从Google Mesa到百度PALO(数仓)_查理王的博客-CSDN博客_google mesa

最近在研究OLAP相关的技术，正好看到Google 2014年的论文《Mesa: Geo-Replicated, Near RealTime, Scalable Data Warehousing》，以及百度最近2017年开源的基于Mesa+Impala的实现系统PALO，本篇就尝试结合起来看下二者，主要是学习介绍性质的文章。

1. Mesa

Mesa是一个Google内部使用的数据仓库系统，从论文的标题可以抓住几个关键词：可实现跨DC复制的、近实时的、可扩展的。这几点算是Mesa的特色所在，同时和Mesa要解决的问题背景有很大关系，Mesa主要解决Google在线广告报表和分析业务，论文里提到的use cases包括reporting, internal auditing, analysis, billing和forecasting等方面。举个例子，广告主需要通过Google的AdWords业务系统查看报告：2017.1整月的预算消费情况，包括所有推广计划（Campaign）的展现量、点击量、消费等指标，这就是一个典型的应用场景。

为了满足Google内部的业务功能需求，需要设计一个data store，它的非功能需求要满足：

1. Atomic Updates. 原子更新。

一个用户的动作，比如一个点击行为，会被影响成百上千的视图的指标，比如影响推广计划、分网站、创意等等一系列具体报表，这个点击行为要么全部生效，要么全不生效，不能存在中间状态。

2. Consistency and Correctness. 一致性和正确性。

强一致性必须保证，可重复读，即使是跨DC也需要保证读出来的一致，这么高的要求和广告系统的严谨性有直接关系。

3. Availability. 高可用。

不能存在单点（SPOF），不能停服（downtime）。

4. Near RealTime Update Throughput. 近实时的高吞吐更新。

系统要支持增量实时更新，吞吐要达到百万行/秒。增量在分钟级即可被查询到的queryability，这么高的要求和广告系统角度来说很必要，每秒钟Google都会有百万级别的广告展现，而广告主或者系统的其他模块需要更短的时间看到报表，辅助决策。

5. Query Performance. 高性能查询。

系统既要支持低延迟的用户报表查询请求，也要支持高吞吐的Ad-hoc即席分析查询。低延迟要保证99分位平响在百毫秒。

6. Scalability. 高扩展。

随着数据量和访问量增量，系统的能力可线性（linear）的增长。

7. Online Data and Metadata Transformation. 在线的schema变更。

业务不断变化，对于schema的变更，包括加表、删表、加列、减列，新建索引，修改物化视图等的都必须不能停服的在线完成，而且不能影响数据更新和查询。

有了需求，那么就一句话总结下Google把Mesa看做一个什么系统。

> Mesa is a distributed, replicated, and highly available data processing, storage, and query system for structured data. Mesa ingests data generated by upstream services, aggregates and persists the data internally, and serves the data via user queries.

翻译下，Mesa是一个分布式、多副本的、高可用的数据处理、存储和查询系统，针对结构化数据。一般数据从上游服务产生（比如一个批次的spark streaming作业产生），在内部做数据的聚合和存储，最终把数据serve到外面供用户查询。

对于Mesa的技术选型，论文里提到了Mesa充分利用了Google内部已有的building blocks，包括Colossus (对应Hadoop的HDFS)、BigTable（对应Hadoop的HBase）和MapReduce。Mesa的存储是多副本的并且分区做sharding的，很好理解，分治策略几乎是分布式系统的必备元素。批量更新，包括大批量，小批量（mini-batch）。使用MVCC机制，每个更新都有个version。为实现跨DC工作，还需要一个分布式一致性技术支持，例如Paxos。

论文里还对比了业界的其他方案，比如基于数据立方体cube的方案，很难做近实时更新（当年是了，现在kylin也支持了），Google内部的系统中BigTable不支持跨行事务，Megastore、Spanner和F1都是OLTP系统，不支持海量数据的高吞吐写入。

下面进入正题，在海量数据规模下，实时性和吞吐率两个指标，鱼与熊掌不可兼得，Mesa基于广告数据可聚合性的特质，从存储，查询等角度进行了大量针对性的设计，那么Mesa到底提出了什么创新的设计来应对它提出的需求呢？其实就两方面，1）存储设计，2）系统架构。其中我认识1）是这个论文最大的contribution。

1.1 存储设计

1.1.1 数据模型（data model）

Mesa仅支持结构化数据，逻辑上存储在一张表（table）里，表包括很多列，表都有一个schema，和传统的数据库类似，schema会定义各个列的类型，比如int32、int64、string等。

Mesa的列要能分成两类，分别是维度列（dimensional attributes）和指标列（measure attributes），这实际可以看做是一种KV模型，Keys就是维度，Values就是指标。

同时指标列需要定义一个聚合函数aggregation function，例如SUM,MIN,MAX,COUNT等等，用于作用于Key相同的记录，做聚合使用，聚合函数必须满足结合律，可以选择性满足交换律。

Mesa中定义的索引Index其实只能是被动的符合Key的顺序（因为物理上没有多余的存储索引，全靠数据有序存储，后面存储格式章节会细讲）。

一个记录或者用户行为，叫做single fact会原子地、一致的影响多个物化视图（materilized view），物化视图一般利用维度列做上卷表（roll-up），这样就可以做多维分析（MOLAP）的下钻（drill down）和上卷（roll up）查询了。

Google中Mesa存储了上千张表，每张表最多几百列。

下图是论文中的例子，三张典型的表。

Table A的维度列包括Date, PublisherId, Country，指标列是Clicks, Cost，聚合函数是SUM。

Table B的维度列包括Date, AdvertiserId, Country，指标列是Clicks, Cost，聚合函数是SUM。

Table C是Table B的物化视图，维度列是AdvertiserId, Country，指标列是Clicks Cost，聚合函数是SUM。

Date	PublisherId	Country	Clicks	Cost
2013/12/31	100	US	10	32
2014/01/01	100	US	205	103
2014/01/01	200	UK	100	50

(a) Mesa table A

Date	AdvertiserId	Country	Clicks	Cost
2013/12/31	1	US	10	32
2014/01/01	1	US	5	3
2014/01/01	2	UK	100	50
2014/01/01	2	US	200	100

(b) Mesa table B

AdvertiserId	Country	Clicks	Cost
1	US	15	35
2	UK	100	50
2	US	200	100

(c) Mesa table C

Figure 1: Three related Mesa tables

1.1.2 数据更新和查询

为实现高吞吐的更新，Mesa必须按照批量的方式来实现，这些更新的小数据集合通常从upstream系统来，一般是分钟级别产生一个，这个可以理解为Storm或者Spark Streaming产生的数据。所有的更新批次就是串行处理的。每个更新批次都会带一个自增的版本号，其实这就是MVCC机制，这样就可以做到无锁的更新，对于查询就需要指定一个版本号。同时，Mesa要求查询除了包含版本号，还得有一个Predicate，也就是在Key space上做filter的谓词条件。

论文中举了一个例子，如下图所示，在刚刚的数据模型中Table A和Table B是通过两个更新的批次来的，经历了两次版本变化而来，可以看做是fact table。同时Table C是Table B的物化视图，rollup的SQL如下：

```
SELECT SUM(Clicks), SUM(Cost) GROUP BY AdvertiserId,
Country
```

对于Table B的每个mini-batch更新，物化视图都保持了和fact table的一致原子更新。

Date	PublisherId	Country	Clicks	Cost
2013/12/31	100	US	+10	+32
2014/01/01	100	US	+150	+80
2014/01/01	200	UK	+40	+20

(a) Update version 0 for Mesa table A

Date	AdvertiserId	Country	Clicks	Cost
2013/12/31	1	US	+10	+32
2014/01/01	2	UK	+40	+20
2014/01/01	2	US	+150	+80

(b) Update version 0 for Mesa table B

Date	PublisherId	Country	Clicks	Cost
2014/01/01	100	US	+55	+23
2014/01/01	200	UK	+60	+30

(c) Update version 1 for Mesa table A

Date	AdvertiserId	Country	Clicks	Cost
2013/01/01	1	US	+5	+3
2014/01/01	2	UK	+60	+30
2014/01/01	2	US	+50	+20

(d) Update version 1 for Mesa table B

Figure 2: Two Mesa updates

另外对于一些backfill和回滚数据的需求，比如某天的数据有问题，通常广告领域就是反作弊后而后知，那么Mesa提出了negative facts的概念，也就是做减法即可，从最终一致的角度来做回滚。

1.1.3 数据版本化

上一节提到了每个批次都版本化的概念，但是具体实现的困难要考虑：

1）每个版本独立存储很昂贵，浪费空间（而聚合后的数据往往更加的小）。

2）在查询的时候going over所有的的版本并且做聚集，考虑每个版本是分钟级生成了，那么每天的量也会很大，这种expensive的操作很影响在线的查询延时。

3）傻傻的针对每一次更新，都在所有的版本上做预聚合，也非常的expensive。（看看bigtable、leveldb的多级存储结构，就知道merge sort实时做每一个批次，系统是吃不消的）

为了解决这三个问题，Mesa的方案是：

提出Delta的概念，对于每次的更新，相同的Key都做预聚合，形成一个独立的Singleton delta，一个Singleton delta包括很多rows，以及一个version = [V1, V2]。在某些场景下可能会不存储原始数据，也就不能drill down到最细的粒度了，但是做了上卷所以会非常节省空间。

Delta之间可以做merge，例如[V1, V2]和[V2+1, V3]可以合并成[V1, V3]，下面物理存储章节会提到每个delta内部数据都是有序的，所有只需要线性时间复杂度（linear time），即最简单的merge sorted array就可以合并好两个delta。

Mesa要求查询指定的版本号不能无限的小，需要在一个时间范围前（比如24小时之内），这是因为还会存在一个Base compaction的策略，用来归并所有的历史delta，这和bigtable中的概念一样，主要从查询效率来说，通过合并小文件来减少随机I/O的次数。合并了base之后，这些老版本的delta就可以删除掉了。

但是base compaction往往是天级别做，因为很expensive，但是考虑分钟级别的导入，也会有成百上千的小文件需要在runtime的时候做查询，也就多了非常多的随机I/O。为了加速实时的在线查询，并且平衡导入的高吞吐，Mesa提出了多级的compaction策略，这里Mesa实际用了两级存储，会存在一个cumulative compaction的过程，例如每当积累到10个Singleton delta，就做一次小的多路归并，合并成一个cumulative delta。再积累了10个之后再做一次多路归并即可。

举个例子，下图的中Base是24小时之前的文件，天粒度聚合而成。存在61-92这些个singleton delta，它们都是每个mini-batch导入的预聚合好的数据，如果不存在cumulative delta，那么假如查询条件的版本指定到91，那么就需要base，外加61-91这32次的随机I/O，这种延迟明显太大了，那么如果有了cumulative就可以按照最短路径的算法，做一次查询只需要base，加61-90这个cumulative，加91这一个delta，一共3次随机I/O就可以查询出来结果。

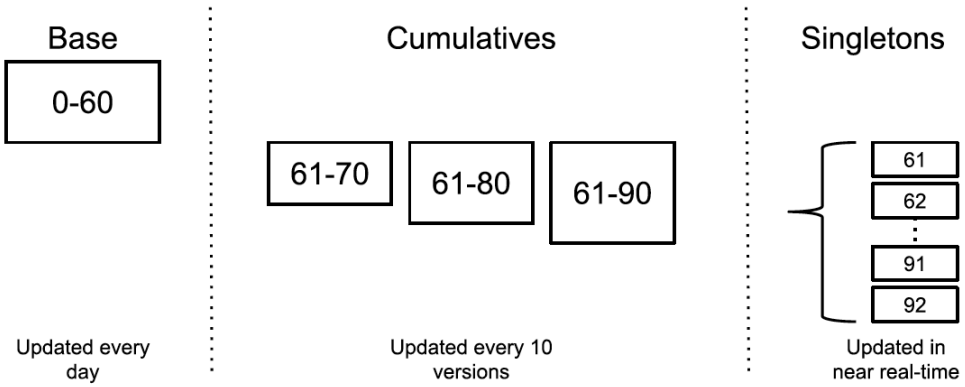


Figure 3: A two level delta compaction policy

1.1.4 物理存储格式

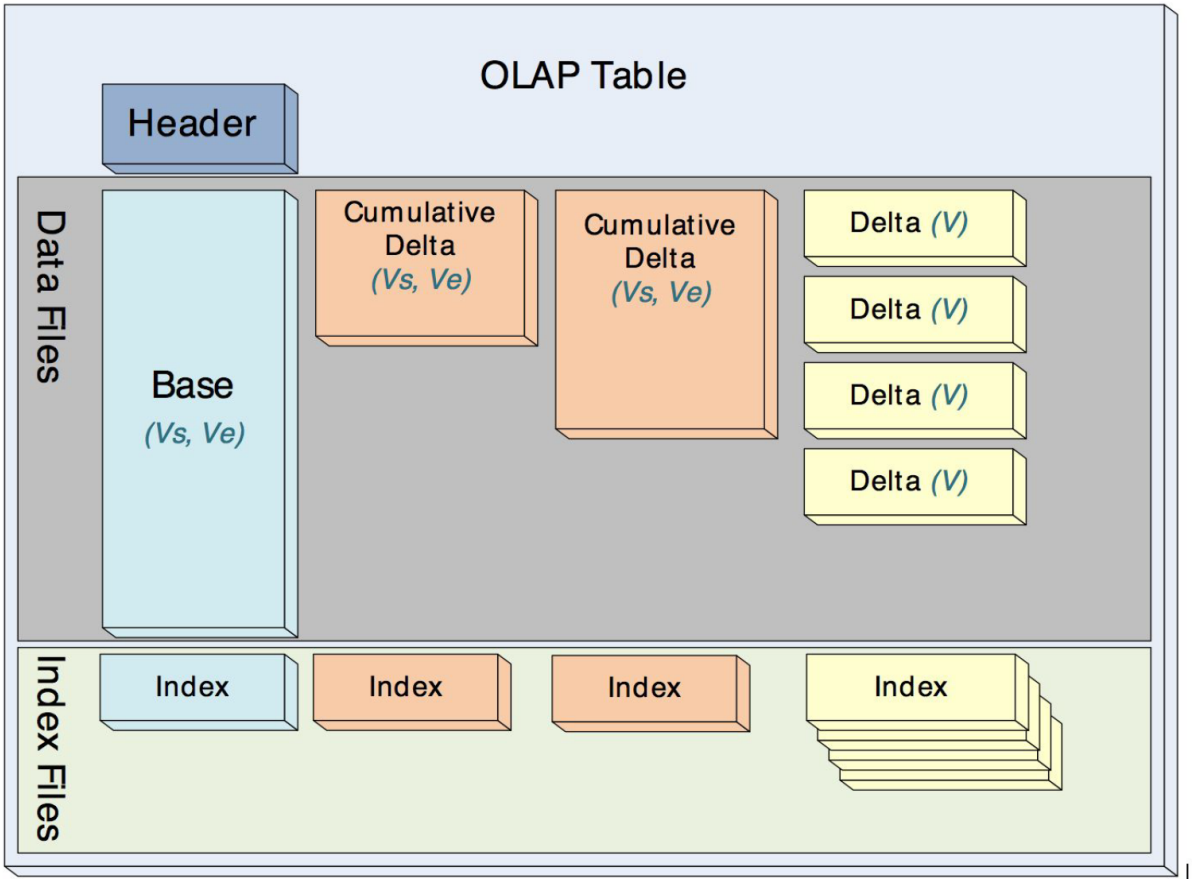
Mesa中的delta、cumulative和base在物理存储上格式一样，它们都是immutable的，这样就很方便做mini-batch的增量的更新，而不至于很影响吞吐，因为compaction过程都是异步的。

Mesa的存储格式要尽可能的节约空间，同时支持点查（fast seeking to a specific key），Mesa设计了索引Index和数据Data文件，物理上Index和Data数据是分开的，每个Index实际就是Short Key的顺序排列外加offset偏移量，每个Data就是Key+Value的顺序存储。每个表都是这样多个Index和多个Data的集合。

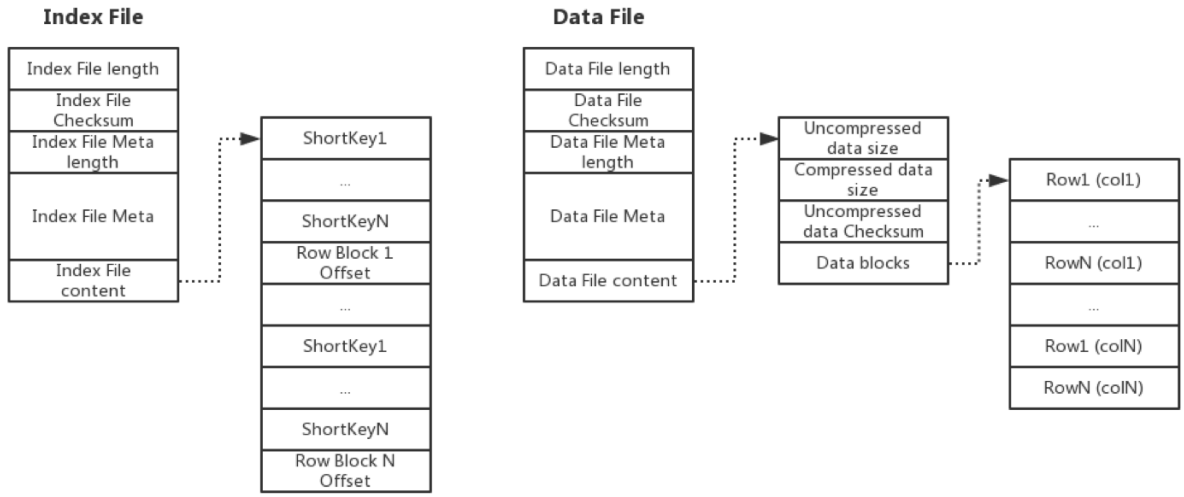
Mesa对于存储格式并没有展开说很多，但是提到了一些重点。Data文件中的数据按照Key有序排列，按行切块形成row block，按列存储，这种格式和现在的ORC、Parquet很像，Row Block的大小一般不大，它是从磁盘load到内存的最小粒度，使用这种格式很容易做压缩，因为每一列的格式都是相同的，可以做一些轻量级的编码比如RLE、字典编码、Bitpacking等，在这个基础之上再做重量级的压缩，比如LZO、Snappy、GZIP等，就可以实现压缩比很高的存储。

Index文件存储了Short Key，Short Key关联一个Row Block，这样只需要把Index加载到内存，在Index文件中做naive的二分查找定位Row Block在Data文件中偏移量offset，然后load Row Block加载到内存，再做一些Predicate filter的Scan，对于Key相同的按照聚合函数做聚合即可把结果查到。

对于Mesa的存储模型，实际的物理上的文件可能会存在多个，如下图所示。



每一对Index file和Data file的格式如果实现的最简单，可以如下图所示。如果按列存储可以设计的更丰富，比如Parquet的数据存储格式就为了支持嵌套的数据结构、方便做谓词下推做了很多的设计。



1.2 系统架构

这一部分分为两块，第一是单DC（Datacenter）部署，第二是跨DC部署。这里不得不说Google的论文虽然抛出来的，但是细节都是很模糊的。

1.2.1 单DC部署

两个子系统Update/Maintenance Subsystem和Query Subsystem分开，这样也是为了满足其高吞吐准实时导入，低延迟查询的系统要求而做的技术选型。

Update/Maintenance Subsystem

主要职责包括，

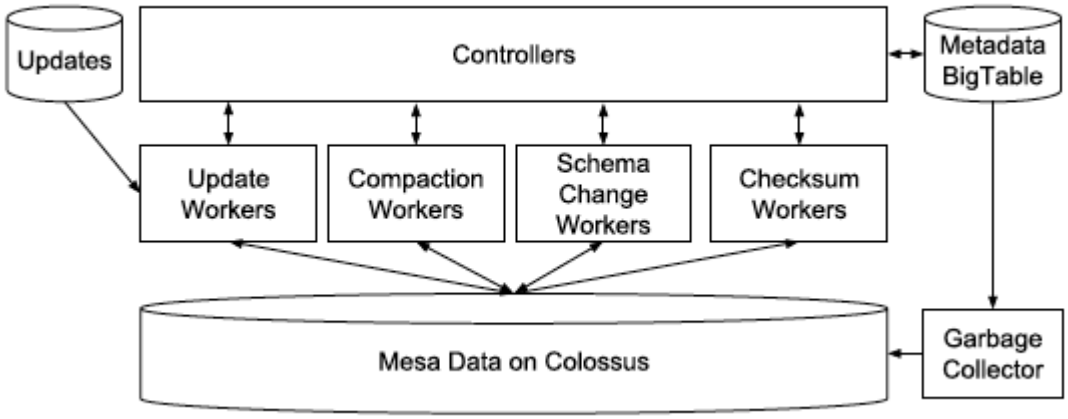
1）加载update，并且按照存储模型保存到Mesa的物理存储上。

2）执行多级的compaction。

3）在线做schema change。

4）执行一些表的checksum检查。

系统架构图如下：



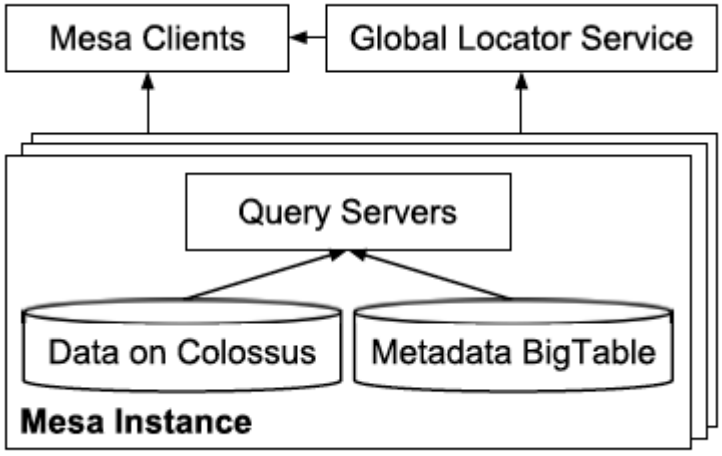
Controller可以看做是一个metadata的cache，worker的调度和queue的管理都它来。所有的metadata都存储在BigTable中，所以Controller可以是一个无状态的stateless的服务。Controller管理了4类worker，就是刚在提到的4个职责，各对应4种worker，Controller通过RPC接收外部的请求，然后把任务Task投递到queue中。

Worker采用隔离的策略，4种职责各4个Worker Pool。Worker采用“拉”的策略，从queue中取任务，然后执行，例如加载update，取到任务后从任务的metadata中获取原始数据（比如CSV文件）存储的位置以及做一些数据校验工作，然后做预聚合形成Singleton delta，存储在Google的HDFS即Colussus中，然后再更新metadata commit这个版本已经incorperate到系统中形成了delta，外部可供查询。图中还有一个GC（Garbage Collector），这个就是Worker销毁的，防止Worker死掉从而saturate整个Worker Pool。

这套Controller/Worker的架构，从下面要说的查询系统中分离出来，充分体现了分治的策略，互不干扰。这里Table可能很大，所以Controller也是做了sharding的，来更好的做扩展，同时Controller不存在单点（SPOF），一旦有问题handoff到另外一个stand by即可，因为所有的metadata都在BigTable中存储。

Query Subsystem

查询子系统架构如下图所示。



一次查询的步骤如下：获取用户请求，例如SQL，根据metadata，做校验、语法解析、词法解析、查询计划生成等，决定了需要查询哪些文件；发起查询请求，并且做归并聚合处理；将结果转换为客户端需要的格式，响应回去。

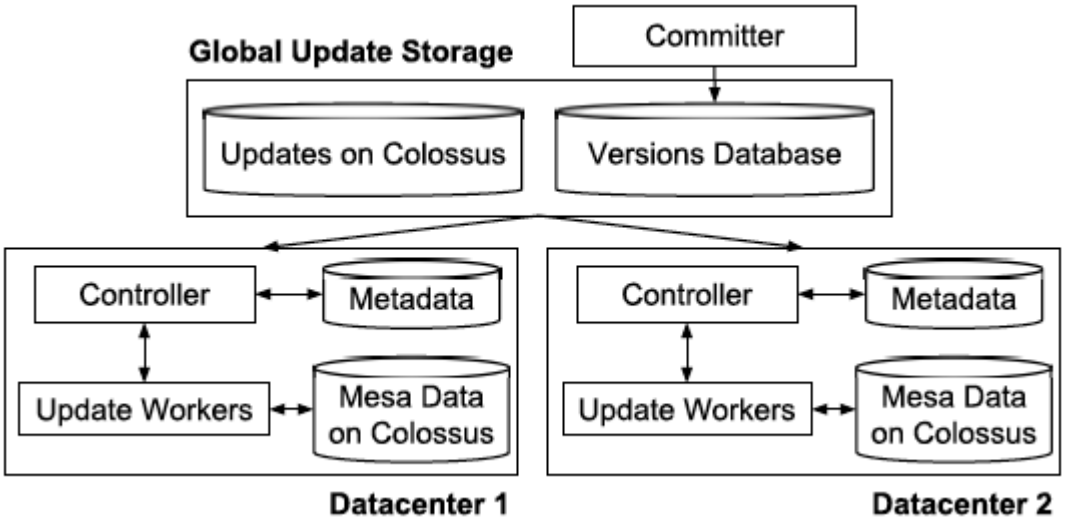
Mesa作为一个简单的通用存储查询系统，只提供了有限的语义，包括filter和group-by，剩下的Higher-level的语义包括JOIN、子查询等等都由上层系统做，比如Google的Dremel或者MySQL。

这里论文还提到查询系统的lable化，因为在线的reporting要求低延迟，一般是点查，而Ad-hoc的分析查询一般要求高吞吐，为了防止二者互相干扰，还是采用了分治策略，把不同的query system贴上不同的label，这样在查询的时候可以有选择的路由。

图中的global locator service是每个query system启动时候去注册的，这样client就可以根据label或者要查询的表路由到正确的query server上。

1.2.2 跨DC部署

架构图如下。



由于每个更新批次都是版本化的，所以采用MVCC机制，存在一个committer做upstream service和mesa的桥梁，对于每个update都保存在一个versions database – a globally replicated and consistent data store build on top of the Paxos consensus algorithm，实际可以看做spanner/F1中，然后依次的下发各个DC，每个DC内部都是刚刚提到的架构，Controller负责监听新的version，拉取update并且更新本DC，成功后notify versions database，committer不断的检查是否commit criteria满足了，比如5个里面3个成功了，那么commit这个version，再继续下个批次的更新。

这种方案的好处在于，多个DC无锁化和异步化，用以满足高吞吐的导入和低延迟的查询。

最后，论文还提到了一些Enhancements，包括query server的，使用MapReduce并行化处理worker任务的，如何做在线schema变更的，如果防止数据损坏（包括存储的checksum，和异步的检查等等）。一些lesson learned，可以说是分布式系统设计里面的common patter和容易踩到的坑的总结，可以好好读读。剩下的就是metrics对比了，这里不再赘述。

基本来说，Mesa论文还是很偏理论的，并且集中聚焦在数据模型上，这点我认为是贡献最大的，下面要讲的PALO也是借鉴了其数据模型。

2. PALO

2.1 简介

说完了Mesa，说说PALO，PALO是百度2017年开源的项目，由于笔者之前有百度6年的工作的经历，也使用过该项目的前身OlapEngine，所以这里简单的介绍下。

Palo名字的由来是“玩转OLAP”，把OLAP倒过来就是PALO。还是抓住github首页的介绍关键词：

> A MPP-based Interactive Data Analysis SQL DB

PALO是基于MPP架构的，一个交互式的数据分析的SQL DB。注意其定位是一个DB，而不是像大数据领域的MPP比如开源的Presto、Impala那样的纯查询引擎（query engine），所以PALO即包含存储引擎，也包含查询引擎（这里借鉴了Impala），而Presto、Impala的存储都采用了开源的格式和存储引擎，比如ORC、Parquet等，PALO的存储格式借鉴了Mesa，所以这就是PALO和Mesa的联系，PALO=Mesa的存储引擎+Impala查询引擎的开源实现。

百度内部一直有各种需求，比如statistics广告统计报表就是典型。要支持增量更新，近实时，还需要提供低延迟的查询，又要给批量的、高吞吐的Ad-hoc查询做多维分析（比如BI系统）。过去用Mysql、Doris支持，但是都不理想。 而大家真正需要的是一个MPP SQL Engine。所以大家就有的搞MPP类的SparkSQL、Impala、Presto、Drill，有的搞MOLAP类的Druid、Kylin，有的考虑买商业数据库（比如Greenplum，Vertica，AtScale），有的考虑用Amazon Redshift、Google BigQuery，有的尝试了MonetDB等，所有方案基本都是因为较为复杂，或者不免费，或者不稳定，或者并不能很好的各种满足需求，所以才逐步研发了PALO。

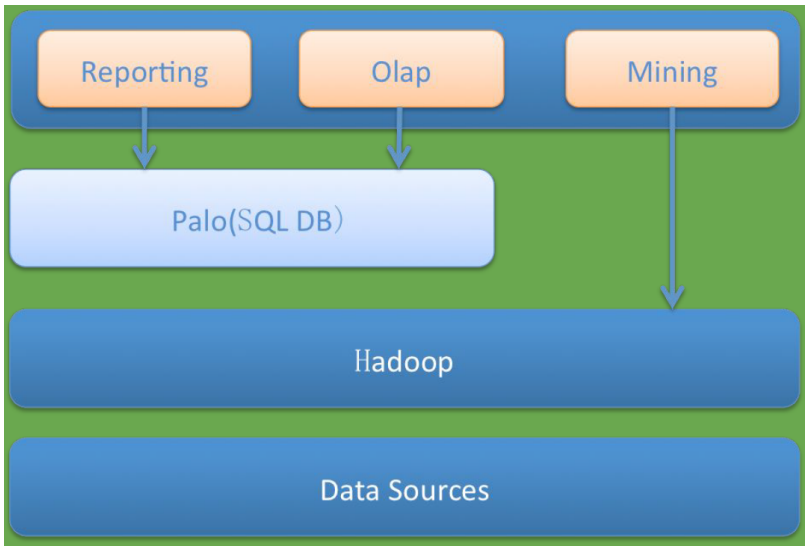
PALO是面向百TB ~ PB级别的查询的产品，仅支持结构化数据，可供毫秒/秒级分析，是由百度大数据部团队研发的，经历了三代的产品Doris -> OlapEngine ->PALO，其中Doris是2012年之前广告团队采用的报表查询系统，而OlapEngine是基于MySQL的一个查询引擎，类似InnoDB或者MyISAM，也是借鉴了Mesa，最早是James Peng在凤巢、网盟实施指导研发的项目，2014着手改造OlapEngine到PALO，PALO代表了当下state of the art的该类系统，目前广泛应用于百度，150+产品线使用，600+台机器，单一业务最大百TB。

PALO也可以看做是一个数据仓库DW，因为借鉴的Mesa的模型，所以兼具低延迟的点查和高吞吐的Ad-hoc查询功能。PALO支持batch loading和mini-batch即近实时的loading。和其他SQL-on-hadoop不同的是，PALO官方给出的特殊卖点是：

1）低成本的构建稳定可扩展的OLAP系统，开源免费并且可工作在普通机器上。

2）简单易用的单一系统，拒绝hybrid architectures，不依赖Hadoop那套，架构简单，并且可以使用MySQL协议接入。

下图展示了PALO的定位。

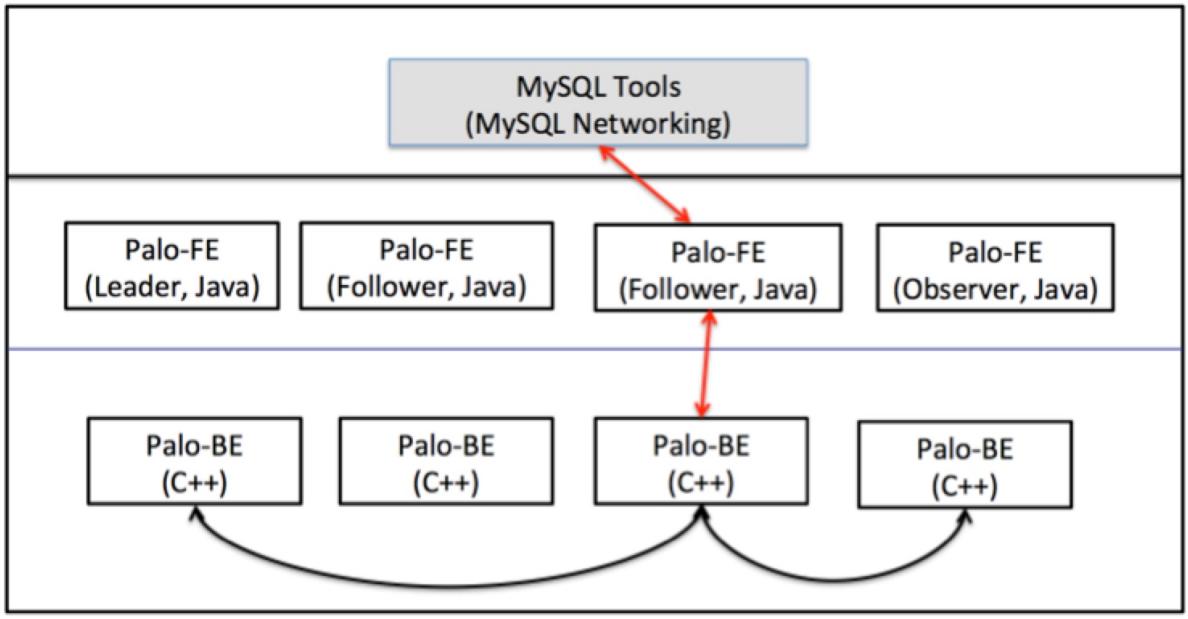


2.2 PALO的特点

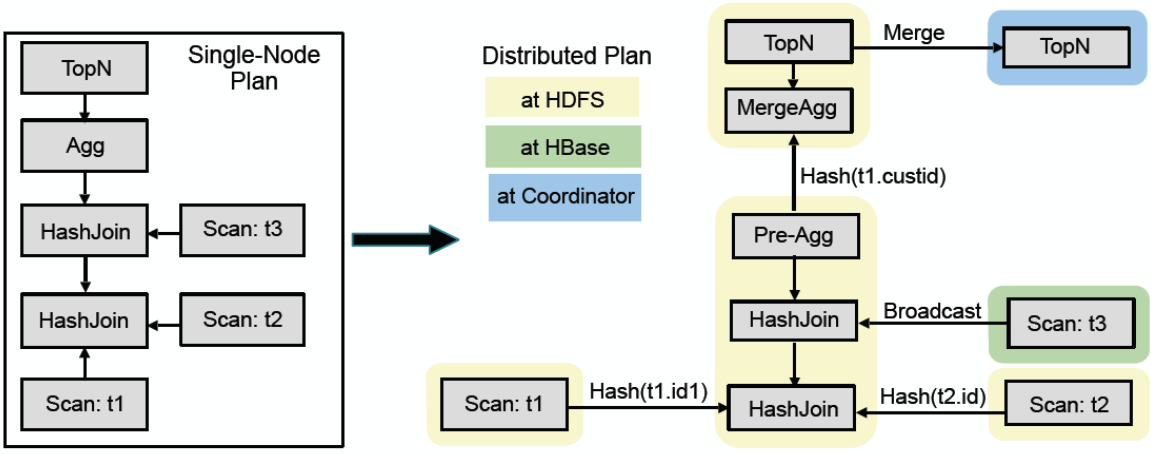
- 1) 高性能的行列存储引擎
- 2) 小批量更新，批量原子提交，多版本支持
- 3) 高效的分布式数据导入
- 4) 支持Rollup Table, Scheme Change, Data Recovery
- 5) 较完备的分布式管理框架，使得整个PALO易用易运维
- 6) Range partition: 全局key排序，自动分裂还没有满足
- 7) MPP Query Engine – 低并发大查询 + 高并发低延迟小查询
- 8) 调度和资源隔离还在完善，支持优先级划分和多租户
- 9) 存储分级支持，老数据用SATA，热的新数据用SSD
- 10) 实现了Mysql网络协议，可以很容易与各种上层工具打通
- 11) 支持多表join（这点由于自己实现了查询引擎，所以弥补了Mesa存储引擎的不能实现的）
- 12) Rollup表智能选择
- 13) 支持谓词下推

2.3 PALO的系统架构

架构图如下。



FE包含query coordinator and catalog manager。Query coordinator接收SQL请求，根据元数据，编译成query plan，然后建立query plan fragments，生成一个DAG执行的pipeline用于分发给BE执行（如下图所示，是impala中的query plan到实际物理执行的DAG的转换，可以把at HDFS和at HBase看做是BE执行的节点），Query coordinator统筹管理调度执行，这相当于Impalad的Query coordinator。Catalog manager存metadata，包括数据库、表、分区、副本位置等等。多个FE可以保证HA和负载均衡。



(图片来源：Impala论文)

FE是非对称的架构，这和Hive、Impala等的中心架构不同，所有的metadata不是存储在一个公共的服务上，在FE当中做了一个基于Paxos-like consensus算法的复制状态机，这样可以可靠的存储数据，并且检具扩展性，满足高并发的查询。FE分为三个角色，包括leader, follower和observer，leader负责写入，follower用分布式一致性算法做同步日志，quorum方式使得follower成功，然后再commit。高并发场景下，多个follower会有问题，就像Zookeeper一样，所以引入了observer角色，专门做异步同步。FE中的复制状态机用Berkeley DB java version实现，FE和BE的通信使用Thrift框架。

PALO易用性好的一个方面也体现在其FE兼容MySQL协议上，也就是可以用MySQL client，JDBC等直接连接FE，发起DML、DDL语句，这样也就非常好的可以和BI系统集成。

BE负责存储数据、执行query fragments（这是impala论文里面提到的），BE就是一个query engine。BE没有依赖任何分布式存储，例如HDFS，而是自己负责管理多个副本，副本数量是可以指定的，由写入的updater负责写入多个副本，文件系统全是PALO自己管理的，所以PALO是一个DB，而不是一个其他大数据开源产品，例如Presto、Impala那样的查询引擎。FE在做query调度的时候会考虑数据的本地性（locality）以及最大化scan的能力。多个BE部署可以达到scalability and fault-tolerance。

PALO中的数据是水平分区的。按照桶bucket分区，但是Single-level不管是hash或者range都可能会有问题，比如hash(userid)或者range(date)都会不均匀，存在数据倾斜现象，所以PALO把这个shard的策略做成了可以支持Two-level。第一级是range partitioning，一般采用日期，方便做冷热数据区分不同的存储介质（SATA或者SSD），第二级是hash partitioning，可以看做是分桶，所以PALO要求使用者做好这个分区分桶。如果使用者执意把1TB大小的数据放到一个桶里面，那么这种不合理的使用和规划，会影响PALO BE的查询性能，因为BE执行一个Scan query fragments就是去按照Mesa的模型读取Index和Data数据，一个MPP的思想就是并行化，这种大分桶也就限制了系统发挥能力，目前PALO还不支持自动分裂，不像HBase那这样，所以这个分区和分桶的策略是做schema design的时候要提前考虑好的。

这里要提下PALO基于Mesa的数据模型和存储模型，但是Mesa需要区分维度列和指标列，而一个通用的OLAP系统往往不能区分这些列，所以PALO为了做一个通用的OLAP，可以做到不区分这个维度列和指标列。即使不区分维度列和指标列，但是PALO借鉴了Mesa的存储模型，所以如果没有Key Space，那么就必须指定一个排序列，用于存储需要。

PALO在性能追求上也是尽量做到最好。PALO的核心BE是使用C++开发的，这和Impala的思路很类似，Java的GC和内存控制一直是诟病，为了追求性能的极致，PALO选择了C++作为开发语言更好的控制。同时PALO支持一些流行的OLAP优化手段，包括向量化执行和JIT，C++使用LLVM。PALO支持分区剪枝（Partition pruning），支持bloomfilter做某列的索引，同时Index中会存储MIN/MAX等基本信息，方便做Predicate pushdown谓词下推。由于基于Mesa，会利用预聚合的方式，使用物化视图和做上卷表，在某些场景下可以大大加速查询效率。综上，这些都加速了OLAP的查询性能。

2.4 PALO总结

这里由于笔者的精力有限，还没有大规模使用PALO，暂且对于PALO的认识就限上面所述。作为一个前老百度人，对于百度开源产品还是很看好的，在公司内这可是明星级别的并且广泛应用的产品，虽然现在刚刚开源，在产品化、文档、工具、排查、稳定性等方面还需要完善和经受考验，但是如果这个产品可以解决大家的痛点，作为building blocks可以帮助企业快速解决问题，我想社区的力量是巨大的，一定会把它发扬光大好，希望PALO未来的路越走越好。