Buffer Replacement Algorithms 학습 문서

문서 정보

항목	내용
학습 대상	Oracle Database Concepts 19c - Memory Architecture - Database Buffer Cache - Buffer Replacement Algorithms
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

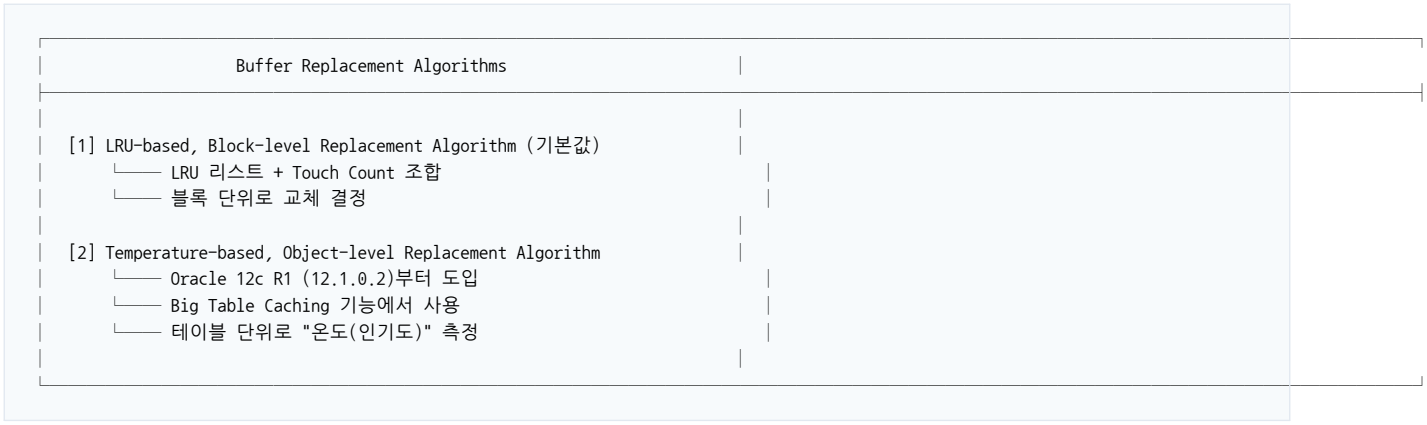
1. 이 섹션이 다루는 핵심 질문

> "Buffer Cache가 가득 찼을 때, Oracle은 어떤 버퍼를 메모리에 남기고 어떤 버퍼를 내보낼 것인가?"

이 질문이 중요한 이유는 Buffer Cache의 크기가 유한하기 때문이다. 전체 데이터베이스가 메모리에 다 올라갈 수 없으므로, Oracle은 "어떤 블록을 캐싱할 것인가"에 대한 결정 알고리즘이 필요하다. 이 결정이 비효율적이면 Physical I/O가 증가하고, 효율적이면 대부분의 요청이 메모리에서 처리된다.

2. Oracle이 사용하는 두 가지 알고리즘

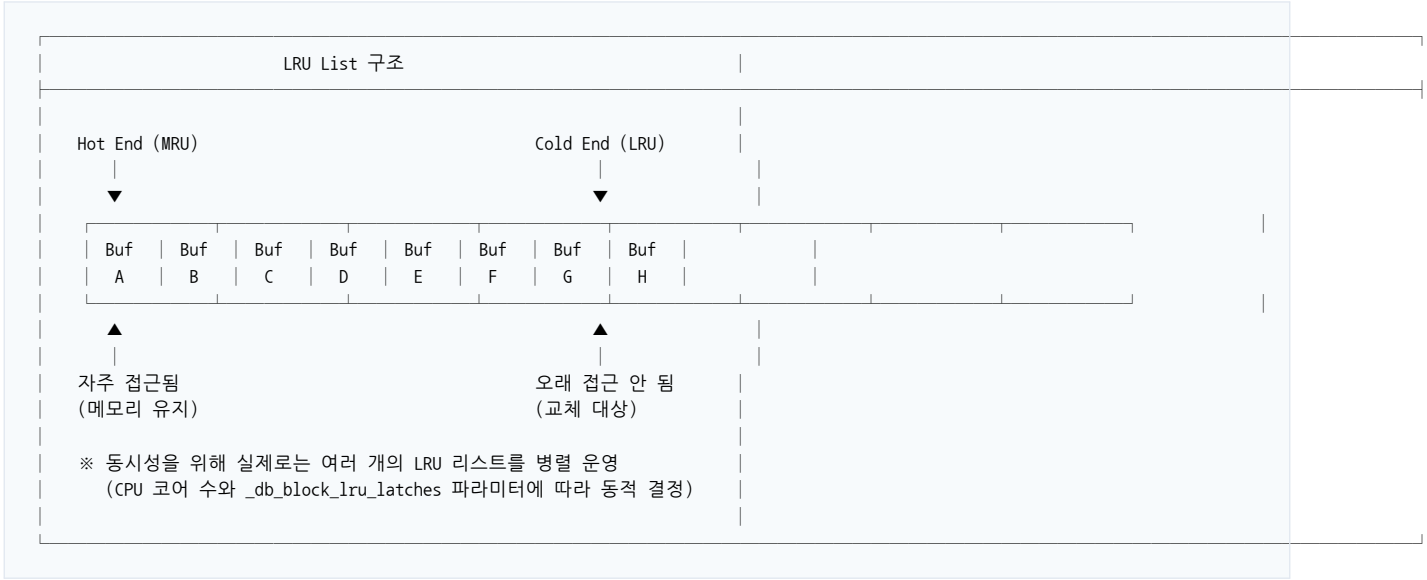
공식 문서에 따르면 Oracle은 두 가지 교체 알고리즘을 제공한다:



백엔드 개발자 관점에서 [1]번 알고리즘이 핵심이다. [2]번은 대용량 테이블 스캔에 특화된 기능이며, DBA가 `DB_BIG_TABLE_CACHE_PERCENT_TARGET` 파라미터로 활성화하는 영역이므로 개념만 인지하면 충분하다.

3. LRU-based Algorithm의 구조

3.1 LRU 리스트의 구성



Hot End (MRU - Most Recently Used): 최근에 자주 접근된 버퍼들이 위치한다. 이 영역의 버퍼는 메모리에 계속 유지된다.

Cold End (LRU - Least Recently Used): 오랫동안 접근되지 않은 버퍼들이 위치한다. 새로운 블록을 캐싱해야 할 때, 이 영역의 버퍼가 교체 대상이 된다.

3.2 왜 여러 개의 LRU 리스트를 사용하는가?

공식 문서에서 "Conceptually, there is only one LRU, but for data concurrency the database actually uses several LRUs"라고 명시한다.

단일 LRU 리스트를 사용하면 모든 Server Process가 버퍼 접근 시 동일한 리스트를 수정해야 하므로, Latch Contention(동시 접근 충돌)이 심각해진다. Oracle은 이를 방지하기 위해 여러 개의 LRU 리스트를 병렬로 운영하며, 각 버퍼는 해시 함수에 의해 특정 LRU 리스트에 할당된다. 구체적인 리스트 개수는 `_db_block_lru_latches` 히든 파라미터와 CPU 코어 수에 따라 동적으로 결정되며, 공식 문서에서는 "several"이라고만 명시한다.

4. Touch Count 메커니즘

4.1 Touch Count가 필요한 이유

순수 LRU 알고리즘의 문제점이 있다:

```
[순수 LRU의 문제]

시점 T1: 버퍼 A 접근 → Hot End로 이동
시점 T2: 버퍼 B 접근 → Hot End로 이동 (A는 밀려남)
시점 T3: 버퍼 C 접근 → Hot End로 이동 (A, B 밀려남)
...
시점 T100: 버퍼 A가 Cold End에 도달 → 교체 대상

문제: 버퍼 A가 과거에 1000번 접근되었더라도,
     "최근에" 접근되지 않았다는 이유만으로 교체됨
```

이 문제를 해결하기 위해 Oracle은 Touch Count를 도입했다. "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."

4.2 Touch Count의 동작 원리

핵심 설계 의도: 버퍼 접근 시마다 LRU 리스트를 재정렬하면 Latch Contention이 심해진다. Touch Count는 카운터만 증가시키고, 실제 리스트 위치 변경은 나중에(교체 시점에) 수행함으로써 동시성을 확보한다.

4.3 X$BH에서 Touch Count 확인

```
-- 특정 테이블의 버퍼 Touch Count 확인
SELECT
    o.object_name,
    bh.tch AS touch_count,    -- Touch Count
    bh.status,
    bh.dirty
FROM x$bh bh
JOIN dba_objects o ON bh.obj = o.data_object_id
WHERE o.object_name = 'REVIEW_STATISTICS'
ORDER BY bh.tch DESC;
```

`X$BH.TCH`가 높은 버퍼는 Hot Block으로 간주되어 메모리에 오래 유지된다.

참고: `X$BH`는 SYS 사용자만 접근 가능한 Fixed Table이며, `V$BH`는 이를 기반으로 한 Dynamic Performance View이다. Touch Count(`TCH`) 컬럼은 `X$BH`에서만 직접 확인할 수 있다.

5. Full Table Scan과 Buffer Cache 보호 메커니즘

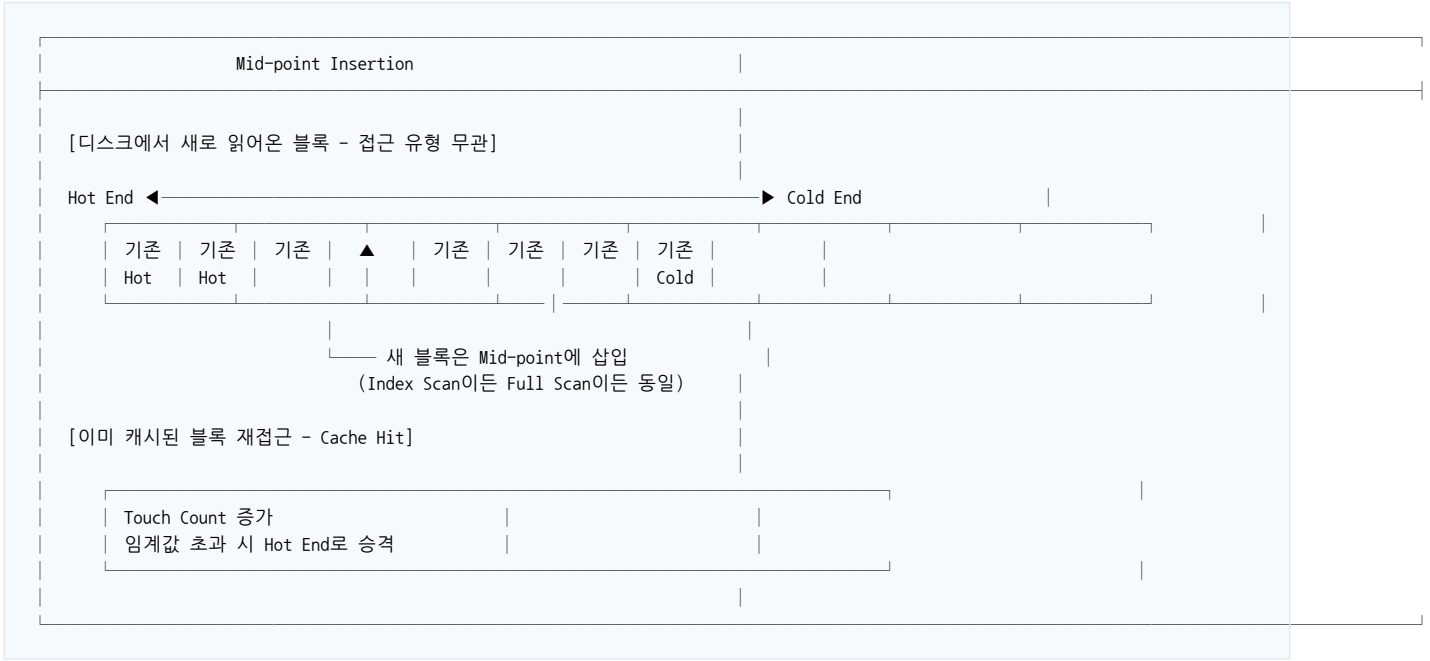
5.1 문제 상황: Full Table Scan이 Buffer Cache를 오염시킴

5.2 Oracle의 해결책: Mid-point Insertion과 Direct Path Read

Oracle은 Buffer Cache 오염을 방지하기 위해 두 가지 메커니즘을 사용한다.

5.2.1 Mid-point Insertion (모든 디스크 읽기의 기본 동작)

공식 문서에 따르면 "By default, when buffers must be read from disk, the database inserts the buffers into the middle of the LRU list. In this way, hot blocks can remain in the cache so that they do not need to be read from disk again."



핵심 설계 철학: 새로 들어온 블록은 그 가치가 아직 검증되지 않았다. 따라서 접근 유형(Index Scan, Full Table Scan)과 무관하게 LRU 리스트의 중간에서 시작한다. 반복적으로 접근되어 Touch Count가 높아진 블록만이 Hot End로 승격될 자격을 얻는다.

5.2.2 Direct Path Read (대용량 Full Table Scan의 실제 보호 메커니즘)

Mid-point Insertion만으로는 대용량 테이블 스캔의 Cache Pollution을 완전히 방지할 수 없다. Oracle은 이를 위해 Direct Path Read를 사용한다.

```
┌──────────────────────────────────────────────────────────────────────┐
│                  Direct Path Read                                    │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│  [테이블 크기 < _small_table_threshold]                               │
│                                                                      │
│     Disk → Buffer Cache(Mid-point) → Server Process                  │
│            (기존 방식)                                                │
│                                                                      │
│  [테이블 크기 > _small_table_threshold × buffer_cache_size 비율]      │
│                                                                      │
│     Disk → PGA (Direct Path Read) → Server Process                   │
│              │                                                       │
│              └──── Buffer Cache를 완전히 우회                         │
│                    Cache Pollution 원천 차단                         │
│                                                                      │
│  ※ _small_table_threshold: 기본값은 buffer_cache의 약 2%             │
│  ※ 11g R2부터 Automatic Big Table Caching 기능으로 발전              │
│                                                                      │
└──────────────────────────────────────────────────────────────────────┘
```

설계 의도: Full Table Scan으로 읽어온 블록은 대부분 한 번만 사용되고 다시 접근되지 않는다. 테이블이 충분히 크면 Buffer Cache 자체를 우회하여 PGA로 직접 읽음으로써, 기존 캐시된 Hot Block을 보호한다.

6. 프로젝트와의 연결: N+1 쿼리가 Buffer Cache에 미치는 영향

6.1 N+1 패턴의 Buffer Cache 접근 특성

```
[1차 테스트: N+1 패턴 - 25번의 개별 쿼리]

쿼리 1: SELECT * FROM REVIEW_STATISTICS WHERE PROPERTY_ID IN (id1, id2, ...)
        → Index Range Scan → 버퍼 10~15개 접근 → Touch Count 증가

쿼리 2: SELECT * FROM REVIEW_STATISTICS WHERE PROPERTY_ID IN (id3, id4, ...)
        → Index Range Scan → 버퍼 10~15개 접근 → Touch Count 증가

... (25번 반복)

특성:
- 동일 테이블의 인덱스 블록을 반복 접근
- 인덱스 루트/브랜치 블록의 Touch Count가 급격히 증가
- 이 블록들은 Hot Block으로 유지됨 → Cache Hit 유지
```

6.2 Bulk Fetch 패턴의 Buffer Cache 접근 특성

```
[2차 테스트: Bulk Fetch - 8,660개 ID를 한 번에]

쿼리: SELECT * FROM REVIEW_STATISTICS
      WHERE PROPERTY_ID IN (...1000...) OR IN (...1000...) ... OR IN (...660...)

특성:
- CONCATENATION 실행 계획으로 9개 서브쿼리 실행
- 각 서브쿼리가 동일 인덱스 블록을 반복 스캔
- 중복 블록 접근 가능성 (같은 블록을 여러 OR 절에서 읽음)
- 결과: buffer_gets 비정상적 증가 가능
```

6.3 Buffer I/O 학습 문서와의 연결

이전에 학습한 Buffer I/O 문서에서 다음 판단 기준을 제시했다:

```
[Logical I/O 효율성 판단]

buffer_gets / rows_processed = gets_per_row

< 1    : 효율적 (인덱스 활용)
1 ~ 5  : 보통 (조인/정렬 포함)
> 10   : 비효율 의심
> 100  : 심각한 비효율
```

Buffer Replacement Algorithm 학습을 통해 이제 다음을 추가로 설명할 수 있다:

1. 왜 gets_per_row가 낮으면 좋은가? - 적은 블록으로 많은 행을 반환 = 같은 블록을 재사용 - Touch Count가 높아져 Hot Block으로 유지 - 다음 쿼리에서 Cache Hit 확률 증가

2. 왜 OR 분해가 비효율적인가? - 각 OR 절이 개별적으로 인덱스 스캔 - 동일 블록을 여러 번 접근하지만, 이는 불필요한 중복 - Logical I/O 수치만 올라가고 실질적 이득 없음

7. 면접 대응 포인트

Q: "Buffer Cache의 교체 알고리즘을 설명해주세요"

합격선 답변:

> "Oracle은 LRU 리스트와 Touch Count를 조합한 알고리즘을 사용합니다. 새로운 블록이 디스크에서 읽혀오면 LRU 리스트의 중간(Mid-point)에 삽입되고, 반복 접근을 통해 Touch Count가 높아진 블록만 Hot End로 승격됩니다. 이 설계는 '새로 들어온 블록이 자신의 가치를 증명해야 한다'는 철학을 반영합니다. 대용량 Full Table Scan의 경우, 테이블 크기가 임계값을 초과하면 Buffer Cache를 우회하는 Direct Path Read가 적용되어 기존 Hot Block을 보호합니다. 제 프로젝트에서 N+1 쿼리가 발생했을 때, 인덱스 루트 블록은 25번 반복 접근되어 Touch Count가 높아졌고, 이로 인해 Cache Hit이 유지되어 개별 쿼리 실행 시간(343ms/25회 = 약 14ms)은 양호했습니다. 다만 문제는 Buffer Cache가 아니라 Connection Holding Time이었습니다."

문서 이력

버전	일자	변경 내용
1.0	2024-12-26	최초 작성 - Buffer Replacement Algorithms 섹션 학습 내용 정리

1.1	2024-12-26	기술적 오류 수정: Mid-point Insertion 설명 정정(모든 디스크 읽기가 Mid-point에 삽입됨), Full Table Scan 보호 메커니즘으로 Direct Path Read 추가, V$BH→X$BH 제목 수정, LRU 리스트 개수 출처 명확화