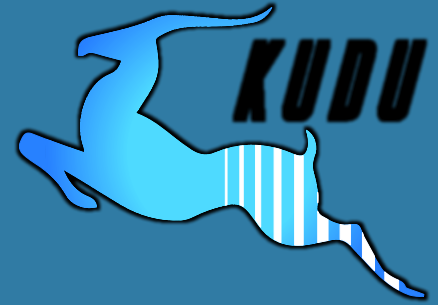


cloudera



Hadoop最新结构化存储利器 Kudu

陈飏

cb@cloudera.com

Cloudera

关于我...

陈飏

Cloudera售前技术经理、资深方案架构师

<http://biaobean.pro>

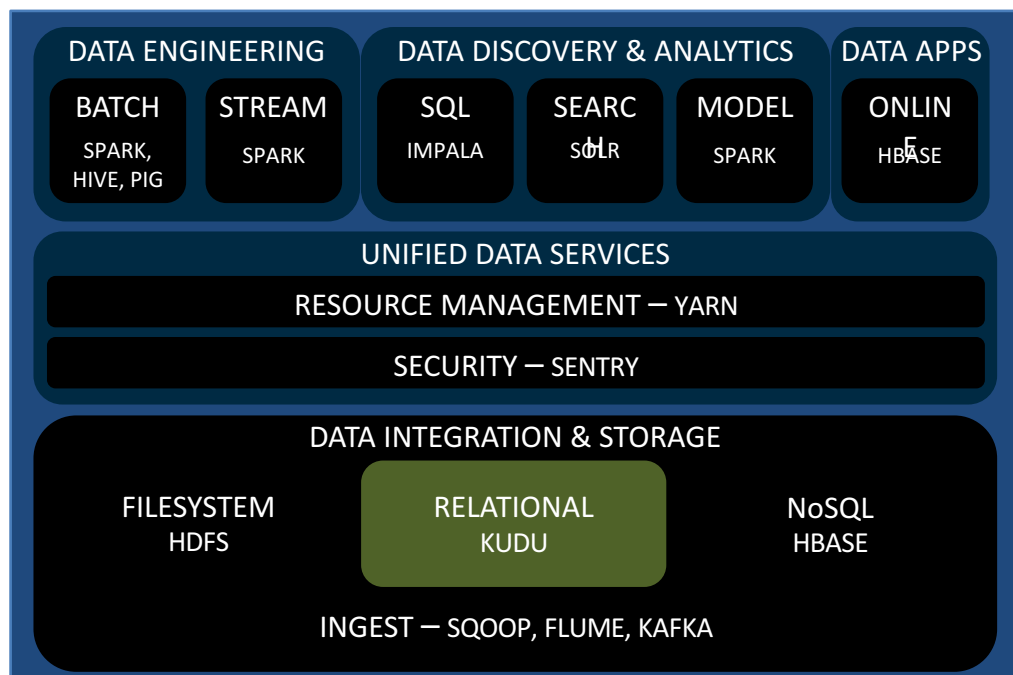


原Intel Hadoop发行版核心开发人员，成功实施并运维多个上百节点Hadoop大数据集群。

- 曾在Intel编译器部门从事服务器中间件软件开发，擅长服务器软件调试与优化，与团队一起开发出世界上性能领先的XSLT 语言处理器
- 2010年后开始Hadoop 产品开发及方案顾问，先后负责Hadoop 产品化、HBase 性能调优，以及行业解决方案顾问



Kudu: 高效分析快数据的存储系统



- Hadoop生态中新的支持数据修改的列式存储系统
 - 简化可变数据上构建分析应用的架构
 - 旨在提升快速分析的性能
 - 与Hadoop原生集成
- Apache顶级项目，100% 开源
- 2016年9月发布1.0正式版本





开发Kudu的初衷和设计目标

Why build Kudu?

为什么Cloudera要开发Kudu?

- 有没有因为Hadoop生态系统中缺少某种存储技术而使我们无法解决客户面临的业务问题?
- 我们的大数据平台充分利用了过去10年硬件的进步吗?



硬件行业的进展

- **机械硬盘 -> 固态硬盘**
 - **NAND闪存**: 读数据时IOPS达到45万个, 写数据时IOPS达到25万个, 读数据的吞吐率约为每秒2GB, 写数据的吞吐率达到每秒1.5GB, 存储成本低于每GB3美元并且持续下降
 - **3D XPoint memory** (比NAND快1000倍, 比RAM更有成本优势)
- **内存成本更低, 单机内存量更大:**
 - 过去几年64->128->256GB
- **启示1: CPU将成为性能瓶颈, 现有存储系统设计并未考虑高效使用CPU**
- **启示2: 列式存储也可以用于随机访问**



车联网



- 分析
 - 开发人员希望知道如何优化充电性能
 - 新版本软件升级后随着时间推移是如何影响汽车性能的?
- 实时
 - 客户希望知道是否是未成年人在驾驶。他们加速多快? 时速多少? 他们在哪里?
 - 汽车设备——比如在服务前或服务中拿到最新的诊断数据包



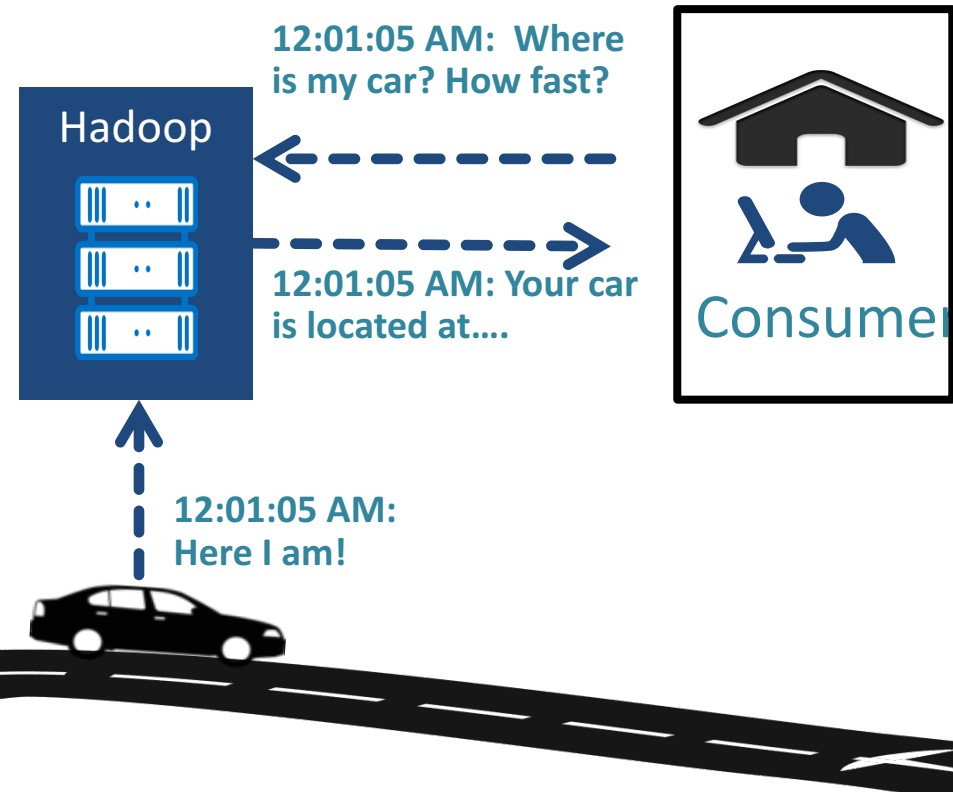
车联网



- 分析
 - 开发人员希望知道如何优化快速有效率的Scan = HDFS
 - 随着时间推移是如何影响汽车性能的?
- 实时
 - 客户希望知道是否是未成年人在驾驶。他们加速多他们在哪
 - 快速插入/查找 = HBase
 - 汽车设备 例如在服务前或服务中拿到最新的诊断数据包

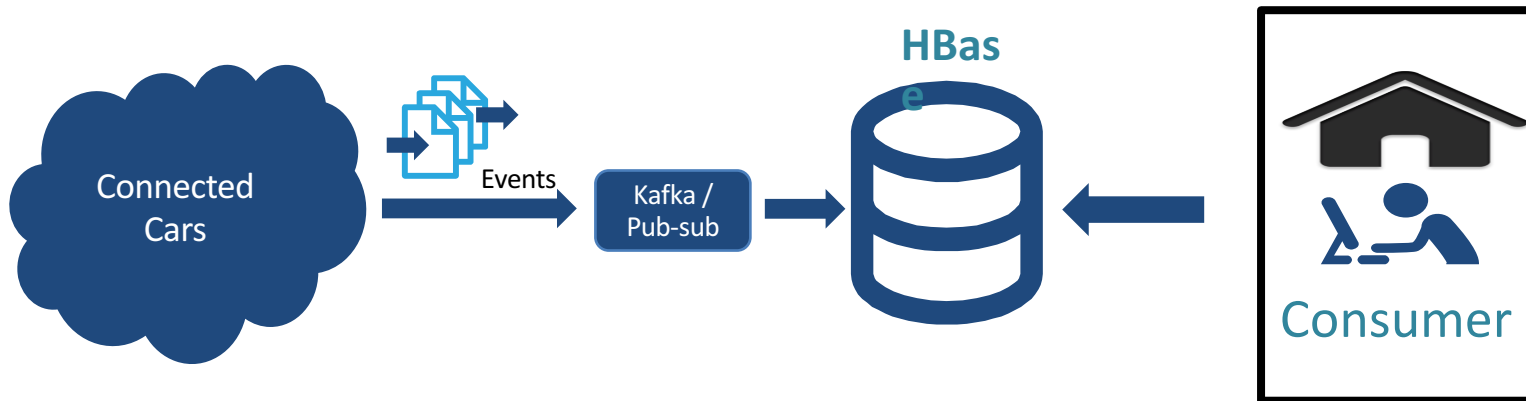
How would we build the Real-time System Today?

- Cars are constantly producing data, being ingested in real-time or micro-batches
- A consumer wants to know what is the location and speed of their car in real-time.
- Requires:
 - potentially millions of low latency writes / second
 - **low latency, random access**



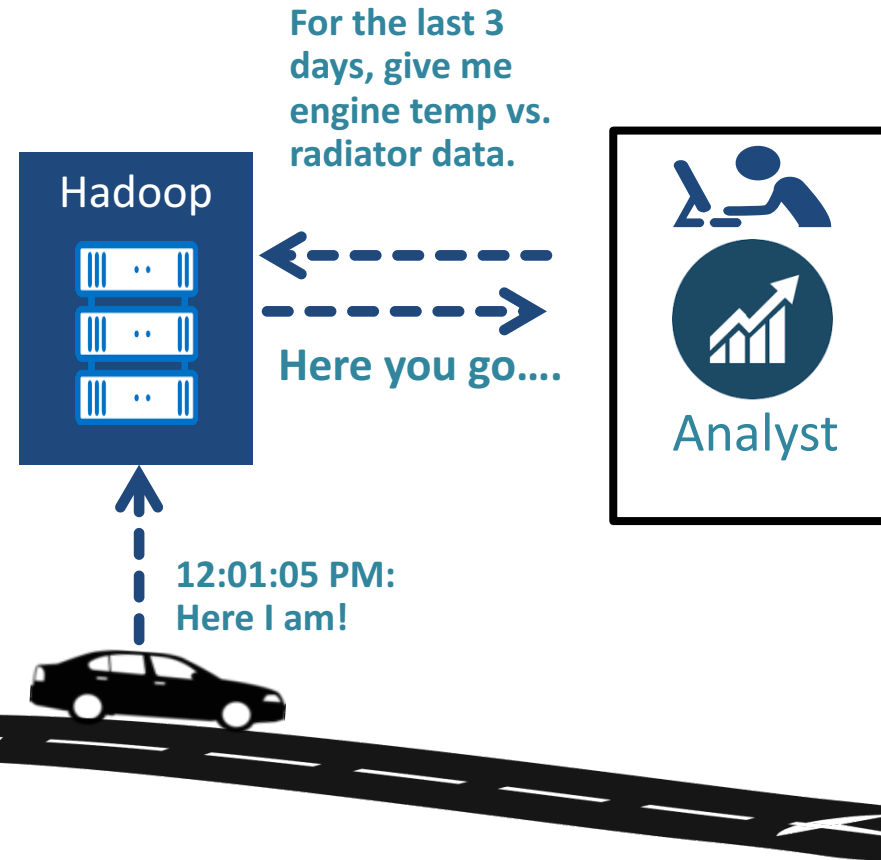
How would we build the Real-time System Today?

- Hbase Provides:
 - **Fast, Random Read & Write Access**
 - **“Mini-scans”**
 - Scale-out architecture capable of serving Big Data to many users



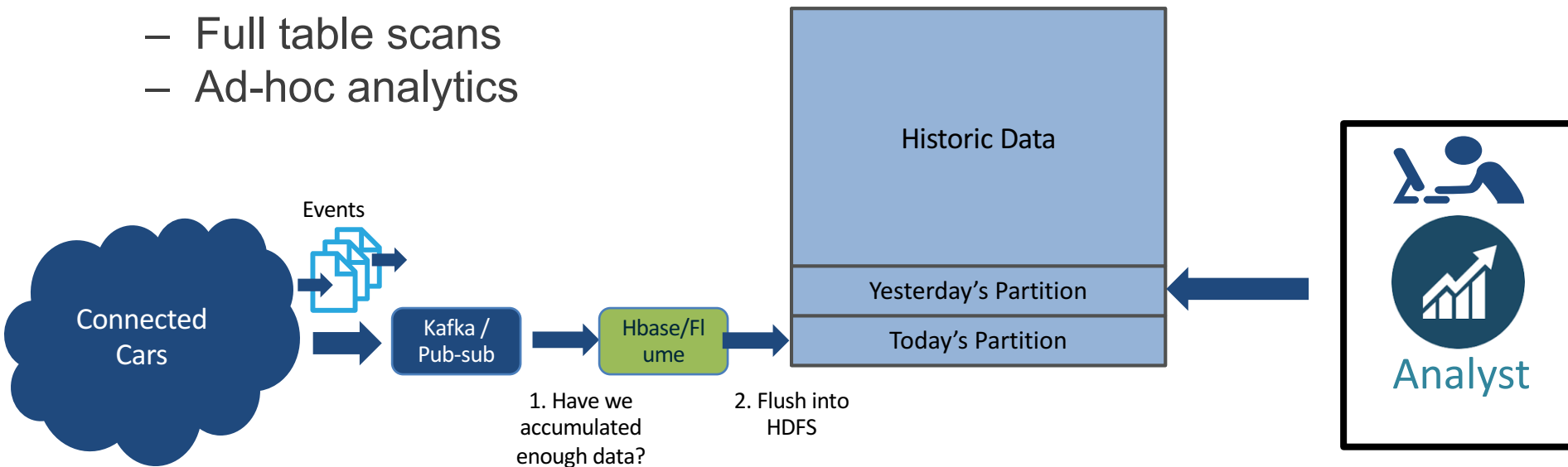
How would we build the Analytics System Today?

- Cars are constantly producing data, being ingested in real-time or micro-batches
- GM just made a software update to temp control module. How's it working?
- **Requires fast, efficient scan performance**



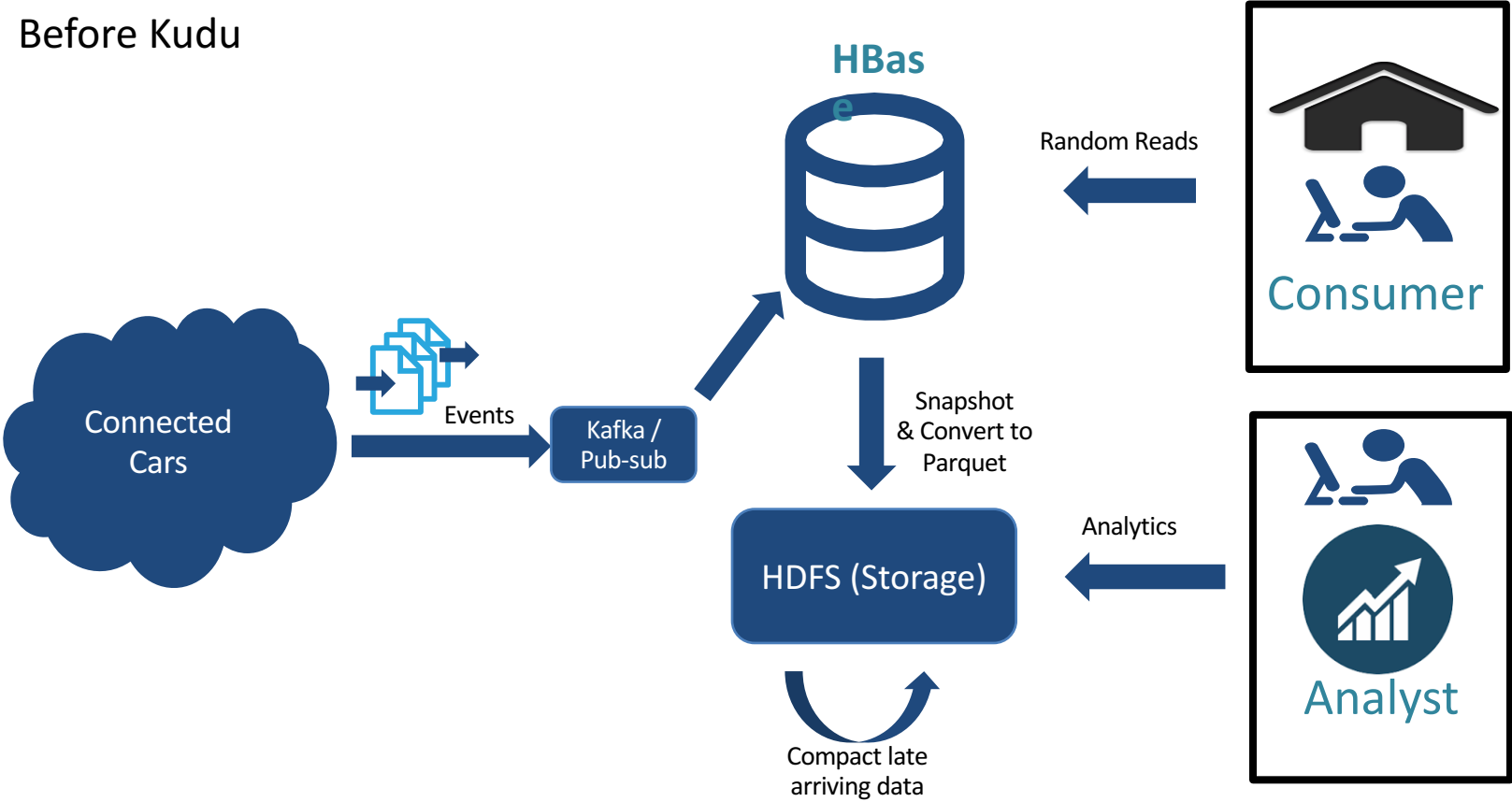
How would we build the Analytics System Today?

- HDFS Excels at:
 - Full table scans
 - Ad-hoc analytics



Hybrid big data analytics pipeline

Before Kudu



HBase+HDFS混合架构的复杂性

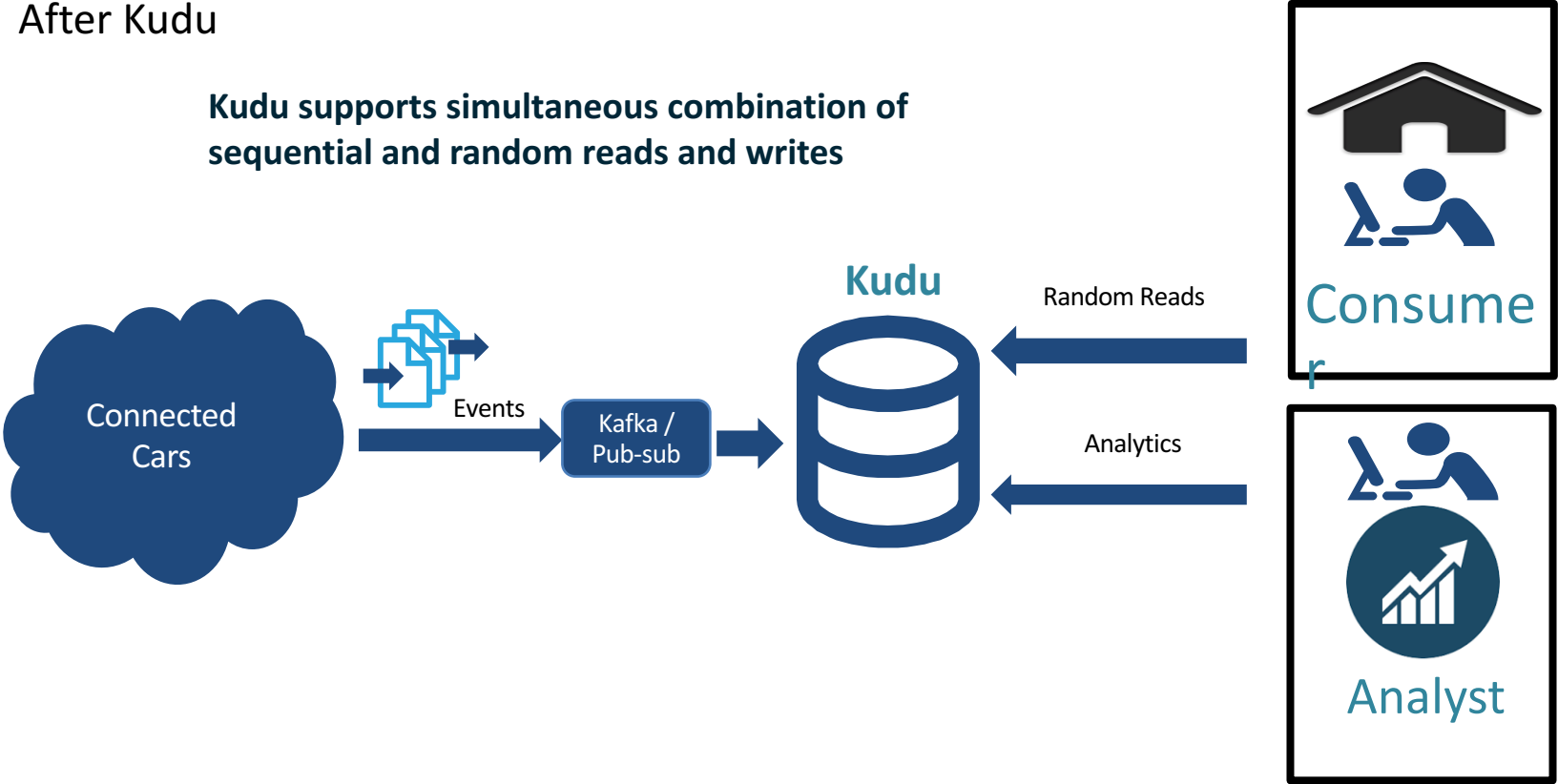
- 同时提供高性能的顺序扫描和随机查询，避免使用HBase+HDFS混合架构的复杂性：
 - 开发：必须编写复杂的代码来管理两个系统之间的数据传输及同步
 - 运维：必须管理跨多个不同系统的一致性备份、安全策略以及监控
 - 业务：新数据从达到HBase到HDFS中有时延，不能马上供分析
 - 在实际运行中，系统通常会遇到数据延时达到，因此需要对过去的数据进行修正等。如果使用不可更改的存储（如HDFS文件），将会非常不便。



Hybrid big data analytics pipeline

After Kudu

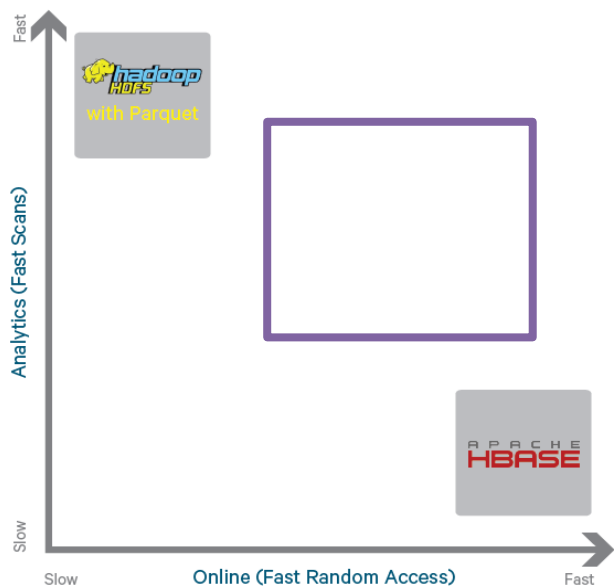
Kudu supports simultaneous combination of sequential and random reads and writes





What is Kudu?

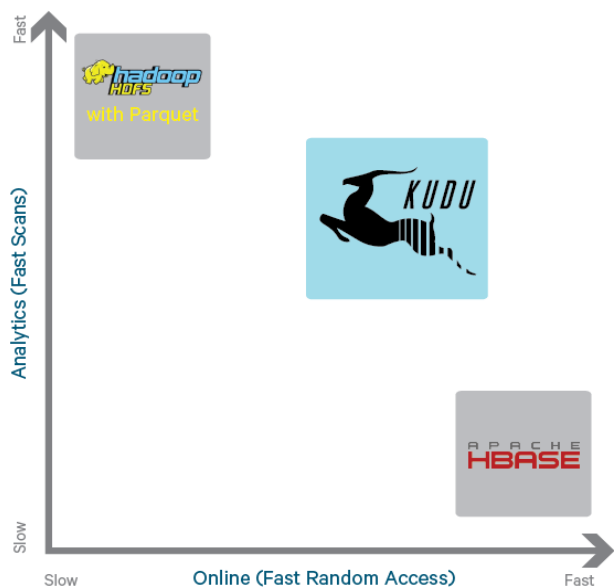
当前Hadoop平台上的存储组件概括



- **HDFS的强项:**
 - 高效的顺序扫描能力
 - 支持高吞吐的数据追加
- **HBase的强项:**
 - 高效的按行随机存取能力
 - 支持数据的修改
- 可以“鱼”和“熊掌”兼得吗？
 - 如何实现对实时变化的数据集做高效的数据分析呢(Fast Analysis on Fast Data)？



Kudu的设计目标



- 扫描大数据量时吞吐率高(列式存储和多副本机制)
 - 目标: 相对Parquet的扫描性能差距在2x之内
- 访问少量数据时延时低(主键索引和多数占优复制机制)
 - 目标: SSD上读写延时不超过1毫秒
- 类似的数据库语义(初期支持单行记录的ACID)
- 关系数据模型
 - SQL查询
 - “NoSQL”风格的扫描/插入/更新(Java客户端)



Kudu: 可扩展的高速结构化存储

可扩展性

- 通过 275 台服务器测试(集群数据量约3PB)
- 设计能支持扩展到上千节点的服务器和几十PB的数据量

高速

- 集群吞吐率达到每秒上百万读/写
- 每节点数据读取吞吐率每秒几GB

表

- 如同传统数据库一样用结构化表来表示数据
- 单表记录行数可超过一千亿条



Kudu表中的行记录访问

- 类似SQL模式的表
 - 有限的列数 (不同于HBase/Cassandra)
 - 数据类型: BOOL, INT8, INT16, INT32, INT64, FLOAT, DOUBLE, STRING, BINARY, TIMESTAMP
 - 一部分列构成联合主键
 - 快速ALTER TABLE
- “NoSQL”风格的 Java, Python和C++语言API
 - Insert(), Update(), Delete(), Scan()
- 与MapReduce, Spark和Impala无缝对接
 - 正在对接更多处理引擎, 如Drill, Hive



Kudu不是…

- 不是一个SQL引擎，“Bring Your Own SQL” (“BYO SQL”)
 - Kudu是存储层，SQL处理层需要Impala或Spark等
- 不是一个文件系统
 - 数据必须按表的形式结构化存储
- 不是一个运行在HDFS上的应用
 - Kudu是另外一个原生的Hadoop存储引擎
 - 通常是要和HDFS并肩同部署
- 不是一个内存数据库
 - 能很快的处理数据可全部载入内存的任务，也能支持非常大的数据量！
- 不是用来替代HDFS或HBase
 - 要为适当的应用场景选择适当的存储
 - 在很多应用场景下HDFS或HBase仍然会更适合
 - Cloudera会继续耕耘所有这些组件



Kudu+Impala与MPP DWH的比较

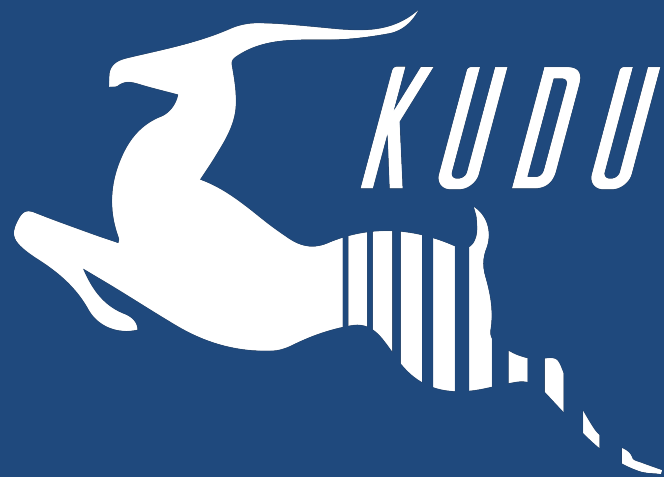
共性

- ✓ 通过SQL进行快速分析查询，并包含大多数常用的最新特性
- ✓ 能插入、更新和删除数据

不同

- ✓ 更快的流式数据插入
- ✓ 更好的Hadoop集成
 - 在同一集群中能进行HDFS和Kudu表之间的JOIN
 - 与Spark, Flume等的集成
- ✗ 批量插入变慢
- ✗ 没有基于事务的数据加载、跨行事务以及索引





Kudu应用场景

Kudu的应用场景

Kudu最适合需要同时支持顺序和随机读和写的应用场景

- 时间序列
 - 举例: 股市行情数据; 欺诈检测和预防; 风险监控
 - 应用类型: Insert, updates, scans, lookups
- 机器数据分析
 - 举例: Network threat detection
 - 应用类型: Inserts, scans, lookups
- Online Reporting
 - 举例: ODS
 - 应用类型: Inserts, updates, scans, lookups



业界应用

Financial Services



- Streaming market data
- Real-time fraud detection & prevention
- Risk monitoring

Retail



- Real-time offers
- Location-based targeting

Public Sector

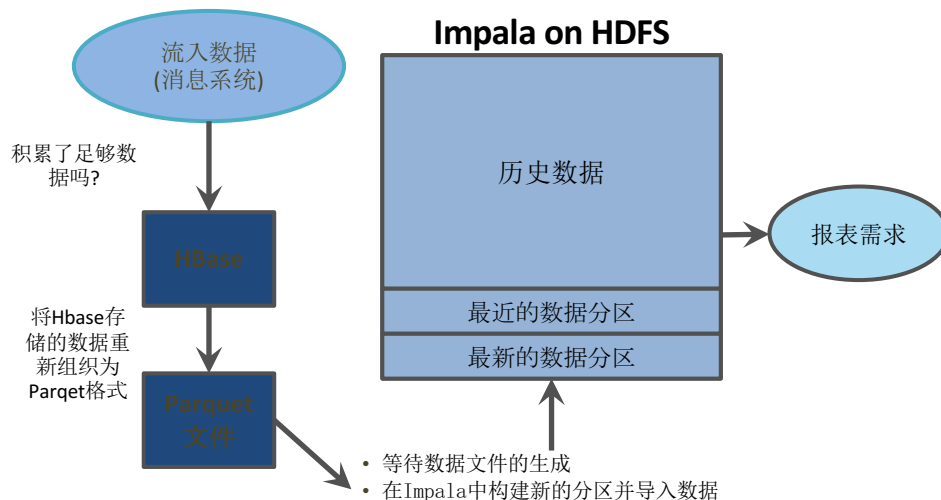


- Geospatial monitoring
- Risk and threat detection (real time)



当前Hadoop实时数据分析的现状

当前的大数据架构：存储架构太复杂

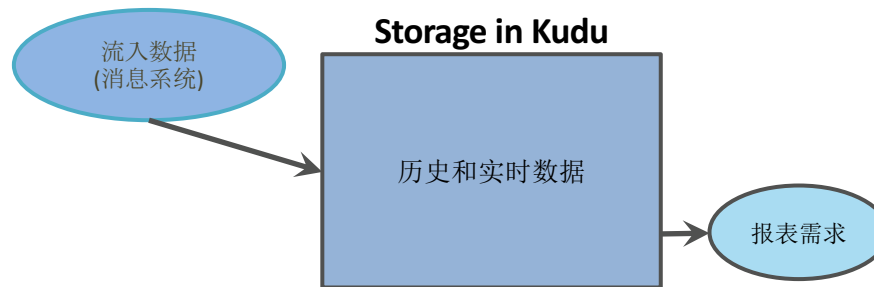


但是怎样处理下面的问题?

- 怎么有效处理转换过程中的错误?
- 如何定义将HBase数据转换成Parquet格式作业的周期?
- 从数据进入到报表中能体现之间的时延如何量化?
- 作业流程怎么保障不被其他操作打断?



使用Kudu的Hadoop实时数据分析



改进点:

- 只要一套系统
- 不需要后台定时的批处理任务
- 轻松应对数据迟到和数据修正
- 新数据立即用于在分析和业务运营





Design & Internals

Kudu的核心设计

- 有类型的存储
- 基础构件：子表(*Tablet*)
 - 数据表被水平划分为子表，类似于分区(*Partition*)
- 通过类似于Paxos的投票协议(*Raft*)来管理一致性
- 框架支持地理上的分开部署，支持多活(*Active/Active*)系统



Tables(表)和Tablets(子表)

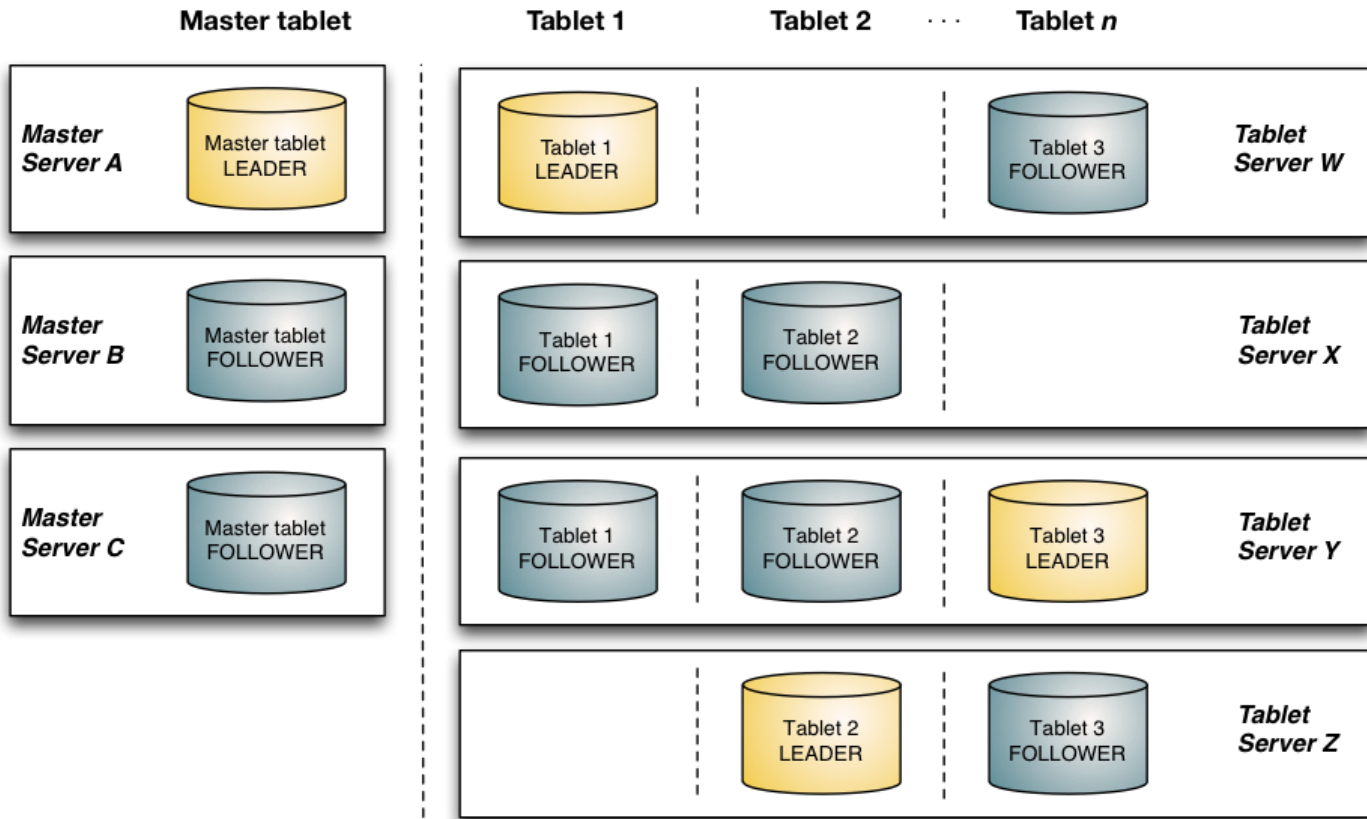
- 表被水平划分为子表
 - 支持按范围或哈希分区
 - PRIMARY KEY (host, metric, timestamp)
DISTRIBUTE BY HASH(timestamp) INTO 100
BUCKETS
- 每个子表有多个副本(3个或5), 采用**Raft一致性协议**
 - 数据可以从任一副本读取, 并且数据由Leader更新保证, 自动容错
 - 低MTTR: ~5 seconds
- **子表服务器保存子表**
 - 数据存放在本地, 而不是HDFS
- **Master** 节点负责元数据管理服务
 - 创建/删除表和子表(Tablet)
 - 定位子表(Tablet)



元数据

- **带副本的master***
 - 用作子表目录 (“META” 表)
 - 用作catalog (table schemas, etc)
 - 用于负载均衡(跟踪子表服务器是否在线, 增加低于要求副本数的子表的副本数)
- **所有元数据都缓存在内存, 性能更好**
 - 80节点的压力测试, GetTableLocations RPC 性能:
 - 99th percentile: 68us, 99.99th percentile: 657us
 - <2% peak CPU usage
- **在客户端上配置master地址**
 - 首次访问时向master询问子表地址, 随后缓存在客户端





LSM vs Kudu

- LSM – Log Structured Merge (Cassandra, HBase, etc)
 - Inserts和updates首先都写入内存 (MemStore) 然后再刷到磁盘文件(HFile/SSTable)
 - 读数据时合并磁盘HFile得到结果
- Kudu
 - 有些共同点(Memstore, Compaction)
 - 实现更精巧
 - 为了更好的读(特别是扫描)性能牺牲少许写入性能



性能特性

- 非常**高效率地利用CPU**
 - 使用最新特性的C++实现，包括特殊CPU指令、LLVM的JIT编译等技术
- 延时取决于存储硬件的性能
 - 在SSD或更新技术上的响应延时一般不超过1毫秒
- **没有内存GC**，使用大内存也不会有暂停
- 使用**Bloom Filter**减少了很多磁盘访问的要求



Kudu的折衷处理

- 随机更新速度稍慢
 - HBase的实现模型无需任何磁盘访问就能完成随机的数据更新
 - Kudu在更新时需要一次主键查找，在插入时需要Bloom查询一次，有可能需要到磁盘上寻址
- 单行读也可能稍慢
 - 列式存储更有利于扫描(Scan)场景
 - 在读取最近更新过很多次的行时尤其慢 (例如YCSB的“zipfian”测试)
 - **未来**：可能引入列组(Column Group)来解决更注重单行读取的场景



容错

- FOLLOWER临时故障:
 - Leader仍旧可以获得多少投票
 - 在5分钟内重启子表服务器,子表服务器会重新加入集群,对用户透明
- LEADER临时故障:
 - Leader每1.5秒应该向Followers发送心跳信息
 - 如果连续3次没收到Leader的心跳信息将触发新一轮的Leader选举3!
 - 新LEADER在几秒钟内选出
 - 旧Leader5分钟内重启后以FOLLOWER身份重新加入
- N个副本可以容忍 $(N-1)/2$ 个节点失败



容错(2)

- 长期失败:
 - Leader感知到某个 follower失败超出5分钟
 - 排除该follower
 - Master选择一个新副本
 - Leader将数据拷贝到该新副本



子表的设计

- 先将Inserts写入内存 (类似于HBase的Memstore)
- 然后刷到磁盘
 - 列式存储, 与Apache Parquet格式类似
- Updates采用MVCC (更新都带有时间戳, 不会直接修改原始数据)
 - 允许“SELECT AS OF <timestamp>” 查询和跨子表的一致性扫描
- 扫描当前数据, 性能接近最优
 - 无需按行进行分支, 快速的矢量化解码和谓词计算
- 如果对最新数据执行高频率的更新, 性能下降





APIs

cloudera

原生支持的客户端

- First class clients implemented in Java and C++
 - Experimental Cython client that wraps the C++
- Supported operations
 - DDL (create, alter, delete tables)
 - DML (insert, update, delete)
 - Scan (with predicate pushdown, column projection)



Impala + Kudu

```
CREATE TABLE my_first_table (  
id BIGINT,  
name STRING  
)  
DISTRIBUTE BY HASH (id) INTO 16 BUCKETS  
TBLPROPERTIES(  
  'storage_handler' = 'com.cloudera.kudu.hive.KuduStorageHandler',  
  'kudu.table_name' = 'my_first_table',  
  'kudu.master_addresses' = 'kudu-master.example.com:7051',  
  'kudu.key_columns' = 'id'  
);
```



Impala + Kudu – Pre Splitting

```
CREATE TABLE cust_behavior (  
  _id BIGINT,  
  salary STRING,  
  edu_level INT,  
  usergender STRING,  
  rating INT)  
DISTRIBUTE BY RANGE(_id)  
SPLIT ROWS((1439560049342), (1439566253755), (1439572458168), (1439578662581), (1439584866994),  
(1439591071407))  
TBLPROPERTIES(  
  'storage_handler' = 'com.cloudera.kudu.hive.KuduStorageHandler',  
  'kudu.table_name' = 'my_first_table',  
  'kudu.master_addresses' = 'kudu-master.example.com:7051',  
  'kudu.key_columns' = 'id'  
);
```



Impala + Kudu – Update & Delete

```
UPDATE my_first_table SET name="bob" where id = 3;
```

```
DELETE FROM my_first_table WHERE id < 3;
```



Impala integration

- `CREATE TABLE ... DISTRIBUTE BY HASH(col1) INTO 16 BUCKETS AS SELECT ... FROM ...`
- `INSERT / UPDATE / DELETE`
- Optimizations: predicate pushdown, scan locality, scan parallelism
- More optimizations on the way
- Not an Impala user? Community working on other integrations (Hive, Drill, Presto, etc)



Spark DataSource integration

Available as of Kudu 0.9

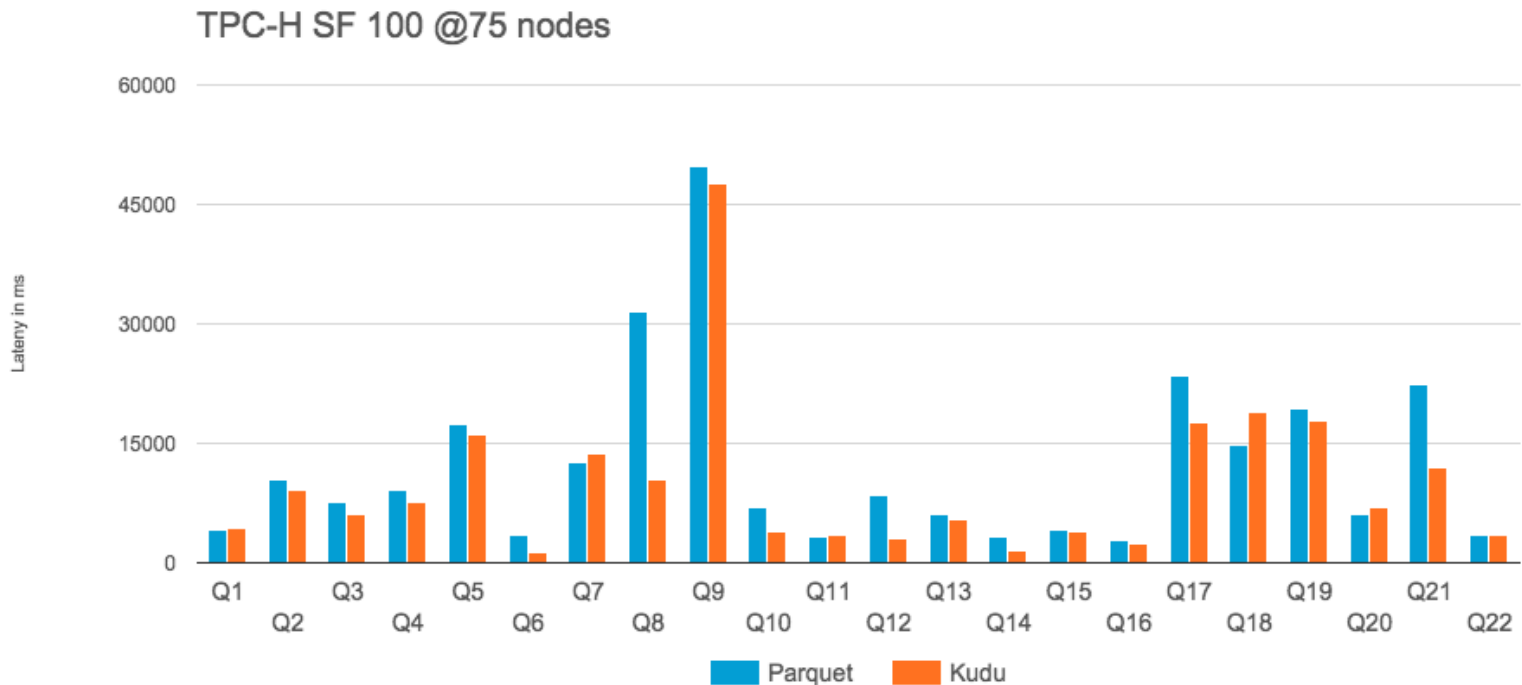
```
// Import kudu datasource
import org.kududb.spark.kudu._
val kuduDataFrame = sqlContext.read.options(
    Map("kudu.master" -> "master.address.example.com", "kudu.table" -> "my_table_name")).kudu
// Then query using spark api or register a temporary table and use spark sql
kuduDataFrame.select("id").filter("id" >= 5).show()
// (prints the selection to the console)

// Register kuduDataFrame as a temporary table for spark-sql
kuduDataFrame.registerTempTable("kudu_table")
// Select from the dataframe
sqlContext.sql("select id from kudu_table where id >= 5").show()
// (prints the sql results to the console)
```



TPC-H (Analytics benchmark)

- 集群由75个TS 和一个master构成
 - 每个节点12 块硬盘, 128GRAM
 - Kudu 0.5.0+Impala 2.2+CDH 5.4
 - TPC-H Scale Factor 100 (100GB)
- 结果
 - 对内存数据,Kudu性能比Parquet高31% (几何平均)
 - 对硬盘数据,Parquet性能应该比Kudu更好(larger IO requests)

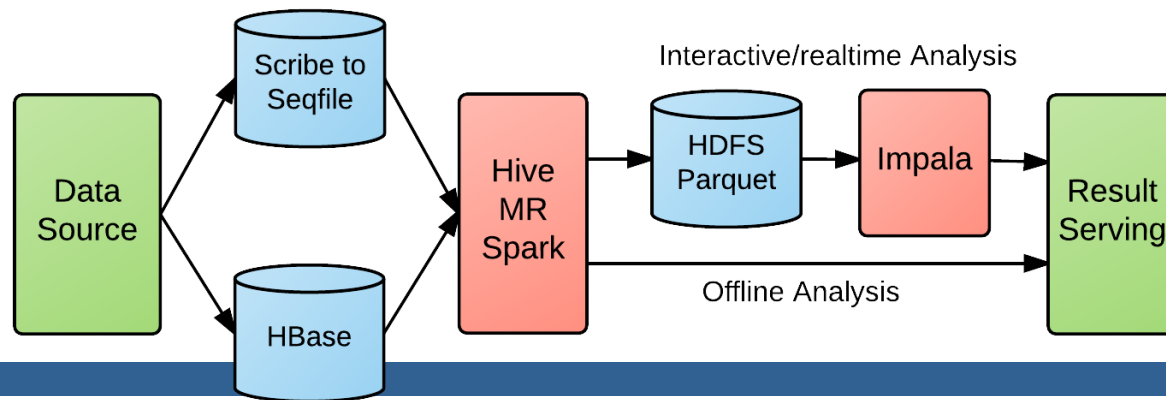




用户案例

大数据分析处理流程 使用Kudu之前

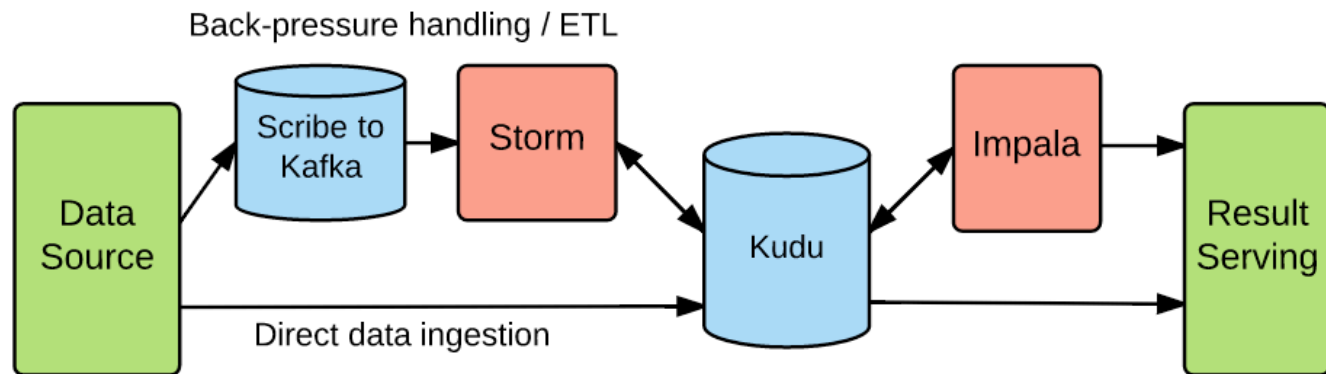
- 在使用Kudu前，小米的大数据分析Pipeline主要有几种：
 1. 数据源-> Scribe将日志输出到HDFS -> MR/Hive/Spark -> HDFS Parquet -> Impala -> 将结果对外服务，这个数据流一般用于分析各种日志
 2. 数据源-> 实时更新HBase/MySQL -> 每天批量导出Parquet-> Impala ->将结果对外服务，这个数据流一般用来分析状态数据，也就是一般需要随机更新的数据比如用户Profile之类的。
- 这两条数据流主要有几个问题：
 1. 数据从生成到落地成能被高效查询的列式存储，整个延时比较大，一般是小时级别到一天
 2. 很多数据的日志到达时间和逻辑时间不一致，一般存在一些随机延时
- 比如很多mobile app统计应用，这些tracing event发生后很可能过一段时间才被后段tracing server收集到
- 对于一些实时分析需求，有一些可以通过流处理来解决，不过没有SQL方便，另外流式处理只能做固定的数据分析，对ad-hoc查询无能为力
- Kudu的特点正好可以来配合Impala搭建实时ad-hoc分析应用



大数据分析处理流程

使用Kudu简化

- 改进后的数据流大概是这样：
 1. 数据源 -> Kafka -> ETL(Storm) -> Kudu -> Impala
 2. 数据源 -> Kudu -> Impala
- 数据流1主要是为需要进一步做ETL的应用使用的，另外Kafka可以作为Buffer，当写吞吐有毛刺时，Kafka可以做一个缓冲。
- 如果应用有严格的实时需求，就是只要数据源写入就必须能够查到，就需要使用数据流2。



来自小米的基准测试



- 从程序跟踪分析应用中抽取的6个真实的查询场景
 - Q1: SELECT COUNT(*)
 - Q2: SELECT hour, COUNT(*) WHERE module = 'foo' GROUP BY HOUR
 - Q3: SELECT hour, COUNT(DISTINCT uid) WHERE module = 'foo' AND app='bar' GROUP BY HOUR
 - Q4: analytics on RPC success rate over all data for one app
 - Q5: same as Q4, but filter by time range
 - Q6: SELECT * WHERE app = ... AND uid = ... ORDER BY ts LIMIT 30 OFFSET 30
- 测试集群：71台服务器
 - 硬件：CPU: E5-2620 2.1GHz * 24 core Memory: 64GB Network: 1Gb Disk: 12 HDD
 - 软件：Hadoop2.6/Impala 2.1/Kudu
- 数据量：一天的服务器端跟踪日志数据
 - 约26亿条数据
 - 每条数据约270字节
 - 表有17个字段，其中5个为主键字段

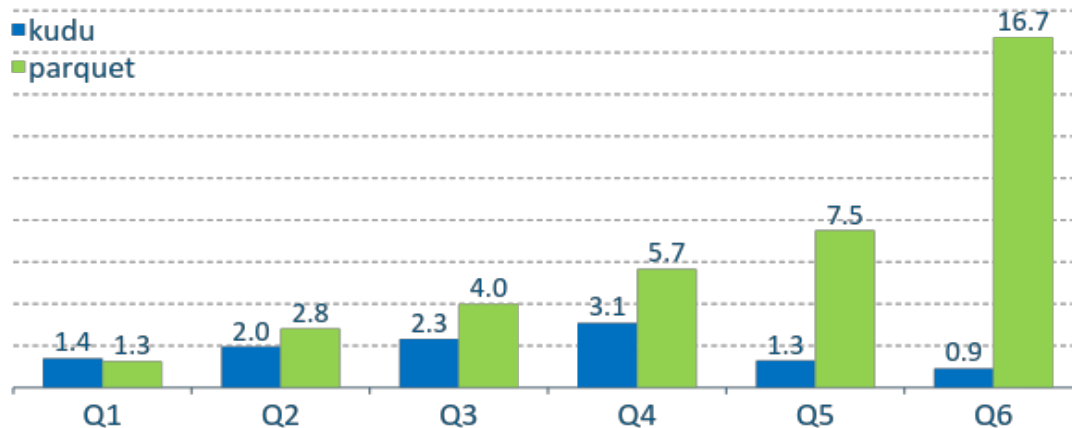


测试结果

- Bulk load using impala (INSERT INTO):

	Total Time(s)	Throughput(Total)	Throughput(per node)
Kudu	961.1	2.8M record/s	39.5k record/s
Parquet	114.6	23.5M record/s	331k records/s

- Query latency (seconds):



- HDFS parquet file replication = 3
- Kudu table replication = 3
- Each query run 5 times then averaged

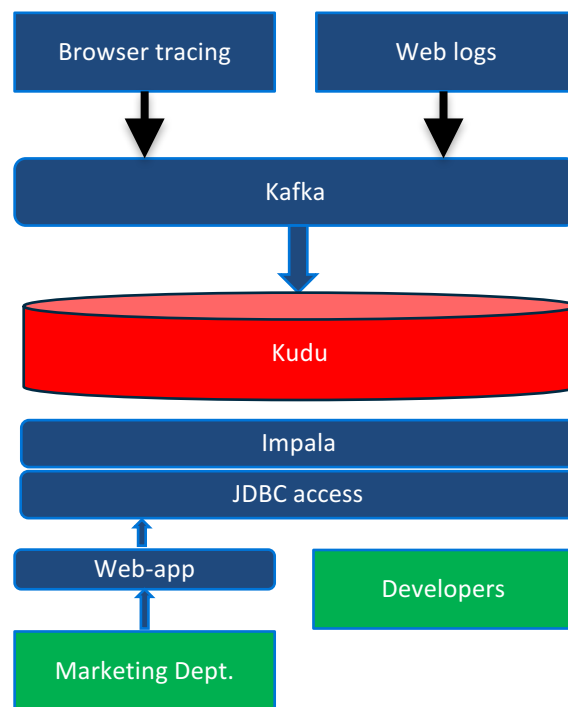


京东案例



Jd.com 中国第二大在线电商

- 使用Kafka实时收集数据
 - 点击流日志
 - 应用/浏览器Trace日志
 - 每条记录约70字段
- 6/18 sale day
 - 150亿笔交易
 - 高峰期每秒一千万条数据插入
 - 集群200台服务器
- 查询使用JDBC -> Impala -> Kudu

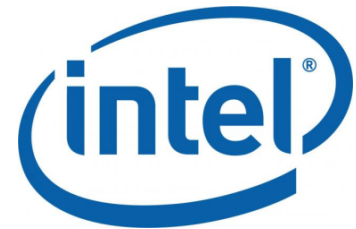




怎样开始?

Apache Kudu社区

cloudera



Personal



作为用户

- <http://getkudu.io>
- kudu-user@googlegroups.com

- Quickstart VM
 - 轻松上手!
 - Impala and Kudu in an easy-to-install VM
- CSD and Parcels
 - For installation on a Cloudera Manager-managed cluster
- 白皮书: getkudu.io/kudu.pdf



作为开发人员

- <http://github.com/cloudera/kudu>
 - 所有的代码首先都会提交到这里
- Public Gerrit: <http://gerrit.cloudera.org>
 - 所有的代码审查会提交到这里
- Public JIRA: <http://issues.cloudera.org>
 - 自2013以来的缺陷修复. Come see our dirty laundry!
- kudu-dev@googlegroups.com
- Apache 2.0 开源代码许可
- 欢迎大家贡献代码和功能!





<http://getkudu.io/>
[@getkudu](#)

Thanks!



@Cloudera中国



@陈飏