Oracle 실행 계획 분석 학습 문서

1. INLIST ITERATOR 개요

1.1 실행 계획 구조 예시

```
SELECT * FROM emp WHERE empno IN (7876, 7900, 7902);
```

```
OPERATION          OPTIONS           OBJECT_NAME
----------------   ---------------   --------------
SELECT STATEMENT
INLIST ITERATOR
TABLE ACCESS       BY ROWID          EMP
INDEX              UNIQUE SCAN       EMP_EMPNO
```

1.2 실행 순서

실행 계획은 들여쓰기가 깊은 Operation부터 실행된다. 이 계획의 실행 순서는 다음과 같다:

1. **INDEX RANGE SCAN (EMP_EMPNO)**: empno 인덱스에서 7876에 해당하는 ROWID를 찾는다
2. **TABLE ACCESS BY ROWID (EMP)**: 찾은 ROWID로 EMP 테이블의 실제 행에 접근한다
3. **INLIST ITERATOR**: 1-2번 과정을 IN 절의 다음 값(7900)에 대해 반복한다
4. 7902에 대해서도 동일하게 반복한다
5. **SELECT STATEMENT**: 모든 결과를 반환한다

1.3 INLIST ITERATOR의 의미

문서의 설명:

> "The INLIST ITERATOR operation iterates over the next operation in the plan for each value in the IN-list predicate."

INLIST ITERATOR는 IN 절의 각 값에 대해 하위 Operation을 반복 실행하는 제어 구조다. 핵심은 인덱스를 활용한다는 점이다. IN 절의 값이 3개면 INDEX RANGE SCAN을 3번 수행하고, 각각에서 얻은 ROWID로 테이블에 접근한다.

1.4 Wherehouse 프로젝트 맥락에서의 의미

REVIEW_STATISTICS 테이블에 295개 PROPERTY_ID로 IN 절 쿼리를 실행할 때, 실행 계획에 INLIST ITERATOR가 나타나면 Oracle이 PK 인덱스(PK_REVIEW_STATISTICS)를 효율적으로 사용하고 있다는 증거다.

반대로 CONCATENATION이 나타나면 IN 절이 OR 조건으로 분해되어 비효율적으로 처리되고 있다는 의미다. 이것이 Bulk Fetch에서 발생했을 가능성이 있고, 이를 실행 계획으로 증명하는 것이 피드백에서 요구한 "DBMS 병목 판단 근거"가 된다.

---

2. INLIST ITERATOR 심층 분석

2.1 정의

INLIST ITERATOR는 Oracle Optimizer가 IN 절(IN-list predicate)을 처리할 때 생성하는 실행 계획 Operation이다. 이 Operation은 IN 절에 나열된 각 값에 대해 하위 Operation(주로 인덱스 스캔)을 반복 실행하는 제어 구조 역할을 한다.

2.2 왜 INLIST ITERATOR가 필요한가

IN 절의 본질적 특성

IN 절은 논리적으로 여러 개의 OR 조건과 동치다.

```
WHERE empno IN (7876, 7900, 7902)
```

이것은 논리적으로 다음과 같다:

```
WHERE empno = 7876 OR empno = 7900 OR empno = 7902
```

문제: OR 조건의 비효율성

OR 조건을 그대로 처리하면 Oracle은 두 가지 선택지가 있다.

첫 번째 선택지는 **TABLE ACCESS FULL**이다. 테이블 전체를 스캔하면서 각 행이 OR 조건 중 하나라도 만족하는지 검사한다. 52,020행 테이블에서 3개 행만 필요한데 전체를 읽는 것은 비효율적이다.

두 번째 선택지는 **CONCATENATION**이다. 각 OR 조건을 별도 쿼리로 분리하여 실행한 후 결과를 합친다. 이 경우 중복 제거 오버헤드가 발생하고, 실행 계획이 복잡해진다.

해결책: INLIST ITERATOR

INLIST ITERATOR는 세 번째 선택지다. IN 절의 각 값을 순회하면서 **동일한 인덱스 접근 경로를 반복 사용**한다. OR로 분해하지 않고, Full Scan도 하지 않으면서 인덱스의 효율성을 유지한다.

2.3 INLIST ITERATOR의 동작 메커니즘

실행 계획 구조

```
SELECT STATEMENT
  INLIST ITERATOR                ← 제어 구조 (반복자)
    TABLE ACCESS BY ROWID        ← 테이블 접근
      INDEX RANGE SCAN           ← 인덱스 스캔
```

단계별 동작

IN 절에 3개 값(7876, 7900, 7902)이 있을 때:

```
[INLIST ITERATOR 시작]
│
├── 반복 1: 값 = 7876
│   ├── INDEX RANGE SCAN: EMP_EMPNO 인덱스에서 7876의 ROWID 탐색
│   └── TABLE ACCESS BY ROWID: 해당 ROWID로 EMP 테이블 행 접근
│
├── 반복 2: 값 = 7900
│   ├── INDEX RANGE SCAN: EMP_EMPNO 인덱스에서 7900의 ROWID 탐색
│   └── TABLE ACCESS BY ROWID: 해당 ROWID로 EMP 테이블 행 접근
│
├── 반복 3: 값 = 7902
│   ├── INDEX RANGE SCAN: EMP_EMPNO 인덱스에서 7902의 ROWID 탐색
│   └── TABLE ACCESS BY ROWID: 해당 ROWID로 EMP 테이블 행 접근
│
[INLIST ITERATOR 종료 - 결과 반환]
```

핵심 특징

INLIST ITERATOR는 하위 실행 계획을 재컴파일하지 않고 재사용한다. INDEX RANGE SCAN → TABLE ACCESS BY ROWID 경로가 한 번 수립되면, IN 절의 값만 바꿔가며 동일한 경로를 반복 실행한다. 이것이 CONCATENATION과의 결정적 차이다.

2.4 INLIST ITERATOR가 선택되는 조건

필수 조건

첫째, IN 절 컬럼에 인덱스가 존재해야 한다. 인덱스가 없으면 INLIST ITERATOR가 생성될 수 없다. TABLE ACCESS FULL이 선택된다.

둘째, Optimizer가 인덱스 사용이 효율적이라고 판단해야 한다. IN 절의 값 개수가 너무 많거나, 테이블이 작아서 Full Scan이 더 효율적이면 INLIST ITERATOR가 선택되지 않을 수 있다.

구체적 시나리오

시나리오	실행 계획	이유
IN 절 컬럼에 인덱스 있음 + 소수의 값	INLIST ITERATOR	인덱스 반복 접근이 효율적
IN 절 컬럼에 인덱스 없음	TABLE ACCESS FULL	인덱스 접근 불가
IN 절 값 개수가 매우 많음 (1000개 초과)	CONCATENATION 또는 변형	Oracle IN 절 제한으로 OR 분해 발생
테이블 행 수가 매우 적음	TABLE ACCESS FULL	Full Scan이 오히려 효율적

2.5 CONCATENATION과의 비교

CONCATENATION이 발생하는 경우

Oracle은 IN 절에 1000개 초과의 값이 포함되면 이를 여러 개의 IN 절로 분해하고, 각각을 OR로 연결한다. 이때 실행 계획에 CONCATENATION이 나타난다.

```
-- 원본 쿼리 (1500개 값)
WHERE id IN (val1, val2, ..., val1500)

-- Oracle 내부 변환
WHERE id IN (val1, ..., val1000)
   OR id IN (val1001, ..., val1500)
```

실행 계획 비교

INLIST ITERATOR (효율적)

```
SELECT STATEMENT
  INLIST ITERATOR
    TABLE ACCESS BY INDEX ROWID
      INDEX RANGE SCAN
```

CONCATENATION (비효율적)

```
SELECT STATEMENT
  CONCATENATION
    TABLE ACCESS BY INDEX ROWID
      INDEX RANGE SCAN        ← 첫 번째 IN 절
    TABLE ACCESS BY INDEX ROWID
      INDEX RANGE SCAN        ← 두 번째 IN 절
```

비효율성의 원인

CONCATENATION은 각 분기(branch)가 독립적으로 실행된다. 동일한 인덱스를 여러 번 접근하지만, 각 접근이 별도의 Operation으로 처리되어 오버헤드가 증가한다. 또한 결과 병합 과정에서 중복 제거가 필요할 수 있다.

2.6 Wherehouse 프로젝트에서의 적용

현재 테스트 시나리오

295개 PROPERTY_ID로 REVIEW_STATISTICS 테이블을 조회한다. 295개는 1000개 미만이므로 Oracle의 IN 절 제한에 걸리지 않는다.

예상 실행 계획

```
SELECT PROPERTY_ID, AVG_RATING, LAST_CALCED,
       NEGATIVE_KEYWORD_COUNT, POSITIVE_KEYWORD_COUNT, REVIEW_COUNT
FROM REVIEW_STATISTICS
WHERE PROPERTY_ID IN ('e03d65e70b49ab5bb17258be672024c2', ... /* 295개 */)
```

PK_REVIEW_STATISTICS 인덱스가 존재하고, 295개는 적정 범위이므로 **INLIST ITERATOR**가 나타나야 한다.

```
SELECT STATEMENT
  INLIST ITERATOR
    TABLE ACCESS BY INDEX ROWID  REVIEW_STATISTICS
      INDEX RANGE SCAN           PK_REVIEW_STATISTICS
```

포트폴리오 증명 포인트

Chunk 방식(1000개 단위 분할)에서는 각 쿼리가 1000개 이하의 IN 절을 가지므로 모든 쿼리에서 INLIST ITERATOR가 나타난다.

Bulk Fetch 방식(8660개 한 번에 조회)에서는 1000개 제한을 초과하므로 Hibernate가 OR로 분해하고, Oracle 실행 계획에 CONCATENATION이 나타날 가능성이 있다. 이것이 Bulk Fetch가 오히려 느렸던 원인의 DBMS 레벨 증거가 된다.

실제로 두 방식의 실행 계획을 캡처하여 INLIST ITERATOR vs CONCATENATION을 비교하면, 피드백에서 요구한 "왜 그것이 병목이었는지"에 대한 정량적 근거가 된다.

---

3. IN 절과 OR 조건의 동치성

3.1 IN 절의 정의

IN 절은 특정 컬럼의 값이 주어진 값 목록 중 하나와 일치하는지 검사하는 조건이다.

```
SELECT * FROM emp WHERE empno IN (7876, 7900, 7902);
```

이 쿼리의 의미는 "empno가 7876이거나, 7900이거나, 7902인 행을 반환하라"다.

3.2 OR 조건의 정의

OR는 여러 조건 중 하나라도 참이면 참을 반환하는 논리 연산자다.

```
SELECT * FROM emp WHERE empno = 7876 OR empno = 7900 OR empno = 7902;
```

이 쿼리의 의미도 "empno가 7876이거나, 7900이거나, 7902인 행을 반환하라"다.

3.3 동치성 증명

두 쿼리가 동치라는 것은 **어떤 입력 데이터에 대해서도 동일한 결과 집합을 반환**한다는 의미다.

empno	IN (7876, 7900, 7902)	empno=7876 OR empno=7900 OR empno=7902
7876	TRUE	TRUE OR FALSE OR FALSE = TRUE
7900	TRUE	FALSE OR TRUE OR FALSE = TRUE
7902	TRUE	FALSE OR FALSE OR TRUE = TRUE
7839	FALSE	FALSE OR FALSE OR FALSE = FALSE
7566	FALSE	FALSE OR FALSE OR FALSE = FALSE

모든 경우에서 두 조건의 결과가 동일하다. 따라서 논리적으로 동치다.

3.4 SQL 표준에서의 정의

SQL 표준(ISO/IEC 9075)에서 IN 절은 다음과 같이 정의된다:

```
x IN (v1, v2, v3)  ≡  (x = v1) OR (x = v2) OR (x = v3)
```

IN은 OR 조건의 **축약 표현(syntactic sugar)**이다. 개발자가 읽고 쓰기 편하도록 제공되는 문법이며, 내부적으로는 동일한 논리 연산이다.

3.5 왜 이것이 중요한가

Optimizer의 관점

Oracle Optimizer는 IN 절을 만나면 두 가지 처리 전략을 선택할 수 있다.

전략 1: INLIST ITERATOR

IN 절을 그대로 유지하고, 인덱스를 반복 접근한다. IN 절의 각 값에 대해 동일한 인덱스 경로를 재사용한다.

전략 2: OR Expansion (CONCATENATION)

IN 절을 OR 조건으로 변환(확장)한 후, 각 OR 분기를 독립적인 쿼리 블록으로 처리한다.

```
-- 원본
WHERE empno IN (7876, 7900, 7902)

-- OR Expansion 후
WHERE empno = 7876
UNION ALL
WHERE empno = 7900
UNION ALL
WHERE empno = 7902
```

Oracle IN 절 1000개 제한의 배경

Oracle이 IN 절에 1000개 제한을 둔 이유도 이 동치성에서 비롯된다. IN 절이 OR 조건과 동치이므로, IN 절의 값이 많아지면 OR 조건이 그만큼 길어진다. 파싱 복잡도, 실행 계획 수립 비용, 메모리 사용량이 증가한다.

1000개를 초과하면 Oracle(또는 Hibernate)이 강제로 여러 IN 절로 분할하고 OR로 연결한다:

```
-- 1500개 값을 가진 IN 절
WHERE id IN (val1, ..., val1500)

-- 분할 후
WHERE id IN (val1, ..., val1000) OR id IN (val1001, ..., val1500)
```

이때 실행 계획에 CONCATENATION이 나타나며, 이것이 Bulk Fetch 방식의 성능 저하 원인이다.

3.6 결론

IN 절이 OR 조건과 동치라는 것은 SQL 기초 문법이므로 별도 문서 학습이 필요 없다. 이 동치성을 이해하면 Oracle이 IN 절을 왜 INLIST ITERATOR 또는 CONCATENATION으로 처리하는지, 그리고 1000개 제한이 왜 존재하는지의 논리적 배경을 파악할 수 있다.

3.7 논리적 동치와 물리적 실행의 분리

IN 절의 의미론적 제약

`col IN (a, b, c)`는 **단일 컬럼**에 대해 **동일한 비교 연산(=)**을 **여러 값**에 적용한다는 명확한 의미를 가진다. 이 구조적 제약 덕분에 Oracle Optimizer는 값들을 인덱스 키 순서로 정렬하고, 인덱스 리프 블록을 순차 스캔하는 최적화(INLIST ITERATOR)를 적용할 수 있다.

OR 절의 일반성

`cond1 OR cond2 OR cond3`에서 각 조건은 서로 다른 컬럼, 다른 테이블, 다른 연산자를 사용할 수 있다. Oracle은 이 일반성 때문에 각 조건을 독립적인 실행 단위로 처리해야 할 수 있다.

단, 모든 OR 조건이 `col = value` 형태로 동일 컬럼을 참조하면, Oracle은 이를 IN으로 변환하여 INLIST ITERATOR로 최적화한다. 이 경우 논리적 동치성이 물리적 실행 계획에서도 동일하게 나타난다.

```
-- Oracle이 내부적으로 IN으로 변환 가능
WHERE empno = 7876 OR empno = 7900 OR empno = 7902
→ INLIST ITERATOR 사용 가능
```

Hibernate OR 분해가 최적화되지 않는 이유

```
WHERE col IN (1000개) OR col IN (1000개) OR ...
```

이 구조에서 각 OR 브랜치가 이미 IN 절을 포함하고 있다. Oracle이 전체를 하나의 IN으로 병합하면 1000개 제한을 다시 위반하게 된다. 따라서 Oracle은 병합을 수행할 수 없고, OR의 일반적 처리 방식인 CONCATENATION을 적용한다.

조건 형태	Oracle 변환	실행 계획
`col = a OR col = b OR col = c`	IN으로 병합 가능	INLIST ITERATOR
`col IN (a,b,c)`	그대로 유지	INLIST ITERATOR
`col IN (1000개) OR col IN (1000개)`	병합 불가	CONCATENATION

결론적으로, IN과 OR의 **논리적 동치성**은 결과 집합의 동일성을 보장하지만, **물리적 실행 방식**은 SQL 구조에 따라 달라진다. 단순한 OR 조건은 IN으로 최적화될 수 있지만, Hibernate가 생성하는 복합 OR 구조는 CONCATENATION으로 처리되어 성능 저하가 발생한다.

---

4. CONCATENATION 비효율성 상세 분석

4.1 "각 분기가 독립적으로 실행된다"의 의미

INLIST ITERATOR의 실행 구조

INLIST ITERATOR는 단일 실행 계획을 수립하고, IN 절의 값만 교체하며 반복 실행한다.

```
[실행 계획 수립: 1회]
  INDEX RANGE SCAN → TABLE ACCESS BY ROWID

[실행 단계]
  값 7876 대입 → 실행
  값 7900 대입 → 실행 (동일 계획 재사용)
  값 7902 대입 → 실행 (동일 계획 재사용)
```

실행 계획 수립은 1회만 발생한다. 이후 반복 실행에서는 값만 바꿔서 동일한 Row Source를 재사용한다.

CONCATENATION의 실행 구조

CONCATENATION은 각 OR 분기를 별도의 쿼리 블록으로 취급한다.

```
WHERE id IN (val1, ..., val1000) OR id IN (val1001, ..., val1500)
```

이것은 내부적으로 다음과 같이 처리된다:

```
[쿼리 블록 1]
  SELECT ... WHERE id IN (val1, ..., val1000)
  → 별도 실행 계획 수립
  → 별도 Row Source 생성

[쿼리 블록 2]
  SELECT ... WHERE id IN (val1001, ..., val1500)
  → 별도 실행 계획 수립
  → 별도 Row Source 생성

[CONCATENATION]
  쿼리 블록 1 결과 + 쿼리 블록 2 결과 병합
```

독립 실행의 오버헤드

각 쿼리 블록이 독립적이라는 것은 다음을 의미한다:

항목	INLIST ITERATOR	CONCATENATION (2분기)
실행 계획 수립 횟수	1회	2회
Row Source 생성 횟수	1개	2개
메모리 할당	1세트	2세트
컨텍스트 스위칭	없음	분기 간 전환 발생

실행 계획 수립 자체가 CPU를 소모하는 작업이다. CONCATENATION에서는 이 작업이 분기 수만큼 반복된다.

4.2 "별도의 Operation으로 처리되어 오버헤드가 증가한다"의 의미

실행 계획 비교

INLIST ITERATOR 실행 계획

```
-------------------------------------------------------------
| Id | Operation                     | Name              |
-------------------------------------------------------------
|  0 | SELECT STATEMENT              |                   |
|  1 |  INLIST ITERATOR              |                   |
|  2 |   TABLE ACCESS BY INDEX ROWID| REVIEW_STATISTICS |
|  3 |    INDEX RANGE SCAN           | PK_REVIEW_STATS   |
-------------------------------------------------------------
```

Operation 수: 4개

CONCATENATION 실행 계획 (2분기)

```
-------------------------------------------------------------
| Id | Operation                     | Name              |
-------------------------------------------------------------
|  0 | SELECT STATEMENT              |                   |
|  1 |  CONCATENATION                |                   |
|  2 |   TABLE ACCESS BY INDEX ROWID| REVIEW_STATISTICS |
|  3 |    INDEX RANGE SCAN           | PK_REVIEW_STATS   |
|  4 |   TABLE ACCESS BY INDEX ROWID| REVIEW_STATISTICS |
|  5 |    INDEX RANGE SCAN           | PK_REVIEW_STATS   |
-------------------------------------------------------------
```

Operation 수: 6개

Operation 증가가 의미하는 것

각 Operation은 Oracle 내부에서 **Row Source**라는 실행 단위로 변환된다. Row Source는 다음 리소스를 소모한다:

메모리 할당: 각 Row Source는 자체 작업 메모리를 할당받는다. CONCATENATION에서 INDEX RANGE SCAN이 2번 나타나면, 인덱스 스캔을 위한 메모리가 2세트 할당된다.

커서 관리: 각 Row Source는 별도의 커서 상태를 유지한다. 현재 위치, 버퍼 상태, 페치 카운트 등이 각각 관리된다.

함수 호출 오버헤드: Row Source 간 데이터 전달은 함수 호출을 통해 이루어진다. Operation이 많을수록 함수 호출 횟수가 증가한다.

인덱스 접근 패턴의 차이

INLIST ITERATOR의 인덱스 접근

```
[단일 INDEX RANGE SCAN Row Source]
│
├── B-tree 루트 블록 접근
├── 값 7876으로 리프 블록 탐색 → ROWID 획득
├── 값 7900으로 리프 블록 탐색 → ROWID 획득  ← 루트 블록 캐시 재사용
├── 값 7902로 리프 블록 탐색 → ROWID 획득   ← 루트 블록 캐시 재사용
```

인덱스의 루트 블록과 브랜치 블록이 Row Source의 로컬 버퍼에 캐시되어 재사용된다.

CONCATENATION의 인덱스 접근

```
[첫 번째 INDEX RANGE SCAN Row Source]
├── B-tree 루트 블록 접근
├── 리프 블록 탐색 → ROWID 획득
└── Row Source 종료, 로컬 버퍼 해제

[두 번째 INDEX RANGE SCAN Row Source]
├── B-tree 루트 블록 재접근  ← 이전 Row Source의 캐시 사용 불가
├── 리프 블록 탐색 → ROWID 획득
└── Row Source 종료
```

각 Row Source가 독립적이므로 이전 Row Source에서 캐시된 인덱스 블록을 재사용할 수 없다. Buffer Cache에 남아있으면 재사용 가능하지만, Row Source 내부의 로컬 버퍼 캐시는 공유되지 않는다.

4.3 "결과 병합 과정에서 중복 제거가 필요할 수 있다"의 의미

중복 발생 가능 시나리오

단순 IN 절에서는 중복이 발생하지 않는다. 그러나 다음 상황에서 중복이 발생할 수 있다:

복합 조건이 있는 경우

```
SELECT * FROM emp
WHERE (deptno = 10 AND sal > 3000)
   OR (deptno = 10 AND job = 'MANAGER')
```

deptno=10이고 sal>3000이면서 동시에 job='MANAGER'인 행은 두 분기 모두에서 반환된다.

IN 절 값에 중복이 있는 경우 (Hibernate 생성 쿼리)

Hibernate가 IN 절을 분할할 때 경계값 처리 오류로 중복이 포함될 가능성이 있다(드물지만).

CONCATENATION의 중복 처리

CONCATENATION은 기본적으로 **UNION ALL** 시맨틱이다. 중복을 제거하지 않고 모든 분기의 결과를 그대로 합친다.

```
[분기 1 결과]: {A, B, C}
[분기 2 결과]: {C, D, E}
[CONCATENATION 결과]: {A, B, C, C, D, E}  ← C가 중복
```

중복 제거가 필요하면 상위에 **SORT UNIQUE** 또는 **HASH UNIQUE** Operation이 추가된다:

```
-----------------------------------------------------------
| Id | Operation                    | Name               |
-----------------------------------------------------------
|  0 | SELECT STATEMENT             |                    |
|  1 |  SORT UNIQUE                 |                    |  ← 추가 Operation
|  2 |   CONCATENATION              |                    |
|  3 |    TABLE ACCESS BY INDEX ROWID| REVIEW_STATISTICS|
|  4 |     INDEX RANGE SCAN         | PK_REVIEW_STATS    |
|  5 |    TABLE ACCESS BY INDEX ROWID| REVIEW_STATISTICS|
|  6 |     INDEX RANGE SCAN         | PK_REVIEW_STATS    |
-----------------------------------------------------------
```

SORT UNIQUE는 전체 결과를 메모리에 올려 정렬한 후 중복을 제거한다. 결과 집합이 크면 디스크 소트가 발생할 수 있다.

Wherehouse 프로젝트에서의 해당 여부

REVIEW_STATISTICS 테이블의 쿼리는 단순 IN 절이고, PROPERTY_ID가 Primary Key이므로 중복이 발생하지 않는다. 따라서 중복 제거 오버헤드는 해당 케이스에서 발생하지 않을 가능성이 높다.

다만 CONCATENATION 자체의 오버헤드(독립 실행, 별도 Row Source)는 여전히 존재한다.

4.4 비용(Cost) 관점에서의 비교

가상 시나리오

REVIEW_STATISTICS 테이블(52,020행)에서 1,500개 PROPERTY_ID를 조회한다고 가정한다.

INLIST ITERATOR (1,500개를 한 번에 처리 가능했다면)

```
Cost 계산:
- 인덱스 루트 블록 접근: 1회
- 인덱스 브랜치 블록 접근: ~3회 (B-tree 높이)
- 인덱스 리프 블록 접근: 1,500회 (각 값마다)
- 테이블 블록 접근: 1,500회 (각 ROWID마다)

총 예상 Cost: ~50
```

CONCATENATION (1,000개 + 500개로 분할)

```
[분기 1: 1,000개]
- 인덱스 루트 블록 접근: 1회
- 인덱스 브랜치 블록 접근: ~3회
- 인덱스 리프 블록 접근: 1,000회
- 테이블 블록 접근: 1,000회
Cost: ~35

[분기 2: 500개]
- 인덱스 루트 블록 접근: 1회  ← 중복
- 인덱스 브랜치 블록 접근: ~3회  ← 중복
- 인덱스 리프 블록 접근: 500회
- 테이블 블록 접근: 500회
Cost: ~20

[CONCATENATION 오버헤드]
- 결과 병합: +5

총 예상 Cost: ~60
```

CONCATENATION이 약 20% 더 높은 Cost를 가진다. 분기 수가 많아질수록 이 차이는 커진다.

4.5 비효율성 요약

비효율성 원인	구체적 내용
독립 실행	각 분기가 별도 쿼리 블록으로 처리되어 실행 계획 수립, Row Source 생성이 분기 수만큼 반복됨
별도 Operation	동일 인덱스에 대한 접근이 별도 Row Source로 처리되어 로컬 버퍼 캐시 공유 불가, 메모리 중복 할당
결과 병합	중복 발생 시 SORT UNIQUE 추가 필요. 단순 IN 절에서는 해당 없으나 CONCATENATION 자체 병합 오버헤드 존재

---

5. Oracle IN 절 1000개 제한의 기술적 배경

5.1 제한의 정확한 정의

Oracle 공식 제한

Oracle Database는 단일 IN 절에 최대 1000개의 리터럴 값만 허용한다. 이것은 Oracle SQL 파서의 하드코딩된 제한이다.

```
-- 허용됨 (1000개 이하)
SELECT * FROM emp WHERE id IN (1, 2, 3, ..., 1000);

-- ORA-01795 오류 발생 (1000개 초과)
SELECT * FROM emp WHERE id IN (1, 2, 3, ..., 1001);
```

```
ORA-01795: maximum number of expressions in a list is 1000
```

이 오류는 파싱 단계에서 발생한다. 쿼리가 Optimizer에 도달하기 전에 SQL Parser가 거부한다.

5.2 왜 1000개 제한이 존재하는가

SQL 파싱 복잡도

Oracle SQL Parser는 IN 절을 파싱할 때 각 값을 개별 노드로 변환하여 파스 트리(Parse Tree)를 생성한다.

```
WHERE id IN (1, 2, 3, 4, 5)
```

이 조건은 다음과 같은 파스 트리 구조로 변환된다:

```
        IN_LIST
       /  |  \  \  \
      1   2   3  4  5
```

IN 절의 값이 N개면 파스 트리에 N개의 리프 노드가 생성된다. 파스 트리 구축, 시맨틱 분석, 메모리 할당 모두 N에 비례하여 증가한다.

1000개 제한의 근거: Oracle 엔지니어들이 실험적으로 결정한 임계값이다. 1000개까지는 파싱 오버헤드가 수용 가능한 범위이고, 이를 초과하면 파싱 시간과 메모리 사용량이 급격히 증가한다.

Shared Pool 메모리 압박

파싱된 SQL 문장은 **Library Cache**(Shared Pool의 일부)에 저장된다. IN 절의 값이 많을수록 저장해야 할 SQL 텍스트와 파스 트리가 커진다.

```
[Shared Pool]
├── Library Cache
│   ├── SQL Text: "SELECT * FROM emp WHERE id IN (1,2,3,...,10000)"
│   │             → 약 50KB (10000개 값 가정, 각 값 평균 5바이트)
│   │
│   ├── Parse Tree: 10000개 노드
│   │             → 약 400KB (노드당 ~40바이트)
│   │
│   └── 실행 계획
│               → 추가 메모리
```

대량의 IN 절이 포함된 쿼리가 Library Cache에 캐싱되면:

첫째, 다른 SQL 문장을 위한 공간이 부족해진다.

둘째, Library Cache LRU 알고리즘에 의해 유용한 캐시 엔트리가 밀려난다.

셋째, 전체 시스템의 Hard Parse 비율이 증가한다.

OR Expansion 시 실행 계획 폭발

IN 절이 OR 조건과 동치이므로, Optimizer는 IN 절을 OR로 확장(OR Expansion)하여 처리할 수 있다. 이때 실행 계획의 복잡도가 급증한다.

```
-- 원본
WHERE id IN (1, 2, 3, ..., 10000)

-- OR Expansion 후 (개념적)
WHERE id = 1 OR id = 2 OR id = 3 OR ... OR id = 10000
```

Optimizer가 각 OR 분기에 대해 최적 접근 경로를 계산해야 한다. N개의 값이 있으면 Optimizer가 고려해야 할 경우의 수가 기하급수적으로 증가한다. 1000개 제한은 이 폭발을 방지한다.

네트워크 패킷 크기 고려

SQL 문장은 클라이언트에서 서버로 네트워크를 통해 전송된다. Oracle Net의 SDU(Session Data Unit) 기본값은 8KB다.

```
10000개 값을 가진 IN 절 SQL 문장 크기:
- "SELECT * FROM emp WHERE id IN (" = ~30바이트
- 각 값 (평균 5자리 숫자 + 콤마) = ~6바이트 × 10000 = 60KB
- 총 크기: ~60KB

→ 8KB SDU로 전송 시 최소 8회의 패킷 전송 필요
→ 네트워크 라운드트립 증가
```

1000개 제한은 단일 SQL 문장의 크기를 합리적인 범위로 유지한다.

5.3 제한 초과 시 발생하는 일

순수 Oracle SQL의 경우

1000개를 초과하면 ORA-01795 오류가 즉시 발생한다. 쿼리가 실행되지 않는다.

```
ORA-01795: maximum number of expressions in a list is 1000
```

Hibernate/JPA의 경우

Hibernate는 이 제한을 알고 있으며, 자동으로 우회 처리한다. 그러나 이 우회 처리가 성능 문제를 유발한다.

Hibernate의 처리 방식

```
// 애플리케이션 코드
List<String> ids = getPropertyIds(); // 8660개
repository.findAllByPropertyIdIn(ids);
```

Hibernate가 생성하는 SQL:

```
-- 8660개를 1000개 단위로 분할하고 OR로 연결
SELECT * FROM REVIEW_STATISTICS
WHERE PROPERTY_ID IN (?, ?, ..., ?)      -- 1000개
   OR PROPERTY_ID IN (?, ?, ..., ?)      -- 1000개
   OR PROPERTY_ID IN (?, ?, ..., ?)      -- 1000개
   OR PROPERTY_ID IN (?, ?, ..., ?)      -- 1000개
   OR PROPERTY_ID IN (?, ?, ..., ?)      -- 1000개
   OR PROPERTY_ID IN (?, ?, ..., ?)      -- 1000개
   OR PROPERTY_ID IN (?, ?, ..., ?)      -- 1000개
   OR PROPERTY_ID IN (?, ?, ..., ?)      -- 1000개
   OR PROPERTY_ID IN (?, ?, ..., ?)      -- 660개
```

9개의 IN 절이 OR로 연결된다.

5.4 OR 분해가 유발하는 실행 계획 변화

단일 IN 절 (1000개 이하)

```
SELECT * FROM REVIEW_STATISTICS WHERE PROPERTY_ID IN (?, ?, ..., ?)  -- 500개
```

실행 계획:

```
-----------------------------------------------------------
| Id | Operation                    | Name              |
-----------------------------------------------------------
|  0 | SELECT STATEMENT             |                   |
|  1 |  INLIST ITERATOR             |                   |
|  2 |   TABLE ACCESS BY INDEX ROWID| REVIEW_STATISTICS |
|  3 |    INDEX RANGE SCAN          | PK_REVIEW_STATS   |
-----------------------------------------------------------
```

INLIST ITERATOR가 IN 절의 각 값에 대해 인덱스를 효율적으로 반복 접근한다.

OR로 분해된 IN 절 (1000개 초과)

```
SELECT * FROM REVIEW_STATISTICS
WHERE PROPERTY_ID IN (?, ..., ?)    -- 1000개
   OR PROPERTY_ID IN (?, ..., ?);   -- 660개
```

실행 계획:

```
-----------------------------------------------------
| Id | Operation                    | Name              |
-----------------------------------------------------
|  0 | SELECT STATEMENT             |                   |
|  1 |  CONCATENATION               |                   |
|  2 |   INLIST ITERATOR            |                   |
|  3 |    TABLE ACCESS BY INDEX ROWID| REVIEW_STATISTICS|
|  4 |     INDEX RANGE SCAN         | PK_REVIEW_STATS   |
|  5 |   INLIST ITERATOR            |                   |
|  6 |    TABLE ACCESS BY INDEX ROWID| REVIEW_STATISTICS|
|  7 |     INDEX RANGE SCAN         | PK_REVIEW_STATS   |
-----------------------------------------------------
```

CONCATENATION이 나타난다. 각 OR 분기가 독립적인 INLIST ITERATOR로 처리된 후 결과가 병합된다.

8660개의 경우 (Wherehouse 프로젝트)

9개의 OR 분기가 생성되므로:

```
-----------------------------------------------------
| Id | Operation                    | Name              |
-----------------------------------------------------
|  0 | SELECT STATEMENT             |                   |
|  1 |  CONCATENATION               |                   |
|  2 |   INLIST ITERATOR            |                   |  ← 분기 1
|  3 |    TABLE ACCESS BY INDEX ROWID| REVIEW_STATISTICS|
|  4 |     INDEX RANGE SCAN         | PK_REVIEW_STATS   |
|  5 |   INLIST ITERATOR            |                   |  ← 분기 2
|  6 |    TABLE ACCESS BY INDEX ROWID| REVIEW_STATISTICS|
|  7 |     INDEX RANGE SCAN         | PK_REVIEW_STATS   |
      ... (분기 3~8 생략) ...
| 26 |   INLIST ITERATOR            |                   |  ← 분기 9
| 27 |    TABLE ACCESS BY INDEX ROWID| REVIEW_STATISTICS|
| 28 |     INDEX RANGE SCAN         | PK_REVIEW_STATS   |
-----------------------------------------------------
```

Operation 수가 급증한다. 각 분기마다 별도의 Row Source가 생성되고, CONCATENATION이 모든 결과를 병합한다.

5.5 Hard Parse 문제

IN 절 값 개수가 가변적일 때

Hibernate가 생성하는 SQL 문장의 형태가 매번 달라진다:

```
-- 요청 1: 8660개 → 9개 IN 절 OR 연결
SELECT ... WHERE PROPERTY_ID IN (?,?,...) OR PROPERTY_ID IN (?,?,...) OR ...

-- 요청 2: 7500개 → 8개 IN 절 OR 연결
SELECT ... WHERE PROPERTY_ID IN (?,?,...) OR PROPERTY_ID IN (?,?,...) OR ...

-- 요청 3: 9200개 → 10개 IN 절 OR 연결
SELECT ... WHERE PROPERTY_ID IN (?,?,...) OR PROPERTY_ID IN (?,?,...) OR ...
```

각 SQL 문장의 구조가 다르므로 **Library Cache에서 일치하는 SQL을 찾을 수 없다**. 매번 Hard Parse가 발생한다.

Hard Parse의 비용

```
[Hard Parse 발생 시]
1. SQL Parser: 문법 검사, 파스 트리 생성
2. Semantic Analyzer: 테이블/컬럼 존재 확인, 권한 검사
3. Query Transformer: 쿼리 변환 (OR Expansion 등)
4. Estimator: 통계 기반 Cardinality 추정
5. Plan Generator: 가능한 실행 계획 후보 생성 및 비용 계산
6. 최적 계획 선택 및 캐싱

→ CPU 집약적 작업, 수 ms ~ 수십 ms 소요
```

Chunk 방식이 Soft Parse를 유도하는 이유

Chunk 방식은 항상 동일한 크기(1000개)로 IN 절을 구성한다:

```
-- 쿼리 1: 항상 1000개
SELECT ... WHERE PROPERTY_ID IN (?,?,?,...,?)  -- 1000개 placeholder

-- 쿼리 2: 항상 1000개 (값만 다름)
SELECT ... WHERE PROPERTY_ID IN (?,?,?,...,?)  -- 1000개 placeholder

-- 쿼리 9: 나머지 (660개지만, 패딩하면 1000개로 통일 가능)
SELECT ... WHERE PROPERTY_ID IN (?,?,?,...,?)  -- 1000개 placeholder
```

SQL 문장의 구조가 동일하므로 Library Cache에서 일치하는 SQL을 찾을 수 있다. 첫 번째 실행에서만 Hard Parse가 발생하고, 이후 실행은 Soft Parse로 처리된다.

5.6 Wherehouse 프로젝트에서의 구체적 영향

Bulk Fetch 방식 (8660개 한 번에)

```
[발생하는 문제]
1. Hibernate가 9개의 IN 절을 OR로 연결
2. Oracle 실행 계획에 CONCATENATION 발생
3. 9개 분기 각각이 독립적 Row Source로 처리
4. SQL 문장 크기 증가 (네트워크 오버헤드)
5. 매 요청마다 Hard Parse 가능성 (값 개수 변동 시)

[Image 3 데이터]
- 평균 응답 시간: 6,001ms
- RDB 조회 시간: 1,748ms (28.8%)
- 쿼리 실행 횟수: 1회
```

Chunk 방식 (1000개 단위 분할)

```
[해결되는 문제]
1. 각 쿼리가 단일 IN 절 (1000개 이하)
2. Oracle 실행 계획에 INLIST ITERATOR 사용
3. 단일 Row Source로 효율적 처리
4. SQL 문장 크기 일정 (네트워크 예측 가능)
5. 동일 구조 SQL로 Soft Parse 유도

[Image 2 데이터]
- 평균 응답 시간: 4,434ms
- RDB 조회 시간: 548ms (12.1%)
- 쿼리 실행 횟수: 9회
```

역설적 결과의 설명

9회 쿼리 실행이 1회보다 빠른 이유:

항목	Bulk Fetch (1회)	Chunk (9회)
실행 계획	CONCATENATION (9분기)	INLIST ITERATOR × 9
Row Source 수	27개 (분기당 3개 × 9)	3개 × 9 = 27개 (동일하지만 독립 실행)
Hard Parse	매번 발생 가능	첫 쿼리만 (이후 Soft Parse)
SQL 문장 크기	~100KB	~12KB × 9
CONCATENATION 병합	있음	없음

Chunk 방식은 쿼리 횟수는 증가하지만, 각 쿼리가 단순하고 효율적인 실행 계획을 가진다. 총 비용은 오히려 감소한다.

5.7 1000개 제한 요약

제한 배경	구체적 내용
파싱 복잡도	IN 절의 값 개수에 비례하여 파스 트리 노드 증가, 1000개는 실험적 임계값
메모리 압박	Library Cache에 대형 SQL 캐싱 시 다른 SQL 밀려남, 전체 Hard Parse 비율 증가
실행 계획 폭발	OR Expansion 시 Optimizer 고려 경우의 수 기하급수 증가
네트워크	대형 SQL 문장 전송 시 패킷 분할, 라운드트립 증가
Hibernate 우회	1000개 초과 시 자동 OR 분해, CONCATENATION 실행 계획 유발

---

6. CONCATENATION이 OR 분기를 별도 쿼리 블록으로 취급하는 이유

6.1 OR 조건의 본질적 특성

각 OR 분기가 서로 다른 최적 접근 경로를 가질 수 있다

다음 쿼리를 보자:

```
SELECT * FROM emp
WHERE deptno = 10
   OR job = 'MANAGER'
```

이 쿼리에서 두 조건은 서로 다른 컬럼을 참조한다:

- `deptno = 10` : DEPTNO 컬럼에 인덱스가 있다면 해당 인덱스 사용이 최적
- `job = 'MANAGER'` : JOB 컬럼에 인덱스가 있다면 해당 인덱스 사용이 최적

문제: 단일 테이블 스캔에서 두 개의 서로 다른 인덱스를 동시에 사용할 수 없다.

단일 실행 경로의 한계

Optimizer가 이 쿼리를 단일 실행 경로로 처리하려면 다음 중 하나를 선택해야 한다:

```
[선택지 1] DEPTNO 인덱스 사용
- deptno = 10 조건은 인덱스로 처리
- job = 'MANAGER' 조건은 테이블 접근 후 필터링
- job = 'MANAGER'이면서 deptno ≠ 10인 행은 누락됨 ← 오류

[선택지 2] JOB 인덱스 사용
- job = 'MANAGER' 조건은 인덱스로 처리
- deptno = 10 조건은 테이블 접근 후 필터링
- deptno = 10이면서 job ≠ 'MANAGER'인 행은 누락됨 ← 오류

[선택지 3] TABLE ACCESS FULL
- 두 조건 모두 테이블 스캔 후 필터링
- 정확하지만 인덱스 활용 불가 ← 비효율
```

OR 조건에서는 어느 한쪽 조건만 만족해도 결과에 포함되어야 한다. 단일 인덱스 접근으로는 이를 보장할 수 없다.

6.2 Oracle의 해결책: OR Expansion

OR Expansion 변환

Oracle Query Transformer는 OR 조건을 UNION ALL로 변환한다. 이 변환을 OR Expansion이라 한다.

```
-- 원본 쿼리
SELECT * FROM emp
WHERE deptno = 10
   OR job = 'MANAGER'

-- OR Expansion 후 (개념적 변환)
SELECT * FROM emp WHERE deptno = 10
UNION ALL
SELECT * FROM emp WHERE job = 'MANAGER' AND LNNVL(deptno = 10)
```

LNNVL 함수의 역할

`LNNVL(deptno = 10)`은 "deptno가 10이 아니거나 NULL인 경우 TRUE"를 반환한다. 이것은 **중복 제거**를 위한 것이다.

두 조건을 모두 만족하는 행(deptno=10 AND job='MANAGER')이 있을 때: - 첫 번째 분기에서 반환됨 (deptno = 10 조건) - 두 번째 분기에서 LNNVL에 의해 제외됨

UNION ALL이지만 중복이 발생하지 않는다.

실행 계획에서의 표현

이 UNION ALL 변환이 실행 계획에서 **CONCATENATION**으로 나타난다:

```
-----------------------------------------------------
| Id | Operation                    | Name          |
-----------------------------------------------------
|  0 | SELECT STATEMENT             |               |
|  1 |  CONCATENATION               |               |
|  2 |   TABLE ACCESS BY INDEX ROWID| EMP           |
|  3 |    INDEX RANGE SCAN          | IDX_DEPTNO    |  ← deptno = 10
|  4 |   TABLE ACCESS BY INDEX ROWID| EMP           |
|  5 |    INDEX RANGE SCAN          | IDX_JOB       |  ← job = 'MANAGER'
-----------------------------------------------------
```

CONCATENATION = UNION ALL의 실행 계획 표현이다.

6.3 왜 "별도 쿼리 블록"이어야 하는가

쿼리 블록(Query Block)의 정의

Oracle에서 쿼리 블록은 **독립적으로 최적화되는 SQL 단위**다. 각 쿼리 블록은:

- 자체 FROM 절을 가진다
- 자체 WHERE 조건을 가진다
- 독립적인 실행 계획을 가진다

OR Expansion이 별도 쿼리 블록을 생성하는 이유

이유 1: 각 분기에 최적 접근 경로 적용

```
[쿼리 블록 1: deptno = 10]
- Optimizer 분석: DEPTNO 인덱스 사용이 최적
- 선택된 계획: INDEX RANGE SCAN (IDX_DEPTNO)

[쿼리 블록 2: job = 'MANAGER']
- Optimizer 분석: JOB 인덱스 사용이 최적
- 선택된 계획: INDEX RANGE SCAN (IDX_JOB)
```

별도 쿼리 블록이므로 각 분기가 **자신에게 최적인 인덱스를 독립적으로 선택**할 수 있다.

이유 2: 통계 기반 비용 계산의 독립성

```
[쿼리 블록 1]
- Cardinality 추정: deptno = 10인 행 = 500개
- Cost 계산: 인덱스 스캔 비용 + 테이블 접근 비용

[쿼리 블록 2]
- Cardinality 추정: job = 'MANAGER'인 행 = 50개
- Cost 계산: 인덱스 스캔 비용 + 테이블 접근 비용
```

각 분기의 선택도(Selectivity)가 다르므로 비용 계산도 독립적으로 수행되어야 한다.

이유 3: 실행 엔진의 단순화

Oracle 실행 엔진(Row Source Generator)은 각 쿼리 블록을 **독립적인 Row Source Tree**로 변환한다:

```
[CONCATENATION Row Source]
├── [Row Source Tree 1]
│   └── TABLE ACCESS BY INDEX ROWID
│       └── INDEX RANGE SCAN (IDX_DEPTNO)
│
└── [Row Source Tree 2]
    └── TABLE ACCESS BY INDEX ROWID
        └── INDEX RANGE SCAN (IDX_JOB)
```

CONCATENATION Row Source는 단순히 각 하위 Row Source의 결과를 순차적으로 반환한다. 복잡한 OR 로직을 실행 엔진이 직접 처리하지 않아도 된다.

6.4 IN 절에서의 CONCATENATION

IN 절 OR 분해의 특수성

IN 절이 OR로 분해되는 경우는 위 예시와 다르다. 모든 분기가 동일 컬럼을 참조한다:

```
-- Hibernate가 생성한 쿼리 (1000개 초과)
WHERE PROPERTY_ID IN (val1, ..., val1000)
   OR PROPERTY_ID IN (val1001, ..., val1660)
```

동일 컬럼이므로 두 분기 모두 동일한 인덱스(PK_REVIEW_STATISTICS)를 사용한다. 그럼에도 CONCATENATION이 발생하는 이유는:

Oracle Parser의 처리 순서

```
[1단계: SQL Parser]
- IN 절 1000개 제한 검사
- 초과 시 파싱 오류 발생 (ORA-01795)

[Hibernate의 개입]
- 1000개 초과 감지
- 여러 IN 절로 분할하고 OR로 연결
- 변환된 SQL을 Oracle에 전송

[2단계: Oracle Query Transformer]
- OR 조건 감지
- OR Expansion 변환 규칙 적용
- 각 OR 분기를 별도 쿼리 블록으로 분리

[3단계: Optimizer]
- 각 쿼리 블록에 대해 독립적으로 최적화
- 동일 인덱스가 선택되더라도 별도 Row Source 생성

[4단계: 실행 계획 생성]
- CONCATENATION으로 결과 병합
```

Oracle Query Transformer는 OR 조건의 의미론적 내용을 분석하지 않는다. OR 조건이 있으면 OR Expansion 규칙을 적용할지 여부만 판단한다. 동일 컬럼인지, 다른 컬럼인지는 고려하지 않는다.

왜 동일 컬럼이어도 단일 INLIST ITERATOR로 병합하지 않는가

```
WHERE PROPERTY_ID IN (val1, ..., val1000)
   OR PROPERTY_ID IN (val1001, ..., val1660)
```

이 쿼리를 다음과 같이 변환할 수 있다고 가정하자:

```
WHERE PROPERTY_ID IN (val1, ..., val1000, val1001, ..., val1660)
```

이 변환이 발생하지 않는 이유:

첫째, 1000개 제한 위반. 변환 결과가 다시 1000개를 초과하므로 원점으로 돌아간다.

둘째, Query Transformer의 설계 원칙. Query Transformer는 쿼리를 확장(Expansion)하는 방향으로 설계되었다. 축소(Contraction) 변환은 제한적으로만 수행된다. IN 절 병합은 축소 변환에 해당하며, 1000개 제한과 충돌하므로 수행되지 않는다.

셋째, 변환 규칙의 일반성. OR Expansion은 일반적인 OR 조건에 대해 동작한다. "동일 컬럼 IN 절의 OR 연결"이라는 특수 케이스를 별도로 처리하는 규칙이 없다.

6.5 공식 문서 근거

Oracle SQL Tuning Guide - OR Expansion

Oracle 공식 문서에서 OR Expansion에 대해 다음과 같이 설명한다:

> "In OR expansion, the optimizer transforms a query block containing OR conditions into the form of a UNION ALL query that contains two or more branches."

번역: OR Expansion에서 Optimizer는 OR 조건을 포함하는 쿼리 블록을 두 개 이상의 분기를 가진 UNION ALL 형태로 변환한다.

> "Each branch of the UNION ALL query is optimized independently."

번역: UNION ALL 쿼리의 각 분기는 독립적으로 최적화된다.

Table 7-3 OPERATION and OPTIONS Values

앞서 fetch한 문서의 Table 7-3에서:

Operation	Option	Description
CONCATENATION		Operation accepting multiple sets of rows returning the union-all of the sets.

CONCATENATION은 여러 행 집합을 받아서 union-all을 반환하는 Operation으로 정의된다.

6.6 요약

질문	답변
왜 OR 분기가 별도 쿼리 블록이 되는가	OR Expansion 변환 규칙에 의해 OR 조건이 UNION ALL로 변환되고, 각 분기가 독립적인 쿼리 블록이 된다
왜 이 변환이 필요한가	각 OR 분기가 서로 다른 최적 접근 경로(인덱스)를 가질 수 있으므로, 독립적 최적화가 필요하다
CONCATENATION은 무엇인가	UNION ALL의 실행 계획 표현이다. 여러 Row Source의 결과를 순차적으로 반환한다
동일 컬럼 IN 절도 분리되는 이유	Query Transformer가 OR 조건의 의미론적 내용을 분석하지 않고, 일반적인 OR Expansion 규칙을 적용하기 때문이다

7. "OR Expansion 시 실행 계획 폭발" 정정

7.1 원래 설명의 문제점

원문:

> "Optimizer가 각 OR 분기에 대해 최적 접근 경로를 계산해야 한다. N개의 값이 있으면 Optimizer가 고려해야 할 경우의 수가 기하급수적으로 증가한다."

이 설명은 조인이 포함된 복잡한 쿼리에서 해당되는 이야기다. Wherehouse 프로젝트의 REVIEW_STATISTICS 쿼리는 단일 테이블 조회이므로 이 설명이 적용되지 않는다.

7.2 "실행 계획 폭발"이 실제로 발생하는 시나리오

조인이 있는 쿼리에서의 OR 조건

```
SELECT *
FROM orders o, customers c, products p
WHERE o.customer_id = c.id
  AND o.product_id = p.id
  AND (c.region = 'EAST' OR p.category = 'ELECTRONICS')
```

이 쿼리에서 OR 조건이 서로 다른 테이블의 컬럼을 참조한다. Optimizer가 고려해야 할 것:

- 조인 순서: orders → customers → products? customers → orders → products? ...
- 조인 방법: Nested Loops? Hash Join? Merge Join?
- 각 테이블 접근 경로: 인덱스? Full Scan?
- OR 조건 처리 방식: OR Expansion? Filter?

OR 조건이 여러 테이블에 걸쳐 있으면 조합의 수가 증가한다. 이것이 "실행 계획 폭발"이다.

Wherehouse 프로젝트에서는 해당 없음

```
SELECT PROPERTY_ID, AVG_RATING, ...
FROM REVIEW_STATISTICS
WHERE PROPERTY_ID IN (val1, ..., val8660)
```

- 단일 테이블 조회
- 조인 없음
- OR 조건이 단일 컬럼(PROPERTY_ID)에만 적용

• Optimizer가 고려할 조합: PK 인덱스 사용 vs Full Scan (2가지뿐)

"경우의 수 기하급수 증가"가 발생하지 않는다.

7.3 Wherehouse 프로젝트에서 실제로 문제가 되는 것

"실행 계획 폭발"은 해당되지 않는다. 실제 문제는 다음 두 가지다:

CONCATENATION으로 인한 Row Source 증가

8660개 IN 절이 9개 OR 분기로 분해되면:

```
CONCATENATION
├── INLIST ITERATOR (분기 1)
│   └── TABLE ACCESS BY INDEX ROWID
│       └── INDEX RANGE SCAN
├── INLIST ITERATOR (분기 2)
│   └── TABLE ACCESS BY INDEX ROWID
│       └── INDEX RANGE SCAN
... (분기 3~9)
```

Operation 수가 증가하고, 각 분기가 독립적인 Row Source로 처리된다. 이것은 "폭발"이 아니라 선형 증가다. 분기 수에 비례하여 오버헤드가 증가한다.

Hard Parse 문제

IN 절 값 개수가 가변적이면 SQL 문장 구조가 매번 달라져서 Library Cache에서 일치하는 SQL을 찾을 수 없다. 매번 Hard Parse가 발생한다.

7.4 결론

"OR Expansion 시 실행 계획 폭발"은 Wherehouse 프로젝트 맥락에서 학습할 필요가 없다. 이 개념은 조인이 포함된 복잡한 쿼리에서 해당되며, 단일 테이블 IN 절 조회에서는 발생하지 않는다.

1000개 제한의 실질적 영향은:

항목	해당 여부	설명
파싱 복잡도 증가	O	IN 절 값 개수에 비례하여 파스 트리 노드 증가
Shared Pool 메모리 압박	O	대형 SQL 문장이 Library Cache 공간 점유
실행 계획 폭발	X	조인이 없는 단일 테이블 쿼리에서는 해당 없음
CONCATENATION 오버헤드	O	OR 분기 수에 비례하여 Row Source 증가
Hard Parse	O	SQL 구조 변동 시 매번 발생

8. 실행 계획 구조 (1개 실행 계획, N개 분기)

8.1 용어 정리

"2개의 실행 계획"이 아니라 "1개의 실행 계획 안에 2개의 분기"

```
| Id | Operation                     | Name              |
|----|-------------------------------|-------------------|
|  0 | SELECT STATEMENT              |                   |
|  1 |  CONCATENATION                |                   |  ← 병합 지점
|  2 |   INLIST ITERATOR             |                   |  ┐
|  3 |    TABLE ACCESS BY INDEX ROWID| REVIEW_STATISTICS|  | 분기 1
|  4 |     INDEX RANGE SCAN          | PK_REVIEW_STATS   |  ┘
|  5 |   INLIST ITERATOR             |                   |  ┐
|  6 |    TABLE ACCESS BY INDEX ROWID| REVIEW_STATISTICS|  | 분기 2
|  7 |     INDEX RANGE SCAN          | PK_REVIEW_STATS   |  ┘
```

실행 계획은 1개다. 그러나 그 안에 **2개의 분기(branch)**가 존재한다.

8.2 실행 흐름

```
[단일 SQL 문장]
    ↓
[단일 실행 계획 수립] (Hard Parse 또는 Soft Parse)
    ↓
[실행 계획 내부 구조]
    │
    └── CONCATENATION (Id=1)
            │
            ├── 분기 1 실행 (Id=2,3,4)
            │    └── 1000개 값에 대해 인덱스 스캔 → 결과 집합 A
            │
            └── 분기 2 실행 (Id=5,6,7)
                 └── 660개 값에 대해 인덱스 스캔 → 결과 집합 B
            │
            └── 결과 집합 A + B 반환
```

8.3 핵심 정리

표현	정확성	설명
"2개의 실행 계획"	△	실행 계획은 1개, 내부에 2개 분기
"2개의 분기로 나뉜다"	O	CONCATENATION 하위에 2개 분기 존재
"각 분기가 독립적으로 실행된다"	O	각 분기가 별도 Row Source로 처리됨

성능 저하 원인: 분기가 2개면 Row Source도 2세트 생성되고, 인덱스 접근도 2번의 독립적인 시작점에서 수행된다. 분기 수가 9개(8660개 값의 경우)면 이 오버헤드가 9배로 증가한다.

---

9. 실행 계획 해석 (Section 7.1 기준)

9.1 295개 IN 절 실행 계획 캡처 분석

실행 계획 구조

Operation	Object Name	Options	Cardinality	Cost
SELECT STATEMENT			295	266
INLIST ITERATOR				
TABLE ACCESS	REVIEW_STATISTICS	BY INDEX ROWID	295	266
INDEX	PK_REVIEW_STATISTICS	UNIQUE SCAN	295	222

9.2 실행 순서 (Section 7.1 핵심)

실행 계획은 들여쓰기가 깊은 Operation부터 실행된다. 동일 깊이면 위에서 아래로 실행된다.

실행 순서:

```
4. INDEX UNIQUE SCAN (PK_REVIEW_STATISTICS)  ← 가장 먼저
3. TABLE ACCESS BY INDEX ROWID               ← 두 번째
2. INLIST ITERATOR                           ← 반복 제어
1. SELECT STATEMENT                          ← 결과 반환
```

실제 동작:

1. INLIST ITERATOR가 IN 절의 첫 번째 값('00002ba8f5f4651131019bf9ff57b5d')을 선택

2. INDEX UNIQUE SCAN이 PK_REVIEW_STATISTICS 인덱스에서 해당 값의 ROWID를 찾음
3. TABLE ACCESS BY INDEX ROWID가 ROWID로 테이블 행에 접근
4. INLIST ITERATOR가 다음 값으로 이동, 2-3번 반복 (295회)
5. SELECT STATEMENT가 모든 결과를 반환

9.3 각 컬럼 해석

컬럼	값	의미
CARDINALITY	295	Optimizer가 예상하는 반환 행 수. IN 절 값 개수(295)와 정확히 일치하므로 통계가 정확함
COST	266	Optimizer가 추정한 상대적 비용. 단위 없음. 다른 실행 계획과 비교 시 사용
OPTIONS	UNIQUE SCAN	PK/Unique 인덱스에서 단일 값을 찾는 스캔. RANGE SCAN보다 효율적
OPTIONS	BY INDEX ROWID	인덱스에서 얻은 ROWID로 테이블 행에 직접 접근

9.4 핵심 확인 사항

CONCATENATION 없음: 295개는 1000개 미만이므로 OR 분해가 발생하지 않았다. 단일 INLIST ITERATOR로 처리된다.

INLIST ITERATOR 존재: IN 절이 효율적으로 처리되고 있다.

INDEX UNIQUE SCAN: PK 인덱스를 사용하여 각 값을 O(log N) 복잡도로 탐색한다. TABLE ACCESS FULL이 아니므로 효율적이다.

9.5 다음 확인 필요 사항

이 실행 계획은 295개 IN 절의 정상 케이스다. 비교를 위해 다음 실행 계획이 필요하다:

1000개 초과 IN 절 (OR 분해 발생 케이스)의 실행 계획을 캡처하면 CONCATENATION이 나타나는 것을 확인할 수 있다.

---

10. 실행 계획 계층별 역할

10.1 실행 계획은 Row Source Tree다

Oracle 실행 계획은 트리 구조다. 각 Operation은 Row Source라는 실행 단위로 변환되고, 이들이 부모-자식 관계로 연결된다.

```
[Row Source Tree]

SELECT STATEMENT (루트)
       ↑ 행(row) 전달
INLIST ITERATOR (제어 노드)
       ↑ 행 전달
TABLE ACCESS BY INDEX ROWID (데이터 접근 노드)
       ↑ ROWID 전달
INDEX UNIQUE SCAN (리프 노드)
```

데이터 흐름 방향: 리프 노드에서 루트 노드로 아래에서 위로 행이 전달된다.

10.2 계층별 역할 분류

리프 노드 (Leaf Node) - 데이터 소스

트리의 가장 하위에 위치하며, 실제 데이터를 읽어오는 시작점이다.

Operation	역할
INDEX UNIQUE SCAN	인덱스에서 단일 키 값의 ROWID를 반환
INDEX RANGE SCAN	인덱스에서 범위 조건에 해당하는 ROWID들을 반환
TABLE ACCESS FULL	테이블 전체를 순차적으로 스캔하여 행 반환

캡처한 실행 계획에서:

```
INDEX UNIQUE SCAN (PK_REVIEW_STATISTICS)
→ PROPERTY_ID 값에 해당하는 ROWID를 반환
→ 출력: ROWID 1개
```

중간 노드 (Intermediate Node) - 데이터 변환/접근

리프 노드에서 받은 데이터를 변환하거나 추가 접근을 수행한다.

Operation	역할
TABLE ACCESS BY INDEX ROWID	ROWID를 받아 테이블에서 실제 행 데이터를 반환
SORT ORDER BY	하위에서 받은 행들을 정렬하여 반환
HASH JOIN	두 하위 노드의 결과를 해시 조인하여 반환
FILTER	하위에서 받은 행 중 조건에 맞는 것만 반환

캡처한 실행 계획에서:

```
TABLE ACCESS BY INDEX ROWID (REVIEW_STATISTICS)
→ 입력: ROWID
→ 테이블 블록에서 해당 ROWID의 행을 읽음
→ 출력: 완전한 행 데이터 (PROPERTY_ID, AVG_RATING, ...)
```

제어 노드 (Control Node) - 반복/분기 제어

하위 Operation의 실행 흐름을 제어한다. 직접 데이터를 읽지 않고, 하위 노드에 "다음 값으로 실행하라"는 신호를 보낸다.

Operation	역할
INLIST ITERATOR	IN 절의 각 값에 대해 하위 Operation을 반복 실행
CONCATENATION	여러 분기의 결과를 순차적으로 병합
NESTED LOOPS	외부 테이블의 각 행에 대해 내부 테이블 접근을 반복

캡처한 실행 계획에서:

```
INLIST ITERATOR
→ IN 절의 295개 값을 순회
→ 각 값에 대해 하위 트리(INDEX SCAN → TABLE ACCESS)를 실행
→ 하위 트리가 반환한 행을 상위로 전달
```

루트 노드 (Root Node) - 결과 반환

항상 SELECT STATEMENT가 루트에 위치하며, 최종 결과를 클라이언트에 반환한다.

```
SELECT STATEMENT
→ 하위에서 올라온 모든 행을 클라이언트에 전달
→ CARDINALITY: 최종 반환 예상 행 수
→ COST: 전체 쿼리의 총 비용
```

10.3 캡처한 실행 계획의 계층 구조 분석

```
[루트] SELECT STATEMENT ──────────────────── 클라이언트에 결과 반환
              ↑
[제어] INLIST ITERATOR ──────────────────── 295개 값에 대해 반복 제어
              ↑
[변환] TABLE ACCESS BY INDEX ROWID ────────── ROWID → 행 데이터 변환
              ↑
[리프] INDEX UNIQUE SCAN ──────────────────── PROPERTY_ID → ROWID 탐색
```

단일 값 처리 흐름 (예: '00002ba8f5f4651131019bf9ff57b5d'):

```
1. INLIST ITERATOR: "첫 번째 값 '00002ba8...'으로 하위 실행"
       ↓
2. INDEX UNIQUE SCAN: 인덱스 B-tree에서 '00002ba8...' 탐색
       ↓ (ROWID 반환: 예를 들어 'AAAHG7AAEAAAAJkAAA')
3. TABLE ACCESS BY INDEX ROWID: 해당 ROWID의 테이블 블록 접근
       ↓ (행 데이터 반환)
4. INLIST ITERATOR: 행을 상위로 전달, "다음 값 '0171653a...'으로 하위 실행"
       ↓
   ... 295번 반복 ...
       ↓
5. SELECT STATEMENT: 295개 행을 클라이언트에 반환
```

10.4 Row Source의 인터페이스

모든 Row Source는 동일한 인터페이스를 가진다. 이것이 트리 구조가 가능한 이유다.

```
[Row Source 인터페이스]

open()   → 초기화
fetch()  → 다음 행 반환 (없으면 EOF)
close()  → 리소스 해제
```

부모-자식 간 상호작용:

```
[부모 Row Source: INLIST ITERATOR]
    │
    │  child.open()
    │  while (row = child.fetch()) != EOF:
    │      parent에게 row 전달
    │  child.close()
    │
    ↓
[자식 Row Source: TABLE ACCESS BY INDEX ROWID]
    │
    │  child.open()
    │  while (rowid = child.fetch()) != EOF:
    │      row = table_block.get(rowid)
    │      return row
    │  child.close()
    │
    ↓
[손자 Row Source: INDEX UNIQUE SCAN]
```

INLIST ITERATOR는 IN 절의 각 값마다 이 과정을 반복한다. 값을 바꿀 때 하위 Row Source를 다시 open/fetch/close하는 것이 아니라, 바인드 변수만 교체하고 fetch를 계속한다. 이것이 INLIST ITERATOR가 효율적인 이유다.

10.5 CONCATENATION과의 비교

CONCATENATION은 제어 노드지만 INLIST ITERATOR와 다르게 여러 개의 독립적인 자식을 가진다.

```
[INLIST ITERATOR 구조]
INLIST ITERATOR
    └── 단일 자식 (값만 교체하며 반복)

[CONCATENATION 구조]
CONCATENATION
    ├── 자식 1 (첫 번째 IN 절)
    └── 자식 2 (두 번째 IN 절)
```

CONCATENATION의 실행:

```
1. 자식 1의 open() → fetch() 반복 → close()
2. 자식 2의 open() → fetch() 반복 → close()
3. 두 결과를 순차적으로 상위에 전달
```

각 자식이 독립적인 Row Source이므로, 각각 별도의 메모리와 상태를 유지한다. 이것이 CONCATENATION의 오버헤드 원인이다.

10.6 계층별 역할 요약

계층	역할	캡처 실행 계획에서
루트	결과 반환	SELECT STATEMENT
제어	실행 흐름 제어 (반복/분기)	INLIST ITERATOR
변환	데이터 형태 변환	TABLE ACCESS BY INDEX ROWID
리프	데이터 소스 (실제 읽기)	INDEX UNIQUE SCAN

---

11. ROWID 개념

11.1 정의

ROWID는 Oracle 테이블에서 각 행의 물리적 저장 위치를 나타내는 주소다. 파일 시스템에서 파일 경로가 파일의 위치를 가리키듯, ROWID는 테이블 내 특정 행의 위치를 가리킨다.

11.2 ROWID의 구조

ROWID는 18자리 Base64 인코딩 문자열이다.

```
예시: AAAHG7AAEAAAAJkAAB
      ──────  ──  ──────  ──
        ①     ②     ③     ④
```

구성 요소	길이	의미
① 데이터 오브젝트 번호	6자리	어떤 테이블(세그먼트)인지
② 데이터 파일 번호	3자리	어떤 데이터 파일인지
③ 블록 번호	6자리	파일 내 어떤 블록인지
④ 행 번호	3자리	블록 내 몇 번째 행인지

11.3 물리적 저장 구조와의 관계

```
[Oracle 물리적 저장 구조]

데이터베이스
  └── 테이블스페이스
        └── 데이터 파일 (.dbf)  ← ② 데이터 파일 번호
              └── 블록 (8KB 단위)  ← ③ 블록 번호
                    └── 행 (Row)  ← ④ 행 번호
```

ROWID가 AAAHG7AAEAAAAJkAAB라면:

```
1. AAAHG7 → REVIEW_STATISTICS 테이블
2. AAE    → 데이터 파일 5번
3. AAAAJk → 블록 번호 612
4. AAB    → 블록 내 2번째 행
```

Oracle은 이 정보로 디스크의 정확한 위치에 직접 접근할 수 있다.

11.4 인덱스와 ROWID의 관계

인덱스는 키 값과 ROWID의 매핑을 저장한다.

```
[PK_REVIEW_STATISTICS 인덱스 내부 구조]

키 값 (PROPERTY_ID)                          | ROWID
─────────────────────────────────────────────┼──────────────────
'00002ba8f5f4651131019bf9ff57b5d'            | AAAHG7AAEAAAAJkAAA
'0171653ab52177079476c7449397d086'           | AAAHG7AAEAAAAJkAAB
'02e275042695686587bf78d99897327'            | AAAHG7AAEAAAAJkAAC
...                                          | ...
```

11.5 INDEX SCAN → TABLE ACCESS BY INDEX ROWID 흐름

캡처한 실행 계획의 동작:

```
[1단계: INDEX UNIQUE SCAN]

입력: PROPERTY_ID = '00002ba8f5f4651131019bf9ff57b5d'

인덱스 B-tree 탐색:
  루트 블록 → 브랜치 블록 → 리프 블록

리프 블록에서 발견:
  키: '00002ba8f5f4651131019bf9ff57b5d'
  ROWID: AAAHG7AAEAAAAJkAAA

출력: ROWID 'AAAHG7AAEAAAAJkAAA'


[2단계: TABLE ACCESS BY INDEX ROWID]

입력: ROWID = 'AAAHG7AAEAAAAJkAAA'

ROWID 해석:
  데이터 파일 5번, 블록 612, 행 1번

디스크 접근:
  해당 블록을 Buffer Cache로 로드 (또는 캐시에서 조회)
  블록 내 1번 행 데이터 추출

출력: (PROPERTY_ID, AVG_RATING, LAST_CALCED, ...)
```

11.6 왜 ROWID가 빠른가

TABLE ACCESS FULL의 경우

```
테이블의 모든 블록을 순차적으로 읽음
→ 52,020행이면 수백~수천 블록 읽기
→ 각 행마다 WHERE 조건 검사
```

TABLE ACCESS BY INDEX ROWID의 경우

```
ROWID가 가리키는 정확한 블록 1개만 읽음
→ 블록 내 정확한 행 위치로 직접 접근
→ WHERE 조건 검사 불필요 (이미 인덱스에서 필터링됨)
```

ROWID는 O(1) 접근을 가능하게 한다. 테이블 크기와 무관하게 단일 블록 I/O로 행에 접근한다.

11.7 ROWID 직접 조회

REVIEW_STATISTICS 테이블에서 ROWID를 확인할 수 있다:

```
SELECT ROWID, PROPERTY_ID, AVG_RATING
FROM REVIEW_STATISTICS
WHERE ROWNUM <= 5;
```

```
ROWID              | PROPERTY_ID                          | AVG_RATING
-------------------+--------------------------------------+-----------
AAAHG7AAEAAAAJkAAA | 00002ba8f5f4651131019bf9ff57b5d | 4.2
AAAHG7AAEAAAAJkAAB | 0171653ab52177079476c7449397d086 | 3.8
...
```

11.8 ROWID 요약

항목	설명
ROWID란	테이블 내 행의 물리적 저장 위치 주소
구성	오브젝트 번호 + 파일 번호 + 블록 번호 + 행 번호
인덱스에서의 역할	키 값 → ROWID 매핑 저장
성능 의의	ROWID로 직접 접근 시 O(1) 블록 I/O

---

12. Table 7-1: PLAN_TABLE 주요 컬럼

12.1 COST

정의

Optimizer가 해당 Operation을 수행하는 데 필요한 상대적 비용 추정치다. 단위가 없으며, 절대적인 시간이나 리소스를 나타내지 않는다.

계산 요소

```
COST = (Single Block I/O 횟수 × 1)
     + (Multi Block I/O 횟수 × 가중치)
     + (CPU 연산 비용)
```

Oracle Optimizer는 내부적으로 I/O 비용과 CPU 비용을 합산하여 COST를 산출한다.

캡처한 실행 계획에서의 COST

Operation	Cost
SELECT STATEMENT	266
INLIST ITERATOR	
TABLE ACCESS BY INDEX ROWID	266
INDEX UNIQUE SCAN	222

해석: - INDEX UNIQUE SCAN 완료 시점 누적 비용: 222 - TABLE ACCESS BY INDEX ROWID 완료 시점 누적 비용: 266 (증분: 44) - 인덱스 스캔이 전체 비용의 83%를 차지

활용 방법

COST는 동일 쿼리의 서로 다른 실행 계획을 비교할 때 사용한다.

```
[실행 계획 A] COST: 266  ← Optimizer가 선택
[실행 계획 B] COST: 1,200

→ Optimizer는 COST가 낮은 A를 선택함
```

Wherehouse 프로젝트에서는 Chunk 방식 vs Bulk Fetch 방식의 COST를 비교하여 어느 쪽이 Optimizer 관점에서 효율적인지 판단할 수 있다.

12.2 CARDINALITY (Rows)

정의

해당 Operation이 반환할 것으로 예상되는 행 수다. Optimizer가 테이블 통계를 기반으로 추정한다.

캡처한 실행 계획에서의 CARDINALITY

Operation	Cardinality
SELECT STATEMENT	295
INLIST ITERATOR	
TABLE ACCESS BY INDEX ROWID	295
INDEX UNIQUE SCAN	295

해석: - IN 절에 295개 값이 있음 - Optimizer가 295행 반환을 예상 - 실제 반환 행 수와 일치하면 통계가 정확한 것

통계 정확성 판단

```
[정확한 통계]
예상 CARDINALITY: 295
실제 반환 행 수: 295
→ 일치. Optimizer가 올바른 판단을 함

[부정확한 통계]
예상 CARDINALITY: 295
실제 반환 행 수: 52,000
→ 불일치. Optimizer가 잘못된 실행 계획을 선택했을 가능성
```

확인 방법

실제 반환 행 수를 확인하려면 DBMS_XPLAN.DISPLAY_CURSOR에 ALLSTATS LAST 옵션을 사용한다:

```
SELECT /*+ GATHER_PLAN_STATISTICS */ PROPERTY_ID, AVG_RATING, ...
FROM REVIEW_STATISTICS
WHERE PROPERTY_ID IN (...);

SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(NULL, NULL, 'ALLSTATS LAST'));
```

이렇게 하면 E-Rows(예상)와 A-Rows(실제)를 비교할 수 있다.

12.3 ACCESS_PREDICATES

정의

인덱스 접근 조건이다. 인덱스에서 어떤 범위의 키를 스캔할지 결정하는 조건이다.

캡처한 실행 계획에서의 ACCESS_PREDICATES

이미지에서 Access Predicates 하위에 OR로 연결된 PROPERTY_ID 조건들이 표시되어 있다:

```
Access Predicates
  └── OR
        ├── PROPERTY_ID='00002ba8f5f4651131019bf9ff57b5d'
        ├── PROPERTY_ID='0171653ab52177079476c7449397d086'
        ├── PROPERTY_ID='02e275042695686587bf78d99897327'
        ... (295개)
```

의미: 이 조건들이 인덱스 스캔의 **시작점과 끝점**을 결정한다. 인덱스 B-tree에서 해당 키 값을 직접 탐색한다.

ACCESS_PREDICATES의 효율성

ACCESS_PREDICATES에 조건이 있으면 **인덱스를 효율적으로 사용**하고 있다는 의미다.

```
[효율적인 경우]
ACCESS_PREDICATES: PROPERTY_ID IN (...)
→ 인덱스에서 해당 값들만 탐색

[비효율적인 경우]
ACCESS_PREDICATES: (없음)
FILTER_PREDICATES: PROPERTY_ID IN (...)
→ 전체 스캔 후 필터링
```

12.4 FILTER_PREDICATES

정의

인덱스 접근 후 **추가 필터링 조건**이다. 인덱스나 테이블에서 읽어온 행 중 조건에 맞지 않는 행을 제거한다.

ACCESS_PREDICATES vs FILTER_PREDICATES

```
SELECT * FROM REVIEW_STATISTICS
WHERE PROPERTY_ID = 'abc123'     -- 인덱스 컬럼
  AND REVIEW_COUNT > 10;         -- 비인덱스 컬럼
```

실행 계획:

```
TABLE ACCESS BY INDEX ROWID
  Access Predicates: PROPERTY_ID = 'abc123'    ← 인덱스로 접근
  Filter Predicates: REVIEW_COUNT > 10         ← 접근 후 필터링
```

처리 흐름:

```
1. INDEX SCAN: PROPERTY_ID = 'abc123'인 ROWID 탐색
       ↓
2. TABLE ACCESS: ROWID로 행 접근
       ↓
3. FILTER: REVIEW_COUNT > 10 검사
       ↓ (조건 만족 시)
4. 결과 반환
```

성능 영향

조건 위치	성능 영향
ACCESS_PREDICATES	인덱스에서 필요한 행만 접근. 효율적
FILTER_PREDICATES	접근 후 필터링. 불필요한 I/O 발생 가능

예시:

```
ACCESS_PREDICATES로 100행 접근 → FILTER로 10행 반환
→ 90행은 불필요하게 읽음

ACCESS_PREDICATES로 10행 접근 → FILTER 없음
→ 필요한 행만 읽음
```

캡처한 실행 계획에서

FILTER_PREDICATES가 표시되지 않았다. 이것은 **모든 조건이 인덱스로 처리**되었음을 의미한다. IN 절의 모든 PROPERTY_ID 조건이 ACCESS_PREDICATES로 처리되어 효율적이다.

12.5 Wherehouse 프로젝트에서의 활용

COST 비교 시나리오

```
[295개 IN 절 - 단일 INLIST ITERATOR]
COST: 266

[1660개 IN 절 - CONCATENATION 2분기]
COST: ???  ← 이 값을 캡처하여 비교

→ CONCATENATION의 COST가 더 높으면 비효율성의 정량적 증거
```

CARDINALITY 검증

```
[예상]
CARDINALITY: 295

[실제 확인 방법]
SELECT COUNT(*) FROM REVIEW_STATISTICS
WHERE PROPERTY_ID IN (...295개...);

→ 결과가 295면 통계 정확
→ 결과가 다르면 통계 갱신 필요
```

ACCESS_PREDICATES 확인

1000개 초과 IN 절의 실행 계획에서 ACCESS_PREDICATES가 어떻게 나타나는지 확인해야 한다. CONCATENATION의 각 분기마다 별도의 ACCESS_PREDICATES가 있을 것이다.

12.6 PLAN_TABLE 컬럼 요약

컬럼	의미	Wherehouse 프로젝트 활용
COST	상대적 비용 추정치	Chunk vs Bulk Fetch 비용 비교
CARDINALITY	예상 반환 행 수	통계 정확성 검증
ACCESS_PREDICATES	인덱스 접근 조건	IN 절이 인덱스로 처리되는지 확인
FILTER_PREDICATES	추가 필터링 조건	불필요한 I/O 발생 여부 판단

---

13. Access Predicates에서 IN 절이 OR로 표시되는 이유

13.1 핵심 답변

앞서 설명한 **"IN 절은 논리적으로 OR 조건과 동치"**이기 때문이다. Oracle 내부에서 IN 절을 해석할 때 OR 조건으로 변환하여 표현한다.

13.2 SQL 작성 vs Oracle 내부 표현

```
[사용자가 작성한 SQL]
WHERE PROPERTY_ID IN ('00002ba8...', '0171653a...', '02e27504...')

[Oracle 내부 해석]
WHERE PROPERTY_ID = '00002ba8...'
   OR PROPERTY_ID = '0171653a...'
   OR PROPERTY_ID = '02e27504...'
```

Access Predicates는 Oracle이 내부적으로 해석한 형태를 보여준다. IN 절로 작성했더라도 OR로 표시된다.

13.3 중요한 구분

구분	설명
Access Predicates의 OR 표시	단순 표현 방식. 실행 계획 구조에 영향 없음
CONCATENATION의 OR 분기	실제 실행 계획 분기. 별도 Row Source 생성

캡처한 실행 계획에서:

```
SELECT STATEMENT
  INLIST ITERATOR          ← 단일 제어 노드
    TABLE ACCESS BY INDEX ROWID
      INDEX UNIQUE SCAN
        Access Predicates: OR (...)  ← 표현만 OR, 분기 아님
```

CONCATENATION이 없다. INLIST ITERATOR가 단일 Row Source로 295개 값을 효율적으로 처리한다. Access Predicates의 OR는 표현일 뿐, 실행 계획이 분기된 것이 아니다.

13.4 비교: CONCATENATION이 발생하는 경우

1000개 초과 IN 절에서 CONCATENATION이 발생하면:

```
SELECT STATEMENT
  CONCATENATION              ← 실제 분기
    INLIST ITERATOR          ← 분기 1
      TABLE ACCESS BY INDEX ROWID
        INDEX UNIQUE SCAN
          Access Predicates: OR (1000개)
    INLIST ITERATOR          ← 분기 2
      TABLE ACCESS BY INDEX ROWID
        INDEX UNIQUE SCAN
          Access Predicates: OR (660개)
```

이 경우는 실행 계획 자체가 2개 분기로 나뉜다. 캡처한 실행 계획과 다르다.

13.5 실제 실행 방식 정리

논리적으로는 OR와 동치이지만, 실행 방식은 다르다

구분	내용
논리적 해석	IN 절을 OR 조건으로 해석 (Access Predicates 표시)
실행 방식	INLIST ITERATOR가 결정 (실제 처리 방식)

Access Predicates의 OR는 "의미 표현"

Oracle이 "이 295개 값 중 하나와 일치하는 행을 찾아야 한다"는 논리적 의미를 OR로 표현한 것이다. 실제로 OR 연산을 295번 순차 평가하는 것이 아니다.

실제 실행 방식은 INLIST ITERATOR가 결정

```
INLIST ITERATOR
  └── INDEX UNIQUE SCAN (반복 호출)
```

INLIST ITERATOR의 실행 방식:

```
for each value in IN절_값_목록:
    INDEX UNIQUE SCAN(value)  ← 인덱스 B-tree 탐색
    TABLE ACCESS BY ROWID     ← 행 접근
```

이것은 OR 조건을 순차 평가하는 것과 다르다. 인덱스 구조를 활용한 최적화된 반복 접근이다.

만약 실제로 OR 연산처럼 수행된다면

```
-- 비효율적인 OR 평가 방식 (실제로 이렇게 동작하지 않음)
for each row in TABLE:
    if (row.PROPERTY_ID = 'val1' OR row.PROPERTY_ID = 'val2' OR ... 295개):
        return row
```

이 방식이라면 TABLE ACCESS FULL이 나타나고, INLIST ITERATOR가 나타나지 않는다.

13.6 결론

Access Predicates의 OR 표시는 조건의 논리적 의미를 나타낸다. 실제 실행은 INLIST ITERATOR가 인덱스를 반복 탐색하는 방식으로 수행된다. OR 연산을 순차 평가하는 것이 아니다.

---

14. INDEX RANGE SCAN

14.1 정의

INDEX RANGE SCAN은 인덱스에서 특정 범위의 키 값을 탐색하는 Operation이다. 단일 값이 아닌 여러 값을 반환할 수 있다.

14.2 INDEX UNIQUE SCAN과의 비교

캡처한 실행 계획에서 INDEX UNIQUE SCAN이 나타났다:

```
INDEX    PK_REVIEW_STATISTICS    UNIQUE SCAN    295    222
```

구분	INDEX UNIQUE SCAN	INDEX RANGE SCAN
사용 조건	PK 또는 Unique 인덱스에서 등호(=) 조건	범위 조건 또는 Non-unique 인덱스
반환 행 수	최대 1행 (키당)	0행 ~ N행
탐색 방식	B-tree에서 정확히 일치하는 키 1개 탐색	B-tree에서 시작점 찾고 범위 내 키들을 순차 스캔

14.3 INDEX RANGE SCAN이 발생하는 조건

범위 조건 사용 시

```
SELECT * FROM REVIEW_STATISTICS
WHERE PROPERTY_ID >= 'a' AND PROPERTY_ID < 'b';
```

```
INDEX RANGE SCAN (PK_REVIEW_STATISTICS)
  Access Predicates: PROPERTY_ID >= 'a' AND PROPERTY_ID < 'b'
```

Non-unique 인덱스에서 등호 조건 사용 시

```
-- AVG_RATING에 Non-unique 인덱스가 있다고 가정
CREATE INDEX IDX_AVG_RATING ON REVIEW_STATISTICS(AVG_RATING);

SELECT * FROM REVIEW_STATISTICS
WHERE AVG_RATING = 4.5;
```

```
INDEX RANGE SCAN (IDX_AVG_RATING)
  Access Predicates: AVG_RATING = 4.5
```

AVG_RATING = 4.5인 행이 여러 개 존재할 수 있으므로 RANGE SCAN이다.

LIKE 조건 (앞부분 고정)

```
SELECT * FROM REVIEW_STATISTICS
WHERE PROPERTY_ID LIKE 'abc%';
```

```
INDEX RANGE SCAN (PK_REVIEW_STATISTICS)
  Access Predicates: PROPERTY_ID LIKE 'abc%'
```

14.4 B-tree에서의 동작 차이

INDEX UNIQUE SCAN

```
[B-tree 탐색]
루트 → 브랜치 → 리프 블록에서 'abc123' 키 발견
→ ROWID 1개 반환
→ 탐색 종료
```

INDEX RANGE SCAN

```
[B-tree 탐색]
루트 → 브랜치 → 리프 블록에서 시작점 'a' 발견
→ 'a'의 ROWID 반환
→ 다음 키 'a001'의 ROWID 반환
→ 다음 키 'a002'의 ROWID 반환
→ ... (리프 블록 간 링크를 따라 순차 스캔)
→ 'b' 도달 시 탐색 종료
```

INDEX RANGE SCAN은 리프 블록 간 연결 리스트를 따라 이동하며 여러 키를 읽는다.

14.5 Wherehouse 프로젝트에서의 의미

현재 REVIEW_STATISTICS 쿼리는 PK(PROPERTY_ID)에 대한 IN 절 조회이므로 INDEX UNIQUE SCAN이 나타난다. 만약 다음과 같은 쿼리를 사용한다면 INDEX RANGE SCAN이 나타날 것이다:

```
-- 특정 평점 이상의 매물 조회
SELECT * FROM REVIEW_STATISTICS
WHERE AVG_RATING >= 4.0;
```

이 경우 AVG_RATING 컬럼에 인덱스가 없다면 TABLE ACCESS FULL이 발생한다.

---

15. TABLE ACCESS FULL

15.1 정의

TABLE ACCESS FULL은 테이블의 모든 블록을 순차적으로 읽는 Operation이다. 인덱스를 사용하지 않는다.

15.2 발생 조건

인덱스가 없는 컬럼에 조건 사용

```
SELECT * FROM REVIEW_STATISTICS
WHERE REVIEW_COUNT > 100;
```

REVIEW_COUNT에 인덱스가 없으면:

```
TABLE ACCESS FULL (REVIEW_STATISTICS)
  Filter Predicates: REVIEW_COUNT > 100
```

인덱스가 있어도 Optimizer가 Full Scan이 효율적이라 판단

```
SELECT * FROM REVIEW_STATISTICS
WHERE AVG_RATING > 0;  -- 대부분의 행이 조건 만족
```

반환 행이 테이블의 대부분이면 인덱스 사용보다 Full Scan이 효율적이다.

인덱스 컬럼에 함수 적용

```
SELECT * FROM REVIEW_STATISTICS
WHERE UPPER(PROPERTY_ID) = 'ABC123';
```

```
TABLE ACCESS FULL (REVIEW_STATISTICS)
  Filter Predicates: UPPER(PROPERTY_ID) = 'ABC123'
```

인덱스는 원본 컬럼 값으로 구성되어 있으므로, 함수 적용 시 인덱스를 사용할 수 없다.

15.3 동작 방식

```
[TABLE ACCESS FULL]

1. 테이블의 첫 번째 블록부터 시작
2. 블록 내 모든 행을 읽음
3. 각 행에 대해 WHERE 조건 검사 (Filter Predicates)
4. 조건 만족 시 결과에 포함
5. 다음 블록으로 이동
6. 마지막 블록까지 반복

I/O 패턴: Multi-block Read (한 번에 여러 블록 읽기)
```

15.4 비효율성의 원인

REVIEW_STATISTICS 테이블(52,020행)에서 295개 행을 찾는 상황:

INDEX UNIQUE SCAN 사용 시

```
필요한 블록 읽기: 295개 행 × (인덱스 블록 + 테이블 블록)
≈ 295 × 4 = 약 1,180 블록 I/O (최악의 경우)
실제로는 Buffer Cache 히트로 훨씬 적음
```

TABLE ACCESS FULL 사용 시

```
필요한 블록 읽기: 테이블 전체 블록
52,020행 ÷ 행당 ~100바이트 ÷ 8KB 블록 ≈ 약 650 블록
그러나 모든 블록을 읽고 52,020행 모두에 WHERE 조건 검사
```

행 수가 적을 때는 INDEX SCAN이 효율적이고, 행 수가 많을 때는 TABLE ACCESS FULL이 효율적일 수 있다. Optimizer가 통계 기반으로 판단한다.

15.5 TABLE ACCESS FULL이 오히려 효율적인 경우

반환 행이 테이블의 상당 부분일 때

```
SELECT * FROM REVIEW_STATISTICS;  -- 전체 조회
```

전체 조회에서 인덱스를 사용하면 오히려 비효율적이다.

테이블이 매우 작을 때

```
-- 100행 미만의 작은 테이블
SELECT * FROM CODE_TABLE WHERE CODE_TYPE = 'STATUS';
```

작은 테이블은 몇 개 블록만 읽으면 되므로 인덱스 오버헤드가 불필요하다.

Parallel Query 사용 시

```
SELECT /*+ PARALLEL(r, 4) */ * FROM REVIEW_STATISTICS r
WHERE REVIEW_COUNT > 100;
```

여러 프로세스가 테이블을 분할하여 동시에 스캔한다.

15.6 Wherehouse 프로젝트에서 TABLE ACCESS FULL이 나타나면

문제 상황: IN 절 조회에서 TABLE ACCESS FULL이 나타남

```
SELECT STATEMENT
  TABLE ACCESS FULL (REVIEW_STATISTICS)
    Filter Predicates: PROPERTY_ID IN (...)
```

원인 가능성: 1. 인덱스가 없거나 삭제됨 2. 통계 정보가 오래됨 3. IN 절 값이 너무 많아 Optimizer가 Full Scan 선택

확인 방법:

```
-- 인덱스 존재 확인
SELECT index_name, column_name
FROM user_ind_columns
WHERE table_name = 'REVIEW_STATISTICS';

-- 통계 정보 확인
SELECT table_name, num_rows, last_analyzed
FROM user_tables
WHERE table_name = 'REVIEW_STATISTICS';
```

15.7 스캔 방식 요약

Operation	발생 조건	Wherehouse 프로젝트 관련성
INDEX UNIQUE SCAN	PK/Unique 인덱스 + 등호 조건	현재 실행 계획에서 사용 중 (정상)
INDEX RANGE SCAN	범위 조건 또는 Non-unique 인덱스	범위 조회 추가 시 나타날 수 있음
TABLE ACCESS FULL	인덱스 없음 또는 대량 반환	IN 절 조회에서 나타나면 비효율적
