ChatGPT

PostgreSQL vs MySQL: 마이그레이션 이유, SQL 차이점, EXPLAIN 계획 이해 및 Spring Boot 풀링 전략

1. MySQL에서 PostgreSQL로 마이그레이션하거나 초기 설계 단계에서 PostgreSQL을 선택하는 이유

기업들이 **PostgreSQL**을 선택하는 주된 이유는 다음과 같습니다 (MySQL 대비 장점):

- **높은 성능 및 동시성**: PostgreSQL은 대용량 트랜잭션과 복잡한 쿼리에 최적화되어 있으며, 복잡한 조인이나 윈도우 함수 등의 고급 쿼리도 효율적으로 처리합니다 [1] [2]. 일반적인 OLTP(읽기/쓰기 혼합) 환경에서 PostgreSQL과 MySQL의 성능은 비슷하지만, **대규모 동시성** 환경에서 PostgreSQL은 프로세스당 1커넥션 모델로 안정적인 처리를 보장합니다. MySQL은 스레드당 1커넥션 모델로 고성능 쓰기(workload)에서 약간 유리할 수 있으나, PostgreSQL도 인터넷 규모의 부하를 처리할 수 있고 고도 최적화된 쿼리 플래너로 복잡한 질의를 빠르게 실행합니다 [3]. 예를 들어 **윈도우 함수**의 경우 PostgreSQL 구현이 MySQL보다 효율적이라는 평가도 있습니다 [4]. 다만 **매우** 쓰기 집중적인 워크로드에서는 MySQL이 VACUUM 불필요 등의 설계로 유리한 면도 있다는 분석이 있으므로(예: Uber의 사례) 각 워크로드에 맞는 DB를 선택해야 합니다 [5].

- **풍부한 기능**: PostgreSQL은 **표준 SQL 호환성과 고급 기능**을 폭넓게 제공합니다. 예를 들어 **JSON/JSONB 데이터 타입과 인덱스 지원**, **CTE(Common Table Expression)**를 통한 서브쿼리 활용, **윈도우 함수** 및 **부분 집계**, **레인지 타입과 배열 타입**, **함수/프로시저**, **트리거와 이벤트** 등 기능 면에서 MySQL보다 앞서 있습니다 [1]. MySQL도 8.0 버전 이후로 많은 기능이 추가되었지만, PostgreSQL은 **JSONB** 자료형을 통해 JSON 데이터를 바이너리로 저장하고 인덱싱하여 MySQL보다 효율적으로 JSON 데이터를 처리할 수 있습니다 [6] [7]. 또한 CTE의 경우 PostgreSQL은 SELECT, INSERT, UPDATE, DELETE 모두 CTE에서 사용할 수 있으나 MySQL은 제한적으로 지원합니다 [8]. 이런 풍부한 기능 덕분에 개발자는 PostgreSQL 하나로 다양한 요구사항을 충족할 수 있어 별도의 NoSQL을 도입하지 않아도 되는 등 **설계 단순화** 이점이 있습니다.

- **확장성 및 대용량 처리**: PostgreSQL은 **수평적/수직적 확장**에 모두 대응이 가능합니다. 수직적으로는 인덱스 튜닝, 파티셔닝, 병렬 쿼리 등의 기능으로 큰 단일 인스턴스 성능을 끌어올릴 수 있습니다. 수평적으로도 스트리밍 리플리케이션, 논리적 복제(Publish/Subscribe) 지원, 샤딩을 위한 확장(예: Citus) 등을 통해 확장성을 갖춥니다 [9] [10]. 특히 PostgreSQL 생태계에는 **TimescaleDB**(시계열), **Citus**(샤딩), **Postgres-XL** 등 확장 솔루션이 다양합니다. 반면 MySQL도 그룹 복제나 Vitess 등의 확장 솔루션이 있지만, PostgreSQL은 오픈소스 커뮤니티 주도로 이러한 확장 기능들이 활발히 개발되고 있습니다. 또한 PostgreSQL은 **프로세스당 연결** 모델이라 **동시 연결 수** 증가 시 오버헤드가 존재하지만, 일반적으로 수백 개 수준의 커넥션까지는 성능이 잘 유지되며, 대규모 연결 환경에서는 pgBouncer 같은 풀링을 통해 안정적으로 확장합니다.

- **오픈소스 라이선스의 이점**: PostgreSQL은 PostgreSQL License라는 BSD 유사 라이선스로 배포되며 **사실상 아무 제약 없이 무료**로 사용할 수 있습니다 [11]. 소스 수정, 재배포, 상용 소프트웨어에 내장 등 모든 사용이 자유롭습니다. 반면 MySQL은 커뮤니티 에디션의 경우 GPL 라이선스이므로, 만약 **제품에 MySQL을 내장**하여 배포하거나 소스코드를 정적으로 링크하는 경우 **상용 라이선스 구매 또는 소스 공개 의무**가 발생할 수 있습니다 [12] [13]. 예를 들어 **자체 소프트웨어에 DB를 번들**하여 배포해야 하는 경우 PostgreSQL을 택하면 라이선스 위험 없이 이용 가능합니다. 이 같은 이유로 Yahoo 등 일부 기업은 폐쇄형 제품의 내장 DB로 PostgreSQL을 선택해 왔습니다 [13].

- **안정성과 신뢰성**: PostgreSQL은 20년 넘게 커뮤니티에 의해 개발되며 **검증된 안정성**을 자랑합니다. 완벽한 ACID 트랜잭션 준수, 외래키 등 제약 조건의 철저한 enforcement, Write-Ahead Logging에 기반한 **신뢰성**

높은 복구 메커니즘 등을 갖추고 있어 금융, 정부 등 분야에서도 신뢰할 수 있는 DB로 평가됩니다. 실제 데이터 무결성과 일관성이 중요한 **업무 핵심 시스템**에서 PostgreSQL은 선호되며, 전문가들은 PostgreSQL을 "훨씬 더 강력하고 안정적인 RDBMS"로 평하기도 합니다 [14]. 반면 MySQL은 과거에 제약 조건 무시나 오류 처리 미흡 사례(예: 잘못된 쿼리에 의미없는 결과 반환 등)가 있었고, 외래키 트리거 문제 등의 버그가 오래 방치되는 등 (DELETE CASCADE 시 트리거 동작 버그, 2005년 보고 후 미해결 [15]) 안정성 이슈가 지적된 바 있습니다. Oracle 인수 이후 MySQL의 개발 속도가 정체되었다는 의견도 있으며 [16], 이런 불안감이 PostgreSQL로의 이동을 가속하는 요소입니다. 다만 두 DB 모두 현재는 성숙한 상태로, 일반적인 OLTP 운영에서 **신뢰성은 충분히 검증**되었다고 볼 수 있습니다.

- **활발한 커뮤니티와 생태계**: PostgreSQL은 **커뮤니티 주도로 개발**되는 오픈소스 프로젝트로, 전 세계 개발자들의 참여가 활발합니다 [17]. 그 결과 기능 개선과 버그 수정이 빠르게 이루어지며, 풍부한 확장 기능들도 커뮤니티에서 쏟아져 나오고 있습니다 [18]. 예를 들어 지리정보를 위한 PostGIS, AI 임베딩 검색을 위한 pgvector 등 수많은 **서드파티 확장**을 손쉽게 설치해 기능을 확장할 수 있습니다 [18]. 또한 **클라우드 호스팅 생태계**에서도 PostgreSQL이 주도적입니다. Heroku를 시작으로 Supabase, Render, Fly.io 등 거의 모든 서비스형 플랫폼이 기본 RDBMS로 PostgreSQL을 제공하며 [19], 이는 곧 신규 스타트업들이 PostgreSQL을 채택하는 사례로 이어집니다. Stack Overflow 개발자 설문에서도 PostgreSQL 선호도가 계속 높아져 2020년대 들어 MySQL을 앞지르는 추세입니다. 전체적으로 "Postgres는 더 많은 기능과 활발한 커뮤니티, MySQL은 학습용이성과 방대한 사용자층"으로 요약되는데 [20], 최근 업계 트렌드는 PostgreSQL 쪽으로 기울고 있습니다. 실제로 많은 신규 프로젝트가 PostgreSQL을 기본으로 선택하며, 기존 MySQL 사용 기업들도 기능 확장이나 라이선스 이슈 등을 이유로 PostgreSQL로 마이그레이션하는 사례가 늘고 있습니다.

요약하면, **PostgreSQL은 풍부한 기능과 엄격한 표준 준수, 유연한 라이선스, 커뮤니티 지원** 등의 이유로 **현대 애플리케이션에 매력적인 선택지**입니다. 반대로 MySQL은 **초기 학습 및 설정이 쉬운 편이고, 특정 극한의 쓰기 부하 시 성능**이나 **여전히 방대한 기존 사용자층** 등의 강점이 있어, 두 DB 모두 상황에 따라 병행 활용되기도 합니다. 하지만 **추가 기능 요구사항이나 라이선스 제약, 기술 전략** 등을 고려할 때 PostgreSQL로의 전환을 검토하는 기업이 많아지는 추세입니다.

2. PostgreSQL과 MySQL의 SQL 쿼리문 차이 – CRUD 및 JOIN 비교

PostgreSQL과 MySQL은 기본적인 SQL 문법은 유사하지만, **세부적인 DML 구문과 동작에 차이**가 있습니다. 여기서는 **CRUD**(CREATE=INSERT, READ=SELECT, UPDATE, DELETE)와 **JOIN** 관련 차이를 중심으로 살펴보고, 예제 코드와 함께 사용상의 주의점을 설명합니다.

INSERT 문 및 UPSERT 비교

- **자동 증가 기본키**: MySQL에서는 `AUTO_INCREMENT` 속성을 사용하여 자동 증가하는 기본키를 만들고 INSERT 시 값을 생략합니다. PostgreSQL에서는 유사 기능으로 **시퀀스(sequence)**를 활용하며, 간편하게는 `SERIAL` 또는 `BIGSERIAL` 타입을 사용하거나 PostgreSQL 10+에서는 `GENERATED AS IDENTITY`를 사용합니다. 예를 들어 MySQL에서:

```
CREATE TABLE users (
  id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(100)
);
INSERT INTO users (name) VALUES ('Alice');
```

PostgreSQL에서는 `SERIAL` 타입을 사용하거나 별도 시퀀스를 생성합니다:

```
CREATE TABLE users (
 id SERIAL PRIMARY KEY,
 name VARCHAR(100)
);
INSERT INTO users (name) VALUES ('Alice');
```

MySQL의 `AUTO_INCREMENT`는 테이블당 하나만 사용할 수 있고, 명시적으로 초기값을 설정하려면 `ALTER TABLE ... AUTO_INCREMENT=n`을 사용합니다. PostgreSQL의 시퀀스는 독립 객체로 `currval`, `setval` 등을 통해 제어할 수 있습니다. **주의**: PostgreSQL에서 SERIAL은 내부적으로 시퀀스를 생성하며, 테이블을 삭제해도 시퀀스는 남을 수 있으므로 관리에 유의합니다.

- **UPSERT (Insert or Update)**: MySQL은 전통적으로 **UPSERT**를 위해 `INSERT ... ON DUPLICATE KEY UPDATE` 구문을 사용합니다. 즉, INSERT 중 충돌(기존 PRIMARY KEY나 UNIQUE 키 중복)이 발생하면 지정한 UPDATE 절을 실행합니다. 예를 들어 MySQL:

```
INSERT INTO users (id, name) VALUES (1, 'Alice')
ON DUPLICATE KEY UPDATE name = 'Alice';
```

이 구문은 id=1이 이미 존재하면 name 필드를 'Alice'로 업데이트합니다. MySQL에서는 `VALUES(col_name)` 함수를 사용하여 INSERT하려던 값을 참조할 수 있으며, 여러 행 INSERT 시 충돌된 행에 대해서만 UPDATE가 적용됩니다 [21] [22]. **주의점**: `ON DUPLICATE KEY UPDATE`는 **멀티컬럼 UNIQUE 제약** 등에도 적용되며, 충돌 없는 행은 그대로 INSERT됩니다. 또한 MySQL에는 `REPLACE INTO` 구문도 있는데, 이것은 "기존 행이 있으면 지우고 새로 INSERT"하는 동작으로, 외래키 제약 등이 있을 경우 주의가 필요합니다 (실무에선 잘 사용되지 않음).

PostgreSQL은 9.5버전부터 SQL 표준에 맞춘 `INSERT ... ON CONFLICT` 구문으로 UPSERT를 지원합니다 [23]. 구문 형태는 `ON CONFLICT (<충돌대상 열>) DO UPDATE SET ...` 혹은 `DO NOTHING`입니다. 예를 들어 동일한 동작을 PostgreSQL에서 구현하면:

```
INSERT INTO users (id, name) VALUES (1, 'Alice')
ON CONFLICT (id) DO UPDATE
 SET name = EXCLUDED.name;
```

여기서 `EXCLUDED`는 이번에 INSERT하려던 행을 가리키는 특별한 alias이며, 해당 충돌 열(id)이 겹치는 경우에만 UPDATE 절이 수행됩니다 [24]. `DO NOTHING` 옵션을 사용하면 충돌 시 아무 작업도 하지 않고 조용히 넘어갑니다. **주의점**: `ON CONFLICT` 절에는 반드시 충돌 대상 인덱스나 제약을 지정해야 하며, 여러 개의 UNIQUE 제약 조건이 있는 경우 특정 하나만 대상으로 삼습니다. PostgreSQL의 UPSERT는 **원자적**으로 동작하여 높은 동시성에서도 Insert 또는 Update 중 하나만 일어나도록 보장합니다 [25]. MySQL의 UPSERT와 거의 동일한 목적이지만 구문이 다르므로, 마이그레이션 시 해당 부분을 변환해야 합니다.

- **INSERT 후 생성된 ID 반환**: MySQL에서는 `LAST_INSERT_ID()` 함수를 통해 마지막 AUTO_INCREMENT 값을 가져올 수 있습니다. PostgreSQL에서는 `INSERT ... RETURNING <컬럼>` 절을 통해 방금 INSERT한 행의 특정 컬럼(예: 기본키)을 바로 반환받는 것이 가능합니다 [26]. 예를 들어 `INSERT ... RETURNING id;` 형태로 사용하며, 한 번의 DB왕복으로 INSERT와 ID 취득이 동시에 이뤄집니다 (MySQL 8.0.20+에서도 `INSERT ... RETURNING` 일부 지원이 추가되었지만 PostgreSQL만큼 범용적이진 않습니다).

- **기타 차이**: MySQL은 `INSERT IGNORE` 옵션을 제공하여 UNIQUE 충돌 시 에러를 무시하고 경고만 발생시킨 후 건너뛰게 할 수 있습니다. PostgreSQL에는 이런 옵션은 없고 `ON CONFLICT DO NOTHING`으로 유사 효과를 냅니다. 또한 MySQL은 다중 테이블에 대한 `INSERT ... SELECT`에도 ON DUPLICATE를 사용할 수 있고, `SET`을 사용한 INSERT 문법도 제공합니다 [27] [28]. PostgreSQL에서는 표준 `INSERT ... SELECT`만 지원합니다.

SELECT 및 JOIN 문 비교

- **ANSI 조인 구문**: 기본적인 `SELECT` 문과 `JOIN` 구문은 PostgreSQL과 MySQL 모두 ANSI SQL 표준을 따릅니다. 예를 들어 두 테이블을 JOIN하는 구문은 동일합니다:

```
SELECT o.order_id, c.customer_name
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
WHERE c.region = 'ASIA';
```

다만 **MySQL은 여러 데이터베이스 간 조인**을 지원하는 반면(PostgreSQL 용어로는 **스키마**가 아니라 서로 다른 DB 간), **PostgreSQL은 하나의 데이터베이스 내에서만 직접 조인**이 가능합니다 [29]. MySQL에서는 `dbname.tableName` 식으로 다른 DB의 테이블을 조인할 수 있지만, PostgreSQL은 그 대신 **스키마(schema)** 개념을 사용하여 동일 DB 내 논리 분리를 하고 조인합니다 [30] [31]. PostgreSQL에서 서로 다른 DB의 테이블을 조인하려면 FDW(Foreign Data Wrapper)를 통해 원격 DB를 연결해야 하므로, **어플리케이션 설계 시 스키마를 활용하거나 DB 구성**을 고려해야 합니다 [29].

- **GROUP BY와 엄격한 모드**: **MySQL은 기본 설정에서 관대한(SQL 표준 위배 허용) 그룹바이**를 허용합니다. 즉 `SELECT` 절에 그룹화되지 않은 컬럼이 있어도 에러를 내지 않고 비정의(non-deterministic) 값으로 결과를 반환합니다. 예를 들어 MySQL에서는 `SELECT department, name FROM employees GROUP BY department;`가 에러 없이 실행되어 각 부서별 임의의 name이 선택될 수 있습니다. PostgreSQL은 **SQL 표준을 준수**하여, `GROUP BY`시 집계되지 않은 컬럼은 `SELECT`에 넣을 수 없고 에러를 발생시킵니다 [32]. (MySQL에서도 `ONLY_FULL_GROUP_BY` 모드를 켜면 PostgreSQL과 동일한 엄격 모드가 적용됩니다.) 따라서 MySQL용 쿼리를 PostgreSQL로 옮길 때는 GROUP BY 구문을 특히 주의해서 검토해야 합니다.

- **대소문자와 식별자 인용**: MySQL은 기본적으로 **대소문자 구분을 하지 않는** 규칙이 많습니다(테이블/컬럼명은 대소문자 무시. 단, 파일시스템 따라 테이블명 민감성 달라질 수 있음). PostgreSQL은 **모든 식별자를 소문자로 자동 변환**하며 대소문자 구분이 기본적으로 있습니다 [32]. 즉, PostgreSQL에서 쿼리를 작성할 때 대소문자를 구분하고 싶지 않다면 **모든 식별자를 소문자**로 만들거나, 대소문자가 섞인 식별자는 `"`로 인용하여 정의하고 쿼리에서도 `"`로 동일하게 인용해야 합니다. 예를 들어 MySQL에서 `SELECT * FROM Users;` (대소문자 섞인 테이블명)이 동작할 수 있지만 PostgreSQL에서는 해당 테이블을 `"Users"`로 생성하지 않는 한 인식하지 못합니다. **실무 팁**: PostgreSQL에서는 식별자를 소문자 사용을 권장하며, MySQL에서 백틱<code>`</code>으로 감싸는 습관은 PostgreSQL에서는 **더블쿼트(")**로 대체된다는 점을 알아두세요.

- **JOIN 구문 확장**: 두 DB 모두 INNER/LEFT/RIGHT/FULL JOIN 및 CROSS JOIN, SELF JOIN 등을 지원합니다. 차이가 있다면, MySQL은 `JOIN ... USING (column)` 구문 시 결과 컬럼 정리가 약간 PostgreSQL과 다를 수 있습니다만 대동소이합니다. 한 가지 MySQL만의 구문으로, `STRAIGHT_JOIN` (옵티마이저 힌트로 조인 순서 강제) 등이 있지만 PostgreSQL에는 그런 전용 키워드는 없습니다 (대신 `ENABLE_NESTLOOP` 등 세션 파라미터로 힌트 효과를 줄 수 있음).

- **서브쿼리 및 WITH (CTE)**: 공통 테이블 표현식(CTE, `WITH` 구문)은 PostgreSQL이 더 강력합니다. PostgreSQL에서는 CTE내에서 `INSERT/UPDATE/DELETE`를 수행하거나 재귀 CTE 등 다양하게 활용 가능

합니다 [8] . MySQL도 8.0부터 `WITH`를 지원하지만, CTE내 DML은 지원하지 않고 SELECT용으로만 씁니다. 또한 PostgreSQL CTE는 기본적으로 **실제 실행 시 최적화 배제(fenced)**되어 독립 쿼리처럼 동작하지만, MySQL CTE는 그냥 뷰처럼 인라인 최적화됩니다. 이러한 최적화 차이로 성능 영향이 있을 수 있어, PostgreSQL 12 이후 `WITH ... MATERIALIZED/NOT MATERIALIZED` **힌트**로 제어할 수 있습니다.

• **예제 – 서로 다른 DB 조인**: (MySQL에서는 가능, PostgreSQL에서는 불가)

• MySQL

```
SELECT *
FROM db1.customers AS c
JOIN db2.orders AS o ON c.id = o.customer_id
WHERE c.name = 'Alice';
```

• PostgreSQL

```
-- 서로 다른 DB의 테이블 조인은 직접 지원되지 않음 (오류 발생).
-- 대신 두 테이블이 하나의 DB 내 서로 다른 스키마에 있다면 가능:
SELECT *
FROM customers_schema.customers AS c
JOIN orders_schema.orders AS o ON c.id = o.customer_id
WHERE c.name = 'Alice';
```

• **예제 – GROUP BY 차이**:

• MySQL (ONLY_FULL_GROUP_BY 설정 끈 상태)

```
SELECT department, name, COUNT(*)
FROM employees
GROUP BY department;
-- MySQL: name이 GROUP BY에 없지만 에러 없이 실행 (임의의 name 선택됨)
```

• PostgreSQL

```
SELECT department, name, COUNT(*)
FROM employees
GROUP BY department, name;
-- 또는 원하는 집계값에 따라 name에 집계함수를 사용해야 함.
```

위 PostgreSQL 쿼리는 부서별 직원수를 세면서 이름도 그룹화한 경우이고, 만약 "부서별 직원수와 그 부서의 임의 직원 이름"을 의도했다면 하나의 쿼리로는 정의되지 않은 동작이므로 PostgreSQL에서는 애초에 허용하지 않습니다 (서브쿼리 등으로 명시적으로 처리해야 합니다). **교훈**: MySQL에서 동작하던 느슨한 GROUP BY 쿼리는 PostgreSQL로 옮길 때 에러가 날 수 있으므로 쿼리를 명확히 수정해야 합니다 [32] .

UPDATE 문 비교

- **다중 테이블 업데이트 (조인 업데이트)**: MySQL은 한 번의 UPDATE에서 **여러 테이블을 동시에 갱신하거나, 조인을 사용하여 다른 테이블 값을 활용**할 수 있습니다 [33]. 구문은 `UPDATE <테이블들> SET ... FROM ... WHERE ...` 형태 또는 `UPDATE t1, t2 SET ... WHERE t1.x = t2.y ...` 형태를 취합니다. 예를 들어 **MySQL**에서 두 테이블의 값을 조인해서 업데이트할 때:

```
-- MySQL: items 테이블의 price를 month 테이블의 price로 일괄 업데이트
UPDATE items
 JOIN month ON items.id = month.id
SET items.price = month.price
WHERE month.year = 2025;
```

혹은 동치 문법:

```
UPDATE items, month
SET items.price = month.price
WHERE items.id = month.id AND month.year = 2025;
```

위 쿼인은 items와 month를 조인한 결과에 대해 items의 가격을 변경합니다 [33]. MySQL에서는 `UPDATE ... JOIN` 구문에 `LEFT JOIN` 등의 외부 조인도 사용할 수 있습니다 [34].

PostgreSQL에서는 기본적으로 **UPDATE 대상은 한 개 테이블**이어야 하며, 다른 테이블과 조인하려면 `FROM` 절을 사용합니다. PostgreSQL에서 위 작업을 구현하면:

```
UPDATE items i
SET price = m.price
FROM month m
WHERE i.id = m.id AND m.year = 2025;
```

`FROM` 절에 조인할 테이블을 명시하고, `WHERE`로 조인 조건을 지정합니다 [35]. 이렇게 하면 items를 업데이트하면서 month 테이블을 참조할 수 있습니다. **주의점**: `FROM`절에 조인된 테이블들 중 실제 SET으로 업데이트되는 것은 대상 테이블(items)뿐입니다. MySQL의 다중 테이블 UPDATE처럼 한 쿼리로 두 테이블의 값을 모두 바꾸는 것은 PostgreSQL에서는 지원하지 않으므로, 필요하다면 별도 UPDATE 두 번으로 나누거나 함수 이용, 규칙(RULE) 등을 사용해야 합니다. 또한 PostgreSQL에서 `UPDATE ... FROM`으로 조인할 때 **한쪽 조인 값이 여러 행과 매치**될 경우 임의의 하나만 사용되므로, 조인 키가 **유일**하도록 (보통 PK/FK 관계) 쿼리를 작성해야 합니다 [36] [37]. 만약 다대다 조인 등의 상황이라면 서브쿼리 등을 통해 결정적으로 값을 정하도록 쿼리를 구성해야 합니다.

- **UPDATE ... LIMIT / ORDER BY**: MySQL은 단일 테이블 UPDATE 시 `ORDER BY` 절과 `LIMIT` 절을 지원하여, **특정 개수의 행만 업데이트**하거나 특정 순서로 처리하도록 할 수 있습니다 [38] [39]. 예를 들어:

```
UPDATE transactions
SET processed = TRUE
WHERE processed = FALSE
ORDER BY created_at
LIMIT 100;
```

위 쿼인은 조건에 맞는 행 중 가장 오래된 것 100개만 업데이트합니다. **PostgreSQL는 UPDATE에 LIMIT를 직접 지원하지 않으며**, 이 효과를 내려면 서브쿼리를 사용해야 합니다. 예를 들어 위와 동일한 작업을 하려면:

```
UPDATE transactions t
SET processed = TRUE
FROM (SELECT id FROM transactions
    WHERE processed = FALSE
    ORDER BY created_at
    LIMIT 100) sub
WHERE t.id = sub.id;
```

처럼 서브쿼리로 100개의 ID를 선정하여 조인 업데이트를 해야 합니다. PostgreSQL 14부터는 TOP-N 업데이트를 위해 `CTE`와 `LIMIT`을 조합하는 패턴(`WITH ... SELECT ... LIMIT ... UPDATE`)이 최적화되기도 했습니다. **주의**: MySQL의 `UPDATE ... LIMIT`는 종종 임시 순위컬럼 업데이트 등의 편법에 쓰이지만, PostgreSQL에서는 명시적으로 처리해야 하므로 마이그레이션 시 그런 쿼리가 있는지 점검해야 합니다.

- **서브쿼리 사용**: MySQL과 PostgreSQL 모두 UPDATE문에서 `SET col = (SELECT ...)` 형태로 서브쿼리를 사용 가능합니다. 다만 MySQL은 동일 쿼리 내에서 업데이트 대상 테이블을 서브쿼리에서 바로 참조하면 에러(LOCK 문제) 발생하여, 자기 자신의 값을 기준으로 업데이트할 때는 조인 업데이트로 우회해야 합니다 [40] [41]. PostgreSQL은 그러한 제약은 없고 트랜잭션 수준에서 동시성만 조심하면 됩니다.

- **예제 – UPDATE 조인**:

- MySQL

```
-- 직원 테이블과 부서 테이블 조인해서 직원의 dept_name 컬럼 갱신
UPDATE employee e
  JOIN department d ON e.dept_id = d.dept_id
SET e.dept_name = d.dept_name
WHERE d.active = 1;
```

- PostgreSQL

```
UPDATE employee e
SET dept_name = d.dept_name
FROM department d
WHERE e.dept_id = d.dept_id
  AND d.active = TRUE;
```

위 PostgreSQL 쿼인은 조인을 통해 department의 값을 employee 업데이트에 사용합니다 [35]. MySQL과 달리 두 테이블 모두를 나열하지 않고, 첫 줄에 업데이트 대상 테이블만 명시하는 것에 유의합니다.

DELETE 문 비교

- **DELETE ... JOIN**: MySQL은 UPDATE와 마찬가지로, 한 번의 DELETE 문에서 **여러 테이블을 동시에 삭제**하거나 다른 테이블과 조인하여 대상 행을 삭제할 수 있습니다. 다중 테이블 삭제 구문은 `DELETE t1, t2 FROM ... JOIN ... WHERE ...` 형태로 사용합니다. 예를 들어 **MySQL**에서 주문과 연관된 항목을 함께 지우는 경우:

```
DELETE orders, order_items
FROM orders
JOIN order_items ON orders.id = order_items.order_id
WHERE orders.id = 1234;
```

이 쿼리는 orders와 order_items를 조인한 후 해당 주문 ID에 해당하는 두 테이블의 행을 모두 삭제합니다. PostgreSQL에서는 한 DELETE문으로 여러 테이블을 직접 삭제하는 구문은 지원하지 않습니다. 따라서 위 작업을 하려면 **트랜잭션 내에서 두 개의 DELETE를 수행**하거나, **외래키 ON DELETE CASCADE** 제약을 활용해야 합니다. PostgreSQL에서는 일반적으로 **참조 무결성**을 활용해 자식 테이블(order_items)이 부모(orders)를 참조하도록 설정하고, 부모 삭제 시 자식을 자동 삭제(CASCADE)하는 식으로 처리합니다.

- **DELETE ... USING**: PostgreSQL은 DELETE에서 조인 조건을 사용할 때 `USING` 절을 지원합니다. 이는 UPDATE ... FROM과 유사하게, 삭제 대상 테이블 외에 추가로 조인할 테이블을 명시하는 방식입니다. 예를 들어 **PostgreSQL**에서 특정 고객(region이 'EU')의 주문을 삭제하려면:

```
DELETE FROM orders o
USING customers c
WHERE o.customer_id = c.id
 AND c.region = 'EU';
```

이렇게 `USING`을 사용하면 조인한 다른 테이블의 조건을 활용해 대상 테이블(o)의 행을 삭제할 수 있습니다. MySQL에서도 비슷하게 `DELETE o FROM orders o JOIN customers c ...` 형식으로 조인 조건 삭제가 가능합니다 (MySQL은 `USING` 키워드 대신 DELETE 대상 별칭을 FROM절 앞에 적습니다: `DELETE o FROM orders o JOIN customers c ON ... WHERE ...`). **주의점**: PostgreSQL `DELETE ... USING`에서도 UPDATE와 동일하게, 조인으로 한쪽에 여러 매칭이 있을 경우 예측 불가능하며, 보통 FK 관계상 1:Many 조인이므로 문제가 되진 않습니다.

- **DELETE ... LIMIT**: MySQL은 `DELETE FROM ... WHERE ... LIMIT n` 구문을 지원하여 일부 행만 삭제할 수 있습니다. PostgreSQL은 DELETE에 LIMIT이 없으므로, 필요하다면 `CTE + LIMIT` 트릭 (`WITH del AS (SELECT pk FROM table ... LIMIT n) DELETE FROM table WHERE pk IN (SELECT pk FROM del)`) 등을 사용합니다. PostgreSQL 13부터는 `DELETE ... RETURNING`으로 삭제된 행을 반환받을 수 있으므로 이를 활용해 로직을 짤 수도 있습니다.

- **예제 – DELETE 조인**:

- MySQL

```
-- region이 'ASIA'인 고객의 주문을 삭제 (주문과 관련 주문아이템 모두 삭제)
DELETE o, oi
FROM orders o
JOIN order_items oi ON o.id = oi.order_id
JOIN customers c ON o.customer_id = c.id
WHERE c.region = 'ASIA';
```

- PostgreSQL

```
BEGIN;
 DELETE FROM order_items oi
 USING orders o, customers c
 WHERE oi.order_id = o.id
  AND o.customer_id = c.id
  AND c.region = 'ASIA';
 DELETE FROM orders o
 USING customers c
 WHERE o.customer_id = c.id
  AND c.region = 'ASIA';
COMMIT;
```

PostgreSQL에서는 한 문장에서 두 테이블을 함께 삭제할 수 없으므로, 위와 같이 트랜잭션 내 두 문장으로 처리하거나 외래키 CASCADE를 설정해야 합니다. **참고**: FK로 order_items가 orders를 참조하고 ON DELETE CASCADE이면 orders만 지우면 order_items는 자동 삭제됩니다.

그 외 주의할 SQL 차이점

- **MySQL 전용 SQL 확장**: MySQL은 `SHOW TABLES`, `SHOW CREATE TABLE` 등의 **관리용 SQL**이나 `EXPLAIN EXTENDED`, `USE INDEX/FORCE INDEX` 힌트 등 특정 확장 문법이 있습니다. PostgreSQL에서는 `SHOW` 대신 `\d` 같은 **psql 메타커맨드**나 `information_schema` / `pg_catalog` 조회로 대체해야 하고, 옵티마이저 힌트는 공식적으로 지원하지 않으며(`ENABLE_*` 세션 설정이나 pg_hint_plan 확장 사용 정도만 가능) 기본은 **통계에 의한 자동 최적화**에 의존합니다.

- **데이터 타입과 함수 차이**: 세부적으로 문자열 처리나 날짜 함수 등도 차이가 있습니다. 예를 들어 MySQL의 `CONCAT_WS`, `DATEDIFF` 등은 PostgreSQL에서 함수명이 다르거나 존재하지 않을 수 있어 대체 함수 또는 표현식이 필요합니다. 문자열 비교 시 MySQL은 기본적으로 대소문자 무시(`varchar` 기본 collation이 case-insensitive)인 반면 PostgreSQL은 case-sensitive이므로, 대소문자 구분없는 검색은 `ILIKE` 연산자나 `citext` 타입을 사용해야 합니다.

결론적으로, **CRUD 및 JOIN 관련 SQL 문법은 큰 맥락에서는 유사**하지만, **PostgreSQL이 표준에 더 엄격**하고 **MySQL은 편의 확장이 많기 때문에**, 마이그레이션 시에는 MySQL에서 편의상 사용하던 비표준 구문이나 관대한 동작들을 모두 명시적으로 바꾸어야 합니다. 특히 **GROUP BY, JOIN, UPSERT, LIMIT 사용 부분**을 면밀히 검토해야 하며, 사소하게는 **식별자 인용과 대소문자** 처리도 신경 써야 합니다 [42]. 공식 문서와 호환 가이드 등을 참고하여 한 쪽 DB에만 있는 구문은 변환 공식을 찾아 적용하는 것이 좋습니다.

3. PostgreSQL의 EXPLAIN 실행계획 구조와 요소별 의미, 최적화 활용 방법

PostgreSQL의 실행계획(EXPLAIN 출력)은 **트리(tree) 구조**로 표현되며, 각 연산 노드들의 계층 구조와 예상 비용, 예상 행수 등을 보여줍니다. EXPLAIN은 쿼리를 **어떻게 실행할지** 계획한 내용을 알려주며, `EXPLAIN ANALYZE`를 사용하면 실제 실행하여 **실제 수행 시간과 처리 행수**까지 보여줍니다. 이를 해석함으로써 쿼리의 성능 병목을 파악하고 튜닝에 활용할 수 있습니다.

EXPLAIN 출력의 기본 구조

EXPLAIN의 출력은 들여쓰기 된 트리 형태의 **플랜 노드 목록**입니다. **상위 노드**는 최종 결과를 출력하는 단계이고, 하위로 들여쓴 **자식 노드**들은 상위 노드가 필요로 하는 중간 데이터를 어떻게 가져오는지 나타냅니다 [43] . 각 노드는 다음 정보를 포함합니다 [44] [45] :

- **연산 종류(노드 타입)**: 예를 들어 `Seq Scan`, `Index Scan`, `Bitmap Heap Scan`, `Nested Loop`, `Hash Join`, `Sort`, `Aggregate` 등이 표시됩니다 [44] . 이것만 봐도 어떤 방식으로 테이블을 읽고(join이나 sort를 하는지) 대략 파악할 수 있습니다.
- **대상 테이블/인덱스**: 연산이 가리키는 테이블명 혹은 인덱스명이 함께 나타나며, 어떤 개체에 대한 스캔인지 알 수 있습니다 [44] .
- **예상 비용 (cost)**: `cost= 시작값..종료값` 형태로 표시됩니다. 이는 PostgreSQL 옵티마이저의 **상대적 비용 단위**로, 시작값은 이 노드가 첫 번째 튜플을 출력하기까지의 비용(Startup Cost), 종료값은 모든 튜플을 처리하는 총 비용(Total Cost)입니다 [45] . 비용 단위는 임의이지만, **비교용으로 의미**가 있으며 보통 1.0은 한 페이지 순차 I/O 비용(기본 설정)입니다. 예를 들어 `cost=0.00..32.60`이라면, 이 노드는 32.60이라는 단위만큼의 연산 비용이 예상된다는 뜻입니다 [46] .
- **예상 행수 (rows)**: 해당 노드가 예측하는 출력 행의 개수입니다 [45] . 통계에 기반한 추정치이며, 옵티마이저가 플랜을 짤 때 사용합니다. 예를 들어 `rows=2260`이라면 이 노드 출력이 약 2260행일 것으로 예상한 것입니다. **참고**: 실제와 차이가 날 수 있습니다.
- **폭 (width)**: 한 행의 예상 크기(바이트)입니다 [47] . SELECT하는 컬럼의 평균 크기를 나타내며, 메모리 사용량 등을 가늠할 때 참고합니다.
- **필터/조건**: 해당 노드에서 사용된 조건이 있으면 `Filter: (조건식)` 또는 `Index Cond: ...`, `Join Filter: ...` 등의 형태로 나타납니다 [48] [49] . 예를 들어 Seq Scan 노드에 `Filter: (age > 30)`가 있다면 테이블 풀스캔 후에 age > 30을 필터링한다는 뜻이고, Index Scan의 `Index Cond: (id = 5)`는 인덱스를 사용해 id=5인 항목만 탐색한다는 의미입니다. Join의 경우 조인 조건은 대개 상위에 `Hash Cond:`나 `Merge Cond:`로 표시되고, 조인 후 추가로 걸리는 조건은 `Join Filter:`로 구분되기도 합니다 [48] [49] (INNER JOIN에서는 Join Filter와 Where Filter 차이가 없지만 OUTER JOIN에서는 Join Filter는 매칭에 쓰이고 Filter는 최종 필터링에 사용).

`EXPLAIN ANALYZE`를 사용하면 각 노드에 **실제 실행 통계**가 추가로 표기됩니다. 예를 들어 `(actual time=0.120..0.121 rows=1 loops=1)`과 같이 나타나는데 [50] : - **actual time=시작..끝(ms)**: 이 노드의 실행에 걸린 실제 시간(밀리초)입니다. 시작은 첫 출력까지 시간, 끝은 마지막 튜플 처리까지 누적 시간입니다. - **rows**: 실제 출력된 행의 개수 (실행 후 계측)입니다. - **loops**: 이 노드가 몇 번 반복 실행되었는지를 나타냅니다 [50] . 주로 상위 Nested Loop의 내부노드인 경우 loops>1이 되며, 예를 들어 loops=10이면 이 노드가 10번 재실행되었음을 의미합니다 (아래 Nested Loop 설명 참조).

EXPLAIN 출력은 **들여쓰기** 수준으로 부모-자식 관계를 표시합니다. 예를 들어:

```
Nested Loop  (cost=... rows=...)
   ->  Index Scan using idx_orders_customer on orders o  (cost=... rows=...)
   ->  Seq Scan on order_items oi  (cost=... rows=...)
```

이런 경우 `Nested Loop` 노드가 상위(root)이고, 그 아래 두 노드(`Index Scan`, `Seq Scan`)가 **브랜치(branch) 노드**들입니다 [43] . Nested Loop (루프 조인)은 두 개의 자식 노드를 가지며, 첫 번째가 outer, 두 번째가 inner 쪽입니다 [51] [52] . 최하위 들여쓰기 노드들은 더 이상 자식이 없는 **리프(leaf)** 노드들인데, 보통 실제 테이블을 읽는 **Scan 노드**들입니다 [43] .

주요 실행 계획 노드 유형과 의미

이제 EXPLAIN에서 흔히 볼 수 있는 **연산자 노드**들의 의미를 살펴보겠습니다:

- **Seq Scan (Sequential Scan)**: **순차 스캔**, 즉 테이블의 모든 행을 순차적으로 읽는 방식입니다 [53]. `Seq Scan on 테이블명`으로 표시되며, 해당 테이블에 조건없이 full scan하거나, 인덱스를 사용하지 못하는 경우 발생합니다. Seq Scan은 작은 테이블이거나 조건이 테이블 대부분을 차지할 때 효율적이지만, 대량의 불필요한 읽기를 할 수 있으므로 큰 테이블에서 선택도가 높은(일부만 해당되는) 조건에 Seq Scan이 나타나면 **잠재적 튜닝 포인트**입니다. (예: 수백만 건 테이블에서 `WHERE id = 5`인데 Seq Scan이면 인덱스가 없거나 사용 못 한 것이므로 인덱스 추가 검토).

- **Index Scan**: B-트리 등의 **인덱스를 탐색**하여 필요한 테이블 튜플을 읽는 방식입니다 [54]. `Index Scan using 인덱스명 on 테이블명 (Index Cond: ...)` 형태로 나타납니다. 인덱스를 통해 조건에 맞는 tuple pointer(CTID)들을 찾아가며, 각 튜플을 **즉시 테이블에서 랜덤 액세스**로 가져옵니다 [54] [55]. 예를 들어 `Index Scan using users_pkey on users (Index Cond: (id = 42))`는 `users_pkey` 인덱스로 id=42을 찾고 해당 행을 테이블에서 읽는 동작입니다. Index Scan은 **매우 선택적인 조건**(예: PK 조회나 소수 행만 추출)에 유리하며, 한 번에 한 튜플씩 처리하므로 결과 건수가 많아지면 오히려 순차 스캔+필터보다 느릴 수 있습니다. PostgreSQL은 비용 모델을 통해 어느 쪽이 나은지 판단합니다. **주의**: Index Scan 노드에는 추가로 `Filter: ...`가 붙을 수도 있는데, 이것은 인덱스 조건으로 못 걸은 추가 조건을 테이블 튜플 읽은 후 체크한다는 의미입니다. 인덱스가 완전히 조건을 커버 못 하면 이러한 filter 단계가 나오며, 이 경우 Index Scan임에도 불구하고 조건 일부는 여전히 튜플 단위 필터링이라 비용이 더 듭니다.

- **Index Only Scan**: **테이블에 액세스하지 않고 인덱스에서 바로 결과를 반환**하는 스캔입니다 [56]. 인덱스에 필요한 모든 컬럼이 포함되어 있고, 해당 튜플들이 모두 **진행중인 트랜잭션에 MVCC상 가시**한 경우(index 페이지의 visibility map 활용) 나타납니다. 예를 들어 `Index Only Scan using idx_users_email on users (Index Cond: (email = 'foo@bar.com'))`처럼 나오고, `Heap Fetches`라는 추가 통계가 표시되어 테이블 접근한 건수를 보여줍니다. Index Only Scan은 **커버링 인덱스** 역할을 하는 경우로, 테이블 I/O를 생략하여 상당한 성능 향상을 얻을 수 있습니다. 다만 쓰기 트랜잭션이 많은 테이블은 visibility map이 자주 꺼져 Index Only Scan 사용이 제한될 수 있습니다.

- **Bitmap Index Scan / Bitmap Heap Scan**: **비트맵 스캔**은 PostgreSQL이 다수의 인덱스 결과를 모아서 테이블을 효율적으로 읽는 기법입니다. `Bitmap Index Scan on ...`과 그 상위에 `Bitmap Heap Scan` 노드가 같이 나타납니다 [57] [58]. 예를 들어 `Bitmap Index Scan on idx_users_age (Index Cond: age > 30)`은 조건에 맞는 tuple 위치들을 모두 모으고, 상위 `Bitmap Heap Scan on users (Recheck Cond: age > 30)`가 그 비트맵(페이지별 정렬된 튜플 목록)을 이용해 **테이블 페이지들을 순차적으로 읽는** 동작을 합니다 [58]. 이렇게 하면 개별 튜플을 여기저기 랜덤하게 가져오는 대신, 필요한 페이지들을 모아 한 번씩 읽고 그 안에서 여러 튜플을 처리하므로 I/O 효율이 높아집니다 [55]. Bitmap Scan은 **중간 정도 선택도**(5~10% 정도) 조건에서 등장하며, **여러 인덱스 조건(AND/OR)** 조합 시 `BitmapAnd`, `BitmapOr` 노드와 함께 쓰여 둘 이상의 인덱스를 결합하는 경우도 있습니다 [59] [60]. Bitmap Heap Scan의 `Recheck Cond`는 인덱스에서 얻은 후보 튜플들을 실제 테이블에서 한 번 더 조건 확인하는 부분입니다 (인덱스에 없는 컬럼 조건이나 MVCC 검사). **참고**: Tom Lane의 설명에 따르면, "Index Scan은 튜플 하나 찾을 때마다 즉시 테이블을 방문하지만, Bitmap Scan은 인덱스로부터 모든 튜플 포인터를 가져와 정렬(비트맵)하고 테이블을 물리적 순서대로 방문한다" [55]. 이를 통해 Random I/O를 줄이고 순차 I/O로 최적화합니다.

- **Nested Loop (조인)**: **중첩 루프 조인**으로, 작은 테이블(또는 외부 쿼리 결과)을 기준으로 **반복해서** 다른 테이블을 조회하는 방식입니다 [61]. EXPLAIN에서는 `Nested Loop`로 표시되고, 두 개의 하위 노드를 갖습니다 (첫 번째가 **Outer** 소스, 두 번째가 **Inner** 소스). 동작은 Outer 노드에서 나온 각각의 행에 대해 Inner 노드를 실행하여 매칭되는 행을 찾는 것입니다 [52] [62]. 예를 들어 Outer가 orders 테이블 10건을 주고, Inner가 order_items에 대해 각 주문ID로 Index Scan을 했다면, Inner Index Scan은 10번 반복(loop=10) 실행되고

그 결과가 Nested Loop로 결합됩니다 [52] [63]. Nested Loop는 **소규모 집합과의 조인**이나 **인덱스를 통한 키 조인**에 유리합니다. 만약 Outer가 100만 건이고 Inner도 100만 건이라면 100만100만 번 탐색은 엄청나게 느리므로, 이런 경우 Nested Loop는 피해야 합니다. PostgreSQL은 통계로 row수가 많으면 Hash Join 등으로 우회하지만, 통계가 부정확하면 Nested Loop로 잡혀 심각한 성능 문제가 발생할 수 있습니다. 실무 활용: EXPLAIN ANALYZE에서 Nested Loop의 Inner 노드에 `loops=N`으로 많이 반복되고 시간이 크다면, 조인 전략을 점검해야 합니다. 해결책으로는 조인 키에 인덱스를 추가하여 각 루프 비용을 낮추거나, 아예 Hash Join이 되도록 쿼리를 다시 쓰거나 통계 갱신/파라미터 튜닝을 고려합니다. Nested Loop는 소량 출력에 좋은 조인*이므로 대량일 경우 플랜이 잘못된 것일 수 있습니다.

- **Hash Join**: **해시 조인**은 한 테이블(또는 하위 계획)의 결과를 메모리에 해시 테이블로 만들고, 다른 테이블을 순차적으로 읽으면서 해시 테이블을 조회하여 매칭시키는 조인입니다 [64] [65]. EXPLAIN에서 `Hash Join`으로 표시되고, 보통 하위에 `Seq Scan ...` -> `Hash` 로 해시테이블 빌드 단계와, 다른 쪽 `Seq Scan` (혹은 인덱스/소트 등)으로 프로브하는 단계로 나타납니다 [64] [66]. 예를 들어:

```
Hash Join  (cost=226.23..709.73 rows=100 width=...)
  Hash Cond: (t2.unique2 = t1.unique2)
  ->  Seq Scan on t2 ...
  ->  Hash  (cost=224.98..224.98 rows=100 ...)
        ->  Bitmap Heap Scan on t1 ... (rows=100)
```

이런 플랜은 t1에서 조건에 맞는 100개를 Bitmap Scan으로 가져와 해시 테이블을 만들고, t2 전체를 Seq Scan하면서 해시 조건으로 매칭하는 구조입니다 [67] [68]. Hash Join은 **대량의 조인에 효율적**이며, 조인 키에 인덱스가 없어도 작동합니다. 다만 해시 테이블 크기만큼 메모리가 필요하며, 크게 부족하면 디스크 스필이 발생할 수 있습니다. 해시 조인은 대개 통계상 출력이 꽤 많거나, 인덱스가 없는 경우 옵티마이저가 선택합니다. EXPLAIN에서 Hash Join 노드의 자식으로 `Hash` 노드가 보이면 그것이 build 측을 의미하고, Hash Cond는 매칭 조건입니다. **활용**: 만약 Hash Join의 실제 해시 테이블 빌드(rows)와 프로브(rows)가 예상보다 훨씬 크다면 work_mem 등을 조정해 메모리를 충분히 주거나, 또는 Hash Join 대신 Merge Join/인덱스 조인을 고려해볼 수 있습니다. (Hash Join이 튜닝 포인트가 되는 경우는 메모리 부족으로 디스크 사용이 일어날 때입니다. `EXPLAIN (ANALYZE, BUFFERS)`로 BufUsage 등을 보면 Disk I/O 여부를 알 수 있습니다.)

- **Merge Join**: **정렬 병합 조인**은 양쪽 입력이 **조인 키로 정렬된 상태**에서 병합하는 조인입니다 [69] [70]. EXPLAIN에 `Merge Join`으로 나타나고, `Merge Cond: (A.key = B.key)`와 함께, 각 하위에 정렬된 출력이 보장되는 노드(예: 인덱스 스캔이나 `Sort` 노드)로 나타납니다 [69]. Merge Join은 두 입력을 병렬로 훑으며 키를 비교해가며 매칭시키므로, **양 입력 모두 대량이지만 정렬만 되어 있다면** 매우 빠릅니다. 정렬되어 있지 않으면, 옵티마이저는 정렬 비용까지 고려하여 Merge Join을 선택할지 결정합니다. 예를 들어:

```
Merge Join  (cost=0.56..233.49 rows=10 ...)
  Merge Cond: (t1.unique2 = t2.unique2)
  ->  Index Scan using t1_unique2 on tenk1 t1 ... (Filter: unique1 <
100)
  ->  Index Scan using onek_unique2 on onek t2 ...
```

위 플랜은 두 테이블 모두 조인 키(unique2) 기준 인덱스 스캔으로 이미 정렬된 결과를 제공하여 Merge Join을 수행합니다 [69] [71]. 만약 정렬 비용이 들어간다면 `Sort` 노드가 Merge Join 아래에 붙어 나올 것입니다. Merge Join은 **양쪽이 모두 큰 테이블이고, 조인에 사용할 인덱스가 둘 다 있을 때** 특히 유용합니다. 또는 해시 조인을 메모리 때문에 못 쓰는 경우(예: 너무 큰 조인) work_mem 제한으로 Merge Join+Sort가 선택되기도 합니다. **활용**: Merge Join이 나오려면 양측 정렬이 중요한데, 만약 `ORDER BY`와 조합된 조인이라면

Merge Join이 이점이 있습니다. 또한 `EXPLAIN`으로 Merge Join을 강제로 써보고 싶다면 다른 조인 전략을 끄는 방법도 있습니다(`SET enable_hashjoin = off;` 등) [72].

- **Sort**: 정렬 노드입니다. 보통 `Sort  (cost=...) (Sort Key: ...)` 형태로 나타나며, 바로 아래에서 올라온 결과를 해당 키로 정렬합니다 [73]. 정렬은 **비파이프라인(pipeline되지 않는)** 연산으로, 전체를 모아 정렬하므로 메모리 사용량이 크고, `LIMIT`가 없으면 끝까지 다 수행해야 결과가 나옵니다 [74]. EXPLAIN ANALYZE에서 Sort 노드의 Memory 정보가 나오며, work_mem을 넘으면 Disk를 사용합니다 (실행 시 "(external sort)" 등으로 표시). 만약 ORDER BY가 인덱스를 통해 이미 충족되면 Sort 노드는 나타나지 않고, 인덱스 스캔에서 `*(costX..costY) rows=N*`처럼 비용이 계산됩니다 [75]. Sort가 큰 비용으로 예상되면 인덱스 활용이나 Merge Join 등의 대안도 고려합니다.

- **Aggregate (GroupAggregate / HashAggregate)**: **집계 노드**로, GROUP BY나 집계 함수가 있을 때 나타납니다. 두 종류가 있는데, 입력이 미리 정렬되어 있을 경우 **GroupAggregate**(=Sorted Aggregate)을 수행하고, 정렬되어 있지 않으면 해시 테이블을 사용한 **HashAggregate**를 수행합니다 [76] [77]. GroupAggregate는 `Group Key: ...`로 표시되고, 데이터를 정렬된 순서로 받아서 같은 키 묶음을 차례로 집계합니다 (메모리 효율적) [76]. HashAggregate는 `HashAggregate` 노드로 표시되며, 내부적으로 해시 테이블에 그룹 키별로 누적합니다 [77]. HashJoin과 마찬가지로 HashAggregate도 메모리 초과 시 디스크 스필이 생길 수 있습니다. EXPLAIN ANALYZE 시 HashAggregate의 Disk 사용 여부도 나오므로, 많은 그룹이 있을 땐 work_mem 설정을 유념해야 합니다.
  PostgreSQL은 큰 그룹 집계시 HashAggregate를 선호하는 경향이 있습니다. 만약 ORDER BY와 GROUP BY가 같이 있다면 Incremental Sort + GroupAggregate 같은 복합 플랜도 있습니다.

- **Limit**: 출력 행수를 제한하는 노드입니다. `Limit (rows=N)` 형태로 나타나며, **하위 노드의 처리를 N개 행 이후 중단**시킵니다 [78]. LIMIT은 윗단에서 보면 O(N)만 처리하고 끝나므로 효율적입니다. 다만 **하위 노드가 Sort같이 한 번에 다 모아야 하는 경우** Limit이 있어도 이미 비용을 치른 뒤라 의미가 작아집니다 [79]. 그래서 PostgreSQL은 Limit과 Sort가 함께 있을 때 **Top-N 최적화**를 시도하기도 합니다 (Incremental Sort 등).
  EXPLAIN에서 Limit 노드는 자식 노드에 `rows`보다 큰 출력이 예상되어도 일단 N까지만 가져오고 중단한다고 알려주므로, large query에 작은 LIMIT이 있으면 주로 인덱스 통해 ORDER BY를 만족시키는 계획을 짜서 Sort를 회피합니다 [80] [81].

- **Materialize**: **매테리얼라이즈 노드**는 중간 결과를 **임시 메모리 테이블에 저장**하여 여러 번 재사용할 때 등장합니다. 주로 Nested Loop 조인에서 **Inner쪽 서브플랜이 반복될 때** 비용절감을 위해 첫 실행 결과를 캐싱하는 용도입니다 [82] [83]. EXPLAIN에서는 `Materialize`로 표시됩니다. 예를 들어 Nested Loop 조인에서 inner가 Materialize로 감싸여 있다면, 한 번만 실행해서 메모리에 올려두고 Outer 루프 반복 시 재사용함을 뜻합니다 [82]. Materialize는 메모리 소비와 약간의 CPU 소비가 있지만, **디스크 I/O를 크게 줄일 수 있다면** 자동으로 사용됩니다. 개발자가 직접 등장시킬 수는 없고, 옵티마이저가 판단하여 넣는 노드입니다.

이 밖에도 **Subquery Scan**, **CTE Scan**, **Append** (여러 하위 계획 결과를 이어붙임, 파티션 테이블에서 주로 등장), **Gather/Gather Merge** (병렬 쿼리에서 worker들의 결과 수집) 등 여러 노드 타입이 존재하지만, CRUD 튜닝 관점에서 핵심적인 것은 위에 설명한 부분들입니다.

실행계획을 활용한 실무적 최적화 방법

EXPLAIN (또는 EXPLAIN ANALYZE)를 통해 얻은 실행계획을 **해석**하여 쿼리나 인덱스 튜닝에 활용하는 방법을 정리합니다:

1. **주요 비용 병목 파악**: `EXPLAIN ANALYZE` 출력에서 **가장 비용이 많이 든**(또는 시간이 오래 걸린) 노드를 찾습니다. 일반적으로 **상위 노드의 Total Cost**가 전체 쿼리 비용이며, ANALYZE를 켠 경우 각 노드의 `(actual time=... loops=...)`로 실제 시간이 표시됩니다. 예를 들어 `Nested Loop`가 있고 그

아래 `Seq Scan`이 매우 많은 loops로 반복되며 time 합산이 큰 경우, 그 부분이 병목입니다. 혹은 `Hash Join`의 해시 빌드나 `Sort` 노드가 큰 메모리와 시간을 잡아먹고 있으면 거기가 문제입니다. **EXPLAIN 출력에서 cost 숫자**는 비교 지표로 사용할 수 있는데, cost가 유난히 큰 부분이 있다면 튜닝 대상입니다 (예: cost 수십만 단위). 다만 cost는 추정치이므로, ANALYZE의 실제 시간 데이터를 함께 봐야 합니다. 예: 아래와 같은 (가상의) EXPLAIN ANALYZE:

```
Nested Loop  (cost=1000..500000 rows=1000000) (actual time=0.5..1200.0
rows=1000000 loops=1)
  ->  Seq Scan on big_table ... (actual time=0.3..300.0 rows=100000
loops=1)
  ->  Index Scan on other_table ... (actual time=0.002..0.005 rows=10
loops=100000)
```

여기서 Nested Loop의 inner Index Scan이 loops=100000번 돌아 1200ms 중 대부분을 차지한다면, 이 Nested Loop를 의심해야 합니다.

2. **인덱스 활용 여부 점검**: Seq Scan 노드가 큰 테이블에 대해 발생하고 있는데 조건절에 적절한 인덱스가 있다면, **왜 인덱스를 쓰지 않았는지** 파악합니다. 인덱스를 안 쓴 이유는:

3. 인덱스가 존재하지 않는다 ⇒ **인덱스 생성**을 고려.
4. 인덱스는 있으나, 너무 많은 건수를 읽어야 해서 옵티마이저가 Seq Scan이 유리하다고 판단 ⇒ 이 경우 실제로 대부분 행이 해당되면 Seq Scan이 맞습니다. 하지만 통계가 부정확해 잘못 판단했을 수도 있으므로, **ANALYZE로 통계 갱신** 후 플랜 재확인.
5. 인덱스는 있으나 조건이 범위 조건 등으로 선택도가 낮다 ⇒ 부분 범위라도 인덱스 태워볼지 강제해볼 수 있지만 보통 옵티마이저 판단이 맞습니다.
6. 함수나 타입 캐스팅으로 인해 인덱스를 못 타는 경우 ⇒ **인덱스 표현식을 맞추거나 함수 인덱스** 등을 고려.

예를 들어 `WHERE date(created_at) = '2023-01-01'` 이런 쿼리는 인덱스(created_at)가 있어도 함수로 감싸져 못 쓰므로 Seq Scan이 나옵니다. 이때 EXPLAIN에서 Seq Scan + Filter로 보일 것이고, 해결책은 **인덱스 적용 가능하게 쿼리 수정**(예: `created_at >= '2023-01-01' AND created_at < '2023-01-02'`)하거나 **함수 기반 인덱스** (`CREATE INDEX ON tbl(date(created_at))`)를 만드는 것입니다.

인덱스 스캔 최적화: EXPLAIN에서 `Index Cond`와 `Filter`를 구분해 보여주므로, Filter에 남은 조건은 인덱스로 못 걸린 것입니다. 인덱스 복합 컬럼 순서나, 조건 간소화 등을 통해 Filter를 줄이고 Index Cond로 포함시키는 방향을 생각합니다. 또한 `Index Only Scan`이 가능할 경우, SELECT 절에 불필요한 컬럼을 빼서 인덱스만으로 처리되게 유도할 수 있습니다 (혹은 covering index 추가).

1. **조인 순서 및 방식 확인**: JOIN이 여러 개 있는 복잡한 쿼리의 경우, **어떤 순서로 어떤 방식**으로 조인이 되는지 EXPLAIN으로 알 수 있습니다.
2. 조인 순서가 비효율적이면 (예: 매우 큰 테이블 둘을 먼저 조인 후 작은 테이블과 조인), 실행계획이 비효율일 수 있으므로 **JOIN 순서를 힌트 없이 바꾸도록 쿼리 리팩토링**을 검토합니다. PostgreSQL 옵티마이저는 일반적으로 통계에 근거해 최적 순서를 고르지만, 통계가 부準확하면 잘못 선택할 수 있습니다. 이런 경우 FROM 절 서브쿼리로 의도적으로 순서를 묶어주거나, 통계를 조정(`ALTER TABLE ... SET STATISTICS`)해서 좀 더 정확하게 만들 수 있습니다.
3. 조인 방식(Nested Loop / Hash Join / Merge Join)이 적절한지 봅니다. **대용량 조인에 Nested Loop**가 나오면 위험 신호일 수 있습니다. 이때 해당 조인 조건 필드에 인덱스가 없으면 생긴 일일 수 있으니 **인덱스 추가**로 해결하거나, 인덱스로도 힘든 크기면 차라리 Hash Join이 낫습니다. Hash Join이 안 나온 이유는 작은 테이블 쪽 통계가 매우 작게 잡혀 Nested가 싸게 평가됐거나, enable_hashjoin이 false이거나, work_mem이 너무 작아서 해시가 기피되었을 수 있습니다. 통계 갱신과 파라미터 튜닝을 고려해보고, 그래도 Nested

Loop가 계속 나오면 USE HASH JOIN 전략으로 강제 실행(위에서 언급한 enable 설정)시켜 비교해 봅니다 [84].

4. **Hash Join vs Merge Join**: Hash Join은 메모리 용량이 충분한지 확인합니다. 실제 실행 시 `Hash:` 노드에 `(memory: 2048kB)` 같은 출력이 있고, Hash Join 자체에 "Buckets: 1024kB Batches: 1 Memory Usage: ..." 등이 나옵니다. 만약 Batches가 1보다 크다면 디스크 분할이 일어났다는 뜻입니다. 이러면 성능이 떨어지므로 **work_mem 증가**로 메모리에 올릴 수 있도록 합니다. Merge Join은 주로 양쪽 Index Scan 비용이 높지 않을 때 나오므로, Merge Join이 안 나오고 Hash Join이 나오는데 Hash Join이 느리다면, 인덱스를 활용하는 쿼리로 변경하거나 (예: 미리 정렬된 임시 테이블과 조인) 고려할 수 있습니다.

5. **Join Filter**: OUTER JOIN의 경우 EXPLAIN에 `Join Filter`와 `Filter`가 따로 나옵니다 [49]. Join Filter는 ON 절 조건인데 OUTER JOIN에서는 매치 안 돼도 튜플이 살아남을 수 있고, Filter는 WHERE 조건으로 null-extended를 걸러냅니다. 이러한 구조를 이해하면, OUTER JOIN을 INNER JOIN으로 바꾸고 WHERE로 이동 가능 여부 등을 판단할 수 있습니다. (원칙적으로는 SQL변경이지만, 비즈니스 로직상 문제 없으면 내부 조인으로 단순화하면 속도가 향상될 수 있습니다.)

6. **쿼리 재작성 및 힌트**: PostgreSQL은 힌트가 공식 지원되지 않지만, **SQL 재구성**으로 옵티마이저에 영향을 줄 수 있습니다. 예를 들어 쿼리를 둘로 나눠 WITH로 분리하면 해당 CTE는 독립 실행되어 옵티마이저가 조인 순서를 바꾸지 못하게 할 수 있습니다 (이건 성능엔 오히려 제약일 수 있으나 특정 계획을 강제하는 효과는 있음). 또는 `UNION ALL`로 OR 조건을 분해하거나, INDEX를 타게 함수 대신 case when 등을 쓰는 등으로 쿼리를 다시 작성하면 플랜이 개선될 수 있습니다. 이러한 변경은 **EXPLAIN로 확인하며 반복**해야 합니다. 필요한 경우 pg_hint_plan 확장을 설치하면 주석 힌트로 제어할 수 있지만, 가능하면 통계와 쿼리 자체로 해결하는 것이 바람직합니다.

7. **실제 실행 정보 활용**: `EXPLAIN ANALYZE`는 실제 runtime을 보여주므로, **예상치 대비 실제치 차이**를 확인하는 것이 중요합니다.

8. **Rows 추정 오차**: `actual rows` vs `estimated rows`(rows=)를 비교합니다. 만약 크게 차이가 난다면 옵티마이저가 잘못된 가정으로 플랜을 짰을 수 있습니다. 예를 들어 어떤 노드는 rows=10 이라 예상했는데 actual rows=10000 이었다면, 그 하위에서 통계가 부정확한 것입니다. 이런 경우 `ANALYZE`로 최신 통계를 확보하거나, complex한 조건이라 통계로 잡히지 않는 경우 **상대적으로 더 안정적인 플랜**을 선택하도록 유도해야 합니다. (예: OR 조합 등으로 추정 어려운 경우 Nested Loop 대신 Hash Join을 쓰도록 함).
9. **Buffers 등의 추가정보**: `EXPLAIN (ANALYZE, BUFFERS)`를 하면 각 노드별로 공유 버퍼 hit/miss, 디스크 read 등의 정보가 나옵니다. 이를 통해 쿼리가 CPU바운드인지 I/O 바운드인지 파악 가능합니다. 만약 특정 노드에서 disk read(O)나 temp read/write(O) 등이 보이면, **I/O 병목**이므로 인덱스로 I/O 줄이기나 work_mem 증설로 메모리 처리 유도 등을 고려합니다. CPU 바운드이면 반복 연산을 줄이는게 관건이므로 알고리즘(조인 방식 등)을 바꾸거나 불필요 연산 제거를 생각합니다.

10. **Planning Time vs Execution Time**: EXPLAIN ANALYZE 결과 맨 마지막에 Planning Time과 Execution Time이 표시됩니다 [85]. 보통 Execution이 훨씬 길지만, 수천 테이블 조인 등 극단적 경우 Planning도 무시못할 수 있습니다. 일반적 OLTP 쿼리에서는 planning << execution입니다.

11. **인덱스 및 스키마 튜닝**: 실행계획 분석 결과 인덱스가 필요하다고 판단되면, 적절한 인덱스를 생성하고 다시 EXPLAIN으로 사용되는지 확인합니다. 때로는 다중컬럼 인덱스가 효과적일 수도 있고, 기존 인덱스 중 사용되지 않는 것이 있으면 정리할 수도 있습니다. 또한 파티셔닝된 테이블이라면 `Append` 노드로 여러 파티션을 스캔하는지, Pruning이 제대로 되는지 등을 확인합니다. Pruning이 안 되면 파티션 키 조건이 옵티마이저에 전달 안 된 것이므로 쿼리 수정이나 파티션 키 함수일치 등을 해야 합니다.

12. **사례 예시**:

13. 예시1: 한 보고서 쿼리가 10초 이상 걸려 프로파일링해보니, EXPLAIN ANALYZE에서 `Nested Loop` 안에 `Seq Scan`이 100만 loops 이상 수행되는 것이 확인되었습니다. 이는 작은 테이블과 큰 테이블 조인 시 작은 쪽을 Outer로 잡고 큰쪽에 인덱스가 없어 발생한 문제였습니다. 해결책으로 큰 테이블의 조인 키에 인덱스를 생성했더니, 다음 실행부터 `Hash Join`으로 바뀌며 0.5초로 단축되었습니다. **교훈**: Nested Loop + 대량 loops -> 인덱스 없음을 의미, 인덱싱으로 개선.

14. 예시2: 복잡한 SELECT에서 여러 테이블을 JOIN하고 GROUP BY를 하는데 매우 느렸습니다. EXPLAIN 보니 `HashAggregate`가 Disk spill을 하고 있었습니다. 그룹 수가 매우 많고 work_mem이 기본 4MB라 해시 버킷을 나눠 처리하느라 Disk I/O가 발생한 겁니다. 이때 해당 서버의 work_mem을 16MB로 늘리고 쿼리를 재실행하니 HashAggregate가 메모리 내에서 처리되고 전체 쿼리가 3배 이상 빨라졌습니다. **교훈**: HashAggregate/HashJoin에서 스필이 보이면 work_mem 조정.

15. 예시3: 어떤 SELECT ... WHERE ... 쿼리가 1분 이상 걸렸는데, EXPLAIN ANALYZE 결과 `Bitmap Heap Scan`으로 100만 행을 필터링하고 있었습니다. 조건 중 하나가 함수 `LOWER(email) = 'abc@xyz.com'`였고, 이메일 컬럼에 인덱스가 있었지만 함수 때문에 못 쓰고 전체 스캔 후 필터링했던 것입니다. **해결**: 함수 인덱스 `CREATE INDEX ON users ((LOWER(email)));`를 만들고 쿼리를 다시 실행하니, `Index Scan`으로 바뀌며 1초 내로 응답했습니다. 이처럼 **함수나 연산으로 인덱스를 못 타는 경우** EXPLAIN의 Filter에 해당 조건이 드러나므로, 이를 발견하고 대응하는 것이 중요합니다.

16. **모니터링과 반복 튜닝**: EXPLAIN은 개발 및 운영 단계에서 쿼리 최적화에 지속 활용해야 합니다. 특히 **새로운 쿼리를 작성할 때** 미리 EXPLAIN으로 계획을 살펴보고 인덱스 사용 여부를 검증하는 습관이 좋습니다. 또한 PostgreSQL 13+에서는 `auto_explain` 모듈을 사용하여 일정 시간 이상 걸린 쿼리의 실행계획을 로그로 남길 수 있으므로, 프로덕션 환경에서 튜닝 대상을 찾는 데 유용합니다. **실행계획 관리**를 통해 병목쿼리를 선제 파악하고, 인덱스나 쿼리를 개선함으로써 시스템 성능을 유지할 수 있습니다.

요약하면, **EXPLAIN/EXPLAIN ANALYZE는 PostgreSQL 쿼리 최적화의 핵심 도구**입니다. Seq Scan, Index Scan, Nested Loop, Hash Join 등의 개별 요소를 이해함으로써 우리는 **쿼리가 어떻게 동작하고 어디서 비용이 드는지** 파악할 수 있고, 그 정보를 토대로 **인덱스 설계, 쿼리 구조 최적화, 파라미터 튜닝**을 수행하게 됩니다. 항상 실제 실행 결과와 비교해가며, 필요한 경우 PostgreSQL 공식 문서나 **실행계획 분석 툴**(예: pgAdmin의 Graphical Explain, 또는 pevz) 등을 활용해 꾸준히 개선해 나가는 것이 좋습니다.

4. Java Spring Boot (v3.5+) 환경에서 PostgreSQL 커넥션 풀링 전략

Spring Boot 3.x 환경에서는 **HikariCP**가 기본 커넥션 풀(DataSource)로 사용됩니다. **커넥션 풀링**은 데이터베이스 연결을 재활용하여 애플리케이션의 DB 접속 성능을 높여주므로, PostgreSQL을 사용할 때도 HikariCP의 적절한 설정이 중요합니다. 여기서는 **HikariCP 설정 방법**, **커넥션 누수(leak) 방지**, **커넥션 풀 사이즈 튜닝 전략**, **모니터링 도구** 등을 실무 관점에서 설명합니다.

HikariCP 설정 및 Spring Boot 적용 방법

- **Spring Boot 기본값**: Spring Boot에서 `spring-boot-starter-jdbc`나 `spring-boot-starter-data-jpa` 의존성을 추가하면 HikariCP가 자동 선택됩니다 [86] [87]. 기본 설정으로 최대 풀 크기 10개 등 **상식적인 기본값**들이 적용되어 있어 소규모 애플리케이션은 손댈 필요 없이 동작합니다. 그러나 트래픽 규모나 DB 성능에 따라 **튜닝**이 필요할 수 있습니다.

- **설정 구성**: Spring Boot에서는 `application.properties` 또는 `application.yml`을 통해 HikariCP를 설정할 수 있습니다. 프로퍼티 접두사는 `spring.datasource.hikari.*`이며, HikariCP의 `HikariConfig` 빈의 속성과 1:1로 매핑됩니다 [88] [89]. 주요 설정 항목:

- `spring.datasource.hikari.maximum-pool-size`: **커넥션 풀의 최대 커넥션 수**입니다. 기본값은 10이며, 동시 연결이 많은 경우 늘릴 수 있습니다 [90].
- `spring.datasource.hikari.minimum-idle`: **유지할 최소 유휴 커넥션 수**입니다. 기본값은 maximum과 동일하게 설정되므로 명시하지 않으면 풀 크기만큼 만들어놓지만, 적절히 낮춰서 사용하지 않는 연결은 줄일 수 있습니다 [91].
- `spring.datasource.hikari.connection-timeout`: **풀에서 커넥션을 빌려올 때 최대 대기 시간**(ms)입니다. 기본 30000(30초)이며, 이 시간 내에 커넥션을 얻지 못하면 SQLException이 발생합니다. 애플리케이션 요구에 따라 너무 길면 줄이고, 짧게해서 타임아웃 빠르게 감지할 수도 있습니다.
- `spring.datasource.hikari.idle-timeout`: **유휴 연결을 풀에서 제거하는 시간**(ms)입니다. 기본 600000(10분) [77]. minimumIdle보다 초과되는 유휴 연결이 idle-timeout 이상 놀고 있으면 풀에서 닫습니다. 너무 짧게 잡으면 커넥션이 빈번히 생성/종료되고, 너무 길면 쓰지 않는 연결이 오래 살아있습니다. 기본값 10분이 무난하며, PG 서버의 idle timeout보다 짧거나 비슷하게 맞추면 좋습니다.
- `spring.datasource.hikari.max-lifetime`: **커넥션의 최대 수명**(ms)입니다. 기본 1800000(30분) [92]. 이 시간이 지나면 커넥션은 풀에서 제거되고 새로 생성됩니다. 이는 DB 서버 쪽에서의 타임아웃 또는 장기간 열린 커넥션의 문제(메모리 누수 등)를 예방합니다. 일반적으로 30분~1시간 사이로 설정합니다. (주의: DB 서버가 `PG_IDLE_TIMEOUT` 같은 설정이 있다면 그보다 약간 작은 값으로 설정 권장).
- `spring.datasource.hikari.pool-name`: 풀의 이름입니다. 기본은 `HikariPool-1` 등으로 지정되며, 로그나 JMX 모니터링 시 식별용이므로 명시적으로 부여하기도 합니다.
- `spring.datasource.hikari.leak-detection-threshold`: **커넥션 누수 감지 시간**(ms)입니다. 이 시간을 넘도록 반환되지 않은 커넥션이 있으면 경고 로그를 남겨줍니다 [93] [94]. 기본값 0 (비활성)이며, 개발/테스트 환경에서 적당한 시간 (예: 2000ms = 2초)으로 설정하면 누수 여부를 조기에 발견할 수 있습니다 [95]. 운영에서는 쿼리가 2초 넘게 걸리는 정상 케이스도 있을 수 있으므로 false positive 가능성에 유의해야 합니다 [96]. HikariCP의 LeakDetection은 **Connection을 획득하고 일정 시간 내 반환하지 않으면** 스택트레이스를 로깅해주므로, 누수가 의심될 때 활성화하는 유용한 디버깅 수단입니다.

아래는 예시 **application.yml** 설정:

```
spring:
  datasource:
    url: jdbc:postgresql://dbhost:5432/mydb
    username: myuser
    password: secret
    driver-class-name: org.postgresql.Driver
    hikari:
      pool-name: MyAppHikariPool
      maximum-pool-size: 20
      minimum-idle: 5
      idle-timeout: 300000      # 5분
      max-lifetime: 1800000     # 30분
      connection-timeout: 30000  # 30초
      leak-detection-threshold: 2000  # 2초 (개발환경에서 설정)
```

위 설정은 풀 크기 최대 20, 유휴 5개 유지, 5분 이상 놀면 제거, 30분마다 커넥션 갱신, 얻기 대기 최대 30초, 2초 넘게 사용 중인 커넥션 있으면 로그 경고 등의 의미입니다.

- **베스트 프랙티스**: HikariCP는 기본값이 상당히 잘 튜닝되어 있지만, **사용 패턴에 따라 파인튜닝**이 필요합니다 [97].

- 트래픽이 스파이크하는 경우 `minimum-idle`를 너무 낮게 잡으면 갑자기 커넥션을 여러 개 생성하느라 지연이 생길 수 있습니다. 따라서 예상 피크에 대비해 어느 정도 커넥션을 미리 확보해 두는 것이 좋습니다 (minIdle을 적절히).
- 반대로 항상 유휴가 많은 환경이면 minIdle을 낮춰서 리소스 낭비를 막습니다.
- `maximum-pool-size`는 너무 크면 오히려 **DB서버에 부하**를 한꺼번에 줄 수 있습니다 (PostgreSQL는 연결당 프로세스를 띄우므로, 수백개 프로세스가 동시에 쿼리하면 컨텍스트 스위칭/메모리 부담) [98]. HikariCP 설계자도 "(코어 수 * 2) + 디스크 스핀들 수" 정도를 적정 커넥션 수로 제안한 바 있습니다 [99]. 예를 들어 8코어 머신에 SSD 하나면 (82 + 1) = 17개 정도가 적당한 풀 크기로 언급됩니다. 물론 이것은 일반론이고, 실제로는 동시 요청 수, 쿼리 종류(읽기 많은지, I/O 대기 많은지), DBMS의 max_connections 설정* 등을 종합 고려해야 합니다.
- PostgreSQL 기본 `max_connections`는 100 (superuser reserved 제외)인데, 너무 풀을 크게 잡으면 몇 개의 애플리케이션 인스턴스만으로 DB 커넥션을 소진할 수 있습니다. 예를 들어 풀 50으로 3대 돌리면 150 연결이라 기본 100을 넘습니다. **풀 사이즈 합계가 DB max_connections 이하**인지 확인하고, 필요하면 DB 설정도 늘려야 합니다. 또는 더 근본적으로 pgBouncer 같은 **외부 풀러**를 사용해 수백개 애플리케이션 연결을 수십개 DB백엔드로 매핑하는 것도 방법입니다 (PG 자체 프로세스 연결 오버헤드 개선).

한 줄 요약: **"풀 크기는 클수록 좋지 않다"**. 너무 작으면 대기(queueing)이 발생하지만, 너무 크면 DB에 과부하가 갑자기 몰릴 수 있습니다. 적정 수준을 결정하기 위해 모니터링을 통해 조정해나가는 것이 중요합니다 [100]. DigitalOcean 가이드에서는 "가용 DB 연결의 절반 정도로 시작해 보고, CPU 사용률 등을 보며 조절"을 권장합니다 [100].

커넥션 누수 방지 (Connection Leak Prevention)

- **자동 리소스 해제**: Java에서 JDBC Connection을 열고 닫지 않으면 누수가 발생합니다. Spring JDBC나 JPA 사용 시 일반적으로 **프레임워크가 Connection을 풀에 반환**하므로 개발자가 직접 close할 일이 드뭅니다. 하지만 간혹 **사용자가 JDBC Connection을 직접 얻어 쓸 때** (예: `DataSource.getConnection()`) try-with-resources 또는 명시적 `close()`를 호출해야 합니다. Spring Boot 환경에서는 보통 `@Transactional`을 사용하거나, `JdbcTemplate`/`EntityManager` 등이 알아서 처리를 해주지만, **외부 API에서 Connection을 획득**하거나 **ResultSet/Statement를 제대로 닫지 않는** 실수가 있을 수 있습니다. 이러한 **코드 상의 실수**를 방지하는 것이 1차적 대책입니다 (코드 리뷰, 교육 등).

- **HikariCP Leak Detection**: 앞서 언급한 `leakDetectionThreshold` 설정을 활용하면, 정해진 시간 이상 반환되지 않는 커넥션을 **경고 로그**로 추적할 수 있습니다 [101]. 예를 들어 5초로 설정하면, Connection을 빌려간 후 5초 내 반환 안 하면 로그에 "Connection leak detection"과 함께 그 Connection을 얻은 스택 트레이스를 찍어줍니다. 이를 통해 어느 비즈니스 로직에서 커넥션이 누락되었는지 찾아낼 수 있습니다 [94]. 실무에서는 개발/QA 환경에 이 값을 짧게 설정해두고 테스트하면서 누수를 조기에 발견합니다. 운영에서도 너무 짧지 않은 임계값(예: 60초)을 설정해두어 장시간 반환 안 되는 케이스를 감지할 수 있지만, 쿼리가 정말 오래 걸리는 경우 혼동될 수 있으므로 상황에 맞게 사용합니다 [102].

- **Connection 테스트 및 Validation**: HikariCP는 기본적으로 **커넥션이 고장났을 때** (예: DB 재시작 등) 감지하여 새로 고칩니다. `connection-test-query` (Spring Boot에서는 `spring.datasource.hikari.connection-test-query`) 같은 설정이 있지만 HikariCP는 **Keepalive**와 **idleTimeout/maxLifetime** 메커니즘으로 별도 쿼리 없이도 문제 연결을 정리합니다. 굳이 설정할 필요는 없으나, 방화벽 등으로 인해 **오래 idle한 연결이 끊기는 환경**에서는 `maxLifetime`을 방화벽 타임아웃보다 짧게 설정하거나, `keepalive-time` (2.6.0+ 버전) 설정으로 주기적으로 테스트 쿼리를 날리게 할 수 있습니다.

- **쿼리 타임아웃**: 누수는 아니지만 **긴 시간 점유로 인한 풀 고갈**도 유사한 문제를 야기합니다. 예를 들어 한 개 요청이 트랜잭션을 시작하고 DB 커넥션을 오래 들고 있으면(pool에서 안 돌려주고), 풀에 유휴 커넥션이 없어 다른 요청들이 대기합니다. 이 상황이 심해지면 "HikariPool-1 - Connection is not available, request timed out after 30000ms" 같은 에러가 발생합니다. 이를 예방하려면 **쿼리 자체의 타임아웃**(예:

`spring.jpa.properties.hibernate.jdbc.timeout`이나 `JdbcTemplate.setQueryTimeout` 등)을 설정해 **지나치게 긴 쿼리를 중단**시키고, 커넥션이 영원히 점유되지 않도록 하는 것도 중요합니다.

- **Connection 재검증**: HikariCP는 풀에 반환된 커넥션을 다시 사용할 때 **자동으로 유효성**을 체크합니다 (디폴트로 lastAccess와 idleTimeout 기준). 필요한 경우 `validationTimeout` 등을 조정할 수 있습니다. 일반적으로 PostgreSQL JDBC 드라이버는 커넥션이 끊어졌으면 Exception을 던지므로 HikariCP가 이를 감지해 버립니다.

정리: Spring/Hibernate 환경에서는 코드를 적절히 작성하면 커넥션 누수는 드물지만, 혹시 모를 누수를 **Leak Detection으로 조기에 잡고**, **ConnectionTimeout** 등을 설정해 풀 고갈 상황에서 애플리케이션이 영구 대기하지 않도록 보호하는 것이 좋습니다. 또한 DB 세션에 남아있는 **트랜잭션 락**이나 **임시 테이블** 등의 부작용을 막기 위해, 미리 정해진 패턴대로 connection을 닫는 습관이 중요합니다.

커넥션 풀 사이즈 튜닝 전략

풀 크기(maximumPoolSize) 튜닝은 성능과 자원의 **Trade-off**를 다루는 작업입니다. 고려할 요소:

- **애플리케이션 동시성**: 동시에 DB를 사용하는 스레드 수를 파악합니다. 예를 들어 최대 동시 웹 요청 수가 100인데, 그 중 30%만 DB를 hit 한다면 이론상 30개 커넥션이면 충분합니다. 하지만 모든 요청이 DB를 쓰는 경우라면 동시 요청 수만큼 필요할 수도 있습니다. 비동기 혹은 다중스레드 작업도 있다면 그 수도 고려해야 합니다. **쓰레드풀 크기**와 연관지어 생각하면 좋습니다 (예: Tomcat maxThreads, Async executors 등). **Rule of Thumb**으로, 커넥션 풀 크기를 애플리케이션 레이어의 동시 처리 수와 비슷하거나 약간 작게 잡습니다.

- **DB 서버 자원**: PostgreSQL 서버의 CPU 코어, I/O 능력을 감안해야 합니다. 많은 커넥션이 동시에 쿼리를 던지면 CPU 컨텍스트 스위칭과 디스크 경합으로 오히려 느려집니다 [98]. 일반적으로 CPU 코어 수 x 2 정도까지는 병렬 쿼리가 효율적이고, 그 이상 커넥션이 동시에 active하면 성능 향상이 둔화되거나 악화됩니다 [99]. 따라서 (core*2) 정도를 1차 기준으로 삼아 봅니다. 만약 애플리케이션이 I/O bound (쿼리가 느리다든지)라서 커넥션 상당수가 기다리는 상태라면 이보다 풀을 늘려도 CPU에 여력이 있을 수 있습니다. 하지만 쿼리가 매우 빠르고 CPU bound이면 풀을 늘린다고 더 처리 못합니다 (한계는 CPU이므로).

- **PostgreSQL 내부 제한**: `max_connections` 설정보다 풀 총합이 크면 안 됩니다. 또한 PostgreSQL이 효율적으로 관리할 수 있는 연결 수는 수백 개 선입니다. 만약 애플리케이션 인스턴스가 많아서 커넥션 합계가 수백~천 단위로 갈 것 같다면, PG에서는 **Connection Pooler(PgBouncer)** 사용을 검토할 필요가 있습니다 [103]. PgBouncer는 애플리케이션 수많은 연결을 실제 소수의 DB backend로 multiplexing하여 Postgres에 부담을 줄여줍니다 [104].

- **실시간 모니터링**: 최적의 풀 사이즈는 **모니터링을 통해 결정**해야 합니다. Spring Boot Actuator의 metrics를 이용하면 **HikariCP 풀 상태**(active/idle/pending/total)를 알 수 있습니다 [105] [106]. `/actuator/metrics/hikaricp.connections` 엔드포인트를 보면 현재 활성 연결 수, 유휴 수, 대기 중 쓰레드 수 등을 알 수 있습니다 [105] [106]. 이상적으로는 active가 peak 시에도 maxPoolSize보다 조금 작고, pending (대기 쓰레드)가 거의 0이어야 합니다. 만약 active가 max까지 자주 차고 pending이 발생한다면 풀을 늘려야 할 가능성이 있고, 반대로 active가 항상 훨씬 낮고 idle이 max와 가까우면 풀 크기가 과도한 것이므로 줄일 수 있습니다.

또한 DB측 모니터링 (`pg_stat_activity` 뷰나 클라우드 DB 콘솔)을 통해 얼마나 많은 세션이 동시에 활성화되는지 본 뒤, 풀 사이즈를 조절할 수 있습니다. **DigitalOcean의 조언**처럼 처음엔 절반 정도 여유를 두고 설정한 후, 모니터링해보면서 CPU 사용률과 쿼리응답 시간을 살피며 조정하면 됩니다 [100]. CPU가 항상 꽉 차있다면 풀을 줄여 오버로드

를 막고 [100], CPU 여유가 많은데 대기(pending)가 많다면 풀을 늘려봅니다. 이러한 **계측 기반 조정**이 가장 합리적입니다.

- **특수 케이스**: 읽기 전용 많고 복제슬레이브가 있다면, 슬레이브에 별도 풀(멀티 DataSource)로 분산해 풀 크기를 나눠 설정할 수도 있습니다. 또는 대용량 배치작업은 전용 DB 세션을 써서 풀 영향을 안 주게 할 수도 있습니다. 이런 설계로 커넥션 사용을 분리하면 튜닝도 용이합니다.

요약하면, **풀 사이즈 튜닝은 "적당히 작으면서도 충분한" 균형점을 찾는 과정**입니다. 지나치게 크면 DB 부하와 컨텍스트 스위칭 낭비, 너무 작으면 요청 대기가 늘어납니다. 일반적으로 웹 애플리케이션은 10~50 사이에서 결정되고, 고부하 시스템도 100 이상으로 하는 일은 드뭅니다. 관계형 DB는 **동시 처리 쿼리 수를 제한**하는 것이 성능에 이득인 경우도 많습니다 [98]. 따라서 "모든 요청이 즉시 DB커넥션을 잡도록 풀을 크게" 하기보다는, 서비스 레벨에 맞는 **적절한 대기와 제한을 두는 것**이 전체 성능엔 나을 수 있습니다. Spring Boot/HikariCP 조합에서는 기본 10으로 시작해서, 단계적으로 모니터링하며 올리는 접근이 좋습니다.

커넥션 풀 모니터링 및 관리 도구

모니터링은 **풀링 전략이 잘 작동하는지** 파악하고 문제 발생 시 원인분석에 필수입니다:

- **Spring Boot Actuator Metrics**: 위에서 언급한 대로, Actuator를 사용하면 애플리케이션 레벨에서 HikariCP의 실시간 상태를 확인할 수 있습니다 [105]. `management.metrics.enable.jdbc=true` 설정으로 HikariCP 메트릭 수집을 켜고, `/actuator/metrics/hikaricp.connections` 엔드포인트를 조회하면 **현재 풀의 active/idle/total** 값이 나옵니다 [105] [106]. 또한 `hikaricp.connections.acquire`, `usage`, `creation` 등 세부 metric도 제공되어, 커넥션 획득 시간, 사용 시간 등의 히스토그램을 볼 수 있습니다. 이러한 지표들은 Prometheus 등으로 수집하여 대시보드로 시각화하면 유용합니다. 예를 들어 active 커넥션 수가 max에 근접한 시간을 알거나, 평균 사용 시간이 갑자기 늘어나면 문제가 있음을 감지할 수 있습니다 [107].

- **JMX 모니터링**: HikariCP는 MBean을 통해 Pool 정보를 노출할 수 있습니다. `spring.datasource.hikari.register-mbeans=true` 설정을 켜면 MBean Server에 Hikari 관련 속성이 등록됩니다 [108]. JConsole이나 VisualVM으로 MBean을 모니터링하거나, JavaMelody 등 모니터링 툴에서 JMX 수치를 볼 수 있습니다. Actuator를 쓰지 않는 환경에서 유용합니다.

- **로그 및 LeakDetection 로그**: HikariCP는 `DEBUG` 레벨에서 풀 상태 변화를 로깅합니다. 필요한 경우 `com.zaxxer.hikari` 패키지 로그를 낮춰두면 커넥션 획득/반납, 누수 감지 등의 로그를 볼 수 있습니다. 누수 감지(logLevel WARN) 시 로그를 면밀히 조사하여 누수 지점을 찾아야 합니다. 또한 Spring의 `DataSourceHealthIndicator`를 이용해 `/actuator/health`에서 DB 연결 확인도 가능하므로, health check에 실패하는 경우 (예: 풀을 다 써버려 getConnection 타임아웃) 알람으로 대응할 수 있습니다.

- **데이터베이스 측 모니터링**: PostgreSQL의 `pg_stat_activity` 뷰를 조회하면 현재 열린 세션 목록, 실행 중인 쿼리, 대기 상태 등을 파악할 수 있습니다. 이를 통해 예상치 못한 장시간 쿼리나 idle in transaction 상태 세션 등을 발견해 문제를 해결합니다. 예를 들어 connection leak으로 닫히지 않은 세션은 `state`가 'idle'로 계속 남아있고, `query`가 `<IDLE>`로 표시될 것입니다. 그리고 `backend_start` 시각을 보면 오래된 idle이 식별됩니다. 이러한 세션은 수동 종료(`pg_terminate_backend`)하고, 애플리케이션 로그의 leak detection과 대조하여 수정합니다.

- **기타 툴**: APM (예: New Relic, AppDynamics) 도구들도 JDBC 커넥션 풀 및 SQL 쿼리 모니터링을 제공합니다. New Relic 예시로, HikariCP JMX 등록 후 New Relic JMXFetch로 Hikari pool metrics를 수집해볼 수 있습니다 [108]. 또 PostgreSQL 자체 모니터링 툴인 pgHero, pganalyze 등도 세션 및 쿼리 정보를 분석해주므로, 풀 설정 문제 (예: too many connections) 감지에 도움이 됩니다.

- **경고 임계치 설정**: 운영 시 특정 기준을 넘으면 알람을 주도록 설정합니다. 예: 활성 커넥션 수가 90% 이상 일정 시간 지속, 대기 쓰레드가 발생, DB 서버의 connection 수 nearing max, etc. 이러한 알람으로 사고를 미리 방지할 수 있습니다.

마지막으로, **Connection Pool** 관련 흔한 문제는 **Connection leak**과 **Connection storm**(짧은 시간에 연결 대량 생성)입니다. 앞서 논의한 기법들(LeakDetection, 적절한 pool size, idleTimeout 설정 등)을 활용하면 이런 문제를 사전에 방지할 수 있습니다. 특히 PostgreSQL은 커넥션 생성/소멸에 비용이 크므로(프로세스 생성), 풀링이 필수적이고, 풀 튜닝이 성능에 직접 영향을 줍니다. Spring Boot + HikariCP 조합은 비교적 손쉬운 설정으로 안정적인 풀링을 제공하니, **문제 발생 시 EXPLAIN을 보듯 Actuator metrics와 DB 통계를 보며** 원인을 찾아 튜닝해나가는 것이 최선의 전략입니다 [109].

참고 자료: HikariCP 공식 문서 및 Spring Boot 문서의 데이터소스 설정 가이드 [88] [110], PostgreSQL Connection Pooling 관련 권장사항 [100], Spring Boot Actuator를 통한 HikariCP 모니터링 예제 [105] [106] 등. 이를 종합하여 최적의 PostgreSQL 연결 풀 구성을 도출할 수 있습니다.

---

[1] [2] [3] [4] [7] [8] [9] [10] [18] [19] [20] [29] [32] [42] Postgres vs. MySQL: a Complete Comparison in 2025
https://www.bytebase.com/blog/postgres-vs-mysql/

[5] [15] [16] Ask HN: It's 2023, how do you choose between MySQL and Postgres? | Hacker News
https://news.ycombinator.com/item?id=35906604

[6] 사이드 프로젝트를 MySQL에서 PostgreSQL 로 마이그레이션 하게 된 이유
https://velog.io/@peace_e/%EC%82%AC%EC%9D%B4%EB%93%9C-%ED%94%84%EB%A1%9C%EC%A0%9D%ED%8A%B8%EB%A5%BC-MySQL%EC%97%90%EC%84%9C-PostgreSQL-%EB%A1%9C-%EB%A7%88%EC%9D%B4%EA%B7%B8%EB%A0%88%EC%9D%B4%EC%85%98-%ED%95%98%EA%B2%8C-%EB%90%9C-%EC%9D%B4%EC%9C%A0

[11] [12] [13] [14] MySQL and PostgreSQL Licence issue - Database Administrators Stack Exchange
https://dba.stackexchange.com/questions/3914/mysql-and-postgresql-licence-issue

[17] [30] [31] Migrating from MySQL to PostgreSQL - What You Should Know | Severalnines
https://severalnines.com/blog/migrating-mysql-postgresql-what-you-should-know/

[21] [22] [27] [28] MySQL :: MySQL 8.4 Reference Manual :: 15.2.7.2 INSERT ... ON DUPLICATE KEY UPDATE Statement
https://dev.mysql.com/doc/refman/8.4/en/insert-on-duplicate.html

[23] [24] [25] [26] PostgreSQL: Documentation: 17: INSERT
https://www.postgresql.org/docs/current/sql-insert.html

[33] [34] [38] [39] [40] [41] MySQL :: MySQL 8.4 Reference Manual :: 15.2.17 UPDATE Statement
https://dev.mysql.com/doc/refman/8.4/en/update.html

[35] [36] [37] PostgreSQL: Documentation: 17: UPDATE
https://www.postgresql.org/docs/current/sql-update.html

[43] [85] Today I Learned: Understanding PostgreSQL Explain Query | by Iman Tumorang | Easyread
https://medium.easyread.co/today-i-learned-understanding-postgres-explain-query-5670dd042c99?gi=86c957eacd3b

[44] [45] [46] [47] [50] The EXPLAIN query plan - AWS Prescriptive Guidance
https://docs.aws.amazon.com/prescriptive-guidance/latest/postgresql-query-tuning/explain-query-plan.html

48 49 51 52 57 58 59 60 62 63 64 65 66 67 68 69 70 71 72 73 75 80 81 82 83 84 PostgreSQL: Documentation: 17: 14.1. Using EXPLAIN
https://www.postgresql.org/docs/current/using-explain.html

53 54 55 56 61 74 76 77 78 79 PostgreSQL execution plan operations
https://use-the-index-luke.com/sql/explain-plan/postgresql/operations

86 87 90 91 92 93 95 96 97 99 101 102 105 106 107 109 Optimizing Database Connections with HikariCP in Spring Boot 3+ | Medium
https://medium.com/@ahmettemelkundupoglu/optimizing-database-connections-with-hikaricp-in-spring-boot-3-and-java-21-80fab58cc1c7

88 110 81. Data Access - Spring
https://docs.spring.io/spring-boot/docs/2.0.6.RELEASE/reference/html/howto-data-access.html

89 Common Application Properties :: Spring Boot
https://docs.spring.io/spring-boot/appendix/application-properties/index.html

94 Connection leak detected by HikariPool LeakDetectionThreshold
https://stackoverflow.com/questions/60424898/connection-leak-detected-by-hikaripool-leakdetectionthreshold

98 What is the relation between Connection pool size and number of ...
https://www.reddit.com/r/PostgreSQL/comments/17ve883/what_is_the_relation_between_connection_pool_size/

100 103 How to Manage Connection Pools for PostgreSQL Database Clusters | DigitalOcean Documentation
https://docs.digitalocean.com/products/databases/postgresql/how-to/manage-connection-pools/

104 Why you should use Connection Pooling when setting ... - EDB
https://www.enterprisedb.com/postgres-tutorials/why-you-should-use-connection-pooling-when-setting-maxconnections-postgres?lang=en

108 Springboot 3 default Connection Pool Hikari not showing JVM Data ...
https://forum.newrelic.com/s/hubtopic/aAXPh00000013dNOAQ/springboot-3-default-connection-pool-hikari-not-showing-jvm-data-sources-tab