Oracle Developer Days

Oracle9i SQL Tuning Guide

이진호
한국오라클

1 인덱스, 조인, 정렬

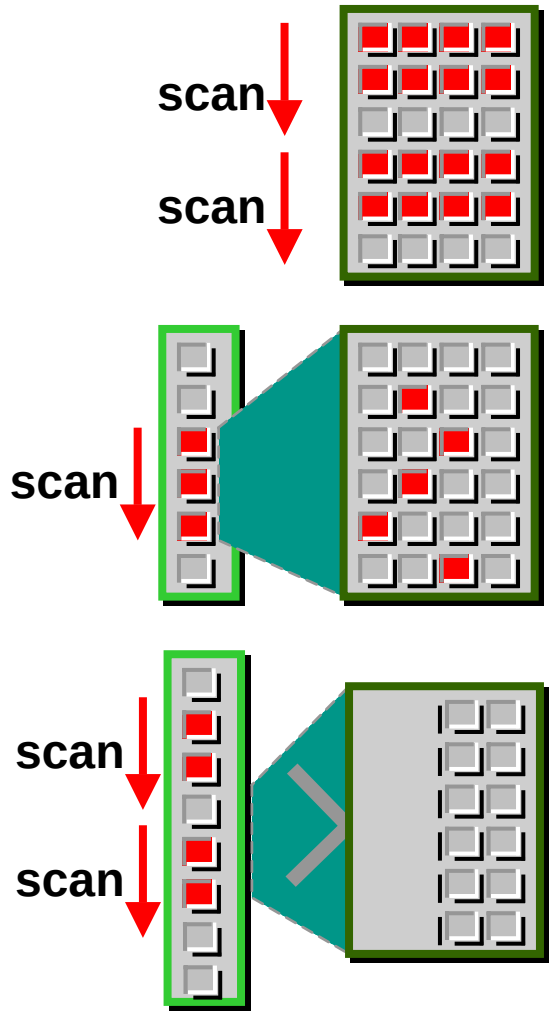
- 인덱스의 개념
- 조인의 종류
- 인덱스와 정렬

인덱스와 테이블

- 테이블의 데이터 분포
  - 입력 순서대로 데이터 저장 : 비 순차적인 블록
  - DELETE 등으로 비 연속적인 블록 구성
  - 대용량 테이블 액세스 시 과도한 블록 읽기 발생
  - 데이터베이스에서 블록 액세스가 가장 고비용
- 인덱스
  - 목적: 최소한의 데이터 블록 읽기를 통한 성능 향상
  - 대표 키와 ROW의 주소 정보(ROWID)만 미리 정렬
    - 대표 키 값을 통해 미리 정렬된 인덱스 만을 검색하여 해당 row만 바로 액세스
    - ROWID를 통한 액세스가 가장 빠른 조회 경로
- 블록 단위의 액세스 : 가장 작은 단위 디스크 I/O

데이터 블록 액세스 방식

- Full Table Scan
  - 순차적 블록 액세스
  - 멀티 블록 I/O 및 병렬화 가능
- Index Scan
  - 인덱스 블록 액세스 후
    ROWID를 통해 데이터 블록 획득
  - 비 순차적인 블록 액세스
  - 멀티 블록 I/O 사용 불가
- Fast Full Index Scan
  - 질의에 필요한 모든 컬럼이 인덱스에 포함된 경우
  - 멀티 블록 I/O 및 병렬화 가능



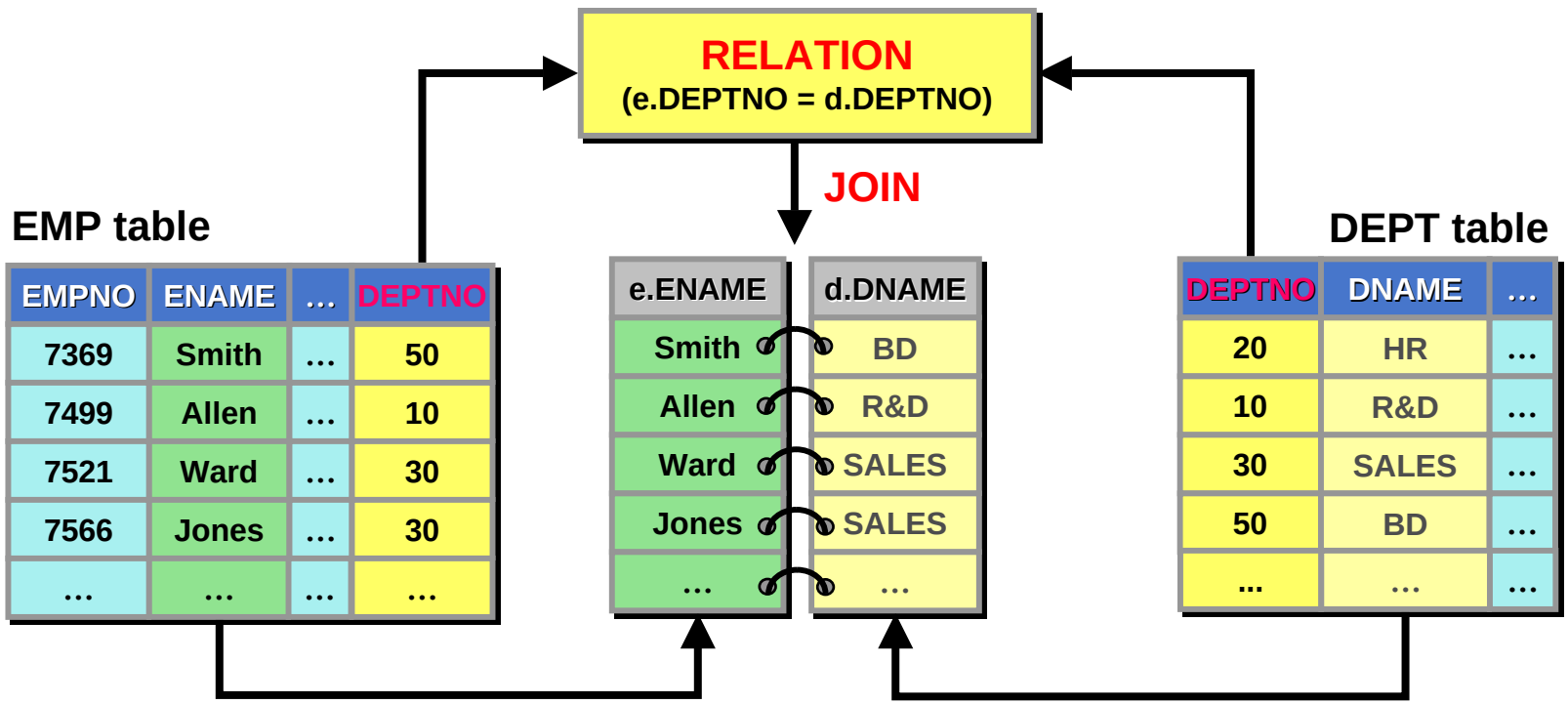
인덱스의 종류

- Unique, nonunique 인덱스
  - 고유 인덱스가 효율이 좋다.
- 결합 인덱스(Concatenated Index)
  - 여러 컬럼 값을 결합하여 하나의 인덱스화
- 인덱스 구조에 따른 구분
  - B* Tree
  - Bitmap
  - Reverse Key
  - Descending
  - Function-based
  - Domain Indexes

관계(Relation)와 조인

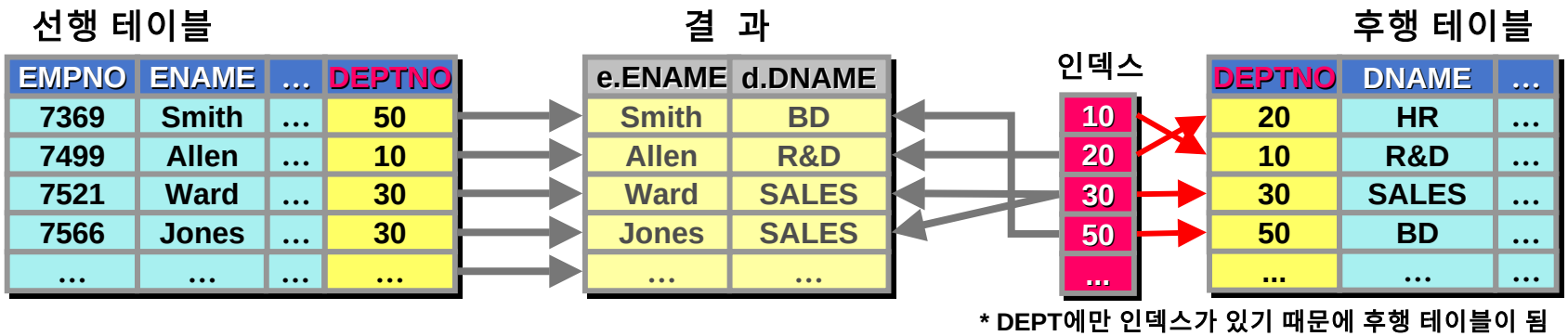
- 한 개 이상의 테이블을 특정 컬럼 값이 같은 것을 연계하여 데이터를 조회

```
SELECT e.ename, d.dname
FROM   DEPT d, EMP e
WHERE  e.deptno = d.deptno;
```



Nested Loop Join

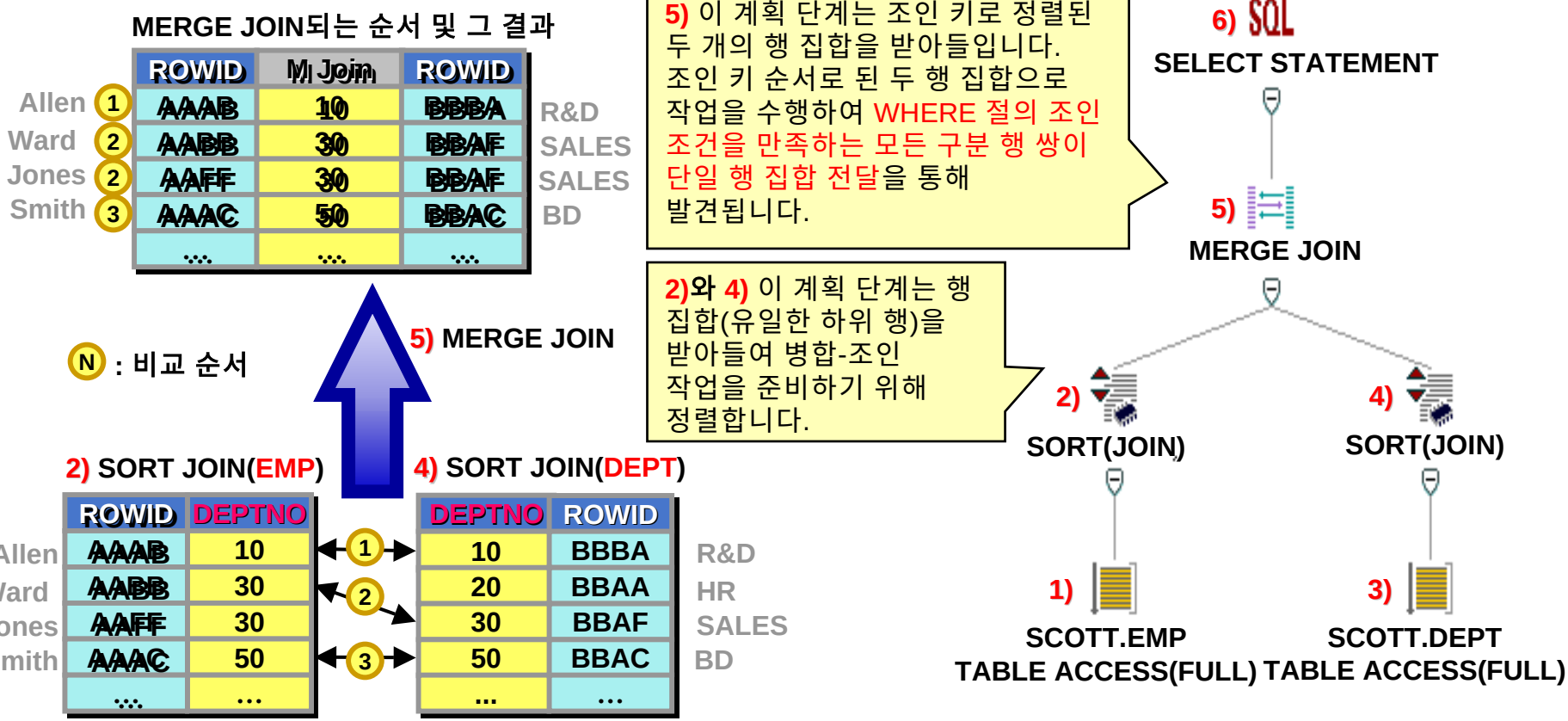
- 선행(Driving) 테이블의 조건에 맞는 데이터와 매치되는 값을 다른(후행) 테이블에서 참조
- 가장 많이 사용되는 기본적인 조인
- 상대적으로(?) 가장 빠른 조인 방법




```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   NESTED LOOPS
   2    1     TABLE ACCESS (FULL) OF 'EMP'
   3    1     TABLE ACCESS (BY INDEX ROWID) OF 'DEPT'
   4    3       INDEX (UNIQUE SCAN) OF 'DEPTNO_PK' (UNIQUE)
```

Sort Merge Join

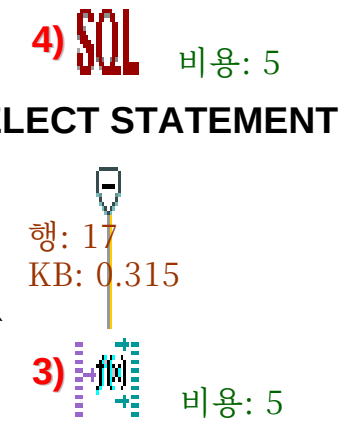
- 양쪽 컬럼에 대한 정렬 결과를 합쳐서 조인



Hash Join

- 한쪽 컬럼에 대한 Hash 테이블을 만들고 나머지 컬럼과 일치되는 값을 선별
- PGA에 생성(HASH_AREA_SIZE)

3) 이 계획 단계는 각각 다른 테이블에서 두 개의 행 집합을 받아들입니다. 해시 테이블은 첫번째 하위 행에서 반환된 행을 사용하여 작성됩니다. 두번째 하위에서 반환된 각 행은 해시 테이블을 조사한 다음 질의의 WHERE 절에 지정된 조건을 만족하는 행 쌍을 찾는 데 사용됩니다. 참고: Oracle 비용 기반의 최적기는 두 테이블 중 작은 것을 사용하여 해시 테이블을 구축합니다. 어느 것이 작은지 결정하기 위해 통계를 사용하므로 오래된 통계는 최적기가 잘못된 결정을 내리게 할 수도 있습니다.



HASH 함수, MOD(x,6) 적용 예제

1) HASH 테이블 생성(DEPT)

	값 + ROWID	HASH 값
SALES	(30,BBAF)	0
	.	1
HR BD	(20,BBAA) (50,BBAC)	2
	.	3
R&D	(10,BBBA)	4
	.	5

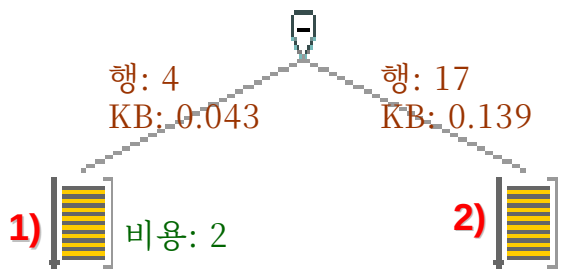
3) $f_h(50) = 2$

$f_h(10) = 4$

2) FULL ACCESS(EMP)

DEPTNO	ENAME	…
50	Smith	…
10	Allen	…
30	Ward	…
30	Jones	…
…	…	…

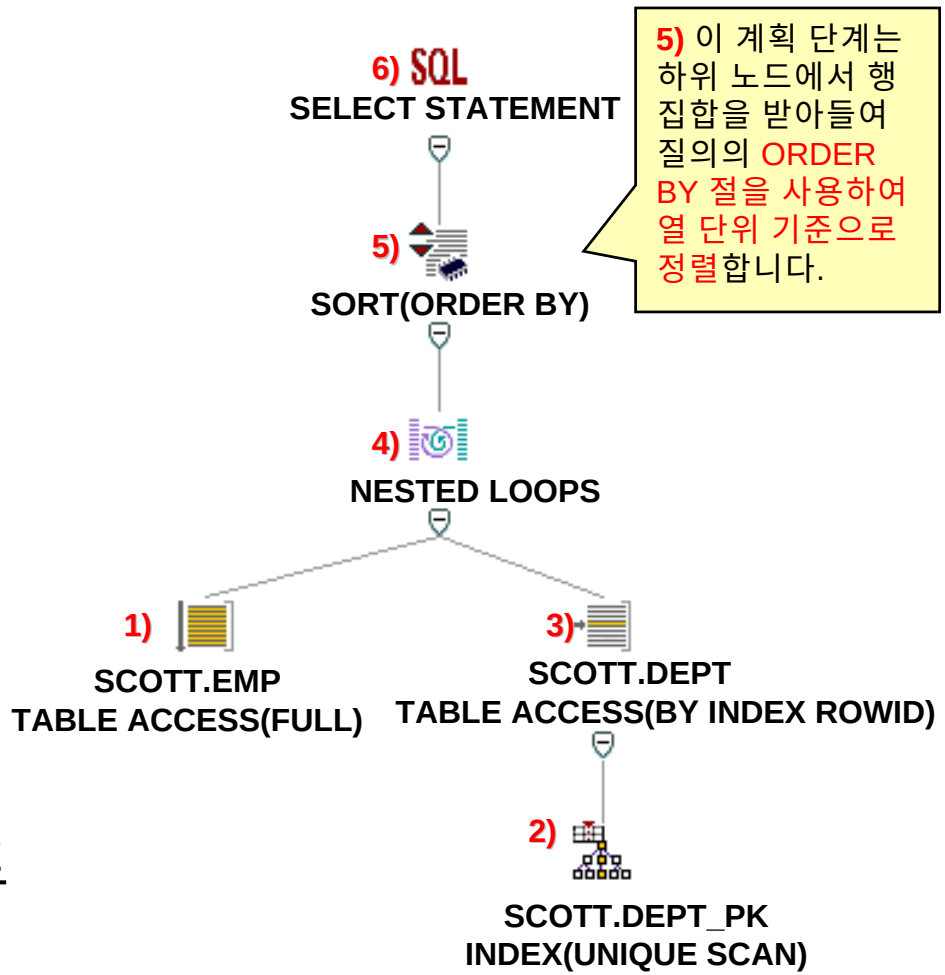
HASH JOIN



정렬

```
SELECT e.ename, d.dname
FROM   DEPT d, EMP e
WHERE e.deptno = d.deptno
ORDER BY e.ename;
```

- 고비용 실행 계획
- 정렬 종료 전에 인출 작업 불가
- PGA 공간에서 정렬
  - SORT_AREA_SIZE
- 정렬 영역 부족 시
  - 임시 테이블 세그먼트



2

Oracle Optimizer

- 개요
- RBO와 CBO
- Optimizer 조정

2.1 개 요

Optimizer의 역할

- SQL은 선언적 언어
  - 절차적(Procedural) 언어는 모든 처리 과정을 사용자가 기술
  - 선언적(Declarative) 언어인 SQL 문장이 표현하는 것은 :
    - 원하는 데이터의 집합 O
    - 그 데이터를 얻기 위한 구체적 방법 X
- 하나의 SQL을 수행하는 방법은 매우 다양.
- 그 다양한 방법 중 어떤 것을 이용하느냐에 따라 실행 효율에 커다란 차이가 존재.

Optimizer의 역할 (계속)

- SQL의 실행 계획을 수립하는 것이 바로 Optimizer
  - GOAL: 가장 효율적인 실행 계획을 찾는다.
  - 다양한 요소들이 전체적으로 고려되어야 하는 복잡한 문제 → RDBMS의 핵심 요소
- 최적화에는 크게 두 가지의 접근 방법이 존재
  - Rule-Based Optimization (RBO)
    - 미리 정해진 규칙에 따라 실행 계획을 수립
  - Cost-Based Optimization (CBO)
    - 실행 비용(Cost)의 예측을 통한 지능적인 실행 계획의 수립

2.2 Rule-Based Optimization과 Cost-Based Optimization

Rule-Based Optimization (RBO)

- 미리 정해진 규칙에 의한 실행 계획 결정
  - 연산자 우선 순위
  - 접근 경로의 우선 순위
  - SQL 문장의 syntax 규칙

  ➡ 상식에 의거하지만 융통성은 없음
- 데이터에 대한 통계 내지 실제 SQL을 수행할 경우에 소요될 비용에 대한 고려를 하지 않음
  - 예: 인덱스가 있다면 무조건 사용
    - 테이블의 크기, 인덱스의 효율 등 무시
- Oracle 버전 6 이하와의 역 호환성을 위해 지원

RBO 순위 결정 방식

순위	접근 경로
1	Single row by ROWID
2	Single row by cluster join
3	Single row by hash cluster key with unique or primary key
4	Single row by unique or primary key
5	Cluster join
6	Hash cluster key
7	Indexed cluster key
8	Composite index
9	Single-column index
10	Bounded range search on indexed columns
11	Unbounded range search on indexed columns
12	Sort-merge join
13	MAX or MIN of indexed column
14	ORDER BY on indexed column
15	Full table scan

Cost-Based Optimization (CBO)

- 데이터에 대한 각종 통계를 사용
- 실제 SQL을 수행할 때 소요될 비용을 예측하고 그 값을 기준으로 실행 계획 결정
  - I/O 횟수
  - CPU 사용량
  - 메모리 사용량
  - 네트워크 전송량

  ➡ 보다 현실적이고 지능적인 판단을 내림
- Oracle7에 처음 도입
  - 지속적으로 향상
  - 일반적으로 RBO보다 우수

CBO에서만 사용 가능한 기능들

- 테이블 및 인덱스의 Partitioning
- 인덱스 구성 테이블 (Index-organized table)
- Reverse key 인덱스
- Function-based 인덱스
- SELECT 문장에서의 SAMPLE 절
- 병렬 Query 및 병렬 DML
- Star Join 및 Star 변형
- Optimizer 확장
- Materialized View를 이용한 Query rewrite
- Enterprise Manager progress meter
- 해쉬 Join
- bitmap 인덱스 및 bitmap Join 인덱스
- 인덱스 skip scan 알고리즘

2.3 Optimizer 동작의 조정

Optimizer 모드 및 목표의 선택

- Oracle9*i* Optimizer의 기본 동작
  - 해당 SQL 문이 참조하는 개체들에 대해 사용할 수 있는 통계가 전혀 없는 경우 RBO 사용
  - 최소한 하나의 참조 개체에 대한 통계가 준비되어 있다면 CBO 사용
    - 기본적으로는 최적의 throughput이 목표
      - 전체 row의 빠른 처리를 지향
    - 빠른 Response time을 목표로 할 수 있도록 조정 가능
      - 처음 몇 row의 빠른 반환을 지향

Optimizer 모드 및 목표의 선택 (계속)

- 사용자 또한 Optimizer 모드 및 목표를 설정할 수 있음
  - 인스턴스 레벨
    - OPTIMIZER_MODE 파라미터의 설정 (파라미터 파일의 수정 또는 alter system 명령)
  - 세션 레벨
    - OPTIMIZER_MODE 파라미터의 설정 (alter session 명령)
  - 명령문 레벨
    - Optimizer Hint 사용

인스턴스 또는 세션 레벨

- OPTIMIZER_MODE 파라미터

값	의미
CHOOSE	Oracle9*i* Optimizer의 기본 동작에 따름. 디폴트 값.
RULE	통계 존재 여부에 상관없이 RBO 사용
ALL_ROWS	전체 처리율의 최적화
FIRST_ROWS[_n]	처음 결과가 나올 때까지의 시간을 줄이기 위해 최적화. n = 1, 10, 100, 또는 1000

SQL 문장 레벨

- Optimizer Hint

값	의미
/*+ CHOOSE */	Oracle9*i* Optimizer의 기본 동작에 따름.
/*+ RULE */	통계 존재 여부에 상관없이 RBO 사용
/*+ ALL_ROWS */	전체 처리율의 최적화
/*+ FIRST_ROWS[(n)] */	처음 결과가 나올 때까지의 시간을 줄이기 위해 최적화. n = 임의의 자연수

RBO 동작의 조정

- RBO는 SQL의 syntax에 기반하며, 내장된 규칙에 따라 동작
- 따라서 조정이 어려우며 조정할 수 있는 범위 또한 제한적
  - FROM 절에서 테이블 순서 변경
  - WHERE 절에서 술어 순서 변경
  - 인덱스의 생성 및 삭제
  - 기타 syntax 트릭의 사용

CBO의 조정

- 데이터에 대한 통계 수집
  - 가장 원칙적인 방법
  - CBO가 최적의 실행 계획을 산출하기 위해서는 정확한 통계 정보의 제공이 필수적
- Hint의 사용
  - SQL 문장 레벨
- CBO의 행동에 영향을 주는 파라미터의 설정
  - 인스턴스 또는 세션 레벨

Optimizer Hint

- 실행 계획의 결정은 원래 Optimizer의 역할, 그러나
  - Optimizer가 반드시 최적의 실행 계획만을 산출하는 것은 아님
  - 사용자가 Optimizer는 알 수 없는 정보를 알고 있을 수 있음
- 이러한 경우 사용자는 Hint를 이용하여 Optimizer의 결정에 영향을 미칠 수 있음
- 필요한 경우 Hint를 전역적으로 저장하는 방법에 의해 실행 계획의 일관성을 제공해 주는 Stored Outline 기능을 사용할 수 있음
- Hint는 /*+ RULE */을 제외하면 항상 CBO를 호출

CBO에 영향을 주는 파라미터들

- OPTIMIZER_MODE
- OPTIMIZER_FEATURES_ENABLE
- CURSOR_SHARING
- DB_FILE_MULTIBLOCK_READ_COUNT
- SORT_AREA_SIZE, HASH_AREA_SIZE, PGA_AGGREGATE_TARGET
- HASH_JOIN_ENABLED
- OPTIMIZER_INDEX_CACHING
- OPTIMIZER_INDEX_COST_ADJ
- OPTIMIZER_MAX_PERMUTATIONS
- PARTITION_VIEW_ENABLED
- QUERY_REWRITE_ENABLED
- STAR_TRANSFORMATION_ENABLED

3

일반적인 SQL 튜닝 가이드 및 예제

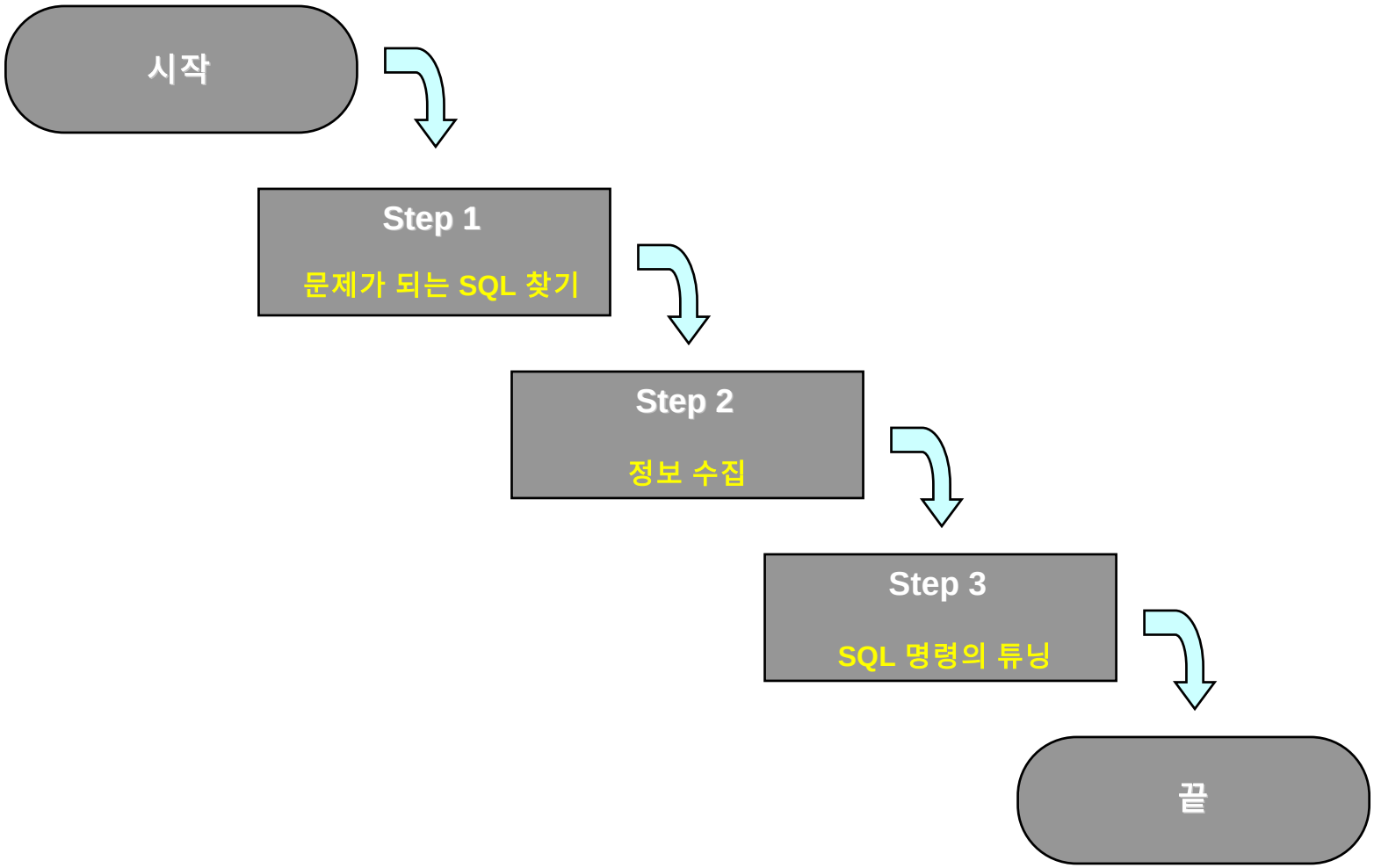
- 일반적 튜닝 가이드
- 예제

3.1 일반적인 SQL 튜닝 가이드

SQL 튜닝의 목표

- Response Time의 감소 또는 같은 작업을 처리하는 데 소요되는 Resource의 양 감소
- 접근 방법
  - Workload의 감소
    - 같은 작업을 보다 효율적으로 처리할 수 있는 방법의 모색
  - Workload의 re-scheduling
    - 각종 report나 batch 성격의 작업은 야간에 수행
  - Workload의 병렬화
    - Parallel Query, Parallel DML
    - 대용량의 데이터를 다루어야 하는 DW 환경에 적합

SQL 튜닝의 절차



문제가 되는 SQL 찾기

- 도구
  - 사용자의 feedback
  - SQL Trace와 TKPROF
  - STATSPACK Report
  - 각종 Dictionary View들
- 가장 눈 여겨 보아야 할 실행 통계
  - 논리적 읽기 수
  - 물리적 I/O 횟수 ➡ 처리된 데이터의 볼륨
  - Sort 횟수

찾아낸 SQL에 관련된 각종 정보 수집

- 해당 SQL 문장의 전문
- SQL 명령 내에서 참조된 오브젝트들의 구조
  - 테이블 구조
  - View의 정의 및 Base 테이블 구조
  - 관련된 인덱스 정보
- SQL 명령 내에서 참조된 오브젝트들의 CBO 통계
  - 마지막 분석 일자 포함
- 해당 SQL 명령의 수행에 사용된 실행 계획
  - 비교 대상이 될 수 있는 이전의 실행 계획이 있다면 그것도 포함

SQL 명령의 튜닝

- Optimizer의 의사 결정 지원
  - SQL의 실행 계획을 작성하는 것은 사용자가 아닌 Optimizer의 역할
  - 따라서 Optimizer가 최적의 판단을 내릴 수 있는 조건을 제공해 주는 것이 최우선의 과제
    - 적절한 데이터 설계
    - 필수적인 접근 경로의 제공
    - Optimizer 관련 파라미터의 조정
    - 정확한 Optimizer 통계
    - Optimizer Hint의 활용

SQL 명령의 튜닝 (계속)

- 효율적인 어플리케이션의 작성
  - SQL 명령은 결국 전체 어플리케이션의 맥락 내에서 사용되어 지는 것.
  - 이 부분은 Optimizer의 역할이 아닌 사용자의 역할
    - 효율적인 Logic, 효율적인 SQL의 구사
    - DBMS의 부하 감소
      - 호출 회수의 감소
      - contention의 제거

3.2 예제

인덱스와 데이터 형

```
SELECT empno, ename, job
FROM   emp
WHERE  empno = :b1;
```

입력된 직원 번호를 이용하여 직원 정보를 조회하라

조건

- empno 컬럼에 Primary Key. 즉, 인덱스가 존재
- empno 컬럼의 데이터 형은 VARCHAR2(4)
- 입력되는 bind 변수의 데이터 형은 숫자

실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   TABLE ACCESS (FULL) OF 'EMP'
```

예제

인덱스와 데이터 형 (계속)

Full Table Scan이 수행된 원인

컬럼과, 대응되는 바인드 변수의 데이터 형이 달라 묵시적 형 변환 발생

"WHERE TO_NUMBER(emp_id) = :b1"

WHERE 절 내에서 인덱스 컬럼에 변형이 있는 경우 인덱스를 사용할 수 없음

해법 1 : SQL의 재작성

```
SELECT empno, ename, job
FROM   emp
WHERE  empno = TO_CHAR(:b1);
```

해법 2: 데이터 형의 조정

empno 컬럼의 데이터 형을 number로 바꾼다.
→ 이 방법이 보다 근본적인 대책

예제

인덱스와 부정 조건

```
SELECT 'Not Exists'
FROM   dept
WHERE  dname != :b1;
```

특정 부서가 없음(삭제 되었음)을 확인하라.

조건

dname 컬럼에 대해 인덱스 존재

실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   TABLE ACCESS (FULL) OF 'DEPT'
```

Full Table Scan이 수행된 원인

부정형 조건에 대해 인덱스를 사용할 수 없음

예제

인덱스와 부정 조건 (계속)

해법 : 긍정형 문장으로 재작성

```
SELECT 'Not Exists'
FROM   dual
WHERE  NOT EXISTS (
    SELECT *
    FROM   dept
    WHERE  dname = :b1
);
```

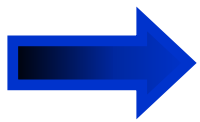
실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   FILTER
   2    1     TABLE ACCESS (FULL) OF 'DUAL'
   3    1     INDEX (RANGE SCAN) OF 'I_DEPT_DNAME' (NON-UNIQUE)
```

인덱스와 부정 조건 (계속)

실행 결과의 비교

```
'NOTEXISTS
----------
Not Exists
Not Exists
Not Exists
Not Exists
...
n 개의 행이 선택되었습니다.
```



```
'NOTEXISTS
----------
Not Exists

1 개의 행이 선택되었습니다.
```

Full Table Scan 수행
모든 행을 읽고 비교해야만 최종 결과를 얻을 수 있음

EXISTS의 특성 활용
단 한번의 읽기에 의해 최종 결과를 얻음

결론

인덱스의 사용이라는 점 뿐만 아니라 결과를 반환 받기 까지 처리해야 할 데이터의 볼륨을 줄였다는 점도 중요함

예제

인덱스와 null

```
SELECT empno, ename, job, comm
FROM   emp
WHERE  comm IS NULL;
```

보너스가 “없는” 직원을 조회하라.

조건

comm 컬럼에 대해 인덱스 존재

실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   TABLE ACCESS (FULL) OF 'EMP'
```

Full Table Scan이 수행된 원인

- IS NULL 조건에 대해 인덱스를 사용할 수 없음
- NULL 값의 분포도가 좁다면 더욱 큰 문제

예제

인덱스와 null (계속)

```
SELECT empno, ename, job, comm
FROM   emp
WHERE  comm IS NOT NULL;
```

보너스를 받은 직원을 조회하라.

실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   TABLE ACCESS (FULL) OF 'EMP'
```

Full Table Scan이 수행된 원인

- 부정형 조건에 대해 인덱스를 사용할 수 없음
- 그러나 이 경우에는 해법이 있음.

예제

인덱스와 null (계속)

해법 1 : SQL의 재작성

```
SELECT empno, ename, job, comm
FROM   emp
WHERE  comm > 0;
```

실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMP'
   2    1     INDEX (RANGE SCAN) OF 'I_EMP_COMM' (NON-UNIQUE)
```

- comm > 0 이라는 전제 조건 필요
- 불필요한 수식을 사용하지 않도록 주의한다.

  NVL(comm, -1) > 0

예제

인덱스와 null (계속)

해법 2 : 힌트의 사용

```
SELECT /*+ INDEX (emp i_emp_comm) */
       empno, ename, job, comm
FROM   emp
WHERE  comm IS NOT NULL;
```

실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=66 Card=20 Bytes=780)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (Cost=66 Card=20
Bytes=780)
   2    1     INDEX (FULL SCAN) OF 'I_EMP_COMM' (NON-UNIQUE) (Cost=26
Card=20)
```

- 인덱스의 전 범위를 scan하였음: INDEX (FULL SCAN)
- 이 경우 통계 정보는 내부 알고리즘에 의한 추정치를 사용하였음

예제

인덱스와 null (계속)

해법 3 : 데이터 구조의 조정

```
CREATE TABLE emp (
    ...    comm   NUMBER(7, 2) DEFAULT -1, ...
);

또는

ALTER TABLE emp MODIFY(comm DEFAULT -1);
```

IS NOT NULL

```
SELECT empno, ename, job, comm
FROM   emp
WHERE  comm > -1;
```

IS NULL

```
SELECT empno, ename, job, comm
FROM   emp
WHERE  comm = -1;
```

모두 인덱스 사용

예제

인덱스와 데이터 분포도

```
SELECT *
FROM   registrations
WHERE  status = 'HOLD';
```

상태가 'HOLD'인 수강 등록을 조회하라.

조건

- status 컬럼에 대해 인덱스 존재
- registrations 테이블에 대한 "통상의" 통계 정보는 존재

실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=31 Card=11250
Bytes=303750)
   1    0   TABLE ACCESS (FULL) OF 'REGISTRATIONS' (Cost=31
Card=11250 Bytes=303750)
```

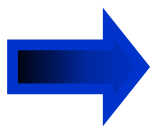
인덱스와 데이터 분포도 (계속)

Full Table Scan이 수행된 원인

데이터의 분포도에 대한 정확한 통계 정보는 없었음

status 컬럼의 분포도 조사

```
SELECT status, COUNT(*)
FROM   registrations
GROUP BY status;
```



```
STAT COUNT(*)
---- --------
CANC     2400
HOLD        5
PEND        2
REGI    53575
WAIT      270

5 개의 행이 선택되었습니다.
```

- status 컬럼의 데이터 분포는 skewed 되어 있음
- 그러나 Optimizer는 그 사실을 모르고 다음과 같이 계산

  selectivity = 1 / #distinct values = 1/ 5 = 20%

예제

인덱스와 데이터 분포도 (계속)

Histogram의 생성

```
ANALYZE TABLE registrations
COMPUTE STATISTICS FOR COLUMNS status;
```

동일한 Query를 다시 수행한 경우 실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=5 Bytes=135)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'REGISTRATIONS' (Cost=2
Card=5 Bytes=135)
   2    1     INDEX (RANGE SCAN) OF 'I_REGISTRATIONS_STATUS' (NON-
UNIQUE) (Cost=1 Card=5)
```

인덱스를 사용함

인덱스와 데이터 분포도 (계속)

status = 'REGI'로 변경한 Query의 실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=31 Card=53575
Bytes=1446525)
   1    0   TABLE ACCESS (FULL) OF 'REGISTRATIONS' (Cost=31
Card=53575 Bytes=1446525)
```

- Full Table Scan을 이용함
- 이제 CBO는 데이터의 실제 분포도에 따라 지능적인 판단을 할 수 있다.

인덱스와 데이터 분포도 (계속)

문제점 : 바인드 변수의 사용

```
SELECT *
FROM   registrations
WHERE  status = :b1;
```

bind는 실행 계획 생성 이후에 이루어지므로 이 경우 histogram은 사용될 수 없다.

한 가지 해법 : 데이터 구조의 조정

가장 분포도가 넓은 값('REGI')을 NULL로 변경한다.

보통 분포도의 값

```
SELECT *
FROM   registrations
WHERE  status = :b1;
```

인덱스 사용

넓은 분포도의 값

```
SELECT *
FROM   registrations
WHERE  status IS NULL;
```

Full Table Scan

예제

인덱스를 이용한 sort의 회피

```
SELECT empno, ename, job, comm
FROM   emp
ORDER BY comm;
```

직원을 보너스 순으로 조회하라.

실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   SORT (ORDER BY)
   2    1     TABLE ACCESS (FULL) OF 'EMP'
```

sort 작업이 발생된다. sort 작업은 테이블의 모든 row들을 읽어야만 결과를 반환할 수 있다.

예제

인덱스를 이용한 sort의 회피 (계속)

해법 : comm 컬럼에 인덱스를 생성하고 다음과 같은 Query를 이용

```
SELECT empno, ename, job, comm
FROM   emp
WHERE  comm >= 0;
```

실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMP'
   2    1     INDEX (RANGE SCAN) OF 'I_EMP_COMM' (NON-UNIQUE)
```

- sort 작업이 제거되었다. 이 경우 테이블의 모든 row들을 읽지 않더라도 즉시 결과를 반환할 수 있다.
- 전체 처리율보다는 빠른 응답 시간이 더 요구되는 상황에 유용

예제

인덱스를 이용한 sort의 회피 (계속)

```
SELECT  empno, ename, job, hiredate
FROM    emp
WHERE   hiredate >= TO_DATE(:b1, 'YYYY')
ORDER BY job DESC;
```

또 다른 예: 직원을 job 의 역순으로 조회하라.

job컬럼에 대한 싱글 인덱스 또는 job 컬럼이 앞에 오는 결합 인덱스가 있다면 다음 Query 사용

```
SELECT  /*+ INDEX_DESC (emp i_emp_job) */
        empno, ename, job, hiredate
FROM    emp
WHERE   hiredate >= TO_DATE(:b1, 'YYYY')
AND     job > ' ';
```

실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1 Card=1 Bytes=35)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (Cost=1 Card=1
Bytes=35)
   2    1     INDEX (RANGE SCAN DESCENDING) OF 'I_EMP_JOB' (NON-UNIQUE)
(Cost=2 Card=4)
```

예제

인덱스를 이용한 sort의 회피 (계속)

```
SELECT MAX(sal)
FROM   emp
WHERE  hiredate > TO_DATE(:b1, 'YYYY');
```

또 다른 예:
1980년부터 입사한 직원들의 봉급 중 최대값을 조회하라.

실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'EMP'
   3    2       INDEX (RANGE SCAN) OF 'I_EMP_HIREDATE_ JOB' (NON-
UNIQUE)
```

MIN/MAX 함수 역시 sort를 필요로 한다.

예제

인덱스를 이용한 sort의 회피 (계속)

해법 : (sal,_hiredate)의 결합 인덱스를 생성하고 다음과 같은 Query를 이용

```
SELECT /*+ INDEX_DESC (emp i_emp_sal_hiredate) */
       sal
FROM   emp
WHERE  hiredate > TO_DATE(:b1, 'YYYY')
AND    ROWNUM = 1;
```

실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=26 Card=1 Bytes=440)
   1    0   COUNT (STOPKEY)
   2    1     INDEX (FULL SCAN DESCENDING) OF 'I_EMP_SAL_HIREDATE'
(NON-UNIQUE) (Cost=26 Card=20 Bytes=440)
```

- 역시 sort 작업이 제거되었다. 이때 STOPKEY에 의해 읽는 row의 수는 단 한 개이다.
- select list에 sal만 있으므로 테이블 scan이 불필요했음

예제

결합 인덱스의 사용

먼저 결합 인덱스가 없는 경우를 먼저 생각해 본다.

```
SELECT *
FROM   registrations
WHERE  status = :b1
AND    appr_by = :b2;
```

상태와 승인자의 사번에 의해 수강 등록을 조회하라.

조건

- status 컬럼 및 appr_by 컬럼에 대해 각각의 인덱스 존재
- registrations 테이블에 대한 통상의 통계 정보는 존재

결합 인덱스의 사용 (계속)

OPTIMIZER_MODE = RULE인 경우의 실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'REGISTRATIONS'
   2    1     AND-EQUAL
   3    2       INDEX (RANGE SCAN) OF 'I_REGISTRATIONS_APPR_BY' (NON-
UNIQUE)
   4    2       INDEX (RANGE SCAN) OF 'I_REGISTRATIONS_STATUS' (NON-
UNIQUE)
```

- AND-EQUAL은 인덱스 Merge 방식의 접근 경로를 의미
- 인덱스 Merge
  - 다수의 단일 컬럼 인덱스를 사용하는 접근 경로
  - 각각의 인덱스로부터 해당하는 rowid의 집합을 얻은 후 그 교집합을 구하는 방식
  - 최종적인 테이블 접근은 즉시 이루어지지만 그 전에 인덱스 row 내에서의 scan이 많이 발생

결합 인덱스의 사용 (계속)

CBO의 실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=13 Card=112
Bytes=3024)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'REGISTRATIONS' (Cost=13
Card=112 Bytes=3024)
   2    1     INDEX (RANGE SCAN) OF 'I_REGISTRATIONS_APPR_BY' (NON-
UNIQUE) (Cost=1 Card=815)
```

- 인덱스 Merge 방식을 사용하지 않고 하나의 인덱스만을 사용
- 분포도가 좋은 컬럼에 대한 단일 인덱스를 사용하는 경우 인덱스 Merge 작업 도중 발생하는 인덱스 row의 scan 비용을 만회할 수 있음
  - 실제로 위의 Query에 대해 AND_EQUAL Hint를 사용하여 인덱스 Merge를 강제적으로 사용한 경우의 cost는 227임.

예제

결합 인덱스의 사용 (계속)

(status, appr_by)에 대한 결합 인덱스를 사용했을 때의 실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=5 Card=112
Bytes=3024)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'REGISTRATIONS' (Cost=5
Card=112 Bytes=3024)
   2    1     INDEX (RANGE SCAN) OF 'I_REGISTRATIONS_STATUS_APPR_BY'
(NON-UNIQUE) (Cost=1 Card=272)
```

- cost = 5로 최상의 성능 발휘
- 결합 인덱스란 미리 Merge되어 있는 당일 인덱스와 동등
  - 보통의 인덱스 Merge 방식에서 발생하는 인덱스 row의 scan없이 바로 해당하는 테이블의 rowid를 얻을 수 있기 때문
- 결론
  - WHERE 조건에서 AND 연산자에 의해 결합되어 자주 함께 사용되는 컬럼들이 있다면 그 컬럼들에 대해 결합 인덱스를 생성하는 것이 좋다.

예제

결합 인덱스와 인덱스 Skip Scan

```
SELECT *
FROM   registrations
WHERE  appr_by = :b1;
```

승인자의 사번에 의해 수강 등록을 조회하라.

실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=16 Card=815
Bytes=22005)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'REGISTRATIONS' (Cost=16
Card=815 Bytes=22005)
   2    1     INDEX (SKIP SCAN) OF 'I_REGISTRATIONS_STATUS_APPR_BY'
(NON-UNIQUE) (Cost=4 Card=12)
```

- Oracle8*i* 까지는 결합 인덱스의 Leading 컬럼이 WHERE 조건에 나타나지 않는 경우 그 결합 인덱스를 사용할 수는 없음
  - B* Tree 인덱스의 기본적 특징 : 결합 인덱스에 지정된 순서대로 정열 되어있음.
- Oracle9*i* 는 그러한 경우에도 결합 인덱스를 사용하여 접근 성능을 높일 수 있도록 하는 방법인 인덱스 Skip Scan 알고리즘을 제공
  - 위 경우 cost = 16으로 appr_by 컬럼에 대한 단일 인덱스를 사용한 경우의 성능(cost = 13)에 근접

예제

결합 인덱스의 순서

```
SELECT *
FROM   registrations
WHERE  status = :b1
AND    appr_by = :b2;
```

상태와 승인자의 사번에 의해 수강 등록을 조회하라.

분포도의 확인

appr_by → 1/74, status → 1/5 : (appr_by, status)의 순서가 더 좋지 않을까?

(appr_by, status)의 결합 인덱스를 사용한 경우의 실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=5 Card=112
Bytes=3024)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'REGISTRATIONS' (Cost=5
Card=112 Bytes=3024)
   2    1     INDEX (RANGE SCAN) OF 'I_REGISTRATIONS_APPR_BY_STATUS'
(NON-UNIQUE) (Cost=1 Card=272)
```

결합 인덱스의 순서는 동등 조건에 의한 인덱스 Scan 비용에 영향을 주지 않음

예제

결합 인덱스의 순서 (계속)

```
SELECT *
FROM   registrations
WHERE  appr_by BETWEEN :b1 AND :b2
AND    status = :b3;
```

상태와 승인자의 사번에 의해 수강 등록을 조회하라.

Cost의 비교

- (status, appr_by) 결합 인덱스를 사용한 경우
  - **cost = 14 :** status = :b3인 인덱스 로우를 찾았다면 이후에는 빠르게 appr_by 검색
- (appr_by, status) 결합 인덱스를 사용한 경우
  - **cost = 15:** :b1 <= appr_by <= :b2를 만족하는 인덱스 로우 범위에 대해 일일이 status 검색

결론

- 결합 인덱스에서는 앞 쪽의 컬럼이 scan 범위의 축소에 결정적 역할을 담당. 다만 동등 조건에 대해서는 차이가 없음
- 따라서 결합 인덱스의 순서는 다음과 같이 결정
  1. 동등 조건으로 쓰이는 컬럼이 앞에 와야 한다.
  2. 1번 조건이 차이가 없다면 그때에는 분포도가 좋은 컬럼을 앞에 둔다.

Fast Full 인덱스 Scan

```
SELECT appr_by, status
FROM   registrations;
```

수강 등록에 대해 그 상태와 승인자의 사번을 모두 조회하라.

조건

- (appr_by, status) 결합 인덱스가 존재
- appr_by 컬럼 및 status 컬럼에는 NOT NULL 제약 조건이 존재

실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=17 Card=56252
Bytes=506268)
   1    0   INDEX (FAST FULL SCAN) OF 'I_REGISTRATIONS_APPR_BY_STATUS'
(NON-UNIQUE) (Cost=17 Card=56
          252 Bytes=506268)
```

인덱스 Fast Full Scan은 순서를 보장하지 않음에 주의

예제

AND가 아닌 복합 조건

```
SELECT *
FROM   registrations
WHERE  class_id = :b1
OR     appr_by  = :b2;
```

특정 수업 또는 특정 승인자에 대한 수강 등록을 조회하라.

조건

(class_id, appr_by) 컬럼에 결합 인덱스존재

실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=31 Card=778
Bytes=21006)
   1    0   TABLE ACCESS (FULL) OF 'REGISTRATIONS' (Cost=31 Card=778
Bytes=21006)
```

OR 복합 조건에 대해서는 순차적 검색이 불가능하므로 결합 인덱스의 사용이 오히려 불리
→ 단일 인덱스라면 도움이 될 것인가?

예제

AND가 아닌 복합 조건 (계속)

class_id 컬럼과 appr_by 컬럼에 각각의 단일 인덱스가 있는 경우의 실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=18 Card=761 Bytes=20547)
   1    0   CONCATENATION
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'REGISTRATIONS' (Cost=6 Card=1
Bytes=27)
   3    2       INDEX (RANGE SCAN) OF 'I_REGISTRATIONS_APPR_BY' (NON-UNIQUE)
(Cost=2 Card=18)
   4    1     TABLE ACCESS (BY INDEX ROWID) OF 'REGISTRATIONS' (Cost=6 Card=1
Bytes=27)
   5    4       INDEX (RANGE SCAN) OF 'REG_PK' (UNIQUE) (Cost=2 Card=18)
```

내부적인 처리 : OR Expansion

```
SELECT * FROM registrations WHERE class_id = :b1
UNION ALL
SELECT * FROM registrations WHERE appr_by  = :b2;
```

예제

AND가 아닌 복합 조건 (계속)

```
SELECT *
  FROM registrations
 WHERE class_id IN (:b1, :b2, :b3);
```

조건에 맞는 특정 수업들에 대한 수강 등록을 조회하라.

실행 계획 : IN-List Iterator

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=3 Card=54 Bytes=1458)
   1    0   INLIST ITERATOR
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'REGISTRATIONS' (Cost=3
Card=54 Bytes=1458)
   3    2       INDEX (RANGE SCAN) OF 'I_REGISTRATIONS_CLASS_ID' (NON-
UNIQUE) (Cost=2 Card=54)
```

결론

AND가 아닌 조건에 대해서는 단일 인덱스의 존재가 유용하다.

AND가 아닌 복합 조건 (계속)

```
SELECT    *
FROM      registrations
WHERE     (status = 'HOLD' OR appr_by  = 85617)
AND NOT   (status = 'WAIT' AND appr_by  = 95482)
AND       (attended = 'Y');
```

조건에 맞는 특정 수업들에 대한 수강 등록을 조회하라.

실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=127 Card=2117
Bytes=57159)
   1    0   TABLE ACCESS (FULL) OF 'REGISTRATIONS' (Cost=127 Card=2117
Bytes=57159)
```

- OR Expansion 등의 변형을 적용하기에는 조건이 너무 복잡하다.
- 그러나 해법이 존재
  - Bitmap 인덱스

예제

AND가 아닌 복합 조건 (계속)

status, appr_by, attended 각각의 컬럼에 대해 Bitmap 인덱스를 생성한 후의 실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=81 Card=4855
Bytes=131085)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'REGISTRATIONS' (Cost=114
Card=4855 Bytes=131085)
   2    1     BITMAP CONVERSION (TO ROWIDS)
   3    2       BITMAP AND
   4    3         BITMAP OR
   5    4           BITMAP INDEX (SINGLE VALUE) OF
'BI_REGISTRATIONS_STATUS'
   6    4           BITMAP INDEX (SINGLE VALUE) OF
'BI_REGISTRATIONS_APPR_BY'
   7    3         BITMAP INDEX (SINGLE VALUE) OF
'BI_REGISTRATIONS_ATTENDED'
```

정리: 어떤 형태의 Query를 많이 사용하느냐에 따라 인덱스의 선정이 이루어져야 한다.

OPTIMIZER_MODE와 Join

```
SELECT co.short_name, cl.start_date
FROM   classes cl, courses co
WHERE  cl.instr_id = co.dev_id;
```

과목을 개발한 사람이 직접 강의를 하는 경우를 조회하라.

조건

- join key인 classes.instr_id 및 courses.dev_id 에 대해 두 테이블 모두 인덱스가 없음

OPTIMIZER_MODE = RULE일 때의 실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   MERGE JOIN
   2    1     SORT (JOIN)
   3    2       TABLE ACCESS (FULL) OF 'COURSES'
   4    1     SORT (JOIN)
   5    4       TABLE ACCESS (FULL) OF 'CLASSES'
```

- 인덱스가 없으므로 가용한 접근 경로는 Full Table Scan이 유일
- Sort/Merge 방식 사용 : RBO는 Hash 방식에 대해 알지 못함

예제

OPTIMIZER_MODE와 Join (계속)

OPTIMIZER_MODE = ALL_ROWS 일 때의 실행 계획

HASH_JOIN_ENABLED = FALSE

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=24
Card=3599 Bytes=82777)
   1    0   MERGE JOIN (Cost=24 Card=3599 Bytes=82777)
   2    1     SORT (JOIN) (Cost=8 Card=1018 Bytes=14252)
   3    2       TABLE ACCESS (FULL) OF 'COURSES' (Cost=3
Card=1018 Bytes=14252)
   4    1     SORT (JOIN) (Cost=16 Card=1729 Bytes=15561)
   5    4       TABLE ACCESS (FULL) OF 'CLASSES' (Cost=9
Card=1729 Bytes=15561)
```

HASH_JOIN_ENABLED = TRUE

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=13
Card=3599 Bytes=82777)
   1    0   HASH JOIN (Cost=13 Card=3599 Bytes=82777)
   2    1     TABLE ACCESS (FULL) OF 'COURSES' (Cost=3
Card=1018 Bytes=14252)
   3    1     TABLE ACCESS (FULL) OF 'CLASSES' (Cost=9
Card=1729 Bytes=15561)
```

OPTIMIZER_MODE와 Join (계속)

OPTIMIZER_MODE = FIRST_ROWS 일 때의 실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=FIRST_ROWS (Cost=13
Card=3599 Bytes=82777)
   1    0   HASH JOIN (Cost=13 Card=3599 Bytes=82777)
   2    1     TABLE ACCESS (FULL) OF 'COURSES' (Cost=3
Card=1018 Bytes=14252)
   3    1     TABLE ACCESS (FULL) OF 'CLASSES' (Cost=9
Card=1729 Bytes=15561)
```

- Join Key에 대한 인덱스가 없으므로 Nested Loop를 사용하는 경우 Full Table Scan이 수행되며 총 Cardinality(courses) * Cardinality(classes) 회 만큼의 테이블 읽기 및 Join 발생
- 따라서 OPTIMIZER_MODE = FIRST_ROWS임에도 불구하고 Hash 방식을 사용함.

Join Key 인덱스와 Join 순서

```
SELECT co.short_name, cl.start_date
FROM   classes cl, courses co
WHERE  cl.instr_id = co.dev_id;
```

과목을 개발한 사람이 직접 강의를 하는 경우를 조회하라.

조건

- join key에 대해 courses.dev_id 에만 인덱스가 존재

OPTIMIZER_MODE = RULE일 때의 실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=RULE
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'COURSES'
   2    1     NESTED LOOPS
   3    2       TABLE ACCESS (FULL) OF 'CLASSES'
   4    2       INDEX (RANGE SCAN) OF 'I_COURSES_DEV_ID'
(NON-UNIQUE)
```

- 인덱스가 있으므로 Nested Loop 방식 사용
- Join Key에 인덱스가 없는 classes 테이블이 무조건 driving 테이블

예제

Join Key 인덱스와 Join 순서 (계속)

OPTIMIZER_MODE = ALL_ROWS 일 때의 실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=13
Card=3599 Bytes=82777)
   1    0   HASH JOIN (Cost=13 Card=3599 Bytes=82777)
   2    1     TABLE ACCESS (FULL) OF 'COURSES' (Cost=3
Card=1018 Bytes=14252)
   3    1     TABLE ACCESS (FULL) OF 'CLASSES' (Cost=9
Card=1729 Bytes=15561)
```

- courses 테이블이 driving 테이블
- 그러나 Hash 방식에서 driving 테이블의 결정은 Join Key에 대한 인덱스 유무와는 상관없음
  - Hash 테이블을 구축하기 위해 우선 cardinality가 작은 쪽을 선택
  - 위 경우 courses 테이블이 classes 테이블에 비해 작은 테이블이므로 courses 테이블이 먼저 읽혀짐.

Join Key 인덱스와 Join 순서 (계속)

OPTIMIZER_MODE = FIRST_ROWS 일 때의 실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=FIRST_ROWS (Cost=3467
Card=3599 Bytes=82777)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'COURSES'
(Cost=2 Card=2 Bytes=28)
   2    1     NESTED LOOPS (Cost=3467 Card=3599 Bytes=82777)
   3    2       TABLE ACCESS (FULL) OF 'CLASSES' (Cost=9
Card=1729 Bytes=15561)
   4    2       INDEX (RANGE SCAN) OF 'I_COURSES_DEV_ID'
(NON-UNIQUE) (Cost=1 Card=5)
```

- 이번에는 OPTIMIZER_MODE의 설정에 알맞게 Nested Loop 방식 사용
  - 최적화의 목표가 비용보다 중요
- Join Key에 인덱스가 없는 classes 테이블이 무조건 driving 테이블
  - 이는 RBO, CBO 공통임

Join Key 인덱스와 Join 순서 (계속)

이번에는 OPTIMIZER_MODE는 여전히 FIRST_ROWS이고 Join Key에 대한 인덱스가 classes.instr_id에만 존재할 때의 실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=FIRST_ROWS (Cost=2039
Card=3599 Bytes=82777)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'CLASSES'
(Cost=2 Card=4 Bytes=36)
   2    1     NESTED LOOPS (Cost=2039 Card=3599 Bytes=82777)
   3    2       TABLE ACCESS (FULL) OF 'COURSES' (Cost=3
Card=1018 Bytes=14252)
   4    2       INDEX (RANGE SCAN) OF 'I_CLASSES_INSTR_ID'
(NON-UNIQUE) (Cost=1 Card=4)
```

- Join Key에 인덱스가 없는 courses 테이블이 무조건 driving 테이블
- cost는 조금 더 낮아짐
  - 만일 OPTIMIZER_MODE를 FIRST_ROWS로 설정하여 Nested Loop 방식을 사용하고자 한다면 Join Key에 대한 인덱스는 양쪽 모두에 만들어 주는 것이 바람직

Join Key 인덱스와 Join 순서 (계속)

OPTIMIZER_MODE = FIRST_ROWS이고 Join Key에 대한 인덱스가 양쪽 모두에 존재할 때의 실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=FIRST_ROWS (Cost=2039
Card=3599 Bytes=82777)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'CLASSES'
(Cost=2 Card=4 Bytes=36)
   2    1     NESTED LOOPS (Cost=2039 Card=3599 Bytes=82777)
   3    2       TABLE ACCESS (FULL) OF 'COURSES' (Cost=3
Card=1018 Bytes=14252)
   4    2       INDEX (RANGE SCAN) OF 'I_CLASSES_INSTR_ID'
(NON-UNIQUE) (Cost=1 Card=4)
```

- 예상대로 courses 테이블이 driving 테이블
  - 이번에는 어느 쪽을 driving 테이블로 하더라도 데이터 접근에 문제점은 없음
  - 다만 courses 테이블의 cardinality가 작기 때문에 driving 테이블이 된 것임
- 정리
  - Join 순서은 ① Join Key 인덱스의 상태 ② 다음 단계로 넘겨지는 row set의 cardinality에 의해 결정

Non-Join 조건과 Join 순서

```
SELECT co.short_name, cl.start_date
FROM   classes cl, courses co
WHERE  cl.instr_id = co.dev_id
AND    cl.class_id = :b1;
```

주어진 class_id에 대해 과목 이름 및 수업 시작일을 조회하라.

조건

- join key에 대해 courses.dev_id 및 classes.instr_id 모두에 인덱스 존재
- classes.class_id에 대한 조건 추가
- class_id는 classes 테이블의 Prinary Key. 즉 Unique 인덱스가 존재함

Non-Join 조건과 Join 순서 (계속)

OPTIMIZER_MODE = FIRST_ROWS 일 때의 실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=FIRST_ROWS (Cost=4
Card=2 Bytes=54)
   1    0   NESTED LOOPS (Cost=4 Card=2 Bytes=54)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'CLASSES'
(Cost=2 Card=1 Bytes=13)
   3    2       INDEX (UNIQUE SCAN) OF 'CLS_PK' (UNIQUE)
(Cost=1 Card=11163)
   4    1     TABLE ACCESS (BY INDEX ROWID) OF 'COURSES'
(Cost=2 Card=2 Bytes=28)
   5    4       INDEX (RANGE SCAN) OF 'I_COURSES_DEV_ID'
(NON-UNIQUE) (Cost=1 Card=5)
```

- classes 테이블이 driving 테이블인 Nested Loop 방식
  - OPTIMIZER_MODE = ALL_ROWS인 경우에도 동일
- classes 테이블에 대해 주어진 Non-Join 조건은 소위 “Single Row Predicate”로서 Nested Loop의 Outer Loop를 제거하는 효과
  - 선행 row set의 cardinality는 1 !

Star Join

```
SELECT    cs.description,
          c.start_date
FROM      cls_statuses cs,
          class_types  ct,
          classes      c
where     cs.description = 'Class rescheduled or
consolidated'
and       c.status = cs.cls_status
and       c.type = ct.type;
```

스케쥴이 재조정되었거나 다른 수업과 통합된 수업의 시작일을 조회하라

조건

- Join Key 중 하나인 class status에 대해 cls_statuses 테이블에 Primary Key가 존재
- 또 다른 Join Key인 class type에 class_types 테이블에 Primary Key가 존재
- Classes 테이블에는 Join Key에 관련한 인덱스는 존재하지 않음

Star Join (계속)

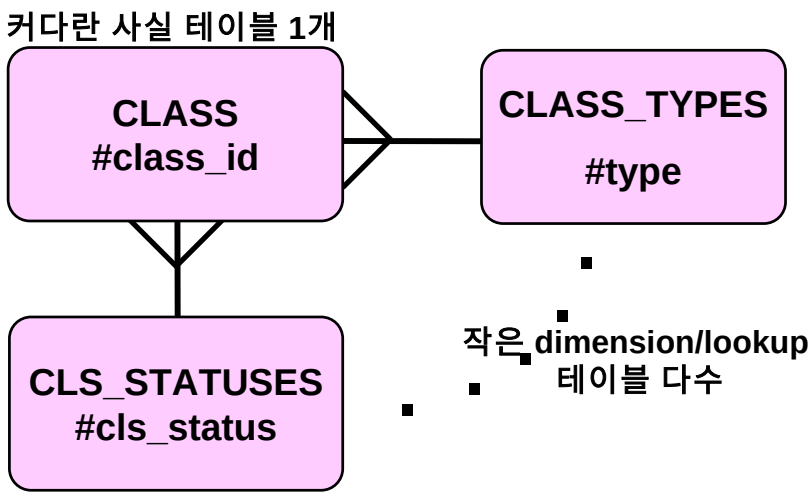
실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=60 Card=2847
Bytes=128115)
   1    0   NESTED LOOPS (Cost=60 Card=2847 Bytes=128115)
   2    1     HASH JOIN (Cost=60 Card=2847 Bytes=116727)
   3    2       TABLE ACCESS (FULL) OF 'CLS_STATUSES' (Cost=2 Card=2
Bytes=52)
   4    2       TABLE ACCESS (FULL) OF 'CLASSES' (Cost=24 Card=11163
Bytes=167445)
   5    1     INDEX (UNIQUE SCAN) OF 'CTYPE_PK' (UNIQUE)
```

- Join Key에 대한 인덱스가 없는 classes 테이블로 drive되는 2 중 Nested Loop Join은 선택되지 않음 : cls_status와 class_types 사이에는 연결 고리가 없음
- 이에 따라 먼저 가장 cardinality가 작은 cls_statuses 테이블이 driving 테이블이 되어 classes 테이블과 hash join을 수행한 후 그 결과가 class_types 테이블과 nested loop join

Star Join (계속)

Star Schema



DW 환경에서
쉽게 볼 수 있는
구조

Star Join

- Star schema를 위한 Oracle CBO의 기능
- 조건: 사실 테이블에 join key에 대한 결합 인덱스 필요 (Status + type)

Star Join (계속)

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=ALL_ROWS (Cost=28 Card=2847
Bytes=128115)
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'CLASSES' (Cost=2 Card=1861
Bytes=27915)
   2    1     NESTED LOOPS (Cost=28 Card=2847 Bytes=128115)
   3    2       MERGE JOIN (CARTESIAN) (Cost=4 Card=12 Bytes=360)
   4    3         TABLE ACCESS (FULL) OF 'CLS_STATUSES' (Cost=2 Card=2
Bytes=52)
   5    3         BUFFER (SORT) (Cost=2 Card=1 Bytes=4)
   6    5           INDEX (FULL SCAN) OF 'CTYPE_PK' (UNIQUE) (Cost=1
Card=1 Bytes=4)
   7    2       INDEX (RANGE SCAN) OF 'I_CLASSES_STATUS_TYPE' (NON-
UNIQUE) (Cost=1 Card=279)
```

- 먼저 cls_statuses와 class_types 테이블이 Cartesian Product
  - 둘 다 작은 테이블이므로 결과 row set의 크기가 크지 않음
- 이후 class 테이블과 nested loop Join을 수행
  - classes 테이블에 해당 결합 인덱스가 있으므로 최적

Top-N SQL

```
SELECT *
  FROM (SELECT COUNT(*), class_id
          FROM registrations
         GROUP BY class_id
         ORDER BY COUNT(*) DESC)
 WHERE ROWNUM <= 10
```

가장 많이 수강 신청된
10 개의 수업을
조회하라

실행 계획

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1443 Card=10 Bytes=234)
   1    0   COUNT (STOPKEY)
   2    1     VIEW (Cost=1443 Card=2828 Bytes=73528)
   3    2       SORT (ORDER BY STOPKEY) (Cost=1443 Card=2828 Bytes=11312)
   4    3         SORT (GROUP BY) (Cost=1443 Card=2828 Bytes=11312)
   5    4           INDEX (FULL SCAN) OF 'REG_PK' (UNIQUE) (Cost=345
Card=56252 Bytes=225008)
```

전체 집합을 sort할 필요가 없음 : sort 속도 향상 및 메모리 사용 감소

예제

비슷한 SQL들의 통합

```
SELECT   COUNT (*)
FROM     employees
WHERE    salary < 2000;

SELECT   COUNT(*)
FROM     employees
WHERE    salary BETWEEN 2000 AND 4000;

SELECT   COUNT(*)
FROM     employees
WHERE    salary > 4000;
```

직원들의 수를
봉급 범위 별로
카운트해라

- 3 개의 서로 다른 query로 분리하여 사용
- 그러나 기본 구조는 유사

예제

비슷한 SQL들의 통합 (계속)

통합된 SQL

```
SELECT COUNT (CASE WHEN salary < 2000
                   THEN 1 ELSE null END) count1,
       COUNT (CASE WHEN salary BETWEEN 2001 AND 4000
                   THEN 1 ELSE null END) count2,
       COUNT (CASE WHEN salary > 4000
                   THEN 1 ELSE null END) count3
FROM  employees;
```

- CASE 문장을 이용하여 하나의 query로 통합 (DECODE 함수를 사용해도 무방)
- Oracle Server 호출 및 테이블 방문 횟수가 1/3으로 줄어듬
  - 유사한 조작을 루프 내의 코드에 적용할 경우 효과는 배가

예제

절차적 처리를 하나의 SQL로

```
cursor c is select crs_id, short_name from courses;
..
loop
  fetch c into cid, sname;
  select count(*) into n from classes where crs_id = cid;
  if n = 0 then
    c_schedule := 'Not Scheduled';
  else
    schedule := 'Scheduled';
    select start_date into sdate from classes where
crs_id = cid;
    if sdate is null then
      c_date := ‘N/A’;
    elsif sdate > sysdate then
      c_date := to_char(sdate, 'YYYY-MM-DD');
    else
      c_date := 'Already Started';
    end if;
  end if;
  exit when c%notfound;
end loop;
```

과목들에 대한 수업 개설 여부와 수업 시작일을 출력하라.

예제

절차적 처리를 하나의 SQL로 (계속)

SQL로 표현

```
SELECT co.crs_id, co.short_name,
       CASE WHEN cl.class_id IS NULL THEN 'Not Scheduled‘
            ELSE 'Scheduled' END class_id,
       CASE WHEN cl.start_date IS NULL THEN 'N/A' else
         CASE WHEN cl.start_date > sysdate THEN
                TO_CHAR(cl.start_date, 'YYYY-MM-DD')
              ELSE 'Arleady Started' END
       END start_date
FROM   courses co, classes cl
WHERE  co.crs_id = cl.crs_id(+);
```

- CASE 문장과 outer join을 이용하여 하나의 SQL 문장으로 구사
- Optimizer의 최적화 단위는 하나의 SQL임을 이용하여 최적화를 전적으로 Optimizer에게 위임하는 형태
- 그러나 이러한 형태의 용법은 제한적으로 사용하여야 함


ORACLE®