Scaling Memcache at Facebook (2013.04.03)

https://research.facebook.com/publications/scaling-memcache-at-facebook/

1. 소개 (Introduction)

- 페이스북에서 사용되는 Memcache의 개요와 그 중요성에 대해 설명합니다. 소셜 네트워크의 높은 요구사항을 처리하기 위해 페이스북이 어떻게 Memcache를 사용하여 수억 명의 사용자 요청을 빠르고 효율적으로 처리하는지에 대한 전반적인 개요를 제공합니다.

2. 개요 (Overview)

- 페이스북의 Memcache 시스템이 어떻게 데이터베이스 및 백엔드 서비스와 상호작용하며, 읽기와 쓰기 작업이 각각 Memcache와 데이터베이스에서 어떻게 수행되는지 설명합니다. 읽기-쓰기 작업 흐름과 시스템 아키텍처의 전체적인 구조를 다룹니다.
- **데이터 읽기와 캐시 전략**
  - Memcache를 사용한 수요 기반 조회 캐시
  - 캐시 미스 시 데이터베이스에서 데이터 조회 후 캐시에 저장
- **캐시 무효화**
  - SQL 업데이트 후 캐시에서 해당 키 삭제
  - 삭제 방식으로 일관성 유지 및 캐시 데이터 관리
- **아키텍처 구성 요소**
  - Memcache 인스턴스를 분산 키-값 저장소로 확장
  - 구성, 집계, 라우팅 서비스를 통한 관리
    - **구성 서비스 (Configuration Service)**
      - 각 서버가 어떤 Memcache 인스턴스에 연결할지 설정하는 서비스.
      - 새로운 서버가 추가되거나, 서버 구성 변경이 필요할 때 전체 시스템에 이를 **자동으로 반영**해 일관성을 유지.
    - **집계 서비스 (Aggregation Service)**
      - 여러 Memcache 인스턴스에서 **데이터를 모으고 합치는 역할**을 수행.
      - 예를 들어, 여러 서버에서 분산 요청을 처리하고 나서 **결과를 한곳에 집계**해 웹 서버가 하나의 응답으로 받을 수 있도록 함.
    - **라우팅 서비스 (Routing Service)**
      - 요청된 데이터의 키를 기반으로 **해당 데이터가 저장된 올바른 Memcache 인스턴스로 요청을 라우팅**.
      - 페이스북에서는 `mcrouter` 라는 프록시를 사용하여 각 요청이 올바른 Memcache 서버로 효율적으로 전달되도록 함.

- **읽기 트래픽 효율화**
  - 대부분이 조회 요청으로 데이터베이스 읽기 부하 감소
  - MySQL, HDFS 등 다양한 소스에서 캐시 활용
- **유연한 캐시 전략**
  - 머신러닝 결과를 캐시에 저장하여 애플리케이션 공유
    - 페이스북은 머신러닝 모델의 결과나 복잡한 계산의 중간 결과를 캐싱.
  - 튜닝, 서버 관리 없이 인프라 활용
- **확장 시 과제와 접근법**
  - 작은 규모에서는 데이터 복제 미필요
  - 서버 증가 시 최적 통신 스케줄 필요
    - 서버가 많아지면, 데이터 요청이 여러 서버에 분산되면서 **통신량이 급증**합니다.
    - 모든 서버가 동시에 다수의 Memcache 서버와 통신하게 되면 **인캐스트 혼잡(incast congestion)** 같은 네트워크 문제 발생 가능성 증가.
    - 이 문제를 해결하기 위해 **통신 요청을 효율적으로 스케줄링**하고, **병렬 처리 및 요청 일괄 처리(batch processing)** 등을 통해 통신을 최적화하는 방법이 필요.
    - 이를 위해 페이스북에서는 Memcache 클라이언트 측에서 요청을 동시에 처리할 수 있도록 **슬라이딩 윈도우(sliding window) 기법**을 사용하는 등 네트워크 과부하를 줄이는 전략을 사용
- **설계 목표**
  - 사용자 경험 및 운영 효율성 개선 목표
  - 일관성 유지와 응답성 조정 가능

3. 클러스터 내에서: 지연 시간과 부하 (In a Cluster: Latency and Load)

- 클러스터 내에서 수천 대의 서버에 걸친 Memcache 확장의 문제를 다룹니다. 지연 시간을 줄이고 캐시 누락으로 인한 부하를 최소화하기 위한 전략을 설명하며, 예를 들어 병렬 요청 및 일괄 처리와 같은 최적화 기술을 설명합니다.

1. 지연 시간 감소 전략

- **캐시 히트(Cache Hit) 및 미스(Cache Miss) 시 지연 시간 관리**:
  - 사용자 요청 시 캐시에 데이터가 있으면 빠르게 응답할 수 있지만(캐시 히트), 데이터가 없으면(캐시 미스) 데이터베이스나 다른 소스로부터 데이터를 불러와야 하므로 지연 시간이 발생.
  - 페이스북은 캐시 조회 성공률을 높이고, 조회 실패 시 지연 시간을 최소화하는 여러 기법을 사용.
    1. **슬라이딩 윈도우(Sliding Window) 기법**
       - **요청 수 조절**: 많은 데이터를 한 번에 요청할 때 서버에 과부하가 걸리지 않도록 **슬라이딩 윈도우**를 사용하여 동시 요청 수를 제한.
       - **효율적 요청 처리**: 클라이언트가 응답을 받으면 다음 요청을 전송하는 방식으로 네트워크 혼잡을 줄이며, 요청이 실패하거나 응답이 지연되면 윈도우 크기를 줄여 부하를 자

동 조절.
    - 이를 통해 캐시 조회 성공률을 높이며, 조회 실패 시 데이터베이스에 과도한 부하가 발생하지 않도록 조정.
2. **병렬 요청 및 일괄 처리(Batching)**
    - 사용자 요청 시 여러 데이터 항목이 필요할 경우, 각 항목을 개별적으로 요청하는 대신 **병렬로 요청**하여 네트워크 왕복 시간을 줄임.
    - **의존성 그래프(DAG)** 를 사용하여 서로 독립적인 데이터 항목은 동시에 요청, 의존 관계가 있는 항목은 순차적으로 요청함.
    - 평균적으로 하나의 요청에 약 24개의 키를 일괄 요청해, 여러 항목을 동시에 가져와 응답 시간을 단축.
3. **Gutter 시스템**
    - **장애 시 데이터 백업**: 캐시 서버 일부가 장애가 나면, 해당 요청을 **Gutter 서버**로 전송하여 사용자가 지속적으로 데이터를 조회할 수 있도록 지원.
    - Gutter 서버는 **전체 Memcache 서버의 약 1%**로 구성되며, 특정 서버가 장애가 발생할 때 해당 서버의 요청을 대신 처리하도록 설계되었습니다.
    - 장애 시마다 특정 Memcache 서버의 역할을 임시로 맡아 데이터베이스 부하를 줄입니다.
    - 기존 캐시 서버가 장애가 발생하면 일반적으로 데이터베이스에 바로 요청이 몰리게 되지만, Gutter 서버가 대신 이를 처리하여 데이터베이스의 과부하를 방지.
        - Gutter 시스템을 사용함으로써 Memcache 서버가 일시적으로 응답하지 않더라도 Gutter 서버를 통해 조회를 처리하여 **캐시 실패율을 최대 99%까지 줄일 수 있음**.
4. **데이터 풀(Data Pool) 분리**
    - **자주 액세스되는 데이터와 그렇지 않은 데이터를 분리**: 자주 요청되는 데이터와 드물게 요청되는 데이터를 다른 캐시 풀에 저장하여 캐시 메모리의 효율을 극대화.
    - **고속 접근과 비용 절감**: 자주 사용하는 데이터는 빠르게 조회할 수 있도록 설정하고, 덜 사용하는 데이터는 별도의 저비용 풀에 저장하여 메모리 사용을 최적화.
    - 이로써 캐시 조회 성공률이 높아져 데이터베이스 조회 빈도와 부하가 줄어듦.

- **Memcache 클라이언트 최적화**:
    - 각 웹 서버에 **Memcache 클라이언트**를 배포
        - 클라이언트요청의 직렬화, 압축, 라우팅, 에러 핸들링 등을 수행.
        - 이 클라이언트가 **서버 목록**을 보유하여 각 요청이 어떤 Memcache 서버로 가야 하는지를 빠르게 결정할 수 있도록 함.
- **UDP와 TCP 프로토콜 활용**:
    - 조회 요청(get)은 **UDP**를 사용해 연결 설정 없이 바로 통신, 패킷 손실 시 간단히 오류로 처리.
    - 데이터 변경(set, delete) 요청은 **TCP**를 사용해 신뢰성을 높임.
    - 이를 통해 조회 속도를 높이면서 데이터 무결성 보장.

2. 인캐스트 혼잡(incast congestion) 방지

- **모든 서버 간 통신**: 각 웹 서버가 다수의 Memcache 서버와 통신하는데, 많은 요청이 동시에 발생하면 네트워크 혼잡이 발생할 수 있음.
- **슬라이딩 윈도우(sliding window)** 기법:
  - 동시에 요청하는 키 개수를 제한하는 방식으로, 응답을 받으면 다음 요청을 보내는 구조.
  - 요청이 성공할 때는 **윈도우 크기 증가**, 실패 시 **윈도우 크기 감소**로 동적 조정.
  - TCP의 혼잡 제어 방식과 유사하며, 이를 통해 단시간에 과도한 요청이 네트워크를 압박하지 않도록 함.
- **요청 흐름 제어**:
  - 슬라이딩 윈도우를 통해 과도한 요청이 네트워크 스위치나 서버에 몰리지 않도록 함.
  - 최적의 윈도우 크기를 유지해 불필요한 지연 없이 네트워크 부하를 최소화.

3. 부하 감소 전략

- **데이터베이스 접근 빈도 감소**:
  - 캐시에 데이터가 없는 경우에만 데이터베이스나 다른 소스에서 데이터를 불러옴으로써 데이터베이스의 과부하를 방지.
- **임대(lease) 메커니즘**:
  - 특정 키의 데이터가 자주 업데이트될 때 **임대(lease)**라는 일종의 토큰을 클라이언트에 발행하여, 한 번에 하나의 클라이언트만 캐시에 데이터를 업데이트할 수 있도록 제한.
  - 여러 클라이언트가 동시에 같은 키를 요청할 경우, 첫 번째 클라이언트만 데이터 업데이트를 수행하고, 이후 요청은 대기 후 다시 캐시를 조회.
  - 이 메커니즘으로 인해 **스테일 데이터(stale set)** 문제와 **서버에 과도한 요청이 몰리는 현상(thundering herd)**을 줄임.
- **Memcache 풀(Pool)**:
  - 다양한 유형의 요청에 대응하기 위해 클러스터 내의 Memcache 서버를 여러 **풀(pool)**로 나누어 관리.
  - 예: 자주 액세스되는 데이터는 고속 풀에 저장, 덜 사용되는 데이터는 저속 풀에 저장하여 메모리 효율 극대화.
  - 캐시 미스가 적절히 발생하도록 풀의 특성을 조정하여 데이터 접근의 효율성을 높임.

4. 실패 관리(Failure Handling)

- **소규모 장애**:
  - 개별 서버가 일시적으로 응답하지 않을 경우, **Gutter 서버**로 요청을 재전송하여 장애가 전체 시스템에 미치는 영향을 최소화.
  - 장애가 발생한 서버가 복구되기 전까지 Gutter 서버가 데이터를 제공하며, 사용자 요청이 실패하지 않도록 보호.

- **대규모 장애**:
  - 전체 클러스터에 문제가 생길 경우, 사용자 요청을 다른 클러스터로 우회하여 Memcache에 대한 부하를 제거.
  - 이러한 방식을 통해 데이터베이스와 클러스터 간의 원활한 통신과 빠른 복구가 가능.

4. 영역 내에서: 복제 (In a Region: Replication)

In a Region: Replication은 페이스북에서 **지역 내 클러스터 간 데이터 복제(region-level replication)**를 통해 **데이터 일관성(consistency)**과 **부하 분산(load distribution)**을 유지하는 방식을 설명합니다. 이 방식은 주로 페이스북과 같은 대규모 환경에서 **클러스터(cluster)**를 효율적으로 구성하고, 데이터 조회와 무효화의 속도를 높이는 데 목적이 있습니다.

1. 지역 구조 (Region Architecture)

- 데이터 센터를 물리적 지역(region)으로 구분하여 운영
- 각 지역에 프론트엔드 클러스터(frontend cluster)와 스토리지 클러스터(storage cluster) 존재
  - 프론트엔드 클러스터: 사용자 요청 처리 역할
  - 스토리지 클러스터: 데이터베이스 포함한 영구 저장소 역할
- 지역 내 클러스터 간 데이터 복제 가능
- 지역 내 장애 발생 시 다른 지역으로 트래픽 우회 가능
  - 고가용성(high availability) 유지 가능

2. 클러스터 간 복제 (Inter-Cluster Replication)

- 동일한 지역 내 여러 프론트엔드 클러스터들이 동일한 데이터 사용
- 데이터 일관성(consistency) 유지를 위한 데이터 복제 필요
- 모든 클러스터가 최신 데이터 접근 가능하도록 지역 내 데이터 복제
- 특정 클러스터 장애 발생 시 다른 클러스터에서 데이터 제공 가능

4. 데이터 무효화 파이프라인 (Invalidation Pipeline)

- 데이터 최신 상태 유지 위한 무효화(invalidation) 작업 수행
- 데이터베이스에서 데이터 변경 시 무효화 데몬(invalidation daemon)이 각 클러스터에 삭제 명령 전달
- 무효화 데몬이 SQL 문에 포함된 삭제 정보(memcache key) 읽어들여 각 클러스터의 mcrouter에 전달
- mcrouter가 각 Memcache 서버에 무효화 명령 전달
- 무효화 지연(invalidation delay) 최소화하여 스테일 데이터 문제 방지

5. 무효화의 효율성 (Efficiency of Invalidation)

- 무효화 작업으로 인한 네트워크 트래픽 발생 방지 위해 무효화 요청을 배치(batch)로 묶어서 전송

- 클러스터 경계를 넘어서는 무효화 요청을 mcrouter가 수신하여 해당 클러스터의 올바른 서버로 라우팅
- 패킷 전송 효율성(packet efficiency) 향상
- 무효화 지연 최소화하여 빠른 데이터 일관성 확보 가능

6. 지역 풀 (Regional Pool)

- 메모리 사용량(memory usage) 절감 위해 지역 풀(regional pool) 도입
- 지역 내 모든 클러스터가 동일한 데이터를 복제하는 방식 대신 특정 데이터는 공유 메모리 풀(shared memory pool)에 저장
- 각 프론트엔드 클러스터가 지역 풀을 통해 데이터 접근
- 자주 조회되지 않으며 큰 크기를 차지하는 데이터는 지역 풀에 저장
- 메모리 사용량 줄이고 지역 내 중복 데이터(replication redundancy) 최소화

7. 데이터 조회 지연 및 비용 (Latency and Cost in Data Retrieval)

- 프론트엔드 클러스터별로 자체적으로 데이터 캐싱
- 공유 데이터의 경우 지역 풀에서 데이터 가져옴
- 클러스터 경계를 넘는 데이터 조회 시 약간의 지연(latency) 발생 가능
  - 자주 액세스되는 데이터(frequently accessed data)는 각 클러스터에 개별적으로 복제하여 빠른 조회 가능
- 덜 자주 사용되는 데이터는 지역 풀로 관리하여 메모리 사용 비용 절감

8. 콜드 클러스터 웜업 (Cold Cluster Warmup)

- 새로운 클러스터 추가 또는 장애 복구 후 클러스터가 빈 상태로 복구되는 경우 콜드 스타트(cold start) 문제 발생 가능
  - 콜드 스타트 문제 해결 위해 콜드 클러스터 웜업(cold cluster warmup) 시스템 사용
  - 빈 클러스터가 데이터베이스 대신 따뜻한 클러스터(warm cluster)에서 데이터 가져와 초기 캐시 상태 빠르게 구축
- 콜드 클러스터의 캐시 적중률(cache hit rate) 빠르게 상승
  - 정상 운영 상태로 전환되는 시간 단축 가능

5. 지역을 넘어: 일관성 유지 (Across Regions: Consistency)

Across Regions: Consistency는 페이스북이 **여러 지역 데이터 센터(across multiple regional data centers)**에 분산된 데이터 일관성(consistency)을 유지하는 방법을 설명합니다. 페이스북은 글로벌 사용자에게 일관된 데이터를 제공하기 위해 다양한 기술을 도입하여 데이터가 여러 지역 간에 **동기화(synchronization)**될 수 있도록 설계했습니다. 이러한 설계는 사용자 경험을 개선하면서도, 네트워크 지연을 최소화하고 데이터 일관성을 보장하는 데 목적이 있습니다.

1. 원격 복제 (Remote Replication)

- 여러 지역에 걸친 데이터 동일 복제(replication across regions)
- 지역 간 데이터 일관성(consistency) 유지
- 사용자가 작성한 데이터의 여러 지역 복제 통한 글로벌 최신 데이터 조회
- 양방향 복제(bi-directional replication) 통한 각 지역에서 데이터 업데이트 가능

2. 읽기와 쓰기 요청 처리 (Read and Write Request Handling)

- 쓰기 요청(write requests)의 데이터 초기 생성 지역 데이터베이스 기록
- 각 지역 Memcache로의 데이터 복제 통한 조회 성능 향상
- 읽기 요청(read requests)의 사용자 위치 지역 캐시 처리로 읽기 지연(read latency) 감소
- 분산된 읽기/쓰기 처리 통한 사용자 위치 무관 빠른 응답 제공
- 데이터베이스 부하 감소

3. 원격 마커 (Remote Marker)

- 글로벌 데이터 일관성(global consistency) 유지 위한 Remote Marker 개념 도입
- 지역 데이터베이스 간 동기화 상태 추적
- 특정 지역 데이터 변경 시 다른 지역으로 전송
- 각 지역 데이터베이스 최신 상태 업데이트
- 특정 시점 데이터 전파 지역 확인 통한 일관성 지연(consistency lag) 감소
- 데이터 최신 상태 파악 가능

4. 일관성 지연 관리 (Managing Consistency Lag)

- 일관성 지연(consistency lag): 데이터 변경의 지역 간 전파 시간 차이
- 데이터베이스 이벤트 스트림(event stream) 통한 지연 최소화
- 각 지역 데이터 센터에 변경 사항 신속 전파
- 네트워크 상태 및 물리적 거리로 인한 지연 발생 가능성
- Remote Marker 활용한 지연 효과적 관리

5. 최종 일관성 (Eventual Consistency)

- 최종 일관성(eventual consistency) 모델 사용
- 일정 시간 후 모든 지역에서 동일 데이터 상태 유지 보장
- 즉각적 지역 반영 어려움 허용, 최종적 일관성 보장
- 데이터 전파 전 임시 다른 지역에서 이전 데이터 조회 가능성 허용

6. 일관성 유지 방식의 장점 (Benefits of Consistency Maintenance)

- 글로벌 사용자 경험 개선: 사용자 위치 무관한 빠르고 일관된 데이터 조회 가능

- 네트워크 지연 감소: 지역 간 데이터 전파 지연 상황에서 사용자 요청 지역 최신 데이터 접근
- 서버 부하 분산: 지역별 자체 요청 처리로 특정 지역 트래픽 몰림 방지

7. 장애 복구와 데이터 보존 (Disaster Recovery and Data Preservation)

- 재난 복구(disaster recovery) 시스템 구축 통한 특정 지역 장애 시 데이터 복구 가능
- 동일 데이터의 다른 지역 데이터 센터 복제 통한 데이터 손실 없는 서비스 지속
- 시스템 복원력(resilience) 향상 및 데이터 무결성(data integrity) 유지

6. 단일 서버 개선 (Single Server Improvements)

단일 서버 개선 (Single Server Improvements)는 페이스북이 Memcache 시스템에서 **단일 서버 (single server)**의 성능을 최적화하고 **지연 시간(latency)**과 **부하(load)**를 줄이기 위해 적용한 다양한 기법을 설명합니다. 이 과정에서 페이스북은 **잠금(locking)**과 **데이터 전송 방식(data transmission)**을 개선하여 단일 서버의 성능을 극대화하고, 대규모 트래픽을 효율적으로 처리할 수 있도록 했습니다.

1. 세밀한 잠금 (Fine-Grained Locking)

- 데이터 일관성(data consistency) 보장을 위한 잠금 필요성
- 기존 단일 잠금 방식 대신 개별 데이터 항목별 세밀한 잠금(fine-grained locking) 적용
- 작은 단위로 데이터 잠금을 설정하여 동시성(concurrency) 향상
- 여러 요청의 동시 접근 가능성 증가로 병목(bottleneck) 현상 완화
- 서버의 전반적인 성능(performance) 개선

2. UDP 사용 통한 데이터 조회 속도 향상 (UDP for Fast Data Retrieval)

- 데이터 조회(get requests) 요청에 UDP(User Datagram Protocol) 사용
- 연결 설정(connection setup) 없는 UDP를 통한 데이터 전송으로 네트워크 왕복 시간(round-trip time) 감소
- 데이터 조회 요청 시 빠른 응답 가능
- 패킷 손실 시 재전송이 없는 UDP 특성 활용
- 지연 시간(latency) 최소화로 사용자 응답성 향상

3. TCP 사용 통한 데이터 무결성 보장 (TCP for Data Integrity)

- 데이터 수정(set requests) 요청에 TCP(Transmission Control Protocol) 사용
- TCP의 신뢰성(reliability) 높은 특성을 통해 데이터 전송 중 손실 방지
- 데이터 무결성(data integrity) 보장
- 중요한 데이터 수정 작업에 안전한 전송 방식 확보

4. 데이터 압축 (Data Compression)

- 단일 서버의 효율적 데이터 저장과 전송을 위한 압축(compression) 기법 사용
- 메모리 사용량(memory usage) 감소를 위한 데이터 압축 저장
- 더 많은 데이터 캐시에 저장 가능
- 압축된 데이터 전송 시 네트워크 대역폭(bandwidth) 절감
- 메모리 효율(memory efficiency) 향상 및 데이터 조회 시 응답 시간 단축

5. 비동기 입출력 (Asynchronous I/O)

- I/O 작업을 비동기 방식(asynchronous I/O)으로 처리
- 요청 완료 대기 없이 다음 작업 진행으로 동시성(concurrency) 증가
- 네트워크 대기 시간 감소를 통한 서버 처리 속도 향상
- 서버의 지연 시간(latency) 줄여 효율성 극대화
- 고성능 요구에 맞춘 빠른 데이터 처리 가능

6. 핫 데이터 캐싱 (Hot Data Caching)

- 자주 요청되는 데이터(hot data)의 우선적 캐싱
- 반복적 접근 데이터의 빠른 조회 제공
- LRU(Least Recently Used) 알고리즘 활용하여 자주 사용되는 데이터 캐시에 유지
- 메모리 효율성(memory efficiency) 높이고 덜 사용되는 데이터 제거
- 사용자 요청에 대한 지연 시간 감소

7. 메모리 할당 최적화 (Memory Allocation Optimization)

- 고정 크기 메모리 블록(fixed-size memory block) 할당 방식 도입
- 메모리 조각화(memory fragmentation) 문제 감소
- 동적 할당 대신 사전 정의된 메모리 블록 할당으로 데이터 접근 속도 증가
  - 데이터의 크기에 따라 자주 사용하는 크기의 메모리 블록을 미리 정의하고, 필요한 데이터에 맞춰 적절한 크기의 블록을 할당합니다.
- 메모리 할당 및 해제 시 오버헤드(overhead) 최소화
- 단일 서버의 성능(performance) 극대화

7. Memcache 작업 부하 (Memcache Workload)

1. 요청 빈도와 유형 (Request Frequency and Type)

- Memcache에서 대부분의 요청은 **읽기 요청(get requests)**으로 구성
- 읽기 요청의 비율이 쓰기 요청(set requests)에 비해 압도적으로 높음
- 읽기 요청: 데이터베이스 접근을 줄이고 빠른 응답 제공을 위한 주된 작업
- 쓰기 요청: 새로운 데이터를 추가하거나 기존 데이터를 갱신할 때 발생

- 읽기-쓰기 요청의 불균형을 고려한 최적화 필요성

2. 데이터 크기와 분포 (Data Size and Distribution)

- 다양한 크기의 데이터가 Memcache에 저장되며, 데이터 크기는 용도에 따라 차이가 큼
- 작은 데이터: 사용자 세션 정보, 사용자 상태와 같은 자주 조회되는 데이터
- 큰 데이터: 캐싱된 검색 결과, 피드와 같은 비정기적 데이터
- 주로 작은 데이터가 많으며, 이로 인해 메모리 할당 효율성 최적화가 중요

3. 트래픽 패턴 (Traffic Pattern)

- 트래픽 패턴의 높은 변동성, 특히 특정 시간대에 집중되는 트래픽 증가
- 이벤트나 특정 시기에 발생하는 트래픽 급증 문제
- 트래픽 증가 시 Memcache의 캐시 적중률(cache hit rate) 관리 필요
- 예상치 못한 트래픽 급증 대비한 캐싱 전략 필요

4. 캐시 적중률 (Cache Hit Rate)

- Memcache에서의 데이터 조회 시 캐시 히트(hit)와 미스(miss) 발생
- 캐시 적중률 높일수록 데이터베이스 부하 감소 및 응답 속도 향상
- 캐시 미스를 줄이기 위한 효율적인 캐시 관리 필요

5. 데이터 만료와 갱신 빈도 (Data Expiration and Update Frequency)

- Memcache는 일정 시간이 지나면 데이터를 자동으로 만료(expire) 시킴
- 자주 갱신되거나 실시간 데이터의 경우 주기적인 업데이트 필요
- 만료 정책(expiration policy)과 갱신 빈도에 따라 캐시 적중률과 효율성에 영향
- 최신 정보 유지와 메모리 효율성을 위해 최적의 만료 정책 필요
    - 데이터의 갱신 빈도에 따라 만료 시간을 설정하여, 자주 변경되는 데이터는 더 짧은 만료 시간, 안정적인 데이터는 긴 만료 시간 부여.
    - **정적 데이터**와 **동적 데이터**에 서로 다른 만료 시간을 적용하여 메모리를 효율적으로 사용.
    - 특정 시간대에 높은 트래픽이 예상되는 경우, 만료 정책을 조정해 트래픽이 적은 시간대에 갱신을 수행하도록 함.
    - 사용자 요청에 따라 필요한 경우만 캐시를 무효화하는 **지능형 무효화 정책**을 도입하여, 불필요한 메모리 소모를 줄이고 캐시 적중률을 높임.
    - **자주 조회되는 데이터는 무효화를 지연**하여 유지하고, 덜 자주 조회되는 데이터는 무효화를 더 자주 실행.
    - 데이터 변경 후 일정 시간 동안 무효화를 지연하여, 여러 번의 변경을 묶어서 처리.

6. 데이터 일관성 문제 (Data Consistency Issues)

- 여러 클라이언트가 Memcache에서 동일 데이터에 접근할 때 일관성(consistency) 문제 발생 가능
- 데이터베이스에서 데이터가 변경되었으나, Memcache에 이전 데이터가 남아 있는 경우 발생
- 스테일 데이터(stale data) 문제를 방지하기 위한 캐시 무효화(invalidation) 전략 필요
- 특정 이벤트 발생 시 데이터 갱신 또는 무효화 통한 데이터 일관성 유지

7. 작업 부하 관리 전략 (Workload Management Strategy)

- 자주 요청되는 데이터와 그렇지 않은 데이터를 구분하여 메모리 사용 최적화
- 자주 요청되는 데이터는 Memcache에 오래 유지, 덜 요청되는 데이터는 빠르게 만료
- Memcache 부하를 줄이기 위해 핫 데이터(hot data)와 콜드 데이터(cold data) 구분
- 주기적인 모니터링과 분석을 통해 적절한 캐시 용량(capacity) 설정 필요

8. 네트워크 및 I/O 대역폭 관리 (Network and I/O Bandwidth Management)

- Memcache가 처리하는 많은 양의 읽기 요청으로 네트워크 부하 증가 가능
- 네트워크 대역폭을 최적화하기 위한 데이터 압축(compression) 및 일괄 처리(batch processing) 적용
- I/O 대역폭을 효율적으로 관리하여 서버 병목(bottleneck) 방지
- 데이터 전송과 네트워크 부하를 줄이기 위한 비동기 I/O 방식 활용

9. 서버 자원 효율화 (Server Resource Optimization)

- CPU, 메모리 사용 최적화를 통해 서버의 효율성 극대화
- 메모리 할당 최적화와 데이터 압축을 통한 메모리 효율 향상
- I/O 작업을 비동기적으로 처리하여 CPU 자원 낭비 감소
- 서버 자원 효율화를 통해 Memcache의 성능 및 처리 용량 향상

8. 관련 연구 (Related Work)

- 이와 유사한 시스템을 구축한 다른 주요 기술 기업의 사례와 함께, Memcache가 적용된 주요 연구 및 관련 연구들을 소개합니다.

9. 결론 (Conclusion)

이 논문에서는 Facebook의 증가하는 수요를 충족하기 위해 memcached 기반 아키텍처를 확장하는 방법을 보여줍니다. 논의된 많은 상충은 근본적이지 않지만 엔지니어링 리소스의 균형을 맞추는 현실에 뿌리를 두고 있으며
지속적인 제품 개발 하에 라이브 시스템을 진화시킵니다. 시스템을 구축, 유지 관리 및 진화시키는 동안 다음과 같은 교훈을 얻었습니다.

(1) Separating cache and persistent storage systems allows us to independently scale them.
(2) Features that improve monitoring, debugging and operational efficiency are as important as performance.
(3) Managing stateful components is operationally more complex than stateless ones. As a result keeping logic in a stateless client helps iterate on features and minimize disruption.
(4) The system must support gradual rollout and rollback of new features even if it leads to temporary heterogeneity of feature sets.
(5) Simplicity is vital.

(1) 캐시와 영구 저장 시스템을 분리하면 독립적으로 확장할 수 있습니다.
(2) 모니터링, 디버깅 및 운영 효율성을 개선하는 기능은 성능만큼 중요합니다.
(3) 상태 저장 구성 요소를 관리하는 것은 상태 없는 구성 요소보다 운영상 더 복잡합니다. 그 결과 상태 없는 클라이언트에 논리를 유지하면 기능을 반복하고 중단을 최소화하는 데 도움이 됩니다.
(4) 시스템은 기능 세트의 일시적인 이질성으로 이어지더라도 새로운 기능의 점진적인 출시 및 롤백을 지원해야 합니다.
(5) 단순성이 중요합니다.

같이 보면 좋은 글

- 캐시 문제 해결 가이드 - DB 과부하 방지 실전 팁