INZENT

IT Player, it's INZENT

PostgreSQL Performance

PostgreSQL
Performance

2024.07

데이터플랫폼개발팀
신주한

Contents

01

INDEX

01 | Index 정의
02 | Index 생성
03 | Index 동작
04 | Practice

02

EXECUTION PLAN

01 | Execution plan 정의
02 | Plan node
03 | Explain
04 | System catalog
05 | Practice

03

PARTITION

01 | Partitioned table 정의
02 | Partition 종류
03 | Partition pruning
04 | Practice

Contents

04
TRANSACTION

01 | Transaction 정의
02 | ACID
03 | 격리 수준
04 | Transaction 제어
05 | Auto-commit
06 | Practice

05
LOCK

01 | Lock 정의
02 | Shared and exclusive
03 | Lock type
04 | Table level lock mode
05 | Row level lock mode
06 | Lock holder and Lock waiter
07 | Deadlock
08 | pg_locks
09 | Practice

06
SESSION

01 | Session 정의
02 | pg_stat_activity
03 | Session kill
04 | Statement cancel
05 | Practice

PostgreSQL Performance

INDEX

1.1. Index 정의

▶ Index

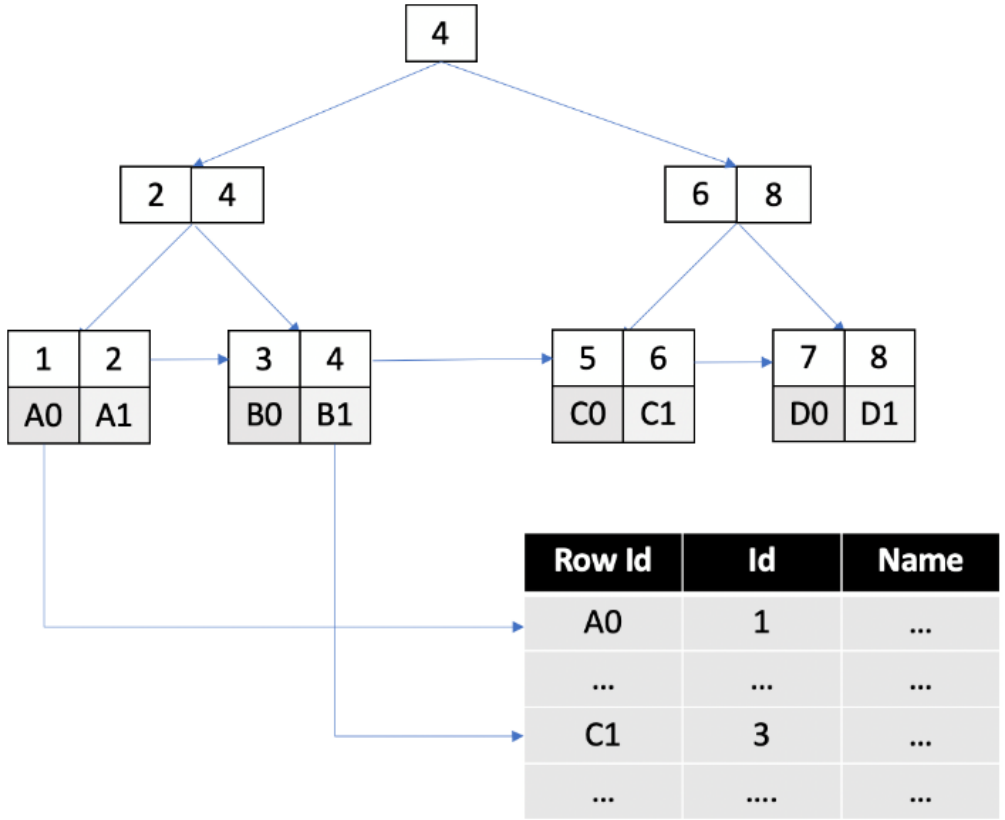
- 테이블의 특정 컬럼에 대한 검색 속도를 향상시키기 위해 생성된 트리(Tree) 구조의 데이터베이스 객체
- 인덱스는 일반적으로 B+트리 구조를 사용하여, 검색, 삽입, 삭제 시 일정한 성능을 보장
- 인덱스의 리프 노드(Leaf node)가 연결되어 있어서 범위 검색 시 효율적
- 인덱스 컬럼 값을 기준으로 정렬을 유지하고 있어서 필요한 범위의 로우에 빠르게 접근 가능
- 인덱스와 테이블은 항상 동기화되어야 하며, 테이블 데이터가 변경될 때마다 인덱스도 갱신됨

▶ 장점

- 특정 조건에 맞는 로우들을 빠르게 검색할 수 있음
- 정렬되어있기 때문에 범위 조건이나 정렬이 필요한 경우 성능이 향상됨

▶ 단점

- 인덱스를 유지하기 위한 저장 공간 증가
- DML을 수행할 때마다 인덱스도 수정해야 하므로 오버헤드가 발생



1.2. Index 생성

INZENT

▶ CREATE [UNIQUE] INDEX

- 특정 컬럼이나 컬럼 조합에 대한 인덱스를 생성
- UNIQUE 키워드를 사용하면 고유 인덱스(Unique index)를 생성하여 값이 중복되지 않도록 보장

▶ Example

```
-- 일반 인덱스 생성
CREATE INDEX idx_emp_email ON employee(email);

-- 고유 인덱스 생성
CREATE UNIQUE INDEX idx_emp_email ON employee(email);
```

1.2. Index 생성

INZENT

▶ PRIMARY KEY constraint

- 컬럼을 PRIMARY KEY로 지정하면 해당 컬럼에 대한 UNIQUE INDEX가 자동으로 생성
- PRIMARY KEY는 NULL 값을 허용하지 않으며, 테이블에서 고유한 값을 가짐

▶ Example

```
CREATE TABLE employee (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    dept_id INT
);
```

1.2. Index 생성

INZENT

▶ UNIQUE constraint

- 컬럼에 UNIQUE constraint를 추가하면 해당 컬럼에 대한 UNIQUE INDEX가 자동으로 생성
- UNIQUE constraint는 해당 컬럼의 값이 중복되지 않도록 보장

▶ Example

```
CREATE TABLE employee (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100) UNIQUE
);
```

1.2. Index 생성

INZENT

▶ Functional index

- 특정 컬럼의 값을 변환하거나 함수의 결과를 기반으로 인덱스를 생성
- 함수 또는 표현식을 사용하여 인덱스가 필요한 경우 유용

▶ Syntax

```
CREATE INDEX index_name ON table_name (function(column_name));
```

▶ Example

```
-- 이메일 주소를 소문자로 변환한 결과를 기반으로 인덱스 생성
CREATE INDEX idx_emp_lower_email ON employee (LOWER(email));

-- 날짜의 연도를 기반으로 인덱스 생성
CREATE INDEX idx_emp_year ON employee (EXTRACT(YEAR FROM hire_date));
```

1.2. Index 생성

INZENT

▶ Partial index

- 특정 조건을 만족하는 로우에만 인덱스를 생성하여 성능을 최적화
- 조건에 맞는 데이터에 대해서만 인덱스를 생성하여 인덱스 크기를 줄이고, 검색 성능을 향상시킴

▶ Syntax

```
CREATE INDEX index_name ON table_name (column_name) WHERE condition;
```

▶ Example

```
CREATE INDEX idx_salary_emp ON employee (salary) WHERE salary > 10000;
```

1.2. Index 생성

▶ 인덱스 조회

- PSQL에서 ₩d 명령어로 조회

```
₩d table_name
```

- SQL에서 조회

```
SELECT tablename, indexname
FROM pg_indexes
WHERE tablename = table_name;
```

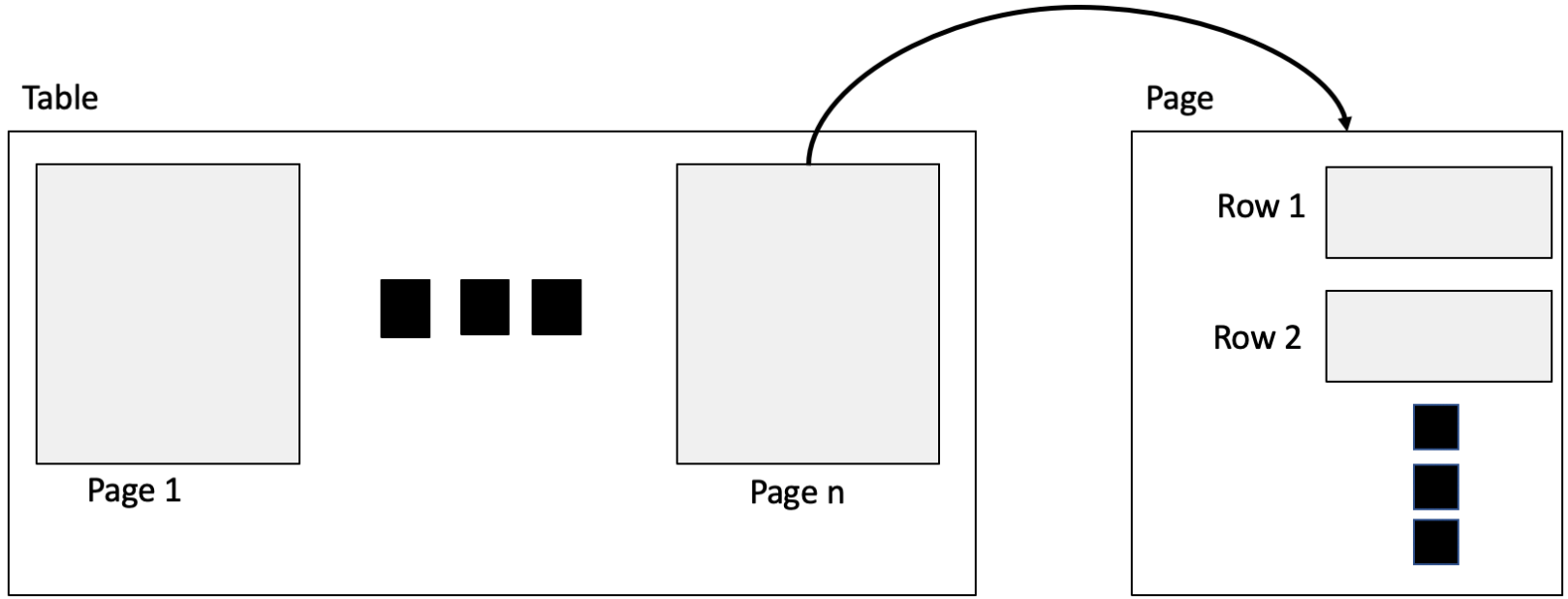
1.3. Index 동작

▶ Sequential scan

- Sequential scan은 테이블의 모든 로우를 처음부터 끝까지 순차적으로 읽는 방식
- 인덱스가 없거나, 테이블의 대부분 또는 모든 로우를 읽어야 할 때 사용

▶ Example

```
SELECT *
FROM employee;
```



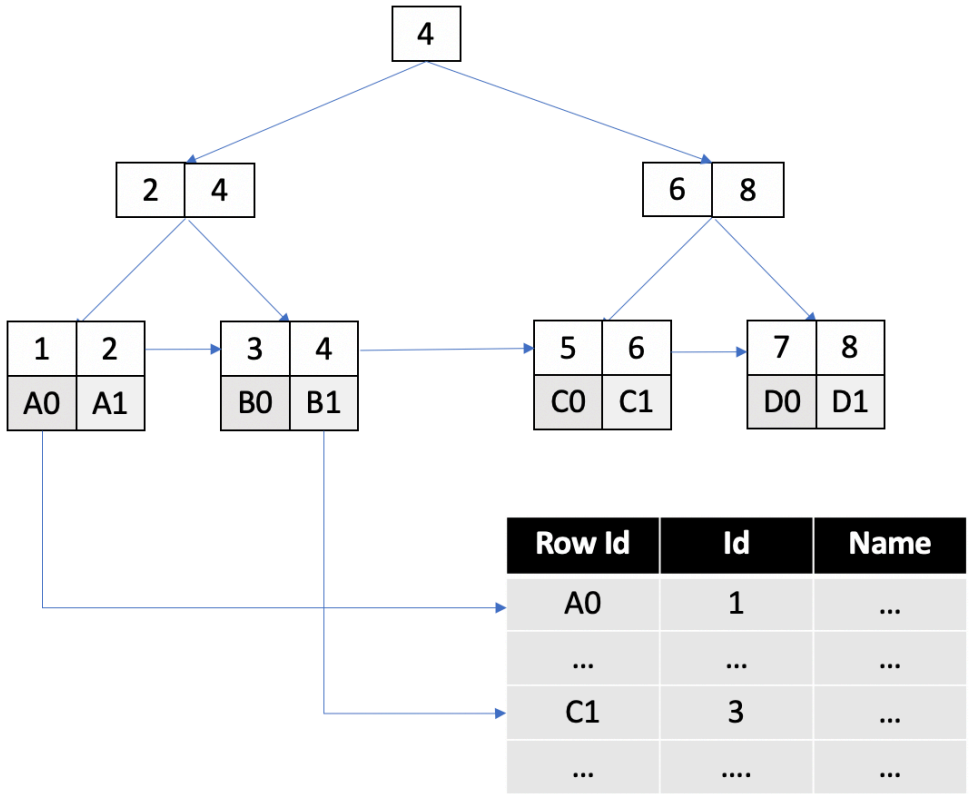
1.3. Index 동작

▶ Index scan

- 인덱스를 사용하여 특정 조건을 만족하는 로우를 빠르게 찾는 방식
- 인덱스를 사용하여 특정 조건을 만족하는 행을 빠르게 찾는 방식
- 정렬된 데이터가 필요한 경우에도 사용됨

▶ Example

```
SELECT id, name
FROM employee
WHERE id = 1 OR id = 3;
```



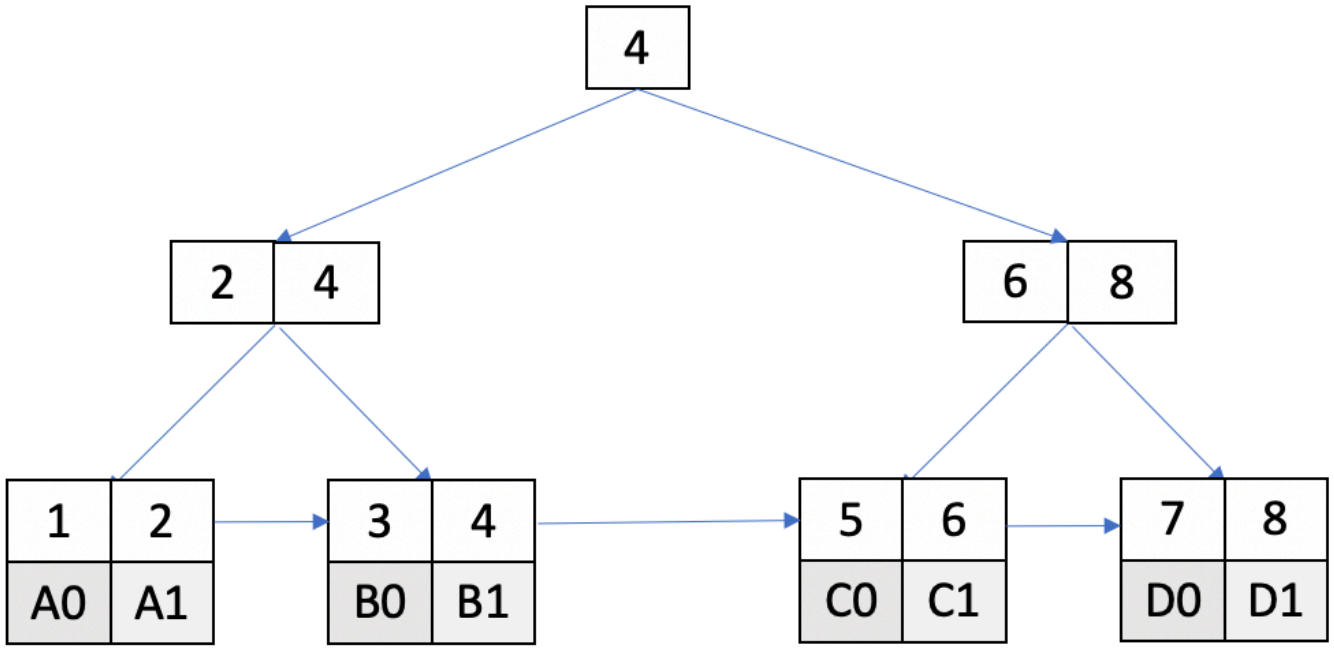
1.3. Index 동작

▶ Index only scan

- 필요한 모든 데이터가 인덱스에 포함되어 있어, 테이블 자체를 접근할 필요 없이 인덱스만으로 쿼리를 처리하는 방식
- 쿼리가 필요한 모든 컬럼이 인덱스에 포함되어 있고, 인덱스의 최신 통계가 유지되고 있을 때 사용

▶ Example

```
SELECT id
FROM employee
WHERE id = 1 OR id = 3;
```



1.3. Index 동작

▶ 카디널리티 (Cardinality)

- 컬럼의 고유한 값의 개수
  - 카디널리티가 높은 컬럼 : 사용자 아이디, 주민등록 번호 등
  - 카디널리티가 낮은 컬럼 : 성별, 혈액형 등
- 카디널리티가 낮은 컬럼에 인덱스를 생성하는 것은 비효율적인 경우가 많음

▶ **INSERT > CREATE INDEX vs CREATE INDEX > INSERT**

PostgreSQL Performance

02

EXECUTION PLAN

2.1. Execution plan 정의

INZENT

▶ Execution plan

- Cost를 기반으로 SQL을 수행하는 데 가장 적합한 방식을 비교하여 결정된 최적의 경로
- 동일한 SQL 문이라도 수집된 통계 정보나 데이터베이스 객체 정보에 따라 cost가 달라질 수 있으므로 execution plan도 달라질 수 있음
- 대부분의 데이터베이스는 최적의 수행속도를 위해 SQL이 전달되면 execution plan을 생성
- Query plan이라고도 부름

2.2. Plan node

INZENT

▶ Plan node

- Execution plan은 plan node들로 이루어진 Tree 구조로 구성됨
- 각 plan node는 특정 작업(예: 테이블 스캔, 조인, 정렬 등)을 수행하며, 이 작업의 비용(cost)과 예상 결과물의 크기(rows)를 포함한 메타 정보를 가지고 있음
- Plan node는 부모-자식 관계를 가지며, 자식 노드의 결과를 부모 노드가 받아 처리하는 방식으로 동작함
- 최종 결과를 반환하는 루트 노드는 전체 쿼리 실행의 최종 단계를 나타냄

▶ Scan

- Sequential scan
- Index scan
- Index only scan
- Bitmap index scan
- Bitmap heap scan

▶ Join

- Nested loop join
- Hash join
- Merge Join

▶ Sort

- Sort

▶ Aggregate

- Aggregate
- Window Function

▶ Set Operation

- SetOp

2.2. Plan node

▶ Sequential scan

- 테이블의 모든 로우를 처음부터 끝까지 순차적으로 읽는 방식
- Index가 없는 경우나 소량의 데이터를 처리할 때 효율적

▶ Index scan

- Index를 사용하여 특정 조건에 맞는 로우를 검색하는 방식
- Index를 통해 필요한 로우에 빠르게 접근할 수 있어, 범위와 같은 조건 검색이 효율적
- 적은 수의 로우를 필요로 할 때 효과적

▶ Bitmap index scan

- Index를 이용해 필요한 로우의 물리적인 위치를 정렬해서 비트맵(bitmap)을 생성
- 비트맵의 각 bit는 페이지에 필요한 로우가 존재하는지를 표시

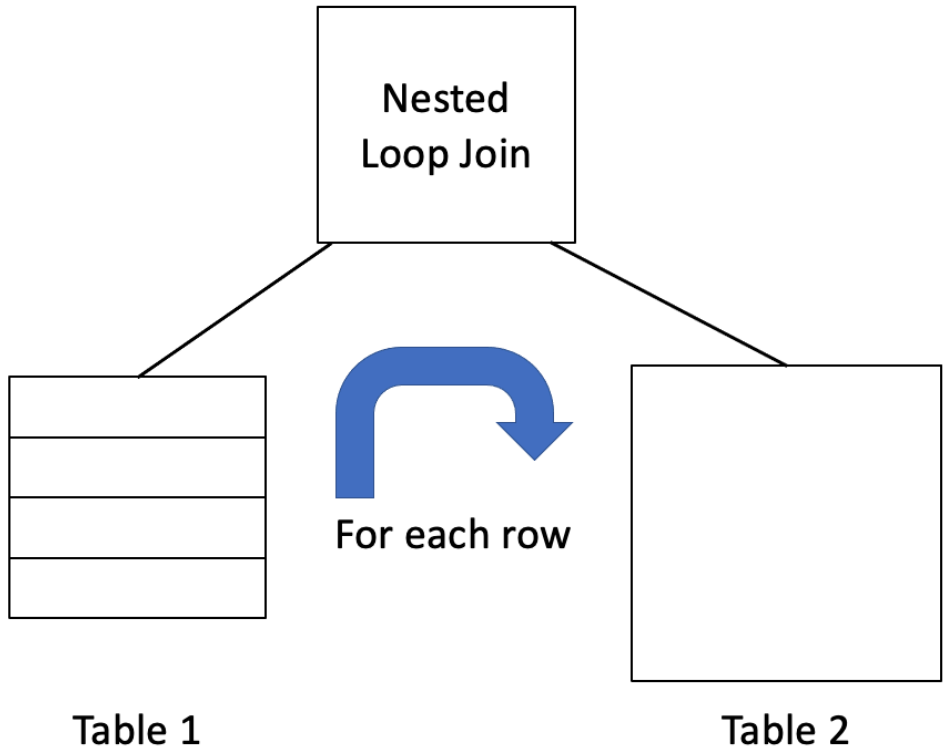
▶ Bitmap heap scan

- Bitmap index scan으로부터 받은 비트맵을 이용해 필요한 페이지만 스캔

2.2. Plan node

▶ Nested loop join

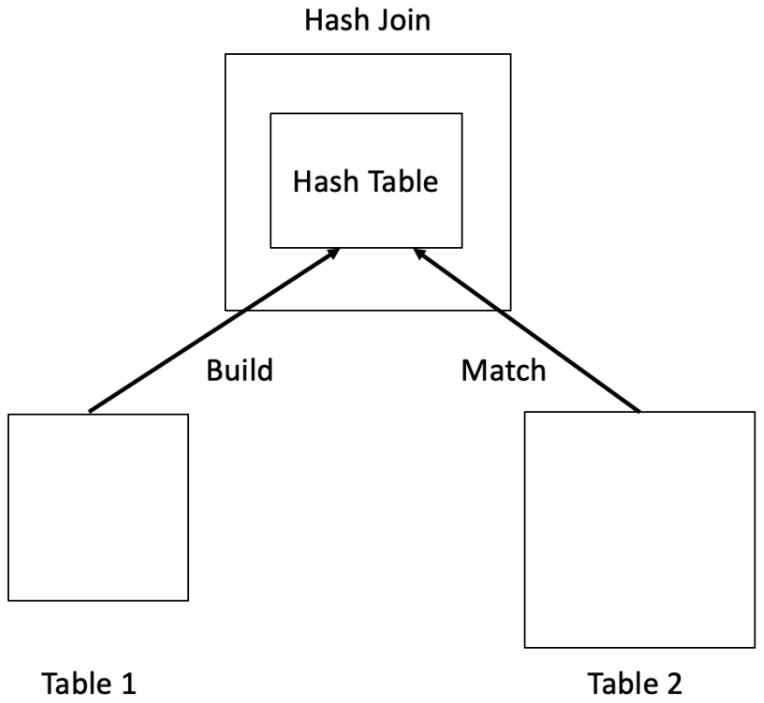
- 한 테이블의 각 로우 마다 다른 테이블을 반복적으로 검색
- 일반적으로 작은 테이블을 드라이빙 테이블로 선택하여 처리하는데 적합
- 시간 복잡도 : O(n * m)



2.2. Plan node

▶ Hash join

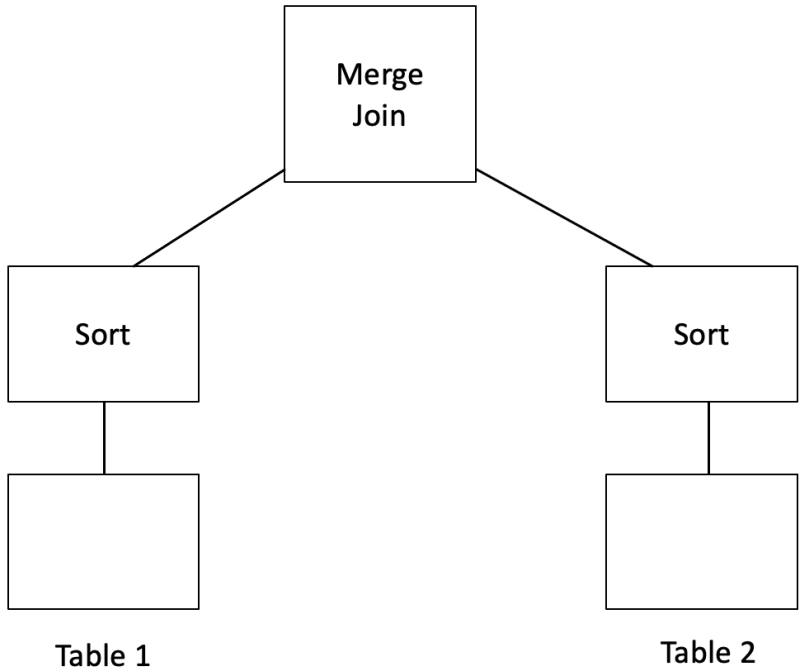
- 해시 테이블(Hash table)을 생성하여 조인하는 방식
- 해시 테이블을 사용하여 빠르게 매칭할 수 있어, 큰 테이블을 조인할 때에도 효율적임
- 하나의 테이블을 조회해서 로우들로 해시 테이블을 만들고, 다른 테이블의 로우마다 해시 테이블을 검색해서 Join 결과를 생성
- 일반적으로 작은 테이블로 해시를 만드는 것이 효과적임
- 시간 복잡도 : O(n + m)



2.2. Plan node

▶ Merge join

- 두 테이블을 정렬하여 병합하는 방식
- 양쪽 테이블이 이미 정렬되어 있거나, 정렬이 빠른 경우에 효과적
- 즉, 조인 컬럼에 Index가 있는 경우 효과적
- 시간 복잡도
  - 정렬이 필요한 경우 : O(nlogn + mlogm)
  - 정렬이 되어있는 경우 : O(n + m)



2.2. Plan node

▶ Sort

- 데이터를 정렬하는 노드
- ORDER BY 절이나 Merge Join에서 필요할 때 사용됨

▶ Aggregate

- 데이터를 그룹화하여 집계 함수(SUM, COUNT, AVG 등)를 적용하는 노드
- GROUP BY 절과 함께 사용됨

▶ Window function

- 윈도우 함수(ROW_NUMBER, RANK, PARTITION BY 등)를 적용하는 노드
- 데이터의 특정 범위나 그룹 내에서 계산을 수행함

▶ SetOp

- 집합 연산(UNION, INTERSECT, EXCEPT 등)을 수행하는 노드
- 두 개 이상의 결과 집합을 결합하거나 비교할 때 사용됨

2.3. Explain

▶ Explain

- 수집된 통계 정보(Statistics)를 기반으로 SQL의 execution plan을 출력해주는 명령어
- SQL을 실제로 수행하지 않음
- 지금까지 수집된 통계 정보를 기반으로 해당 SQL을 수행했을 경우 예상되는 execution plan과 근거가 되는 Cost, Rows, Width를 표시

▶ Syntax

```
EXPLAIN SQL_statement;
```

▶ Example

```
EXPLAIN SELECT * FROM employee;
```

2.3. Explain

▶ Cost

- 각 plan node를 수행하는 작업에 대한 비용을 나타냄
  - Cost는 다음 요소들을 적용한 공식으로 산출함
    - 디스크 I/O
    - CPU usage
    - Memory usage
- Startup cost
  - 해당 플랜 노드에서 첫 번째 로우를 얻기 전까지 소요된 비용
- Running cost
  - 해당 플랜 노드에서 모든 로우를 처리하기 위해 소요된 비용
- Total cost
  - Startup cost + Running cost

▶ Rows

- Plan node에서 상위 plan node로 전달한 로우의 수

▶ Width

- 로우의 평균 길이
- 바이트(Byte) 단위이므로 Rows와 Width를 곱하면 이동한 데이터의 양을 유추할 수 있음

2.3. Explain

INZENT

▶ EXPLAIN ANALYZE

- Planner(Optimizer)의 정확도를 확인하기 위해 SQL을 실행한 후, 실제로 소요된 Cost, Rows, Width를 함께 표시
- EXPLAIN은 수집된 통계 정보를 기반으로 예측된 실행 계획을 보여주지만, EXPLAIN ANALYZE는 실제 실행된 결과를 보여줌
- SELECT 문의 경우 조회만 하기 때문에 문제가 없지만, DML의 경우 실제로 수행을 해버리기 때문에 주의가 필요

2.4. System catalog

INZENT

▶ System catalog

- 데이터베이스의 메타데이터를 조회 및 관리하기 위해 생성된 특별한 테이블

▶ System catalog 조회

```
₩dt pg_catalog.*

SELECT schemaname, tablename
FROM pg_catalog.pg_tables
WHERE schemaname = 'pg_catalog'
ORDER BY tablename;
```

▶ pg_statistic

- Execution plan을 작성하기 위해 데이터베이스 객체의 통계 정보를 저장해놓은 테이블
- 각 테이블의 컬럼에 대한 통계 정보 (예: 최소값, 최대값, NULL 값의 개수, 컬럼 값의 분포 등)를 포함
- 쿼리 최적화를 위해 PostgreSQL 내부에서 사용

2.4. System catalog

INZENT

▶ System view

- 시스템 카탈로그 테이블의 정보를 보다 사용자 친화적인 방식으로 제공하는 뷰
- 시스템 뷰는 데이터베이스 관리자가 쉽게 접근할 수 있도록 설계되었으며, 시스템 카탈로그 테이블의 복잡한 정보를 간편하게 조회할 수 있음

▶ System view 조회

```
SELECT viewname FROM pg_views ORDER BY viewname;
```

▶ pg_stats

- pg_statistic 테이블의 통계 정보를 읽기 쉽게 제공하는 시스템 뷰
- 테이블과 인덱스 컬럼의 통계 정보를 조회할 수 있으며, 데이터베이스 관리자가 쿼리 성능을 튜닝하고, 통계 정보를 분석하는 데 유용

2.5. Practice

▶ **customer 테이블에 아래의 SQL문을 수행시킬 때, index scan을 사용할 수 있도록 인덱스를 생성하는 SQL을 작성하세요.**

```
SELECT * FROM customer where email = '4000@example.com';
```

PostgreSQL Performance

PARTITION

3.1. Partitioned table 정의

INZENT

▶ 파티션 테이블 (Partitioned table)

- 테이블을 더 작은 여러 개의 파티션으로 분할하여 데이터를 저장하는 구조
- 파티션 프루닝을 통해 불필요한 스캔을 줄일 수 있음
- 데이터는 각 파티션에 로우 단위로 저장되며, 파티션 키 컬럼을 기준으로 적절한 파티션에 분할됨

▶ 파티션 (Partition)

- 테이블의 데이터를 분산 저장하는 단위
- 각 파티션은 논리적으로 독립된 데이터 저장 단위로 관리됨 (물리적으로 독립된 저장공간을 가질 수도 있고, 그렇지 않을 수도 있음)
- 각 파티션은 기본 테이블과 동일한 구조를 가짐

▶ 파티션 키 컬럼(Partition key column)

- 테이블의 데이터를 분할하기 위한 기준이 되는 컬럼

3.2. Partition 종류

INZENT

▶ List Partition

- 파티션 키 컬럼 값을 정의된 목록에 따라 구분하여 데이터를 각 파티션에 분산 저장

▶ Syntax

```
CREATE TABLE table_name (
    column1 data_type,
    column2 data_type,
    ...
) PARTITION BY LIST (partition_key_column);

CREATE TABLE partition_name1 PARTITION OF table_name
FOR VALUES IN (value1, value2, ...);
CREATE TABLE partition_name2 PARTITION OF table_name
FOR VALUES IN (value3, value4, ...);
```

▶ Example

```
CREATE TABLE employee (
    id SERIAL,
    name VARCHAR(100),
    department VARCHAR(50)
) PARTITION BY LIST (department);

CREATE TABLE emp_sales PARTITION OF employee
FOR VALUES IN ('Sales');

CREATE TABLE emp_hr PARTITION OF employee
FOR VALUES IN ('HR');
```

3.2. Partition 종류

INZENT

▶ Range Partition

- 파티션 키 컬럼 값의 범위를 기준으로 구분하여 데이터를 각 파티션에 분산 저장
- 범위는 start_value >= x < end_value

▶ Syntax

```
CREATE TABLE table_name (
    column1 data_type,
    column2 data_type,
    ...
) PARTITION BY RANGE (partition_key_column);

CREATE TABLE partition_name PARTITION OF table_name
FOR VALUES FROM (start_value) TO (end_value);
```

▶ Example

```
CREATE TABLE sales (
    id SERIAL,
    sale_date DATE,
    amount NUMERIC
) PARTITION BY RANGE (sale_date);

CREATE TABLE sales_2020 PARTITION OF sales
FOR VALUES FROM ('2020-01-01') TO ('2021-01-01');

CREATE TABLE sales_2021 PARTITION OF sales
FOR VALUES FROM ('2021-01-01') TO ('2022-01-01');
```

3.2. Partition 종류

▶ Hash Partition

- 파티션 키 컬럼을 해시 값(Hash value)으로 변환한 후, 모듈러스(Modulus) 연산을 통해 데이터를 각 파티션에 분산 저장

▶ Hash Partition의 장점

- Data를 균등하게 분할하여 병렬처리에 용이함
- Non-partitioned table에 대해서도 병렬처리(Parallel execution)이 가능하지만 테이블의 분할이 불균등할 수 있음

▶ Syntax

```
CREATE TABLE table_name (
    column1 data_type,
    column2 data_type,
    ...
) PARTITION BY HASH (partition_key_column);

CREATE TABLE partition_name PARTITION OF table_name
FOR VALUES WITH (modulus n, remainder r);
```

▶ Example

```
CREATE TABLE users (
    id SERIAL,
    username VARCHAR(50),
    email VARCHAR(100)
) PARTITION BY HASH (id);

CREATE TABLE users_p0 PARTITION OF users
FOR VALUES WITH (modulus 4, remainder 0);

CREATE TABLE users_p1 PARTITION OF users
FOR VALUES WITH (modulus 4, remainder 1);

CREATE TABLE users_p2 PARTITION OF users
FOR VALUES WITH (modulus 4, remainder 2);

CREATE TABLE users_p3 PARTITION OF users
FOR VALUES WITH (modulus 4, remainder 3);
```

3.2. Partition 종류

INZENT

▶ 파티션 정보 조회

PSQL에서 ₩d 명령어로 조회

```
₩d table_name
```

▶ SQL로 조회

pg_class

테이블 및 인덱스에 대한 정보를 저장하는 시스템 카탈로그

pg_inherits

테이블 상속 관계를 저장하는 시스템 카탈로그

파티션 관계를 저장

▶ Syntax

```
SELECT
    child.relname AS partition_name,
    parent.relname AS parent_table
FROM
    pg_inherits
JOIN
    pg_class child ON pg_inherits.inhrelid = child.oid
JOIN
    pg_class parent ON pg_inherits.inhparent = parent.oid
WHERE
    parent.relname = 'your_table_name';
```

3.2. Partition 종류

INZENT

▶ Expression-based partitioning

- 파티션 키 컬럼으로 표현식(함수 결과)을 사용할 수 있음

▶ Syntax

```
CREATE TABLE table_name (
    column1 data_type,
    column2 data_type,
    ...
) PARTITION BY { RANGE | LIST | HASH } (expression);

CREATE TABLE partition_name PARTITION OF table_name
FOR VALUES { FROM (value1) TO (value2) | IN (value1, value2, ...) | WITH
(modulus n, remainder r) };
```

▶ Example

```
CREATE TABLE sales (
    id SERIAL,
    sale_date DATE,
    amount NUMERIC
) PARTITION BY RANGE (EXTRACT(YEAR FROM sale_date));

CREATE TABLE sales_2023 PARTITION OF sales
FOR VALUES FROM (2023) TO (2024);

CREATE TABLE sales_2024 PARTITION OF sales
FOR VALUES FROM (2024) TO (2025);
```

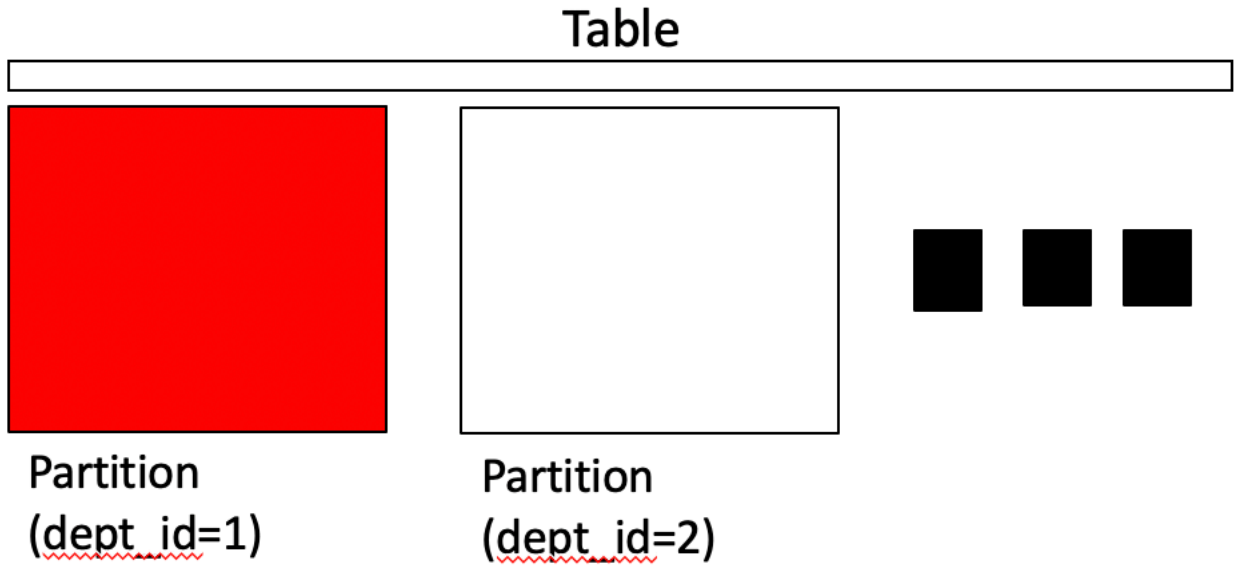
3.3. Partition pruning

INZENT

▶ Partitioned pruning

- SQL 조건에 따라 테이블 전체가 아닌 필요한 파티션에 대해서만 데이터를 처리할 수 있기 때문에 불필요한 스캔을 감소시킴

```
SELECT *
FROM employee
WHERE dept_id=1;
```



3.4. Practice

▶ **날짜 컬럼을 파티션 키 컬럼으로 사용하여 월 단위의 파티션을 생성하는 SQL문을 작성하세요. (extract())**

PostgreSQL Performance

TRANSACTION

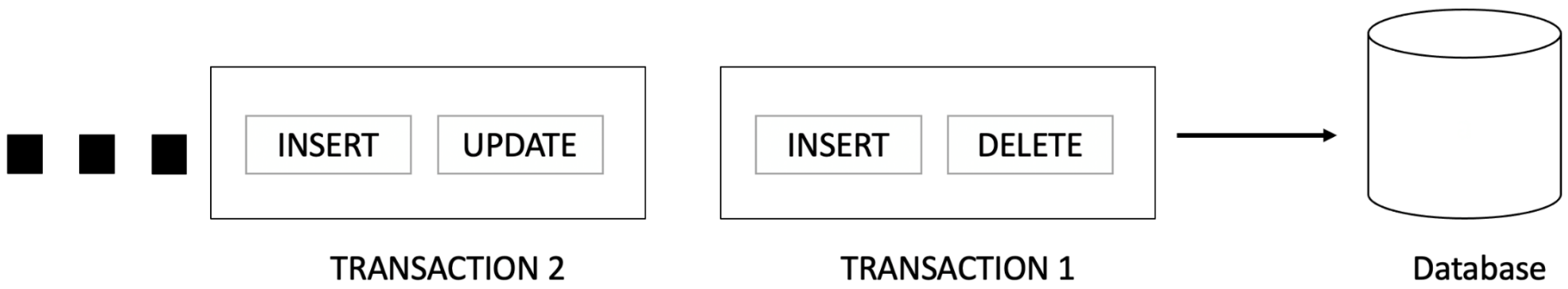
4.1. Transaction 정의

▶ Transaction

- 데이터베이스의 상태를 변화시키기 위해 수행되는 일련의 작업들을 하나로 묶은 논리적인 작업 단위
- 데이터베이스는 트랜잭션 단위로 변경사항이 반영됨

▶ Transaction의 특징

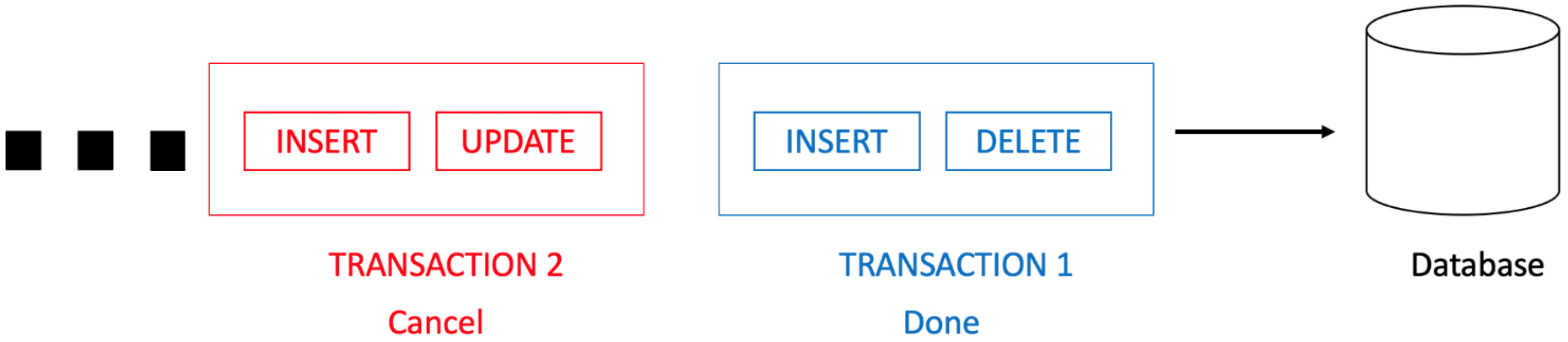
- 원자성(Atomic)
- 일관성(Consistency)
- 격리성(Isolation)
- 영속성(Durability)



4.2. ACID

▶ 원자성(Atomic)

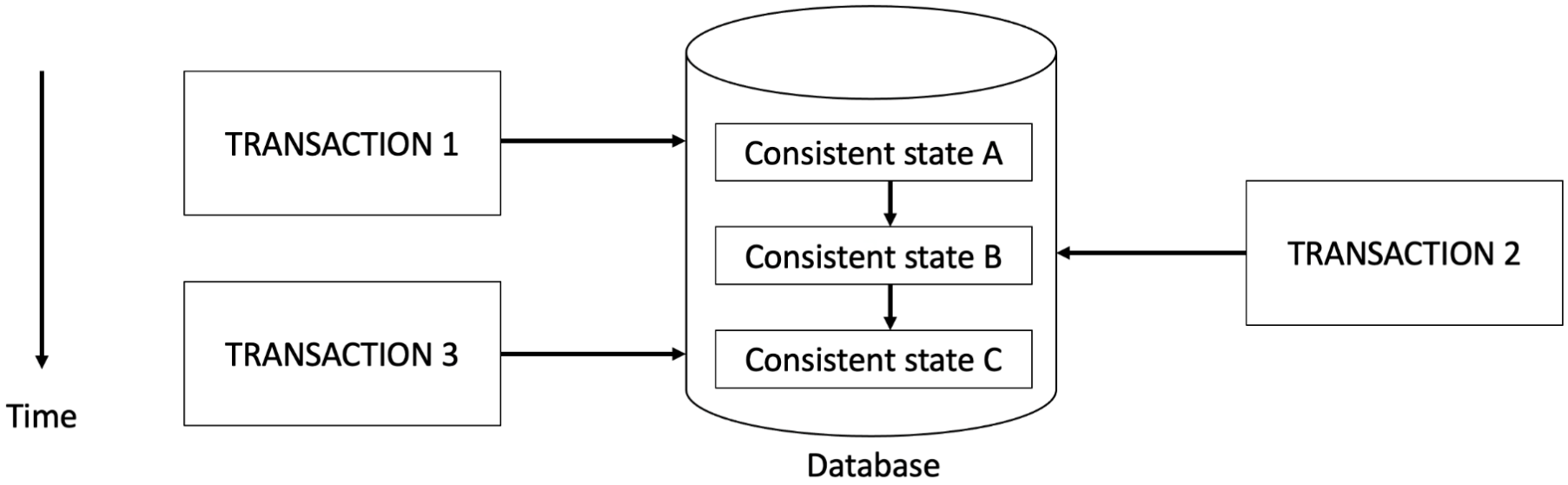
- 트랜잭션의 모든 작업이 반영되거나 모든 작업이 반영되지 않아야 함
- 트랜잭션 반영이 실패할 경우 데이터베이스는 해당 트랜잭션을 수행하기 전 상태로 돌아가야 함



4.2. ACID

▶ 일관성(Consistency)

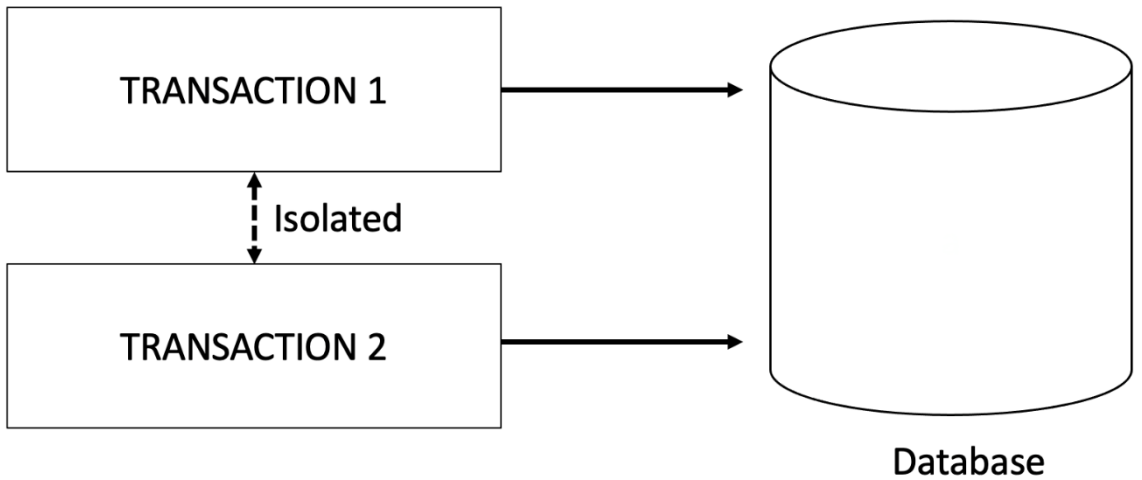
- 트랜잭션이 실행되기 전과 후에 데이터베이스는 항상 일관된 상태를 유지해야 함
- 데이터베이스 내부에 정의된 모든 규칙을 준수해야 함
  - ex) Constraints, Table-index relation, Table-view relation
- 하나의 스냅샷에서는 항상 동일한 상태의 데이터를 조회할 수 있어야 함



4.2. ACID

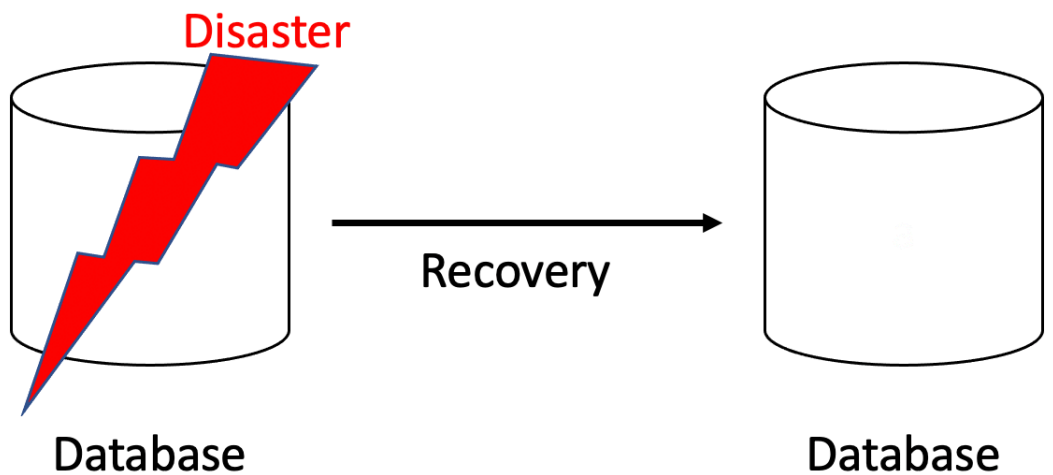
▶ 격리성(Isolation)

- 트랜잭션들이 동시에 수행될 때, 하나의 트랜잭션은 다른 트랜잭션에 영향을 주면 안됨
- 완벽하게 영향을 주지 않는 것이 아니라, 격리 수준(Isolation level)에 따라 영향을 줄 수 있는 정도가 달라짐



▶ 영속성(Durability)

- 트랜잭션이 완료되면 그 변경사항은 영구적으로 저장되어야 함
- 시스템 장애 등이 발생하더라도 트랜잭션에 의해 완료된 변경사항은 변할 수 없음



4.3. 격리 수준

INZENT

▶ READ UNCOMMITTED

- 다른 트랜잭션에서 완료되지 않은 중간 상태의 변경사항을 읽을 수 있음
- Dirty Reads 발생
  - 트랜잭션이 아직 완료되지 않은 데이터를 다른 트랜잭션이 읽을 수 있는 상황
  - 한 트랜잭션이 데이터를 수정했지만 아직 커밋되지 않은 상태에서 다른 트랜잭션이 그 데이터를 읽어 들이는 경우
- PostgreSQL은 지원하지 않음

▶ Dirty read

Time	Transaction1	Transaction2
0	BEGIN	BEGIN
1		UPDATE t1 SET c1 = 100;
2	SELECT * FROM t1;	
3		COMMIT;
4		
5		

4.3. 격리 수준

INZENT

▶ READ COMMITTED

- 다른 트랜잭션에서 완료되지 완료한 변경사항은 읽을 수 있음
- Non-Repeatable Reads 발생
  - 트랜잭션이 같은 데이터를 두 번 읽을 때 그 사이에 다른 트랜잭션이 데이터를 수정하여 두 번째 읽기에서 다른 값을 얻는 상황
- PostgreSQL의 기본 격리 수준

▶ Non-repeatable read

Time	Transaction1	Transaction2
0	BEGIN	BEGIN
1	SELECT * FROM t1;	
2		UPDATE t1 SET c1 = 100; COMMIT;
3	SELECT * FROM t1;	
4		
5		

4.3. 격리 수준

▶ REPEATABLE READ

- 트랜잭션 수행 중에 한 번 읽은 데이터는 계속 같은 값을 유지
- 트랜잭션 시작 시점의 값으로 일관성을 유지
- Phantom read 발생
  - 트랜잭션이 같은 쿼리를 두 번 실행할 때 그 사이에 다른 트랜잭션이 새로운 로우를 삽입해서 첫 번째 쿼리와 두 번째 쿼리의 결과가 달라지는 상황
  - PostgreSQL에서는 발생하지 않음
- Write skew 발생
  - 트랜잭션 내에서 읽은 데이터를 변경하려고 시도할 때, 이미 다른 트랜잭션에 의해 변경된 상태
  - 두 트랜잭션이 같은 조건을 만족하는 로우를 수정하려 할 때 발생할 수 있음

▶ Write skew

Time	Transaction1	Transaction2
0	BEGIN	BEGIN
1	UPDATE t1 SET c1 = 100 WHERE c2 = 1;	UPDATE t1 SET c1 = 100 WHERE c2 = 1; COMMIT;
2		
3		
4		
5		

4.3. 격리 수준

▶ SERIALIZABLE

- 모든 트랜잭션이 순차적으로 실행되는 것처럼 동작함
- 데이터베이스 동시성이 떨어지기 때문에 성능이 느려짐
- 가장 높은 격리 수준

▶ SERIALIZABLE

Time	Transaction1	Transaction2
0	BEGIN	
1	UPDATE t1 SET c1 = 100;	
2	COMMIT;	
3		BEGIN
4		UPDATE t1 SET c1 = 100;
5		COMMIT;

4.4. Transaction 제어

▶ BEGIN

- 트랜잭션의 시작을 알림
- 격리 수준을 함께 명시할 수 있음

▶ Syntax

```
BEGIN;
BEGIN ISOLATION LEVEL READ COMMITTED;
BEGIN ISOLATION LEVEL REPEATABLE READ;
BEGIN ISOLATION LEVEL SERIALIZABLE;
```

4.4. Transaction 제어

INZENT

▶ Commit

- 트랜잭션의 변경사항을 데이터베이스에 영구적으로 저장

▶ Syntax

```
COMMIT;
```

▶ Example

```
BEGIN;
INSERT INTO t1(c1, c2) VALUES (1, 1);
COMMIT;
```

4.4. Transaction 제어

INZENT

▶ Rollback

- 트랜잭션의 변경사항을 트랜잭션 시작 이전 시점으로 복구시킴

▶ Syntax

```
ROLLBACK;
```

▶ Example

```
BEGIN;
INSERT INTO t1(c1, c2) VALUES (1, 1);
ROLLBACK;
```

4.5. Auto-commit

INZENT

▶ Auto-commit

- SQL 문이 수행될 때마다 각각의 문이 자동으로 트랜잭션으로 처리되어 변경 사항이 데이터베이스에 즉시 반영되는 기능
- 이전까지 BEGIN을 사용하지 않아도 변경 사항이 데이터베이스에 반영된 것은 psql을 비롯한 대부분의 SQL 클라이언트 툴에서 auto-commit 옵션이 기본적으로 활성화되어 있었기 때문임

▶ 장점

- 사용자가 명시적으로 트랜잭션을 관리할 필요가 없어 편리함
- 실수로 커밋하지 않아 데이터가 저장되지 않는 상황을 방지할 수 있음

▶ 단점

- Commit은 상대적으로 시간이 오래 걸리는 작업이기 때문에 속도가 오래 걸림
- 여러 SQL 문을 하나의 트랜잭션으로 묶어야 할 경우 수동으로 트랜잭션을 관리해야 함

4.6. Practice

▶ Practice

-
-
-

PostgreSQL Performance

LOCK

5.1. Lock 정의

▶ Lock

- 여러 트랜잭션이 동시에 동일한 데이터에 접근할 때 발생하는 충돌을 방지하기 위한 메커니즘
  Lock 대상은 데이터베이스 객체, 페이지, 로우 등 여러가지가 있음

5.2. Shared and exclusive

INZENT

▶ Shared lock

- 여러 트랜잭션이 동시에 자원을 읽을 수 있게 허용하는 Lock
- 데이터 읽기 작업을 수행할 때 사용되며, 다른 트랜잭션의 읽기 접근도 허용

▶ Exclusive lock

- 특정 트랜잭션만이 자원에 접근할 수 있도록 배타적인 액세스를 제공하는 Lock
- 데이터 수정 작업을 수행할 때 사용되며, 다른 트랜잭션의 읽기와 쓰기 접근을 모두 차단

5.3. Lock type

▶ Table level lock

- 데이터베이스 테이블 전체에 적용되는 Lock
- 테이블 구조 변경이나 전체 테이블에 영향을 미치는 작업에 사용
- 여러 트랜잭션의 동시 접근을 제한

▶ Row level lock

- 테이블 내 특정 로우(row)에만 적용되는 Lock
- 여러 트랜잭션이 같은 테이블의 다른 로우를 독립적으로 수정 가능
- 데이터의 세밀한 제어와 동시성을 높임

5.4. Table level lock mode

▶ ACCESS SHARE

- ACCESS EXCLUSIVE Lock 과 충돌하고 나머지 Lock과는 충돌하지 않음
- SELECT 명령으로 대상 테이블에서 해당 Lock을 획득
- 테이블을 읽기만 하고 수정, 삭제 등을 하지 않는 쿼리에서 해당 Lock을 획득

▶ ROW SHARE

- EXCLUSIVE, ACCESS EXCLUSIVE Lock과 충돌
- SELECT FOR UPDATE, SELECT FOR SHARE 명령으로 대상 테이블에서 해당 Lock을 획득

▶ ROW EXCLUSIVE

- SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE Lock과 충돌
- UPDATE, DELETE, INSERT 명령으로 대상 테이블에서 해당 Lock을 획득(테이블의 데이터를 수정하는 모든 명령어에서 획득)

▶ SHARE UPDATE EXCLUSIVE

- SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE Lock과 충돌
- 해당 모드는 스키마 동시 변경 또는 VACUUM 실행으로부터 테이블을 보호
- VACUUM(FULL X) ANALYZE, CREATE INDEX CONCURRENTLY, ALTER TABLE VALIDATE, 기타 ALTER TABLE에 의해 Lock을 획득

5.4. Table level lock mode

▶ SHARE

- ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE Lock과 충돌
- CREATE INDEX(CONCURRENCY 옵션 X)에서 해당 Lock을 획득

▶ SHARE ROW EXCLUSIVE

- ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE Lock과 충돌
- 해당 모드는 동시 데이터 변경으로부터 테이블을 보호하고 한 번에 하나의 세션만 테이블을 차지할 수 있도록 처리
- CREATE TRIGGER와 다양한 형태의 ALTER TABLE에 의해 Lock을 획득

▶ EXCLUSIVE

- ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, ACCESS EXCLUSIVE Lock과 충돌
- ACCESS SHARE Lock만 허용
- 테이블에서 데이터를 읽기만 하는 트랜잭션이 있다면, 병렬로 해당 Lock을 잡을 수 있음

▶ ACCESS EXCLUSIVE

- 모든 Lock과 충돌, 일관성이 가장 유지가 되는 Lock
- DROP TABLE, TRUNCATE, REINDEX, CLUSTER, VACUUM FULL, REFRESH MATERIALIZED VIEW(CONCURRENTLY 옵션 X) 명령에 의해 얻을 수 있고
- Lock 모드를 명시적으로 지정하지 않는다면 LOCK TABLE 구문 기본 Lock

5.4. Table level lock mode

INZENT

▶ Conflicting Lock Modes

- Lock 충돌 체크 표 - Conflicting Lock Modes

Requested Lock Mode	Existing Lock Mode							
	ACCESS SHARE	ROW SHARE	ROW EXCL .	SHARE UPDATE EXCL .	SHARE	SHARE ROW EXCL .	EXCL .	ACCESS EXCL .
ACCESS SHARE								x
ROW SHARE							x	x
ROW EXCL .					x	x	x	x
SHARE UPDATE EXCL .				x	x	x	x	x
SHARE			x	x		x	x	x
SHARE ROW EXCL .			x	x	x	x	x	x
EXCL .		x	x	x	x	x	x	x
ACCESS EXCL .	x	x	x	x	x	x	x	x

5.5. Row level lock mode

▶ FOR UPDATE

- FOR UPDATE는 SELECT 문에 의해 검색된 행을 업데이트를 위해 Lock을 획득
- 트랜잭션이 끝날 때까지 다른 트랜잭션이 Lock을 얻을 수 없으므로 수정이나 삭제를 할 수 없음
- 해당 Lock이 걸려있는 행에서 UPDATE, DELETE, SELECT FOR UPDATE, SELECT FOR NO KEY UPDATE, SELECT FOR SHARE
- 또는 SELECT FOR KEY SHARE를 시도한다면 Lock이 풀릴 때까지 차단 됨
- FOR UPDATE Lock은 행의 DELETE와 특정 열의 값을 수정하는 UPDATE에 의해 획득

▶ FOR NO KEY UPDATE

- FOR UPDATE 보다 유한적인 Lock
- 동일한 행에서 lock을 얻으려고 하는 SELECT FOR KEY SHARE 명령을 차단하지 않음
- 해당 Lock은 FOR UPDATE Lock을 얻지 않은 모든 UPDATE에서 획득

▶ FOR SHARE

- SHARED Lock은 다른 트랜잭션이 해당 행에 대해 UPDATE, DELETE, SELECT FOR UPDATE, SELECT FOR NO KEY UPDATE를 수행하지 못하도록 차단
- SELECT FOR SHARE, SELECT FOR KEY SHARE 는 차단하지 않음

▶ FOR KEY SHARE

- FOR SHARE와 유사하지만 Lock의 강도가 더 약함
- SELECT FOR UPDATE 차단
- DELETE 또는 KEY 값을 변경하는 UPDATE는 수행하지 못하도록 차단
- SELECT FOR NO KEY UPDATE, SELECT FOR SHARE, SELECT FOR KEY SHARE를 차단하지 않음

5.5. Row level lock mode

▶ FOR UPDATE

- Row Level Lock 충돌 체크 표 - Conflicting Row Level Locks

Requested Lock Mode	Current Lock Mode			
	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR KEY SHARE				X
FOR SHARE			X	X
FOR NO KEY UPDATE		X	X	X
FOR UPDATE	X	X	X	X

5.6. Lock holder and Lock waiter

INZENT

▶ Lock holder

- Lock holder는 데이터베이스에서 특정 자원에 대한 잠금을 보유하고 있는 트랜잭션

▶ Lock waiter

- Lock waiter는 데이터베이스에서 특정 자원에 접근하려고 하지만, 이미 다른 세션이 해당 자원에 대한 잠금을 보유하고 있어 대기 중인 트랜잭션

5.7. Deadlock

▶ Deadlock

- 두 트랜잭션이 서로 상대방이 원하는 데이터에 대한 exclusive lock을 잡고 있는 경우 발생
- PostgreSQL은 자체적으로 deadlock을 감지하고 deadlock을 강제적으로 풀어버림

5.8. pg_locks

INZENT

▶ pg_locks

- 데이터베이스에서 현재 활성화된 모든 Lock에 대한 정보를 제공하는 시스템 뷰
  주요 컬럼
  - locktype: 잠금의 유형 (e.g., relation, page, tuple)
  - database: 잠금이 걸린 데이터베이스
  - relation: 잠금이 걸린 테이블의 OID
  - mode: 잠금 모드 (e.g., ExclusiveLock, AccessShareLock)
  - granted: 잠금이 획득되었는지 여부 (true/false)
- 데이터베이스 성능 저하나 데드락 문제를 해결하기 위해 현재 어떤 테이블이나 로우에 Lock이 걸려 있는지 확인

▶ Example

```
SELECT * FROM pg_locks;

SELECT * FROM pg_locks WHERE relation = (SELECT oid FROM
pg_class WHERE relname = 'your_table_name');

SELECT * FROM pg_locks WHERE mode = 'ExclusiveLock' AND
granted = true;

SELECT * FROM pg_locks WHERE granted = true;
SELECT * FROM pg_locks WHERE granted = false;
```

5.9. Practice

▶ **두 개의 트랜잭션으로 각각 UPDATE와 DELETE문으로 deadlock을 발생시키세요. (BEGIN)**

▶ **Lock waiter가 lock holder를 기다리는 SQL을 두 개의 터미널(psql)을 이용해 재현시키세요**

▶ **Lock holder와 Lock waiter를 하나의 로우로 조회하는 SQL을 작성하세요 (JOIN)**

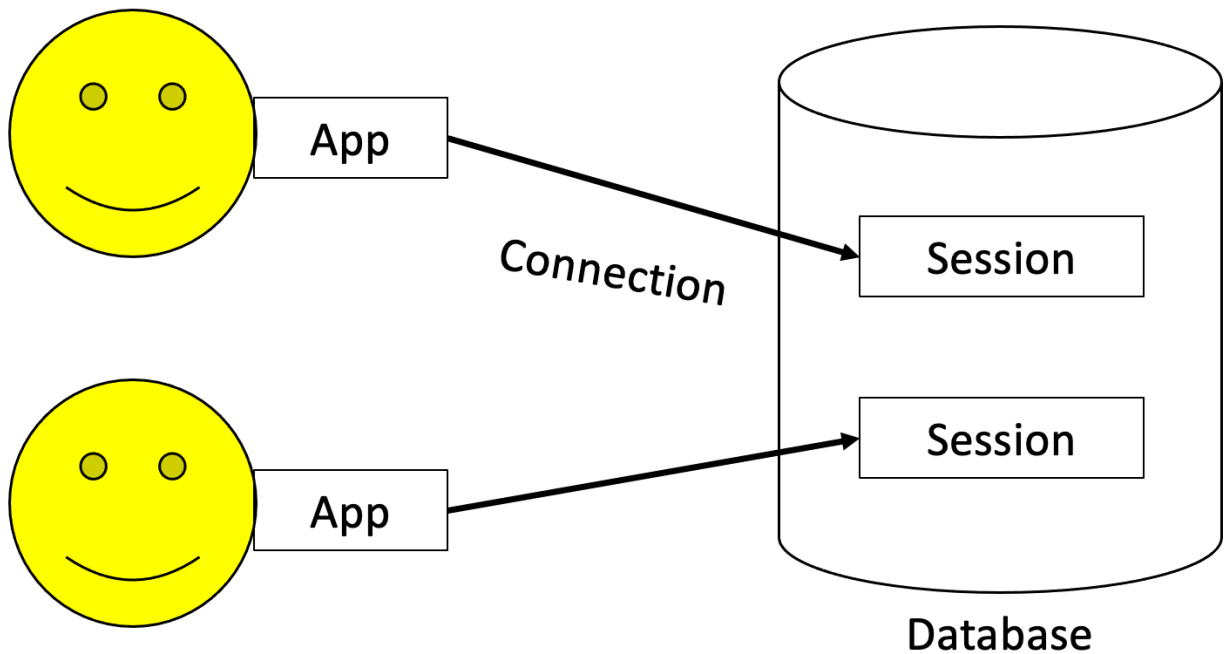
PostgreSQL Performance

SESSION

6.1. Session 정의

▶ Session

- 클라이언트가 데이터베이스에 접속하는 동안 연결을 유지하는 데 필요한 사용자 및 트랜잭션 정보를 저장해 놓은 임시 저장공간



6.2. pg_stat_activity

▶ pg_stat_activity

- 현재 데이터베이스에서 실행 중인 모든 세션에 대한 정보를 제공하는 시스템 뷰

  주요 컬럼
  - datid: 세션이 연결된 데이터베이스의 OID
  - datname: 세션이 연결된 데이터베이스 이름
  - pid: 세션의 프로세스 ID
  - usename: 세션을 시작한 사용자의 이름
  - application_name: 애플리케이션 이름
  - state: 세션의 현재 상태 (e.g., active, idle, idle in transaction, fastpath function call, disabled)
  - query: 현재 실행 중인 SQL 문
- 현재 데이터베이스에서 어떤 작업이 수행되고 있는지 모니터링하고, 긴 쿼리나 비효율적인 세션을 식별하여 성능을 최적화할 때 사용

6.3. Session kill

INZENT

▶ Session kill

- 세션을 종료시켜 클라이언트와 데이터베이스의 연결을 강제로 끊는 작업
- 접속한 클라이언트에 의해 수행 중이던 모든 SQL은 즉시 종료됨

▶ Example

```
SELECT pid, usename, datname, query
FROM pg_stat_activity;

-- pid : 12345
SELECT pg_terminate_backend(12345);
```

6.4. Statement cancel

INZENT

▶ Statement cancel

- 현재 실행 중인 SQL 문을 취소하는 작업
- 실행 중이던 SQL 문은 즉시 중단되며, 세션은 유지됨

▶ Example

```
SELECT pid, usename, datname, query
FROM pg_stat_activity;

-- pid : 12345
SELECT pg_cancel_backend(12345);
```

Practice

Contact Us

(주)인젠트
서울 영등포구 국제금융로2길 36, 유화증권빌딩 8층, 9층

Tel : 070.8209.6189 | Fax : 02.787.3699
info@inzent.com

www.inzent.com