Part 1 데이터 시스템의 기초

태그	
생성일	@2021년 8월 29일 오후 2:53

01. 신뢰할 수 있고 확장 가능하며 유지보수하기 쉬운 애플리케이션

- 이제 애플리케이션은 계산 중심(compute-intensive) → 데이터 중심(data-intensive)으로 이동
- 즉 신뢰할 수 있고 확장 가능하며 유지보수하기 쉬운 데이터 시스템 구축이 중요함

데이터 시스템

데이터를 다루는 도구들이 많아짐, 또한 새로 나오는 도구들은 효율적으로 수행할 수 있는 Task가 존재함. 즉 여러 도구들을 애플리케이션코드로 묶어서 잘 사용해야함. 즉 여러 데이터 쓰기/읽기 도구들과 이를 애플리케이션으로 알잘딱 묶는 것. 이것을 통틀어 데이터 시스템이라고 표함.

데이터 시스템의 관심사

- 신뢰성 (Reliability)
    - 어떤 상황에서도 시스템은 지속적으로 올바르게 동작(원하는 성능 수준에서 정확한 기능 수행)
- 확장성 (Scalability)
    - 데이터 양, 트래픽 양, 복잡도가 증가해도 이를 처리할 수 있는 적절한 방법이 있어야함

- 유지보수성 (Maintinaibility)
    - 생산성이 높아야함

> Tip : fault (결함) ≠ failure (장애)
> 장애는 막을 수 없지만 결함은 막을 수 있음 (fault-tolerant, resilient)
> e.g. Netflix Chaos Monkey

신뢰성

SLA(서비스 수준 규약) 생각해보자. 시스템 정상동작의 멈춤은 비즈니스적 손해, 사용자에 대한 책임도 생각해야 함.

신뢰성을 깨뜨리는 것들

- 하드웨어 결함
    ▼ 해결
        - 중복으로 해결한다 (SPOF 제거하자)
- 소프트웨어 오류
    ▼ 해결
        - 빡집중 코딩
        - 빈틈 없는 테스트
        - 프로세스 격리
        - 죽은 프로세스 재시작 허용
        - 모니터링
- 인적 오류
    ▼ 해결
        - 교육 잘하자
        - 모니터링, 알림 설정 잘하자

- 샌드박스(stage, dev 서버등) 제공하자
- 테스트 잘하자
- 인터페이스 설계 잘하자. 애초에 잘못된 행동을 못하도록 제약 걸자
- rollback과 failover 설정 잘하자

확장성

부하를 다루기 위해 어떻게 해야하는가. 모든 서비스/시스템에 맞는 아키텍처는 없다. 현재 상황에서 최선의 선택을 하자.

- 부하를 정확히 기술하자/측정하자
  - 부하 매개변수(load parameter) 잘 나누자 (CPU, Memory, Request Per minute 등)
  - 부하 매개변수가 각각 증가되었을 때 시스템에 어떤 영향이 있을지 생각하자
- 성능을 정확히 기술/측정하고 목표를 설정하자
  - 일반적으로 백분위가 좋다 → 꼬리지연시간(tail latency) 로 측정하자 → p99, p999

> Tip : latency (지연) ≠ response time (응답 시간)
> 클라이언트 기준에서 측정한 것이 response time이다. Queue의 경우 하나의 메시지로 뒷 작업들이 전부 느려질 수 있다. 이 때 latency는 빠르고 response time은 느리기 때문에 response time 측정이 중요함

부하 대응 접근 방식

- scaling up (vertical scaling, 수직확장)
- scaling out (horizontal scaling, 수평확장)

  → 다수 장비에 분산하는 아키텍처를 비공유(shared-nothing)아키텍처라고 한다. 소프트웨어의 stateless application과는 조금 다름. 데이터시스템(DB 등)도 scaling out될 수 있어야 shraed-nothing 아키텍처라고 할 수 있음.

  → DB는 scaling out이 어려우니 최대한 단일노드로 버티는 추세

유지보수성

버그 수정, 시스템 운영 유지, 장애 조사, 새로운 플랫폼 적응, 새 사용 사례를 위한 변경, 기술 채무 상환, 새로운 기능 추가 등. 모든 개발자는 Legacy 시스템을 좋아하지 않는다.

- 운용성 (operavility)
  -
- 단순성 (simplicity)
  - 추상화 잘하자
- 발전성 (evolvability)
  - 유연성(extensibility), 수정 가능성(modifiability), 적응성 (plasticity)

02. 데이터 모델과 질의 언어

관계형 모델과 문서 모델

너무나 익숙한 관계형 데이터베이스, 한계를 깨기 위해 NoSQL이 등장함

> Tip : 다중 저장소 지속성(polyglot persistence) 관계형 데이터베이스와 비관계형 데이터베이스가 함께 사용되며 데이터 시스템을 이루고 데이터 지속성을 이룬다

객체 관계형 불일치 (OR Mapping, Object - Relation Mapping)

> 임피던스 불일치 (impedance mismatch)라고도 한다

- 데이터 스토어를 이용하는 개발자라면 풀어야하는 해결못한 숙제..
- NoSQL을 이용하면 임피던스 불일치를 RDB보다는 줄일 수 있겠지만 JSON이 가진 문제가 있다. (4장)
- RDB 중복제거, 정규화 좋음 NoSQL 비정규화, 데이터 중복으로 인한 쓰기 오버헤드와 불일치 발생할 수 있음. 다대일 → 문서모델(NoSQL)에 적합하지 않다..

문서 데이터베이스는 역사를 반복하고 있나? → 아직 네트워크모델 수준의 접근경로를 입력하지 않음..

어떤 데이터 모델이 애플리케이션코드를 더 간단하게 하는가?

→ 결국 애플리케이션 상황에 따라 다름.

→ 다대다, 다대일이 많을 때 문서 데이터 모델은 매력적이지 못하다.

→ Schmaless에 대해서 생각해보자? 좋은거 맞나? NoSQL은 Schemaless가 맞나?

→ 쓰기 스키마(schema-on-write), 읽기 스키마(shcema-on-read)로 나누어 생각해보자. NoSQL은 읽기스키마라고 볼 수 있다.

→ 점점 관계형 데이터베이스와 문서 데이터베이스가 서로 비슷해져가고 있다.. 서로의 단점을 보완하기 위해..

데이터를 위한 질의 언어 : 선언형 질의와 명령형 질의

- SQL
- 맵리듀스 (Map, Reduce)

그래프형 데이터 모델

정점(vertex)와 간선(edge)로 이루어진 데이터. 다대다가 일반적이라면? → 그래프형 데이터 모델 고려해보자

- 속성 그래프 모델 (네오포제이(Neo4j), 타이탄(Titan))

- 트리플 저장소 모델 (데이토믹(Datomic), 알레그로그래프(Allegrograph)
- 선언형 질의 언어 : 사이퍼(Cypher), 스파클(SPARQL), 데이터로그(Datalog)
- 명령형 질의 언어 : 그렘린(Gremlin), 프리글(Pregel)

속성 그래프

- 고유한 식별자
- 유출(outgoing), 유입(incoming) 간선 집합
- 속성 컬렉션 (Key-Value 쌍)
- 간선은 다음 요소로 구성됨
  - 고유한 식별자
  - 꼬리 정점(간선이 시작하는 정점), 머리 정점(간선이 끝나는 정점)
  - 두 정점 간 관계 유형을 설명하는 레이블
  - 속성 컬렉션 (Key-Value 쌍)

트리플 저장소

- 주어 서술어 목적어처럼 매우 간단한 세 부분 구문 형식으로 저장. 주어는 그래프의 정점과 동등
- 목적어는 속성이거나 두 정점을 잇는 간선의 레이블

데이터 모델 정리

→ 데이터를 트리형태로 표현해보려고 하다가 실패. 관계형 데이터베이스 등장

→ 관계형 모델의 한계를 극복하기 위해 NoSQL 등장

→ NoSQL은 문서 데이터 모델과 그래프형 데이터 모델로 나뉨

→ 결국 관계형, 문서형, 그래프형에서 적절한 것을 사용해야함

03. 저장소와 검색

작업부하(workload)에 따라 적절한 스토리지 엔진을 선택할 수 있어야함

스토리지 엔진

- 로그 구조 (log-structured)
- 페이지 지향 (page-oriented)

로그 구조 (log-structured)

여기서 로그란 애플리케이션 로그의 의미가 아니고 `연속된 추가전용 레코드`이다.

- 데이터 로그가 쌓이면 색인(indexing)이 필요하다.
- 색인은 쓰기연산과 읽기연산 성능의 트레이드 오프
- 로그(추가전용 레코드)가 계속 쌓이면 세그먼트 컴팩션 과정이 필요하다.
  (중복키 제거해서 최신값만 유지하여 새로운 세그먼트를 생성함)

추가전용 레코드(로그) 왜씀?

- 추가와 세그먼트 병합은 순차쓰기여서 무작위 쓰기보다 훨씬 빠름
- 세그먼트 파일이 추가전용이거나 불변이면 동시성과 고장 복구는 매우 간단함
- 오래된 세그먼트 병합은 시간이 진마에 따라 조각화되는 데이터 파일 문제를 피할 수 있음

해시테이블 색인의 제약사항

- 인메모리에 유지해야해서 키가 크면 안됨
- Range Query 불가능함 (해시니까)

SS(Sorted String) 테이블

- 세그먼트에서 키-값 쌍이 키로 정렬되어있음
- 각 키는 병합된 세그먼트 안에서 하나만 있어야함 (오래된 세그먼트에는 중복 있을 수 있음)

LSM 트리

- 정렬된 파일 병합과 컴팩션 원리를 기반으로 하는 저장소
- 백그라운드에서 연쇄적으로 SS테이블을 지속적으로 병합

B 트리

- 루트(Root) ~ 리프 페이지(Leaf Page)
- B 트리의 한페이지에서 하위 페이지를 참조하는 수를 분기 계수(branching facotr)라고 부른다
- n개의 키를 가진 B 트리는 깊이가 항상 `O(log n)` 이다

- B 트리는 데이터를 덮어씀. 이 때 장애시 손실을 방지하기 위해 redo log라고 하는 로그 구조 저장소를 사용함 (먼저 로그에 쌓고 트리에 저장, 이를 쓰기전 로그 `wrtie-ahead log` 라고함)

LSM트리와 B트리 비교

- 일반적으로 LSM트리가 B트리보다 쓰기처리량이 좋다고 알려져 있음 읽기 성능은 B 트리가 더 좋다고 한다
- LSM트리의 압축률이 B트리보다 좋음
- LSM트리는 같은 키가 여러 세그먼트에 존재할 수 있음
- SS 테이블 컴팩션도 결국 쓰기처리량 높아지면 장애가 일어날 수 있음

색인안에 값 저장하기

- 색인과 색인된 로우를 같이 저장한다 → Clustered Index
- 색인에 로우에 대한 참조만 저장한다(이 참조를 힙파일,Heap file이라고함) → None Clustered Index
- 데이터 일부를 색인에 저장한다 → Covering Index → 색인이 질의를 커버(Cover) 했다.
  → Index with included column이라고도함

다중 컬럼 색인

→ 결합 색인(concatenated index)라고 함

전문 검색과 퍼지(fuzzy) 색인

→ 퍼지 색인은 글자 유사도 검색 (1글자 틀려도되는 이런)

→ 루씬에서는 레벤슈타인 오토마톤 형태의 인메모리 색인을 쓴다.. 이게 뭔소리?

인메모리 데이터베이스

→ 모든 것을 메모리에 저장한다

→ 비동기로 디스크에 쓰기하면서 약한 지속성(durability)을 제공하기도함 e.g. Redis Persistence 옵션

→ 비휘발성 메모리 주목하자

OLTP와 OLAP

- OLTP(트랜잭션 처리 시스템) : Online Transaction Processing
- OLAP(분석 시스템) : Online Analytic Processing → Busingess Intelligenece

OLAP 목적으로 데이터 웨어하우스(Data Warehouse)가 발전함

데이터 웨어하우스

일반적으로 분석을 위한 SQL을 사용하기 위해 관계형 모델을 사용함.

- 분석용 스키마 : 별모양 스키마(star schema), 차원모델링(dimesional modeling) → 분석하려고 만든 스키마, 극한의 역정규화 테이블 혹은 필요한 데이터들의 PK들을 모두 모아놓은 테이블로 구성할 수 있을 것 같음
- 칼럼지향저장소 : Row기반이 아니고 Column(열)의 Aggregation 저장소를 사용해보자..
- 칼럼 압축 → 비트맵 부호화 → 극한의 칼럼 지향.. → AND OR로 벡터화 처리(vectorized processing)도 가능함

04. 부호화와 발전

아래 것들을 알아보자!

데이터 부호화를 위한 형식

- JSON, XML, 프로토콜 버퍼(Protocol Buffers), 스리프트(Thrift), 아브로(Avro)

위를 이용한 다양한 통신

- REST (Representational State Transfer, REST)
- 원격 프로시저 호출(remote procedure call)

중요한 것

- 하위 호환성
- 상위 호환성

데이터 부호화 형식

- 메모리에서 데이터를 사용하기 위해서는 포인터(Pointer, 주소접근)을 기반으로한 Object, Struct, list, array, hash table, tree 등으로 이용함
- 하지만 이것을 네트워크를 통해 데이터를 보내려면 포인터(주소)는 사용할 수 없기 때문에 부호화해서 보내야함

- 부호화 = 직렬화 = 마샬링
- 복호화 = 파싱 = 역직렬화 = 언마샬링

언어별 부호화 방식

Java의 Serializable 등의 언어에서 부호화를 지원하는 경우가 있지만 안쓰는 편이 좋다.

JSON, XML, 이진부호화

- JSON, XML 사람이 읽기 좋음 호환성 짱짱 만들기도 편함 ~ 하지만 바이너리를 관리할 수 없기 때문에 Base64 등을 이용하기도함 솔직히 성능적인 면에서는 떨어짐
- JSON 이진 부호화는 JSON Message Pack 등을 이용해 Binary로 부호화할 수 있음 하지만 사람의 가독성을 해칠만큼 유용한지는 모르겠음

스리프트(Thrift)와 프로토콜 버퍼(Protocol Buffer)

- 같은 원리에서 출발한 부호화 형식임
- 스키마 정의가 필요함

스리프트(Thirft)

- 앞서 본 이진 부호화처럼 Binary로 데이터를 만듬
- 컴팩트 프로토콜과 바이너리 프로토콜로 나뉨
- 그런데 JSON 이진부호화처럼 필드명을 사용하지 않음 Field Tag를 이용함 (스키마, IDL에서 이 필드태그 값을 미리 정의함)
- 그래서 JSON 이진부호화보다 데이터 작아짐
- 가변길이 정수 사용함

프로토콜 버퍼

- 스리프트의 컴팩트 프로토콜과 비슷함
- 가변길이 정수 사용함

필드태그와 스키마의 발전

스키마는 필연적으로 시간이 지남에 발전한다. 이를 스키마 발전(schema evolution)이라 함. 쉽게 말하면 API 스펙(필드나 값)이 추가되는 것

- 스리프트와 프로토콜 버퍼는 데이터를 필드태그로 인식함 (필드태그=값의 키)
- 새로운 필드태그 생김 → 기존 필드태그 값은 동일함 → 옛날 코드에서 새로운 데이터 스키마를 읽을 수 있음 → 새로 생긴 필드태그는 무시하기 때문
- 새로운 필드태그 생김 → 기존 필드태그 값은 동일함 → 하위호환성 계속 유지됨
- required는 지우거나 하면 안됨. 바로 깨짐 optional만 사용가능 (그래서 proto3에서 optional default 되었나보다)

데이터타입과 스키마 발전

- 데이터타입 바꾸면 안됨. 예전 코드에서 데이터를 중간까지 읽거나할 수 있음
- 프로토콜 버퍼의 repeated는 중간에 추가해도됨 optional에서 0이나 1까지만 읽음 (굿)

아브로

- 아브로는 프로토콜버퍼와 스리프트보다 짧은 이진부호화 형식을 제공함
- 데이터에 필드태그를 넣지 않기 때문
- 그래서 읽기스키마와 쓰기스키마가 동일해야함
- 아브로는 쓰기스키마와 읽기스키마를 모두 보고 데이터 온거를 복호화해서 가져옴. 이때 필드네임으로 매핑함
- 아브로는 상위호환성, 하위호환성을 지키기 위해 기본 값(default value)와 유니온 타입(union, nullable)이 있다.

쓰기 스키마가 뭐임? 어떻게 두 버전의 스키마를 가질 수 있는가

어떻게 하냐면 아래처럼 할 수 있음

- 많은 레코드가 있는 대용량 파일에서는 파일 시작부분에 쓰기스키마 그냥 포함시킨다 (데이터의 첫부분에 쓰기스키마 그냥 포함시킴, 데이터가 워낙에 크기 때문에 가능함)
- 개별적으로 기록된 레코드를 가진 데이터베이스, 버전별의 스키마를 미리 가지고있고 가져와서 쓴다
- 네트워크 연결을 통해 버전을 공유하고 사전에 합의하고 통신한다 (HTTP의 Accept, TLS의 암호화 스위트처럼..) 아브로 RPC가 이처럼 동작한다고 한다

동적 생성 스키마

아브로는 필드태그가 없다.. 이는 아브로가 스키마를 동적으로 생성한다는 의미이다.

스키마의 장점

- 스키마가 있어서 정적타입 언어에서 효율적으로 데이터타입을 사용할 수 있음 → 인메모리에서 정적언어의 데이터타입이 효율적으로 동작함
- 스키마발전도 지원 잘됨 스키마리스(schemaless) 또는 읽기 스키마(schema-on-read)급으로 호환성 좋음

- 안쓸 이유가 없음.

Data Flow mode (프로세스간 데이터를 전달하는 방법을 알아보자)

데이터베이스를 통한 데이터플로

- 데이터베이스에 부호화하여 저장하면 미래의 애플리케이션이 이걸 읽을 수 있다 (하위 호환성)
- 데이터베이스에 부호화하여 저장하면 과거의 애플리케이션이 이걸 읽을 수 있다 (상위 호환성)
- 데이터가 코드보다 더 오래 산다 (data outlives code)

보관 저장소

백업 목적이나 데이터 웨어하우스로 적재하기 위해 데이터베이스 스냅숏을 수시로 만들 떄 일관적으로 부화하자

- 데이터베이스는 항상 최신버전의 스키마로 부화할 것이다
- 이 때 칼럼지향형식으로 칼럼압축도 가능

서비스를 통한 데이터플로 (REST와 RPC)

- HTTP는 HTTP처럼. Restful하게.
- RPC 해보자.. ! 정적언어에서는 유용하다. 그런데 REST보다 유연하지 못하다 (SOAP, gRPC등)

원격 프로시저 호출(RPC)의 문제

원격 프로시저 호출은 네트워크 서비스 요청을 같은 프로세스 안에서 로컬 함수 호출하는 것과 동일하게 사용 가능하게 해준다

로컬 함수 호출과 비교해보자

- 네트워크 Timeout을 통해 응답이 안올 수 있음

- 요청이 처리되고 응답이 안올 수 있음 이 때 재호출되고 멱등성을 위한 처리 (중복 제거) 등이 안되면 데이터가 잘못될 수 있음
- 네트워크 지연에 따라 매번 호출 처리 시간이 다를 수 있음
- 로컬함수호출 데이터 전송보다 메모리 효율적으로 사용하지 못할 수 있음 (하지만 정적 언어라면 부호화 방식에 따라 효율적으로 사용 가능할 것 이라고 생각함)

RPC의 데이터 부호화와 RPC의 발전(스키마 발전

- 결국 사용하는 RPC 프로토콜의 데이터부호화 프로토콜로부터 호환성을 상속받음
- gRPC는 프로토콜 버퍼를 사용하기 때문에 프로토콜 버퍼의 호환성 규칙에 따라 발전할 수 있다

메시지 전달 Data Flow

비동기 메시지 전달 시스템(asynchronous message-passing system)에서 데이터 통신하는거 알아보자

- 메시지 브로커(Message Brocker)는 데이터를 바로 네트워크를 통해 보내지 않고 잠시 저장한다
- 메시지 지향 미들웨어(message-oriented middleware)도 메시지 브로커와 비슷한 역할임

RPC와 비교했을때 다음과 같은 장점이 있음

- 버퍼 역할이 가능. 바로 처리하지 않아도 잠시 가지고 있을 수 있다 (Queue)
- 죽었던 프로세스에 메시지를 다시 전달할 수 있어서 메시지 유실을 방지할 수 있다
- 송신자(Sender)가 수신자의 IP 주소나 포트 번호를 알 필요가 없음
- 하나의 메시지를 여러 수신자로 전송할 수 있음(Pub-Sub 등)
- 논리적으로 송신자와 수신자는 분리됨 (loose coupling)

주의할건 통신패턴이 비동기임

메시지 브로커

- RabbitMQ, AcctiveMQ(Amazon), 아파치 카프카(Apache Kafka) 등이 있다.
- 프로세스가 메시지를 이름이 지정된 큐나 토픽으로 전송함
- 브로커는 해당 큐나 토픽 하나 이상의 소비자(consumer) 도는 구독자(subscriber)에게 메시지를 전달함
- 동일한 토픽에 여러 생산자(producer)가 있을 수 있음

메시지 브로커는 특정 데이터 모델을 강요하지 않음. 원하는 데이터 부호화 형식으로 부호화해서 보내면 됨

분산 액터 프레임워크라는 것도 있음

- 액터 모델은 단일 프로세스 안에서 동시성을 위한 프로그래밍 모델임
- 로직이 액터에 캡슐화되고 각 액터는 격리된 프로세스에서 각자 실행됨

결론

데이터 부호화 형식 신중하게 결정하자. 하위호환성 상위호환성 잘 생각하자. 프로세스간 어떻게 데이터 부호화해서 통신할건지 생각해보자. 서비스의 발전이 어떻게 될지 생각해보자.