맹준영_12장

목표

- 대규모 엑세스를 감당해야하는 애플리케이션은 RDB를 이용해서는 성능의 한계를 직면한다.
- 데이터 모델을 유지하면서 물리적인 한계에 대응하는 법을 알아본다.

▼ 12.1 캐시라는 개념

- 캐시가 사용되는 곳
  - CPU 캐시 메모리
  - TLB(Translation Lookaside Buffer, 가상 메모리의 논리 주소와 물리 주소 매핑하는 캐시 메모리)
  - 디스크 캐시, 파일 시스템 캐시
  - 브라우저 콘텐츠 캐시, DNS 캐시
  - DB 버퍼 풀 등
- **캐시는 확실한 장점과 단점을 갖기 때문에 확실하게 성능을 향상시킬 수 있을 때 적용하는 것이 바람직하다.**

캐시의 장점

- **캐시의 본질은 비용이 많이드는 작업을 비용이 낮은 동일한 행위로 처리하는 작업**
- 높은 비용 처리의 생략으로 수백 ,수천 배의 성능 향상도 가능하다.

캐시의 단점

- 캐시 레이어가 추가됨으로써 시스템의 동작이 복잡해진다.
- 이로 인한 **유지보수성이 증가**한다.

DB 응용프로그램에서의 캐시

- **시스템의 복잡성은 증가하지만, 대체할 수 있는 낮은 비용의 처리를 지속함으로써 발생하는 높은 비용의 처리를 가능한 실행되지 않게 하는 것**이다.
- DB 응용프로그램에서 캐시의 대상은 **테이블 데이터, 데이터의 시간적, 공간적 지역성이 존재하는 경우 캐시의 효과가 증대**

캐시는 어디까지나 캐시

- 캐시는 어디까지나 캐시로 다뤄야한다.
- 데이터를 메모리 상에 캐시하는 경우 시스템 문제로 인해 메모리 상 데이터가 손실될 수 있다.
- 캐시 데이터는 메모리에 존재하지만, 실제 데이터는 디스크 상에 적재되므로 실제로는 손실되지 않는다.
- **캐시 중심 설계가 아닌 캐시가 사라져도 기존 시스템에 영향을 주지 않도록 실제 데이터와 캐시는 명확하게 구별해야한다.**

캐시로 사용하기 위한 요건

1. **캐시에 대한 쿼리가 캐시에 없는 경우 대처법**
2. **캐시에 없는 경우 DB에 쿼리하는 방법**
3. **캐시에 없을 때 새롭게 캐시에 추가하는 방법**
4. **데이터가 갱신되는 경우 캐시와 동기화하는 방법 → ex) 트리거를 통한 동기화**
5. **데이터가 모두 손실되는 경우 1회 재구성 하는 방법**
6. **캐시의 정합성을 확인하는 방법**

- 위의 요건들은 **캐시 설계 시 캐시와 실제 DB간 레이턴시를 어느정도 허용하냐에 따라 방법이 달라진다.**

캐시하면 안되는 데이터

- **트랜잭션에서 참조하는 데이터**
  - 트랜잭션 내에서 참조하는 데이터와 갱신하는 데이터는 완전히 동기화되어야 한다.
  - 캐시와 실제 데이터 사이의 레이턴시가 발생할 수 있으므로 정합성이 맞지 않는 경우가 존재한다.

캐시할 수 있는 데이터

- **빈번하게 엑세스하는 데이터**
- **엑세스에 편차가 있는 데이터**
- **장기간 변경될 예정이 없는 데이터**
- **표시하는 것이 목적인 데이터**
- 이러한 데이터의 예시
  - 사용자 인증을 위한 데이터, 블로그나 컨텐츠 관리 시스템 같은 페이지 데이터, 검색용 데이터

▼ 12.2 캐시의 구축 방법

🔥 NoSQL을 캐시로 사용

- NoSQL의 특징
  - **오버헤드가 적고 엑세스가 빠르다.**
  - **RDB가 처리하기 힘든 종류의 데이터 검색이 특기다.**
  - **여러 대의 서버로 분산처리가 가능하다.**
- KVS(Key - Value Store)는 엑세스 시 오버헤드가 적다.
  - 대량의 엑세스에도 안정적인 응답과 처리를 제공한다.
  - 여러 대의 호스트를 통해 스케일 아웃하기 편리하다

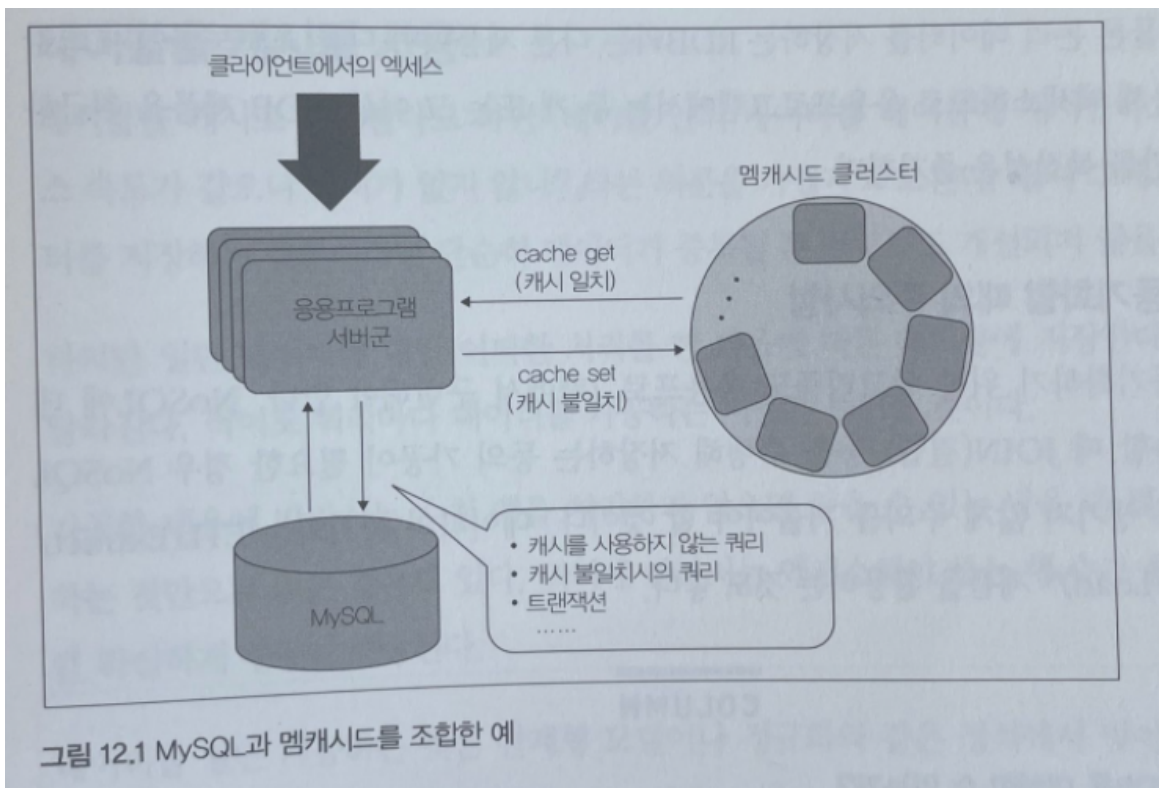


그림 12.1 MySQL과 멤캐시드를 조합한 예

- 특정 데이터에 대한 엑세스의 고속화
- DB에서 가져온 데이터 임의의 가공해 저장
- KVS를 이용할 수 있는 경우는 등가비교(=)로 데이터를 가져오는 경우만 해당된다.
- 응용 프로그램 → 캐시에 데이터 저장 / 트리거를 이용해 데이터 갱신 시 MySQL에서 멤캐시드로 데이터 동기화
- **복잡한 검색이 필요한 경우 그래프 DB, 전문 검색 엔진, 분석 DB, 도큐먼트 DB를 사용한다.**
- **NoSQL은 검색에 특화, 검색에 필요한 데이터를 NoSQL에 캐시해 전반적인 처리 성능 향상 가능**

논리 데이터는 RDB에서 관리

- 캐시는 캐시로 다루는 것이 중요하다.

- NoSQL은 데이터에 이상이 생겨도 막을 수 없지만 RDB는 데이터 이상현상을 억제할 수 있다.

데이터를 동기화 할 때 주의사항

- RDB 테이블의 JOIN연산을 통해 NoSQL에 저장하는 경우 데이터 부정합이 생기지 않도록 주의 필요

> NoSQL로 RDB를 대체할 수 있는가?
>
> - RDB를 NoSQL로 대체하고자 하는 이유는 **정형화된 스키마가 없기 때문에 설계가 간단하다고 생각하기 때문**이다.
> - 정형화된 스키마가 없으므로 **유연하게 데이터 구조를 바꾸는 것이 가능하지만, 쿼리는 스키마와 밀접한 관련이 있으므로 데이터의 중복이나 이상현상 문제를 벗어나기 어렵다.**
> - **정형화된 스키마가 없는 경우 데이터 구조의 변경을 막기 어렵다.**
> - **NoSQL은 정규화를 사용할 수 없다.**
> - RDB와 NoSQL의 장점이 다르다. 각 영역에 맞게 알맞은 용도로 사용하는 것이 바람직하다.

테이블을 캐시로 사용

- **1차적으로 데이터에 대해 가공하고 다른 테이블에 저장하는 경우 성능 개선 효과가 존재**할 수 있다.
  - 수백만 행의 데이터가 존재하는데 필요한 데이터만 가공해 테이블에 적재해 1000개로 만든 경우
  - 행의 전체 데이터에 대한 엑세스가 빨라진다.
  - **데이터 가공을 통해 새로운 데이터를 만들면 정규화가 깨질 수도 있다.**
  - **비정규화 된 경우 캐시와 논리 데이터의 명확한 구분이 필요하다.**
- NoSQL을 이용하지 않으므로 RDB 내에서 모두 관리가 가능하다.
- **한개의 DB 서버에서 관리할 데이터가 많아진다.**
- **갱신할 테이블이 많아져 갱신 오버헤드가 커진다.**
- **데이터의 갱신보다 참조가 많은 경우 테이블을 캐시로 사용하기에 적합하다.**

집계 테이블

- 어떤 테이블에서 구한 집계 결과를 별도의 테이블에 저장하는 것
- 집계 처리는 **여러 행에 엑세스하기 때문에 집계 데이터를 미리 가공해 처리하는 경우 엑세스를 줄일 수 있다.**
- 대표적으로 “랭킹”은 집계 테이블로 표시하는 것이 좋다.

예시)

✔️ 1:1 QNA 게시판이 있고 해당 테이블엔 (질문자, 답변자) 만 존재한다고 생각해보자.

- Top 3 답변자를 뽑기 위해선 기본적으로 테이블 풀스캔이 필요하다.

```
# 기존 테이블 풀스캔
SELECT answerer, count(*)
FROM QNA
GROUP BY answerer
ORDER BY COUNT(*) DESC
LIMIT 3;
```

- 이 경우 테이블 행이 1천만 개라면 테이블 풀스캔이 일어나고 랭킹 서비스에 접근할 때마다 테이블 풀스캔이 일어난다!!!
- 하지만, **집계 테이블로 캐싱한다면 행이 기하급수적으로 줄어들어 엑세스를 빠르게 할 수 있다.**

답변자	답변 횟수
맹준영	39349

답변자	답변 횟수
김민섭	12324
백예린	1004
홀란드	1003
…	…

```
# 기존 테이블 행이 1천만 개에서 기하급수적으로 줄었다.
SELECT answerer, answer_count
FROM answer_summary
ORDER BY answer_count DESC
LIMIT 3;
```

- answer_count에 인덱스를 걸어 더욱 빠른 효과를 얻을 수 있다.
- 집계 테이블을 갱신하는 방법
  - **원본 QNA 테이블이 갱신될 때 마다 트리거를 이용해 answer_summary 테이블도 갱신**한다.
  - 정기적으로 **원본 테이블을 스캔해 집계 테이블을 갱신하는 배치**를 만든다.
    - 배치를 이용하는 경우 실제 테이블과 캐시간의 레이턴시가 존재하므로 실제 테이블 결과와 검색 결과가 즉시 동기화가 필요하지 않은 경우 유용
- 이를 “실체화 뷰”라고 한다.

조인(JOIN) 데이터

- 매우 많은 양의 엑세스가 존재하는 프로그램은 단 한번의 JOIN연산도 성능 저하를 가져올 수 있다.
- 조인된 데이터를 테이블을 이용해 캐시해 사용한다.
- 조인된 데이터는 사이즈가 늘어나지만 이 경우 **조인 연산 속도 > 늘어난 데이터 테이블을 조회하는 속도**이다.
- 캐시의 갱신은 기존 데이터 값의 갱신과 삭제를 외부키를 이용하고, 새로운 데이터 삽입시에는 트리거로 결합해 동기화한다.

조인 데이터의 장점

- **디스크 I/O를 줄일 수 있다.**
  - 조인된 테이블 데이터는 한 군데 저장되므로 캐시 미스시 테이블 엑세스 I/O 횟수를 줄일 확률이 높다.
- **정렬과 궁합이 좋다.**
  - 두 테이블의 데이터를 조인한 다음 정렬하는 경우 실행 계획에 따라 인덱스로 해결하지 못한다.
  - **조인된 테이블에 대해 인덱스를 생성하면 인덱스를 이용한 정렬이 가능하다.**
- **NewSQL과 궁합이 좋다.**
  - NewSQL = RDB + KVS 인터페이스 = 하이브리드 DB
  - 사전에 데이터를 미리 결합하면 KVS 인터페이스를 이용해 엑세스할 수 있다.

태그

- 태그란 개체를 나타내는 데이터에 연상되는 속성을 나타내고자 붙이는 라벨
  - ex) 축구공, 축구화, 유니폼 → 축구용품 (태그)
- 태그는 검색조건을 지정하기 편리하다.
- 태그는 한 개의 아이템에 대해 여러 개의 태그가 붙을 수 있고, 하나의 태그에는 여러 개의 아이템이 속할 수 있다.

태그의 문제점

- 하나의 태그에 여러 개의 아이템이 존재할 수 있기 때문에 태그를 통한 검색은 많은 데이터 행을 반환한다.

```
# 욕실용품 100만 가지
# 주방용품 100만 가지
# 가전제품 100만 가지

SELECT item_name
```

```
FROM tags
WHERE tag = '욕실용품'; # 100만건 반환
```

- 두 가지 태그에 속한 아이템을 찾는 경우라면, 이 많은 데이터의 JOIN 연산이 발생한다!
- 더 많은 태그가 붙은 아이템을 찾고자 한다면 그만큼 데이터가 많아지며 연산이 발생한다!

```
SELECT item_name
FROM tags t1
INNER JOIN tags t2
USING (item_name)
WHERE t1.tag = '주방용품' AND t2.tag = '가전제품';
```

- 이러한 문제를 해결하기 위해 **태그를 사용한 검색 연산 비용을 줄일 필요가 존재**한다.

태그 사용

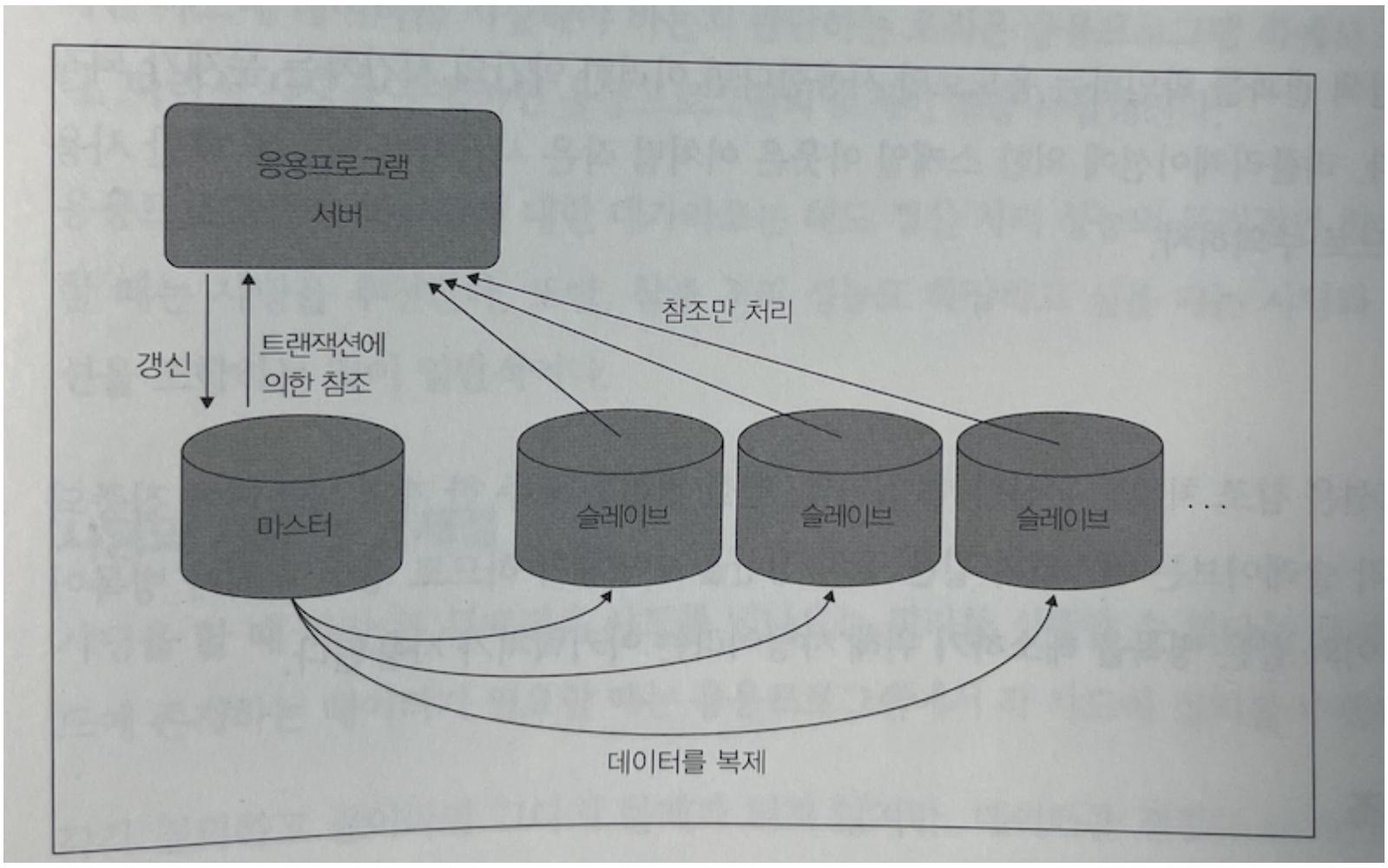
- 태그는 주로 해당 태그의 검색 결과 순위를 나타내는데 많이 사용된다.
- **하나의 태그를 이용하는 경우 해당 태그의 순위값과 같은 컬럼을 이용**
- **두 개 이상의 태그를 이용하는 경우 태그 간의 모든 조합(nCm)을 만들고 각 태그 조합에 대한 순위값을 추가해 캐시에 데이터를 저장한다.**
  - 태그의 개수가 많아질수록 태그의 **조합이 늘어나며 캐시 사이즈가 증가한다.**
  - 따라서, 위의 방식을 사용할 때는 **태그의 최소, 최대 개수를 제한하는 것이 중요하다.**

> 전치 인덱스를 이용해 검색을 빠르게
>
> 아이템 개수가 많지 않은 경우 **전치 인덱스를 이용해 태그 검색의 성능을 향상**시킬 수 있다.
> 태그에 대해 전치 인덱스를 붙히고, 전치 인덱스를 통해 해당 태그에 접근한다.
>
> *** 전치 인덱스 : 행에 포함된 단어나 부분 문자열에 그 행의 포인터를 저장한 인덱스**

▼ 12.3 스케일 아웃

- 단일 DB 서버만으로 성능 한계에 도달하는 경우 비용을 들이지 않고 처리량을 늘리는 방법
- **여러 개의 서버를 이용해 부하를 분산하고 서버 수에 따라 처리량을 늘리는 것 → 물리적인 처리 능력을 높인다.**

리플리케이션 (복제)



리플리케이션 구조

- **RDB 서버에 포함된 데이터를 다른 DB서버로 복제하는 기능**
- **복제 원본 DB를 마스터 DB, 복제된 DB를 슬레이브 DB**
- 마스터 DB와 슬레이브 DB는 1:N 관계가 되며, **참조의 부하를 여러 대의 DB로 분산하는 것이 가능**
- 트랜잭션이 필요한 참조는 마스터 DB에서 사용, 단순 참조는 슬레이브 DB에서 사용
  - 쓰기 연산 (insert, update, delete) → 마스터 DB
  - 읽기 연산 (select) → 슬레이브 DB

슬레이브 질의 방식

- 애플리케이션 서버와 슬레이브 DB가 같다면 항상 같은 서버에상에서 동작하는 슬레이브 DB가 된다.
- 애플리케이션 서버와 슬레이브 DB가 다른 서버인 경우
  - **라운드 로빈이나 랜덤으로 연결 대상 슬레이브 DB를 바꾼다.**

데이터의 논리 정합성과 비동기 리플리케이션

- 애플리케이션 서버가 **슬레이브 DB에 쿼리호출 시 마스터 DB에서 조회하는 것과 동일한 효과**를 내야한다.
  - **슬레이브 DB의 데이터를 마스터 DB와 같은 논리적 정합성을 유지해야한다.**
- 동기 리플리케이션을 사용하는 경우 오버헤드가 많이 발생하기 때문에 **비동기 리플리케이션**을 이용한다.
  - 슬레이브 DB와 마스터 DB간 데이터 정합성에 약간의 레이턴시가 발생할 수 있다.
- **스케일 아웃은 슬레이브 DB와 마스터 DB간 약간의 시간차를 허용하는 범위내에서 사용하는 것이 바람직**하다.

샤딩

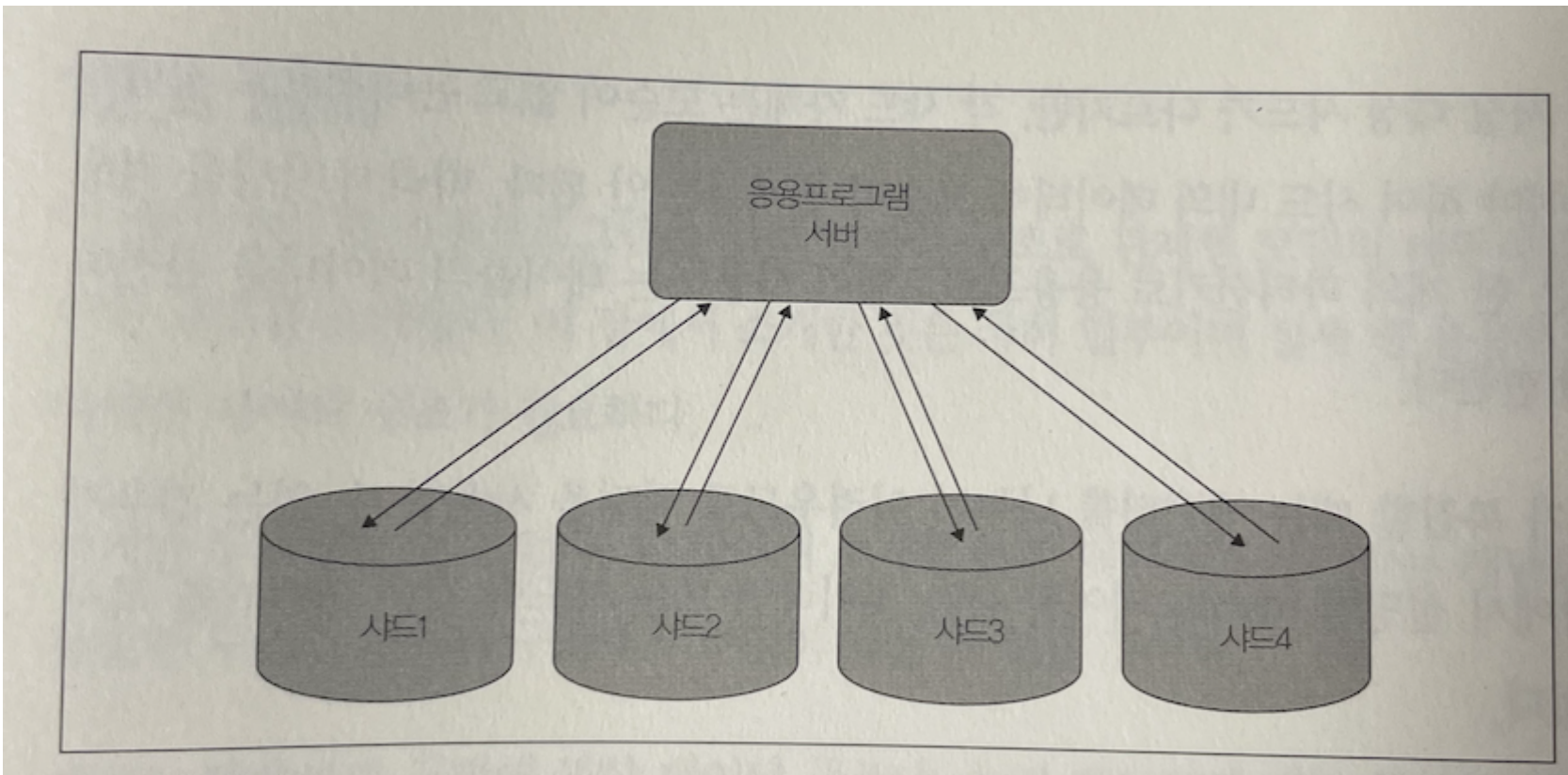


그림 12.7 샤딩의 아키텍처

- 리플리케이션을 이용하는 경우 하나의 마스터 DB에서 트랜잭션 처리가 집중되고 슬레이브 DB는 동기화를 위해 같은 양의 갱신 처리를 수행 → **병목현상이 발생**

샤딩의 구조

- 행 별로 데이터의 저장 위치를 변경하는 구조
- **파티셔닝의 수평 분할과 비슷하지만, 데이터의 저장 대상이 각 DB서버가 된다는 점이 다르다.**
- 각 샤드의 DB 서버 스키마 구조는 동일하며 저장되는 데이터 내용만 다르다.
- **애플리케이션 단에서 어떤 샤드에 데이터를 적재할지 로직을 구현**한다.
- 애플리케이션의 복잡성이 증가하지만, 성능 향상을 가져온다.

샤딩의 가장 큰 문제점

- 샤드를 넘나드는 쿼리를 실행할 수 없다.
  - 다른 샤드의 데이터가 필요한 경우 애플리케이션 단에서 다른 샤드로 추가적인 쿼리를 수행해야한다.
  - 샤드끼리 데이터 조인이 불가능하다.
- 샤딩을 사용하는 경우 각각 같은 구조의 스키마를 갖지만, 동기화 되지 않으므로 **스키마 구조 변경 시 모든 샤드에 적용해야하므로 오버헤드가 발생한다.**
- 스키마 구조가 복잡한 경우 데이터를 나누기 힘들기 때문에 샤딩을 사용하기 어렵다.
- 샤드 자체는 모순이 존재하면 안되기 때문에 **한 개의 파티션 키로 테이블 데이터를 완전히 나눈 경우에 사용**한다.

▼ 12.4 요약

- 실제 논리 데이터와 캐시를 이용해 복잡한 구조의 데이터 구조도 해결할 수 있다.
- 캐시는 **데이터가 많거나 데이터 구조가 복잡하여 빠르게 엑세스 할 수 있는 데이터가 필요한 경우 실행을 빠르게 하기 위해 사용한다.**
- 인덱스와 캐시는 데이터 엑세스를 빠르게 하기 위해 사용되는 별도의 구현 데이터라는 공통점이 존재
  - RDB의 인덱스로 성능 개선이 어려운 경우 캐시를 이용
  - 쿼리를 이용해 논리적 데이터 조회의 한계가 발생하는 경우에 물리적인 관점에서 해결한다.