Library Cache Miss(Hard Parse) 비용 상세 분석

Oracle 19c Database Concepts 공식 문서 기반

---

1. Library Cache란 무엇인가

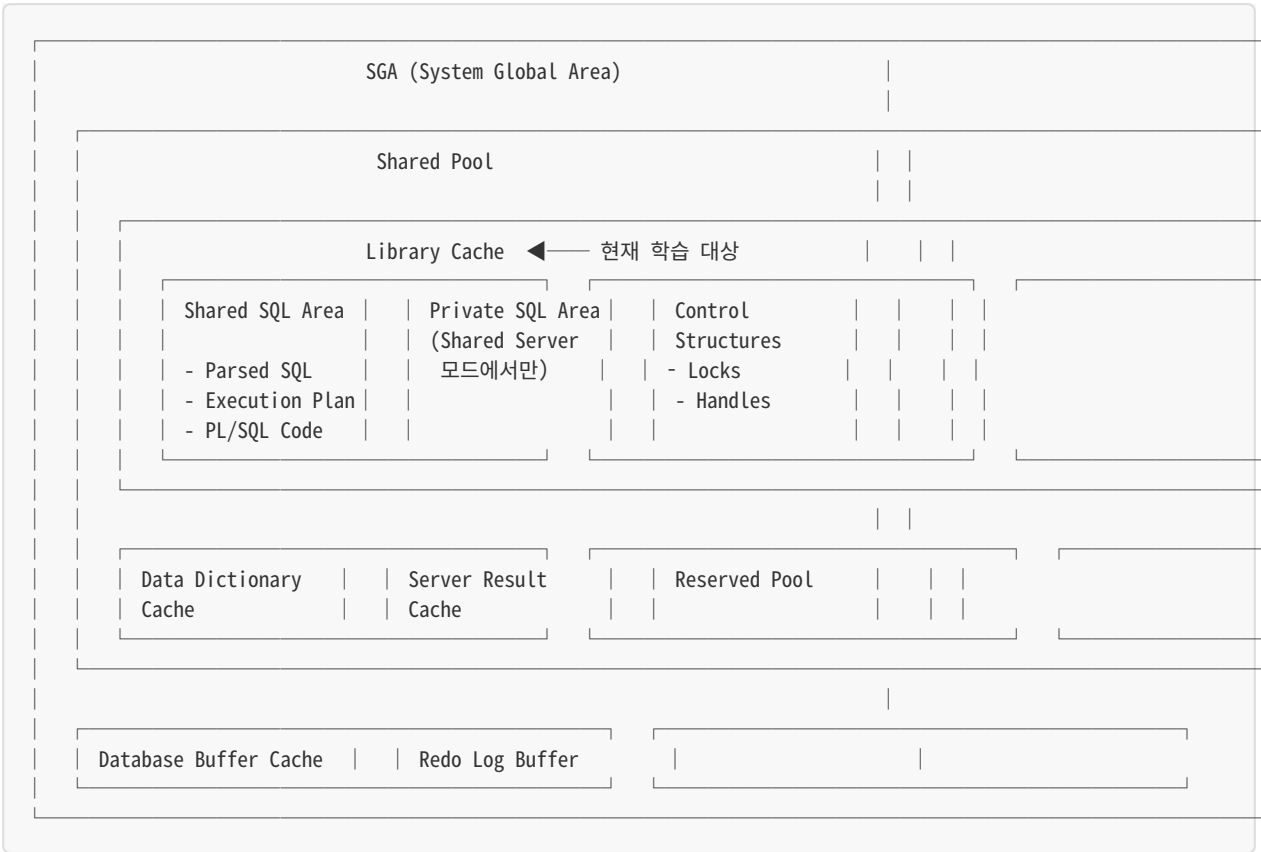
공식 정의

Oracle 공식 문서에 따르면:

> "The library cache is a shared pool memory structure that stores executable SQL and PL/SQL code. This cache contains the shared SQL and PL/SQL areas and control structures such as locks and library cache handles."

Library Cache는 단순히 "SQL을 저장하는 곳"이 아니다. **실행 가능한 형태(executable form)**로 변환된 코드를 보관하는 구조다. 여기서 "실행 가능한 형태"란 파싱이 완료되어 즉시 Execute 단계로 진입할 수 있는 상태를 의미한다.

Oracle 메모리 아키텍처 내 위치



Library Cache의 내부 구성

```
Library Cache
├─── Shared SQL Area
│    ├─── Parsed SQL Statements (파싱된 SQL 텍스트)
│    ├─── SQL Execution Plans (실행 계획)
│    └─── Parsed and Compiled PL/SQL Program Units
│
├─── Private SQL Area (Shared Server 모드에서만 여기에 위치)
│    └─── 일반적으로는 PGA에 위치
│
└─── Control Structures
     ├─── Locks (동시성 제어)
     └─── Library Cache Handles (객체 참조 포인터)
```

Library Cache의 핵심 역할: Soft Parse vs Hard Parse

Oracle 공식 문서에 따르면:

> "When a SQL statement is executed, the database attempts to reuse previously executed code. If a parsed representation of a SQL statement exists in the library cache and can be shared, then the database reuses the code, known as a soft parse or a library cache hit. Otherwise, the database must build a new executable version of the application code, known as a hard parse or a library cache miss."

구분	Library Cache 상태	수행 작업	비용
Soft Parse	Hit (동일 SQL 존재)	기존 실행 계획 재사용	낮음
Hard Parse	Miss (동일 SQL 없음)	Syntax 검사 → Semantic 검사 → Optimizer → 실행 계획 생성	**리소스 집약적** (CPU, Latch, 메모리)

"동일한 SQL"의 판정 기준

Oracle이 Library Cache에서 SQL을 찾을 때 **정확히 일치(exact match)**해야 한다. 다음은 모두 **다른 SQL**로 인식된다:

```
-- 이 네 개는 모두 별개의 SQL로 Library Cache에 저장됨
SELECT * FROM users WHERE id = 1;
SELECT * FROM users WHERE id = 2;
SELECT * FROM USERS WHERE id = 1;  -- 대소문자 다름
select * from users where id = 1;  -- 키워드 대소문자 다름
```

Oracle 공식 문서에 따르면:

> "This type of problem is commonly caused when similar SQL statements are written which differ in space, case, or some combination of the two. You may also consider using bind variables rather than explicitly specified constants in your statements whenever possible."

Library Cache Miss가 Buffer Cache Miss보다 비싼 이유

Oracle 공식 문서에 따르면:

> "A cache miss on the data dictionary cache or library cache is more expensive than a miss on the buffer cache."

Buffer Cache Miss는 단순히 디스크에서 데이터 블록을 읽어오는 I/O 작업이다. 반면 Library Cache Miss(Hard Parse)는 다음의 **CPU 집약적 작업**을 모두 수행해야 한다:

단계	Hard Parse 시 수행 작업
1	SQL 텍스트 해싱 및 검색
2	Syntax 검사
3	Semantic 검사 (Data Dictionary Cache 접근)
4	권한 검사
5	Optimizer 호출 → 실행 계획 생성
6	Library Cache에 새 항목 할당 (Latch 경합 가능)

백엔드 개발자가 취해야 할 액션

Bind Variable 사용 강제화

```
// Bad - 매번 Hard Parse 발생
String sql = "SELECT * FROM users WHERE id = " + userId;

// Good - Soft Parse 가능
String sql = "SELECT * FROM users WHERE id = ?";
PreparedStatement ps = conn.prepareStatement(sql);
ps.setLong(1, userId);
```

JPA/Hibernate에서의 함의

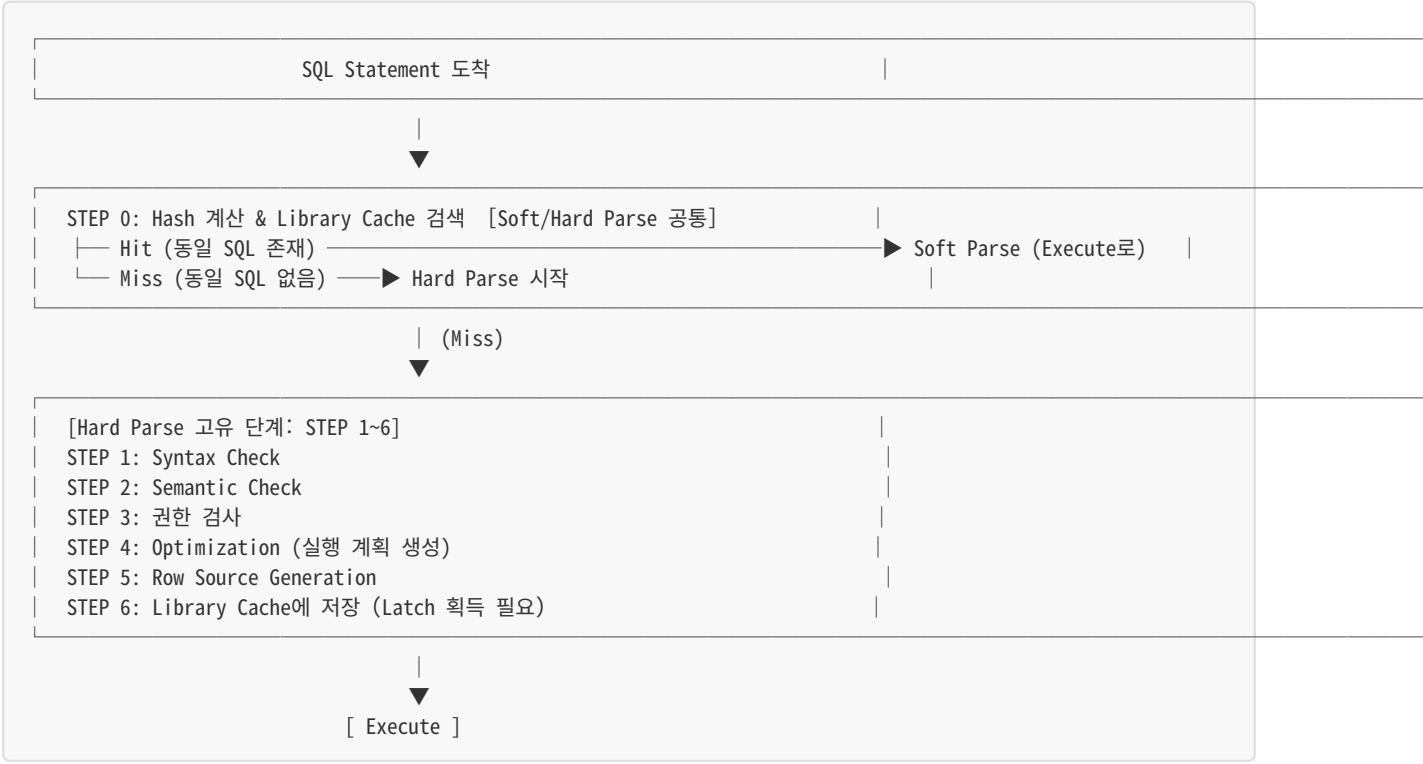
- `@Query`에서 파라미터 바인딩 사용: `:paramName` 또는 `?1`
- IN 절 파라미터 개수 고정 또는 Chunk 방식으로 제어

---

2. Hard Parse 비용 상세 분석: 전체 흐름

Oracle 공식 문서에 따르면:

> "SQL processing is the parsing, optimization, row source generation, and execution of a SQL statement."



---

3. STEP 0: SQL 텍스트 해싱 및 Library Cache 검색

왜 필요한가?

SQL 문장이 Server Process에 도착하면, Oracle은 먼저 "이 SQL을 이전에 처리한 적이 있는가?"를 확인해야 한다. 전체 SQL 텍스트를 문자열 비교하면 O(n) 시간이 걸리므로, 해시 값을 먼저 계산하여 O(1)에 가깝게 검색한다.

무엇을 수행하는가?

```
1. SQL 텍스트 전체를 해시 함수에 입력 → 고정 길이 해시 값 생성
2. Library Cache의 해시 버킷에서 동일 해시 값을 가진 항목 검색
3. 해시 충돌 가능성이 있으므로, 해시 값이 같으면 실제 SQL 텍스트도 비교
```

비용 발생 지점

작업	비용 유형	설명
해시 계산	CPU	SQL 텍스트 길이에 비례
버킷 검색	CPU + 메모리 접근	Library Cache Latch 획득 필요
텍스트 비교	CPU	해시 충돌 시 전체 문자열 비교

Wherehouse 연결

IN 절에 8,660개 파라미터가 들어간 SQL은 텍스트 자체가 매우 길다. 해시 계산 시간도 증가하고, 파라미터 개수가 달라질 때마다 다른 해시 값이 생성되어 Library Cache Miss가 발생한다.

---

4. STEP 1: Syntax Check (구문 검사)

왜 필요한가?

Oracle 공식 문서에 따르면:

> "Oracle Database must check each SQL statement for syntactic validity. A statement that breaks a rule for well-formed SQL syntax fails the check."

SQL이 문법적으로 올바른지 확인하지 않고 다음 단계로 진행하면, Semantic Check나 Optimizer에서 예측 불가능한 오류가 발생한다. 이 단계는 SQL 텍스트를 **파싱 트리(Parse Tree)**로 변환하는 과정이다.

무엇을 수행하는가?

```
-- 이 SQL이 도착했다고 가정
SELECT * FROM employees WHERE department_id = 10;
```

1. **Lexical Analysis (어휘 분석)**: SQL 문자열을 토큰으로 분리 `[SELECT] [*] [FROM] [employees] [WHERE] [department_id] [=] [10] [;]`
2. **Syntax Analysis (구문 분석)**: 토큰 시퀀스가 SQL 문법 규칙을 따르는지 검증 `SELECT_STMT ::= SELECT select_list FROM table_ref [WHERE condition]`
3. **Parse Tree 생성**: 문법 구조를 트리 형태로 표현 `SELECT_STMT / | \ SELECT FROM WHERE | | | * employees CONDITION / | \ department_id = 10`

비용 발생 지점

작업	비용 유형	설명
Lexer	CPU	SQL 길이에 비례하여 토큰화
Parser	CPU	문법 규칙 매칭, 트리 구조 생성
메모리 할당	메모리	Parse Tree를 위한 임시 메모리

실패 예시

Oracle 공식 문서에 따르면:

> "For example, the following statement fails because the keyword FROM is misspelled as FORM: SQL> SELECT * FORM employees; ERROR at line 1: ORA-00923: FROM keyword not found where expected"

---

5. STEP 2: Semantic Check (의미 검사)

왜 필요한가?

Oracle 공식 문서에 따르면:

> "The semantics of a statement are its meaning. A semantic check determines whether a statement is meaningful, for example, whether the objects and columns in the statement exist."

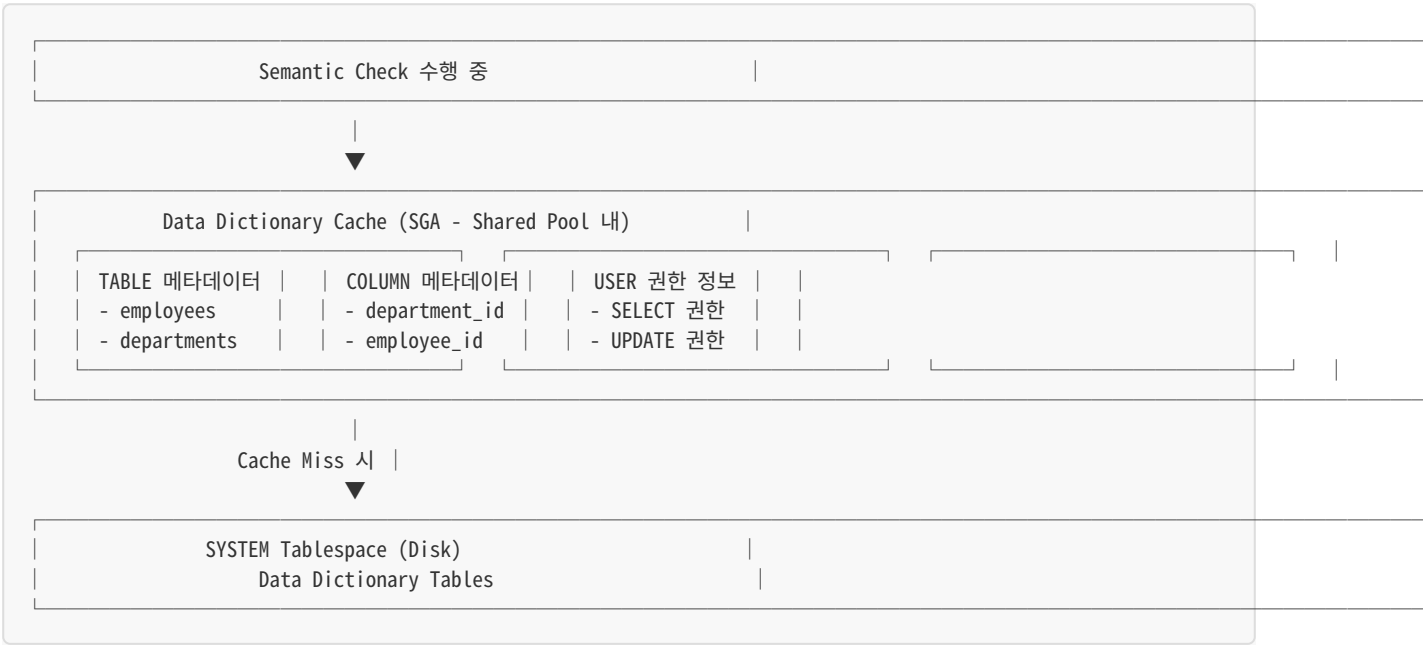
Syntax Check를 통과한 SQL이 "문법적으로는 맞지만 의미적으로 틀린" 경우를 걸러낸다.

무엇을 수행하는가?

```
1. 테이블 존재 여부 확인: employees 테이블이 실제로 존재하는가?
2. 컬럼 존재 여부 확인: department_id 컬럼이 employees 테이블에 있는가?
3. 데이터 타입 호환성: department_id (NUMBER)와 10 (NUMBER)이 비교 가능한가?
4. 객체 참조 해결: employees가 테이블인지, 뷰인지, 시노님인지 확인
```

Data Dictionary Cache 접근

이 단계에서 **Data Dictionary Cache**에 대한 접근이 발생한다. 테이블/컬럼 정보는 디스크의 Data Dictionary에 저장되어 있으며, 성능을 위해 SGA의 Data Dictionary Cache에 캐싱된다.



비용 발생 지점

작업	비용 유형	설명
Dictionary Cache 조회	CPU + Latch	캐시 검색 및 동시성 제어
Dictionary Cache Miss	**Disk I/O**	디스크에서 메타데이터 읽기 (매우 비쌈)
객체 해석	CPU	시노님 → 테이블 → 스키마 체인 해석

실패 예시

Oracle 공식 문서에 따르면:

> "A syntactically correct statement can fail a semantic check, as shown in the following example of a query of a nonexistent table: SQL> SELECT * FROM nonexistent_table;"

동일 SQL 텍스트도 다른 의미를 가질 수 있다

Oracle 공식 문서에 따르면:

> "For example, suppose two different users log in to the database and issue the following SQL statements: CREATE TABLE my_table (some_col INTEGER); SELECT * FROM my_table; The SELECT statements for the two users are syntactically identical, but two separate schema objects are named my_table."

이것이 Library Cache에서 SQL을 찾을 때 **해시 값 + SQL 텍스트** 일치만으로는 부족하고, **Semantic 동일성**도 확인해야 하는 이유다.

---

6. STEP 3: 권한 검사 (Security Check)

왜 필요한가?

Oracle 공식 문서에 따르면:

> "checking privileges to access referenced schema objects"

SQL이 참조하는 객체에 대해 현재 사용자가 적절한 권한을 가지고 있는지 확인해야 한다. 권한 없이 실행을 허용하면 보안 위반이다.

무엇을 수행하는가?

```
1. 현재 세션의 사용자 식별
2. 해당 사용자의 시스템 권한 확인 (예: SELECT ANY TABLE)
3. 객체 권한 확인 (예: SELECT ON employees)
4. 역할(Role)을 통한 간접 권한 확인
5. VPD(Virtual Private Database) 정책 적용 여부 확인
```

비용 발생 지점

작업	비용 유형	설명
권한 테이블 조회	CPU + Dictionary Cache	사용자-권한 매핑 확인
역할 해석	CPU	중첩된 역할 구조 해석
VPD 정책 평가	CPU	정책 함수 실행 (있는 경우)

---

7. STEP 4: Optimization (실행 계획 생성)

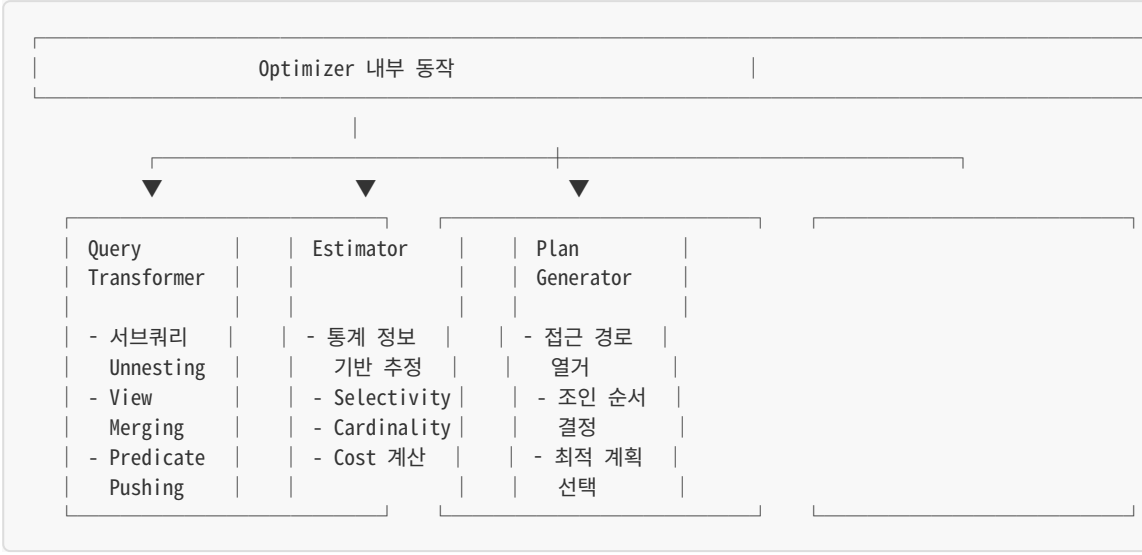
왜 필요한가?

Oracle 공식 문서에 따르면:

> "During the optimization stage, Oracle Database must perform a hard parse at least once for every unique DML statement and performs the optimization during this parse."

동일한 결과를 반환하는 SQL도 실행 방법에 따라 성능이 천차만별이다. Optimizer는 가능한 모든 실행 경로를 평가하여 **비용(Cost)**이 가장 낮은 계획을 선택한다.

무엇을 수행하는가?



1. Query Transformer

SQL을 의미적으로 동등하지만 더 효율적인 형태로 변환한다.

```
-- 원본
SELECT * FROM employees WHERE department_id IN
    (SELECT department_id FROM departments WHERE location_id = 1800);

-- 변환 후 (Subquery Unnesting)
SELECT e.* FROM employees e, departments d
WHERE e.department_id = d.department_id AND d.location_id = 1800;
```

2. Estimator

각 실행 경로의 비용을 추정한다.

- **Selectivity**: 조건을 만족하는 행의 비율 (예: department_id = 10 → 전체의 5%)
- **Cardinality**: 각 단계에서 반환되는 행 수
- **Cost**: CPU 사용량 + I/O 횟수의 가중 합

3. Plan Generator

가능한 실행 계획을 열거하고 최적의 계획을 선택한다.

```
가능한 실행 계획들:
Plan A: Full Table Scan (Cost: 150)
Plan B: Index Range Scan + Table Access by ROWID (Cost: 25)  ← 선택됨
Plan C: Index Full Scan (Cost: 200)
```

비용 발생 지점 (Hard Parse의 핵심 병목)

작업	비용 유형	설명
통계 정보 조회	CPU + Dictionary Cache	테이블 통계, 인덱스 통계, 히스토그램
비용 계산	**CPU 집약적**	모든 가능한 조합에 대해 수학적 계산
메모리	메모리	후보 계획들을 메모리에 유지

Oracle 공식 문서에 따르면:

> "The optimizer determines the most efficient way to execute a SQL statement. This is an important step in the processing of any data manipulation language (DML) statement: SELECT, INSERT, UPDATE, or DELETE. There are often many different ways to execute a SQL statement; for example, by varying the order in which tables or indexes are accessed."

Optimizer 환경이 다르면 같은 SQL도 다른 계획

Oracle 공식 문서에 따르면:

> "Even if two statements are semantically identical, an environmental difference can force a hard parse. In this context, the optimizer environment is the totality of session settings that can affect execution plan generation, such as the work area size or optimizer settings (for example, the optimizer mode)."

---

8. STEP 5: Row Source Generation

왜 필요한가?

Optimizer가 선택한 실행 계획은 아직 "논리적 계획"이다. 이것을 실제로 실행 가능한 **이진 프로그램(Binary Program)** 형태로 변환해야 한다.

무엇을 수행하는가?

Oracle 공식 문서에 따르면:

> "The row source generator is software that receives the optimal execution plan from the optimizer and produces an iterative execution plan that is usable by the rest of the database. The iterative plan is a binary program that, when executed by the SQL engine, produces the result set."

```
Optimizer 출력 (논리적 계획):
┌─────────────────────────────────┐
│ SELECT STATEMENT │
│    └── NESTED LOOPS
│        ├── INDEX RANGE SCAN (emp_dept_ix)
│        └── TABLE ACCESS BY ROWID (employees)
└─────────────────────────────────┘

Row Source Generator 출력 (실행 가능한 형태):
┌──────────────────────────────────────────────────────────────
│  Row Source Tree (이진 실행 구조)                         │
│                                                           │
│  rowSource[0] = new SelectStatement()                     │
│  rowSource[1] = new NestedLoopJoin(rowSource[2], rowSource[3])  │
│  rowSource[2] = new IndexRangeScan("emp_dept_ix")         │
│  rowSource[3] = new TableAccessByRowid("employees")       │
│                                                           │
│  각 rowSource는 next() 메서드를 가진 Iterator 패턴           │
└──────────────────────────────────────────────────────────────
```

Oracle 공식 문서에 따르면:

> "The row source generator receives the optimal plan from the optimizer. It outputs the execution plan for the SQL statement. The execution plan is a collection of row sources structured in the form of a tree. A row source is an iterative control structure. It processes a set of rows, one row"

비용 발생 지점

작업	비용 유형	설명
Row Source 생성	CPU + 메모리	각 연산자별 실행 구조 생성
트리 구조 연결	CPU	부모-자식 관계 설정

---

9. STEP 6: Library Cache에 저장 (Latch 경합)

왜 필요한가?

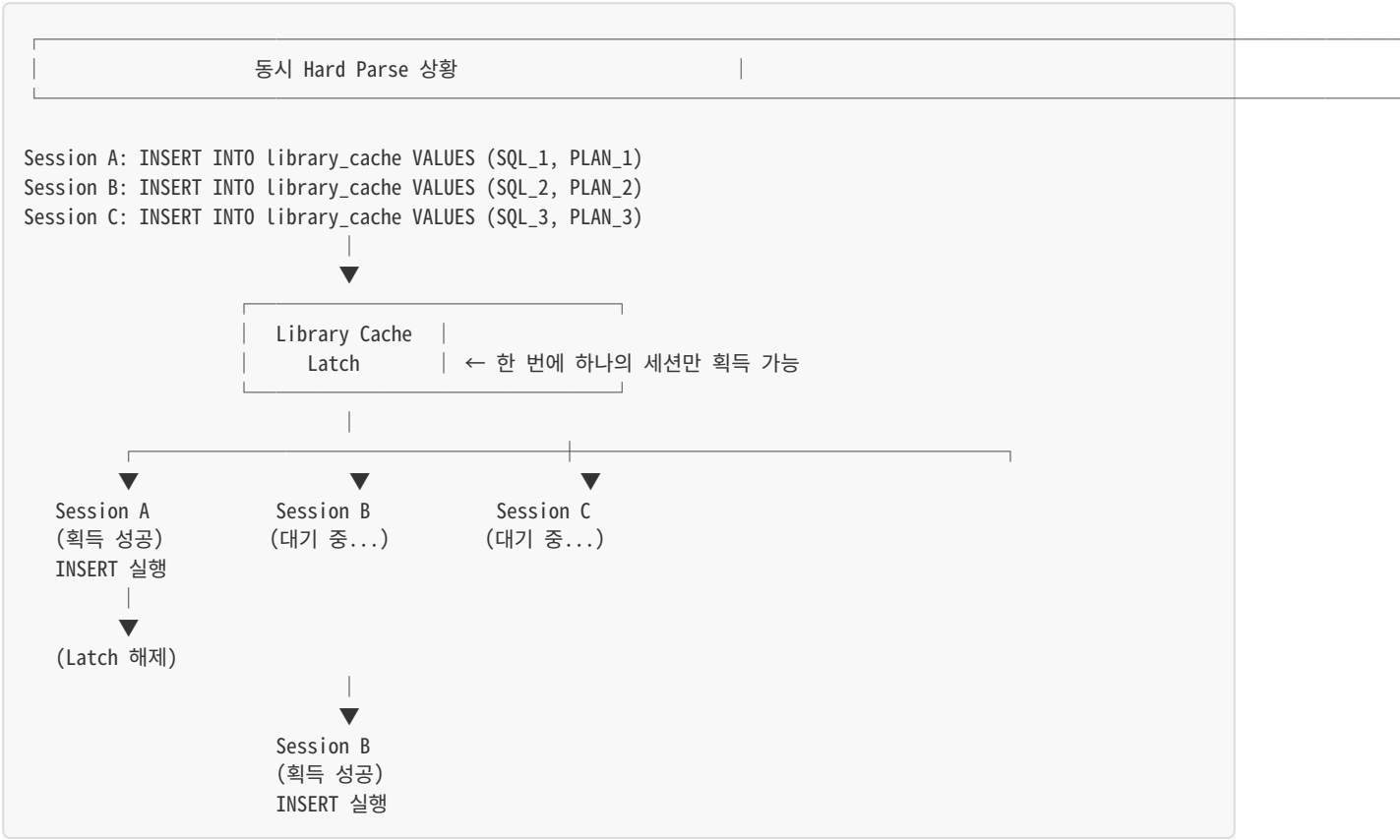
Hard Parse로 생성된 실행 계획을 Library Cache에 저장해야 다음에 동일한 SQL이 들어왔을 때 **Soft Parse**로 재사용할 수 있다.

무엇을 수행하는가?

```
1. Library Cache에서 빈 공간 확보 (필요시 LRU로 기존 항목 제거)
2. Shared SQL Area 할당
3. 파싱된 SQL 텍스트, 실행 계획, 의존성 정보 저장
4. Hash Bucket에 새 항목 연결
```

Latch 경합이 발생하는 이유

Library Cache는 **모든 세션이 공유**하는 메모리 영역이다. 동시에 여러 세션이 Library Cache를 수정하려 하면 데이터 손상이 발생할 수 있다. 이를 방지하기 위해 **Latch**(저수준 잠금 메커니즘)를 사용한다.



비용 발생 지점

작업	비용 유형	설명
Latch 획득 대기	**CPU Spin + Context Switch**	경합 시 CPU를 소모하며 대기
메모리 할당	CPU + 메모리	Shared Pool에서 청크 할당
Hash Bucket 연결	CPU	링크드 리스트 조작

---

10. 전체 비용 요약: Soft Parse vs Hard Parse

단계	Soft Parse	Hard Parse
Hash 계산 & 검색	✓ 수행	✓ 수행
Syntax Check	캐시된 Parse Tree 재사용 (재파싱 없음)	✓ 수행
Semantic Check	캐시된 결과 재사용 (객체 유효성만 검증)	✓ 수행 (Dictionary Cache 접근)
권한 검사	경량화된 검증 (캐시된 권한 정보 유효성 확인)	✓ 수행 (전체 권한 트리 해석)
Optimization	**건너뜀**	✓ 수행 (**CPU 집약적**)
Row Source Gen	**건너뜀**	✓ 수행
Library Cache 저장	**건너뜀**	✓ 수행 (Latch 경합)

Oracle 공식 문서에 따르면:

> "In general, a soft parse is preferable to a hard parse because the database skips the optimization and row source generation steps, proceeding straight to execution."

공식 문서는 Soft Parse가 건너뛰는 단계로 **optimization과 row source generation만** 명시한다. Syntax/Semantic Check와 권한 검사의 경우, Soft Parse에서는 이미 캐싱된 Shared SQL Area를 재사용하므로 **전체 재수행은 하지 않지만**, 다음의 경량화된 검증은 수행된다:

- **객체 유효성 검증**: DDL(ALTER, DROP 등)로 인해 커서가 invalidate되었는지 확인
- **권한 유효성 검증**: REVOKE 등으로 권한이 변경되어 캐시된 권한 정보가 무효화되었는지 확인

이러한 검증을 통해, Library Cache에 캐싱된 이후 객체나 권한이 변경된 경우에도 보안과 정합성이 유지된다.

---

11. Wherehouse 프로젝트에 대한 함의

1차 테스트 (N+1, 25회 실행)

```
첫 번째: Hard Parse (6단계 모두 수행)
2~25번째: Soft Parse (Step 0만 수행) ← Bind Variable 덕분
```

→ Hard Parse 1회 + Soft Parse 24회 = **상대적으로 낮은 파싱 오버헤드**

2차 테스트 (Bulk, IN 절 8,660개)

```
매 요청마다 파라미터 개수가 다르면:
요청 1: WHERE property_id IN (?, ?, ... 8660개) → Hard Parse
요청 2: WHERE property_id IN (?, ?, ... 8500개) → Hard Parse (다른 SQL로 인식)
요청 3: WHERE property_id IN (?, ?, ... 9000개) → Hard Parse (다른 SQL로 인식)
```

→ 매번 Hard Parse = **Optimization 비용 + Latch 경합 누적**

이것이 Bulk Fetch가 오히려 느려진 원인 중 하나다. RDB 조회 시간 증가(343ms → 1,748ms)의 상당 부분이 다음 요인들에 기인한다:

- **파라미터 개수 변동**으로 인한 매 요청마다 Hard Parse 발생
- **8,660개 파라미터**로 인한 Parse Tree 크기 증가와 Optimizer 처리 부담
- **Hard Parse 누적**으로 인한 Library Cache Latch 경합