
Freedom
A continued devotion to the value of the Digital Age (Speed) and the value of
humanity (passion) is the beginning to feeling the tranquility and salvation in what it means to be alive.
As your world unfolds before you with genuine technical skill in your hand,
you will feel leadership and independence in your environment.
Obsession with work allows you to attain freedom through it. It's the best feeling a man can get.

대용량 SQL 튜닝

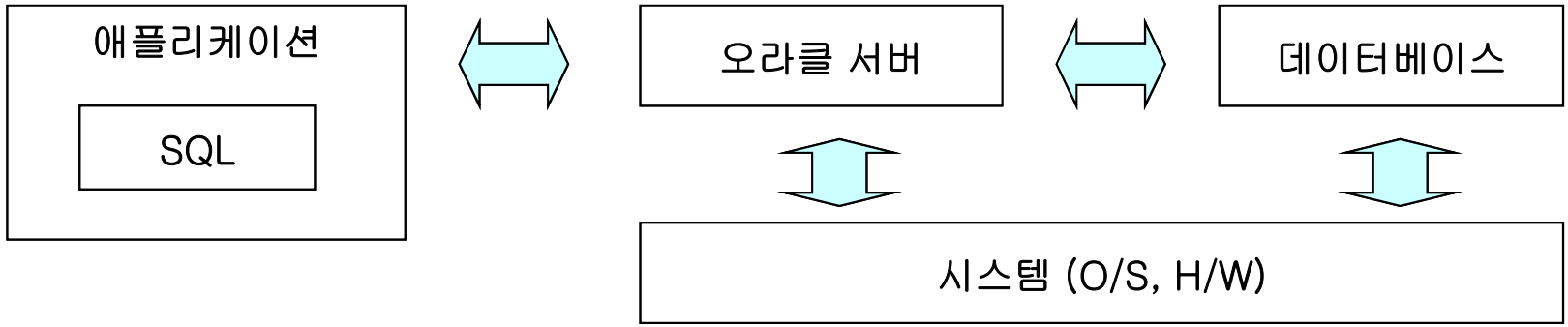
Contents

I. Introduction

1. **Tuning** 이란
2. **SQL Tuning**의 개념
3. **SQL Tuning** 선결 과제
4. **SQL Tuning Process**

I-1. Tuning 이란

- 쓸데없는 일을 시키지 않는 것.
  비효율 제거 (부분 범위 처리 와 처리 변경)
- 병목현상을 없애어 모든 처리 과정이 최상의 성능을 발휘하도록 하는 것.
- 최소의 자원으로 최대의 일을 시키는 것.
- 놀고 있는 자원이 없도록 하는 것.



I-2. SQL Tuning 이란

- SQL = 원하는 결과 집합을 표현하는 언어.
- SQL은 사용자가 RDBMS와 통신 할 수 있는 유일한 방법!
- SQL은 어떻게(HOW)가 아니라 무엇(WHAT)을 기술.
- 하나의 결과집합을 위해서 많은 SQL 사용 가능.

  ex) {1,2} 라는 집합
- 하나의 SQL을 수행하기 위해서 많은 수행 방법 가능. (일량을 결정)

- SQL 튜닝은 가장 적은 일량으로 원하는 결과집합을 도출해 내는 것.
- RDBMS는 모든 처리를 Optimizer가 수립한 실행계획으로 처리.
- 사용자의 모든 SQL은 DBMS Call로 처리.
- DATA와 처리에 대한 접근 방법

  - 사용자 처리 관점 접근:

    RDBMS는 단지 DATA의 저장소란 개념으로 접근, 모든 처리는 사용자의 로직 으로 건별 처리.

  - 서버 처리 관점 접근:

    모든 처리는 RDBMS 안에서 사용자가 요청한 결과 집합(SQL)에 대해서 RDBMS 내부적으로 처리.

  → 백만건을 처리하는 SQL도 한건을 처리하는 SQL도 수행시 1번의 DBMS Call()을 발생!!!.

I-2. SQL Tuning 이란

❑ **부분 범위 처리**

부분 범위 처리는 두가지 의미가 있다.

첫째 **전체 처리 중 일부분씩만을 처리** 하는 것.

- 페이징 처리, 운반 단위 처리 등.

둘째 **전체처리를 일부 처리만으로 대체** 하는 것.

- Ordering 을 Index로 대체.
- Table 처리 → Index 만으로 처리.(결합 인덱스 구성, PK Count() 등)
- 불필요한 조인 제거.
- 반복 수행 제거.(임시 테이블 사용, 함수 반복 제거 등)
- Index -> Table Access → IOT 사용.

I-3. SQL Tuning 선결과제

❑ 데이터 모델 설계(설계자)

- 업무를 가장 잘 표현하는 단순 명료한 데이터 모델 필요
- 명확한 업무 분석이 선결 과제
- 업무 범위의 명확한 구분

❑ 오라클 서버 튜닝 (데이터베이스 관리자)

- SQL 특성에 맞도록 오라클 서버튜닝
- 테이터 베이스 메모리 및 I/O 튜닝

❑ 운영 체제 튜닝 (운영 체제 관리자)

- 오라클 서버가 운용되는 데 필요한 기본적인 리소스 파라미터 튜닝

❑ 업무 기능 분석 (개발자)

- User Data에 대한 정확한 이해
- 명확한 업무 분석 및 설계

I-4. SQL Tuning Process

- **튜닝 대상 선정 : 업무적 선정, 모니터링 선정, Trace 대상 선정 등.**
  - 비용 대비 효과 검증 : 평균 튜닝 시간, 투입 인력 대비 효과.
- **SQL 확인.**
  - SQL 결과 집합, 처리 조건, 조건 범위, 수행 빈도, 수행 시간대 등.
  - 처리 오브젝트 정보(테이블, 인덱스의 기본 정보-크기, 건수, 키수, 컬럼 구성 등)
- **SQL 수행 확인**
  - 실행계획, TRACE, 실행 환경 등 파악.
- **튜닝 포인트 확인**
  - 처리 비효율 및 개선점 파악.
- **SQL 튜닝**
  - 인덱스 변경/조정, 조인 조정/변경, 엑세스 변경 등.
  - SQL 재 작성.
  - 오브젝트, 처리 프로세스 변경 등.
- **튜닝 검증 및 적용**
  - 튜닝 전후 결과 일치 확인 및 성능 개선 검증.
  - '튜닝 포인트'와 '개선 포인트' 의 정확한 정리 및 검증, 문서화.
  - 유사 사례 적용.

    (인덱스 또는 프로시져(펑션)등의 조정/ 변경 시 관련 SQL 검증 및 개선 필요.

I-4. SQL Tuning Process

❑ 튜닝 대상 선정.

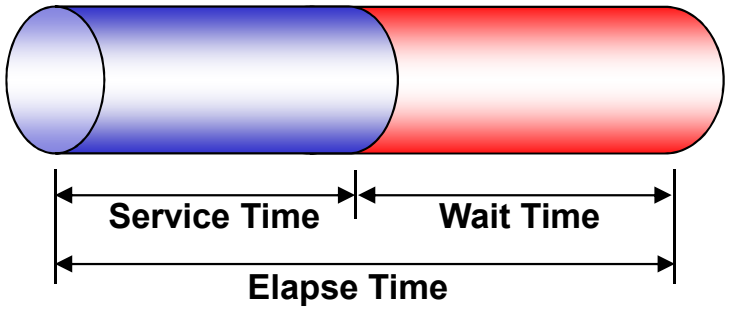
- 업무적 접근 : 사용자별 튜닝 대상 선정, 중요도 우선 업무단위 튜닝 대상 선정.
- TRACE 분석 : 전체 TRACE 수집 후 수행시간, 수행횟수, 처리량 등의 기준으로 정렬 후 대상 선정.
- STATPACK : 분석 보고서의 TOP SQL 등으로 대상 선정.
- Clinet Tool : 모니터링 시간에 수행된 세션별 선정으로 대상 선정.
- Server Log : 리얼타임 분석, 수집된 로그 분석으로 다양한 기준으로 대상 선정 가능.

❑ 튜닝 접근 방법.

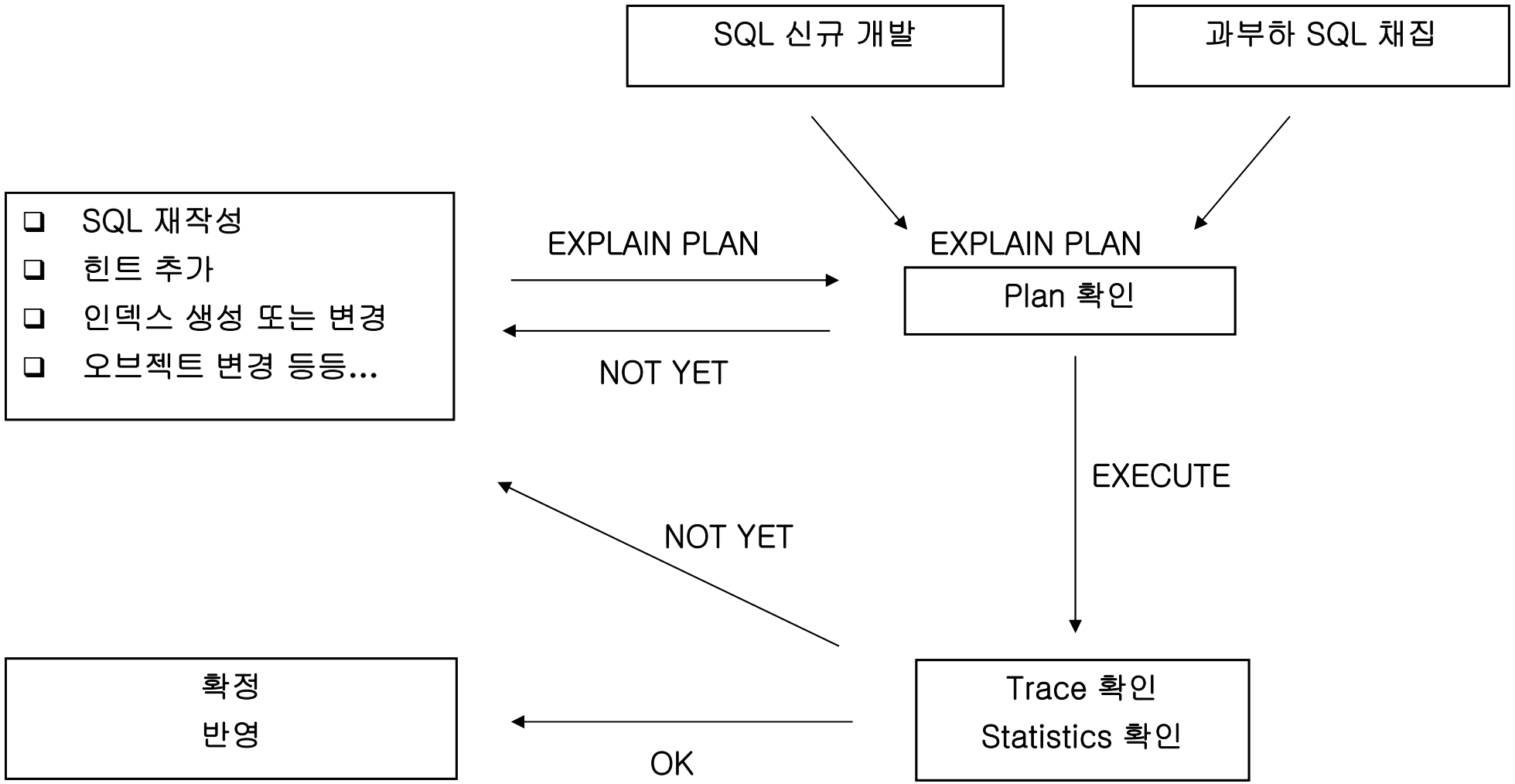
- Stat 기준 : 주로 수행시간, 또는 처리 로우수를 튜닝 기준.(주로 Speed 기준)
- Wait 기준 : 대기 발생량, 총 대기 시간을 튜닝 기준.

❑ Response Time 분석(Wait 분석)

- SQL의 전체 수행시간은 Service Time(CPU Time)과 Wait Time(Queue Time)으로 구성.
- SQL의 수행을 일량 과 Wait Event 별 Wait Time까지 분석을 하여 정확한 튜닝.



I-4. SQL Tuning Process



I-5. 효율적인 SQL -기본.

- **RDBMS에 대한 사전 지식**

  RDBMS Architacture, RDB 원리 , Optimizer 이해 등.

- **정확한 SQL 문법 숙지**

  SQL 내장 함수 이해 (DECODE, CASE, FLOOR, CEIL,SUBSTRB,INSTRB,REPLACE,LTRIM,
  LPAD, TRANSLATE, MOD, ROUND, TRUNCATE, TO_CHAR, TO_DATE,
  LEAST, GREATEST, NVL , 날짜함수, 그룹함수 등)

  DBMS별 SQL 기능 파악 (ROWID, ROWNUM, Scalar SubQuery, AnalyticFunction 등)

- **엑세스 이해**

  Index, Table 엑세스 이해

  조인 이해 및 적용 파악

  SELECT, INSERT, UPDATE, DELETE 처리 이해.

- **고급 SQL 기법 이해**

  다양한 데이터 및 조건 처리 방법 파악(퀴즈 등의 예시자료)

- **실행계획 확인 및 보완 기술.**

  실행계획, SQL Trace, Wait/Latch 비교.

  단일사용자 수행 과 동시사용자 수행 환경 고려.

I-5. 효율적인 SQL - 작성순서.

- **SQL 목적 파악**
  SQL로 처리하고자 하는 정확한 결과 도출

- **처리 조건 파악**
  상수 조건, 사용자 입력 조건 파악(필수 조건, 조건 범위, 조건 한계 등)

- **처리 오브젝트 확인**
  필요한 테이블, 뷰, 인덱스 파악(테이블간 관계, 건수-전체건수,값의 종류 등)

- **처리 순서/ 방법 결정**
  업무적 처리 순서 결정 -> SQL 처리 순서와 동일.

- **SQL 작성**

- **실행계획 확인 및 보완**
  실행계획, SQL Trace, Wait/Latch 비교
  힌트 및 조건 변경.

I-5. 효율적인 SQL – 작성 방법.

- SQL Format 사용.
  들여쓰기, 명명 규칙 등의 작성 규칙 사용으로 가독성 향상 및 정확한 의미/처리 분류.

- SQL 주석 사용.
  SQL별로 사용 프로그램, 모듈, 작성일, 업무명칭, 최종 수정일 등 고정형식으로 관리.

- 오브젝트 처리 순서 지정
  업무적으로 처리 순서가 확정되는 경우(대부분의 경우) FROM 절에 순서대로 집합 지정.

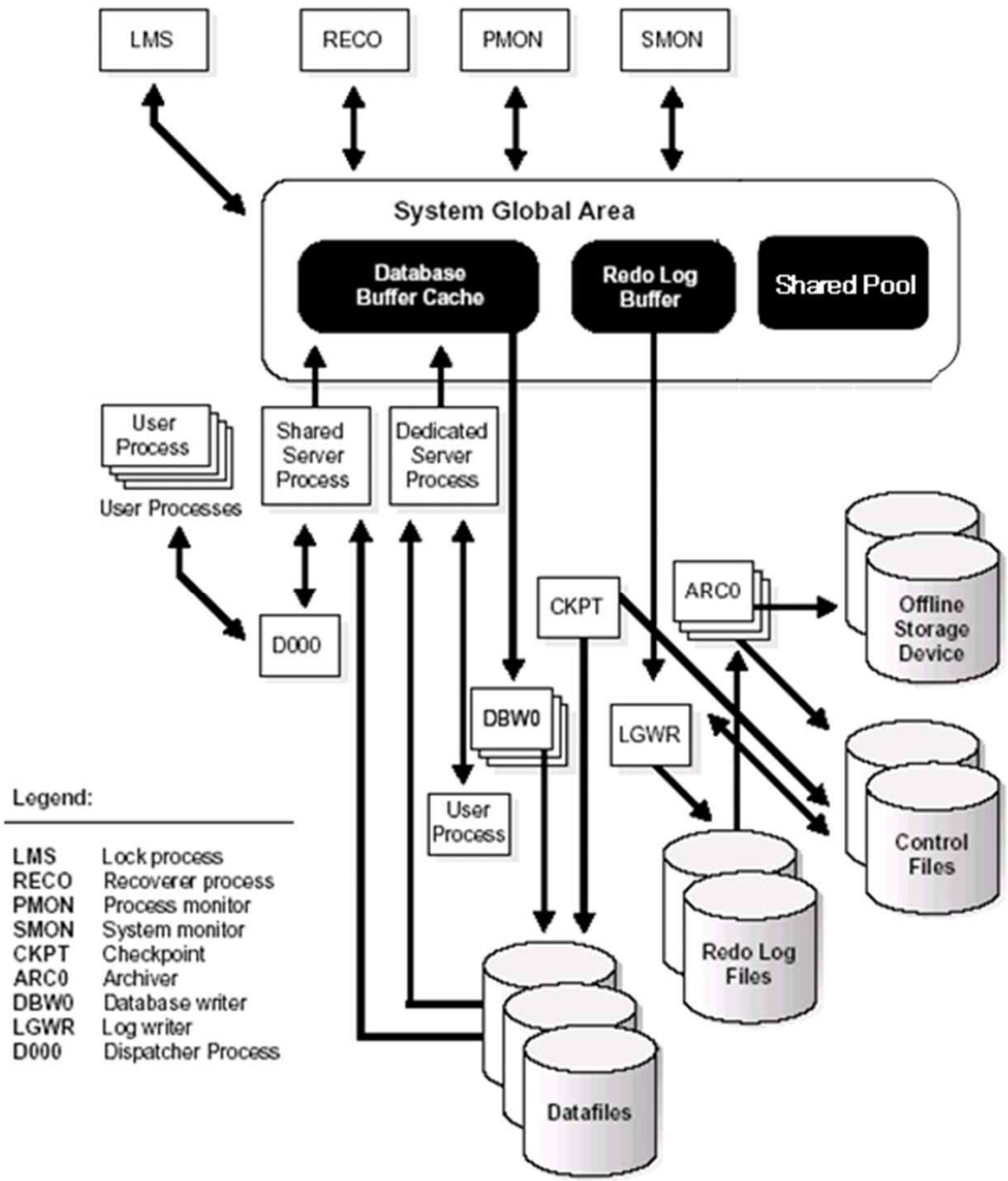
- 처리 조건 기술
  모든 상수 조건은 실제 조건이 수행되는 곳에 명시적으로 분명히 작성한다.
  처리 집합의 순서대로 같이 기술한다.
  동일 집합 처리에서 위에서 아래로 순서대로 기술한다.
  조건절의 '=' 에 따른 열 맞춤 한다.

- 힌트 사용 제한.
  가능한 힌트의 사용을 적게 한다.
  가능한 인덱스 명칭 지정 없이 처리 한다.(인덱스 명명시 고려)
  - Index Supressing 처리 등.

II. Oracle Architecture

1. **Memory & Process**
2. **Server Process**
3. **SELECT**의 이해
4. **DML**의 이해
5. **Database Object Relationships Diagram**

II-1. Memory & Process



Oracle Server =
Instance + Database

Instance =
SGA Memory + Background Processes

Oracle Process =
Background Process + Server Process

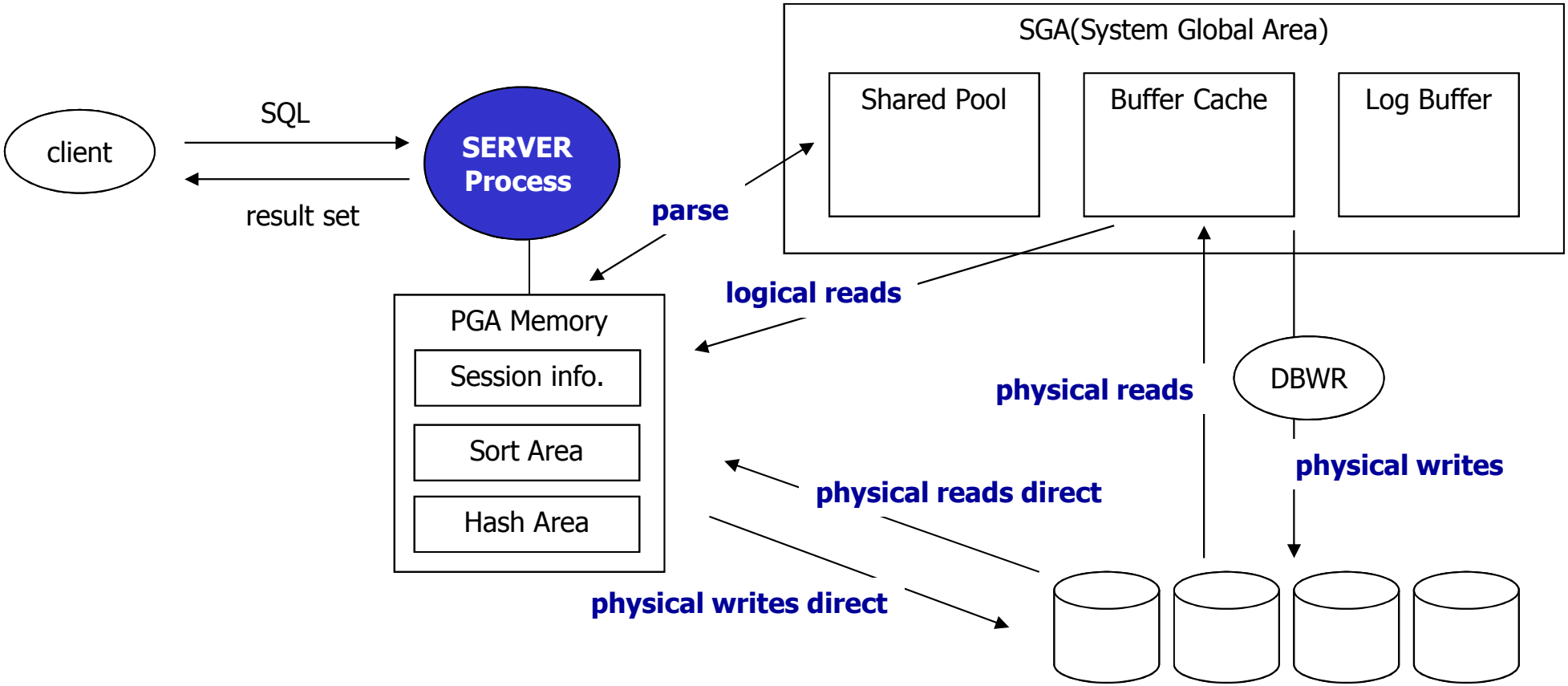
Server Process =
Dedicated or Shared (MTS)

Database =
Datafile + Redolog file + Control file + Parameter file

SGA Memory =
Shared Pool + Buffer Cache +Log buffer

Shared Pool =
Library Cache + Dictionary Cache + ……

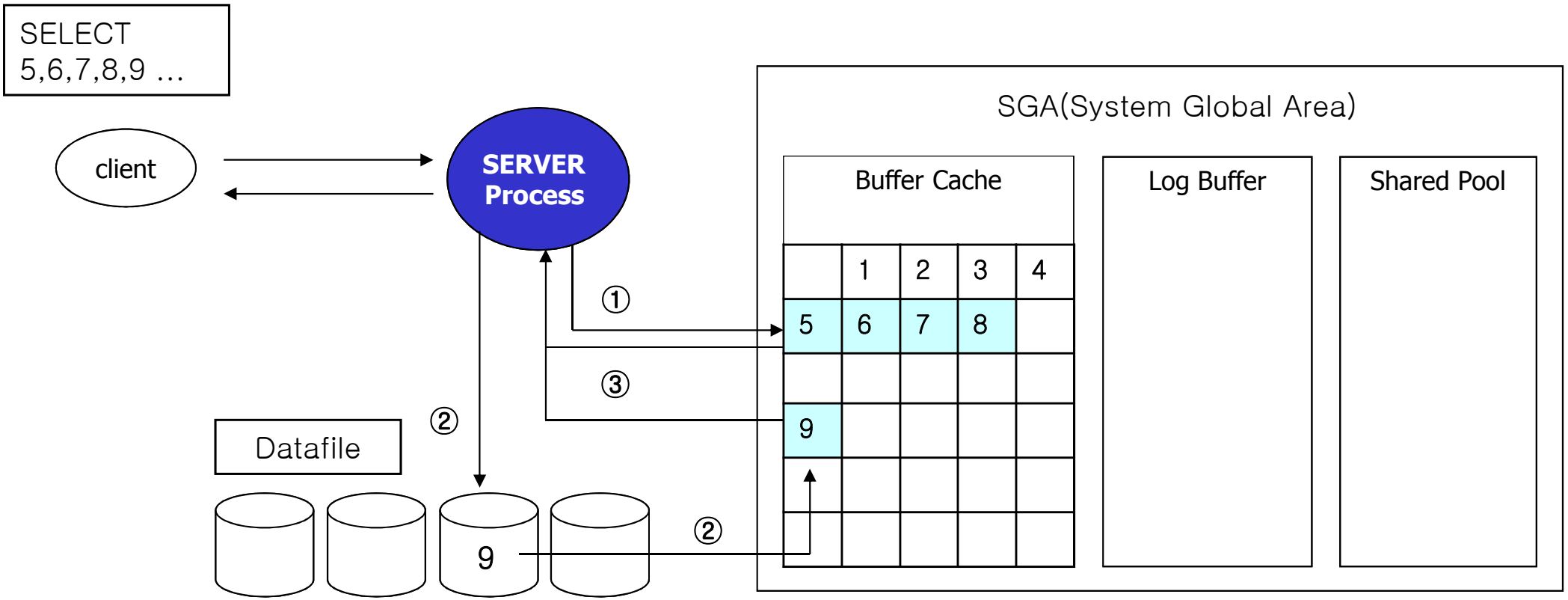
II-2. Server Process



- ❑ 서버 프로세스는 SQL을 접수 받아 파싱, 실행계획 수립하고 실행하는 역할을 수행한다.
- ❑ DB로부터 블록을 읽는 경우 buffer cache를 통해서 읽게 된다.
- ❑ Sorting 작업, Hashing 작업 중에 메모리가 모자라는 경우에는 buffer cache를 거치지 않는다.
- ❑ Shared(MTS) 서버의 경우에는 session info를 SGA 영역에 보관한다.

II-3. SELECT 의 이해

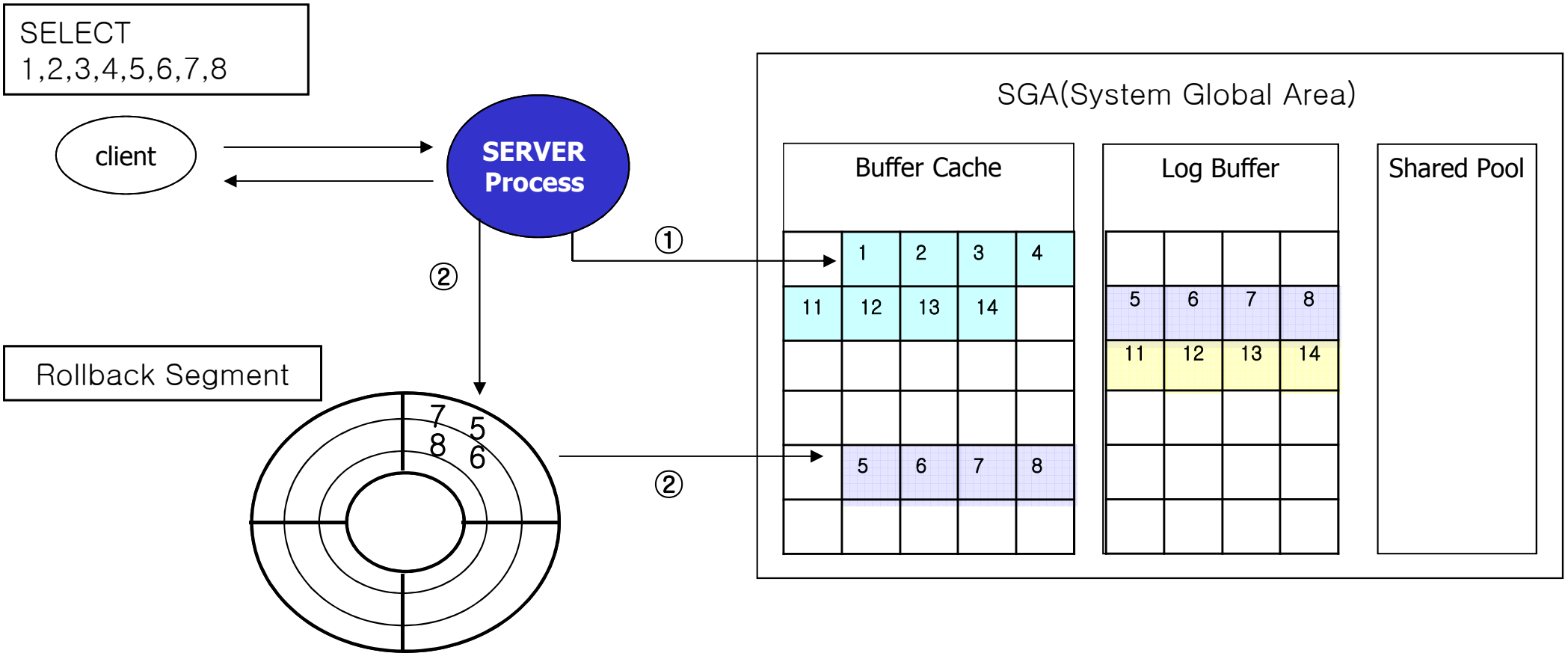
1. 다른 Session에서 DML이 없는 경우



① Buffer Cache 에서 해당 Row를 찾는다.
② 만약 Buffer Cache 에 없으면, Datafile 에서 읽은 후, Buffer Cache 에 Load.
(By LRU Algorithm)
③ Buffer Cache 로부터 읽는다.

II-3. SELECT 의 이해

2. 자신은 DML이 없고, 다른 Session에서 UPDATE후 COMMIT하지 않은 경우
   다른 Session에서 (5,6,7,8) → (11,12,13,14)

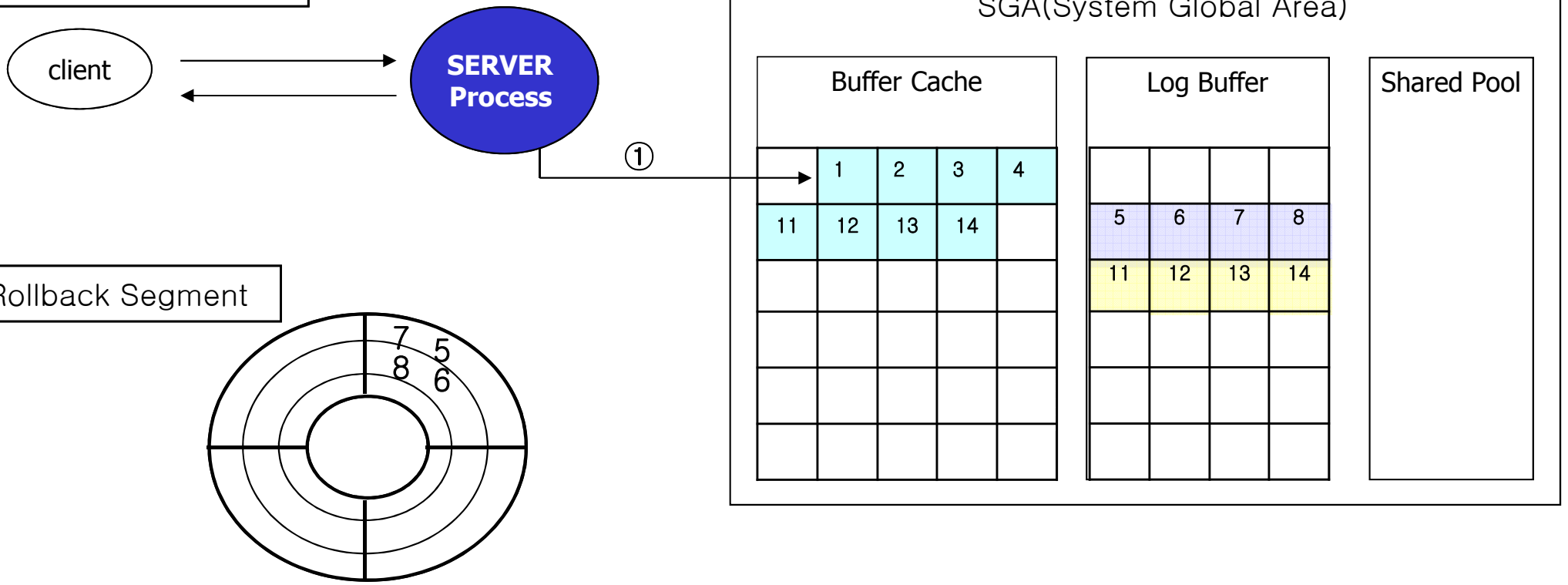


① Buffer Cache 에서 해당 Row를 찾는다.
② 다른 Session 에서 Update 후 Uncommitted 상태라면, Old Image 를 읽기 위해 CR Operation 을 수행한다.
필요하다면 Rollback Block 을 읽고, Block 을 새로운 Version 으로 Copy 한다.

II-3. SELECT 의 이해

3. 자기 Session에서 UPDATE후 COMMIT를 하지 않은 상태에서 SELECT

SELECT
1,2,3,4,11,12,13,14

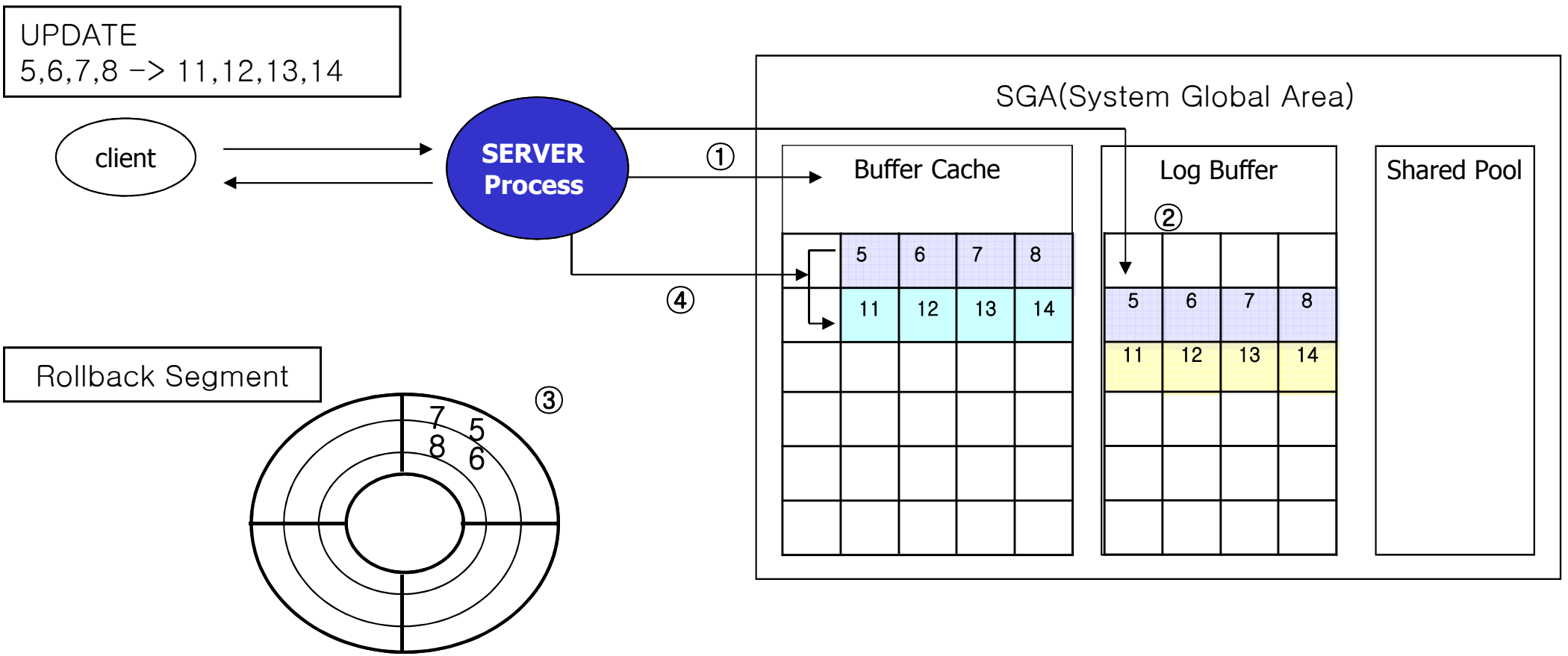


① Buffer Cache 에서 해당 Row를 찾는다.
자기 Session 에서 Update 후 Uncommitted 상태라면, 현재 Block 을 읽는다.
- Commit 이 되지 않았어도 Buffer Cache 의 Block 은 항상 마지막 DML 결과가 현재 Block 이 된다.

II-4. DML 의 이해

1. UPDATE

UPDATE
5,6,7,8 -> 11,12,13,14

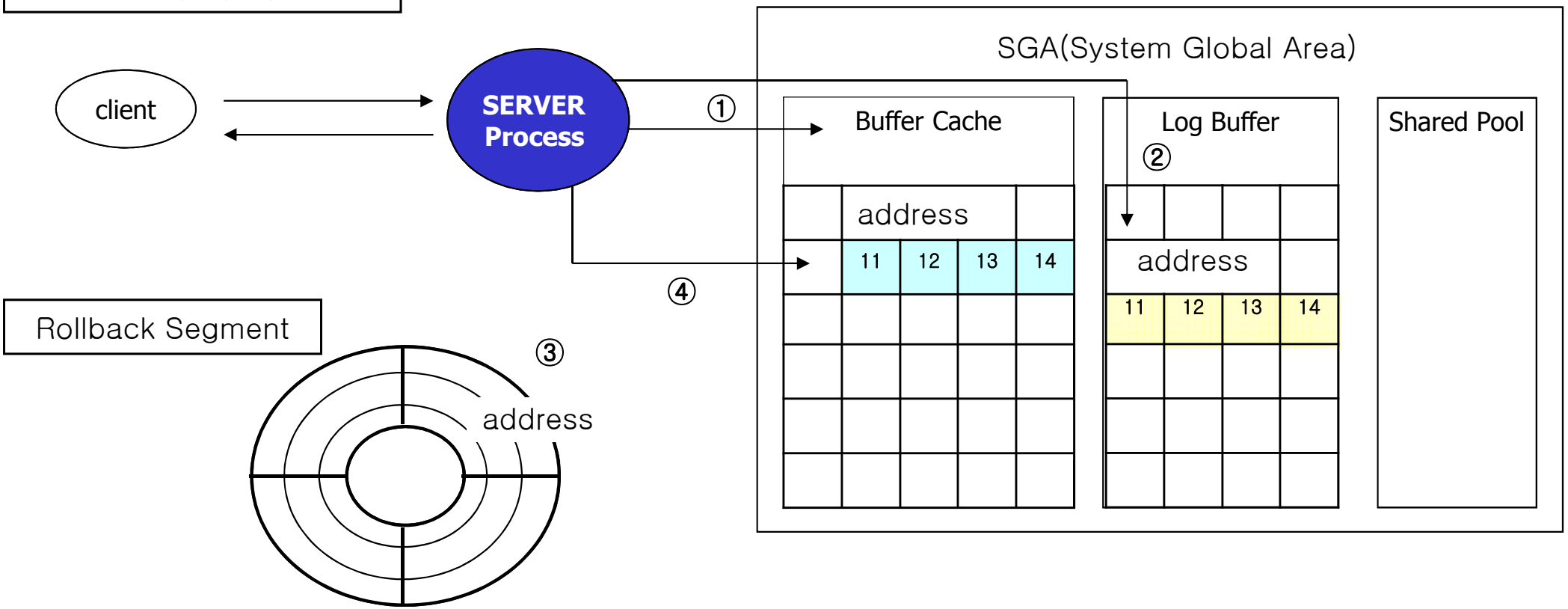


① Buffer Cache 에서 해당 Row를 찾는다. 만약, 없으면 Datafile 에서 읽어서 Buffer Cache 의 Data Block 에 Cache한다 (Cache 후, Row Lock 설정)
② Old Image 와 New Image 를 Redo Log Buffer에 기록 한다.
③ Rollback Segment 의 Undo Block 에 Data block 의 Old Image 를 기록
④ Buffer Cache 의 Data Block 에 New Image를 Update 한다.

II-4. DML 의 이해

2. INSERT

INSERT 11,12,13,14

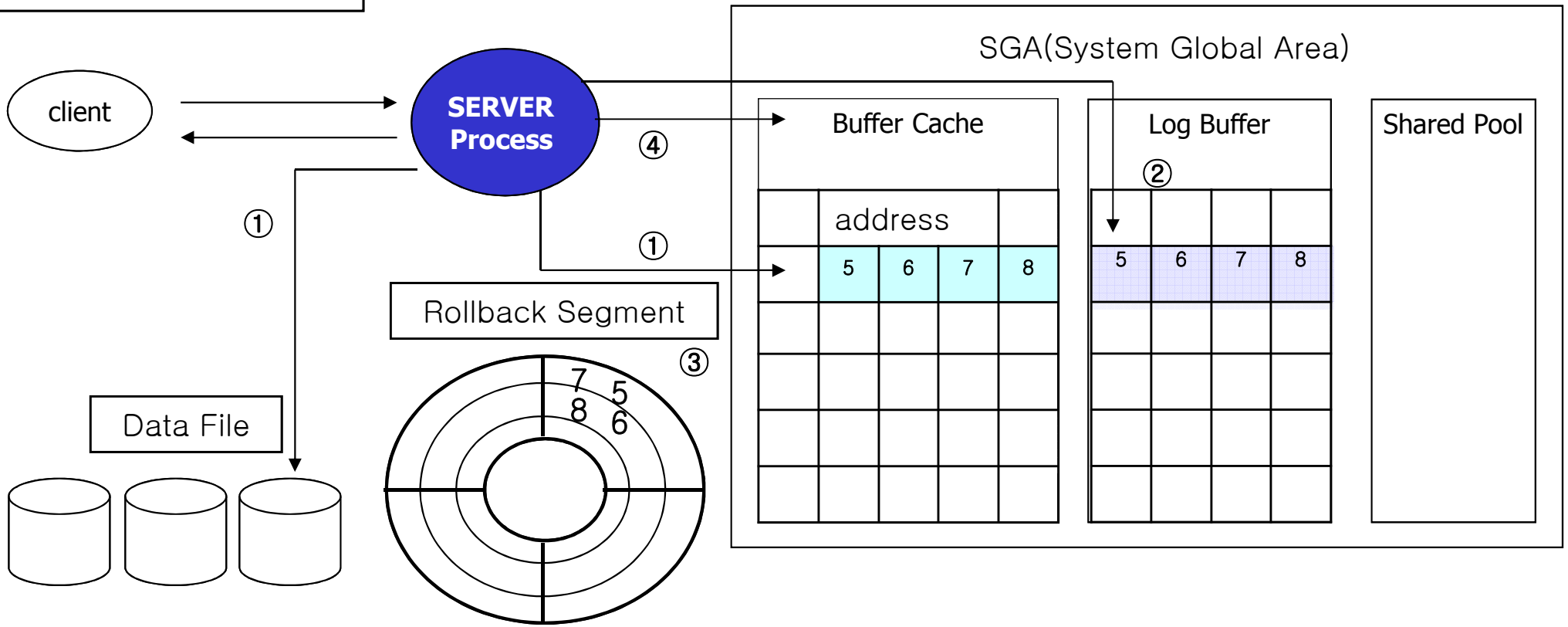


① 해당 Table 의 Free Block 을 Buffer Cache 에 Cache (Cache 후, Row Lock 설정)
② 위치정보와 New Image 를 Redo Log Buffer에 기록 한다.
③ 대상 Row 의 위치정보를 Rollback Block 에 기록
④ Buffer Cache 의 Free Block 에 New Image 기록

II-4. DML 의 이해

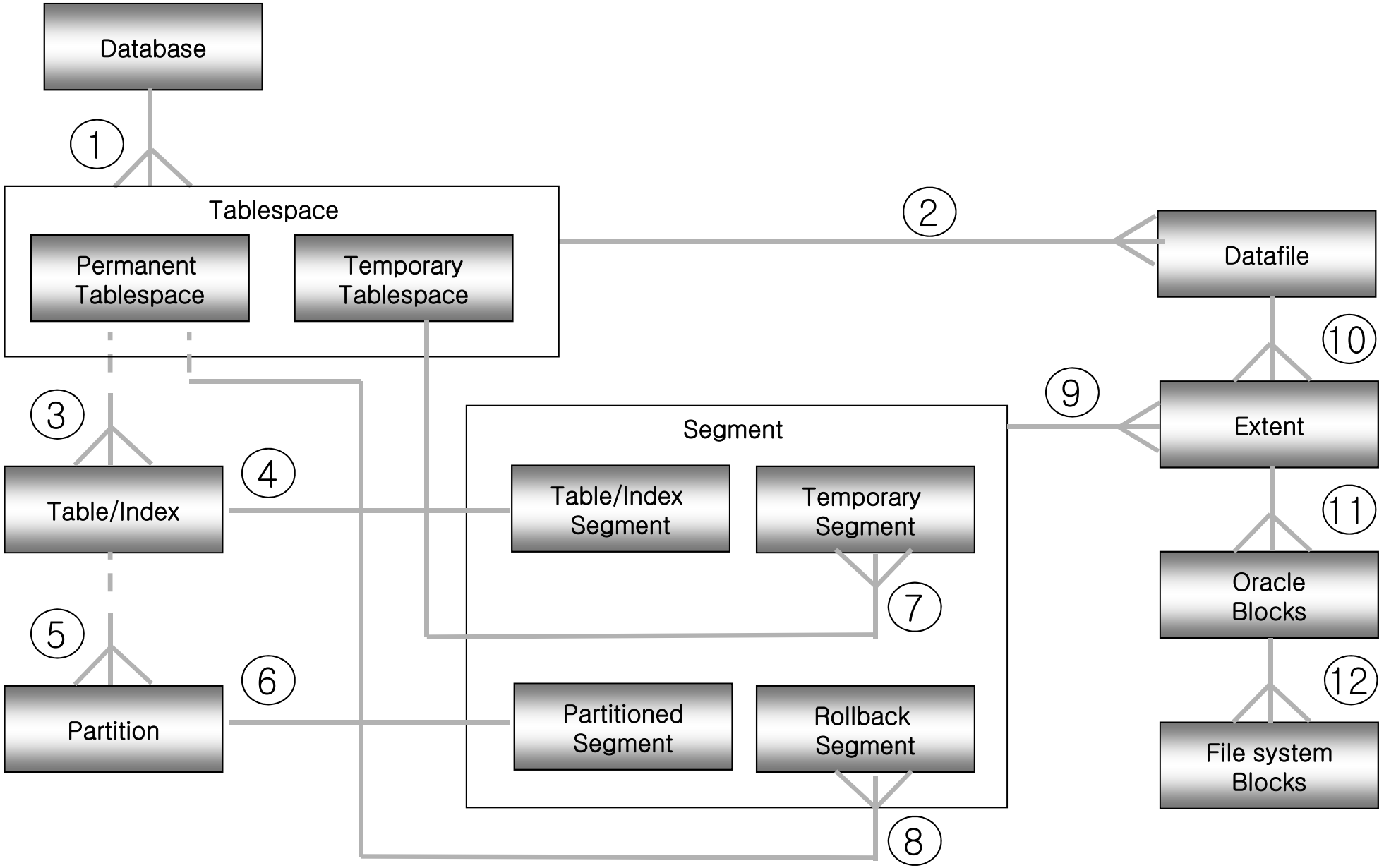
3. DELETE

DELETE 5,6,7,8



① 해당 Block 을 DB File 에서 읽어서 Delete할 대상 Row를 Buffer Cache 에 Load 시킨다
(Cache 후, Row Lock 설정)
② Deleted Row를 Redo Log Buffer 에 기록
③ 대상 Row 를 Rollback Block 에 기록
④ Buffer Cache 의 Block 에서 Delete

II-5. Database Object Relationships Diagram



II-5. Object Relationships Diagram

① 데이터베이스는 한 개 이상의 테이블스페이스(SYSTEM 포함)로 구성되어 있다.

② 테이블스페이스는 한 개 이상의 데이터파일로 구성되어 있다.

③ 테이블스페이스에는 0개 이상의 테이블과 인덱스를 위치시킬 수 있다.
테이블스페이스는 permanent와 temporary로 나뉘어진다.

④ 테이블과 인덱스는 물리적으로는 하나의 세그먼트로 나타내어진다. (Non-partitioned 가정)

⑤ 테이블과 인덱스는 복수 개의 파티션으로 구성될 수 있다.

⑥ 테이블과 인덱스가 파티션되는 경우 각 파티션 당 물리적인 하나의 세그먼트로 나타내어진다.

⑦ TEMP 세그먼트는 temporary tablespace에 위치시켜야 한다.

⑧ ROLLBACK 세그먼트는 permanent tablespace에 위치시킨다. (대부분 별도의 RBS 스페이스)

⑨ 세그먼트는 복수개의 익스텐트로 구성된다.

⑩ 하나의 데이터파일은 여러 개의 익스텐트를 가진다.

⑪ 하나의 익스텐트는 여러 개의 연속된 오라클 블록으로 구성된다.

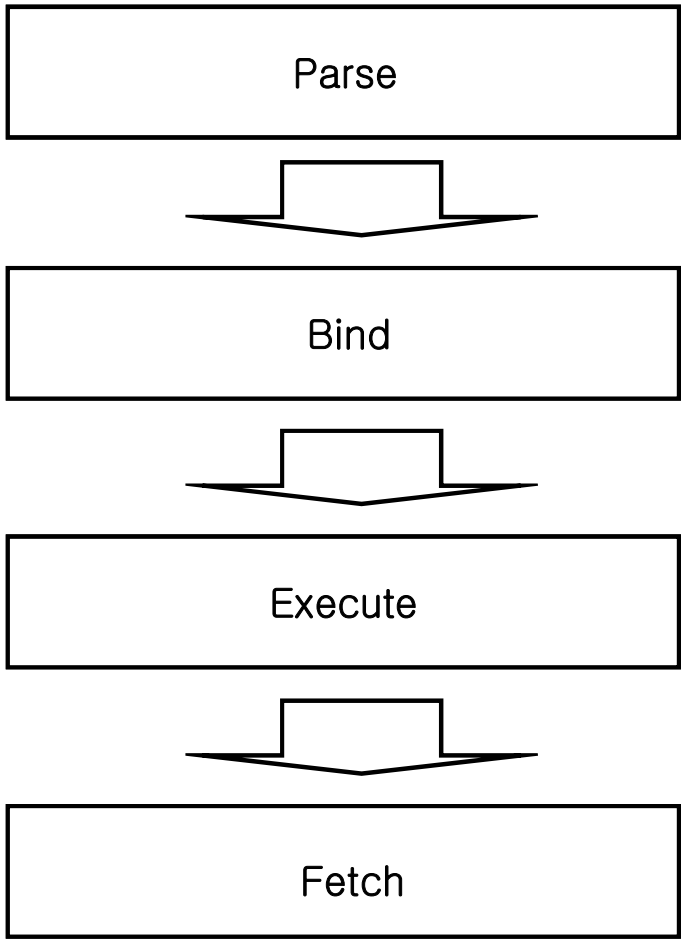
⑫ 하나의 오라클 블록은 여러 개의 파일시스템의 블록으로 구성된다.

III. SQL Execution

1. SQL Execution Steps
2. Optimizer
3. Hint
4. SQL Trace
5. Reading Explain Plan
6. Plan Steps

III-1. SQL Execution steps

User process가 요청한 SQL을 Server process 가 PGA를 통해 수행



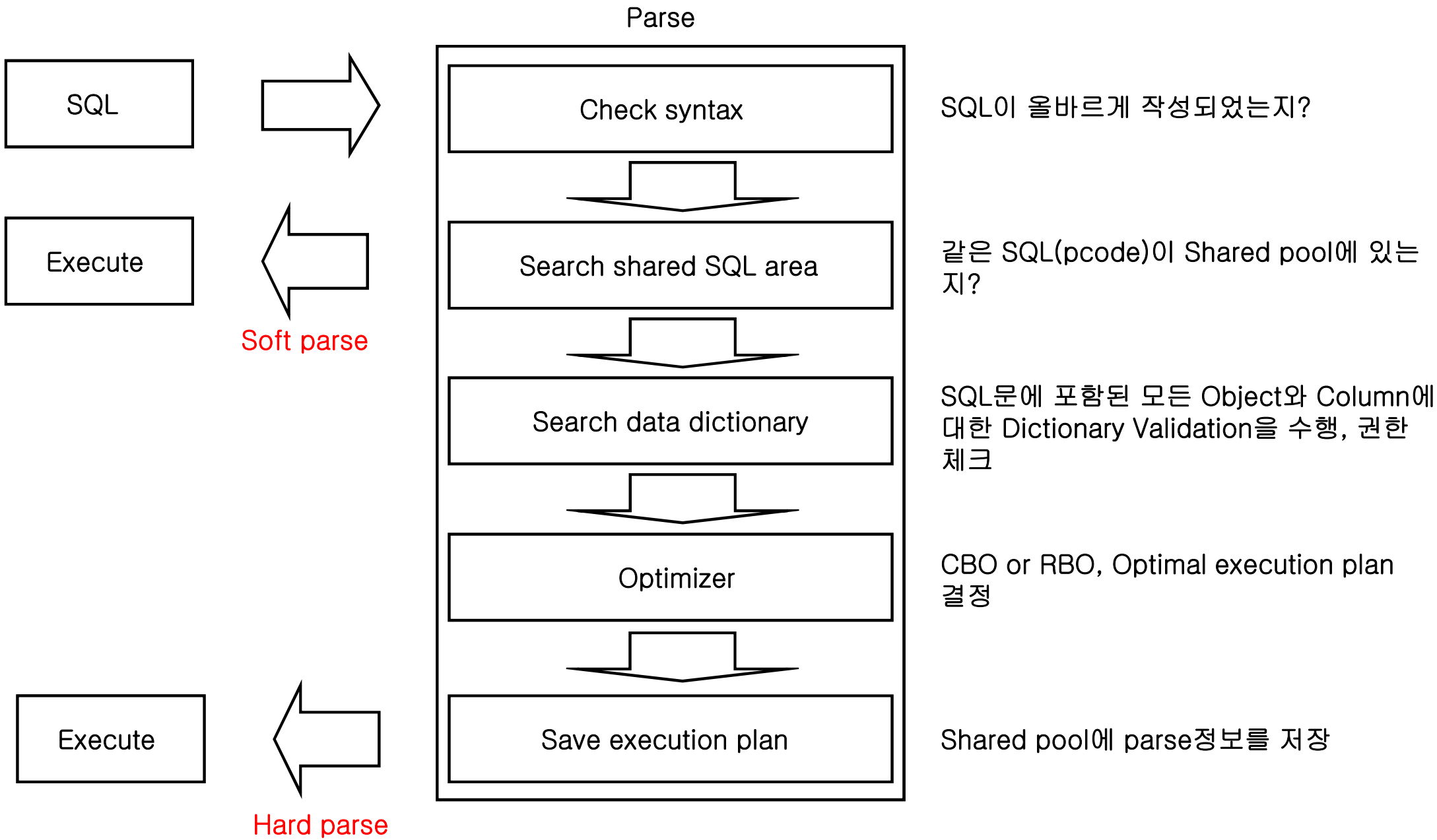
SQL에 대한 구문분석과 Optimize

Bind 변수값이 있는 경우에 할당(Histogram 사용불가)

파싱 과정에서 만든 데이터 조회 경로를 이용하여 원하는 데이터를 버퍼 캐시 내에 위치시키는 작업
DML문에 대한 물리적 읽기 또는 논리적 읽기/쓰기 수행
필요한 경우 데이터에 대한 정렬
INSERT, UPDATE, DELETE 는 이 단계에서 종료

Execute 결과값을 검색하여 반환, array 처리

III-1. SQL Parse



III-2. Optimizer

- Optimizer의 목표는 가장 효율적인 Execution Plan을 찾아내는 일
- 주어진 SQL을 parsing한 결과에 대해서 최소의 일량을 가지는 수행방법을 결정
- Optimizer의 판단시 통계 정보 이용
  - CBO가 주어진 쿼리에 대해서 Memory, I/O 등의 소요량을 계산하기 위한 근거 데이터
  - DBA가 주기적인 Analyze 작업을 통해서 통계정보를 생성
- 수행방법은 Execution Plan(약칭, 플랜)으로 표현됨
- Optimizer의 한계
  - Computing power의 부족
  - Optimizing algorithm의 한계
  - 제한된 시간 내에 방법 결정
  - User data에 대한 이해 부족
  - 실행방법을 결정하는데 있어서 오판이 많음
- Optimizer의 판단에 도움이 되는 Hint를 제공할 필요
- Oracle이 Version Up 이 될수록 Optimizing algorithm 이 향상 되고 있음

III-2. Optimizer

Oracle(Optimizer)이 SQL문을 최적화 하는 방법

- 수식과 조건의 평가
  - 옵티마이져는 수식과 상수를 포함하는 조건을 평가 한다.
    (Syntactic Check And Semantic Check And Simple Transformation)
- 문장 변형
  - 부분질의어를 포함하는 복잡한 문장들을 동등한 조인 문장으로 변형할 수 도 있다.
- 뷰 합성
  - 뷰를 엑세스 하는 경우 뷰에서 표현된 부분을 질의어와 합성한 후 그 결과를 최적화 한다.
- 옵티마이져 모드 선택
  - 규칙 기반, 또는 비용 기반 최적화 방식 중의 하나를 선택한다.
- 접근 경로의 선택
  - 각 테이블에 대하여, 데이터를 얻기 위해서 옵티마이저는 하나 또는 그 이상의 유용한 접근 경로를 선택한다.
- 조인 순서의 선택
  - 두 개이상의 테이블을 조인하는 조인문장에서, 옵티마이저는 어떤 테이블이 먼저 조인되고 어떤 테이블이 결과와 조인될 것인지 등을 선택한다
- 조인 오퍼레이션의 선택
  - 어떤 조인 문장에 대해 옵티마이저는 조인을 수행하기 위해 사용하는 오퍼레이션을 선택한다.

III-2. Optimizer

SQL 최적화 처리 (더 효율적이라고 판단될 경우에만)

❑ 수식과 조건의 평가

- 상수 : 상수 연산은 최초에 한번 수행된다. 조건절의 오른쪽만 단순화 한다.
- LIKE : '%'가 없는 LIKE 연산은 '='연산으로 변경한다.(Varchar 타입만 가능)
- IN : 'IN' 조건은 '=' And OR 조건절로 확장한다.
- ANY & SOME : OR And 비교 연산 조건절로 확장한다. (Any SubQuery => Exists SubQuery)
- ALL : AND And 비교 연산 조건절로 확장한다.
- BETWEEN : '>=' And '<='로 확장한다.
- NOT : NOT을 제거하고 조건 반대 연산자로 확장한다.
- Transtivity : 동등 조건에서 상수조건의 변환이 가능하다.

  A = B And A = '10' → A = B And A = '10' And B = '10'

❑ 문장 변환

- OR 절을 복합 질의어로 : OR 절을 UNION ALL 절로 변경한다.
- 복잡한 문장 최적화 :
  1. 복잡한 서브쿼리를 조인으로 변환 후 최적화 다시 수행.
  2. 복잡한 서브쿼리를 조인으로 변환 불가능 시 개별적으로 최적화 수행.

❑ 뷰 쿼리의 문장 변환 (구문상의 매칭, 의미상의 매칭이 가능한지 여부에 따라)

- 엑세스 쿼리를 뷰 쿼리로 변환
- 뷰 쿼리를 엑세스 쿼리로 변환

III-2. Optimizer

Rule-Based Optimizer(RBO)

- 미리 정해진 15 개의 규칙마다 우선순위(rank)가 설정되어 있고, 적용 가능한 규칙 중에서 우선순위가 가장 높은 규칙을 이용하여 플랜을 작성.(전문가 시스템)
- TABLE FETCH BY INDEX ROWID는 TABLE FULL SCAN보다 우선순위가 높으므로 가능한 인덱스가 있다면 그것을 사용한다. 선택도가 좋지 않은 인덱스의 경우에는 오히려 악영향.
- optimizer_mode = RULE인 상태에서 사용
- Oracle Version 7 이후 새로운 Rule은 추가되지 않은 상태
- 우선 순위로만 실행계획 수립의 비효율.

```
Rows     Row Source Operation
------   ---------------------------
     2    NESTED LOOPS
     1     TABLE ACCESS FULL CM_GIBON
     2     TABLE ACCESS FULL JH_MASTER
```

순위	DESC
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
…	…..
15	Full table scan

III-2. Optimizer

Cost-Based Optimizer(CBO)

- 가능한 플랜들에 대해서 처리비용(cost)를 계산하고, 처리비용이 가장 적은 플랜을 선택
- 처리비용은 테이블과 인덱스, 컬럼 들에 대한 통계정보(statistics)를 참조하여 계산
- optimizer_mode = CHOOSE 인 상태에서 통계정보가 있다면 CBO를 사용
- Selectivity : 많이 걸러낼 수 있는 것을 우선적으로. # of distinct values와 histogram 참조
- Cardinality : 결과건수를 의미. Base 건수와 selectivity를 이용하여 계산
- optimizer_goal = FIRST_ROWS 이면 CBO가 Response time을 목표로 최적화.
  예를 들면, 인덱스 사용 또는 Nested Loops 조인
- optimizer_goal = ALL_ROWS 이면 CBO가 Throughput을 목표로 최적화. Default goal임

CBO Only

- 테이블 및 인덱스의 Partitioning
- Function-based 인덱스
- Parallel Query 및 Parallel DML
- Star Join 및 Star 변형
- 인덱스 skip scan 알고리즘 (9i 이상)
- Reverse key 인덱스
- Materialized View를 이용한 Query rewrite
- Hash Join
- bitmap 인덱스
- bitmap Join 인덱스 (9i이상)

III-2. Optimizer

Cost-Based Optimizer(CBO) New Feature 소개

- Scalar Expression, Analytic Function, Object Type 지원 확장 등.
- ANSI SQL 지원, MATERIALIZED VIEW, FBI, Query Rewrite 기능 확대.
- UPSET, Multi Table Insert 등 새로운 SQL 기능 추가.
- Physical 환경 적용 : 처리 Cost 수립시 CPU, DISK, NETWORK Cost 반영 등.

OPTIMIZER_FEATURES_ENABLE에 따른 확장이력.

```
8.0.4  Index fast full scan
       Ordered nested loop join method
8.0.6  Improved outer join cardinality estimation
8.1.4  No new features
8.1.5  Improved verification of NULLs inclusion in B-tree indexes
8.1.6  Use of FIRST_ROWS or ALL_ROWS mode for user recursive SQL
       Random distribution of left input of nested loop join
       Improved row-length calculation
       Improved method of computing selectivity based on histogram
       Partition pruning based on predicates in a subquery
8.1.7  Common subexpression optimization
       Statistics of a column imbedded in some selected functions such as
         TO_CHAR to compute selectivity
       Improved partition statistics aggregation
9.0.1  Peeking of user-defined bind variables
       Complex view merging
       Push-join predicate
       Consideration of bitmap access paths for tables with only B-tree indexes
       Subquery unnesting
       Index joins
9.0.2  Parallel broadcast in parallel query using joins to small reference or lookup tables
```

III-2. Optimizer

Analyze

ANALYZE TABLE sh.customers ESTIMATE STATISTICS FOR TABLE FOR ALL INDEXES FOR ALL COLUMNS SAMPLE 20 PERCENT;

- Table (USER_TABLES)
  - NUM_ROWS : 로우 개수
  - AVG_ROW_LEN : 로우의 평균 길이
  - BLOCKS : 블록 개수
  - EMPTY_BLOCKS : 한번도 사용된 적이 없는 블록의 개수
- Column (USER_TAB_COLUMNS)
  - NUM_DISTINCT : 서로 다른 값의 개수
  - NUM_NULLS : NULL값을 가지는 로우의 개수
  - HISTOGRAM (USER_HISTORAMS) : 분포도를 표현
- Index (USER_INDEXES)
  - LEAF_BLOCKS : Leaf 블록의 개수
  - BLEVEL : B*Tree의 Non-leaf 블록의 depth
  - CLUSTERING_FACTOR : 테이블 로우의 ordering된 척도, BLOCKS <= cf <= NUM_ROWS
- Partition (USER_TAB_PARTITIONS)
  - Table과 동일한 정보

III-3. READING EXPLAIN PLAN

Understanding Execution plan

EXPLAIN PLAN 은 Oracle optimizer가 선택한 execution plan(실행계획)을 보여줌

- An ordering of the tables referenced by the statement
- An access method for each table mentioned in the statement
- A join method for tables affected by join operations in the statement
- Data operations like filter, sort, or aggregation
- Optimization, such as the cost and cardinality of each operation
- Partitioning, such as the set of accessed partitions
- Parallel execution, such as the distribution method of join inputs

Plan 보는 방법

- 위에서 아래방향으로
- 안에서 밖으로
- 제일 안쪽에서부터. (단계 내에서 밖으로, 위로 진행)
- JOIN은 PAIR로
- JOIN 순서, JOIN 방법, 각 Operation에 따라

III-3. READING EXPLAIN PLAN

Plan 과 조건 절

❑ 인덱스 드라이빙/검색 조건.(Driving/Search Condition)

- 인덱스를 드라이빙/스캔 하는 양을 결정하는 조건.
- Inner 테이블인 경우 성능을 결정할 정도로 중요.
- 선행 컬럼 부터 연속된 조건만 가능.

❑ 인덱스 체크 조건.

- 인덱스 드라이빙/검색 조건 이외의 인덱스 컬럼 절의 조건.
  인덱스의 엑세스 범위를 줄이지는 못하지만 테이블 엑세스량을 줄이는 역할.
- 인덱스 체크 조건은 Table로의 Random 엑세스를 줄여 준다.

❑ 테이블 체크 조건.

- 드라이빙 인덱스 칼럼이 아닌 테이블의 모든 조건절의 상주 조건.

예) Select a.col1, a.col2, a.col3, a.col4, a.col5
From Tab1 a → Index Tab1_idx01 : col1 + col2 + col3
Where a.Col1 = 'aaa' → Index Driving 조건.
And a.col3 = 'ccc' → Index Check 조건.
And a.Col5 = 111 → Table Check 조건.

❑ TABLE FULL SCAN 일 경우.

- 모든 조건절이 테이블 체크 조건이 된다.(파티션은 파티션 키로 드라이빙)

III-3. READING EXPLAIN PLAN

Example 1

```
SELECT * FROM emp
WHERE upper(ename) like 'PARK%'

ENAME_IDX : ENAME

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

Example 2

```
SELECT * FROM emp
WHERE ename like 'PARK%'

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

III-3. READING EXPLAIN PLAN

Example 3

각각 INDEX column이 where의 조건이 미치는 영향은?

```
SELECT saledate, cust_code, description, item_id
FROM    sale
WHERE  saledate = :b1
AND      cust_code LIKE '%-BOM'
AND      NVL(end_date_active,sysdate+1) > SYSDATE ;
```

sale_idx1 INDEX : saledate + cust_code + item_id

Execution Plan

```
----------------------------------------------------
SELECT STATEMENT
TABLE ACCESS BY INDEX ROWID SALE
  INDEX RANGE SCAN SALE_IDEX1
```

III-3. READING EXPLAIN PLAN

Example 4

Execution plan이 실행되는 순서는?

- NEST LOOP JOIN

```
SELECT h.order_number, l.revenue_amount, l.ordered_quantity
FROM    sale h, item l
WHERE  h.saledate = :b1
AND       h.date_ordered > SYSDATE-30
AND       l.item_id = h.item_id ;
```

```
Plan
----------------------------------------------------
SELECT STATEMENT
 NESTED LOOPS                                  ①
   TABLE ACCESS BY INDEX ROWID SALE            ②
     INDEX RANGE SCAN SALE_N1                  ③
   TABLE ACCESS BY INDEX ROWID ITEM            ④
     INDEX RANGE SCAN ITEM_N1                  ⑤
```

III-3. READING EXPLAIN PLAN

Example 5

Execution plan이 실행되는 순서는?

- HASH JOIN

```
SELECT *
FROM    sale a , item b
WHERE  a.item_id = b.item_id
AND      a.saledate = '20020303'
AND      b.unit_price = '1'
```

Execution Plan

```
----------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE
  HASH JOIN                                          ①
    TABLE ACCESS (FULL) OF 'ITEM‘                    ②
    TABLE ACCESS (BY INDEX ROWID) OF 'SALE‘          ③
      INDEX (RANGE SCAN) OF 'PK_SALE' (UNIQUE)       ④
```

III-3. READING EXPLAIN PLAN

Example 6

Execution plan이 실행되는 순서는?

- INLINE VIEW

```
SELECT a.item_id ,b.sale_amt
FROM   item a ,
       (
        SELECT MAX( sale_amt ) sale_amt
        FROM   sale
        WHERE  saledate BETWEEN '20010101' AND '20020101'
        GROUP  BY saledate
       ) b
WHERE  a.unit_price = b.sale_amt
```

Execution Plan

```
----------------------------------------------------
SELECT STATEMENT
  NESTED LOOPS                                    ①
    VIEW                                          ②
      SORT (GROUP BY)                             ③
        TABLE ACCESS (BY INDEX ROWID) OF 'SALE'   ④
          INDEX (RANGE SCAN) OF 'PK_SALE' (UNIQUE)⑤
    TABLE ACCESS (FULL) OF 'ITEM'                 ⑥
```

III-3. READING EXPLAIN PLAN

Example 7

Execution plan이 실행되는 순서는?

- IN SUBQUERY

```
SELECT item_id
FROM   item
WHERE  item_id IN (
                    SELECT item_id
                    FROM   sale
                    WHERE  saledate = '20021212' )
```

Execution Plan

```
----------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE
  NESTED LOOPS                                          ①
    VIEW OF 'VW_NSO_1'                                  ②
      SORT (UNIQUE)                                     ③
        INDEX (RANGE SCAN) OF 'PK_SALE' (UNIQUE)        ④
    INDEX (UNIQUE SCAN) OF 'PK_ITEM' (UNIQUE)           ⑤
```

III-3. READING EXPLAIN PLAN

Example 8

Execution plan이 실행되는 순서는?

- NOT IN SUBQUERY

```
SELECT 'A-2-2 ' ,
          NVL( pstn_brch_cd , ' ' ) ,
          NVL( jung_no , ' ' ) ,
          NVL( jung_seq_no , 0 ) ,
          NVL( firm_sym , ' ' )
FROM    tbjg12
WHERE  sangsil_dt = '99991231'
AND      pstn_brch_cd NOT IN (
                                  SELECT brch_cd
                                  FROM    tbtd10
                                  WHERE  brch_adpt_yn = 'Y'
     )
Execution Plan
----------------------------------------
SELECT STATEMENT - FIRST_ROWS- Cost Estimate:969
 FILTER                                                              ①
  TABLE ACCESS BY GLOBAL INDEX ROWID :JUNG                           ②
   INDEX RANGE SCAN :IX_JUNG_06(NU)(JUNG_SANGSIL_DT)                 ③
  TABLE ACCESS BY INDEX ROWID :ZZT                                   ④
   INDEX RANGE SCAN :PK_ZZT (U) (BRCH_CD,PSTN_TYPE)                  ⑤
```

III-3. READING EXPLAIN PLAN

Example 9

– UPDATE

```
UPDATE rm402 a
SET ( a.mat_cost , a.mat_amt) = (
                                    SELECT  b.in_cost ,
                                            TRUNC( a.gy_wqty + a.gy_jqty)
                                     FROM    rm405 b
                                     WHERE  a.import_num = b.import_num
                                      AND     a.io_date LIKE '200209' ||'%'       )
WHERE  NVL( a.status , 'x' ) <> 'C'
AND    NVL( a.subl_flag , 'x' ) = 'Y'
AND    a.io_date LIKE '200209'||'%'
AND    EXISTS (
           SELECT 'x'
           FROM    rm405 b
           WHERE  a.import_num = b.import_num
           AND       a.io_date LIKE '200209' ||'%')
```

Execution Plan

```
----------------------------------------
UPDATE STATEMENT HINT=CHOOSE
  UPDATE TB_RM402                                    ①
     FILTER                                          ②
       TABLE ACCESS BY INDEX ROWID RM402             ③
           INDEX RANGE SCAN PK_RM402                 ④
        INDEX UNIQUE SCAN PK_RM405                   ⑤
   FILTER                                            ⑥
     TABLE ACCESS BY INDEX ROWID RM405               ⑦
       INDEX UNIQUE SCAN PK_RM405                    ⑧
```

III-3. READING EXPLAIN PLAN

Example 10

Execution plan이 실행되는 순서는?

```
Execution Plan
----------------------------------------
SELECT STATEMENT - CHOOSE
 SORT GROUP BY                                                    ①
  NESTED LOOPS                                                    ②
   VIEW US_SWJP.(1)                                               ③
    UNION-ALL                                                     ④
     SORT GROUP BY                                                ⑤
      NESTED LOOPS                                                ⑥
       TABLE ACCESS FULL :US_SWJP.TB_CM206(3)                     ⑦
       TABLE ACCESS BY INDEX ROWID :US_SWJP.TB_DC530(2)           ⑧
         INDEX RANGE SCAN :US_SWJP.IX_DC530_01(NU)                ⑨
     SORT GROUP BY                                                ⑩
      NESTED LOOPS                                                ⑪
       TABLE ACCESS FULL :US_SWJP.TB_CM206(5)                     ⑫
       TABLE ACCESS BY INDEX ROWID :US_SWJP.TB_DC400(4)           ⑬
         INDEX RANGE SCAN :US_SWJP.PK_DC400 (U)                   ⑭
   TABLE ACCESS BY INDEX ROWID :US_SWJP.TB_CM110(6)               ⑮
     INDEX UNIQUE SCAN :US_SWJP.PK_CM110 (U) (GRAS)               ⑯
```

III-4. PLAN STEPS

CATEGORY	OPERATION	OPTION	DESC
Table Access Paths	TABLE ACCESS	FULL	full-table scan, HWM 까지 읽는다
		HASH	hash key로 table access
		BY INDEX ROWID	index를 통해서 얻은 rowid로 table access
		BY USER ROWID	user가 입력한 bind 변수, 상수값
		BY GLOBAL INDEX ROWID	globally partitioned index
		BY LOCAL INDEX ROWID	locally partitioned index
Index Operations	AND-EQUAL		2개의 다른 index를 결합하여 조회
	INDEX	UNIQUE SCAN	unique한 index에서 한건을 조회
		RANGE SCAN	index에서 rowid를 range로 조회
		FULL SCAN	index의 모든 rows를 순서적으로 조회
		FULL SCAN(MAX/MIN)	index을 이용하여 최대값, 최소값을 조회
		FAST FULL SCAN	index의 모든 rows를 multi-block read 로 조회
Bitmap Operations	BITMAP	CONVERSION	rowid 를 bitmap or bitmap을 rowid로 변환
		INDEX	bitmap index를 이용하여 조회
		MERGE	merge multiple bitmaps
		MINUS	subtract one bitmap from another
		OR	create a bitwise OR of tow bitmaps
Join Operations	CONNECT BY		hierarchical self-join
	ALL JOIN	MERGE JOIN	a merge join performed on the output of the preceding steps
		NESTED LOOPS	맞는 row를 찾기위해 nested loop
		HASH JOIN	a hash join is performed of two row sources
		OUTER	outer join
		ANTI	없는 row를 찾기위한 조인
		SEMI	조건에 만족하는 row만을 찾는 선택적 조인
		CARTESIAN	조인조건이 없는 조인

III-4. PLAN STEPS

CATEGORY	OPERATION	OPTION	DESC
Set operations	CONCATENATION		union, OR statement is used with indexed columns
	INTERSECTION		교집합을 조회할때 사용
	MINUS		집합간의 minus 연산
	UNION-ALL		집합간의 합집합(union, union all)
	VIEW		view or inline view
Miscellaneous	FOR UPDATE		for update 사용시
	FILTER		filtering에 의해 조회(주로 subquery 사용시)
	REMOTE		dblink을 통해 연결될때
	SEQUENCE		oracle sequence 사용시
	INLIST ITERATOR		in 구문사용시
	LOAD AS SELECT		direct path INSERT
	FIXED TABLE		X$ table을 조회시
	FIXED INDEX		X$ table의 index을 이용할때
Partition operations	PARTITION	SINGLE	access a single partition
		ITERATOR	access a mutiple partitions
		ALL	access all partitions
		INLIST	partition key에 in 구문사용시
Aggregation	COUNT		count() function 사용시
	COUNT	STOPKEY	where절에 rownum 사용시
	SORT	ORDER BY	order by 사용시
		AGGREGATE	group function 사용시
		JOIN	sort the rows in preparation for a merge join
		UNIQUE SCAN	distinct, subquery에 in 사용시
		GROUP BY	group by 사용시
		GROUP BY ROLLUP	group by 에 rollup 사용시
		GROUP BY CUBE	group by 에 cube 사용시

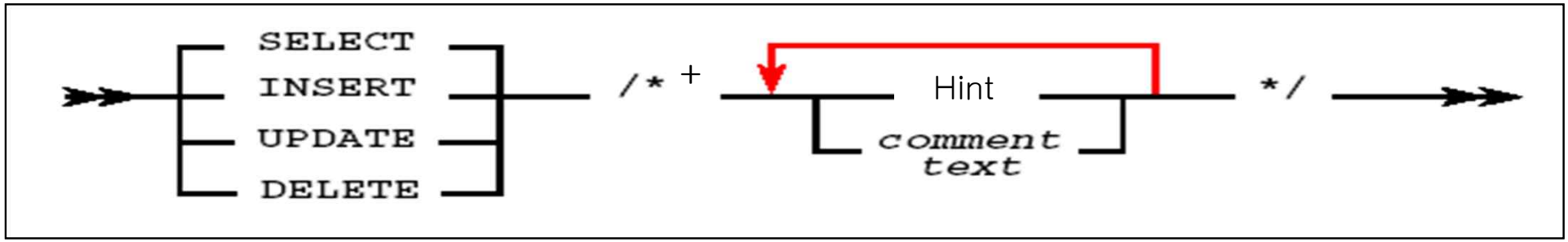
III-5. Hint

Hint의 정의

- Optimizer가 항상 최적의 Execution Plan을 생성하지는 않음.
- CBO가 주어진 쿼리에 대해서 최적의 Plan을 생성하는데 도움을 제공하는 키워드 (가이드이며 명령이 아니다).
- SQL 개발자가 액세스되는 User Data에 대해서 Optimizer보다 더 잘 안다고 가정.
- Hint는 Query의 결과에 영향을 주지 않음.
- 잘못 사용된 Hint는 Optimizer에 의해 무시됨 (문법 오류, Query 결과에 영향을 주는 Hint 등)
- Hint는 RULE, APPEND, CURSOR_SHARING_EXACT 를 제외하고는 항상 CBO 를 호출하며, FIRST_ROWS를 제외하고 모두 ALL_ROWS로 수행.

Hint의 사용 규칙

- SQL 블록의 첫 키워드 바로 뒤에 입력.
- 각 블록에서 첫번째 Hint 주석만 항상 인식, 하나의 Hint 주석은 여러 개의 Hint 포함 가능.
- Hint는 해당 블록에만 적용.
- 문장에 alias를 사용하는 경우 힌트는 그 alias를 참조해야 함.



III-5. Hint

Hint의 파싱

- 힌트는 앞에서 하나의 쿼리 블록에 여러 개 기술 가능, 단 첫번째 힌트만 인식 나머지는 주석처리.
  (주 처리 와 검증시 등의 개별 목적에 따라 사용 가능하다.)
- 힌트절 안에서 힌트 구문 이외는 다 주석으로 인식.
  (구문이 힌트절 인지의 여부는 옵티마이져가 판단 – 철자 오류에 대한 처리 여부)
- 힌트의 종류에 따라 이후 힌트의 파싱 및 적용 범위 및 여부 결정
  - 철자 오류, 구문 오류시 해당 힌트만, 또는 이후의 모든 힌트 무시가 각각 발생 한다.
- 의도적 무시
  - 전체 범위 처리 SQL에서 FIRST_ROWS와 같은 힌트는 무시.
  - 개별 쿼리 블록마다 OPTIMIZER_GOAL이 다른 경우도 무시.
  - 서로 상반되는 성격의 힌트 지정시 무시.

Hint의 종류

- Hints for Optimization Approaches and Goals
- Hints for Access Methods
- Hints for Join Orders
- Hints for Join Operations
- Hints for Parallel Execution

III-5. Hint

Hint Example

```
SELECT /*+ index(e employee_mgr_idx) */ empno....
FROM    employee e

UPDATE /*+ index(e employee_mgr_idx) */ employee e

SELECT /*+ index_desc(e employee_sal_idx) */ empno....
FROM    employee e

SELECT /*+ ordered use_nl(e f) */ empno....
FROM   employee e, department f

SELECT /*+ ordered use_hash(e f) */ empno....
FROM    employee e, department f

SELECT /*+ parallel(e 4) full(e) */ empno....
FROM    employee e

SELECT  empno ...
FROM    employee e
WHERE  empno not in (SELECT /*+ HASH_AJ */ ....
```

III-5. Hint

자주 사용하는 Hints

Hint	USE
/*+ ALL_ROWS */	cost-based optimizer에서 전체 응답시간이 가장 적은 plan 선택(DW)
/*+ FIRST_ROWS */	cost-based optimizer에서 첫번째 row가 가장 빨리 나오는 plan으로 선택(OLTP)
/*+ RULE */	rule-based optimization로 plan 작성
/*+ FULL(table) */	index 유무에 상관없이 full table scan 선택
/*+ HASH_AJ(table) */	NOT IN subquery를 hash antijoin으로 변환
/*+ INDEX(table index) */	특정 table의 특정 index를 순방향으로 사용
/*+ INDEX_DESC(table index) */	특정 table의 특정 index를 역방향으로 사용
/*+ INDEX_FFS(table index) */	index만으로 구성된 sql에서 fast full index scan 사용
/*+ ORDERED */	FROM 절에 나온 순서대로 join 순서 조정
/*+ USE_HASH (table) */	hash join 사용
/*+ USE_NL (table) */	nested-loops join 사용
/*+ APPEND */	INSERT mode에서만 사용되며 기존의 HWM 밑의 free space를 사용하지 않고 HWM위에 append 함
/*+ PARALLEL(table degree) */	table의 parallel degree 지정

III-6. SQL Trace

SQL Autotrace

❑ PLUSTRACE 권한 생성

```
SQL> conn / as sysdba
SQL> @?/sqlplus/admin/plustrce.sql
```

❑ 해당 사용자에게 PLUSTRACE 권한 부여

```
SQL> conn / as sysdba
SQL> grant PLUSTRACE to scott;
```

❑ 해당 사용자에게 PLAN Table 생성

```
SQL> @?/rdbms/admin/utlxplan.sql
```

❑ Autotrace Mode 설정

```
SQL> set autotrace on
SQL> set autot off
SQL> set autotrace traceonly
SQL> set autotrace traceonly explain
SQL> set autotrace traceonly statistics
```

III-6. SQL Trace

SQL Autotrace 예제

```
SQL*Plus > SET AUTOTRACE ON STAT
SQL*Plus > COLUMN PLAN_PLUS_EXP FORMAT A120
SQL*Plus > SET LINESIZE 150
SQL*Plus > select e.ename, d.dname
             from emp e, dept d
            where e.sal > 1000  and e.deptno = d.deptno;

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=3 Card=14 Bytes=294)
   1    0   HASH JOIN (Cost=3 Card=14 Bytes=294)
   2    1     TABLE ACCESS (FULL) OF 'DEPT' (Cost=1 Card=4 Bytes=44)
   3    1     TABLE ACCESS (FULL) OF 'EMP' (Cost=1 Card=14 Bytes=140)
Statistics
----------------------------------------------------------
          0  recursive calls
          8  db block gets
          3  consistent gets
          0  physical reads
          0  redo size
        996  bytes sent via SQL*Net to client
        424  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
         12  rows processed
```

III-6. SQL Trace

SQL Trace 기능

- ❑ 모든 SQL 수행에 대한 Trace 파일 생성
  - 인스턴스 레벨 추적(많은 부하 발생)
  - 각 프로세스별 추적 파일 생성
  - 세션 레벨 추적
- ❑ 구문 분석, 실행 및 인출 단계에 대한 크기 및 시간, 통계 정보
- ❑ 시스템의 전체적인 분석에 필요
- ❑ 관련 초기화 파일
  - USER_DUMP_DEST에 추적 파일 생성 됨
  - TIMED_STATISTICS = true (정확한 시간 통계치)

III-6. SQL Trace

SQL Trace 설정

- init parameter 설정
  - USER_DUMP_DEST = /oracle/admin/ORA9/udump
  - MAX_DUMP_FILE_SIZE = 409600 #200MB
  - TIMED_STATISTICS = TRUE

- 모든 세션에 일괄 설정 (전체 시스템 부하 20~30% 증가되므로 주의!!!)
  - SQL_TRACE = TRUE
  - ALTER SYSTEM SET SQL_TRACE = TRUE;

- 자기 세션을 설정
  - ALTER SESSION SET TIMED_STATISTICS = TRUE;
  - ALTER SESSION SET SQL_TRACE = TRUE;

- 다른 세션을 설정
  - GRANT execute ON sys.dbms_system TO xxx;
    SYS.DBMS_SYSTEM.SET_SQL_TRACE_IN_SESSION(&sid, &serial, TRUE/FALSE);
    SYS.DBMS_LOCK.SLEEP(0.1);
  - SYS.DBMS_SYSTEM.SET_EV(&sid, &serial, 10046, &trace_level, '');
    * 설정을 해도 무시되는 경우가 있으므로 설정이 잘 되었는지 확인 필요.

III-6. SQL Trace

SQL Event Trace 설정

❑ Event Trace 설정

```
SQL> alter session set events '10046 trace name context forever,  level 12' ;
```

❑ 트레이스 LEVEL

- 1 : Call Statistics
- 4 : Call stat + Bind (Loop query시 크기가 급속히 커짐)
- 8 : Call stat + Wait Event (disk I/O가 많을 때 급속히 커짐)
- 12: Call stat + Bind + Wait Event

주의 : Level 4,8,12 시에는 트레이스 파일의 크기가 매우 커지게 되므로 가능한 시스템 전체로 걸지 말고 세션단위로 걸어야 한다.(DB 시스템 전체의 부하가 크며 트레이스 파일 기록에 대한 부하로 성능정보가 왜곡될 수도 있다.)

❑ 트레이스 정보 출력 순서

- PARSING IN CURSOR
- PARSE
- BIND
- EXEC
- FETCH
- STAT
- WAIT,ERROR는 모든 곳에 가능

III-6. SQL Trace

SQL Trace 예제

```
SQL*Plus > alter session set sql_trace=true;

# tkprof ora_09136.trc output.prf sys=no explain=scott/tiger

 SELECT e.ename, d.dname
 FROM   emp e, dept d
 WHERE  e.sal > 1000
 AND    e.deptno = d.deptno

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.01       0.02          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2      0.00       0.00          0          6          0          12
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.01       0.02          0          6          0          12

Misses in library cache during parse: 1
Optimizer goal: ALL_ROWS
Parsing user id: 20  (SCOTT)

Rows     Row Source Operation
-------  ---------------------------------------------------
     12  HASH JOIN
      4   TABLE ACCESS FULL DEPT
     12   TABLE ACCESS FULL EMP
```

III-6. SQL Trace

SQL Trace 일량 판단

- ❑ count : 수행회수
- ❑ Cpu : 내가 소비한 CPU time slice 합계
- ❑ elapsed: CPU + wait(DISK I/O등)
- ❑ Disk : physical read한 Oracle block수
- ❑ query+current: Cache에서 read한 Oracle block 수.
- ❑ rows : 처리/조회된 최종 결과 건수
- ❑ disk/(query+current)*100=cache miss
- ❑ (elapsed-cpu)/disk*1000:disk access time

SQL Trace Rows

- ➢ Trace의 Row Source가 실제 수행시의 실행 계획이다.
- ➢ Trace의 Row 수는 8 이전에는 **처리 로우 수**.
  8i 이후에는 **결과 로우 수** 이다.
- ➢ Trace의 Row 수는 8, 8i의 경우 표현의 변경에 따른 방식의 혼란이 가능 하다.
  (버그 및 표현의 기준 변경에 따른 혼동 – Unique Scan 같은 경우)

III-6. SQL Trace

Oracle 9.2 Event 10046 Segment-level Statistics

- Oracle 9iR2 New Feature로 추가.
- 실행계획의 각 단계별, 개별 처리별 수행 정보(Stat) 표현.
  - 처리 결과만 표현되는 Trace 정보에 대한 보완.
  - 개별 엑세스는 각 단계의 처리 정보를 표시.
  - JOIN 엑세스는 해당 단계까지의 누적 처리 시간을 표시.
- 예)

```
SELECT G.CUST_NO, G.NAME, G.JUMIN_NO, G.SILMYUNG_YMD, M.JUKSU_YMD, M.NOWAMT
  FROM CM_GIBON G, JH_MASTER M
 WHERE G.NAME IN ('노유진')
   AND G.CUST_NO = M.CUST_NO

Rows     Row Source Operation
-------  ---------------------------------------------------
      2  TABLE ACCESS BY INDEX ROWID JH_MASTER (cr=7 r=0 w=0 time=354 us)
      4   NESTED LOOPS  (cr=5 r=0 w=0 time=236 us)
      1    TABLE ACCESS BY INDEX ROWID CM_GIBON (cr=3 r=0 w=0 time=133 us)
      1     INDEX RANGE SCAN CM_GIBON_NAME (cr=2 r=0 w=0 time=89 us)
      2    INDEX RANGE SCAN JH_MASTER_CUST_NO (cr=2 r=0 w=0 time=49 us)
```

- cr= : logical I/O for consistent reads.
- r= : physical reads.
- w= : physical writes.
- tims= : elapsed time and the timing precision (e.g. us Microseconds)

III-6. SQL Trace

TKPROF 사용법.

❑ Oracle Utitlty tkprof 사용방법.

```
Usage: tkprof tracefile outputfile [explain= ] [table= ] [print= ] [insert= ] [sys= ] [sort= ]
  table=schema.tablename   explain 옵션에서 사용할 PLAN_TABLE 위치.
  explain=user/password    오라클에 접속하여 EXPLAIN PLAN을 보여줄때.
  print=integer            SQL 구문을 출력할 LIST 숫자.
  aggregate=yes|no         동일한 SQL을 하나로 통합할지 개별 SQL을 표현할지 여부.(기본은 yes)
  insert=filename          SQL 구문과 수행 통계를 TABLE에 Insert 구문으로 만들경우 파일명 지정.
  sys=no                   SYS 사용자로 수행된(내부 처리를 위해 사용된 Recursive) SQL을 포함
                             할지 여부(기본은 yes)
  record=filename          Record non-recursive statements found in the trace file.
  waits=yes|no             Record summary for any wait events found in the trace file.
  sort=option              trace 파일을 sort 옵션에 지정한 값의 순서대로 정렬하기 위한 순서 지정.
```

예) tkprof ora_SID_11234.trc plan_test.txt sys=no sort=exeela, fchela, prscpu explain=SCOTT/TIGER

❑ 기타 용법.

- trace 파일들을 머지하여 하나의 trc 파일로 통합 후 tkprof 가능.
- 개별 수행을 보고 싶을 때는 aggregate 옵션 사용.
- 특정 이슈별로 튜닝 대상 선정 등에는 sort 옵션으로 대상 정렬.(수행시간, 수행 건수, 처리 블록 수 등등)

III-6. SQL Trace

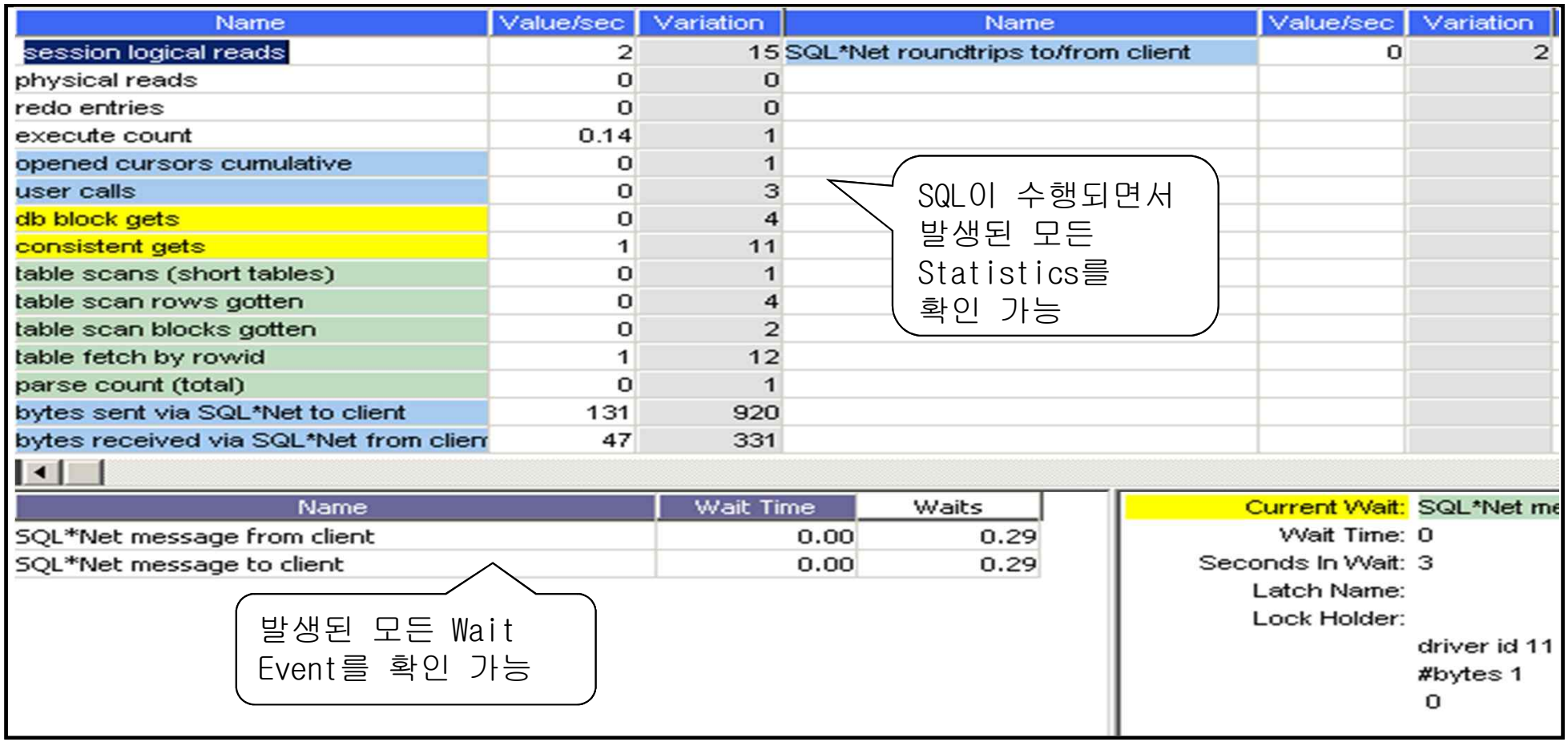
SQL Trace 결과분석

CASE	DESC
Elapsed / 100rows > 1	100 rows를 처리하는데 보통 1초 이내 소요
Elapsed >> cpu	reponse time = service time + wait time 일때 wait time 이 많음
(query+current)/rows >= 5 * 테이블 갯수	넓은 범위의 index range scan
Plan.rows가 튀는 부분	Plan이 비효율일 가망성이 많음
Parse > 1	동일한 SQL문이 여러 번 실행된 결과를 합한 통계치
Parse=Execute=Fetch	수행 시 rows를 한번에 처리하여 인출
Parse=1, Execute=Fetch=10	루프내에서 동일한 SQL 문장이 10번 반복 수행
Parse=Execute=1, Fetch=10	단일 SQL 문장 수행에서 10번의 인출 발생
Fetch=10, Rows=200	20건씩 Array Processing
CPU와 Elapsed간 과도한 차이	과도한 트랜잭션에 의한 병목, 대량 데이터 처리(I/O 부하)
Rows 건수에 비해 과도한 CPU	실행 경로 설정이 잘못(SQL 튜닝)
과도한 disk, query, current	버퍼 캐쉬의 적중률이 낮다.

III-6. SQL Trace

SQL Trace 일량 판단

STEP1. SQL*Plus로 접속한다.
STEP2. BizMax의 SessionManager 화면 상에서 접속한 SQL*Plus 세션을 찾는다.
STEP3. 해당 세션의 SessionDetail로 이동하여 Delta 탭을 띄워놓는다.
STEP4. SQL*Plus 상에서 SQL을 수행한다.
STEP5. SQL의 수행이 끝난 후에 Delta 탭에서 refresh 버튼을 누른다.



Name	Value/sec	Variation	Name	Value/sec	Variation
session logical reads	2	15	SQL*Net roundtrips to/from client	0	2
physical reads	0	0			
redo entries	0	0			
execute count	0.14	1			
opened cursors cumulative	0	1			
user calls	0	3			
db block gets	0	4			
consistent gets	1	11			
table scans (short tables)	0	1			
table scan rows gotten	0	4			
table scan blocks gotten	0	2			
table fetch by rowid	1	12			
parse count (total)	0	1			
bytes sent via SQL*Net to client	131	920			
bytes received via SQL*Net from clien	47	331			

SQL이 수행되면서 발생된 모든 Statistics를 확인 가능

Name	Wait Time	Waits
SQL*Net message from client	0.00	0.29
SQL*Net message to client	0.00	0.29

발생된 모든 Wait Event를 확인 가능

Current Wait:	SQL*Net me
Wait Time:	0
Seconds In Wait:	3
Latch Name:	
Lock Holder:	
	driver id 11
	#bytes 1
	0

III-7. Parameter Tuning

- 파라메터 튜닝 종류
  - 서버 수행 환경
  - 옵티마이져 기능
  - 특정 기능 튜닝
- 서버 수행 환경
  - SORT_AREA_SIZE, SORT_RETAINED_SIZE, HASH_AREA_SIZE
  - DB_FILE_MULTIBLOCK_READ_COUNT
- 옵티마이져 기능.
- 특정 기능 튜닝 – Parallel.
- 특정 기능 튜닝 – RAC
- 특정 기능 튜닝 – SQL*Net

III-7. Parameter Tuning

❑ Oracle 9iR2 CBO 실행계획 수립 사용 파라메터

```
**************************************
PARAMETERS USED BY THE OPTIMIZER
********************************
OPTIMIZER_FEATURES_ENABLE = 9.2.0
ALWAYS_ANTI_JOIN = CHOOSE
ALWAYS_SEMI_JOIN = CHOOSE
DB_FILE_MULTIBLOCK_READ_COUNT = 16
HASH_AREA_SIZE = 1048576
HASH_JOIN_ENABLED = TRUE
HASH_MULTIBLOCK_IO_COUNT = 0
OPTIMIZER_DYNAMIC_SAMPLING = 1
OPTIMIZER_INDEX_CACHING = 0
OPTIMIZER_INDEX_COST_ADJ = 100
OPTIMIZER_MAX_PERMUTATIONS = 2000
OPTIMIZER_MODE/GOAL = All_Rows
OPTIMIZER_SEARCH_LIMIT = 5
PARALLEL_BROADCAST_ENABLED = TRUE
PARTITION_VIEW_ENABLED = TRUE
QUERY_REWRITE_ENABLED = TRUE
QUERY_REWRITE_EXPRESSION = TRUE
QUERY_REWRITE_INTEGRITY = ENFORCED
SORT_AREA_SIZE = 524288
STAR_TRANSFORMATION_ENABLED = TRUE
_ALWAYS_STAR_TRANSFORMATION = FALSE
_B_TREE_BITMAP_PLANS = TRUE
_COMPLEX_VIEW_MERGING = TRUE
_CPU_TO_IO = 0
_DEFAULT_NON_EQUALITY_SEL_CHECK = TRUE
_ENABLE_TYPE_DEP_SELECTIVITY = TRUE
_FAST_FULL_SCAN_ENABLED = TRUE
_GSETS_ALWAYS_USE_TEMPTABLES = FALSE
_GS_ANTI_SEMI_JOIN_ALLOWED = TRUE
_IMPROVED_OUTERJOIN_CARD = TRUE
_IMPROVED_ROW_LENGTH_ENABLED = TRUE
_INDEX_JOIN_ENABLED = TRUE
_LIKE_WITH_BIND_AS_EQUALITY = FALSE
_NESTED_LOOP_FUDGE = 100
_NEW_INITIAL_JOIN_ORDERS = TRUE
_NEW_SORT_COST_ESTIMATE = TRUE
_NO_OR_EXPANSION = FALSE
_ONESIDE_COLSTAT_FOR_EQUIJOINS = TRUE
_OPTIMIZER_ADJUST_FOR_NULLS = TRUE
_OPTIMIZER_CHOOSE_PERMUTATION = 0
_OPTIMIZER_COST_MODEL = CHOOSE
_OPTIMIZER_DYN_SMP_BLKS = 32
_OPTIMIZER_MODE_FORCE = TRUE
_OPTIMIZER_PERCENT_PARALLEL = 101
_OPTIMIZER_UNDO_CHANGES = FALSE
_OPTIM_ENHANCE_NNULL_DETECTION = TRUE
_ORDERED_NESTED_LOOP = TRUE
_OR_EXPAND_NVL_PREDICATE = TRUE
_PRED_MOVE_AROUND = TRUE
_PUSH_JOIN_PREDICATE = TRUE
_PUSH_JOIN_UNION_VIEW = TRUE
_QUERY_COST_REWRITE = TRUE
_SORTMERGE_INEQUALITY_JOIN_OFF = FALSE
_SORT_ELIMINATION_COST_RATIO = 0
_SUBQUERY_PRUNING_COST_FACTOR = 20
_SUBQUERY_PRUNING_ENABLED = TRUE
_SUBQUERY_PRUNING_REDUCTION_FACTOR = 50
_SYSTEM_INDEX_CACHING = 0
_TABLE_SCAN_COST_PLUS_ONE = TRUE
_UNNEST_SUBQUERY = TRUE
_USE_COLUMN_STATS_FOR_FUNCTION = TRUE
_USE_NOSEGMENT_INDEXES = FALSE
**************************************
```

III-6. SQL Trace

Trace 비효율 분석예.

❑ Trace File의 Rows와 실체 처리량의 관계 확인.

❑ BIG_OBJECTS_IDX11 : OBJECT_TYPE + OWNER + OBJECT_NAME

```
************************************************
SELECT COUNT(*)
  FROM BIG_OBJECTS
 WHERE OBJECT_TYPE = 'SYNONYM'
   -- AND OWNER       = 'PUBLIC'
   AND OBJECT_NAME LIKE 'DBA_%'
   AND OBJECT_ID   > 29000

Rows     Row Source Operation
-------  ---------------------------------------
      1  SORT AGGREGATE
     36   TABLE ACCESS BY INDEX ROWID BIG_OBJECTS
    379    INDEX RANGE SCAN BIG_OBJECTS_IDX11
************************************************

→ OBJECT_TYPE = 'SYNONYM'
  11560   INDEX FAST FULL SCAN BIG_OBJECTS_IDX11

→ OBJECT_TYPE = 'SYNONYM' AND OWNER       = 'PUBLIC'
  11540   INDEX FAST FULL SCAN BIG_OBJECTS_IDX11

→ OBJECT_TYPE = 'SYNONYM‘ AND OWNER       = 'PUBLIC' AND OBJECT_NAME   LIKE 'DBA_%'
    379   INDEX RANGE SCAN BIG_OBJECTS_IDX11
```

❑ 인덱스 드라이빙 조건, 처리 건수?
  –

❑ 인덱스 체크 조건, 처리 건수?
  –

❑ 테이블 체크 조건, 처리 건수?
  –

❑ 각 조건 체크 방법은?
  –

IV. Table Access

1. Index Planning
2. B* Tree Index
3. Index Access
4. Bitmap Index/ Function Base Index
5. Index Suppressing
6. Table Full Scan

IV-1. INDEX PLANNING

1. 인덱스의 역할

- ❑ 튜닝의 가장 중요한 Point (적절한 인덱스 선택, 인덱스 변경, 생성, 삭제...)
- ❑ Oracle Optimizer가 Plan을 만드는 가장 중요한 판단 기준 (통계정보, algorithm...)
- ❑ 튜닝시 가장 Cost가 적게 드는 방법(소량 처리시)

2. 인덱스의 의미

- ❑ 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
  (TABLE의 경우는 입력순서대로 DATA를 저장하므로 비 순차적인 BLOCK구성)
- ❑ ROWID를 통한 액세스가 가장 빠른 조회 경로
- ❑ 블록 단위의 액세스 : 가장 작은 단위 디스크 I/O
- ❑ NULL과 인덱스
  - 인덱스 구성 컬럼 전체 값이 NULL이면 인덱스에 미 포함.(즉 일부 NULL은 포함)
    즉 NOT NULL 컬럼이 인덱스 컬럼이면 그 인덱스는 테이블의 모든 ROW를 가르킨다.
    단 Bitmap Index는 정합성 보장을 위해 NULL 값도 포함한다.
  - 인덱스 에서 NULL 값은 자기 단계의 가장 끝에 위치.
- ❑ PK, UK와 같은 정합성 보장의 의미.(FBI로 부분처리도 가능)

IV-1. INDEX PLANNING

3. 인덱스의 사용여부

❑ 인덱스를 사용할 수 있는 조건 : equal(=), in, like, between,<= , < , > , >= , not between, is null

❑ 인덱스를 사용할 수 없는 조건 : 부정형 표현, not equal(!=), not like, is not null, index supressing, 컬럼간 비교, 자기 컬럼 참조,

(예외사항 : is not null의 경우 Optimizer의 판단으로 HISTOGRAM을 이용하여 인덱스 사용가능, COL1+COL2 결합인덱스에서 COL1 IS NULL AND COL2 = 'abc' 조건은 사용 가능-CBO)

```
SELECT * FROM emp WHERE ename like '%JACK%';

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

SELECT * FROM emp WHERE ename like 'JACK%';

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

IV-1. INDEX PLANNING

4. 인덱스의 활용

- ❑ 프로그램에서 index Suppressing이 발생하지 않는지 점검.
- ❑ 자주 access 되는 테이블을 대상으로 모든 쿼리에 대해서 액세스 경로 조사표를 작성 및 유지.
- ❑ 중복되는 인덱스가 있는지 검사 : COL1 인덱스와 COL1 + COL2 인덱스가 동시에 있을 때.
- ❑ 각 컬럼들에 대해 Number of Distinct Values가 어느 정도인지 검사할 필요.
- ❑ 분포도가 어떻게 구성되어 있는지 검사할 필요 (히스토그램 생성)
- ❑ Range Scan하는 쿼리들이 어느 정도의 넓은 범위로 액세스하는 지 검사할 필요.
- ❑ Binding SQL를 사용하는 경우 히스토그램 정보를 사용할 수 없으므로, 실제 어떤 값들이 Binding이 되는지 조사
- ❑ 자주 access되는 짧은 쿼리에서 인덱스만으로 처리할 수 있는지 조사.

5. 결합 인덱스의 구성

- ❑ 결합 인덱스는 구성 컬럼 전체를 하나의 값으로 인식하여야 한다.
- ❑ 자주 같이 Access되는 Column으로 구성.
- ❑ 항상 (=, in) 로 들어오고, Selectivity가 높은 column을 선두 컬럼으로 구성
- ❑ Selectivity가 떨어지더라도 Table access되는 범위를 줄일 수 있다면 결합 index에 추가
- ❑ 대량의 Table를 access할때 일반적으로 단일 index보다 selectivity 유리
- ❑ 결합 index 만으로 결과를 조회할 수 있다면 유리

IV-1. INDEX PLANNING

6. OLTP SYSEM에서 인덱스가 필요한 경우

- ❑ Nested Loop Join에서 조인조건에 사용되는 컬럼들은 인덱스가 있어야 한다.
- ❑ Sub Query의 경우 대부분 특별한 Hint를 주지 않는한 Filter, NL Join처럼 풀리므로 연결조건에 인덱스가 반드시 필요하다.
- ❑ Primary Key, Unique Constraint에는 오라클이 내부적으로 인덱스를 생성하나, FK Constraint 에는 오라클이 내부적으로 인덱스를 생성하지 않으므로 꼭 인덱스를 생성해야 한다.
- ❑ Hierarchical구조에서 START WITH 와 CONNECT BY에는 양쪽 모두 인덱스가 있어야 한다.

7. 인덱스 사용시 문제점

- ❑ 과다한 Disk Block Access
  - 동일 값이 많은 인덱스
  - Range Scan의 범위가 너무 넓은 경우
  - 건수가 너무 적은 테이블(조인을 위해서 추가 가능)
- ❑ 인덱스 관리 비용 부담
  - DML 작업 시 인덱스 정보도 같이 갱신
  - BitMap 이나 IOT의 경우 특히 심함

IV-1. INDEX PLANNING

8. 통계정보를 이용한 올바른 인덱스의 선택

SCOTT.SALE

Table Name: SCOTT.SALE

Num Rows: 309,200 | Empty Blocks: 66 | Avg Row Len: 62 | Init Trans: 1 | Freelists: 1 | Degree: 1

Chain Cnt: 0 | Blocks: 2,749 | Avg Space: 847 | Sample Size: 309,200 | Freelist Groups: 1 | Cache: N

Index Name	Uni-que	Clustering Factor	Leaf Block	Distinct Keys	B Level	# of Rows	Leaf Blks / Key	Data Blks / Key	De-gree	Init Trans	Free Lists	Sample Size
SALE_ITEM_ID_SALEDATE_IDX	NU	29,245	1,027	15,460	2	309,200	1	1	1	2	1	309,200
ITEM_ID,SALEDATE												
BIT_A1	NU								1	2	1	
SALE_CLASS												
BIT_A2	NU								1	2	1	
SALE_PRICE												
PK_SALE	U	2,748	1,287	309,200	2	309,200	1	1	1	2	1	309,200
SALEDATE,CUST_CODE,ITEM_ID												
SALE_CUST_SALEDATE_IDX	NU	33,376	1,167	30,920	2	309,200	1	1	1	2	1	309,200
CUST_CODE,SALEDATE												

Column Name	Data Type	Density	Null	Num Nulls	Num Distinct	Num Buckets	Low Value	High Value	Sample Size	Last Analyzed	Data Default
SALEDATE	CHAR(8)	0.00064	N	0	1546	1	19990101	20030320	309200	2003-03-26	
CUST_CODE	CHAR(6)	0.00161	N	0	618	1	C30023	C30952	309200	2003-03-26	
ITEM_ID	NUMBER(22,7/0)	0.00324	N	0	308	1	13	462	309200	2003-03-26	
SALE_AMT	NUMBER(22)	0.02	Y	0	50	1	1	50	309200	2003-03-26	
SALE_CLASS	CHAR(1)	0.5	Y	0	2	1	R	S	309200	2003-03-26	
SALE_PRICE	NUMBER(22)	0.00094	Y	0	1055	1	1800	525000	309200	2003-03-26	
REGION	VARCHAR2(10)	0.2	Y	0	5	1			309200	2003-03-26	
ATTRIBUTE1	VARCHAR2(2000)	0.00054	Y	6,143	1824	1	A	yyyyyyy	309200	2003-03-26	

Num Rows

- table의 건수

Chain Cnt

- data chain count

Clustering Factor

- data의 정렬 여부

Leaf Block

- leaf node의 block 수

Distinct keys

- distinct한 row의 수

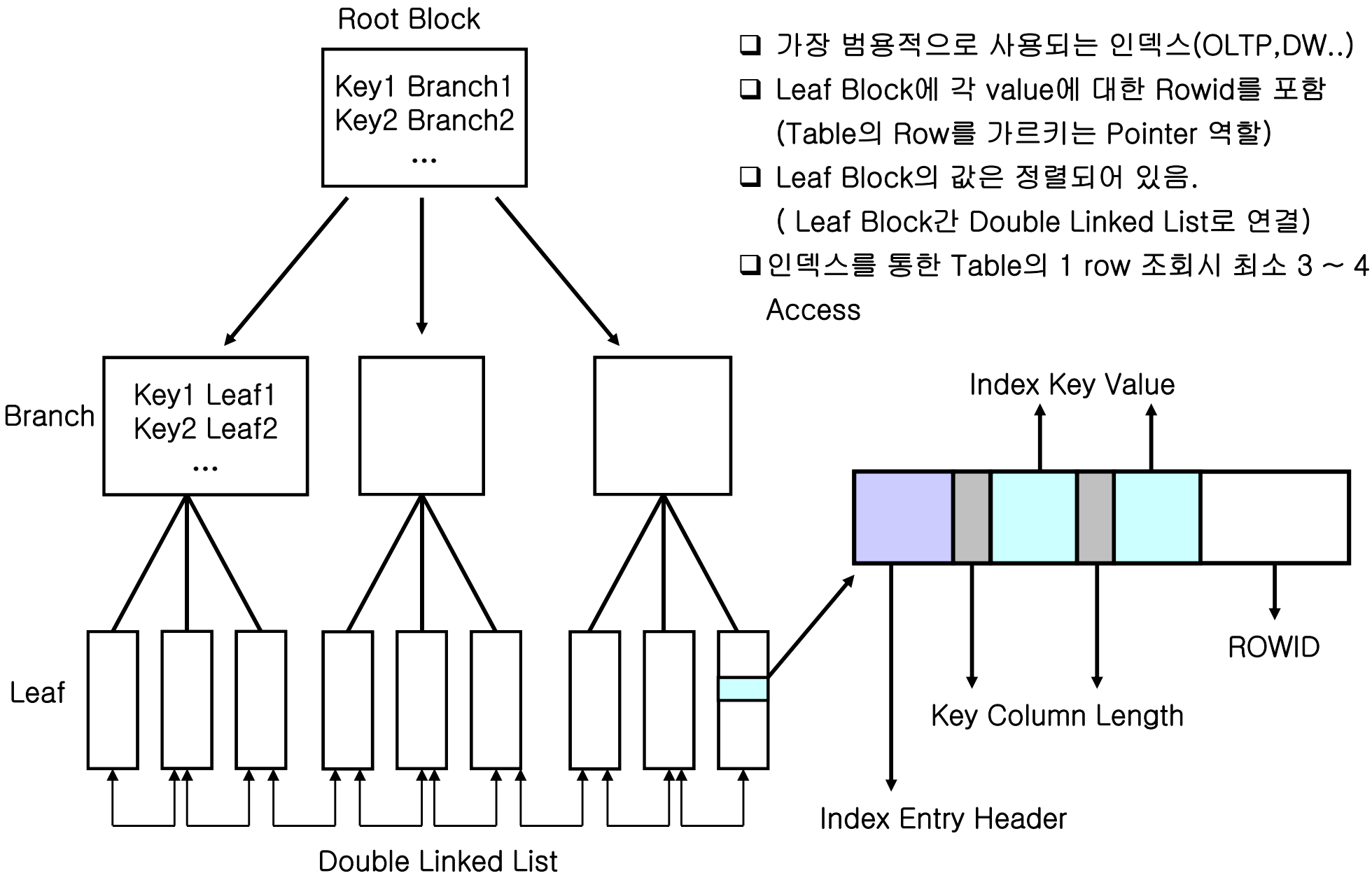
Num Nulls

- null count

Density

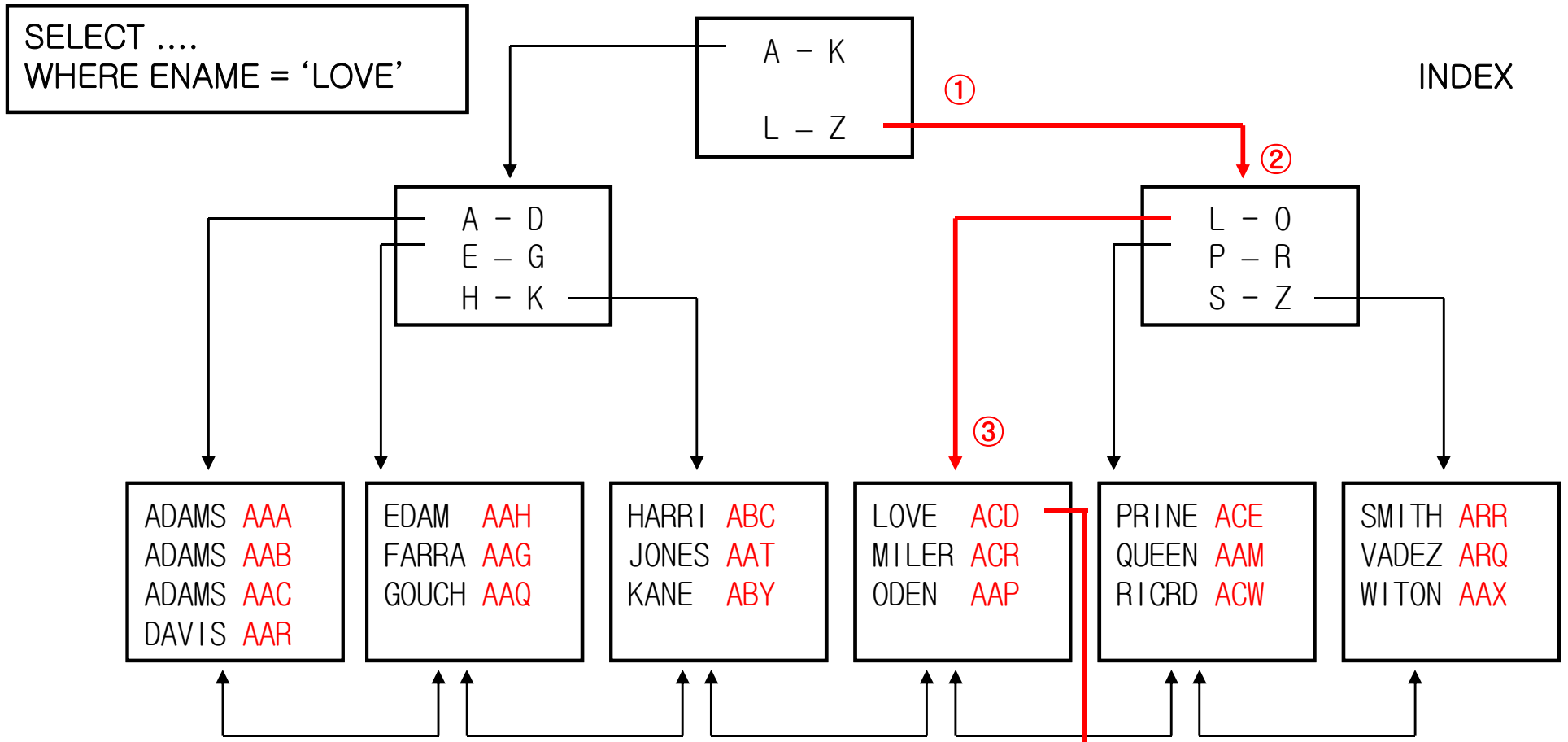
- data의 밀도

IV-2. B* TREE INDEX



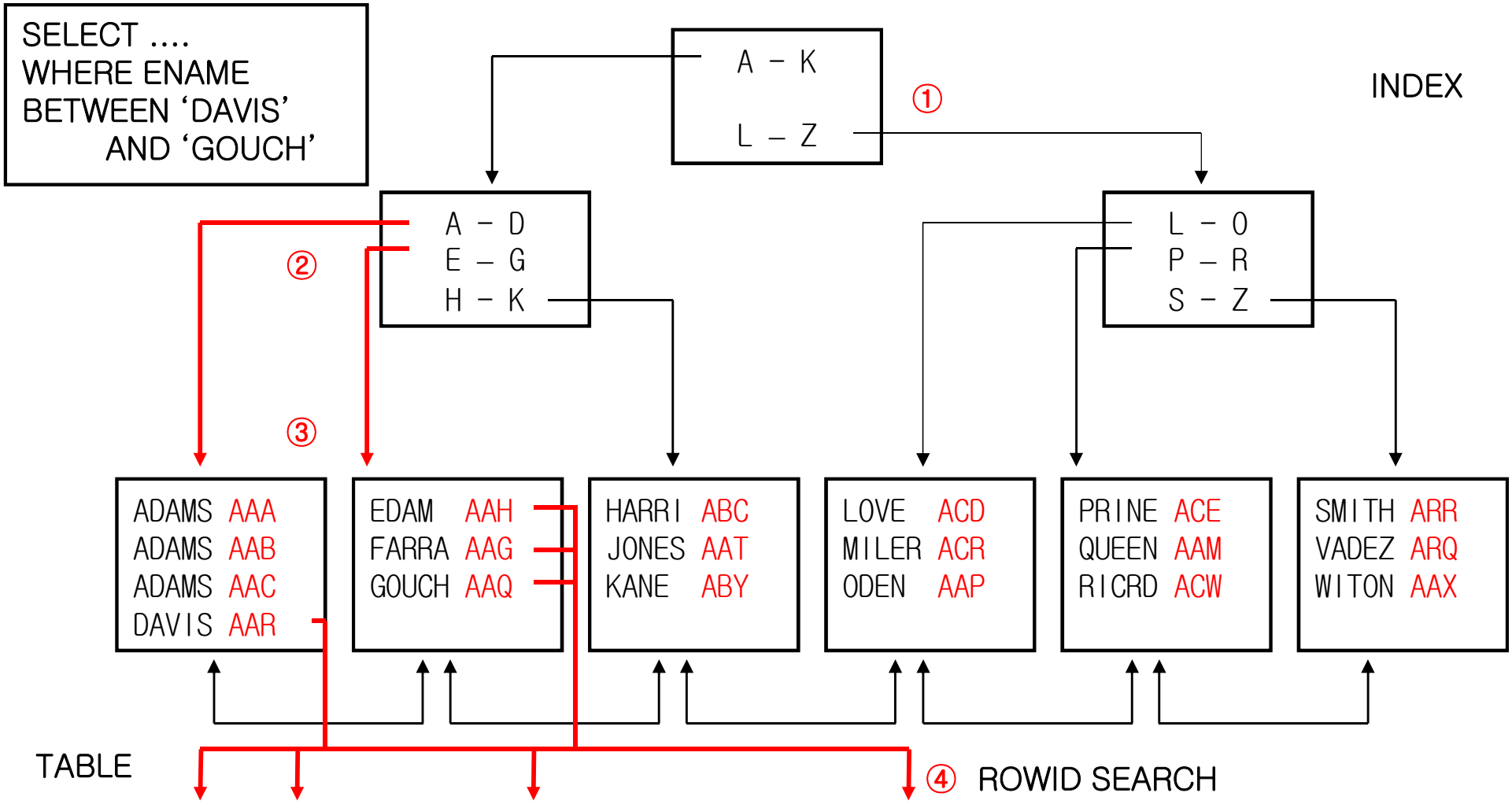
- 가장 범용적으로 사용되는 인덱스(OLTP,DW..)
- Leaf Block에 각 value에 대한 Rowid를 포함
  (Table의 Row를 가르키는 Pointer 역할)
- Leaf Block의 값은 정렬되어 있음.
  (Leaf Block간 Double Linked List로 연결)
- 인덱스를 통한 Table의 1 row 조회시 최소 3 ~ 4 Block Access

IV-2. B* TREE INDEX



ROWID	AAH	AAG	AAC	AAR	AAA	AAB	AAQ	ABC	AAT	ABY	ACD	ACR	AAP	....
ENAME	EDAM	FARRA	ADAMS	DAVIS	ADAMS	ADAMS	GOUCH	HARRI	JONES	KANE	LOVE	MILER	ODEN	....
ETC	....	....	....	....	....	....	....	....	....	....	....	....	....	....

IV-2. B* TREE INDEX

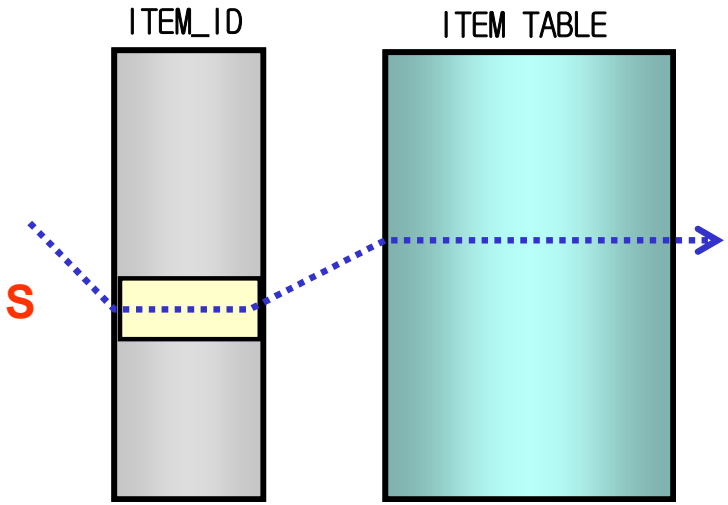


ROWID	AAH	AAG	AAC	AAR	AAA	AAB	AAQ	ABC	AAT	ABY	ACD	ACR	AAP	....
ENAME	EDAM	FARRA	ADAMS	DAVIS	ADAMS	ADAMS	GOUCH	HARRI	JONES	KANE	LOVE	MILER	ODEN	....
ETC	....	....	....	....	....	....	....	....	....	....	....	....	....	....

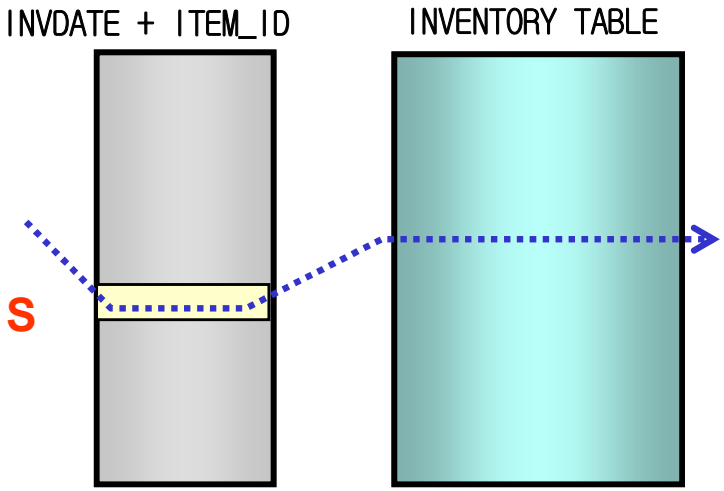
IV-3. INDEX ACCESS

UNIQUE INDEX의 UNIQUE SCAN

❑ NON-UNIQUE INDEX인 경우는 UNIQUE SCAN을 할 수 없다.



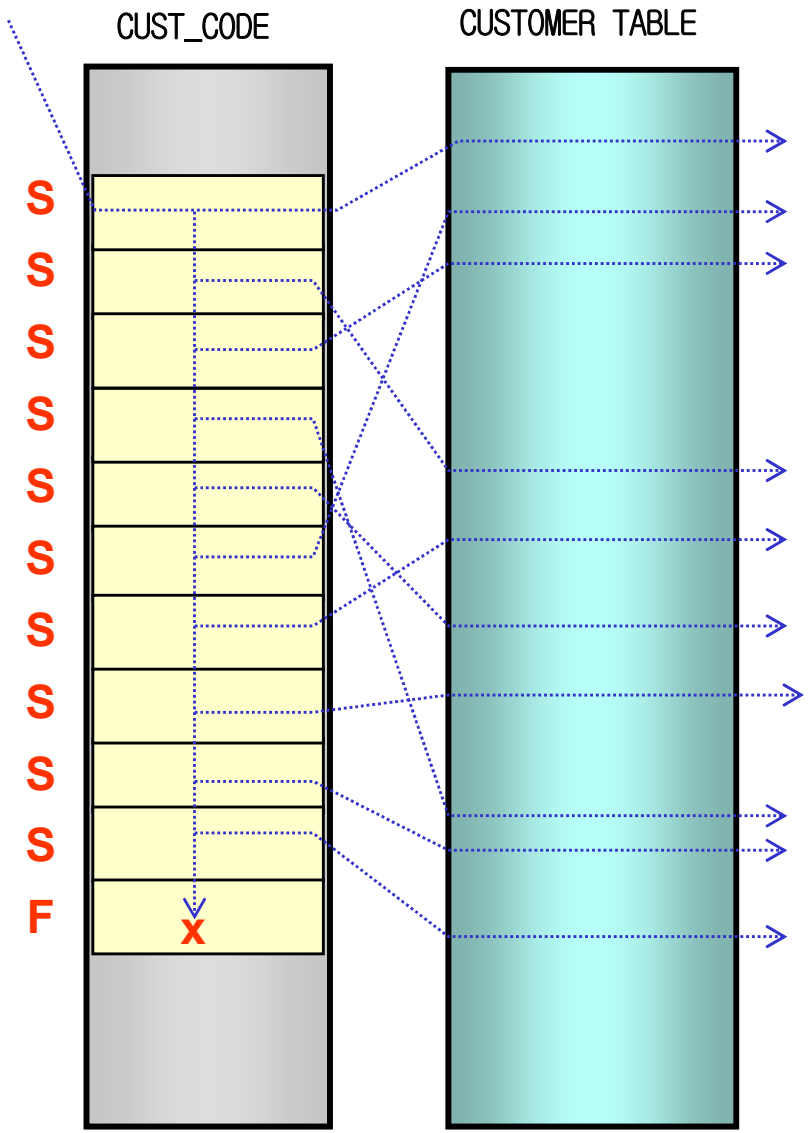

```
SELECT *
FROM   item
WHERE  item_id = 100

 Execution Plan
----------------------------------------
 SELECT STATEMENT
  TABLE ACCESS BY INDEX ROWID :ITEM
    INDEX UNIQUE SCAN :PK_ITEM (U)
```




```
SELECT *
FROM   inventory
WHERE  invdate = '19990101'
AND    item_id = 100

 Execution Plan
 ----------------------------------
 SELECT STATEMENT
  TABLE ACCESS BY INDEX ROWID :INVENTORY
    INDEX UNIQUE SCAN :PK_INVENTORY (U)
```

IV-3. INDEX ACCESS

INDEX의 RANGE SCAN



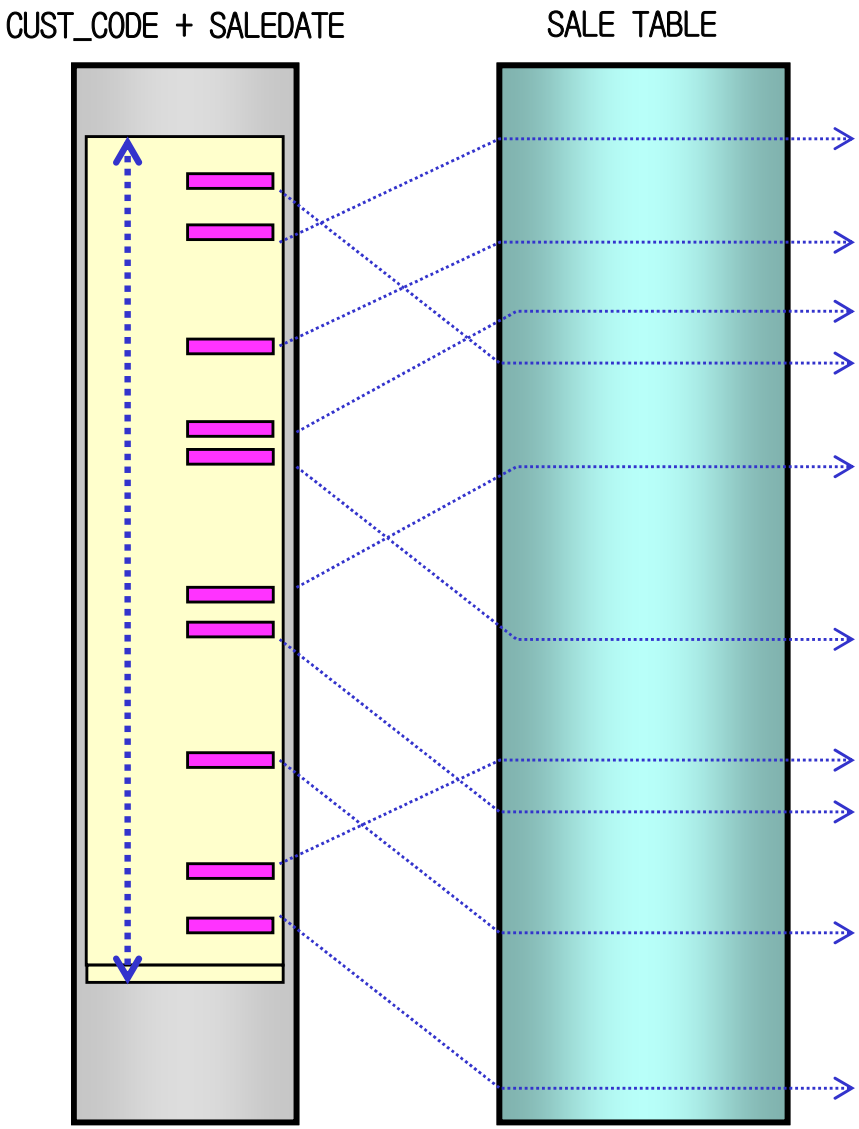

```
SELECT *
FROM   customer
WHERE  cust_code LIKE 'C3005%'

Execution Plan
----------------------------------------

 SELECT STATEMENT
   TABLE ACCESS BY INDEX ROWID :CUSTOMER
     INDEX RANGE SCAN :PK_CUSTOMER (U)
```

❑ 범위 경계값이 UNIQUE하게 들어 오는 경우도 RANGE SCAN

```
SELECT *
FROM   customer
WHERE  cust_code BETWEEN 'C30051' AND 'C30060'

Execution Plan
----------------------------------------
SELECT STATEMENT
  TABLE ACCESS BY INDEX ROWID :CUSTOMER
     INDEX RANGE SCAN :PK_CUSTOMER (U)
```

IV-3. INDEX ACCESS

결합 INDEX의 RANGE SCAN



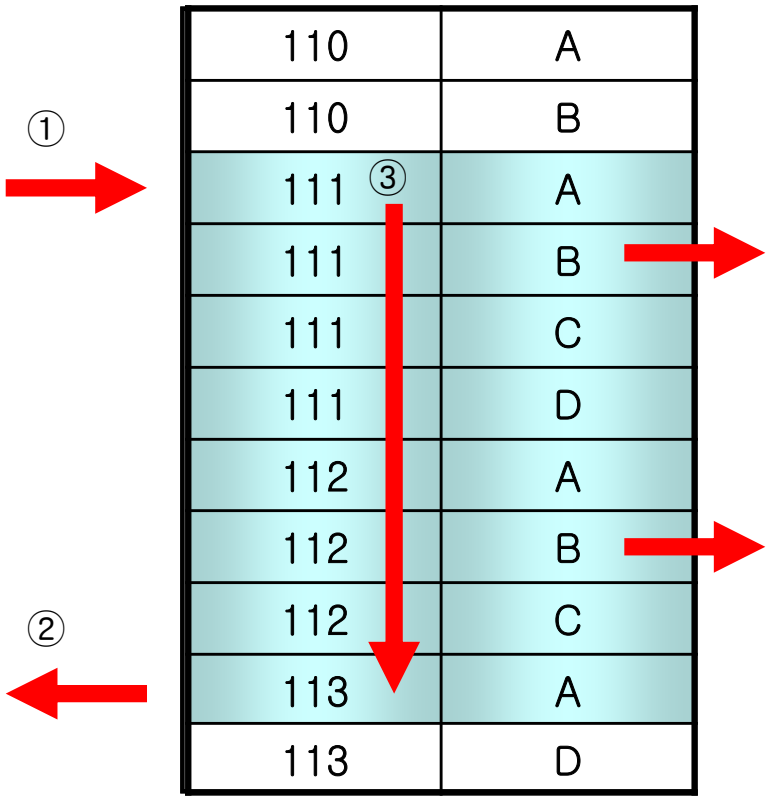
❑ 결합 index의 가장 큰 특징은 선두 컬럼이 (=,in) 으로 들어오지 않아서 범위가 넓어질 경우에 후행 컬럼의 selectivity가 좋다고 할지라도 index를 access하는 범위를 줄여주지 못하고, table을 access하는 범위만을 줄여 줄 수 있다.

```
SELECT *
FROM   sale
WHERE saledate = '20020219'
AND     cust_code like 'C3006%'

Execution Plan
----------------------------------------------------
 SELECT STATEMENT
  TABLE ACCESS BY INDEX ROWID :SALE
    INDEX RANGE SCAN :SALE_CUST_SALEDATE_IDX(NU)
```

IV-3. INDEX ACCESS

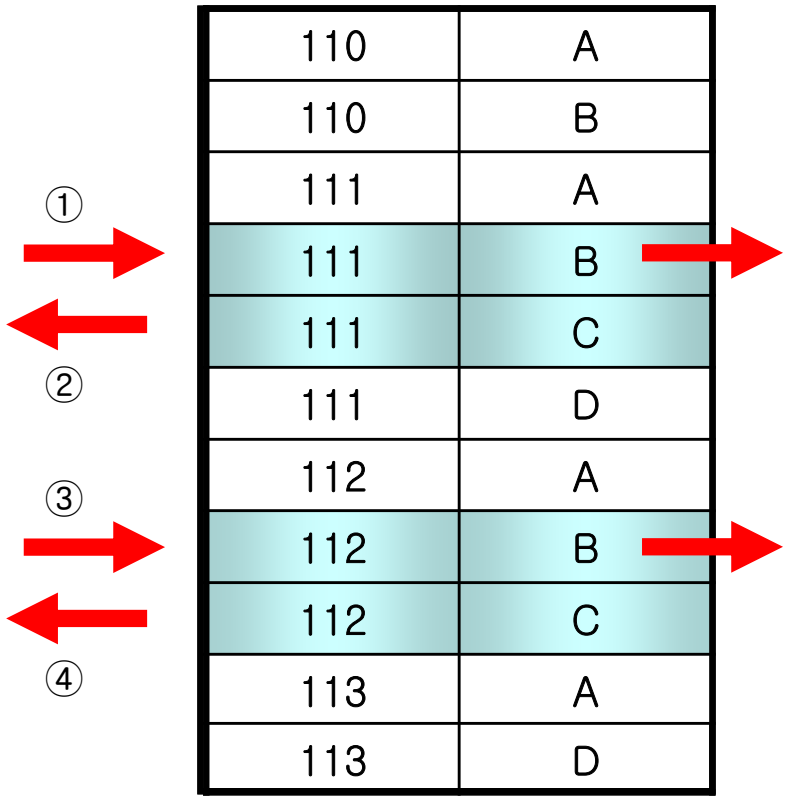
결합 INDEX의 BETWEEN과 IN의 비교

```
SELECT *  FROM  item
WHERE item_id = 'B'
AND      item_class  between '111' and '112'
```




```
Execution Plan
----------------------------
TABLE ACCESS BY ROWID ITEM
  INDEX RANGE SCAN ITEM_IDX1
```

```
SELECT *  FROM item
WHERE item_id = 'B'
    and item_class in ('111' , '112')
```




```
Execution Plan
------------------------------------
CONCATENATION
  TABLE ACCESS BY ROWID ITEM
    INDEX RANGE SCAN ITEM_IDX1
  TABLE ACCESS BY ROWID ITEM
    INDEX RANGE SCAN ITEM_IDX1
```

IV-3. INDEX ACCESS

INDEX FULL SCAN

- ❑ index full scan은 index를 순차적으로 처음부터 끝까지 읽어들인다.
- ❑ parallel process를 사용하지 못한다.
- ❑ 사용하지 않는것이 좋다.
- ❑ Ordering을 보장.

INDEX FAST FULL SCAN

- ❑ index fast full scan은 index를 비순차적으로 처음부터 끝까지 읽어들인다.(Ordering 미보장)
- ❑ parallel process를 사용할 수 있다.
- ❑ index로 구성된 컬럼만을 조회할때 사용이 가능하다.
- ❑ Multi-block I/O가 가능하다.

Index: deptno + sal

```
SELECT /*+ INDEX_FFS(emp emp_deptno_sal_ix) */
          deptno, sum(sal)
FROM     emp
GROUP BY deptno;

Execution Plan
----------------------------------------------------------
 SELECT STATEMENT Optimizer=CHOOSE
   SORT (GROUP BY)
     INDEX (FAST FULL SCAN) OF 'EMP_DEPTNO_SAL_IX ' (NON-UNIQUE)
```

IV-4. Bitmap INDEX

```
SELECT  count(*)
FROM    emp
WHERE   gender = 'MAN'
AND     married = 'YES'
```

empno	gender	married	
001	WOMAN	NO	…
002	WOMAN	NO	…
003	MAN	YES	…
004	MAN	NO	…
005	WOMAN	NO	…
006	MAN	YES	…
…	….	…..	…

Bitmap Function

Bitmap Index

gender	=	'MAN '	0	0	1	1	0	1	0	1	1	1	0	0	1	1	1	1
gender	=	'WOMAN'	1	1	0	0	1	0	1	0	0	0	0	1	0	0	0	0
married	=	'YES '	0	0	1	0	0	1	0	1	0	1	0	0	0	1	0	1
married	=	'NO'	1	1	0	1	1	0	1	0	1	0	0	1	1	0	0	0

- ❑ each different key value has its own bitmap
- ❑ each position in the bitmap maps to a possible ROWID
- ❑ if value 1 true, 0 false

IV-4. Bitmap INDEX

장점

- ❑ Cardinality가 낮은 열에 대해 사용
- ❑ Small Storage 사용
- ❑ AND/OR 등으로 결합된 복합 조건에 최적
- ❑ 전통적인 B*_Tree 인덱스의 단점 해소(or, not, null,..)
- ❑ 전체 Null column도 Index에 저장
- ❑ Bitmap 압축을 통한 세그먼트 사이즈 감소.(NULL값에 대한)
- ❑ DATAWARE HOUSE등 대량의 Data를 Read Only Mode로 사용시에 적당

단점

- ❑ DML 작업에 취약
- ❑ Block Level Locking
- ❑ Rule Base Optimizer에서는 사용 못함
- ❑ Online option(build, rebuild) 사용 못함

IV-4. Function Base Index

❑ Oracle 8i New Feature 로서 조건식의 결과값에 대해 인덱스 생성.

❑ FBI 사용 조건

- 8i CBO + QUERY_REWRITE_INTEGRITY + QUERY_REWRITE_ENABLED 이어야 FBI 사용 가능.
- 9i CBO + QUERY_REWRITE_INTEGRITY 로 FBI 사용 가능. → CBO 만으로 가능.

❑ 예

```
CREATE INDEX  BIG_OBJECTS_IDX12 ON BIG_OBJECTS(OWNER || OBJECT_NAME)

SELECT  *  FROM BIG_OBJECTS
 WHERE OWNER || OBJECT_NAME = 'FIRE' || 'PLAN_TABLE'

SELECT STATEMENT CHOOSE-Cost : 2
 TABLE ACCESS BY INDEX ROWID FIRE.BIG_OBJECTS(1) Analyzed : 20050111
  INDEX RANGE SCAN FIRE.BIG_OBJECTS_IDX12(NU) (SYS_NC00014$) NOT ANALYZED
        ("BIG_OBJECTS"."OWNER"||"BIG_OBJECTS"."OBJECT_NAME"='FIREPLAN_TABLE')


CREATE INDEX COPY_TF_F01 ON COPY_TF(NO + 1)     ;
 SELECT * FROM COPY_TF WHERE NO + 1 = 2        -- COPY_TF_F01 인덱스 엑세스 가능!
 SELECT * FROM COPY_TF WHERE NO  +  1 = 2      -- COPY_TF_F01 인덱스 엑세스 가능!
 SELECT * FROM COPY_TF WHERE NO  +  ' 01 ' = 2 -- COPY_TF_F01 인덱스 엑세스 가능!
```

❑ 조건식의 파싱과 매칭은 일반 상수조건 비교와 같다.(문자, 숫자 비교 등)

- FBI 표현식에서 표현식의 구문의 대소문자나 공백 등의 차이는 FBI 구문인식과는 상관없다.

→ 의미상으로 동일하기만 하면 FBI로 인식, 처리. **(→ 최종 변환식이 인덱스 구문이 된다.)**

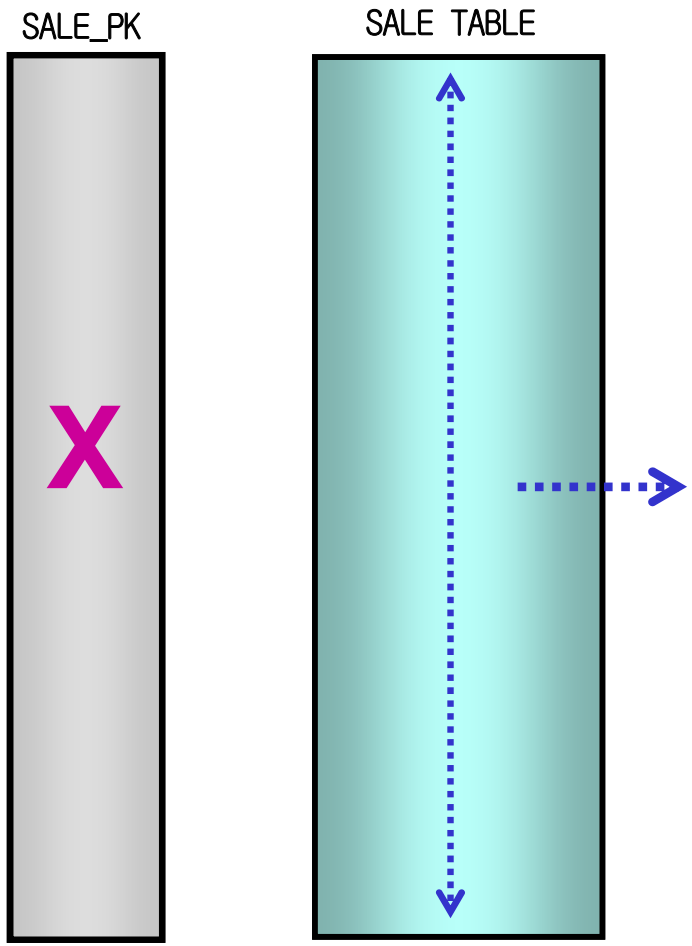
❑ Descending Index

- Oracle 8i 이전에는 DB2와의 호환을 위해 인덱스 생성시 DESC 구문이 존재 했었으며

실제 Descending Index는 8i 부터 제공되었다. → Descending Index는 FBI 로 생성된다.!

IV-5. INDEX SUPPRESSING

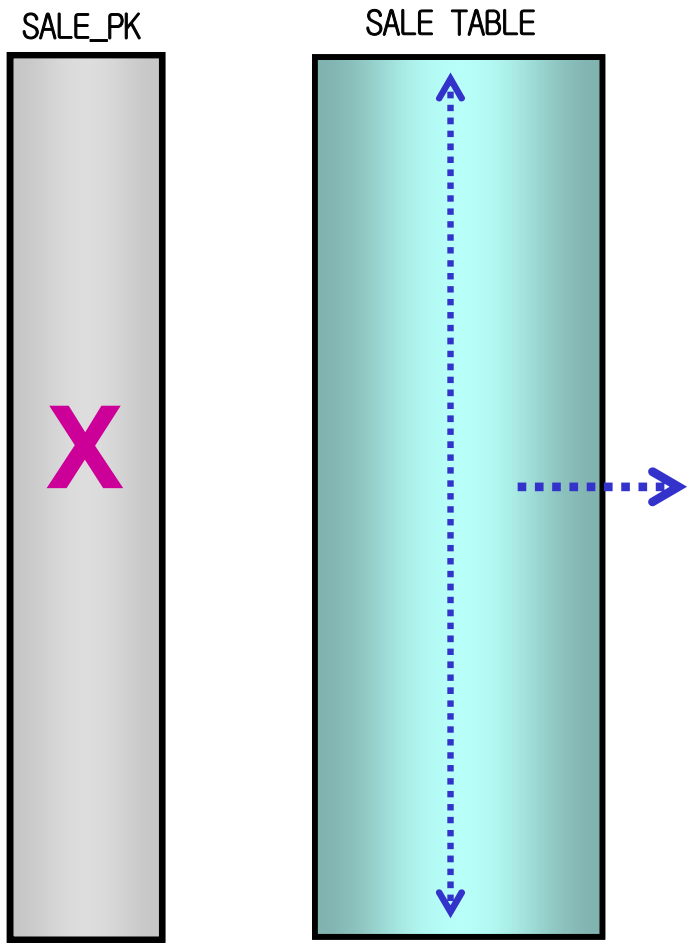
INDEX COLUMN의 EXTERNAL SUPPRESSING




```
SELECT *
FROM   sale
WHERE substr(saledate,1,4) = '2003'

Execution Plan
----------------------------------------
SELECT STATEMENT
  TABLE ACCESS FULL :SALE

(튜닝후)
SELECT *
FROM   sale
WHERE saledate like '2003%'

Execution Plan
----------------------------------------
SELECT STATEMENT
  TABLE ACCESS BY INDEX ROWID :SALE
    INDEX RANGE SCAN :SALE_PK (U)
```

IV-5. INDEX SUPPRESSING

INDEX COLUMN의 INTERNAL SUPPRESSING




```
SELECT *
FROM   sale
where saledate = 20020103

Execution Plan
----------------------------------
 SELECT STATEMENT
    TABLE ACCESS FULL :SALE

(튜닝후)
SELECT *
FROM   sale
WHERE   saledate = '20020103'

Execution Plan
----------------------------------
 SELECT STATEMENT
     TABLE ACCESS BY INDEX ROWID :SALE
        INDEX UNIQUE SCAN :SALE_PK (U)
```

IV-5. INDEX SUPPRESSING

INDEX COLUMN의 SUPPRESSING 유형

❑ 인덱스를 구성하고 있는 컬럼에 대한 조건이 함수의 적용 등의 이유에 의해서 변형됨으로써 인덱스를 사용할 수 없는 상태의 SQL

❑ 모델링 이상 – 데이터 타입

```
SELECT …
FROM  a, b
WHERE a.reg_ymd = to_char(b.reg_date,'YYYYMMDD')
```

❑ 모델링 이상 – Concatenated column

```
SELECT …
FROM  a, b
WHERE a.reg_ymd = b.yyyy || b.mm || b.dd
```

❑ 편의주의적인 코딩 습관

```
SELECT …
FROM  b
WHERE  trunc(b.reg_date) = trunc(SYSDATE)
```

❑ Bind 변수 타입 불일치

```
SELECT …
FROM  a
WHERE  a.reg_ymd = :vd_reg_date
```

IV-5. INDEX SUPPRESSING

INDEX COLUMN의 SUPPRESSING 해결방안

❑ Expression

```
SELECT ... FROM   tab
WHERE  col1 + 10 > 100;

SELECT ... FROM   tab
WHERE  col1 > 100-10;
```

❑ DATE type

```
SELECT ... FROM   tab
WHERE  trunc(reg_date) = truc(sysdate);

SELECT ... FROM   tab
WHERE  reg_date >= trunc(sysdate) AND reg_date < trunc(sysdate) + 1;
```

❑ SUBSTR()을 LIKE로

```
SELECT ... FROM   tab
WHERE  substr(col1,1,3) = 'abc';

SELECT ... FROM   tab
WHERE  col1 like 'abc%';
```

IV-6. Table Full Scan

❑ Table Full Scan이란 Table의 전체 Data을 처음부터 끝(HWM)까지 차례대로 읽는 것.

Table Segment 의 구조도

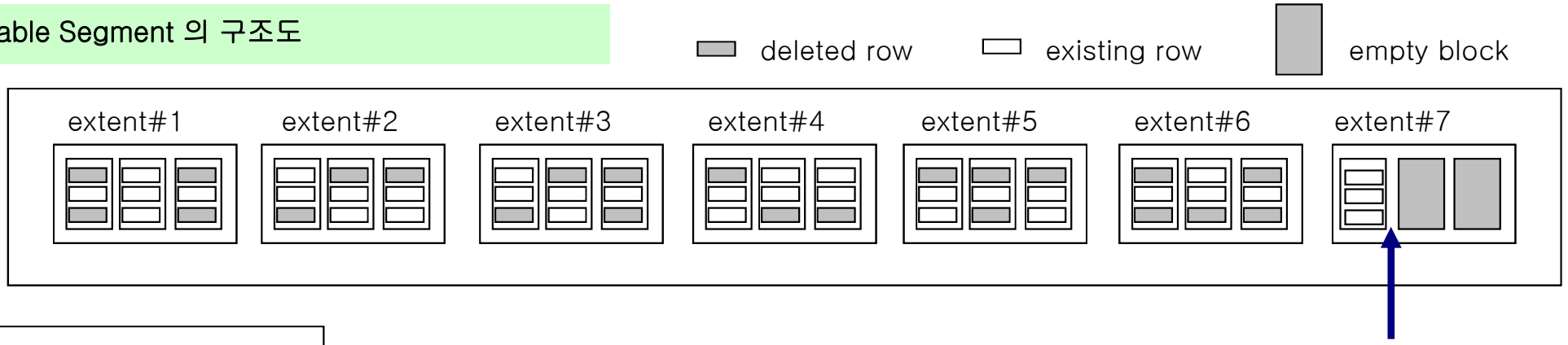


Table Full Scan 조건

- ❑ No index (Column suppressing)
- ❑ hint -> /*+ full(tablename or alias) */
- ❑ Analyze (Small table)
- ❑ Sort join or Hash join

CBO Optimizer의 INDEX VS TABLE FULL SCAN 판단기준

(※ RBO는 Rule에 의해 항상 Index가 좋다고 판단)

- ❑ number of blocks(under High Water Mark)
- ❑ Size of data blocks
- ❑ Number of blocks(db_file_multiblock_read_count)
- ❑ Selectivity of the index
- ❑ Depth of the index

IV-6. Table Full Scan

Table Full Scan 특징

- 0 건인 테이블을 count(*)하였을 때 느린 경우
- 전체 Table을 대상으로 작업하는 경우는 항상 Index Range Scan보다 유리.
  (Full Scan Vs Index Scan 선택기준: 0.1 ~ 50 %로 Data상황에 따라 유동적임)
- I/O 량을 예측할 수 있음 -> Multiblock read(index scan은 단일 Block read)
  alter session set db_file_multiblock_read_count = 32
- Parallel processing가능 -> hint (/*+ parallel(a 4) */)
- Caching 여부 선택 가능 -> hint (/*+ cache(tablename) */)
- Buffer 메모리의 효율적인 사용 (LRU List에서만 사용 및 경합이 적다.)
- Read Only 성격의 Table인 경우는 Pctfree, Pctused를 조정하여 한개의 Block에 최대한 Data를 입력하면 Full Scan 효율향상

V. Join

1. **Join** 이란
2. **Nested Loop Join**
3. **Hash Join**
4. **Sort Merge Join**
5. **Join** 방법 비교
6. **Special Joins**

V-1. Join 이란

- Join이란 2개 이상의 table들을 Relation key로 연결하여 merge 시키는 것

  조인이란 항상 2개의 집합 간의 처리이다.
- Semi join을 제외하고 무조건 Relation key로 연결되는 모든 value들을 조회 (Anti Join)
- Join의 종류(각 조인별로 Anti Join, Semi Join 가능)

  Nested Loop Join

  Hash Join

  Sort Merge Join
- Relation

  - 1:1 relation

    data가 많은 table을 2개이상의 table로 분할 하였을 경우 발생

  - 1:m relation

    일반적인 modeling에 가장 자주 발생하는 관계

  - m:m relation

    ERD 상에서 멀리 떨어져 있는 table과의 join에서 자주 발생하며, 연결고리의 관계가 m:m이므로

    원하는 집합보다 data가 많아 질 수 있음.

    Subquery를 사용하면 m:m join관계를 극복할 수 있음.
- Cartesian Product

  Join key가 생략된 join으로 결과값에 m * m 의 전체집합이 나오는 것(처리를 의도한 경우에만 사용해야 한다.)

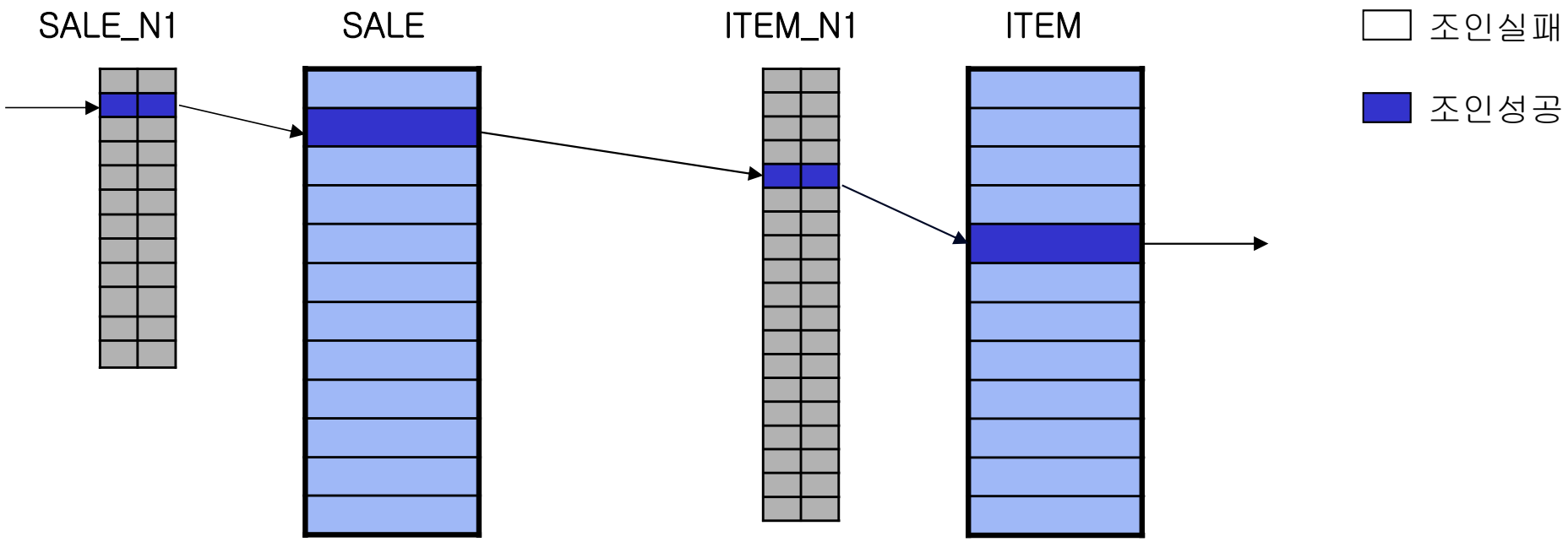
V-2. Nested Loop Join

Nested Loop Join 이란

- ❑ 선행(Driving) table의 조건에 맞는 data와 매치되는 값을 다른(후행) table에서 참조
- ❑ OLTP SYSTEM에서 가장 많이 사용되는 기본적인 Join
- ❑ 두 개의 결과를 순차적으로 검증하면서 Join
  - 별도의 결과 대기 영역이 필요 없다.
  - FIRST_ROWS 에 최적
- ❑ Index 생성에 따른 여파
  - Index를 적절히 설정하면 가장 빠른 성능 보장.
  - Index를 잘못 설계한 경우 최악의 성능 가능.
- ❑ Join order
  - Driving table의 선택은 범위를 가장 줄여줄 수 있는 table 부터 시작하여 많이 걸러지는 순서대로 Join 하는것이 유리.
  - INNER 테이블의 Access Path에 매우 민감
  - FROM 절의 테이블 순서 조정 필요 (ORDERED hint 사용)
  - 때로는 조인 연결조건의 변경 필요
  - 필요없는 Outer Join을 사용하지 않도록 한다.

V-2. Nested Loop Join

Nested Loop Join (Unique:Unique)

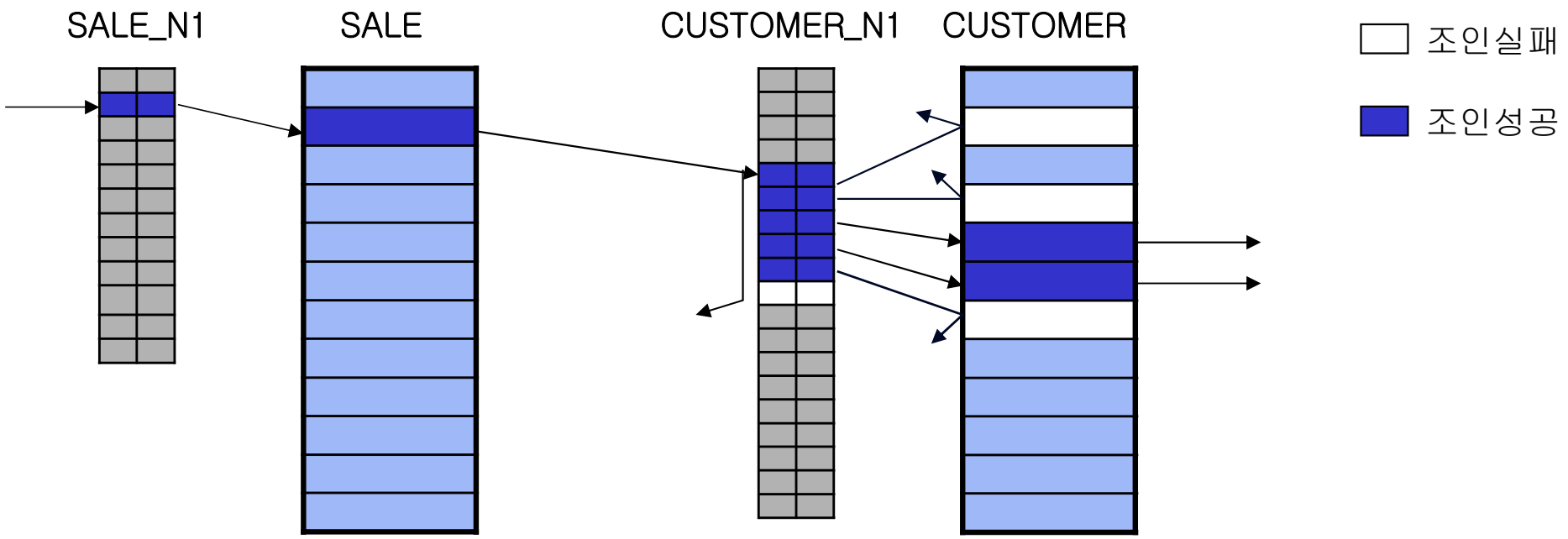



```
SELECT a.saledate, a.item_id,
b.item_name, b.item_class
FROM    sale a
        ,item b
WHERE  a.saledate   = '20030405'
AND      a.item_id  =  b.item_id
```

```
EXPLAIN PLAN
----------------------------------------
SELECT STATEMENT
  NESTED LOOPS
    TABLE ACCESS (BY INDEX ROWID) :SALE
      INDEX (UNIQUE SCAN) :SALE_N1
    TABLE ACCESS (BY INDEX ROWID) :ITEM
      INDEX (UNIQUE SCAN) :ITEM_N1
```

V-2. Nested Loop Join

Nested Loop Join (Unique:Range)

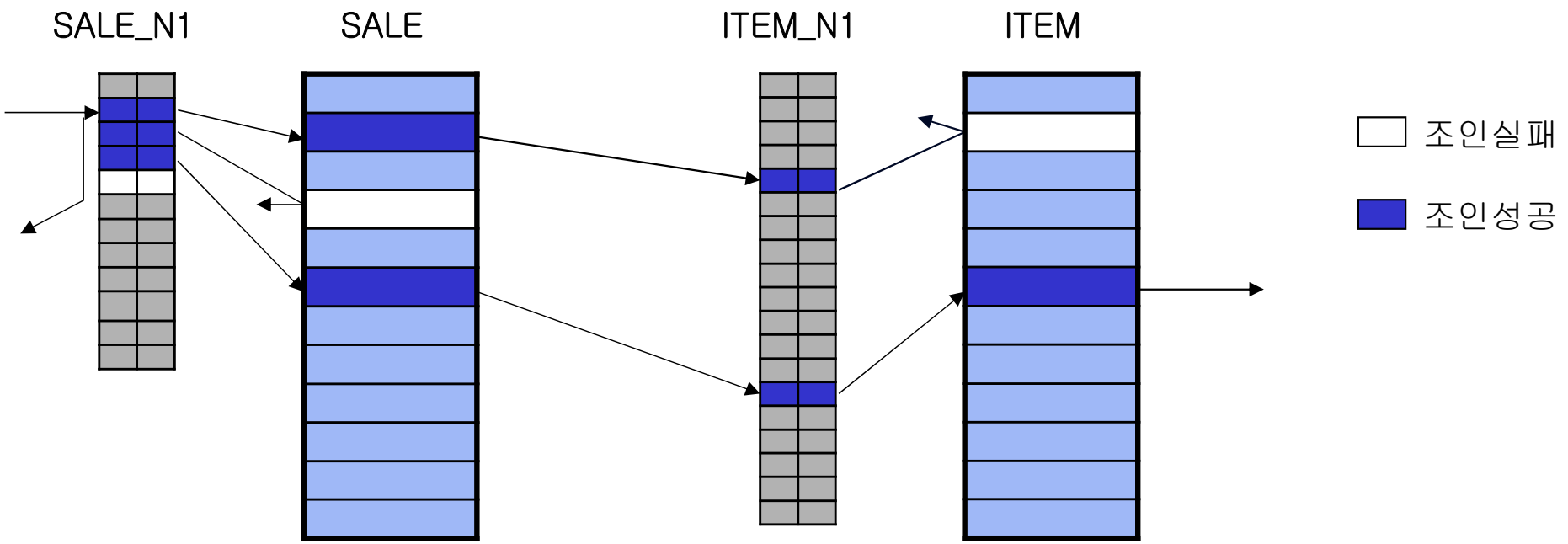



```
SELECT a.saledate, a.name,
       b.customer_id,b.customer_level
FROM   sale a
       ,customer b
WHERE a.saledate  = '20030405'
AND     a.name      =  b.name
AND     b.customer_level =  '상'
```

```
EXPLAIN PLAN
----------------------------------------
SELECT STATEMENT
  NESTED LOOPS
    TABLE ACCESS (BY INDEX ROWID) :SALE
      INDEX (UNIQUE SCAN) :SALE_N1
    TABLE ACCESS (BY INDEX ROWID) :CUSTOMER
      INDEX (RANGE SCAN) :CUSTOMER_N1
```

V-2. Nested Loop Join

Nested Loop Join (Range:Unique)

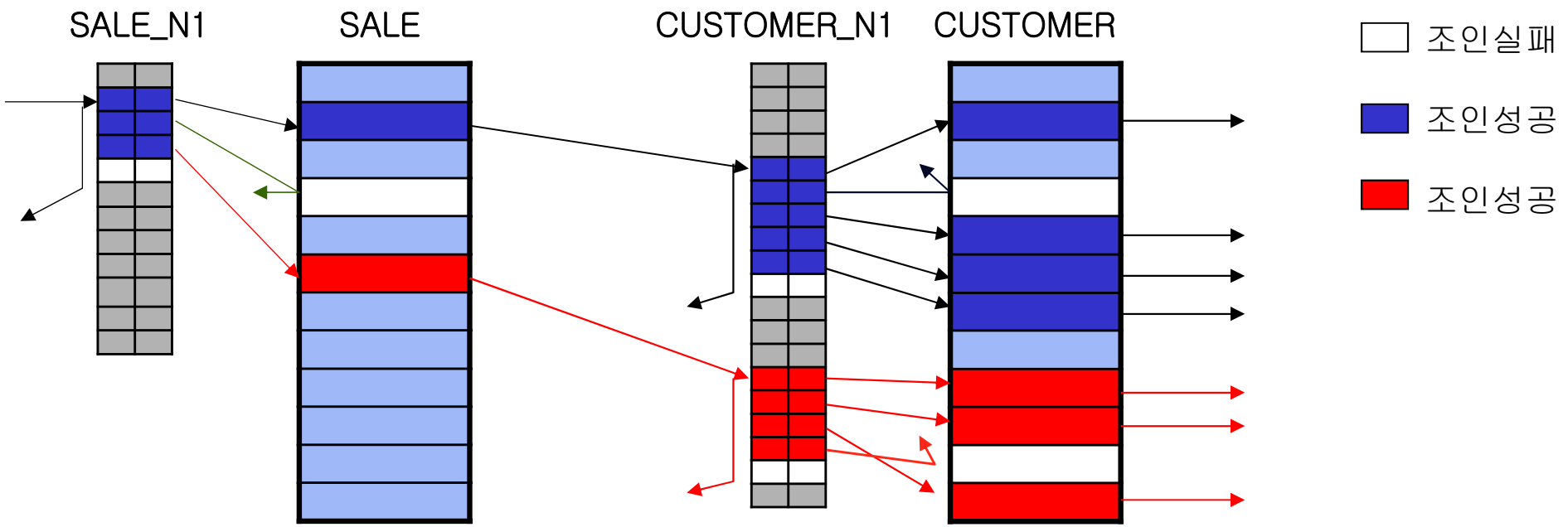



```
SELECT a.saledate, a.item_id,
b.item_name, b.item_class
FROM    sale a
        ,item b
WHERE  a.saledate like '200304%'
AND     a.item_id  =  b.item_id
```

```
EXPLAIN PLAN
----------------------------------------
SELECT STATEMENT
  NESTED LOOPS
    TABLE ACCESS (BY INDEX ROWID) :SALE
      INDEX (RANGE SCAN) :SALE_N1
    TABLE ACCESS (BY INDEX ROWID) :ITEM
      INDEX (UNIQUE SCAN) :ITEM_N1
```

V-2. Nested Loop Join

Nested Loop Join (Range:Range)



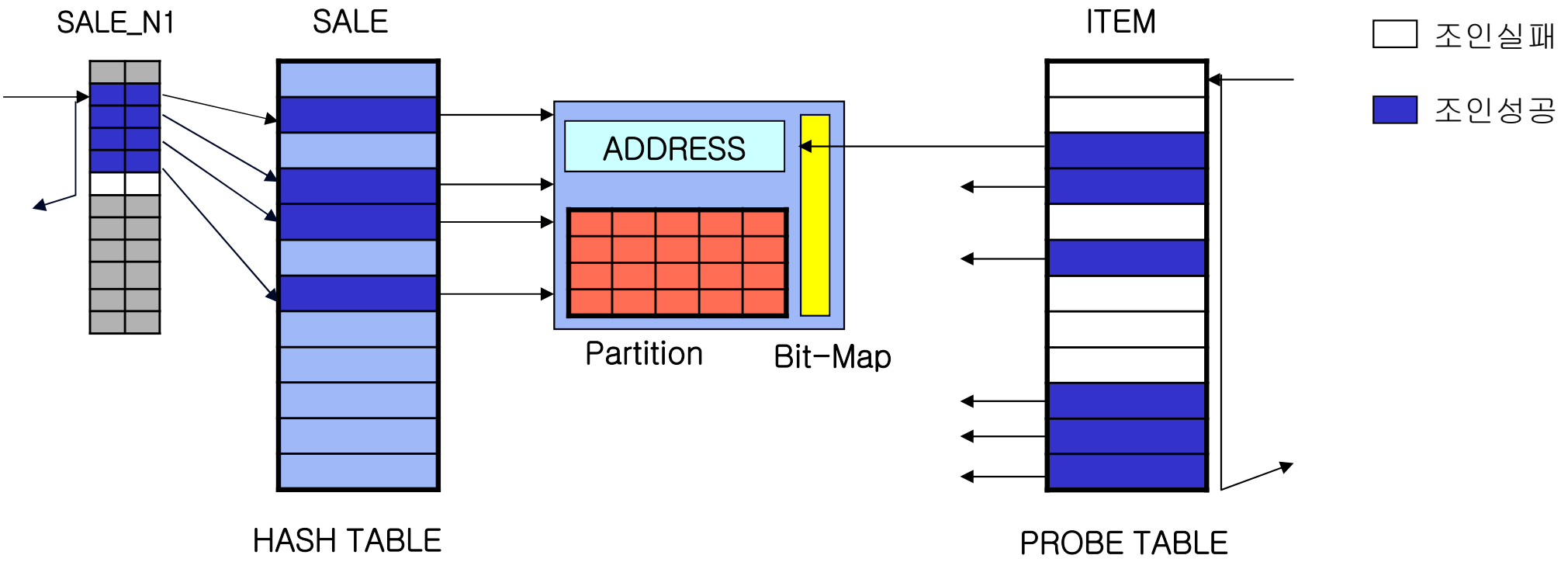

```
SELECT a.saledate, a.name,
       b.customer_id,b.customer_level
FROM   sale a
       ,customer b
WHERE a.saledate like '200304%'
AND    a.name      =  b.name
AND    b.customer_level =  '상'
```

```
EXPLAIN PLAN
----------------------------------------
SELECT STATEMENT
  NESTED LOOPS
    TABLE ACCESS (BY INDEX ROWID) :SALE
      INDEX (RANGE SCAN) :SALE_N1
    TABLE ACCESS (BY INDEX ROWID) :CUSTOMER
      INDEX (RANGE SCAN) :CUSTOMER_N1
```

V-3. Hash Join

- CBO에서만 가능
- Equi-Join에서만 가능 (가상적으로 집합을 만들어서 유도 가능)
- hash_area_size에 의해 성능 결정.
- CPU POWER 에 의존적(hash 연산)
- 대용량의 자료를 Access 하는 경우 Sort Merge Join, Nested Loop Join 보다 빠름.
- hash_join_enabled=true의 상태에서 가능
- hash_multiblock_io_count 개수만큼의 블록들을 동시에 read/write 가능(Hash Partition)
- 절반의 부분범위 처리.
- USE_HASH hint.
- Batch Job에서 Parallel SQL와 함께 사용 가능.
- 선행 table은 가능하면 작은 cardinality, 조건에 의해 선택도가 적은 것을 선택.

V-3. Hash Join

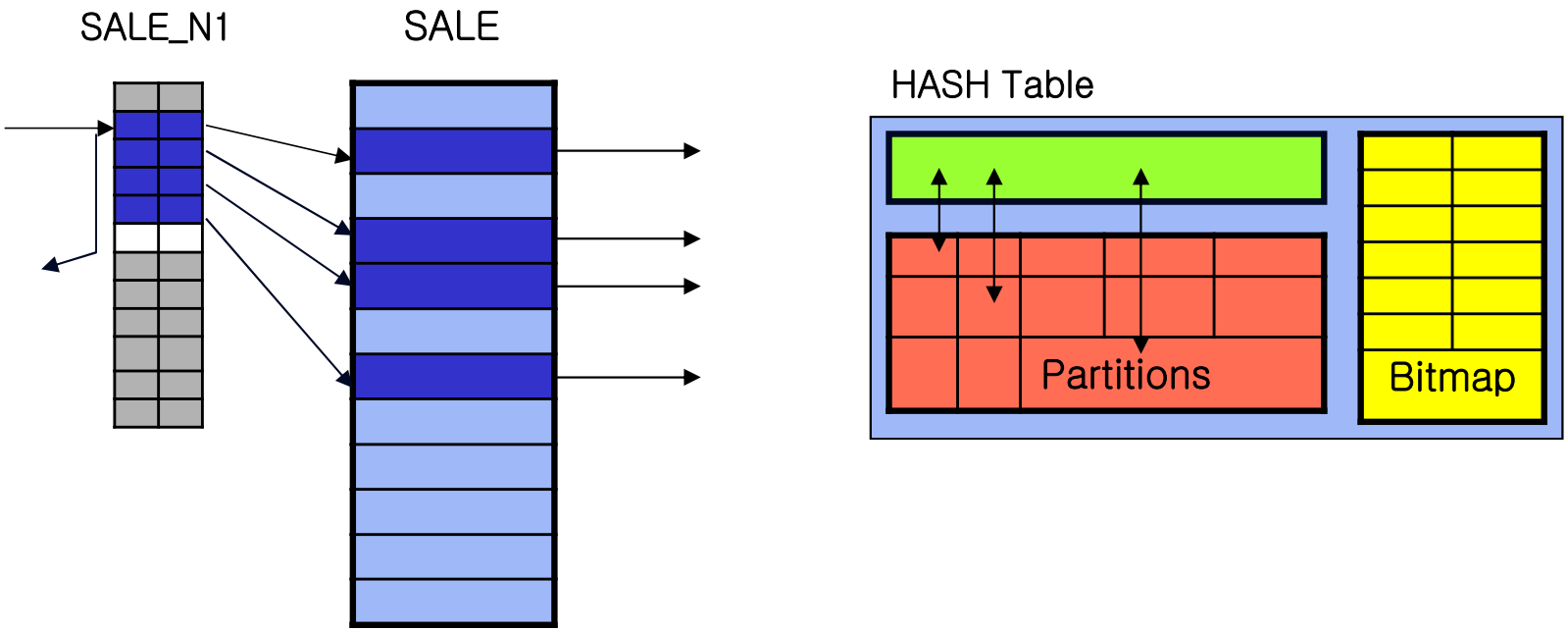



```
SELECT  /*+ ORDERED USE_HASH( B )  */
          a.saledate, a.name,
          b.item_id, b. item_level
FROM    sale a
          ,item b
WHERE a.saledate like '200304%'
AND     a.name   = b.name
```

```
EXPLAIN PLAN
----------------------------------------
SELECT STATEMENT
  HASH JOIN
    TABLE ACCESS (BY INDEX ROWID) :SALE
      INDEX (RANGE SCAN) :SALE_N1
    TABLE ACCESS  FULL  :ITEM
```

V-3. Hash Join

Hash Join 세부 처리 절차 1



1. Partition 개수 결정

※ Number of Partition = $\frac{0.8 * \text{hash\_area\_size}}{\text{db\_block\_size} * \text{hash\_multiblock\_io\_count}}$

2. Hash Table 생성 (Hash_area_size 내에 생성 가능한 경우)

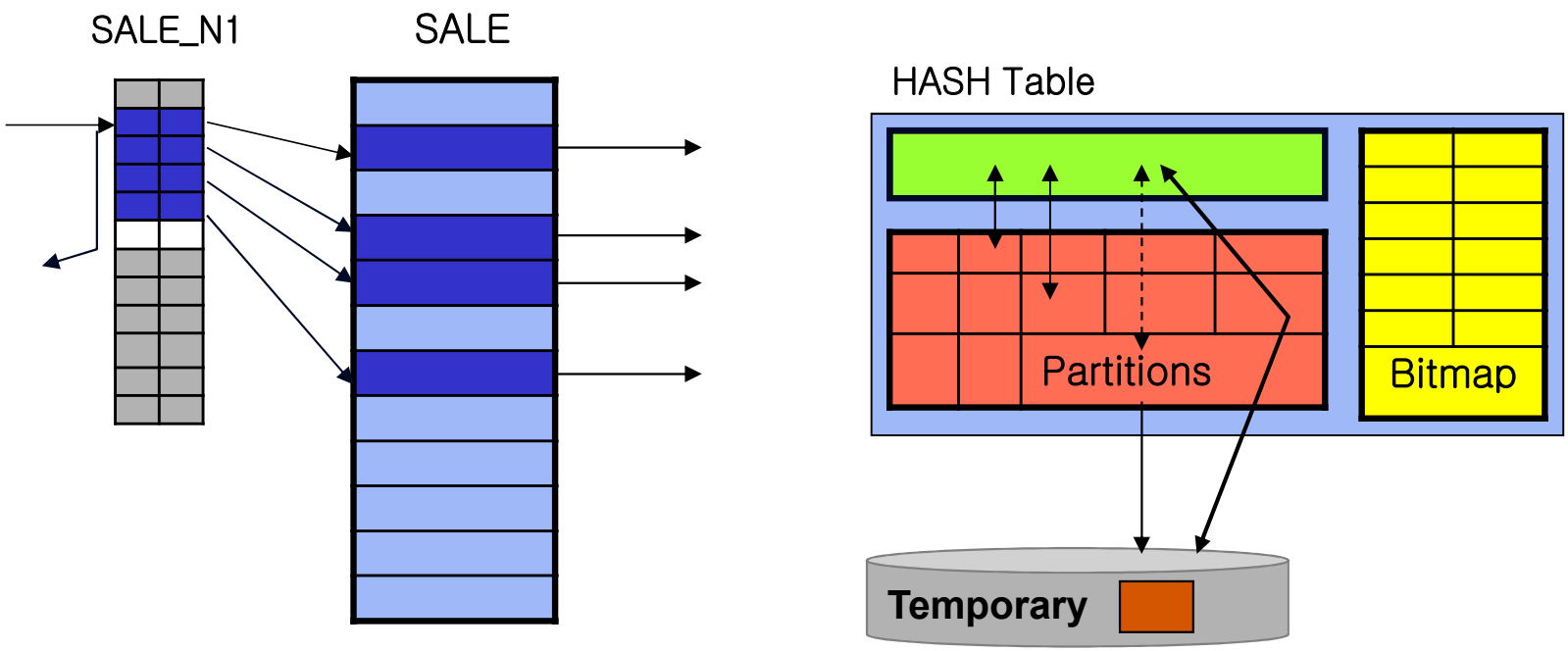
① Hash Function : Partition 결정

② Partition : Join column(s) + Select list(s) + Hash value (조인컬럼에 대해 Sort 시킴)

③ Bitmap : Hash value bitmap

V-3. Hash Join

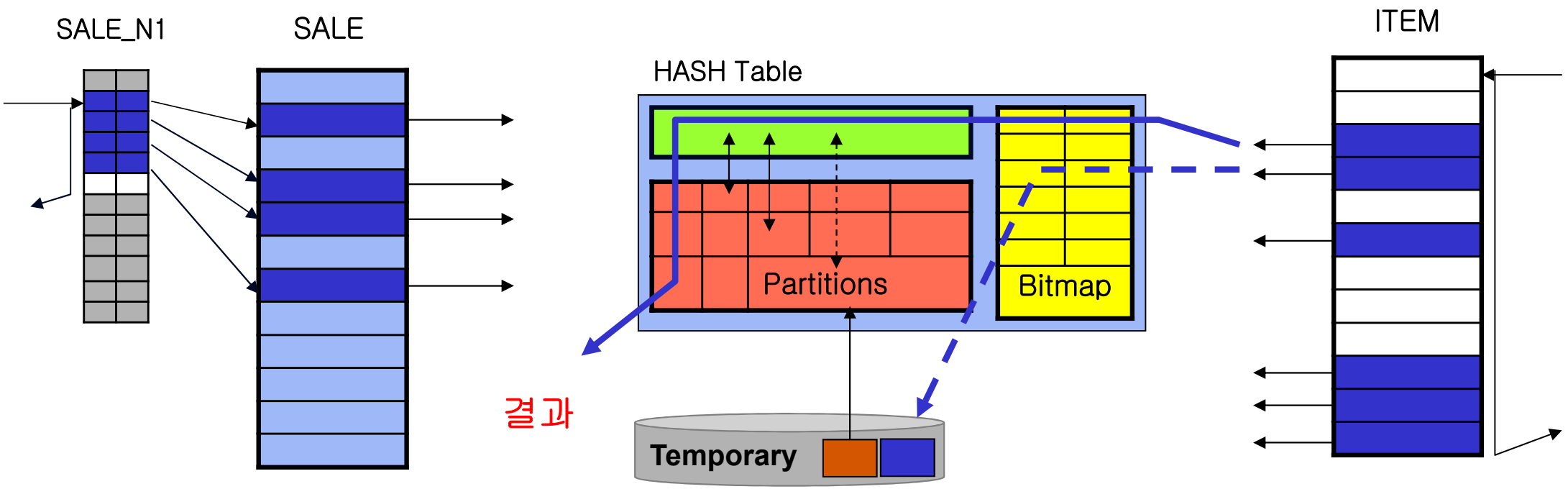
Hash Join 세부 처리 절차 2



Hash Table 생성 (Hash_area_size 를 초과할 경우)
① 가장 큰 Partition 을 Temporary table (On disk) 에 Write

V-3. Hash Join

Hash Join 세부 처리 절차 3



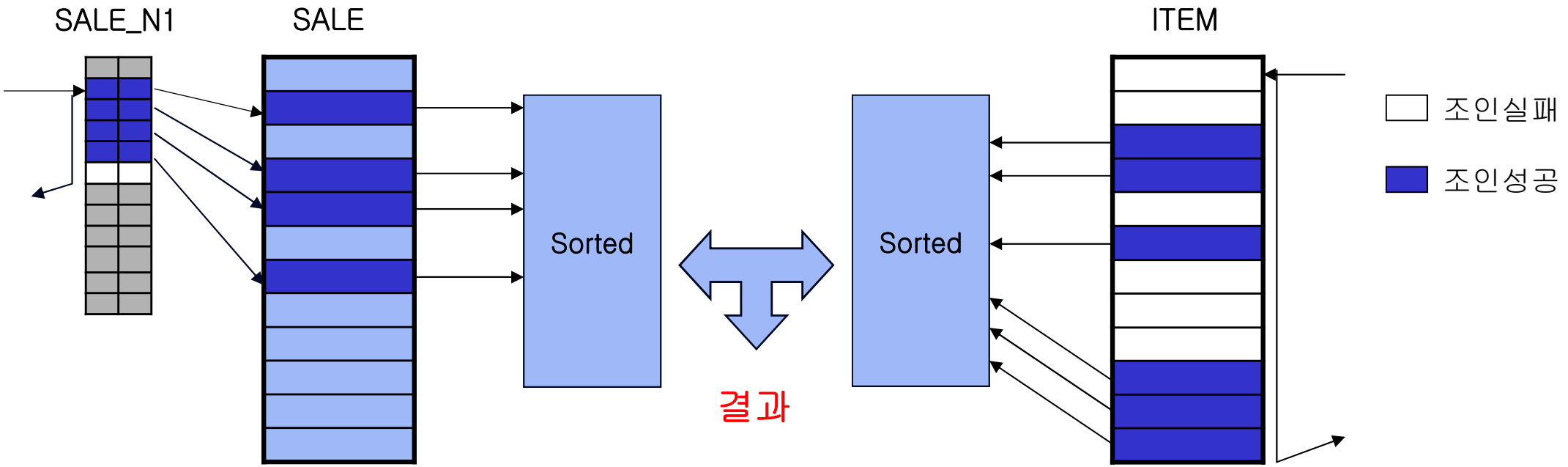
3. Probing Table Read & Join
   ① Join Column(s) 에 대한 Hash Function 적용
   ② Bitmap 에 Filtering
   ③ Memory 내에 존재하는 Partition 일 경우
      - Hash value 로 Partiton을 찾은 후, Partition 내에서 Join 수행
   ④ Memory 내에 존재하지 않는 Partition 일 경우
      - Hash Value 로 최초 Hash Table 생성 시 Temporary 에 만든 Partition 과 연계해서 Probe Partiton생성 (On disk)

V-4. Sort Merge Join

- ❑ 전체범위처리
- ❑ SORT_AREA_SIZE 의 크기에 따라 성능 결정.
- ❑ 대량 데이터의 SORT에 Direct I/O 유발(Temp 사용).
- ❑ sort_multiblock_io_count 개수만큼의 블록들을 동시에 read/write 가능(Sort Runs)
- ❑ USE_MERGE 힌트
- ❑ CARTESIAN 곱, BETWEEN JOIN 시 많이 발생함
- ❑ 결과값에 정렬을 요구하지 않는 경우 HASH JOIN을 이용하는 것이 Performance 에 유리
- ❑ 단순 MERGE JOIN시 선행 TABLE을 SORT하지 않으므로 큰 TABLE을 선행 TABLE로 하는 것이 유리

```
Execution Plan
----------------------------------------
 SELECT STATEMENT Optimizer=CHOOSE
   MERGE JOIN (CARTESIAN)
     TABLE ACCESS (FULL) OF 'SALE'
     SORT (JOIN)
       TABLE ACCESS (FULL) OF 'CUSTOMER'
```

V-4. Sort Merge Join




```
SELECT  /*+ USE_MERGE(A B)  */
          a.saledate, a.name,
          b.item, b.item_level
FROM    sale a
          ,item b
WHERE a.saledate like '200304%'
AND     a.name    = b.name
```

```
EXPLAIN PLAN
-------------------------------
SELECT STATEMENT
  MERGE JOIN
    SORT (JOIN)
      TABLE ACCESS (BY INDEX ROWID) :SALE
        INDEX (RANGE SCAN) :SALE_N1
    SORT (JOIN)
      TABLE ACCESS {{ FULL }} :ITEM
```

V-5. Join 방법 비교 평가

	Nested Loop	Hash	Sort Merge
처리 방식	순차적(완전 부분범위)	반 부분범위	동시적(전체범위처리)
Access 방식	Random Access	Hash Function	Scan 방식
연결 고리	절대 영향	영향 없음	영향 없음
Join 방향	영향 큼	영향 있음 (Hash Table 구성)	영향 없음
사용 resource	BUFFER CACHE	PGA	PGA
처리량	좁은 범위에 유리	넓은 범위에 유리	넓은 범위에 유리
주요 Check 요소	연결 고리 상태 및 처리량	Hash_area_size Hash Table size	Sort_area_size 각 Table 의 Sort 량 (Temp 사용량)

- ❑ Nested Loop Join은 대용량의 data에서 random access 매우 취약.
- ❑ Hash Join은 Sort Merge Join에 비해서 거의 모든 면에서 유리.

V-6. Special Joins

Outer Join

- 조인조건에 만족되지 않더라도 결과에 포함시키기 위한 특별한 조인기법
- 조인순서가 미리 정해지므로 조인순서를 이용한 튜닝이 불가. 가능하다면 outer join 회피 필요.
- ORDERED 힌트가 outer join 순서에 위배된다면 무시됨.
- outer join을 당하는 테이블에 대한 모든 조건에 (+) 기호를 붙여야 원하는 결과를 얻는다.
- (+) 기호를 이용하여 IN, OR의 연산자를 이용하여 비교할 수 없다.
  Inline View를 통해서 해결 또는 DECODE() 함수 이용 해결.
- (+) 기호를 이용하여 Subquery와 비교할 수 없다. IS NULL OR 조건과 같이 비교한다.
- (+) 기호 일부 누락시 RBO는 OUTER JOIN + FILTER로 처리
  CBO는 NORMAL JOIN 으로 처리한다.

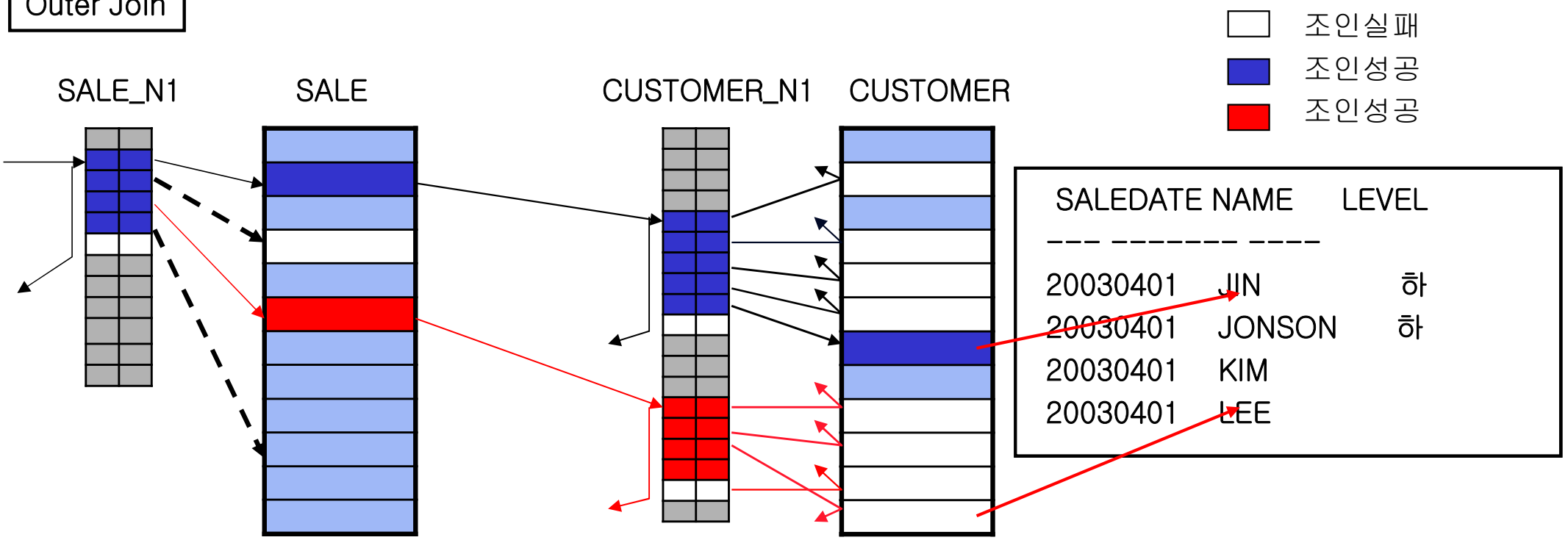
```
SELECT d.dname, e.ename, e.job
FROM   dept d, emp e
WHERE  d.deptno = e.deptno(+)
AND    e.job(+) in ('MANAGER','CLERK') ;

ORA-01719: outer join operator (+) not allowed in operand of OR or IN
```

- 수정안

```
SELECT d.dname, e.ename, e.job
FROM   dept d, (SELECT * FROM emp WHERE job in ('MANAGER','CLERK')) e
WHERE  d.deptno = e.deptno(+) ;
```

V-6. Special Joins

Outer Join



SALEDATE	NAME	LEVEL
20030401	JIN	하
20030401	JONSON	하
20030401	KIM	
20030401	LEE	

```
SELECT a.saledate, a.name,
          b.customer_level level
FROM    sale a
          ,customer b
WHERE a.saledate like '200304%'
AND     a.name      =  b.name(+)
AND     b.customer_level (+) =  '하'
```

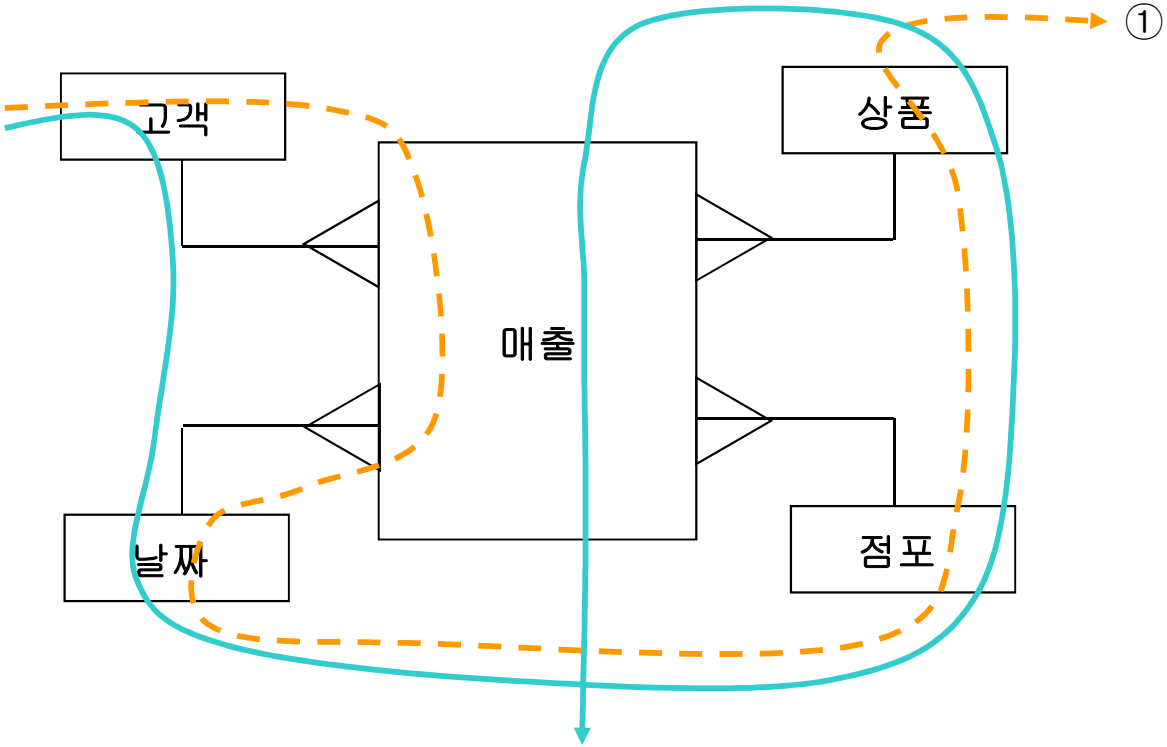
```
EXPLAIN PLAN
-----------------------
SELECT STATEMENT
  NESTED LOOPS (OUTER )
    TABLE ACCESS (BY INDEX ROWID) :SALE
      INDEX (RANGE SCAN) :SALE_N1
    TABLE ACCESS (BY INDEX ROWID) :CUSTOMER
      INDEX (RANGE SCAN) :CUSTOMER_N1
```

join 순서고정
A -> B

V-6. Special Joins

Star Join

DW 등의 환경에서 Star Schema 상의 다수의 디멘젼(dimension) 테이블들과 한 개의 대용량 팩트(fact) 테이블을 조인하는 과정에서 디멘젼들을 Catesian Product로 먼저 조인한 후에 그 결과를 이용하여 팩트 테이블을 Nested Loops로 조인함으로써 대용량 팩트 테이블의 조인의 연결회수를 줄이고자 하는 조인 방법.



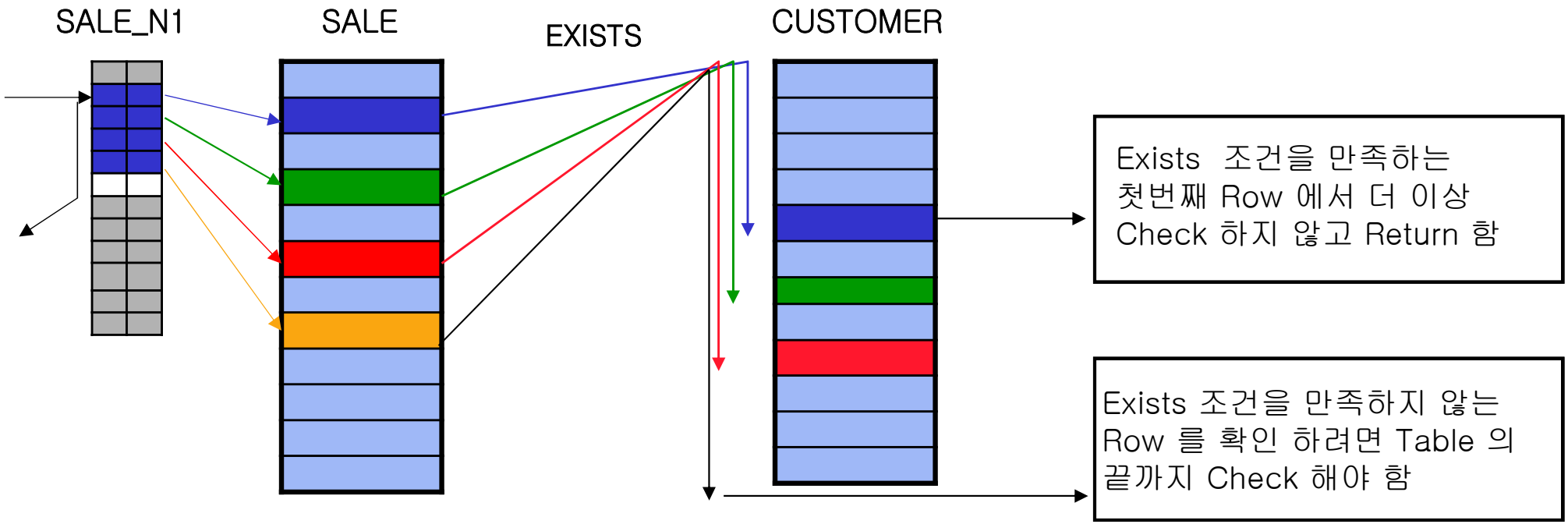
① 보통의 조인

```
SELECT  SUM(매출.금액)
FROM    고객, 매출, 상품, 날짜, 점포
WHERE   고객.성별 = '여'
AND     상품.분류 = '의류'
AND     점포.주소 LIKE '서울시%'
AND     날짜.요일 = '일요일'
AND     고객.고객번호 = 매출.고객번호
AND     날짜.일자 = 매출.매출일자
AND     점포.점포번호 = 매출.점포번호
AND     상품.상품코드 = 매출.상품코드 ;
```

- ❑ Optimizer가 CBO인 상태에서,
- ❑ SQL에 /*+ STAR */ 힌트 사용

V-6. Special Joins

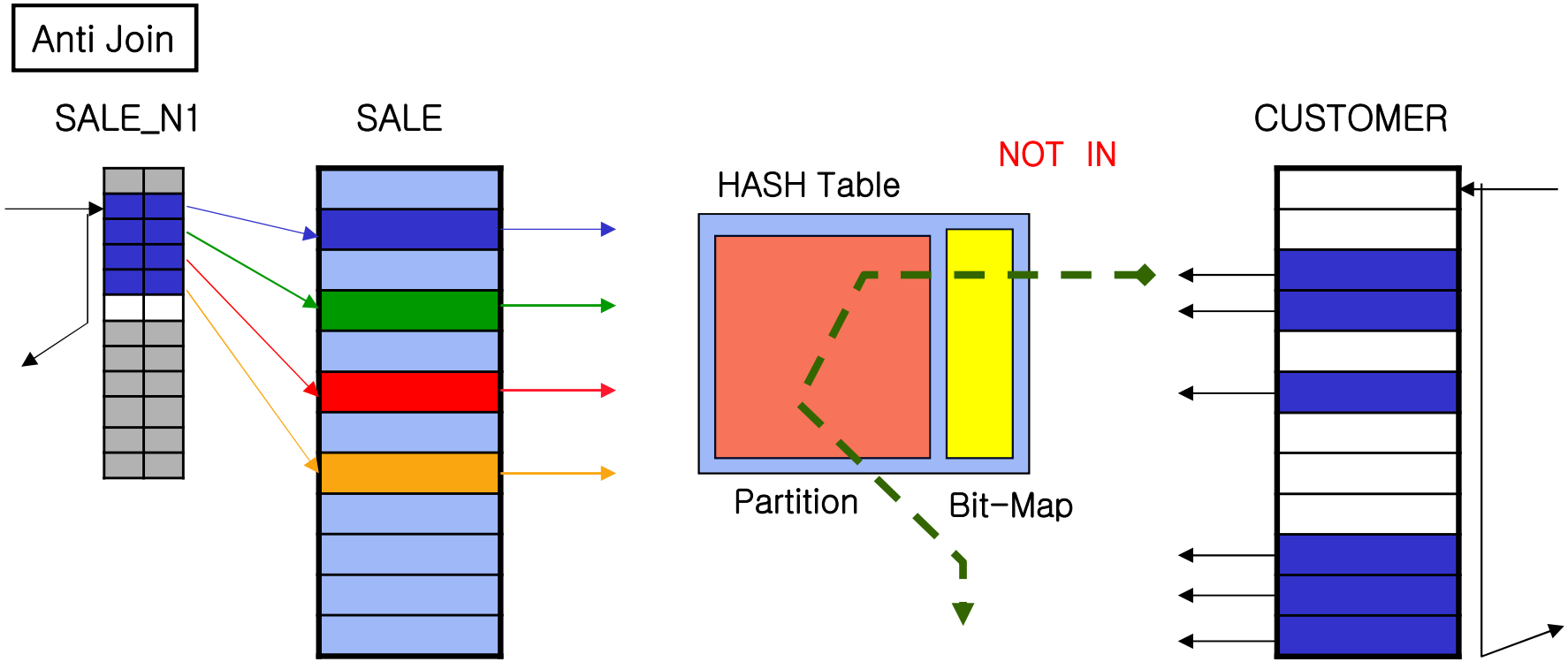
Semi Join




```
SELECT a.saledate, a.name
FROM    sale a
WHERE a.saledate like '200304%'
AND     EXISTS( SELECT 1
                FROM   customer b
                WHERE b.customer_level = '상'
                AND    b.name = a.name )
```

```
EXPLAIN PLAN
-----------------------------------
SELECT STATEMENT
  FILTER
    TABLE ACCESS (BY INDEX ROWID) :SALE
      INDEX (RANGE SCAN) :SALE_N1
    TABLE ACCESS ( FULL ) :CUSTOMER
```

V-6. Special Joins




```
SELECT  a.name ,a.saledate
FROM     sale a
WHERE   a.saledate like '200304%'
AND a.name  not in (  SELECT /*+ HASH_AJ */
                                    b.name
                           FROM  customer b
                           WHERE b.customer_level = '상'
                           AND b.name is not null)
 AND a.name is not null   ;
```

```
EXPLAIN PLAN
----------------------------------
SELECT STATEMENT
  HASH JOIN (ANTI)
    TABLE ACCESS (BY INDEX ROWID) :SALE
      INDEX (RANGE SCAN) :SALE_N1
    VIEW :VW_NSO_1
      TABLE ACCESS ( FULL) :CUSTOMER
```

VI. Subquery

1. **Subquery**의 이해
2. **In**
3. **Subquery**의 선/후 실행비교
4. **NOT IN**
5. **EXISTS**
6. **FILTER**의 **One Buffer** 효과
7. **NOT EXISTS**
8. **Join**을 **Subquery**로 변환

VI-1. Subquery의 이해

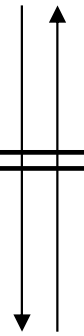
- 집합 연결시 조인은 같은 level 이지만 서브쿼리는 주종관계
- 선택적으로 Join 가능 (Exists)
- 메인쿼리의 속성은 서브쿼리에 상속가능
- 서브쿼리의 속성을 메인쿼리의 SELECT – LIST 에 제공 못함
- 서브쿼리의 결과값은 연결 속성에만 제공 가능
- 서브쿼리의 선실행, 후실행에 따른 차이가 크다.
- M:M 조인의 해결(서브쿼리는 항상 조인 관계상 1쪽 집합)
- 상수값처럼 사용가능
- 서브쿼리가 논리적으로 유일하다면 조인과 동일한 방법으로 처리
- 다양한 ACCESS 형태로 처리 결과의 예측이 어렵다.(PLAN 확인 필)
- SubQuery UnNesting 기능 추가(8i 이후).

```
SELECT  a.empno, b.loc
FROM    emp a
WHERE  a.job = 'SALES'
AND     a.ename in (SELECT b.dname
                    FROM dept b
                    WHERE a.deptno = b.deptno)
```

VI-2. In

- IN subquery는 join으로 수행되는데 연결고리의 index여부에 따라서 선/후 실행이 결정됨
- IN subquery를 선실행 시킬것인지 후실행 시킬것인지에 따라서 IN , EXIST를 선별적으로 사용

```
SELECT    empno, ename
FROM      emp e
WHERE     deptno IN (SELECT deptno FROM dept WHERE loc = 'CHICAGO');

 Execution Plan
----------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE
  NESTED LOOPS
    TABLE ACCESS (BY INDEX ROWID) OF 'DEPT'
      INDEX (RANGE SCAN) OF 'DEPT_IDX1' (NON-UNIQUE)
    TABLE ACCESS (BY INDEX ROWID) OF 'EMP'
      INDEX (RANGE SCAN) OF 'EMP_IDX' (NON-UNIQUE)
```



```
SELECT   empno, ename
FROM     emp e
WHERE  EXISTS (SELECT 1 FROM dept a
                    WHERE loc = 'CHICAGO' AND e.deptno = a.deptno)

  Execution Plan
  ----------------------------------------------------
  SELECT STATEMENT Optimizer=CHOOSE
    FILTER
      TABLE ACCESS (FULL) OF 'EMP'
      TABLE ACCESS (BY INDEX ROWID) OF 'DEPT'
        INDEX (RANGE SCAN) OF 'DEPT_IDX1' (NON-UNIQUE)
```

VI-2. In – 제공자 서브쿼리.

제공자 서브쿼리 처리

❑ 제공자 서브 쿼리

- 서브쿼리에 메인 쿼리의 조건이 없어야 한다.(논리적 필수조건)
- 서브쿼리에 연결된 메인쿼리 컬럼에 인덱스 존재하여 드라이빙 조건으로 사용 가능시.
- 서브쿼리 집합은 메인쿼리 집합 레벨을 변경할 수 없으므로 UNIQUE 해야 한다. → SORT(UNIQUE) 발생.

```
SELECT m.COL1, m.COL2
  FROM TAB1 m
 WHERE ( KEY1, KEY2 ) IN
       ( SELECT COL11, COL12
           FROM TAB2 s
          WHERE s.COL1 = :Val1
            AND s.COL2 = :Val2 )
```

```
NESTED LOOPS
 VIEW
  SORT(UNIQUE)
   TABLE ACCESS BY ROWID OF TAB2
    INDEX RANGE SCAN OF TAB2_IDX1
 TABLE ACCESS BY ROWID OF TAB1
  INDEX RANGE SCAN OF TAB1_IDX1
```

❑ 주의 사항.

- 서브쿼리 제공 조건절이 인덱스 선두 컬럼이 아닌 경우 (예: KEY1 + KEY2로 구성시)

  선두컬럼이 '=' 조건(점조건)이 아닌 경우 선두 컬럼만 드라이빙 조건으로 처리되고

  서브쿼리는 확인자로 처리되어 반복 수행하게 된다.

  즉 서브쿼리에서 제공된 조건값 KEY2 값의 수만큼 KEY1 범위조건을 반복 수행한다.

  (일종의 인덱스 카테시안이 발생)

- 선두 컬럼의 조건을 서브쿼리에서 같이 제공 할 수 있어야 한다.

VI-2. In – 제공자 서브쿼리.

확인자 서브쿼리 처리

❑ 확인자 서브 쿼리

- 서브쿼리에 메인 쿼리의 조건이 있는 경우 (Corelated SubQuery).
- EXISTS 서브 쿼리 등.

❑ 확인자 서브쿼리의 수행.

- FILTER로 처리 (NL 방식).
- SORT MERGE 조인 방식으로 처리.(연결조건에 모두 인덱스가 없는 경우).
- HASH SEMI JOIN, HASH ANTI JOIN 등의 처리.

  → NL FULL SCAN 방지.

부정형 조인

❑ 처리 범위로 방식 결정.

- FILTER 방식의 수행에서 서브쿼리 수행횟수가 과다 하거나 수행범위가 넓은 경우 비효율 발생.
- HASH_ANTI, HASH_SEMI, MERGE_ANTI, MERGE_SEMI 조인으로 처리하여 개선 가능.

VI-3. Subquery의 선/후 실행비교

subquery의 후 실행 trace

```
SELECT *
FROM  sale
WHERE saledate LIKE '200201%'
AND   cust_code IN (SELECT cust_code
                    FROM   customer
                    WHERE  cust_code BETWEEN 'C30100' AND 'C30200')

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.01       0.01          0          0          0           0
Execute      1         0          0          0          0          0           0
Fetch       63      2.82       2.82          0      12926          0        6200
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       63      2.82       2.82          0      12926          0        6200

Rows     Execution Plan
-------  ---------------------------------------------------
          SELECT STATEMENT
   6200     NESTED LOOPS
   6200      TABLE ACCESS BY INDEX ROWID :SALE
  24401       INDEX RANGE SCAN :SALE_CUST_SALEDATE_IDX(NU)
   6200      INDEX UNIQUE SCAN :PK_CUSTOMER (U)
```

VI-3. Subquery의 선/후 실행비교

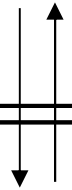
subquery의 선 실행 trace

```
SELECT *
FROM  sale
WHERE saledate LIKE '200201%'
AND   cust_code IN (SELECT cust_code
                    FROM   customer
                    WHERE  cust_code BETWEEN 'C30100' AND 'C30200')

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1         0          0          0          0           0           0
Execute      1         0          0          0          0           0           0
Fetch       63      1.02       1.02          0       6610           0        6200
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       63      1.02       1.02          0       6610           0        6200

Rows     Execution Plan
-------  ---------------------------------------------------
          SELECT STATEMENT
   6200     NESTED LOOPS
    101      VIEW SYS.
    101       SORT UNIQUE
    102        INDEX RANGE SCAN :PK_CUSTOMER (U)
   6200      TABLE ACCESS BY INDEX ROWID :SALE
   6301       INDEX RANGE SCAN :SALE_CUST_SALEDATE_IDX(NU)
```

VI-4. NOT IN

❑ 대량의 data를 not in으로 연결시 hash anti join 고려

```
SELECT     empno, ename
FROM       emp e
WHERE      deptno NOT IN (SELECT deptno FROM dept WHERE loc = 'CHICAGO');

SELECT STATEMENT Optimizer=CHOOSE
  FILTER
    TABLE ACCESS (FULL) OF EMP
    TABLE ACCESS (BY INDEX ROWID) OF DEPT
      INDEX (RANGE SCAN) OF DEPT_PK (UNIQUE)
```



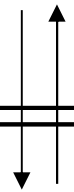
```
SELECT     empno, ename
FROM       emp e
WHERE      deptno NOT IN (SELECT /*+ HASH_AJ */ deptno FROM dept WHERE loc = 'CHICAGO')
AND        deptno IS NOT NULL;

SELECT STATEMENT Optimizer=CHOOSE
  HASH JOIN (ANTI)
    TABLE ACCESS (FULL) OF EMP
    VIEW
      TABLE ACCESS (BY INDEX ROWID) OF DEPT
        INDEX (RANGE SCAN) OF DEPT_LOC_IX (NON-UNIQUE)
```

❑ always_anti_join = NESTED_LOOPS / HASH / MERGE

VI-5. EXISTS

❑ 대량의 data를 exists으로 연결시 hash semi join 고려

```
SELECT    deptno, dname
FROM      dept d
WHERE     EXISTS (SELECT * FROM emp WHERE deptno = d.deptno AND comm > 500);

 SELECT STATEMENT Optimizer=CHOOSE
   FILTER
     TABLE ACCESS (FULL) OF DEPT
     TABLE ACCESS (BY INDEX ROWID) OF EMP
       INDEX (RANGE SCAN) OF EMP_DEPTNO_IX (NON-UNIQUE)
```



```
 SELECT    deptno, dname
 FROM      dept d
 WHERE EXISTS (SELECT /*+ HASH_SJ*/ * FROM emp WHERE deptno = d.deptno AND comm > 500);

 SELECT STATEMENT Optimizer=CHOOSE (Cost=3 Card=1 Bytes=48)
   HASH JOIN (SEMI) (Cost=3 Card=1 Bytes=48)
     TABLE ACCESS (FULL) OF 'DEPT' (Cost=1 Card=41 Bytes=902)
     TABLE ACCESS (FULL) OF 'EMP' (Cost=1 Card=3 Bytes=78)
```

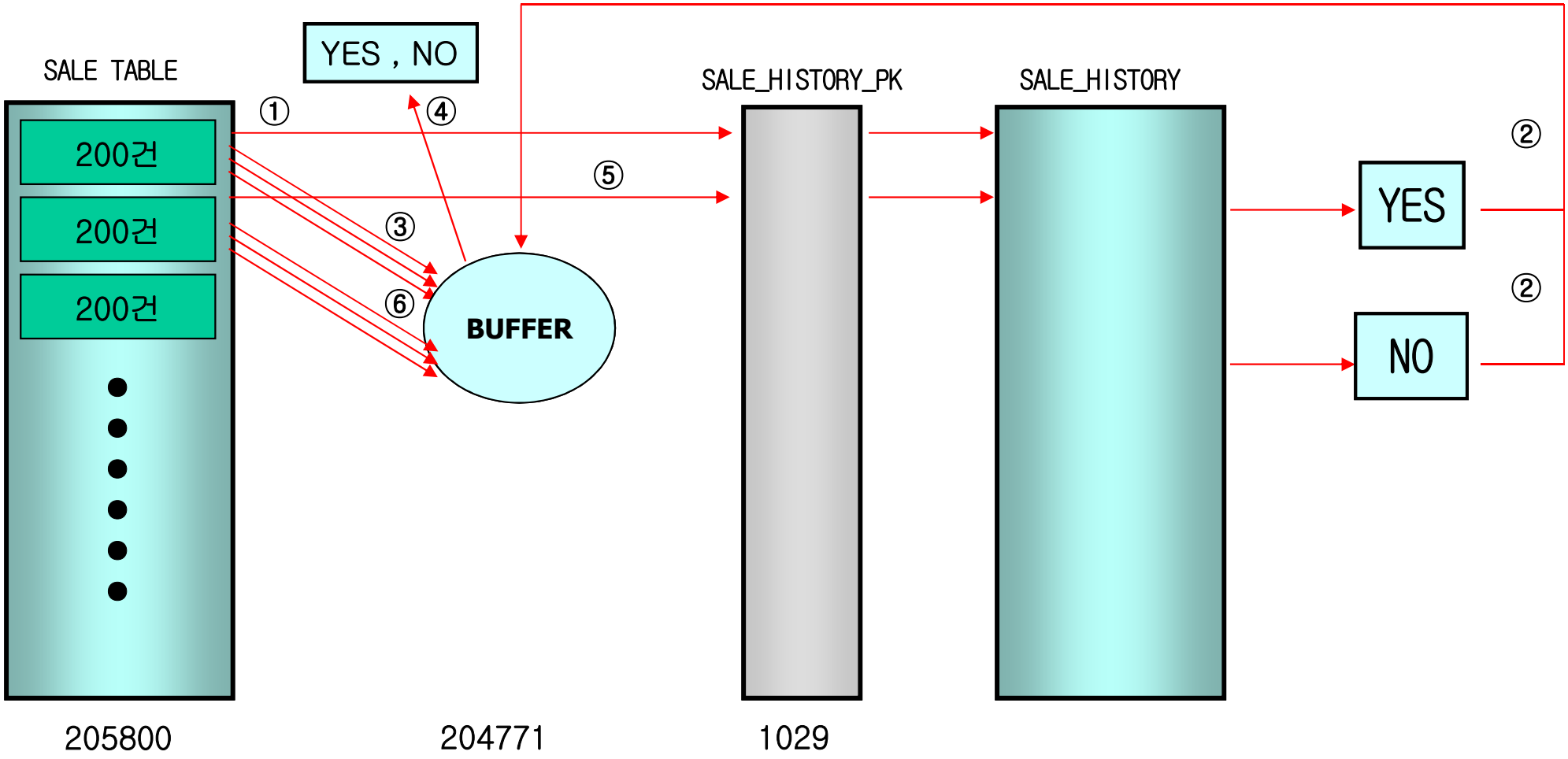
❑ always_semi_join = NESTED_LOOPS / HASH / MERGE

VI-6. FILTER의 One Buffer 효과

- ❑ Subquery 사용시 자주 발생
- ❑ Table의 Clustering Factor에 따라 속도가 좌우
- ❑ 자주 access되는 table들은 정기적으로 reorg를 할 필요가 있음

```
SELECT a.*
FROM   sale a
WHERE  EXISTS (SELECT 'x'
               FROM   sale_history b
               WHERE  a.saledate = b.saledate)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch     2059     14.39      14.44          3       9113         11      205800
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total     2059     14.39      14.44          3       9113         11      205800

Rows     Execution Plan
-------  ---------------------------------------------------
         SELECT STATEMENT
 205800     FILTER
 205800       TABLE ACCESS FULL :SALE
   1029       INDEX RANGE SCAN :PK_SALE_HISTORY (U)
```

VI-6. FILTER의 One Buffer 효과



❑ SALE TABLE의 1029 * 200 = 205800 건중에서 실제 SUBQUERY 조건인 SALE_HISTORY INDEX까지 조회하러 가는 건수는 1029 건이라고 말 할 수있다.

❑ 나머지 205800 – 1029 = 204771건은 BUFFER에서 FILTER 처리 되었다.

VI-6. Subquery Multi Buffer 처리.

❑ Oracle 8i 이후 CBO 옵티마이져에 추가된 기능.

❑ SubQuery를 UnNesting 으로 처리. (Scalar Expression 처리로 가능.)

```
select e.* from scott.emp e
 where exists ( select 'OK' from scott.dept d where d.deptno = e.deptno )

Rows     Row Source Operation
-------  ---------------------------------------------------
     14  NESTED LOOPS SEMI
     14   TABLE ACCESS FULL EMP
     14   INDEX UNIQUE SCAN PK_DEPT (object id 30138)
******************************************************

select e.* from scott.emp e
 where exists ( select 'OK' from scott.dept d where d.deptno = e.deptno )

Rows     Row Source Operation
-------  ---------------------------------------------------
     14  FILTER
     14   TABLE ACCESS FULL EMP
      3   INDEX UNIQUE SCAN PK_DEPT (object id 30138)
******************************************************
```

→ UnNest 처리 미 발생시 유도 가능.

```
select e.*
  from scott.emp e
 where ( select 'OK' from scott.dept d where d.deptno = e.deptno ) = 'OK'
```

VI-7. NOT EXISTS

❑ NOT EXISTS와 NOT IN은 서로 같은 의미의 query

```
SELECT deptno, dname
FROM   dept d
WHERE NOT EXISTS (SELECT * FROM emp WHERE deptno = d.deptno AND comm > 500)

 Execution Plan
 ----------------------------------------------------
 SELECT STATEMENT Optimizer=CHOOSE
   FILTER
     TABLE ACCESS (FULL) OF 'DEPT'
     TABLE ACCESS (BY INDEX ROWID) OF 'EMP'
       INDEX (RANGE SCAN) OF 'EMP_DEPTNO_IX' (NON-UNIQUE)

SELECT deptno, dname
FROM    dept d
WHERE  deptno NOT IN (SELECT /*+ hash_aj */ deptno FROM emp where comm > 500)

Execution Plan
----------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE
  HASH JOIN (ANTI)
    TABLE ACCESS (FULL) OF 'DEPT'
    VIEW OF 'VW_NSO_1'
      TABLE ACCESS (FULL) OF 'EMP'
```

VI-7. NOT EXISTS

❑ NOT EXISTS를 OUTER JOIN을 이용하여 Parallel Query로도 가능

```
SELECT /*+ use_hash(d f) parallel(d 4) */
          d.deptno, d.dname
FROM    dept d, (SELECT distinct deptno
                 FROM    emp
                 WHERE  comm > 500) f
WHERE d.deptno = f.deptno(+)
AND f.deptno is null


Execution Plan
----------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE
  FILTER
    HASH JOIN (OUTER)
      TABLE ACCESS (FULL) OF 'DEPT'
      VIEW
        SORT (UNIQUE)
          TABLE ACCESS (FULL) OF 'EMP'
```

VI-8. Join을 Subquery로 변환

❑ 연봉이 2,000 이상인 사원이 있는 부서명을 출력하라.

< 조인의 경우 >

```
SELECT distinct dname
FROM   dept d, emp e
WHERE  d.deptno = e.deptno
AND    e.sal > 2000 ;
```

```
Rows  Row Source Operation
----  -------------------------------------
   3  SORT UNIQUE
  14   NESTED LOOPS
  15    TABLE ACCESS FULL EMP
  14    TABLE ACCESS BY INDEX ROWID DEPT
  28     INDEX UNIQUE SCAN (object id 3392)
```

< Subquery의 경우 >

```
SELECT dname
FROM   dept d
WHERE  EXISTS (SELECT empno
               FROM   emp e
               WHERE  d.deptno = e.deptno
               AND    e.sal > 2000) ;
```

```
Rows  Row Source Operation
----  -------------------------------------
   3  FILTER
   5   TABLE ACCESS FULL DEPT
   4   TABLE ACCESS BY INDEX ROWID EMP
   4    INDEX RANGE SCAN (object id 3404)
```

VI-9. Scalar Subquery의 이해

❑ 스칼라(Scalar)란?

- 방향을 가지지 않는 하나의 값.

❑ 스칼라 서브쿼리(Scalar Subquery)란?

- SQL에서 컬럼이 사용될 수 있는 모든 곳에 사용 가능한 SubQuery 이다.

- 하나의 컬럼, 하나의 로우 즉 하나의 값(Scalar)만을 반환 가능하므로 InLine Function의 의미를 가진다.

❑ Oracle 8i부터 사용 가능하며 9i 에서는 SQL의 모든 곳에서의 사용이 가능.

❑ 실행계획.

- Oracle 9.2.0.3 부터 Scalar Subquery의 실행계획이 표현 가능하다. (버그:1721097)

❑ Scalar Expression 은 UnNesting으로 처리된다.

❑ 조인 대체.

- 명칭 참조와 같은 코드성 조인의 대체. → From 절 감소로 실행계획 제어 유연해 진다.

- 기본적으로 NL 방식 수행 → 대량의 처리시에는 MERGE, HASH 방식이 유리하다.

❑ 함수 대체.

- InLine 함수로 사용하여 Stored Function 미 생성 처리 및 다양한 사용자 함수 사용 요법 가능.

❑ 서브쿼리 대체.

- 1 : M 처리나 Arc Join 등을 대신하여 처리 가능.

❑ 오브젝트 타입의 사용.

VI-9. Scalar Subquery의 이해-실행계획.

❑ 실행계획 예)

```
SELECT E.EMPNO, E.ENAME, E.JOB, E.MGR ,
       NVL( ( SELECT MAX('U')
                FROM SCOTT.DEPT D
               WHERE D.DEPTNO BETWEEN E.DEPTNO AND E.DEPTNO + 1
                 AND ROWNUM <= 1
         ), 'I' ) FLAG
 FROM SCOTT.EMP E
WHERE E.EMPNO > 7600  ;

SELECT STATEMENT Optimizer Mode=CHOOSE
  SORT AGGREGATE
    COUNT STOPKEY
      INDEX RANGE SCAN PK_DEPT
  TABLE ACCESS BY INDEX ROWID   EMP
    INDEX RANGE SCAN   PK_EMP
```

❑ Scalar Expression 의 실행계획.

- 스칼라 서브쿼리는 전체 SQL 중 어느 곳에도 기술가능하며 단지 최종 SELECT절에서 결과값으로 기술시에만 스칼라 서브쿼리 플랜으로 표현된다.
- 최종 SELECT절 이외 WHERE절에서 조건으로 기술시, 또는 INLINE-VIEW안의 스칼라 서브쿼리가 뷰밖에서 WHERE절의 조건으로 처리시에는 FILTER로 표현된다.
- 플랜 및 트레이스 에서는 가장 상위 단계 즉 최상위 레벨에 표현된다.
- 스칼라 서브쿼리는 플랜, 트레이스 에서 생략 가능하다.
  NLV() 절 기술시, INLINE-VIEW안에서 ROWNUM 기술시 등등.
- 스칼라 서브쿼리의 결과는 결과값으로 처리 된다.

VII. ETC

1. **Parallel SQL**
2. **Distributed SQL**
3. **Inline View**
4. **Function Call**
5. **SQL** 활용기법.
6. **New Feature.**

VII-1. Parallel SQL

Parallel SQL 사용 필요 조건

- ❑ Server computer has multiple CPUs (CPU * 2 = max parallel process)
- ❑ The data to be accessed is on multiple disk drives
- ❑ System capability에 여유가 있을 때 사용
- ❑ Long Runing SQL or Resource intensive SQL

Parallel SQL 사용 가능 작업

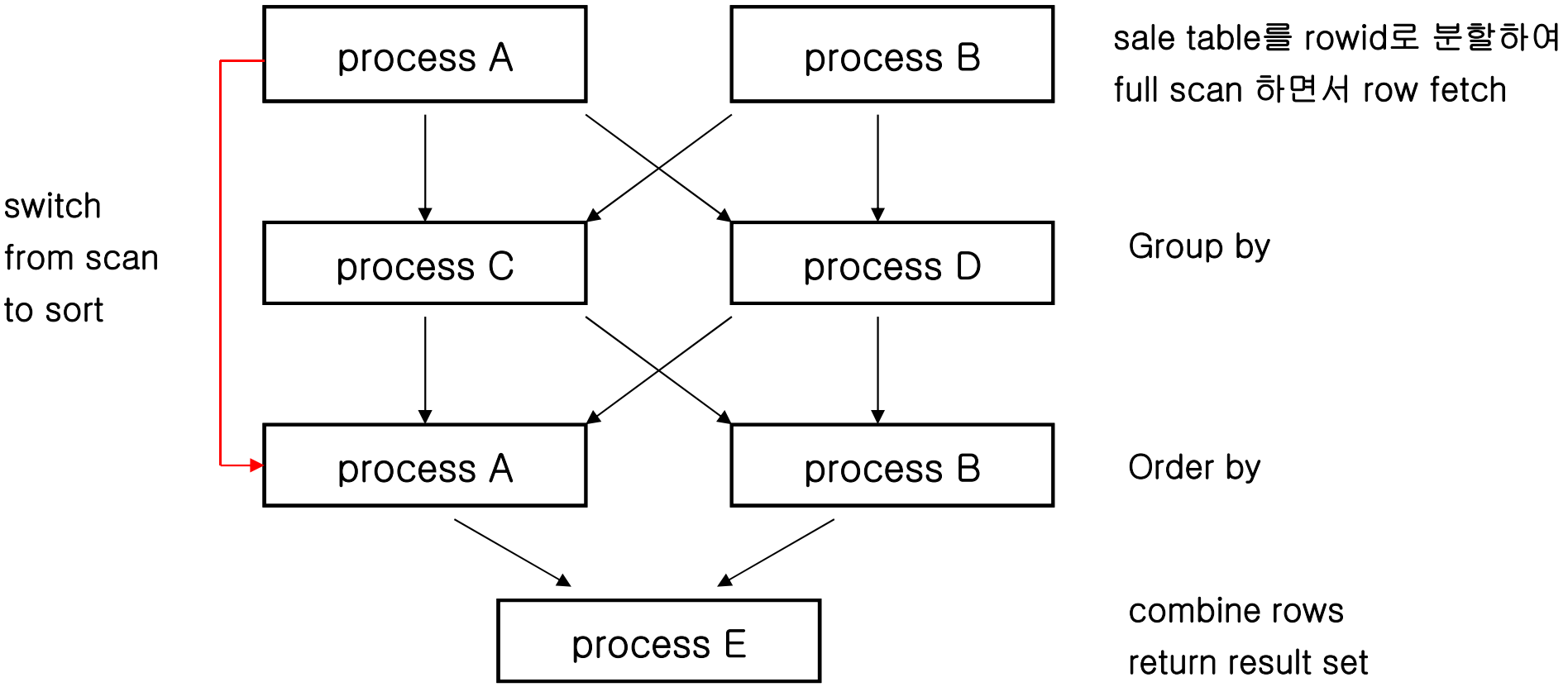
- ❑ 1개의 full table scan 작업이 포함되어 있는 SQL
- ❑ Building or Rebuilding an index
- ❑ Local index를 사용하는 partitioned table access
- ❑ analyze 작업
- ❑ Parallel DML (full table scan을 포함하는 작업)
  - Oracle 8이하
    select만 parallel로 실행
  - Oracle 8i
    non-partition table insert 문의 경우는 parallel로 insert하지만 delete, update의 경우는 select만 parallel로 방식으로 실행
  - Oracle 9i
    insert,update,delete가 모두 parallel 실행가능

VII-1. Parallel SQL

```
SELECT /*+ parallel(a 2) full(a) */
       sale_class, sum(sale_price)
FROM   sale a
GROUP BY sale_class
ORDER BY 2
```

❑ session의 sort_area_size가 10M이면 각각 parallel slave process 도 10M씩 할당을 받는다. 옆에 SQL에서는 총 5개의 process가 50M를 사용하게 된다.

❑ Parallel slave process 는 parallel degree에 2배 이상 slave process 가 사용될 수 없다



VII-1. Parallel SQL

Parallel SQL Tuning

```
SELECT /*+ parallel(b 2) parallel(c 2)  */
          customer_name, product_description, sum(sale_value)
 FROM   sale a, customers b, products c
 WHERE a.customer_id = b.customer_id
 AND     a.product_id = c.product_id
 GROUP BY b.customer_name, c.product_description

Explain Plan
----------------------------------------------------------
SELECT STATEMENT
      SORT GROUP BY   (PARALLEL_TO_SERIAL)
         SORT GROUP BY   (PARALLEL_TO_PARALLEL)
            HASH JOIN   (PARALLEL_COMBINED_WITH_PARENT)
               TABLE ACCESS FULL PRODUCTS (PARALLEL_TO_PARALLEL)
                HASH JOIN   (PARALLEL_TO_PARALLEL)
                    TABLE ACCESS FULL CUSTOMERS  (PARALLEL_TO_PARALLEL)
                    TABLE ACCESS FULL SALES(PARALLEL_TO_SERIAL)
```

VII-2. Distributed SQL

❑ Database link를 통해서 Remote DB에 연결하여 처리

```
SELECT a.saledate, a.sale_amt, b.item_name
 FROM    sale a, item@node2 b
 WHERE  ....
```

LOCAL REMOTE

❑ 모든 table이 remote DB에 있다면, 전체 SQL를 remote DB로 보내 처리 (Remote SQL)

❑ 여러 remote DB에 table이 분산되어 있다면 Optimizer가 결정 (Distribute SQL)

- Remote table의 통계정보 수집
- Network costs 계산

❑ Optimizer가 execute plan 생성시 remote DB에 있는 table들을 local DB 처럼 생각하여 Execute Plan을 생성할 경우 과도한 Network Traffic이 발생할 수 있음

Distributed SQL tuning 방안

❑ 적절한 driving site 선택
- CPU intensive한 작업인 join이나 sort 작업이 일어나는 site
- system capability가 높거나, oracle version이 높은 site
- DRIVING_SITE(table_alias) hint 사용
- 서브쿼리나 리모트 사이트간의 조인 작업의 처리 순서 확인.

VII-2. Distributed SQL

Distributed SQL Tuning 방안

❑ 적절한 Remote site or Local site에 View 생성

```
CREATE VIEW sale_and_item AS
SELECT e.surname, e.employee_id,....
FROM    item c, sale e
WHERE  c.sales_id = e.employee_id
```

```
SELECT d.depart_name,....
FROM    sale_and_item@node2 e,
        department d@node1
WHERE  d.department_id = e.department_id
AND      d.department_name = :1
```

```
Execution Plan
----------------------------------
SELECT STATEMENT
  MERGE JOIN
    SORT (JOIN)
      NESTED LOOPS
        REMOTE [NODE1]
         ....
        REMOTE [NODE2]
         ...
     SORT(JOIN)
         ...
```



```
Execution Plan
----------------------------------
SELECT STATEMENT
  HASH JOIN
    REMOTE [NODE1]
      SELECT  ...
          FROM DEPARTMENT D
        WHERE DEPARTMENT_NAME = :1
    REMOTE [NODE2]
      SELECT ...
          FROM SALE_AND_ITEM E
     ...
```

VII-3. Inline View

Inline View

- ❑ InLine View는 SQL Block 안에 Static View와 같은 의미.
- ❑ Inline View의 집합 레벨이 달라지지 않는다면 Main Query Block과 통합된다.
- ❑ Inline View의 집합 레벨이 변경된다면 처리 결과가 임시집합으로 처리된다.
- ❑ Inline View로의 외부조건 적용은 Static View와 같다.

VII-3. Inline View

Inline View Join시 Group By 활용

❑ 조인을 수행한 후에 group by하는 쿼리는 인라인뷰를 이용하여 조인 연결회수를 줄일 수 있다.

❑ 단, 조인 연결 컬럼이 group by 컬럼의 앞부분으로 작용하는 경우에만 해당됨.

```
SELECT d.dname, sum(e.sal)
FROM   dept d, emp e
WHERE  d.deptno = e.deptno
GROUP BY d.dname ;
```

```
Rows  Row Source Operation
----  -------------------------------------
   3  SORT GROUP BY
  14   NESTED LOOPS
  15    TABLE ACCESS FULL EMP
  14    TABLE ACCESS BY INDEX ROWID DEPT
  28     INDEX UNIQUE SCAN (object id 3392)
```

```
SELECT d.dname, ev.sum_sal
FROM  (SELECT deptno, sum(sal) sum_sal
       FROM   emp
       GROUP BY deptno) ev, dept d
WHERE  ev.deptno = d.deptno ;
```

```
Rows    Row Source Operation
----    -------------------------------
   3    NESTED LOOPS
   4     VIEW
   4      SORT GROUP BY
  14       TABLE ACCESS FULL EMP
   3     TABLE ACCESS BY INDEX ROWID DEPT
   6      INDEX UNIQUE SCAN (object id 3392)
```

VII-3. Inline View

Inline View 내 Order By

❑ Inline View 내에서 order by를 사용함으로써 Top-N 개의 로우를 추출
❑ Oracle 8i 부터 가능
❑ order by 컬럼에 인덱스가 없는 경우
❑ order by 컬럼이 결합된 컬럼인 경우

```
SELECT empno, sal
FROM   (SELECT /*+ INDEX(EMP EMP_SAL_IX) */
               empno,sal
        FROM   emp
        WHERE sal > 1000
        ORDER BY sal desc)
WHERE  rownum <= 5 ;
```

❑ Oracle 8i 이하

```
SELECT empno, -sal sal
FROM   (SELECT /*+ INDEX(EMP EMP_SAL_IX) */
               empno,sal*(-1) sal
        FROM   emp
        WHERE sal > 1000
        GROUP BY sal*(-1),empno)
WHERE  rownum <= 5 ;
```

VII-4. Function Call – NULL 처리.

Null과 Aggregation

KEY	COL1	COL2
1	10	NULL
2	NULL	NULL

```
SQL> SELECT col1 + col2 FROM tab WHERE key = 1;
SQL> SELECT sum(col1) FROM tab;
SQL> SELECT sum(col2) FROM tab;
SQL> SELECT sum(nvl(col1,0)) FROM tab;
SQL> SELECT nvl(sum(col1),0) FROM tab;
SQL> SELECT nvl(sum(col2),0) FROM tab;
SQL> SELECT count(col1) FROM tab;
SQL> SELECT count(col2) FROM tab;
SQL> SELECT avg(col1) FROM tab;
SQL> SELECT avg(col2) FROM tab;
```

- NULL은 함수의 연산에 포함되지 않는다.
- NULL은 IS NULL, IS NOT NULL, DECODE(), NULLIF() 함수만 비교 및 처리 가능하다.

```
SELECT * FROM DUAL
DUMMY
-----
X
1 row selected


select 1 from dual
union
select 1 from dual;


select 1 from dual
union all
select 1 from dual;
```

VII-4. Function Call – 반복 수행.

Decode의 효율적 사용

❑ Inline View 내에서 GROUP BY를 사용함으로써 바깥쪽에서의 DECODE 회수를 줄일 수 있다.

```
SELECT count(decode(substr(to_char(hiredate,'yyyymmdd'),5,2),'01',empno)) M01,
       count(decode(substr(to_char(hiredate,'yyyymmdd'),5,2),'02',empno)) M02,
       count(decode(substr(to_char(hiredate,'yyyymmdd'),5,2),'03',empno)) M03,
       count(decode(substr(to_char(hiredate,'yyyymmdd'),5,2),'04',empno)) M04,
       count(decode(substr(to_char(hiredate,'yyyymmdd'),5,2),'05',empno)) M05,
       count(decode(substr(to_char(hiredate,'yyyymmdd'),5,2),'06',empno)) M06,
       count(decode(substr(to_char(hiredate,'yyyymmdd'),5,2),'07',empno)) M07,
       ....
FROM   emp ;
```

❑ 수정안

```
SELECT nvl(sum(decode(MM,'01',cnt)),0) M01, nvl(sum(decode(MM,'02',cnt)),0) M02,
       nvl(sum(decode(MM,'03',cnt)),0) M03, nvl(sum(decode(MM,'04',cnt)),0) M04,
       nvl(sum(decode(MM,'05',cnt)),0) M05, nvl(sum(decode(MM,'06',cnt)),0) M06,
       nvl(sum(decode(MM,'07',cnt)),0) M07, nvl(sum(decode(MM,'08',cnt)),0) M08,
       nvl(sum(decode(MM,'09',cnt)),0) M09, nvl(sum(decode(MM,'10',cnt)),0) M10,
       nvl(sum(decode(MM,'11',cnt)),0) M11, nvl(sum(decode(MM,'12',cnt)),0) M12
FROM  (SELECT substr(to_char(hiredate,'yyyymmdd'),5,2) MM, count(*) cnt
         FROM emp
        GROUP BY substr(to_char(hiredate,'yyyymmdd'),5,2));
```

VII-4. Function Call – 반복 수행.

Stored Function의 수행 횟수

❑ AMT_FUN0이 반복적으로 수행되고 있음

```
SELECT itemno,
       AMT_FUN(itemno,sysdate) ,
       AMT_FUN(itemno,sysdate-1),
      (AMT_FUN(itemno,sysdate) -
       AMT_FUN(itemno,sysdate-1)) * 100 /
       AMT_FUN(itemno,sysdate)
FROM   item
WHERE icode = '110' ;
```

❑ 수정안

```
SELECT itemno,
            tosel, yessel,
           (tosle-yessel)*100 / tosel selratio
FROM ( SELECT itemno,
                  AMT_FUN(itemno, sysdate)    tosel,
                  AMT_FUN(itemno, sysdate-1) yessel
          FROM    item
          WHERE icode = '110'
          GROUP BY itemno  ) ;    <-- GROUP BY 를 사용하면 수행결과가 내부적으로
                                      저장되었다가 제공되므로 한번씩만 수행
```

VII-5. SQL 활용 기법.

Logic 처리

- SQL Function의 조합.
- DECODE, CASE 로 IF 조건 처리.
- FLOOR, CEIL, MOD로 그룹 처리.
- 날짜함수로 날짜 연산 가능.
- COPY_T 테이블 사용으로 데이터 복제- 집계, 컬럼을 로우로 변환 등
- SUM(DECODE()) 사용으로 로우를 컬럼으로 변환.
- UNION ALL + SUM(DECODE())로 상이한 집합의 통합.

  DECODE() 주의 : 단순 비교에 사용, 불필요한 ELSE 조건처리는 제외, 중복 사용시에는 CASE 절 사용.
- DECODE() 내 반복수행 함수는 먼저 처리하여 수행 횟수 감소.
- 문자 함수처리.

  INSTRB(), SUBSTRB(), REPLACE()
- MIN/MAX값의 처리.

  고정길이 형태로 변환하여 한번의 엑세스로 처리 → MAX(처리일자 || RPAD(처리금액,15))

VII-5. SQL 활용 기법.

확장 UPDATE문 사용

❑ 구문.

```
UPDATE TAB1 M
   SET ( m.col1, m.col2, m.col3, ... m.coln ) =          → ⑤ 대상별 UPDATE.
       ( SELECT any_style_logical_column, ...
           FROM TAB3 s1, TAB4 s2, ... ,TABN sn
          WHERE s1.key1  = m.col11                       → ③ 대상별 갱신값의 조건 제공.
            AND s1.jcol1 = s2.jcol1                      → ④ 대상별 갱신값을 조인.
            AND ...      )
WHERE  (M.key1, M.key2,... ) IN                          → ② 갱신 대상 집합 조건으로 드라이빙.
       ( SELECT s.col1, s.col2, .....
           FROM TAB2 s
          WHERE s.col1 = :VAL1 )                         → ① 갱신 대상 집합 조건 제공.
```

❑ 처리 순서.

- ① → ⑤ 까지 순서별로 처리
- SELECT → FETCH → UPDATE 의 건별 처리를 하나의 SQL로 처리.

❑ 주의 사항.

- ③, ④ 번의 조건에는 반드시 인덱스가 존재 하여 이를 이용한 범위 처리로 수행 되어야 한다.
- ② 번의 조건절에 인덱스가 존재하면 인덱스로 처리범위 결정.

❑ ROWID 이용.

- UPDATE 대상과 UPDATE 값과 ROWID를 미리 조인하여 임시집합으로 만들고 이 ROWID를 이용하여 처리.

VII-5. SQL 활용 기법.

데이터 연결 방법.

- 조인
  - Equi Join, Between Join
- UNION ALL + GROUP BY
  - 상이한 집합의 조인, M:M 관계 처리, 양측 아웃터 조인 해결.
- 저장형 함수 사용
  - 부분범위 처리, 1:M 관계 처리, Arc 관계 해소.
- 서브쿼리
  - 조건 체크, 조건 제공.
- 스칼라 서브쿼리
  - 저장형 함수, 서브쿼리의 모든 경우 가능

VII-5. SQL 활용 기법.

다양한 기법들.

❑ GROUP BY와 AGGRIGATE FUNCTION의 차이.

- Group By 절은 조건절의 결과를 Group By 하여 결과 집합 생성 → 조건결과가 없으면 결과도 없다.
- AGGRIGATE Function은 무조건 결과 값을 리턴한다. → 조건결과가 없으면 NULL 또는 0 리턴.
- Group By + Aggrigate Function → 조건결과가 없으면 ?

❑ SQL Error Or SQL Fail

- 구문상의 오류는 SQL ERROR.
- 논리상의 오류(예 1 = 2)는 FAIL 로 NULL 처리. → Aggrigate Function 사용하여 결과 ROW 반환.

❑ 수정가능 조인뷰

- 키보존 테이블 만 수정 가능.
- 키보존 테이블 자격(개별 로우 단위로 처리가 가능한 경우만 해당된다.)
- 키보존 테이블간의 처리(9i 에서는 FROM절 선행테이블로 고정 – 오라클 구현)

❑ 소계 처리

- COPY_T 복제, ROLLUP, CUBE 사용.

❑ UNION ALL 사용

- 중복제거의 목적이 아니라면 UNION 대신 UNION ALL로 처리.

VII-5. SQL 활용 기법.

다양한 기법들.

❑ IN의 사용.

- IN 은 n개의 '=' 조건이다. → 점조건!
- IN 은 OR의 부분 집합이다. → '=' + OR 조건과 같다.

❑ 결합 인덱스와 범위 조건.

- 결합 인덱스의 선두 컬럼 이후로 범위 조건 까지만 드라이빙 범위를 결정한다.
- 범위 조건 이후는 인덱스 체크조건으로 사용된다.

❑ IN의 파싱과 처리.

- IN의 파싱은 선두에서 부터.(FILO – Stack 구조로 처리)
- IN의 처리는 파싱된 뒤에서 부터.

❑ IN 과 NULL.

- IN과 NULL 처리.

  AND A.COL1 IN ('AAA','BBB',NULL)

  → AND (A.COL1 = 'AAA' OR A.COL1 = 'BBB' OR A.COL3 = NULL) → NULL 사용 가능.

- NOT IN 과 NULL 처리.

  AND A.COL1 NOT IN ('AAA','BBB',NULL)

  → AND (A.COL1 != 'AAA' AND A.COL1 != 'BBB' AND A.COL1 != NULL) → NULL 사용 불가.

  즉 NOT IN 조건에서는 NULL 사용에 주의 해야 한다.

VII-5. SQL 활용 기법.

결합 인덱스와 범위조건.

❑ 결합 인덱스와 범위 조건.

- 결합 인덱스의 선두 컬럼 이후로 범위 조건 까지만 드라이빙 범위를 결정한다.
- 범위 조건 이후는 인덱스 체크조건으로 사용된다.

→ 범위조건을 점조건으로 변경 하는것도 중요한 튜닝 포인트가 될 수 있다.

❑ 점조건 추가 - 상수값으로 추가.

- '선분은 점의 연속이다.' → 범위조건을 IN 을 이용한 상수조건으로 변경한다.

❑ 점조건 추가 – 서브쿼리 추가.

- 서브쿼리를 이용하여 조건 셋을 생성하여 처리한다. - '조건의 다리를 놓는다.'
- COPY_T, COPY_YMD, 코드성 테이블(부서코드, 점코드, 상태코드 등) 등을 이용 하거나, 복제 등으로 필요한 집합 SET을 만들어서 제공한다. → 서브쿼리 제공자로 수행!
- 서브쿼리 제공 조건 컬럼이 선두가 아니면서 선행 컬럼이 범위조건이면 서브쿼리 확인자로 처리 여부 확인!

→ 가능한 서브쿼리에서 모든 조건이 한번에 제공되게 IN 등을 사용 하여 처리한다.

이때 CARTATION PRODUCT이 너무 커지면 범위조건으로 수행 하는 것보다 더 부하가 클 수 있다.

예) 부서코드 20가지 * 사원타입 10가지 * 처리일자 30일 = 6000 가지 조건 SET!!!

VII-5. SQL 활용 기법.

INLINE VIEW

❑ 조인 횟수 감소

- 조인 후 그룹핑 → 그룹핑 후 조인으로 변경. (코드성 조인과 같이 결과집합의 레벨에 일치시)
- 동일한 유형의 함수 수행도 해당.(DECODE()나 사용자 함수 등)

❑ 순환관계 처리.

- 전개 후 조인을 위한 임시 집합 처리.

❑ 1 : M, M : M 조인의 해결

- 모든 집합 레벨을 1 로 통합하여 조인.

❑ OUTER JOIN의 OR조건 처리.

- OUTER JOIN의 OR 조건 지정시 인라인뷰로 해결.
- (A.COL1 IN ('10', '20', '70') OR A.COL1 IS NULL)

  → DECODE(A.COL1(+), '10', 'OK', '20', 'OK', '70', 'OK') = 'OK'

❑ 실행계획 제어

- 인라인뷰와 힌트로 처리 단계별 실행계획 제어.

❑ 부분범위 처리

- 전체 부분범위 처리 불가능시 개별 쿼리 단위 부분범위 처리 유도.

❑ 실행계획 분리

- 처리 조건, 처리 범위 별로 실행계획 분리시 사용.

VII-5. SQL 활용 기법.

BETWEEN 처리(이력 처리)

❑ BETWEEN 조건.

- 선분 이력 관리에서 발생 : 시작일, 종료일
- 최대값 수렴 : 진행 중인 이력의 종료일은 미도래 이므로 논리상으로는 NULL 값이지만
  SQL 작성 및 인덱스 엑세스를 위해서 NULL 대신 기준 최대값인 '99991231' 사용(일자 예)
- :Value BETWEEN 시작컬럼 AND 종료컬럼 → 시작컬럼 <= :value AND 종료컬럼 >= :value

  인덱스 구성1 : 시작컬럼 + 종료컬럼
  인덱스 구성2 : 종료컬럼 + 시작컬럼

  인덱스 구성1 시 : 시작컬럼 <= :사용자조건 → 처음부터 사용자 조건까지가 드라이빙 범위.
  종료컬럼 >= :사용자조건 (인덱스 선두 데이터 엑세스시 유리!)
  인덱스 구성2 시 : 종료컬럼 >= :사용자조건 → 사용자 조건부터 끝까지가 드라이빙 범위.
  시작컬럼 <= :사용자조건 (인덱스 끝 데이터 엑세스시 유리!)

  → BETWEEN 조건의 경우 시작 + 종료 의 범위가 하나의 기간이므로 대개의 경우에는 조건절에 ROWNUM <= 1 추가하여 처리를 제한 하는 것이 필요하다.

❑ 한편 넣기 Vs 양편 넣기

- 한편 넣기 : 관리되는 데이터의 단위(예:- 일자)별로 하나의 단위별로 한번의 이벤트만 허용되는 방식.
  일반적인 이력 관리 방법으로 SQL 작성이 쉬움.
- 양편 넣기 : 관리되는 데이터의 단위별로 하나의 단위별로 다수의 이벤트가 허용되는 방식.
  사용자가 원하는 데이터가 해당 시점의 시작, 중간, 최종에 모두 포함 가능하므로
  원하는 조건별로 SQL이 달라져야 한다.

VII-5. SQL 활용 기법.

BETWEEN 처리(이력 처리)

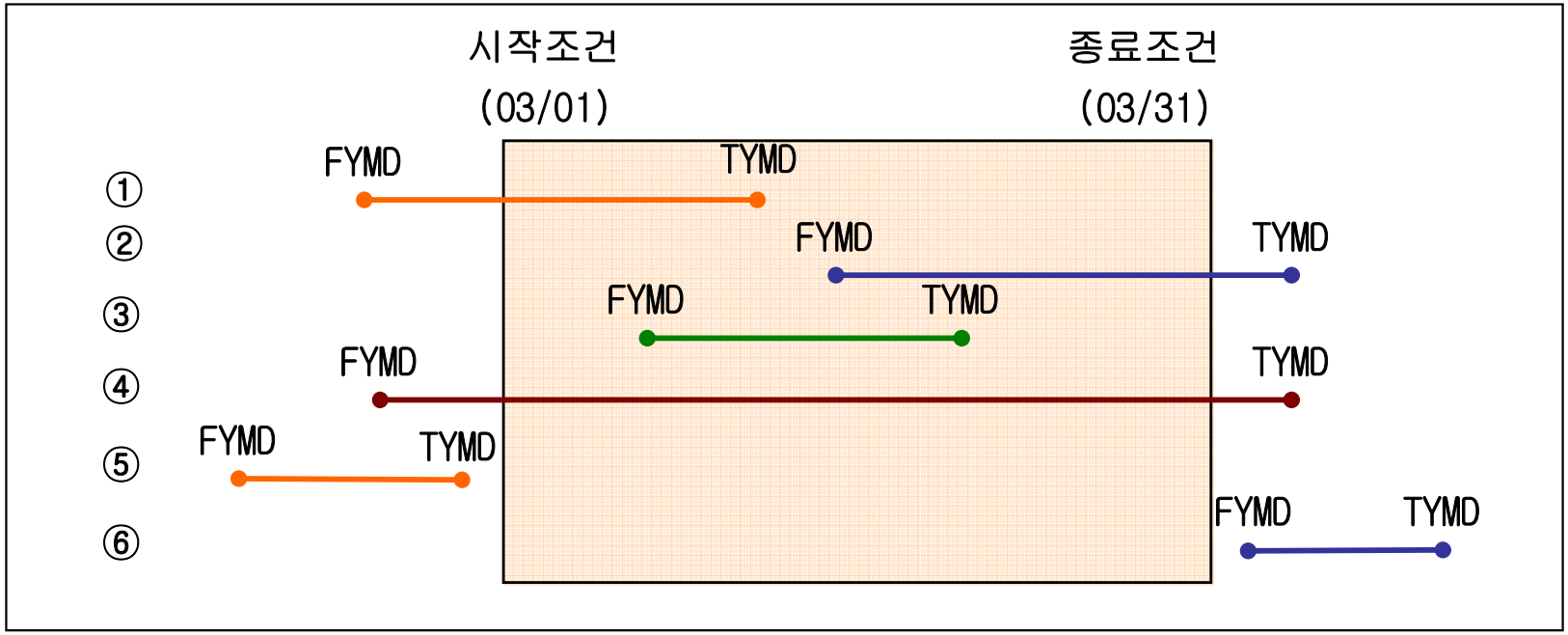
❑ 선분조건의 6가지 형태.

- ① 조건범위가 시작조건에만 걸린경우.
- ② 조건범위가 종료조건에만 걸린경우.
- ③ 조건범위가 시작/종료 조건 사이에 속한 경우.
- ④ 시작/종료 조건이 조건범위 사이에 속한 경우
- ⑤ 조건범위보다 이전.
- ⑥ 조건범위보다 이후.

시점 조건절은 항상 → :조건일 <= 시작일 AND :조건일 >= 종료일(양편은 > 조건)

범위 조건절은 항상 → :종료일 <= 시작일 AND :시작일 >= 종료일(한편 넣기)

→ :종료일 < 시작일 AND :시작일 >= 종료일(양편 넣기)



VII-6. NEW FEATURE.

WITH

❑ WITH 절.

- Oracle 9iR1 New Feature.
- 물리적인 임시 테이블을 SQL에서 동적으로 사용 가능하게 지원.
  With절은 With절로 정의한 집합을 시스템 내부적으로 관리되는 CREATE GLOBAL TEMPORARY TABLE 로 저장한 후 해당 SQL질의에서 원 집합 대신 이 임시 테이블을 엑세스 하게 하는 기능을 제공한다.

```
WITH summary AS (
     SELECT d.department_name, SUM(e.salary) AS dept_total
       FROM employees e, departments d
      WHERE e.department_id  = d.department_id
      GROUP BY d.department_name )
SELECT department_name, dept_total
  FROM summary
 WHERE dept_total > ( SELECT SUM( dept_total ) * 1/8
                        FROM summary  )
```

❑ WITH절 사용.

- 재사용성이 제일 중요하다. – 하나의 집합을 한번 이상 사용하는 SQL 구문에서 만 구현된다.
  재사용되지 않는 구문에서는 원집합의 정의로 그대로 처리된다.
- 내부적인 처리는 아래와 같이 사용자가 TEMP 테이블을 사용 하는것과 동일하다.

```
CREATE GLOBAL TEMPORARY TABLE "SYS"."SYS_TEMP_1_0_FD9D6600"

INSERT /*+ APPEND BYPASS_RECURSIVE_CHECK */
  INTO "SYS"."SYS_TEMP_1_0_FD9D660E"
```

VII-6. NEW FEATURE.

MULTITABLE INSERT

❑ MULTITABLE INSERT란?

- 하나의 결과집합으로 하나 이상의 TABLE에 대해서 하나 이상의 INSERT를 가능하게 한다.

Unconditional Insert

```
INSERT ALL
    INTO IN_TAB1 (COL1, COL2, COL3, VAL1)
      VALUES (COL1, COL2, UVAL1, ETC1)
    INTO IN_TAB1 (COL1, COL2, COL3, VAL1)
      VALUES (COL1, COL2, UVAL1, ETC2)
    INTO IN_TAB1 (COL1, COL2, COL3, VAL1)
      VALUES (COL1, COL2, UVAL1, ETC3)
SELECT COL1, COL2, UVAL1, ETC1, ETC2, ETC3
FROM   SEL_TAB1;
```

Conditional ALL Insert

```
INSERT ALL
WHEN COL1 IN (SELECT KEY1 FROM SCALAR_SQL) THEN
   INTO IN_TAB1 (COL1, COL2, COL3, VAL1)
     VALUES (COL1, COL2, UVAL1, TOT_VAL)
   INTO IN_TAB2 (COL1, COL2, COL3)
     VALUES (COL1, COL2, UVAL1)
WHEN TOT_VAL > 5000 THEN
   INTO IN_TAB3 (COL1, COL2, UVAL1, TOT_AMT)
     VALUES (COL1, COL2, UVAL1,TOT_VAL)
SELECT COL1, COL2, UVAL1,ETC1+ETC2+ETC3 TOT_VAL
FROM   SEL_TAB1;
```

Conditional FIRST Insert

```
INSERT FIRST
WHEN C_ID IN ('111','222','333') THEN
   INTO IN_TAB2 (COL1, COL2, TOT_AMT1)
     VALUES (COL1, COL2, DT)
WHEN C_ID IN ('101','202','303') THEN
   INTO IN_TAB1 (COL1, COL2, COL3, VAL1)
     VALUES (COL1, COL2, UVAL1, TOT_AMT2)
ELSE
   INTO IN_TAB3 (COL1, COL2, UVAL1, TOT_AMT)
     VALUES (COL1, COL2, UVAL1,TOT_AMT3)
SELECT COL1, COL2, UVAL1, ETC1 TOT_AMT1,
       ETC1+ETC2 TOT_AMT2,
       ETC1+ETC2+ETC3 TOT_AMT3
FROM   SEL_TAB1;
```

❑ 조건절에 Scalar Expression도 가능하다.

VII-6. NEW FEATURE.

MERGE(UPSET)

❑ MERGE절은 임의의 테이블의 데이터를 조회하여 목적 테이블에 INSERT 또는 UPDATE를 하고자 할 때 사용한다.

조회된 데이터가 목적테이블에 이미 존재한다면 갱신을 수행하고, 그렇지 않으면 새로 행을 삽입한다.

- 내부적으로 OUTER JOIN 방식의 조인으로 처리한다.

```
MERGE INTO TAR_TAB1   T
USING (SELECT COL1, COL2
       FROM BAS_TAB1) B
ON    (T.COL1 = B.COL1)
WHEN       MATCHED THEN UPDATE SET T.COL3 = B.COL3
WHEN NOT MATCHED THEN INSERT (T.COL1, T.COL2, T.COL3)
                              VALUES (B.COL1, '111', B.COL3) ;
```

```
MERGE STATEMENT Hint=ALL_ROWS
  MERGE TAR_TAB1
    NESTED LOOPS OUTER
      TABLE ACCESS FULL BAS_TAB1
      TABLE ACCESS BY INDEX ROWID TAR_TAB1
        INDEX RANGE SCAN TAR_TAB1_IDX1
```

1. 효율적인 배치 프로세싱 가이드.
2. 사례**1 -** 절차형 **SQL** 개선.
3. 사례**2 -** 조인 및 **Access** 방법의 개선.
4. 사례**3 -** 불필요한 수행 방지.
5. 사례**4 -** 임시 집합 사용.
6. 기타 **-** 사례 및 기능들.

VIII-0 주요 튜닝 유형.

주요 튜닝 유형
(OLTP/Batch)

❑ 잘못 작성된 SQL 개선.

❑ 인데스 전략 개선.
엑세스 패스 도출로 부적절한 인덱스 개선하여 엑세스 효율 개선.

❑ 조인 순서 변경.
가장 적은 처리 범위 부터 조인 수행되게 순서 변경.(NL JOIN)

❑ 코드성 조인 감소.
코드성 테이블을 모두 조인 후 최종 GROUPING 처리를 먼저 GROUPING 처리 후 최종 결과 집합만 조인으로 조인횟수 감소.

❑ 조인 방법 비효율 개선.
NL , HASH, SORT MERGE 조인의 특성에 맞게 조인 수행.

❑ 부분 범위 처리 유도.
인덱스 등을 이용 최소의 처리 만으로 사용자 응답을 가능하게 유도.

❑ 중복 수행 방지.
동일 테이블, 함수, 스칼라 서브쿼리의 중복 수행 방지.

❑ 확장 UPDATE문 이용.
SELECT와 UPDATE를 하나로 통합, 키보존 테이블 활용.

❑ 실행계획 분리.
처리 조건, 처리 범위에 따라 실행계획 분리로 최적 수행 보장.

VIII-1 효율적인 Batch Processing 가이드

주요 개선 방안

- 절차형 SQL의 개선.
  - 가능하면 CURSOR SQL에서 모든 처리가 이루어지도록
    배치는 처리건수가 다르므로 온라인 P/G과는 다른 접근 방법이 필요.
    1건 수행 0.001초 => 100만건 수행 1,000 초로 약 17분 소요.
  - 처리범위 증가로 인덱스를 이용한 개별 처리보다는 전체범위 처리 유리.
- 배열 처리.
- Partitioning 고려.
- Parallel 사용 검토.
- Temp Table의 사용.
- 대량의 DML 처리를 위한 동적 환경 제어.
  - 처리 대상 건수 파악이 중요

<u>효율적인 처리를 위한 동적 할당 가능 파라미터</u>

```
ALTER SESSION SET WORKAREA_SIZE_POLICY              = MANUAL  ;
ALTER SESSION SET SORT_AREA_SIZE                    = 20971520;
ALTER SESSION SET SORT_AREA_RETAINED_SIZE           = 20971520;
ALTER SESSION SET HASH_AREA_SIZE                    = 41943040;
ALTER SESSION SET HASH_MULTIBLOCK_IO_COUNT          = 64;
ALTER SESSION SET DB_FILE_MULTIBLOCK_READ_COUNT     = 64;
```

→ 각각의 시스템 및 수행에 맞는 적절한 환경 설정 값 도출이 중요!

VIII-2 사례1-절차형SQL의 개선.

사용자 로직과 LOOP QUERY 비효율

```
CURSOR CUR_COUNT1 IS
SELECT ...
  FROM EC_USER@KWEBDB1 A, EC_ECUSER@KWEBDB1 B
 WHERE TO_CHAR(B.CREDATE, 'YYYYMMDD') =
   AND A.USERID = B.USERID;

CURSOR CUR_COUNT2 IS
SELECT ECUSERID
  FROM KLOG_CUSTOMER
 WHERE ECUSERID = v_ecuserid;

CURSOR CUR_COUNT3 IS
SELECT ...
  FROM KM_SINGLE@KWEBDB1
 WHERE ECUSERID = temp_ECUSERID;

CURSOR CUR_COUNT4 IS
SELECT DO_GUBUN
  FROM KLOG_ADDR
 WHERE ECUSERID = v_ecuserid;

CURSOR CUR_COUNT5 IS
SELECT ...
  FROM EC_USER@KWEBDB1 A, EC_ECUSER@KWEBDB1 B
 WHERE TO_CHAR(B.UPDDATE, 'YYYYMMDD') = ...
   AND A.USERID = B.USERID;

BEGIN
```

1

```
OPEN CUR_COUNT1;
LOOP
    FETCH CUR_COUNT1 INTO  ... ;
```

2

```
    OPEN CUR_COUNT3;
    LOOP
        FETCH CUR_COUNT3 INTO ...;
    END LOOP;
    CLOSE CUR_COUNT3;
```

3

```
    OPEN CUR_COUNT4;
    LOOP
        FETCH CUR_COUNT4 INTO ...;
    END LOOP;
    CLOSE CUR_COUNT4;
```

4

```
    OPEN CUR_COUNT2;
    LOOP
        FETCH CUR_COUNT2 INTO nYESNO;
        UPDATE KLOG_CUSTOMER
           SET ...
         WHERE ECUSERID = v_ecuserid;
        COMMIT;
    END LOOP;
    CLOSE CUR_COUNT2;

    IF nYESNO = 0 THEN
        INSERT INTO KLOG_CUSTOMER ... ;
    END IF;
```

5

```
END LOOP;
CLOSE CUR_COUNT1;
COMMIT;

OPEN CUR_COUNT5;
LOOP
    FETCH CUR_COUNT5 INTO  ... ;
    OPEN CUR_COUNT3;
```

6

```
    LOOP
        FETCH CUR_COUNT3 INTO ...;
        IF  v_socialid IS NOT NULL THEN
            ...
        END IF;
    END LOOP;
    CLOSE CUR_COUNT3;
```

7

```
    OPEN CUR_COUNT4;
    LOOP
    FETCH CUR_COUNT4 INTO ...;
    END LOOP;
    CLOSE CUR_COUNT4;

    UPDATE KLOG_CUSTOMER
       SET ...
     WHERE ECUSERID = v_ecuserid;
END LOOP;
CLOSE CUR_COUNT5;
COMMIT;
```

VIII-2 사례1-절차형SQL의 개선.

주요 개선 방안

- SQL 통합
- Array Processing
- DBMS CALL 부하 감소

1

```
CURSOR CUR_COUNT
IS
SELECT /*+ ORDERED USE_NL(S C) */
       ...,
       (SELECT DO_GUBUN
          FROM KLOG_ADDR A
         WHERE A.ECUSERID = S.ECUSERID
           AND ROWNUM=1) DO_GUBUN
  FROM (SELECT /*+ ORDERED USE_NL(B A Y) */
               ...
          FROM (SELECT /*+ INDEX(B EC_ECUSER_CREDATE_IND)*/
                       ...
                  FROM EC_ECUSER@KWEBDB1 B
                 WHERE B.CREDATE > SYSDATE - 2
                   AND B.CREDATE < SYSDATE
                 UNION ALL
                SELECT /*+ INDEX(B EC_ECUSER_IDX01)*/
                       ...
                  FROM EC_ECUSER@KWEBDB1 B
                 WHERE B.UPDDATE > SYSDATE - 2
                   AND B.UPDDATE < SYSDATE
                   AND TO_CHAR(B.CREDATE,'YYYYMMDD') != '20
               ) B, EC_USER@KWEBDB1 A, KM_SINGLE@KWEBDB1 Y
         WHERE  A.USERID = B.USERID
           AND  B.ECUSERID = Y.ECUSERID
       ) S, KLOG_CUSTOMER C
 WHERE S.ECUSERID = C.ECUSERID(+);
```

2

```
BEGIN
    v_rows :=1000;
    OPEN CUR_COUNT;
    LOOP
        FETCH CUR_COUNT  BULK COLLECT
         INTO T_ECUSERID, ... LIMIT v_rows;

        FORALL i in 1 .. T_ECUSERID.COUNT
            UPDATE KLOG_CUSTOMER
               SET CUST_TYPE = T_CUST_TYPE(i), ...
             WHERE ECUSERID = T_OLD_ECUSERID(i);

        FORALL i in 1 .. T_ECUSERID.COUNT
            INSERT /*+ APPEND */
              INTO KLOG_CUSTOMER ( ... )
            SELECT T_ECUSERID(i), ...
              FROM DUAL
             WHERE T_OLD_ECUSERID(i) IS NULL;

        EXIT WHEN CUR_COUNT%NOTFOUND;
    END LOOP;
    COMMIT;
CLOSE CUR_COUNT;
```

VIII-2 사례1-절차형SQL의 개선.

개선 원인

- 불필요한 SQL 분리로 인한 Loop Query 발생.
  (Loop Query의 경우, 과도한 DBMS Call과
  과도한 Random I/O에 의한 속도 저하 현상 발생).

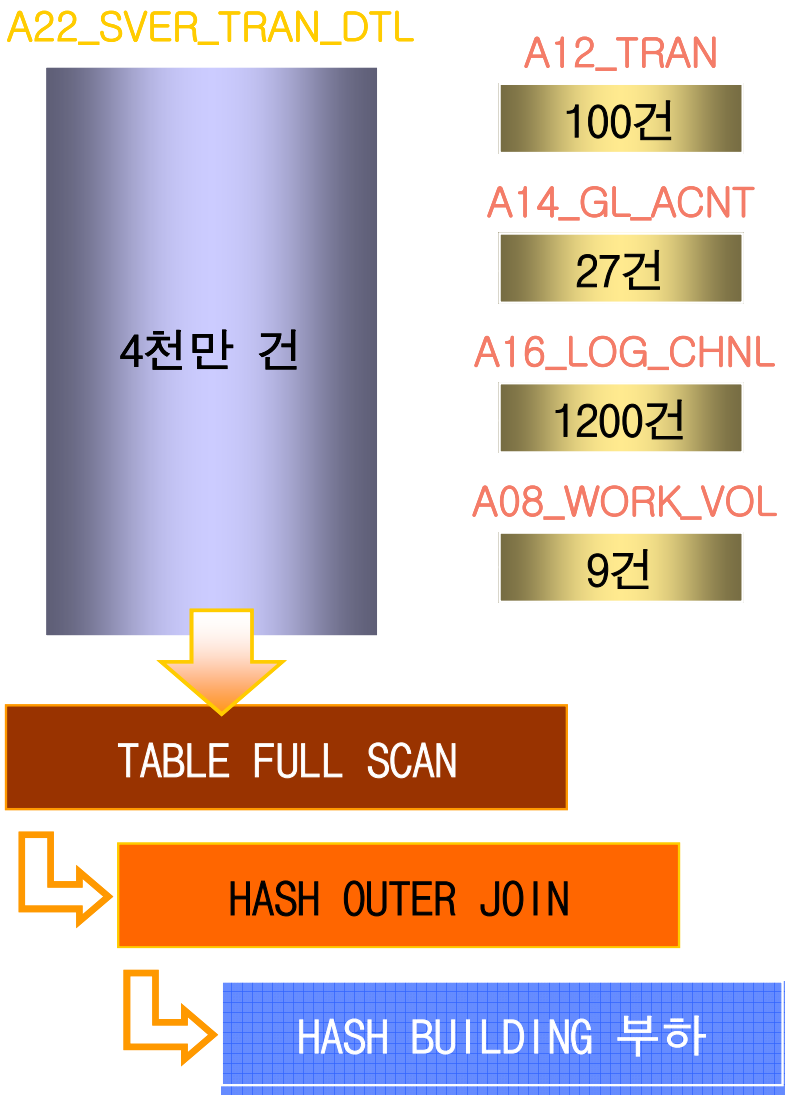
- 불필요한 SQL 분리로 인한 Insert, Update와 같은 DML
  발생과 이에 따른 Rollback Image, Redo Log 발생에
  따른 Overhead 발생.

- SQL 통합하여 Direct Insert Loading을 함으로써
  한 번의 DBMS Call과 다량의 범위에 대하 Scan 위주의
  I/O를 함으로 써 I/O에 대한 Overhead를 최소화 시키고,
  Direct Insert Loading을 함으로써 Rollback Image와
  Redo Log발생을 없앰으로써 성능 개선.

VIII-3 사례2-조인 및 Access방법의 개선.

LARGE HASH BUILD INPUT의 비효율 개선

```
INSERT INTO A22_SVER_TRAN_DTL_TMP ( ... )
SELECT ...
  FROM (SELECT ...
          FROM (SELECT ...
                  FROM A22_SVER_TRAN_DTL A22,  A12_TRAN A12
                 WHERE A22.SVER_TYPE_CD = A12.SVER_TYPE_CD(+)
                   AND A22.UPMU_CATE_CD   = A12.UPMU_CATE_CD(+)
                   ...
                ) T12,  A14_GL_ACNT  A14
          WHERE T22_GL_ACCOUNT_ID  = GL_ACCOUNT_ID(+)
        ) T14,  A16_LOG_CHNL A16
 WHERE '00' = TM_TYPE_CD(+)
   AND '0'  = INPUT_EQIP_CD(+)
   AND T22_CHNL_OP_BR_CD = CHNL_OP_BR_CD(+) ;

UPDATE A22_SVER_TRAN_DTL_TMP
   SET T22_ACNT_SBU_CD = '10'
 WHERE T22_ACNT_BR_CD NOT IN (SELECT DISTINCT WORK_BR_CD
                                FROM A08_WORK_VOL
                               WHERE ASIN_SBU_CD = '30'
                                 AND BASE_YM = :IN_DATE)
   AND T22_ACNT_SBU_CD = '30' ;

UPDATE A22_SVER_TRAN_DTL_TMP
   SET T22_OP_SBU_CD = '10'
 WHERE TCT_CHAG_BR_CD NOT IN (SELECT DISTINCT WORK_BR_CD
                                FROM A08_WORK_VOL
                               WHERE ASIN_SBU_CD = '30'
                                 AND BASE_YM = :IN_DATE)
   AND T22_OP_SBU_CD = '30' ;
```



VIII-3 사례2-조인 및 Access방법의 개선.

개선 방안– Hash Build Input 감소

```
INSERT /*+ PARALLEL(A22_SVER_TRAN_DTL_TMP 8) */
  INTO A22_SVER_TRAN_DTL_TMP ( ... )
SELECT /*+ ORDERED USE_HASH(A22 A12 A16) PARALLEL(A22 8)
          PARALLEL(A12 8) PARALLEL(A16 8) FULL(A) */
       ...
  FROM A22_SVER_TRAN_DTL A22,
       (SELECT ...
          FROM A12_TRAN A12, A14_GL_ACNT A14,
               (SELECT DISTINCT WORK_BR_CD ACNT_BR_CD
                  FROM A08_WORK_VOL
                 WHERE ASIN_SBU_CD = '30'
                   AND BASE_YM = :IN_DATE
               ),
               (SELECT DISTINCT WORK_BR_CD TCT_CHAG_BR_CD
                  FROM A08_WORK_VOL
                 WHERE ASIN_SBU_CD = '30'
                   AND BASE_YM = :IN_DATE
               )
         WHERE ROWNUM > 0
       ) A12, A16_LOG_CHNL A16
 WHERE A22.SVER_TYPE_CD   = A12_SVER_TYPE_CD(+)
   AND A22.UPMU_CATE_CD   = A12_UPMU_CATE_CD(+)
   ...
   AND A22.GL_ACCOUNT_ID  = A14_GL_ACCOUNT_ID(+)
   AND A22.ACNT_BR_CD     = A08_ACNT_BR_CD(+)
   AND A22.TCT_CHAG_BR_CD = A08_TCT_CHAG_BR_CD(+)
   AND '00'               = A16.TM_TYPE_CD(+)
   AND '0'                = A16.INPUT_EQIP_CD(+)
   AND A22.CHNL_OP_BR_CD  = A16.CHNL_OP_BR_CD(+)
```

- 작은 디멘젼 테이블들로 Cartesian 집합 생성.
- Hash Table Build 횟수 감소.

```
HASH JOIN OUTER
  HASH JOIN OUTER
    TABLE ACCESS FULL        A22_SVER_TRAN_DTL
    VIEW
      COUNT
        FILTER
          MERGE JOIN CARTESIAN
            MERGE JOIN CARTESIAN
              MERGE JOIN CARTESIAN
                TABLE ACCESS FULL              A12_TRAN
                SORT JOIN
                  TABLE ACCESS FULL            A14_GL_ACNT
              SORT JOIN
                VIEW
                  SORT UNIQUE
                    TABLE ACCESS FULL          A08_WORK_VOL
            SORT JOIN
              VIEW
                SORT UNIQUE
                  TABLE ACCESS FULL            A08_WORK_VOL
  TABLE ACCESS FULL          A16_LOG_CHNL
```

VIII-4 사례3-불필요한 수행 방지.

기존 사례1

```
EXEC SQL SELECT COUNT(*) INTO :ll_row
         FROM SOR09
        WHERE VENDOR     = SUBSTR(Trim(:file_log.c_sourceflnm), 1,  6)
          AND JNO        = SUBSTR(Trim(:file_log.c_sourceflnm), 7,  8)
          AND HHMMSS     = SUBSTR(Trim(:file_log.c_sourceflnm), 15, 6);
if (sqlca.sqlcode != 0) {
   if (sqlca.sqlcode != 1403)   {
      sprintf(m_logerr, "파일[%s] QUERY FAIL!", file_log.c_sourceflnm);
      LogTrace();
      return(-1);
   }
}

if (ll_row > 0) {
   EXEC SQL UPDATE SOR09
            SET OCCURTIME   = Trim(:file_log.c_occurtime),
                ...
        WHERE VENDOR      = SUBSTR(Trim(:file_log.c_sourceflnm), 1,  6)
          AND JNO         = SUBSTR(Trim(:file_log.c_sourceflnm), 7,  8)
          AND HHMMSS      = SUBSTR(Trim(:file_log.c_sourceflnm), 15, 6);

}
Else {
   EXEC SQL INSERT INTO SOR09 (VENDOR, JNO, HHMMSS, OCCURTIME, ERRCODE,
                               ERRSEQ, ERRCONTENTS, ORGFILENM,
                               RETRYNUM, RETRYDELAY, LMODDATE, LMODTIME)
                     VALUES( SUBSTR(Trim(:file_log.c_sourceflnm), 1,  6),
                        .........,
                        TO_CHAR(SYSDATE, 'YYYYMMDD'),
                        TO_CHAR(SYSDATE, 'HH24MISS'));
}
```

기존 방법

두번의 DBMS CALL 발생.
1번째 : 존재 여부 체크
Count(*)
2번째 :
존재하면 UPDATE 수행.
없으면 INSERT 수행.

개선 방법

최소 한번의 CALL이 발생.
1번째 : 먼저 UPDATE 수행.

2번째 :
sqlca.sqlerrd[2] 값 체크
= 0 이면 INSERT 수행.
> 0 이면 UPDATE 완료.

VIII-4 사례3-불필요한 수행 방지.

기존 사례2

```
-- 기존 자료 삭제
DELETE SOR06
 WHERE storecode BETWEEN :frstore AND :tostore;

=> 전체 점의 경우에도 DELETE로 삭제
   전체 데이터의 삭제시에는 TRUNCATE 사용.

-- 조건상이 중복 엑세스
SELECT s.storecode, s.skucode
  FROM SMS19 s, SMS12 c, SMS01 t
 WHERE c.skucode = s.skucode
       ...
   AND s.storecode BETWEEN :frstore
                       AND :tostore
   AND s.orddate <= :ueeddate
   AND c.ordtype NOT IN ('1')
UNION
SELECT s.storecode, s.skucode

  FROM SMS19 s, SMS12 c, SMS01 t
 WHERE c.skucode = s.skucode
       ...
   AND s.storecode BETWEEN :frstore AND :tostore
   AND s.orddate <= :ueeddate
   AND (c.linecode = '06101' OR c.divcode = '05')
```

```
SELECT s.storecode, s.skucode
  FROM SMS19 s, SMS12 c, SMS01 t
 WHERE c.skucode = s.skucode
       ...
   AND s.storecode BETWEEN :frstore
                       AND :tostore
   AND s.orddate <= :ueeddate
   AND ( ( c.ordtype NOT IN ('1') )
         OR
         ( c.linecode = '06101' OR
           c.divcode  = '05' )
       )
```

VIII-5 사례4-임시 집합 사용.

중복/반복 수행 해결 예제

예제는 유통업체의 배치로 반품 재매입 갱신용 배치입니다.

1. 반복/ 중복 수행.(전형적인 LOOP QUERY 형태입니다.)
   cur_sbu05, cur_sbu06 두 커서로 정의된 집합에 대해서 반복 수행의 경우.

```
      sbu05_cursor();
       sbu05_cursor1();
       if (sbu05_banpum()        < 0) log_off_oracle(-1);
       if (sbu05_maip()          < 0) log_off_oracle(-1);
       if (sbu05_banpum_second() < 0) log_off_oracle(-1);
       if (sbu05_maip_second()   < 0) log_off_oracle(-1);
       if (sbu05_update()        < 0) log_off_oracle(-1);
```

   또한 5개의 프로세스 별로 다음의 두개의 작업을 반복 수행합니다.

```
       if (sst01_update_rtn() < 0)
       if (sst23_update_rtn() < 0)
```

 즉 다수의 반복작업과 건별 UPDATE OR INSERT 가 수행되고 있는 배치입니다.
  총 6개 테이블에 대한 INSERT, UPDATE 처리가 건별로 수행되고 있습니다.
 처리 건수가 많을수록 수행시간은 비례이상으로 증가하는 형태입니다.

튜닝 방안 :

 1. 집합처리로 중간집합 사용.
    6개 테이블에 대한 INSERT OR UPDAE이므로 하나의 SQL로 모든 처리가 불가능.
    처리를 위한 공통 중간 집합을 생성, 이를 이용 대상 테이블 별 하나의 SQL로 처리.
 2. 처리 단순화.
    하나의 대상 테이블에 대한 2번씩의 중복 처리를 하나의 처리로 통합.
    처리 프로세스와 처리 대상 한번의 수행으로 모두 처리.

VIII-6 사례5-기타 사례 및 기능들.

- 파티션 Exchange.
  대상 파티션에 대량의 변경이 발생시 동일 구조의 임시 테이블에 작업,
  로컬 인덱스 생성 후 파티션 및 인덱스 Exchange.

- 업무 매핑 테이블 사용.
  프로그램에서 처리하던 로직을 업무매핑 테이블을 이용 조인을 이용하여
  SQL 만으로 DBMS 내부 처리로 수행.

- RDBMS 기능 이용.
  Analytic Function, Scalar SubQuery 등 오라클에서 제공하는 기본 기능을
  활용하여 복잡한 처리 감소.

감사합니다

(주)엑셈

www.ex-em.com / support@ex-em.com