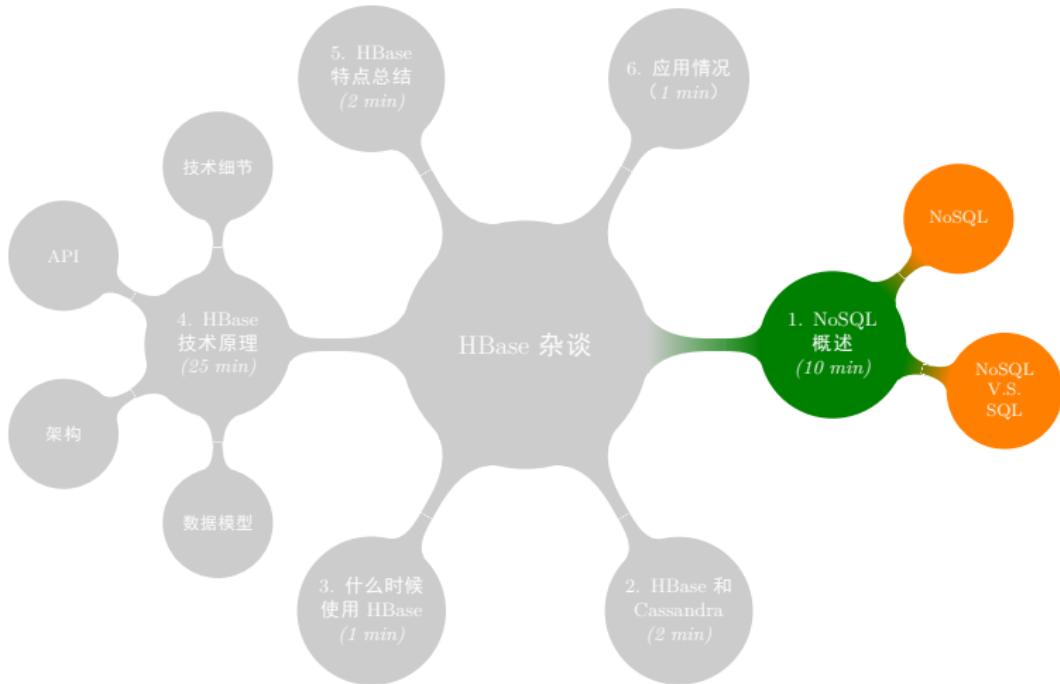


# 心智图



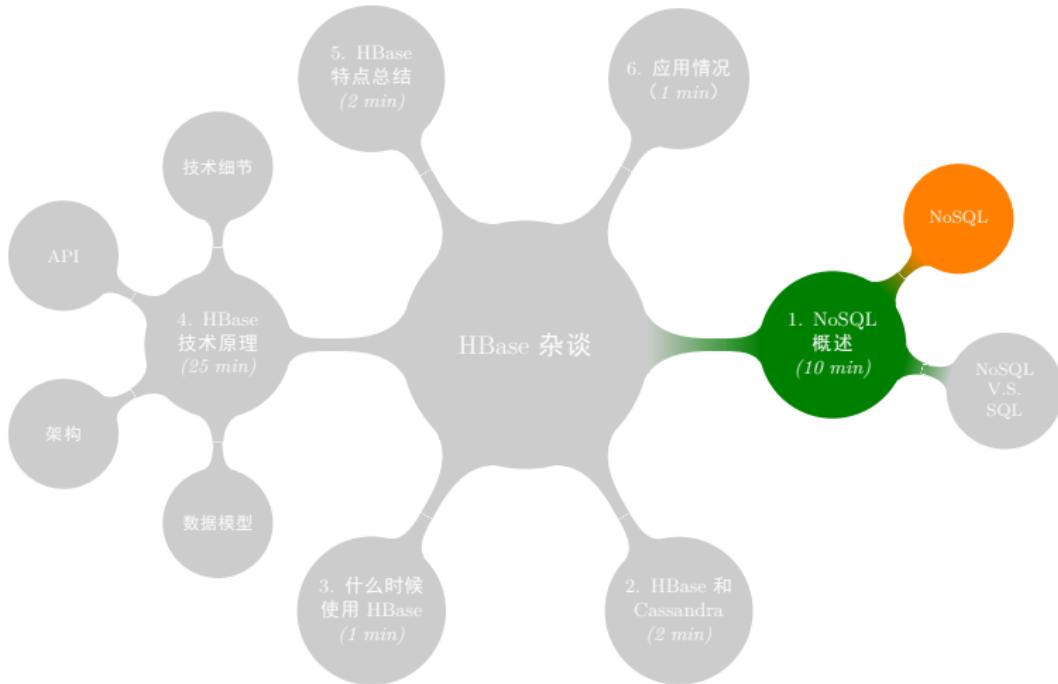
# NoSQL 概述





Licensed under a Creative Commons  
Attribution-ShareAlike 3.0 Unported License.

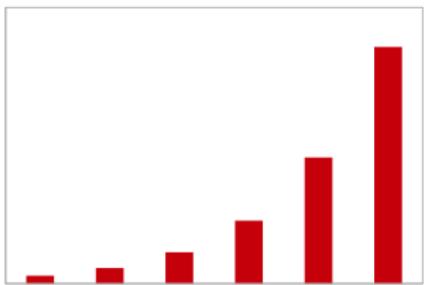
# NoSQL



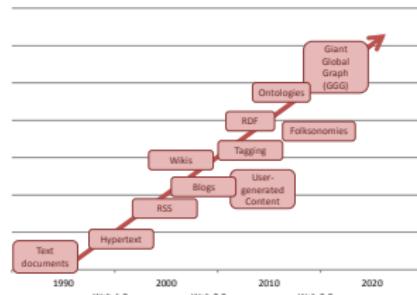


NoSQL: Why

# 数据存储的趋势



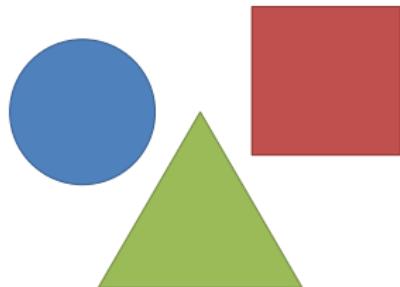
数据增长



信息连接



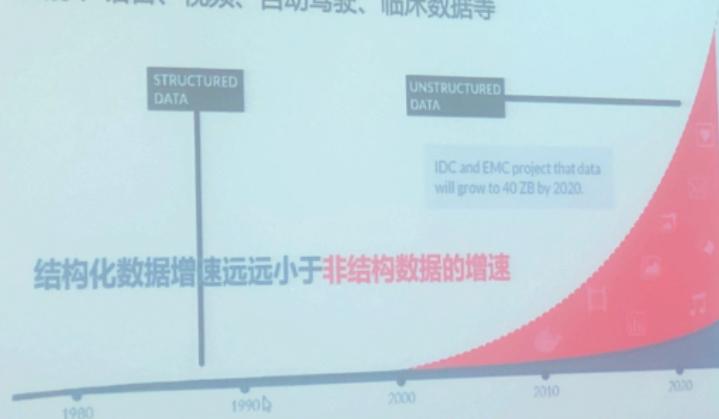
并发性



结构多样性

## 当前智能计算发展趋势和特征：

- 提高性能→降低能耗
  - 大量应用场景对于能耗约束严格
- 计算密集型→数据密集型
  - 片上存储容量、访存带宽等更加重要
- 结构化数据→半结构化或非结构化数据
  - 图像、语音、视频、自动驾驶、临床数据等



NoSQL: Why

# 关系型数据库的不足

## 关系型数据库：管理结构化数据

- 复杂导致不确定性
- 有些问题不适合采用关联型的数据模型
- 当数据量增长到一台机器已经不能容纳，我们需要将不同的数据表分布到不同的机器

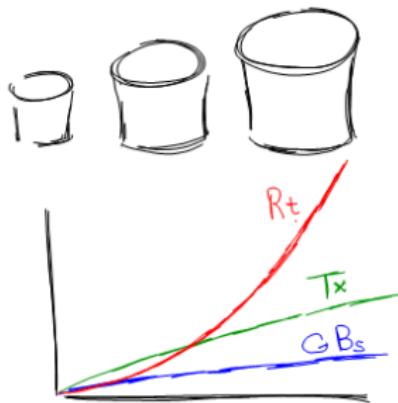


Figure 1: 数据量增长对性能的消极影响

NoSQL: Why

# Help!





# Martin Fowler 的定义

## NosqlDefinition

“Some characteristics are common amongst these databases, but none are definitional.

- Not using the relational model (nor the SQL language)
- Open source
- Designed to run on large clusters
- Based on the needs of 21st century web properties
- No schema, allowing fields to be added to any record without controls ”



Figure 2: Martin Fowler

# NoSQL 的特点

- 简单灵活

- 复杂的操作交给应用层
- 使用 NoSQL 数据模型，通常需要你对存储的内部结构和实现算法有一定的了解
- 使用 NoSQL 数据库，通常要自己处理数据结构解析和数据的冗余复制问题
- 需要预先考虑数据的分级处理

- 牺牲稳定，换取性能

可能**不具备**关系数据库的 ACID 特性的一种或几种：

- Atomic : 原子性
- Consistency : 一致性
- Isolation : 隔离性
- Durability : 持久性

# 数据模型进化论

SQL→NoSQL, 进步 or 倒退?

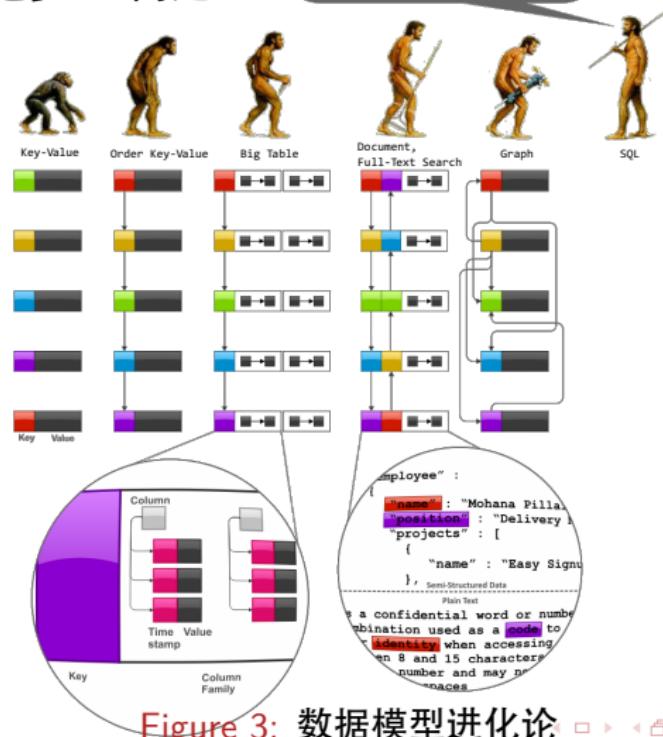


Figure 3: 数据模型进化论

# NoSQL 族谱

目前简单将 NoSQL 几个分类的代表产品列举如下：

- Key-Value 存储: Amazon Dynamo, Voldemort, BDB, Kyoto Cabinet
- Key 结构化数据存储 : Redis
- 类 BigTable 存储: Google BigTable, Apache HBase, Apache Cassandra
- 文档数据库: MongoDB, CouchDB
- 全文索引: Apache Lucene, Apache Solr
- 图数据库: neo4j, HyperGraphDB, FlockDB

# NoSQL 族谱

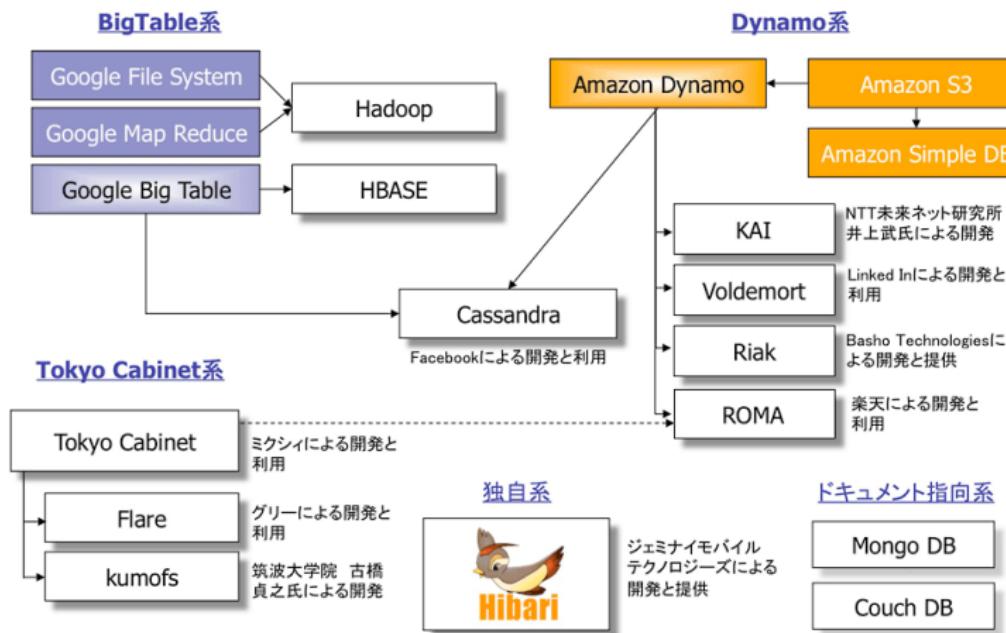
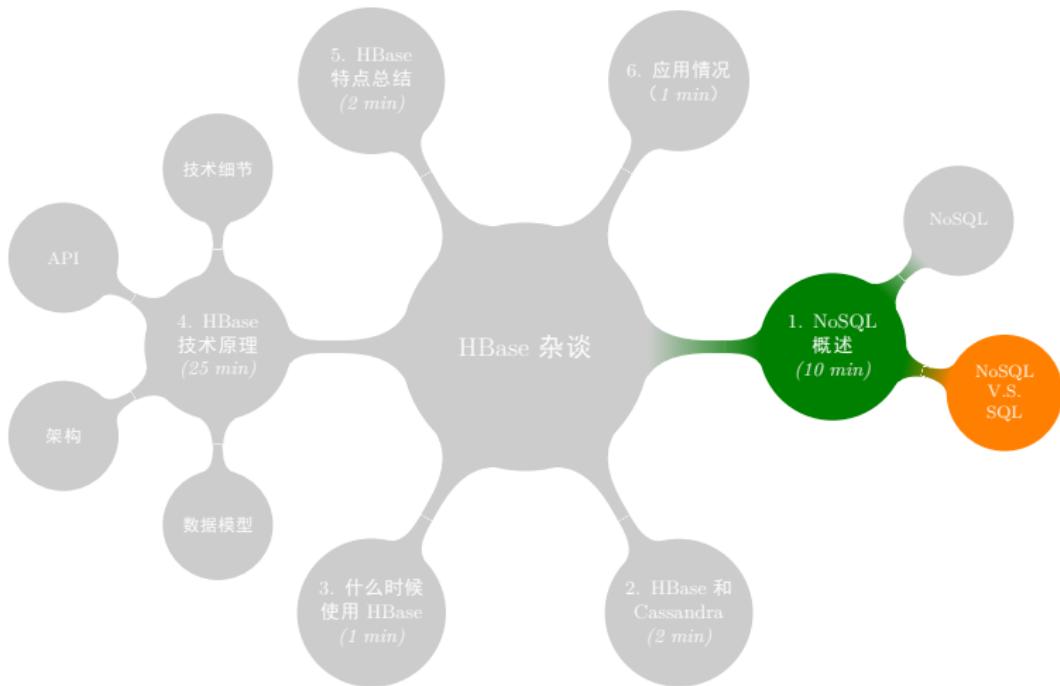


Figure 4: NoSQL 族譜

# NoSQL V.S. SQL



# 当 SQL 遇见 NoSQL

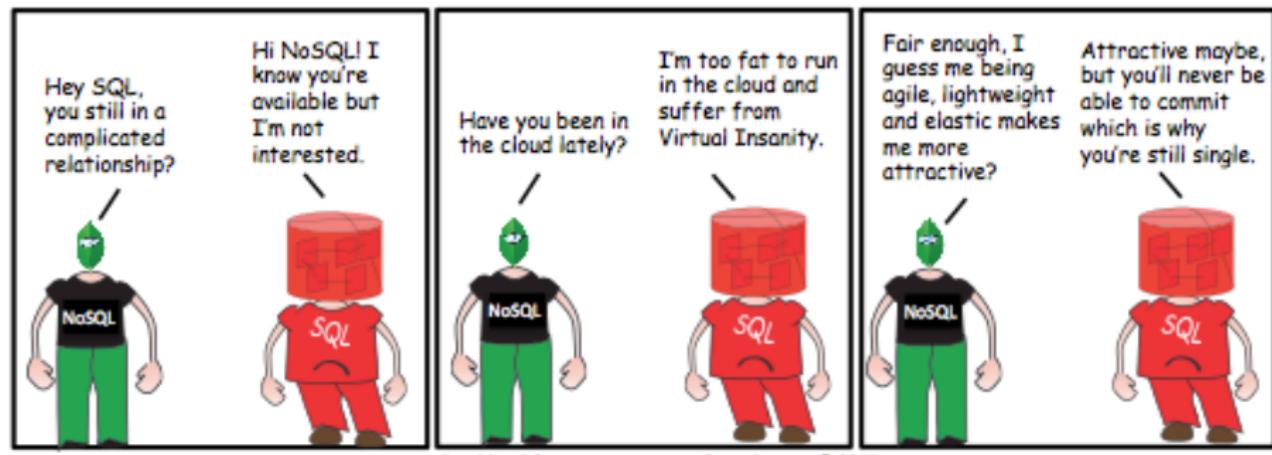


Figure 5: NoSQL Meets SQL

# SQL 还是 NoSQL ?

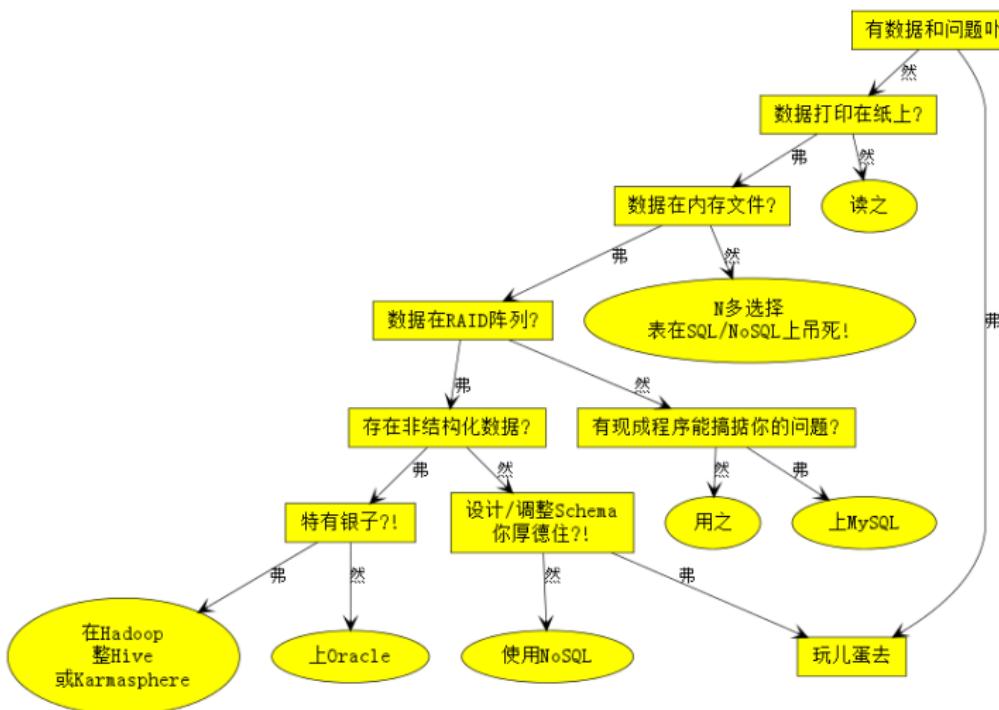
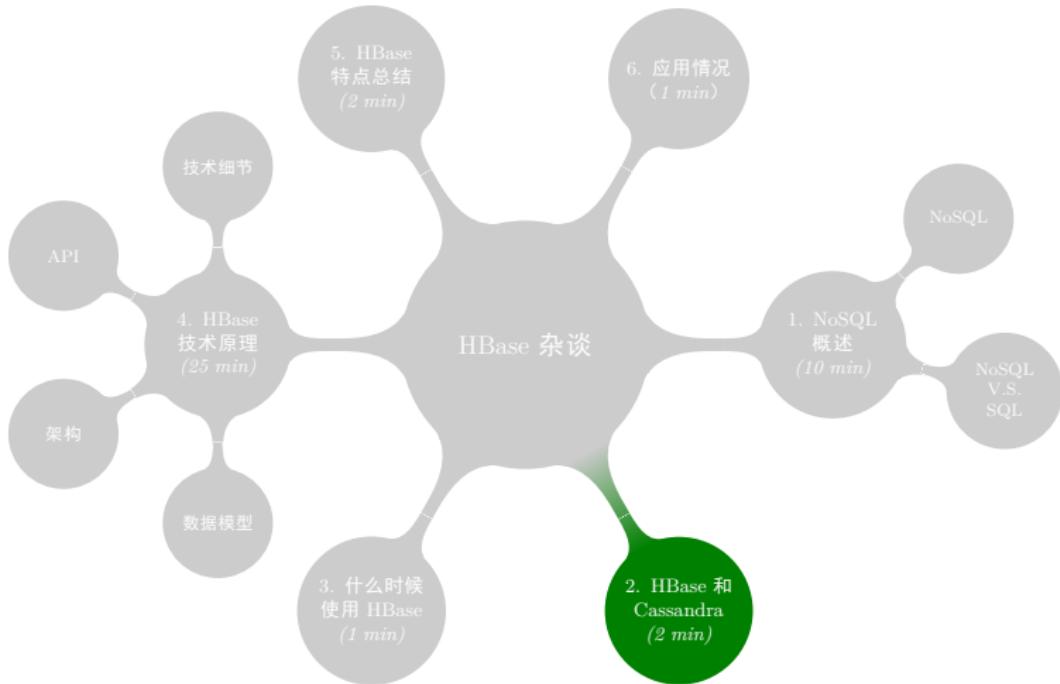


Figure 6: SQL 还是 NoSQL?

# HBase 和 Cassandra



# HBase 和 Cassandra



Hadoop Database



*Cassandra*

# HBase: What

- 一个高可靠性、高性能、面向列、可伸缩的分布式存储系统
- Google BigTable 的开源山寨版本

Bigtable: A distributed storage system for structured data

- 利用 HBase 技术可在廉价 PC Server 上搭建起大规模结构化存储集群
- 实时随机读写补充 HDFS 的不足
- 主要贡献者: Yahoo!, Facebook, Cloudera

# Hadoop 生态系统

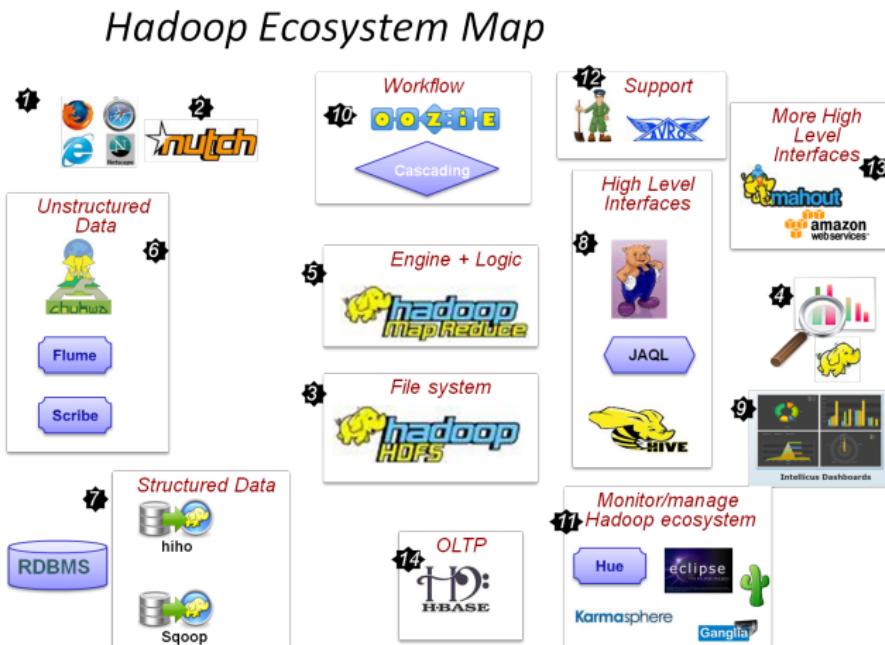


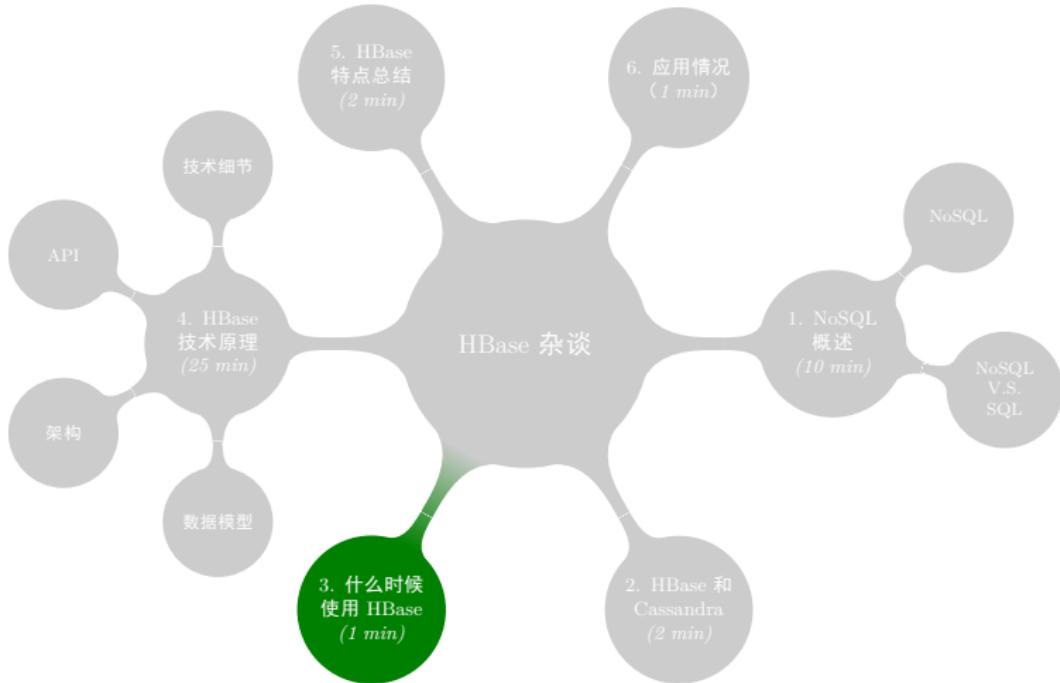
Figure 7: Hadoop 生态图

# Cassandra: What

Cassandra 是一套开源分布式 NoSQL 数据库系统。它最初由 Facebook 开发，用于储存收件箱等简单格式数据，是 Facebook 数据库系统的开源分支。集 Google BigTable 的数据模型与 Amazon Dynamo 的完全分布式的架构于一身。Facebook 于 2008 将 Cassandra 开源，此后，由于 Cassandra 良好的可扩放性，被 Digg、Twitter 等知名 Web 2.0 网站所采纳，成为了一种流行的分布式结构化数据存储方案。

实际上，Cassandra 的最初开发工作就是由两位从 Amazon 跳槽到 Facebook 的 Dynamo 工程师 Avinash Lakshman 和 Prashant Malik 完成的。

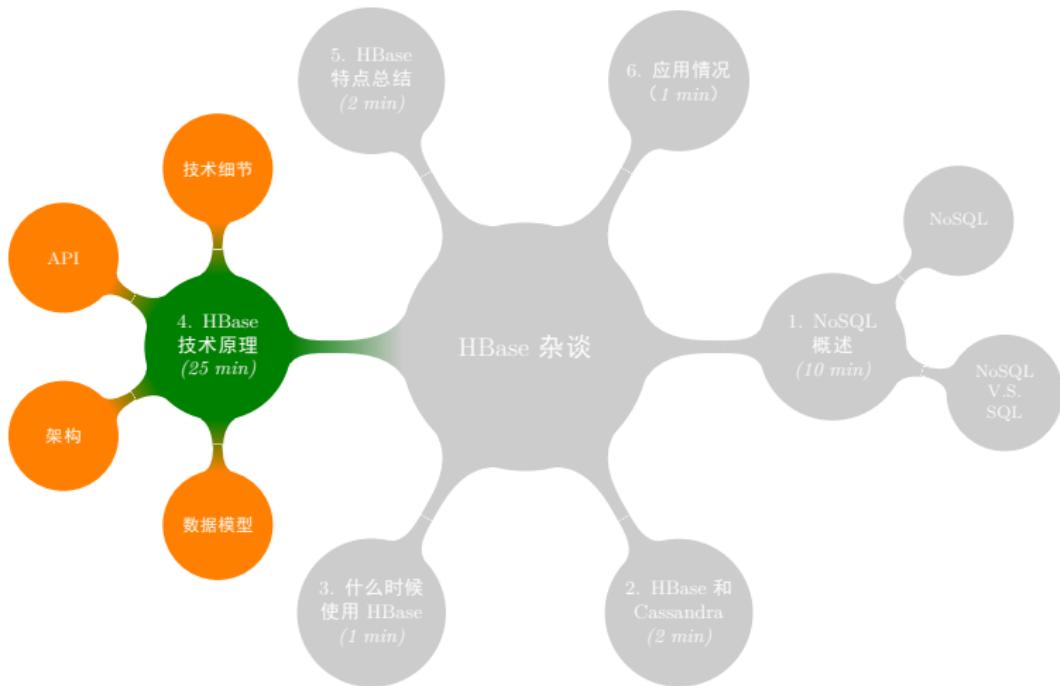
# 什么时候使用 HBase



# HBase: When

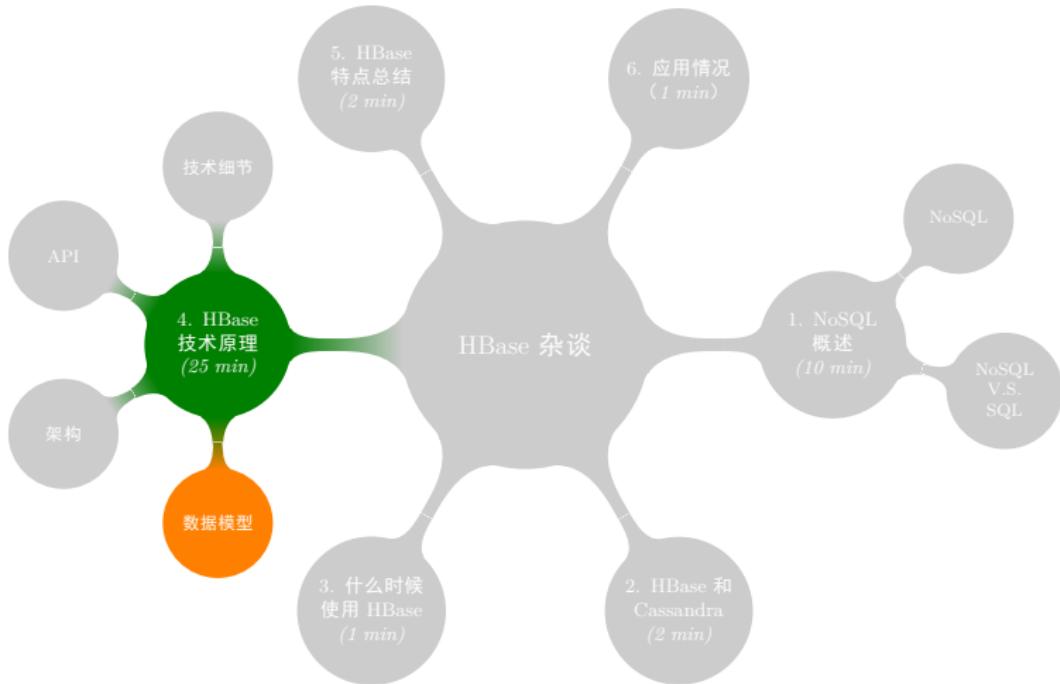
- 海量数据 (上百个 TB 或更多)
- 需要很高的吞吐量
- 需要在海量数据中实现高效的随机读取
- 需要很好的伸缩能力
- 能够同时处理结构化和非结构化的数据
- 不需要完全拥有 RDMS 所具备的 ACID 特性

# 技术原理



## 数据模型

## 数据模型





# 逻辑视图

Table 1: 表 webtable

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>...”	

# 构造函数

./src/main/java/org/apache/hadoop/hbase/keyvalue.java :

```
public KeyValue(final byte[] row, final byte[] family, final byte[]
    qualifier, final long timestamp, final byte[] value)
{
    this(row, family, qualifier, timestamp, Type.Put, value);
}

public KeyValue(final byte[] row, final int roffset, final int rlength,
    final byte[] family, final int foffset, final int flength,
    final byte[] qualifier, final int qoffset, final int qlength,
    final long timestamp, final Type type,
    final byte[] value, final int voffset, final int vlength) {
    this.bytes = createByteArray(row, roffset, rlength,
        family, foffset, flength, qualifier, qoffset, qlength,
        timestamp, type, value, voffset, vlength);
    this.length = bytes.length;
    this.offset = 0;
}
```

物理存储

## 物理视图

尽管在概念视图里，表可以被看成是一个稀疏的行的集合。但在物理上，它是区分 column family 存储的。新的 columns 可以不经过声明直接加入一个 column family。

另外值得注意的是概念视图中的空白 cell 在物理上是不存储的。

Table 2: ColumnFamily anchor

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"

Table 3: ColumnFamily contents

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>..."

# 分片：Region

- 按 rowkey 将 Table 动态分割为一个个 region, 每个 region 包含一个连续的行范围 [startkey,endkey)

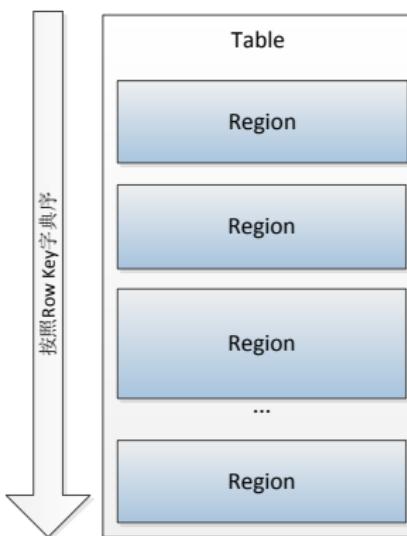


Figure 9: 表格分割成 region

# Region

- region 按大小分割的，每个表一开始只有一个 region，随着数据不断插入表，region 不断增大，当增大到一个阀值的时候，region 就会等分为个新的 region。当 table 中的行不断增多，就会有越来越多的 region，并在存储集群内分发以达到负载均衡。

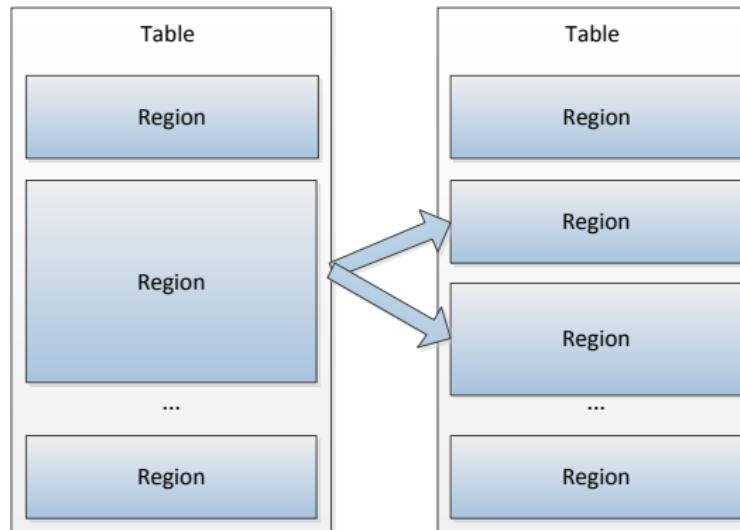


Figure 10: Region 的分割

# RegionServer

HRegion 是 Hbase 中分布式存储和负载均衡的最小单元。最小单元就表示不同的 HRegion 可以分布在不同的 HRegion server 上，但一个 HRegion 不会拆分到多个 server 上。

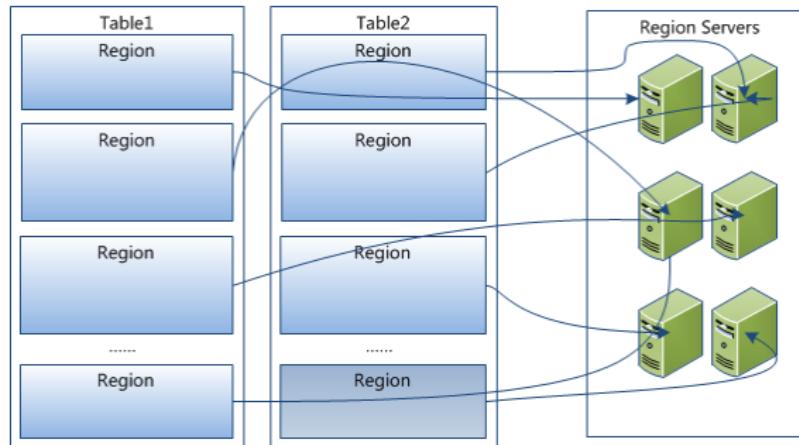


Figure 11: RegionServer

# Store

HRegion 虽然是分布式存储的最小单元，但并不是存储的最小单元。事实上，HRegion 由一个或者多个 Store 组成，每个 Store 保存一个 columns family。每个 Store 又由一个 memStore 和 0 至多个 StoreFile 组成。

StoreFile 以 HFile 格式保存在 HDFS 上。

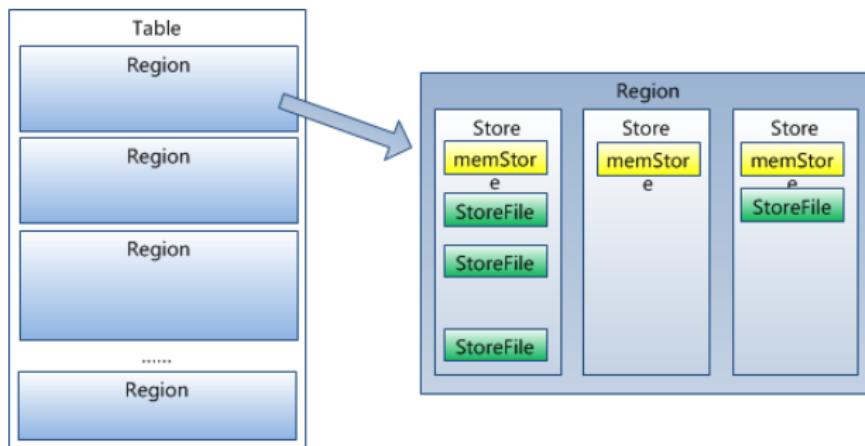


Figure 12: Store

# HFile 的格式

HFile 分为 6 个部分：

- ① Data Block 段：保存表中的数据，这部分可以被压缩；
- ② Meta Block 段 (可选)：保存用户自定义的元信息，可以被压缩；
- ③ File Info 段：Hfile 的元信息，不被压缩，用户也可以在这一部分添加自己的元信息；
- ④ Data Block Index 段：Data Block 的索引。每条索引的 key 是被索引的 block 的第一条记录的 key；
- ⑤ Meta Block Index 段 (可选的)：Meta Block 的索引；
- ⑥ Trailer：这一段是定长的。保存了每一段的偏移量，读取一个 HFile 时，会首先读取 Trailer，Trailer 保存了每个段的起始位置 (段的 Magic Number 用来做安全 check)，然后，DataBlock Index 会被读取到内存中，这样，当检索某个 key 时，不需要扫描整个 HFile，而只需从内存中找到 key 所在的 block，通过一次磁盘 io 将整个 block 读取到内存中，再找到需要的 key。DataBlock Index 采用 LRU 机制淘汰。

物理存储

# HFile 的格式

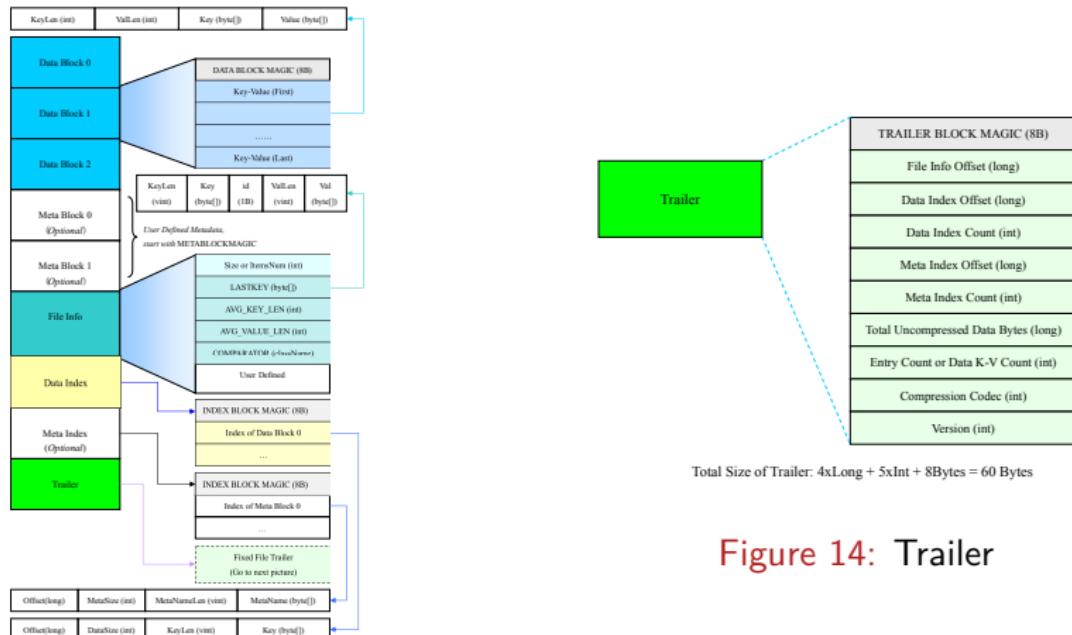
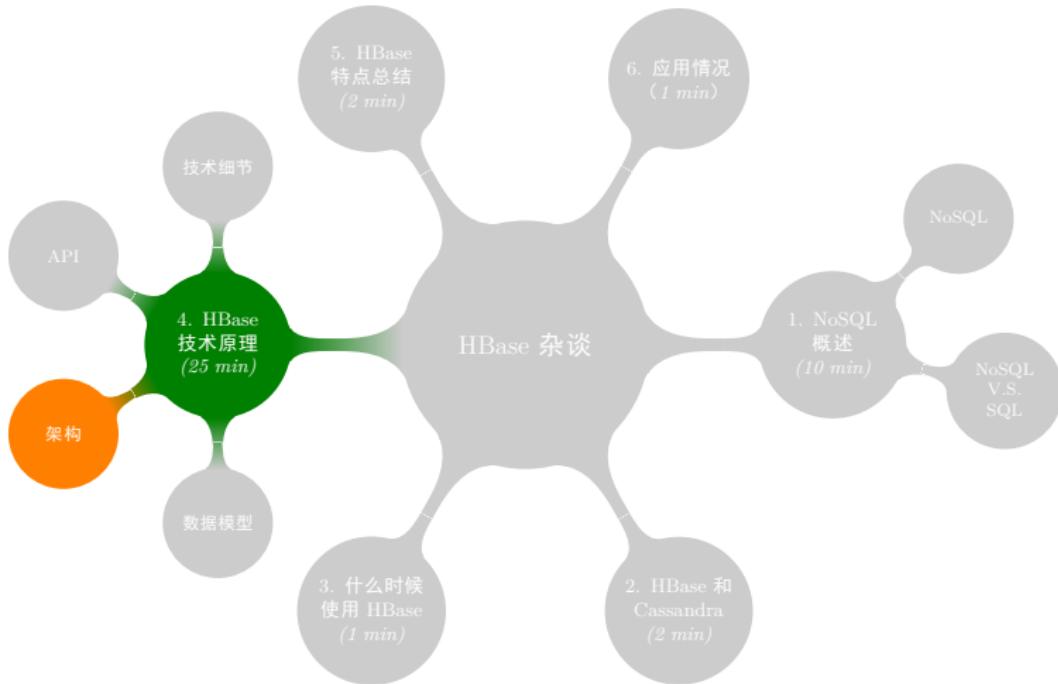


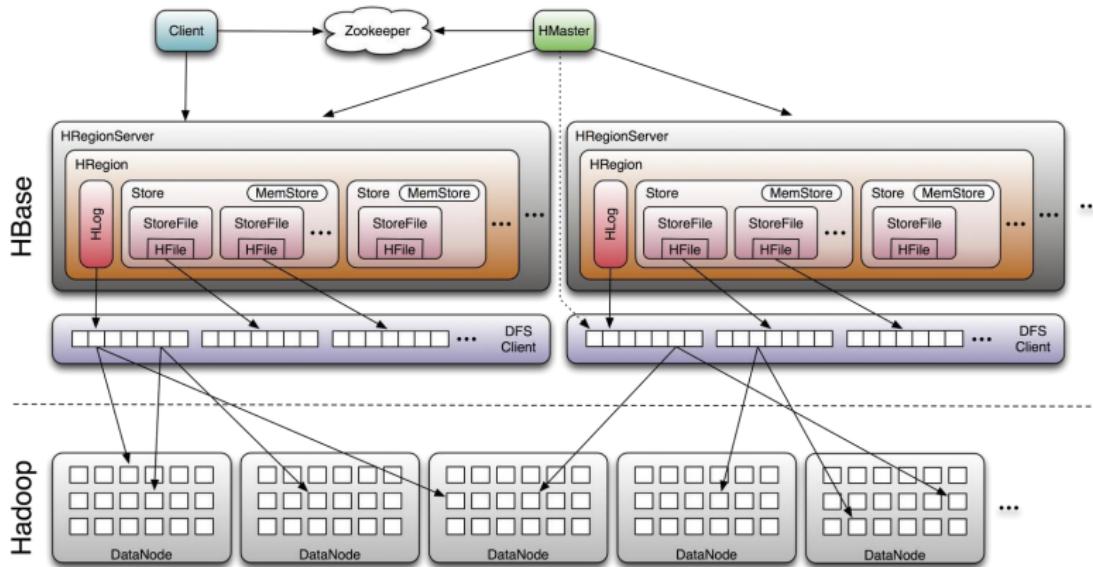
Figure 13: HFile 的格式

Figure 14: Trailer

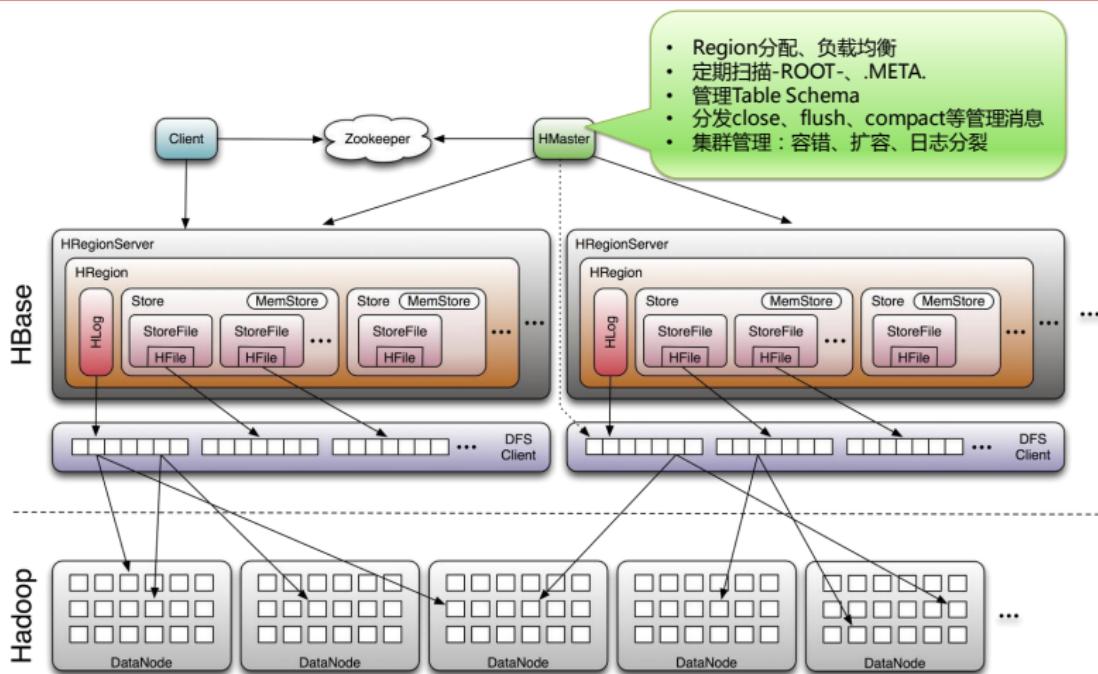
# 架构



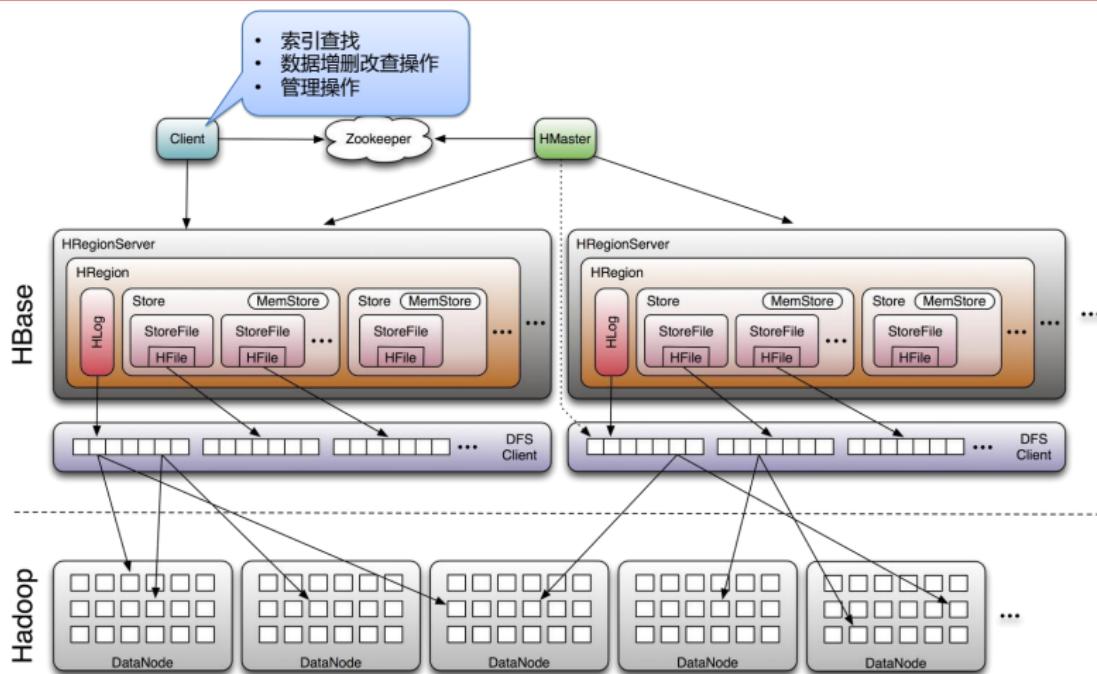
# HBase 架构



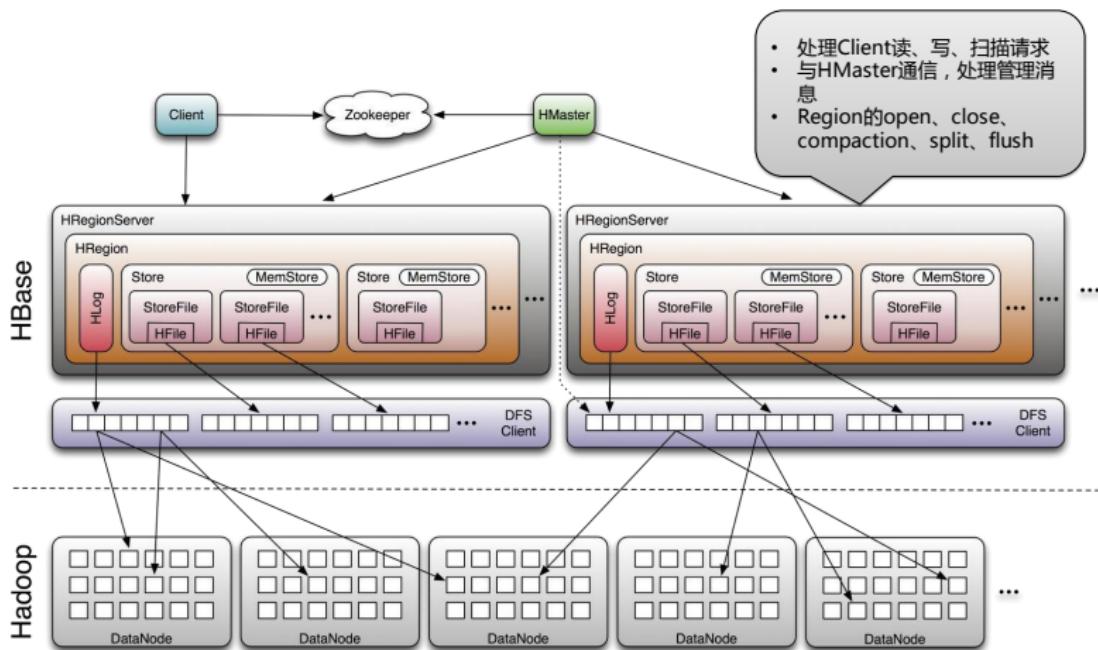
# HBase 架构



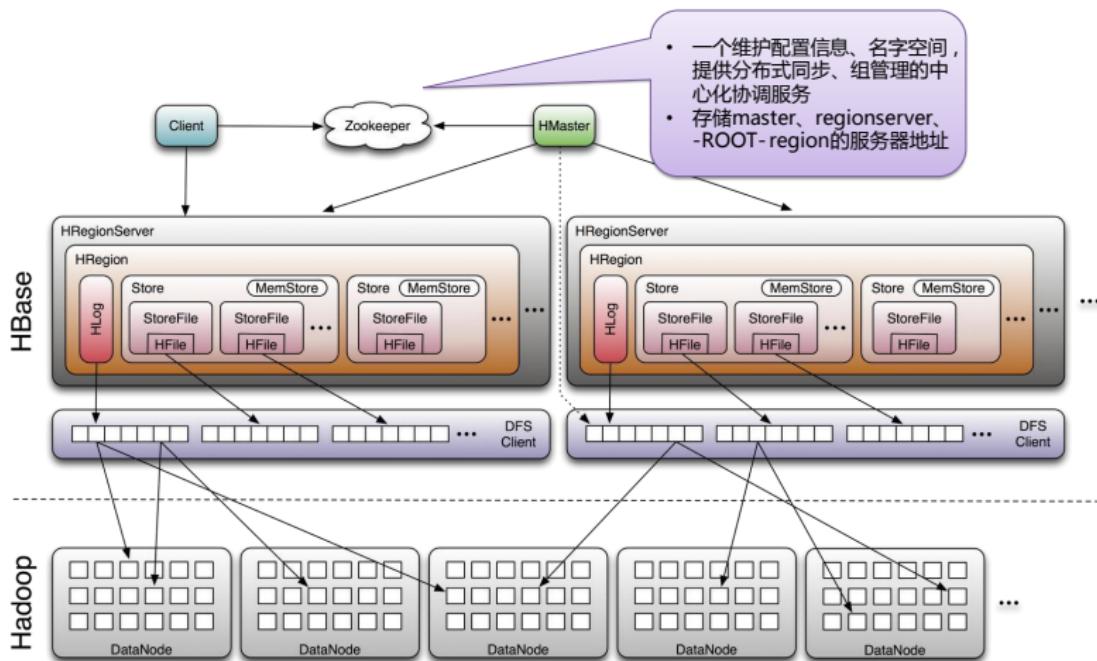
# HBase 架构



# HBase 架构

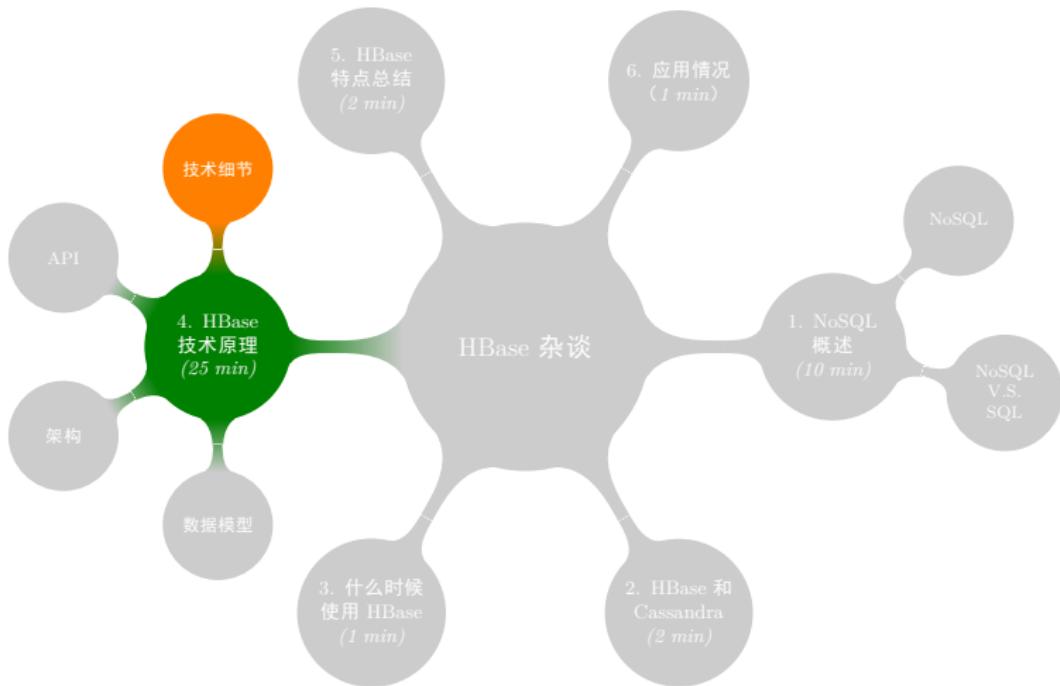


# HBase 架构



## 技术细节

## 技术细节



# 索引方式：三层 B+ 树

- ZooKeeper 的“-ROOT-” znode
- -ROOT- region
- .META. regions

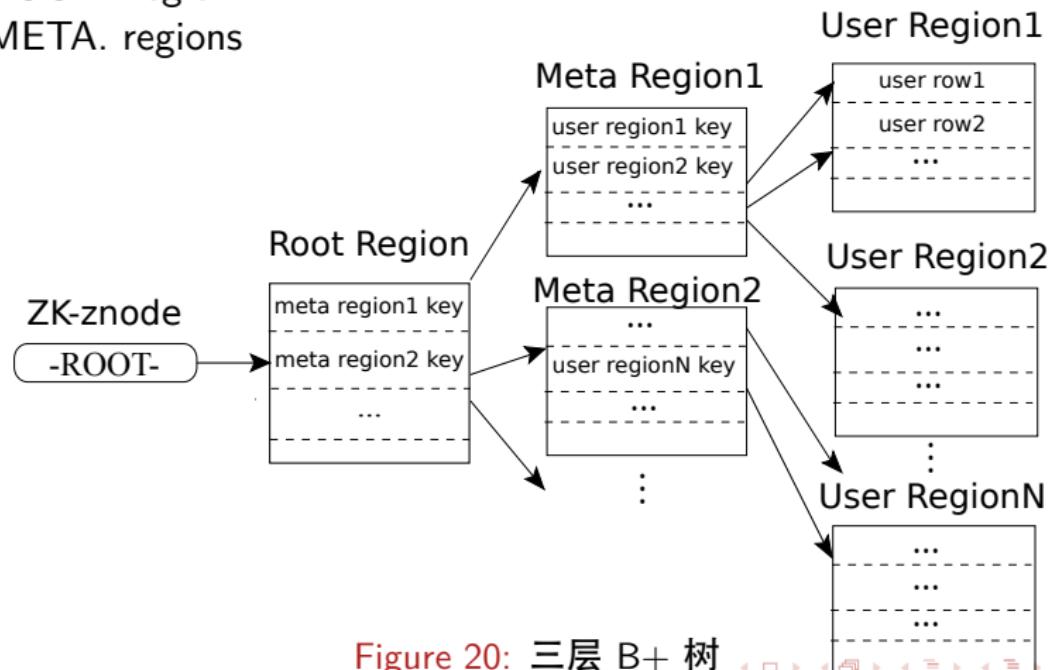


Figure 20: 三层 B+ 树

# 索引方式：三层 B+ 树

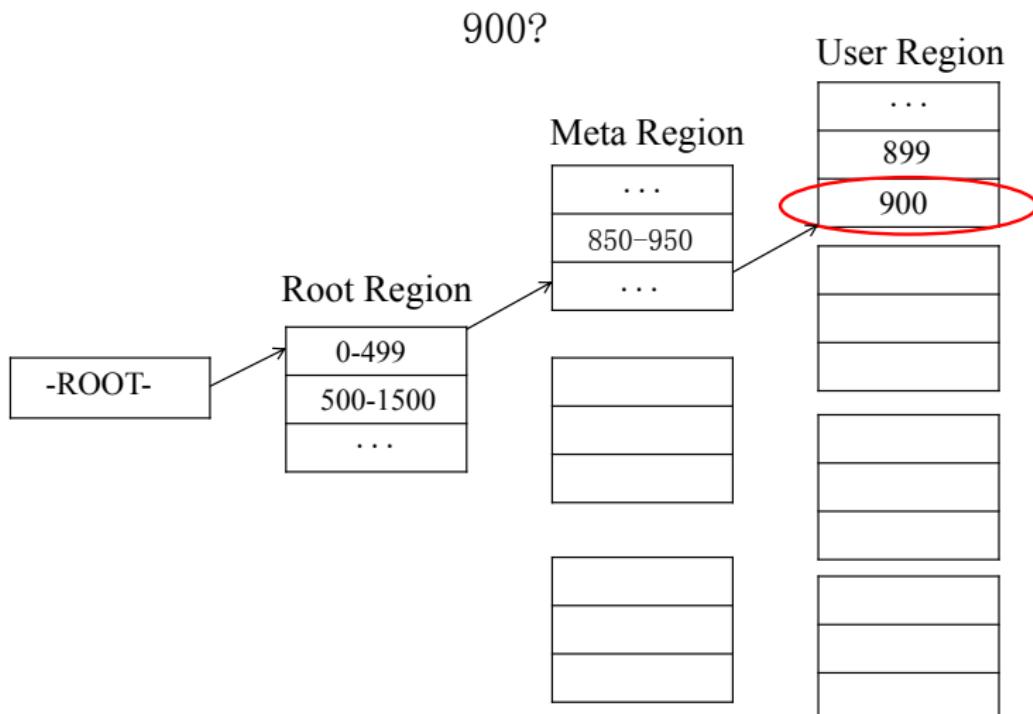


Figure 22: 在三层 B+ 树中找 Region 号为 900 的数据

技术细节

# 问题

算一下

在 -ROOT- 和.META. 的 region 中记录大小约 1KB, 按 region 默认大小 256MB 计算, 能够映射多少用户数据?

ROOT region 能映射  $2.6 \times 10^5$  个.META. region,  
依次能映射总的  $6.9 \times 10^{10}$  个 user region,  
意味着大约是  $1.8 \times 10^{19} (2^{64})$  字节用户数据 (即  $2^{14}$  PB)

技术细节

# 问题

算一下

在 -ROOT- 和.META. 的 region 中记录大小约 1KB, 按 region 默认大小 256MB 计算, 能够映射多少用户数据?

ROOT region 能映射  $2.6 \times 10^5$  个.META. region,  
依次能映射总的  $6.9 \times 10^{10}$  个 user region,  
意味着大约是  $1.8 \times 10^{19} (2^{64})$  字节用户数据 (即  $2^{14}$  PB)

技术细节

# Client Cache

client 会将查询过的位置信息保存缓存起来，缓存不会主动失效，因此如果 client 上的缓存全部失效，则需要进行 6 次网络来回，才能定位到正确的 region(其中三次用来发现缓存失效，另外三次用来获取位置信息)。

# 读写过程

“RAM 是硬盘，硬盘是磁带” — Jim Gray

对于随机访问，硬盘慢得不可忍受，但如果你把硬盘当成磁带来用，它吞吐连续数据的速率令人震惊；它天生适合用来给以 RAM 为主的应用做日志。

- 使用缓存来实现顺序写！
- 使用日志来实现数据恢复！

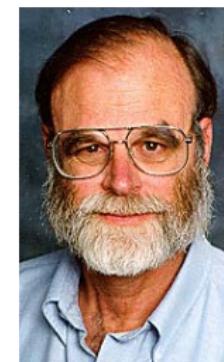


Figure 23: Jim Gray

技术细节

## MemStore+WAL

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)

当系统出现意外时，可能导致内存 (MemStore) 中的数据丢失，此时使用 Log(WAL log) 来恢复 checkpoint 之后的数据。

前面提到过 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 split，等分为两个 StoreFile。

# MemStore+WAL

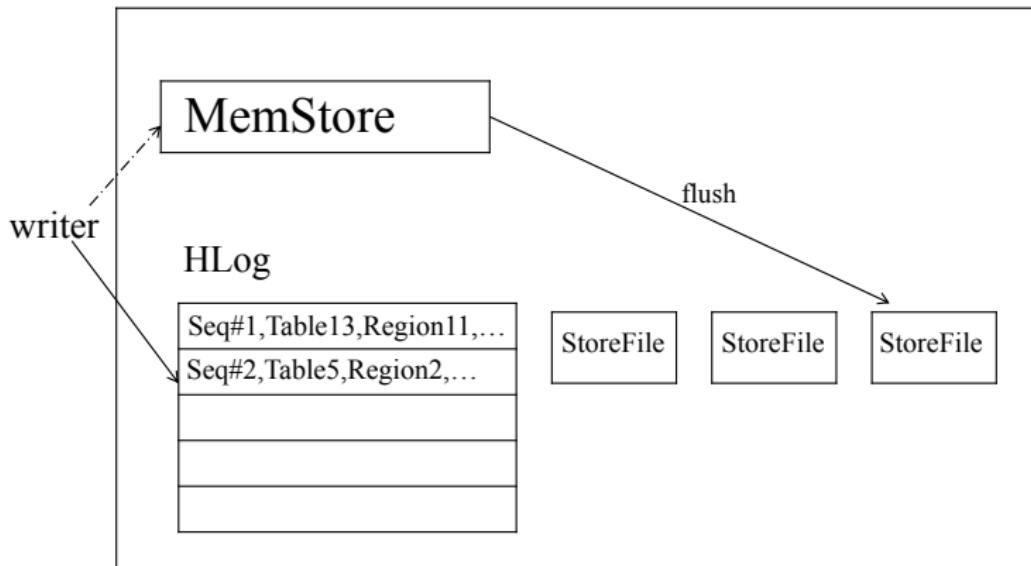


Figure 24: MemStore+WAL

技术细节

# 版本更新 |

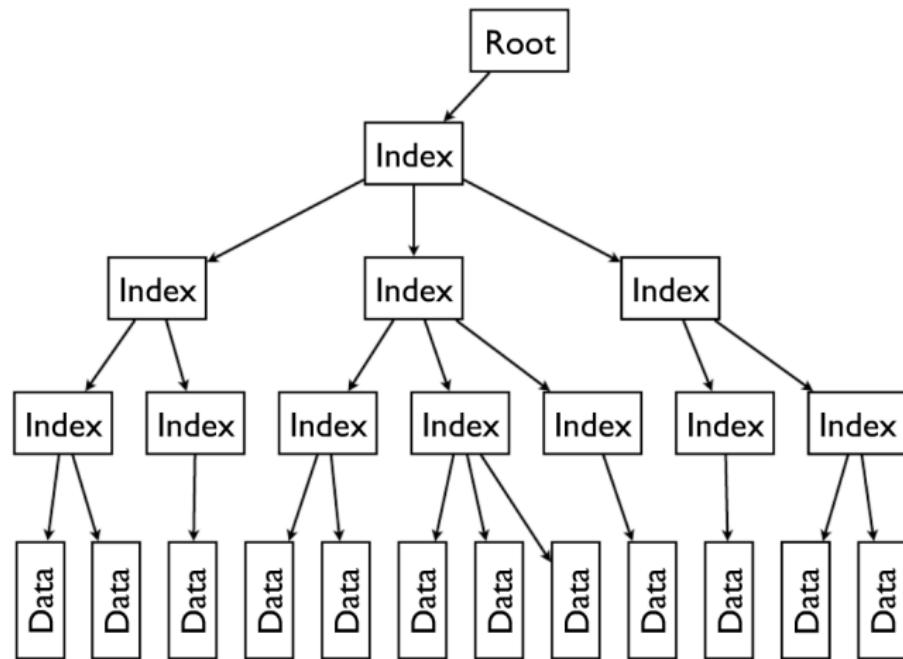


Figure 25: 版本更新

○○○○○○○○○○○○

○○○○○○○○○○○○●○○○○○○○

技术细节

# 版本更新 II

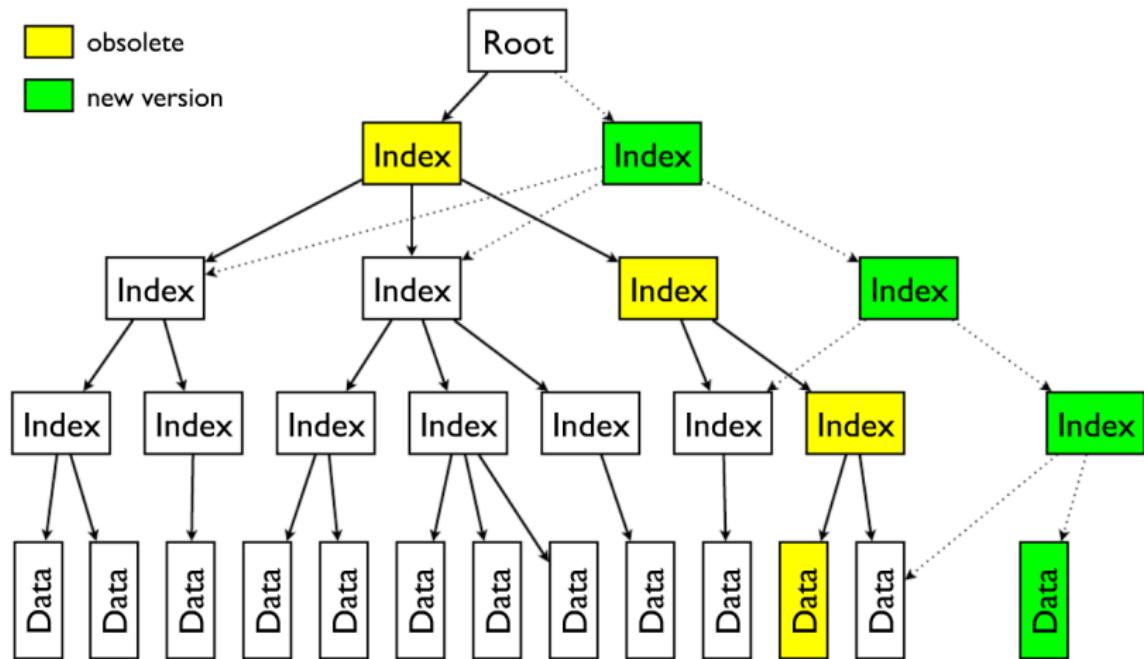


Figure 26: 版本更新

技术细节

# 版本更新 III

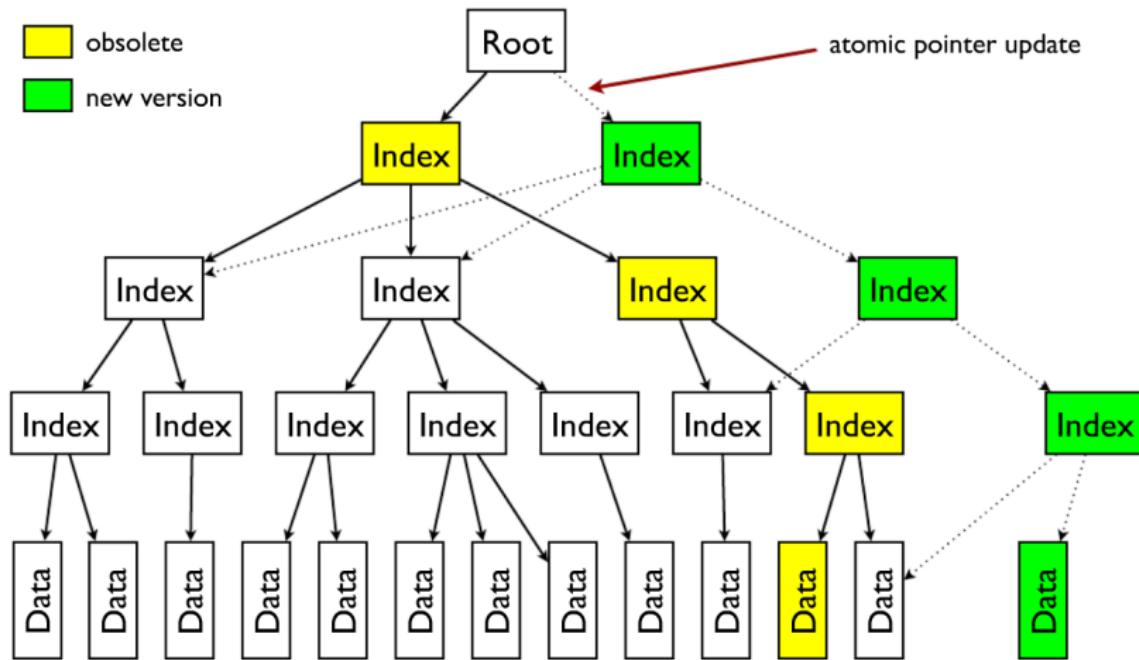


Figure 27: 版本更新

○○○○○○○○○○○○

○○○○○○○○○○○○○○●○○○○○

技术细节

# 版本更新 IV

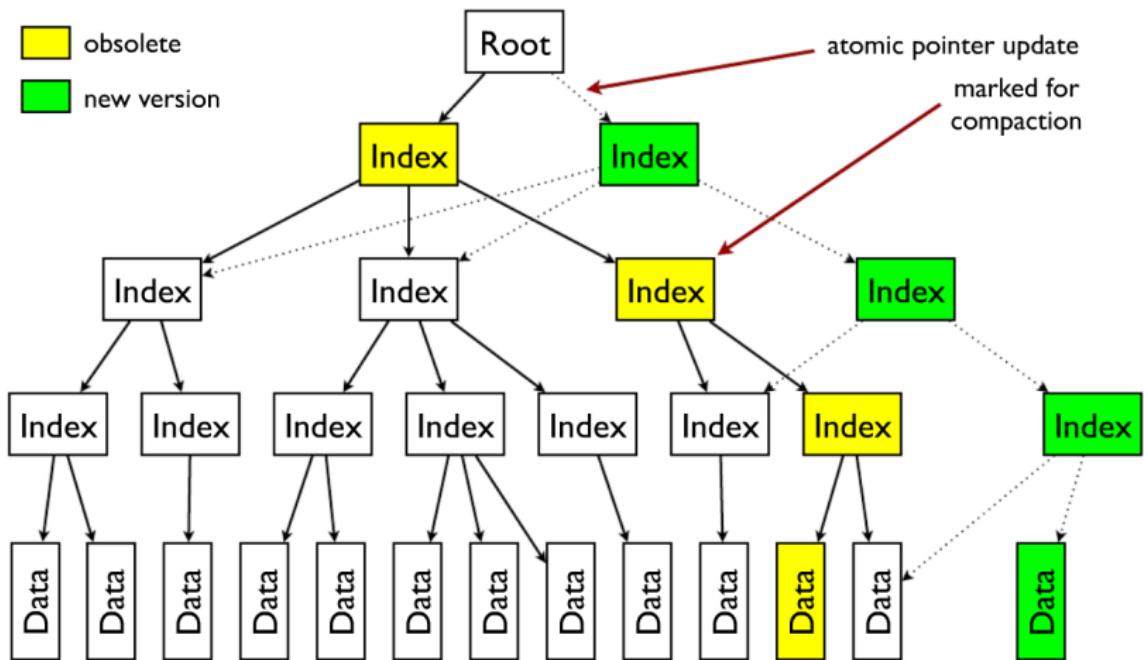


Figure 28: 版本更新

技术细节

# 版本更新 V

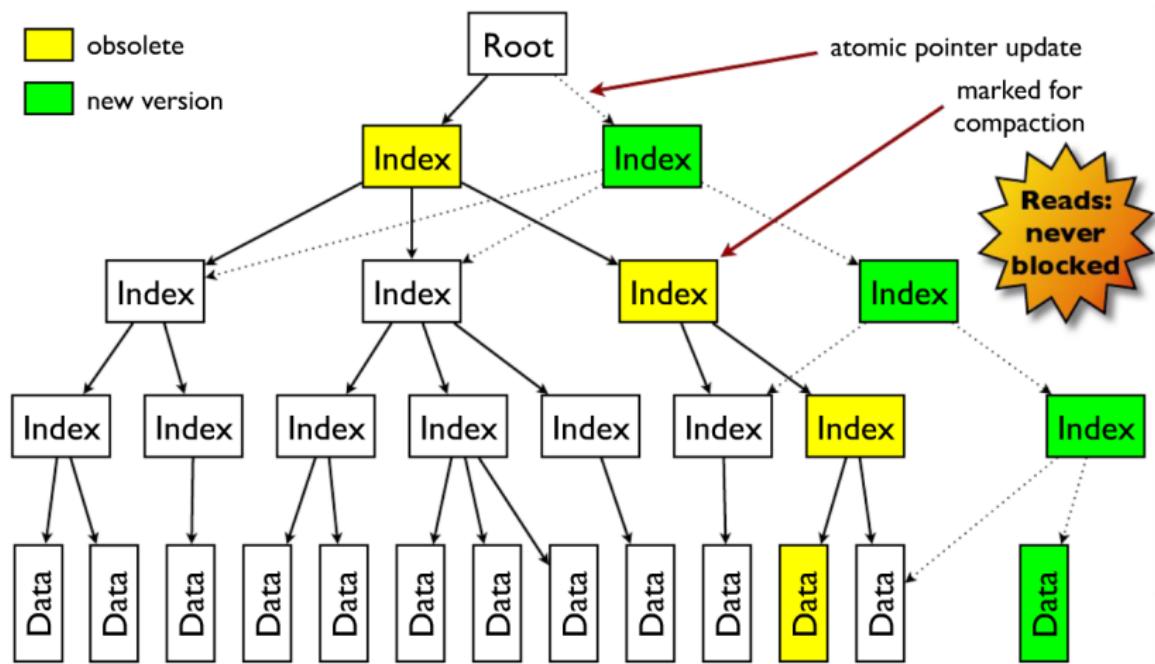


Figure 29: 版本更新

# CAP 理论

10 年前，Eric Brewer 教授指出了著名的 CAP 理论，CAP 理论告诉我们，一个分布式系统不可能满足一致性，可用性和分区容错性这三个需求，最多只能同时满足两个。

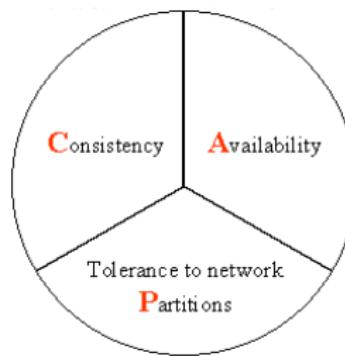


Figure 30: CAP

技术细节

# CAP 理论

不同于传统的 RDBMS 的“ACID”特性，分布式数据库通常需要具备的基本特性称为“BASE”：

- Basically Available — 基本可用
- Soft-state / 柔性事务
- Eventual Consistency — 最终一致性

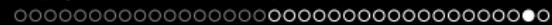


技术细节

# CAP 理论

因此系统的关注点不同，相应的采用的策略也是不一样的，只有真正理解了系统的需求，才有可能利用好 CAP 理论。

HBase 选择了 CP，即一致性和分区容错性，能够表现出强一致性以及很高的分区容错性。



技术细节

# 一致性

同一行数据的读写只在同一台 regionserver 上进行，因此 HBase 具有强一致性。

# 可靠性

- ① 存在单点故障，Region Server 宕机后，短时间内该 server 维护的 region 无法访问，等待 failover 生效。
- ② 通过 Master 维护各 Region Server 健康状况和 Region 分布。
- ③ 多个 Master，Master 宕机有 zookeeper 的 paxos 投票机制选取下一任 Master。Master 就算全宕机，也不影响 Region 读写。Master 仅充当一个自动运维角色。
- ④ HDFS 为分布式存储引擎，一备三，高可靠，0 数据丢失。
- ⑤ HDFS 的 NameNode 是一个 SPOF。

# 读写性能

## ● Read

- Key/Value
- 客户端 Cache
- 数据读写定位可能要通过最多 6 次的网络 RPC，性能较低。
- 如果 client 上的缓存全部失效，则需要进行 6 次网络来回，才能定位到正确的 region（其中三次用来发现缓存失效，另外三次用来获取位置信息）
- BloomFilter（用于检索一个元素是否在一个集合中，速度快，但有一定的误识别率）

## ● Write

- 使用日志型的数据结构 (WAL, Write-Ahead Logging)
- Cache(MemStore)
- 合并写操作 group commit

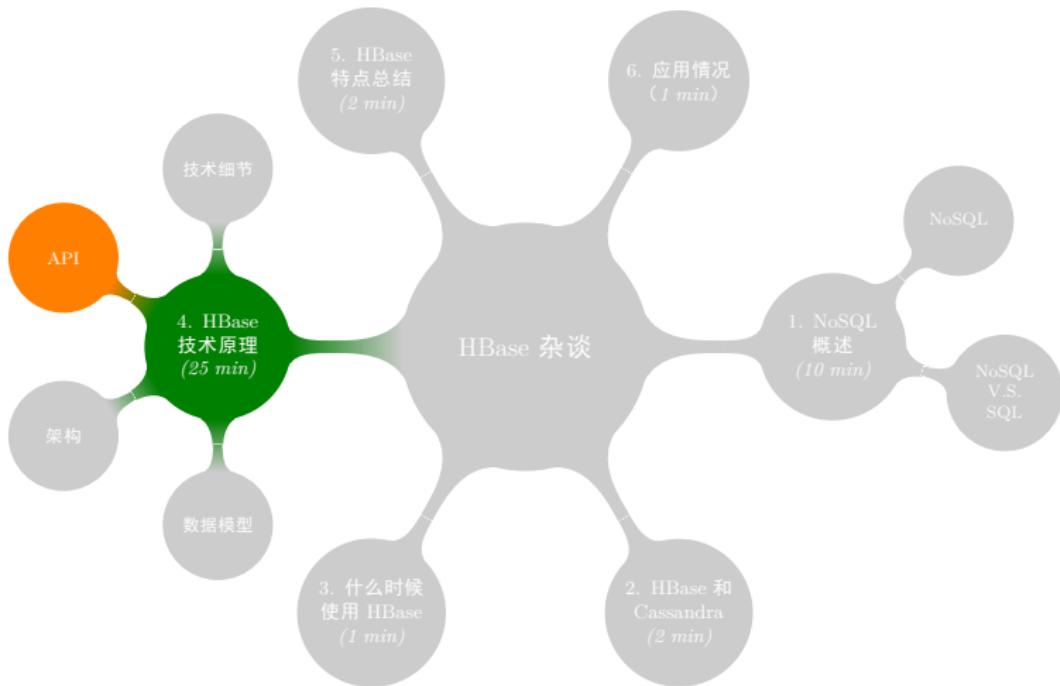
# 伸缩性

## ① 扩容：直接新增机器

- region 的自动分裂以及 master 的 balance；
- 只用增加 datanode 机器即可增加容量；
- 只用增加 regionserver 机器即可增加读写吞吐量；
- regionserver 扩容，通过将自身发布到 Master，Master 均匀分布。

## ② Schema 变化：动态增删列（族）

# HBase API



# HBase API

- Java API
  - Get
  - Put
  - Delete
  - Scan
  - HBaseAdmin
  - MapReduce
- HBase shell (like mysql/hive)
- Thrift
- REST
- Jython, Scala, Groovy DSL, Cascading, Pig, Hive...

# Get/Scan

Gets 是在 Scan 的基础上实现的。下面的讨论 Get 同样可以用 Scan 来描述。

## ① 默认 Get 例子

```
Get get = new Get(Bytes.toBytes("row1"));
Result r = htable.get(get);
byte[] b = r.getValue(Bytes.toBytes("cf"), Bytes.toBytes("attr"))
; // returns current version of value
```

## ② 含有版本的 Get 例子

```
Get get = new Get(Bytes.toBytes("row1"));
get.setMaxVersions(3); // will return last 3 versions of row
Result r = htable.get(get);
byte[] b = r.getValue(Bytes.toBytes("cf"), Bytes.toBytes("attr"))
; // returns current version of value
List<KeyValue> kv = r.getColumn(Bytes.toBytes("cf"), Bytes.
toBytes("attr")); // returns all versions of this column
```

# Put

一个 Put 操作会給一个 cell, 创建一个版本, 默认使用当前时间戳, 当然你也可以自己设置时间戳。这就意味着你可以把时间设置在过去或者未来, 或者随意使用一个 Long 值。要想覆盖一个现有的值, 就意味着你的 row,column 和版本必须完全相等。

## ① 不指明版本的例子 (Hbase 会用当前时间作为版本)

```
Put put = new Put(Bytes.toBytes(row));
put.add(Bytes.toBytes("cf"), Bytes.toBytes("attr1"), Bytes.
          toBytes( data));
htable.put(put);
```

## ② 指明版本的例子

```
Put put = new Put( Bytes.toBytes(row ) );
long explicitTimeInMs = 555; // just an example
put.add(Bytes.toBytes("cf"), Bytes.toBytes("attr1"),
          explicitTimeInMs , Bytes.toBytes(data));
htable.put(put);
```

# Delete

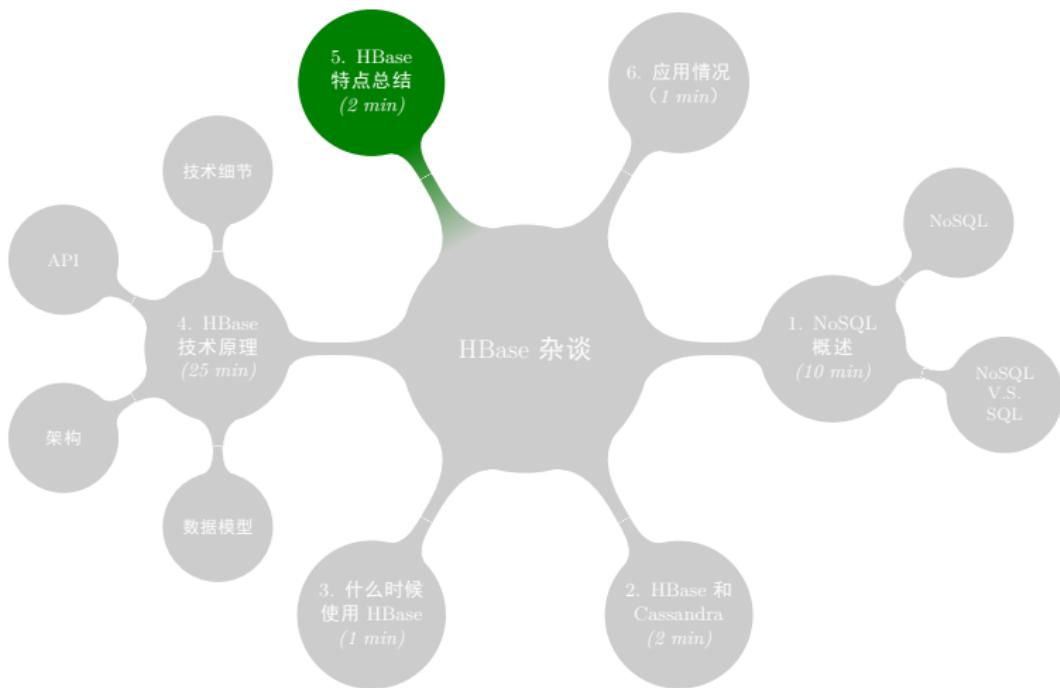
有两种方式来确定要删除的版本。

- 删除所有比当前早的版本。
- 删除指定的版本。

一个删除操作可以删除一行，也可以是一个 column family，或者仅仅删除一个 column。你也可以删除指明的一个版本。若你没有指明，默认情况下是删除比当前时间早的版本。

删除操作的实现是创建一个删除标记。当写下一个删除标记后，只有下一个 major compaction 操作发起之后，这个删除标记才会消失。

# HBase 特点总结



# HBase 特点总结

- 行操作的强一致性
- 线性扩展，自动分表
- 支持 RegionServers 间的自动故障转移
- 和 Hadoop 无缝集成，支持 MapReduce
- 高性能随机写
- 丰富的 API 接口

# HBase 特点总结

- 行操作的强一致性
- 线性扩展，自动分表
- 支持 RegionServers 间的自动故障转移
- 和 Hadoop 无缝集成，支持 MapReduce
- 高性能随机写
- 丰富的 API 接口

# HBase 特点总结

- 行操作的强一致性
- 线性扩展，自动分表
- 支持 RegionServers 间的自动故障转移
- 和 Hadoop 无缝集成，支持 MapReduce
- 高性能随机写
- 丰富的 API 接口

# HBase 特点总结

- 行操作的强一致性
- 线性扩展，自动分表
- 支持 RegionServers 间的自动故障转移
- 和 Hadoop 无缝集成，支持 MapReduce
- 高性能随机写
- 丰富的 API 接口

# HBase 特点总结

- 行操作的强一致性
- 线性扩展，自动分表
- 支持 RegionServers 间的自动故障转移
- 和 Hadoop 无缝集成，支持 MapReduce
- 高性能随机写
- 丰富的 API 接口

# HBase 特点总结

- 行操作的强一致性
- 线性扩展，自动分表
- 支持 RegionServers 间的自动故障转移
- 和 Hadoop 无缝集成，支持 MapReduce
- 高性能随机写
- 丰富的 API 接口

# HBase V.S. Cassandra

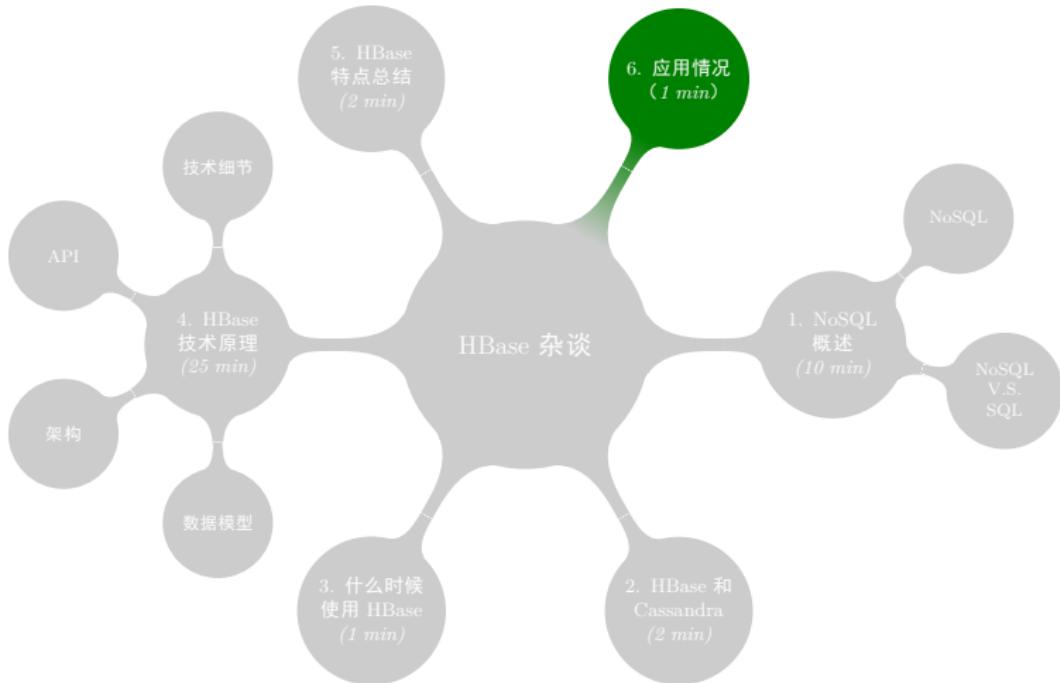
- Cassandra 只有一种节点，而 HBase 有多种不同角色，又架构在 Hadoop 底层平台之上，部署上 Cassandra 更简单；
- Cassandra 的数据一致性策略是可配置的；
- HBase 提供了 Cassandra 没有的行锁机制，Cassandra 要想使用锁需要配合其他系统，如 Hadoop Zookeeper；
- HBase 提供更好的 MapReduce 并行计算支持，Cassandra 在 0.6 版本也提供了这个功能，但还需要有一个 Hadoop 集群来运行它。需要将数据从 Cassandra 集群迁移到 Hadoop 集群。不适合对大型数据运行 MapReduce 任务。；
- Cassandra 的读写性能和可扩展性更好，但不擅长区间扫描。

# My point of view

- HBase 更成熟，Cassandra 更有活力；
- HBase 更加适合于数据仓库、大型数据的处理和分析（如进行 Web 页面的索引等），而 Cassandra 更适合于实时事务处理和提供交互型数据。

更多比较：[Cassandra 和 HBase 主要设计思路对比](#)

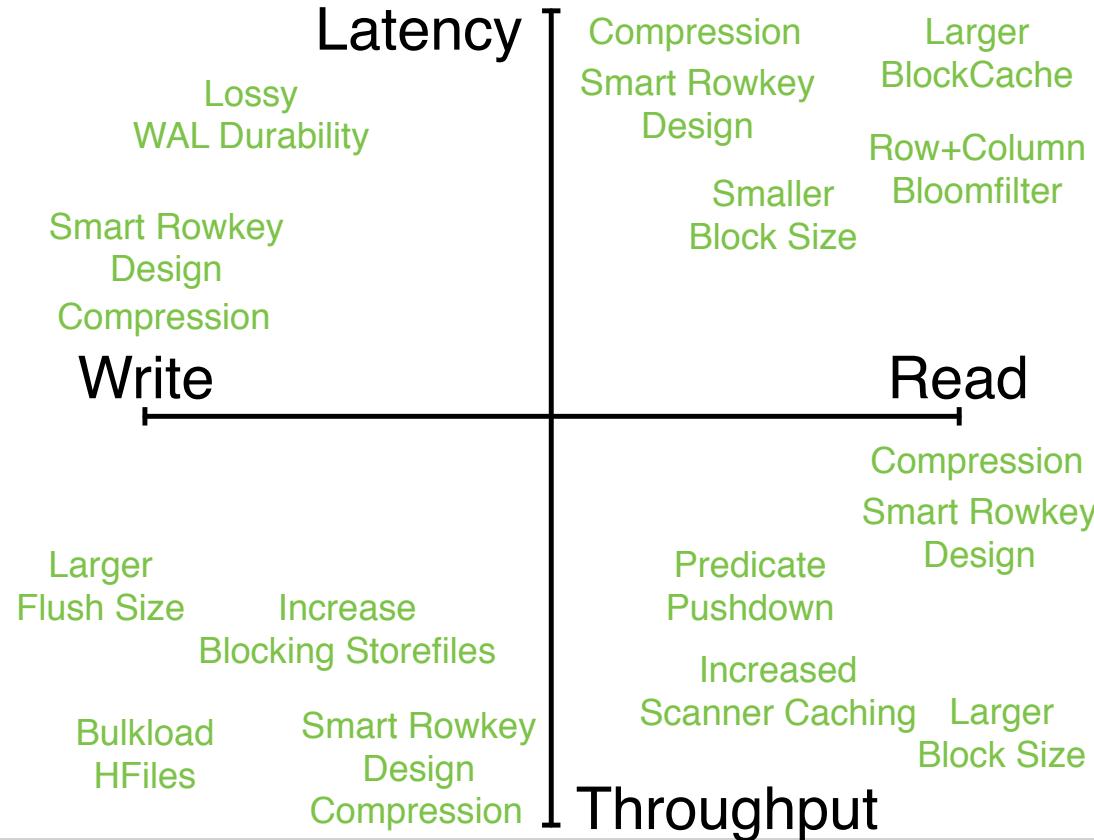
# 应用情况



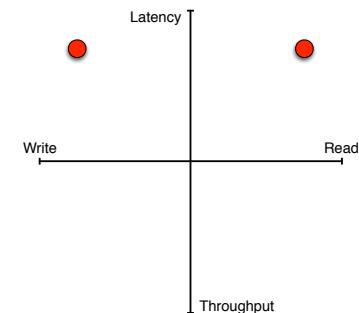
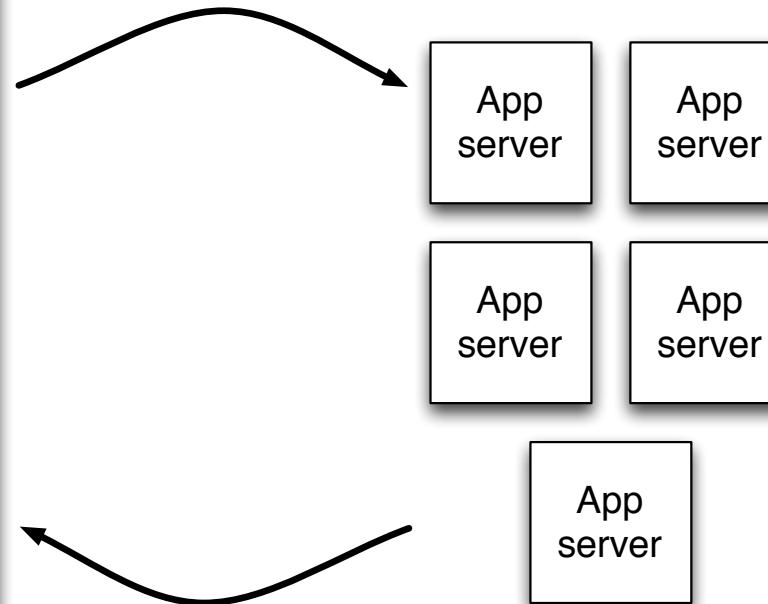
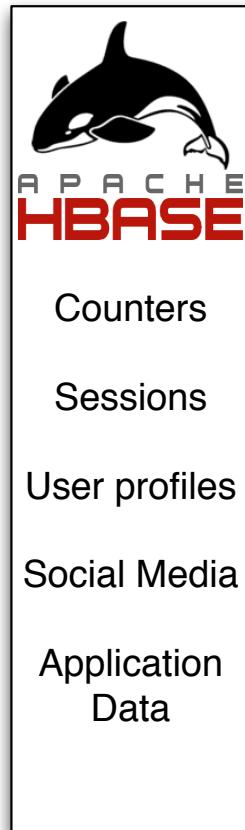


# By Example

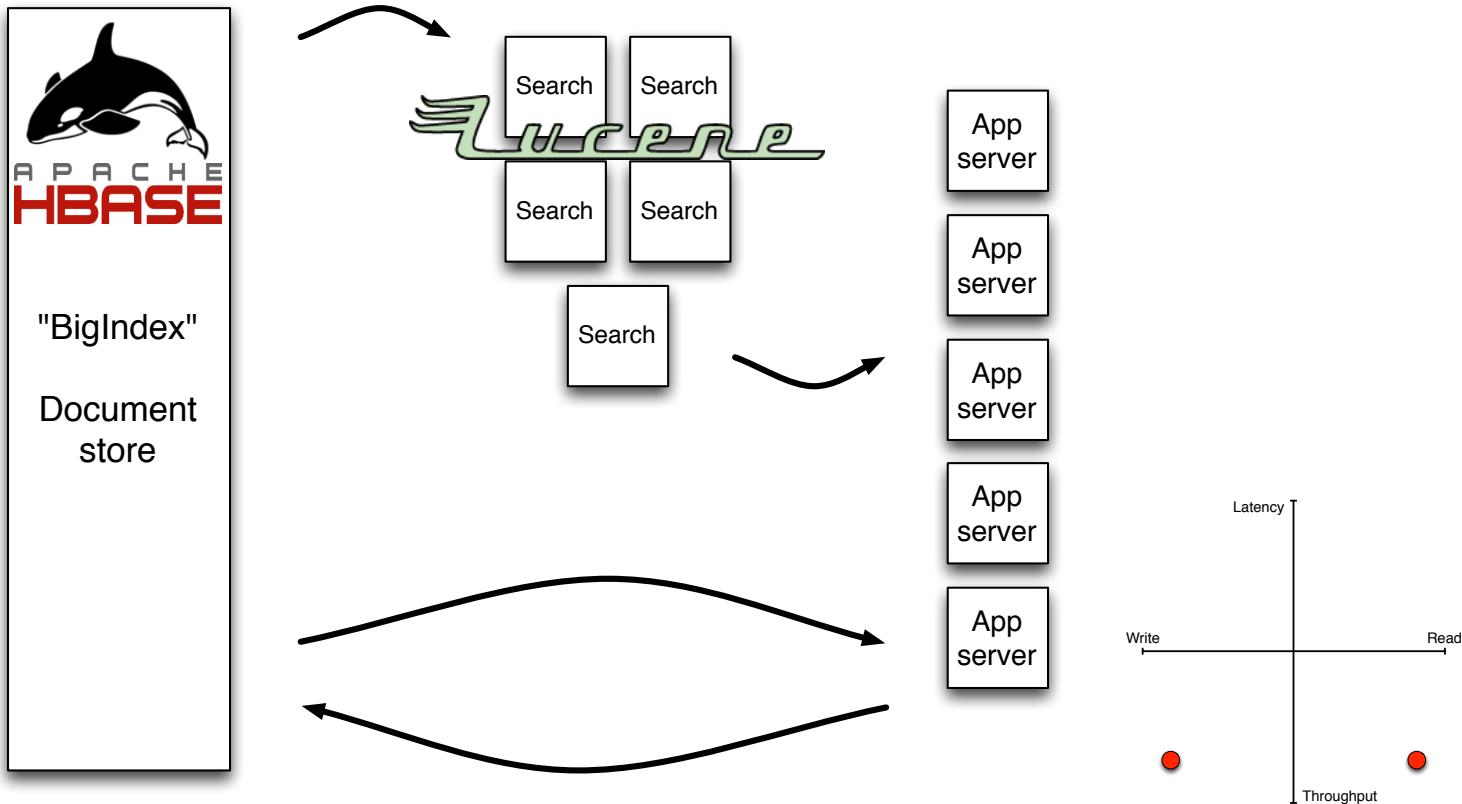
# Database Dichotomy



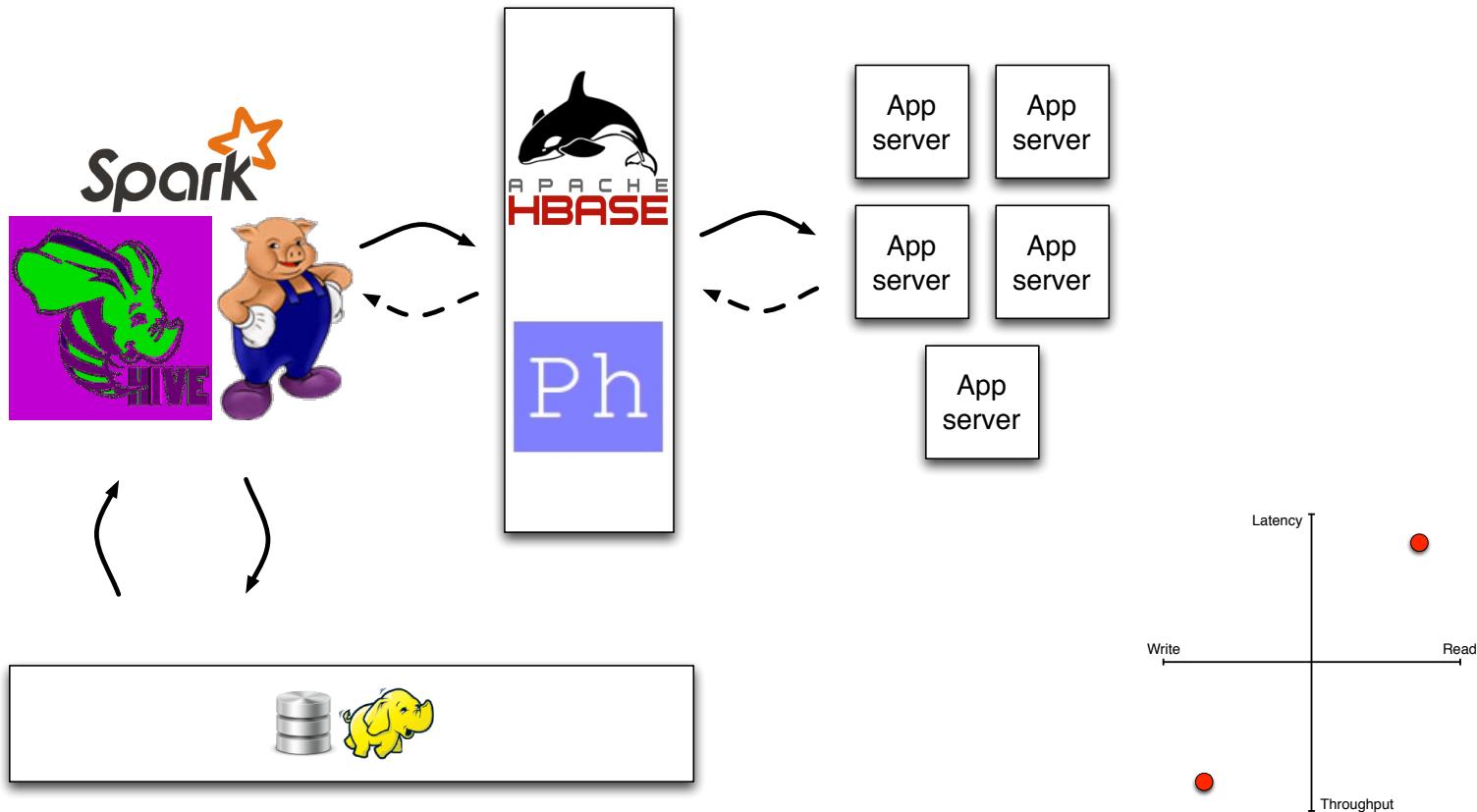
# Web-scale Database



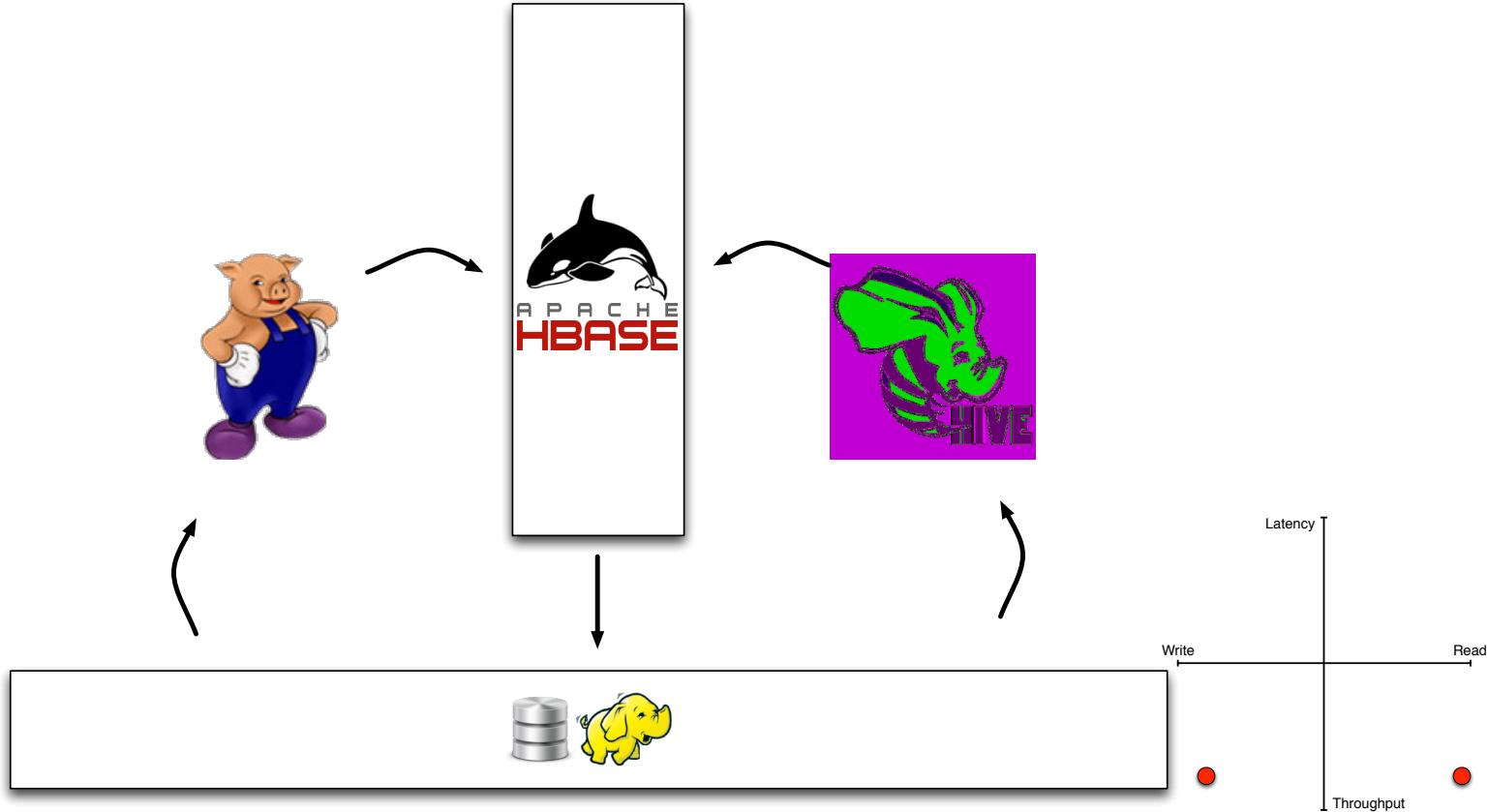
# “BigIndex”



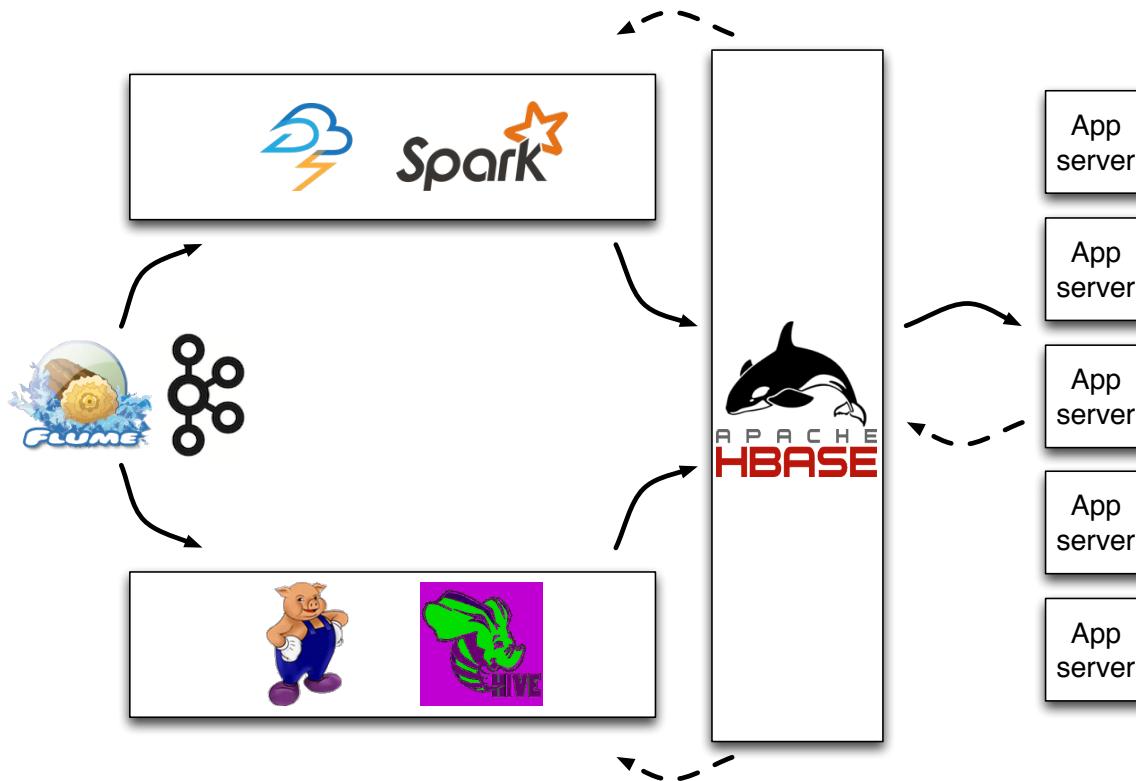
# Materialized View



# ETL Assist



# Lambda Architecture



# 谁在用 HBase ?

- Yahoo!
- Facebook
- Hulu
- LinkedIn
- Last.fm
- 百度
- 阿里巴巴
- 中国移动
- ...

# Thank You!