



# HBase最佳实践及优化

陈飏

[cb@cloudera.com](mailto:cb@cloudera.com)

Cloudera

# 关于我...

陈飏

Cloudera售前技术经理、资深方案架构师

<http://biaobean.pro>



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

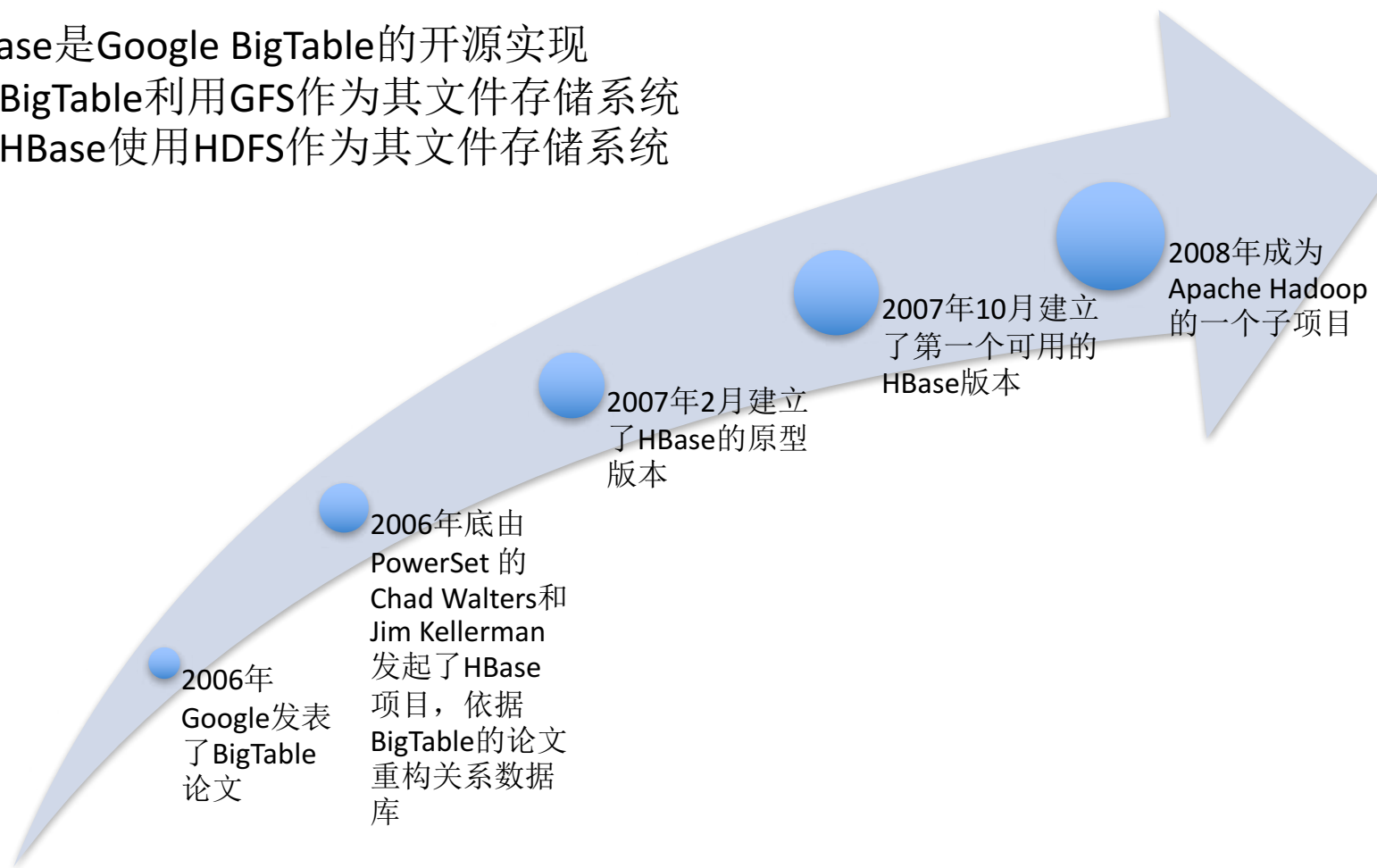
- 曾在Intel编译器部门从事服务器中间件软件开发, 擅长服务器软件调试与优化, 与团队一起开发出世界上性能领先的XSLT 语言处理器
- 2010 年后开始Hadoop 产品开发及方案顾问, 先后负责Hadoop 产品化、HBase 性能调优, 以及行业解决方案顾问



# HBase的历史

HBase是Google BigTable的开源实现

- BigTable利用GFS作为其文件存储系统
- HBase使用HDFS作为其文件存储系统



# HBase的模型特性

Hadoop database and NoSQL database

- 基本的数据库操作CRUD
- 强一致性
- 无SQL语言支持
- 稀疏的多维映射表
  - 列存储
  - 只用row key来定位行
  - 每行可以有不同的列
  - 数据有多个版本（在不同的时间点的快照信息）
- 分布式的多层次映射表结构（key-value形式，value有多个）
  - 固定一个数据模型（固定数据模型能得到高性能，同时满足应用需求）
  - 无数据类型



# HBase的实现特性

- 非常高的数据读写速度，为写特别优化
  - 高效的随机读取
  - 对于数据的某一个子集能够进行有效地扫描
- 具有容错特性，能够将数据持久化的非易失性存储中
  - 使用HDFS做底层存储，可利用Hadoop的压缩Codec等减少空间占用
- 自动水平扩展
  - 只需要加入新的结点即可提高存储容量和吞吐量
  - 服务器能够被动态加入或者删除（用以维护和升级）
  - 服务器自动调整负载平衡



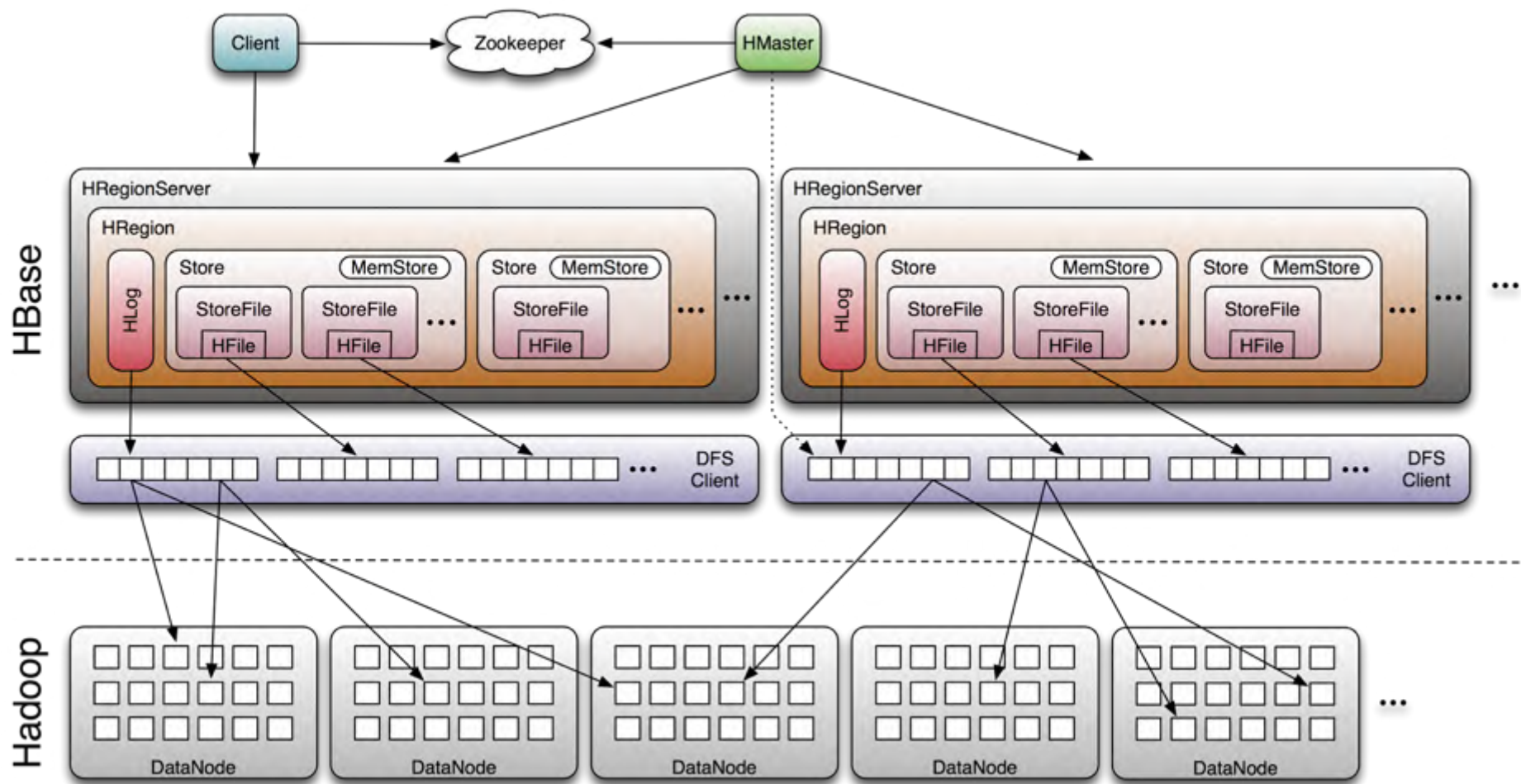
# HBase的原子性保证

## HBase仅保证对行操作的原子性

- 任何行级的操作是原子的
  - 一条记录的Put操作要么完全成功，要么完全失败。
    - 操作返回成功(success)表示操作完成
    - 操作返回失败(failure)表示操作全部失败
    - 超时操作可能是成功也可能失败，但不可能部分成功
  - 即使跨column family的行操作也是原子的
- 支持一次性修改多行的API并不保证跨行的原子性操作
  - 一般情况下，API会在结果中分别返回执行成功、失败以及超时的行操作列表



# HBase体系结构



# 场景及应用





# HBase Sweet Spot

1. 使用主流廉价服务器搭建的单一**大规模**集群  
(服务器数目大于100甚至1000台)
  2. 小规模**Scan**操作(<1百万行)和**Get**操作
  3. 运维难度大, 大规模部署后单位运维成本低
  4. **强一致性**、**开源**、兼容私有部署/公有云部署
  5. 通用的低延迟的基础存储引擎
- 尚未有系统同时很好地处理分析和OLTP任务
  - 在HBase擅长的场景至今尚未有可替代品



# 典型用户案例：Data Storage

- 场景
  - 用于收集并存储非结构化以及半结构化数据
  - 数据存储要求可靠
  - 保证数据强一致性
  - 数据可被排序以便提供低延时的随机查询
- 案例
  - 原始日志查询系统
  - 在线指标查询系统
- 主要组件
  - HBase, Flume, Sqoop



# HBase适用场景

- **高并发高性能读写访问场景**
  - 数据有随机更新、删除
  - 数据写入性能高于读取性能，适合写多读少或数据加载有实时性要求的场景
- 需**按主键排序**的半结构化数据存储
- 支持基于固定有限条件的高并发高性能**查询**
- 高速计数器aggregation类型的任务
  - HBase强一致性(Strongly consistent)读写保证
- 其他适用Hadoop的NoSQL场景
  - HBase基于HDFS存储，和MapReduce/Hive/Spark等紧密结合

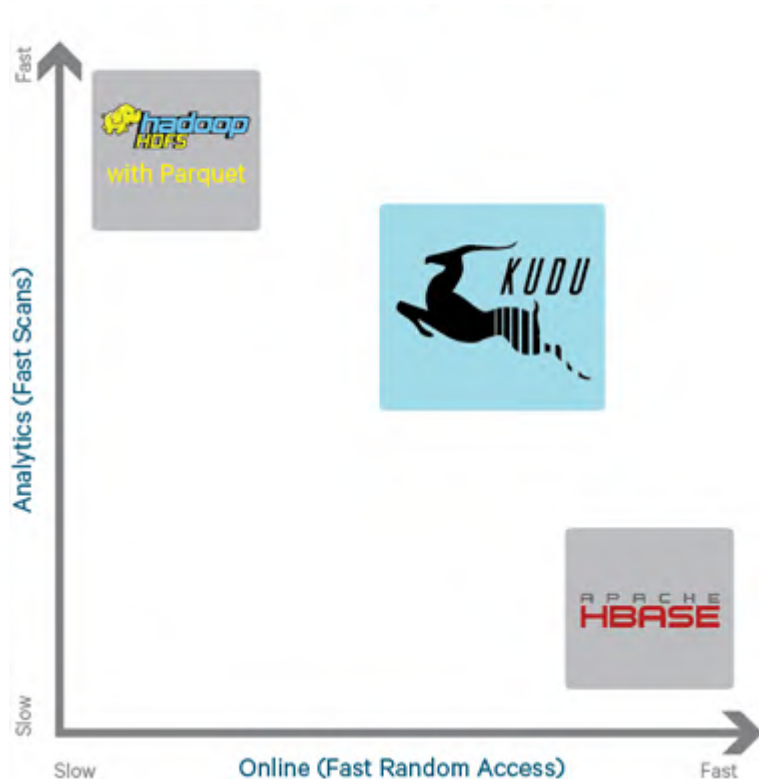


# HBase现存缺点

- SQL(传统BI)不友好，不支持很多传统DBMS功能，如外键，约束...
- 数据无类型
- 非RowKey查询性能差
- Column Family限制(数目，Partition对齐)
- Region资源消耗大，实例数目不能太多
- 无法保证服务质量\*
  - Split/Compaction等操作对集群性能影响极大
- 多租户隔离能力差
- 大内存(>100GB)管理差



# Kudu的设计目标



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

28日下午 15:40 - 16:20

**Hadoop最新结构化存储利器Kudu介绍**  
分会场2



# 案例：运营商清帐单系统关键需求

- 必须能够高效处理海量数据
  - 单月清单数据量约1000亿条  $\times$  1k/条=100TB，6个月总量高达~600TB
  - 从600TB清单数据中检索某用户某个月的清单记录，响应时间应小于1秒
  - 支持高峰期每秒2000个并发访问查询
  - 满足现在清帐单业务的查询统计需求(23类)
  - 实时入库，清单文件无积压（清单文件最大2万条，最小1条记录。实时生产，平均每秒2个20MB的清单文件，高峰期到每秒10个20MB文件）
  - 对联机分析必须提供标准编程接口，支持SQL/JDBC/ODBC等
- 高可扩展和高可用
  - 用户程序查询数据不需要知道底层细节，比如数据分布细节
  - 可以水平扩展
  - 允许多台机器故障的场景下，业务不中断



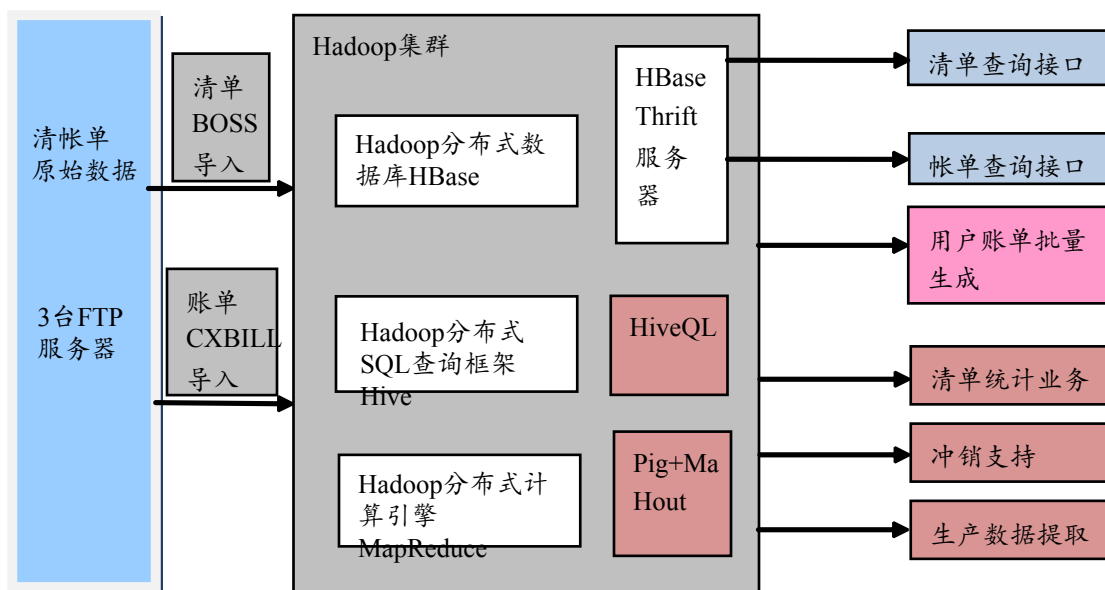
# 原有方案：小型机+存储+Oracle

- 成本高、扩展性差
  - 价格昂贵
    - 服务器采用P595的两个分区（48CPU），部署不同的地市，互为主备
    - 存储使用2台DS8300，RAID5方式，有效容量54TB
  - 数据量大，增长迅速，但数据库的扩容工程施工风险高
- 数据风险高
  - 灾难恢复依赖磁带，业务中断时间长
- 效率低
  - 关系数据库处理困难，查询慢（超过15秒）
  - 关系数据库入库慢，常有清单文件积压，不能实时入库，从而不能实时查询



# 基于Hadoop的清帐单系统架构

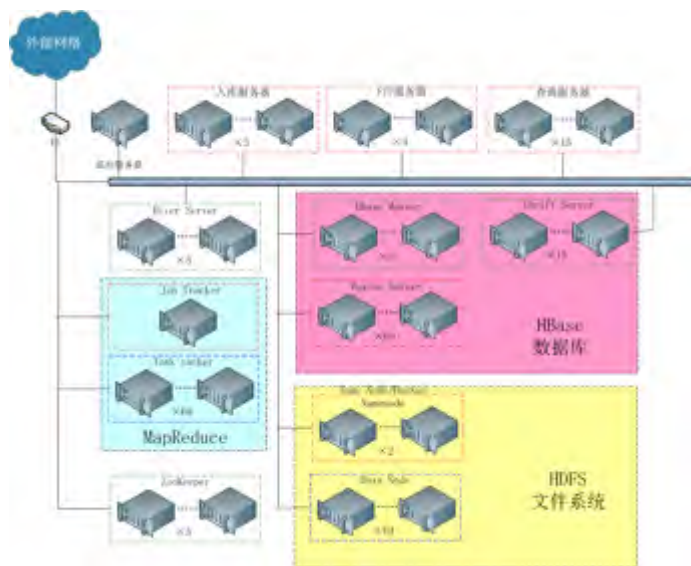
新版清帐单系统采用了基于Hadoop的大数据平台，使用分布式文件系统HDFS,数据存储则采用了分布式数据库HBase，同时结合云计算的其他组件构成





# 部署方案

- 底层通过78台X3650 PC服务器组构建Hadoop集群,有效容量138TB
- 数据的分发、复制、任务调度、容错都是由系统软件来控制,同时具备线性的横向扩展能力
- 3份冗余的数据保证对硬件的容错和读处理的支持



| 设备                                 | 硬件设备   | 数量          |
|------------------------------------|--|-------------|
| Hadoop 集群管理节点                      | IBM 3650 PC, 双路六核, Intel X5650处理器, 2.66GHz主频, 48GB内存, 6*1TB SATA硬盘 | 1台          |
| Hadoop集群 NameNode/JobTracker       | IBM 3650 PC, 双路六核, Intel X5650处理器, 2.66GHz主频, 48GB内存, 6*1TB SATA硬盘 | 1台          |
| NameNode/JobTracker HA备份节点         | IBM 3650 PC, 双路六核, Intel X5650处理器, 2.66GHz主频, 48GB内存, 6*1TB SATA硬盘 | 1台          |
| Secondary NameNode                 | IBM 3650 PC, 双路六核, Intel X5650处理器, 2.66GHz主频, 48GB内存, 6*1TB SATA硬盘 | 1台          |
| HBase 集群Master和Zookeeper节点         | IBM 3650 PC, 双路六核, Intel X5650处理器, 2.66GHz主频, 48GB内存, 6*1TB SATA硬盘 | 5台          |
| DataNode/TaskTracker/Region Server | IBM 3650 PC, 双路六核, Intel X5650处理器, 2.66GHz主频, 48GB内存, 6*1TB SATA硬盘 | 69台         |
| HBase Thrift服务器节点/查询服务器            | IBM 3650 PC, 双路六核, Intel X5650处理器, 2.66GHz主频, 48GB内存, 6*1TB SATA硬盘 | 15台(使用集群节点) |
| 入库服务器                              | IBM 3650 PC, 双路六核, Intel X5650处理器, 2.66GHz主频, 48GB内存, 6*1TB SATA硬盘 | 3台(不属于集群节点) |
| FTP服务器                             | IBM 3650 PC, 双路六核, Intel X5650处理器, 2.66GHz主频, 48GB内存, 6*1TB SATA硬盘 | 3台(不属于集群节点) |



# 案例2: 上网记录集中查询与分析

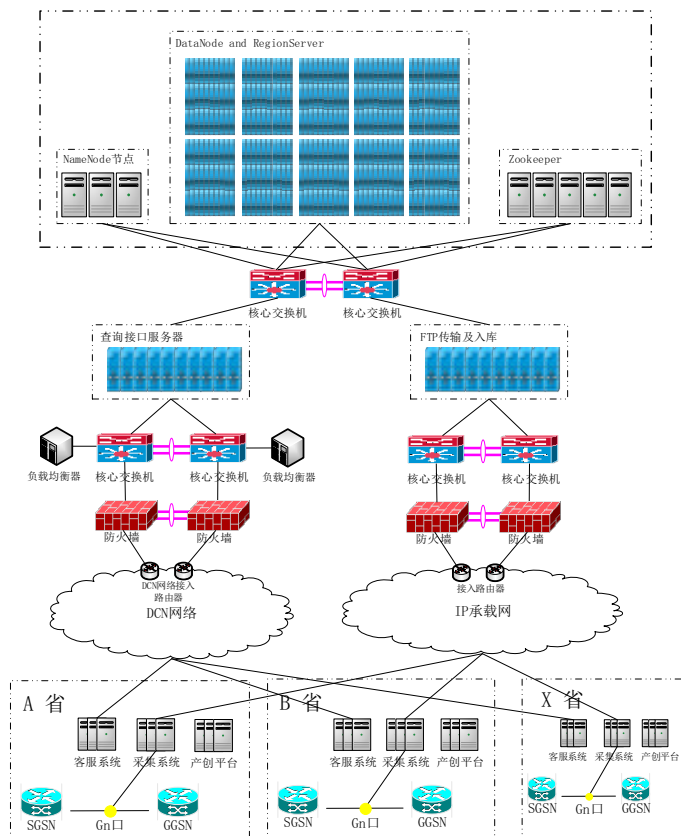
- 采用**全国集中的一级架构**方案进行建设
- 主要包含数据采集子系统、数据入库子系统、数据存储子系统、数据查询与分析子系统
- 采用Hadoop/HBase作为上网记录存储方案
- 采用MapReduce/Hive作用统计分析和数据挖掘工具

## 【关键性能指标】

- 每日入库**>5TB**数据
- 上网记录入库时间：一般小于30分钟，实际约**10分钟**
- 存储全国移动用户不小于6个月的原始上网记录，统计分析中间报表数据保存不小于5年
- 上网记录查询速度：不高于1秒（不含用户访问查询页面的时间）
- 支持并发查询数目：1000请求/秒



# 系统部署

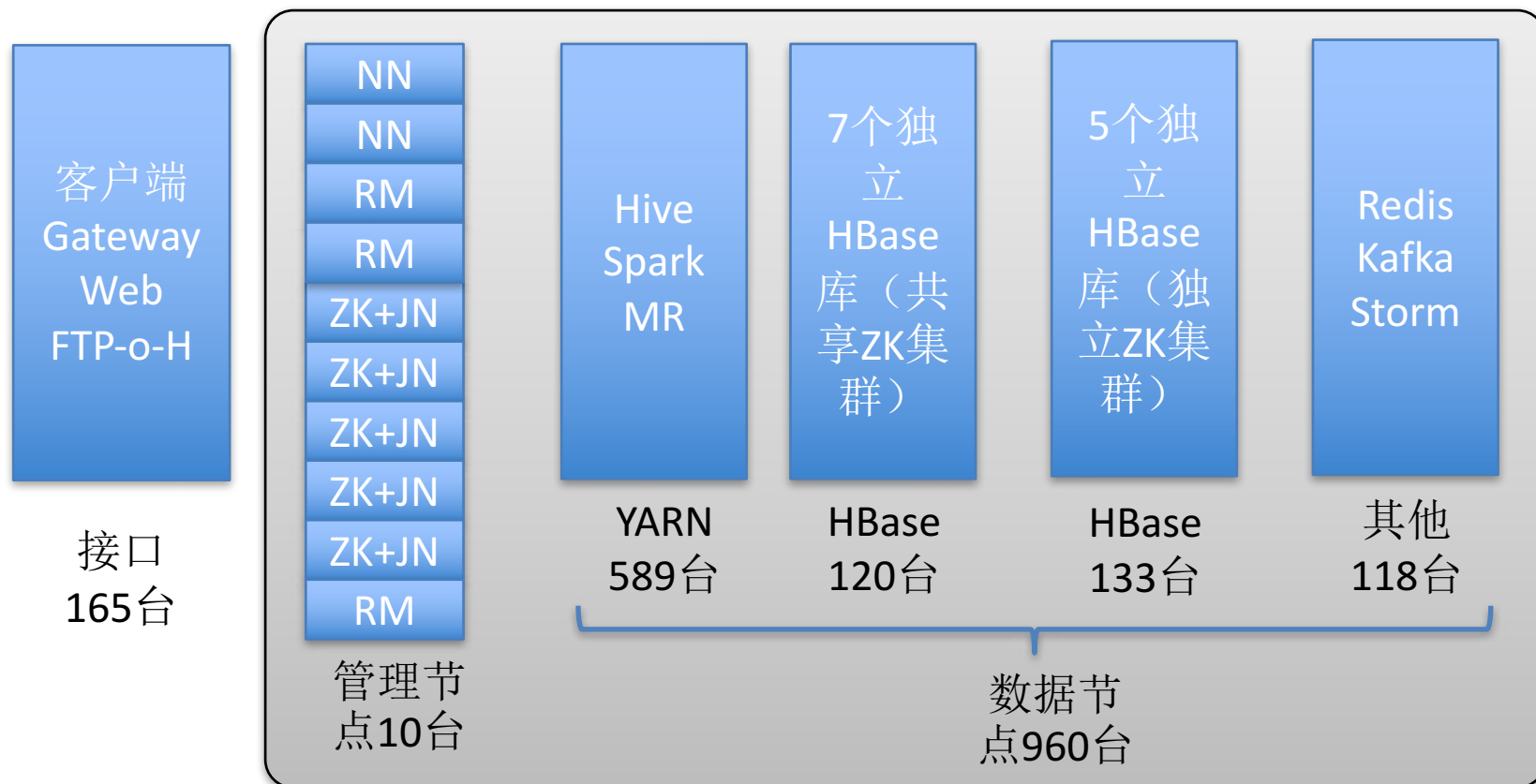


- NameNode节点：3台
- **DataNode（数据存储节点）：178台**
- Zookeeper节点：7台
- 集群监控节点：1台
- 入库服务节点：24台
- Web查询应用服务节点：20台
- 机架间通过万兆交换机连接
- 网络冗余



# 某大规模HBase多用户服务平台

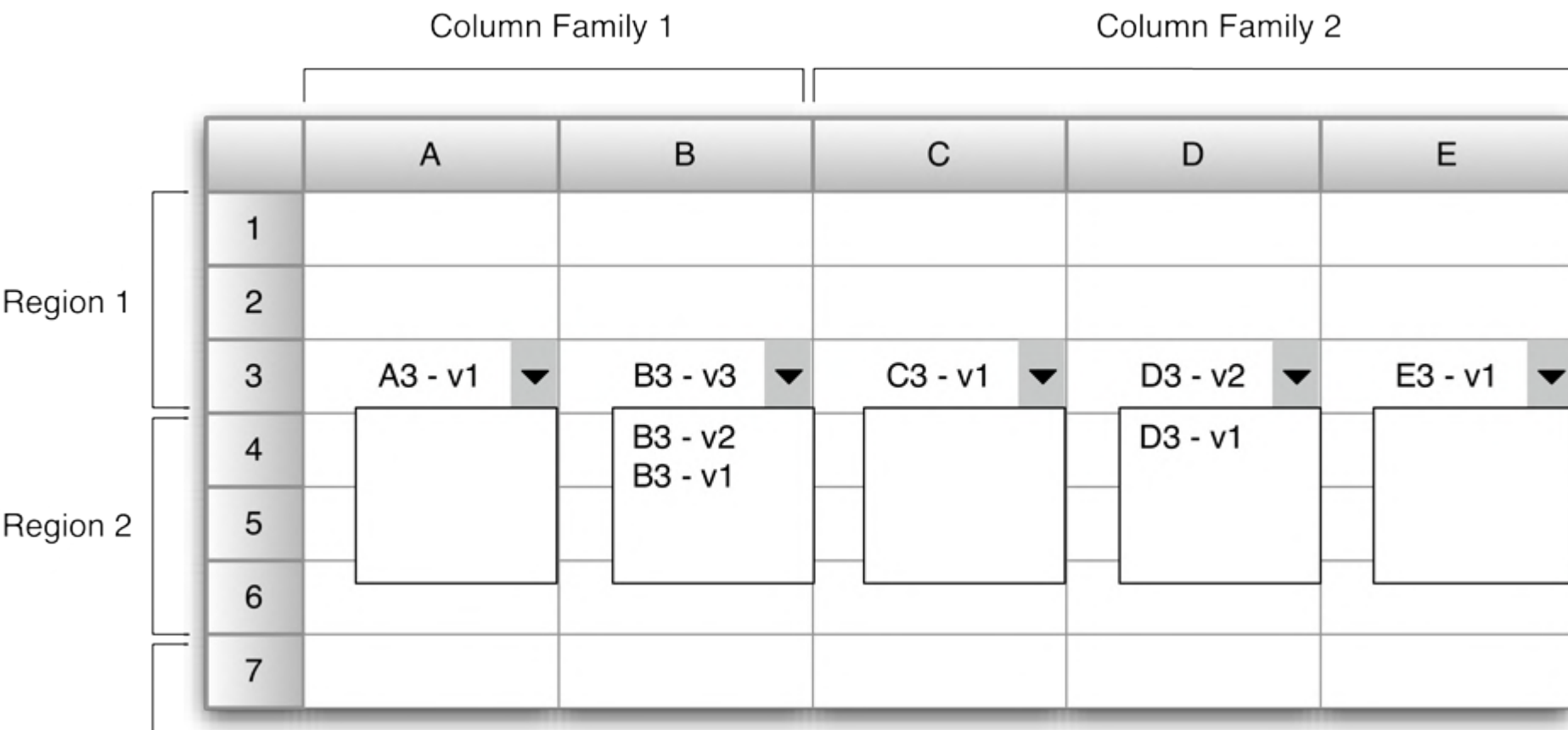
- Hadoop平台集群**1135**个节点，其中集群970，客户端165，总存储16PB，每日采集数据压缩后40TB，集群数据块4700万个



# 开发指南



# HBase表结构逻辑图



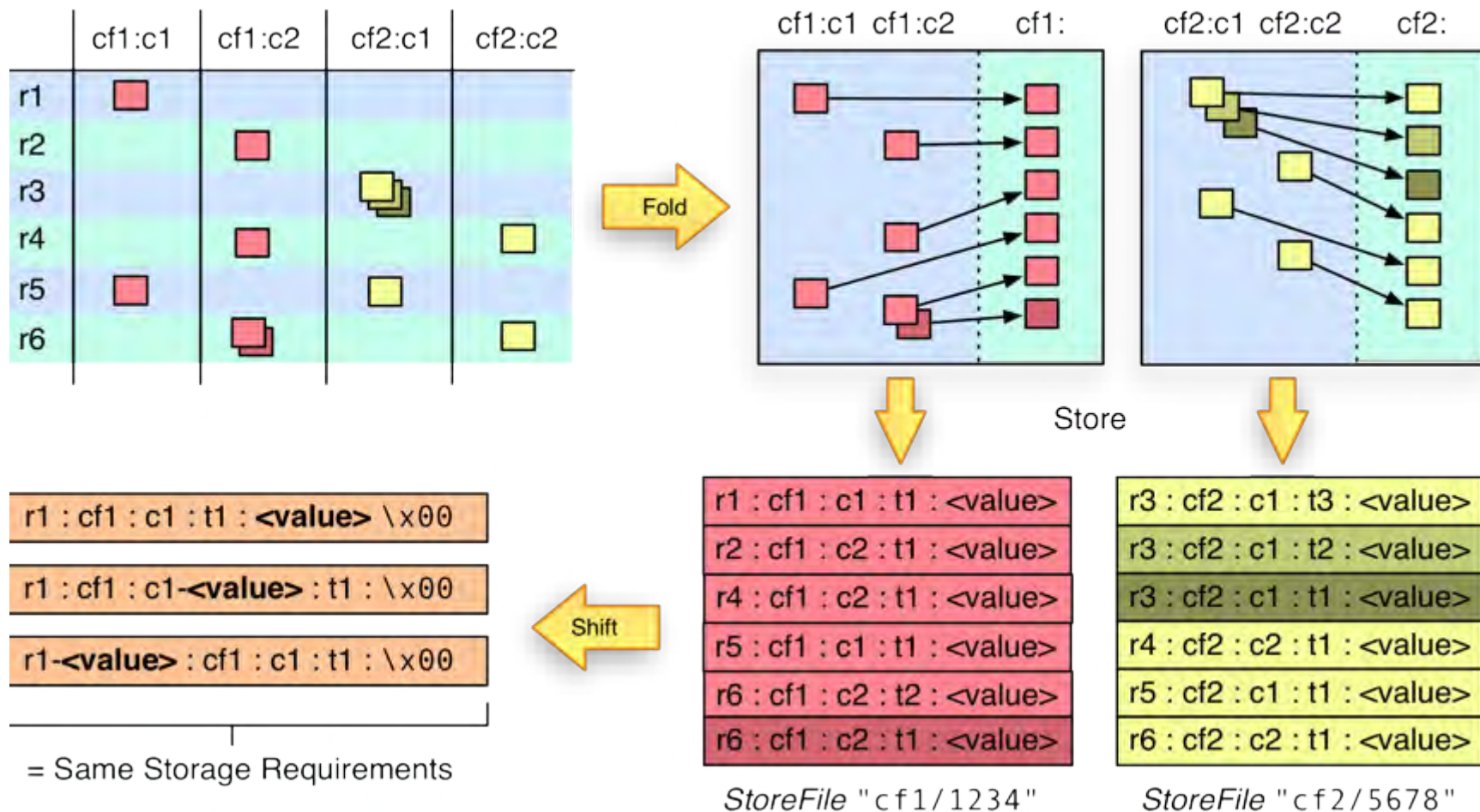
# HBase基础

- 表记录按RowKey字典序存储
- 表Schema只定义到Column Family级别属性
  - 每个Column Family可以有任意多个Column
  - 每个Column有可以有任意多个版本(version)的数据
  - Column只有在有赋值时才被真正存储，NULL值无存储消耗
  - 一个Column Family内的Column统一存储并排序
  - 除表名外所有数据皆为无类型数据(byte数组)



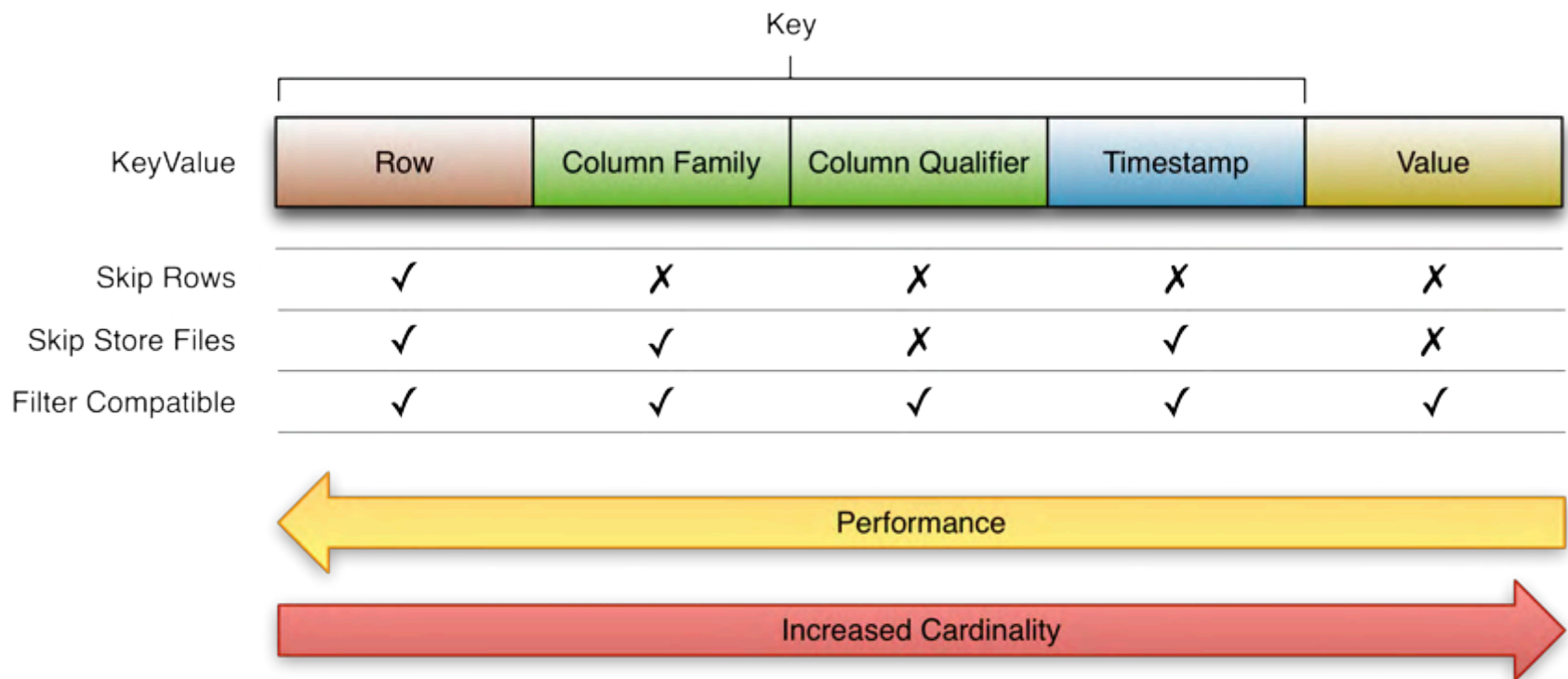


# HBase数据模型





# Key粒度及性能



# Key对数据查询的影响

- 使用RowKey进行查询的性能最好
- 指定Timestamp能减少store file级别的读操作
  - Bloom Filter也能达到同样目的
- 选择指定的Column Family可以减少查询需要读取的数据量
- 简单的纯基于filter的值查找是一个全表扫描操作
  - 但使用filter可以减少网络传输数据量



# 关系型数据库中的数据模型

- Entity
- Attribute
- Relationship

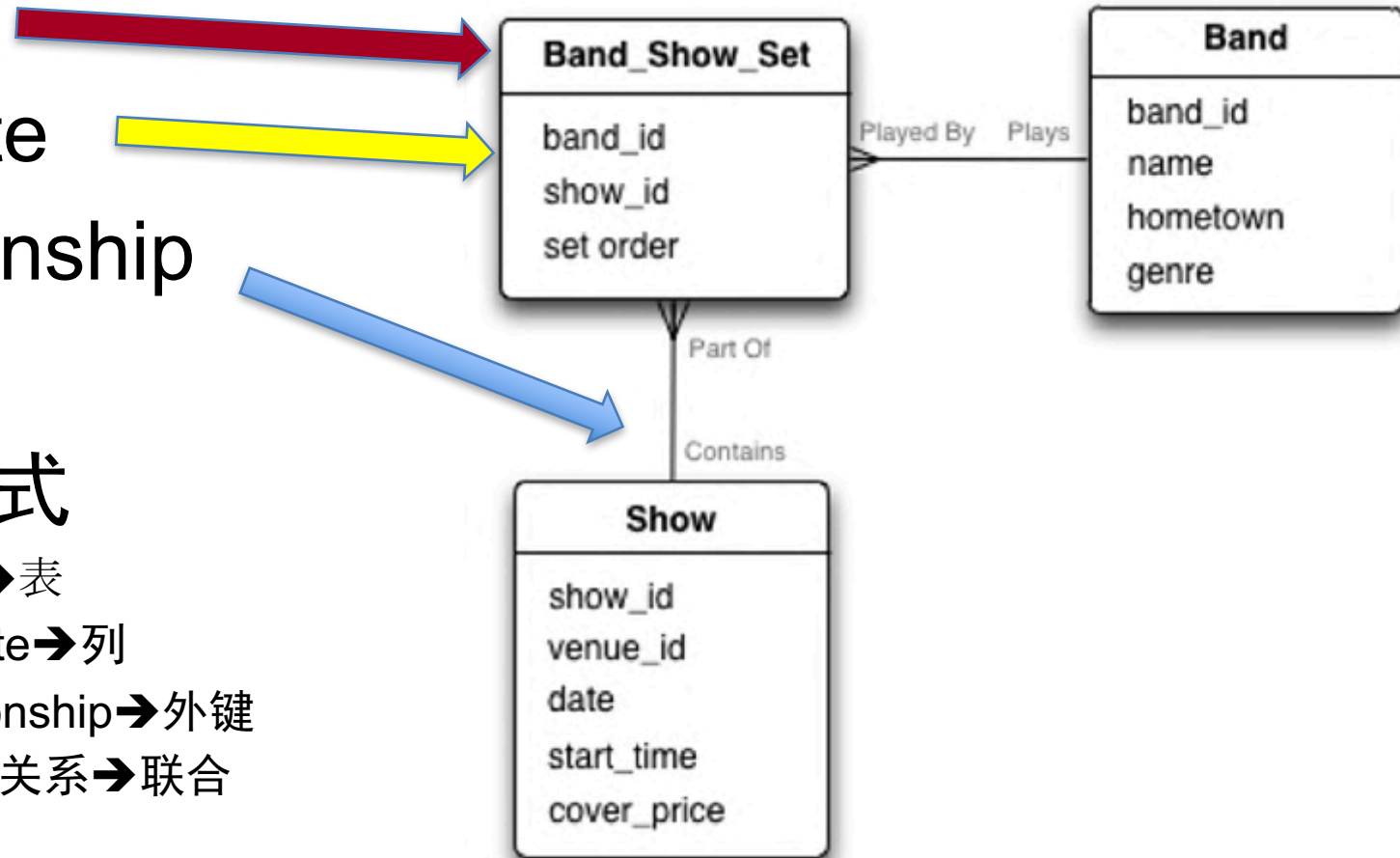
- 第四范式

Entity → 表

Attribute → 列

Relationship → 外键

多对多关系 → 联合



# NoSQL数据库的典型数据模型 (muddle)

- 将所有数据关系放在一行宽记录中存储
  - 避免数据查找及跨网络访问
  - 随机数据读取变为顺序读取
  - 易于分表
  - 空间换时间
  - 原子性更新?
- 第三范式

| Band_Show_Set    |
|------------------|
| show_id          |
| set order        |
| band_id          |
| band_name        |
| band_hometown    |
| band_genre       |
| show_venue_id    |
| show_date        |
| show_start_time  |
| show_cover_price |



# 行记录设计

- 采用多行存储还是单行多列存储？
  - 将数据另存储为一行还是“覆盖”存储为列(Column)的不同版本
  - 将数据另存储为一行还是增加一个列
- 建议
  - 通常情况下的回答: 分行存储
    - » 能获得更好的Get以及scan的性能
    - » 太长的行记录不利于做Region的split
  - 行设计必须符合数据原子性操作要求
    - » HBase只保证行级别数据的原子性操作



# RowKey设计

- RowKey的组成元素
  - 尽量将所有常用查询所使用的域放入RK
    - » 优先使用RK filter, 其次使用value filter
  - 保持RK值得唯一性 (添加序列号)
  - RK长度越短越好 (通用数据定义规则, 适用于其他)
    - » 考虑KeyValue的物理存储规则
  - 一般建议RK长度 < 50B
- RK的设计首要考虑便于能将最常用的查询转化为HBase的get或者基于RK范围的scan操作
  - 最重要的域放在首位, 依次...



# 单调递增序列数据/时间序列数据

- RK = <time> <other-keys>
- **不好!**
  - 单Region成热点Region
- 解决方案:
- 为单调值添加Hash函数使其分布平均, 具体Hash方法包括
  - 基于节点信息等静态配置信息 (静态Hash)
  - 基于其他数据信息, 如其他key值 (动态Hash)
  - 直接使用RowKey Hash值 (Random)
  - 值交织 (Interwave)

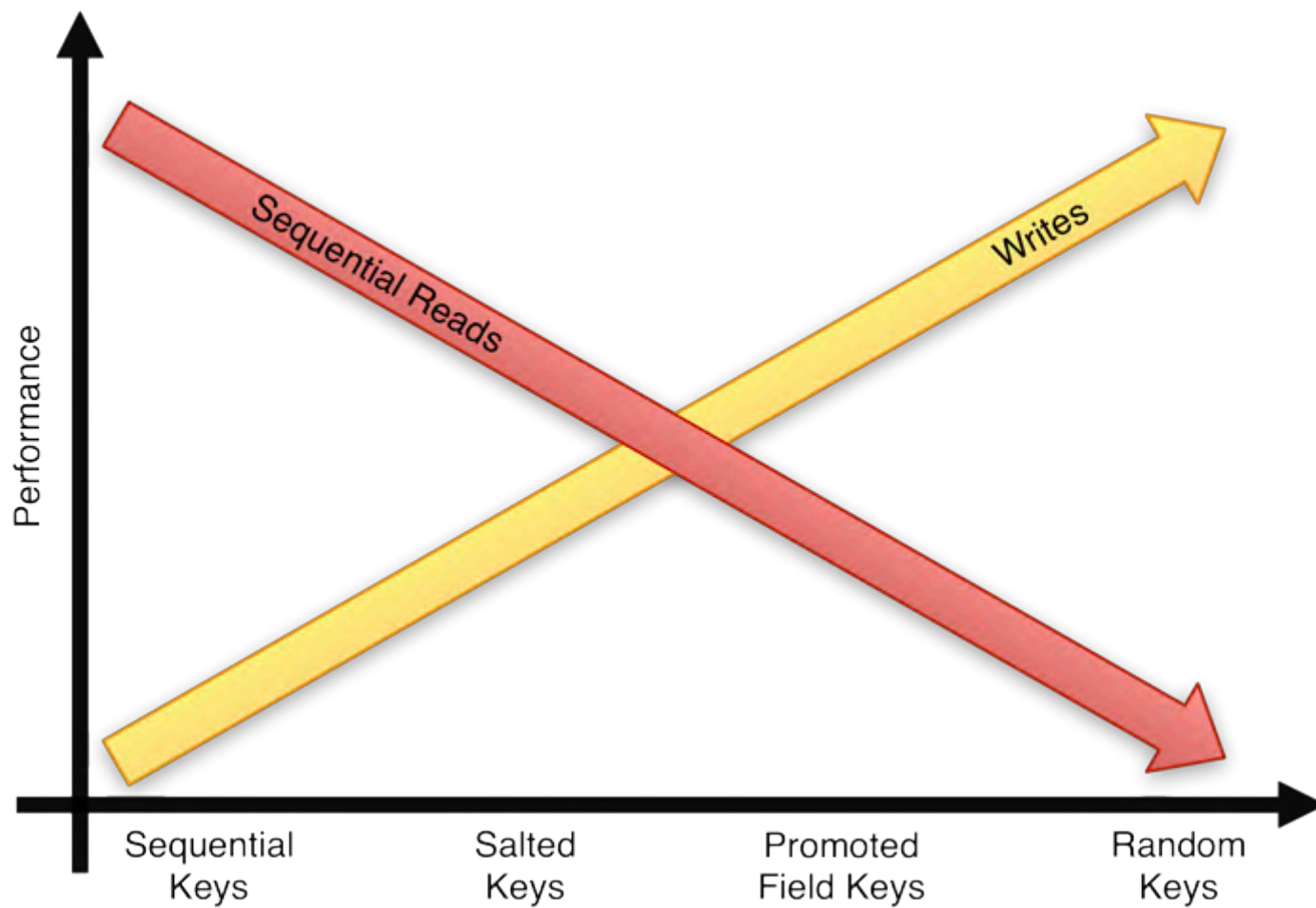


# 性能优化





# 读写性能取舍\*



# 性能优化目标

- 读 vs. 写？
  - 读需要合并HFile，因此文件越少越好
  - 写需要减少Compaction操作，因此文件越多越好
  - 优化读或者写之一，而不是全部
- 顺序 vs. 随机？
- 参考值——每个RegionServer吞吐率>20MB/s
  - 读吞吐率>3000ops/s, 写吞吐率>10000ops/s
- 尽量在HBase表结构设计时就考虑解决性能问题，而不是通过设置参数来调整HBase性能！



# HBase的性能优化

- 预分配region
- 启用压缩以减少HDFS数据量，可提高读性能
- Region Server进程配置大内存(>16G)，但不要太大(<100G)
- 每个Region Server拥有的region数量<200
- 优化表结构设计，防止少数几个region成为瓶颈
  - 一个简单的经验公式：每台region server纯写入时高负载应能达到>1万条记录/秒（每记录200字节）



# Region数目

- 通常每台服务器能服务的Region数目为20到150个，集群粗略估计可按**100个Region为上限**，具体视服务器能力（CPU内核）及访问压力（每个Region的服务量）而定
- 过多Region的症状：
  - CPU线程切换频繁
  - 内存使用过大，造成GC频繁，服务Timeout
  - 每个Region的Memstore太小，磁盘flush频繁，HFile文件过多小文件



# Major Compaction

- HBase根据时间来计划执行Major Compaction
  - `hbase.hregion.majorcompaction` = 604800000 (缺省值为一周)
  - `hbase.hregion.majorcompaction.jitter` = 0.5 (缺省值为半周)
- 执行过程非常长，且非常耗资源
  - 无法控制只在合适的时间执行
- 建议在生产环境禁用计划Major Compaction，通过命令行手工触发或自己进行物理数据删除



# Compaction

- 检测：通过HBase管理页面查看CompactionQueue长度以及Region中StoreFile的个数，如果CompactionQueue队列长度过长（如>10）或增长过快，则需要考虑调整Compaction参数
  - 注意查看Region以及StoreFile的大小，确认是否因为太多过小文件的原因导致文件数目多。如是，需要检查内存使用及设置
- 主要参数
  - `hbase.hstore.compactionThreshold`，建议值10
  - `hbase.hstore.blockingStoreFiles`，建议值30
  - 更大的`hbase.hregion.memstore.flush.size`能减少Compaction的次数
    - 现在缺省128MB，一般不用修改



# HBase的GC特点

- 由单个RPC带来的操作类垃圾对象是短期的
- Memstore是相对长期驻留的，按2MB为单位分配
- Blockcache是长期驻留的，按64KB为单位分配
- 如何有效的回收RPC操作带来的临时对象是HBase的GC重点
- 不建议HBase的堆大小操作操过64GB，否则GC压力大、执行时间太长

[https://blogs.apache.org/hbase/entry/tuning\\_g1gc\\_for\\_your\\_hbase](https://blogs.apache.org/hbase/entry/tuning_g1gc_for_your_hbase)

<http://blog.cloudera.com/blog/2014/12/tuning-java-garbage-collection-for-hbase/>



# G1GC

**-XX:+UseG1GC**

**-XX:+UnlockExperimentalVMOptions**

**-Xmx32g -Xms32g**

32 GB heap, initial and max should be the same

**-XX:ParallelGCThreads = 8+(logical  
processors-8)(5/8)**

**-XX:G1NewSizePercent= 3-9**

Minimum size for Eden each epoch, differs by cluster

This value is intentionally very low and doesn't actually represent a pause time upper bound. We recommend keeping it low to pin Eden to the low end of its range (see Tuning #2).

**-XX:MaxGCPauseMillis=50**

Optimistic target, most clusters take 100+ ms

**-XX:-OmitStackTraceInFastThrow**

Better stack traces in some circumstances, traded for a bit more CPU usage

**-XX:+ParallelRefProcEnabled**

Helps keep a lid on [reference processing time](#) issues we were seeing

**-XX:+PerfDisableSharedMem**

Alleged to protect against [a bizarre Linux issue](#)

**-XX:-ResizePLAB**

Alleged to save some CPU cycles in between GC epochs





# RegionServer硬件建议

- 服务器硬盘空间不大于**6TB**\*RegionServer
- 足够的内存堆大小（约等于**硬盘空间/200**）
- HBase对于CPU要求高，越多core越好
- 磁盘与网络的速度匹配
  - 比如如果是24块硬盘，吞吐率约2.4GB/s，则网络需要至少万兆网络。而千兆网一般配4到6块硬盘。
- 更多的硬盘数量能增加并发，提高HBase的读性能



# 写性能

- HBase理论平均写延时<10ms，时间复杂度 $O(1)$
- 没有可用的handler响应
  - 考虑增加handler数目或硬件资源
- 更常见的情况是95%—99%的写入都很快，但有些写入非常慢，甚至慢上万倍，一般问题在服务器端：
  - 写入Memstore慢
    - HLog写入超时——考虑HDFS及硬盘异常
    - GC——考虑优化内存使用（GC参数及算法调优有限）
  - Flush慢
    - HFile写入超时——考虑HDFS及硬盘异常
    - Compaction被触发且运行时间长——优化高峰期Compaction策略



# 读性能优化

- 使用Redis、Memcache等缓存
- 使用Read Replica
- 使用Bloom Filter
- Filter等过滤结果数据
- Block cache大小
  - 查看cache命中率
- StoreFile过多，影响查找效率，手工Compact
  - 相比单个文件，10个StoreFile文件性能下降约50%—100%
- 本地化读写太差，网络IO消耗严重



# HBase客户端性能优化

- 使用**批量**数据处理接口
- 保持**2MB**的Chunk Size
- 使用内存pool缓存HTable及其他可重用对象
- 使用多线程并发技术
  - Parallel Scanner
- 使用异步调用接口
  - AsyncClient
- 使用数据预取以及预缓存



# Thanks!



@Cloudera中国



@陈飏