## 테이블

테이블은 행(Row)과 칼럼(Column)으로 구성되는 가장 기본적인 데이터베이스 객체로 데이터베 이스 내에서 모든 데이터는 테이블을 통해 저장된다. 상용 DBMS들은 데이터를 저장하는 방식이 상 이한 여러 종류의 테이블을 제공하고 있으므로 테이블 설계 시에 성능, 확장성, 가용성 등을 고려해 테이블 유형을 선택하여야 한다.

테이블, 칼럼 등 데이터베이스에서 사용되는 객체의 명명 규칙은 표준화 관점에서 별도로 정의한다.

### 테이블(Table)

상용 DBMS에서 제공되는 테이블들은 제품마다 명칭이나 기능이 다르다. 테이블은 데이터의 저장 형태, 파티션 여부, 데이터의 유지 기간 등에 따라 다양하게 분류할 수 있다.

#### Heap-Organized Table

대부분의 상용 DBMS에서 표준 테이블로 사용하고 있는 테이블 형태로, 테이블 내에서 로우의 저 장 위치는 특정 속성의 값에 기초하지 않고 해당 로우가 삽입될 때 결정된다.

#### Clustered Index Table

Primary Key 값이나 인덱스 키 값의 순서로 데이터가 저장되는 테이블을 클러스터 인덱스라 한다. 클러스터 인덱스는 B 트리의 리프 노드에 RID 대신 데이터 페이지가 존재하는 구조이다.
데이터 페이지가 검색하고자 하는 키 값의 순서로 정렬되어 있기 때문에 프리 패치가 가능하고, 인덱 스에서 테이블로 탐침하는 경로가 단축되기 때문에 일반적인 인덱스를 이용하는 것보다 데이터를 더 빠르게 액세스할 수 있다. 그러나 데이터가 삽입될 때 키 순서에 따라 지정된 위치에 저장되어야 하 므로 데이터 페이지를 유지하는 데 많은 비용이 발생한다. 그리고 Heap 테이블로 사용하던 중에 클 러스터 인덱스로 전환하게 되면 데이터 페이지의 개편이 일어나고 관련된 모든 인덱스의 RID가 클 러스터 인덱스의 PK로 변경되어야 하므로 많은 오버헤드가 발생한다.

#### Partitioned Table

파티셔닝은 대용량의 테이블을 파티션이라는 보다 작은 논리적인 단위로 나눔으로써 성능이 저하 되는 것을 방지하고 관리를 보다 수월하게 하고자 하는 개념으로, 파티셔닝을 하는 방식에 따라 범 위 분할(Range Partitioning), 해쉬 분할(Hash Partitioning), 결합 분할(Composite Partitioning) 등이 있다.

파티셔닝은 대용량 데이터를 관리하는 데 아주 효과적이지만 무조건 파티션만 한다고 해서 파티션 이 가지고 있는 이점을 모두 취할 수 있는 것은 아니다. 잘못된 인덱스가 오히려 처리 속도에 나쁜 영향을 미치듯이 파티션 키를 어떻게 구성하느냐에 따라 많은 비효율을 초래할 수도 있으므로 다 음과 같은 전략적인 관점에서 파티션 키가 결정되어야 한다.

첫째, 액세스 유형에 따라 파티셔닝이 이루어질 수 있도록 파티션 키를 선정해야 한다. 분포도가 나빠 인덱스를 사용할 수 없을 경우 테이블 스캔을 해야 하는데, 대상 테이블이 대용량일 경우 절 대적인 작업량 때문에 성능적인 문제를 해소하기 어렵게 된다. 이럴 경우 검색의 범위가 파티션의단위와 일치하면 인덱스를 이용하지 않고도 원하는 범위의 데이터를 읽을 수 있게 된다.

둘째, 이력 데이터를 파티셔닝할 경우 파티션의 생성주기와 소멸주기를 일치시켜야 한다. 이력을 관리하는 데이터는 데이터 관리 전략 및 업무 규칙에 따라 그 수명이 다하게 되면 별도의 저장장치 에 기록되고 데이터베이스에서 삭제된다. 즉, 이력 데이터는 활용 가치에 따라 생성주기와 소멸주 기가 결정되므로 그 주기에 따라 데이터베이스를 정리해야만 한다. 만약 삭제해야 하는 데이터가 여러 파티션에 분산되어 있다면 그 데이터를 추출하여 삭제하는 데 많은 노력과 시간이 필요할 것 이다. 하지만 파티션이 데이터의 생성주기 및 소멸주기와 일치하면 파티션을 대상으로 작업을 수 행하므로 관리가 용이하게 된다.

■ External Table

외부 파일을 마치 데이터베이스 안에 존재하는 일반 테이블 형태로 이용할 수 있는 데이터베이스 객체이다. 데이터웨어하우스(DW, Data Warehouse)에서 ETL(Extraction, transformation, Loading) 작업 등에 유용한 테이블이다.

■ Temporary Table

트랜잭션이나 세션별로 데이터를 저장하고 처리할 수 있는 임시 테이블이다. 저장된 데이터는 트 랜잭션이 종료되면 휘발되며, 다른 세션에서 처리되는 데이터는 공유할 수 없다. 절차적인 처리를 하기 위해 임시적으로 사용할 수 있는 테이블이다.

### 칼럼(Column)

칼럼은 테이블을 구성하는 요소로, 데이터 타입(Data Type)과 길이(Length)로 정의된다. 데이터 타입은 데이터 일관성을 유지하는 가장 기본적인 기능이다. 표준화된 도메인을 정의하였다면 그에 따라 데이터 타입과 데이터 길이를 정의한다.

DBMS는 문자, 숫자, 시간, 대형 객체 등을 정의할 수 있는 내장 데이터 형식과 행, 컬렉션, 사용 자 정의 데이터 형식 등의 확장 데이터 형식을 지원하고 있다.

비교 연산에서 두 칼럼이 서로 데이터 타입과 길이가 다르면 DBMS는 내부적으로 데이터 타입을 변형한 후 비교 연산을 수행하므로 칼럼이 서로 참조 관계일 경우는 가능한 한 동일한 데이터 타입과 길이를 사용해야 한다. 만약 서로 다른 데이터 타입을 사용하게 되면 조인이나 비교 연산 시 인덱스 가 있어도 사용할 수 없게 되거나 실행 계획을 예측할 수 없게 된다.

칼럼 데이터 타입에 따라 물리적인 칼럼 순서 조정이 필요하다.

- 고정 길이 칼럼이고 NOT NULL인 칼럼은 선두에 정의한다.
- 가변 길이 칼럼을 뒤편으로 배치한다.
- NULL 값이 많을 것으로 예상되는 칼럼을 뒤편으로 배치한다.

이는 DBMS마다 차이는 있지만 값이 변경될 때 체인(Chain) 발생을 억제하고 저장 공간의 효율적 인 사용을 위해 필요하다.

오라클의 경우 인덱스를 구성하고 있는 모든 칼럼의 값이 NULL인 로우는 인덱스에 저장하지 않는 다. 그리고 NULL과의 비교는 IS NULL 혹은 IS NOT NULL을 통해서만 가능하다. 오라클의 NULL 처리 장점은 인덱스의 저장 공간을 절약하고 테이블을 전체 범위 스캔으로 쉽게 유도할 수 있다.SQL 서버는 NULL도 하나의 값으로 인식하고 관리하므로 인덱스를 구성하는 모든 칼럼 값이 NULL인 경우도 인덱스에 저장된다. 그리고 NULL과의 비교도 ansi_nulls 옵션을 적용하면 equal 연산자로도 비교가 가능하다.

set ansi_nulls off를 실행하면 where col1 = null과 같은 형태의 비교가 가능해지고, set ansi_nulls on을 실행하면 오라클과 마찬가지로 is null 또는 is not null 형태의 비교만 가능하게 된다. SQL 서버의 NULL 처리 장점은 is null 또는 is not null 같은 조건으로도 인덱스를 사용할 수 있다는 점이다.

데이터 타입과 길이 지정 시 다음과 같은 사항을 고려한다.

- 가변 길이 데이터 타입은 예상되는 최대 길이로 정의한다.
- 고정 길이 데이터 타입은 최소의 길이를 지정한다
- 소수점 이하 자리 수의 정의는 반올림되어 저장되므로 정확성을 확인하고 정의한다..

DBMS마다 다소 차이는 있지만 문자열을 비교하는 방법은 크게 두 가지로 구분할 수 있다. 하나 는 길이가 작은 칼럼 끝에 공백을 추가하여 길이를 같게 한 후 비교하는 방식이고, 다른 하나는 공백 을 추가하지 않고 비교하는 방식이다.

오라클의 경우 비교하는 칼럼이 모두 CHAR, NCHAR, 문자 상수인 경우나 사용자 정의 함수에 의하 여 리턴된 값일 경우에는 공백 추가 후 비교하는 방식을 사용하고, 비교하는 값 중에 VARCHAR2 혹은 NVARCHAR2가 존재하면 공백 추가 없이 비교를 수행한다. 이때 비교하는 칼럼의 길이가 서로 다르면 짧은 칼럼의 길이까지 비교하고 길이가 긴 칼럼을 크다고 판단하므로 CHAR와 VARCHAR2를 비교할 경우 VARCHAR2의 칼럼이 길이보다 짧은 값이 입력된 데이터는 동일한 데이터 값이 들어 있어도 서 로 다른 값으로 인식한다.

SQL 서버는 문자열을 처리하는 방식이 오라클과 약간 차이가 있다. 오라클의 경우 CHAR 타입은 공백이 추가되어 저장되고, VARCHAR 타입은 공백 추가 없이 저장하지만 SQL 서버는 ANSI_PADDING이라는 옵션에 의하여 공백 추가 여부를 결정한다. ON으로 설정하면 오라클과 동일 한 방식으로 저장되지만 OFF로 설정하면 CHAR와 VARCHAR 타입 모두 저장된 문자열 값에서 후행 공백이 지워진다. SQL 서버에서 ANSI_ PADDING을 ON으로 설정하고 CHAR와 VARCHAR를 비 교할 경우 오라클과 달리 길이보다 짧은 값이 입력된 데이터도 동일한 데이터로 인식한다.

다른 두 데이터 타입을 비교할 경우 오라클과 SQL 서버는 내부적으로 우선순위가 높은 데이터 타 입으로 변환하여 비교를 수행한다. 예를 들어 CHAR 데이터 타입과 NUMBER 데이터 타입을 비교 할 경우 NUMBER 데이터 타입이 CHAR 데이터 타입보다 우선순위가 높기 때문에 내부적으로 CHAR 데이터 타입을 NUMBER 데이터 타입으로 변환하여 비교를 수행한다. 이때 문자열 칼럼에 알파벳이 등이 섞여있어 변환이 불가능하면 SQL 오류가 발생한다.

그렇지만 다른 두 데이터 타입이 Like로 비교되면 위의 규칙과는 반대의 상황이 발생한다.

```
SELECT a.사원번호, a.입사일자, a.부서,
...
FROM 사원 a
WHERE a.사원번호          Like '200902%'
...
```

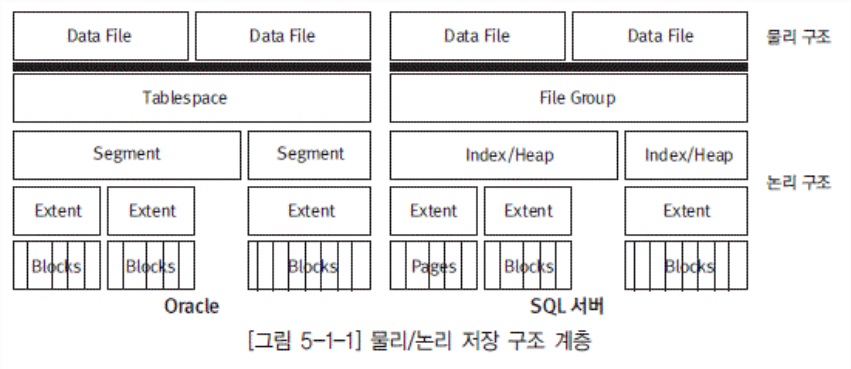
위 SQL에서 사원번호가 NUMBER 타입일 경우 앞에서 언급한 변환 규칙을 적용하면 문자열 '200902%'를 숫자로 변환해야 하지만 변환이 불가능한 문자를 포함하고 있다. 변환도 불가능하지 만 Like가 문자열을 비교하기 위한 연산자이므로 문자를 숫자로 변환하면 Like 비교가 불가능해진 다. 즉, Like 비교 시에는 위의 규칙을 적용하지 않고 비교 연산에 참여하는 대상을 문자열로 변환하 여 비교를 수행한다.

### 테이블 설계시 고려사항

- 칼럼 데이터 길이 합이 1 블록(Block) 사이즈보다 큰 경우 수직 분할을 고려한다. 1 블록 사이즈보 다 크면 체인이 발생하여 속도 저하 현상을 유발한다.
- 칼럼 길이가 길고 특정 칼럼의 사용 빈도 차이가 심한 경우이거나 각기 다른 사용자 그룹이 특정 칼럼만을 사용하고 같이 처리되는 경우가 드문 경우는 수직 분할을 고려한다.
- 수직 분할을 고려할 때는 분할되는 테이블이 하나의 트랜잭션에 의해 동시에 처리되는 경우나 조 인이 빈번히 발생되는 경우가 없어야 한다.
- "주문일자", "계약일자"등과 같이 검색 조건으로 빈번하게 사용되는 칼럼은 시간 데이터 타입을 사용하면 비교 연산을 하거나 조인일 때 동일 데이터 타입으로 가공하는 경우가 일어날 수 있으므 로 액세스 측면만 고려한다면 문자 타입을 사용하는 것이 더 효율적이다.
- 사건의 일자나 시간을 기록하는 속성을 문자 타입으로 정의하면 일자 범위를 벗어나는 값이 입력 될 수 있으므로 처리 시 문제가 발생하지 않도록 하려면 오류 데이터들을 클린징하거나 제외하기 위한 복잡한 로직을 추가해야 한다. 이러한 문제와 더불어 데이터 품질의 중요성이 대두되면서 최 근에는 시간을 기록하는 속성인 경우, 시간 데이터 타입을 선택하는 추세이다.

## 테이블(Table)과 테이블스페이스(Table space)

테이블은 테이블스페이스라는 논리적인 단위를 이용하여 관리하고, 테이블스페이스는 물리적인 데이터 파일을 지정하여 저장된다. 테이블, 테이블스페이스, 데이터 파일로 분리하여 관리함으로써 논리적인 구성이 물리적인 구성에 종속되지 않고 투명성을 보장할 수 있다.



[그림 5-1-1] 물리/논리 저장 구조 계층

테이블스페이스(파일 그룹)는 저장되는 내용에 따라 테이블용, 인덱스용, 임시(Temporary)용을 구분하여 설계한다. 이는 백업 단위나 공간 확장 단위인 물리적인 파일 크기를 적정하게 유지하기 위 해서이다. 따라서 테이블스페이스는 데이터 용량을 관리하는 단위로 이용된다.

다음은 일반적인 데이터용/인덱스용 테이블스페이스 설계 유형들이다.

- 테이블이 저장되는 테이블 스페이스는 업무별로 지정한다.
- 대용량 테이블은 독립적인 테이블 스페이스를 지정한다.
- 테이블과 인덱스는 분리하여 저장한다.
- LOB 타입 데이터는 독립적인 공간을 지정한다.

## 용량 설계

용량 설계는 다음과 같은 목적으로 진행된다.

- 정확한 데이터 용량을 예측하여 저장 공간을 효과적인 사용과 저장 공간에 대한 확장성을 보장하 여 가용성을 높이기 위함
- H/W 특성을 고려하여 디스크 채널 병목을 최소화하기 위함
- 디스크 I/O를 분산하여 접근 성능을 향상하기 위함
- 테이블이나 인덱스에 맞는 저장 옵션을 지정하기 위함

테이블 저장 옵션에 대한 고려사항은 다음과 같다.

- 초기 사이즈, 증가 사이즈
- 트랜잭션 관련 옵션
- 최대 사이즈와 자동 증가

저장 용량 설계 절차는 다음과 같다.

- 용량 분석 – 데이터 증가 예상 건수, 주기, 로우 길이(Row Length)등을 고려함
- 오브젝트별 용량 산정 – 테이블, 인덱스에 대한 크기
- 테이블스페이스별 용량 산정 – 테이블스페이스별 오브젝트 용량의 합계
- 디스크 용량 산정 – 테이블스페이스에 따른 디스크 용량과 I/O 분산 설계

저장 공간 설계 | 무결성 설계 | 인덱스 설계 | 분산 설계 | 보안 설계

## 데이터 무결성

데이터의 정확성, 일관성, 유효성, 신뢰성을 위해 무효 갱신으로부터 데이터를 보호하기 위해 무결 성 설계가 필요하다. 데이터 모델링 과정에서 정의된 일련의 규칙에 따라 데이터가 생성, 수정, 삭제 될 수 있도록 프로그램이나 데이터베이스 기능을 이용한다. 그 결과로 권한이 부여된 사용자에 의해 야기될 수 있는 의미적 에러를 방지하고, 데이터베이스 내의 데이터가 현실 세계의 올바른 데이터를 갖도록 보장하는 것이다.

### 데이터 무결성 종류

데이터 무결성은 실체 무결성, 영역 무결성, 참조 무결성, 사용자 정의 무결성 4가지가 있다.

[표 5-1-1] 데이터 무결성 분류

| 분류 | 설명 |
|---|---|
| 실체 무결성 | 실체는 각 인스턴스를 유일하게 식별할 수 있는 속성이나 속성 그룹을 가져야 한다. |
| 영역 무결성 | 칼럼 데이터 타입, 길이, 유효 값이 일관되게 유지되어야 한다. |
| 참조 무결성 | 데이터 모델에서 정의된 실체 간의 관계 조건을 유지하는 것이다. |
| 사용자 정의 무결성 | 다양하게 정의될 수 있는 비즈니스 규칙이 데이터적으로 일관성을 유지하는 것이다. |

### 데이터 무결성 강화 방법

데이터 무결성은 데이터 품질에 직접적인 영향을 준다. 프로그램이 완성되고 데이터가 축적된 후 데이터 크린징을 하거나 무결성 방법을 강구할 때는 많은 비용이 발생된다. 데이터 품질을 확보하기 위해서는 데이터베이스 구축 과정에서 적정한 무결성 방안을 확보해야 한다.

데이터베이스에서 모든 무결성 제약을 정의할 수 없으므로 복잡한 규칙에 의해 데이터 상호 간에 유지해야 할 정합성은 애플리케이션 내에서 처리를 해야 한다.

[표 5-1-2] 무결성 강화 방법

| 분류 | 설명 |
|---|---|
| 애플리케이션 | 데이터를 조작하는 프로그램 내에 데이터 생성, 수정, 삭제 시 무결성 조건을 검증하는 코드를 추가함 |
| 데이터베이스 트리거 | 트리거 이벤트 시 저장 SQL을 실행하여 무결성 조건을 실행함 |
| 제약 조건 | 데이터베이스 제약 조건 기능을 선언하여 무결성을 유지함 |

3가지 방법은 무결성 종류에 따라 장단점이 존재하므로 선택적으로 적용한다. 필요 이상 혹은 이 하의 부적절한 무결성 강화 방법을 적용했을 때에는 성능 측면이나 운영적인 측면에서 불필요한 노 력이 발생한다.

[표 5-1-3] 무결성 강화 방법에 따른 장단점

| 구분 | 장점 | 단점 |
|---|---|---|
| 애플리케이션 | - 사용자 정의 같은 복잡한 무결성조건을 구현함 | - 소스코드에 분산되어 관리의 어려움이 있음 - 개별적으로 시행되므로 적정성 검토에 어려움 |
| 데이터베이스 트리거 | - 통합 관리가 가능함 - 복잡한 요건 구현 가능 | - 운영 중 변경이 어려움 - 사용상 주의가 필요함 |
| | - 통합 관리가 가능함 - 간단한 선언으로 구현 가 | |

| | | |
|---|---|---|
| 제약조건 | 능 – 변경이 용이하고 유효/무효 상태 변경이 가능함 – 원천적으로 잘못된 데이터 발생을 막을 수 있음 | – 복잡한 제약 조건 구현이 불가능 – 예외적인 처리가 불가능 |

## 실체 무결성

실체 무결성은 실체에서 개체의 유일성을 보장하기 위한 무결성으로 반드시 보장되어야 하므로 프 로그램이나 트리거 등을 이용하는 것보다 데이터베이스에서 제공하는 PK(Primary key) 제약 조건 과 Unique 제약 조건 등을 이용하여 보장하는 것이 좋다.

■ 기본키 제약

데이터베이스에서 가장 중요한 무결성 조건으로 식별자 값은 NOT NULL이고 유일(UNIQUE)해 야 한다. PK가 없는 테이블 운영은 데이터 조작 시 이상 현상이 발생하거나 조인에 의해 집계된 금액이 잘못 계산될 가능성이 높다.

■ UNIQUE 제약

실체 무결성은 식별자 외에 실체 내에 후보키 대상인 UNIQUE 칼럼도 대상이다. 상품 테이블에서 상품 코드가 PK이고 업무 요건에 의해서 상품 바코드도 유일한 값을 가져야 한다면 실체 무결성 을 위해 상품 바코드에 UNIQUE 제약 조건을 적용해야 한다. PK 제약 조건과 UNIQUE 제약 조 건의 차이는 상품 코드는 NULL이 존재해서는 안되고, 상품 바코드는 경우에 따라 바코드 값이 NULL이 될 수 있다.

PK 제약 조?? 적용하면 DBMS 내부적 인 적용은 다르지만 동일한 결과를 얻을 수 있다. 상용 DBMS 중 PK를 선언할 경우 해당 테이블 이 Clustered Index 테이블 구조로 만들어진다. 만약 대용량의 Heap 테이블에 PK를 생성할 경 우 데이터 저장 구조가 변경되어 많은 시간이 소요될 것이다.

PK나 UNIQUE 제약 조건을 적용하면 Unique Index가 생성된다. 이는 데이터 변경 시 제약 조 건을 만족하는지 확인하는 수단으로 인덱스가 필요하기 때문이다. 실체 무결성은 칼럼을 대상으로 제약 조건을 생성하지만 테이블 내의 모든 행에 대해 칼럼 값을 비교하여 실체 무결성을 검증한다.

■ 식별자 설계 – 채번

정보시스템 구축 과정에서 이슈화되는 요소 중 하나가 채번 문제이다. 식별자 발생 규칙에 따라 심 각한 블로킹 현상이 발생하여 전체적인 시스템 성능 저하를 초래한다. 대량의 트랜잭션 처리를 위 해서는 식별자를 데이터베이스에서 제공하는 일련번호를 발생시키는 시퀀스(Sequence) 같은 객 체나 시리얼(Serial) 같은 데이터 타입을 이용하여 해결이 가능하다.

## 영역 무결성

영역 무결성은 칼럼에 적용되어 단일 로우의 칼럼 값만으로 만족 여부를 판단할 수 있다. 영역 무결성에 대한 예는 다음과 같은 것이 있다.

예1) 주문일 칼럼 값이 20050230이라면 유효한 데이터가 아니다. 2월 30일은 존재할 수 없기 때문이다.

예2) 근무상태를 1 : 정상, 2 : 휴직, 3 : 퇴직 등으로 3가지 상태를 비즈니스 규칙에 의해서 유효값으로 정의된 경우 4 : 복직 이나 NULL 상태는 존재할 수 없다.

예3) 상품 테이블에 상품명은 필수 입력 사항이면 상품명에 NULL 값은 존재할 수 없다.

프로그램 소스와 제약 조건을 상호 보완적으로 사용을 하는 것이 효과적이다. 영역 무결성은 프로그램 기능에 의해서 유효 값에 대한 검증을 선행하고 추가적으로 제약 조건을 선언하여 무결성을 강화하면 효과적이다.

■ 데이터 타입(Data Type) & 길이(Length)

예1)에서 주문일 칼럼을 시간 타입인 DATE로 정의했다면 2월 30일이라는 데이터를 등록되지 않을 것이다. 하지만 DBMS에서 DATE TYPE은 년월일의 일자와 시분초의 시간을 포함한 값을 가진다. 이런 이유로 비교 연산을 하기 위해서는 데이터 값 변환이나 데이터 타입 변환이 필요하다. 따라서 주문일과 같이 조회 조건이나 비교 연산에 많이 사용되는 칼럼은 문자 데이터 타입으로 정의하고 프로그램에 의해서 유효일자인지를 확인하는 것이 효과적이다.

■ 유효 값(CHECK)

예2)와 같은 경우는 칼럼 유효 값에 대한 제약 조건으로 CHECK를 이용한다. CHECK 무결성 제한은 참이 되어야 하는 조건을 명시적으로 정의한다.

■ NOT NULL

예3)에서 상품명이 반드시 존재하기 위해 NOT NULL 제약 조건을 사용한다. 숫자 타입의 칼럼은 계산에 이용되는 경우가 많다. 금액 계산시 NULL 값이 존재하면 연산이 불가능하여 예외 사항이 발생한다. 이를 방지하기 위해 숫자 타입의 칼럼은 NOT NULL 제약 조건을 부여하고 기본(DEFAULT) 값으로 0'을 정의한다.

## 참조 무결성

참조 무결성은 두 실체 사이의 관계 규칙을 정의하기 위한 제약 조건으로 데이터가 입력, 수정, 삭 제될 때 두 실체의 튜플들 사이의 정합성과 일관성을 유지하는 데 사용된다. 참조 무결성 제약 조건 은 어떤 실체의 튜플이 다른 실체에 있는

튜플을 참조하려면 참조되는 튜플이 반드시 그 실체 내에 존재하여야 한다는 것이다.

■ 입력 참조 무결성

[표 5-1-4] 입력 참조 무결성

| 구분 | 설명 |
|---|---|
| DEPENDENT | 참조되는(부모) 테이블에 PK 값이 존재할 때만 입력을 허용 |
| AUTOMATIC | 참조되는(부모) 테이블에 PK 값이 없는 경우는 PK를 생성 후 입력 |
| DEFAULT | 참조되는(부모) 테이블에 PK 값이 없는 경우 지정된 기본값으로 입력 |
| CUSTOMIZED | 특정한 조건이 만족할 때만 입력을 허용 |
| NULL | 참조되는(부모) 테이블에 PK 값이 없는 경우 외부키를 NULL 값으로 처리 |
| NO EFFECT | 조건 없이 입력을 허용 |

입력 참조 무결성은 복잡한 처리 규칙을 위해 애플리케이션에서 구현한다. 반드시 참조 무결성을 유지해야 하는 경우에는 추가적으로 FK 제약을 사용한다. 이는 부모 없는 자식 데이터를 생성하 지 않는 확실한 방법이다.

■ 수정 삭제 참조 무결성

[표 5-1-5] 수정/삭제 참조 무결성

| 구분 | 설명 |
|---|---|
| RESTRICT | 참조하는(자식) 테이블에 PK 값이 없는 경우 삭제/수정 허용 |
| CASCADE | 참조되는(부모) 테이블과 참조하는 테이블의 외부키를 연쇄적 삭제/수정 |
| DEFAULT | 참조되는(부모) 테이블의 수정을 항상 허용하고 참조하는(자식) 테이블의 외부키를 지정된 기본값으로 변경 |
| CUSTOMIZED | 특정한 조건이 만족할 때만 수정/삭제 허용 |
| NULL | 참조되는(부모) 테이블의 수정을 항상 허용하고 참조하는(자식) 테이블의 외부키를 NULL 값으로 수정 |
| NO EFFECT | 조건 없이 삭제/수정 허용 |

수정 참조 무결성은 부모 식별자가 변경되었을 경우이다. 식별자 값 변경이 없이 데이터가 운영되 는 것이 바람직하다. 변경이 불가피한 경우라도 삭제되고 다시 입력되는 규칙을 적용하여 애플리 케이션에서 구현한다.

삭제 참조 무결성은 제한(RESTRICT) 기능으로 DBMS가 FK 제약을 이용한다.

■ 디폴트 규칙 정의의 필요성

참참조 무결성에서 NULL 값을 정의하는 것은 바람직하지 않다. NULL은 알 수 없는 상태이거나 아 직 정의되지 않은 상태를 의미한다. 따라서 직관적으로 예상할 수 없는 상태이다. 따라서 NULL 조건은 DEFAULT 조건으로 설계하는 것이 바람직하다.

■ 모델상에서 슈퍼타입(SUPER-TYPE)-서브타입(SUB-TYPE) 관계

삽입 시에는 DEPENDENT, AUTOMATIC 조건을 적용하고, 삭제나 변경 시에는 CASCADE 조 건을 적용하는 것이 바람직하다.

참조 무결성은 데이터베이스에서 제공하는 FK를 정의하는 것만으로는 구현이 불가능하다. 모든 제약 옵션을 데이터베이스가 지원하지 않기 때문이다. 따라서 프로그램, 데이터베이스 트리거, FK 제약을 사용해야 한다. 추가적으로 대량의 트랜잭션이 발생하는 데이터베이스 환경에서는 FK 제약 조건 생성 범위를 조정할 필요가 있다. FK 제약은 가장 확실하고 간편한 방법이지만 성능상에 문제 를 유발할 수 있다. 반드시 적용되어야 하는 부모 자식 관계에 적용하는 것이 일반적이다.

FK 제약은 인덱스를 생성한다. 자식 테이블의 데이터 변경은 부모 테이블에 PK 제약으로 만들어 진 UNIQUE 인덱스를 이용하고, 부모 테이블의 데이터 변경은 자식 테이블의 인덱스를 이용한다. 인덱스가 없다면 테이블 FULL SCAN이 발생할 것이다.

## 인덱스 기능

인덱스는 어떤 종류의 검색 연산을 최적화하기 위해 데이터베이스상에 로우들의 정보를 구성하는 데이터 구조이다. 인덱스를 이용하면 전체 데이터를 검색하지 않고 데이터베이스에서 원하는 정보를 빠르게 검색할 수 있다. 예를 들어, 테이블에는 수백만의 고객 정보가 저장되어 있고 고객명, 고객번 호, 주민번호 등을 이용해 데이터를 검색하고자 할 때 인덱스가 없다면 찾고자 하는 대상이 한 명이 더라도 수백만의 고객 데이터 전체를 읽어야 한다. 인덱스는 인덱스를 생성한 칼럼 값으로 정렬되어 있고 테이블 내 값들이 저장된 위치를 갖고 있으므로 인덱스를 이용하면 전체 고객 테이블을 읽지 않 고도 찾으려는 고객 데이터를 찾을 수 있다. 그래서 테이블에 데이터가 수천만 개로 증가하여도 인덱 스를 이용한 접근 경로와 검색 범위가 동일하다면 속도의 변화는 거의 발생하지 않는다.

1개 이상의 데이터를 이용하여 비교 연산을 수행하는 가장 효과적인 방법은 정렬 후 값을 비교하 는 것이다. 그런데 비교 연산을 수행할 때마다 정렬을 하면 많은 비용이 발생한다. 이때 정렬된 결과 를 유지하는 데 효과적인 B 트리 형태로 정렬된 결과를 저장해두면 비교 연산을 수행할 때마다 정렬 을 수행하지 않고 이용할 수 있다.

인덱스의 가장 중요한 기능은 접근 경로를 단축함으로써 데이터의 탐색 속도를 향상시키는 것이 다. 데이터의 탐색은 데이터베이스에서 가장 중요하고 빈번한 연산이므로 DBMS는 인덱스를 활용하 여 탐색 시 발생하는 비용을 최소화하고자 한다.

예를 들어, 무결성 검증을 위해 어떤 집합을 탐색하고자 하는 경우에 인덱스를 활용할 수 있다. PK(Primary Key), FK(Foreign Key), UK(Unique Key)에 대한 제약 조건을 확인하려면 해당하 는 속성으로 데이터가 정렬되어 있어야 한다.

그 외에 드라이빙 집합을 탐색하거나 Outer 집합의 결과를 가지고 Inner 집합에서 검색 조건과 일치하는 데이터를 추출하고자 할 때 주로 활용한다. 드라이빙 집합을 탐색할 때 검색 조건에 인덱스 가 존재하지 않으면 전체 데이터를 읽어야 하고, Inner 집합을 탐침할 때 Inner 집합의 조인 속성에 인덱스가 존재하지 않으면 조인 시 카티션 프로덕트가 발생한다 .

## 인덱스 설계 절차

인덱스는 특정 응용 프로그램을 위해서 생성되는 것이 아니다. 최소의 인덱스 구성으로 모든 접근 경로를 제공할 수 있어야 전략적인 인덱스 설계가 된다. 따라서 인덱스 선정은 테이블에 접근하는 모 든 경로를 수집하고 수집된 결과를 분석하여 종합적인 판단에 의해서 결정되는 것이 바람직하다.인덱스는 특정 애플리케이션을 위해서 생성되는 것이 아니다 . 최소의 인덱스 구성으로 모든 접근 경로를 제공할 수 있어야 전략적인 인덱스 설계가 된다. 따라서 인덱스 선정은 테이블에 접근하는 모 든 경로를 수집하고 수집된 결과를 분석하여 종합적인 판단에 의해 결정하는 것이 바람직하다.

### 접근 경로 수집

접근 경로는 테이블에서 데이터를 검색하는 방법으로, 테이블 스캔과 인덱스 스캔 등이 있다. 접근 경로를 수집한다는 의미는 SQL이 최적화되었을 때 인덱스 스캔을 해야 하는 검색 조건들을 수집하 는 것이므로 데이터베이스 설계 시 혹은 완성되지 않은 프로그램에서 사용될 모든 접근 경로를 예측 하기는 불가능하다. 따라서 프로그램 설계서, 화면 설계 자료, 프로그램 처리 조건 등을 고려하여 예 상되는 접근 경로를 수집하여야 한다. 수집은 테이블 단위로 진행하고, 다음과 같은 점을 고려하여 접근 유형을 목록화한다.

■ 반복 수행되는 접근 경로

대표적인 것이 조인 칼럼이다. 조인 칼럼은 FK 제약 대상이기도 하다. 주문 1건당 평균 50개의 주 문 내역을 갖는다면 주문 테이블과 주문 내역 테이블을 이용하여 주문서를 작성하는 SQL은 조인 을 위해 평균 50번의 주문 내역 테이블을

반복 액세스한다.

■ 분포도가 양호한 칼럼

주문번호, 청구번호, 주민번호 등은 단일 칼럼 인덱스로도 충분한 수행 속도를 보장 받을 수 있는 후보를 수집한다.

■ 조회 조건에 사용되는 칼럼

성명, 상품명, 고객명 등 명칭이나 주문일자, 판매일, 입고일 등 일자와 같은 칼럼은 조회 조건으 로 많이 이용되는 칼럼이다.

■ 자주 결합되어 사용되는 칼럼

판매일 + 판매부서, 급여일+급여부서와 같이 조합에 의해서 사용되는 칼럼 유형을 조사한다.

■ 데이터 정렬 순서와 그룹핑 칼럼

조건뿐만 아니라 순방향, 역방향 등의 정렬 순서를 병행하여 찾는다. 인덱스는 구성 칼럼 값들이 정렬되어 있어 인덱스를 이용하면 별도의 ORDER BY에 의한 정렬작업이 필요 없다. 동일한 원리 로 그룹핑 단위(GROUP BY)로 사용된 칼럼도 조사한다.

■ 일련번호를 부여한 칼럼

이력을 관리하기 위해서 일련번호를 부여한 칼럼에 대해서도 조사를 실시한다. 마지막 일련번호를 찾는 경우가 빈번히 발생하므로 효과적인 액세스를 위해서 필요하다.

■ 통계 자료 추출 조건

통계 자료는 결과를 추출하기 위해서 넓은 범위의 데이터가 필요하다. 따라서 다양한 추출 조건을 사전에 확보하여 인덱스 생성에 반영하여야 한다.

■ 조회 조건이나 조인 조건 연산자

위에 제시되는 유형의 칼럼과 함께 적용된 =, between, like 등의 비교 연산자를 병행 조사한다. 이는 인덱스 결합 순서를 결정할 때 중요한 정보로 사용된다.

위에서 제시되는 칼럼들은 인덱스 생성이 필요한 칼럼들이다. 운영 중인 시스템을 대상으로 인덱 스를 설계할 때는 사용하고 있는 애플리케이션 내에 SQL 문장을 수집하거나 트레이스(Trace)를 사 용해 SQL들을 추출하여 접근 경로를 수집할 수 있다. 사후 작업은 많은 공수가 필요하지만 정확한 접근 경로를 추출할 수 있다. 따라서 설계 단계에서 위와 같은 칼럼으로 접근 경로로 추출해 인덱스 설계에 이용하고, 개발이 완료된 후 시스템 운영 초기나 성능에 문제가 발생되었을 때 인덱스 설계를 보완하는 것이 일반적인 형태이다.

### 분포도 조사에 의한 후보 칼럼 선정

수집된 접근 경로 칼럼들을 대상으로 분포도를 조사한다. 설계 단계에서는 실제 분포도를 예측할 수 없으므로 현재 시스템 데이터를 참고하거나 업무에서 예상한 상황을 고려하여 분포도를 예측한다.

분포도(%) = 데이터별 평균 로우 수 / 테이블의 총 로우 수 × 100

- 일반적으로 분포도가 10 ~ 15% 정도이면 인덱스 칼럼 후보로 사용한다. 그런데 B 트리 인덱스의 특성상 분포도가 증가하게 되면 인덱스의 효율이 점차적으로 떨어져 어느 시점이 되면 오히려 인 덱스를 통해 검색하는 것이 전체 데이터를 읽는 것보다 더 느리게 된다. 이러한 반환점을 인덱스의 손익 분기점이라 한다. 분포도가 15% 정도일 때 인덱스 후보로 추천하는 것은 평균적인 인덱스의 손익??덱스의 손익 분기점이 생기는 것은 인덱스를 통해 액세스하는 방식과 테이블 스캔으로 액세스하 는 방식이 다르기 때문이다. 인덱스를 통한 액세스는 싱글 블록으로 입출력(I/O)을 하고, 테이블 스캔은 멀티 블록으로 입출력을 하기 때문이다. 분포도가 나쁜 경우 인덱스를 경유하여 액세스하 면 검색하는 로우의 수는 적지만 입출력 횟수는 테이블 스캔보다 훨씬 많아진다.
- 분포도는 단일 칼럼을 대상으로 조사한다.
- 단일 칼럼으로 만족하지 않는 경우 검색 조건을 좁히기 위해 결합 칼럼들에 대한 분포도 조사를 실 시한다. 예를 들어, 고객번호 + 주문일자로 분포도를 조사한다.
- 분포도 조사 결과를 만족하는 칼럼은 인덱스 후보로 선정하고 인덱스 후보 목록을 작성한다. 인덱 스 후보는 중복이 없게 최소 조합으로 선별해야 한다. 예를 들어, 주문일자 + 고객번호, 고객번호 + 주문일자, 주문일자 + 고객번호 + 주문상품 등이 접근 경로로 도출된 경우는 주문일자 + 고객번 호에 대해 분포도를 조사하고, 만족한 결과라면 후보를 단일화한다.
- 기형적으로 분포도가 불규칙한 경우는 별도 표시하여 접근 형태에 따라 대책을 마련해야 한다.
- 빈번히 변경이 발생하는 칼럼은 인덱스 후보에서 제외한다.

### 접근 경로 결정

인덱스 후보 목록을 이용하여 접근 유형에 따라 어떤 인덱스 후보를 사용할 것인가를 결정한다. 만약 누락된 접근 경로가 있다면 분포도 조사를 실시하고 인덱스 후보 목록에 추가 작업을 반복한다.

### 칼럼 조합 및 순서 결정

단일 칼럼의 분포도가 양호하면 단일 칼럼 인덱스로 확정한다. 하지만 하나 이상의 칼럼 조합이 필 요한 경우는 아래와 같은 요소를 고려하여 인덱스 칼럼 순서를 결정한다.

■ 항상 사용되는 칼럼을 선두 칼럼으로 한다.

결합 인덱스는 선행되는 칼럼이 존재하지 않을 경우 인덱스를 이용하지 못한다. 따라서 항상 접근 경로에 있는 칼럼을 선두 칼럼 또는 선행 칼럼으로 사용해야 한다.

■ 등치( = )조건으로 사용되는 칼럼을 선행 칼럼으로 한다.

인덱스 특성상 <, >, <=, >=, BETWEEN, LIKE 등의 비교 연산이 적용된 결합 인덱스는 해당 칼럼 이후는 Range scan 이 실시된다.

예를 들어 접근경로가 col1 LIKE 'A100%'AND col2 = '20051010'인 경우

- 인덱스가 COL1 + COL2로 구성된 경우는 A100으로 시작하는 모든 일자에 대한 인덱스를 검색한다.
- 인덱스가 COL2 + COL1로 구성된 경우는'20051010'을 만족하는 데이터 중 A100으로 시작 된 데이터만 인덱스를 검색한다.

■ 분포도가 좋은 칼럼을 선행 칼럼으로 한다.

분포도가 좋은 칼럼을 선두로 하는 결합 인덱스를 구성하면 후행 칼럼 조건이 없을 경우에도 효과적으로 인덱스 사용이 가능하다.

■ ORDER BY, GROUP BY 순서를 적용 한다.

칼럼 값이 정렬되어 있으므로 ORDER BY나 GROUP BY 절에 사용되는 칼럼 순으로 인덱스를 생성하며 별도의 정렬 부하를 줄일 수 있다.

### 적용 시험

설계된 인덱스를 적용하고 접근 경로별로 인덱스가 사용되는지 시험할 필요가 있다. 여러 개의 접 근 경로가 존재하는 테이블은 여러 개의 인덱스가 만들어지므로 의도한 실행 계획으로 동작하는지 확인해야 한다. 특히 운영 중인 시스템에서 새로운 인덱스를 생성할 경우는 적용되는 SQL뿐만 아니 라 동일한 테이블을 사용하는 기존 SQL에 영향을 줄 수 있으므로 반드시 확인 작업을 하여야 한다.

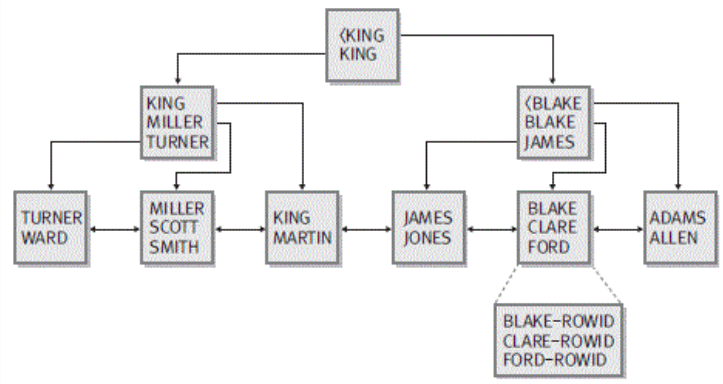
## 인덱스 구조

인덱스는 인덱스를 구성하는 구조나 특징에 따라 트리 기반 인덱스, 해쉬 기반 인덱스, 비트맵 인 덱스, 함수 기반 인덱스, 조인 인덱스, 도메인 인덱스 등으로 나눌 수 있다.

### 트리 기반 인덱스

대부분의 상용 DBMS에서는 트리 구조를 기반으로 하는 B+ 트리 인덱스를 주로 사용한다. B+ 트 리는 루트에서 리프 노드까지 모든 경로의 깊이가 같은 밸런스 트리(balanced tree) 형태로, 다른 인덱스에 비해 대용량 처리의 데이터 삽입과 삭제 등에 좋은 성능을 유지한다.

B 트리는 키 값이 트리상에 한 번만 나타나기 때문에 키 값을 순차적으로 액세스하고자 할 때 트리 를 탐색하는 데 많은 비용이 발생한다. 이러한 트리 탐색 비용을 최소화하기 위해 B+ 트리에서는 모 든 키 값을 리프 노드에 내리고 양방향으로 연결(doubly linked list)해 놓았다. 이러한 구조로 인해 B+ 트리는 범위 검색 시에 아주 좋은 성능을 낸다.




[그림 5-1-2] B+ 트리 구조

B+ 트리의 탐색 성능은 전체 로우 수보다 이분화해 가는 깊이에 더 많은 영향을 받는다. 그러므로 한 번 이분화를 했을 때 얼마나 처리할 범위를 줄여 주었느냐가 처리할 깊이에 영향을 주게 된다. 그 런데 스캔은 랜덤 액세스보다 비용 측면에서 더 유리하기 때문에 몇 번의 이분화를 통해 어느 정도 처리 범위가 줄어들었다면 더 이상 이분화하지 않고 스캔을 통해 비교하는 것이 더 유리하다. 이러한 원리를 바탕으로 DBMS들은 몇 개의 칼럼까지만 이분화를 수행하고 그 이상의 칼럼은 인덱스 로우 의 스캔을 통해 비교한다.

B+ 트리는 삽입과 삭제 시의 성능이 노드의 분할과 머지의 횟수에 따라 결정되기 때문에 노드가 분 할될 확률을 줄이기 위해"각 노드는 최소한 반 이상 차 있어야 한다"라는 제약 조건을 가지고 있다.

그런데 데이터 량이 대용량화되면서 이러한 제약 조건만으로는 분할을 최소화하는 데 한계가 있어 이 제약 조건을"각 노드는 최소한 2/3 이상 차 있어야 한다"라는 제약 조건으로 변경되었다. 이러한 제약 조건을 가진 트리를 B* 트리 라고 한다. 현재 대부분의 상용 DBMS에서 사용하고 있는 B 트리 인덱스는 B+ 트리와 B* 트리의 구조와 특징을 모두 포함하고 있는 인덱스를 지원한다.

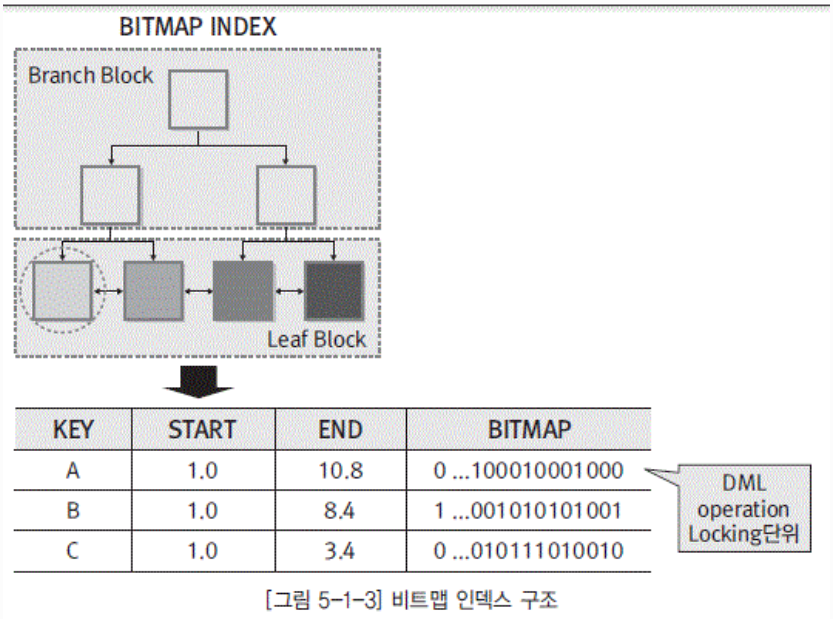
B+ 트리에서 노드를 가득 채우는 방법은 데이터 삽입 시 발생하는 분할의 횟수를 줄여 삽입 시의 성능은 향상되지만 삭

제 시에는 머지가 발생할 가능성이 더 높아지기 때문에 삭제가 빈번하게 발생 하는 경우에는 더 치명적인 문제를 야기한다. 이러한 부분을 피하는 방법은 삭제 시에 제약 조건을 완화하는 방법이다. 실제로 대부분의 상용 DBMS에서는 삭제 시에 발생할 수 있는 머지에 대한 성 능상의 저하를 막기 위해 삭제 시에 머지 자체가 발생하지 않도록 하는 방법을 사용하고 있다. 예를 들어, 오라클의 경우도 전체 데이터를 전부 삭제하더라도 머지가 발생하지 않고 삽입 시에 확장된 구 조와 스토리지를 계속 유지하도록 설계되어 있다. 때문에 삭제가 빈번하게 발생하는 테이블의 경우 인덱스 사이즈가 지속적으로 증가할 수 있으므로 저장 공간 낭비를 막기 위해서는 인덱스를 주기적 으로 재생성해 주어야 한다ass="text">해쉬 인덱스의 기본 개념은 키 값에 해쉬 함수를 적용하여 해당 키 값에 대응하는 버켓을 식별하고 탐색한다는 것이다. 버켓은 처음에는 하나의 기본 페이지로 구성되지만 해당 버켓에 새로운 데이터 엔트리를 위한 저장 공간이 없을 때에는 새로운 오버플로우 페이지를 하나 할당하고 그 페이지에 데 이터를 삽입하여 해당 버켓과 연결한다. 그런데 오버플로우 체인이 길어지면 버켓을 탐색하는 성능 이 저하되므로 초기 생성 시 80% 정도만 채우고, 또한 주기적으로 다시 해싱해 오버플로우 페이지를 제거한다.

### 비트맵 인덱스

인덱스의 목적은 주어진 키 값을 포함하고 있는 로우에 대한 주소 정보를 제공하는 것이다. B+ 트 리 인덱스의 경우는 각각의 키 값에 Rowid 리스트를 저장함으로써 이러한 목적을 달성하고 있지만, 비트맵 인덱스의 경우는 B+ 트리 인덱스와는 약간 다른 방식으로 로우에 대한 주소 정보를 제공한다.

비트맵 인덱스는'0'혹은'1'로 이루어진 비트맵만 가지고 있기 때문에 이 정보로부터 실제 로우 가 저장되어 있는 물리적인 위치를 아는 것은 불가능해 보인다. 그러나 비트맵에서 비트의 위치는 테 이블에서 로우의 상대적인 위치를 의미하므로 해당 테이블이 시작되는 물리적인 주소만 알면 실제 로우의 물리적인 위치를 계산해 낼 수 있다. DBMS는 내부적으로 비트의 위치를 가지고 로우의 물리적인 위치를 계산하는 함수를 지원한다.




[그림 5-1-3] 비트맵 인덱스 구조

비트맵 인덱스는 다중 조건을 만족하는 튜플의 개수를 계산하는 데 아주 적합하다. 이러한 현상은 비트맵 인덱스가 B+ 트리 인덱스에 비해 저장 공간이 획기적으로 줄고 키 값의 비교를 비트 연산으 로 처리하므로 비교 연산도 획기적으로 줄어들기 때문이다. 그 외에 비트맵 인덱스가 가지는 장점으 로 압축을 들 수 있다. 비트맵은 키 값 단위로 생성되므로 동일한 값이 반복될 확률이 높아 압축 효율 도 매우 좋아진다.

[표 5-1-6] B-tree와 Bitmap 구조 비교

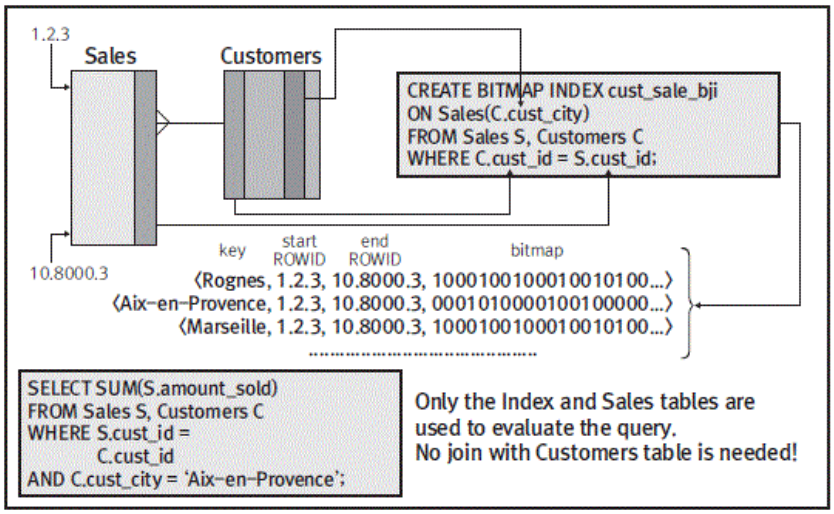
| 구분 | B-tree | Bitmap |
|---|---|---|
| 구조 특징 | Root block, branch block, leaf block으로 구성되며, 인덱스 깊이를 동일하게 유지하는 트리 구조 | 키 값이 가질 수 있는 각 값에 대해 하나의 비트맵을 구성 |
| 사용 환경 | OLTP | DW, Mart 등 |
| 검색 속도 | 처리 범위가 좁은 데이터 검색 시 유리함 | 다중 조건을 만족하는 데이터 검색 시에 유리함(특히, 비정형 쿼리) |
| 분포도 | 데이터 분포도가 좋은 칼럼에 적합 | 데이터 분포도가 나쁜 칼럼에 적합 |
| 장점 | 입력, 수정, 삭제가 용이함 | 비트 연산으로 OR 연산, NULL 값 비교 등이 가능함 |
| 단점 | 처리 범위가 넓을 때 수행 속도 저하 | 전체 인덱스 조정의 부하로 입력, 수정, 삭제가 어려움 |

### 함수 기반 인덱스

사원 테이블에서 이름이'MIKE'로 시작하는 자료를 찾는 전형적인 방법은 Where절에 “like name='MIKE'”처럼 사용하는 것이다. NAME이라는 칼럼에 인덱스가 생성되어 있다면 옵티마이저 는 해당 인덱스를 사용해 자료를 찾을 것이다. 그런

데 “like Upper(name) = 'MIKE'”처럼 조건절을 기술한다면 인덱스 칼럼에 변형이 일어나므로 옵티마이저는 인덱스를 사용하지 않고 Full Table Scan을 하게 된다.

그러나 함수 기반 인덱스(Function-based indexes)는 함수(function)나 수식(expression)으 로 계산된 결과에 대해 B+ 트리 인덱스나 Bit-Map Index를 생성, 사용할 수 있는 기능을 제공한 다. 함수 기반 인덱스에서 사용되는 함수는 산술식(Arithmetic expression), PL/SQL Function, SQL Function, Package, C callout이 가능하지만 동일한 입력 값에 대해 시간의 흐름에 결과 값 이 변경되는 함수에는 적용이 불가능하다. 또한 Object Type도 해당 칼럼에 정의된 method에 대 해 함수 기반 인덱스의 생성이 가능하지만 LOB, REF Type, Nested Table Column 또는 이들을 포함하고 있는 Object type에 대해서는 함수 기반 인덱스의 생성이 불가능하다.




[그림 5-1-4] 비트맵 조인 인덱스 구조

### 비트맵 조인 인덱스

비트맵 조인 인덱스는 단일 객체로만 구성되었던 기존 인덱스와 달리 여러 객체의 구성 요소(칼럼) 로 인덱스 생성이 이뤄지므로 기존의 인덱스와 테이블 액세스 방법과는 구조가 다르다.

비트맵 조인 인덱스의 물리적인 구조는 비트맵 인덱스와 완전히 동일하다. 단지 인덱스의 구성 칼 럼이 베이스 테이블의 칼럼 값이 아니라 조인된 테이블의 칼럼 값이라는 차이만 존재할 뿐이다. 즉, 비트맵 조인 인덱스는 기본 테이블을 기준으로 조인된 결과를 기존의 비트맵 인덱스와 같은 구조로 저장하여 카디날리티가 낮은 데이터의 액세스에 좋은 성능을 얻을 수 있게 하는 방법이다.

### 도메인 인덱스

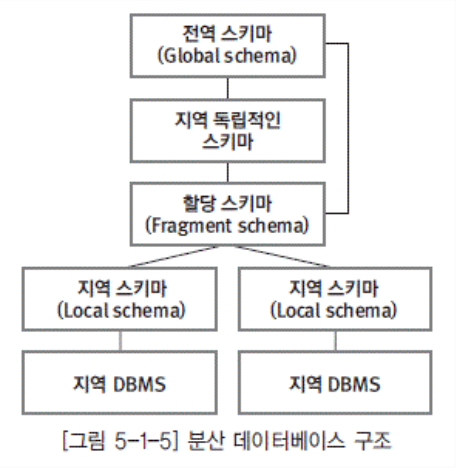
도메인 인덱스는 오라클8i에서부터 새롭게 도입된 개념으로 개발자가 자신이 원하는 인덱스 타입 을 생성할 수 있게 함으로써 인덱스 시스템에 많은 확장을 가져다주었다.

즉, 데이터베이스에는 아직 존재하지도 않는 새로운 구조의 인덱스 타입을 자신이 스스로 정의하 여 DBMS에서 지원하는 인덱스처럼 사용할 수 있다. InterMedia text index나 오라클 9i부터 지원 되는 Context 타입 인덱스와 Catalog 타입 인덱스 등은 모두 비정형 테스트를 빠르게 검색하려고 새롭게 도입된 도메인 인덱스

## 분산 데이터베이스 개요

하나의 논리적 데이터베이스가 네트워크상에서 여러 컴퓨터에 물리적으로 분산되어 있지만 사용 자가 하나의 데이터베이스처럼 인식할 수 있도록 논리적으로 통합되어 공유되는 데이터베이스를 분 산 데이터베이스라 한다.

분산 데이터베이스는 데이터베이스를 하나 이상의 장소에 분산시켰기 때문에 이들을 하나의 논리적 인 데이터베이스로 인식하려면 지역 데이터베이스 관리 시스템 이외에 각 장소의 정보를 교환하고 관 리해 주는 시스템이 필요한데, 이러한 역할을 수행하는 시스템이 분산 데이터베이스 관리 시스템이다.



[그림 5-1-5] 분산 데이터베이스 구조

분산 데이터베이스의 장점은 자신의 데이터를 지역적으로 제어하여 원격 데이터에 대한 의존도를 감소시키고 단일 서버에서 불가능한 대용량 처리가 가능하며, 기존 시스템에 서버를 추가하여 점진 적 증가가 용이하다. 또한 데이터베이스를 사용하는 중에 한 사이트가 고장 나더라도 고장 난 사이트 의 데이터만 사용하지 못하게 됨으로써 신뢰도와 가용성이 향상된다.

분산 데이터베이스는 중앙 집중 방식에 비해 이러한 장점을 가지고 있지만 분산 처리에 의해 복잡 도가 증가하여 소프트웨어 개발 비용이 증가하고, 통제 기능이 취약하고, 분산 처리에 따른 오류 발 생 가능성이 증가하게 된다. 그리고 데이터가 물리적으로 저장된 장소와 해당 지역 시스템의 상황에 따라 응답 속도가 불규칙할 수 있으며, 데이터의 무결성을 완전히 보장하기는 불가능하다.

## 분산 데이터베이스 관리 시스템

여러 개의 물리적인 데이터베이스를 논리적인 단일 데이터베이스처럼 인식하려면 사용자들이 데 이터가 물리적으로 어디에 배치되어 있는지, 또는 특정 지역 사이트에서 위치한 데이터를 어떻게 액 세스해야 하는지 알 필요가 없어야 한다. 이러한 특성을 데이터 투명성이라고 하며, 분산 데이터베이 스 관리 시스템은 다음과 같은 투명성이 제공될 수 있어야 한다.

■ 분할 투명성

분할 투명성은 사용자에게 전역 스키마가 어떻게 분할되었는지를 알려주는 역할을 한다. 즉, 사용 자가 입력한 전역 질의를 여러 개의 단편 질의로 변환해 주기 때문에 사용자는 전역 스키마가 어떻 게 분할되어있는지를 알 필요가 없게 된

다. 분할의 방법은 수직 분할과 수평 분할이 있다. 수직 분 할은 한 릴레이션을 속성들의 부분 집합으로 이루어진 릴레이션들로 나누고, 수평 분할은 튜플들 의 집합으로 이루어진 릴레이션들로 나누는 것을 말한다.

■ 위치 투명성

위치 투명성은 사용자나 애플리케이션에서 어떤 작업을 수행하기 위해 분산 데이터베이스상에 존 재하는 어떠한 데이터의 물리적인 위치도 알 필요가 없어야 한다는 것이다. 즉, 사용자는 데이터의 위치나 입력 시스템의 위치와 무관하게 동일한 명령을 사용하여 데이터에 접근할 수 있어야 한다.

분산 데이터베이스 관리 시스템은 위치 투명성을 보장하기 위해 분산 데이터베이스에 저장되어 있 는 모든 데이터에 대한 메타 데이터와 위치 정보를 참조하여 지역 트랜잭션은 지역 데이터베이스 에 처리를 일임하고, 전역 트랜잭션은 다른 지역의 데이터베이스에 처리를 일임하여 결과를 통보 받는다.

■ 중복 투명성

중복 투명성은 어떤 데이터가 중복되었는지, 또는 어디에 중복 데이터를 보관하고 있는지 사용자 가 알 필요가 없어야 한다는 것이다. 즉, 사용자는 자신이 사용하고 있는 데이터가 논리적으로 유 일하다고 생각할 수 있어야 한다.

중복 투명성과 위치 투명성이 보장될 경우 분산 데이터베이스 관리 시스템은 사용자의 데이터 처 리 요청을 가장 적은 시간이 걸리는 지역을 선택하여 처리하므로 수행 속도가 향상되지만 복제된 데이터에 대한 갱신을 처리하려면 복제된 모든 지역의 데이터를 갱신하여야 하므로 데이터 무결성 을 보장하기가 더 어려워진다.

■ 장해 투명성

장애 투명성은 데이터베이스가 분산되어 있는 각 컴퓨터 시스템이나 네트워크에 장애가 발생하더 라도 데이터의 무결성이 보장되어야 한다는 것이다. 즉, 분산 데이터베이스는 그 구성 요소의 장애 에 무관하게 트랜잭션의 원자성이 유지되어야 한다.

■ 병행 투명성

병행 투명성은 다수 트랜잭션이 동시에 수행되어도 결과의 일관성이 유지되어야 하는 성질을 말한 다. 분산 데이터베이스 관리 시스템은 분산 트랜잭션의 일관성을 유지하기 위해 잠금(Locking)과 타임스탬프(Timestamp)의 두 가지 방법을 주로 사용한다.

## 분산 설계 전략

분산 데이터베이스의 주요 목적은 사용자 및 애플리케이션 프로그램이 원격지 데이터와 지역 데이 터에 접근할 수 있도록 하는 것이다. 그런데 분산 데이터베이스는 위에서 언급한 것처럼 잘못 설계할 경우 복잡성과 비용이 증가하고 불규칙한 응답 속도, 데이터 무결성에 대한 위협이 발생할 수 있으므로 분산 데이터베이스 설계 시에는 분산 환경의 형태와 분산 데이터베이스의 구조가 현재 구현된 시스템에 적합한지를 고려해 분산 데이터베이스의 장점은 향상시키고 단점은 최소화될 수 있도록 설계해야 한다.

기본적인 전략으로는 수직 분할, 수평 분할, 복제의 세 가지 방법을 사용하지만 데이터의 분할 및 복제, 지역 복제본의 갱신 주기, 데이터베이스의 유지 방식에 따라 다음과 같은 여러 가지 전략으로 나눌 수 있다.

- 중앙 집중형 데이터베이스처럼 한 컴퓨터에서만 데이터베이스를 관리하고 여러 지역에서 접근할 수 있도록 하는 방식
- 지역 데이터베이스에 데이터를 복제하고 실시간으로 복제본을 갱신하는 방식
- 지역 데이터베이스에 데이터를 복제하고 주기적으로 복제본을 갱신하는 방식
- 데이터 분할 시 전체 지역 데이터베이스를 하나의 논리적 데이터베이스로 유지하는 방식
- 데이터 분할 시 각각의 지역 데이터베이스들을 독립된 데이터베이스로 유지하는 방식

## 분산 설계 방식

분산 설계는 먼저 전역 릴레이션을 논리적으로 중복되지 않는 작은 단위로 나누고, 이를 여러 노드에 할당하는 순서로 진행된다. 이때 할당의 기준이 되는 작은 단위를 분할(Fragment)이라고 하며, 분할 스키마에 의하여 전역 릴레이션과의 사상(mapping) 관계가 관리된다.

분할이 할당될 때에는 하나의 분할이 분산 네트워크상에 있는 하나의 노드에만 존재하도록 하여 단일 복사본(single copy)을 유지하거나 복제(replication)하여 유지할 수 있다. 이처럼 분할은 분산 데이터베이스에 중복 혹은 비중복되어 존재하게 되는데, 할당 스키마에 의해 분할이 어느 노드에 위치하는지를 정의한다.

예를 들어, 고객 테이블이 고객 관리 서버에 존재하고 다른 서버에는 고객 테이블이 존재하지 않게 설계하거나 사용하는 모든 서버에 고객 테이블이 복제되도록 설계할 수 있다.

### 테이블 위치 분산

테??는 것이다. 테이블이 전역적으로 중복되지 않 는다. 예를 들어 고객 테이블은 고객 관리 서버에 존재하고 다른 서버에는 고객 테이블이 존재하지 않게 설계하는 것이다. 이를 위해서는 각각의 테이블마다 위치가 다르게 지정을 위해 테이블마다 존재할 서버를 결정해야 한다.

### 분할(Fragmentation)

분할은 앞에서 기술한 바와 같이 분할 방법을 먼저 결정하고 결정된 분할을 할당하는 개념으로, 완 전성, 재구성, 상호 중첩 배제의 3가지 분할 규칙이 준수되어야 한다.

■ 완전성(Completeness)

완전성은 분할 시에 전역 릴레이션 내의 모든 데이터가 어떠한 손실도 없이 분할로 사상 (mapping)되어야 한다는 것으로 분할의 대상이 전체 데이터를 대상으로 이뤄져야 함을 의미한다.

■ 재구성(Reconstruction)

분할은 관계 연산을 사용하여 원래의 전역 릴레이션으로 재구성이 가능하여야 한다.

■ 상호 중첩 배제(Dis-jointness)

상호 중첩 배제는 분할 시에 하나의 분할에 속한 데이터 항목이 다른 분할의 데이터 항목에 속하지 않아야 한다는 의미로, 수평 분할은 각각의 분할에 속한 튜플들이 서로 중첩되지 않아야 하고, 수 직 분할은 식별자를 제외한 속성들이 중복되지 않아야 한다.
분할 방법은 로우(Row) 단위로 분리하는 수평 분할 방법과 테이블 칼럼을 분할하는 수직 분할이 있다.

■ 수평 분할

수평 분할은 특정 속성의 값을 기준으로 분할을 수행한다. 속성 값에 의해 서버 간 데이터가 상호 배타적으로 존재하므로 분할을 통합하여도 식별자의 중복이 발생하지 않는다.

■ 수직 분할

수직 분할은 속성을 기준으로 분할을 수행한다. 튜플을 기준으로 분리하지 않았기 때문에 각 서버 에는 동일한 식별자를 가진 튜플이 중복되어 존재한다. 동일한 식별자 구조를 가지므로 식별자를 이용하여 분할 전 릴레이션을 재구성할 수 있다.

### 할당(Allocation)

할당은 동일한 분할을 복수 서버에 생성하는 분산 방법으로, 분할의 중복이 존재하는 할당 방법과 중복이 존재하지 않는 할당 방법이 있다. 중복되지 않는 할당 방식은 최적의 노드를 선택하여 분할이 분산 데이터베이스상에서 하나의 노드에서만 존재하도록 하는 것이다. 그런데 애플리케이션은 릴레 이션을 상호 배타적인 분할로 분리하기 어려운 요구 사항을 가지는 경우가 대부분이므로 분할 간의 의존성을 무시한 이 방법은 비용이 증가하고 성능상 문제를 야기할 수도 있다.

분할이 중복되지 않아 비용이 증가하고 성능이 저하된다면 각각의 노드에 분할을 중복하여 할당하 는 방법을 고려하여야 한다. 이때 데이터 복제는 실시간 처리의 필요성이 없는 경우 야간 일괄 복제 방식을 사용하도록 한다.

복제는 분할의 일부만 복제하는 부분 복제와 데이터베이스 전체를 복제하는 완전 복제 두 가지 방 식으로 구성이 가능하다.

■ 부분 복제

전역 서버 테이블의 일부 데이터를 지역 서버에 복제하는 방식이다.

■ 광역 복제

전역 서버 테이블의 전체를 지역 서버에 복제하는 방식이다.

### 데이터 통합

분산 설계는 최적의 비용으로 지역적으로 분산된 데이터베이스에서 데이터 일관성을 유지할 수 있 는가에서 출발했다. 현재는 데이터 통합이라는 관점에서 분산아키텍처의 단점을 보완하고 정보의 적 시성과 실시간 데이터 교환이라는 목적으로 통합 아키텍처를 구축한다. 통합 방식은 DW를 이용하 는 방법과 EAI(Enterprise Application Integration)를 이용하는 방법, 두 가지를 혼합해 통합하 는 방법이 있다.

데이터베이스 보안이란 데이터베이스 정보가 비인가자에 의해 노출, 변조, 파괴되는 것을 막는 것 이다. 사용자가 원하는 작업을 하려면 필요한 자원에 대한 허가가 있어야 한다. “누가, 어느 데이터 에 대해, 무슨 연산을 수행할 수 있도록 허락받았는가?”를 명시적으로 정의하고, 정의된 내용을 구현 하는 과정이 데이터베이스 보안 설계이다.

보안 설계의 주요 목표는 권한이 없는 사용자에게 정보가 노출되는 것을 방지하고, 권한이 있는 사 용자는 데이터에 접근하여 수정할 수 있도록 보장하는 것이다. 이러한 목표를 달성하려면 일관성 있 는 보안 정책이 수립되어야 하며, 이러한 보안 정책은 보안 모델을 통해 운영체제와 DBMS에 의하 여 보장되어야 한다.

일반적으로 데이터베이스 관리 시스템은 데이터 보안을 유지하기 위해 접근 통제 기능, 보안 규칙, 뷰(View), 암호화 등과 같은 다양한 보안 모델의 기능을 제공하고 있다.

다음은 일관성 있는 데이터베이스 보안 정책을 수립하기 위해 요구되는 사항들이다.

■ 자원에 접근하는 사용자 식별 및 인증 – 사용자, 비밀번호, 사용자 그룹

자원에 접근하는 사용자는 인증을 통해 그 실체가 보장되어야 한다. 누군가가 자신의 실체를 인증 받았다면 다른 임의의 사용자가 그 사람의 인증을 사용할 수 없도록 보장되어야 한다.

■ 보안 규칙 또는 권한 규칙에 대한 정의

보안 규칙은 어떤 사용자가 접근할 수 있는 데이터와 그 데이터에 대하여 허용된 행위나 제한 조건을 기록하는 것으로, 보안 모델을 통해 구현된다.

■ 사용자의 접근 요청에 대한 보안 규칙 검사 구현 – 보안 관리 시스템 구현

보안 규칙에 대한 검사 구현은 운영체제와 DBMS를 활용하여 보장되어야 한다.

## 접근 통제 기능

접근 통제는 보안 시스템의 중요한 기능적 요구 사항 중 하나로, 임의의 사용자가 어떤 데이터에 접근하고자 할 때 접근을 요구하는 사용자를 식별하고, 사용자의 요구가 정상적인 것인지를 확인, 기 록하고 보안 정책에 근거하여 접근을 승인하거나 거부함으로써 비인가자의 불법적인 자원 접근 및 파괴를 예방하는 보안 관리의 모든 행위를 의미한다.

데이터베이스 보안을 위한 접근 통제는 크게 강제적 접근 통제와 임의적 접근 통제 두 가지로 분류한다.

■ 임의적 접근 통제(DAC, Discretionary Access Control)

임의적 접근 통제는 사용자의 신원에 근거를 두고 권한을 부여하고 취소하는 메커니즘을 기반으로 하고 있다. 여기서 권한은 사용자가 어떤 객체에 대해 특정 행위를 할 수 있도록 허용하는 것으로, 임의적 접근 통제에서 객체를 생성한 사용자는 그 객체에 대해 적용 가능한 모든 권한을 가지게 되 고, 또한 이 권한들을 다른 사용자에게 허가할 수 있는 허가 옵션도 가지게 된다. 그 외 사용자는 어떤 객체에 대해 데이터 조작 행위를 하려면 DBMS로부터 권한을 부여 받아야 한다.

데이터베이스 관리 시스템은 임의적 접근 통제를 지원하기 위해 SQL의 GRANT와 REVOKE 명령어를 사용한다. GRANT는 사용자에게 객체에 대한 권한을 부여하기 위해 사용하는 명령어이 고, REVOKE는 부여한 권한을 취소하기 위해 사용하는 명령어이다.

다음은 GRANT에 대한 구문 형식을 표현한 것이다.

다음은 REVOKE에 대한 구문 형식을 표현한 것이다.

임의적 접근 통제는 권한 관리에 효과적이지만 통제의 기준이 사용자의 신분에 근거를 두고 있기 때문에 다른 사람의 신분을 사용하여 불법적인 접근이 이루어진다면 접근 통제 본래의 기능에 중 대한 결함이 발생한다. 그리고 트로이 목마 공격에 취약하다는 문제점을 가지고 있다. 그래서 보안 등급에 따라 데이터와 사용자를 분류하는 부가적인 보안 정책이 고려되어야 한다.

■ 강제적 접근 통제(MAC, Mandatory Access Control)

강제적 접근 통제는 주체와 객체를 보안 등급 중 하나로 분류하고, 주체가 자신보다 보안 등급이 높은 객체를 읽거나 쓰는 것을 방지한다. 각 데이터베이스 객체에는 보안 분류 등급이 부여되고 사 용자마다 인가 등급을 부여하여 접근을 통제하는 것이다. 읽기는 사용자의 등급이 접근하는 데이 터 객체의 등급과 같거나 높은 경우에만 허용된다. 수정 및 등록은 사용자의 등급이 기록하고자 하 는 데이터 객체의 등급과 같은 경우에만 허용한다. 이는 높은 등급 데이터가 사용자에 의해 의도적 으로 낮은 등급 데이터로 쓰여지거나 복사되는 것을 방지하기 위한 것이다.

## 보안 모델

보안 모델이란 보안 정책을 실제로 구현하기 위한 이론적인 모델로 군사적 목적으로 개발된 기밀 성 모델, 데이터 일관성 유지에 중점을 둔 무결성 모델, 접근 통제 메커니즘에 기반을 둔 접근 통제 모델 등이 있다.

### 접근 통제 행렬(Access control matrix)

접근 통제 행렬은 임의적 접근 통제를 위한 보안 모델로, 행은 주체를 나타내고, 열은 객체를 나타내며, 행과 열은 주체와 객체가 가지는 권한의 유형을 나타낸다.

■ 주체

데이터베이스에 접근할 수 있는 조직의 개체로 일반적으로 객체에 대하여 접근을 시도하는 사용자를 의미한다.

■ 객체

보호되고 접근이 통제되어야 하는 데이터베이스의 개체로, 테이블, 칼럼, 뷰, 프로그램, 논리적인 정보의 단위 등이 될 수 있다.

■ 규칙

주체가 객체에 대하여 수행하는 데이터베이스의 조작으로, 입력, 수정, 삭제, 읽기와 객체의 생성과 파괴 등이 존재한다. 그리고 객체가 프로그램으로 확장된다면 실행, 출력 등의 작업 유형을 정의할 수 있다.

[표 5-1-7] 접근 제어 매트릭스

| | 사원 | 급여 | 상여 | 사원평가 |
|---|---|---|---|---|
| 김명민 | ALL | C/R/U/D | R | - |
| 김아중 | ALL | ALL | ALL | R |
| 하지원 | R | R | R | ALL |

### 기밀성 모델

기밀성 모델은 군사용 보안 구조의 요구 사항을 충족시키기 위하여 정보의 불법적인 파괴나 변조 보다는 기밀성(Confidentiality) 유지에 초점을 둔 최초의 수학적인 모델로, 다른 보안 모델과의 비 교를 위한 참조 모델로서 이용되지만 정보의 무결성이 비밀성보다 중요하게 요구되는 상용 환경에 적용하기에는 부적합한 모델이다.

이 모델은 각 주체(사용자, 계정, 프로그램)와 객체(릴레이션, 튜플, 속성, 뷰, 연산)를 보안 등급인 극비(Top Secret), 비밀(Secret), 일반(Confidential), 미분류(Unclassified) 중의 하나로 분류하 며, 데이터 접근에 대해 주체/객체의 등급을 기반으로 다음과 같은 제약 조건을 준수하여야 한다.

- 단순 보안 규칙 : 주체는 자신보다 높은 등급의 객체를 읽을 수 없다.
  인가 받은 비밀 등급이 낮은 주체는 비밀 등급이 높은 객체를 읽어서는 안된다는 정책으로 BLP는 기밀성 보장을 위한 보안 모델이므로 기밀성 보장을 위해 낮은 비밀 등급을 가진 주체가 높은 비밀 등급의 객체에 접근하는 것은 당연히 통제되어야 한다.
- *(스타)-무결성 규칙 : 주체는 자신보다 낮은 등급의 객체에 정보를 쓸 수 없다. 높은 비밀 등급을 인가 받은 주체가 자신이 접근 가능한 비밀 정보를 낮은 등급으로 복사하여 정보를 유출시키는 행 위를 금지하여 정보의 기밀성(Confidentiality)을 보호하고자 하는 정책이다.
- 강한 *(스타) 보안 규칙 : 주체는 자신과 등급이 다른 객체에 대하여 읽거나 쓸 수 없다.

### 무결성 모델

정보의 일방향 흐름 통제를 이용하여 정보의 비밀성을 제공하는 기밀성 모델에서 발생하는 정보의 부당한 변경 문제를 해결하기 위해 개발된 무결성 기반의 보안 모델로, 기밀성 모델처럼 주체와 객체 의 보안 등급으로 표현되며, 다음과 같은 제약 조건을 준수하여야 한다.

- 단순 보안 규칙 : 주체는 자신보다 낮은 등급의 객체를 읽을 수 없다.
- *(스타)-무결성 규칙 : 주체는 자신보다 높은 등급의 객체에 정보를 쓸 수 없다.

## 접근 통제 정책

접근 통제 정책은 어떤 주체(Who)가 언제(When), 어떤 위치에서(Where), 어떤 객체(What)에 대하여, 어떤 행위(How)를 하도록 허용할 것인지 접근 통제의 원칙을 정의하는 것으로, 신분-기반 정책, 규칙-기반 정책, 역할-기반 정책 등이 있다.

- 신분-기반 정책

개인 또는 그들이 속해 있는 그룹들의 신분에 근거하여 객체에 대한 접근을 제한하는 방법으로, 한 사용자가 하나의 객체에 대하여 허가를 부여받는 IBP(Individual-Based Policy)와 다수의 사용자가 하나의 객체에 대하여 동일한 허가를 부여받는 GBP(Group-Based Policy)로 표현할 수 있다(예 : 경영진, 관리자, 감사, CEO 등).

- 규칙-기반 정책

강제적 접근 통제와 동일한 개념으로, 주체가 갖는 권한에 근거하여 객체에 대한 접근을 제한한다. 이 정책은 사용자 및 객체별로 부여된 기밀 분류에 따른 정책(MLP)과 조직 내의 부서별로 구분된 기밀 허가에 따르는 정책(CBP)으로 표현될 수 있다.

- 역할-기반 정책

역할-기반 정책은 GBP의 한 변형된 형태로 생각할 수 있다. 즉, 정보에 대한 사용자의 접근이 개별적인 신분이 아니라 개인의 직무 또는 직책에 따라서 결정된다(예 : 인사 담당자, 출고 담당자, DBA 등).

## 접근 통제 매커니즘

사용자 통제를 기술적으로 구현하는 방법은 패스워드, 암호화, 접근 통제 목록 적용, 제한된 사용자 인터페이스, 보안 등급 등의 방법이 이용된다.

- 패스워드

어떤 통신 주체가 자신이라고 주장하는 것을 증명하려고 사용하는 인증 방법 중 하나로, 시스템을 액세스할 때 패스워드를 제시하면 인증 시스템이 보유 목록과 비교하여 사용자의 신분을 확인하는 기법이다.

- 암호화

암호화란 인간에 의해 해석될 수 없는 형태로 데이터를 변형시키는 것으로, 통신망을 통해 중요한 데이터를 전송할 때 무단 도용을 방지하기 위해 주로 사용된다. 암호화 시스템은 데이터를 암호화하는 프로그램과 암호화된 데이터를 풀 수 있는 복호화 프로그램으로 구성되며, 데이터에 접근하기 위해서는 복호화 키를 소유하여야 한다.

- 접근 통제 목록(Access Control List)

접근 통제 행렬의 문제점을 해결하기 위해 객체를 기준으로 접근 통제 정보를 저장하는 방식으로, 어떤 사용자들이 객체에 대하여 어떤 행위를 할 수 있는지를 나타낸다. 접근 통제 목록은 주체의 수가 많아지면 관리가 어려워지므로 대부분의 운영체제에서는 객체에 대한 접근 권한을 동일한 권한을 가진 주체들의 그룹에 부여함으로써 접근 통제 목록의 관리를 용이하게 하고 있다.

- 능력 리스트(Capability List)

접근 통제 행렬의 문제점을 해결하기 위해 주체가 접근할 수 있는 객체와 접근 권한을 주체에 저장하는 방식이다.
능력 리스트는 비교적 객체가 적을 경우에 적합하다. 그러나 주어진 객체에 접근할 수 있는 사용자들을 파악하는 데는 많은 시간이 소요된다.

- 보안 등급

보안 등급은 주체나 객체 등에 부여된 보안 속성의 집합으로, 다단계 접근 통제 정책을 지원하기위해 각 주체와 객체를 보안 등급 중 하나로 분류하고, 데이터 접근 요청을 처리할 때 주체/객체의 보안 등급을 기반으로 접근 승인 여부를 결정하는 방식이다.

- 통합 정보 매커니즘

과거에는 접근 통제 목록, 능력 리스트, 보안 등급과 같은 매커니즘은 접근 통제 정책을 구현하는 별도의 방법으로 고려되어 왔다. 그러나 최근에는 보안 요구의 다양화와 복잡성으로 인해 적어도 두 가지 이상의 복합된 특성으로 구현되는 추세이다.

## 접근 통제 조건

접근 통제 매커니즘의 취약점을 보완하기 위해 접근 통제 정책에 적용할 수 있는 조건들로, 어떤 임계 값, 사용자 간의 동의, 사용자의 특정 위치 및 시간 등을 지정할 수 있다.

- 값 종속 통제(Value-Dependent Control)

대부분의 통제 정책들은 객체에 저장된 데이터의 값에 상관없이 동일한 접근 통제 허가를 부여하 지만 객체에 저장된 값에 따라 접근 통제 허가가 다양화되어야 하는 경우도 많이 발생한다. 예를 들어, 계약 금액에 따라 기밀 수준이 다르다면 특정 임계 값이나 계약 금액에 따라 보안 등급을 설 정하고 해당 보안 등급을 가진 사용자만 접근을 허용해야 한다.

- 다중 사용자 통제(Multi-User Control)

지정된 객체에 대해 다수의 사용자가 연합하여 접근을 요청할 경우 접근 통제를 지원하기 위한 수 단이 제공되어야 한다. 예를 들어, 명시된 두 개인이 동의할 것을 요구하는 경우나 하나의 그룹에 서 다수결에 의하여 접근 통제를 수행할 필요가 있을 수 있다.

- 컨텍스트 기반 통제(Context-Based Control)

이 통제 방법은 특정 시간, 네트워크 주소 등 확인이 가능한 접근 경로나 위치, 인증 수준 등과 같 은 외부적인 요소에 의존하여 객체의 접근을 제어하는 방법으로, 다른 보안 정책들과 결합하여 보 안 시스템의 취약점을 보완하기 위해 주로 사용된다. 예를 들어, 업무 시간대인 월요일에서 금요일 까지 09:00 - 18:00 시간대에만 데이터 접근을 허용하는 것과 같다.

## 감사 추적

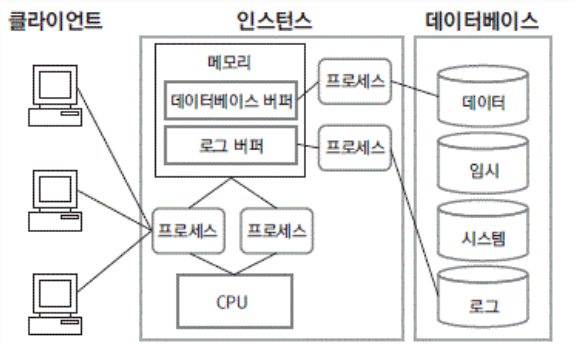
애플리케이션 및 사용자가 데이터베이스에 접근하여 수행한 모든 활동을 일련의 기록으로 남기는 기능으로, 오류로 인해 데이터베이스가 파괴되었을 때 복구하기 위한 중요한 정보로 사용하거나 데 이터베이스에 대한 부당한 조작을 파악하기 위한 수단으로 사용되기도 한다. 즉, 감사 추적을 실시하 므로 개인 책임성을 보조하고, 문제가 발생했을 때 사건의 재구성이 가능하게 된다. 사전에 침임 탐 지를 확인한다거나 사후 문제를 분석하여 보안을 강화하기 위해 필요하다.

감사 추적 시에는 사용자 실행 프로그램, 사용 클라이언트, 사용자, 날짜 및 시간, 접근하는 데이 터의 이전 값 및 이후 값 등을 저장한다.

데이터베이스 관리 시스템(DBMS) | 데이터 액세스 | 트랜잭션 | 백업 및 복구

## 개념적 데이터베이스 관리 시스템 아키텍처

DBMS 서버는 인스턴스(Instance)와 데이터베이스(Database)로 구성된다. 인스턴스는 메모리 (Memory) 부문과 프로세스(Process) 부문으로 구성된다. 그 외 데이터베이스의 기동과 종료를 위하여 DBMS 환경을 정의한 매개변수 파일과 파일 목록(데이터 파일, 로그 파일)을 기록한 제어 파일이 있다.




[그림 5-2-1] 데이터베이스 관리 시스템 구조

## 데이터베이스 서버 시작과 종료

데이터베이스를 사용하려면 권한을 가진 데이터베이스 관리자가 DBMS 인스턴스를 시작해야 한 다. 인스턴스 시작은 매개변수 파일을 읽어야 한다. 매개변수 파일은 인스턴스와 데이터베이스에 대 한 구성 매개변수(초기화 매개변수)의 목록이 있는 텍스트 파일이다. 인스턴스 구성 매개변수를 특정 값으로 설정하여 인스턴스의 메모리와 프로세스 설정을 초기화한다.

대부분의 초기화 매개변수는 다음 그룹 중 하나이다.

- 파일과 같은 항목에 이름을 지정하는 매개변수
- 최대 값과 같은 한계를 설정하는 매개변수
- 메모리 크기와 같은 용량에 영향을 주는 매개변수(가변 매개변수라고 함)
- 인스턴스를 시작할 데이터베이스 이름
- 로그 파일의 처리 방법
- 데이터베이스 제어 파일의 이름과 위치

### 데이터베이스 서버 시작

데이터베이스 서버의 기동은 인스턴스 시작, 데이터베이스 마운트(Mount), 데이터베이스 오픈(Open)순으로 진행된다.

■ 인스턴스 시작

매개변수 파일을 읽어 초기화 매개변수 값을 결정하고 데이터베이스 정보를 위해 사용되는 메모리나. 데이터베이스 서버 종료 데이터베이스 서버 종료는 데이터베이스 닫기, 마운트 해제, 인스턴스 종료 순으로 진행된다.

■ 데이터베이스 닫기

데이터베이스를 닫으면 메모리에 있는 모든 데이터베이스 데이터와 로그를 데이터 파일과 리두 로그 파일에 각각 기록하고 온라인 데이터 파일과 온라인 로그 파일을 닫는다. 이때부터 데이터베이스는 닫혀 있어 정상적인 작업을 수행할 수 없다. 그러나 제어 파일은 열린 상태이다.

■ 마운트 해제

데이터베이스 마운트를 해제하여 데이터베이스와 인스턴스 간의 관계를 끊고 데이터베이스의 제어 파일을 닫는다.

■ 인스턴스 종료

데이터베이스를 종료하는 마지막 단계로 인스턴스가 종료되면서 할당된 공유 영역이 메모리에서 제거되고 백그라운드 프로세스가 종료된다.

## 데이터베이스 구조

### 데이터 딕셔너리

데이터 딕셔너리는 데이터베이스의 가장 주요한 부분 중 하나이다. 연관된 데이터베이스 정보를 제공하는 읽기 전용 테이블 또는 뷰 집합이다. 데이터 딕셔너리에는 다음 정보가 있다.

- 데이터베이스의 모든 스키마 객체 정보
- 스키마 객체에 대해 할당된 영역의 사이즈와 현재 사용 중인 영역의 사이즈
- 열에 대한 기본 값
- 무결성 제약 조건에 대한 정보
- 사용자 이름, 사용자에게 부여된 권한과 역할
- 기타 일반적인 데이터베이스 정보

DBMS마다 제공되는 딕셔너리 정보의 양에는 차이가 있지만 데이터베이스의 형상을 관리하는 데 중요한 정보를 제공한다. 그 외 작업 수행 시 현재 데이터베이스 작업을 기록하는 동적 성능 테이블이 있다. 동적 성능 테이블은 세션, 록킹, SQL pool 등 다양한 정보를 제공하므로 데이터베이스 모니터링에 이용된다.

### 데이터베이스, 테이블 스페이스 및 데이터 파일

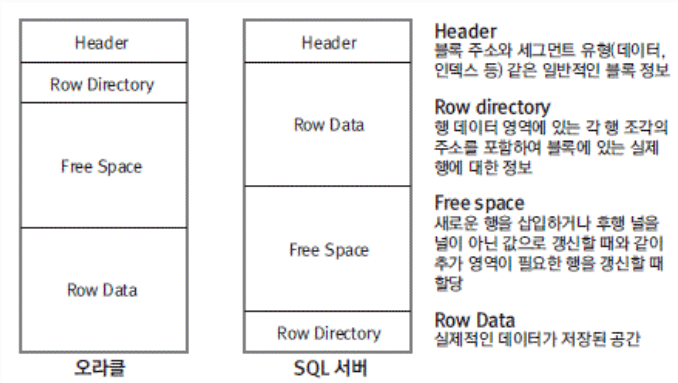
데이터는 테이블을 통해서 논리적으로는 테이블스페이스에, 물리적으로는 해당 테이블스페이스와 연관 데이터 파일에 데이터를 저장한다.

### 데이터 블록, 확장 영역 및 세그먼트 간의 관계

DBMS는 데이터베이스의 모든 데이터에 대한 논리적 데이터베이스 영역을 할당한다. 데이터베이스 영역의 할당 단위는 데이터 블록(Data block), 확장 영역(Extent), 세그먼트(Segment)이다.

■ 데이터 블록

DBMS가 데이터를 저장하는 가장 작은 단위는 데이터 블록(또는 페이지)이라 한다. 하나의 데이터 블록은 디스크상의 물리적 데이터베이스 영역의 특정 바이트 수에 해당한다. 일반적으로 2K, 4K, 8K, 16K 등 다양하다. 과거에는 데이터베이스에 단일 블록 사이즈를 이용했으나 현재는 테이블 스페이스별로 사이즈를 결정할 수 있다. 데이터베이스 용도가 OLTP, DW 등 다양해져 데이터 블 록을 사이즈가 다르게 사용할 수 있다. 데이터 블록은 1회 물리적인 디스크 입출력량을 결정하므 로 성능에 직접적인 영향이 있다. DBMS에 따라 블록의 구조도 차이가 있지만 확장 가능 영역인 Free space에 따라 데이터의 체인을 억제할 수 있는 방법을 확보해야 한다.




[그림 5-2-2] 블록 구조

■ 데이터 확장 영역(Extent)

확장 영역은 특정 유형의 정보를 저장하기 위해 할당된 몇 개의 연속적인 데이터 블록이다. 테이블 을 생성하면 DBMS는 지정된 몇 개 데이터 블록의 초기 확장 영역을 테이블 데이터 세그먼트에 할 당한다. 아직 행을 삽입하지 않았지만 초기 확장 영역에 해당되는 데이터 블록은 해당 테이블의 행에 대해 예약된 것이다. 예약된 데이터 블록이 모두 차면 새로운 증분 확장 영역을 자동으로 할당 한다. 확장 영역의 크기와 한계를 결정함으로써 불필요한 저장 공간 낭비를 줄이고 무한정 확장되 는 것을 방지할 수 있다. 확장된 영역은 데이터를 삭제하여도 확장된 영역을 반환하지 않는다. 생 성된 객체를 Drop하거나 Truncate해야 테이블스페이스로 반환된다. 아니면 직접 해제 명령 SQL 구문을 사용하여 해제할 수 있다.

■ 세그먼트(Segment)

테이블 스페이스 내에 어떤 논리적인 구조를 정의하기 위해 할당한 확장 영역의 집합으로, 테이블, 인덱스, 임시용 세그먼트가 지원된다. 각 테이블은 하나 이상의 확장 영역을 할당하여 해당 테이블 의 데이터 세그먼트를 형성하고, 각 인덱스는 하나 이상의 확장 영역을 할당하여 테이블의 인덱스 세그먼트를 형성한다.

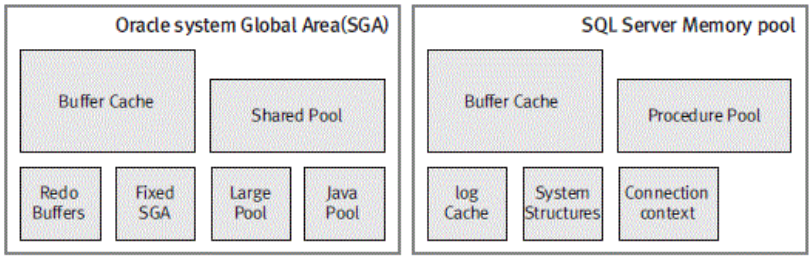
## 메모리 구조

### DBMS 정보 저장

DBMS는 다음과 같은 정보를 저장하기 위해 메모리를 사용한다.

- 실행되는 프로그램 코드
- 현재 사용하지 않더라도 접속되어 있는 세션 정보
- 프로그램 실행되?하거나 교환되는 정보(예 : 록킹 정보)
- 보조 메모리에 영구적으로 저장된 캐시 데이터

메모리는 사용 용도에 따라 소프트웨어 코드 영역, 시스템 메모리 영역, 프로그램 영역으로 나눌 수 있다. 소프트웨어 코드 영역은 수행되고 있거나 수행될 수 있는 소프트웨어의 코드를 저장하기 위 한 메모리 영역이다. 시스템 메모리 영역은 모든 프로세스가 공유하는 메모리 영역으로 데이터베이 스 버퍼와 로그 버퍼로 구성되어 있다. 프로그램 영역은 프로세스가 시작될 때 DBMS에 의하여 할 당되는 비공유 메모리 영역으로, 프로세스에서 필요로 하는 데이터나 제어 정보 등을 저장한다.




[그림 5-2-3] DBMS 메모리 구조

### 데이터베이스 버퍼(Buffer)

데이터베이스 버퍼는 데이터 파일로부터 읽어 들인 데이터 블록의 복사본을 가지고 있다. 인스턴 스에 동시 접속된 모든 사용자 프로세스는 데이터베이스 버퍼에 대한 액세스를 공유한다. 버퍼는 더티 목록(Dirty list)과 LRU(Least Recently Used) 목록을 가지고 있다. 더티 목록은 더티 버 퍼를 가진다(더티 버퍼는 수정되었지만 아직 디스크에 기록되지 않은 버퍼이다). LRU 목록은 빈 버퍼, 현재 액세스 중인 고정된 버퍼, 더티 목록으로 이동되지 않은 더티 버퍼를 가진다.

데이터 액세스 순서는 다음과 같다.

- 데이터베이스 사용자 프로세스가 데이터를 요구할 때 데이터베이스 버퍼에 있는 데이터를 검색한다.
- 데이터를 찾으면 메모리에서 직접 데이터를 읽는다.
- 데이터베이스 버퍼에 없으면 데이터 블록을 디스크의 데이터 파일에서 버퍼로 복사한다. 버퍼에 있는 데이터를 적중하면 수행 속도가 빠르다.
- 데이터 블록을 디스크에서 버퍼로 읽어 들이기 전에 프로세스는 먼저 빈 버퍼를 찾는다. 프로세스는 끝에서부터 LRU(Least Recent Used) 목록을 검색한다. 프로세스는 빈 버퍼를 찾거나 버퍼의 임계점에 도달할 때까지 검색한다.
  * 더티 버퍼를 찾을 경우 이 버퍼를 더티 목록으로 이동시킨다.
  * 사용자 프로세스가 빈 버퍼를 찾으면 데이터 블록을 디스크에서 버퍼로 읽어 들이고 이를 LRU 목록이 MRU(Most Recent Used) 끝으로 이동시킨다.

사용자 프로세스가 전체 테이블을 스캔한 경우에는 테이블 블록을 버퍼로 읽어 들여 LRU 목록의 LRU 끝에 놓는다. 전체를 스캔하는 테이블은 짧은 시간 동안 사용될 가능성이 높으므로 더 자주 사용되는 블록이 남아 있도록 빨리 제거한다. 버퍼 캐시의 크기는 데이터를 요구했을 때 적중률에 영향을 준다. 크면 그만큼 요구한 데이터를 포함하고 있을 가능성이 크다. 고정된 캐시 사이즈에서 는 가능한 한 불필요한 데이터를 메모리에 올리지 않으면 적중률이 높아진다. 효과적인 인덱스 디 자인이나 SQL 사용으로 적중률을 높일 수 있다.

### 로그 버퍼(Log Buffer)

로그 버퍼는 데이터베이스의 변경 사항 정보를 유지하는 것으로, 일반적으로 원형 버퍼를 사용한 다. Insert, delete, create, alter 또는 drop 작업으로 변경된 사항을 재구성하거나 재실행하는 데 필요한 정보인 REDO 입력 항목을 가진다. REDO 입력 항목은 데이터베이스 복구에 사용된 다. 로그 버퍼의 내용은 서버 프로세스에 의해서 로그 파일에 작성된다.

### 공유 풀(Shared Pool)

공유 풀은 라이브러리 캐시, 딕셔너리 캐시, 제어 구조 등으로 구성되어 있다. 라이브러리 캐시는 SQL 영역, 저장 SQL 프로시저 영역, 제어 구조 등을 공유하고, 딕셔너리 캐시는 데이터 딕셔너리 정보를 공유한다. 공유 풀에는 LRU 알고리즘에 의해 영역을 할당하고 해제한다. 여러 세션이 사용 하는 공유 풀 항목은 원래 항목을 생성한 프로세스가 종료되어도 해당 항목이 유용한 한 계속 메모 리에 남아 있다. 이는 SQL을 실행하기 위한 파싱 오버 헤드와 처리가 최소한으로 유지되게 한다.

### 정렬 영역(Sort Area)

정렬을 하려면 메모리에 영역이 필요하다. 인덱스를 생성하거나 SQL문에 GROUP BY 연산, ORDER BY가 있을 경우 정

렬 작업을 한다. DBMS별이나 동일 DBMS라도 버전에 따라서 정렬 영역이 존재하는 영역이 차이가 있다. 정렬되어야 하는 데이터량이 메모리 영역을 초과할 때는 데이터를 작은 부분으로 나눈 후 각 부분을 개별적으로 정렬하고 개별적으로 정렬된 결과는 병합하여 최종 결과를 생성한다. 일정한 성능 을 유지하려면 정렬해야 할 대상이 클 경우 정렬 영역을 크게 한 후 작업을 실시하고 불필요한 정 렬 작업을 최소화 해야 한다.

## 프로세스 구조

DBMS에서 프로세스는 사용자 프로세스와 DBMS 프로세스로 분류된다. DBMS 프로세스는 서버 프로세스와 백그라운드 프로세스로 나뉜다.

■ 사용자 프로세스

사용자 프로세스는 애플리케이션이나 데이터베이스 도구를 실행할 때 생성된다. 이때 세션이 만들어지고 세션은 사용자 프로세스와 데이터베이스 인스턴스 간 통신 경로가 된다.

■ 서버 프로세스

서버 프로세스는 사용자 프로세스와 통신을 하는 역할을 한다. 다중 스레드 서버 방식과 단일 서버 프로세스 방식이 있다. 다중 스레드 서버를 사용할 경우는 단일 서버 프로세스를 여러 사용자 세션 간에 공유한다. 단일 서버 프로세스 방식은 각 사용자 세션에 대해 하나의 서버 프로세스를 생성한다.

■ 백그라운드 프로세스

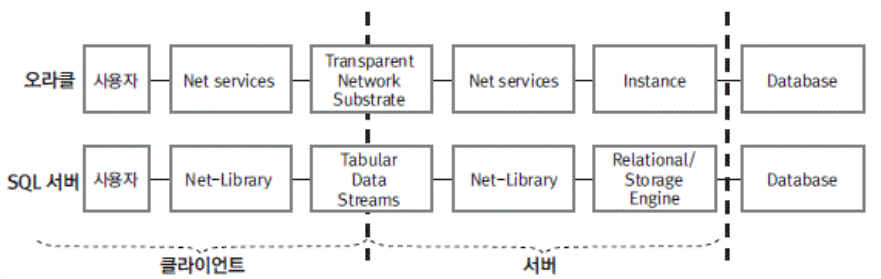
데이터베이스가 동작하기 위한 프로세스들로 구성되면 대부분의 DBMS에서 다중 프로세스 방식이다.

[표 5-2-1] DBMS 주요 백그라운드 프로세스

| 오라클 | SQL 서버 | 설명 |
|---|---|---|
| PMON<br>process monitor | ODS<br>Open data services | 사용자 프로세스에 장애가 발생하면 프로세스 복구를 수행 |
| SMON<br>system monitor | DB cleanup/shrinking | 인스턴스 시작 시 필요한 경우 고장 복구를 수행, 임시 세그먼트 정리 수행 |
| DBWn<br>Database Writers | Lazywriter thread | 버퍼의 내용을 데이터 파일에 기록 |
| CKPT checkpoint | DB checkpoint thread | 체크 포인트가 발생하면 데이터 파일의 헤더를 갱신 |
| LGWR<br>Log Writer | Log writer thread | 로그 버퍼를 관리하여 로그 버퍼를 디스크의 로그 파일에 기록 |

## 실행 구조

사용자는 데이터베이스 내에 있는 데이터를 조작하기 위해 데이터베이스에서 제공하는 인터페이 스를 사용한다. 대부분의 다중 사용자를 지원하는 데이터베이스는 클라이언트/서버 구조이며, 클라 이언트 부분에 사용자 인터페이스를 제공한다.




[그림 5-2-4] 사용자와 데이터베이스 인터페이스

사용자는 데이터베이스를 이용할 수 있는 사용자 인터페이스에서 명령어를 입력하고 데이터베이 스에 결과를 요청하면 네트워크 서비스를 통하여 데이터베이스 인스턴스(엔진)에 전달된다. 전달된 명령어는 문법적인 오류나 의미적인 오류를 확인하고 옵티마이저(Optimizer)에 의해 SQL로 요구 된 결과를 최소의 비용으로 처리할 수 있는 최적의 처리 경로를 결정하여 실행 계획(Execution Plan)을 작성한다. 실행 계획에 의해 데이터베이스 엔진은 실행(Execution) 과정이 반복된다. 사용 자에게 전달될 데이터 결과가 있으면 네트워크 서비스를 통하여 정해진 버퍼 사이즈만큼씩 전달 (Fetch)한다.

### 옵티마이저

관계형 DBMS에서 사용되는 언어는 SQL이다. SQL 언어의 특징은 사용자가 데이터베이스에서 자신이 원하는 데이터(What)만 정의하고 그 데이터를 어떻게(How) 구하는가는 DBMS가 자동으로 결정해 처리해 준다. 어떻게 처리할 것인가를 결정하는 것이 옵티마이저이다.

여러 개의 테이블들을 조인할 때 조인 순서, 조인 방법, 테이블 액세스 방법을 선택하는 것을 실행 계획이라고 한다. 옵티마이저는 가능한 실행 계획들을 모두 검토하고 이중에서 가장 효과적인 것을 결정한다. 옵티마이저가 최적의 실행 계획을 찾는 과정을 최적화라 한다.

최적화 과정은 주어진 SQL 질의를 처리할 수 있는 모든 실행 계획을 다 고려할 수 없으므로 비용 을 산정한다. 비용 산정은 데이터베이스 내의 데이터들에 대해 갖고 있는 통계 정보와 비용을 예측하 는 모델을 이용하여 비용을 계산한다. 이를 비용 기준 최적화(CBO, Cost-Based Optimization)라 한다. 이때 실행 계획에 대한 것은 예상 비용이며, 실제 수행할 때의 비용과는 차이가 있다. 비용 산 정의 과정 없이 일정한 액세스 방법에 따라 정해진 우선순위로 실행 계획을 작성하는 것을 규칙 기준 최적화(RBO, Rule-Based Optimization)라 한다.

SQL 튜닝은 특정 SQL 질의의 수행 시간을 단축하는 것이다. 옵티마이저와 관련한 방법으로는 SQL 재작성, 힌트 사용, 새로운 인덱스 추가, 통계 데이터의 추가/갱신 등을 통해 옵티마이저가 더 욱 더 효율적인 실행 계획을 생성하도록 하는 것이다.

### SQL 실행 단계

사용자의 SQL 질의는 크게 다음 4단계를 거쳐서 수행된다.

1. 파싱(Parser)
2. 옵티마이저(Query Optimizer)
3. 로우 소스 생성(Row Source Generator)
4. SQL 실행(SQL Execution Engine)

■ 파싱(Parser) 단계

SQL의 구문(syntactic)과 의미(semantic)가 정확한지 검사하고, 참조된 테이블에 대해 사용자가 접근 권한을 가지고 있는지를 검사한다. 그리고 라이브러리 캐시에서 같은 SQL 문장이 존재하는 지 찾는다. 같은 SQL 문장 중에 같은 버전이 존재하면 기존 정보를 이용하여 실행하고, 존재하지 않으면 다음 단계를 진행한다.

■ 옵티마이저(Query Optimizer) 단계

앞에서 넘겨받은 결과 정보(parsed query)를 이용해 최적의 실행 계획을 선택한다.

■ 로우 소스 생성(Row Source Generator) 단계

옵티마이저에서 넘겨받은 실행 계획을 내부적으로 처리하는 자세한 방법을 생성하는 단계이다.
'로우 소스'란 실행 계획을 실제로 구현하는 각 인터페이스를 지칭하는 말로, 테이블 액세스 방법, 조인 방법, 정렬(sorting) 등을 위한 다양한 로우 소스가 제공된다. 따라서 이 단계에서는 실행 계 획에 해당하는 트리 구조의 로우 소스들이 생성된다.

■ SQL 실행(SQL Execution Engine)

생성된 로우 소스를 SQL 수행 엔진에서 수행해 결과를 사용자에게 돌려주는 과정이다. 소프트 파싱(Soft Parsing)과 하드 파싱(Hard Parsing)은 크게 옵티마이저 단계의 포함 여부에 따른 차이이다. 즉, 소프트 파싱은 이미 최적화를 한 번 수행한 SQL 질의에 대해 옵티마이저 단계와 로우 소스 생성 단계를 생략하는 것이고, 하드 파싱은 이 두 단계를 새로 수행하는 것이다. 따라서 하 드 파싱은 통계 정보 접근과 실행 계획 탐색으로 인해 시간이 많이 걸린다.

## 명령어

데이터베이스와 사용자 간의 통신을 하기 위해 사용자는 DBMS에서 제공하는 명령어를 사용한다. 이들 명령어는 DBMS마다 문법 차이는 있지만 ANSI SQL92 Entry Level을 준수하고 있다. 명령 어에 대한 분류 방법도 차이가 있으나 데이터 정의 언어(DDL), 데이터 조작 언어(DML), 제어 명령 어(Control Statements)로 나뉜다.

### 데이터 정의 언어(DDL, Data Definition Language)

스키마 객체를 생성(CREATE)하고, 구조를 변경(ALTER)하고, 삭제(DROP), 명칭을 변경(Rename)하는 데 사용한다. 데이터베이스, 사용자, 테이블, 칼럼, 데이터 타입, 참조 무결성 제약 정의, 영역 무결성 제약 정의, 인덱스 등 모든 객체는 DDL에 의해서 관리된다.

DDL 실행은 현재 진행되는 트랜잭션에 대해 암시적으로 COMMIT을 실행하므로 트랜잭션 명령 어에서 제어할 수 없다. 생성, 변경, 삭제 작업은 실행 후 취소가 불가능하므로 사용상 주의가 필요 하다. Truncate 명령어는 데이터를 삭제하는 delete와 유사하지만 Truncate는 DDL이고 delete는 DML이므로 내부적으로 수행 구조가 다르다. 그 외 사용자 특권(Privilege)과 역할(Role)을 허가 (Grant)하고 취소(Revoke, Deny)하거나 데이터 사전에 코멘트를 추가하고 오디팅(Auditing) 옵션 을 설정하는 등의 명령어가 있다.

### 데이터 조작 언어(DML, Data Manipulation Language)

DML은 데이터베이스에 있는 데이터를 조작할 수 있게 해주는 명령어로 DELETE, INSERT, SELECT, UPDATE 등이 대표적인 ANSI 표준 DML 명령어이다. 그 외 MERGE, LOCK TABLE, EXPLAIN PLAN 등이 있다. DML은 현재 트랜잭션을 암시적으로 처리 하지 않으므로 별도의 트랜 잭션 관리 명령어와 같이 수행한다.

DML문 처리 단계는 다음과 같다.

■ 1단계 커서 생성

커서는 SQL문에 대해 독립적으로 생성된다. 커서는 모든 SQL문에 사용될 수 있도록 생성된다. 대부분의 애플리케이션에서 커서는 자동으로 생성되지만 선행 컴파일러 프로그램에서는 커서 생 성이 암시적으로 발생할 수도 있고 명시적으로 선언될 수도 있다.

■ 2단계 명령문 구문 분석

SQL문을 변화하여 유효한 명령문인지 검증

- 데이터 딕셔너리를 탐색해 테이블과 열 정의 검사
- 구문 분석 잠금을 획득하여 객체 정의가 변경되지 않도록 함
- 참조한 스키마 객체에 액세스 권한 검사
- 명령문에 대한 최적의 실행 계획을 결정
- 공유 SQL 영역으로 로드
- 분산 명령문의 경우 명령문 모두 또는 일부를 원격 노드로 라우트

■ 3단계 질의 결과 설명(SELECT일 때)

데이터 유형, 길이, 이름 등 질의 결과의 특성을 판별한다.

■ 4단계 질의 결과 출력 정의(SELECT일 때)

질의에 대한 정의 단계에서 위치, 크기, 인출한 각 값을 받기 위해 정의된 변수의 데이터 유형을 지정한다.

■ 5단계 변수 바인드

값을 찾을 수 있는 메모리 주소를 지정한다.

■ 6단계 명령문 병렬화(병렬 처리일 때)

병렬화는 다중 서버 프로세스로 하여금 동시에 SQL문의 작업을 수행하도록 하므로 작업이 더 신속하게 완료될 수 있다.

■ 7단계 명령문 실행

SELECT나 INSERT문이면 데이터의 내용이 변경되지 않으므로 어떠한 행에 대해서도 잠금이 필요하지 않다. UPDATE, DELETE문에 영향을 받는 모든 행은 트랜잭션에 대한 다음 처리까지 잠 겨있어 데이터베이스의 다른 사용자는 사용할 수 없다.

■ 8단계 질의 로우 인출(SELECT일 때)

인출 단계에서 행이 선택되고 정렬된다. 연속적인 인출(Fetch)을 통해 마지막 행을 인출할 때까지 다른 결과 행이 읽혀진다.

■ 9단계 커서 닫기

### 제어 명령어(Control Statement)

제어 명령어는 DBMS마다 사용자에게 제공하는 형태에 차이가 있다. 단순한 명령어 형태로 제공되 기도 하고 저장 프로시저 형태로 제공되기도 한다. 오라클과 MS-SQL을 비교하면 아래 표와 같다.

[표 5-2-2] 제어 명령어

| 구분 | Oracle | MS-SQL |
|---|---|---|
| Transaction Control | COMMIT [work] | COMMIT [work\|transaction] |
| | ROLLBACK | ROLLBACK [work\|transaction] |
| | SAVEPOINT | SAVE TRAN[SACTION] |
| | Set transaction | SET |
| Session control | Alter session | SET |
| | Set role | Sp_setapprole |
| | Alter system | Sp_configure |
| | Alter database | Sp_dboption |

트랜잭션 제어문은 1개 이상의 SQL문을 논리적으로 하나의 처리 단위로 적용하기 위해 사용하는 명령어이다. 트랜잭션의 처음과 종료는 명시적으로 COMMIT되거나 ROLLBACK을 처리한다.

세션 제어문은 실행하고 저장하는 SQL문에는 사용할 수 없으며 사용자 세션의 특성을 정의하는 명 령어이다. 개별적인 사용자 세션에 따라 환경 변수를 변경할 수 있다. 예를 들어, 대량의 배치 작업을 하기 위해 메모리 사이즈를 변경하거나 NLS 환경 변수를 클라이언트 프로그램과 동일하게 조정하는 등의 작업을 지원한다.

시스템 제어문은 시스템이나 데이터베이스 레벨에서 재기동(Restart) 없이 환경 변수 등을 조정할 수 있다. 데이터베이스 전체적으로 NLS 환경 변수를 변경하는 작업 등이 가능하다.

## 저장 프로시저

데이터 조작 언어는 비절차적 언어이다. 대부분의 DBMS에서는 이를 보완하기 위해 절차적 언어 를 제공한다. 오라클은 PL/SQL로, MS-SQL은 Transaction-SQL로 지원한다. 절차적 언어를 이 용하여 저장 프로시저를 생성할 수 있다.

### 저장 프로시저 설계 지침

높은 응집도와 낮은 결합도를 유지한 설계가 필요하다. 하나의 작업을 중점적으로 완료하도록 프 로시저를 정의한다. 여러 프로시저 코드에서 불필요하게 중복될 수 있는 공통적인 하위 작업이 있을 수 있으므로 여러 개의 서로 다른 하위 작업을 갖는 긴 프로시저는 정의하지 않는다. DBMS에서 제공되는 기능과 중복되는 프로시저는 정의하지 않는다. 선언적 무결성 제약 조건을 사용하여 수행할 수 있는 간단한 데이터 무결성 규칙을 프로시저로 정의하지 않는다.

### 저장 프로시저의 장점

■ 보안

데이터 보안을 강제로 수행한다. 사용자는 작성자의 권한으로 실행되는 프로시저와 함수를 통해서 만 데이터에 액세스하도록 데이터베이스 작업을 제한할 수 있다. 프로시저를 실행하는 권한만 있 는 사용자는 프로시저를 호출할 수는 있지만 테이블 데이터를 조작할 수는 없다.

■ 성능

각각의 SQL문 실행과 비교할 때 네트워크를 통해 보내야 하는 정보의 양을 현격하게 줄인다. 한 번 정보를 보낸 후에는

사용될 때마다 호출되기 때문이다. 데이터베이스에서 프로시저를 컴파일한 상태로 사용되므로 실행 시 별도 컴파일이 필요 없고 공유 풀을 이용하여 재사용된다.

■ 메모리 할당

많은 사용자의 실행을 위해 프로시저의 단일 복사본만이 메모리에 로드된다. 동일한 코드를 공유 하면 애플리케이션에 의한 메모리 요구를 줄인다.

■ 생산성

개발 생산성을 증가시킨다. 프로시저 집합으로 애플리케이션을 설계하여 불필요한 코딩을 피하고 생산성을 증가시킨다. 작업 수행에 필요한 SQL문을 재작성하지 않고도 모든 애플리케이션에 의 해 호출될 수 있다. 데이터 관리 방법이 변경되면 사용하는 애플리케이션이 아닌 프로시저만 수정 하면 된다.

■ 무결성

애플리케이션의 무결성과 일관성을 향상시킨다. 검증된 프로시저는 다시 테스트하지 않고 많은 애 플리케이션에서 재사용할 수 있다. 프로시저가 참조하는 데이터 구조가 변경되었다면 프로시저만 재컴파일하면 된다. 프로시저를 호출하는 애플리케이션은 수정하지 않아도 된다.

## 트리거(Trigger)

DBMS에서 INSERT, UPDATE, DELETE문을 관련 테이블에 대해 실행하거나 데이터베이스 시 스템 작업이 발생하면 암시적으로 실행되는 트리거를 정의할 수 있다. 저장 프로시저와 트리거는 호 출하는 방법이 다르다. 프로시저는 사용자, 애플리케이션 또는 트리거에 의해 명시적으로 실행된다. 반면에 하나 이상의 트리거는 접속된 사용자나 사용되는 애플리케이션에 관계없이 트리거 이벤트가 발생되면 DBMS에 의해 암시적으로 실행된다.

### 트리거 사용

과다한 트리거 사용은 복잡한 내부 종속성을 초래하여 대규모 애플리케이션에서 유지 관리를 어렵 게 하므로 주의가 필요하다. 트리거는 DML 작업이 정규 업무 시간 동안 실행되도록 테이블에 대해 DML 작업을 제한할 수 있다. 트리거는 다음과 같이 사용한다.

- 자동적으로 파생된 열 값 생성(예 : 합계, 잔액, 재고량 등)
- 잘못된 트랜잭션 방지(예 : 무결성 제약 구현)
- 복잡한 보안 권한 강제 수행
- 분산 데이터베이스의 노드상에서 참조 무결성 강제 수행
- 복잡한 업무 규칙 강제 수행
- 이벤트 로깅 작업이나 감사 작업
- 동기 테이블 복제 작업
- 테이블 액세스에 대한 통계 수집

### 트리거 유형

■ 행 트리거 및 명령문 트리거

행 트리거는 테이블이 트리거링 명령문에 의해 영향을 받을 때마다 실행된다. 예를 들어 UPDATE 문이 테이블의 여러 행을 갱신하면 각 행에 대해 한 번씩 실행된다. 명령문 트리거는 테이블에서 트 리거링 명령문에 의해 영향을 받는 행 수에 관계없이 한 번 실행한다. 예를 들어, DELETE문이 여 러 행을 삭제하면 테이블에서 삭제되는 행 수에 관계없이 한 번만 실행된다. 보안 감사나 감사 레코 드를 만들 때 사용된다.

■ BEFORE 및 AFTER 트리거

BEFORE 트리거는 명령문이 실행되기 전에 트리거 작업을 실행한다.

- 불필요한 rollback을 제거하기 위해 트리거 작업이 실행 여부를 결정할 때 사용된다.
- 트리거링 INSERT 또는 UPDATE문을 완료하기 전에 특정 열 값을 구하기 위해 사용된다.

AFTER 트리거는 명령문이 실행된 후에 트리거 작업을 실행한다.

### 트리거링 이벤트와 제한 조건

트리거링 이벤트는 특정 테이블에 대한 INSERT, UPDATE, DELETE문이 실행될 때이다. 모든 DBMS에서 지원하는 사항은 아니지만 그 외에 시스템 차원에서 이벤트를 발생할 수 있다. 트리거 제한 사항은 트리거 실행을 위해 TRUE여야 하는 논리적 표현식을 지정한다.

데이터베이스 관리 시스템(DBMS) | 데이터 액세스 | 트랜잭션 | 백업 및 복구

트랜잭션은 ATM이나 데이터베이스 등의 시스템에서 더 이상 나눌 수 없는 업무 처리의 단위로, 하 나 이상의 SQL문으로 구성된다. 여기에서 더 이상 나눌 수 없다는 것은 실제로 나눌 수 없다기보다는 나눌 경우 시스템이나 데이터에 심각한 오류를 초래할 수 있다는 의미이다. 이러한 개념의 기능을 데이 터베이스에서 제공하는 것을 트랜잭션이라고 하며, ACID와 같은 기술적인 요건을 충족해야만 한다.

그런데 다중 사용자 환경의 데이터베이스에서 트랜잭션의 개념만 충족한다고 데이터의 일관성을 유 지할 수 있는 것은 아니다. 다중 사용자 환경의 데이터베이스 관리 시스템들은 여러 사용자의 질의나 프 로그램을 동시에 수행하므로 Dirty Read, Non-Repeatable Read, Phantom Read 등의 문제가 발 생할 수 있다. 때문에 트랜잭션들이 동시에 수행될 경우 각 트랜잭션이 고립적으로 수행된 것과 동일한 결과를 내려면 트랜잭션들이 서로 간섭을 일으키는 현상을 최소화하고 데이터의 일관성과 무결성을 보 장하도록 트랜잭션을 제어해야 하는데, 이러한 기능을 트랜잭션의 동시성 제어라 한다.

이 외에 데이터의 일관성을 유지하기 위해서는 트랜잭션 처리 중 장애가 발생했을 경우 데이터를 트 랜잭션이 시작되기 이전 상태로 되돌려 놓는 기능이 필요한데, 이러한 기능을 고장 회복(Recovery)이 라 한다.

## 트랜잭션 관리

트랜잭션은 하나의 논리적 작업 단위를 구성하는 하나 이상의 SQL문으로 구성되며, 모든 트랜잭 션은 두 가지 상황으로 종료된다. 실행한 논리적 작업 단위 전체가 성공적으로 종료되면 그 트랜잭션 은 영구적으로 데이터베이스에 저장된다. 이것을 COMMIT이라 한다. 다른 한 가지는 실행한 SQL 중 하나라도 정상 종료되지 않으면 논리적인 작업 단위 전체를 이전 상황으로 rollback한다.

다중 사용자 환경에서 트랜잭션은 동시성 제어(Concurrency control)와 고장 회복(recovery) 기 법에 의하여 관리된다. 동시성 제어는 한 사용자의 작업이 다른 사용자의 작업에 의해 방해 받지 않 도록 하는 조치들로 구성되고, 고장 회복은 데이터 처리 중 통신, 하드웨어, 소프트웨어 오류 발생 등 예기치 않은 예외 상황에 대한 조치들로 구성된다.

## 트랜잭션 특성

■ 원자성(Atomicity)

하나의 트랜잭션은 하나의 원자적 수행이다. 트랜잭션은 완전히 수행하거나 전혀 수행되지 않은 상태로 회복되야 한다. 계좌이체에서 송신 계좌에서 출금과 수신 계좌에 입금은 전체가 완전하게 수행되어야 한다.

■일관성 유지(Consistence)

트랜잭션을 실행하면 데이터베이스를 하나의 일관된 상태에서 또 다른 일관된 상태로 바뀐다. 일관성은 프로그래머나 무결성 제약 조건을 시행하는 DBMS에서 처리된다.

■고립성(Isolation)

하나의 트랜잭션은 완료될 때까지 자신이 갱신한 값을 다른 트랜잭션들이 보게 해서는 안된다. 고립성이 시행되므로 임시 갱신 문제를 해결하며 트랜잭션들의 연쇄 복귀는 불필요하다. 고립성은 갱신에 따른 손실이 없어야 하며 오손 판독이 없고 반복 읽기 성질을 갖는다.

■영속성(Durability)

단 한 트랜잭션이 데이터베이스를 변경시키고 그 변경이 완료되면 결과는 이후의 어떠한 고장에도 손실되지 않아야 한다. 지속성을 보장하는 것은 회복 기법의 책임이다.

## 트랜잭션의 일관성

트랜잭션 수준 읽기 일관성(Transaction-Level Read Consistency)은 트랜잭션이 시작된 시점 을 기준으로 일관성 있게 데이터를 읽는 것을 말한다. 트랜잭션이 진행되는 동안 다른 트랜잭션에 의 해 변경이 발생하더라도 이를 무시하고 트랜잭션 내에서 계속 일관성 있는 데이터를 보고자 하는 업 무 요건이 있을 수 있다. 물론 트랜잭션이 진행되는 동안 자신이 발생시킨 변경 사항은 읽을 수 있어 야 한다.

대부분 DBMS가 기본적으로 트랜잭션 수준 읽기 일관성을 보장하지 않으며, 트랜잭션 수준으로 읽기 일관성을 강화하려면 고립화 수준을 다음과 같이 높여 주어야 한다.

set transaction isolation level serializable;

### 낮은 단계 트랜잭션 고립화 수준에서 발생할 수 있는 현상들

■ Dirty Read(= Uncommitted Dependency)

다른 트랜잭션이 변경 중인 데이터를 읽었는데, 그 트랜잭션이 최종 롤백됨으로써 현재 트랜잭션이 비일관성(inconsistency) 상태에 놓이는 것을 말한다.

■ Non-Repeatable Read(= Inconsistent Analysis)

한 트랜잭션 내에서 같은 쿼리를 두 번 수행할 때 그 사이에 다른 트랜잭션이 값을 수정 또는 삭제 함으로써 두 쿼리의 결과가 상이하게 나타나는 현상을 말한다.

■ Phantom Read

한 트랜잭션 내에서 같은 쿼리를 두 번 수행할 때 첫 번째 쿼리에서 없던 유령(Phantom) 레코드가 두 번째 쿼리에서 나타나는 현상을 말한다.

### 트랜잭션 고립화 수준(Transaction Isolation Level)

ANSI/ISO SQL standard(SQL92)에서 정의하고 있는 네 가지 트랜잭션 고립화 수준을 요약하면 다음과 같다.

■ 레벨 0(= Read Uncommitted)

트랜잭션에서 처리 중인 아직 커밋되지 않은 데이터를 다른 트랜잭션이 읽는 것을 허용한다. Dirty Read, Non-Repeatable Read, Phantom Read 현상 발생

■ 레벨 1(= Read Committed)

대부분의 DBMS가 기본 모드로 채택하고 있는 일관성 모드로서, 트랜잭션이 커밋되어 확정된 데 이터만 읽는 것을 허용한다. Non-Repeatable Read, Phantom Read 현상 발생

■ 레벨 2(= Repeatable Read)

선행 트랜잭션이 읽은 데이터는 트랜잭션이 종료될 때까지 후행 트랜잭션이 갱신하거나 삭제하는 것을 불허함으로써 같은 데이터를 두 번 쿼리했을 때 일관성 있는 결과를 리턴한다. Phantom Read 현상 발생

■ 레벨 3(= Serializable Read)

선행 트랜잭션이 읽은 데이터를 후행 트랜잭션이 갱신하거나 삭제하지 못할 뿐만 아니라 중간에 새로운 레코드를 삽입하는 것도 막아줌으로써 완벽한 읽기 일관성을 제공한다. 참고로 오라클은 잠김을 사용하지 않고 Undo 데이터를 이용해 Serializable Read를 구현한다.

## 동시성 제어(Concurrency Control)

동시성 제어(Concurrency Control)란 다수의 사용자가 데이터베이스에 동시에 접근하여 같은 데이터를 조회 또는 갱신을 할 때 데이터 일관성을 유지하기 위한 일련의 조치를 의미한다. 여기서 데이터 동시성(Data Concurrency)이란 다수의 사용자가 동시에 데이터에 접근할 수 있어야 한다 는 의미이고, 데이터 일관성(Data Consistency)이란 각각의 사용자가 자신의 트랜잭션이나 다른 사람의 트랜잭션에 변경된 내용을 포함하여 일관된 값을 본다는 의미이다. 동시성 제어는 낙관적 동 시성 제어(Optimistic Concurrency Control)와 비관적 동시성 제어(Pessimistic Concurrency Control)로 나뉜다. 낙관적 동시성 제어 알고리즘은 다수 사용자가 동시에 같은 데이터에 접근할 경 우가 적다고 보고 구현한 알고리즘이고, 비관적 동시성 제어는 다수 사용자가 동시에 같은 데이터에 접근할 경우가 많다고 보고 구현한 알고리즘이다.

### 낙관적 동시성 제어

낙관적 동시성 제어(Optimistic Concurrency Control)는 사용자들이 같은 데이터를 동시에 수 정하지 않을 것이라고 가정한다. 따라서 데이터를 읽을 때는 잠김을 설정하지 않는다. 그러나 낙관적 입장에 섰더라도 동시 트랜잭션에 의한 데이터의 잘못된 갱신에 주의를 기울여야 한다. 읽는 시점에는 잠김을 사용하지 않지만 데이터를 수정하고자 하는 시점에 앞서 읽은 데이터가 다른 사용자에 의해 변경되었는지를 반드시 검사해야 하는 것이다.

```
select 적립포인트, 방문횟수, 최근방문일시, 구매실적, 변경일시
into :a, :b, :c, :d, :mod_dt
from 고객
where 고객번호 = :cust_num;
-- 새로운 적립포인트 계산
update 고객 set 적립포인트 = :적립포인트, 변경일시 = SYSDATE
where 고객번호 = :cust_num
and 변경일시 = :mod_dt ; → 최종 변경 일시가 앞서 읽은 값과 같은지 비교
  if sql%rowcount = 0 then
dbms_output.put_line('다른 사용자에 의해 변경되었습니다.');
end if;
```

낙관적 동시성 제어를 사용하면 잠김이 유지되는 시간이 매우 짧아져 동시성을 높이는 데 유리하다.

### 비관적 동시성 제어

비관적 동시성 제어(Pessimistic Concurrency Control)는 사용자들이 같은 데이터를 동시에 수 정할 것이라고 가정한다. 따라서 한 사용자가 데이터를 읽는 시점에 잠김을 걸고 조회 또는 갱신 처 리가 완료될 때까지 이를 유지한다. 구체적으로 말해 다음과 같이 for update절을 사용해 select 시 점에 해당 레코드에 잠김을 걸어두는 식이다.

```
select 적립포인트, 방문횟수, 최근방문일시, 구매실적 from 고객
where 고객번호 = :cust_num for update;
```

```
-- 새로운 적립포인트 계산
update 고객 set 적립포인트 = :적립포인트 where 고객번호 = :cust_num;
```

잠김은 첫 번째 사용자가 트랜잭션을 완료하기 전까지 다른 사용자들이 그 데이터를 수정할 수 없 게 만들기 때문에 비관적 동시성 제어는 자칫 시스템 동시성을 심각하게 떨어뜨릴 우려가 있다. 이를 방지하려면 다음과 같이 wait 또는 now ait 옵션을 함께 사용해야 한다.

```
for update nowait → Lock이 걸렸다면 대기 없이 Exception을 던짐
for update wait 3 → Lock이 걸렸다면 3초간 대기하고 Exception을 던짐
```

## 동시성 제어 기능

다중 사용자 환경에서는 트랜잭션들의 동시성을 제어하기 위해 Locking, 2PC, Timestamp 등의 기법을 주로 사용한다. 잠김(Locking)은 트랜잭션의 동시성을 제어하기 위해 가장 많이 사용되는 기 법으로 데이터 처리 과정에 있는 데이터를 읽지 못하게 하는 기법이다. 잠김은 암시적인 잠김 (Implicit Locking)과 명시적인 잠김(Explicit Locking)으로 구분된다. 암시적인 잠김은 DDL(Data Definition Language)를 실행할 때와 같이 DBMS에 의해 자동으로 실시된다. 명시적 인 잠김은 사용자에 의한 트랜잭션 제어(Transaction Control)에 의해 실시된다.

■ 잠김 단위(Lock Granularity 또는 Isolation Level)

잠김 단위는 잠김 대상의 크기를 뜻하며, 단위가 커지면 관리해야 하는 대상의 수가 적어지므로 DBMS가 관리하기 쉬워지지만 동일한 잠김 대상에 동시 액세스할 확률이 높아져 충돌이 자주 발 생하게 된다. 반대로 잠김의 단위가 작아지면 관리해야 하는 대상의 수가 많아져 관리하기는 어려 워지지만 동일한 잠김 대상에 동시 액세스할 확률이 낮아져 충돌 횟수는 적어지게 된다. DBMS에 따라 다소 차이는 있지만 일반적으로 데이터베이스 레벨, 테이블 레벨, 페이지(블록) 레벨, 행 레벨 이 대부분의 DBMS에 의하여 지원되고 있다.

■ 잠김 확산(Locking Escalation)

잠김 확산이란 관리해야 하는 잠금 단위의 개수가 미리 설정한 임계치에 도달하게 되면 잠김의 단 위를 현재 관리하고 있는 단위보다 하나 높은 수준으로 올리는 기능을 말한다. 이러한 개념은 하위 수준에서 관리해야 하는 대상을 상위 수준에서 관리함으로써 그 아래 수준에서는 개별적으로 잠금 들을 관리할 필요가 없어지므로 잠금 대상의 개수가 줄어들며, 그에 따른 자원들도 해제되어 잠김 을 관리하기 위한 DBMS의 부담도 최소화된다.

■ 잠김(Locking)의 유형

일반적으로 읽기 작업에서는 공용 잠김(Shared lock)을 필요로 하고 쓰기 작업에서는 배타적 잠 김(Exclusive lock)을 필요로 한다. 오라클의 경우, 읽기 작업에 공용 잠김을 사용하지 않고 Undo 데이터를 이용하는 방식으로 읽기 일관성을 제공한다. 대부분 DBMS가 하나의 행을 잠글 때 해당 테이블에 대한 잠김도 동시에 일어나는데, 이를 오라 클의 경우'테이블(TM) 잠김', SQL 서버의 경우'Intent 잠김'이라 부른다. 그럼으로써 현재 트 랜잭션이 갱신 중인 테이블에 대한 호환되지 않는 DDL 또는 DML 오퍼레이션을 방지한다. 테이블 잠김에는 다음과 같은 여러 가지 잠김 모드가 있으며, 이 중 RX 모드 테이블 잠김은 DML 작업에 사용되고, RS 모드 테이블 잠김은 select for update 문을 위해 사용된다.

- RS : row share(또는 SS : sub share)
- RX : row exclusive(또는 SX : sub exclusive)
- S : share
- SRX : share row exclusive(또는 SSX : share/sub exclusive)
- X : exclusive

잠김 모드 간 호환성(Compatibility)을 정리하면 있 음을 의미함).

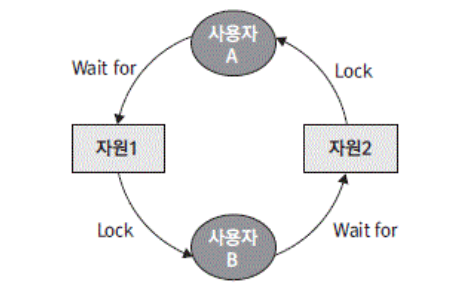
| | RS | RX | S | SRX | X |
|---|---|---|---|---|---|
| RS | O | O | O | O | |
| RX | O | O | | | |
| S | O | | O | | |
| SRX | O | | | | |
| X | | | | | |

■ 2PC(2 Phased Commit)

2개 이상의 트랜잭션들이 병행적으로 처리되었을 때의 데이터베이스 결과는 그 트랜잭션들을 임 의의 직렬적인 순서로 처리했을 때의 결과와 논리적으로 일치해야 한다. 이처럼 병렬로 수행되는 트랜잭션의 직렬 가능성을 보장하기 위해 주로 사용하는 방법이 2PC(Two-Phased Locking 또 는 2 Phased Commit) 기법이다. 2PC에서는 트랜잭션 필요시 잠김을 필요한 만큼 걸 수 있지만 일단 첫 번째 Locking을 해지하면(Unlock이 되면) 더 이상의 Locking을 걸 수 없다. 따라서 트 랜잭션은 Locking을 거는 성장 단계(Growing Phase)와 Locking을 푸는 축소 단계(Shrinking Phase)의 2단계로 구성된다. 이것은 분산 트랜잭션에서도 데이터 일관성을 유지하기 위해 동일하 게 적용되고 있다.

■ 교착 상태(Dead Lock)

다른 사용자가 잠근 자원이 해제되기를 기다리면서 자신이 잠근 자원을 해제하지 않는 상태로 영 원히 처리를 할 수 없는 무한 대기 상태를 교착 상태라 한다.




[그림 5-2-5] 교착 상태(Dead Lock) - 대기 그래프

교착 상태의 필수 조건은 4가지가 있다.

■ 상호 배제(Mutual Exclusive)

어느 자원에 대해 한 프로세스가 이미 사용 중이면 다른 프로세스는 기다려야 하는 것

■ 점유와 대기(Wait for)

하나 이상의 자원을 할당 받은 채로 나머지 자원을 할당 받기 위해 다른 프로세스의 자원이 해제되 기를 기다리는 프로세스가 존재하는 경우

■ 비중단(no preemption)

자원을 할당 받은 프로세스로부터 자원을 강제로 빼앗지 못하는 것

■ 환형 대기(circular wait)

자원 할당 그래프상에서 프로세스의 환형 사슬이 존재하는 것

위 4가지 교착 상태 필수 조건을 부정함으로써 교착 상태를 예방할 수 있다. 예를 들어, 점유와 대기의 부정으로 사용자가 필요한 자원을 한번에 요청하는 것이다.
4가지 교착 상태를 부정할 수 없는 경우가 발생하므로 트랜잭션을 처리할 때 교착 상태를 회피하 는 방법이 적용된다. 예를 들면, 개발자들이 마스터 테이블과 디테일 테이블을 변경한다면 마스터 테 이블 처리 후 디테일 테이블 처리 혹은 디테일 테이블 처리 후 마스터 테이블 처리로 동일한 순서를 사용하는 것이다.

## 동시성 구현 사례

잠김을 이용해 선분 이력을 추가하고 갱신할 때 발생할 수 있는 동시성 이슈를 해결하는 사례를 살펴보자.

선분 이력 모델은 여러 측면에서 장점이 있지만 잘못하면 데이터 정합성이 쉽게 깨질 수 있다는 단 점이 있다. 아래 모델을 예로 들어 선분 이력이 동시성과 관련해 어떤 문제를 일으킬 수 있고 어떻게 해결할 수 있는지 살펴보기로 하자.

```
declare
cur_dt varchar2(14);
begin
① cur_dt := to_char(sysdate, 'yyyymmddhh24miss');
② update 부가서비스이력
set 종료일시 = to_date(:cur_dt, 'yyyymmddhh24miss') - 1/24/60/60
where 고객ID = 1
and 부가서비스ID = 'A'
and 종료일시 = to_date( '99991231235959', 'yyyymmddhh24miss' ) ;
③ insert into 부가서비스이력(고객ID, 부가서비스ID, 시작일시, 종료일시)
values ( 1, 'A' , to_date(:cur_dt, 'yyyymmddhh24miss')
      , to_date('99991231235959', 'yyyymmddhh24miss') ) ;
④ commit;
end;
```

위 트랜잭션은 기존 최종 선분 이력을 끊고 새로운 이력 레코드를 추가하는 전형적인 처리 루틴이며, 신규 등록 건이면 ②번 update문에서 실패(0건 갱신)하고, ③번에서 한 건이 insert될 것이다. 첫 번째 트랜잭션이 ①을 수행하고 ②로 진입하기 직전에 어떤 이유에서건 두 번째 트랜잭션이 동 일 이력에 대해 ①~④를 먼저 진행해 버린다면 선분 이력이 깨지게 된다. 따라서 트랜잭션이 순차적 으로 진행할 수 있도록 직렬화 장치를 마련해야 하는데, ①번 문장을 수행하기 직전에 select for update문을 이용해 해당 레코드에 잠김을 설정하면 된다.

그런데 아래처럼 부가서비스 이력에 잠김을 걸어 동시성을 관리하려 한다면 기존에 부가서비스 이 력이 전혀 없던 고객일 경우 잠김이 걸리지 않는다. 그러면 동시에 두 개 트랜잭션이 ③번 insert문으 로 진입할 수 있고, 결과적으로 시작일시는 다르면서 종료일시가 같은 두 개의 이력 레코드가 생긴다.

```
select 고객ID from 부가서비스이력
where 고객ID = 1
and 부가서비스ID = 'A'
and 종료일시 = to_date( '99991231235959', 'yyyymmddhh24miss' )
FOR UPDATE NOWAIT ;
```

따라서 부가서비스 이력의 상위 엔터티인 고객 테이블에 잠김을 걸면 완벽하게 동시성 제어를 할 수 있다.

```
select 고객ID from 고객 where 고객ID = 1
FOR UPDATE NOWAIT ;
```

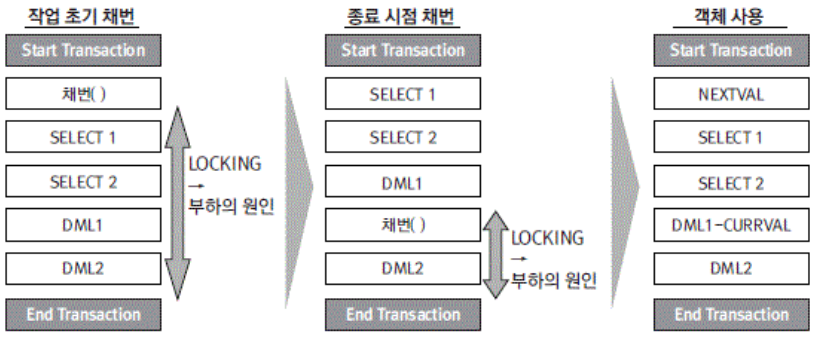
또 다른 상위 엔터티인 부가서비스는 여러 사용자가 동시에 접근할 가능성이 있어 여기에 잠김을 설정하면 동시성에 나빠질 수 있지만, 고객 테이블은 그럴 가능성이 희박하기 때문에 동시성에 미치 는 영향은 거의 0에 가깝다.

### 고장 회복(Recovery)

트랜잭션 처리 중 장애가 발생했을 경우 데이터를 트랜잭션이 시작되기 이전 상태로 회복해야 한다. 이를 위해 데이터베이스는 로그를 사용해 Before Image로 UNDO(취소)를 실시하여 롤백 처리한다.

### 로킹 지속 시간(Locking duration)

Locking duration을 최소화하는 것이 Locking에 의한 지연 문제를 최소화하는 것이다.




[그림 5-2-6] Locking duration

Locking에 의한 경합은 식별자 번호를 얻기 위한 채번 로직에서 많이 발생한다. 따라서 채번은 트랜잭션 종료 시점에 실시하여 locking duration을 최소화하거나 시퀀스나 데이터 타입으로 자동 번호 발생 객체를 사용한다.

데이터베이스는 전산 장비의 고장이나 사고에 대비하여 주기적인 백업을 실시하고 장애의 원인을 해결한 후 데이터베이스를 복구해야 한다. 데이터베이스 관리 시스템은 데이터베이스가 파괴되거나 실행이 중단되었을 경우 복구할 수 있는 기능을 제공한다. 비즈니스 영속성 요구 수준에 따라 제공된 기능을 활용하거나 추가적인 비용을 투자하여 대비를 해야 한다. 본 절은 데이터베이스의 기능을 중심으로 백업 및 복구에 대한 설명한다.

## 장애 유형

장애 원인에 따라서 데이터베이스 기능에 의해서 자연 복구되거나 장애에 대비한 백업 전략에 의해 백업 매체를 통해 복구가 가능한 것도 있다.

### 사용자 실수

사용자 실수로 테이블의 삭제하거나 잘못된 트랜잭션 처리로 데이터의 일관성에 문제가 되는 경우이다. 이는 운영자의 충분한 교육으로 사전에 대비를 충분히 이루어져야 한다. 이런 종류의 장애가 발생했다면 백업된 파일을 이용하여 복구한다.

### 미디어 장애

하드웨어 장애는 CPU, 메모리, 디스크 등 다양하게 발생할 수 있다. 단순한 하드웨어 장애는 하드웨어 교체로 문제를 해결할 수 있지만 미디어 장애는 데이터가 파손되는 장애이다. 대표적인 것이 하드디스크에 장애가 발생하거나 데이터 파일이 파손되는 경우이다. 주기적 또는 실시간으로 백업을 실시하거나 하드디스크 밀러링으로 이에 대비한다. 장애 발생시는 백업 파일을 이용하여 복구를실시한다.

### 구문 장애

구문 장애는 프로그램 오류나 사용 용량이 부족하거나 여유 공간이 부족하여 발생하는 장애이다.

### 사용자 프로세스 장애

사용자 프로그램이 비정상 종료되거나 네트워크 이상으로 세션이 종료되므로 발생하는 오류이다.
이는 데이터베이스 기능에 의해서 복구 처리가 된다.

### 인스턴스 장애

시스템이 비정상적인 요인으로 메모리나 데이터베이스 서버 프로세스가 중단된 경우를 말한다. 하드웨어 장애, 정전, 시스템 파일 파손 등으로 발생할 수 있다. 이 경우는 물리적인 장애 요소를 제거하고 데이터베이스를 재기동하면 롤포워드와 롤백을 수행하여 인스턴스를 복구한다.

## 로그 파일

로그 파일은 데이터베이스에서 처리되는 트랜잭션의 내용을 모두 기록한 것으로 데이터베이스 복구를 위한 가장 기본적인 매체이다. 이를 매체를 사용하여 데이터베이스를 과거의 상태로 복귀시키거나 또는 과거의 상태로부터 현재 상태로 재생성 할 수 있다.

### 로그 파일 기록 시기

로그 파일은 다음과 같은 작업이 발생될 때 기록된다.

- 트랜잭션 시작시점
- 데이터의 입력, 수정, 삭제 시점
- 트랜잭션 rollback 시점

### 로그 파일 내용

- 로그 파일은 트랜잭션이 발생할 때마다 COMMIT이나 ROLLBACK에 관계없이 모든 내용을 기록한다.
- 트랜잭션 식별
- 트랜잭션 레코드
- 데이터 식별자
- 갱신 이전 값(Before Image)
- 갱신 이후 값(After Image)

■ 로그 파일 보관

로그 파일은 로그 버퍼 내용을 Flat-file 형식으로 기록한다. 이는 데이터베이스가 운영 중에는계속해서 생성된다. 따라서 로그 파일을 저장 매체(테이프)로 영구 보관하는 것이 일반적이다.

### 데이터베이스 복구 알고리즘

트랜잭션 실행 내용인 데이터베이스 버퍼를 저장매체에 동기적으로 기록하는 것을 동기적 갱신(Synchronous I/O)라 하고 트랜잭션이 완료된 내용을 일정 시간이나 작업량에 따라 시간 차이를 두고 데이터베이스 버퍼 내용을 저장 매체에 기록하는 것을 비동기적 갱신(Asynchronous I/O)라 한다. 장애가 발생하여 시스템을 재기동 했을 때는 로그 파일을 이용하여 유효한 트랜잭션 처리 데이터를 복구해야 한다. 따라서 동기적/비동기적 갱신에 의해서 데이터베이스 복구 알고리즘이 분류 된다.

■ NO-UNDO/REDO

데이터베이스 버퍼의 내용을 비동기적으로 갱신한다는 의미는 트랜잭션이 성공적으로 수행되어 완료점에 도달할 때까지 데이터베이스 변경 내용이 기록되지 않는다는 것을 의미한다. 즉, 트랜잭 션이 완료되기 이전에는 변경 내용이 데이터베이스에 기록되지 않으므로 완료되지 않은 트랜잭션 은 취소할 필요가 없다. 그러나 트랜잭션이 완료된 후 데이터베이스 버퍼에 기록되고 저장 매체에 기록되지 않는 상태에서 시스템이 파손되었다면 트랜잭션의 내용을 재실행해야 한다. 이러한 방식 의 알고리즘을 NO-UNDO/REDO 복구 알고리즘이라 한다.

■ UNDO/NO-REDO

데이터베이스 버퍼의 내용을 동기적 갱신하는 경우 트랜잭션이 완료되기 전에 데이터베이스 버퍼 내용을 모두 동시적으로 기록하므로 완료된 트랜잭션들은 어떤 연산도 재실행할 필요가 없다. 그 러나 트랜잭션들이 완료되기 이전에 시스템 파손이 발생할 경우 변경된 내용을 취소해야 한다. 이 러한 방식의 알고리즘을 UNDO/NO-REDO 복구 알고리즘이라 한다.

■ UNDO/REDO

데이터베이스 버퍼의 내용을 동기/비동기적으로 갱신할 경우 모든 갱신이 데이터베이스에 기록되 기 전에 트랜잭션이 완료될 수 있으므로 완료된 트랜잭션이 데이터베이스에 기록되지 못했다면 재 실행을 해야 한다. 이러한 방식의 알고리즘을 UNDO-REDO 복구 알고리즘이라고 한다. 이 복구 방법은 가장 일반적인 기법이지만 가장 복잡한 기법이기도 하다.

■ NO-UNDO/NO-REDO

데이터베이스 버퍼 내용을 동시적으로 저장 매체에 기록하나 데이터베이스와는 다른 영역에 기록 하는 경우이다. 항상 트랜잭션의 실행 상태와 데이터베이스의 내용이 일치하며, 따라서 데이터베 이스 버퍼의 내용을 취소하거나 재실행할 필요가 없다. 이러한 방식의 알고리즘을 NOUNDO/ NO-REDO 복구 알고리즘이라고 한다.

### 백업 종류

데이터베이스 백업은 장애 시 복구를 위한 작업이다. 따라서 복구 수준에 따라서 백업의 종류가 결 정된다. 데이터베이스 백업은 운영체제를 이용한 물리 백업과 DBMS에서 제공한 유틸리티를 이용한 논리 백업으로 나뉜다.

[표 5-2-3] 백업 종류

| 구분 | 설명 | 복구 수준 |
|---|---|---|
| 물리 백업 | 로그 파일 백업 실시 | 완전 복구 |
| | 로그 파일 백업 없음 | 백업 시점까지 복구 |
| 논리 백업 | DBMS 유틸리티 | |

### 데이터베이스 백업 가이드 라인

- 정기적인 Full-backup을 수행한다.
- 데이터베이스 구조적 변화가 생긴 전후 Full-backup을 수행한다.
- 테이블 스페이스의 생성 또는 삭제
- 테이블 스페이스에 데이터 파일을 추가하거나 변경했을 때
- Log 파일을 변경했을 때
- Archive Log Mode로 전환 시 Control 파일만이라도 백업을 실시하며, No-archive log mode로 전환할 때는 Full backup을 수행함
- Read-write 수행이 많은 테이블스페이스는 자주 온라인 백업을 실시함
- 백업 파일은 2본 이상을 보유한다. Incomplete Recovery를 용이하게 함
- 논리 백업은 특정 데이터 또는 특정 테이블 오류 시 복구가 용이함
- 분산 데이터베이스는 동일 모드에서 백업을 수행함
- Read-only 테이블 스페이스는 온라인 백업할 필요가 없다.

성능 개선 방법론 | 조인(Join) | 애플리케이션 성능 개선 | 서버 성능 개선

## 성능 개선 목표

DBMS 성능 개선을 위해 개선 목표를 설정하는 것은 매우 중요하다. 목적에 따라 목표가 다를 수 있으며, 현재 가용한 비용에 대한 효과를 고려하여 목표를 설정하여야 한다.

■ 처리 능력(Throughput)

처리 능력은 해당 작업을 수행하기 위해서 소요되는 시간으로 수행되는 작업량을 나눔으로써 정의 된다. 만약 수행 작업이 트랜잭션이라면 시스템의 처리 능력은 다음과 같다.

처리능력 = 트랜잭션 수 / 시간

처리 능력은 전체적인 시스템 시각에서 측정되고 평가 된다.

■ 처리 시간(Throughput Time)

처리 시간(Throughput time)은 작업이 완료되는 데 소요되는 시간을 의미한다. 처리 시간은 배 치 프로그램의 성능 목표로 설정한다. 대량 배치 작업의 수행 시간을 단축하기 위해서는 다음과 같 은 작업을 고려한다.

- 병행 처리(Parallel Processing)를 실시한다.
- 인덱스 스캔보다 Full 테이블 스캔으로 처리한다.
- Nest-Loop 조인보다 Hash 조인으로 처리한다.
- 대량 작업을 하기 위한 SORT_AREA, HASH_AREA 의 메모리를 확보한다.
- 병목을 없애기 위해서 작업 계획을 한다.
- 대형 테이블인 경우는 파티션으로 생성한다.

■ 응답 시간(Response Time)

응답 시간은 입력을 위해 사용자가 키를 누른 때부터 시스템이 응답할 때까지 시간이다. 최종 사용 자가 느끼는 시스템의 성능 척도이다. 일반적으로 OLTP 시스템에서 성능 지표가 된다. 응답 시간 을 향상하기 위해서는 다음과 사항을 고려한다.

- 인덱스를 이용하여 액세스 경로를 단축한다.
- 부분 범위 처리를 실시한다.
- Sort-Merge 조인이나 Hash 조인을 사용하지 않고 Nest-Loop 조인으로 처리한다.
- Sort-merge 조인이나 해쉬 조인을 사용하지 않고 Nest-Loop 조인으로 처리한다.
- 잠김(Locking) 발생을 억제한다. 예를 들어 시퀀스(Sequence) 오브젝트 이용한다.
- 하드 파싱을 억제한다.

■ 로드 시간(Load Time)

다음날의 비즈니스를 위해 매일 밤 데이터를 로드하거나 시스템을 재구축하고, 목표 시간 내에 데 이터 마이그레이션을 완료해야 한다. 로드 시간은 이와 같은 정기적이거나 비정기적으로 발생되는 데이터베이스에 데이터를 로드하는 작업 수행 시간을 뜻한다. 로드 시간을 단축하려면 다음과 같 은 사항을 고려한다.

- 로그 파일을 생성하지 않는 다이렉트 로드(Direct Load)를 사용한다.
- 병렬 로드 작업을 실시한다.
- DISK IO 경합이 없도록 작업을 분산한다.
- 인덱스가 많은 테이블인 경우는 인덱스를 삭제하고 데이터 로드 후 인덱스를 생성한다.
- 파티션을 이용하여 작업을 단순화한다.

## 성능 개선 절차

데이터베이스 성능 튜닝 방법론은 튜닝 작업에 필요한 여러 가지 수행 방법과 이러한 작업들을 효 율적으로 수행하려는 과정에서 필요한 각각의 단계들을 체계적으로 정리하여 표준화한 것으로써 분 석, 이행, 평가 3단계를 거쳐 성능 최적화의 목적을 달성하기 위한 단계별 접근 전략이다.

### 분석

튜닝 분석 단계에서는 자료 수집과 목표 설정이라는 2단계로 나뉜다.

■ 자료 수집

데이터베이스 모니터링과 데이터베이스 객체 현황 파악 및 물리 설계 요소에 대해 성능과 관련된 지표들을 분석하기 위한 기초 자료를 수집하는 단계이다.

■ 목표 설정

수집된 기초 자료를 통해 데이터 모델 분석, 액세스 패스 분석, 시스템 자원 현황 분석, SQL 성능 분석, SQL 효율 분석 등을 종합하여 성능상에 병목이나 지연 등과 같은 문제 요소 등을 구체적으 로 파악하고 성능 튜닝의 대상이 되는 목표들을 구체화하여 방향을 설정하는 단계이다.

### 이행

튜닝 이행 단계에서는 성능상의 문제 요소로 파악된 대상에 대해 최적화 방안을 수립하고 적용하는 단계이다.

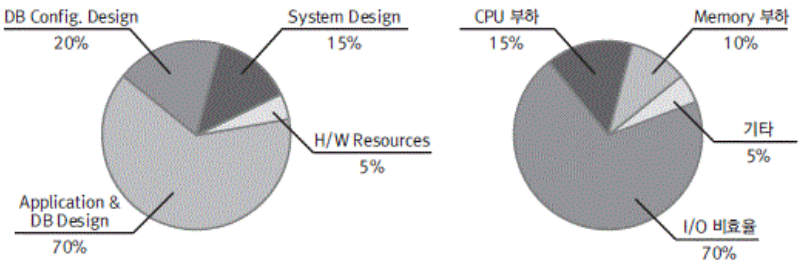
- 데이터베이스 파라미터(Parameter) 조정
- 전략적인 저장 기법 적용을 위한 물리 설계 및 디자인 검토
- 비효율적으로 수행되는 SQL 문에 대한 최적화
- 네트워크 부하 등을 고려한 데이터베이스 분산 구조에 대한 최적화
- 적절한 인덱스 구성 및 사용을 위한 인덱스 설계 등의 최적화 작업

### 평가

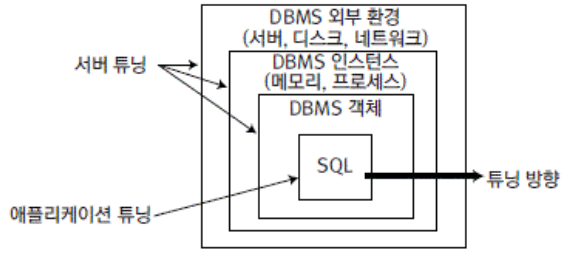
튜닝 평가 단계에서는 분석 단계에서 진단을 통해 분류된 문제 요소들에 대해 설정된 개선 목표와 이행단계에서 구체적인 튜닝 작업을 수행한 후의 성과를 비교 측정하는 단계이다. 튜닝 목표와 튜닝성과에 차이가 있다면 그 요소들을 파악한 후 목표와 성과를 합치시키는 과정을 거친다.

## 성능 개선 접근방법

시스템 성능 문제는 하드웨어(CPU, 메모리, 네트워크 등) 자원 부족, DBMS 설계, SQL 비효율 등의 문제로 발생되는 경우가 대부분이다. 많은 비용을 들여 고성능의 하드웨어 교체 및 증설을 통해 성능상의 문제를 해결하기 이전에 데이터베이스의 성능상 문제점을 파악한 후 문제점의 튜닝을 통한 데이터베이스의 최적화를 우선적으로 고려해야 한다.




[그림 5-3-1] 성능 저하 요인




[그림 5-3-2] 성능 개선 접근 방법

## SQL 성능 분석

SQL이 실행되기 위해서는 실행 가능한 소스코드로 변환되어야 하는데, 이 소스코드를 생성하기 위한 실행 전략을 실행

계획이라 한다. 옵티마이저는 실행 계획을 수립하기 위해 SQL을 관계 대수 형태로 변환하고 개별 연산자의 알고리즘들을 다양한 방법으로 결합해 이들을 트리 형태로 표현한다.

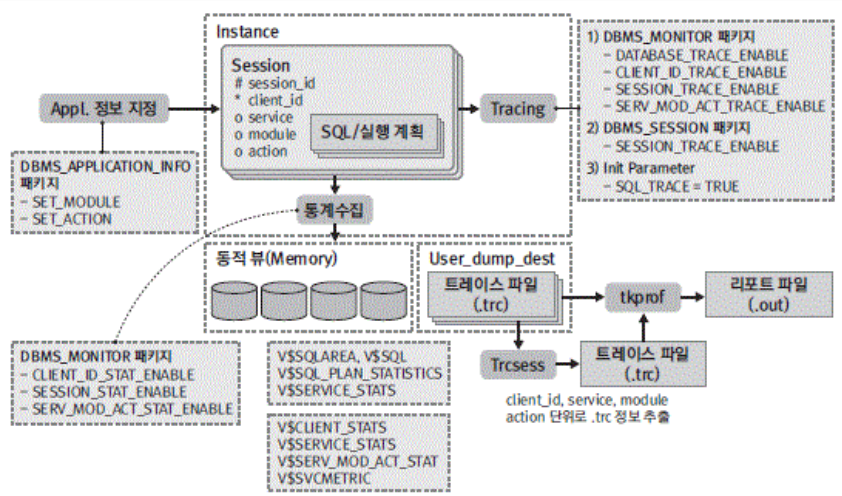
실행 계획이 수립되면 최적화 과정을 거쳐 최적의 실행 계획이 선택되지만 옵티마이저의 지능적 한계로 인해 최종 선택된 실행 계획이 항상 최적의 실행 계획이 아닐 수 있게 된다.

실행 계획 분석이란, 이러한 옵티마이저의 한계를 인식하고 옵티마이저가 최적화된 실행 계획을 수립할 수 있도록 유도하는 데 목적이 있하는 방법은 DBMS별로 약간 차이가 있지만 상용 혹은 DBMS 벤더가 제공하는 SQL 개발 도구를 사용하면 쉽게 실행 계획 보기를 수행할 수 있다.

■ SQL 트레이스 분석

오라클은 데이터베이스의 인스턴스 또는 세션 단계에서 수행되는 모든 SQL의 통계치 및 대기 이벤트에 대한 정보를 수집해주는 트레이스 기능을 제공하고 있다. 초기 SQL 트레이스 기능은 주로 Instance, Session 수준에서 이루어졌다. 그런데 최근부터는 End to End Application Tracing이 가능하도록 Client Identifier, Service, Module, Action 수준의 트레이스 기능을 추가하였는데, 이로 인해 cross-instance, multi-session, multi-user 수준의 SQL 트레이스가 가능하게 되었다.

다음은 오라클 SQL 트레이스 기능을 요약하여 도식화한 것이다.




[그림 5-3-3] 오라클 SQL 트레이스 기능

오라클에서 트레이스 기능이 활성화되면 트레이스 결과 파일은 세션 단위로 지정 디렉토리에 .trc 확장자로 기록된다. 트레이스 파일은 로그와 같이 알기 힘든 약어들이 많이 사용되어 해석이 용이하 지 않다. 그래서 tkprof 유틸리티를 사용해 분석하기 용이한 형태로 변환해 사용한다.

리포트 파일은 여러 개의 세션 트레이스 파일들을 머지하여 하나의 리포트 파일로 생성하며, 헤드 부분, 바디 부분, 써머리 부분의 3부분으로 나눠져 있다.

다음은 헤드 부분에 대한 사례이다.

```
TKPROF: Release 10.2.0.2.0 - Production on Fri Oct 9 15:12:282009
Copyright (c) 1982, 2005, Oracle.  All rights reserved.
Trace file: oraods_ora_4709.trc
Sort options: default
********************************************************************************
count    = number of times OCI procedure was executed
cpu      = cpu time in seconds executing
elapsed  = elapsed time in seconds executing
disk     = number of physical reads of buffers from disk
query    = number of buffers gotten for consistent read
current  = number of buffers gotten in current mode (usually for update)
rows     = number of rows processed by the fetch or execute call
********************************************************************************
```

헤드 부분은 릴리즈 정보, 리포트를 생성한 시간, 저작권 문구, 트레이스 파일명, 리포트 생성 옵션 등의 정보를 포함하고 있다.

다음은 바디 부분에 대한 사례이다.

```
********************************************************************************
select a.relno from sn_ap010 a
where a.rs_doc_cd = '0012501'  and not exists (select a.relno
  from sn_ap010 b where a.relno = b.rsno)

   call   count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ----------  ----------  ----------  ----------
  Parse        1      0.02       0.01          0          0          0           0
Execute        1      0.00       0.00          0          0          0           0
  Fetch     2521     92.84     118.47     759544    3056433          0      252006
------- ------  -------- ---------- ----------  ----------  ----------  ----------
  total     2523     92.86     118.49     759544    3056433          0      252006

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 26

Rows     Row Source Operation
-------  ---------------------------------------------------
252006  FILTER  (cr=3056433 pr=759544 pw=0 time=75357802 us)
833802  TABLE ACCESS FULL TB_AP010 (cr=735085 pr=732538 pw=0 time=82554476 us)
580070  INDEX RANGE SCAN IX_AP010_01 (cr=2321348 pr=27006 pw=0 time=28631927 us)
********************************************************************************
SELECT A.RCPTNO, A.IR_INTNL_APPLNO, B.INTNL_APPLNO
FROM SN_IR001 A, SN_ES201 B
WHERE A.RCPTNO = B.RCPTNO AND A.IR_INTNL_APPLNO != B.INTNL_APPLNO

   call   count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ----------  ----------  ----------  ----------
  Parse        1      0.02       0.01          0          0          0           0
Execute        1      0.00       0.00          0          0          0           0
  Fetch        1    239.03     322.00    1482520    1327375          2           0
------- ------  -------- ---------- ----------  ----------  ----------  ----------
  total        3    239.05     322.02    1482520    1327375          2           0

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: 26

Rows     Row Source Operation
-------  ---------------------------------------------------
      0  HASH JOIN  (cr=1327375 pr=1482520 pw=156829 time=322000741 us)
14840320  TABLE ACCESS FULL TB_ES201 (cr=792364 pr=792248 pw=0 time=89049740 us)
14983938  TABLE ACCESS FULL TB_IR001 (cr=535011 pr=533443 pw=0 time=74924847 us)
********************************************************************************
```

바디 부분은 머지된 모든 트레이스 파일 각각의 실행 통계와 이벤트 관련 정보들을 포함하고 있다.

■ SQL Statement

수행된 쿼리문 텍스트(Embeded /Static 쿼리문의 경우 포맷이 일부 변경)

■ Execution statistics

DBMS가 결과 집합을 만들기 위해 내부적으로 Parse, Execute, Fetch 단계에서 수행한 수행 횟 수, cpu 사용시간, 소요시간, 물리적으로 읽은 블록 수(disk), 논리적으로 읽은 블록 수(query+ current), 처리한 로우 건 수 등에 대한 정보들을 포함하고 있다.

■ Misses in library cache during parse : 1

실행 통계 정보에 표시된 Execute count를 수행하는 동안 하드 파싱이 일어난 횟수를 의미한다. 위 사례에서는 1회의 하드 파싱이 발생하였다.

■ Execution statistics

옵티마이져 모드는 옵티마이져가 최적화 목표를 어디에 두고 최적화를 수행해야 하는지 최적화 접 근 방법에 대한 선택 옵션으로 first_rows_n, first_rows, all_rows 등의 옵션이 지원된다.

- first_rows_n
  최초의 n개 로우를 가장 빠르게 리턴하는 것을 최적화 목표로 하여 최적화를 수행한다.
- first_rows
  최초의 몇몇 로우들을 가장 빠르게 리턴하기 위한 최적의 플랜을 찾는 것을 최적화 목표로 한다.
-

- all_rows

최적의 throughput을 목표로 최적화를 수행한다.

■ Parsing user id: 26

SQL을 실행한 사용자의 id를 표시하는 부분으로, id에 대한 명을 확인하려면 dba_users 테이블을 참조하여야 한다.

■ Execution plan

SQL을 처리하기 위한 실행 계획을 Row Source Operation 단위의 정보와 함께 표시한다.

- Rows : 해당 오퍼레이션 단계에서 리턴된 로우 수
- cr은 consistent reads로 논리적으로 읽은 블록 수를 의미
- pr은 physical reads로 물리적으로 읽은 블록 수를 의미
- pw는 physical writes로 물리적으로 쓴 블록 수를 의미
- time은 micro-sec 단위의 소요 시간을 의미

다음은 써머리 부분에 대한 사례이다.

```
OVERALL TOTALS FOR ALL NON-RECURSIVE STATEMENTS

   call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ----------  ----------  ----------  ----------
  Parse       19      1.09       1.07          0          0          0           0
Execute      20      0.00       0.00          0          0          0           0
  Fetch    5125      3735.50    5327.07   16701710   43370685         67      511155
------- ------  -------- ---------- ----------  ----------  ----------  ----------
  total    5164      3736.59    5328.15   16701710   43370685         67      511155

Misses in library cache during parse: 19
Misses in library cache during execute: 1

OVERALL TOTALS FOR ALL RECURSIVE STATEMENTS

   call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ----------  ----------  ----------  ----------
  Parse      286      0.58       0.54          0          0          0           0
Execute 474873      15.27      13.15          0          0          0           0
  Fetch  923705      26.73      43.56      11752    1789229          0      462998
------- ------  -------- ---------- ----------  ----------  ----------  ----------
  total 1398864      42.58      57.25      11752    1789229          0      462998

Misses in library cache during parse: 54
Misses in library cache during execute: 52

    21  user  SQL statements in session.
  1682  internal SQL statements in session.
  1703  SQL statements in session.
********************************************************************************
Trace file: oraods_ora_4709.trc
Trace file compatibility: 10.01.00
Sort options: default

     1  session in tracefile.
    21  user  SQL statements in trace file.
  1682  internal SQL statements in trace file.
  1703  SQL statements in trace file.
    57  unique SQL statements in trace file.
1415437  lines in trace file.
 50285  elapsed seconds in trace file.
```

써머리 부분은 Tkprof로 변환한 모든 트레이스 파일에 대한 실행 통계를 누적한 부분으로, 사용 자가 요청한 SQL을 직접적으로 처리하기 위해 수행한 실행 통계는 NON-RECURSIVE에 집계되 고, 추가적으로 수행한 실행 통계는 RECURSIVE에 집계된다.

실행 통계 정보는 패턴에 따라 다음과 같은 분석이 가능하다.

| | count | 분석 |
|---|---|---|
| Parse<br>Execute<br>Fetch | 1<br>1<br>1 | .동일 SQL에 조건만 다른 SQL 반복을 사용하거나 Dynamic SQL 형태로 개발된 SQL일 가능성이 높다.<br>.DBMS에 심각한 Parsing 부하를 가중시킬 수 있다.<br>.Trace TKPROF File size가 과대하게 될 수 있다. |
| | | |

| | | |
|---|---|---|
| Parse<br>Execute<br>Fetch | 100<br>100<br>100 | .프로그램 100번 수행 : 프로그램이 여러 번 실행되면 SQL을 파싱하지 않 고 Shared SQL AREA에서 찾아 왔더라도 Parse 횟수가 증가한다.<br>.SQL 100번 수행되었는데, Parse가 100이라는 것은 프로그램이 100번 수행??복 수 행되었다는 것을 의미한다. |
| Parse<br>Execute<br>Fetch | 1<br>100<br>100 | .프로그램 1번 수행 : 애플리케이션이 여러 번 실행되면 SQL을 파싱하지 않고 Shared SQL AREA에서 찾아 왔더라도 Parse 횟수가 증가한다.<br>.SQL 100번 수행되었는데, Parse가 1이라는 것은 PL/SQL 내에서 반복 수행된 SQL이거나 프로그램(Pro*c)에서 Hold Cursor 지정하여 사용했 다는 것을 의미한다. |
| Parse<br>Execute<br>Fetch | 1<br>1<br>100 | .SQL은 1번 수행되었고 패치만 100번 수행했다는 것은 CURSOR로 지정된 SQL이 LOOP 내에서 반복 Fetch 처리를 수행하였다는 것을 의미한다. |
| Parse<br>Execute<br>Fetch | 200<br>100<br>10000 | .Parse count가 Execute count보다 큰 것은 SQL이 파싱만 되고 수행되 지 못한 경우로, 개발 시 오류로 인하여 실행이 취소되거나 시스템 오버헤 드로 인하여 SQL이 실행되지 못한 경우에 발생할 수 있다.<br>.SQL이 100번 수행될 때 패치를 10000번 수행했다는 것은 CURSOR로 지정 된 SQL이 LOOP 내에서 반복 Fetch 처리를 수행하였다는 것을 의미한다. |

| | count | rows | 분석 |
|---|---|---|---|
| Parse<br>Execute<br>Fetch | 2500<br>2500<br>0 | 0<br>9800<br>0 | .HOLD_CURSOR를 지정하지 않고 3GL처럼 INSERT, UPDATE, DELETE가 반복 수행되는 구조의 프로그램 |
| Parse<br>Execute<br>Fetch | 1<br>100<br>100 | 0<br>0<br>200 | .다중 처리로 수행된 SQL<br>.한 번에 2 Rows가 Fetch 처리 |
| Parse<br>Execute<br>Fetch | 1<br>1<br>100 | 0<br>0<br>200 | .다중 처리로 수행된 SQL<br>.한 번에 2 Rows가 Fetch 처리 |

| 구분 | 분석 항목 | 분석 내용 |
|---|---|---|
| 일반 | SQL당 수행 시간 | .total cpu / execute count [0.01초 이내]<br>.total elapsed / execute count [0.03초 이내] |
| | Parse Overhead | .parse elapsed / execute elapsed [10% 이내]<br>.Misses in library cache during parse<br>.parse count * 0.01보다 parse cpu time이 크면 Dynamic SQL 이 많이 사용되었거나 라이브러리 캐시가 적게 지정된 경우임<br>※ 위의 결과로 Dynamic SQL의 사용 비율을 추정할 수 있음 |
| | Block I/O 비효율 | .total disk / total rows [1미만의 수치일수록 양호한 상태]<br>.Memory Block Access = total query + total current<br>.Memory Hit Ratio = Memory Block Access / (total disk + Memory Block Access) [90% 이상]<br>.Execute vs. Fetch의 비율<br>- 처리된 row의 절대량이 많지 않은데, DB Block I/O가 많은 경우는 Optimizing 전략 부재일 가능성이 크고,<br>- 처리된 row의 절대량이 많으면서 DB Block I/O가 많은 경우 DB Block Buffer Hit Ration가 낮으면 DB Block Buffer 크기의 문제일 가능성이 크다고 판단 가능 |
| | Total elapsed / Total cpu | .100%보다 지나치게 크게 나타나면 Optimizing 전략 부재로 인한 Full Scan 등으로 Disk I/O 병목 현상에 의한 것인 가능성이 큼 |
| 응용 PGM 문제 Pattern | fetch rows / fetch count | .1보다 큰 특정 10의 배수 수치에 가깝게 나타나는 경우 애플리케 이션에 Array Processing이 적용되고 있다고 판단할 수 있으 면 1에 가까운 수치로 나타나는 경우 Array Processing이 적용 되지 않았다고 판단할 수 있음 |
| | RECURSIVE CALL OVERHEAD | .OVERALL TOTALS FOR ALL RECURSIVE STATEMENTS 의 execute count / OVERALL TOTALS FOR ALL NONRECURSIVE STATEMENTS Execute의 비율<br>.애플리케이션 내에서 DB Function 사용 비율을 판?? 여부를 조사할 필요가 있음 |
| 악성 SQL의 분포 추정 | | .TKPROF utility에 의해 처음 생성된 파일의 크기와 Full scan 및 cputime에 의해 추출된 결과 파일의 크기를 비교하여 전체 SQL대비 악성 SQL의 비율을 추정할 수 있음 |

### ◘ 실행 계획 분석 사례

1) 오라클 실행 계획 분석 사례 1

**개선 전 SQL과 실행 계획**

```
SELECT B.사절고객PK
     ...
  FROM    위험고객기본 A,   사절고객정보 B,    보험계약기본 C,    계약선택경로정보 D
 WHERE    A.위험고객번호      LIKE :1    -- 주민번호만 있을 경우
 AND A.   위험고객성명        LIKE :2     -- 성명만 있을 경우
 AND A.   위험고객고객번호    <>          '111111111111'
 AND A.   위험고객PK          =           B.위험고객PK
 AND B.   사용여부            =           '1'
 AND B.   증권번호            =           C.증권번호(+)
 AND B.   사절고객PK          =           D.사절고객PK(+)
 AND D.   위험입력경로코드(+)  =           '2'
 AND D.   계약관계구분코드(+)  =           '21'
 AND D.   삭제여부(+)          =           '0'
 AND D.   최종내역여부(+)      =           '1'
ORDER BY TO_NUMBER(B.사절고객PK)

Call     Count       CPU Time    Elapsed Time       Disk      Query    Current        Rows
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
Parse        1      0.000       0.002          0          0          0           0
Execute      1      0.020       0.041          0          0          0           0
Fetch        1     19.590      87.693      87947      95221          0           0
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
Total        3     19.610      87.736      87947      95221          0           0

Elapsed Time for Client(sec.): 87.735
Misses in library cache during parse: 1
Misses in library cache during execute: 1

  Rows  Row Source Operation
------  ---------------------------------------------------------------------------
     0  STATEMENT
     0   SORT ORDER BY (cr=95221 pr=87947 pw=0 time=87693218 us)
     0    NESTED LOOPS OUTER (cr=95221 pr=87947 pw=0 time=87693182 us)
     0     HASH JOIN RIGHT OUTER (cr=95221 pr=87947 pw=0 time=87693177 us)
     2      TABLE ACCESS FULL 계약선택경로정보 (cr=7310 pr=7301 pw=0 time=3988417 us)
     0      HASH JOIN (cr=87911 pr=80646 pw=0 time=83697681 us)
     1       TABLE ACCESS BY INDEX ROWID 위험고객기본 (cr=51022 pr=43769 pw=0 time=73795000 us)
     1        INDEX RANGE SCAN 위험고객기본_IX_03 (cr=51021 pr=43768 pw=0 time=73792623 us)
2454718       TABLE ACCESS FULL 사절고객정보 (cr=36889 pr=36877 pw=0 time=7377160 us)
     0      TABLE ACCESS BY INDEX ROWID 보험계약기본 (cr=0 pr=0 pw=0 time=0 us)
     0       INDEX UNIQUE SCAN 보험계약_PK (cr=0 pr=0 pw=0 time=0 us)(Object ID 251964)
```

위의 실행 계획을 보면 위험고객기본 테이블과 사절고객정보 테이블이 해쉬 조인으로 수행되고, 그 결과를 가지고 계약선택경로정보 테이블과 RIGHT OUTER 해쉬 조인을 수행하였다. 그리고 다 시 보험계약기본정보 테이블과 Nested-Loop 조인을 수행한 후 그 결과를 가지고 그룹핑을 하였다.

첫 번째 해쉬 조인에서 위험고객기본_IX_03을 RANGE SCAN하여 1건이 추출되었다. 그런데 DISK I/O가 43,769 블록으로 매우 크게 나타나고 있는 것으로 보아 매우 큰 범위를 RANGE SCAN하였다는 것을 알 수 있다. 그리고 1건이 추출되었다는 것은 분포도가 아주 좋은 체크 조건을 가지고 있다는 의미이기도 하다.

SQL을 보면 위험고객기본 테이블에 인덱스를 사용할 수 있는 두 가지 상수 조건이 주어져 있다. 그런데 두 조건 모두 Like 조건으로 비교되었으므로 하나를 사용하면 다른 하나는 체크 조건으로만 사용이 가능하다. 그런데 위 실행 계획에서는 두 조건 중 값이 입력되지 않은 조건을 드라이빙 조건으 로 사용하고 값이 입력된 조건을 체크 조건으로 사용해 위와 같은 비효율이 발생한 것으로 판단된다.

이 비효율을 개선하기 위해서는 이 두 조건을 UNION ALL로 분리하여 값이 입력되는 조건에 따 라 인덱스를 선택적으로 사용할 수 있도록 하여야 한다.

두 번째 비효율은 사절고객정보를 해쉬 조인으로 수행한 부분이다. 해쉬 조인의 경우 먼저 스스로 처리 범위를 줄인 후 조인에 참여하므로 대용량 데이터의 조인 시에는 문제가 되지 않지만 위의 경우 처럼 조인의 대상 집합이 1건인 경우에는 더 많은 비효율이 발생하게 된다.

이 비효율을 개선하기 위해서는 위험고객기본 테이블을 차례로 액세스하면서 그 결과를 가지고 사 절고객정보 테이블을 탐침할 수 있도록 Nested-Loop 조인으로 조인 방식을 변경하여야 한다. 마지막 비효율은 계약선택경로 테이블과의 조인 방식에 대한 부분이다. 계약선택경로 테이블의 경 우 코드성 테이블로 비교적 적은 집합이라 해쉬 조인으로 인한 비효율이 크게 나타나지 않았다. 하지 만 선행 집합이 아주 작은 집합이므로 선행 집합의 결과로 조인을 수행하는 Nested-Loop 조인으로 변경을 해야 한다.

아래의 SQL과 실행 계획은 위와 같은 관점에서 최적화를 수행한 결과이다.

**개선 후 SQL과 실행 계획**

```
SELECT /*+ NO_MERGE(A) LEADING(A) USE_NL(B C D) */
    B.사절고객PK
    ...
 FROM (SELECT /*+ INDEX(A (위험고객번호)) */  A.*
       FROM 위험고객기본 A
       WHERE 위험고객번호 <> '%'
       UNION ALL
       SELECT /*+ INDEX(A (위험고객성명)) */  A.*
       FROM 위험고객기본 A
       WHERE 위험고객번호 = '%'
     ) A,          --위험고객기본
             사절고객정보 B,    보험계약기본 C,    계약선택정로정보 D
WHERE A.     위험고객번호        LIKE  :1    -- 주민번호만 있을 경우
  AND A.     위험고객성명        LIKE  :2    -- 성명만 있을 경우
  AND A.     위험고객고객번호      <>          '111111111111'
  AND B.     위험고객PK          =           A.위험고객PK
  AND B.     사용여부            =           '1'
  AND C.     증권번호(+)         =           B.증권번호
  AND D.     사절고객PK(+)       =           B.사절고객PK
  AND D.     위험입력경로코드(+)   =           '2'
  AND D.     계약관계구분코드(+)   =           '21'
  AND D.     삭제여부(+)         =           '0'
  AND D.     최종내역여부(+)      =           '1'
ORDER BY TO_NUMBER(B.사절고객PK)
```

| Call | Count | CPU Time | Elapsed Time | Disk | Query | Current | Rows |
|---|---|---|---|---|---|---|---|
| Parse | 1 | 0.000 | 0.002 | 0 | 0 | 0 | 0 |
| Execute | 1 | 0.030 | 0.045 | 0 | 0 | 0 | 0 |
| Fetch | 1 | 0.010 | 0.036 | 5 | 7 | 0 | 0 |
| Total | 3 | 0.040 | 0.082 | 5 | 7 | 0 | 0 |

Elapsed Time for Client(sec.): 0.082
Misses in library cache during parse: 1
Misses in library cache during execute: 1

```
  Rows  Row Source Operation
-----  -------------------------------------------------------------------
     0  STATEMENT
     0   SORT ORDER BY (cr=7 pr=5 pw=0 time=35609 us)
     0    NESTED LOOPS OUTER (cr=7 pr=5 pw=0 time=35562 us)
     0     NESTED LOOPS OUTER (cr=7 pr=5 pw=0 time=35558 us)
     0      NESTED LOOPS  (cr=7 pr=5 pw=0 time=35553 us)
     1       VIEW  (cr=4 pr=3 pw=0 time=16057 us)
     1        UNION-ALL  (cr=4 pr=3 pw=0 time=16050 us)
     1         TABLE ACCESS BY INDEX ROWID 위험고객기본 (cr=4 pr=3 pw=0 time=16019 us)
     1          INDEX RANGE SCAN 위험고객기본_IX_02 (cr=3 pr=2 pw=0 time=14137 us)
     0         FILTER  (cr=0 pr=0 pw=0 time=12 us)
     0          TABLE ACCESS BY INDEX ROWID 위험고객기본 (cr=0 pr=0 pw=0 time=0 us)
     0           INDEX RANGE SCAN 위험고객기본_IX_03 (cr=0 pr=0 pw=0 time=0 us)
     0        TABLE ACCESS BY INDEX ROWID 사절고객정보 (cr=3 pr=2 pw=0 time=19489 us)
     0         INDEX RANGE SCAN 사절고객정보_IX_01 (cr=3 pr=2 pw=0 time=19481 us)
     0       TABLE ACCESS BY INDEX ROWID 계약선택정로정보 (cr=0 pr=0 pw=0 time=0 us)
     0        INDEX RANGE SCAN 계약선택정로정보_IX_04 (cr=0 pr=0 pw=0 time=0 us)
     0      TABLE ACCESS BY INDEX ROWID 보험계약기본 (cr=0 pr=0 pw=0 time=0 us)
     0       INDEX UNIQUE SCAN 보험계약기본_PK (cr=0 pr=0 pw=0 time=0 us)
```

2) 오라클 실행 계획 분석 사례 2

```
SELECT a.증권번호, b.그룹증권번호
  FROM 케이스기본 a, 보험계약기본 b
WHERE a.        증권번호          =          b.증권번호
  AND a.        최종내역여부      =          '1'
  AND a.        삭제여부          =          '0'
  AND a.        케이스처리구분코드 =          'N'
  AND b.        개별그룹구분코드  =          '2'
  AND b.        그룹증권번호      =          :b

Call     Count       CPU Time    Elapsed Time       Disk      Query    Current        Rows
-------  ------  --------  ------------  ------  --------  --------  --------
Parse        1      0.000         0.000       0         0         0           0
Execute      1      0.000         0.003       0         0         0           0
Fetch       67      0.150         4.591     696      2288         0         657
-------  ------  --------  ------------  ------  --------  --------  --------
Total        69      0.150         4.594     696      2288         0         657

··· "내용 일부 생략"

  Rows  Row Source Operation
------  ---------------------------------------------------------------------------
     0  STATEMENT
   657   TABLE ACCESS BY INDEX ROWID 케이스기본  (cr=2288 pr=696 pw=0 time=5551452
  1315    NESTED LOOPS  (cr=1631 pr=48 pw=0 time=3490005 us)
   657     TABLE ACCESS BY INDEX ROWID 보험계약기본  (cr=182 pr=45 pw=0 time=26544 us)
   657       INDEX RANGE SCAN 보험계약기본_IX_02  (cr=71 pr=5 pw=0 time=22778 us)(Object
   657     INDEX RANGE SCAN 케이스기본_IX_01 (cr=1449 pr=3 pw=0 time=29502 us)(Obj
```

위의 실행 계획을 보면 케이스기본 테이블 액세스에 대부분의 시간이 소요되고 있는 것을 알 수 있다 (주의! 위 사례에서는 오퍼레이션 수준 통계에서 보이는 5.55초, 즉“time=5551452 us”는 쿼리 수행 통계 부분의 4.594초와 불일치하지만 대부분의 소요 시간이 케이스기본 테이블 액세스에서 발생했다 는 사실은 변함없다).

케이스기본 테이블을 657회 탐침하면서 651번의 물리 블록 읽기(pr 696 – pr 45)가 발생하였다는 것은 읽고자 하는 657개의 데이터 로우 각각이 서로 다른 데이터 블록에 저장되어 있다는 의미이다. 이 때문에 처리 시간의 대부분이 651번의 물리 블록 읽기에서 소요되었다. 개선을 위해서는 케이스기본 테이블 로우가 동일한 그룹증권번호로 클러스터링될 수 있도록 테이블 을 정렬하는 것이다.

3) MS-SQL 실행 계획 분석 사례 1 : 스칼라 서브쿼리에서의 비효율

**개선 전 SQL과 실행 계획**

```
select ...
        (
        Select isnull (max (bar_code), 0)
        From mBarItm
        Where bar_code like m.lot_no + '%'
        ) as itm_bar
        ...
from    mBarMst m,
        xUntMst u,
        mItmMst im,
        xCstMst cm,
        xZipMst z
where ...

'mBarItm'   테이블. 스캔 수 3,   논리적 읽기 수 4076,   물리적 읽기 수 0,   미리 읽기 수 0.
 'xZipMst'  테이블. 스캔 수 1,   논리적 읽기 수    3,   물리적 읽기 수 0,   미리 읽기 수 0.
 'xCstMst'  테이블. 스캔 수 1,   논리적 읽기 수    3,   물리적 읽기 수 0,   미리 읽기 수 0.
'mItmMst'   테이블. 스캔 수 1,   논리적 읽기 수    3,   물리적 읽기 수 0,   미리 읽기 수 0.
'xUntMst'   테이블. 스캔 수 1,   논리적 읽기 수    2,   물리적 읽기 수 0,   미리 읽기 수 0.
'mBarMst'   테이블. 스캔 수 1,   논리적 읽기 수    2,   물리적 읽기 수 0,   미리 읽기 수 0.

SQL Server 실행 시간:
   CPU 시간 = 282ms, 경과 시간 = 1765ms.
Rows Exes StmtText
---------- ---------- ----------------------------------------------------------------------------
    1     1 select m.itm_kind,
    1     1  |--Compute Scalar(DEFINE:([Expr1012]=isnull([Expr1010], '0'), [Expr1002]=[Expr1002], [Expr1019
    1     1       |--Nested Loops(Left Outer Join, OUTER REFERENCES:([m].[bar_seq], [m].[bar_date]))
    1     1            |--Nested Loops(Inner Join, OUTER REFERENCES:([u].[zip_no]))
    1     1            |    |--Nested Loops(Inner Join)
    1     1            |    |    |--Stream Aggregate(DEFINE:([Expr1002]=MAX([mBarItm].[bar_no])))
    1     1            |    |    |    |--Top(1)
    1     1            |    |    |    |--Index Scan(OBJECT:([b2en].[dbo].[mBarItm].[IX_mBarItm_02]), O
    1     1            |    |    |--Nested Loops(Inner Join, OUTER REFERENCES:([m].[lot_no]))
    1     1            |    |         |--Nested Loops(Left Outer Join, OUTER REFERENCES:([m].[cst_code]))
    1     1            |    |         |    |--Nested Loops(Left Outer Join, OUTER REFERENCES:([m].[itm_code
    1     1            |    |         |    |    |--Nested Loops(Inner Join, OUTER REFERENCES:([m].[unt_cd])
    1     1            |    |         |    |    |    |--Clustered Index Seek(OBJECT:([b2en].[dbo].[mBarMst]
    1     1            |    |         |    |    |    |--Clustered Index Seek(OBJECT:([b2en].[dbo].[xUntMst]
    1     1            |    |         |    |    |--Clustered Index Seek(OBJECT:([b2en].[dbo].[mItmMst].[PK_
    1     1            |    |         |    |--Clustered Index Seek(OBJECT:([b2en].[dbo].[xCstMst].[PK_xCstM
    1     1            |    |         |--Stream Aggregate(DEFINE:([Expr1010]=MAX([mBarItm].[bar_code])))
    0     1            |    |              |--Top(1)
    0     1            |    |                   |--Index Scan(OBJECT:([dbo].[mBarItm].[IX_mBarItm_01]
    1     1            |    |--Clustered Index Seek(OBJECT:([b2en].[dbo].[xZipMst].[PK_xZipMst] AS [z]), SE
    1     1            |--Hash Match(Cache, HASH:([m].[bar_seq], [m].[bar_date]), RESIDUAL:([m].[bar_seq]=[
    1     1                 |--Compute Scalar(DEFINE:([Expr1017]=Convert([Expr1030])))
    1     1                      |--Stream Aggregate(DEFINE:([Expr1030]=Count(*)))
    1     1                           |--Clustered Index Seek(OBJECT:([b2en].[dbo].[mBarItm].[PK_mBarItm]),
```

위 실행 계획은 from절에서 테이블 5개를 조인하고 select-list에서 스칼라 서브쿼리 3개를 기술한 SQL의 트레이스 결과다. 오라클에서는 스칼라 서브쿼리가 메인 쿼리의 실행 계획 맨 위에 별도로 표 시되지만 SQL 서버에서는 일반적인 조인 형태로 표현된다.

트레이스 파일의 상단에는 테이블별 스캔 수와 페이지 읽기 수가 표시되는데, 위 트레이스 파일에서 는 mBarItm 테이블의 페이지 읽기 수가 매우 큰 것을 알 수 있다. 트레이스 파일의 아래에서 6번째 줄을 보면 IX_mBarItm_01 인덱스를 Index Scan 방식으로 액세스해 0건을 출력했다. 이 부분에 별 다른 비효율이 없어 보이지만 SQL 서버에서 Index Scan 연산은 해당 인덱스를 처음부터 끝까지 스캔 하는 것을 의미하므로 인덱스를 전체 스캔해 결과가 0건이라는 의미이다.

Nested Loop 조인 시 Inner 집합의 조인 속성에 인덱스가 없으면 Outer 집합에서 출력된 건 수만 큼 반복해서 Inner 집합을 전체 스캔해야 하기 때문에 많은 비효율이 발생한다. 그런데 다행히 outer 집합에서 1건이 출력되었으므로 IX_mBarItm_01 인덱스를 1번만 전체 스캔하였다.

비효율을 개선하려면 스칼라 서브쿼리에서 IX_mBarItm_01 인덱스를 Index Seek(* 오라클의 index range scan 또는 index unique scan과 같음) 방식으로 액세스하도록 SQL을 개선해야 한다.

다음은 mBarItm에 생성된 인덱스 정보이다.

```
reate index IX_mBarItm_01 on mBarItm (bar_code);

create index IX_mBarItm_02 on mBarItm (bar_no);
```

개선 전 SQL에서 메인 쿼리가 먼저 수행되면 메인 쿼리의 m.lot_no는 상수가 되기 때문에 스칼라 서 브쿼리는 where b

ar_code like [상수값]||'%'형태의 검색 조건으로 대체되므로 인덱스 IX_mBarItm_01을 사용해 수행되어야 한다. 그런데 우리가 원하는 방식으로 실행되지 않았다.

옵티마이저가 우리가 원하는 액세스 패스를 가지도록 유도하기 위해 논리적으로 동일한 결과를 가지는 m.lot_no is not null이라는 dummy 조건을 추가하였다. 개선 후 실행 계획에서 옵티마이저가 인덱스 IX_mBarItm_01을 사용하여 정상적으로 액세스하고 있는 것을 확인할 수 있다.

**개선 후 SQL과 실행 계획**

```
select ...
       (
       Select isnull (max (bar_code), 0)
       From mBarItm
       Where bar_code like m.lot_no + '%'
       and m.lot_no is not null --▶ dummy 조건 추가
       ) as itm_bar
       ...
from   mBarMst m,
       xUntMst u,
       mItmMst im,
       xCstMst cm,
       xZipMst z
where  ...
'mBarItm'  테이블. 스캔 수 3,   논리적 읽기 수 4076,   물리적 읽기 수 0,   미리 읽기 수 0.
 'xZipMst'  테이블. 스캔 수 1,   논리적 읽기 수    3,   물리적 읽기 수 0,   미리 읽기 수 0.
 'xCstMst'  테이블. 스캔 수 1,   논리적 읽기 수    3,   물리적 읽기 수 0,   미리 읽기 수 0.
'mItmMst'  테이블. 스캔 수 1,   논리적 읽기 수    3,   물리적 읽기 수 0,   미리 읽기 수 0.
'xUntMst'  테이블. 스캔 수 1,   논리적 읽기 수    2,   물리적 읽기 수 0,   미리 읽기 수 0.
'mBarMst'  테이블. 스캔 수 1,   논리적 읽기 수    2,   물리적 읽기 수 0,   미리 읽기 수 0.

SQL Server 실행 시간:
   CPU 시간 = 282ms, 경과 시간 = 1765ms.
Rows Exes StmtText
---- ---- ----------------------------------------------------------------
   1    1 select m.itm_kind,
   1    1  |--Compute Scalar(DEFINE:([Expr1012]=isnull([Expr1010], '0'), [Expr1002]=[Expr1002], [Expr1019
   1    1       |--Nested Loops(Left Outer Join, OUTER REFERENCES:([m].[bar_seq], [m].[bar_date]))
   1    1            |--Nested Loops(Inner Join, OUTER REFERENCES:([u].[zip_no]))
   1    1            |    |--Nested Loops(Inner Join)
   1    1            |    |    |--Stream Aggregate(DEFINE:([Expr1002]=MAX([mBarItm].[bar_no])))
   1    1            |    |    |    |--Top(1)
   1    1            |    |    |    |--Index Scan(OBJECT:([b2en].[dbo].[mBarItm].[IX_mBarItm_02]), O
   1    1            |    |    |--Nested Loops(Inner Join, OUTER REFERENCES:([m].[lot_no]))
   1    1            |    |         |--Nested Loops(Left Outer Join, OUTER REFERENCES:([m].[cst_code])
   1    1            |    |         |    |--Nested Loops(Left Outer Join, OUTER REFERENCES:([m].[itm_code
   1    1            |    |         |    |    |--Nested Loops(Inner Join, OUTER REFERENCES:([m].[unt_cd])
   1    1            |    |         |    |    |    |--Clustered Index Seek(OBJECT:([b2en].[dbo].[mBarMst]
   1    1            |    |         |    |    |    |--Clustered Index Seek(OBJECT:([b2en].[dbo].[xUntMst]
   1    1            |    |         |    |    |--Clustered Index Seek(OBJECT:([b2en].[dbo].[mItmMst].[PK_
   1    1            |    |         |    |--Clustered Index Seek(OBJECT:([b2en].[dbo].[xCstMst].[PK_xCstM
   1    1            |    |         |--Stream Aggregate(DEFINE:([Expr1010]=MAX([mBarItm].[bar_code])))
   0    1            |    |              |--Top(1)
   0    1            |    |              |--Index Scan(OBJECT:([dbo].[mBarItm].[IX_mBarItm_01]
   1    1            |    |--Clustered Index Seek(OBJECT:([b2en].[dbo].[xZipMst].[PK_xZipMst] AS [z]), SE
   1    1            |--Hash Match(Cache, HASH:([m].[bar_seq], [m].[bar_date]), RESIDUAL:([m].[bar_seq]=[
   1    1                 |--Compute Scalar(DEFINE:([Expr1017]=Convert([Expr1030])))
   1    1                      |--Stream Aggregate(DEFINE:([Expr1030]=Count(*)))
   1    1                           |--Clustered Index Seek(OBJECT:([b2en].[dbo].[mBarItm].[PK_mBarItm]),
```

4) MS-SQL 실행 계획 분석 사례 2 : OR 조건에 의한 비효율

**개선 전 SQL과 실행 계획**

```
Select ...

From    mBarHis b left join mShipMst s on b.wrk_date = s.ship_date
and     b.wrk_no = s.ship_no inner join mItmMst i on b.itm_code = i.itm_code
Where   b.wrk_kind = '1'
and     (b.bar_no   = @v1  or b.bar_code = @v1)
and     s.cst_code = '1'

'mItmMst' 테이블. 스캔 수  1, 논리적 읽기 수    3, 물리적 읽기 수 0, 미리 읽기 수 0.
'mBarHis' 테이블. 스캔 수 454, 논리적 읽기 수 2102, 물리적 읽기 수 0, 미리 읽기 수 0.
'mShipMst' 테이블. 스캔 수  1, 논리적 읽기 수    5, 물리적 읽기 수 0, 미리 읽기 수 0.

SQL Server 실행 시간:
   CPU 시간 = 43ms, 경과 시간 = 43ms.
Rows Exes StmtText
---------- ----------------------------------------------------------------------------
    1     1 Select b.bar_code,
    1     1  |--Nested Loops(Inner Join, OUTER REFERENCES:([b].[itm_code]))
    1     1       |--Nested Loops(Inner Join, OUTER REFERENCES:([s].[ship_no], [s].[ship_date]) WITH PREFET
  454     1       |    |--Index Seek(OBJECT:([b2en].[dbo].[mShipMst].[IX_mShipMst_01] AS [s]), SEEK:([s].[c
    1   454       |    |--Clustered Index Seek(OBJECT:([b2en].[dbo].[mBarHis].[PK_mBarHis] AS [b]), SEEK:([
    1     1       |--Clustered Index Seek(OBJECT:([b2en].[dbo].[mItmMst].[PK_mItmMst] AS [i]), SEEK:([i].[i
```

위 SQL의 트레이스 결과를 분석해 보면 IX_mShipMst_01 인덱스에서 454건을 추출하여 mBarHis 테이블과 Nested Loops 방식으로 454회 탐침을 시도하였으나 결과는 1건만 추출되었다는 사실을 알 수 있다.

탐침을 시도한 횟수가 454회인데 결과 건수가 1이라는 것은 테이블 액세스에서 453건이 제거된다는 의미이며, Inner 집합을 탐침할 때 처리 범위를 결정하는 데 참여하지 못하고 있는 아주 변별력이 좋은 조건이 있다는 의미이다.

실제로 mBarHis 테이블의 2개 칼럼에 똑똑한 변수 값이 각각 제공되었지만 or 조건에 의해 각 칼럼 에 생성된 인덱스를 사용하지 못했다. 문제의 or 조건 대신 union 연산자로 실행 계획을 분리하여 비 효율을 제거할 수 있다.

개선 후 SQL을 보면 union으로 분리된 각각의 SQL이 IX_mBarHis_01 인덱스와 IX_mBarHis_02 인덱스를 효율적으로 사용했음을 알 수 있다. union all 대신 union을 사용함으로써 Sort(DISTINCT …) 연산이 발생한 것이 불만이기는 하지만 이 사례에서는 중복 건이 존재하므로 불가피한 선택이었다.

**개선 후 SQL과 실행 계획**

```
Select ...

From   mBarHis b left join mShipMst s on rtrim(b.wrk_date) = s.ship_date
and    b.wrk_no = s.ship_no inner join mItmMst i on b.itm_code = i.itm_code
Where b.wrk_kind = '1'
and    b.bar_no  = @v1
and    s.cst_code = '1'
union
Select ...
From   mBarHis b left join mShipMst s on rtrim(b.wrk_date) = s.ship_date
and    b.wrk_no = s.ship_no inner join mItmMst i on b.itm_code = i.itm_code
Where b.wrk_kind = '1'
and    b.bar_code = @v1
and    s.cst_code = '1'

'mItmMst'   테이블. 스캔 수 1,   논리적 읽기 수     3,   물리적 읽기 수 0,   미리 읽기 수 0.
'mShipMst' 테이블. 스캔 수 2,   논리적 읽기 수     6,   물리적 읽기 수 0,   미리 읽기 수 0.
'mBarHis'   테이블. 스캔 수 2,   논리적 읽기 수    14,   물리적 읽기 수 0,   미리 읽기 수 0.

SQL Server 실행 시간:
   CPU 시간 = 0ms, 경과 시간 = 2ms.
Rows Exes StmtText
---------- ---------- ----------------------------------------------------------------------------
    1     1 Select b.bar_code, b.bar_no, b.itm_code, b.lot_no, i.wh_pos
    1     1   |--Sort(DISTINCT ORDER BY:([Union1006] ASC, [Union1007] ASC, [Union1008] ASC, [Union1009] ASC,
    1     1        |--Concatenation
    1     1             |--Nested Loops(Inner Join, OUTER REFERENCES:([b].[itm_code]))
    1     1             |    |--Nested Loops(Inner Join, OUTER REFERENCES:([b].[wrk_no], [b].[wrk_date]))
    2     1             |    |    |--Bookmark Lookup(BOOKMARK:([Bmk1000]), OBJECT:([b2en].[dbo].[mBarHis] AS
    2     1             |    |    |    |--Index Seek(OBJECT:([b2en].[dbo].[mBarHis].[IX_mBarHis_01] AS [b]),
    1     2             |    |    |--Clustered Index Seek(OBJECT:([b2en].[dbo].[mShipMst].[PK_mShipMst] AS [
    1     1             |    |--Clustered Index Seek(OBJECT:([b2en].[dbo].[mItmMst].[PK_mItmMst] AS [i]), SE
    0     1             |--Nested Loops(Inner Join, OUTER REFERENCES:([b].[itm_code]))
    0     1                  |--Nested Loops(Inner Join, OUTER REFERENCES:([b].[wrk_no], [b].[wrk_date]))
    0     1                  |    |--Bookmark Lookup(BOOKMARK:([Bmk1008]), OBJECT:([b2en].[dbo].[mBarHis] AS
    0     1                  |    |    |--Index Seek(OBJECT:([b2en].[dbo].[mBarHis].[IX_mBarHis_02] AS [b]),
    0     0                  |    |--Clustered Index Seek(OBJECT:([b2en].[dbo].[mShipMst].[PK_mShipMst] AS [
    0     0                  |--Clustered Index Seek(OBJECT:([b2en].[dbo].[mItmMst].[PK_mItmMst] AS [i]), SE
```

## 성능 개선 도구

DBMS에서는 성능 개선이나 모니터링을 하기 위한 기능을 제공하고 있다. 각 기능은 차이가 있 다. 기본적으로 서버 상태, 트레이스(Trace), 딕셔너리(Dictionary) 정보, 실행 계획을 제공하고 있 다. 여기에 소개하는 오라클, UDB, SQL 서버 외의 DBMS에서도 유사한 기능을 제공한다.

### Oracle

#### StatsPack/AWR

8i부터 사용하던 Statspack, 10g 이후 사용하게 된 AWR(Automatic Workload Repository)은 오라클이 제공하는 표준 성능 관리 도구다(그 이전에는 utlbstat/utlestat 스크립트를 이용해 비 슷한 리포트를 뽑아볼 수 있었다). 이들 도구가 제공하는 기능을 간단히 요약하면 오라클이 내부적으로 누적 관리하는 다양한 동적 성능 뷰를 주기적으로 특정 리파지토리에 별도 저장했다가 사용자가 원하는 시점에 특정 기간 동 안의 성능 분석 리포트를 출력해 봄으로써 데이터베이스 전반의 건강 상태를 체크할 수 있게 해 주 는 것이다.
부하 프로파일, 인스턴스 효율성, 공유 풀 통계, Top 5 대기 이벤트 발생 현황 등을 일목요연하게 보여줄 뿐만 아니라 분석 항목별 상세 분석 자료를 제공함으로써 데이터베이스 성능 병목 현상이 발생하는 주원인을 쉽게 찾을 수 있게 해 준다. Statspack과 AWR은 거의 비슷한 내용을 담고 있 으며, 다른 점이 있다면 정보를 수집하는 방식에 있다. Statspack은 SQL을 이용해 데이터 사전 을 조회하는 방식인데 반해, AWR은 SGA를 DMA(Direct Memory Access) 방식으로 직접 액세 스하므로 좀 더 빠르게 정보를 수집한다. 상대적으로 부하가 적으므로 이전보다 더 많은 정보를 수 집하고 제공할 수 있게 된 것이다.

#### SQL 트레이스

SQL 트레이스는 데이터베이스의 인스턴스 또는 세션 단계에서 수행되는 모든 수행 SQL의 통계 치 및 대기 이벤트에 대한 정보를 수집해 주는 기능을 제공한다. 주요 수집 정보로는 파스(Parse), 실행(Execute), 패치(Fetch) 시의 CPU 사용 시간, 수행(Elapsed) 시간(대기 시간 포함), 메모리 블록 I/O 횟수 및 디스크 블록 I/O 횟수 등이 있다. 또 선택적으로 대기 이벤트에 대한 정보도 같 이 수집하게 할 수 있다. TKPROF를 이용하면 수집된 정보를 가공하여 분석하기 용이한 레포팅 기능을 제공한다. SQL 트레이스를 통해 응답 시간 또는 처리 시간을 줄일 수 있는 대상 SQL을 식 별할 수 있고, 수집 기간 동안 수행된 모든 SQL에 대한 정보를 담고 있기 때문에 인덱스 설계 시 에 직접적인 자료로 활용된다.

IBM UDB

■ 스냅샷 모니터

DB2의 운영, 성능, 애플리케이션에 대한 정보를 수집할 때 사용할 수 있는 모니터링 방법이며, 스 냅샷이나 이벤트 모니터를 사용하고 특정 시점에서의 데이터베이스 활동에 관한 정보를 제공하며, DB2 활동 상태에 대한 정보이다. 스냅샷을 취했을 때 사용자에게 반환되는 정보 양은 사용하는 모니터 스위치에 의해 결정된다. 이 스위치들은 인스턴스 또는 인용 프로그램 레벨에서 설정할 수 있다. 이 데이터는 DB2가 수행될 때 유지되고, 성능과 문제 해결을 위한 중요한 정보로 제공된다.

■ 이벤트 모니터

이벤트 모니터링은 DB2 이벤트의 특정 사건 발생을 기록한다. 교착 상태, 연결, SQL 문장들을 포 함한 일시적인 이벤트 정보를 수집할 수 있다. 스냅샷 모니터링이 스냅샷을 취할 때 데이터베이스 활동의 상태를 기록하는 반면, 이벤트 모니터는 이벤트나 전이가 발생할 때 데이터베이스 활동을 기록한다. 이벤트 모니터는 SQL DDL을 사용하여 생성된다.

■ SQL 모니터링

SQL이 어떻게 수행되는지를 알기 위해서는 액세스 플랜을 분석한다. 실행 계획 기능은 SQL 문장 을 해석하기 위해 어떻게 DB2가 데이터를 액세스하는지에 대한 정보를 제공한다. 다.

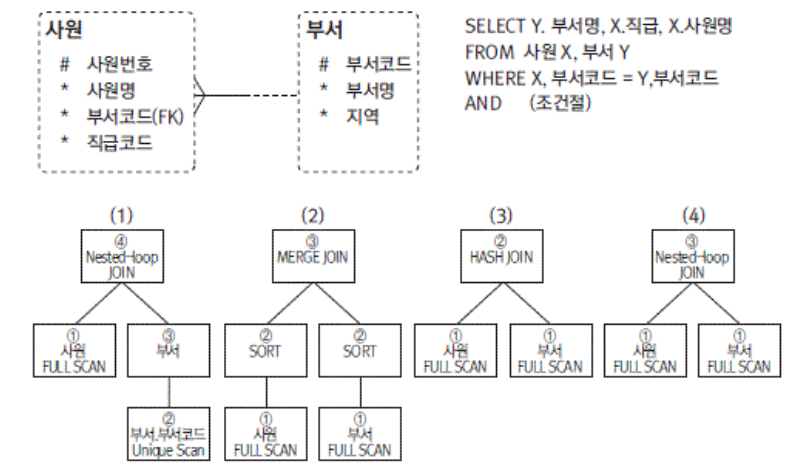
■ NT 성능 모니터링

운영체제의 특정 구성 요소에서 사용하는 리소스 및 프로그램에서 사용하는 리소스에 대한 자세한 데이터를 제공하며, 성능 모니터링을 통해 CPU, 메모리, 디스크 등에 대한 다양한 정보를 얻을 수 있다. 상태를 실시간으로 모니터링할 수도 있으며 그 내역을 파일로 저장하여 엑셀 등에서 다양한 방법으로 분석할 수도 있다. 물론 원하는 시간 동안 자료를 수집하도록 스케줄링도 가능하다.

■ SQL 프로파일러

SQL 서버 인스턴스의 이벤트에 대한 데이터를 캡처하고 파일 또는 테이블에 저장하여 분석할 수 있는 기능을 제공한다. 수집된 트레이스를 분석하여 주요 액세스 경로를 파악할 수 있으며 인덱스 설계 시에 기초 자료가 된다. SQL 프로파일러를 사용해 SQL 서버 인스턴스의 성능을 모니터링하 고, 트랜잭션 SQL문과 저장 프로시저를 디버그하며, 실행 속도가 느린 쿼리를 확인할 수 있으며, 운영 시스템에서 이벤트를 캡처하고 테스트 시스템에서 그 이벤트를 재생하여 SQL 서버의 문제 를 해결할 수 있다. 또한 SQL 서버 인스턴스에서 발생하는 동작을 감시하고 검토할 수 있다.

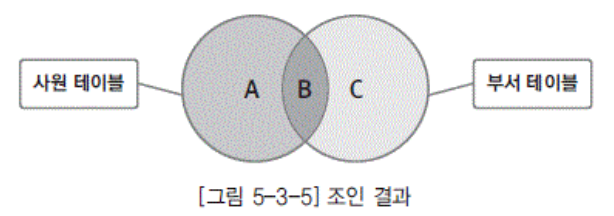
조인은 카티션 프로덕트를 수행 후 셀렉션과 프로젝션을 수행하는 데이터베이스의 연산으로, 관계형 데이터베이스에서 공통적으로 사용하고 있는 조인 기법(Join Technique)에는 Nested loops 조인, Sort merge 조인, Hash 조인 등이 있다. 이 이외에도 일부 데이터베이스에서는 Hybrid 조인, Star 조인과 Semi 조인 등을 지원한다.



[그림 5-3-4] 조인 설명

조인은 위에서 언급한 조인을 수행하는 내부적인 매카니즘으로 구분을 할 수도 있지만, 조인을 기술한 조인 조건의 연산자를 기준으로 조인의 종류를 나누기도 한다. 예를 들어, 조인 조건이 (=)로 정의되었다 면 equi 조인이라 하며, >=, <=, between 등이 조인 조건에 기술되었다면 between 조인이라 한다.

마지막으로 조인을 분류하는 방법은 조인의 결과가 어떤 집합에 의하여 결정되느냐에 따라 분류하는 방법으로, 조인의 결과가 Outer 집합에 의하여 결정되는 Outer 조인과 Inner 집합에 의하여 결정되 는 Inner 조인으로 구분할 수 있다. 아래 조인의 결과를 나타내기 위한 밴 다이어그램을 보면 Inner 조인의 결과는 B 집합이 되고, 사원 집합을 Outer 집합으로 하여 Outer 조인을 수행한 결과는 A + B 집합이 된다. 그런데 Inner 조인의 결과인 집합 B는 사원과 부서의 교집합으로, Inner 집합인 부서를 액세스하여야만 결정될 수 있다. 그리고 Outer 조인의 결과인 A + B 집합은 Outer 집합인 사원과 동 일한 집합이므로 Outer 조인의 결과는 Outer 집합을 액세스할 때 이미 결정된다.



[그림 5-3-5] 조인 결과

위에서 조인을 분류한 내용들을 이해하려면 Nested Loop 조인의 알고리즘을 이해하여야 한다. 이 알고리즘을 보면 두 개의 Loop가 중첩되어 조인이 수행되고 있는 것을 알 수 있다. 여기에서 EMP 테 이블을 Outer 집합이라 하고, DEPT 테이블을 Inner 집합이라 한다. 그 이유는, EMP 테이블은 바깥 쪽 루프에서 액세스되고 DEPT 테이블은 안쪽 루프에서 액세스되는 테이블이기 때문이다. 즉, Outer 집합이란 Nested Loop 조인에서 바깥쪽 루프에서 액세스되는 집합을 의미하고, Inner 집합은 안쪽 루프에서 액세스되는 집합을 의미한다. 그런데 Outer 집합이라는 용어가 Nested Loop 조인에서

만 사용되는 것은 아니다. 조인 연산이 가지는 특성상 어떤 조인 기법을 사용하더라도 항상 Outer 집합과 Inner 집합이 나타나게 된다. 물론 다른 조인 기법에서는 Nested-Loop 조인처럼 Outer 집합과 Inner 집합의 구분이 명확하지 않아 이 용어의 사용이 적절하지 않다고 주장하는 사람들도 있지만 이 러한 용어가 나타난 배경을 이해하고 사용한다면 굳이 사용하지 못할 이유가 없다고 생각한다.

Outer 집합과 Inner 집합은 조인을 수행하는 각 단계에 그 집합이 참여하는 위치로 구분을 한 것이 므로 조인 각 단계마다 정의될 수 있다. 그러나 드라이빙 집합은 조인 전 과정에서 하나의 집합만이 가 질 수 있는 특징으로 조인에 참여하는 집합 중 최초로 액세스되는 집합을 의미한다. 예를 들어, 아래 알 고리즘에서 사원 테이블은 Loop의 바깥쪽에 있고 최초로 액세스되므로 Outer 집합이면서 드라이빙 집합도 된다.

```
while( (ch=read(EMPr))!=EOF)    /*OuterLoop*/
{
  ......
  while( (ch=read(DEPTs))!=EOF) /*InnerLoop*/
  {
    If ( EMPr==DEPTs)
    Add EMPr ? DEPTstotheresult:
  }
}
```

[그림 5-3-6] Nested-Loop 조인 알고리즘

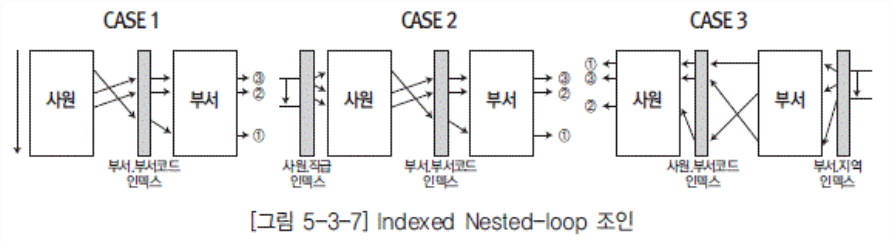
## Nested-Loop 조인

조인 연산은 두 집합을 카티션 프로덕트 형태로 모든 튜플을 열거한 다음, 조인에 만족하지 않는 튜플 을 제거하는 두 가지 기본 알고리즘으로 구성되어 있다. 특히 Simple Nested-Loop 조인은 이러한 기 본 개념만 가지고 구성되었기 때문에 인덱스를 필요로 하지 않을 뿐만 아니라 어떠한 조인 조건에서도 사용할 수 있는 조인 알고리즘이다. 그러나 조인을 수행하기 위해 두 집합의 모든 튜플을 카티션 프로덕 트 형태로 검사하므로 비용이 매우 많이 든다. 예를 들어, 사원과 부서 테이블이 Nested-Loop 조인을 수행한다고 가정해 보자. 사원 테이블은 10,000개의 로우를 가지고 있고 부서 테이블은 200개의 로우 를 가지고 있다면 이 조인에서 액세스되어야 하는 튜플의 개수는 10,000 * 200 이다. 즉, 모든 사원이 모든 부서를 스캔해야 한다. 이처럼 처리해야 하는 대상 집합이 커지면 탐침해야 하는 일량도 급속하게 증가해 수행 속도가 저하된다. 그리고 Outer 집합에 추출된 로우의 수만큼 Inner 집합을 반복적으로 액세스하므로 Inner 집합을 메모리로 읽어 들인 횟수가 증가할수록 수행 속도는 더욱더 저하된다.

위의 두 가지 성능 저하 조건이 모두 충족된다면 최악의 시나리오가 나올 가능성이 높기 때문에 옵티 마이저는 Nested-Loop 조인으로 수행하는 것을 포기하게 될 것이다.

그러나 Inner 집합에 조인 조건으로 기술된 속성에 인덱스가 존재한다면 옵티마이저는 테이블 스캔을 하지 않고 인덱스를 통해 조인 조건을 만족시키는 튜플을 얻어낼 수 있다. 이러한 조인 기법을 Indexed Nested-Loop 조인이라 한다. 이 조인 기법에서는 이미 구축되어 있는 인덱스를 사용할 수 도 있고 조인을 수행하기 위해 임시로 인덱스를 만들어 사용할 수도 있다.

다음은 Indexed Nested-Loop 조인에 대한 몇 가지 사례를 도식화한 것이다.




[그림 5-3-7] Indexed Nested-loop 조인

CASE1의 경우라면 다음과 같은 절차로 수행을 하게 될 것이다.

① CASE1의 경우 사원 테이블을 드라이빙 테이블로 선정하였다. 그런데 사원 테이블은 주어진 조건 중에서 사용 가능한 인덱스가 없으므로 테이블 스캔으로 전체 데이터를 검색한다.
② 사원 테이블에서 추출되는 로우의 수만큼 반복해 부서.부서코드 인덱스를 탐침한다. 부서.부서코 드는 부서 테이블의 PK이므로 탐침 시에는 인덱스 Unique scan을 한다.
③ 부서 테이블의 ROWID를 이용해 부서 테이블을 액세스한다.
④ 부서 테이블에서 SELECT-LIST에 포함된 속성을 추출하여 결과 집합을 완성한다.
⑤ 완성된 결과 집합을 일정한 크기로 사용자에게 반환한다.

■ 조인 조건

Indexed nested-loop 조인으로 수행되려면 Inner 테이블에 조인 조건으로 사용된 칼럼에 반드 시 인덱스가 존재해야만 한다. Inner 테이블에 인덱스가 존재하지 않는다면 Outer 테이블에서 추 출된 로우의 수만큼 Inner 테이블을 반복해서 테이블 스캔하게 된다. 이러한 비효율을 없애기 위 해 옵티마이저는 드라이빙 조건의 범위가 넓거나 Inner 테이블의 사이즈가 크면 Sort-merge 조 인이나 Hash 조인으로 실행 계획을 유도하려oop 조인에서 결과 집합이 출력되는 순서는 드라이빙 테이블을 액세스한 순서와 동일하 다. 즉, 드라이빙 테이블을 Full Scan했다면 드라이빙 테이블의 데이터 저장 순서로 출력되고, 드 라이빙 테이블 액세스 시에 인덱스를 이용했다면 이용한 인덱스의 순서로 결과 집합이 출력된다.

- CASE 1인 경우 사원 테이블에 데이터가 저장된 순서
- CASE 2인 경우는 사원.직급인덱스 순서
- CASE 3인 경우는 부서.지역 순서

■ 조인 순서

옵티마이저는 조인 시에 먼저 드라이빙 테이블을 결정하고 나머지 집합의 조인 순서를 결정한다. 드라이빙 테이블은 드라이빙의 범위가 가장 작은 집합, 즉 가장 작은 작업량을 가지고 드라이빙할 수 있는 집합을 선정하고 조인의 순서를 정할 때에는 조인의 효율이 좋은 집합을 먼저 조인에 참여시키려고 노력한다. 여기에서 조인의 효율이 좋다는 의미는 조인의 결과로 출력하는 로우의 수가 작다는 것을 의미한다. 이것은 Nested Loop 조인의 경우 선행 집합의 결과가 다음 집합을 액세스 하는 작업량을 결정하기 때문이다.
CASE 3과 같이 사원.부서 코드에 인덱스가 있다면 옵티마이저는 사원을 드라이빙 테이블로 선정 할 수도 있다. 그리고 SQL문의 조건절에 부서.지역=.제주'AND 사원.직급 LIKE 'A%'가 추가 된 경우는 CASE 2나 CASE 3로 실행 계획이 작성될 수 있다.

- CASE 2 인 경우 : 사원.직급이.A'로 시작되는 사원을 검색하고 검색된 결과 집합으로 부서 테이블을 탐침한다. 직급이.A'로 시작되는 사원 100명이 검색되었다면 100명을 대상으로 부서를 탐침하게 되고, 이들 중 부서.지역이.제주'인 대상을 최종 결과 집합으로 출력하게 된다.
- CASE 3 인 경우 : 부서.지역이.제주'인 부서를 검색하고 검색된 결과 집합으로 사원 테이블을 탐침한다. 만약, 2개의 부서가 검색되었다면 2개의 부서를 대상으로 모든 사원을 탐침하면서 사원.직급이.A'로 시작되는 사원을 최종 결과 집합으로 출력하게 된다. 위에서 언급한 것처럼 Nested-Loop 조인은 드라이빙 테이블의 선정과 조인 순서가 작업과 성능에 많은 영향을 미친다. 그리고 응답 시간에 대한 성능은 다음과 같은 규칙을 가지고 있다.
- 드라이빙 조건의 범위가 좁은 경우는 항상 성능이 양호하다.
- 드라이빙 조건의 범위가 넓은 경우는 체크 조건도 검색 범위가 넓어야 성능이 양호하다.

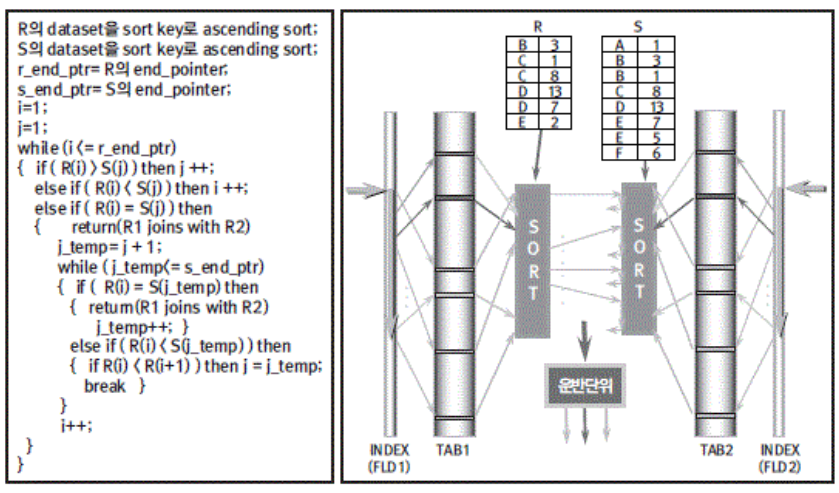
## Sort-Merge 조인

Sort-merge 조인은 조인하려는 두 집합을 조인 속성으로 정렬하여 sorted lists를 만든 후 이들 을 merge하는 조인 기법으로, 인덱스가 없을 때 Simple Nested-Loop 조인으로 수행하는 비효율 을 개선하기 위한 방안으로 연구되었기 때문에 Inner 집합에 인덱스가 존재하지 않을 경우 수행하는 Simple Nested-Loop 조인보다 훨씬 더 좋은 성능을 나타낸다. Sort-merge 조인은 Simple Nested-Loop 조인의 가장 큰 비효율인 카티션 프로덕트를 해결하 기 위해 모든 튜플들이 조인 속성에 대해 같은 값을 갖도록 파티션함으로써 조인해야 하는 대상 튜플 들의 그룹들을 쉽게 검색이 가능하도록 구성하였다.

Sort-merge 조인이 동등 조인 조건에 대해서만 수행이 가능한 것은 이러한 파티션 기반의 방식 을 지원하고 있기 때문이다.

[그림 5-3-8]을 보면 Sort-Merge 조인은 다음과 같은 절차로 수행되고 있는 것을 알 수 있다.

① 정렬 단계 : 조인에 참여하는 두 집합을 조인 조건에서 사용되는 속성을 기준으로 정렬한다.
② 머지 단계 : 정렬된 두 집합 중에서 하나의 집합을 차례로 검색하면서 다른 나머지 하나의 집합 에서 검색 조건에 맞는 튜플을 찾아 결과 집합에 포함시킨다. 머지 단계의 기본 알고리즘은 Nested-Loop 조인의 기본 알고리즘과 매우 흡사하다. 차이가 있다면 두 집합이 조인 속성으 로 정렬되어 있고 머지는 동등 조인 조건에서만 가능하므로 동등 조건이 아니면 머지를 시도하 지 않는다는 것이다. 동등 조건을 찾는 알고리즘이 인덱스의 머지 알고리즘과 비슷하여 머지 단 계가 인덱스 머지의 알고리즘과 동일하다고 이야기를 하는 경우도 있지만 그것은 절대 그렇지 않다.
③ 머지를 수행해 가면서 처음 동등 조건이 나타나면 Inner 집합을 하나씩 증가하면서 동등 조건인 경우 계속해서 머지를 수행한다. Inner 집합이 더 큰 값을 가지게 되면 검색을 멈추고 Outer 집합으로 회귀하여 Outer 집합에서 포인트를 증가시키고 다음 데이터를 읽는다.
④ ②, ③ 단계를 반복 수행한다.




[그림 5-3-8] Sort-merge 조인

■ 조인 조건

Sort-Merge 조인은 조인 속성인 사원.부서코드와 부서.부서코드 칼럼에 인덱스가 없는 경우에 주로 발생한다. 조인은 위에서 언급한 것처럼 정렬 단계와 머지 단계로 나누어 수행된다.

■ 출력 및 연결 순서

정렬된 결과를 순차적으로 비교하여 머지를 수행하므로 Outer 집합의 정렬 순서와 Inner 집합의 정렬 순서가 머지되어 출력된다. 그리고 조인에 참여하는 집합이 3개 이상이라면 조인의 순서가 조인 성능에 영향을 미칠 수 있다. Sort-Merge 조인도 조인의 순서를 결정하는 원리는 다른 조인 기법처럼 조인 효율이 좋은 집합을 조인에 먼저 참여시킨다. 즉, 선행 집합의 조인 결과가 다음 조 인에 참여하는 집합의 작업량에 영향을 미치므로 조인 효율이 좋은 집합을 조인에 먼저 참여시켜 야 전체적인 조인 성능이 좋아지게 된다.

■ 조건절

조건절이 부서.지역='제주' AND 사원.직급 LIKE 'A%'가 추가된 경우는 사원.직급 LIKE 'A%'를 만족하는 100건을 정렬하고, 부서.지역이 제주인 2개 부서를 정렬하여 만족하는 5명을 출력한다. Sort-merge 조인은 소트에 참여하는 로우의 수에 의해 수행 속도가 결정된다. 정렬할 로우의 수 가 많은 경우는 정렬하는 데 수행되는 시간이 길어지므로 OLTP 환경에서 사용할 수 없다. 배치 작 업인 경우에도 정렬 작업이 메모리 내 정렬 영역(Sort area)을 초과할 경우는 스와핑이 발생하고 수행 속도가 더욱 느려진다. 필요시 정렬 영역을 추가 할당하는 것을 고려해야 한다.

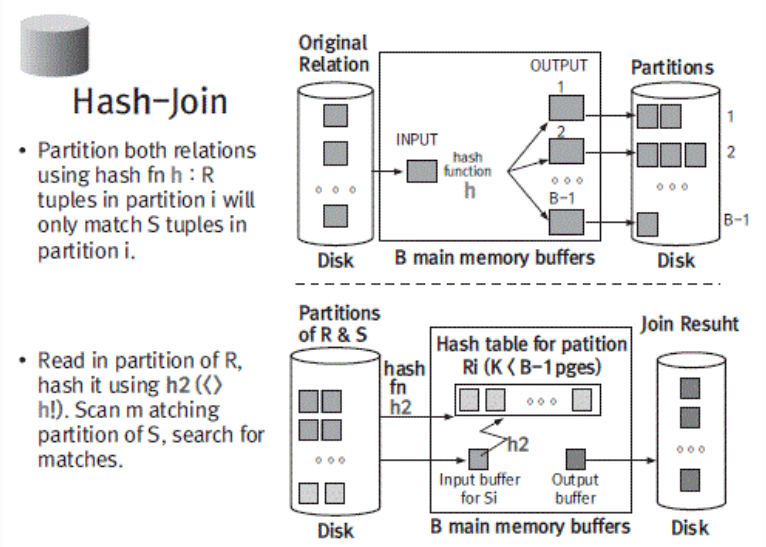
■ 이용

- 독립적으로 처리 범위를 줄인 후 조인에 참여하므로 테이블 각각의 조건에 의해 대상 집합을 줄일 수 있을 때 유리하다.
- 처리 대상이 전체 테이블일 때 랜덤 IO 부하가 큰 Nested-Loop 조인보다 유리하다.
- 조인 속성에 인덱스가 없을 때 Simple Nested-Loop 조인보다 성능이 우수하다.
- 효과적인 수행을 위해서는 적정한 SORT AREA 사이즈가 확보되어야 한다.
- 정렬에 대한 부하가 많이 발생하므로 대용량 처리 시 수행 속도가 저하될 수 있다.

## ➕ Hash 조인

Hash 조인은 관계형 데이터베이스에서 비용이 가장 많이 들어가는 조인 방법이지만 대용량의 데 이터 조인 시에 Sort-merge 조인이나 Nested-Loop 조인보다 더 좋은 성능을 나타낸다. 이러한 이 유는 Sort-merge 조인의 경우 데이터량이 증가할수록 양쪽 집합을 정렬해야 하는 비용 부담이 커지 고, Nested-Loop 조인은 Inner 집합을 반복 탐침해야 하는 비효율이 더욱더 증가하기 때문이다. Hash 조인의 기본 알고리즘은"조인에 참여한 두 집합 중에서 작은 집합의 해쉬 테이블을 메모리 상에 만들고 큰 집합은 조인을 위해 해쉬 테이블을 탐침한다"는 것이다. Hash 조인은 이용 가능한 Hash area가 작은 집합을 유지할 만큼 충분히 클 경우에 만족할 정도로 좋은 성능을 나타내지만 이 용 가능한 메모리가 작은 집합을 유지할 정도로 충분하지 않다면 Hash area는 오버플로우가 발생한다. Hash 조인은 이 문제를 다루기 위해 두 집합을 좀 더 작은 단위로 나누게 되는데, 이를 파티션 이라 한다.

Hash 조인은 다음과 같은 여러 단계로 나누어 조인을 수행한다.




[그림 5-3-9] Hash 조인

Step 1 - 파티션 수

파티션의 개수가 많으면 I/O 효율을 떨어뜨리고 적으면 메모리 적중률을 떨어뜨리기 때문에 이를 적절하게 가져가는 것이 가장 중요하다.

Step2 - 해쉬 함수

조인에 참여한 두 집합 중에서 작은 집합을 드라이빙 테이블로 선택하여 메모리로 읽어 들인다. 이 때 해쉬되는 로우가 파티션에 골고루 분산되게 하기 위해 해쉬 함수1을 사용하여 해당 파티션에 매핑하고, 해쉬 함수2를 사용하여 생성된 해쉬 값은 다음 단계를 위하여 조인키와 함께 저장한다.

Step3 - 비트맵 벡트

비트맵은 2차원 버켓으로 이루어져 있으며 해쉬 함수 1,2를 통과하여 만든다. 즉, 파티션이 100개 라면 100*100

의 10000개의 셀로 이루어지게 된다.

Step 4 – 디스크에 파티션 쓰기

Hash area가 모두 차면 가장 큰 파티션이 디스크로 내려간다. 디스크의 파티션은 파티션에 할당 되는 로우에 의해 디스크에서 갱신된다. Hash area의 부족으로 하나의 파티션만이 메모리에 올 라간다면 나머지 파티션은 모두 디스크에 놓여진다.

Step5 – 메모리에 적재 가능한 최대 파티션

Step5.1 – 최대 파티션 수를 정하고 크기순으로 파티션을 정렬한다.

Step5.2 – 파티션 번호가 가장 작은 것부터 선택하여 메모리에 로드한다.

Step5.3 – 나머지는 디스크에 저장한다.

Step6 – 작은 집합의 해쉬 테이블 생성

이미 생성된 해쉬 값을 사용하여 해쉬 테이블을 생성한다.

Step7 – 비트 벡트 필터링

조인 수행 시 비트맵과 대조하여 1이면 조인을 해야 하는 로우로 인식하고 아니면 조인이 필요 없 는 로우이므로 버린다. 이러한 과정을 비트 벡트 필터링이라고 하며, 조인되는 칼럼의 카디날리티 가 아주 적고 probe input 대부분이 일치하지 않을 때 아주 효과적이다.

Step8 – 처리되지 않은 파티션 쌍을 디스크로부터 읽어오기

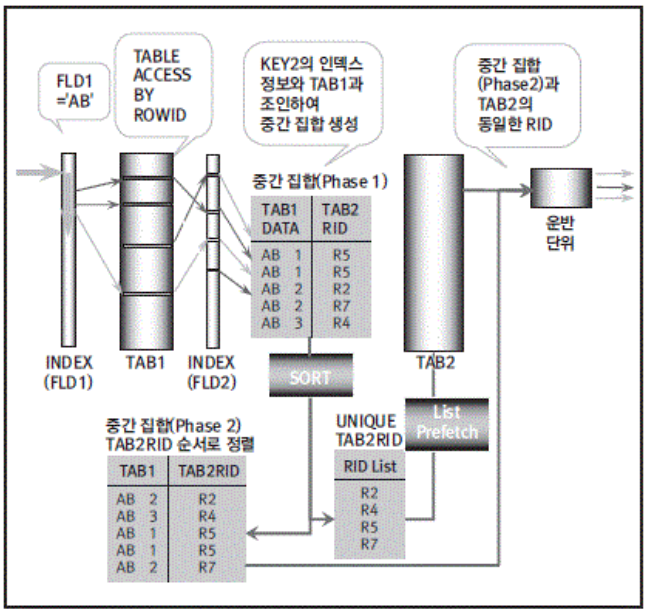
큰 집합의 파티션이 끝나면 작은 집합의 수행되지 않은 파티션들이 최대한 메모리로 올려지고 해 쉬 테이블이 생성된다. 그리고 그에 대응되는 큰 집합의 파티션이 다시 읽혀져 메모리 조인이 실행 된다. 최초 파티션 조인에서 옵티마이저는 작은 집합을 build input으로 사용하고 큰 집합은 probe input으로 사용하지만 그 이후에는 파티션 쌍에서 작은 파티션을 build input으로 사용하 고 큰 파티션은 probe input으로 사용하는 동적 역할 전환(dynamic role reversal)이 일어난다.

### Hybrid 조인

하이브리드 조인은 이름이 의미하는 것처럼 Nested-Loop 조인과 Sort-merge 조인의 알고리즘 을 혼합한 개념이다.

Indexed Nested-Loop 조인에서 Inner 테이블의 조인 속성이 non-clustered index로 되어있 을 경우 각 데이터 엔트리는 서로 다른 페이지를 가리키므로 Outer 테이블에서 반환되는 로우의 수만큼 Inner 테이블의 페이지를 탐침하여야 한다.

하이브리드 조인은 이러한 비효율을 개선하기 위해 Inner 테이블의 인덱스를 탐침할 때마다 테이 블을 검색하는 것이 아니라 Inner 테이블의 인덱스를 탐침한 후 그 결과를 중간 집합으로 생성하여 중복이 없는 Rowid 리스트를 만들고, 그 결과를 가지고 Inner 테이블의 페이지를 탐침하는 조인 방 식이다.




[그림 5-3-10] Hybrid 조인

성능 개선 방법론 | 조인(Join) | 애플리케이션 성능 개선 | 서버 성능 개선

## 온라인 프로그램 성능 개선

온라인 프로그램의 성능 개선 목표는 응답 시간 단축인 경우가 대부분이다. 애플리케이션 사용자 가 업무를 수행하기 위해 최소한 응답 시간을 보장하고 더 나아가 효과적인 진행이 가능한 수준을 유 지해야 한다.

온라인 프로그램의 특징은 다음과 같다.

- 화면 조회가 가능할 정도로 1회 조회 데이터가 소량이다.
- 신속한 트랜잭션 처리가 요구된다.
- 조회 조건이 단순하다.
- 업무 형태에 따라 데이터 액세스 패턴이 고정되어 있다.

온라인 프로그램 성능 개선 작업은 다음 사항을 고려한다.

- 사용 빈도가 높은 SQL 문을 개선하는 것이 효과적이다.
- 인덱스를 이용하여 데이터 액세스 범위를 줄이는 것이 효과적이다.
- 부분 범위 처리로 응답 시간을 단축한다.
- 부분 범위 처리를 하기 위해서 Nested-Loop 조인과 인덱스를 이용한 정렬을 유도한다.
- 장기 트랜잭션 처리를 억제한다.

## 온라인 프로그램의 성능 개선 사례

### 상수 바인딩에 의해 발생되는 파싱 부하

■ 문제점

최근 대중적인 개발 형태인 웹 기반의 애플리케이션은 SQL문 작성 시 조건절에 변수 바인딩을 실 행하는 정적인 형태의 작성 기법이 아니라 SQL문을 저장한 문자열에 사용자로부터 입력 받은 상 수 값을 직접 결합시켜 동적으로 실행시키는 상수 바인딩 형태의 작성 기법을 주로 사용한다. 이런 방법으로 개발된 시스템은 사용자가 작은 경우에는 문제가 되지 않으나 동시 접속자 수가 증가하 면 심각한 성능 저하 현상이 발생한다.

■ 원인

이런 형태의 SQL 작성 기법은 개발이 용이하지만 데이터베이스에 과도한 파싱 부하 및 옵티마이 저 최적화 작업을 어렵게 만든다. 대부분의 데이터베이스들은 옵티마이저에 의해 실행 계획을 미 리 작성하여 정적으로 보관하고 있거나 동적으로 작성하여 작성된 정보를 담고 있는 파싱 정보를 데이터베이스의 공유 메모리 영역에 적재한다. 동일한 SQL문이 실행될 때 메모리에 적재된 실행 계획을 재사용한다. 상수가 바인딩되어 실행되는 SQL문은 상수 값의 변화에 따라 SQL문이 동적 으로 바뀌어 데이터베이스 메모리 영역에 보관되어 있는 실행 계획 정보를 재사용할 수 없거나 재 사용 확률이 상당히 낮아진다. 상수가 바인딩된 부분을 제외하고는 동일한 SQL문임에도 하드 파 싱을 해야 하는 일이 발생한다. 파싱에 소요되는 비용은 대부분 CPU 사용 시간이므로 SQL문 실 행 시 변수가 바인딩되어 파싱 결과를 재사용할 수 있도록 구현하는 것이 데이터베이스 성능을 안 정화시키는 데 유리하다.

### 웹 게시글 형태의 인터페이스 시 부분 범위 처리

인터넷이 대중화되면서 웹 서비스 아키텍처 구조가 보편화되었다. 웹아키텍처는 N-티어 구조로 실행 프로그램이 데이터베이스와 지속적인 세션을 가질 수 없다. 미들웨어를 사용하는 환경에서도 동일하다. 이런 환경에서는 클라이언트/서버의 2계층 구조에서 GUI 툴을 이용하여 쉽게 구현하던 부분 범위 처리에 의한 스크롤 처리가 불가능하다.

이러한 현실적인 어려움으로 인해 대부분의 게시판 인터페이스에서는 사용자 액션이 들어올 때마 다 해당 테이블 전체를 다시 읽거나 조건절에 만족하는 대상을 읽어 필요한 부분을 잘라 질의 결과를 반환하는 형태를 취한다. 이 방식은 대상 건수가 증가하면 할수록 응답 속도가 떨어질 수밖에 없으 며, 조회에서 조인을 하는 대상 테이블들의 조건이 어느 한

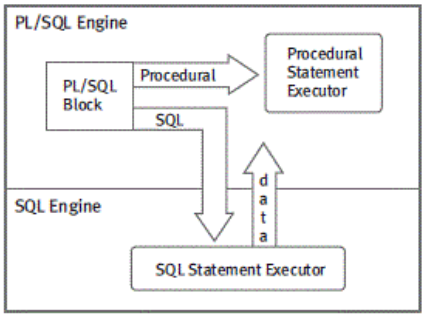
테이블에만 특정되는 것이 아니라 각각의 테이블에 조건이 분산되어있는 경우나 정렬 또는 집계가 필요한 경우 전체 범위 처리를 하지 않고는 실질적으로 필요한 부분만을 읽어 응답 속도를 줄여 줄 수 있는 부분 범위 처리가 어렵다. 이러한 문 제를 해결하기 위해 IBM의 DB2나 오라클 9i R2의 경우 Scrollable 커서를 지원하고 있지만 2-티 어 환경에서만 적용 가능하여, 3-티어와 같은 비연결 지향적인 서비스 형태를 가지는 구조에서는 이 기능을 사용할 수 없다.

액세스에 필요한 I/O를 최소화하기 위한 전략적인 테이블 설계와 인덱스의 설계가 성능 향상을 위 해 가장 중요하다. Page-Up/Page-Down 스크롤에 사용되는 정렬 기준 값으로 인덱스를 생성하고 스크롤 처리 시 인덱스를 이용한다. 인덱스는 칼럼 값으로 정렬되어 있으므로 현재 위치에서 다음 위 치로 옮겨갈 때 스크롤 조건을 SQL문에 제공하여 구현하면 불필요한 IO를 줄일 수 있다.

### 과다한 함수 사용으로 인한 부하 발생

저장 함수는 절차적 처리가 불가능한 SQL의 단점을 보완하기 위해 사용되는 언어이다. 목적에 맞게 효과적으로 이용하면 높은 생산성과 관리의 편리성을 제공하지만 잘못 사용하면 성능에 악영향을 미친 다. 대표적인 예가 코드명, 고객명, 상품명 등과 같이 이름 찾기 저장 함수 사용이다. 이름 찾기용 저장 함수는 프로그램 구조가 단순해져 편리성 측면에서 사용될 수는 있겠지만 DBMS 측면에서 보면 아주 비효율적인 개발 방법이다.

저장 함수는 where 조건을 만족하는 로우의 수만큼 실행되기 때문에 처리 범위가 넓은 경우 많은 비 효율이 발생하게 된다. 1,000개의 결과를 반환하는 SQL에 하나의 저장 함수를 사용하면 SQL을 한 번 수행할 때마다 저장 함수를 1,000번 호출한다.




[그림 5-3-11] 저장 SQL 구조

그리고 SQL 엔진과 절차적인 처리를 위한 PL/SQL 엔진이 분리되어 있으므로 SQL 엔진에서 처리 한 결과를 절차적으로 처리하려면 결과 값을 PL/SQL엔진으로 전송해 주어야 한다. 이러한 과정을 문 맥 전환(Context Switching)이라 하며, PL/SQL의 성능이 저하되는 주요한 원인이다. 이러한 문맥 전환을 없애기 위해서는 절차적인 부분 대신 조인 연산을 통해 처리해야 한다.

저장 함수는 다음과 같은 경우에 효과적으로 이용될 수 있다.

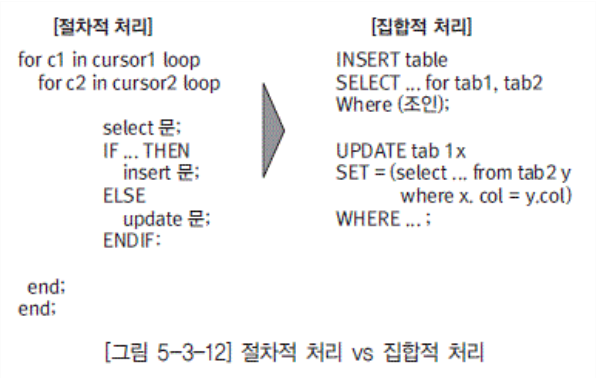
- 결과 행 중 일부에서 행에 대해 복잡한 연산이 필요한 경우
- 집계 부분으로 인해 전체 범위 처리로 수행되는 있는 SQL을 부분 범위 처리로 변경하고자 하는 경우
- 복잡한 계산 처리가 자주 변경되어 이를 통합 관리할 필요가 있는 경우(단, 과도한 함수 호출이 예 상되는 업무에 적용할 경우 부하를 감수해야 한다)

## 배치 프로그램의 성능 개선

데이터베이스에서 배치 프로그램의 성능 개선은 처리 시간의 최소화에 가장 큰 목적이 있다. IT 기 술이 발전하고 정보의 활용 요구가 증가하면서 데이터 량은 기하급수적으로 증가하고 있다. 단순히 업 무 수행을 위한 데이터 증가뿐만 아니라 DW, 고객, 상품, 수익 등을 대상으로 분석에 활용하기 위한 정보량도 증가하였다. 처리해야 할 데이터량이 증가하게 되어 배치 작업, ETL, 마트 생성 등을 처리하 기 위한 방안이 제시되어야 할 시점이다. 일부에서는 배치 작업을 위한 전용 서버??서 이슈화 되는 대표적인 요소는 다음과 같다.

- 절대 수행 시간 부족
- 수행 결과 검증 시간 확보의 어려움
- 오류에 따른 재처리 시간 확보가 불가능
- 미완료 시 대안 제시에 어려움
- 미처리 또는 지연으로 파급되는 문제 해결에 장시간이 소요됨

옵티마이저가 최적화를 수행하는 단위는 SQL 단위이므로 절차적인 방식으로 작성된 프로그램의 성능 개선 작업은 개별적인 SQL 단위로 이루어져야 한다. 그런데 개별적인 SQL의 최적화를 수행한 다고 해서 전체적인 프로그램의 최적화가 이루어지는 것은 아니므로 전체적인 최적화를 위해서는 절 차적인 프로그램을 집합적인 형태로 변경하여야 한다.




[그림 5-3-12] 절차적 처리 vs 집합적 처리

### 절차적인 처리 방식의 비효율

- SQL이 루프 내에서 반복적으로 수행되는 구조이므로 DBMS call이 과도하게 발생한다.
- 단위 SQL이 반복적으로 수행되는 구조이므로 Random I/O 발생이 증가한다.
- 동일 데이터를 반복해서 읽는다.
- 업무 규칙을 절차적으로 구성하였기 때문에 업무 규칙이 변경되면 프로그램 구조의 수정이 불가피하다.
- 다수의 단위 SQL로 구성되어 있어 개별적인 단위 SQL의 개선만 가능하다.

### 절차적인 처리방식의 보완 요소

- 이중 커서 사용을 하지 않고 조인을 이용하여 단일 커서를 사용한다.
- 동일 모듈 내에서는 같은 데이터를 2회 이상 읽지 않게 프로그램을 구조화한다.
- 최소한 개별적인 SQL 단위 비효율은 제거한다.

### 집합적인 처리방식의 고려사항

집합적 처리란 사용자가 기술한 SQL을 한 번의 DBMS Call로 결과 집합(Result Set)을 생성하는 DBMS 연산을 의미한다. 사용자는 결과 집합(WHAT)을 정의하고 DBMS는 처리 절차(How)를 결정 하므로 집합적인 처리 방식을 사용하려면 다음과 같은 사항을 고려해야 한다.

- DBMS는 실행 계획을 수립하기 위한 내부적인 매커니즘을 가지고 있지만 옵티마이저의 지능적 한 계나 정보의 부족 등 여러 가지 변수에 의하여 최적의 실행 계획을 수립하지 못하는 경우가 많이 발 생하므로 SQL 작성 후 원하는 방식으로 실행 계획이 수립되었는지 반드시 확인하여야 한다.
- 대량 배치 처리와 같이 대용량 데이터를 처리해야 하는 경우는 Hash 조인을 사용하는 것이 유리한다.
- 분포도가 나쁘면 Random IO 비효율이 급속도로 증가하므로 인덱스 스캔보다 Table full scan 방식이 유리하다.
- 절대적인 작업량이 정해져 있는 대용량 데이터 처리 시 병렬 처리(parallel processing)을 사용하 여 처리 시간을 단축한다.
- Hash 조인이나 집계를 위한 소트 작업을 고려하여 추가 메모리를 세션에 할당한다. 집합 처리에 의한 작업은 병행 처리, Full scan, 메모리 영역 확보 등으로 짧은 시간에 데이터베이 스의 자원을 확보하여 처리를 해야 한다. 따라서 작업의 종속성을 고려하여 배치 프로그램 작업 계획 을 수립하여 자원의 경합을 낮춰야 한다.

## 배치 프로그램의 성능 개선 방안

### 분석 함수(Analytic Function)를 통한 성능 개선 방안

초기 RDBMS는 집합 처리의 장점이 있었지만 포인터(Pointer)와 오프셋(Offset)에 대한 연산을 할 수 없는 것이 단점이었다. 따라서 표준 SQL로는 이와 관련된 연산을 하지 못했다. 이른 이유로 절차적 언어를 사용하거나 데이터를 복제하여 처리할 수 있는 SQL문을 작성하였다.

Red Brick은 데이터 분석이나 DSS(Decision-Support System)에 적합한 다양하고 강력한 기 능을 가진 SQL 문법을 제안하였다. 이 제안에는 집합적 개념인 표준 SQL문에서 처리가 어려워 절 차적으로 처리할 수밖에 없었던 업무 분석 요구를 수용하기 위해 포인터(Pointer)와 오프셋(offset)의 개념을 추가시킨 다양한 분석 함수의 다양한 기능을 포함한다.

분석 함수를 지원하는 관계형 데이터베이스를 사용하는 경우 자체 조인(Self-join) 또는 클라이언 트 프로그램의 절차적 로직으로 처리하거나 SQL문으로 표현하기 위해 고난도의 여러 기법을 적용 하였던 것을 Native SQL문에서 하나의 명령어로 바로 적용할 수 있게 되어 개발자가 명백하고 간 결한 SQL문으로 복잡한 분석 작업을 수행할 수 있으며, 개발 및 유지 보수가 편하므로 생산력이 향 상되었다. ANSI 표준 SQL로 채택되어 대부분의 데이터베이스에서 동일한 SQL 문법을 사용할 수 있다.

다음은 상용 데이터베이스인 오라클에서 제공하는 분석 함수들이다.

■ Ranking Family

대상 집합에 대하여 특정 칼럼(들) 기준으로 순위나 등급을 매기는 분석 함수 그룹으로, 다음과 같 은 종류가 있다. 예를 들어 RANK, DENSE_RANK, CUME_DIST, PERCENT_RANK, NTILE, ROW_NUMBER 등이 있다.

■ Window Aggregate Family

현재 행(current row)을 기준으로 지정된 윈도우(window) 내의 행들을 대상으로 집단화 (aggregation)를 수행하여 여러 가지 유용한 집계 정보(running summary, moving average 등)를 구하는 분석 함수군이다. SUM, AVG, MIN, MAX, STDDEV, VARIANCE, COUNT, FIRST_VALUE, LAST_VALUE 등이 있다.

■ Reporting Aggregate Family

서로 다른 두 가지의 집계 레벨을 비교하고자 하는 목적으로 사용하는 분석 함수군이다. SUM, AVG, MAX, MIN, COUNT, STDDEV, VARIANCE 등이 있다.

■ LEAD/LAG Family

서로 다른 두 행 값을 오프셋(Offset)으로 비교하기 위한 분석 함수이다. LEAD, LAG가 있다.

### 파티션 스토리지 전략을 통한 성능 향상 방안

대용량의 데이터를 신속하게 처리하기 위해서는 파티셔닝(분할)과 같은 스토리지 전략이 중요하 다. 파티셔닝은 대용량의 큰(지속적으로 증가하는) 테이블을 파티션이라는 보다 작은 단위로 나눔으 로써 성능이 저하되는 것을 방지하고 관리를 보다 수월하게 하고자 하는 개념이다. 파티셔닝은 칼럼 단위의 수직 파티션을 제공하고 있는 사이베이스를 제외한 DB2, 인포믹스, 오라클 등과 같은 대용 량의 데이터베이스 상용 벤더들은 범위 파티션(Range Partition)과 같이 유사한 형태의 파티션을 제공한다. 파티셔닝을 하게 되면 하나의 테이블이 동일한 논리적 속성을 공유하는 여러 개의 단위(파티션)로 나눠지게 된다. 각 파티션은 열(Column)과 제약 조건에 대한 정의를 공유하지만 별도의 세 그먼트로 저장되어 물리적인 속성인 블록 파라미터나 스토리지 파라미터를 독립적으로 지정할 수 있 다. 이러한 특성으로 파티셔닝 테이블은 다음과 같은 장점이 있다.

- 데이터 액세스 시에 파티션 단위로 액세스 범위를 줄여 I/O에 대한 성능 향상을 가져올 수 있다.
- 여러 분할 영역으로 나눔으로써 전체 데이터의 훼손 가능성이 감소하고 데이터의 가용성이 향상된다.
- 각 분할 영역을 독립적으로 백업하고 복구할 수 있다.
- 디스크 스트라이핑으로 I/O 성능을 향상(디스크 암에 대한 경합의 감소)시킬 수 있다.

성능 개선 방법론 | 조인(Join) | 애플리케이션 성능 개선 | 서버 성능 개선

## 객체 튜닝

테이블, 인덱스, 세그먼트에 관련한 사항이 대상이다.

- 객체는 성능을 고려하여 설계되어야 한다.
- 저장 장치를 이루는 블록, 확장 영역, 세그먼트에 관련된 사항을 튜닝한다.
- 인덱스는 삭제, 갱신으로 스큐 현상이 심한 경우는 재구성 작업이 필요하다.
- IO 병목이 발생하지 않게 물리적인 배치를 실시한다.

## 인스턴스 튜닝

DBMS 인스턴스는 메모리 부분과 프로세스가 튜닝 대상이다. 이는 DBMS 종속적인 요소가 많으므로 DBMS별 확인이 필요하다.

### 메모리

- 메모리는 Buffer 캐시, library 캐시 등의 히트율(HIT ratio)에 의해서 평가하여 조정한다.
- sort area, hash area는 스와핑(swapping) 발생 여부에 따라 사이즈를 결정한다. 특정한 대형 작업은 작업을 실시하는 세션에서 조정하여 작업을 실시한다.

### 프로세스

- 대부분의 DBMS가 다중 프로세스 시스템이고 필요에 따라서 추가적인 프로세스 가동이 가능하다.

### Latch 경합

- 트랜잭션 처리를 위한 경합이 발생한다.
- 객체 생성이나 변경 등으로 경합이 발생할 수 있다.

## 환경 튜닝

환경 튜닝은 하드웨어나 운영체제 관점에서의 튜닝이다. CPU, 메모리, 디스크 I/O, 네트워크가 환경 튜닝 대상이다. 환경 튜닝은 기본값으로 설정되어 있는 경우를 제외하고 많은 성능의 향상을 기 대하기 어렵다. 하드웨어 성능이나 구성에 따라 환경 설정된 상태에서 운영하기 때문이다.

예외적으로 고가용성을 위한 시스템 구성, RAID의 구성, 버전에 따른 패치 적용이 정상적이지 않을 경우에 성능에 결정적인 영향을 줄 수 있으므로 확인이 필요하다.

### CPU

- 튜닝 대상이라기보다는 성능을 평가하기 위한 기준으로 CPU 사용율(Utilization)을 평가한다.
- SAR(System Activity Report)로 모니터링 했을 때 CPU 사용이 %usr > %sys > %wio 순으로 되는 것이 바람직하다.
- %idle이 일반적으로 20~30%을 유지하는 것이 바람직하며, 0%인 상태가 지속적으로 유지되면 증설을 고려한다.
- Sun – ps, HP – top 또는 glance, IBM – monitor 등의 프로세스별 모니텅링이 가능하다.

이와같은 도구는 문제 프로세스를 OS 차원에서 확인할 수 있다.

### 메모리 튜닝

- Paging(page-in, page-out)과 프로세스 단위의 Paging 현상인 Swapping 발생 상태를 확인한다.
- DBMS를 포함한 사용자 사용 메모리 크기가 전체 크기의 40 ~ 60%를 유지하는 것이 바람직하다.

### I/O 튜닝

- 데이터베이스 병목은 I/O에 의해서 발생한다.
-

물리적인 디스크와 디스크 채널을 분산하므로 성능을 개선할 수 있다.
- 읽기/쓰기 작업에 따른 분산이 필요하다
- Cook Device보다는 Raw Device가 IO 성능에 유리하다.

■ 네트워크 튜닝