Dedicated Server Process: 아키텍처적 이해

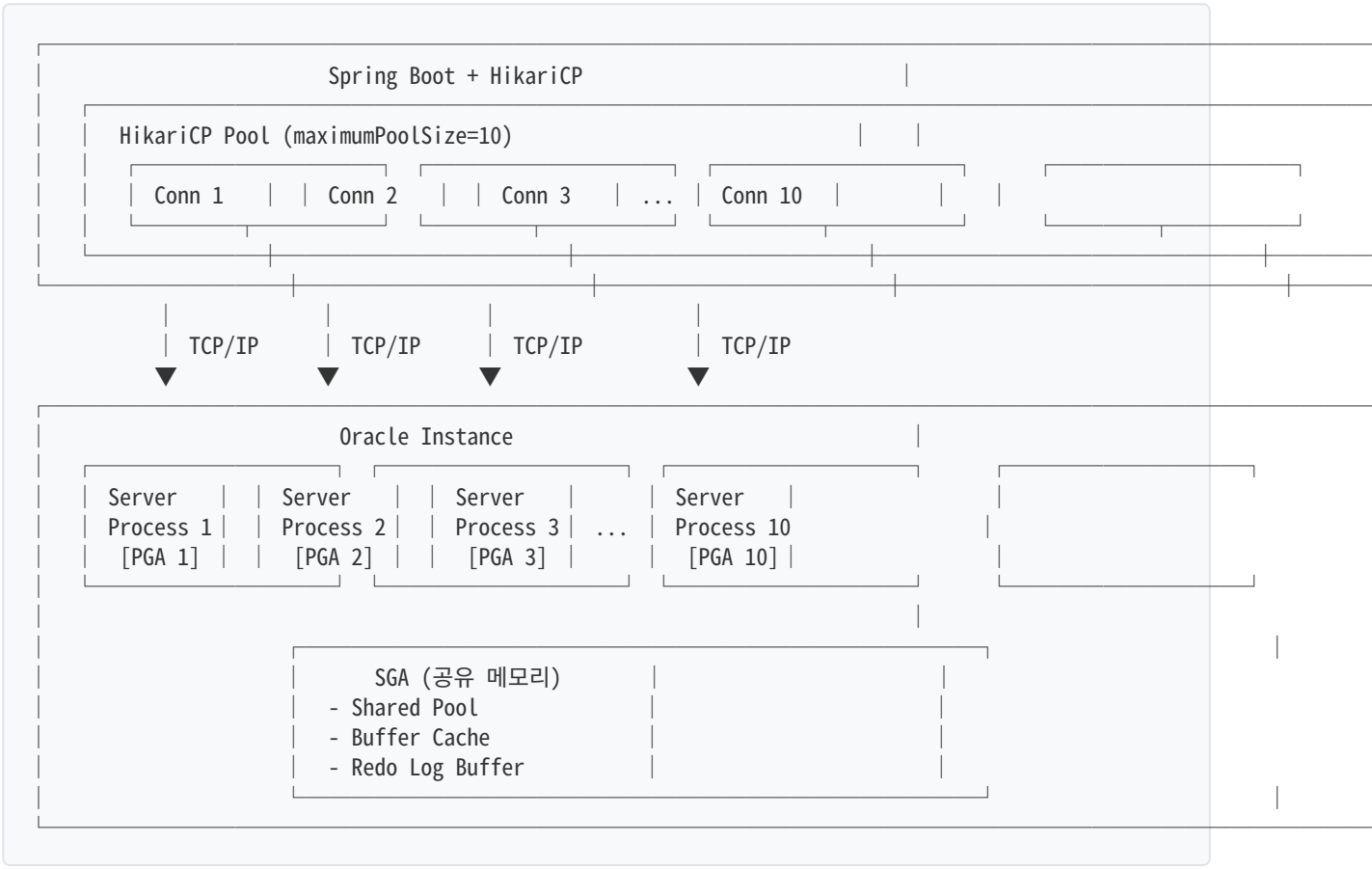
목차

1. 핵심 개념: 1:1 매핑의 의미
2. Server Process가 수행하는 작업
3. PGA의 구성요소
4. Connection 생성 비용의 구체적 근거
5. 프로젝트 연결: 1차 테스트의 아키텍처적 해석
6. Dedicated Server의 설계적 Trade-off
7. PGA 구성요소별 동작 메커니즘
8. UGA (User Global Area) 상세
9. Private SQL Area 상세
10. Cursor (커서)
11. Fetch (페치)
12. Server Process와 SGA의 상호작용
13. 프로젝트 연결: IN절 파라미터 개수와 Library Cache
14. 정리: Server Process, PGA, SGA 상호작용
15. 면접 대응 포인트

---

1. 핵심 개념: 1:1 매핑의 의미

Dedicated Server 모드에서 가장 중요한 특성은 Client Connection과 Server Process 간의 1:1 관계다.



HikariCP의 maximumPoolSize=10 설정은 Oracle 관점에서 최대 10개의 Server Process가 생성된다는 의미다. 각 Server Process는 OS 레벨의 독립적인 프로세스이며, 자신만의 PGA 메모리 영역을 갖는다.

---

2. Server Process가 수행하는 작업

공식 문서에서 Server Process의 역할을 명시하고 있다:

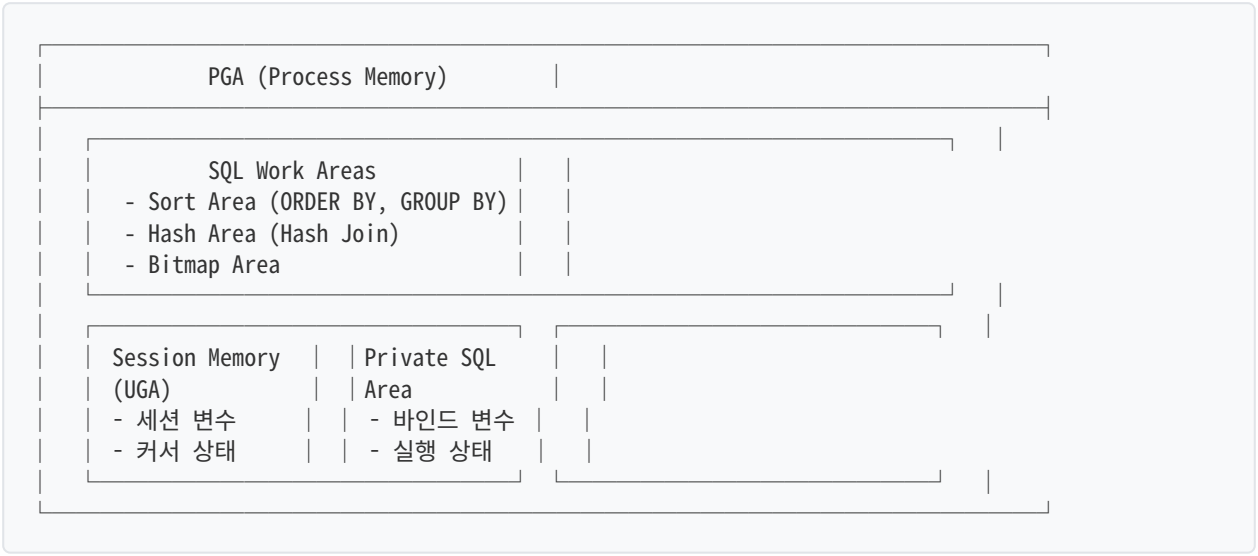
작업	설명	관련 메모리
SQL Parsing	SQL 문장의 문법/의미 검사, 실행 계획 수립	Shared Pool (Library Cache)
Query Execution	실행 계획에 따른 데이터 처리	Buffer Cache, PGA
Data Block Read	디스크에서 데이터 블록을 Buffer Cache로 로드	Buffer Cache
Result Return	처리 결과를 클라이언트에게 반환	PGA → Network

중요한 점은 Server Process가 SGA에 직접 접근한다는 것이다. Client Process는 SGA에 접근할 수 없고, 반드시 Server Process를 통해서만 데이터베이스와 상호작용한다.

---

3. PGA의 구성요소

PGA 내부 구조:

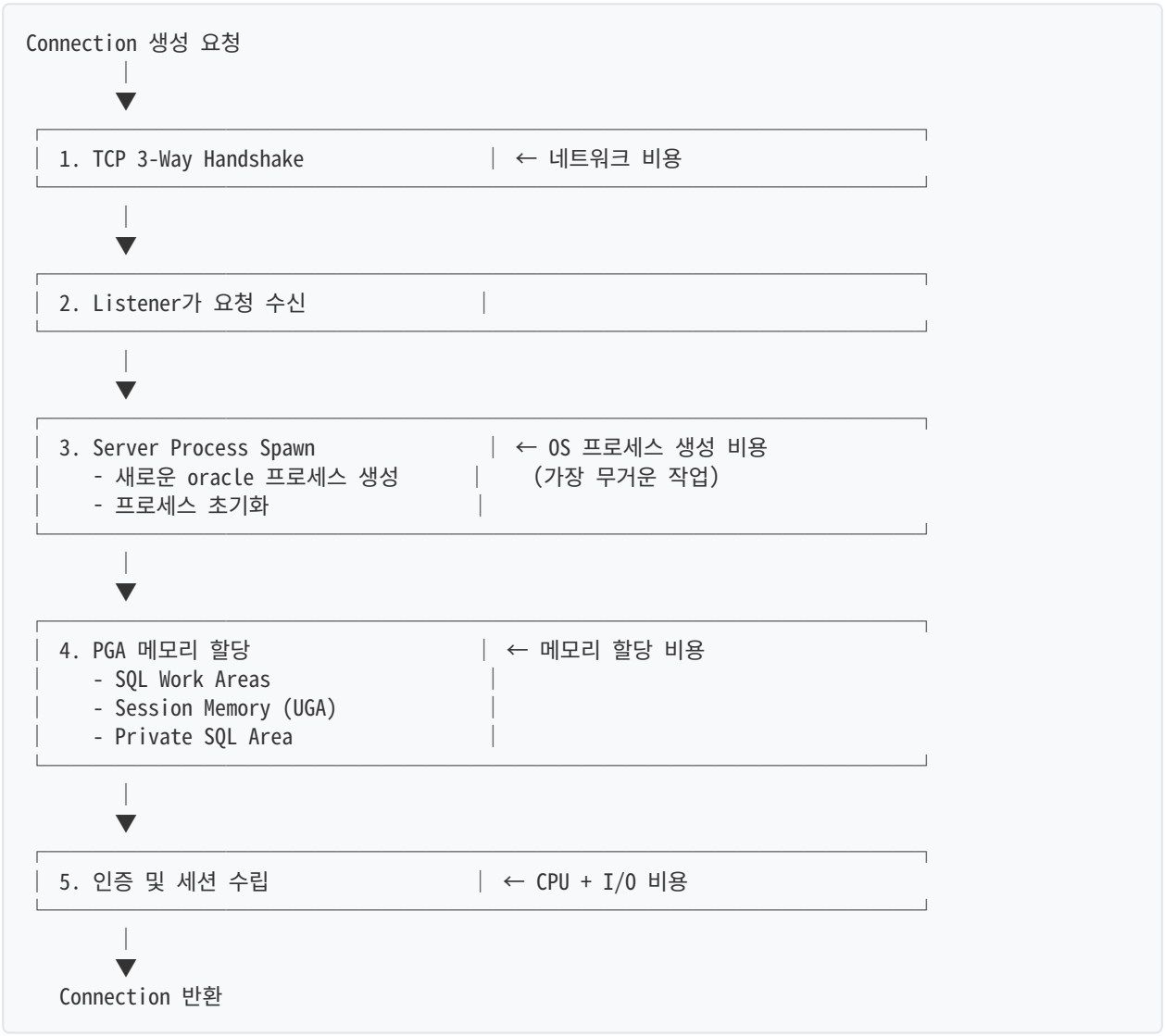


Session Memory (UGA)는 Dedicated Server 모드에서 PGA 내부에 위치한다. 이것이 Shared Server 모드와의 핵심 차이점이다. Shared Server에서는 UGA가 SGA의 Large Pool로 이동하는데, 이는 여러 Shared Server가 세션 정보를 공유해야 하기 때문이다.

---

4. Connection 생성 비용의 구체적 근거

Dedicated Server 모드에서 Connection을 생성할 때 발생하는 일을 단계별로 분해하면:

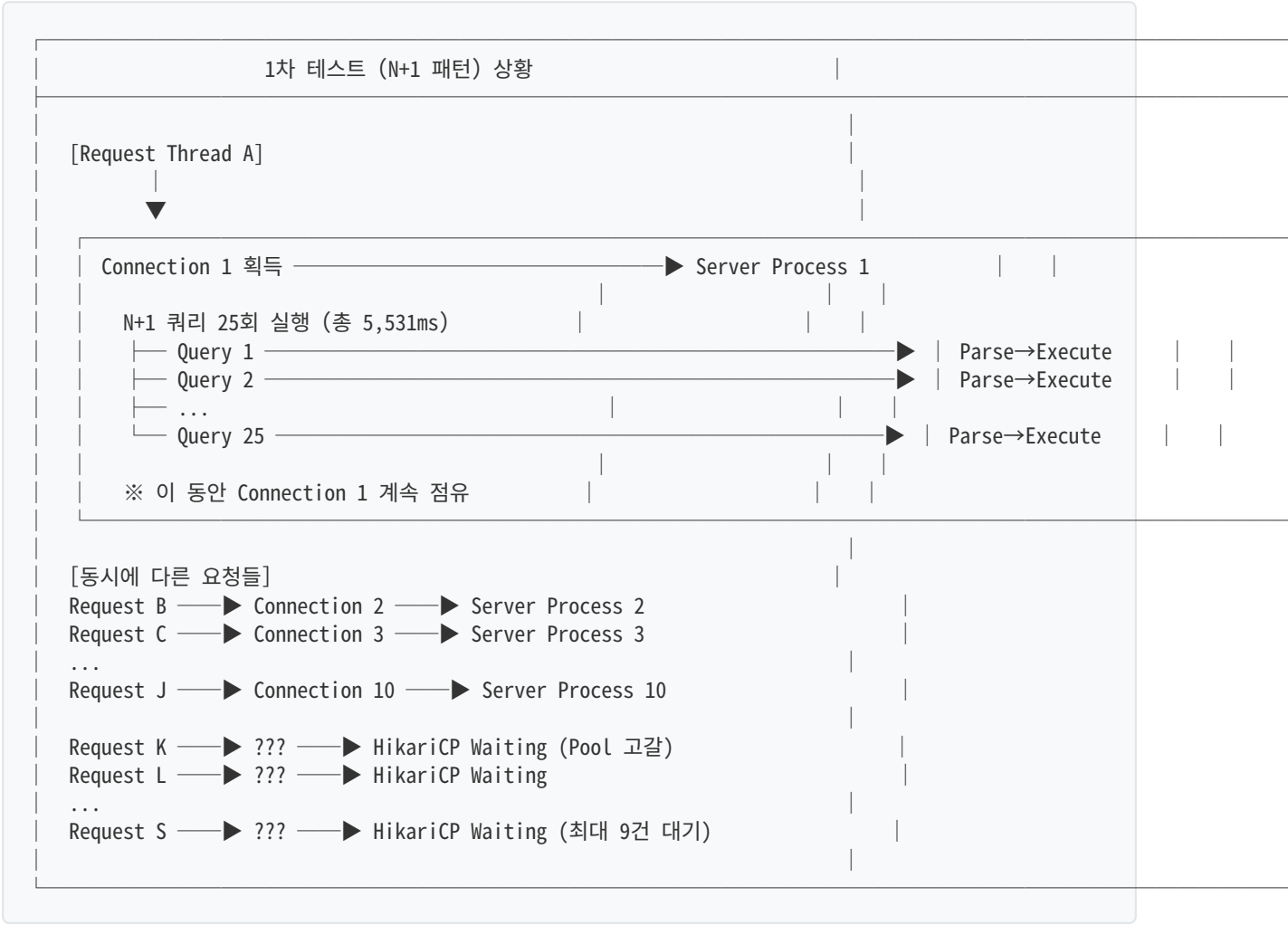


3단계와 4단계가 Connection 생성이 "비싼" 이유의 핵심이다. OS가 새로운 프로세스를 생성하고, 해당 프로세스에 메모리를 할당하는 작업은 단순 소켓 연결과는 비교할 수 없는 오버헤드를 수반한다.

---

5. 프로젝트 연결: 1차 테스트의 아키텍처적 해석

Wherehouse 프로젝트 1차 테스트 결과를 Dedicated Server 아키텍처로 해석하면:



핵심 인과관계

1. N+1 패턴으로 25회 쿼리 실행 → 하나의 Connection(= 하나의 Server Process)을 장시간 점유
2. HikariCP Pool 크기가 10개 → 최대 10개의 Server Process만 존재
3. 10개 Connection이 모두 사용 중일 때 추가 요청 → Connection 획득 대기 (Waiting)
4. RDB 시간 비율 6.2%인데 전체 응답 5,531ms → 나머지 94%의 상당 부분은 Connection 대기 시간 및 네트워크 왕복 오버헤드

---

6. Dedicated Server의 설계적 Trade-off

장점

- 구현이 단순하고 예측 가능
- 세션 상태가 PGA에 있으므로 접근 속도 빠름
- 장시간 실행 쿼리나 관리 작업에 적합

단점

- Connection 수 = Server Process 수 = 메모리 사용량 비례 증가
- 유휴 Connection도 Server Process와 PGA를 점유
- 동시 연결 수가 많으면 시스템 리소스 부담

공식 문서에서도 이 Trade-off를 언급한다:

> "Even when the user is not actively making a database request, the dedicated server process remains—although it is inactive and can be paged out on some operating systems."

사용자가 아무 요청을 하지 않아도 Server Process는 살아있고, PGA 메모리를 점유한다. 이것이 Connection Pool을 사용하되 Pool 크기를 적절히 제한해야 하는 이유다.

---

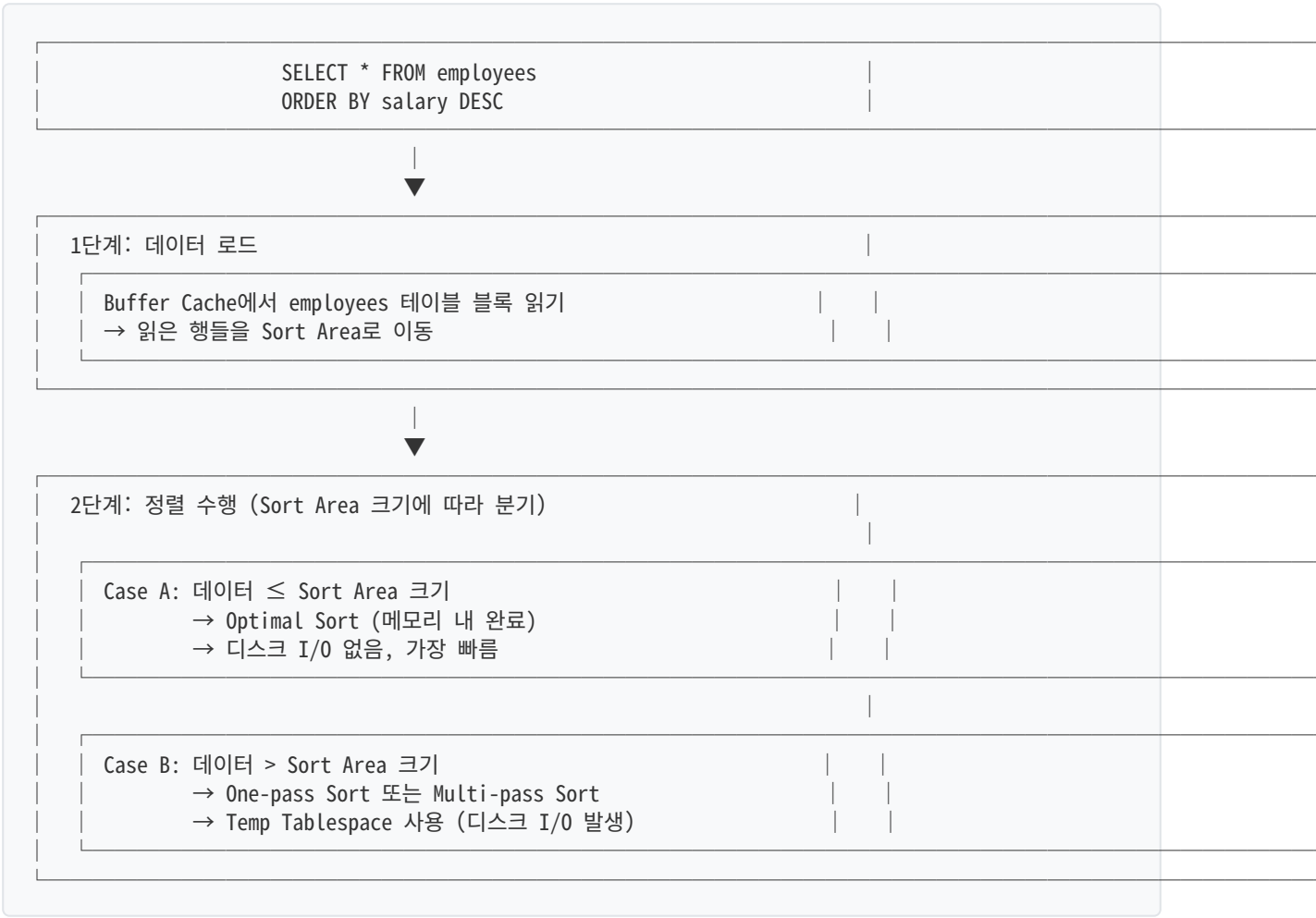
7. PGA 구성요소별 동작 메커니즘

SQL Work Areas는 메모리 집약적 연산을 수행하는 공간이다. 세 가지 주요 영역이 있다.

7.1 Sort Area

역할: ORDER BY, GROUP BY, DISTINCT, UNION, MERGE JOIN 등 정렬이 필요한 연산 수행

동작 흐름:



Optimal / One-pass / Multi-pass 정렬의 차이:

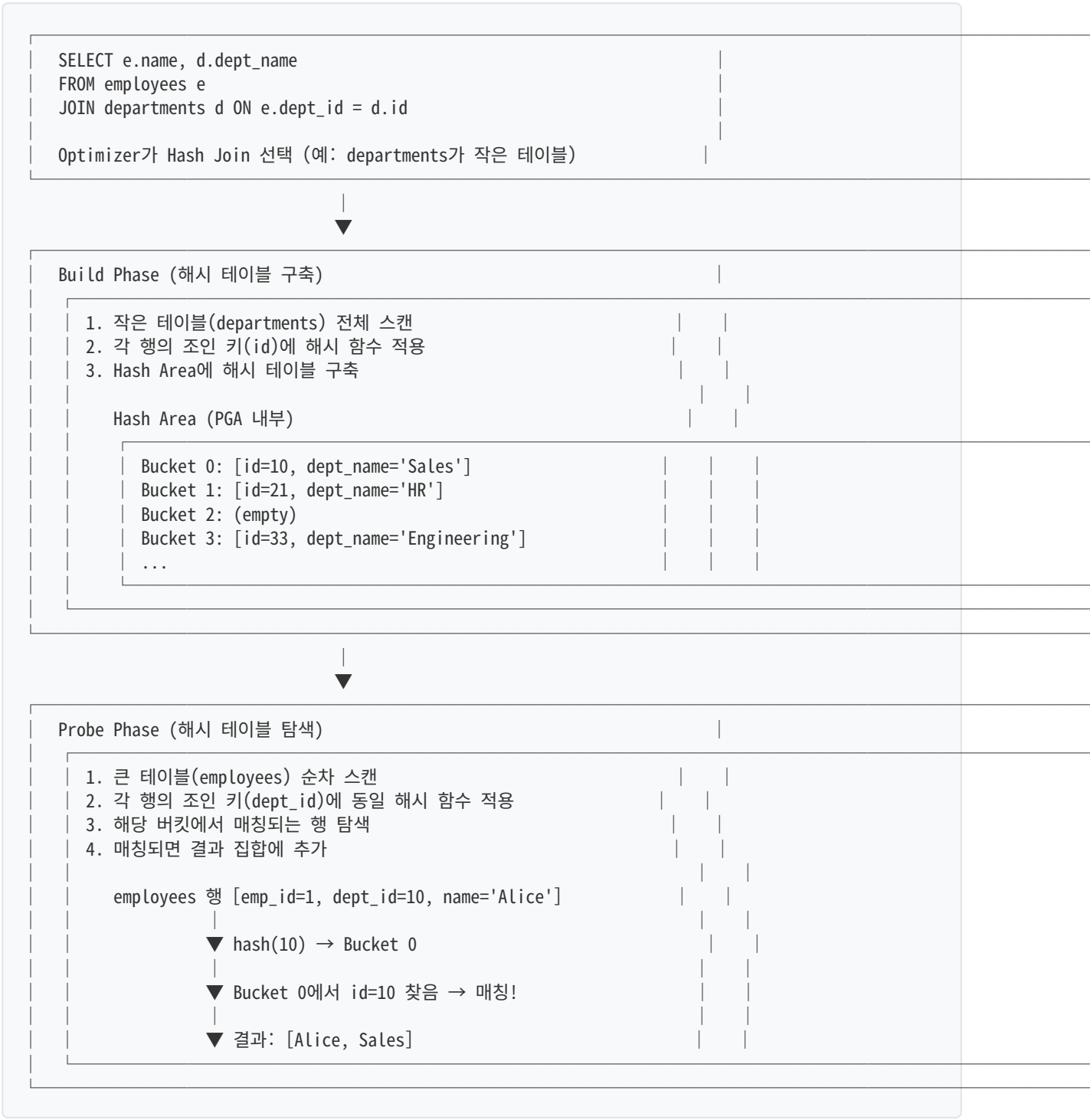
방식	조건	동작
Optimal	전체 데이터가 Sort Area에 적재	메모리에서 한 번에 정렬 완료. 디스크 I/O: 0
One-pass	Sort Area 크기의 약 2배 이내	1. Sort Area 크기만큼 정렬 2. 정렬된 청크를 Temp에 기록 3. 청크들을 한 번 병합하여 완료. 디스크 I/O: 1회 쓰기 + 1회 읽기
Multi-pass	Sort Area 크기의 2배 초과	1. 여러 청크로 분할하여 Temp에 기록 2. 청크 병합을 여러 번 반복. 디스크 I/O: 다수 발생 (성능 저하)

프로젝트 연결: 대용량 데이터 정렬 시 Sort Area가 부족하면 Temp Tablespace를 사용하게 되어 응답 시간이 급격히 증가한다. 이것이 페이징 쿼리에서 OFFSET이 커질수록 느려지는 원인 중 하나다.

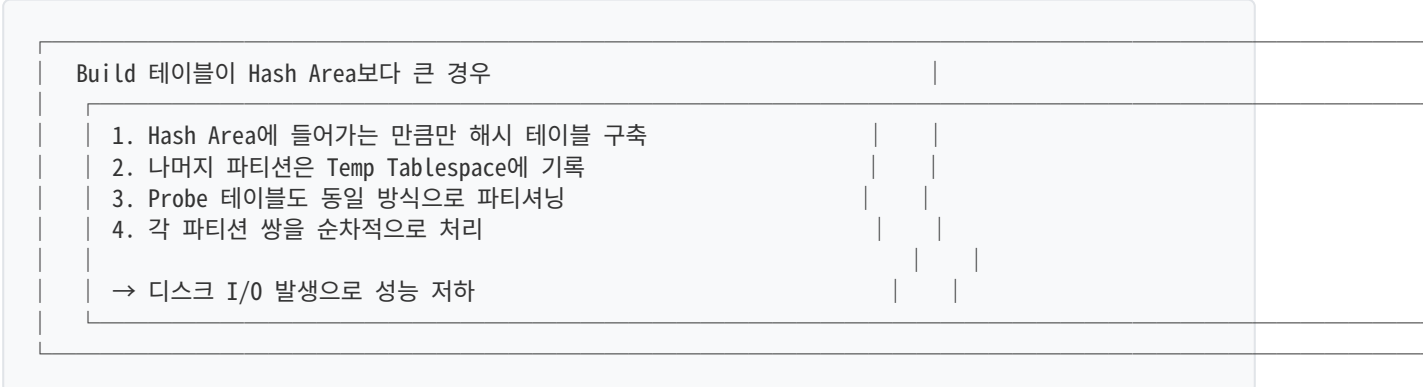
7.2 Hash Area

역할: Hash Join 수행 시 해시 테이블 구축 및 탐색

동작 흐름:

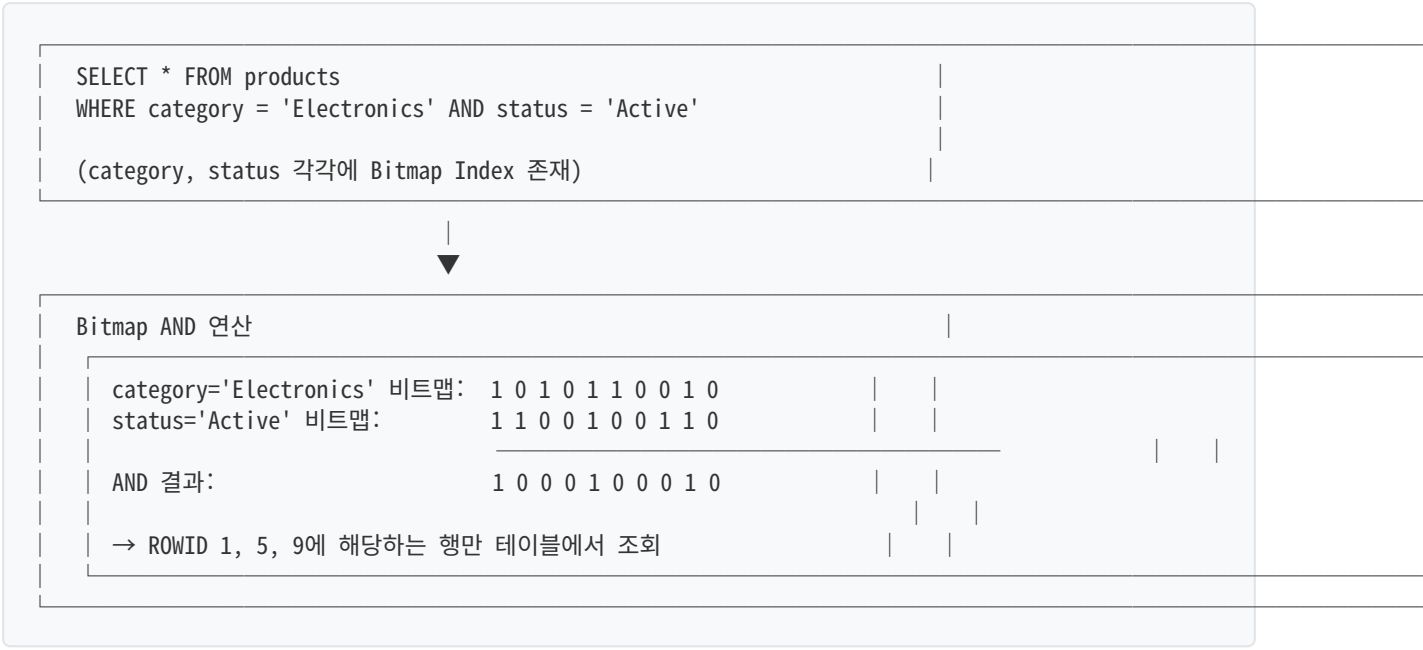


Hash Area 부족 시:



7.3 Bitmap Merge Area

역할: Bitmap Index 연산 시 비트맵 병합



Bitmap Merge Area는 이러한 비트맵 연산(AND, OR, MINUS)을 수행하는 공간이다. OLTP보다는 DW 환경에서 주로 사용된다.

7.4 학습 우선순위 판단

구성요소	프로젝트 연관성	면접 출현 가능성	학습 권장
Sort Area	중간 (ORDER BY 사용)	낮음	개념만
Hash Area	낮음	낮음	생략 가능
Bitmap Merge Area	없음 (DW 영역)	거의 없음	생략
Session Memory (UGA)	높음	중간	필수
Private SQL Area	높음	중간	필수

Hash Join, Bitmap Index 연산은 Oracle Optimizer가 실행 계획을 수립할 때 선택하는 **접근 방식(Access Path)**이다. 백엔드 개발자가 직접 제어하는 영역이 아니라, Optimizer가 통계 정보를 기반으로 자동 결정한다.

Wherehouse 프로젝트와 면접 목표에서 핵심은: 1. Connection 생성 비용 → Server Process + PGA 할당 2. N+1 문제의 아키텍처적 영향 → Library Cache, Connection 점유 시간 3. HikariCP Waiting 원인 → Dedicated Server 1:1 매핑, Pool 고갈

Sort Area에 대해 알아두면 좋은 수준:

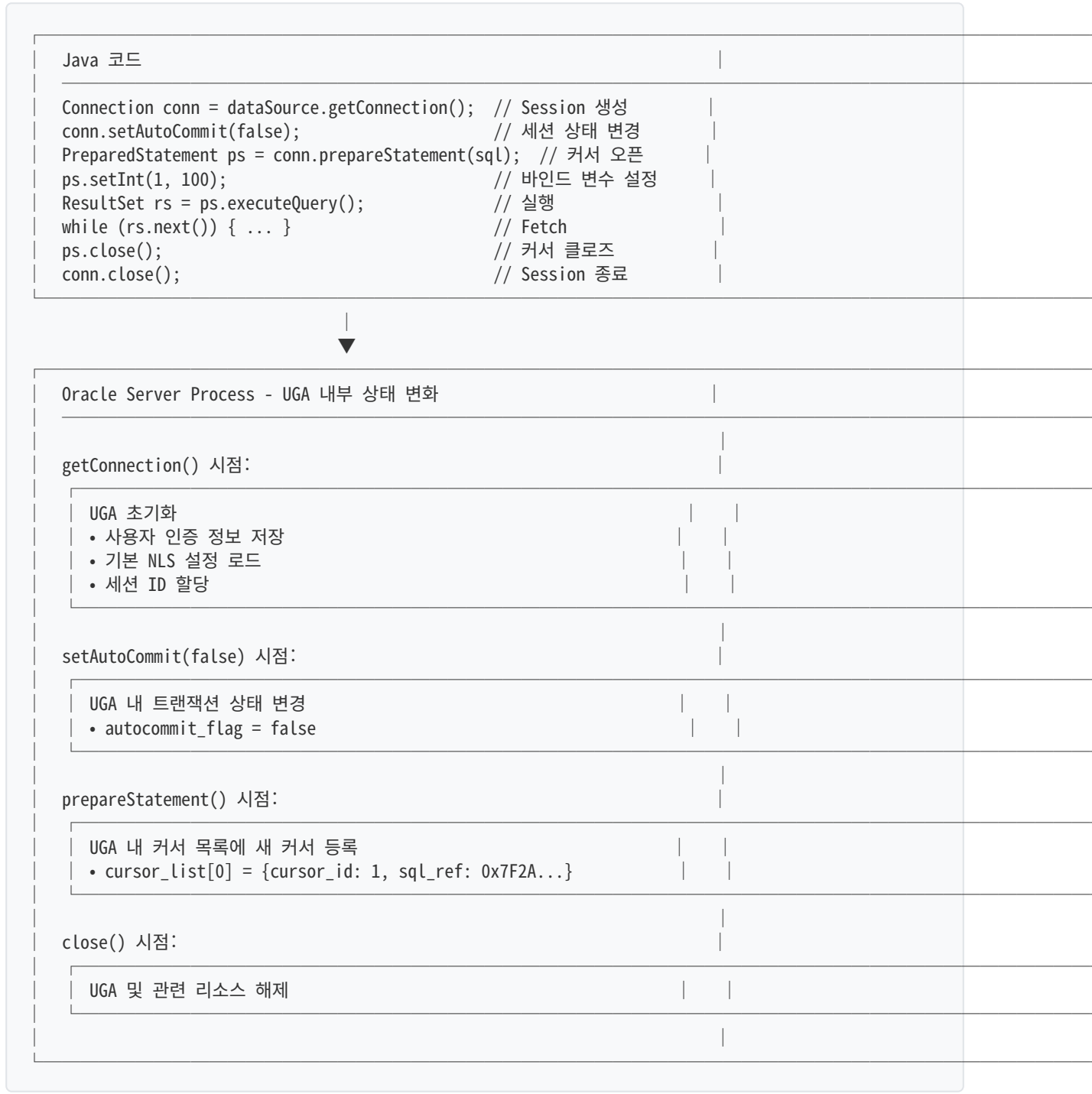
> "ORDER BY, GROUP BY 같은 정렬 연산은 PGA의 Sort Area에서 수행된다. 데이터가 Sort Area 크기를 초과하면 Temp Tablespace(디스크)를 사용하게 되어 성능이 저하된다."

---

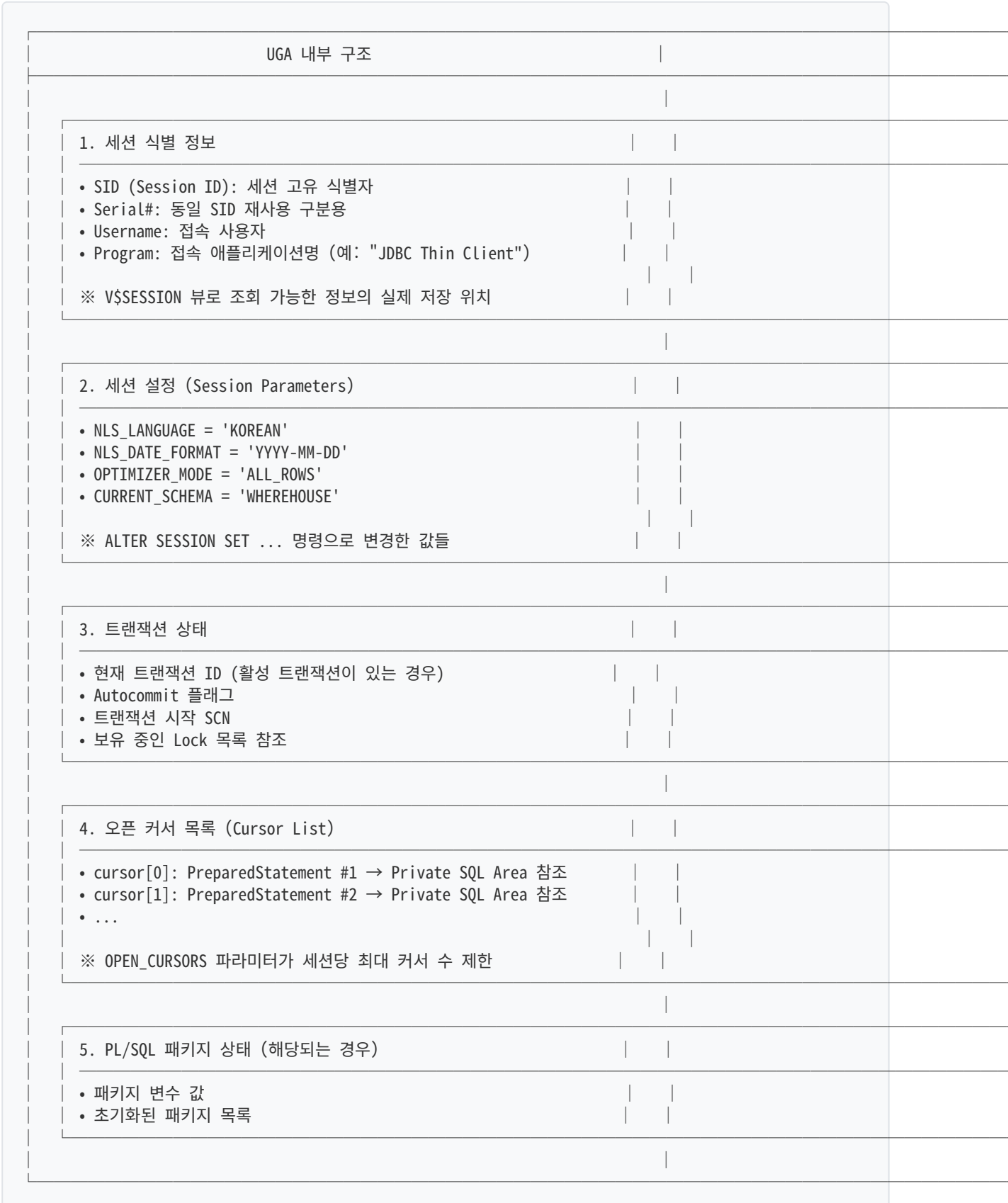
8. UGA (User Global Area) 상세

8.1 UGA의 정체: "세션"이라는 개념의 물리적 실체

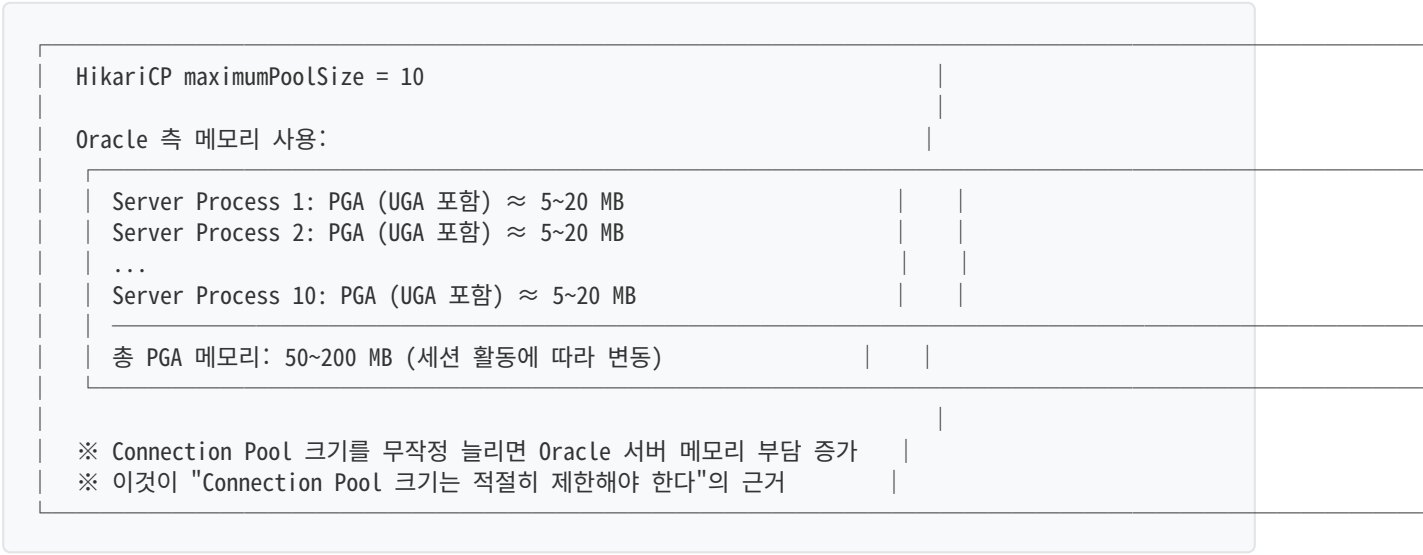
JDBC에서 Connection을 얻으면 Oracle 측에서는 Session이 생성된다. 이 Session의 상태 정보가 저장되는 물리적 공간이 UGA다.


Java 코드
Connection conn = dataSource.getConnection(); // Session 생성
conn.setAutoCommit(false); // 세션 상태 변경
PreparedStatement ps = conn.prepareStatement(sql); // 커서 오픈
ps.setInt(1, 100); // 바인드 변수 설정
ResultSet rs = ps.executeQuery(); // 실행
while (rs.next()) { ... } // Fetch
ps.close(); // 커서 클로즈
conn.close(); // Session 종료
Oracle Server Process - UGA 내부 상태 변화
getConnection() 시점:
UGA 초기화
• 사용자 인증 정보 저장
• 기본 NLS 설정 로드
• 세션 ID 할당
setAutoCommit(false) 시점:
UGA 내 트랜잭션 상태 변경
• autocommit_flag = false
prepareStatement() 시점:
UGA 내 커서 목록에 새 커서 등록
• cursor_list[0] = {cursor_id: 1, sql_ref: 0x7F2A...}
close() 시점:
UGA 및 관련 리소스 해제

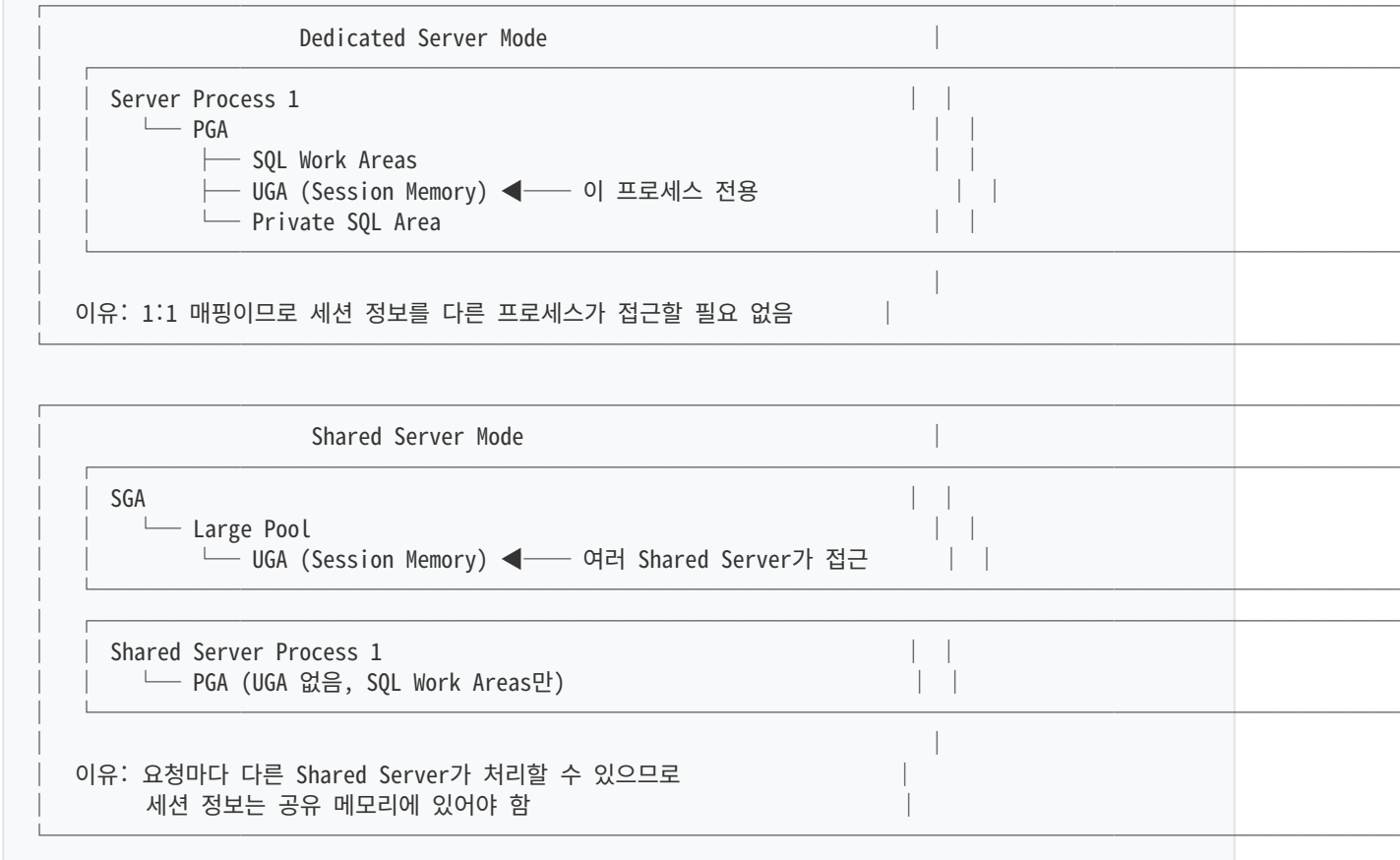
8.2 UGA에 저장되는 구체적 내용



8.3 UGA 크기가 Connection 비용에 미치는 영향



8.4 Dedicated Server vs Shared Server에서 UGA 위치



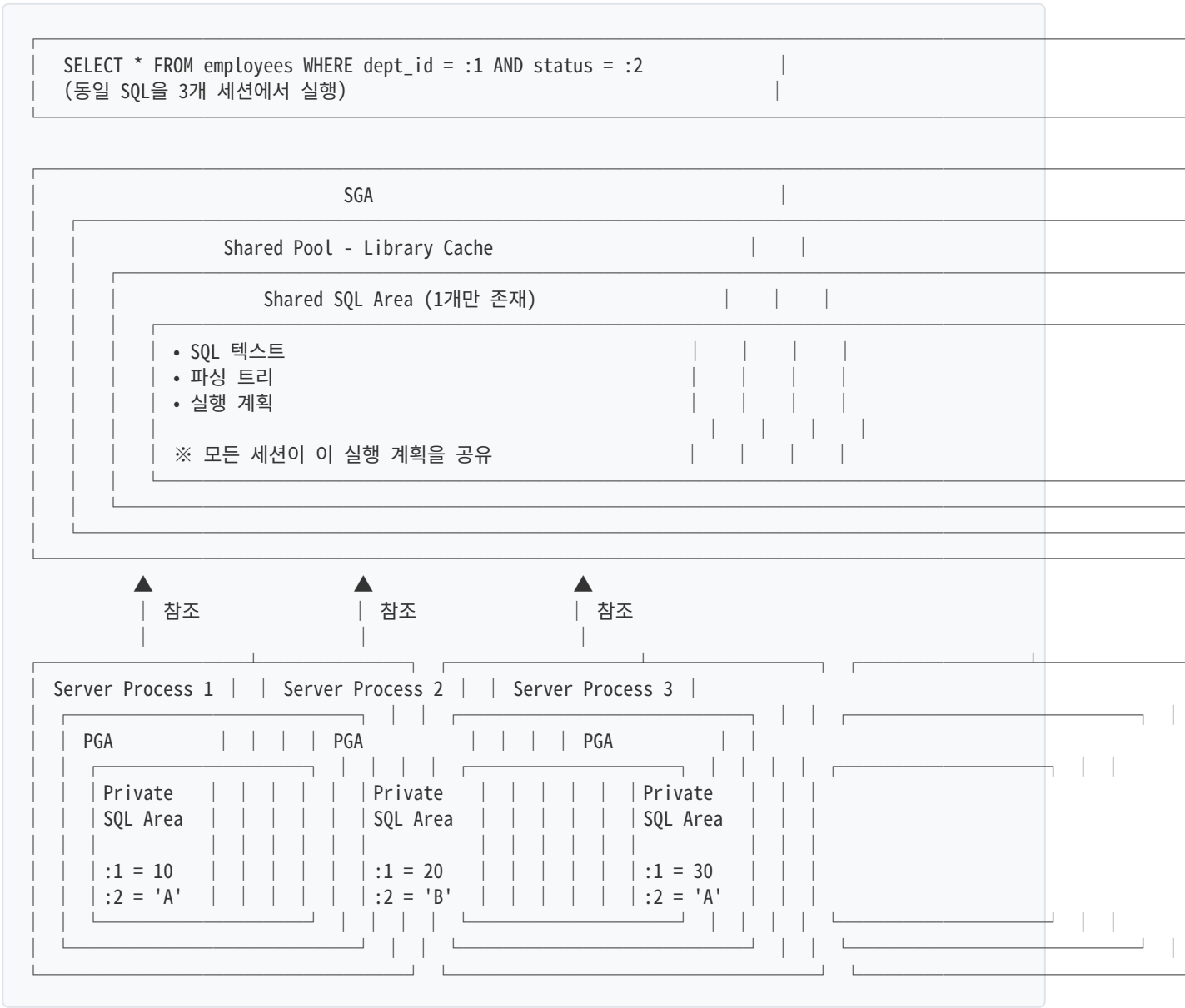
---

9. Private SQL Area 상세

9.1 Private SQL Area의 역할

Shared SQL Area(Library Cache)에는 SQL 텍스트, 파싱 트리, 실행 계획이 저장된다. 이것은 모든 세션이 공유하는 읽기 전용 템플릿이다.

Private SQL Area는 이 템플릿을 특정 세션이 실제로 실행할 때 필요한 세션 고유 데이터를 보관하는 공간이다.



핵심: - Shared SQL Area: SQL 텍스트 + 실행 계획 → 1개만 존재, 모든 세션 공유 - Private SQL Area: 바인드 변수 값 + 실행 상태 → 세션마다 별도 존재

9.2 Private SQL Area의 두 영역

9.3 Java 코드와 Private SQL Area 생명주기 매핑

```
// 1. PreparedStatement 생성 → Persistent Area 할당
PreparedStatement ps = conn.prepareStatement(
    "SELECT * FROM employees WHERE dept_id = ? AND status = ?"
);

// 2. 바인드 변수 설정 → Persistent Area에 값 저장
ps.setInt(1, 10);           // :1 = 10
ps.setString(2, "ACTIVE");  // :2 = 'ACTIVE'

// 3. 실행 → Runtime Area 할당
ResultSet rs = ps.executeQuery();

// 4. Fetch → Runtime Area 내 현재 위치 갱신
while (rs.next()) {
    String name = rs.getString("name");
}

// 5. ResultSet 종료 → Runtime Area 해제
rs.close();

// 6. 동일 PreparedStatement 재사용 (바인드 값만 변경)
ps.setInt(1, 20);           // Persistent Area 값 덮어쓰기
ps.setString(2, "ACTIVE");
rs = ps.executeQuery();     // 새로운 Runtime Area 할당
// ...

// 7. PreparedStatement 종료 → Persistent Area 해제
ps.close();
```

9.4 프로젝트 연결: N+1 쿼리와 Private SQL Area

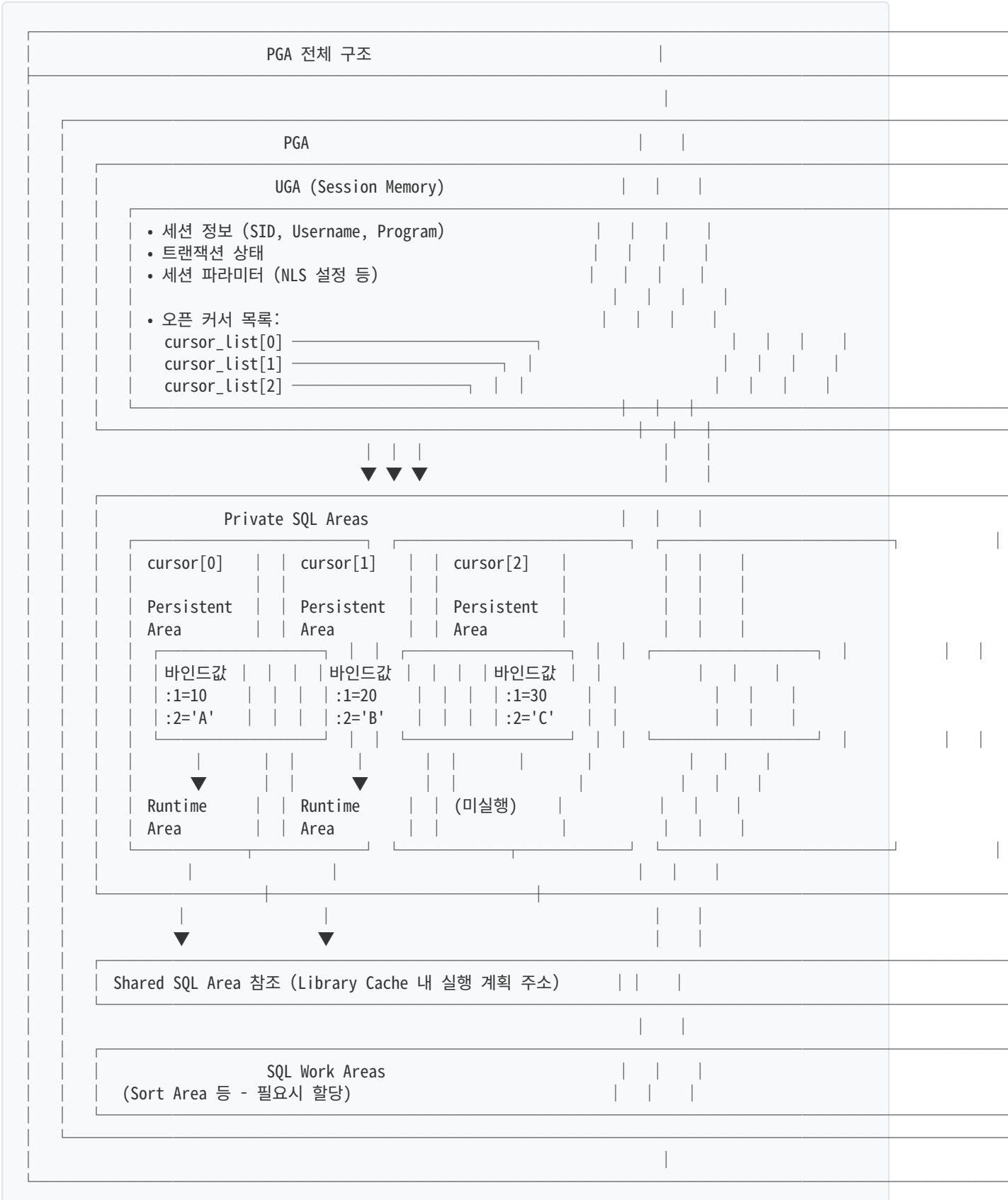
1차 테스트 (파라미터 개수 가변):

```
┌──────────────────────────────────────────────────────────────────────────────┐
│ Hibernate가 생성하는 쿼리 패턴                                               │
│ SELECT * FROM review_statistics WHERE property_id IN (?, ?, ..., ?)          │
├──────────────────────────────────────────────────────────────────────────────┤
│                                                                              │
│ 쿼리 1: IN (?, ?, ?, ?, ?)          → 5개 placeholder                        │
│ 쿼리 2: IN (?, ?, ?, ?, ?, ?, ?)    → 7개 placeholder                        │
│ 쿼리 3: IN (?, ?, ?)                → 3개 placeholder                        │
│ ...                                                                          │
│ 쿼리 25: IN (?, ?, ?, ?, ?, ?, ?, ?) → 8개 placeholder                       │
│                                                                              │
│ Oracle 관점:                                                                 │
│ • placeholder 개수가 다르면 SQL 텍스트가 다름                                │
│ • 다른 SQL → Library Cache에 별도의 Shared SQL Area 생성                     │
│ • 25개의 서로 다른 실행 계획 생성 (Hard Parse 25회)                          │
│                                                                              │
│ Private SQL Area:                                                            │
│ • 각 쿼리마다 새로운 Private SQL Area 할당                                   │
│ • 각각 다른 Shared SQL Area를 참조                                           │
│                                                                              │
└──────────────────────────────────────────────────────────────────────────────┘
```

3차 테스트 (1000개 고정):

```
┌──────────────────────────────────────────────────────────────────────────────┐
│ Chunk 방식: 항상 1000개 파라미터                                             │
│ SELECT * FROM review_statistics                                              │
│ WHERE property_id IN (?,?,?,...,?)  -- 항상 1000개 placeholder               │
├──────────────────────────────────────────────────────────────────────────────┤
│                                                                              │
│ Library Cache (SGA):                                                         │
│  ┌──────────────────────────────────────────────────────────────────┐        │
│  │ Shared SQL Area (1개만 존재)                                     │        │
│  │ • SQL 텍스트: SELECT ... IN (?,?,?,...,?) -- 1000개              │        │
│  │ • 실행 계획: 1번 Hard Parse 후 고정                              │        │
│  └──────────────────────────────────────────────────────────────────┘        │
│                                                                              │
│ PGA (각 실행 시):                                                            │
│  ┌──────────────────────────────────────────────────────────────────┐        │
│  │ 쿼리 1 실행:                                                     │        │
│  │ • Private SQL Area 할당                                          │        │
│  │ • Persistent Area: [:1=id1, :2=id2, ..., :1000=id1000]           │        │
│  │ • Shared SQL Area 참조: 0x7F2A... → Soft Parse                   │        │
│  │                                                                  │        │
│  │ 쿼리 2 실행:                                                     │        │
│  │ • Private SQL Area 할당 (또는 재사용)                            │        │
│  │ • Persistent Area: [:1=id1001, :2=id1002, ..., :1000=id2000]     │        │
│  │ • Shared SQL Area 참조: 0x7F2A... (동일) → Soft Parse            │        │
│  │                                                                  │        │
│  │ ... 9번 반복                                                     │        │
│  └──────────────────────────────────────────────────────────────────┘        │
│                                                                              │
│ 결과:                                                                        │
│ • Hard Parse: 1회 (최초 실행 시)                                             │
│ • Soft Parse: 8회 (2~9번째 실행)                                             │
│ • 1차 테스트 대비 Parse 부하 감소                                            │
│                                                                              │
└──────────────────────────────────────────────────────────────────────────────┘
```

9.5 UGA와 Private SQL Area의 관계

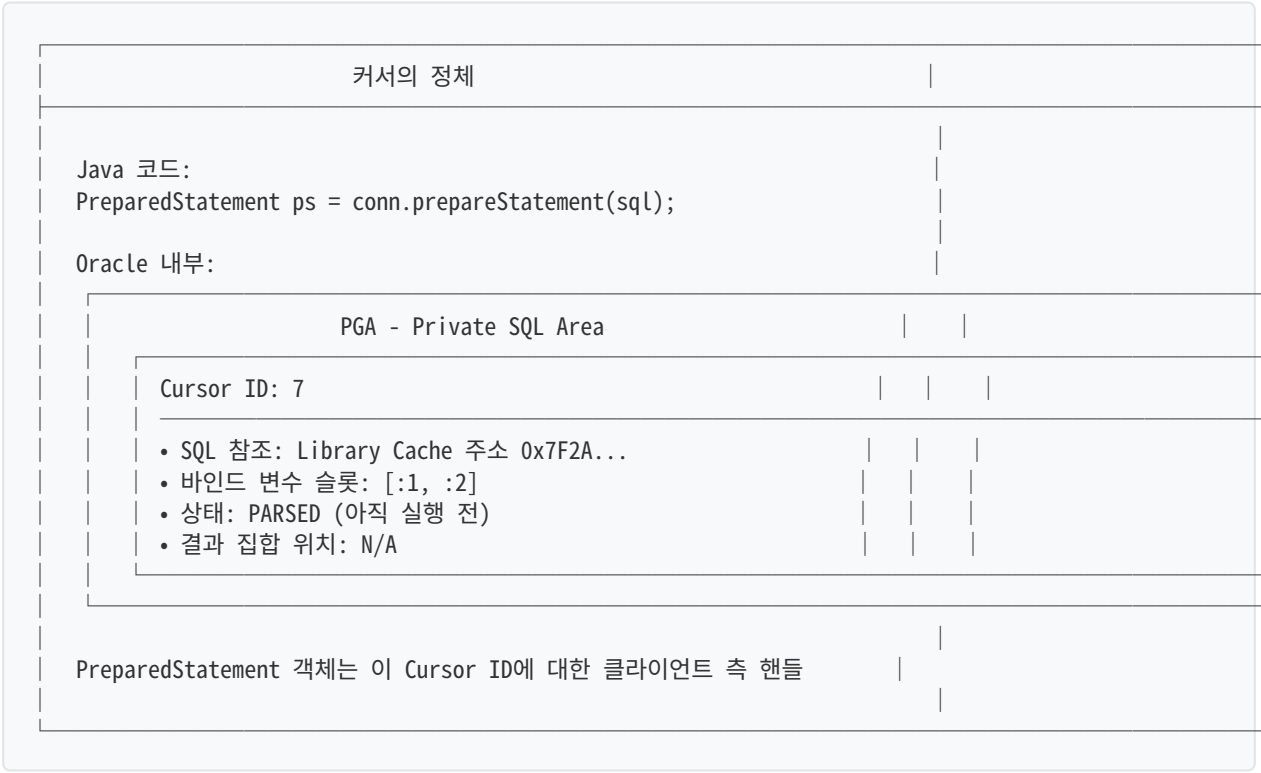


구조적 관계: 1. UGA는 세션 전체의 상태를 관리하며, 오픈된 커서 목록을 유지한다 2. 각 커서는 대응되는 Private SQL Area를 가리킨다 3. Private SQL Area는 해당 SQL의 바인드 변수와 실행 상태를 보관한다 4. 모든 Private SQL Area는 Library Cache의 Shared SQL Area를 참조한다

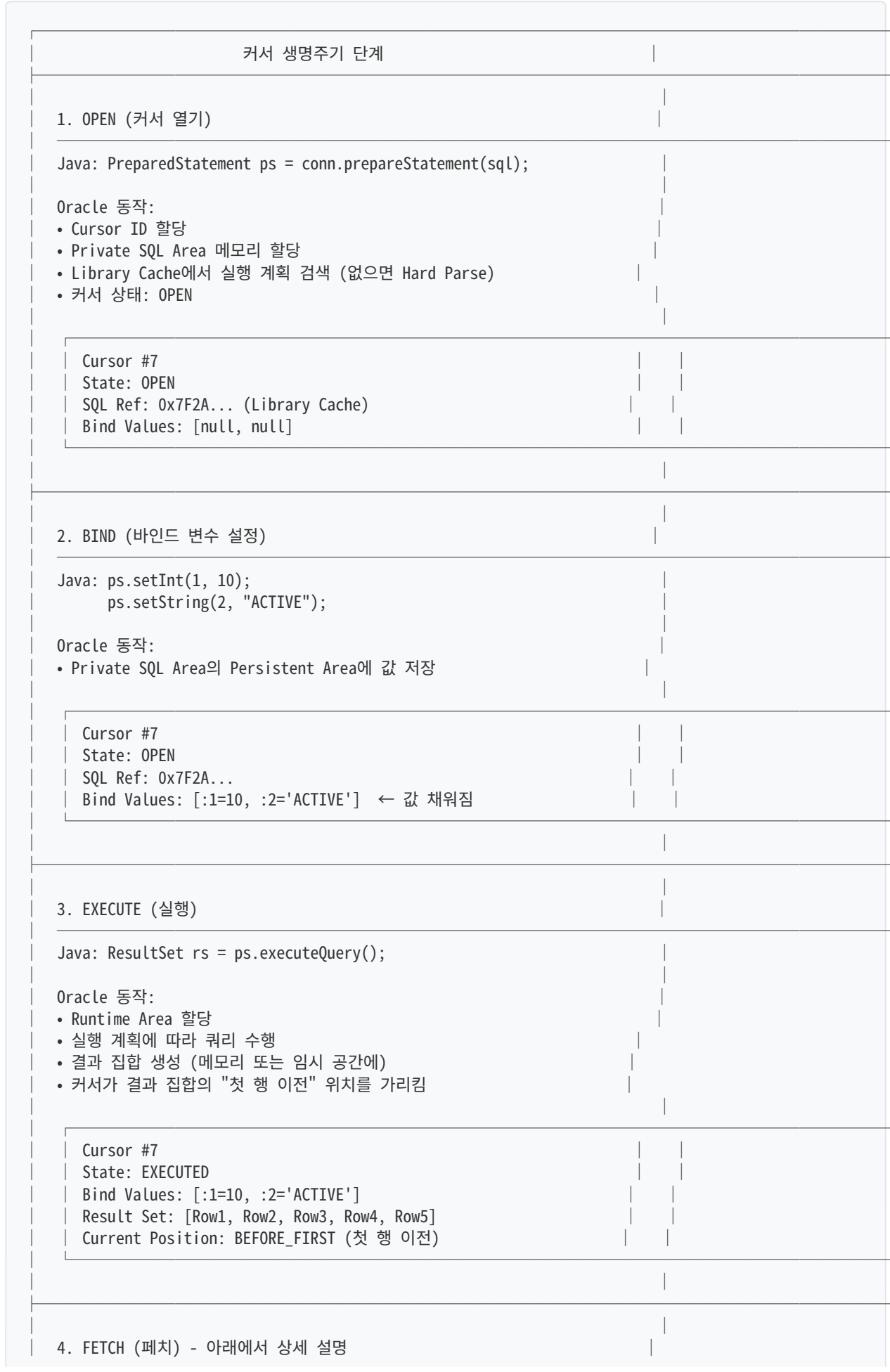
10. Cursor (커서)

10.1 커서의 정의

커서는 SQL 실행을 제어하고 결과 집합을 탐색하기 위한 Oracle 내부의 제어 구조다. 물리적으로는 Private SQL Area에 대한 핸들(handle) 또는 포인터다.



10.2 커서의 생명주기


커서 생명주기 단계
1. OPEN (커서 열기)
Java: PreparedStatement ps = conn.prepareStatement(sql);
Oracle 동작:
• Cursor ID 할당
• Private SQL Area 메모리 할당
• Library Cache에서 실행 계획 검색 (없으면 Hard Parse)
• 커서 상태: OPEN
Cursor #7
State: OPEN
SQL Ref: 0x7F2A... (Library Cache)
Bind Values: [null, null]
2. BIND (바인드 변수 설정)
Java: ps.setInt(1, 10);
ps.setString(2, "ACTIVE");
Oracle 동작:
• Private SQL Area의 Persistent Area에 값 저장
Cursor #7
State: OPEN
SQL Ref: 0x7F2A...
Bind Values: [:1=10, :2='ACTIVE'] ← 값 채워짐
3. EXECUTE (실행)
Java: ResultSet rs = ps.executeQuery();
Oracle 동작:
• Runtime Area 할당
• 실행 계획에 따라 쿼리 수행
• 결과 집합 생성 (메모리 또는 임시 공간에)
• 커서가 결과 집합의 "첫 행 이전" 위치를 가리킴
Cursor #7
State: EXECUTED
Bind Values: [:1=10, :2='ACTIVE']
Result Set: [Row1, Row2, Row3, Row4, Row5]
Current Position: BEFORE_FIRST (첫 행 이전)
4. FETCH (페치) - 아래에서 상세 설명

```
┌──────────────────────────────────────────────────────────────────────┐
│ Java: while (rs.next()) { ... }                                      │
└──────────────────────────────────────────────────────────────────────┘
                                    │
                                    ▼
┌──────────────────────────────────────────────────────────────────────┐
│ 5. CLOSE (커서 닫기)                                                 │
├──────────────────────────────────────────────────────────────────────┤
│ Java: ps.close();                                                    │
│                                                                      │
│ Oracle 동작:                                                         │
│ • Runtime Area 해제                                                  │
│ • Persistent Area 해제                                               │
│ • Cursor ID 반환 (재사용 가능)                                       │
│ • UGA의 커서 목록에서 제거                                           │
└──────────────────────────────────────────────────────────────────────┘
```

10.3 커서와 UGA의 관계

```
┌──────────────────────────────────────────────────────────────────────┐
│                        UGA 내 커서 관리                              │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│ Java 코드:                                                           │
│ PreparedStatement ps1 = conn.prepareStatement(sql1);                 │
│ PreparedStatement ps2 = conn.prepareStatement(sql2);                 │
│ PreparedStatement ps3 = conn.prepareStatement(sql3);                 │
│                                                                      │
│ Oracle UGA:                                                          │
│ ┌──────────────────────────────────────────────────────────────────┐ │
│ │ Session ID: 47                                                   │ │
│ │ ──────────────────────────────────────────────────────────────── │ │
│ │ Open Cursor Count: 3                                             │ │
│ │ Max Cursors (OPEN_CURSORS): 300                                  │ │
│ │                                                                  │ │
│ │ Cursor List:                                                     │ │
│ │ ┌──────────────────────────────────────────────────────────────┐ │ │
│ │ │ [0] Cursor #7  → Private SQL Area for sql1                   │ │ │
│ │ │ [1] Cursor #12 → Private SQL Area for sql2                   │ │ │
│ │ │ [2] Cursor #15 → Private SQL Area for sql3                   │ │ │
│ │ └──────────────────────────────────────────────────────────────┘ │ │
│ └──────────────────────────────────────────────────────────────────┘ │
│                                                                      │
│ ※ OPEN_CURSORS 파라미터: 세션당 동시에 열 수 있는 최대 커서 수       │
│ ※ 초과 시 ORA-01000: maximum open cursors exceeded 에러 발생        │
└──────────────────────────────────────────────────────────────────────┘
```

---

11. Fetch (페치)

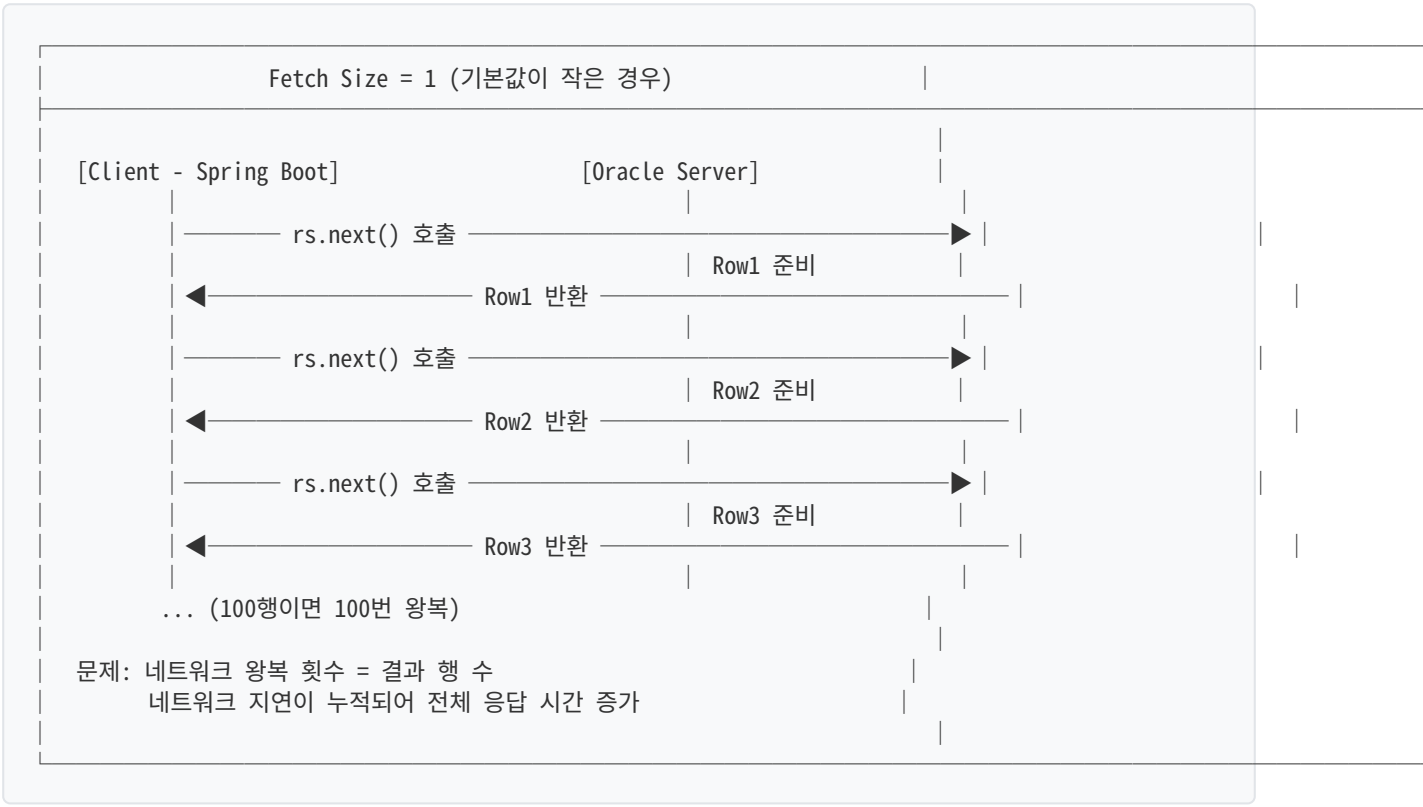
11.1 페치의 정의

페치는 커서가 가리키는 결과 집합에서 행(row)을 클라이언트로 가져오는 작업이다.

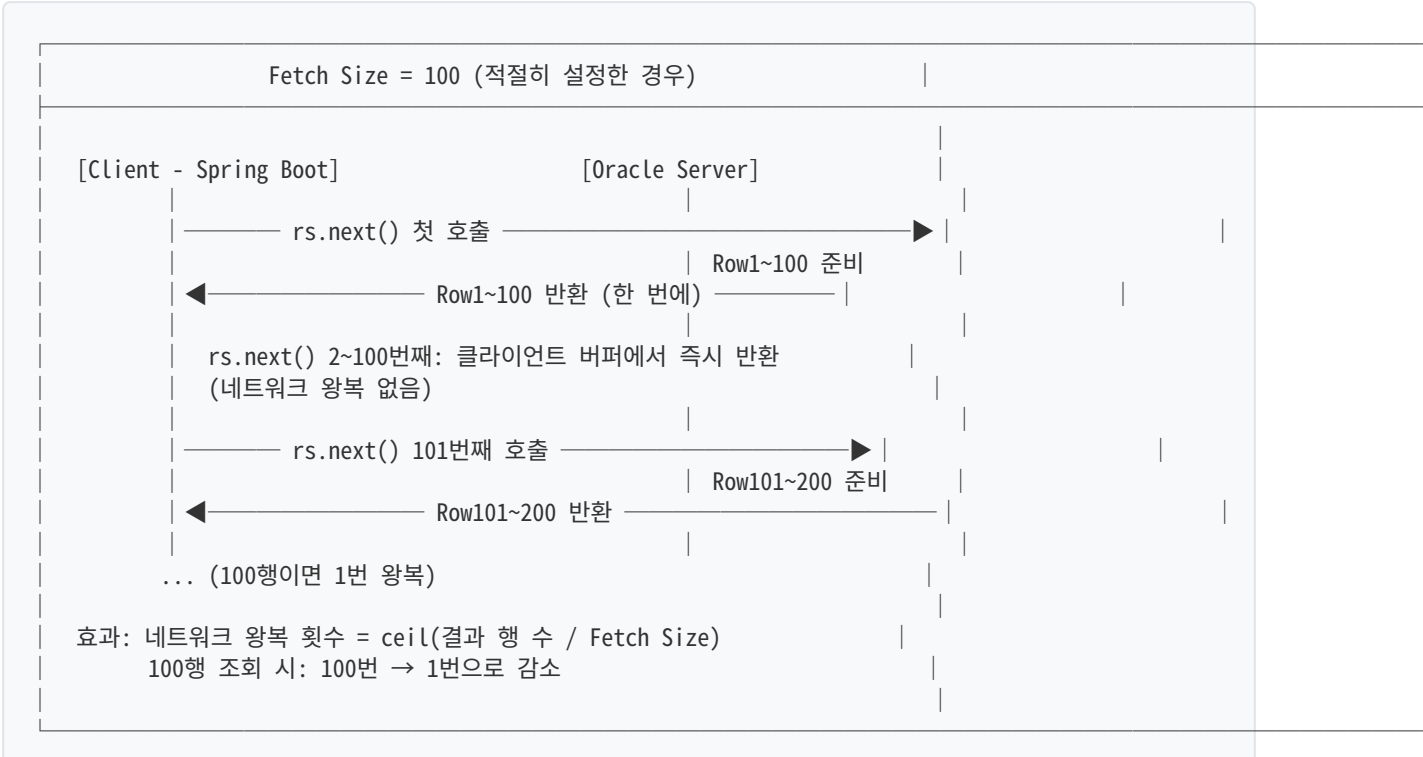
11.2 페치와 네트워크: Fetch Size의 의미

페치는 **네트워크 왕복(Round Trip)**을 수반한다. 여기서 성능 최적화 포인트가 발생한다.

극단적인 예로 Fetch Size = 1인 경우:



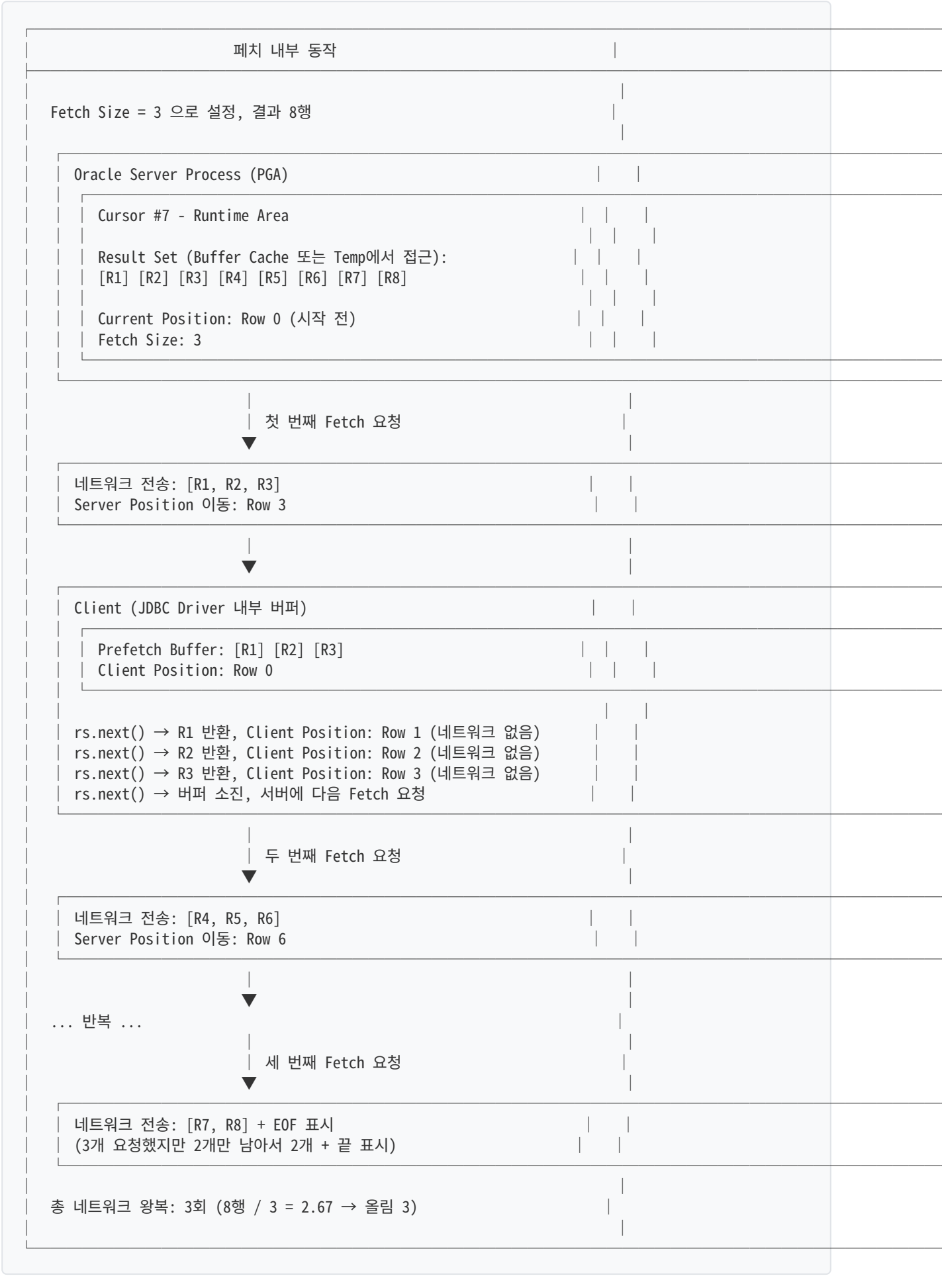
Fetch Size = 100 (적절히 설정한 경우):



11.3 Fetch Size 설정 방법

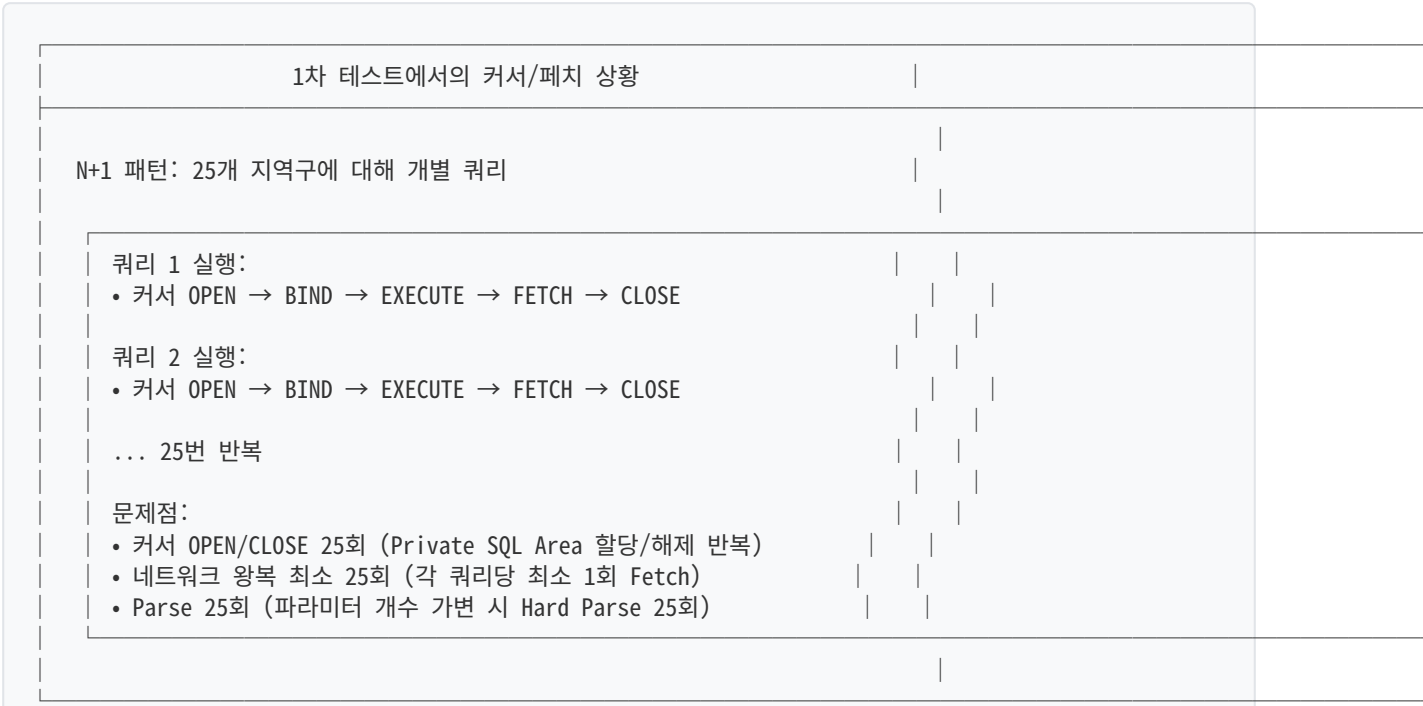
```
// 방법 1: Statement 레벨 설정
PreparedStatement ps = conn.prepareStatement(sql);
ps.setFetchSize(100);  // 한 번에 100행씩 가져옴
ResultSet rs = ps.executeQuery();

// 방법 2: ResultSet 레벨 설정
ResultSet rs = ps.executeQuery();
rs.setFetchSize(100);

// 방법 3: Connection 레벨 기본값 (Oracle JDBC)
// oracle.jdbc.defaultRowPrefetch 시스템 프로퍼티
// 또는 OracleConnection.setDefaultRowPrefetch(100)
```

11.4 페치 동작의 내부 구조

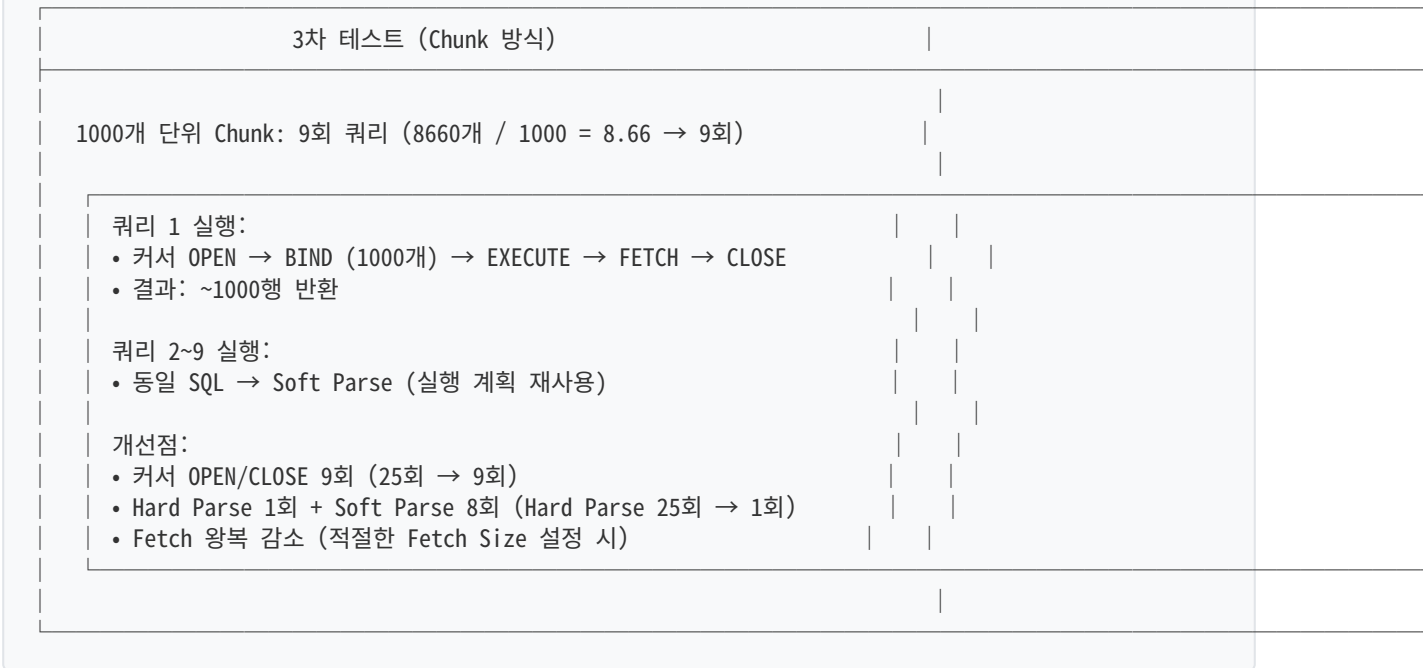

페치 내부 동작
Fetch Size = 3 으로 설정, 결과 8행
Oracle Server Process (PGA)
Cursor #7 - Runtime Area
Result Set (Buffer Cache 또는 Temp에서 접근):
[R1] [R2] [R3] [R4] [R5] [R6] [R7] [R8]
Current Position: Row 0 (시작 전)
Fetch Size: 3
첫 번째 Fetch 요청
네트워크 전송: [R1, R2, R3]
Server Position 이동: Row 3
Client (JDBC Driver 내부 버퍼)
Prefetch Buffer: [R1] [R2] [R3]
Client Position: Row 0
rs.next() → R1 반환, Client Position: Row 1 (네트워크 없음)
rs.next() → R2 반환, Client Position: Row 2 (네트워크 없음)
rs.next() → R3 반환, Client Position: Row 3 (네트워크 없음)
rs.next() → 버퍼 소진, 서버에 다음 Fetch 요청
두 번째 Fetch 요청
네트워크 전송: [R4, R5, R6]
Server Position 이동: Row 6
... 반복 ...
세 번째 Fetch 요청
네트워크 전송: [R7, R8] + EOF 표시
(3개 요청했지만 2개만 남아서 2개 + 끝 표시)
총 네트워크 왕복: 3회 (8행 / 3 = 2.67 → 올림 3)

11.5 프로젝트 연결: N+1과 Fetch

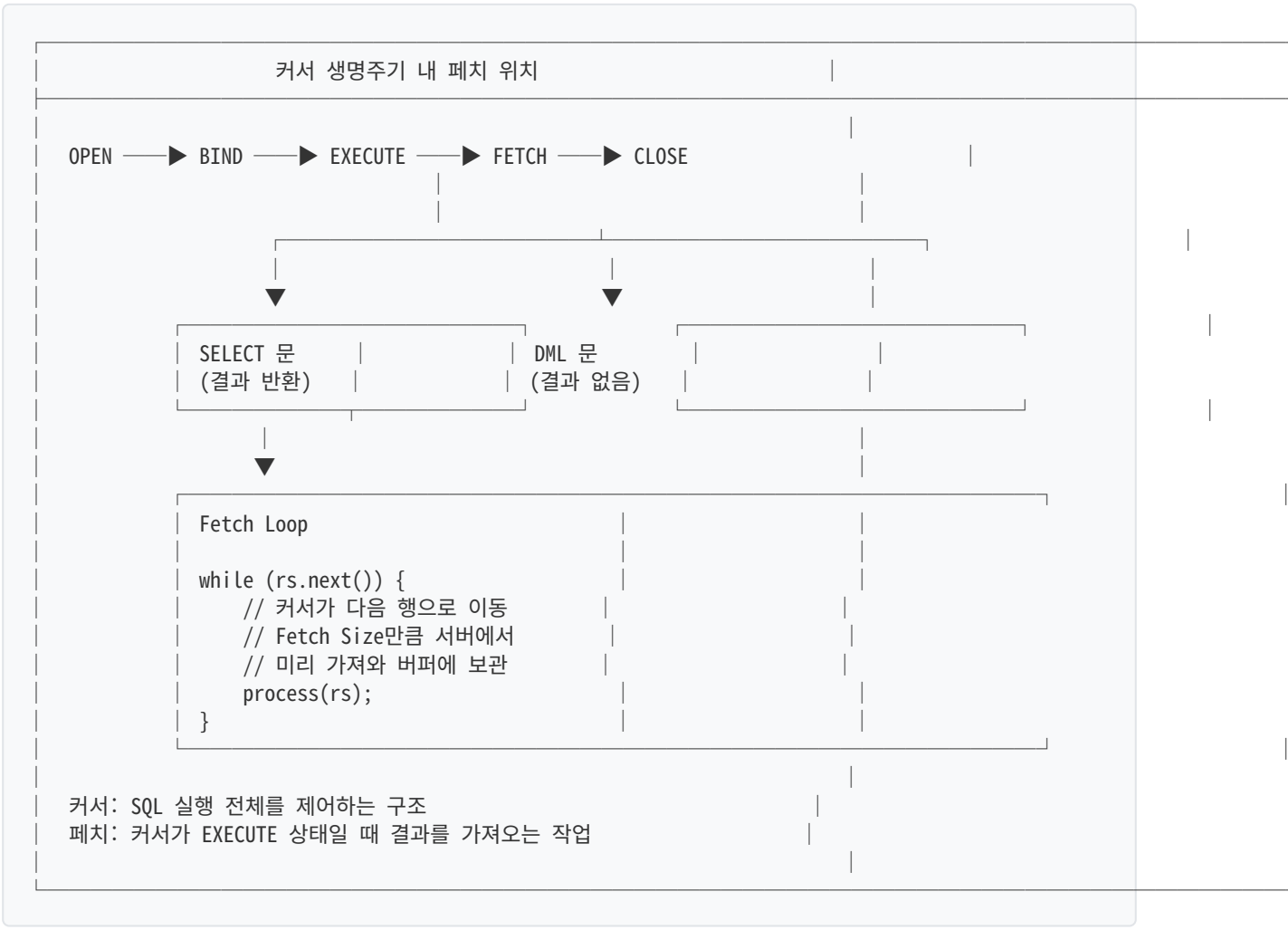
1차 테스트에서의 커서/페치 상황:



3차 테스트 (Chunk 방식):



11.6 정리: 커서와 페치의 관계

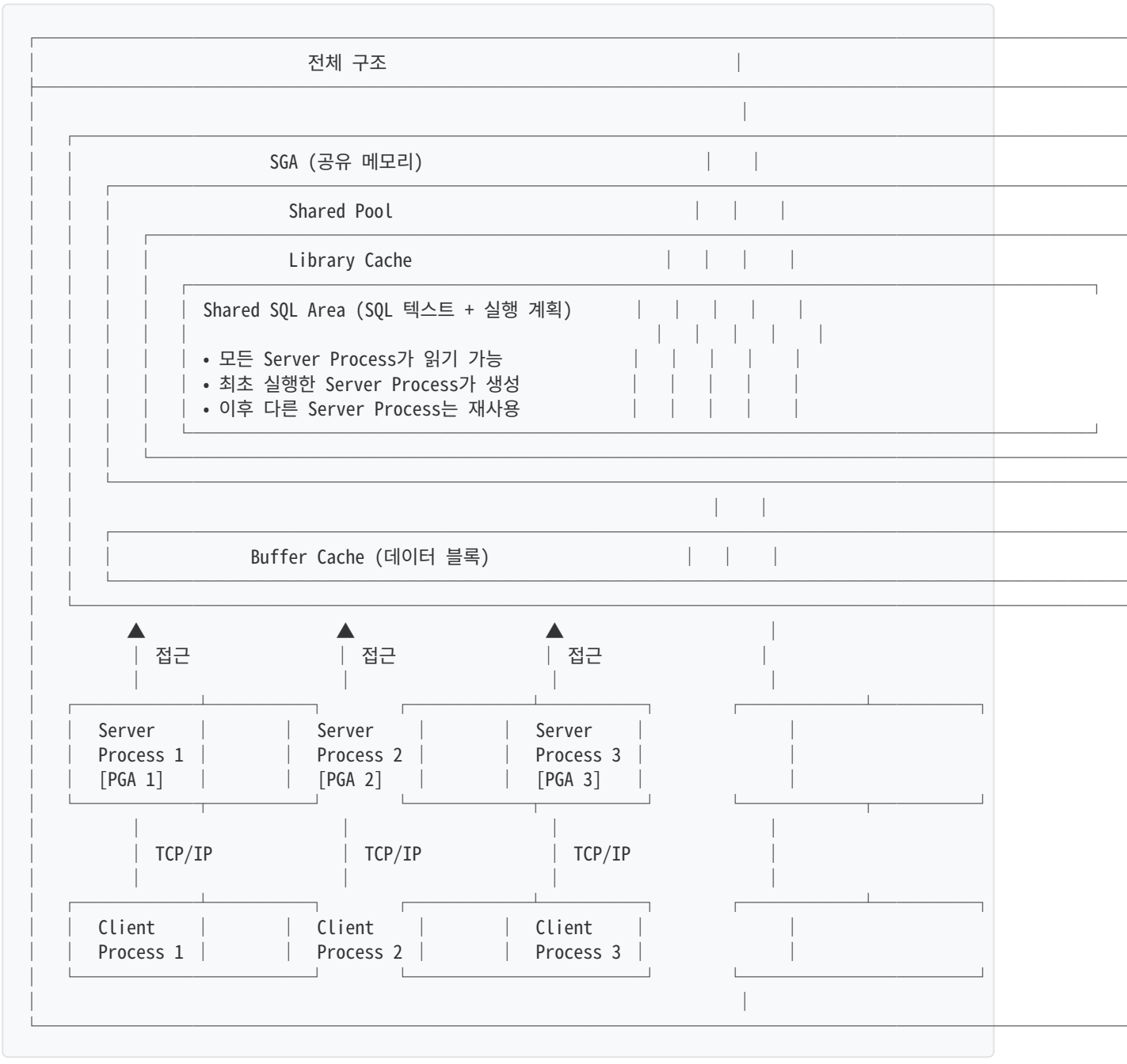


개념	역할	Java 대응	Oracle 내부
커서	SQL 실행 제어 구조	PreparedStatement	Private SQL Area + Cursor ID
페치	결과 행 가져오기	rs.next()	네트워크 전송 + 위치 이동
Fetch Size	한 번에 가져올 행 수	ps.setFetchSize(n)	네트워크 왕복 최적화

---

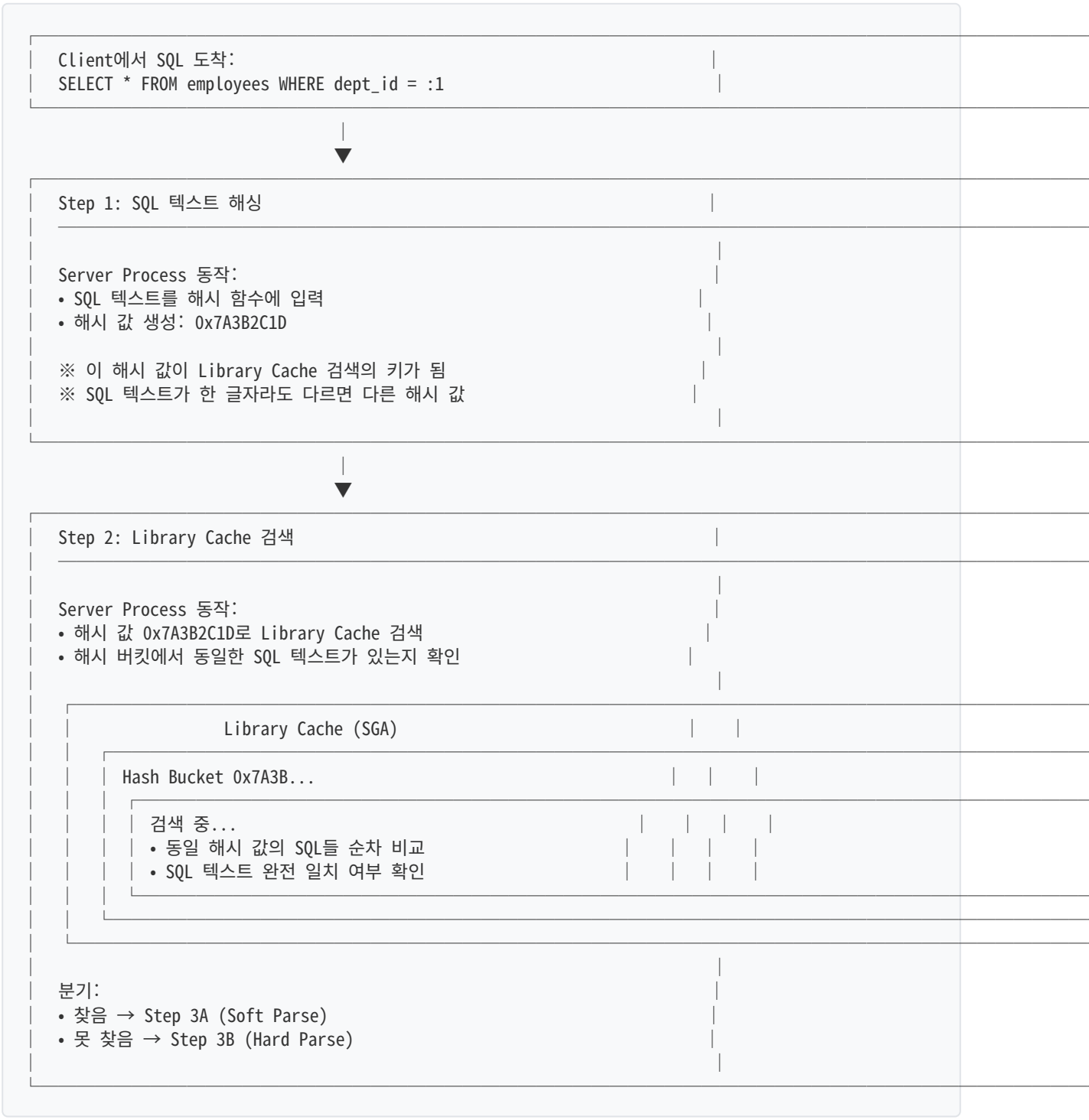
12. Server Process와 SGA의 상호작용

12.1 전체 구조: 누가 무엇을 어디에 저장하는가

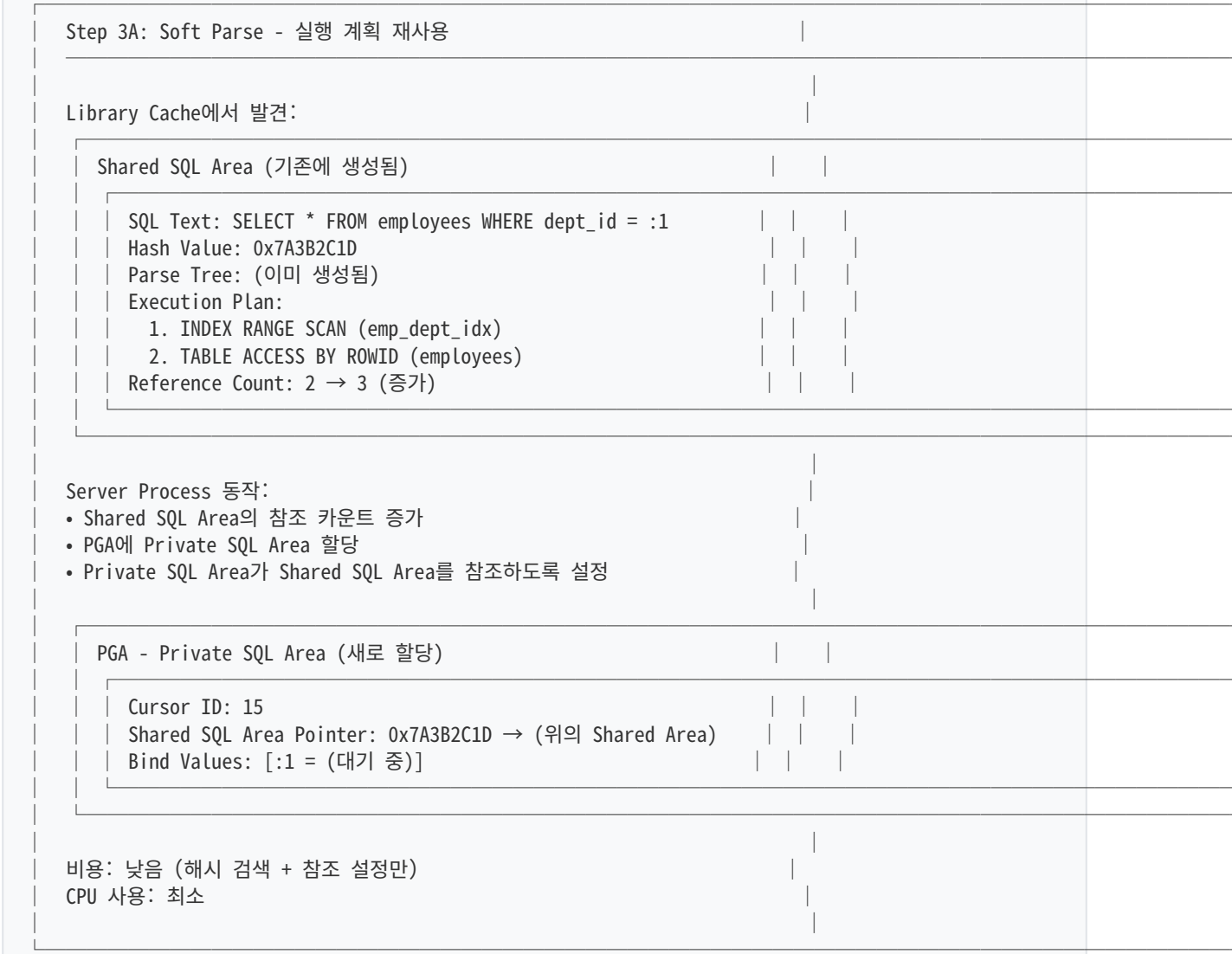


핵심: 1. Server Process가 SQL 파싱, 실행 계획 수립을 수행한다 2. 수립된 실행 계획은 SGA의 Library Cache에 저장된다 3. 다른 Server Process는 Library Cache를 조회하여 기존 실행 계획을 재사용한다

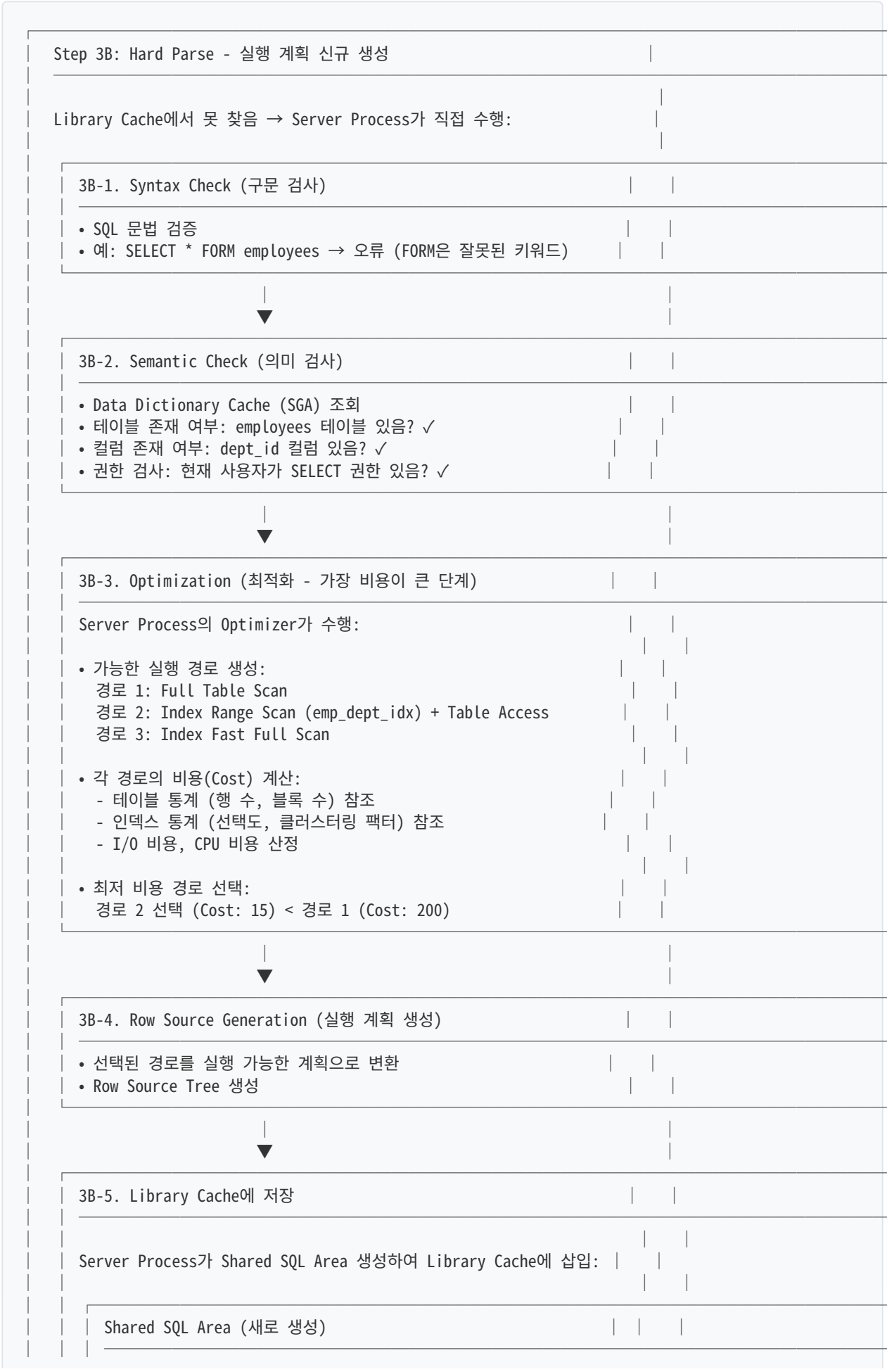
12.2 SQL 실행 시 Server Process의 동작 흐름

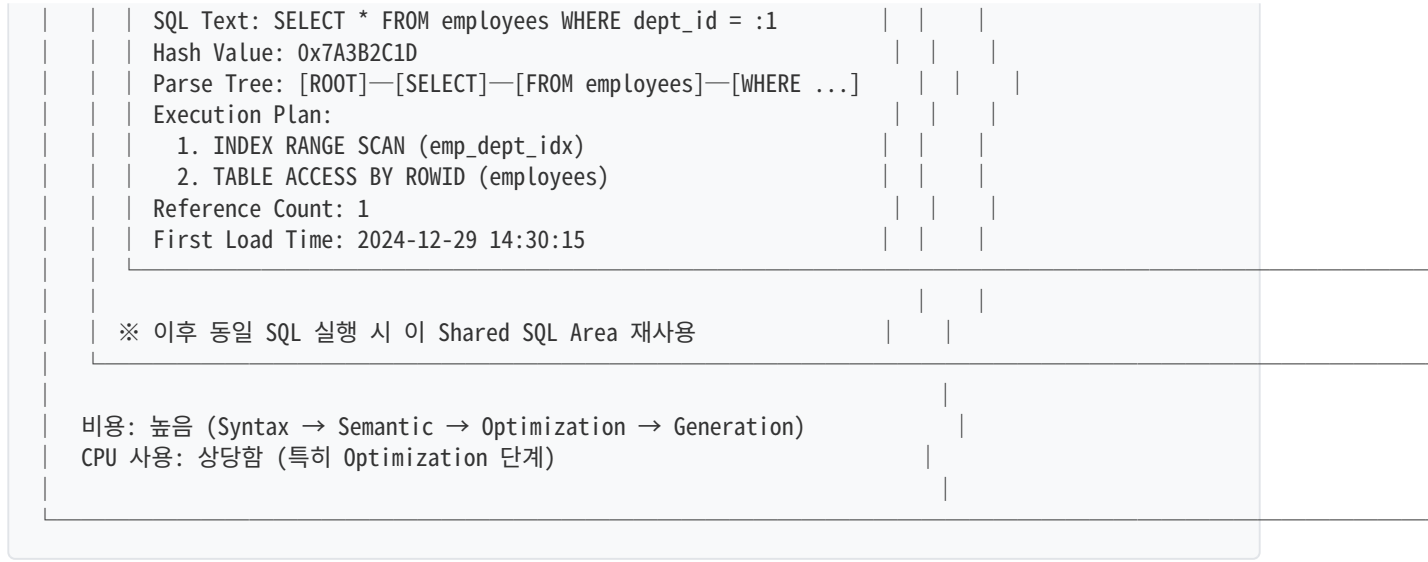


12.3 Soft Parse 경로 (Library Cache Hit)



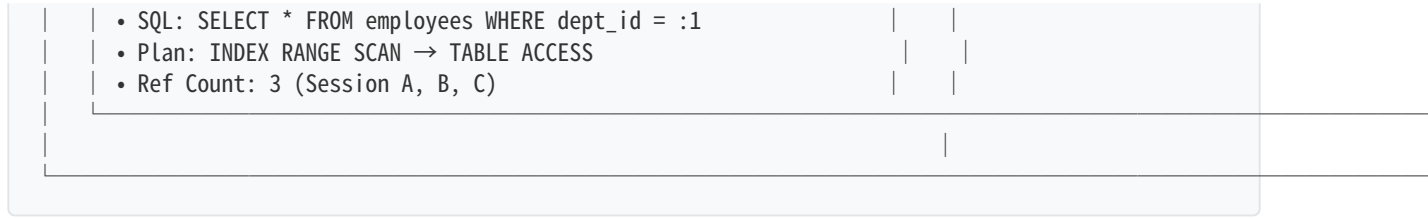
12.4 Hard Parse 경로 (Library Cache Miss)


Step 3B: Hard Parse - 실행 계획 신규 생성
Library Cache에서 못 찾음 → Server Process가 직접 수행:
3B-1. Syntax Check (구문 검사)
• SQL 문법 검증
• 예: SELECT * FORM employees → 오류 (FORM은 잘못된 키워드)
3B-2. Semantic Check (의미 검사)
• Data Dictionary Cache (SGA) 조회
• 테이블 존재 여부: employees 테이블 있음? ✓
• 컬럼 존재 여부: dept_id 컬럼 있음? ✓
• 권한 검사: 현재 사용자가 SELECT 권한 있음? ✓
3B-3. Optimization (최적화 - 가장 비용이 큰 단계)
Server Process의 Optimizer가 수행:
• 가능한 실행 경로 생성:
경로 1: Full Table Scan
경로 2: Index Range Scan (emp_dept_idx) + Table Access
경로 3: Index Fast Full Scan
• 각 경로의 비용(Cost) 계산:
- 테이블 통계 (행 수, 블록 수) 참조
- 인덱스 통계 (선택도, 클러스터링 팩터) 참조
- I/O 비용, CPU 비용 산정
• 최저 비용 경로 선택:
경로 2 선택 (Cost: 15) < 경로 1 (Cost: 200)
3B-4. Row Source Generation (실행 계획 생성)
• 선택된 경로를 실행 가능한 계획으로 변환
• Row Source Tree 생성
3B-5. Library Cache에 저장
Server Process가 Shared SQL Area 생성하여 Library Cache에 삽입:
Shared SQL Area (새로 생성)

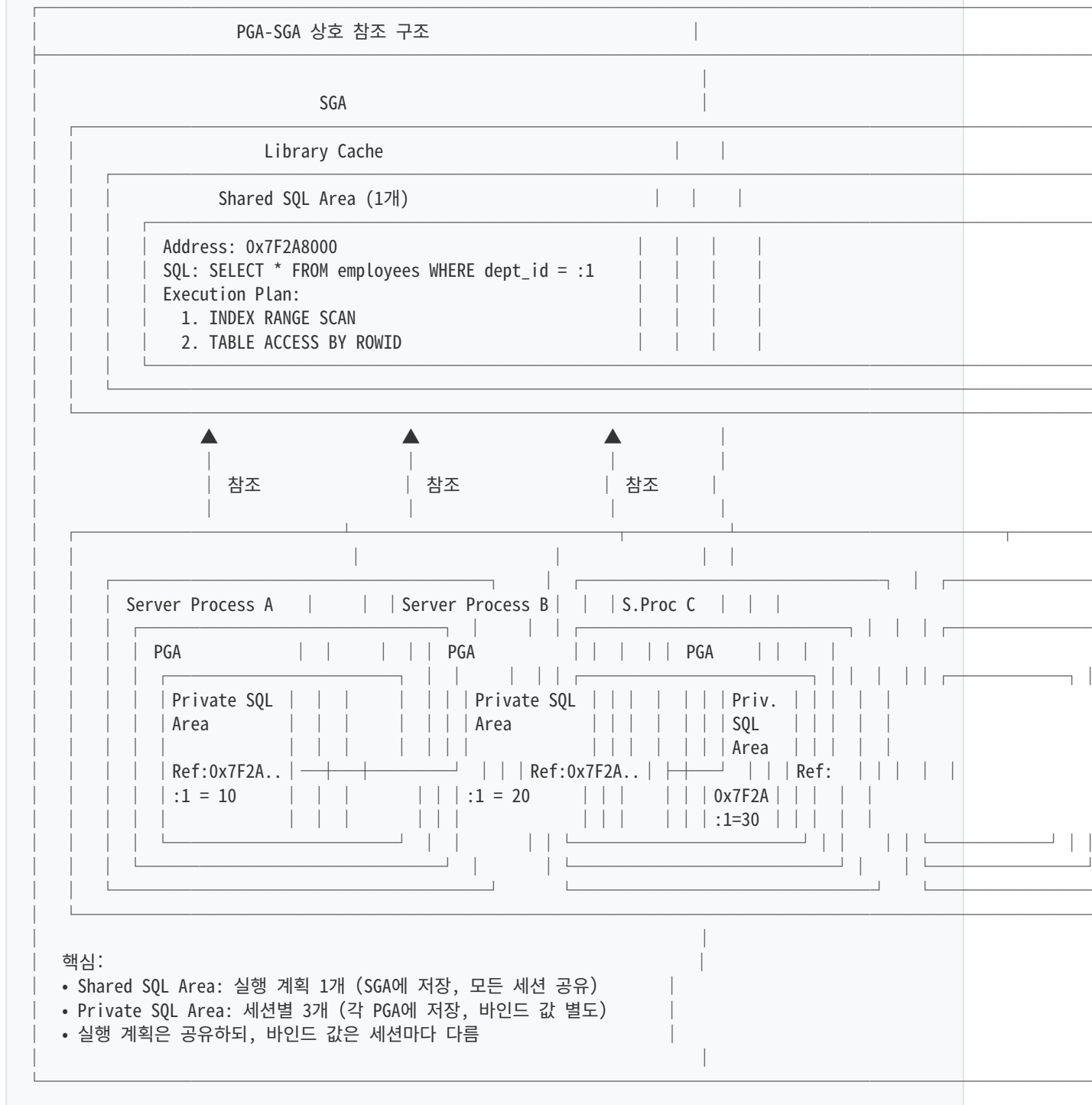

SQL Text: SELECT * FROM employees WHERE dept_id = :1
Hash Value: 0x7A3B2C1D
Parse Tree: [ROOT]—[SELECT]—[FROM employees]—[WHERE ...]
Execution Plan:
1. INDEX RANGE SCAN (emp_dept_idx)
2. TABLE ACCESS BY ROWID (employees)
Reference Count: 1
First Load Time: 2024-12-29 14:30:15
※ 이후 동일 SQL 실행 시 이 Shared SQL Area 재사용
비용: 높음 (Syntax → Semantic → Optimization → Generation)
CPU 사용: 상당함 (특히 Optimization 단계)

12.5 다중 세션에서의 실행 계획 공유


시나리오: 3개 세션이 동일 SQL 실행
SELECT * FROM employees WHERE dept_id = :1
시간
T1: Session A가 최초 실행
Server Process A:
1. Library Cache 검색 → 못 찾음
2. Hard Parse 수행 (Syntax → Semantic → Optimize → Generate)
3. Shared SQL Area 생성하여 Library Cache에 저장
4. Private SQL Area 할당, Shared SQL Area 참조
5. 바인드: :1 = 10
6. Execute
Library Cache 상태:
Shared SQL Area
• SQL: SELECT * FROM employees WHERE dept_id = :1
• Plan: INDEX RANGE SCAN → TABLE ACCESS
• Ref Count: 1 (Session A)
T2: Session B가 동일 SQL 실행
Server Process B:
1. Library Cache 검색 → 찾음!
2. Soft Parse (Hard Parse 생략)
3. Private SQL Area 할당, 기존 Shared SQL Area 참조
4. 바인드: :1 = 20 (다른 값이지만 같은 실행 계획 사용)
5. Execute
Library Cache 상태:
Shared SQL Area (동일)
• SQL: SELECT * FROM employees WHERE dept_id = :1
• Plan: INDEX RANGE SCAN → TABLE ACCESS
• Ref Count: 2 (Session A, Session B)
T3: Session C가 동일 SQL 실행
Server Process C:
1. Library Cache 검색 → 찾음!
2. Soft Parse
3. Private SQL Area 할당, 기존 Shared SQL Area 참조
4. 바인드: :1 = 30
5. Execute
Library Cache 상태:
Shared SQL Area (동일 - 여전히 1개)

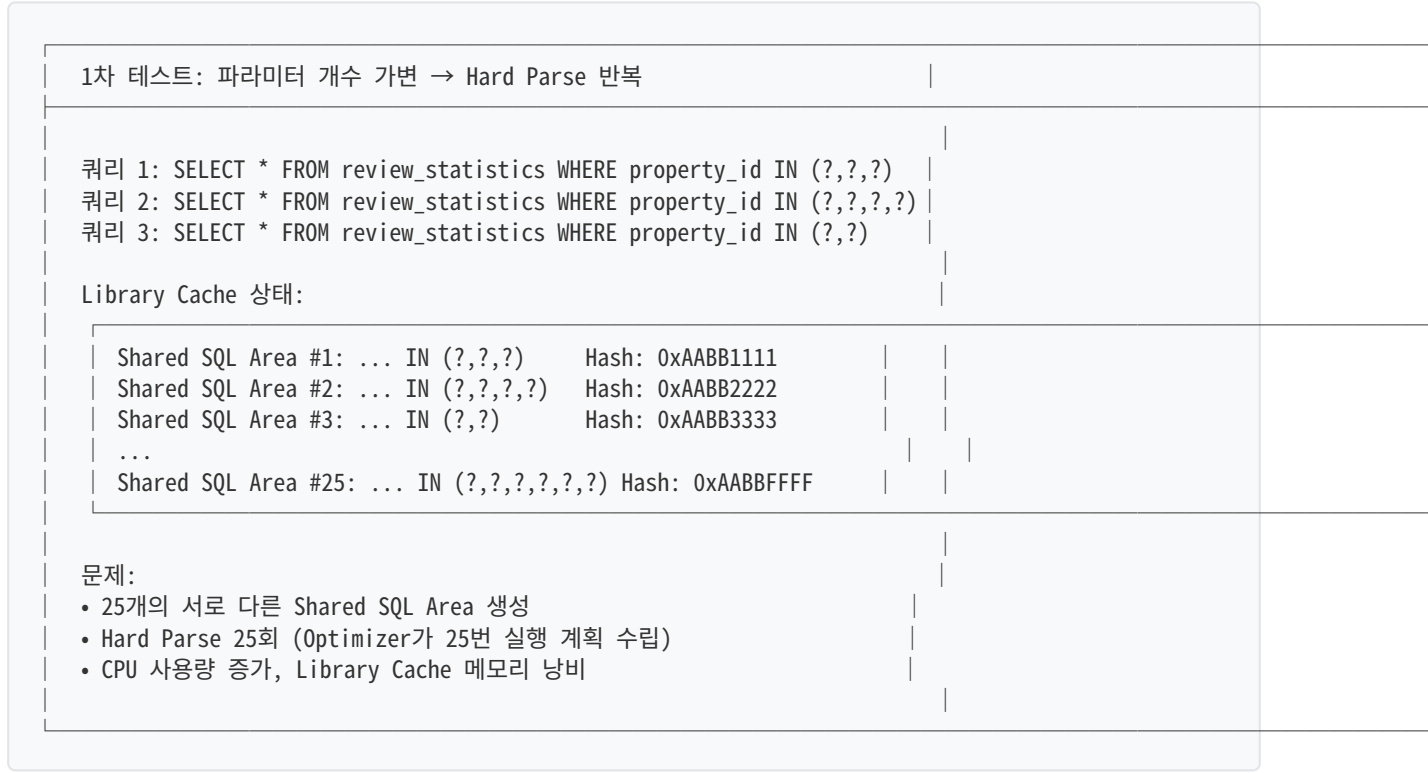


12.6 PGA와 SGA의 상호 참조 구조

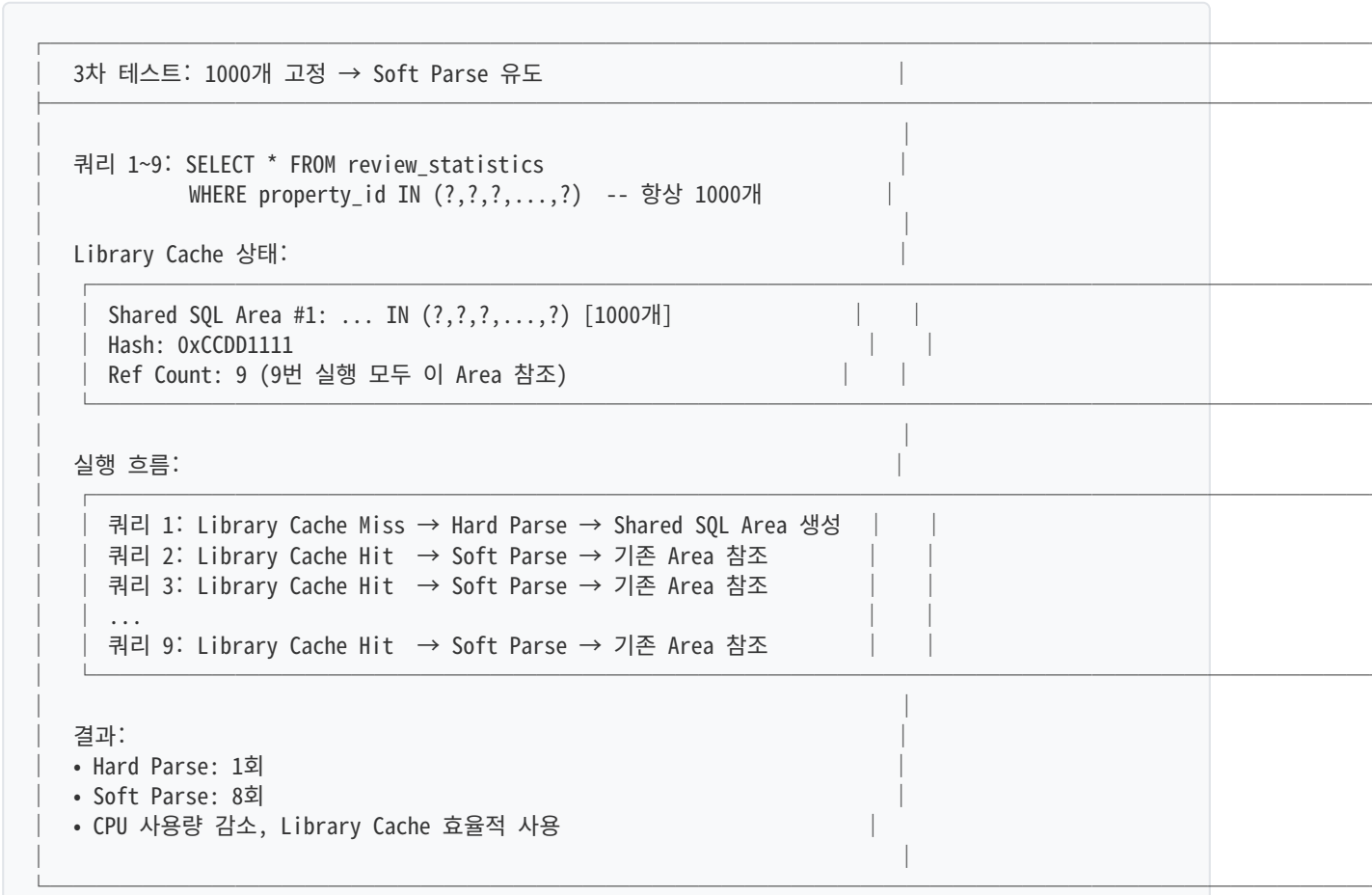


13. 프로젝트 연결: IN절 파라미터 개수와 Library Cache

13.1 1차 테스트: 파라미터 개수 가변 → Hard Parse 반복

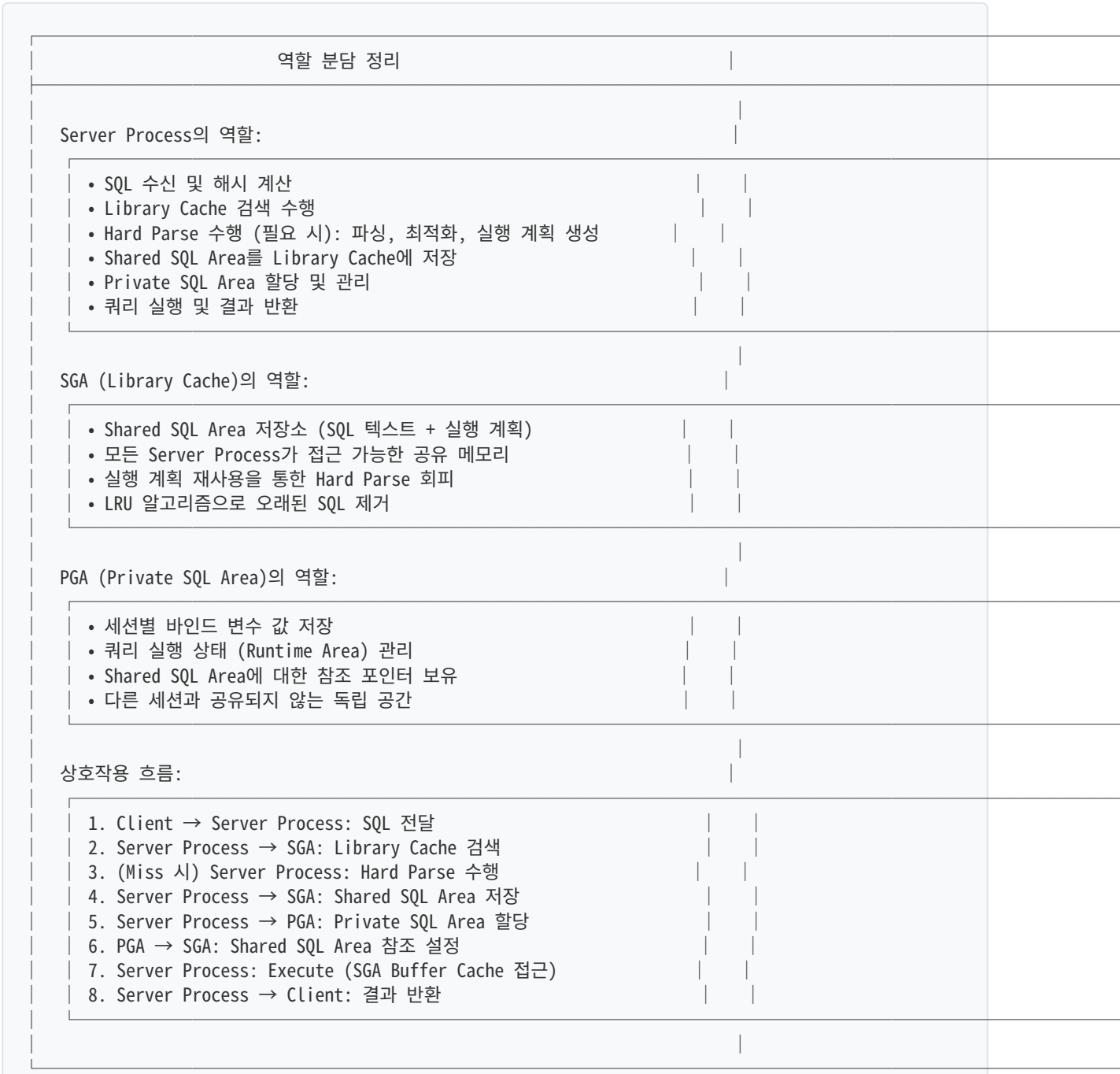


13.2 3차 테스트: 1000개 고정 → Soft Parse 유도



---

14. 정리: Server Process, PGA, SGA 상호작용



15. 면접 대응 포인트

Q: "Connection Pool의 크기를 어떻게 결정하나요?"

A: Dedicated Server 모드에서 Connection 하나당 Server Process 하나가 생성되고, 각 프로세스는 PGA 메모리를 점유합니다. 따라서 Pool 크기를 무작정 늘리면 Oracle 서버의 메모리 사용량이 비례 증가합니다. 제 프로젝트에서는 HikariCP 10개로 설정했는데, N+1 패턴으로 인해 Connection Holding Time이 길어지면서

Waiting이 최대 9건까지 발생했습니다. 이 경우 Pool 크기를 늘리는 것보다 쿼리 패턴을 개선하여 Connection 점유 시간을 줄이는 것이 근본적인 해결책입니다.

Q: "바인드 변수를 사용하면 왜 성능이 좋아지나요?"

A: 바인드 변수를 사용하면 SQL 텍스트가 동일하게 유지되어 Library Cache에서 기존 실행 계획을 재사용할 수 있습니다. 이것을 Soft Parse라고 하는데, Hard Parse에서 수행하는 Syntax Check, Semantic Check, Optimization 과정을 생략하므로 CPU 사용량이 크게 줄어듭니다. 바인드 변수의 실제 값은 각 세션의 PGA 내 Private SQL Area에 저장되므로, 값이 달라도 SGA의 동일한 실행 계획을 공유합니다.

Q: "프로젝트에서 이 개념을 어떻게 적용했나요?"

A: IN 절의 파라미터 개수를 1000개로 고정했습니다. 파라미터 개수가 가변적이면 Oracle이 각각 다른 SQL로 인식하여 매번 Hard Parse가 발생했습니다. 1000개로 고정하니까 SQL 텍스트가 동일해져서 9번 실행 중 첫 번째만 Hard Parse, 나머지 8번은 Soft Parse로 처리되었습니다.

Temp Tablespace 참고

학습 우선순위상 깊이 파고들 필요는 없으나, 개념적으로 알아두면 좋은 수준:

> "Sort Area나 Hash Area 같은 PGA 메모리가 부족하면 Oracle은 Temp Tablespace(디스크)를 사용한다. 디스크 I/O가 발생하므로 메모리에서 처리할 때보다 성능이 저하된다."

항목	프로젝트 연관성	면접 출현 가능성	학습 권장
Temp Tablespace 개념	낮음	낮음	한 문장으로 충분
Temp 사이징/튜닝	없음	거의 없음 (DBA 영역)	생략

Temp Tablespace 관리는 DBA 영역이며, 개발자가 크기를 조정하거나 모니터링할 권한이 일반적으로 없다. Wherehouse 프로젝트의 핵심은 N+1 → Connection 점유 → HikariCP Waiting이므로 직접 연결되지 않는다.