

Local Cache와 Invalidation Message Propagation 전략을 활용하여 API 성능 튜닝하기

김민규

(pkgonan1@naver.com)

소개



김민규

야놀자 플랫폼실
쿠폰 API 서버 개발 & 운영

트래픽이 쏟아질 때 희열을 느끼며
이벤트 하는 날이 가장 설렙니다

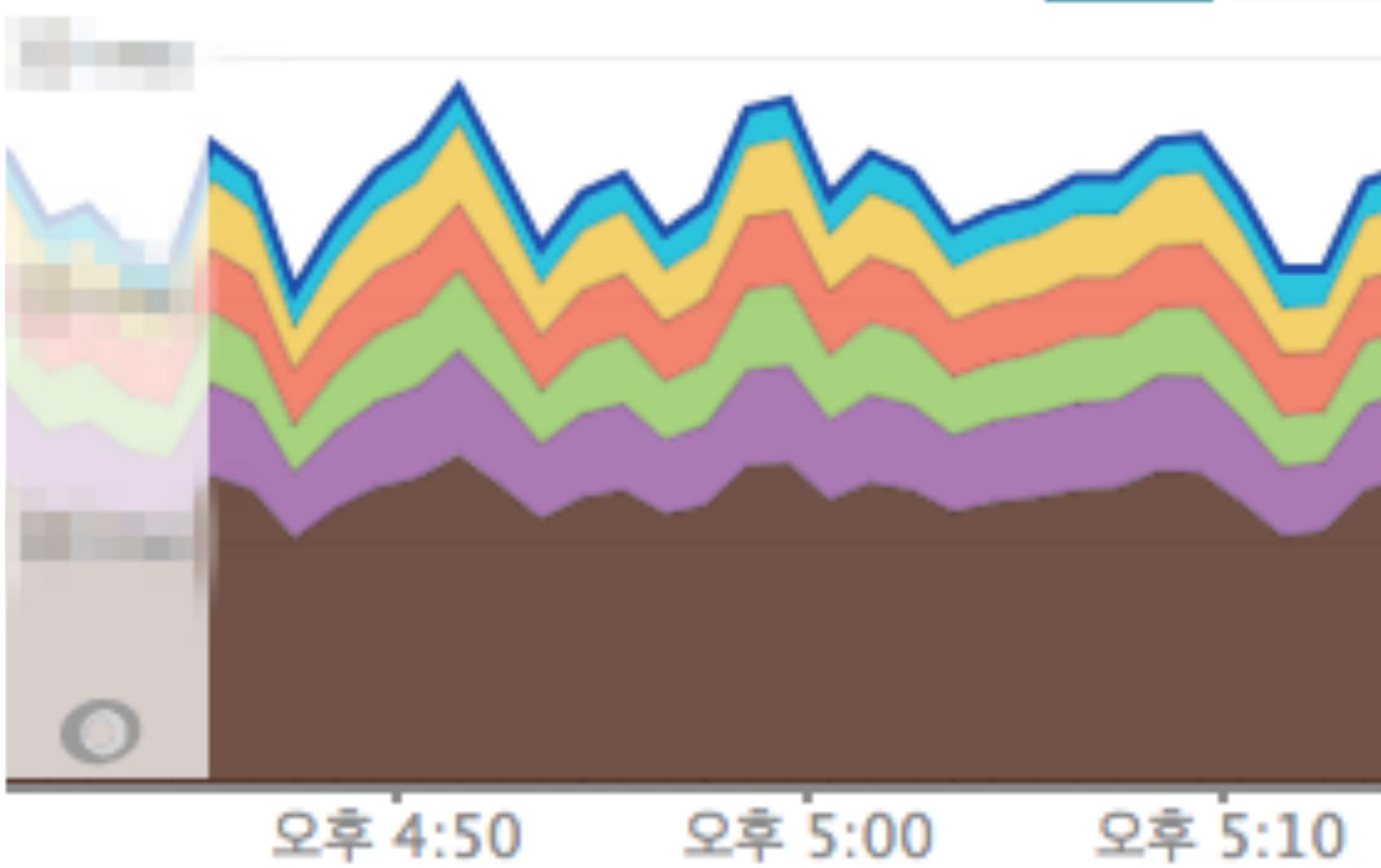
이벤트 때문에 인스턴스 잔뜩 늘렸는데..
트래픽 없을때가 가장 아쉽습니다

후덥의 기술블로그 : <https://pkgonan.github.io>
Facebook : <https://www.facebook.com/pkgonan>
Email : pkgonan1@naver.com

**다음 이미지를 보면
어떤게 떠오르나요 ?**

App performance

App server breakdown



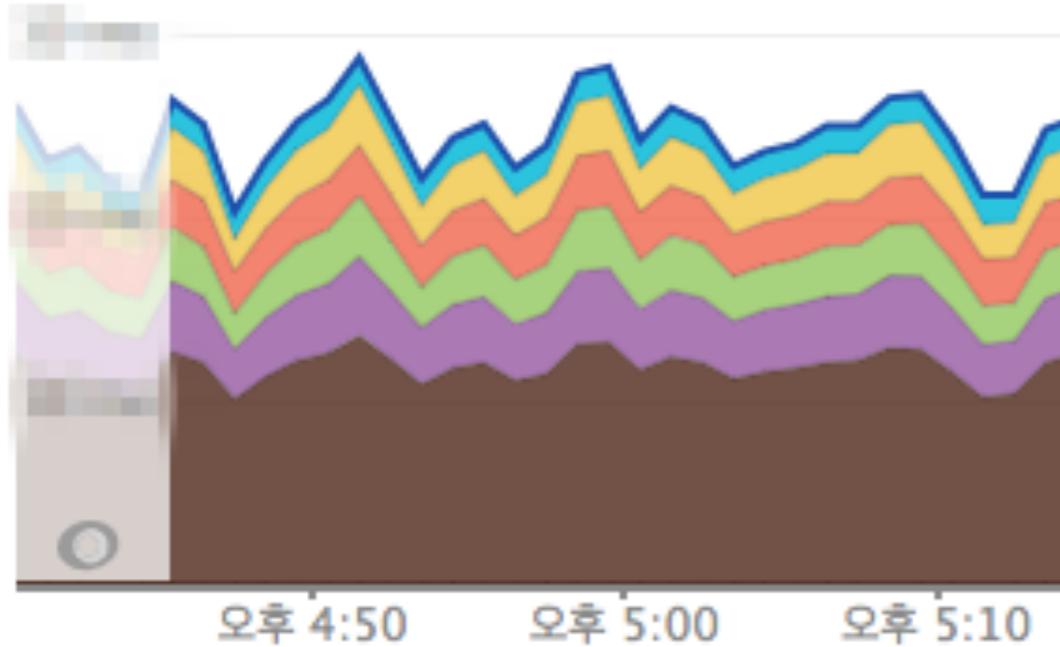
저는 사실 ...



**다시 한번 자세히 살펴 봅시다
이번에는 무엇이 보이나요?**

App performance

App server breakdown



/CouponProgressController/getPlaceCouponProgressesWithMetaCouponAttributes
CouponProgressController.getPlaceCouponProgressesWithMetaCouponAttributes()
MySQL select MySQL select MySQL select MySQL other Other
Response time

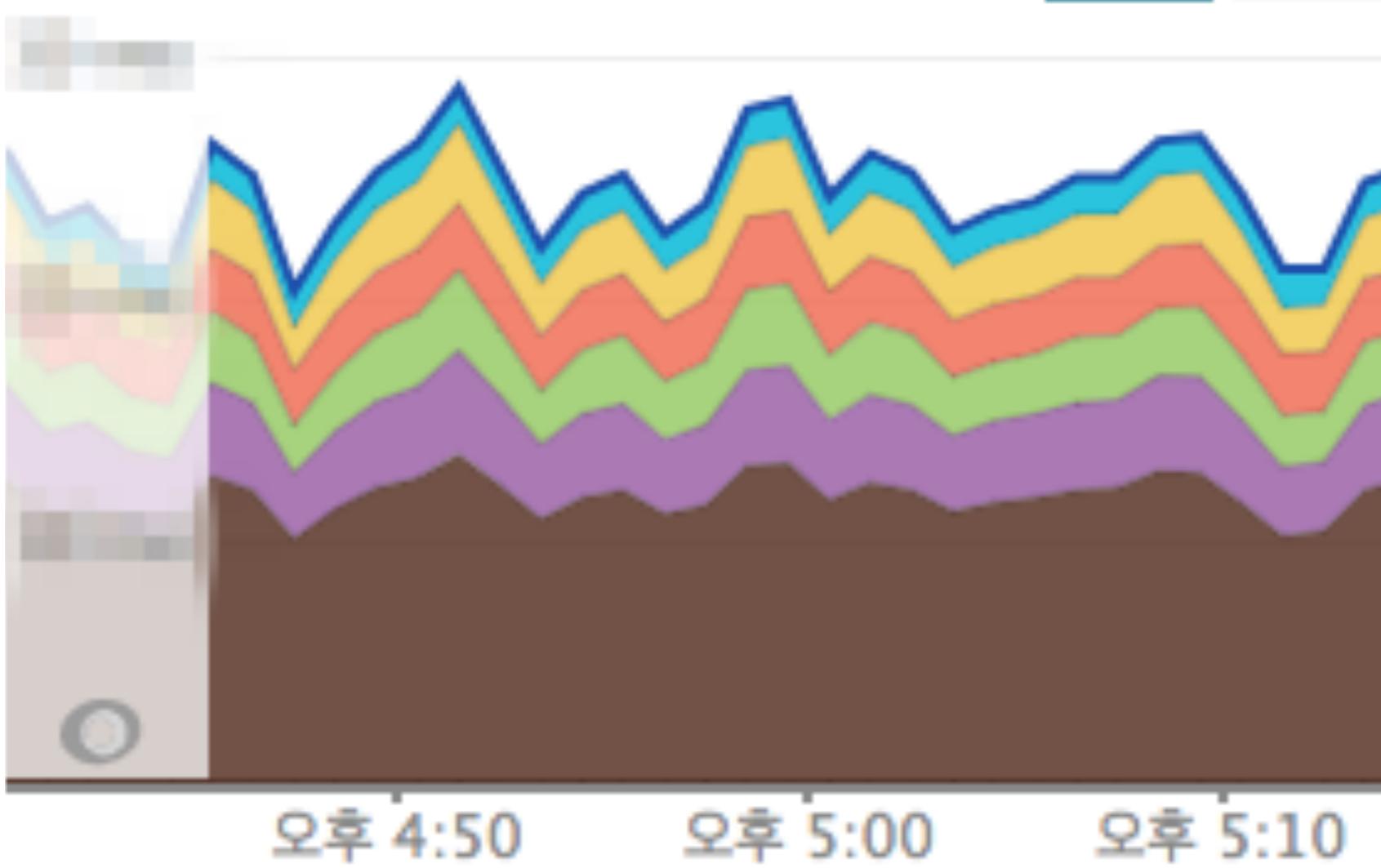
MySQL Select...

쿼리 시간 \geq 전체 수행시간의 절반

쿼리 조회를 줄일 수는 없을까 ?

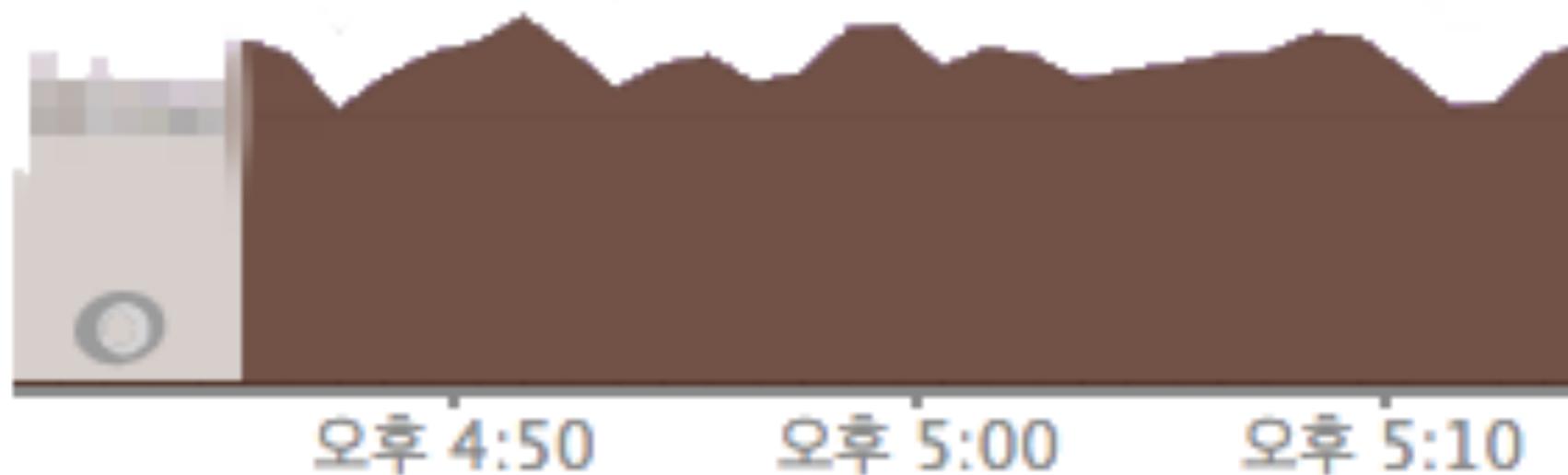
App performance

App server breakdown



App performance

App server breakdown



그리고 생각을 현실로

App performance

App server breakdown

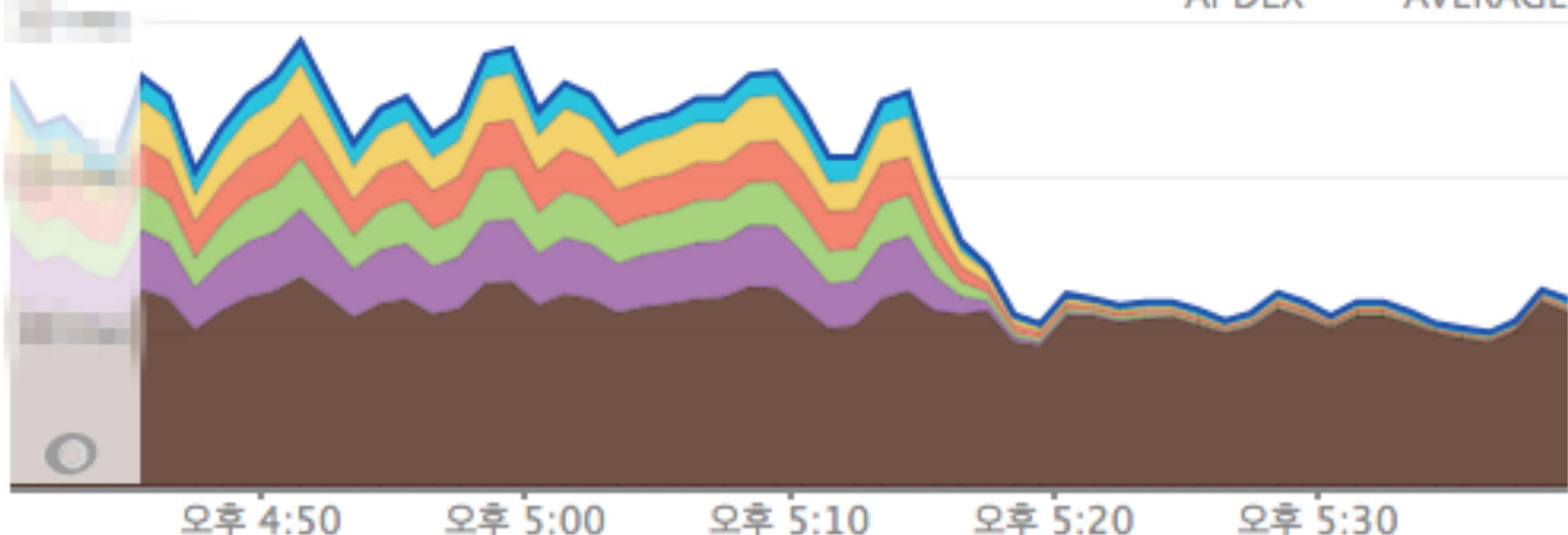


%

1.0
APDEX



AVERAGE



다루는 내용

- Cache
- Hibernate First & Second Level Cache
- IMDG - In memory data grid

목차

1. Local Cache & Invalidation Message Propagation 전략 도입 배경
2. Hibernate Cache 동시성 전략과 구현체의 선택
3. Hazelcast 그리고 Second Level Cache
4. Cache Hit율 극대화 전략의 적용
5. Local Cache & Invalidation Message Propagation 전략 적용 결과
6. 결론

Chapter 1

Local Cache & Invalidation Message

Propagation 전략 도입 배경

Entity 패턴 분석

Most time consuming

/coupon/[REDACTED] (GET)

43.7%

/progress/[REDACTED] (POST)

30.3%

/progress/[REDACTED] (POST)

20.2%

/user...Id}/[REDACTED] (GET)

1.83%

/coup...uponDetail/[REDACTED] (GET)

1.78%

/coupon/[REDACTED] (GET)

1.35%

- 쿠폰 메타 성격의 Entity 조회비율 99.9%
- 전체 트래픽의 95% 차지하는 상위 3개 API에서 쿠폰 메타 Entity 다수 호출
- 자주 변경되지 않으며 호출량은 아주 많은
- Cache에 매우 적합한 성격의 Entity

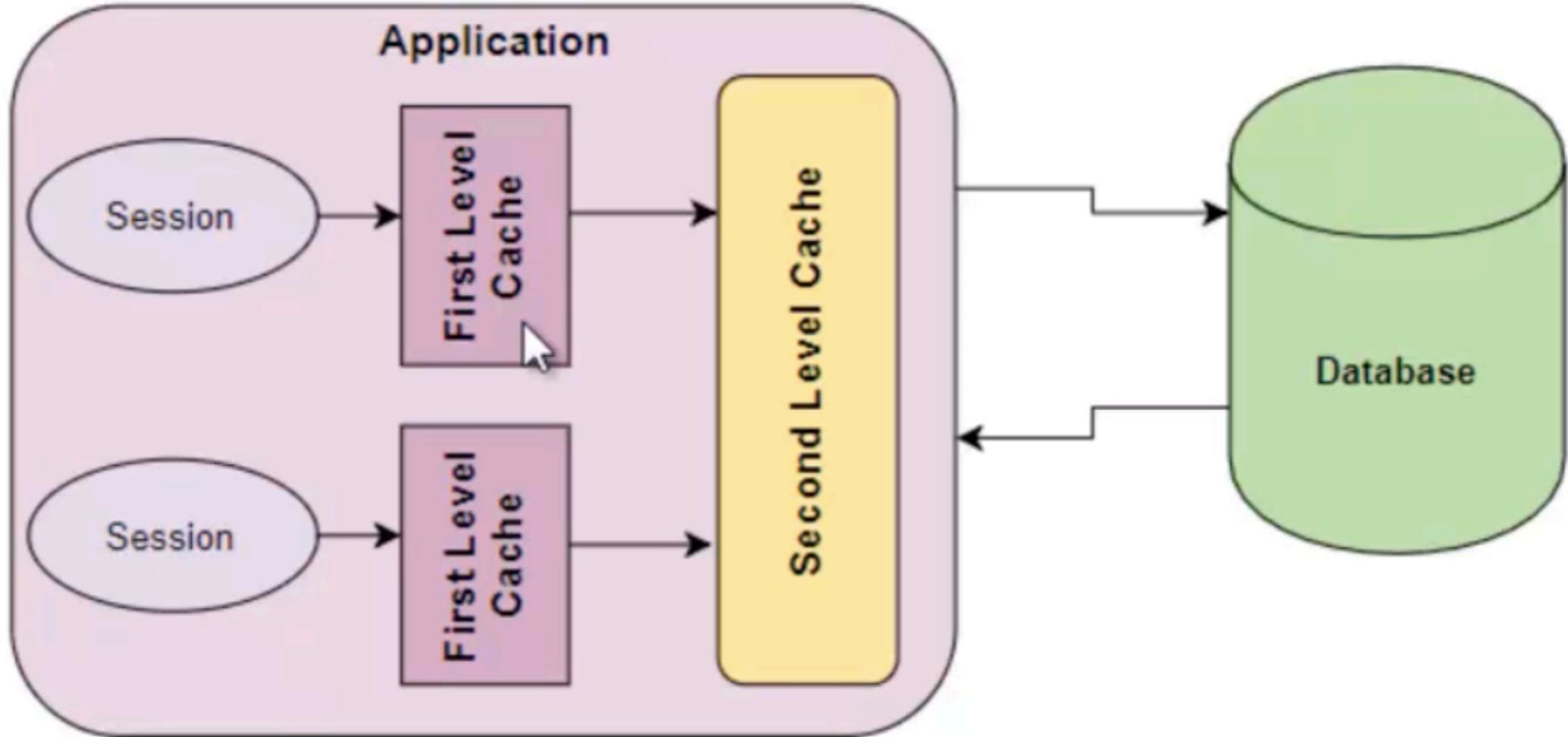
**Cache가 효과를 볼 것이라는 것은 알겠는데
어떻게 사용할 건가요 ?**

**Hibernate를 ORM으로 사용 중이며
API Call마다 특정 Entity를 중복 조회하는 상황**

세션 내부에서만 Cache를 공유하는
First Level Cache Hit율 낮음

여러 세션에서 Cache를 공유하는
Second Level Cache를 적용하면

First Level Cache Miss -> Hit 가능



현재 조회 패턴에서 Cache Hit율 증가를 위해
Hibernate Second Level Cache 사용 결정

**Second Level Cache를 쓴다는 것은 알겠는데
Local Cache vs Remote Cache
어떤 것이 더 적합한 상황인가요 ?**

성능 ?

Local Cache > Remote Cache

일관성 ?

~~Local Cache~~

<

Remote Cache

시스템의 Entity 사용 패턴은 ?

**쿠폰 메타 Entity에 대해
1초에 수 백 ~ 수 천번의 극단적인 중복 조회**

성능 측면에서 매번 Network를 타는 Remote Cache 보다 Local Cache가 적합한데...

Local Cache는 일관성을 보장할 수 없잖아 ?!

**일관성과 성능
두마리 토끼를 함께 잡을 수는 없을까 ?**

Local Cache with IMDG

IMDG ?

In Memory Data Grid

데이터를 분산 적재 및 처리하며, 확장성을 보장

Local Cache vs Remote Cache 무엇이 더 적합한 상황 ?

- IMDG로 설계된 Cache 구현체는 Entity의 상태가 변경 되었을때
변경사항을 다른 인스턴스에게 전파하는 기능 제공

Local Cache vs Remote Cache 무엇이 더 적합한 상황 ?

- IMDG로 설계된 Cache 구현체는 Entity의 상태가 변경 되었을때 변경사항을 다른 인스턴스에게 전파하는 기능 제공
- IMDG Cache 구현체를 Local Cache로 활용하면 일관성과 성능 모두 얻을 수 있다

Local Cache vs Remote Cache 무엇이 더 적합한 상황 ?

- IMDG로 설계된 Cache 구현체는 Entity의 상태가 변경 되었을때 변경사항을 다른 인스턴스에게 전파하는 기능 제공
- IMDG Cache 구현체를 Local Cache로 활용하면 일관성과 성능 모두 얻을 수 있다
- 성능과 일관성 모두 얻기 위해 **Local Cache with IMDG** 결정

Chapter 2

Hibernate Cache 동시성 전략과 구현체의 선택

**Hibernate Cache 사용시 동시성에 대한 고민 필요
현재 상황에서 어떤 동시성 전략이 적합한가 ?**

Cache Concurrency Strategy	적합성	판단 근거
Read-only	✗	Entity가 수정이 가능한 상황으로 부적합
Nonstrict read/write	✓	Entity에 동시 수정이 빈번하지 않으며, 조회가 빈번하여 적합
Read/write	✗	Entity 수정이 빈번하지 않으며, Lock으로 인한 성능 저하로 부적합
Transactional	✗	JTA를 위한 동시성 전략으로 부적합

현 상황에서 **Nonstrict Read/Write** 전략이 적합

그래서 Cache 구현체의 선택은 ?

**Invalidation Message Propagation 기능을 제공하는
IMDG 기반 Cache 구현체가 필요 !**

Cache 구현체 별 적합성에 대해 분석해보자

Local Cache & Invalidation Message Propagation 전략과 Cache 구현체 별 적합성 비교

이름	구분	저장소	적합여부	판단사유
Hazelcast	IMDG	In-Memory	✓	IMDG로 Invalidation Message Propagation 기능 제공
Infinispan	IMDG	In-Memory	✓	상동
Redis	NON-IMDG	In-Memory	✗	NON-IMDG로 Invalidation Message Propagation 기능 미제공
Ehcach	NON-IMDG	In-Memory	✗	상동
Ehcach & Terracotta	IMDG	In-Memory	✗	Terracotta 서버 추가 구성으로 인한 부담

Infinispan은 9.4.9.Final 기준
Invalidation Mode에서 NONSTRICT_READ_WRITE 미지원

NONSTRICT_READ_WRITE 전략을 지원하는
Hazelcast를 Local Cache 구현체로 결정

Chapter 3

Hazelcast 그리고 Second Level Cache

**Cache 구현체가 정해졌으니
구현체에 대해 파악해보자**

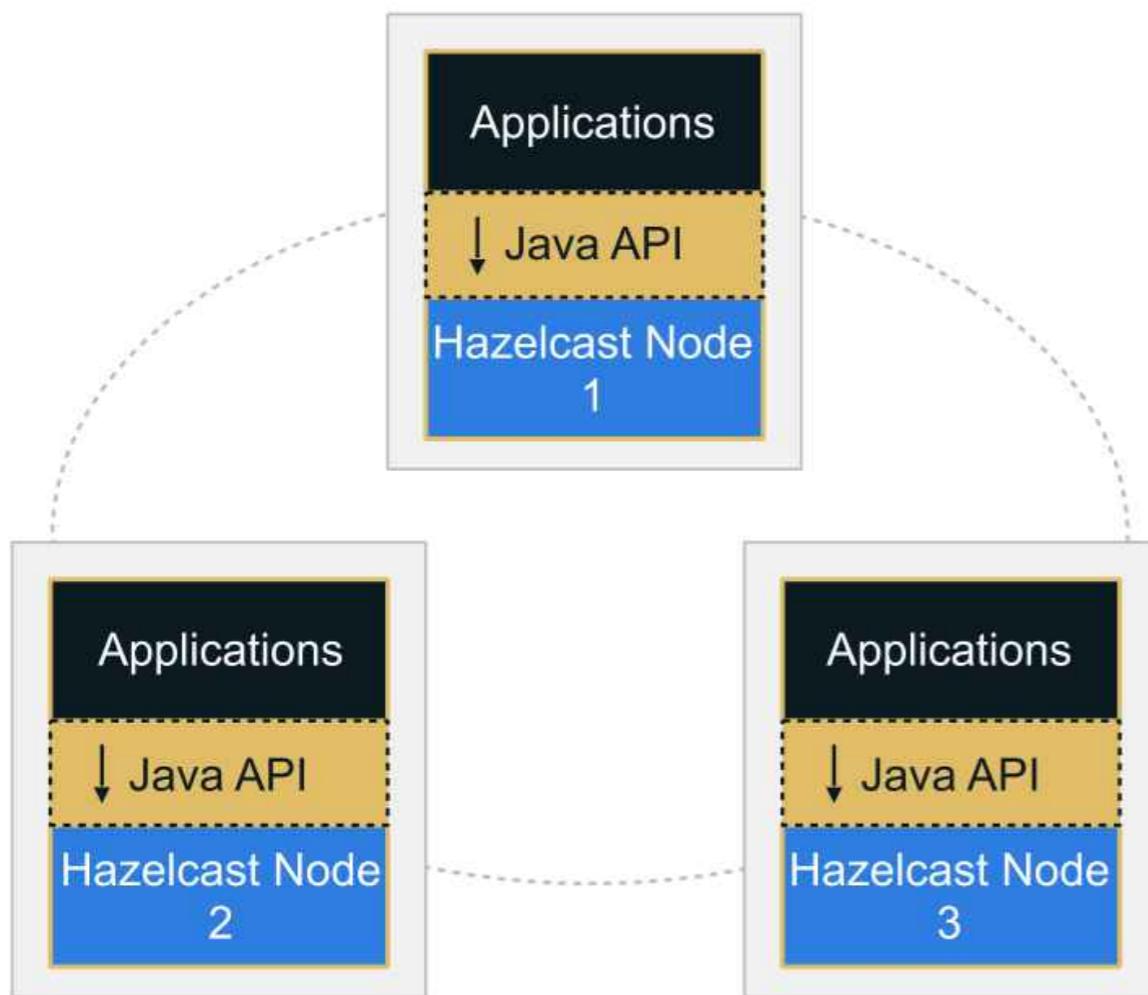
Hazelcast ?

- IMDG
- Open Source
- Java Base
- Multi Thread

Hazelcast 배포 방식

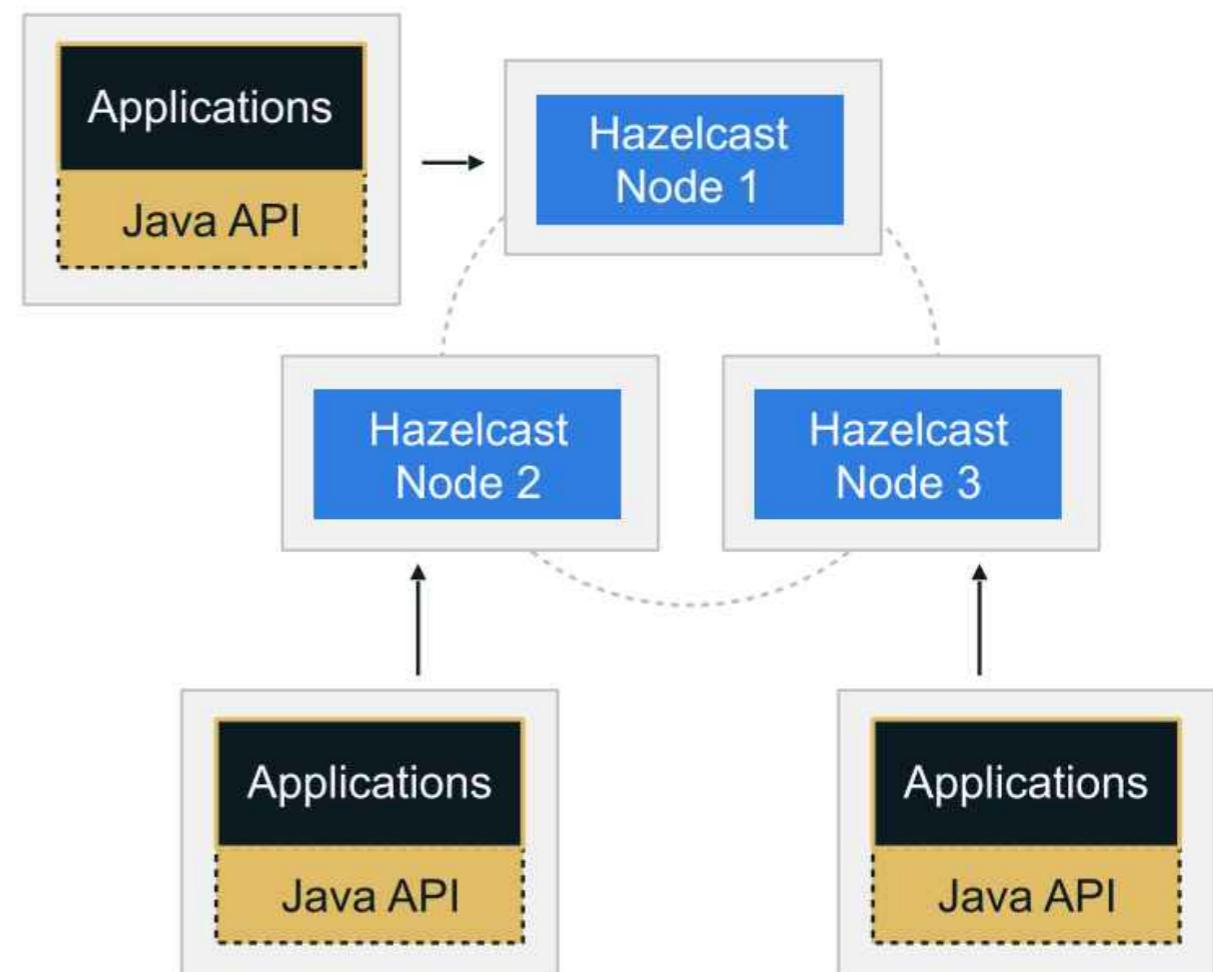
Deployment Options

Embedded Hazelcast



Great for early stages of rapid application development and iteration

Client-Server Mode



Necessary for scale up or scale out deployments – decouples upgrading of clients and cluster for long term TCO

Hazelcast 배포 방식의 선택

Embedded Mode

- 같은 JVM
- 조회시 Network Overhead X
- 조회시 Serialization Overhead X

Client-Server Mode

- 다른 JVM
- 조회시 Network Overhead O
- 조회시 Serialization Overhead O

Hazelcast 배포 방식 무엇이 더 적합한 상황 ?

- Second Level Cache로 사용되며 매우 잦은 Entity 조회

Hazelcast 배포 방식 무엇이 더 적합한 상황 ?

- Second Level Cache로 사용되며 매우 잦은 Entity 조회
- Network Latency를 0에 수렴할 목적 Embedded Mode 결정

Hazelcast 배포 방식 무엇이 더 적합한 상황 ?

- Second Level Cache로 사용되며 매우 잦은 Entity 조회
- Network Latency를 0에 수렴할 목적 Embedded Mode 결정
- Client-Server Mode를 선택할 바에는 AWS Elastic Cache

Hazelcast 클러스터링

TCP/IP

Multicast

AWS Cloud Discovery

GCP Cloud Discovery

Azure Cloud Discovery

etc ...

Hazelcast 클러스터링 방식의 선택

AWS Beanstalk 환경에서 운영중
따라서, **AWS Discovery** 방식으로 결정

hazelcast-aws의 Discovery SPI 기능을 활용하여
AWS 환경에서 클러스터링

AWS Discovery 기능으로 구체적으로 어떻게 클러스터링 할까 ?

```

public final class MetadataUtil {

    /**
     * This IP is only accessible inside AWS and is used to fetch metadata of running EC2 Instance.
     * Outside connection is only possible with the keys.
     * See details at http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-metadata.html.
     */
    public static final String INSTANCE_METADATA_URI = "http://169.254.169.254/latest/meta-data/";

    /**
     * Post-fix URI to fetch IAM role details
     */
    public static final String IAM_SECURITY_CREDENTIALS_URI = "iam/security-credentials/";
}

```

AWS에서 제공하는 API를 호출 인스턴스의 메타 정보를 가져와 TCP 통신을 통해 클러스터링

VPC Security Group의 Inbound TCP Port를 Open !

```

/**
 * Performs the HTTP request to retrieve AWS Instance Metadata from the given URI.
 *
 * @param uri          the full URI where a `GET` request will retrieve the metadata information, repr
 * @param timeoutInSeconds timeout for the AWS service call
 * @param retries       number of retries in case the AWS request fails
 * @return The content of the HTTP response, as a String. NOTE: This is NEVER null.
 */
public static String retrieveMetadataFromURI(final String uri, final int timeoutInSeconds, int retries) {
    return RetryUtils.retry(() -> {
        return retrieveMetadataFromURI(uri, timeoutInSeconds);
    }, retries);
}

```

AWS에서 Hazelcast 클러스터링 Flow

- Application 실행

AWS에서 Hazelcast 클러스터링 Flow

- Application 실행
- AWS IAM Role 기반으로 AWS에서 제공하는 API를 호출

AWS에서 Hazelcast 클러스터링 Flow

- Application 실행
- AWS IAM Role 기반으로 AWS에서 제공하는 API를 호출
- 이때, AWS Instance에 등록한 TAG 등으로 필터링

AWS에서 Hazelcast 클러스터링 Flow

- Application 실행
- AWS IAM Role 기반으로 AWS에서 제공하는 API를 호출
- 이때, AWS Instance에 등록한 TAG 등으로 필터링
- 인스턴스 N개의 메타 정보 수신

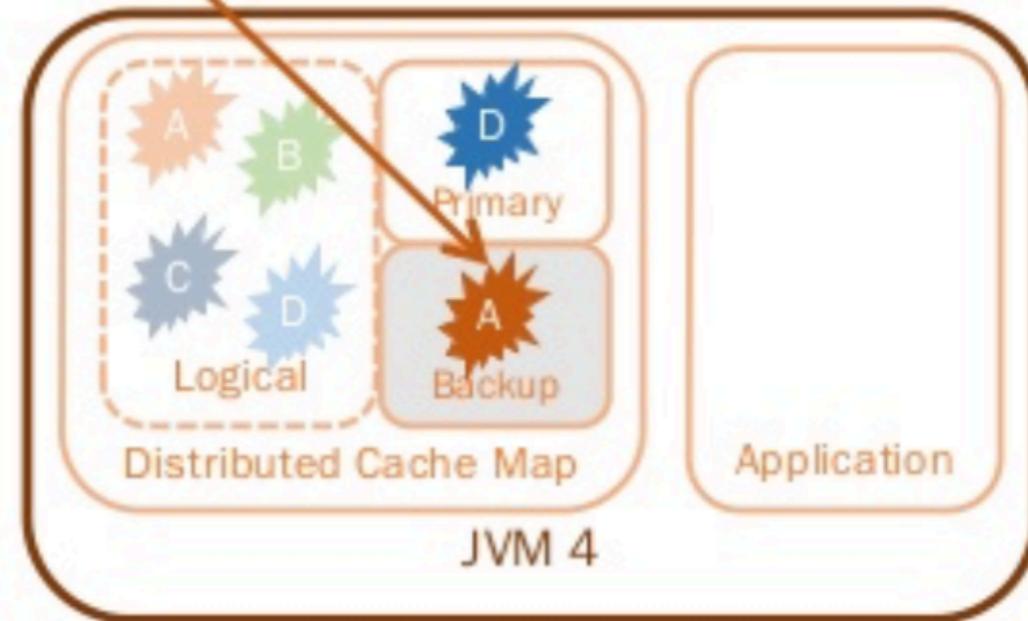
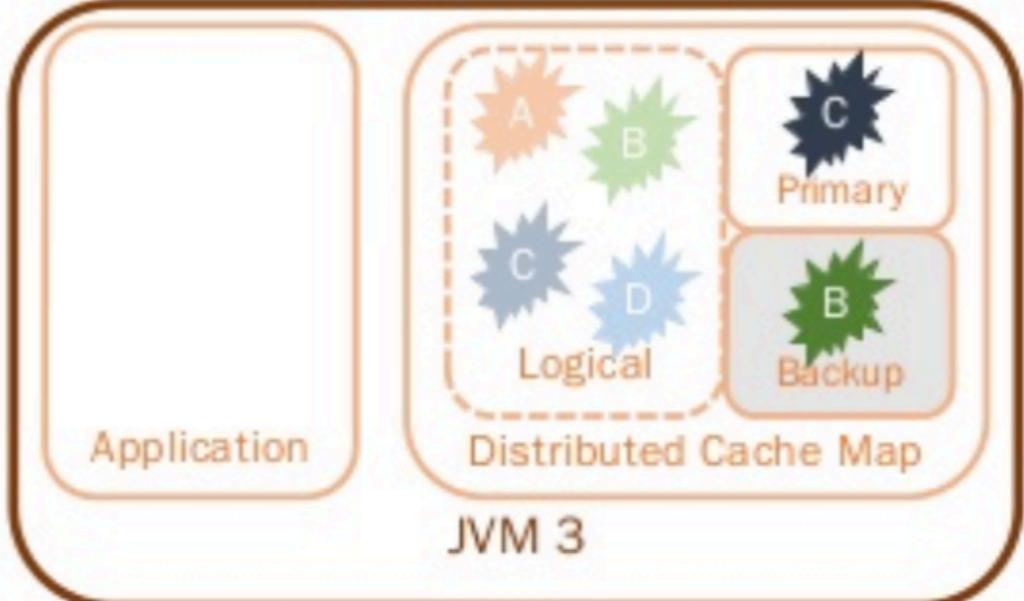
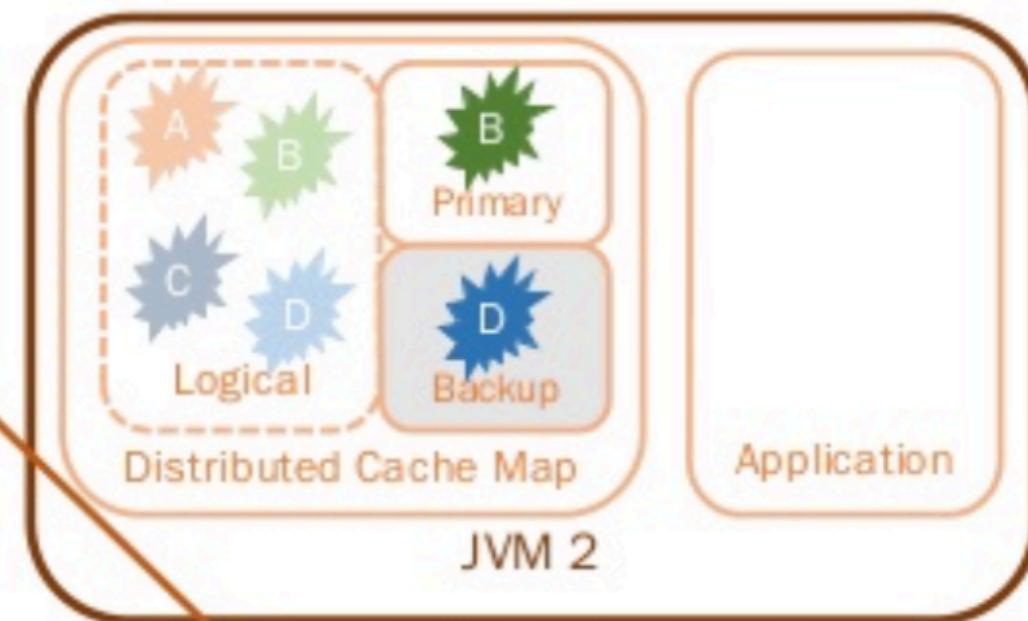
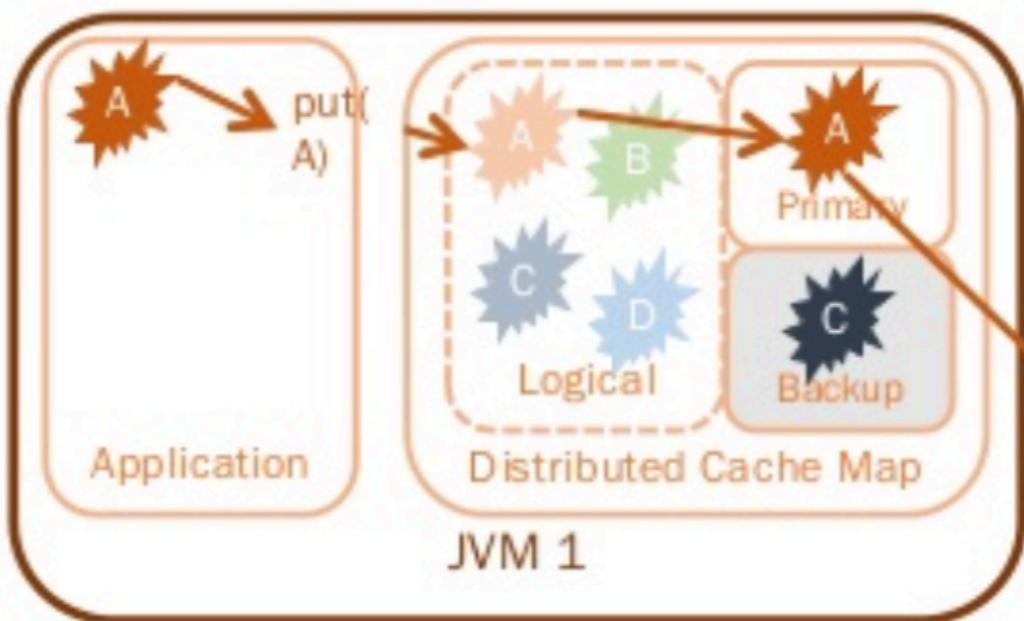
AWS에서 Hazelcast 클러스터링 Flow

- Application 실행
- AWS IAM Role 기반으로 AWS에서 제공하는 API를 호출
- 이때, AWS Instance에 등록한 TAG 등으로 필터링
- 인스턴스 N개의 메타 정보 수신
- 인스턴스 N개의 IP 주소로 TCP 통신을 사용해 클러스터링

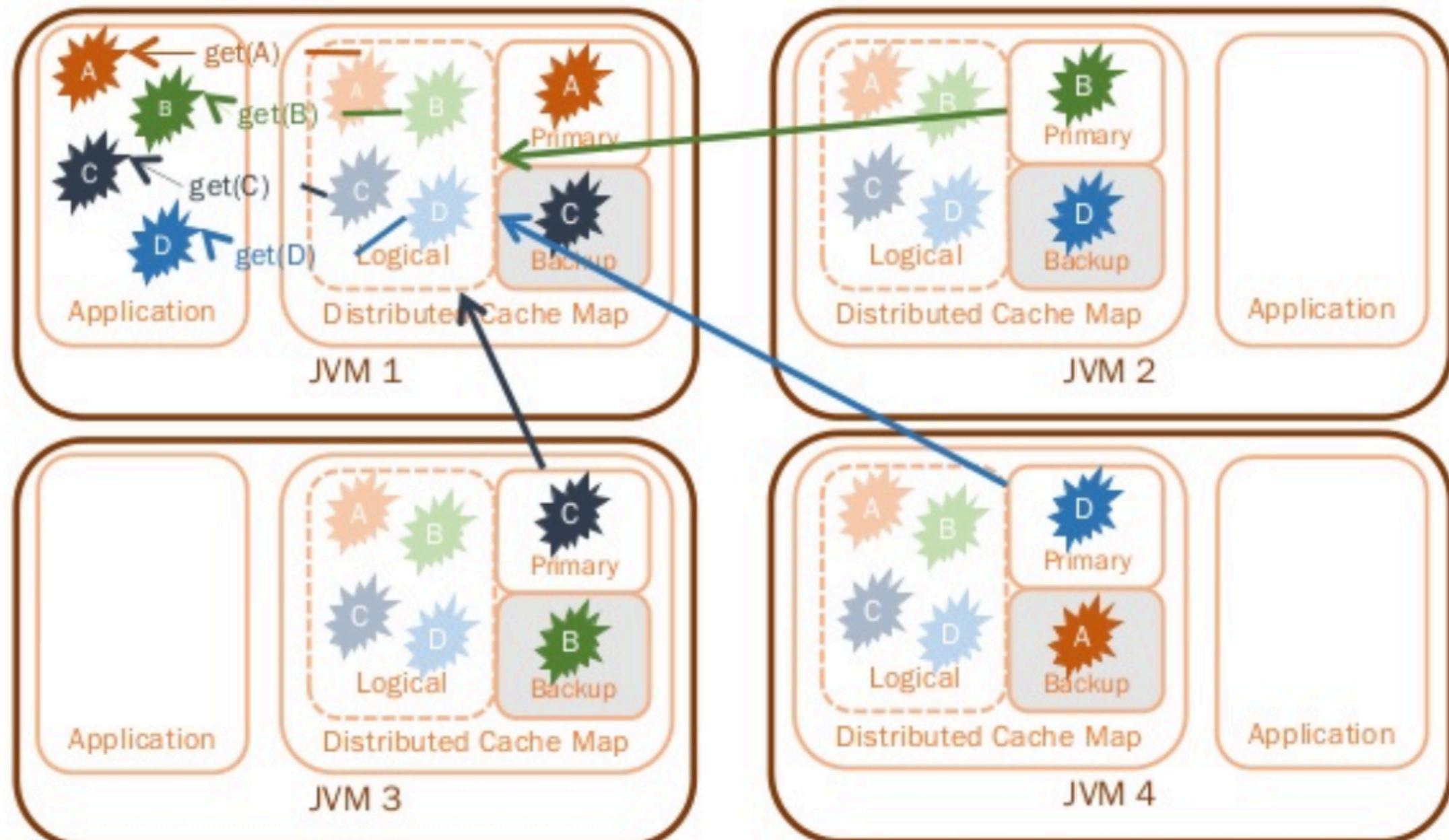
Hazelcast 데이터 적재 방식

분산 적재 ?

Put in Distributed Cache



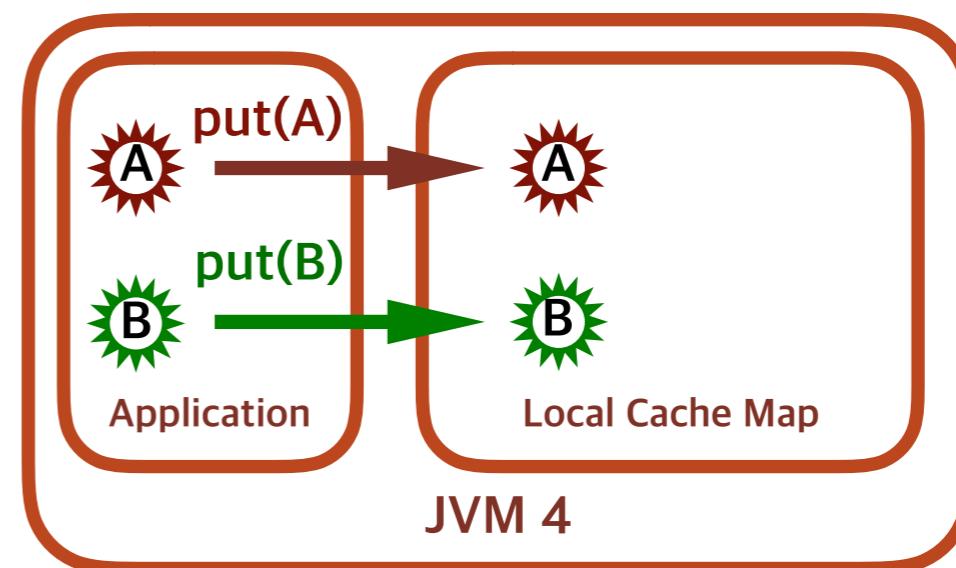
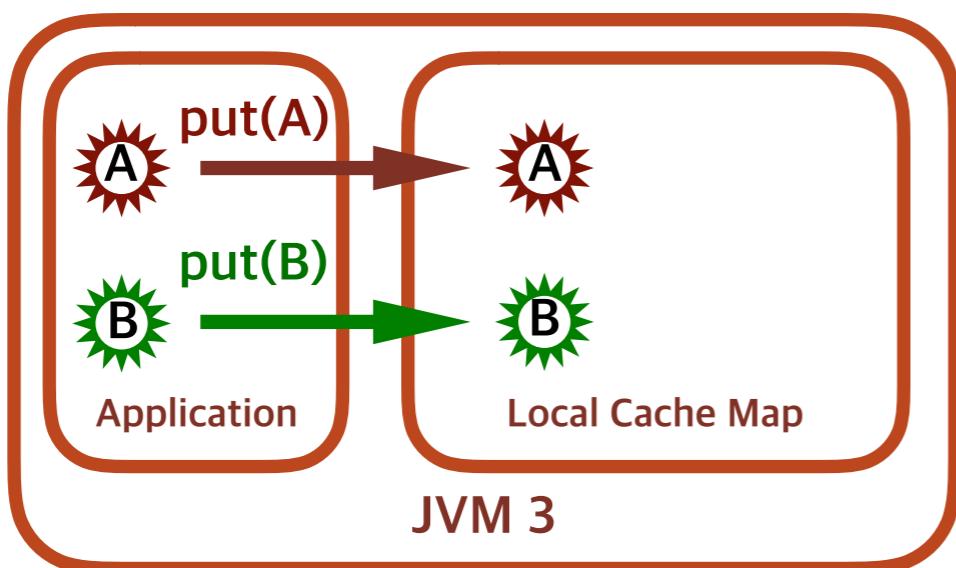
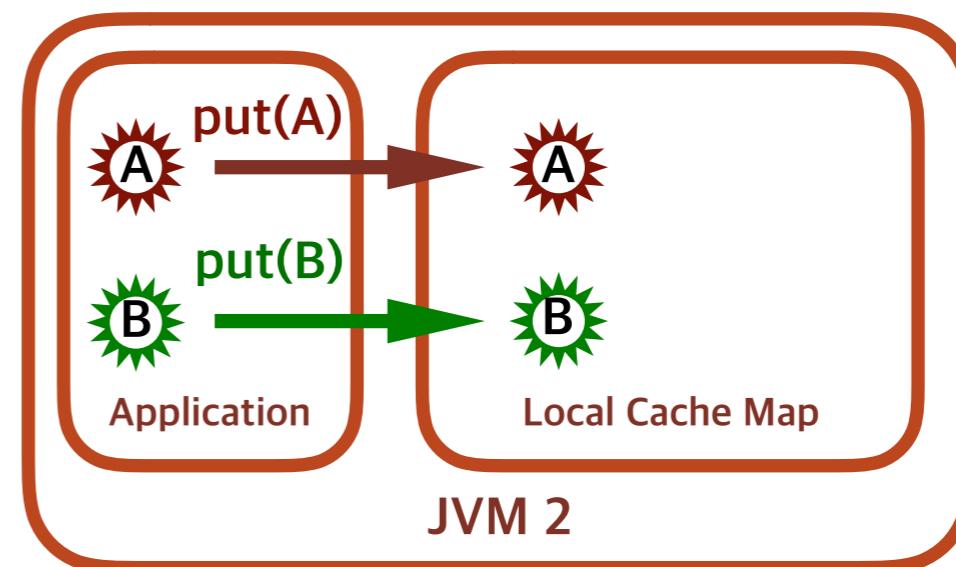
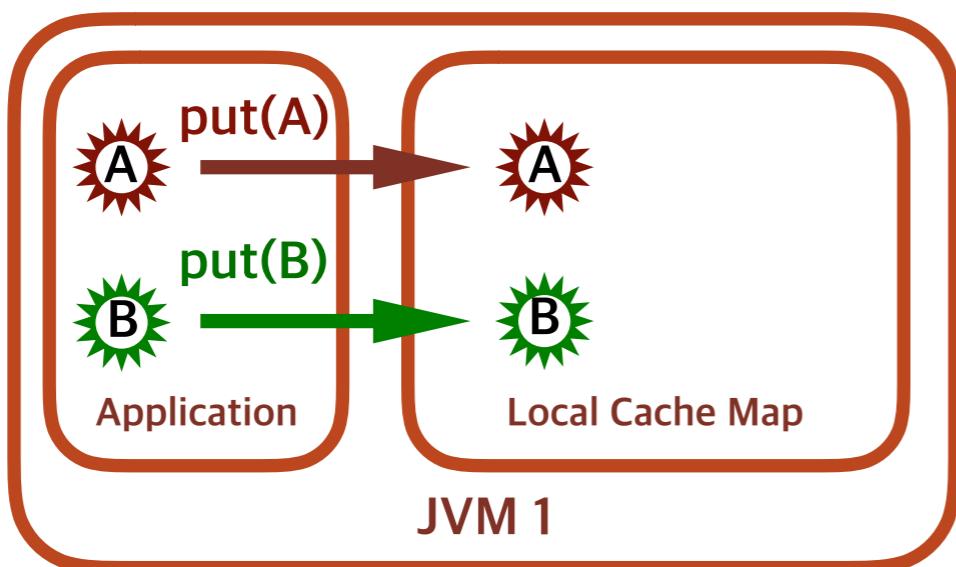
Get in Distributed Cache



LOCAL 적재 ?

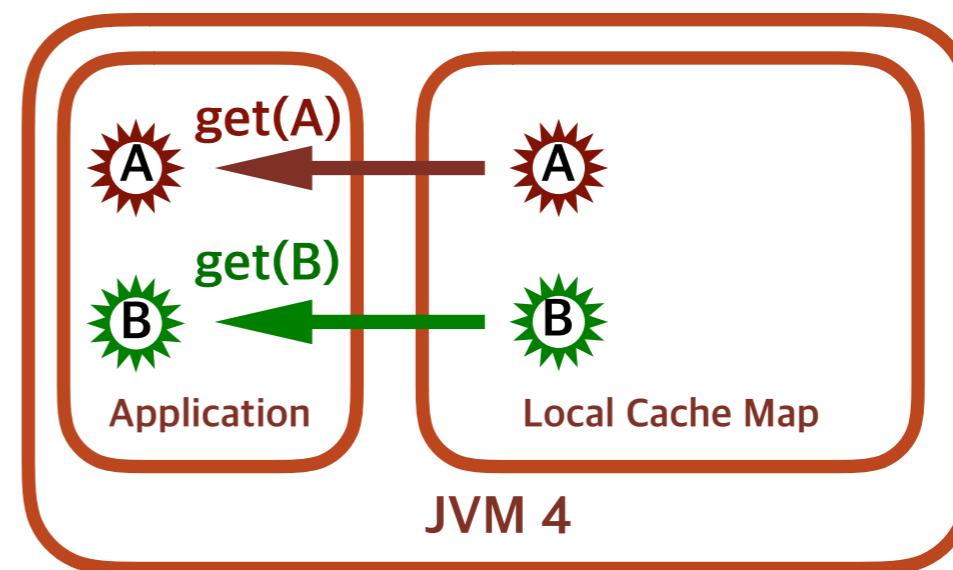
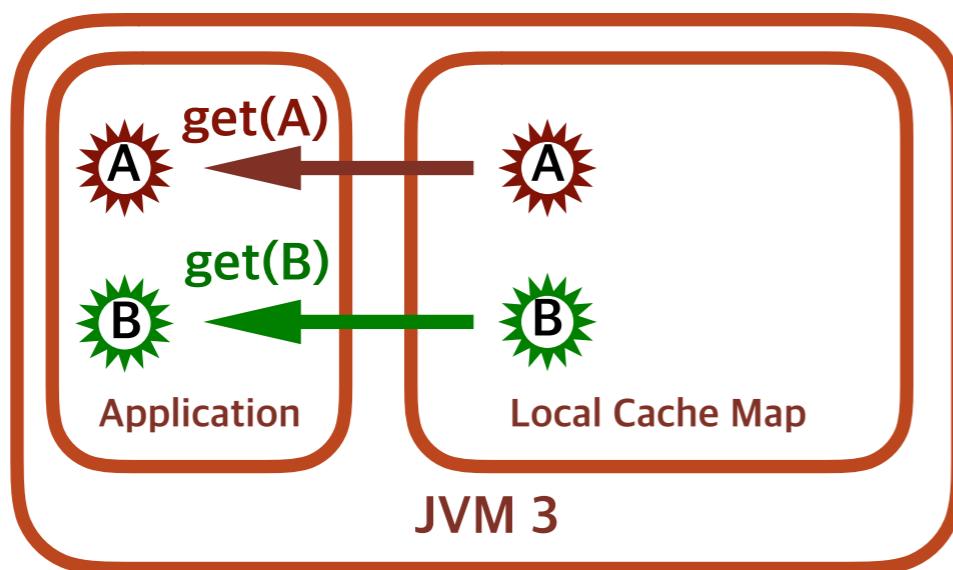
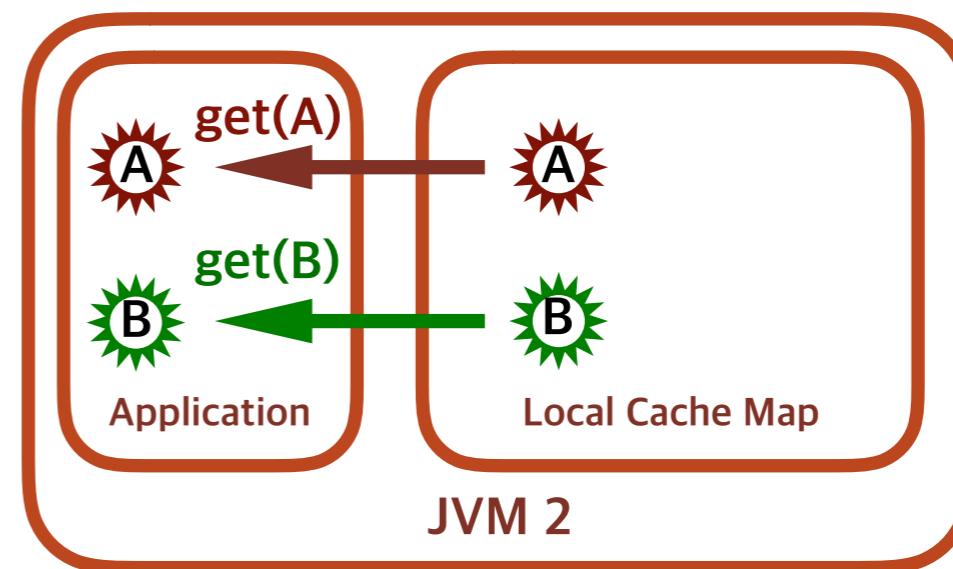
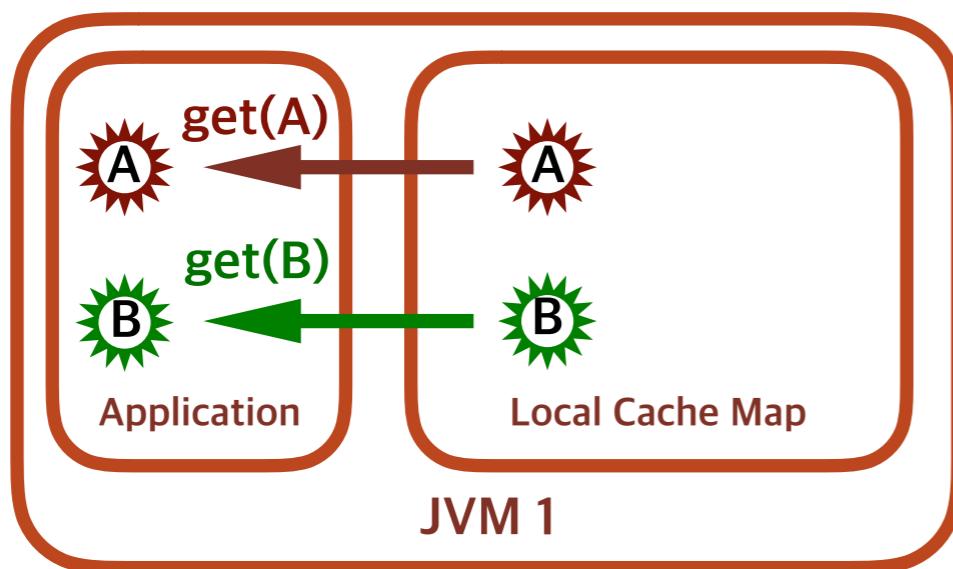
LOCAL 적재 - PUT

Put in Local Cache



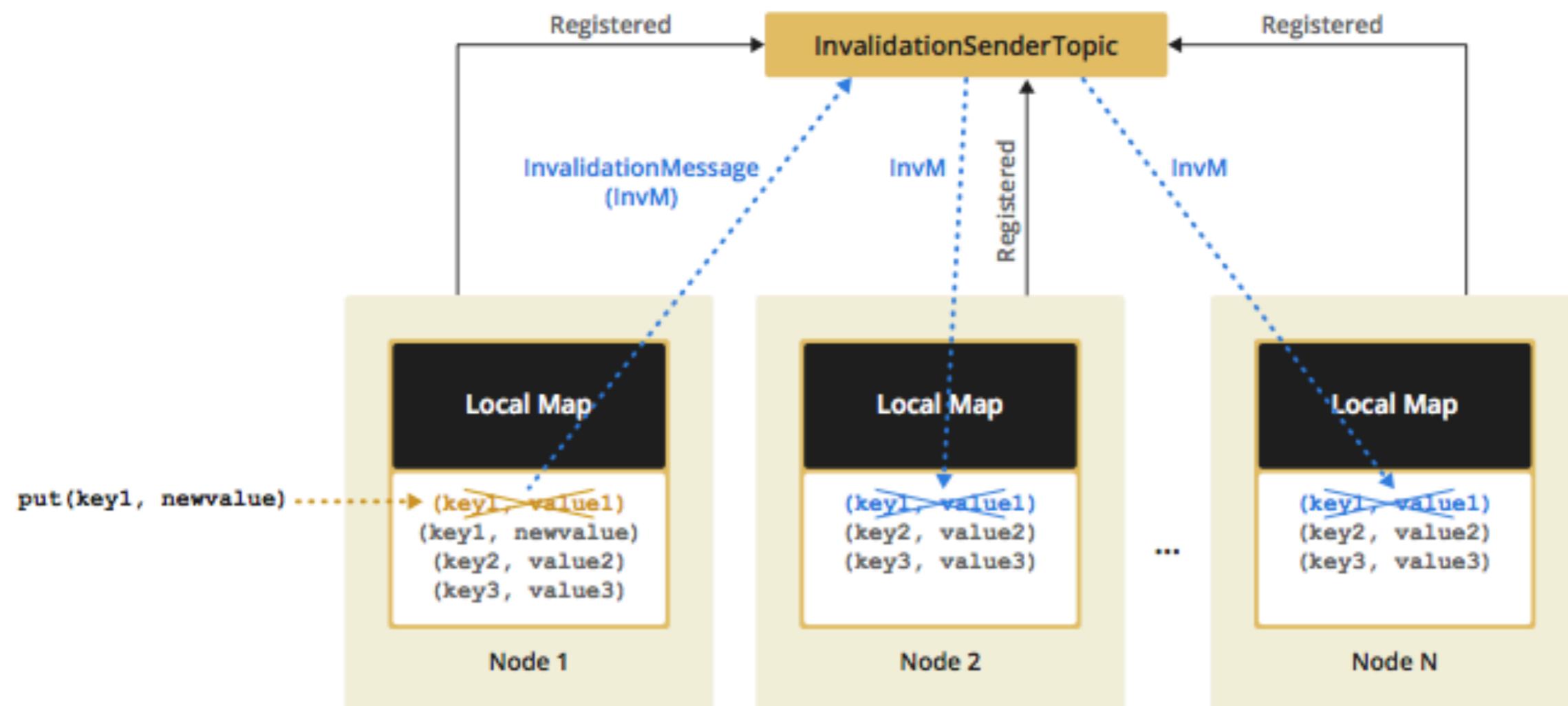
LOCAL 적재 - GET

Get in Local Cache



LOCAL 적재 - UPDATE

Update in Local Cache



Hazelcast 데이터 적재 방식의 결정

데이터 적재 방식 결정을 위해 고려한 부분은 두 가지
성능 그리고 AWS Beanstalk 환경에서의 배포 이슈 여부

Hazelcast 데이터 적재 방식의 결정

- 성능 측면
 - 성능을 위해 Network Overhead 없는 Local 처리 방식 적합

Hazelcast 데이터 적재 방식의 결정

- 성능 측면
 - 성능을 위해 Network Overhead 없는 Local 처리 방식 적합
- 배포 측면
 - 분산 처리 방식의 경우 데이터가 분산되어 저장
 - Entity 수정 후 **Rolling Update & Blue Green Deploy**시 역직렬화 이슈
 - 역직렬화 이슈를 회피하기 위해서는 Local 처리 방식 적합

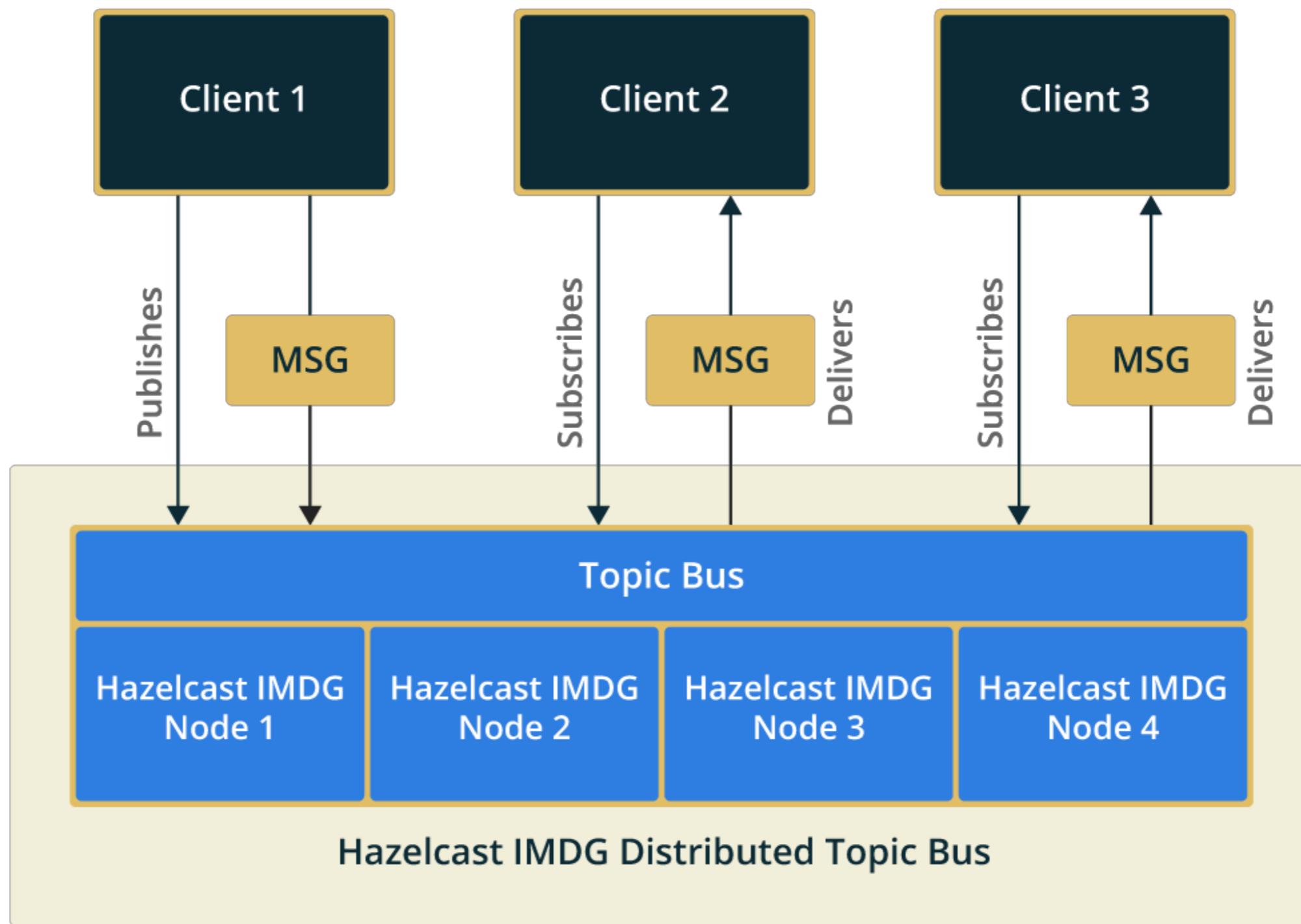
Hazelcast 데이터 적재 방식의 결정

- 성능 측면
 - 성능을 위해 Network Overhead 없는 Local 처리 방식 적합
- 배포 측면
 - 분산 처리 방식의 경우 데이터가 분산되어 저장
 - Entity 수정 후 Rolling Update & Blue Green Deploy 시 역직렬화 이슈
 - 역직렬화 이슈를 회피하기 위해서는 Local 처리 방식 적합
- 종합적으로 판단한 결과 Local 처리 방식이 더 적합하다는 결론

Hazelcast Distributed Event

Hazelcast Distributed Event

- Entity 수정시 클러스터링 된 다른 인스턴스에게 전파



Hazelcast Distributed Event 전송 방식

Hazelcast Distributed Event 전송 방식

- Clustering 된 Hazelcast Node들은 이벤트의 Subscriber를 알고 있으며 해당 Node의 IP 주소를 알고 있습니다

Hazelcast Distributed Event 전송 방식

- Clustering 된 Hazelcast Node들은 이벤트의 Subscriber를 알고 있으며 해당 Node의 IP 주소를 알고 있습니다
- 따라서, 특정 이벤트가 Publish 되면, 해당 이벤트를 Subscribe 하고 있는 Node들을 가져와서 Loop를 돌며 데이터를 전송

```
@Override
public void publishEvent(String serviceName, Collection<EventRegistration> registrations, Object event, int orderKey) {
    Data eventData = null;
    for (EventRegistration registration : registrations) {
        if (!(registration instanceof Registration)) {
            throw new IllegalArgumentException();
        }
        if (isLocal(registration)) {
            executeLocal(serviceName, event, registration, orderKey);
            continue;
        }

        if (eventData == null) {
            eventData = serializationService.toData(event);
        }
        EventEnvelope eventEnvelope = new EventEnvelope(registration.getId(), serviceName, eventData);
        sendEvent(registration.getSubscriber(), eventEnvelope, orderKey);
    }
}
```

Hazelcast Distributed Event 전송 방식

- **기본적으로 비동기 전송**

Hazelcast Distributed Event 전송 방식

- 기본적으로 비동기 전송
- 100,000 번째 요청마다 동기 전송을 수행 (10만, 20만...)

Hazelcast Distributed Event 전송 방식

- 기본적으로 비동기 전송
- 100,000 번째 요청마다 동기 전송을 수행 (10만, 20만...)
- **EVENT_SYNC_FREQUENCY = 100000**

Hazelcast Distributed Event 전송 실패 처리 방식

- 전송 실패 처리 방식에 따라 **신뢰도와 성능은 일반적으로 반비례**

	Nothing	Retry	Temporary Storage	Permanent Storage
성능 순위	1	2	3	4
신뢰성 순위	4	3	2	1
예시	X	Retry	In memory	In Disk

Hazelcast Distributed Event 전송 실패 처리 방식

- 전송 실패 처리 방식에 따라 신뢰도와 성능은 일반적으로 반비례

	Nothing	Retry	Temporary Storage	Permanent Storage
성능 순위	1	2	3	4
신뢰성 순위	4	3	2	1
예시	X	Retry	In memory	In Disk

- Local Cache + Invalidation 메세지 전파 방식은 **Retry 정책 사용**

Hazelcast Distributed Event 전송 실패 처리 방식

- 전송 실패 처리 방식에 따라 신뢰도와 성능은 일반적으로 반비례

	Nothing	Retry	Temporary Storage	Permanent Storage
성능 순위	1	2	3	4
신뢰성 순위	4	3	2	1
예시	X	Retry	In memory	In Disk

- Local Cache + Invalidation 메세지 전파 방식은 Retry 정책 사용
- 동기 전송 재시도 회수 50, 비동기 전송 재시도 회수 5

Hazelcast Distributed Event를 통한 Cache Invalidation Message Propagation 신뢰성 분석

Distributed Event를 통한 Cache 무효화 신뢰성 분석 - 1

- 인스턴스간의 Cache 무효화 메세지 도착 시간 차이로 인한 신뢰성
 - 당연하게도 미세한 시간 차이가 존재
 - 시간차로 이슈가 발생하지 않는 성격의 Entity를 적재해야 한다
 - 따라서, 이슈 없음

Distributed Event를 통한 Cache 무효화 신뢰성 분석 - 1

- 인스턴스간의 Cache 무효화 메세지 도착 시간 차이로 인한 신뢰성
 - 당연하게도 미세한 시간 차이가 존재
 - 시간차로 이슈가 발생하지 않는 성격의 Entity를 적재해야 한다
 - 따라서, 이슈 없음
- 네트워크 이슈로 Cache 무효화 메세지를 받지 못해 생길 수 있는 문제
 - TCP 전송 사용, 패킷 유실 등은 프로토콜에서 재처리
 - 따라서, 이슈 없음

Distributed Event를 통한 Cache 무효화 신뢰성 분석 - 2

- 일시적 500에러로, 로드밸런서에서 인스턴스 제외 후 재 포함된 경우
 - LB 통하지 않고, 인스턴스간 직접 통신으로 Eviction 정상 처리
 - 따라서, 이슈 없음

Distributed Event를 통한 Cache 무효화 신뢰성 분석 - 2

- 일시적 500에러로, 로드밸런서에서 인스턴스 제외 후 재 포함된 경우
 - LB 통하지 않고, 인스턴스간 직접 통신으로 Eviction 정상 처리
 - 따라서, 이슈 없음
- 동기 & 비동기 이벤트가 최대 재시도 회수인 50회, 5회를 넘길 경우
 - 장애 상황으로 판단
 - 몇번까지 Retry를 할 것인가는 정책
 - 따라서, 이슈 없음

Hazelcast Second Level Cache 적용

Hazelcast Second Level Cache 적용

- Maven

```
<dependencies>
    <!-- AWS -->
    <dependency>
        <groupId>com.amazonaws</groupId>
        <artifactId>aws-java-sdk</artifactId>
        <version>${aws-java-sdk.version}</version>
    </dependency>

    <!-- Hazelcast -->
    <dependency>
        <groupId>com.hazelcast</groupId>
        <artifactId>hazelcast</artifactId>
        <version>${hazelcast.version}</version>
    </dependency>
    <dependency>
        <groupId>com.hazelcast</groupId>
        <artifactId>hazelcast-hibernate52</artifactId>
        <version>${hazelcast.hibernate.version}</version>
    </dependency>
    <dependency>
        <groupId>com.hazelcast</groupId>
        <artifactId>hazelcast-spring</artifactId>
        <version>${hazelcast.version}</version>
    </dependency>
    <dependency>
        <groupId>com.hazelcast</groupId>
        <artifactId>hazelcast-aws</artifactId>
        <version>${hazelcast-aws.version}</version>
    </dependency>
```

Hazelcast Second Level Cache 적용

- YML

```
spring:  
  jpa:  
    properties:  
      hibernate.cache.use_second_level_cache: true  
      hibernate.cache.hazelcast.instance_name: hazelcast-instance-live  
      hibernate.cache.region.factory_class: com.hazelcast.hibernate.HazelcastLocalCacheRegionFactory
```

Hazelcast Second Level Cache 적용

- Hazelcast Config

```
@EnableCaching
@Configuration
public class HazelcastConfig {

    @Bean
    public Config config() {
        Map<String, Comparable> properties = new HashMap<>();
        properties.put("region", "region");
        properties.put("tag-key", "tag-key");
        properties.put("tag-value", "tag-value");
        properties.put("hz-port", "5701-5701");

        NetworkConfig networkConfig = new NetworkConfig();
        DiscoveryStrategyConfig discoveryStrategyConfig = new DiscoveryStrategyConfig(
            new AwsDiscoveryStrategyFactory(), properties);

        networkConfig.getJoin().getDiscoveryConfig()
            .addDiscoveryStrategyConfig(discoveryStrategyConfig);

        Config config = new Config()
            .setProperty(GroupProperty.DISCOVERY_SPI_ENABLED.getName(), "true")
            .setInstanceId("hazelcast-instance-live")
            .setGroupConfig(new GroupConfig().setName("hazelcast-instance-live"))
            .setNetworkConfig(networkConfig);

        return config;
    }
}
```

Hazelcast Second Level Cache 적용

- Entity

```
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;

import javax.persistence.Cacheable;
import java.io.Serializable;

@Cacheable
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Entity implements Serializable {  
}
```

Chapter 4

Cache Hit율 극대화 전략의 적용

Cache Hit율 극대화 전략의 적용 배경

Cache Hit율 극대화 전략의 적용 배경

- Spring Data JPA에서 2LC는 **findById()** PK 단건 조회만 HIT

Cache Hit율 극대화 전략의 적용 배경

- Spring Data JPA에서 2LC는 findById() PK 단건 조회만 HIT
- findAllById() PK 복수 조회는 Cache를 타지 않음

Cache Hit율 극대화 전략의 적용 배경

- Spring Data JPA에서 2LC는 findById() PK 단건 조회만 HIT
- findAllById() PK 복수 조회는 Cache를 타지 않음
- 쿠폰 API 에서는 findAllById()를 빈번하게 사용

Cache Hit율 극대화 전략의 적용 배경

- Spring Data JPA에서 2LC는 findById() PK 단건 조회만 HIT
- findAllById() PK 복수 조회는 Cache를 타지 않음
- 쿠폰 API 에서는 findAllById()를 빈번하게 사용
- 따라서, findAllById() 일때 Cache Hit율을 높이는 것이 핵심

Cache Hit율 극대화 전략의 적용 배경

구분	대상	Second Level Cache Hit
AS-IS	findAllById()	X
TO-BE	findAllById()	O

그래서 Cache Hit율 극대화 전략은 ?

**Cache에 있으면 바로 가져오고
없는 것들은 모아서 한번에 DB에서 가져오자**

그리고 그 전체 결과를 합쳐서 반환하자

Cache Hit율 극대화 전략의 적용 - Spring Data JPA

- Custom Repository

```
interface CouponCustomRepository {  
    List<Coupon> findAllByCachedId(Set<Long> ids);  
}
```

- extends Custom Repository

```
public interface CouponRepository extends JpaRepository<Coupon, Long>,  
    CouponCustomRepository {  
}
```

Cache Hit율 극대화 전략의 적용 - Spring Data JPA

- Custom Repository Impl

```
class CouponRepositoryImpl implements CouponCustomRepository {  
  
    @PersistenceContext  
    private EntityManager entityManager;  
  
    @Autowired  
    private CouponRepository repository;  
  
    @Override  
    public List<Coupon> findAllByCachedId(Set<Long> ids) {  
        if (ids.isEmpty()) {  
            return Collections.emptyList();  
        }  
  
        final Cache cache = entityManager.getEntityManagerFactory().getCache();  
        final List<Coupon> entities = Lists.newArrayList();  
        final Set<Long> notExistIds = Sets.newHashSet();  
  
        for (Long id : ids) {  
            if (cache.contains(Coupon.class, id)) {  
                repository.findById(id).ifPresent(entities::add);  
            } else {  
                notExistIds.add(id);  
            }  
        }  
  
        if (!notExistIds.isEmpty()) {  
            entities.addAll(repository.findAllById(notExistIds));  
        }  
  
        return entities;  
    }  
}
```

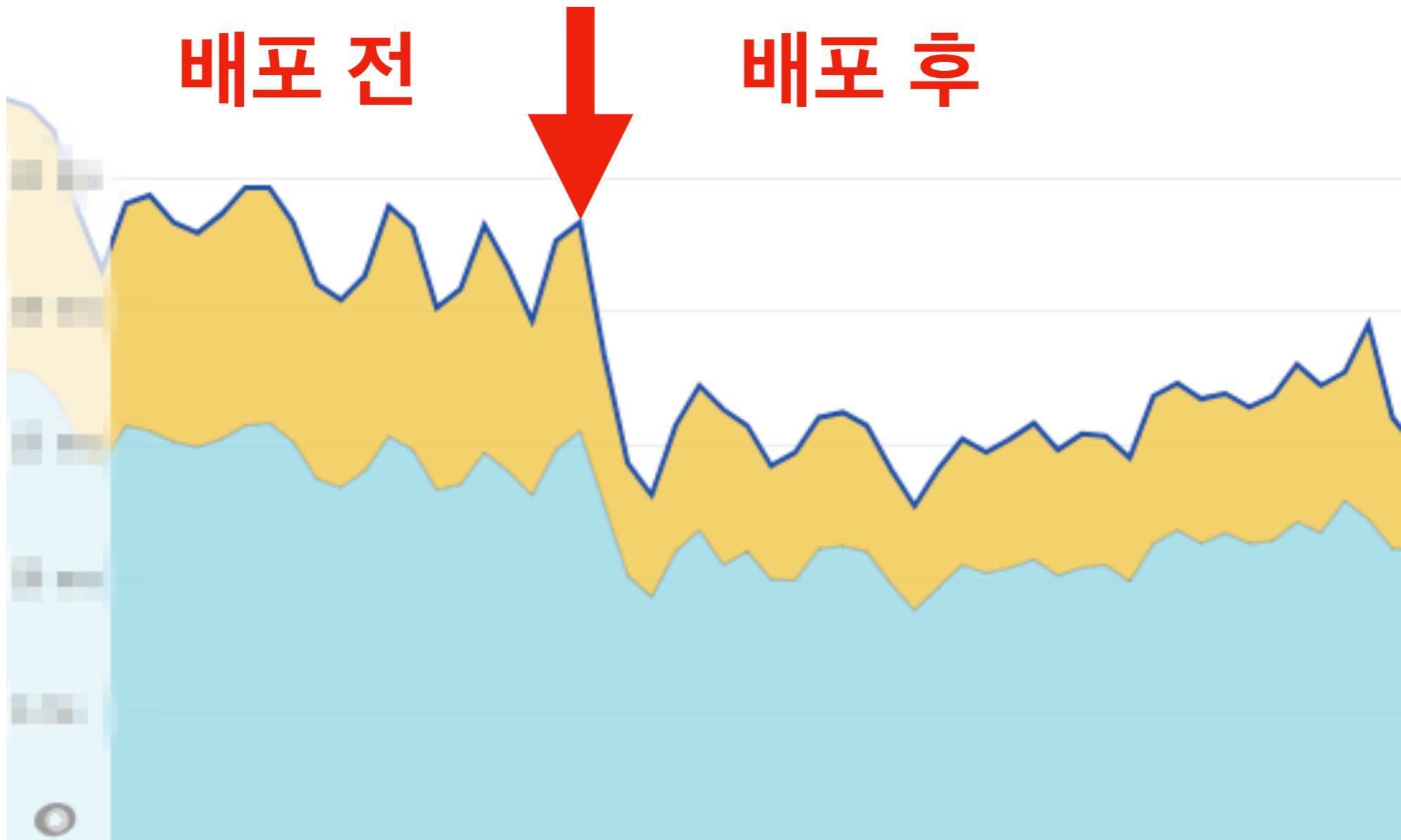
Chapter 5

Local Cache & Invalidation Message

Propagation 전략 도입 결과

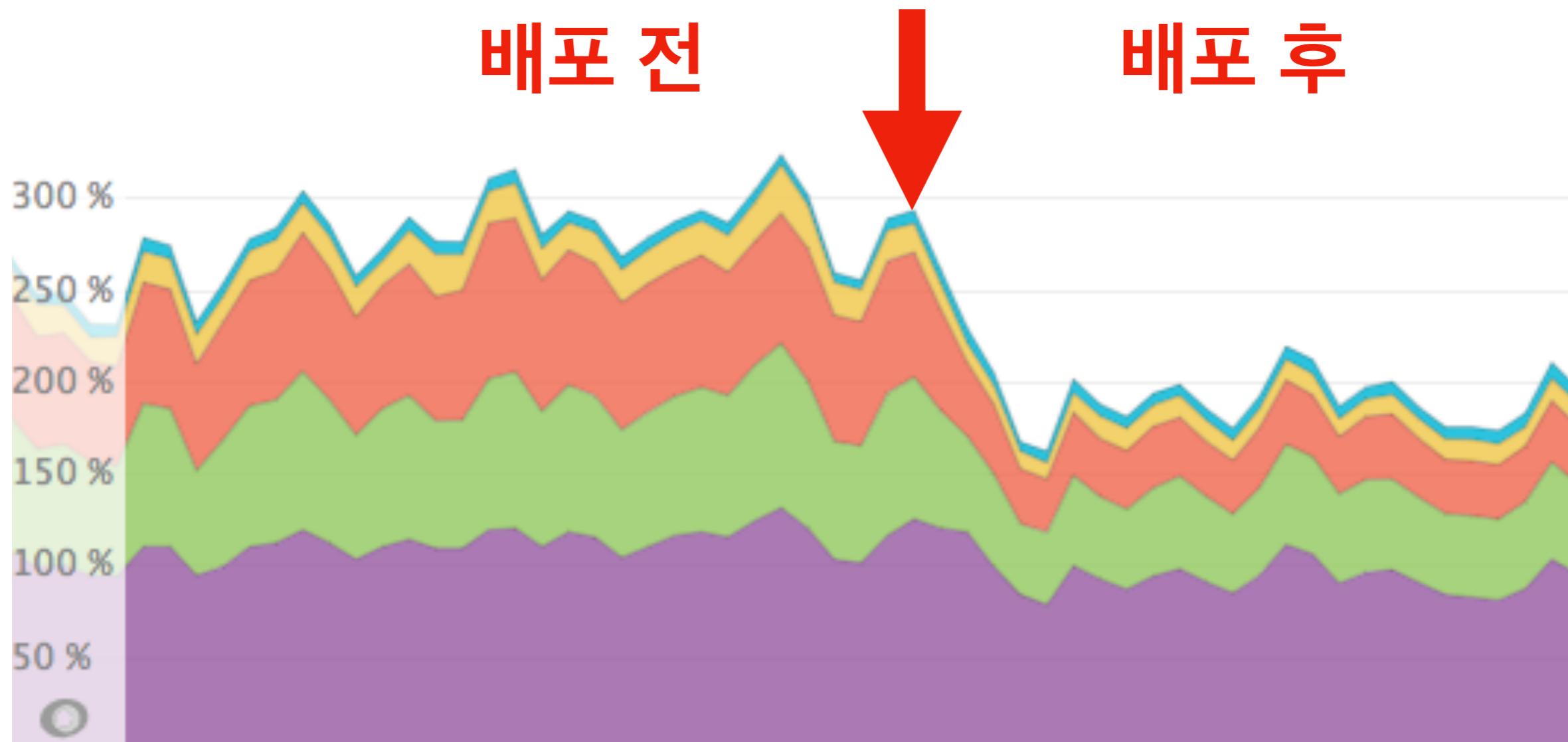
Local Cache & Invalidation 메세지 전파 전략 도입 결과

- 쿠폰 API 전체 Latency 변화
 - 전체 평균 100% 성능 향상



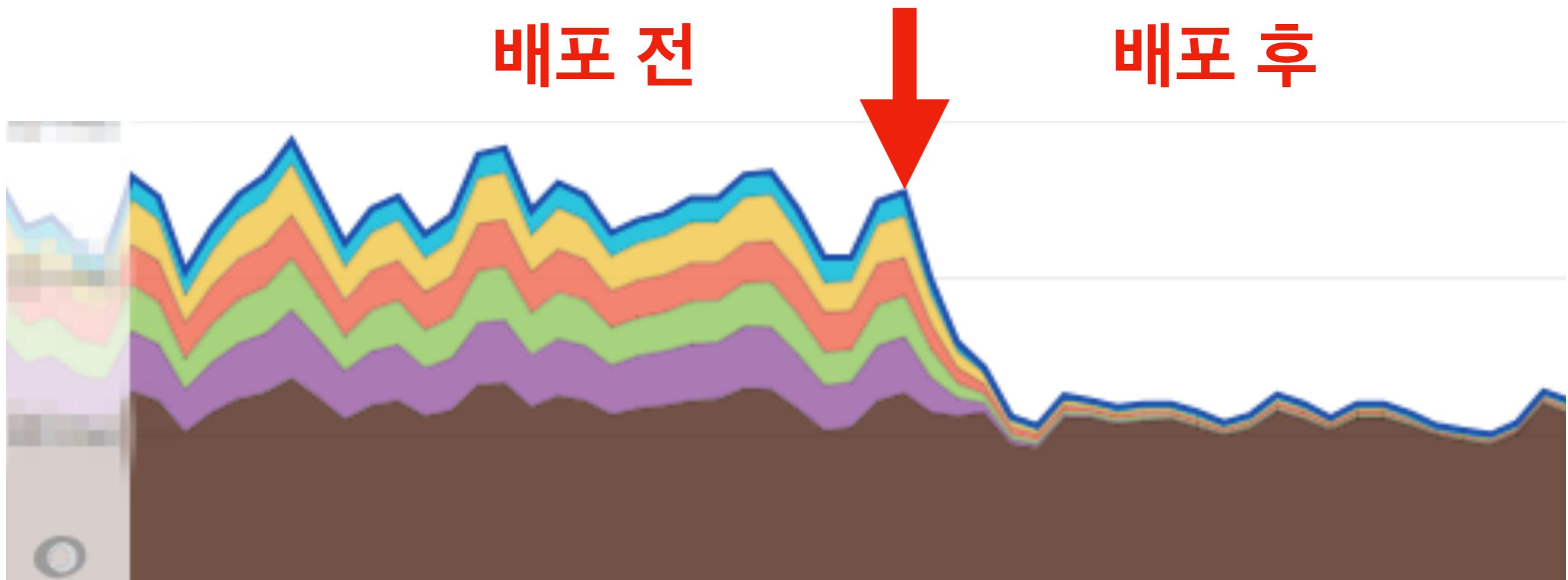
Local Cache & Invalidation 메세지 전파 전략 도입 결과

- TOP 5 트래픽 API Transaction 시간 변화



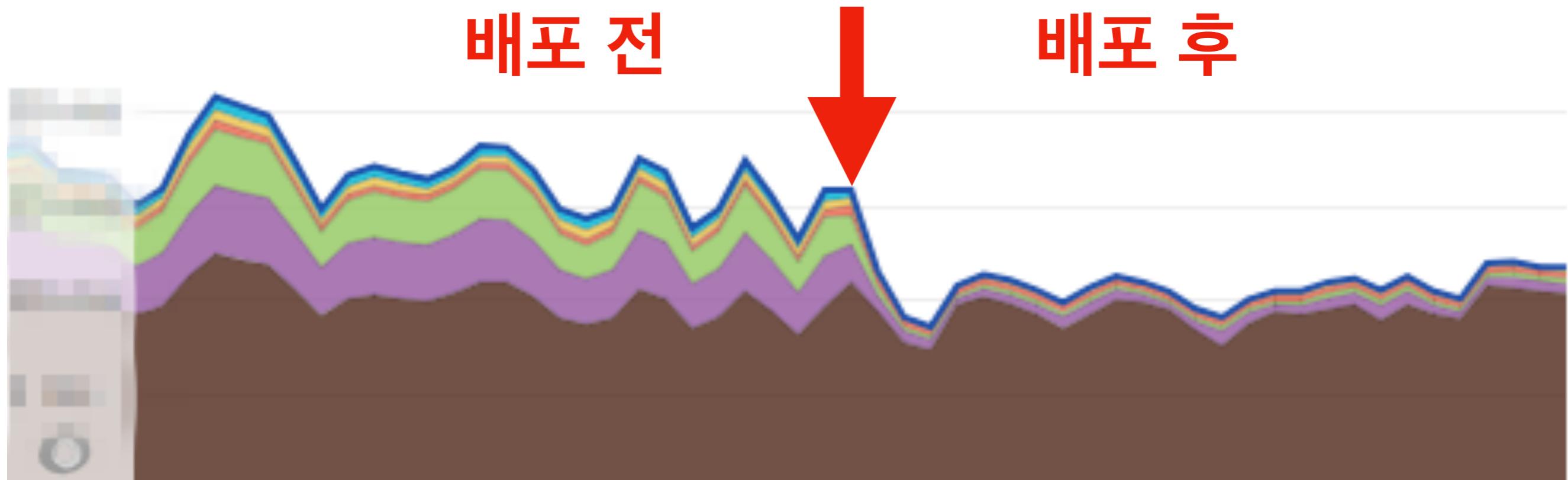
Local Cache & Invalidation 메세지 전파 전략 도입 결과

- 리스트용 쿠폰 실시간 집계 API 변화 (트래픽 비중 20%)
 - 150% 성능 향상



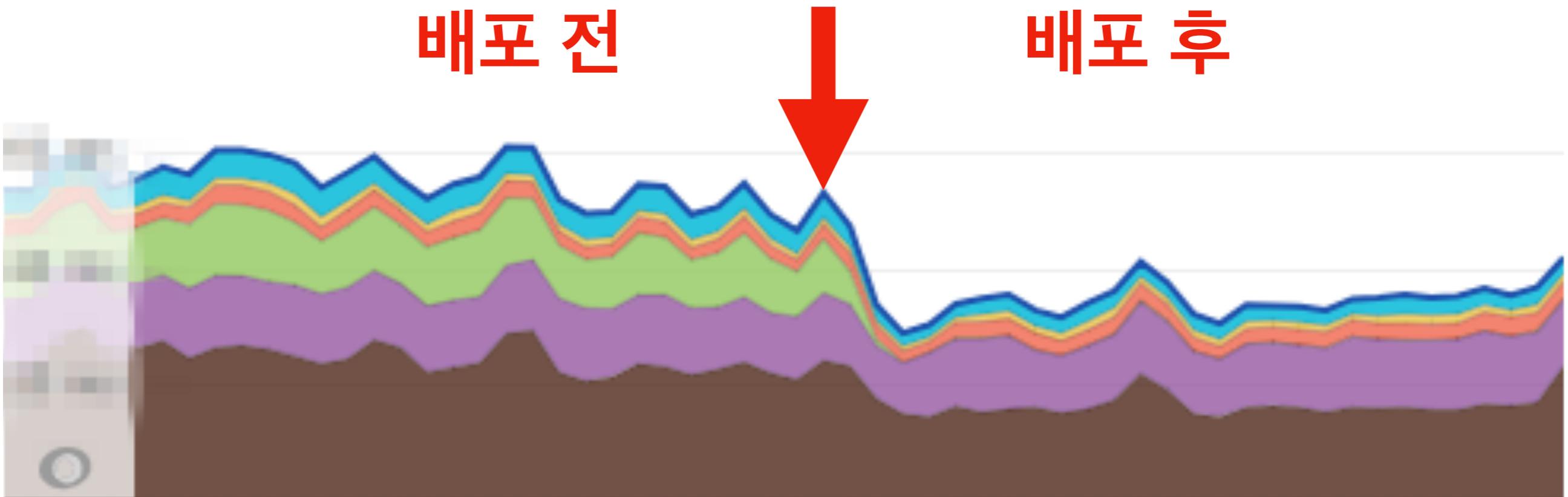
Local Cache & Invalidation 메세지 전파 전략 도입 결과

- 쿠폰 실시간 집계 API 변화 (트래픽 비중 27%)
 - 100% 성능 향상



Local Cache & Invalidation 메세지 전파 전략 도입 결과

- 예약시 사용가능한 쿠폰 조회 API 변화
 - 71% 성능 향상

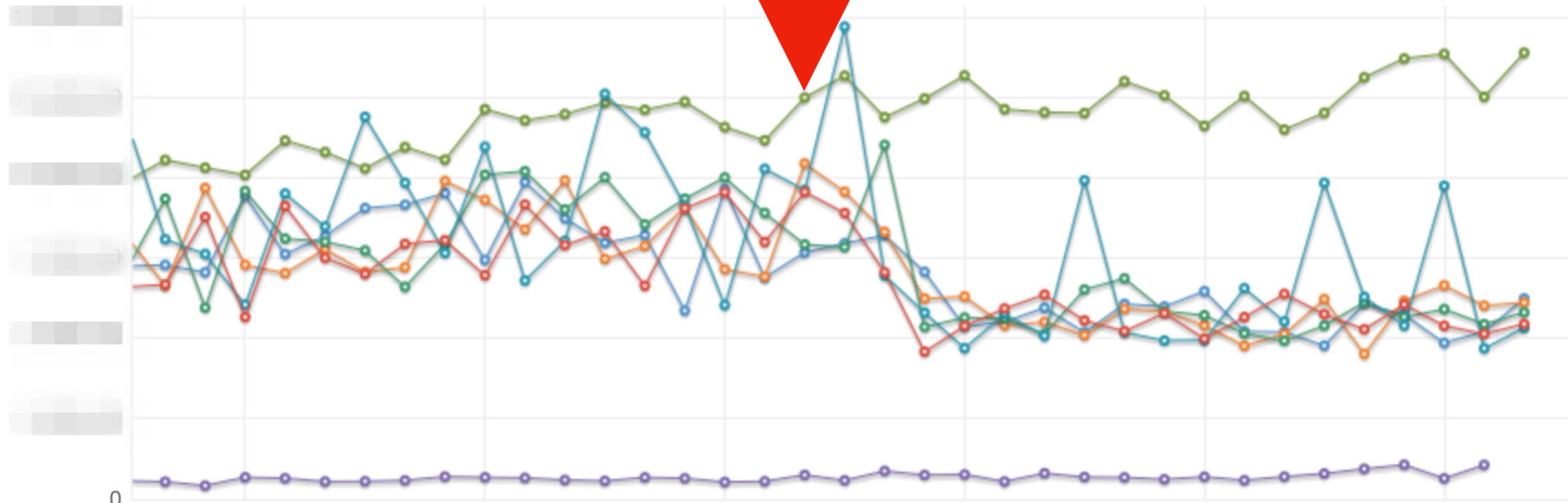


Local Cache & Invalidation 메세지 전파 전략 도입 결과

- Database 트래픽 변화
 - 30% 트래픽 경감

배포 전

배포 후



Cache Hit율 극대화 전략은 성능을 얼마나 향상 시켰을까 ?

- Entity 조회 성능 변화 - findAllById() N개의 PK 조회
 - Cache Hit 증가로 조회 성능 대폭 향상

적용 전 - 25MS

```
ids : 407, existIds : 0, notExistIds : 407, cacheHitRate : 0.0%
StopWatch 'findAllByCachedId': running time (millis) = 25
```

```
-----  
ms % Task name  
-----
```

```
00000 000% contains with ids : 407
```

```
00000 000% findById with existIds : 0
```

```
00025 100% findAllById with notExistIds : 407
```

적용 후 - 1MS

```
ids : 407, existIds : 407, notExistIds : 0, cacheHitRate : 100.0%
StopWatch 'findAllByCachedId': running time (millis) = 1
```

```
-----  
ms % Task name  
-----
```

```
00000 000% contains with ids : 407
```

```
00001 100% findById with existIds : 407
```

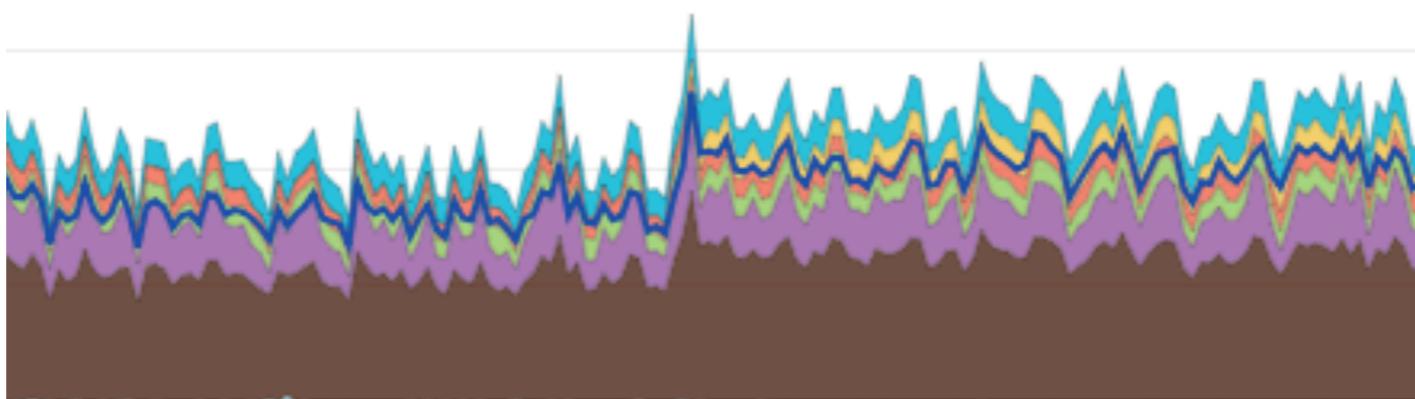
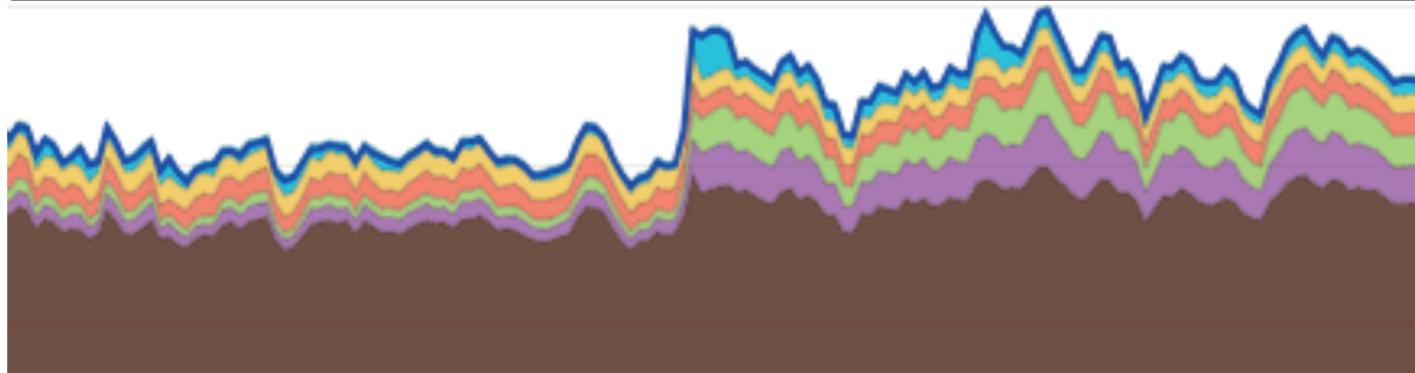
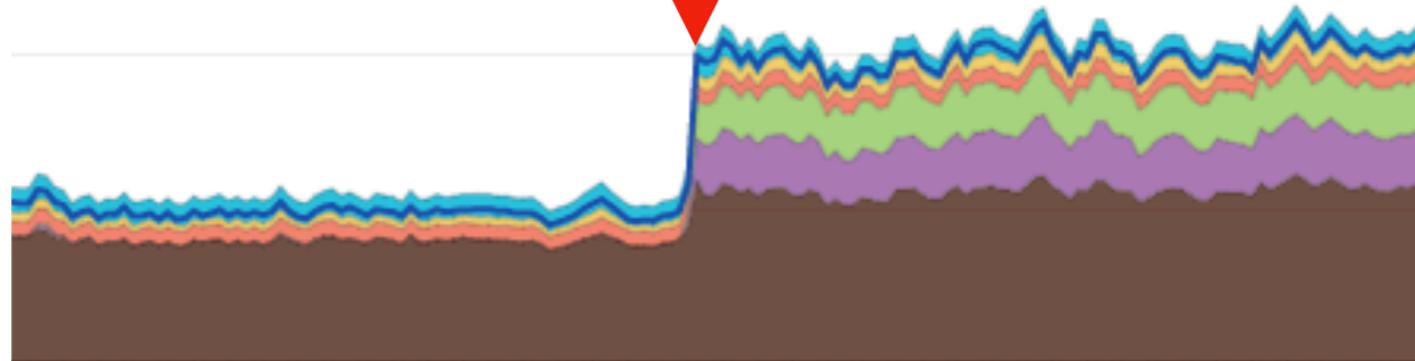
```
00000 000% findAllById with notExistIds : 0
```

Cache Hit율 극대화 전략은 성능을 얼마나 향상 시켰을까 ?

- Entity 조회 성능 변화 - findAllById() N개의 PK 조회
 - 2LC는 유지하고 findAllById() 튜닝만 제거하면 ?

튜닝 상태

튜닝 제거



Chapter 6

결론

그래서 Local Cache & Invalidation Message
Propagation 전략은 언제 써야 적절한건데 ?

Local Cache & 무효 메세지 전파 전략은 언제 써야 적절 ?!

- Entity 조회 > 수정

Local Cache & 무효 메세지 전파 전략은 언제 써야 적절 ?!

- Entity 조회 > 수정
- Entity가 1초에 수백 ~ 수천번 이상 조회될 경우

Local Cache & 무효 메세지 전파 전략은 언제 써야 적절 ?!

- Entity 조회 > 수정
- Entity가 1초에 수백 ~ 수천번 이상 조회될 경우
- Entity가 **Eventual Consistency**에 문제가 없을 경우

장점과 단점은?

장점 ?

성능
운영

인스턴스 추가 X

단점 ?

확장성
GC

결론 ?!

결론

- 조회가 매우 빈번한 Table에 대해 Local Cache & Invalidation Message Propagation 전략을 제한적으로 적용하여 자주 참조하는 Entity는 Local에서 처리하여 성능 향상 가능

결론

- 조회가 매우 빈번한 Table에 대해 Local Cache & Invalidation Message Propagation 전략을 제한적으로 적용하여 자주 참조하는 Entity는 Local에서 처리하여 성능 향상 가능
- GC 부담 최소화를 위해 적절한 Heap 사이즈를 설정하고 Max Cache Entry Size 제한을 통해 Application 내의 가용 메모리 유지 가능

결론

- 조회가 매우 빈번한 Table에 대해 Local Cache & Invalidation Message Propagation 전략을 제한적으로 적용하여 자주 참조하는 Entity는 Local에서 처리하여 성능 향상 가능
- GC 부담 최소화를 위해 적절한 Heap 사이즈를 설정하고 Max Cache Entry Size 제한을 통해 Application 내의 가용 메모리 유지 가능
- Result Cache 혹은 조회 빈도가 적은 데이터는 Remote Cache 추천

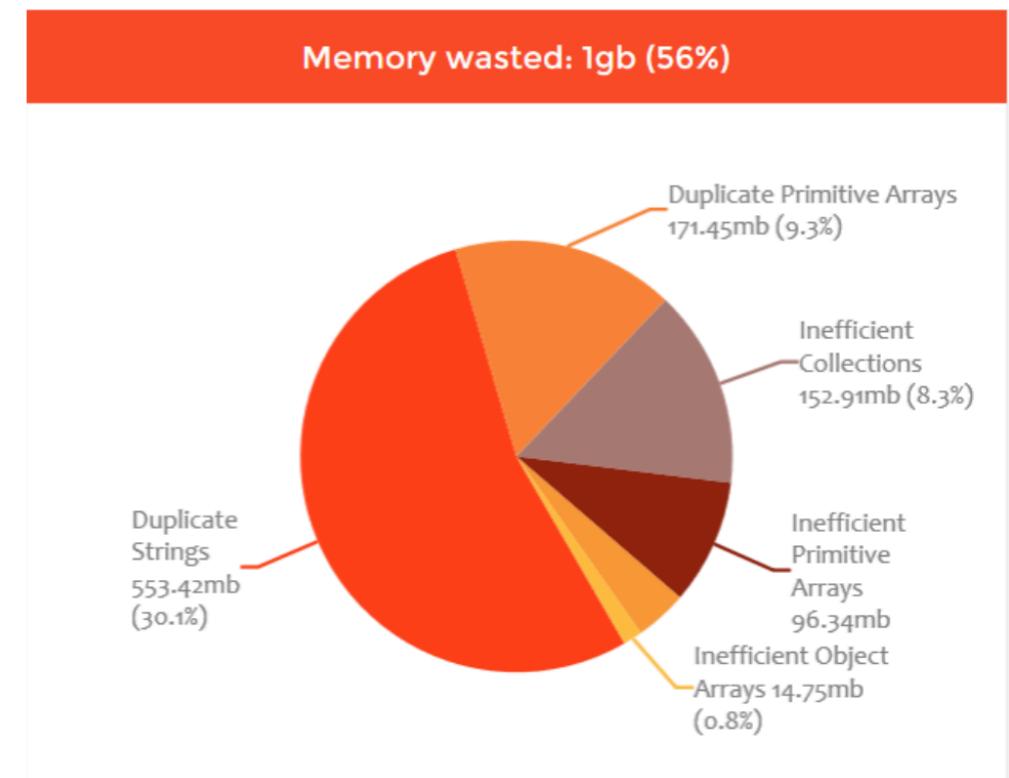
결론

- 조회가 매우 빈번한 Table에 대해 Local Cache & Invalidation Message Propagation 전략을 제한적으로 적용하여 자주 참조하는 Entity는 Local에서 처리하여 성능 향상 가능
- GC 부담 최소화를 위해 적절한 Heap 사이즈를 설정하고 Max Cache Entry Size 제한을 통해 Application 내의 가용 메모리 유지 가능
- Result Cache 혹은 조회 빈도가 적은 데이터는 Remote Cache 추천
- 이벤트와 같이 유사한 패턴의 트래픽이 쏟아질 수록 본 튜닝의 효과는 커짐
트래픽이 몰릴수록 평균 Latency는 점점 더 빨라지는 기이한 현상 체감 가능

튜닝 팁

JVM 튜닝 - Only G1 GC

- 먼저, Heap Size가 너무 큰 것은 아닌지 점검하자
- **-XX:MaxGCPauseMillis** (default : 200ms)
 - GC Pause Time을 줄이면, GC 빈도가 증가
 - GC Pause Time을 늘리면, GC 빈도가 감소
- **-XX:+UseStringDeduplication**
 - String duplication 제거, 가용 Heap 증가
 - 중복 제거로 GC Pause Time 다소 증가
 - 대신 Heap 차는 속도 저하로 GC 빈도 감소
 - String 중복 현황 파악 (<https://heaphero.io>)
 - <https://blog.gceasy.io/2018/12/23/usestringdeduplication/>



Q & A

We're Hiring!

야놀자 플랫폼실에서 개발자를 모십니다

<http://yanolja.in/recruitment/>

<https://www.wanted.co.kr/company/52>

pkgonan1@naver.com