SIGMOD '11
'Apache Hadoop Goes Realtime at Facebook' Overview

2011.07.06

Data Analysis Platform Team
Minwoo Kim(김민우)
michael.kim@nexr.com

kt

NEXR

ACM SIGs

- ACM(Association for Computing Machinery)
  - 1947년 설립
  - 컴퓨터 분야 학회 연합체
  - 약 83,000명의 회원, 170여개의 지역조직, 500여개 대학조직
- 특화분야그룹 (Special Interest Groups, SIGs)
  - 구체적인 연구 분야마다 SIG(Special Interest Group) 분과회 구성
  - 약 30여개
  - SIGCOMM, SIGGRAPH, SIGMOD, SIGOPS. SIGCHI, ...
- SIGMOD (Special Interest Group on Management of Data)
  - 데이터 베이스관련 역사와 전통이 있는 최고 수준의 학회 중 하나
- 그밖에 데이터베이스 관련 학회
  - VLDB(Very Large Data Bases)
  - ACM POD(ACM SIGMOD Conf On Principles of DB Systems)
  - IEEE ICDE (International Conference on Data Engineering)
  - ...

2011 ACM SIGMOD/PODS Conference

- SIGMOD 2011
  - Athens, Greece, June 12-16, 2011
- Research Papers
  - http://www.sigmod2011.org/research_list.shtml
- Industry Papers
  - Oracle Database Filesystem, Oracle
  - Noba: Continuous Pig/Hadoop Workflows, Yahoo!
  - A Batch of PNUTS: Experiences Connecting Cloud Batch and Serving Systems, Yahoo!
  - Efficient Processing of Data Warehousing Queries in a Split Execution Environment, Hadapt
  - Automated Partitioning Design in Parallel Database Systems, Microsoft
  - Turbocharging DBMS Buffer Pool Using SSDs, Microsoft
  - SQL Server Column Store Indexes, Microsoft
  - A Hadoop Based Distributed Loading Approach to Parallel Data Warehouses, Teradata
  - LCI: A Social Channel Analysis Platform for Live Customer Intelligence, HP
  - Online Reorganization in Read Optimized MMDBS, IBM
  - An Analytic Data Engine for Visualization in Tableau, Tableau
  - Bistro Data Feed Management System, AT&T

Facebook Papers

- Facebook Data Infrastructure Team Papers
  - FATE and DESTINI: a framework for cloud recovery testing, NSDI '11 (proceeding)
  - Apache hadoop goes realtime at Facebook, SIGMOD' 11
  - YSmart: Yet Another SQL-to-MapReduce Translator, ICDE, 11
  - RCFile: A Fast and Space-efficient Data Placement Structure in MapReduce-based Warehouse Systems, ICDE' 11 (2)
  - Finding a needle in Haystack: facebook's photo storage, OSDI' 10 (7)
  - Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling, EuroSys '10
  - Data warehousing and analytics infrastructure at facebook, SIGMOD' 10 (12)
  - Hive: a warehousing solution over a map-reduce framework, VLDB 09 (85)

- Etc.
  - Facebook immune system, SNS' 11

Apache Hadoop Goes Realtime at Facebook

- http://portal.acm.org/citation.cfm?id=1989438

- General Terms
  - Management, Measurement, Performance, Distributed Systems, Design, Reliability, Languages

- Keywords
  - Data, Scalability, resource sharing, distributed file system, Hadoop, Hive, Hbase, Facebook, Scribe, Log aggregation, distributed System

- Authors
  - Dhruba Borthakur , Kannan Muthukkaruppan , Karthik Ranganathan
  - Samuel Rash , Joydeep Sen Sarma , Nicolas Spiegelberg
  - Dmytro Molkov , Rodrigo Schmidt , Jonathan Gray
  - Hairong Kuang , Aravind Menon , Amitanand Aiye

Abstract

- 최근 Deploy한 페이스북 메시지 서비스
  - 처음으로 Hadoop 플랫폼 위에 올라간 User-facing 애플리케이션
  - 하루에 수십억 건의 메시지를 생산

- 페이스북이 Hadoop과 Hbase을 선택한 이유
- 새로운 애플리케이션 요구사항 대한 논의
  - 일관성, 가용성, 파티션 내구성(Tolerance), 데이터 모델, 확장성 등
- Hadoop을 실시간 시스템으로서 효율적으로 만드는 강화 방안을 설명
  - 페이스북과 다른 많은 웹 스케일 회사에서 사용되는 Shared MySQL DB 체계보다 어떤 중요한 정점을 가지는지 Tradeoff는 무엇인지
  - 설계 선택의 동기
  - 직면한 Day-to-Day 작업의 도전과제들
  - 미래 수용능력
  - 개발에서 기능개선 등

- 전통적인 Shared RDBMS 개발에서 Hadoop기반 솔루션을 고려중인 다른 회사들을 위한 관찰 결과를 제시

Introduction

- Apache Hadooop
  - Top-level 아파치 프로젝트
  - Google의 GFS, Mapreduce에서 영감을 얻음
- Hadoop Ecosystem
  - Google의 BigTable에서 영감을 얻은 HBase
  - Hadoop 위에 올라가는 DW인 Hive
  - 분산 시스템을 위한 코디네이션 서비스 Zookeeper 등을 포함
- At Facebook
  - 전통적으로 Hadoop은 대용량 데이터셋의 분석과 저장을 위해서 Hive 사용
  - 이런 분석의 대부분 오프라인 배치 작업
  - 효율성과 전송량(Throughput)을 최대로 하는 것을 강조
  - 배치의 작업부하(Workload)는 전형적으로 디스크에서 순차적으로 대용량 데이터를 읽고 쓰는 것
  - HDFS에 Low-Latency Random Access를 제공하 작업부하 성능은 덜 강조
  - 하지만 최근 페이스북에서 떠오르는 차세대 애플리케이션들은 매우 높은 쓰기 전송량, 싸고, 유연한 저장소가 요구됨

NEXR

Introduction

- 반면에 기존 MySQL 클러스터
  - 높은 Uptime과 좋은 부하 분산을 유지하면서 급격하게 확장하는 것은 어렵다
  - 관리에서 비교적 높은 Overhead
  - 전형적으로 더 비싼 하드웨어를 사용

- 경험적으로 HDFS의 신뢰성과 확장성에 대한 높은 신뢰가 있기 때문에 Hadoop과 HBase 조사

- 첫 번째 애플리케이션 군
  - 순차적(Sequential)이 아닌 실시간으로 Concurrent하게 HDFS에 저장된 데이터에 댜량의 스트림 읽기 Access가 요구
  - 실시간 분석과 MySQL 백업이 이런 범주에 들어가는 두 가지 주요 애플리케이션
  - 이런 애플리케이션들에 데이터를 생성하고 저장하는 시스템은 Scribe
  - 페이스북에서 이런 오픈소스 분산 로그 집계 서비스는 널리 사용 됨
  - 이전에는 Scribe에서 생성된 데이터는 비싸고 관리하기 어려운 NFS 서버에 저장
  - 우리는 HDFS가 높은 성능의 low-latency 파일 시스템이 되도록 향상시켜 왔다
  - 그 결과 비싼 파일 서버들의 사용을 줄임

Introduction

- 두 번째 Non-Mapreduce Hadoop 애플리케이션 세대
  - 빠른 랜덤 룩업(look-up)을 하기 위해서 급격히 증가하는 데이터셋을 동적으로 인덱스 할 수 있어야 함
  - 페이스북 메시지 경우에는 런칭 후에 바로 즉시 5억명 이상의 사람들이 사용할 수 있어야 함
  - 엄격한 Uptime 요구사항을 가지면서 페타바이트 데이터로 확장하는 것이 필요
  - 우리는 이 프로젝트에서 HBase를 사용하는 것을 결정

작업부하 종류 (Workload Types)

- MySQL 기반 아키텍처에서 Hadoop기반 플랫폼으로 옮기는 것을 결정하기 전에 몇 가지 특정 애플리케이션을 살펴 보겠다
- 다음 Use Case는 매우 높은 쓰기 전송량, 대용량 데이터 셋, 예측할 수 없는 성장, 등의 패턴들 때문에 Sharded RDBMS 환경에서 확장하는 것이 매우 어려운 작업부하를 가진다

- Facebook Messaging
- Facebook Insights
- Facebook Metrics System (ODS)

작업부하 종류 : Facebook Message

- 최신 세대의 메시지 서비스는 기존 메시지와 E-mail, Chat, SMS를 조합한다.
- 모든 메시지는 영속화되어야 한다
- 새로운 쓰레드 모델에 의해 각각 유저에 대해서 나눠져서 저장되어야 한다
- 각각의 사용자는 하나의 데이터센터에 끈적하게(sticky) 붙어 있어야 한다

- 높은 쓰기 처리량(High Write Throughput)
  - 매일 몇 백만의 메세지와 수억 건의 인스턴스 메시지의 기존의 증가 속도를 볼 때, 섭취되는 데이터의 볼륨은 매우 커야 한다.
  - 이런 데이터는 하루 하루 계속해서 증가한다.
- 큰 테이블 (Large Tables)
  - 메시지는 사용자가 명시적으로 수행하지 않는 한 지워지지 않음
  - 각각의 메일 박스는 무한히 증가
  - 전형적인 메시징 애플리케이션의 모습
  - 메시지는 최근에 쓰여진 것만 몇 번 읽고 아주 가끔 다시 본다.
  - 대부분 데이터는 읽혀지지 않지만 최소 지연시간으로 언제든지 사용 가능 해야 된다
  - 이런 요구사항들은 아카이빙을 매우 어렵게 한다.
  - 사용자의 수천 개의 메시지를 모두 저장한다는 것은 사상 최대 쓰레드와 메시지들이 사용자별로 인덱스된 데이터베이스 스키마를 가진다는 것을 의미한다.
  - 이런 종류의 랜덤 쓰기 작업부하에서 MySQL의 쓰기 성능은 전형적으로 테이블의 ROW 수가 증가 할수록 저하된다.

작업부하 종류 : Facebook Message

- 데이터 이주(Data Migration)
  - 새로운 메시징 제품에 또 하나의 주요 어려운 측면은 새로운 데이터 모델
  - 기존 레거시 시스템에 있는 사용자의 모든 메시지는 새로운 쓰레드 패러다임을 위해서 조인되고 조작되고 나서 이주(migration)된다.
  - Large Scan, Random Access, Fast Bulk Import의 능력은 새로운 시스템으로 사용자들을 이주시키는데 드는 시간을 줄이는데 도움이 된다.

작업부하 종류 : Facebook Insights

- 웹사이트와 소셜 플러그인, 페이스북 페이지, 페이스북 광고 사이에 페이스북 활동과 연관된 실시간 분석을 개발자와 웹사이트 소유자에게 제공
- Impression, 클릭율, 웹사이트 방문자 수와 같은 활동을 다룬다
- 이런 통계는 회사와 블로거 모두 어떻게 사람들이 컨텐츠에 관심을 가지는지에 대한 통찰을 얻는 데 도움이 된다. 그래서 이를 기반으로 서비스를 최적화 할 수 있다.
- 우리는 Hadoop과 Hive DW를 통해 오프라인 방식으로 도메인과 URL 분석은 이미 정기적으로 수행하고 있다
- 그러나 활동이 일어난 몇 시간 후에야 사용할 수 있어서 열악한 사용자 경험을 제공한다

- 실시간 분석 (Realtime Analytics)
  - Insights팀은 이전에 몇 시간 걸리는 것을 사용자 행동 이후 수 초 내로 사용자가 이용할 수 있는 것을 원했다
  - 이런 이벤트들을 영속화하고 집계하고 처리하기 위한 시스템 뿐 아니라 사용자 활동을 위한 대용량, 비동기 큐잉 시스템이 요구된다.
  - 이들 시스템은 모두 내고장성(Fault-Tolerant)이 요구되고 초 당 백만 건을 지원할 수 있어야 한다

- 높은 인크리먼트 처리량 (High Throughput Increments)
  - 기존 Insight 기능들을 지원하려면 시간과 인구통계학 기반의 집계가 필요하다.
  - 이들 집계들은 최신의 상태를 유지해야 한다
  - 따라서 수많은 숫자 카운터 (numeric counts)를 통해서 동시에 처리되어야 한다
  - 수백만의 유일한 집계와 수십억 건의 사건들이 있다면, 이에 대항하는 아주 많은 오퍼레이션

작업부하 종류 : Facebook Metrics System(ODS)

  - 페이스북의 모든 하드웨어와 소프트웨어는 ODS(Operations Data Store)라고 불리는 메트릭 수집 시스템에 통계 정보를 제공한다
  - 예를 들면, 우리는 단일 서버, 또는 다수의 서버 층의 CPU 사용량을 수집할지도 모른다
  - 또는 HBase 클러스터에 쓰기 작업의 수를 추적할지도 모른다.
  - 각각 노드 또는 노드의 그룹에 대해서 수백, 수천개의 메트릭을 추적한다
  - 엔지니어는 다양한 입도로 시간에 대한 그래프를 요구 한다
  - 이 애플리케이션은 쓰기 전송량에 대해서 매우 Heavy 요구사항을 가진다.
  - 더 고통은 기존 MySQL기반 시스템에서 데이터 Resharding과 Time roll-ups 분석을 위해 테이블 스캔하는 능력에 있다.
- 자동적인 Sharding
  - 대량의 인덱스된 타임시리즈 쓰기들과 예측 불가능한 증가 패턴은 하나의 Sharded MySQL 구성과 화해하는 것은 어렵다
  - 예를 들면 어떤 제품은 긴 시간 간격으로 열 개의 메트릭만 수집될 수 있다. 그러나 대규모 릴리즈 또는 제품 출시에 따라서, 동일 제품이 수천 개의 메트릭들을 생산할 수도 있다.
  - 기존 시스템에서는 단일 MySQL 서버는 갑자기 감당할 수 있는 것보다 더 많은 부하를 다뤄야 할지도 모른다.
  - 이 경우 수동으로 단일 서버에서 다중 서버로 데이터를 Re-Shard하는 데 집중을 해야 한다
- 최근 데이터와 테이블 스캔의 빠른 읽기
  - 메트릭 시스템으로 가는 대량의 읽기의 대부분은 최근 Raw 데이터에 관한 것이다.
  - 그러나 모든 역사적인 데이터 또한 사용 가능해야 한다.

Why Hadoop and HBase

- 이전에 언급한 작업부하로부터 저장 시스템의 요구사항 요약
  - 탄력성(Elasticity)
  - 높은 쓰기 처리량(High write throughput)
  - 한 데이터 센터 안에서 효율적인 low-latency 강한 일관성 시멘틱
  - 효율적인 랜덤 디스크 읽기
  - 고가용성과 재난 극복(High Availability and Disaster Recovery)
  - 내고장성 (Fault Isolation)
  - 원자적 읽기-수정-쓰기 Primitives
  - 범위 스캔

- 요구사항은 아니지만 중요한 포인트들
  - 단일 데이터 센터 안에서 내트웍 파티션의 내구성
  - 개별 데이터 센터 정지하는 경우의 제로 다운타임
  - Active-active serving capability across different data centers

Why Hadoop and Hbase : 새로운 저장 시스템 요구사항

- 탄력성 (Elasticity)
  - 우리는 저장 시스템이 최소한의 오버헤드와 정지시간 없이 신속하게 용량 추가가 가능해야 한다
  - 시스템은 자동적으로 새 하드웨어의 활용(utilization)과 부하의 균형을 맞춰야 한다.
- 높은 쓰기 전송량 (High Write Throughput)
  - 우리의 대부분 애플리케이션들은 거대한 데이터 셋을 저장하고(선택적으로 인덱스도) 높은 집계 쓰기 처리량이 요구된다
- 하나의 데이터센터 안에서 효율적인 low-latency 강한 일관성 시멘틱
  - 한 데이터 센터 안에서 강한 일관성(consistency)을 요구하는 메시지 같은 중요한 애플리케이션이 있다
  - 이런 요구사항은 대부분 사용자 기대로부터 직접적으로 발생한다
  - 예를 들어 '읽지 않은' 메시지의 수가 홈페이지에 표시된다.
  - Inbox 페이지 뷰에서 보여지는 메시지는 사용자 서로에 대해서 일치해야 한다
  - 전 세계적인 강한 일관성을 지원하는 분산 시스템은 실제적으로는 불가능하다
  - 최소한 하나의 데이터 센터 안에서 강한 일관성을 제공할 수 있는 시스템은 좋은 사용자 경험을 제공할 수 있는 것이 가능할 지도 모른다
- 디스크로부터 효율적인 랜덤 읽기
  - 애플리케이션 수준의 캐시(임베디드 또는 memcached) 사용이 넓게 퍼졌음에도 불구하고페이스북 규모에서는 수많은 접근이 캐시에서 미스가 나고 백엔드 시스템에서 히트 한다
  - 디스크 랜덤 읽기에서는 MySQL은 아주 효율적인 수행을 한다.

Why Hadoop and Hbase : 새로운 저장 시스템 요구사항

- 고가용성과 재난 극복 (High Availability and Disaster Recovery)
  - 우리는 계획되거나 계획되지 않은 사건에 대해서 매우 높은 Uptime으로 사용자에게 서비스를 제공하는 것이 필요하다.
    - 예) 소프트웨어 업그레이드, 하드웨어 추가, 하드웨어 컴포넌트 실패
  - 우리는 최소한의 데이터 손실로 데이터 센터의 실패를 견뎌낼 수 있어야 한다.
  - 데이터를 또 다른 데이터 센터 밖으로 적절한 시간 안에 제공할 수 있어야 한다
- 실패 격리 (Fault Isolation)
  - MySQL 데이터 베이스 Large Farm을 운영해 온 우리의 오랜 경험으로 실패 고립이 중요하다는 것을 알게 되었다.
  - 개별 데이터베이스들은 내릴 수(go down) 있어야 하고, 그러나 사용자의 적은 수만 이런 사건에 대해서 영향을 받아야 한다.
  - Hadoop을 사용하는 우리의 Warehouse에서, 개별 디스크의 고장은 데이터의 작은 부분에만 영향을 미치고, 시스템은 빠르게 그런 고장으로부터 회복된다
- 원자적 읽기-수정-쓰기 Primitives
  - Lockless 동시성 애플리케이션을 만들 때 원자적 Increment와 compare-and-swap API는 매우 유용한다.
  - 기반(underlying) 저장 시스템은 이런 기능을 반드시 가져야 한다
- 범위 스캔 (Range Scan)
  - 몇몇 애플리케이션은 효율적으로 특정 범위에 대한 Rrow의 집합을 가져오는 것이 요구된다.
  - 예) 특정 광고주에 대해서 지난 24시간 동안 시간 단위로 Impression의 수, 특정 사용

Why Hadoop and Hbase : 요구사항이 아니지만 중요한 포인트들

- 단일 데이터센터 안에서 네트웍 파티션 내구성
  - 원래, 다른 시스템 컴포넌트들은 종종 집중화 된다.
  - 예를 들면, MySQL 서버들은 모두 몇 개의 Rack들 안에 위치하게 된다.
  - 하나의 데이터 센터 안에 네트웍 파티션은 서비스를 제공하는 능력 중에서 주요한 손실을 발생시킬 수 있다.
  - 따라서 모든 노력은 높은 다중화(Redundant) 네트웍 설계를 통해서 하드웨어에서 이런 이벤트의 가능성을 제거하는데 집중해야 한다.

- 단일 데이터 센터 정지 시 제로 다운타임
  - 우리의 경험에서 이런 데이터 센터의 정지는 매우 드물지만 불가능하지는 않다.
  - 이상적인 세상의 시스템 설계를 선택하는 것보다 수용 가능한 타협안을 선택해야 한다. 이런 이벤트의 발생 비율을 낮추는 것이 타협안 중 하나가 될 수 있다.

- Active-active serving capability across different data centers
  - 미리 언급했듯이, 우리는 사용자 데이터가 다른 데이터 센터들 사이에서 제휴될 수 있다고 가정하는 것이 자연스럽다. (이상적으로 사용자 지역성에 기반하는)
  - 사용자와 데이터 지역성이 맞지 않을 때의 지연시간은 사용자와 가까운 애플리케이션 캐시를 이용하여 가릴 수(masked) 있다.

Why Hadoop and HBase

- 페이스북은 생산 경험이 있는 시스템과 사내 전문가들이 훨씬 선호된다.
- 오픈 소스 프로젝트를 고려 할 때 커뮤니티의 힘은 중요한 요소이다
- 시스템을 만들고 유지하는 데 드는 엔지니어링 투자 수준을 고려할 때도 넓게 적용 가능한 솔루션을 선택하는 것이 이치에 맞다
- 우리는 상당한 연구와 실험 후에 차세대 애플리케이션을 위한 기반 저장 기술로서 Hadoop과 Hbase를 선택했다
- 사내 엔지니어링을 통해서 Hbase의 부족한 기능을 해결할 수 있다는 우리의 자신감 또한 평가에 반영되었다

Why Hadoop and HBase

- HBase는 이미 높은 일관성,높은 쓰기 전송량의 key-value 저장소를 제공한다.
- HDFS의 네임노드는 단일실패지점이다. 그러나 우리는 HDFS 팀은 적절한 기간 안에 고가용성 NameNode를 만들 수 있는 자신감이 있다. 또 한 이 것은 우리의 Warehouse 작업에도 도움이 된다.
- Hive/Hadoop 웨어하우스 운용 경험에서, 우리는 HDFS가 디스크 서브시스템의 고장를 격리하고 견뎌내는데 우수하다는 것을 알고 있다.
- 전체 HBase/HDFS 클러스터의 정지는 결함 격리의 목표와 충돌하는 시나리오다.
- 그러나 더 작은 HBase 클러스터들에게 데이터를 저장하므로서 상당히 완화될 수 있다.
- 다른 복제 프로젝트들, 사내인력과 HBase 커뮤니티 모두, 장애 극복을 달성하기 위한 약속된 길을 가고 있다.
- HBase 는 masively 확장가능하고 랜덤과 스트리밍 읽기 뿐 아니라 빠른 랜덤 쓰기를 전달한다. row 수준의 원자성(Atomicity) 보장한다. 그러나 cross-row 트랜젝션 지원은 없다.
- 데이터 모델 perspective로부터, 컬럼 오리엔테이션은 단일 테이블 안에 데이터를 저장하고 수십억 건의 인덱스 값을 생성하고 데이터를 저장하는데 극한의 유연성(flexibility)을 제공한다. HBase는 Write-intensive한 작업부하에 이상적이다. 많은양의 데이터, 인덱스와 빠른 스케일 아웃의 유연성을 유지하는데 필요하다.

Realtime HDFS

- 본래 HDFS는 배치 시스템과 확장성과 스트리밍 성능이 가장 중요한 오프라인 MapReduce 애플리케이션을 지원하기 위한 파일 시스템으로 설계 되었다.
- 우리는 HDFS 사용의 장점을 경험했다. 선형 확장성과 내고장성(Fault-Tolerance)은 엔터프라이즈 사이에 많은 비용 절감이 결과로 나타났다.
- HDFS의 실시간과 온라인 사용은 새로운 요구사항을 밀고 나아갔고 이제는 다목적의 low-latency 파일시스템으로서 사용된다
- 우리는 새로운 애플리케이션을 지원하기 위한 HDFS 핵심 변경사항 몇 가지를 설명한다.

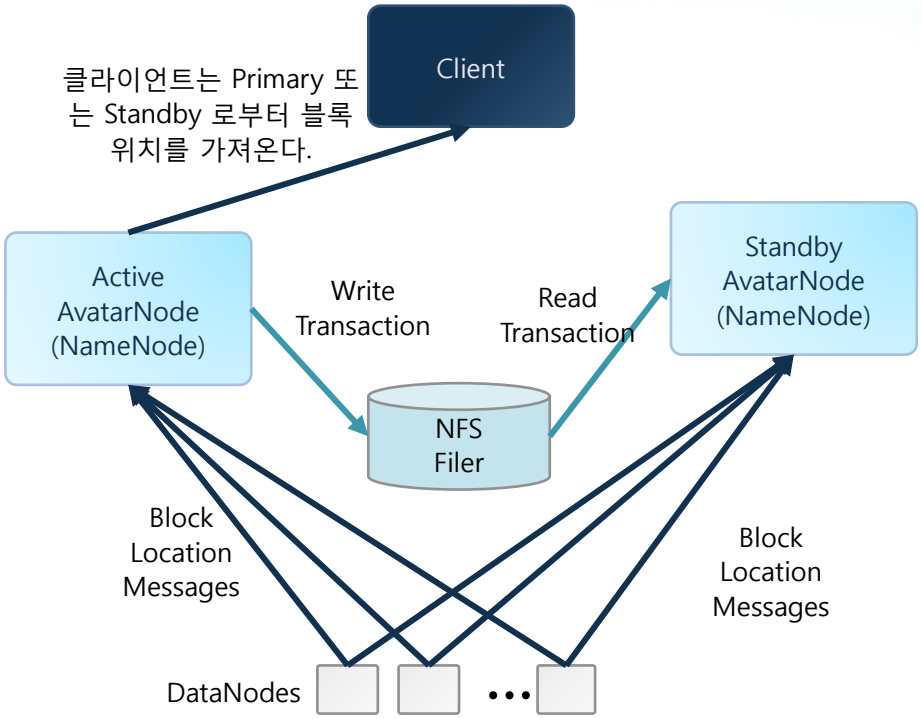
고가용성 (High Availability)

- HDFS의 설계는 하나의 마스터(NameNode)를 가진다.
- 마스터가 고장 날 때는 언제나 HDFS 클러스터는 NameNode가 백업될 때까지 안전하지 않다.
- 이것은 단일실패지점(SPOF) 이다.
- 이것은 사람들이 Uptime 요구사항이 하루 24시 일주일 내내인 애플리케이션을 HDFS로 deploy하는 것을 주저하게 만드는 이유이다
- 우리의 경험에서는, 클러스터 정지시간의 주된 이유는 HDFS 서버 소프트웨어를 새로운 소프트웨어 업그레이드하는 것이다
- 생산 클러스터로 Deploy되기 전에 하드웨어를 전적으로 신뢰할 수 없기 때문에, 소프트웨어는 잘 테스트되어야 한다
- 우리는 HDFS 클러스터를 관리하는 4년 동안 오직 한번 트랜잭션 로그를 저장하는 파일 시스템의 문제 때문에 네임노드가 정지했다

Hot StandBy – AvatarNode

- HDFS 구동 시, 네임노드는 fsimage 파일로부터 파일 시스템의 메타데이터를 읽는다
- 이 메타데이터는 HDFS 안에 있는 디렉토리, 모든 파일들의 이름, 메타데이터를 포함한다.
- 그러나 네임노드는 각 블록의 위치를 영속화해서 저장하지 않는다.
- 따라서 네임노드를 Cold-start 하는 시간은 주로 두 가지 부분으로 구성된다.
- 첫 번째는, 파일 시스템 이미지를 읽어서, 트랜젝션 로그를 적용하고 새로운 파일 시스템 이미지를 디스크에 저장하는 것이다.
- 두 번째는 클러스터의 모든 블록의 위치를 모두 알려진 상태로 복구하기 위해서 다수의 데이터노드들로부터 보내온 블록 리포트를 처리하는 것이다.
- 우리의 거대한 HDFS 클러스터는 약 1억5천만개의 파일을 가진다.
- 위의 두 가지 스테이지는 동일한 시간만큼 걸린다는 것을 관찰했다.
- 전체 Cold-Restart는 약 45분 걸린다.
- 아파치 HDFS에서 사용 가능한 백업노드는 장애 시에 디스크에서 fsimage를 읽는 것을 피한다.
- 그러나 이것은 여전히 모든 데이터노드들로부터 블록 리포트를 얻어야 할 필요가 있다.
- 따라서, 백업노드 솔루션의 장애극복 시간은 20분 정도까지 될 수 있다.
- 우리의 목표는 수 초 내로 장애극복을 하는 것이다
- 따라서 백업노드 솔루션은 빠른 장애극복을 위한 우리의 목표를 충족 시키지 못한다.
- 또 다른 문제는 네임노드가 모든 트랜잭션에 대해서 백업노드를 동기적으로 업데이트한다
- 따라서 전체 시스템의 신뢰성(reliability)은 이제 Standalone 네임노드의 신뢰성 보다 더 낮아질 수 있다

Hot StandBy – AvatarNode



Hot StandBy – AvatarNode

- 하나의 HDFS 클러스터는 Active 아바타노드와 Standby 아바타노드 2개의 아바타노드를 가진다.
- Active-passive-hot-standby 쌍을 이룬다.
- 하나의 아바타노드는 보통 네임노드의 wrapper이다.
- 페이스북의 모든 HDFS 클러스터들은 트랜젝션 로그의 하나의 카피와 파일시스템 이미지의 하나의 카피를 저장하기 위해서 NFS를 사용한다.
- Active 아바타노드는 NFS 파일 시스템에 트랜젝션 로그를 쓴다.
- 동시에 Standby 노드는 NFS 파일 시스템으로부터 같은 트랜젝션 로그 파일을 읽기 위해 연다.
- 자기의 namespace에 트랜젝션을 적용하기 시작한다.
- 따라서 namespace는 primary로서 접근 가능하도록 유지된다.
- Standby 아바타노드는 Primary 노드의 check-point(검사지점)을 만드는 것 또한 처리한다.
- 그리고 나서 새로운 파일 시스템 이미지를 생성한다. 그래서 더 이상 분리된 SecondaryNameNode는 없다.
- 데이터노드는 하나의 네임노드에 말하는 것 대신에 Active와 Standby 아바타노드 모두에게 말을 한다.
- 이것은 Standby 아바타노드가 일분 이내로 활성화 될 수 있고 블록 위치에 대한 대부분의 최신의 상태 또한 가진다는 것을 의미한다.
- 아바타 데이터노드는 heartbeat, block reports, block received를 두 아바타노드 모두에게 보낸다.
- 아바타 데이터노드들은 주키퍼와 통합됐다.
- 그래서 데이터노드들은 primary로서 일하는 아바타 노드가 어느 것인지 안다. 오직 primary

Enhancements to HDFS transaction logging

- HDFS 레코드는 오직 파일이 닫히거나 sync/flushed될 때만 트랜젝션 로그에 새롭게 block-ids를 할당했다.
- 우리는 가능한 투명하게(transparent) 장애극복을 만들기를 원하기 때문에, Standby 노드는 각각의 블록이 할당 될 때마다 알고 있어야 한다.
- 그래서 우리는 각 블록 할당에 대한 edit 로그 위해서 새로운 트랜젝션을 쓴다(write)
- 이것은 클라이언트가 장애극복 바로 직전 순간에 쓰여진 파일들을 이어서(continue) 쓸 수 있도록 한다.
- Standby 노드가 Active 아바타노드에 의해서 쓰여진 트랜잭션 로그로부터 트랜잭션을 읽을 때, 부분적인 트랜잭션을 읽을 가능성이 있다.
- 이 문제를 피하기 위해서, 우리는 파일로 쓰여질 각각의 트랜잭션 당 트랜잭션의 길이, 트랜젝션 id와 checksum을 가지도록 edits 로그의 형식을 바꿔야 했다.

Transparent Failover : DAFS

- 우리는 DistributedAvatarFileSystem(DAFS)를 개발했다.
- 장애극복 이벤트에 대해서 HDFS가 투명한 접근을 제공할 수 있는 클라이언트를 위에 있는 계층(layered) 파일시스템이다.
- DAFS는 주키퍼와 통합됐다.
- 주키퍼는 Primary 아바타노드의 물리적 주소를 가지는 zNode를 유지한다.
- 클라이언트가 HDFS 클러스터에 접속을 시도할 때, DAFS는 Primary 아바타노드의 실제 주소를 가지고 있는 주키퍼에서 적절한 zNode를 찾아서 모든 성공적인 호출을 Primary 아바타노드를 향하게 한다.
- 만약 요청이 네트웍 에러를 만나게 되면, DAFS 는 주키퍼에서 primary 노드의 변화를 검사한다.
- 장애극복 이벤트가 있었던 경우에는 zNode는 새로운 Primary 아바타노드의 이름을 포함하게 될 것이다.
- DAFS 는 이제 새로운 Primary 아바타노드로의 호출을 재시도하게 될 것이다. 우리는 주키퍼를 subscription model로 사용하지 않는다.
- 왜냐하면 그것은 주키퍼 서버들에게 더 많은 자원들을 바치게(dedicate) 되게 때문이다. 만약 장애극복이 진행 중이라면, DAFS는 자동적으로 장애극복이 완료될 때 까지 막혀 있게 된다. 장액극복 이벤트는 HDFS로부터 데이터 접근하는 애플리케이션들에게 완전히 투명(transparent)하다.

Hadoop RPC compatibility

- 초기에, 우리는 메시지 애플리케이션을 위한 여러 개의 Hadoop 클러스터들을 실행하게 될 것이라는 점이 명확했다.
- 우리는 때에 맞추어 다른 클러스터들의 다른 지점에서의 새로운 소프트웨어 버전을 deploy할 수 있는 능력이 필요하다.
- 이것은 우리가 다른 버전의 Hadoop 소프트웨어가 실행중인 Hadoop 서버들과 Hadoop 클라이언트가 interoperate 할 수 있도록 향상 시키는 것이 요구된다.
- 같은 클러스터 안에서 다양한 서버 처리는 같은 버전의 소프트웨어를 실행한다.
- 우리는 서버 위에서 실행 중인 소프트웨어의 버전을 자동적으로 판단하기 위해서 Hadoop RPC 소프트웨어를 향상시켰다.

Block Availability : Placement Policy

- 디폩트 HDFS 블록 배치 정책은 여전히 최소한의 제약이 있다.
- 비지역성(non-local) 복제물(replica)을 위한 배치 결정은 random이다.
- 이것은 어떤 rack이든지, rack안에 어떤 node안이든지 배치가 될 수 있다.
- 여러 개의 노드가 동시에 고장이 날 때 데이터 손실의 가능성을 줄이기 위해서, 우리는 더 작고 설정 가능한 노드의 그룹으로 블록의 복제가 배치되는 것을 제약하는 pluggable 블록 배치 정책을 구현했다.
- 이것은 데이터 손실의 확률을 열 배 정도까지 줄여 준다. (그룹을 위해서 선택된 크기에 따라서) 우리의 전략은 오리지널 블록 주변에 복제(replica)가 배치 될 수 있는 머신과 rack의 window를 정의하는 것이다.
- rack들의 논리적인 ring을 사용한다.
- 이 ring은 머신들의 논리적인 ring을 포함한다.
- 우리는 random 블록을 손실할 확률이 노드 그룹의 크기에 따라서 증가함을 알게 되었다.
- 우리의 클러스터들에서는 rack window의 크기는 2, 머신 window의 크기는 5로 시작했다.
- 이 선택은 디폴트 블록 배치 정책보다 약 100배 정도 데이터 손실의 확률을 줄이기 때문에 선택되었다.

Performance Improvements for a realtime workload

- HDFS는 본래 MapReduce같은 high-throughput 시스템들을 위해 설계되었다.
- 최초 설계의 많은 원리들은 throughput을 높이기 위함이었지 응답 시간에 초점을 맞추지 않았다.
- 예를 들면 에러를 처리할 때, 빠른 실패에 대해서 기다리거나 retry 하는 것을 선호한다.
- 실시간 애플리케이션을 지원하기 위해서는, 에러가 나는 경우에 적절한 응답시간 제공하는 것이 HDFS의 주요한 도전과제가 되었다.

RPC Timeout

- 하나의 예는 Hadoop이 RPC 타임아웃을 다루는 방법이다.
- Hadoop은 Hadoop-RPC를 보내기 위해서 tcp 연결을 사용한다.
- RCP 클라이언트가 tcp 소켓 타임아웃을 감지할 때, RPC 타임아웃을 정의한 대신, RPC 서버에 ping을 보낸다. 만약 서버가 살아 있다면, 클라이언트는 계속해서 응답을 기다린다.
- 이 아이디어는 만약 RPC 서버가 커뮤니케이션 burst, 일시적으로 높은 부하, a stop the world GC를 경험한다면, 클라이언트는 기다려야 하거나 서버로 가는 이들 트래픽을 억눌러야(throttle) 한다.
- 반대로 타임아웃 예외를 던지거나 RPC 요청을 다시 시도하는 것은 불필요하게 작업의 실패를 유발하거나 RPC 서버로 가는 부가적인 부하를 더하게 된다.
- 그러나, 무한정 기다리는 것은 실시간 요구사항을 가지는 모든 애플리케이션에 악영향을 미친다.
- 하나의 HDFS 클라이언트는 때때로 몇몇 데이터 노드로 가는 RPC를 만든다
- 그리고 데이터 노드가 제시간에 응답을 돌려 받는 것에 실패 하면, 클라이언트는 RPC 안에서 막히게(stuck) 된다.
- 더 나은 전략은 빨리 실패해서 다른 데이터 노드로 읽기/쓰기를 시도하는 것이다.
- 따라서 우리는 서버에서 RPC 세션을 시작할 때 RPC 타임아웃을 명시하는 기능을 추가했다.

Recover File Lease

- 또 다른 향상은 writer의 임대(lease)를 빠르게 취소하는 것이다. HDFS는 하나의 파일에 오직 하나의 writer만 지원한다. 네임노드는 이런 시멘틱을 시행하기 위해서 lease들을 유지한다. 애플리케이션은 읽기를 위해 파일을 열기를 원하지만 깔끔하게 더 일찍 파일을 닫지는 않는 경우가 많이 있다. 이전에 호출이 성공할 때까지 반복적으로 로그 파일에 HDFS-append를 호출함으로서 이런 문제들이 발생했다.
- Append operation들은 파일의 soft lease에 소멸을 유발한다. 그래서 애플리케이션은 최소한의 soft lease 시간만 기다려야 한다. (기본값은 1분) 그리고 나서 HDFS 네임노드는 로그 파일의 lease를 취소한다. 두 번째로 HDFS-append operation은 보통 하나 이상의 데이터노드와 연관된 write 파이프라인을 설치(establish)하는데 불필요한 부가적인 비용이 든다. 에러가 발생했을 때, 파이프라인 설치는 10분까지 걸릴 때도 있다.
- HDFS-append 오버해드를 피하기 위해서, 우리는 파일 lease를 명시적으로 취소하는 recoverLease 라고 불리는 HDFS API를 추가했다. 네임노드가 recoverLease 요청을 받으면, 즉시 파일의 lease 홀더가 자신이 되도록 바꾼다. Lease 복구 처리를 시작한다. recoverLease RPC는 lease 복구가 완료 되었는지의 상태를 반환한다. 어플리케이션은 파일을 읽기 시도하기 전에 recoverLease로부터 성공 반환 코드를 기다린다.

Reads from Local Replica

- 애플리케이션이 확장성과 성능의 이유로 HDFS에 데이터를 저장하기를 원할 때가 있다.
- 그러나 HDFS 파일에 읽기 또는 쓰기의 latency는 머신의 로컬 파일에 읽기 또는 쓰기의 latency 보다 열 배정도까지 더 크다.
- 이 문제를 완화하기 위해서, 우리는 HDFS 클라이언트가 데이터의 로컬 복제본이 있는지 탐지하고, 데이터노드를 통한 데이터의 전송 없이 로컬 복제물로부터 투명하게 데이터를 읽도록 향상시켰다.
- 이것은 HBase를 사용하는 어떤 작업부하에서는 두 배의 성능 프로파일링 결과가 나왔다.

Realtime Hadoop : New Features

- HDFS sync
  - Hflush/sync는 Hbase와 Scribe 모두를 위한 중요한 operation이다. 클라이언트 또는 데이터노드 중 하나의 파이프라인이 실패할 때, 데이터 durability를 증가시키고 데이터를 어떤 새로운 reader에게 보이도록 만들면서, 클라이언트 쪽에 버퍼된(buffered) 쓰여진 데이터를 쓰기 파이프라인으로 밀어낸다. Hflush/sync는 동기적 operation이다. 이것은 쓰기 파이프라인으로부터 승인(ack)을 받을 때까지 반환하지 않는다는 것을 의미한다. 이 operation은 빈번히 invoke되기 때문에, 이것의 효율을 높이는 것이 중요하다. 우리가 가지는 한가지 최적화 방법은 Hflush/sync operation이 응답을 기다리는 동안에 다음 쓰기를 진행을 허용하는 것이다. 이것은 Hbase와 Scribe에서 Hflush/sync를 정기적으로 invoke하는 선정된 쓰레드에서 write throughput을 크게 증가시킨다.
- Concurrent Readers
  - 우리는 파일이 쓰여지는 동안에도 읽을 수 있는 능력이 요구되는 애플리케이션을 가지고 있다. Reader는 처음에는 파일의 메타 정보를 얻기 위해서 네임노드에게 말한다. 네임노드는 파일의 마지막 블록의 길이에 대한 가장 최신 정보를 가지고 있지 않기 때문에, 클라이언트는 복제물들 중 하나가 있는 데이터노드 중 하나로부터 정보를 가져온다. 그리고 나서 파일을 읽기 시작한다. Concurrent reader와 writer의 도전과제는 데이터의 내용과 checksum이 역동적으로(dynamically) 변하고 있을 때 데이터의 마지막 chuck를 제공(provision)하는 방법이다. 우리는 요구가 있을 때마다 마지막 chunk의 checksum을 다시 계산 함으로서 이 문제를 해결 했다.

Production HBase

- ACID Compliance
  - Atomicity
  - Consistency
- Availability Improvements
  - HBase Master Rewrite
  - Online Upgrades
  - Distributed Log Splitting
- Performance Improvements
  - Compaction
  - Read Optimizations

NEXR

Production HBase

- ACID Compliance
  - 애플리케이션 개발자는 그들의 데이터베이스 시스템이 ACID 준수를 기대해 왔다. 정말로, 강한 일관성 보장은 우리의 이전 평가에서 Hbase의 장점 중 하나다. 기존의 MVCC(multiversion concurrency control)같은 RWCC(read-write consistency control)는 충분한 격리(isolation)를 보장한다. HDFS 위에 HLog는 충분한 durability를 제공한다. 그러나, HBase가 우리가 필요한 낮은 수준의 ACID compliance의 원자성(atomicity)과 일관성(consistency)을 고수하는 것을 확실하게 하기 위해 약간의 수정이 필요하다.

- Atomicity
  - 첫 번째 걸음은 낮은 수준의 atomicity를 보장하는 것이다.
  - 그러나 노드가 고장나는 상황에서 이런 보장을 잃게 될 가능성이 있었다. 원래, 하나의 row 트랜젝션에서 여러 개의 엔트리는 HLog에 차례대로 쓰여지게 된다.
  - 만약 리젼서버(RegionServer)가 이런 쓰기 중에 죽으면, 트랜젝션은 부분적으로 쓰여질 수 있다. 새로운 개념의 로그 트랜젝션(WALEdit)을 사용해서, 각각 쓰기 트랜젝션은 이제 완전히 완료되거나 전혀 쓰여지지 않는다.

Production Hbase

- Consistency
  - HDFS는 HBase를 위해서 replication을 제공한다. 따라서 우리의 사용을 위해서 Hbase가 요구하는 대부분의 강한 일관성 보장을 다룬다. 쓰기를 하는 동안에, HDFS는 각각 복제물(replica)에 파이프라인 연결(connection)을 만들고(set up) 모든 복제물들은 어떤 데이터라도 보내질 때 ACK를 받아야 한다. Hbase는 응답 또는 실패 알림을 받을 때까지 계속하지 않는다.
  - 연속적인 숫자들을 사용할지라도, 네임노드는 어떤 잘못된 행동을 하는 복제물들을 식별하고 배제 할 수 있다. 네임노드가 이런 파일을 복구하는 데는 시간이 걸린다. HLog인 경우에는,
  - 일관성과 내구성(durability)를 유지하는 것이 절대적인 의무인 동안이 이전 전행단계
  - Where forward progress while maintaining consistency and durability anre an absolute must
  - 단 하나의 HDFS 복제물일지라도 데이터 쓰기에 실패가 탐지되면 HBase는 즉시 로그를 굴리고(roll) 새로운 블록을 얻는다.
  - HDFS는 또한 데이터 부패(corruption)에 대한 보호를 제공한다. HDFS 블록 하나를 읽는 데서, checksum 유효입증(validation) 이 선호된다 .그리고 전체 블록은 checksum 실패인 경우 폐기된다. 데이터 폐기는 또 다른 두 개의 복제물이 존재하기 때문에 드물게 문제가 된다.
  - 만약 모든 3개의 복제물이 부패한 데이터를 포함한다면, 이 블록들은 부검(post-mortem) 분석을 위해서 격리되는 것을 보장하는 부가 기능이 추가되었다.

Availability Improvements

- Hbase Master Rewrite
  - 우리는 원래 HBase 리전들을 오프라인으로 만드는 곳을 kill testing 중의 다수의 이슈를 다루지 않았다.
  - 우리는 곧 이런 문제를 발견했다.
  - 클러스터의 일시적인 상태는 오직 현재의 active Hbase 마스터의 메모리에 저장된다. 마스터를 잃게 될 경우, 이 상태는 잃게 된다. 우리는 Hbase master rewrite하는 일에 착수했다.
  - 이 rerwite의 중요한 요소는 마스터의 in-memory 상태에 있는 리전 할당 정보를 주키퍼로 옮기는 것이다. 주키퍼는 노드의 다수에 쓰여지기 때문에, 이 일시적인 상태는 마스터 장애극복 시에 잃지 않고 다수의 서버 정전에도 살아남을 수 있다.

Production HBase : Availability Improvements

- Online Upgrades
  - 클러스터 정지시간의 가장 큰 원인은 무작위의 서버 죽음이 이니라 시스템 유지관리(maintenance)였다. 우리는 이 정지시간을 최소로 하기 위해서 풀어야 하는 몇 개의 문제를 가지고 있었다.
  - 첫번 째, 우리는 리전서버들은 정지 요청을 낸 후에 shutdown을 위해서 간헐적으로 수 분이 요구된다. 이 간헐적인 문제는 긴 compaction 주기에 의해서 발생한다. 이것을 해결하기 위해서, 우리는 완료 응답에 찬성하게 하기 위해서 (?) compaction을 중단 가능하도록 만들었다.
  - 이것은 리전 서버의 중지시간을 수 초로 줄였다. 그리고 우리에게 클러스터 shutdown 시간에 대한 합리적인 경계(bound)를 주었다.

  - 또 다른 가용성 향상은 rolling restart이다. 본래, HBase는 오직 업그레이드를 위해서 전체 클러스터의 정지와 시작만 지원한다. 우리는 소프트웨어 업그레드를 수행하는 rolling restart 스크립트를 추가했다. 마스터는 자동적으로 리전서버가 정지하면 리전들을 재할당하기 때문에, 이것은 우리의 사용자들이 경험하는 정지시간을 최소화했다. 새로운 시작의 결과로 생기는 다양한 edge case 이슈들을 수정했다. 부수적으로, rolling restart 하는 도중의 다수의 버그들은 리전 오프라인과 재할당과 관련이 되어 있었다. 그래서 주키퍼 통합된 우리의 master rewrite는 이런 몇몇의 문제 또한 해결하는데 도움이 되었다.

Production HBase : Availability Improvements

- Distributed Log Splitting
  - 리전서버가 죽을 때, 그 서버의 HLog들은 스플릿되고 재수행 되야 한다. 그리고 나서 리전들은 다시 열릴 수 있고 읽기와 쓰기에 이용할 수 있도록 할 수 있다. 이전에는, 마스터는 로그를 쪼개야 했다. 그리고 나서 그들은 남겨진 리전서버들 사이에 재수행된다. 이것은 복구 처리에 가장 느린 부분이다. 서버 당 많은 HLog들이 있기 때문에 평행하게 수행될 수 있다.
  - 리전서버들 사이에 스플릿 테스크들을 관리하도록 주키퍼를 이용하면, 마스터는 이제 분산된 로그 스플릿을 condinate한다. 이것은 열 배 정도까지 복구 시간을 줄이고 리전서버들이 장애극복 성능에 심각한 영향을 끼치는 것 없이 더 많은 HLog들을 계속 유지할 수 있게 한다.

Performance Improvements

- Hbase에서 데이터 삽입은 redundant read들의 가끔 발생하는 비용에서 순차 쓰기에 초점을 맞춤으로서 쓰기 성능에 최적화되어 있다.
- 데이터 트랜잭션은 첫 번째로 커밋 로그에 쓰여진다. 그리고 나서 MemStore라고 불리는 in-memory 캐시에 적용이 된다. MemStore가 어떤 threshold에 도달하면 HFile로 쓰여진다.
- HFile은 정렬된 Key/Value 쌍을 포함하는 변경 불가능한 HDFS 파일이다. 기존의 HFile을 편집하는 대신에, 모든 flush에서 새로운 파일로 쓰여지고 per-region 목록에 추가된다. 읽기 요청들은 마지막 결과를 위해서 평행하고 결집된 다수의 HFile들로 발행된다. 효율성을 위해서, 이런 HFile들은 읽기 성능의 저하를 피하기 위해서 정기적으로 컴팩트되거나 함께 병합될 필요가 있다.

컴팩션(Compaction)

- 읽기 성능은 리전안에 파일 수와 연관이 되어 있다. 따라서 잘 튜닝된 컴패션 알고리즘에 의해 정해진다.
- 약간 더 미묘하지만, 만약 컴팩션 알고리즘이 적절하게 튜닝되지 않는 다면 네트웍 IO 효율은 또한 현저하게 영향을 미칠 수 있다. 우리의 사용 사례에 맞는 효율적인 컴팩션 알고리즘을 가지는 것에 중요한 노력이 들어간다.
- 컴팩션들은 최초의 두가지 서로다른 코드 패스로 분리되었다. minor이냐 major이냐에 따라서,.
- 마이너 컴팩션은 크기 메트릭에 기초하여 모든 파일의 부분집합을 선택한다. 반면 시간기반 메이저 컴팩션들은 무조건 모든 HFile들을 컴팩트한다. 이전에는 오직 메이저 컴팩션만 삭제, 오버라이트, 만료된 데이터를 축출하기를 처리했다.
- 이것은 마이너 컴팩션의 결과의 HFile들이 필요이상으로 크다는 것을 의미한다, 이것은 블록 캐시 효율을 떨어트리고, 이후 컴팩션에 패널티를 준다. 코드 패스들 통일 함으로서, 코드기반이 단순화되고, 파일들은 가능한 작게 유지된다.
- 다음 작업은 컴팩션 알고리즘의 향상이다. 우리는 put과 sync 지연시간이 매우 길다는 것을 알았다. 우리는 3개의 5MB 파일이 정기적으로 1 GB 파일로 컴팩트 되는 pathological case를 발견했다. 이런 네트웍 IO 낭비는 컴팩션 큐가 적체되기 시작할 때까지 계속된다. 이런 문제는 기존 알고리즘이 무조건 처음 4개의 HFile을 마이너 컴팩트하기 때문에 발생한다. 3개의 HFile이 도착한 후에 마이너 컴팩션이 시작한다.
- 해결책은 특정 크기 이상의 파일들이 무조건 컴팩트하지 않고 충분한 후보 파일들을 찾을 수 없다면 컴팩션을 건너뛰는 것이다. 후에, 우리의 put 지연시간은 25밀리초에서 3밀리초로 떨어졌다.
- 우리는 또한 컴팩션 알고리즘의 크기 비율 결정을 향상 시키는 작업 또한 했다. 본래, 컴팩션 알고리즘은 파일 나이로 정렬이 되고 근접 파일들과 비교된다. 만약 older 파일이 newer 파일의 크기의 두 배보다 작다면, 컴팩션 알고리즘은 이 파일을 포함하고 반복한다.
- 그러나 이 알고리즘은 HFile들의 수나 크기가 상당히 증가할 때는 차선책이라고 할 수 있다.
- 향상을 위해, 우리는 이제 모든 새로운 HFile들의 합계 크기의 2배 안에 있는 older 파일을 포

Read Optimizations

- 읽기 성능은 리전 안의 파일들의 수를 적게 유지하는 것과 랜덤 IO 오퍼레이션을 줄이는 것에 의해 정해진다.
- 디스크에 파일들의 수를 적게 유지하기 위해서 컴팩션을 이용하는 것 이외에도, 몇몇 쿼리에 대해서 어떤 파일을 건너 뛰는 것 또한 가능하다. 유사하게 IO 오퍼레이션을 줄인다.
- 블룸 필터는 HFile안에 로우와 컬럼이 존재하는 지 검사하기 위한 효율적이고 상수시간 방법을 제공한다. 각각 HFile은 블록 끝에 선택적으로 메타 데이터를 가지면서 차례대로 쓰여지기 때문에 블럼필터의 추가는 큰 변화 없이 끼워 넣었다.(fit in) 디스크나 메모리 캐시될 때 가각의 블룸 필터는 최대한 작게 유지된다. 특정 row 또는 컬럼을 요청하는 쿼리들에 대해서, 각각 Hfile의 캐시된 블룸 필터를 검사해서 몇몇 파일들은 완전히 건너 뛸 수 있게 됐다.

Deployment and operational experiences

- 과거 몇 년 동안에, 우리는 10노드를 가지는 작은 HBase 테스트 클러스터에서 수천개의 노드를 실행하는 많은 클러스터들로 이동했다.
- 이런 Deployment는 이미 라이브 프로덕션 트래픽을 수백만 사용자에게 제공하고 있다. 같은 기간 동안에, 우리는 HBase와 대항하는 애플리케이션 로직과 마찬가지로 핵심 소프르웨어 (HBase/HDFS)위에서 빠르게 반복해왔다.
- 이런 유동적인 환경에서, 고품질 소프트웨어를 만들고, 정확히 Deploy하고 실행중인 시스템을 모니터하고 최소한의 정지시간으로 예외를 탐지하고 수정하는 능력이 중요하다.

Testing

- 우리 HBase 솔루션의 설계의 초기 부터, 우리는 코드 안정성에 대해서 걱정되었다.
- 우리는 처음에 오픈 소스 HBase 코드의 안정성과 내구성을 태스트 하는 것이 필요했다.
- 그리고 부가적으로 우리가 미래에 변경사항들의 안정성을 보장하는 것 또한 필요했다.
- 그래서, 우리는 HBase 테스트 프로그램을 작성했다.
- 이 태스트 프로그램은 생성된 데이터를 Hbase로 쓴다. 데이터는 미리 결정되어 있기도 하고 랜덤으로 생성된다. 테스터는 데이터를 Hbase 클러스터에 쓰고, 동시에 일기고 모든 데이터가 추가되었는지 검증한다. 우리는 테스터가 클러스터의 프로세스들을 무작위로 선택해서 죽이고 나서 반환된 데이터베이스 트랜젝션들이 성공적으로 실제로 쓰여졌는지 검증하도록 향상 시켰다.
- 이것은 많은 이슈를 잡아내는데 도움이 됐다. 여전히 우리의 변경사항을 테스트 하는 첫 번째 방법이다.
-
- 우리의 공통 클러스터가 분산된 방식으로 운용되는 많은 서버들을 포함할지라도, 우리의 로컬 개발 검증은 일반적으로 유닛 테스트와 단일 서버구성을 포함한다. 우리는 단일서버 구성과 실제 분산 시나리오 사이의 의존성에 대해서 걱정을 했다. 우리는 라이브 서버에 간단한 CRUD 작업부하를 실행하는 HBase Verify 라고 불리는 유틸리티를 만들었다.
- 이것은 간단한 API 호출을 시험할 수 있고 수분내로 부하 테스트를 실행 할 있게 한다.
- 이 유틸리티는 심지어 알고리즘들이 대규모 환경에서 처음으로 평가되는 곳인 어둠의 출시 클러스터(Dark Launch Cluster) 보다 더 중요하다.

Monitoring and tools

- HBase의 생산 사용에서 더 많은 경험을 얻었기 때문에, 리전서버들에게 리전을 일관되게 할당하는 것이 우리의 주요한 문제라는 것이 명확했다. 두개의 리전서버는 같은 리전을 서브하거나 리전이 할당되지 않은 채 남겨지게 된다. 이런 문제들은 다른 장소에 저장된 리전들의 상태에 대한 메타정보가 불일치하기 때문에 발생한다.
- Hbase, zookeeper 에 META 리전들
- HDFS 에 리전과 대응되는 파일들
- 리전서버들의 in-memory 상태들
- 많은 문제들이 시스템적으로 해결이 되고, HBase Master rewrite의 일부가 광범위하게 테스트되었다 할지라도, 우리는 생산 부하에서 보여지는 엣지 케이스가 걱정되었다. 결국 우리는 데이터베이스 수준 FSCK 같은 HBCK를 만들었다. 이것은 메타 데이터의 다른 소스들 사이의 일관성을 검증하도록 도와준다.
- 공통된 불일치성 때문에, 우리는 HBCK 'fix' 옵션에 in-memory 상태를 비우고, HMaster가 불일치 리전을 재할당 하는 것을 추가했다. 요즘에는 우리는 가능한 일찍 문제들을 잡아내기 위해서 생산 클러스터에 거의 연속적으로 HBCK를 실행한다.
-
- 클러스터 모니터링에서 중요한 요소는 운영 메트릭들이다
- 특히, 리전서버 메트릭은 HMaster 또는 Zookeeper 메트릭들 보다 클러스터 건강을 평가하는 훨씬 유용한다. HBase는 이미 JMX 를 통해서 추출되는 몇 개 메트릭들을 가지고 있다. 모든 메트릭은 RPC 요청이나 로그 쓰기 같은 짧게 실행되는 오프레이션을 위한 것들이다. 우리는 컴팩션, 플러시, 로그 스플릿 같은 길게 실행되는 사건들을 모니터링 하기 위한 메트릭을 추가할 필요가 있었다. 간간히 중요해지는 메트릭은 버전 정보이다.
- 우리는 분기된 버전들을 가지는 다수의 클러스터를 가지고 있다. 만약 클러스터 crash가 발생

Manual versus automatic splitting

- 새로운 시스템을 배울 때, 우리는 즉시 사용해야 하는 기능과 적용을 미룰 수 있는 기능을 결정하는 것이 필요했다. Hbase는 리전의 크기가 매우 커질 때 단일 리젼을 두 개의 리전으로 분리하는 자동 스플릿 기능을 제공한다. 우리는 우리의 사용사례에서 자동 스필릿은 선택적인 기능으로 결정했다. 그리고 대신 수동 스플릿 유틸리티를 개발했다.

Dark Launch

- 레거시 메시징 시스템을 이주하는 것은 하나의 주요한 장점을 제공한다. : 실제 세계 테스팅 능력이다.
- 페이스북에서, 우리는 "Dark Launch"라고 불리는 테스팅/롤아웃 프로세스를 사용한다.
- 사용자들에게 어떤 UI 변경사항도 노출되지 않으면서 사용자기반의 부분집합으로 중요한 백엔드 기능을 시험하는 장소이다.
- 이 시설은 몇몇 사용자에 대한 이중-쓰기 메시징 트래픽를 레거시 시스템과 HBase 양쪽에게 보낸다. 이것은 유용한 성능 밴치마크를 한다. 인공적인 밴치마크와 추정에 순전히 의존하는 대신에 실제적 Hbase 명복지점을 찾게 해준다.
- 생산 출시 이후일지라도 우리는 여전히 Dark Launch 클러스터를 많이 사용한다는 것을 알았다.
- 모든 코드 변경사항은 Dark Launch 위에서 보통 일주일을 보낸다. 그리고 나서야 생산 배치가 고려된다. 부가적으로 Dark Launch는 보통 생산 클러스터가 다루는 예상의 최소 두배의 부하를 준다.

Dashboards / ODS integration

- 우리는 JMX로 추출되는 메트릭들을 가진다. 그러나 이런 메트릭들을 시각화하고 클러스터 핼스를 분석하는 쉬운 방법이 필요하다
- 우리는 ODS, Ganglia와 비슷한 내부 툴을 사용하기로 결정했다.
- 라인 그래프로 중요한 메트릭을 시각화하기 위해서.
- 우리는 클러스터당 하나의 대시보드를 가진다. 이것은 평균이나 이상행동을 시각화하는 다양한 그래프를 가진다.
- 잘못 행동하는 리전서버들을 식별하기 위해 최소/최대를 그래프로 그리는 것 중요하다.
- 이것은 애플리케이션 서버가 처리하는 큐가 정체를 유발 할지도 모른다.
- 가장 큰 장점은 작업부하 상황에서 클러스터가 변경사항에 대해 어떻게 반응하는지를 실시간으로 관찰하기 위한 통계수치를 관찰 할 수 있다는 것이다.

Backups at the application layer

- 큰 데이터셋의 정기적인 백업은 어떻게 수행하나?
- 하나의 선택은 하나의 HDFS 클러스터에서 또 다른 HDFS 클러스터로 데이터를 복사 또는 복제하는 것이다. 이런 접근 방법은 연속적이지 않기 때문에, 다음 백업 이벤트 전에 HDFS안에 데이터가 이미 부패 했을 가능성이 있다. 이것은 명백히 수용할 수 있는 위험은 아니다.
- 대신, 우리는 대체적인 애플리케이션 로그를 연속적으로 생성하도록 애플리케이션을 강화하기로 결정했다.
- 이 로그는 Scribe를 통해서 전송되고 웹 분석에 사용되는 클러스터와 구분된 HDFS 클러스터에 저장된다. 우리의 웹 애플리케이션의 대용량 클릭 로그를 Hive 분석 저장소로 전송하기 위한 동일한 소프트웨어 스택을 사용한다.

Schema Changes

- HBase는 현재 기존 테이블의 온라인 스키마 변경을 지원하지 않는다. 이것은 만약 기존 테이블에 새로운 컬럼 패밀리를 추가하는 것이 필요하다면, 테이블 접근을 중단하고, 테이블을 사용불가능하게 만들고, 새로운 컬럼 패밀리를 추가하고, 테이블을 다시 온라인 가져오고, 로드를 제시각 한다. 이것은 심각한 결점이다. 우리는 작업부하를 멈춰야하는 아주 비싼 비용을 지불해야 한다.
- 대신에, 우리는 몇몇 추가적인 컬럼 패밀리들을 핵심 Hbase 테이블에 미리 생성했다. 애플리케이션은 현재 이 컬럼 패밀리에는 데이터를 저장하지 않지만, 미리에는 사용할 수 있다.

Reducing Network IO

- 몇 개월 생산 실행 이후에, 우리는 대시보드로부터 우리가 네트웍 IO 한계라는 것을 빠르게 알 수 있었다.
- 우리는 어디서 네트워 IO 트래픽가 들어오는지 분석하는 방법이 필요했다.
- 우리는 JMX 통계정보의 조합을 사용하고 24시간 간격으로 단일 리전 서버의 총 네트웍 IO를 추정하기 위해서 로그를 수집했다.
  - MemStore 플러시 (15%)
  - 크기 기반 마이너 컴팩션 (38%)
  - 시간 기반 메이저 컴팩션 (47%)

- 우리는 메이저 컴팩션 주기를 하루에서 일주일로 증가시키는 것만으로 네트웍 IO를 40% 감소시킬 수 있었다. 우리는 또한 특정 컬럼은 HLog에 로그되는 것을 배제시켜서 큰 이득을 봤다.

Future work

- 페이스북에서 Hadoop과 Hbase의 사용은 이제막 시작했다.
- 우리는 우리의 애플리케이션을 최적화 하는 것을 지속할 것을 기대한다.
- 더 많은 애플리케이션에서 HBase를 사용하는 것을 시도하므로써, 우리는 HBASE에 secondary Index와 summary view의 유지관리를 위한 지원을 추가 하는 것을 논의하고 있다.