2018
HBase技术总结

中国HBase技术社区 出品 http://hbase.group

生态篇

案例篇

技术篇

平台篇

中国 HBase 技术社区微信公众号

HBase+Spark 钉钉技术社区群

谨以此书献给所有 HBase 爱好者

HBase 是一个高性能，并且支持无限水平扩展的在线数据库，其存储计算分离的特性非常好地适应了目前的趋势，并且在国内大公司内都被广泛地应用，具有非常好的生态，是构建大数据系统的不二选择。

杨文龙 HBase PMC&HBase Committer

本文档的制作特别感谢以下同学（微信名，排名不分先后）

快看，阳光 中国人寿保险股份有限公司 大数据研发工程师

米麒麟 陆金所

杨继飞 点评微生活 大数据研发工程师

诸葛子房　京东 大数据研发工程师

赵昆鹏 万达信息股份有限公司 大数据研发工程师

张少华

沈宗强

扎啤

过往记忆

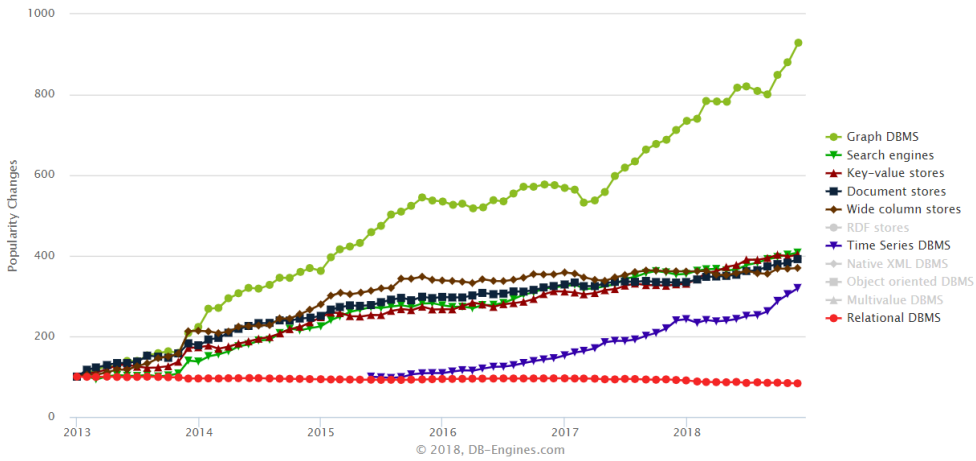
封神

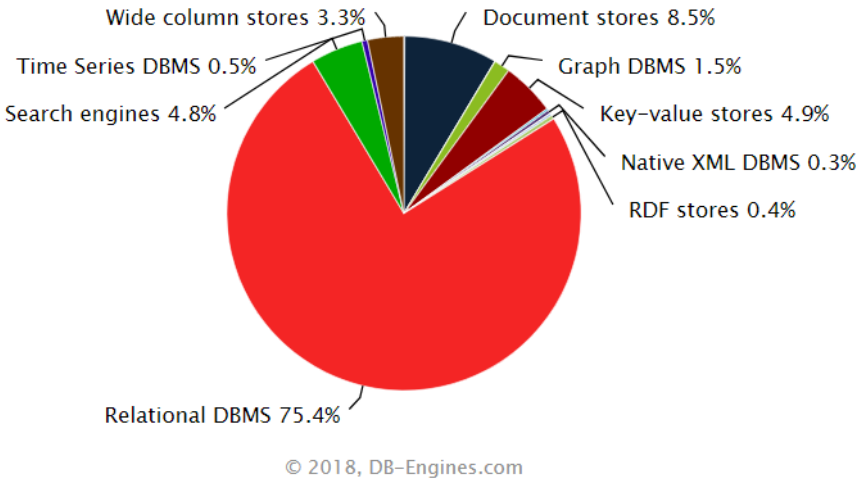
所在

同时感谢中国 HBase 技术社区所有小伙伴以及 2018 年 HBase Meetup 各位讲师。

HBase 生态介绍

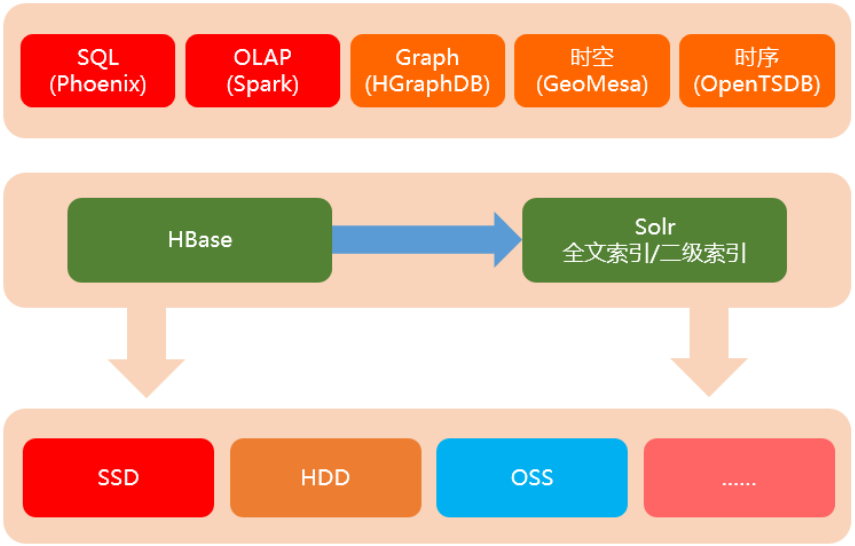
我们都知道，HBase 是受 Google 公布的 BigTable 论文而产生的一种分布式、多版本、面向列的开源 KV 数据库。HBase 稀疏矩阵的设计使得其特别适合存储非结构化的数据，比如用户画像、日志以及消息等数据。但是随着业务的快速发展，我们面临着各种新挑战和新需求，数据格式也随着业务的发展变得多种多样，其中包括：KV 数据、关系数据、文档数据、图数据以及时空时序等数据。而且随着时间的推移，各种数据占比越来越大，如下图所示：





从上图可以看出，从 2013 年开始，关系型数据的总体占比在逐年下降；而图数据、搜索数据、KV 数据、文档数据以及时序数据等却在逐年上升。到了 2018 年，关系型数据的占比已经由 2013 年的 90%多下降到 2018 年的 75.4%。

面对如此多样的数据，我们急需一种系统，能够存储这些逐年增长的数据。所以很有必要在 HBase 之上引入各种组件，使得 HBase 能够支持 SQL、时序、时空、图、全文检索能力、及复杂分析。所以，完整的 HBase 生态如下：



从最底下开始看，这里面可以根据不同的需求选择不同的存储介质。比如热数据我们可以存储在 SSD 中；温数据存储在 HDD 中，冷数据存储在 OSS 中。中间一层就是 HBase 以及 Solr。最上层是解决各种场景的组件。下面简单介绍下每种组件的作用。

1. Phoenix：主要提供 SQL 的方式来查询 HBase 里面的数据。一般能够在毫秒级别返回，比较适合 OLTP 以及操作性分析等场景。目前 Phoenix 支持 ANSI 92 语法，支持构建二级索引。
2. Spark：很多企业使用 HBase 存储海量数据，一种常见的需求就是对这些数据进行离线分析，我们可以使用 Spark（Spark SQL） 来实现海量数据的离线分析需求。同时，Spark 还支持实时流计算，我们可以使用 HBase+Spark Streaming 解决实时广告推荐等需求。
3. HGraphDB：HGraphDB 是分布式图数据库，可以使用其进行图 OLTP 查询，同时我们还可以结合 Spark GraphFrames 实现图分析需求。通过依托图关联技术，帮助金融机构有效识别隐藏在网络中的黑色信息，在团伙欺诈、黑中介识别等。
4. GeoMesa：目前基于 NoSQL 数据库的时空数据引擎中功能最丰富、社区贡献人数最多的开源系统。提供高效时空索引，支持点、线、面等空间要素存储，百亿级数据实现毫秒(ms)级响应；提供轨迹查询、区域分布统计、区域查询、

密度分析、聚合、OD 分析等常用的时空分析功能;提供基于 Spark SQL、REST、GeoJSON、OGC 服务等多种操作方式，方便地理信息互操作。

5. OpenTSDB：基于 HBase 的分布式的，可伸缩的时间序列数据库。适合做监控系统；譬如收集大规模集群（包括网络设备、操作系统、应用程序）的监控数据并进行存储，查询。
6. Solr：原生的 HBase 只提供了 Rowkey 单主键，如果我们需要对 Rowkey 之外的列进行查找，这时候就会有问题。幸好我们可以使用 Solr 来建立二级索引/全文索引充分满足我们的查询需求。

通过在 HBase 之上引入了各种组件之后，使得 HBase 应用场景得到了极大的扩展，满足了监控、车联网、风控、实时推荐、政企、人工智能等场景的需求。

目前阿里云提供了 HBase 及 X-Pack 组件，其 X-Pack 组件形式和上面的 HBase 生态很类似；除此之外，X-Pack 组件还针对 HBase 做了大量的优化，满足丰富业务处理需求、同时更加易用、更加强大功能。

HBase 进化之从 NoSQL 到 NewSQL，凤凰涅槃成就 Phoenix

陈明 阿里巴巴 技术专家

1.背景概述

近些年来，数据爆炸或者大数据成为 IT 行业发展的高频词汇，传统单机数据库处理数据能力的瓶颈成为摆在 IT 工程师面前十分常见且亟待解决的问题。单机硬件存储容量和计算力的增长远远赶不上数据的增长。在单机软件中，数据库是数据相关处理技术的集大成者，集合了数据存储、数据实时读写、在线事务和数据分析等技术，并通过主备、多活等方案保证了可靠性。但是，在实际业务场景中，我们往往并没有同时用到所有数据库提供的能力，这也为我们解决业务中实际遇到的大数据难题提供了解决思路。

当前的大数据系统大多是通过分布式原理，重点解决某个方面或者某些方面的困境和难题，比如 HDFS 解决了大数据存储的问题，MapReduce 解决了大数据量复杂分析和计算的难题，HBase 解决了实时读写的问题。下面笔者将介绍 HBase 从解决实时读写问题出发，逐步进化，从 NoSQL 发展到 NewSQL 的过程和最新进展。

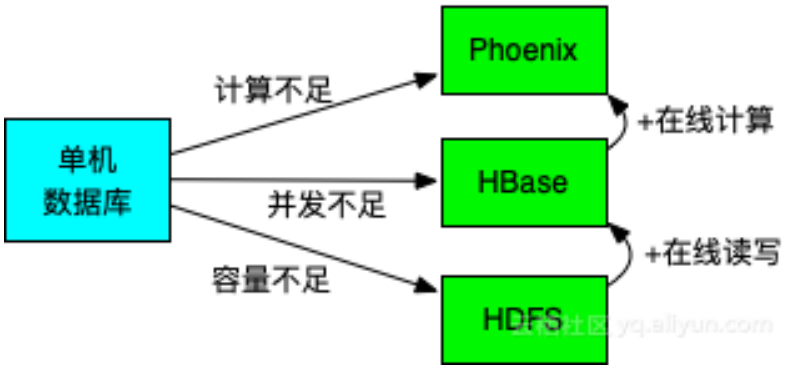


图 1 NoSQL 到 NewSQL

单纯从解决大数据实时读写问题角度，最初的 HBase 系统可以去掉很多传统数据库无关的功能，比如事务，SQL 表达与分析等，重点关注于分布式系统的扩展

性，容错性，分布式缓存的设计，读写性能的优化以及毛刺的减少等方面。这也就是 NoSQL 最初的含义，解决大数据的实时存取的核心问题是第一位的，提供简单的 Get，Put，Scan 接口就可以解决用户的燃眉之急。通过第一阶段的努力，HBase 成为了优秀的大数据实时存取引擎，传统数据库的容量问题解决了，HDFS 实时性不够的问题也解决了。

走过了从无到有的第一阶段，我们需要让 HBase 更强大，更好用，门槛更低，让 HBase 帮助更多的用户解决他们遇到的实际问题。众所周知，SQL 是数据处理领域的语言标准，简单，好用，表达力强，用户使用广泛。HBase 很有必要回过头来支持 SQL，包括语言表达的支持和相关的数据处理方式和能力的支持。当然，HBase SQL 的实现和发展跟传统单机数据库有很多不同，便于区别，我们称之为 NewSQL。这也是社区 Phoenix 项目的初衷，如果说 HBase 是功能强大的存储引擎，那么支持 NewSQL 之后，就变成了新一代的大飞机。

接下来，第二章将介绍 Phoenix 项目是如何让 HBase 从 NoSQL 成长为 NewSQL 的；第三章介绍 Phoenix 在阿里的实践，增强和典型案例；第四章总结全文以及展望 Phoenix 未来的发展。

2.方案与实现

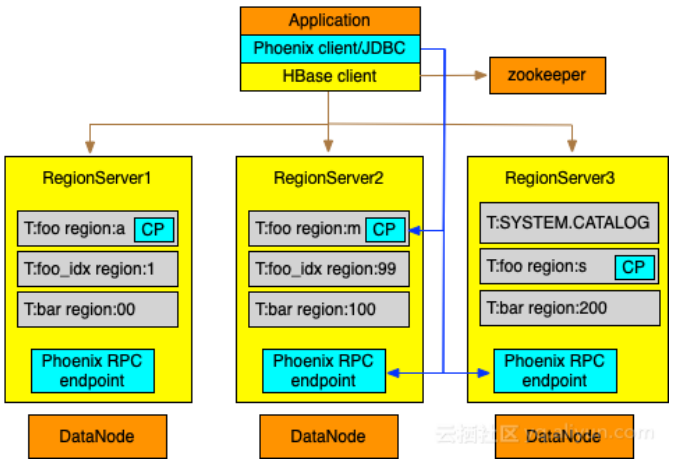


图 2 Phoenix 架构

系统整体架构如图 2 所示，其中黄色部分是 HBase 的组件，蓝色部分是 Phoenix，我们可以看出 Phoenix 跟 HBase 是紧密结合的。Phoenix 首先要能通过 SQL 暴露

HBase 的能力，然后再通过一系列功能和特性增强 HBase 能力，并且还要做到足够易用。下面我分别从基于 HBase 的 SQL，SQL 执行与优化以使用接口三个方面介绍 Phoenix 已经做到的事情。

2.1 基于 HBase 的 SQL

让 HBase 支持 SQL，我们首先能想到的是，把 SQL 语言翻译成 HBase API，并且 HBase 本身的功能特性必须得能通过 SQL 暴露出来。Phoenix SQL 语法即 SQL-92 标准语法+方言。Phoenix 支持标准语法的绝大部分特性，包括：标准类型；聚合，连接，in，排序以及子查询等查询语法；create，drop，delete 等数据操作语法，这些操作在底层都会转变为 HBase API。

此外，HBase 的某些特性也能通过 SQL 方言的形式表达，比如：

1. HBase 的列族，可以把相关的列放到一起，以减少 IO，优化读性能，在创建 Phoenix 表的时候直接写成”cf.col”即可，Phoenix 会自动创建 cf 列族，如果不指定，则放在默认列族中；
2. Phoenix 将 insert 和 update 合并成 upsert 关键字，来对应 HBase 的 Put 概念；
3. HBase 为了避免在表初始时只有一个 Region 带来数据热点，支持预分区，Phoenix 支持在建表的时候通过 split on 关键字来表达。
4. HBase 支持动态列的特性在 Phoenix 中也得以保留，存入数据的时候直接声明新的字段即可使用。

由于充分结合，Phoenix 天然继承了 HBase 所拥有的高并发，大容量，实时存取等特性。

```
SELECT * FROM foo  WHERE bar > 100
```

VS

```
HTable table = (HTable) conn.getTable(TableName.valueOf("foo"));
Scan scan = new Scan();
scan.setFilter(new ValueFilter(CompareOp.GREATER, new
CustimizedTypeComparator(“bar”, 30)));
ResultScanner results = table.getScanner(scan);
for (Result result : results) {
    //processing code
}
results.close();
table.close();
```


图 3 SQL vs HBase API

从图 3 中可以看出，有了 SQL，用户无需编写繁琐的 java 代码，大大简化了代码逻辑，传统数据库用户可以快速上手，原有基于传统数据库的代码逻辑，只需要少量修改即可运行起来。

2.2 SQL 查询优化

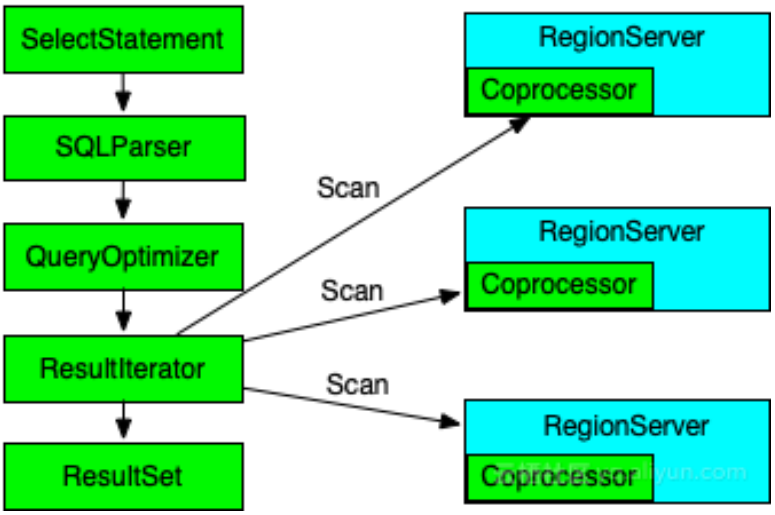


图 4 Phoenix SQL 执行流程

在很多场景下仅拥有实时存取的特性是不够的，大数据在线原地分析也是自然而然的需求，Phoenix 基于 HBase 拥有的高效缓存和 LSM 索引，可以做到对 PB 级数据做操作型分析的毫秒级或秒级返回。Phoenix SQL 的整体执行流程如图 4 所示，用户输入的查询语句首先经过 SQLParser 解析为执行计划，然后经过 QueryOptimizer 的优化，选取最优的执行计划，执行计划最终会转变为 HBase 的 Scan，RegionServer 端的协处理器会作用于该 Scan，做本地过滤和聚合，然后把结果返回到客户端做汇总聚合形成最终结果，并通过 ResultSet 的形式返回给用户。其中主要使用到的策略有：无需数据传输的算子原地计算；实时同步的二级索引；热点数据自动打散；以及基于代价和规则的执行计划优化等。后面笔者介绍前三个最具有 Phoenix 特色的策略。

在没有 Phoenix 之前，用户如果需要分析 HBase 数据，只能从 HBase 拖出去或者绕过 HBase 直接读取 HDFS 上面的 HFile 的方式进行，前者浪费了大量的网络 IO，后者用不到 HBase 本身做的大量缓存和索引优化。Phoenix 使用 HBase 的协处理器机制，直接在 RegionServer 上执行算子逻辑，然后将算子的结果返回即可，也就是大数据中的“Move Operator to Data”理念。比如，用户执行“select

count(*) from mytable where mytime > timestamp'2018-11-11 00:00:00'"，在实际执行中，会先找到符合过滤条件的 region，然后在 RegionServer 本地计算 count，最后再对各个 Region 上的结果进行汇总。

索引是传统数据库中常见的技术，HBase 表中可以指定主键，对主键使用 LSM 算法构建索引。Phoenix 中，用户在创建表的时候，可以指定索引列，可以是单列，也可以是组合列。很多时候仅有主键索引是不够的，特别是对于列特别多的宽表，为此，Phoenix 提供二级索引功能，用户能够对表中非主键的列添加索引，并可以加入其它相关列到索引表中，如果某个查询涉及到的列全部在索引表中，直接查询索引表即可，无需访问原数据表。索引表跟原表做到实时同步更新，以保证数据一致性。

索引示例如图 5 所示，创建索引的时候通过"on"关键字指定索引表的主键，实际 HBase 表中会使用 BA 来作为联合主键；"include"关键字指定加入到索引表的其他列。当执行"select c from data_table where b > xx "时，Phoenix 直接查询索引表即可，否者需要返回原表查找。

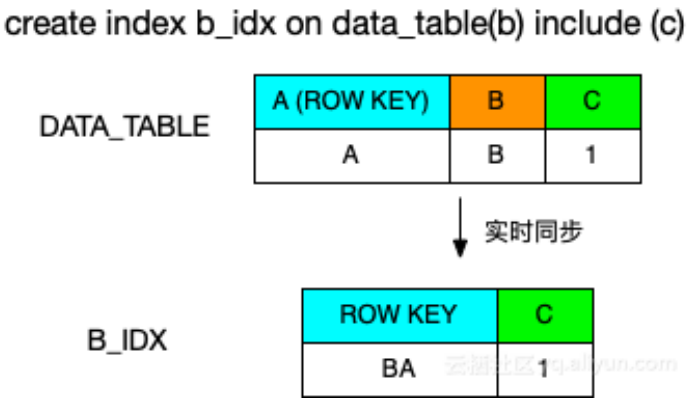


图 5 Phoenix 二级索引

当数据的访问比较集中，比如物联网场景中需要对最新的的监控数据做查询分析，而这部分数据往往会集中分布在某个 RegionServer 上，那么就会形成热点，性能也会大大折扣。此时需要能够把数据打散到不同的 RegionServer 上来解决，加盐就是这样一个技术。加盐的过程本质上是对原有的主键加上一个字节的前缀，如下面公式所示：

```
new_row_key = (++index % BUCKETS_NUMBER) + original_key
```

其中，BUCKETS_NUMBER 为桶的个数。主键的分布决定了数据的分布，把主键打散也就意味着数据打散。具体使用时，一般建议桶的数目等于 RegionServer 的数目。

2.3 使用接口

Phoenix 提供 JDBC 的方式访问，并支持重客户端和轻客户端两种方式，重客户端的 JDBC 串前缀是“jdbc:phoenix:”，轻客户端的 JDBC 串前缀为“jdbc:phoenix:thin:”。重客户端中，Phoenix 初执行在 HBase 协处理器之外的逻辑均运行在客户端，这样可以带来最优的性能，但也带来了客户端的复杂性。轻客户端则将上诉逻辑运行在单独不熟的 QueryServer 中，客户端仅有很薄的 JDBC 协议转换。轻客户端除支持 Java 语言外，也支持 Python、Go、C#等多种语言。

3.实践案例

阿里云 HBase 产品中集成 Phoenix 功能已经有一年多的时间了，阿里云 HBase 团队对 Phoenix 在稳定性和性能方面做了一系列的改进优化，并积极反馈回社区，团队的瑾谦同学也成长为了 Phoenix 社区的 commiter。Phoenix 的用户既有来自于新型大数据业务，如物联网、互联网金融以及新零售等；也有来自传统行业，比如游戏、养殖业和运输业等。Phoenix 数据既有来在线实时业务，也有从单机数据库同步进来。下面，笔者以物联网场景为例，说明 Phoenix 如何在实际业务中解决用户难题。

在物联网场景下，大量传感器会把实时监测到的数据上传到云平台，经过一定的预处理后实时写入到 Phoenix 中，管控平台从 Phoenix 中直接在线查询和分析数据，如生成报表，监控异常等。该场景会用到 Phoenix 的一下特性：

1. 对实时写入的并发和 TPS 要求很高，甚至可以达到百万级。
2. 数据量比较大，一般在 TB 到 PB 级别，且需要根据业务增长动态扩容。
3. 有动态列的需求，比如根据业务调整，动态添加监控指标。
4. 在线查询，甚至是毫秒级查询要求。
5. 维度比较多，需要用到二级索引，以加速在线查询。
6. 频繁查询最近一段时间的数据，会造成热点，可以使用加盐的特性。

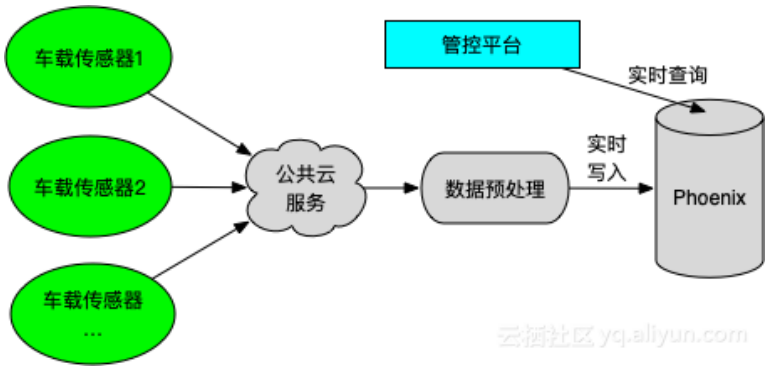


图 6 Phoenix 在物联网的应用

4.总结与展望

开源的 HDFS、MapReduce 和 HBase 来源于 Google 的三篇论文，但最终驱动的还是大数据场景下的实际需求。Phoenix 的出现也是需求驱动的必然结果，它继承了 HBase 的各种特性，如高并发，水平扩展，实时读写等，并在其基础上实现了一套 SQL 引擎，增加了如语法解析、二级索引、查询优化以及 JDBC 等功能和特性，完成了 NoSQL 到 NewSQL 的转变。

阿里云 HBase 团队和社区对 Phoenix 后续还会根据实际需求做进一步的进化，包括稳定性的进一步加强；执行计划和运行时的进一步优化；轻客户端在易用性支持上的持续完善；实时交易场景中事务的支持以及跟 Spark 的深度集成等等。笔者非常希望对 HBase 和 Phoenix 感兴趣的读者朋友可以参与到社区里面，共同推动技术的发展，满足更多的场景需求。

HBase 在新能源汽车监控系统中的应用

颜禹 重庆博尼施科技有限公司 CTO

重庆博尼施科技有限公司是一家商用车全周期方案服务商，利用车联网、云计算、移动互联网技术，在物流领域 为商用车的生产、销售、使用、售后、回收各个环节提供一站式解决方案，其中的新能源车辆监控系统就是由该公司提供的。该系统主要用于东风轻卡等新能源商用车监控服务，目前该系统正在阿里云线上稳定运行。

新能源车辆监控系统是一个车辆网服务平台，具有高并发、数据量大、实时性要求高等特点。对车辆监控系统来说最重要的问题就是如何处理车辆产生的海量数据，我们估计，当车辆数量增长到 10 万时，每天会产生大约 2TB 的数据，这些数据不仅需要存储，还需要做到实时可查。本文将介绍项目的背景和系统的基本架构，随后介绍我们在开发过程中遇到的各种问题以及解决方案。

1.项目背景

本项目为车联网监控系统，系统由车载硬件设备、云服务端构成。车载硬件设备会定时采集车辆的各种状态信息，并通过移动网络上传到服务器端。服务器端接收到硬件设备发送的数据首先需要将数据进行解析，校验，随后会将该消息转发到国家汽车监测平台和地方汽车监测平台，最后将解析后的明文数据和原始报文数据存储到系统中。车辆的数据和其他数据需要通过 web 页面或 rest API 接口进行查询访问。要求半年内的数据查询响应时间在毫秒级别内，超过半年的数据需要放到更加低成本的介质上，查询延迟在 3s 以内，这些数据的查询频次比较低。系统的主要参数有以下几项:

1. 10 万台车辆同时在线；
2. 车辆正常情况下平均每分钟发送两个数据报文到监控平台；
3. 若车辆处于报警状态，则平均一秒钟发送一次数据报文；
4. 数据情况：（1）、车辆数据报文平均大小为 1KB；（2）、解析后的数据包大小为 7KB；（3）、平均一台车每天会产生 20MB 的数据；（4）、系统每

天会产生 2TB 的数据；（5）同时系统有 2.9 亿行的数据需要写入到数据库中；

5. 系统并发量：（1）、3300 的持续并发量；（2）、10 万个长连接；（3）、每秒 3.3MB 的原始数据需要被解析；（4）、每秒 23.1MB 的解析数据需要被存储。

2.系统设计

系统的技术选型对初创公司来说至关重要，所以在设计系统的时候尤为小心。经过仔细分析，我们要求新系统必须满足以下几个条件：

1. 必须能够支持海量数据的不间断写入，而且能够存储 PB 级别的数据，具有高扩展性、高可靠性等；
2. 能够支持简单的关键字查询，响应时间在秒级别内；
3. 能够兼容大数据生态产品(如 Spark、Hive、Hadoop 等)，同时支持离线和准实时 OLAP；
4. 优先选择有雄厚实力的商业公司支持的云平台，最大限度减少运维成本。

2.1 为何选择 HBase

因为车辆的监控数据非常大，传统关系型数据库（如 Mysql、pg 等）已经无法胜任存储工作，所以我们需要选用一种分布式数据库用于存储车辆实时数据。
我们在市场上能够找到分布式数据库有 MongoDB 和 HBase。

1. MongoDB：MongoDB 是一种我们曾经使用过的数据库，该数据库是一种基于文档的数据库。支持分片副本集扩展，但是由于 MongoDB 的分片集群维护成本过高，另外查询性能严重依赖索引，扩容时分片的迁移速度过慢。所以在这一版本的监控系统中我们并未采用。
2. HBase：HBase 底层存储基于 HDFS 的面向列数据库，其核心思想来源于谷歌三大论文内的 bigtable。在谷歌和开源界均拥有丰富的应用实践经验。除此之外，HBase 还有以下特点：（1）、支持 PB 级别的数据量；（2）、压缩效率非常高；（3）、支持亿级别的 QPS；（4）、在国内外很多大型互联网公司使用；（5）、HBase 添加节点及扩容比较方便，无需 DBA 任何干预。

经过对这几种数据库的分析，我们最终选用 HBase，其满足我们前面提到的四个要求，而且还提供 Phoenix 插件用于 SQL 语句的查询。

作为初创公司，我们的运维能力有限，我们需要业务的快速落地。所以自建机房以及运维团队意味着前期较大的投入以及高昂的运维成本，所以我们决定使用云方案。

经过比较国内的各大云厂商，我们最终选用了阿里云平台，因为阿里云提供 SaaS 化的 HBase 服务，同时阿里云 HBase 支持很全面的多模式，支持冷数据存放在 OSS 之中，节约成本；支持备份恢复等特性，做到了真正的 native cloud 的数据库服务。另外，HBase 在阿里内部部署超过 12000 台机器，历经 7 年天猫双 11 的考验，这些实际数据以及经验增强了我们对阿里云 HBase 的技术信心，同时满足了我们的技术和业务需求。

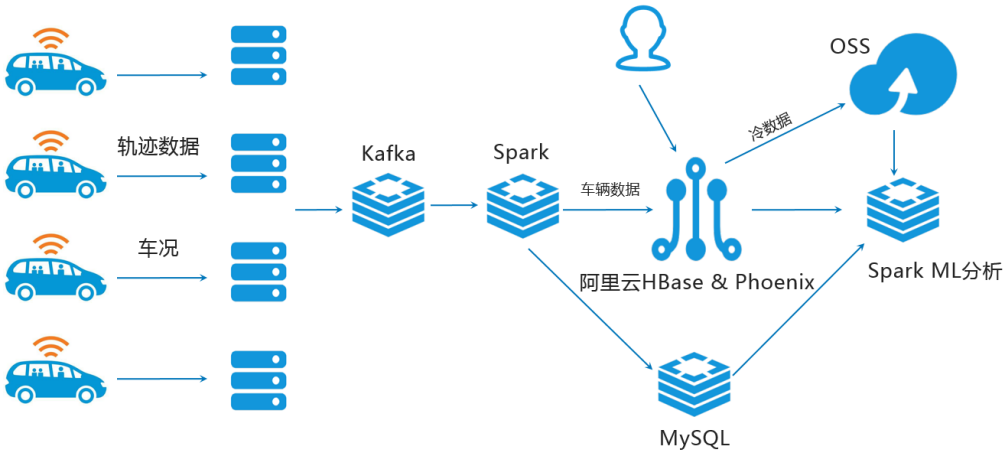
2.2 层级系统架构

系统采用层级架构以方便后期扩展和维护，现在主要分为以下几层：

1. **接入层：**主要负责管理车辆设备的长连接，认证接收车辆发送的数据，下放数据和指令等操作。
2. **消息队列层：**主要负责缓存接入层收到的车辆实时数据。
3. **实时数据处理与解析层：**主要负责解析车辆上传的实时数据，并将其存储到数据库中。另外还需要负责数据转发、指令生成等工作。
4. **应用层：**主要负责处理各种业务逻辑、将解析后的数据进行分析整理提供给用户使用。

2.3 系统架构预览

最终新能源监控系统的系统架构设计如下



图中最左端为监控的车辆，它会实时采集车辆的各项数据，并把采集到的数据通过移动互联网发送到平台。平台验证完数据会将其写入到 Kafka 消息队列中。流式计算服务器从 Kafka 消息队列中取出车辆的原始数据，并对车辆的数据进行解析、存储、转发等操作。HBase 集群负责存储车辆实时数据，MySQL 负责存储组织关系数据。同时，我们还会将超过一定时间（比如半年前）的数据转存到 OSS 存储介质中，以便降低存储成本。Spark ML 会对系统中的各项数据进行分析。终端用户会从 HBase 中查询一些数据。

3.HBase 使用难点

3.1 Row key 设计

团队在使用 HBase 之前一直使用 MySQL 关系型数据库，在系统设计之初并没有考虑 HBase 的特性，而是按照关系型数据库的设计原则设计。经过一段时间的了解后才知道 HBase 主要使用 Row key 进行数据查询。Row key 的设计至关重要。目前系统中设计的 Row key 如下

1. **设备 ID + 时间戳：**此种方式可以快速定位单台车辆。但是由于设备 ID 前缀为型号，一旦某批次的设备或车辆发送故障，则会造成 HBase 的某个 Region 写入压力过大。
2. **subString(MD5(设备 ID),x) + 设备 ID + 时间戳：**此种方式可以将所有车辆打散到每个 Region 中，避免热点数据和数据倾斜问题，式中的 x 一般取 5 左右。但对于某个型号的车辆查询就只能够每台车辆单独进行查询了。

3.2 复杂查询问题

虽然通过 Row key 的设计可以解决部分数据查询的需求，但是在面对复杂需求时难以通过 Row key 直接索引到数据，若索引无法命中，则只能进行大范围或全表扫描才能够定位数据。所以我们在使用 HBase 时尽量避免复杂的查询需求。但业务方面仍然会有部分较为复杂的查询需求。针对这些需求，我们主要使用两种方式来建立二级索引。

1. **手动在事务产生时将索引写入到 HBase 表中：**使用这种方式建立索引虽然可以不借用第三方插件，但是事务的原子性很难得到保障，业务代码也会因为索引的变化而难以维护。另外索引的管理也较为麻烦，后期的数据迁移很难能够。
2. **通过 Phoenix 构建索引：**通过 Phoenix 构建索引可以避免事务原子性问题，另外也可以通过重建索引来进行数据迁移。因为使用的 SQL 语句，开发人员更能够利用之前关系型数据库的设计经验建立数据索引。

目前新能源监控系统中主要使用 Phoenix 实现二级索引，大大增加了数据的查询使用场景。虽然 Phoenix 能够通过二级索引实现较为复杂的数据查询，但对于更为复杂的查询与分析需求就显得捉襟见肘。所以我们选用了 Spark 等其他数据分析组件对数据进行离线分析，分析后对结果通过接口提供给用户。

3.3 多语言连接问题

团队使用 Python 语言构建系统，但 HBase 使用 Java 语言编写，原生提供了 Java API，并未对 Python 语言提供直接的 API。我们使用 happybase 连接 HBase，因为它提供了更为 Pythonic 的接口服务。另外我们也是用 QueryServer 作为 Python 组件和 Phoenix 连接的纽带。

4.HBase 冷数据存储

系统中车辆数据分为热数据和冷数据，热数据需要 HBase 中实时可查，冷数据虽不需要实时可查，但却需要一直保存在磁盘中。阿里云 HBase 支持将冷数据直接存储在 OSS 中，而这些数据的转存只需要简单的设置表相关属性，操作非常简单。将冷数据存储在 OSS 之中大大减少了数据的存储成本。

5.总结

首先，本文介绍了新能源车辆监控系统的项目背景，随后分析了本项目的项目难点，并介绍了我们团队的各种解决方案。针对项目需求，介绍了我们选择 HBase 的原因，及在 HBase 数据库使用过程中的经验和痛点。

6.展望

未来，我们会在系统接入大量车辆后，使用 golang 重写高性能组件以满足后期的并发性能需求。由于项目初期考虑到开发时间的问题，并未采用服务拆分的方式进行开发，这限制了系统的可扩展性，后期我们会根据实际业务需求，将系统切分成相对独立的模块，增强扩展性可维护性。

另外，车辆数据积累到一定程度后，我们可以利用这些数据进行大数据分析，如车辆的故障诊断，车辆状态预测等，这样就可以在车辆出现问题前提前发出预警，为车主和保险公司避免更大的损失，降低运营成本。

HBase 在滴滴出行的应用场景和最佳实践

李扬 滴滴出行 资深软件开发工程师

1.背景

1.1 对接业务类型

HBase 是建立在 Hadoop 生态之上的 Database，源生对离线任务支持友好，又因为LSM树是一个优秀的高吞吐数据库结构，所以同时也对接了很多线上业务。在线业务对访问延迟敏感，并且访问趋向于随机，如订单、客服轨迹查询。离线业务通常是数仓的定时大批量处理任务，对一段时间内的数据进行处理并产出结果，对任务完成的时间要求不是非常敏感，并且处理逻辑复杂，如天级别报表、安全和用户行为分析、模型训练等。

1.2 多语言支持

HBase 提供了多语言解决方案，并且由于滴滴各业务线 RD 所使用的开发语言各有偏好，所以多语言支持对于 HBase 在滴滴内部的发展是至关重要的一部分。我们对用户提供了多种语言的访问方式：HBase Java native API、Thrift Server（主要应用于 C++、PHP、Python）、JAVA JDBC（Phoenix JDBC）、Phoenix QueryServer（Phoenix 对外提供的多语言解决方案）、MapReduce Job（Htable/Hfile Input）、Spark Job、Streaming 等。

1.3 数据类型

HBase 在滴滴主要存放了以下四种数据类型：

1. 统计结果、报表类数据：主要是运营、运力情况、收入等结果，通常需要配合 Phoenix 进行 SQL 查询。数据量较小，对查询的灵活性要求高，延迟要求一般。

2. 原始事实类数据：如订单、司机乘客的 GPS 轨迹、日志等，主要用作在线和离线的数据供给。数据量大，对一致性和可用性要求高，延迟敏感，实时写入，单点或批量查询。
3. 中间结果数据：指模型训练所需要的数据等。数据量大，可用性和一致性要求一般，对批量查询时的吞吐量要求高。
4. 线上系统的备份数据：用户把原始数据存在了其他关系数据库或文件服务，把 HBase 作为一个异地容灾的方案。

2. 使用场景介绍

2.1 场景一：订单事件

这份数据使用过滴滴产品的用户应该都接触过，就是 App 上的历史订单。近期订单的查询会落在 Redis，超过一定时间范围，或者当 Redis 不可用时，查询会落在 HBase 上。业务方的需求如下：

1. 在线查询订单生命周期的各个状态，包括 status、event_type、order_detail 等信息。主要的查询来自于客服系统。
2. 在线历史订单详情查询。上层会有 Redis 来存储近期的订单，当 Redis 不可用或者查询范围超出 Redis，查询会直接落到 HBase。
3. 离线对订单的状态进行分析。
4. 写入满足每秒 10K 的事件，读取满足每秒 1K 的事件，数据要求在 5s 内可用。

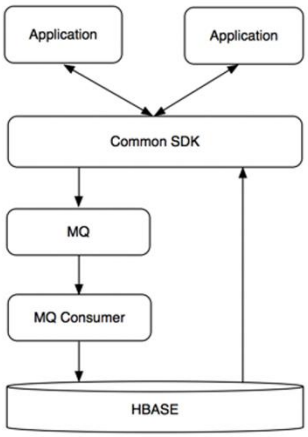


图 1 订单流数据流程

按照这些要求，我们对 Rowkey 做出了下面的设计，都是很典型的 scan 场景。

订单状态表

Rowkey：reverse(order_id) + (MAX_LONG - TS)

Columns：该订单各种状态

订单历史表

Rowkey：reverse(passenger_id | driver_id) + (MAX_LONG - TS)

Columns：用户在时间范围内的订单及其他信息

2.2 场景二：司机乘客轨迹

这也是一份滴滴用户关系密切的数据，线上用户、滴滴的各个业务线和分析人员都会使用。举几个使用场景上的例子：用户查看历史订单时，地图上显示所经过的路线；发生司乘纠纷，客服调用订单轨迹复现场景；地图部门用户分析道路拥堵情况。

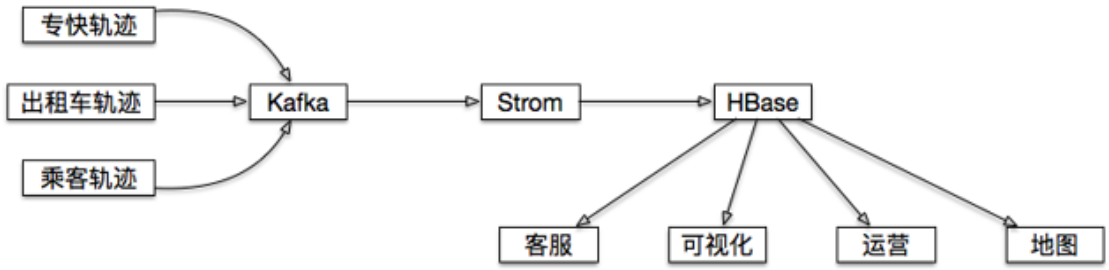


图 2 司乘轨迹数据流程

用户们提出的需求：

1. 满足 App 用户或者后端分析人员的实时或准实时轨迹坐标查询；
2. 满足离线大规模的轨迹分析；
3. 满足给出一个指定的地理范围，取出范围内所有用户的轨迹或范围内出现过的用户。

其中，关于第三个需求，地理位置查询，我们知道 MongoDB 对于这种地理索引有源生的支持，但是在滴滴这种量级的情况下可能会发生存储瓶颈，HBase 存储和扩展性上没有压力但是没有内置类似 MongoDB 地理位置索引的功能，没有就

需要我们自己实现。通过调研，了解到关于地理索引有一套比较通用的GeohHash算法 。

GeoHash 是将二维的经纬度转换成字符串，每一个字符串代表了某一矩形区域。也就是说，这个矩形区域内所有的点（经纬度坐标）都共享相同的 GeoHash 字符串，比如说我在悠唐酒店，我的一个朋友在旁边的悠唐购物广场，我们的经纬度点会得到相同的 GeoHash 串。这样既可以保护隐私（只表示大概区域位置而不是具体的点），又比较容易做缓存。

图 3 GeoHash 示意图

但是我们要查询的范围和 GeohHash 块可能不会完全重合。以圆形为例，查询时会出现如图 4 所示的一半在 GeoHash 块内，一半在外面的情况（如 A、B、C、D、E、F、G 等点）。这种情况就需要对 GeoHash 块内每个真实的 GPS 点进行第二次的过滤，通过原始的 GPS 点和圆心之间的距离，过滤掉不符合查询条件的数据。

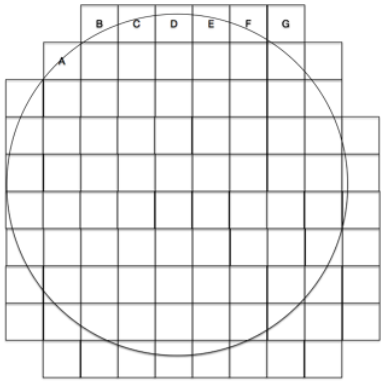


图 4 范围查询时，边界 GeoHash 块示意图

最后依据这个原理，把 GeoHash 和其他一些需要被索引的维度拼装成 Rowkey，真实的 GPS 点为 Value，在这个基础上封装成客户端，并且在客户端内部对查询逻辑和查询策略做出速度上的大幅优化，这样就把 HBase 变成了一个 MongoDB 一样支持地理位置索引的数据库。如果查询范围非常大（比如进行省级别的分析），还额外提供了 MR 的获取数据的入口。

两种查询场景的 Rowkey 设计如下：

1. 单个用户按订单或时间段查询： reverse(user_id) + (Integer.MAX_LONG-TS/1000)
2. 给定范围内的轨迹查询：reverse(geohash) + ts/1000 + user_id

2.3 场景三：ETA

ETA 是指每次选好起始和目的地后，提示出的预估时间和价格。提示的预估到达时间和价格，最初版本是离线方式运行，后来改版通过 HBase 实现实时效果，把 HBase 当成一个 KeyValue 缓存，带来了减少训练时间、可多城市并行、减少人工干预的好处。

整个 ETA 的过程如下：

1. 模型训练通过 Spark Job，每 30 分钟对各个城市训练一次；
2. 模型训练第一阶段，在 5 分钟内，按照设定条件从 HBase 读取所有城市数据；
3. 模型训练第二阶段在 25 分钟内完成 ETA 的计算；
4. HBase 中的数据每隔一段时间会持久化至 HDFS 中，供新模型测试和新的特征提取。

Rowkey：salting+cited+type0+type1+type2+TS

Column：order, feature

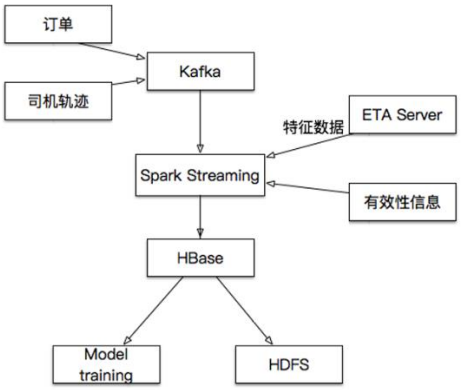


图 5 ETA 数据流程

2.4 场景四：监控工具 DCM

用于监控 Hadoop 集群的资源使用（Namenode，Yarn container 使用等），关系数据库在时间维度过程以后会产生各种性能问题，同时我们又希望可以通过 SQL 做一些分析查询，所以使用 Phoenix，使用采集程序定时录入数据，生产成报表，存入 HBase，可以在秒级别返回查询结果，最后在前端做展示。

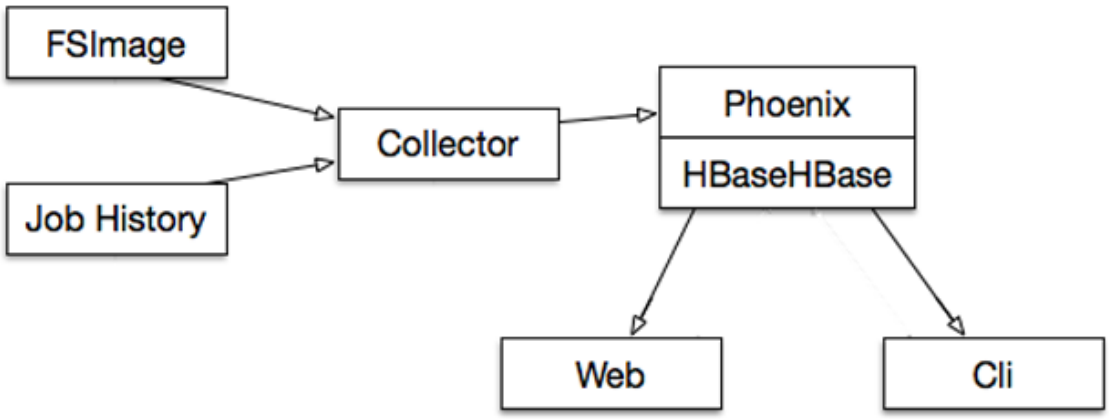


图 6 DCM 数据流程

图 7、图 8、图 9 是几张监控工具的用户 UI，数字相关的部分做了模糊处理。

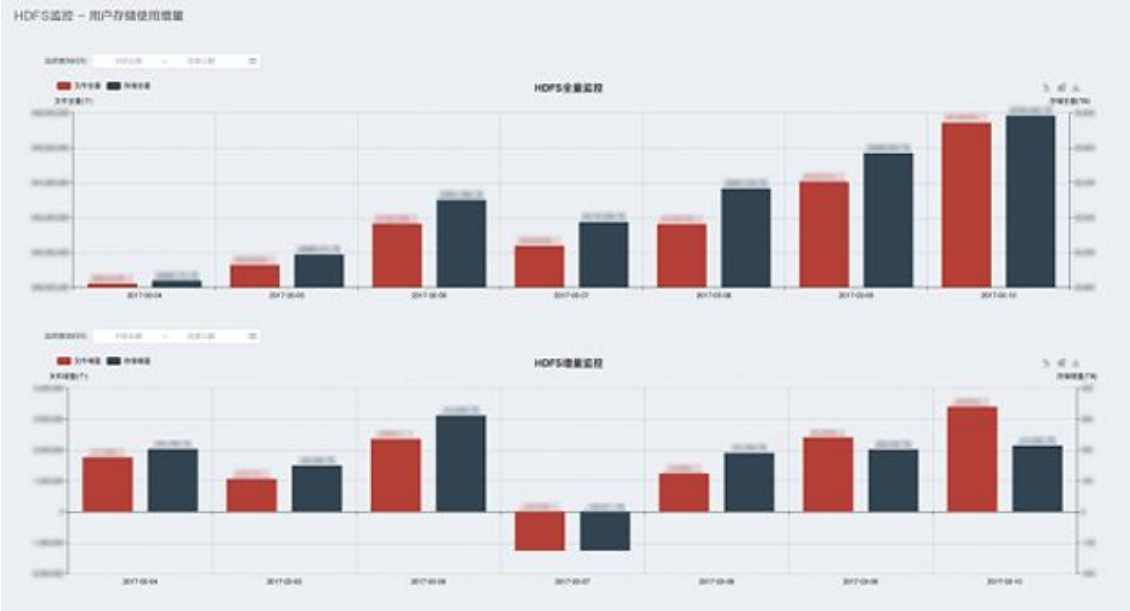


图 7 DCM HDFS 按时间统计使用全量和增量

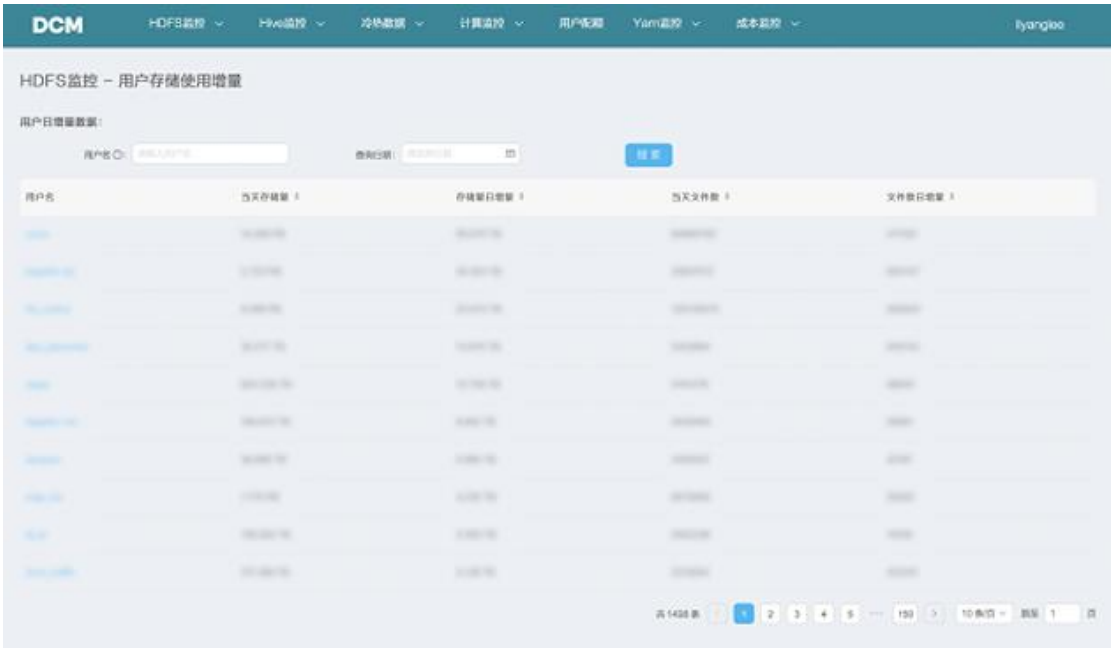


图 8 DCM HDFS 按用户统计文件数

图 9 DCM，MR Job 运行结果统计

3. 滴滴在 HBase 对多租户的管理

我们认为单集群多租户是最高效和节省精力的方案，但是由于 HBase 对多租户基本没有管理，使用上会遇到很多问题：在用户方面比如对资源使用情况不做分析、存储总量发生变化后不做调整和通知、项目上线下线没有计划、想要最多的资源和权限等；我们平台管理者也会遇到比如线上沟通难以理解用户的业务、对每个接入 HBase 的项目状态不清楚、不能判断出用户的需求是否合理、多租户在集群上发生资源竞争、问题定位和排查时间长等。

针对这些问题，我们开发了 DHS 系统（Didi HBase Service）进行项目管理，并且在 HBase 上通过 Namespace、RS Group 等技术来分割用户的资源、数据和权限。通过计算开销并计费的方法来管控资源分配。

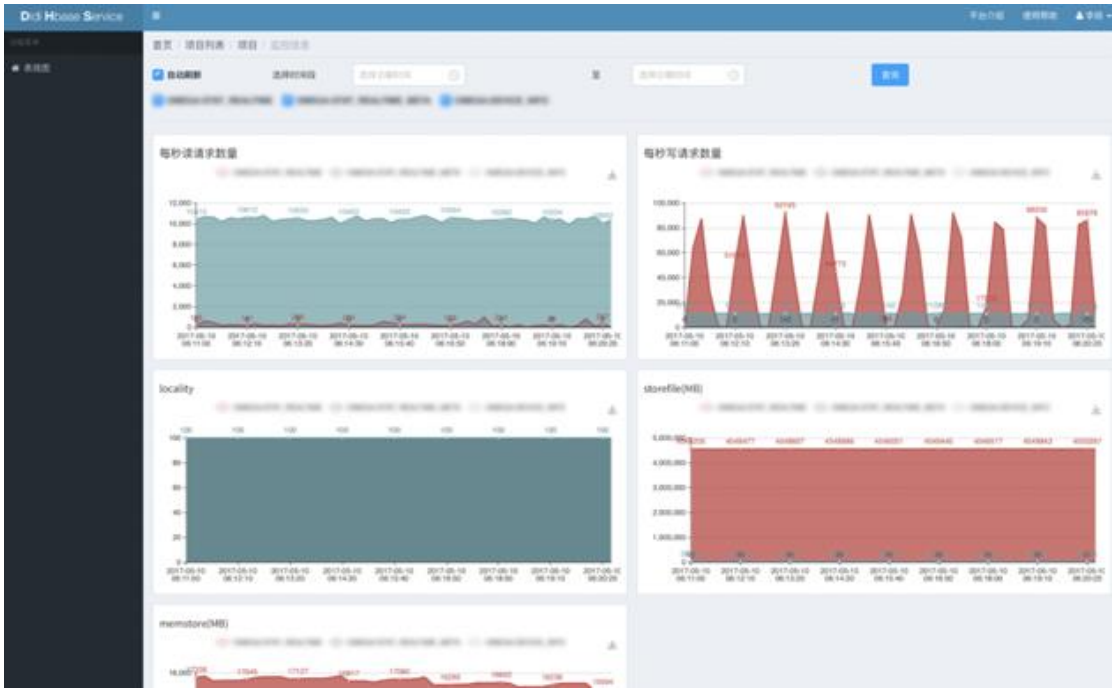


图 10 DHS 项目表监控

DHS 主要有下面几个模块和功能：

1. 项目生命周期管理：包括立项、资源预估和申请、项目需求调整、需求讨论；
2. 用户管理：权限管理，项目审批；
3. 集群资源管理；
4. 表级别的使用情况监控：主要是读写监控、memstore、blockcache、locality。

当用户有使用 HBase 存储的需求，我们会让用户在 DHS 上注册项目。介绍业务的场景和产品相关的细节，以及是否有高 SLA 要求。

之后是新建表以及对表性能需求预估，我们要求用户对自己要使用的资源有一个准确的预估。如果用户难以估计，我们会以线上或者线下讨论的方式与用户讨论帮助确定这些信息。然后会生成项目概览页面，方便管理员和用户进行项目进展的跟踪。

HBase 自带的 jxm 信息会汇总到 Region 和 RegionServer 级别的数据，管理员会经常用到，但是用户却很少关注这个级别。根据这种情况我们开发了 HBase 表级别的监控，并且会有权限控制，让业务 RD 只能看到和自己相关的表，清楚自己项目表的吞吐及存储占用情况。

通过 DHS 让用户明确自己使用资源情况的基础之上，我们使用了 RS Group 技术，把一个集群分成多个逻辑子集群，可以让用户选择独占或者共享资源。共享和独占各有自己的优缺点，如表 1。

	好处	坏处
多租户共享	资源利用率高，维护简单	用户竞争资源，发生问题定位时间长
多租户独占	资源冲突减少，可用性高，可细粒度调优和维护	业务低峰时段资源浪费，使用成本高

表 1 多租户共享和独占资源的优缺点

根据以上的情况，我们在资源分配上会根据业务的特性来选择不同方案：

1. 对于访问延迟要求低、访问量小、可用性要求低、备份或者测试阶段的数据：使用共享资源池；
2. 对于延迟敏感、吞吐要求高、高峰时段访问量大、可用性要求高、在线业务：让其独占一定机器数量构成的 RegionServer Group 资源，并且按用户预估的资源量，额外给出 20%~30%的余量。

最后我们会根据用户对资源的使用，定期计算开销并向用户发出账单。

4. RS Group

RegionServer Group，实现细节可以参照 HBase HBASE-6721 这个 Patch。滴滴在这个基础上作了一些分配策略上的优化，以便适合滴滴业务场景的修改。RS Group 简单概括是指通过分配一批指定的 RegionServer 列表，成为一个 RS Group，每个 Group 可以按需挂载不同的表，并且当 Group 内的表发生异常后，Region 不会迁移到其他的 Group。这样，每个 Group 就相当于一个逻辑上的子集群，通过这种方式达到资源隔离的效果，降低管理成本，不必为每个高 SLA 的业务线单独搭集群。

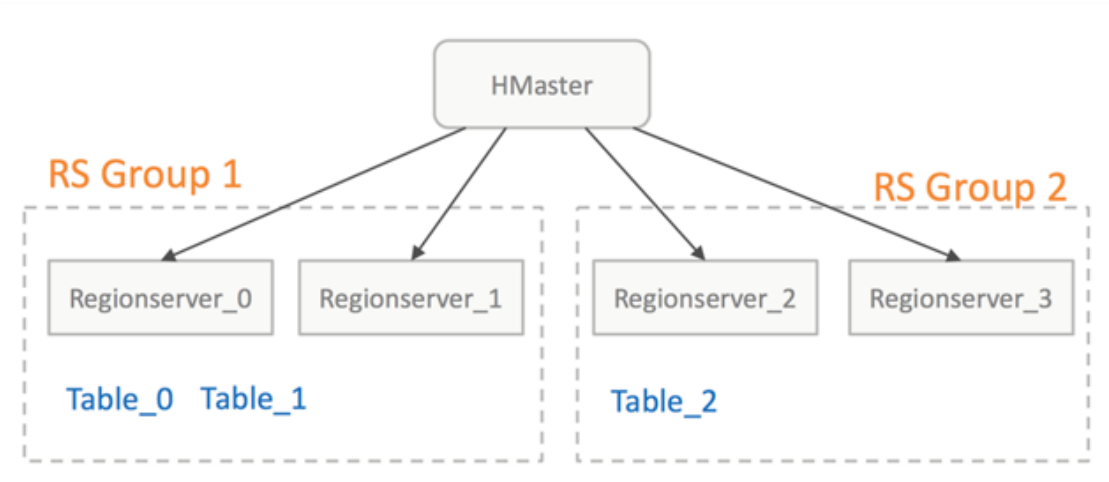


图 11 RS Group 示意图

5. 总结

在滴滴推广和实践 HBase 的工作中，我们认为至关重要的两点是帮助用户做出良好的表结构设计和资源的控制。有了这两个前提之后，后续出现问题的概率会大大降低。良好的表结构设计需要用户对 HBase 的实现有一个清晰的认识，大多数业务用户把更多精力放在了业务逻辑上，对架构实现知之甚少，这就需要平台管理者去不断帮助和引导，有了好的开端和成功案例后，通过这些用户再去向其他的业务方推广。资源隔离控制则帮助我们有效减少集群的数量，降低运维成本，让平台管理者从多集群无止尽的管理工作中解放出来，将更多精力投入到组件社区跟进和平台管理系统的研发工作中，使业务和平台都进入一个良性循环，提升用户的使用体验，更好地支持公司业务的发展。

HBase 在人工智能场景的使用

近几年来，人工智能逐渐火热起来，特别是和大数据一起结合使用。人工智能的主要场景又包括图像能力、语音能力、自然语言处理能力和用户画像能力等等。这些场景我们都需要处理海量的数据，处理完的数据一般都需要存储起来，这些数据的特点主要有如下几点：

1. 大：数据量越大，对我们后面建模越会有好处；
2. 稀疏：每行数据可能拥有不同的属性，比如用户画像数据，每个人拥有属性相差很大，可能用户 A 拥有这个属性，但是用户 B 没有这个属性；那么我们希望存储的系统能够处理这种情况，没有的属性在底层不占用空间，这样可以节约大量的空间使用；
3. 列动态变化：每行数据拥有的列数是不一样的。

为了更好的介绍 HBase 在人工智能场景下的使用，下面以某人工智能行业的客户案例进行分析如何利用 HBase 设计出一个快速查找人脸特征的系统。

目前该公司的业务场景里面有很多人脸相关的特征数据，总共 3400 多万张，每张人脸数据大概 3.2k。这些人脸数据又被分成很多组，每个人脸特征属于某个组。目前总共有近 62W 个人脸组，每个组的人脸张数范围为 1 ~ 1W 不等，每个组里面会包含同一个人不同形式的人脸数据。组和人脸的分布如下：

1. 43%左右的组含有 1 张人脸数据；
2. 47%左右的组含有 2 ~ 9 张人脸数据；
3. 其余的组人脸数范围为 10 ~ 10000。

现在的业务需求主要有以下两类：

1. 根据人脸组 id 查找该组下面的所有人脸；
2. 根据人脸组 id +人脸 id 查找某个人脸的具体数据。

1. MySQL + OSS 方案

为 MySQL 以及 OSS(对象存储)。相关表主要有人脸组表 group 和人脸表 face。表的格式如下：

group 表：

group_id	size
1	2

face 表：

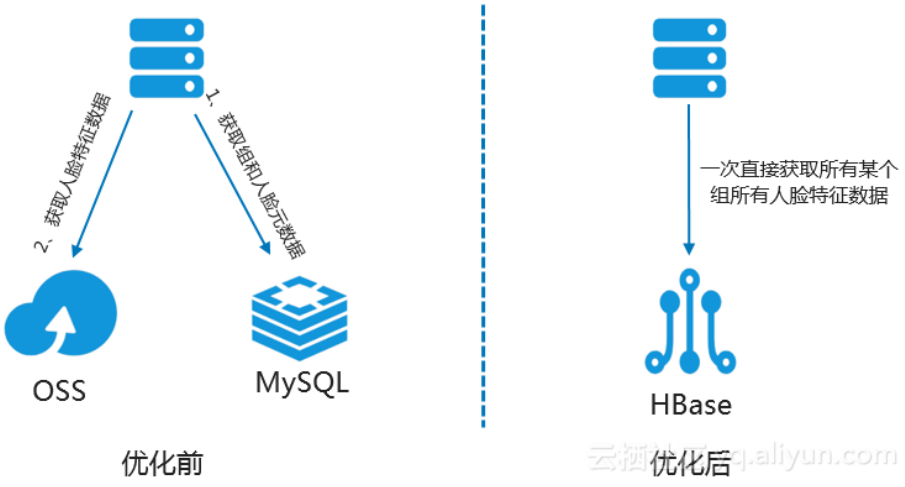
face_id	group_id	feature
"c5085f1ef4b3496d8b4da050cab0efd2"	1	"cwl4S/HO/nm6H……"

其中 feature 大小为 3.2k，是二进制数据 base64 后存入的，这个就是真实的人脸特征数据。

现在人脸组 id 和人脸 id 对应关系存储在 MySQL 中，对应上面的 group 表；人脸 id 和人脸相关的特征数据存储在 OSS 里面，对应上面的 face 表。

因为每个人脸组包含的人类特征数相差很大（1 ~ 1W），所以基于上面的表设计，我们需要将人脸组以及每张人脸特征 id 存储在每一行，那么属于同一个人脸组的数据在 MySQL 里面上实际上存储了很多行。比如某个人脸组 id 对应的人脸特征数为 1W，那么需要在 MySQL 里面存储 1W 行。

我们如果需要根据人脸组 id 查找该组下面的所有人脸，那么需要从 MySQL 中读取很多行的数据，从中获取到人脸组和人脸对应的关系，然后到 OSS 里面根据人脸 id 获取所有人脸相关的特征数据，如下图的左部分所示。



我们从上图的查询路径可以看出，这样的查询导致链路非常长。从上面的设计可看出，如果查询的组包含的人脸张数比较多的情况下，那么我们需要从 MySQL

里面扫描很多行，然后再从 OSS 里面拿到这些人脸的特征数据，整个查询时间在 10s 左右，远远不能满足现有业务快速发展的需求。

2. HBase 方案

上面的设计方案有两个问题：

1. 原本属于同一条数据的内容由于数据本身大小的原因无法存储到一行里面，导致后续查下需要访问两个存储系统；
2. 由于 MySQL 不支持动态列的特性，所以属于同一个人脸组的数据被拆成多行存储。

针对上面两个问题，我们进行了分析，得出这个是 HBase 的典型场景，原因如下：

1. HBase 拥有动态列的特性，支持万亿行，百万列；
2. HBase 支持多版本，所有的修改都会记录在 HBase 中；
3. HBase 2.0 引入了 MOB（Medium-Sized Object） 特性，支持小文件存储。HBase 的 MOB 特性针对文件大小在 1k~10MB 范围的，比如图片，短视频，文档等，具有低延迟，读写强一致，检索能力强，水平易扩展等关键能力。

我们可以使用这三个功能重新设计上面 MySQL + OSS 方案。结合上面应用场景的两大查询需求，我们可以将人脸组 id 作为 HBase 的 Rowkey，系统的设计如上图的右部分显示，在创建表的时候打开 MOB 功能，如下：

```
create 'face', {NAME => 'c', IS_MOB => true, MOB_THRESHOLD => 2048}
```

上面我们创建了名为 face 的表，IS_MOB 属性说明列族 c 将启用 MOB 特性，MOB_THRESHOLD 是 MOB 文件大小的阈值，单位是字节，这里的设置说明文件大于 2k 的列都当做小文件存储。大家可能注意到上面原始方案中采用了 OSS 对象存储，那我们为什么不直接使用 OSS 存储人脸特征数据呢，如果有这个疑问，可以看看下面表的性能测试：

对比属性	对象存储	云 HBase
建模能力	KV	KV、表格、稀疏表、SQL、 全文索引、时空、时序、图查询
查询能力	前缀查找	前缀查找、过滤器、索引
性能	优	优，特别对小对象有更低的延迟；在复杂 查询场景下，比对象存储有10倍以上的性能提升
成本	按流量，请求次数计费， 适合访问频率低的场景	托管式，在高并发，高吞吐场景有更低的成本
扩展性	优	优
适用对象范围	通用	<10MB

根据上面的对比，使用 HBase MOB 特性来存储小于 10MB 的对象相比直接使用对象存储有一些优势。我们现在来看看具体的表设计，如下图：

上面 HBase 表的列族名为 c，我们使用人脸 id 作为列名。我们只使用了 HBase 的一张表就替换了之前方面的三张表！虽然我们启用了 MOB，但是具体插入的方法和正常使用一样，代码片段如下：

```
String CF_DEFAULT = "c";
Put put = new Put(groupId.getBytes());
put.addColumn(CF_DEFAULT.getBytes(),faceId1.getBytes(), feature1.getBytes());
put.addColumn(CF_DEFAULT.getBytes(),faceId2.getBytes(), feature2.getBytes());
……
put.addColumn(CF_DEFAULT.getBytes(),faceIdn.getBytes(), featuren.getBytes());
table.put(put);
```

用户如果需要根据人脸组 id 获取所有人脸的数据，可以使用下面方法：

```
Get get = new Get(groupId.getBytes());
Result re=table.get(get);
```

这样我们可以拿到某个人脸组 id 对应的所有人脸数据。如果需要根据人脸组 id+人脸 id 查找某个人脸的具体数据，看可以使用下面方法：

```
Get get = new Get(groupId.getBytes());
get.addColumn(CF_DEFAULT.getBytes(), faceId1.getBytes())
Result re=table.get(get);
```

经过上面的改造，在 2 台 HBase worker 节点内存为 32GB，核数为 8，每个节点挂载四块大小为 250GB 的 SSD 磁盘，并写入 100W 行，每行有 1W 列，读取一行的时间在 100ms-500ms 左右。在每行有 1000 个 face 的情况下，读取一行的时间基本在 20-50ms 左右，相比之前的 10s 提升 200~500 倍。

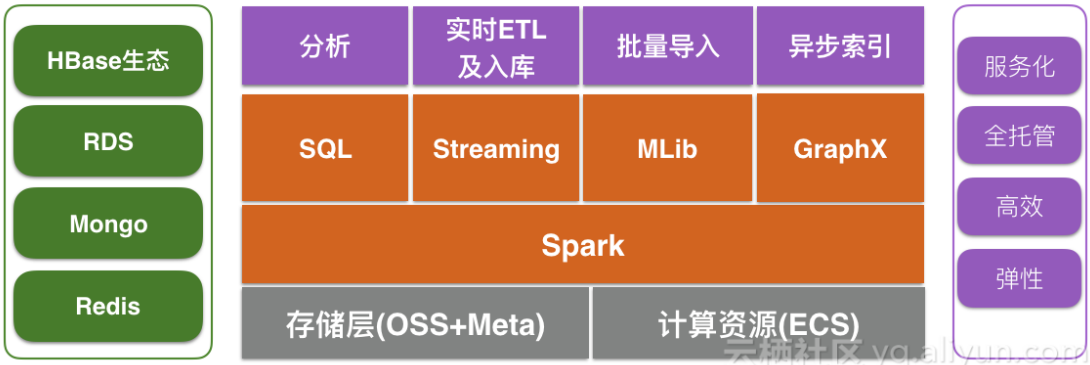
下面是各个方案的对比性能对比情况。

对比属性	对象存储	MySQL+对象存储	HBase MOB
读写强一致	Y	N	Y
查询能力	弱	强	强
查询响应时间	高	高	低
运维成本	低	高	低
水平扩展	Y	Y	Y

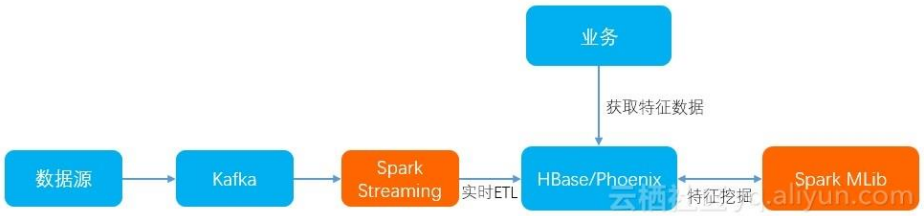
3. 使用 Spark 加速数据分析

我们已经将人脸特征数据存储在 HBase 之中，这个只是数据应用的第一步，如何将隐藏在这些数据背后的价值发挥出来？这就得借助于数据分析，在这个场景就需要采用机器学习的方法进行聚类之类的操作。我们可以借助 Spark 对存储于 HBase 之中的数据进行分析，而且 Spark 本身支持机器学习的。但是如果直接采用开源的 Spark 读取 HBase 中的数据，会对 HBase 本身的读写有影响的。

针对这些问题，我们可以对 Spark 进行了相关优化，比如直接读取 HFile、算子下沉等；通过 SQL 服务 ThriftServer、作业服务 LivyServer 简化 Spark 的使用等。目前这套 Spark 的技术栈如下图所示。



通过 Spark 服务，我们可以和 HBase 进行很好的整合，将实时流和人脸特征挖掘整合起来，整个架构图如下：



我们可以收集各种人脸数据源的实时数据，经过 Spark Streaming 进行简单的 ETL 操作；其次，我们通过 Spark MLib 类库对刚刚试试收集到的数据进行人脸特征挖掘，最后挖掘出来的结果存储到 HBase 之中。最后，用户可以通过访问 HBase 里面已经挖掘好的人脸特征数据进行其他的应用。

HBase 基本知识介绍及典型案例分析

吴阳平 阿里巴巴 HBase 业务架构师

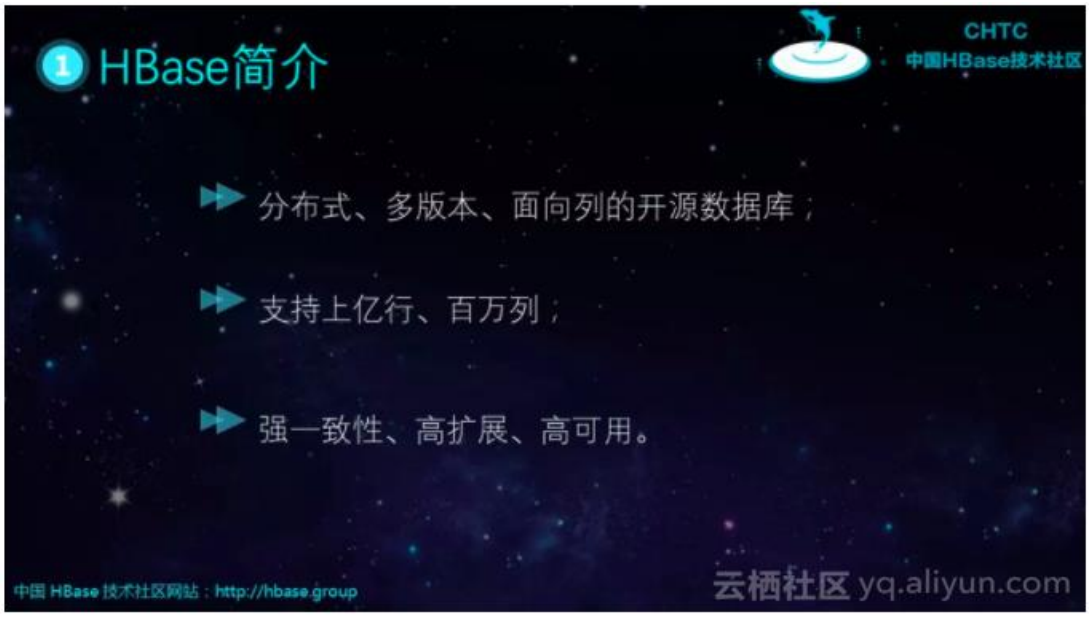
本文来自于 2018 年 10 月 20 日由中国 HBase 技术社区在武汉举办的中国 HBase Meetup 第六次线下交流会。HBase 基本知识介绍及典型案例分析 PPT 下载：https://yq.aliyun.com/download/3259

本次分享的内容主要分为以下五点

1. HBase 基本知识
2. HBase 读写流程
3. RowKey 设计要点
4. HBase 生态介绍
5. HBase 典型案例分析

1. HBase 基本知识

首先我们简单介绍一下 HBase 是什么？



HBase 最开始是受 Google 的 BigTable 启发而开发的分布式、多版本、面向列的开源数据库。其主要特点是支持上亿行、百万列，支持强一致性、并且具有高扩展、高可用等特点。

既然 HBase 是一种分布式的数据库，那么其和传统的 RMDB 有什么区别的呢？我们先来看看HBase表核心概念，理解这些基本的核心概念对后面我理解 HBase 的读写以及如何设计 HBase 表有着重要的联系。

HBase 表主要由以下几个元素组成：

1. RowKey：表中每条记录的主键；
2. Column Family：列族，将表进行横向切割，后面简称 CF；
3. Column：属于某一个列族，可动态添加列；
4. Version Number：类型为 Long，默认值是系统时间戳，可由用户自定义；
5. Value：真实的数据。

大家可以从上面的图看出：一行（Row）数据是可以包含一个或多个 Column Family，但是我们并不推荐一张 HBase 表的 Column Family 超过三个。Column 是属于 Column Family 的，一个 Column Family 包含一个或多个 Column。

在物理层面上，所有的数据其实是存放在 Region 里面的，而 Region 又由 RegionServer 管理，其对于的关系如下：

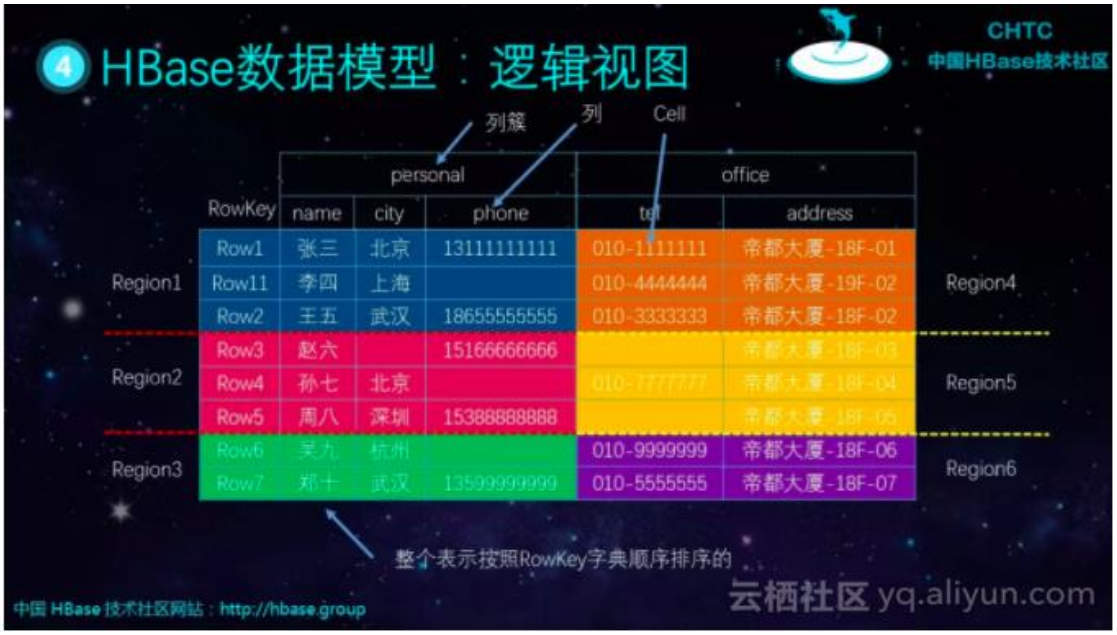
1. Region：一段数据的集合；
2. RegionServer：用于存放 Region 的服务。

从上面的图也可以清晰看到，一个 RegionServer 管理多个 Region；而一个 Region 管理一个或多个 Column Family。

2. HBase 读写流程

到这里我们已经了解了 HBase 表的组成，但是 HBase 表里面的数据到底是怎么存储的呢？

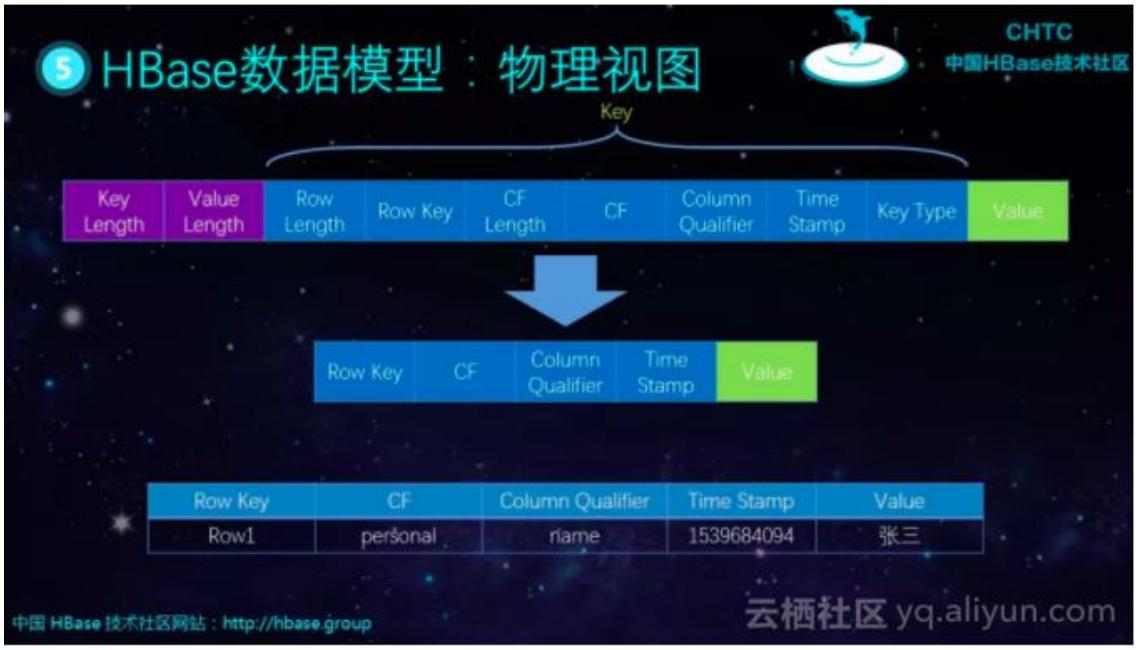
上面是一张从逻辑上看 HBase 表形式，这个和关系型数据库很类似。那么如果我们再深入看，可以看出，这张表的划分可以如下图表示。



从上图大家可以明显看出，这张表有两个 Column Family ，分别为 personal 和 office。而 personal 又有三列 name、city 以及 phone；office 有两列 tel 以及 address。由于存储在 HBase 里面的表一般有上亿行，所以 HBase 表会对整个数据按照 RowKey 进行字典排序，然后再对这张表进行横向切割。切割出来的数据是存储在 Region 里面，而不同的 Column Family 虽然属于一行，但是其在底层存储是放在不同的 Region 里。所以这张表我用了六种颜色表示，也就是

说，这张表的数据会被放在六个 Region 里面的，这就可以把数据尽可能的分散到整个集群。

在前面我们介绍了 HBase 其实是面向列的数据库，所以说一行 HBase 的数据其实是分了好几行存储，一个列对应一行，HBase 的 KV 结构如下：



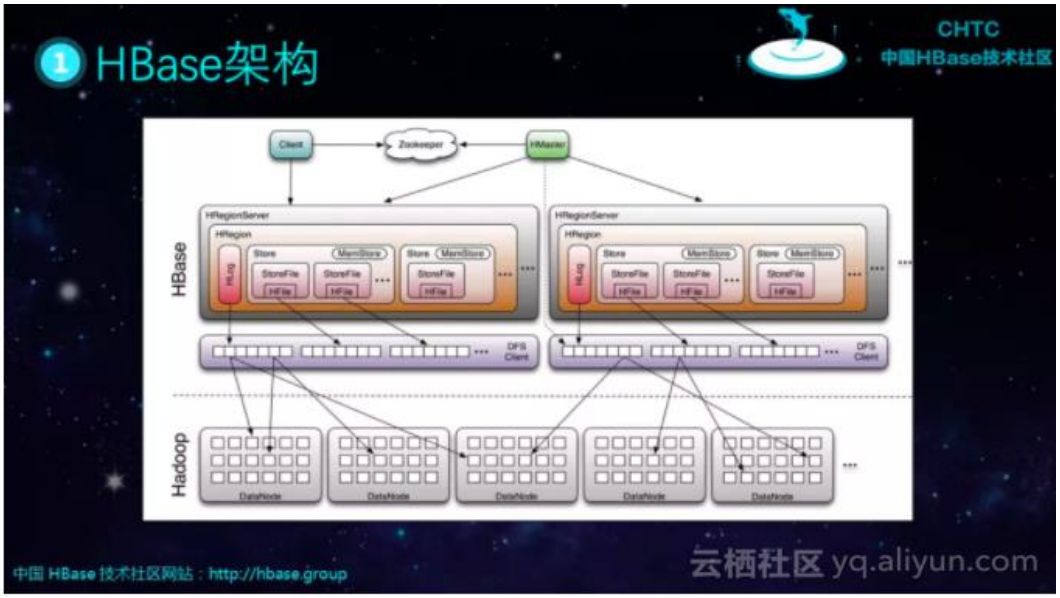
为了简便期间，在后面的表示我们删除了类似于 Key Length 的属性，只保留 Row Key、Column Family、Column Qualifier 等信息。所以 RowKey 为 Row1 的数据第一列表示为上图最后一行的形式。以此类推，整个表的存储就可以如下表示：

大家可以从上面的 kv 表现形式看出，Row11 的 phone 这列其实是没有数据的，在 HBase 的底层存储里面也就没有存储这列了，这点和我们传统的关系型数据库有很大的区别，有了这个特点， HBase 特别适合存储稀疏表。我们前面也将了 HBase 其实是多版本的，那如果我们修改了 HBase 表的一列，HBase 又是如何存储的呢？

比如上如中我们将 Row1 的 city 列从北京修改为上海了，如果使用 KV 表示的话，我们可以看出其实底层存储了两条数据，这两条数据的版本是不一样的，最新的一条数据版本比之前的新。总结起来就是：

1. HBase 支持数据多版本特性，通过带有不同时间戳的多个 KeyValue 版本来实现的；
2. 每次 put，delete 都会产生一个新的 Cell，都拥有一个版本；
3. 默认只存放数据的三个版本，可以配置；
4. 查询默认返回最新版本的数据，可以通过制定版本号或版本数获取旧数据。

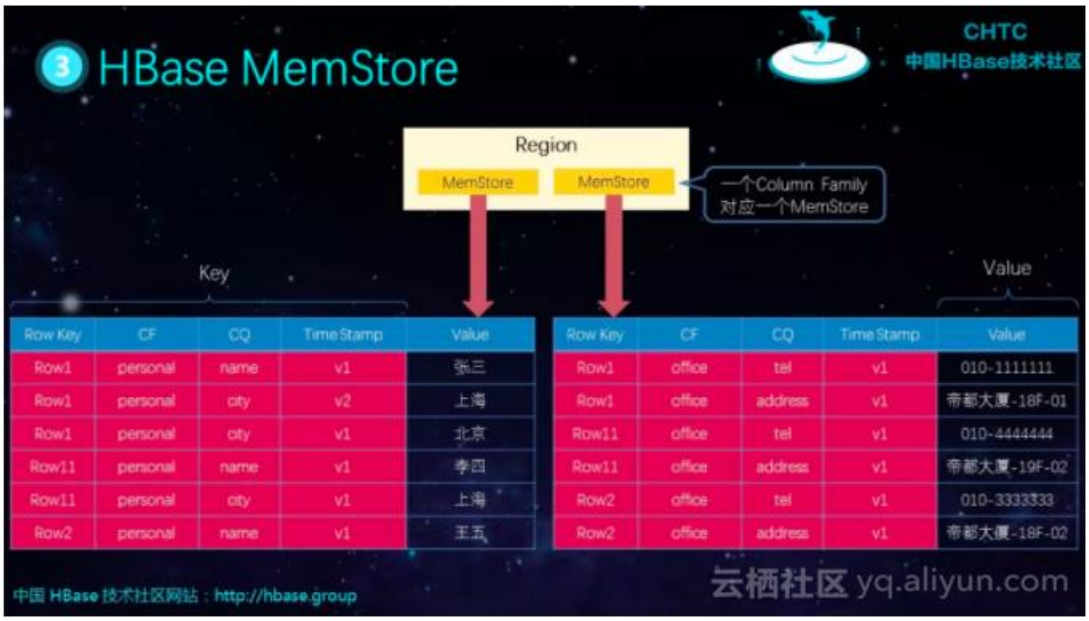
到这里我们已经了解了 HBase 表及其底层的 KV 存储了，现在让我们来了解一下 HBase 是如何读写数据的。首先我们来看看 HBase 的架构设计，这种图来自于社区：



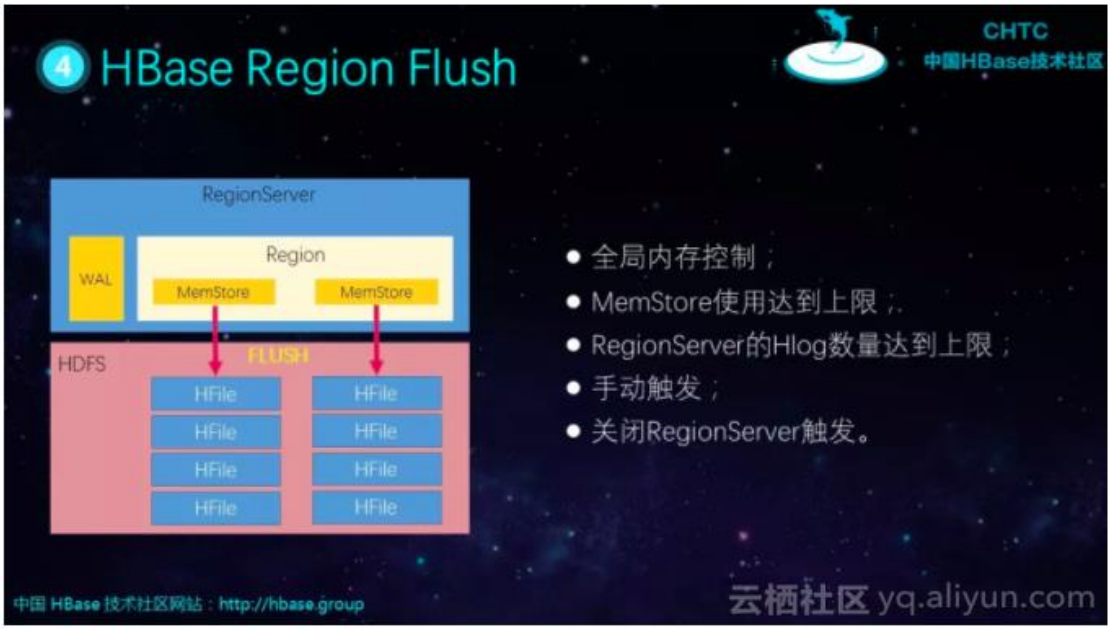
HBase 的写过程如下：

1. 先将数据写到 WAL 中；
2. WAL 存放在 HDFS 之上；
3. 每次 Put、Delete 操作的数据均追加到 WAL 末端；
4. 持久化到 WAL 之后，再写到 MemStore 中；
5. 两者写完返回 ACK 到客户端。

MemStore 其实是一种内存结构，一个 Column Family 对应一个 MemStore，MemStore 里面的数据也是对 Rowkey 进行字典排序的，如下：



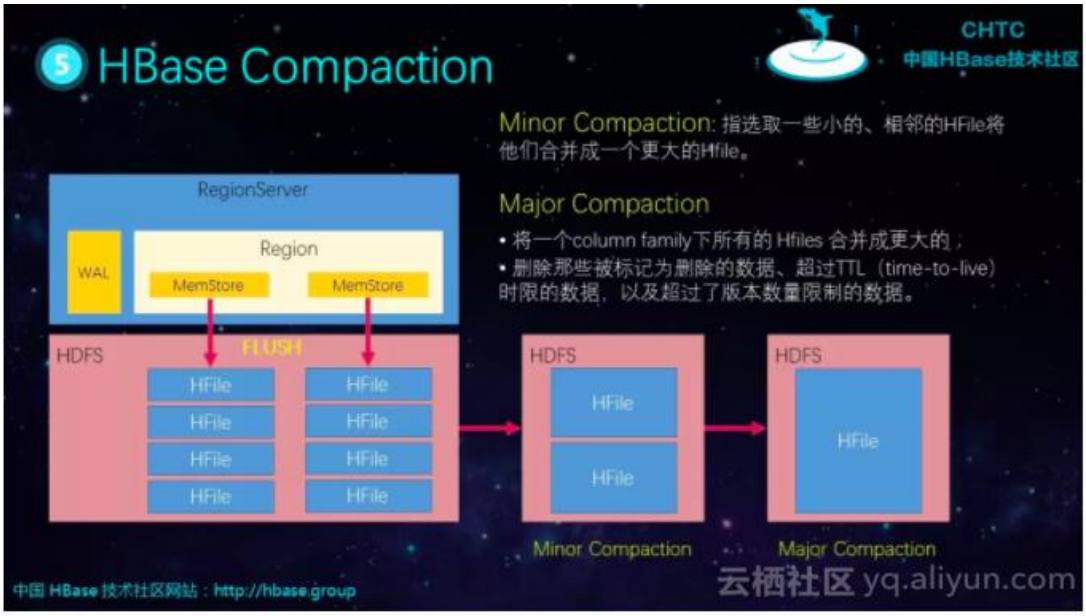
既然我们写数都是先写 WAL，再写 MemStore ，而 MemStore 是内存结构，所以 MemStore 总会写满的，将 MemStore 的数据从内存刷写到磁盘的操作成为 flush：



以下几种行为会导致 flush 操作

1. 全局内存控制；
2. MemStore 使用达到上限；
3. RegionServer 的 Hlog 数量达到上限；
4. 手动触发；
5. 关闭 RegionServer 触发。

每次 flush 操作都是将一个 MemStore 的数据写到一个 HFile 里面的，所以上图中 HDFS 上有许多个 HFile 文件。文件多了会对后面的读操作有影响，所以 HBase 会隔一定的时间将 HFile 进行合并。根据合并的范围不同分为 Minor Compaction 和 Major Compaction：

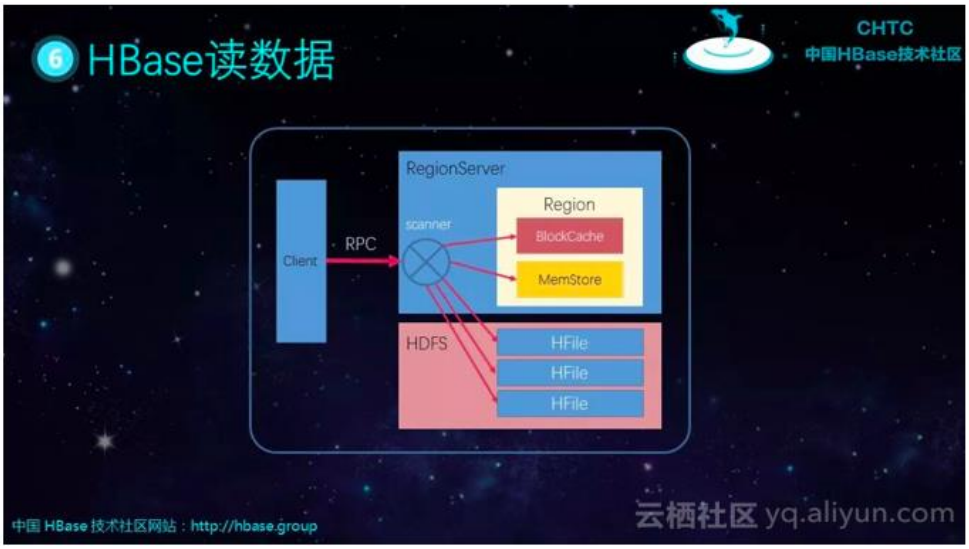


Minor Compaction: 指选取一些小的、相邻的 HFile 将他们合并成一个更大的 Hfile。

Major Compaction：

1. 将一个 column family 下所有的 Hfiles 合并成更大的；
2. 删除那些被标记为删除的数据、超过 TTL（time-to-live）时限的数据，以及超过了版本数量限制的数据。

HBase 读操作相对于写操作更为复杂，其需要读取 BlockCache、MemStore 以及 HFile。



上图只是简单的表示 HBase 读的操作，实际上读的操作比这个还要复杂，我这里就不深入介绍了。

到这里，有些人可能就想到了，前面我们说 HBase 表按照 Rowkey 分布到集群的不同机器上，那么我们如何去确定我们该读写哪些 RegionServer 呢？这就是 HBase Region 查找的问题，

客户端按照上面的流程查找需要读写的 RegionServer 。这个过程一般是第一次读写的时候进行的，在第一次读取到元数据之后客户端一般会把这些信息缓存到自己内存中，后面操作直接从内存拿就行。当然，后面元数据信息可能还会变动，这时候客户端会再次按照上面流程获取元数据。到这里整个读写流程得基本知识就讲完了。

3. RowKey 设计要点

现在我们来看看 HBase RowKey 的设计要点。我们一般都会说，看 HBase 设计的好不好，就看其 RowKey 设计的好不好，所以 RowKey 的设计在后面的写操作至关重要。我们先来看看 Rowkey 的作用

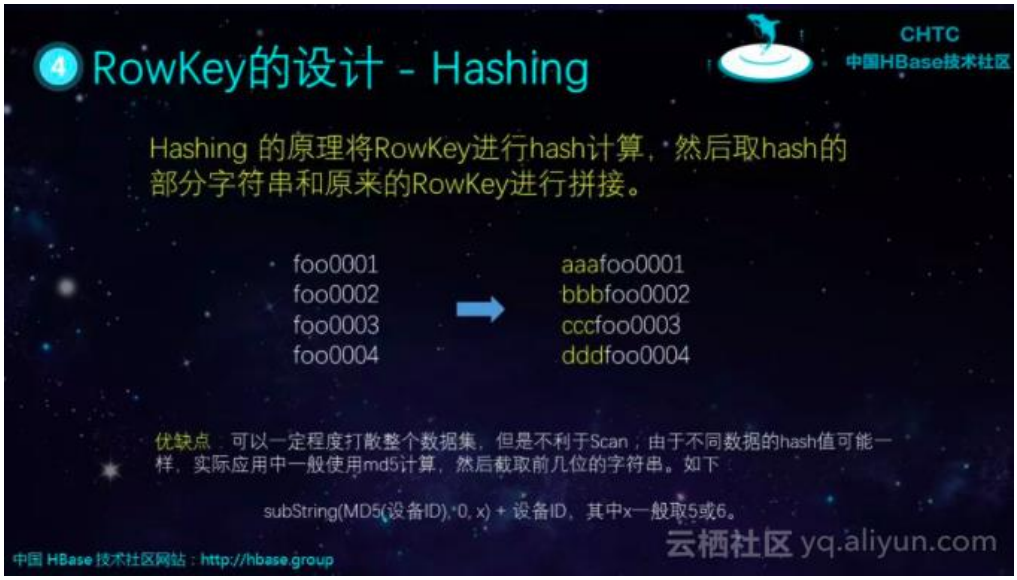
RowKey	personal			office	
	name	city	phone	tel	address
Row1	张三	上海	13111111111	010-1111111	帝都大厦-18F-01
Row11	李四	上海		010-4444444	帝都大厦-19F-02
Row2	王五	武汉	18655555555	010-3333333	帝都大厦-18F-02
Row3	赵六		15166666666		帝都大厦-18F-03
Row4	孙七	北京		010-7777777	帝都大厦-18F-04
Row5	周八	深圳	15388888888		帝都大厦-18F-05
Row6	吴九	杭州		010-9999999	帝都大厦-18F-06
Row7	郑十	武汉	13599999999	010-5555555	帝都大厦-18F-07

HBase 中的 Rowkey 主要有以下的作用：

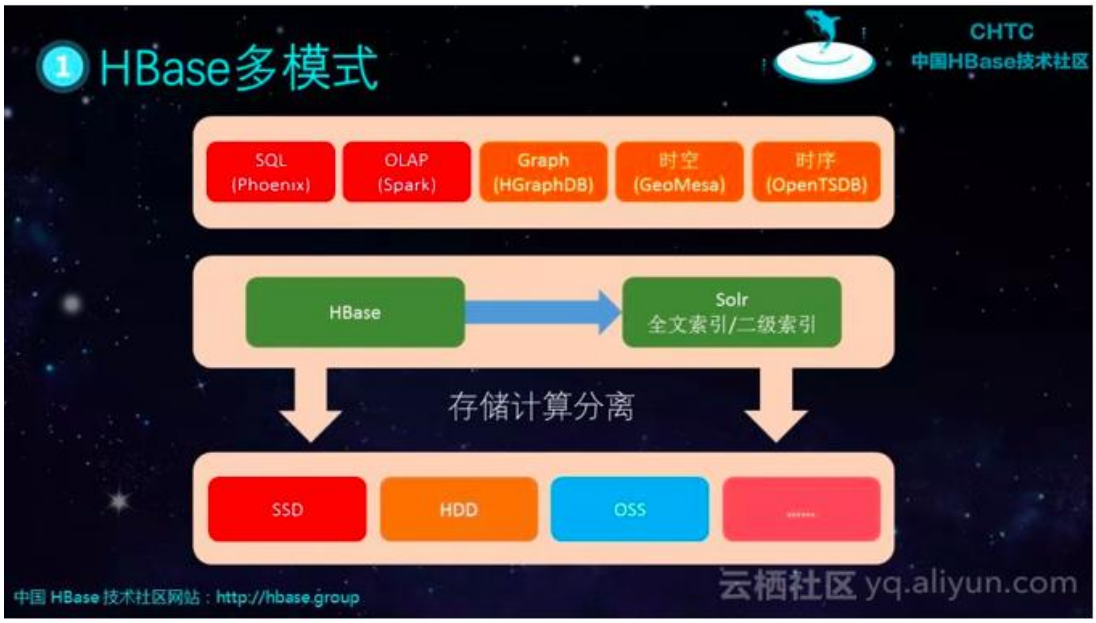
1. 读写数据时通过 Row Key 找到对应的 Region
2. MemStore 中的数据按 RowKey 字典顺序排序
3. HFile 中的数据按 RowKey 字典顺序排序

从下图可以看到，底层的 HFile 最终是按照 Rowkey 进行切分的，所以我们的设计原则是结合业务的特点，并考虑高频查询，尽可能的将数据打散到整个集群。

一定要充分分析清楚后面我们的表需要怎么查询。下面我们来看看三种比较场景的 Rowkey 设计方案。



这三种 Rowkey 的设计非常常见，具体的内容图片上也有了，我就不打文字了。数据如果只是存储在哪里其实并没有什么用，我们还需要有办法能够使用到里面的数据。幸好的是，当前 HBase 有许多的组件可以满足我们各种需求。如下图是 HBase 比较常用的组件：



4. HBase 生态介绍

HBase 的生态主要有：

1. Phoenix：主要提供使用 SQL 的方式来查询 HBase 里面的数据。一般能够在毫秒级别返回，比较适合 OLTP 场景。
2. Spark：我们可以使用 Spark 进行 OLAP 分析；也可以使用 Spark SQL 来满足比较复杂的 SQL 查询场景；使用 Spark Streaming 来进行实时流分析。
3. Solr：原生的 HBase 只提供了 Rowkey 单主键，如果我们需要对 Rowkey 之外的列进行查找，这时候就会有问题。幸好我们可以使用 Solr 来建立二级索引/全文索引充分满足我们的查询需求。
4. HGraphDB：HGraphDB 是分布式图数据库。依托图关联技术，帮助金融机构有效识别隐藏在网络中的黑色信息，在团伙欺诈、黑中介识别等。
5. GeoMesa：目前基于 NoSQL 数据库的时空数据引擎中功能最丰富、社区贡献人数最多的开源系统。
6. OpenTSDB：基于 HBase 的分布式的，可伸缩的时间序列数据库。适合做监控系统；譬如收集大规模集群（包括网络设备、操作系统、应用程序）的监控数据并进行存储，查询。

下面简单介绍一下这些组件。


2 Phoenix
CHTC
中国HBase技术社区
构建在HBase之上的关系型数据库层，支持使用SQL进行HBase数据的查询；
将用户编写的sql查询编译为一系列的scan操作，直接使用HBase的API。
结合协处理器和自定义的过滤器的话，小范围的查询在毫秒级响应，千万数据的话响应速度为秒级。
CREATE TABLE IF NOT EXISTS us_population (
state CHAR(2) NOT NULL,
city VARCHAR NOT NULL,
population BIGINT
CONSTRAINT my_pk PRIMARY KEY (state, city)
);
SELECT state as "State",count(city) as "City Count",sum(population) as "Population Sum"
FROM us_population
GROUP BY state
ORDER BY sum(population) DESC;
中国 HBase 技术社区网站：http://hbase.group
云栖社区 yq.aliyun.com


3 Spark
CHTC
中国HBase技术社区
OLAP；
利用Spark-SQL查询一些比较复杂的分析；
利用Spark Streaming进行实时流分析，结果存入HBase；
直接读取Hfile。
中国 HBase 技术社区网站：http://hbase.group
云栖社区 yq.aliyun.com


4 HGraphDB
CHTC
中国HBase技术社区
HGraphDB是分布式图数据库，底层基于HBase；
支持数百亿点与边的即时查询；
支持OLAP分析。
应用：依托图关联技术，帮助金融机构有效识别隐藏在网络中的黑色信息，在团伙欺诈、黑中介识别等。
用户
手机
账号
手机
IP
设备
卡
账号
账号
用户
卡
中国 HBase 技术社区网站：http://hbase.group
云栖社区 yq.aliyun.com


5 GeoMesa
CHTC
中国HBase技术社区
目前基于NoSQL数据库的时空数据引擎中功能最丰富、社区贡献人数最多的开源系统；
提供了多种空间索引方式供用户灵活选择；
提供了基于Coprocessor的空间查询方式，将计算过程放置在server端，能够减少通讯开销，从而获得很好的性能提升；
提供了丰富数据入库、操作等工具，便于用户处理数据；
提供了多种空间数据分析算法，如KNN、直方图、热点分析、TubeSelect等；
基于OGC标准设计，便于系统间的集成与互操作。
中国 HBase 技术社区网站：http://hbase.group
云栖社区 yq.aliyun.com

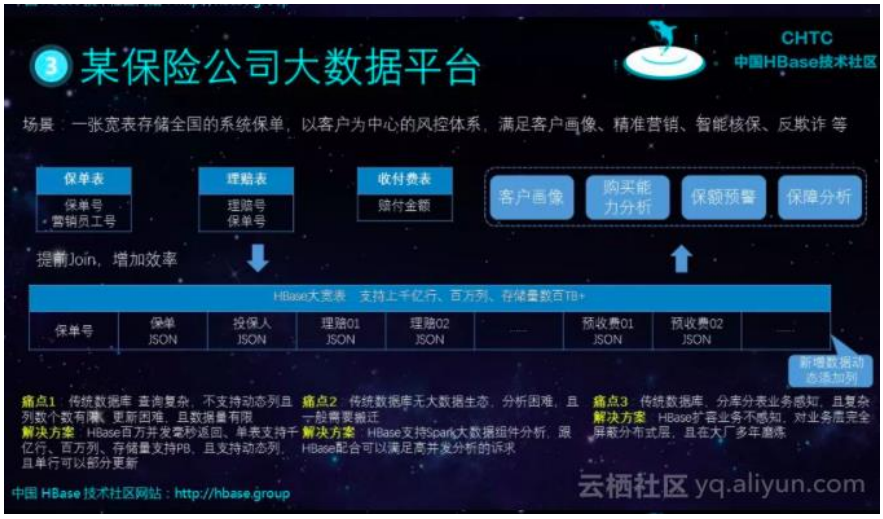

6 OpenTSDB
CHTC
中国HBase技术社区
基于HBase的分布式的，可伸缩的时间序列数据库；
海量数据存储、高并发高吞吐写；
适合做监控系统，譬如收集大规模集群（包括网络设备、操作系统、应用程序）的监控数据并进行存储，查询。
mysql.bytes_received 1287333217 327810227706 schema=foo host=db1
mysql.bytes_sent 1287333217 6604859181710 schema=foo host=db1
mysql.bytes_received 1287333232 327812421706 schema=foo host=db1
mysql.bytes_sent 1287333232 6604901075387 schema=foo host=db1
mysql.bytes_received 1287333321 340899533915 schema=foo host=db2
mysql.bytes_sent 1287333321 5506469130707 schema=foo host=db2
中国 HBase 技术社区网站：http://hbase.group
云栖社区 yq.aliyun.com


7 Solr
CHTC
中国HBase技术社区
基于Lucene的全文搜索引擎；
为HBase添加二级索引功能；
提供范围查找、模糊查找等。
Client
HBase
索引同步
Solr
HDFS
中国 HBase 技术社区网站：http://hbase.group
云栖社区 yq.aliyun.com

5. HBase 典型案例分析

有了这么多组件，我们都可以干什么呢？来看看 HBase 的典型案例。

HBase 在风控场景、车联网/物联网、广告推荐、电子商务等行业有着广泛的使用。下面是四个典型案例的架构，由于图片里有详细的文字，我就不再打出来了。



大家工作学习遇到 HBase 技术问题，把问题发布到 HBase 技术社区论坛 http://hbase.group，欢迎大家论坛上面提问留言讨论。想了解更多 HBase 技术关注 HBase 技术社区公众号(微信号:hbasegroup)，非常欢迎大家积极投稿。

HBase RowKey 设计指南

吴阳平 阿里巴巴 HBase 业务架构师

1.为什么 Rowkey 这么重要

1.1 RowKey 到底是什么

我们常说看一张 HBase 表设计的好不好，就看它的 RowKey 设计的好不好。可见 RowKey 在 HBase 中的地位。那么 RowKey 到底是什么？RowKey 的特点如下：

1. 类似于 MySQL、Oracle 中的主键，用于标示唯一的行；
2. 完全是由用户指定的一串不重复的字符串；
3. HBase 中的数据永远是根据 Rowkey 的字典排序来排序的。

1.2 RowKey 的作用

1. 读写数据时通过 RowKey 找到对应的 Region；
2. MemStore 中的数据按 RowKey 字典顺序排序；
3. HFile 中的数据按 RowKey 字典顺序排序。

1.3 Rowkey 对查询的影响

如果我们的 RowKey 设计为 uid+phone+name，那么这种设计可以很好的支持以下的场景：

uid = 111 AND phone = 123 AND name = iteblog
uid = 111 AND phone = 123
uid = 111 AND phone = 12?
uid = 111

难以支持的场景：

phone = 123 AND name = iteblog
phone = 123
name = iteblog

1.4 Rowkey 对 Region 划分影响

HBase 表的数据是按照 Rowkey 来分散到不同 Region，不合理的 Rowkey 设计会导致热点问题。热点问题是大量的 Client 直接访问集群的一个或极少数个节点，而集群中的其他节点却处于相对空闲状态。

如上图，Region1 上的数据是 Region 2 的 5 倍，这样会导致 Region1 的访问频率比较高，进而影响这个 Region 所在机器的其他 Region。

2.RowKey 设计技巧

我们如何避免上面说到的热点问题呢？这就是这章节谈到的三种方法。

2.1 避免热点的方法 – Salting

这里的加盐不是密码学中的加盐，而是在 rowkey 的前面增加随机数。具体就是给 rowkey 分配一个随机前缀 以使得它和之前排序不同。分配的前缀种类数量应该和你想使数据分散到不同的 region 的数量一致。 如果你有一些 热点 rowkey 反复出现在其他分布均匀的 rwokey 中，加盐是很有用的。考虑下面的例子：它将写请求分散到多个 RegionServers，但是对读造成了一些负面影响。

假如你有下列 rowkey，你表中每一个 region 对应字母表中每一个字母。 以 'a' 开头是同一个 region，'b'开头的是同一个 region。在表中，所有以 'f'开头的都在同一个 region， 它们的 rowkey 像下面这样：

```
foo0001
foo0002
foo0003
foo0004
```

现在，假如你需要将上面这个 region 分散到 4 个 region。你可以用 4 个不同的盐：'a', 'b', 'c', 'd'.在这个方案下，每一个字母前缀都会在不同的 region 中。加盐之后，你有了下面的 rowkey:

```
a-foo0003
b-foo0001
c-foo0004
d-foo0002
```

所以，你可以向 4 个不同的 region 写，理论上说，如果所有人都向同一个 region 写的话，你将拥有之前 4 倍的吞吐量。

现在，如果再增加一行，它将随机分配 a,b,c,d 中的一个作为前缀，并以一个现有行作为尾部结束：

```
a-foo0003
b-foo0001
c-foo0003
c-foo0004
d-foo0002
```

因为分配是随机的，所以如果你想要以字典序取回数据，你需要做更多工作。加盐这种方式增加了写时的吞吐量，但是当读时有了额外代价。

2.2 避免热点的方法 - Hashing

Hashing 的原理是计算 RowKey 的 hash 值，然后取 hash 的部分字符串和原来的 RowKey 进行拼接。这里说的 hash 包含 MD5、sha1、sha256 或 sha512 等算法。比如我们有如下的 RowKey：

```
foo0001
foo0002
foo0003
foo0004
```

我们使用 md5 计算这些 RowKey 的 hash 值，然后取前 6 位和原来的 RowKey 拼接得到新的 RowKey：

```
95f18cfoo0001
6ccc20foo0002
b61d00foo0003
1a7475foo0004
```

优缺点：可以一定程度打散整个数据集，但是不利于 Scan；比如我们使用 md5 算法，来计算 Rowkey 的 md5 值，然后截取前几位的字符串。subString(MD5(设备 ID), 0, x) + 设备 ID，其中 x 一般取 5 或 6。

2.3 避免热点的方法 - Reversing

Reversing 的原理是反转一段固定长度或者全部的键。比如我们有以下 URL，并作为 RowKey：

```
flink.iteblog.com
www.iteblog.com
carbondata.iteblog.com
def.iteblog.com
```

这些 URL 其实属于同一个域名，但是由于前面不一样，导致数据不在一起存放。我们可以对其进行反转，如下：

```
moc.golbeti.knilf
moc.golbeti.www
moc.golbeti.atadnobrac
moc.golbeti.fed
```

经过这个之后，这些 URL 的数据就可以放一起了。

2.4 RowKey 的长度

RowKey 可以是任意的字符串，最大长度 64KB（因为 Rowlength 占 2 字节）。建议越短越好，原因如下：

1. 数据的持久化文件 HFile 中是按照 KeyValue 存储的，如果 rowkey 过长，比如超过 100 字节，1000w 行数据，光 rowkey 就要占用 100*1000w=10 亿个字节，将近 1G 数据，这样会极大影响 HFile 的存储效率；
2. MemStore 将缓存部分数据到内存，如果 rowkey 字段过长，内存的有效利用率就会降低，系统不能缓存更多的数据，这样会降低检索效率；
3. 目前操作系统都是 64 位系统，内存 8 字节对齐，控制在 16 个字节，8 字节的整数倍利用了操作系统的最佳特性。

3.RowKey 设计案例剖析

3.1 交易类表 Rowkey 设计

查询某个卖家某段时间内的交易记录
sellerId + timestamp + orderId

查询某个买家某段时间内的交易记录
buyerId + timestamp + orderId

根据订单号查询
orderNo

如果某个商家卖了很多商品，可以如下设计 Rowkey 实现快速搜索 salt + sellerId + timestamp 其中，salt 是随机数。
可以支持的场景：

1. 全表 Scan
2. 按照 sellerId 查询
3. 按照 sellerId + timestamp 查询

3.2 金融风控 Rowkey 设计

查询某个用户的用户画像数据

1. prefix + uid
2. prefix + idcard
3. prefix + tele

其中 prefix = substr(md5(uid),0 ,x)， x 取 5-6。uid、idcard 以及 tele 分别表示用户唯一标识符、身份证、手机号码。

3.3 车联网 Rowkey 设计

查询某辆车在某个时间范围的交易记录
carId + timestamp

某批次的车太多，造成热点
prefix + carId + timestamp 其中 prefix = substr(md5(uid),0 ,x)

3.4 查询最近的数据

查询用户最新的操作记录或者查询用户某段时间的操作记录，RowKey 设计如下：
uid + Long.Max_Value - timestamp
支持的场景

查询用户最新的操作记录

Scan [uid] startRow [uid][000000000000] stopRow [uid][Long.Max_Value - timestamp]

查询用户某段时间的操作记录

Scan [uid] startRow [uid][Long.Max_Value – startTime] stopRow [uid][Long.Max_Value - endTime]

如果 RowKey 无法满足我们的需求，可以尝试二级索引。Phoenix、Solr 以及 ElasticSearch 都可以用于构建二级索引。

HBase 实战之 MOB 使用指南

1. 背景

HBase 可以很方便的将图片、文本等文件以二进制的方式进行存储。虽然 HBase 一般可以处理从 1 字节到 10MB 大小的二进制对象，但是 HBase 通常对于读写路径的优化主要是针对小于 100KB 的值。当 HBase 处理数据为 100KB~10MB 时，由于分裂（split）和压缩（compaction）会引起写的放大，从而会降低 HBase 性能。所以在 HBase2.0+引入了 MOB 特性，这样保持了 HBase 的高性能、强一致性和低开销。

若要启用 MOB 功能，需要在每个 RegionServer 进行配置，并在建表或者修改表时对指定列族启用 MOB 特性。在 HBase 尝鲜版中启用 MOB 功能，需要由 admin 用户设置定期进程，以重新优化 MOB 数据的分布。

2. 启用和配置 RegionServer 上的 MOB 特性

增加或者修改 hbase-site.xml 文件中的某些配置

2.1 设置 MOB 文件的缓存配置

```
<property>
    <name>hbase.mob.file.cache.size</name>
    <value>1000</value>
</property>
<property>
    <name>hbase.mob.cache.evict.period</name>
    <value>3600</value>
</property>
<property>
    <name>hbase.mob.cache.evict.remain.ratio</name>
    <value>0.5f</value>
 </property>
```

说明：

1. hbase.mob.file.cache.size 打开的文件句柄缓存数，默认值是 1000。通过增加文件句柄数可以提高读的性能，可以减少频繁的打开、关闭文件。若

这个值设置过大，会导致“too many opened file handers”。

2. hbase.mob.cache.evict.period MOB 缓存淘汰缓存的 MOB 文件时间间隔（以秒为单位），默认值为 3600 秒。
3. hbase.mob.cache.evict.remain.ratio 当缓存的 MOB 文件数目超过 hbase.mob.file.cache.size 设置的数目后，会触发 MOB 缓存淘汰机制（eviction），0.5f 为剩余的 MOB 缓存比率（0~1），默认的比率为 0.5f。

2.2 配置 MobMasterObserver 作为协处理器的 master

主要用于表在删除后，MOB 文件的归档。

```
<property>
   <name>hbase.coprocessor.master.classes</name>
   <value>org.apache.hadoop.hbase.coprocessor.MobMasterObserver</value>
</property>
```

MOB 的管理 MOB 特性为 HBase 引入新的读写路径，此时我们采用外部工具对其进行优化处理，一个是 expiredMobFileCleanerTTL 处理 TTL 和时间的过期数据，另一个是清理工具用来合并小的 MOB 文件或者多次更新、删除的 MOB 文件。

2.2.1 清理过期的 MOB 数据（expiredMobFileCleaner）

```
org.apache.hadoop.hbase.mob.compactions.expiredMobFileCleaner tableName familyName
```

设置清理延时

```
<property>
    <name>hbase.mob.cleaner.delay</name>
    <value>60 * 60 * 1000</value>
</property>
```

2.2.2 清理工具

```
org.apache.hadoop.hbase.mob.compactions.Sweeper tableName familyName
```

属性值设置如下：

```
<property>
    <name>hbase.mob.compaction.invalid.file.ratio</name>
    <value>0.3f</value>
</property>
<property>
    <name>hbase.mob.compaction.small.file.threshold</name>
    <value>67108864</value>
</property>
<property>
    <name>hbase.mob.compaction.memstore.flush.size</name>
    <value>134217728</value>
</property>
```

说明：

1. hbase.mob.compaction.invalid.file.ratio 如果在 MOB 文件中删除了太多的单元格，则被视为作为无效文件，需要重新写入或者合并。当 MOB 文件（mobFileSize）大小减去存在的单元格（existingCellsSize）大小之差除以 MOB 文件（mobFileSize）的比率小于设定的值时，我们就认为其为无效文件。默认值为 0.3f。
2. hbase.mob.compaction.small.file.threshold 如果 MOB 的大小小于阈值，则视为小文件，需要合并。默认值为 64MB。
3. hbase.mob.compaction.memstore.flush.size MOB 里 memstore 大小，超过此大小就会 flush，并且每个 sweep reducer 拥有各自 memstore。

警告

使用清理工具最坏的情况 :MOB 文件压缩合并成功，但是相关的(put)更新失败。这意味着新的 MOB 文件已经创建但未能将新的 MOB 文件路径。存入 HBase 中，因此 HBase 不会指向这些 MOB 文件。

小贴士

请检查 yarn-site.xml 的配置，在 yarn.application.classpath 中添加 hbase 的安装路径：$HBASE_HOME/* 和 hbase 的 lib 路径: $HBASE_HOME/lib/*

```
<property>
    <description>Classpath for typical applications.</description>
     <name>yarn.application.classpath</name>
     <value>
      $HADOOP_CONF_DIR
        $HADOOP_COMMON_HOME/*,$HADOOP_COMMON_HOME/lib/*
        $HADOOP_HDFS_HOME/*,$HADOOP_HDFS_HOME/lib/*,
        $HADOOP_MAPRED_HOME/*,$HADOOP_MAPRED_HOME/lib/*
        $HADOOP_YARN_HOME/*,$HADOOP_YARN_HOME/lib/*
        $HBASE_HOME/*, $HBASE_HOME/lib/*
    </value>
</property>
```

在表中开启 MOB 特性 a 将列族设置为 MOB

```
HColumnDescriptor hcd = new HColumnDescriptor("f");
hcd.setValue(MobConstants.IS_MOB, Bytes.toBytes(Boolean.TRUE));
```

2.2.3 设置 MOB 单元格的阈值，默认为 102400

```
HColumnDescriptor hcd;
hcd.setValue(MobConstants.MOB_THRESHOLD, Bytes.toBytes(102400L);
```

对于客户端而言，MOB 单元格操作和普通单元格类似。

2.2.4 插入 MOB 值

```
KeyValue kv = new KeyValue(row1, family, qf1, ts, KeyValue.Type.Put, value );
Put put = new Put(row1);
put.add(kv);
region.put(put);
```

2.2.5 获取 MOB 值

```
Scan scan = new Scan();
InternalScanner scanner = (InternalScanner) region.getScanner(scan);
scanner.next(result, limit);
```

2.2.6 获取 MOB 所有数据（raw =true）

```
Scan scan = new Scan();
scan.setAttribute(MobConstants.MOB_SCAN_RAW, Bytes.toBytes(Boolean.TRUE));
InternalScanner scanner = (InternalScanner) region.getScanner(scan);
scanner.next(result, limit);
```

运行一个 MOB 示例

```
sudo -u hbase hbase org.apache.hadoop.hbase.IntegrationTestIngestMOB
```

HBase 最佳实践 – 读性能优化策略

范欣欣 网易杭州研究院后台技术中心 数据库技术组

摘要

任何系统都会有各种各样的问题，有些是系统本身设计问题，有些却是使用姿势问题。HBase 也一样，在真实生产线上大家或多或少都会遇到很多问题，有些是 HBase 还需要完善的，有些是我们确实对它了解太少。总结起来，大家遇到的主要问题无非是 Full GC 异常导致宕机问题、RIT 问题、写吞吐量太低以及读延迟较大。

Full GC 问题之前在一些文章里面已经讲过它的来龙去脉，主要的解决方案目前主要有两方面需要注意，一方面需要查看 GC 日志确认是哪种 Full GC，根据 Full GC 类型对 JVM 参数进行调优，另一方面需要确认是否开启了 BucketCache 的 offheap 模式，建议使用 LRUBlockCache 的童鞋尽快转移到 BucketCache 来。当然我们还是很期待官方 2.0.0 版本发布的更多 offheap 模块。

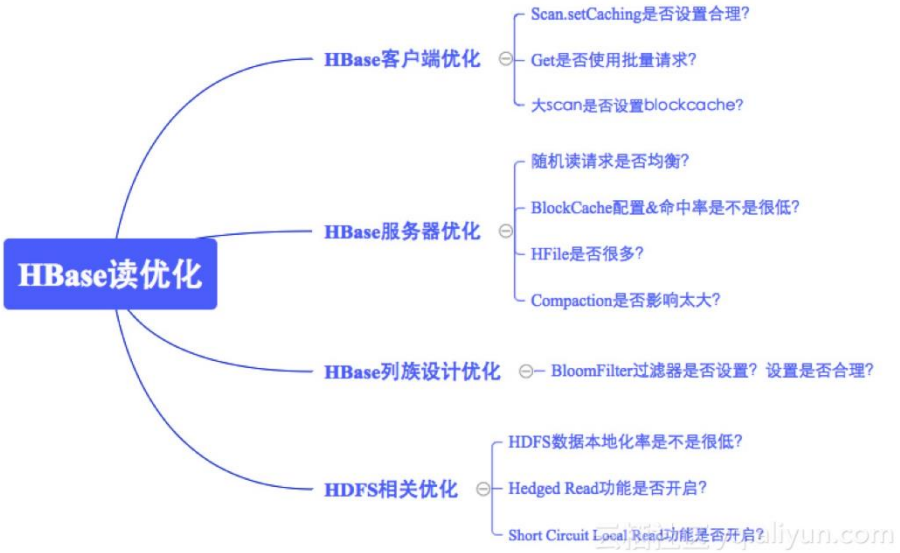
RIT 问题，我相信更多是因为我们对其不了解，具体原理可以戳这里，解决方案目前有两个，优先是使用官方提供的 HBCK 进行修复（HBCK 本人一直想拿出来分享，但是目前案例还不多，等后面有更多案例的话再拿出来说），使用之后还是解决不了的话就需要手动修复文件或者元数据表。

而对于写吞吐量太低以及读延迟太大的优化问题，笔者也和很多朋友进行过探讨，这篇文章就以读延迟优化为核心内容展开，具体分析 HBase 进行读延迟优化的那些套路，以及这些套路之后的具体原理。希望大家在看完之后能够结合这些套路剖析自己的系统。

一般情况下，读请求延迟较大通常存在三种场景，分别为：

1. 仅有某业务延迟较大，集群其他业务都正常
2. 整个集群所有业务都反映延迟较大
3. 某个业务起来之后集群其他部分业务延迟较大

这三种场景是表象，通常某业务反应延迟异常，首先需要明确具体是哪种场景，然后针对性解决问题。下图是对读优化思路的一点总结，主要分为四个方面：客户端优化、服务器端优化、列族设计优化以及 HDFS 相关优化，下面每一个小点都会按照场景分类，文章最后进行归纳总结。下面分别进行详细讲解：



1. HBase 客户端优化

和大多数系统一样，客户端作为业务读写的入口，姿势使用不正确通常会导致本业务读延迟较高实际上存在一些使用姿势的推荐用法，这里一般需要关注四个问题：

1.1 scan 缓存是否设置合理？

优化原理：在解释这个问题之前，首先需要解释什么是 scan 缓存，通常来讲一次 scan 会返回大量数据，因此客户端发起一次 scan 请求，实际并不会一次就将所有数据加载到本地，而是分成多次 RPC 请求进行加载，这样设计一方面是因为大量数据请求可能会导致网络带宽严重消耗进而影响其他业务，另一方面也有可能因为数据量太大导致本地客户端发生 OOM。在这样的设计体系下用户会首先加载一部分数据到本地，然后遍历处理，再加载下一部分数据到本地处理，如此往复，直至所有数据都加载完成。数据加载到本地就存放在 scan 缓存中，默认 100 条数据大小。

通常情况下，默认的 scan 缓存设置就可以正常工作的。但是在一些大 scan（一次 scan 可能需要查询几万甚至几十万行数据）来说，每次请求 100 条数据意味着一次 scan 需要几百甚至几千次 RPC 请求，这种交互的代价无疑是很大的。因此可以考虑将 scan 缓存设置增大，比如设为 500 或者 1000 就可能更加合适。笔者之前做过一次试验，在一次 scan 扫描 10w+条数据量的条件下，将 scan 缓存从 100 增加到 1000，可以有效降低 scan 请求的总体延迟，延迟基本降低了 25%左右。

优化建议：大 scan 场景下将 scan 缓存从 100 增大到 500 或者 1000，用以减少 RPC 次数。

1.2 get 请求是否可以使用批量请求？

优化原理：HBase 分别提供了单条 get 以及批量 get 的 API 接口，使用批量 get 接口可以减少客户端到 RegionServer 之间的 RPC 连接数，提高读取性能。另外需要注意的是，批量 get 请求要么成功返回所有请求数据，要么抛出异常。

优化建议：使用批量 get 进行读取请求

1.3 请求是否可以显示指定列族或者列？

优化原理：HBase 是典型的列族数据库，意味着同一列族的数据存储在一起，不同列族的数据分开存储在不同的目录下。如果一个表有多个列族，只是根据 Rowkey 而不指定列族进行检索的话不同列族的数据需要独立进行检索，性能必然会比指定列族的查询差很多，很多情况下甚至会有 2 倍~3 倍的性能损失。

优化建议：可以指定列族或者列进行精确查找的尽量指定查找

1.4 离线批量读取请求是否设置禁止缓存？

优化原理：通常离线批量读取数据会进行一次性全表扫描，一方面数据量很大，另一方面请求只会执行一次。这种场景下如果使用 scan 默认设置，就会将数据从 HDFS 加载出来之后放到缓存。可想而知，大量数据进入缓存必将其他实时业务热点数据挤出，其他业务不得不从 HDFS 加载，进而会造成明显的读延迟毛刺

优化建议：离线批量读取请求设置禁用缓存，scan.setBlockCache(false)

2. HBase 服务器端优化

一般服务端端问题一旦导致业务读请求延迟较大的话，通常是集群级别的，即整个集群的业务都会反映读延迟较大。可以从 4 个方面入手：

2.1 读请求是否均衡？

优化原理：极端情况下假如所有的读请求都落在一台 RegionServer 的某几个 Region 上，这一方面不能发挥整个集群的并发处理能力，另一方面势必造成此台 RegionServer 资源严重消耗（比如 IO 耗尽、handler 耗尽等），落在该台 RegionServer 上的其他业务会因此受到很大的波及。可见，读请求不均衡不仅会造成本身业务性能很差，还会严重影响其他业务。当然，写请求不均衡也会造成类似的问题，可见负载不均衡是 HBase 的大忌。

观察确认：观察所有 RegionServer 的读请求 QPS 曲线，确认是否存在读请求不均衡现象

优化建议：RowKey 必须进行散列化处理（比如 MD5 散列），同时建表必须进行预分区处理

2.2 BlockCache 是否设置合理？

优化原理：BlockCache 作为读缓存，对于读性能来说至关重要。默认情况下 BlockCache 和 Memstore 的配置相对比较均衡（各占 40%），可以根据集群业务进行修正，比如读多写少业务可以将 BlockCache 占比调大。另一方面，BlockCache 的策略选择也很重要，不同策略对读性能来说影响并不是很大，但是对 GC 的影响却相当显著，尤其 BucketCache 的 offheap 模式下 GC 表现很优越。另外，HBase 2.0 对 offheap 的改造（HBASE-11425）将会使 HBase 的读性能得到 2 ~ 4 倍的提升，同时 GC 表现会更好！

观察确认：观察所有 RegionServer 的缓存未命中率、配置文件相关配置项一级 GC 日志，确认 BlockCache 是否可以优化

优化建议：JVM 内存配置量 < 20G，BlockCache 策略选择 LRUBlockCache；否则选择 BucketCache 策略的 offheap 模式；期待 HBase 2.0 的到来！

2.3 HFile 文件是否太多？

优化原理：HBase 读取数据通常首先会到 Memstore 和 BlockCache 中检索（读取最近写入数据&热点数据），如果查找不到就会到文件中检索。HBase 的类 LSM 结构会导致每个 store 包含多数 HFile 文件，文件越多，检索所需的 IO 次数必然越多，读取延迟也就越高。文件数量通常取决于 Compaction 的执行策略，一般和两个配置参数有关：hbase.hstore.compactionThreshold 和 hbase.hstore.compaction.max.size，前者表示一个 store 中的文件数超过多少就应该进行合并，后者表示参数合并的文件大小最大是多少，超过此大小的文件不能参与合并。这两个参数不能设置太'松'（前者不能设置太大，后者不能设置太小），导致 Compaction 合并文件的实际效果不明显，进而很多文件得不到合并。这样就会导致 HFile 文件数变多。

观察确认：观察 RegionServer 级别以及 Region 级别的 storefile 数，确认 HFile 文件是否过多

优化建议：hbase.hstore.compactionThreshold 设置不能太大，默认是 3 个；设置需要根据 Region 大小确定，通常可以简单的认为 hbase.hstore.compaction.max.size = RegionSize / hbase.hstore.compactionThreshold

2.4 Compaction 是否消耗系统资源过多？

优化原理：Compaction 是将小文件合并为大文件，提高后续业务随机读性能，但是也会带来 IO 放大以及带宽消耗问题（数据远程读取以及三副本写入都会消耗系统带宽）。正常配置情况下 Minor Compaction 并不会带来很大的系统资源消耗，除非因为配置不合理导致 Minor Compaction 太过频繁，或者 Region 设置太大情况下发生 Major Compaction。

观察确认：观察系统 IO 资源以及带宽资源使用情况，再观察 Compaction 队列长度，确认是否由于 Compaction 导致系统资源消耗过多

优化建议：

1. Minor Compaction 设置：hbase.hstore.compactionThreshold 设置不能太小，又不能设置太大，因此建议设置为 5 ～ 6；hbase.hstore.compaction.max.size = RegionSize / hbase.hstore.compactionThreshold

2. Major Compaction 设置：大 Region 读延迟敏感业务（ 100G 以上）通常不建议开启自动 Major Compaction，手动低峰期触发。小 Region 或者延迟不敏感业务可以开启 Major Compaction，但建议限制流量；

3. 期待更多的优秀 Compaction 策略，类似于 stripe-compaction 尽早提供稳定服务

3. HBase 列族设计优化

HBase 列族设计对读性能影响也至关重要，其特点是只影响单个业务，并不会对整个集群产生太大影响。列族设计主要从两个方面检查：

3.1 Bloomfilter 是否设置？是否设置合理？

优化原理 :Bloomfilter 主要用来过滤不存在待检索 RowKey 或者 Row-Col 的 HFile 文件，避免无用的 IO 操作。它会告诉你在这个 HFile 文件中是否可能存在待检索的 KV，如果不存在，就可以不用消耗 IO 打开文件进行 seek。很显然，通过设置 Bloomfilter 可以提升随机读写的性能。

Bloomfilter 取值有两个，row 以及 rowcol，需要根据业务来确定具体使用哪种。如果业务大多数随机查询仅仅使用 row 作为查询条件，Bloomfilter 一定要设置为 row，否则如果大多数随机查询使用 row+cf 作为查询条件，Bloomfilter 需要设置为 rowcol。如果不确定业务查询类型，设置为 row。

优化建议：任何业务都应该设置 Bloomfilter，通常设置为 row 就可以，除非确认业务随机查询类型为 row+cf，可以设置为 rowcol

4. HDFS 相关优化

HDFS 作为 HBase 最终数据存储系统，通常会使用三副本策略存储 HBase 数据文件以及日志文件。从 HDFS 的角度望上层看，HBase 即是它的客户端，HBase 通过调用它的客户端进行数据读写操作，因此 HDFS 的相关优化也会影响 HBase 的读写性能。这里主要关注如下三个方面：

4.1 Short-Circuit Local Read 功能是否开启？

优化原理：当前 HDFS 读取数据都需要经过 DataNode，客户端会向 DataNode 发送读取数据的请求，DataNode 接受到请求之后从硬盘中将文件读出来，再通过 TPC 发送给客户端。Short Circuit 策略允许客户端绕过 DataNode 直接读取本地数据。（具体原理参考引用[1]）

优化建议：开启 Short Circuit Local Read 功能，具体配置参考引用[2]

4.2 Hedged Read 功能是否开启？

优化原理：HBase 数据在 HDFS 中一般都会存储三份，而且优先会通过 Short-Circuit Local Read 功能尝试本地读。但是在某些特殊情况下，有可能会出现因为磁盘问题或者网络问题引起的短时间本地读取失败，为了应对这类问题，社区开发者提出了补偿重试机制 –Hedged Read。该机制基本工作原理为：客户端发起一个本地读，一旦一段时间之后还没有返回，客户端将会向其他 DataNode 发送相同数据的请求。哪一个请求先返回，另一个就会被丢弃。

优化建议：开启 Hedged Read 功能，具体配置参考引用[3]

4.3 数据本地率是否太低？

数据本地率：HDFS 数据通常存储三份，假如当前 RegionA 处于 Node1 上，数据 a 写入的时候三副本为(Node1,Node2,Node3)，数据 b 写入三副本是(Node1,Node4,Node5)，数据 c 写入三副本(Node1,Node3,Node5)，可以看出来所有数据写入本地 Node1 肯定会写一份，数据都在本地可以读到，因此数据本地率是 100%。现在假设 RegionA 被迁移到了 Node2 上，只有数据 a 在该节点

上，其他数据（b 和 c）读取只能远程跨节点读，本地率就为 33%（假设 a，b 和 c 的数据大小相同）。

优化原理：数据本地率太低很显然会产生大量的跨网络 IO 请求，必然会导致读请求延迟较高，因此提高数据本地率可以有效优化随机读性能。数据本地率低的原因一般是因为 Region 迁移（自动 balance 开启、RegionServer 宕机迁移、手动迁移等），因此一方面可以通过避免 Region 无故迁移来保持数据本地率，另一方面如果数据本地率很低，也可以通过执行 major_compact 提升数据本地率到 100%。

优化建议：避免 Region 无故迁移，比如关闭自动 balance、RS 宕机及时拉起并迁回飘走的 Region 等；在业务低峰期执行 major_compact 提升数据本地率

5. HBase 读性能优化归纳

在本文开始的时候提到读延迟较大无非三种常见的表象，单个业务慢、集群随机读慢以及某个业务随机读之后其他业务受到影响导致随机读延迟很大。了解完常见的可能导致读延迟较大的一些问题之后，我们将这些问题进行如下归类，读者可以在看到现象之后在对应的问题列表中进行具体定位：

现象：某个业务慢，集群并不慢	
客户端问题	Scan缓存是否设置合理？
	Get请求是否设置为批量？
列族设计问题	Bloomfilter是否设置？是否合理？
	TTL是否设置合理？
HDFS相关问题	本地率是否太低？
现象：集群随机读慢	
HBase服务器端优化	BlockCache配置是否合理？GC参数是否配置合理？
	HFile数是否太多？
	Compaction是否影响很大？是否太过频繁？
HDFS相关优化	Short-Circuit Local Read功能是否开启？
	Hedged Read功能是否开启？（Hadoop 2.4+）
现象：某个业务起来之后其他业务很慢	
客户端问题	大scan是否设置setBlockCache=false？
服务器端问题	读请求是否均衡？

结束语

性能优化是任何一个系统都会遇到的话题，每个系统也都有自己的优化方式。HBase 作为分布式 KV 数据库，优化点又格外不同，更多得融入了分布式特性以及存储系统优化特性。文中总结了读优化的基本突破点，有什么不对的地方还望指正，有补充的也可以一起探讨交流！

深入解读 HBase2.0 新功能之 AssignmentManagerV2

杨文龙 阿里巴巴 技术专家 HBase Committer&HBase PMC

1.背景

AssignmentManager 模块是 HBase 中一个非常重要的模块，Assignment Manager（之后简称 AM）负责了 HBase 中所有 region 的 Assign，UnAssign，以及 split/merge 过程中 region 状态变化的管理等等。在 HBase-0.90 之前，AM 的状态全部存在内存中，自从 HBASE-2485 之后，AM 把状态持久化到了 Zookeeper 上。在此基础上，社区对 AM 又修复了大量的 bug 和优化（见此文章），最终形成了用在 HBase-1.x 版本上的这个 AM。

2.老 Assignment Mananger 的问题

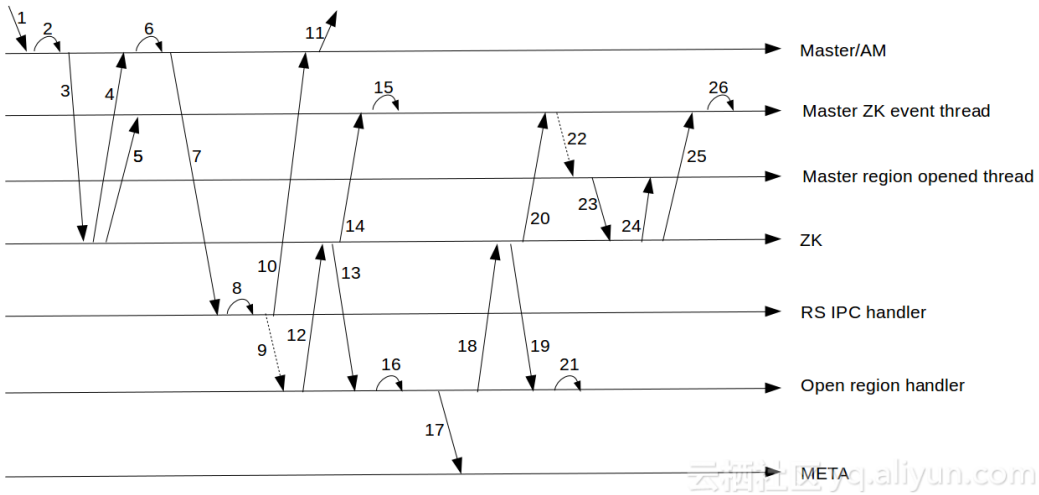
相信深度使用过 HBase 的人一般都会被 Region RIT 的状态困扰过，长时间的 region in transition 状态简直令人抓狂。

Regions in Transition

Region	State	RIT time (ms)
f178ca4fa4f99ae86c478bfb292be6d6	prod.test, state=PENDING_CLOSE, ts=Mon Feb 06 10:37:29 CST 2017 (615309s ago), server= 60020,1485888713418	615309428
b4cc0ee4521564fe54529002483f2879	,1480474973996.b4cc0ee4521564fe54529002483f2879. state=FAILED_CLOSE, ts=Fri Feb 10 23:43:19 CST 2017 (222559s ago), server= 60020,1486059118319	222559973
Total number of Regions in Transition for more than 60000 milliseconds	2	
Total number of Regions in Transition	2	

除了一些确实是由于 Region 无法被 RegionServer open 的 case，大部分的 RIT，都是 AM 本身的问题引起的。总结一下 HBase-1.x 版本中 AM 的问题，主要有以下几点：

3.region 状态变化复杂



这张图很好地展示了 region 在 open 过程中参与的组件和状态变化。可以看到，多达 7 个组件会参与 region 状态的变化。并且在 region open 的过程中多达 20 多个步骤！越复杂的逻辑意味着越容易出 bug

4.region 状态多处缓存

region 的状态会缓存在多个地方，Master 中 RegionStates 会保存 Region 的状态，Meta 表中会保存 region 的状态，Zookeeper 上也会保存 region 的状态，要保持这三者完全同步是一件很困难的事情。同时，Master 和 RegionServer 都会修改 Meta 表的状态和 Zookeeper 的状态，非常容易导致状态的混乱。如果出现不一致，到底以哪里的状态为准？每一个 region 的 transition 流程都是各自为政，各自有各自的处理方法

5.重度依赖 Zookeeper

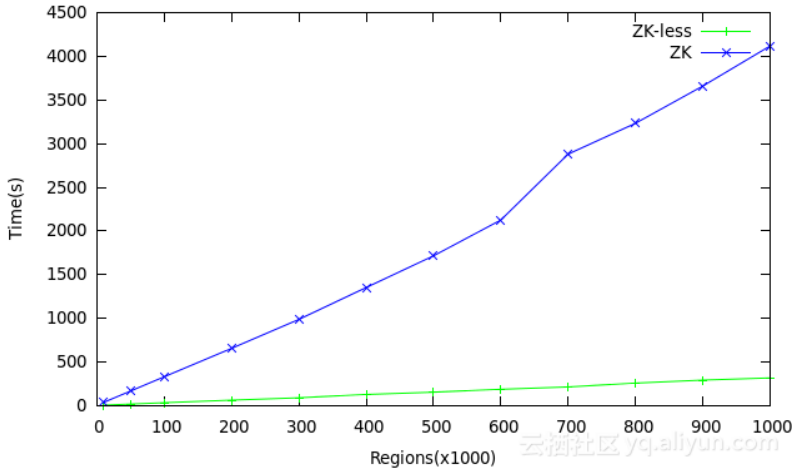
在老的 AM 中，region 状态的通知完全通过 Zookeeper。比如说 RegionServer 打开了一个 region，它会在 Zookeeper 把这个 region 的 RIT 节点改成 OPEN 状态，而不去直接通知 Master。Master 会在 Zookeeper 上 watch 这个 RIT 节点，通过 Zookeeper 的通知机制来通知 Master 这个 region 已经发生变化。Master 再根据 Zookeeper 上读取出来的新状态进行一定的操作。严重依赖 Zookeeper 的通知机

制导致了 region 的上线/下线的速度存在了一定的瓶颈。特别是在 region 比较多的时候，Zookeeper 的通知会出现严重的滞后现象。

正是这些问题的存在，导致 AM 的问题频发。我本人就 fix 过多个 AM 导致 region 无法 open 的 issue。比如说这三个相互关联的“连环”case :HBASE-17264,HBASE-17265,HBASE-17275。

6.Assignment Mananger V2

面对这些问题的存在，社区也在不断尝试解决这些问题，特别是当 region 的规模达到 100w 级别的时候，AM 成为了一个严重的瓶颈。HBASE-11059 中提出的 ZK-less Region Assignment 就是一个非常好的改良设计。在这个设计中，AM 完全摆脱了 Zookeeper 的限制，在测试中，zk-less 的 assign 比 zk 的 assign 快了一个数量级！

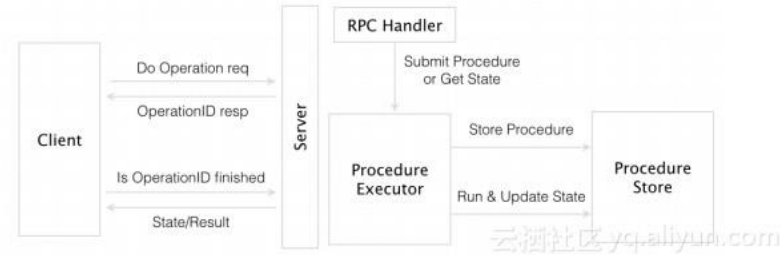


但是在这个设计中，它摒弃了 Zookeeper 这个持久化的存储，一些 region transition 过程中的中间状态无法被保存。因此，在此基础上，社区又更进了一步，提出了 Assignment Mananger V2 在这个方案。在这个方案中，仍然摒弃了 Zookeeper 参与 Assignment 的整个过程。但是，它引入了 ProcedureV2 这个持久化存储来保存 Region transition 中的各个状态，保证在 master 重启时，之前的 assing/unassign，split 等任务能够从中断点重新执行。具体的来说，AMv2 方案中，主要的改进有以下几点：

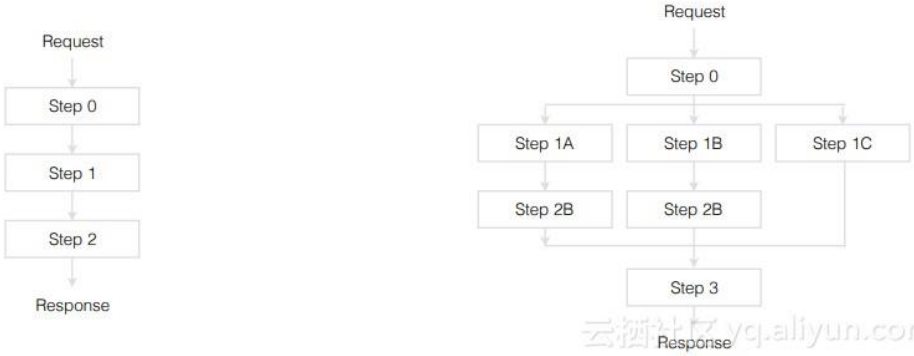
7.Procedure V2

关于 Procedure V2，我之后将独立写文章介绍。这里，我只大概介绍下 ProcedureV2 和引入它所带来的价值。

我们知道，Master 中会有许多复杂的管理工作，比如说建表，region 的 transition。这些工作往往涉及到非常多的步骤，如果 master 在做中间某个步骤的时候宕机了，这个任务就会永远停留在了中间状态（RIT 因为之前有 Zookeeper 做持久化因此会继续从某个状态开始执行）。比如说在 enable/disable table 时，如果 master 宕机了，可能表就停留在了 enabling/disabling 状态。需要一些外部的手段进行恢复。那么从本质上来说，ProcedureV2 提供了一个持久化的手段（通过 ProcedureWAL，一种类似 RegionServer 中 WAL 的日志持久化到 HDFS 上），使 master 在宕机后能够继续之前未完成的任务继续完成。同时，ProcedureV2 提供了非常丰富的状态转换并支持回滚执行，即使执行到某一个步骤出错，master 也可以按照用户的逻辑对之前的步骤进行回滚。比如建表到某一个步骤失败了，而之前已经在 HDFS 中创建了一些新 region 的文件夹，那么 ProcedureV2 在 rollback 的时候，可以把这些残留删除掉。



Procedure 中提供了两种 Procedure 框架，顺序执行和状态机，同时支持在执行过程中插入 subProcedure，从而能够支持非常丰富的执行流程。在 AMv2 中，所有的 Assign，UnAssign，TableCreate 等等流程，都是基于 Procedure 实现的。



8.去除 Zookeeper 依赖

有了 Procedure V2 之后，所有的状态都可以持久化在 Procedure 中，Procedure 中每次的状态变化，都能够持久化到 ProcedureWAL 中，因此数据不会丢失，宕机后也能恢复。同时，AMv2 中 region 的状态扭转（OPENING, OPEN, CLOSING, CLOSE 等）都会由 Master 记录在 Meta 表中，不需要 Zookeeper 做持久化。再者，之前的 AM 使用的 Zookeeper watch 机制通知 master region 状态的改变，而现在每当 RegionServer Open 或者 close 一个 region 后，都会直接发送 RPC 给 master 汇报，因此也不需要 Zookeeper 来做状态的通知。综合以上原因，Zookeeper 已经在 AMv2 中没有了存在的必要。

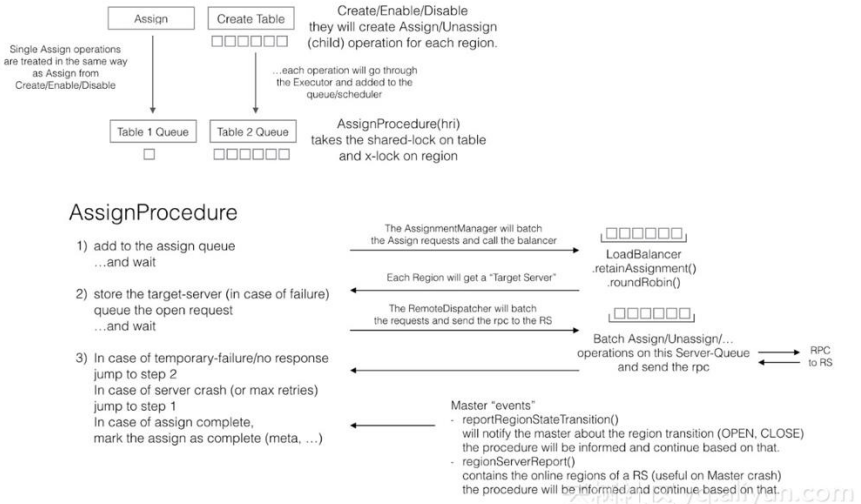
9.减少状态冲突的可能性

之前我说过，在之前的 AM 中，region 的状态会同时存在于 meta 表，Zookeeper 和 master 的内存状态。同时 Master 和 regionserver 都会去修改 Zookeeper 和 meta 表，维护状态统一的代价非常高，非常容易出 bug。而在 AMv2 中，只有 master 才能去修改 meta 表。并在 region 整个 transition 中做为一个“权威”存在，如果 regionserver 汇报上来的 region 状态与 master 看到的不一致，则 master 会命令 RegionServer abort。Region 的状态，都以 master 内存中保存的 RegionStates 为准。

除了上述这些优化，AMv2 中还有许多其他的优化。比如说 AMv2 依赖 Procedure V2 提供的一套 locking 机制，保证了对于一个实体，如一张表，一个 region 或者一个 RegionServer 同一时刻只有一个 Procedure 在执行。同时，在需要往 RegionServer 发送命令，如发送 open，close 等命令时，AMv2 实现了一个 RemoteProcedureDispatcher 来对这些请求做 batch，批量把对应服务器的指令一起发送等等。在代码结构上，之前处理相应 region 状态的代码散落在 AssignmentManager 这个类的各个地方，而在 AMv2 中，每个对应的操作，都有对应的 Procedure 实现，如 AssignProcedure，DisableTableProcedure，SplitTableRegionProcedure 等等。这样下来，使 AssignmentManager 这个之前杂乱的类变的清晰简单，代码量从之前的 4000 多行减到了 2000 行左右。

10.AssignProcedure

AMv2 中有太多的 Procedure 对应各种不同的 transition，这里不去详细介绍每个 Procedure 的操作。我将以 AssignProcedure 为例，讲解一下在 AMv2 中，一个 region 是怎么 assign 给一个 RegionServer，并在对应的 RS 上 Open 的。



AssignProcedure 是一个基于 Procedure 实现的状态机。它拥有 3 个状态：

1. REGION_TRANSITION_QUEUE： Assign 开始时的状态。在这个状态时，Procedure 会对 region 状态做一些改变和存储，并丢到 AssignmentManager 的 assign queue 中。对于单独 region 的 assign，AssignmentManager 会把他们 group 起来，再通过 LoadBalancer 分配相应的服务器。当这一步骤完成后，Procedure 会把自己标为 REGION_TRANSITION_DISPATCH，然后看是否已经分配服务器，如果还没有被分配服务器的话，则会停止继续执行，等待被唤醒。
2. REGION_TRANSITION_DISPATCH：当 AssignmentManager 为这个 region 分配好服务器时，Procedure 就会被唤醒。或者 Procedure 在执行完 REGION_TRANSITION_QUEUE 状态时 master 宕机，Procedure 被恢复后，也会进入此步骤执行。所以在此步骤下，Procedure 会先检查一下是否分配好了服务器，如果没有，则把状态转移回 REGION_TRANSITION_QUEUE，否则的话，则把这个 region 交给 RemoteProcedureDispatcher，发送 RPC 给对应的 RegionServer 来 open 这个 region。同样的，RemoteProcedureDispatcher 也会对相应的指令做一个 batch，批量把一批 region open 的命令发送给某一台服务器。当命令发送完成之后，Procedure 又会进入休眠状态，等待 RegionServer 成功 OPen 这个 region 后，唤醒这个 Procedure

3. REGION_TRANSITION_FINISH: 当有 RegionServer 汇报了此 region 被打开后，会把 Procedure 的状态置为此状态，并唤醒 Procedure 执行。此时，AssignProcedure 会做一些状态改变的工作，并修改 meta 表，把 meta 表中这个 region 的位置指向对应的 RegionServer。至此，region assign 的工作全部完成。

AMv2 中提供了一个 Web 页面（Master 页面中的'Procedures&Locks'链接）来展示当前正在执行的 Procedure 和持有的锁。其实通过 log，我们也可以看到 Assign 的整个过程。假设，一台 server 宕机，此时 master 会产生一个 ServerCrashProcedure 来处理，在这个 Procedure 中，会做一系列的工作，比如 WAL 的 restore。当这些前置的工作做完后，就会开始 assign 之前在宕掉服务器上的 region，比如 56f985a727afe80a184dac75fbf6860c。此时会在 ServerCrashProcedure 产生一系列的子任务：

```
2017-05-23 12:04:24,175 INFO  [ProcExecWrkr-30] procedure2.ProcedureExecuto
r: Initialized subprocedures=[{pid=1178, ppid=1176, state=RUNNABLE:REGION_T
RANSITION_QUEUE; AssignProcedure table=IntegrationTestBigLinkedList, region
=bfd57f0b72fd3ca77e9d3c5e3ae48d76, target=ve0540.halxg.cloudera.com,16020,1
495525111232}, {pid=1179, ppid=1176, state=RUNNABLE:REGION_TRANSITION_QUEU
E; AssignProcedure table=IntegrationTestBigLinkedList, region=56f985a727afe
80a184dac75fbf6860c, target=ve0540.halxg.cloudera.com,16020,1495525111232}]
```

可以看到，ServerCrashProcedure 的 pid（Procedure ID）为 1178，在此 Procedure 中产生的 assign 56f985a727afe80a184dac75fbf6860c 这个 region 的子 Procedure 的 pid 为 1179，同时他的 ppid（Parent Procedure ID）为 1178。在 AMv2 中，通过追踪这些 ID，就非常容易把一个 region 的 transition 整个过程全部串起来。接下来，pid=1170 这个 Procedure 开始执行，首先执行的是 REGION_TRANSITION_QUEUE 状态的逻辑，然后进入睡眠状态。

```
2017-05-23 12:04:24,241 INFO  [ProcExecWrkr-30] assignment.AssignProcedu
re: Start pid=1179, ppid=1176, state=RUNNABLE:REGION_TRANSITION_QUEUE; A
ssignProcedure table=IntegrationTestBigLinkedList, region=56f985a727afe8
0a184dac75fbf6860c, target=ve0540.halxg.cloudera.com,16020,149552511123
2; rit=OFFLINE, location=ve0540.halxg.cloudera.com,16020,1495525111232;
forceNewPlan=false, retain=false
```

当 target server 被指定时，Procedure 进入 REGION_TRANSITION_DISPATCH 状态，dispatch 了 region open 的请求，同时把 meta 表中 region 的状态改成了 OPENING，然后再次进入休眠状态

```
2017-05-23 12:04:24,494 INFO  [ProcExecWrkr-38] assignment.RegionStateSt
ore: pid=1179 updating hbase:meta row=IntegrationTestBigLinkedList,H\xE3
@\x8D\x964\x9D\xDF\x8F@9'\x0F\xC8\xCC\xC2,1495566261066.56f985a727afe80
a184dac75fbf6860c., regionState=OPENING, regionLocation=ve0540.halxg.clo
udera.com,16020,1495525111232
 2017-05-23 12:04:24,498 INFO  [ProcExecWrkr-38] assignment.RegionTransi
tionProcedure: Dispatch pid=1179, ppid=1176, state=RUNNABLE:REGION_TRANS
ITION_DISPATCH; AssignProcedure table=IntegrationTestBigLinkedList, regi
on=56f985a727afe80a184dac75fbf6860c, target=ve0540.halxg.cloudera.com,16
020,1495525111232; rit=OPENING, location=ve0540.halxg.cloudera.com,1602
0,1495525111232
```

最后，当 RegionServer 打开了这个 region 后，会发 RPC 通知 master，那么在通知过程中，这个 Procedure 再次被唤醒，开始执行 REGION_TRANSITION_FINISH 的逻辑，最后更新 meta 表，把这个 region 置为打开状态。

```
2017-05-23 12:04:26,643 DEBUG
[RpcServer.default.FPBQ.Fifo.handler=46,queue=1,port=16000]
assignment.RegionTransitionProcedure: Received report OPENED
seqId=11984985, pid=1179, ppid=1176,
state=RUNNABLE:REGION_TRANSITION_DISPATCH; AssignProcedure
table=IntegrationTestBigLinkedList,
region=56f985a727afe80a184dac75fbf6860c,
target=ve0540.halxg.cloudera.com,16020,1495525111232; rit=OPENING,
location=ve0540.halxg.cloudera.com,16020,1495525111232
2017-05-23 12:04:26,643 INFO  [ProcExecWrkr-9]
assignment.RegionStateStore: pid=1179 updating hbase:meta
row=IntegrationTestBigLinkedList,H\xE3@\x8D\x964\x9D\xDF\x8F@9'\x0F\xC8
\xCC\xC2,1495566261066.56f985a727afe80a184dac75fbf6860c.,
regionState=OPEN, openSeqNum=11984985,
regionLocation=ve0540.halxg.cloudera.com,16020,1495525111232
2017-05-23 12:04:26,836 INFO  [ProcExecWrkr-9]
procedure2.ProcedureExecutor: Finish suprocedure pid=1179, ppid=1176,
state=SUCCESS; AssignProcedure table=IntegrationTestBigLinkedList,
region=56f985a727afe80a184dac75fbf6860c,
target=ve0540.halxg.cloudera.com,16020,1495525111232
```

一路看下来，由于整个 region assign 的过程都是在 Procedure 中执行，整个过程清晰明了，非常容易追述，也没有了 Zookeeper 一些 event 事件的干扰。

11.总结

Assignment Mananger V2 依赖 Procedure V2 实现了一套清晰明了的 region transition 机制。去除了 Zookeeper 依赖，减少了 region 状态冲突的可能性。整

体上来看，代码的可读性更强，出了问题也更好查错。对于解决之前 AM 中的一系列“顽疾”，AMv2 做了很好的尝试，也是一个非常好的方向。

AMv2 之所以能保持简洁高效的一个重要原因就是重度依赖了 Procedure V2，把一些复杂的逻辑都转移到了 Procedure V2 中。但是这样做的问题是：一旦 ProcedureWAL 出现了损坏，或者 Procedure 本身存在 bug，这个后果就是灾难性的。事实上在我们的测试环境中，就出现过 PRocedureWAL 损坏导致 region RIT 的情况。

另外需要注意的是，截止目前为止，HBCK 仍然无法支持 AMv2，这会导致一旦出现问题，修复起来会比较困难。当然，新的事务还是要有一段成熟期，相信经过一段时间的 bug 修复和完善后，我相信 AMv2 一定会完美解决之前的一些问题，给 HBase 的运维上带来一些不同的体验。愿世界不再被 HBase 的 RIT 困扰 :-)。

深入解读 HBase2.0 新功能之高可用读 Region Replica

杨文龙 阿里巴巴 技术专家 HBase Committer&HBase PMC

1.前言

基于时间线一致的高可用读（Timeline-consistent High Available Reads），又称 Region replica。其实早在 HBase-1.2 版本的时候，这个功能就已经开发完毕了，但是还是不太稳定，离生产可用级别还有一段距离，后来社区又陆陆续续修复了一些 bug，比如说 HBASE-18223。这些 bug 很多在 HBase-1.4 之后的版本才修复，也就是说 region replica 功能基本上在 HBase-1.4 之后才稳定下来。介于 HBase-1.4 版本目前实际生产中使用的还比较少，把 region replica 功能说成是 HBase2.0 中的新功能也不为过。

2.为什么需要 Region Replica

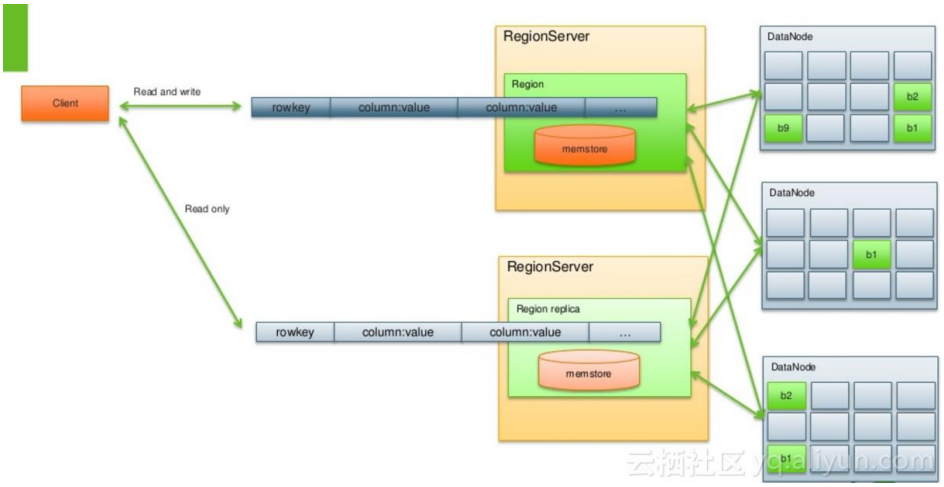
在 CAP 理论中，HBase 一直是一个 CP（Consistency&Partition tolerance）系统。HBase 一直以来都在遵循着读写强一致的语义。所以说虽然在存储层，HBase 依赖 HDFS 实现了数据的多副本，但是在计算层，HBase 的 region 只能在一台 RegionServer 上线提供读写服务，来保持强一致。如果这台服务器发生宕机时，Region 需要从 WAL 中恢复还缓存在 memstore 中未刷写成文件的数据，才能重新上线服务。

由于 HBase 的 RegionServer 是使用 Zookeeper 与 Master 保持 lease。而为了不让 JVM GC 停顿导致 RegionServer 被 master“误判”死亡，这个 lease 时间通常都会设置为 20~30s，如果 RegionServer 使用的 Heap 比较大时，这个 lease 可能还会设的更长。加上宕机后，region 需要 re-assign，WAL 可能需要 recoverlease 和被 replay 操作，一个典型的 region 宕机恢复时间可能长达一分钟！这就意味着在这一分钟内，这个 region 都无法被读写。由于 HBase 是一个分布式系统，同一张表的数据可能分布在非常多的 RegionServer 和 region 里。如果这是一个大 HBase 集群，有 100 台 RegionServer 机器，那么宕机一台的话，可能只有 1% 的用户数据被影响了。但是如果这是小用户的 HBase 集群，一共就只有 2 台 RegionServer，宕机一台意味着 50%的用户数据都在 1~2 分钟之内无法服务，这是很多用户都无法忍受的。

其实，很大一部分用户对读可用性的需求，可能比读强一致的需求还要高。在故障场景下，只要保证读继续可用，“stale read”，即读到之前的数据也可以接受。这就是为什么我们需要 read replica 这个功能。

3.Region Replica 技术细节

Region replica 的本质，就是让同一个 region host 在多个 regionserver 上。原来的 region，称为 Default Replica（主 region），提供了与之前类似的强一致读写体验。而与此同时，根据配置的多少，会有一个或者多个 region 的副本，统称为 region replica,在另外的 RegionServer 上被打开。并且由 Master 中的 LoadBalancer 来保证 region 和他们的副本，不会在同一个 RegionServer 打开，防止一台服务器的宕机导致多个副本同时挂掉。



Region Replica 的设计巧妙之处在于，额外的 region 副本并不意味着数据又会多出几个副本。这些 region replica 在 RegionServer 上 open 时，使用的是和主 region 相同的 HDFS 目录。也就是说主 region 里有多少 HFile，那么在 region replica 中，这些数据都是可见的，都是可以读出来的。region replica 相对于主 region，有一些明显的不同。

首先，region replica 是不可写的。这其实很容易理解，如果 region replica 也可以写的话，那么同一个 region 会在多个 regionserver 上被写入，连主 region 上的强一致读写都没法保证了。

再次，region replica 是不能被 split 和 merge 的。region replica 是主 region 的附属品，任何发向 region replica 的 split 和 merge 请求都会被拒绝掉。只有当主 region split/merge 时，才会把这些 region replica 从 meta 表中删掉，建立新生成 region 的 region 的 replica。

4.replica 之间的数据同步

那么，既然 region replica 不能接受写，它打开之后，怎么让新写入的数据变的可见呢？这里，region replica 有两种更新数据的方案：

4.1. 定期的 StoreFile Refresher

这个方案非常好理解，region replica 定期检查一下它自己对应的 HDFS 目录，如果发现文件有变动，比如说 flush 下来新的文件，文件被 compaction 掉，它就刷新一下自己的文件列表，这个过程非常像 compaction 完成之后删除被 compact 掉的文件和加入新的文件的流程。StoreFile Refresher 方案非常简单，只需要在 RegionServer 中起一个定时执行的 Chroe，定期去检查一下它上面的 region 哪些是 region replica，哪些到了设置好的刷新周期，然后刷新就可以了。但这个方案缺点也十分明显，主 region 写入的数据，只有当 flush 下来后，才能被 region replica 看到。而且 storeFile Refresher 本身还有一个刷新的周期，设的太短了，list 文件列表对 NN 的冲击太频繁，设的太长，就会造成数据长时间在 region replica 中都不可见

4.2. Internal Replication

我们知道，HBase 是有 replication 链路的，支持把一个 HBase 集群的数据通过 replication 复制到另外一个集群。那么，同样的原理，可以在 HBase 集群内部建立一条 replication 通道，把一个 Server 上的主 region 的数据，复制到另一个 Server 的 region replica 上。那么 region replica 接收到这些数据之后，会把他们写入 memstore 中。对，你没看错，刚才我说了 region replica 是不接受写的，这是指 replica 不接受来自客户端的写，如果来自主 region 的 replication 的数据，它还是会写入 memstore 的。但是，这个写和普通的写有很明显的区别。第一个，replica region 在写入来自主 region 的时候，是不写 WAL 的，因为这些数据已经在主 region 所在的 WAL 中持久化了，replica 中无需再次落盘。第二个，replica region 的 memstore 中的数据是不会被 flush 成 HFile。我们知道，HBase 的 replication 是基于复制 WAL 文件实现的，那么在主 region 进行 flush 时，也会写入特殊的标记 Flush Marker。当 region replica 收到这样的标记时，就直接会把所有 memstore 里的数据丢掉，再做一次 HDFS 目录的刷新，把主 region 刚刚刷下去的那个 HFile include 进来。同样，如果主 region 发生了 compaction，也会写入相应的 Compaction Marker。读到这样的标记后，replica region 也会做类似的动作。

Internal replication 加快了数据在 region replica 中的可见速度。通过 replication 方案，只要 replication 本身不发生阻塞和延迟，region replica 中的数据可以做到和主 region 只差几百 ms。但是，replication 方案本身也存在几个问题：

1. META 表无法通过 replication 来同步数据。如果给 meta 表开了 region replica 功能，meta 表主 region 和 replica 之间的数据同步，只能通过定期的 StoreFile Refresher 机制。因为 HBase 的 replication 机制中会过滤掉 meta 表的数据。
2. 需要消耗额外的 CPU 和网络带宽来做 Replication。由于 region replica 的数据同步需要，需要在 HBase 集群内部建立 replication 通道，而且有几个 replica，就意味着需要从主 region 发送几份数据。这会增加 RegionServer 的 CPU 使用，同时在 server 之间复制数据还需要占用带宽
3. 写 memstore 需要额外的内存开销。为了让 replica region 的数据缺失的内容尽量的少，主 region 的数据会通过 replication 发送到 replica 中，这些数据都会保存在 memstore 中。也就是说同样的一份数据，会同时存在主

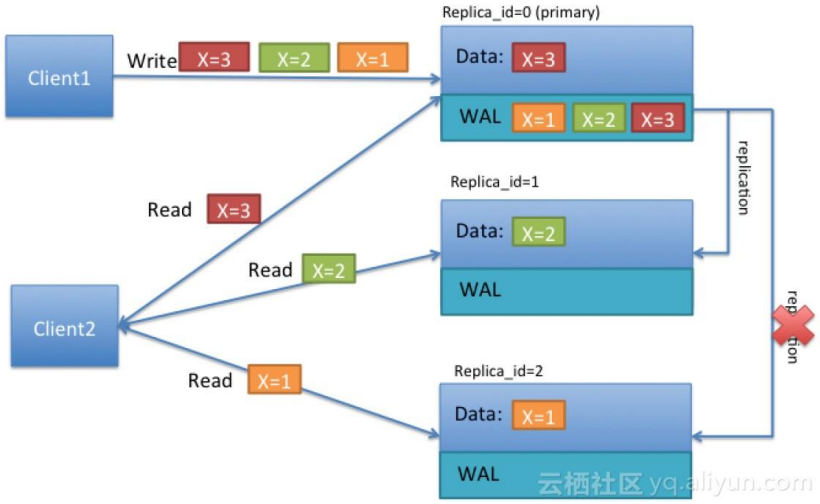
region 的 memstore 中，也会存在 replica region 的 memstore 中。replica 的数量是几，那么 memstore 的内存使用量就是几倍。

下面的两个问题虽然可以通过配置一些参数解决，但是列在这里，仍然需要注意，因为一旦参数没有配对，就会产生这样的问题。

1. 在 replica region failover 后，读到的数据可能会回退。我们假设一个情况。客户端写入 X=1，主 region 发生 flush，X=1 刷在了 HFile 中，然后客户端继续写入 X=2，X=3，那么在主 region 的 memstore 中 X=3。同时，通过 replication，X=2，X=3 也被复制到了 replica region 的 memstore 中。如果客户端去 replica 中去读取 X 的数据，也能读到 3。但是由于 replica region memstore 中的数据是不写 WAL 的，也不刷盘。那么当 replica 所在的机器宕机后，它是没有任何数据恢复流程的，他会直接在其他 RegionServer 上线。上线后它只能读取 HFile，无法感知主 region memstore 里的数据。这时如果客户端来 replica 上读取数据，那么他只会读到 HFile 中的 X=1。也就是说之前客户端可以读到 X=3，但后来却只能读到 X=1 了，数据出现了回退。为了避免出现这样的问题，可以配置一个 hbase.region.replica.wait.for.primary.flush=true 的参数，配置之后，replica region 上线后，会被标记为不可读，同时它会去触发一次主 region 的 flush 操作。只有收到主 region 的 flush marker 之后，replica 才把自己标记为可读，防止读回退
2. replica memstore 过大导致写阻塞。上面说过，replica 的 region 中 memstore 是不会主动 flush 的，只有收到主 region 的 flush 操作，才会去 flush。同一台 RegionServer 上可能有一些 region replica 和其他的主 region 同时存在。这些 replica 可能由于复制延迟(没有收到 flush marker)，或者主 region 没有发生 flush，导致一直占用内存不释放。这会造成整体的内存超过水位线，导致正常的写入被阻塞。为了防止这个问题的出现，HBase 中有一个参数叫做 hbase.region.replica.storefile.refresh.memstore.multiplier，默认值是 4。这个参数的意思是说，如果最大的 replica region 的 memstore 已经超过了最大的主 region memstore 的内存的 4 倍，就主动触发一次 StoreFile Refresher 去更新文件列表，如果确实发生了 flush，那么 replica 内存里的数据就能被释放掉。但是，这只是解决了 replication 延迟导致的未 flush 问题，如果这个 replica 的主 region 确实没有 flush 过，内存还是不能被释放。写入阻塞还是会存在。

5.Timeline Consistency Read

无论是 StoreFile Refresher 还是 Internal replication，主 region 和 replica 之间的数据更新都是异步的，这就导致在 replica region 中读取数据时，都不是强一致的。read replica 的作者把从 region replica 中读数据的一致性等级定为 Timeline Consistency。只有用户明确表示能够接受 Timeline consistency，客户端的请求才会发往 replica 中。



比如说上图中，如果客户端是需要强一致读，那么客户端的请求只会发往主 region，即 replica_id=0 的 region，他就会读到 X=3.如果他选择了 Timeline consistency 读，那么根据配置，他的读可能落在主上，那么他仍然会读到 X=3，如果他的读落在了 replica_id=1 的 region 上，因为复制延迟的存在，他就只能读到 X=2.如果落在了 replica_id=2 上，由于 replication 链路出现了问题，他就只能读到 X=1。

6.Region replica 的使用方法

6.1 服务端配置

```
hbase.regionserver.storefile.refresh.period
```

如果要使用 StoreFile Refresher 来做为 Region replica 之间同步数据的策略，就必须把这个值设置为一个大于 0 的数，即刷新 storefile 的间隔周期（单位为 ms）上面的章节讲过，这个值要不能太大，也不能太小。

```
hbase.regionserver.meta.storefile.refresh.period
```

由于 Meta 表的 region replica 不能通过 replication 来同步，所以如果要开启 meta 表的 region replica，必须把这个参数设成一个不为 0 的值，具体作用参见上一个参数，这个参数只对 meta 表生效。

```
hbase.region.replica.replication.enabled
hbase.region.replica.replication.memstore.enabled
```

如果要使用 Internal replication 的方式在 Region replica 之间同步数据的策略，必须把这两个参数都设置为 true

```
hbase.master.hfilecleaner.ttl
```

在主 region 发生 compaction 之后，被 compact 掉的文件会放入 Achieve 文件夹内，超过 hbase.master.hfilecleaner.ttl 时间后，文件就会被从 HDFS 删除掉。而此时，可能 replica region 正在读取这个文件，这会造成用户的读取抛错返回。如果不想要这种情况发生，就可以把这个参数设为一个很大的值，比如说 3600000（一小时），总没有读操作需要读一个小时了吧？

```
hbase.meta.replica.count
```

mata 表的 replica 份数，默认为 1，即不开启 meta 表的 replica。如果想让 meta 表有额外的一个 replica，就可以把这个值设为 2，依次类推。此参数只影响 meta 表的 replica 份数。用户表的 replica 份数是在表级别配置的，这个我后面会讲

```
hbase.region.replica.storefile.refresh.memstore.multiplier
```

这个参数我在上面的章节里有讲，默认为 4

```
hbase.region.replica.wait.for.primary.flush
```

这个参数我在上面的章节里有讲，默认为 true

需要注意的是，开启 region replica 之后，Master 的 balancer 一定要用默认的 StochasticLoadBalancer，只有这个 balancer 会尽量使主 region 和他的 replica 不在同一台机器上。其他的 balaner 会无区别对待所有的 region。

6.2 客户端配置

```
hbase.ipc.client.specificThreadForWriting
```

因为当存在 region replica 时，当客户端发往主 region 的请求超时后，会发起一个请求到 replica region，当其中一个请求放回后，就无需再等待另一个请求的结果了，通常要中断这个请求，使用专门的的线程来发送请求，比较容易处理中断。所以如果要使用 region replica，这个参数要配为 true。

```
hbase.client.primaryCallTimeout.get
hbase.client.primaryCallTimeout.multiget
hbase.client.replicaCallTimeout.scan
```

分别对应着，get、multiget、scan 时等待主 region 返回结果的时间。如果把这个值设为 1000ms，那么客户端的请求在发往主 region 超过 1000ms 还没返回后，就会再发一个请求到 replica region（如果有多个 replica 的话，就会同时发往多个 replica）

```
hbase.meta.replicas.use
```

如果服务端上开启了 meta 表的 replica 后，客户端可以使用这个参数来控制是否使用 meta 表的 replica 的 region。

6.3 建表

在 shell 建表时，只需在表的属性里加上 REGION_REPLICATION => xx 就可以了，如

```
create 't1', 'f1', {REGION_REPLICATION => 2}
```

Replica 的份数支持动态修改，但修改之前必须 disable 表

```
diable 't1'
alter 't1', {REGION_REPLICATION => 1}
enable 't1'
```

6. 4 访问有 replica 的表

如果可以按请求设置一致性级别，如果把请求的一致性级别设为 Consistency.TIMELINE，即有可能读到 replica 上

```
Get get1 = new Get(row);
get1.setConsistency(Consistency.TIMELINE);
...
ArrayList<Get> gets = new ArrayList<Get>();
gets.add(get1);
...
Result[] results = table.get(gets);
```

另外，用户可以通过 Result.isStale()方法来获得返回的 result 是否来自主 region，如果为 isStale 为 false，则结果来自主 region。

```
Result result = table.get(get);
if (result.isStale()) {
...
}
```

7.总结和建议

Region Replica 功能给 HBase 用户带来了高可用的读能力，提高了 HBase 的可用性，但同时也存在一定的缺点：

1. 高可用的读基于 Timeline consistency，用户需要接受非强一致性读才能开启这个功能
2. 使用 Replication 来做数据同步意味着额外的 CPU，带宽消耗，同时根据 replica 的多少，可能会有数倍的 memstore 内存消耗
3. 读取 replica region 中的 block 同样会进 block cache（如果表开启了 block cache 的话），这意味着数倍的 cache 开销
4. 客户端 Timeline consistency 读可能会把请求发往多个 replica，可能带来更多的网络开销

Region Replica 只带来了高可用的读，宕机情况下的写，仍然取决于主 region 的恢复时间，因此 MTTR 时间并没有随着使用 Region replica 而改善。虽然说 region replica 的作者在规划中有写计划在宕机时把一个 replica 提升为主，来优化 MTTR 时间，但截至目前为止，还没有实现。

个人建议，region replica 功能适合于用户集群规模较小，对读可用性非常在意，同时又可以接受非强一致性读的情况下开启。如果集群规模较大，或者读写流量非常大的集群上开启此功能，需要留意内存使用和网络带宽。Memstore 占用内存过高可能会导致 region 频繁刷盘，影响写性能，同时 cache 容量的翻倍会导致一部分读请求击穿 cache 直接落盘，导致读性能的下降。

HBase 2.0 之修复工具 HBCK2 运维指南

田竞云 小米 HBase Committer

概述

目前社区已经发布了 HBase 的 2.0 版本，很多公司都希望去尝试新版本上的新功能，但是不得不面对的问题就是当集群出了问题应该如何解决。在之前的 HBase 版本中，我们可以依赖 hbck 来帮助检查问题和修复问题，在新的版本上我们应该如何去处理呢？HBASE-19121[1]给了我们答案——HBCK2。HBCK2 目前发布了 1.0 版本，还在一直开发中，感兴趣的同学看看这个 issue

1. WHY HBCK2

由于之前的 hbck(hbck-1)是会直接去向 region server 或者 hdfs 发送请求进行修复，而在 HBase 2.0 版本上集群内部操作全部都被挪到了 procedure v2(下文都称为 procedure)上进行处理。

因为所有的命令都是经过 master 来协调处理，所以在修复时也需要通过 master 进行修复。否则反而可能导致更严重的不一致问题。所以 hbck-1 在 HBase-2.x 版本是不适用的。

除此以外，很多 hbck-1 需要处理的问题对于使用 procedure 的 HBase-2.x 已经不再是问题了，所以相比起 hbck-1 来说也精简了很多功能。

2. 开始使用

2.1 下载

社区希望把 HBase 相关的外围工具抽离出 HBase 项目，所以在 github 上建了一个 project hbase-operator-tools: hbase-operator-tools[2]，HBCK2 也在其中

2.2 将项目拉取到本地

```
git clone https://github.com/apache/hbase-operator-tools.git
```

2.3 编译

修改 pom 中依赖的 HBase 版本 ， 实际使用的 HBase 版本.version>

2.4 编译出 jar 包

```
mvn clean package
```

2.5 使用

cd 到 HBase 目录下，执行命令:

```
HBASE_CLASSPATH_PREFIX=
```

编译出的 hbck2_jar 包地址

```
 ./bin/hbase org.apache.hbase.HBCK2 <命令>
```

或者也可以：

```
./bin/hbase hbck -j <jar 包地址> <命令>
```

3. Procedure 简介

由于目前几乎所有的集群操作都是通过 procedure 进行的，所以 HBCK2 的工作实际就是修复各种不正常的 procedure。所以在这里有必要对 procedure 进行一个简单的介绍。

一个 procedure 是由一系列的操作组成的，一旦完成后，要么成功，要么失败(ROLL BACK)，不存在中间状态，所以 procedure 是支持事务的。procedure 执行的每一步都会以 log 的形式持久化在 HBase 的 MasterProcWals 目录下，这样 master 在重启时也能通过日志来恢复之前的状态并且继续执行。对于运维而言最重要的一点就是 procedure 在执行过程中会拿好几把锁，这个在处理问题时是很重要的，因为一旦锁没有释放，再做任何操作也只能是卡住等锁。

1. IdLock：procedure 级别的锁，保证一个 procedure 不会被多个线程同时执行。
2. 资源锁:对 HBase 的内部资源进行加锁,不同的 procedure 加锁的粒度不同，目前有 region/table/namespace/region server 级别的锁。

举例来说，假设我 assign 一个 region,那么 procedure 在执行的时候就需要对这个 region 进行加锁，这样如果有别的人想要 unassign 这个 region,或者 drop 这个 region 所在的 table,都需要等最早的 assignment 结束后释放锁了才能执行。这样防止有不一致的情况出现。

4. 功能介绍

4.1 bypass[OPTIONS]<PID>...

HBCK2 的核心功能，bypass 可以将一个或多个卡住的 procedure 进行释放。
原理很简单，在 procedure 的类里有一个 bypass 的 flag，每次执行时会检查这个 flag 是否为 true，如果为 true 则直接返回 null，这样 procedure 就会被认为执行成功。而我们的 bypass 就是把这个 procedure 对象中的这个 flag 设为 true。 这样 stuck 的 procedure 就能够不再执行,后续的修复工作才能继续。

返回值为 true 则是成功，false 是失败。

参数解析：

```
-o,--overide
```

在执行 bypass 之前先会尝试去拿 ldLock，如果 procedure 还在运行就会超时返回 null，但是设置了这个参数即使拿不到 ldLock 也会去将 procedure 的 bypass flag 设为 true。

```
-r, --recursive
```

在 bypass 一个 procedure 时也会将这个 procedure 的所有子 procedure 进行递归的 bypass。例如我们 bypass 一个对 table schema 修改的 procedure，就需要加上-r 参数，才能把这个操作的所有子 procedure 都 bypass 掉。

```
-w, --lockWait
```

上文提到的等待 ldLock 的超时时间配置，默认为 1ms

4.2 assigns [OPTIONS] <ENCODED_REGIONNAME>

将一个或多个 region 再次随机 assign 到别的机器上，返回值是创建的 pid 则为成功，-1 则为失败。

参数解析：

```
-o,--override
```

这里的 override 跟 bypass 的 override 不同，因为 assign 本身就会创建一个新的 procedure，所以肯定是不涉及到拿 ldLock 的，但是这里涉及到资源锁的问题。因为之前卡住的资源锁即使在 bypass 后也不会释放(用于 fence，防止更多未知的错误操作)，所以需要加一个-o 去手动释放这个资源锁。

4.3 unassigns <ENCODED_REGIONNAME>...

将一个或多个 region unassign，返回值是创建的 pid 则为成功，-1 则为失败。

参数解析：

```
-o,--override，与 assigns 的一致
```

4.4 setTableState<TABLE><STATE>

可能的 table 状态, ENABLED, DISABLED, DISABLING, ENABLING

在 table 的状态和所有的 region 状态不一致时可以用这个命令进行修复。

4.5 serverCrashProcedures<ServerName>

手动 schedule 一个或多个 serverCrashProcedure，如果有 serverCrashProcedure 没有执行成功，但是 procedure log 已经丢失了，那么可以利用这个命令进行修复。返回值为创建的 pid 则为成功，-1 则为失败。

patch 在 HBASE-21393[3]，目前这个功能在 release 版本还没有。

所有的武器我们都有了，再回顾一下之前提到的问题，我们应该怎么处理呢？在 Case 解决中我们会详细阐述应该怎么处理，这里大家可以先思考一下。

5.查找问题

这个章节我们会介绍怎么去发现目前集群可能存在的问题。

5.1 canary tool

模拟用户的读写请求，去访问集群上的表。当我们需要检查集群 meta 上记录的 region assignment 跟实际 region server 上打开的 region 是否一致时，可以使用这个命令去检查：

```
hbase canary -f false -t 6000000
```

这个命令会向 meta 上的记录的每个 region 发送一个 get 请求，将-f 设为 false 是为了不在遇到第一个错误时退出，-t 则是这个命令的超时时间，我们设成了 6000 秒。在执行完成以后可以通过 grep ERROR 来找到那些有问题的 region。

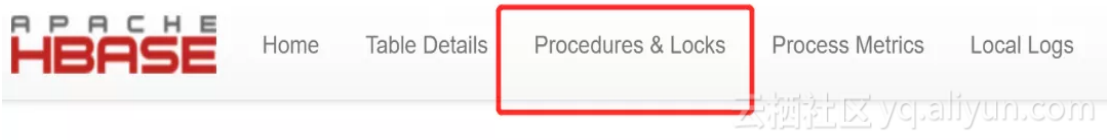
需要注意的是因为是模拟客户端发送的 get 请求，最好将 HBase 的客户端超时时间和超时次数配的小一些，否则会很慢。

PS: canary 本身也很适合用来作为集群可用性的监控，有兴趣的同学可以去了解一下。

5.2 页面状态

其实大部分的信息都会在 master 的页面上展示出来，我们来详细的介绍一下：

Procedures & Locks:



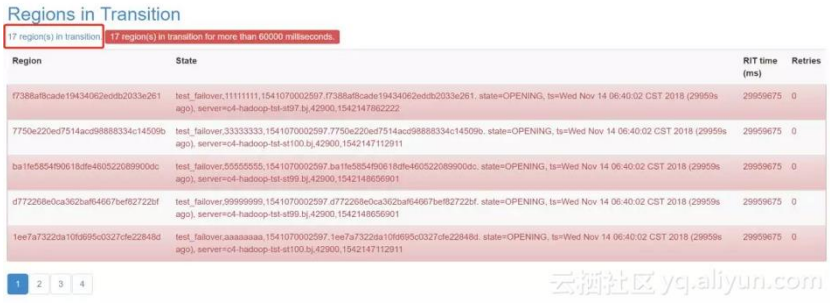
可以检查当前所有没有执行完的 procedure 以及所有资源锁，当我们想要 assign 或者 unassign 一个 region 时，需要先去检查下是由有别的 procedure 已经占有了这个资源锁，如果是的话需要现将那个 procedure bypass 掉，或者等待那个 procedure 释放锁。

以看到 EXCLUSIVE 的 lock 只有 region 级别的，图中红框圈出来的就是占有这个锁的 procedure id 以及它的 parent procedure id，由此我们知道如果想要重新 assign/unassign 这个 region,那么一定要 bypass 这个 procedure。

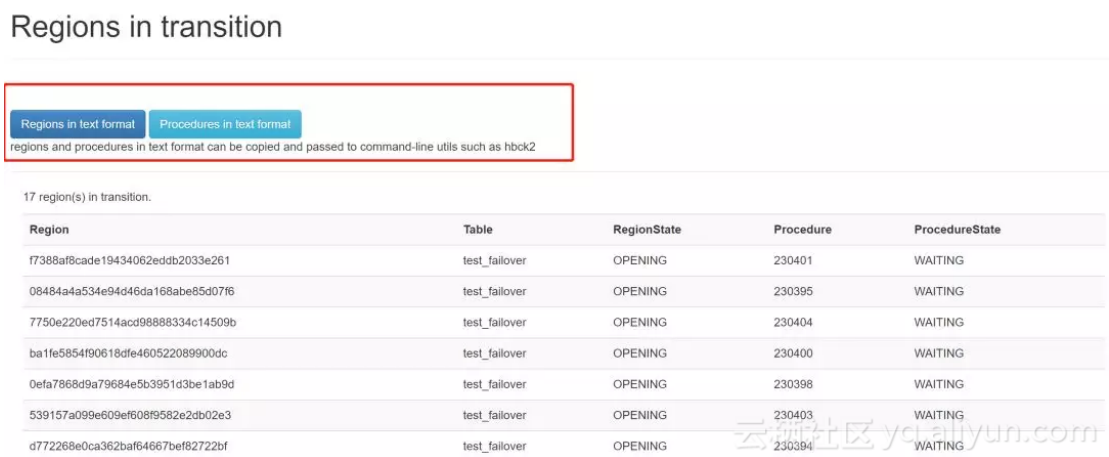
同理，当 Locks 这块没有任何 EXLUSIVE 锁时，我们可以放心的去执行操作而不用担心被卡住。

5.3 OPENING/CLOSING region 的查找

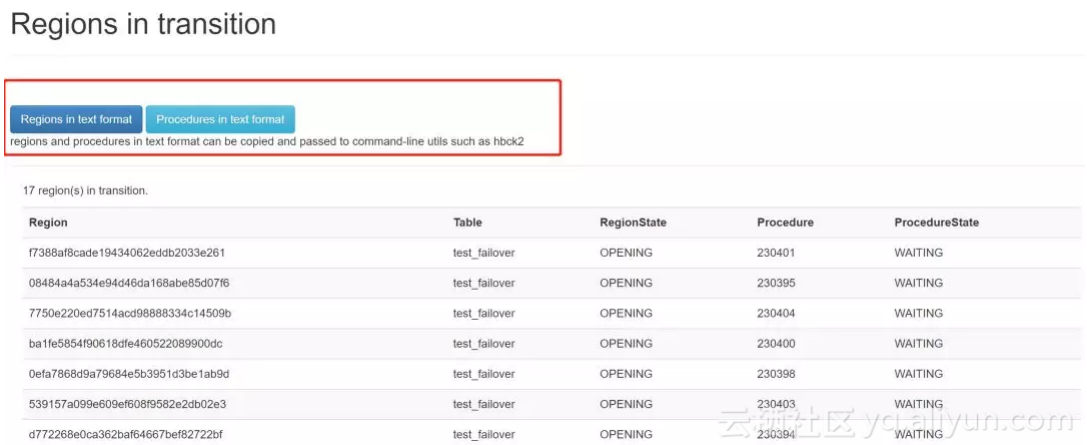
branch-2.0 上最容易出现的问题就是 region 卡在了 OPENNING/CLOSING 状态，一般处于这两种状态的 region 都会在 rit 的队列中，可以通过点击页面上的链接拿到所有的 region 以及对应的 procedure id。



可以看到现在有 17 个 region 处在 transition 中，我们可以点击红框圈住的这个链接，会展示所有的 region。



可以看到现在有 17 个 region 处在 transition 中，我们可以点击红框圈住的这个链接，会展示所有的 region。



因为我们最后是希望通过 HBCK2 来进行处理，那么最好是可以复制粘贴需要处理的 region 或者 procedure，所以可以点击圈出的这两个按钮，会以 text 形式展示所有 region 或者所有 procedure。

```
f7388af8cade19434062eddb2033e261
08484a4a534e94d46da168abe85d07f6
7750e220ed7514acd98888334c14509b
ba1fe5854f90618dfe460522089900dc
0efa7868d9a79684e5b3951d3be1ab9d
539157a099e609ef608f9582e2db02e3
d772268e0ca362baf64667bef82722bf
1ee7a7322da10fd695c0327cfe22848d
8cff0d125d988af874e27514d1c10aed
cc8b17da74126bb12b2d6010711b35d4
9e28c4776415ccf343b736f61f6ea515
e6f00a8bde6b4ce1f40e4bb119af0072
197299af308c8ed772b86458840d9b98
93ef71ffe0b4d704594d504872f31f71
0e68c897f6ac5c078916b7140082910c
8d81f74b324d0503c3fc87f34e9a17cb
6952b58157df8be9b12a03d659d60048
```

5.4 Master 日志

stuck 的 region 会打印以下日志：

```
WARN [ProcExecTimeout] org.apache.hadoop.hbase.master.assignment.Assignment
Manager: STUCK Region-In-Transition rit=OPENING, location=c4-hadoop-tst-st9
9.bj,42900,1542148656901, table=test_modify, region=8d81f74b324d0503c3fc87f
34e9a17cb
```

6.Case 解决

6.1 region 卡在 OPENING/CLOSING 状态

首先找到这些 region 对应的 pid，然后执行 bypass，检查是否锁都释放了，如果释放了就再 assign 一遍，如果需要 close，就再 unassign 一次

6.2 对 table 的修改有问题如何回退

找到这个修改的 root procedure, bypass -or 来 bypass 所有相关的 procedure，利用 table unset 来重置 meta，因为 bypass 之后资源锁还是没有释放，所以需要手动加上 override 参数再去全部 assigns 一遍

6.3 Master 起不起来

日志里一般会有这个:

```
WARN org.apache.hadoop.hbase.master.HMaster: hbase:meta,,1.1588230740 is NO
T online; state={1588230740 state=CLOSING, ts=1538456302300, server=ve1017.
example.org,22101,1538449648131}; ServerCrashProcedures=true. Master startu
p cannot progress, in holding-pattern until region onlined.
```

手动去 assign 一下 meta 表即可，hbase:meta 表的 encoded name 是一个时间戳，比如上面日志的 encoded name 就是 1588230740

另外 hbase:namespace 表没有 online 也会造成这个问题，同样需要我们去手动 assign 一下

6.4 table 卡在 disabling 状态

因为要求是所有 region 都 disabled，那么解决办法可以是手动把没有 closed 的 region 根据 case1 来解决。如果所有 region 都已经是 closed 状态了，那么我可

以利用 setTableState 手动将表的状态设为 DISABLED。之后再 drop 都是安全的了。

7. 总体解决思路

其实 HBase-2.x 版本的运维思路很简单，因为使用了 procedure，集群出现 meta 跟 regionserver 不一致的状态是很少的，一般都是有 procedure 出问题了。那么我们主要就是看怎么解决这个有问题的 procedure。

如果是 table/namespace 级别的修改，因为设计到很多 region 的锁，如果需要 bypass 的话需要找到 root procedure 然后使用 bypass -or.

如果只是 region 级别的问题，则 bypass -o 即可。

bypass 之后检查 locks 的页面，看看是不是锁都释放了，如果没有锁了则根据需求进行 assign 或者 unassign,或者对 table 的属性进行还原。
参考链接：

[1] https://issues.apache.org/jira/browse/HBASE-19121

[2] https://github.com/apache/hbase-operator-tools

[3] https://issues.apache.org/jira/browse/HBASE-21393

HBase 2.0 新特性之 In-Memory Compaction

陆豪 阿里巴巴 技术专家

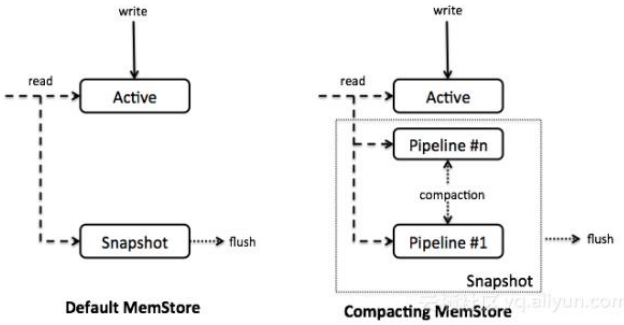
In-Memory Compaction 是 HBase2.0 中的重要特性之一，通过在内存中引入 LSM 结构，减少多余数据，实现降低 flush 频率和减小写放大的效果。本文根据 HBase2.0 中相关代码以及社区的讨论、博客，介绍 In-Memory Compaction 的使用和实现原理。

1. 原理

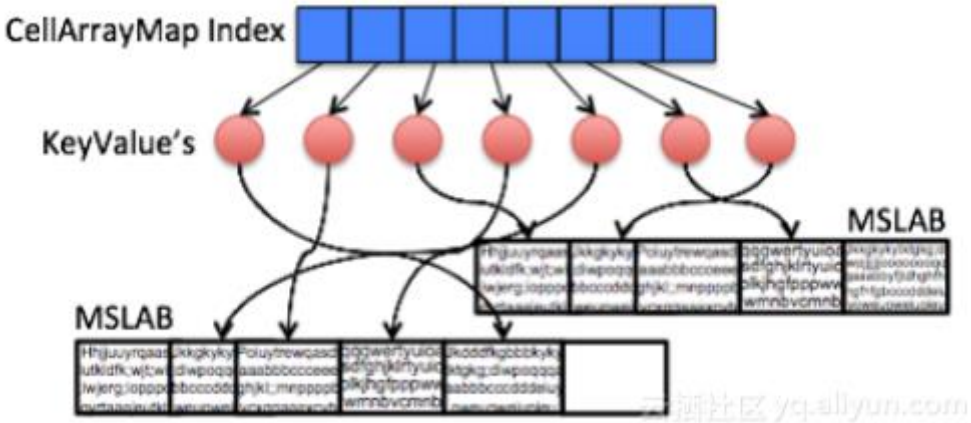
1.1 概念和数据结构

In-Memory Compaction 中引入了 MemStore 的一个新的实现类 CompactingMemStore。顾名思义，这个类和默认 memst。CompactingMemStore 中，数据以 segment 作为单位进行组织，一个 memStore 中包含多个 segment。数据写入时首先进入一个被称为 active 的 segment，这个 segment 是可修改的。当 active 满之后，会被移动到 pipeline 中，这个过程称为 in-memory flush。pipeline 中包含多个 segment，其中的数据不可修改。CompactingMemStore 会在后台将 pipeline 中的多个 segment 合并为一个更大、更紧凑的 segment，这就是 compaction 的过程。

如果 RegionServer 需要把 memstore 的数据 flush 到磁盘，会首先选择其他类型的 memstore，然后再选择 CompactingMemStore。这是因为 CompactingMemStore 对内存的管理更有效率，所以延长 CompactingMemStore 的生命周期可以减少总的 I/O。当 CompactingMemStore 被 flush 到磁盘时，pipeline 中的所有 segment 会被移到一个 snapshot 中进行合并然后写入 HFile。



在默认的 MemStore 中，对 cell 的索引使用 ConcurrentSkipListMap，这种结构支持动态修改，但是其中存在大量小对象，内存浪费比较严重。而在 CompactingMemStore 中，由于 pipeline 里面的数据是只读的，就可以使用更紧凑的数据结构来存储索引，减少内存使用。代码中使用 CellArrayMap 结构来存储 cell 索引，其内部实现是一个数组。



1.2 Compaction 策略

当一个 active segment 被 flush 到 pipeline 中之后，后台会触发一个任务对 pipeline 中的数据进行合并。合并任务会对 pipeline 中所有 segment 进行 scan，将他们的索引合并为一个。有三种合并策略可供选择：Basic,Eager,Adaptive。

Basic compaction 策略和 Eager compaction 策略的区别在于如何处理 cell 数据。Basic compaction 不会清理多余的数据版本，这样就不需要对 cell 的内存进行拷贝。而 Eager compaction 会过滤重复的数据，并清理多余的版本，这意味着会有额外的开销：例如如果使用了 MSLAB 存储 cell 数据，就需要把经过清理之后的 cell 从旧的 MSLAB 拷贝到新的 MSLAB。basic 适用于所有写入模式，eager 则主要针对数据大量淘汰的场景：例如消息队列、购物车等。

Adaptive 策略则是根据数据的重复情况来决定是否使用 Eager 策略。在 Adaptive 策略中，首先会对待合并的 segment 进行评估，方法是在已经统计过不重复 key 个数的 segment 中，找出 cell 个数最多的一个，然后用这个 segment 的 numUniqueKeys / getCellsCount 得到一个比例，如果比例小于设定的阈值，则使用 Eager 策略，否则使用 Basic 策略。

2. 使用

2.1 配置

2.0 中，默认的 In-Memory Compaction 策略为 basic。可以通过修改 hbase-site.xml 修改：

```
<property>
 <name>hbase.hregion.compacting.memstore.type</name>
 <value><none|basic|eager|adaptive></value>
</property>
```

也可以单独设置某个列族的级别：

```
create '<tablename>',
{NAME => '<cfname>', IN_MEMORY_COMPACTION => '<NONE|BASIC|EAGER|ADAPTIVE>'}
```

2.2 性能提升

社区的博客中给出了两个不同场景的测试结果。使用 YCSB 测试工具，100-200 GB 数据集。分别在 key 热度符合 Zipf 分布和平均分布两种情况下，测试了只有写操作情况下写放大、吞吐、GC 相比默认 Memstore 的变化，以及读写各占 50% 情况下尾部读延时的变化。测试结果如下表：

key 热度分布	写放大	吞吐	GC	尾部读延时
Zipf	30% ↓	20% ↑	22% ↓	12% ↓
平均分布	25% ↓	50% ↑	36% ↓	无变化

HBase Coprocessor 的实现与应用

叶锽 烽火大数据平台 研发负责人

本次分享的内容主要分为以下五点：

1. Coprocessor 简介
2. Endpoint 服务端实现
3. Endpoint 客户端实现
4. Observer 实现二级索引
5. Coprocessor 应用场景

1. Coprocessor 简介

HBase 协处理器的灵感来自于 Jeff Dean 09 年的演讲，根据该演讲实现类似于 Bigtable 的协处理器，包括以下特性:每个表服务器的任意子表都可以运行代码客户端的高层调用接口(客户端能够直接访问数据表的行地址，多行读写会自动分片成多个并行的 RPC 调用)，提供一个非常灵活的、可用于建立分布式服务的数据模型，能够自动化扩展、负载均衡、应用请求路由。HBase 的协处理器灵感来自 Bigtable，但是实现细节不尽相同。HBase 建立框架为用户提供类库和运行时环境，使得代码能够在 HBase Region Server 和 Master 上面进行处理。

(1)实现目的

1. HBase 无法轻易建立“二级索引”；
2. 执行求和、计数、排序等操作比较困难，必须通过 MapReduce/Spark 实现，对于简单的统计或聚合计算，可能会因为网络与 IO 开销大而带来性能问题。

(2)灵感来源

灵感来源于 Bigtable 的协处理器，包含如下特性：

1. 每个表服务器的任意子表都可以运行代码；
2. 客户端能够直接访问数据表的行，多行读写会自动分片成多个并行的 RPC 调用。

(3)提供接口

1. RegionObserver：提供客户端的数据操纵事件钩子：Get、Put、Delete、Scan 等；
2. WALObserver：提供 WAL 相关操作钩子；
3. MasterObserver：提供 DDL-类型的操作钩子。如创建、删除、修改数据表等；
4. Endpoint：终端是动态 RPC 插件的接口，它的实现代码被安装在服务器端，能够通过 HBase RPC 调用唤醒。

(4)应用范围

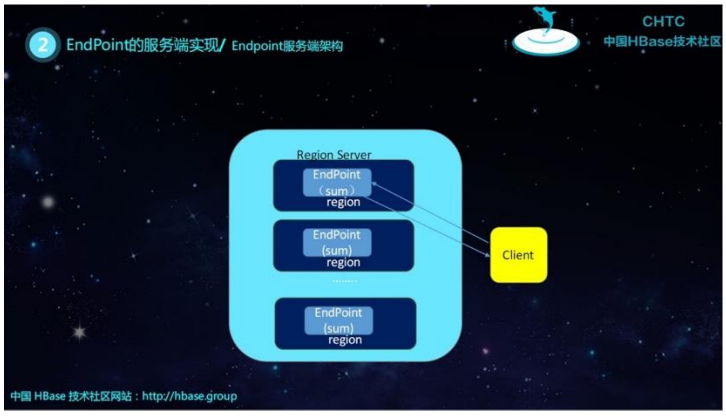
1. 通过使用 RegionObserver 接口可以实现二级索引的创建和维护；
2. 通过使用 Endpoint 接口，在对数据进行简单排序和 sum，count 等统计操作时，能够极大提高性能。

本文将通过具体实例来演示两种协处理器的开发方法的详细实现过程。

2. Endpoint 服务端实现

在传统关系型数据库里面，可以随时的对某列进行求和 sum，但是目前 HBase 目前所提供的接口，直接求和是比较困难的，所以先编写好服务端代码，并加载到对应的 Table 上，加载协处理器有几种方法，可以通过 HTableDescriptor 的 addCoprocessor 方法直接加载，同理也可以通过 removeCoprocessor 方法卸载协处理器。

Endpoint 协处理器类似传统数据库的存储过程，客户端调用 Endpoint 协处理器执行一段 Server 端代码，并将 Server 端代码的结果返回给 Client 进一步处理，最常见的用法就是进行聚合操作。举个例子说明：如果没有协处理器，当用户需要找出一张表中的最大数据即 max 聚合操作，必须进行全表扫描，客户端代码遍历扫描结果并执行求 max 操作，这样的方法无法利用底层集群的并发能力，而将所有计算都集中到 Client 端统一执行，效率非常低。但是使用 Coprocessor，用户将求 max 的代码部署到 HBase Server 端，HBase 将利用底层 Cluster 的多个节点并行执行求 max 的操作即在每个 Region 范围内执行求最大值逻辑，将每个 Region 的最大值在 Region Server 端计算出，仅仅将该 max 值返回给客户端。客户端进一步将多个 Region 的 max 进一步处理而找到其中的 max，这样整体执行效率提高很多。但是一定要注意的是 Coprocessor 一定要写正确，否则导致 RegionServer 宕机。



2.1 Protobuf 定义

如前所述，客户端和服务端之间需要进行 RPC 通信，所以两者间需要确定接口，当前版本的 HBase 的协处理器是通过 Google Protobuf 协议来实现数据交换的，所以需要通过 Protobuf 来定义接口。

如下所示：

```
option java_package = "com.my.hbase.protobuf.generated";
option java_outer_classname = "AggregateProtos";
option java_generic_services = true;
option java_generate_equals_and_hash = true;
option optimize_for = SPEED;

import "Client.proto";

message AggregateRequest {
    required string interpreter_class_name = 1;
    required Scan scan = 2;
    optional bytes  interpreter_specific_bytes = 3;
}

message AggregateResponse {
    repeated bytes first_part = 1;
    optional bytes second_part = 2;
}

service AggregateService {
    rpc GetMax (AggregateRequest) returns (AggregateResponse);
    rpc GetMin (AggregateRequest) returns (AggregateResponse);
    rpc GetSum (AggregateRequest) returns (AggregateResponse);
    rpc GetRowNum (AggregateRequest) returns (AggregateResponse);
    rpc GetAvg (AggregateRequest) returns (AggregateResponse);
    rpc GetStd (AggregateRequest) returns (AggregateResponse);
    rpc GetMedian (AggregateRequest) returns (AggregateResponse);
}
```

可以看到这里定义 7 个聚合服务 RPC，名字分别叫做 GetMax、GetMin、GetSum 等，本文通过 GetSum 进行举例，其他的聚合 RPC 也是类似的内部实现。RPC 有一个入口参数，用消息 AggregateRequest 表示；RPC 的返回值用消息 AggregateResponse 表示。Service 是一个抽象概念，RPC 的 Server 端可以看作一个用来提供服务的 Service。在 HBase Coprocessor 中 Service 就是 Server 端需要提供的 Endpoint Coprocessor 服务，主要用来给 HBase 的 Client 提供服务。AggregateService.java 是由 Protobuf 软件通过终端命令“protoc filename.proto--java_out=OUT_DIR”自动生成的，其作用是将.proto 文件定义的消息结构以及服务转换成对应接口的 RPC 实现，其中包括如何构建 request 消息和 response 响应以及消息包含的内容的处理方式，并且将 AggregateService 包装成一个抽象类，具体的服务以类的方法的形式提供。AggregateService.java 定义 Client 端与 Server 端通信的协议，代码中包含请求信息结构 AggregateRequest、响应信息结构 AggregateResponse、提供的服务种类 AggregateService，其中 AggregateRequest 中的 interpreter_class_name 指的是 column interpreter 的类名，此类的作用在于将数据格式从存储类型解析成所需类型。AggregateService.java 由于代码太长，在这里就不贴出来了。

下面我们来讲一下服务端的架构：

首先，Endpoint Coprocessor 是一个 Protobuf Service 的实现，因此需要它必须继承某个 ProtobufService。我们在前面已经通过 proto 文件定义 Service，命名为 AggregateService，因此 Server 端代码需要重载该类，其次作为 HBase 的协处理器，Endpoint 还必须实现 HBase 定义的协处理器协议，用 Java 的接口来定义。具体来说就是 CoprocessorService 和 Coprocessor，这些 HBase 接口负责将协处理器和 HBase 的 RegionServer 等实例联系起来以便协同工作。Coprocessor 接口定义两个接口函数：start 和 stop。

加载 Coprocessor 之后 Region 打开的时候被 RegionServer 自动加载，并会调用器 start 接口完成初始化工作。一般情况该接口函数仅仅需要将协处理器的运行上下文环境变量 CoprocessorEnvironment 保存到本地即可。

CoprocessorEnvironment 保存协处理器的运行环境，每个协处理器都是在一个 RegionServer 进程内运行并隶属于某个 Region。通过该变量获取 Region 的实例等 HBase 运行时环境对象。

Coprocessor 接口还定义 stop()接口函数，该函数在 Region 被关闭时调用，用来进行协处理器的清理工作。本文里我们没有进行任何清理工作，因此该函数什么也不干。

我们的协处理器还需要实现 CoprocessorService 接口。该接口仅仅定义一个接口函数 getService()。我们仅需要将本实例返回即可。HBase 的 Region Server 在接收到客户端的调用请求时，将调用该接口获取实现 RPCService 的实例，因此本函数一般情况下就是返回自身实例即可。

完成以上三个接口函数之后，Endpoint 的框架代码就已完成。每个 Endpoint 协处理器都必须实现这些框架代码而且写法雷同。

Server 端的代码就是一个 Protobuf RPC 的 Service 实现，即通过 Protobuf 提供的某种服务。其开发内容主要包括：

1. 实现 Coprocessor 的基本框架代码
2. 实现服务的 RPC 具体代码

2.2 Endpoint 协处理的基本框架

Endpoint 是一个 Server 端 Service 的具体实现，其实现有一些框架代码，这些框架代码与具体的业务需求逻辑无关。仅仅是为了和 HBase 运行时环境协同工作而必须遵循和完成的一些粘合代码。因此多数情况下仅仅需要从一个例子程序拷贝过来并进行命名修改即可。不过我们还是完整地对这些粘合代码进行粗略的讲解以便更好地理解代码。

```
public Service getService() {
    return this;
}

public void start(CoprocessorEnvironment env) throws IOException {
    if(env instanceof RegionCoprocessorEnvironment) {
        this.env = (RegionCoprocessorEnvironment)env;
    } else {
        throw new CoprocessorException("Must be loaded on a table region!");
    }
}

public void stop(CoprocessorEnvironment env) throws IOException {
}
```

Endpoint 协处理器真正的业务代码都在每一个 RPC 函数的具体实现中。

在本文中，我们的 Endpoint 协处理器仅提供一个 RPC 函数即 getSUM。我将分别介绍编写该函数的几个主要工作 :了解函数的定义, 参数列表 ;处理入口参数；实现业务逻辑；设置返回参数。

```
public void getSum(RpcController controller, AggregateRequest request, RpcCal
lbackdone) {
       AggregateResponse response = null;
       RegionScanner scanner = null;
```

```
        long sum = 0L;
        try {
            ColumnInterpreter ignored = this.constructColumnInterpreterFromRequ
est(request);
            Object sumVal = null;
            Scan scan = ProtobufUtil.toScan(request.getScan());
            scanner = this.env.getRegion().getScanner(scan);
            byte[] colFamily = scan.getFamilies()[0];
            NavigableSet qualifiers = (NavigableSet) scan.getFamilyMap().get(co
lFamily);
            byte[] qualifier = null;
            if (qualifiers != null && !qualifiers.isEmpty()) {
                qualifier = (byte[]) qualifiers.pollFirst();
            }

            ArrayList results = new ArrayList();
            boolean hasMoreRows = false;

            do {
                hasMoreRows = scanner.next(results);
                int listSize = results.size();

                for (int i = 0; i < listSize; ++i) {
                    //取出列值
                    Object temp = ignored.getValue(colFamily, qualifier,
                            (Cell) results.get(i));
                    if (temp != null) {
                        sumVal = ignored.add(sumVal, ignored.castToReturnType(tem
p));
                    }
```

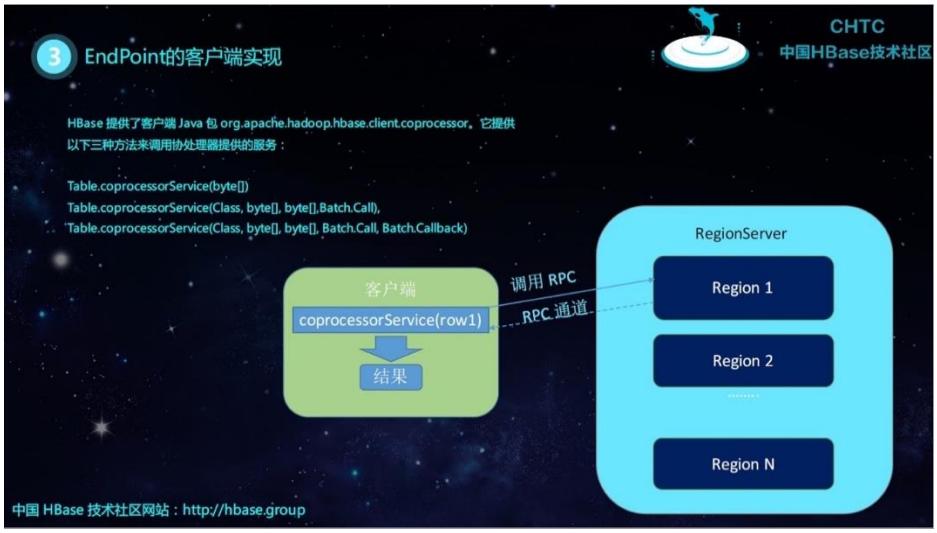
```
            }


            results.clear();
        } while (hasMoreRows);


        if (sumVal != null) {
            response = AggregateResponse.newBuilder().addFirstPart(
                    ignored.getProtoForPromotedType(sumVal).toByteString()).
build();
        }
    } catch (IOException var27) {
        ResponseConverter.setControllerException(controller, var27);
    } finally {
        if (scanner != null) {
            try {
                scanner.close();
            } catch (IOException var26) {
                ;
            }
        }


    }


    log.debug("Sum from this region is " +
            this.env.getRegion().getRegionInfo().getRegionNameAsString() +
": " + sum);
    done.run(response);
  }
```

Endpoint 类比于数据库的存储过程，其触发服务端的基于 Region 的同步运行再将各个结果在客户端搜集后归并计算。特点类似于传统的 MapReduce 框架，服务端 Map 客户端 Reduce。

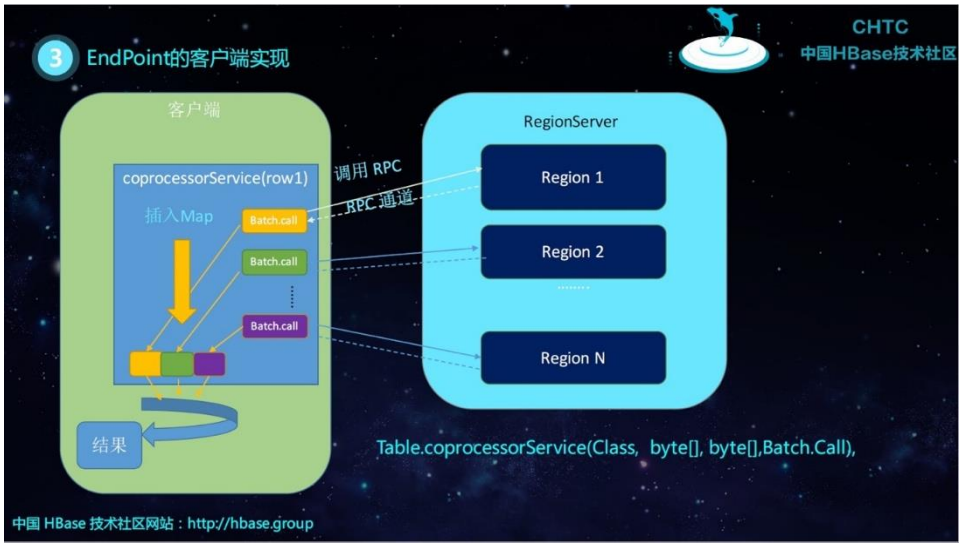
3. Endpoint 客户端实现

HBase 提供客户端 Java 包 org.apache.hadoop.hbase.client.HTable，提供以下三种方法来调用协处理器提供的服务：

1. `coprocessorService(byte[])`
2. `coprocessorService(Class, byte[], byte[],Batch.Call)`,
3. `coprocessorService(Class, byte[], byte[],Batch.Call, Batch.Callback)`



该方法采用 rowkey 指定 Region。这是因为 HBase 客户端很少会直接操作 Region，一般不需要知道 Region 的名字；况且在 HBase 中 Region 名会随时改变，所以用 rowkey 来指定 Region 是最合理的方式。使用 rowkey 可以指定唯一的一个 Region，如果给定的 Rowkey 并不存在，只要在某个 Region 的 rowkey 范围内依然用来指定该 Region。比如 Region 1 处理[row1, row100]这个区间内的数据，则 rowkey=row1 就由 Region 1 来负责处理，换句话说我们可以用 row1 来指定 Region 1，无论 rowkey 等于"row1"的记录是否存在。CoprocessorService 方法返回类型为 CoprocessorRpcChannel 的对象，该 RPC 通道连接到由 rowkey 指定的 Region 上面，通过此通道可以调用该 Region 上面部署的协处理器 RPC。



有时候客户端需要调用多个 Region 上的同一个协处理器，比如需要统计整个 Table 的 sum，在这种情况下，需要所有的 Region 都参与进来，分别统计自身 Region 内部的 sum 并返回客户端，最终客户端将所有 Region 的返回结果汇总，就可以得到整张表的 sum。

这意味着该客户端同时和多个 Region 进行批处理交互。一个可行的方法是，收集每个 Region 的 startkey，然后循环调用第一种 coprocessorService 方法：用每一个 Region 的 startkey 作为入口参数，获得 RPC 通道创建 stub 对象，进而逐一调用每个 Region 上的协处理器 RPC。这种做法需要写很多的代码，为此 HBase 提供两种更加简单的 coprocessorService 方法来处理多个 Region 的协处理器调用。先来看第一种方法 coprocessorService(Class, byte[],byte[],Batch.Call),

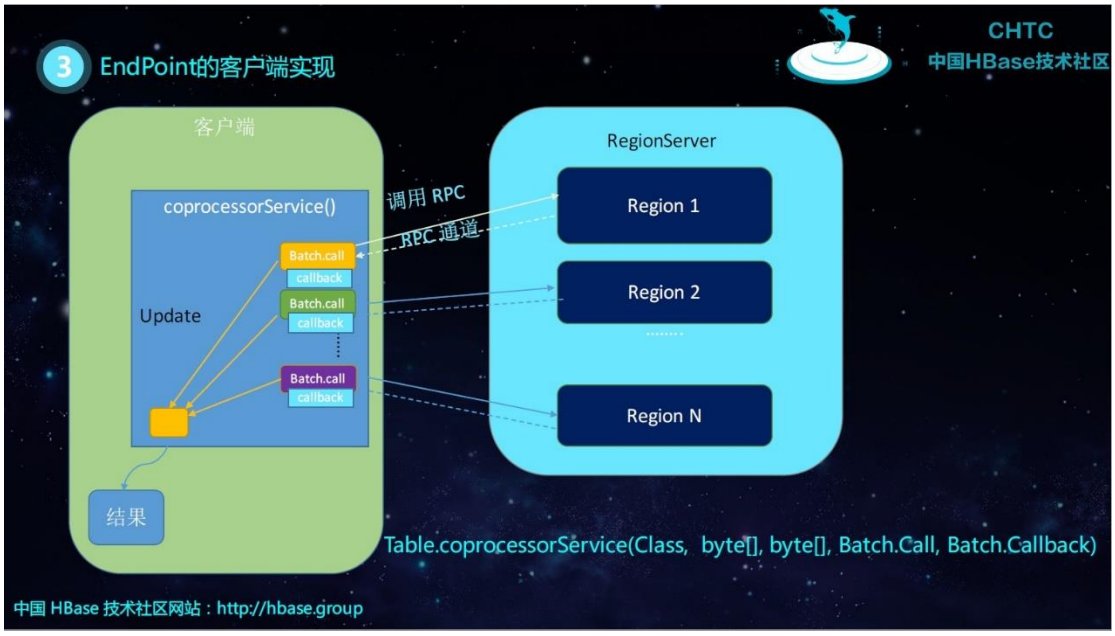
该方法有 4 个入口参数。第一个参数是实现 RPC 的 Service 类，即前文中的 AggregateService 类。通过它，HBase 就可以找到相应的部署在 Region 上的协处理器，一个 Region 上可以部署多个协处理器，客户端必须通过指定 Service 类来区分究竟需要调用哪个协处理器提供的服务。

要调用哪些 Region 上的服务则由 startkey 和 endkey 来确定，通过 rowkey 范围即可确定多个 Region。为此，coprocessorService 方法的第二个和第三个参数分别是 startkey 和 endkey，凡是落在[startkey，endkey]区间内的 Region 都会参与本次调用。

第四个参数是接口类 Batch.Call。它定义了如何调用协处理器，用户通过重载该接口的 call()方法来实现客户端的逻辑。在 call()方法内，可以调用 RPC，并对返回值进行任意处理。即前文代码清单 1 中所做的事情。coprocessorService 将负责对每个 Region 调用这个 call()方法。

coprocessorService 方法的返回值是一个 Map 类型的集合。该集合的 key 是 Region 名字，value 是 Batch.Call.call 方法的返回值。该集合可以看作是所有 Region 的协处理器 RPC 返回的结果集。客户端代码可以遍历该集合对所有的结果进行汇总处理。

这种 coprocessorService 方法的大体工作流程如下。首先它分析 startkey 和 endkey，找到该区间内的所有 Region，假设存放在 regionList 中。然后，遍历 regionList，为每一个 Region 调用 Batch.Call，在该接口内，用户定义具体的 RPC 调用逻辑。最后 coprocessorService 将所有 Batch.Call.call()的返回值加入结果集合并返回。



coprocessorService 的第三种方法比第二个方法多了一个参数 callback。coprocessorService 第二个方法内部使用 HBase 自带的缺省 callback，该缺省 callback 将每个 Region 的返回结果都添加到一个 Map 类型的结果集中，并将该集合作为 coprocessorService 方法的返回值。

HBase 提供第三种 coprocessorService 方法允许用户定义 callback 行为，

coprocessorService 会为每一个 RPC 返回结果调用该 callback，用户可以在 callback 中执行需要的逻辑，比如执行 sum 累加。用第二种方法的情况下，每个 Region 协处理器 RPC 的返回结果先放入一个列表，所有的 Region 都返回后，用户代码再从该列表中取出每一个结果进行累加 ;用第三种方法，直接在 callback 中进行累加，省掉了创建结果集合和遍历该集合的开销，效率会更高一些。

因此我们只需要额外定义一个 callback 即可，callback 是一个 Batch.Callback 接口类，用户需要重载其 update 方法。

```
public S sum(final HTable table, final ColumnInterpreter<R, S, P, Q, T> ci,fin
al Scan scan)throws Throwable {


final AggregateRequest requestArg = validateArgAndGetPB(scan, ci, false);


class SumCallBack implements Batch.Callback  {


S sumVal = null;


public S getSumResult() {

  return sumVal;

}


@Override

public synchronized void update(byte[] region, byte[] row, S result) {

  sumVal = ci.add(sumVal, result);

}}


SumCallBack sumCallBack = new SumCallBack();

table.coprocessorService(AggregateService.class, scan.getStartRow(), scan.get
StopRow(),

     new Batch.Call<AggregateService, S>() {
```

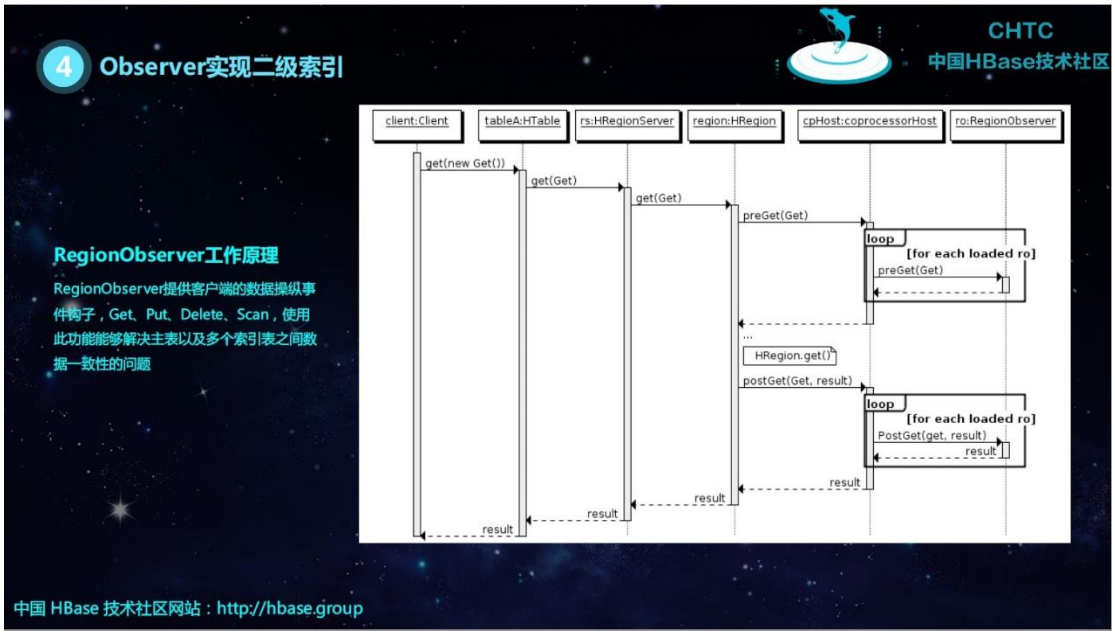
```
      @Override
      public S call(AggregateService instance) throws IOException {
        ServerRpcController controller = new ServerRpcController();
        BlockingRpcCallback<AggregateResponse> rpcCallback =
              new BlockingRpcCallback<AggregateResponse>();
          //RPC 调用
        instance.getSum(controller, requestArg, rpcCallback);
        AggregateResponse response = rpcCallback.get();
        if (controller.failedOnException()) {
          throw controller.getFailedOn();
        }
        if (response.getFirstPartCount() == 0) {
          return null;
        }
        ByteString b = response.getFirstPart(0);
        T t = ProtobufUtil.getParsedGenericInstance(ci.getClass(), 4, b);
        S s = ci.getPromotedValueFromProto(t);
        return s;
      }
    }, sumCallBack);
return sumCallBack.getSumResult();
```

4. Observer 实现二级索引

Observer 类似于传统数据库中的触发器，当发生某些事件的时候这类协处理器会被 Server 端调用。Observer Coprocessor 是一些散布在 HBase Server 端代码的 hook 钩子， 在固定的事件发生时被调用。比如：put 操作之前有钩子函数 prePut，该函数在 pu 操作执 行前会被 Region Server 调用；在 put 操作之后则有 postPut 钩子函数。

RegionObserver 工作原理

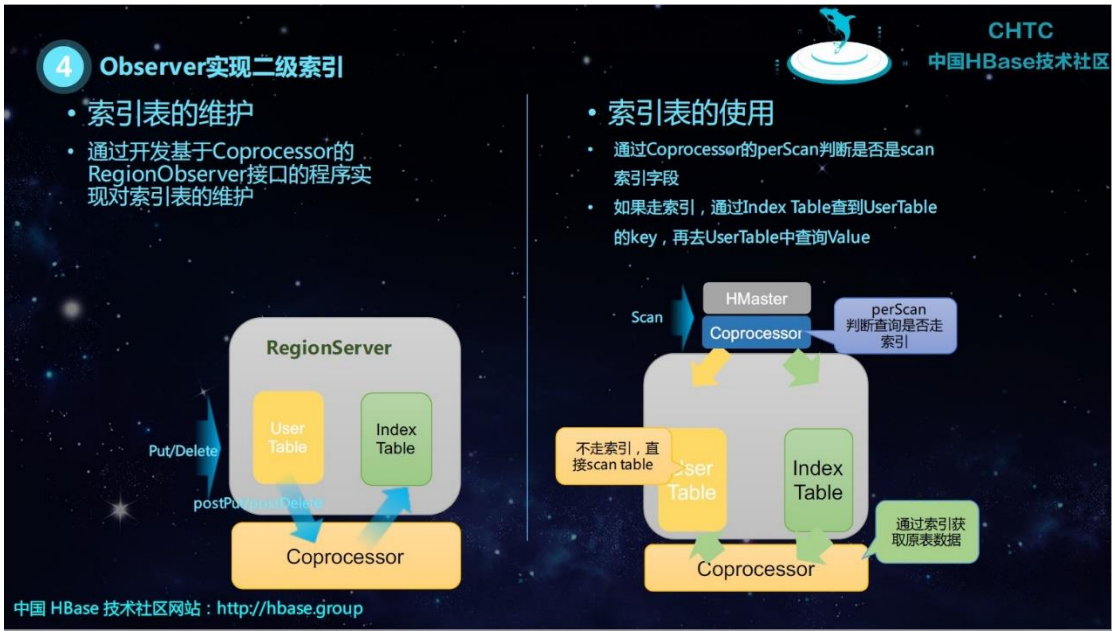
RegionObserver 提供客户端的数据操纵事件钩子，Get、Put、Delete、Scan，使用此功能能够解决主表以及多个索引表之间数据一致性的问题。



1. 客户端发出 put 请求；
2. 该请求被分派给合适的 RegionServer 和 Region；
3. coprocessorHost 拦截该请求，然后在该表上登记的每个 RegionObserver 上调用 prePut()；

4. 如果没有被 preGet()拦截，该请求继续送到 region，然后进行处理；
5. Region 产生的结果再次被 CoprocessorHost 拦截，调用 postGet()；
6. 假如没有 postGet()拦截该响应，最终结果被返回给客户端；

如上图所示，HBase 可以根据 rowkey 很快的检索到数据，但是如果根据 column 检索数据，首先要根据 rowkey 减小范围，再通过列过滤器去过滤出数据，如果使用二级索引，可以先查基于 column 的索引表，获取到 rowkey 后再快速的检索到数据。



如图所示首先继承 BaseRegionObserver 类，重写 postPut，postDelete 方法，在 postPut 方法体内中写 Put 索引表数据的代码，在 postDelete 方法里面写 Delete 索引表数据，这样可以保持数据的一致性。

在 Scan 表的时候首先判断是否先查索引表，如果不查索引直接 scan 主表，如果走索引表通过索引表获取主表的 rowkey 再去查主表。

使用 Elastic Search 建立二级索引也是一样。

我们在同一个主机集群上同时建立了 HBase 集群和 Elastic Search 集群，存储到 HBase 的数据必须实时地同步到 Elastic Search。而恰好 HBase 和 Elastic Search 都没有更新的概念，我们的需求可以简化为两步：

1. 当一个新的 Put 操作产生时，将 Put 数据转化为 json，索引到 ElasticSearch，并把 RowKey 作为新文档的 ID；
2. 当一个新的 Delete 操作产生时获取 Delete 数据的 rowkey 删除 Elastic Search 中对应的 ID。

5. 协处理的主要应用场景

1. Observer 允许集群在正常的客户端操作过程中可以有不同的行为表现；
2. Endpoint 允许扩展集群的能力，对客户端应用开放新的运算命令；
3. Observer 类似于 RDBMS 的触发器，主要在服务端工作；
4. Endpoint 类似于 RDBMS 的存储过程，主要在服务端工作；
5. Observer 可以实现权限管理、优先级设置、监控、ddl 控制、二级索引等功能；
6. Endpoint 可以实现 min、max、avg、sum、distinct、group by 等功能

例如 HBase 源码 org.apache.hadoop.hbase.security.access.AccessController 利用 Observer 实现对 HBase 进行了权限控制，有兴趣的读者可以看看相关代码。

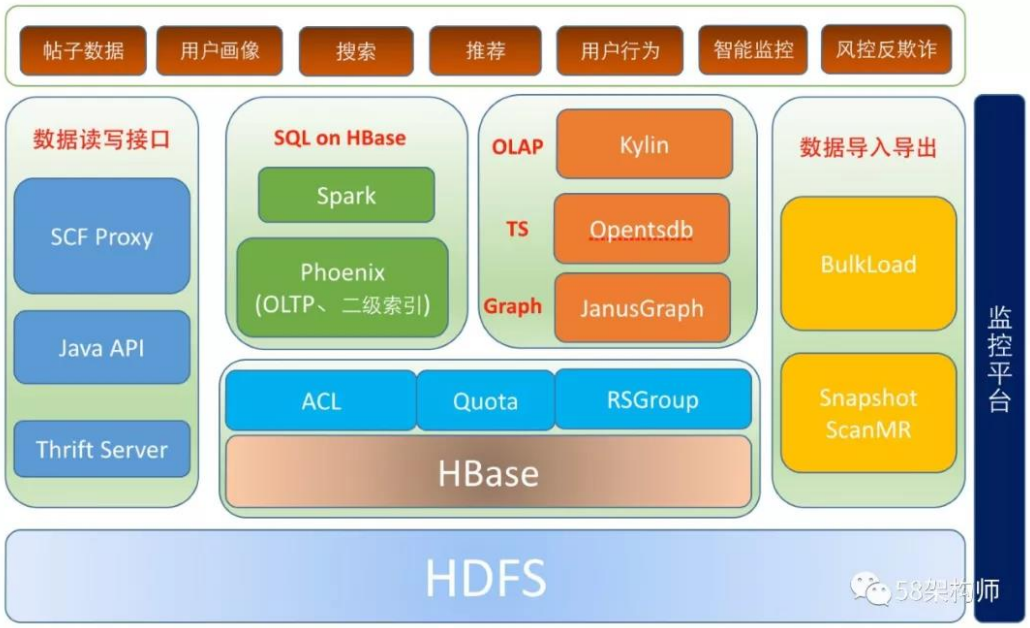
58 HBase 平台实践和应用-平台建设篇

何良均/张祥 58 同城 资深研发工程师

HBase 是一个基于 Hadoop 的分布式、面向列的 Key-Value 存储系统，可以对需要实时读写、随机访问大规模数据集的场景提供高可靠、高性能的服务，在大数据相关领域应用广泛。HBase 可以对数据进行透明的切分，使得存储和计算本身具有良好的水平扩展性。

在 58 的业务场景中，HBase 扮演重要角色。例如帖子信息等公司基础数据都是通过 HBase 进行离线存储，并为各个业务线提供随机查询及更深层次的数据分析。同时 HBase 在 58 还大量用于用户画像、搜索、推荐、时序数据和图数据等场景的存储和查询分析。

HBase 在 58 的应用架构：



HBase 在 58 的应用架构如上图所示，主要内容包括以下几个部分：

1. 多租户支持：包括 SCF 限流、RSGroup、RPC 读写分离、HBase Quota 、ACL；
2. 数据读写接口：包括 SCF 代理 API、原生 Java API 以及跨语言访问 Thrift Server；
3. HBase 数据导入导出：包括数据批量导入工具 BulkLoad，数据批量导出工具 SnapshotMR；

4. OLAP：多维分析查询的 Kylin 平台；
5. 时序数据库：时序数据存储和查询的时序数据库 Opentsdb；
6. 图数据库：图关系数据存储和查询的图数据库 JanusGraph；
7. SQL on HBase：支持二级索引和事务的 Phoenix，以及 Spark SQL 等；
8. HBase 在 58 的应用业务场景包括：全量帖子数据、用户画像、搜索、推荐、用户行为、智能监控以及风控反欺诈等的数据存储和分析；
9. 监控平台：HBase 平台的监控实现。

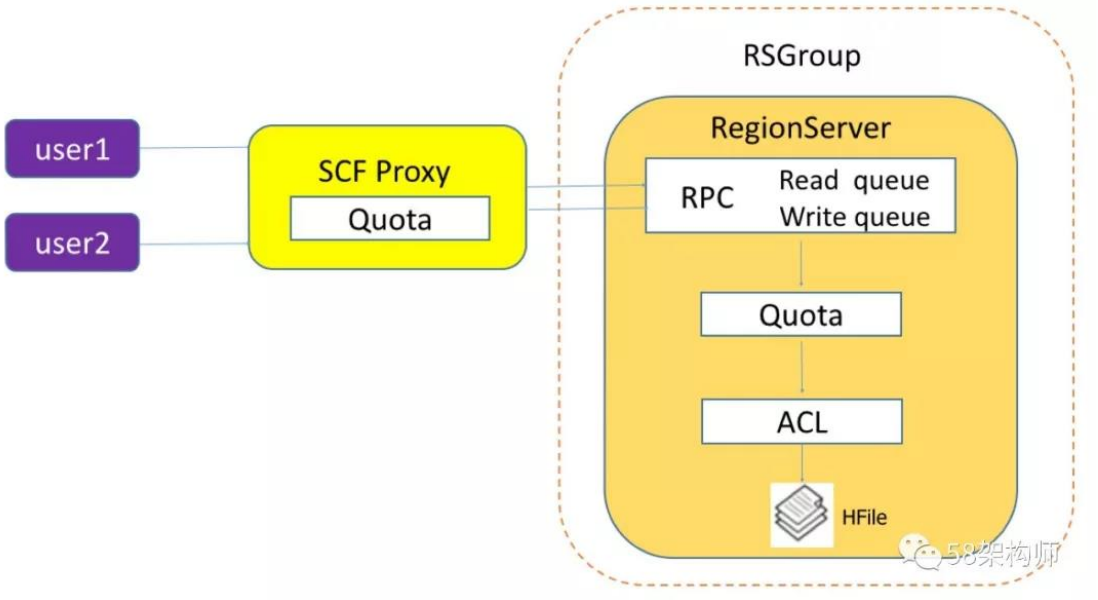
本文将从多租户支持、数据读写接口、数据导入导出和平台优化四个方面来重点讲解 58HBase 平台的建设。

说明：下文中所有涉及到 RegionServer 的地方统一使用 RS 来代替。

1. HBase 多租户支持

HBase 在 1.1.0 版本之前没有多租户的概念，同一个集群上所有用户、表都是同等的，导致各个业务之间干扰比较大，特别是某些重要业务，需要在资源有限的情况下保证优先正常运行，但是在之前的版本中是无法保证的。从 HBase 1.1.0 开始，HBase 的多租户特性逐渐得到支持，我们的 HBase 版本是 1.0.0-cdh5.4.4，该版本已经集成了多租户特性。

以下是 58 用户访问 HBase 的流程图：



我们从多个层面对 HBase 多租户进行了支持，主要分为以下两个大的方面：

1. 资源限制：
   a) SCF Quota；
   b) HBase Quota。
2. 资源隔离：
   a) RS RPC 读写分离；
   b) HBase ACL 权限隔离；
   c) RSGroup 物理隔离。

1.1 资源限制

(1)SCF Quota

SCF 是公司自研的 RPC 框架，我们基于 SCF 封装了原生 HBase API，用户根据应用需要申请 HBase SCF 服务调用时，需要根据应用实际情况填写 HBase 的每分钟调用量(请求次数)，在调用量超限时，SCF 管理平台可以实现应用级的限流，这是全局限流。缺点是只能对调用量进行限制，无法对读写数据量大小限制。

以下是用户申请 HBase SCF 服务调用时需要填写的调用量：

(2)HBase Quota

HBase 的 Quota 功能可以实现对用户级、表级和命名空间级的资源进行限制。这里的资源包括请求数据量大小、请求次数两个维度，这两个维度基本涵盖了常见的资源限制。目前 HBase 的 Quota 功能只能限制到 RS 这一级，不是针对整个集群的。但是因为可以对请求的数据量大小进行限制，一定程度上可以弥补了 SCF Proxy 应用级限流只能对请求次数进行限制的不足。

开启 Quota 的配置如下：

```
<property>
  <name>hbase.quota.enabled</name>
  <value>true</value>
</property>
```

在开启了 HBase 的 Quota 后，Quota 相关的元数据会存储到 HBase 的系统表 hbase:quota 中。

在我们的 HBase 集群中之前遇到过个别用户读写数据量过大导致 RS 节点带宽被打满，甚至触发 RS 的 FGC，导致服务不稳定，影响到了其他的业务，但是应用级的调用量并没有超过申请 SCF 时设置的值，这个时候我们就可以通过设置 HBase Quota，限制读写表级数据量大小来解决这个问题。

以下是设置 HBase Quota 信息，可以通过命令行进行设置和查看：

```
hbase(main):006:0* list_quotas
OWNER                                              QUOTAS
 TABLE => [redacted] TYPE => THROTTLE, THROTTLE_TYPE => REQUEST_SIZE, LIMIT => 50M/sec, SCOPE => MACHINE
 og_sys
1 row(s) in 0.0110 seconds
```

1.2 资源隔离

(1)RS RPC 读写分离

默认场景下，HBase 只提供一个 RPC 请求队列，所有请求都会进入该队列进行优先级排序。这样可能会出现由于读问题阻塞所有 handler 线程导致写数据失败，或者由于写问题阻塞所有 handler 线程导致读数据失败，这两种问题我们都遇到过，在后续篇幅中会提到，这里不细述。

通过设置参数 hbase.ipc.server.callqueue.handler.factor 来设置多个队列，队列个数等于该参数 * handler 线程数，比如该参数设置为 0.1，总的 handler 线程数为

200，则会产生 20 个独立队列。独立队列产生之后，可以通过参数 hbase.ipc.server.callqueue.read.ratio 来设置读写队列比例，比如设置 0.6，则表示会有 12 个队列用于接收读请求，8 个用于接收写请求；另外，还可以进一步通过参数 hbase.ipc.server.callqueue.scan.ratio 设置 get 和 scan 的队列比例，比如设置为 0.2，表示 2 个队列用于 scan 请求，另外 10 个用于 get 请求，进一步还将 get 和 scan 请求分开。

RPC 读写分离设计思想总体来说实现了读写请求队列资源的隔离，达到读写互不干扰的目的，根据 HBase 集群服务的业务类型，我们还可以进一步配置长时 scan 读和短时 get 读之间的队列隔离，实现长时读任务和短时读任务互不干扰。

(2)HBase ACL 权限隔离

HBase 集群多租户需要关注的一个核心问题是数据访问权限的问题，对于一些重要的公共数据，或者要进行跨部门访问数据，我们只开放给经过权限申请的用户访问，没有权限的用户是不能访问的，这就涉及到了 HBase 的数据权限隔离了，HBase 是通过 ACL 来实现权限隔离的。

基于 58 的实际应用情况，访问 HBase 的用户都是 Hadoop 计算集群的用户，而且 Hadoop 用户是按部门分配的，所以 HBase 的用户也是到部门而不是到个人，这样的好处是维护的用户数少了，便于管理，缺点是有的部门下面不同子部门之间如果也要进行数据权限隔离就比较麻烦，需要单独申请开通子部门账号。

要开启 HBase 的 ACL，只需要在配置文件 hbase-site.xml 中关于 Master、RegionServer 和 Region 的协处理器都加上 org.apache.hadoop.hbase.security.access.AccessController 类就可以了。具体 HBase ACL 的配置项如下图所示：

```
<property>
  <name>hbase.superuser</name>
  <value>hbase</value>
</property>
<property>
  <name>hbase.security.authorization</name>
  <value>true</value>
</property>
<property>
  <name>hbase.security.exec.permission.checks</name>
  <value>true</value>
</property>
<property>
  <name>hbase.coprocessor.master.classes</name>
  <value>org.apache.hadoop.hbase.security.access.AccessController</value>
</property>
<property>
  <name>hbase.coprocessor.regionserver.classes</name>
  <value>org.apache.hadoop.hbase.security.access.AccessController</value>
</property>
<property>
  <name>hbase.coprocessor.region.classes</name>
  <value>org.apache.hadoop.hbase.security.access.AccessController,org.apache.hadoop.hbase.security.token.TokenProvider</value>
</property>
```

HBase 的访问级别有读取(R)、写入(W)、执行(X)、创建(C)、管理员(A)，而权限作用域包括超级用户、全局、命名空间、表、列族、列。访问级别和作用域的组合创建了可授予用户的可能访问级别的矩阵。在生产环境中，根据执行特定工作所需的内容来考虑访问级别和作用域。

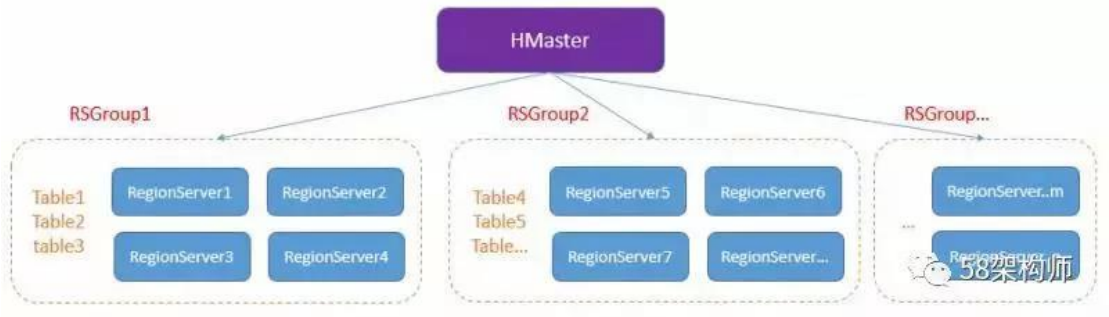
在 58 的实际应用中，我们将用户和 HBase 的命名空间一一对应，创建新用户时，创建同名的命名空间，并赋予该用户对同名命名空间的所有权限(RWCA)。以下以新用户 zhangsan 为例，创建同名命名空间并授权：

```
create_namespace 'zhangsan'

grant 'zhangsan','RWCA','@zhangsan'
```

(3)RSGroup 物理隔离

虽然 SCF Quota 和 HBase Quota 功能可以做到对用户的读写进行限制，一定程度上能降低各业务读写数据的相互干扰，但是在我们的实际业务场景中，存在两类特殊业务，一类是消耗资源非常大，但是不希望被限流，另外一类是非常重要，需要高优先级保证服务的稳定。对于这两种情况下，我们只能对该业务进行物理隔离，物理隔离既能保证重要业务的稳定性，也避免了对其他业务的干扰。我们使用的物理隔离方案是 RSGroup，也即 RegionServer Group。

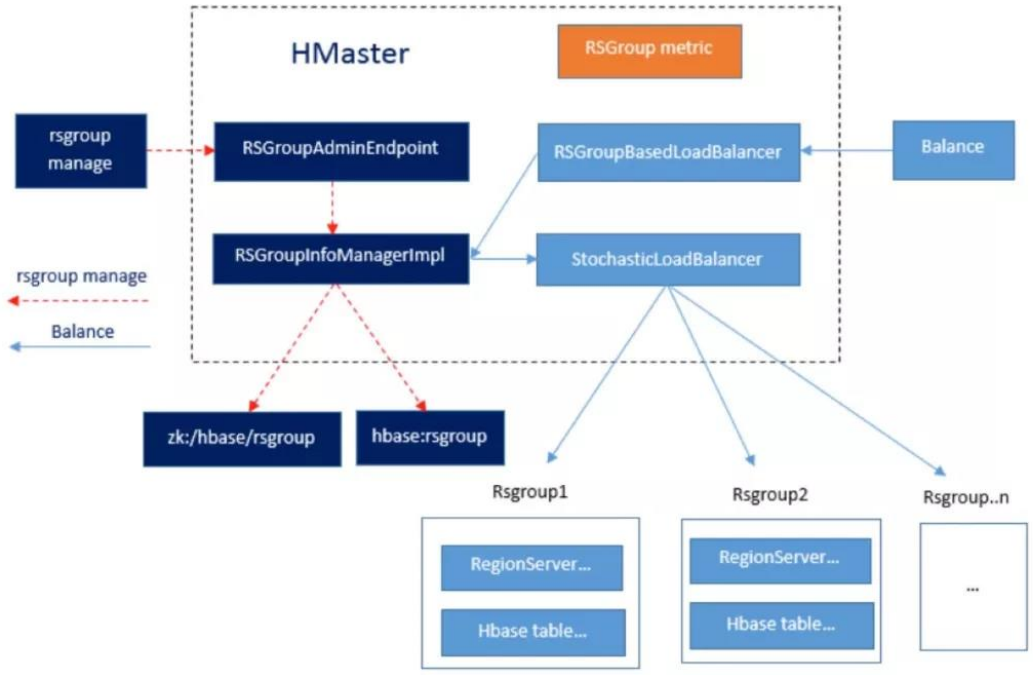
RSGroup 整体架构：



RSGroup 有以下几个特点：

1. 不同 RS 和表划分到不同的 RSGroup；
2. 同一个 RS 只能属于一个 RSGroup；
3. 同一个表也只能属于一个 RSGroup；
4. 默认所有 RS 和表都属于“default”这个 RSGroup。

RSGroup 实现细节：



从以上 RSGroup 实现细节中看出，RSGroup 的功能主要包含两部分，RSGroup 元数据管理以及 Balance。

RSGroup 开启的配置项：

```
<property>
   <name>hbase.coprocessor.master.classes</name>
   <value>org.apache.hadoop.hbase.rsgroup.RSGroupAdminEndpoint</value>
</property>
<property>
   <name>hbase.master.loadbalancer.class</name>
   <value>org.apache.hadoop.hbase.rsgroup.RSGroupBasedLoadBalancer</value>
</property>
```

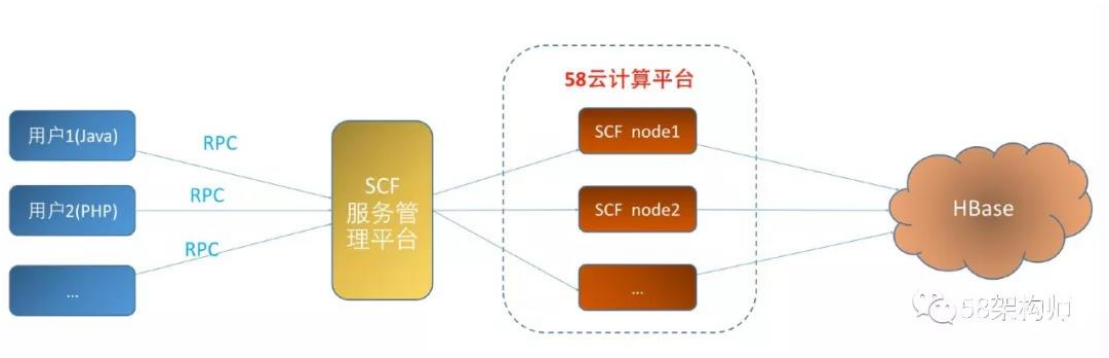
2. 数据读写接口

目前我们提供了三种 HBase 的数据读写接口以便于用户使用，包括 SCF 代理、Java 原生 API 和 Thrift Server。以下分别进行说明：

2.1 SCF Proxy

SCF 是 58 架构部自研的 RPC 框架，我们基于 SCF 封装了原生的 Java API，以 SCF RPC 接口的方式暴露给用户使用，其中以这种方式提供给用户的接口多达 30 个。由于 SCF 支持跨语言访问，很好的解决了使用非 Java 语言用户想要访问 HBase 数据的问题，目前用户使用最多的是通过 Java、Python 和 PHP 这三种语言来访问这些封装的接口。

SCF proxy 接口整体架构：

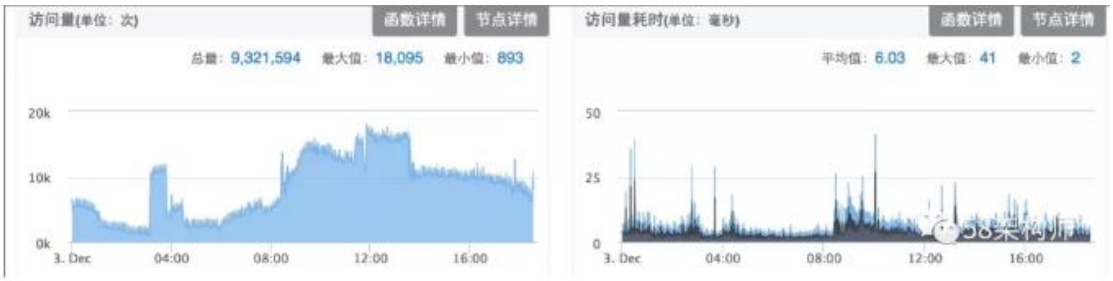


数据读写流程：用户通过 RPC 连接到 SCF 服务管理平台，通过 SCF 服务管理平台做服务发现，找到 58 云计算平台上部署的服务节点，服务节点最终通过访问 HBase 实现用户数据的读写操作。

使用 SCF Proxy 接口的优势：

1. 避免用户直连 HBase 集群，降低 zk 的压力。之前经常遇到因为用户代码存在 bug，导致 zk 连接数暴涨的情况。
2. 针对大量一次性扫描数据的场景，提供单独访问接口，并在接口中设置 scan 的 blockcache 熟悉为 false，避免了对后端读缓存的干扰。
3. 通过服务管理平台的服务发现和服务治理能力，结合业务的增长情况以及基于 58 云计算平台弹性特点，我们很容易对服务节点做自动扩容，而这一切对用户是透明的。
4. 通过服务管理平台可以实现对用户的访问做应用级限流，规范用户的读写操作。
5. 服务管理平台提供了调用量、查询耗时以及异常情况等丰富的图表，用户可以很方便查看。

以下是我们的 SCF 服务在服务管理平台展示的调用量和查询耗时图表：



由于 SCF Proxy 接口的诸多优势，我们对于新接的业务都要求通过申请这种方式来访问 HBase。

2.2 Java API

由于历史原因和个别特殊的新业务还采用 Java 原生的 API 外，其他新业务都通 SCF Proxy 接口来访问。

2.3 Thrift Server

也是由于历史原因，个别用户想使用非 Java 语言来访问 HBase，才启用了 Thrift Server，由于 SCF proxy 接口支持多语言，目前这种跨语言访问的问题都通过 SCF Proxy 来解决了。

3. 数据导入导出

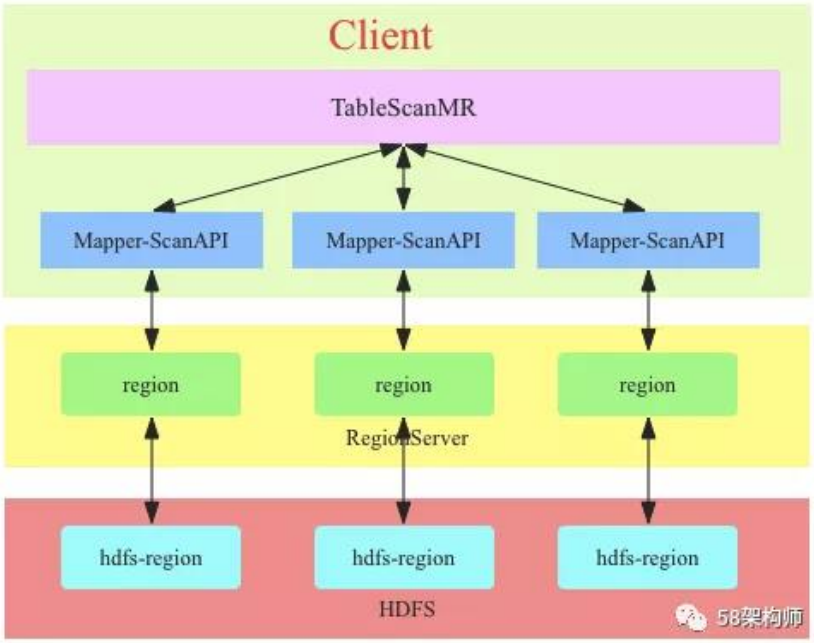
3.1 BulkLoad

HBase 相对于其他 KV 存储系统来说比较大的一个优势是提供了强大的批量导入工具 BulkLoad，通过 BulkLoad，我们很容易将生成好的几百 G，甚至上 T 的 HFile 文件以毫秒级的速度导入 HBase，并能马上进行查询。所以对于历史数据和非实时写入的数据，我们会建议用户通过 BulkLoad 的方式导入数据。

3.2 SnapshotScanMR

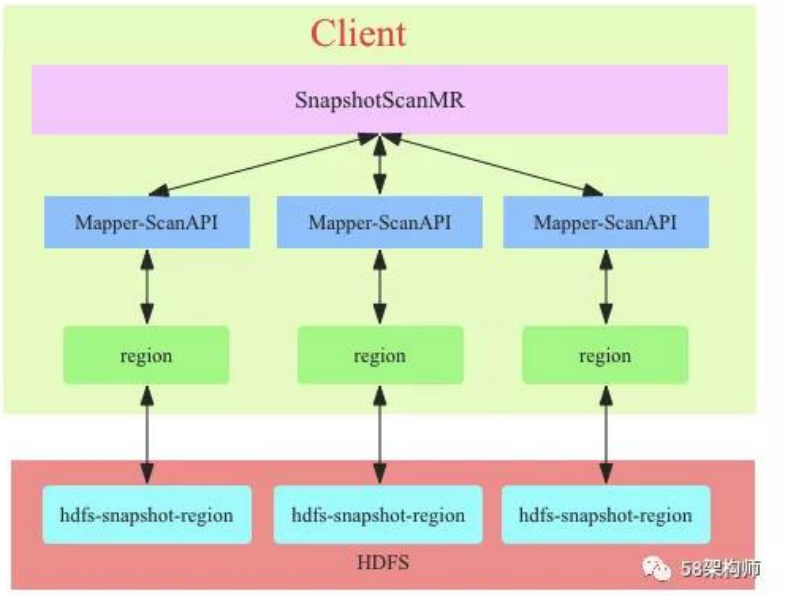
针对全表扫描的应用场景，HBase 提供了两种解决方案，一种是 TableScanMR，另一种就是 SnapshotScanMR，这两种方案都是采用 MR 来并行化对数据进行扫描，但是底层实现原理确是有很大差别，以下会进行对比分析。

TableScanMR 的实现原理图：



TableScanMR 会将 scan 请求根据 HBase 表的 region 分界进行分解，分解成多个 sub-scan(一个 sub-scan 对应一个 map 任务)，每个 sub-scan 内部本质上就是一个 ScanAPI。假如 scan 是全表扫描，那这张表有多少 region，就会将这个 scan 分解成多个 sub-scan，每个 sub-scan 的 startkey 和 stopkey 就是 region 的 startkey 和 stopkey。这种方式只是简单的将 scan 操作并行化了，数据读取链路和直接 scan 没有本质区别，都需要通过 RS 来读取数据。

SnapshotScanMR 的实现原理图：



SnapshotScanMR 总体来看和 TableScanMR 工作流程基本一致，不过 SnapshotScanMR 的实现依赖于 HBase 的 snapshot，通过 shapshot 的元数据信息，SnapshotScanMR 可以很容易知道当前全表扫描要访问那些 HFile，以及这些 HFile 的 HDFS 路径，所以 SnapshotScanMR 构造的 sub-scan 可以绕过 RS，直接借用 Region 中的扫描机制直接扫描 HDFS 中数据。

SnapshotScanMR 优势：

1. 避免对其他业务的干扰：SnapshotScanMR 绕过了 RS，避免了全表扫描对其他业务的干扰。
2. 极大的提升了扫描效率：SnapshotScanMR 绕过了 RS，减少了一次网络传输，对应少了一次数据的序列化和反序列化操作；TableScanMR 扫描中 RS 很可能会成为瓶颈，而 SnapshotScanMR 不需要担心这一点。

基于以上的原因，在全部扫描，以及全部数据导出的应用场景中，我们选择了 SnapshotScanMR，并对原生的 SnapshotScanMR 进行了进一步的封装，作为一个通用工具提供给用户。

4. 平台优化

在使用 HBase 的过程中，我们遇到了很多问题和挑战，但最终都一一克服了，以下是我们遇到一部分典型问题及优化：

4.1 CLOSE_WAIT 偏高优化

问题描述 :在一次排查 HBase 问题的时候发现 RS 进程存在大量的 CLOSE_WAIT，最多的达到了 6000+，这个问题虽然还没有直接导致 RS 挂掉，但是也确实是个不小的隐患。

从 socket 的角度分析产生 CLOSE_WAIT 的原因：对方主动关闭连接或者网络异常导致连接中断，这时我方的状态会变成 CLOSE_WAIT，此时我方要调用 close() 来使得连接正确关闭，否则 CLOSE_WAIT 会一直存在。

对应到咱们这个问题，其实就是用户通过 RS 访问 DataNode（端口 50010）的数据，DataNode 端已经主动关闭 Socket 了，但是 RS 端没有关闭，所以要解决的问题就是 RS 关闭 Socket 连接的问题。

解决办法：社区对该问题的讨论见 HBASE-9393。该问题的修复依赖 HDFS-7694，我们的 Hadoop 版本是 hadoop2.6.0-cdh5.4.4，已经集成了 HDFS-7694 的内容。

HBASE-9393 的核心思想是通过 HDFS API 关闭 HBase 两个地方打开的 Socket：

RS 打开 HFile 读取元数据信息(flush、bulkload、move、balance 时)后关闭 Socket；

每次执行完成用户 scan 操作后关闭 Socket。

优化效果：CLOSE_WAIT 数量降为 10 左右

4.2 DN 慢盘导致 RS 阻塞优化

问题描述：由于集群某个磁盘出现坏道（没有完全坏，表现为读写慢，disk.io.util 为 100%），导致 RS 所有 handler 线程因为写 WAL 失败而被阻塞，无法对外提供服务，严重影响了用户读写数据体验。

最后分析发现，RS 写 WAL 时由于 DN 节点出现磁盘坏道（表现为 disk.io.util 为长时间处于 100%），导致写 WAL 的 pipeline 抛出异常并误将正常 DN 节点标记

为 bad 节点，而恢复 pipeline 时使用 bad 节点进行数据块 transfer，导致 pipeline 恢复失败，最终 RS 的所有写请求都阻塞到 WAL 的 sync 线程上，RS 由于没有可用的 handler 线程，也就无法对外提供服务了。

解决办法：RS 配置 RPC 读写分离：避免由于写阻塞所有 handler 线程，影响到读请求；

pipeline 恢复失败解决：社区已有该问题的讨论，见并 HDFS-9178，不过因为 HDFS 的 pipeline 过程非常复杂，HDFS-9178 能否解决该问题需要进一步验证。

4.3 Compact 占用 Region 读锁优化

问题描述：某次有一个业务执行 BulkLoad 操作批量导入上 T 的数据到 HBase 表时，RS 端报 BulkLoad 操作获取 Region 级写锁出现超时异常：failed to get a lock in 60000 ms，当时该表并没有进行读写操作，最终定位到是该时间段内这个业务的表正在进行 compact 操作，在我们的 HBase 版本中，执行 compact 时会获取 Region 级的读锁，而且会长时间占用，所有导致 BulkLoad 获取写锁超时了。

解决办法：Compact 时不持有 Region 读锁，社区对该问题的讨论见 HBASE-14575。

4.4 HTablePool 问题优化

问题描述：我们的 SCF 服务最初是基于 HTablePool API 开发的，SCF 服务在运行一段时间后经常会出现 JVM 堆内存暴增而触发 FGC 的情况，分析发现 HTablePool 已经是标记为已废弃，原因是通过 HTablePool 的获取 Table 对象，会创建单独的线程池，而且线程个数没有限制，导致请求量大时，线程数会暴增。

解决办法：最后我们换成了官方推荐的 API，通过 Connection 获取 Table，这种方式 Connection 内部的线程池可以在在所有表中共享，而且线程数是可配置的。

4.5 其他优化

BlockCache 启用 BuckCache；Compact 限流优化等。

5. 总结

本文从多租户支持、数据读写接口、数据导入导出和平台优化四个方面讲解了 HBase 相关的平台建设工作。HBase 作为一个开源的平台，有着非常丰富的生态系统。在 HBase 平台基础之上，我们持续不断地引入了各种新的能力，包括 OLAP、图数据库、时序数据库和 SQL on HBase 等，这些我们将在 58HBase 平台实践和应用的后续篇章中进一步介绍。

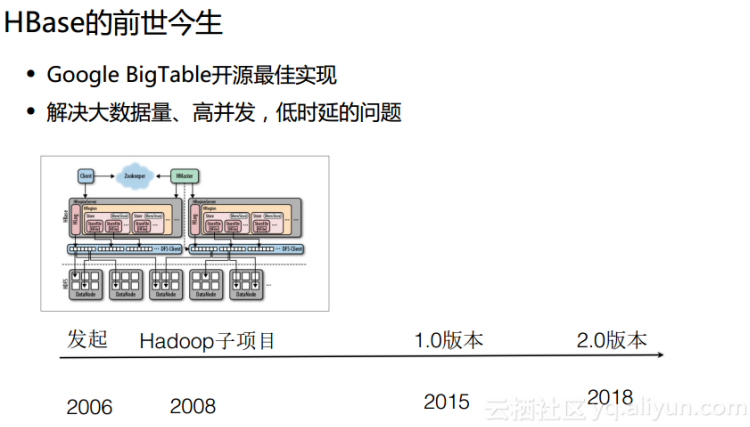
欢迎关注 58 架构师公众号

八年磨一剑，重新定义 HBase——HBase 2.0&阿里云 HBase 解读

1. 八年磨一剑

1.1 HBase 的前世今生

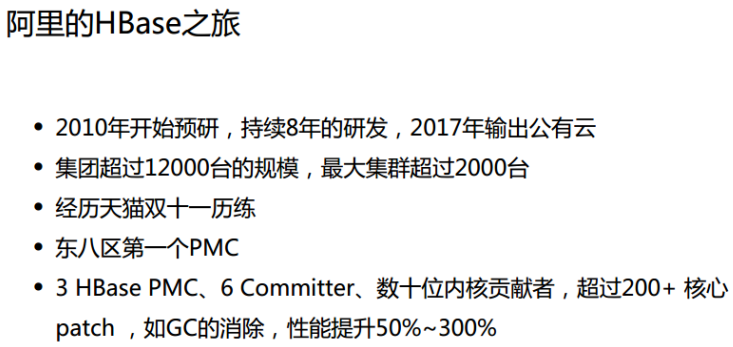
关系型数据库的发展已经经历了 40 多年的历史了，而 HBase 以及大数据这套东西的历史大概从 2006 年被认为是大数据的发起时期到现在，也就是 13 年左右而已。那么，为什么会出现 HBase 以及 Hadoop 整体生态链的这些内容呢？这是因为在大数据时代，传统数据库需要面对很多挑战，出现了数据量增多、业务复杂度提升、非结构化数据和结构化数据并存等诸多问题。这些问题所带来的最直接的就是成本挑战，因此特别需要价格低廉的数据库来解决问题。



这也就是 Google 提出 BigTable 开源最佳实现的原因。Google 是全球最大的搜索引擎，当他们发现出现的存储成本问题之后，通过内部研究就发出来关于 BigTable 的这篇论文，而大概在 2006 年的时候也就发起了 HBase 这个项目，并且在两年之后其就成为 Hadoop 的子项目，经过了十几年的发展，目前演变到了 2.0 版本。HBase 能够帮助我们以低成本解决大数据量、高并发、低时延的问题，并且保证了低成本的存储。

1.2 阿里的 HBase 之旅

为何叫做“八年磨一剑”呢？这其实与阿里巴巴对于 HBase 的研发历程是紧密相关的。在 2010 年，HBase 正式成为了 Apache 的顶级项目，与此同时阿里巴巴内部的业务也达到了瓶颈期，因此在 2010 年阿里巴巴开始对于 HBase 进行预研，经过了持续 8 年的研发，在 2017 年的时候输出到阿里云上，并将 HBase 的能力提供给广大的用户。其实，在阿里集团内部已经有了超过 12000 台的 HBase 服务器规模，而最大集群也超过了 2000 台，这在世界上都是数一数二的，并且也经过了天猫“双 11”的历练。



阿里投入了很多资源和人力来研发 HBase，所以开源社区也给予了非常积极的回馈。目前第一个东八区的 PMC 就诞生在阿里云，而整个阿里集团内有 3 个 HBase PMC、6 个 Committer 以及几十位核心贡献者，并且共享了 200 多个核心 patch。此外，阿里云的 HBase 版本相比于开源版本在很多方面也有极大的提升。

1.3 HBase 适合的场景和问题

(1) 关系型数据库与 HBase 的区别

HBase 等 NoSQL 出现的原因是传统的关系型数据库在面对大数据量、高业务复杂度以及高成本的挑战时，无法对于底层进行优化和改进。如下图所示的表格能够帮助大家对比关系型数据库与 HBase 的主要区别。

关系型数据和HBase区别

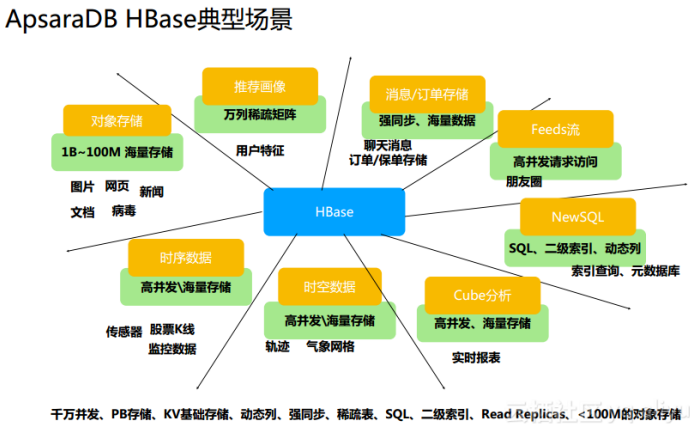
对比	关系型数据库	HBase
应用场景	结构化数据 交易场景 （典型业务：银行交易）	兼容结构化/非结构数据 实时业务+OLAP分析+大规模存储 典型场景：车联网、轨迹、用户画像、气象、朋友圈等
扩展性	单表不超过千列 GB~TB级别	单表百亿行，百万列 数量200G~10P
高可靠&高可用	传统备份+DBA人肉	分布式技术 （高可靠：三副本；高可用：zookeeper）
性能	1w~10w级别	高并发&高吞吐（1w - 5000w）
成本	成本高 SSD/RDMA/高性能CPU	成本低 本地盘/普通CPU

兼容结构化/非结构化、大数量、高并发、低时延、低成本的数据库

关系型数据的主要应用场景基本都是交易类场景，而 HBase 所代表的的非关系型数据库所需要解决的就是兼容结构化和非结构化的数据，解决交易场景之外的实时业务或者 OLAP 分析以及大规模存储。而在可扩展性上，关系型数据库单表不超过千列，一般在 GB~TB 级别，而对于 HBase 而言，这样的数据量就不算什么挑战了，一张表会轻松突破百亿行和百万列，HBase 比较适合 200G 到 10P 数据量级别。在性能方面也能体现出关系型数据和 HBase 最大的区别，对于关系型数据库而言，10W 级别的性能就已经很强了，HBase 能力能够达到 5000 万并发量，其核心需要解决的就是高并发和低吞吐。在成本方面，传统关系型数据库需要使用高性能硬件，随着也带来了成本问题，而 HBase 则是选择了另外一条路，通过本地盘和普通 CPU 实现，其核心的存储结构不再是关系型存储结构，而是合并成批量读写，充分地利用硬盘高吞吐的能力来达到低成本。总结而言，HBase 数据库解决的核心问题就是兼容结构化和非结构化的数据、大数据量、高并发、低延时等问题。

(2) ApsaraDB HBase 典型场景

NoSQL 数据库和传统数据的一个较大区别就是传统数据库比较适合所以行业的交易模型，而 HBase 的能力非常聚焦，其适合时序数据、时空数据、Cube 分析、NewSQL、Feeds 流、消息/订单存储、推荐画像、对象存储等 8 个场景。接下来逐一介绍各个典型场景。



时序数据：在如今这个 IoT 的时代，出现了大量的时序数据。因为各种传感器比较多，时序数据需要满足高并发、海量存储等基本要求，除了 IoT 之外，在股票以及监控数据里面也需要用到这样的时序数据。

时空数据：轨迹以及气象网格数据也需要 HBase 的高并发和海量存储能力。

Cube 分析：实时报表以及数据科学家需要进行实时数据分析，这些需要以很低的时延查询出来，并且并发量非常高，这时候就需要用到 Cube 的能力。

NewSQL：对于传统 SQL 所不能解决的一些问题，比如性能瓶颈，这些就需要使用 NewSQL 能力，并且除了能够解决性能问题之外，还会有一定的更新能力。HBase 能够较好地适应这种场景，除了支持 SQL 之外，还支持二级索引、动态增加列，所以很多元数据库也使用了 HBase。

Feeds 流：Feeds 流的典型应用场景就是微信朋友圈以及微博等社交网络，其所需要的就是高并发的请求访问，因为用户数非常多，需要保证一致性体验，HBase 非常适合这样的场景。

消息/订单存储：订单数量是非常多的，所以需要强同步，并且可靠性不能丢，对于聊天消息而言也是一样的，这就会用到 HBase 的强一致性同步以及大数据存储能力。

推荐画像：推荐画像在用户特征里面使用的比较多，一般而言在做画像时对于用户会勾画很多特征，数据表中的每一列就可以放一个特征，而随着不断深入地挖掘，特征就会越来越多，而传统关系型数据库首先不能够存储太多列，超过 1000 列就遇到瓶颈了，而在真正进行存储的时候可能需要百万列。其次，对于传统关

系型数据库而言，如果某列存在空值，依然占据空间，而 HBase 不仅可以动态地增加列，而且如果某列为空就不会占据空间，所以非常适合这样的场景。

对象存储：通常而言对象存储最可能用到的就是 OSS，但是 OSS 比较适合高并发地写，但是并不适合高并发地读，但是还有很多数据比如图片、网页、文档、新闻等，除了需要能够写入之外，还需要提供高并发地查询能力的场景下，就比较适合使用 HBase 作为前置数据存储。而如果随着时间的迁移，一些数据的重要性降低了，那就可以通过 HBase 本身的能力将数据迁移到 OSS 里面，作为温数据或者冷数据的归档。

上面大致分享了 HBase 的整个发展历程以及 HBase 的适合场景。总结而言，就是阿里巴巴经过了 8 年的研发将自己改进的 HBase 版本能够提供给广大的用户，这就叫做 8 年磨一剑。这首先说明了阿里巴巴有很强的技术实力，其次说明了 HBase 有它自己非常适合的场景，比如高并发、低时延以及低成本需求的场景。

2. 重新定义 HBase

大家都知道开源软件在稳定性以及各方面的能力上都往往不能够达到企业实际应用的要求。虽然这些开源软件的核心能力非常强大，但是当在企业中应用的时候却是比较困难的。而阿里巴巴在 HBase 方面做了很多的工作，也经过内部的使用提升了 HBase 的能力，使其能够更好地适应企业的应用，并且针对不同的场景提供了不同的产品形态。

2.1 HBase2.0 解读

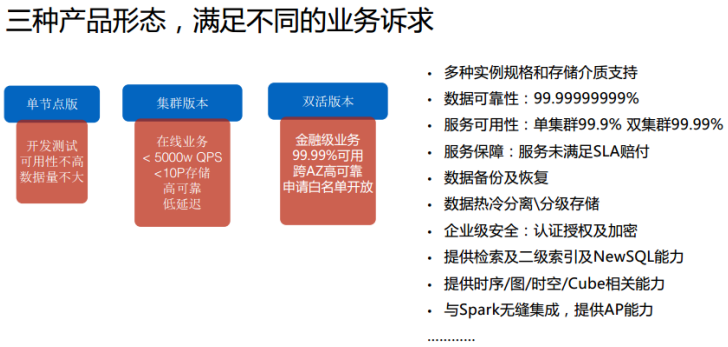
HBase 2.0 版本是本次重点发布的版本。实际上，社区在 2014 年 6 月开始迭代，经过了 4 年的时间，终于在 2018 年 4 月 30 日正式发布。而阿里巴巴也立即将原来的一些能力反向地融合到 HBase 最新的版本中，并且经过了稳定性和性能测试，这个版本将会在 6 月 6 日正式发布公测。

HBase 2.0 版本经过四年时间的研发，其在架构上也发生了较大的变化，在性能以及稳定性上也有了较大的提升。总结而言，产生了两个最有价值的场景，其中一个是大容量、高并发、低延时的写场景，相比于 1.X 版本，HBase 2.0 版本提

升了稳定性，将内存全部放在堆外进行管理，不再完全依赖 JVM，这是因为 JVM 存在一些始终难于解决的问题。此外 HBase 2.0 版本还提升了性能，对于资源管理器进行了优化，解决了性能毛刺问题。此外，性能的增强还体现在了时延的提升上。HBase 处理时延可以稳定在 50ms 以内。这个对 HBase 来说拓宽了一个大容量推荐场景；这个也是其他目前数据库引擎不具备的能力。例如金融，社交朋友圈等行业有广泛的诉求。另外一个场景就是高性能对象存储场景，以前包括阿里集团更多的是适合结构化的数据的存储。HBase 2.0 版本支持高效对象存储，能高效地存储那些 100k~10M 中等大小的对象。有了这个能力之后，就相当于在对象存储能力上有了非常强的提升，可以包装出一种新的产品形态或者场景，解决不满意对象存储性能的，对性能有一定要求的大容量对象存储的需求。这样的能力预计在传媒，教育，企业办公等领域有广泛的诉求。HBase 2.0 不仅能够提供很强大的写能力之外，还能够提供很强大的读能力。

2.2 重新定义产品形态

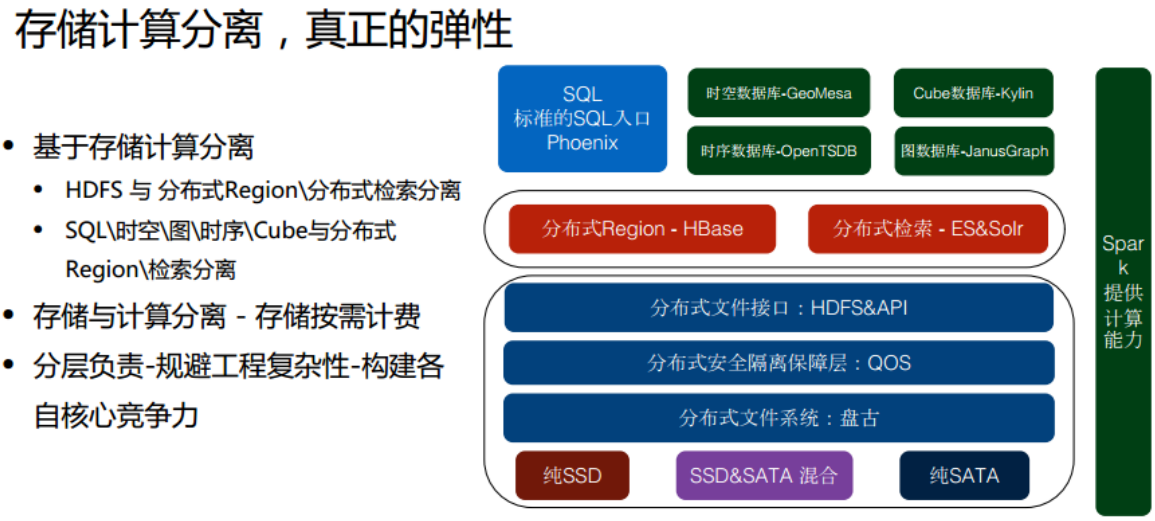
HBase 作为开源软件，并没有考虑很多的企业级能力，而阿里云的 HBase 在开源软件的基础之上进行较大的创新和优化。首先针对于不同的业务场景，提供了不同产品形态的 HBase。在开发测试环境下，可用性要求不高，数据量也不大，而需要比较低的成本，这时候就可以使用单节点版本。而针对于在线业务，QPS 在 5000 万以内，存储在 10P 以内，需要高可靠、低时延的处理能力，阿里云优先推荐集群版本。还有第三种双活版本，在很多企业的金融级业务里面，可用性要求很高，也需要跨 AZ 的高可靠，需要双活版本，一个集群除了故障，另外一个集群能够实时地进行接管。



除了提供了以上三种不同的产品形态之外，阿里云 HBase 还在可用性、数据冷热分离、写安全以及二级索引等能力上做了较大的提升。

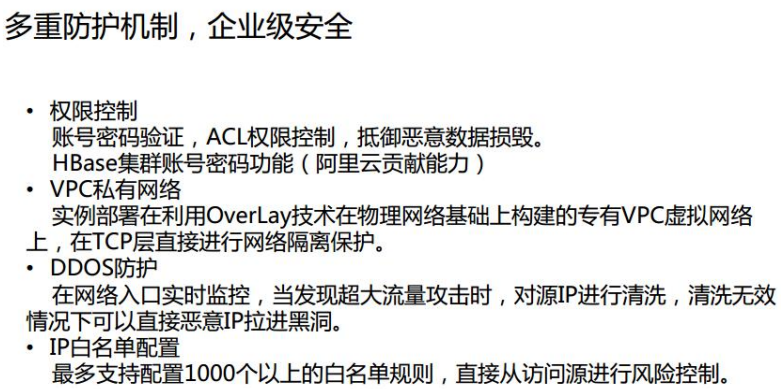
2.3 重新定义 HBase 能力

(1) 存储计算分离，真正的弹性



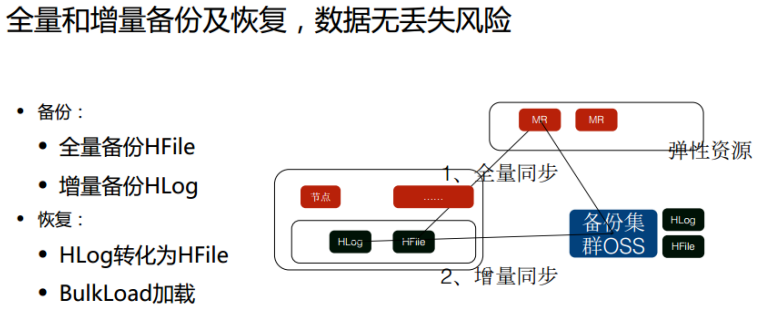
存储计算分离这个特性是非常有价值的能力，虽然在现在企业往往也能够自己搭建 HBase 服务器，但是却很难实现存储计算分离的能力。这是因为业务会飞速发展变化，所以难以在最初规划时确定究竟多少资源是足够的，后续扩充就会变得比较麻烦，也会造成资源的浪费和比较高的成本。在阿里云上做了存储于计算分离，使得存储和计算可以分开进行计费，可以单独扩充存储或者计算资源，这极大地有利于企业业务的灵活变化，同样也极大地降低了成本。

(2) 多重防护机制，企业级安全



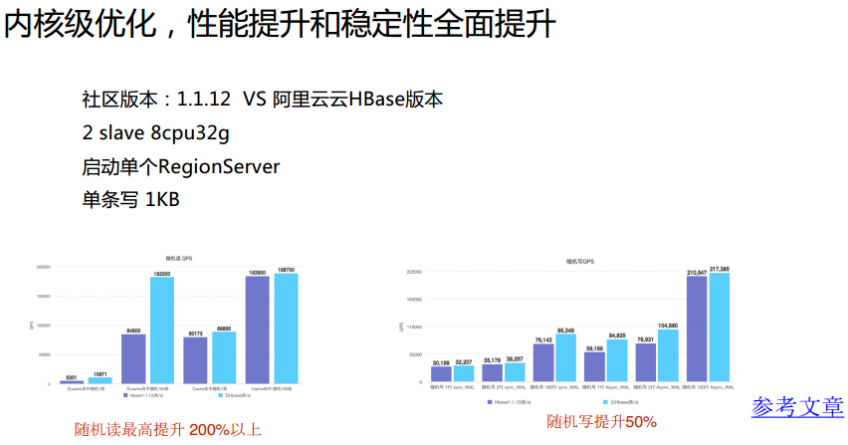
大家都知道开源版本的 HBase 基本上没有安全能力，完全属于“裸奔”状态。这使得企业数据的安全性无法得到有效的保证，因此阿里云在 HBase 的安全方面也做了大量的工作。比如权限控制管理上，提供了账号密码验证、ACL 权限控制以及抵御恶意数据损毁上，这些方面阿里云都贡献了很大的能力。而在 VPC 隔离、防 DDOS 攻击以及 IP 白名单配置上，阿里云也做了非常多的事情，通过多重机制保证用户的数据安全以及可靠性。

（3）全量和增量备份以及恢复，数据无丢失风险



对于企业而言，最具有价值的就是数据，因此企业所最担心的也就是数据的丢失。借助阿里云 HBase 的全量和增量备份以及恢复的能力则能够尽量降低数据丢失的风险。首先，阿里云在机器里面有高可靠的能力，其次再通过全量或者增量备份一份数据到对象存储里面来。如果万一出现了数据故障，也能够迅速地将数据恢复回来，不会有数据丢失的风险，这对于 DBA 或者企业而言，能够有效地对于数据进行保障。

（4）内核级优化，性能和稳定性全面提升



阿里云具备可以称得上东半球对于 HBase 最强的研发实力，这也是能够非常深刻地体现到阿里云 HBase 产品里面的一点。如上图所示的是用阿里云 HBase 与社区的 HBase 的一个版本进行的对比情况，可以看到随机读最高可以提升 200%以上，而随机写提升 50%以上。此外，在稳定性方面还实现了读写分离机制，能够确保读写不会冲突，进而保证稳定性。

(5) 重新定义运维能力，客户基本免运维

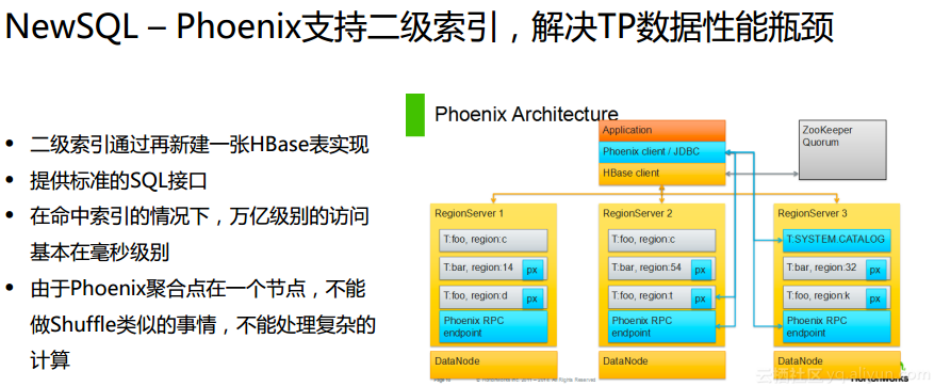
大家都知道，21 世纪最具有价值的就是人才，HBase 的确非常好，但是其使用起来的难度也非常高，因为其技术难度的门槛放在那里，如果没有能力较强的人才，很难帮助企业恢复数据，并且实时地保障业务的平稳运行。而阿里云所提供的 HBase 版本基本上能够实现客户免运维。在阿里云提供的运维自动化里面，专家会提供 24 小时在线的服务，可以帮助客户搞定一切运维问题。

3.生态和案例

其实企业在选择 HBase 不仅仅是看重的是其高并发、低时延的处理能力，还看重了其背后的大数据生态，因为当有了这样的大数据生态之后将可以做很多的事情。

3.1 使用场景和生态

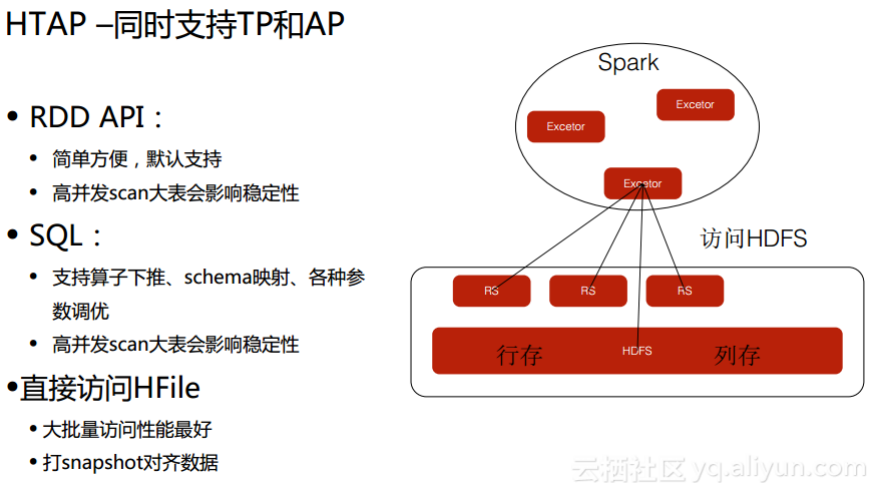
(1) NewSQL-Phoenix 支持二级索引，解决 TP 数据性能瓶颈



图片来自互联网

在 NewSQL 方面，阿里云 HBase 默认搭配了 Phoenix 组件，这样就可以用标准的 SQL 接口去访问数据。此外，相比于开源版本，阿里云的 HBase 在扩展性方面也有极大的提升，在这里面可以实现更新、高并发的读写。并且 Phoenix 还支持二级索引能力，进而可以进一步提升性能。

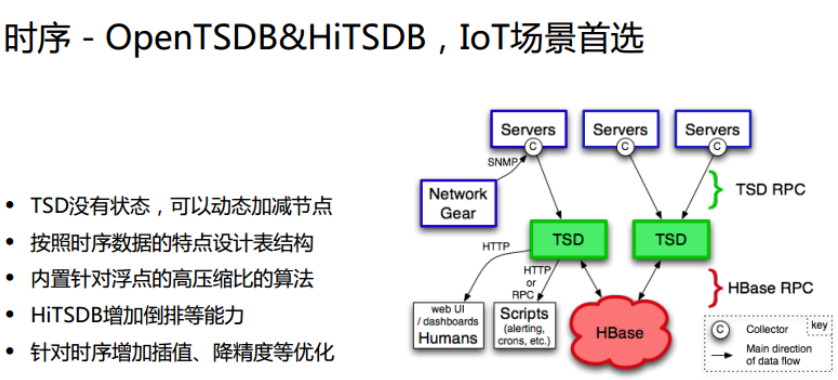
(2) HTAP-同时支持 TP 和 AP



当需要大数据分析能力时，HBase 就需要结合大数据生态中一个非常重要的组件——Spark。Spark 可以通过三种方式访问数据，而每种方式都有自己不同的特点，比如直接通过 API 的方式，比较简单，但是只适合基本的使用；其次可以使用

Spark SQL，其自带了算子下推、schema 映射以及各种参数调优的能力；第三种就是在打批量访问的时候可以直接访问 HFile 进行分析。HBase 搭配 Spark 可以实现混合的 TP 和 AP 能力。

(3) 时序-OpenTSDB&HiTSDB，IoT 场景首选



图片来自互联网

HBase 本身自带了开源组件 OpenTSDB，直接搭配就可以满足时序需求。

(4) 时空-GeoMesa

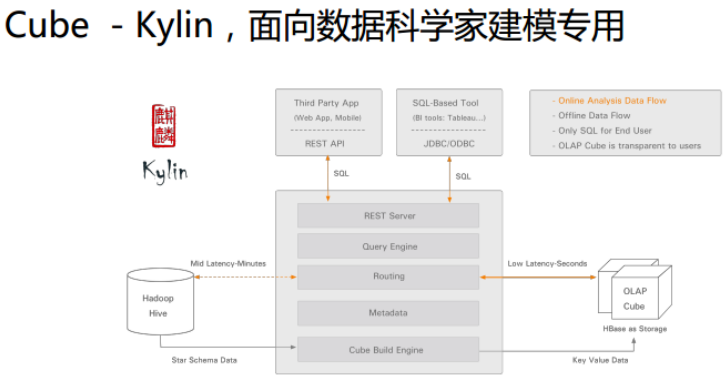
来自官网：http://www.geomesa.org/

HBase 结合 GeoMesa 的能力就可以将三维的精度、维度以及时间进行降维，得到一维的数据并存储到 HBase 里面。

(5) 图数据库-HGraphDB，关系分析，风控场景必备

图数据库虽然不常见，但是在很多行业中使用的也非常多，比如关系分析以及风控场景。

(6) Cube-Kylin，面向数据科学家建模专用

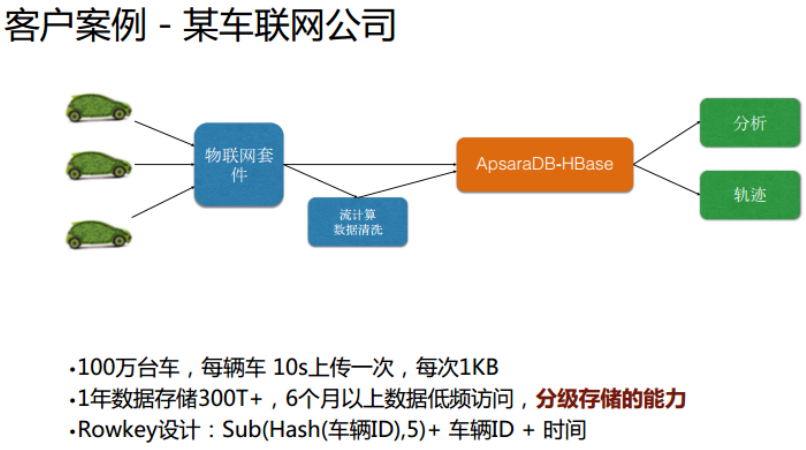


图片来自互联网

在大数据时代，数据的价值的挖掘需要数据科学家来实现。数据科学家在进行数据建模之前需要将数据拿出来，反复地进行分析，所以需要很多随机的条件下进行，想要得到低时延的反馈就需要建立 Cube 来实现。

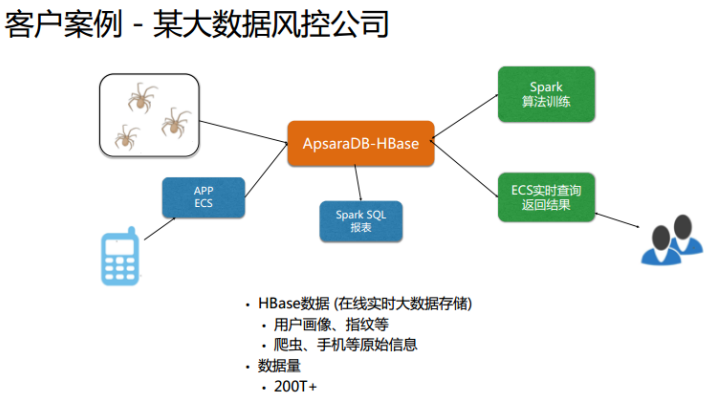
3.2 实际客户案例

(1) 客户案例-某车联网公司



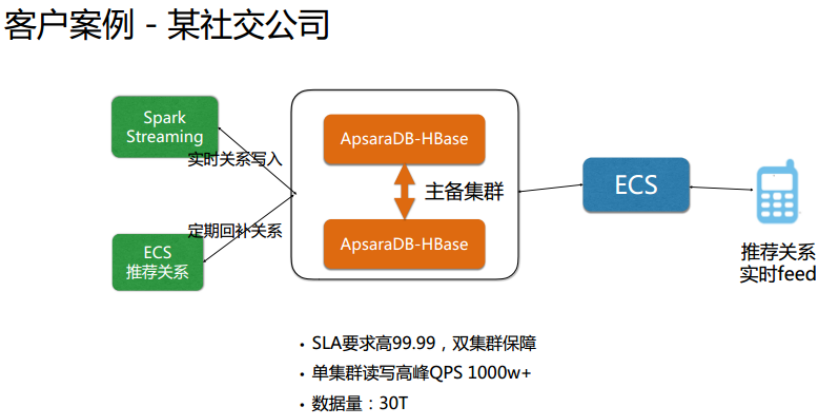
对于现在的很多互联网公司而言，往往都需要将数据高并发地写入进来，但是超期之后数据的价值就会降低，但是因为法律法规等方面政策的要求，这些数据不能被删除。这样就需要分级存储的能力，能够以很低的成本解决大量数据的存储问题。

(2) 客户案例-某大数据风控公司



大数据风控公司通过爬虫或者 ECS 上的 APP 记录传感器等将数据收集上来，并对于用户进行画像，这就用到了 HBase 的画像能力以及稀疏矩阵存储能力。

(3) 客户案例-某社交公司



对于社交网络而言，比如一个帖子需要瞬间分发给三百万或者五百万用户还需要保证在几十毫秒之内，这样的能力目前只有 HBase 才能做到。案例中的用户使用了双集群，最高有四个 9 的可靠性，并且其 QPS 能够达到 1 千万以上，能够瞬间将 Feed 流推到所有用户上面去。

(4) 客户案例-某基金公司

客户案例 - 某基金公司

对于某基金公司而言，单张表有一万亿以上数据，而对于传统关系型数据库而言，有个 1000 数据已经非常大了，而像这种百 TB 级别的数据的查询以及少量的更新只能使用 HBase+Phoenix 搞定。

(5) 客户案例–某公司报表系统

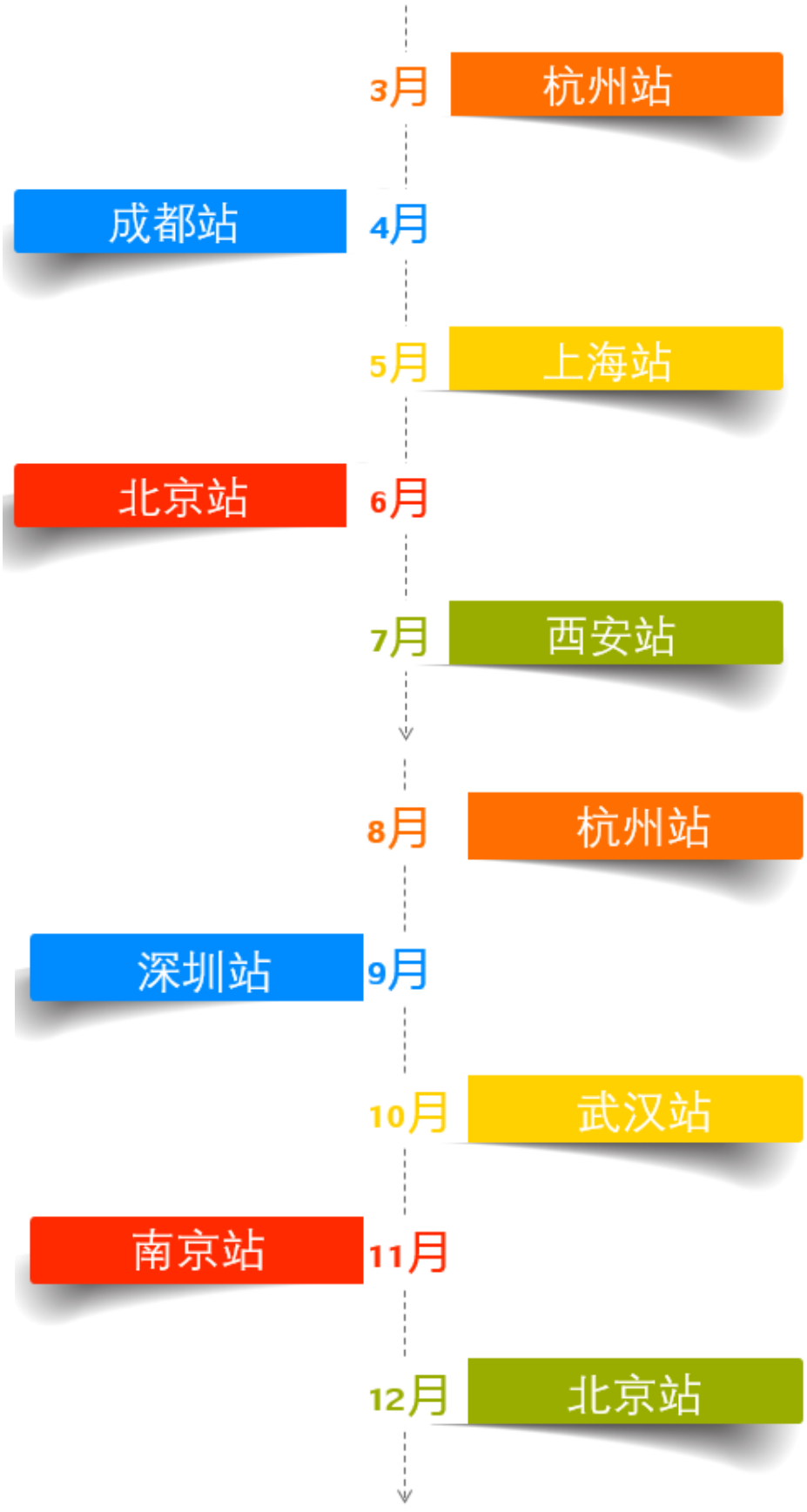
客户案例 - 某公司报表系统

某公司的报表系统通过离线提前接好 Cube 实现了数据的实时更新和查询。

4.总结

在本文中，首先介绍了阿里巴巴集团对于 HBase 的八年研发历程。第二部分，分享了 HBase 作为一款开源软件在很多企业级软件功能上的不足，所以阿里云重新定义了产品形态，并增强了 HBase 产品能力，使得企业能够更快更好地使用 HBase 服务。最后，选择了 HBase 所代表的不仅仅是 HBase 本身的能力，而是其背后的整个大数据生态，在未来进行业务能力扩展上也是非常有帮助的。而相比于传统关系型数据库 40 几年的发展历程，HBase 的短短十几年的发展历史还是比较短的，在使用门槛上相对比较高，因此阿里云也借助 HBase2.0 版本发布的契机，联合了一些国内顶尖的公司，比如滴滴和小米，大家一起深度地解读 HBase 的能力和使用场景，也希望大家持续关注后续的相关解读。

中国 HBase 技术社区 2019 年全国 meetup 计划



2018 年由中国 HBase 技术社区举办的 HBase Meetup 现场

中国 HBase 技术社区微信公众号

HBase+Spark 钉钉技术社区群