

Oracle Buffer I/O 학습 문서 개요

문서 정보

항목	내용
학습 대상	Oracle 19c Database Concepts - Memory Architecture - Database Buffer Cache - Buffer I/O
공식 문서 URL	https://docs.oracle.com/en/database/oracle/oracle-database/19/cncpt/memory-architecture.html
학습 목적	Wherehouse 프로젝트 N+1 최적화 경험에 대한 DBMS 레벨 병목 판단 근거 확보
학습 관점	백엔드 개발자 관점, DBA 수준 튜닝은 제외

1. Buffer I/O 개요 분석

1.1 공식 문서 원문

"A logical I/O, also known as a buffer I/O, refers to reads and writes of buffers in the buffer cache."

"When a requested buffer is not found in memory, the database performs a physical I/O to copy the buffer from either the flash cache or disk into memory. The database then performs a logical I/O to read the cached buffer."

1.2 공식 문서 하위 섹션 구조

```
Buffer I/O
└── Buffer Replacement Algorithms
    To make buffer access efficient, the database must decide which buffers
    to cache in memory, and which to access from disk.

└── Buffer Writes
    The database writer (DBW) process periodically writes cold, dirty buffers to disk.

└── Buffer Reads
    When the number of unused buffers is low, the database must remove
    buffers from the buffer cache.

└── Buffer Touch Counts
    The database measures the frequency of access of buffers on the LRU list
    using a touch count. This mechanism enables the database to increment a
    counter when a buffer is pinned instead of constantly shuffling buffers
    on the LRU list.
```

2. Why: Logical I/O와 Physical I/O를 구분하는 설계 의도

2.1 Oracle이 이 두 개념을 분리한 이유

Oracle이 Logical I/O와 Physical I/O를 명시적으로 분리한 설계 의도는 성능 병목 지점의 계층적 식별에 있다.



2.2 Logical I/O와 Physical I/O의 특성 비교

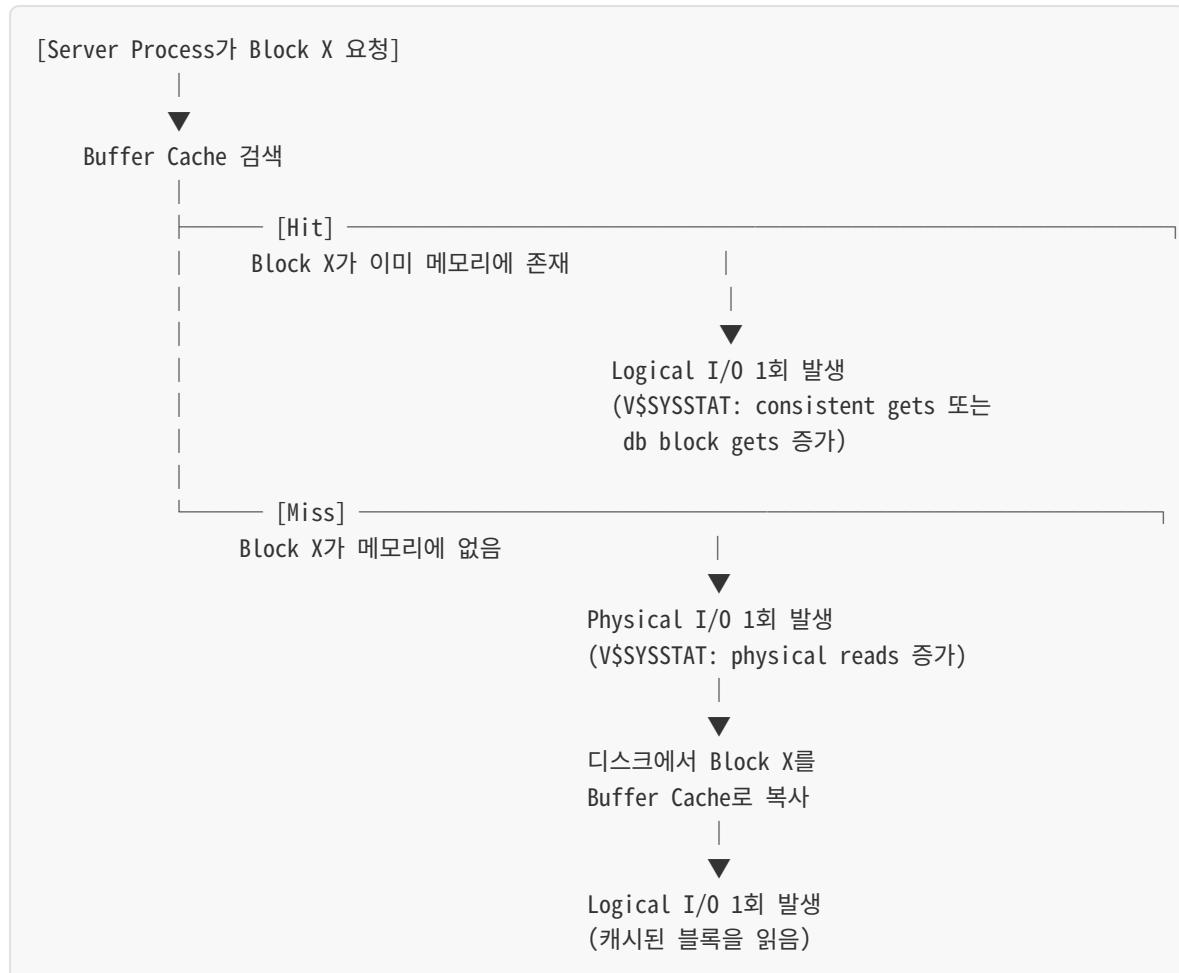
구분	Logical I/O	Physical I/O
접근 대상	Buffer Cache (메모리)	Data File (디스크)
지연 시간	μ s (마이크로초) 단위	ms (밀리초) 단위
비용 차이	기준 = 1	약 100~10,000배
발생 조건	모든 블록 접근 시	Cache Miss 시에만

2.3 구분의 실용적 의의

Oracle이 이 구분을 명시적으로 제공하는 이유는, "느린 쿼리의 원인이 메모리 접근량 자체인지, 디스크 접근 때문인지"를 분리해서 진단할 수 있게 하기 위함이다.

3. How: Cache Hit과 Cache Miss 처리 흐름

3.1 공식 문서 두 번째 문장의 단계별 분해



3.2 핵심 포인트

Physical I/O가 발생하면 반드시 Logical I/O도 뒤따른다. 디스크에서 읽어온 블록을 Buffer Cache에 적재한 후, 그 캐시된 블록을 읽는 것이 Logical I/O이기 때문이다.

따라서 다음이 항상 성립한다:

$$\text{Logical I/O} \geq \text{Physical I/O} \text{ (항상 성립)}$$

$$\text{Logical I/O} - \text{Physical I/O} = \text{Cache Hit으로 처리된 블록 수}$$

4. So What: V\$SQL에서의 Buffer I/O 지표

4.1 V\$SQL 컬럼과 Buffer I/O 용어 맵핑

V\$SQL 컬럼	의미	Buffer I/O 용어 맵핑
buffer_gets	Buffer Cache 접근 횟수	Logical I/O
disk_reads	디스크 읽기 횟수	Physical I/O

4.2 V\$SQL 기본 조회 쿼리

```

SELECT
    sql_id,
    sql_text,
    executions,
    buffer_gets,      -- Logical I/O 총합
    disk_reads,       -- Physical I/O 총합
    buffer_gets / NULLIF(executions, 0) AS avg_logical_io,
    disk_reads / NULLIF(executions, 0) AS avg_physical_io
FROM v$sql
WHERE sql_text LIKE '%REVIEW_STATISTICS%';

```

5. 피드백 지적사항과의 연결: DBMS 병목 판단 로직

5.1 피드백 지적사항 3번

"DBMS 병목"이라고 판단한 근거가 문서에 명시되어 있지 않습니다.

5.2 Buffer I/O 개념을 활용한 판단 로직

[DBMS 병목 판단 로직]

```

IF (avg_physical_io / avg_logical_io) > 0.1 THEN
    → Physical I/O 비율 10% 초과
    → Buffer Cache 크기 부족 또는 대량 데이터 접근
    → "DBMS 레벨 I/O 병목" 판단 가능

ELSE IF avg_logical_io가 비정상적으로 높음 THEN
    → Physical I/O는 적지만 Logical I/O가 과다
    → 실행 계획 비효율 (불필요한 블록 접근)
    → "DBMS 레벨 CPU/메모리 병목" 판단 가능

ELSE
    → DBMS 자체는 효율적으로 동작
    → 병목은 다른 계층 (네트워크, Connection Pool 등)

```

5.3 프로젝트 테스트 데이터 해석

1차 테스트 결과 (이미지 1에서 확인):

항목	값
RDB 시간 (평균)	343ms
RDB 시간 비율	6.2%
총 소요 시간	5,531ms

이 수치가 의미하는 것:

전체 응답 시간 5,531ms 중:

- └── RDB 처리 시간: 343ms (6.2%) ← Buffer I/O + 네트워크 왕복 포함
- └── 나머지: 5,188ms (93.8%) ← Connection 경합, 애플리케이션 처리

결론: DBMS 자체의 Buffer I/O가 병목이 아님
 25번의 쿼리가 각각 빠르게 처리되더라도 (Logical I/O 효율적)
 누적된 네트워크 왕복 + Connection Holding이 실제 병목

6. Logical I/O가 "높다"는 판단의 근거

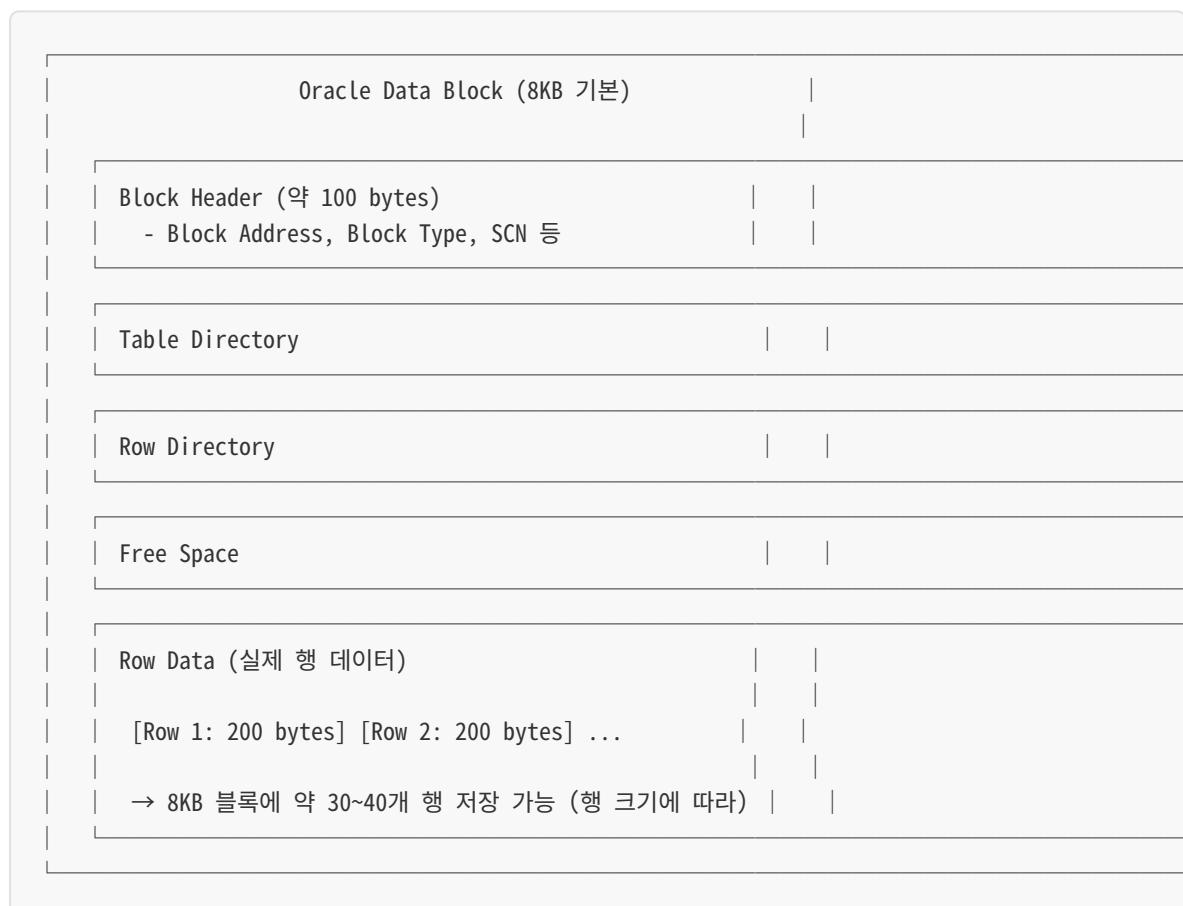
6.1 절대적 기준은 없다

특정 숫자(예: 10000)를 절대적 기준으로 삼을 수 없다. 실제 판단은 다음 두 가지 상대적 기준으로 이루어진다:

판단 기준 1: 반환 행 수 대비 블록 접근 수 (Buffer Gets per Row)
 판단 기준 2: 동일 쿼리의 Before/After 비교

7. Oracle 블록의 물리적 구조

7.1 Data Block 내부 구조



7.2 블록당 행 수 계산 예시

예시: REVIEW_STATISTICS 테이블
 - 행 하나의 평균 크기: 200 bytes
 - 블록 크기: 8KB (8,192 bytes)
 - 블록당 행 수: 약 35개 (헤더 등 오버헤드 제외)

→ 1,000개 행을 읽으려면 이론적으로 약 29개 블록 접근 필요
 → 8,660개 행을 읽으려면 이론적으로 약 248개 블록 접근 필요

8. 효율적인 쿼리 vs 비효율적인 쿼리

8.1 동일한 결과를 반환하는 두 쿼리의 비교

```
-- 시나리오: property_id로 8,660개 행 조회

-- [Case A] 효율적인 경우 - Index Range Scan
SELECT * FROM REVIEW_STATISTICS
WHERE PROPERTY_ID IN (?, ?, ..., ?); -- 8,660개 ID

-- V$SQL 결과
rows_processed: 8,660
buffer_gets: 350
buffer_gets_per_row: 0.04 ← 행당 0.04 블록 (인덱스 + 테이블 접근)

-- [Case B] 비효율적인 경우 - Full Table Scan
SELECT * FROM REVIEW_STATISTICS; -- 전체 테이블 스캔 후 필터링

-- V$SQL 결과
rows_processed: 8,660
buffer_gets: 15,000
buffer_gets_per_row: 1.73 ← 행당 1.73 블록 (불필요한 블록까지 모두 읽음)
```

8.2 Buffer Gets per Row 해석 기준

구분	buffer_gets / rows_processed	의미
< 1	효율적	인덱스 활용, 필요한 블록만 접근
1 ~ 5	보통	조인이나 정렬이 포함된 경우
> 10	비효율적 의심	불필요한 블록 접근, 실행 계획 검토 필요
> 100	심각한 비효율	Full Table Scan 또는 잘못된 인덱스 사용

9. Logical I/O가 높으면 왜 문제인가

9.1 "메모리 접근이니까 빠르지 않나?"에 대한 답변

Logical I/O도 공짜가 아니다. 각 Logical I/O마다 내부적으로 상당한 오버헤드가 발생한다.

9.2 Logical I/O 1회당 발생하는 내부 동작

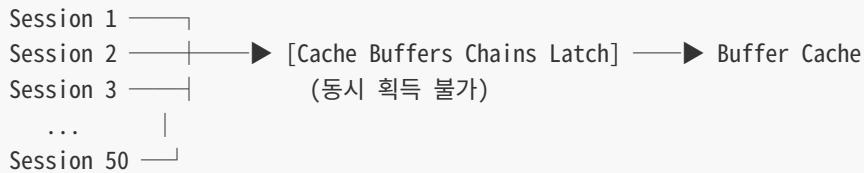
[Buffer Cache에서 블록 1개 읽기 - 내부 동작]

1. Hash Bucket 검색
 - | Block Address를 해시하여 해당 버킷 찾기
 - |
2. Cache Buffers Chains Latch 획득 ←———— 동시성 제어 지점
 - | 다른 세션과의 충돌 가능
 - |
3. Buffer Header 검색
 - | 해시 체인을 순회하며 원하는 블록 찾기
 - |
4. Buffer Pin 획득
 - | 해당 버퍼를 "사용 중"으로 표시
 - |
5. 블록 내용 읽기
 - | 실제 데이터 접근
 - |
6. Buffer Unpin
 - | 사용 완료 표시
 - |
7. Latch 해제

→ 블록 1개 읽는 데 약 0.01ms ($10\mu\text{s}$) 소요
 → 10,000 블록 = 100ms 순수 Logical I/O 시간
 → 100,000 블록 = 1,000ms (1초)

9.3 Latch Contention 문제

[동시 세션 50개가 Buffer Cache 접근 시]



Latch는 짧은 시간 점유하지만, 동시 요청이 많으면:

- Latch 획득 대기 발생
- V\$SYSTEM_EVENT의 'latch: cache buffers chains' 증가
- CPU 사용률은 높은데 처리량은 낮아지는 현상

10. 프로젝트 데이터에 Buffer I/O 개념 적용

10.1 N+1 패턴 (1차 테스트)에서의 Logical I/O 추정

시나리오:

- 25개 지역구, 각 지역구당 평균 346개 매물 (8,660 / 25)
- 매 쿼리당 REVIEW_STATISTICS 테이블 접근

추정 계산:

- 쿼리 1회당 반환 행: 346개
 - 행당 약 200 bytes → 블록당 약 35행
 - 쿼리 1회당 buffer_gets (Index Scan 가정): 약 10~15 블록
 - 25회 쿼리 총 buffer_gets: 250~375 블록
 - buffer_gets_per_row: 약 0.03~0.04
- Logical I/O 자체는 효율적
→ 문제는 Logical I/O 효율이 아니라 "25번 반복 실행"의 오버헤드

10.2 Bulk Fetch (2차 테스트)에서의 문제

시나리오:

- 8,660개 ID를 단일 IN 절로 조회
- Oracle IN 절 1000개 제한 → Hibernate가 OR로 분해

추정 계산:

- IN (...1000개...) OR IN (...1000개...) ... OR IN (...660개...)
- 실행 계획: CONCATENATION (9개 서브쿼리 결과 합침)

문제:

- 각 OR 절이 개별적으로 인덱스 스캔 수행
- 중복 블록 접근 가능성 (동일 블록을 여러 OR 절에서 반복 읽기)
- buffer_gets가 비정상적으로 증가할 수 있음

테스트 결과에서 확인된 현상:

- RDB 조회 시간: 343ms → 1,748ms (410% 증가)
- 이 증가분 중 일부는 Logical I/O 증가로 설명 가능

11. 실제 판단 방법: V\$SQL 상세 쿼리

11.1 SQL 효율성 판단 쿼리

```
-- 특정 SQL의 효율성 판단
SELECT
    sql_id,
    executions,
    rows_processed,
    buffer_gets,
    disk_reads,
    -- 핵심 지표
    ROUND(buffer_gets / NULLIF(rows_processed, 0), 2) AS gets_per_row,
    ROUND(buffer_gets / NULLIF(executions, 0), 2) AS gets_per_exec,
    -- CPU 시간 (Logical I/O의 실제 비용)
    ROUND(cpu_time / NULLIF(executions, 0) / 1000, 2) AS cpu_ms_per_exec
FROM v$sql
WHERE sql_text LIKE '%REVIEW_STATISTICS%'
ORDER BY buffer_gets DESC;
```

11.2 결과 해석 예시

11.2.1 컬럼 명세

컬럼명	데이터 타입	설명	누적 여부
sql_id	VARCHAR2(13)	SQL 문장의 고유 식별자. SQL 텍스트의 해시 값으로 생성되며, 동일한 SQL 텍스트는 항상 동일한 sql_id를 가짐	해당 없음 (식별자)
executions	NUMBER	해당 SQL이 Library Cache에 로드된 이후 실행된 총 횟수. Instance 재시작 또는 SQL이 Library Cache에서 age-out되면 0으로 초기화됨	누적값 (SQL 로드 시점 이후)
rows_processed	NUMBER	해당 SQL이 처리한 총 행 수. SELECT의 경우 반환된 행 수, DML의 경우 영향받은 행 수를 의미함	누적값 (SQL 로드 시점 이후)
buffer_gets	NUMBER	해당 SQL이 Buffer Cache에서 읽은 총 블록 수 (Logical I/O). consistent gets와 db block gets의 합계	누적값 (SQL 로드 시점 이후)
disk_reads	NUMBER	해당 SQL이 디스크에서 읽은 총 블록 수 (Physical I/O). Cache Miss로 인해 Data File에서 직접 읽은 횟수	누적값 (SQL 로드 시점 이후)
gets_per_row	NUMBER (계산)	행 하나를 반환하기 위해 접근한 평균 블록 수. buffer_gets / rows_processed로 계산. 값이 낮을수록 효율적	계산값
gets_per_exec	NUMBER (계산)	실행 1회당 평균 블록 접근 수. buffer_gets / executions로 계산. 쿼리의 평균 작업량을 나타냄	계산값
cpu_ms_per_exec	NUMBER (계산)	실행 1회당 평균 CPU 사용 시간(밀리초). cpu_time / executions / 1000으로 계산. Logical I/O 처리에 소요된 CPU 비용	계산값

11.2.2 결과 해석 테이블

sql_id	executions	rows_processed	buffer_gets	gets_per_row	판단
abc123	25	8,660	375	0.04	효율적: 인덱스를 활용하여 필요한 블록만 접근
def456	1	8,660	15,000	1.73	비효율 의심: 불필요한 블록 접근 발생, 실행 계획 검토 필요
ghi789	1	8,660	85,000	9.81	심각한 비효율: Full Table Scan 또는 잘못된 인덱스 사용 가능성

11.3 피드백 대응 시 주장 방법

피드백 대응 시 "DBMS 병목"을 주장하려면, 단순히 buffer_gets 절대값이 아니라 "행당 블록 접근 수가 비정상적으로 높다"는 근거를 제시해야 설득력이 있다.

12. 실측 근거 확보를 위한 Gap 분석

12.1 현재 학습으로 이해 가능한 영역

[Buffer I/O 섹션 학습 후 이해 가능한 것]

- Logical I/O와 Physical I/O의 개념적 구분
- Cache Hit/Miss의 성능 영향
- buffer_gets, disk_reads 지표의 "의미"
- 효율적/비효율적 쿼리의 판단 기준 (gets_per_row)

12.2 실제 디버깅에 추가로 필요한 지식

[실측 근거 확보를 위해 추가로 필요한 지식]

- V\$SQL 뷰의 구조와 각 컬럼의 정확한 의미
- SQL 식별 방법 (sql_id, sql_text 매칭)
- 통계 수집 시점과 누적 방식 이해
- 테스트 전후 Delta 측정 방법
- Hibernate가 생성한 SQL과 V\$SQL 매칭 방법
- 실행 계획(V\$SQL_PLAN)과 buffer_gets의 연관 분석

13. V\$SQL의 특성과 한계

13.1 V\$SQL의 본질적 특성

V\$SQL은 Library Cache에 캐싱된 SQL 커서의 **누적 통계**를 보여준다.

Library Cache		
SQL Cursor (sql_id: abc123)		
sql_text: SELECT * FROM REVIEW_STATISTICS...		
executions: 150 (누적 실행 횟수)		
buffer_gets: 4,500 (누적 Logical I/O)		
disk_reads: 45 (누적 Physical I/O)		
rows_processed: 12,000 (누적 처리 행 수)		
elapsed_time: 3,500,000 (누적 경과 시간, µs)		
cpu_time: 2,800,000 (누적 CPU 시간, µs)		
first_load_time: 2024-12-26 10:30:00		
last_active_time: 2024-12-26 15:45:00		
...		
SQL Cursor (sql_id: def456)		
...		

13.2 V\$SQL 통계값의 누적 기준

V\$SQL의 모든 통계 컬럼(executions, buffer_gets, disk_reads, rows_processed, elapsed_time, cpu_time 등)은 해당 SQL 커서가 Library Cache에 최초 로드된 시점부터 현재까지의 **누적값**이다.

[누적 기준점: SQL 커서의 Library Cache 로드 시점]

시점 T0: SQL 최초 실행

- Library Cache에 SQL 커서 생성 (`first_load_time = T0`)
- 모든 통계값 = 이 실행의 값

시점 T1: 동일 SQL 두 번째 실행

- 기존 커서 재사용
- 통계값 = T0 실행값 + T1 실행값 (누적)

시점 T2: 동일 SQL 세 번째 실행

- 기존 커서 재사용
- 통계값 = T0 + T1 + T2 실행값 (누적)

...

시점 Tn: SQL 커서가 Library Cache에서 age-out (메모리 부족 등)

- 커서 제거, 통계값 소멸

시점 Tn+1: 동일 SQL 다시 실행

- 새로운 커서 생성 (`first_load_time = Tn+1`)
- 통계값 = 이 실행의 값부터 다시 시작

[Instance 재시작 시]

- 모든 Library Cache 내용 소멸
- 모든 SQL 커서 제거
- 재시작 후 SQL 실행 시 새로운 커서부터 통계 시작

13.3 누적값의 실무적 의미

예시: V\$SQL에서 다음 결과를 조회했다고 가정

```
sql_id: abc123
executions: 150
buffer_gets: 4,500
```

이 수치의 의미:

- "abc123 SQL이 Library Cache에 로드된 이후 총 150번 실행되었다"
- "150번의 실행 동안 Buffer Cache에서 총 4,500개 블록을 읽었다"
- "평균적으로 1회 실행당 30개 블록을 읽었다" ($4,500 / 150 = 30$)

주의사항:

- 이 150번이 "오늘" 실행된 것인지, "지난 한 달간" 실행된 것인지 알 수 없음
- `first_load_time`과 `last_active_time`을 함께 확인해야 기간 추정 가능
- 특정 테스트의 정확한 측정을 위해서는 테스트 전후 스냅샷 비교 필요

13.4 V\$SQL로 확인 가능한 항목

항목	가능 여부	비고
SQL별 총 Logical I/O	가능	buffer_gets
SQL별 총 Physical I/O	가능	disk_reads
실행 횟수	가능	executions
평균 실행 시간	가능	elapsed_time / executions
행당 블록 접근 수	가능	buffer_gets / rows_processed
실행 계획 식별	가능	plan_hash_value로 V\$SQL_PLAN 조인

13.5 V\$SQL의 한계

한계 1: 누적값이므로 특정 테스트의 Delta 측정이 번거로움

테스트 전: executions = 100, buffer_gets = 3,000
 테스트 후: executions = 125, buffer_gets = 3,750

→ 테스트 중 발생량: executions = 25, buffer_gets = 750
 → 수동으로 전후 스냅샷 비교 필요

한계 2: Hibernate 동적 SQL과의 매칭 어려움

Hibernate가 생성하는 SQL:

```
SELECT ... FROM REVIEW_STATISTICS WHERE PROPERTY_ID IN (?, ?, ?, ...)
```

IN 절의 파라미터 "개수"가 다르면 다른 sql_id로 저장됨:

- IN (?, ?) → sql_id: aaa111
- IN (?, ?, ?) → sql_id: bbb222
- IN (?, ?, ?, ?) → sql_id: ccc333

→ 1차 테스트(N+1)에서 25개의 다른 sql_id가 생성될 수 있음
 → 2차 테스트(Bulk)에서 OR로 분해된 복잡한 SQL은 찾기 어려움
 → sql_text LIKE '%REVIEW_STATISTICS%'로 필터링 후 수동 분석 필요

한계 3: 바인드 변수 값은 확인 불가

V\$SQL은 SQL 텍스트만 저장, 실제 바인드 값은 V\$SQL_BIND_CAPTURE 참조 필요
 그러나 이 뷰는 샘플링 방식이라 모든 값을 보장하지 않음

14. 실제 측정 가능한 디버깅 시나리오

14.1 시나리오 1: N+1 패턴의 쿼리 횟수 및 효율성 확인

목표: 25번의 개별 쿼리가 실제로 발생하는지, 각 쿼리의 효율은 어떤지

방법:

1. 테스트 전 V\$SQL 스냅샷 저장
2. API 1회 호출 (N+1 발생)
3. 테스트 후 V\$SQL 스냅샷과 비교
4. REVIEW_STATISTICS 관련 SQL의 executions 증가량 확인

예상 결과:

- 유사한 형태의 SQL이 25개 존재 (IN 절 파라미터 개수 다름)
- 또는 동일 SQL이 executions += 25

확인 가능한 근거:

- "N+1로 인해 25번의 개별 쿼리 실행 확인"
- "쿼리당 평균 buffer_gets: X 블록"
- "총 buffer_gets: Y 블록"

14.2 시나리오 2: Bulk Fetch의 OR 분해 확인

목표: Hibernate가 IN 절을 OR로 분해했는지, 실행 계획이 비효율적인지

방법:

1. Hibernate SQL 로깅 활성화 (show-sql, format_sql)
2. API 호출하여 생성된 SQL 텍스트 확인
3. 해당 SQL의 sql_id로 V\$SQL 조회
4. plan_hash_value로 V\$SQL_PLAN 조인하여 실행 계획 확인

예상 결과:

- sql_text에 "OR" 연산자 포함 확인
- 실행 계획에 CONCATENATION 오퍼레이션 존재

확인 가능한 근거:

- "IN 절 1000개 초과 시 Hibernate가 OR로 분해"
- "실행 계획: CONCATENATION 발생으로 비효율"
- "buffer_gets: X (N+1 대비 Y% 증가)"

14.3 시나리오 3: Chunk 방식의 Soft Parse 효과 확인

목표: SQL 형태 고정으로 Library Cache 재사용이 발생하는지

방법:

1. V\$SYSSTAT에서 parse count (hard), parse count (total) 테스트 전 기록
2. Chunk 방식 API 호출 (9회 쿼리 예상)
3. 테스트 후 parse count 증가량 비교

예상 결과:

- parse count (hard) 증가: 1 (첫 실행만)
- parse count (total) 증가: 9 (Soft Parse 8회)

확인 가능한 근거:

- "Chunk 방식으로 Soft Parse 비율 89% 달성 (8/9)"
- "Hard Parse 1회만 발생, Library Cache 재사용 확인"

15. 추가 학습이 필요한 V\$ 뷰 목록

15.1 필수 학습 - 직접 사용할 뷰

V\$SQL	
- SQL별 실행 통계의 핵심 뷰	
- buffer_gets, disk_reads, executions, rows_processed	
- 학습 자료: Oracle 19c Database Reference	
- URL: docs.oracle.com/en/database/.../refrn/V-SQL.html	

V\$SQL_PLAN	
- SQL의 실행 계획 저장	
- sql_id, plan_hash_value로 V\$SQL과 조인	
- INLIST ITERATOR, CONCATENATION 등 오퍼레이션 확인	

V\$SYSSTAT	
- 시스템 전체 통계 (누적)	
- parse count (hard), parse count (total)	
- consistent gets, db block gets, physical reads	

V\$SESSTAT + V\$STATNAME	
- 세션별 통계 (V\$SYSSTAT의 세션 레벨 버전)	
- 특정 테스트 세션만 격리하여 측정 가능	

15.2 선택 학습 - 심화 분석 시 필요

V\$SQL_PLAN_STATISTICS_ALL	
- 실행 계획의 각 단계별 실제 통계	
- GATHER_PLAN_STATISTICS 힌트 사용 시 수집	
- 예상 행 수 vs 실제 행 수 비교 가능	
V\$SESSION_EVENT	
- 세션별 Wait Event 통계	
- buffer busy waits, db file sequential read 등	

16. 실행 가능한 디버깅 플로우 (로컬 환경 기준)

16.1 Phase 1: 환경 준비

```
# application.yml - Hibernate SQL 로깅 활성화

spring:
  jpa:
    show-sql: true
    properties:
      hibernate:
        format_sql: true

logging:
  level:
    org.hibernate.SQL: DEBUG
    org.hibernate.type.descriptor.sql.BasicBinder: TRACE
```

```
-- Oracle 세션 식별용 설정 (선택)
-- application.yml에서 HikariCP 설정

spring:
  datasource:
    hikari:
      connection-init-sql: >
        ALTER SESSION SET MODULE='WHEREHOUSE_TEST' ACTION='N+1_TEST'

-- → V$SESSION에서 MODULE, ACTION으로 필터링 가능
```

16.2 Phase 2: 베이스라인 측정

```
-- 테스트 전 시스템 통계 스냅샷
CREATE TABLE test_baseline AS
SELECT name, value FROM v$sysstat
WHERE name IN (
    'parse count (total)',
    'parse count (hard)',
    'consistent gets',
    'db block gets',
    'physical reads'
);

-- 테스트 전 SQL 통계 스냅샷
CREATE TABLE sql_baseline AS
SELECT sql_id, plan_hash_value, executions, buffer_gets,
       disk_reads, rows_processed, elapsed_time
FROM v$sql
WHERE UPPER(sql_text) LIKE '%REVIEW_STATISTICS%';
```

16.3 Phase 3: 테스트 실행

```
# Spring Boot API 호출
curl http://localhost:8080/api/charter/recommendations?districtCount=25
```

16.4 Phase 4: Delta 측정

```
-- 시스템 통계 변화량
SELECT
    b.name,
    a.value - b.value AS delta
FROM v$sysstat a
JOIN test_baseline b ON a.name = b.name
WHERE a.name IN (
    'parse count (total)',
    'parse count (hard)',
    'consistent gets',
    'db block gets',
    'physical reads'
);

-- SQL별 통계 변화량
SELECT
    a.sql_id,
    a.executions - NVL(b.executions, 0) AS exec_delta,
    a.buffer_gets - NVL(b.buffer_gets, 0) AS gets_delta,
    a.disk_reads - NVL(b.disk_reads, 0) AS reads_delta,
    SUBSTR(a.sql_text, 1, 100) AS sql_preview
FROM v$sql a
LEFT JOIN sql_baseline b ON a.sql_id = b.sql_id
WHERE UPPER(a.sql_text) LIKE '%REVIEW_STATISTICS%'
    AND (a.executions - NVL(b.executions, 0)) > 0
ORDER BY gets_delta DESC;
```

16.5 Phase 5: 실행 계획 확인

```
-- 특정 SQL의 실행 계획
SELECT
    operation,
    options,
    object_name,
    cardinality,
    cost,
    bytes
FROM v$sql_plan
WHERE sql_id = '&sql_id_from_phase4'
ORDER BY id;
```

16.6 Phase 6: 결과 문서화

수집된 데이터를 포트폴리오 형식으로 정리: - Before/After 수치 비교 표 - 실행 계획 캡처 - 판단 근거 서술

17. 학습 경로 요약 및 최종 판단

17.1 질문과 답변 정리

질문	답변
Buffer I/O 학습만으로 충분한가?	개념 이해는 가능하나, 실측 근거 확보는 불가
추가로 필요한 것은?	V\$SQL, V\$SYSSTAT, V\$SQL_PLAN 학습
로컬 환경에서 측정 가능한가?	DBA 권한 있으면 모두 가능
신뢰성 있는 근거 확보 가능한가?	위 플로우대로 수행하면 가능
예상 추가 학습 시간	V\$ 뷰 학습: 2-3시간, 실측 연습: 3-4시간

17.2 권장 학습 순서

1. Buffer I/O 섹션 완료 (현재)
2. V\$SQL Reference 문서 학습 (주요 컬럼 중심)
3. V\$SYSSTAT에서 Parse 관련 통계 학습
4. 로컬 환경에서 위 디버깅 플로우 1회 실습
5. 실습 결과를 포트폴리오 형식으로 문서화

이 경로를 따르면 피드백에서 요구한 "실측 데이터 기반 DBMS 병목 판단 근거"를 스스로 확보할 수 있다.

문서 이력

버전	일자	변경 내용
1.0	2024-12-26	최초 작성 - Buffer I/O 섹션 전체 학습 내용 정리
1.1	2024-12-26	문서명 변경, 이모티콘 제거, 하위 섹션 학습 우선순위 섹션 삭제, 컬럼 명세 추가, 누적값 기준 상세화