ChatGPT

PostgreSQL 인덱스 유형과 고급 최적화 전략

인덱스의 역할과 비용

인덱스는 데이터베이스의 **조회 성능을 향상**시키는 핵심 도구입니다. 예를 들어 어떤 테이블에서 `WHERE id = 42`와 같은 조건으로 행을 찾을 때, **인덱스가 없다면** 데이터베이스는 전체 테이블을 훑어야 하지만, **인덱스가 있다면** 몇 단계만에 원하는 행을 찾을 수 있습니다 [1] [2]. 이러한 이점 때문에 **PostgreSQL**에서는 다양한 종류의 인덱스를 지원하며, 적절한 인덱스를 사용하면 `SELECT`, `UPDATE`, `DELETE` 등의 **쿼리를 대폭 빠르게** 만들 수 있습니다 [3].

그러나 **인덱스는 공짜가 아닙니다.** 인덱스를 유지관리하기 위해서는 부가적인 **공간과 연산 비용**이 듭니다 [4]. 테이블에 인덱스를 만들면, 이후 **행이 추가되거나 수정될 때마다** 해당 인덱스도 갱신되어야 합니다 [5]. 그 결과 쓰기 작업(`INSERT`, `UPDATE`, `DELETE`)에는 추가 **오버헤드**가 발생합니다. 또한 인덱스가 존재하면 PostgreSQL은 **HOT 업데이트(Heap-Only Tuple)** 최적화를 일부 사용하지 못하게 되어 테이블 업데이트 성능에 영향이 있을 수 있습니다 [6]. **사용 빈도가 낮거나 거의 사용되지 않는 인덱스는 오히려 성능 저하와 공간 낭비만 초래**하므로 이런 인덱스는 제거하는 것이 좋습니다 [6]. 따라서 **인덱스 설계 시에는 조회 이득과 쓰기 비용의 균형**을 고려해야 합니다.

인덱스 **생성 작업** 자체도 큰 테이블의 경우 **시간이 오래 걸릴 수 있고** 시스템에 부하를 줍니다 [7]. 기본적으로 인덱스를 만드는 동안 **읽기 쿼리**는 동시에 수행될 수 있지만, **쓰기 작업은 인덱스 생성이 끝날 때까지 블로킹**됩니다 [7]. (이를 완화하기 위해 `CREATE INDEX CONCURRENTLY` 옵션을 사용하면 쓰기 차단을 피할 수 있으나, 그만큼 시간이 더 걸리고 몇 가지 제약이 있습니다.) 인덱스를 만든 후에는 **주기적으로** `ANALYZE`를 실행하여 **통계 정보를 갱신**해야 쿼리 플래너가 인덱스를 활용할지 말지를 정확히 판단할 수 있습니다 [5]. 실제 쿼리에서 인덱스가 사용되는지 확인하려면 `EXPLAIN`으로 실행 계획을 검사하거나, PostgreSQL이 제공하는 **인덱스 통계 뷰**(`pg_stat_user_indexes` 등)를 확인할 수 있습니다 [8].

아래에서는 **PostgreSQL이 지원하는 모든 인덱스 유형**의 구조와 작동 방식, 장단점과 사용 시나리오를 공식 문서를 기반으로 상세히 설명합니다. 또한 **다중 컬럼 인덱스**, **함수/표현식 기반 인덱스**, **파셜(부분) 인덱스**, **커버링 인덱스**(`INCLUDE` **사용)** 등의 고급 활용법과 **인덱스 튜닝 및 통계 활용 전략**에 대해서도 다룹니다.

PostgreSQL의 주요 인덱스 유형

PostgreSQL은 **여러 가지 인덱스 방식**을 제공합니다 [9]. 인덱스 생성 시 `USING 인덱스유형` 구문으로 원하는 방식을 지정할 수 있으며 (지정하지 않으면 **기본값으로 B-Tree** 인덱스가 생성됩니다 [9]), 각 인덱스 유형마다 **내부 구조와 알고리즘**이 다르므로 적합한 용도가 다릅니다. 아래에는 **주요 인덱스 유형**인 **B-Tree**, **Hash**, **GiST**, **SP-GiST**, **GIN**, **BRIN**의 원리와 특성을 설명하고, 공식 문서 링크와 예제 쿼리를 함께 제시합니다. (이 밖에 **Bloom 필터 인덱스**도 확장으로 제공되지만, 여기서는 PostgreSQL 코어에 포함된 인덱스에 집중합니다 [9].)

B-Tree 인덱스

B-트리(B-Tree) 인덱스는 PostgreSQL에서 **가장 일반적이고 기본**으로 사용되는 인덱스 유형입니다. B-트리는 **균형 이진 트리(Balanced Tree)**의 한 형태로, 인덱스 키들을 **정렬된 순서로 저장**하여 **로그arithmic 검색 성능**을 제공합니다. 내부적으로 **트리의 각 노드**는 여러 키 값을 담은 **페이지(page)**이며, 자식 포인터를 통해 트리 구조를 이룹니다. PostgreSQL의 B-트리는 **동시성 제어**와 **페이지 분할** 알고리즘(예: Lehman-Yao 알고리즘)을 사용하여 다중 사용자 환경에서도 효율적으로 **균형을 유지**합니다. 그 결과, **키 값을 기준으로 한 동등 비교와 범위 검색에 매우 뛰어난 성능**을 보여주며, 대부분의 일반적인 질의에 적합합니다 [10].

B-트리 인덱스는 **정렬 가능**한 데이터에 대해 동등(`=`) 조건뿐 아니라 **크다/작다(<, <=, >, >=)**와 같은 **범위 조건**도 빠르게 처리합니다 [10]. 예를 들어 숫자나 날짜, 텍스트 등의 순서가 정의된 타입에 대한 `WHERE 컬럼 = 값`, `WHERE 컬럼 BETWEEN A AND B` 등의 조건에서 B-트리 인덱스를 사용할 수 있습니다. 또한 `BETWEEN`, `IN (...)`과 같은 조건은 내부적으로 해당 연산자로 변환되어 B-트리 검색에 사용될 수 있습니다 [10]. B-트리 인덱스는 `IS NULL`/`IS NOT NULL` 조건도 지원하며, 일부 패턴 매칭 연산자에도 활용 가능합니다 [11]. 예를 들어 `col LIKE 'foo%'` (접두어가 고정된 패턴)이나 `col ~ '^foo'` (정규식 접두어 일치) 같은 경우 B-트리로 가속할 수 있습니다 [12]. 다만 데이터베이스 로케일이 C가 아닌 경우 이러한 패턴 매칭을 위해 **특수 operator class**로 인덱스를 생성해야 합니다 [13]. B-트리 인덱스는 결과를 **정렬된 순서로 반환**할 수 있기 때문에, `ORDER BY` 절이 있는 쿼리에서 **추가 정렬을 생략**하도록 도와주기도 합니다 [14]. (물론 소량의 결과나 특정 상황이 아니라면, 인덱스 순회 + 정렬 vs. 전체 검색 + 정렬의 비용을 비교하여 **플래너가 더 빠른 쪽**을 선택합니다.)

장점: B-트리는 대부분의 상황에서 **가장 범용적이고 효율적**입니다. **동등 질의와 범위 질의**를 모두 가속할 수 있으며, 데이터가 **균등 분포**되어 있든 **편중 분포**되어 있든 안정적인 성능을 제공합니다. 또한 **인덱스만으로 정렬된 결과**를 얻을 수 있어, 정렬 비용을 줄이거나 **MIN/MAX 연산을 빠르게** 처리하는 데도 쓸 수 있습니다. **다중 컬럼 인덱스**, **부분 인덱스**, **표현식 인덱스** 등 PostgreSQL의 다양한 인덱스 기능 대부분을 **B-트리 방식**으로 구현할 수 있으며, **고유 인덱스(UNIQUE)**나 **외래키 지원**도 B-트리로 이루어집니다. 쓰기 작업(삽입/갱신) 시에도 과도한 부하가 걸리지 않아 **전천후 인덱스**로 볼 수 있습니다.

단점: 데이터 타입에 **정렬 기준(순서)**이 정의되어 있어야 하며, **비정렬 데이터(예: JSON 전체, 문서형 텍스트 등)**에는 직접 적용할 수 없습니다. 또한 **아주 드물지만** 키 값이 계속 증가하거나 감소하여 **한쪽으로 몰릴 경우**(예: 자동 증가 시퀀스만으로 구성된 PK 인덱스에 지속 삽입) **페이지 분할과 조각화**가 발생할 수 있습니다. 이때 **인덱스 재구성(REINDEX)**이나 **Fillfactor 조정** 등의 튜닝이 필요할 수 있습니다. 하지만 일반적으로 B-트리는 관리 비용 대비 얻는 이점이 커서 특별한 이유가 없다면 **가장 먼저 고려해야 할 인덱스**입니다.

예제: 기본적으로 `CREATE INDEX` 명령은 B-트리 인덱스를 생성합니다. 예를 들어 테이블 `orders(amount numeric, created_at date)`에서 **금액**이나 **날짜 범위**로 자주 검색한다면, 해당 컬럼에 B-트리 인덱스를 만들 수 있습니다:

```
-- 금액 컬럼에 B-트리 인덱스 생성 (기본 인덱스)
CREATE INDEX idx_orders_amount ON orders(amount);

-- 날짜 컬럼에 B-트리 인덱스 생성
CREATE INDEX idx_orders_date ON orders(created_at);

-- 인덱스 활용 질의 예시
SELECT *
FROM orders
WHERE created_at BETWEEN DATE '2023-01-01' AND DATE '2023-01-31'
ORDER BY created_at;
```

위 질의에서 `orders.created_at` 컬럼의 범위를 조건으로 조회하면, **B-트리 인덱스**를 통해 해당 기간의 행만 빠르게 찾아낼 수 있습니다 [10]. 또한 결과를 `created_at`으로 정렬하도록 요구하지만 인덱스 자체가 날짜 순으로 정렬되어 있으므로 **추가 정렬 없이** 빠르게 결과를 반환합니다 [14].

Hash 인덱스

해시(Hash) 인덱스는 **해시 테이블** 자료구조를 기반으로 한 인덱스입니다. PostgreSQL의 Hash 인덱스는 각 인덱스 키 값에 대해 **32비트의 해시 코드**를 계산하여 저장합니다 [15]. **동등 비교(Equality)**에 최적화된 구조이기 때문에 `=`

연산에 대해서만 인덱스 검색을 지원하며, **범위 질의(<, > 등)**에는 사용할 수 없습니다 [15]. 질의 플래너도 해시 인덱스는 **같은 값 비교(`=`)**일 때만 활용합니다 [15].

Hash 인덱스는 내부적으로 **버킷(bucket)**이라 불리는 페이지 체계를 사용합니다. 해시 함수를 통해 키가 **어느 버킷에 속할지 결정**되며, 해당 버킷 페이지에 인덱스 엔트리가 저장됩니다. **버킷 하나에 너무 많은 엔트리**가 몰리면 **오버플로(overflow) 페이지**를 연결 리스트로 이어붙여 저장하는 **확장형 해시 구조**를 사용합니다 [16] [17]. 이러한 설계로 해시 인덱스는 **해시 값이 균등 분포**된 경우 **O(1)**에 가까운 매우 빠른 검색 성능을 보입니다. 특히 **테이블 행이 수백만 건 이상으로 매우 큰 경우**, B-트리는 트리 높이가 높아져 탐색 단계가 늘어나지만 해시 인덱스는 **해시 계산과 단일 버킷 접근만으로 검색**이 가능해 **큰 테이블의 동등 질의에 효과적**일 수 있습니다 [18]. 또한 저장 시 **실제 키 대신 해시 값(4바이트)만 저장**하므로, **키 크기가 큰 데이터(UUID, 긴 문자열 등)**를 인덱싱할 때 **인덱스 크기를 줄이는** 장점이 있습니다 [19]. (키 원본값을 저장하지 않기 때문에, 해시 인덱스는 **동일 해시 충돌**에 대비해 결과를 가져온 뒤 **테이블에서 실제 값을 재검증**하는 과정을 거칩니다. 이러한 특성상 해시 인덱스 스캔을 **"후처리 필요"**로 간주하여 **lossy 스캔**이라고도 합니다 [20].)

장점: 동등 비교만 매우 빈번하게 일어나는 경우에 특화되어 있습니다. 예를 들어 매우 큰 테이블에서 **키 값 하나로만 조회**하는 쿼리(`WHERE key = X`)가 압도적으로 많다면, 해시 인덱스가 B-트리보다 **빠른 검색**을 제공할 수 있습니다 [18]. 특히 **정렬 불가능한 데이터 타입**(예: JSON 전체, 배열 등)에 대해 **동등 비교만** 필요하다면, B-트리가 지원되지 않으므로 해시 인덱스를 고려할 수 있습니다 [21]. (PostgreSQL의 해시 인덱스는 **임의 데이터 타입**에 대해 사용할 수 있도록 설계되어 있습니다 [21].) 또한 키 값이 큰 경우 인덱스 크기 절감 효과도 이점입니다 [19].

단점: 동등 비교 이외의 용도에 전혀 사용할 수 없으며, 다중 컬럼 인덱스를 지원하지 않기 때문에 범위 조회나 복합 조건이 있다면 해시 인덱스로는 가속이 불가능합니다 [22]. **고유 인덱스**나 **다중 열**도 지원되지 않아 제약 조건 용도로 사용할 수도 없습니다 [22]. 게다가 **데이터 분포가 고르지 못한 경우** (특정 몇몇 값이 대량 존재하는 경우) **해시 충돌이 증가**하여 한 버킷에 너무 많은 엔트리가 몰리면, 해당 버킷의 오버플로 페이지들을 **순차로 모두 검사**해야 하므로 **성능이 급격히 떨어질 수 있습니다** [23]. 이런 경우 동일한 상황에서 B-트리는 같은 값을 인접한 범위로 저장하되 **여러 트리 수준에 걸쳐 분산**되므로, 해시 인덱스보다 유리할 수 있습니다 [24]. 따라서 해시 인덱스는 **키 값이 대부분 고유하거나 충돌이 적은 경우**에 적합합니다 [17]. (만약 **중복도가 높은 값**을 일부 제외하고 싶다면 **부분 인덱스**와 결합해 특정 값들을 인덱싱하지 않는 방법도 고려할 수 있지만 [17], 일반적으로는 그런 값은 처음부터 해시 인덱스로 이득을 보기 어렵습니다.) 또한 해시 인덱스는 B-트리에 비해 구현 역사가 짧아서인지 **활용 빈도가 낮고**, 충분한 이득을 보지 못하는 경우도 많습니다. **PostgreSQL 10 버전 이전**까지는 해시 인덱스가 **WAL 로그에 기록되지 않아** 장애 발생 시 일관성 문제가 있어 권장되지 않았으나, 현재 버전은 **완전하게 WAL 및 크래시 리커버리를 지원**하므로 안정성 측면 문제는 개선되었습니다 [21]. 그럼에도 불구하고, 대다수 동등 질의는 B-트리로도 충분히 빠르며 다른 용도까지 포괄할 수 있으므로, 해시 인덱스는 **특정 상황에서만 한정적으로 사용하는 인덱스**입니다.

예제: 해시 인덱스는 명시적으로 `USING HASH`를 지정하여 생성합니다. 예를 들어, 매우 큰 테이블 `users(username text, ...)`에서 사용자 이름으로만 자주 조회하며 (`WHERE username = '...'`) 다른 조건은 거의 사용되지 않는다고 가정합니다. 이 경우 `username` 컬럼에 해시 인덱스를 만들어 볼 수 있습니다:

```
-- username 컬럼에 대한 해시 인덱스 생성
CREATE INDEX idx_users_username_hash ON users USING HASH (username);

-- 인덱스 활용 질의 예시 (동등 비교)
SELECT email
FROM users
WHERE username = 'johndoe';
```

위와 같이 생성된 해시 인덱스는 `username = 'johndoe'`와 같은 **동등 조건**에서만 사용되며, PostgreSQL 플래너는 해당 조건이 있을 때에만 이 인덱스를 고려합니다 [15]. 만약 `username LIKE 'jo%'`나 범위 조건 등이 들어오면 해시 인덱스는 사용되지 못하고, 그 때는 B-트리 인덱스가 필요합니다.

GiST 인덱스

GiST는 “일반화 검색 트리(**G**eneralized **S**earch **T**ree)”의 약자로, **단일한 인덱스 기법이라기보다는 인덱스 인프라 구조**입니다 [25]. GiST 인덱스는 다양한 데이터 타입의 특성에 맞는 **사용자 정의 인덱싱 전략**을 구현할 수 있는 **플러그인 형 인덱스**로 이해할 수 있습니다 [25]. 내부적으로는 **B-트리와 유사한 트리 구조**를 취하지만, 각 **노드가 갖는 키값의 의미와 비교 연산**을 개발자가 정의하는 방식입니다. 예를 들어 **이차원 공간**의 도형 데이터를 위한 GiST 인덱스에서는 **노드가 사각형 범위(바운딩 박스)**를 키로 갖고, **왼쪽/오른쪽 자식**을 해당 범위로 분할하는 **R-트리**와 비슷한 동작을 합니다. 다른 예로 **문서 검색용 풀텍스트** 데이터에 GiST를 적용하면 각 노드가 **용어 집합**을 요약 정보로 담고, 단어 비교를 위한 트리를 구성할 수 있습니다. 즉, **GiST 자체는 어떤 자료구조의 구현이라기보다** “이런 방식으로 트리를 탐색/분할할 수 있다”는 **추상화된 프레임워크**이며, **구체적인 로직은 오퍼레이터 클래스(operator class)**로 제공됩니다 [25].

PostgreSQL에 기본 내장된 GiST **연산자 클래스**들을 살펴보면, **2차원 기하 타입** (point, box 등)이나 **범위(range) 타입**, 그리고 일부 **텍스트 검색** 등이 포함되어 있습니다 [26]. 예를 들어 **기본 기하학 연산자 클래스**는 **점, 사각형, 원 등**에 대해 **“왼쪽에 있다(<<)”, “오른쪽에 있다(>>)”, “포함한다(@>)”, “포함된다(<@)”, “겹친다(&&)”** 등의 연산자를 GiST 인덱스로 가속합니다 [26]. 이러한 연산은 B-트리로는 처리 불가능하므로 GiST 인덱스를 사용하면 **공간 질의**나 **범위 겹침 질의** 등을 효율화할 수 있습니다 [26]. 또한 GiST는 **최근접 이웃(nearest-neighbor) 검색**을 지원하는 인덱스이기도 합니다 [27]. 특정 GiST 연산자 클래스는 **<->** 같이 두 값 간 거리를 계산하는 연산자를 정의하고, **ORDER BY 절에서 이 연산자를 사용**하면 인덱스가 **가장 가까운 순서대로** 결과를 찾을 수 있습니다 [27]. 예를 들어 어떤 위치 좌표 테이블에서 **특정 지점과 가장 가까운 상위 N개 좌표를 찾는** 쿼리를 GiST 인덱스로 매우 빠르게 처리할 수 있습니다 [27]. 이러한 KNN 검색 능력은 **GiST 연산자 클래스에 따라** 제공되며, PostgreSQL 내장 GiST 클래스 중에는 2차원 공간 데이터 등이 이를 지원합니다 (공식 문서의 **주문 연산자(Ordering Operators)** 열에 표시) [27].

장점: GiST 인덱스는 **매우 유연하고 확장 가능**합니다. **사용자 정의 데이터 타입**이나 **특수한 질의 요구사항**이 있을 때, GiST 프레임워크를 사용하여 새로운 인덱스 방법을 정의할 수 있습니다. 덕분에 PostgreSQL은 **공간 데이터(PostGIS 확장)**, **IP 주소 범위 검색**, **텍스트 유사도 검색(pg_trgm 확장)** 등 다양한 분야에서 GiST 인덱스를 활용합니다. **범위 비교**, **위치 겹침 여부**, **최근접 검색** 등 **고차원적인 질의 조건**을 효율적으로 처리할 수 있다는 것이 가장 큰 강점입니다. 또한 **다중 컬럼 GiST 인덱스**도 지원하여, 첫 번째 컬럼으로 대략적인 후보군을 줄이고 추가 컬럼으로 세부 필터링하는 식으로 사용할 수 있습니다 [28]. (예를 들어 **첫 번째 컬럼**은 위치, **두 번째 컬럼**은 카테고리인 GiST 복합 인덱스를 만들어 **“위치로 1차 필터 + 카테고리로 2차 필터”** 같은 작업을 인덱스 한 번에 수행 가능하게 할 수 있습니다.)

단점: GiST는 **범용성**을 위해 설계되었기에 **특정 작업에 B-트리만큼 특화된 최적화**는 없습니다. 예를 들어 **동등 조건만 필요한 단순 정수 키**라면 B-트리가 GiST보다 빠르고 효율적입니다. 또한 GiST 인덱스는 내부 노드에 **포함 관계나 근사 정보**(예: 바운딩 박스)를 저장하므로, **정확한 일치가 아닌 “후보”를 찾아낸 후 후처리**가 필요할 때가 많습니다. 이를테면 **기하학적 겹침(&&)** 연산을 GiST로 처리하면 후보 객체들을 찾아낸 뒤 실제로 겹치는지 **재검증(recheck)**하는 단계를 거칩니다. 따라서 **쿼리 조건의 선택도가 낮을 때** (즉, 많은 후보가 나올 때) **인덱스 이득이 줄어드는 경향**이 있습니다. 또 하나 유의할 점은, **GiST 다중 컬럼 인덱스**에서 **첫 컬럼의 분포가 너무 제한적이면** 효과가 떨어질 수 있다는 것입니다 [28]. 첫 컬럼 값이 몇 가지로만 나뉜다면 트리의 가지가 충분히 갈라지지 않아 결국 **대부분의 페이지를 탐색**해야 할 수 있습니다. 따라서 다중 컬럼 GiST를 설계할 때는 **첫 번째 컬럼에 충분한 판별력**이 있도록 하는 편이 좋습니다 [28]. 저장 공간 면에서는, GiST 인덱스는 **키 요약 정보를 저장**하므로 B-트리보다 다소 크거나 깊이가 깊어질 수 있습니다. 그리고 쓰기 작업 시 **균형 유지를 위한 작업**이 B-트리보다 복잡할 수 있으나, PostgreSQL이 알아서 처리합니다. 전반적으로 GiST는 **특수한 용도**를 위해 사용되며, 해당 용도가 아닌 경우에는 사용 이점을 얻기 어렵습니다.

예제: GiST 인덱스는 보통 **공간 데이터**나 **복합 자료형**에 쓰입니다. 예를 들어, `locations`라는 테이블에 사용자 위치 (`geom` 컬럼, Point 타입)이 저장되어 있고, 이 테이블에서 **특정 좌표와 반경 5km 내의 점 찾기** 또는 **가장 가까운 상위 10개 점 찾기** 같은 질의를 자주 실행한다고 해봅시다. 이를 위해 `geom` 컬럼에 GiST 인덱스를 생성할 수 있습니다:

```
-- PostGIS의 geometry(Point) 컬럼에 대한 GiST 인덱스 생성
CREATE INDEX idx_locations_geom ON locations USING GiST (geom);
```

```sql
-- 인덱스 활용 질의 예시 1: 특정 영역(사각형 Bounds) 내의 포인트 검색
SELECT *
FROM locations
WHERE geom && ST_MakeEnvelope(100, 200, 110, 210, 4326);

-- 인덱스 활용 질의 예시 2: 특정 좌표와 가장 가까운 10개 포인트 찾기
SELECT id, geom
FROM locations
ORDER BY geom <-> ST_Point(105, 205)
LIMIT 10;
```

첫 번째 질의에서 `&&` **연산자는 두 기하객체의 겹침(Overlap)** 여부를 나타내며, **GiST 인덱스**를 통해 해당 사각형 영역과 겹치는 점들을 빠르게 찾아냅니다 [26]. 두 번째 질의는 `<->` 연산자를 사용하여 특정 점과의 **거리 순으로 정렬**하고 상위 10개를 가져오는데, 이 경우 GiST 인덱스가 **최근접 이웃** 질의를 수행하여 가장 가까운 점 10개를 효율적으로 반환합니다 [27]. 이러한 질의들은 B-트리로는 처리가 어렵거나 매우 비효율적이지만, GiST 인덱스를 통해 실용적인 성능을 얻을 수 있습니다.

SP-GiST 인덱스

SP-GiST는 **공간 분할 GiST(Space-Partitioned GiST)**의 약자입니다 [29]. 이름에서 알 수 있듯, **GiST의 일반화된 트리 구조를 변형**하여 **공간 분할 방식**의 다양한 트리를 구현할 수 있게 한 인덱스입니다. **SP-GiST 인덱스는 균형을 반드시 유지하지 않고도** 데이터 공간을 재귀적으로 분할하는 **비균형 트리**들을 지원합니다 [29]. 예를 들면 **사분 트리(quadtree)**, **k-평면 트리(k-d tree)**, **트라이(trie)** 등의 구조를 SP-GiST 프레임워크를 통해 만들 수 있습니다 [29].

PostgreSQL에 내장된 SP-GiST 연산자 클래스의 예로, **2차원 Point 데이터**에 대한 것이 있습니다. 이는 2차원 평면을 사분할하는 **쿼드트리(quadtree)** 방식으로 좌표를 분할하여 저장하며, 해당 클래스는 **"왼쪽/오른쪽에 있음(<<, >>)", "포함됨(<@)"** 등의 몇 가지 공간 연산자를 지원합니다 [30]. SP-GiST는 **텍스트에 대한 접두사 트라이(prefix trie)** 구현에도 활용됩니다. 예를 들어, 긴 문자열들에 대해 **공통 접두사를 분기**로 하여 트리 구조를 만들면, `col LIKE 'abc%'` 같은 **접두사 검색**을 매우 빠르게 처리할 수 있습니다. (이런 용도의 SP-GiST 연산자 클래스로 text_ops 등이 제공됩니다.) 이처럼 SP-GiST는 **자료가 가지는 자연스러운 분할 기준**(공간적 위치, 문자열 접두사, IP주소 계층 등)을 이용해 **트리를 구성**하므로, **데이터가 한쪽으로 치우치거나 한정된 범위에 몰려 있어도** 그 특성에 맞게 분할하여 탐색 효율을 높일 수 있습니다.

장점: 균형 트리보다 공간적으로 효율적인 분할을 할 수 있는 데이터에 적합합니다. 예를 들어 **IP 주소**와 같이 계층적 구조(프리픽스)를 지닌 데이터, 또는 **문자열 접두사**로 검색하는 경우, SP-GiST는 이러한 구조를 활용해 **매우 빠른 검색**을 제공합니다. **트라이(trie)** 구조를 통해 문자열 **공통 부분은 한 번만 저장**하고 분기하기 때문에 **메모리 사용량도 효율적**일 수 있습니다. 또한 **빈도 분포가 고르지 않은 데이터** (어떤 값 범위에 데이터가 편중된 경우)도 해당 영역을 세부적으로 분할하여 처리하므로, B-트리나 균형 트리에 비해 **깊이는 깊어져도 검색 범위를 크게 줄이는** 이점을 가질 수 있습니다. GiST와 마찬가지로 **다양한 확장**이 가능하며, **내장 opclass 외에도** 개발자가 SP-GiST를 활용한 새로운 인덱스를 정의할 수 있습니다.

단점: SP-GiST는 **여러 컬럼을 한 인덱스에 함께 가지는 것을 지원하지 않습니다.** 하나의 트리가 하나의 키 공간을 분할하는 역할에 충실하기 때문에, **다중 열 SP-GiST 인덱스는 생성할 수 없습니다** [31]. 따라서 여러 컬럼을 동시에 인덱싱해야 한다면 다른 인덱스 방식으로 보완하거나 별도 인덱스를 만들어야 합니다. 또한 **데이터 분포를 설계자가 잘 알고 있어야** 최상의 효과를 얻습니다. 예를 들어 문자열 접두사 검색이 거의 없는데 SP-GiST 트라이 인덱스를 만들어봐야 얻는 이익이 없습니다. **특정 질의 패턴에 매우 특화**되는 경향이 있으므로, 범용 쿼리에는 맞지 않습니다. **트리의 균형을 강제하지 않으므로** 최악의 경우 트리가 한쪽으로 매우 깊어질 수 있지만, 이는 해당 분야 알고리즘으로 완화됩니다. (예컨대 트라이는 최악의 경우 문자열 길이만큼 깊어질 수 있으나, 실제 검색은 접두사 매칭 조건이 있을 때만 수행되므로 문

제가 되지 않습니다.) 요약하면, SP-GiST는 **특정 유형의 질의(접두사, 공간 분할 등)**에 대해 **최적화된 인덱스**를 제공하며, 그 외 경우에는 사용할 이유가 없습니다.

예제: SP-GiST의 대표적 사용 사례로 **문자열 접두사 검색**을 들 수 있습니다. `posts(title text, ...)` 테이블이 있고, 제목 컬럼으로 **접두사 일치 검색**(예: `WHERE title LIKE 'Hello%'`)을 자주 수행한다고 가정합니다. B-트리로는 `LIKE 'prefix%'`를 처리할 수 있지만, 텍스트 패턴 전용 operator class가 필요하고 성능이 제한적입니다. 이때 **SP-GiST 인덱스**를 사용하면 **트라이 기반**으로 효율적인 접두사 검색이 가능합니다:

```
-- 텍스트 컬럼에 대한 SP-GiST 인덱스 생성 (접두사 검색 최적화)
CREATE INDEX idx_posts_title_trie ON posts
USING SPGIST (title text_pattern_ops);

-- 인덱스 활용 질의 예시: 접두사로 제목 검색
SELECT *
FROM posts
WHERE title LIKE 'Hello%';
```

위 예시는 `text_pattern_ops`라는 SP-GiST 연산자 클래스를 사용하여 `title` 컬럼의 접두사별로 분기하는 **trie 인덱스**를 만든 것입니다. 그런 다음 `WHERE title LIKE 'Hello%'` 조건의 질의를 수행하면, 이 인덱스를 통해 `'Hello'`로 시작하는 제목들만 빠르게 탐색하여 찾습니다. 결과적으로 **대량의 문자열 중 일부 접두사를 가지는 항목들만 신속히 조회**할 수 있게 됩니다. (만약 이러한 패턴 검색이 빈번하지 않다면, 굳이 별도 인덱스를 만들기보다 B-트리 또는 풀텍스트 검색을 고려해야 합니다.)

또 다른 예로, `ip_addr` 컬럼에 `inet` 타입으로 IP주소가 저장된 경우, **CIDR 같은 네트워크 대역 검색** (`WHERE ip_addr <@ '192.168.0.0/16'` 등)에 SP-GiST 인덱스를 활용할 수 있습니다. SP-GiST의 inet 연산자 클래스는 IP주소를 접두사 비트스트링으로 보아 트리를 구성하므로, **IP 블록 질의**에 효과적입니다.

GIN 인덱스

GIN은 "일반화된 **도치 인덱스**(Generalized **In**verted **Index**)"의 약자입니다 [32]. 이름 그대로 **역색인(Inverted Index)** 구조를 일반화한 것으로, **하나의 테이블 행이 여러 개의 키를 가질 수 있는 경우**에 적합한 인덱스입니다 [33]. GIN 인덱스는 각 "구성 요소 값(component value)"를 키로 하여, 그 값을 포함하는 테이블의 **모든 행 번호 목록(posting list)**을 저장합니다 [33]. 마치 책에서 **단어별 색인**을 만들어 단어가 등장하는 모든 페이지를 기록해두는 것과 비슷한 방식입니다.

이러한 구조 덕분에 **하나의 열에 복수의 요소**가 들어가는 데이터 타입들 – 예를 들어 **배열**, **JSON (키/값)**, **문서형 텍스트(tsvector)** 등에 GIN 인덱스를 많이 사용합니다 [32]. 예를 들어 배열 컬럼에 GIN 인덱스를 만들면 **배열에 특정 값이 포함되어 있는지**를 검사하는 `@>` 또는 `&&` 연산을 효율적으로 처리할 수 있습니다 [34]. GIN 인덱스는 배열의 모든 요소에 대해 별도의 키를 갖추고 있으므로 **"이 값이 들어있는 모든 행"**을 바로 찾아낼 수 있습니다. 마찬가지로 텍스트 전체를 대상으로 하는 **풀텍스트 검색** (`to_tsvector` / `to_tsquery`)에서도 문서에 포함된 **각 단어에 대해 역색인**을 구축해 두어 **특정 단어가 나타나는 문서들을 신속히 검색**합니다.

GIN의 내부 구조: GIN 인덱스는 키로 **개별 요소값**(예: 단어, 배열원소 등)을 사용하고, 그 키가 등장하는 **heap 튜플 ID 리스트**를 저장합니다. 키의 개수가 매우 많을 수 있으므로, GIN은 키를 **B-트리 또는 B+트리 형태**로 관리하며, 각 키에 연결된 **포스팅 리스트(posting list)**는 효율적으로 압축 저장됩니다. (경우에 따라 포스팅 **트리(posting tree)** 형태로 저장되기도 합니다 – 예를 들어 매우 많은 행에 등장하는 키의 경우 트리로 관리.) GIN은 일반적으로 **키 -> [다수의 행]** 매핑 관계이므로, 하나의 행이 여러 키를 가질 때 **동일한 행 위치 정보를 여러 곳에 중복 저장**하게 됩니다. 하지만 **비트 압축**이나 **일괄 갱신 기법** 등을 통해 저장 효율과 갱신 성능을 높이는 최적화가 포함되어 있습니다.

장점: 한 행에 여러 값이 들어가는 경우의 질의를 가속하는 데 특화되어 있습니다. 전형적인 예가 **풀텍스트 검색**인데, GIN 인덱스가 없다면 수백만 건의 문서 텍스트를 일일이 검색해야 할 것을 **몇 개의 단어 키 탐색**으로 대체할 수 있습니다. **JSON 데이터**에서도 특정 키가 존재하는 레코드 찾기나, **배열 값 포함 여부** 찾기 등에 GIN은 뛰어난 성능을 발휘합니다 [32] [35]. 또한 GIN 인덱스는 **여러 조건의 조합 검색**에도 강점을 갖습니다. 예를 들어 **배열에 값 A와 B가 모두 포함된 행**을 찾는 경우, GIN은 **키 A의 포스팅 리스트**와 **키 B의 포스팅 리스트**를 각기 가져와 **교집합**을 구하면 됩니다. 이는 **MANY OR 조건**이나 **문서에 여러 단어 동시 포함 여부** 등을 판별할 때, **여러 비트리 인덱스 결과를 조합하는 것보다 훨씬 효율적**입니다. 또한 **다중 컬럼 GIN 인덱스**를 만들면 각 컬럼에 대한 역색인을 한꺼번에 관리하므로, 한 인덱스로 **여러 컬럼의 포함 질의**를 모두 처리할 수 있습니다 [36]. (예: 두 개의 텍스트 컬럼을 가진 테이블에 하나의 GIN을 걸어 놓으면, 각 컬럼에 대한 키를 모두 관리하고 쿼리에 따라 필요한 키 리스트를 합쳐 결과를 산출합니다. 어느 컬럼 조건이든 **동등한 성능으로** 동작합니다 [36].)

단점: 삽입/갱신 성능 비용이 높습니다. 하나의 행이 여러 키를 가지므로, 행이 추가되거나 바뀔 때 **해당 행의 모든 키에 대한 인덱스 엔트리를 수정**해야 합니다. 특히 **긴 문서**처럼 **키 발생 빈도가 많은 데이터**는 한 행 추가가 수천 개의 키 추가로 이어질 수 있습니다. 이로 인해 GIN 인덱스는 B-트리 등에 비해 **쓰기 부담이 크며**, 트랜잭션 처리 시 WAL 로그 양도 많아질 수 있습니다. PostgreSQL은 이러한 문제를 완화하기 위해 GIN에 **fastupdate**라는 지연 업데이트 기능을 두고 있습니다. 기본값으로 활성화된 이 기능은, 새로 추가되는 키들을 인덱스에 즉시 반영하지 않고 **별도 펜딩 리스트에 모아두었다가** 일정 조건(인덱스 조회 발생 또는 autovacuum 시)에 몰아서 처리합니다. 이를 통해 **쓰기 I/O를 분산**시켜 성능 저하를 줄입니다. 하지만 결국 해당 인덱스를 이용하는 **첫 질의 시에 지연된 업데이트를 반영**해야 하므로, **실시간 업데이트가 매우 빈번하고 조회가 드문 패턴**이라면 GIN 인덱스의 이점이 크지 않을 수 있습니다. 또 하나, GIN 인덱스는 **인덱스 자체만으로는 원본 데이터를 완전히 재구성할 수 없기 때문에 Index-Only Scan을 지원하지 못합니다** [37]. (예: GIN은 특정 단어가 있는 행들을 알려주지만, 그 단어가 몇 번 등장하는지 등의 추가 정보나 문서의 다른 부분은 알 수 없으므로, 최종 결과 출력 시 테이블 접근이 필요합니다.) 따라서 인덱스 **커버링** 용도로는 부적합합니다. 저장 공간 측면에서 GIN은 일반적으로 **인덱스 크기가 큰 편**입니다. 역색인은 많은 키를 보유하므로, 테이블 크기에 비해 인덱스가 상당히 커질 수 있습니다. 하지만 이는 활용상의 트레이드오프이며, 필요하다면 **압축 옵션이나 부분 인덱스** 등을 병행해 조절할 수 있습니다.

예제: GIN 인덱스는 주로 **배열, JSON, 텍스트**에 사용됩니다. 예를 들어 `documents(content text)` 테이블이 있고 여기에 **풀텍스트 검색**을 적용한다고 합시다. PostgreSQL의 텍스트 검색 함수 `to_tsvector`로 내용을 색인하면 각 문서에 등장하는 **단어들의 집합**이 만들어지는데, 이에 대해 GIN 인덱스를 생성하면 특정 단어를 포함하는 문서를 빠르게 찾을 수 있습니다:

```
-- 풀텍스트 검색을 위한 GIN 인덱스 생성 (내용 텍스트 전체에 대해)
CREATE INDEX idx_docs_content_tsv ON documents
USING GIN (to_tsvector('english', content));

-- 인덱스 활용 질의 예시: 'database'와 'index' 단어를 모두 포함한 문서 검색
SELECT id, content
FROM documents
WHERE to_tsvector('english', content) @@ to_tsquery('database & index');
```

위의 인덱스는 각 문서의 텍스트를 `english` 분석기로 토큰화한 결과(단어 목록)에 대해 GIN 역색인을 만든 것입니다. `@@ to_tsquery('database & index')`라는 질의 조건은 **두 단어를 모두 포함**하는 문서를 찾는 것인데, GIN 인덱스는 **'database'라는 키의 포스팅 리스트**와 **'index'라는 키의 포스팅 리스트**를 내부적으로 빠르게 **교집합**하여 해당 문서들을 찾아냅니다. 이는 GIN이 아니었다면 불가능에 가깝거나, 적어도 두 번의 인덱스 검색 후 결과 집합을 소프트웨어적으로 교집합해야 할 작업이지만, GIN은 이를 자체 구조로 효율적으로 처리합니다.

또 다른 예로, `tags text[]` (태그 문자열의 배열) 컬럼을 가진 `articles` 테이블에서 **특정 태그가 붙은 글**을 찾는다고 가정합시다. 이때 태그 배열 컬럼에 GIN 인덱스를 만들고 **배열 포함 연산자**를 사용하면 효과적입니다:

```sql
-- 텍스트 배열 컬럼에 대한 GIN 인덱스 생성
CREATE INDEX idx_articles_tags ON articles USING GIN (tags);

-- 인덱스 활용 질의: 'PostgreSQL' 태그를 가진 모든 글 검색
SELECT title
FROM articles
WHERE tags @> ARRAY['PostgreSQL'];
```

여기서 `@>` 연산자는 **배열이 특정 요소를 포함하는지**를 나타내며, GIN 인덱스가 이 연산을 지원하므로 (`tags @> ...`) 해당 조건으로 **즉시 결과를 찾아낼 수 있습니다** [34] .

BRIN 인덱스

BRIN은 "**블록 범위 인덱스**(Block Range Index)"의 약자이며, **대용량 테이블에 적합한 경량 인덱스**입니다 [38] . BRIN 인덱스는 테이블을 **物理적 페이지 단위로 일정 범위**(기본값 128페이지씩 등)로 묶고, 각 범위에 있는 값들의 **요약 정보(summary)**를 저장합니다 [38] . 가장 흔하게는 해당 범위의 **최소값과 최대값**을 기록하여, 질의 시에 **찾는 값이 해당 범위의 최소~최대 사이에 속하는지 여부**를 보고 필요한 페이지 범위만 읽는 식으로 동작합니다 [39] . 쉽게 말해, 전화번호부에서 **가나다… 순으로 구획을 나눠** 색인을 만드는 것과 비슷합니다.

BRIN의 가장 큰 특징은 **인덱스 크기가 매우 작고 생성 속도가 빠르다**는 것입니다. 각 범위(수백 KB~수 MB 단위)에 대해 몇 바이트에서 수십 바이트 정도의 요약만 저장하므로, 테이블이 수억 건이어도 인덱스는 금방 만들 수 있고 크기도 작게 유지됩니다. 따라서 **매우 큰 테이블**에서 전체 데이터를 모두 인덱싱하기 부담스러울 때 유용합니다. 하지만 BRIN 인덱스가 **효과를 발휘하려면** 테이블의 **데이터가 물리적으로 정렬되어 있거나 유사한 값끼리 뭉쳐있는 경우**가 좋습니다 [38] . 예를 들어 **시간순으로 추가되는 로그 테이블**이나, **자동 증가 ID 순으로 저장되는 테이블**이라면, 일정 블록 내의 값들이 서로 비슷할 것이므로 그 **범위(min~max)가 좁아집니다**. 이때 **특정 값이나 범위를 찾는** 질의라면, BRIN 인덱스는 많은 범위 중에서 **해당 값이 속할 수 있는 범위만 골라서** 테이블을 읽도록 해 줍니다. 반면 데이터가 물리적으로 무작위로 섞여 있다면, 각 범위의 min~max 차이가 매우 커져서 **모든 범위가 모든 값 대상을 가질 수 있게 되고**, 결국 **인덱스가 쓸모없어지는** 상황이 됩니다. 요약하면, **테이블의 물리적 클러스터링을 활용**하여 **대략적인 위치를 빨리 찾도록** 돕는 인덱스가 BRIN입니다 [38] .

장점: 인덱스의 크기와 유지비용이 가장 낮습니다. 테이블이 너무 커서 B-트리 인덱스를 만드는 데만도 시간이 오래 걸리는 경우, BRIN은 훌륭한 대안이 될 수 있습니다. 또한 **추가/갱신 작업 부담이 매우 적습니다**. 새로운 행이 추가되면 해당 행이 속한 범위의 요약정보만 갱신하면 되는데, 이는 대체로 **메모리에 올려진 채로 빠르게 처리**됩니다. (만약 새로운 범위가 생기면 인덱스에 한 줄 요약이 추가되는 정도입니다.) BRIN 인덱스는 다중 컬럼을 지원하므로, 예를 들어 **(date, category)** 두 컬럼에 대한 BRIN을 만들면 각 범위 내에 날짜의 min~max와 카테고리의 min~max를 모두 저장합니다. 이 경우 **날짜나 카테고리 조건, 또는 둘 다 있는 조건**에서 이 인덱스를 활용할 수 있습니다 [40] . (단, 둘 중 하나 조건만 주어져도 동작하기는 하지만, 두 조건 모두를 이용해 범위를 더 좁혀주지는 못합니다. 어차피 각 조건별로 범위를 걸러내는 비용이 작으므로 문제되진 않습니다.) 또한 BRIN은 **여러 개의 작은 인덱스를 만들 필요 없이** 하나의 인덱스로 커버리지가 넓기 때문에, 파티션닝되지 않은 아주 큰 테이블에서도 유용합니다.

단점: 데이터가 물리적으로 정렬 또는 클러스터되지 않은 경우 효과가 미미합니다. 극단적으로 말해, 완전히 랜덤하게 섞인 데이터라면 BRIN 인덱스를 쓰더라도 **걸러지는 페이지 범위가 거의 없으므로** 테이블 풀스캔과 다를 바 없게 됩니다. 따라서 BRIN은 **시간 흐름에 따라 추가되는 로그, 시계열 데이터**처럼 자연 정렬이 있는 경우에 주로 쓰입니다 [38] . 또한 **정확한 키 값을 저장하지 않기 때문에**, 인덱스만으로는 정확한 값을 찾을 수 없습니다. 인덱스는 후보 **페이지 블록**만 알려주고, 실제 조건에 맞는지는 각 블록을 읽으며 **재확인**해야 합니다. 즉, **BRIN 인덱스를 사용한 검색은 "인덱스 -> 페이지 덩어리 -> 실제 행 확인"**의 단계를 거칩니다. 하지만 이는 의도된 동작이며, 걸러야 할 페이지의 비율이 충분히 작으면 여전히 이득입니다. BRIN 인덱스는 **Index-Only Scan**이 불가능하다는 점도 유의해야 합니다. (인덱스에는 min/max 등의 요약만 있고 각 행의 개별 값은 없으므로, 인덱스만으로 질의 결과를 구성할 수 없습니다. 반드시 테이블 페이지를 읽어야 합니다. 다만 보통 BRIN은 결과 행 자체가 엄청 많을 상황에는 쓰이지 않으므로 문제가 되진 않습니

다.) 마지막으로, **정밀한 검색이 필요한 경우에는 BRIN을 보조 수단으로만** 사용해야 합니다. 예를 들어 아주 큰 테이블에서 **80% 이상의 행이 범위 조건에 걸리는 경우**, BRIN은 대부분의 범위를 다 읽어야 해서 효과가 없습니다. 이런 땐 차라리 B-트리가 낫습니다.

예제: 수십억 행을 가진 `logs(timestamp timestamp, level int, message text)` 테이블이 있다고 가정하고, 주로 **날짜 범위**로 조회한다고 해봅시다. 이 경우 `timestamp` 컬럼에 BRIN 인덱스를 걸면 특정 기간의 로그를 읽을 때 전체 테이블을 스캔하지 않고 필요한 블록만 읽도록 도와줍니다:

```
-- timestamp 컬럼에 대한 BRIN 인덱스 생성 (기본 pages_per_range 사용)
CREATE INDEX idx_logs_time ON logs USING BRIN (timestamp);

-- 인덱스 활용 질의 예시: 2023년 1월의 로그 검색
SELECT *
FROM logs
WHERE timestamp >= '2023-01-01'
  AND timestamp < '2023-02-01';
```

위 쿼리를 실행하면 BRIN 인덱스 `idx_logs_time`은 **각 페이지 범위에 기록된 최소~최대 타임스탬프**를 참조하여, 2023-01-01~2023-02-01 범위의 값이 존재할 가능성이 있는 블록들만 선별합니다 [39]. 만약 로그 테이블이 시간 순으로 축적되어 있다면, 이 기간에 해당하는 블록 외에는 전혀 읽지 않으므로 **대용량 데이터에서 매우 효율적인 범위 검색**이 가능합니다.

또한, `logs` 테이블에 `level` (로그 레벨) 컬럼이 있다고 했을 때, 예컨대 **ERROR 레벨(높은 숫자) 로그는 드물고 INFO 레벨(낮은 숫자) 로그는 많다**고 합시다. 이러한 상황에서는 `(level, timestamp)` 복합 BRIN 인덱스를 만들어서 **level별로도 범위를 염두**에 두게 할 수 있습니다:

```
-- level, timestamp 복합 BRIN 인덱스 생성
CREATE INDEX idx_logs_level_time ON logs USING BRIN (level, timestamp);

-- 인덱스 활용 질의: ERROR 레벨의 2023년 1월 로그
SELECT *
FROM logs
WHERE level = 5  -- (예: 5 = ERROR)
  AND timestamp >= '2023-01-01'
  AND timestamp < '2023-02-01';
```

이 경우 BRIN 인덱스는 **level=5인 로그가 저장된 페이지 범위 중** 시간 조건에 맞는 범위만 읽도록 할 것입니다. 다만 복합 BRIN에서 **두 번째 컬럼 이후는 선별에 큰 영향이 없을 수도** 있습니다 [41]. (왜냐하면 첫 컬럼 level의 min~max가 이미 5~5로 좁혀져 나올 것이므로, timestamp는 사실상 해당 범위 내 전체가 대상이 됩니다.) 따라서 이런 경우 차라리 **두 개의 BRIN 인덱스(level별 하나, 시간별 하나)**를 만들어서 **플래너가 조건에 따라 둘 중 유리한 쪽을 쓰도록** 하는 방법도 있습니다.

고급 인덱스 활용 기법

앞서 각 인덱스 유형의 기본 동작과 용도를 살펴보았습니다. 이제는 **여러 컬럼을 함께 인덱싱**하거나, **컬럼이 아닌 표현식에 인덱스**를 걸거나, **일부 행만 인덱싱**하거나, **인덱스에 추가 컬럼을 포함**시키는 등, PostgreSQL이 제공하는 고급 인덱스 기능과 최적화 전략을 알아보겠습니다. 이러한 기법들은 제대로 활용하면 **불필요한 인덱스를 줄이고 성능을 극대**

화할 수 있는 반면, 잘못 사용하면 **인덱스 효율을 떨어뜨리거나 플래너가 인덱스를 못 쓰게** 만들 수 있으므로 이해가 필요합니다.

다중 컬럼 인덱스 (Multicolumn Index)

인덱스는 **둘 이상의 컬럼을 조합하여** 만들 수 있습니다. **복합 인덱스** 또는 **다중 열 인덱스**라고 불리는 이 기능은, 특히 **여러 컬럼이 동시에 조건에 사용**되는 쿼리에서 유용합니다 [42] [43]. 예를 들어 `(col1, col2)` 복합 인덱스가 있으면 `WHERE col1 = X AND col2 = Y` 같은 쿼리에 대해 하나의 인덱스만으로 두 조건을 모두 빠르게 처리할 수 있습니다 [43]. PostgreSQL에서는 **B-Tree, GiST, GIN, BRIN** 인덱스 유형이 다중 컬럼 인덱스를 지원하며, 최대 32개 컬럼까지 한 인덱스에 포함할 수 있습니다 [31]. (Hash와 SP-GiST는 다중 컬럼을 지원하지 않습니다 [31].) 다중 컬럼 인덱스는 **검색 조건 패턴**을 잘 분석하여 설계해야 효과적이며, 경우에 따라 **여러 단일 컬럼 인덱스를 조합하는 것**이 더 나을 수도 있습니다.

B-Tree 복합 인덱스: B-트리 인덱스에서 다중 컬럼은 **사전식(lexicographical) 순서**로 정렬되어 저장됩니다. 예를 들어 `(a, b, c)` 인덱스가 있으면, 키는 우선 `a` 값으로 정렬되고, `a` 값이 같으면 `b` 값으로, 둘 다 같으면 `c` 값으로 정렬됩니다. 이로 인해 **인덱스 검색 효율은 선두 열에 크게 좌우**됩니다 [44]. **선두 컬럼부터 순서대로 조건이 주어질 때** 인덱스 탐색 범위를 크게 좁힐 수 있지만, **첫 컬럼에 조건이 없으면** 인덱스 전체를 훑어야 하므로 실효성이 거의 없습니다 [45]. 정확한 규칙은 다음과 같습니다: **인덱스의 앞쪽 컬럼들에 대한 동등 조건**이 주어지고, **그 다음 컬럼에 범위 조건**이 주어졌을 때, B-트리는 해당 범위만 탐색하면 됩니다 [44]. 그러나 **첫 컬럼 자체에 범위 조건이 오면** 그 뒤 컬럼들은 인덱스 탐색 범위를 줄이진 못하고 인덱스 내부 검사로만 활용됩니다 [46]. 또한 **첫 컬럼에 조건이 없고 두 번째 이후 컬럼에만 조건이 있는 경우** 인덱스를 전체 범위 스캔해야 하므로, 보통 **시퀀셜 스캔보다도 느려서** 플래너가 사용하지 않습니다 [47]. 따라서 **B-트리 복합 인덱스 설계의 핵심**은 **자주 함께 쓰이는 컬럼들을 포함**하되, **가장 선별력이 높은 컬럼을 앞에 배치**하는 것입니다. 일반적으로 첫 컬럼은 **자주** `WHERE` **에 등장하고 선택도가 높은** 컬럼이어야 합니다. 다중 인덱스는 2~3개 컬럼 조합까지는 유용하나, **4개 이상 많은 컬럼을 한 인덱스로 만드는 것은 권장되지 않습니다** [48]. (그럴 바엔 다른 전략을 고려해야 한다는 것이 문서의 조언입니다.) 예를 들어, `WHERE region = ? AND category = ? AND date >= ?` 같은 쿼리를 자주 쓰는 경우 `(region, category, date)` 인덱스를 만들 수 있습니다. 이때 `region`과 `category`는 보통 동등 조건이므로 선두에 두고, `date`는 범위 조건일 가능성이 높으므로 뒤에 둡니다. 이 인덱스가 있으면 `region`과 `category`가 일치하는 범위만 탐색하고 그 안에서 `date`로 필요한 부분만 스캔하게 됩니다.

GiST 복합 인덱스: GiST도 다중 컬럼을 지원하지만, **첫 컬럼의 영향력이 특히 큽니다** [28]. GiST 인덱스는 각 컬럼마다 자체 전략이 있을 수 있으나, 트리 구조상 첫 컬럼의 값으로 **상위 분기가 결정**됩니다. 만약 **첫 컬럼의 값 종류가 너무 적다면** (예: 대부분 동일한 값), 트리가 효과적으로 분산되지 않아 인덱스 효율이 떨어집니다 [49]. 추가 컬럼들은 **후속 필터링**에 사용될 뿐 인덱스 탐색 범위를 크게 줄이지 못합니다. 따라서 GiST 복합 인덱스 역시 **첫 컬럼을 신중히 선택**해야 하며, 종종 **공간+속성** 조합으로 쓰는 정도가 적합합니다. (예: 위치 + 카테고리 인덱스에서 위치로 1차 좁히고 카테고리로 2차 필터.) GiST는 특수 용도가 많으므로, **복합 인덱스의 효용도 해당 용도에 종속**적입니다.

GIN 복합 인덱스: GIN의 경우 특이하게도 **인덱스 탐색 효율이 컬럼 순서에 무관**합니다 [36]. GIN 인덱스는 내부적으로 각 컬럼별 별도 역색인을 관리하므로, 복합 GIN이라도 **사실상 여러 GIN 인덱스를 하나로 합친 격**입니다. 따라서 쿼리가 복합 GIN 인덱스의 일부 컬럼만 조건으로 사용해도 **모든 해당 컬럼의 키를 빠르게 찾아올 수 있으며**, 여러 컬럼 조건이면 **각각 찾은 결과를 내부적으로 병합**합니다. 그 때문에 **컬럼 순서나 개수에 따른 성능 차이가 거의 없고**, 굳이 여러 GIN을 각각 만드는 것보다 한데 모아 관리할 수 있다는 이점이 있습니다 [36]. (단, GIN에 너무 많은 컬럼을 넣으면 인덱스가 비대해지고 쓰기 부하가 커질 수 있으므로 현실적인 한계는 있습니다. 일반적으로 2~3개 관련 컬럼 정도 묶는 데에 그칩니다.)

BRIN 복합 인덱스: BRIN도 여러 컬럼을 가질 수 있는데, 이 경우 각 컬럼별로 **요약 정보(min, max 등)**를 인덱스 테이블에 함께 저장합니다 [40]. BRIN은 기본적으로 **컬럼별 독립적인 요약**이므로, **특정 컬럼 조건이 있으면 그 컬럼의 요약으로 범위를 판단**하고, 다중 컬럼 모두 조건이 있다면 **각각 범위를 계산한 후 교집합** 범위를 취합니다. 흥미로운 점은, PostgreSQL 문서에 따르면 **굳이 여러 BRIN 인덱스로 나누기보다 한 인덱스에 넣는 편**이 일반적으로 낫다고 합니다 [41]. (다만 **pages_per_range 파라미터**를 컬럼마다 다르게 주고 싶을 땐 별도 인덱스가 필요할 수 있습니다.) 복합

BRIN의 활용 예를 들면, (`date, category`) 인덱스가 있으면 `WHERE date = ? AND category = ?`에서 두 조건을 모두 만족하는 페이지 범위만 선택할 수 있습니다. 그러나 `WHERE category = ?` 단독 조건이라면 date 요약은 무시되고 category 기준만 보게 됩니다.

복합 인덱스 설계 팁: 일반적으로 **한 테이블에 존재하는 여러 단일 인덱스들이 복합적으로 활용될 수 있다면**, 굳이 복합 인덱스를 만들지 않아도 되는 경우가 있습니다. PostgreSQL **플래너는 여러 인덱스를 함께 사용**하여 **비트맵 인덱스 스캔(bitmap index scan)**을 수행할 수 있습니다 [50]. 예를 들어 `WHERE col1 = X AND col2 = Y` 질의에 대해 `col1`과 `col2` 각각에 단일 B-트리 인덱스가 있으면, 두 인덱스를 모두 탐색한 결과를 **비트맵으로 결합**하여 교집합만 실제 접근하는 방식입니다. 이 방법은 **둘 중 하나의 컬럼 조건만으로는 대상이 너무 넓을 때** 유용합니다. 따라서 **모든 복합 인덱스가 항상 단일 인덱스보다 우월한 것은 아니며**, 때로는 단일 인덱스 두 개로 대체되기도 합니다. 복합 인덱스를 만들지 않고 **필요한 조합마다 AND/OR로 인덱스 결합**을 활용하는 것이 유연성을 줄 수도 있습니다. 다만, **OR 조건** 등에서는 인덱스 결합이 항상 효율적이지 않을 수 있으므로, 자주 함께 사용되는 조건이면 복합 인덱스를 고민하는 것이 좋습니다.

마지막으로, **너무 많은 컬럼을 한 인덱스에 넣는 것**은 피해 합니다. 3개를 넘는 복합 인덱스는 대부분의 일반적인 테이블에서 **효과가 미미**하며, 만약 4개 이상의 컬럼 조합으로만 성능 문제가 있다면 **테이블 구조 개편**이나 **쿼리 리팩토링**을 고려하는 편이 바람직합니다 [48]. 복합 인덱스는 **2개, 많아야 3개 컬럼 조합**이 일반적이며, 그 이상은 매우 특수한 경우입니다.

함수 기반 인덱스 (Indexes on Expressions)

일반적인 인덱스는 **테이블의 실제 컬럼 값**을 키로 삼지만, PostgreSQL에서는 **함수 또는 표현식 결과를 인덱스 키로** 사용할 수도 있습니다 [51]. 이를 흔히 **함수 기반 인덱스** 혹은 **표현식 인덱스**라고 부릅니다. 이 기능은 **컬럼 값을 어떤 함수나 연산으로 변환한 결과**를 자주 질의하는 경우에 유용합니다 [51]. 예를 들어 테이블에 `name` 컬럼이 있고 대소문자를 구분하지 않고 검색을 한다면, `WHERE LOWER(name) = 'jane'` 같은 조건을 많이 사용하게 됩니다. 이때 `LOWER(name)`**에 인덱스**를 만들어 두면, 매 질의마다 문자열 함수를 계산하며 검색하는 대신 **인덱스를 통해 바로 결과를 찾을** 수 있습니다 [52]. PostgreSQL 공식 문서의 예시를 보면, 아래와 같이 표현식 인덱스를 생성하고 활용합니다:

```
-- 컬럼의 소문자 값을 키로 하는 표현식 인덱스 생성
CREATE INDEX test1_lower_col1_idx ON test1 (LOWER(col1));

-- 활용 예: 인덱스가 LOWER(col1) = 'value' 조건을 가속
SELECT *
FROM test1
WHERE LOWER(col1) = 'value';
```

위와 같이 하면, 질의 조건 `LOWER(col1) = 'value'`는 **인덱스에 저장된 계산 결과**를 바로 이요하여 처리되므로, 일반 인덱스의 `col1 = 'Value'`처럼 빠르게 동작합니다 [53] [54]. 결과적으로 **질의 시에는 표현식을 다시 계산할 필요가 없고, 단순한 값 비교로 취급**됩니다 [55]. 표현식 인덱스는 이처럼 **계산 비용이 있는 비교를 빠르게** 만들거나, **트랜스포메이션을 적용한 값**(예: 문자열 길이, 수식 결과 등)으로 검색해야 할 때 효과적입니다. 또한 일반 고유 인덱스로는 강제할 수 없는 **비즈니스 규칙을 UNIQUE 표현식 인덱스**로 구현할 수도 있습니다 [56]. 예를 들어 **대소문자 구분 없이 유니크**한 값 제한은 `LOWER(col)`에 UNIQUE 인덱스를 걸면 실현できます.

장점: 쿼리에 사용되는 계산된 값에 대한 **별도 저장소** 역할을 하므로, **복잡한 연산도 O(log N)**으로 검색 가능하게 해줍니다. 흔히 사용되는 사례는 **대소문자 변환**, **문자열 일부 추출**(예: 전화번호 끝 4자리 등), **산식 결과**(예: 좌표 변환) 등입니다. 또한 표현식 인덱스는 **컬럼 조합으로 만든 계산 결과**도 인덱싱할 수 있어 (예: `col1 || '-' || col2`) 편리합니다 [57]. UNIQUE 옵션을 주면 **그 표현식 결과에 대한 유일성 제약**도 가능합니다 [56]. 이는 일반 컬럼 조합으로는 표현하기 어려운 **비즈니스 규칙을 데이터베이스 수준에서 보장**하는 수단이 됩니다.

단점: INSERT/UPDATE 시 부하가 증가합니다. 표현식 인덱스를 유지하려면 **각 행이 추가되거나 변경될 때마다 표현식을 다시 계산**해야 합니다 [54]. CPU 계산 비용이 높은 함수라면, 많은 행이 쓰일 때 병목이 될 수 있습니다. 또한 표현식 인덱스가 걸려 있으면, 해당 표현식에 참여하는 컬럼이 변경되지 않더라도 **HOT 업데이트가 불가능해져** (튜플 자체에 인덱스 참조가 있으므로) **업데이트 비용이 늘어납니다** [54]. 따라서 **쓰기 빈도가 높은 컬럼이나 복잡한 함수**에는 신중해야 합니다. 표현식 인덱스를 남용하면 오히려 전체 시스템 성능이 저하될 수 있으므로, **"계산 비용 vs. 저장/유지 비용"**을 저울질해서 정말 **질의 성능 향상 효과가 큰 경우에만** 사용해야 합니다 [54]. 또 한 가지 유의점은, **인덱스가 특정 표현식 형태로 정의되었기 때문에 쿼리에서도 똑같은 형태로 써야** 인덱스를 탈 수 있다는 것입니다. 예를 들어 `LOWER(col)`로 인덱스를 만들었는데, 어떤 쿼리가 `WHERE lower(col || '') = 'abc'`처럼 미묘하게 다른 표현을 쓰면 (의미상 동일해도) 인덱스를 사용하지 못합니다. PostgreSQL은 **표현식이 문자 그대로 일치**해야 인덱스를 활용하므로, 애플리케이션 쿼리 작성시 이 점을 주의해야 합니다. 마지막으로, 표현식 인덱스에 사용되는 **함수나 연산자는 반드시** IMMUTABLE(불변) **속성**이어야 합니다 [58]. 현재 시간이나 랜덤값처럼 호출 시마다 결과가 달라지거나, 다른 테이블을 참고하는 함수 등은 인덱스 표현식에 사용할 수 없습니다. 사용자 정의 함수를 활용하려면 생성 시 `IMMUTABLE`로 선언해야 인덱스에 사용할 수 있습니다 [58].

예제: 사용자의 이름을 저장한 `users(full_name)` 컬럼에 **대소문자 구분 없이 검색**을 지원하고 싶다면 아래와 같이 표현식 인덱스를 만들 수 있습니다.

```
-- 이름 컬럼의 소문자값에 대한 표현식 인덱스 생성 (대소문자 무시 검색 용도)
CREATE INDEX idx_users_name_lower ON users (LOWER(full_name));

-- 활용 예: 대소문자 구분 없이 특정 이름 검색
SELECT *
FROM users
WHERE LOWER(full_name) = 'john doe';
```

이렇게 하면 인덱스에는 `full_name`의 소문자 버전이 저장되고, 질의도 `LOWER(full_name)` 조건을 사용하므로 **인덱스를 완전히 활용**하게 됩니다 [53]. 반면 인덱스 없이 저 질의를 실행했다면 모든 행의 `full_name`에 대해 LOWER를 계산하며 비교해야 하므로 훨씬 느렸을 것입니다.

또 다른 예로, `products(name, price)` 테이블에서 **이름과 가격을 결합한 키**로 검색하는 경우를 생각해봅시다. `WHERE name || ':' || price = 'Book:19.99'` 같은 조건은 일반 인덱스로는 지원되지 않지만, 해당 결합 표현식에 인덱스를 만들면 가능합니다:

```
-- 이름과 가격을 결합한 표현식에 대한 인덱스
CREATE INDEX idx_products_nameprice ON products ((name || ':' || price));

-- 활용 예: 결합 문자열로 제품 찾기
SELECT *
FROM products
WHERE name || ':' || price = 'Book:19.99';
```

이 인덱스는 `name`과 `price`를 문자열로 이어 붙인 결과를 키로 가지며, 질의에서도 똑같이 쓰였기 때문에 동작합니다. 다만 이런 케이스는 합리적인 사용 예는 아니고, **응용에서 복합 검색을 간단히 하기 위해** 가끔 쓰일 수 있음을 보여주는 것입니다. (실제로는 별도의 파싱 없이 `name`과 `price`를 한 번에 찾고자 할 때 쓸 수 있지만, 일반적으로는 권장되는 방식은 아닙니다.)

부분 인덱스 (Partial Index)

부분 인덱스(Partial Index)는 테이블의 **일부 행만을 인덱싱**하는 것입니다[59]. 인덱스를 만들 때 `WHERE <조건>` 절을 함께 지정하면, 그 조건을 만족하는 행들만 인덱스에 포함됩니다[60]. 이렇게 하면 **전체 테이블보다 작은 범위에 대해서만 인덱스 유지**를 하면 되므로, **인덱스 크기를 줄이고 유지 비용을 절약**할 수 있습니다[61]. 부분 인덱스는 **특정 값들이 너무 흔해서 인덱스 효율이 떨어지는 경우**나, **특정 조건의 행들만 자주 조회되는 경우** 등에 유용합니다[61][62].

부분 인덱스를 고려해볼 대표적인 상황은 **테이블에 한두 가지 값이 압도적으로 많이 등장하는 컬럼**입니다[61]. 예를 들어 컬럼 값 중 90%가 `NULL`이고 10%만 유의미한 값이라면, 일반 인덱스를 만들 경우 `NULL`에 대한 엔트리가 가득하지만 정작 쿼리는 `WHERE col IS NOT NULL`만 자주 쓰인다면 인덱스 90%는 죽은 공간입니다. 이때 `WHERE col IS NOT NULL` **조건의 부분 인덱스**를 만들면, `NOT NULL`인 행만 인덱싱하여 크기를 1/10 수준으로 줄일 수 있고, 인덱스 탐색도 빨라집니다. 어떻게 보면 **"어차피 인덱스로 안 쓰일 값"**을 빼버리는 셈입니다[61]. 공식 문서 예시로, **웹 서버 로그 테이블**에서 **내부 트래픽 IP**들은 너무 흔해서 대부분의 검색(외부 IP 조회)에 인덱스가 도움 안 되므로, **자사 IP가 아닌 것만 인덱싱**하는 경우가 소개됩니다[61][63]. 이처럼 **자주 조회되는 부분만 인덱싱하고 자주 안 하는 부분은 빼거나**, 또는 **너무 흔해서 인덱스효과 없는 값은 빼는 전략**이 부분 인덱스입니다[61][62].

또 다른 활용은 **데이터 일부만 관심 대상인 경우**입니다[62]. 예를 들어 주문 테이블에서 `status = 'pending'`인 주문은 소수지만 중요하게 자주 조회되고, 나머지 완료된 주문은 수백만 건이라 관심 없다면, `WHERE status = 'pending'` 부분 인덱스를 두어 **진행 중 주문** 검색을 최적화할 수 있습니다[62]. 그렇게 하면 완료된 주문을 조회할 때는 인덱스를 안 쓰고 풀스캔하겠지만, 어차피 완료 주문은 많아서 인덱스 있어도 거의 효용이 없을 수 있습니다.

부분 인덱스는 **고유 제약**에도 활용됩니다. 예를 들어, 어떤 표에서 `status = 'active'`인 행들만 고유해야 하고 다른 행들은 상관 없다면, `WHERE status = 'active'` 조건과 함께 UNIQUE 인덱스를 만들면 **활성 레코드 간의 유일성**을 보장할 수 있습니다[64][65]. (예: 한 사용자에 대해 active 주문은 하나만 존재해야 한다 등.) 이 기법으로 **널(null)**에 대한 제약도 구현 가능하며, 공식 문서에서는 **"컬럼에 NULL이 하나만 존재하도록"** UNIQUE 부분 인덱스로 제약을 거는 예시도 있습니다[66].

장점: 필요한 일부만 인덱싱하므로 **인덱스 크기가 작아져 메모리/디스크 효율**이 좋아지고, 그 범위 내 질의는 더욱 빠릅니다[61]. **인덱스 유지 오버헤드도 감소**합니다. 예를 들어 자주 변하는 값인데 굳이 인덱싱할 필요 없는 행이라면, 부분 인덱스로 제외하여 업데이트 시 인덱스 갱신 비용을 들이지 않을 수 있습니다[61]. 또한 **특정 질의패턴에 대해 플래너가 잘못된 판단을 내릴 때** 부분 인덱스로 강제 방향을 줄 수도 있습니다[67]. (그러나 이는 매우 신중해야 하는 사항입니다. 일반적으로 PostgreSQL 플래너는 빈도 높은 값에 대해서는 알아서 인덱스를 피하는 등 합리적 결정을 하므로, 플래너가 오판한다면 통계 조정이나 버그 리포트 고려가 먼저입니다[68].) 부분 인덱스는 **동적 파티셔닝** 비슷하게 활용할 수도 있는데, 예를 들어 **최근 1년 데이터만 인덱싱**해서, 1년 지난 데이터는 인덱스 없이 냅둔다든지 하는 것이 가능합니다. 이런 경우 오래된 데이터 조회는 느려지겠지만, 새로운 데이터에 대한 인덱스 유지비용을 줄일 수 있습니다 (물론 보다 세련된 방법은 **파티션 테이블**을 사용하는 것입니다[69]).

단점: 설계와 관리가 어렵습니다. 부분 인덱스는 **인덱스 조건(predicates)**이 **질의 조건에 포함되어야만 사용**됩니다[70]. **플래너는 쿼리의 WHERE이 인덱스 프레디킷을 "수학적으로 포함함"을 추론할 수 있어야** 그 인덱스를 씁니다[70]. 단순한 동일, 부등호 관계 등은 비교적 잘 인식하지만, **논리가 복잡하거나 형태가 다르면 인덱스를 못 씁니다**[71]. 예를 들어 `WHERE x < 5`로 부분 인덱스를 만들었다면, 쿼리에서 `WHERE x < 4`는 인덱스를 쓸 수 있지만 `WHERE x < (1 + 3)`처럼 변수나 표현식으로 주어지면 **계획 단계에서 그 값이 확정되지 않는 한** 인덱스를 사용하지 않습니다[72]. 또한 **프레디킷과 정확히 동일한 조건이 WHERE에 있지 않고 간접적으로 참일 경우** (논리적으로 동치이지만 형태가 다른 경우) PostgreSQL은 일반적으로 판별하지 못합니다[71]. 따라서 부분 인덱스가 빛을 보려면, **애플리케이션 쿼리가 항상 인덱스 프레디킷을 포함**하도록 신경 써야 합니다. 예를 들어 `WHERE status = 'active' AND ...` 조건을 항상 넣는 식으로 협약이 필요합니다.

또한 부분 인덱스는 **데이터 분포가 변하면 비효율적이 될 수 있습니다**[73]. 예를 들어 처음에는 값 A가 드물어서 A만 빼고 인덱스를 만들었는데, 시간이 지나 A 값이 많아지면 인덱스가 커지고 가치가 떨어집니다. 이러한 경우 **부분 인덱스를**

주기적으로 재평가하여 조건을 바꾸거나 인덱스를 재작성해야 하는 부담이 있습니다 [73]. 이런 유지관리 부담 때문에 부분 인덱스는 **데이터 분포나 업무 규칙이 안정적인 경우**에 주로 사용합니다.

마지막으로, **부분 인덱스로 파티셔닝을 흉내내는 것은 금물**입니다 [74] [75]. 서로 다른 조건으로 여러 개의 부분 인덱스를 만들어 놓으면 플래너가 쿼리마다 **모든 부분 인덱스를 다 검사**해봐야 하는 경우가 생겨 오히려 더 느려질 수 있습니다 [75]. 예를 들어 `category = 1`부터 N까지 각각 부분 인덱스를 만드는 것보다, 차라리 `(category, data)` 복합 B-트리 인덱스 하나가 효율적입니다 [76] [77]. 부분 인덱스 간에 상호 배타적 관계를 Planner는 이해하지 못하므로, **여러 부분 인덱스를 남발하는 건 좋지 않습니다** [75]. 정말 데이터 부분별로 분리가 필요하다면, **아예 테이블을 파티션**으로 쪼개는 것이 올바른 해법입니다 [78].

예제: 주문 테이블 `orders(status, order_no, ...)`가 있다고 하고, `status = 'pending'`인 주문(미처리 주문)은 소수이지만 자주 조회된다고 합시다. 이때 **미처리 주문에 대해서만** 인덱스를 생성하면, 완료된 주문은 인덱스 유지 대상에서 제외되어 쓰기 부하를 줄이고, 미처리 주문 조회는 빨라집니다:

```
-- 'pending' 상태 주문만 인덱싱하는 부분 인덱스 생성
CREATE INDEX idx_orders_pending_no ON orders (order_no)
WHERE status = 'pending';

-- 활용 예: 미처리 주문 조회 (인덱스 사용)
SELECT *
FROM orders
WHERE status = 'pending' AND order_no < 10000;
```

위 부분 인덱스는 `status = 'pending'`인 행들의 `order_no`만 인덱싱합니다. 따라서 `status` 조건이 포함된 질의(예에서처럼)에만 사용됩니다. 특히 `status = 'pending'`인 행이 얼마 없다면 인덱스가 매우 작을 것이고, `order_no < 10000` 같은 추가 조건이 있으면 그 작은 인덱스 안에서 범위를 찾아 금방 결과를 얻습니다 [79]. 반면 `status = 'completed'`와 같은 조건의 질의는 이 인덱스가 전혀 도움을 주지 않고, 해당 경우는 아예 인덱스 없는 것과 동일하게 풀스캔할 것입니다. 이처럼 **업무상 중요하고 자주 조회되는 부분에만 인덱스를 제공**하는 것이 부분 인덱스의 핵심입니다.

공식 문서의 다른 예로, **웹 로그 테이블**에서 사내 IP를 제외한 외부 IP만 인덱싱하는 부분 인덱스 사례가 있습니다 [63] [80]. 이는 "어차피 사내 IP로 조회하는 경우는 거의 없으니 인덱스를 생략하고, 외부 IP만 인덱스로 관리하여 크기를 줄이자"는 전략입니다. 또한 **부분 인덱스 + 고유 제약** 조합으로, 특정 조건에서만 고유성을 요구하는 비즈니스 규칙을 만들 수 있습니다 [64] [65]. 예를 들어 한 테이블에서 `flag = true`인 경우에만 `user_id`가 유일해야 한다면:

```
-- flag=true인 경우에만 user_id 유일성 보장하는 부분 인덱스
CREATE UNIQUE INDEX idx_sample_user_unique ON sample_table(user_id)
WHERE flag = true;
```

이렇게 하면 `flag = true`인 행들끼리만 `user_id` 중복을 막고, `flag = false`인 행들은 중복되든 말든 상관없게 됩니다. 이처럼 부분 인덱스는 **데이터의 부분적인 제약이나 최적화**를 가능하게 해줍니다.

커버링 인덱스와 Index-Only Scan (INCLUDE 컬럼)

커버링 인덱스(Covering Index)란 **쿼리가 필요로 하는 모든 컬럼 데이터를 인덱스가 포함하여, 테이블에 접근하지 않고도** 결과를 반환할 수 있는 인덱스를 말합니다 [81] [82]. 일반적으로 PostgreSQL의 인덱스는 **보조 인덱스(secondary index)**이므로, 인덱스를 탐색한 후 **실제 테이블(Heap)**에 가서 해당 행을 읽어오는 과정이 필요합니다 [83]. 하지만 **인덱스만으로** 필요한 데이터가 모두 확보된다면, 이러한 **랜덤 테이블 접근을 생략**하여 성능을 높일 수 있

습니다 [83] [81]. 특히 **디스크 기반 HDD**에서 랜덤 I/O를 크게 줄일 수 있어, **Index-Only Scan** 최적화는 중요한 기술입니다 [84] [85].

PostgreSQL은 **일부 인덱스 유형(B-Tree 등)**에 대해 **Index-Only Scan**을 지원합니다 [86]. 다만, **쿼리가 요구하는 모든 컬럼**이 인덱스에 있어야 실제로 **Index-Only Scan**이 가능합니다 [82]. 이를 달성하기 위해 도입된 기능이 `INCLUDE` **절**입니다 [87] [88]. `INCLUDE`를 사용하면, 인덱스를 만들 때 **검색 키로 사용하지 않을 추가 컬럼들**을 인덱스에 **부가적으로 저장**할 수 있습니다 [89] [90]. 이러한 컬럼들은 **비검색용 페이로드(payload)**라고 불리며, **인덱스의 트리 구조에는 관여하지 않고** 리프 페이지에만 저장됩니다 [91] [92]. 덕분에 인덱스 순서에는 영향 주지 않으면서도, 인덱스가 갖고 있는 부가 정보로서 **쿼리 결과를 충족시킬 추가 데이터**를 담을 수 있습니다 [91] [93].

예를 들어, 자주 사용하는 쿼리가 `SELECT y FROM table WHERE x = 'key'` 라고 합시다 [82]. 전통적인 방법으로는 `x`에 대한 인덱스를 만들고, 질의 시에 인덱스로 위치를 찾은 뒤 해당 테이블 튜플을 읽어와 `y`를 얻었습니다. 그런데 `x`**로만 탐색하고** `y` **값을 읽는 작업이 빈번하다면**, 인덱스를 약간 키우더라도 `y` 값을 같이 담아두면 테이블 접근을 생략할 수 있습니다 [87] [88]. `CREATE INDEX ... ON table(x) INCLUDE(y)` 와 같이 하면, 인덱스 키는 여전히 `x`지만 인덱스 항목에 `y` 값도 함께 기록됩니다 [94] [88]. 이제 `SELECT y FROM ... WHERE x='key'`를 수행하면 **인덱스에서 곧바로 x='key'인 항목의 y 값을 읽어와** 결과를 반환할 수 있고, 테이블을 전혀 건드리지 않습니다 [82] [95].

동작 원리: Index-Only Scan을 사용할 때는 각 인덱스 튜플에 대한 **가시성 검사(MVCC visibility)**가 필요합니다 [95]. PostgreSQL은 **Visibility Map**이라고 해서, 테이블의 각 페이지에 **"이 페이지의 모든 튜플이 현재 트랜잭션에 가시적이다"**라는 비트를 관리합니다 [96]. Index-Only Scan시 인덱스에서 튜플을 찾으면 테이블을 안 가더라도, **해당 튜플의 페이지의 VM 비트를 확인**합니다 [97]. 만약 해당 페이지가 **완전히 손대지 않은 안정된 데이터**라면 VM 비트가 설정되어 있고, 이 경우 **테이블을 확인하지 않아도 된다**고 판단하여 인덱스 값만으로 결과를 사용합니다 [97]. 하지만 VM 비트가 꺼져 있다면 (해당 페이지에 최근 변경된 튜플이 있을 수 있음) 안전을 위해 **해당 테이블 튜플을 읽어 실제 가시 여부를 확인**해야 합니다 [96] [98]. 그러므로 **Index-Only Scan의 실효성**은 **테이블의 대부분 페이지가 불변 상태(모두 커밋되고 변경 없고 vacuum됨)**인지에 달려 있습니다 [99] [100]. 일반적으로 **오래된 히스토리 데이터 테이블**이나, **자주 변하지 않는 테이블**에서 Index-Only Scan 효율이 높습니다 [99] [100].

INCLUDE 컬럼 제약: INCLUDE로 추가된 컬럼은 **인덱스의 키가 아니므로**, 인덱스 탐색 조건에 사용될 수 없습니다 [89]. 또한 인덱스가 UNIQUE인 경우, **고유 제약은 키 컬럼에만 적용**되고 INCLUDE 컬럼은 무시됩니다 [101]. 예를 들어 `CREATE UNIQUE INDEX idx ON t(x) INCLUDE(y)`라면, `x`만 유일조건이고 `y`는 관계없습니다 [101]. (이 점을 이용해, **주 컬럼은 고유하지만 보조 정보는 중복 허용**하는 제약을 깔끔하게 구현할 수 있습니다. `INCLUDE`는 UNIQUE 제약 문법에도 지원되어 `UNIQUE (x) INCLUDE (y)` 같은 식으로도 사용할 수 있습니다 [101].) INCLUDE된 컬럼은 **데이터 타입상 인덱스로 지원되지 않아도 저장만 하는 것이므로 문제없고** [102], expression은 지원되지 않습니다 [103]. 또한 **B-Tree, GiST, SP-GiST만 현재 INCLUDE를 지원**하며, (Hash, GIN, BRIN은 지원 안 함) 각 인덱스 구현체마다 포함된 컬럼을 리프에 저장하는 방식으로 동작합니다 [104].

주의사항: 너무 많은 컬럼을 포함하면 인덱스 크기가 커져서 본말전도가 될 수 있습니다 [93]. 특히 **크기가 큰 컬럼(VARCHAR등)**을 넣으면 인덱스 페이지에 저장 가능한 엔트리 수가 줄어들고 **트리 깊이가 증가**하여 오히려 **검색 성능이 떨어질 수 있습니다** [93]. PostgreSQL에서는 B-트리 인덱스 엔트리가 하나의 페이지 크기를 넘을 수 없는데, INCLUDE 컬럼을 넣다가 그 한계를 넘으면 **인서트가 실패**할 수도 있습니다 [93]. 일반적으로는 그런 일이 드물지만, **wide 컬럼**은 되도록 포함하지 않는 것이 좋습니다. 그리고 **테이블이 자주 변경된다면** Index-Only Scan 혜택이 줄어드므로, INCLUDE로 컬럼을 추가하는 보람이 없을 수 있습니다 [105]. 어차피 **테이블 페이지를 매번 들쳐봐야 한다면**, 굳이 인덱스에 그 값을 넣어둘 이유가 없습니다 [103]. 그러므로 **Include 컬럼은 변경이 적은 컬럼**이거나, **해당 테이블이 insert-only나 거의 append-only에 가까운 경우**에 주로 고려합니다 [93] [106].

예제: 아래는 거래 테이블 `trades(id, code, price)`에서 **거래코드(code)**로 검색하고 **가격(price)**을 출력하는 쿼리를 가정한 커버링 인덱스 예시입니다.

```
-- code를 키로 하고 price를 포함하는 커버링 인덱스 생성
CREATE INDEX idx_trades_code_price ON trades(code) INCLUDE(price);

-- 활용 예: code로 조회하면서 price를 읽어오는 쿼리 (Index-Only Scan 가능)
SELECT price
FROM trades
WHERE code = 'ABC123';
```

`idx_trades_code_price` 인덱스는 `code`를 기준으로 트리를 구성하고, 각 엔트리에 `price`값을 함께 저장합니다. `SELECT price FROM trades WHERE code='ABC123'` 쿼리가 실행되면 PostgreSQL은 이 인덱스를 사용해 **code가 'ABC123'인 항목을 찾고**, **직접 그 인덱스 엔트리의 price 값을 반환**합니다 [82] [94]. 만약 해당 페이지가 모두 가시한 상태라면(오래되어 변경 없는 데이터라면) **테이블을 전혀 읽지 않고** 결과를 얻어 성능이 매우 빠릅니다 [95] [97].

또 다른 예로, **복합 인덱스 + INCLUDE**를 응용할 수도 있습니다. 예를 들어 `users(user_id, org_id, name, email)` 테이블에서 `user_id`로 자주 조회하지만 결과로 `name`과 `email`을 함께 보여준다고 합시다. 기본적으로 `user_id` PK인덱스를 타고 찾아온 후 테이블을 읽어 name, email을 가져올 텐데, 이조차도 생략하고 싶다면:

```
-- user_id를 키로 하고 name, email을 포함하는 커버링 인덱스
CREATE INDEX idx_users_cover ON users(user_id) INCLUDE(name, email);

-- 활용 예: user_id로 사용자 찾으면서 이름과 이메일 얻기
SELECT name, email
FROM users
WHERE user_id = 42;
```

이렇게 하면 Index-Only Scan으로 `user_id = 42`인 엔트리를 찾고, **포함된 name과 email을 바로 반환**할 수 있습니다. 물론 이 예시에서는 user_id가 PRIMARY KEY라서 PK 인덱스 자체가 존재하므로, 차라리 PRIMARY KEY를 **INCLUDE(name, email)** 형태로 생성하는 편이 더 나을 것입니다 (PostgreSQL에서는 `PRIMARY KEY(user_id) INCLUDE(name, email)` 구문도 가능합니다 [101]). 단, name이나 email이 자주 바뀌면 Index-Only Scan 혜택이 떨어지므로 이러한 설계가 합리적인지는 따져봐야 합니다.

인덱스 튜닝과 통계 활용

인덱스 성능을 극대화하고 불필요한 인덱스를 줄이려면, **데이터 특성과 쿼리 패턴에 맞는 튜닝과 통계 관리**가 필요합니다. PostgreSQL의 쿼리 플래너는 **통계 정보**에 기반해 **인덱스 사용 여부를 결정**하므로, 올바른 통계가 핵심입니다 [107]. 또한 인덱스와 관련된 몇 가지 **설정 파라미터와 운영 전략**을 알고 있으면 유용합니다.

- **ANALYZE로 통계 갱신:** 데이터가 변경된 후 **반드시 주기적으로** `ANALYZE`를 실행하여 **컬럼 값 분포와 빈도** 정보를 업데이트해야 합니다 [107]. 통계가 오래되었거나 부정확하면 플래너가 **인덱스 사용이 이득인지 오판**할 수 있습니다. 예를 들어 어떤 값이 100만 건 중 5건 뿐인데 통계가 이를 모르면, 플래너는 인덱스 사용 안 하고 풀스캔해버릴 수 있습니다. ANALYZE는 자동으로도 실행되지만, 대량 데이터 적재 후나 데이터 분포가 급변한 후엔 수동 실행이 도움이 됩니다 [107].

- **통계 설정 (n_distinct, 스킵율 등):** PostgreSQL은 각 컬럼에 대해 **고유값 개수(n_distinct)**, **값 분포 히스토그램**, **상관도(correlation)** 등을 통계로 유지합니다. 특히 **상관도**는 **컬럼 값이 테이블 물리 저장 순서와 얼마나 일치하는지**를 나타내며, **인덱스 스캔 비용 평가**에 사용됩니다. 상관도가 높으면 (예: 날짜 순으로 저장) 인덱스

스캔 시 **디스크 접근이 순차적**일 가능성이 높다고 판단하여 인덱스 사용에 유리한 비용 산정이 됩니다. 반대로 상관도가 낮으면 인덱스 스캔이 **랜덤 I/O**로 이어진다고 보고 비용을 높게 책정합니다. 사용자는 `ALTER TABLE ... ALTER COLUMN ... SET STATISTICS N` 명령으로 **통계 수집 샘플 크기**를 조정하여, 특정 컬럼에 대해 **더 정밀한 히스토그램/상관도 통계**를 유지하도록 할 수 있습니다 [108]. 예를 들어 매우 중요한 쿼리 키 컬럼이라면 기본값(100~500 샘플)보다 큰 값으로 설정해 통계 품질을 높이면 플래너의 판단이 정교해집니다.

- **인덱스 사용 임계치:** 일반적으로 플래너는 **질의 결과가 테이블의 5~10% 이하일 때** 인덱스 사용을 고려하고, 그보다 많이 나온다면 시퀀셜 스캔을 선호합니다. 왜냐하면 결과가 너무 많으면 인덱스로 일일이 찾아가는 것보다 차라리 **테이블을 한번 쭉 읽는 편이 빠르기 때문**입니다 [109]. (예: 100만 행 중 500개 찾는 건 인덱스가 유리, 100만 중 300k 찾는 건 차라리 풀스캔+필터가 나을 수 있음.) 따라서 **인덱스가 존재해도 쿼리가 너무 광범위하면** 사용되지 않을 수 있습니다. 이런 경우 인덱스 튜닝보다 쿼리나 데이터 모델을 바꾸는 게 나은 경우가 많습니다.

- **테스트 데이터 vs 실제 데이터:** 인덱스 설계는 반드시 **실제 데이터 분포와 패턴**을 고려해야 합니다 [110]. 소량의 테스트 데이터로는 인덱스 필요성을 판단하기 어렵습니다. 예를 들어 100행 중 1행 찾는 쿼리는 인덱스가 별 소용 없지만, 100만행 중 1000행 찾는 쿼리는 인덱스가 매우 유용할 수 있습니다 [111]. 또, 테스트 데이터가 지나치게 랜덤이거나 한쪽으로 치우치면 통계가 왜곡될 수 있습니다 [112]. 가능하다면 **실서비스에 가까운 분포**로 데이터를 준비하고, 인덱스 실험을 하는 것이 좋습니다 [110].

- **플래너 강제 옵션:** 때로는 플래너가 인덱스를 쓰지 않을 때 **강제로 써보도록** 실험해볼 수 있습니다. `SET enable_seqscan = off;` 등으로 시퀀셜 스캔을 꺼서 **인덱스 경로를 강제로 선택**하게 한 뒤, `EXPLAIN ANALYZE`로 실제 성능을 비교해볼 수 있습니다 [113] [114]. 만약 인덱스 경로가 확실히 빠른데도 플래너가 선택 안 한다면, **통계나 코스트 파라미터에 문제가 있는 것**입니다 [115]. 이때 **random_page_cost** 등 I/O 비용 파라미터나 **cpu_index_tuple_cost** 등을 조정해서 플래너의 비용 산정을 현실에 맞게 조절할 수 있습니다 [108]. 혹은 앞서 말한 **통계 타겟 조정**으로 selectivity 예측을 개선해야 할 수도 있습니다 [108]. 그래도 플래너가 이상하게 행동한다면, 그것은 버그일 가능성도 있으므로 개발자들과 논의해볼 수 있습니다 [116] [117].

- **인덱스 일괄 생성과 유지 보수:** 대량의 인덱스를 한꺼번에 생성하거나 재색인해야 할 땐, **병렬 세션에서 동시에 진행**하여 시간을 단축할 수 있습니다. 그러나 쓰기 부하와 I/O를 감안해야 합니다. 또한 `maintenance_work_mem` 파라미터를 충분히 키워두면 인덱스 생성 시 내부 정렬이나 버퍼링에 유리합니다. `fillfactor` 설정은 B-트리 인덱스에서 **페이지 당 여유 공간**을 지정하는데, **랜덤 키 삽입이 많은 경우** 약간 낮춰(예: 90) 조각화를 완화할 수 있습니다. 반대로 **append 전용** 상황이라면 기본 90에서 높여도 괜찮습니다.

- **인덱스 모니터링:** PostgreSQL은 `pg_stat_user_indexes` 뷰에서 각 인덱스별 **사용된 횟수**와 **효율**을 제공합니다. `idx_scan` 횟수가 현저히 0에 가까운 인덱스는 **실제로 안 쓰이고 있는 인덱스**일 가능성이 높습니다. 이런 인덱스들은 유지비만 들고 효과가 없으므로 **드롭하는 것이 좋습니다** [6]. 다만 꼭 운용 통계만 믿지 말고, 주기적인 배치 작업이나 특정 드문 상황에서 필요한 인덱스일 수도 있으니, 용도를 파악한 후 결정합니다.

- **인덱스와 진공(vacuum):** 인덱스도 테이블처럼 **죽은 튜플 정리**가 필요합니다. Autovacuum 프로세스가 백그라운드에서 인덱스의 쓰레기 엔트리를 제거하고, B-트리의 경우 **페이지 병합** 등을 합니다. 하지만 너무 많은 삭제가 일어나 한꺼번에 인덱스가 부풀어 오른 경우 **REINDEX**로 재구축하는 것이 나을 수 있습니다. 또한 PostgreSQL 13부터 B-트리는 **중복 키에 대한 deduplication**을 자동 수행하여 인덱스 크기를 줄여 주는데, **INCLUDE 인덱스가 있으면 deduplication이 작동하지 않는다**는 제약이 있습니다 [118]. (INCLUDE 포함 인덱스는 상위 노드에 key만 저장하므로 중복 제거 로직이 제한됨.) 이런 세부 사항도 인덱스 튜닝 시에는 참고해야 합니다.

- **운영 중 인덱스 생성:** 앞서 언급했듯, 기본 인덱스 생성은 배타 락을 걸기 때문에, **서비스 중 대용량 테이블에 인덱스를 추가**해야 한다면 `CREATE INDEX CONCURRENTLY`를 사용합니다 [119]. Concurrently 모드는 여러

단계에 나눠 락을 최소화하지만, 일반 모드보다 느리고 실패 시 롤백 처리도 복잡합니다. 상황에 맞게 선택해야 합니다.

- **전문 검색과 인덱스 종류 선택:** 만약 **풀텍스트 검색**처럼 GIN과 GiST 중 어느 것을 쓸지 고민되는 경우, 일반적으로 **GIN이 조회 성능 우수**, **GiST가 업데이트 적은 부하**라는 점을 기억합니다. GIN은 한 번 만들어두면 검색은 빠르나 업데이트는 비싸고, GiST는 검색이 다소 느려도 업데이트는 수월한 편입니다. 데이터와 쿼리 빈도에 따라 맞추어야 합니다.

정리하면, **인덱스는 만들고 끝이 아니라 지속적인 관리 대상**입니다. 적절한 통계 관리와 모니터링을 통해 **유효하지 않은 인덱스를 과감히 정리**하고, 필요한 인덱스는 **최적의 형태**로 유지하는 것이 데이터베이스 성능 튜닝의 중요한 부분입니다.

以上、PostgreSQL의 인덱스 유형별 특성과 고급 최적화 기법을 모두 살펴보았습니다. 인덱스 전략을 수립할 때는 **공식 문서의 권고 사항**과 여기 첨부한 레퍼런스를 참고하시길 바랍니다. 필요하다면 PostgreSQL 공식 문서의 해당 장(Indexes)을 직접 읽어보는 것도 많은 도움이 됩니다 [120] [121]. **올바른 인덱스 활용**으로 PostgreSQL 데이터베이스의 성능을 극대화하시길 바랍니다!

---

[1] [2] [3] [4] [5] [6] [7] PostgreSQL: The World's Most Advanced Open Source Relational Database
https://access.crunchydata.com/documentation/postgresql15/15.13/indexes-intro.html

[8] [107] [108] [109] [110] [111] [112] [113] [114] [115] [116] [117] PostgreSQL: The World's Most Advanced Open Source Relational Database
https://access.crunchydata.com/documentation/postgresql15/15.13/indexes-examine.html

[9] [10] [11] [12] [13] [14] [15] [25] [26] [27] [29] [30] [32] [33] [34] [35] [38] [39] PostgreSQL: Documentation: 17: 11.2. Index Types
https://www.postgresql.org/docs/current/indexes-types.html

[16] [17] [18] [19] [20] [21] [22] [23] [24] PostgreSQL: Documentation: 17: 64.6. Hash Indexes
https://www.postgresql.org/docs/current/hash-index.html

[28] [31] [36] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] PostgreSQL: Documentation: 17: 11.3. Multicolumn Indexes
https://www.postgresql.org/docs/current/indexes-multicolumn.html

[37] [81] [82] [83] [84] [85] [86] [87] [88] [93] [94] [95] [96] [97] [98] [99] [100] [101] [103] [104] [105] [106] PostgreSQL: The World's Most Advanced Open Source Relational Database
https://access.crunchydata.com/documentation/postgresql15/15.13/indexes-index-only-scans.html

[50] How does PostgreSQL combine the use of multiple indexes via AND?
https://stackoverflow.com/questions/77893519/how-does-postgresql-combine-the-use-of-multiple-indexes-via-and

[51] [52] [53] [54] [55] [56] [57] PostgreSQL: The World's Most Advanced Open Source Relational Database
https://access.crunchydata.com/documentation/postgresql15/15.13/indexes-expressional.html

[58] [60] [89] [90] [91] [92] [102] [118] [119] PostgreSQL: The World's Most Advanced Open Source Relational Database
https://access.crunchydata.com/documentation/postgresql15/15.13/sql-createindex.html

[59] [61] [62] [63] [64] [65] [66] [67] [68] [69] [70] [71] [72] [73] [74] [75] [76] [77] [78] [79] [80] PostgreSQL: The World's Most Advanced Open Source Relational Database
https://access.crunchydata.com/documentation/postgresql15/15.13/indexes-partial.html

[120] [121] PostgreSQL: The World's Most Advanced Open Source Relational Database
https://access.crunchydata.com/documentation/postgresql15/15.13/indexes.html