백엔드 & 인프라 스터디 1 회차 문제

2025-09-19

작성자 : 김호중

데이터베이스 설계: 정규화와 비정규화 사이의 트레이드오프

상황

빠르게 성장하는 이커머스 플랫폼의 백엔드를 개발하고 있습니다. 초기 데이터베이스는 **제 3 정규형(3NF)**을 철저히 준수하여 설계되었고, 데이터의 중복이 거의 없어 쓰기(Write) 성능과 데이터 정합성 측면에서 매우 만족스러웠습니다.

하지만 서비스의 핵심 페이지인 상품 목록 페이지의 로딩 속도가 사용자가 늘어남에 따라 점점 느려지는 문제가 발생했습니다. 이 페이지는 상품명(products), 카테고리명(categories), 대표 이미지 URL(product_images), 평균 별점(reviews) 정보를 한 번에 보여주어야 합니다. 이 정보를 가져오기 위해 4 개의 테이블을 JOIN 해야 했고, 특히 수백만 건의 데이터가 쌓인 reviews 테이블에 AVG()와 COUNT() 같은 집계 함수를 실행하는 것이 극심한 성능 병목을 유발하는 원인으로 지목되었습니다.

문제

1. 병목의 근본 원인 분석: '읽기' 연산이 많은 위 시나리오에서, 교과서적으로 '잘 설계된' 정규화된 스키마가 오히려 어떻게 성능 저하의 주범이 되는지 조인(JOIN)의 비용과 실시간 집계의 부하 관점에서 구체적으로 설명하시오.

2. 비정규화전략 제시: 이 문제를 해결하기 위해 products 테이블의 스키마를 어떻게 변경하시겠습니까? products 테이블에 어떤 컬럼들을 추가하여 JOIN 과 실시간 집계를 최소화할 수 있을지, 구체적인 비정규화 스키마 변경안을 제시하시오.

3. 새로운 트레이드오프 분석: 제시한 비정규화 전략은 읽기 성능을 극적으로 향상시키지만, '쓰기' 연산의 복잡성과 데이터 불일치 위험이라는 새로운 트레이드오프를 낳습니다. 만약 사용자가 리뷰를 작성하거나, 관리자가 카테고리 이름을 수정할 경우, products 테이블에 중복 저장된 데이터의 정합성을 어떻게 보장할 수 있을지 최소 2 가지 이상의 전략을 비교하여 설명하시오

4. 아키텍처적 접근 :비정규화가 기존 OLTP 데이터베이스를 너무 복잡하게 만든다고 판단될 경우, 어떤 대안을 고려할 수 있을까요? 읽기(Query)용 모델과 쓰기용 모델을 분리하는 CQRS(Command Query Responsibility Segregation) 패턴의 관점에서, 분석/조회 전용의 읽기 전용 저장소를 별도로 구축하는 접근법의 장단점을 비정규화 전략과 비교하여 논하시오.

데이터베이스 설계 심층 탐구: 초거대 물류 플랫폼의 성능 병목 해결

상황

당신은 B2B 스마트 물류 플랫폼 'Logis-Flow'의 백엔드 수석 엔지니어로 합류했습니다. Logis-Flow 는 여러 고객사(화주, 운송사)의 상품 입고부터 창고 관리, 최종 목적지 배송까지 전 과정을 추적하고 관리하는 SaaS 솔루션입니다. 서비스 초기, 데이터베이스는 데이터 정합성과 확장성을 최우선으로 고려하여 제 3 정규형을 철저히 준수하여 설계되었습니다.

서비스가 폭발적으로 성장하며 하루에 수백만 건의 물류 이동이 발생하자, 시스템의 핵심 기능인 '실시간 화물 추적 대시보드'에서 심각한 성능 문제가 발생하기 시작했습니다. 이 대시보드는 고객사가 특정 화물(Shipment)의 현재 상태를 한눈에 파악하는 가장 중요한 화면입니다. 이 화면을 렌더링하기 위해 시스템은 다음과 같은 정보를 조합해야 합니다.

화물의 고유 ID, 출발 창고 이름, 도착 창고 주소

화물의 현재 상태(예: '집화 완료', '터미널 간 이동 중', '배송 중')와 최종 업데이트 시각

화물에 포함된 모든 상품의 이름과 수량

해당 화물이 생성된 이후 발생한 모든 상태 변경 이력(타임라인)

시스템의 데이터베이스 스키마는 다음과 같이 정규화되어 있습니다.

companies: 고객사 정보 (약 1 만 개)

warehouses: 창고 정보, 주소 포함 (약 5 만 개)

products: 상품 마스터 정보 (약 1 천만 개)

shipments: 화물 정보. origin_warehouse_id, destination_warehouse_id 를 외래 키로 가짐 (누적 5 억 건)

shipment_items: shipments 와 products 를 잇는 다대다 관계 테이블 (누적 50 억 건)

shipment_updates: 모든 화물의 상태 변경 기록이 append-only 방식으로 저장되는 로그 테이블. shipment_id, status_code, timestamp, notes 등의 컬럼을 가짐 (누적 500 억 건)

대시보드 로딩 시, 단일 화물 정보를 조회하는 쿼리는 shipments 테이블을 시작으로 warehouses 테이블을 두 번 조인하고, shipment_items 와 products 를 조인합니다. 가장 치명적인 부분은 화물의 '현재 상태'를 가져오기 위해 500 억 건의 shipment_updates 테이블에서 특정 shipment_id 에 해당하는 기록 중 timestamp 가 가장 최신인 1 건을 찾아야 한다는 점입니다. 또한, 상태 변경 이력 타임라인을 보여주기 위해 해당 shipment_id 의 모든 기록을 timestamp 순으로 정렬해야 합니다.

이 복합적인 쿼리는 피크 시간대에 10 초 이상 소요되어 고객의 불만을 야기하고 있으며, 데이터베이스의 읽기 전용 복제본(Read Replica) 마저 CPU 사용률이 100%에 달하는 상황을 만들고 있습니다.

문제

첫째, 현재의 정규화된 스키마가 왜 이토록 심각한 읽기 성능 저하를 유발하는지 데이터베이스의 내부 동작 원리와 연관 지어 분석하시오. 특히 거대한 로그성 테이블인 shipment_updates 에서 '가장 최신 상태' 하나를 조회하는 작업이 왜 본질적으로 비효율적인 연산인지 인덱스 전략과 데이터 스캔 범위의 관점에서 설명해야 합니다.

둘째, 이 문제를 해결하기 위한 첫 단계로써 비정규화 전략을 구체적으로 제시하시오. shipments 테이블의 구조를 어떻게 변경하여 대시보드 로딩에 필요한 조인 연산과 실시간 집계 작업을 최소화할 수 있을지 설명해야 합니다. 예를 들어, current_status, last_updated_at, origin_warehouse_name 과 같은 컬럼을 추가하는 방안을 포함하여 논리적인 스키마 변경안을 제안하시오.

셋째, 당신이 제안한 비정규화 전략은 쓰기 경로의 복잡성을 증가시키고 데이터 정합성을 해칠 새로운 위험을 내포합니다. 새로운 상태 변경이 shipment_updates 테이블에 삽입될 때마다, shipments 테이블에 비정규화된 current_status 와 같은 컬럼의 값을 어떻게 일관성 있게 유지할 것인지 세 가지 서로 다른 아키텍처적 접근법, 즉 동기적 애플리케이션 트랜잭션, 데이터베이스 트리거, 그리고 메시지 큐를 활용한 비동기적 최종 일관성 모델의 장단점을 각각 비교하여 설명하시오.

넷째, 대시보드의 '상태 변경 이력 타임라인' 기능은 여전히 shipment_updates 테이블의 많은 데이터를 읽어야 하는 부담을 안고 있습니다. 이 특정 조회 패턴을 최적화하기 위해, 기존 관계형 데이터베이스의 기능을 심화 활용하는 방안과 외부 시스템을 도입하는 방안을 각각 제시하고 비교하시오. PostgreSQL 의 테이블 파티셔닝 기능을 활용하는 방안과, 이력 데이터를 로그 분석이나 시계열 데이터 처리에 최적화된 외부 데이터 저장소, 예를 들어 Elasticsearch 나 DynamoDB 로 이관하는 방안의 기술적 트레이드오프를 논하시오.

다섯째, shipment_updates 테이블은 시간이 지남에 따라 무한히 커질 것이 자명합니다. 서비스의 법적 요구사항과 운영 비용을 고려하여, 데이터 생명주기 관리 전략을 수립하시오. 예를 들어, 배송이 완료된 지 2 년이 지난 화물의 상태 변경 이력은 어떻게 처리할 것인지, 운영 데이터베이스에서 저비용의 장기 보관 스토리지, 예를 들어 AWS S3 Glacier 로 데이터를 안전하게 이전하고 필요시 다시 조회할 수 있는 아카이빙 프로세스를 구체적으로 설계하시오.

리눅스 시스템 심층 탐구: 대규모 라이브 스트리밍 플랫폼의 병목 분석과 튜닝

상황

당신은 전 세계 수백만 명의 사용자를 보유한 라이브 스트리밍 플랫폼 **'Core-Stream'**의 SRE(Site Reliability Engineer)로 근무하고 있습니다. Core-Stream 의 핵심 백엔드 시스템은 방송인이 송출하는 고화질 원본 영상(RTMP)을 입력받아, 실시간으로 1080p, 720p, 480p 등 여러 화질로 트랜스코딩하고, 이를 HLS 또는 DASH 포맷의 작은 비디오 조각(Segment)으로 만들어 수만 명의 시청자에게 동시 전송하는 역할을 합니다.

이 시스템은 고성능 C++ 애플리케이션으로 구현되어 있으며, 다수의 Ubuntu 22.04 LTS 서버 클러스터 위에서 동작합니다. 평상시 수천 명의 시청자가 접속하는 환경에서는 아무런 문제가 없지만, e 스포츠 결승전이나 인기 스트리머의 특별 방송과 같이 서버당 동시 접속자가 5 만 명을 초과하는 대규모 이벤트가 발생할 때마다, 특정 서버들에서 영상 버퍼링이 심해지고 간헐적인 끊김 현상이 발생한다는 보고가 급증합니다.

기술팀의 초기 분석 결과는 더욱 혼란스러웠습니다. 문제의 서버들은 CPU 의 사용자 시간(us)과 시스템 시간(sy) 점유율이 합계 50% 미만으로 여유가 있었고, 메모리 사용량이나 네트워크 인터페이스의 대역폭(Bandwidth) 역시 한계치에 도달하지 않았습니다. 하지만 top 이나 htop 으로 CPU 상태를 상세히 들여다보면, **'si'**로 표시되는 소프트웨어 인터럽트(softirq) 처리 시간이 비정상적으로 높게 치솟는 현상이 관찰되었습니다. 또한, netstat 으로 네트워크 통계를 확인하면 TCP 소켓의 수신 큐가 가득 차 패킷이 버려지는 'listen queue overflows' 카운터가 급증하고 있었고, 애플리케이션 로그에는 간헐적으로 'Too many open files' 오류가 기록되기도 했습니다.

문제

첫째, 시스템의 가장 명백한 이상 징후인 높은 소프트웨어 인터럽트(softirq) CPU 사용률의 근본 원인을 설명하시오. 리눅스 커널의 네트워크 스택에서 소프트웨어 인터럽트가 어떤 역할을 하는지 설명하고, 이 현상이 왜 '애플리케이션이 바쁜 것'과는 다른 차원의 문제이며, 오히려 커널 수준에서 네트워크 패킷을 처리하는 데 모든 자원을 소모하고 있음을 시사하는지 그 이유를 논리적으로 분석해야 합니다. 또한, cat /proc/softirqs 나 perf 와 같은 도구를 사용하여 수많은 소프트웨어 인터럽트 유형 중 구체적으로 어떤 항목(예: NET_RX)이 병목을 일으키고 있는지 특정하는 과정을 설명하시오.

둘째, 대규모 동시 접속으로 인한 TCP listen queue overflow 와 패킷 손실 문제를 해결하기 위한 커널 네트워크 파라미터 튜닝 전략을 제시하시오. sysctl 명령을 통해 변경할 수 있는 리눅스 커널의 핵심 파라미터 중 최소 세 가지 이상을 선택하고, 각 파라미터(예: net.core.somaxconn, net.core.netdev_max_backlog, net.ipv4.tcp_max_syn_backlog)가 구체적으로 무엇을 제어하며, 이 값을 상향 조정하는 것이 현재 문제 상황을 어떻게 개선할 수 있는지 상세히 설명해야 합니다.

셋째, 'Too many open files' 오류는 단순한 문제가 아님을 설명하시오. 리눅스에서 파일 디스크립터(File Descriptor) 제한이 프로세스별 제한과 시스템 전역 제한으로 나뉘어 관리된다는 점을 지적하고, 이 두 가지 제한의 차이점과 각각을 확인하고 영구적으로 변경하는 방법(예: /etc/security/limits.conf, /etc/sysctl.conf)을 설명해야 합니다. 특히, 수만 개의 동시 TCP 연결을 유지해야 하는 Core-Stream 의 애플리케이션이 왜 기본 설정값을 쉽게 초과할 수밖에 없는지 그 구조적 이유를 분석하시오.

넷째, 문제의 근원은 애플리케이션이 수만 개의 네트워크 연결을 처리하는 방식 자체에 있을 수 있습니다. 전통적인 select 나 poll 과 같은 동기적 I/O 다중화(Multiplexing) 모델이 왜 이러한 초고성능 환경에 부적합한지 그 확장성의 한계를 설명하시오. 이어서, 현대적인 리눅스 애플리케이션이 사용하는 epoll 시스템 콜이 어떻게 커널의 도움을 받아 이 문제를 해결하는지, epoll 의 이벤트 기반(event-driven) 및 엣지 트리거(edge-triggered) 동작 방식이 어떻게 수만 개의 유휴 연결을 효율적으로 무시하고 활성 연결에만 집중하여 CPU 낭비를 막고 응답성을 높이는지 그 내부 동작 원리를 심층적으로 설명하시오.

REST API 심층 탐구: 실시간 주문 처리 시스템의 상태 전이와 동시성 제어

상황

당신은 국내 최대 음식 배달 플랫폼 **'QuickEats'**의 주문 처리 마이크로서비스를 담당하는 백엔드 엔지니어입니다. 이 서비스는 고객, 레스토랑, 라이더라는 세 명의 행위자(Actor) 사이의 복잡한 상호작용을 관장하며, 주문의 전체 생명주기를 관리하는 REST API 를 외부에 제공합니다.

서비스 V1 API 는 초기에 빠른 개발을 위해 주문(Order)이라는 단일 리소스를 중심으로 설계되었습니다. 하지만 일일 주문량이 수백만 건을 넘어서면서, V1 API 의 설계적 한계가 명확한 기술 부채로 돌아오고 있습니다.

주문(Order)의 핵심 생명주기:

결제 대기중(pending_payment) → 결제 실패(payment_failed) 또는 주문 접수 대기중(pending_acceptance) → 주문 거절됨(rejected) 또는 조리중(preparing) → 픽업 대기중(ready_for_pickup) → 배달중(in_transit) → 배달 완료(delivered)

현재 V1 API 의 문제점:

모호한 상태 전이 API: 주문의 모든 상태 변경이 PUT /api/v1/orders/{order_id}라는 단일 엔드포인트에 { "status": "new_status" }와 같은 요청 본문을 보내는 방식으로 처리됩니다. 이로 인해 서버의 비즈니스 로직은 거대한 if-else 분기문으로 가득 차 있으며, 예를 들어 고객이 주문을 '취소'하는 행위와 라이더가 '픽업 완료'를 알리는 행위가 동일한 API 엔드포인트를 사용해 논리적 구분이 모호합니다.

치명적인 동시성 문제(Race Condition): 고객이 주문을 취소하기 위해 {"status": "cancelled"} 요청을 보내는 순간, 거의 동시에 레스토랑 점주가 주문을 접수하기 위해 {"status": "preparing"} 요청을 보내는 경우가 발생합니다. 이 두 요청이 거의 동시에 처리될 경우, 데이터베이스의 최종 상태를 예측할 수 없어 이미 취소된 주문의 음식이 조리되는 등의 심각한 데이터 불일치 문제가 발생합니다.

멱등성(Idempotency) 부재: 고객의 결제 요청이 네트워크 문제로 타임아웃되었을 때, 클라이언트 앱은 안전하게 재시도를 합니다. 하지만 V1 의 POST /api/v1/orders/{order_id}/pay 엔드포인트는 멱등성이 보장되지 않아, 재시도된 요청이 중복 결제를 유발하는 심각한 금융 사고를 일으킨 전적이 있습니다.

데이터 로딩 비효율(N+1 문제): 고객 앱의 '주문 현황' 목록 화면은 여러 주문의 상태와 함께 각 주문의 레스토랑 이름, 라이더의 현재 위치를 보여줘야 합니다. GET /api/v1/orders API 는 restaurant_id 와 rider_id 만 반환하여, 클라이언트는 목록에 있는 N 개의 주문에 대해 N 번의 추가적인 API 호출(레스토랑 정보 조회, 라이더 위치 조회)을 해야만 전체 화면을 그릴 수 있습니다.

이러한 문제들을 해결하기 위해, 비즈니스 규칙을 명확히 표현하고, 동시성을 안전하게 처리하며, 클라이언트와 효율적으로 소통하는 V2 API 를 설계해야 합니다.

문제

첫째, V1 API 의 모호한 상태 전이 문제를 해결하기 위해, 주문의 생명주기를 RESTful 하게 표현하는 API 를 새롭게 설계하시오. PUT 메서드에 의존하는 대신, 주문의 상태를 변경시키는 각 '행위' 자체를 리소스로 간주하는 '행위 기반 리소스(Action-oriented Resource)' 접근법을 사용하여 API 엔드포인트를 설계해야 합니다. 예를 들어, 레스토랑의 '주문 접수' 행위나 고객의 '주문 취소' 행위를 위한 구체적인 엔드포인트(HTTP 메서드와 URI 경로)를 제시하고, 이러한 설계가 왜 V1 에 비해 시스템의 비즈니스 규칙을 더 명확하게 만들고 확장성을 높이는지 설명하시오.

둘째, 결제 재시도 시 중복 처리 문제를 방지하기 위해 멱등성을 어떻게 보장할 것인지 설명하시오. 클라이언트가 네트워크 오류 발생 시 안전하게 재시도할 수 있도록, POST /api/v2/orders/{order_id}/payment 와 같은 엔드포인트를 어떻게 수정해야 하는지 Idempotency-Key 헤더를 활용하는 표준적인 방법을 통해 그 요청-응답 흐름과 서버 측의 처리 로직을 상세히 기술해야 합니다.

셋째, 고객의 '주문 취소'와 레스토랑의 '주문 접수'가 충돌하는 것과 같은 동시성 문제를 해결하기 위한 낙관적 락(Optimistic Locking) 전략을 API 수준에서 어떻게 구현할 수 있을지 설명하시오. HTTP 의 ETag 와 If-Match 조건부 요청 헤더를 사용하여, 클라이언트가 항상 리소스의 최신 버전을 기반으로 상태 변경을 시도하도록 보장하는 전체적인 과정을 설명해야 합니다. 만약 클라이언트가 오래된 버전의 정보를 기반으로 요청했을 때, 서버가 412 Precondition Failed 응답을 반환하여 충돌을 방지하는 시나리오를 포함하여 논하시오.

넷째, 클라이언트 앱의 N+1 조회 문제를 해결하기 위한 두 가지 서로 다른 데이터 제공 전략을 제시하고 비교하시오. 첫 번째로, RESTful API 의 유연성을 유지하면서 관련 리소스를 함께 포함하여 전달하는 '컴파운드 도큐먼트(Compound Document)' 또는 '사이드로딩(Side-loading)' 방식을 ?include=restaurant,rider 와 같은 쿼리 파라미터를 통해 어떻게 구현할 수 있는지 설명해야 합니다. 두 번째로, 대안적인 아키텍처로서 GraphQL 을 도입한다면 이 문제가 어떻게 더 근본적으로 해결될 수 있는지, 클라이언트가 필요한 데이터의 구조를 직접 정의하는 GraphQL 의 특징을 중심으로 두 방식의 장단점을 기술하시오.

다섯째, 기존 V1 API 를 사용하는 구버전 앱 사용자들을 중단 없이 지원하면서, 새로운 V2 API 를 안전하게 출시하기 위한 API 버전 관리 전략을 제시하시오. 가장 널리 사용되는 URI 경로 기반 버전 관리(예: /api/v2/...) 방식의 장점을 설명하고, 이를 Django 와 같은 웹 프레임워크에서 어떻게 구현할 수 있는지 기술해야 합니다.