Oracle Checkpoint 메커니즘 완전 해설

백엔드 개발자를 위한 아키텍처 심층 분석

문서 개요

본 문서는 Oracle Database의 Checkpoint 메커니즘을 백엔드 개발자 관점에서 체계적으로 정리한 학습 자료다. DBA 수준의 운영 지식이 아닌, 애플리케이션 개발자가 알아야 할 아키텍처 수준의 이해를 목표로 한다.

학습 목표

- Checkpoint의 존재 이유와 설계 원리 이해
- Instance Recovery 과정에서 Checkpoint의 역할 파악
- 비정상 종료 시 애플리케이션에 미치는 영향 이해
- 면접에서 활용할 수 있는 수준의 기술적 설명 능력 확보

참고 자료

- Oracle Database 19c Concepts Guide
- Oracle Database Instance 공식 문서
- "Importance of Checkpoints for Instance Recovery" 섹션

Part 1. 서론: Checkpoint는 왜 존재하는가

1.1 근본적 딜레마: 성능과 내구성의 충돌

데이터베이스 설계의 가장 근본적인 긴장 관계는 **성능(Performance)**과 **내구성(Durability)** 사이에 존재한다.

내구성 극대화 접근:

```
모든 변경 → 즉시 디스크에 기록

장점: 장애 시 데이터 손실 없음
단점: 디스크 I/O가 병목 (메모리 대비 수만 배 느림)
```

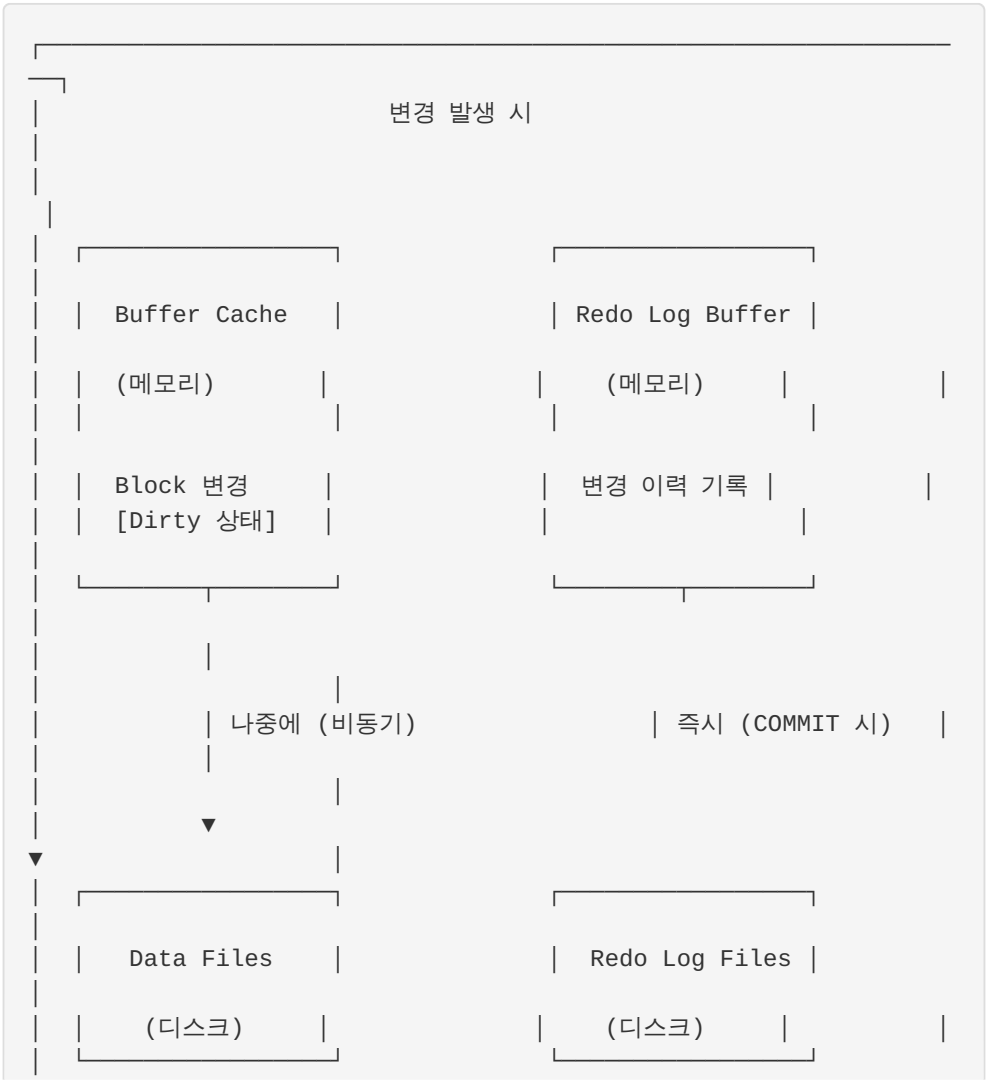
성능 극대화 접근:

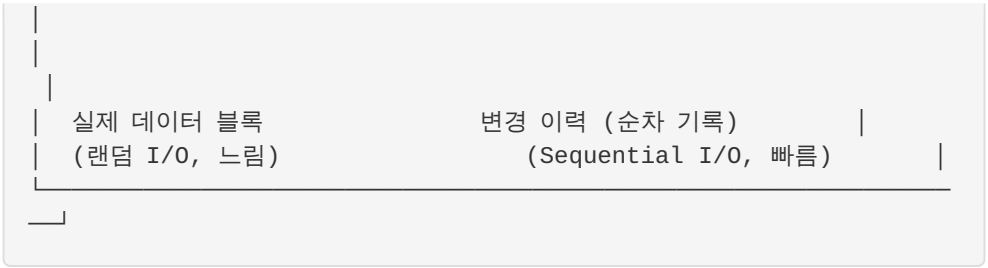
```
모든 변경 → 메모리에서만 처리

장점: 극한의 성능
단점: 장애 시 모든 데이터 손실 (메모리는 휘발성)
```

Oracle은 이 딜레마를 **Write-Ahead Logging(WAL)** 전략으로 해결했다.

1.2 Write-Ahead Logging 전략

핵심 아이디어: **"실제 데이터 블록 기록은 나중으로 미루되, 변경 이력(Redo Log)만은 즉시 디스크에 기록한다."**

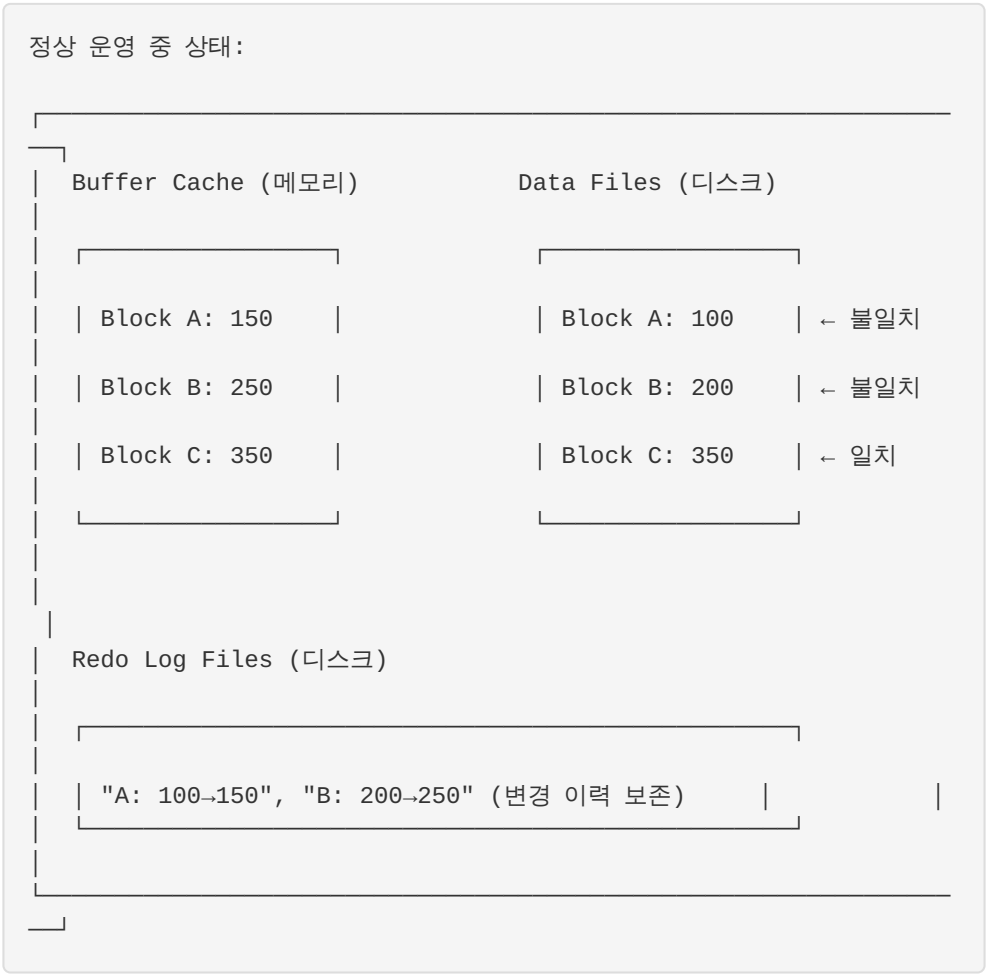




WAL의 복구 원리: 변경 이력이 디스크에 있으면, 실제 데이터 블록이 손실되어도 이력을 "재생(Replay)"하여 복구할 수 있다.

1.3 WAL이 만들어낸 새로운 문제

WAL 전략은 필연적으로 **메모리와 디스크 사이의 불일치 상태**를 만들어낸다.



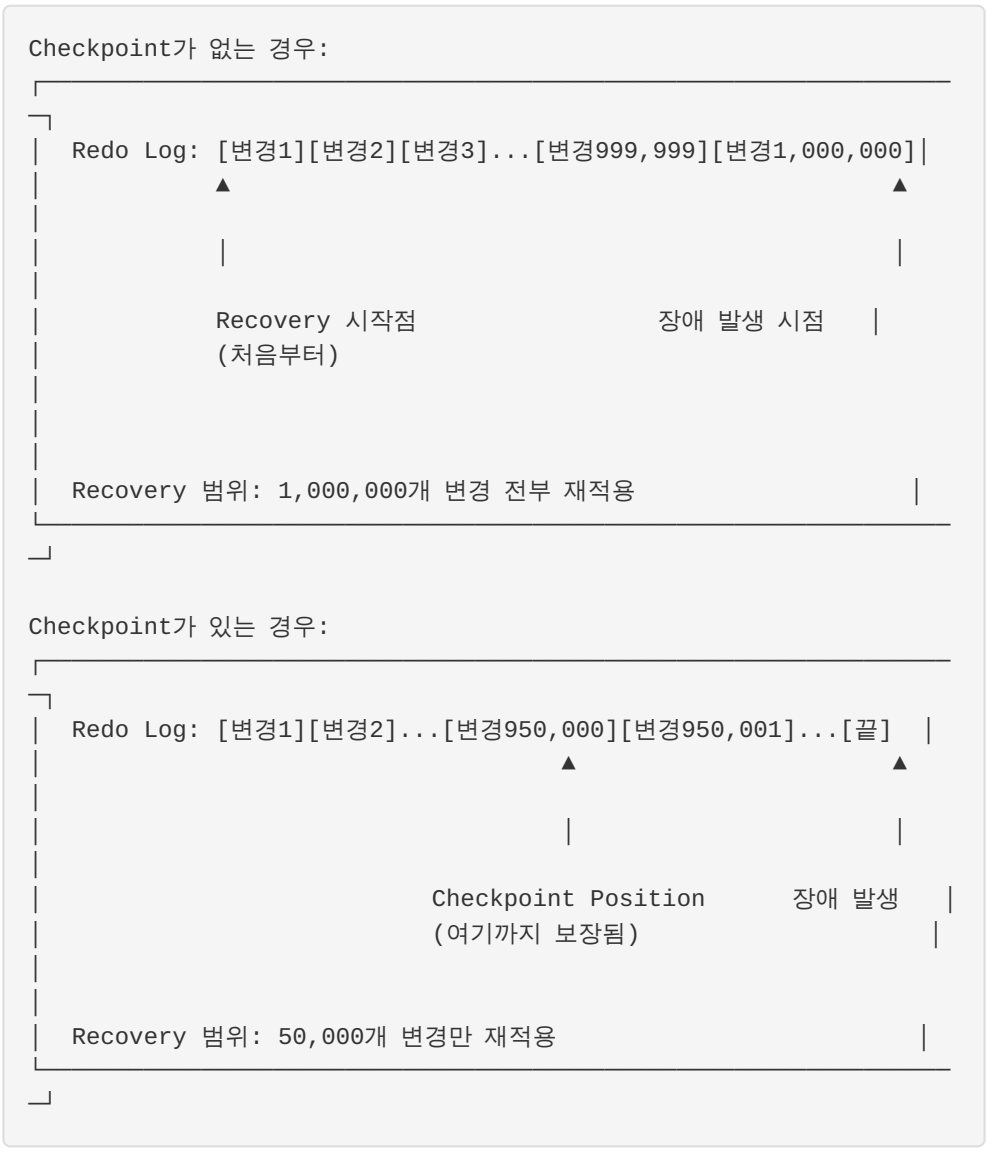
이 상태에서 장애가 발생하면:

- 메모리(Buffer Cache) 내용 전부 손실
- Data Files에는 오래된 값(100, 200)만 존재
- Redo Log Files의 이력으로 복구해야 함

문제: Redo Log 전체를 처음부터 재생하면 복구 시간이 수 시간~수일까지 소요될 수 있다.

1.4 Checkpoint의 존재 목적

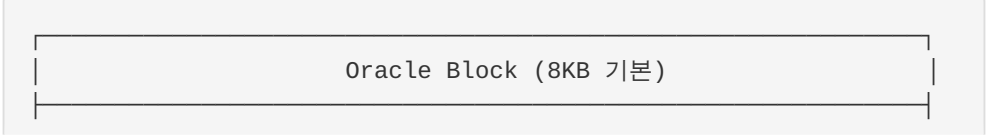
Checkpoint는 "이 시점까지의 모든 변경은 Data Files에 기록 완료되었다"는 동기화 지점을 생성하는 메커니즘이다.

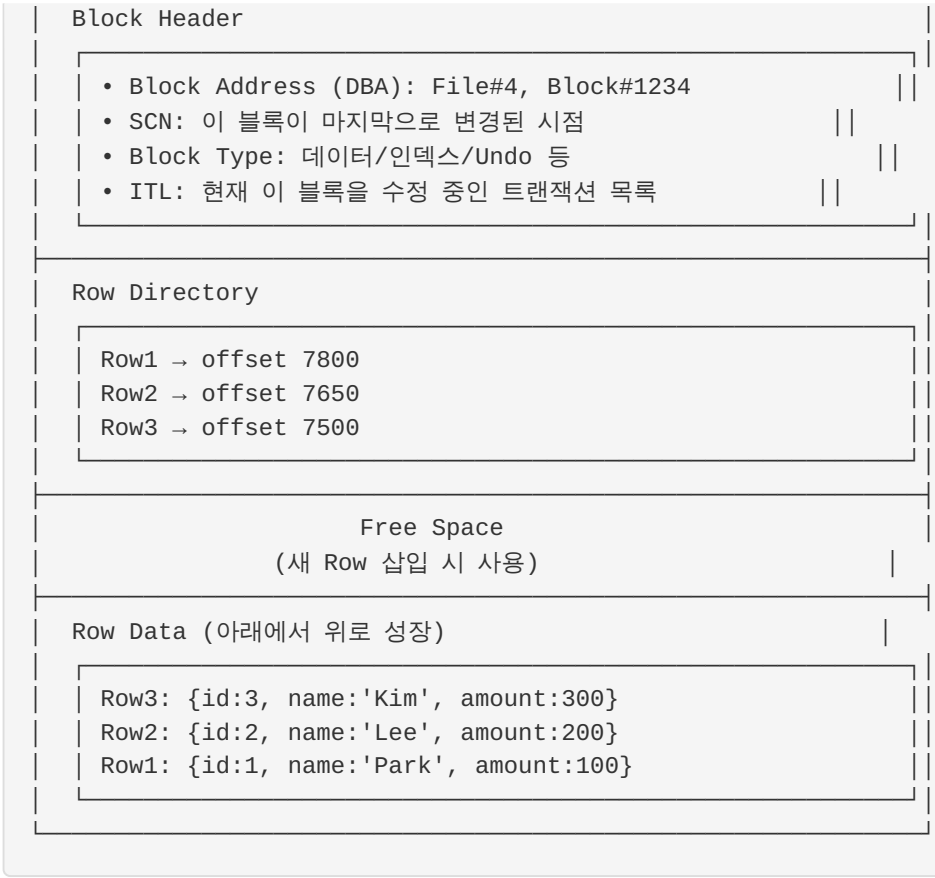


Part 2. 핵심 개념 정의

2.1 블록(Block)

Oracle이 데이터를 읽고 쓰는 **최소 I/O 단위**다. 기본 크기는 8KB(DB_BLOCK_SIZE 파라미터로 설정)이며, 다음과 같은 내부 구조를 갖는다.



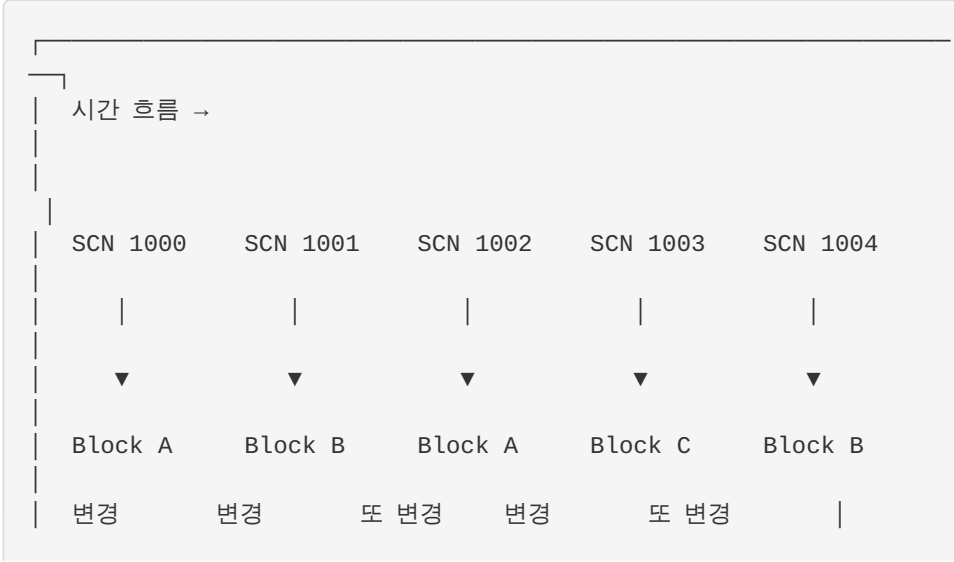


핵심 특성:

- 한 Row를 읽으려면 그 Row가 포함된 **전체 8KB 블록**을 읽음
- 한 Row를 수정하면 그 **전체 블록**이 Dirty 상태가 됨
- 블록의 SCN이 Checkpoint와 Recovery의 핵심 기준값이 됨

2.2 SCN (System Change Number)

Oracle의 **논리적 타임스탬프**다. 물리적 시간(clock)이 아니라, 변경 순서를 나타내는 단조 증가 정수다.



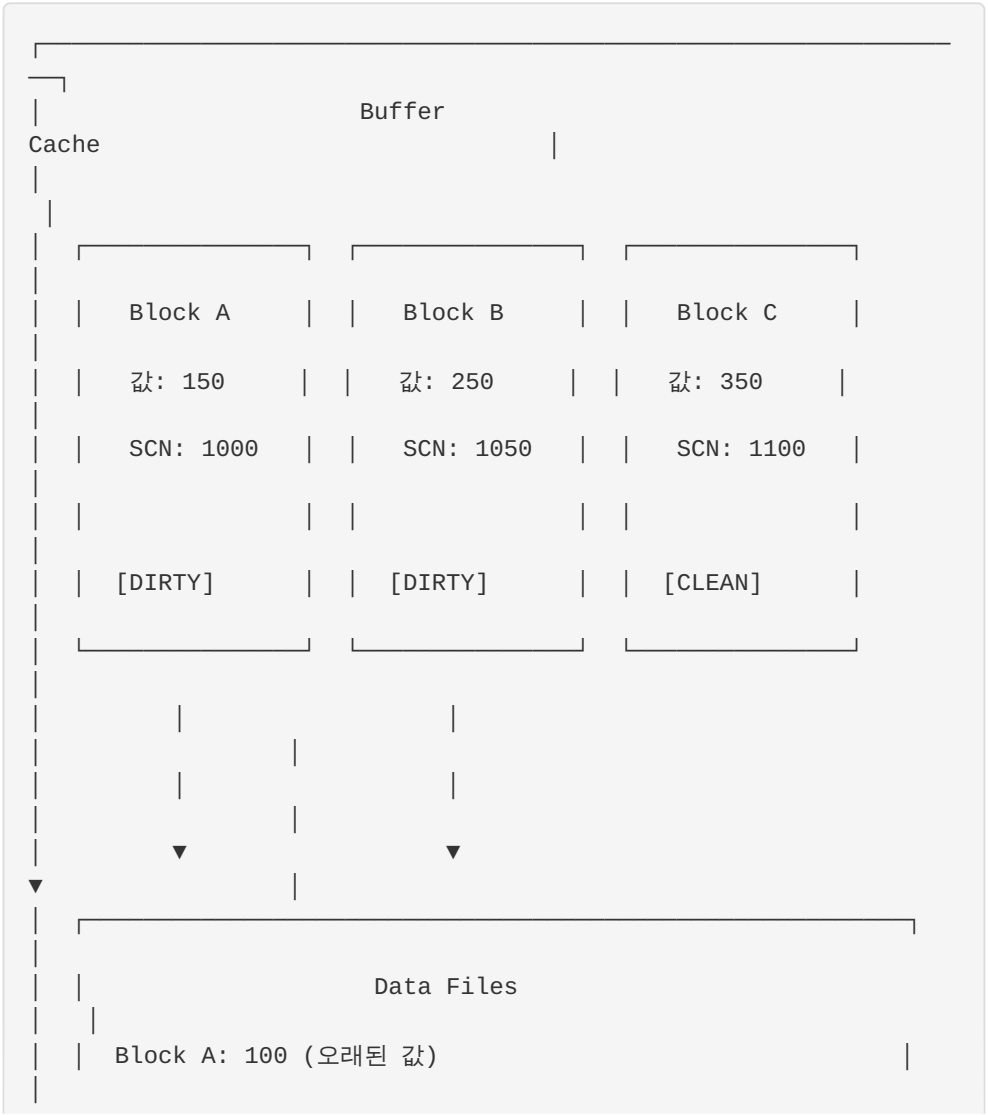
```
|
 |
|  Block A의 현재 SCN = 1002 (마지막 변경 시점)                    |
|  Block B의 현재 SCN = 1004 (마지막 변경 시점)                    |
|  Block C의 현재 SCN = 1003 (마지막 변경 시점)                    |
└──────────────────────────────────────────────────────────────────
──┘
```

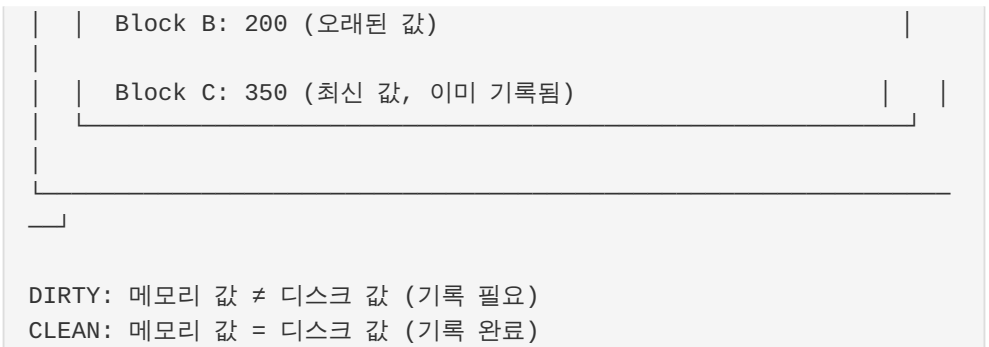
SCN의 역할:

- 모든 변경에 고유한 순서 부여
- 블록의 "나이"를 판단하는 기준
- Checkpoint Position의 단위
- Recovery 시 적용 범위 결정

2.3 Dirty Buffer

Buffer Cache 내에서 **변경되었지만 아직 Data Files에 기록되지 않은 블록**을 지칭하는 상태다.





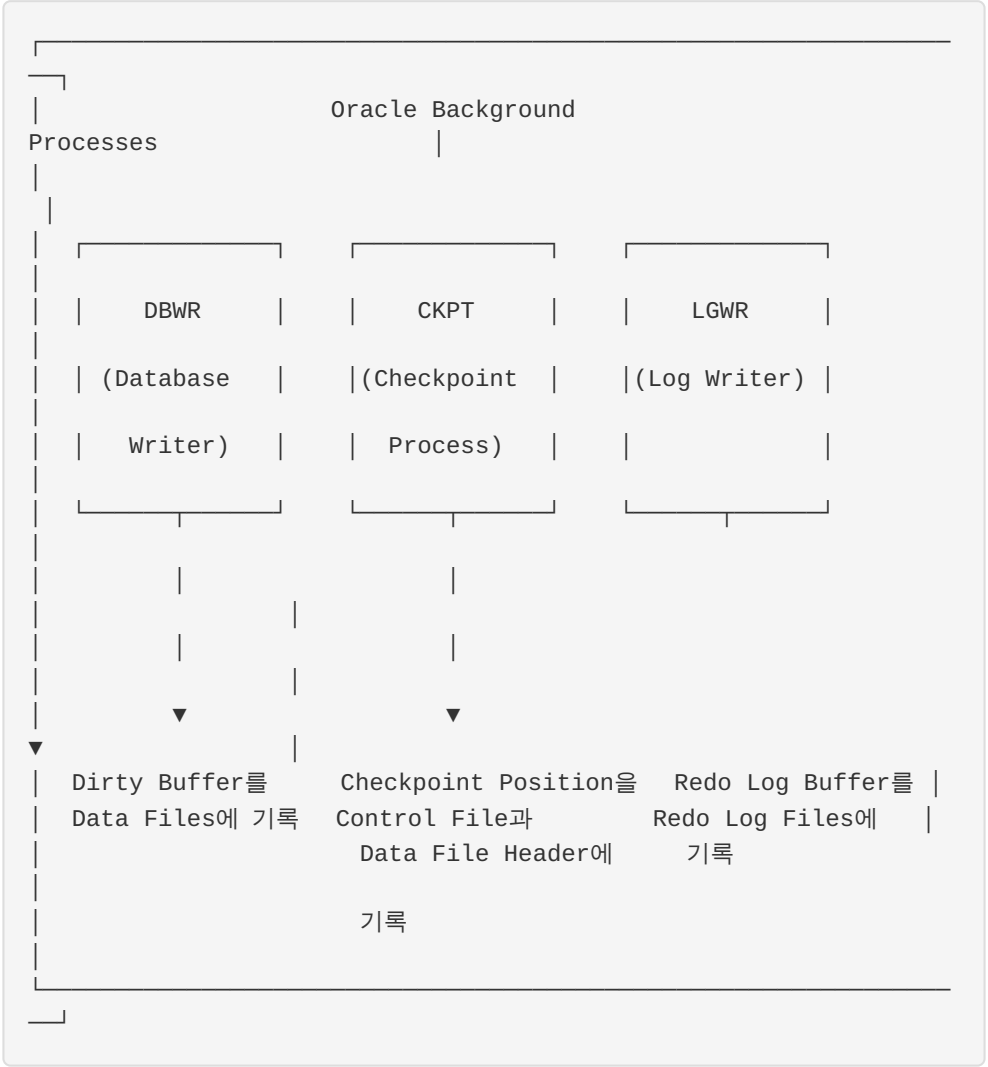
중요: Dirty Buffer는 별도의 메모리 영역이 아니라, Buffer Cache 내 블록의 **상태 플래그**다.

2.4 Checkpoint Position

"가장 오래된 Dirty Buffer의 SCN"으로 정의되며, **"이 SCN 미만의 모든 변경은 Data Files에 보장됨"**을 의미한다.

Part 3. Checkpoint 메커니즘의 구성요소

3.1 관련 프로세스



DBWR (Database Writer):

- Dirty Buffer를 Data Files에 기록하는 실제 작업 수행
- Checkpoint Queue에서 가장 오래된 것부터 기록
- COMMIT과 무관하게 비동기적으로 동작

CKPT (Checkpoint Process):

- Checkpoint Position을 Control File에 기록
- Data File Header에 Checkpoint 정보 기록
- Incremental Checkpoint: DBWR가 기록한 결과(가장 오래된 Dirty Buffer 위치)를 수집하여 반영

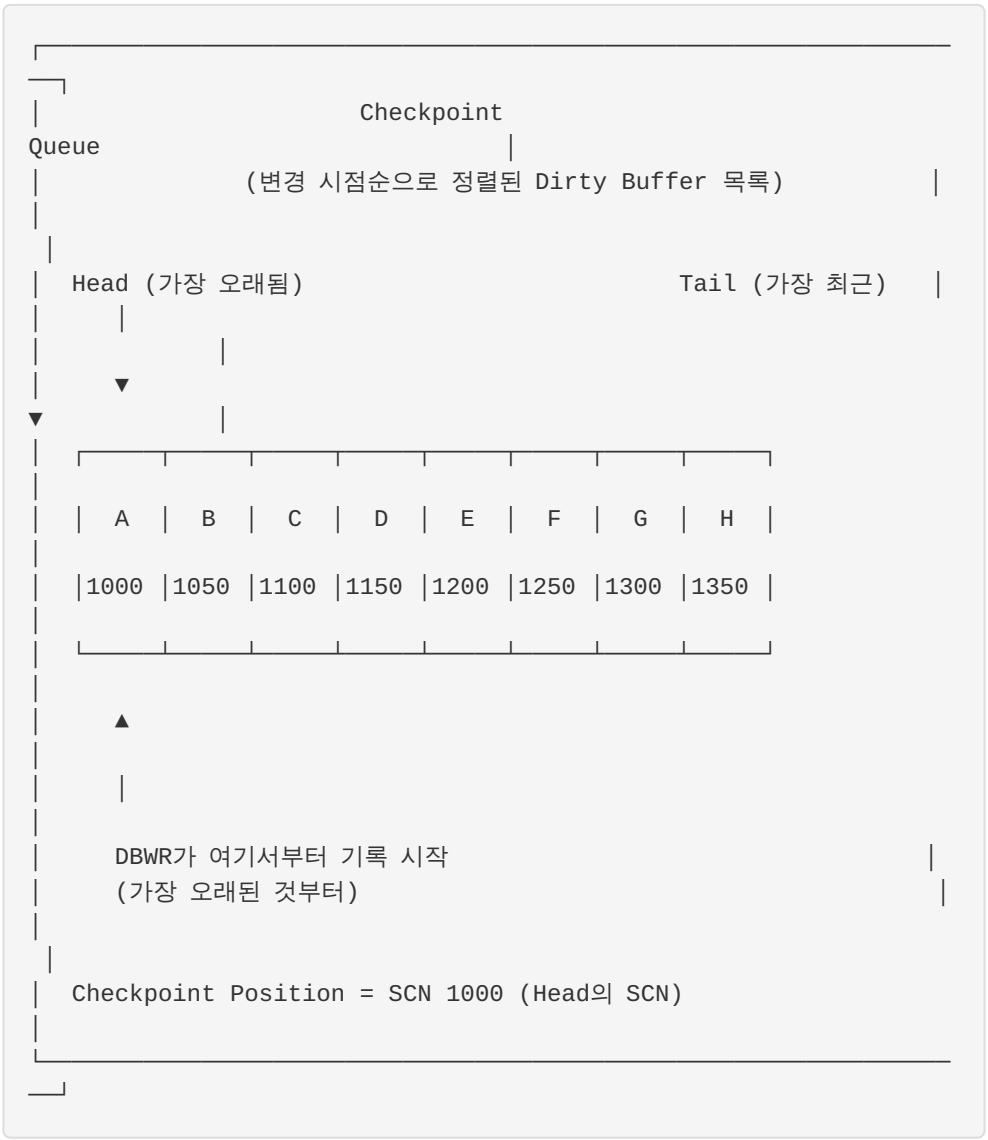
- Full Checkpoint (SHUTDOWN, ALTER SYSTEM CHECKPOINT 등): DBWR에게 전체 기록 지시

LGWR (Log Writer):

- Redo Log Buffer를 Redo Log Files에 기록
- COMMIT 시 반드시 동작 (동기적)
- Checkpoint와 직접 관련은 없으나, Log Switch 시 Checkpoint 트리거

3.2 Checkpoint Queue

Dirty Buffer를 **변경 시점(SCN) 순서로 관리**하는 자료구조다.



왜 "가장 오래된 것부터" 기록하는가:

```
시나리오 비교:

[무작위 순서 기록]
┌─────┬─────┬─────┬─────┬─────┐
```

```
┌──────┬──────┬──────┬──────┬──────┐
│  A   │  B   │  C   │  D   │  E   │  ← 초기 상태
│1000  │1050  │1100  │1150  │1200  │
└──────┴──────┴──────┴──────┴──────┘
Checkpoint Position = 1000

DBWR가 C, E를 기록 →

┌──────┬──────┬──────┬──────┬──────┐
│  A   │  B   │      │  D   │      │
│1000  │1050  │      │1150  │      │
└──────┴──────┴──────┴──────┴──────┘
Checkpoint Position = 여전히 1000 (A가 Dirty)

→ Recovery 범위: SCN 1000 ~ 끝 (개선 없음)


[가장 오래된 것부터 기록]
┌──────┬──────┬──────┬──────┬──────┐
│  A   │  B   │  C   │  D   │  E   │  ← 초기 상태
│1000  │1050  │1100  │1150  │1200  │
└──────┴──────┴──────┴──────┴──────┘
Checkpoint Position = 1000

DBWR가 A, B를 기록 →

┌──────┬──────┬──────┬──────┬──────┐
│      │      │  C   │  D   │  E   │
│      │      │1100  │1150  │1200  │
└──────┴──────┴──────┴──────┴──────┘
Checkpoint Position = 1100 (C가 새로운 Head)

→ Recovery 범위: SCN 1100 ~ 끝 (대폭 감소)
```

물리적 I/O 위치와 논리적 기록 순서의 구분:

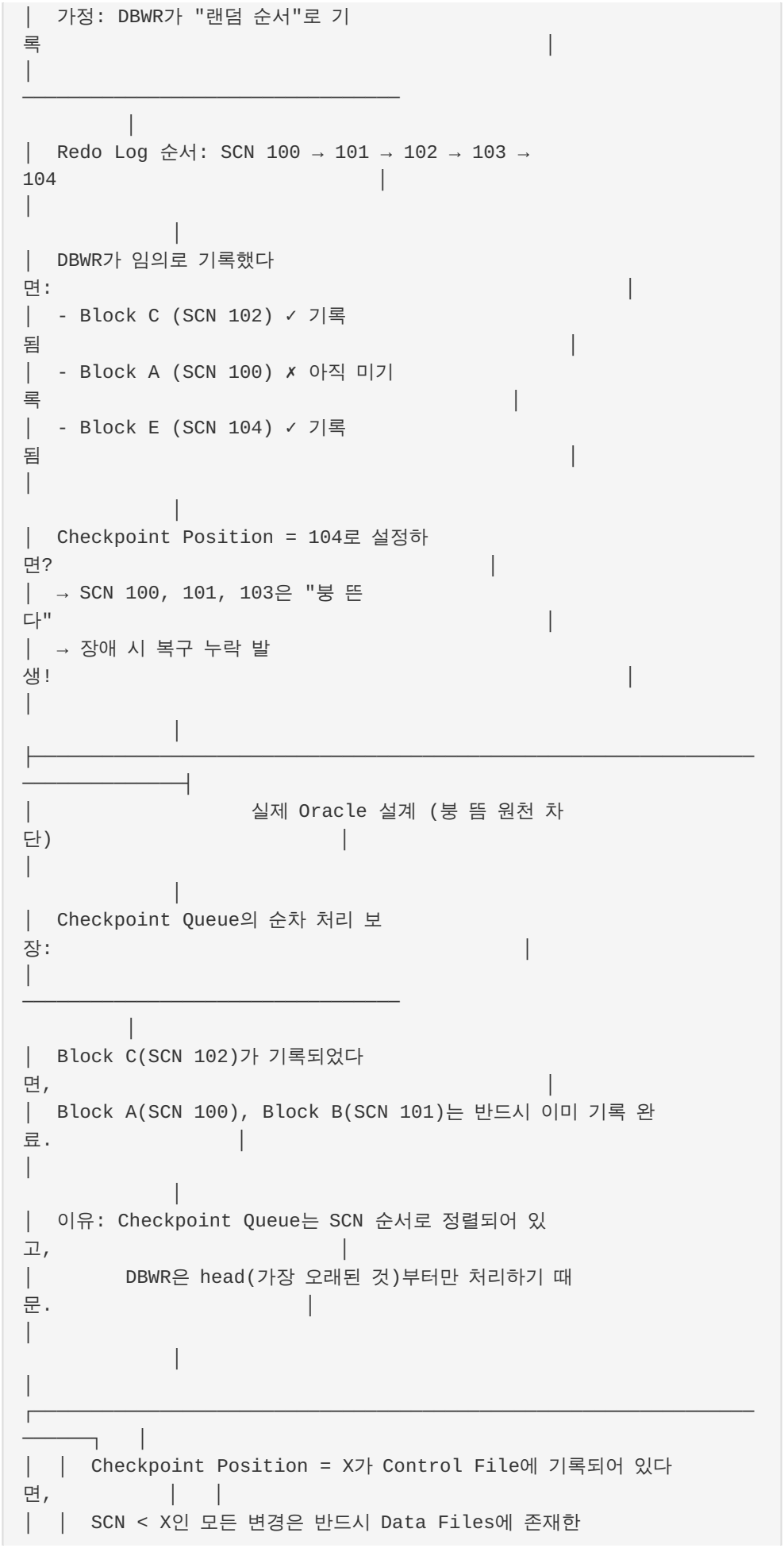
DBWR의 기록 동작에서 흔히 발생하는 오해가 있다. Data Files에 대한 물리적 I/O가 랜덤 위치에서 발생한다는 사실과, DBWR가 Checkpoint Queue 순서를 무시하고 임의로 기록한다는 것은 전혀 다른 이야기다.

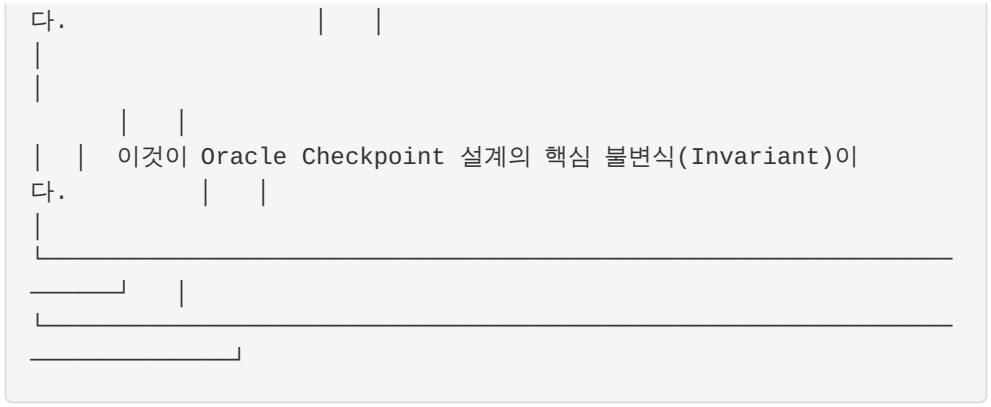
```
┌──────────────────────────────────────────────────────────────────────
──────────────┐
│              물리적 I/O 위치 vs 논리적 기록 순
서                                 │
│
             │
│  Checkpoint Queue 상
태:                                                    │
│
┌──────────────────────────────────────────────────────────────────────
──────┐    │
│  │ Block A: File 1, Block 500  (SCN 1000)  ← head (가장 오래
됨)        │   │
│  │ Block B: File 3, Block 12   (SCN
1050)                              │   │
│  │ Block C: File 1, Block 501  (SCN
```

```
1100)                                  |   |
|  | Block D: File 2, Block 999  (SCN 1150)  ← tail (최
근)               |   |
|
└──────────────────────────────────────────────────────────────────
──────┘   |
|
             |
|  DBWR 처리 순
서:                                                              |
|
──────────────────

         |
|  1단계: Block A (File 1, Block 500) 기록  ← 물리적으로 랜덤 위
치           |
|  2단계: Block B (File 3, Block 12) 기록   ← 다른 파일, 다른 위
치           |
|  3단계: Block C (File 1, Block 501) 기록  ← 또 다른 위
치                   |
|
             |
|
┌──────────────────────────────────────────────────────────────────
──────┐   |
|  |  물리적 I/O 패턴: 랜덤 (디스크 여러 위치에 흩어져 기
록)               |   |
|  |  논리적 처리 순서: 순차 (Checkpoint Queue head부터 SCN 순서대
로)     |   |
|
└──────────────────────────────────────────────────────────────────
──────┘   |
|
             |
|  핵
심:
         |
|  DBWR이 Block C(SCN 1100)를 기록 완료한 시점
에,                                |
|  Block A(SCN 1000), Block B(SCN 1050)는 반드시 이미 기록 완료 상태
다.      |
|  왜냐하면 Checkpoint Queue에서 A, B가 C보다 앞에 있었
고,                      |
|  head부터 순차 처리되어 이미 큐에서 제거되었기 때문이
다.                      |
└──────────────────────────────────────────────────────────────────
──────────────┘
```

"중간이 붕 뜨는" 시나리오가 불가능한 구조적 이유:

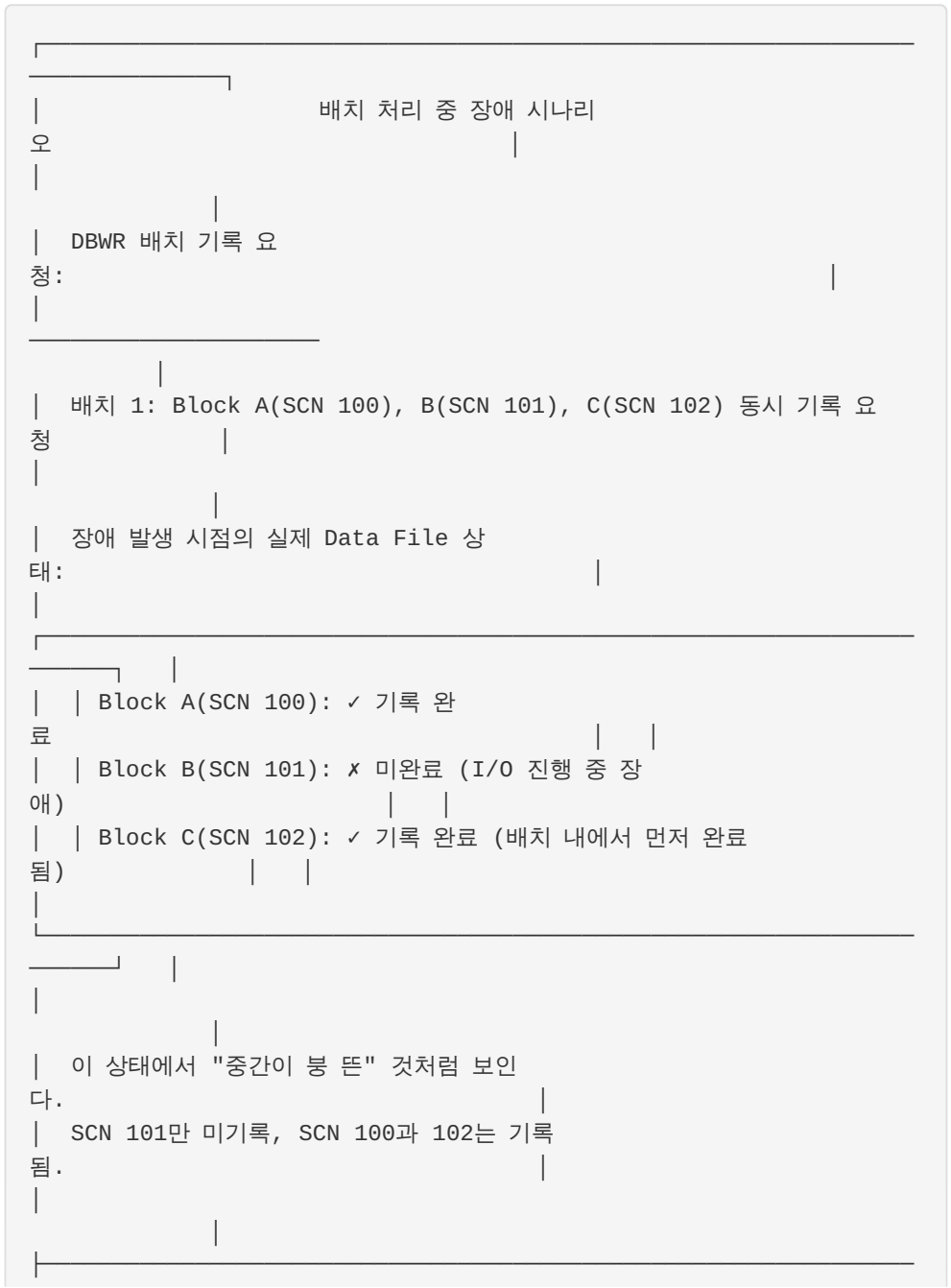
```
┌──────────────────────────────────────────────────────────────────
──────────────┐
|                    우려 시나리오 (실제로는 발생 불
가)                          |
|
             |
```

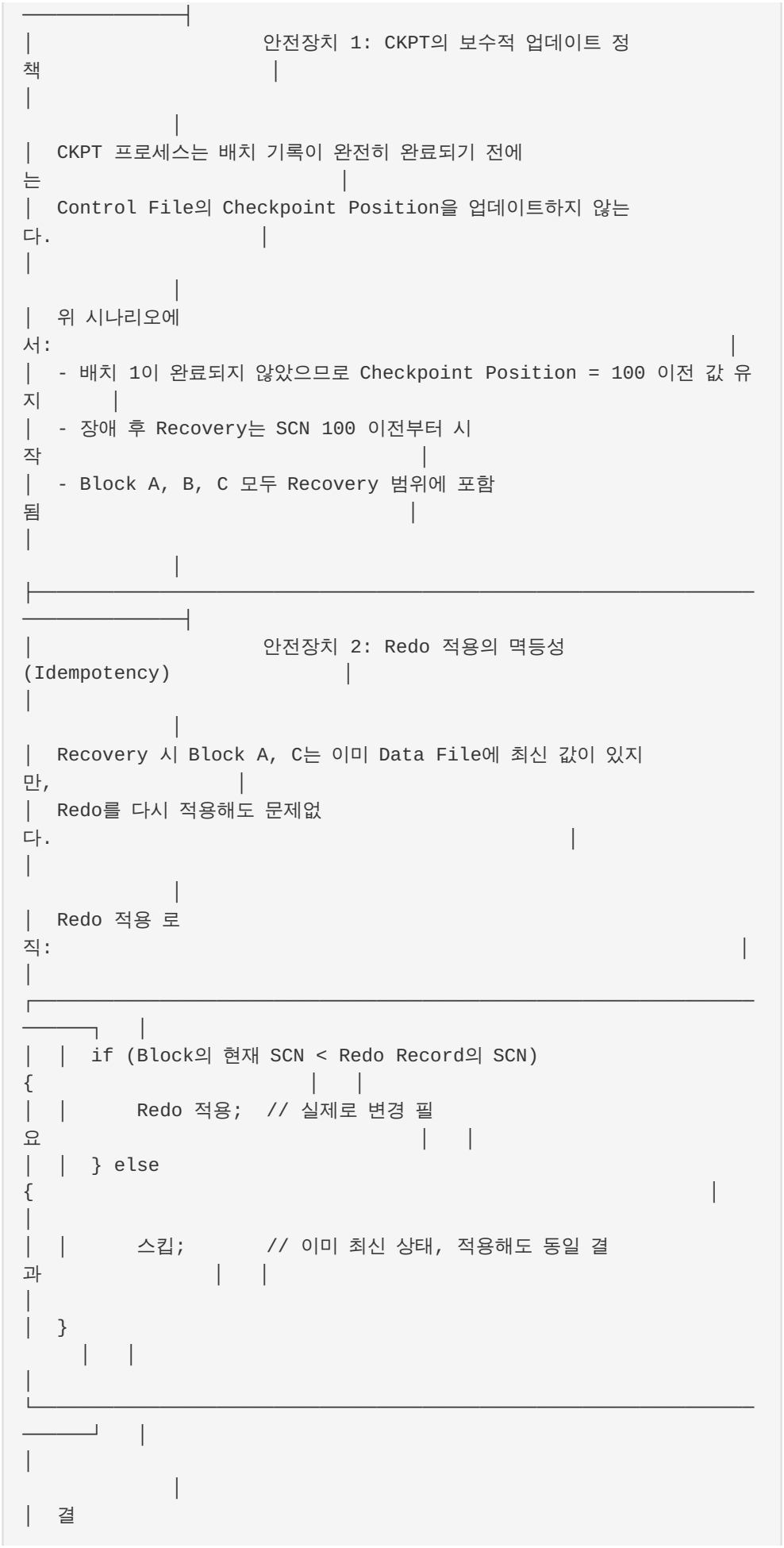

가정: DBWR가 "랜덤 순서"로 기록
Redo Log 순서: SCN 100 → 101 → 102 → 103 → 104
DBWR가 임의로 기록했다면:
- Block C (SCN 102) ✓ 기록됨
- Block A (SCN 100) ✗ 아직 미기록
- Block E (SCN 104) ✓ 기록됨
Checkpoint Position = 104로 설정하면?
→ SCN 100, 101, 103은 "붕 뜬다"
→ 장애 시 복구 누락 발생!
실제 Oracle 설계 (붕 뜸 원천 차단)
Checkpoint Queue의 순차 처리 보장:
Block C(SCN 102)가 기록되었다면,
Block A(SCN 100), Block B(SCN 101)는 반드시 이미 기록 완료.
이유: Checkpoint Queue는 SCN 순서로 정렬되어 있고,
DBWR은 head(가장 오래된 것)부터만 처리하기 때문.
Checkpoint Position = X가 Control File에 기록되어 있다면,
SCN < X인 모든 변경은 반드시 Data Files에 존재한

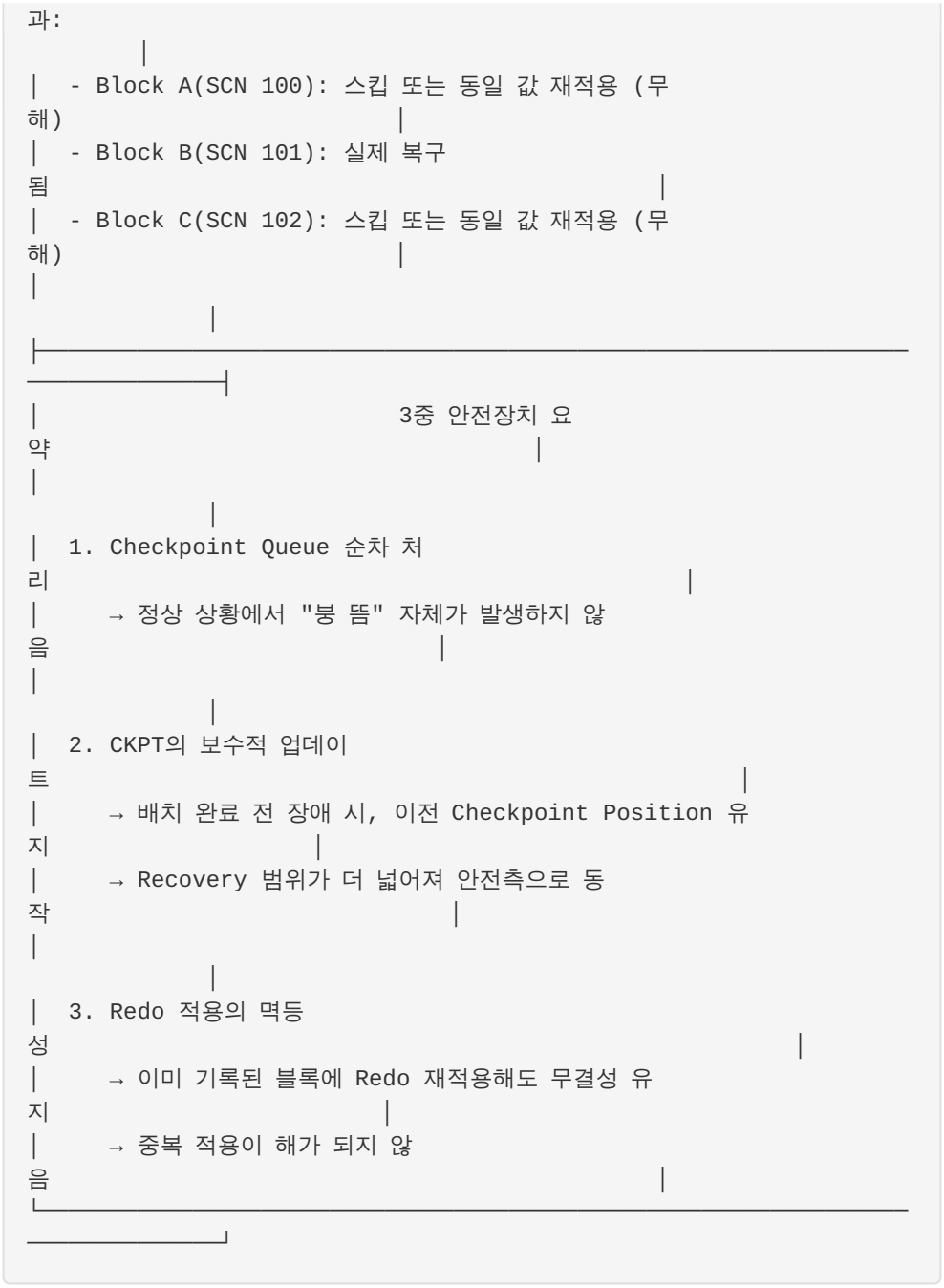


배치 처리와 비동기 I/O 시의 안전장치:

실제 DBWR은 성능을 위해 여러 블록을 배치로 묶어 비동기 I/O를 수행한다. 이 경우 배치 내 블록들의 기록 완료 순서가 SCN 순서와 다를 수 있다.

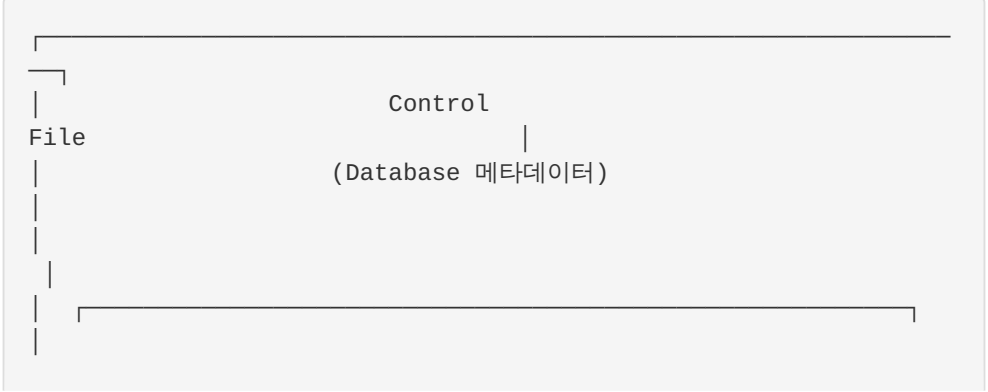



안전장치 1: CKPT의 보수적 업데이트 정책
CKPT 프로세스는 배치 기록이 완전히 완료되기 전에는
Control File의 Checkpoint Position을 업데이트하지 않는다.
위 시나리오에서:
- 배치 1이 완료되지 않았으므로 Checkpoint Position = 100 이전 값 유지
- 장애 후 Recovery는 SCN 100 이전부터 시작
- Block A, B, C 모두 Recovery 범위에 포함됨
안전장치 2: Redo 적용의 멱등성 (Idempotency)
Recovery 시 Block A, C는 이미 Data File에 최신 값이 있지만,
Redo를 다시 적용해도 문제없다.
Redo 적용 로직:
if (Block의 현재 SCN < Redo Record의 SCN) {
Redo 적용; // 실제로 변경 필요
} else {
스킵; // 이미 최신 상태, 적용해도 동일 결과
}
결

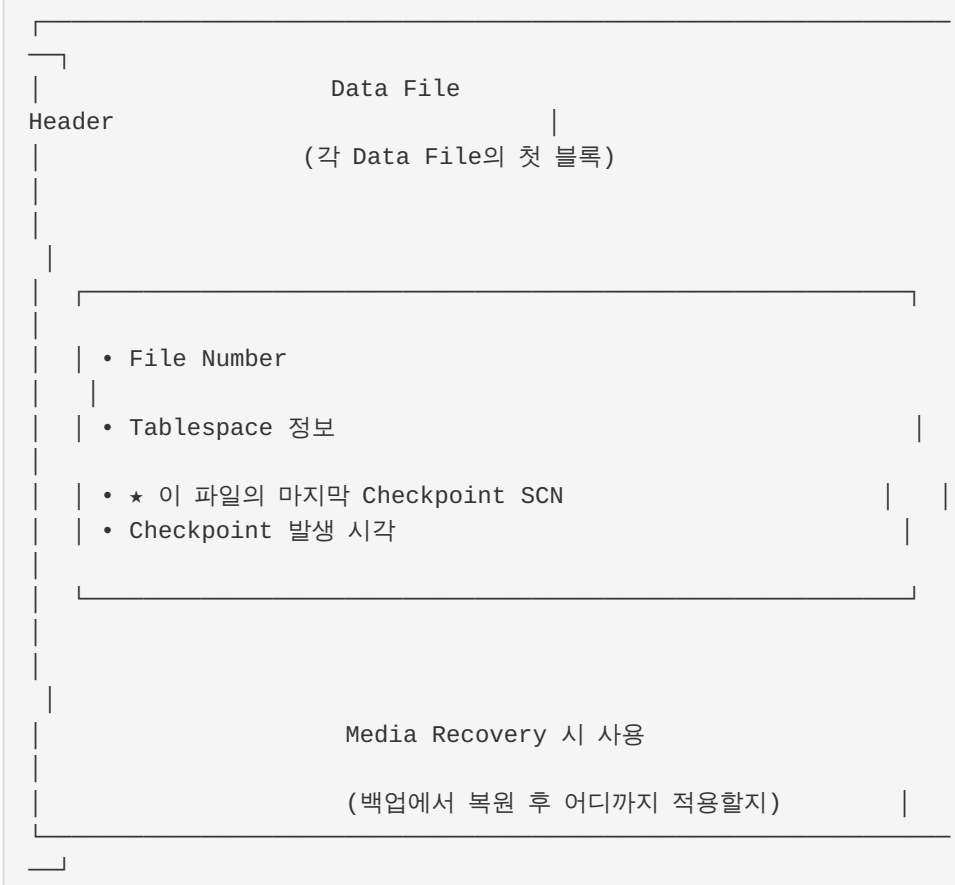


3.3 Control File과 Data File Header

Checkpoint 정보가 **영구 저장**되는 위치다.




• Database Name
• Data Files 목록 및 위치
• Redo Log Files 목록 및 위치
• 현재 SCN
• ★ Checkpoint Position (SCN) ← 핵심
• Checkpoint 발생 시각
Recovery 시 이 값을 읽어서
어디서부터 Redo를 적용할지 결정


Data File Header
(각 Data File의 첫 블록)
• File Number
• Tablespace 정보
• ★ 이 파일의 마지막 Checkpoint SCN
• Checkpoint 발생 시각
Media Recovery 시 사용
(백업에서 복원 후 어디까지 적용할지)

Part 4. Checkpoint의 동작 원리

4.1 Checkpoint Position 결정 메커니즘

Oracle 공식 문서 정의:

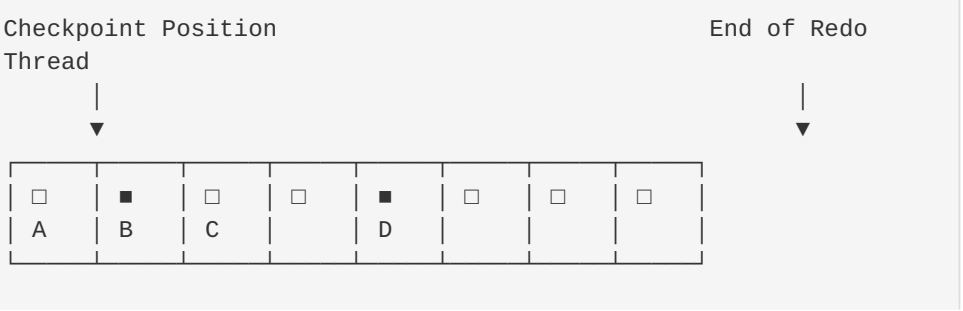
> "The checkpoint position is determined by the oldest dirty buffer in the database buffer cache."

이 정의가 **무결성을 보장하는 논리적 근거**가 된다:

```
논리적 증명:

전제 1: Checkpoint Position = 가장 오래된 Dirty Buffer의 SCN
전제 2: Dirty Buffer는 아직 Data Files에 기록되지 않은 블록

추론:
1. Checkpoint Position이 SCN 1000이면,
   SCN 1000인 블록이 "가장 오래된" Dirty Buffer다.

2. SCN 999인 Dirty Buffer가 존재한다고 가정하면,
   그 블록이 "가장 오래된" Dirty Buffer가 되어야 한다.

3. 그러나 Checkpoint Position이 1000이므로,
   SCN 999인 Dirty Buffer는 존재할 수 없다.

4. Dirty가 아니면 Clean이다 = Data Files에 기록되었다.

결론: SCN 1000 미만의 모든 블록은 Data Files에 반드시 있다.
```

4.2 Checkpoint Position 이후에 이미 기록된 블록이 존재하는 이유

Oracle 공식 문서 Figure 14-5를 분석하면, Checkpoint Position 이후에 일부 블록은 이미 Data Files에 기록되어 있고(검은색), 일부는 아직 기록되지 않은(흰색) 상태가 혼재한다.



```
□ = 아직 Data Files에 기록 안됨 (Dirty)
■ = 이미 Data Files에 기록됨 (Clean)
```

이 상태가 발생하는 이유:

Checkpoint 메커니즘과 별개로, DBWR는 **다른 트리거**에 의해서도 Dirty Buffer를 기록한다. 대표적인 경우가 **Free Buffer 확보**다.

```
┌──────────────────────────────────────────────────────────────────────
──────────────┐
│                    Checkpoint 외의 DBWR 기록 트리
거                          │
│
             │
│  1. Free Buffer 확보 (LRU 기
반)                                          │
│
──────────────────────────
      │
│     서버 프로세스가 새 블록을 Buffer Cache로 읽으려 할
때                     │
│     Free Buffer가 부족하면, LRU 리스트의 Cold 끝에 있는 Dirty
Buffer를       │
│     Data Files에 기록하여 Free Buffer로 전환한
다.                            │
│
             │
│     → 이 경우 Checkpoint Queue 순서와 무관하게 LRU 순서로 기록
됨             │
│
             │
│  2. 테이블스페이스 오프라인, 테이블 DROP
등                                    │
│     해당 객체와 관련된 Dirty Buffer만 선별적으로 기
록                         │
│
             │
│  3. Direct Path
Write                                                          │
│     특정 작업(대량 로드 등)에서 Buffer Cache를 거치지 않고 직접 기
록         │
└──────────────────────────────────────────────────────────────────────
──────────────┘
```

```
시점 T1: 모두 Dirty 상태
┌──────────────────────────────────────────────────┐
│ Block A (SCN 1000) [Dirty]  ← 가장 오래된 Dirty    │
│ Block B (SCN 1050) [Dirty]                       │
│ Block C (SCN 1100) [Dirty]                       │
│ Block D (SCN 1150) [Dirty]                       │
└──────────────────────────────────────────────────┘
```

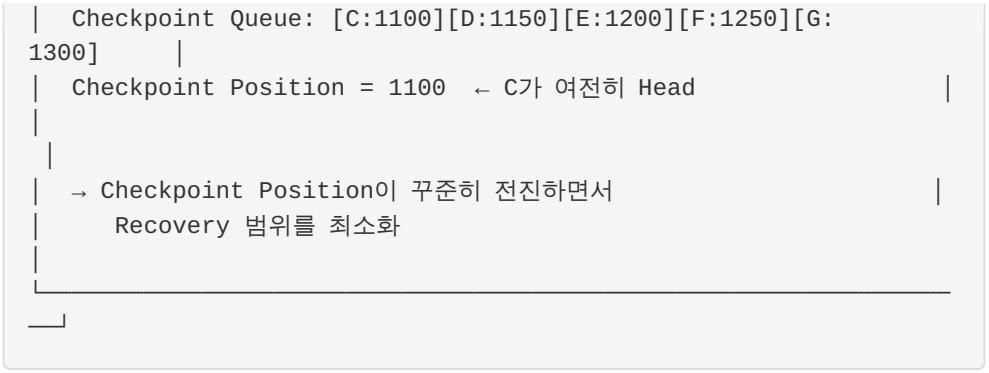
```
Checkpoint Position = SCN 1000 (A의 SCN)


시점 T2: Free Buffer 확보를 위해 LRU Cold 끝의 B, D가 기록됨
         (Checkpoint Queue 순서가 아닌 LRU 순서에 의한 기록)
┌──────────────────────────────────────────────────┐
│ Block A (SCN 1000) [Dirty]  ← 여전히 가장 오래됨   │
│ Block B (SCN 1050) [Clean]  ← Data Files에 기록됨  │
│ Block C (SCN 1100) [Dirty]                       │
│ Block D (SCN 1150) [Clean]  ← Data Files에 기록됨  │
└──────────────────────────────────────────────────┘

Checkpoint Position = 여전히 SCN 1000
(B, D가 기록되었어도 A가 여전히 Dirty이므로 전진하지 않음)
```

핵심: Checkpoint Position은 "DBWR가 기록한 지점"이 아니라 **"가장 오래된 Dirty Buffer의 SCN"**이다. DBWR는 Checkpoint Queue(SCN 순서)와 LRU 리스트(접근 빈도 순서) 두 가지 경로로 Dirty Buffer를 기록하며, 후자의 경우 Checkpoint Position 이후의 블록이 먼저 기록될 수 있다.

4.3 Incremental Checkpoint

Checkpoint Position을 **점진적으로 전진**시키는 메커니즘이다.

```
┌──────────────────────────────────────────────────────────────────────
──┐
│                   Incremental Checkpoint 동
작                    │
│
 │
│  [T1] 초기 상태
│
│  Checkpoint Queue: [A:1000][B:1050][C:1100][D:1150][E:
1200]     │
│  Checkpoint Position =
1000                                      │
│
 │
│  [T2] DBWR가 3초 후 동작, A를 기록                                  │
│  Checkpoint Queue: [B:1050][C:1100][D:1150][E:
1200]             │
│  Checkpoint Position = 1050  ← CKPT가 Control File 업데이트     │
│
 │
│  [T3] DBWR가 3초 후 동작, B를 기록                                  │
│  Checkpoint Queue: [C:1100][D:1150][E:
1200]                      │
│  Checkpoint Position = 1100  ← 또 전진
│
│
 │
│  [T4] 새 변경 발생 (F:1250, G:1300)
│
```

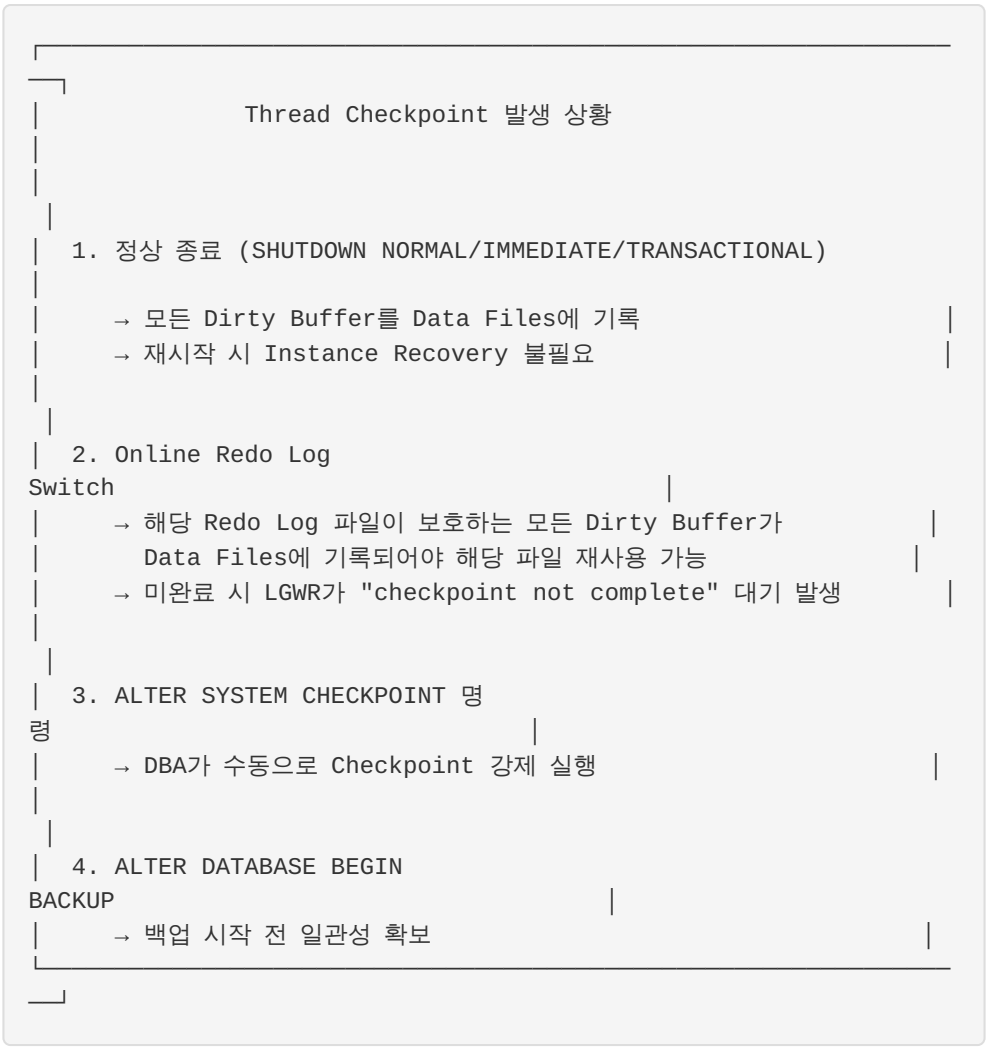


Incremental Checkpoint의 목적:

- Recovery 시간 최소화 (MTTR 보장) - 본질적 목적
- Recovery 범위를 지속적으로 최소화
- I/O 부하를 분산하여 Log Switch 시 대량 기록 방지 (부수적 효과)

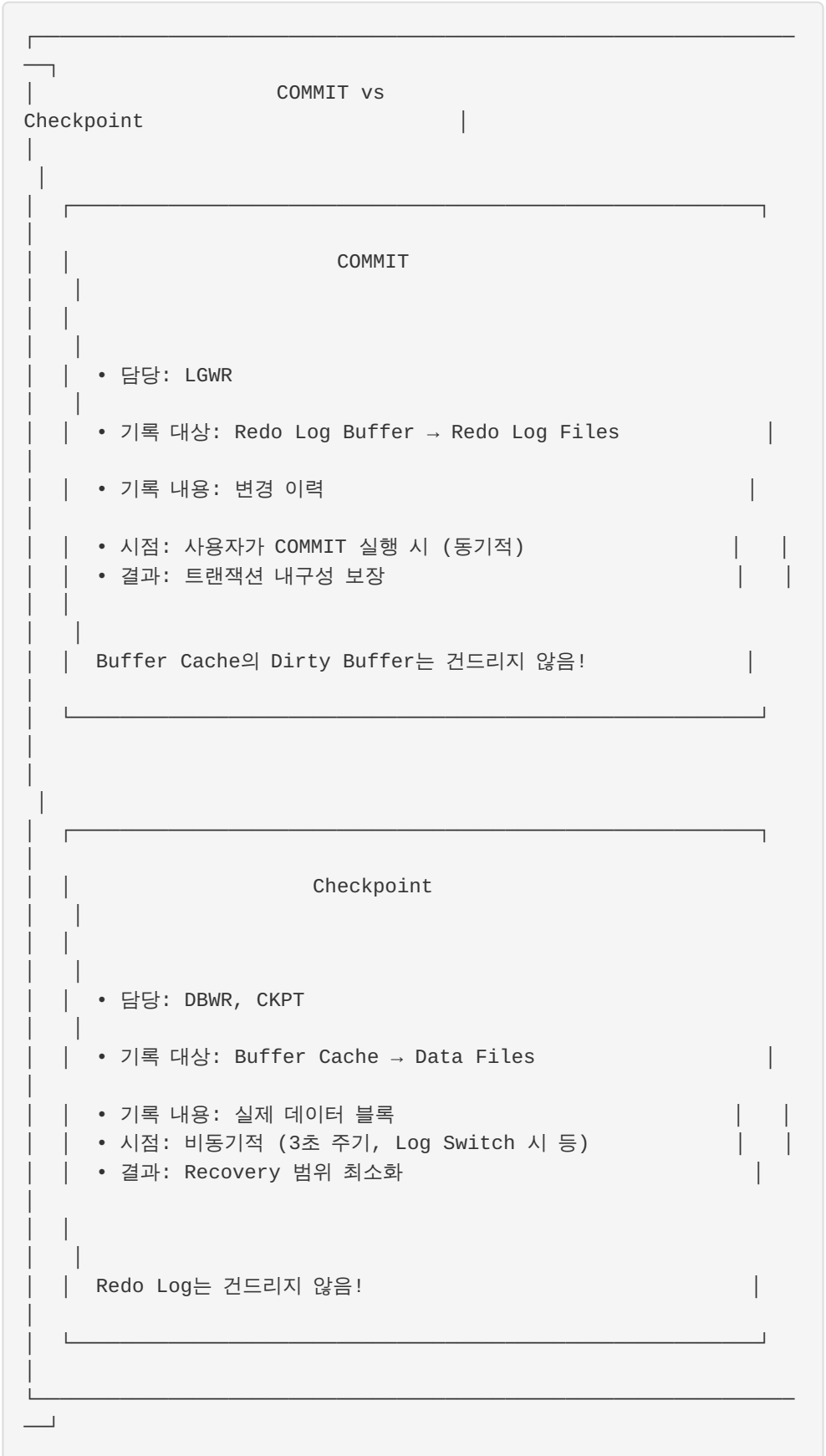
4.4 Thread Checkpoint (Full Checkpoint)

특정 이벤트 발생 시 **모든 Dirty Buffer를 기록**하는 Checkpoint다.



4.5 Checkpoint와 COMMIT의 관계

핵심: COMMIT과 Checkpoint는 완전히 별개의 작업이다.

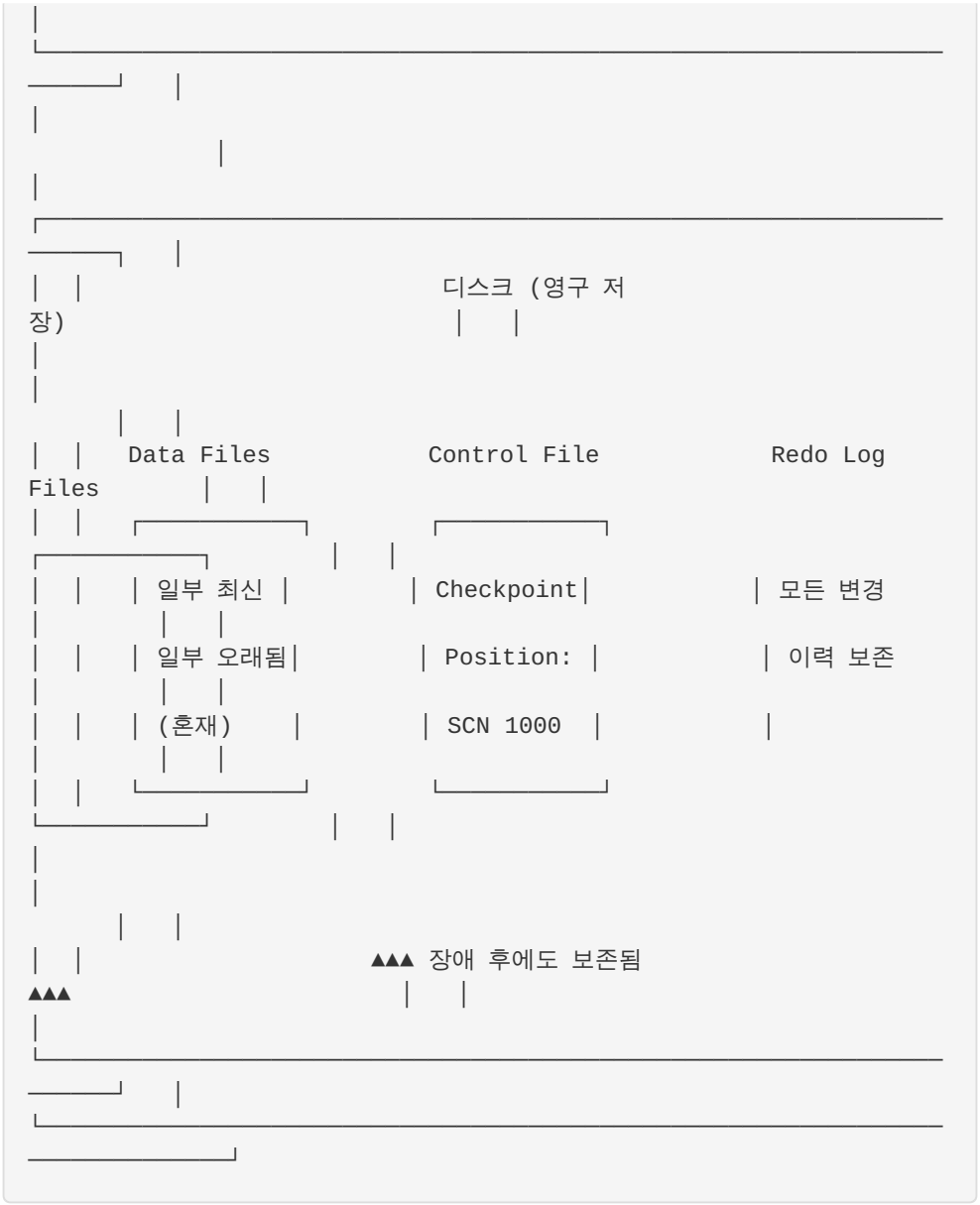


시간 흐름 예시:

```
T1: UPDATE 실행 → Buffer Cache 변경, Redo Buffer 기록
T2: COMMIT 실행 → LGWR가 Redo Log Files에 기록
    (이 시점에 Data Files는 변경 안됨!)
T3: (나중에) DBWR가 Dirty Buffer를 Data Files에 기록

T2~T3 사이에 장애 발생하면?
→ COMMIT은 완료됨 (Redo Log에 기록됨)
→ Data Files에는 아직 반영 안됨
→ Instance Recovery로 Redo 재적용하여 복구
```

Part 5. Instance Recovery에서의 Checkpoint 역할

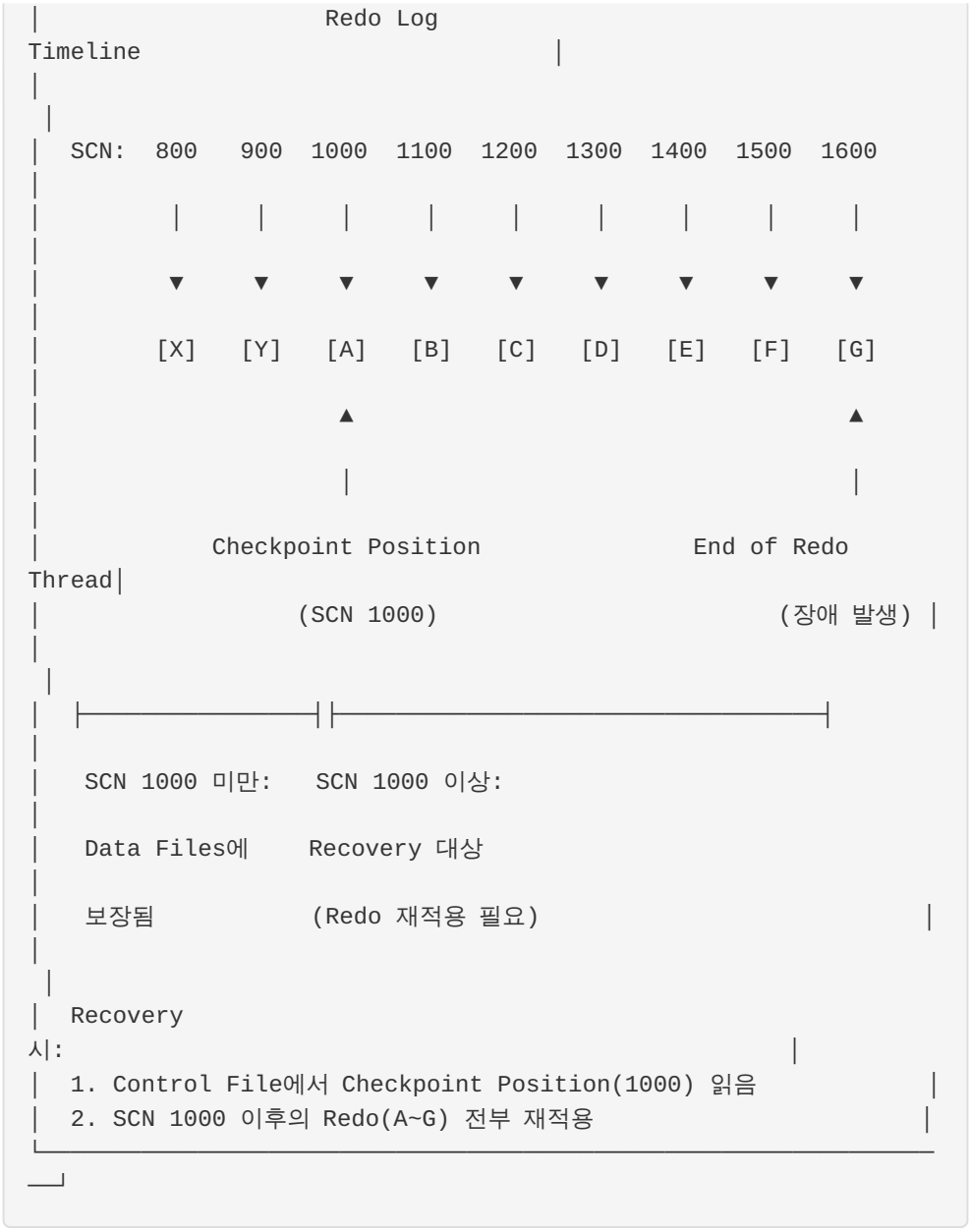
5.1 메모리 vs 디스크: 장애 시 각 구성요소의 운명



핵심 포인트:

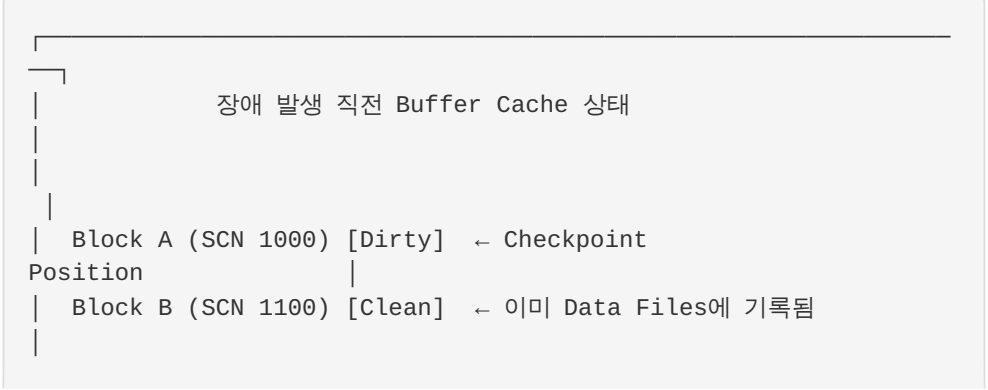
- **Checkpoint Queue는 메모리에 있다** → 장애 시 손실됨
- **Checkpoint Position은 Control File(디스크)에 있다** → 장애 후에도 "어디까지 보장되는지" 알 수 있음
- **Redo Log Files도 디스크에 있다** → 모든 변경 이력이 보존됨
- **두 가지의 결합으로 복구 가능:**
- Control File의 Checkpoint Position → "어디서부터" 복구할지 결정
- Redo Log Files → "무엇을" 복구할지 제공

5.2 Recovery 범위 결정



5.3 "이미 기록된 블록"의 처리와 멱등성

Checkpoint Position 이후에 일부 블록은 DBWR에 의해 이미 Data Files에 기록되어 있을 수 있다. 그러나 Oracle은 어떤 것이 기록되었는지 개별 추적하지 않는다. 따라서 Recovery 시 해당 범위의 모든 Redo를 재적용한다.



```
|  Block C (SCN 1200)
[Dirty]                                          |
|  Block D (SCN 1300) [Clean]  ← 이미 Data Files에 기록됨
|
|  Block E (SCN 1400)
[Dirty]                                          |
|
 |
|  Checkpoint Position = SCN
1000                                  |
|
 |
|  ──────────────────────────────────────────────────────────
|
|
 |
|  Recovery
시:                                                        |
|  • Oracle은 B, D가 이미 기록되었는지 모름                          |
|  • SCN 1000 이후 모든 Redo(A, B, C, D, E)를 전부 재적용          |
|
 |
|  왜 문제가 안 되는가?                                          |
|  • Redo 적용은 멱등(Idempotent)함
|
|  • "Block B의 offset 100을 값 X로 변경"                          |
|  • 이미 X라면 → X로 다시 변경 → 결과 동일                        |
|  • 아직 X가 아니라면 → X로 변경 → 정상 복구                      |
└──────────────────────────────────────────────────────────
──┘
```

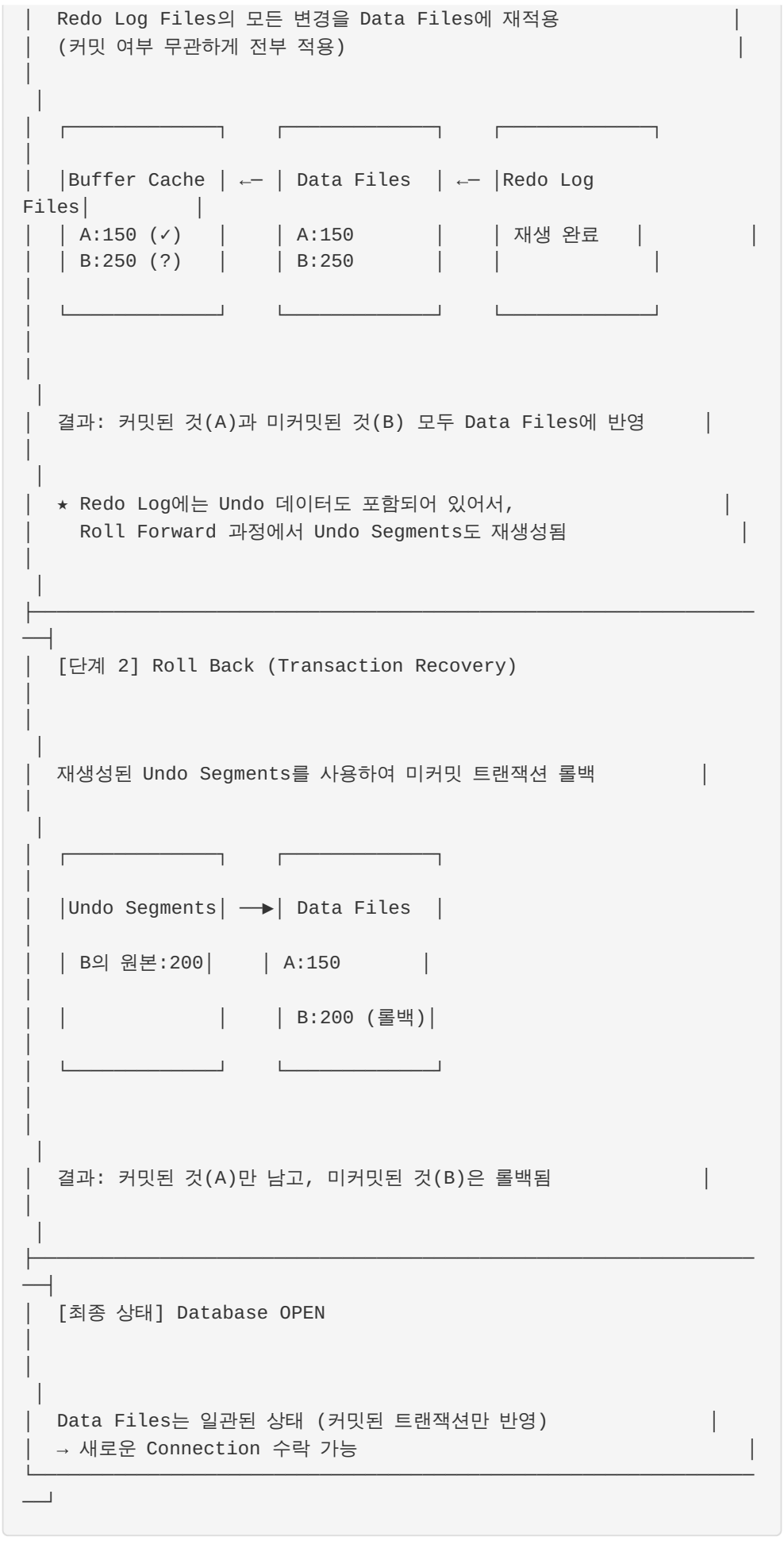
블록 SCN 비교를 통한 멱등성 구현:

```
┌──────────────────────────────────────────────────────────
──┐
|                          멱등성 보장 메커니즘
|
|
 |
|  Redo
Record:                                                  |
|  ┌──────────────────────────────────────────┐
|
|  |  DBA: File 4, Block 12345                |
|
|  |  SCN: 1050                               |
|
|  |  변경: offset 1840을 150으로               |             |
|  └──────────────────────────────────────────┘
|
|
 |
|  Case 1: 블록이 아직 오래된 상태                                  |
|  Block 12345 Header: Block SCN = 900
|
|  → 900 < 1050 이므로 이 블록에 Redo 적용 필요                    |
```

```
|  → offset 1840을 150으로 변경                                    |
|  → Block SCN을 1050으로 업데이트                                 |
|
 |
|  Case 2: 블록이 이미 최신 상태                                   |
|  Block 12345 Header: Block SCN = 1050
|
|  → 1050 >= 1050 이므로 이 Redo는 이미 적용됨                     |
|  → Skip (적용하지 않음)                                          |
|
 |
|  Case 3: 블록이 더 최신 상태                                     |
|  Block 12345 Header: Block SCN = 1100
|
|  → 1100 > 1050 이므로 이 Redo는 이미 적용됨                      |
|  → Skip
|
└──────────────────────────────────────────────────────────────────
──┘
```

5.4 Instance Recovery의 두 단계

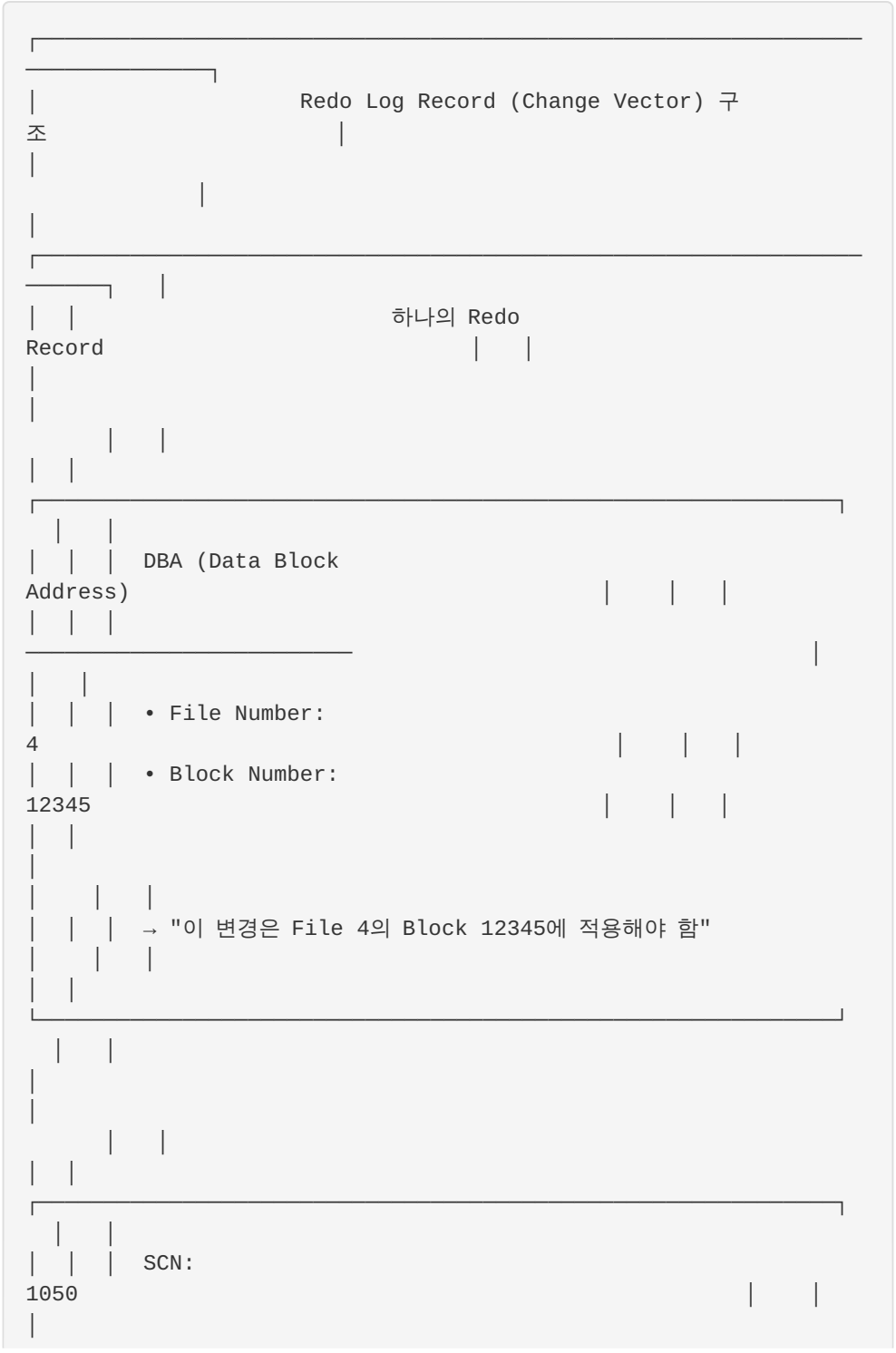
```
┌──────────────────────────────────────────────────────────────────
──┐
|                  Instance Recovery 전체 흐름
|
|
 |
|  [단계 0] 장애 발생 직전 상태                                    |
|
 |
|  ┌──────────────┐    ┌──────────────┐    ┌──────────────┐
|
|  |Buffer Cache |    | Data Files  |    |Redo Log
Files|         |
|  | A:150 (✓)   |    | A:100       |    | A:100→150(✓)|
|
|  | B:250 (?)   |    | B:200       |    | B:200→250(?)|
|
|  └──────────────┘    └──────────────┘    └──────────────┘
|
|     (손실됨)          (오래된 값)        (보존됨)              |
|
 |
|  ✓ = 커밋됨, ? = 미커밋
|
|
 |
├──────────────────────────────────────────────────────────────────
──┤
|  [단계 1] Roll Forward (Cache Recovery)
|
|
 |
```


Redo Log Files의 모든 변경을 Data Files에 재적용
(커밋 여부 무관하게 전부 적용)
Buffer Cache
A:150 (✓)
B:250 (?)
Data Files
A:150
B:250
Redo Log Files
재생 완료
결과: 커밋된 것(A)과 미커밋된 것(B) 모두 Data Files에 반영
★ Redo Log에는 Undo 데이터도 포함되어 있어서,
Roll Forward 과정에서 Undo Segments도 재생성됨
[단계 2] Roll Back (Transaction Recovery)
재생성된 Undo Segments를 사용하여 미커밋 트랜잭션 롤백
Undo Segments
B의 원본:200
Data Files
A:150
B:200 (롤백)
결과: 커밋된 것(A)만 남고, 미커밋된 것(B)은 롤백됨
[최종 상태] Database OPEN
Data Files는 일관된 상태 (커밋된 트랜잭션만 반영)
→ 새로운 Connection 수락 가능

Part 6. Redo Log Record와 블록 매핑

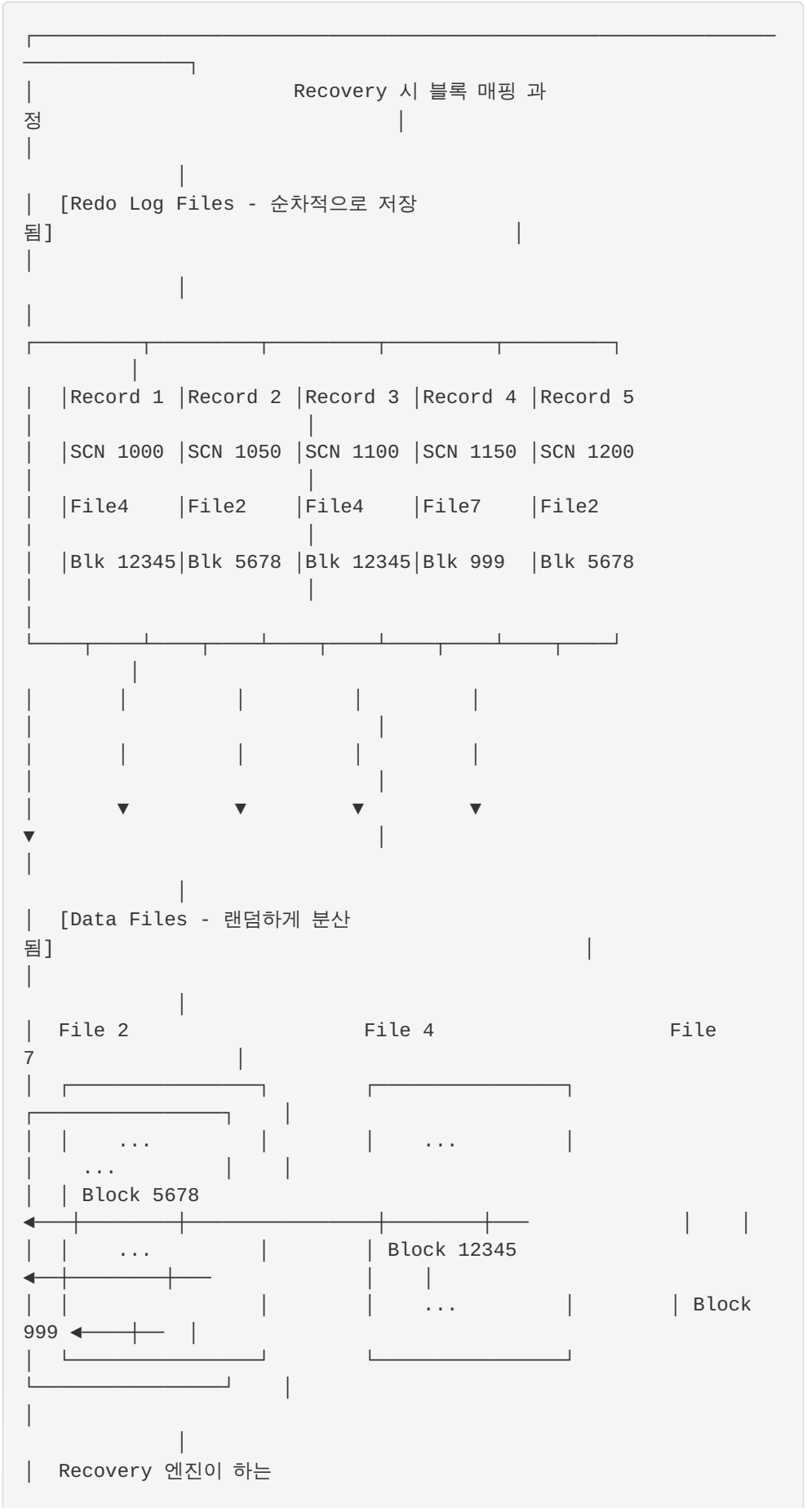
6.1 핵심 질문: 랜덤하게 분산된 블록을 어떻게 Redo Log와 매핑하는가?

Redo Log는 단순히 "A 변경, B 변경"이라는 추상적 기록이 아니다. 각 Record는 **물리적 주소(DBA: Data Block Address)**를 포함한다.




→ "이 변경이 발생한 논리적 시점"
Operation Code: 11.5
(UPDATE)
→ "어떤 종류의 변경인
지"
Offset:
1840
Length: 4
bytes
New Value: 0x00000096
(150)
→ "블록 내 offset 1840 위치에 4바이트를 150으로 변경"
Redo Record는 "자기 완결적(Self-Contained)"이
다.
이 Record 하나만 있으면 어디에 무엇을 어떻게 적용할지 전부 알 수 있
다.

6.2 Recovery 시 매핑 과정

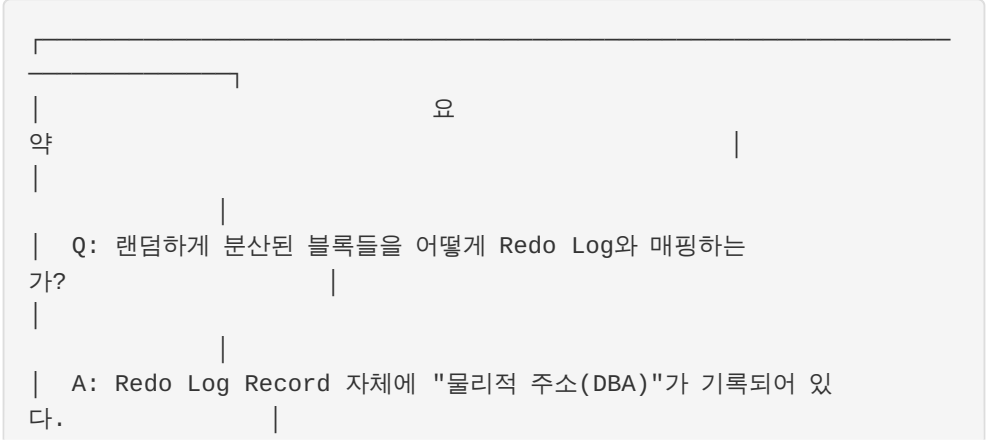


```
일:                                                    |
|
──────────────────────

         |
|  1. Redo Record를 순차적으로 읽
음                                                |
|  2. 각 Record의 DBA(File#, Block#) 추
출                                       |
|  3. 해당 블록을 Data Files에서 Buffer Cache로 읽음 (랜덤 I/
O)              |
|  4. Record의 변경 내용을 블록에 적
용                                          |
└──────────────────────────────────────────────────────────────────
──────────────┘
```

6.3 전체 Recovery 흐름 (정확한 버전)

```
┌──────────────────────────────────────────────────────────────────
──────────────┐
|                      Instance Recovery 상세 흐
름                                |
|
             |
|  [1] Control File에서 Checkpoint Position 읽
기                                  |
|      → SCN
1000
|
|
             |
|  [2] Redo Log Files에서 SCN 1000 이후 Record들을 순차적으로 읽
기             |
|
             |
|      for each Redo Record (SCN >=
1000):                                        |
|
┌──────────────────────────────────────────────────────────────────
──┐   |
|
|
  |   |
|      |  [2-1] Record에서 DBA 추출 (예: File 4, Block
12345)            |   |
|
|
  |   |
|      |  [2-2] 해당 블록이 Buffer Cache에 있는지 확
인                     |   |
|      |        - 없으면: Data Files에서 읽어옴 (Random
Read)             |   |
|      |        - 있으면: 그대로 사
용                                      |   |
|
```

6.4 요약: Redo Log와 블록 매핑



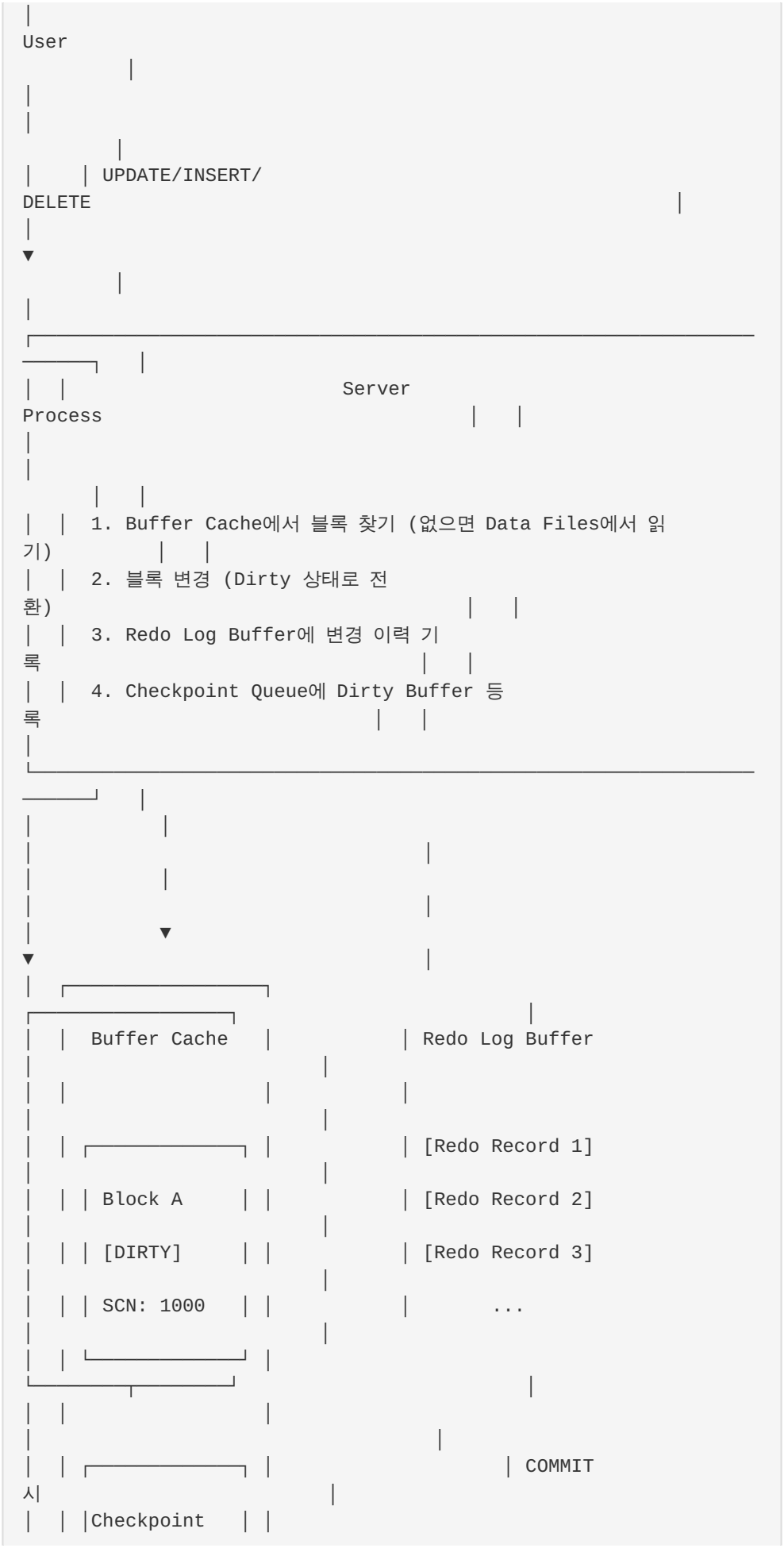
```
|
             |
|     Redo Record =
{                                                                  |
|         DBA: (File#, Block#),    // ← 이것으로 블록 식
별                    |
|         SCN: 변경 시점,          // ← 이것으로 적용 여부 판
단              |
|         Operation: 변경 유
형,                                                  |
|         Offset: 블록 내 위
치,                                                  |
|         NewValue: 새
값                                                       |
|     }
             |
|
             |
|     Recovery
시:                                                                |
|     1. Redo Record 읽
기                                                       |
|     2. DBA로 해당 블록을 Data Files에서 찾아 Buffer Cache로 로
드           |
|     3. Block SCN vs Redo SCN 비교하여 적용 여부 결
정                         |
|     4. 필요시 변경 적
용                                                       |
|
             |
|     → 블록이 디스크 어디에 있든, DBA로 정확히 찾아서 적용 가
능              |
└──────────────────────────────────────────────────────────────────
──────────────┘
```

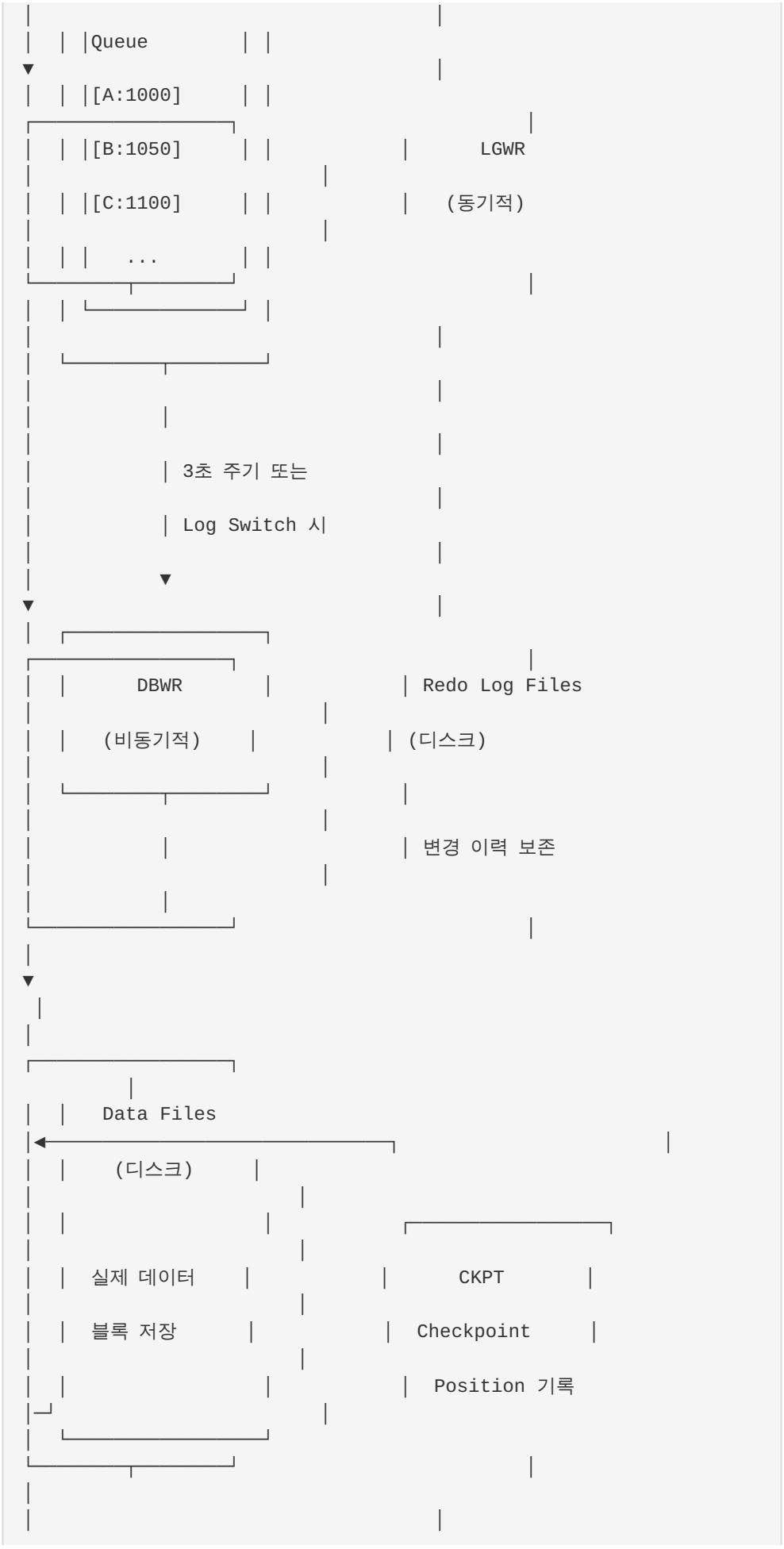
핵심: Redo Log가 "순차적으로 기록된다"는 것은 **Redo Log Files 내에서의 기록 순서**이고, 각 Record가 가리키는 **실제 블록의 위치는 랜덤**하다. Record 내의 DBA가 이 둘을 연결하는 키(Key) 역할을 한다.

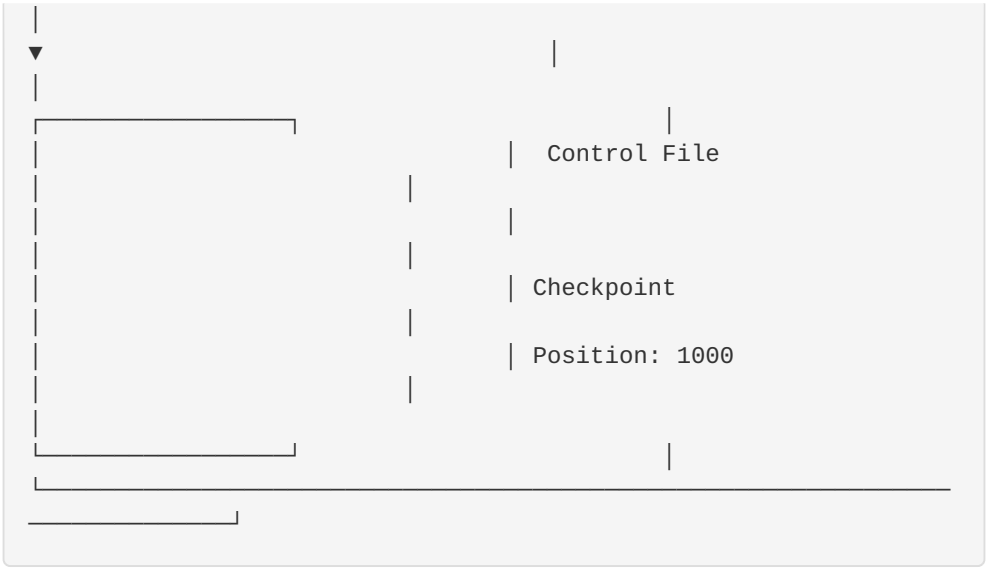
Part 7. 전체 아키텍처 다이어그램

7.1 정상 운영 시 흐름

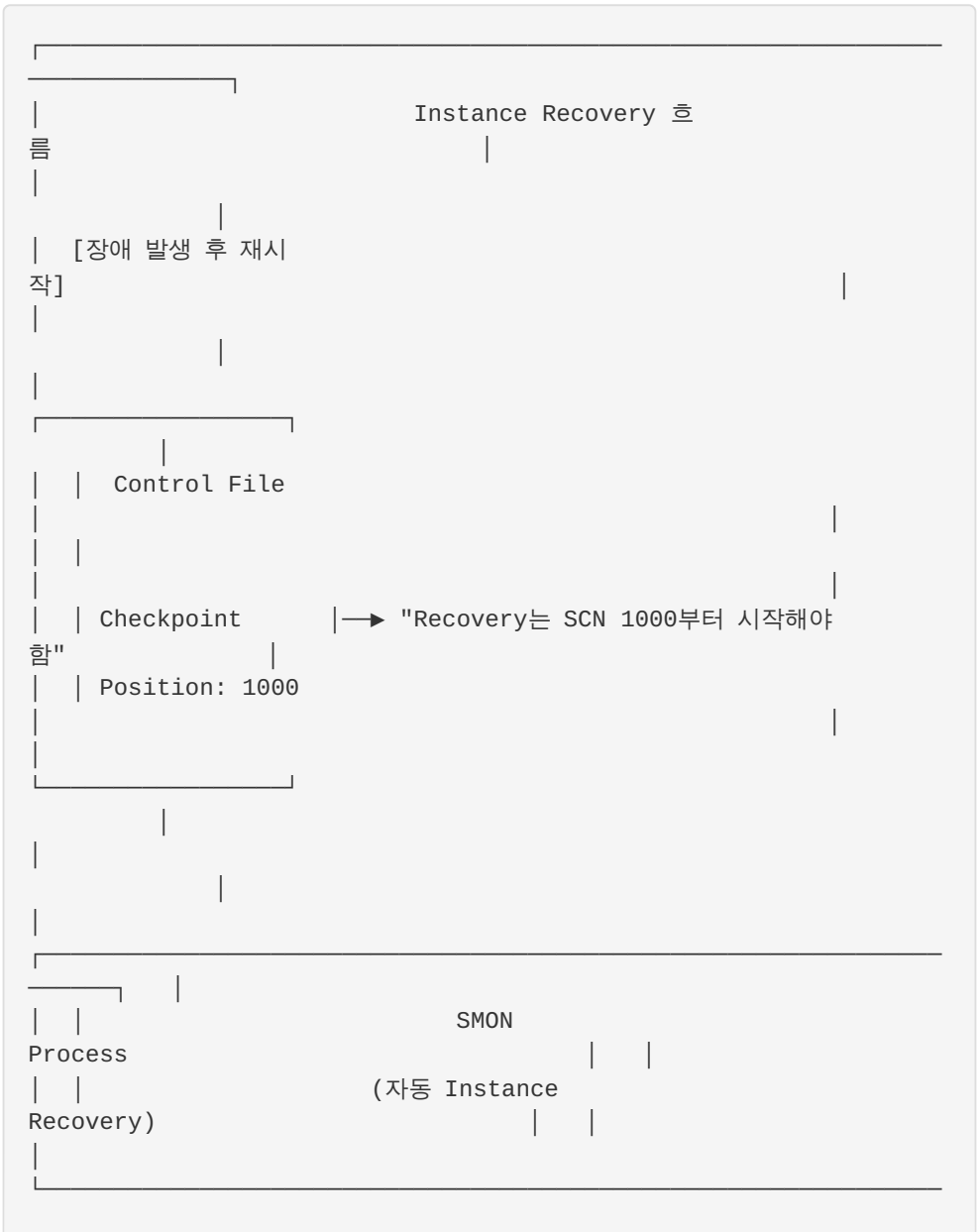
```
┌──────────────────────────────────────────────────────────────────
──────────────┐
|                            정상 운영 시 Checkpoint 흐
름                        |
|
             |
```

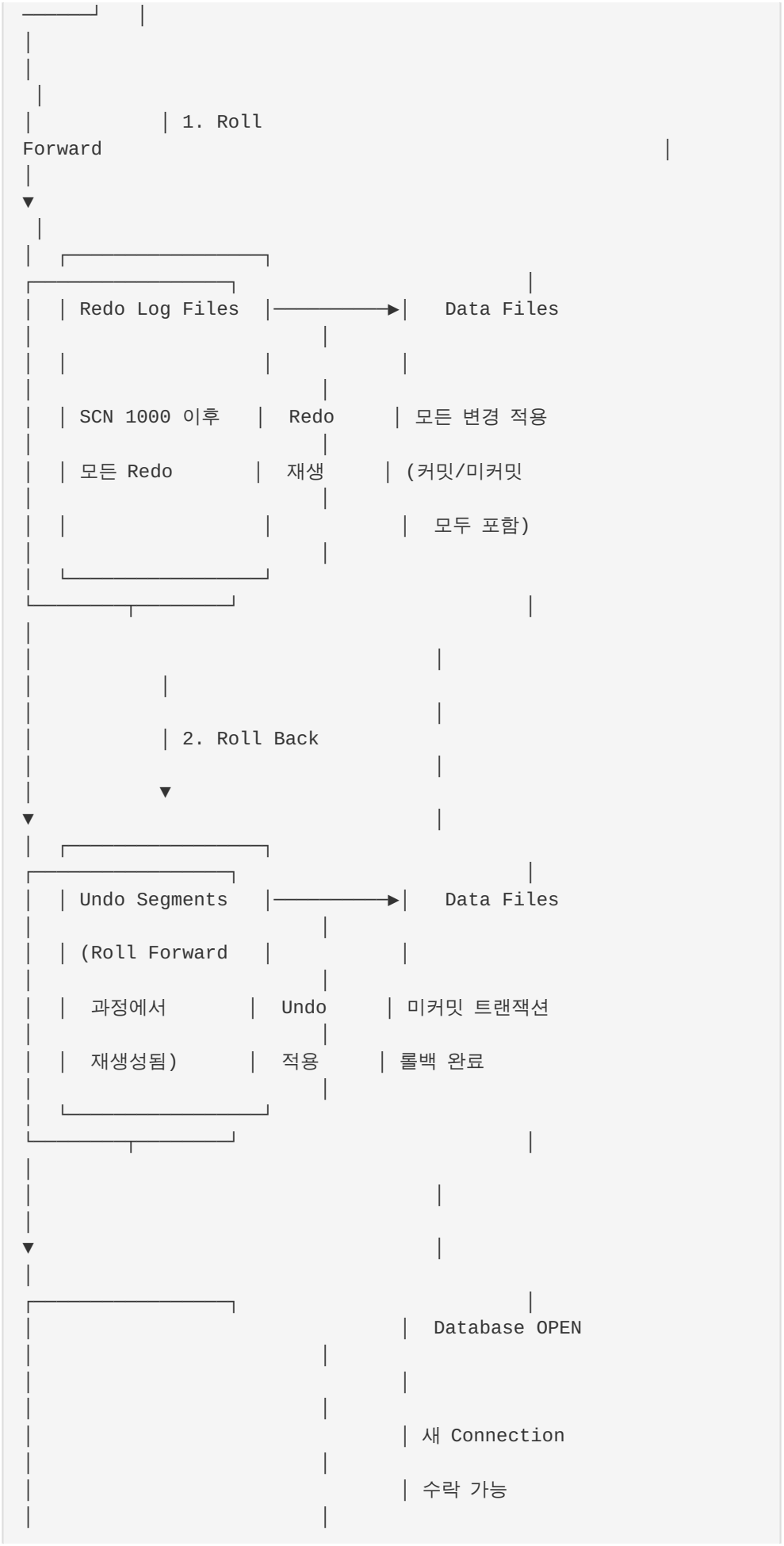

User
UPDATE/INSERT/
DELETE
Server
Process
1. Buffer Cache에서 블록 찾기 (없으면 Data Files에서 읽
기)
2. 블록 변경 (Dirty 상태로 전
환)
3. Redo Log Buffer에 변경 이력 기
록
4. Checkpoint Queue에 Dirty Buffer 등
록
Buffer Cache
Block A
[DIRTY]
SCN: 1000
Redo Log Buffer
[Redo Record 1]
[Redo Record 2]
[Redo Record 3]
...
Checkpoint
COMMIT
시


Queue
[A:1000]
[B:1050]
[C:1100]
...
LGWR
(동기적)
3초 주기 또는
Log Switch 시
DBWR
(비동기적)
Redo Log Files
(디스크)
변경 이력 보존
Data Files
(디스크)
실제 데이터
블록 저장
CKPT
Checkpoint
Position 기록



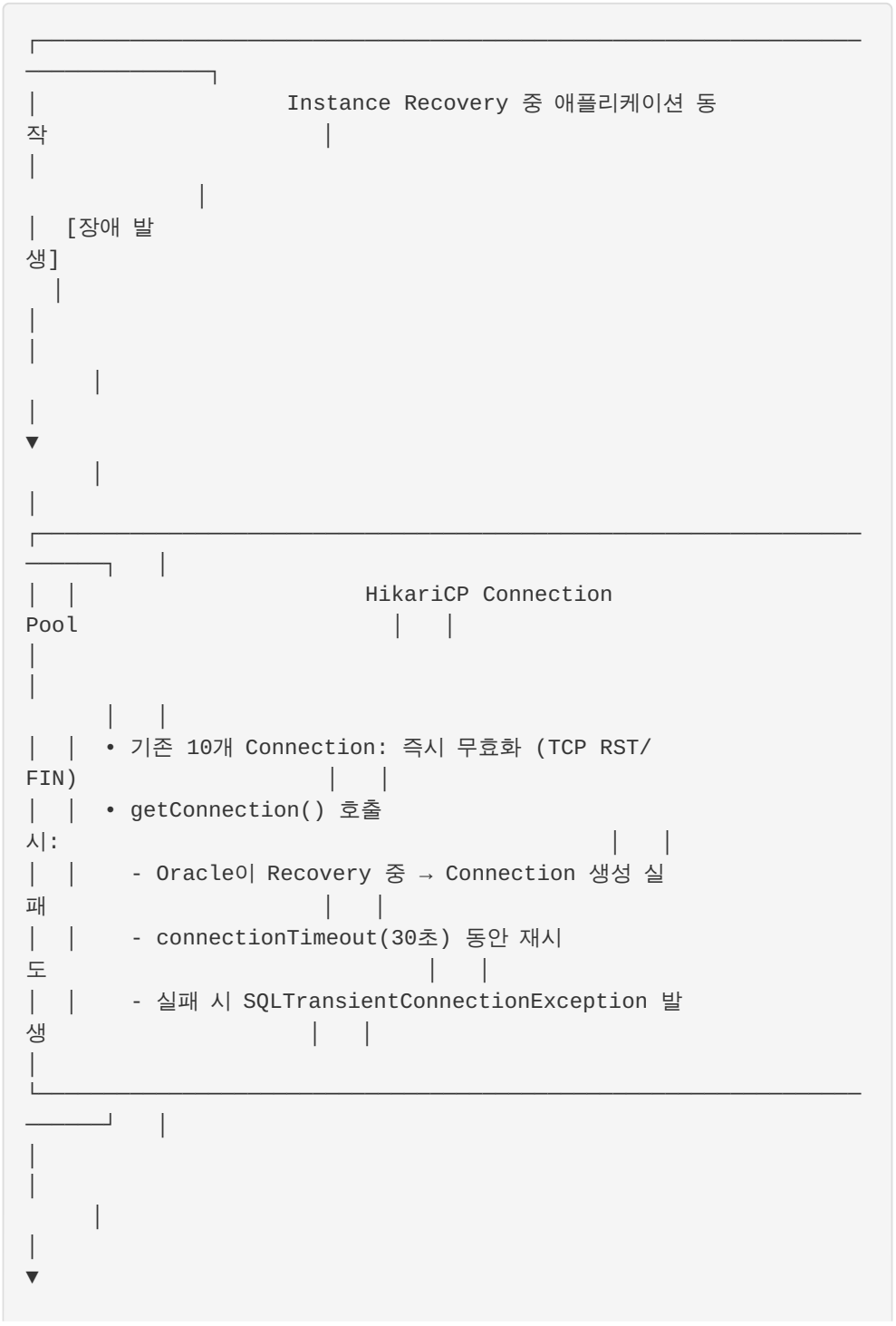
7.2 장애 후 Recovery 흐름




1. Roll Forward
Redo Log Files
Data Files
SCN 1000 이후 모든 Redo
Redo 재생
모든 변경 적용 (커밋/미커밋 모두 포함)
2. Roll Back
Undo Segments
Data Files
(Roll Forward 과정에서 재생성됨)
Undo 적용
미커밋 트랜잭션 롤백 완료
Database OPEN
새 Connection 수락 가능

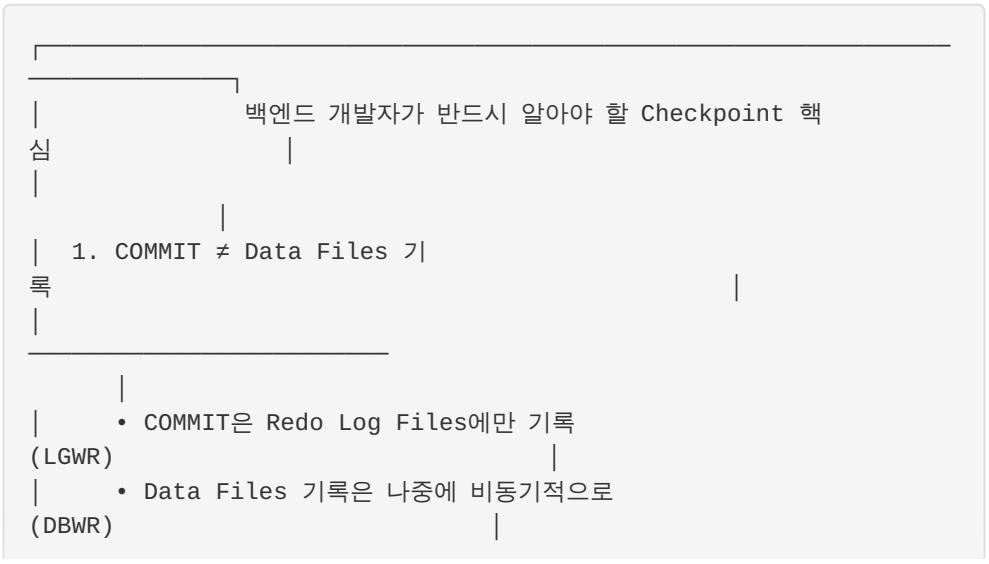
Part 8. 백엔드 개발자 관점의 의의

8.1 애플리케이션에 미치는 영향




Spring Boot Application
try {
// DB 작업
} catch (SQLTransientConnectionException e) {
// 일시적 장애 - 재시도 또는 Circuit Breaker 동작
}
고려사항:
• Recovery 시간은 Checkpoint 주기에 반비례
• Recovery 완료 후에도 10개 Connection 동시 재생성 → 일시적 지연

8.2 핵심 인사이트 정리


백엔드 개발자가 반드시 알아야 할 Checkpoint 핵심
1. COMMIT ≠ Data Files 기록
• COMMIT은 Redo Log Files에만 기록 (LGWR)
• Data Files 기록은 나중에 비동기적으로 (DBWR)


• 이 사이에 장애 발생하면 Recovery로 복구
2. Checkpoint Position = Recovery 시작점
• "가장 오래된 Dirty Buffer의 SCN"으로 정의
• 이 값 미만은 Data Files에 보장됨
• Checkpoint가 자주 발생할수록 Recovery 범위 감소
3. Recovery 중에는 Database 접근 불가
• MOUNT 상태에서 Recovery 수행
• 완료 후 OPEN 상태로 전환
• 애플리케이션은 이 기간 동안 Connection 획득 실패
4. SHUTDOWN ABORT = Checkpoint 미수행
• 다른 종료 방식은 모두 Checkpoint 수행
• ABORT만 Recovery 필요
• 서버 크래시도 ABORT와 동일한 효과
5. Redo Log Record의 자기 완결성
• 각 Record에 DBA(File#, Block#)가 포함됨
• 디스크의 어느 위치에 있는 블록이든 정확히 식별 가능
• Block SCN 비교로 멱등성 보장

8.3 면접 답변 템플릿

질문: "Oracle의 Checkpoint 메커니즘을 설명해주세요."

> "Checkpoint는 메모리(Buffer Cache)와 디스크(Data Files) 간의 동기화 지점을 생성하는 메커니즘입니다.
>
> Oracle은 성능을 위해 변경 사항을 즉시 디스크에 쓰지 않고 메모리에 유지합니다. 대신 변경 이력(Redo Log)만 즉시 디스크에 기록하는 Write-Ahead Logging 전략을 사용합니다. 이로 인해 메모리에는 Dirty Buffer라 불리는, 아직 디스크에 기록되지 않은 변경된 블록들이 존재합니다.
>
> Checkpoint Position은 '가장 오래된 Dirty Buffer의 SCN'으로 정의됩니다. 이 값보다 낮은 SCN의 모든 변경은 Data Files에 기록이 보장됩니다. 장애 발생 시 Instance Recovery는 Checkpoint Position 이후의 Redo Log만 재적용하면 되므로, Checkpoint가 자주 발생할수록 복구 시간이 단축됩니다.
>
> 백엔드 애플리케이션 관점에서 중요한 점은, COMMIT이 완료되어도 실제 데이터가 Data Files에 기록된 것이 아니라 Redo Log에 기록된 것이라는 점입니다. 또한 비정상 종료 후 Recovery 기간 동안에는 새 Connection을 받을 수 없으므로, HikariCP의 connectionTimeout 설정과 예외 처리가 중요합니다."

질문: "Instance Recovery가 어떻게 동작하는지 설명해주세요."

> "Instance Recovery는 비정상 종료 후 데이터베이스 일관성을 복원하는 자동 프로세스입니다. Recovery 범위는 Checkpoint Position에 의해 결정됩니다.
>
> 첫 번째 단계인 Roll Forward에서는 Checkpoint Position 이후의 모든 Online Redo Log를 Data Files에 재적용합니다. 이때 커밋 여부와 관계없이 모든 변경이 적용됩니다. 일부 변경은 장애 전에 이미 Data Files에 기록되어 있을 수 있지만, Oracle은 개별 추적 대신 전체 재적용 전략을 사용합니다. Redo 적용이 멱등(idempotent)하기 때문에 중복 적용해도 결과가 동일합니다.
>
> 두 번째 단계인 Roll Back에서는 Roll Forward 과정에서 재생성된 Undo Segments를 사용하여 커밋되지 않은 변경을 롤백합니다.
>
> 애플리케이션 관점에서 중요한 것은 Recovery가 완료될 때까지 Database가 OPEN 상태가 아니라는 점입니다. HikariCP는 이 기간 동안 Connection 획득에 실패하고, connectionTimeout 이후 SQLTransientConnectionException을 발생시킵니다."

Part 9. 학습 범위 경계

영역	백엔드 개발자 (본 문서 범위)	DBA 영역 (본 문서 범위 외)
Checkpoint 개념	존재 이유, 동작 원리, Recovery와의 관계	파라미터 튜닝, AWR 분석
프로세스	DBWR, CKPT, LGWR의 역할	다중 DBWR 구성, 프로세스 모니터링
Checkpoint Queue	개념과 순서 기반 기록 원리	VBH, VINSTANCE_RECOVERY 뷰 분석
Recovery	범위 결정 원리, 애플리케이션 영향	병렬 Recovery 튜닝, FAST_START_MTTR_TARGET 설정
종료 모드	ABORT만 Recovery 필요하다는 사실	각 모드별 운영 판단 기준
Redo Log Record	DBA를 통한 블록 식별 원리, 멱등성 개념	LogMiner 분석, Archive Log 관리

Part 10. 용어 정리

용어	정의	위치
Buffer Cache	디스크에서 읽어온 데이터 블록을 메모리에 임시 저장하는 영역	메모리 (SGA)
Dirty Buffer	Buffer Cache 내에서 변경되었지만 아직 Data Files에 기록되지 않은 블록의 상태	Buffer Cache 내부
Redo Log Buffer	변경 이력을 임시 저장하는 메모리 영역	메모리 (SGA)
Redo Log Files	Redo Log Buffer의 내용이 디스크에 영구 저장된 파일	디스크
Data Files	실제 데이터 블록이 저장된 파일	디스크
Control File	Database 메타데이터(Checkpoint Position 포함)가 저장된 파일	디스크

용어	정의	위치
Checkpoint Queue	Dirty Buffer를 변경 시점(SCN) 순서로 관리하는 자료구조	메모리
Checkpoint Position	가장 오래된 Dirty Buffer의 SCN, Recovery 시작점	Control File (디스크)
SCN	System Change Number, 변경 순서를 나타내는 논리적 타임스탬프	블록 헤더, Control File 등
DBA	Data Block Address, File Number + Block Number로 블록의 물리적 주소 식별	Redo Record, 블록 헤더
DBWR	Database Writer, Dirty Buffer를 Data Files에 기록하는 백그라운드 프로세스	OS 프로세스
CKPT	Checkpoint Process, Checkpoint Position을 Control File에 기록하는 프로세스	OS 프로세스
LGWR	Log Writer, Redo Log Buffer를 Redo Log Files에 기록하는 프로세스	OS 프로세스
SMON	System Monitor, Instance Recovery를 수행하는 백그라운드 프로세스	OS 프로세스

부록: 무결성 보장 구조의 핵심 원리

CKPT 프로세스의 불변식(Invariant)

```
┌──────────────────────────────────────────────────────────────────────────────
──────────────┐
│                          설계의 핵심 보
장                                          │
│
             │
│  CKPT 프로세스의 규
칙:                                                          │
│
──────────────────────
         │
│  "Control File에 Checkpoint Position을 기록하는 시점
에,                        │
│   그 SCN 미만의 모든 Dirty Buffer는 이미 Data Files에 기록 완료되어
있다"    │
```

```
|   모든 변경은 반드시 Data Files에 있
다"                                        |
|
                |
|  이 불변식이 깨지지 않도록 CKPT는 DBWR의 기록 완료를 확인한 후에
만           |
|  Control File을 업데이트한
다.                                                |
└──────────────────────────────────────────────────────────────────
──────────────┘
```

복구 가능성의 보장

```
메모리의 Checkpoint Queue가 손실되어도:

• 어디서부터: Control File의 Checkpoint Position이 알려줌
• 무엇을: Redo Log Files가 제공함

이 두 가지가 디스크에 영구 저장되어 있기 때문에 복구가 가능하다.
```

문서 작성일: 2025년 6월
참고: Oracle Database 19c Concepts Guide, "Importance of Checkpoints for Instance Recovery" 섹션 기반