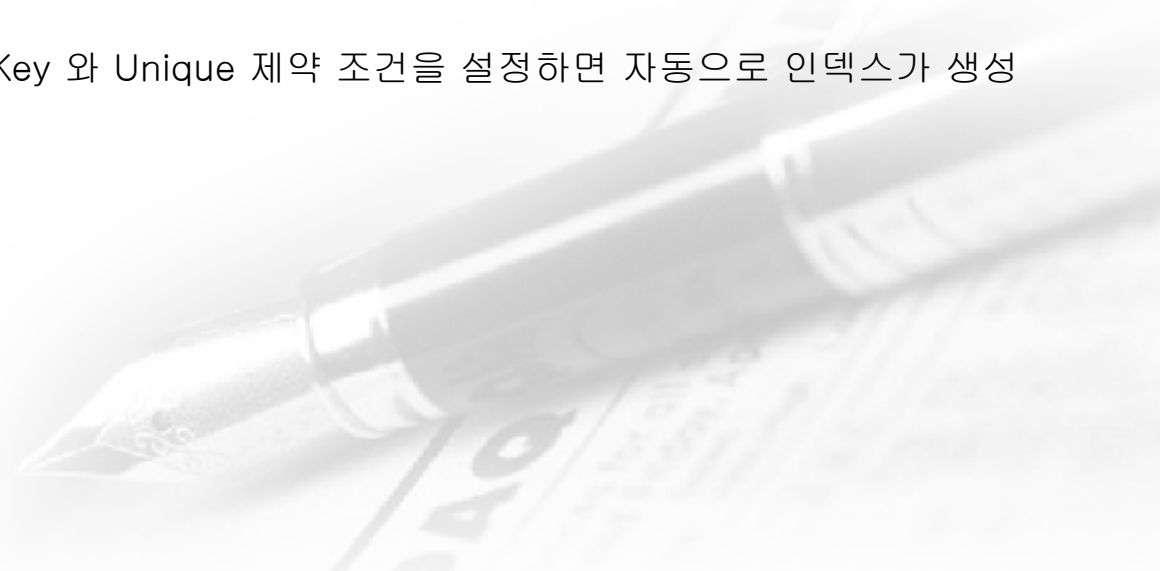


Optimizing

INDEX

❖ 개요

- ✓ 인덱스는 원하는 데이터를 쉽게 찾을 수 있도록 돕는 책의 찾아보기(색인)과 유사한 개념으로 키와 참조를 쌍으로 저장하는 구조
- ✓ 인덱스는 테이블을 기반으로 선택적으로 생성할 수 있는 구조
- ✓ 인덱스의 기본적인 목적은 검색 성능의 최적화
- ✓ 검색 조건을 만족하는 데이터를 인덱스를 통해 효과적으로 찾을 수 있도록 해 줌
- ✓ 인덱스가 설정된 경우 DML 작업은 테이블과 인덱스를 함께 변경하므로 느려지는 단점이 있음
- ✓ 인덱스는 오름차순(ASCENDING) 및 내림차순(DSCENDING) 탐색이 가능
- ✓ 하나의 테이블에 여러 개의 인덱스를 생성할 수 있고 하나의 인덱스는 여러 개의 컬럼으로 구성할 수 있음
- ✓ 테이블을 생성할 때 Primary Key 와 Unique 제약 조건을 설정하면 자동으로 인덱스가 생성됨



INDEX

❖ 개요

✓ 인덱스의 장점

- 검색 속도가 빨라짐
- 시스템에 걸리는 부하를 줄여서 시스템 전체 성능을 향상시킴

✓ 인덱스의 단점

- 인덱스의 내부 구조는 B 트리 형식으로 구성하는데 컬럼에 인덱스를 설정하면 이를 위한 B 트리도 생성되어야 하기 때문에 인덱스를 생성하기 위한 시간도 필요하고 인덱스를 위한 추가적인 공간이 필요
- DML 작업이 무거워짐
 - ◆ 인덱스가 생성된 후에 새로운 행을 추가하거나 삭제할 경우 인덱스로 사용된 컬럼 값도 함께 변경되는 경우가 발생
 - ◆ 인덱스로 사용된 컬럼 값이 변경되는 이를 위한 내부 구조(B 트리) 역시 함께 수정
 - ◆ 인덱스의 수정 작업은 서버에 의해 자동으로 일어나는데 그렇기 때문에 인덱스가 없는 경우 보다 인덱스가 있는 경우에 작업이 무거워짐

INDEX

❖ 종류

✓ 클러스터 기반 인덱스

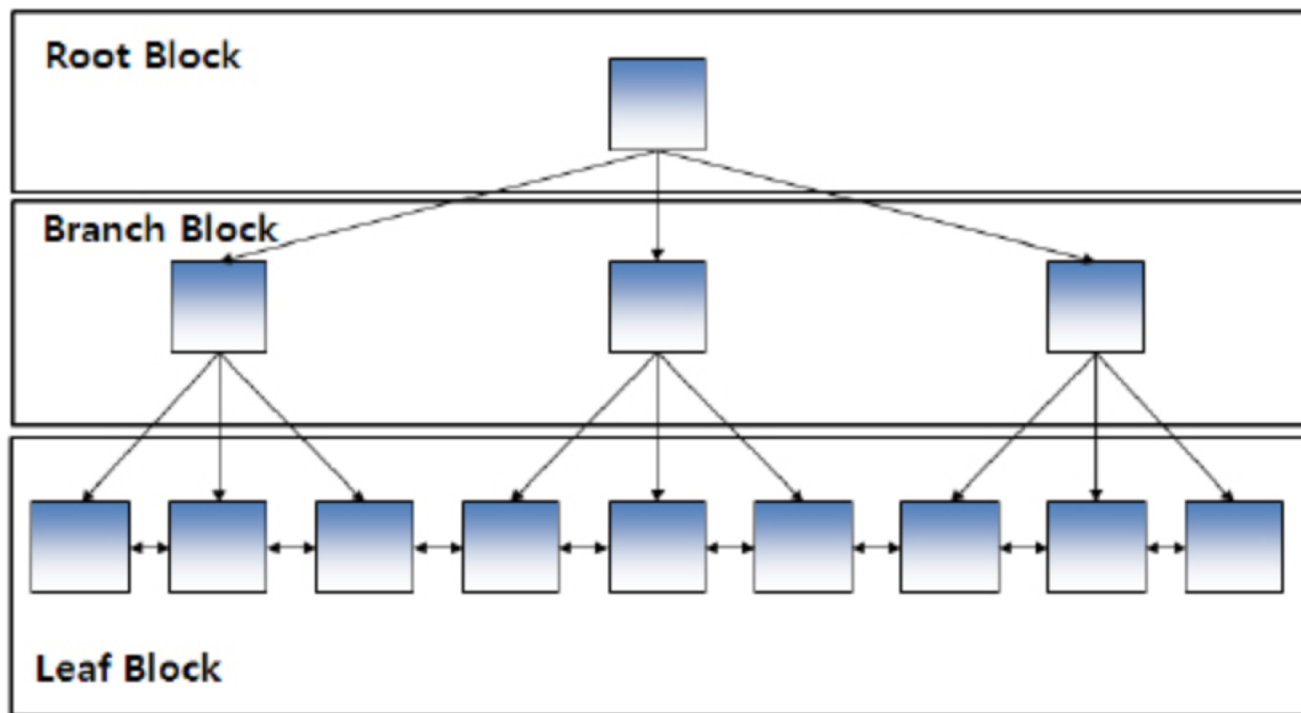
- 모든 로우(데이터)는 인덱스 키 컬럼 순으로 물리적으로 정렬되어 저장
- 테이블 로우는 물리적으로 한 가지 순서로만 정렬될 수 있으므로 클러스터형 인덱스는 테이블당 한 개만 생성 할 수 있음
- 기본키를 만들면 기본키가 클러스터 기반 인덱스가 설정되고 기본키가 없는 경우는 Unique 에 NOT NULL 제약조건을 지닌 컬럼에 클러스터 기반 인덱스가 설정됨

탐색 키(학년)	포인터(주소 값)	학번	이름	학과	학년
1		2007150003	황도경	기계공학과	1
2		2009150001	강지영	전자공학과	1
3		2009150002	김대기	컴퓨터공학과	1
4		2009150003	한우경	컴퓨터공학과	1
		2005150001	강성호	컴퓨터공학과	2
		2006150003	하정현	기계공학과	2
		2008150001	김승현	컴퓨터공학과	2
		2008150002	유진이	컴퓨터공학과	2
		2008150003	정유나	기계공학과	2
		2007150001	곽나리	컴퓨터공학과	3
		2007150002	이민구	컴퓨터공학과	3
		2005150002	김슬기	전자공학과	4
		2005150003	박신애	컴퓨터공학과	4
		2006150001	김민우	전자공학과	4
		2006150002	나기환	컴퓨터공학과	4

INDEX

❖ 종류

- ✓ 트리 기반 인덱스



INDEX

❖ 종류

✓ 트리 기반 인덱스

- DBMS에서 가장 일반적인 인덱스
- B 트리 인덱스는 브랜치 블록과 리프 블록으로 구성
- 브랜치 블록 중 가장 상위에 있는 블록을 루트 블록이라고 함
- 브랜치 블록은 분기를 목적으로 하는 블록
- 리프 블록은 트리 가장 아래 단계에 존재
- 리프 블록은 인덱스를 구성하는 컬럼의 데이터와 해당 데이터를 가지고 있는 행의 위치를 가리키는 레코드 식별자로 구성
- 인덱스 데이터는 인덱스를 구성하는 컬럼의 값으로 정렬
- 인덱스 데이터의 값이 동일하면 레코드 식별자 순서로 저장
- 리프 블록은 양방향 링크(Double Linked List)를 가지고 있음(이것을 통하여 오름차순과 내림차순 검색을 쉽게 할 수 있음)
- B 트리 인덱스는 = , BETWEEN, > 등과 같은 연산자로 검색 구조에 적합한 구조

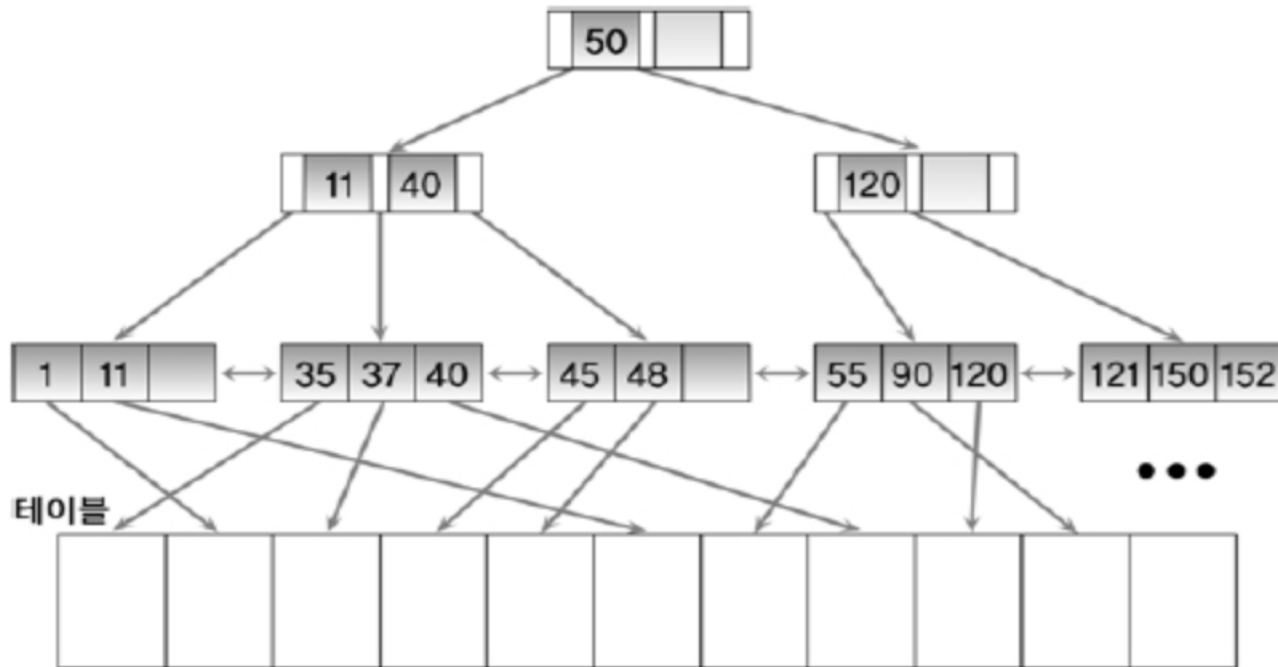
INDEX

❖ 종류

✓ 트리 기반 인덱스

- 데이터를 찾아가는 과정

브랜치 블록이 3개의 포인터로 구성된 B트리 인덱스에서 원하는 값을 찾는과정



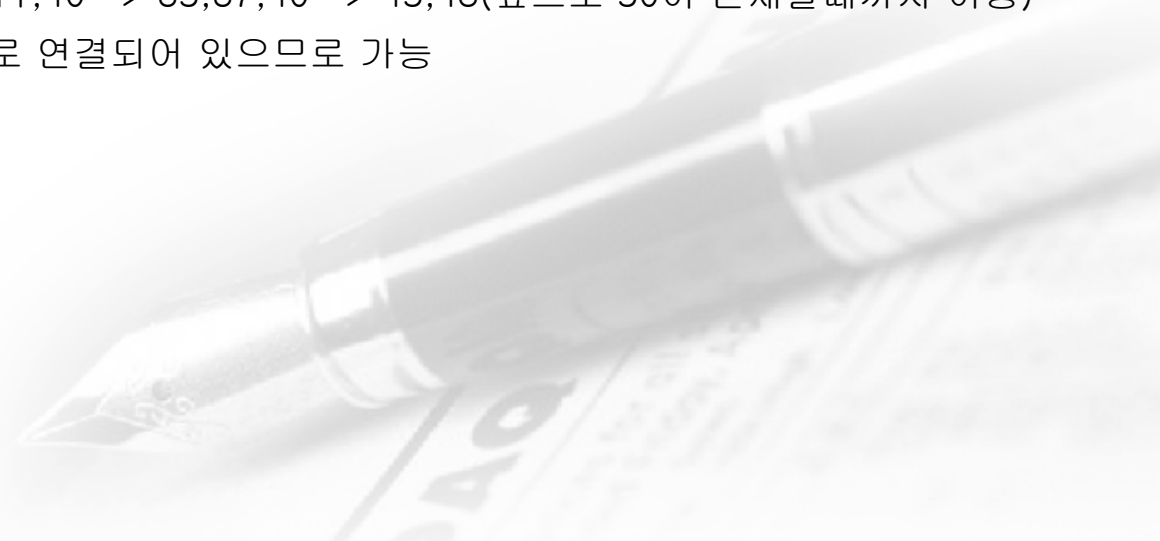
INDEX

❖ 종류

✓ 트리 기반 인덱스

● 데이터를 찾아가는 과정

- ◆ 1단계: 브랜치 블록의 가장 왼쪽 값이 찾고자 하는 값보다 작거나 같으면 왼쪽 포인터로 이동
- ◆ 2단계: 찾고자 하는 값이 브랜치 블록의 값 사이에 존재하면 가운데 포인터로 이동
- ◆ 3단계: 오른쪽에 있는 값보다 크면 오른쪽 포인터로 이동
- ◆ 37을 찾는 과정(=37)
 - 루트 50 -> 11,40 -> 35,37,40
- ◆ 37~50 사이 값을 찾는 과정(BETWEEN 37 AND 50)
 - 루트 50 -> 11,40 -> 35,37,40 -> 45,48(옆으로 50이 존재할때까지 이동)
 - 양방향 링크로 연결되어 있으므로 가능



INDEX

❖ 종류

✓ 트리 기반 인덱스

- 여러 개의 컬럼으로 인덱스를 생성할 때 동일 컬럼으로 구성된 인덱스를 추가로 생성할 수 없음
- 인덱스 구성 컬럼은 동일하지만 컬럼의 순서가 다르면 서로 다른 인덱스로 생성 할 수 있음(JOB+SAL , SAL+JOB)
- 인덱스 컬럼의 순서는 성능에 중요한 영향을 미치는 요소



INDEX

❖ 자동으로 생성되는 인덱스

- ✓ 쿼리를 빠르게 수행하기 위한 용도로 사용되는 인덱스는 기본 키나 유일 키와 같은 제약 조건을 지정하면 따로 생성하지 않더라도 자동으로 생성
- ✓ 기본 키나 유일 키는 데이터 무결성을 확인하기 위해서 수시로 데이터를 검색하기 때문에 빠른 조회를 목적으로 내부적으로 해당 컬럼에 인덱스를 자동으로 생성하는 것
- ✓ SHOW INDEX FROM 테이블이름 으로 인덱스 조회 가능

탐색 키(학번)	포인터(주소 값)	학번	이름	학과	학년
2005150001		2005150001	강성호	컴퓨터공학과	2
2006150001		2005150002	김슬기	전자공학과	4
2007150001		2005150003	박신애	컴퓨터공학과	4
2008150001		2006150001	김민우	전자공학과	4
2009150001		2006150002	나기환	컴퓨터공학과	4
		2006150003	하정현	기계공학과	2
		2007150001	곽나리	컴퓨터공학과	3
		2007150002	이민구	컴퓨터공학과	3
		2007150003	황도경	기계공학과	1
		2008150001	김승현	컴퓨터공학과	2
		2008150002	유진이	컴퓨터공학과	2
		2008150003	정유나	기계공학과	2
		2009150001	강지영	전자공학과	1
		2009150002	김대기	컴퓨터공학과	1
		2009150003	한우경	컴퓨터공학과	1

INDEX

❖ 자동으로 생성되는 인덱스

- ✓ EMP 테이블의 인덱스 확인

SHOW INDEX FROM EMP;

	ABC Table	123 Non_unique	ABC Key_name	123 Seq_in_index	ABC Column_name
1	EMP	0	PRIMARY	1	EMPNO
2	EMP	1	FK_DEPTNO	1	DEPTNO



INDEX

❖ 자동으로 생성되는 인덱스

- ✓ 새로운 테이블을 만들어서 확인

```
CREATE TABLE IndexTable (  
  a INT PRIMARY KEY,  
  b INT UNIQUE,  
  c INT UNIQUE,  
  d INT  
);
```

```
SHOW INDEX FROM IndexTable;
```

	ABC Table	123 Non_unique	ABC Key_name	123 Seq_in_index	ABC Column_name
1	IndexTable	0	PRIMARY	1	a
2	IndexTable	0	b	1	b
3	IndexTable	0	c	1	c

INDEX

❖ 인덱스 직접 생성

- ✓ 제약 조건에 의해 자동으로 생성되는 인덱스 외에 CREATE INDEX 명령어로 직접 인덱스를 생성할 수 있는데 기본 형식은 아래와 같음

```
CREATE [OR REPLACE] [UNIQUE] INDEX [IF NOT EXISTS] index_name  
[index_type]  
ON table_name (column_name)  
[WAIT n | NOWAIT]  
[index_option]  
[algorithm_option | lock_option];
```

- CREATE INDEX 다음에 인덱스 객체 이름을 지정
- 어떤 테이블의 어떤 컬럼에 인덱스를 설정할 것인지를 결정하기 위해서 ON 절 다음에 테이블 이름과 컬럼 이름을 기술



INDEX

❖ 인덱스 직접 생성

- ✓ 제약 조건에 의해 자동으로 생성되는 인덱스 외에 CREATE INDEX 명령어로 직접 인덱스를 생성할 수 있는데 기본 형식은 아래와 같음
 - UNIQUE: 고유한 인덱스를 만들 것인지 여부
 - index_type: 인덱스의 자료구조로 BTREE 와 HASH, RTREE 가 있음
 - WAIT n (초 단위로 설정) 또는 NOWAIT 를 사용하여 명령문에서 잠금 대기 시간 종료를 명시적으로 설정할 수 있는데 잠금을 확보 할 수 없으면 명령문이 즉시 실패하고 WAIT 0 은 NOWAIT 와 같음
 - ALGORITHM 모드의 종류: ALTER 명령을 사용할 때의 옵션
 - ◆ ALGORITHM=COPY는 대상 테이블의 원본은 놔두고 변경될 테이블을 새로 생성한 후 원본으로부터 데이터를 로우 단위로 복제하는 방식으로 모든 복제 작업과 인덱스 생성 작업이 완료되면 원본을 변경된 테이블로 바꾸는 방식으로 복제본을 만들어야 하기 때문에 시스템 자원을 많이 소모한다.
 - ◆ ALGORITHM=INPLACE은 대상 테이블의 복제 과정 없이 메타 데이터 변경 만으로 빠르게 변경사항을 반영하는 방식으로 속도가 가장 빠르며 시스템 자원도 거의 소모하지 않음
 - LOCK=NONE|SHARED|EXCLUSIVE 형태로 설정

INDEX

❖ 인덱스 직접 생성

✓ 인덱스 제거

DROP INDEX 인덱스이름 ON 테이블이름

✓ 인덱스 생성 및 제거

- 테이블 EMP의 컬럼 중에서 이름(ENAME)에 대해서 인덱스를 생성

CREATE INDEX IDX_EMP_ENAME

ON EMP(ENAME);

- EMP 테이블의 IDX_EMP_ENAME 인덱스를 제거

DROP INDEX IDX_EMP_ENAME

ON EMP;



INDEX

❖ 인덱스 사용 여부 판단

인덱스를 사용해야 하는 경우	인덱스를 사용하지 말아야 하는 경우
테이블에 행의 수가 많을 때	테이블에 행의 수가 적을 때 나 중복되는 데이터가 많은 열
WHERE 문에 해당 컬럼이 많이 사용될 때	WHERE 문에 해당 컬럼이 자주 사용되지 않을 때
검색 결과가 전체 데이터의 2%~4% 정도 일 때	검색 결과가 전체 데이터의 10%~15% 이상 일 때
JOIN에 자주 사용되는 컬럼이나 NULL을 포함하는 컬럼이 많은 경우	테이블에 DML 작업이 많은 경우 즉, 입력 수정 삭제 등이 자주 일어날 때

INDEX

❖ 인덱스 형식

✓ 고유 인덱스 와 비 고유 인덱스

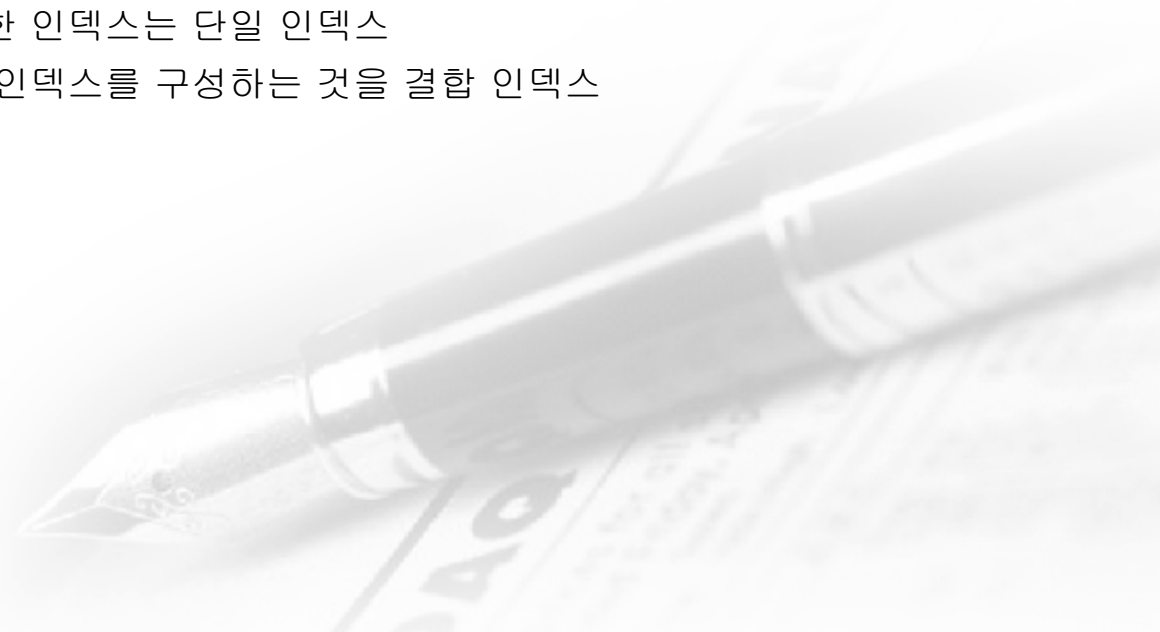
- 고유 인덱스(유일 인덱스라고도 부름)는 기본키나 유일키처럼 유일한 값을 갖는 컬럼에 대해서 생성하는 인덱스
- 반면 비고유 인덱스는 중복된 데이터를 갖는 컬럼에 대해서 인덱스를 생성하는 경우
- 고유 인덱스를 설정하려면 UNIQUE 옵션을 추가해서 인덱스를 생성

```
CREATE UNIQUE INDEX index_name
```

```
ON table_name (column_name);
```

✓ 단일 인덱스와 결합 인덱스

- 한 개의 컬럼으로 구성된 인덱스는 단일 인덱스
- 두 개 이상의 컬럼으로 인덱스를 구성하는 것을 결합 인덱스



INDEX

❖ 인덱스 형식

- ✓ 부서 번호와 부서명을 결합하여 인덱스를 설정할 수 있는데 이것이 결합 인덱스

```
CREATE INDEX IDX_DEPT_COM  
ON DEPT(DEPTNO, DNAME);
```

- ✓ 인덱스 확인

```
SHOW INDEX FROM DEPT;
```

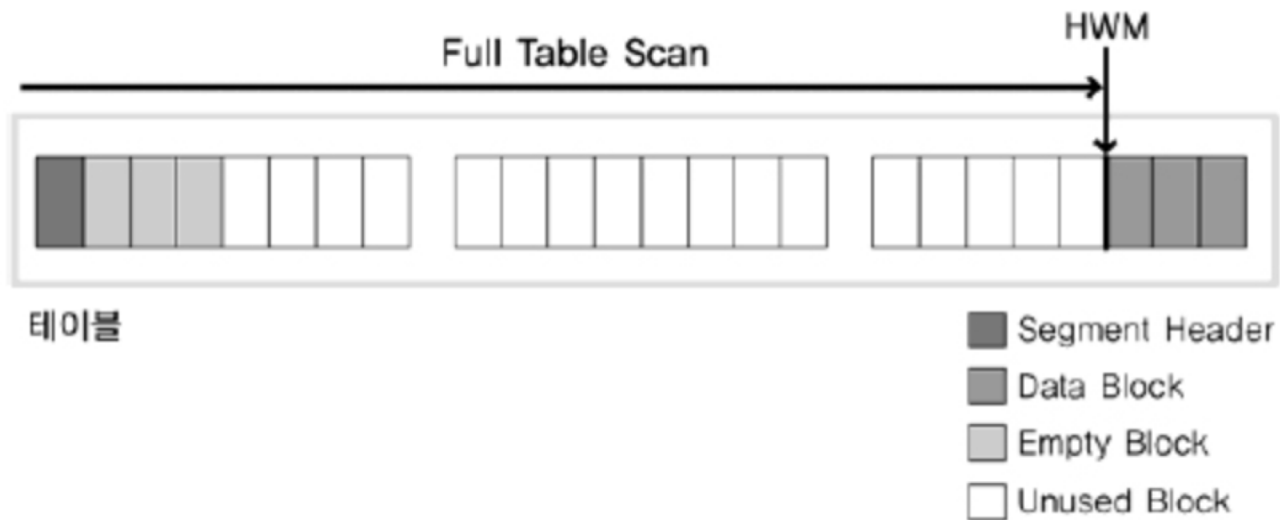


INDEX

❖ 전체 테이블 스캔과 인덱스 스캔

✓ 전체 테이블 스캔

- 테이블에 존재하는 모든 데이터를 읽어 가면서 조건이 맞으면 결과로 추출하고 조건에 맞지 않으면 버리는 방식

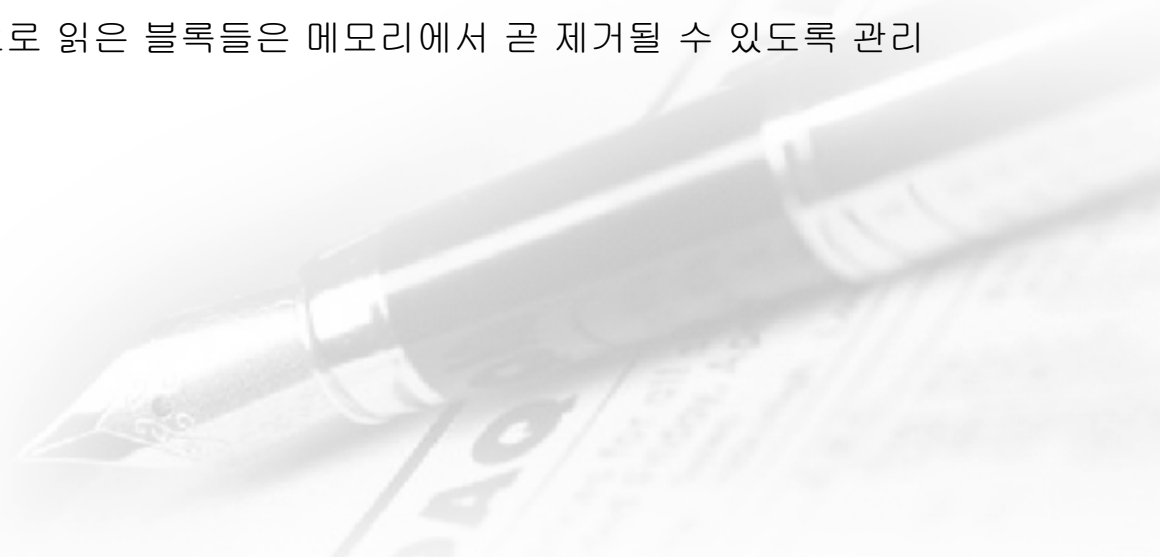


INDEX

❖ 전체 테이블 스캔과 인덱스 스캔

✓ 전체 테이블 스캔

- 검색 조건에 맞는 데이터를 찾기 위해 테이블의 고수위 마크(HWM, High Water Mark) 아래의 모든 블록을 읽음
- 고수위 마크는 테이블에 데이터가 쓰여졌던 블록 상위 최상위 위치(현재는 지워져서 데이터가 존재하지 않을 수도 있음)를 의미
- 전체 테이블 스캔 방식으로 데이터를 검색할 때 고수위 마크까지의 블록 내 모든 데이터를 읽어야 하기 때문에 모든 결과를 찾을 때 까지 시간이 오래 걸릴 수 있음
- 전체 테이블 스캔 방식은 테이블에 존재하는 모든 블록의 데이터를 읽음
- Full Table Scan 연산이었기 때문에 모든 블록을 읽은 것
- 이렇게 읽은 블록은 재 사용성이 떨어짐
- 전체 테이블 스캔 방식으로 읽은 블록들은 메모리에서 곧 제거될 수 있도록 관리



INDEX

❖ 전체 테이블 스캔과 인덱스 스캔

✓ 전체 테이블 스캔

● 전체 테이블 스캔이 발생하는 경우

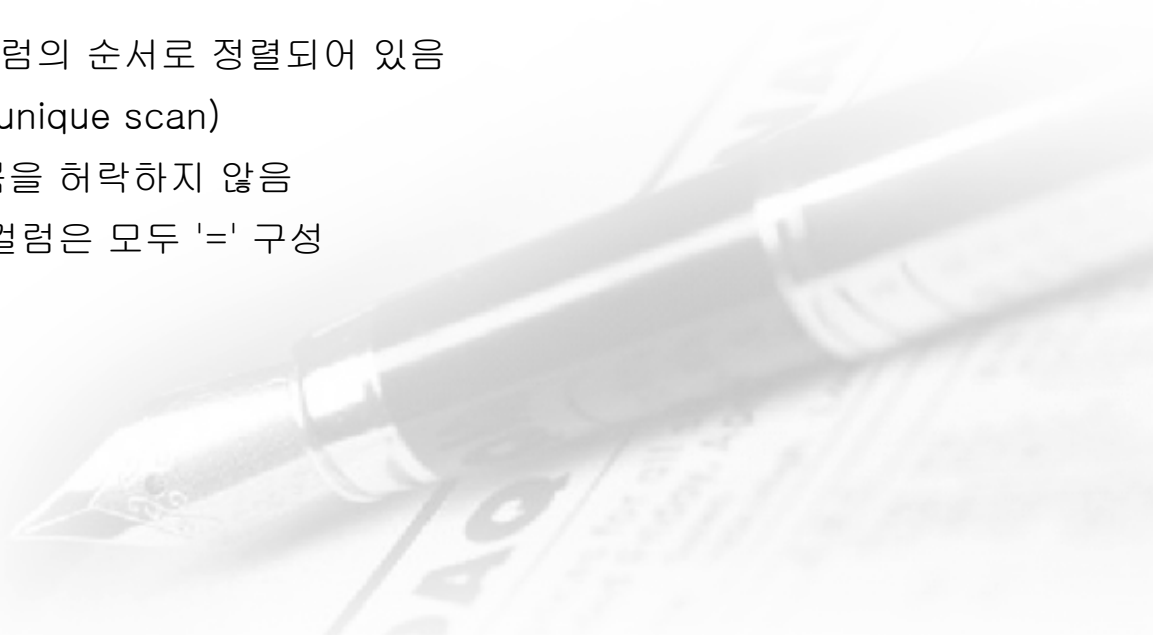
- ◆ SQL문에 조건이 존재하지 않은 경우: SQL문에 조건이 존재하지 않는다는 것은 테이블에 존재하는 모든 데이터가 답이 된다는 것 이므로 모든 블록을 읽으면서 결과로서 반환
- ◆ SQL문의 주어진 조건에 사용 가능한 인덱스가 존재하지 않는 경우: 사용 가능한 인덱스가 존재하지 않는다면 데이터를 액세스할 수 있는 방법은 테이블의 모든 데이터를 읽으면서 주어진 조건을 만족하는지를 검사하는 방법 뿐이며 또한 주어진 조건에 사용 가능한 인덱스는 존재하나 함수를 사용하여 인덱스 컬럼을 변형한 경우에는 인덱스를 사용할 수 없음
- ◆ 옵티마이저의 취사 선택: 조건을 만족하는 데이터가 많은 경우 결과를 추출하기 위해서 테이블의 대부분의 블록을 액세스해야 한다고 옵티마이저가 판단하면 조건에 사용 가능한 인덱스가 존재해도 전체 테이블 스캔 방식으로 읽을 수 있음
- ◆ 그 밖의 경우: 병렬 처리 방식으로 처리하는 경우 또는 전체 테이블 스캔 방식의 힌트를 사용한 경우에 전체 테이블 스캔 방식으로 읽을 수 있음

INDEX

❖ 전체 테이블 스캔과 인덱스 스캔

✓ 인덱스 스캔

- 인덱스 스캔은 인덱스를 구성하는 컬럼의 값을 기반으로 데이터를 추출하는 방식
- 인덱스 리프 블록은 인덱스 구성하는 컬럼과 레코드 식별자로 구성되어 있음
- 인덱스의 리프 블록을 읽으면 인덱스 구성 컬럼의 값 과 테이블의 레코드 식별자를 알 수 있음
- 인덱스에 존재하지 않는 컬럼의 값이 필요한 경우 현재 읽은 레코드 식별자를 이용하여 테이블을 액세스 해야함
- SQL문에 필요로 하는 모든 컬럼이 인덱스 구성 컬럼에 포함된 경우 테이블에 대한 액세스는 발생하지 않음
- 인덱스는 인덱스 구성 컬럼의 순서로 정렬되어 있음
- 인덱스 유일 스캔(index unique scan)
 - ◆ 유일 인덱스는 중복을 허락하지 않음
 - ◆ 유일 인덱스 구성 컬럼은 모두 '=' 구성



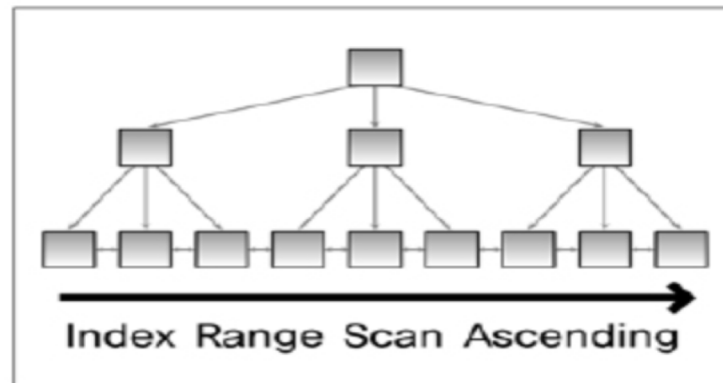
INDEX

❖ 전체 테이블 스캔과 인덱스 스캔

✓ 인덱스 스캔

● 인덱스 범위 스캔(index range scan)

- ◆ 인덱스를 이용하여 한 건 이상 데이터를 추출하는 방식
- ◆ 컬럼 모두에 대해 '='로 값이 주어지지 않은 경우와 비유일 인덱스(non-unique index)를 이용한 스캔방식으로 데이터를 액세스 함
- ◆ INDEX Range SCAN은 SELECT문에서 특정 범위를 조회하는 WHERE 구문을 사용할 경우 발생
- ◆ Like, Between이 그 대표적인 예로 데이터 양이 적은 경우는 인덱스를 실행하지 않고 TABLE FULL SCAN이 될 수 있음
- ◆ 인덱스의 Leaf Block의 특정 범위를 스캔한 것
- ◆ 하단 그림과 같은 방식으로 읽음



INDEX

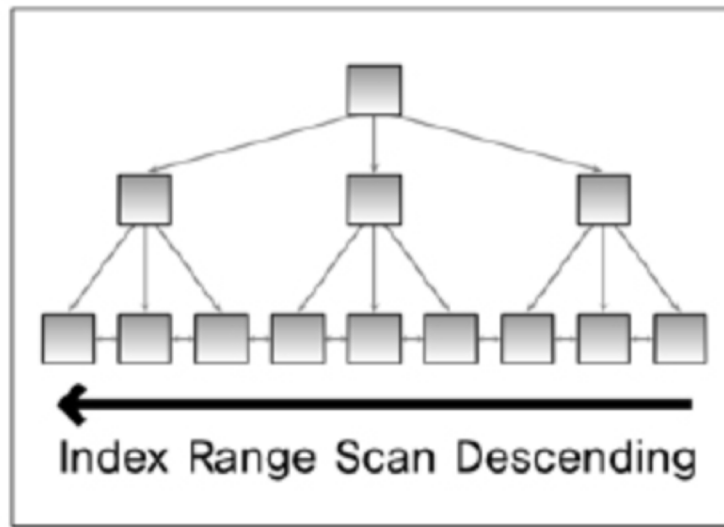
❖ 전체 테이블 스캔과 인덱스 스캔

✓ 인덱스 스캔

● 인덱스 범위 스캔(index range scan)

◆ 인덱스 역순 범위 스캔(index range scan descending)

- ◆ 그림과 같이 인덱스 리프 블록의 양방향 양방향 링크를 이용하여 내림차순으로 데이터를 읽는 방식
- ◆ 이 방식을 이용하여 최대값을 쉽게 찾을 수 있음



INDEX

❖ 전체 테이블 스캔과 인덱스 스캔

✓ 비교

- 데이터를 액세스하는 방법은 인덱스를 이용해서 읽는 인덱스 스캔 방식과 테이블을 전체 데이터를 모두 읽으면서 데이터를 추출하는 전체 테이블을 스캔하는 방식이 있음
- 인덱스 스캔은 사용가능 한 적절한 인덱스가 존재할 때만 이용할 수 있는 방식이고 전체 테이블 스캔 방식은 인덱스의 존재 유무와 상관없이 항상 사용가능 한 스캔 방식
- 전체 테이블 스캔이나 인덱스 스캔이나 차이는 대용량 데이터 중 극히 일부의 데이터를 찾을 때는 인덱스 스캔 방식을 이용하여 몇번의 I/O만으로 데이터를 찾고 전체 테이블 스캔은 모든 데이터를 읽으면서 원하는 데이터를 찾아야 하므로 비효율적인 검색을 수행
- 한번에 여러 블록 단위로 읽는 경우라면 전체 테이블 스캔 방식이 유리할 수 있음



Optimizer Join

❖ Nested Loop Join

- ✓ NL 조인은 프로그램에서 사용하는 중첩된 반복문과 유사한 방식으로 조인을 수행
- ✓ 반복문의 외부에 있는 테이블을 선행 테이블 또는 외부 테이블(OUTER TABLE)이라고 하고 반복문의 내부에 있는 테이블을 후행 테이블 또는 내부 테이블(INNER TABLE)이라고 함
- ✓ NL JOIN의 수행 방식
 - 선행 테이블에서 조건을 만족하는 첫 번째 행을 찾음: 이때 선행 테이블에 주어진 조건을 만족하지 않는 경우 해당 데이터는 필터링
 - 선행 테이블의 조인 키를 가지고 후행 테이블에 조인 키가 존재하는지 찾기로 감: 조인 시도
 - 후행 테이블의 인덱스에 선행 테이블의 조인 키가 존재하는지 확인: 선행 테이블의 조인 값이 후행 테이블에 존재하지 않으면 선행 테이블 데이터는 필터링 됨(조인 작업x)
 - 인덱스에서 추출한 레코드 식별자를 이용하여 테이블을 액세스: 인덱스 스캔을 통한 테이블 액세스
 - 반복 수행



Optimizer Join

❖ Nested Loop Join

- ✓ 추출 버퍼는 SQL문의 실행 결과를 보관하는 버퍼로 일정 크기를 설정하여 결과가 모두 차거나 더 이상 결과가 없어서 추출 버퍼를 채울 것이 없으면 결과를 사용자에게 반환
- ✓ 추출 버퍼는 운반 단위, Array Size, prefetch Size라고도 함
- ✓ NL Join 기법은 성공하면 바로 결과를 사용자에게 보여주므로 온라인 프로그램에 적당한 조인 기법
- ✓ 결과 행의 수가 적은 테이블을 조인 순서 상 선행 테이블로 선택하는 것이 전체 조인 작업의 양을 줄일 수 있음
- ✓ 랜덤 방식으로 데이터 액세스하기 때문에 처리 범위가 좁은 것의 조건으로 선택하는 것이 유리
- ✓ use_nl(선행 테이블 이름) 힌트를 이용해서 수행

```
SELECT /*+ use_nl(d) */ *  
FROM EMP e, DEPT d  
WHERE e.DEPTNO = d.DEPTNO AND e.DEPTNO = 10;
```



Optimizer Join

❖ Sort Merge Join

- ✓ Sort Merge Join은 조인 컬럼을 기준으로 데이터를 정렬하여 조인을 수행
- ✓ Sort Merge Join은 주로 스캔 방식으로 데이터를 읽음
- ✓ Sort Merge Join은 랜덤 액세스로 부담이 되는 넓은 범위의 데이터를 처리 할 때 이용
- ✓ Sort Merge Join은 정렬할 데이터가 많아 메모리에 모든 정렬 작업을 수행하기 어려운 경우에는 임시 영역을 사용하므로 성능이 떨어질 수 있음
- ✓ Sort Merge Join은 Hash Join 과는 달리 동등 조인 뿐만 아니라 비 동등 조인에 대해서 조인 작업이 가능
- ✓ Sort Merge Join 동작
 - 선행 테이블에서 주어진 조건을 만족하는 행을 찾음
 - 선행 테이블의 조인 키를 기준으로 정렬 작업을 수행(1 ~ 2번 작업을 선행 테이블의 조건을 만족하는 모든 행에 대해 반복 수행)
 - 후행 테이블에서 주어진 조건을 만족하는 행을 찾음
 - 후행 테이블의 조인 키를 기준으로 정렬 작업을 수행(3 ~ 4번 작업을 후행 테이블의 조건을 만족하는 모든 행에 대해 반복 수행)
 - 정렬된 결과를 이용하여 조인을 수행하며 조인에 성공하면 추출 버퍼에 넣음
- ✓ Sort Merge Join은 조인 컬럼의 인덱스가 존재하지 않을 경우에도 사용할 수 있는 기법
- ✓ Sort Merge Join에서 조인 작업을 위해 항상 정렬 작업이 발생하는 것은 아님

Optimizer Join

❖ Sort Merge Join

- ✓ ordered use_merge(선행 테이블 이름) 힌트를 이용해서 수행

```
SELECT /*+ ordered use_merge(d) */ *
```

```
FROM EMP e, DEPT d
```

```
WHERE e.DEPTNO = d.DEPTNO AND e.DEPTNO = 10;
```



Optimizer Join

❖ Hash Join

- ✓ Hash Join은 해싱 기법을 이용하여 조인을 수행
- ✓ 조인 컬럼을 기준으로 해시 함수를 수행하여 서로 동일한 해시 값을 갖는 것들 사이에서 실제 값이 같은지를 비교하면서 조인을 수행
- ✓ NL Join 랜덤 액세스 와 Sort Merge Join 의 문제점인 정렬 작업의 부담을 해결하기 위한 대안으로 등장
- ✓ 수행 방식
 - 1) 선행 테이블에서 주어진 조건을 만족하는 행을 찾음
 - 2) 선행 테이블의 조인 키를 기준으로 해시 함수를 적용하여 해시 테이블을 생성(조인 컬럼과 SELECT 절에서 필요로 하는 컬럼도 함께 저장)
 - 3) (1 ~ 2번 작업을 선행 테이블의 조건을 만족하는 모든 행에 대해 반복 수행)
 - 4) 후행 테이블에서 주어진 조건을 만족하는 행을 찾음
 - 5) 후행 테이블의 조인 키를 기준으로 해시 함수를 적용하여 해당 버킷을 찾음(조인 키를 이용해서 실제 조인될 데이터를 찾음)
 - 6) 5) 조인에 성공하면 추출 버퍼에 넣음(3 ~ 5번 작업을 후행 테이블의 조건을 만족하는 모든 행에 대해서 반복 수행)|

Optimizer Join

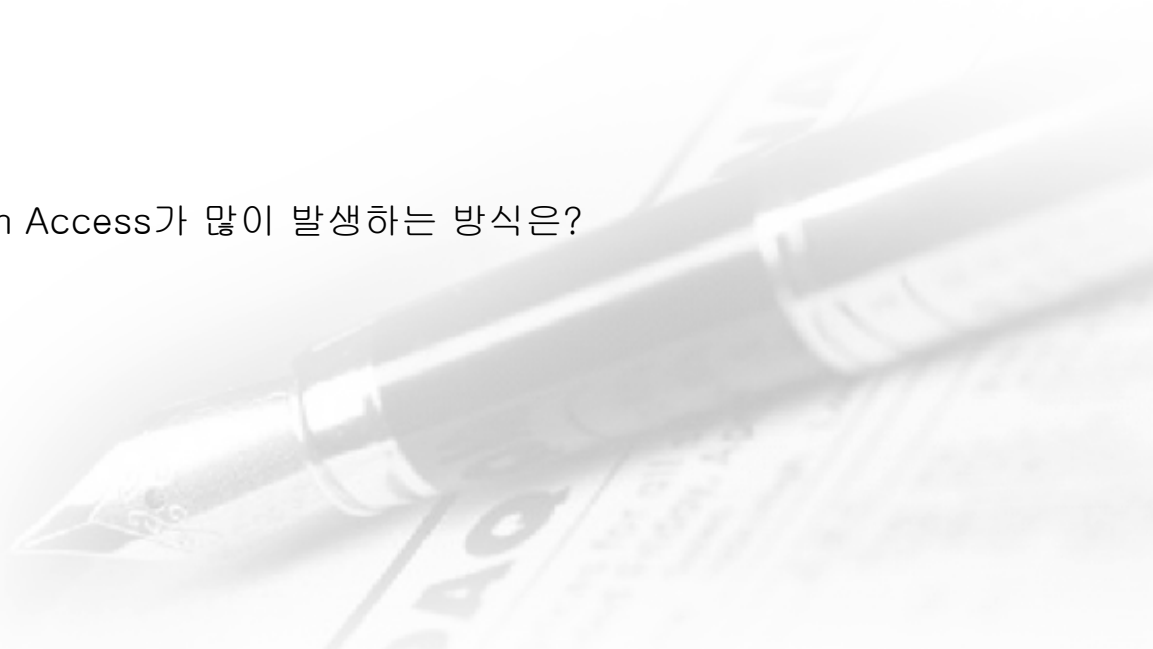
❖ Hash Join

- ✓ Hash Join은 조인 컬럼의 인덱스가 존재하지 않을 경우에도 사용할 수 있는 기법
- ✓ Hash Join은 해시 함수를 이용하여 조인을 수행하기 때문에 =로 수행하는 조인, 동등 조건에만 사용 할 수 있음
- ✓ 해시 함수가 적용될 때 동일한 값을 항상 같은 값으로 해싱 되는 것을 보장
- ✓ Hash Join 작업을 수행하기 위해 해시 테이블을 메모리에 생성
- ✓ 메모리에 적재할 수 있는 영역의 크기보다 커지면 임시 영역(디스크)에 해시 테이블을 저장
- ✓ Hash Join을 할 때는 결과 행의 수가 적은 테이블을 선행 테이블로 사용하는 것이 좋음
- ✓ 선행 테이블을 Build Input 이라하며, 후행 테이블은 Probe Input 이라고 함
- ✓ use_hash(선행 테이블 이름) 힌트를 이용해서 수행

```
SELECT /*+ use_hash(d) */ *  
FROM EMP e, DEPT d  
WHERE e.DEPTNO = d.DEPTNO AND e.DEPTNO = 10;
```



Optimizing

1. INDEX의 장단점을 기술
 2. 테이블 생성시 자동적으로 생성되는 INDEX가 있는데 어떤 제약 조건을 기술하면 생성되는가?
 3. EMP 테이블의 직급(JOB) 컬럼을 인덱스로 설정하되 인덱스 이름을 IDX_EMP_JOB 설정하고 확인
 4. 다음 중에서 검색 속도가 가장 빠른 것은?
 - 1) INDEX FULL SCAN
 - 2) INDEX RANGE SCAN
 - 3) INDEX UNIQUE SCAN
 - 4) INDEX DESC
 5. 옵티마이저의 조인 방식 중 Random Access가 많이 발생하는 방식은?
 - 1) Nested Loop JOIN
 - 2) Sort Merge JOIN
 - 3) Hash JOIN
 - 4) Full Table Scan
- 

Partitioning

❖ Table Partition

- ✓ <https://dev.mysql.com/doc/refman/8.0/en/partitioning.html>
- ✓ 파티션은 대용량의 테이블을 여러 개의 데이터 파일에 분리해서 저장하는 것
- ✓ 테이블의 데이터가 물리적으로 분리된 데이터 파일에 저장되면 입력, 수정, 삭제, 조회 성능이 향상 될 수 있음
- ✓ 파티션은 각각의 파티션 별로 독립적으로 관리될 수 있기 때문에 파티션 별로 백업하고 복구가 가능하며 파티션 전용 인덱스 생성도 가능
- ✓ 파티션은 Oracle 데이터베이스의 논리적 관리 단위인 테이블 스페이스 간에 이동이 가능
- ✓ 데이터를 조회할 때 데이터의 범위를 줄여서 성능을 향상시킴
- ✓ 파티션 테이블은 Primary Key를 설정하면 안됨
- ✓ 파티션 종류
 - RANGE Partition
 - LIST Partition
 - HASH Partition
 - KEY Partition



Partitioning

❖ Range Partition

- ✓ Column 값 의 범위를 기준으로 하여 행을 분할하는 형태로, 달, 분기 등의 logical 한 범위의 분산에 주로 사용
- ✓ 정해진 범위에 따라 비슷한 크기로 partition 이 예상되는 곳에 효율적
- ✓ Range Partition을 생성할 때 어느 행을 기준으로 어느 만큼의 값의 범위로 분할 할지를 다음 두 절에서 정의
 - PARTITION BY RANGE (column list..): 기본 Table에서 어느 Column을 기준으로 분할 할지를 설정
 - VALUES LESS THAN(value list..): 각 Partition이 어떤 값의 범위를 포함 할지 MAX Value값을 설정
 - PARTITION BY RANGE 절에서 지정 할 수 있는 Column은 한 개의 Column만으로 구성 할 수도 있고 결합 인덱스처럼 여러 개의 Column이 지정될 수 도 있음

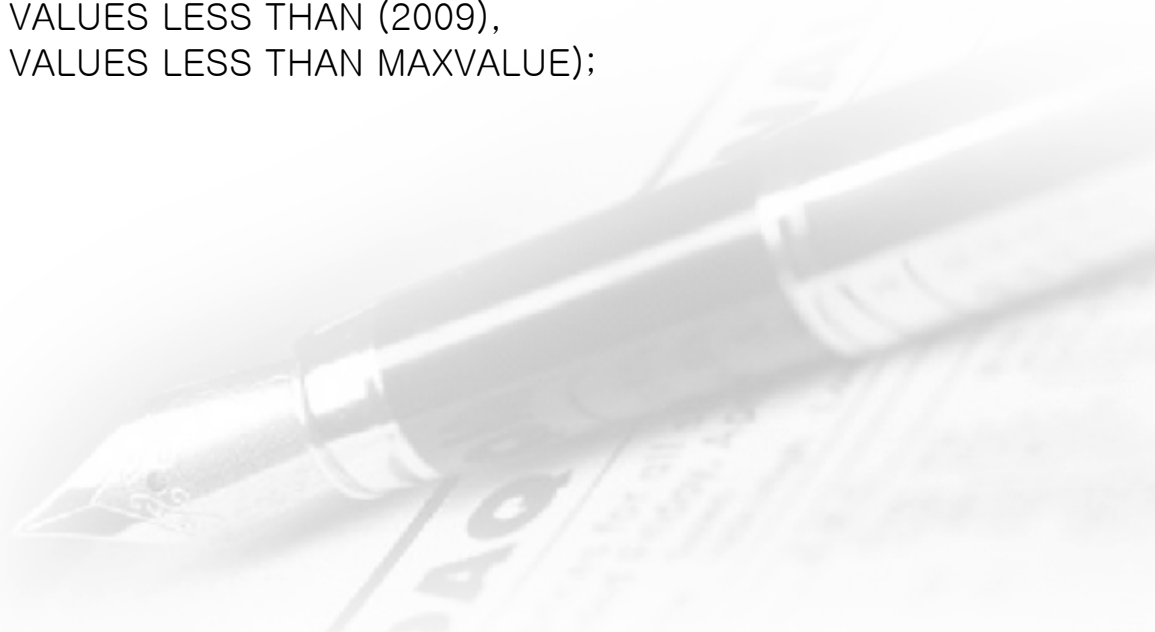


Partitioning

❖ Range Partition

- ✓ 파티션 생성 및 데이터 삽입

```
CREATE TABLE sales
(sales_no INT,
 sale_year INT NOT NULL,
 sale_month INT NOT NULL,
 sale_day INT NOT NULL,
 customer_name VARCHAR(30),
 price INT)
PARTITION BY RANGE (sale_year)
(PARTITION sales_q1 VALUES LESS THAN (2005),
 PARTITION sales_q2 VALUES LESS THAN (2009),
 PARTITION sales_q4 VALUES LESS THAN MAXVALUE);
```



Partitioning

❖ Range Partition

✓ 파티션 생성 및 데이터 삽입

```
INSERT INTO sales VALUES(1, 2004, 06, 12, 'scott', 2500);  
INSERT INTO sales VALUES(2, 2005, 06, 17, 'jones', 4300);  
INSERT INTO sales VALUES(3, 2005, 12, 12, 'miller', 1200);  
INSERT INTO sales VALUES(4, 2006, 06, 22, 'ford', 5200);  
INSERT INTO sales VALUES(5, 2005, 01, 01, 'lion', 2200);  
INSERT INTO sales VALUES(6, 2006, 12, 22, 'tiger', 3300);  
INSERT INTO sales VALUES(7, 2010, 11, 11, 'adam', 3800);
```

```
SELECT *  
FROM sales;
```



Partitioning

❖ Range Partition

✓ 파티션 확인

```
SELECT TABLE_SCHEMA, TABLE_NAME, PARTITION_NAME,  
       PARTITION_ORDINAL_POSITION, TABLE_ROWS  
FROM INFORMATION_SCHEMA.PARTITIONS  
WHERE TABLE_NAME = 'sales';
```



Partitioning

❖ Range Partition

- ✓ 특정 파티션의 데이터 확인

```
SELECT * FROM 테이블명 PARTITION (파티션명);
```

```
SELECT * FROM sales PARTITION (sales_q2);
```



Partitioning

❖ 파티션 작업

✓ 파티션의 병합(MERGE)

```
ALTER TABLE tbl_name  
  REORGANIZE PARTITION partition_list  
  INTO (partition_definitions);
```

```
ALTER TABLE sales  
  REORGANIZE PARTITION sales_q4 INTO (  
  PARTITION sales_q3 VALUES LESS THAN MAXVALUE);
```



Partitioning

❖ 파티션 작업

✓ 주의

- 파티션 테이블에는 외래 키를 설정할 수 없음(부모 테이블로서의 역할만 됨)
- 스토어드 프로시저, 스토어드 함수, 사용자 변수 등을 파티션 식에 사용할 수 없음
- 임시 테이블(with)은 파티션을 사용할 수 없음
- 파티션 키에는 일부 함수만 사용할 수 없음
- 파티션 개수는 최대 1,024개까지 지원
- 레인지 파티션은 숫자형 연속된 범위를 사용하고 리스트 파티션은 숫자형 또는 문자형 연속되지 않은 값(지역별, 혈액형 등)을 사용



Partitioning

❖ List Partition

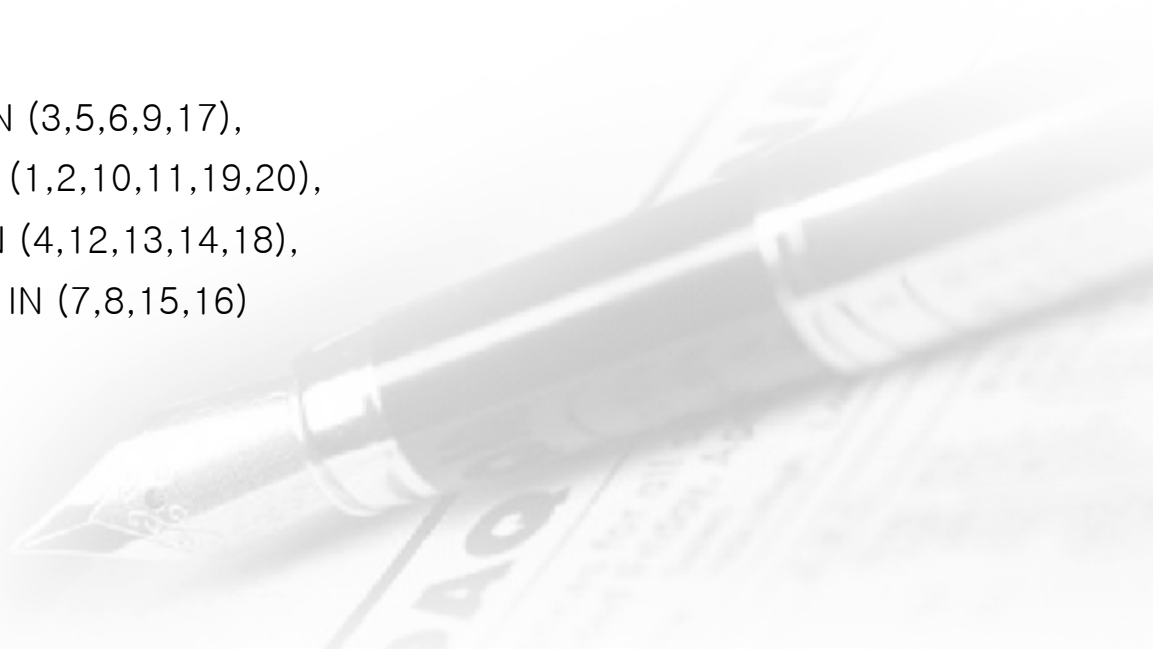
- ✓ 불연속적인 값의 목록을 각 파티션에 지정
- ✓ 순서와 상관없이 사용자가 미리 정한 그룹핑 기준에 따라 데이터를 분할 저장
- ✓ Column의 특정 값으로 Partitioning
- ✓ List partition의 장점은 연관되지 않은 데이터, 순서에 맞지 않는 데이터의 grouping 을 쉽게 할 수 있음
- ✓ 각 값 별로 분포도가 비슷하며 많은 SQL에서 해당 Column의 조건이 많이 들어오는 경우 유용
- ✓ 각 Partition의 분포도가 현격한 차이가 발생하지 않도록 해야 함
- ✓ 여러 컬럼으로 partition key 생성이 가능하지 않으며 오직 하나의 column 으로 구성
- ✓ Partition key 값은 NULL 값 또한 명시 가능하며 NULL 값을 포함한 어떠한 값이라도 한번만 명시할 수 있음
- ✓ 대소문자를 구분



Partitioning

❖ List Partition

```
CREATE TABLE employees (  
    id INT NOT NULL,  
    fname VARCHAR(30),  
    lname VARCHAR(30),  
    hired DATE NOT NULL DEFAULT '1970-01-01',  
    separated DATE NOT NULL DEFAULT '9999-12-31',  
    job_code INT,  
    store_id INT  
    )PARTITION BY LIST(store_id) (  
    PARTITION pNorth VALUES IN (3,5,6,9,17),  
    PARTITION pEast VALUES IN (1,2,10,11,19,20),  
    PARTITION pWest VALUES IN (4,12,13,14,18),  
    PARTITION pCentral VALUES IN (7,8,15,16)  
    );
```



Partitioning

❖ Hash Partition

- ✓ Partitioning Key 값에 해시 함수를 적용하여 Data를 분할하는 방식으로 History Data의 관리의 목적 보다는 성능 향상의 목적으로 나온 개념
- ✓ Hash Partition 은 Range Partition 으로 만들기 힘든 사항 즉, 조건을 주기 힘든 경우, 각 파티션 이 고르게 나누어지지 않아 밸런스를 유지하기 힘든 경우라고 판단되는 경우에 유리
- ✓ Hash Partition 의 경우 각각 다른 파티션에 데이터가 고르게 분산시키기 위해서는 반드시 파티션 의 개수를 명시하여야 하며, 파티션의 수를 2 의 거듭 제곱수 (즉, 2,4,8,16 ...)로 설정하여야 함
- ✓ NULL 값은 첫 번째 파티션에 위치해야 함



Partitioning

❖ Hash Partition

```
CREATE TABLE hash_employees (  
    id INT NOT NULL,  
    fname VARCHAR(30),  
    lname VARCHAR(30),  
    hired DATE NOT NULL DEFAULT '1970-01-01',  
    separated DATE NOT NULL DEFAULT '9999-12-31',  
    job_code INT,  
    store_id INT  
)PARTITION BY HASH( YEAR(hired) )  
PARTITIONS 4;
```



Partitioning

❖ Key Partition

- ✓ HASH 파티셔닝 과 유사하나 사용자가 분류 기준 컬럼을 정의하지 않고 데이터베이스 자체가 알아서 분류를 해서 나누는 차이가 있음

```
CREATE TABLE k1 (  
    id INT NOT NULL,  
    name VARCHAR(20),  
    UNIQUE KEY (id)  
)PARTITION BY KEY()  
PARTITIONS 2;
```



Partitioning

❖ 파티션 작업

✓ 파티션 이름 변경

● 형식

ALTER TABLE 테이블이름 REORGANIZE PARTITION 기존 파티션 이름 INTO (PARTITION 변경할 파티션 이름 VALUES 조건);

● sales 테이블의 sales_q4 파티션 이름을 sales_four로 변경하는 예제

```
ALTER TABLE sales
```

```
REORGANIZE PARTITION sales_q4 INTO (
```

```
PARTITION sales_q3 VALUES LESS THAN MAXVALUE);
```



Partitioning

❖ 파티션 작업

✓ 파티션 삭제

- Range, List 파티션만 가능
- 하나의 파티션은 반드시 남아 있어야 함
- 한번의 하나의 파티션만 삭제 가능
- 여러 개의 파티션을 삭제하고자 할 때는 삭제 문장을 여러 번 실행
ALTER TABLE 테이블이름 DROP PARTITION 파티션이름;



Partitioning

❖ 파티션 작업

✓ 파티션 추가

- MAXVALUE partition 이 존재하면 추가가 불가능
- 파티션 추가는 기존파티션 사이에 넣을 수는 없고 가장 이후 파티션 뒤로 생성이 가능
- 파티션을 생성했는데 range 범위를 잘못 생성했을 경우 삭제하고 재구성 해야 할 수도 있음
- 기본 형식

ALTER TABLE 테이블이름

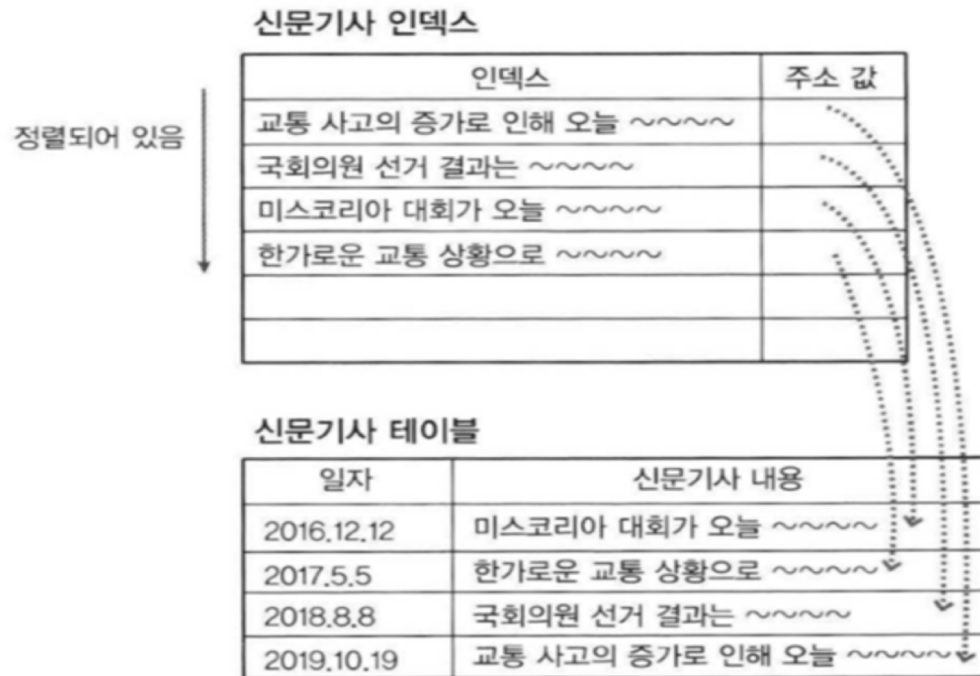
ADD PARTITION 파티션이름 VALUES 조건;



텍스트 검색

❖ MySQL 의 전체 텍스트 검색

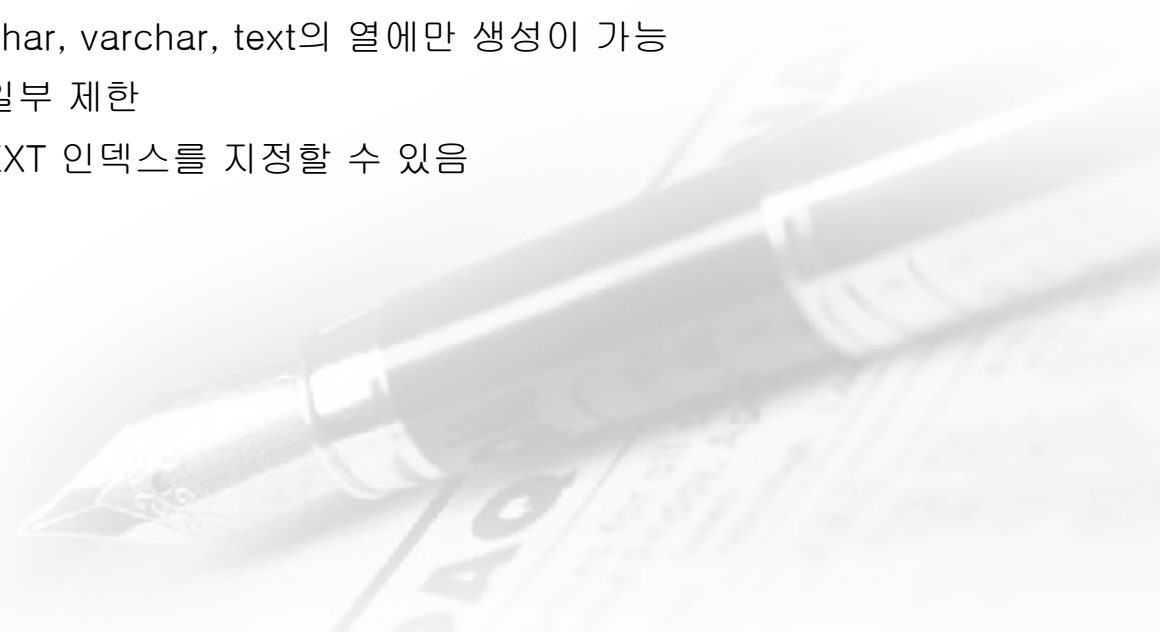
- ✓ 전체 텍스트 검색은 긴 문자의 텍스트 데이터를 빠르게 검색하기 위한 MySQL의 부가적인 기능
- ✓ 전체 텍스트 검색을 사용하면 기사의 내용에 포함된 여러 단어들에 인덱스가 설정되어서 검색 시에 인덱스를 사용하여 검색 속도가 월등히 빨라짐
- ✓ 전체 텍스트 인덱스는 신문기사와 같이 텍스트로 이루어진 문자열 데이터의 내용을 가지고 생성한 인덱스



텍스트 검색

❖ MySQL 의 전체 텍스트 검색

- ✓ 만약 10년치 기사에서 검색한다면 과부하가 발생하고 응답시간도 길어지는데 이것을 전체 텍스트 검색이 해결
- ✓ 전체 텍스트 검색은 첫 글자 뿐 아니라 중간의 단어나 문장으로도 인덱스를 생성해 주기 때문에 이와 같은 상황에서도 인덱스(정확히는 전체 텍스트 인덱스)를 사용할 수 있어 순식간에 검색 결과를 얻을 수 있음
- ✓ 전체 텍스트 인덱스는 신문기사와 같이 텍스트로 이루어진 문자열 데이터의 내용을 가지고 생성한 인덱스
- ✓ MySQL에서 생성한 일반적인 인덱스와 차이점
 - 전체 텍스트 인덱스는 InnoDB와 MyISAM 테이블만 지원
 - 전체 텍스트 인덱스는 char, varchar, text의 열에만 생성이 가능
 - 인덱스 힌트의 사용이 일부 제한
 - 여러 개의 열에 FULLTEXT 인덱스를 지정할 수 있음



텍스트 검색

❖ MySQL 의 전체 텍스트 검색

✓ 전체 인덱스 생성

```
CREATE TABLE 테이블이름(  
    열이름 데이터형식,  
    ...,  
    FULLTEXT 인덱스이름 (열이름)  
);
```

```
CREATE TABLE 테이블이름(  
    열이름 데이터형식,  
    ...,  
);  
ALTER TABLE 테이블이름  
    ADD FULLTEXT (열이름);
```

```
CREATE TABLE 테이블이름(  
    열이름 데이터형식,  
    ...,  
);  
CREATE FULLTEXT INDEX 인덱스이름  
    ON 테이블이름 (열이름);
```



텍스트 검색

❖ MySQL 의 전체 텍스트 검색

✓ 전체 인덱스 삭제

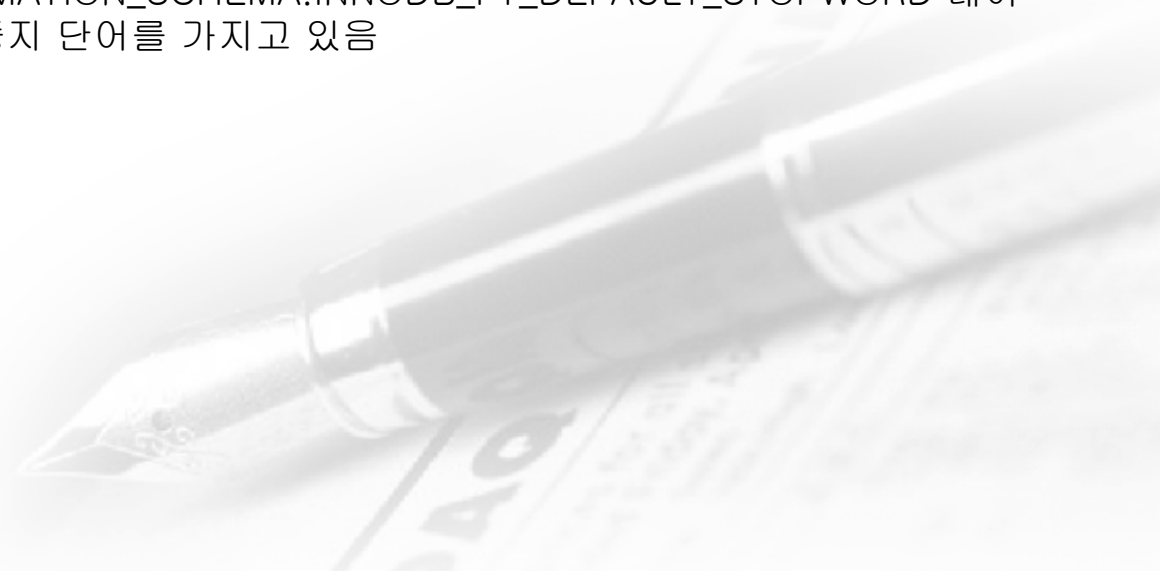
ALTER TABLE 테이블이름
DROP INDEX FULLTEXT (열이름);

✓ 중지 단어(stopwords)

- 실제로 검색할 때 무시할 만한 단어들은 아예 전체 텍스트 인덱스로 생성하지 않는 편이 좋음

이번 선거는 아주 중요한 행사이므로 모두 꼭 참여 바랍니다

- ◆ ‘이번’, ‘아주’, ‘모두’, ‘꼭’ 등과 같은 단어는 검색할 이유가 없으므로 제외하는데 이것이 중지 단어
- ◆ MySQL은 INFORMATION_SCHEMA.INNODB_FT_DEFAULT_STOPWORD 테이블에 약 36개의 중지 단어를 가지고 있음



텍스트 검색

❖ Maria DB 의 전체 텍스트 검색

✓ 자연어 검색

MATCH (col1,col2,...) AGAINST (단어)

- 단어와 일치하는 경우에만 검색
- 단어가 포함된 경우는 검색하지 않음
- 여러 개의 단어가 포함된 경우는 공백으로 구분
- 샘플

newspaper이라는 테이블의 article이라는 열에 전체 텍스트 인덱스가 생성되어 있다고 가정

‘영화’ 라는 단어가 들어간 기사를 찾으려면 다음과 같이 사용

```
SELECT * FROM newspaper  
WHERE MATCH(article) AGAINST('영화');
```

‘영화’ 또는 ‘배우’ 두 단어 중 하나가 포함된 기사 검색.

```
SELECT * FROM newspaper  
WHERE MATCH(article) AGAINST('영화 배우');
```

텍스트 검색

❖ Maria DB 의 텍스트 검색 최적화

✓ IN BOOLEAN MODE

MATCH (col1,col2,...) AGAINST (단어 IN BOOLEAN MODE)

- 단어가 포함된 경우 검색
- 단어 +단어를 이용하면 2개의 단어가 모두 포함된 경우 검색
- 단어 -단어를 이용하면 두번째 단어는 없는 경우만 검색
- 단어 ~ 단어를 이용하면 없는 열 보다 있는 열이 아래 순위
- 단어* 을 이용하면 포함된 데이터를 검색
- "단어"를 이용하면 정확히 일치하는 철자를 검색

