SQLD 준비하시는 분들 도움되시라고 공부하면서 직접 정리한 SQLD 'SQL 기본과 활용'입니다!

이것만 보고 시험을 다 준비하기는 어렵지만 노베이스에서 읽으면 80% 정도 얻어가실 수 있는 것 같아요!

시험 전 최종 정리본으로 보셔도 좋아요.

다들 합격하세요~👏 👏

"SQLD PART2" PDF - matiii

: https://hec-ker.tistory.com/

PART2: SQL 기본과 활용

SECTION01 SQL 기본

POINT1: 관계형 데이터베이스Relation DB

> *관계형 DB의 등장*

- 1960년대: 플로우차트 중심의 개발. 파일 구조를 통해 데이터를 저장/관리 함.
- 1970년대: DB 관리 기법이 태동되던 시기. 계층형Hierarchical DB, 망형Network DB같은의 상용화.
- 1980년대: 현재 대부분의 기업에서 사용되는 관계형 DB의 상용화. Oracle, Sybase, DB2의 등장.
- 1990년대: 인터넷 환경을 급속한 발전과 객체 지향 정보를 지원하기 위해 객체 관계형 DB로 발전.
    - 관계형 DB
        - 1970년대 E.F. Codd박사의 논문에서 처음 소개
        - 릴레이션에 데이터를 저장, 관리하며 릴레이션을 사용해 연산한다.
        - 관계Relation와 조인 연산을 통해 합집합/교집합/차집합 등을 만들 수 있다.

> *DB와 DBMS의 차이점*

DB는 데이터를 어떤 형태의 자료구조Data Structure로 사용하느냐에 따라 나뉜다.

- DB의 종류
    - 계층형 DB: **트리Tree** 형태에 데이터를 저장, 관리함. 1:N 관계를 표현함.
    - 네트워크형 DB: **오너Owner와 멤버Member** 형태로 데이터를 저장, 관리
    - 관계형 DB: 릴레이션에 데이터를 저장, 관리함. 릴레이션을 사용해 집합, 관계 연산함.
- DBMS
    - 계층형/네트워크형/관계형 DB 등을 관리하기 위한 SW
    - Oracle, MS-SQL, MySQL, Sybase 등

> *관계형 DB의 집합 연산*

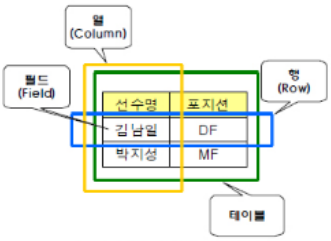
집합 연산	설명
합집합Union	두 릴레이션을 하나로 합하는 것 중복된 튜플(행)은 한 번만 조회
차집합Diference	본래 릴레이션에는 존재하고 다른 릴레이션에는 존재하지 않는 것을 조회
교집합Intersection	두 릴레이션 간 공통된 것을 조회
곱집합Cartesian product	각 릴레이션에 존재하는 모든 데이터를 조합하여 연산

> *관계형 DB의 관계 연산*

관계 연산	설명
선택 연산 Selection	릴레이션에서 조건에 맞는 행(튜플)만을 조회
투영 연산 Projection	릴레이션에서 조건에 맞는 속성만을 조회
결합 연산 Join	여러 릴레이션의 공통된 속성을 사용해 새로운 릴레이션을 만듦
나누기 연산Division	기준 릴레이션에서 나누는 릴레이션이 가진 속성과 동일한 값을 가지는 행(튜플)을 추출하고 나누는 릴레이션의 속성을 삭제한 후 중복된 행을 제거

> *테이블Table의 구조*

데이터는 관계형 데이터베이스의 기본 단위인 **테이블의 형태로 저장**된다.



[그림 Ⅱ-1-4] 테이블의 구조

[표 Ⅱ-1-5] 테이블 용어

용어	설명
테이블 (Table)	행과 칼럼의 2차원 구조를 가진 데이터의 저장 장소이며, 데이터베이스의 가장 기본적인 개념
칼럼/열 (Column)	2차원 구조를 가진 테이블에서 세로 방향으로 이루어진 하나하나의 특정 속성 (더이상 나눌 수 없는 특성)
행 (Row)	2차원 구조를 가진 테이블에서 가로 방향으로 이루어진 연결된 데이터

- 테이블은 행Row/튜플Tuple과 열Column로 구성된다.
- 칼럼은 어떤 데이터를 저장하기 위한 필드로 속성이라고도 한다.
- 정리하자면
  - 행 = 로우 = 튜플
  - 열 = 칼럼 = 속성

[그림 Ⅱ-1-5] 테이블의 정규화

[표 Ⅱ-1-6] 테이블 관계 용어들

용어	설명
정규화 (Normalization)	테이블을 분할하여 데이터의 정합성을 확보하고, 불필요한 중복을 줄이는 프로세스
기본키 (Primary Key)	테이블에 존재하는 각 행을 한 가지 의미로 특정할 수 있는 한 개 이상의 칼럼
외부키 (Foreign Key)	다른 테이블의 기본키로 사용되고 있는 관계를 연결하는 칼럼

- 기본키PK는 한 테이블에서 유일성(Unique)과 최소성(Not Null)을 만족하면서 해당 테이블을 대표한다.
- 외래키FK는 다른 테이블의 PK를 참조(조인)하는 컬럼이다.
    - FK는 관계 연산 중 결합 연산(조인)을 하기 위해 사용한다.

POINT2: SQL(Structured Query Language) 종류

> *SQL이란?*

- SQL은 관계형 DB에서 데이터 정의/조작/제어를 위해 사용하는 절차형 언어이다.
- SQL은 ANSI/ISO 표준을 준수하므로 DBMS가 변경되어도 그대로 사용할 수 있다.

SQL 표준

표준	설명
ANSI/ISO SQL 표준	INNER JOIN, NATURAL JOIN, USING 조건, ON 조건절 사용
ANSI/ISO SQL3 표준	DBMS 벤더 별 차이를 표준화하여 제정

SQL 실행 순서

- 개발자가 작성한 SQL문(DDL, DML, DCL 등)은 3단계를 걸쳐서 실행된다.
- 순서: 파싱(문법검사 - 구문분석) - 실행 - 인출

SQL 실행 순서	설명
파싱Parsing	SQL문의 문법 확인, 구문 분석 구문분석한 SQL문은 Library Cache에 저장
실행Execution	옵티마이저가 수립한 실행 계획에 따라 실행
인출Fetch	데이터를 읽어서 전송

파싱에 대해서

파싱은 소프트 파싱과 하드 파싱이 있다.

- 소프트 파싱
    - SQL 파싱 정보를 저장하는 라이브러리 캐시Library Cache에서 파싱된 정보를 찾을 수 있는 경우
    - 파싱 단계를 거치지 않고 바로 실행 단계로 넘어간다.
    - 효율성이 높다.
- 하드 파싱
    - 라이브러리 캐시에서 SQL문 파싱 정보를 찾을 수 없는 경우
    - 파싱 후 실행 단계로 넘어간다.
    - 처음 하드 파싱 후 변수의 값만 다른 SELECT문은 소프트 파싱된다.

> *☆SQL 종류*

[표 Ⅱ-1-1] SQL 문장들의 종류

명령어의 종류	명령어	설명
데이터 조작어 (DML: Data Manipulation Language)	SELECT	데이터베이스에 들어 있는 데이터를 조회하거나 검색하기 위한 명령어를 말하는 것으로 RETRIEVE 라고도 한다.
	INSERT UPDATE DELETE	데이터베이스의 테이블에 들어 있는 데이터에 변형을 가하는 종류의 명령어들을 말한다. 예를 들어 데이터를 테이블에 새로운 행을 집어넣거나, 원하지 않는 데이터를 삭제하거나 수정하는 것들의 명령어들을 DML이라고 부른다.
데이터 정의어 (DDL: Data Definition Language)	CREATE ALTER DROP RENAME	테이블과 같은 데이터 구조를 정의하는데 사용되는 명령어들로 그러한 구조를 생성하거나 변경하거나 삭제하거나 이름을 바꾸는 데이터 구조와 관련된 명령어들을 DDL이라고 부른다.
데이터 제어어 (DCL: Data Control Language)	GRANT REVOKE	데이터베이스에 접근하고 객체들을 사용하도록 권한을 주고 회수하는 명령어를 DCL이라고 부른다.
트랜잭션 제어어 (TCL: Transaction Control Language)	COMMIT ROLLBACK	논리적인 작업의 단위를 묶어서 DML에 의해 조작된 결과를 작업단위(트랜잭션) 별로 제어하는 명령어를 말한다.

- DB는 DDL 명령어와 DML 명령어를 다르게 처리한다.
  - DDL(CREATE, ALTER, RENAME, DROP) 명령어는 직접 DB 테이블에 영향을 미치기 때문에 DDL 명령어를 입력하는 순간 명령어에 해당하는 작업이 즉시 완료(AUTO COMMIT)된다.
  - 하지만 DML(INSERT, UPDATE, DELETE, SELECT) 명령어는, 조작하려는 테이블을 메모리 버퍼에 올리고 작업하기 때문에 테이블에 실시간으로 영향을 미치지는 않는다.
    - 따라서 버퍼에서 처리한 DML 명령어가 실제 테이블에 반영되기 위해서는 COMMIT 명령어를 입력하여 TRANSACTION을 종료해야 한다.
    - 하지만 SQL Server의 경우 DML 명령어도 DDL 명령어처럼 AUTO COMMIT 처리되므로 COMMIT 명령어가 필요없다.

작업 순서

DCL(권한 부여) - DDL (데이터 구조 정의) - DML(데이터 조회 등)

트랜잭션Transaction

DB의 작업을 처리하는 단위

트랜잭션의 특성

트랜잭션의 특성	설명
원자성 Atomicity	DB 연산의 전부또는 일부(ALL OR NOTHING) 실행만이 있다. 트랜잭션의 처리가 완전히 끝나지 않았을 때는 전혀 이루어지지 않은 것과 같아야 한다.
일관성 Consistency	트랜잭션 실행 결과로 DB 상태에 모순이 없고, 실행 후에도 일관성이 유지되어야 한다.
고립성 Isolation	트랜잭션 실행 중 생성하는 연산의 중간 결과는 다른 트랜잭션이 접근할 수 없다 (부분적인 실행 결과를 다른 트랜잭션이 볼 수 없다).
연속성 Durability	트랜잭션이 그 실행을 성공적으로 완료하면 그 결과는 영구적으로 보장되어야 한다.

POINT3: DDL(Data Definition Language)

> *테이블 생성*

- DB를 사용하기 위해서는 테이블을 먼저 생성해야 한다.
- 테이블 관리 SQL문

SQL문	설명
Create Table	새로운 테이블을 생성한다. 테이블을 생성할 때 기본키/외래키/제약사항 등을 설정할 수 있다.
Alter Table	생성된 테이블을 변경한다. 칼럼을 추가/변경/삭제할 수 있고 기본키/외래키를 설정할 수 있다.
Drop Table	해당 테이블을 삭제한다. 테이블의 데이터 구조와 저장된 데이터 모두 삭제된다.

기본적인 테이블 생성, Create Table문의 구조

- CREATE TABLE 구문

```
CREATE TABLE 테이블명 ( 칼럼명1 DATATYPE [DEFAULT 형식] primary key, 칼럼명2 DATATYPE [DEFAULT 형식], 칼럼명2 DATATYPE [DEFAULT 형식] );
```

- CREATE TABLE문의 구조

CREATE TABLE문	설명
1. CREATE TABLE	'CREATE TABLE A': 'A'라는 테이블을 생성하라는 뜻 '();': 괄호 사이에 칼럼을 쓰고 세미콜론으로 끝냄
2. 칼럼 정보	테이블에 생성되는 칼럼 이름(영문자)과 데이터 타입을 입력한다.
3. 데이터 타입	number: 숫자형 타입 / varchar2: 가변 길이 문자열 / char: 고정 길이 문자열 / date: 날짜형 타입
4. 기본키	칼럼 옆에 'primary key'를 입력하여 기본키로 지정

SELECT 문장을 통한 테이블 생성, CTAS

- CTAS 구문

```
CREATE TABLE 테이블명 AS SELECT 칼럼명 FROM 복사할 테이블명;
```

- 장점

칼럼별로 데이터 유형을 다시 재정의하지 않아도 된다.

- 단점

기존 테이블의 제약조건 중에 NOT NULL만 새로운 복제 테이블에 적용되고, PK, FK, 고유키, CHECK 등의 다른 제약 조건은 없어진다.

테이블의 구조 확인하기

```
DESC 테이블명;
```

CONSTRAINT제약조건 사용하기

- 제약조건이란 사용자가 원하는 조건의 데이터만 유지하기 위한, 즉 **데이터 무결성**을 위한 방법으로, 테이블의 특정 칼럼에 설정하는 제약이다.
- 테이블을 생성할 때 제약조건을 반드시 기술할 필요는 없지만 이후 ALTER TABLE을 이용해서 추가, 수정하는 경우 데이터가 이미 입력된 상태라면 처리 과정이 쉽지 않아서 초기 테이블 생성 시점부터 적합한 제약 조건에 대한 검토가 필요하다.
- 칼럼명 앞에 'constraint'를 입력하여 제약조건을 설정한다.

[표 Ⅱ-1-10] 제약조건의 종류

구분	설명
PRIMARY KEY (기본키)	테이블에 저장된 행 데이터를 고유하게 식별하기 위한 기본키를 정의한다. 하나의 테이블에 하나의 기본키 제약만 정의할 수 있다. 기본키 제약을 정의하면 DBMS는 자동으로 UNIQUE 인덱스를 생성하며, 기본키를 구성하는 칼럼에는 NULL을 입력할 수 없다. 결국 '기본키 제약 = 고유키 제약 & NOT NULL 제약'이 된다.
UNIQUE KEY (고유키)	테이블에 저장된 행 데이터를 고유하게 식별하기 위한 고유키를 정의한다. 단, NULL은 고유키 제약의 대상이 아니므로, NULL 값을 가진 행이 여러 개가 있더라도 고유키 제약 위반이 되지 않는다.
NOT NULL	NULL 값의 입력을 금지한다. 디폴트 상태에서는 모든 칼럼에서 NULL을 허가하고 있지만, 이 제약을 지정함으로써 해당 칼럼은 입력 필수가 된다. NOT NULL을 CHECK의 일부분으로 이해할 수도 있다.
CHECK	입력할 수 있는 값의 범위 등을 제한한다. CHECK 제약으로는 TRUE or FALSE로 평가할 수 있는 논리식을 지정한다.
FOREIGN KEY (외래키)	관계형 데이터베이스에서 테이블 간의 관계를 정의하기 위해 기본키를 다른 테이블의 외래키로 복사하는 경우 외래키가 생성된다. 외래키 지정시 참조 무결성 제약 옵션을 선택할 수 있다.

테이블 생성 시에 CASCADE 사용하기

- CASECADE 옵션은 참조 관계(PK와 FK 관계)가 있을 경우 참조되는 데이터도 자동으로 삭제할 수 있는 것이다.
- 다른 테이블을 참조하는 컬럼 앞에 'ON DELETE CASCADE'를 입력하여 사용할 수 있다.
  - 이 옵션을 통해 참조 무결성을 준수할 수 있다.
  - 마스터 테이블에는 해당 칼럼이 없는데 슬레이브 테이블에는 그 칼럼이 있는 경우, 참조 무결성 위배라고 한다.

> *테이블 변경*

- 테이블 변경은 ALTER TABLE문을 사용한다.
  - ALTER TABLE문은 칼럼 추가/변경/삭제 등을 할 수 있다.

테이블명 변경

```
ALTER TABLE 기존테이블명 RENAME TO 새로운테이블명;
```

칼럼 추가

```
ALTER TABLE 테이블명 ADD (추가할칼럼명 데이터유형);
```

칼럼 변경

```
ALTER TABLE 테이블명 MODIFY (칼럼명 데이터유형);
```

칼럼 삭제

```
ALTER TABLE 테이블명 DROP COLUMN 삭제할컬럼명;
```

칼럼명 변경

```
ALTER TABLE 테이블명 RENAME COLUMN 기존칼럼명 TO 새로운칼럼명;
```

제약조건 추가

```
ALTER TABLE 테이블명 ADD CONSTRAINT 제약조건명 제약조건 (칼럼명);
```

제약조건 삭제

```
ALTER TABLE 테이블명 DROP CONSTRAINT 제약조건명;
```

> *테이블 삭제*

- 테이블 삭제는 DROP TABLE문을 사용해서 삭제할 수 있다.
  - DROP TABLE은 테이블의 구조와 데이터를 모두 삭제한다.

DROP TABLE문

```
DROP TABLE 테이블명;
```

CASCADE CONSTRAINT 옵션

- CONSTRAINT 옵션은 해당 테이블과 관계가 있던 참조되는 제약조건에 대해서도 삭제한다는 것을 의미한다.

```
DROP TABLE 테이블명 CASCADE CONSTRAINT;
```

***TRUNCATE TABLE*, DROP TABLE 과 DELETE**

구분	테이블 정의 존재 유무	저장 공간	작업 속도	SQL 구분
DROP	삭제	반납	빠름	DDL
TRUNCATE	존재	반납	빠름	DDL
DELETE	존재	유지	느림	DML

- TRUNCATE TABLE은 테이블 자체가 삭제되는 것이 아니고, 해당 테이블에 들어있던 모든 행들이 제거되고 저장 공간을 재사용 가능하도록 해제한다.

    - 빠른 삭제를 위해 로그도 기록하지 않는다.
- 테이블 구조를 완전히 삭제하기 위해서는 DROP TABLE을 실행하면 된다.

> *뷰View의 생성과 삭제*

뷰란?

뷰란 테이블로부터 유도된 가상의 테이블이다.

- 실제 데이터를 가지고 있지 않고 테이블을 참조해서 원하는 칼럼만을 조회할 수 있게 한다.
- 뷰는 데이터 딕셔너리에 SQL문 형태로 저장하고 실행 시에 참조된다.

뷰의 생성

```
CREATE VIEW 뷰명 AS SELECT 칼럼명 FROM 참조할테이블명;
```

뷰의 조회

```
SELECT 칼럼명 FROM 뷰명;
```

뷰의 삭제

```
DROP VIEW 뷰명;
```

뷰의 특징

- 참조한 테이블이 변경되면 뷰도 변경된다.
- 뷰의 검색은 참조한 테이블과 동일하게 할 수 있지만, 뷰에 대한 입력/수정/삭제에는 제약이 있다.
- 뷰는 특정 칼럼만 조회하므로 **보안성을 향상**시킨다.
- 한 번 생성된 뷰는 변경이 불가능하다.
    - 변경을 원하면 삭제 후 재생성해야 한다.

뷰의 장단점

장점	단점
특정 칼럼만 조회할 수 있기 때문에 보안 기능이 있다.	독자적 인덱스를 만들 수 없다.
데이터 관리와 SELECT문이 간단해진다.	삽입/수정/삭제 연산에 제약이 있다.
한 테이블에 여러 뷰를 생성할 수 있다.	데이터 구조 변경이 불가능하다.

POINT4: DML(Data Manipulation Language)

> *INSERT문*

INSERT문

- INSERT문은 테이블에 데이터를 입력하는 DML문이다.

```
INSERT INTO 테이블명 (컬럼명1, 컬럼명2, ...) VALUES (값1, 값2, ...);
```

- 만약 테이블의 모든 칼럼에 데이터를 입력한다면 칼럼명을 생략한다.
    - 칼럼 순서대로 빠짐없이 데이터가 입력되어야 한다.

```
INSERT INTO 테이블명 VALUES (값1, 값2, ...);
```

- INSERT문을 실행했다고 데이터 파일에 저장되는 것은 아니고 최종적으로 TCL문인 COMMIT을 실행해야 한다.

SELECT문

- 테이블1의 특정 칼럼을 테이블2에 입력하기

```
INSERT INTO 테이블2 SELECT 칼럼명 FROM 테이블1;
```

- 테이블1의 모든 데이터를 테이블2에 입력하기

```
INSERT INTO 테이블2 SELECT * FROM 테이블1;
```

Nologging 옵션

- DB에 데이터를 입력하면 로그파일에 그 정보를 기록한다.
- Check point라는 이벤트가 발생하면 로그파일 데이터를 데이터 파일에 저장한다.
- Nologging 옵션은 로그파일의 기록을 최소화시켜 입력 시 성능을 향상시키는 방법으로, Buffer Cache라는 메모리 영역을 생략하고 기록한다.
- INSERT문에만 효과가 있다.
    - Delete와 Update에서는 지원되지 않는다.

```
ALTER TABLE 테이블명 NOLOGGING;
```

> *UPDATE문*

- 입력된 데이터의 값을 수정하려면 UPDATE문을 사용한다.
    - UPDATE문은 원하는 조건으로 검색해서 해당 데이터를 수정하는 것이다.
    - UPDATE문에 조건문을 입력하지 않으면 모든 데이터가 수정되므로 유의해야 한다.

UPDATE문의 구조

```
UPDATE 테이블명 SET 수정할칼럼명 = 새로운값 WEHRE 조건;
```

> *DELETE문*

- DELETE문은 원하는 조건을 검색해서 해당되는 행을 삭제한다.
    - 조건문을 입력하지 않으면 테이블의 모든 데이터가 삭제된다.

DELETE문의 구조

- 테이블 내 행 삭제

```
DELETE FROM 테이블명 WHERE 조건;
```

- 테이블 삭제

```
DELETE 삭제할 테이블명;
```

> *SELECT문*

- 테이블에 입력된 데이터를 조회하기 위해서 SELECT문을 사용한다.
    - SELECT문은 특정 칼럼이나 특정 행만을 조회할 수 있다.

1) SELECT문의 구조

```
SELECT 조회할데이터 FROM 테이블명 WHERE 조건;
```

SELECT문 예제

```
SELECT * FROM EMP WHERE 사번 = 10;
```

SELECT 문법	설명
1. SELECT *	'*(애스터리스크)'는 모든 칼럼을 의미한다. 즉, 모든 칼럼을 출력한다
2. FROM EMP	FROM절에는 테이블명을 쓴다. 즉, EMP 테이블을 지정하였다
3.WHERE 사번 = 10;	WHERE절에서는 조건문을 지정한다. 즉, EMP 테이블에서 사번이 10인 행을 조회한다.

SELECT 칼럼 지정 예제

```
SELECT A || '맛' FROM FLAVOUR;
```

- FLAVOUR 테이블의 모든 행에서 'A' 칼럼을 조회한다.
- 'A' 칼럼 뒤에 '맛'이라는 문자를 결합한다.
- 'A맛'이라는 형태로 출력된다.

2) Distinct와 Alias

- DISTINCT
    - DISTINCT 옵션은 칼럼명 앞에 지정하여 중복된 데이터를 한 번만 조회하게 한다.
    - 모든 조인을 다 실행한 다음에 중복을 제거한다.
        - 그래서 효율적이지는 않다.

```
SELECT DISTINCT 칼럼명 FROM 테이블명 WHERE 조건;
```

- ALIAS
    - 조회된 결과에 일종의 별명(ALIAS, ALIASES)을 부여해서 칼럼명/테이블명을 변경할 수 있다.

```
SELECT 칼럼명 [AS] 별칭 FROM 테이블명 WHERE 조건;
```

3) Order by를 사용한 정렬

- Order by는 데이터를 오름차순(ASC) 또는 내림차순(DESC)으로 출력 직전에 정렬한다.

- 기본값은 **ASC(오름차순)**이며 내림차순으로 정렬하고 싶을 때는 Order by 절 가장 마지막에 **'DESC'**라고 명시한다.
- Order by는 정렬하므로 DB 메모리를 많이 사용하여 성능 저하가 발생한다.

```
SELECT 칼럼명 FROM 테이블명 ORDER BY 칼럼명 [DESC];
```

Index를 사용한 정렬 회피

- 정렬은 DB에 부하를 주기 때문에 인덱스(기본키)를 사용해 회피할 수 있다.
- 기본키를 지정하면 자동으로 기본키에 대한 오름차순 인덱스가 생성된다.
- 내림차순으로 출력하고 싶다면 **힌트**의 개념을 사용한다.
  - SELECT 칼럼명 /*+ INDEX_DESC(테이블명) */ FROM 테이블명;
  - 위처럼 힌트(/*+ INDEX_DESC(테이블명) */)를 사용한 SELECT 문을 실행하면 PK에 대해 내림차순으로 출력된다.

POINT5: WHERE문

> *WHERE문의 연산자*

- 연산자 종류

[표 Ⅱ-1-15] 연산자의 종류

구분	연산자	연산자의 의미
비교 연산자	=	같다.
	〉	보다 크다.
	〉=	보다 크거나 같다.
	〈	보다 작다.
	〈=	보다 작거나 같다.
SQL 연산자	BETWEEN a AND b	a와 b의 값 사이에 있으면 된다.(a와 b 값이 포함됨)
	IN (list)	리스트에 있는 값 중에서 어느 하나라도 일치하면 된다.
	LIKE '비교문자열'	비교문자열과 형태가 일치하면 된다.(%, _ 사용)
	IS NULL	NULL 값인 경우
논리 연산자	AND	앞에 있는 조건과 뒤에 오는 조건이 참(TRUE)이 되면 결과도 참(TRUE)이 된다. 즉, 앞의 조건과 뒤의 조건을 동시에 만족해야 한다.
	OR	앞의 조건이 참(TRUE)이거나 뒤의 조건이 참(TRUE)이 되어야 결과도 참(TRUE)이 된다. 즉, 앞뒤의 조건 중 하나만 참(TRUE)이면 된다.
	NOT	뒤에 오는 조건에 반대되는 결과를 되돌려 준다.
부정 비교 연산자	!=	같지 않다.
	^=	같지 않다.
	◇	같지 않다.(ISO 표준, 모든 운영체제에서 사용 가능)
	NOT 칼럼명 =	~와 같지 않다.
	NOT 칼럼명 〉	~보다 크지 않다.
부정 SQL 연산자	NOT BETWEEN a AND b	a와 b의 값 사이에 있지 않다. (a, b 값을 포함하지 않는다)
	NOT IN (list)	list 값과 일치하지 않는다.
	IS NOT NULL	NULL 값을 갖지 않는다.

- 연산자의 우선순위

[표 Ⅱ-1-17] 비교 연산자의 종류

연산자	연산자의 의미
=	같다.
〉	보다 크다.
〉=	보다 크거나 같다.
〈	보다 작다.
〈=	보다 작거나 같다.

- 추가내용: 문자 유형 비교법

[표 Ⅱ-1-18] 문자 유형 비교 방법

구분	비교 방법
비교 연산자의 양쪽이 모두 CHAR 유형 타입인 경우	길이가 서로 다른 CHAR형 타입이면 작은 쪽에 SPACE를 추가하여 길이를 같게 한 후에 비교한다.
	서로 다른 문자가 나올 때까지 비교한다.
	달라진 첫 번째 문자의 값에 따라 크기를 결정한다.
	BLANK의 수만 다르다면 서로 같은 값으로 결정한다.
비교 연산자의 어느 한 쪽이 VARCHAR 유형 타입인 경우	서로 다른 문자가 나올 때까지 비교한다.
	길이가 다르다면 짧은 것이 끝날 때까지만 비교한 후에 길이가 긴 것이 크다고 판단한다.
	길이가 같고 다른 것이 없다면 같다고 판단한다.
	VARCHAR는 NOT NULL까지 길이를 말한다.
상수값과 비교할 경우	상수 쪽을 변수 타입과 동일하게 바꾸고 비교한다.
	변수 쪽이 CHAR 유형 타입이면 위의 CAHAR 유형 타입의 경우를 적용한다.
	변수 쪽이 VARCHAR 유형 타입이면 위의 VARCHAR 유형 타입의 경우를 적용한다.

Like문의 사용

- Like문은 와일드카드를 사용해서 데이터를 조회할 수 있다.
- Like문 뒤에 와일드카드를 사용하지 않으면 '='와 같다.

와일드카드	설명
%	어떤 문자를 포함한 모든 것을 조회한다. ex. '%es%'는 문자 중간에 'es'가 포함된 모든 문자를 조회한다 정상적인 Index Range Scan이 불가능하다.
_	한 개인 단일 문자를 의미한다. ex. 'name_'은 'name'으로 시작하고 그 뒤에 한 글자만 더 붙는 것을 조회한다. (name1, name2, ...)
*	all, 모든 것을 의미한다.

Between문의 사용

- Between문은 지정된 범위에 있는 값을 조회한다.

```
SELECT 컬럼명 FROM 테이블명 WHERE 컬럼명 BETWEEN 조건1 AND 조건2;
```

예를 들어, SELECT * FROM EMP WHERE SAL BETWEEN 1000 AND 2000; 이라면, EMP 테이블에서 SAL(급여)이 1000에서 2000사이인 직원을 조회한다.

이 때 1000과 2000도 포함한다.

반대로 1000과 2000 사이가 아닌 것을 조회하고 싶다면 BETWEEN 앞에 'NOT'을 붙인다.

> *IN문의 사용*

- IN문은 "OR"의 의미를 가지고 있어서 한 조건만 만족해도 조회가 가능하다.

```
SELECT * FROM EMP WHERE JOB IN ('CLERK', 'MANAGER');
```

라면, JOB이 'CLERK'이나 'MANAGER'인 행을 조회한다.

- IN 조건에 여러 개의 칼럼을 사용할 수도 있다.

```
SELECT * FROM EMP WHERE (JOB, ENAME) IN (('CLERK', 'SALES'), ('aaa',
'bbb'));
```

라면, JOB이 'CLERK'이거나 'SALES'이면서 ENAME이 'aaa'이거나 'bbb'인 행을 조회한다.

> *NULL값 조회*

NULL의 특징

- NULL은 모르는 값, 값의 부재를 뜻한다.
- NULL과의 모든 비교는 '알 수 없음'을 반환한다.
- NULL에 숫자/날짜를 더하면 NULL이 된다.
- NULL은 비교연산자로 비교할 수 없다.
    - 비교한다면 거짓(False)가 나온다.

NULL값 조회하기

```
SELECT * FROM 테이블명 WHERE 칼럼명 IS NULL;
```

```
SELECT * FROM 테이블명 WHERE 칼럼명 IS NOT NULL;
```

NULL 관련 함수

NULL 함수	예시
NVL()	NVL(MGR, 0): MGR 칼럼이 NULL이면 0으로 바꾼다.
NVL2()	NVL2(MGR, 1, 0): MGR 칼럼이 NULL이 아니면 1을, NULL이면 0을 반환한다.
NULLIF()	NULLIF(n1, n2): n1과 n2가 같으면 NULL을, 같지 않으면 n1을 반환한다.
COALESCE()	COALESCE(mgr, 1): mgr이 NULL이 아니면 1을 반환한다.

추가내용:

POINT6: GROUP 연산

> *GROUP BY문*

- GROUP BY는 테이블에서 행을 소규모 그룹화하여 합계, 평균, 최대, 최소 등을 계산할 수 있다.
- HAVING 절에 조건문을 사용한다.
- ORDER BY절도 사용할 수 있다.

```
SELECT DPT_ID, SUM(SAL) FROM EMP GROUP BY DPT_ID;
```

위의 예시에서 조회하는 것은 부서 아이디(DPT_ID) 별 직원들의 월급(SAL) 합계이다.

만약 부서 아이디가 10, 11, 12, 13인 직원들의 월급 합계를 조회한다면 아래와 같다.

```
SELECT DPT_ID, SUM(SAL) FROM EMP WHERE DPT_ID BETWEEN 10 AND 13 GROUP BY
DPT_ID;
```

> *HAVING문 사용*

- GROUP BY문에 조건절을 사용하려면 HAVING문을 사용해야 한다.
- 만약 WHERE 절에 조건문을 사용하게 된다면, GROUP BY 대상에서 제외된다.

```
SELECT DPT_ID, SUM(SAL) FROM EMP GROUP BY DPT_ID HAVING SUM(SAL) > 30000;
```

위의 예시에서는 부서 아이디(DPT_ID) 별 직원들의 월급(SAL) 합계를 출력하지만, HAVING문의 조건으로 인해 월급 합계가 30000을 넘는 행들만 조회한다.

> *집계함수의 종류*

- WHERE 절에 사용할 수 없다!

집계함수	설명
COUNT()	행 수를 조회한다 COUNT(*): NULL 행 포함해서 계산 COUNT(칼럼): NULL값을 제외한 행 수 계산
SUM()	합계를 계산한다
AVG()	평균을 계산한다
MAX()와 MIN()	최대값/최소값을 계산한다
STDDEV()	표준편차를 계산한다
VARIAN()	분산을 계산한다

POINT7: SELECT문의 실행 순서

- GROUP BY 절과 ORDER BY절이 같이 사용될 때, SELECT 문장은 6개의 절로 구성된다.
    - SELECT 칼럼명 [별칭]
    - FROM 테이블명
    - WHERE 조건식
    - GROUP BY 컬럼/표현식
    - HAVING 그룹조건식
    - ORDER BY 칼럼/표현식;

- SELECT 문장의 수행 단계는 아래와 같다.
    1. 발췌 대상 테이블 참조 (FROM)
    2. 발췌 대상 데이터가 아닌 것은 제거한다 (WHERE)
    3. 행들을 소그룹화 한다 (GROUP BY)
    4. 그룹핑된 값의 조건에 맞는 것만을 출력한다 (HAVING)
    5. 데이터 값을 출력/계산한다 (SELECT)
    6. 데이터를 정렬한다 (ORDERY BY)

POINT8: 명시적Esxplicit 형변환과 암시적Implicit 형변환

- 형변환이란, 두 개의 데이터의 데이터 타입(형)이 일치하도록 변환하는 것이다.
    - 명시적 형변환과 암시적 형변환으로 나뉜다.
- **인덱스 칼럼에 형변환을 수행한다면 인덱스를 사용하지 못한다.**

명시적 형변환과 암시적 형변환

- 명시적 형변환
    - 형변환 함수를 사용해서 데이터 타입을 일치시키는 것
    - 개발자가 SQL을 사용할 때 형변환 함수를 사용해야 한다.
- 암시적 형변환
    - 개발자가 형변환을 하지 않은 경우, DBMS가 자동으로 형변환하는 것

형변환 함수

형변환 함수	설명
TO_NUMBER(문자열)	문자열을 숫자로 변환
TO_CHAR(숫자/날짜, [FORMAT])	숫자/문자를 지정된 FORMAT의 문자로 변환
TO_DATE(문자열, FORMAT)	문자열을 지정된 FORMAT의 날짜형으로 변환

POINT9: 내장형 함수BUILT-IN Function

> *내장형 함수*

모든 DB는 SQL에서 사용할 수 있는 내장형 함수를 가진다.

> *DUAL 테이블*

- DUAL 테이블은 Oracle DB에 의해 자동으로 생성되는 테이블이다.
    - Oracle DB 사용자가 임시로 사용할 수 있는 테이블로, 내장형 함수를 실행할 때도 사용할 수 있다.
    - Oracle DB의 모든 사용자가 사용할 수 있다.
    - Oracle은 기본적으로 DUAL 테이블이라는 Dummy 테이블이 존재한다.

```
SELECT 1 FROM DUAL
    UNION SELECT 2 FROM DUAL
    UNION SELECT 1 FROM DUAL;
```

위 쿼리 실행 결과는 '1, 2'이다. (UNION은 정렬하며)

> *내장형 함수의 종류*

[표 Ⅱ-1-24] 단일행 함수의 종류

종류	내용	함수의 예
문자형 함수	문자를 입력하면 문자나 숫자 값을 반환한다.	LOWER, UPPER, SUBSTR/SUBSTRING, LENGTH/LEN, LTRIM, RTRIM, TRIM, ASCII,
숫자형 함수	숫자를 입력하면 숫자 값을 반환한다.	ABS, MOD, ROUND, TRUNC, SIGN, CHR/CHAR, CEIL/CEILING, FLOOR, EXP, LOG, LN, POWER, SIN, COS, TAN
날짜형 함수	DATE 타입의 값을 연산한다.	SYSDATE/GETDATE, EXTRACT/DATEPART, TO_NUMBER(TO_CHAR(d,'YYYY'|'MM'|'DD')) / YEAR|MONTH|DAY
변환형 함수	문자, 숫자, 날짜형 값의 데이터 타입을 변환한다.	TO_NUMBER, TO_CHAR, TO_DATE / CAST, CONVERT
NULL 관련 함수	NULL을 처리하기 위한 함수	NVL/ISNULL, NULLIF, COALESCE

※ 주: Oracle함수/SQL Server함수 표시, '/' 없는 것은 공통 함수

문자형 함수

문자형 함수	설명
ASCII(문자)	문자/숫자 -> ASCII 코드로 변환
CHAR(ASCII 값)	ASCII 코드 -> 문자로 변환
SUBSTR(문자열, m, n)	문자열 m번째 위치부터 n개를 자름
CONCAT(문자열1, 문자열2)	문자열1과 문자열2를 결합(||)
LOWER(문자열)	영문자를 소문자로 변환
UPPER(문자열)	영문자를 대문자로 변환
LENGTH(문자열) or LEN(문자열)	공백을 포함한 문자열의 길이를 알려줌
LTRIM(문자열, 지정문자)	왼쪽에 지정된 문자를 삭제 지정문자를 생략하면 공백을 삭제함
RTRIM(문자열, 지정문자)	오른쪽에 지정된 문자를 삭제 지정문자를 생략하면 공백을 삭제함
TRIM(문자열, 지정된 문자)	양쪽에 지정된 문자를 삭제 지정문자를 생략하면 공백을 삭제함

- 예시

[표 Ⅱ-1-26] 단일행 문자형 함수 사례

문자형 함수 사용	결과 값 및 설명
LOWER('SQL Expert')	'sql expert'
UPPER('SQL Expert')	'SQL EXPERT'
ASCII('A')	65
CHR(65) / CHAR(65)	'A'
CONCAT('RDBMS',' SQL') 'RDBMS' || ' SQL' / 'RDBMS' + ' SQL'	'RDBMS SQL'
SUBSTR('SQL Expert', 5, 3) SUBSTRING('SQL Expert', 5, 3)	'Exp'
LENGTH('SQL Expert') / LEN('SQL Expert')	10
LTRIM('xxxYYZZxYZ','x') RTRIM('XXYYzzXYzz','z') TRIM('x' FROM 'xxYYZZxYZxx')	'YYZZxYZ' 'XXYYzzXY' 'YYZZxYZ'
RTRIM('XXYYZZXYZ ') → 공백 제거 및 CHAR와 VARCHAR 데이터 유형을 비교할 때 용이하게 사용된다.	'XXYYZZXYZ'

날짜형 함수

날짜형 함수	설명
SYSDATE	오늘의 날짜를 날짜 타입으로 알려줌
EXTRACT('YEAR' | 'MONTH' | 'DAY' from d)	날짜에서 년 ,월, 일을 조회

- 연산

[표 Ⅱ-1-30] 단일행 날짜형 데이터 연산

연산	결과	설명
날짜 + 숫자	날짜	숫자만큼의 날수를 날짜에 더한다.
날짜 - 숫자	날짜	숫자만큼의 날수를 날짜에서 뺀다.
날짜1 - 날짜2	날짜수	다른 하나의 날짜에서 하나의 날짜를 빼면 일수가 나온다.
날짜 + 숫자/24	날짜	시간을 날짜에 더한다.

숫자형 함수

숫자형 함수	설명
ABS(숫자)	절대값 반환
SIGN(숫자)	양수, 음수, 0 구별
MOD(숫자1, 숫자2)	숫자1을 숫자2로 나눈 나머지를 계산 (%)
CEIL/CEILING(숫자)	숫자보다 크거나 같은 최소의 정수를 반환
FLOOR(숫자)	숫자보다 작거나 같은 최대의 정수를 반환
ROUND(숫자, m)	숫자를 소수점 m자리까지 반올림하고, 출력한다. 기본값: 0
TRUNC(숫자, m)	숫자를 소수점 m자리에서 버림

- 예시

[표 Ⅱ-1-28] 단일행 숫자형 함수 사례

숫자형 함수 사용	결과 값 및 설명
ABS(-15)	15
SIGN(-20) SIGN(0) SIGN(+20)	-1 0 1
MOD(7,3) / 7%3	1
CEIL(38.123) / CEILING(38.123) CEILING(-38.123)	39 39 -38
FLOOR(38.123) FLOOR(-38.123)	38 -39
ROUND(38.5235, 3) ROUND(38.5235, 1) ROUND(38.5235, 0) ROUND(38.5235)	38.524 38.5 39 39 (인수 0이 Default)
TRUNC(38.5235, 3) TRUNC(38.5235, 1) TRUNC(38.5235, 0) TRUNC(38.5235)	38.523 38.5 38 38 (인수 0이 Default)

POINT10: DECODE와 CASE문

> *DECODE문*

DEOCDE문으로 IF문을 구현할 수 있다. 즉, 특정 조건이 참이면 A, 거짓이면 B로 응답한다.

```
SELECT DECODE(DPT_ID, 23, 'TRUE', 'FALSE') FROM EMP;
```

위의 SQL 쿼리에서 DPT_ID가 23이면 TRUE를 응답하고 그렇지 않으면 FALSE를 응답한다.

> *CASE문*

CASE문은 IF~THEN ELSE-END 구문을 상요해 조건문으로 사용한다.

```
SELECT CASE
        THEN DPT_ID = 23 THEN 'GROUP A'
        THEN DPT_ID = 24 THEN 'GROUP B'
        ELSE 'GROUP C'
    FROM EMP;
```

위 쿼리에서 DPT_ID가 23이면 'GROUP A'를, 24면 'GROUP B'를, 그렇지 않으면 'GROUP C'를 출력한다.

POINT11:ROWNUM과 ROWID

> *ROWNUM*

- ROWNUM은 SELECT문의 결과에 대해 논리적인 일렬번호를 부여한다.
  - 조회되는 행 수를 제한할 때 자주 사용된다.
- ROWNUM을 사용해서 한 개의 행을 가지고 올 수는 있지만 여러 개의 행을 가지고 올 때는 인라인 뷰Inline view(FROM절에 SELECT문을 쓰는 것)를 사용하고 ROWNUM에 별칭을 붙여야 한다.
  - `SELECT * FROM EMP WHERE ROWNUM <=1; #한 행 조회`
  - `SELECT * FROM (SELECT ROWNUM list, ENAME FROM EMP) WHERE list <= 5;`

> *ROWID*

- 모든 테이블은 ROWID를 가진다.
- ROWID는 ORACLE DB 내에서 데이터를 구분할 수 있는 유일한 값이다.
  - "SELECT ROWID, EMPNO FROM EMP"와 같은 SELECT문으로 확인할 수 있다.
  - ROWID는 데이터가 어떤 데이터 파일, 어떤 블록에 저장되어 있는지를 알 수 있다.

```
SELECT ROWID FROM 테이블명;
```

ROWID 구조

구조	길이	설명
오브젝트 번호	1~6	오브젝트 별로 가진 유일한 값. 해당 오브젝트가 속한 값
상대 파일 번호	7~9	tablespace에 속한 데이터 파일에 대한 상대 파일 번호
블록 번호	10~15	데이터가 데이터 파일 내부의 어느 블록에 있는 지를 의미
데이터 번호	16~18	데이터가 데이터 블록에 저장된 순서를 의미

POINT12: WITH구문

- WITH 구문은 서브쿼리Subquery를 사용해 임시 테이블이나 뷰처럼 사용할 수 있다.
- 서브쿼리 블록에 별칭을 지정할 수도 있다.

```
WITH 임시테이블명 AS (SELECT 칼럼명 FROM 테이블명)
    SELECT * FROM 임시테이블명
```

POINT13: DCL(Data Control Language)

> *GRANT*

GRANT문은 DB 사용자에게 권한을 부여한다.

- DB 사용에 권한을 부여해 연결/입력/수정/삭제/조회를 할 수 있다.

GRANT문 구조

```
GRANT 권한 ON 테이블명 TO 사용자;
```

권한Privileges

권한	설명
SELECT	지정된 테이블에 대해 SELECT 권한 부여
INSERT	지정된 테이블에 대해 INSERT 권한 부여
UPDATE	지정된 테이블에 대해 UPDATE 권한 부여
DELETE	지정된 테이블에 대해 DELETE 권한 부여
REFERENCES	지정된 테이블을 참조하는 제약조건을 생성하는 권한 부여
ALTER	지정된 테이블에 대해 수정 권한 부여
INDEX	지정된 테이블에 대해 인덱스를 생성할 수 있는 권한 부여
ALL	테이블에 대한 모든 권한 부여

GRANT 옵션

GRANT 옵션	설명
WITH GRANT OPTION	"특정 사용자에게 권한을 부여할 수 있는" 권한을 부여 A가 B에게 권한을 부여하고, B가 C에게 다시 권한을 부여한 후 권한을 취소(Revoke)하면 모든 권한이 회수된다.
WITH ADMIN OPTION	테이블에 대한 모든 권한 부여 A가 B에게 권한을 부여하고, B가 C에게 다시 권한을 부여한 후 권한을 취소(Revoke)하면 B의 권한만 취소된다.

> *REVOKE*

- REVOKE문은 DB 사용자에게 부여된 권한을 회수한다.

```
REVOKE 권한 ON 테이블명 FROM 사용자;
```

POINT14: TCL(Transaction Control Language)

> *COMMIT*

입력/수정/삭제한 자료에 대해서 전혀 문제가 없다고 판단되었을 경우 COMMIT 명령어를 통해 트랜잭션을 완료한다.

COMMIT이나 ROLLBACK 이전의 데이터 상태는 다음과 같다.

- 단지 메모리 BUFFER에만 영향을 받았기 때문에 데이터의 변경 이전 상태로 복구 가능하다.
- 현재 사용자는 SELECT 문장으로 결과를 확인 가능하다.
- 다른 사용자는 현재 사용자가 수행한 명령의 결과를 볼 수 없다.
- 변경된 행은 잠금LOCKING이 설정되어 다른 사용자가 변경할 수 없다.

COMMIT은 이렇게 INSERT, UPDATE, DELETE 문장을 사용한 후 이런 변경 작업이 완료되었음을 DB에 알려 주기 위해 사용한다.

COMMIT 이후의 데이터 상태는 다음과 같으며 COMMIT으로 하나의 트랜잭션 과정을 종료한다.

- 데이터에 대한 변경 사항이 데이터베이스에 반영된다.
- 이전 데이터는 영원히 잃어버린다.
- 모든 사용자는 결과를 볼 수 있다.
- 관련된 행에 대한 잠금이 풀리고, 다른 사용자들이 행을 조작할 수 있게 된다.
- 하나의 트랜잭션 과정이 종료된다.

AUTO COMMIT

- SQL*PLUS 프로그램, DDL 및 DCL을 사용하는 경우 자동 커밋된다
- "set autocommit on;"을 실행하면 자동 커밋된다

> *ROLLBACK*

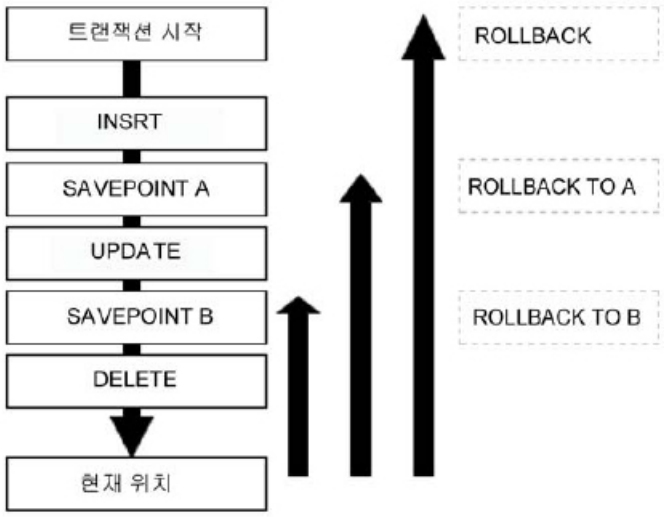
ROLLBACK을 실행하면 데이터에 대한 변경 사용을 모두 취소하고 트랜잭션을 종료한다.

- INSERT/UPDATE/DELETE문의 모든 작업을 취소한다.
- 이전 COMMIT한 곳까지만 복구한다.
- ROLLBACK을 실행하면 UNLOCK되고 다른 사용자도 DB 행을 조작할 수 있다.

COMMIT과 ROLLBACK을 사용함으로써 다음과 같은 효과를 볼 수 있다.

- **데이터 무결성 보장**
- 영구적인 변경을 하기 전에 데이터의 변경 사항 확인 가능
- 논리적으로 연관된 작업을 그룹핑하여 처리 가능

> *저장점SAVEPOINT*



[그림 Ⅱ-1-11] ROLLBACK 원리 (Oracle 기준)

SAVEPOINT는 트랜잭션을 작게 분할하여 관리하는 것으로, 지정된 위치까지만 트랜잭션을 ROLLBACK 할 수 있다.

복수의 저장점을 정의할 수 있으며, 동일이름으로 저장점을 정의했을 때는 나중에 정의한 저장점이 유효하다.

복잡한 대규모 트랜잭션에서의 에러가 발생했다면 저장점까지의 트랜잭션만 롤백하여 실패한 부분에 대해서만 다시 실행할 수 있다.

- SAVEPOINT 정의

```
SAVEPOINT 세이브포인트명;
```

- 정의한 SAVEPOINT까지 ROLLBACK

```
ROLLBACK TO 세이브포인트명;
```

만약 'ROLLBACK'을 실행하면 SAVEPOINT와 관계없이 변경된 모든 데이터를 취소한다.

PART2: SQL 기본과 활용

SECTION02 SQL 활용

POINT1: 조인Join

> *EQUI(등가) 조인(교집합)*

EQUI(등가) 조인이란?

- 조인은 여러 개의 릴레이션을 사용해 새로운 릴레이션을 만드는 과정이다.
    - 조인의 가장 기본을 교집합을 만드는 것
    - 두 테이블 간 일치하는 것을 조인한다.
    - HASH 조인은 EQUI 조인에서만 사용 가능하다.

```
SELECT * FROM EMP, DPT WHERE EMP.NO = DPT.NO;
```

"="로 두 개의 테이블을 연결한다.

```
SELECT * FROM OEMP, DPT WHERE EMP.NO = DPT.NO AND EMP.NAME LIKE '강%'
ORDER BY NAME;
```

조인문에 추가 조건이나 정렬문도 사용할 수 있다.

INNER JOIN

INNER JOIN은 'ON구'를 사용해 테이블을 연결한다.

```
SELECT * FROM EMP INNER JOIN DPT ON EMP.NO = DPT.NO;
```

이렇게 INNER JOIN구에 두 테이블 명을 서술하고 ON구를 사용해 조인 조건을 넣는다.

INTERSECT 연산

- INTERSECT 연산은 두 테이블에서 교집합을 조회한다.
    - 즉 두 테이블에서 공통된 값을 조회한다.

> *Non-EQUI(비등가) 조인*

- Non-EQUI는 두 개의 테이블 간 조인하는 경우 "=" ㅇ ㄹ 사용하지 않고 ">", "<", ">=", "<=" 등을 사용한다.
    - 즉 정확하게 일치하지 않는 것을 조인한다.

> *OUTER JOIN*

- Outer join은 두 테이블 간 교집합(EQUI join)을 조회하고 한 쪽 테이블에만 있는 데이터도 포함시켜서 조회한다.
    - DPT테이블과 EMP테이블을 Outer join하면 DNO가 같은 것을 조회하고, DPT 테이블에만 있는 NO도 포함시킨다.
    - 이 때 왼쪽 테이블에만 있는 행도 포함하면 Left Outer join, 오른쪽 테이블의 행만 포함시키면 Right Outer join이라고 한다.
    - FULL Outer join은 Left Outer join과 Right Outer join을 모두 사용하는 것으로 Oracle DB에서는 "(+)" 기호를 사용해서 할 수 있다.

```
SELECT * FROM DPT, EMP WHERE EMP.NO (+)= DPT.NO;
```

LEFT OUTER JOIN과 RIGHT OUTER JOIN

LEFT OUTER JOIN은 두 개의 테이블에서 같은 것과, 왼쪽 테이블에만 있는 것을 포함해서 조회한다.

```
SELECT * FROM DPT LEFT OUTER JOIN EMP ON EMP.NO = DPT.NO;
```

RIGHT OUTER JOIN은 두 테이블에서 같은 것과, 오른쪽 테이블에만 있는 것을 포함해서 조회한다.

```
SELECT * FROM DPT RIGHT OUTER JOIN EMP ON EMP.NO = DPT.NO;
```

CROSS JOIN

- CROSS JOIN은 조인 조건 구 없이 2개의 테이블을 하나로 조인한다.
    - 조인구가 없어서 카테시안 곱이 발생한다.
    - 만약 행이 14개 있는 테이블과 행이 4개 있는 테이블을 조인하면 총 56개의 행이 조회된다.
    - FROM 절에 'CROSS JOIN'구를 사용하면 된다.

```
SELECT * FROM EMP CROSS JOIN DPT;
```

> *UNION을 사용한 합집합 구현*

- UNION 연산은 두 테이블을 하나로 만든다.
- 주의사항은 두 테이블의 **칼럼 수, 칼럼의 데이터 형식** 모두가 일치해야 한다.
    - 그렇지 않으면 오류
- UNION 연산은 두 테이블을 하나로 합치면서 중복된 데이터를 제거한다.
    - 그래서 UNION은 정렬(SORT) 과정을 발생시킨다.
- UNION ALL
    - UNION ALL도 두 테이블을 하나로 합친다.
    - 하지만 UNION처럼 중복을 제거하거나 정렬을 유발하지는 않는다.

> *차집합을 만드는 MINUS*

- MINUS 연산은 두 테이블에서 차집합을 조회한다.
    - 먼저 쓴 SELECT문에는 있고 뒤에 쓰는 SELECT문에는 없는 집합을 조회한다

POINT2: 계층형 조회Connect by

> *Connect by*

- 트리Tree 형태의 구조로 쿼리를 실행한다.
- START WITH구는 시작 조건을, CONNECT BY PRIOR는 조인 조건을 의미한다.
- 이렇게 루트 노드부터 하위 노드의 질의를 실행한다.
- 최대 계층의 수를 구하는 함수: **MAX(LEVEL)** #LEVEL은 계층값으로 루트가 1이다.

```
SELECT LEVEL, SAL, EMP_ID, NAME FROM EMP
    START WITH EMP_ID IS NULL
    CONNECT BY PRIOR SAL = EMP_ID;
```

위의 쿼리에서 SAL과 EMP_ID 칼럼 모두 각각이 값을 가진다.

하지만 EMP_ID는 부서별 사원 번호를 가진다. 즉, EMP_ID의 10번은 11번과 12번 부서를 관리한다.

계층형 조회 결과를 명확히 보기 위해 LPAD()를 사용할 수 있다.

- ex. LPAD(' ', 4): 왼쪽 공백 4칸을 화면에 찍어 트리 형태처럼 출력한다.

```
SELECT LEVEL, LPAD(' ', 4 * (LEVEL - 1)) || SAL, EMP_ID, CONNECT_BY_ISLEAF
    FROM EMP
    START WITH EMP_ID IS NULL
    CONNECT BY PRIOR SAL = EMP_ID;
```

> *CONNECT BY 키워드*

키워드	설명
LEVEL	검색 항목의 깊이. 계층 구조에서 가장 상위 레벨은 1이다.
CONNECT_BY_ROOT	계층구조의 최상위 값을 표시
CONNECT_BY_ISLEAF	계층구조의 최하위 값을 표시
SYS_CONNECT_BY_PATH	계층구조의 전체 전개 경로 표시
NOCYCLE	순환구조가 발생지점까지만 전개되는 것
CONNCET_BY_ISCYCLE	순환구조 발생 지점을 표시

POINT3: 서브쿼리Subquery

> *Main query와 Subquery*

- 서브쿼리는 SELECT문 내에 다시 SELECT문을 사용하는 SQL문이다.
- 서브쿼리의 형태
  - FROM구에 SELECT문을 사용하는 인라인 뷰
  - SELECT문에 서브쿼리를 사용하는 스칼라 서브쿼리
  - WHERE구에 SELECT문을 사용하는 서브쿼리

메인쿼리와 서브쿼리

```
SELECT * FROM EMP WHERE DPTNUM = (SELECT DPTNUM FROM DPT WHERE DPTNUM =
10);
```

위 쿼리문에서 괄호로 묶인 부분이 서브쿼리, 그렇지 않은 부분이 메인쿼리이다.

인라인 뷰

```
SELECT * FROM (SELECT ROWNUM NUM, ENAME FROM EMP) a WEHRE NUM < 5;
```

FROM구에 있는 SELECT문을 인라인 뷰라고 한다.

이렇게 FROM구에 SELECT문을 사용하여 가상의 테이블을 만드는 효과를 가진다.

> *단일 행 서브쿼리와 다중 행(멀티 행) 서브쿼리*

서브쿼리는 반환하는 행 수가 한 개인 단일 행 서브쿼리과 여러 개인 다중 행 서브쿼리로 나뉜다.

- 단일 행 서브쿼리
  - 반환하는 행 = 1개
  - 비교 연산자(=, <, <=, >=, <>) 사용
- 다중 행 서브쿼리
  - 반환하는 행이 여러 개
  - 다중 행 비교 연산자인 IN, ANY, ALL, EXISTS 사용

다중 행 연산자	설명
IN(서브쿼리)	메인쿼리의 비교 조건이 서브쿼리의 결과 중 하나만 동일하면 참(OR)
ALL(서브쿼리)	메인쿼리와 서브쿼리의 결과가 모두 동일하면 참 < ALL: 최소값 반환 > ALL: 최대값 반환
ANY(서브쿼리)	메인 쿼리의 비교 조건이 서브쿼리의 결과 중 하나 이상 동일하면 참 < ANY: 하나라도 크면 참 > ANY: 하나라도 작으면 참
EXISTS(서브쿼리)	메인쿼리와 서브쿼리의 결과가 하나라도 존재하면 참

IN

IN은 반환되는 여러 개의 행 중에서 하나만 참이 되어도 참이 되는 연산이다.

```
SELECT NAME, SAL FROM EMP, DPT WHERE EMP.NO = DPT.NO
    AND EMP.NO IN (SELECT EMPNUM FRO EMP WHERE SAL > 2000);
```

SAL(급여)이 2000원 이상인 EMPNUM(사번)을 반환하고 반환된 사번과 메인쿼리에 있는 사번(EMP.NO, DPT.NO)과 비교해서 같은 것을 조회한다.

ALL

ALL은 메인쿼리와 서브쿼리의 결과가 모두 동일하면 참이 되는 연산이다.

```
SELECT * FROM EMP WHERE DPTNUM <= ALL(20, 30);
```

위 쿼리에서는 DPNUM이 20, 30보다 작거나 같을 행을 조회한다.

EXISTS

EXISTS는 서브쿼리로 어떤 데이터의 존재 여부(T, F)를 확인하는 것이다.

```
SELECT NAME, SAL FROM EMP, DPT WHERE EMP.NO = DPT.NO AND EXISTS (SELECT 1
FROM EMP WHERE SAL > 2000);
```

위 쿼리에서는 SAL이 2000보다 큰 사원이 있으면 TRUE가, 그렇지 않으면 FALSE가 반환된다.

> *스칼라 서브쿼리Scala Subquery*

스칼라 서브쿼리는 **반드시 한 행과 한 칼럼만 반환**하는 서브쿼리이다. 만약 여러 행이 반환되면 오류가 발생한다.

> *연관 서브쿼리Correlated Subquery*

연관 서브쿼리는 서브쿼리 내에서 메인쿼리 내의 칼럼을 사용하는 것을 의미한다.

```
...
WHERE A.DPTNO = (SELECT DPTNO FROM DPT b WHERE b.DPTNO = a.DPTNO);
```

POINT4: 그룹 함수Group Function

> *ROLLUP*

- ROLLUP은 GROUP BY의 칼럼(순서 상관 있음)에 대해 부분합Subtotal을 만들어 준다.
  - 예를 들어 소계/합계 등이 계산되면 GROUPING 함수는 0을 반환하고 그렇지 않으면 1을 반환해서 합계 값을 식별할 수 있다.
- GROUPING 함수에서 사용될 컬럼은 반드시 GROUP BY 절에서 명시되어야 한다.

> *GROUPING SETS 함수*

GROUPING SETS 함수는 GROUP BY에 나오는 칼럼의 순서와 관계없이 개별적으로 처리하여, **다양한 소계**를 만들 수 있다.

```
SELECT DEPTNO, JOB, SUM(SAL) FROM EMP GROUP BY GROUPING SETS (DEPTNO,
JOB);
```

위 쿼리는 DEPTNO와 JOB을 각각 그룹으로 하여 합계를 계산한다(직업별 합계와 부서별 합계를 조회).

DEPTNO와 JOB의 순서가 SELECT문에서 바뀌어도 결과는 같다.

> *CUBE 함수*

CUBE 함수는 제시한 칼럼에 대해 결합 가능한 모든 집계(합의 경우의 수)를 계산한다.

```
SELECT DEPTNO, JOB, SUM(SAL) FROM EMP GROUP BY CUBE(DEPTNO, JOB);
```

위 쿼리문의 결과로 전체합계, 직업별 합계, 부서별 합계, 부서별/직업별 합계가 조회된다.

POINT5: 윈도우 함수Window Function

> *윈도우 함수*

윈도우 함수는 행과 행의 관계를 정의하기 위해 제공된다.

위도우 함수를 사용해 순위, 합계, 평균, 행 위치 등을 조작할 수 있다.

윈도우 함수 구조

```
SELECT 윈도우 함수(인수) OVER (PARTITION BY 칼럼명 ORDER BY WINDOWING절)
FROM 테이블명;
```

- 인수: 윈도우 함수에 따라 0~n개의 인수를 설정한다
- PARTITION BY: 전체 집합을 기준에 의해 소그룹으로 나눈다.

- ORDER BY: 어떤 항목에 대해서 정렬한다.
- WINDOWING: 행 기준의 범위를 정한다. ROWS는 물리적 행 수, RNAGE는 논리적인 값에 의한 행 수다.

WINDOWING

구조	설명
ROWS	부분집합인 윈도우 크기를 물리적 단위로 행의 집합을 지정한다.
RANGE	논리적 주소에 의해 행 집합을 지정한다.
BETWEEN ~ AND	윈도우 시작과 끝의 위치를 지정한다.
UNBOUNDED PRECEDING	윈도우 시작 위치가 가장 첫 행임을 의미한다. end point에 사용 불가
UNBOUNDED FOLLOWING	윈도우 마지막 위치가 가장 마지막 행임을 의미한다.
CURRENT ROW	윈도우 시작 위치가 현재 행임을 의미한다.

```
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
```

첫 번째 행부터 마지막 행까지, 즉 모든 행을 뜻한다.

```
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

첫 번째 행부터 현재 행까지를 뜻한다.

```
ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
```

현재 행부터 마지막 행까지를 뜻한다.

> *순위 함수RANK Function*

윈도우 함수는 특정 항목과 파티션에 대해서 순위를 계산할 수 있는 함수를 제공한다.

순위 함수	설명
RANK	특정항목 및 파티션에 대해 순위를 계산한다. **동일한 순위는 동일한 값**이 부여된다.
DENSE_RANK	**동일한 순위를 하나의 건수**로 계산한다.
ROW_NUMBER	**동일한 순위에 대해 고유의 순위**를 부여한다.

```
RANK() OVER (ORDER BY SAL DESC) ALL_RANK,
```

SAL 데이터의 순위를 계산한다. **동일한 순위는 동일하게 조회된다.**

```
RANK() OVER (PARTITION BY JOB ORDER BY SAL DESC) JOB_RANK
```

JOB으로 파티션을 만들고 JOB별 순위를 조회한다.

ALL_RANK	JOB_RANK
1	1
2	1
2	1
4	1
5	2
6	3

```
RANK() OVER (ORDER BY SAL DESC) ALL_RANK,

DENSE_RANK() OVER (ORDER BY SAL DESC) DENSE_RANK
```

SAL 데이터의 순위를 계산한다. **동일한 순위를 하나의 건수로 계산한다.**

ALL_RANK	JOB_RANK
1	1
2	2
2	2
4	3
5	4
6	5

```
RANK() OVER (ORDER BY SAL DESC) ALL RANK,
ROW NUMBER() OVER (ORDER BY SAL DESC) ROW NUM
```

ROW_NUMBER 함수는 동일한 순위에 대해 고유의 순위를 부여한다. (=말그대로 행의 개수)

ALL_RANK	JOB_RANK
1	1
2	2
2	3
4	4
5	5
6	6

> *집계함수RANK Function*

집계함수는 윈도우 함수를 제공한다.

- 집계AGGREGATE 관련 윈도우 함수

집계함수	설명
SUM	파티션 별 합계 계산
AVG	파티션 별 평균 계산
COUNT	파티션 별 행 수 계산
MAX/MIN	파티션 별 최대/최소값 계산

> *행 순서 관련 함수*

행 순서 관련 함수는 상위 행의 값을 하위에 출력하거나 하위 행의 값을 상위 행에 출력할 수 있다.

즉, 특정 위치의 행을 출력할 수 있다.

- 행 순서 관련 윈도우 함수

행 순서	설명
FIRST_VALUE	파티션에서 가장 처음에 나오는 값을 구한다 = MIN()
LAST_VALUE	파티션에서 가장 나중에 나오는 값을 구한다 = MAX()
LAG	이전 행을 가지고 온다.
LEAD	윈도우에서 특정 위치의 행을 가지고 온다. (기본값: 1)

```
FIRST_VALUE(ENAME) OVER (PARTITION BY DPTNO ORDER BY SAL DESC ROWS
UNBOUNDED PRECEDING) AS DPT_A
```

FIRST_VALUE 함수는 파티션에서 조회된 행 중 첫 번째 행의 값을 가지고 온다.

PARTITION BY DPTNO: 부서 파티션을 만든다.

ORDER BY SAL DESC: 급여가 많은 순으로 정렬하므로, 결국 부서 내에서 **급여가 가장 많은 사람**을 의미한다.

```
LAST_VALUE(ENMAE) OVER (PARTITION BY DPTNO ORDER BY SAL DESC ROWS BETWEEN
CURRENT ROW AND UNBOUNDED FOLLOWING) AS DPT_A
```

LAST_VALUE 함수는 파티션에서 조회된 행 중 가장 마지막 행의 값을 가지고 온다.

PARTITION BY DPTNO: 부서 파티션을 만든다.

ORDER BY SAL DESC: 급여가 많은 순으로 정렬하므로, 결국 부서 내에서 **급여가 가장 적은 사람**을 의미한다.

```
LAG(SAL) OVER(ORDER BY SAL DESC) AS PRE_SAL
```

LAG는 이전의 값을 가져온다. PRE_SAL은 이전 행의 SAL 값이다.

```
LEAD(SAL, 2) OVER(ORDER BY SAL DESC) AS PRE_SAL
```

LEAD는 지정된 행의 값을 가져온다(기본값: 1). PRE_SAL은 다음 다음 행의 SAL값이다.

> *비율 관련 함수*

- 비율 관련 윈도우 함수

비율 함수	설명
CUME_DIST	파티션 전체 건수에서 현재 행보다 작거나 같은 건수에 대한 누적 백분율을 조회 누적 분포상에 위치를 0~1 사이의 값을 가짐
PERCENT_RANK	파티션에서 제일 먼저 나온 것을 0으로, 제일 늦게 나온 것을 1로 하여 (값이 아닌) **행의 순서별 백분율**을 조회
NTILE	파티션별 전체 건수를 인자 값으로 N등분한 결과를 조회
RATIO_TO_REPORT	파티션 내 전체 SUM(칼럼)에 대한 행 별 칼럼 값의 백분율을 소수점까지 조회

```
PERCENT_RANK() OVER(PARTITION BY DPTNO ORDER BY SAL DESC) AS PERCENT_SAL
```

PERCENT_RANK()는 파티션에서 등수의 퍼센트를 구한다.

```
NTILE(4) OVER(ORDER BY SAL DESC) AS N_TILE
```

급여가 높은 순으로 4개로 등분한다.

POINT6: 테이블 파티션Table Partition

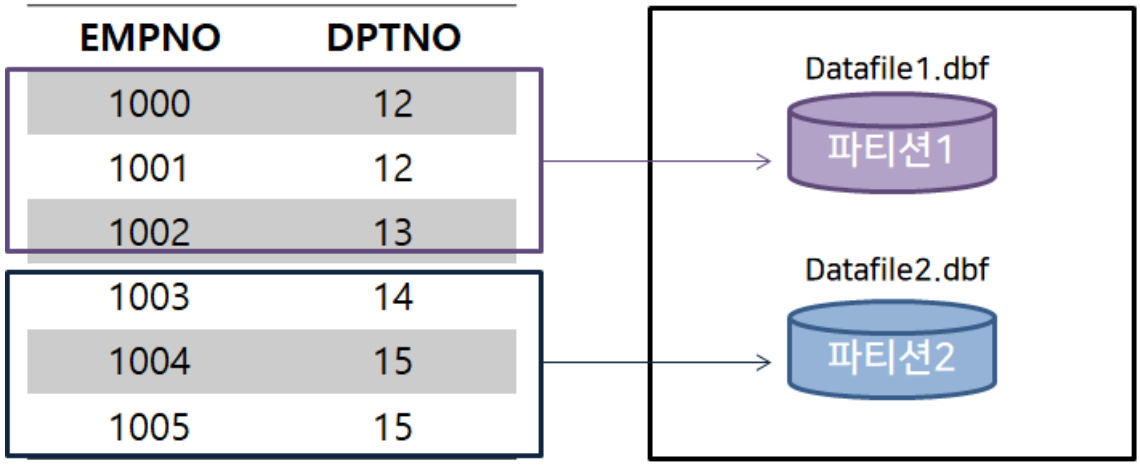
파티션의 기능

- 파티션은 대용량의 테이블을 여러 개의 데이터 파일에 분리해서 저장한다.
  - 데이터가 물리적으로 분리된 데이터 파일에 저장되면 입력/수정/삭제/조회의 성능이 향상된다.
  - 데이터 조회시 데이터의 범위를 줄여서 성능을 향상시킨다.
- 파티션은 각각의 파티션 별로 독립적으로 관리될 수 있다.
  - 즉, 파티션 별 백업/복구가 가능하면 파티션 전용 생성도 가능하다.

Range Parition

Range Partition은 테이블의 칼럼 중 값의 범위를 기준으로 데이터를 나누어 저장한다.

ex.

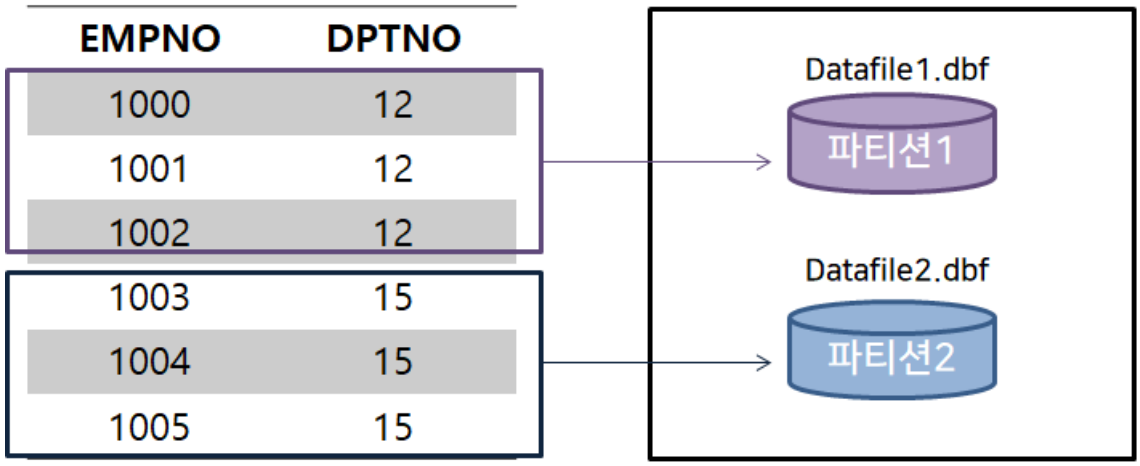


DPTNO가 12~13이면 파티션1, 14~15이면 파티션2에 저장한다.

List Partition

List Partition은 특정 값을 기준으로 테이블을 분할한다.

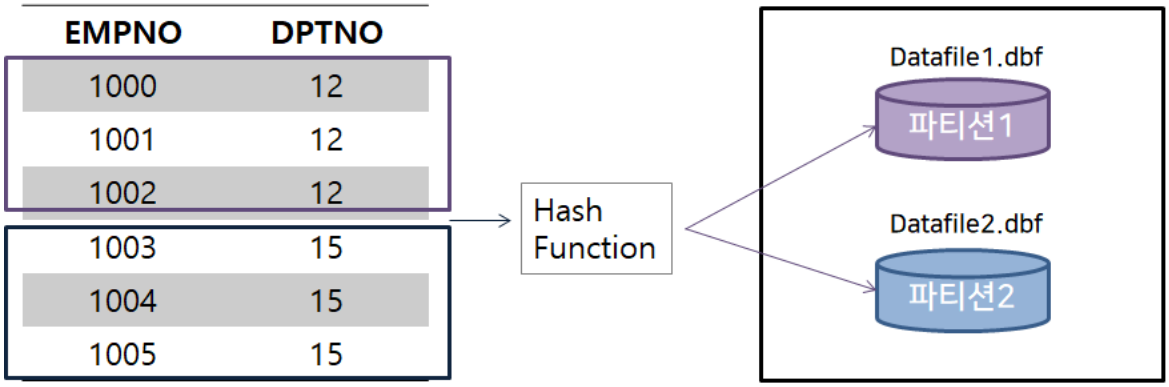
ex.



DPTNO가 12이면 파티션1, 15이면 파티션2에 저장한다.

Hash Partition

Hash Partition은 DBMS가 내부적으로 해시함수를 사용해 테이블을 분할한다. 즉 DBMS가 알아서 한다.



> *Composite Partition*

여러 개의 파티션 기법을 조합해서 사용하는 것이다.

> *파티션*

파티션은 4가지 유형이 있다.

- 파티션 Partition Index

구분	내용
Global Index	여러 개의 파티션에서 한 개의 사용
Local Index	해당 파티션 별 각자의 사용
Prefixed Index	파티션 키와 키가 동일
Non Prefixed Index	파티션 키와 키가 다름

SECTION03 SQL 최적화의 원리

POINT1: 옵티마이저Optimizer와 실행 계획

> *옵티마이저*

- SQL 개발자가 SQL을 작성하여 실행할 때, 옵티마이저는 SQL을 어떻게 실행할 것인지 계획하게 된다.
  - 즉, 옵티마이저는 SQL 실행 계획(Execution Plan)을 수립하고 SQL을 실행하는 DBMS의 SW이다.
- 동일한 결과가 나오는 SQL도 어떻게 실행하느냐에 따라 성능이 달라진다.
  - 따라서 옵티마이저의 실행 계획은 SQL 성능에 아주 중요한 역할을 한다.

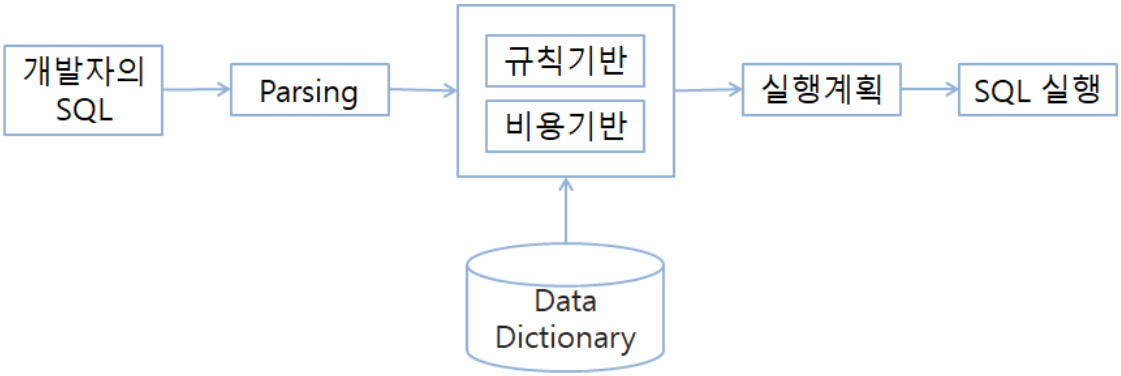
> *옵티마이저의 특징*

- 데이터 딕셔너리에 있는 오브젝트 통계, 시스템 통계 등의 정보를 사용해서 예상되는 비용을 산정한다.
- 여러 개의 실행 계획 중 최저비용을 가진 계획을 선택해 SQL을 실행한다.

POINT2: 옵티마이저의 종류

> *옵티마이저의 실행 방법*

- 개발자가 SQL을 실행하면 파싱Parsing을 실행해서 SQL의 문법 검사 및 구문 분석을 실행한다.
- 구문분석이 완료되면 옵티마이저가 규칙 기반 혹은 비용 기반으로 실행 계획을 ㅅ립한다.
  - 기본적으로 비용 기반 옵티마이저를 사용한다.
  - 비용 기반 옵티마이저는 통계 정보를 활용해서 최적의 실행 계획을 수립한다.
- 실행 계획 수립이 완료되면 최종적으로 SQL을 실행하고 실행이 완료되면 데이터를 인출Fetch한다.
-



> *옵티마이저 엔진*

옵티마이저	설명
Query Transformer	SQL문을 효율적으로 실행하기 위해 옵티마이저가 변환한다 SQL이 변환되어도 그 결과는 동일하다
Estimator	통계 정보를 사용해서 SQL 실행 비용을 계산한다 총 비용은 최적의 실행 계획을 수립하기 위함이다
Plan Generator	SQL을 실행할 실행 계획을 수립한다

- 규칙기반 옵티마이저는 실행 계획을 수립할 때 15개의 우선순위를 기준으로 실행 계획을 수립한다.
  - 최신 Oracle 기본적으로 버전은 규칙기반 옵티마이저보다 비용기반 옵티마이저를 사용한다.

> *규칙 기반 옵티마이저*

- 옵티마이저 엔진의 우선 순위

우선 순위	설명
1	ROWID를 사용한 단일 행인 경우
2	클러스터 조인에 의한 단일 행인 경우
3	유일하거나 PK를 가진 해시 클러스터 키에 의한 단일 행인 경우
4	유일하거나 PK에 의한 단일 행인 경우
5	클러스터 조인인 경우
6	해시 클러스터 조인인 경우
7	클러스터 키인 경우
8	복합 칼럼 인 경우
9	단일 칼럼 인 경우
10	가 구성된 칼럼에서 제한된 범위를 검색하는 경우
11	가 구성된 칼럼에서 무제한 범위를 검색하는 경우
12	정렬-병합(Sort Merge) 조인인 경우
13	가 구성된 칼럼에서 MAX/MIN을 구하는 경우
14	가 구성된 칼럼에서 ORDER BY를 실행하는 경우
15	전체 테이블을 스캔(FULL TABLE SCAN)하는 경우

> *비용 기반 옵티마이저*

- 비용 기반 옵티마이저는 오브젝트 통계 및 시스템 통계를 사용해서 총 비용을 계산한다.
- 총 비용이란 SQL문을 실행하기 위해 예상되는 소요시간이나 자원의 사용량을 뜻한다.
    - 총 비용이 적은 쪽으로 실행 계획을 수립한다.
    - 단, 비용 기반 옵티마이저에서 통계정보가 부적절한 경우 성능 저하가 발생할 수 있다.

POINT3: Index

> *인덱스*

- 인덱스는 데이터를 빠르게 검색할 수 있는 방법을 제공한다.
    - 인덱스는 키로 정렬되어 있기 때문에 원하는 데이터를 빠르게 조회할 수 있다.
- 는 오름차순(ASC), 내림차순(DESC) 탐색이 가능하다.
- 한 테이블에 여러 개의 를 생성할 수 있고 하나의 는 여러 개의 칼럼으로 구성될 수 있다.
- 테이블을 생성할 때 PK는 자동으로 가 만들어진다.
    - 이름은 SYSXXXX이다.
- 인덱스는 Root Block, Brach Block, Leaf Block으로 구성된다.
    - Root Block은 트리에서 가장 상위에 있는 노드다.
    - Branch Block은 다음 단계의 주소를 가진 포인터로 되어 있다.
    - Leaf Block은 키와 ROWID로 구성된다.

- Leaf Block은 Double linked list 형태로 되어 있어 양방향 탐색이 가능하다.
- 키는 정렬된 상태로 저장되어 있다.

> *생성*

```
CREATE INDEX 명 테이블명 ON 테이블명(칼럼명1 차순, 칼럼명2 차순, ...);
```

를 생성할 때는 한 개 이상의 칼럼을 사용해서 생성할 수 있다.

키는 기본적으로 오람차순으로 정렬하고 'DESC' 구를 포함하면 내림차순으로 정렬할 수 있다.

```
CREATE INDEX IND EMP ON EMP (ENAME ASC, SAL DESC);
```

> *스캔*

- 유일 스캔Index Unique SCAN
  - 키 값이 중복되지 않는 경우, 해당 를 사용할 때 발생된다.
  - `SELECT * FROM EMP WHERE EMPNO = 100`
- 범위 스캔Index Range SCAN
  - 의 Leaf Block의 특정 범위를 스캔한 것으로, SELECT문에서 특정 범위를 조회하는 WHERE문을 사용할 경우 발생된다.
  - Like, Between이 대표적인 예시다.
  - 물론 데이터 양이 적은 경우는 자체를 실행하지 않고 TABLE FULL SCAN이 될 수 있다.
  - `SELECT EMPNO FROM EMP WHERE EMPNO >= 100;`
- 전체 스캔Index Full SCAN
  - 에서 검색되는 키가 많은 경우에 Leaf Block의 처음부터 끝까지 전체를 읽어 들인다.
  - `SELECT ENAME, SAL FROM EMP WHERE ENAME Like '%' AND SAL > 0;`

POINT4: 옵티마이저 조인Optimizer Join

> *Nested Loop 조인*

- Nested Loop 조인은 한 테이블에서 데이터를 먼저 찾고, 그 다음 테이블을 조인하는 방식이다.
- 먼저 조회되는 테이블 테이블을 외부 테이블Outer Table이라고 하고, 그 다음 조회되는 테이블은 내부 테이블Inner Table이라고 한다.
  - 이 때 외부 테이블의 크기가 작은 것을 먼저 찾는 것이 중요하다.
  - 그래야 데이터 스캔 범위를 줄일 수 있다.

- Nested Loop 조인은 RANDOM ACCESS가 발생하는데 RANDOM ACCESS가 많이 발생하면 성능 지연이 발생한다.
  - 따라서 RANDOM ACCESS의 양을 줄여야 한다.
  - 이런 특성 때문에 데이터 양이 적을 때 사용한다.

- Nested Loop의 조인 절차

1. 선행 테이블에서 조건을 만족하는 첫 번째 행을 찾는다.
2. 선행 테이블의 조인 키를 가지고 후행 테이블에 조인 키가 존재하는지 찾으러 가서 조인을 시도한다.
3. 후행 테이블의 인덱스에 선행 테이블의조인 키가 존재하는지 확인한다.
4. 인덱스에서 추출한 레코드 식별자를 이용해 후행 테이블을 액세스한다.

```
SELECT /*+ ordered use_nl(b) */ *
FROM EMP a, DPT b, WHERE a.NO = b.NO AND a.NO = 10;
```

위의 쿼리에서 사용한 힌트처럼 use_nl 을 사용해서 의도적으로 Nested Loop 조인을 실행할 수 있다.

> *Sort Merge 조인*

- Sort Merge 조인은 두 개의 테이블을 SORT_AREA라는 메모리 공간에 모두 로딩하고 정렬한다.
- 두 테이블에 대해 정렬이 완료되면 이를 병합한다.
    - Sort Merge 조인은 정렬이 발생하므로 데이터 양이 많으면 성능이 떨어진다.
    - 정렬 데이터 양이 너무 많으면 정렬은 임시 영역에서 수행된다.
        - 임시 영역은 디스크에 있기 때문에 성능이 급격히 떨어진다.
- PK와 FK 관계에서 FK 인덱스가 없을 때 Sort Merge 방식으로 옵티마이저가 동작한다.

```
SELECT /*+ ordered use_merge(b) */ *
FROM EMP a, DPT b WHERE a.NO = b.NO AND a.NO = 10;
```

위의 쿼리에서 사용한 힌트처럼 use_merge 을 사용해서 의도적으로 Sort Merge 조인을 실행할 수 있다.

> *Hash 조인*

- Hash 조인은 두 테이블 중에서 작은 테이블을 HASH 메모리에 로딩하고 두 테이블의 조인 키를 사용해서 해시 테이블을 생성한다.
- Hash 조인은 **Equal Join만** 사용 가능하다.
- Hash 조인은 해시 함수를 사용해서 주소를 계산하고 해당 주소를 사용해서 테이블을 조인하므로 CPU 연산을 많이 한다.
    - 그래서 Hash 조인 시에는 선행 테이블이 충분히 메모리에 로딩되는 크기여야 한다.

```
SELECT /*+ ordered use_hash(b) */ *
FROM EMP a, DPT b WHERE a.NO = b.NO AND a.NO = 10;
```

위의 쿼리에서 사용한 힌트처럼 use_hash 을 사용해서 의도적으로 Hash 조인을 실행할 수 있다.

"SQLD PART2" PDF - matiii

: https://hec-ker.tistory.com/