Spring Framework 캐싱 최적화 전략

허진[01], 하란[1]
홍익대학교 컴퓨터공학과
hurjin1109@naver.com, rhanha@hongik.ac.kr

Cache Optimization Strategies for Spring Framework

Jin Heo[01], Rhan Ha[1]
[1]School of Computer Engineering, Hongik University

요 약

본 논문은 Hibernate Framework와 Spring Framework을 이용한 캐싱 최적화 전략을 다룬다. Spring Framework에서 사용할 수 있는 여러 캐싱 전략, 즉 Data JPA, JPQL, Spring 내장 Cache Manager를 활용한 방법을 설명한다. 실험을 통해 성능을 평가하고, AOP와 JMeter를 사용하여 결과를 비교 분석한다. 같은 SQL 쿼리를 생성하는 Data JPA는 JPQL에 비해 Insert 쿼리에서 약 2.25배, Cache Manager을 사용한 경우 기존 Data JPA에 비해 Select 쿼리에서 약 5.75배의 Throughput 성능향상 및 송수신 데이터량 증가를 확인하였다.

1. 서론

Java 프로그래밍 환경에서 애플리케이션을 만드는 데 사용할 수 있는 몇몇 Framework가 있다. Web Application에 더 집중하면 Spring Framework가 있으며, 특히 Hibernate Framework, Mybatis 등 다양한 다른 Framework와 결합하여 사용할 수 있다[1]. 특히 Hibernate와 결합한 Spring Framework는 JPA를 통해 데이터베이스와의 상호작용을 간편하게 처리하여 개발 효율성을 높여준다. Hibernate는 객체 관계 매핑(Object Relational Mapping) 도구로, JPA의 구현체이자 데이터베이스 처리에 다양한 기능을 제공한다[2].

앞서 소개한 객체 관계 매핑(ORM) 기술은 객체 지향 애플리케이션 개발 모델과 관계형 데이터베이스 모델 간 불균형 문제를 해결해 준다. 하지만 객체 지향 애플리케이션에서 생성된 SQL 쿼리는 성능 저하와 비용 기반 최적화의 불일치를 초래할 수 있으며, 이에 대한 개선방안이 필요하다[3]. 본 논문에서는 이러한 성능 문제를 해결하기 위한 핵심 전략 중 하나로 캐싱을 제안한다. 캐싱을 통해 데이터베이스 접근 횟수를 줄이고 애플리케이션의 성능을 향상시킬 수 있다[4]. 또한, Spring Framework가 지원하는 다양한 캐싱 기법을 분석하고 이를 통해 ORM 성능 향상 방안을 논의한다.

2. 관련 연구

Spring Framework와 캐싱에 대한 연구는 오랜 기간동안 이루어져 왔다. 특히 다양한 분야에서 캐싱을 통해 성능 향상시킨 사례가 있다.

아래 선례 연구 결과(그림 2)는 주기억장치 상주형 DBMS를 활용하여 기존 DBMS에 비해 높은 트랜잭션 처리 속도를 통해 성능을 향상시켰다. 캐싱 전략(클라이언트 엑세스 가능성이 높은 문서를 분석, Cache 서버에 저장)을 통해 서버의 과부하, 네트워크 트래픽 증가, 대기시간의 증가 등의 문제를 해결하여 성능 향상을 이끌었다.[5]

그림 1 Cacging DB 와 Oracle DB 성능비교[5]

DB구분	Kairos Caching DB	Oracle EIP DB
1	0.031005	1.125000
2	0.016006	1.171875
3	0.014999	1.175781
4	0.031005	1.187500
5	0.014999	1.136719
6	0.094024	1.156250
7	0.046997	1.203125
8	0.046997	1.175781
9	0.030975	1.136719
10	0.031005	1.171875
평균	**0.035801**	**1.164063**

다른 선례 연구로 기존의 비효율적으로 저장 공간을 사용하고 중복 저장 문제를 LFU 및 LRU 알고리즘을 적용하여 성능향상을 이끌었다. 캐싱 기법을 통해 Cache 저장 공간 내에 더 많은 변환된 마크업 페이지를 저장하고, 이를 통해 Cache 성능을 향상시켰다[6].

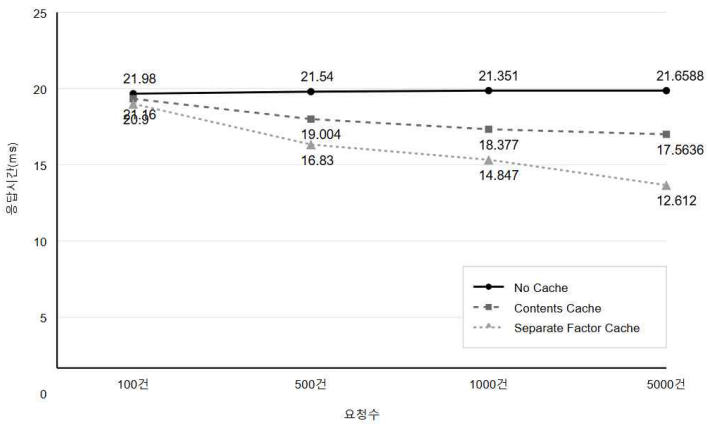

그림 2 LFU Caching 기법과 기존 모델의 성능 비교[6]

이외에도 다양한 캐싱 전략에 대한 연구가 있다. 본 논문에서는 Spring Framework의 캐싱 기법을 통해 성능을 향상시키고자 한다.

3.1 연구 설계

본 연구에서는 Persistence Context(1차 캐시)를 사용한 Data JPA, JPQL, Spring 내장 Cache manager을 사용한 method level Caching등을 미리 구축한 Spring Web Application을 활용해 실험하고자 한다. 로직을 제외한 나머지 조건은 모두 동일하게 진행된다.

3.2 연구 환경

본 연구에서 RDBMS는 Amazon RDS for MySQL을 이용하여 진행된다. DB의 경우 클라우드 환경에서 진행되며, AWS에서 제공하는 T4g 계열(Arm 기반 Gravtion 2 프로세서) 인스턴스를 사용하였다.

Spring Web Application의 경우 Java 17 기반으로 작성되었으며, Spring Boot 3.2.5 버전을 사용하였다. Spring Application 구동의 경우 Intel i7-1195G7 Cpu로 구동되며, 16GB의 RAM을 사용하였다.

3.3 결과 측정 및 분석 도구

본 연구에서 결과 분석 도구로 Apache Jmeter를 사용하였다. Jmeter는 Apache에서 만든 자바로 만들어진 웹 어플리케이션 성능 테스트 오픈소스이다. Application의 지연 시간, Throughput 등을 측정하는 기능을 가지고 있다. Apache Jmeter을 사용하여 각 로직별로 Thread Group을 생성한 후, Spring Application에 요청을 보내 성능 측정 및 부하 테스트를 진행하고자 한다.

또한 본 코드의 가독성 유지하고 로깅 로직을 최대한 분리하여 Test 별 로직 메소드 동작 방식을 분석하기 위해 AOP 기법을 사용하였다. AOP는 Aspect-Oriented Programming의 약어로, 이 기법을 활용해 method 실행 전후에 로그를 남기는 방식을 구현하였다[7].

3.4 연구 로직

- 로직(로그) 분석

그림 3 로그 분석을 통해 간략화한 로직

그림 3은 로직의 전체적 구조와 데이터 흐름을 파악하기 쉽게 로그를 분석하여 작성하였다. 화살표로 데이터의 흐름을 나타내었다. 동일한 연구 환경을 위해 Data JPA와 JPQL을 사용한 로직 모두 Hibernate를 통해 동일한 SQL이 생성되도록 코드를 구성했다. 인증은 Token Provider가 담당하며, 실제 환경과 유사하게 연구할 수 있도록 설정하였다.

- 로직 공통 SQL 쿼리

```
select m1_0.member_id,m1_0.activated,a1_0.member_id, ...
from member m1_0
left join member_authority a1_0
        on m1_0.member_id=a1_0.member_id
left join authority a1_1
        on a1_1.authority_name=a1_0.authority_name
where m1_0.email=?
```

- SELECT 로직 쿼리

```
select c1_0.member_id,c1_0.category_id,c1_0.category_name
from category c1_0 where c1_0.member_id=?
select b1_0.category_id, b1_0.book_id, b1_0.isbn13
from books b1_0 where b1_0.category_id=?
```

Select 로직에 관한 쿼리의 경우 두 개의 LEFT JOIN이 사용된 쿼리 하나, 정보를 조회하는 형태의 쿼리 두 개가 사용되었다.

- INSERT 로직 쿼리

```
select c1_0.category_id,c1_0.category_name,c1_0.me...
from category c1_0
where c1_0.category_name=? and c1_0.member_id=?
insert into books (category_id, isbn13) values (?, ?)
```

Insert 로직의 경우 두 개의 LEFT JOIN이 사용된 쿼리 하나, 정보를 조회하는 형태의 쿼리 하나, 정보를 삽입하는 쿼리 하나가 사용되었다.

주목할 점은 Spring 내장 Cache Manager를 활용한 로직을 제외한 나머지 세 로직은 쿼리를 포함한 구조를 나타내었다. 그러나 Cache Manager을 활용한 로직의 경우 반복된 실행에서는 Service Layer에 도달되지 않고 종료되었다.

4. 성능 분석

4.1 연구 결과

- Spring Data JPA Logic, JPQL Logic(Insert)

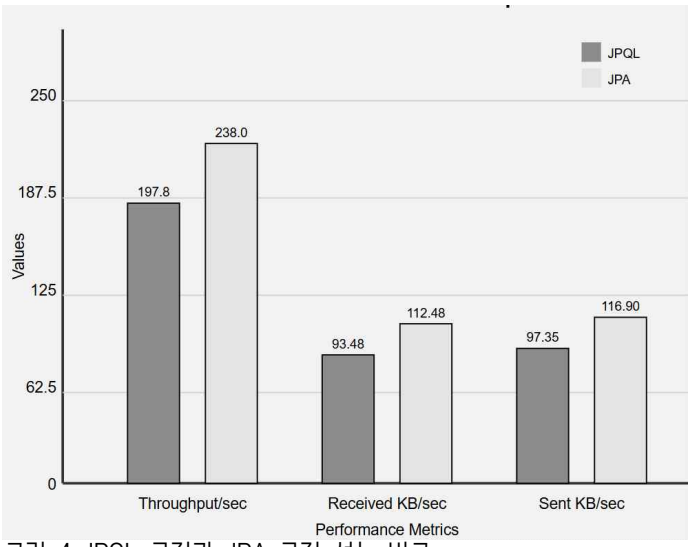

그림 4 JPQL 로직과 JPA 로직 성능 비교

JPQL을 사용한 SQL Insert문의 경우 Throughput/sec : 197.8/sec, Received KB/sec : 93.48, Sent KB/sec : 97.35로 측정되었다.

같은 기능을 수행하는 JPA로직의 경우 Throughput/sec : 238.0/sec, Received KB/sec : 112.48, Sent KB/sec : 116.90로 측정되었다.

- Spring Data JPA Logic, Use Cache Manager Logic(Select)

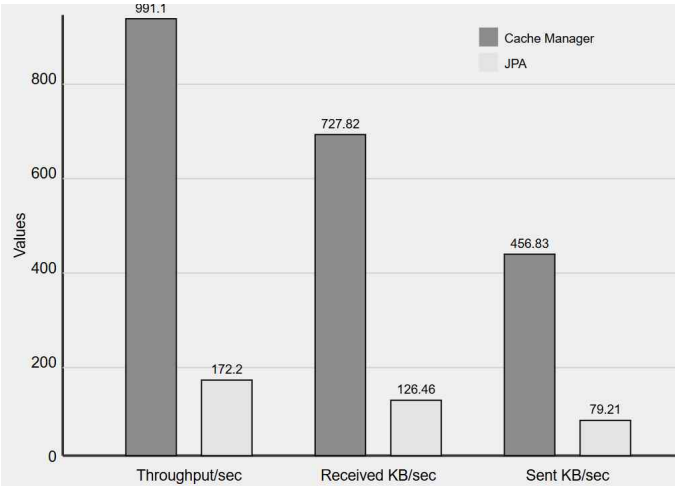

그림 5 Cache Manager 사용 로직과 JPA 로직 성능 비교

Spring Framework 내장 Cache Manager을 활용한 Select 로직의 경우Throughput/sec : 991.1/sec, Received KB/sec : 727.82, Sent KB/sec : 456.83 로 측정되었다.
같은 기능을 수행하는 JPA 로직의 경우 Throughput/sec : 172.2, Received KB/sec : 126.46, Sent KB/sec : 79.21로 측정되었다.
앞선 4개의 측정 모두 200개의 Thread에서 5번의 Loop count로 측정하였다.

4.2 결과 분석

본 연구에서는 반복적인 Insert SQL 쿼리 실행에 있어 JPQL 로직과 Data JPA 로직의 성능을 비교 분석하였다. 연구 결과, Data JPA 로직이 JPQL로직에 비해 전반적으로 20% 빠른 속도를 보였다.

앞선 단계에서 AOP를 활용한 메소드 수준의 로깅을 수행했었다. 로그 분석 결과, 두 로직 접근 방식에서 실행되는 메소드의 순서와 종류는 동일했다. 따라서, 앞선 두 성능 차이는 JPA와 JPQL 각각의 내부 매커니즘의 차이에서 기인한 것으로 추정된다. 이는 Data JPA가 제공하는 효율적인 캐싱 전략 및 쿼리 생성, 실행 메커니즘과 내부적인 배치처리 기능 등이 복합적으로 작용한 결과로 해석될 수 있다[7].

Data JPA 로직과 내장 Cache Manager를 사용한 로직의 반복적인 Select 실행에 대한 성능 비교의 경우 Cache Manager를 사용한 로직이 약 5.75배의 성능향상을 보였다. 앞선 로그 분석 결과를 통해 Cache Manger을 사용한 경우 데이터베이스(MySQL)를 대상으로 한 실제 쿼리 실행이 발생하지 않는 것을 확인할 수 있었다. 반면 Data JPA 로직은 매 요청마다 SQL 쿼리 수행을 로그를 통해 확인할 수 있었다.

Cache Manager를 활용한 구현에서는 초기 조회된 데이터를 메모리에 캐싱하여 후속 요청에서 재사용하는 방식이 효과적으로 작동함을 확인하였다. 이러한 캐싱 메커니즘은 디스크 I/O 연산, 데이터베이스 쿼리 실행에 따른 컴퓨팅 리소스 소비, 그리고 네트워크 레이턴시와 같은 시스템 부하 요소들을 현저히 감소시켰으며, 이는 전반적인 시스템 성능 향상으로 이어졌다.

5. 결론 및 향후 과제

본 연구를 통해 반복적 쿼리 작업이 빈번한 애플리케이션에서 적절한 캐싱 전략의 도입이 현저한 성능 향상을 가져올 수 있음을 실증적으로 검증하였다. 특히 데이터베이스 접근 빈도와 I/O 작업의 감소를 통해 상당한 성능 개선 효과를 확인할 수 있었으며, 이는 갱신 주기가 긴 데이터를 다루는 시스템에서 캐싱 전략의 효과적인 활용 가능성을 시사한다.

그러나 본 연구는 다음과 같은 한계점을 가지고 있다. 첫째, 캐싱 전략 도입 시 데이터의 일관성(consistency) 보장 문제가 발생할 수 있다. 둘째, 본 연구에서 수행된 실험이 단순 Select 및 Insert 쿼리에 국한되어 있어, 실제 운영 환경의 복잡성을 완전히 반영하지 못한다는 제약이 있다.

향후 연구에서는 데이터베이스 쿼리의 다양성, 스키마의 복잡도, 동시성 처리 등 실제 서비스 환경에서 발생할 수 있는 다양한 변수들을 고려한 포괄적인 분석이 필요하다. 또한 최적의 시스템 성능 달성을 위해서는 본 연구에서 제시된 캐싱 전략들의 효과적인 조합과 함께, 각 애플리케이션의 특성에 따른 세밀한 성능 요소 분석이 수반되어야 할 것이다.

참 고 문 헌

[1]Ginanjar, A., & Hendayun, M. (2019). Spring Framework Reliability Investigation Against Database Bridging Layer Using Java Platform. Procedia Computer Science. Elsevier BV. https://doi.org/10.1016/j.procs.2019.11.214

[2]Kisman, I., & Isa, S. M. (2016). Hibernate ORM query simplification using Hibernate Criteria Extension (HCE). In Information and Computer Science (NICS), pp. 23-28. 3rd National Foundation for Science and Technology Development Conference, September 2016.

[3]Niehaus, D., Nahum, E., Stankovic, J. A., "Predictable Real-Time Caching in the Spring System", IFAC Proceedings Volumes, vol.24, No2, pp. 57-62, 1991

[4]Colley, D., & Stanier, C. (2017). Identifying New Directions in Database Performance Tuning. Procedia Computer Science. Elsevier BV. https://doi.org/10.1016/j.procs.2017.11.036

[5] 강용식, "기업정보포탈(EIP)의 성능향상을 위한 caching 기법에 관한 연구", 충남대학교 대학원, 석사학위논문, 53p, 2006

[6]심근정, Kang, Euisun, 김종근, 고희애, & Lim, Younghwan. (2007). Separate Factor Caching Scheme for Mobile Web Service. The KIPS Transactions:PartD, 14D(4), 447-458. https://doi.org/10.3745/KIPSTD.2007.14-D.3.447

[7] Spring Framework, "Spring Framework Overview", Spring an official document, 2024, (https://docs.spring.io/spring-framework/reference/overview.html)