Part 2 분산데이터

태그	
생성일	@2021년 8월 29일 오후 2:53

합의 알고리즘
분산 트랜잭션, 합의 알고리즘쪽 한번 보자 ..

단일데이터 장비가 아닌 여러 데이터 장비를 사용하는 분산 데이터 시스템은 왜 사용할까?

- 확장성 : 여러 장비로 부하를 분배한다
- 내결함성/고가용성 : 하나의 장비가 중지되어도 다른 장비로 서비스한다
- 지연 시간 : 여러 장비를 여러 지역에 놓고 가장 가까운곳에서 응답을 받는다

공유 메모리 아키텍처와 비공유 아키텍처

- 공유 메모리 아키텍처(shared-memory architecture) → 거대한 단일 장비 → 수직확장, 용량확장
- 비공유 아키텍처(shared-nothing) → 다수의 작은 장비 → 수평확장, 규모 확장

분산 데이터에는 두가지 종류가 있다

- 복제
- 파티셔닝

데이터가 분산되어있으면 트랜잭션 처리는 어떻게할까? 분산 데이터 트랜잭션에 대해서도 알아보자

05. 복제

복제는 여러 장비에 같은 데이터를 복제해놓는다. 복제가 필요한 이유는 다음과 같다.

- 지리적으로 가까운 곳에 데이터를 복제해놓고 서빙하여 지연시간을 줄인다
- 시스템 일부에 장애가 발생해도 지속적으로 서빙가능하다
- 읽기 질의를 제공하는 장비의 수를 확장해 읽기 처리량을 늘릴 수 있따

복제 알고리즘

리더 구분

- 단일 리더(single-leader)
- 다중 리더(multi-leader)
- 리더 없는(leaderless)

동기 구분

- 동기적 복제
- 비동기적 복제

일관성

- 최종적 일관성 (eventually consistency)
- 약한 일관성 (weak consistency)
- 강한 일관성 (strong consistency)

- 쓰기 읽기(read-yourt-writes) → 스냅샷 격리, 트랜잭션
- 단조 읽기(monotonic read) → 항상 무조건 일관적인 데이터 읽기

리더와 팔로워

- Master - Slave, Master - Read Replica, Active - Passive, Primary - Secondary 랑 같은 말
- 리더가 로컬 저장소에 새로운 데이터를 기록하거나 변경할 때마다 복제 로그(Replication Log)나 변경 스트림(Change Stream)을 통해 팔로워에게 전송하여 복사본을 갱신한다
- 쓰기는 리더에게만 허용된다

동기식 복제와 비동기식 복제

- 동기식 복제(synchronous replication)는 리더와 팔로워가 모두 쓰기가 되었을 때 쓰기 요청을 완료처리한다. 하지만 이는 여러 노드가 모두 정상이어야하고 쓰기 동작이 매우 느려질 수 있다

- 비동기식 복제(asynchronous replication)는 팔로워가 낮은 결합으로 복제되어서 위의 문제가 해결되지만. 복제지연문제가 있다.
- 보통 리더 하나 팔로워 하나를 동기식 복제하고 나머지는 비동기식 복제처리를 하는데 이러한 방식을 반동기식(semi-synchronous replication)이라고 한다.
- 리더를 통한 비동기식 복제는 완전 비동기식 복제로 리더가 죽을경우 쓰기가 모두 유실될 수 있어 위험할 수 있지만 리더 노드의 부하를 줄일 수 있다.

새로운 팔로워를 추가하기

- 리더 노드는 스냅샷을 남긴다
- 스냅샷으로부터 팔로워를 생성한다. 그리고 이 스냅샷의 마지막 로그 좌표로부터 리더 노드에서 발생한 변경분을 모두 팔로워 노드에 맞춘다
- 그 이후 팔로워 노드는 리더 노드의 변경 스트림을 따라간다

노드에 장애가 났다면

팔로워 노드에 장애가 났을 때

- 팔로워 노드의 마지막 트랜잭션 혹은 로그를 기억하고 있다가 노드가 정상화되면 리더 노드의 그 이후 변경 스트림을 모두 받아서 데이터를 변경한다

리더에 장애가 났을 때

- 새로운 리더를 선출하거나 리더를 다시 정상화 시킨다

리더 장애 failover 과정

- 리더가 장애인지 판단
- 새로운 리더를 선택
- 새로운 리더 사용을 위해 시스템을 재설정 (팔로워 노드가 새로운 리더노드로부터 변경 스트림을 받도록)

리더 노드 장애복구 과정중 문제가 생겼을 때

- 쓰기 복제 요청을 팔로워노드가 받지 못했을 때 쓰기 데이터가 유실될 수 있음

- Split Brain 발생 가능

복제 로그 구현

구문(statement) 기반 복제

- SQL을 로그로 남기고 팔로워 노드에서 똑같이 실행하도록함
- NOW()나 RAND()같은 함수는 노드마다 다른 결과를 낼 수 있어서 조심해야함
- 자동 증가 컬럼(Auto Increment)이 있을 경우 같은 순서로 실행되어야만 데이터가 맞음
- 트리거, 스토어드 프로시저, 사용자 정의 함수 등 완벽하게 같지 않으면 데이터가 달라질 수 있음

로우(row, logical log) 기반 복제

- 명령(statement)이 아닌 Row의 Data를 그대로 로그로 남긴다
- 순서가 달라도 데이터는 알맞게 잘 들어간다
- 이진화된 데이터 로그여서 이 데이터를 캡처할 수 있음 이것을 변경 데이터 캡처(change data capture)라함

트리거 기반 복제

- 데이터베이스의 트리거 기능을 이용하여 복제 직접 가능

복제 지연 (Replica Lag...)

리더노드의 변경분이 팔로워노드에 아직 반영되지 않았을 때 읽을 경우 복제지연문제가 발생함

- 최종적 일관성
  - 시간에 지남에 따라 결국 가지게 되는 일관성을 뜻함 .. 이것이 바로 Replica Lag이다..! 리더에 변경분이 팔로워에게 결국 시간이 지남에 따라 반영될 것이기 때문 ..
- 쓰기 후 읽기 일관성
  - 자신이 쓰고 읽을 때 없을 수 있음 (리더에 쓰고 팔로워에 아직 반영이 안되어서!)

    - 쓰기후 일정시간동안 리더노드에서 읽게한다
- 단조 읽기(monotonic read)
    - 리더노드간에 복제속도 차이가 있을 때 사용자는 데이터가 있었다가 없어질 수 있음
    - 최신의 데이터를 읽었다면 그 데이터만 읽어야함
    - 사용자는 팔로워노드를 한번 읽었다면 이후로 계속 그것만 읽게하자
- 일관된 순서로 읽기
    - 일부 파티션이 다른 것보다 느리게 복제 되는 경우 사용자는 데이터를 다른 순서로 읽을 수 있음

복제 지연을 위한 해결책

- 트랜잭션으로 해결 가능함 → 분산 데이터베이스에서는 트랜잭션 안됨 .. 포기? → 다른 해결방법 있음

다중 리더 복제

- 리더가 복수이다
- 리더간 데이터를 맞추다가 충돌이 발생할 수 있다

다중 리더 복제의 이점

- 성능 → 단일 쓰기 노드에서는 무조건 쓰기노드로 가야하는데 가까운 곳에 쓰기노드를 위치시킬 수 있음!
- 데이터센터 중단 내성 → 다중리더에서는 데이터센터 고장나도 다른 리더를 사용하면 됨!
- 네트워크 문제 내성 → 다른 네트워크간 리더-팔로워 복제는 굉장히 불안정함! 이를 해소할 수 있음

쓰기 충돌 다루기

- 쓰기 전에 다른 리더노드들에게 쓰기대기를 하도록 할 수 있음 → 이거는 다중 리더의 장점을 잃어버림

- 충돌 회피 → 리더마다 변경할 수 있는 데이터를 제한둔다
- 일관된 상태 수렴(convergent) → 각 쓰기마다 고유한 ID(타임스탬프 등)을 가지고 쓰기의 승자를 결정한다.

데이터 충돌 자동 해소 알고리즘

아는척하기 좋은 것들

- 충돌 없는 복제 데이터 타입(conflict-free replicated datatype)
- 병합 가능한 영속 데이터 구조(mergeable persistent data structure)
- 운영 변환(operational transformation)

데이터 충돌 감지

현재 충돌감지 제대로 안되고 있다. 인과성을 알 수 없다 (어떻게 충돌이 되었는지 어느 리더로부터 어느리더로)

다중 **리더 복제 토폴로지**

- 원형 토폴로지 → 노드 → 노드 → 순서대로 메시지 전달
- 별모양 토폴로지 → 루트에서 다른 노드들로 전달
- 전체 연결 토폴로지 → 일반적인 것, 모든 리더가 모든 리더에게 연결되어 있어서 내결함성이 가장 높다. 하지만 일부 네트워크가 다른 네트워크와 속도차이가 나면 일부 복제 메시지가 다른 메시지를 **추월할 수 있다**
  → **추월할 경우 해소하기 위해 버전 벡터(version vector)라는 것을 이용함**

리더 없는 복제 (leaderless architecture)

- AWS의 다이나모에서 시작함. 이런걸 다이나모 스타일이라 한다..(갓마존)
- 모든 노드에 쓰기요청할 수 있다. 쓰기요청 받은 노드는 다른 노드들에게 복제요청을 한다.

읽기 쓰기

- 쓰기와 읽기를 병렬로 보낸다

- 쓰기노드중에 하나라도 성공하면 성공임!! → 그러면 특정 노드 읽기는 outdated 상태임 → 병렬로 읽기를 보내기 때문에 그중에 가장 최신값을 가져옴

복제 실패로 누락된 데이터는?

- 읽기 복구 → 사용자가 읽었을 때 다른 노드의 데이터들과 비교해서 outdated상태인지 알 수 있음 그리고나서 복제 다시한다!
- 안티 엔트로피 처리 → 백그라운드 프로세스가 다른 노드들과 데이터 같은지 계속 검사함 → 발견하면 처리 → 느리게 반영될 수 있음

읽기 쓰기를 위한 정족수

- 모든 노드 수 : n, 모든 쓰기 : w, 모든 읽기 : r
- 정족수 잘 설정해도 읽기 읽관성에 한계가 있음 (오래된 데이터를 읽을 수 있음)
- 느슨한 정족수를 통해 노드 많이죽었을때 조금 대응 가능

동시 쓰기 감지, 최종 쓰기 승리

최종 쓰기(이전 발생)은 어떻게 감지함?

- A는 삽입, B는 A로 삽입한 값의 증가라면? → 인과성이 있다(causally dependent)
- 두개의 삽입은 인과성이 없음
- 읽기 버전을 가지고 클라에서 저장하고 있음으로써 이전 발생에 대해 알거나 의존했다는 사실 알 수 있음

병합도 하나의 해결법...

버전 벡터

- 버전 번호를 통해 작업간 의존성을 파악했다
- 모든 복제본의 버전 번호 모음을 버전 벡터라고 부름

06. 파티셔닝

여러가지 단어로 불린다

→ 일반적으로 **샤딩**

→ 몽고DB,엘라스틱서치, 솔라클라우드에서 샤드(shard)

→ HBase에서는 리전(region)

→ 빅테이블에서는 태블릿(tablet)

→ 카산드라와 리악에서는 브이노드(vnode)

→ 카우치베이스에서는 브이버켓(vBucket)

파티셔닝할 때 데이터는 무슨 기준으로 나누는가??

키-값 데이터 파티셔닝

데이터 균등하게 분리하는 것이 파티셔닝의 목적임

→ 데이터가 한쪽으로 쏠렸다(skewed)라고 표현함

→ 이렇게 한쪽으로 몰려서 부학 높은 파티션을 핫스팟(hot spot)이라고 함

키 범위 기준 파티셔닝 (Range 파티셔닝)

→ 만들기 편함

→ 각 파티션에서 키를 정렬된 순서로 저장할 수 있음

→ 특정한 접근 패턴이 핫스팟을 유발하는 단점이 있음 (최신의 데이터만 본다면? 등등)

키의 해시값 기준 파티셔닝

→ 키를 해시연산함(md5 등)

→ 언어에 해시함수 내장되어 있어도 쓰지 마셈 자바의 Object.hashCode()는 프로세스마다 다른 결과를 냄

→ 일관성 해싱 가능

→ 범위 질의를 효율적으로 실행할 수 없음 (Index Range Scan 등... ㅠㅠ)

→ 해싱용 키와 정렬된 키를 복합 기본키로 사용하면 해결할 수 있긴함 (하지만 이렇게 하려면 첫번째 키는 고정하고 사용해야함)

쏠린 작업부하와 핫스팟 완화

해시파티셔닝이 균등하게 분배해주지만 특정한 첫번째 키 고정으로 사용해서 계속 쓰면 결국 균등 분배가 무너짐

→ 이럴 때는 그 첫번째 키에 임의의 숫자(10진수 두자리 등)를 붙이면 파티셔닝 될텐데

→ 위처럼 하면 애플리케이션에서 추가작업이 필요함 ...

파티셔닝과 보조 색인

보조색인이 들어가면 그 키로 필터나 정렬해야하는데 그러면 파티셔닝된 데이터에서 어떻게 그렇게함??ㅇ어

문서 기준 보조 색인 파티셔닝

문서 데이터베이스에서 파티셔닝하면 각 문서는 각각 다른 파티션으로 들어감. 그럼 그 파티션은 각자 다른 문서들을 각각 관리함 → 이래서 문서 파티셔닝 색인은 직역 색인(local index)라고 함

이 때 보조색인으로 질의하면 어떡함?

→ 결국 파티션 전부에 질의해서 결과를 가지고 모아서 반황해야함

→ 위같은 질의 방법을 스캐터/개더(scatter/gather)라고 함

→ 위처럼 질의하면 결국 마지막 질의 결과까지 기다려야하는데 이러면 꼬리지연(tail latency)가 발생할 수 있음

용어 기준 보조 색인 파티셔닝

문서기준 보조 색인이 지역색인이라면 용어기준 보조 색인은 전역 색인(Global Index)임

→ term-partitioned 라고 표현함

→ 키와 값이 색인 하나에 저장됨

→ 전역색인이라 모든 노드가 이 색인을 가지고 있어야함

→ 용어의 해시값을 이용해 파티셔닝하면 부하가 좀 더 고르게 분산될 수 있음

→ 스캐터/개더 안해도 되어서 일긱에 좋음 그런데 쓸 때 모든 문서는 색인에 반영되어야 하기 때문에 느려질 수 있음 → 그래서 비동기로 처리함

파티션 재균형화 (Partition Rebalancing)

쓰면 안되는 방법 : 해시 값에 모드 N 연산 실행

→ 노드 추가되면 데이터가 다른 노드로 이동되어야함

파티션 개수 고정

한 노드에 복수의 파티션을 가지고 있다가 노드 추가되면 그 노드에 몇개씩 줌

→ 리악, ES, 카우치베이스, 볼드모트에서 이렇게 사용됨

→ 처음 설정된 파티션 개수가 사용 가능한 노드 대수의 최대치가 되므로 미래에 증가될 것을 수용하기에 충분히 높은 값으로 설정해야함. 너무 높으면 파티션관리에 큰 오버헤드가 듬

→ 데이터셋 크기가 너무 유동적이라면? → 동적 파티셔닝

동적 파티셔닝

파티션 개수가 전체 데이터 용량에 맞춰 조정됨

→ 파티션 크기가 임계값 아래로 떨어지면 인접한 파티션과 합쳐짐

→ 빈 데이터베이스는 파티션 경계 정하기 어려움 **사전 정보**가 없으면 이 파티셔닝은 불가능함

→ 빈 데이터 베이스에 초기 파티션 집합 설정 가능 → 사전 분할(pre-spliting)

노드 비례 파티셔닝

노드당 할당되는 파티션 개수를 고정함

→ 노드가 하나 늘어나면 고정한 파티션 늘어남 → 파티션이 늘어나면 고정된 개수의 파티션을 무작위로 선택해 분할함

→ 파티션 경계를 무작위로 선택하려면 해시 기반 파티셔닝을 사용해야함

운영: 자동 재균형화와 수동 재균형화

→ 자동 재균형화 예측 어려움 → DB모니터가 장애로 인식할 수 있음

→ 데이터 과부화 걸림 → 자동으로 재균형화 → 과부화 더 심해짐 → 망함

→ 수동으로 하자

요청 라우팅

→ 서비스 찾기 (Service Discovery)

→ 어느 노드에 어느 키가 있는지 어떻게암?

1. 클라이언트가 아무노드에 요청하고 만약 그 노드에 데이터 없다면 그 노드가 다른 노드로 요청 돌려줌
2. 라우팅 계층을 두고 그 라우팅 계층이 어느 노드에 어떤 데이터가 있는지 모두 가지고있음
3. 클라이언트가 노드들 정보를 모두 가지고있음 → 올바르게 요청 가능

보통 (2)의 방법으로 함

주키퍼(Zoo Keeper)같은 애가 데이터 메타뎅이터 관리하면서 라우팅계층이 최신정보를 유지할 수 있도록 해줌

→ HBase, 솔라클라우드, 카프카 등이 주키퍼 사용함

몽고DB는 아키텍처 비슷하지만 자체적인 설정서버(config 서버)구현에 의존하고 몽고스(mongos) 데몬을 라우팅 계층으로 사용함

카산드라와 리악은 가십 프로토콜(gossip protocol)이란걸 사용함 → 클러스터 상태 변화를 노드 사이에 퍼뜨림. **(1)의 방법**

병렬 질의

분석용으로 사용할 때 대규모 병렬 처리(massively parallel processing, MPP) 등도 있음

→ 병렬로 모든 노드에 질의해서 이걸 다시 aggregation함

정리

→ 파티셔닝의 목적? 핫스팟 안만들면서 데이터와 질의 부하를 여러 장비에 균일하게 분배하기

파티셔닝 기법

→ 키 범위 파티셔닝

→ 해시 파티셔닝

파티셔닝 색인

→ 문서 파티셔닝 색인

→ 용어 파티셔닝 색인

07. 트랜잭션

트랜잭션이 머냐? → 일련의 작업 단위 → 성공(commit) 혹은 실패(abort, rollback)

→ 프로그래밍 모델을 단순화하려는 목적 → DB가 이런것도 해준다..! → 안전성 보장

- read commiteed
- snapshot isiolation
- seirializability

ACID

ACID Compliant(ACID 준수)가 뭐임..?

- Atomicity (원자성)

  어떤 작업의 시작과 끝만 존재한다. 중간은 없다..! → 성공하면 Commit, 실패하면 Rollback

- Consistency (일관성)

  데이터는 불변식(invariant, immutable)이어야 한다...!

- Isolation (격리성)

  동시에 실행되었을때 트랜잭션은 서로 격리됨. 같은 데이터에 접근하더라도 → 직렬성(serialzability) 라고함

  동시에 실행되어도 **순차적으로** 결과가 커밋됨

- Durability (지속성)

  트랜잭션이 커밋되었다면 하드웨어 결함이 발생하거나 데이터베이스가 죽더라도 모든 데이터는 손실이 없어야함

  복구용으로 쓰기전 로그(write-ahead log) 등이 있는 이유.

  지속성을 보장하려면 트랜잭션이 성공되면 복제될때까지 기다리고 성공했다고 응답해줘야함

  → 완벽한 지속성은 없음

단일 객체 연산과 다중 객체 연산

→ 원자성이 보장되지 않으면 하나의 트랜잭션이 도중에 실패하거나 했을 때 변경된 값들이 남아있을 수 있음

→ 격리성이 보장되지 않으면 다른 트랜잭션의 커밋되지 않은 내용을 읽는 Dirty Read가 발생할 수 있음

→ 트랜잭션의 시작과 끝은 어떻게 암 ? → RDB의 경우 클라이언트와 서버간 TCP연결, BEGIN TRANSACTION과 COMMIT 사이.

트랜잭션에서 원자성과 격리성이 젤 중요한듯?

단일 객체 쓰기

단일 객체라도 원자성과 격리성이 보장되지 않을 수 있음 → 갱신 손실(lost update) 등 동시성에 의해서

그러면 어떻게 해결함?

- 20KB Json 문서를 데이터베이스에 업데이트하다가 10KB만 읽고 변경하고 나머지 10KB 실패하면 어떡함?
- 결국 read-modify-write 연산 자체가 문제가 잇음
- 데이터베이스에서 좀더 복잡한 원자적인 연산을 제공하거나 (Redis의 INCR 같은 연산)
- compare-and-set 연산도 참고해보자

트랜잭션은 보통 다중 객체에 대한 다중 연산을 하나의 실행단위로 묶는 것임. → 단일객체도 이렇게 원자성 격리성 보장이 힘든데 트랜잭션은 어케함?

오류와 어보트(abort)

트랜잭션의 핵심 기능은 오류가 생기면 어보트 되고 안전하게 재시도할 수 있음

→ 모든 시스템이 이 철학을 따르지 않음 → 이럴 때는 애플리케이션이 복구 책임이 있음

오류나면 재시도를 해야할까?

→ 간단하고 효과적인 메커니즘이지만 완벽하지는 않음.

→ 영구적으로 실패하는 트랜잭션이라면(제약조건 위반등) 재시도 의미가 없음

→ 네트워크 지연으로 인한 실패라면 재시도할경우 트랜잭션이 두번 실행될 수 있음

격리 수준 ..

→ 직렬성(serializability) 수준의 격리 수준을 사용하면 트랜잭션 동시성 문제 없음 왜? 동시 동작을 안하니까

→ 하지만 병렬성, 동시성을 지키기 위해 어느정도 완화된 격리 수준을 사용하는 것이 일반적이다

커밋 후 읽기 (Read Committed)

→ 커밋된 데이터만 읽는다

→ Dirty Read 하지 않는다 (다른 트랜잭션의 변경분을 Commit 전까지 읽지 않음)

→ Dirty Write 하지 않는다 (이전 트랜잭션이 이후 트랜잭션의 변경된 commit값을 덮어쓰지 않음)

→ Oracle 11g, PostgreSQL 등이 이것이 기본값임

Dirty Read와 Dirty Wirte 방지는 결국 잠금으로 구현함

→ Dirty Write는 어떤 트랜잭션에서 Row를 갱신할 때 그 Row에 대한 쓰기 잠금을 commit 전까지 가지고있음

→ Dirty Read는 트랜잭션에서 Row을 읽을때 잠깐 읽기(공유) 잠금을 획득하고 읽기 완료 후 다시 잠금을 품

→ 하지만 Dirty Read 방지를 위해서 읽기 잠금을 획득할 경우 다른 쓰기 트랜잭션에 때문에 기다릴 수 있어서 그냥 잠금 안검, 커밋된 값만 읽자..!

스냅샷 격리와 반복 읽기(Snapshot Isolation and Repeatable Read)

각 트랜잭션은 데이터베이스의 일관된 스냅샷으로부터 읽는다

→ PostgreSQL와 InnoDB를 사용하는 MySQL, Oracle 등이 사용함

어떻게 구현함?

→ 읽을 때 쓰기 트랜잭션이 읽는 것을 방해하면 안됨 (서비스 성능이 낮아지니까)

→ MVCC라는 것을 사용함. → 객체의 여러 버전을 유지하는 것

PostgreSQL의 구현 (MySQL도 동일한걸로 알고있음)

→ 트랜잭션마다 계속 증가하는 고유한 트랜잭션 ID(txID)가 붙음

→ DB에 데이터 쓸 때마다 쓰기를 실행한 트랜잭션 ID가 같이 붙음

→ 각 트랜잭션은 끝날때까지 자신의 트랜잭션 ID까지의 데이터만 읽음 (이후 트랜잭션의 데이터는 읽지 않음)

색인은 어떻게함??

색인은 여러 버전에 대한 데이터를 어떻게 참조하고 있을까?

→ PGSQL의 경우 객체의 다른 버전들이 같은페이지에 저장될 수 있으면 색인 갱신 회피함

→ append-only/copy-on-write 방식을 사용함. B트리를 덮어쓰지 않고 그것들을 복사해 새로운 B트리를 만들어 관리함 → 백그라운드에서 컴팩션 과정등이 필요함

갱신 손실 방지 (Lost Update)

- read-modify-write에서 자주 발생하는 문제이다.. **정말~~~ 자주 발생함**
- 카운터같은 것들 ..

해결 어떻게 할까?

- 원자적 쓰기 연산

```
UPDATE contuer SET value = value + 1 WHERE key = 'foo';
```

위와 같은 연산인데. 이 연산을 실행할 때 베타적락(exclusive lock)을 획득하고 사용함. 이런 기법을 커서 안정성(cursor stability) 라고 함

Redis에서는 우선순위 큐(priority queue) 같은 데이터 구조를 변경하는 원자적 연산을 제공함

하지만 .. ORM같은 것 사용하면 이런 연산을 사용하지 않을 수 있음

- 명시적 잠금

read-modify-write 연산이 필요하다면 명시적으로 exclusive lock을 획득하고 진행한다.

`SELECT ~ FOR UPDATE` 같은 구문들 ..

- 갱신 손실 자동 감지

트랜잭션 관리자(데이터베이스)에서 자동으로 갱신 손실을 감소하고 문제 되는 트랜잭션을 어보트(abort)처리한다. 데이터베이스가 이 확인을 스냅샷격리와 알잘딱해서 효율적으로 수행가능함. Oracle의 직렬성, SQL 서버의 스냅샷격리는 이를 지원하지만 MySQL은 지원하지 않음.

- Compare-and-set

쓰기를 수행할 때 이전에 읽은 값과 일치하지 않으면 변경을 반영하지 않는다. (어보트 처리해버린다)

충돌 해소와 복제

다중 리더 혹은 리더리스 데이터베이스에서는 어떻게하지? 잠금과 compare-and-set도 결국 단일 리더일 때 유효함.

→ 다중 리더간 데이터 충돌을 감지하고 병합하거나 해야됨

→ 최종쓰기승리 → 결국 갱신 손실이 발생하기 쉬움

쓰기 스큐(write skew)

- 이것도 결국 read-modify-write이다 ..
- 예를들어 어떤 주문을 취소한다고 해보자.. 동시에 한 주문에 대해 취소하는 요청을 보내면 둘다 주문취소 로직이 돌것이다. 왜 Why? 두트랜잭션 모두 읽을 시점에는 주문이 취소된 상태가 아닐 것이다.

어떻게 해결함?

- 여러 객체가 원인이므로 단일 객체 연산은 도움이 되지 않음
- 읽을 때 명시적으로 락을 걸거나 직렬성 격리 수준을 이용한다 → 만약에 존재하지 않을 때 쓰는 로직같은 경우에는 이방법도 불가능함 .. 락을 못거니까

→ 이것처럼 다른 트랜잭션의 쓰기 결과가 다른 트랜잭션의 검색 질의 결과를 바꾸는 효과를 팬텀이라함.

그럼 이런 없는 객체에 대한 팬텀은 어떻게 해결함?

→ 데이터 미리 만들어 놓으면 됨. 하지만 그렇게 좋은 방법은 아님

직렬성(Serializable) 격리 수준

동시성 문제를 해결하는 가장 좋은 방법은 동시성을 제거하는 것이다..! (...?)

→ 트랜잭션을 순차적으로 하나씩 처리

→ 단일스레드에서 트랜잭션 처리 (Redis처럼)

트랜잭션을 스토어드 프로시저 안에 캡슐화하기

트랜잭션이 애플리케이션과 DB간에 대화형식 (하나 질의하고.. 또 다음에 또 질의하고)로 이루어지면 성능적인 측면에서 오버헤드가 크다. (네트워크 지연등이 있으니까)

프로시저 써볼래? → 음.. 현실적으로 가능한가?

파티셔닝

여러 파티션에 걸친 트랜잭션은 성능이 만힝 떨어질 수 있음 → 코디네이션 오버헤드가 있음

2단계 잠금 (2PL, Two-Phase Locking)

> TIP : 2PC(two-phase commit)과 다르다 → 9장 가자

스냅샷 격리(반복 읽기, Repeatable Read)에서는 다음을 꼭 지킨다

- 읽는 쪽은 결코 쓰는 쪽을 막지 않으며 쓰는 쪽도 결코 읽는 쪽을 막지 않는다

하지만 격리성 격리 수준과 2PL은 다르다

- 읽는쪽에서도 쓰는 쪽을 막고 쓰는 쪽에서도 읽는 쪽을 막는다

그러면 격리성 격리 수준은 어떻게 구현함? 그것이 바로 2단계 잠금이다

- 읽기 잠금, 공유 잠금(write lock, shared lock) → 읽기 잠금을 걸면 다른 트랜잭션에서 읽기잠금을 공유(share)할 수 있지만 쓰기잠금을 얻을 수는 없음
- 쓰기 잠금, 독점 잠금, 베타적 잠금(write lock, exclusive lock) → 쓰기 잠금을 걸면 다른 트랜잭션에서 읽기잠금도 쓰기잠금도 얻을 수 없음

→ 교착상태에 빠지기 쉽다.

→ 성능 당연히 안좋다 계속 트랜잭션끼리 기다려야하니까

직렬성 격리 수준에서는 없는 객체에 대한 팬텀을 어떻게 해결할까?

서술 잠금으로 해결함.. (predicate lock)

어떤 검색 조건에 대하여 락을 거는 것이다 (SQL Statement, SELECT 절에 락을 건다고 생각할 수 있음 → 어지럽네)

위같은 팬텀을 막는 또다른 방법 색인 범위 잠금과 다음 키 잠금

- 색인 범위 잠금 (index-range locking) → 인덱스 범위에 대해 잠그는 것 (검색조건 등)
- 다음 키 잠금(next-key locking) → 인덱스 범위뿐만 아니라 Next Key라고 하는 것도 잠근다 → 이게 뭐냐면 그 인덱스 범위내에 또 새로운 Row가 생기는 것을 방지하는 것임

→ 서술 잠금을 간략화한 것임

→ 서술 잠금은 오버헤드가 크니까 좋은 타협일 수 있음

직렬성 스냅숏 격리(SSI, Serializable Snapshot Isolation)

직렬성 격리와 좋은 성능은 근본적으로 공존할 수 없을까? → 이것을 해결한 알고리즘이 SSI이다. 궁금하면 찾아보자

비관적 동시성 vs 낙관적 동시성 제어 (비관적 잠금 vs 낙관적 잠금)

이전까지 본 잠금들은 (2단계 잠금이나.. 직렬성 격리 수준이나) 모두 비관적 잠금이다. SSI는 낙관적 동시성 제어 기법이다.

→ SSI는 스냅샷 격리(반복 읽기)를 이용하여 트랜잭션이 커밋될 때 직렬성 충돌을 감지하고 어보트시킬 트랜잭션을 결정하는 알고리즘을 추가함

→ 이걸 어떻게할까? 두가지 가능성이 있음

- 오래된(stale) MVCC 객체 버전을 읽었는지 감지하기(읽기 전에 커밋되지 않은 쓰기가 발샐함)
- 과거의 읽기에 영향을 미치는 쓰기 감지하기(읽은 후에 쓰기가 실행됨)

08. 분산 시스템의 골칫거리

부분 장애는 항상 있을 수 있다.. 부분 장애를 견딜 수 있는 내결함성을 지닌 시스템을 구축하자

신뢰성 없는 네트워크

분산 시스템의 요소들은 대부분 네트워크만을 통해서 통신한다. → 일반적으로 비동기 패킷 네트워크다. (응답을 기다리지 않음) → 잘못된것들은 보통 타임아웃(Time out)으로 처리한다

> TIP : Network Partition(네트워크 분단) = Netsplit(네트워크 분리) = Network fault(네트워크 결함)

결함 감지

시스템에서 자동으로 노드의 결함과 장애를 감지할 수 있어야한다 → Health Check

타임아웃과 기약없는 지연

타임아웃 시간 설정 어떻게 할래?

→ 너무 짧으면 노드가 일시적으로 느려졌을 때 죽었다고 판단할 수 있음 (갑작스러운 RTT 증가)

→ 너무 길면 노드가 장애라고 판단하는데 너무 긴 시간이 걸림

네트워크 혼잡과 큐 대기

결국 네트워크 트래픽이 터질 때 큐대기가 발생하면서 지연시간이 느려짐 어떤 상황에서?

→ 스위치에서 패킷 전송할 때 큐에 넣고 순차적으로 하나씩 보냄

→ CPU 코어가 바쁘면 OS가 들어온 요청을 큐에 넣음

→ TCP 흐름제어

→ 위같은 것들이 네트워크 지연에 변동성을 주고 예측하기 어려운 지연을 발생시킨다. → 변동성이 크다는 것은 예측하기 힘들다는 거 → 측정 잘하자. 응답시간 분포에따라 자동으로 타임아웃 조절해보자.

동기 네트워크 vs 비동기 네트워크

패킷 전송 지연 시간의 최대치가 고정되어 있고 패킷을 유실하지 않는 네트워크에 기댈 수 있으면 좋지 않을까?

→ 전화 회선은 극단적으로 높은 신뢰성을 보장한다. 정적으로 대역폭을 설정해서 양쪽이 전화할 때 어떠한 방해도 없고 최대지연이 있다.

→ 인터넷에서 위 같은 대역폭 예측이 가능한가? 불가능함 인터넷이 패킷 전송 교환 방식을 사용하는 이유 → 동적으로 대역폭을 설정하고 대응하기 위해서 → 이는 순간적으로 몰리는 트래픽(bursty traffic)에 효과적이다.

신뢰성 없는 시계

분산시스템에서는 시스템들이 서로 네트워케 의존해 연결되어 있다. 각각에서 재는 시간이 달라질 수 있다. 이런 것들 어떻게 맞춰줄것인가?

> TIP : 네트워크 시간 프로토콜(Network Time Protocol) → 시간 동기화 프로토콜

일 기준 시계

일 기준 시계는 현재 날짜와 시간을 알고 싶을때! 사용하면 좋음. 에포크(epoch) 타임이라고 흔히 부르는 것

단조 시계

단조 시계는 어느 시점부터 어느시점까지 시간이 얼마나 걸렸는지 측정할 때 좋음 (수행시간 측정 등)

시계 동기화와 정확도

단조 시계는 동기화가 필요 없지만 일 기준 시계는 NTP 서버나 다른 외부 시간 출처에 맞춰 설정되어야 유용함

- 컴퓨터 수정 시계의 드리프트(drift) 현상 → 컴퓨터 온도에 따라 시계가 더 빨리 돌거나 더 느리게 돔
- NTP 서버와의 통신 지연
- 윤초

→ 정밀 시간 프로토콜 (PTP, Precision Time Protocol)

이벤트 순서화용 타임스탬프

최종 쓰기 승리(LWW, Last Wrtie Win) 등을 구현하기 위해서 타임스탬프를 사용할 경우 가장 최근 데이터가 살아남는다. 여기서 "최근"의 정의는 무엇일까?

→ 시간이 잘못되어있다면 이것은 잘못된 최근을 불러올 수 있음

→ 논리적 시계(Logical Clock)을 사용하자 (증가하는 카운트 등)

→ 일기준시계와 단조시계는 물리적 시계(Physical Clock)이라고 함

스냅숏 격리도 물리적 시계를 사용하려고 하지만 아직 사례가 없음 → 분산 트랜잭션 시맨틱 용... 시계 동기화

시계의 신뢰 구간 (→ True Time)

위처럼 시간이 밀리거나 당겨지겨나 한다. 그럼 우리가 받는 시간은 결국 진짜 시간(True Time)의 근사치이라는 뜻임.. 이를 위해서는 시계의 신뢰 구간을 알 필요가 있음

→ 구글의 True Time API는 시계의 신뢰구간도 같이 준다(가장 빠를경우, 가장 느릴경우)

프로세스 중단과 실시간 시스템

시간을 확인하는 시점과 실제 로직이 처리되는 시점이 프로세스 중단과 지연등으로 인해 그 사이 시간이 달라지는 경우가 있을 수 있음

→ 데드라인(Deadline)을 주고 이 시간 안에 처리되지 못할경우 Timeout → Hard Real-Time

→ RTOS (실시간 운영체제)

→ 프로세스 선점 스케줄링

지식, 진실, 그리고 거짓말

위처럼 분산시스템에서 여러 이상현상이 발생되면 무엇이 진짜인지 어떻게 확인함??

→ 진실은 다수결 → 노드가 죽은 것을 어떻게 판단함??? → 다른 노드들의 다수결 투표로 이루어짐

펜싱 토큰 (fencing tokne)

임차권 등을 사용할 때 오래된(outdated) 토큰이나 임차권은 거부시켜버린다

비잔틴 내결함성

노드와 서비스도 "거짓말"을 할 수 있다.

시스템 모델과 현실

분산 시스템은 위처럼 말도 안되는 일이 많이 발생한다. 분산시스템의 다양한 결함을 견딜 수 있어야한다.

시스템 모델 종류

- 동기식 모델 → 네트워크 지연, 프로세스 중단, 시계 오차에 모두 제한이 있다고 가정 → 없는건 아님
- 부분 동기식 모델 → 대부분의 시간에는 동기식 시스템처럼 동작함 → 때때로 한계치 초과
- 비동기식 모델 → 타이밍에 대한 어떤 가정도 할 수 없음

노드용 시스템 모델

→ crash-stop 결함 → 노드 장애 = 노드 죽음

→ crash-recovery 결함 → 노드 장애 후 복원될 예정

→ 비잔틴 결함 → 다른 노드에게 거짓말 등 무슨일이든 할 수 있음

안정성(safety)과 활동성(liveness)

- 안정성 → 나쁜 일은 일어나지 않는다 → 속성이 깨진 특정 시점을 가리킬 수 있다
- 활동성 → 좋은 일은 결국(eventually) 일어난다 → 최종적 일관성 (eventual consistency) → 속성이 깨진 특성 시점을 가리킬 수 없다. 하지만 미래에는 그 속성을 만족시킬 수 있다

결론

→ 분산 시스템에서는 부분결함(partial failure)가 있을 수 있다는 것을 항상 생각하자

→ 이를 감지하기 위한 노력을 하자

→ 신뢰성이 없을 때 어떻게 이것을 내결함성 있게 구현할 수 있을지 고민해보자

09. 일관성과 합의

앞서 살펴본 것처럼 분산시스템에서는 여러 결함이 발생할 수 있다. 이런 것을 처리하는 간단한 해결 방법은 그냥 전체 서비스가 실패하도록 두고 사용자에게 오류메시지를 보여주는 것.

위같은 방법이 싫다면 결함을 견뎌낼(tolerating) 즉 내부 구성 요소 중 뭔가에 결함이 있더라도 서비스는 올바르게 동작하게 할 방법을 찾아야함

내결함성을 지닌 시스템을 구축하는 좋은 방법은 **유용한 보장을 해주는 추상화를 사용하고 애플리케이션이 이것에 의존하게 하는것 (예 : 트랜잭션 등)**

분산시스템에서는 어떤 이러한 추상화를 제공하는가?? 이것을 알아보는 것이 이번 장

일관성 보장

데이터베이스 복제는 약한 일관성 보장(최종적 일관성, eventual consistency)을 제공한다. 이런데에서 나오는 문제는 테스트도 어렵고 찾기도 힘들다. 어떻게 할까 ?

→ 강한 일관성 모델 중 하나인 선형성을 살펴보고 장점과 단점을 검토한다

→ 분산 시스템에서 이벤트 순서화 문제, 특히 인과성과 전체 순서화와 관련된 문제를 검토한다

→ 분산 트랜잭션을 원자적으로 커밋하는 방법을 알아본다. 분산 트랜잭션은 마침내 합의 문제의 해결책으로 우리를 안내해준다.

선형성

최종적 일관성을 지닌 데이터베이스에서 복제본이 하나만 있다는 환상을 만들어준다면??
→ 이것이 선형성이다.

선형성은 클라이언트가 쓰기를 성공적으로 완료하자마자 그 데이터베이스를 읽는 모든 클라이언트는 방금 쓰여진 값을 볼 수 있어야 한다

- 원자적 일관성(atomic consistency)
- 강한 일관성(strong consistency)

- 즉각 일관성(immediate consistency)
- 외부 일관성(external consistency)
- 최신성 보장(recency guarantee)

선형성은 모든 요청과 응답 시점을 기록하고 그것들이 유효한 순차 순서로 배열되는지 확인함으로써 시스템의 동작이 선형적인지 테스트할 수 있다.

결국 선형성은 모든 (동시적인) 요청이 선형적으로 아름답게 배치되는가..? (시간의 순서 등으로)

> TIP :
> 직렬성 → 트랜잭션들의 격리 속성 → 어떤 순서에 따라 실행되는 것처럼 동작하도록 보장
> 선형성 → 레지스터(개별객체)에 실행되는 읽기와 쓰기에 대한 최신성 보장 → 선형성은 트랜잭션에 묶이지 않는다

잠금과 리더 선출

리더 선출을 할 때 모든 노드들이 잠금 획득을 시도하고 성공한 노드가 리더가 된다 → 이게 가능하려면 "잠금 획득"이라는 것이 선형성이 있어야함 (순서가 있어야한다..!)

제약 조건과 유일성 보장

예를 들어 Unique Key 제약조건의 경우 쓰기 요청들이 모두 잠금 획득을 시도하고 쓰기 성공한 애가 남는다..!

채널간 타이밍 의존성

메시지 브로커 등 비동기로 처리되는게 있으면 쓰고나서 바로 읽을 수 없을 수도 있다..!

선형성 시스템

- 단일 리더 복제 (선형적이 될 가능성이 있음)

- 합의 알고리즘 (선형적)
- 다중 리더 복제 (비선형적)
- 리더 없는 복제 (아마도 비선형적)

선형성의 비용

예를 들어 다중리더 복제에서 리더간 네트워크가 분단되었을때 선형성(일관성)을 지킬것인가? 가용성을 지킬것인가? → CAP 정리

선형성을 깨는 것은 내결함성의 문제가 아닌 성능이다. (선형성을 보장하기 위해서는 성능을 많이 깍아먹어야함.. 이건 결국 직렬성과도 비슷한 것 같기도 하다)

순서화 보장

결국 순서화, 선형성, 합의 사이에는 깊은 연결 관계가 있음.

순서화와 인과성

순서화는 인과성을 보존하는데 도움을 준다

인과성에 의해 부과된 순서를 지키면 그 시스템은 인과적으로 일관적(casually consistent)라고 한다

인과적 순서가 전체 순서는 아니다

전체 순서(total order)는 어떤 두 요소를 비교할 수 있게 하므로 두요소가 있으면 항상 어떤 것이 더 크고 더 작은지 말할 수 있음

수학적 집합은 부분적으로 순서가 정해짐(partially ordered)

- 선형성

  모든 연산이 원자적으로 실행하며 데이터 복사본이 하나만 있는 것처럼 동작하면 두 연산에 대해 항상 하나가 먼저 실행됐다고 말할 수 있음

- 인과성

두 연산중 어떤 것도 다른 것보다 먼저 실행되지 않았다면 두 연산이 동시적 → 부분 순서(partially ordered)

선형성은 인과적 일관성보다 강하다

선형성은 결국 인과성을 내포한다..! → 어떤 시스템이던지 선형적이라면 인과성도 올바르게 유지한다

선현성까지 필요한 경우는 거의 없음 결국 필요한 것은 인과적 일관성임 → (부분 순서)

일관적 의존성 어케만듬?

일련번호 순서화 (feat. timestamp 순서화)

→ 일련번호나 타임스탬프써서 이벤트의 순서를 정한다

→ 물리적 시계 대신 논리적 시계에서 얻어도됨

(다중리더, 리더 리스) 비인과적 일련번호

→ 짝수 / 홀수로 나누기 → 짝수 / 홀수 간 순서 알 수 없음

→ 물리적시계에서 얻을 경우 → 시계 스큐 (time skew)

→ 일련번호 블록을 미리 할당 → 순서 알 수 없음 (짝수 / 홀수와 같은 맥락)

전체 순서 브로드캐스트 (total order broadcast, atomic broadcast)

모든 복제 서버가 같은 쓰기 연산을 같은 순서로 처리하면 서버들은 서로 일관성 있는 상태를 유지함

→ 상태 기계 복제(state machine replication)이라고 함

전체 순서 브로드캐스트

→ reiable delivery, totally ordered delivery

→ 비동기식임, 메시지는 고정된 순서로 전달되지만 **언제 전달될지는 모름**

→ 쓰기 선형성은 보장하지만 읽기 선형성은 보장하지 않는다

→ 선형성이란 것은 최신의 데이터를 읽을 수 있어야하는데 전체 순서 브로드캐스트는 비동기로 쓰기 순서를 유지하기 때문에 생성하고 읽었을때 복제되지 않았을 수 있음

분산 트랜잭션과 합의

여러 노드들이 뭔가에 동의하게 만드는 것

→ 리더 선출

→ 원자적 커밋 → 모든 노드에서 트랜잭션이 원자적으로 커밋(commit)되거나 어보트(abort)되거나.

트랜잭션은 커밋된 이후에 **보상 트랜잭션**이 아니라면 어보트 되면 안됨

> TIP : 보상 트랜잭션 패턴은 애프리케이션에서 직접 Rollback하는 Query를 날리는거 (MSA Saga Pattern의 보상 패턴과도 같음)

2PC (2단계 커밋)

→ 데이터 쓰기 → 준비(1단계) → 커밋(2단계) 과정을 거친다

→ 트랜잭션에 참여한 노드들은 참여자(participant)라고 함

→ 각 노드에 준비 요청을 보내서 커밋을 할 수 있는지(dry run) 물어본다

→ 모든 노드가 준비되었다고 하면 커밋하고 트랜잭션 로그에 기록함 → 이를 커밋 포인트라고함

> TIP: 여러 노드에 걸쳐 이런 것을 제어하는 것을 코디네이터(coordinator)라고 함

코디네이터 장애

→ 코디네이터가 준비요청을 보내기 전에 장애가 나면 참여자가 트랜잭션 어보트가 가능함

→ 준비요청에 "네"라고 대답하고 코디네이터가 장애가 난다면 어보트할 수 없음

→ 위같은 상황이면 코디네이터가 복구되기를 기다릴 수밖에 없음

→ 위같은 이유로 2PC는 블로킹 원자적 커밋 프로토콜(blocking atmoic commit protocol)이라고 함

→ 위를 해결하기 위해 3단계 커밋(3PC)라는 알고리즘(논블로킹 원자적 커밋)이 있는데 이건 완벽한 장애 감지기 메커니즘이 필요함 → 궁금하면 찾아보길

현실의 분산 트랜잭션

- 데이터베이스 내부 분산 트랜잭션
- 이종 분산 트랜잭션

XA 트랜잭션

의심스러운 트랜잭션

- 트랜잭션이 의심스러운 상태 (동작은 안하고 커밋해야할지 어보트해야할지.. 모를때) 코디네이터가 장애가 터졌을 때 등 → 매뉴얼하게 처리하거나 → 경험적 결정(heuristic decision)

분산 트랜잭션 제약

→ 코디네이터는 대부분 SPOF 시스템임 → 코디네이터 장애시 서비스 중단임 그냥 ..

→ 코디네이터가 같이도는 애플리케이션 서버는 상태 비저장 서버가 아니다.. stateful하다

→ XA는 SSI(직렬성 스냅숏 격리)랑 같이 돌지 않는다

→ XA가 아닌 데이터베이스 내부 분산 트랜잭션은 제한이 크지 않다. 2PC가 성공적으로 커밋하려면 모든 참여자가 응답해야함 → 하나라도 고장나면 장애 증폭

합의 알고리즘

- 균일한 동의 : 어떤 두 노드도 다르게 결정하지 않는다
- 무결성 : 두번 결정하지 않는다
- 유효성 : 한 노드가 v를 결정하면 v는 어떤 노드에서 제안된 것이다
- 종료 : 죽지 않은 노드는 결국 어떤 값을 결정한다

분산 트랜잭션, 합의 알고리즘쪽 한번 보자 ..