HikariCP 내부 동작 원리

1. 왜 HikariCP 내부 구조를 이해해야 하는가 (Why)

Wherehouse 프로젝트에서 관측된 HikariCP Pool 상태를 보면:

메트릭	Origin	Chunk	Bulk Fetch
Active	10 (최대)	10	0
Idle	0	0	10
Waiting	9 (최대)	0~19	0

Image 1의 Origin 코드에서 Active=10, Waiting=9라는 수치가 나왔다. 이 수치가 왜 나왔는지, 이것이 실제로 병목의 근거인지를 판단하려면 HikariCP가 내부적으로 커넥션을 어떻게 관리하는지 알아야 한다.

피드백에서 요구하는 핵심 판단 능력:

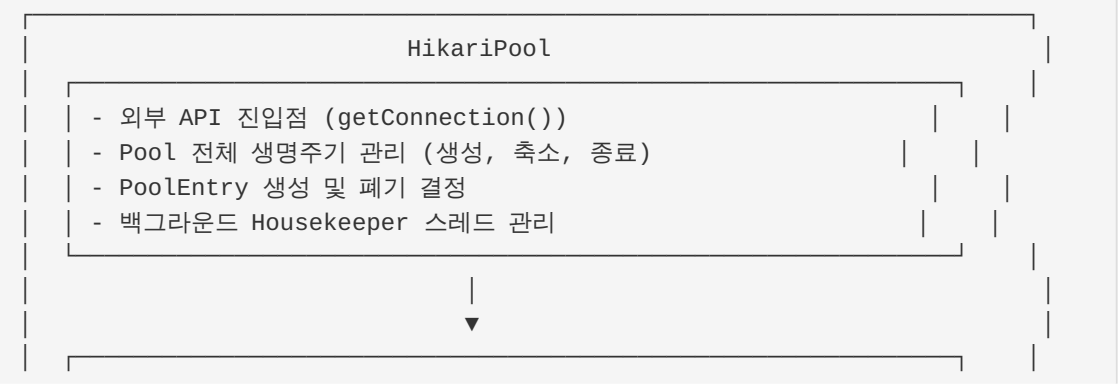
> "커넥션 풀이 25번 acquire/release를 반복하면서 지연이 발생했는지 판단"

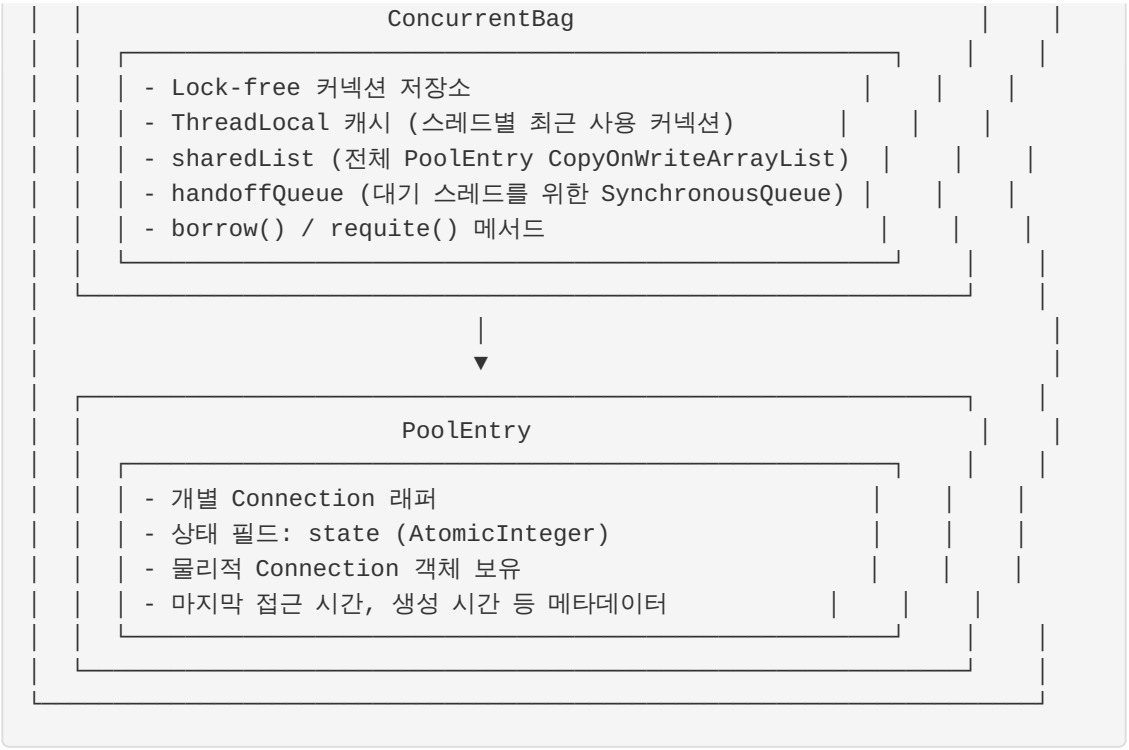
이 판단을 하려면 단순히 "Active가 높으면 병목"이라는 피상적 이해가 아니라, HikariCP의 상태 전이 메커니즘과 대기 큐 진입 조건을 정확히 이해해야 한다.

2. HikariCP 아키텍처 개요 (How)

2.1 핵심 컴포넌트 3개의 역할 분담

HikariCP는 세 가지 핵심 클래스로 구성된다:

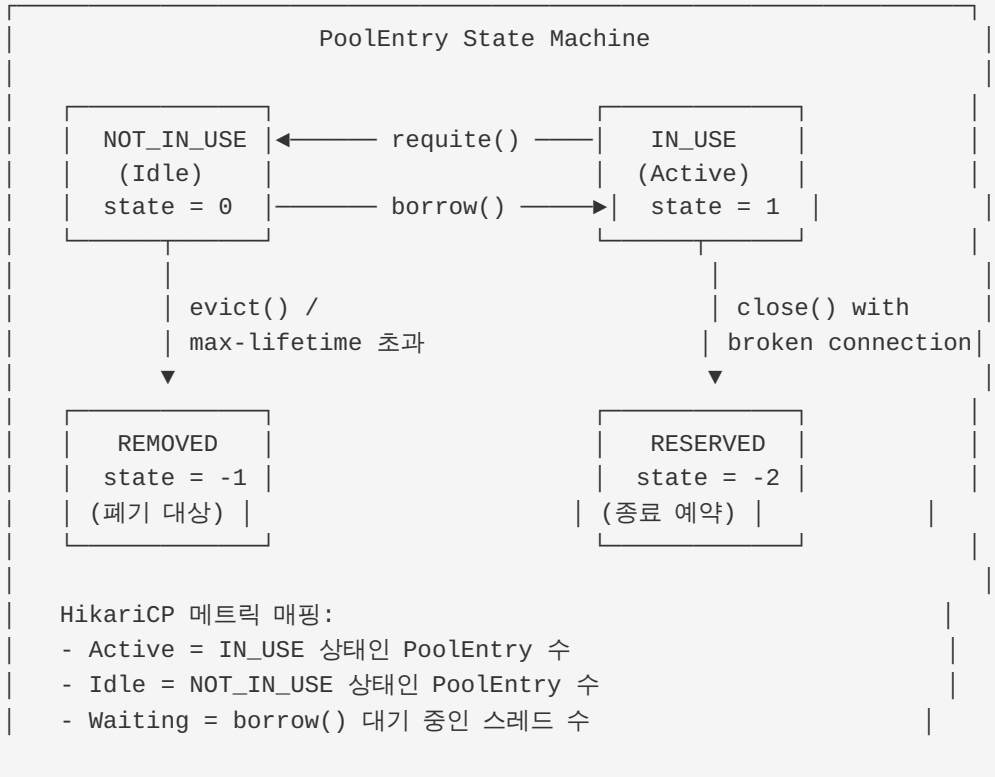




설계 의도: HikariPool은 정책 결정(언제 커넥션을 만들고 폐기할지), ConcurrentBag은 고성능 자료구조(어떻게 빠르게 커넥션을 찾을지), PoolEntry는 개별 커넥션의 상태 관리(현재 사용 가능한지)를 분리했다.

2.2 PoolEntry의 상태 전이 (State Transition)

PoolEntry는 4가지 상태를 가지며, 이 상태가 HikariCP 메트릭의 근간이 된다:



```
|   - Total = sharedList.size() (REMOVED 제외)                          |
└──────────────────────────────────────────────────────────────────────┘
```

상태 전이의 원자성: state 필드는 AtomicInteger로 구현되어 있어, compareAndSet(NOT_IN_USE, IN_USE) 연산이 원자적으로 수행된다. 이 덕분에 Lock 없이도 동시성 문제가 발생하지 않는다.

2.3 ConcurrentBag.borrow() - 커넥션 획득 흐름

getConnection() 호출 시 실제로 실행되는 핵심 로직이다:

```
┌──────────────────────────────────────────────────────────────────────────┐
│                    ConcurrentBag.borrow(timeout) 흐름                    │
│                                                                          │
│  1단계: ThreadLocal 검색 (가장 빠름)                                      │
│  ┌──────────────────────────────────────────────────────────────────┐    │
│  │ ThreadLocal<List<PoolEntry>> threadList                          │    │
│  │                                                                  │    │
│  │ for (PoolEntry entry : threadList.get()) {                       │    │
│  │     if (entry.compareAndSet(NOT_IN_USE, IN_USE)) {               │    │
│  │         return entry;  // ★ Cache Hit: ~100ns                    │    │
│  │     }                                                            │    │
│  │ }                                                                │    │
│  │                                                                  │    │
│  │ 설계 의도: 동일 스레드가 반복적으로 커넥션을 사용할 때,          │    │
│  │           같은 커넥션을 재사용하면 CPU 캐시 효율 극대화          │    │
│  └──────────────────────────────────────────────────────────────────┘    │
│                              │                                           │
│                              ▼ (ThreadLocal에 유휴 커넥션 없음)          │
│                                                                          │
│  2단계: SharedList 순회 검색                                             │
│  ┌──────────────────────────────────────────────────────────────────┐    │
│  │ CopyOnWriteArrayList<PoolEntry> sharedList                       │    │
│  │                                                                  │    │
│  │ for (PoolEntry entry : sharedList) {                             │    │
│  │     if (entry.compareAndSet(NOT_IN_USE, IN_USE)) {               │    │
│  │         return entry;  // ★ Shared Hit: ~1μs                     │    │
│  │     }                                                            │    │
│  │ }                                                                │    │
│  │                                                                  │    │
│  │ 설계 의도: 다른 스레드가 반환한 커넥션을 바로 사용               │    │
│  └──────────────────────────────────────────────────────────────────┘    │
│                              │                                           │
│                              ▼ (SharedList에도 유휴 커넥션 없음)         │
│                                                                          │
│  3단계: HandoffQueue 대기 진입                                           │
│  ┌──────────────────────────────────────────────────────────────────┐    │
│  │ SynchronousQueue<PoolEntry> handoffQueue                         │    │
│  │                                                                  │    │
│  │ waiters.incrementAndGet();  // ★ Waiting 카운터 증가             │    │
│  │                                                                  │    │
│  │ PoolEntry entry = handoffQueue.poll(timeout, NANOSECONDS);       │    │
│  │                                                                  │    │
│  │ if (entry == null) {                                             │    │
│  │     throw SQLTransientConnectionException(                       │    │
```

```
|   |          "Connection is not available, request timed out");      |   |
|   | }                                                                 |   |
|   |                                                                   |   |
|   | waiters.decrementAndGet();  // Waiting 카운터 감소              |   |
|   | return entry;                                                     |   |
|   |                                                                   |   |
|   | 설계 의도: 다른 스레드가 requite()로 반환하는 커넥션을           |   |
|   |           직접 전달받음 (Producer-Consumer 패턴)                |   |
|   └──────────────────────────────────────────────────────────────────┘   |
└──────────────────────────────────────────────────────────────────────────┘
```

2.4 ConcurrentBag.requite() - 커넥션 반환 흐름

connection.close() 호출 시 (실제로는 ProxyConnection.close()) 실행되는 로직:

```
┌──────────────────────────────────────────────────────────────────────────┐
│                   ConcurrentBag.requite(entry) 흐름                    │
│                                                                          │
│  1단계: 상태 전이                                                        │
│  ┌──────────────────────────────────────────────────────────────────┐   │
│  │ entry.setState(NOT_IN_USE);  // IN_USE → NOT_IN_USE             │   │
│  └──────────────────────────────────────────────────────────────────┘   │
│                              │                                           │
│                              ▼                                           │
│                                                                          │
│  2단계: 대기 스레드 확인 및 직접 전달                                    │
│  ┌──────────────────────────────────────────────────────────────────┐   │
│  │ if (waiters.get() > 0) {                                        │   │
│  │     // ★ 대기 중인 스레드가 있으면 직접 전달                    │   │
│  │     handoffQueue.offer(entry);                                  │   │
│  │ }                                                               │   │
│  │                                                                 │   │
│  │ 설계 의도: Waiting 스레드가 SharedList를 다시 순회하지 않고     │   │
│  │           SynchronousQueue를 통해 즉시 커넥션을 받음            │   │
│  └──────────────────────────────────────────────────────────────────┘   │
│                              │                                           │
│                              ▼                                           │
│                                                                          │
│  3단계: ThreadLocal 캐시 갱신                                            │
│  ┌──────────────────────────────────────────────────────────────────┐   │
│  │ threadList.get().add(entry);                                    │   │
│  │                                                                 │   │
│  │ 설계 의도: 다음에 같은 스레드가 borrow()하면                    │   │
│  │           이 커넥션을 우선적으로 재사용                         │   │
│  └──────────────────────────────────────────────────────────────────┘   │
└──────────────────────────────────────────────────────────────────────────┘
```

3. Wherehouse 프로젝트 데이터와의 매핑 (So What)

3.1 Origin 코드에서 Waiting=9가 발생한 메커니즘

Image 1의 Origin 코드 상황을 HikariCP 내부 흐름으로 분석하면:

```
┌──────────────────────────────────────────────────────────────────────┐
│ Origin 코드 실행 시나리오 (동시 요청 50개, Pool Size 10)              │
│                                                                      │
│ 시점 T0: 50개 스레드가 거의 동시에 calculateCharterPropertyScores() 호출│
│                                                                      │
│  Thread-1 ~ Thread-10:                                               │
│  ┌──────────────────────────────────────────────────────────────┐    │
│  │ @Transactional 시작 → borrow() → 성공 (NOT_IN_USE→IN_USE)    │    │
│  │ → 25회 쿼리 실행 시작 (Connection 점유 유지)                  │    │
│  └──────────────────────────────────────────────────────────────┘    │
│                                                                      │
│  Thread-11 ~ Thread-50:                                              │
│  ┌──────────────────────────────────────────────────────────────┐    │
│  │ @Transactional 시작 → borrow() 시도                          │    │
│  │                                                              │    │
│  │ 1단계: ThreadLocal 검색 → 없음 (첫 요청이므로)                │    │
│  │ 2단계: SharedList 순회 → 모두 IN_USE 상태                     │    │
│  │ 3단계: HandoffQueue 대기 진입                                 │    │
│  │        ↓                                                     │    │
│  │   waiters.incrementAndGet() → Waiting 카운터 증가             │    │
│  │        ↓                                                     │    │
│  │   handoffQueue.poll(30초, ...) → 대기 시작                    │    │
│  └──────────────────────────────────────────────────────────────┘    │
│                                                                      │
│ 관측 결과:                                                           │
│ - Active = 10 (Thread-1~10이 모두 점유 중)                           │
│ - Idle = 0 (모든 커넥션이 IN_USE)                                    │
│ - Waiting = 최대 40까지 증가 가능 (Image 1에서 9 관측)               │
│                                                                      │
│ ★ 핵심 병목 원인:                                                    │
│   Thread-1~10이 25회 쿼리를 모두 완료할 때까지 Connection을 반환하지  │
│   않기 때문에, Thread-11~50은 최소 (25회 쿼리 × 평균 쿼리 시간) 동안  │
│   HandoffQueue에서 대기해야 함                                       │
└──────────────────────────────────────────────────────────────────────┘
```

3.2 Chunk 방식에서 개선된 이유

Image 2의 Chunk 방식(1,000개 단위 분할):

```
┌──────────────────────────────────────────────────────────────────────┐
│ Chunk 방식 실행 시나리오                                             │
│                                                                      │
│ 변화 요인:                                                           │
│ - 25회 개별 쿼리 → 9회 Chunk 쿼리 (약 64% 감소)                      │
│ - 각 쿼리의 처리 시간은 증가하나, 총 Network I/O 왕복 횟수 감소      │
```

```
┌──────────────────────────────────────────────────────────────────────┐
│ 결과:                                                                │
│ - Connection Holding Time: 약 35% 감소                               │
│ - Waiting 스레드의 대기 시간 단축 → Timeout 발생 확률 감소           │
│ - 하지만 여전히 동시성 피크에서 Waiting 발생 (최대 19 관측)          │
│                                                                      │
│ 관측 데이터 해석:                                                    │
│ - Waiting = 0~19 범위로 변동                                         │
│ - 쿼리 수 감소로 Connection 반환이 빨라져 HandoffQueue 통한          │
│   커넥션 전달 빈도 증가                                              │
└──────────────────────────────────────────────────────────────────────┘
```

3.3 Bulk Fetch에서 Waiting=0이 된 이유

Image 3의 Bulk Fetch 방식:

```
┌──────────────────────────────────────────────────────────────────────┐
│ Bulk Fetch 방식 실행 시나리오                                        │
│                                                                      │
│ 변화 요인:                                                           │
│ - 25회 쿼리 → 1회 Bulk 쿼리 (96% 감소)                               │
│ - Connection Holding Time: 약 70% 감소 (343ms → 1,748ms이나          │
│   전체 요청 처리 시간 대비 비율 감소)                                │
│                                                                      │
│ 결과:                                                                │
│ - Active = 0~10, Idle = 0~10 (요청 패턴에 따라 변동)                 │
│ - Waiting = 0 (★ 대기 큐 미발생)                                     │
│                                                                      │
│ 메커니즘:                                                            │
│ - 각 스레드의 Connection 점유 시간이 충분히 짧아짐                   │
│ - requite() 호출 시점에 waiters.get() = 0이므로                      │
│   HandoffQueue를 통한 직접 전달 없이 SharedList에 바로 복귀          │
│ - 다음 스레드가 borrow() 시 SharedList에서 즉시 획득                 │
└──────────────────────────────────────────────────────────────────────┘
```

4. 면접 대응용 핵심 질문과 답변

Q1: "maximum-pool-size를 10에서 20으로 늘리면 문제가 해결되는가?"

답변 프레임워크:

결론: 근본적 해결이 아니며, 오히려 부작용이 발생할 수 있다.

이유:

1. **DB 서버 관점:**
2. Oracle의 동시 세션 수 증가 → PGA 메모리 사용량 증가
3. 동시 실행 쿼리 증가 → Latch 경합, Buffer Busy Wait 증가 가능성

4. **애플리케이션 관점:**
5. Pool Size만 늘리면 더 많은 스레드가 동시에 DB를 호출
6. N+1 패턴이 해결되지 않았으므로 DB 부하만 증가
7. **적절한 Pool Size 결정 공식 (HikariCP 공식 가이드):** `connections = (core_count * 2) + effective_spindle_count`
8. SSD 기준: connections ≈ core_count * 2
9. 예: 4코어 서버 → 8~10개가 적정

올바른 해결: - Pool Size 증가 전에 Connection Holding Time 최적화 - N+1 해결 후 Pool Size 조정 검토

Q2: "N+1 패턴이 커넥션 풀 고갈을 유발하는 메커니즘은?"

답변 프레임워크:

핵심 인과관계: @Transactional 메서드 내에서 N+1이 발생하면, 단일 트랜잭션이 Connection을 점유한 채로 N+1회 쿼리를 순차 실행한다.

구체적 흐름:

1. @Transactional 시작 → HikariCP.borrow() → Connection 1개 획득
2. 메서드 내부에서 N+1 쿼리 발생 (25회 가정)
3. 25회 쿼리가 완료될 때까지 Connection 반환 불가
4. 동시 요청이 Pool Size를 초과하면:
5. 남은 요청들은 HandoffQueue 대기 진입
6. waiters 카운터 증가 → Waiting 메트릭 상승
7. connection-timeout 초과 시:
8. SQLTransientConnectionException 발생
9. 500 Internal Server Error 응답

Wherehouse 프로젝트 실측: - Origin: 25회 쿼리 × 평균 13.4ms/쿼리 = 335ms 점유 - Bulk Fetch: 1회 쿼리 × 1,748ms = 1,748ms 점유 (단일 쿼리 시간은 증가했으나, 동시성 관점에서 개선)

5. 세 클래스별 학습 범위 판단

```
┌──────────────┬──────────────────────────────────┬──────────────────────────────┐
│              │ • borrow() 3단계 흐름             │ • Lock-free CAS 알고리즘  │
│ ConcurrentBag│   (ThreadLocal→SharedList→Queue) │   상세 구현                  │
│              │ • requite() 반환 흐름             │ • CopyOnWriteArrayList    │
│              │ • Waiting 카운터 증감 시점        │   선택 이유                  │
│              │                                  │ • SynchronousQueue 내부   │
├──────────────┼──────────────────────────────────┼──────────────────────────────┤
│              │ • 4가지 상태 정의와 전이 조건     │ • Javassist 프록시 생성    │
│  PoolEntry   │   (NOT_IN_USE ↔ IN_USE)          │ • FastList 최적화 구현     │
│              │ • 상태와 메트릭 매핑              │ • Connection 유효성 검증  │
│              │   (IN_USE=Active, NOT_IN_USE=Idle)│   상세 로직                │
└──────────────┴──────────────────────────────────┴──────────────────────────────┘
```

결론: 학습해야 할 것과 하지 말아야 할 것

학습해야 할 것 (면접/포트폴리오 설명에 직접 사용)

항목	학습 목적	프로젝트 연관성
PoolEntry 상태 전이	Active/Idle 메트릭의 정확한 의미	"Active=10은 모든 PoolEntry가 IN_USE 상태"
borrow() 3단계 흐름	Waiting 발생 조건 설명	"SharedList에 NOT_IN_USE가 없으면 HandoffQueue 대기 진입"
requite() 반환 흐름	커넥션 반환 시 대기 스레드에 전달되는 메커니즘	"Waiting>0이면 HandoffQueue로 직접 전달"
connection-timeout	에러율 15% 원인 설명	"30초 대기 후 SQLTransientConnectionException"

학습하지 말아야 할 것 (시간 낭비)

항목	이유
CAS 연산의 CPU 수준 동작	라이브러리 개발자 영역, 면접에서 물어보지 않음
Javassist 프록시 생성	HikariCP 내부 최적화, 포트폴리오와 무관
FastList vs ArrayList	마이크로 최적화, 설명할 맥락 없음
Housekeeper 스레드	idle-timeout/max-lifetime 관련, 현재 프로젝트에서 해당 설정 미사용

6. 참조

- HikariCP Dead lock에서 벗어나기 (이론편) - 우아한형제들 기술블로그