HBase 调研报告

目 录

[HBase 调研报告 1](#_Toc14302103)

[目 录 2](#_Toc14302104)

[1 介绍 5](#_Toc14302105)

[2 HBase理论基础 5](#_Toc14302106)

[2.1 HBase应用场景 5](#_Toc14302107)

[2.2 数据模型 5](#_Toc14302108)

[2.2.1 概念（逻辑）视图 6](#_Toc14302109)

[2.2.2 物理视图 6](#_Toc14302110)

[2.3 物理模型 7](#_Toc14302111)

[2.4 系统架构 8](#_Toc14302112)

[2.5 HBase的容错 9](#_Toc14302113)

[2.6 特殊的表 9](#_Toc14302114)

[2.7 读取流程（0.96版本之后(没有.ROOT表)） 10](#_Toc14302115)

[2.8 写入流程 10](#_Toc14302116)

[2.9 memstore 10](#_Toc14302117)

[2.10 HBase Region Flush 11](#_Toc14302118)

[2.11 HBase Crash Recovery 11](#_Toc14302119)

[2.12 HBase的Compaction和Split 11](#_Toc14302120)

[2.12.1 Compaction和Split 11](#_Toc14302121)

[2.12.2 管理Compaction和Spilit 12](#_Toc14302122)

[2.12.3 相关优化 12](#_Toc14302123)

[2.13 HBase 排序 13](#_Toc14302124)

[2.14 HBase 的LSM思想 13](#_Toc14302125)

[2.15 小结 13](#_Toc14302126)

[3 HBase 竞品分析 14](#_Toc14302127)

[3.1 NoSQL数据库 14](#_Toc14302128)

[3.2 主流的NoSQL数据库 15](#_Toc14302129)

[3.2.1 CoucHBase 15](#_Toc14302130)

[3.2.2 MongoDB 16](#_Toc14302131)

[3.2.3 CouchDB 17](#_Toc14302132)

[3.2.4 Redis 18](#_Toc14302133)

[3.2.5 Riak 19](#_Toc14302134)

[3.2.6 Neo4j 20](#_Toc14302135)

[3.2.7 CoucHBase 20](#_Toc14302136)

[3.2.8 HBase 21](#_Toc14302137)

[3.3 HBase直接竞品分析 22](#_Toc14302138)

[3.3.1 HBase 与 Cassandra 23](#_Toc14302139)

[3.3.2 HBase 与 MongDB 25](#_Toc14302140)

[3.4 小结 27](#_Toc14302141)

[4 HBase的周边生态 28](#_Toc14302142)

[4.1 HBase的底层存储—HDFS (Hadoop File System) 28](#_Toc14302143)

[4.2 能配合HBase存储的查询工具 28](#_Toc14302144)

[4.2.1 HBase Shell 28](#_Toc14302145)

[4.2.2 HBase thrift 28](#_Toc14302146)

[4.2.3 Hive 29](#_Toc14302147)

[4.2.4 Solr 29](#_Toc14302148)

[4.2.5 ES 29](#_Toc14302149)

[4.2.6 Phoenix 30](#_Toc14302150)

[4.2.7 Impala 30](#_Toc14302151)

[4.2.8 Spark 31](#_Toc14302152)

[4.2.9 Kylin 32](#_Toc14302153)

[4.3 能配合HBase存储的分析计算框架 33](#_Toc14302154)

[4.3.1 MapReduce 33](#_Toc14302155)

[4.3.2 Spark 33](#_Toc14302156)

[4.3.3 Flink 34](#_Toc14302157)

[4.4 能配合HBase存储的消息队列 34](#_Toc14302158)

[4.4.1 Kafka 34](#_Toc14302159)

[4.4.2 HQueue 35](#_Toc14302160)

[4.5 能配合HBase存储的监控组件 35](#_Toc14302161)

[4.5.1 Ganglia 35](#_Toc14302162)

[4.5.2 Zebbix 36](#_Toc14302163)

[4.5.3 CDH 37](#_Toc14302164)

[4.5.4 Ambari 38](#_Toc14302165)

[4.5.5 Nagios 38](#_Toc14302166)

[4.5.6 OpenTSDB 38](#_Toc14302167)

[4.6 能配合HBase存储的分布式协调组件 40](#_Toc14302168)

[4.6.1 Zookeeper 40](#_Toc14302169)

[4.6.2 ETCD 40](#_Toc14302170)

[4.6.3 Consul 41](#_Toc14302171)

[4.7 其他周边生态组件 41](#_Toc14302172)

[4.7.1 Kerberos安全认证组件 41](#_Toc14302173)

[4.7.2 GeoMesa地理数据处理套件 42](#_Toc14302174)

[4.7.3 JanusGraph图数据库 42](#_Toc14302175)

[4.8 小结 43](#_Toc14302176)

[5 HBase应用场景及案例 44](#_Toc14302177)

[5.1 阿里 44](#_Toc14302178)

[5.1.1 HBase在阿里的使用场景 44](#_Toc14302179)

[5.1.2 阿里对HBase的开发 44](#_Toc14302180)

[5.2 滴滴 46](#_Toc14302181)

[5.2.1 HBase在滴滴的使用场景 47](#_Toc14302182)

[5.2.2 滴滴基于HBase的开发工作 49](#_Toc14302183)

[5.3 58同城 51](#_Toc14302184)

[5.3.1 HBase在58同城的使用场景 51](#_Toc14302185)

[5.3.2 58同城基于HBase的开发 52](#_Toc14302186)

[5.4 小结 54](#_Toc14302187)

# 介绍

HBase是开源的、非关系型分布式数据库（NoSQL），适合随机、实时读写大量数据，参考了Google的BigTable建模，实现的编程语言是Java。作为Apache软件基金会Hadoop项目的一部分，运行于HDFS文件系统之上，可以容错的存储海量稀疏数据。

主要特性：高可靠（WAL机制）、高并发读写、列存储、可伸缩、易构建。

# HBase理论基础

参考文档：

http://abloz.com/HBase/book.html#schema

[https://mapr.com/blog/in-depth-look-HBase-architecture/](https://mapr.com/blog/in-depth-look-hbase-architecture/)

## HBase应用场景

* 搜索引擎数据存储
* 海量数据写入
* 消息中心
* 内容服务系统
* 复杂的大表
* 大批量数据读取

## 数据模型

**rowkey:**

(byte array) 表中的主键，二进制码流，最大64KB，编码过程可能存在数据倾斜，导致集群性能下降。rowkey设计应该尽可能地短，以减少持久化和读缓存开销，最好是8的倍数（系统64位，内存8字节对齐）。

**column family:**

列族，拥有一个名称（string)，包含一个或者多个相关列。

**column:**

属于column family， familyname:columnName， 每条记录可动态添加。

**version number：**

类型为Long，默认值是系统时间戳，可由用户自定义。

**value(cell):**

byte array，{rowkey => {family =>{qualifier =>{version => value }}}}

### 概念（逻辑）视图

概念视图中空白cell在物理上是不存储的，因为根本没有必要存储。

表2- 1 概念（逻辑）视图

|  |  |  |  |
| --- | --- | --- | --- |
| **Row Key** | **Time Stamp** | **ColumnFamily contents** | **ColumnFamily anchor** |
| "com.cnn.www" | t9 |  | anchor:cnnsi.com = "CNN" |
| "com.cnn.www" | t8 |  | anchor:my.look.ca = "CNN.com" |
| "com.cnn.www" | t6 | contents:html = "<html>..." |  |
| "com.cnn.www" | t5 | contents:html = "<html>..." |  |
| "com.cnn.www" | t3 | contents:html = "<html>..." |  |

### 物理视图

物理视图即是实际在磁盘上存储的格式。

表2- 2 物理视图

|  |  |  |
| --- | --- | --- |
| **Row Key** | **Time Stamp** | **Column Family anchor** |
| "com.cnn.www" | t9 | anchor:cnnsi.com = "CNN" |
| "com.cnn.www" | t8 | anchor:my.look.ca = "CNN.com" |

|  |  |  |
| --- | --- | --- |
| **Row Key** | **Time Stamp** | **ColumnFamily "contents:"** |
| "com.cnn.www" | t6 | contents:html = "<html>..." |
| "com.cnn.www" | t5 | contents:html = "<html>..." |
| "com.cnn.www" | t3 | contents:html = "<html>..." |

## 物理模型

HBase一张表是由一个或者多个HRegion组成。

各条记录之间按照rowkey的字典来排序。

Region按照大小分割，每个表初始只有一个Region，随着数据的不断插入，Region不断增大，当增大到一个阈值的时候，HRegion就会等分成两个HRegion。当table中的数据不断增多，以此方式进行分裂增加。

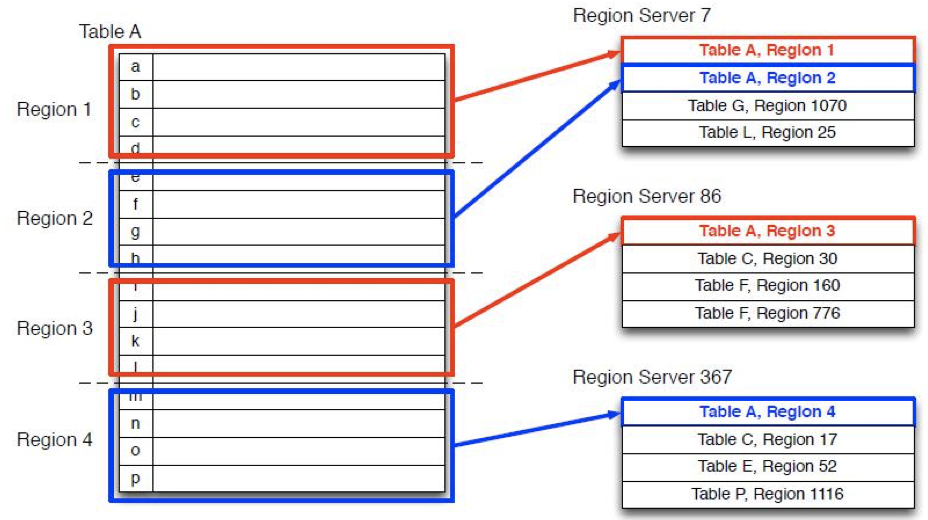


图2- 1 物理存储模型图

具体解释如下：

* table -> HTable
  + 按照rowkey范围来分的Region -> HRegion -> HRegion Servers
    - HRegion按照column family -> 多个HStore
      * + HStore -> memstore + HFile(已经按照kv排好序的

HFile -> HDFS

HRegion是分布式存储和负载均衡的最小单元。但是不是存储的最小单元。最小单元就表示不同的HRegion可以分不到不同HRegion Servers上。但是一个HRegion不会被拆分到多个server上。

## 系统架构

1. **Client**

访问HBase的接口，并维护Cache加速Region Server的访问。

1. **Master**

负载均衡，分配Region到RegionServer。通过从Zookeeper那里得到的通知来监控RegionServer。管理用户对Table的CURD操作。

1. **RegionServer**

维护Region，负责Region的IO。每个RegionServer能负载1000个Regions。

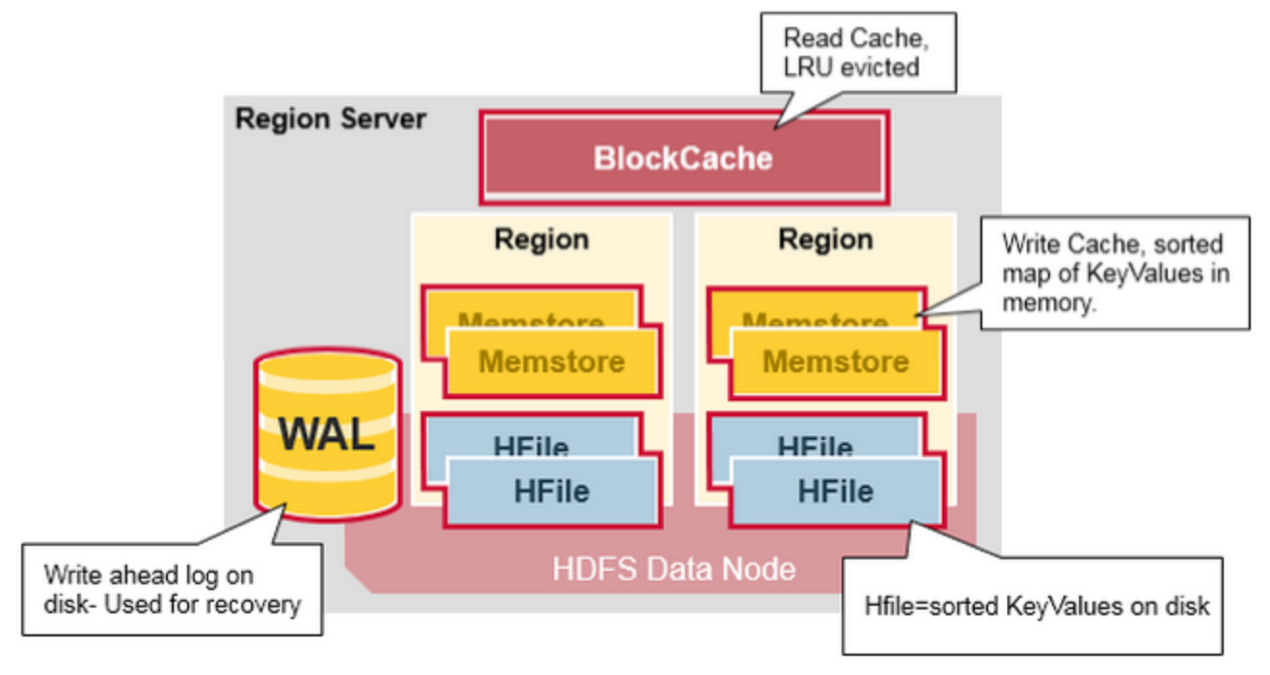


图2- 2 RegionServer

1. **Zookeeper**

保证集群中只有一个HMaster。

存储所有的Region的入口。

实时监控RegionServer的上下线消息，并通知HMaster。

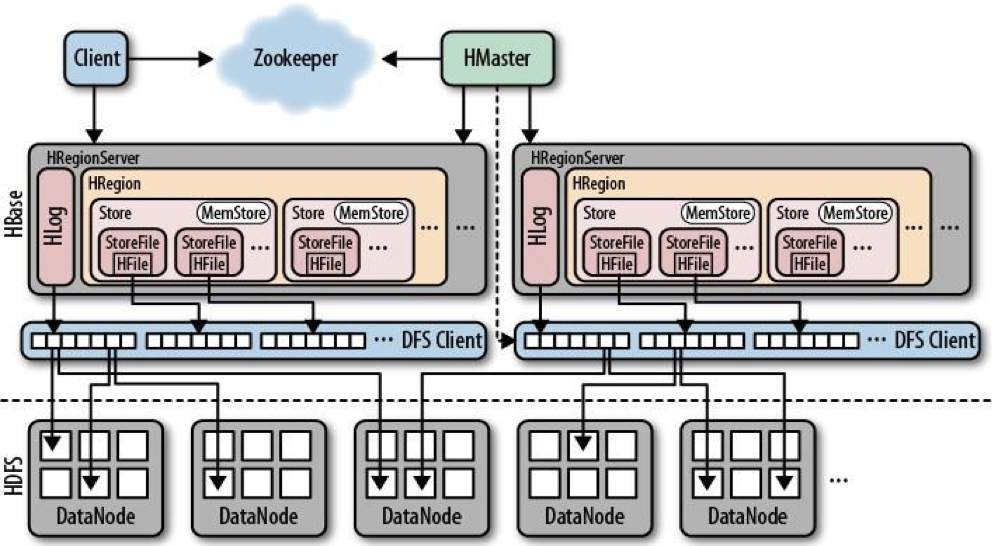


图2- 3 Zookeeper在HBase集群

## HBase的容错

Zookeeper协调集群所有节点的共享信息，在HMaster和HRegionServer连接到Zookeeper后创建ephemeral节点，并使用heartbeat机制维持这个节点的存货状态，如果某个ephemeral节点失效，则HMaster会收到通知，并做相应的处理。除了HDFS存储信息，HBase还在Zookeeper中存储信息

1. **master的容错：**

Zookeeper重新选择一个新的master。

无master的时候，数据可以读取，其他操作不行。

有master的时候，数据才能进行Region的切分、负载均衡等。

1. **RegionServer容错**

定时向Zookeeper回报心跳，如果约定时间内未出现心跳，master会将RegionServer上的Region分配到其他RegionServer上，失效服务器上“预写”日志由主服务器进行分割并派送给新的RegionSever。

1. **Zookeeper容错**

奇数个（大于等于3）配置。

1. **WAL(write-ahead-log)**

WAL，HBase的RegionServer在处理数据插入和删除的过程中用来记录操作内容的一种日志。在每次put、delete等一条记录时，首先将其数据写入到Regionsever对应的hlog文件的过程。客户端向RegionServer端提交数据的时候，会写WAL日志，只有当WAL日志写成功之后，客户端才会被告诉提交数据成功，如果写WAL失败会告知客户的提交失败。

## 特殊的表

.META: 记录了用户表Region信息，.META可以有多个Region。

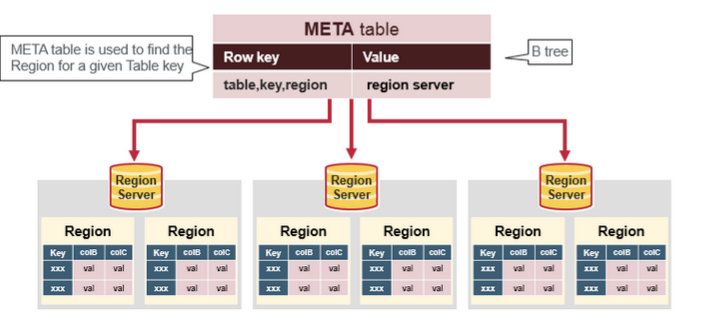


图2- 4 .META表

## 读取流程（0.96版本之后(没有.ROOT表)）

1．从Zookeeper（/HBase/meta-Region-serveer)获取HBase:meta的位置（HRegionServer的位置），缓存该位置的信息。

2. HRegionServer中查询到用户表对应请求rowkey所在的RegionServer，缓存该位置。

3. 从查询到RegionServer中读取row。在一个RegionServer上所有的Region都共享一个hlog，一次数据的提交是先写WAL，写入成功后，再写memstore。当memstore值到达一定阈值，就会形成一个个storefile（HFile的格式封装，本质上是以HFile的形式来存储的）。

## 写入流程

1. client发起put请求，首先从HBase:meta中查出数据的目的地HRegionServer(首次和后面有区别，主要在于获得.META表的方式不同，首次得访问HMaster，后面就客户端可以cache住.META表)。

2. client将put请求发送个对应的HRegionServer。

3. 在HRegionServer中，首先将put操作写入WAL。

4. HRegionServer根据put的tablename和rowkey找到对应的HRegion，根据column family 找到对应的HStore，并将put写入到HStore的memstore中。此时写入成功。(memstore是一个写缓存，每个column family 有一个自己的memstore)。

## memstore

memstore是一个in memory sorted buffer，在每个HStore中都有一个memstore，即它是一个HRegion的一个columnfamily对应的一个实例。排列顺序为rowkey\column family\colunm\timestamp的倒序。

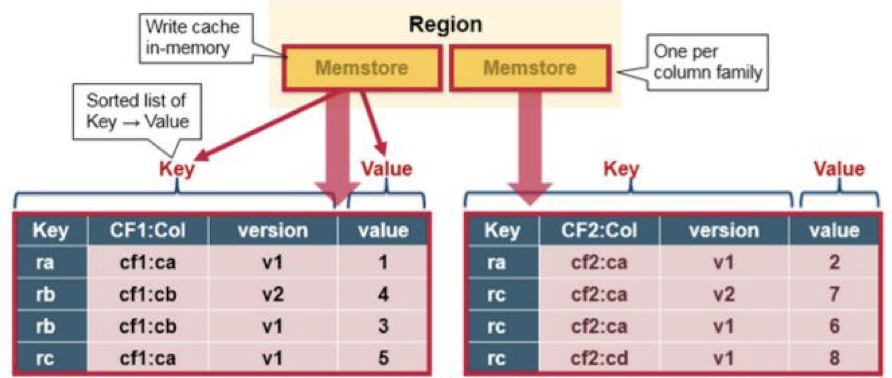


图2- 5 memstore

## HBase Region Flush

当memstore存满了之后，整个会转换为一个新的HFile写到HDFS上。每个列族会有多个HFiles。

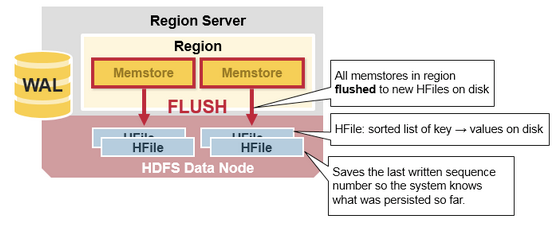


图2- 6 flush机制

## HBase Crash Recovery

为了恢复未刷新到磁盘的崩溃区域服务器的memstore编辑。HMaster将属于崩溃区域服务器的WAL拆分为单独的文件，并将这些文件存储在新的区域服务器的数据节点中。然后，每个Region Server从相应的拆分WAL重放WAL，以重建该区域的memstore。

## HBase的Compaction和Split

提出这两个机制的原因：随着写入次数的不断增多，flush次数的不断增多，Hfile文件越来越多，所以HBase需要对这些文件进行合并。更大的Region可以使得集群上Region的总数量较少，进而使得集群更加流畅。

在HBase中，数据在更新时首先写入WAL 日志(HLog)和内存(MemStore)中，MemStore中的数据是排序的，当MemStore累计到一定阈值时，就会创建一个新的MemStore，并且将老的MemStore添加到flush队列，由单独的线程flush到磁盘上，成为一个StoreFile。于此同时， 系统会在Zookeeper中记录一个redo point，表示这个时刻之前的变更已经持久化了(minor compact)。

### Compaction和Split

Compaction会从一个Region的一个store中选择一些HFile文件来进行合并。合并方法，先读取keyvalues，再按照由小到大的顺序排序后，写入一个新的文件中。之后，这个新生成的文件就会取代之前的文件。

以下是两种方式：

**1. Minor Compaction**：选取那些小的、相邻的StoreFile将他们合并成一个StoreFile，此过程中不会处理已经Deleted和Expired的Cell。得到的是结果更少，更大的StoreFile。

**2. Major Compaction**：将所有的StoreFile合并成一个，并清理无意义的三种数据（已删除，TTL过期，版本号超过设定版本号）。此过程消耗大量资源，时间长。

Compaction本质：是使用短时间内的IO和带宽消耗，来换取查询操作的低延迟。Compaction的速度远低于HFile的生成速度，这样就会适合HFile的数量会越来越多，导致读性能下降，为了避免这种情况，在HFile的数量过多的时候会限制写请求的速度。

Split，Region的分裂。默认状态是HBase自动分割，也可以手动切割。

### 管理Compaction和Spilit

随着数据量的增大，split会被持续执行。如果需要知道现在有几个Region，比如长时间的debug或者做调优，需要手动切割。如果数据是均匀的，随着数据增长，很容易导致split / compaction疯狂的运行。因为所有的Region都是差不多大的。用手动切割，就可以交错执行定时的合并和切割操作，降低IO负载。

建议：关闭自动split。因为自动的splite是配置文件中的HBase.hRegion.max.filesize决定的。把它设置成Long.MAX\_VALUE是不推荐的做法，要是忘记手工切割，推荐的做法是设置成100GB，一旦到达这样的值，至少需要一个小时执行 major compactions。

关于最佳的在pre-splite Regions的数。这个决定于应用程序。可以先从低的开始，比如每个server，10个pre-splite Regions.然后花时间观察数据增长。有太少的Region至少比出错好，可以之后再rolling split。一个更复杂的答案是这个值是取决于Region中的最大的storefile。随着数据的增大，这个也会跟着增大。可以当这个文件足够大的时候，用一个定时的操作使用Store的合并选择算法(compact selection algorithm)来仅合并这一个HStore。如果不这样做，这个算法会启动一个 major compactions，很多Region会受到影响，集群会疯狂的运行。需要注意的是，这样的疯狂合并操作是数据增长造成的，而不是手动分割操作决定的。

### 相关优化

如果pre-split 导致 Regions 很小，你可以通过配置HConstants.MAJOR\_COMPACTION\_PERIOD把major compaction参数调大。如果数据变得太大，可以使用org.apache.hadoop.HBase.util.RegionSplitter 脚本来执行针对全部集群的一个网络IO安全的rolling split操作。

StoreFile是只读的，一旦创建后就不可以再修改。因此HBase的更新其实是不断追加的操作。当一个Store中的StoreFile达到一定的阈值后，就会进行一次合并(major compact)，将对同一个key的修改合并到一起，形成一个大的StoreFile，当StoreFile的大小达到一定阈值后，又会对 StoreFile进行分割(split)，等分为两个StoreFile。由于对表的更新是不断追加的，处理读请求时，需要访问Store中全部的StoreFile和MemStore，将它们按照row key进行合并，由于StoreFile和MemStore都是经过排序的，并且StoreFile带有内存中索引，通常合并过程还是比较快的。

实际应用中，可以考虑必要时手动进行major compact，将同一个row key的修改进行合并形成一个大的StoreFile。同时，可以将StoreFile设置大些，减少split的发生。

## HBase 排序

所有数据模型操作HBase返回排序的数据。先是row，再是column family，然后再是column qualifier，最后是时间戳（全部是倒序，最新的在前）。

## HBase 的LSM思想

LSM树的设计思想非常朴素：将对数据的修改增量保持在内存中，达到指定的大小限制后将这些修改操作批量写入磁盘，不过读取的时候稍微麻烦，需要合并磁盘中历史数据和内存中最近修改操作，所以写入性能大大提升，读取时可能需要先看是否命中内存，否则需要访问较多的磁盘文件。极端的说，基于LSM树实现的HBase的写性能比Mysql高了一个数量级，读性能低了一个数量级。

LSM树原理把一棵大树拆分成N棵小树，它首先写入内存中，随着小树越来越大，内存中的小树会flush到磁盘中，磁盘中的树定期可以做merge操作，合并成一棵大树，以优化读性能。

因为小树先写到内存中，为了防止内存数据丢失，写内存的同时需要暂时持久化到磁盘，对应了HBase的MemStore和HLog MemStore上的树达到一定大小之后，需要flush到HRegion磁盘中（一般是Hadoop DataNode），这样MemStore就变成了DataNode上的磁盘文件StoreFile，定期HRegionServer对DataNode的数据做merge操作，彻底删除无效空间，多棵小树在这个时机合并成大树，来增强读性能。

## 小结

以上内容介绍了HBase的基本原理和读写相关内容，同时也说明了一些HBase的工作机制和特点。作为广泛使用的海量数据存储组件，HBase还有很多技术细节需要研究和根据具体业务进行配置。

# HBase 竞品分析

NoSQL，泛指非关系型的数据库。

SQL，关系型数据库。

|  | **关系型数据库** | **非关系型数据库** |
| --- | --- | --- |
| **优点** | 1. 易于维护：都是使用表结构，格式一致;  2. 使用方便：SQL语言通用，可用于复杂查询； 3. 复杂操作：支持SQL，可用于一个表以及多个表之间非常复杂的查询。 | 1. 格式灵活：存储数据的格式可以是key，value形式、文档形式、图片形式等等，文档形式、图片形式等等，使用灵活，应用场景广泛，而关系型数据库则只支持基础类型。 2. 速度快：NoSQL可以使用硬盘或者随机存储器作为载体，而关系型数据库只能使用硬盘； 3. 高扩展性； 4. 成本低：NoSQL数据库部署简单，基本都是开源软件。 |
| **缺点** | 1. 读写性能比较差，尤其是海量数据的高效率读写； 2. 固定的表结构，灵活度稍欠； 3. 高并发读写需求，传统关系型数据库来说，硬盘I/O是一个很大的瓶颈。 | 1. 不提供sql支持，学习和使用成本较高； 2. 无事务处理； 3. 数据结构相对复杂，复杂查询方面稍欠。 |

## NoSQL数据库

NoSQL，泛指非关系型数据库，主要分为四大类：

1. key-value存储数据库。该类数据库使用哈希表，在哈希表中包含特定的key和与其对应的指向特定数据的指针。常用的有Redis。
2. 列存储数据库。该类数据库主要用来应对分布式存储的海量数据，一个键指向了多个列。常用的有HBase。
3. 文档型数据库。该类数据库将结构化、半结构化的文档以特定格式存储，如json格式。一个文档相当于关系型数据库中的一条记录，也是处理信息的基本单位。常用的有MongoDB。
4. 图形数据库。该类数据库使用图形理论来存储实体之间的关系信息，最主要的组成部分是：结点集、连接节点的关系。常用的有Neo4j。

**非关系型数据库的显著特点：**

1. 数据模型比较简单。
2. 对数据库性能的要求比较高。
3. 不需要高度的数据一致性。

## 主流的NoSQL数据库

主流的NoSQL数据库包括，Cassandra、MongoDB、CouchDB、Redis、 Riak、Membase、Neo4j 、CoucHBase 和 HBase，下面将分别介绍。

### CoucHBase

* 所用语言：C/C++。
* 特点：低延迟访问、水平伸缩、高可用。
* 使用许可： BSD。
* 协议：DCP。
* 良好的 cluster 支持。
* Key 可以动态分散（Auto Sharding）在不同的服务器上，可以通过动态添加服务器节点增加系统容量。
* 没有单点失效，任何一个单点都不会造成数据不可访问。
* 读写负载可以均匀分布在系统的不同节点上。
* 支持异步持久化支持。
* 方便快速恢复，甚至可以直接用作 key/value 数据库。经常在跟业界朋友交流时，会提到用 key 分段的方法来做容量扩展以及负载均衡。但是用静态的 key 分段会有不少问题。
* Cache 系统本身及使用 cache 的客户端都需要预设一个分段逻辑，这个逻辑后期如果需要调整将会非常困难。不能解决单点失效的问题，还需要额外的手段。运维需要更多的人为参与，避免 key 超出现有分区，一旦出现 key 找不到对应服务器，访问直接失败。

**优点**：

1.高并发性，高灵活性，高拓展性，容错性好。

2.以 vBucket 的概念实现更理想化的自动分片以及动态扩容。

**缺点:**

1.CoucHBase 的存储方式为 Key/Value，但 Value 的类型很为单一，不支持数组。另外也不会自动创建doc id，需要为每一文档指定一个用于存储的 Document Indentifer。

2.各种组件拼接而成，都是c++实现，导致复杂度过高，遇到奇怪的性能问题排查比较困难，（中文）文档比较欠缺。

3.采用缓存全部key的策略，需要大量内存。节点宕机时 failover 过程有不可用时间，并且有部分数据丢失的可能，在高负载系统有假死现象。

4.逐渐倾向于闭源，社区版本（免费，但不提供官方维护升级）和商业版本之间差距比较大。

**最佳应用场景:**

适合对读写速度要求较高，但服务器负荷和内存花销可遇见的需求；需要支持 Memcached 协议的需求。

### MongoDB

* 所用语言：C++。
* 特点：保留了SQL一些友好的特性（查询，索引）。
* 使用许可： AGPL（发起者： Apache）。
* 协议： Custom， binary（ BSON）。
* Master/slave复制（支持自动错误恢复，使用 sets 复制）。
* 内建分片机制。
* 支持 JavaScript表达式查询。
* 可在服务器端执行任意的 JavaScript函数。
* update-in-place支持比CouchDB更好。
* 在数据存储时采用内存到文件映射。
* 对性能的关注超过对功能的要求。
* 建议最好打开日志功能（参数 –journal）。
* 在32位操作系统上，数据库大小限制在约2.5Gb。
* 空数据库大约占 192Mb。
* 采用 GridFS存储大数据或元数据（不是真正的文件系统）。

**优势：**

1. 强大的自动化 shading 功能。
2. 全索引支持，查询非常高效。
3. 面向文档（BSON）存储，数据模式简单而强大。
4. 支持动态查询，查询指令也使用JSON形式的标记，可轻易查询文档中内嵌的对象及数组。
5. 支持 JavaScript 表达式查询，可在服务器端执行任意的 JavaScript函数。

**缺点：**

1. 单个文档大小限制为16M，32位系统上，不支持大于2.5G的数据。
2. 对内存要求比较大，至少要保证热数据（索引，数据及系统其它开销）都能装进内存。
3. 非事务机制，无法保证事件的原子性。

**最佳应用场景**：

适用于需要动态查询支持；需要使用索引而不是 Map/Reduce功能；需要对大数据库有性能要求；需要使用 CouchDB但因为数据改变太频繁而占满内存的应用程序。

### CouchDB

Apache CouchDB 是一个面向文档的数据库管理系统。它提供以 JSON 作为数据格式的 REST 接口来对其进行操作，并可以通过视图来操纵文档的组织和呈现。 CouchDB 是 Apache 基金会的顶级开源项目。

CouchDB是用Erlang开发的面向文档的数据库系统，其数据存储方式类似Lucene的Index文件格式。CouchDB最大的意义在于它是一个面向Web应用的新一代存储系统，事实上，CouchDB的口号就是：下一代的Web应用存储系统。

* 所用语言： Erlang。
* 特点：DB一致性，易于使用。
* 使用许可： Apache。
* 协议： HTTP/REST。
* 双向数据复制。
* 持续进行或临时处理。
* 处理时带冲突检查。
* 因此，采用的是master-master复制。
* MVCC – 写操作不阻塞读操作。
* 可保存文件之前的版本。
* Crash-only（可靠的）设计。
* 需要不时地进行数据压缩。
* 视图：嵌入式 映射/减少。
* 格式化视图：列表显示。
* 支持进行服务器端文档验证。
* 支持认证。
* 根据变化实时更新。
* 支持附件处理。。
* 需要 jQuery程序库。

**优势:**

1. 完全面向Web应用的分布式数据库。
2. 大规模、高并发的面向文档存储。
3. 丰富的REST API。

**缺点**：

1. 维护性差，一旦失败，就需要终止所有查询，甚至包括复制和压缩。
2. 视图索引只能在查询时更新，无法在插入记录时更新索引，影响查询性能。

**最佳应用场景：**

适用于数据变化较少，执行预定义查询，进行数据统计的应用程序。适用于需要提供数据版本支持的应用程序。

例如： CRM、CMS系统。 master-master复制对于多站点部署是非常有用的。

### Redis

* 所用语言：C/C++。
* 特点：运行异常快。
* 使用许可： BSD。
* 协议：类 Telnet。
* 有硬盘存储支持的内存数据库。
* 但自2.0版本以后可以将数据交换到硬盘（2.4以后版本不支持该特性）。
* Master-slave复制。
* 虽然采用简单数据或以键值索引的哈希表，但也支持复杂操作。
* INCR & co （适合计算极限值或统计数据）。
* 支持 sets（同时也支持 union/diff/inter）。
* 支持列表（同时也支持队列；阻塞式 pop操作）。
* 支持哈希表（带有多个域的对象）。
* 支持排序 sets（高得分表，适用于范围查询）。
* Redis支持事务。
* 支持将数据设置成过期数据（类似快速缓冲区设计）。
* Pub/Sub允许用户实现消息机制。

**优势：**

1. 非常丰富的数据结构。
2. Redis提供了事务的功能，可以保证一串 命令的原子性，中间不会被任何操作打断。
3. 数据存在内存中，读写非常的高速，可以达到10w/s的频率。

**缺点：**

1. Redis3.0后才出来官方的集群方案，但仍存在一些架构上的问题。
2. 持久化功能体验不佳——通过快照方法实现的话，需要每隔一段时间将整个数据库的数据写到磁盘上，代价非常高；而AOF方法只追踪变化的数据，类似于MySQL的binlog方法，但追加log可能过大，同时所有操作均要重新执行一遍，恢复速度慢。
3. 由于是内存数据库，所以，单台机器，存储的数据量，跟机器本身的内存大小。虽然Redis本身有key过期策略，但是还是需要提前预估和节约内存。如果内存增长过快，需要定期删除数据。

**最佳应用场景：**

适用于数据变化快且数据库大小可遇见（适合内存容量）的应用程序。

例如：股票价格、数据分析、实时数据搜集、实时通讯。

### Riak

* 所用语言：Erlang和C，以及一些JavaScript。
* 特点：具备容错能力。
* 使用许可： Apache。
* 协议： HTTP/REST或者 custom binary。
* 可调节的分发及复制(N， R， W)。
* 用 JavaScript or Erlang在操作前或操作后进行验证和安全支持。
* 使用JavaScript或Erlang进行 Map/Reduce。
* 连接及连接遍历：可作为图形数据库使用。
* 索引：输入元数据进行搜索（1.0版本即将支持）。
* 支持Masterless多站点复制及商业许可的 SNMP监控。

**优势：**

* 1. 没有主节点的概念，因此在处理故障方面有更好的弹性。
  2. Riak 的数据模型更加灵活。
  3. 更好地支持分布式、容错应用程序。

**最佳应用场景：**

适用于想使用类似 Cassandra（类似Dynamo）数据库但无法处理 bloat及复杂性的情况。适用于你打算做多站点复制，但又需要对单个站点的扩展性，可用性及出错处理有要求的情况。

例如：销售数据搜集，工厂控制系统；对宕机时间有严格要求；可以作为易于更新的 web服务器使用。

### Neo4j

* 所用语言： Java。
* 特点：基于关系的图形数据库。
* 使用许可： GPL，其中一些特性使用 AGPL/商业许可。
* 协议： HTTP/REST（或嵌入在 Java中）。
* 可独立使用或嵌入到 Java应用程序。
* 图形的节点和边都可以带有元数据。
* 很好的自带web管理功能。
* 使用多种算法支持路径搜索。
* 使用键值和关系进行索引。
* 为读操作进行优化。
* 支持事务（用 Java api）。
* 使用 Gremlin图形遍历语言。
* 支持 Groovy脚本。
* 支持在线备份，高级监控及高可靠性支持使用 AGPL/商业许可。

**优势：**

1. 数据的插入，查询操作很直观，不用再像之前要考虑各个表之间的关系。
2. 提供的图搜索和图遍历方法很方便，速度也是比较快的。

**缺点：**

1. 插入速度较慢。
2. 超大节点。当有一个节点的边非常多时（常见于大V），有关这个节点的操作的速度将大大下降。这个问题很早就有了，官方也说过会处理，现在仍然不能让人满意。
3. 提高数据库速度的常用方法就是多分配内存，但无法直接设置数据库内存占用量，而是需要计算后为其”预留“内存。

**最佳应用场景：**

适用于图形一类数据。这是 Neo4j与其他NoSQL数据库的最显著区别。

例如：社会关系，公共交通网络，地图及网络拓谱。

### CoucHBase

* 所用语言：C/C++。
* 特点：低延迟访问、水平伸缩、高可用。
* 使用许可： BSD。
* 协议：DCP。
* 良好的 cluster 支持。
* Key 可以动态分散（Auto Sharding）在不同的服务器上，可以通过动态添加服务器节点增加系统容量。
* 没有单点失效，任何一个单点都不会造成数据不可访问。
* 读写负载可以均匀分布在系统的不同节点上。。
* 支持异步持久化支持。
* 方便快速恢复，甚至可以直接用作 key/value 数据库。经常在跟业界朋友交流时，会提到用 key 分段的方法来做容量扩展以及负载均衡。
* Cache 系统本身及使用 cache 的客户端都需要预设一个分段逻辑，这个逻辑后期如果需要调整将会非常困难。不能解决单点失效的问题，还需要额外的手段。运维需要更多的人为参与，避免 key 超出现有分区，一旦出现 key 找不到对应服务器，访问直接失败。

**优势：**

* 1. 高并发性，高灵活性，高拓展性，容错性好。
  2. 以 vBucket 的概念实现更理想化的自动分片以及动态扩容。

**缺点：**

1. CoucHBase 的存储方式为 Key/Value，但 Value 的类型很为单一，不支持数组。另外也不会自动创建doc id，需要为每一文档指定一个用于存储的 Document Indentifer。
2. 各种组件拼接而成，都是c++实现，导致复杂度过高，遇到奇怪的性能问题排查比较困难，（中文）文档比较欠缺。
3. 采用缓存全部key的策略，需要大量内存。节点宕机时 failover 过程有不可用时间，并且有部分数据丢失的可能，在高负载系统上有假死现象。
4. 逐渐倾向于闭源，社区版本（免费，但不提供官方维护升级）和商业版本之间差距比较大。

**最佳应用场景:**

适合对读写速度要求较高，但服务器负荷和内存花销可遇见的需求；需要支持 memcached 协议的需求。

### HBase

* 所用语言： Java。
* 特点：支持数十亿行X上百万列。
* 使用许可： Apache。
* 协议：HTTP/REST 。
* 在 BigTable之后建模。
* 采用分布式架构 Map/Reduce。
* 对实时查询进行优化。
* 高性能 Thrift网关。
* 通过在server端扫描及过滤实现对查询操作预判。
* 支持 XML， Protobuf， 和binary的HTTP。
* Cascading， hive， and pig source and sink modules。
* 基于 Jruby（ JIRB）的shell。
* 对配置改变和较小的升级都会重新回滚。
* 不会出现单点故障。
* 堪比MySQL的随机访问性能。

**优势：**

1. 存储容量大，一个表可以容纳上亿行，上百万列。
2. 可通过版本进行检索，能搜到所需的历史版本数据。
3. 负载高时，可通过简单的添加机器来实现水平切分扩展，跟Hadoop的无缝集成保障了其数据可靠性（HDFS）和海量数据分析的高性能（MapReduce）。
4. 可有效避免单点故障的发生。

**缺点：**

1. 基于Java语言实现及Hadoop架构意味着其API更适用于Java项目。
2. node开发环境下所需依赖项较多、配置麻烦（或不知如何配置，如持久化配置），缺乏文档。
3. 占用内存很大，且鉴于建立在为批量分析而优化的HDFS上，导致读取性能不高。
4. API相比其它 NoSQL 的相对笨拙。

**最佳应用场景：**

适用于偏好BigTable。并且需要对大数据进行随机、实时访问的场合。 例如： Facebook消息数据库。

对比其他NoSQL系统（例如Redis、MongoDB、Cassandra等），HBase基于HDFS不支持复杂事务、最初设计中最大的考量因素就是扩展性，其设计的初衷就是基于集群、扩展性好、故障恢复机制清晰高效、基于水平分片的负载分发模式易于调整。

## HBase直接竞品分析

面对不同类型的数据，本应该由相对应特点的数据库进行存储，但是在企业生产环境中，无论是运营成本还是研究开发成本都不可能去使用冗余的存储技术。现在市场主要围绕在三个NoSQL数据库上：MongoDB，Cassandra（主要由DataStax开发的，诞生于Facebook），和HBase的（和Hadoop紧密关联在一起，也被相同社区开发出来）。

### HBase 与 Cassandra

Cassandra在架构上更多借鉴了dynamo，一种完全的区中心对等的分布式数据库，她的每个节点维护一份元信息，每一个节点在集群中的身份完全一样。

#### 系统部署

对于HBase而言，部署HBase前，需要部署的组件有Zookeeper，HDFS，然后才是HBase。对应的Cassandra就比较简单很多，编译完成一个jar包，单台服务器启动一个Cassandra进程即可。

在部署HBase的时候，需要规划好，哪些机器跑HMaser，RS，ZK，HDFS的相关进程等。为了集群的性能，还要预先规划好多少个RS。人工部署一个HBase集群工作量比较大，同时后期运维工作量也很大。

Cassandra部署的时候比较简单，一个tar包搞定，由于Cassandra数据落本地盘，需要人为的配置一些参数比如是否需要虚拟节点（vnode）以及多少vnode；需要基于业务的场景选择特定的key的放置策略（partitioner），这个放置策略的选择以及一些参数的配置需要一定的门槛。

简单总结下：部署运维的话，HBase依赖组件多，部署麻烦一点，但是相关资料很多，降低了难度；Cassandra部署依赖少，但配置参数多，相关资料较少。

特别是使用云HBase完全避免了部署造成的各种麻烦，比手工部署运维任何大数据数据库都方便很多。

#### 支持特性

表3- 1 HBase与Cassandra对比

|  |  |
| --- | --- |
| **HBase** | **Cassandra** |
| 强一致性的读写：不是一个最终一致性的存储。 | C\*借鉴Dynamo的架构思想，把自己叫做一个最终一致性的系统； |
| 自动sharding：HBase的table在集群种被分布在各个Region，Region可以做自动切分。 | C\*的sharding方式：一致性hash，有2种：（1）人为配置好initial\_token；2.使用vnode，集群初始化以及节点bootstrap的时候会计算token，基于这些token做数据sharding。 |
| RegionServer的failover； | 可以容忍：replicator\_number - (read/write level sufficient nodes)个节点挂了，比如3个副本，读写级别QUORUM（sufficient nodes是2），能容忍1节点挂； |
| Hadoop/HDFS的集成； | 支持MapReduce; |
| MapReduce：支持大数据的并行处理； | Thrift、CQL访问; |
| JAVA Client 以及Thrift/RESR API 访问； | 大数据处理的bloom filter 必备； |
| Block Cache 以及Bloom filter； | 有jmx等常见管理，且datastax 公司有提供ops center； |

两者都有自己适合的场景，如果业务对数据一致性要求比较苛刻，那么HBase可能更合适。毕竟Cassandra还是存在一定的问题，比如删除数据可能复现；Cassandra推荐放置策略用Random以及Murmur3等方式，这样把key打散的很随机，以此做节点负载均衡，这样做scan业务自己的需求数据，可能数据库要全集群都访问，当然可以使用OrderPreserving 和 ByteOrdered可以不用全集群都访问，可是负载不是很均匀，HBase这点却支持的不错；

就易用性来说，Cassandra做的还是不错的，在数据库内部提供cql，一种类sql的语句。HBase还是主要JAVA API/Thrift的接口支持。

#### 使用功能

1. 一致性

HBase，主要是依赖底层的HDFS来支持副本冗余，一次写入被RegionServer收到以后，发送给底层的HDFS，而HDFS会对应的给这个数据在整个文件系统里面写多份；这里是多份副本全部写完成才会返回。读数据的话，在底层HDFS的处理是读主要的这份数据就可以，因为多份数据都是一样的。

Cassandra在建表的时候会指定表的副本数（常见3副本），一次数据写入，会基于当前表的副本数以及节点的snitch策略来找到需要写的数据节点，发出多份请求（3份），然后基于传递的写入级别等待对应的响应数即可。但是无论是网络还是节点原因导致，3份数据中有失败的，那么3份数据将不一致。但是，Cassandra自己具有修复机制，1.读修复；2.Hinted-Handoff；3.Anti-entropy repair ;但是各种机制的引入也不是很完美的解决问题，此外还相应的会引入一些问题，比如使用1的话，那些读不到的数据存在一直数据不一致的风险；使用3的方式去进行全量/增量数据对比，会消耗很多物理资源，影响在线服务的请求，这是在线服务不能忍受的。

1. Sharding

HBase 的各个RegionServer在最初负责的Region，是可以在最初的建表时候，可以做预分配，也可以让HBase自己做这件事情，那么每个RegionServer就会负责相应的Region的数据的读写等。对于出现热点Region的情况的话，HBase自己支持Region的split操作，将热点Region一分为二。

Cassandra不支持所谓的热点数据split Region的功能，那么对于这种情况的话，其做了一个预先设置，输入的数据做hash打散，也就是我们知道的一致性hash，内部支持4种hash策略，Murmur3，Random，OrderPreserving等，其中，前2种是做了随机的hash，OrderPreserving 是类似字典序的方式，最初无论是使用vnode的方式还是initial\_tokne的方式人为设置节点token，来一个请求，计算随机hash可以把key比较随机打散到集群中的某个节点，通过snitch 和 keyspace 副本方式找到落得节点信息，因为前面的随机hash可以人为是比较随机的，那么这实际上可以理解为一种负载均衡。但是如果OrderPreserving 这种方式，实际上就会有问题，出现热点也没办法。

1. 使用方式

HBase现在提供给用户主要是JAVA/THrift/REST的接口，大概的操作也就是CRUD操作。

Cassandra而言，支持2种方式，1.THrift；2.CQL。

1. 数据复制

一般数据复制，有全量数据复制和增量数据复制2个情况。

对HBase而言，全量复制的时候，有的使用copytable的方式，这边比较简单，但是因为是MR做批量读取写入，但是会比较耗时，因为每次请求一个网络来回；也有基于snapshot做复制的方式，这种方式会好点；增量复制的话，使用的是复制Hlog的方式，这是一种异步的复制方式，在zk记录checkpoint，复制完log以后修改checkpoint的问位置。

Cassandra的数据复制，也与两种方案，但是都是需要时在Network的拓扑配置下，同cluster的不同dc环境下的操作。全量复制是一种叫做rebuild的方式，直接拖数据的stream，类似Cassandra做节点启动bootstrap的过程。Cassandra的增量复制提供了2种方式：EACH\_QUORUM，LOCAL\_QUORUM 前者是同步写主备，后者是写本地，返回client，远端是否成功并不关系；对于这个而言的话，异步存在丢数据的风险，同步在跨region以及并发量大的情况下，请求失败率会很高。但是，上面的方式是需要部署在同cluster下面的不同dc，不同cluster没有解决方案。此外对于同步和异步方案之间没有一种折中的方式，毕竟有的场景对性能要求高，同步复制影响线上写成功的概率，异步复制会丢数据。

#### 各自的优势

通过上述 的描述以及各自的功能特性的对比，可以得到HBase 和 Cassandra对比，2种产品各有千秋，其中HBase对数据一致性要求更高，Cassandra相应的强调可用性。且现阶段Cassandra的接口比较有亲和性，但是HBase和Hadoop系统天然的无缝对接，这是Cassandra还有点欠缺的。Cassandra也使用了各自方式去弥补欠缺，但是实际上数据一致性上做的还是和这种强一致的系统略差一些。

### HBase 与 MongDB

MongoDB 是一种文档性的数据库。存放xml、json、bson类型的数据。这些数据具备自述性（self-describing），呈现分层的树状数据结构。

但是，MongoDB和已经熟悉的关系型数据库分享了很多同样的概念、操作、策略和过程。监控、索引、调整和备份等内容的流程和最佳实践可以应用到MongoDB。

#### 系统部署

前文3.3.1.1已经阐述过HBase的部署较为复杂，运维也比较复杂。

MongoDB特点就是高性能、易部署、易使用，存储数据非常方便，最大的特点在于它支持的查询语言非常强大，其语法有点类似于面向对象的查询语言。部署关系型数据库的概念、操作和流程可以被直接地应用到MongoDB上。分片集群的部署较为复杂，但因为只是单个数据库组件的部署，分片集群需要涉及到规划内容包括：mongos、configserver、shard、replica set等几条，相较于HBase的部署和配置，还是显得较为简单。

#### 支持特性

此处重点说明MongoDB的特性，和HBase是一个完全的NoSQL数据库不同。MongoDB和其最大的区别就是，它更加接近于关系型数据库的使用特点。

1. MongoDB和HBase都支持MapReduce。不过MongoDB的MapReduce支持不够强大，如果没有使用MongoDB分片，MapReduce实际上不是并行执行。
2. MongoDB支持shard分片，HBase根据row key自动负载均衡，这里shard key和row key的选取尽量用非递增的字段，尽量用分布均衡的字段，因为分片都是根据范围来选择对应的存取server的，如果用递增字段很容易热点server的产生，由于是根据key的范围来自动分片的，如果key分布不均衡就会导致有些key根本就没法切分，从而产生负载不均衡。
3. MongoDB的读效率比写高，HBase默认适合写多读少的情况，可以通过hfile.block.cache.size配置，该配置storefile的读缓存占用Heap的大小百分比，0.2表示20%。该值直接影响数据读的性能。如果写比读少很多，开到0.4-0.5也没问题。如果读写较均衡，0.3左右。如果写比读多，果断默认0.2吧。设置这个值的时候，同时要参考HBase.regionserver.global.memstore.upperLimit，该值是memstore占heap的最大百分比，两个参数一个影响读，一个影响写。如果两值加起来超过80-90%，会有OOM的风险。
4. HBase采用的LSM思想(Log-Structured Merge-Tree)，就是将对数据的更改hold在内存中，达到指定的threadhold后将该批更改merge后批量写入到磁盘，这样将单个写变成了批量写，大大提高了写入速度，不过这样的话读的时候就费劲了，需要merge disk上的数据和memory中的修改数据，这显然降低了读的性能。MongoDB采用的是mapfile+Journal思想，如果记录不在内存，先加载到内存，然后在内存中更改后记录日志，然后隔一段时间批量的写入data文件，这样对内存的要求较高，至少需要容纳下热点数据和索引。
5. MongoDB的update是update-in-place，也就是原地更新，除非原地容纳不下更新后的数据记录。而HBase的修改和添加都是同一个命令：put，如果put传入的row key已经存在就更新原记录，实际上HBase内部也不是更新，它只是将这一份数据已不同的版本保存下来而已，HBase默认的保存版本的历史数量是3。

#### 使用功能

1. MongoDB 主键是“\_id”，主键上面可以不建索引，记录插入的顺序和存放的顺序一样；HBase的主键就是row key，可以是任意字符串(最大长度是 64KB，实际应用中长度一般为 10-100bytes)，在HBase内部，row key保存为字节数组。存储时，数据按照Row key的字典序(byte order)排序存储。设计key时，要充分排序存储这个特性，将经常一起读取的行存储放到一起。
2. MongoDB支持二级索引，而HBase本身不支持二级索引。
3. MongoDB支持集合查找，正则查找，范围查找，支持skip和limit等等，是最像mysql的NoSQL数据库，而HBase只支持三种查找：通过单个row key访问，通过row key的range，全表扫描 。

#### 各自的优势

MongoDB更像传统的关系型数据库，更善于做查询。Hbase更偏向非关系型数据库，扩展储存能力强。

## 小结

在3.1节和3.2节对主流的NoSQL数据库进行了简单的介绍，同时对各自的特点进行了分析，对应用场景进行了描述。3.3节对HBase主要的两个竞品进行了详细地对比分析。虽然同为NoSQL数据库，也都支持海量数据库，但是HBase、Cassandra和MongoDB还是有各自的特点和偏向，HBase还是以最偏向非关系型数据库的结构和水平扩展能力强两个特点，是与其他两者明显的区别。

# HBase的周边生态

## HBase的底层存储—HDFS (Hadoop File System)

HBase作为Hadoop生态系统中的一个重要组件，其开发初衷就是为了解决存储在HDFS上文件的实时操作缺陷。

## 能配合HBase存储的查询工具

### HBase Shell

HBase的命令行工具，最简单的接口，适合HBase管理使用，可以使用shell命令来查询HBase中数据的详细情况。安装完HBase之后，启动hadoop集群(利用HDFS存储)，启动Zookeeper，使用start-HBase.sh命令开启HBase服务，最后在shell中执行HBase shell就可以进入命令行界面。

### HBase thrift

Thrift server是HBase中的一种服务，主要用于对多语言API的支持。基于Apache Thrift（多语言支持的通信框架）开发，目前有两种版本thrift和thrift2。

thrift2是当时为了适应新的Java API，提出来的。由于种种原因，thrift2没有完美兼容并替代thrift，所有就留下了两个版本。

Thrift其实就是个代理，你的请求发到Thrift server上后，server通过Java API再帮你访问HBase。

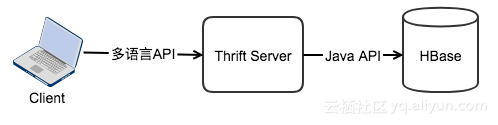


图 ‑1 Thrift 原理图

Thrift实现类是org.apache.hadoop.HBase.thrift.ThriftServer，thrift2的实现类是org.apache.hadoop.HBase.thrift2.ThriftServer。它们访问HBase使用的也是普通的HBase client API，所以当你的请求到达Thrift server后，它通过client API去帮你定位数据，然后读取数据。这么来看，Thrift Server比较灵活，你可以部署在[客户机](https://www.baidu.com/s?wd=%E5%AE%A2%E6%88%B7%E6%9C%BA&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)上，也可以独立部署一个thrift集群。

### Hive

Hive是建立在Hadoop之上为了减少MapReduce jobs编写工作的批处理系统。使用Hive可以通过编写SQL语句来完成MapReduce的批处理任务。早期的Hive并不支持HBase的查询操作，使得对HBase进行MapReduce计算需要通过Java语言来完成，代码量较大。但是在Hive1.x之后的版本，开始对HBase支持。

具体版本支持情况如下：

Hive1.x与HBase0.98.x或则更低版本是兼容的；

Hive2.x与HBase1.x及比HBase1.x更高版本兼容；

如果想HBase1.x与Hive1.x整合，需要编译Hive1.x stream 代码本身。

### Solr

Solr（读作“solar”）是Apache Lucene项目的开源企业搜索平台。其主要功能包括全文检索、命中标示、分面搜索、动态聚类、数据库集成，以及富文本（如Word、PDF）的处理。Solr是高度可扩展的，并提供了分布式搜索和索引复制。Solr是最流行的企业级搜索引擎，Solr4 还增加了NoSQL支持。

Solr是用Java编写、运行在Servlet容器（如 Apache Tomcat 或Jetty）的一个独立的全文搜索服务器。 Solr采用了 Lucene Java 搜索库为核心的全文索引和搜索，并具有类似REST的HTTP/XML和JSON的API。Solr强大的外部配置功能使得无需进行Java编码，便可对 其进行调整以适应多种类型的应用程序。Solr有一个插件架构，以支持更多的高级定制。

因为2010年 Apache Lucene 和 Apache Solr 项目合并，两个项目是由同一个Apache软件基金会开发团队制作实现的。提到技术或产品时，Lucene/Solr或Solr/Lucene是一样的。

HBase可以通过协处理器Coprocessor的方式向Solr发出请求，Solr对于接收到的数据可以做相关的同步：增、删、改索引的操作，这样就可以同时使用HBase存储量大和Solr检索性能高的优点了，更何况HBase和Solr都可以集群。这对海量数据存储、检索提供了一种方式，将存储与索引放在不同的机器上。

### ES

Elasticsearch是一个建立在全文搜索引擎 Apache Lucene™ 基础上的搜索引擎，可以说Lucene是当今最先进，最高效的全功能开源搜索引擎框架。

但是Lucene只是一个框架，要充分利用它的功能，需要使用JAVA，并且在程序中集成Lucene。需要很多的学习了解，才能明白它是如何运行的，Lucene确实非常复杂。

Elasticsearch使用Lucene作为内部引擎，但是在使用它做全文搜索时，只需要使用统一开发好的API即可，而不需要了解其背后复杂的Lucene的运行原理。

当然Elasticsearch并不仅仅是Lucene这么简单，它不但包括了全文搜索功能，还可以进行以下工作:

分布式实时文件存储，并将每一个字段都编入索引，使其可以被搜索。

实时分析的分布式搜索引擎。

可以扩展到上百台服务器，处理PB级别的结构化或非结构化数据。

ES+HBase对接大致有两种方式，需要根据当前的业务场景做相应的选择，

**方案1：**

如果是对写入数据性能要求高的业务场景，那么一份数据先写到HBase，然后再写到ES中，两个写入流程独立，这样可以达到性能最大，目前某公安厅使用该方案，每天需要写入数据200亿，6T数据，每个记录建20左右的索引。

**缺点：**可能存在数据的不一致性。

**方案2：**

这也是目前网上比较流行的方案，使用HBase的协处理监听数据在HBase中的变动，实时的更新ES中的索引，

**缺点：**是协处理器会影响HBase的性能

### Phoenix

Phoenix，由saleforce.com开源的一个项目，后又捐给了Apache。它相当于一个Java中间件，帮助开发者，像使用JDBC访问关系型数据库一些，访问NoSQL数据库HBase。

Phoenix，操作的表及数据，存储在HBase上。Phoenix只是需要和HBase进行表关联起来。然后再用工具进行一些读或写操作。

其实，可以把Phoenix只看成一种代替HBase的语法的一个工具。虽然可以用Java可以用JDBC来连接Phoenix，然后操作HBase，但是在生产环境中，不可以用在OLTP中。在线事务处理的环境中，需要低延迟，而Phoenix在查询HBase时，虽然做了一些优化，但延迟还是不小。所以依然是用在OLAT中，再将结果返回存储下来。

phoenix与HBase版本对应关系：

Phoenix 2.x - HBase 0.94.x

Phoenix 3.x - HBase 0.94.x

Phoenix 4.x - HBase 0.98.1+

Phoenix的优势还在于能够减少开发代码量，如将SQL编译成原生的HBase scans、确定scan关键字的最佳开始和结束和让scan并行执行等等。

### Impala

Impala是Cloudera在受到Google的Dremel启发下开发的实时交互SQL大数据查询工具（实时SQL查询引擎Impala），Impala没有再使用缓慢的Hive + MapReduce批处理，而是通过使用与商用并行关系数据库中类似的分布式查询引擎（由Query Planner、Query Coordinator和Query Exec Engine三部分组成），可以直接从HDFS或HBase中用SELECT、JOIN和统计函数查询数据，从而大大降低了延迟。

Impala相对于Hive，其突出特点就是快速。其优化的重点如下：

1. 没有使用MapReduce进行计算，将查询操作分成一个执行计划树，而不是MapReduce批处理任务，通过拉式获取数据，将结果数据组成按执行树流式传递汇集，减少中间结果落盘再读区的时间、空间开销。Impala使用服务的方式避免每次执行查询都需要启动的开销，同时使用Inline的方式减少函数调用的开销，加快执行效率。
2. 使用LLVM产生运行代码，针对特定查询生成特定代码，同时使用Inline的方式减少函数调用的开销，加快执行效率。
3. 充分利用可用的硬件指令（2）。
4. 更好的I/O调度，Impala知道数据块所在的磁盘位置能够更好的利用多磁盘的优势，同时Impala支持直接数据块读取和本地代码计算checksum。
5. 通过选择合适的数据存储格式可以得到最好的性能（Impala支持多种存储格式）。
6. 最大使用内存，中间结果不写磁盘，及时通过网络以stream的方式传递。

**优点：**

支持SQL查询，快速查询大数据；

可以对已有数据进行查询，减少数据的加载，转换；

多种存储格式可以选择（Parquet， Text， Avro， RCFile， SequeenceFile）；

可以与Hive配合使用。

**缺点：**

不支持用户定义函数UDF；

不支持text域的全文搜索；

不支持Transforms；

不支持查询期的容错；

对内存要求高。

### Spark

HBase可以划分在OLTP领域，它基于Row key点查性能好，能够自动sharding，具有高可用的特性。而Spark可以划分在OLAP领域，它是一款通用的DAG分析引擎，能够做高性能的内存迭代计算，具有完善的SQL优化层一系列特点。这两款产品的结合映射成了目前比较流行的一类数据库HTAP，它既具备OLAP的功能又具备OLTP的功能。

目前社区做Spark on HBase主要会做以下三方面的功能和优化：支持Spark SQL、Dataset、DataFrame API，支持分区裁剪、列裁剪、谓词下推等优化，Cache HBase的Connections。

使用Spark对HBase进行查询操作包括两种方式，一种式传统的使用常用的TableInputFormat和TableOutputFormat来读写HBase，另一种是是利用Cloudera-labs开源的一个HBaseContext的工具类来支持Spark用RDD的方式批量读写HBase。

其中第二种方式相对于第一种方式对HBase数据进行分析的优势如下，

1. 无缝的使用HBase connection。
2. 和Kerberos无缝集成。
3. 通过get或者scan直接生成rdd。
4. 利用RDD支持HBase的任何组合操作。
5. 为通用操作提供简单的方法，同时通过API允许不受限制的未知高级操作。
6. 支持Java和Scala。
7. 为Spark和 Spark streaming提供相似的API。

### Kylin

Apache Kylin™是一个开源的分布式分析引擎，提供Hadoop之上的SQL查询接口及多维分析（OLAP）能力以支持超大规模数据，最初由eBay Inc. 开发并贡献至开源社区。它能在亚秒内查询巨大的Hive表。Kylin相当于给HBase提供了一个多维查询的SQL能力。

它主要是通过预计算的方式将用户设定的多维立方体缓存到HBase中（目前还仅支持HBase）。

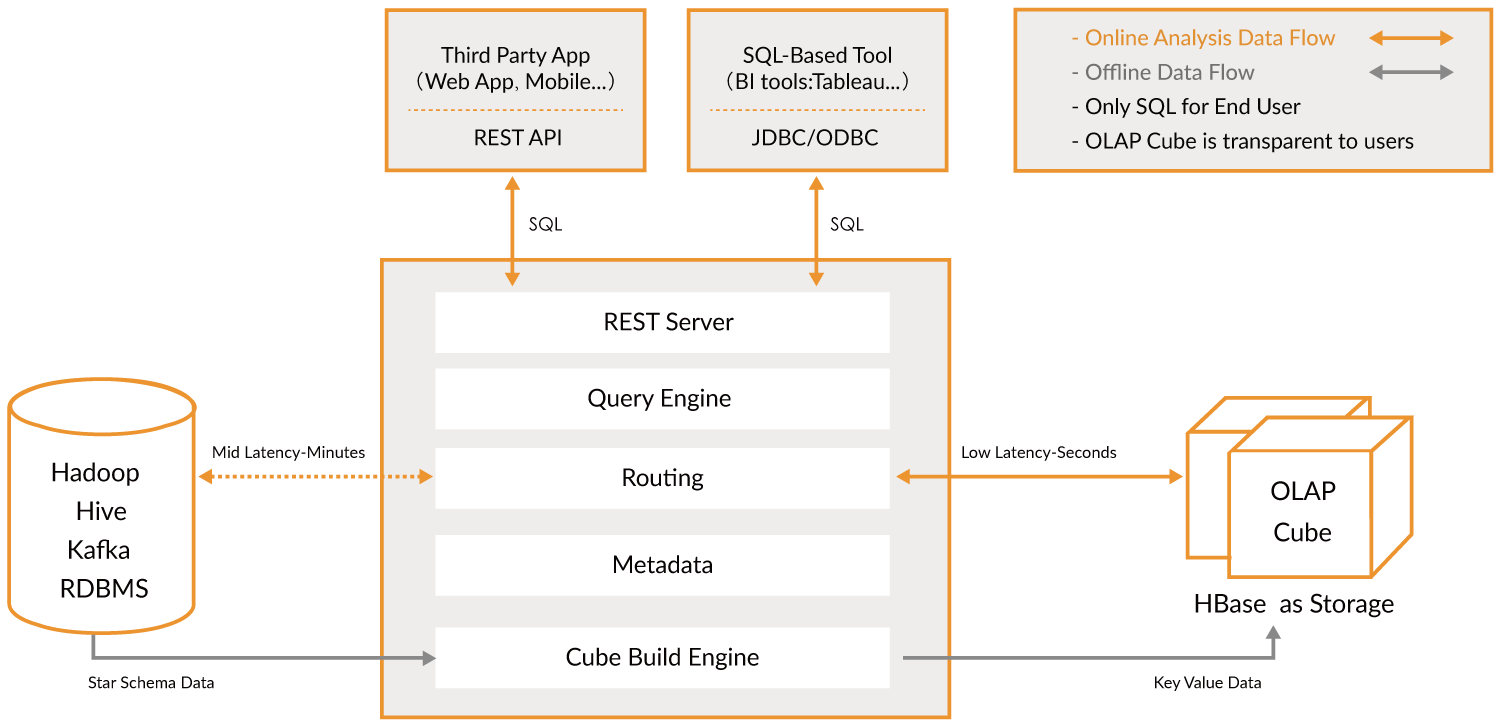


图 4‑2 Kylin系统框架

* 可扩展超快OLAP引擎:

Kylin是为减少在Hadoop/Spark上百亿规模数据查询延迟而设计。

* 交互式查询能力:

通过Kylin，用户可以与Hadoop数据进行亚秒级交互，在同样的数据集上提供比Hive更好的性能。

* 与BI工具无缝整合:

Kylin提供与BI工具的整合能力，如Tableau，PowerBI/Excel，MSTR，QlikSense，Hue和SuperSet。

**其他特性**：

* Job管理与监控
* 压缩与编码
* 增量更新
* 利用HBase Coprocessor
* 基于HyperLogLog的Dinstinc Count近似算法
* 友好的web界面以管理，监控和使用立方体
* 项目及表级别的访问控制安全
* 支持LDAP、SSO

## 能配合HBase存储的分析计算框架

### MapReduce

Hadoop MapReduce是一个软件框架，基于该框架能够容易地编写应用程序，这些应用程序能够运行在由上千个商用机器组成的大集群上，并以一种可靠的，具有容错能力的方式并行地处理上TB级别的海量数据集。

MapReduce是第一代分布式计算框架，同时也是一种变成思想。作为Hadoop生态的主要组件之一，MapReduce可以和HBase紧密地集成使用。在HBase中没有提供更好的二级索引的方式，在操作数据过程中，如果使用scan进行全表扫描，会极大的降低HBase的效率。使用MapReduce可以很好地解决HBase的读和写，以及数据迁移等场景下的大量读写需求。

### Spark

Apache Spark是用于大规模数据处理的统一分析引擎。Spark是一个实现快速通用的集群计算平台。它是由加州大学伯克利分校AMP实验室开发的通用内存并行计算框架，用来构建大型的、低延迟的数据分析应用程序。它扩展了广泛使用的MapReduce计算模型。高效的支撑更多计算模式，包括交互式查询和流处理。Spark的一个主要特点是能够在内存中进行计算，及时依赖磁盘进行复杂的运算，Spark依然比MapReduce更加高效。

基于MapReduce的计算引擎通常会将中间结果输出到磁盘上，进行存储和容错。出于任务管道承接的考虑，当一些查询翻译到MapReduce任务时，往往会产生多个Stage，而这些串联的Stage又依赖于底层文件系统（如HDFS）来存储每一个Stage的输出结果。

HBase client以put的方式封装数据，并支持逐条或批量插入。Spark读写HBase有两种方式，一种是传统方式，使用Spark中内置saveAsHadoopDataset和saveAsNewAPIHadoopDataset写入HBase；另一种方式是SparkOnHBase方式，可以使用Cloudera-labs开源的一个HBaseContext的工具类来支持Spark用RDD的方式批量读写HBase。第二种方式的优势在于：

1. 无缝的使用HBase connection。
2. 和Kerberos无缝集成。
3. 通过get或者scan直接生成rdd。
4. 利用RDD支持HBase的任何组合操作。
5. 为通用操作提供简单的方法，同时通过API允许不受限制的未知高级操作。
6. 支持Java和Scala。
7. 为Spark和 Spark streaming提供相似的API。

### Flink

Flink核心是一个流式的数据流执行引擎，其针对数据流的分布式计算提供了数据分布、数据通信以及容错机制等功能。基于流执行引擎，Flink提供了诸多更高抽象层的API以便用户编写分布式任务：

DataSet API， 对静态数据进行批处理操作，将静态数据抽象成分布式的数据集，用户可以方便地使用Flink提供的各种操作符对分布式数据集进行处理，支持Java、Scala和Python。

DataStream API，对数据流进行流处理操作，将流式的数据抽象成分布式的数据流，用户可以方便地对分布式数据流进行各种操作，支持Java和Scala。

Table API，对结构化数据进行查询操作，将结构化数据抽象成关系表，并通过类SQL的DSL对关系表进行各种查询操作，支持Java和Scala。

此外，Flink还针对特定的应用领域提供了领域库，例如：Flink ML，Flink的机器学习库，提供了机器学习Pipelines API并实现了多种机器学习算法。Gelly，Flink的图计算库，提供了图计算的相关API及多种图计算算法实现。

在Flink文档中，提供connector读取源数据和把处理结果存储到外部系统中。但是没有提供数据库的connector，如果要读写数据库，官网给出了异步IO(Asynchronous I/O)专门用于访问外部数据。

还有一种方法是继承RichSourceFunction，重写里面的方法，所有的数据库Flink都可以通过这两种方式进行数据的读写。

## 能配合HBase存储的消息队列

### Kafka

Kafka是最初由Linkedin公司开发，是一个分布式、支持分区的（partition）、多副本的（replica），基于Zookeeper协调的分布式消息系统，它的最大的特性就是可以实时的处理大量数据以满足各种需求场景：比如基于hadoop的批处理系统、低延迟的实时系统、storm/Spark流式处理引擎，web/nginx日志、访问日志，消息服务等。用Scala语言编写，Linkedin于2010年贡献给了Apache基金会并成为顶级开源 项目。

Kafka的特性:

1.高吞吐量、低延迟：Kafka每秒可处理几十万条消息，延迟最低仅几毫秒，每个topic可分多个partition， consumer group 对partition进行consume操作。

2.可扩展性：kafka集群支持热扩展 。

3.持久性、可靠性：消息被持久化到本地磁盘，支持数据备份防止数据丢失。

4.容错性：允许集群中节点失败（若副本数量为n，则允许n-1个节点失败）。

5.高并发：支持数千个客户端同时读写。

### HQueue

HQueue是一淘搜索网页抓取离线系统团队基于HBase开发的一套分布式、持久化消息队列。它利用HTable存储消息数据，借助HBase Coprocessor将原始的KeyValue数据封装成消息数据格式进行存储，并基于HBase Client API封装了HQueue Client API用于消息存取。

HQueue可以有效使用在需要存储时间序列数据、作为MapReduce Job和iStream等输入、输出供上下游共享数据等场合。

由于HQueue是基于HBase进行消息存取的，因此站在HDFS和HBase的肩膀上，使得其具备如下特点：

* + 1. 支持多Partitions，可根据需求设置Queue的规模，支持高并发访问（HBase的多Region）；
    2. 支持自动Failover，任何机器Down掉，Partition可自动迁移至其他机器（HBase的Failover机制）；
    3. 支持动态负载均衡，Partition可以动态被调度到最合理的机器上（HBase的LoadBalance机制，可动态调整）；
    4. 利用HBase进行消息的持久化存储，不丢失数据（HBase HLog和HDFS Append）；
    5. 队列的读写模式与HBase的存储特性天然切合，具备良好的并发读写性能（最新消息存储在MemStore中，写消息直接写入MemStore，通常场景下都是内存级操作）；
    6. 支持消息按Topic进行分类存取（HBase中的Qualifier）；
    7. 支持消息TTL，自动清理过期消息（HBase支持KeyValue级别的TTL）；
    8. HQueue = HTable Schema Design + HQueue Coprocessor + HBase Client Wrapper，完全扩展开发，无任何Hack工作，可随HBase自动升级；
    9. HQueue Client API基于HBase Client Wrapper进行简单封装，HBase的ThriftServer使得其支持多语言API，因此HQueue也很容易封装出多语言API；
    10. HQueue Client API可以天然支持Hadoop MapReduce Job和iStream的InputFormat机制，利用Locality特性将计算调度到存储最近的机器；
    11. HQueue支持消息订阅机制（HQueue 0.3及后续版本）。

## 能配合HBase存储的监控组件

### Ganglia

Ganglia是UC Berkeley发起的一个开源集群监视项目，设计用于测量数以千计的节点。Ganglia的核心包含gmond、gmetad以及一个Web前端。主要是用来监控系统性能，如：CPU 、mem、硬盘利用率，I/O负载、网络流量情况等，通过曲线很容易见到每个节点的工作状态，对合理调整、分配系统资源，提高系统整体性能起到重要作用。

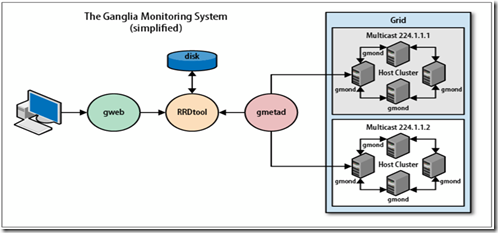


图 4‑3 Ganglia 系统框架

Ganglia 监控套件包括三个主要部分：gmond，gmetad，和网页接口，通常被称为ganglia-web。

Gmond :是一个守护进程，他运行在每一个需要监测的节点上，收集监测统计，发送和接受在同一个组播或单播通道上的统计信息 如果他是一个发送者(mute=no)他会收集基本指标，比如系统负载（load\_one），CPU利用率。他同时也会发送用户通过添加C/Python模块来自定义的指标。 如果他是一个接收者（deaf=no）他会聚合所有从别的主机上发来的指标，并把它们都保存在内存缓冲区中。

Gmetad:也是一个守护进程，他定期检查gmonds，从那里拉取数据，并将他们的指标存储在RRD存储引擎中。他可以查询多个集群并聚合指标。他也被用于生成用户界面的web前端。

Ganglia-web :顾名思义，他应该安装在有gmetad运行的机器上，以便读取RRD文件。 集群是主机和度量数据的逻辑分组，比如数据库服务器，网页服务器，生产，测试，QA等，他们都是完全分开的，你需要为每个集群运行单独的gmond实例。

一般来说每个集群需要一个接收的gmond，每个网站需要一个gmetad。

HBase提供了集成了ganglia的配置文件。

### Zebbix

Zabbix 是由 Alexei Vladishev 开发的一种网络监视、管理系统，基于 Server-Client 架构。可用于监视各种网络服务、服务器和网络机器等状态。

使用各种 Database-end 如 MySQL， PostgreSQL， SQLite， Oracle 或 IBM DB2 储存资料。Server 端基于 C语言、Web 管理端 frontend 则是基于 PHP 所制作的。Zabbix 可以使用多种方式监视。可以只使用 Simple Check 不需要安装 Client 端，亦可基于 SMTP 或 HTTP 等各种协定做死活监视。

在客户端如 UNIX， Windows 中安装 Zabbix Agent 之后，可监视 CPU Load、网络使用状况、硬盘容量等各种状态。而就算没有安装 Agent 在监视对象中，Zabbix 也可以经由 SNMP、TCP、ICMP、利用 IPMI、SSH、telnet 对目标进行监视。

另外，Zabbix 包含 XMPP 等各种 Item 警示功能。

Zabbix特点：

1. 支持多语言(包括中文)。
2. 免费开源。
3. 主动发现[服务器](https://www.baidu.com/s?wd=%E6%9C%8D%E5%8A%A1%E5%99%A8&tn=24004469_oem_dg&rsv_dl=gh_pl_sl_csd)与网络设备。
4. 分布式监视以及web集中管理功能。
5. 可无agent监视。
6. 用户安全认证和柔软的授权方式。
7. 通过web界面设置或查看监视结果。
8. email等通知功能，并且兼容各种通知(电话，短信，微信，邮件等等)。

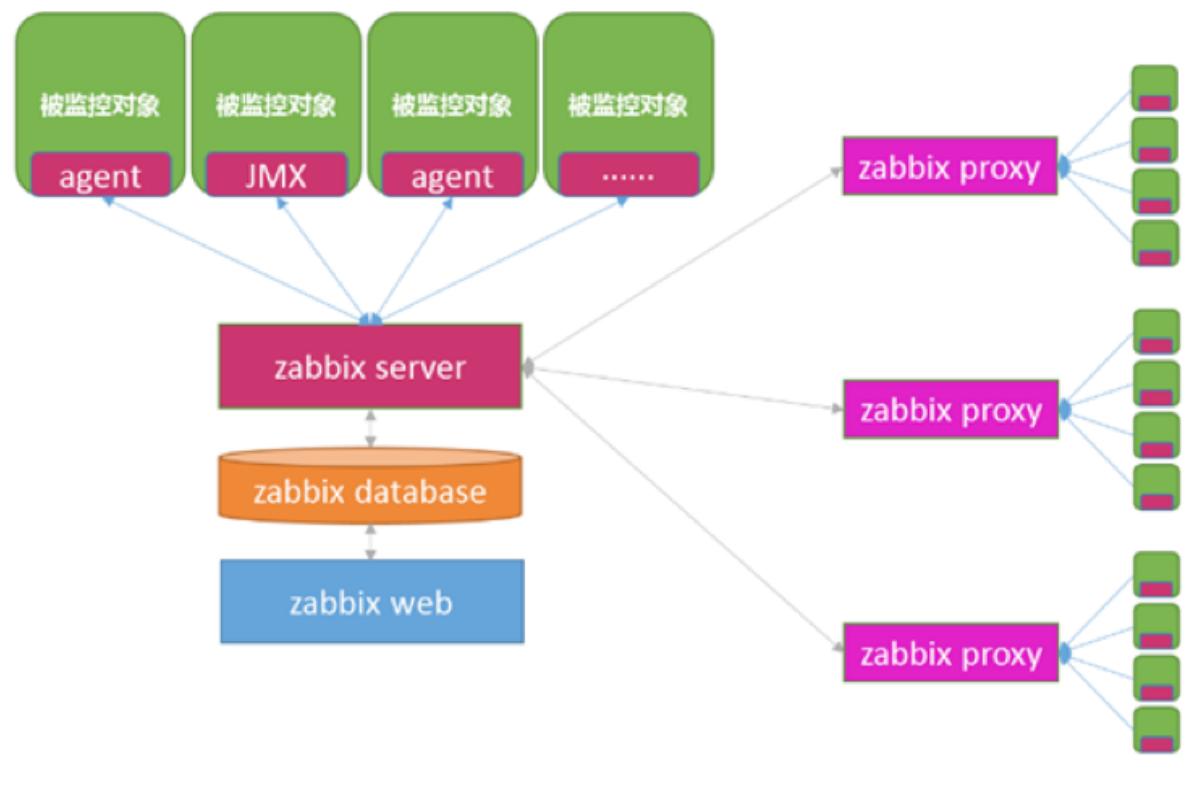


图 4‑4 Zabbix 系统框架

### CDH

CDH，Cloudera's Distribution， including Apache Hadoop，是Hadoop众多分支中的一种，由Cloudera维护，基于稳定版本的Apache Hadoop构建，提供了Hadoop的核心，同时支持可扩展存储、分布式计算并且提供基于Web的用户界面。

因此，CDH严格意义上并不是专门用来监控HBase的工具，而是具有Hadoop核心的一个分布式集群框架。此处只涉及用CDH部署的时候，对HBase的监控功能。

CDH提供的监控功能包括以下方面，

1. 操作系统层面，包括集群网络I/O、磁盘I/O和HDFS I/O，CPU和RAM。
2. 正在执行的MapReduce作业。
3. JAVA内核的指标，如GC情况。
4. 重要的HBase指标，如Region的数量和大小等。

### Ambari

Apache Ambari是一种基于Web的工具，支持Apache Hadoop集群的供应、管理和监控。Ambari已支持大多数Hadoop组件，包括HDFS、MapReduce、Hive、Pig、 HBase、Zookeeper、Sqoop和Hcatalog等。

Apache Ambari 支持HDFS、MapReduce、Hive、Pig、HBase、Zookeepr、Sqoop和Hcatalog等的集中管理。Ambari提供了对Hadoop更加方便快捷的管理功能，主要包含：

* + 1. 通过一步一步的安装向导简化了集群供应。
    2. 预先配置好关键的运维指标（metrics），可以直接查看Hadoop Core（HDFS和MapReduce）及相关项目（如HBase、Hive和HCatalog）是否健康。
    3. 支持作业与任务执行的可视化与分析，能够更好地查看依赖和性能。
    4. 通过一个完整的RESTful API把监控信息暴露出来，集成了现有的运维工具。
    5. 用户界面非常直观，用户可以轻松有效地查看信息并控制集群。

### Nagios

Nagios是一款开源的电脑系统和网络监视工具，能有效监控Windows、Linux和Unix的主机状态，交换机路由器等网络设置，打印机等。在系统或服务状态异常时发出邮件或短信报警第一时间通知运维人员，在状态恢复后发出正常的邮件或短信通知。

Nagios原名为NetSaint，由Ethan Galstad开发并维护至今。NAGIOS是一个缩写形式：“Nagios Ain't Gonna Insist On Sainthood” Sainthood翻译为圣徒，而"Agios"是"saint"的希腊表示方法。Nagios被开发在Linux下使用，但在Unix下也工作得非常好。

Nagios的功能是监控服务和主机，但是他自身并不包括这部分功能，所有的监控、检测功能都是通过各种插件来完成的。

启动Nagios后，它会周期性的自动调用插件去检测服务器状态，同时Nagios会维持一个队列，所有插件返回来的状态信息都进入队列，Nagios每次都从队首开始读取信息，并进行处理后，把状态结果通过web显示出来。

Nagios提供了许多插件，利用这些插件可以方便的监控很多服务状态。安装完成后，在nagios主目录下的/libexec里放有nagios自带的可以使用的所有插件，如，check\_disk是检查磁盘空间的插件，check\_load是检查CPU负载的，等。每一个插件可以通过运行./check\_xxx –h 来查看其使用方法和功能。

Nagios可以识别4种状态返回信息，即 0(OK)表示状态正常/绿色、1(WARNING)表示出现警告/黄色、2(CRITICAL)表示出现非常严重的错误/红色、3(UNKNOWN)表示未知错误/深黄色。Nagios根据插件返回来的值，来判断监控对象的状态，并通过web显示出来，以供管理员及时发现故障。

### OpenTSDB

OpenTSDB，OpenTSDB is a distributed， Scalable Time Series Database (TSDB) written on top of HBase，也就是基于HBase的分布式的，可伸缩的时间序列数据库。

主要用途，就是做监控系统；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。

openTSDB架构如下，

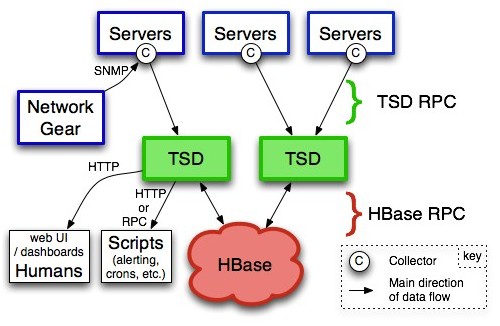


图 4‑5 OpenTSDB监控HBase

Servers：就是服务器了，上面的C就是指Collector，可以理解为OpenTSDB的agent，通过Collector收集数据，推送数据；

TSD：TSD是对外通信的无状态的服务器，Collector可以通过TSD简单的RPC协议推送监控数据；另外TSD还提供了一个web UI页面供数据查询；另外也可以通过脚本查询监控数据，对监控数据做报警；

HBase：TSD收到监控数据后，是通过AsyncHBase这个库来将数据写入到HBase；AsyncHBase是完全异步、非阻塞、线程安全的HBase客户端，使用更少的线程、锁以及内存，可以提供更高的吞吐量，特别对于大量的写操作。存储到OpenTSDB的数据，是以metric为单位的，metric就是1个监控项，譬如服务器的话，会有CPU使用率、内存使用率这些metric；

OpenTSDB使用HBase作为存储，由于有良好的设计，因此对metric的数据存储支持到秒级别；

OpenTSDB支持数据永久存储，即保存的数据不会主动删除；并且原始数据会一直保存（有些监控系统会将较久之前的数据聚合之后保存）。

## 能配合HBase存储的分布式协调组件

### Zookeeper

Zookeeper是一个分布式的，开放源码的分布式应用程序协调服务，是Google的Chubby一个开源的实现，它是集群的管理者，监视着集群中各个节点的状态根据节点提交的反馈进行下一步合理操作。最终，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

Zookeeper提供了文件系统、通知机制和Zookeeper文件系统。每个子目录项如 NameService 都被称作为znode，和文件系统一样，能够自由的增加、删除znode，在一个znode下增加、删除子znode，唯一的不同在于znode是可以存储数据的。

HBase使用Zookeeper作为分布式协调服务来维护集群中的服务器状态。Zookeeper维护哪些服务器处于活动状态并可用，并提供服务器故障通知。Zookeeper使用共识来保证共同的共享状态。请注意，应该有三到五台机器达成共识。

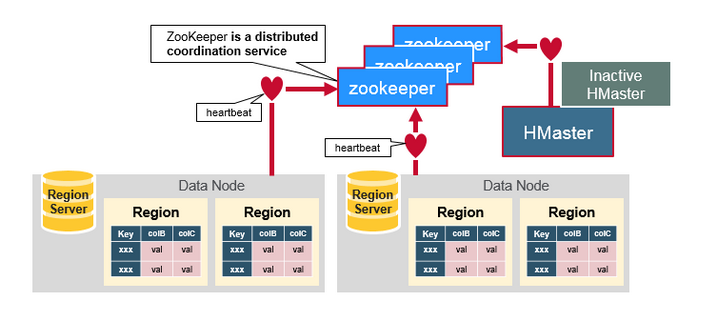


图 4‑6 HBase使用Zookeeper协同工作

Zookeeper用于协调分布式系统成员的共享状态信息。区域服务器和活动HMaster通过会话连接到Zookeeper。Zookeeper通过心跳维护活动会话的临时节点。

### ETCD

ETCD是用于共享配置和服务发现的分布式，一致性的KV存储系统。ETCD是CoreOS公司发起的一个开源项目，授权协议为Apache。ETCD作为一个受到Zookeeper与doozer启发而催生的项目，除了拥有与之类似的功能外，更专注于以下四点。

简单：基于HTTP+JSON的API让你用curl就可以轻松使用。

安全：可选SSL客户认证机制。

快速：每个实例每秒支持一千次写操作。

可信：使用Raft算法充分实现了分布式。

分布式系统中的数据分为控制数据和应用数据。ETCD的使用场景默认处理的数据都是控制数据，对于应用数据，只推荐数据量很小，但是更新访问频繁的情况。

ETCD 对比 Zookeeper：

一致性协议： ETCD使用[Raft]协议，Zookeeper使用ZAB（类PAXOS协议），前者容易理解，方便工程实现。

运维方面：ETCD方便运维，Zookeeper难以运维。

项目活跃度：ETCD社区与开发活跃，Zookeeper已经快死了。

API：ETCD提供HTTP+JSON， gRPC接口，跨平台跨语言，Zookeeper需要使用其客户端。

访问安全方面：ETCD支持HTTPS访问，Zookeeper在这方面缺失。

### Consul

Consul是google开源的一个使用go语言开发的服务发现、配置管理中心服务。内置了服务注册与发现框 架、分布一致性协议实现、健康检查、Key/Value存储、多数据中心方案，不再需要依赖其他工具（比如Zookeeper等）。服务部署简单，只有一个可运行的二进制的包。每个节点都需要运行agent，他有两种运行模式server和client。每个数据中心官方建议需要3或5个server节点以保证数据安全，同时保证server-leader的选举能够正确的进行。

Consul的三个主要应用场景：服务发现、服务隔离、服务配置。

服务发现场景中，Consul作为注册中心，服务地址被注册到Consul中以后，可以使用Consul提供的dns、http接口查询，Consul支持health check。

服务隔离场景中，Consul支持以服务为单位设置访问策略，能同时支持经典的平台和新兴的平台，支持tls证书分发，service-to-service加密。

服务配置场景中，Consul提供key-value数据存储功能，并且能将变动迅速地通知出去，通过工具Consul-template可以更方便地实时渲染配置文件。

与Zookeeper和ETCD不一样，Consul内嵌实现了服务发现系统，所以这样就不需要构建自己的系统或使用第三方系统。这一发现系统除了上述提到的特性之外，还包括节点健康检查和运行在其上的服务。

Zookeeper和ETCD只提供原始的键/值队存储，要求应用程序开发人员构建他们自己的系统提供服务发现功能。而Consul提供了一个内置的服务发现的框架。客户只需要注册服务并通过DNS或HTTP接口执行服务发现。其他两个工具需要一个亲手制作的解决方案或借助于第三方工具。

## 其他周边生态组件

### Kerberos安全认证组件

默认Hadoop各个组件间无任何认证，因此可以恶意伪装某一组件（比如NameNode）接入到集群中搞破坏。而通过Kerberos，可以将密钥事先放到可靠的节点上并只允许有限制的访问，该节点的服务启动时读取密钥，并与Kerberos交互以做认证，从而接入到Hadoop集群中。

Kerberos是一个基于共享密钥对称加密的安全网络认证系统，它避免了将密码（包括密码hash）在网上传输，而是将密码作为对称加密的密钥，通过能不能解密来验证用户的身份。Kerberos在验证完用户身份后会发给用户Ticket，这个Ticket包含了用户的授权，用户拿着这个Ticket去享受各种服务，所以在Kerberos管理的范围内用户只需要登录一次就可以享用所有的服务。

因为HBase的存储系统是基于Hadoop的存储，现在Hadoop已经增加了Kerberos认证机制，这样HBase的客户端访问HBase数据库的时候也需要进行身份的认证。Kerberos是一个认证中心，客户端在访问HBase前必须通过认证才能访问。

当HBase客户端访问HBase的时候，首先必须访问KDC获取一个经过授权的票据，以后Client在访问HBase server的时候可以通过这个票据进行访问。

正常情况下当通过HBase客户端访问的时候，都需要进行一次认证的过程，认证过后，KDC返回的票据具有有效期，一般默认是10小时，换句话说在这10个小时内你不需要再次登录KDC进行认证。

### GeoMesa地理数据处理套件

GeoMesa支持多种可扩展的基于云的数据存储技术，包括Accumulo， HBase和Bigtable，以及用于流数据的Kafka消息代理。Storm允许您定义信息源和操作，以允许使用GeoMesa批量分布式处理流数据，GeoMesa环境还可以利用Spark对存储和流数据进行大规模分析。

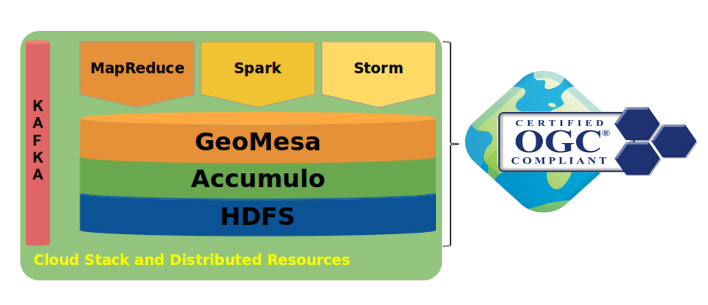


图 ‑7 GeoMesa 系统框架

### JanusGraph图数据库

Titan在停止更新了很长一段时间后，fork出了JanusGraph继续开源发展。JanusGraph是一个图形数据库引擎。JanusGraph本身专注于紧凑的图形序列化、丰富的图形数据建模和高效的查询执行。此外，JanusGraph利用Hadoop进行图形分析和批处理图处理。JanusGraph实现了健壮的模块化接口，用于数据持久性、数据索引和客户端访问。JanusGraph的模块化体系结构允许它与广泛的存储、索引和客户端技术进行互操作；它还简化了扩展JanusGraph以支持新用户的过程。

在JanusGraph和磁盘之间，有一个或多个存储和索引适配器。JanusGraph以以下适配器为标准，但是JanusGraph的模块化体系结构支持第三方适配器

JanusGraph 特点如下：

1. 支持大规模图数据存储，Titan图数据库是建立在分布式集群上，数据存储容量和集群节点数量成正比。

2. 支持弹性和线性扩展，高可用，高容错。

3. 支持Gremlin图查询语言。

4. 支持利用Hadoop计算框架对图数据进行分析。

5. 支持外部索引：ElasticSearch、Solr、Lucene。

6. 支持多储存引擎：Cassandra、HBase、Berkeley DB和InMemory模式。

7. 基于apache license 2.0。



图 ‑8 JanusGraph系统架构

## 小结

随着大数据在车联网、风控和各种数据分析等场景中真实落地，支持Hadoop生态的周边生态组件越来越多，也促使大数据技术进一步地向前发展。

# HBase应用场景及案例

## 阿里

2018年6月6日，阿里云ApsaraDB for HBase2.0正式发布。从2010年开始“试水”到2018年，拥有了3个PMC，6个Committer，阿里巴巴成为额拥有中国最多HBase Committer的公司之一。HBase作为开源组件，在很多企业级软件功能上的不足是需要进行二次开发的。

### HBase在阿里的使用场景

Ali-HBase作为阿里巴巴技术大厦的基础存储设施，全面服务于淘宝、天猫、蚂蚁金服、菜鸟、阿里云、高德、优酷等各个领域，满足业务对于大数据分布式存储的基本需求。

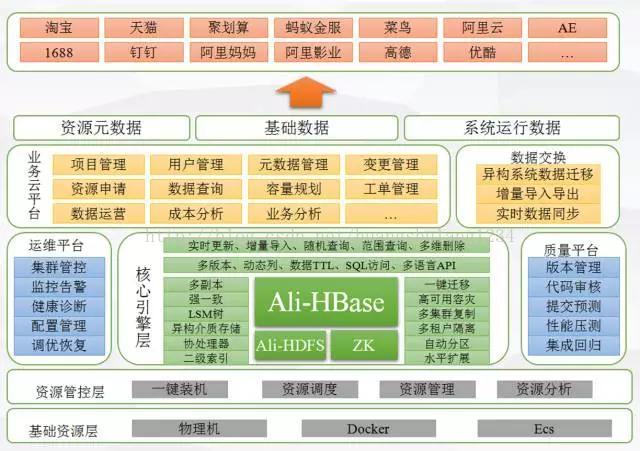


图5- 1 阿里HBase生态整体框架

### 阿里对HBase的开发

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。

#### 不同产品形态的HBase

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。

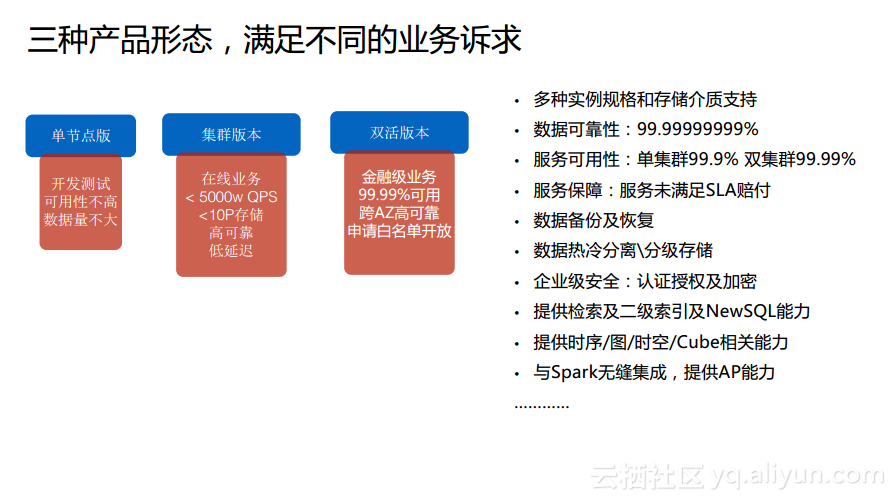


图5- 2 阿里云HBase

#### 重新定义HBase能力

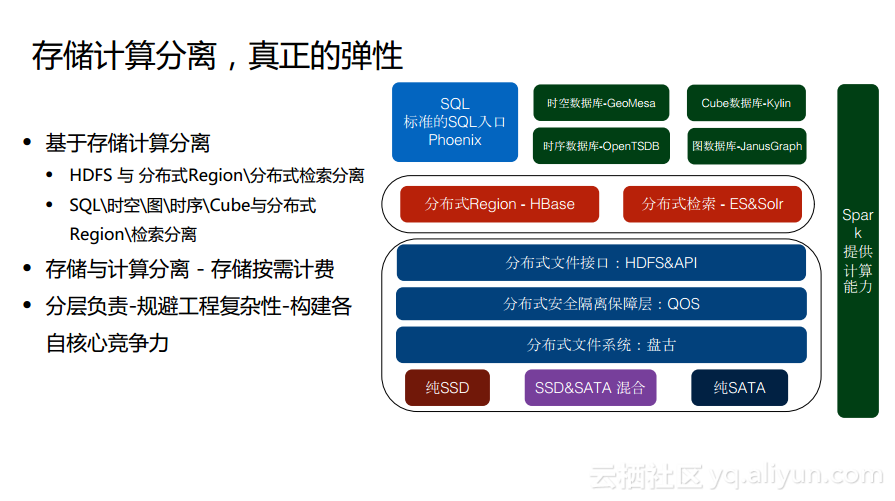


图5- 3 阿里重新定义HBase

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。

#### 多重防护机制，企业级安全

开源版本的HBase基本上没有安全能力，完全属于“裸奔”状态。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。

#### 其他开发

1. 优化内核级，全面提升性能和稳定性；
2. **全量和增量备份以及恢复；**
3. 对HBase进行自动化运维研发。

## 滴滴[[1]](#footnote-1)

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。

### HBase在滴滴的使用场景

#### 场景一：订单事件

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5s内可用。

按照这些要求，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

#### 场景二：司机乘客轨迹

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。

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. 满足一个指定的地理范围，取出范围内所有用户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，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，滴滴使用一套比较通用地理索引的GeohHash，基于HBase算法实现了相同的索引功能。

两种查询场景的Rowkey设计如下：

单个用户按订单或时间段查询： reverse(user\_id) + (Integer.MAX\_LONG-TS/1000)

给定范围内的轨迹查询：reverse(geohash) + ts/1000 + user\_id

#### 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：

模型训练通过Spark Job，每30分钟对各个城市训练一次；

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；

模型训练第二阶段在25分钟内完成ETA的计算；

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

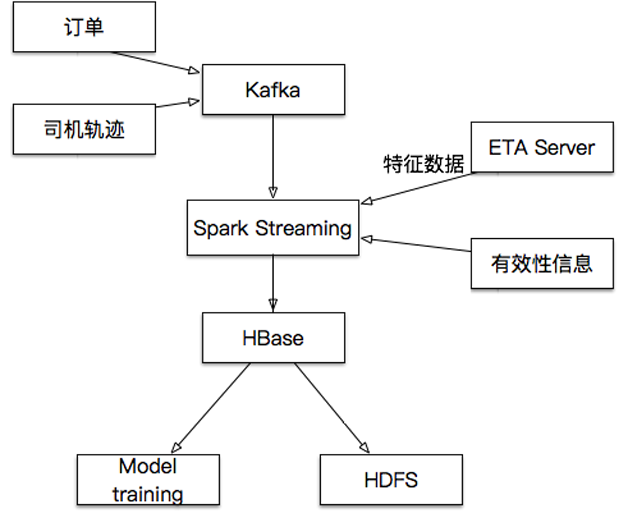


图5- 4 ETA数据流程图

#### 场景四：监控工具DCM

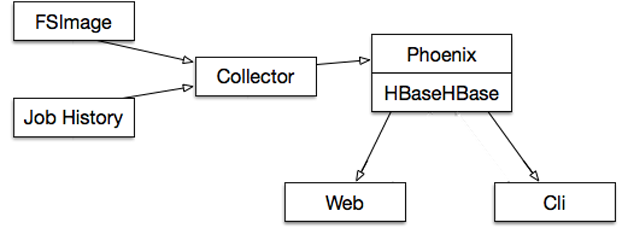


图5- 5 DCM监控

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。

### 滴滴基于HBase的开发工作

#### 滴滴在HBase对多租户的管理

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；滴滴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，滴滴开发了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。

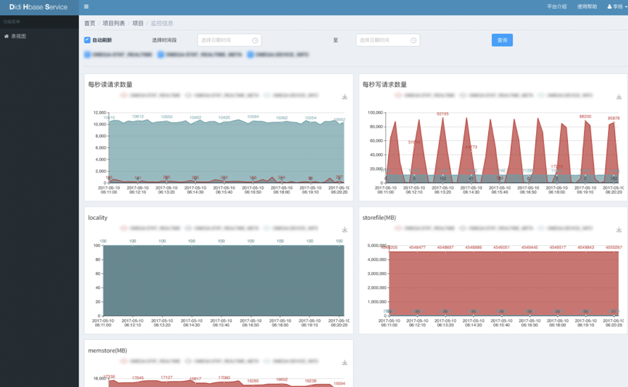


图5- 6 DHS项目表监控

DHS主要有下面几个模块和功能：

1. 项目生命周期管理：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；

表级别的使用情况监控：主要是读写监控、memstore、blockcache、locality。

表5- 1 多租户共享和独占资源的优缺点

|  |  |  |
| --- | --- | --- |
|  | 优点 | 缺点 |
| 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 |

根据以上的情况，在资源分配上会根据业务的特性来选择不同方案：

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；

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RegionServerGroup资源，并且按用户预估的资源量，额外给出20%~30%的余量。

最后，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。

#### RS Group

RegionServerGroup，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。

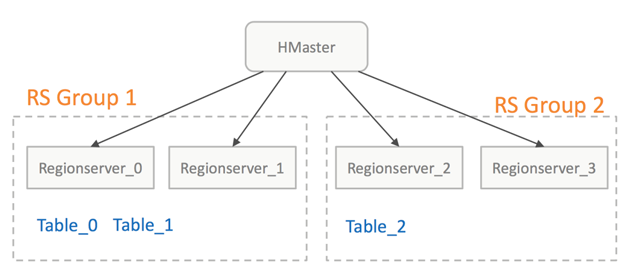


图5- 7 RS Group示意图

## 58同城

### HBase在58同城的使用场景

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。

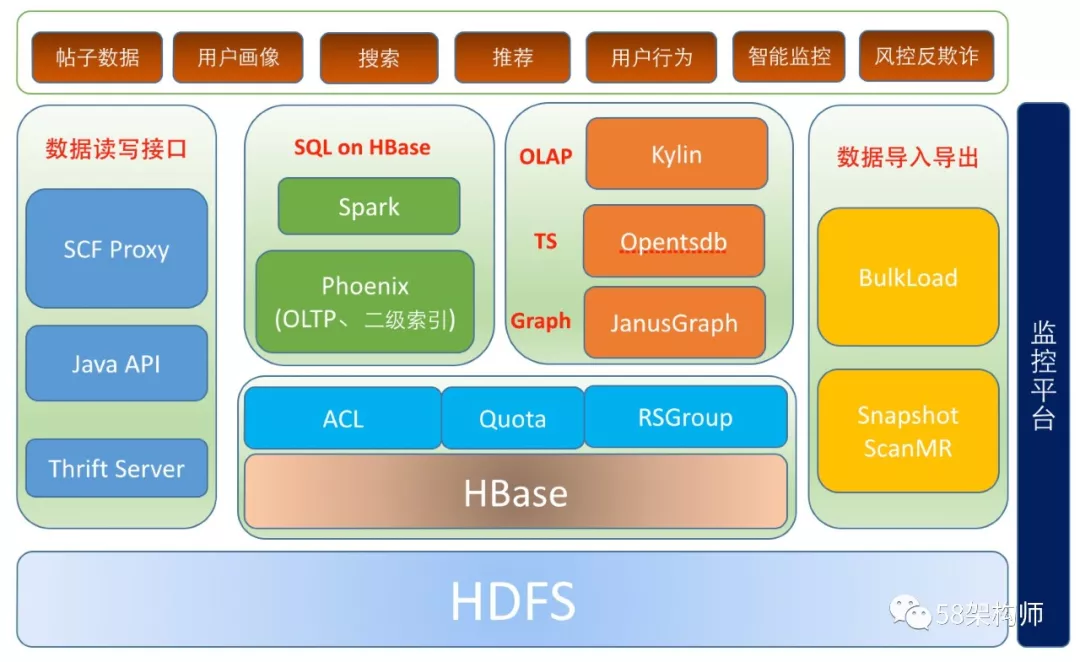


图5- 8 HBase在58的应用架构

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。

### 58同城基于HBase的开发

和滴滴相同，58同城基于HBase根据自己的业务逻辑，基于HBase进行二次开发，主要工作包括多租户支持、数据读写接口、数据导入导出和平台优化。

#### 多租户支持

因为58同城的业务面相较滴滴更加多样，所以其从多个层面对HBase 多租户进行了支持，主要分为以下两个大的方面：

资源限制：

SCF Quota；

HBase Quota。

资源隔离：

RS RPC读写分离；

HBase ACL权限隔离；

RSGroup物理隔离。

#### **数据读写接口**

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。

1. SCF Proxy

SCF是58架构部自研的RPC框架，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。

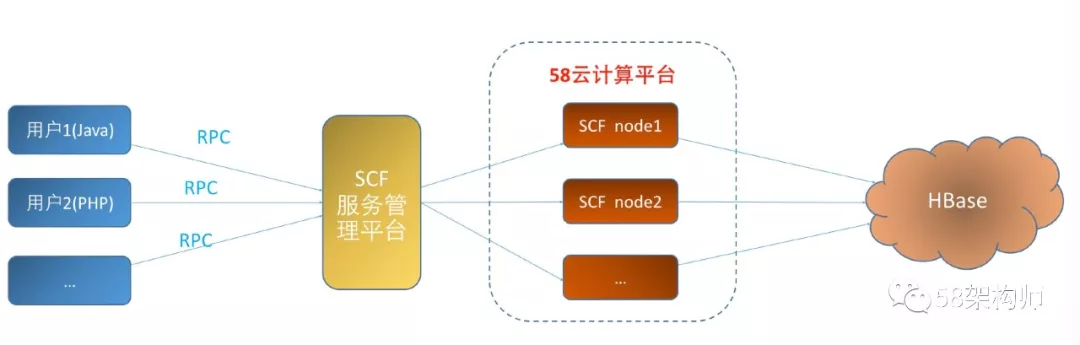


图5- 9 SCF proxy接口整体架构

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：

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。

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。

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。

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。

#### 数据导入导出

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，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，会建议用户通过BulkLoad的方式导入数据。

1. 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HBase原生提供的MR来并行化对数据进行扫描。

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：

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。

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，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。

## 小结

HBase最先由Fackbook使用而得到极大发展，国内大型互联网公司在大数据平台架构中均使用了HBase作为海量存储组件。阿里和华为对Hadoop生态群进行二次开发，当中也包含HBase的易用性开发，并包装成阿里云和华为云生态系统中一部分对外服务。另一些互联网公司，根据自身产品业务逻辑，也都基于HBase进行海量数据存储，并取得了很好的效果。

1. 以下内容，参考CCTC 2017大数据峰会上所做分享内容 [↑](#footnote-ref-1)