13장. 동시성 제어

이 장에서는 두 개 이상의 세션이 동시에 동일한 데이터에 액세스하려고 할 때 PostgreSQL 데이터베이스 시스템의 동작에 대해 설명합니다. 이러한 상황의 목표는 엄격한 데이터 무결성을 유지하면서 모든 세션에 대한 효율적인 액세스를 허용하는 것입니다. 데이터베이스 애플리케이션을 개발하는 모든 개발자는 이 장에서 다루는 주제를 숙지하고 있어야 합니다.

13.1. 소개

PostgreSQL은 개발자가 데이터에 대한 동시 액세스를 관리할 수 있는 다양한 도구를 제공합니다. 내부적으로는 다중 버전 모델(다중 버전 동시성 제어, MVCC)을 사용하여 데이터 일관성을 유지합니다. 즉, 각 SQL 문은 기초 데이터의 현재 상태와 관계없이 얼마 전의 데이터 스냅샷(*데이터베이스 버전*)을 보게 됩니다. 이렇게 하면 동일한 데이터 행에 대해 업데이트를 수행하는 동시 트랜잭션에서 생성된 일관되지 않은 데이터를 문에서 볼 수 없게 되어 각 데이터베이스 세션에 대해 *트랜잭션 격리*가 제공됩니다. MVCC는 기존 데이터베이스 시스템의 잠금 방법론을 피함으로써 잠금 경합을 최소화하여 다중 사용자 환경에서도 성능을 유지할 수 있도록 합니다.

잠금 대신 MVCC 모델의 동시성 제어를 사용하면 얻을 수 있는 가장 큰 장점은 MVCC에서 데이터 쿼리(읽기)를 위해 획득한 잠금이 데이터 쓰기를 위해 획득한 잠금과 충돌하지 않기 때문에 읽기가 쓰기를 차단하지 않고 쓰기가 읽기를 차단하지 않는다는 점입니다. PostgreSQL은 혁신적인 *직렬화 가능 스냅샷 격리*(SSI) 수준을 사용하여 가장 엄격한 수준의 트랜잭션 격리를 제공하는 경우에도 이 보장을 유지합니다.

일반적으로 완전한 트랜잭션 격리가 필요하지 않고 특정 충돌 지점을 명시적으로 관리하는 것을 선호하는 애플리케이션을 위해 테이블 및 행 수준 잠금 기능도 PostgreSQL에서 사용할 수 있습니다. 그러나 MVCC를 적절히 사용하면 일반적으로 잠금보다 더 나은 성능을 제공합니다. 또한 애플리케이션 정의 자문 잠금은 단일 트랜잭션에 묶이지 않는 잠금을 획득할 수 있는 메커니즘을 제공합니다.

13.2. 트랜잭션 격리

SQL 표준은 4가지 수준의 트랜잭션 격리를 정의합니다. 가장 엄격한 것은 직렬화 가능으로, 표준에서는 직렬화 가능 트랜잭션 집합을 동시에 실행하면 순서대로 한 번에 하나씩 실행하는 것과 동일한 효과가 발생한다고 단락에서 정의하고 있습니다. 나머지 세 가지 수준은 동시 실행 트랜잭션 간의 상호 작용으로 인해 발생하는 현상으로 정의되며, 각 수준에서 발생하지 않아야 합니다. 이 표준은 직렬화 가능의 정의로 인해 해당 수준에서는 이러한 현상이 발생하지 않는다고 명시하고 있습니다. (트랜잭션의 효과가 한 번에 하나씩 실행되는 것과 일치해야 한다면 어떻게 상호작용으로 인한 현상

을 볼 수 있을까요?) 이는 놀라운 일이 아닙니다.

다양한 수준에서 금지되는 현상은 다음과 같습니다:

더티 읽기

트랜잭션은 동시 커밋되지 않은 트랜잭션이 쓴 데이터를 읽습니다. 반복할 수 없는

읽기

트랜잭션이 이전에 읽었던 데이터를 다시 읽다가 다른 트랜잭션(최초 읽기 이후 커밋된 트랜잭션)에 의해 데이터가 수정된 것을 발견합니다.

팬텀 읽기

트랜잭션이 검색 조건을 만족하는 행 집합을 반환하는 쿼리를 다시 실행하고 최근에 커밋된 다른 트랜잭션으로 인해 조건을 만족하는 행 집합이 변경된 것을 발견합니다.

직렬화 이상

트랜잭션 그룹을 성공적으로 커밋한 결과는 해당 트랜잭션을 한 번에 하나씩 실행하는 모든 가능한 순서와 일치하지 않습니다.

SQL 표준 및 PostgreSQL로 구현된 트랜잭션 격리 수준은 표 13.1에 설명되어 있습니다.

표 13.1. 트랜잭션 격리 수준

격리 수준	더티 읽기	반복할 수 없는 읽기	팬텀 읽기	직렬화 이상
커밋되지 않은 읽기	허용되지만 PG에서는 허용되지 않음	가능	가능	가능
읽기 커밋	불가능	가능	가능	가능
반복 읽기	불가능	불가능	허용되지만 PG에서는 허용되지 않음	가능
직렬화 가능	불가능	불가능	불가능	불가능

PostgreSQL에서는 네 가지 표준 트랜잭션 격리 수준 중 하나를 요청할 수 있지만 내부적으로는 세 가지 격리 수준만 구현됩니다(즉, PostgreSQL의 읽기 커밋되지 않음 모드는 읽기 커밋된 것처럼 작동합니다). 이는 표준 격리 수준을 PostgreSQL의 다중 버전 동시성 제어 아키텍처에 매핑하는 유일한 합리적인 방법이기 때문입니다.

이 표는 또한 PostgreSQL의 반복 가능 읽기 구현이 팬텀 읽기를 허용하지 않음을 보여줍니다. 이는 SQL 표준에서 특정 격리 수준에서 발생해서는 *안 되는* 예외를 지정하기 때문에 허용되며, 더 높은 수준의 보장이 허용됩니다. 사용 가능한 격리 수준의 동작은 다음 하위 섹션에 자세히 설명되어 있습니다.

트랜잭션의 트랜잭션 격리 수준을 설정하려면 SET TRANSACTION 명령을 사용합니다.

중요

일부 PostgreSQL 데이터 유형과 함수에는 트랜잭션 동작에 관한 특별한 규칙이 있습니다. 특히 시퀀스(따라서 `serial`을 사용하여 선언된 열의 카운터)에 대한 변경 사항은 다른 모든 트랜잭션에 즉시 표시되며, 변경을 수행한 트랜잭션이 중단되는 경우 롤백되지 않습니다. 섹션 9.17 및 섹션 8.1.4를 참조하세요.

13.2.1. 커밋된 격리 수준 읽기

읽기 커밋은 PostgreSQL의 기본 격리 수준입니다. 트랜잭션이 이 격리 수준을 사용하는 경우, SELECT 쿼리(FOR UPDATE/SHARE 절이 없는)는 쿼리가 시작되기 전에 커밋된 데이터만 보고, 쿼리가 실행되는 동안 커밋되지 않은 데이터나 동시 트랜잭션에 의해 커밋된 변경 사항은 보지 못합니다. 사실상 SELECT 쿼리는 쿼리 실행이 시작되는 시점의 데이터베이스 스냅샷을 보게 됩니다. 그러나 SELECT는 아직 커밋되지 않았더라도 자체 트랜잭션 내에서 실행된 이전 업데이트의 효과를 확인합니다. 또한 두 개의 연속된 SELECT 명령이 단일 트랜잭션 내에 있더라도 첫 번째 SELECT가 시작된 후 두 번째 SELECT가 시작되기 전에 다른 트랜잭션이 변경 사항을 커밋하면 서로 다른 데이터를 볼 수 있습니다.

UPDATE, DELETE, SELECT FOR UPDATE 및 SELECT FOR SHARE 명령은 대상 행을 검색하는 측면에서 SELECT와 동일하게 동작하며, 명령 시작 시점에 커밋된 대상 행만 찾습니다. 그러나 이러한 대상 행을 찾을 때 이미 다른 동시 트랜잭션에 의해 업데이트(또는 삭제 또는 잠김)되었을 수 있습니다. 이 경우, 업데이터는 첫 번째 업데이트 트랜잭션이 커밋되거나 롤백될 때까지 기다립니다(아직 진행 중인 경우). 첫 번째 업데이터가 롤백되면 그 효과는 무효화되고 두 번째 업데이터가 원래 발견된 행을 계속 업데이트할 수 있습니다. 첫 번째 업데이터가 커밋하면 두 번째 업데이터는 첫 번째 업데이터가 커밋한 행을 무시합니다.

업데이터가 삭제한 경우, 그렇지 않으면 업데이트된 버전의 행에 작업을 적용하려고 시도합니다. 명령의 검색 조건(WHERE 절)을 다시 평가하여 업데이트된 버전의 행이 여전히 검색 조건과 일치하는지 확인합니다. 일치하는 경우 두 번째 업데이터는 업데이트된 버전의 행을 사용하여 작업을 진행합니다. SELECT FOR UPDATE 및 SELECT FOR SHARE의 경우, 이는 잠겨서 클라이언트에 반환되는 행의 업데이트된 버전임을 의미합니다.

ON 충돌 시 업데이트 절이 있는 INSERT도 비슷하게 작동합니다. 읽기 커밋 모드에서는 삽입을 위해 제안된 각 행이 삽입되거나 업데이트됩니다. 관련 없는 오류가 없는 한, 이 두 가지 결과 중 하나가 보장됩니다. INSERT에 아직 효과가 표시되지 않는 다른 트랜잭션에서 충돌이 발생하는 경우, 일반적으로 해당 행의 버전이 명령에 표시되지 *않더라도* UPDATE 절은 해당 행에 영향을 미칩니다.

ON CONFLICT DO NOTHING 절이 있는 INSERT는 INSERT 스냅샷에 효과가 표시되지 않는 다른 트랜잭션의 결과로 인해 삽입이 연속으로 진행되지 않을 수 있습니다. 다시 말하지만, 이것은 읽기 커밋 모드에서만 발생합니다.

MERGE를 사용하면 INSERT, UPDATE, DELETE 하위 명령의 다양한 조합을 지정할 수 있습니다. INSERT 및 UPDATE 하위 명령이 모두 포함된 MERGE 명령은 ON CONFLICT DO UPDATE 절이 있는 INSERT와 비슷해 보이지만 INSERT 또는 UPDATE가 수행된다는 것을 보장하지는 않습니다. MERGE가 UPDATE 또는 DELETE를 시도하고 행이 동시에 업데이트되지만 현재 대상 및 현재 원본 튜플에 대해 조인 조건이 여전히 통과하는 경우 MERGE는 UPDATE 또는 DELETE 명령과 동일하게 작동하며 업데이트된 버전의 행에 대해 작업을 수행합니다. 그러나 MERGE는 여러 작업을 지정할 수 있고 조건부일 수 있으므로 원래 일치했던 작업이 작업 목록의 나중에 나타나더라도 각 작업의 조건은 첫 번째 작업부터 시작하여 업데이트된 버전의 행에 대해 다시 평가됩니다. 반면에 행이 동시에 업데이트되거나 삭제되어 조인 조건이 실패하는 경우 MERGE는 조건의 NOT MATCHED 작업을 다음에 평가하고 성공한 첫 번째 작업을 실행합니다. MERGE가 INSERT를 시도할 때 고유 인덱스가 존재하고 중복 행이 동시에 삽입되는 경우 고유성 위반 오류가 발생하며, MERGE는 MATCHED 조건의 평가를 다시 시작하여 이러한 오류를 피하려고 시도하지 않습니다.

위의 규칙 때문에 업데이트 명령이 일관되지 않은 스냅샷을 볼 수 있습니다. 업데이트하려는 동일한 행에 대한 동시 업데이트 명령의 효과는 볼 수 있지만 데이터베이스의 다른 행에 대한 해당 명령의 효과는 볼 수 없습니다. 이러한 동작으로 인해 읽기 커밋 모드는 복잡한 검색 조건이 포함된 명령에는 적합하지 않지만, 간단한 경우에는 적합합니다. 예를 들어 다음과 같은 거래로 은행 잔액을 업데이트하는 경우를 생각해 보세요:

```
시작;
업데이트 계정 SET 잔액 = 잔액 + 100.00 WHERE acctnum = 12345;
업데이트 계정 SET 잔액 = 잔액 - 100.00 WHERE acctnum = 7534;
커밋;
```

이러한 두 개의 트랜잭션이 동시에 계정 12345의 잔액을 변경하려고 시도하는 경우, 두 번째 트랜잭

션이 업데이트된 버전의 계정 행으로 시작되기를 원합니다. 각 명령은 미리 정해진 행에만 영향을 미치기 때문에 행의 업데이트된 버전을 확인해도 문제가 되는 불일치가 발생하지 않습니다.

더 복잡한 사용법은 읽기 커밋 모드에서 바람직하지 않은 결과를 초래할 수 있습니다. 예를 들어 웹사이트가 웹사이트.hits가 9와 10인 2행 테이블이고 다른 명령에 의해 제한 기준에서 추가 및 제거되는 데이터에 대해 DELETE 명령이 작동한다고 가정해 보겠습니다:

시작;
웹사이트 업데이트 SET 조회 수 = 조회 수 + 1;
다른 세션에서 실행: DELETEFROM website WHERE hits = 10;
COMMIT;

UPDATE 전후에 website.hits = 10 행이 있어도 DELETE는 아무런 효과가 없습니다. 이는 업데이트 전 행 값 9를 건너뛰고 업데이트가 완료되고 DELETE가 잠금을 획득하면 새 행 값이 더 이상 10이 아니라 11이 되어 더 이상 기준과 일치하지 않기 때문에 발생합니다.

커밋된 읽기 모드는 해당 시점까지 커밋된 모든 트랜잭션을 포함하는 새 스냅샷으로 각 명령을 시작하므로, 동일한 트랜잭션의 후속 명령은 어떤 경우에도 커밋된 동시 트랜잭션의 효과를 볼 수 있습니다. 위의 문제의 핵심은 *단일* 명령이 데이터베이스의 절대적으로 일관된 보기를 볼 수 있는지 여부입니다.

읽기 커밋 모드에서 제공하는 부분 트랜잭션 격리는 많은 애플리케이션에 적합하며 이 모드는 빠르고 사용하기 간편하지만 모든 경우에 충분하지는 않습니다. 복잡한 쿼리 및 업데이트를 수행하는 애플리케이션은 읽기 커밋 모드가 제공하는 것보다 더 엄격하게 일관된 데이터베이스 보기가 필요할 수 있습니다.

13.2.2. 반복 가능한 읽기 격리 수준

반복 읽기 격리 수준은 트랜잭션이 시작되기 전에 커밋된 데이터만 보고, 트랜잭션이 실행되는 동안 커밋되지 않은 데이터나 동시 트랜잭션에 의해 커밋된 변경 사항은 볼 수 없습니다. (단, 각 쿼리는 아직 커밋되지 않았더라도 자체 트랜잭션 내에서 실행된 이전 업데이트의 효과를 볼 수 있습니다.) 이는 이 격리 수준에 대해 SQL 표준에서 요구하는 것보다 더 강력한 보장이며, 직렬화 이상을 제외한 표 13.1에 설명된 모든 현상을 방지합니다. 위에서 언급했듯이, 이는 각 격리 수준이 제공해야 하는 *최소한의* 보호 기능만 설명하는 표준에서 특별히 허용하는 것입니다.

이 수준은 반복 가능한 읽기 트랜잭션의 쿼리가 트랜잭션 내의 현재 문이 시작되는 시점이 아니라 *트랜잭션의* 첫 번째 비 트랜잭션 제어 문이 시작되는 시점의 스냅샷을 본다는 점에서 읽기 커밋과 다릅니다. 따라서 *단일* 트랜잭션 내의 연속적인 SELECT 명령은 동일한 데이터를 보게 되며, 즉 자신의 트랜잭션이 시작된 후에 커밋된 다른 트랜잭션에서 변경된 내용은 볼 수 없습니다.

이 레벨을 사용하는 애플리케이션은 직렬화 실패로 인해 트랜잭션을 다시 시도할 준비가 되어 있어야 합니다.

업데이트, 삭제, 병합, 업데이트 선택 및 공유 선택 명령은 다음과 같습니다.
는 대상 행을 검색한다는 점에서 SELECT와 동일합니다. 즉, 트랜잭션 시작 시점에 커밋된 대상 행만 찾습니다. 그러나 이러한 대상 행을 찾을 때 이미 다른 동시 트랜잭션에 의해 업데이트(또는 삭제 또는 잠김)되었을 수 있습니다. 이 경우 반복 가능한 읽기 트랜잭션은 첫 번째 업데이트 트 랜 잭 션 이 커밋되거나 롤백될 때까지 기다립니다(아직 진행 중인 경우). 첫 번째 업데이터가 롤백되면 그 효과는 무효화되고 반복 가능한 읽기 트랜잭션은 원래 발견된 행을 계속 업데이트할 수 있습니다. 그러나 첫 번째 업데이터가 커밋하면(그리고 단순히 행을 잠근 것이 아니라 실제로 행을 업데이트하거나 삭제하면) 반복 가능한 읽기 트랜잭션은 다음과 같은 메시지와 함께 롤백됩니다.

오류: 동시 업데이트로 인해 액세스를 직렬화할 수 없습니다.

반복 읽기 트랜잭션은 반복 읽기 트랜잭션이 시작된 후 다른 트랜잭션에 의해 변경된 행을 수정하거나 잠글 수 없기 때문입니다.

애플리케이션이 이 오류 메시지를 받으면 현재 트랜잭션을 중단하고 전체 트랜잭션을 처음부터 다시 시도해야 합니다. 두 번째 시도에서는 트랜잭션이 데이터베이스의 초기 보기의 일부로 이전에 커밋된 변경 사항을 보게 되므로 새 버전의 행을 새 트랜잭션 업데이트의 시작점으로 사용하는 데 논리적 충돌이 없습니다.

업데이트 트랜잭션만 다시 시도해야 할 수 있으며 읽기 전용 트랜잭션은 직렬화 충돌이 발생하지 않습니다.

반복 읽기 모드는 각 트랜잭션이 데이터베이스의 완전히 안정적인 보기를 볼 수 있도록 엄격하게 보장합니다. 그러나 이 보기가 항상 일부 직렬

(한 번에 하나씩) 같은 수준의 동시 트랜잭션을 실행합니다. 예를 들어, 이 수준의 읽기 전용 트랜잭션도 배치가 완료되었음을 표시하기 위해 제어 레코드가 업데이트되었지만 제어 레코드의 이전 개정판을 읽었기 때문에 배치의 논리적으로 일부인 세부 레코드 중 하나는 볼 수 *없습니다*. 이 격리 수준에서 실행되는 트랜잭션으로 비즈니스 규칙을 적용하려는 시도는 충돌하는 트랜잭션을 차단하기 위해 명시적 잠금을 신중하게 사용하지 않으면 제대로 작동하지 않을 가능성이 높습니다.

반복 읽기 격리 수준은 학술 데이터베이스 문헌과 일부 다른 데이터베이스 제품에서 *스냅샷 격리로* 알려진 기술을 사용하여 구현됩니다. 동시성을 줄이는 기존 잠금 기술을 사용하는 시스템과 비교할 때 동작 및 성능의 차이가 관찰될 수 있습니다. 일부 다른 시스템에서는 반복 읽기 및 스냅샷 격리를 서로 다른 동작을 가진 별개의 격리 수준으로 제공할 수도 있습니다. 이 두 기술을 구분하는 허용되는 현상은 SQL 표준이 개발된 후에야 데이터베이스 연구자들에 의해 공식화되었으며, 이 매뉴얼의 범위를 벗어납니다. 자세한 내용은 [berenson95]를 참조하세요.

> **참고**
>
> PostgreSQL 버전 9.1 이전에는 직렬화 가능 트랜잭션 격리 수준에 대한 요청이 여기에 설명된 것과 똑같은 동작을 제공했습니다. 기존 직렬화 가능 동작을 유지하려면 이제 반복 가능한 읽기를 요청해야 합니다.

13.2.3. 직렬화 가능한 격리 수준

직렬화 가능 격리 수준은 가장 엄격한 트랜잭션 격리를 제공합니다. 이 수준은 커밋된 모든 트랜잭션에 대해 직렬 트랜잭션 실행을 에뮬레이션하며, 마치 트랜잭션이 동시에 실행되지 않고 순차적으로 차례로 실행되는 것처럼 보입니다. 그러나 반복 읽기 수준과 마찬가지로 이 수준을 사용하는 애플리케이션은 직렬화 실패로 인해 트랜잭션을 다시 시도할 수 있도록 준비해야 합니다. 실제로 이 격리 수준은 반복 읽기 수준과 완전히 동일하게 작동하지만, 직렬화 가능한 트랜잭션의 동시 실행이 가능한 모든 직렬(한 번에 하나씩) 실행과 일치하지 않는 방식으로 동작할 수 있는 조건도 모니터링합니다. 이 모니터링은 반복 가능한 읽기에 존재하는 것 이상의 차단을 도입하지는 않지만 모니터링에 약간의 오버헤드가 있으며 *직렬화 이상을* 유발할 수 있는 조건이 감지되면 *직렬화 오류가* 트리거됩니다.

예를 들어, 처음에 다음을 포함하는 테이블 `mytab`을 생각해 보겠습니다:

```
 클래스 | 값
-------+-------
     1 |    10
     1 |    20
     2 |   100
     2 |   200
```

직렬화 가능한 트랜잭션 A가 계산된다고 가정해 보겠습니다:

```
SELECT SUM(value) FROM mytab WHERE class = 1;
```

를 계산한 다음 결과(30)를 클래스 = 2의 새 행에 값으로 삽입합니다. 동시에 직렬화 가능한 트랜잭션 B가 계산됩니다:

```
SELECT SUM(value) FROM mytab WHERE class = 2;
```

를 호출하여 결과 300을 얻고 클래스 = 1의 새 행에 삽입합니다. 그런 다음 두 트랜잭션 모두 커밋을 시도합니다. 두 트랜잭션 중 하나가 반복 읽기 격리 수준에서 실행 중이라면 둘 다 커밋이 허용되지만 결과와 일치하는 직렬 실행 순서가 없으므로

직렬화 가능한 트랜잭션은 한 트랜잭션이 커밋되고 이 메시지와 함께 다른 트랜잭션이 롤백되도록 허용합니다:

오류: 트랜잭션 간의 읽기/쓰기 종속성으로 인해 액세스를 직렬화할 수 없습니다.

A가 B보다 먼저 실행했다면 B는 300이 아닌 330을 계산했을 것이고, 마찬가지로 다른 주문도 A가 계산한 합계가 달라졌을 것이기 때문입니다.

이상 현상을 방지하기 위해 직렬화 가능 트랜잭션에 의존하는 경우, 영구 사용자 테이블에서 읽은 모든 데이터는 이를 읽은 트랜잭션이 성공적으로 커밋될 때까지 유효한 것으로 간주하지 않는 것이 중요합니다. 이는 읽기 전용 트랜잭션의 경우에도 마찬가지이지만, *지연 가능한* 읽기 전용 트랜잭션 내에서 읽은 데이터는 데이터를 읽기 시작하기 전에 이러한 문제가 없는 것으로 보장된 스냅샷을 얻을 수 있을 때까지 기다리기 때문에 읽은 즉시 유효한 것으로 알려져 있다는 점을 제외하면, 읽기 전용 트랜잭션의 경우에도 마찬가지입니다. 다른 모든 경우에는 애플리케이션은 나중에 중단된 트랜잭션 중에 읽은 결과에 의존해서는 안 되며, 트랜잭션이 성공할 때까지 트랜잭션을 다시 시도해야 합니다.

진정한 직렬화 가능성을 보장하기 위해 PostgreSQL은 조건부 *잠금을* 사용하는데, 이는 쓰기가 먼저 실행되었다면 동시 트랜잭션의 이전 읽기 결과에 영향을 미칠 수 있는 시점을 판단할 수 있는 잠금을 유지한다는 의미입니다. PostgreSQL에서 이러한 잠금은 어떠한 차단도 일으키지 않으므로 교착 상태를 일으키는 데 아무런 역할을 할 수 *없습니다*. 이러한 잠금은 특정 조합에서 직렬화 이상을 초래할 수 있는 동시 직렬화 가능 트랜잭션 간의 종속성을 식별하고 플래그를 지정하는 데 사용됩니다. 반대로 데이터 일관성을 보장하려는 읽기 커밋 또는 반복 읽기 트랜잭션은 전체 테이블에 대한 잠금을 해제해야 하므로 해당 테이블을 사용하려는 다른 사용자가 차단될 수 있고, 다른 트랜잭션을 차단할 뿐만 아니라 디스크 액세스를 유발할 수 있는 `SELECT FOR UPDATE` 또는 `SELECT FOR SHARE`를 사용할 수도 있습니다.

대부분의 다른 데이터베이스 시스템과 마찬가지로 PostgreSQL의 술어 잠금은 트랜잭션에서 실제로 액세스한 데이터를 기반으로 합니다. 이러한 잠금은 `pg_locks` 시스템 보기에 `SIReadLock` 모드와 함께 표시됩니다. 쿼리 실행 중에 획득되는 특정 잠금은 쿼리가 사용하는 계획에 따라 달라지며, 잠금을 추적하는 데 사용되는 메모리의 고갈을 방지하기 위해 여러 개의 세분화된 잠금(예: 튜플 잠금)이 트랜잭션 진행 중에 더 적은 수의 세분화된 잠금(예: 페이지 잠금)으로 결합될 수 있습니다. 읽기 전용 트랜잭션은 직렬화 이상을 초래할 수 있는 충돌이 여전히 발생하지 않는다고 판단되면 완료 전에 SIRead 잠금을 해제할 수 있습니다. 실제로 읽기 전용 트랜잭션은 시작 시 이 사실을 확인하고 술어 잠금을 취하지 않을 수 있는 경우가 많습니다. 명시적으로 직렬화 가능한 읽기 전용 거부 트랜잭션을 요청하면 이 사실을 확인할 수 있을 때까지 차단됩니다. (이 경우는 직렬화 가능 트랜잭션은 차단되지만 반복 읽기 트랜잭션은 차단되지 않는 *유일한* 경우입니다.) 반면에 SIRead 잠금은 중복 읽기 쓰기 트랜잭션이 완료될 때까지 트랜잭션 커밋 이후에도 유지되어야 하는 경우가 많습니

다.

직렬화 가능한 트랜잭션을 일관되게 사용하면 개발을 간소화할 수 있습니다. 성공적으로 커밋된 동시 직렬화 가능 트랜잭션 집합이 한 번에 하나씩 실행되는 것과 동일한 효과를 갖는다는 보장은 단일 트랜잭션이 작성된 대로 단독으로 실행될 때 올바른 작업을 수행한다는 것을 입증할 수 있다면, 다른 트랜잭션이 어떤 작업을 수행하거나 성공적으로 커밋되지 않을지에 대한 정보가 없어도 직렬화 가능 트랜잭션이 혼합된 경우에도 올바른 작업을 수행한다는 확신을 가질 수 있다는 것을 의미합니다. 이 기술을 사용하는 환경에서는 읽기/쓰기 종속성에 기여할 수 있는 트랜잭션을 정확히 예측하기가 매우 어렵고 직렬화 이상을 방지하기 위해 롤백해야 하는 직렬화 실패(항상 SQLSTATE 값이 '40001'로 반환됨)를 처리하는 일반화된 방법을 갖는 것이 중요합니다. 읽기/쓰기 종속성 모니터링에는 직렬화 실패로 종료된 트랜잭션의 재시작과 마찬가지로 비용이 들지만, 명시적 잠금 및 `SELECT FOR UPDATE` 또는 `SELECT FOR SHARE` 사용과 관련된 비용 및 차단과 균형을 맞추면 직렬화 가능한 트랜잭션이 일부 환경에서는 최상의 성능 선택입니다.

PostgreSQL의 직렬화 가능 트랜잭션 격리 수준은 동일한 효과를 생성하는 직렬 실행 순서가 있음을 증명할 수 있는 경우에만 동시 트랜잭션의 커밋을 허용하지만, 실제 직렬 실행에서는 발생하지 않는 오류 발생을 항상 방지하지는 못합니다. 특히, 겹치는 Serializable과의 충돌로 인해 발생하는 고유한 제약 조건 위반을 볼 수 있습니다.

트랜잭션을 삽입하려고 시도하기 전에 명시적으로 키가 없는지 확인한 후에도 충돌이 발생할 수 있습니다. 잠재적으로 충돌할 수 있는 키를 삽입하는 *모든* 직렬화 가능 트랜잭션이 먼저 삽입할 수 있는지 명시적으로 확인하도록 하면 이러한 문제를 방지할 수 있습니다. 예를 들어, 사용자에게 새 키를 요청한 다음 먼저 키를 선택하려고 시도하여 이미 존재하지 않는지 확인하거나 최대 기존 키를 선택하고 키를 추가하여 새 키를 생성하는 애플리케이션을 상상해 보세요. 일부 직렬화 가능 트랜잭션이 이 프로토콜을 따르지 않고 직접 새 키를 삽입하면 동시 트랜잭션의 직렬 실행에서는 발생할 수 없는 경우에도 고유 제약 조건 위반이 보고될 수 있습니다.

동시성 제어를 위해 직렬화 가능한 트랜잭션에 의존할 때 최적의 성능을 얻으려면 이러한 문제를 고려해야 합니다:

- 가능하면 트랜잭션을 읽기 전용으로 선언하세요.
- 필요한 경우 연결 풀을 사용하여 활성 연결 수를 제어합니다. 이는 항상 중요한 성능 고려 사항이지만 직렬화 가능 트랜잭션을 사용하는 바쁜 시스템에서는 특히 중요할 수 있습니다.
- 무결성을 위해 단일 거래에 필요 이상의 금액을 입력하지 마세요.
- 연결을 필요 이상으로 오래 '트랜잭션 유휴 상태'로 두지 마세요. 구성 매개변수 idle_in_transaction_session_timeout을 사용하여 대기 중인 세션의 연결을 자동으로 끊을 수 있습니다.
- 직렬화 가능한 트랜잭션이 자동으로 제공하는 보호 기능으로 인해 더 이상 필요하지 않은 경우 명시적 잠금, 업데이트 선택, 공유 선택을 제거합니다.
- 프리케이트 잠금 테이블의 메모리가 부족하여 시스템에서 여러 페이지 수준 프리케이트 잠금을 단일 관계 수준 프리케이트 잠금으로 결합해야 하는 경우 시뮬레이션 실패율이 증가할 수 있습니다. 트랜잭션당 max_pred_locks, 관계당 max_pred_locks 및/또는 페이지당 max_pred_locks를 늘려서 이 문제를 방지할 수 있습니다.
- 순차 스캔에는 항상 관계 수준 술어 잠금이 필요합니다. 이로 인해 직렬화 실패율이 증가할 수 있습니다. 무작위 페이지 비용을 줄이거나 cpu_tuple_cost를 늘려 인덱스 스캔을 사용하도록 권장하는 것이 도움이 될 수 있습니다. 트랜잭션 롤백 및 재시작의 감소를 쿼리 실행 시간의 전반적인 변화와 비교하여 고려해야 합니다.

직렬화 가능 격리 수준은 학술 데이터베이스 문헌에서 직렬화 가능 스냅샷 격리로 알려진 기술을 사용하여 구현되며, 이는 스냅샷 격리를 기반으로 직렬화 이상에 대한 검사를 추가하여 구축됩니다. 기존 잠금 기술을 사용하는 다른 시스템과 비교할 때 동작과 성능에 약간의 차이가 있을 수 있습니다. 자세한 내용은 [ports12]를 참조하세요.

13.3. 명시적 잠금

PostgreSQL은 테이블의 데이터에 대한 동시 액세스를 제어하기 위한 다양한 잠금 모드를 제공합니다. 이러한 모드는 MVCC가 원하는 동작을 제공하지 않는 상황에서 애플리케이션 제어 잠금을 위해 사용할 수 있습니다. 또한 대부분의 PostgreSQL 명령은 자동으로 적절한 모드의 잠금을 획득하여 명령이 실행되는 동안 참조된 테이블이 호환되지 않는 방식으로 삭제되거나 수정되지 않도록 합니다. (예를 들어, `TRUNCATE`는 동일한 테이블에서 다른 작업과 동시에 안전하게 실행될 수 없으므로 테이블에 `ACCESS EXCLUSIVE` 잠금을 획득하여 이를 강제합니다.)

데이터베이스 서버에서 현재 미결 상태인 잠금 목록을 확인하려면 `pg_locks` 시스템 보기를 사용합니다. 잠금 관리자 하위 시스템의 상태 모니터링에 대한 자세한 내용은 28장을 참조하세요.

13.3.1. 테이블 수준 잠금

아래 목록에는 사용 가능한 잠금 모드와 PostgreSQL에서 자동으로 사용되는 컨텍스트가 나와 있습니다. LOCK 명령을 사용하여 이러한 잠금을 명시적으로 획득할 수도 있습니다. 기억하세요

이름에 "행"이라는 단어가 포함되어 있더라도 이러한 잠금 모드는 모두 테이블 수준 잠금이며, 잠금 모드의 이름은 과거에 만들어진 것입니다. 잠금 모드의 이름은 각 잠금 모드의 일반적인 사용법을 어느 정도 반영합니다.

- 의 의미는 모두 동일합니다. 한 잠금 모드와 다른 잠금 모드의 유일한 차이점은 각각 충돌하는 잠금 모드의 집합입니다(표 13.2 참조). 두 트랜잭션은 같은 테이블에서 동시에 충돌하는 모드의 잠금을 보유할 수 없습니다. (그러나 트랜잭션은 절대로 자신과 충돌하지 않습니다. 예를 들어, 동일한 테이블에서 액세스 독점 잠금을 획득한 후 나중에 액세스 공유 잠금을 획득할 수 있습니다). 충돌하지 않는 잠금 모드는 여러 트랜잭션이 동시에 보유할 수 있습니다. 특히 일부 잠금 모드는 자체 충돌하는 반면(예: 한 번에 두 개 이상의 트랜잭션이 액세스 독점 잠금을 보유할 수 없음), 다른 잠금 모드는 자체 충돌하지 않습니다(예: 여러 트랜잭션이 액세스 공유 잠금을보유할 수 있음).

테이블 수준 잠금 모드

액세스 공유(액세스셰어락)

액세스 독점 잠금 모드와만 충돌합니다.

SELECT 명령은 참조된 테이블에서 이 모드의 잠금을 획득합니다. 일반적으로 테이블을 *읽기만* 하고 수정하지 않는 모든 쿼리는 이 잠금 모드를 획득합니다.

행 공유(행 공유 잠금)

독점 및 액세스 독점 잠금 모드와 충돌합니다.

SELECT 명령은 명시적인 FOR ... 잠금 옵션 없이 참조되는 다른 모든 테이블에 대한 액세스 공유 잠금에 추가하여 FOR UPDATE, FOR NO KEY UPDATE, FOR SHARE 또는 FOR KEY SHARE 옵션 중 하나가 지정된 모든 테이블에서 이 모드의 잠금을 획득합니다(명시적인 FOR ... 잠금 옵션 없이 참조되는 모든 테이블에 대한 액세스 공유 잠금 포함).

행 독점(행 독점 잠금)

공유, 공유 행 독점, 독점 및 액세스 EX-와 충돌합니다.
폐쇄형 잠금 모드.

업데이트, 삭제, 삽입 및 병합 명령은 대상 테이블에서 이 잠금 모드를 획득합니다(참조된 다른 테이블의 액세스 공유 잠금과 더불어). 일반적으로 이 잠금 모드는 테이블의 *데이터를 수정하는* 모든 명령에 의해 획득됩니다.

업데이트 독점 공유(ShareUpdateExclusiveLock)

공유 업데이트 독점, 공유, 공유 행 독점, EX-와 충돌합니다.
단독 및 액세스 독점 잠금 모드입니다. 이 모드는 동시 임대 스키마 변경 및 VACUUM 실행으로부터 테이블을 보호합니다.

VACUUM(FULL 제외), 분석, 동시 인덱스 생성, 통계 작성, 댓글 달기, 동시 인덱스 재인덱스

및 특정 인덱스 변경으로 획득
및 ALTER TABLE 변형(자세한 내용은 해당 명령의 설명서를 참조하세요).

SHARE(쉐어락)

행 독점, 업데이트 독점 공유, 행 독점 공유와 충돌합니다.
포괄적, 독점 및 액세스 독점 잠금 모드. 이 모드는 동시 데이터 변경으로부터 테이블을 보호합니다.

인덱스 생성 동시 없이)으로 획득합니다. 행 독점 공유

(ShareRowExclusiveLock)

행 독점, 업데이트 독점 공유, 공유, 행 공유와 충돌합니다.
독점, 독점 및 액세스 독점 잠금 모드. 이 모드는 동시 데이터 변경으로부터 테이블을 보호하며, 한 번에 한 세션만 테이블을 보유할 수 있도록 자체 배타적입니다.

생성 트리거 및 일부 형태의 테이블 변경으로 획득합니다. EXCLUSIVE(독점

잠금)

행 공유, 행 독점, 업데이트 독점 공유, 공유, 행 독점 공유, 독점 및 액세스 독점 잠금 모드와 충돌합니다. 이 모드
는 동시 액세스 공유 잠금만 허용합니다. 즉, 이 잠금 모드가 유지되는 트랜잭션에서는 테이블 읽기만 병렬로 진행할 수 있습니다.

자료 새로 고침 보기에서 동시에 획득했습니다.

액세스 독점(액세스 독점 잠금)

모든 모드(액세스 공유, 행 공유, 행 독점, 공유 업데이트 독점, 공유, 공유 행 독점, 독점 및 액세스 EX-) 의 잠금과 충돌합니다.
CLUSIVE). 이 모드는 소유자가 어떤 방식으로든 테이블에 액세스하는 유일한 트랜잭션임을 보장합니다.

테이블 삭제, 잘라내기, 다시 색인, 클러스터, 빈 공간 가득 차기 및 다시
새로 자료화된 보기(동시 사용 안 함) 명령. 많은 형태의 AL- TER INDEX 및 ALTER TABLE도 이 수준에서 잠금을 획득합니다. 또한 모드를 명시적으로 지정하지 않는 LOCK TABLE 문의 기본 잠금 모드이기도 합니다.

> **팁**
>
> 액세스 독점 잠금만 SELECT (FOR UPDATE/SHARE 없이) 문을 차단합니다.

일단 획득한 잠금은 일반적으로 트랜잭션이 끝날 때까지 유지됩니다. 그러나 저장점을 설정한 후 잠금이 획득된 경우, 저장점을 롤백하면 잠금이 즉시 해제됩니다. 이는 롤백이 저장점 이후 명령의 모든 효과를 취소한다는 원칙과 일치합니다. PL/pgSQL 예외 블록 내에서 획득한 잠금도 마찬가지입니다. 블록에서 오류 이스케이프가 발생하면 블록 내에서 획득한 잠금이 해제됩니다.

표 13.2. 충돌하는 잠금 모드

요청된 잠금 모드	기존 잠금 모드							
	액세스 공유	행 공유	ROW EXCL.	공유 업데이트 제외.	공유	공유 행 제외.	EXCL.	액세스 EXCL.
액세스 공유								X

행 공유							X	X
ROW EXCL.					X	X	X	X
공유 업데이트 제외.				X	X	X	X	X
공유			X	X		X	X	X
공유 행 ex-cl.			X	X	X	X	X	X

요청된 잠금 모드	기존 잠금 모드							
	액세스 공유	행 공유	ROW EXCL.	공유 업데이트 제외.	공유	공유 행 제외.	EXCL.	액세스 EXCL.
EXCL.		X	X	X	X	X	X	X
액세스 EXCL.	X	X	X	X	X	X	X	X

13.3.2. 행 수준 잠금

테이블 수준 잠금 외에도 행 수준 잠금이 있으며, PostgreSQL에서 자동으로 사용되는 컨텍스트와 함께 아래와 같이 나열되어 있습니다. 행 수준 잠금 충돌에 대한 전체 표는 표 13.3을 참조하세요. 트랜잭션은 서로 다른 하위 트랜잭션에서도 동일한 행에서 충돌하는 잠금을 보유할 수 있지만, 그 외에는 두 트랜잭션이 동일한 행에서 충돌하는 잠금을 보유할 수 없습니다. 행 수준 잠금은 데이터 쿼리에는 영향을 미치지 않으며, 같은 행에 대한 *쓰기와 잠금만* 차단합니다. 행 수준 잠금은 테이블 수준 잠금과 마찬가지로 트랜잭션 종료 시 또는 저장점 롤백 중에 해제됩니다.

행 수준 잠금 모드

업데이트

FOR UPDATE를 사용하면 SELECT 문에서 검색된 행이 마치 업데이트인 것처럼 잠깁니다. 이렇게 하면 현재 트랜잭션이 종료될 때까지 다른 트랜잭션에 의해 잠기거나 수정 또는 삭제되는 것을 방지할 수 있습니다. 즉, UPDATE, DELETE, SELECT FOR UPDATE, SELECT FOR NO KEY UPDATE, SELECT FOR SHARE 또는 SELECT FOR
현재 트랜잭션이 종료될 때까지 이러한 행의 KEY SHARE는 차단되며, 반대로 SELECT FOR UPDATE는 동일한 행에서 이러한 명령을 실행한 동시 트랜잭션을 기다린 다음 업데이트된 행(또는 행이 삭제된 경우 행이 없는 경우)을 잠그고 반환합니다. 그러나 반복 가능한 읽기 또는 직렬화 가능 트랜잭션 내에서는 트랜잭션이 시작된 이후 잠글 행이 변경된 경우 오류가 발생합니다. 자세한 내용은 섹션 13.4를 참조하세요.

FOR UPDATE 잠금 모드는 행의 모든 DELETE와 특정 열의 값을 수정하는 UPDATE에 의해서도 획득됩니다. 현재 UPDATE 경우에 고려되는 열 집합은 외래 키에 사용할 수 있는 고유 인덱스가 있는 열이지만(따라서 부분 인덱스 및 표현식 인덱스는 고려되지 않음), 향후 변경될 수 있습니다.

키 업데이트가 없는 경우

획득하는 잠금이 더 약하다는 점을 제외하면 FOR UPDATE와 유사하게 동작합니다. 이 잠금은 동일한 행에 대한 잠금을 획득하려는 SELECT FOR KEY SHARE 명령을 차단하지 않습니다.

이 잠금 모드는 FOR UPDATE 잠금을 획득하지 않는 모든 UPDATE에 의해서도 획득됩니다.

공유

검색된 각 행에 대해 독점 잠금이 아닌 공유 잠금을 획득한다는 점을 제외하면 FOR NO KEY UPDATE와 유사하게 작동합니다. 공유 잠금은 다른 트랜잭션이 이러한 행에 대해 업데이트, 삭제, SELECT FOR UPDATE 또는 SELECT FOR NO KEY UPDATE를 수행하지 못하도록 차단합니다.
행을 사용할 수 있지만 SELECT FOR SHARE 또는 SELECT FOR KEY SHARE를 수행하는 것을 막지는 못합니다.

주요 공유

잠금이 더 약하다는 점을 제외하면 FOR SHARE와 유사하게 동작합니다: SELECT FOR UPDATE는 차단되지만 SELECT FOR NO KEY UPDATE는 차단되지 않습니다. 키 공유 잠금은 다른 트랜잭션이 DELETE 또는 키 값을 변경하는 모든 UPDATE를 수행하지 못하도록 차단하지만 다른 UPDATE는 차단하지 않으며, SELECT FOR NO KEY UPDATE, SELECT FOR SHARE 또는 SELECT FOR KEY SHARE도 차단하지 못합니다.

PostgreSQL은 메모리에 수정된 행에 대한 정보를 기억하지 않으므로 한 번에 잠글 수 있는 행 수에는 제한이 없습니다. 그러나 행을 잠그면 디스크 쓰기가 발생할 수 있습니다(예: `SELECT FOR UPDATE`는 선택한 행을 수정하여 잠긴 것으로 표시하므로 디스크 쓰기가 발생할 수 있습니다).

표 13.3. 충돌하는 행 수준 잠금

요청된 잠금 모드	현재 잠금 모드			
	주요 공유	공유	키 업데이트가 없는 경우	업데이트
주요 공유				X
공유			X	X
키 업데이트가 없는 경우		X	X	X
업데이트	X	X	X	X

13.3.3. 페이지 수준 잠금

테이블 및 행 잠금 외에도 페이지 수준 공유/독점 잠금은 공유 버퍼 풀의 테이블 페이지에 대한 읽기/쓰기 액세스를 제어하는 데 사용됩니다. 이러한 잠금은 행을 가져오거나 업데이트하면 즉시 해제됩니다. 애플리케이션 개발자는 일반적으로 페이지 수준 잠금에 대해 신경 쓸 필요가 없지만, 여기서는 완전성을 위해 언급합니다.

13.3.4. 교착 상태

명시적 잠금을 사용하면 두 개 이상의 트랜잭션이 각각 상대방이 원하는 잠금을 보유하는 *교착 상태가 발생할* 가능성이 높아질 수 있습니다. 예를 들어 트랜잭션 1이 테이블 A에 대한 독점 잠금을 획득한 후 테이블 B에 대한 독점 잠금을 획득하려고 시도하는 반면, 트랜잭션 2는 이미 테이블 B를 독점 잠금한 후 테이블 A에 대한 독점 잠금을 원한다면 둘 다 진행할 수 없습니다. PostgreSQL은 교착 상태를 자동으로 감지하여 관련된 트랜잭션 중 하나를 중단하고 다른 트랜잭션이 완료될 수 있도록 함으로써 이를 해결합니다. (정확히 어떤 트랜잭션이 중단될지는 예측하기 어려우므로 의존해서는 안 됩니다.)

교착 상태는 행 수준 잠금의 결과로도 발생할 수 있습니다(따라서 명시적 잠금을 사용하지 않더라도 발생할 수 있습니다). 두 개의 동시 트랜잭션이 테이블을 수정하는 경우를 생각해 보겠습니다. 첫 번째 트랜잭션이 실행됩니다:

```
업데이트 계정 SET 잔액 = 잔액 + 100.00 WHERE acctnum = 11111;
```

이렇게 하면 지정된 계정 번호가 있는 행에 행 수준 잠금이 설정됩니다. 그런 다음 두 번째 트랜잭션이 실행됩니다:

```
업데이트 계정 SET 잔액 = 잔액 + 100.00 WHERE acctnum = 22222;
업데이트 계정 SET 잔액 = 잔액 - 100.00 WHERE acctnum = 11111;
```

첫 번째 `UPDATE` 문은 지정된 행에 대한 행 수준 잠금을 성공적으로 획득하여 해당 행을 업데이트하는 데 성공합니다. 그러나 두 번째 `UPDATE` 문은 업데이트하려는 행이 이미 잠겨 있는 것을 발견하고 잠금을 획득한 트랜잭션이 완료될 때까지 기다립니다. 이제 두 번째 트랜잭션은 실행을 계속하기 전에 첫 번째 트랜잭션이 완료되기를 기다리고 있습니다. 이제 트랜잭션 1이 실행됩니다:

```
업데이트 계정 SET 잔액 = 잔액 - 100.00 WHERE acctnum = 22222;
```

트랜잭션 1은 지정된 행에 대한 행 수준 잠금을 획득하려고 시도하지만, 트랜잭션 2가 이미 그러한 잠금을 보유하고 있기 때문에 획득할 수 없습니다. 그래서 트랜잭션 2가 완료될 때까지 기다립니다. 따라서 트랜잭션 1은 트랜잭션 2에서 차단되고, 트랜잭션 2는 트랜잭션 1에서 차단되는 교착 상태가 발생합니다. PostgreSQL은 이 상황을 감지하고 트랜잭션 중 하나를 중단합니다.

교착 상태에 대한 최선의 방어책은 일반적으로 데이터베이스를 사용하는 모든 애플리케이션이 일관된 순서로 여러 개체에 대한 잠금을 획득하도록 하여 교착 상태를 방지하는 것입니다. 위의 예에서 두 트랜잭션이 동일한 순서로 행을 업데이트했다면 교착 상태가 발생하지 않았을 것입니다. 또한 트랜잭션에서 오브젝트에 대해 획득한 첫 번째 잠금이 해당 오브젝트에 필요한 가장 제한적인 모드인지 확인해야 합니다. 이를 미리 확인할 수 없는 경우 교착 상태로 인해 중단되는 트랜잭션을 다시 시도하여 교착 상태를 즉석에서 처리할 수 있습니다.

교착 상태가 감지되지 않는 한, 테이블 수준 또는 행 수준 잠금을 원하는 트랜잭션은 충돌하는 잠금이 해제될 때까지 무기한 대기하게 됩니다. 즉, 애플리케이션이 사용자 입력을 기다리는 동안 트랜잭션을 장시간 열어두는 것은 좋지 않습니다(예: 사용자 입력을 기다리는 동안).

13.3.5. 자문 잠금

PostgreSQL은 애플리케이션이 정의한 의미를 갖는 잠금을 생성하는 수단을 제공합니다. 이를 *자문 잠금*이라고 하는데, 시스템에서 사용을 강제하지 않으며 올바르게 사용하는 것은 애플리케이션에 달려 있기 때문입니다. 자문 잠금은 MVCC 모델에 적합하지 않은 잠금 전략에 유용할 수 있습니다. 예를 들어, 자문 잠금의 일반적인 용도는 소위 '플랫 파일' 데이터 관리 시스템의 전형적인 비관적 잠금 전략을 에뮬레이션하는 것입니다. 테이블에 저장된 플래그를 같은 목적으로 사용할 수도 있지만, 자문 잠금은 더 빠르고 테이블 부풀림을 방지하며 세션이 끝날 때 서버에서 자동으로 정리합니다.

PostgreSQL에서 자문 잠금을 획득하는 방법에는 세션 수준 또는 트랜잭션 수준의 두 가지가 있습니다. 세션 수준에서 획득한 자문 잠금은 명시적으로 해제되거나 세션이 종료될 때까지 유지됩니다. 표준 잠금 요청과 달리 세션 수준 자문 잠금 요청은 트랜잭션 의미를 따르지 않습니다. 나중에 롤백되는 트랜잭션 중에 획득한 잠금은 롤백 후에도 계속 유지되며, 마찬가지로 잠금 해제도 나중에 호출 트랜잭션이 실패하더라도 유효합니다. 잠금은 소유 프로세스에 의해 여러 번 획득할 수 있으며, 완료된 잠금 요청마다 해당 잠금 해제 요청이 있어야 잠금이 실제로 해제됩니다. 반면 트랜잭션 수준의 잠금 요청은 일반 잠금 요청과 비슷하게 작동하며, 트랜잭션이 끝나면 자동으로 해제되고 명시적인 잠금 해제 작업이 필요하지 않습니다. 이 동작은 자문 잠금을 단기간 사용하는 경우 세션 수준 동작보다 더 편리한 경우가 많습니다. 동일한 자문 잠금 식별자에 대한 세션 수준 및 트랜잭션 수준 잠금 요청은 예상되는 방식으로 서로를 차단합니다. 세션이 이미 특정 자문 잠금을 보유하고 있는 경우 다른 세션이 잠금을 대기 중이더라도 해당 세션의 추가 요청은 항상 성공하며, 이는 기존 잠금 보유와 새 요청이 세션 수준인지 트랜잭션 수준인지에 관계없이 해당됩니다.

PostgreSQL의 모든 잠금과 마찬가지로, 현재 모든 세션이 보유한 자문 잠금의 전체 목록은

pg_locks 시스템 보기에서 확인할 수 있습니다.

권고 잠금과 일반 잠금 모두 공유 메모리 풀에 저장되며, 크기는 구성 변수 max_locks_per_transaction 및 max_connections에 의해 정의됩니다. 이 메모리를 모두 소진하지 않도록 주의해야 하며, 그렇지 않으면 서버에서 잠금을 전혀 부여할 수 없게 됩니다. 이렇게 하면 서버가 부여할 수 있는 권고 잠금의 수에 상한선이 설정되며, 일반적으로 서버 구성 방식에 따라 수만에서 수십만 개까지 설정됩니다.

권고 잠금 방법을 사용하는 특정 경우, 특히 명시적 순서 지정 및 LIMIT 절이 포함된 쿼리에서는 SQL 식이 평가되는 순서 때문에 획득한 잠금을 제어하는 데 주의를 기울여야 합니다. 예를 들어

```
SELECT pg_advisory_lock(id) FROM foo WHERE id = 12345; -- ok
SELECT pg_advisory_lock(id) FROM foo WHERE id > 12345 LIMIT 100; -- 위
 험!
SELECT pg_advisory_lock(q.id) FROM
(
```

```
  SELECT id FROM foo WHERE id > 12345 LIMIT 100
) q; -- 확인
```

위의 쿼리에서 두 번째 형태는 잠금 함수가 실행되기 전에 LIMIT가 적용된다는 보장이 없기 때문에 위험합니다. 이로 인해 애플리케이션이 예상하지 못한 일부 잠금이 획득되어 세션이 종료될 때까지 해제되지 않을 수 있습니다. 애플리케이션의 관점에서는 이러한 잠금이 매달려 있지만 여전히 `pg_locks`에서 볼 수 있습니다.

권고 잠금을 조작하기 위해 제공되는 기능은 섹션 9.27.10에 설명되어 있습니다.

13.4. 애플리케이션 수준에서의 데이터 일관성 검사

읽기 커밋 트랜잭션을 사용하여 데이터 무결성에 관한 비즈니스 규칙을 적용하는 것은 매우 어렵습니다. 왜냐하면 데이터의 보기가 각 문마다 바뀌고, 쓰기 충돌이 발생하면 단일 문이라도 그 문에 대한 스냅샷으로 제한되지 않을 수 있기 때문입니다.

반복 읽기 트랜잭션은 실행 내내 데이터를 안정적으로 볼 수 있지만, 데이터 일관성 검사를 위해 MVCC 스냅샷을 사용할 때 *읽기/쓰기 충돌이라는* 미묘한 문제가 있습니다. 한 트랜잭션이 데이터를 쓰고 동시 트랜잭션이 동일한 데이터를 읽으려고 시도하는 경우(쓰기 전이든 후든) 다른 트랜잭션의 작업을 볼 수 없습니다. 그러면 어느 쪽이 먼저 시작했는지 또는 어느 쪽이 먼저 커밋했는지에 관계없이 리더가 먼저 실행한 것으로 나타납니다. 여기까지는 문제가 없지만, 리더가 동시 트랜잭션에서 읽은 데이터를 쓰면 앞서 언급한 트랜잭션 중 하나보다 먼저 실행된 것으로 보이는 트랜잭션이 생깁니다. 마지막으로 실행된 것으로 보이는 트랜잭션이 실제로는 먼저 커밋되는 경우, 트랜잭션 실행 순서를 나타내는 그래프에 사이클이 나타나기 쉽습니다. 이러한 사이클이 나타나면 무결성 검사는 약간의 도움 없이는 제대로 작동하지 않습니다.

섹션 13.2.3에서 언급했듯이 직렬화 가능 트랜잭션은 반복 가능한 읽기 트랜잭션으로, 읽기/쓰기 충돌의 위험한 패턴에 대한 비차단 모니터링을 추가합니다. 명백한 실행 순서에서 사이클을 유발할 수 있는 패턴이 감지되면 관련된 트랜잭션 중 하나를 롤백하여 사이클을 중단합니다.

13.4.1. 직렬화 가능한 트랜잭션으로 일관성 적용하기

데이터의 일관된 보기가 필요한 모든 쓰기와 모든 읽기에 직렬화 가능 트랜잭션 격리 수준을 사용하는 경우, 일관성을 보장하기 위해 다른 노력이 필요하지 않습니다. 일관성을 보장하기 위해 직렬화 가능 트랜잭션을 사용하도록 작성된 다른 환경의 소프트웨어는 PostgreSQL에서 이 점에서 "그냥 작동"해야 합니다.

이 기술을 사용할 때 애플리케이션 소프트웨어가 직렬화 실패로 롤백된 트랜잭션을 자동으로 재시

도하는 프레임워크를 거치면 애플리케이션 프로그래머에게 불필요한 부담을 주지 않을 수 있습니다. 기본_트랜잭션_격리를 직렬화 가능으로 설정하는 것이 좋습니다. 또한 트리거에서 트랜잭션 격리 수준을 확인하여 실수로 또는 무결성 검사를 우회하기 위해 다른 트랜잭션 격리 수준이 사용되지 않도록 조치를 취하는 것이 현명할 수 있습니다.

성능 제안은 13.2.3항을 참조하세요.

> **경고: 직렬화 가능한 트랜잭션 및 데이터 복제**
>
> 직렬화 가능 트랜잭션을 사용하는 이 수준의 무결성 보호는 아직 핫 스탠바이 모드(섹션 27.4) 또는 논리적 복제본에는 적용되지 않습니다. 따라서 핫 스탠바이 또는 논리적 복제를 사용하는 사용자는 반복 읽기 및 기본값에 대한 명시적 잠금을 사용해야 할 수 있습니다.

13.4.2. 명시적 차단 잠금으로 일관성 유지

직렬화할 수 없는 쓰기가 가능한 경우 행의 현재 유효성을 보장하고 동시 업데이트로부터 행을 보호하려면 SELECT FOR UPDATE, SELECT FOR SHARE 또는 적절한 LOCK TABLE 문을 사용해야 합니다. (SELECT FOR UPDATE 및 SELECT FOR SHARE는 동시 업데이트에 대해 반환된 행만 잠그는 반면, LOCK TABLE은 전체 테이블을 잠급니다.) 다른 환경에서 PostgreSQL로 애플리케이션을 포팅할 때 이 점을 고려해야 합니다.

또한 다른 환경에서 변환하는 경우 주목해야 할 점은 SELECT FOR UPDATE가 동시 트랜잭션이 선택한 행을 업데이트하거나 삭제하지 않도록 보장하지 않는다는 사실입니다. PostgreSQL에서 이를 수행하려면 값을 변경할 필요가 없더라도 실제로 행을 업데이트해야 합니다. SELECT FOR UPDATE는 다른 트랜잭션이 동일한 잠금을 획득하거나 잠긴 행에 영향을 줄 수 있는 UPDATE 또는 DELETE를 실행하는 것을 *일시적으로* 차단하지만, 이 잠금을 보유한 트랜잭션이 커밋되거나 롤백되면 잠금이 유지되는 동안 행의 실제 업데이트가 수행되지 않는 한 차단된 트랜잭션은 충돌하는 작업을 진행하게 됩니다.

글로벌 유효성 검사는 직렬화할 수 없는 MVCC에서 추가적인 고려가 필요합니다. 예를 들어 은행 애플리케이션에서는 두 테이블이 모두 활발하게 업데이트되고 있을 때 한 테이블의 모든 크레딧 합계가 다른 테이블의 차변 합계와 동일한지 확인하고자 할 수 있습니다. 두 개의 연속된 SELECT sum(...) 명령의 결과를 비교하는 것은 읽기 커밋 모드에서 안정적으로 작동하지 않습니다. 두 번째 쿼리에는 첫 번째 쿼리에서 계산되지 않은 트랜잭션의 결과가 포함될 가능성이 높기 때문입니다. 하나의 반복 가능한 읽기 트랜잭션에서 두 합계를 수행하면 반복 가능한 읽기 트랜잭션이 시작되기 전에 커밋된 트랜잭션의 효과만 정확하게 파악할 수 있지만 응답이 전달되는 시점까지 여전히 관련성이 있는지 궁금해할 수 있습니다. 반복 읽기 트랜잭션 자체가 일관성 검사를 시도하기 전에 일부 변경 사항을 적용했다면, 이제 트랜잭션 시작 후의 모든 변경 사항이 아니라 일부만 포함되므로 이 검사의 유용성에 대해 더욱 논쟁의 여지가 생깁니다. 이러한 경우 신중한 사람은 현재 현실을 확실하게 파악하기 위해 검사에 필요한 모든 테이블을 잠그는 것이 좋습니다. 공유 모드(또는 그 이상) 잠금은 잠긴 테이블에 현재 트랜잭션의 변경 내용 외에 커밋되지 않은 변경 내용이 없음을 보장합니다.

동시 변경을 방지하기 위해 명시적 잠금에 의존하는 경우, 읽기 커밋 모드를 사용하거나 반복 가능한 읽기 모드에서 쿼리를 수행하기 전에 잠금을 얻도록 주의해야 한다는 점도 유의하세요. 반복 가능한 읽기 트랜잭션이 잠금을 획득하면 테이블을 수정하는 다른 트랜잭션이 아직 실행되고 있지 않음을 보장하지만, 트랜잭션이 보는 스냅샷이 잠금을 획득하기 전에 있는 경우 테이블에서 현재 커밋된 일부 변경 사항보다 앞설 수 있습니다. 반복 가능한 읽기 트랜잭션의 스냅샷은 실제로 첫 번째 쿼리 또는 데이터 수정 명령(SELECT, INSERT, UP- DATE, DELETE 또는 MERGE)이 시작될 때 고정되므로 스냅샷이 고정되기 전에 명시적으로 잠금을 획득할 수 있습니다.

13.5. 직렬화 실패 처리

반복 읽기 및 직렬화 가능 격리 수준은 모두 직렬화 이상을 사전에 차단하도록 설계된 오류가 발생할 수 있습니다. 앞서 설명한 대로 이러한 격리 수준을 사용하는 애플리케이션은 직렬화 오류로 인해 실패한 트랜잭션을 다시 시도할 수 있도록 준비해야 합니다. 이러한 오류의 메시지 텍스트는 정확한 상황에 따라 다르지만, 항상 SQLSTATE 코드 `40001`(`serializa- tion_failure`)이 포함됩니다.

교착 상태 실패를 다시 시도하는 것이 좋습니다. 이 경우 SQLSTATE 코드가 `40P01`(`dead- lock_detected`)입니다.

경우에 따라 SQLSTATE 코드가 `23505`(고유_위반)인 고유 키 오류와 SQLSTATE 코드가 `23P01`(제외_위반)인 제외 제약 조건 오류를 다시 시도하는 것이 적절할 수도 있습니다. 예를 들어 애플리케이션이 현재 저장된 키를 검사한 후 기본 키 열의 새 값을 선택하는 경우, 다른 애플리케이션 인스턴스가 동일한 새 키를 동시에 선택했기 때문에 고유 키 오류가 발생할 수 있습니다. 이는 사실상 직렬화 실패이지만, 서버는 삽입된 키 사이의 연결을 "볼" 수 없기 때문에 이를 직렬화 실패로 감지하지 못합니다.

값과 이전 읽기를 비교합니다. 또한 원칙적으로 직렬화 문제가 근본적인 원인이라고 판단하기에 충분한 정보가 있음에도 불구하고 서버가 고유 키 또는 제외 제약 조건 오류를 발생시키는 코너 케이스도 있습니다. 직렬화_실패 오류는 무조건 다시 시도하는 것이 좋지만, 다른 오류 코드는 일시적인 오류가 아니라 지속적인 오류 상태를 나타낼 수 있으므로 다시 시도할 때는 더 많은 주의가 필요합니다.

발행할 SQL 및/또는 사용할 값을 결정하는 모든 로직을 포함하여 전체 트랜잭션을 다시 시도하는 것이 중요합니다. 따라서 PostgreSQL은 정확성을 보장할 수 없기 때문에 자동 재시도 기능을 제공하지 않습니다.

트랜잭션 재시도는 재시도된 트랜잭션이 완료된다는 보장은 없으며, 여러 번 재시도가 필요할 수 있습니다. 경합이 매우 심한 경우 트랜잭션이 완료되는 데 여러 번 시도해야 할 수 있습니다. 충돌하는 준비된 트랜잭션이 있는 경우, 준비된 트랜잭션이 커밋되거나 롤백될 때까지 진행하지 못할 수도 있습니다.

13.6. 주의 사항

현재 TRUNCATE와 테이블 재작성 형식의 `ALTER TABLE` 등 일부 DDL 명령은 MVCC에 안전하지 않습니다. 즉, 트리트먼트 또는 다시 쓰기 커밋 후 동시 트랜잭션이 DDL 명령이 커밋되기 전에 생성된 스냅샷을 사용하는 경우 테이블이 비어 있는 것처럼 보입니다. 이는 DDL 명령이 시작되기 전에 해당 테이블에 액세스하지 않은 트랜잭션에서만 문제가 되며, 액세스한 모든 트랜잭션은 최소한 액세스 공유 테이블 잠금을 유지하여 해당 트랜잭션이 완료될 때까지 DDL 명령을 차단합니다. 따라서 이러한 명령은 대상 테이블에 대한 연속적인 쿼리에 대해 테이블 내용에 명백한 불일치를 일으키지는 않지만 대상 테이블의 내용과 데이터베이스의 다른 테이블 간에 눈에 보이는 불일치를 일으킬 수 있습니다.

직렬화 가능 트랜잭션 격리 수준에 대한 지원은 아직 핫 스탠바이 복제 대상에 추가되지 않았습니다(섹션 27.4에 설명되어 있습니다). 현재 핫 스탠바이 모드에서 지원되는 가장 엄격한 격리 수준은 반복 가능한 읽기입니다. 프라이머리에서 반복 가능 읽기 트랜잭션 내에서 모든 영구 데이터베이스 쓰기를 수행하면 모든 스탠바이가 결국 일관된 상태에 도달하지만, 스탠바이에서 실행되는 반복 가능 읽기 트랜잭션은 프라이머리에서 트랜잭션의 연속 실행과 일치하지 않는 일시적인 상태를 볼 수 있습니다.

시스템 카탈로그에 대한 내부 액세스는 현재 트랜잭션의 격리 수준을 사용하여 수행되지 않습니다. 즉, 테이블과 같이 새로 생성된 데이터베이스 개체는 그 안에 포함된 행이 아니더라도 동시 반복 읽기 및 직렬화 가능 트랜잭션에 표시됩니다. 반대로 시스템 카탈로그를 명시적으로 검사하는 쿼리는 격리 수준이 높을수록 동시에 생성된 데이터베이스 개체를 나타내는 행을 볼 수 없습니다.

13.7. 잠금 및 색인

PostgreSQL은 테이블 데이터에 대한 비차단 읽기/쓰기 액세스를 제공하지만, 현재 PostgreSQL에서 구현된 모든 인덱스 액세스 방법에 대해 비차단 읽기/쓰기 액세스가 제공되지는 않습니다. 다양한 인덱스 유형은 다음과 같이 처리됩니다:

B-tree, GiST 및 SP-GiST 인덱스

읽기/쓰기 액세스에는 단기 공유/독점 페이지 수준 잠금이 사용됩니다. 각 인덱스 행이 가져오거나 삽입되면 즉시 잠금이 해제됩니다. 이러한 인덱스 유형은 교착 상태 조건 없이 가장 높은 동시성을 제공합니다.

해시 인덱스

읽기/쓰기 액세스에는 공유/독점 해시 버킷 수준의 잠금이 사용됩니다. 전체 버킷이 처리된 후에 잠금이 해제됩니다. 버킷 수준 잠금은 인덱스 수준 잠금보다 더 나은 동시성을 제공하지만, 하나의 인덱스 작업보다 더 오래 잠금이 유지되므로 교착 상태가 발생할 수 있습니다.

GIN 인덱스

읽기/쓰기 액세스에는 단기 공유/독점 페이지 수준 잠금이 사용됩니다. 각 인덱스 행을 가져오거나 삽입하면 즉시 잠금이 해제됩니다. 그러나 GIN 인덱싱된 값을 삽입하면 일반적으로 행당 여러 개의 인덱스 키 삽입이 발생하므로 단일 값 삽입에 대해 GIN이 상당한 작업을 수행할 수 있습니다.

현재 B-tree 인덱스는 동시 애플리케이션에 가장 적합한 성능을 제공하며, 해시 인덱스보다 더 많은 기능을 가지고 있기 때문에 스칼라 데이터를 인덱싱해야 하는 동시 애플리케이션에 권장되는 인덱스 유형입니다. 스칼라가 아닌 데이터를 처리할 때는 B-tree가 유용하지 않으며 대신 GiST, SP-GiST 또는 GIN 인덱스를 사용해야 합니다.