oid
WHERE amname = 'hash'
ORDER BY opfamily_name, opfamily_operator;
      opfamily_name | opfamily_operator
--------------------+----------------------
        aclitem_ops | =(aclitem,aclitem)
          array_ops | =(anyarray,anyarray)
           bool_ops | =(boolean,boolean)
              ...
           uuid_ops | =(uuid,uuid)
           xid8_ops | =(xid8,xid8)
            xid_ops | =(xid,xid)
(48 rows)
```

24.4 속성

해시 접근 방식이 시스템에 부여하는 인덱스 수준의 속성들을 살펴보겠습니다.

접근 방법 속성

```
=> SELECT a.amname, p.name, pg_indexam_has_property(a.oid, p.name)
FROM pg_am a, unnest(array[
'can_order', 'can_unique', 'can_multi_col',
'can_exclude', 'can_include'
]) p(name)
WHERE a.amname = 'hash';
 amname |          name | pg_indexam_has_property
--------+---------------+-------------------------
   hash |     can_order | f
   hash |    can_unique | f
   hash | can_multi_col | f
   hash |   can_exclude | t
   hash |   can_include | f
(5 rows)
```

해시 인덱스는 행 정렬에 사용될 수 없습니다: 해시 함수는 데이터를 더욱 무작위로 섞습니다.

고유 제약조건도 지원되지 않습니다. 그러나 해시 인덱스는 배타적 제약조건을 적용할 수 있고, 지원되는 유일한 함수가 '동등함'이기 때문에, 이 배타성은 고유성의 의미를 갖게 됩니다:

```
=> ALTER TABLE aircrafts_data
```

```
ADD CONSTRAINT unique_range EXCLUDE USING hash(range WITH =);
=> INSERT INTO aircrafts_data
VALUES ('744','{"ru": "Boeing 747-400"}',11100);
ERROR: conflicting key value violates exclusion constraint
"unique_range"
DETAIL: Key (range)=(11100) conflicts with existing key
(range)=(11100).
```

다중 열 인덱스와 추가적인 `INCLUDE` 열도 지원되지 않습니다.

인덱스 레벨 속성

```
=> SELECT p.name, pg_index_has_property('flights_flight_no_idx', p.name)
FROM unnest(array[
'clusterable', 'index_scan', 'bitmap_scan', 'backward_scan'
]) p(name);
          name ¦ pg_index_has_property
---------------+------------------------
   clusterable ¦ f
    index_scan ¦ t
   bitmap_scan ¦ t
 backward_scan ¦ t
(4 rows)
```

해시 인덱스는 일반 인덱스 스캔과 비트맵 스캔을 모두 지원합니다.

그러나 해시 인덱스를 사용한 테이블 클러스터화는 지원되지 않습니다. 이는 해시 함수 값에 기반하여 힙 데이터를 물리적으로 정렬할 필요성이 상상하기 어렵기 때문에 매우 합리적입니다.

컬럼 레벨 속성

열 수준 속성은 인덱스 접근 방식에 의해 가상으로 정의되며, 항상 동일한 값을 가집니다.

```
=> SELECT p.name,
pg_index_column_has_property('flights_flight_no_idx', 1, p.name)
FROM unnest(array[
'asc', 'desc', 'nulls_first', 'nulls_last', 'orderable',
'distance_orderable', 'returnable', 'search_array', 'search_nulls'
]) p(name);
               name ¦ pg_index_column_has_property
--------------------+-------------------------------
                asc ¦ f
               desc ¦ f
        nulls_first ¦ f
         nulls_last ¦ f
```

```
          orderable ¦ f
 distance_orderable ¦ f
         returnable ¦ f
       search_array ¦ f
       search_nulls ¦ f
(9 rows)
```

해시 함수는 값의 순서를 보존하지 않기 때문에, 순서와 관련된 모든 속성은 해시 인덱스에 적용되지 않습니다. 해시 인덱스는 인덱스 키를 저장하지 않고 힙(heap) 접근이 필요하기 때문에 인덱스-오직(index-only) 스캔에 참여할 수 없습니다.

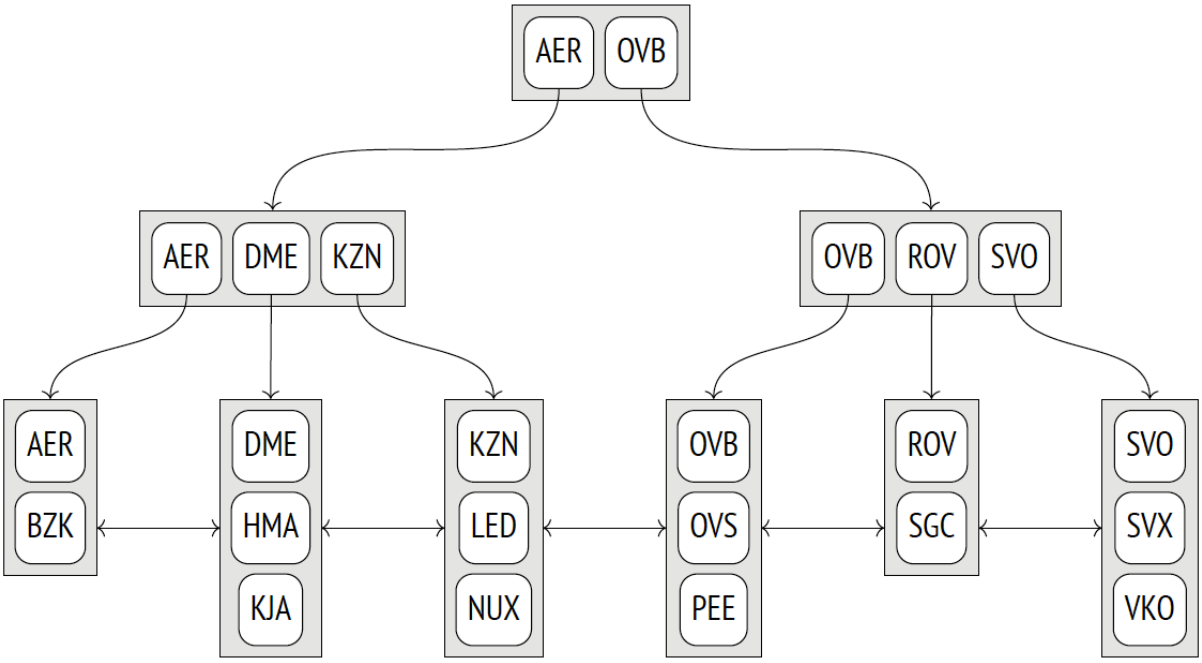
또한, 해시 인덱스는 `NULL` 값에 대해 '동등(equal to)' 연산이 적용될 수 없기 때문에 NULL 값을 지원하지 않습니다.

배열 내의 요소를 검색하는 기능 또한 구현되어 있지 않습니다.

25. B-tree

25.1 개요

B-트리(트리 접근 방식으로 구현됨)는 루트에서 시작하여 리프 노드에서 필요한 요소를 빠르게 찾을 수 있게 해주는 데이터 구조입니다.[374] 검색 경로가 명확하게 식별되려면, 모든 트리 요소가 정렬되어 있어야 합니다. B-트리는 값이 비교되고 정렬될 수 있는 순서형 데이터 타입을 위해 설계되었습니다. 공항 코드에 대한 인덱스를 구축하는 다음의 개략적 다이어그램은 내부 노드를 수평 직사각형으로, 리프 노드는 수직으로 정렬하여 보여줍니다.



각 트리 노드는 인덱스 키와 포인터로 구성된 여러 요소를 포함합니다. 내부 노드 요소는 다음 레벨의 노드를 참조하고, 리프 노드 요소는 힙 튜플을 참조합니다(이러한 참조는 그림에 나타나지 않습니다).

B-트리는 다음과 같은 중요한 특성을 가지고 있습니다:

- 균형이 잡혀 있어서, 트리의 모든 리프 노드가 같은 깊이에 위치합니다. 따라서, 모든 값에 대해 동일한 검색 시간을 보장합니다.
- 많은 분기를 가지고 있으며, 즉 각 노드는 많은 요소를 포함하고 있으며, 종종 수백 개에 이릅니다(명확성을 위해 그림에는 세 요소 노드만 표시됩니다). 결과적으로, B-트리의 깊이는 항상 작으며, 매우 큰 테이블에 대해서도 마찬가지입니다.

  이 구조의 이름에서 B 가 무엇을 의미하는지 절대적으로 확신할 수는 없습니다. 균형잡힌(Balanced)과 덤불 같은(Bushy) 모두 동등하게 잘 어울립니다. 놀랍게도, 종종 이를 이진(Binary)으로 해석하는 것을 볼 수 있는데, 이는 분명히 잘못된 것입니다.

[374] postgresql.org/docs/14/btree.html
backend/access/nbtree/README

- 인덱스 내의 데이터는 오름차순 또는 내림차순으로 정렬되어 있으며, 각 노드 내부뿐만 아니라 같은 레벨의 모든 노드에 걸쳐 정렬됩니다. 동료 노드들은 양방향 리스트로 연결되어 있어서, 루트에서 매번 시작하지 않고 단순히 리스트를 한 방향이나 다른 방향으로 스캔함으로써 정렬된 데이터 세트를 얻을 수 있습니다.

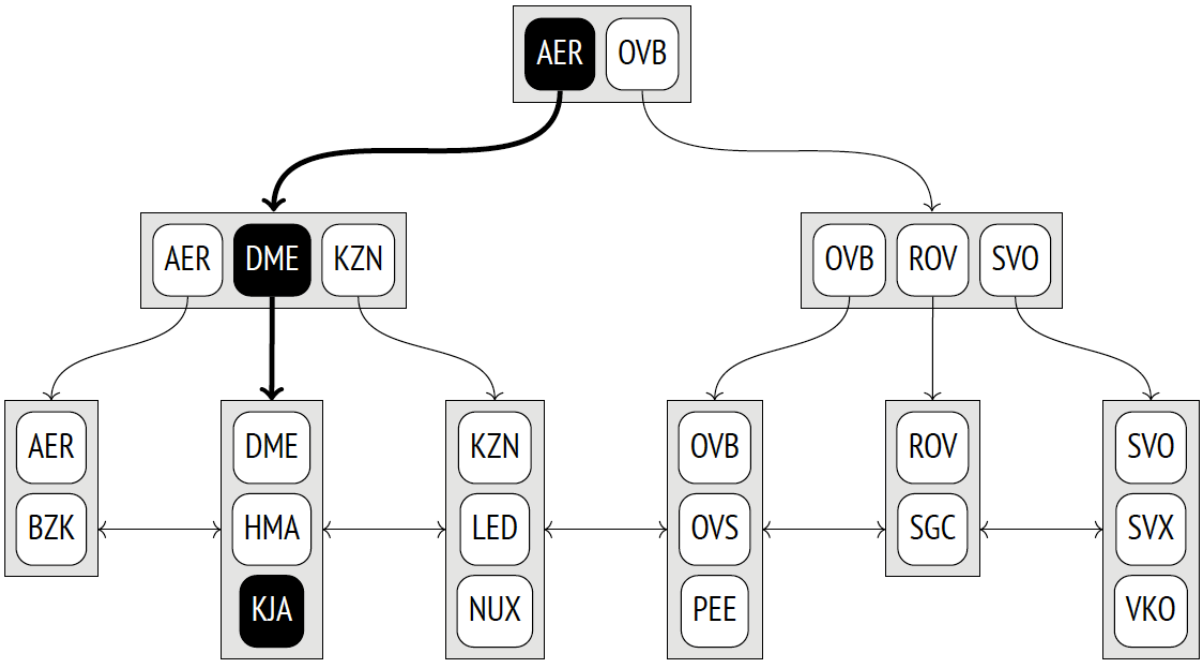
25.2 검색 및 삽입

동일성으로 검색

아래는 "indexedcolumn = expression"[375] 조건으로 트리에서 값을 검색하는 방법을 살펴보겠습니다. 크라스노야르스크의 KJA 공항을 찾아보겠습니다.

검색은 루트 노드에서 시작되며, 액세스 방법은 하위 노드로 하강할 자식 노드를 결정해야 합니다. $K_i \leqslant$ expression $< K_{i+1}$을 만족하는 K_i 키를 선택합니다.

루트 노드에는 AER과 OVB 키가 있습니다. 조건 AER $\leqslant$ KJA $<$ OVB가 참이므로 AER 키가 있는 요소에서 참조하는 자식 노드로 하강해야 합니다.



이 절차는 필요한 튜플 ID를 포함하는 리프 노드에 도달할 때까지 재귀적으로 반복됩니다. 이 경우에 자식 노드는 조건 DME $\leqslant$ KJA $<$ KZN을 충족하므로 DME 키가 있는 요소에서 참조하는 리프 노드로 하강해야 합니다.

알 수 있듯이, 트리의 내부 노드의 최좌측 키는 중복됩니다. 루트의 자식 노드를 선택하기 위해서는 조건 KJA $<$ OVB가 만족되면 충분합니다. B-트리에는 이러한 키가 저장되지 않으므로, 다음 설명의 일러스트레이션에서 해당 요소를 비워둘 것입니다.

[375] backend/access/nbtree/nbtsearch.c, _bt_search function

필요한 요소는 리프 노드에서 이진 검색으로 빠르게 찾을 수 있습니다.

하지만 검색 절차는 보이는 것처럼 단순하지 않습니다. 인덱스의 데이터 정렬 순서는 위의 예시처럼 오름차순일 수도 있고 내림차순일 수도 있다는 것을 고려해야 합니다. 고유 인덱스라도 여러 개의 일치 값이 있을 수 있으며 모두 반환되어야 합니다. 또한, 중복값이 너무 많아 하나의 노드에 담기지 않을 수도 있으므로 인접한 리프 노드도 처리해야 합니다.

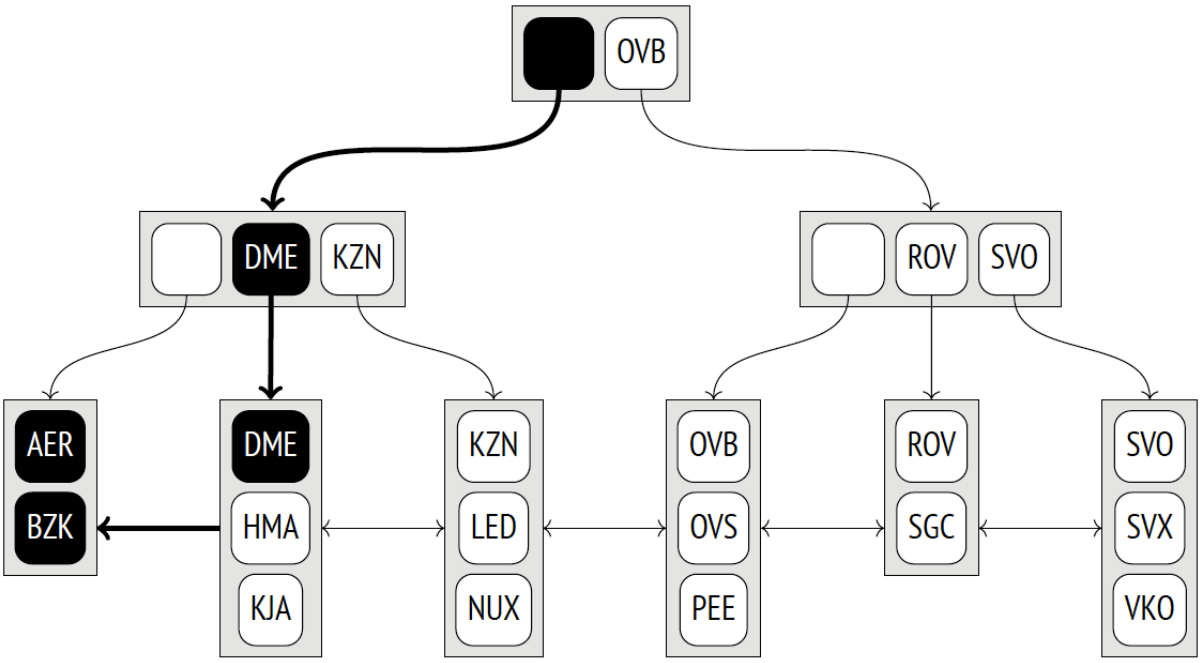
> 인덱스에 고유하지 않은 값이 포함될 수 있으므로 오름차순보다는 비감소 순서, 내림차순보다는 비증가 순서라고 부르는 것이 더 정확할 것입니다. 하지만 더 간단한 용어를 사용하겠습니다. 게다가 튜플 ID 는 인덱스 키의 일부이므로 실제로 값이 동일하더라도 인덱스 항목을 고유한 것으로 간주할 수 있습니다.

게다가 검색이 진행 중일 때 다른 프로세스가 데이터를 수정할 수 있고, 페이지가 두 개로 분할될 수 있으며 트리 구조가 변경될 수 있습니다. 모든 알고리즘은 이러한 동시 작업간의 경쟁을 가능한 한 최소화하고 과도한 잠금을 피하기 위해 설계되어 있지만 여기서는 이 기술적 세부 사항은 다루지 않겠습니다.

불평등 기준으로 검색

"indexed-column ⩽ expression"(또는 "indexed-column ⩾ expression") 조건으로 검색이 수행되면, 먼저 동등 조건을 충족하는 값을 위해 인덱스를 검색한 다음 트리의 끝에 도달할 때까지 필요한 방향으로 그 리프 노드를 횡단해야 합니다.

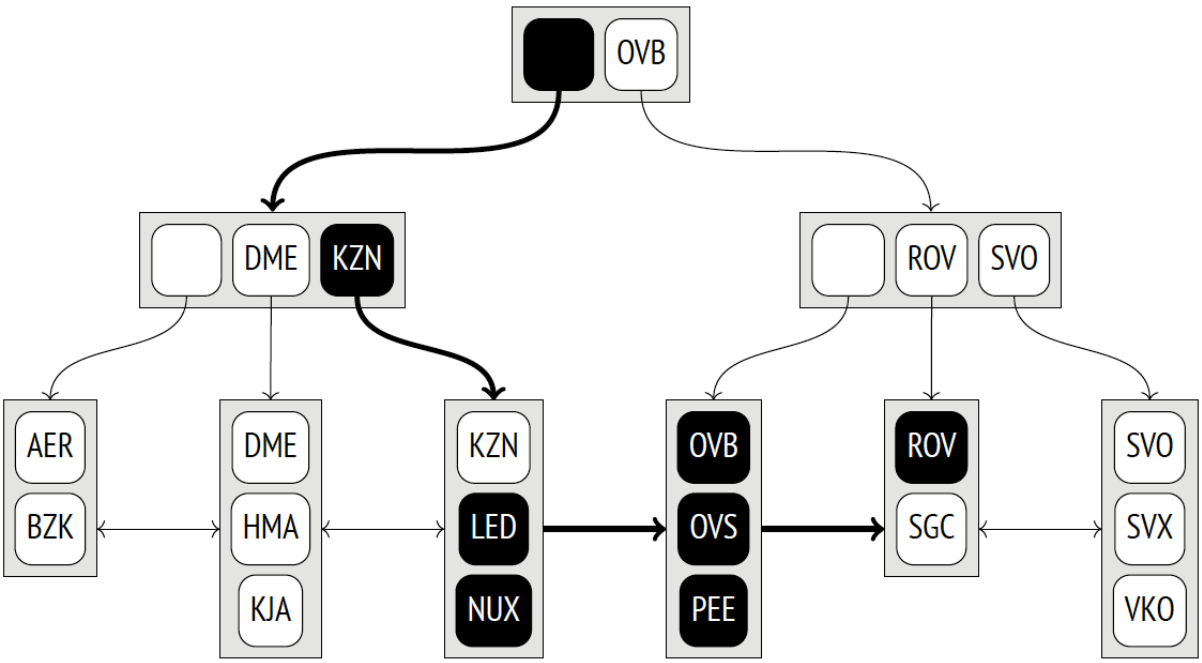
이 다이어그램은 DME(도모데도보)보다 작거나 같은 공항 코드를 검색하는 것을 보여줍니다.



미만 및 초과 연산자의 경우에는 처음 발견된 값을 제외하면 동일한 절차를 따릅니다.

범위로 검색

"expression$_1$ ⩽ indexed-column ⩽ expression$_2$" 범위로 검색할 때, 먼저 expression$_1$을 찾은 다음 expression$_2$에 도달할 때까지 오른쪽 방향으로 리프 노드를 횡단해야 합니다. 이 다이어그램은 LED(상트페테르부르크)와 ROV(로스토프) 사이의 범위(포함)에서 공항 코드를 검색하는 과정을 보여줍니다.

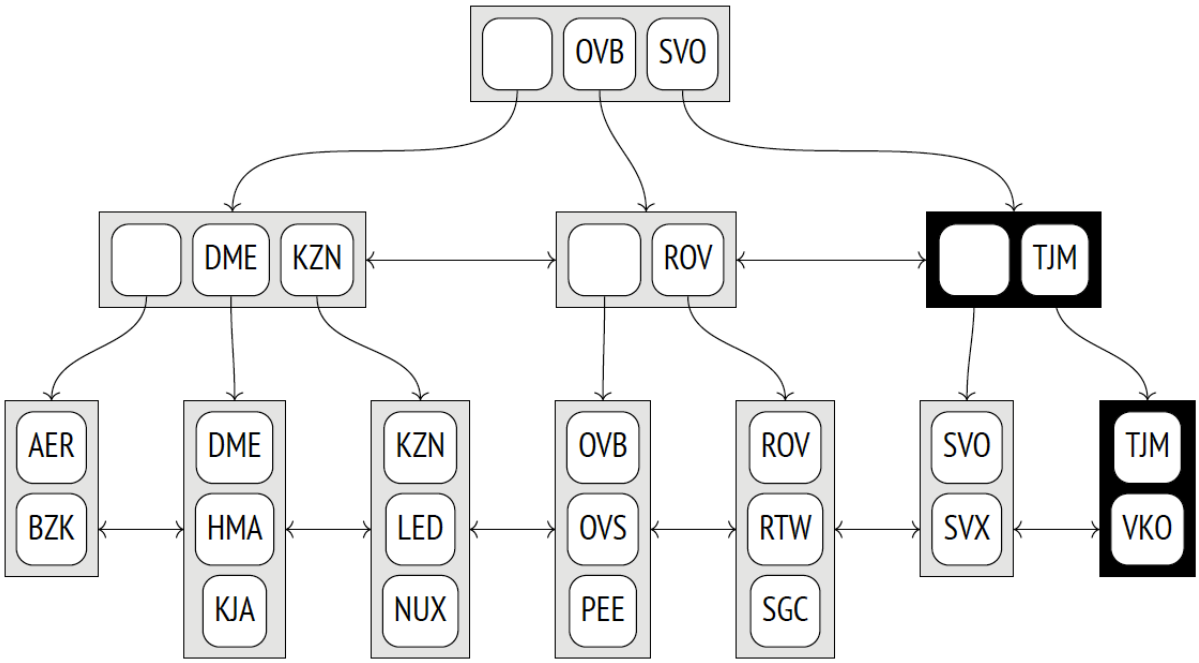


삽입

새 요소의 삽입 위치는 키의 순서에 의해 명확하게 정의됩니다. 예를 들어, RTW(사라토프) 공항 코드를 테이블에 삽입하면, 새 요소는 마지막에서 두 번째 리프 노드인 ROV와 SGC 사이에 나타날 것입니다.

하지만 리프 노드에 새 요소를 위한 공간이 부족한 경우는 어떻게 될까요? 예를 들어(노드가 최대 3개의 요소를 수용할 수 있다고 가정할 때), TJM(튜멘) 공항 코드를 삽입하면 마지막 리프 노드가 가득 차게 됩니다. 이 경우 노드는 두 개로 분할되고, 기존 노드의 일부 요소가 새 노드로 이동되며, 새 자식 노드를 가리키는 포인터가 부모 노드에 추가됩니다. 물론, 부모 노드도 가득 찰 수 있습니다. 그러면 이도 두 개의 노드로 분할되고, 이와 같은 과정이 반복됩니다. 루트를 분할해야 하는 경우, 결과 노드 위에 하나의 노드가 더 생성되어 트리의 새로운 루트가 됩니다. 이 경우 트리의 깊이가 한 레벨 증가합니다.

이 예에서, TJM 공항 삽입으로 인해 두 개의 노드 분할이 발생했습니다. 아래 다이어그램의 새로 생성된 노드가 강조 표시되어 있습니다. 모든 노드를 분할할 수 있도록 보장하기 위해 양방향 연결 리스트가 가장 낮은 레벨 뿐만 아니라 모든 레벨의 노드를 바인딩합니다.

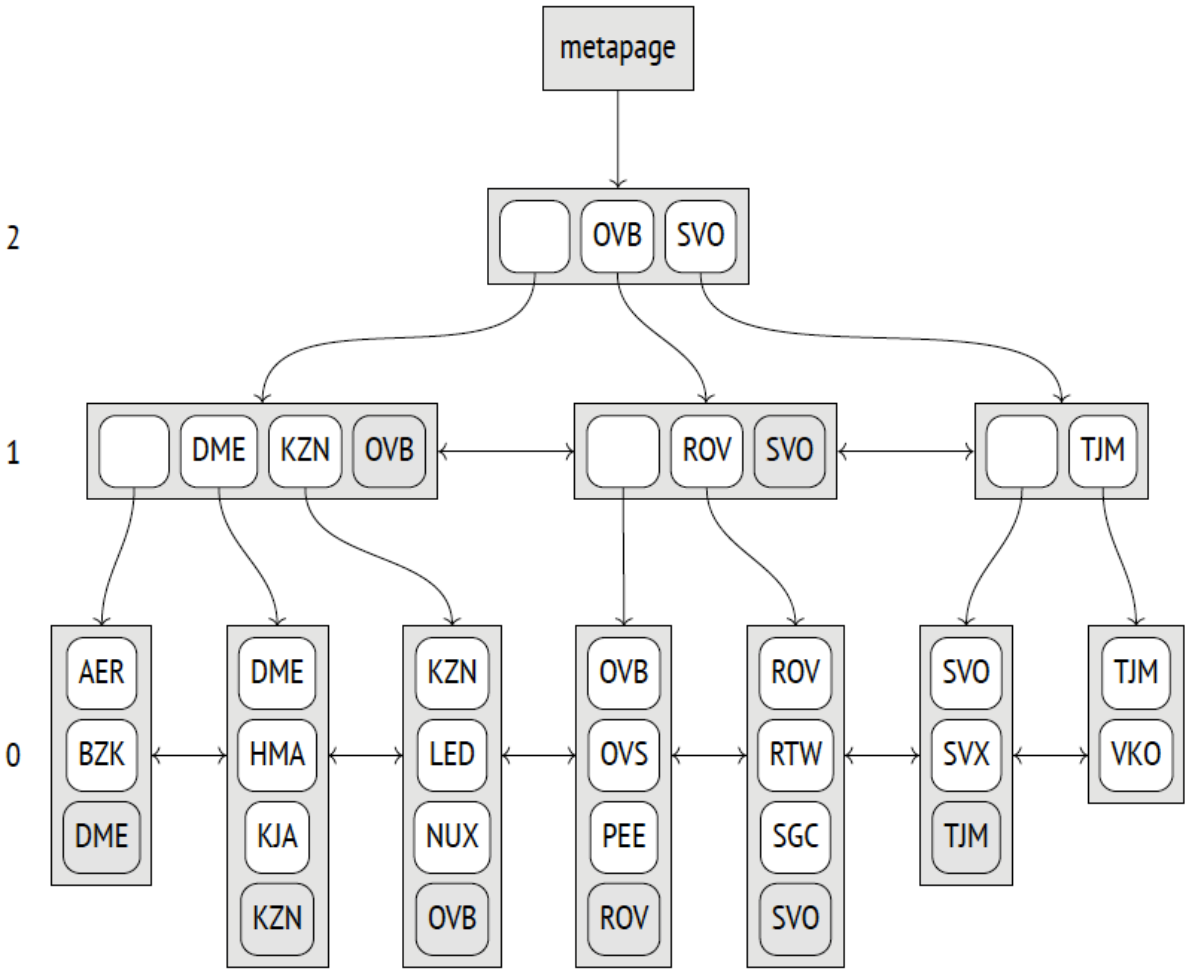


설명한 삽입 및 분할 절차는 트리가 균형을 유지하도록 보장하며, 노드가 수용할 수 있는 요소 수가 일반적으로 아주 크기 때문에 트리 깊이가 드물게 증가합니다.

문제는 한번 분할된 노드는 진공 청소 후에도 요소가 매우 적어도 다시 병합할 수 없다는 것입니다. 이 제한 사항은 B-트리 데이터 구조 자체보다 PostgreSQL 구현에 해당합니다. 따라서 삽입을 시도할 때 노드가 가득 차면 액세스 방법은 먼저 중복된 데이터를 제거하여 공간을 확보하고 추가 분할을 피하려고 시도합니다.

25.3 페이지 구성

B-트리의 각 노드는 하나의 페이지를 차지합니다. 페이지 크기가 노드 용량을 정의합니다. 페이지 분할로 인해 트리의 루트는 시간에 따라 다른 페이지로 표현될 수 있습니다. 하지만 검색 알고리즘은 항상 루트에서 스캔을 시작해야 합니다. 제로 인덱스 페이지(메타페이지라고 함)에서 현재 루트 페이지 ID를 찾습니다. 메타페이지에는 다른 메타데이터도 포함되어 있습니다.



인덱스 페이지의 데이터 레이아웃은 지금까지 본 것과 약간 다릅니다. 각 레벨의 가장 오른쪽 페이지를 제외한 모든 페이지에는 이 페이지의 모든 키보다 작지 않은 것이 보장되는 추가적인 "high key"가 포함됩니다. 위 다이어그램에서 high key는 강조 표시되어 있습니다.

6자리 예약 참조 번호를 기반으로 구축된 실제 인덱스의 페이지를 pageinspect 확장을 사용하여 살펴보겠습니다. 메타페이지에 루트 페이지 ID와 트리의 깊이(레벨 번호는 리프 노드에서 시작되며 0부터 시작)가 나열되어 있습니다.

```
=> SELECT root, level
FROM bt_metap('bookings_pkey');
 root ¦ level
------+-------
  290 ¦ 2
(1 row)
```

인덱스 항목에 저장된 키는 바이트 시퀀스로 표시되므로 정말 편리하지 않습니다:

```
=> SELECT data
FROM bt_page_items('bookings_pkey',290)
WHERE itemoffset = 2;
```

```
       data
-------------------------
0f 30 43 39 41 42 31 00
(1 row)
```

이 값들을 해독하기 위해서는 임시 함수를 작성해야 합니다. 모든 플랫폼을 지원하지는 않고 특정 시나리오에서 작동하지 않을 수 있지만, 이 장의 예제에 대해서는 작동할 것입니다:

```
=> CREATE FUNCTION data_to_text(data text)
RETURNS text
AS $$
DECLARE
  raw bytea := ('\x'||replace(data,' ',''))::bytea;
  pos integer := 0;
  len integer;
  res text := '';
BEGIN
  WHILE (octet_length(raw) > pos)
  LOOP
      len := (get_byte(raw,pos) - 3) / 2;
      EXIT WHEN len <= 0;
      IF pos > 0 THEN
              res := res || ', ';
      END IF;
      res := res || (
              SELECT string_agg( chr(get_byte(raw, i)),'')
              FROM generate_series(pos+1,pos+len) i
              );
      pos := pos + len + 1;
  END LOOP;
  RETURN res;
END;
$$ LANGUAGE plpgsql;
```

이제 루트 페이지의 내용을 살펴볼 수 있습니다:

```
=> SELECT itemoffset, ctid, data_to_text(data)
FROM bt_page_items('bookings_pkey',290);
 itemoffset |     ctid | data_to_text
------------+----------+--------------
          1 |    (3,0) |
          2 |  (289,1) | 0C9AB1
          3 |  (575,1) | 192F03
          4 |  (860,1) | 25D715
          5 | (1145,1) | 32785C
```

```
        ...
         17 ¦ (4565,1) ¦ C993F6
      18 ¦ (4850,1) ¦ D63931
      19 ¦ (5135,1) ¦ E2CB14
      20 ¦ (5420,1) ¦ EF6FEA
      21 ¦ (5705,1) ¦ FC147D
(21 rows)
```

말씀하신 대로, 첫 번째 항목에는 키가 없습니다. ctid 열은 자식 페이지로의 링크를 제공합니다.

E2D725 예약을 찾는다고 가정해보겠습니다. 이 경우 19번 항목(E2CB14 <= E2D725 < EF6FEA)을 선택하고 5135 페이지로 내려가야 합니다.

```
=> SELECT itemoffset, ctid, data_to_text(data)
FROM bt_page_items('bookings_pkey',5135);
 itemoffset ¦     ctid ¦ data_to_text
------------+----------+--------------
          1 ¦ (5417,1) ¦ EF6FEA. <- high key
          2 ¦ (5132,0) ¦
          3 ¦ (5133,1) ¦ E2D71D
          4 ¦ (5134,1) ¦ E2E2F4
          5 ¦ (5136,1) ¦ E2EDE7
         ...
        282 ¦ (5413,1) ¦ EF41BE
        283 ¦ (5414,1) ¦ EF4D69
        284 ¦ (5415,1) ¦ EF58D4
        285 ¦ (5416,1) ¦ EF6410
(285 rows)
```

이 페이지의 첫 번째 항목에는 약간 예상外인 high key가 포함되어 있습니다. 논리적으로 이는 페이지의 끝에 배치되어야 하겠지만, 구현 관점에서 페이지 콘텐츠가 변경될 때마다 이동을 피하기 위해 시작 부분에 두는 것이 더 편리합니다.

여기서 우리는 항목 3(E2D71D <= E2D725 < E2E2F4)을 선택하고 11919 페이지로 내려갑니다.

```
=> SELECT itemoffset, ctid, data_to_text(data)
FROM bt_page_items('bookings_pkey',5133);
 itemoffset ¦        ctid ¦ data_to_text
------------+-------------+--------------
          1 ¦   (11921,1) ¦ E2E2F4
          2 ¦  (11919,76) ¦ E2D71D
          3 ¦  (11919,77) ¦ E2D725
          4 ¦  (11919,78) ¦ E2D72D
          5 ¦  (11919,79) ¦ E2D733
         ...
```

```
        363 ¦ (11921,123) ¦ E2E2C9
        364 ¦ (11921,124) ¦ E2E2DB
        365 ¦ (11921,125) ¦ E2E2DF
      366 ¦ (11921,126) ¦ E2E2E5
      367 ¦ (11921,127) ¦ E2E2ED
(367 rows)
```

이는 인덱스의 리프 페이지입니다. 첫 번째 항목은 high key이며, 다른 모든 항목은 힙 튜플을 가리킵니다.

그리고 여기 우리의 예약이 있습니다:

```
=> SELECT * FROM bookings
WHERE ctid = '(11919,77)';
 book_ref ¦              book_date ¦ total_amount
----------+------------------------+--------------
   E2D725 ¦ 2017-01-25 04:10:00+03 ¦ 28000.00
(1 row)
```

이는 예약 코드로 예약을 검색할 때 낮은 레벨에서 일어나는 일의 개략입니다:

```
=> EXPLAIN (costs off)
SELECT * FROM bookings
WHERE book_ref = 'E2D725';
              QUERY PLAN
-----------------------------------------
  Index Scan using bookings_pkey on bookings
      Index Cond: (book_ref = 'E2D725'::bpchar)
(2 rows)
```

중복 제거

비고유 인덱스는 서로 다른 힙 튜플을 가리키는 많은 중복 키를 포함할 수 있습니다. 비고유 키가 두 번 이상 나타나기 때문에 공간을 많이 차지하므로, 중복은 키와 해당 튜플 ID 목록[376]을 포함하는 단일 인덱스 항목으로 축소됩니다. 이 절차(중복 제거라고 함)는 인덱스 크기를 상당히 줄일 수 있습니다.

하지만 MVCC 때문에 고유 인덱스도 중복을 포함할 수 있습니다: 인덱스는 테이블 행의 모든 버전에 대한 참조를 유지합니다. HOT 업데이트 메커니즘은 구식이고 일반적으로 짧은 수명을 가진 행 버전을 참조하여 인덱스 팽창을 방지하는 데 도움이 될 수 있지만, 때로는 적용할 수 없는 경우도 있습니다. 이 경우 중복 제거는 쓸모없는 힙 튜플을 지우고 추가 페이지 분할을 막는데 필요한 시간을 벌어줄 수 있습니다.

즉각적인 이익을 주지 않을 때 중복 제거에 자원을 낭비하지 않으려면, 페이지에 하나의 튜플을 더 수용할 공간이 없으면서만 축소가 수행됩니다.[377] 그러면 페이지 가위질과 중복 제거[378]가 공간을 확보하고 바람직하

[376] postgresql.org/docs/14/btree-implementation#BTREE-DEDUPLICATION.html

[377] backend/access/nbtree/nbtinsert.c, _bt_delete_or_dedup_one_page function

지 않은 페이지 분할을 방지할 수 있습니다. 하지만 중복이 드문 경우 deduplicate_items 저장 매개변수를 끄면 중복 제거 기능을 비활성화할 수 있습니다.

일부 인덱스는 중복 제거를 지원하지 않습니다. 주된 제한 사항은 키의 동등성을 그 내부 표현의 단순 이진 비교로 확인해야 한다는 것입니다. 결코 모든 데이터 유형이 이런 식으로 비교될 수 있는 것은 아닙니다. 예를 들어, 부동 소수점 숫자(float와 double precision)는 0에 대해 두 가지 다른 표현을 가집니다. 임의 정밀도 숫자(numeric)는 다른 척도에서 동일한 숫자를 나타낼 수 있는 반면 jsonb 유형은 이러한 숫자를 사용할 수 있습니다. 비결정적 정렬[379]을 사용하는 경우에도 텍스트 유형에 대한 중복 제거가 불가능합니다. 이는 동일한 기호를 다른 바이트 시퀀스로 표현할 수 있도록 허용합니다(표준 정렬은 결정적임).

게다가 현재 복합 유형, 범위 및 배열에 대한 중복 제거는 지원되지 않으며, `INCLUDE` 절로 선언된 인덱스에 대해서도 지원되지 않습니다.

특정 인덱스가 중복 제거를 사용할 수 있는지 확인하려면 해당 메타페이지의 allequalimage 필드를 확인할 수 있습니다:

```
=> CREATE INDEX ON tickets(book_ref);
=> SELECT allequalimage FROM bt_metap('tickets_book_ref_idx');
allequalimage
---------------
              t
(1 row)
```

allequalimage가 t이면 중복 제거가 지원됨을 의미합니다. 그리고 실제로 한 개의 튜플 ID(htid)를 가진 인덱스 항목과 ID 목록(tids)을 가진 항목이 혼합된 리프 페이지를 볼 수 있습니다:

```
=> SELECT itemoffset, htid, left(tids::text,27) tids,
data_to_text(data) AS data
FROM bt_page_items('tickets_book_ref_idx',1)
WHERE itemoffset > 1;
 itemoffset ┆       htid ┆                        tids ┆ data
------------+------------+-----------------------------+--------
          2 ┆ (32965,40) ┆                             ┆ 000004
          3 ┆ (47429,51) ┆                             ┆ 00000F
          4 ┆  (3648,56) ┆   {"(3648,56)","(3648,57)"} ┆ 000010
          5 ┆  (6498,47) ┆                             ┆ 000012
       ...
        271 ┆ (21492,46) ┆                             ┆ 000890
        272 ┆ (26601,57) ┆ {"(26601,57)","(26601,58)"} ┆ 0008AC
        273 ┆ (25669,37) ┆                             ┆ 0008B6
(272 rows)
```

---

[378] backend/access/nbtree/nbtdedup.c, _bt_dedup_pass function

[379] postgresql.org/docs/14/collation.html

내부 인덱스 항목의 컴팩트한 저장

중복 제거를 통해 인덱스의 리프 페이지에 더 많은 항목을 수용할 수 있습니다. 하지만 리프 페이지가 인덱스의 대부분을 차지하더라도 추가 분할을 방지하기 위해 내부 페이지에서 수행되는 데이터 압축이 검색 효율성이 트리 깊이에 직접적으로 좌우되기 때문에 똑같이 중요합니다.

내부 인덱스 항목에는 인덱스 키가 포함되어 있지만, 그 값은 검색 중에 하행할 하위 트리를 결정하는 데만 사용됩니다. 다중 열 인덱스의 경우 첫 번째 키 속성(또는 처음 몇 개)을 가져오기만 해도 충분합니다. 다른 속성은 페이지 공간을 절약하기 위해 잘릴 수 있습니다.

이러한 접미사 잘림은 리프 페이지가 분할될 때 발생하며, 내부 페이지가 새 포인터를 수용해야 할 때 발생합니다.[380]

> 이론적으로는 한 걸음 더 나아가 하위 트리를 구분하기에 충분한 행의 첫 몇 개 기호와 같이 속성의 의미 있는 부분만 보관할 수도 있습니다. 하지만 아직 구현되지 않았습니다. 인덱스 항목은 속성 전체를 포함하거나 이 속성을 완전히 배제합니다.

예를 들어, 예약 참조와 승객 이름을 포함하는 열에 대해 구축된 인덱스의 루트 페이지의 몇 가지 항목이 있습니다:

```
=> CREATE INDEX tickets_bref_name_idx
ON tickets(book_ref, passenger_name);
=> SELECT itemoffset, ctid, data_to_text(data)
FROM bt_page_items('tickets_bref_name_idx',229)
WHERE itemoffset BETWEEN 8 AND 13;
 itemoffset ¦     ctid ¦ data_to_text
------------+----------+---------------------------
          8 ¦ (1607,1) ¦ 1A98A0
          9 ¦ (1833,2) ¦ 1E57D1, SVETLANA MAKSIMOVA
         10 ¦ (2054,1) ¦ 220797
         11 ¦ (2282,1) ¦ 25DB06
         12 ¦ (2509,2) ¦ 299FE4, YURIY AFANASEV
         13 ¦ (2736,1) ¦ 2D62C9
(6 rows)
```

일부 인덱스 항목에 두 번째 속성이 없다는 것을 볼 수 있습니다.

당연하게도, 리프 페이지는 모든 키 속성과 `INCLUDE` 열 값(있는 경우)을 유지해야 합니다. 그렇지 않으면 인덱스 전용 스캔을 수행할 수 없습니다. 유일한 예외는 high key입니다. 이는 부분적으로 유지될 수 있습니다.

[380] backend/access/nbtree/nbtinsert.c, _bt_split function

25.4 연산자 클래스

비교 시맨틱

해싱 값뿐만 아니라, 시스템은 사용자 정의 유형을 포함한 다양한 유형의 값을 정렬하는 방법도 알아야 합니다. 이는 정렬, 그룹화, 병합 조인 및 기타 작업에 필수적입니다. 해싱의 경우와 마찬가지로, 특정 데이터 유형에 대한 비교 연산자는 연산자 클래스에 의해 정의됩니다.[381]

연산자 클래스를 통해 이름(>, <, = 등)에서 추상화할 수 있으며, 동일한 유형의 값을 정렬하는 여러 가지 방법을 제공할 수도 있습니다.

여기 bool_ops 패밀리의 btree 방법의 모든 연산자 클래스에서 정의해야 하는 필수 비교 연산자가 있습니다:

```
=> SELECT amopopr::regoperator AS opfamily_operator,
amopstrategy
FROM pg_am am
JOIN pg_opfamily opf ON opfmethod = am.oid
JOIN pg_amop amop ON amopfamily = opf.oid
WHERE amname = 'btree'
AND opfname = 'bool_ops'
ORDER BY amopstrategy;
   opfamily_operator ¦ amopstrategy
---------------------+--------------
  <(boolean,boolean) ¦ 1
 <=(boolean,boolean) ¦ 2
  =(boolean,boolean) ¦ 3
 >=(boolean,boolean) ¦ 4
  >(boolean,boolean) ¦ 5
(5 rows)
```

이 5개의 비교 연산자 각각은 하나의 전략에 해당하며[382], 이는 그 의미를 정의합니다:

- less than: 미만
- less than or equal to: 이하
- equal to: 같음
- greater than or equal to: 이상
- greater than: 초과

B-tree 연산자 클래스에는 여러 개의 지원 함수도 포함됩니다.[383] 첫 번째 함수는 첫 번째 인수가 두 번째 인수보다 큰 경우 1을 반환해야 하고, 첫 번째 인수가 두 번째 인수보다 작은 경우 -1을 반환해야 하며, 인수가 같은 경우 0을 반환해야 합니다.

---

381 postgresql.org/docs/14/btree-behavior.html

382 postgresql.org/docs/14/xindex#XINDEX-STRATEGIES.html

383 postgresql.org/docs/14/btree-support-funcs.html

다른 지원 함수는 선택 사항이지만 액세스 방법의 성능을 향상합니다.

이 메커니즘을 더 잘 이해하기 위해, 비기본 정렬을 사용하는 새 데이터 유형을 정의할 수 있습니다. 문서에는 복소수[384]의 예가 나와 있지만 C로 작성되어 있습니다. 다행히도 B-tree 연산자 클래스는 해석형 언어를 사용하여 구현할 수도 있으므로, 가능한 한 간단한 예제를 만들어 이 장점을 활용하겠습니다(비효율적일지라도 의도적으로).

정보 단위에 대한 새 복합 유형을 정의해보겠습니다:

```
=> CREATE TYPE capacity_units AS ENUM (
'B', 'kB', 'MB', 'GB', 'TB', 'PB'
);
=> CREATE TYPE capacity AS (
amount integer,
unit capacity_units
);
```

이제 새 유형의 열을 가진 테이블을 만들고 임의의 값으로 채웁니다:

```
=> CREATE TABLE test AS
SELECT ( (random()*1023)::integer, u.unit )::capacity AS cap
FROM generate_series(1,100),
              unnest(enum_range(NULL::capacity_units)) AS u(unit);
```

기본적으로 복합 유형의 값은 사전식 순서로 정렬되는데, 이 경우 자연스러운 순서와 같지 않습니다:

```
=> SELECT * FROM test ORDER BY cap;
  cap
-----------
(1,B)
(3,GB)
(4,MB)
(9,kB)
...
(1017,kB)
(1017,GB)
(1018,PB)
(1020,MB)
(600 rows)
```

이제 우리의 연산자 클래스를 만들어 보겠습니다. 우선 용량을 바이트로 변환하는 함수를 정의하겠습니다:

```
=> CREATE FUNCTION capacity_to_bytes(a capacity) RETURNS numeric
AS $$
```

[384] postgresql.org/docs/14/xindex#XINDEX-EXAMPLE.html

```
  SELECT a.amount::numeric *
      1024::numeric ^ ( array_position(enum_range(a.unit), a.unit) - 1 );
$$ LANGUAGE sql STRICT IMMUTABLE;
=> SELECT capacity_to_bytes('(1,kB)'::capacity);
capacity_to_bytes
-----------------------
1024.0000000000000000
(1 row)
```

연산자 클래스를 위한 지원 함수를 만듭니다:

```
=> CREATE FUNCTION capacity_cmp(a capacity, b capacity)
RETURNS integer
AS $$
  SELECT sign(capacity_to_bytes(a) - capacity_to_bytes(b));
$$ LANGUAGE sql STRICT IMMUTABLE;
```

이제 이 지원 함수를 사용하여 비교 연산자를 쉽게 정의할 수 있습니다. 고의로 이상한 이름을 사용하여 임의로 지정할 수 있음을 보여주고 있습니다:

```
=> CREATE FUNCTION capacity_lt(a capacity, b capacity) RETURNS boolean
AS $$
BEGIN
  RETURN capacity_cmp(a,b) < 0;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;

=> CREATE OPERATOR #<# (
  LEFTARG = capacity,
  RIGHTARG = capacity,
  FUNCTION = capacity_lt
);
```

다른 네 개의 연산자도 비슷한 방식으로 정의됩니다.

```
=> CREATE FUNCTION capacity_le(a capacity, b capacity) RETURNS boolean
AS $$
BEGIN
  RETURN capacity_cmp(a,b) <= 0;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;

=> CREATE OPERATOR #<=# (
  LEFTARG = capacity,
  RIGHTARG = capacity,
  FUNCTION = capacity_le
```

```
);

=> CREATE FUNCTION capacity_eq(a capacity, b capacity) RETURNS boolean
AS $$
BEGIN
  RETURN capacity_cmp(a,b) = 0;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;

=> CREATE OPERATOR #=# (
  LEFTARG = capacity,
  RIGHTARG = capacity,
  FUNCTION = capacity_eq,
MERGES -- can be used in merge joins
);

=> CREATE FUNCTION capacity_ge(a capacity, b capacity) RETURNS boolean
AS $$
BEGIN
  RETURN capacity_cmp(a,b) >= 0;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;

=> CREATE OPERATOR #>=# (
  LEFTARG = capacity,
  RIGHTARG = capacity,
  FUNCTION = capacity_ge
);

=> CREATE FUNCTION capacity_gt(a capacity, b capacity) RETURNS boolean
AS $$
BEGIN
  RETURN capacity_cmp(a,b) > 0;
END;
$$ LANGUAGE plpgsql IMMUTABLE STRICT;

=> CREATE OPERATOR #>#(
  LEFTARG = capacity,
  RIGHTARG = capacity,
  FUNCTION = capacity_gt
);
```

이 시점에서 우리는 이미 용량을 비교할 수 있습니다:

```
=> SELECT (1,'MB')::capacity #>#(512, 'kB')::capacity;
```

```
?column?
----------
t
(1 row)
```

연산자 클래스가 생성되면 정렬도 예상대로 작동하기 시작합니다:

```
=> CREATE OPERATOR CLASS capacity_ops
DEFAULT FOR TYPE capacity -- to be used by default
USING btree AS
OPERATOR 1 #<#,
OPERATOR 2 #<=#,
OPERATOR 3 #=#,
OPERATOR 4 #>=#,
OPERATOR 5 #>#,
FUNCTION 1 capacity_cmp(capacity,capacity);

=> SELECT * FROM test ORDER BY cap;
  cap
-----------
(1,B)
(21,B)
(27,B)
(35,B)
(46,B)
...
(1002,PB)
(1013,PB)
(1014,PB)
(1014,PB)
(1018,PB)
(600 rows)
```

새 인덱스가 생성될 때 기본적으로 우리의 연산자 클래스가 사용되며, 이 인덱스는 결과를 올바른 순서로 반환합니다:

```
=> CREATE INDEX ON test(cap);

=> SELECT * FROM test WHERE cap #<# (100,'B')::capacity ORDER BY cap;
cap
--------
(1,B)
(21,B)
(27,B)
(35,B)
```

```
(46,B)
(57,B)
(68,B)
(70,B)
(72,B)
(76,B)
(78,B)
(94,B)
(12 rows)

=> EXPLAIN (costs off) SELECT *
FROM test WHERE cap #<# (100,'B')::capacity ORDER BY cap;
              QUERY PLAN
-----------------------------------------
  Index Only Scan using test_cap_idx on test
      Index Cond: (cap #<# '(100,B)'::capacity)
(2 rows)
```

동등 연산자 선언에서 지정한 MERGES 절은 이 데이터 유형에 대한 병합 조인을 활성화합니다.

다중 열 인덱스 및 정렬

여러 열 인덱스 정렬을 더 자세히 살펴보겠습니다.

먼저, 인덱스를 선언할 때 열의 최적 순서를 선택하는 것이 매우 중요합니다. 페이지 내에서 데이터 정렬은 첫 번째 열에서 시작한 다음 두 번째 열로 이동하고 그렇게 계속합니다. 여러 열 인덱스는 제공된 필터 조건이 가장 첫 번째 열부터 시작하는 연속된 열 순서를 포함하는 경우에만 효율적인 검색을 보장할 수 있습니다. 첫 번째 열, 첫 번째 두 열, 첫 번째와 세 번째 열 사이의 범위 등과 같이 말입니다. 이외의 조건 유형은 다른 기준에 따라 가져온 중복 값을 필터링 하는 데만 사용할 수 있습니다.

여기 tickets 테이블에 생성된 인덱스의 첫 번째 리프 페이지에 있는 인덱스 항목의 순서가 있습니다. 이 인덱스에는 예약 참조와 승객 이름이 포함되어 있습니다.

```
=> SELECT itemoffset, data_to_text(data)
FROM bt_page_items('tickets_bref_name_idx',1)
WHERE itemoffset > 1;
 itemoffset ¦ data_to_text
------------+----------------------------
          2 ¦ 000004, PETR MAKAROV
          3 ¦ 00000F, ANNA ANTONOVA
          4 ¦ 000010, ALEKSANDR SOKOLOV
          5 ¦ 000010, LYUDMILA BOGDANOVA
          6 ¦ 000012, TAMARA ZAYCEVA
          7 ¦ 000026, IRINA PETROVA
          8 ¦ 00002D, ALEKSANDR SMIRNOV
```

```
         ...
        187 ¦ 0003EF, VLADIMIR CHERNOV
        188 ¦ 00040C, ANTONINA KOROLEVA
        189 ¦ 00040C, DMITRIY FEDOROV
        190 ¦ 00041E, EGOR FEDOROV
        191 ¦ 00041E, ILYA STEPANOV
        192 ¦ 000447, VIKTOR VASILEV
        193 ¦ 00044D, NADEZHDA KULIKOVA
(192 rows)
```

이 경우 효율적인 티켓 검색은 예약 참조와 승객 이름으로 하거나 예약 참조만으로 가능합니다.

```
=> EXPLAIN (costs off) SELECT *
FROM tickets
WHERE book_ref = '000010';
       QUERY PLAN
------------------------------------------------
  Index Scan using tickets_book_ref_idx on tickets
      Index Cond: (book_ref = '000010'::bpchar)
(2 rows)

=> EXPLAIN (costs off) SELECT *
FROM tickets
WHERE book_ref = '000010' AND passenger_name = 'LYUDMILA BOGDANOVA';
                      QUERY PLAN
------------------------------------------------------------------------
  Index Scan using tickets_bref_name_idx on tickets
      Index Cond: ((book_ref = '000010'::bpchar) AND (passenger_name...
(2 rows)
```

하지만 승객 이름을 찾기로 결정한다면, 모든 행을 스캔해야 합니다:

```
=> EXPLAIN (costs off) SELECT *
FROM tickets
WHERE passenger_name = 'LYUDMILA BOGDANOVA';
                      QUERY PLAN
------------------------------------------------------------
  Gather
      Workers Planned: 2
      -> Parallel Seq Scan on tickets
              Filter: (passenger_name = 'LYUDMILA BOGDANOVA'::text)
(4 rows)
```

플래너가 인덱스 스캔을 선택하더라도, 모든 인덱스 항목을 반복해야 합니다.[385] 불행히도 계획은 조건이 실제로 결과를 필터링 하는 데만 사용된다는 것을 보여주지 않습니다.

> 첫 번째 열에 너무 많지 않은 고유 값 $v_1, v_2, \ldots, v_n$이 있는 경우, 해당 하위 트리에 대해 여러 패스를 수행하면 실제로 조건 "col2 = 값"에 대한 단일 검색을 일련의 검색으로 가상으로 대체할 수 있습니다. 다음과 같은 조건에 의한 검색 말입니다:
>
> col1 = v_1 AND col2 = 값
> col1 = v_2 AND col2 = 값
> ...
> col1 = v_n AND col2 = 값
>
> 이러한 유형의 인덱스 액세스를 스킵 스캔이라고 하는데, 아직 구현되지 않았습니다.[386]

그리고 반대로, 승객 이름과 예약 번호에 대한 인덱스가 생성되면, 승객 이름만 또는 승객 이름과 예약 참조 둘 다를 사용하는 쿼리에 더 잘 맞습니다:

```
=> CREATE INDEX tickets_name_bref_idx
ON tickets(passenger_name, book_ref);
=> SELECT itemoffset, data_to_text(data)
FROM bt_page_items('tickets_name_bref_idx',1)
WHERE itemoffset > 1;
 itemoffset ¦ data_to_text
------------+----------------------------
          2 ¦ ADELINA ABRAMOVA, E37EDB
          3 ¦ ADELINA AFANASEVA, 1133B7
          4 ¦ ADELINA AFANASEVA, 4F3370
          5 ¦ ADELINA AKIMOVA, 7D2881
          6 ¦ ADELINA ALEKSANDROVA, 3C3ADD
          7 ¦ ADELINA ALEKSANDROVA, 52801E
            ...
        185 ¦ ADELINA LEBEDEVA, 0A00E3
        186 ¦ ADELINA LEBEDEVA, DAEADE
        187 ¦ ADELINA LEBEDEVA, DFD7E5
        188 ¦ ADELINA LOGINOVA, 8022F3
        189 ¦ ADELINA LOGINOVA, EE67B9
        190 ¦ ADELINA LUKYANOVA, 292786
        191 ¦ ADELINA LUKYANOVA, 54D3F9
(190 rows)

=> EXPLAIN (costs off) SELECT * FROM tickets
```

[385] backend/access/nbtree/nbtsearch.c, _bt_first function

[386] commitfest.postgresql.org/34/1741

```
WHERE passenger_name = 'LYUDMILA BOGDANOVA';
                        QUERY PLAN
-----------------------------------------------------------------
  Bitmap Heap Scan on tickets
      Recheck Cond: (passenger_name = 'LYUDMILA BOGDANOVA'::text)
      -> Bitmap Index Scan on tickets_name_bref_idx
              Index Cond: (passenger_name = 'LYUDMILA BOGDANOVA'::text)
(4 rows)
```

열 순서 외에도 새 인덱스를 생성할 때 정렬 순서에도 주의를 기울여야 합니다. 기본적으로 값은 오름차순(ASC)으로 정렬되지만 필요한 경우 내림차순(DESC)으로 반전할 수 있습니다. 인덱스가 단일 열 위에 구축된 경우 어떤 방향으로도 스캔할 수 있기 때문에 크게 중요하지 않습니다. 하지만 여러 열 인덱스의 경우 순서가 중요해집니다.

새로 생성된 인덱스를 사용하여 두 열 모두를 기준으로 데이터를 오름차순이나 내림차순으로 정렬된 상태로 검색할 수 있습니다:

```
=> EXPLAIN (costs off) SELECT *
FROM tickets
ORDER BY passenger_name, book_ref;
               QUERY PLAN
------------------------------------------------
  Index Scan using tickets_name_bref_idx on tickets
(1 row)

=> EXPLAIN (costs off) SELECT *
FROM tickets ORDER BY passenger_name DESC, book_ref DESC;
               QUERY PLAN
-------------------------------------------------------------
  Index Scan Backward using tickets_name_bref_idx on tickets
(1 row)
```

하지만 이 인덱스는 한 열을 기준으로 오름차순 정렬하고 다른 열을 기준으로 내림차순 정렬해야 하는 경우 즉시 데이터를 반환할 수 없습니다.

이 경우 인덱스는 두 번째 속성을 기준으로 추가 정렬이 필요한 부분적으로 정렬된 데이터를 제공합니다:

```
=> EXPLAIN (costs off) SELECT *
FROM tickets ORDER BY passenger_name ASC, book_ref DESC;
                        QUERY PLAN
-----------------------------------------------------------
  Incremental Sort
      Sort Key: passenger_name, book_ref DESC
      Presorted Key: passenger_name
      -> Index Scan using tickets_name_bref_idx on tickets
```

```
(4 rows)
```

NULL 값의 위치는 정렬을 위해 인덱스를 사용할 수 있는 능력에도 영향을 미칩니다. 기본적으로, 정렬 목적으로 NULL 값은 일반 값보다 "더 큰" 것으로 간주되어, 정렬 순서가 오름차순인 경우 트리의 오른쪽에, 내림차순인 경우 왼쪽에 위치합니다. NULL 값의 위치는 NULLS LAST와 NULLS FIRST 절을 사용하여 변경할 수 있습니다.

다음 예에서, 인덱스가 ORDER BY 절을 만족시키지 못하므로, 결과는 정렬되어야 합니다:

```
=> EXPLAIN (costs off) SELECT *
FROM tickets ORDER BY passenger_name NULLS FIRST, book_ref DESC;
                      QUERY PLAN
------------------------------------------------------------
  Gather Merge
      Workers Planned: 2
      -> Sort
              Sort Key: passenger_name NULLS FIRST, book_ref DESC
              -> Parallel Seq Scan on tickets
(5 rows)
```

하지만 원하는 순서를 따르는 인덱스를 생성하면, 그 인덱스가 사용될 것입니다:

```
=> CREATE INDEX tickets_name_bref_idx2
ON tickets(passenger_name NULLS FIRST, book_ref DESC);
=> EXPLAIN (costs off) SELECT *
FROM tickets ORDER BY passenger_name NULLS FIRST, book_ref DESC;
              QUERY PLAN
-----------------------------------------------------
  Index Scan using tickets_name_bref_idx2 on tickets
(1 row)
```

25.5 속성

B-트리의 인터페이스 속성을 살펴봅시다.

액세스 방법 속성

```
=> SELECT a.amname, p.name, pg_indexam_has_property(a.oid, p.name)
FROM pg_am a, unnest(array[
'can_order', 'can_unique', 'can_multi_col',
'can_exclude', 'can_include'
]) p(name)
WHERE a.amname = 'btree';
 amname |          name | pg_indexam_has_property
--------+---------------+-------------------------
  btree |     can_order | t
```

```
  btree ¦    can_unique ¦ t
  btree ¦ can_multi_col ¦ t
  btree ¦   can_exclude ¦ t
  btree ¦   can_include ¦ t
(5 rows)
```

B-트리는 데이터를 정렬하고 그 고유성을 보장할 수 있습니다. 이러한 속성을 가진 유일한 접근 방법입니다.

많은 접근 방법들이 다중 컬럼 인덱스를 지원하지만, B-트리에서 값들이 정렬되어 있기 때문에 인덱스 내의 컬럼 순서에 주의를 기울여야 합니다.

공식적으로, 배타적 제약 조건이 지원되지만, 이는 동등 조건에 한정되어 있어, 고유 제약 조건과 유사합니다. 완전한 고유 제약 조건을 사용하는 것이 훨씬 더 바람직합니다. B-트리 인덱스는 검색에 참여하지 않는 추가 `INCLUDE` 컬럼으로 확장될 수도 있습니다.

인덱스 수준의 속성

```
=> SELECT p.name, pg_index_has_property('flights_pkey', p.name)
FROM unnest(array[
'clusterable', 'index_scan', 'bitmap_scan', 'backward_scan'
]) p(name);
          name ¦ pg_index_has_property
---------------+------------------------
   clusterable ¦ t
    index_scan ¦ t
   bitmap_scan ¦ t
 backward_scan ¦ t
(4 rows)
```

B-트리 인덱스는 클러스터화에 사용될 수 있습니다. 인덱스 스캔과 비트맵 스캔 모두 지원됩니다. 리프 페이지가 양방향 리스트로 묶여 있기 때문에, 인덱스는 역방향으로도 탐색될 수 있으며, 이는 역정렬 순서를 결과로 합니다:

```
=> EXPLAIN (costs off) SELECT *
FROM bookings ORDER BY book_ref DESC;
              QUERY PLAN
-----------------------------------------------------
  Index Scan Backward using bookings_pkey on bookings
(1 row)
```

칼럼 수준의 속성

```
=> SELECT p.name,
pg_index_column_has_property('flights_pkey', 1, p.name)
```

```
FROM unnest(array[
'asc', 'desc', 'nulls_first', 'nulls_last', 'orderable',
'distance_orderable', 'returnable', 'search_array', 'search_nulls'
]) p(name);
               name ¦ pg_index_column_has_property
--------------------+-----------------------------
                asc ¦ t
               desc ¦ f
        nulls_first ¦ f
         nulls_last ¦ t
          orderable ¦ t
 distance_orderable ¦ f
         returnable ¦ t
       search_array ¦ t
       search_nulls ¦ t
(9 rows)
```

`ORDERABLE` 속성은 B-트리에 저장된 데이터가 정렬되어 있다는 것을 나타냅니다. 반면 처음 네 가지 속성(`ASC`와 `DESC`, `NULLS FIRST` 및 `NULLS LAST`)은 특정 컬럼에서의 실제 순서를 정의합니다. 이 예시에서는 컬럼 값들이 오름차순으로 정렬되며 `NULL` 값은 마지막에 나열됩니다.

`SEARCH NULLS` 속성은 `NULL` 값이 검색 가능한지 여부를 나타냅니다.

B-트리는 순서 연산자(`DISTANCE ORDERABLE`)를 지원하지 않습니다. 비록 이를 구현하려는 시도가 있었음에도 불구하고요.[387]

B-트리는 배열 내의 여러 요소를 검색하는 것(`SEARCH ARRAY` 속성)을 지원하며, 힙 접근 없이 결과 데이터를 반환할 수 있습니다(`RETURNABLE`).

[387] commitfest.postgresql.org/27/1804

26 장. GiST

26.1 개요

`GiST(Generalized Search Tree)`[388]는 값의 상대적 위치 지정을 지원하는 데이터 유형을 위한 균형 검색 트리의 일반화라고 할 수 있는 접근 방법입니다. B-트리의 적용 가능성은 비교 연산을 허용하는 순서형 유형에 한정되지만, 이러한 유형에 대한 지원은 매우 효율적입니다. `GiST`의 경우, 그 연산자 클래스는 트리 내의 데이터 분포에 대해 임의의 기준을 정의할 수 있게 합니다. `GiST` 인덱스는 공간 데이터를 위한 R-트리, 집합을 위한 RD-트리, 그리고 텍스트 및 이미지를 포함한 모든 데이터 유형을 위한 시그니처 트리를 수용할 수 있습니다.

확장성 덕분에, PostgreSQL에서 색인 엔진의 인터페이스를 구현함으로써 처음부터 새로운 접근 방법을 생성할 수 있습니다. 그러나, 색인 로직을 설계하는 것 외에도, 페이지 레이아웃, 효율적인 잠금 전략, 그리고 `WAL` 지원을 정의해야 합니다. 이 모든 것은 강력한 프로그래밍 기술과 상당한 구현 노력을 필요로 합니다. `GiST`는 이러한 작업을 단순화하며, 모든 저수준 기술 사항을 처리하고 검색 알고리즘의 기반을 제공합니다. 새로운 데이터 유형과 함께 `GiST` 방법을 사용하려면, 대략 열 개의 지원 함수를 포함하는 새로운 연산자 클래스를 추가하기만 하면 됩니다. B-트리에 제공되는 단순한 연산자 클래스와 달리, 이러한 클래스에는 대부분의 색인 로직이 포함되어 있습니다. 이 측면에서 `GiST`는 새로운 접근 방법을 구축하기 위한 프레임워크로 간주될 수 있습니다.

가장 일반적인 용어로 말하자면, 리프 노드(리프 항목)에 속하는 각 항목은 조건(논리적 조건)과 힙 튜플 `ID`를 포함합니다. 인덱스 키는 조건을 만족해야 하며, 키 자체가 이 항목의 일부인지 여부는 중요하지 않습니다.

내부 리프(내부 항목)의 각 항목에는 또한 조건과 자식 노드에 대한 참조가 포함됩니다; 자식 서브트리의 모든 색인된 데이터는 이 조건을 만족해야 합니다. 다시 말해, 내부 항목의 조건은 그 자식 항목들의 모든 조건의 합집합입니다. 이 중요한 `GiST`의 속성은 B-트리가 사용하는 간단한 순위 결정을 위한 목적을 제공합니다.

`GiST` 트리 검색은 일관성 함수에 의존하는데, 이는 연산자 클래스에 의해 정의된 지원 함수 중 하나입니다.

일관성 함수는 인덱스 항목에 대해 호출되어 이 항목의 조건이 검색 조건("**색인된-컬럼 연산자 표현식**")과 "일관적인지"를 판단합니다. 내부 항목의 경우, 해당 서브트리로 내려가야 하는지를 보여줍니다; 리프 항목의 경우, 그것의 인덱스 키가 조건을 만족하는지를 검사합니다.

검색은 트리 검색의 전형적인 방식인 루트 노드에서 시작됩니다.[389] 일관성 함수는 어떤 자식 노드들이 순회되어야 하고 어떤 것들이 건너뛰어질 수 있는지를 결정합니다. 그런 다음 이 절차는 발견된 각 자식 노드에 대해 반복됩니다; B-트리와 달리, `GiST` 인덱스는 여러 개의 이러한 노드를 가질 수 있습니다. 일관성 함수에 의해 선택된 리프 노드 항목들이 결과로 반환됩니다.

검색은 항상 깊이 우선입니다: 알고리즘은 가능한 한 빨리 리프 페이지에 도달하려고 합니다. 따라서, 바로 결과를 반환하기 시작할 수 있으며, 이는 사용자가 처음 몇 줄만 필요로 할 때 많은 의미를 가집니다.

---

[388] postgresql.org/docs/14/gist.html
backend/access/gist/README

[389] backend/access/gist/gistget.c, gistgettuple function

GiST 트리에 새로운 값을 삽입하기 위해서는 일관성 함수를 사용할 수 없습니다.[390] 우리는 정확히 하나의 노드로 내려가야 할 필요가 있기 때문입니다. 이 노드는 최소 삽입 비용을 가져야 하며, 이는 연산자 클래스의 **패널티 함수**에 의해 결정됩니다.

B-트리의 경우와 마찬가지로, 선택된 노드에 여유 공간이 없을 수 있으며, 이는 분할로 이어집니다.[391] 이 작업은 두 개의 추가 함수를 필요로 합니다. 하나는 항목들을 기존 노드와 새 노드 사이에 분배하는 역할을 하고, 다른 하나는 두 조건의 합집합을 형성하여 부모 노드의 조건을 업데이트합니다.

새로운 값이 추가됨에 따라 기존의 조건들이 확장되며, 페이지가 분할되거나 전체 인덱스가 재구성될 때만 일반적으로 좁혀집니다. 따라서, GiST 인덱스의 빈번한 업데이트는 성능 저하로 이어질 수 있습니다.

이러한 이론적 논의가 너무 모호하게 느껴질 수 있으며, 정확한 로직은 어차피 특정 연산자 클래스에 대부분 의존하기 때문에, 몇 가지 구체적인 예를 제공하겠습니다.

26.2 포인트용 R-트리

첫 번째 예는 평면 위의 점들(또는 다른 기하학적 형태들)을 색인화하는 것과 관련이 있습니다. 정규 B-트리는 점들에 대해 정의된 비교 연산자가 없기 때문에 이 데이터 유형에 사용될 수 없습니다. 물론, 우리 스스로 그러한 연산자들을 구현할 수 있었겠지만, 기하학적 형태들은 완전히 다른 작업들을 위한 인덱스 지원이 필요합니다. 저는 그 중 두 가지만 다루겠습니다: 특정 영역 내에 포함된 객체를 검색하고, 가장 가까운 이웃 검색입니다.

R-트리는 평면 위에 사각형을 그립니다; 이들은 함께 모든 색인된 점들을 커버해야 합니다. 인덱스 항목은 경계 상자를 저장하며, 조건은 다음과 같이 정의될 수 있습니다: 점은 이 경계 상자 내에 있습니다.

R-트리의 루트는 여러 개의 큰 사각형들(겹칠 수도 있음)을 포함합니다. 자식 노드들은 부모 노드에 맞는 더 작은 사각형들을 보유하며; 함께, 그들은 모든 기저 점들을 커버합니다.

리프 노드는 색인된 점들 자체를 포함해야 하지만, GiST는 모든 항목이 동일한 데이터 유형을 가져야 한다고 요구합니다; 따라서, 리프 항목도 사각형으로 표현되며, 단순히 점으로 축소됩니다.

이 구조를 더 잘 시각화하기 위해, 공항 좌표 위에 구축된 R-트리의 세 레벨을 살펴보겠습니다. 이 예를 위해, 데모 데이터베이스의 공항 테이블을 오천 행까지 확장했습니다.[392] 또한, 트리를 더 깊게 만들기 위해 fillfactor(기본값: 90) 값을 줄였습니다; 기본값은 우리에게 단일 레벨 트리를 제공했을 것입니다.

```
=> CREATE TABLE airports_big AS
SELECT * FROM airports_data;

=> COPY airports_big FROM
'/home/student/internals/airports/extra_airports.copy';
```

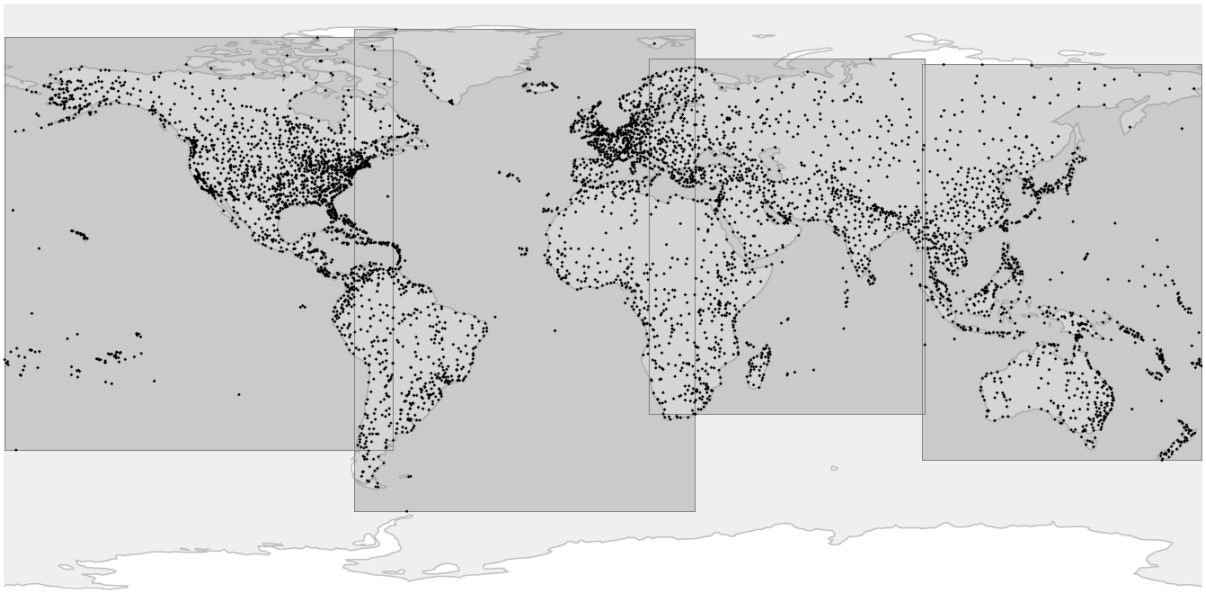
[390] backend/access/gist/gistutil.c, gistchoose function

[391] backend/access/gist/gistsplit.c, gistSplitByKey function

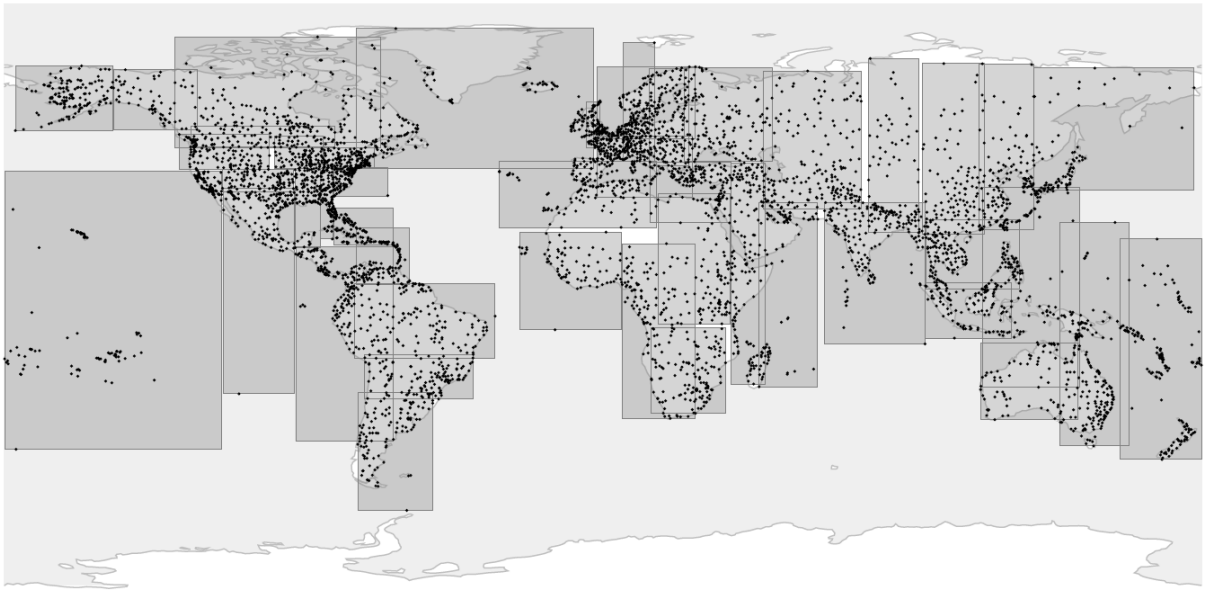
[392] You can download the corresponding file at edu.postgrespro.ru/internals-14/extra_airports.copy (I have used the data available at the openflights.org website).

```
=> CREATE INDEX airports_gist_idx ON airports_big
USING gist(coordinates) WITH (fillfactor=10);
```

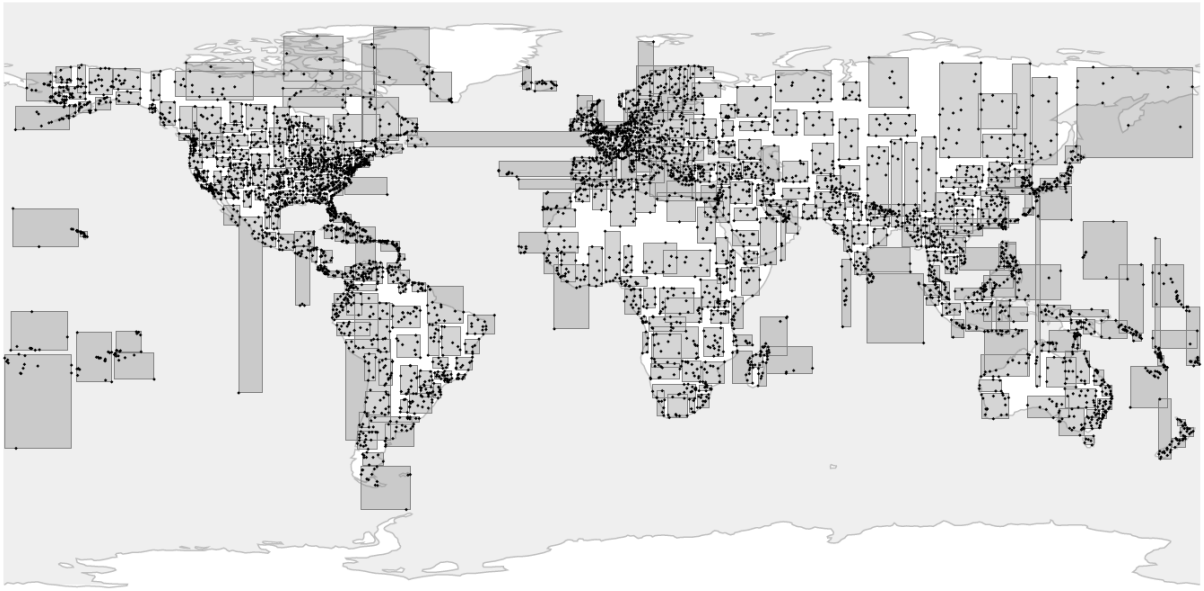
상위 레벨에서, 모든 점들은 여러 개의 (부분적으로 겹치는) 경계 상자에 포함됩니다:



다음 레벨에서, 큰 사각형들은 더 작은 사각형들로 분할됩니다:



마지막으로, 트리의 내부 레벨에서 각 경계 상자는 단일 페이지가 수용할 수 있는 만큼의 점들을 포함합니다:



이 인덱스는 점에 사용 가능한 유일한 연산자 클래스인 `point_ops` 연산자 클래스를 사용합니다.

사각형과 다른 모든 기하학적 형태들 역시 같은 방식으로 색인될 수 있지만, 객체 자체 대신에 그것의 경계 상자를 인덱스가 저장해야 합니다.

페이지 구성

`GiST` 페이지를 연구하는 데에는 `pageinspect` 확장 기능을 사용할 수 있습니다.

B-트리 인덱스와 달리, `GiST`에는 메타페이지가 없으며, 제로 페이지는 항상 트리의 루트입니다. 루트 페이지가 분할되면, 기존 루트는 별도의 페이지로 이동되고, 새로운 루트가 그 자리를 차지합니다.

다음은 루트 페이지의 내용입니다:

```
=> SELECT ctid, keys
FROM gist_page_items(
get_raw_page('airports_gist_idx', 0), 'airports_gist_idx'
);
        ctid ¦ keys
-------------+-------------------------------------------------------
 (207,65535) ¦ (coordinates)=((50.84510040283203,78.246101379395))
 (400,65535) ¦ (coordinates)=((179.951004028,73.51780700683594))
 (206,65535) ¦ (coordinates)=((-1.5908199548721313,40.63980103))
 (466,65535) ¦ (coordinates)=((-1.0334999561309814,82.51779937740001))
(4 rows)
```

이 네 개의 행은 첫 번째 그림에서 보여진 상위 레벨의 네 개의 사각형에 해당합니다. 불행히도, 여기서 키들은 사각형(내부 페이지에 더 합리적일 것)으로가 아닌 점들(리프 페이지에는 의미가 있음)로 표시됩니다. 하지만, 우리는 항상 원시 데이터를 얻고 스스로 해석할 수 있습니다.

더 자세한 정보를 추출하려면, 표준 PostgreSQL 배포판에 포함되지 않은 `gevel` 확장 기능[393]을 사용할 수 있습니다.

연산자 클래스

다음 쿼리는 트리의 검색 및 삽입 작업의 로직을 구현하는 지원 함수 목록을 반환합니다: [394]

```
=> SELECT amprocnum, amproc::regproc
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
JOIN pg_amproc amop ON amprocfamily = opcfamily
WHERE amname = 'gist'
AND opcname = 'point_ops'
ORDER BY amprocnum;
 amprocnum ¦ amproc
-----------+------------------------
         1 ¦ gist_point_consistent
         2 ¦ gist_box_union
         3 ¦ gist_point_compress
         5 ¦ gist_box_penalty
         6 ¦ gist_box_picksplit
         7 ¦ gist_box_same
         8 ¦ gist_point_distance
         9 ¦ gist_point_fetch
        11 ¦ gist_point_sortsupport
(9 rows)
```

이미 위에서 필수 함수들을 나열했습니다:

1 검색 중 트리를 순회하는 데 사용되는 일관성 함수(**consistency** function)
2 사각형을 병합하는 유니온 함수(**union** function)
5 항목을 삽입할 때 하위 트리를 선택하기 위해 사용되는 패널티 함수(**penalty** function)
6 페이지 분할 후 새 페이지 사이에 항목을 분배하는 픽스플릿 함수(**picksplit** function)
7 두 키가 동일한지 확인하는 같음 함수(**same** function)

`point_ops` 연산자 클래스에는 다음 연산자들이 포함됩니다:

```
=> SELECT amopopr::regoperator, amopstrategy AS st, oprcode::regproc,
left(obj_description(opr.oid, 'pg_operator'), 19) description
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
JOIN pg_amop amop ON amopfamily = opcfamily
JOIN pg_operator opr ON opr.oid = amopopr
```

[393] sigaev.ru/git/gitweb.cgi?p=gevel.git

[394] postgresql.org/docs/14/gist-extensibility.html

```
WHERE amname = 'gist'
AND opcname = 'point_ops'
ORDER BY amopstrategy;
           amopopr ¦ st ¦             oprcode ¦ description
-------------------+----+---------------------+---------------------
   <<(point,point) ¦ 1  ¦          point_left ¦ is left of
   >>(point,point) ¦ 5  ¦         point_right ¦ is right of
   ~=(point,point) ¦ 6  ¦            point_eq ¦ same as
  <<¦(point,point) ¦ 10 ¦         point_below ¦ is below
  ¦>>(point,point) ¦ 11 ¦         point_above ¦ is above
  <->(point,point) ¦ 15 ¦      point_distance ¦ distance between
     <@(point,box) ¦ 28 ¦               on_pb ¦ point inside box
   <^(point,point) ¦ 29 ¦         point_below ¦ deprecated, use <<¦
   >^(point,point) ¦ 30 ¦         point_above ¦ deprecated, use ¦>>
 <@(point,polygon) ¦ 48 ¦   pt_contained_poly ¦ is contained by
  <@(point,circle) ¦ 68 ¦ pt_contained_circle ¦ is contained by
(11 rows)
```

연산자 이름들은 보통 연산자의 의미에 대해 많은 정보를 제공하지 않기 때문에, 이 쿼리는 또한 기본 함수들의 이름과 그 설명을 표시합니다. 어떤 방식으로든, 모든 연산자들은 기하학적 형태들의 상대적 위치(**왼쪽, 오른쪽, 위, 아래, 포함함, 포함됨**)와 그들 사이의 거리를 다룹니다.

B-트리와 비교할 때, `GiST`는 더 많은 전략을 제공합니다. 일부 전략 번호들은 여러 유형의 인덱스에 공통적이며[395], 다른 일부는 공식에 의해 계산됩니다(예를 들어, 28, 48, 68은 사실상 같은 전략을 가상으로 나타냅니다: 사각형, 다각형, 원에 대한 포함됨). 또한, `GiST`는 일부 구식 연산자 이름(<<| 및 |>>)을 지원합니다.

연산자 클래스는 사용 가능한 전략 중 일부만을 구현할 수 있습니다. 예를 들어, 포함 전략은 점에 대한 연산자 클래스에서는 지원되지 않지만, 측정 가능한 면적을 가진 기하학적 형태에 대해 정의된 클래스(`box_ops`, `poly_ops`, `circle_ops`)에서는 사용 가능합니다.

포함된 요소 검색

인덱스로 속도를 높일 수 있는 전형적인 쿼리는 지정된 영역의 모든 점을 반환합니다.

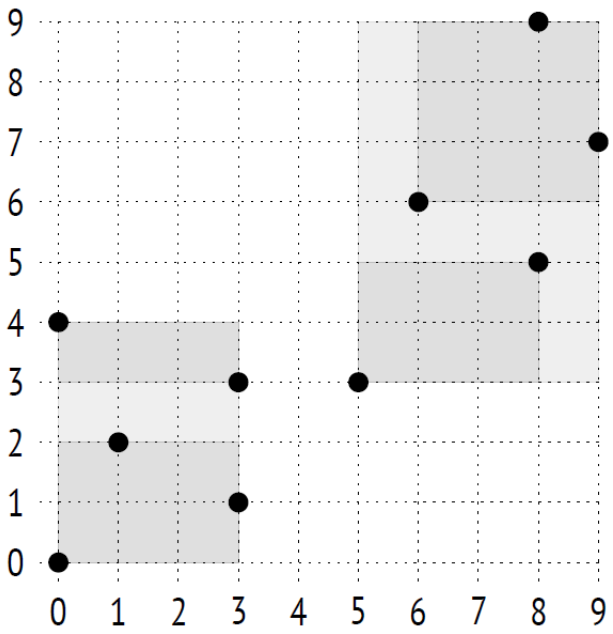
예를 들어, 모스크바 중심에서 한 도 이내에 위치한 모든 공항을 찾아봅시다:

```
=> SELECT airport_code, airport_name->>'en'
FROM airports_big
WHERE coordinates <@ '<(37.622513,55.753220),1.0>'::circle;
 airport_code ¦ ?column?
--------------+-----------------------------------
          SVO ¦ Sheremetyevo International Airport
          VKO ¦ Vnukovo International Airport
```

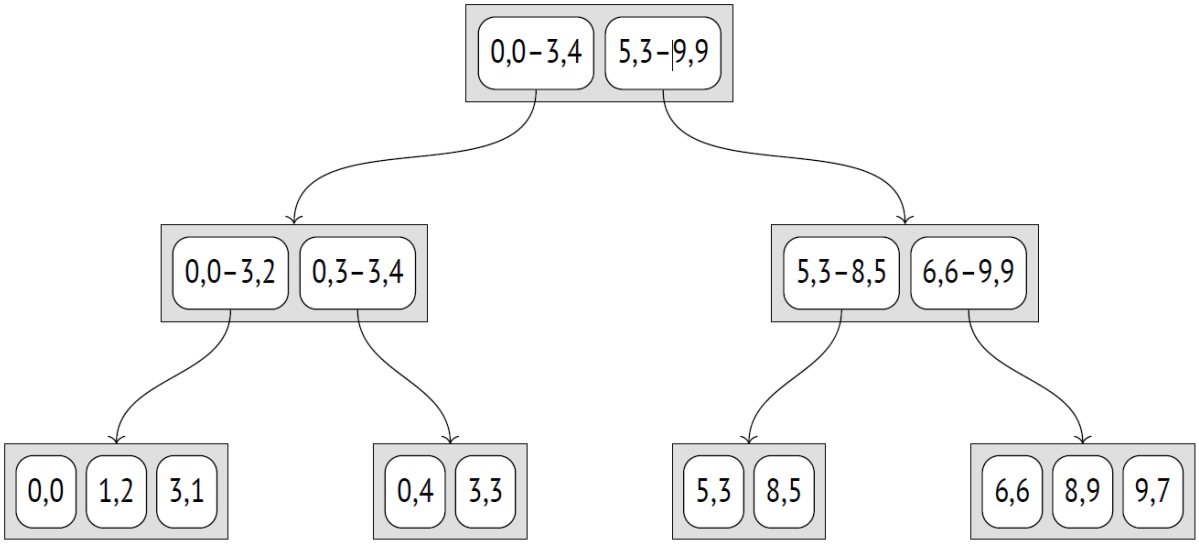
[395] include/access/stratnum.h

```
         DME ¦ Domodedovo International Airport
         BKA ¦ Bykovo Airport
         ZIA ¦ Zhukovsky International Airport
         CKL ¦ Chkalovskiy Air Base
         OSF ¦ Ostafyevo International Airport
(7 rows)

=> EXPLAIN (costs off) SELECT airport_code
FROM airports_big
WHERE coordinates <@ '<(37.622513,55.753220),1.0>'::circle;
                      QUERY PLAN
------------------------------------------------------------------
  Bitmap Heap Scan on airports_big
      Recheck Cond: (coordinates <@ '<(37.622513,55.75322),1>'::circle)
      -> Bitmap Index Scan on airports_gist_idx
              Index Cond: (coordinates <@ '<(37.622513,55.75322),1>'::ci...
(4 rows)
```

아래 그림에 보여진 단순한 예를 사용하여 이 연산자를 더 자세히 살펴볼 수 있습니다:

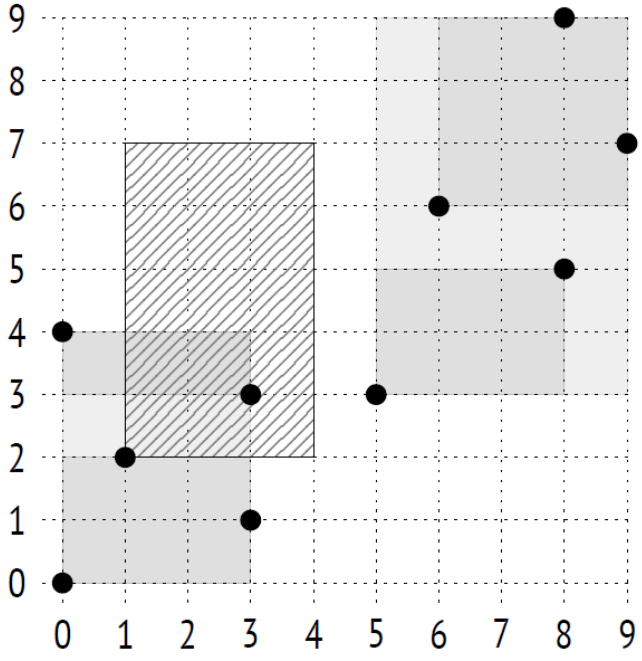


이러한 방식으로 경계 상자가 선택된다면, 인덱스 구조는 다음과 같을 것입니다:

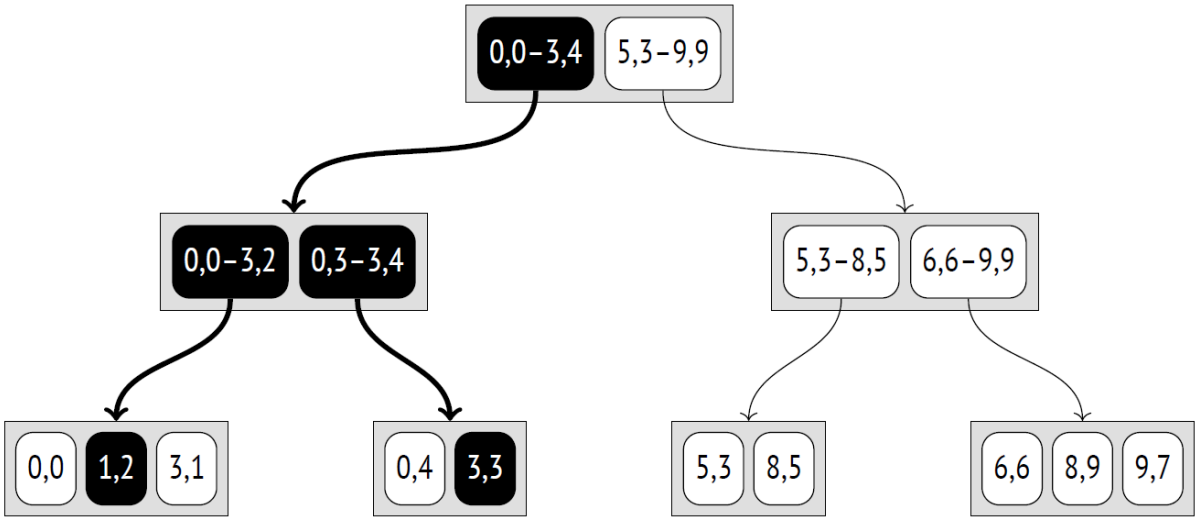


포함 연산자 <@는 특정 점이 지정된 사각형 내에 위치하는지를 결정합니다. 이 연산자에 대한 일관성 함수[396]는 인덱스 항목의 사각형이 이 사각형과 어떤 공통 점을 가지는지 여부에 따라 "예"를 반환합니다. 이는 리프 노드 항목에 대해, 점으로 축소된 사각형을 저장하는 경우, 이 함수는 점이 지정된 사각형 내에 포함되는지를 결정한다는 것을 의미합니다.

예를 들어, 아래 그림에서 음영 처리된 사각형 (1,2)-(4,7)의 내부 점들을 찾아봅시다:



[396] backend/access/gist/gistproc.c, gist_point_consistent function



검색은 루트 노드에서 시작합니다. 경계 상자는 (0,0)-(3,4)와 겹치지만, (5,3)-(9,9)와는 겹치지 않습니다. 이는 우리가 두 번째 서브트리로 내려갈 필요가 없음을 의미합니다.

다음 단계에서, 경계 상자는 (0,3)-(3,4)와 겹치고 (0,0)-(3,2)와 접촉하므로, 두 서브트리 모두를 확인해야 합니다.

리프 노드에 도달하면, 그들이 포함하고 있는 모든 점들을 확인하고 일관성 함수를 만족하는 점들을 반환하기만 하면 됩니다.

B-트리 검색은 항상 정확히 하나의 자식 노드를 선택합니다. 하지만, GiST 검색은 특히 그들의 경계 상자가 겹칠 경우 여러 서브트리를 스캔해야 할 수도 있습니다.

가장 가까운 이웃 검색

인덱스가 지원하는 대부분의 연산자(이전 예제에서 보여진 = 또는 <@와 같은)는 일반적으로 검색 연산자라고 불립니다. 왜냐하면 이들은 쿼리에서 검색 조건을 정의하기 때문입니다. 이러한 연산자들은 술어이며, 즉, 논리값을 반환합니다.

그러나 인자들 사이의 거리를 반환하는 순서 연산자라는 그룹도 있습니다. 이러한 연산자들은 `ORDER BY` 절에서 사용되며, 일반적으로 `DISTANCE ORDERABLE` 속성을 가진 인덱스에 의해 지원됩니다. 이는 당신이 지정된 수의 가장 가까운 이웃을 빠르게 찾을 수 있게 해줍니다. 이 타입의 검색은 `k-NN`, 또는 `k-nearest` 이웃 검색으로 알려져 있습니다.

예를 들어, 우리는 코스트로마[Kostroma]에서 가장 가까운 10개의 공항을 찾을 수 있습니다:

```
=> SELECT airport_code, airport_name->>'en'
FROM airports_big
ORDER BY coordinates <-> '(40.926780,57.767943)'::point
LIMIT 10;
 airport_code ¦ ?column?
--------------+-----------------------------------------------
          KMW ¦ Kostroma Sokerkino Airport
```

```
          IAR ¦ Tunoshna Airport
          IWA ¦ Ivanovo South Airport
          VGD ¦ Vologda Airport
          RYB ¦ Staroselye Airport
          GOJ ¦ Nizhny Novgorod Strigino International Airport
          CEE ¦ Cherepovets Airport
          CKL ¦ Chkalovskiy Air Base
          ZIA ¦ Zhukovsky International Airport
          BKA ¦ Bykovo Airport
(10 rows)

=> EXPLAIN (costs off) SELECT airport_code
FROM airports_big
ORDER BY coordinates <-> '(40.926780,57.767943)'::point
LIMIT 5;
                      QUERY PLAN
------------------------------------------------------------------
  Limit
      -> Index Scan using airports_gist_idx on airports_big
              Order By: (coordinates <-> '(40.92678,57.767943)'::point)
(3 rows)
```

인덱스 스캔은 결과를 하나씩 반환하며 언제든지 중단될 수 있기 때문에, 처음 몇 개의 값은 매우 빠르게 찾을 수 있습니다.

> 인덱스 지원 없이 효율적인 검색을 달성하기는 매우 어려울 것입니다. 우리는 특정 영역에 나타나는 모든 점들을 찾아야 하고, 그 다음 요청된 결과 수가 반환될 때까지 점차적으로 이 영역을 확장해야 합니다. 이는 여러 번의 인덱스 스캔을 필요로 할 것이며, 원래 영역의 크기와 그 증가분을 선택하는 문제는 말할 것도 없습니다.

시스템 카탈로그에서 연산자 유형을 확인할 수 있습니다 ("s"는 검색을 의미하고, "o"는 정렬 연산자를 나타냅니다):

```
=> SELECT amopopr::regoperator, amoppurpose, amopstrategy
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
JOIN pg_amop amop ON amopfamily = opcfamily
WHERE amname = 'gist'
AND opcname = 'point_ops'
ORDER BY amopstrategy;
           amopopr ¦ amoppurpose ¦ amopstrategy
-------------------+-------------+--------------
   <<(point,point) ¦           s ¦ 1
   >>(point,point) ¦           s ¦ 5
   ~=(point,point) ¦           s ¦ 6
```

```
  <<|(point,point) |          s | 10
  |>>(point,point) |          s | 11
  <->(point,point) |          o | 15
     <@(point,box) |          s | 28
   <^(point,point) |          s | 29
   >^(point,point) |          s | 30
 <@(point,polygon) |          s | 48
  <@(point,circle) |          s | 68
(11 rows)
```

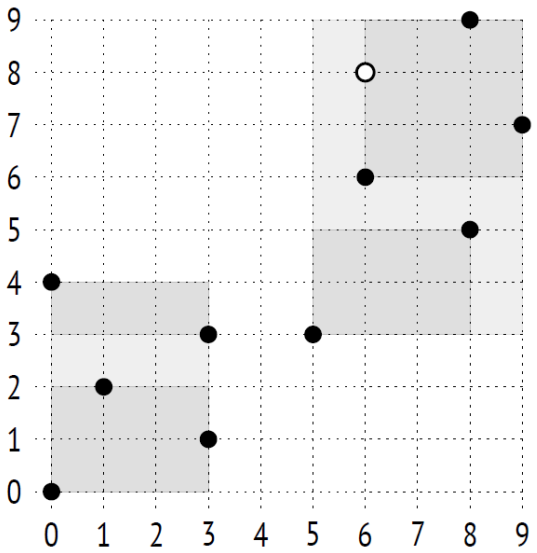
이러한 쿼리를 지원하기 위해, 연산자 클래스는 추가적인 지원 함수를 정의해야 합니다: 그것은 거리 함수로, 인덱스 항목에 저장된 값에서 다른 어떤 값까지의 거리를 계산하기 위해 인덱스 항목에 대해 호출됩니다.

인덱스된 값을 나타내는 리프 요소의 경우, 이 함수는 해당 값까지의 거리를 반환해야 합니다. 점들의 경우[397], 이것은 일반적인 유클리드 거리로, $\sqrt{(X_2 - X_1)^2 + (Y_2 - Y_1)^2}$와 같습니다.

내부 요소의 경우, 함수는 자식 리프 요소들로부터 가능한 모든 거리 중 최소값을 반환해야 합니다. 모든 자식 항목을 스캔하는 것이 상당히 비용이 많이 들기 때문에, 함수는 낙관적으로 거리를 과소평가할 수 있습니다(효율성을 희생하면서), 하지만 결코 더 큰 값을 반환해서는 안 됩니다—검색의 정확성을 저해할 수 있기 때문입니다.

따라서, 경계 상자로 표현되는 내부 요소의 경우, 점까지의 거리는 일반적인 수학적 의미에서 이해됩니다: 그것은 점과 사각형 사이의 최소 거리이거나, 점이 사각형 내부에 있는 경우 0입니다.[398] 이 값은 사각형의 모든 자식 점들을 탐색하지 않고도 쉽게 계산할 수 있으며, 이 점들 중 어느 것에 대한 거리보다 크지 않다는 것이 보장됩니다.

이제 점 (6,8)의 세 가장 가까운 이웃을 찾는 알고리즘을 고려해 봅시다:



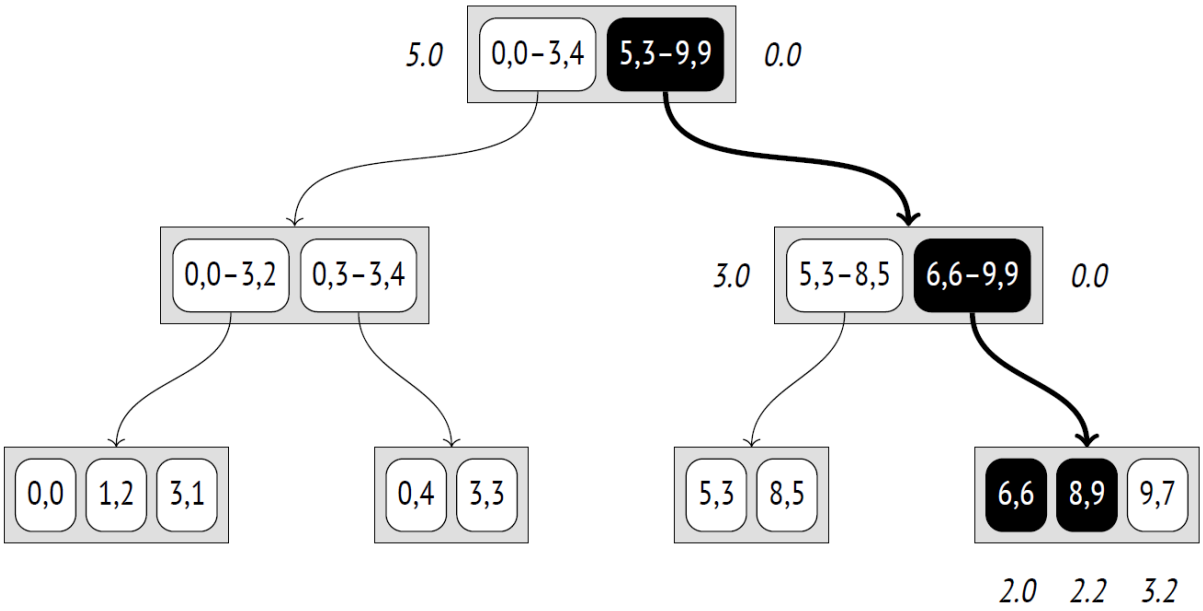
[397] backend/utils/adt/geo_ops.c, point_distance function

[398] backend/utils/adt/geo_ops.c, box_closest_point function

검색은 두 개의 경계 상자를 포함하는 루트 노드에서 시작됩니다. 지정된 점에서 사각형 (0,0)-(3,4)까지의 거리는 사각형의 모서리 (3,4)까지의 거리로 간주되며, 이는 5.0과 같습니다. (5,3)-(9,9)까지의 거리는 0.0입니다. (여기서 모든 값을 첫 번째 소수점 자리까지 반올림할 것입니다; 이 예시에서는 이 정도의 정확도가 충분합니다.)

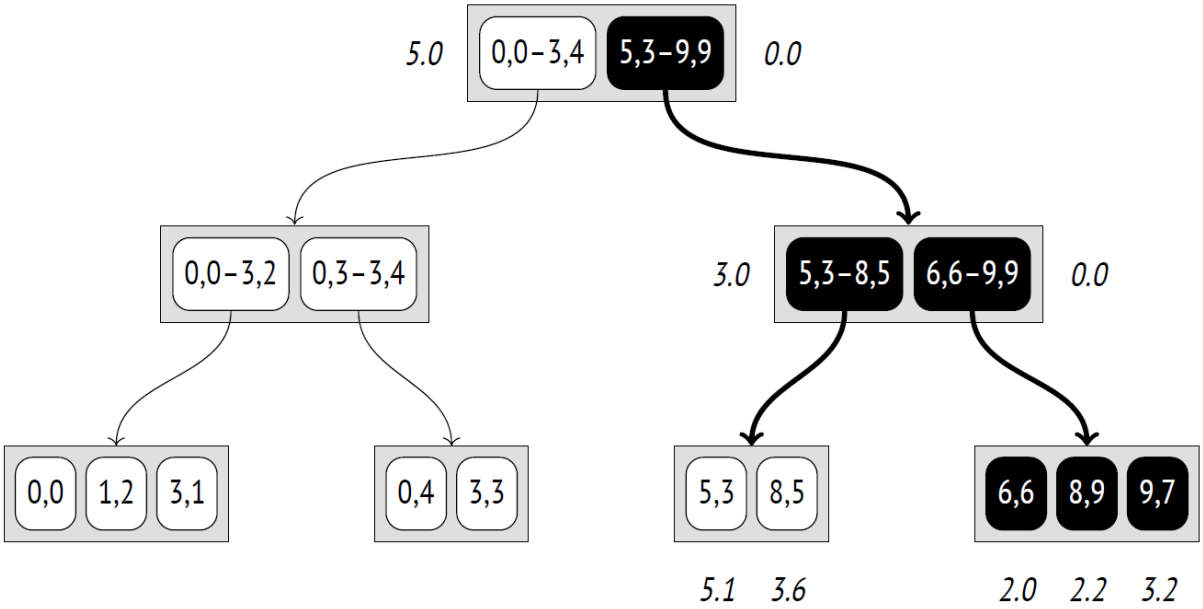
자식 노드는 거리가 증가하는 순서대로 순회됩니다. 따라서, 우리는 우선 오른쪽 자식 노드로 내려가는데, 여기에는 두 개의 사각형: (5,3)-(8,5)와 (6,6)-(9,9)가 포함되어 있습니다. 첫 번째 사각형까지의 거리는 3.0이고, 두 번째 사각형까지의 거리는 0.0입니다.

다시 한번, 우리는 오른쪽 하위 트리를 선택하고 세 개의 점을 포함하는 리프 노드에 도달합니다: (6,6)은 거리가 2.0, (8,9)는 거리가 2.2, 그리고 (9,7)은 거리가 3.2입니다.



그러므로 우리는 첫 번째 두 점인 (6,6)과 (8,9)를 얻었습니다. 그러나 이 노드의 세 번째 점까지의 거리는 사각형 (5,3)-(8,5)까지의 거리보다 큽니다.

그래서 이제 우리는 두 점을 포함하는 왼쪽 자식 노드로 내려가야 합니다. 점 (8,5)까지의 거리는 3.6이고, 점 (5,3)까지의 거리는 5.1입니다. 이전 자식 노드의 점 (9,7)이 왼쪽 하위 트리의 어느 노드보다도 점 (6,8)에 더 가까움이 밝혀졌으므로, 우리는 이를 세 번째 결과로 반환할 수 있습니다.

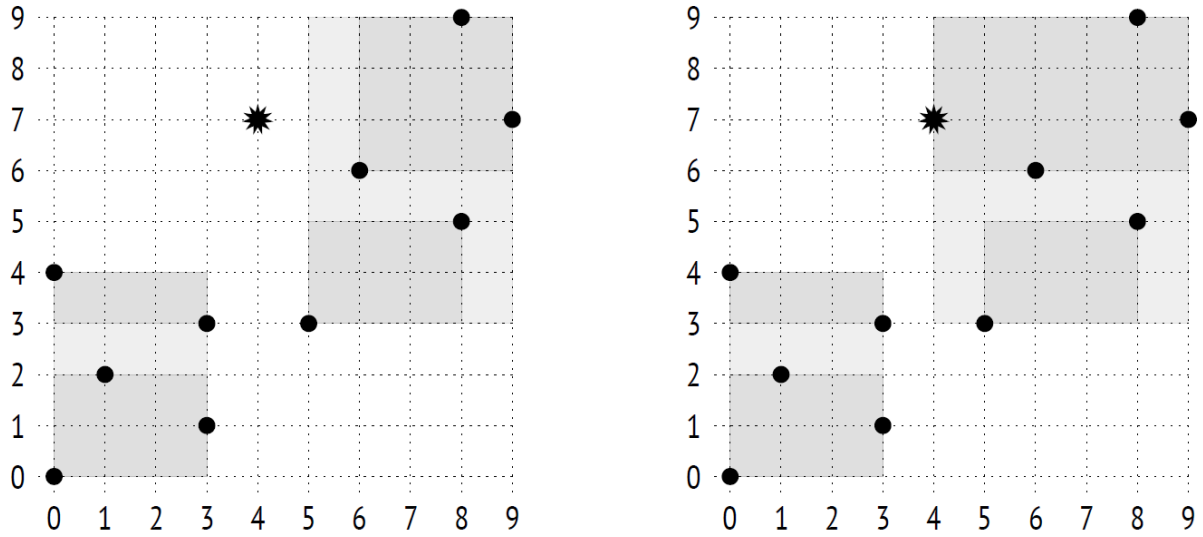


이 예시는 내부 항목에 대한 거리 함수가 충족해야 하는 요구 사항을 보여줍니다. 사각형 (5,3)-(8,5)까지의 거리가 줄어든(3.6 대신 3.0) 때문에 추가 노드를 스캔해야 했으며, 그 결과 검색 효율성이 감소했습니다; 그러나 알고리즘 자체는 정확하게 유지되었습니다.

삽입

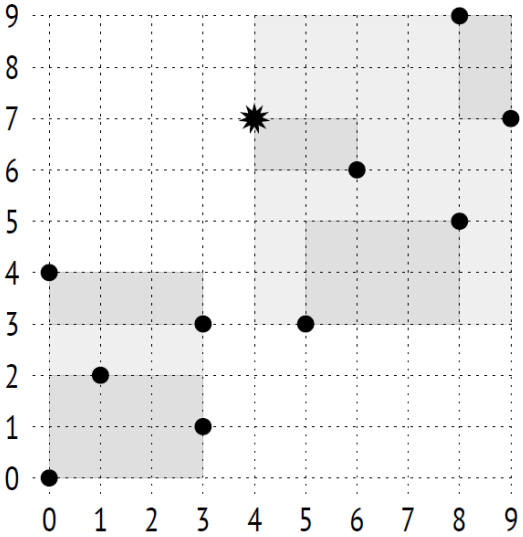
새로운 키가 R-트리에 삽입될 때, 이 키를 위해 사용될 노드는 패널티 함수에 의해 결정됩니다: 경계 상자의 크기는 가능한 한 적게 증가해야 합니다.[399]

예를 들어, 점 (4,7)은 사각형 (5,3)-(9,9)에 추가될 것입니다. 왜냐하면 그 면적이 단 6단위만 증가하기 때문이며, 사각형 (0,0)-(3,4)는 12단위가 증가해야 할 것입니다. 다음(리프) 레벨에서, 점은 동일한 논리를 따라 사각형 (6,6)-(9,9)에 추가될 것입니다.



[399] backend/access/gist/gistproc.c, gist_box_penalty function

페이지가 최대 세 개의 요소를 보유할 수 있다고 가정할 때, 페이지는 둘로 나누어져야 하며, 요소들은 새 페이지들 사이에 분배되어야 합니다. 이 예제에서 결과는 명백해 보이지만, 일반적인 경우 데이터 분배 작업은 그렇게 단순하지 않습니다. 무엇보다도, picksplit 함수는 경계 상자들 사이의 중첩을 최소화하려고 시도하며, 이는 더 작은 사각형과 페이지들 사이의 점들의 균등한 분포를 목표로 합니다.[400]



제외 제약 조건

GiST 인덱스는 배타적 제약 조건에서도 사용될 수 있습니다.

배타적 제약 조건은 힙 튜플의 지정된 필드가 어떤 연산자의 의미에서 서로 일치하지 않는다는 것을 보장합니다. 다음 조건들이 만족되어야 합니다:

- 배타적 제약 조건은 인덱싱 방법에 의해 지원되어야 합니다(`CAN EXCLUDE` 속성).
- 연산자는 이 인덱싱 방법의 연산자 클래스에 속해야 합니다.
- 연산자는 교환 가능해야 합니다. 즉, "a 연산자 b = b 연산자 a"가 참이어야 합니다.

위에서 고려된 해시 및 btree 접근 방식에서 유일하게 적합한 연산자는 동등함(equal to)입니다. 이는 배타적 제약 조건을 사실상 고유한 제약 조건으로 전환하는데, 이는 특별히 유용하지 않습니다.

`GiST` 방법은 두 가지 더 적용 가능한 전략을 가지고 있습니다:

- 중첩overlapping: && 연산자
- 인접adjacency: -|- 연산자 (간격에 대해 정의됨)

이를 시험해보기 위해, 공항들을 서로 너무 가까이 배치하지 못하게 하는 제약 조건을 생성해봅시다. 이 조건은 다음과 같이 정의할 수 있습니다: 공항의 좌표에 중심이 있는 특정 반경의 원들이 중첩되어서는 안 됩니다:

```
=> ALTER TABLE airports_data ADD EXCLUDE
USING gist (circle(coordinates,0.2) WITH &&);
```

[400] backend/access/gist/gistproc.c, gist_box_picksplit function

```
=> INSERT INTO airports_data(
airport_code, airport_name, city, coordinates, timezone
) VALUES (
'ZIA', '{}', '{"en": "Moscow"}', point(38.1517, 55.5533),
'Europe/Moscow'
);
ERROR: conflicting key value violates exclusion constraint "airports_data_circle_excl"
DETAIL: Key (circle(coordinates, 0.2::double precision))=(<(38.1517,55.5533),0.2>)
conflicts with existing key
(circle(coordinates, 0.2::double
precision))=(<(37.90629959106445,55.40879821777344),0.2>).
```

제약 조건이 정의될 때, 그것을 강제하는 인덱스가 자동적으로 추가됩니다. 여기서는 표현식 위에 구축된 `GiST` 인덱스입니다.

더 복잡한 예를 살펴보겠습니다. 공항들이 서로 가까이 있을 수 있되, 같은 도시에 속할 경우에만 허용해야 한다고 가정해 봅시다. 가능한 해결책은 새로운 무결성 제약 조건을 정의하는 것입니다. 이는 다음과 같이 표현될 수 있습니다: 원들의 교차(&&)가 있는 행의 쌍이 있으면 안 되며, 그 중심이 공항의 좌표에 위치하고 해당 도시 이름이 다를 경우(!=) 금지됩니다.

이러한 제약 조건을 생성하려는 시도는 텍스트 데이터 유형에 대한 연산자 클래스가 없기 때문에 오류를 발생시킵니다:

```
=> ALTER TABLE airports_data
DROP CONSTRAINT airports_data_circle_excl; -- delete old data
=> ALTER TABLE airports_data ADD EXCLUDE USING gist (
circle(coordinates,0.2) WITH &&,
(city->>'en') WITH !=
);
ERROR: data type text has no default operator class for access
method "gist"
HINT: You must specify an operator class for the index or define a default operator
class for the data type.
```

그러나, `GiST`는 **엄밀하게 왼쪽에, 엄밀하게 오른쪽에, 동일함과 같은 전략**을 제공하며, 이는 숫자나 텍스트 문자열과 같은 정규 순서 데이터 유형에도 적용될 수 있습니다. `btree_gist` 확장은 일반적으로 B-트리와 함께 사용되는 연산에 대한 `GiST` 지원을 구현하기 위해 특별히 고안되었습니다:

```
=> CREATE EXTENSION btree_gist;
=> ALTER TABLE airports_data ADD EXCLUDE USING gist (
circle(coordinates,0.2) WITH &&,
(city->>'en') WITH !=
);
ALTER TABLE
```

제약 조건이 생성되었습니다. 이제 모스크바 공항들이 너무 가까워서 같은 이름의 도시에 속한 주코프스키 공항을 추가할 수 없습니다:

```
=> INSERT INTO airports_data(
airport_code, airport_name, city, coordinates, timezone
) VALUES (
'ZIA', '{}', '{"en": "Zhukovsky"}', point(38.1517, 55.5533),
'Europe/Moscow'
);
ERROR: conflicting key value violates exclusion constraint
"airports_data_circle_expr_excl"
DETAIL: Key (circle(coordinates, 0.2::double precision), (city ->>
'en'::text))=(<(38.1517,55.5533),0.2>, Zhukovsky) conflicts with
existing key (circle(coordinates, 0.2::double precision), (city ->>
'en'::text))=(<(37.90629959106445,55.40879821777344),0.2>, Moscow).
```

하지만 이 공항의 도시로 모스크바를 지정한다면 그것을 할 수 있습니다:

```
=> INSERT INTO airports_data(
airport_code, airport_name, city, coordinates, timezone
) VALUES (
'ZIA', '{}', '{"en": "Moscow"}', point(38.1517, 55.5533),
'Europe/Moscow'
);
INSERT 0 1
```

`GiST`가 **'크다'**, **'작다'**, 그리고 **'동일하다'** 연산을 지원함에도 불구하고, B-트리가 이러한 측면에서 훨씬 더 효율적이며, 특히 값의 범위에 접근할 때 그렇다는 것을 기억하는 것이 중요합니다. 그러므로 위에서 보여준 btree_gist 확장을 사용하는 트릭은 `GiST` 인덱스가 다른 정당한 이유로 정말 필요한 경우에만 사용하는 것이 합리적입니다.

속성

액세스 메서드 속성. `GiST` 방법의 속성은 다음과 같습니다:

```
=> SELECT a.amname, p.name, pg_indexam_has_property(a.oid, p.name)
FROM pg_am a, unnest(array[
'can_order', 'can_unique', 'can_multi_col',
'can_exclude', 'can_include'
]) p(name)
WHERE a.amname = 'gist';
 amname |          name | pg_indexam_has_property
--------+---------------+-------------------------
   gist |     can_order | f
   gist |    can_unique | f
   gist | can_multi_col | t
```

```
  gist ¦ can_exclude ¦ t
  gist ¦ can_include ¦ t
(5 rows)
```

고유 제약 조건과 정렬은 지원되지 않습니다.

`GiST` 인덱스는 추가적인 `INCLUDE` 열을 포함하여 생성될 수 있습니다.

우리가 알고 있듯이, 여러 열에 대해 인덱스를 구축하고, 무결성 제약 조건에서 이를 사용할 수 있습니다.

인덱스 수준의 속성. 이러한 속성들은 인덱스 수준에서 정의됩니다:

```
=> SELECT p.name, pg_index_has_property('airports_gist_idx', p.name)
FROM unnest(array[
'clusterable', 'index_scan', 'bitmap_scan', 'backward_scan'
]) p(name);
          name ¦ pg_index_has_property
---------------+------------------------
   clusterable ¦ t
    index_scan ¦ t
   bitmap_scan ¦ t
 backward_scan ¦ f
(4 rows)
```

`GiST` 인덱스는 클러스터링에 사용될 수 있습니다.

데이터 검색 방법에 있어서, 일반적인 (행별) 인덱스 스캔과 비트맵 스캔 모두 지원됩니다. 그러나, GiST 인덱스의 역방향 스캔은 허용되지 않습니다.

칼럼 수준의 속성. 대부분의 열 속성은 접근 방식 수준에서 정의되며, 그대로 유지됩니다:

```
=> SELECT p.name,
pg_index_column_has_property('airports_gist_idx', 1, p.name)
FROM unnest(array[
'orderable', 'search_array', 'search_nulls'
]) p(name);
                 name ¦ pg_index_column_has_property
----------------------+-------------------------------------
             orderable ¦ f
          search_array ¦ f
          search_nulls ¦ t
(3 rows)
```

모든 정렬 관련 속성은 비활성화됩니다.

NULL 값은 허용되지만, GiST는 NULL 값을 효율적으로 처리하는 데 있어서 그다지 효과적이지 않습니다. NULL 값은 경계 상자를 증가시키지 않는 것으로 가정되며, 이러한 값은 무작위 하위 트리에 삽입되어 전체 트리에서 검색해야 합니다.

그러나 몇 가지 열 수준 속성은 특정 연산자 클래스에 따라 달라집니다:

```
=> SELECT p.name,
pg_index_column_has_property('airports_gist_idx', 1, p.name)
FROM unnest(array[
'returnable', 'distance_orderable'
]) p(name);
               name | pg_index_column_has_property
--------------------+------------------------------
         returnable | t
 distance_orderable | t
(2 rows)
```

리프 노드가 전체 인덱스 키를 유지하기 때문에 index-only 스캔이 허용됩니다.

위에서 보았듯이, 이 연산자 클래스는 가장 가까운 이웃 검색을 위한 거리 연산자를 제공합니다. NULL 값에 대한 거리는 NULL로 간주되며, 이러한 값은 마지막으로 반환됩니다(B-트리의 NULLS LAST 절과 유사).

그러나 범위 유형(즉, 면적이 아닌 선형 기하학을 나타내는 세그먼트)에 대한 거리 연산자는 없으므로, 이러한 유형에 대해 구축된 인덱스에 대해서는 이 속성이 다릅니다:

```
=> CREATE TABLE reservations(during tsrange);
=> CREATE INDEX ON reservations USING gist(during);
=> SELECT p.name,
pg_index_column_has_property('reservations_during_idx', 1, p.name)
FROM unnest(array[
'returnable', 'distance_orderable'
]) p(name);
               name | pg_index_column_has_property
--------------------+------------------------------
         returnable | t
 distance_orderable | f
(2 rows)
```

26.3 전체 텍스트 검색을 위한 RD-트리

전체 텍스트 검색에 관하여

전체 텍스트 검색[401]의 목적은 제공된 집합에서 검색 쿼리와 일치하는 문서를 선택하는 것입니다.

[401] postgresql.org/docs/14/textsearch.html

검색될 문서는 문서 내에서의 어휘와 그 위치를 포함하는 tsvector 타입으로 변환됩니다. 어휘는 검색에 적합한 형식으로 변환된 단어들입니다. 기본적으로, 모든 단어는 소문자로 정규화되며, 그 끝이 잘립니다:

```
=> SET default_text_search_config = english;
=> SELECT to_tsvector(
'No one can tell me, nobody knows, ' ||
'Where the wind comes from, where the wind goes.'
);
                    to_tsvector
-----------------------------------------------------------------------
'come':11 'goe':16 'know':7 'nobodi':6 'one':2 'tell':4 'wind':10,15
(1 row)
```

일명 불용어stop words (예: "the" 또는 "from" 같은)는 필터링됩니다: 이들은 검색 결과가 의미 있는 결과를 반환하기에는 너무 자주 발생한다고 가정됩니다. 당연히, 이러한 변환은 모두 설정 가능합니다.

검색 쿼리는 다른 타입인 `tsquery`로 표현됩니다. 모든 쿼리에는 논리 연결자: & (AND), | (OR), ! (NOT)로 묶인 하나 이상의 어휘가 포함됩니다. 연산자 우선순위를 정의하기 위해 괄호를 사용할 수도 있습니다.

```
=> SELECT to_tsquery('wind & (comes | goes)');
          to_tsquery
-----------------------------
'wind' & ( 'come' | 'goe' )
(1 row)
```

전체 텍스트 검색에 사용되는 유일한 연산자는 일치 연산자 @@입니다:

```
=> SELECT amopopr::regoperator, oprcode::regproc, amopstrategy
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
JOIN pg_amop amop ON amopfamily = opcfamily
JOIN pg_operator opr ON opr.oid = amopopr
WHERE amname = 'gist'
AND opcname = 'tsvector_ops'
ORDER BY amopstrategy;
              amopopr |     oprcode | amopstrategy
----------------------+-------------+--------------
 @@(tsvector,tsquery) | ts_match_vq | 1
(1 row)
```

이 연산자는 문서가 쿼리를 만족하는지 여부를 결정합니다. 예를 들어보겠습니다:

```
=> SELECT to_tsvector('Where the wind comes from, where the wind goes')
@@ to_tsquery('wind & coming');
?column?
----------
```

```
    t
(1 row)
```

이것은 전체 텍스트 검색에 대한 전면적인 설명이 결코 아니지만, 이 정보는 색인 기초를 이해하는 데 충분해야 합니다.

tsvector 데이터 인덱싱

빠르게 작동하기 위해서는 전체 텍스트 검색이 인덱스에 의해 지원되어야 합니다.[402] 문서 자체가 아니라 `tsvector` 값이 인덱싱되기 때문에 여기에는 두 가지 옵션이 있습니다: 하나는 표현식에 대한 인덱스를 구축하고 타입 캐스트를 수행하거나, 또 다른 하나는 `tsvector` 타입의 별도 컬럼을 추가하고 이 컬럼을 인덱싱하는 것입니다. 첫 번째 접근 방식의 이점은 실제로 필요하지 않은 `tsvector` 값에 대한 저장 공간을 낭비하지 않는다는 것입니다. 그러나 인덱싱 엔진이 접근 방식에 의해 반환된 모든 힙 튜플을 재확인해야 하기 때문에 두 번째 옵션보다 느립니다. 이는 재확인된 각 행에 대해 `tsvector` 값이 다시 계산되어야 함을 의미하며, 곧 보게 될 것처럼, `GiST`는 모든 행을 재확인합니다.

간단한 예를 구성해 보겠습니다. 우리는 두 개의 컬럼을 가진 테이블을 생성할 것입니다: 첫 번째 컬럼은 문서를 저장할 것이고, 두 번째 컬럼은 `tsvector` 값을 보유할 것입니다. 두 번째 컬럼을 업데이트하기 위해 트리거를 사용할 수 있지만[403], 이 컬럼을 생성된 컬럼으로 선언하는 것이 더 편리합니다:[404]

```
=> CREATE TABLE ts(
  doc text,
  doc_tsv tsvector GENERATED ALWAYS AS (
             to_tsvector('pg_catalog.english', doc)
             ) STORED
  );

=> CREATE INDEX ts_gist_idx ON ts
USING gist(doc_tsv);
```

> 위의 예시에서, 저는 단일 인자를 사용하여 `to_tsvector` 함수를 사용했고, 기본 텍스트 검색 설정(default: english) 매개변수를 설정하여 전체 텍스트 검색 구성을 정의했습니다. 이 함수 버전의 변동성 범주는 `STABLE` 입니다, 왜냐하면 이것은 매개변수 값에 암시적으로 의존하기 때문입니다. 하지만 여기서 저는 구성을 명시적으로 정의하는 다른 버전을 적용합니다; 이 버전은 `IMMUTABLE` 이며 생성 표현식에서 사용될 수 있습니다.

몇건의 데이터를 넣어 보면:

```
=> INSERT INTO ts(doc)
VALUES
('Old MacDonald had a farm'),
```

---

[402] postgresql.org/docs/14/textsearch-indexes.html

[403] postgresql.org/docs/14/textsearch-features#TEXTSEARCH-UPDATE-TRIGGERS.html

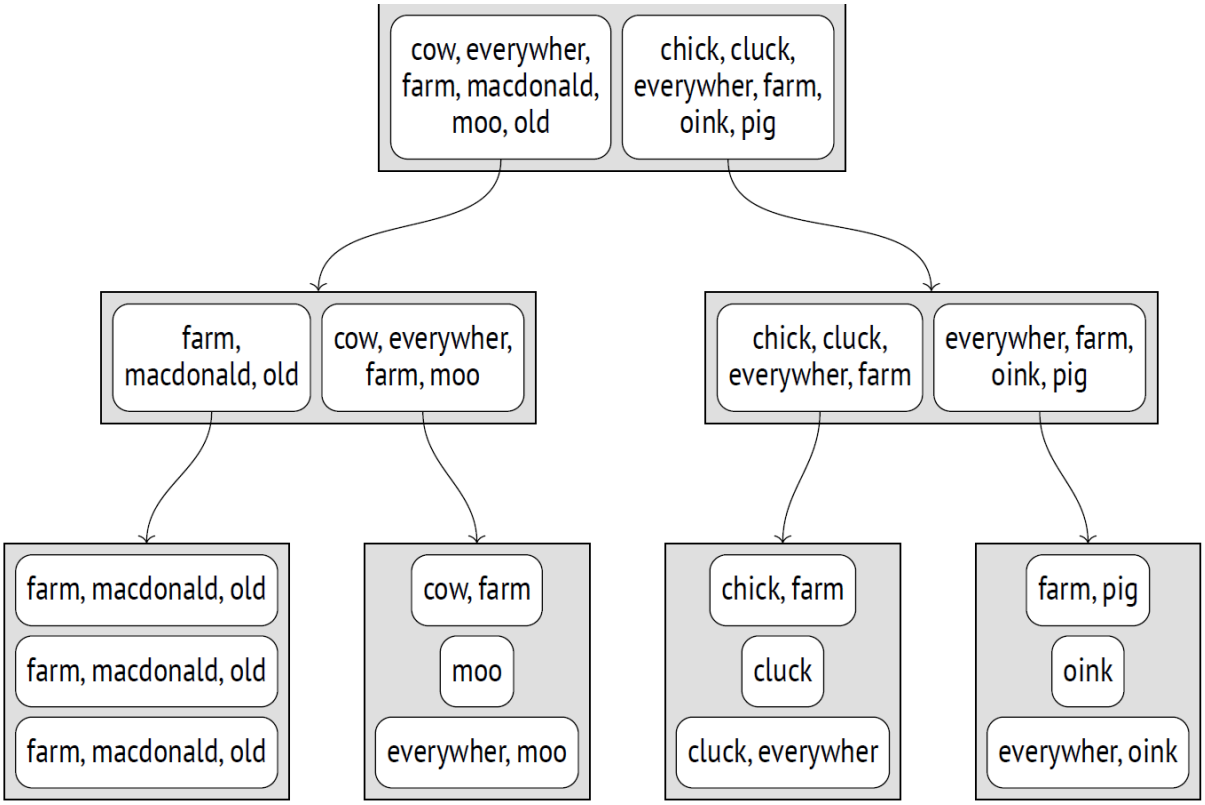
[404] postgresql.org/docs/14/ddl-generated-columns.html

```
('And on his farm he had some cows'),
('Here a moo, there a moo'),
('Everywhere a moo moo'),
('Old MacDonald had a farm'),
('And on his farm he had some chicks'),
('Here a cluck, there a cluck'),
('Everywhere a cluck cluck'),
('Old MacDonald had a farm'),
('And on his farm he had some pigs'),
('Here an oink, there an oink'),
('Everywhere an oink oink')
RETURNING doc_tsv;
doc_tsv
-------------------------------
'farm':5 'macdonald':2 'old':1
'cow':8 'farm':4
'moo':3,6
'everywher':1 'moo':3,4
'farm':5 'macdonald':2 'old':1
'chick':8 'farm':4
'cluck':3,6
'cluck':3,4 'everywher':1
'farm':5 'macdonald':2 'old':1
'farm':4 'pig':8
'oink':3,6
'everywher':1 'oink':3,4
(12 rows)
INSERT 0 12
```

따라서, R-트리는 문서 인덱싱에 적합하지 않습니다. 왜냐하면 경계 상자의 개념이 그들에게는 의미가 없기 때문입니다. 그래서 RD-트리(러시안 돌) 수정이 사용됩니다. 경계 상자 대신, 이러한 트리는 경계 집합을 사용하는데, 즉 자식 집합들의 모든 요소를 포함하는 집합입니다. 전체 텍스트 검색의 경우, 이러한 집합은 문서의 어휘들을 포함하지만, 일반적인 경우에는 경계 집합이 임의로 될 수 있습니다.

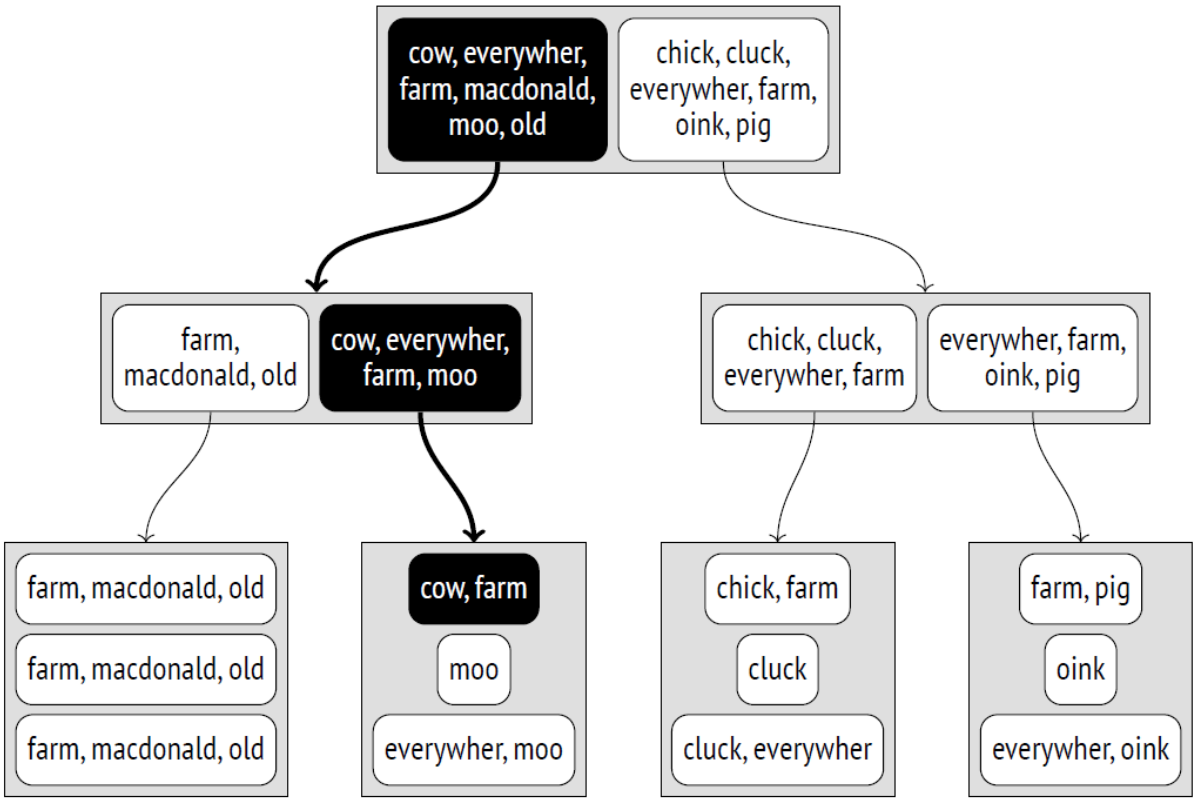
인덱스 항목에서 경계 집합을 나타내는 몇 가지 방법이 있습니다. 가장 간단한 방법은 집합의 모든 요소를 열거하는 것입니다.

여기에 그 예시가 있습니다:



`'DOC_TSV @@ TO_TSQUERY('COW')'` 조건을 만족하는 문서를 찾기 위해서는, "`cow`" 어휘를 포함하고 있는 것으로 알려진 자식 항목들이 있는 노드로 내려가야 합니다.

이러한 표현의 문제점은 명백합니다. 문서에 있는 어휘의 수가 엄청날 수 있고, 페이지 크기는 제한되어 있습니다. 각각의 문서가 별도로 취급될 때 너무 많은 독특한 어휘를 가지고 있지 않더라도, 트리의 상위 레벨에서 그들의 결합된 집합은 여전히 너무 클 수 있습니다.



전체 텍스트 검색은 더욱 압축된 **시그니처 트리**라는 다른 솔루션을 사용합니다. 이는 블룸 필터를 다뤄본 사람이라면 익숙해야 할 것입니다.

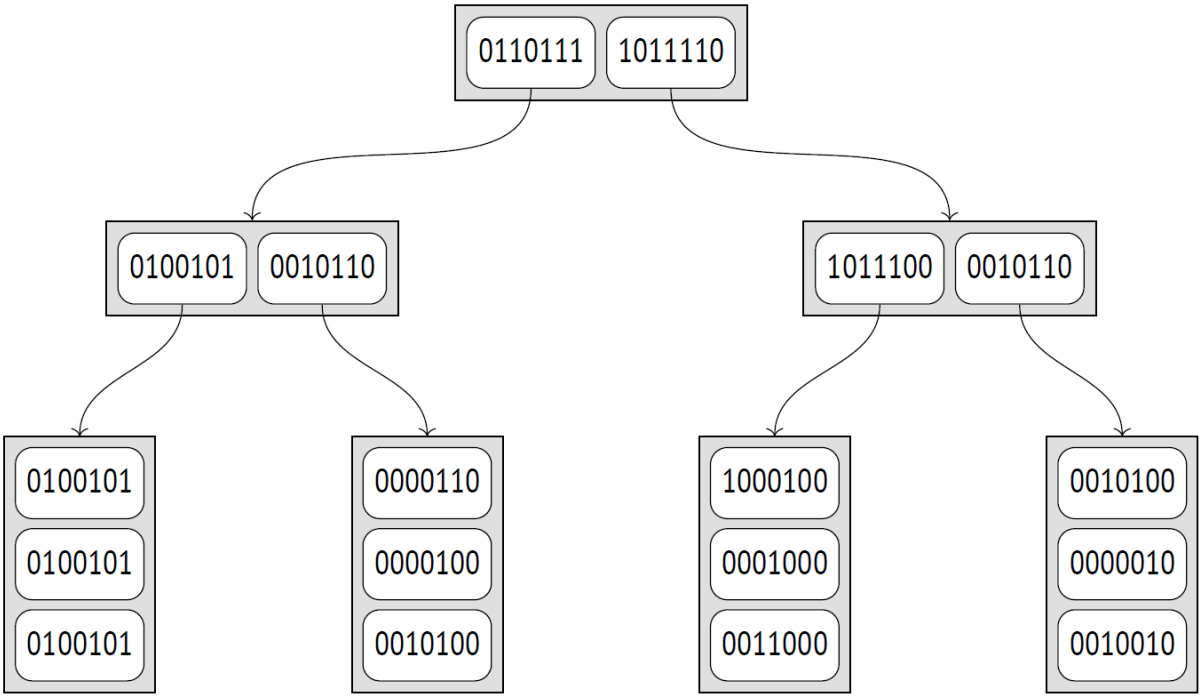
각 어휘는 그것의 **시그니처**로 표현될 수 있는데, 특정 길이의 비트 문자열에서 오직 하나의 비트만 1로 설정됩니다. 설정되어야 할 비트는 해당 어휘의 해시 함수에 의해 결정됩니다.

문서의 시그니처는 이 문서 내 모든 어휘의 시그니처에 대한 비트별 `OR` 연산의 결과입니다.

Suppose we have assigned the following signatures to our lexemes:	chick	1000000
	cluck	0001000
	cow	0000010
	everywher	0010000
	farm	0000100
	macdonald	0100000
	moo	0000100
	oink	0000010
	old	0000001
	pig	0010000
Then the documents' signatures will be as follows:	Old MacDonald had a farm	0100101
	And on his farm he had some cows	0000110
	Here a moo, there a moo	0000100
	Everywhere a moo moo	0010100
	And on his farm he had some chicks	1000100

```
Here a cluck, there a cluck          0001000
Everywhere a cluck cluck             0011000
And on his farm he had some pigs     0010100
Here an oink, there an oink          0000010
Everywhere an oink oink              0010010
```

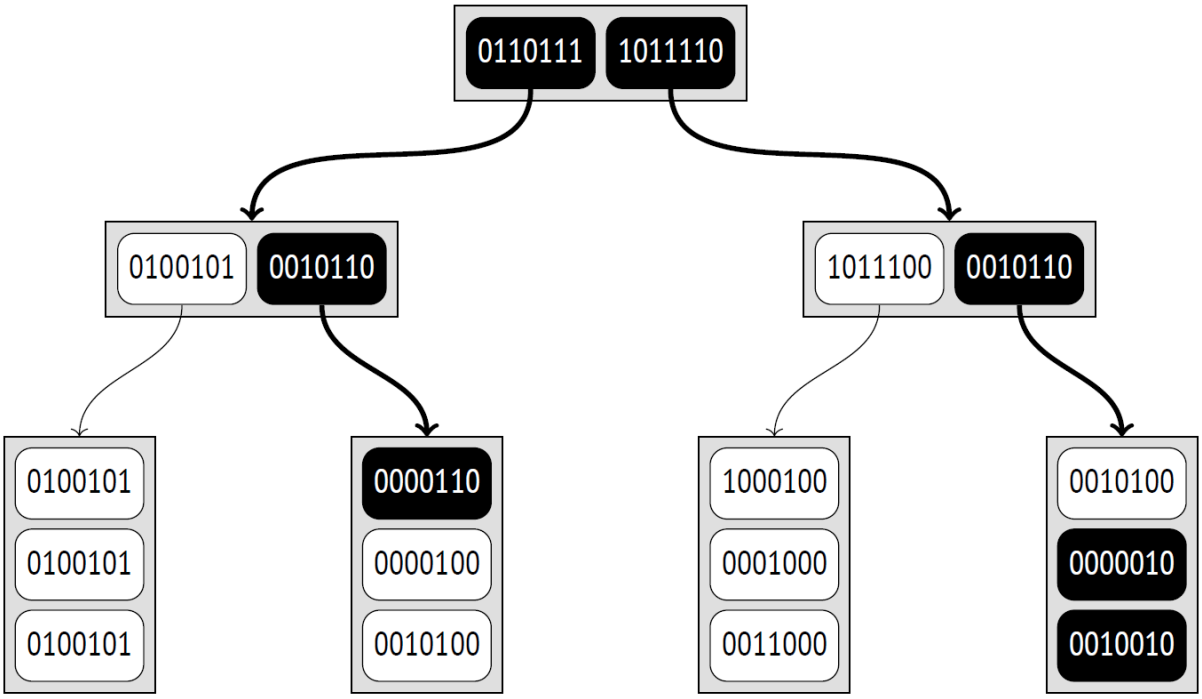
하지만 우리는 이 공항의 도시를 모스크바로 지정한다면 그것을 할 수 있습니다:



이 접근법의 장점은 명백합니다: 인덱스 항목의 크기가 동일하며, 상당히 작아서 인덱스가 매우 컴팩트하게 됩니다. 하지만 몇 가지 단점도 있습니다. 우선, 인덱스가 더 이상 인덱스 키를 저장하지 않기 때문에 인덱스-오직 스캔을 수행할 수 없으며, 반환된 각 TID는 테이블에 의해 재확인되어야 합니다. 정확성도 영향을 받습니다: 인덱스는 많은 거짓 양성을 반환할 수 있으며, 이는 재확인하는 동안 필터링되어야 합니다.

`'DOC_TSV @@ DOC_TSQUERY('COWS')'` 조건을 다시 한 번 살펴봅시다. 쿼리의 시그니처는 문서의 시그니처와 같은 방식으로 계산되며, 이 특정 경우에는 0000010과 같습니다. 일관성 함수[405]는 자신의 시그니처에서 동일한 비트가 설정된 모든 자식 노드를 찾아야 합니다.

[405] backend/utils/adt/tsgistidx.c, gtsvector_consistent function



이전 예시와 비교했을 때, 거짓 양성 타격 때문에 더 많은 노드를 스캔해야 합니다. 시그니처의 용량이 제한되어 있기 때문에, 큰 세트의 어휘 중 일부는 반드시 동일한 시그니처를 가지게 됩니다. 이 예시에서, 그러한 어휘들은 "cow"와 "oink"입니다. 이는 하나의 동일한 시그니처가 다른 문서들과 일치할 수 있음을 의미합니다; 여기서 쿼리의 시그니처는 그 중 세 개와 일치합니다.

거짓 양성은 인덱스의 효율성을 줄이지만, 어떤 방식으로도 그 정확성에 영향을 주지 않습니다: 거짓 음성이 확실히 배제되므로, 필요한 값이 누락될 가능성은 없습니다.

명확하게, 실제 시그니처 크기는 실제로 더 큽니다. 기본적으로, 이는 124바이트(992비트)를 차지하므로, 이 예시에서와 같이 충돌의 확률은 훨씬 낮습니다. 필요하다면, 연산자 클래스 매개변수를 사용하여 시그니처 크기를 대략 2000바이트까지 더 증가시킬 수 있습니다.

```
CREATE INDEX ... USING gist(column tsvector_ops(siglen = size));
```

게다가, 값들이 충분히 작은 경우(표준 페이지에 대해 대략 500바이트를 차지하는 페이지의 1/16보다 약간 작을 때)[406], `tsvector_ops` 연산자 클래스는 인덱스의 리프 페이지에 그들의 시그니처가 아닌 `tsvector` 값 자체를 유지합니다. 실제 데이터에서 인덱싱이 어떻게 작동하는지 보려면, `pgsql-hackers` 메일링 리스트 아카이브를 살펴볼 수 있습니다. 이는 발송 날짜, 제목, 저자 이름, 그리고 본문 텍스트를 포함한 356,125개의 이메일을 담고 있습니다.[407]

`tsvector` 타입의 컬럼을 추가하고 인덱스를 구축해 봅시다. 여기서 저는 세 가지 값을(제목, 저자, 본문 텍스트) 하나의 벡터로 결합하여, 문서가 동적으로 생성될 수 있으며 단일 컬럼에 저장될 필요가 없음을 보여주고자 합니다.

```
=> ALTER TABLE mail_messages ADD COLUMN tsv tsvector
```

[406] backend/utils/adt/tsgistidx.c, gtsvector_compress function

[407] edu.postgrespro.ru/mail_messages.sql.gz

```
GENERATED ALWAYS AS ( to_tsvector(
'pg_catalog.english', subject||' '||author||' '||body_plain
) ) STORED;
NOTICE: word is too long to be indexed
DETAIL: Words longer than 2047 characters are ignored.
...
NOTICE: word is too long to be indexed
DETAIL: Words longer than 2047 characters are ignored.
ALTER TABLE

=> CREATE INDEX mail_gist_idx ON mail_messages USING gist(tsv);
=> SELECT pg_size_pretty(pg_relation_size('mail_gist_idx'));
pg_size_pretty
----------------
127 MB
(1 row)
```

칼럼이 채워지는 동안, 그 크기 때문에 가장 큰 단어들 중 일정 수가 필터링되었습니다. 하지만 인덱스가 준비되면, 검색 쿼리에서 사용될 수 있습니다:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT *
FROM mail_messages
WHERE tsv @@ to_tsquery('magic & value');
              QUERY PLAN
-------------------------------------------------------------
  Index Scan using mail_gist_idx on mail_messages
      (actual rows=898 loops=1)
      Index Cond: (tsv @@ to_tsquery('magic & value'::text))
      Rows Removed by Index Recheck: 7859
(4 rows)
```

조건을 만족하는 898개의 행과 함께, 접근 방식은 나중에 재확인을 통해 필터링될 7859개의 행도 반환했습니다. 시그니처 용량을 늘리면, 정확성(그리고 따라서 인덱스의 효율성)이 향상되지만 인덱스 크기도 커질 것입니다:

```
=> DROP INDEX mail_messages_tsv_idx;
=> CREATE INDEX ON mail_messages
USING gist(tsv tsvector_ops(siglen=248));
=> SELECT pg_size_pretty(pg_relation_size('mail_messages_tsv_idx'));
pg_size_pretty
----------------
139 MB
(1 row)
```

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT *
FROM mail_messages
WHERE tsv @@ to_tsquery('magic & value');
              QUERY PLAN
-------------------------------------------------------
  Index Scan using mail_messages_tsv_idx on mail_messages
      (actual rows=898 loops=1)
      Index Cond: (tsv @@ to_tsquery('magic & value'::text))
      Rows Removed by Index Recheck: 2060
(4 rows)
```

속성

저는 이미 접근 방식의 속성을 보여드렸고, 그 중 대부분은 모든 연산자 클래스에 대해 동일합니다. 하지만 다음의 두 가지 컬럼 수준 속성은 언급할 가치가 있습니다:

```
=> SELECT p.name,
pg_index_column_has_property('mail_messages_tsv_idx', 1, p.name)
FROM unnest(array[
'returnable', 'distance_orderable'
]) p(name);
               name ¦ pg_index_column_has_property
--------------------+-------------------------------
         returnable ¦ f
 distance_orderable ¦ f
(2 rows)
```

이제 인덱스-오직 스캔은 불가능하게 되었습니다. 왜냐하면 원래의 값은 그것의 시그니처로부터 복원될 수 없기 때문입니다. 이 특정한 경우에는 완벽하게 괜찮습니다: tsvector 값은 검색을 위해서만 사용되며, 우리는 문서 자체를 검색해야 합니다.

`tsvector_ops` 클래스를 위한 순서 연산자도 정의되어 있지 않습니다.

26.4 기타 데이터 유형

저는 단지 가장 두드러진 두 가지 예시만 고려했습니다. 이들 예시는 `GiST` 방법이 균형 잡힌 트리를 기반으로 하고 있음에도 불구하고, 다양한 연산자 클래스에서 다른 지원 함수 구현 덕분에 다양한 데이터 유형에 사용될 수 있음을 보여줍니다. 우리가 `GiST` 인덱스에 대해 이야기할 때, 우리는 항상 연산자 클래스를 명시해야 합니다. 왜냐하면 이것은 인덱스의 속성에 있어 매우 중요하기 때문입니다.

`GiST` 액세스 방법에 의해 현재 지원되는 몇 가지 더 많은 데이터 유형이 있습니다.

기하학적 데이터 유형. 점 외에도, `GiST`는 다른 기하학적 객체: 사각형, 원, 다각형을 인덱싱할 수 있습니다. 이 목적을 위해 이러한 객체들은 모두 그들의 경계 상자로 표현됩니다.

큐브 확장은 같은 이름의 데이터 유형을 추가합니다. 이것은 다차원 큐브를 나타냅니다. 그것들은 해당 차원의 경계 상자를 사용하여 R-트리를 사용하여 인덱싱됩니다.

범위 유형. PostgreSQL은 `int4range` 및 `tstzrange`와 같은 여러 내장 숫자 및 시간 범위 유형을 제공합니다.[408] `CREATE TYPE AS RANGE` 명령을 사용하여 사용자 정의 범위 유형을 정의할 수 있습니다.

표준 및 사용자 정의 모든 범위 유형은 `range_ops` 연산자 클래스를 통해 `GiST`에 의해 지원됩니다.[409] 인덱싱을 위해, 일차원 R-트리가 적용됩니다: 이 경우, 경계 상자는 경계 세그먼트로 변환됩니다.
멀티레인지 유형 또한 지원됩니다. 이들은 `multirange_ops` 클래스에 의존합니다. 경계 범위는 멀티레인지 값의 일부인 모든 범위를 포함합니다.

`seg` 확장은 특정 정확도로 정의된 경계를 가진 간격에 대한 같은 이름의 데이터 유형을 제공합니다. 이것은 범위 유형으로 간주되지 않지만, 사실상 그렇기 때문에 정확히 동일한 방식으로 인덱싱됩니다.

서수형 데이터 유형. 다시 한 번 `btree_gist` 확장을 회상해봅시다: 이것은 `GiST` 방법이 다양한 서수형 데이터 유형을 지원할 수 있도록 연산자 클래스를 제공합니다. 이러한 유형은 일반적으로 B-트리에 의해 인덱싱됩니다. 이러한 연산자 클래스는 한 열의 데이터 유형이 B-트리에 의해 지원되지 않을 때 다중 열 인덱스를 구축하는 데 사용될 수 있습니다.

네트워크 주소 유형. `inet` 데이터 유형은 `inet_ops`[410] 연산자 클래스를 통해 구현된 내장 `GiST` 지원을 가지고 있습니다.

정수 배열. `intarray` 확장은 정수 배열의 기능을 확장하여 `GiST` 지원을 추가합니다. 두 가지 연산자 클래스가 있습니다. 작은 배열의 경우, 인덱스 항목에서 키의 전체 표현을 구현하는 RD-트리를 사용하는 `gist__int_ops`를 사용할 수 있습니다. 큰 배열은 `gist__bigint_ops` 연산자 클래스를 기반으로 하는 더 컴팩트하지만 덜 정확한 서명 RD-트리의 이점을 볼 수 있습니다.

> 기본 유형의 배열 이름에 포함된 추가 밑줄은 연산자 클래스의 이름에 속합니다. 예를 들어, 보다 일반적인 `int4[]` 표기법과 함께, 정수 배열은 `_int4`로 표시될 수 있습니다. 그러나 `_int`와 `_bigint` 유형은 존재하지 않습니다.

Ltree. `Ltree` 확장은 레이블을 가진 트리 형태의 구조를 위한 동일한 이름의 데이터 유형을 추가합니다. `GiST` 지원은 `ltree` 값에 대해 `gist_ltree_ops` 연산자 클래스를 사용하고, `ltree` 유형의 배열에 대해 `gist__ltree_ops` 연산자 클래스를 사용하는 `signature` RD-트리를 통해 제공됩니다.

키-값 저장소. `Hstore` 확장은 키-값 쌍을 저장하기 위한 `hstore` 데이터 유형을 제공합니다.
`Gist_hstore_ops` 연산자 클래스는 서명 RD-트리를 기반으로 한 인덱스 지원을 구현합니다.

Trigrams. `Pg_trgm` 확장은 텍스트 문자열 비교 및 와일드카드 검색을 위한 인덱스 지원을 구현하는 `gist_trgm_ops` 클래스를 추가합니다.

---

[408] postgresql.org/docs/14/rangetypes.html

[409] backend/utils/adt/rangetypes_gist.c

[410] backend/utils/adt/network_gist.c

27 장. SP-GiST

27.1 개요

SP-GiST 이름의 첫 글자는 공간 분할Space Partitioning을 의미합니다. 여기서의 공간은 검색이 수행되는 임의의 값 집합으로 이해되며, 반드시 단어의 전통적인 의미(예: 이차원 평면)의 공간일 필요는 없습니다. 이름 속의 GiST 부분은 GiST와 SP-GiST 방법 사이에 특정 유사성을 시사합니다: 두 방법 모두 일반화된 검색 트리이며 다양한 데이터 유형의 색인화를 위한 프레임워크로 작용합니다.

SP-GiST 방법의 아이디어[411]는 검색 공간을 여러 개의 서로 겹치지 않는 영역으로 분할하는 것이며, 이 영역들은 다시 하위 영역으로 재귀적으로 분할될 수 있습니다. 이러한 분할은 B-트리와 GiST 트리와 다른 비균형 트리를 생성하며, 쿼드트리, k-D 트리, 레디스 트리(트라이)와 같은 잘 알려진 구조를 구현하는 데 사용될 수 있습니다.

비균형 트리는 일반적으로 가지가 적고, 그 결과 깊이가 큽니다. 예를 들어, 쿼드트리 노드는 최대 네 개의 자식 노드를 가질 수 있으며, k-D 트리의 노드는 두 개만 가질 수 있습니다. 트리가 메모리에 유지되는 경우 문제가 되지 않지만, 디스크에 저장될 때는 트리 노드를 페이지에 가능한 한 밀집되게 포장하여 I/O를 최소화해야 하며, 이 작업은 그리 단순하지 않습니다. B-트리와 GiST 인덱스는 각각의 트리 노드가 전체 페이지를 차지하기 때문에 이에 대해 신경 쓸 필요가 없습니다.

SP-GiST 트리의 내부 노드는 모든 자식 노드에 대해 참인 조건을 만족하는 값을 포함합니다. 이러한 값은 종종 접두사로 불리며, GiST 인덱스의 술어와 같은 역할을 합니다. SP-GiST 자식 노드로의 포인터는 라벨을 가질 수 있습니다.

리프 노드 요소는 인덱싱된 값을(또는 그 일부분을) 및 해당 TID를 포함합니다.

GiST와 마찬가지로, SP-GiST 접근 방법은 동시 접근, 잠금 및 로깅과 같은 저수준 세부 사항을 처리하는 주요 알고리즘만을 구현합니다. 공간 분할의 새로운 데이터 타입 및 알고리즘은 연산자 클래스 인터페이스를 통해 추가될 수 있습니다. 연산자 클래스는 대부분의 로직을 제공하며 인덱싱 기능의 많은 측면을 정의합니다.

SP-GiST에서 검색은 루트 노드에서 시작하는 깊이 우선 검색입니다.[412] 내려갈 가치가 있는 노드는 GiST에서 사용되는 것과 유사한 일관성 함수에 의해 선택됩니다. 트리의 내부 노드에 대해, 이 함수는 검색 조건과 모순되지 않는 자식 노드의 집합을 반환합니다. 일관성 함수는 이러한 노드로 내려가지 않으며, 단지 해당 라벨과 접두사를 평가합니다. 리프 노드의 경우, 이 노드의 인덱싱된 값이 검색 조건과 일치하는지를 결정합니다.

균형이 맞지 않는 트리에서는 분기 깊이에 따라 검색 시간이 달라질 수 있습니다.

값을 SP-GiST 인덱스에 삽입하는 데 참여하는 두 가지 지원 함수가 있습니다. 루트 노드에서 트리를 탐색하

---

411 postgresql.org/docs/14/spgist.html
backend/access/spgist/README

412 backend/access/spgist/spgscan.c, spgWalk function

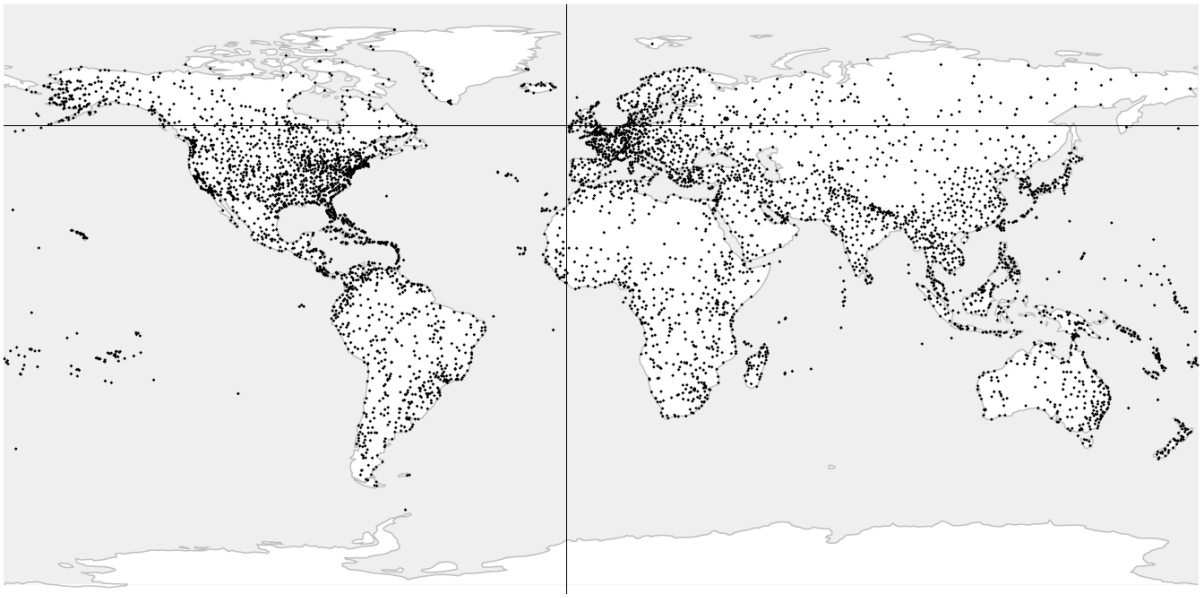
는 동안, choose 함수는 다음 결정 중 하나를 내립니다: 새 값이 기존 자식 노드로 보내지거나, 이 값에 대해 새로운 자식 노드를 생성하거나, 현재 노드를 분할합니다(만약 값이 이 노드의 접두사와 일치하지 않는 경우). 선택된 리프 페이지에 충분한 공간이 없는 경우, picksplit 함수는 어떤 노드가 새 페이지로 이동해야 하는지 결정합니다.

이제 이러한 알고리즘을 설명하기 위해 몇 가지 예를 제공하겠습니다.

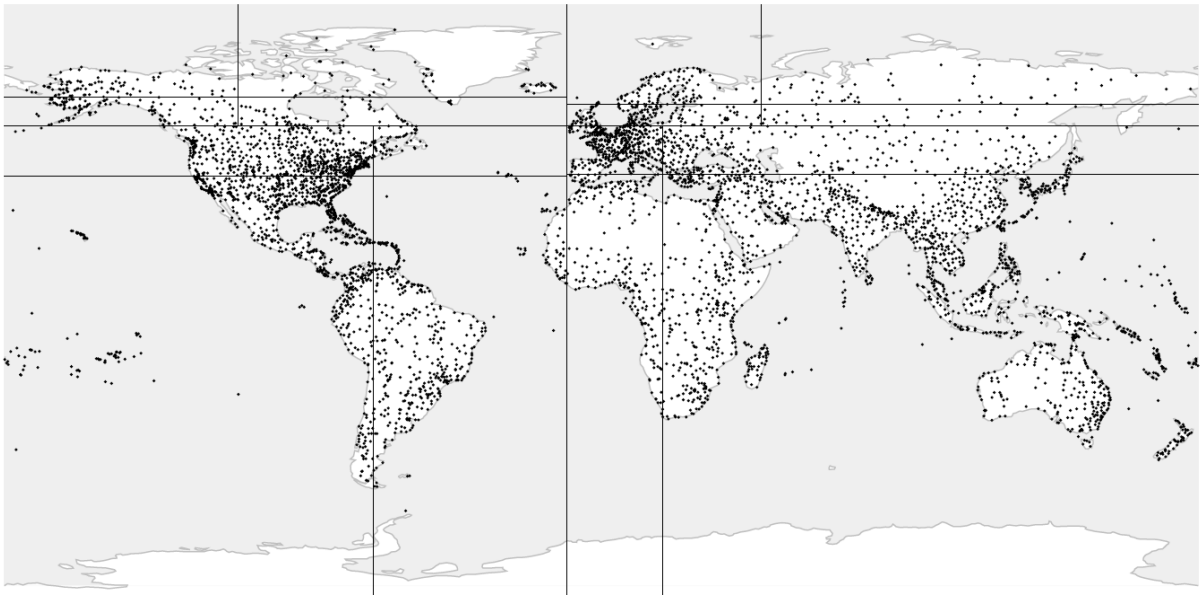
27.2 포인트용 쿼드트리

쿼드트리는 2차원 평면 상의 점들을 색인하기 위해 사용됩니다. 이 평면은 선택된 점을 기준으로 네 영역(사분면)으로 재귀적으로 분할됩니다. 이 점을 중심점이라고 하며, 자식 값들의 위치를 정의하는 조건, 즉 노드 접두사 역할을 합니다.

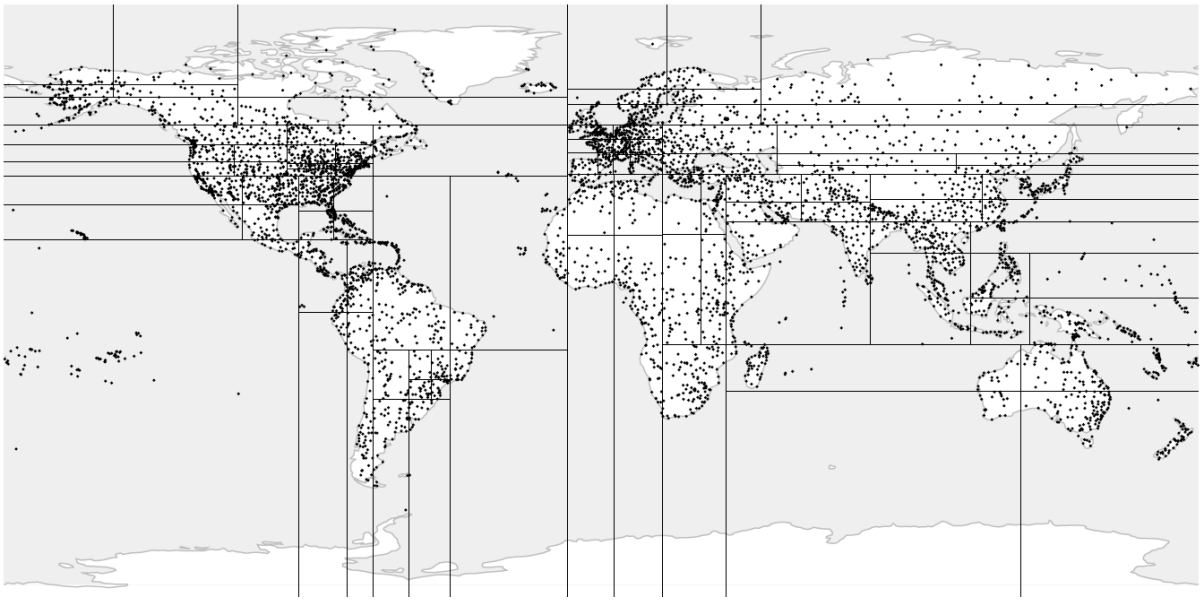
루트 노드는 평면을 네 사분면으로 분할합니다.



그 다음 각각의 사분면은 자신의 사분면으로 더 세분화됩니다.



이 절차는 원하는 분할 수에 도달할 때까지 계속됩니다.



이 예시는 확장된 공항 테이블을 기반으로 구축된 인덱스를 사용합니다. 그림은 해당 사분면에서의 점 밀도에 따라 분기 깊이가 달라짐을 보여줍니다. 시각적 명확성을 위해 저장 파라미터인 fillfactor(기본값: 80)의 작은 값을 설정하여, 트리를 더 깊게 만들었습니다:

```
CREATE INDEX airports_quad_idx ON airports_big
USING spgist(coordinates) WITH (fillfactor = 10);
```

점에 대한 기본 연산자 클래스는 `quad_point_ops`입니다.

연산자 클래스

이미 SP-GiST의 지원 함수에 대해 언급했습니다:[413] 검색을 위한 일관성 함수와 삽입을 위한 picksplit 함수입니다.
이제 quad_point_ops 연산자 클래스[414]의 지원 함수 목록을 살펴보겠습니다. 모든 함수는 필수적입니다.

```
=> SELECT amprocnum, amproc::regproc
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
JOIN pg_amproc amop ON amprocfamily = opcfamily
WHERE amname = 'spgist'
AND opcname = 'quad_point_ops'
ORDER BY amprocnum;
 amprocnum ¦ amproc
-----------+---------------------------
         1 ¦ spg_quad_config
         2 ¦ spg_quad_choose
         3 ¦ spg_quad_picksplit
         4 ¦ spg_quad_inner_consistent
         5 ¦ spg_quad_leaf_consistent
(5 rows)
```

이 함수들은 다음과 같은 작업을 수행합니다:

1. config 함수는 연산자 클래스에 대한 기본 정보를 접근 방식에 보고합니다.
2. choose 함수는 삽입을 위한 노드를 선택합니다.
3. picksplit 함수는 페이지 분할 후 노드를 페이지 사이에 분배합니다.
4. inner_consistent 함수는 내부 노드의 값이 검색 조건을 만족하는지 확인합니다.
5. leaf_consistent 함수는 리프 노드에 저장된 값이 검색 조건을 만족하는지 결정합니다.

또한 몇 가지 선택적 기능들이 있습니다.

quad_point_ops 연산자 클래스는 GiST와 동일한 전략을 지원합니다:[415]

```
=> SELECT amopopr::regoperator, oprcode::regproc, amopstrategy
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
JOIN pg_amop amop ON amopfamily = opcfamily
JOIN pg_operator opr ON opr.oid = amopopr
WHERE amname = 'spgist'
AND opcname = 'quad_point_ops'
ORDER BY amopstrategy;
```

[413] postgresql.org/docs/14/spgist-extensibility.html

[414] backend/access/spgist/spgquadtreeproc.c

[415] include/access/stratnum.h

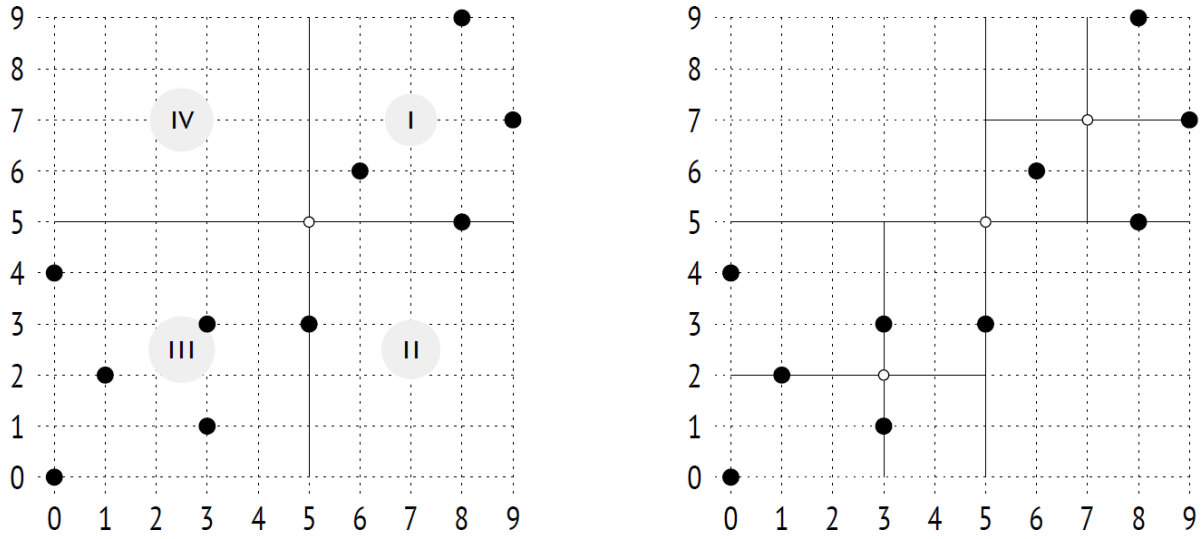
```
          amopopr ¦        oprcode ¦ amopstrategy
------------------+----------------+--------------
  <<(point,point) ¦     point_left ¦ 1
  >>(point,point) ¦    point_right ¦ 5
  ~=(point,point) ¦       point_eq ¦ 6
    <@(point,box) ¦          on_pb ¦ 8
 <<¦(point,point) ¦    point_below ¦ 10
 ¦>>(point,point) ¦    point_above ¦ 11
 <->(point,point) ¦ point_distance ¦ 15
  <^(point,point) ¦    point_below ¦ 29
  >^(point,point) ¦    point_above ¦ 30
(9 rows)
```

예를 들어, 위의 연산자 >^를 사용하여 딕슨 북쪽에 위치한 공항을 찾을 수 있습니다:

```
=> SELECT airport_code, airport_name->>'en'
FROM airports_big
WHERE coordinates >^ '(80.3817,73.5167)'::point;
 airport_code ¦ ?column?
--------------+---------------------------
          THU ¦ Thule Air Base
          YEU ¦ Eureka Airport
          YLT ¦ Alert Airport
          YRB ¦ Resolute Bay Airport
          LYR ¦ Svalbard Airport, Longyear
          NAQ ¦ Qaanaaq Airport
          YGZ ¦ Grise Fiord Airport
          DKS ¦ Dikson Airport
(8 rows)
=> EXPLAIN (costs off) SELECT airport_code
FROM airports_big
WHERE coordinates >^ '(80.3817,73.5167)'::point;
                      QUERY PLAN
------------------------------------------------------------------
  Bitmap Heap Scan on airports_big
      Recheck Cond: (coordinates >^ '(80.3817,73.5167)'::point)
      -> Bitmap Index Scan on airports_quad_idx
              Index Cond: (coordinates >^ '(80.3817,73.5167)'::point)
(4 rows)
```

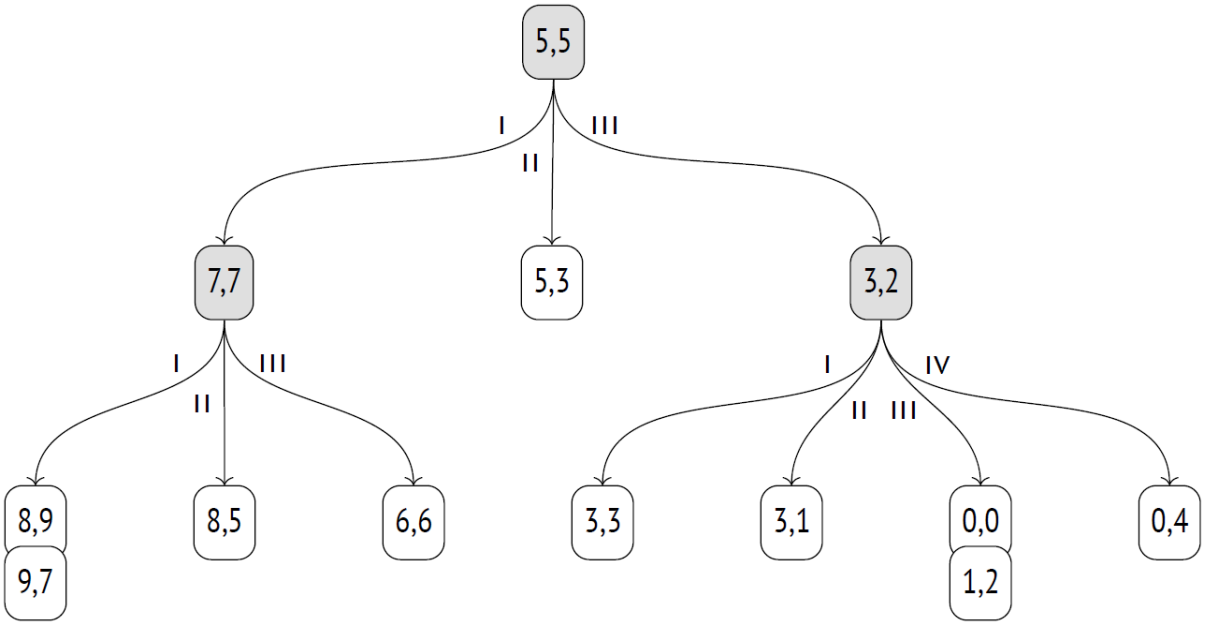
쿼드트리의 구조와 내부 작업에 대해 좀 더 자세히 살펴보겠습니다. GiST와 관련된 장에서 논의한 것과 동일한 간단한 예제를 사용할 것입니다.

이 경우에 비행기는 다음과 같이 분할될 수 있습니다:



왼쪽 그림은 트리 레벨 중 하나에서 사분면 번호를 보여줍니다; 이어지는 그림에서는 명확성을 위해 자식 노드를 왼쪽에서 오른쪽으로 같은 순서대로 배치할 것입니다. 경계선에 있는 점들은 더 작은 번호의 사분면에 포함됩니다. 오른쪽 그림은 최종 분할을 보여줍니다.

아래에서 이 인덱스의 가능한 구조를 볼 수 있습니다. 각 내부 노드는 최대 네 개의 자식 노드를 참조하며, 각각의 포인터는 사분면 번호로 라벨이 붙어 있습니다.



페이지 구조

SP-GiST 인덱스는 B-tree나 GiST 인덱스와 다르게 트리 노드와 페이지 간에 일대일 대응이 없습니다. 내부 노드는 일반적으로 많은 자식을 가지고 있지 않기 때문에 여러 노드를 하나의 페이지에 패킹해야 합니다. 다른 유형의 노드는 다른 페이지에 저장됩니다: 내부 노드는 내부 페이지에 저장되며, 잎 노드는 잎 페이지로 갑니다.

내부 페이지에 저장된 인덱스 항목은 접두사로 사용되는 값을 가지고 있으며 자식 노드로의 포인터 집합을 가지고 있습니다; 각 포인터는 레이블을 동반할 수 있습니다.

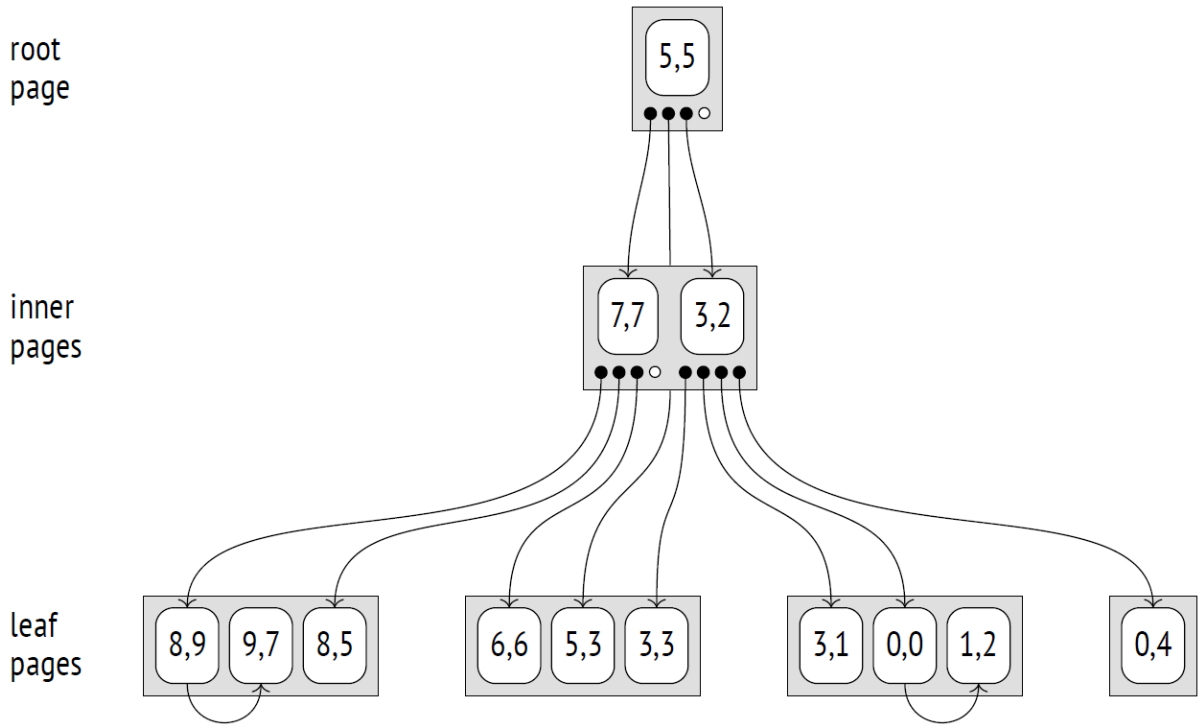
잎 페이지 항목은 값과 TID로 구성됩니다.

특정 내부 노드에 관련된 모든 잎 노드는 단일 페이지에 함께 저장되며 리스트로 묶입니다. 페이지가 더 이상 노드를 수용할 수 없는 경우, 이 리스트는 다른 페이지로 이동[416]하거나 페이지가 분할될 수 있지만, 어떤 경우에도 리스트가 여러 페이지에 걸쳐 있지 않습니다.

공간을 절약하기 위해 알고리즘은 이 페이지들이 완전히 채워질 때까지 같은 페이지에 새 노드를 추가하려고 합니다. 마지막 페이지 번호는 백엔드에 의해 캐싱되며 주기적으로 메타페이지라고 불리는 제로 페이지에 저장됩니다. 메타페이지는 B-tree에서 볼 수 있었던 루트 노드에 대한 참조를 포함하지 않으며, SP-GiST 인덱스의 루트는 항상 첫 페이지에 위치합니다.

> 불행히도 pageinspect 확장 기능은 SP-GiST 를 탐색하는 기능을 제공하지 않지만, gevel 이라는 외부 확장 기능을 사용할 수 있습니다.[417] 이 기능을 `pageinspect` 에 통합하려는 시도가 있었으나 성공하지 못했습니다.[418]

우리의 예제로 돌아가 봅시다. 아래 그림은 트리 노드가 페이지 사이에 어떻게 분배될 수 있는지 보여줍니다. `quad_point_ops` 연산자 클래스는 실제로 라벨을 사용하지 않습니다. 노드는 최대 네 개의 자식 노드를 가질 수 있기 때문에, 인덱스는 고정된 크기의 네 포인터 배열을 유지하며, 그 중 일부는 비어 있을 수 있습니다.



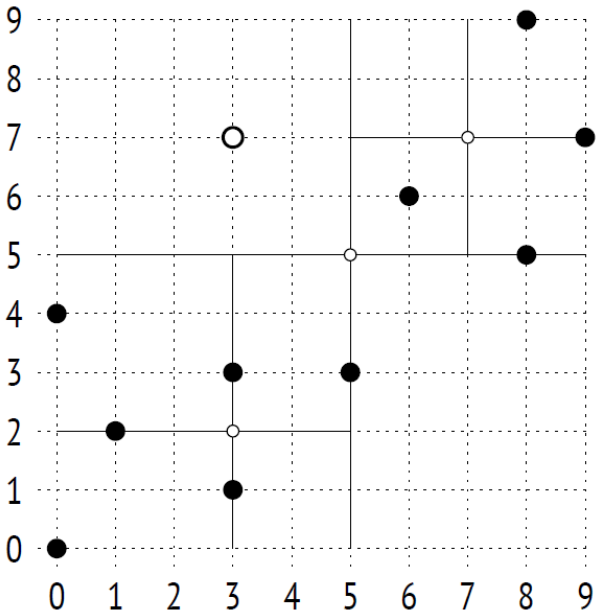
[416] backend/access/spgist/spgdoinsert.c, moveLeafs function

[417] sigaev.ru/git/gitweb.cgi?p=gevel.git

[418] commitfest.postgresql.org/15/1207

탐색

동일한 예를 사용하여 점 (3,7) 위에 위치한 점들을 검색하는 알고리즘을 살펴보겠습니다.

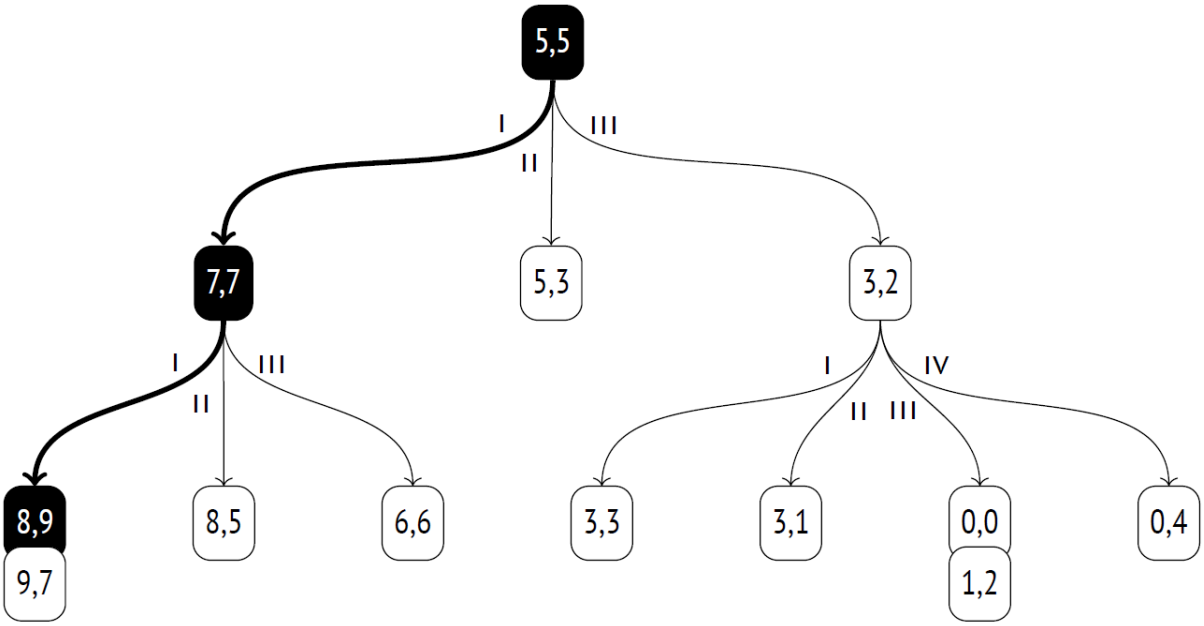


검색은 루트에서 시작합니다. 내부 일관성 함수는 내려갈 자식 노드를 결정합니다. 점 (3,7)은 루트 노드의 중심점(5,5)과 비교하여 찾고자 하는 점들이 있을 수 있는 사분면을 선택합니다; 이 예에서는 사분면 I과 IV입니다.

중심점이 (7,7)인 노드 내부에서 다시 내려갈 자식 노드를 선택해야 합니다. 이들은 사분면 I과 IV에 속하지만, 사분면 IV가 비어 있으므로 하나의 잎 노드만 확인하면 됩니다. 잎 일관성 함수[419]는 이 노드의 점들을 쿼리에서 지정한 점(3,7)과 비교합니다. 위 조건은 (8,9)에만 만족합니다.

이제 우리는 한 단계 돌아가서 루트 노드의 사분면 IV에 해당하는 노드를 확인하기만 하면 됩니다. 이는 비어 있으므로 검색이 완료됩니다.

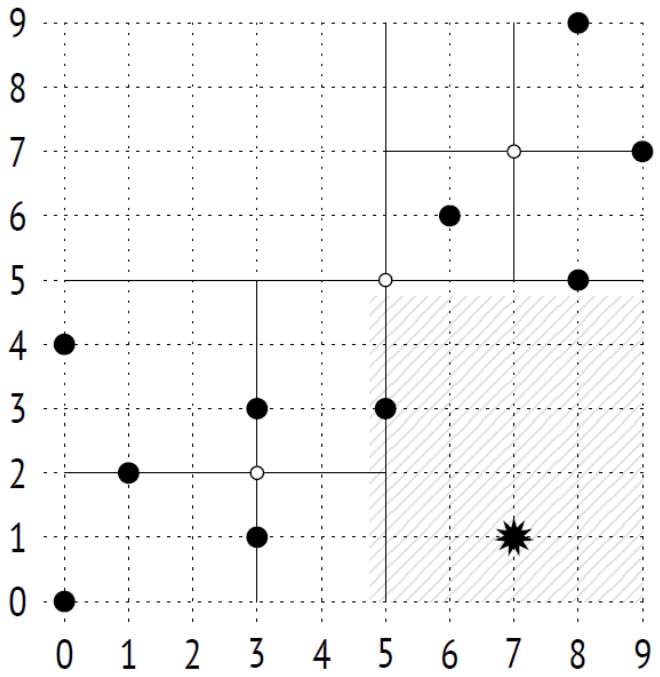
[419] backend/access/spgist/spgquadtreeproc.c, spg_quad_leaf_consistent function



삽입

값이 SP-GiST 트리[420]에 삽입될 때, 그 후에 이루어지는 모든 행동은 선택 함수[421]에 의해 결정됩니다. 이 특별한 경우에, 그것은 단순히 점을 해당하는 사분면의 기존 노드 중 하나로 지시합니다.

예를 들어, 값 (7,1)을 추가해 봅시다:

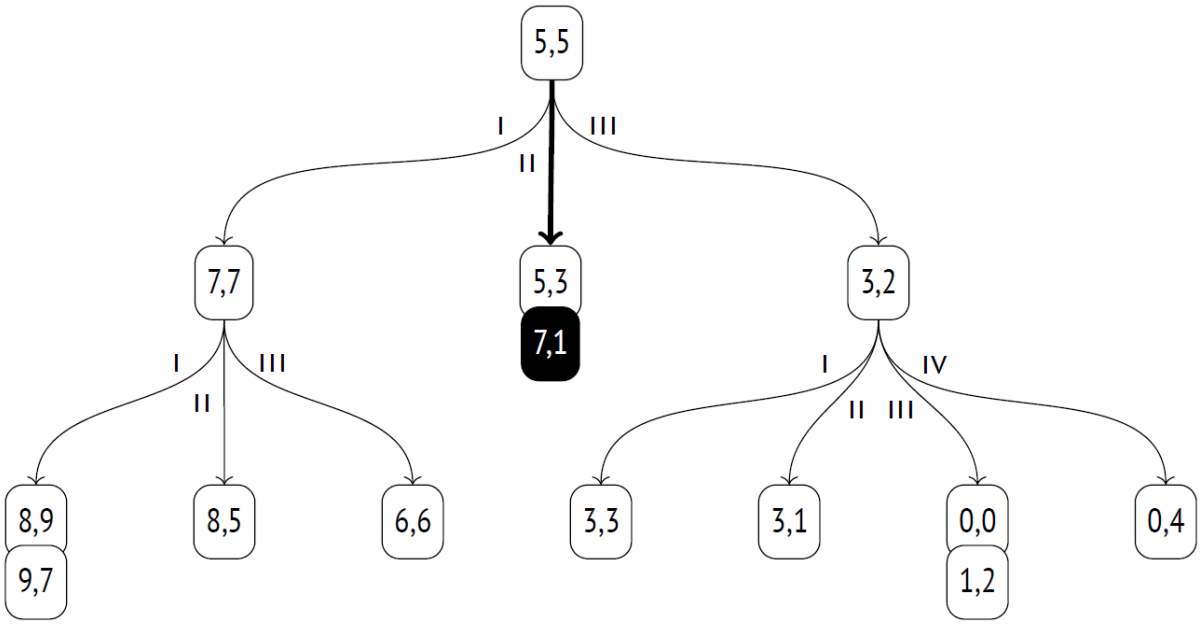


값은 2사분면에 속하며, 해당하는 트리 노드에 추가될 것입니다:

---

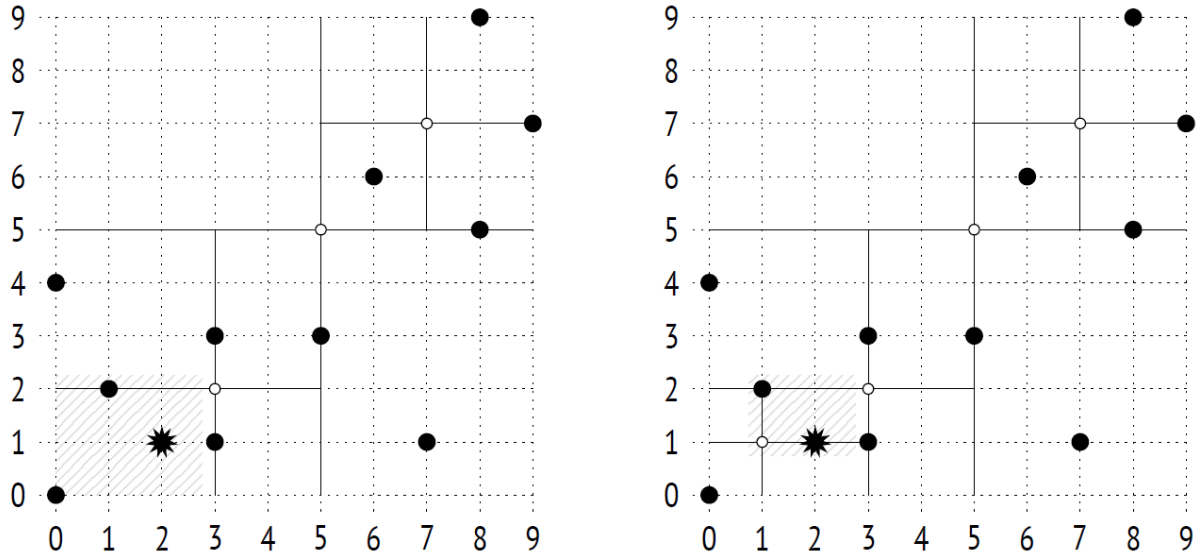
[420] backend/access/spgist/spgdoinsert.c, spgdoinsert function

[421] backend/access/spgist/spgquadtreeproc.c, spg_quad_choose function



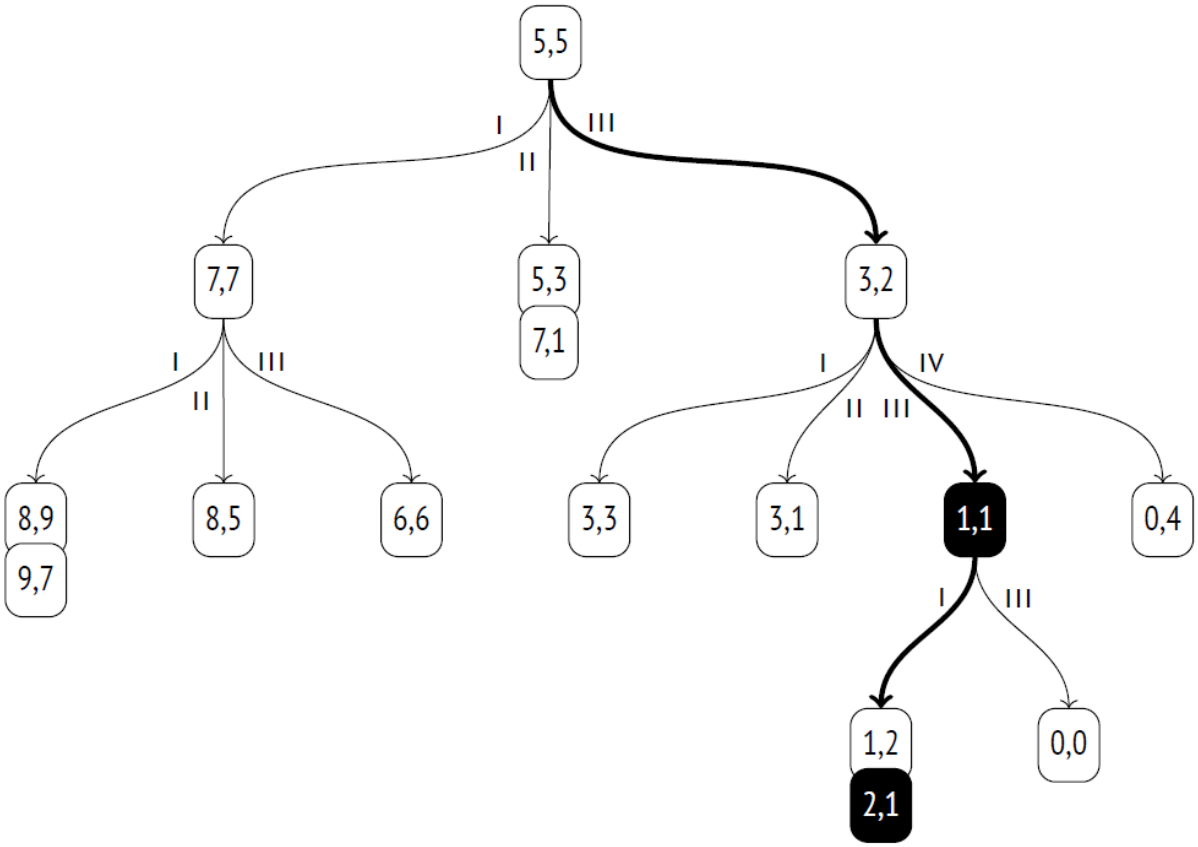
만약 선택된 사분면에 있는 리프 노드의 목록이 삽입 후에 너무 커져서 한 페이지에 들어맞지 않게 되면, 페이지는 분할됩니다. picksplit 함수[422]는 모든 점의 좌표 평균 값을 계산하여 새로운 중심점을 결정함으로써, 자식 노드들을 새로운 사분면에 보다 균등하게 분배합니다.

다음 그림은 점 (2,1)의 삽입으로 인한 페이지 오버플로우를 보여줍니다.



새로운 중심점(1,1)을 가진 내부 노드가 트리에 추가되며, 점(0,0), (1,2), (2,1)이 새로운 사분면에 재분배됩니다.

[422] backend/access/spgist/spgquadtreeproc.c, spg_quad_picksplit function



속성

엑세스 방법의 속성. `spgist` 메서드는 다음과 같은 속성을 반환합니다:

```
=> SELECT a.amname, p.name, pg_indexam_has_property(a.oid, p.name)
FROM pg_am a, unnest(array[
'can_order', 'can_unique', 'can_multi_col',
'can_exclude', 'can_include'
]) p(name)
WHERE a.amname = 'spgist';
 amname |          name | pg_indexam_has_property
--------+---------------+-------------------------
 spgist |     can_order | f
 spgist |    can_unique | f
 spgist | can_multi_col | f
 spgist |   can_exclude | t
 spgist |   can_include | t
(5 rows)
```

정렬 및 고유성 속성에 대한 지원이 제공되지 않습니다. 다중 열 인덱스도 지원되지 않습니다.

배타적 제약 조건은 `GiST`에서와 마찬가지로 지원됩니다.

`SP-GiST` 인덱스는 추가적인 `INCLUDE` 열을 포함하여 생성될 수 있습니다.

인덱스 수준의 속성. GiST와 달리, SP-GiST 인덱스는 클러스터링을 지원하지 않습니다:

```
=> SELECT p.name, pg_index_has_property('airports_quad_idx', p.name)
FROM unnest(array[
'clusterable', 'index_scan', 'bitmap_scan', 'backward_scan'
]) p(name);
          name | pg_index_has_property
---------------+-----------------------
   clusterable | f
    index_scan | t
   bitmap_scan | t
 backward_scan | f
(4 rows)
```

`TID`를 하나씩 또는 비트맵으로 얻는 두 가지 방법 모두 지원됩니다. `SP-GiST`에는 적합하지 않기 때문에 역방향 스캔은 사용할 수 없습니다.

칼럼 수준의 속성. 대부분의 경우, 칼럼 수준의 속성은 동일합니다:

```
=> SELECT p.name,
pg_index_column_has_property('airports_quad_idx', 1, p.name)
FROM unnest(array[
'orderable', 'search_array', 'search_nulls'
]) p(name);
         name | pg_index_column_has_property
--------------+------------------------------
    orderable | f
 search_array | f
 search_nulls | t
(3 rows)
```

정렬은 지원되지 않으므로 관련된 모든 속성들은 의미가 없으며 비활성화됩니다.

지금까지 NULL 값에 대해 언급하지 않았지만, 인덱스 속성에서 볼 수 있듯이 이들은 지원됩니다. GiST와는 다르게, SP-GiST 인덱스는 NULL 값들을 메인 트리에 저장하지 않습니다. 대신, 별도의 트리가 생성되며, 그 뿌리는 두 번째 인덱스 페이지에 위치합니다. 따라서, 첫 세 페이지는 항상 같은 의미를 가집니다: 메타 페이지, 메인 트리의 뿌리, 그리고 NULL 값들을 위한 트리의 뿌리.

어떤 컬럼 수준의 속성들은 특정 연산자 클래스에 따라 달라질 수 있습니다.

```
=> SELECT p.name,
pg_index_column_has_property('airports_quad_idx', 1, p.name)
FROM unnest(array[
'returnable', 'distance_orderable'
```

```
]) p(name);
               name ¦ pg_index_column_has_property
--------------------+-------------------------------
         returnable ¦ t
 distance_orderable ¦ t
(2 rows)
```

이 장의 다른 예제들처럼, 이 인덱스는 인덱스-오직 스캔에 사용될 수 있습니다.

하지만 일반적으로, 연산자 클래스가 리프 페이지에 전체 값을 저장하지 않을 수도 있으며, 대신 테이블을 통해 다시 확인할 수 있습니다. 이는 예를 들어, 잠재적으로 큰 지오메트리 값에 대해 SP-GiST 인덱스를 PostGIS에서 사용할 수 있게 합니다.

가장 가까운 이웃 검색이 지원됩니다. 우리는 연산자 클래스에서 순서 연산자 <->를 보았습니다.

27.3 점을 위한 K-차원 트리

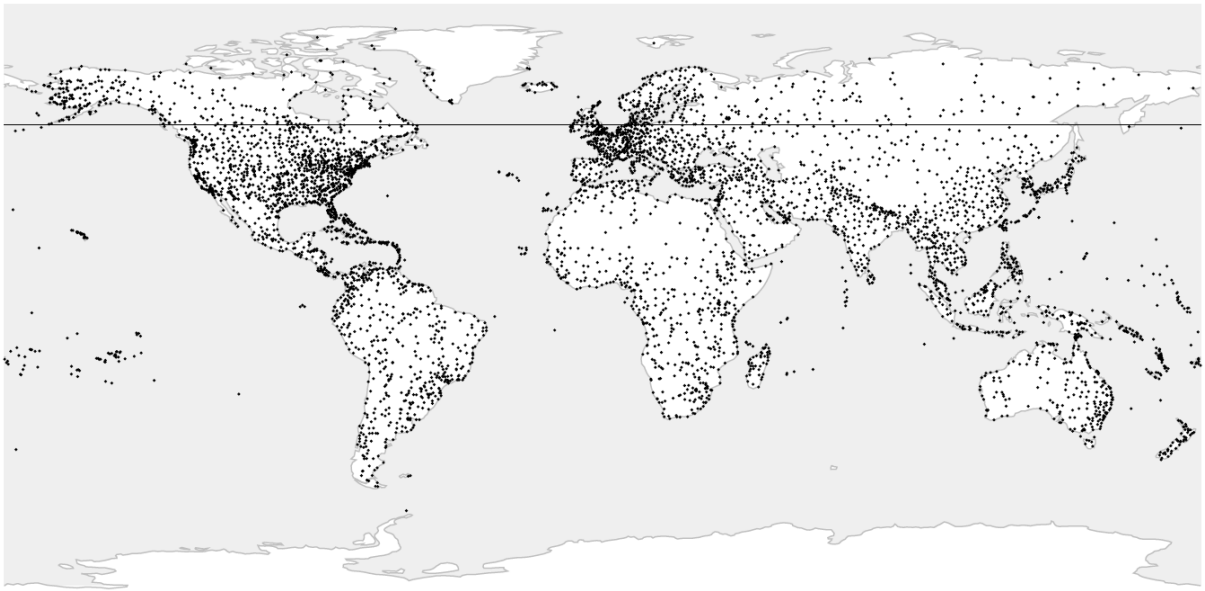
평면 위의 점들은 또 다른 방식의 분할을 사용하여 인덱싱할 수도 있습니다: 우리는 네 개의 하위 영역 대신에 두 개의 하위 영역으로 평면을 분할할 수 있습니다. 이러한 분할은 kd_point_ops[423] 연산자 클래스에 의해 구현됩니다:

```
=> CREATE INDEX airports_kd_idx ON airports_big
USING spgist(coordinates kd_point_ops);
```

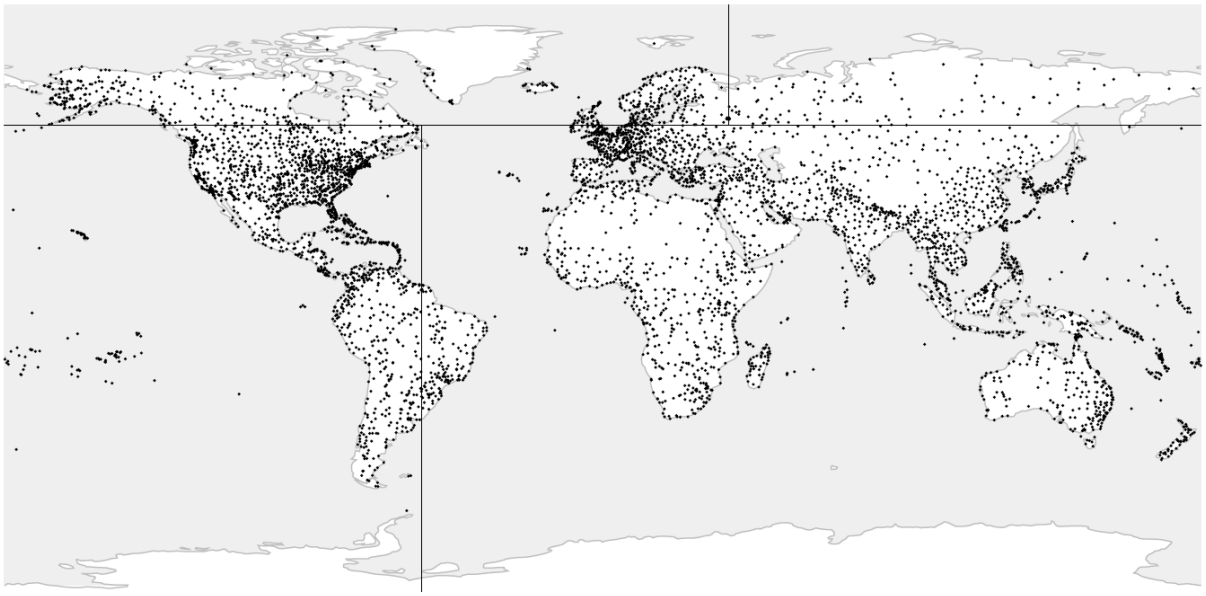
이 연산자 클래스에서는 인덱스된 값, 접두사, 라벨이 서로 다른 데이터 유형을 가질 수 있습니다. 이 경우 값은 점으로 표현되고, 접두사는 실수로 표현되며, 라벨은 제공되지 않습니다(quad_point_ops에서와 같이).

Y축상의 어떤 좌표를 선택합시다(예를 들어 공항의 위도를 정의합니다). 이 좌표는 평면을 상부와 하부의 두 부분으로 나눕니다.

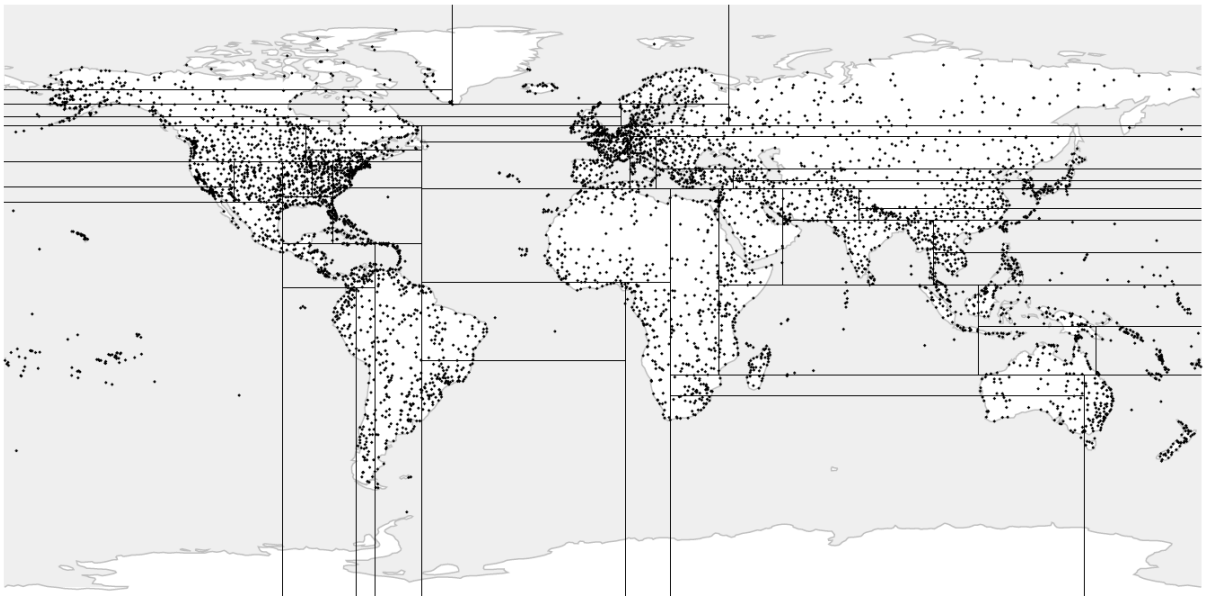
[423] backend/access/spgist/spgkdtreeproc.c



이 서브 리전들 각각에 대해, X축(경도) 상의 좌표를 선택하여 그것들을 왼쪽과 오른쪽 두 서브 리전으로 나눕니다:



각각의 결과로 나온 하위 영역을 수평과 수직 분할을 번갈아 가며 계속해서 나누어, 각 부분의 점들이 단일 인덱스 페이지에 맞을 때까지 분할하겠습니다.



이렇게 구축된 트리의 모든 내부 리프 노드는 단 두 개의 자식 노드만을 가지게 됩니다. 이 방법은 임의의 차원의 공간에 쉽게 일반화될 수 있으므로, 이러한 트리들을 종종 k-차원 트리(k-D 트리)라고 합니다.

27.4 문자열의 기수Radix 트리

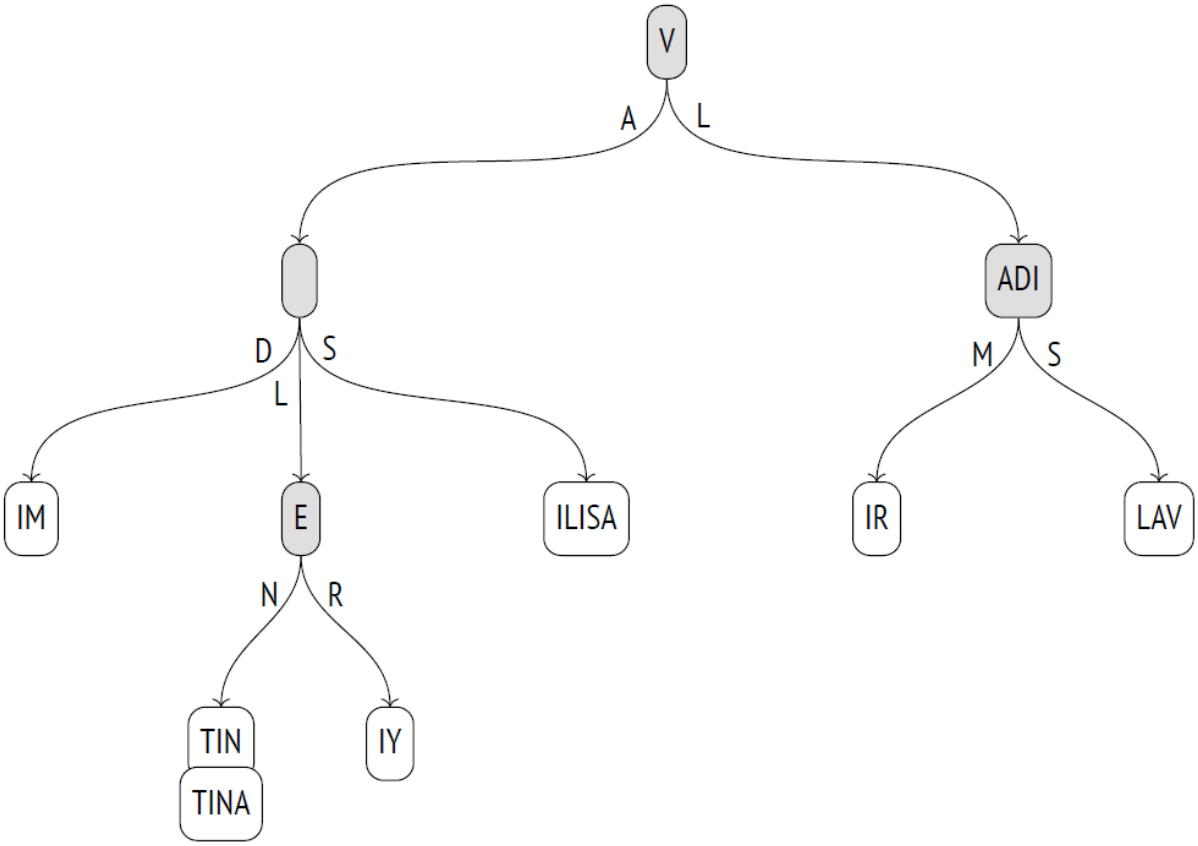
SP-GiST의 text_ops 연산자 클래스는 문자열[424]을 위한 레디스 트리를 구현합니다. 여기서 내부 노드의 접두사는 자식 노드의 모든 문자열에 공통적인 실제 접두사입니다.

자식 노드로의 포인터는 접두사를 따르는 값의 첫 번째 바이트로 표시됩니다.

> 명확성을 위해 저는 접두사를 나타내기 위해 단일 문자를 사용하지만, 이것은 8 바이트 인코딩에만 해당됩니다. 일반적으로, 이 연산자 클래스는 문자열을 바이트의 순서로 처리합니다. 또한, 접두사는 특별한 의미를 가진 여러 다른 값을 취할 수 있으므로 실제로는 접두사마다 2 바이트가 할당됩니다.

자식 노드는 접두사와 라벨을 따르는 값의 일부를 저장합니다. 잎 노드는 접미사만을 보관합니다.
다음은 여러 이름을 기반으로 구축된 기수 트리의 예시입니다:

[424] backend/access/spgist/spgtextproc.c



리프 페이지의 인덱스 키의 전체 값을 재구성하려면, 루트 노드에서 시작하여 모든 접두사와 레이블을 연결할 수 있습니다.

연산자 클래스

text_ops 연산자 클래스는 텍스트 문자열을 포함한 서수 데이터 유형에 일반적으로 사용되는 비교 연산자를 지원합니다.

```
=> SELECT oprname, oprcode::regproc, amopstrategy
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
JOIN pg_amop amop ON amopfamily = opcfamily
JOIN pg_operator opr ON opr.oid = amopopr
WHERE amname = 'spgist'
AND opcname = 'text_ops'
ORDER BY amopstrategy;
 oprname ┆         oprcode ┆ amopstrategy
---------+-----------------+--------------
     ~<~ ┆ text_pattern_lt ┆ 1
    ~<=~ ┆ text_pattern_le ┆ 2
       = ┆          texteq ┆ 3
    ~>=~ ┆ text_pattern_ge ┆ 4
     ~>~ ┆ text_pattern_gt ┆ 5
       < ┆         text_lt ┆ 11
```

```
   <= ¦         text_le ¦ 12
   >= ¦         text_ge ¦ 14
    > ¦         text_gt ¦ 15
   ^@ ¦     starts_with ¦ 28
(10 rows)
```

일반 연산자는 문자를 처리하는 반면, 물결표(~)가 있는 연산자는 바이트를 처리합니다. 이들은 정렬(collation)을 고려하지 않습니다(이는 B-tree용 text_pattern_ops 연산자 클래스와 같습니다), 그래서 LIKE 조건으로 검색을 가속화하는데 사용될 수 있습니다.

```
=> CREATE INDEX tickets_spgist_idx ON tickets
USING spgist(passenger_name);

=> EXPLAIN (costs off) SELECT *
FROM tickets
WHERE passenger_name LIKE 'IVAN%';
                      QUERY PLAN
------------------------------------------------------------
  Bitmap Heap Scan on tickets
      Filter: (passenger_name ~~ 'IVAN%'::text)
      -> Bitmap Index Scan on tickets_spgist_idx
              Index Cond: ((passenger_name ~>=~ 'IVAN'::text) AND
              (passenger_name ~<~ 'IVAO'::text))
(5 rows)
```

> "C" 이외의 collation 을 사용하여 일반 연산자 >= 및 <를 함께 사용하면, 인덱스가 바이트를 다루기 때문에 사실상 쓸모가 없어집니다.

이런 종류의 접두사 검색의 경우, 연산자 클래스는 더 적합한 ^@ 연산자를 제공합니다.

```
=> EXPLAIN (costs off) SELECT *
FROM tickets
WHERE passenger_name ^@ 'IVAN';
               QUERY PLAN
-------------------------------------------------
  Bitmap Heap Scan on tickets
      Recheck Cond: (passenger_name ^@ 'IVAN'::text)
      -> Bitmap Index Scan on tickets_spgist_idx
              Index Cond: (passenger_name ^@ 'IVAN'::text)
(4 rows)
```

기수 트리 표현은 전체 값을 저장하지 않고 필요에 따라 트리를 탐색하면서 값을 재구성하기 때문에, 때때로 B-트리보다 훨씬 더 간결할 수 있습니다.

검색

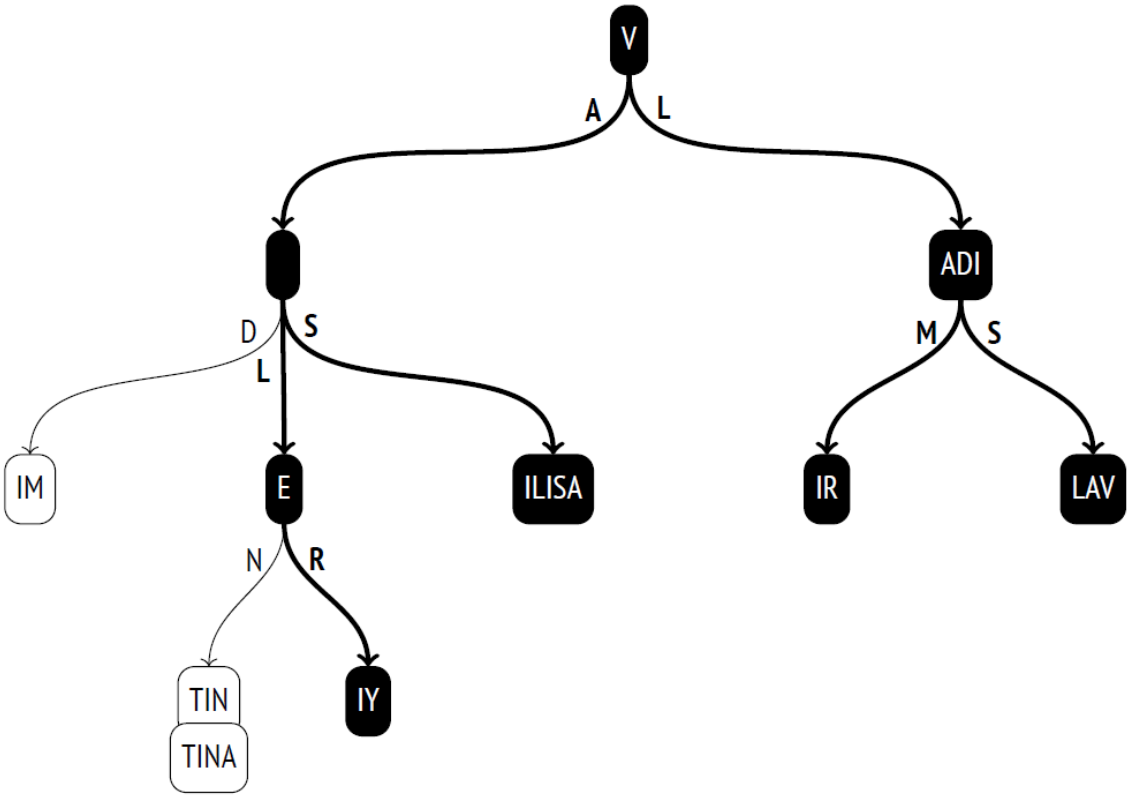
다음 쿼리를 names 테이블에서 실행해 봅시다:

```
SELECT *
FROM names
WHERE name ~>=~ 'VALERIY'
AND name ~<~ 'VLADISLAV';
```

먼저, 내부 일관성 함수[425]가 루트에 호출되어 내려갈 자식 노드들을 결정한다. 이 함수는 접두사 V와 레이블 A, L을 연결한다. 받은 값 VA는 쿼리 조건에 들어가며, 문자열 리터럴은 거기에서 잘리므로, 검사되는 값의 길이를 초과하지 않는다: ☒VA ~>=~ 'VA' 그리고 VA ~<~ 'VL'. 조건이 만족되므로, 레이블 A를 가진 자식 노드를 확인해야 한다. VL 값도 같은 방식으로 확인된다. 이것도 일치하므로, 레이블 L을 가진 노드도 확인해야 한다.

이제 VA 값을 가진 노드를 살펴보자. 그 접두사는 비어 있으므로, 세 자식 노드에 대해 내부 일관성 함수는 이전 단계에서 받은 VA와 레이블을 연결하여 VAD, VAL, VAS 값을 재구성한다. 조건 VAD ~>=~ 'VAL' 그리고 VAD ~<~ 'VER'는 참이 아니지만, 다른 두 값은 적합하다.

이런 식으로 트리를 순회하면서 알고리즘은 일치하지 않는 가지를 필터링하고 리프 노드에 도달한다. 리프 일관성 함수[426]는 트리 순회 동안 재구성된 값이 쿼리 조건을 만족하는지 확인한다. 일치하는 값은 인덱스 스캔의 결과로 반환된다.



[425] backend/access/spgist/spgtextproc.c, spg_text_inner_consistent function

[426] backend/access/spgist/spgtextproc.c, spg_text_leaf_consistent function

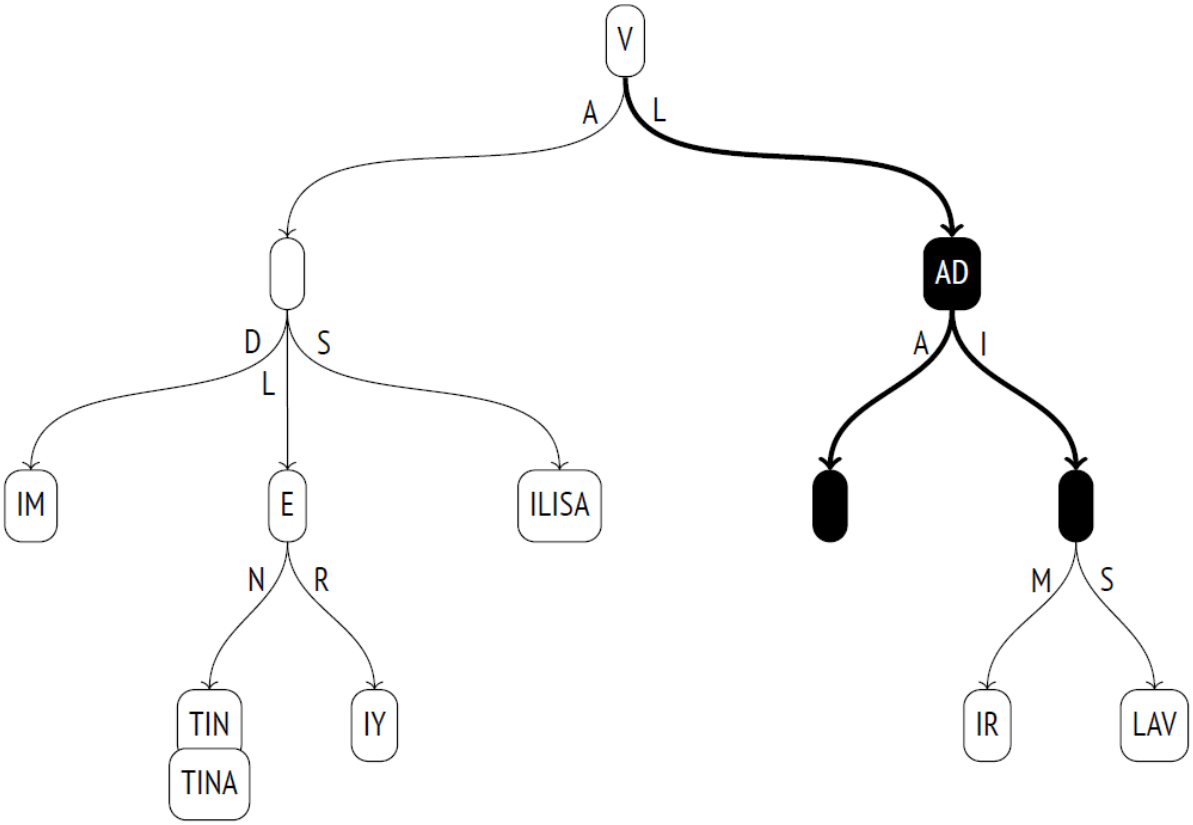
쿼리에서 사용되는 크다와 작다 연산자는 B-트리에 흔하지만, SP-GiST에 의한 범위 검색은 훨씬 덜 효율적임을 유의하세요. B-트리에서는 범위의 단일 경계값으로 내려간 다음 리프 페이지 목록을 스캔하기만 하면 충분합니다.

삽입

연산자 클래스의 선택 함수는 포인트에 대해 새로운 값을 항상 기존의 하위 영역(사분면이나 반쪽 중 하나) 중 하나로 직접 안내할 수 있습니다. 그러나 이것은 레디스 트리에는 해당되지 않습니다: 새로운 값이 기존의 접두사와 일치하지 않을 수 있으며, 이 경우 내부 노드는 분할되어야 합니다.

이미 구축된 트리에 이름 VLADA를 추가해 봅시다.

선택 함수[427]는 루트에서 다음 노드(V + L)로 내려갈 수 있지만, 값의 나머지 부분 ADA는 ADI 접두사와 일치하지 않습니다. 노드는 두 개로 분할되어야 하며, 결과 노드 중 하나는 접두사의 공통 부분(AD)을 포함할 것이고, 나머지 접두사는 한 단계 아래로 이동됩니다:



속성

나는 이미 위에서 접근 방식과 인덱스 수준의 특성에 대해 설명했으며, 이들은 모든 클래스에 공통적입니다. 대부분의 열 수준 속성도 동일하게 유지됩니다.

[427] backend/access/spgist/spgtextproc.c, spg_text_choose function

```
=> SELECT p.name,
pg_index_column_has_property('tickets_spgist_idx', 1, p.name)
FROM unnest(array[
'returnable', 'distance_orderable'
]) p(name);
               name ¦ pg_index_column_has_property
--------------------+-------------------------------
         returnable ¦ t
 distance_orderable ¦ f
(2 rows)
```

비록 인덱스된 값들이 트리에 명시적으로 저장되지 않던, 루트에서 리프 노드까지 트리를 탐색하는 동안 값들이 재구성되기 때문에 인덱스-단일 스캔이 지원됩니다.

거리 연산자의 경우, 문자열에 대해 정의되어 있지 않기 때문에, 이 연산자 클래스에서는 최근접 이웃 검색을 제공하지 않습니다.

> 문자열에 대해 거리 개념이 구현될 수 없다는 것을 의미하지 않습니다. 예를 들어, pg_trgm 확장은 트리그램을 기반으로 한 거리 연산자를 추가합니다: 두 문자열에서 발견된 공통 트리그램이 적을수록, 그들이 서로 멀리 위치하고 있다고 가정합니다. 그리고 레벤슈타인 거리는 한 문자열을 다른 문자열로 변환하기 위해 필요한 단일 문자 편집의 최소 수로 정의됩니다. 이러한 거리를 계산하는 함수는 fuzzystrmatch 확장에서 제공됩니다. 하지만 어떤 확장도 SP-GiST 지원과 함께 연산자 클래스를 제공하지 않습니다.

27.5 다른 데이터 유형

우리가 위에서 논의한 점과 텍스트 문자열에 대한 인덱싱에만 SP-GiST 연산자 클래스가 제한되지 않습니다.

기하학적 유형. box_ops[428] 연산자 클래스는 사각형에 대한 쿼드트리를 구현합니다. 사각형은 네 차원 공간에서 점으로 표현되므로, 영역은 열여섯 파티션으로 나뉩니다.

poly_ops 클래스는 다각형을 인덱싱하는 데 사용할 수 있습니다. 이는 퍼지 연산자 클래스로, 사실상 box_ops와 마찬가지로 다각형 대신 경계 상자를 사용한 다음 테이블에 의해 결과를 재확인합니다.

GiST와 SP-GiST 중에서 선택하는 것은 인덱싱되는 데이터의 성격에 크게 달려 있습니다. 예를 들어, PostGIS 문서에서는 큰 중첩이 있는 객체(일명 "스파게티 데이터")[429]에 대해 SP-GiST를 권장합니다.

범위 유형. 범위에 대한 쿼드트리는 range_ops 연산자 클래스를 제공합니다.[430] 간격은 이차원 점으로 정의됩니다: X축은 하한을 나타내고 Y축은 상한을 나타냅니다.

---

[428] backend/utils/adt/geo_spgist.c

[429] postgis.net/docs/using_postgis_dbmanagement.html#spgist_indexes

[430] backend/utils/adt/rangetypes_spgist.c

네트워크 주소 유형. inet 데이터 유형의 경우, inet_ops[431] 연산자 클래스는 기수 트리를 구현합니다.

[431] backend/utils/adt/network_spgist.c

28 장. GIN

28.1 개요

그것의 저자들에 따르면, GIN은 알코올 음료가 아닌, 강력하고 두려움 없는 정신을 의미합니다.[432] 하지만 이것에는 공식적인 해석도 있습니다: 이 약어는 Generalized Inverted Index로 확장됩니다.

GIN 접근 방식은 별도의 요소들로 구성된 비원자적 값들을 나타내는 데이터 유형을 위해 설계되었습니다(예를 들어, 문서는 전체 텍스트 검색의 맥락에서 어휘들로 구성됩니다). GiST가 값들을 전체로 색인하는 것과는 달리, GIN은 오직 그 요소들만 색인합니다; 각 요소는 그것을 포함하는 모든 값들에 매핑됩니다.

이 방법은 모든 중요한 용어들을 포함하고 이 용어들이 언급된 모든 페이지들을 나열하는 책의 색인과 비교할 수 있습니다. 사용하기 편리하려면, 그것은 알파벳 순서로 컴파일되어야 합니다, 그렇지 않으면 빠르게 탐색하는 것이 불가능할 것입니다. 비슷한 방식으로, GIN은 복합 값들의 모든 요소들이 정렬될 수 있다는 사실에 의존합니다; 그것의 주요 데이터 구조는 B-트리입니다.

GIN 트리의 구현은 일반 B-트리보다 덜 복잡합니다: 작은 요소 집합을 여러 번 반복하여 포함하도록 설계되었습니다.

이 가정은 두 가지 중요한 결론으로 이어집니다:

- 요소는 인덱스에 단 한 번만 저장되어야 합니다.

  각 요소는 TID의 목록에 매핑되며, 이를 포스팅 리스트라고 합니다. 이 목록이 비교적 짧으면 요소와 함께 저장되고, 더 긴 목록은 별도의 포스팅 트리(실제로는 B-트리)로 이동됩니다. 요소 트리와 마찬가지로 포스팅 리스트는 정렬되어 있으며, 사용자 관점에서는 크게 중요하지 않지만 데이터 접근 속도를 높이고 인덱스 크기를 줄이는 데 도움이 됩니다.

- 트리에서 요소를 제거할 필요가 없습니다.

  특정 요소에 대한 TID 목록이 비어 있더라도, 동일한 요소가 다른 값의 일부로 다시 나타날 가능성이 높습니다.

따라서 인덱스는 요소의 트리이며, 그 잎 항목은 평면 목록이나 TID의 트리에 연결됩니다.

GiST와 SP-GiST 접근 방식처럼, GIN도 연산자 클래스의 간단한 인터페이스를 통해 다양한 데이터 유형을 인덱스할 수 있습니다. 이러한 클래스의 연산자는 인덱스된 복합 값이 특정 요소 집합과 일치하는지 여부를 주로 확인합니다(예를 들어, @@ 연산자가 문서가 전체 텍스트 검색 쿼리를 만족하는지 확인하는 것처럼).

특정 데이터 유형을 인덱스하기 위해, GIN 방식은 복합 값을 요소로 분할하고, 이 요소들을 정렬하며, 찾은 값이 쿼리를 만족하는지 확인할 수 있어야 합니다. 이러한 연산은 연산자 클래스의 지원 함수에 의해 구현됩니다.

---

[432] postgresql.org/docs/14/gin.html
backend/access/gin/README

28.2 전체 텍스트 검색을 위한 색인

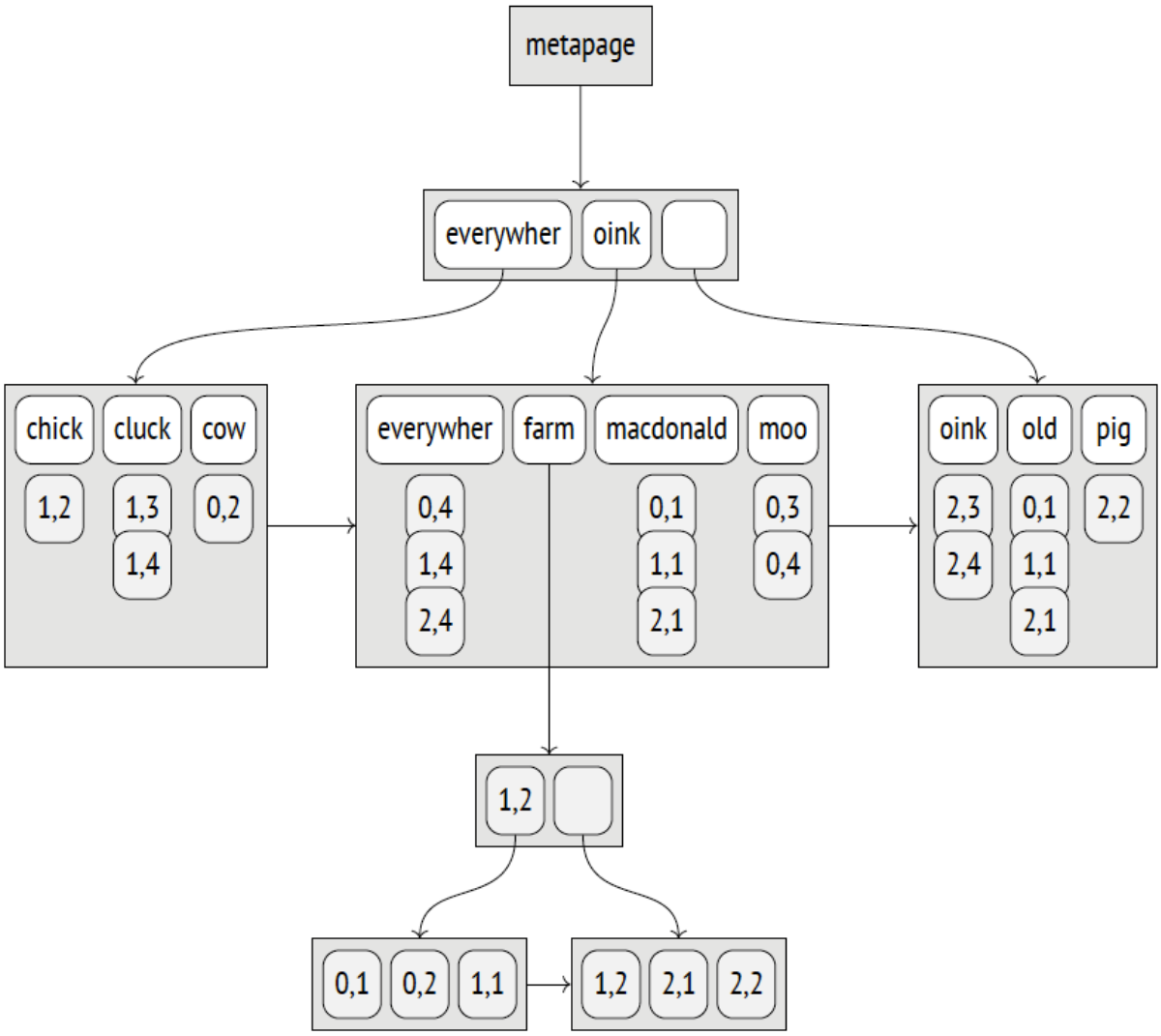
GIN은 주로 전문 검색 속도를 높이는 데 적용되므로, GiST 인덱싱을 설명하기 위해 사용된 예제를 계속해서 사용해 보겠습니다. 이 경우 복합 값은 문서이며, 이러한 값의 요소는 어휘입니다.

"Old MacDonald" 테이블에 GIN 인덱스를 구축해 봅시다:

```
=> CREATE INDEX ts_gin_idx ON ts USING gin(doc_tsv);
```

이 인덱스의 가능한 구조는 아래에 나타나 있습니다. 이전 그림과는 달리, 여기서는 실제 TID 값(회색 배경으로 표시됨)을 제공합니다. 왜냐하면 이 값들은 알고리즘을 이해하는 데 매우 중요하기 때문입니다. 이 값들은 힙 튜플이 다음 ID를 가지고 있음을 제안합니다:

```
=> SELECT ctid, * FROM ts;
ctid ¦ doc ¦ doc_tsv
-------+----------------------------------+-------------------------------
 (0,1) ¦ Old MacDonald had a farm           ¦ 'farm':5 'macdonald':2 'old':1
 (0,2) ¦ And on his farm he had some cows   ¦ 'cow':8 'farm':4
 (0,3) ¦ Here a moo, there a moo            ¦ 'moo':3,6
 (0,4) ¦ Everywhere a moo moo               ¦ 'everywher':1 'moo':3,4
 (1,1) ¦ Old MacDonald had a farm           ¦ 'farm':5 'macdonald':2 'old':1
 (1,2) ¦ And on his farm he had some chicks ¦ 'chick':8 'farm':4
 (1,3) ¦ Here a cluck, there a cluck        ¦ 'cluck':3,6
 (1,4) ¦ Everywhere a cluck cluck           ¦ 'cluck':3,4 'everywher':1
 (2,1) ¦ Old MacDonald had a farm           ¦ 'farm':5 'macdonald':2 'old':1
 (2,2) ¦ And on his farm he had some pigs   ¦ 'farm':4 'pig':8
 (2,3) ¦ Here an oink, there an oink        ¦ 'oink':3,6
 (2,4) ¦ Everywhere an oink oink            ¦ 'everywher':1 'oink':3,4
(12 rows)
```



여기에서 일반 B-트리 인덱스와 몇 가지 차이점을 확인할 수 있습니다. 내부 B-트리 노드의 가장 왼쪽 키는 실제로 중복되기 때문에 비어 있습니다; GIN 인덱스에서는 이러한 키들이 전혀 저장되지 않습니다. 이 때문에 자식 노드에 대한 참조도 이동됩니다. 고유 키는 두 인덱스에서 모두 사용되지만, GIN에서는 그것이 정당한 가장 오른쪽 위치를 차지합니다. 같은 레벨의 노드들은 B-트리에서 양방향 리스트로 연결되어 있지만, GIN은 트리가 항상 한 방향으로만 탐색되기 때문에 단방향 리스트를 사용합니다.

이 이론적 예에서, 모든 포스팅 리스트는 정규 페이지에 맞습니다, "farm" 어휘를 제외하고요. 이 어휘는 무려 여섯 개의 문서에서 발생하여, 그 ID들이 별도의 포스팅 트리로 옮겨졌습니다.

페이지 구성

GIN 페이지 레이아웃은 B-트리의 구조와 매우 유사합니다. pageinspect 확장 기능을 사용하여 인덱스 내부를 살펴볼 수 있습니다. pgsql-hackers 메일링 리스트의 이메일을 저장하는 테이블에 GIN 인덱스를 생성해 봅시다:

```
=> CREATE INDEX mail_gin_idx ON mail_messages USING gin(tsv);
```

제로 페이지(메타페이지)는 요소의 수와 다른 유형의 페이지 수와 같은 기본 통계를 포함합니다:

```
=> SELECT *
FROM gin_metapage_info(get_raw_page('mail_gin_idx',0)) \gx
-[ RECORD 1 ]----+-----------
    pending_head ¦ 4294967295
    pending_tail ¦ 4294967295
  tail_free_size ¦ 0
 n_pending_pages ¦ 0
n_pending_tuples ¦ 0
   n_total_pages ¦ 22957
   n_entry_pages ¦ 13522
    n_data_pages ¦ 9434
       n_entries ¦ 999109
         version ¦ 2
```

GIN은 인덱스 페이지의 특수한 공간을 사용합니다. 예를 들어, 이 공간은 페이지 유형을 정의하는 비트들을 저장합니다:

```
=> SELECT flags, count(*)
FROM generate_series(0,22956) AS p, -- n_total_pages
gin_page_opaque_info(get_raw_page('mail_gin_idx',p))
GROUP BY flags
ORDER BY 2;
                  flags ¦ count
------------------------+-------
                {meta} ¦ 1
                    {} ¦ 137
                {data} ¦ 1525
 {data,leaf,compressed} ¦ 7909
                {leaf} ¦ 13385
(5 rows)
```

메타 속성이 있는 페이지는 당연히 메타페이지입니다. 데이터 속성이 있는 페이지는 포스팅 목록에 속하며, 이 속성이 없는 페이지는 요소 트리와 관련이 있습니다. 리프 페이지는 리프 속성을 가집니다.

다음 예에서, 다른 pageinspect 함수는 트리의 리프 페이지에 저장된 TID에 대한 정보를 반환합니다. 이러한 트리의 각 항목은 단일 TID가 아니라 사실상 TID의 작은 목록입니다.

```
=> SELECT left(tids::text,60)¦¦'...' tids
FROM gin_leafpage_items(get_raw_page('mail_gin_idx',24));
tids
------------------------------------------------------------
{"(4771,4)","(4775,2)","(4775,5)","(4777,4)","(4779,1)","(47...
{"(5004,2)","(5011,2)","(5013,1)","(5013,2)","(5013,3)","(50...
{"(5435,6)","(5438,3)","(5439,3)","(5439,4)","(5439,5)","(54...
...
```

```
{"(9789,4)","(9791,6)","(9792,4)","(9794,4)","(9794,5)","(97...
{"(9937,4)","(9937,6)","(9938,4)","(9939,1)","(9939,5)","(99...
{"(10116,5)","(10118,1)","(10118,4)","(10119,2)","(10121,2)"...
(27 rows)
```

포스팅 리스트는 정렬되어 있기 때문에 압축될 수 있습니다(따라서 같은 이름의 속성이 있습니다). 여섯 바이트의 TID 대신, 이전 값과의 차이를 가변 바이트 수로 저장합니다.[433] 이 차이가 작을수록 데이터가 차지하는 공간이 줄어듭니다.

연산자 클래스

GIN 연산자 클래스를 위한 지원 함수 목록은 다음과 같습니다:[434]

```
=> SELECT amprocnum, amproc::regproc
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
JOIN pg_amproc amop ON amprocfamily = opcfamily
WHERE amname = 'gin'
AND opcname = 'tsvector_ops'
ORDER BY amprocnum;
 amprocnum ¦ amproc
-----------+-----------------------------------
         1 ¦ gin_cmp_tslexeme
         2 ¦ pg_catalog.gin_extract_tsvector
         3 ¦ pg_catalog.gin_extract_tsquery
         4 ¦ pg_catalog.gin_tsquery_consistent
         5 ¦ gin_cmp_prefix
         6 ¦ gin_tsquery_triconsistent
(6 rows)
```

첫 번째 지원 함수는 두 요소(이 경우에는 두 개의 어휘)를 비교합니다. 어휘가 B-tree에서 지원하는 일반 SQL 유형으로 표현된 경우, GIN은 B-tree 연산자 클래스에서 정의한 비교 연산자를 자동으로 사용합니다.

다섯 번째(선택적인) 함수는 부분 검색에 사용되어 인덱스 요소가 검색 키와 부분적으로 일치하는지 확인합니다. 이 특정 경우에서, 부분 검색은 접두사에 의한 어휘 검색으로 구성됩니다. 예를 들어, "c:*" 쿼리는 "c"로 시작하는 모든 어휘에 해당합니다.

두 번째 함수는 문서에서 어휘를 추출하고, 세 번째 함수는 검색 쿼리에서 어휘를 추출합니다. 문서와 쿼리가 최소한 다른 데이터 유형, 즉 tsvector와 tsquery로 표현되기 때문에 다른 함수의 사용이 정당화됩니다. 게다가, 검색 쿼리를 위한 함수는 검색이 어떻게 수행될지 결정합니다. 쿼리가 문서가 특정 어휘를 포함하도록 요구하는 경우, 검색은 쿼리에 지정된 최소한 하나의 어휘를 포함하는 문서로 제한됩니다. 특정 어휘를 포함하

---

[433] backend/access/gin/ginpostinglist.c

[434] postgresql.org/docs/14/gin-extensibility.html
backend/utils/adt/tsginidx.c

지 않는 문서가 필요한 경우(예를 들어, 특정 어휘를 포함하지 않는 문서가 필요한 경우)와 같은 조건이 없다면, 모든 문서를 스캔해야 하며 이는 물론 훨씬 더 많은 비용이 듭니다.

> 만약 쿼리에 다른 검색 키가 포함되어 있다면, 인덱스는 먼저 이러한 키들로 스캔되고, 그 다음에 이 중간 결과들이 재검사됩니다. 따라서 인덱스를 전체적으로 스캔할 필요가 없습니다.

네 번째와 여섯 번째 기능은 일관성 함수로, 찾아낸 문서가 검색 쿼리를 만족하는지 여부를 결정합니다. 네 번째 함수는 쿼리에 명시된 어떤 어휘가 문서에 나타나는지에 대한 정확한 정보를 입력으로 받습니다. 여섯 번째 함수는 불확실성의 맥락에서 작동하며, 일부 어휘가 문서에 존재하는지 여부가 아직 명확하지 않을 때 호출될 수 있습니다. 연산자 클래스는 두 함수 모두를 구현할 필요가 없으며, 하나만 제공해도 충분하지만, 이 경우 검색 효율이 떨어질 수 있습니다.

tsvector_ops 연산자 클래스는 문서가 검색 쿼리와 일치하는지를 확인하는 단 하나의 연산자, @@를 지원하며[435], 이는 GiST 연산자 클래스에도 포함되어 있습니다.

검색

"everywhere | oink" 쿼리에 대한 검색 알고리즘을 살펴보겠습니다. 여기서 두 어휘는 OR 연산자로 연결됩니다. 먼저, 지원 함수[436]는 tsquery 타입의 검색 문자열에서 "everywher"와 "oink" (검색 키)라는 어휘를 추출합니다.

쿼리가 특정 어휘의 존재를 요구하기 때문에, 쿼리에서 지정한 최소 한 개의 키를 포함하는 문서의 TID는 목록으로 묶입니다. 이를 위해 각 검색 키에 해당하는 TID는 어휘의 트리에서 검색되어 공통 목록에 추가됩니다. 인덱스에 저장된 모든 TID는 정렬되어 있어 여러 정렬된 TID 스트림을 하나로 병합할 수 있습니다.[437]

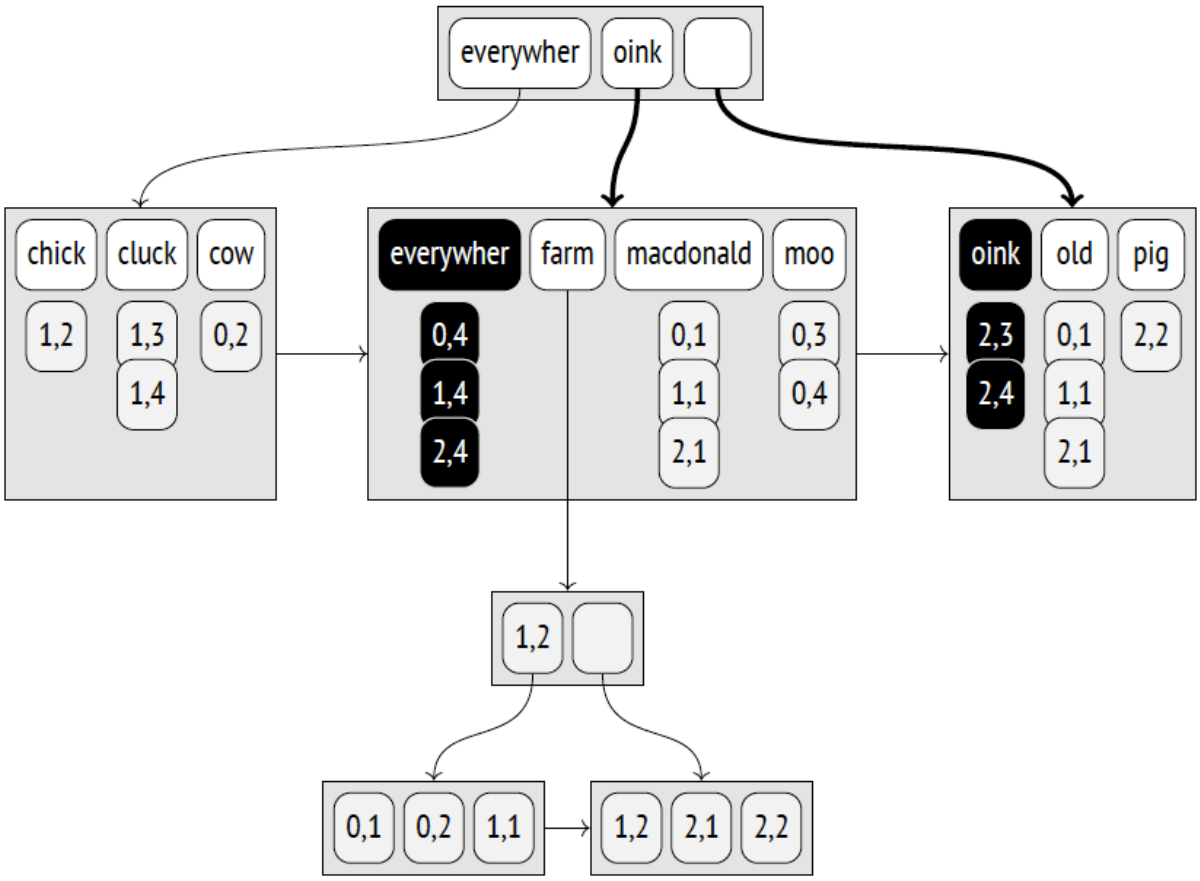
이 시점에서 키가 AND, OR 또는 다른 연산자로 결합되었는지 여부는 아직 중요하지 않습니다. 검색 엔진은 키의 목록을 다루며 검색 쿼리 의미에 대해 아무것도 모릅니다.

---

[435] backend/utils/adt/tsvector_op.c, ts_match_vq function

[436] backend/utils/adt/tsginidx.c, gin_extract_tsquery function

[437] backend/access/gin/ginget.c, keyGetItem function



검색된 각 TID는 일관성 함수에 의해 검사됩니다.[438] 바로 이 함수가 검색 쿼리를 해석하고 쿼리를 만족시키는(또는 적어도 만족시킬 수 있어서 테이블에 의해 재검사가 필요한) TID만을 남깁니다.

이 특별한 경우에서, 일관성 함수는 모든 TID를 남깁니다:

TID	"everywher"	"oink"	consistency function
(0,4)	✓	–	✓
(1,4)	✓	–	✓
(2,3)	–	✓	✓
(2,4)	✓	✓	✓

검색 쿼리는 일반적인 어휘 대신 접두사를 포함할 수 있습니다. 이는 애플리케이션 사용자가 검색 필드에 단어의 첫 글자를 입력할 수 있으며, 즉시 결과를 얻기를 기대하는 경우 유용합니다. 예를 들어, "pig:*" 쿼리는 "pig"로 시작하는 어휘를 포함한 모든 문서와 일치합니다: 여기서 우리는 "pigs"를 얻고, 만약 올드 맥도날드가 그의 농장에서 그들을 키웠다면 "pigeons"도 얻을 수 있습니다.

이러한 부분 검색은 특별한 지원 함수[439]를 사용하여 검색 키와 인덱스된 어휘를 대조합니다; 접두사 일치 외

[438] backend/utils/adt/tsginidx.c, gin_tsquery_triconsistent function

[439] backend/utils/adt/tsginidx.c, gin_cmp_prefix function

에도, 이 함수는 부분 검색을 위한 다른 로직을 구현할 수도 있습니다.

빈번하고 희귀한 어휘

검색된 어휘가 문서에 여러 번 나타난 경우, 생성된 TID 목록은 길어지게 되어 분명히 비효율적입니다. 다행히, 질의에 드문 어휘가 포함되어 있다면, 이러한 상황은 종종 피할 수 있습니다.

"farm & cluck" 질의를 고려해 봅시다. "cluck" 어휘는 두 번 나타나고, "farm" 어휘는 여섯 번 나타납니다. 두 어휘를 동등하게 취급하여 그들에 의한 TID의 전체 목록을 구성하는 대신, 드문 "cluck" 어휘를 필수적으로 간주하고, 더 자주 나타나는 "farm" 어휘를 선택적으로 취급합니다. 왜냐하면, 질의의 의미를 고려할 때, "farm" 어휘를 포함하는 문서가 "cluck" 어휘도 포함할 경우에만 질의를 만족시킬 수 있기 때문입니다.

따라서, 인덱스 스캔은 "cluck"을 포함하는 첫 번째 문서를 결정하며, 그것의 TID는 (1,3)입니다. 그 후, 이 문서가 "farm" 어휘도 포함하는지 알아내야 하지만, (1,3)보다 작은 TID를 가진 모든 문서는 건너뛸 수 있습니다. 자주 나타나는 어휘는 많은 TID에 해당할 가능성이 높기 때문에, 많은 페이지도 건너뛸 수 있습니다. 이 특정 사례에서, "farm" 어휘의 트리에서의 검색은 (1,3)부터 시작합니다.

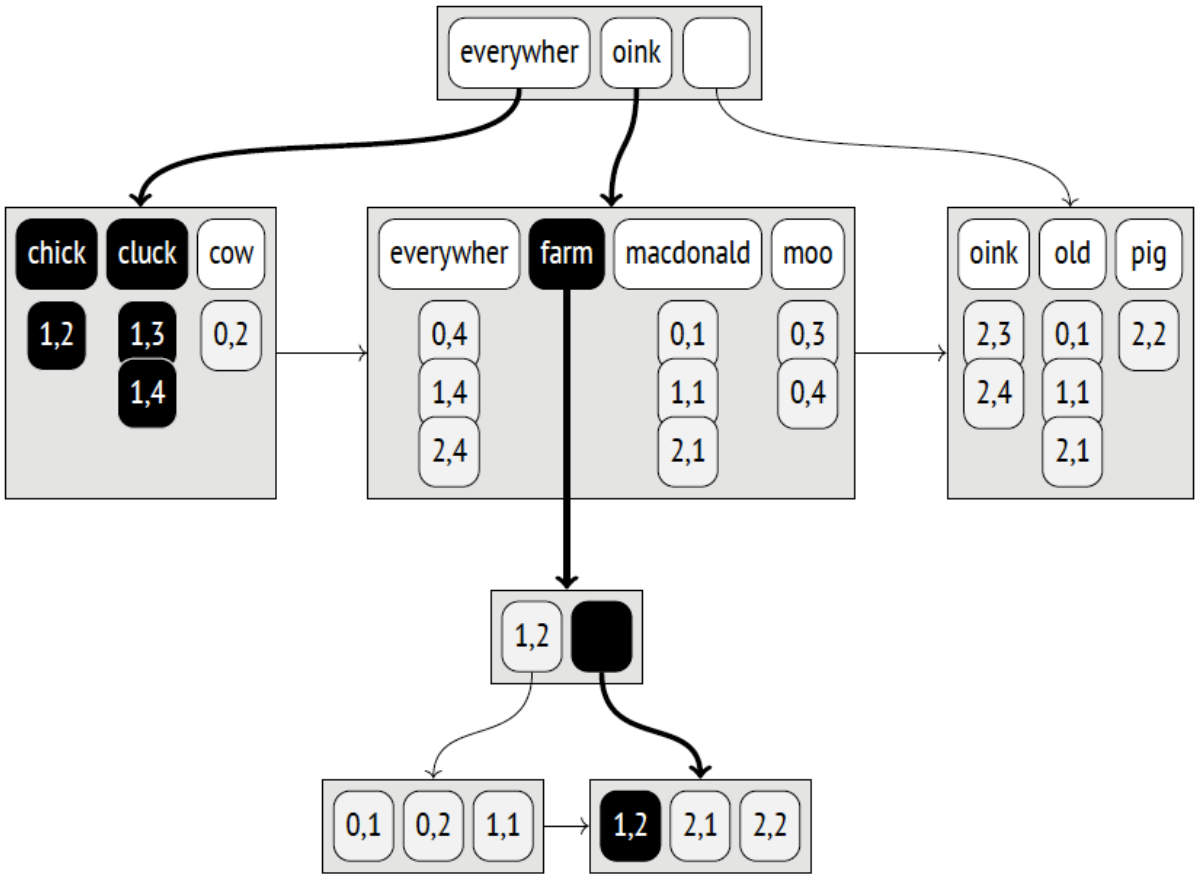
이 절차는 필수 어휘의 후속 값에 대해 반복됩니다.

분명히, 이 최적화는 두 개 이상의 어휘를 포함하는 더 복잡한 검색 시나리오에도 적용될 수 있습니다. 알고리즘은 어휘를 그들의 출현 빈도 순서대로 정렬하고, 하나씩 필수 어휘 목록에 추가하며, 남은 어휘가 문서가 질의를 만족시키는 것을 보장할 수 없을 때 중단합니다.[440]

예를 들어, "farm & (cluck | chick)" 쿼리를 고려해 봅시다. 가장 드문 어휘인 "chick"이 바로 필수 어휘 목록에 추가됩니다. 다른 어휘들이 선택적으로 고려될 수 있는지 확인하기 위해 일관성 함수는 필수 어휘에 대해 false를, 모든 다른 어휘에 대해 true를 취합니다. 함수는 true AND (true OR false) = true를 반환합니다. 이는 남은 어휘들이 "자체 충분하며" 적어도 하나는 필수가 되어야 한다는 것을 의미합니다.

다음으로 드문 어휘인 "cluck"이 목록에 추가되고, 이제 일관성 함수는 true AND (false OR false) = false를 반환합니다. 따라서 "chick"과 "cluck" 어휘가 필수가 되고, "farm"은 선택적으로 남게 됩니다.

[440] backend/access/gin/ginget.c, startScanKey function



게시물 목록의 길이는 필수 어휘가 세 번 나타났기 때문에 세입니다.

TID	"chick"	"cluck"	"farm"	consistency function
(1,2)	✓	–	✓	✓
(1,3)	–	✓	–	–
(1,4)	–	✓	–	–

따라서, 단어 빈도수가 알려져 있다면, 드문 단어부터 시작하여 빈번한 단어의 페이지 범위를 생략하며 단어의 트리를 가장 효율적으로 병합할 수 있습니다. 이는 일관성 함수를 호출해야 하는 횟수를 줄입니다.

이 최적화가 실제로 효과가 있는지 확인하기 위해, pgsql-hackers 아카이브를 쿼리해 볼 필요가 있습니다. 이때, 흔한 단어와 드문 단어 두 가지를 지정해야 합니다.

```
=> SELECT word, ndoc
FROM ts_stat('SELECT tsv FROM mail_messages')
WHERE word IN ('wrote', 'tattoo');
   word ¦ ndoc
--------+--------
  wrote ¦ 231173
 tattoo ¦ 2
```

```
(2 rows)
```

두 단어 모두를 포함하는 문서가 실제로 존재하는 것으로 밝혀졌습니다:

```
=> \timing on
=> SELECT count(*) FROM mail_messages
WHERE tsv @@ to_tsquery('wrote & tattoo');
count
-------
1
(1 row)
Time: 0,631 ms
```

이 쿼리는 단일 단어 "tattoo"를 검색하는 것만큼 거의 빠르게 수행됩니다.

```
=> SELECT count(*) FROM mail_messages
WHERE tsv @@ to_tsquery('tattoo');
count
-------
2
(1 row)
Time: 2,227 ms
```

하지만 우리가 단어 "wrote"만을 찾았다면, 검색에 훨씬 더 오랜 시간이 걸렸을 것입니다.

```
=> SELECT count(*) FROM mail_messages
WHERE tsv @@ to_tsquery('wrote');
count
--------
231173
(1 row)
Time: 343,556 ms
=> \timing off
```

삽입

GIN 인덱스는 중복을 포함할 수 없습니다.[441] 인덱스에 추가될 요소가 이미 존재하는 경우, 해당 요소의 TID는 이미 존재하는 요소의 게시 목록이나 트리에 단순히 추가됩니다. 게시 목록은 인덱스 항목의 일부로 페이지 내에 너무 많은 공간을 차지할 수 없으므로, 할당된 공간을 초과하면 목록은 트리로 변환됩니다.[442]

새로운 요소(또는 새로운 TID)가 트리에 추가될 때, 페이지 오버플로우가 발생할 수 있습니다. 이 경우 페이

[441] backend/access/gin/gininsert.c, ginEntryInsert function

[442] backend/access/gin/gininsert.c, addItemPointersToLeafTuple function

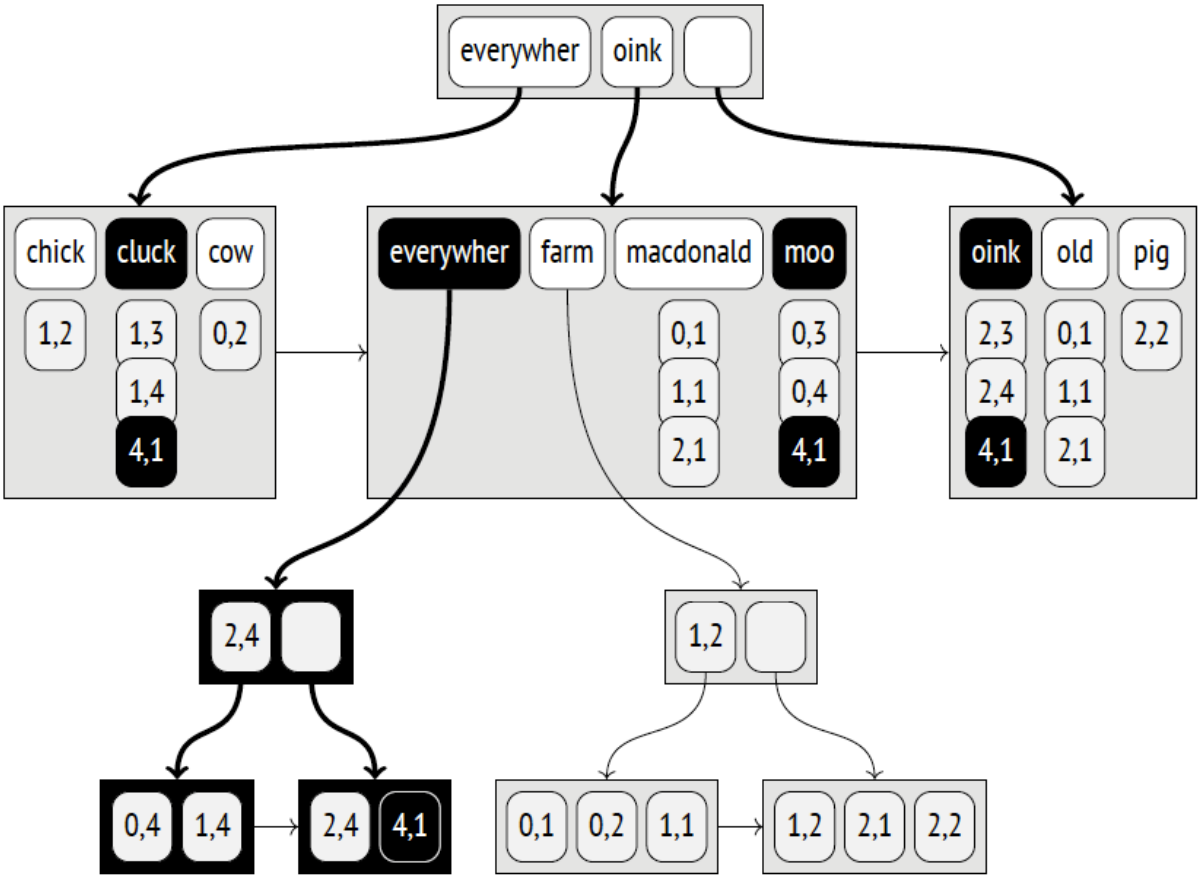
지는 둘로 나뉘어지고 요소들은 그 사이에 재분배됩니다.[443]

그러나 각 문서는 인덱싱해야 할 많은 어휘들을 일반적으로 포함하고 있습니다. 따라서 단 하나의 문서를 생성하거나 수정하더라도 인덱스 트리는 많은 변경을 겪습니다. 이것이 GIN 업데이트가 상대적으로 느린 이유입니다.

아래의 예시는 "Everywhere clucks, moos, and oinks"라는 문장이 TID (4,1)로 테이블에 삽입된 후 트리의 상태를 보여줍니다. "cluck," "moo," 그리고 "oink" 어휘의 게시 목록이 확장되었고, "everywher" 어휘의 목록은 최대 크기를 초과하여 별도의 트리로 분리되었습니다.

만약 인덱스가 한 번에 여러 문서와 관련된 변경 사항을 통합하여 업데이트된다면, 이러한 문서들이 일부 공통 어휘를 포함할 수 있기 때문에 연속적인 변경에 비해 전체 작업량이 줄어들 가능성이 높아집니다.

이 최적화는 fastupdate(기본값: on) 저장소 매개변수에 의해 제어됩니다. 지연된 인덱스 업데이트는 순서가 지정되지 않은 보류 중인 목록에 누적되며, 이 목록은 요소 트리 외부의 별도 목록 페이지에 물리적으로 저장됩니다. 이 목록이 충분히 커지면, 그 내용이 한 번에 인덱스로 전송되고 목록이 비워집니다.[444] 목록의 최대 크기는 gin_pending_list_limit(기본값: 4MB) 매개변수 또는 동일한 이름의 인덱스 저장소 매개변수에 의해 정의됩니다.



[443] backend/access/gin/ginbtree.c, ginInsertValue function

[444] backend/access/gin/ginfast.c, ginInsertCleanup function

기본적으로 이러한 지연된 업데이트는 활성화되어 있지만, 이러한 업데이트가 검색 속도를 늦추게 된다는 점을 염두에 두어야 합니다. 트리 자체뿐만 아니라, 모든 비정렬된 어휘 목록도 검색해야 하기 때문입니다. 또한, 삽입 시간이 덜 예측 가능해지며, 어떤 변경이든 오버플로를 야기할 수 있고 이는 비용이 많이 드는 병합 절차를 초래할 수 있습니다. 이러한 문제는 인덱스가 진공청소하는 동안 병합이 비동기적으로 수행될 수 있다는 사실에 의해 부분적으로 완화됩니다.

새 인덱스가 생성될 때[445], 요소들은 너무 느려질 수 있는 한 번에 하나씩 추가되는 대신 일괄적으로 추가됩니다. 모든 변경 사항이 디스크 상의 비정렬된 목록에 저장되는 대신, maintenance_work_mem 메모리(기본값: 64MB) 청크에 누적되고 이 청크에 더 이상 여유 공간이 없을 때 인덱스로 전송됩니다. 이 작업에 할당된 메모리가 많을수록 인덱스가 구축되는 속도는 더 빨라집니다.

이 장에서 제공된 예시는 검색 정밀도 측면에서 GIN이 GiST 서명 트리보다 우수함을 입증합니다. 이러한 이유로, 전체 텍스트 검색에는 일반적으로 GIN이 사용됩니다. 하지만, 데이터가 활발하게 업데이트되는 경우 GIN 업데이트의 느린 속도는 GiST가 선호될 수 있게 만들 수 있습니다.

결과 세트 크기 제한

GIN 액세스 방법은 항상 결과를 비트맵으로 반환하며, TID를 하나씩 얻는 것은 불가능합니다. 즉, BITMAP SCAN 속성은 지원되지만, INDEX SCAN 속성은 지원되지 않습니다.

이러한 제한의 이유는 정렬되지 않은 지연 업데이트 목록 때문입니다. 인덱스 액세스의 경우, 이 목록이 스캔되어 비트맵을 구축하고, 그 다음 이 비트맵이 트리의 데이터로 업데이트됩니다. 검색이 진행 중인 동안 이 정렬되지 않은 목록이 트리와 합쳐지면(인덱스 업데이트의 결과이거나 vacuuming 동안), 동일한 값이 두 번 반환될 수 있으며, 이는 받아들일 수 없습니다. 하지만 비트맵의 경우 문제가 되지 않습니다: 동일한 비트가 단순히 두 번 설정됩니다.

결과적으로, GIN 인덱스와 LIMIT 절을 사용하는 것은 그다지 효율적이지 않습니다. 왜냐하면 비트맵은 여전히 전체적으로 구축되어야 하며, 이는 전체 비용에 상당한 부분을 차지하기 때문입니다.

```
=> EXPLAIN SELECT * FROM mail_messages
WHERE tsv @@ to_tsquery('hacker')
LIMIT 1000;
                QUERY PLAN
-----------------------------------------------------------
    Limit (cost=481.41..1964.22 rows=1000 width=1258)
        -> Bitmap Heap Scan on mail_messages
            (cost=481.41..74939.28 rows=50214 width=1258)
            Recheck Cond: (tsv @@ to_tsquery('hacker'::text))
            -> Bitmap Index Scan on mail_gin_idx
                (cost=0.00..468.85 rows=50214 width=0)
                Index Cond: (tsv @@ to_tsquery('hacker'::text))
(7 rows)
```

따라서 GIN 방식은 인덱스 검색으로 반환되는 결과의 수를 제한하는 특별한 기능을 제공합니다. 이 제한은

[445] backend/access/gin/gininsert.c, ginbuild function

gin_fuzzy_search_limit(기본값: 0) 매개변수에 의해 적용되며 기본적으로는 비활성화되어 있습니다. 이 매개변수가 활성화되면, 인덱스 접근 방식은 일부 값을 무작위로 건너뛰어 대략 지정된 수의 행을 얻게 됩니다(따라서 "fuzzy"라는 이름이 붙었습니다):[446]

```
=> SET gin_fuzzy_search_limit = 1000;

=> SELECT count(*)
FROM mail_messages
WHERE tsv @@ to_tsquery('hacker');
count
-------
727
(1 row)

=> SELECT count(*)
FROM mail_messages
WHERE tsv @@ to_tsquery('hacker');
count
-------
791
(1 row)

=> RESET gin_fuzzy_search_limit;
```

이 쿼리들에는 LIMIT 절이 없습니다. 인덱스 스캔과 힙 스캔을 사용할 때 다른 데이터를 얻는 유일한 정당한 방법입니다. 플래너는 GIN 인덱스의 이러한 동작에 대해 아무것도 모르며 이 파라미터 값이 비용을 추정할 때 고려되지 않습니다.

속성

GIN 접근 방식의 모든 속성은 모든 수준에서 동일하며 특정 연산자 클래스에 의존하지 않습니다.

액세스 메서드 속성

```
=> SELECT a.amname, p.name, pg_indexam_has_property(a.oid, p.name)
FROM pg_am a, unnest(array[
'can_order', 'can_unique', 'can_multi_col',
'can_exclude', 'can_include'
]) p(name)
WHERE a.amname = 'gin';

 amname |      name     | pg_indexam_has_property
--------+---------------+-------------------------
```

[446] backend/access/gin/ginget.c, dropItem macro

```
    gin ¦     can_order ¦ f
    gin ¦    can_unique ¦ f
    gin ¦ can_multi_col ¦ t
    gin ¦   can_exclude ¦ f
    gin ¦   can_include ¦ f
(5 rows)
```

GIN은 정렬이나 유일성 제약을 지원하지 않습니다.

다중 컬럼 인덱스는 지원되지만, 그 컬럼들의 순서는 중요하지 않다는 점을 언급할 가치가 있습니다. 일반 B-tree와는 달리, 다중 컬럼 GIN 인덱스는 복합 키를 저장하지 않고 대신 해당 컬럼 번호와 함께 별도의 요소를 확장합니다.

INDEX SCAN 속성이 없기 때문에 배타적 제약을 지원할 수 없습니다.

GIN은 추가 INCLUDE 컬럼을 지원하지 않습니다. 이러한 컬럼은 여기서는 거의 의미가 없습니다. 왜냐하면 GIN 인덱스를 커버링으로 사용하기는 어렵기 때문입니다. GIN 인덱스는 인덱스 값의 별도 요소만 포함하며 값 자체는 테이블에 저장됩니다.

인덱스 수준 속성

```
=> SELECT p.name, pg_index_has_property('mail_gin_idx', p.name)
FROM unnest(array[
'clusterable', 'index_scan', 'bitmap_scan', 'backward_scan'
]) p(name);
          name ¦ pg_index_has_property
---------------+------------------------
   clusterable ¦ f
    index_scan ¦ f
   bitmap_scan ¦ t
 backward_scan ¦ f
(4 rows)
```

한 번에 하나씩 결과를 가져오는 것은 지원되지 않으며 인덱스 접근은 항상 비트맵 형태로 결과를 반환합니다.

같은 이유로 GIN 인덱스를 사용하여 테이블을 재정렬하는 것은 의미가 없습니다. 비트맵은 테이블 내 데이터의 물리적 구성을 그대로 반영하기 때문입니다.

역순 스캔 또한 지원되지 않으며, 이 기능은 일반 인덱스 스캔에는 유용할 수 있지만 비트맵 스캔에서는 그렇지 않습니다.

칼럼 수준 속성

```
=> SELECT p.name,
pg_index_column_has_property('mail_gin_idx', 1, p.name)
FROM unnest(array[
'orderable', 'search_array', 'search_nulls',
'returnable', 'distance_orderable'
]) p(name);
               name ¦ pg_index_column_has_property
--------------------+------------------------------
          orderable ¦ f
       search_array ¦ f
       search_nulls ¦ f
         returnable ¦ f
 distance_orderable ¦ f
(5 rows)
```

GIN 인덱스에서는 열 수준의 속성이 지원되지 않습니다. 인덱스를 정렬하거나 커버링 인덱스로 사용하는 것이 불가능한데, 이는 문서 자체가 인덱스에 저장되지 않기 때문입니다. 또한, NULL 값을 지원하지 않습니다. 이는 비원자적 유형의 요소에 대해서는 의미가 없기 때문입니다.

GIN 제한 및 RUM 인덱스

비록 GIN이 강력함에도 불구하고, GIN은 전체 텍스트 검색의 모든 도전을 해결할 수는 없습니다. tsvector 타입은 어휘의 위치를 나타내기는 하지만, 이 정보는 인덱스에 포함되지 않습니다. 따라서 GIN을 사용하여 어휘의 근접성을 고려하는 구문 검색을 가속화하는 것은 불가능합니다. 더욱이, 검색 엔진은 보통 관련성에 따라 결과를 반환하는데(이 용어가 무엇을 의미하든 간에), GIN이 정렬 연산자를 지원하지 않기 때문에 여기서의 유일한 해결책은 각 결과 행에 대해 순위 함수를 계산하는 것이며, 이는 물론 매우 느립니다.

이러한 단점은 RUM 접근 방식으로 해결되었습니다(이 이름으로 미뤄 보았을 때 개발자들이 GIN의 진정한 의미에 대해 얼마나 진정성이 있는지에 대해 의심하게 만듭니다).

이 접근 방식은 확장 기능으로 제공되며, 해당 패키지를 PDGD 저장소[447]에서 다운로드하거나 소스 코드 자체를 얻을 수 있습니다.[448]

RUM은 GIN을 기반으로 하지만, 두 가지 주요 차이점이 있습니다. 첫째, RUM은 지연 업데이트를 제공하지 않으므로 일반 인덱스 스캔을 비트맵 스캔과 함께 지원하며 정렬 연산자를 구현합니다. 둘째, RUM 인덱스 키는 추가 정보로 확장될 수 있습니다. 이 기능은 어느 정도 INCLUDE 컬럼과 유사하지만, 여기서 추가 정보는 특정 키에 연결됩니다. 전문 검색의 맥락에서 RUM 연산자 클래스는 어휘의 발생을 문서 내 위치와 연관시켜 구문 검색 및 검색 결과의 순위 결정 속도를 높입니다.

이 접근법의 단점은 업데이트가 느리고 인덱스 크기가 더 커진다는 것입니다. 게다가, rum 접근 방식은 확장

[447] postgresql.org/download

[448] github.com/postgrespro/rum

기능으로 제공되기 때문에 일반 WAL 메커니즘[449]에 의존하며, 이는 내장 로깅보다 느리고 더 큰 용량의 WAL을 생성합니다.

28.3 트리그램Trigrams

pg_trgm[450] 확장은 세 글자 분절(트리그램)이 일치하는 개수를 비교하여 단어의 유사성을 평가할 수 있습니다. 단어 유사성은 오타가 있는 단어를 검색할 때에도 일부 결과를 반환하기 위해 전문 검색과 함께 사용될 수 있습니다.

gin_trgm_ops 연산자 클래스는 문자열 인덱싱을 구현합니다. 이 클래스는 텍스트 값을 요소로 분리할 때 단어나 어휘 대신 다양한 세 글자 부분문자열을 추출합니다(단, 문자와 숫자만 고려되며, 다른 문자는 무시됩니다). 인덱스 내에서 트리그램은 정수로 표현됩니다. 하지만 유의해야 할 점은 UTF-8 인코딩에서 두 바이트에서 네 바이트를 차지하는 비라틴 문자의 경우, 이러한 표현을 사용해 원래 기호를 디코딩 할 수 없다는 것입니다.

```
=> CREATE EXTENSION pg_trgm;
=> SELECT unnest(show_trgm('macdonald')),
          unnest(show_trgm('McDonald'));

 unnest ¦ unnest
--------+--------
      m ¦ m
     ma ¦ mc
    acd ¦ ald
    ald ¦ cdo
    cdo ¦ don
    don ¦ ld
     ld ¦ mcd
    mac ¦ nal
    nal ¦ ona
    ona ¦
(10 rows)
```

이 클래스는 문자열과 단어의 정확한 비교와 퍼지 비교를 위한 연산자를 지원합니다.

```
=> SELECT amopopr::regoperator, oprcode::regproc
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
JOIN pg_amop amop ON amopfamily = opcfamily
JOIN pg_operator opr ON opr.oid = amopopr
WHERE amname = 'gin'
AND opcname = 'gin_trgm_ops'
ORDER BY amopstrategy;
```

[449] postgresql.org/docs/14/generic-wal.html

[450] postgresql.org/docs/14/pgtrgm.html

```
        amopopr ¦ oprcode
----------------+-----------------------------------
   %(text,text) ¦ similarity_op
  ~~(text,text) ¦ textlike      --- LIKE and ILIKE
 ~~*(text,text) ¦ texticlike    -+
   ~(text,text) ¦ textregexeq   --- regular expressions
  ~*(text,text) ¦ texticregexeq -+
  %>(text,text) ¦ word_similarity_commutator_op
 %>>(text,text) ¦ strict_word_similarity_commutator_op
   =(text,text) ¦ texteq
(8 rows)
```

문자열 간의 흐림 비교를 수행하려면, 쿼리 문자열의 전체 트리그램 수에 대한 공통 트리그램의 비율로서 문자열 간의 거리를 정의할 수 있습니다. 하지만 이미 설명했듯이, GIN은 정렬 연산자를 지원하지 않으므로, 이 클래스의 모든 연산자는 Boolean이어야 합니다. 따라서, 흐림 비교의 전략을 구현하는 %, %>, %>> 연산자의 경우, 계산된 거리가 정의된 임계값을 초과하지 않으면 일관성 함수가 참을 반환합니다.

= 및 LIKE 연산자의 경우, 일관성 함수는 값이 쿼리 문자열의 모든 트라이그램을 포함하도록 요구합니다. 정규 표현식과 문서의 매칭은 훨씬 더 복잡한 검사를 요구합니다.

어떤 경우에도, 트라이그램 검색은 항상 퍼지(fuzzy)하며, 결과는 재확인되어야 합니다.

28.4 인덱싱 배열

GIN 인덱스는 배열 데이터 타입도 지원합니다. 배열 원소를 기반으로 구축된 GIN 인덱스는 어떤 배열이 다른 배열과 겹치는지, 혹은 어떤 배열이 다른 배열에 포함되는지를 빠르게 결정하는 데 사용될 수 있습니다.

```
=> SELECT amopopr::regoperator, oprcode::regproc, amopstrategy
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
JOIN pg_amop amop ON amopfamily = opcfamily
JOIN pg_operator opr ON opr.oid = amopopr
WHERE amname = 'gin'
AND opcname = 'array_ops'
ORDER BY amopstrategy;
               amopopr ¦        oprcode ¦ amopstrategy
-----------------------+----------------+--------------
 &&(anyarray,anyarray) ¦   arrayoverlap ¦ 1
 @>(anyarray,anyarray) ¦  arraycontains ¦ 2
 <@(anyarray,anyarray) ¦ arraycontained ¦ 3
  =(anyarray,anyarray) ¦       array_eq ¦ 4
(4 rows)
```

예를 들어, 데모 데이터베이스의 routes view는 항공편에 대한 정보를 보여줍니다. days_of_week column은 항공편이 운항되는 요일들의 배열입니다. 인덱스를 생성하기 위해서는 먼저 뷰를 구체화해야 합니다:

```
=> CREATE TABLE routes_tbl AS SELECT * FROM routes;
SELECT 710
=> CREATE INDEX ON routes_tbl USING gin(days_of_week);
```

생성한 인덱스를 사용하여 화요일, 목요일, 일요일에 출발하는 비행기를 선택해 보겠습니다. 순차 스캔을 끄지 않으면 계획자가 이렇게 작은 테이블에 대해 인덱스를 사용하지 않습니다.

```
=> SET enable_seqscan = off;
=> EXPLAIN (costs off) SELECT * FROM routes_tbl
WHERE days_of_week = ARRAY[2,4,7];
            QUERY PLAN
-----------------------------------------------------------
    Bitmap Heap Scan on routes_tbl
        Recheck Cond: (days_of_week = '{2,4,7}'::integer[])
        -> Bitmap Index Scan on routes_tbl_days_of_week_idx
            Index Cond: (days_of_week = '{2,4,7}'::integer[])
(4 rows)
```

그 결과, 그러한 항공편이 11편 있는 것으로 밝혀졌습니다:

```
=> SELECT flight_no, departure_airport, arrival_airport,
days_of_week
FROM routes_tbl
WHERE days_of_week = ARRAY[2,4,7];
 flight_no ¦ departure_airport ¦ arrival_airport ¦ days_of_week
-----------+-------------------+-----------------+--------------
    PG0023 ¦               OSW ¦             KRO ¦ {2,4,7}
    PG0123 ¦               NBC ¦             ROV ¦ {2,4,7}
    PG0155 ¦               ARH ¦             TJM ¦ {2,4,7}
    PG0260 ¦               STW ¦             CEK ¦ {2,4,7}
    PG0261 ¦               SVO ¦             GDZ ¦ {2,4,7}
    PG0310 ¦               UUD ¦             NYM ¦ {2,4,7}
    PG0370 ¦               DME ¦             KRO ¦ {2,4,7}
    PG0371 ¦               KRO ¦             DME ¦ {2,4,7}
    PG0448 ¦               VKO ¦             STW ¦ {2,4,7}
    PG0482 ¦               DME ¦             KEJ ¦ {2,4,7}
    PG0651 ¦               UIK ¦             KHV ¦ {2,4,7}
(11 rows)
```

구축된 인덱스에는 일주일 동안의 날짜를 나타내는 정수 1부터 7까지의 7개 요소만 포함되어 있습니다.

이번 쿼리 실행은 앞서 보여드린 전문 검색과 매우 유사합니다. 이 특별한 경우에는 검색 쿼리가 특별한 데이터 유형이 아닌, 일반 배열로 표현됩니다. 인덱싱된 배열이 지정된 모든 요소를 포함해야 한다고 가정합니다. 중요한 차이점은 동등성 조건이 인덱싱된 배열에 다른 요소가 없어야 한다는 것도 요구한다는 점입니다. 일

관성 함수[451]는 전략 번호 덕분에 이 요구 사항을 알고 있지만, 원치 않는 요소가 없다는 것을 확인할 수 없으므로 테이블에 의한 결과의 재검증을 색인 엔진에 요청합니다:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM routes_tbl
WHERE days_of_week = ARRAY[2,4,7];
                QUERY PLAN
-------------------------------------------------------------------
    Bitmap Heap Scan on routes_tbl (actual rows=11 loops=1)
        Recheck Cond: (days_of_week = '{2,4,7}'::integer[])
        Rows Removed by Index Recheck: 482
        Heap Blocks: exact=16
         -> Bitmap Index Scan on routes_tbl_days_of_week_idx (actual ro...
            Index Cond: (days_of_week = '{2,4,7}'::integer[])
(6 rows)
```

GIN 인덱스에 추가 열을 포함시키는 것이 유용할 수 있습니다. 예를 들어, 화요일, 목요일, 그리고 일요일에 모스크바에서 출발하는 항공편을 검색하려면, 인덱스에 출발 도시를 나타내는 departure_city 열이 없습니다. 하지만 정규 스칼라 데이터 유형에 대해 구현된 연산자 클래스는 없습니다.

```
=> CREATE INDEX ON routes_tbl USING gin(days_of_week, departure_city);
ERROR: data type text has no default operator class for access
method "gin"
HINT: You must specify an operator class for the index or define a
default operator class for the data type.
```

이러한 상황은 btree_gin 확장을 사용하여 해결할 수 있습니다. 이 확장은 스칼라 값을 단일 요소로 갖는 합성 값으로 표현함으로써 일반 B-트리 처리를 모방하는 GIN 연산자 클래스를 추가합니다.

```
=> CREATE EXTENSION btree_gin;

=> CREATE INDEX ON routes_tbl USING gin(days_of_week,departure_city);

=> EXPLAIN (costs off)
SELECT * FROM routes_tbl
WHERE days_of_week = ARRAY[2,4,7]
AND departure_city = 'Moscow';
                QUERY PLAN
-------------------------------------------------------------------
    Bitmap Heap Scan on routes_tbl
        Recheck Cond: ((days_of_week = '{2,4,7}'::integer[]) AND
        (departure_city = 'Moscow'::text))
        -> Bitmap Index Scan on routes_tbl_days_of_week_departure_city...
            Index Cond: ((days_of_week = '{2,4,7}'::integer[]) AND
```

[451] backend/access/gin/ginarrayproc.c, ginarrayconsistent function

```
            (departure_city = 'Moscow'::text))
(6 rows)

=> RESET enable_seqscan;
```

btree_gist에 대해 한 언급은 btree_gin에도 해당됩니다: 비교 연산에 있어 B-tree가 훨씬 더 효율적이므로, GIN 인덱스가 정말 필요할 때만 btree_gin 확장을 사용하는 것이 맞습니다. 예를 들어, '미만' 또는 '이하' 조건에 의한 검색은 B-tree에서 역방향 스캔으로 수행할 수 있지만, GIN에서는 그렇지 않습니다.

28.5 JSON 인덱싱

GIN 인덱스를 지원하는 또 다른 비원자적(non-atomic) 데이터 타입은 jsonb[452]입니다. jsonb는 JSON에 대한 다양한 연산자를 제공하며, 그 중 일부는 GIN을 사용하여 더 빠르게 실행될 수 있습니다.

JSON 문서에서 다양한 요소 집합을 추출하는 두 가지 연산자 클래스가 있습니다:

```
=> SELECT opcname
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
WHERE amname = 'gin'
AND opcintype = 'jsonb'::regtype;
opcname
----------------
jsonb_ops
jsonb_path_ops
(2 rows)
```

jsonb_ops 연산자 클래스

jsonb_ops 연산자 클래스는 기본 설정입니다. 원본 JSON 문서의 모든 키, 값 및 배열 요소가 인덱스 항목으로 변환됩니다. 이것은 JSON 값의 포함(@>), 키의 존재(?, ?|, ?&), 또는 JSON 경로의 일치(@? 및 @@)를 확인하는 쿼리를 가속화합니다.

```
=> SELECT amopopr::regoperator, oprcode::regproc, amopstrategy
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
JOIN pg_amop amop ON amopfamily = opcfamily
JOIN pg_operator opr ON opr.oid = amopopr
WHERE amname = 'gin'
AND opcname = 'jsonb_ops'
ORDER BY amopstrategy;
            amopopr |               oprcode | amopstrategy
--------------------+-----------------------+--------------
    @>(jsonb,jsonb) |        jsonb_contains | 7
```

[452] postgresql.org/docs/14/datatype-json.html

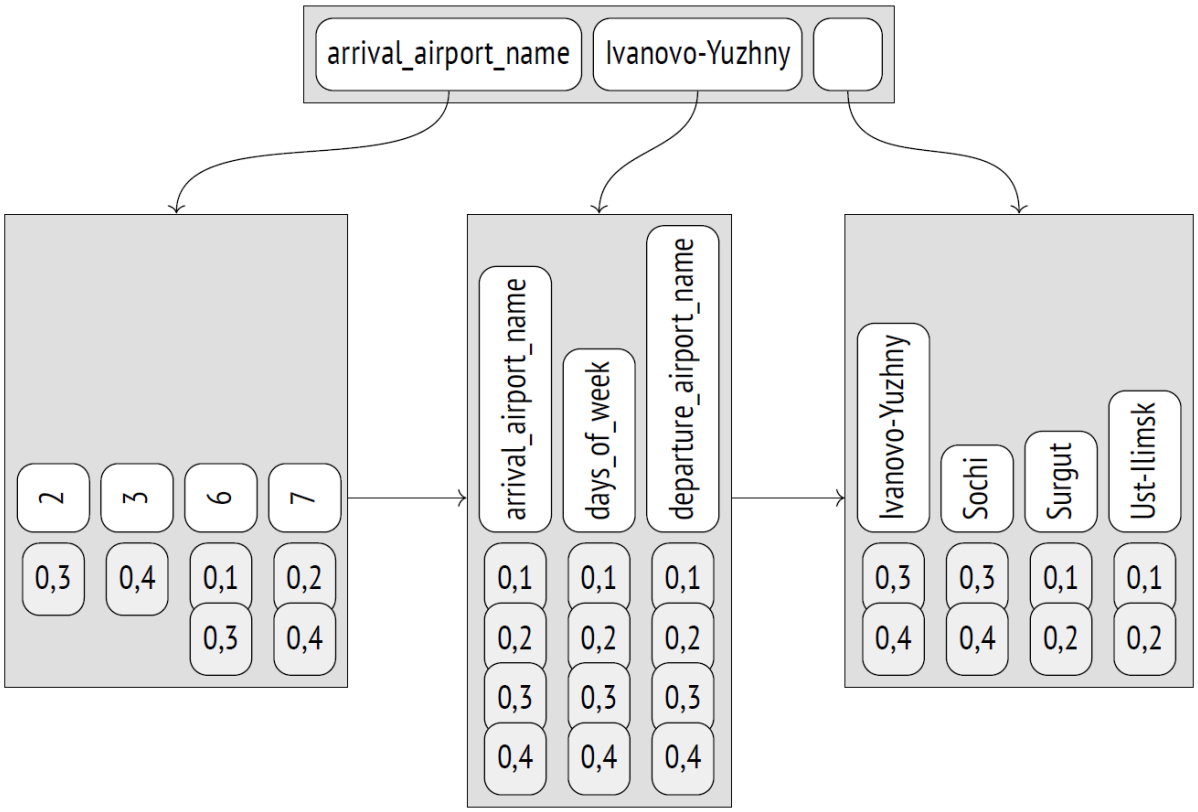
```
     ?(jsonb,text) ¦          jsonb_exists ¦ 9
  ?¦(jsonb,text[]) ¦      jsonb_exists_any ¦ 10
  ?&(jsonb,text[]) ¦      jsonb_exists_all ¦ 11
@?(jsonb,jsonpath) ¦ jsonb_path_exists_opr ¦ 15
@@(jsonb,jsonpath) ¦  jsonb_path_match_opr ¦ 16
(6 rows)
```

여러 개의 루트 뷰 행을 JSON 형식으로 변환해보겠습니다:

```
=> CREATE TABLE routes_jsonb AS
SELECT to_jsonb(t) route
FROM (
SELECT departure_airport_name, arrival_airport_name, days_of_week
FROM routes
ORDER BY flight_no
LIMIT 4
) t;
=> SELECT ctid, jsonb_pretty(route) FROM routes_jsonb;
  ctid ¦ jsonb_pretty
-------+-----------------------------------------------------------
 (0,1) ¦ {                                                        +
       ¦     "days_of_week": [                                    +
       ¦         6                                                +
       ¦     ],                                                   +
       ¦     "arrival_airport_name": "Surgut Airport",            +
       ¦     "departure_airport_name": "Ust-Ilimsk Airport"       +
       ¦ }
 (0,2) ¦ {                                                        +
       ¦     "days_of_week": [                                    +
       ¦         7                                                +
       ¦     ],                                                   +
       ¦     "arrival_airport_name": "Ust-Ilimsk Airport",        +
       ¦     "departure_airport_name": "Surgut Airport"           +
       ¦ }
 (0,3) ¦ {                                                        +
       ¦     "days_of_week": [                                    +
       ¦         2,                                               +
       ¦         6 +
       ¦     ], +
       ¦     "arrival_airport_name": "Sochi International Airport",  +
       ¦     "departure_airport_name": "Ivanovo South Airport"    +
       ¦ }
 (0,4) ¦ {                                                        +
       ¦     "days_of_week": [                                    +
       ¦         3,                                               +
```

```
       ¦        7                                                    +
       ¦    ],                                                       +
       ¦    "arrival_airport_name": "Ivanovo South Airport",         +
       ¦    "departure_airport_name": "Sochi International Airport" +
       ¦ }
(4 rows)
=> CREATE INDEX ON routes_jsonb USING gin(route);
```

생성된 인덱스는 다음과 같이 나타낼 수 있습니다:



조건이 route @> '{"days_of_week": [6]}'인 쿼리를 고려해보겠습니다. 이는 특정 경로(즉, 토요일에 수행되는 항공편)를 포함하는 JSON 문서를 선택합니다.

지원 함수[453]는 검색 쿼리의 JSON 값에서 검색 키를 추출합니다: "days_of_week"와 "6". 이러한 키는 요소 트리에서 검색되며, 그 중 적어도 하나를 포함하는 문서는 일관성 함수[454]에 의해 확인됩니다. contains 전략에 대해, 이 함수는 모든 검색 키가 사용 가능해야 하지만, 결과는 여전히 테이블에 의해 재확인되어야 합니다: 인덱스의 관점에서 볼 때, 지정된 경로는 {"days_of_week": [2], "foo": [6]}과 같은 문서에도 해당할 수 있습니다.

[453] backend/utils/adt/jsonb_gin.c, gin_extract_jsonb_query function

[454] backend/utils/adt/jsonb_gin.c, gin_consistent_jsonb function

jsonb_path_ops 연산자 클래스

두 번째 클래스인 jsonb_path_ops에는 더 적은 연산자가 포함되어 있습니다:

```
=> SELECT amopopr::regoperator, oprcode::regproc, amopstrategy
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
JOIN pg_amop amop ON amopfamily = opcfamily
JOIN pg_operator opr ON opr.oid = amopopr
WHERE amname = 'gin'
AND opcname = 'jsonb_path_ops'
ORDER BY amopstrategy;
            amopopr ¦               oprcode ¦ amopstrategy
--------------------+-----------------------+--------------
    @>(jsonb,jsonb) ¦        jsonb_contains ¦ 7
 @?(jsonb,jsonpath) ¦ jsonb_path_exists_opr ¦ 15
 @@(jsonb,jsonpath) ¦  jsonb_path_match_opr ¦ 16
(3 rows)
```

이 클래스를 사용하면, 인덱스는 격리된 JSON 조각들이 아니라 문서의 루트부터 모든 값과 모든 배열 요소들까지의 경로를 포함하게 됩니다.[455] 이렇게 하면 검색이 훨씬 더 정밀하고 효율적이지만, 경로가 아닌 별도의 키로 표현된 인수들에 대한 연산속도는 향상되지 않습니다.

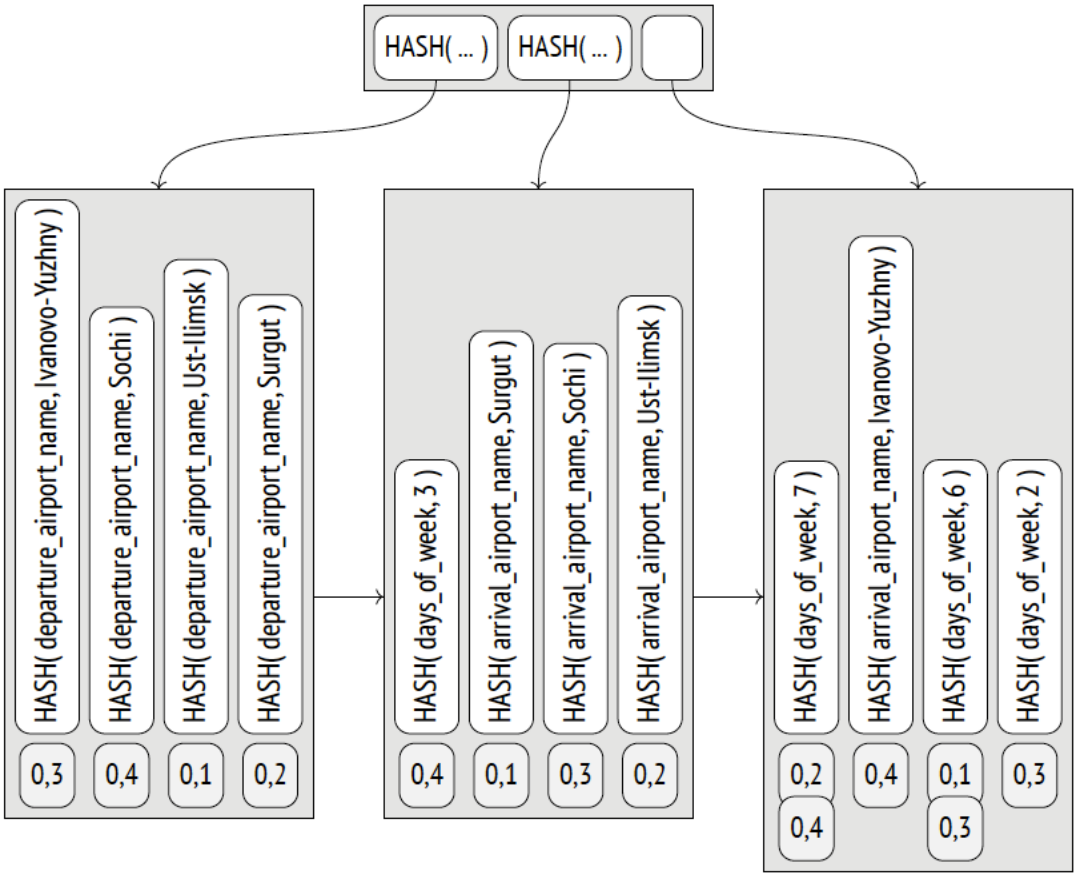
경로는 상당히 길 수 있기 때문에, 실제로 인덱스되는 것은 경로 자체가 아니라 그들의 해시입니다.

이 연산자 클래스를 사용해 같은 테이블에 대한 인덱스를 생성해 봅시다:

```
=> CREATE INDEX ON routes_jsonb USING gin(route jsonb_path_ops);
```

생성된 인덱스는 다음과 같은 트리로 표현될 수 있습니다:

[455] backend/utils/adt/jsonb_gin.c, gin_extract_jsonb_path function



쿼리를 실행할 때 같은 조건 'route @> '{"days_of_week": [6]}'를 사용하면, 지원 함수[456]는 별개의 구성 요소가 아니라 "days_of_week, 6"이라는 전체 경로를 추출합니다. 두 개의 일치하는 문서의 TIDs가 바로 요소 트리에서 찾아집니다.

물론, 이러한 항목은 일관성 함수[457]에 의해 체크되고, 인덱싱 엔진에 의해 재확인됩니다(예를 들어, 해시 충돌을 배제하기 위해). 하지만 트리를 통한 검색이 훨씬 더 효율적이기 때문에, 만약 그것의 연산자가 쿼리에 필요한 인덱스 지원을 충분히 제공한다면 jsonb_path_ops 클래스를 항상 선택하는 것이 좋습니다.

28.6 기타 데이터 유형 인덱싱

GIN 인덱스는 확장을 통해 다음과 같은 데이터 타입에 대해서도 지원합니다:

- **정수 배열**: intarray 확장은 정수 배열을 위한 gin__int_ops 연산자 클래스를 추가합니다. 이것은 표준 array_ops 연산자 클래스와 매우 유사하지만, 문서가 검색 쿼리와 일치하는지 확인하는 매치 연산자 @@를 지원합니다.
- **키-값 저장소**: hstore 확장은 키-값 쌍을 위한 저장소를 구현하고 gin_hstore_ops 연산자 클래스를 제공합니다. 키와 값 모두 인덱스에 포함됩니다.
- **JSON 쿼리 언어**: 외부 jsquery 확장은 자체 쿼리 언어와 JSON을 위한 GIN 인덱스 지원을 제공합니다.

SQL:2016 표준이 채택되고 SQL/JSON 쿼리 언어가 PostgreSQL에 구현된 후에는, 표준 내장 기

[456] backend/utils/adt/jsonb_gin.c, gin_extract_jsonb_query_path function

[457] backend/utils/adt/jsonb_gin.c, gin_consistent_jsonb_path function

능이 더 나은 선택처럼 보입니다.

29 장. BRIN

29.1 개요

BRIN[458] 인덱스는 필요한 행을 빠르게 찾는 다른 인덱스와는 달리 불필요한 행을 필터링하는 데 최적화되어 있습니다. 이 접근 방식은 주로 수 테라바이트 크기의 큰 테이블을 위해 만들어졌으며, 검색 정확도보다는 인덱스 크기의 작음이 우선시됩니다.

검색 속도를 높이기 위해 전체 테이블은 범위로 나뉘며, 여기서 이름이 유래된 Block Range Index 입니다. 각 범위는 여러 페이지로 구성됩니다. 인덱스는 TID를 저장하지 않으며, 각 범위의 데이터에 대한 요약 정보만을 유지합니다. 순서형 데이터 타입의 경우 가장 간단한 형태로 최소값과 최대값이지만, 다양한 연산자 클래스는 범위 내 값에 대한 다양한 정보를 수집할 수 있습니다.

범위당 페이지 수는 인덱스 생성 시 pages_per_range(기본값: 128) 저장 매개변수의 값을 기반으로 정의됩니다.

쿼리 조건이 인덱싱된 컬럼을 참조하는 경우, 일치하는 결과가 결코 없을 것이라고 보장되는 모든 범위는 건너뛸 수 있습니다. 다른 모든 범위의 페이지들은 손실된 비트맵으로 인덱스에 의해 반환되며, 이 페이지들의 모든 로우들은 재검사되어야 합니다. 따라서 BRIN은 값이 서로 가까이 저장되어 유사한 요약 정보 특성을 가지는 컬럼에 잘 작동합니다. 순서형 데이터 타입의 경우, 값들이 오름차순 또는 내림차순으로 저장되어야 하며, 즉 물리적 위치와 크고 작음의 논리적 순서 사이에 높은 상관관계를 가져야 합니다. 다른 종류의 요약 정보의 경우 "유사한 특성"은 달라질 수 있습니다.

BRIN을 전통적인 의미의 인덱스보다는 순차적 힙 스캔의 가속기로 생각하는 것이 틀리지 않습니다. 각 범위가 가상의 파티션을 대표한다고 볼 수 있는 파티셔닝의 대안으로 간주될 수 있습니다.

29.2 예제

우리의 데모 데이터베이스에는 BRIN에 충분히 큰 테이블이 없지만, 특정 공항에서 출발하고 도착한 모든 항공편의 요약 정보를 담은 비정규화 테이블이 분석 보고서에 필요하다고 상상할 수 있습니다. 이 정보에는 사용된 좌석에 대한 정보까지 포함됩니다. 각 공항의 데이터는 해당 시간대에서 자정이 되는 대로 매일 업데이트됩니다. 추가된 데이터는 업데이트되거나 삭제되지 않습니다.

테이블은 다음과 같습니다:

```
CREATE TABLE flights_bi(
    airport_code char(3),
    airport_coord point, -- airport coordinates
    airport_utc_offset interval, -- timezone
    flight_no char(6),
    flight_type text, -- departure or arrival
    scheduled_time timestamptz,
    actual_time timestamptz,
```

[458] postgresql.org/docs/14/brin.html
backend/access/brin/README

```
    aircraft_code char(3),
    seat_no varchar(4),
    fare_conditions varchar(10), -- travel class
    passenger_id varchar(20),
    passenger_name text
);
```

데이터 로딩은 중첩 루프[459]를 사용하여 시뮬레이션할 수 있습니다. 바깥쪽 루프는 일자에 해당하며(데모 데이터베이스는 연간 데이터를 저장합니다), 안쪽 루프는 타임존을 기준으로 합니다. 결과적으로, 루프 내에서 명시적으로 정렬되지는 않았지만, 적어도 시간과 공항별로 어느 정도 순서가 있는 데이터가 로드됩니다.

약 4GB 크기이며 약 3천만 행을 포함하는 기존 데이터베이스의 복사본을 로드하겠습니다:[460]

```
postgres$ pg_restore -d demo -c flights_bi.dump
```

```
=> ANALYZE flights_bi;
=> SELECT count(*) FROM flights_bi;
count
----------
30517076
(1 row)
=> SELECT pg_size_pretty(pg_total_relation_size('flights_bi'));
pg_size_pretty
----------------
4129 MB
(1 row)
```

이 데이터 볼륨을 큰 테이블이라고 부르기는 어렵지만, BRIN이 어떻게 작동하는지 보여주기에는 충분할 것입니다. 미리 인덱스를 생성하겠습니다:

```
=> CREATE INDEX ON flights_bi USING brin(scheduled_time);
=> SELECT pg_size_pretty(pg_total_relation_size(
    'flights_bi_scheduled_time_idx'
));
pg_size_pretty
----------------
184 kB
(1 row)
```

기본 설정으로 매우 적은 공간을 사용합니다.

데이터 중복 제거가 활성화되어 있어도 B-tree 인덱스는 천 배나 더 큽니다. 물론, 그 효율성도 훨씬 높지만,

[459] edu.postgrespro.ru/internals-14/flights_bi.sql

[460] edu.postgrespro.ru/internals-\oldstylenums{14}/flights_bi.dump.

정말 큰 테이블의 경우 추가적인 용량은 감당하기 어려운 사치가 될 수 있습니다.

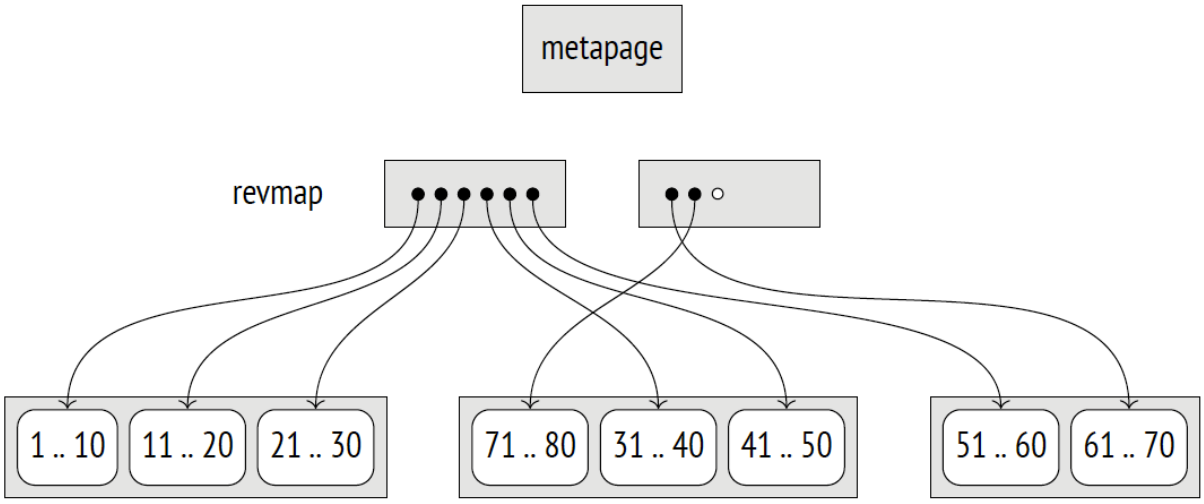
```
=> CREATE INDEX flights_bi_btree_idx ON flights_bi(scheduled_time);
=> SELECT pg_size_pretty(pg_total_relation_size(
    'flights_bi_btree_idx'
));
pg_size_pretty
----------------
210 MB
(1 row)
=> DROP INDEX flights_bi_btree_idx;
```

29.3 페이지 구조

BRIN 인덱스의 제로 페이지는 인덱스 구조에 대한 정보를 유지하는 메타페이지입니다.

메타데이터로부터 특정 오프셋에는 요약 정보가 있는 페이지들이 있습니다. 이러한 페이지의 각 인덱스 항목에는 특정 블록 범위의 요약이 포함되어 있습니다.

메타페이지와 요약 정보 사이의 공간은 범위 맵에 의해 차지되며, 이는 때로는 역 맵으로도 불리며 자주 'revmap' 약어로 언급됩니다. 이것은 사실상 해당 인덱스 행을 가리키는 포인터의 배열이며, 이 배열에서의 인덱스 번호는 범위 번호에 해당합니다.



테이블이 확장됨에 따라, 범위 맵(range map)의 크기도 증가합니다. 만약 맵이 할당된 페이지에 맞지 않게 되면, 다음 페이지를 사용하여 확장하게 되고, 이 페이지에 이전에 있던 모든 인덱스 항목들은 다른 페이지로 옮겨집니다. 한 페이지가 많은 포인터들을 수용할 수 있기 때문에, 이러한 이동은 꽤 드문 경우입니다.

BRIN 인덱스 페이지는 pageinspect 확장을 통해 일반적으로 표시될 수 있습니다. 메타데이터에는 범위 크기와 범위 맵을 위해 예약된 페이지 수가 포함됩니다.

```
=> SELECT pagesperrange, lastrevmappage
FROM brin_metapage_info(get_raw_page(
  'flights_bi_scheduled_time_idx', 0
```

```
));
 pagesperrange ¦ lastrevmappage
---------------+----------------
           128 ¦ 4
(1 row)
```

여기에서 range map은 첫 번째 페이지부터 네 번째 페이지까지 네 페이지를 차지합니다. 우리는 요약된 데이터를 포함하는 인덱스 항목을 가리키는 포인터들을 살펴볼 수 있습니다.

```
=> SELECT *
FROM brin_revmap_data(get_raw_page(
    'flights_bi_scheduled_time_idx', 1
));
pages
----------
(6,197)
(6,198)
(6,199)
...
(6,195)
(6,196)
(1360 rows)
```

만약 범위가 아직 요약되지 않았다면, 범위 맵에서의 포인터는 NULL입니다.

그리고 여기 몇 가지 범위에 대한 요약이 있습니다:

```
=> SELECT itemoffset, blknum, value
FROM brin_page_items(
  get_raw_page('flights_bi_scheduled_time_idx', 6),
'                     flights_bi_scheduled_time_idx'
)
ORDER BY blknum
LIMIT 3 \gx
-[ RECORD 1 ]----------------------------------------------
itemoffset ¦ 197
    blknum ¦ 0
     value ¦ {2016-08-15 02:45:00+03 .. 2016-08-15 16:20:00+03}
-[ RECORD 2 ]----------------------------------------------
itemoffset ¦ 198
    blknum ¦ 128
     value ¦ {2016-08-15 05:50:00+03 .. 2016-08-15 18:55:00+03}
-[ RECORD 3 ]----------------------------------------------
itemoffset ¦ 199
    blknum ¦ 256
```

```
    value ¦ {2016-08-15 07:15:00+03 .. 2016-08-15 18:50:00+03}
```

29.4 검색

쿼리 조건이 BRIN 인덱스[461]에 의해 지원될 경우, 실행자는 각 범위에 대한 범위 맵과 요약 정보를 스캔합니다. 범위 내의 데이터가 검색 키와 일치할 가능성이 있을 경우, 해당 범위에 속하는 모든 페이지가 비트맵에 추가됩니다. BRIN은 개별 튜플의 ID를 유지하지 않기 때문에, 비트맵은 항상 손실이 발생합니다.

검색 키와 데이터의 일치 여부는 일관성 함수에 의해 수행되며, 이 함수는 범위 요약 정보를 해석합니다. 요약되지 않은 범위는 항상 비트맵에 추가됩니다.

수신된 비트맵은 평소와 같은 방식으로 테이블을 스캔하는 데 사용됩니다. 힙 페이지 읽기가 순차적으로, 블록 범위별로 수행되며, 사전 가져오기가 사용된다는 점을 언급하는 것이 중요합니다.

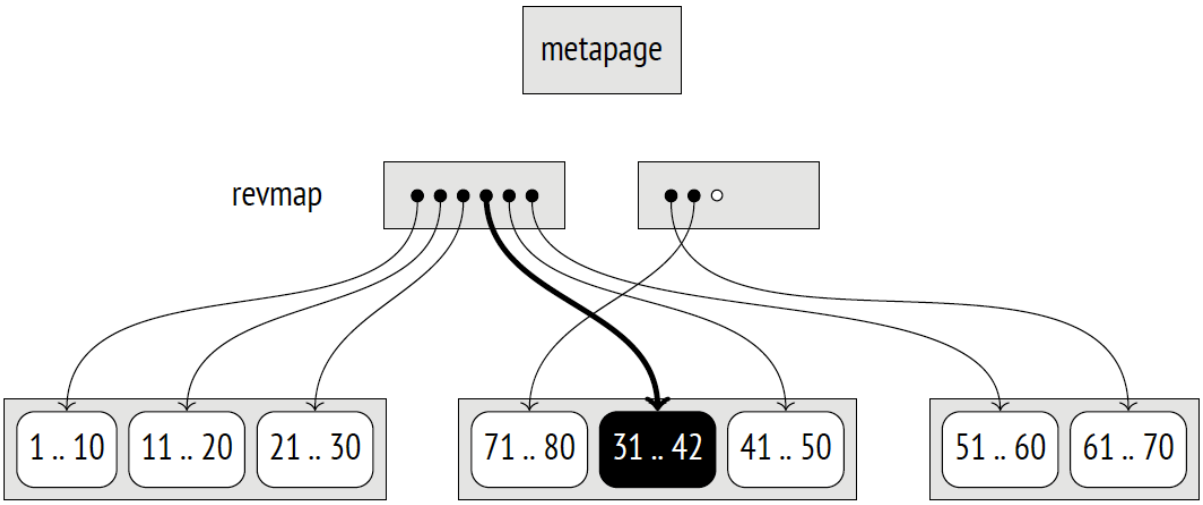
29.5 요약 정보 업데이트

값 입력

새로운 튜플이 heap 페이지에 추가될 때, 해당하는 인덱스 범위의 요약 정보는 업데이트됩니다.[462] 범위 번호는 간단한 산술 연산을 사용하여 페이지 번호를 기반으로 계산되며, 그 후 범위 맵을 통해 요약 정보를 찾습니다.

현재의 요약 정보를 확장해야 할지 결정하기 위해서는 추가 기능이 사용됩니다. 확장이 필요하고 페이지에 충분한 여유 공간이 있다면, 새 인덱스 항목을 추가하지 않고 현장에서 확장됩니다.

예를 들어, 42라는 값을 13페이지에 추가했다고 가정해봅시다. 범위 크기가 4페이지라고 할 때, 페이지 번호를 범위 크기로 나눈 정수 나눗셈을 통해 범위 번호를 계산합니다. 이 경우 범위 번호는 3이 되고; 범위 번호가 0부터 시작하므로 범위 맵의 4번째 포인터를 사용합니다. 이 범위에서의 최소값은 31, 최대값은 40입니다. 추가된 값은 이 범위를 벗어나므로 최대값을 증가시켜야 합니다.



461 backend/access/brin/brin.c, bringetbitmap function

462 backend/access/brin/brin.c, brininsert function

만약 자리를 변경하지 않고 업데이트가 불가능하다면, 새로운 항목을 추가하고 범위 맵이 수정됩니다.

범위 요약

위에서 언급한 모든 내용은 이미 요약된 범위에 새 튜플이 나타나는 시나리오에 해당합니다. 인덱스가 생성될 때는 모든 기존 범위가 요약되지만, 테이블이 성장함에 따라 새 페이지들이 이 범위들을 벗어날 수 있습니다.

인덱스가 자동요약(기본값: 꺼짐) 저장 매개변수를 활성화된 상태로 생성되면, 새 범위는 즉시 요약됩니다. 그러나 데이터 웨어하우스에서는 행들이 대량으로 한꺼번에 추가되는 것이 일반적이므로, 이 모드는 삽입을 심각하게 느리게 할 수 있습니다.

기본적으로 새 범위는 바로 요약되지 않습니다. 요약 정보가 없는 범위는 항상 스캔되기 때문에, 이것은 인덱스의 정확성에 영향을 미치지 않습니다. 요약은 테이블을 청소(vacuuming)하는 동안 또는 `brin_summarize_new_values` 함수(또는 단일 범위를 처리하는 `brin_summarize_range` 함수)를 수동으로 호출하여 비동기적으로 수행됩니다.

범위 요약[463]은 업데이트를 위해 테이블을 잠그지 않습니다. 이 과정이 시작될 때, 해당 범위에 대한 장소 표시자 항목이 인덱스에 삽입됩니다. 이 범위가 스캔되는 동안 범위 내의 데이터가 변경되면, 장소 표시자는 이러한 변경에 대한 요약 정보로 업데이트됩니다. 그 후, 합병 함수가 이 데이터를 해당 범위의 요약 정보와 통합합니다.

이론적으로 어떤 행들이 삭제된 후 요약 정보가 축소될 수도 있지만, GiST 인덱스가 페이지 분할 후 데이터를 재배치할 수 있는 반면에, BRIN 인덱스의 요약 정보는 절대 축소되지 않고 오직 확장만 됩니다. 여기서 축소는 일반적으로 필요하지 않으며, 데이터 저장소는 주로 새 데이터를 추가하는 데만 사용됩니다. 해당 범위를 다시 요약하기 위해 `brin_desummarize_range` 함수를 호출하여 요약 정보를 수동으로 삭제할 수는 있지만, 어떤 범위가 이로부터 혜택을 받을 수 있는지에 대한 단서는 없습니다.

따라서, BRIN은 주로 매우 큰 규모의 테이블을 대상으로 하며, 이 테이블들은 새로운 행이 대부분 파일의 끝에 추가되는 최소한의 업데이트를 가지고 있거나 전혀 업데이트되지 않습니다. 주로 데이터 웨어하우스에서 분석 보고서를 작성하는 데 사용됩니다.

29.6 최소최대 클래스

데이터 유형이 값 비교를 허용하는 경우, 요약 정보에는 최소한 최대값과 최소값이 포함됩니다. 해당 연산자 클래스 이름에는 'minmax'라는 단어가 포함되어 있습니다:[464]

```
=> SELECT opcname
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
WHERE amname = 'brin'
AND opcname LIKE '%minmax_ops'
```

[463] backend/access/brin/brin.c, summarize_range function

[464] backend/access/brin/brin_minmax.c

```
ORDER BY opcname;
opcname
------------------------
bit_minmax_ops
bpchar_minmax_ops
bytea_minmax_ops
char_minmax_ops
...
timestamptz_minmax_ops
timetz_minmax_ops
uuid_minmax_ops
varbit_minmax_ops
(26 rows)
```

이들 연산자 클래스의 지원 함수는 다음과 같습니다:

```
=> SELECT amprocnum, amproc::regproc
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
JOIN pg_amproc amop ON amprocfamily = opcfamily
WHERE amname = 'brin'
AND opcname = 'numeric_minmax_ops'
ORDER BY amprocnum;
 amprocnum ¦ amproc
-----------+-----------------------
         1 ¦ brin_minmax_opcinfo
         2 ¦ brin_minmax_add_value
         3 ¦ brin_minmax_consistent
         4 ¦ brin_minmax_union
(4 rows)
```

첫 번째 함수는 연산자 클래스 메타데이터를 반환하며, 나머지 모든 함수들은 이미 설명된 바와 같이 새로운 값을 삽입하고, 일관성을 검사하며, 유니온(합집합) 연산을 수행합니다.

`minmax` 클래스는 B-트리에서 본 것과 동일한 비교 연산자를 포함합니다.

```
=> SELECT amopopr::regoperator, oprcode::regproc, amopstrategy
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
JOIN pg_amop amop ON amopfamily = opcfamily
JOIN pg_operator opr ON opr.oid = amopopr
WHERE amname = 'brin'
AND opcname = 'numeric_minmax_ops'
ORDER BY amopstrategy;
             amopopr ¦    oprcode ¦ amopstrategy
```

```
--------------------+------------+--------------
  <(numeric,numeric) ¦ numeric_lt ¦ 1
 <=(numeric,numeric) ¦ numeric_le ¦ 2
  =(numeric,numeric) ¦ numeric_eq ¦ 3
 >=(numeric,numeric) ¦ numeric_ge ¦ 4
  >(numeric,numeric) ¦ numeric_gt ¦ 5
(5 rows)
```

인덱싱할 열 선택

어떤 컬럼을 이 연산자 클래스를 사용하여 인덱싱하는 것이 이치에 맞을까요? 앞서 언급했듯이, 이러한 인덱스는 행의 물리적 위치가 값의 논리적 순서와 일치할 때 잘 작동합니다.

위의 예제에서 이를 확인해 봅시다.

```
=> SELECT attname, correlation, n_distinct
FROM pg_stats
WHERE tablename = 'flights_bi'
ORDER BY correlation DESC NULLS LAST;
            attname ¦    correlation ¦ n_distinct
--------------------+----------------+--------------
     scheduled_time ¦      0.9999949 ¦ 25926
        actual_time ¦      0.9999948 ¦ 34469
    fare_conditions ¦      0.7976897 ¦ 3
        flight_type ¦      0.4981733 ¦ 2
 airport_utc_offset ¦      0.4440067 ¦ 11
      aircraft_code ¦     0.19249801 ¦ 8
       airport_code ¦    0.061483838 ¦ 104
            seat_no ¦   0.0024594965 ¦ 461
          flight_no ¦   0.0020146023 ¦ 710
       passenger_id ¦ -0.00046121294 ¦ 2.610987e+06
     passenger_name ¦   -0.012388787 ¦ 8618
      airport_coord ¦                ¦ 0
(12 rows)
```

데이터는 시간(예정된 시간과 실제 시간 모두; 별 차이가 없음)에 의해 정렬됩니다: 새로운 항목들이 시간 순서대로 추가되고, 데이터가 업데이트되거나 삭제되지 않기 때문에 모든 행들이 테이블의 메인 부분에 차례대로 순차적으로 들어갑니다.

'fare_conditions', 'flight_type', 'airport_utc_offset' 열은 상당히 높은 상관관계를 가지고 있지만, 너무 적은 수의 고유한 값을 저장합니다.

다른 열들의 상관관계는 minmax 연산자 클래스로 인덱싱하는 것이 흥미로울 정도로 낮습니다.

범위 크기 및 검색 효율성

적절한 범위의 크기는 특정 값을 저장하는 데 사용되는 페이지 수에 기반하여 결정될 수 있습니다. 예를 들어, 24시간 동안 실행된 모든 비행에 대한 정보를 얻기 위해 'scheduled_time' 열을 살펴봅시다. 이 시간 간격과 관련된 데이터가 차지하는 테이블 페이지의 수를 먼저 알아내야 합니다.

이 숫자를 얻기 위해, TID가 페이지 번호와 오프셋으로 구성된다는 사실을 이용할 수 있습니다. 불행히도, TID를 이 두 구성요소로 분해하는 내장 함수는 없으므로, 텍스트 표현을 통해 타입 캐스팅을 수행하는 우리만의 서툰 함수를 작성해야 합니다.

```
=> CREATE FUNCTION tid2page(t tid) RETURNS integer
LANGUAGE sql
RETURN (t::text::point)[0]::integer;
```

이제 우리는 테이블을 통해 어떻게 날들이 분포되어 있는지 볼 수 있습니다:

```
=> SELECT min(numblk), round(avg(numblk)) avg, max(numblk)
FROM (
SELECT count(distinct tid2page(ctid)) numblk
FROM flights_bi
GROUP BY scheduled_time::date
) t;
  min ¦  avg ¦ max
------+------+------
 1192 ¦ 1447 ¦ 1512
(1 row)
```

데이터 분포가 꽤 균일하지 않다는 것을 알 수 있습니다. 표준 범위 크기가 128페이지인 상황에서, 각 날짜는 9에서 12 범위를 차지할 것입니다. 특정 날짜에 대한 데이터를 가져오는 동안, 인덱스 스캔은 필요한 행뿐만 아니라 같은 범위에 들어간 다른 날짜와 관련된 몇몇 행들도 반환할 것입니다. 범위 크기가 클수록, 더 많은 경계값을 읽게 될 것이며, 범위 크기를 줄이거나 늘려서 이러한 수를 변경할 수 있습니다.

이제 기본 설정으로 생성된 인덱스가 있는 특정 날짜에 대한 쿼리를 시도해 보겠습니다. 간단하게 하기 위해, 병렬 실행은 금지하겠습니다:

```
=> SET max_parallel_workers_per_gather = 0;
=> \set d '2016-08-15 02:45:00+03'
=> EXPLAIN (analyze, buffers, costs off, timing off, summary off)
SELECT *
FROM flights_bi
WHERE scheduled_time >= :'d'::timestamptz
AND scheduled_time < :'d'::timestamptz + interval '1 day';
                    QUERY PLAN
-----------------------------------------------------------------------
    Bitmap Heap Scan on flights_bi (actual rows=81964 loops=1)
        Recheck Cond: ((scheduled_time >= '2016-08-15 02:45:00+03'::ti...
        Rows Removed by Index Recheck: 11606
```

```
        Heap Blocks: lossy=1536
        Buffers: shared hit=1561
        -> Bitmap Index Scan on flights_bi_scheduled_time_idx
            (actual rows=15360 loops=1)
            Index Cond: ((scheduled_time >= '2016-08-15 02:45:00+03'::...
            Buffers: shared hit=25
    Planning:
        Buffers: shared hit=1
(11 rows)
```

BRIN(블록 범위 색인) 인덱스의 특정 쿼리에 대한 효율성 비율을 인덱스 스캔에서 건너뛴 페이지 수와 테이블의 전체 페이지 수 사이의 비율로 정의할 수 있습니다. 효율성 비율이 0이면 인덱스 접근은 (오버헤드 비용을 고려하지 않을 때) 순차 스캐닝으로 저하됩니다. 효율성 비율이 높을수록 읽어야 하는 페이지 수가 줄어듭니다. 하지만 일부 페이지에는 반환되어야 할 데이터가 포함되어 있어 건너뛸 수 없으므로 효율성 비율은 항상 1보다 작습니다.

이 특정 경우에서, 효율성 비율은 528417−1561 / 528417 ≈ 0.997로, 여기서 528,417은 테이블의 페이지 수입니다.

그러나 단일 값에 기반한 의미 있는 결론을 도출할 수 없습니다. 데이터가 균일하고 이상적인 상관 관계를 가지고 있다고 하더라도, 효율성은 최소한 범위 경계가 페이지 경계와 일치하지 않기 때문에 여전히 변동될 수 있습니다. 효율성 비율을 무작위 값으로 취급하고 그 분포를 분석함으로써 전체 그림을 얻을 수 있습니다.

우리의 예제에서, 우리는 연중 다른 모든 날짜를 선택하고, 각각의 값에 대한 실행 계획을 확인하고, 이 선택을 기반으로 통계를 계산할 수 있습니다. EXPLAIN 명령어가 결과를 JSON 형식으로 반환할 수 있기 때문에, 이 과정을 쉽게 자동화할 수 있습니다. 이곳에서 모든 코드를 제공하지는 않겠지만, 다음 코드 조각에는 모든 중요한 세부사항이 포함되어 있습니다:

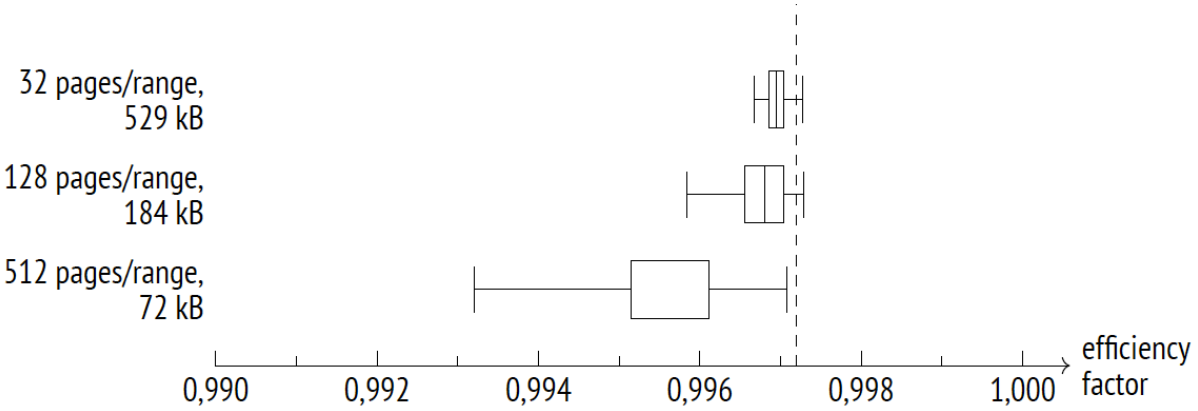
```
=> DO $$
DECLARE
    plan jsonb;
BEGIN
    EXECUTE
        'EXPLAIN (analyze, buffers, timing off, costs off, format json)
            SELECT * FROM flights_bi
            WHERE scheduled_time >= $1
            AND scheduled_time < $1 + interval ''1 day'''
    USING '2016-08-15 02:45:00+03'::timestamptz
    INTO plan;
    RAISE NOTICE 'shared hit=%, read=%',
        plan -> 0 -> 'Plan' ->> 'Shared Hit Blocks',
        plan -> 0 -> 'Plan' ->> 'Shared Read Blocks';
END;
$$;
NOTICE: shared hit=1561, read=0
```

```
DO
```

결과는 상자 수염 그림, 즉 "상자와 수염"으로 시각적으로 표시될 수 있습니다. 여기서 수염은 제1사분위수와 제4사분위수를 나타냅니다(즉, 오른쪽 수염은 가장 큰 값의 25%를, 왼쪽 수염은 가장 작은 값의 25%를 가져갑니다). 상자 자체는 나머지 50%의 값을 보유하고 중앙값을 표시합니다. 더 중요한 것은, 이렇게 간결한 표현으로 다른 결과들을 시각적으로 비교할 수 있다는 것입니다. 다음 그림은 기본 범위 크기와 네 배 더 큰 크기 및 네 배 더 작은 두 가지 크기에 대한 효율성 계수 분포를 보여줍니다.

예상할 수 있듯이, 검색 정확도와 효율성은 상당히 큰 범위에 대해서도 높습니다.

여기서 점선은 테이블의 대략 1/365를 하루가 차지한다고 가정했을 때 이 쿼리에 대해 가능한 최대 효율성 계수의 평균값을 표시합니다.



효율성의 증가는 인덱스 크기 증가의 대가로 이루어집니다. BRIN은 효율성과 크기 사이의 균형을 찾을 수 있도록 상당히 유연합니다.

속성

BRIN 인덱스의 속성은 연산자 클래스에 의존하지 않고 고정되어 있습니다.

액세스 방법 속성

```
=> SELECT a.amname, p.name, pg_indexam_has_property(a.oid, p.name)
FROM pg_am a, unnest(array[
'can_order', 'can_unique', 'can_multi_col',
'can_exclude', 'can_include'
]) p(name)
WHERE a.amname = 'brin';
 amname |          name | pg_indexam_has_property
--------+---------------+-------------------------
   brin |     can_order | f
   brin |    can_unique | f
   brin | can_multi_col | t
```

```
   brin ¦   can_exclude ¦ f
   brin ¦   can_include ¦ f
(5 rows)
```

물론 BRIN 인덱스는 정렬이나 고유성 속성을 지원하지 않습니다. BRIN 인덱스가 항상 비트맵을 반환하기 때문에, 제외 제약 조건도 지원되지 않습니다. 또한, BRIN 인덱스에는 인덱싱 키 자체가 저장되지 않기 때문에 추가 INCLUDE 열은 아무 의미가 없습니다.

그러나, 여러 열에 대한 요약 정보가 각각 별도의 인덱스 항목에 수집되고 저장되지만 여전히 공통된 범위 매핑을 가지는 다중 열 BRIN 인덱스를 생성할 수 있습니다. 이러한 인덱스는 모든 인덱싱된 열에 동일한 범위 크기가 적용될 경우 유용합니다.

또는, 여러 열에 대해 별도의 BRIN 인덱스를 생성하고 비트맵을 함께 병합할 수 있다는 사실을 활용할 수 있습니다. 예를 들어:

```
=> CREATE INDEX ON flights_bi USING brin(airport_utc_offset);
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT *
FROM flights_bi
WHERE scheduled_time >= :'d'::timestamptz
AND scheduled_time < :'d'::timestamptz + interval '1 day'
AND airport_utc_offset = '08:00:00';
                        QUERY PLAN
-----------------------------------------------------------------------
    Bitmap Heap Scan on flights_bi (actual rows=1658 loops=1)
        Recheck Cond: ((scheduled_time >= '2016-08-15 02:45:00+03'::ti...
        Rows Removed by Index Recheck: 14077
        Heap Blocks: lossy=256
        -> BitmapAnd (actual rows=0 loops=1)
            -> Bitmap Index Scan on flights_bi_scheduled_time_idx (act...
                Index Cond: ((scheduled_time >= '2016-08-15 02:45:00+0...
            -> Bitmap Index Scan on flights_bi_airport_utc_offset_idx ...
                Index Cond: (airport_utc_offset = '08:00:00'::interval)
(9 rows)
```

인덱스 수준 속성

```
=> SELECT p.name, pg_index_has_property(
   'flights_bi_scheduled_time_idx', p.name
)
FROM unnest(array[
   'clusterable', 'index_scan', 'bitmap_scan', 'backward_scan'
]) p(name);
          name ¦ pg_index_has_property
```

```
---------------+-----------------------
   clusterable ¦ f
    index_scan ¦ f
   bitmap_scan ¦ t
 backward_scan ¦ f
(4 rows)
```

분명히, 비트맵 스캐닝은 유일하게 지원되는 접근 유형입니다.

클러스터링의 부재는 의아해 보일 수 있습니다. BRIN은 행의 물리적 순서에 민감하기 때문에, 이를 재정렬하여 그 효율성을 극대화하는 것이 논리적이라고 생각할 수 있습니다. 하지만, 대규모 테이블의 클러스터링은 테이블을 재구성하는 데 필요한 모든 처리 및 추가 디스크 공간을 고려할 때 사치입니다. 게다가, 'flights_bi' 테이블의 예시가 보여주듯, 일부 데이터 저장에서는 자연스럽게 일정한 정렬이 발생할 수 있습니다.

칼럼 수준 속성

```
=> SELECT p.name, pg_index_column_has_property(
'flights_bi_scheduled_time_idx', 1, p.name
)
FROM unnest(array[
'orderable', 'distance_orderable', 'returnable',
'search_array', 'search_nulls'
]) p(name);
               name ¦ pg_index_column_has_property
--------------------+-------------------------------
          orderable ¦ f
 distance_orderable ¦ f
         returnable ¦ f
       search_array ¦ f
       search_nulls ¦ t
(5 rows)
```

유일하게 사용할 수 있는 컬럼 레벨 속성은 NULL 값 지원입니다. 범위 내의 NULL 값을 추적하기 위해, 요약 정보는 별도의 속성을 제공합니다:

```
=> SELECT hasnulls, allnulls, value
FROM brin_page_items(
get_raw_page('flights_bi_airport_utc_offset_idx', 6),
'flights_bi_airport_utc_offset_idx'
)
WHERE itemoffset= 1;
 hasnulls ¦ allnulls ¦ value
----------+----------+------------------------
        f ¦        f ¦ {03:00:00 .. 03:00:00}
```

```
(1 row)
```

29.7 최저최고 다중 클래스

데이터 업데이트로 인해 확립된 상관관계가 쉽게 방해받을 수 있습니다. 문제가 되는 것은 특정 값의 실제 수정이 아니라, MVCC(Multi-Version Concurrency Control) 설계 자체 때문입니다: 한 페이지에서 행의 이전 버전이 삭제될 수 있지만, 그 새로운 버전은 현재 비어 있는 어떤 위치에도 삽입될 수 있어 원래의 행 순서를 유지할 수 없습니다.

이러한 효과를 어느 정도 줄이려면 저장소 매개 변수인 fillfactor의 값을 줄여 페이지 내에 미래 업데이트를 위한 공간을 더 많이 남겨둘 수 있습니다. 그러나 이미 거대한 테이블의 크기를 늘리는 것이 정말 가치가 있을까요? 게다가 삭제 작업은 어쨌든 기존 페이지에서 어느 정도 공간을 해방시켜, 그렇지 않았다면 파일 끝에 도달했을 새로운 튜플들을 위한 함정을 준비합니다.

이러한 상황은 쉽게 모사할 수 있습니다. 임의로 선택된 행의 0.1%를 삭제하고 테이블을 정리하여 새 튜플을 위한 공간을 마련하기 위해 vacuum 작업을 수행해 봅시다:

```
=> WITH t AS (
SELECT ctid
FROM flights_bi TABLESAMPLE BERNOULLI(0.1) REPEATABLE(0)
)
DELETE FROM flights_bi
WHERE ctid IN (SELECT ctid FROM t);
DELETE 30180
=> VACUUM flights_bi;
```

이제 한 시간대의 새로운 날에 대한 데이터를 추가해 봅시다. 이전 날의 데이터를 그대로 복사하겠습니다.

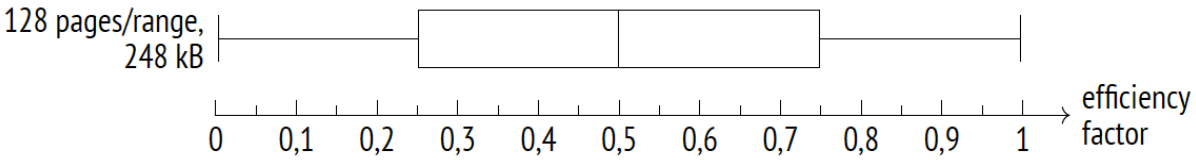
```
=> INSERT INTO flights_bi
SELECT airport_code, airport_coord, airport_utc_offset,
  flight_no, flight_type, scheduled_time + interval '1 day',
  actual_time + interval '1 day', aircraft_code, seat_no,
  fare_conditions, passenger_id, passenger_name
FROM flights_bi
WHERE date_trunc('day', scheduled_time) = '2017-08-15'
AND airport_utc_offset = '03:00:00';
INSERT 0 40532
```

수행된 삭제 작업은 모든 범위 또는 거의 모든 범위에서 일부 공간을 확보하기에 충분했습니다. 파일 중간 어딘가에 위치한 페이지로 들어간 새로운 튜플들은 자동으로 범위를 확장했습니다. 예를 들어, 첫 번째 범위와 관련된 요약 정보는 이전에는 하루 미만을 커버했지만, 이제는 전체 연도를 포함하게 되었습니다.

```
=> SELECT value
FROM brin_page_items(
get_raw_page('flights_bi_scheduled_time_idx', 6),
'flights_bi_scheduled_time_idx'
```

```
)
WHERE blknum = 0;
value
--------------------------------------------------
{2016-08-15 02:45:00+03 .. 2017-08-16 09:35:00+03}
(1 row)
```

쿼리에서 지정된 날짜가 더 작을수록 더 많은 범위를 스캔해야 합니다. 그래프는 재앙의 크기를 보여줍니다:



이 문제를 해결하기 위해서는 요약 정보를 조금 더 복잡하게 만들어야 합니다. 값 전체를 커버하는 단일 연속 범위 대신 여러 개의 작은 범위를 저장하여 함께 사용할 수 있어야 합니다. 그러면 하나의 범위가 주된 데이터 세트를 커버할 수 있고, 나머지는 이따금 발생하는 이상치(outliers)를 처리할 수 있습니다.

이러한 기능은 minmax-multi 연산자 클래스에 의해 제공됩니다.[465]

```
=> SELECT opcname
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
WHERE amname = 'brin'
AND opcname LIKE '%minmax_multi_ops'
ORDER BY opcname;
opcname
------------------------------
date_minmax_multi_ops
float4_minmax_multi_ops
float8_minmax_multi_ops
inet_minmax_multi_ops
...
time_minmax_multi_ops
timestamp_minmax_multi_ops
timestamptz_minmax_multi_ops
timetz_minmax_multi_ops
uuid_minmax_multi_ops
(19 rows)
```

minmax 연산자 클래스와 비교할 때, minmax-multi 클래스는 값 사이의 거리를 계산하는 하나의 추가 지원 함수가 있습니다; 이것은 연산자 클래스가 줄이려고 노력하는 범위 길이를 결정하는 데 사용됩니다.

[465] backend/access/brin/brin_minmax_multi.c

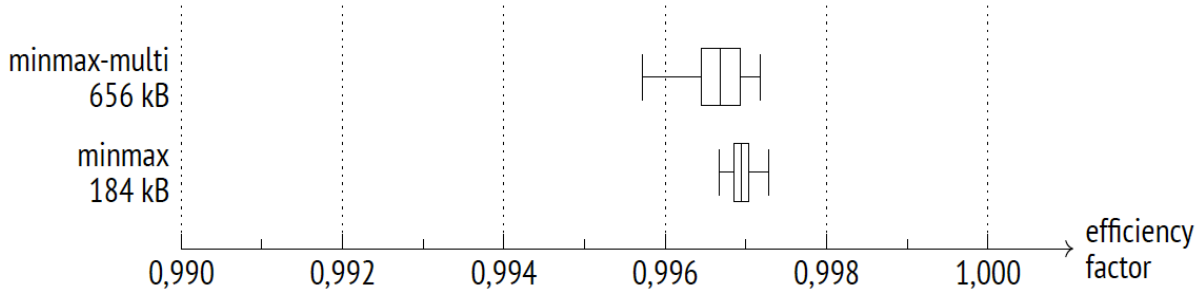
```
=> SELECT amprocnum, amproc::regproc
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
JOIN pg_amproc amop ON amprocfamily = opcfamily
WHERE amname = 'brin'
AND opcname = 'numeric_minmax_multi_ops'
ORDER BY amprocnum;
 amprocnum ¦ amproc
-----------+-----------------------------------
         1 ¦ brin_minmax_multi_opcinfo
         2 ¦ brin_minmax_multi_add_value
         3 ¦ brin_minmax_multi_consistent
         4 ¦ brin_minmax_multi_union
         5 ¦ brin_minmax_multi_options
        11 ¦ brin_minmax_multi_distance_numeric
(6 rows)
```

해당 클래스의 연산자는 minmax 클래스의 것과 완전히 동일합니다.
Minmax-multi 클래스는 값 당 범위(values_per_range, 기본값: 32) 매개변수를 사용할 수 있으며, 이는 범위별로 허용되는 요약된 값의 최대 개수를 정의합니다. 요약된 값은 두 개의 숫자(간격)로 표현되며, 별도의 점은 단 하나의 숫자만 필요로 합니다. 충분한 값이 없는 경우 일부 간격은 축소됩니다.[466]

기존 인덱스 대신 minmax-multi 인덱스를 만들어 보겠습니다. 범위 당 허용되는 값의 수를 16으로 제한하겠습니다:

```
=> DROP INDEX flights_bi_scheduled_time_idx;
=> CREATE INDEX ON flights_bi USING brin(
  scheduled_time timestamptz_minmax_multi_ops(
              values_per_range = 16
  )
);
```

그래프는 새로운 인덱스가 효율성을 원래 수준으로 되돌린 것을 보여줍니다. 예상대로, 이는 인덱스 크기의 증가로 이어집니다.



[466] backend/access/brin/brin_minmax_multi.c, reduce_expanded_ranges function

29.8 포함 클래스

minmax와 포함(inclusion) 연산자 클래스의 차이는 대략 B-트리와 GiST 인덱스의 차이와 유사합니다: 후자는 비교 연산을 지원하지 않는 데이터 유형을 위해 설계되었지만, 값의 상호 정렬이 여전히 그들에게 의미가 있습니다. 특정 범위에 대한 포함 연산자 클래스에 의해 제공되는 요약 정보는 이 범위에 있는 값들의 경계 상자로 표현됩니다.

다음은 이 연산자 클래스입니다; 그들은 많지 않습니다:

```
=> SELECT opcname
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
WHERE amname = 'brin'
AND opcname LIKE '%inclusion_ops'
ORDER BY opcname;
opcname
----------------------
box_inclusion_ops
inet_inclusion_ops
range_inclusion_ops
(3 rows)
```

지원 기능 목록은 두 값을 병합하는 하나의 필수 기능으로 확장되었고, 선택적으로 사용할 수 있는 다양한 기능들도 추가되었습니다.

```
=> SELECT amprocnum, amproc::regproc
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
JOIN pg_amproc amop ON amprocfamily = opcfamily
WHERE amname = 'brin'
AND opcname = 'box_inclusion_ops'
ORDER BY amprocnum;
 amprocnum ¦ amproc
-----------+---------------------------
         1 ¦ brin_inclusion_opcinfo
         2 ¦ brin_inclusion_add_value
         3 ¦ brin_inclusion_consistent
         4 ¦ brin_inclusion_union
        11 ¦ bound_box
        13 ¦ box_contain
(6 rows)
```

값을 비교할 수 있는 경우, 우리는 그들의 상관관계에 의존했습니다; 하지만 다른 데이터 유형에 대해서는 그

러한 통계가 수집되지 않아[467] 포함 기반 BRIN 인덱스의 효율성을 예측하기 어렵습니다. 더 나쁜 것은, 상관관계가 인덱스 스캔의 비용 추정에 큰 영향을 미친다는 점입니다. 이러한 통계가 없을 경우, 그것은 0으로 간주됩니다.[468] 따라서, 플래너는 정확한 포함 인덱스와 모호한 포함 인덱스를 구별할 방법이 없으므로, 일반적으로 이를 전혀 사용하지 않는 것을 선호합니다.

> PostGIS 는 공간 데이터의 상관관계에 대한 통계를 수집합니다.

특정 경우, 공항의 좌표 위에 인덱스를 생성하는 것이 매우 타당합니다. 왜냐하면 경도는 시간대와 상관 관계가 있을 것이기 때문입니다. GiST 예측과는 다르게, BRIN 요약 정보는 인덱싱된 데이터와 같은 형식을 가지므로 포인트에 대한 인덱스를 생성하기가 그다지 쉽지 않습니다. 하지만 포인트를 가짜 사각형으로 변환하여 표현식 인덱스를 생성할 수 있습니다.

```
=> CREATE INDEX ON flights_bi USING brin(box(airport_coord))
WITH (pages_per_range = 8);
=> SELECT pg_size_pretty(pg_total_relation_size(
'flights_bi_box_idx'
));
pg_size_pretty
----------------
3816 kB
(1 row)
```

시간대를 위해 동일한 범위 크기로 구축된 인덱스는 대략 동일한 용량을 차지합니다(3288KB). 이 클래스에 포함된 연산자들은 GiST 연산자들과 유사합니다. 예를 들어, 특정 지역 내의 점들을 검색하는 속도를 높이기 위해 BRIN 인덱스를 사용할 수 있습니다.

```
=> SELECT airport_code, airport_name
FROM airports
WHERE box(coordinates) <@ box '135,45,140,50';
 airport_code ¦ airport_name
--------------+--------------------------
          KHV ¦ Khabarovsk-Novy Airport
(1 row)
```

하지만 앞서 언급했듯이, 계획자는 순차 스캔을 끄지 않는 이상, 인덱스 스캔을 사용하지 않기로 거부합니다.

```
=> EXPLAIN (costs off)
SELECT *
FROM flights_bi
WHERE box(airport_coord) <@ box '135,45,140,50';
                QUERY PLAN
------------------------------------------------------------
    Seq Scan on flights_bi
```

[467] backend/commands/analyze.c, compute_scalar_stats function

[468] backend/utils/adt/selfuncs.c, brincostestimate function

```
        Filter: (box(airport_coord) <@ '(140,50),(135,45)'::box)
(2 rows)

=> SET enable_seqscan = off;
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT *
FROM flights_bi
WHERE box(airport_coord) <@ box '135,45,140,50';
                    QUERY PLAN
-----------------------------------------------------------------------
    Bitmap Heap Scan on flights_bi (actual rows=511414 loops=1)
        Recheck Cond: (box(airport_coord) <@ '(140,50),(135,45)'::box)
        Rows Removed by Index Recheck: 630756
        Heap Blocks: lossy=19656
        -> Bitmap Index Scan on flights_bi_box_idx (actual rows=196560...
            Index Cond: (box(airport_coord) <@ '(140,50),(135,45)'::box)
(6 rows)
=> RESET enable_seqscan;
```

29.9 블룸 클래스

기반 연산자 클래스는 동일한 연산을 지원하고 해시 함수가 정의된 모든 데이터 유형에 대해 BRIN 사용을 가능하게 합니다. 또한, 값이 별도의 범위에 지역화되어 있지만, 물리적 위치가 논리적 순서와 상관관계가 없는 경우, 일반 순서 유형에도 적용될 수 있습니다.

이러한 연산자 클래스의 이름에는 'bloom'이라는 단어가 포함됩니다.[469]

```
=> SELECT opcname
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
WHERE amname = 'brin'
AND opcname LIKE '%bloom_ops'
ORDER BY opcname;
opcname
-----------------------
bpchar_bloom_ops
bytea_bloom_ops
char_bloom_ops
...
timestamptz_bloom_ops
timetz_bloom_ops
uuid_bloom_ops
(24 rows)
```

[469] backend/access/brin/brin_bloom.c

클래식 블룸 필터는 집합에 요소가 속해 있는지 빠르게 확인할 수 있는 데이터 구조입니다. 이 필터는 매우 컴팩트하지만, 거짓 긍정을 허용합니다: 실제보다 더 많은 요소를 포함하고 있다고 가정할 수 있습니다. 하지만 더 중요한 것은, 거짓 부정은 배제된다는 것입니다: 실제로 존재하는 요소가 집합에 없다고 판단할 수 없습니다.

필터는 원래 0으로 채워진 m비트(시그니처라고도 함) 배열입니다. 집합의 모든 요소를 시그니처의 k비트에 매핑하는 k개의 다른 해시 함수를 선택합니다. 요소가 집합에 추가될 때, 시그니처의 각 비트는 1로 설정됩니다. 그 결과, 요소에 해당하는 모든 비트가 1로 설정되어 있다면, 요소가 집합에 있을 수 있습니다; 적어도 하나의 0 비트가 있다면, 요소는 확실히 집합에 없습니다.

BRIN 인덱스의 경우, 필터는 특정 범위에 속하는 인덱싱된 열의 값 집합을 처리하며, 이 범위에 대한 요약 정보는 생성된 블룸 필터로 표시됩니다.

> 블룸 확장[470]은 블룸 필터 기반의 자체 인덱스 접근 방식을 제공합니다. 이는 각 테이블 행에 대한 필터를 구축하고 각 행의 컬럼 값 집합을 다룹니다. 이러한 인덱스는 한 번에 여러 컬럼을 인덱싱하는 데 설계되었으며, 필터 조건에서 참조될 컬럼이 사전에 알려지지 않은 임시 질의에서 사용될 수 있습니다. BRIN 인덱스도 여러 컬럼에 대해 구축될 수 있지만, 그 요약 정보는 이 컬럼들 각각에 대해 여러 개의 독립적인 블룸 필터를 포함하게 됩니다.

블룸 필터의 정확도는 서명 길이에 따라 달라집니다. 이론적으로, 최적의 서명 비트 수는 m = -nlog2P / ln2로 추정할 수 있으며, 여기서 n은 세트의 요소 수이고 p는 오진 확률입니다.

이 두 설정은 해당 연산자 클래스 매개변수를 사용하여 조정할 수 있습니다.

- `n_distinct_per_range(기본값: -0.1)`은 집합 내 요소의 수를 정의합니다; 이 경우에는 색인된 열의 한 범위 내의 고유값의 수입니다. 이 매개변수 값은 고유값에 대한 통계처럼 해석됩니다: 음수 값은 범위 내 행의 비율을 나타내며, 그들의 절대 수를 나타내지 않습니다.
- `false_positive_rate(기본값: 0.01)`은 거짓 양성의 확률을 정의합니다. 거의 0에 가까운 값은 인덱스 스캔이 검색된 값이 없는 범위를 거의 확실히 건너뛸 것임을 의미합니다. 하지만, 검색된 범위에는 쿼리와 일치하지 않는 추가 행이 포함되어 있으므로 정확한 검색을 보장하지 않습니다. 이와 같은 동작은 실제 필터의 특성이 아니라 범위의 너비와 물리적 데이터 위치 때문입니다.

지원하는 함수 목록이 해시 함수로 확장되었습니다:

```
=> SELECT amprocnum, amproc::regproc
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
JOIN pg_amproc amop ON amprocfamily = opcfamily
WHERE amname = 'brin'
AND opcname = 'numeric_bloom_ops'
ORDER BY amprocnum;
 amprocnum ¦ amproc
```

[470] postgresql.org/docs/14/bloom.html

```
-----------+----------------------
         1 ¦ brin_bloom_opcinfo
         2 ¦ brin_bloom_add_value
         3 ¦ brin_bloom_consistent
         4 ¦ brin_bloom_union
         5 ¦ brin_bloom_options
        11 ¦ hash_numeric
(6 rows)
```

블룸 필터는 해싱을 기반으로 하기 때문에, 오직 동등 연산자만 지원됩니다.

```
=> SELECT amopopr::regoperator, oprcode::regproc, amopstrategy
FROM pg_am am
JOIN pg_opclass opc ON opcmethod = am.oid
JOIN pg_amop amop ON amopfamily = opcfamily
JOIN pg_operator opr ON opr.oid = amopopr
WHERE amname = 'brin'
AND opcname = 'numeric_bloom_ops'
ORDER BY amopstrategy;
            amopopr ¦    oprcode ¦ amopstrategy
--------------------+------------+--------------
 =(numeric,numeric) ¦ numeric_eq ¦ 1
(1 row)
```

비행 번호를 저장하는 flight_no 열은 거의 상관 관계가 없으므로 일반 범위 연산자 클래스에는 쓸모가 없습니다. 우리는 기본 false-positive 설정을 유지할 것이며, 범위 내의 고유 값 수는 쉽게 계산할 수 있습니다. 예를 들어, 8페이지 범위의 경우 다음과 같은 값을 얻습니다:

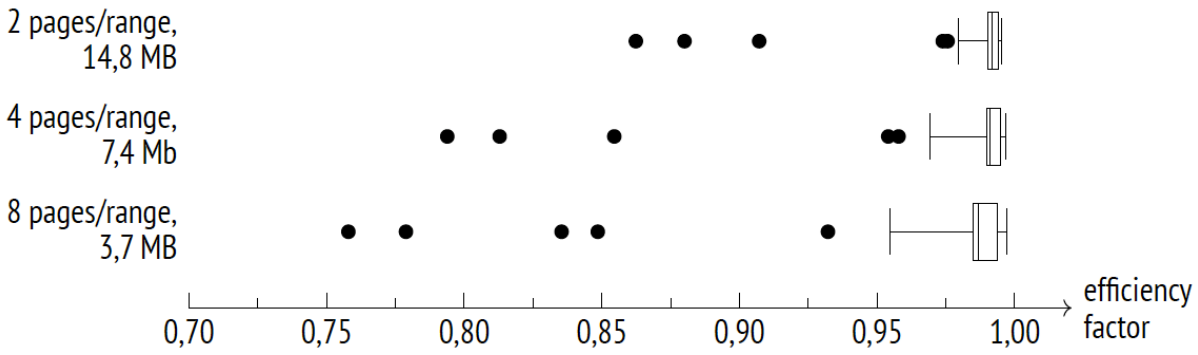
```
=> SELECT max(nd)
FROM (
SELECT count(distinct flight_no) nd
FROM flights_bi
GROUP BY tid2page(ctid) / 8
) t;
max
-----
22
(1 row)
```

더 작은 범위의 경우, 이 숫자는 더 낮을 것입니다(하지만 어떤 경우에도, 연산자 클래스는 16보다 작은 값을 허용하지 않습니다). 이제 인덱스를 만들고 실행 계획을 확인하기만 하면 됩니다.

```
=> CREATE INDEX ON flights_bi USING brin(
flight_no bpchar_bloom_ops(
n_distinct_per_range = 22)
)
```

```
WITH (pages_per_range = 8);
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT *
FROM flights_bi
WHERE flight_no = 'PG0001';
                        QUERY PLAN
-----------------------------------------------------------------------
    Bitmap Heap Scan on flights_bi (actual rows=5192 loops=1)
        Recheck Cond: (flight_no = 'PG0001'::bpchar)
        Rows Removed by Index Recheck: 122894
        Heap Blocks: lossy=2168
        -> Bitmap Index Scan on flights_bi_flight_no_idx (actual rows=...
            Index Cond: (flight_no = 'PG0001'::bpchar)
(6 rows)
=> RESET max_parallel_workers_per_gather;
```

그래프는 일부 항공편 번호들(모든 수염(whiskers)에 속하지 않는 별도의 점들로 표현됨)에 대해서는 인덱스의 작동이 그다지 효율적이지 않음을 보여주지만, 전반적인 효율성은 상당히 높습니다:



마무리

이제 우리의 여정이 끝나가고 있습니다. 이 책이 유용하거나 적어도 흥미롭다고 느끼셨기를 바라며, 여러분이 새로운 것을 배웠기를 바랍니다(저 역시 이 책을 쓰면서 많은 것을 배웠습니다).

대부분의 내용은 상당 기간 동안 최신 상태를 유지할 가능성이 높지만, 일부 세부 정보는 불가피하게 매우 빠르게 변할 것입니다. 저는 이 책의 가장 큰 가치는 특정 사실의 집합이 아니라, 제가 보여주는 시스템 탐색에 대한 접근 방식에 있다고 믿습니다. 이 책이나 문서를 맹목적으로 받아들이지 마십시오. 고민하고, 실험하고, 모든 사실을 직접 확인해보십시오: 포스트그레스큐에은 이를 위한 모든 도구를 제공하며, 저는 그 사용 방법을 보여주려고 노력했습니다. 보통은 포럼에 질문을 올리거나 구글링하는 것만큼 쉽지만, 확실히 더 신뢰할 수 있고 유용합니다.

같은 이유로, 코드를 살펴보라는 권장의 말씀을 드리고 싶습니다. 그 복잡성에 주눅 들지 마시고, 그저 시도해 보십시오. 오픈 소스는 큰 장점이므로, 이 기회를 활용하십시오.

피드백을 보내주시면 기쁠 것입니다; 의견과 아이디어를 edu@postgrespro.ru로 보내주십시오. 저는 정기적으로 책을 업데이트 할 계획이므로, 여러분의 의견이 미래 판을 개선하는데 도움이 될 것입니다. 책의 최신 온라인 버전은 postgrespro.com/community/books/internals에서 무료로 이용할 수 있습니다.
행운을 빕니다!