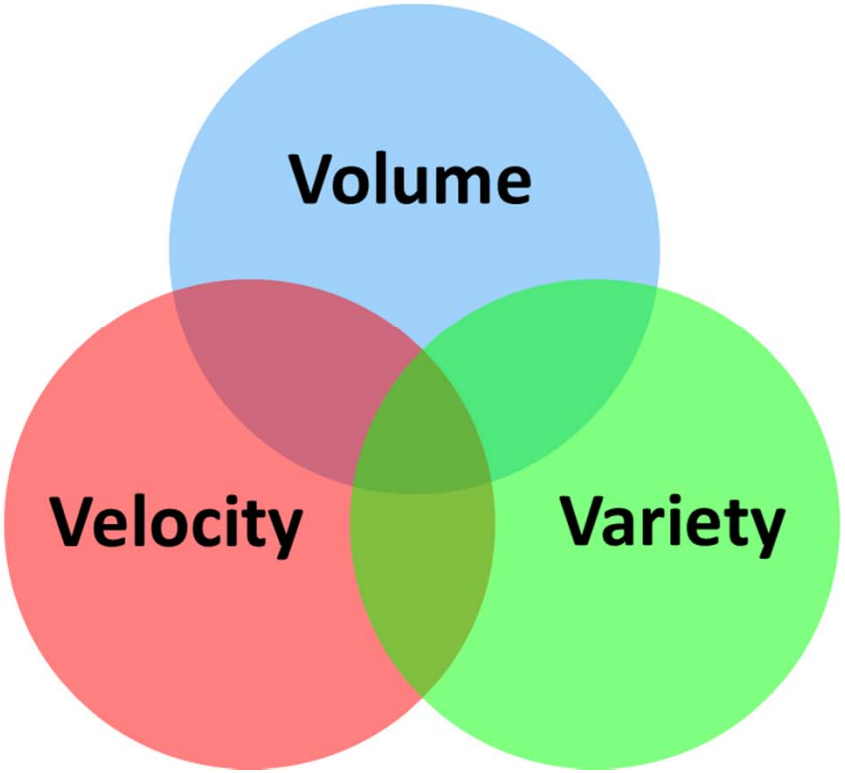
大数据系统与大规模数据分析

大数据存储系统（2）



陈世敏

中科院计算所
计算机体系结构
国家重点实验室

Outline

- Key-Value Store
  - Dynamo
  - Bigtable / Hbase
  - Cassandra
  - RocksDB
- Distributed Coordination: ZooKeeper

为什么叫No-SQL?

- 这些系统大部分是由互联网公司研发的
  - 研发的目标是支持本公司的某类重要的应用
  - 放弃使用关系型系统，转而开发专门的系统以支持目标应用
- 原因1：性能问题
  - 并行数据库系统高配也通常只有几十台服务器
  - 而这些系统则使用成千上万台机器，和存储PB级的数据
- 原因2：功能问题
  - 新的数据类型：图，JSON树状数据类型等
- NoSQL
  - 简化RDBMS的能力：不支持（完全的）SQL，不支持（完全的）ACID
  - 支持非关系的数据模型

为什么叫No-SQL?

- 那么关系型与No-SQL究竟孰优孰劣？
  - 这个不能一概而论
  - 关系型有其生命力，已经存在了40多年，还在被广泛的使用
    - 优美的数学模型支持
    - SQL与ACID等都在实践中被证明了是非常有用的
    - 但是关系型系统的实现确实没有考虑到上述超大规模、多种数据类型
  - No-SQL系统确实很好地支持了它们的目标应用
    - 但是为了支持更加丰富的应用，人们发现已有的No-SQL系统的不足
  - 所以，这两者将以某种方式融合
    - 这种趋势已经出现

Key-Value Store

- Key-Value store是一种分布式数据存储系统
  - 简而言之，数据形式为<key, value>，支持Get/Put操作
  - 实际上，多种不同的系统的数据模型和操作各有差异

- 我们将主要介绍下述系统
  - Dynamo: 由Amazon公司研发
  - Bigtable / HBase: Bigtable起源于Google公司, Hbase是开源实现
  - Cassandra：由Facebook研发，后成为Apache开源项目
  - RocksDB: 由Facebook研发，是在Google LevelDB基础上形成的

Key-Value Store: Dynamo

- "*Dynamo: Amazon's Highly Available Key-Value Store*." Guiseppe DeCandia, Deniz Hastorun, Madan Jampani, et al. (Amazon.com). **SOSP 2007**.
- 支持亚马逊公司电子商务平台上运行的大量服务
  - 例如， best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog
  - 存储这些服务的状态信息

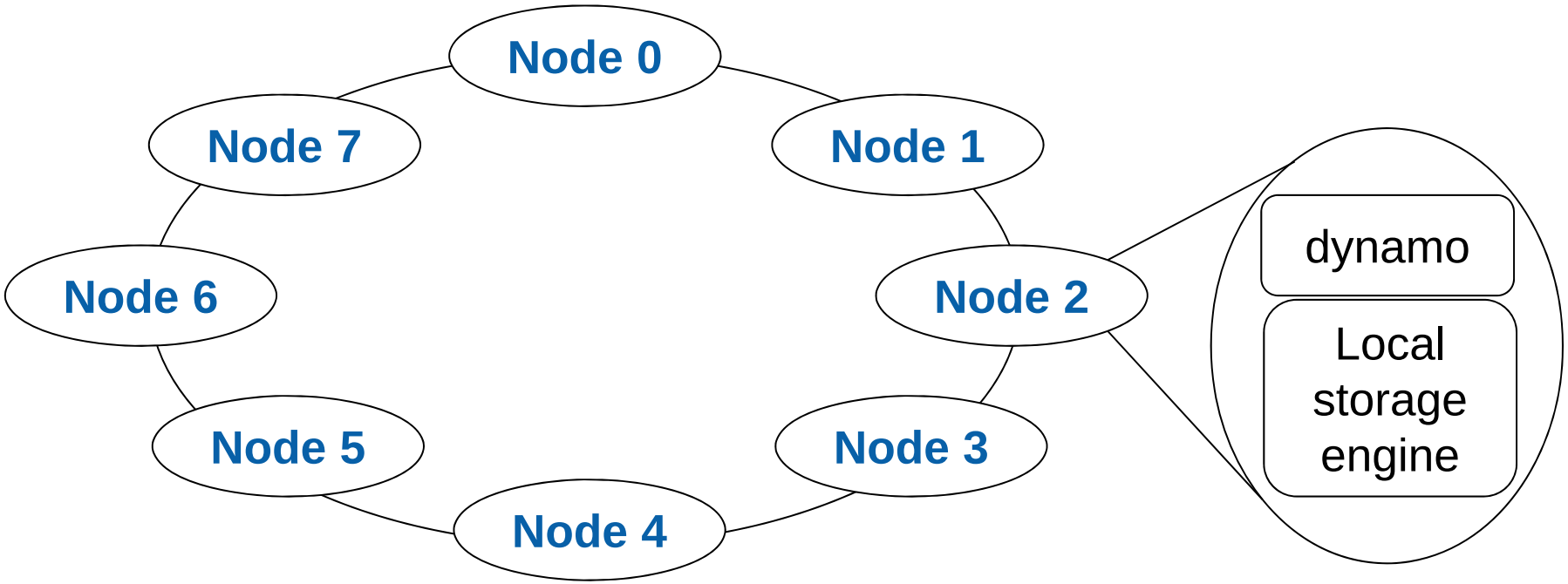
Dynamo数据模型和操作

- 最简单的<key, value>
  - key = primary key：唯一地确定这个记录
  - value： 大小通常小于1MB
- 操作
  - Put(key, version, value)
  - Get(key) ➔ (value, version)
- ACID?
  - 没有Transaction概念
  - 仅支持单个<key,value>操作的一致性

修改多个<key, value>可能出现什么问题？

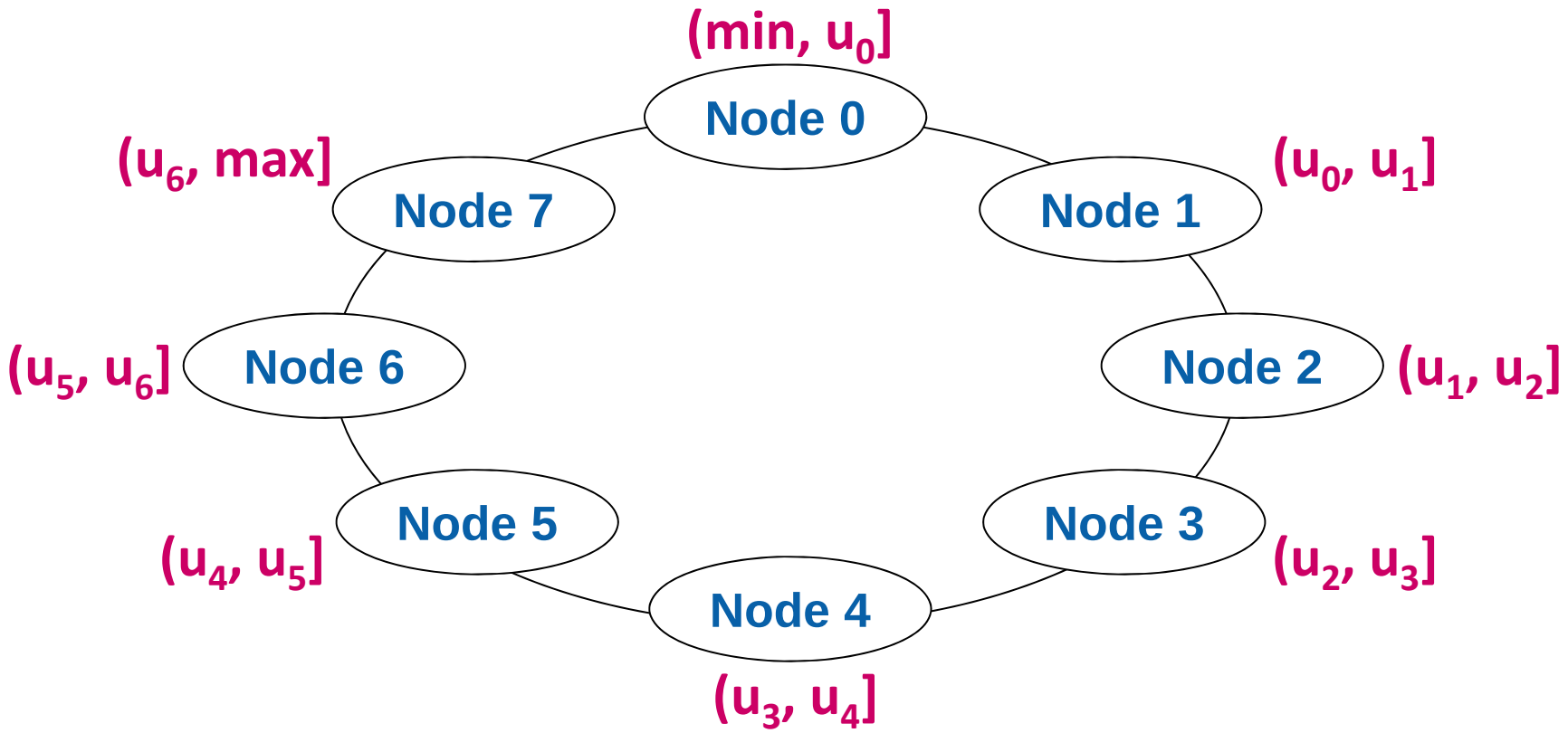
各种不一致情况，要求上层应用设计时考虑这点

Dynamo系统结构



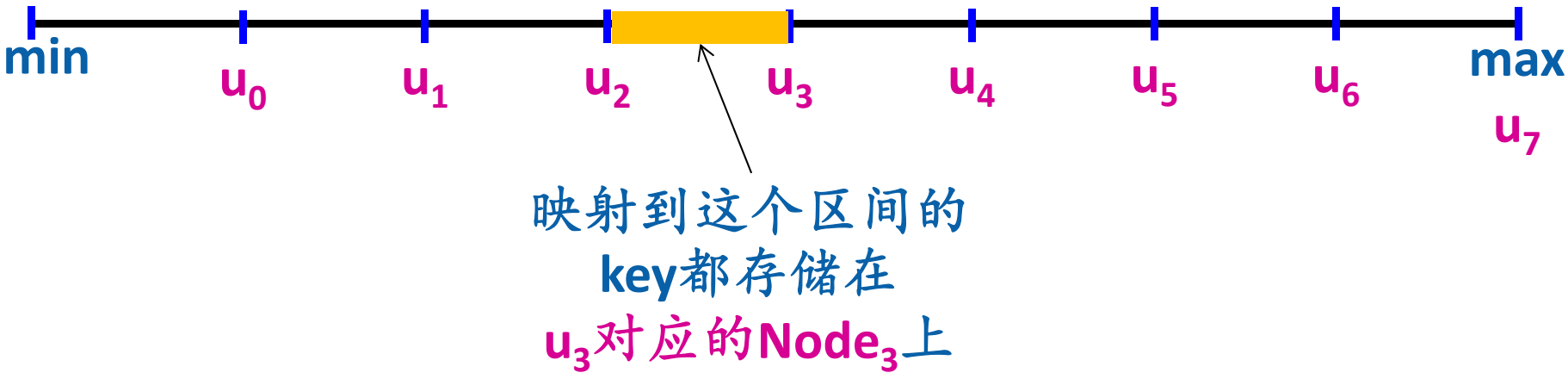
- 多个nodes互连形成分布式系统
- 每个node上由local storage engine + dynamo软件层组成
  - Local storage engine: Berkeley DB, 或MySQL, etc.
  - 用于存储<key, value>

Consistent Hashing (p2p的关键技术)



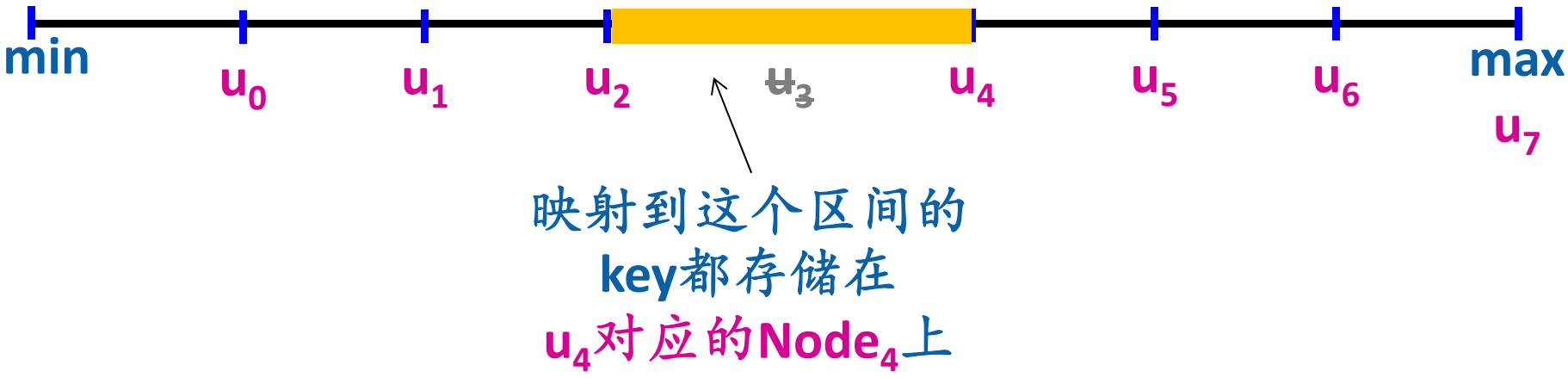
- 把每个key映射为一个token，token∈ (min, max)
  - 例如：token= hash(key)
- 为每个node设置一个token值：min < u_0 < u_1 < u_2 … <u_7=max
  - $Node_j$的token值为u_j, 每个node对应一个区间的所有key

Consistent Hashing (p2p的关键技术)

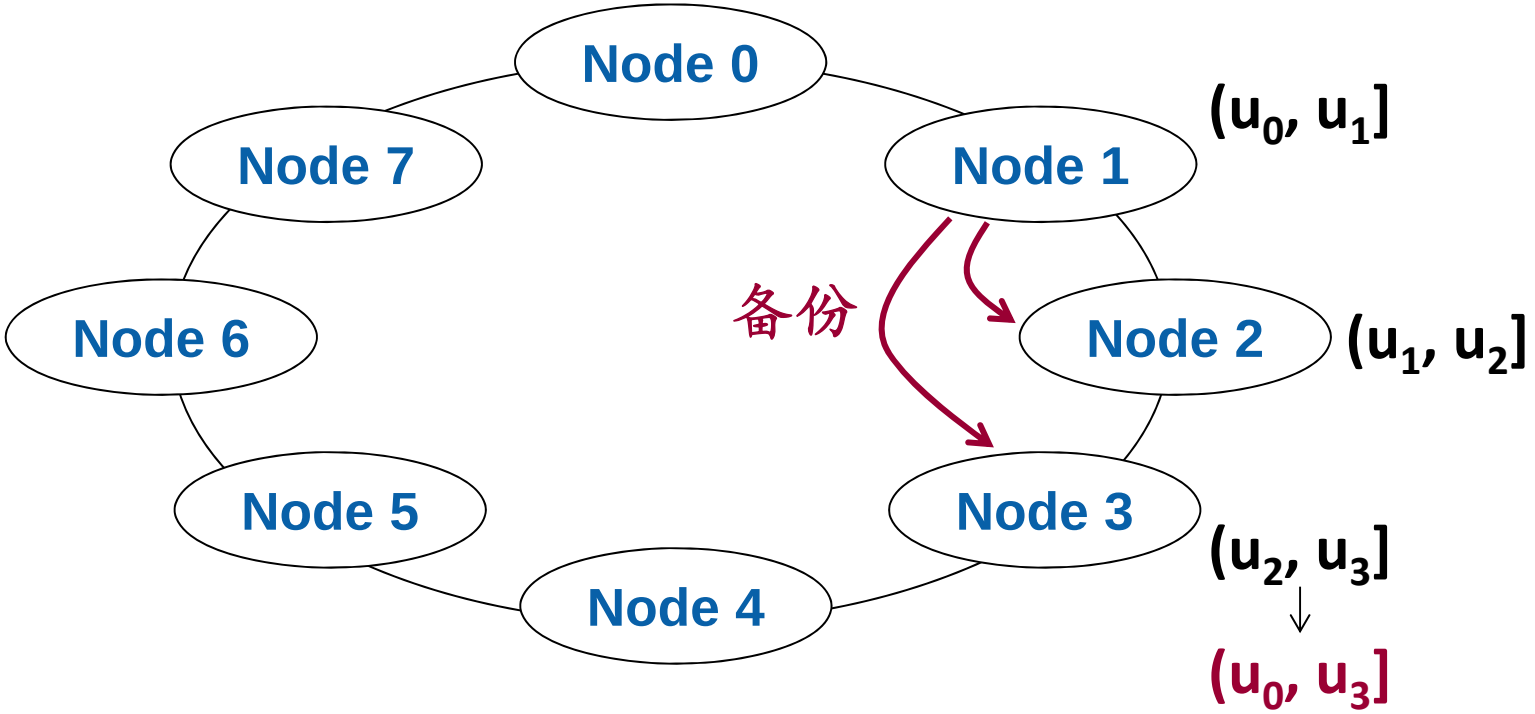


Consistent Hashing (p2p的关键技术)

如果Node3出现故障，删除u3，key的对应关系非常明确

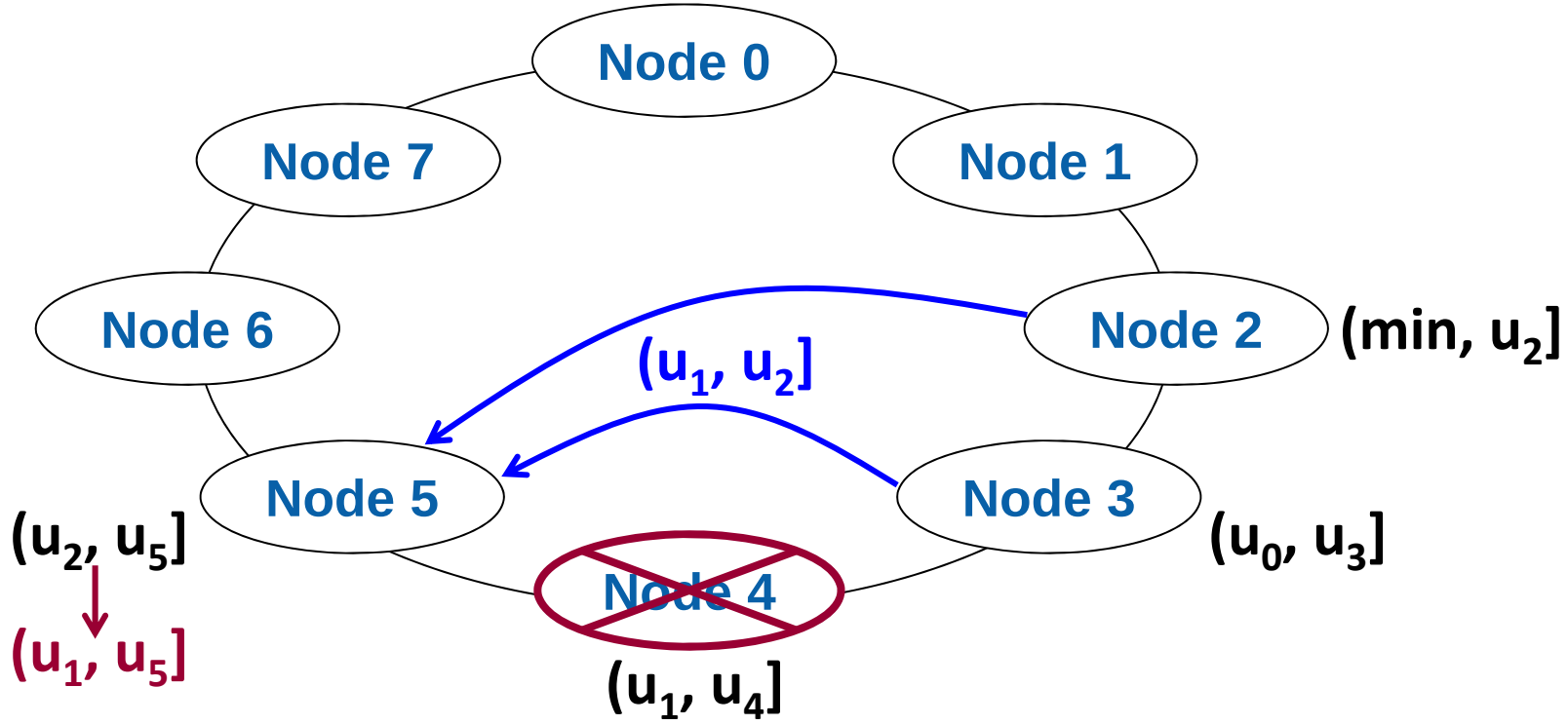


Consistent Hashing：备份



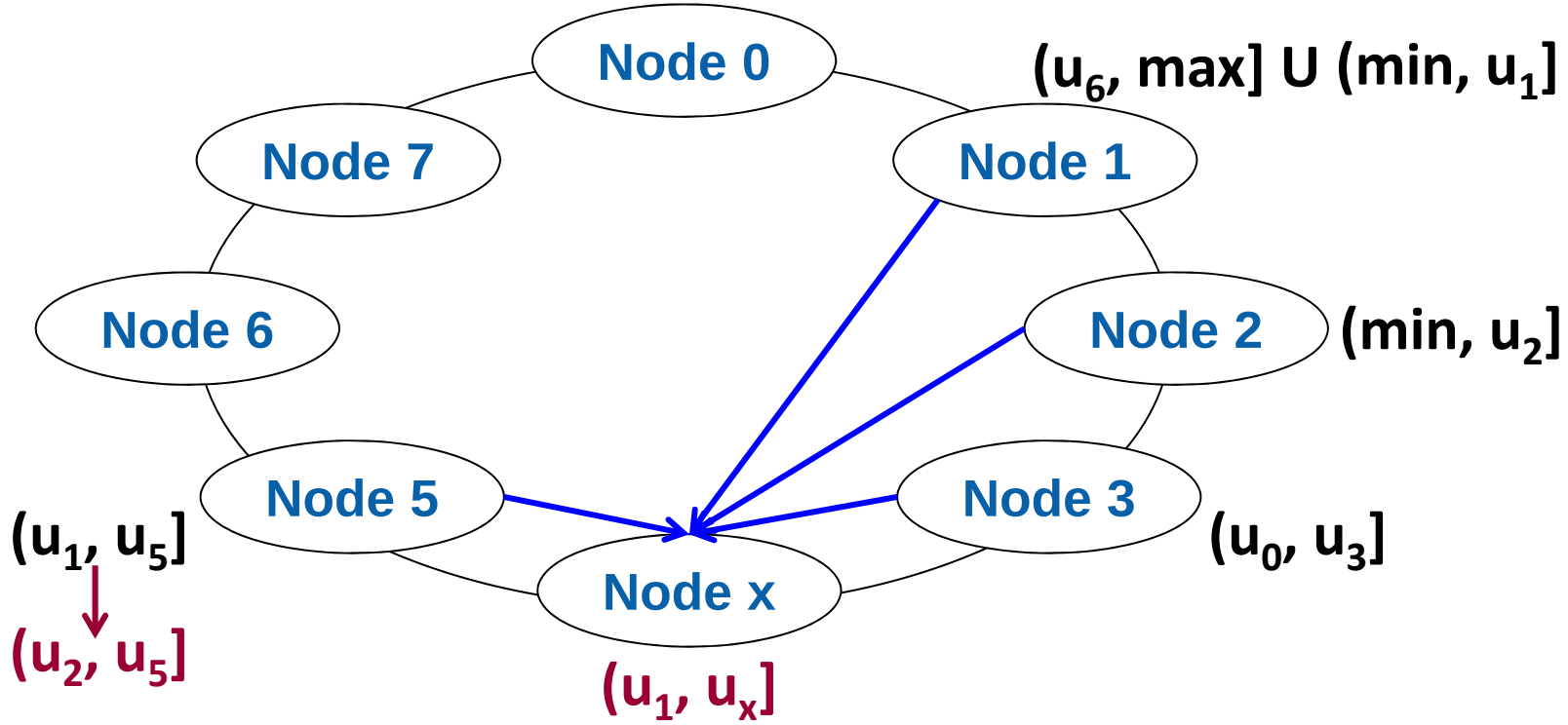
- 举例：3副本备份
  - Put到Node j上的数据，要备份到Node j+1和Node j+2上
  - 在这种设置下，Node j上实际存储的数据是$(u_{j-3}, u_j]$

Consistent Hashing: 减少一个node



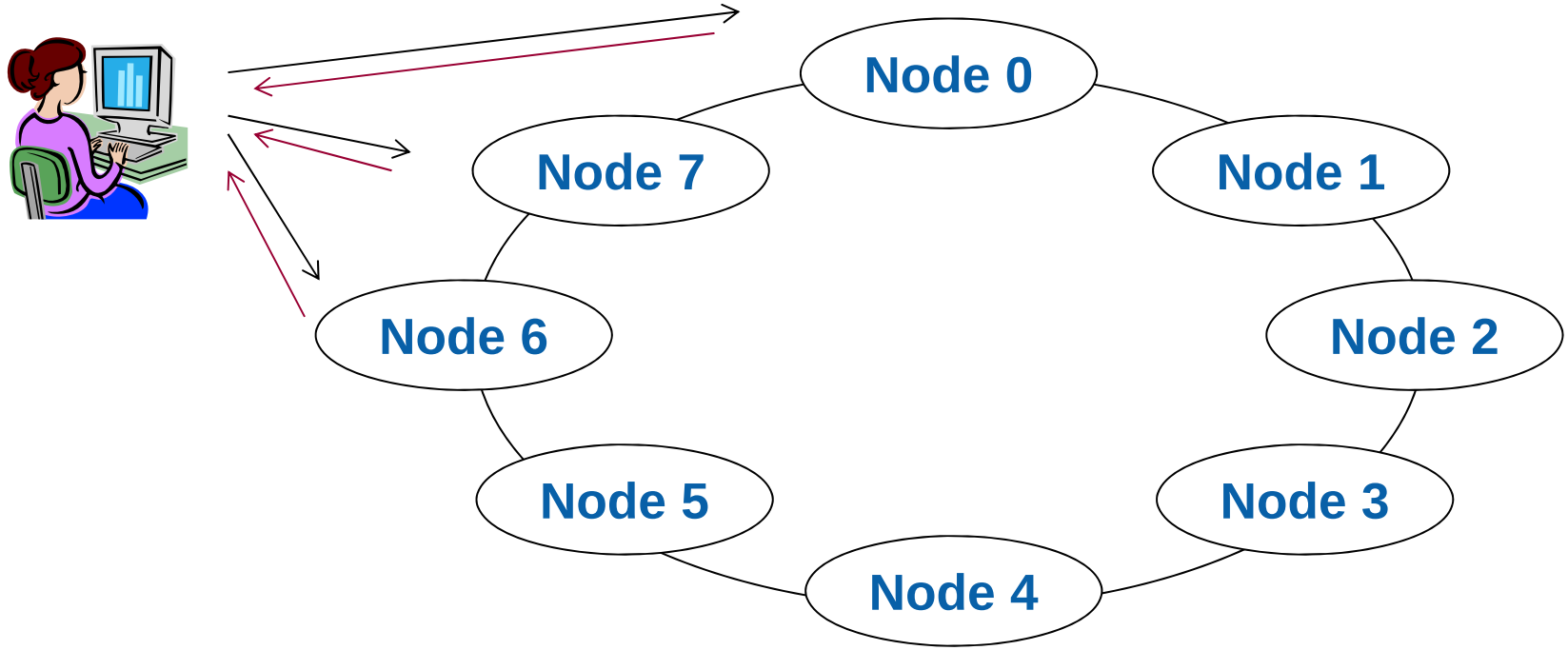
- 改变Node5的区间，拷贝数据
- 对Node 6与node 7有类似修改

Consistent Hashing: 增加一个node



- 给新node赋值（假设：u_x在u_3和u_5之间）
- 改变区间，拷贝数据
- 对Node 6与node 7有类似修改

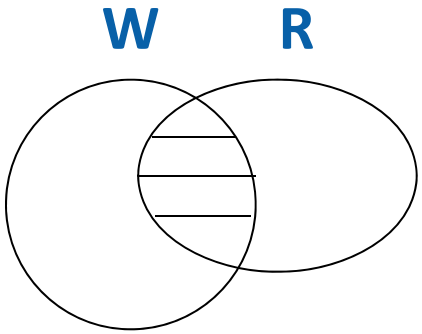
多副本如何进行读写？



- N个副本
- 简单策略：发送N个请求，等待N个node的完成响应
- 但是，可能比较慢（需要等最慢的响应）

Quorum机制：高效+读写一致性

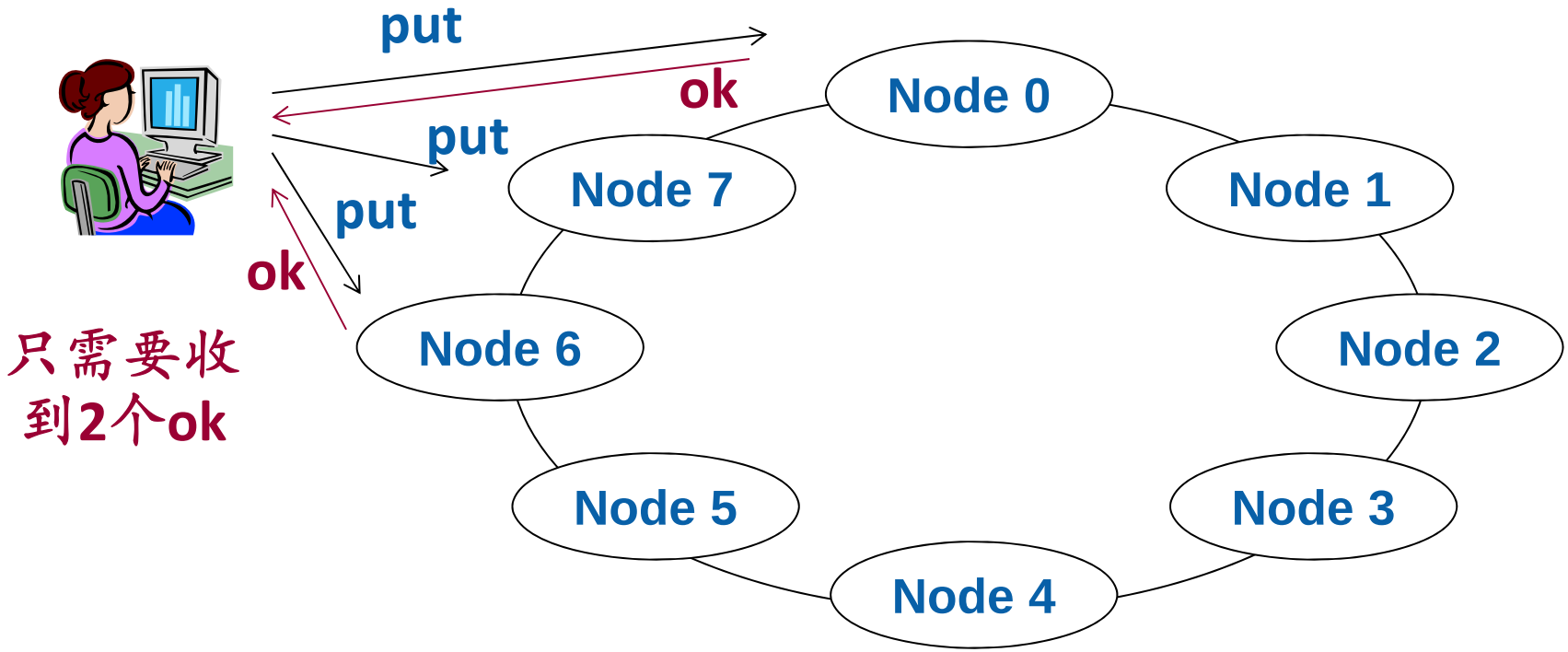
- 问题：多个副本可能存储同一个Key的不同的Value版本，如果能够读到最新数据？



- Quorum (N, W, R)
  - 有N个副本
  - 写：保证>=W个副本的写完成
  - 读：读>=R个副本，选出其中最新版本
- 如果满足**R+W>N**，那么一定读到了最新的数据
- (N, W, R): 例如(3, 2, 2)

Put操作

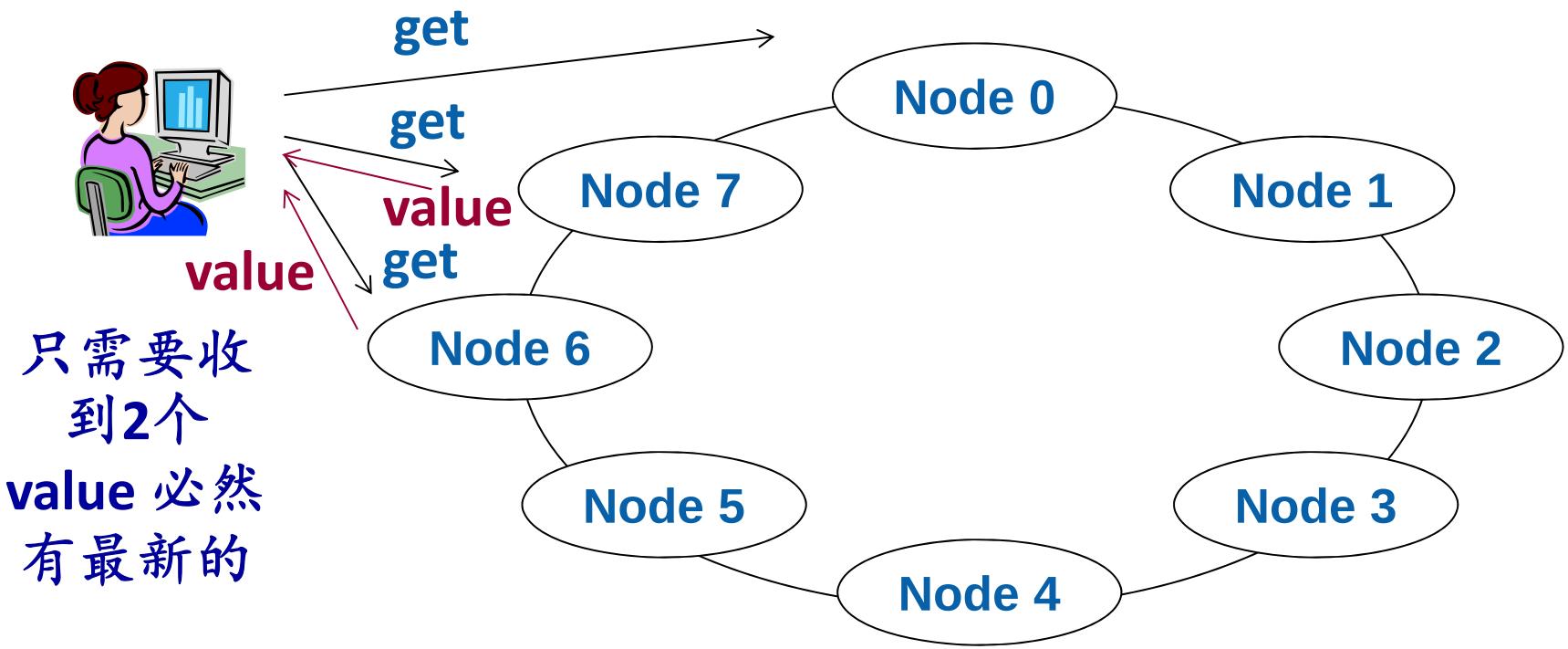
(N, W, R): 例如(3, 2, 2)



- Client根据hash(key)得到所有N个副本所在的节点
- Client向所有N个副本所在的节点发出put
- 等到至少W个节点完成的响应，就认为写成功

Get操作

(N, W, R): 例如(3, 2, 2)



- Client根据hash(key)得到所有N个副本所在的节点
- Client向所有N个副本所在的节点发出get
- 等到至少R个节点的value，就必然包含最新一次写的值

Quorum设计

- N=5
- 哪些是可能的Quorum？（N，R， W）
  - (5, 1, 5)
  - (5, 2, 4)
  - (5, 3, 3)
  - (5, 4, 2)
  - (5, 5, 1)
  - ……
- R小，那么读的效率就高
- W小，那么写的效率就高

Eventual Consistency

- Put操作并没有等待所有N个节点写完成
  - 可以提高写效率
  - 可以避免访问出错/下线的节点，提高系统可用性
- 系统总会最终保证每个<key,value>的N个副本都写成功，都变得一致
  - 但并不保证能够在短时间内达到一致
  - 最终可能需要很长时间才能达到
- 这种“最终”达到的一致性就是eventual consistency

Durability vs. Availability

- Durability：持久性
  - 数据不因为crash/power loss等消失
- Availability：可用性
  - 更进一步，即使出现crash等情况，数据仍然可以被访问
- 在互联网应用中，不仅要durable，而且要available
  - 后者直接关系到用户体验

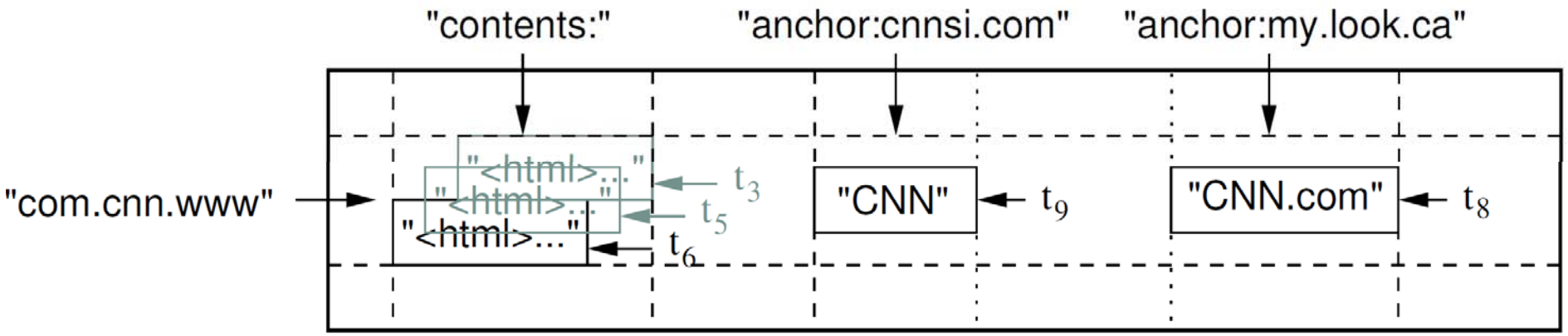
Dynamo小结

- 最简单的<key,value>模型，get/put操作
- 单节点上存储由外部存储系统实现
- 多节点间的数据分布
  - Consistent hashing
  - Quorum (N, W, R)
  - Eventual consistency

Key-Value Store: Bigtable / HBase

- “Bigtable: A Distributed Storage System for Structured Data.” Fay Chang, Jeffrey Dean, Sanjay Ghemawat, et al. (Google). **OSDI 2006**.
- 支持Google多种服务
  - “Bigtable is used by more than sixty Google products and projects, including Google Analytics, Google Finance, Orkut, Personalized Search, Writely, and Google Earth.”
- HBase是Bigtable的Java开源实现，是一个Apache开源项目

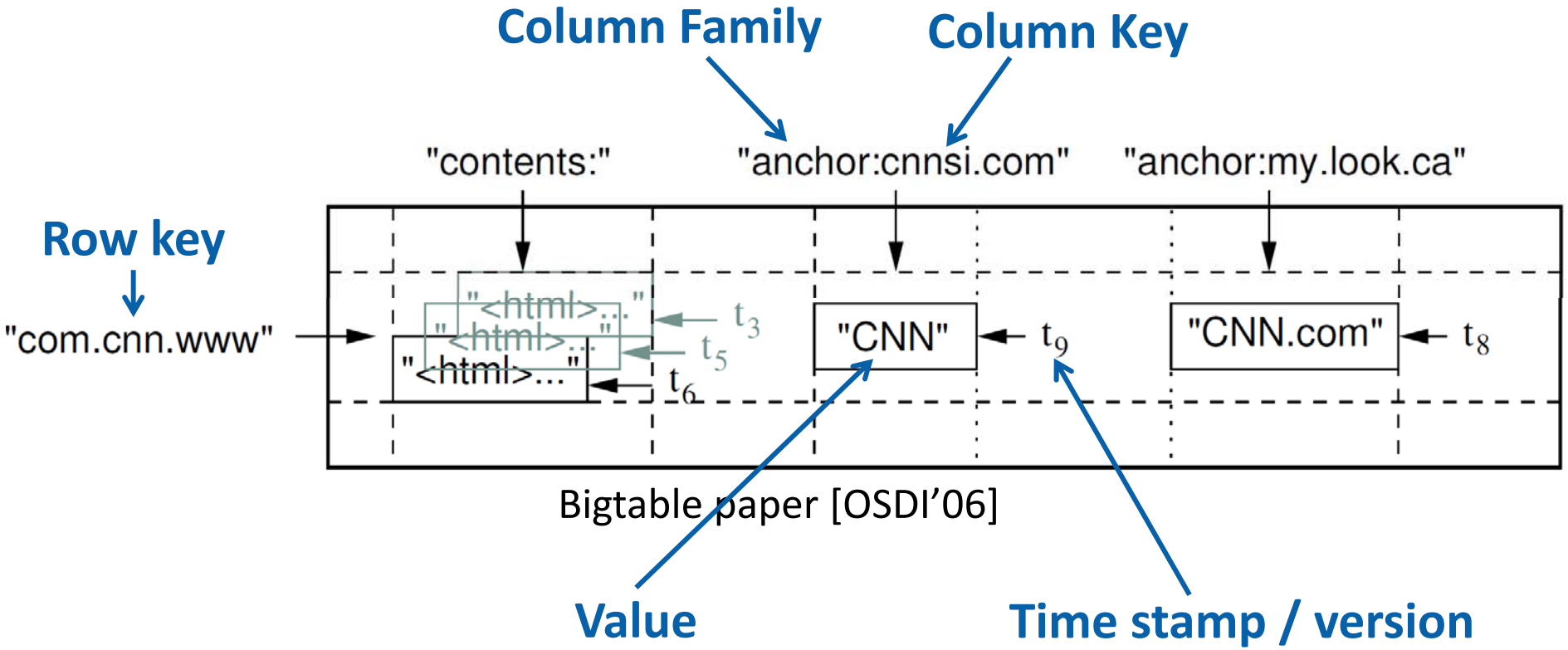
数据模型：举例Bigtable存储Web page



Bigtable paper [OSDI'06]

- Key是domain name的倒置（排序后同一域名会在一起）
- 每个web page记录包含多种类型的信息
  - contents：web page内容
  - anchor：是指向这个web page的源地址和标签信息
- 每个数据都包括产生时间的信息

数据模型：举例Bigtable存储Web page



<row key, column family: column key, version, value>

key

数据模型

key

<row key, column family: column key, version, value>

- Bigtable
  - Key包括row key与column两个部分
  - 所有row key是按顺序存储的
  - 其中column又有column family前缀
    - Column family是需要事先声明的，种类有限（例如~10或~100）
    - 而column key可以有很多
  - 具体存储时，每个column family将分开存储（类似列式数据库）

Key-Value与Relational Schema 忽略version部分

- 简单<key, value>可以对应为一个两列的Table

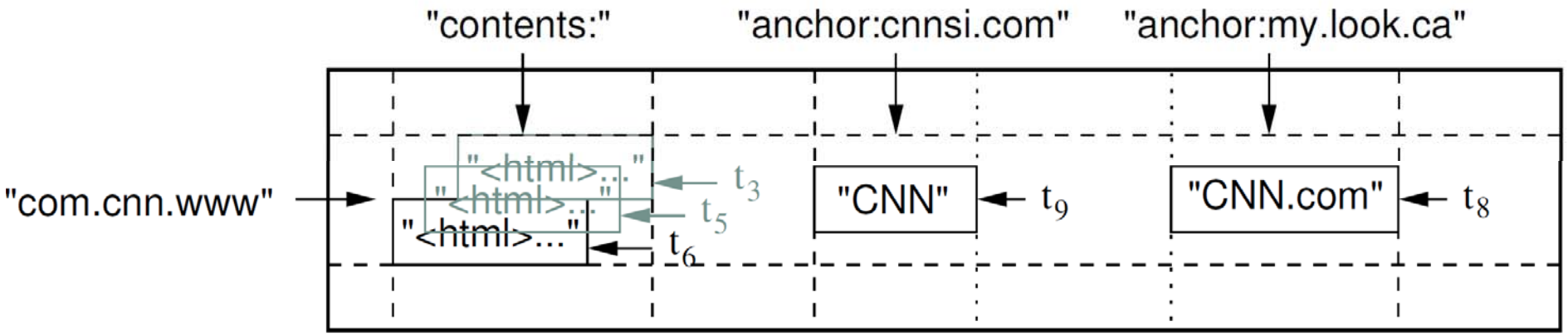
Key	Value
…	…
…	…

- <row key, column family: column key, value>
  每个column family可以对应为一个3列的Table

Row Key	Column family 1’s colum key	Value
…	…	…
…	…	…

Row Key	Column family 2’s column key	Value
…	…	…
…	…	…

Key-Value与Relational Schema 忽略version部分



Row Key	contents	Value
com.cnn.www	‘’	<html>…
…		…

Row Key	anchor	Value
com.cnn.www	cnnsi.com	CNN
com.cnn.www	my.look.ca	CNN.com

Bigtable / Hbase 操作

- Get
  - 给定row key, column family, column key
  - 读取value
- Put
  - 给定row key, column family, column key
  - 创建或更新value
- Scan
  - 给定一个范围，读取这个范围内所有row key的value
  - Row key是排序存储的
- Delete
  - 删除一个指定的value

hbase shell

```
create 'mytable', 'mycf'
      创建表，column family

put 'mytable', 'abc', 'mycf:a', '123'
0 row(s) in 0.0580 seconds

put 'mytable', 'def', 'mycf:b', '456'
0 row(s) in 0.0060 seconds

scan 'mytable'
ROW        COLUMN+CELL
 abc       column=mycf:a, timestamp=1427731972925, value=123
 def       column=mycf:b, timestamp=1427731990058, value=456
2 row(s) in 0.0300 seconds
```

举例： HBase create table & Put

```
public class HBaseTest {
  public static void main(String[] args) throws MasterNotRunningException,
ZooKeeperConnectionException, IOException {
    // create table descriptor
    String tableName= "mytable";
    HTableDescriptor htd = new HTableDescriptor(TableName.valueOf(tableName));

    // create column descriptor
    HColumnDescriptor cf = new HColumnDescriptor("mycf");
    htd.addFamily(cf);

    // configure HBase
    Configuration configuration = HBaseConfiguration.create();
    HBaseAdmin hAdmin = new HBaseAdmin(configuration);

    hAdmin.createTable(htd);
    hAdmin.close();
```

举例： HBase create table & Put

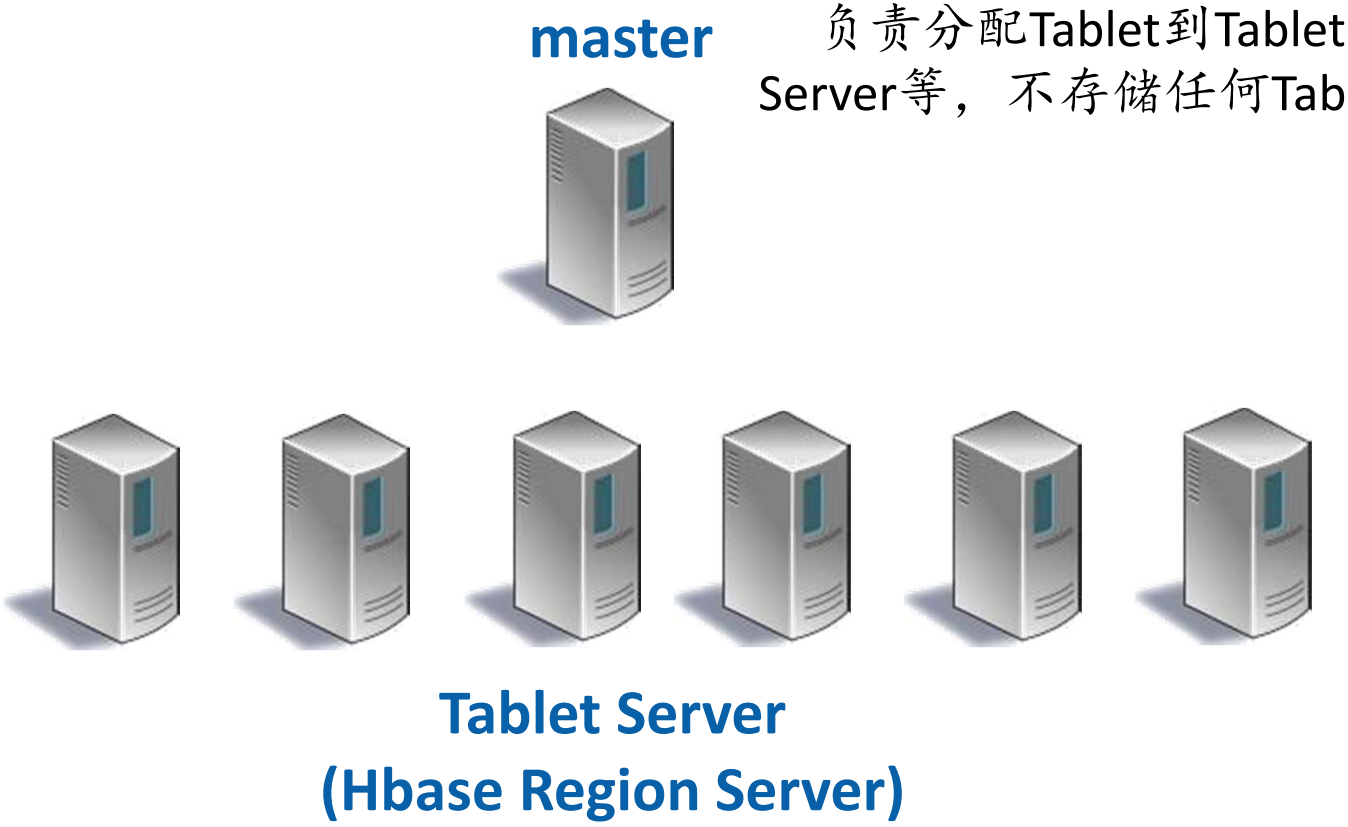
```
    // put "mytable","abc","mycf:a","789"

    HTable table = new HTable(configuration,tableName);
    Put put = new Put("abc".getBytes());
    put.add("mycf".getBytes(),"a".getBytes(),"789".getBytes());
    table.put(put);
    table.close();
    System.out.println("put successfully");
  }
}
```

HBase Scan举例

扫描com.cnn.edition到com.cnn.www之间的数据

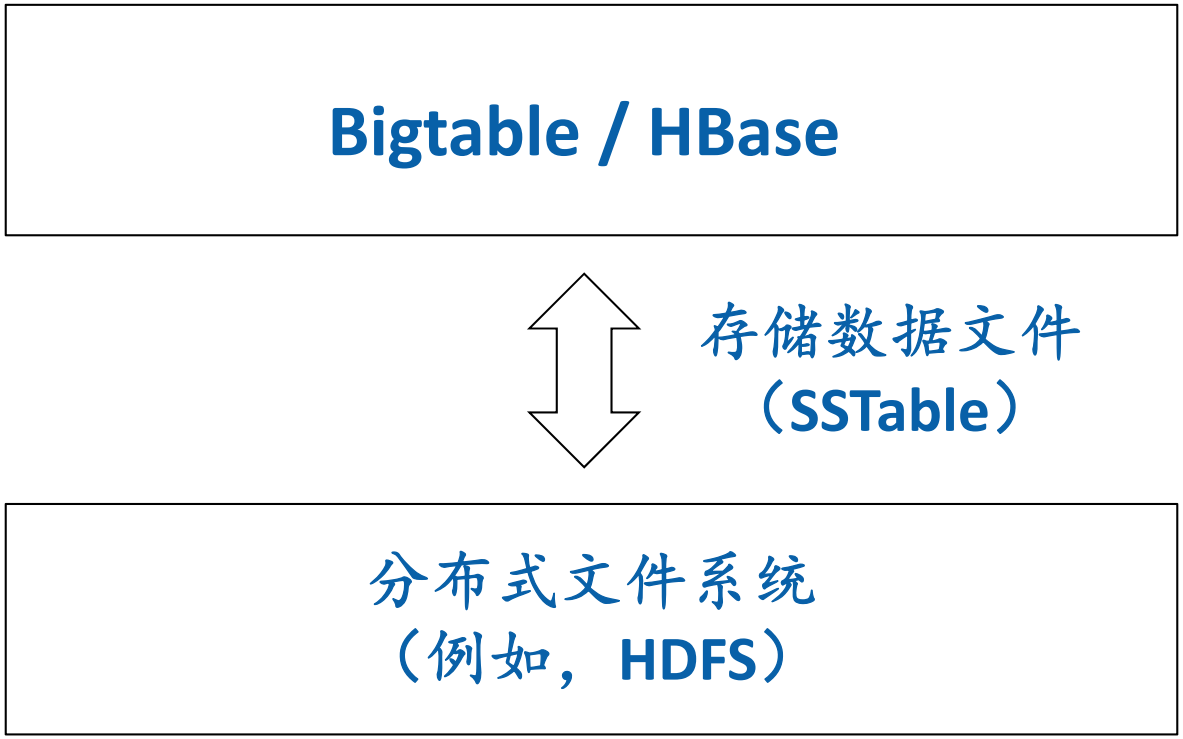
```
HTable htable = ...      // instantiate HTable

Scan scan = new Scan();
scan.addColumn("contents".getBytes(), "".getBytes());
scan.setStartRow(Bytes.toBytes("com.cnn.edition"));
scan.setStopRow(Bytes.toBytes("com.cnn.www"));

ResultScanner rs = htable.getScanner(scan);
try {
  for (Result r = rs.next(); r != null; r = rs.next()) {
       // process result...
  }
} finally {
  rs.close();  // always close the ResultScanner
}
```

Bigtable / HBase系统结构



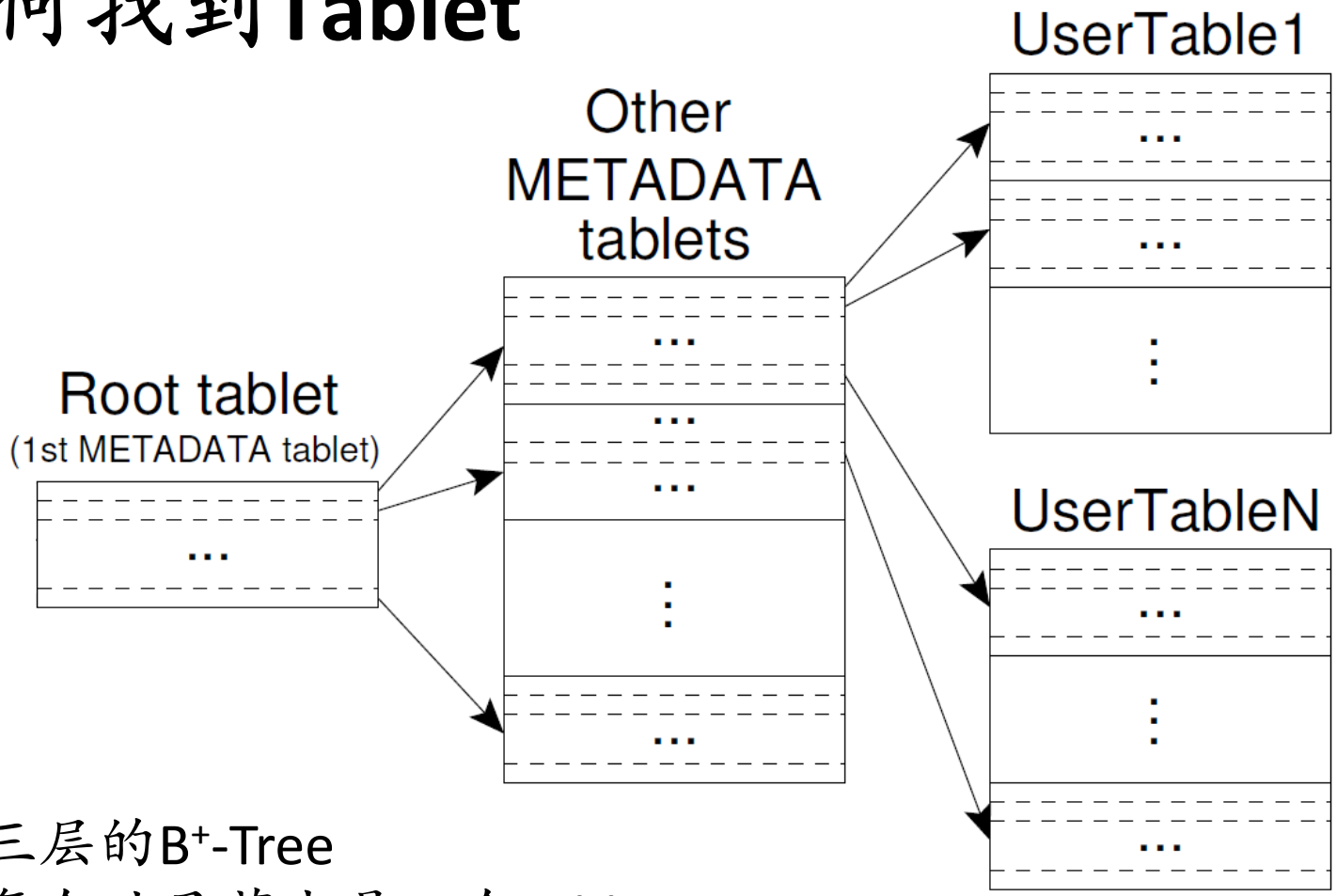
- Tablet是一个分布式Bigtable表的一部分
  - HBase中Tablet被称作Region
  - 我们下面使用Google Bigtable的术语

Bigtable / HBase系统结构



- 注意：数据冗余由下层的分布式文件系统提供，所以在Bigtable中每个Tablet仅存一份

如何找到Tablet



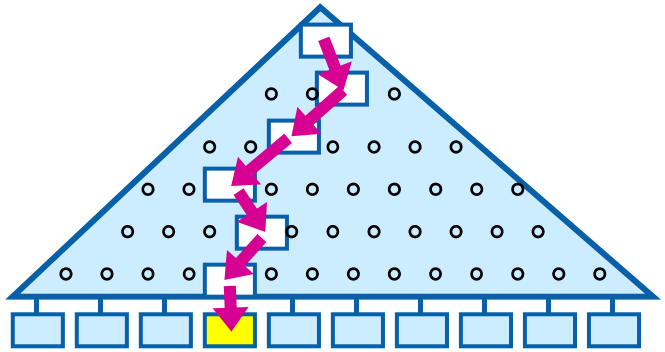
- 三层的B+-Tree
- 每个叶子节点是一个Tablet
- 内部节点是特殊的MetaData Tablet
- MetaData Tablet 包含Tablet位置信息

Bigtable paper [OSDI'06]

单个Tablet内部的存储结构

- 基于Log Structured Merge Tree (LSM-Tree) [O'Neil et al. '96]
  - 有序索引
  - 为写操作而优化

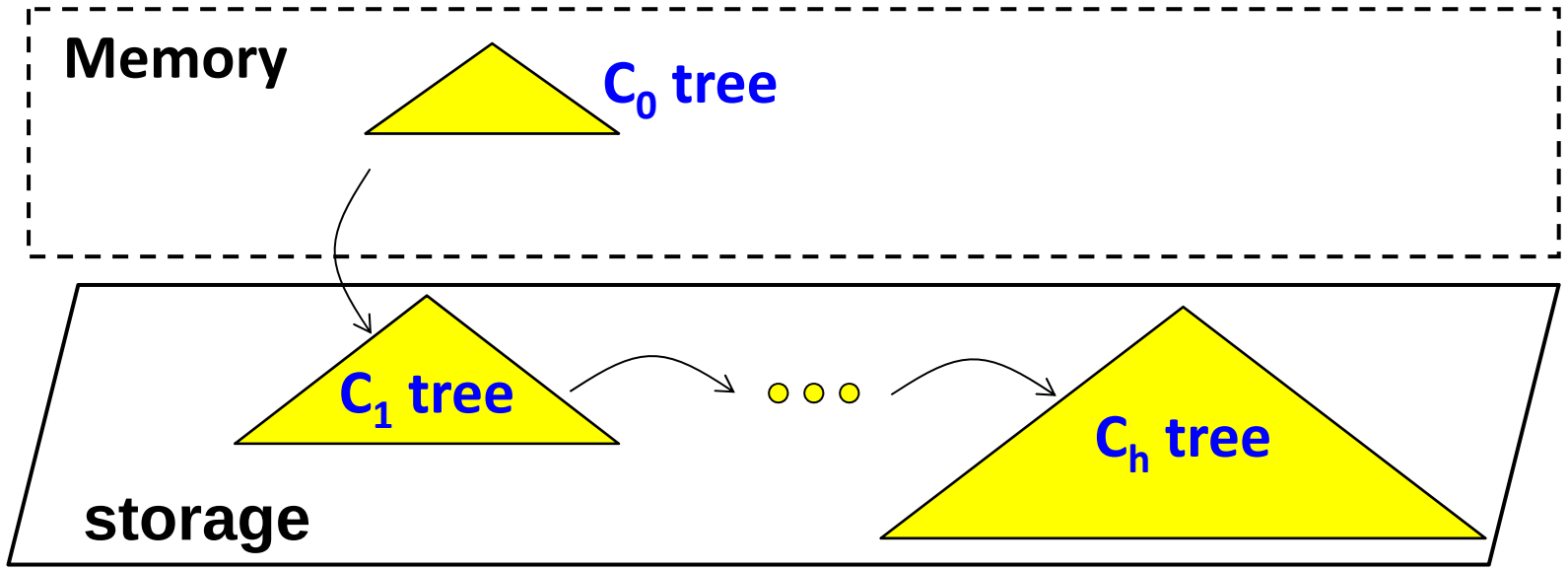
- B+-Tree：每一次Insertion都导致一次随机写



- LSM-Tree目标：减少随机写

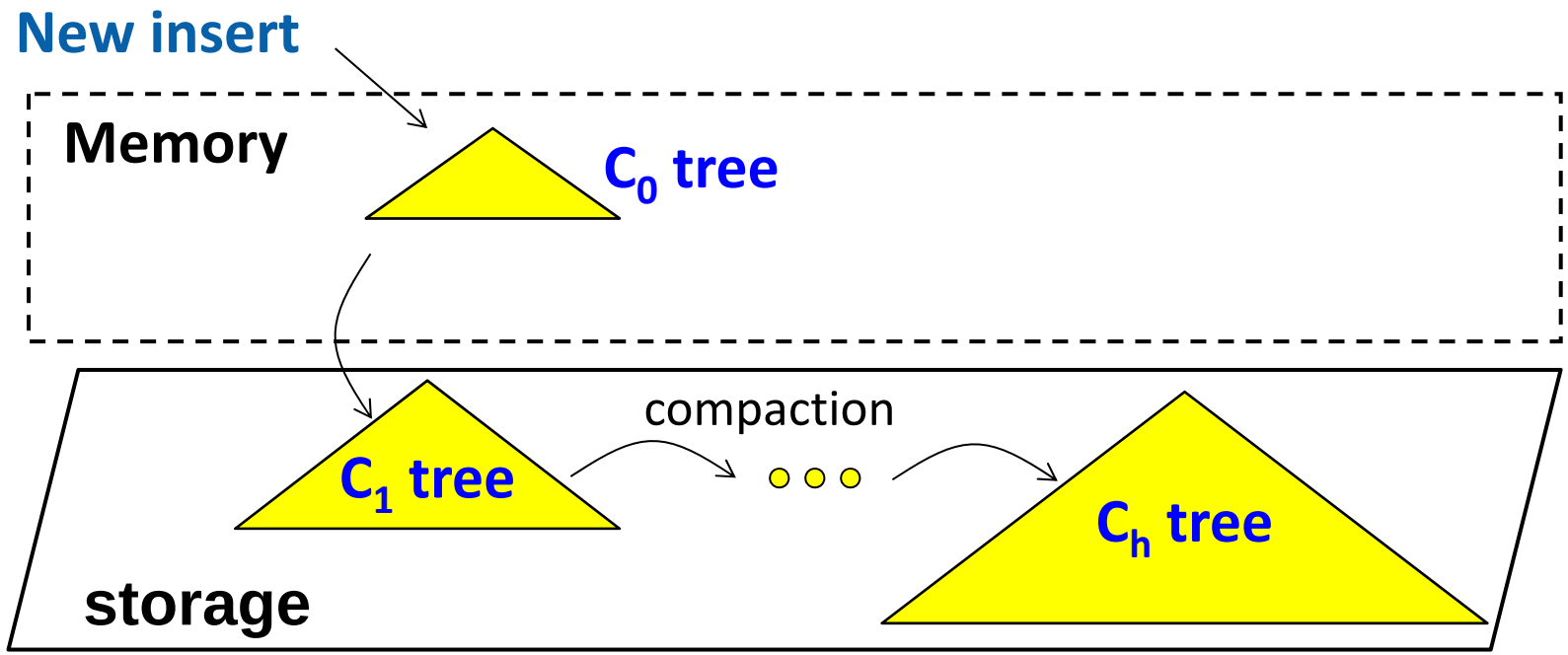
LSM-Tree

[O'Neil et al. '96]



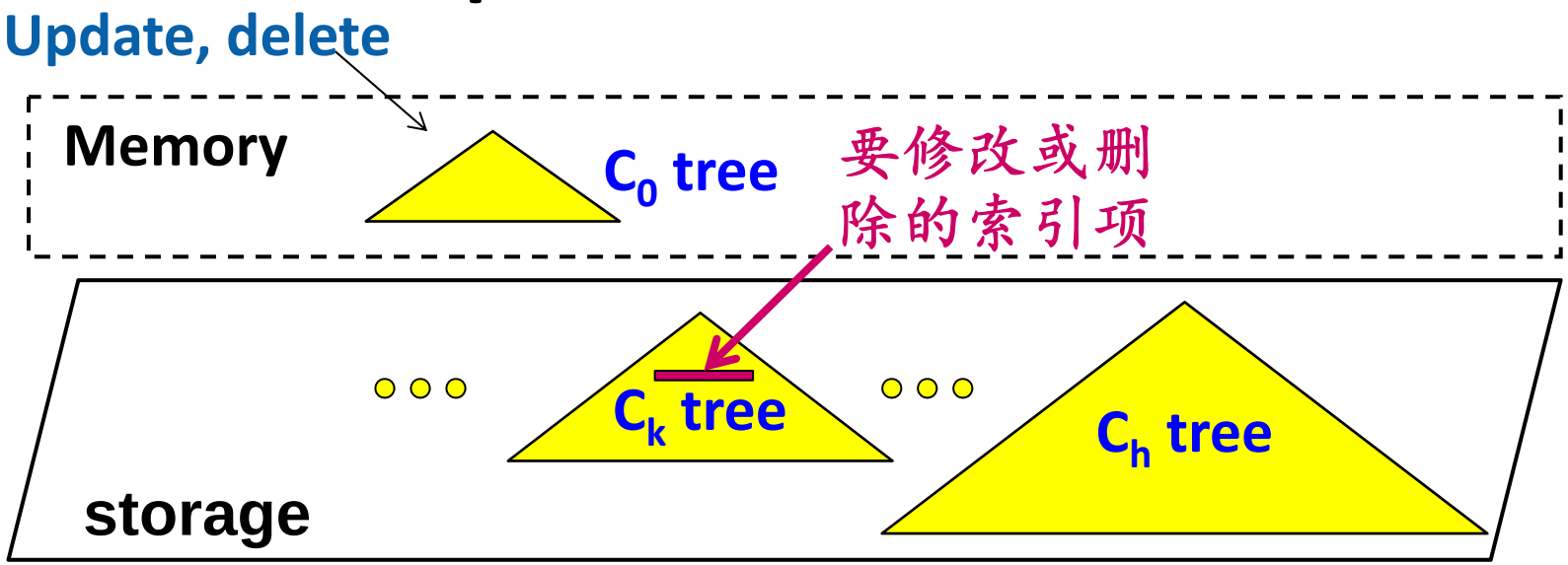
- 由多层组成，大小指数级增长
  - Size(C_k) / size(C_{k-1}) = r，r是相邻层大小的比例
- 每层都是一个有序索引
  - 比如B+-Tree，或者sorted run（排序文件）

LSM-Tree： Insert



- New insert写入内存的C_0中
- 相邻层与层之间进行merge / compaction
  - 例如，同时扫描C_0和C_1两层排序文件，归并形成一个新的C_1。同样地，其他相邻层之间也进行这种归并。
  - 顺序读写操作
- 一个新的insert将逐渐向更深层传递，直到最高C_h层

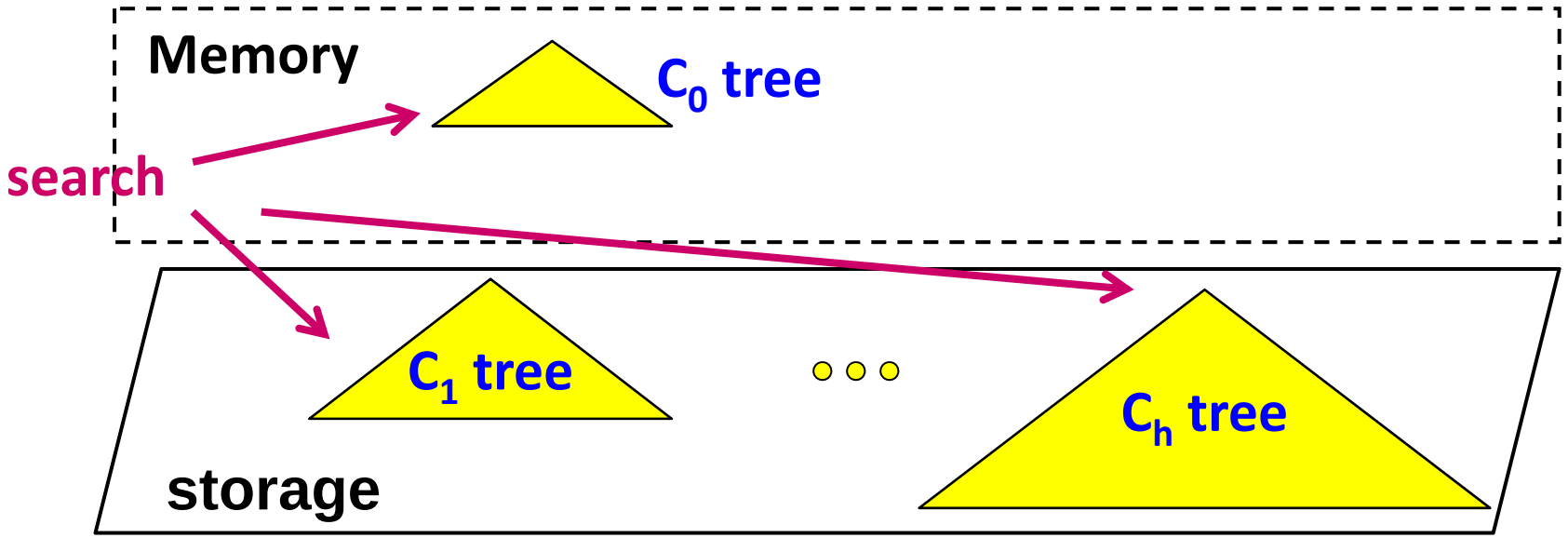
LSM-Tree： Update, Delete



- 不能直接update和delete
  - 原因：被update或delete的索引项可能在某个C_k层
  - 避免随机访问，不能直接操作
- 解决方法：把update和delete都变成insert
  - Update变成了一个insert+new timestamp
  - Delete变成了一个具有Delete标记的特殊insert
  - 在后续compaction时会逐渐去除旧版本

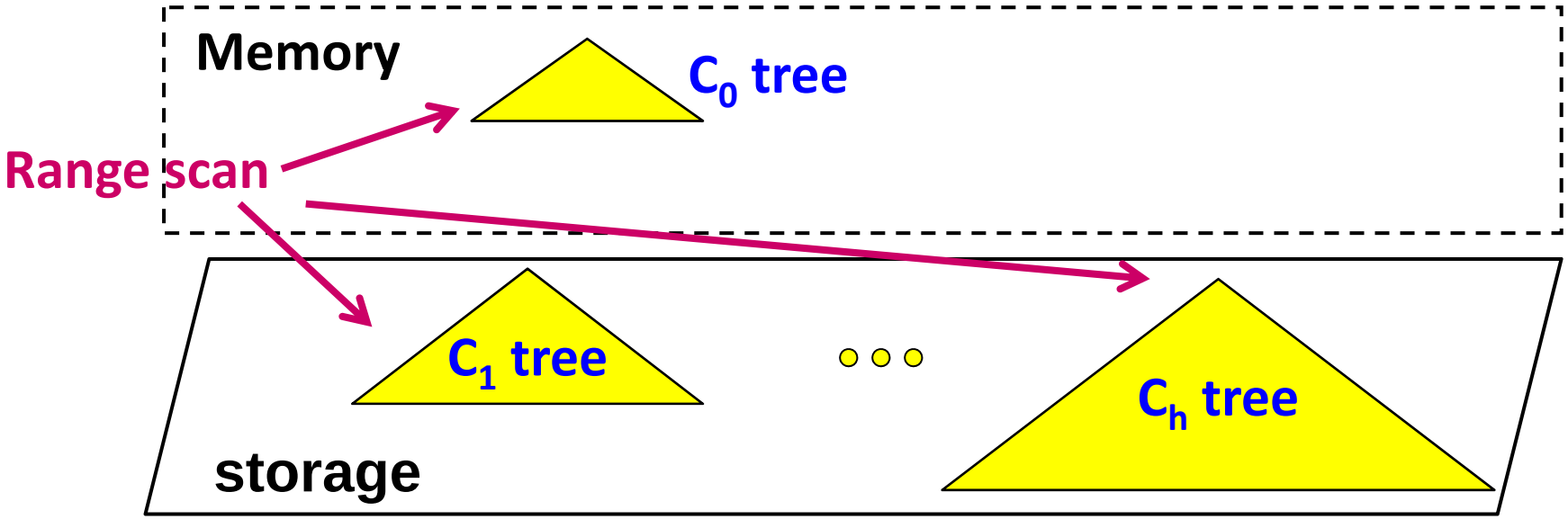
同一个key可能存在多个不同的版本(timestamp)的数据

LSM-Tree： Search



- 问题： 不知道search key在哪一层中？
- 依次访问C_0, C_1, … C_h
  - 第一个找到的版本就是最新版本
- 引起多次随机读访问，比B+-Tree的读性能差
  - 优化：Bloom filter等，可以进一步延伸阅读

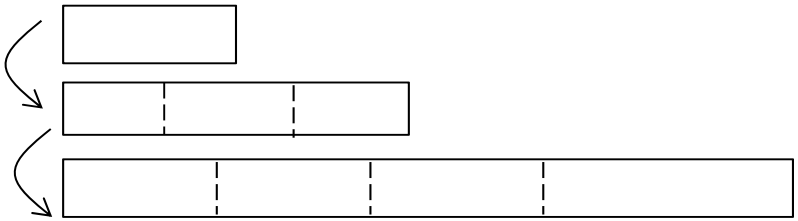
LSM-Tree： Range Scan



- 在所有层C_0, C_1, … C_h都同时进行Range Scan
- 归并多个Scan
- 效率低于B+-Tree

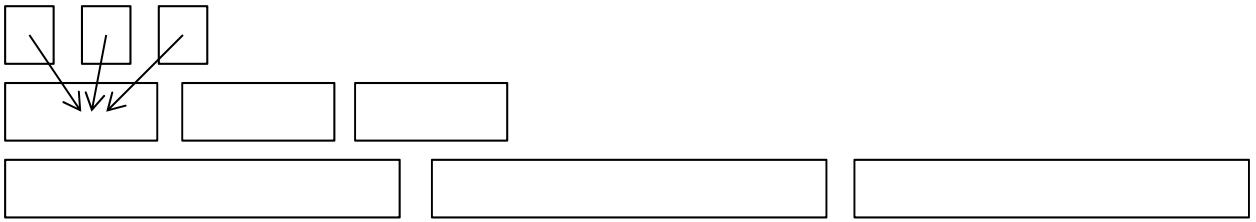
LSM-Tree外存层次的组织

- Leveling：基本设计
  - 每层（逻辑上）仅有一个大的排序文件
  - 实际上可分成多段，每段对应一个区间（参照B+-Tree叶子）

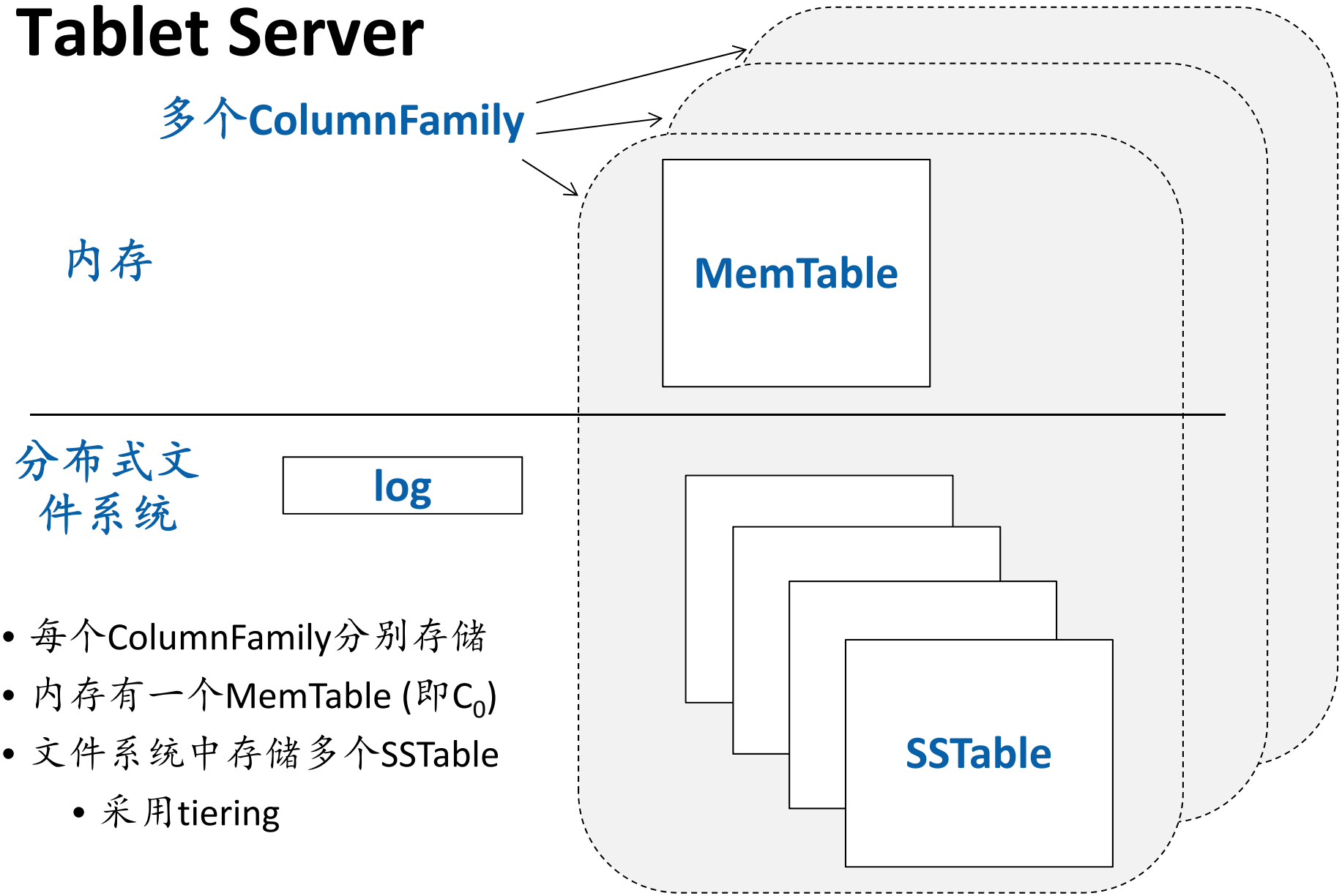


- Tiering：compaction总I/O少，但排序文件更多，读性能下降
  - 每层有多个排序文件，它们之间是有重叠的
  - Compaction归并上层多个排序文件，形成一个新的大文件

compaction

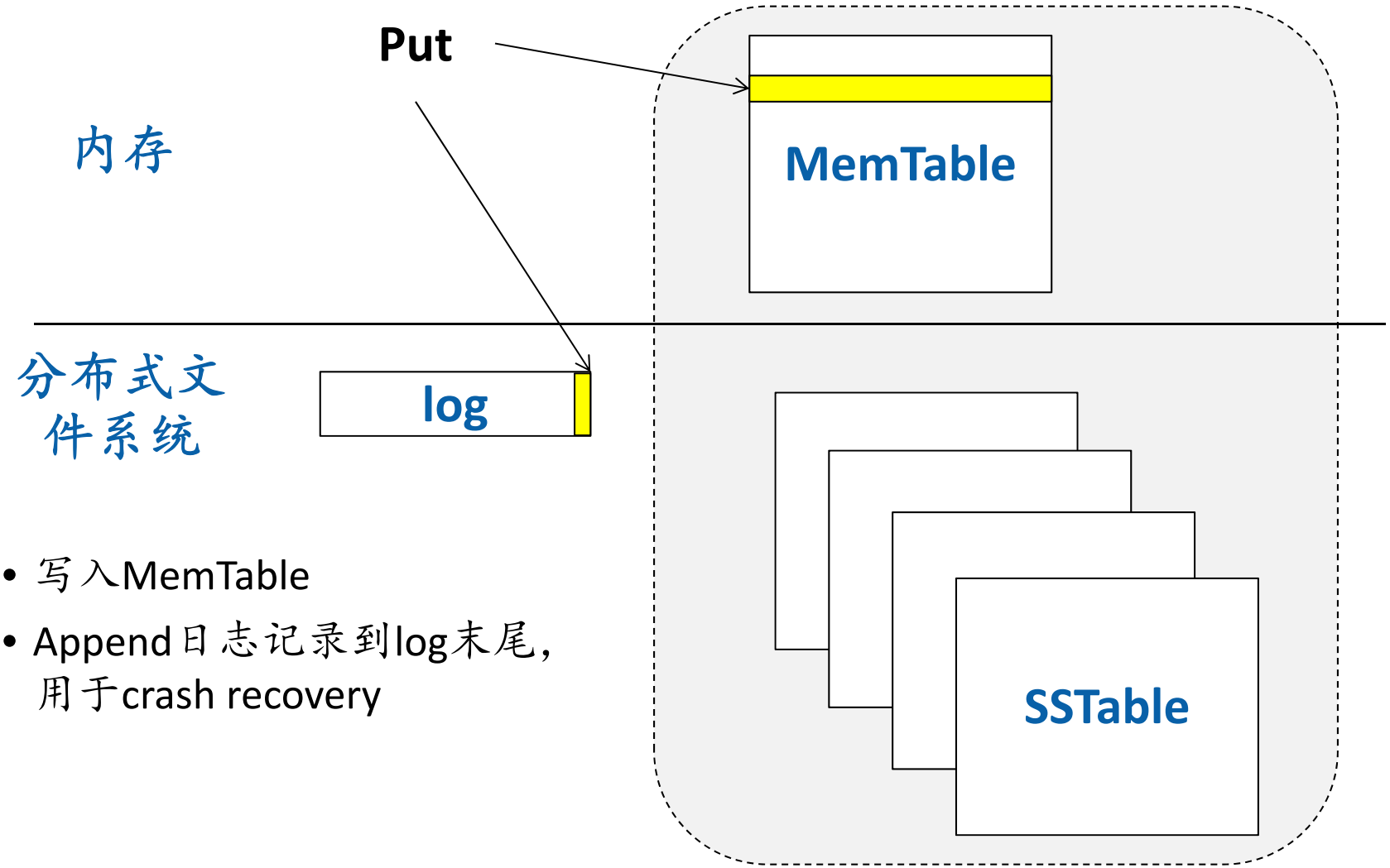


Tablet Server



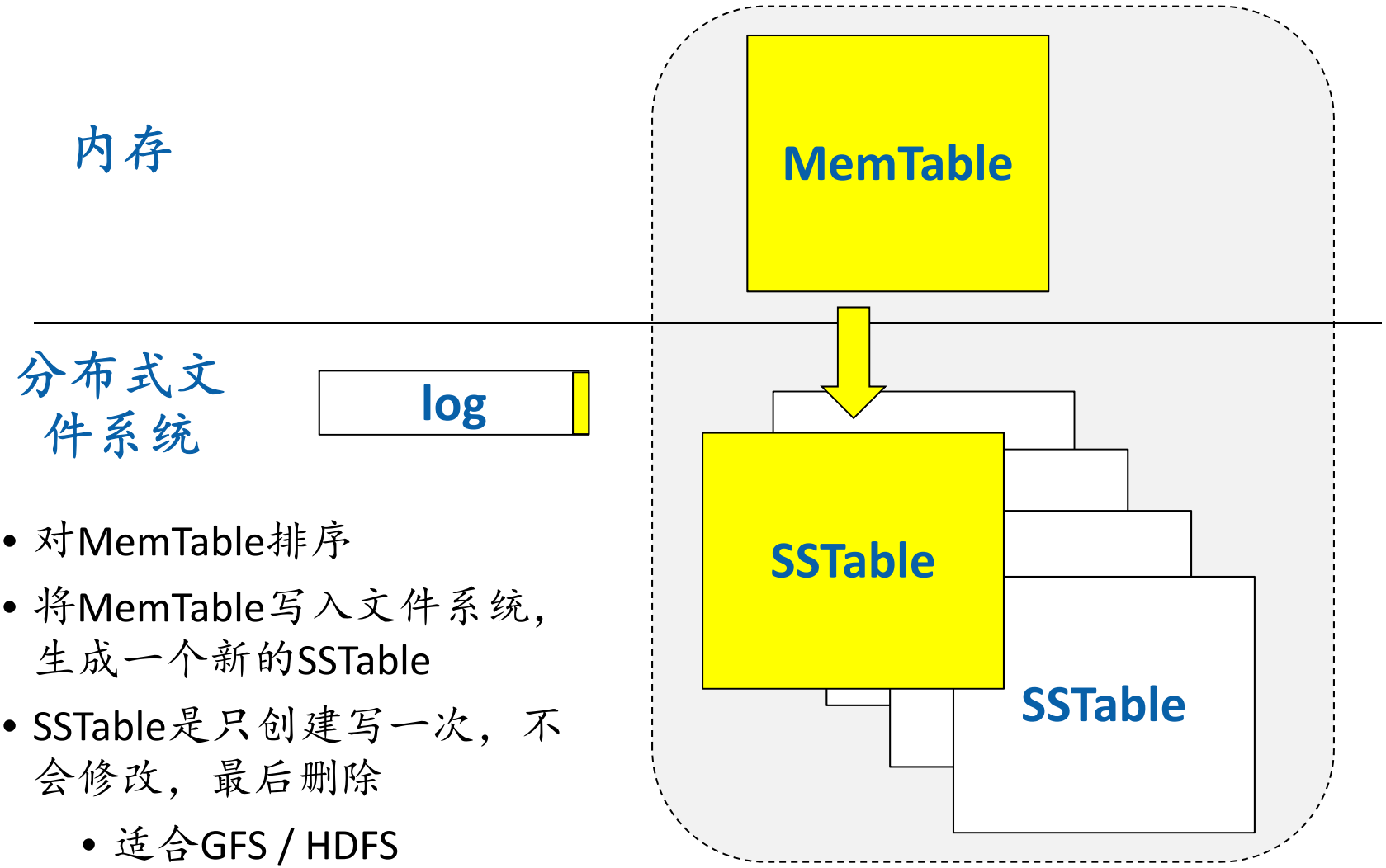
- 每个ColumnFamily分别存储
- 内存有一个MemTable (即C_0)
- 文件系统中存储多个SSTable
  - 采用tiering

Tablet Server： Put操作



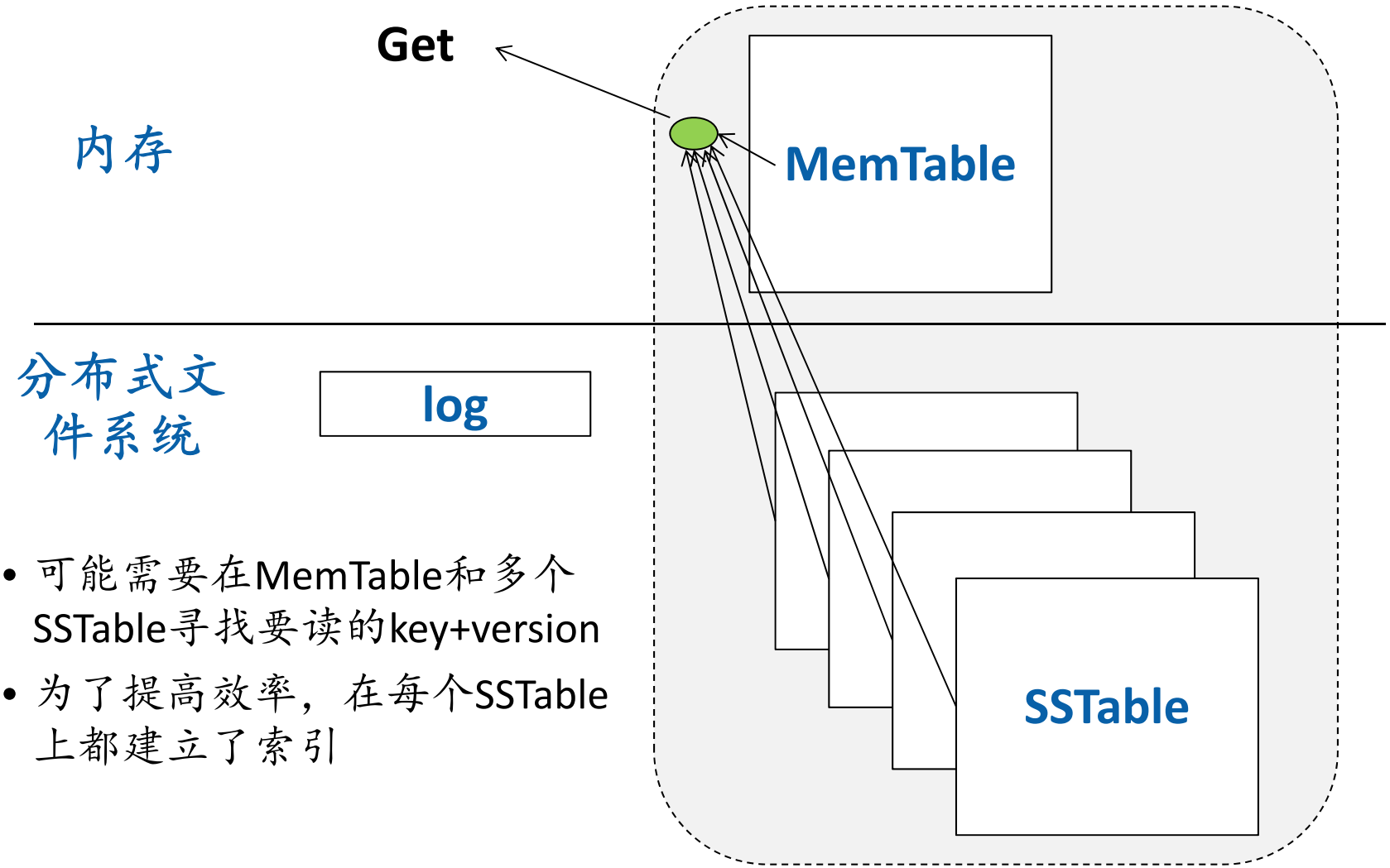
- 写入MemTable
- Append日志记录到log末尾，用于crash recovery

Tablet Server：当MemTable满了



- 对MemTable排序
- 将MemTable写入文件系统，生成一个新的SSTable
- SSTable是只创建写一次，不会修改，最后删除
  - 适合GFS / HDFS

Tablet Server： Get操作



- 可能需要在MemTable和多个SSTable寻找要读的key+version
- 为了提高效率，在每个SSTable上都建立了索引

Bigtable / Hbase小结

- Key包含了row key, column key的结构
- 除了Get/Put，还提供Scan(范围扫描操作)
  - 按照row key有序存储
- 底层存储采用了分布式文件系统
- Master与Tablet Server
- Tablet Server的内部结构
  - LSM-Tree
  - MemTable, SSTable, 和log

Key-Value Store: Cassandra

- Facebook为了Index Search功能研发了Cassandra
  - Cassandra的研发人员中有Dynamo文章的一位作者
- 之后，Facebook把Cassandra开源，2008年在google code上公布了源代码，2010年Cassandra成为了Apache 开源项目
- Cassandra是基于Java实现的

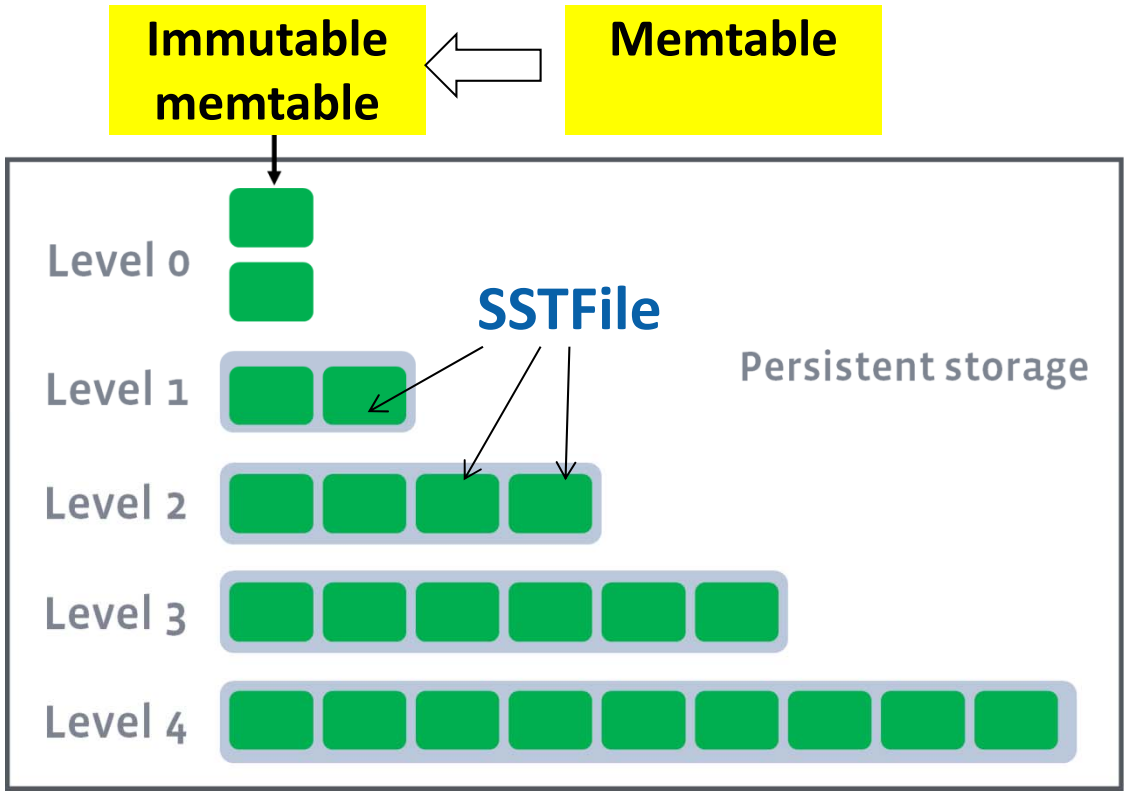
Cassandra与Dynamo和Bigtable

- Cassandra可以看作是Dynamo和Bigtable的结合体

	Dynamo	Bigtable	Cassandra
数据模型中的key	key	row key, column key	row key, column key, **super column key**
数据存储	Berkeley DB, MySQL	LSM-Tree 内存：MemTable 分布式文件系统： SSTable, Log	LSM-Tree 内存：MemTable 本地文件： SSTable, Log
备份冗余	Consistent hashing	分布式文件系统	Consistent hashing

RocksDB

- 2012年Fackbook基于Google LevelDB开发RocksDB
- C/C++实现，库而非单独系统，有序（类似BigTable），单机



https://github.com/facebook/rocksdb/wiki/Leveled-Compaction

- Memtable是C_0
- L0是MemTable直接排序写成的文件(相当于C_1)，采用tiering
- L1..Lk是标准LSM-tree
  - 采用leveling
  - 每个SSTFile对应一个key range

Outline

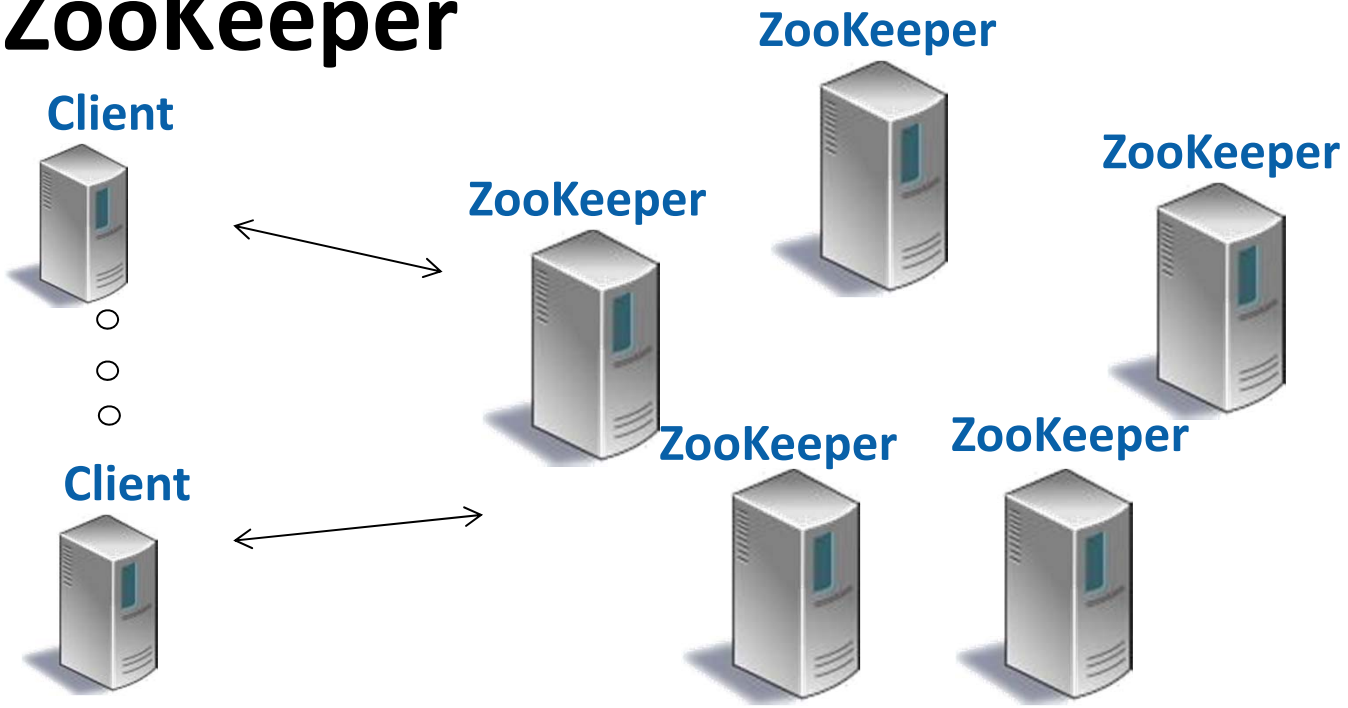
- Key-Value Store
- Distributed Coordination: ZooKeeper
  - 概念
  - 数据模型和API
  - 基本原理
  - 应用举例

Distributed Coordination

- 分布式系统中，多个节点协调
  - Leadership election: 选举一个代表负责节点
  - Group membership: 哪些节点还活着？发现崩溃等故障
  - Consensus: 对一个决策达成一致
  - …
- ZooKeeper
  - Yahoo! 研发的开源分布式协调系统
  - Hadoop/HBase环境的一部分
  - 目前广泛应用于分布式系统对于master节点的容错
    - 使用多台机器运行master节点，一台为主，其余为备份
    - 当主master出现故障，某台备份可以成为主master
  - 例如：HDFS, HBase, Hadoop…

"ZooKeeper: Wait-free Coordination for Internet-scale Systems". USENIX Annual Technical Conference 2010

ZooKeeper



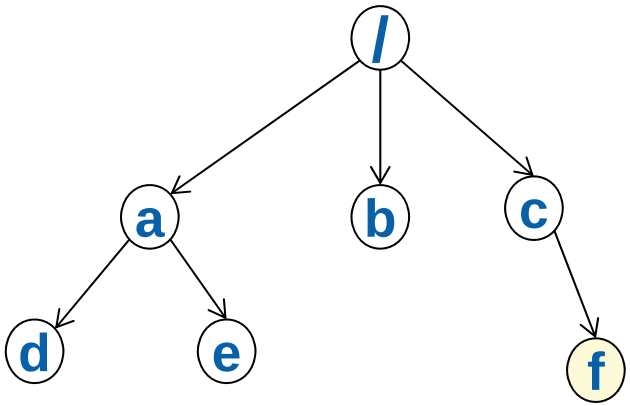
实际上，Client与ZooKeeper可以运行在同一台机器上

- 多个ZooKeeper维护一组共同的数据状态
  - 支持分布式的读和写操作
- 2f+1个ZooKeeper节点可以容忍f个节点故障，仍然正确
  - f=1：3个ZooKeeper节点可以容忍1个节点故障
  - f=2：5个ZooKeeper节点可以容忍2个节点故障
  - f=3：7个ZooKeeper节点可以容忍3个节点故障

ZooKeeper

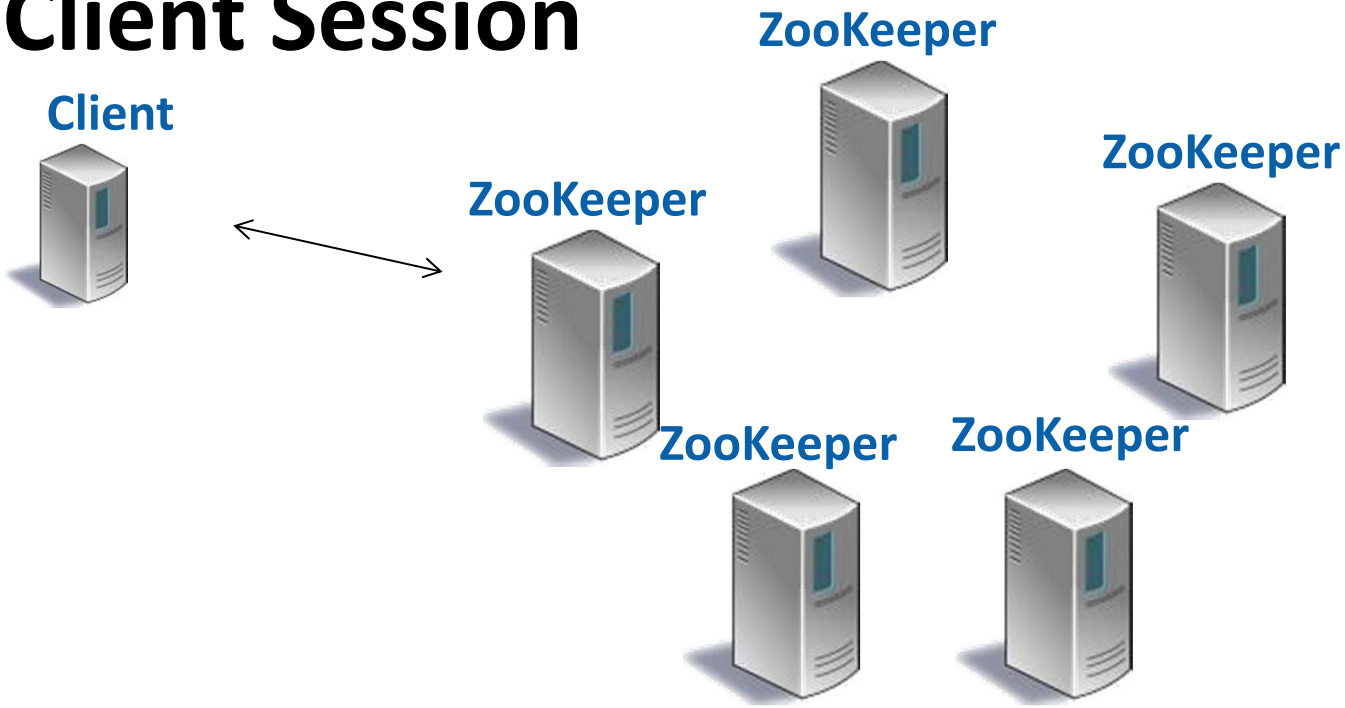
- 数据模型是什么？
- 如何操作？
- 内部是如何实现的？
- 可以用来支持哪些功能？

ZooKeeper的数据模型： Data Tree



- ZooKeeper维护一组共同的数据状态
  - 表达为一棵树，实际上是一个简化的文件系统
- 树的每个顶点称为Znode，有下列属性
  - Name: 一个Znode可以用一条从根开始的路径唯一确定
    - 类比文件路径，例如：/c/f
  - Data: 可以存储任意数据，但长度不超过1MB
  - Version：版本号
  - Regular/Ephemeral：正常的/临时的
    - 对于Ephemeral的Znode，系统将在Client session结束后自动删除

Client Session

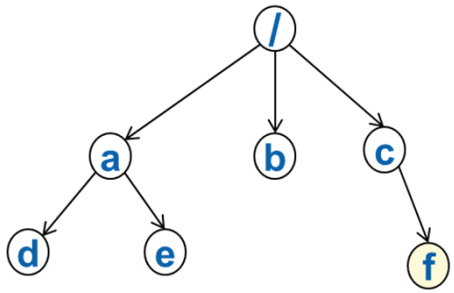


- Session怎么开始？
  - 一个Client连接到ZooKeeper，就开始一个Session(对话)
- Session怎么结束？
  - Client主动关闭
  - 经过一个Timeout时间，ZooKeeper没有收到Client的任何通信
    - 比如Client出现failure了

Client API

- 创建Znode/删除Znode/判断Znode存在
  - create(path, data, flags)
  - delete(path, version)
  - exists(path, watch)
- 读Znode数据/修改Znode数据
  - getData(path, watch)
  - setData(path, data, version)
- 找孩子Znode
  - getChildren(path, watch)
- 等待前面操作完成
  - sync()

Watch机制



- 判断Znode存在
  - exists(path, watch)
    - 返回True/False
    - 可以设置一个watch，当Znode被删除/新建时，收到通知
- ZooKeeper通知
  - 当对应的数据发生改变时，通知Client
- 通知之后，Watch就被删除了
  - 如果需要继续关注，那么需要再次注册watch

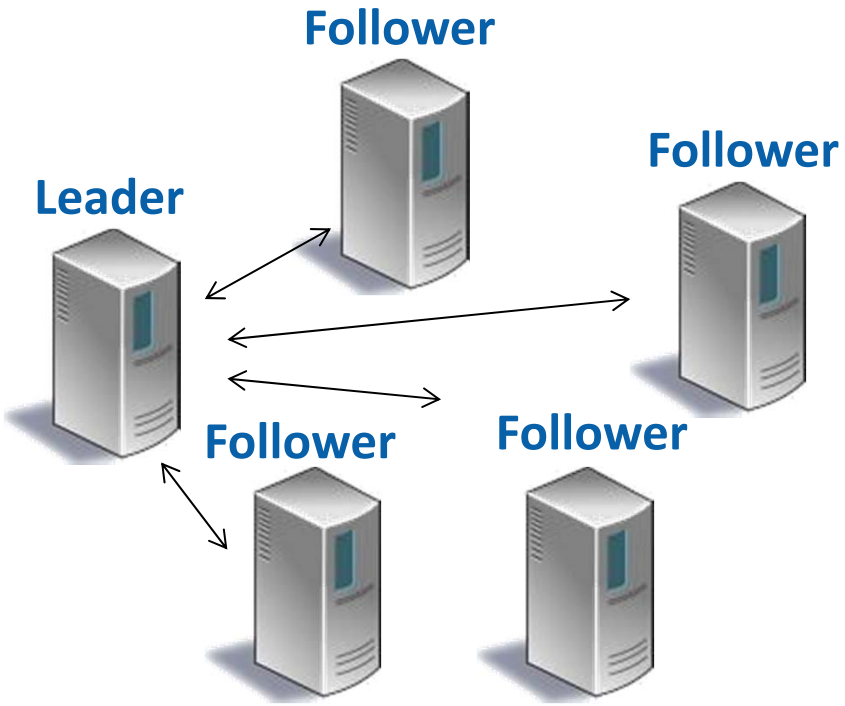
同步和异步方式

- Synchronous：同步
  - Client发一个请求，阻塞等待响应；
    再发下个请求，再阻塞等下个响应
  - 当请求个数很多时，同步操作就很慢

- Asynchronous：异步
  - 允许Client发送多个请求，不需要阻塞等待请求完成
  - 提供Callback函数，当请求完成时，Callback被调用

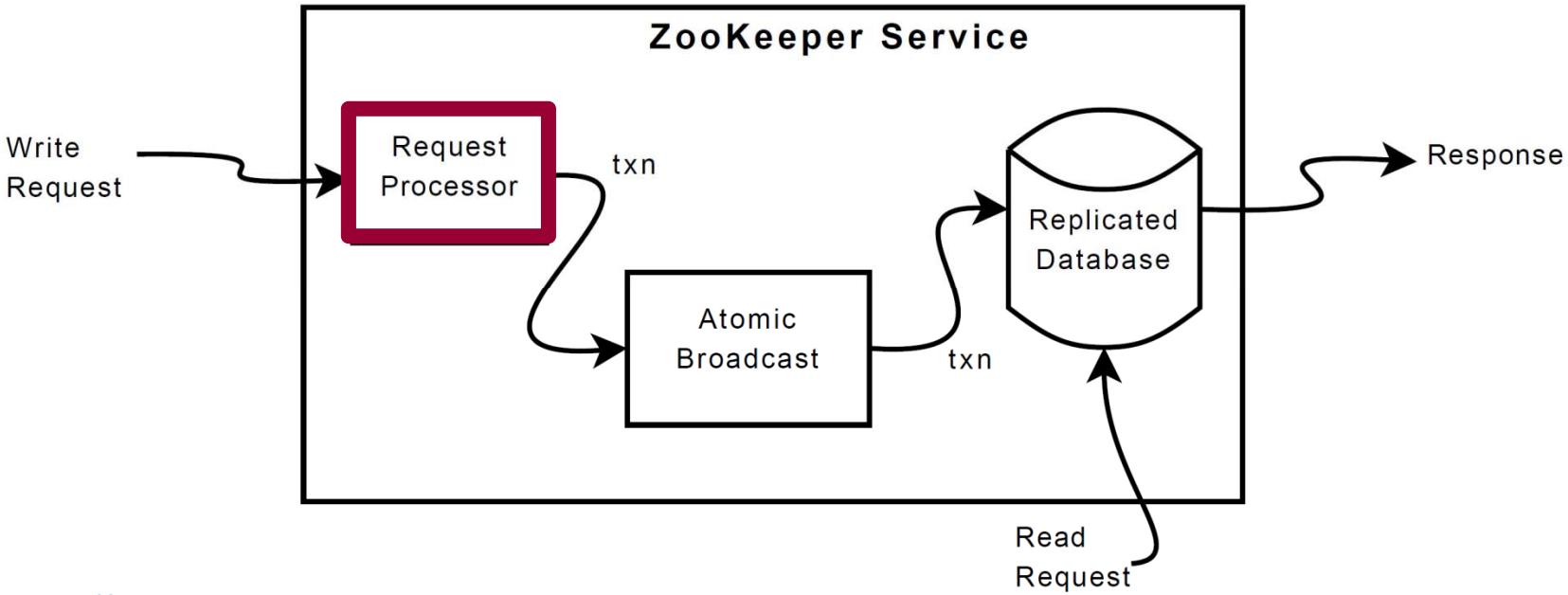
- 前面的API都提供同步和异步两种实现

ZooKeeper系统结构



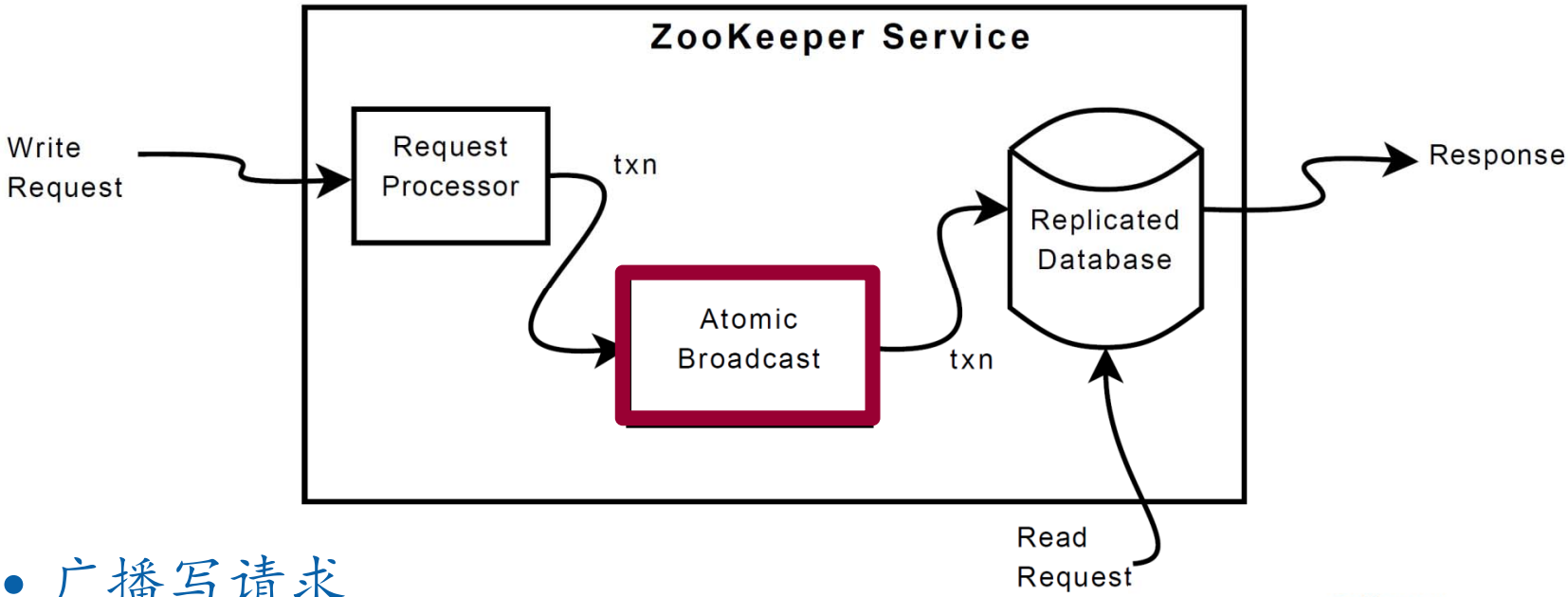
- 所有节点都在内存中维持相同的ZooKeeper树
  - 外存有snapshot+log，来提供crash recovery
- 一个Leader，多个Follower
- 每个Client只连接到一台ZooKeeper服务器
  - 所有的读操作都由这台服务器用其本地的状态来回复

写请求的处理（1）Request Processor

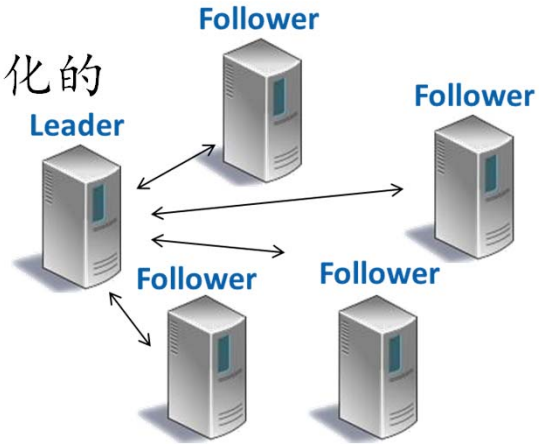


- Follower
  - 对于写请求，将发给Leader统一处理
- Leader
  - 把写请求包装成为一个Idempotent Transaction（包括分配新的Version等），这样每个Txn可以执行多次来恢复（概念与NFS相似）
  - Txn有递增的唯一的ID

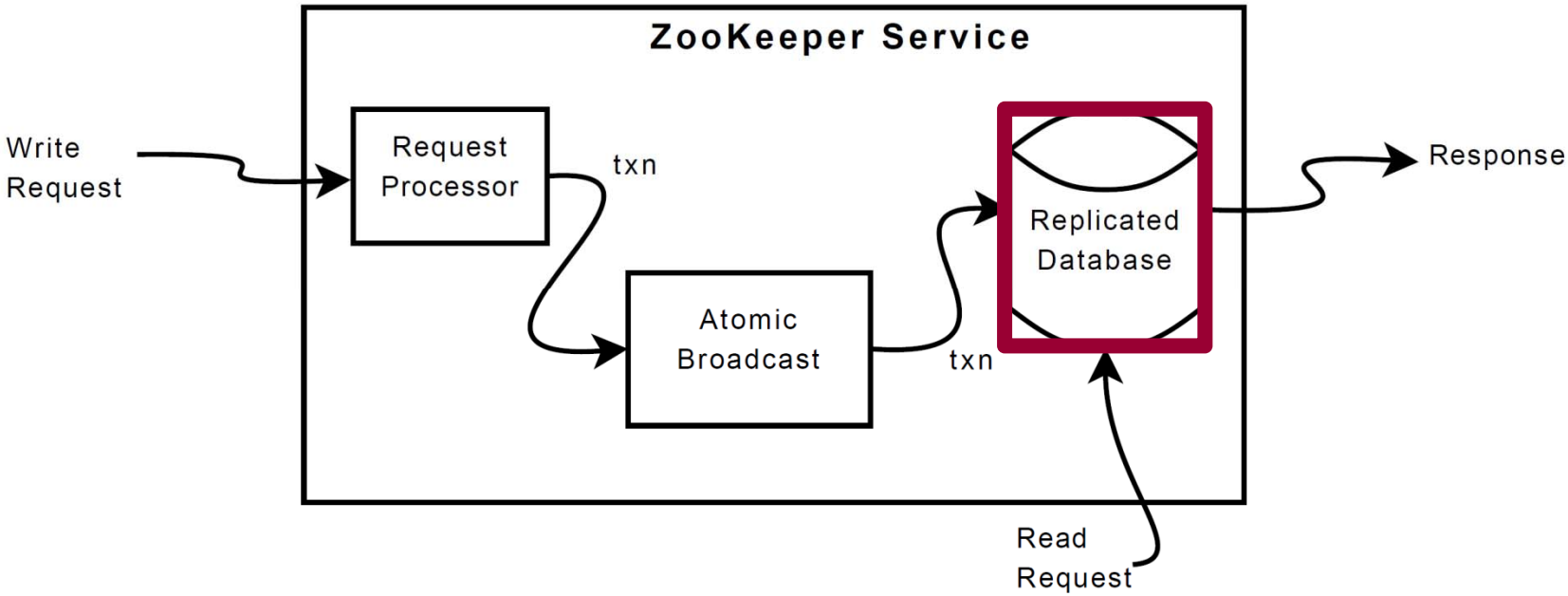
写请求的处理（2）Atomic Broadcast



- 广播写请求
  - Leader带领Follower，保证写操作在全局是串行化的
  - 使用的ZAB协议是2PC的变形

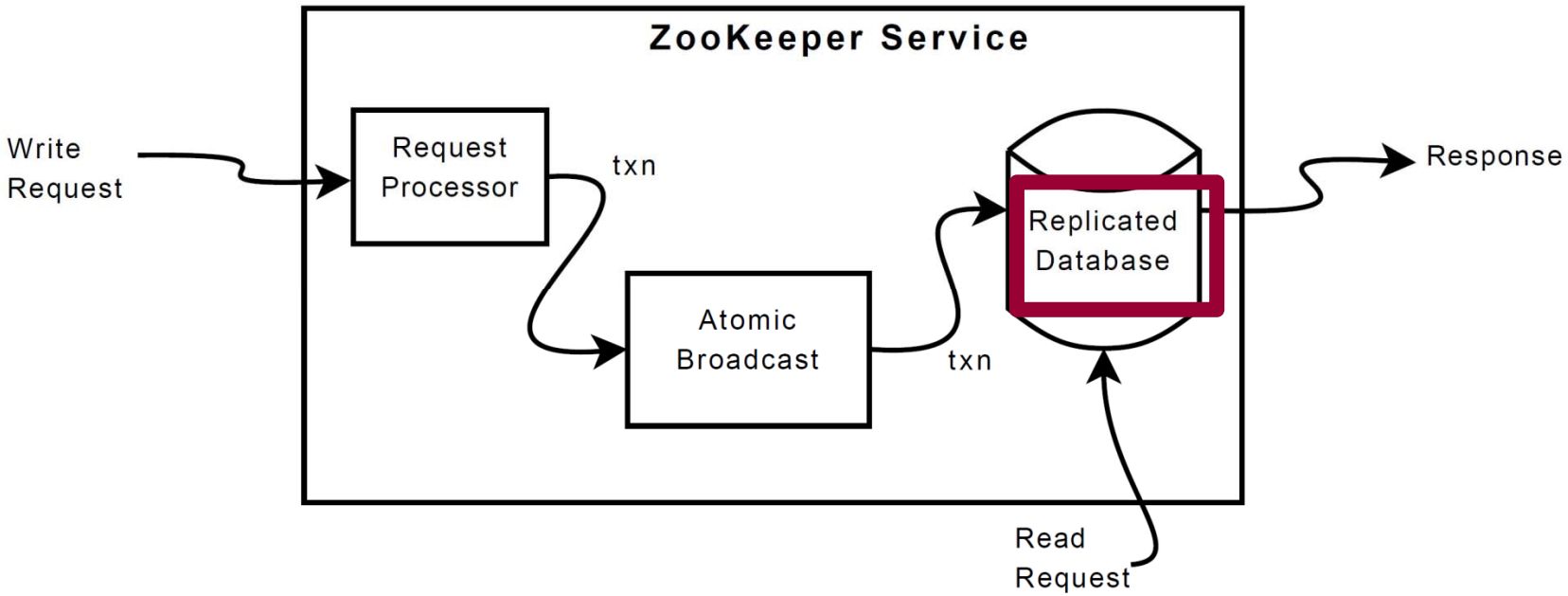


写请求的处理（3）Replicated DB



- 所有节点的Replicated database: ZooKeeper内存的树
  - 在Atomic Broadcast后，写操作修改本地的Replicated database

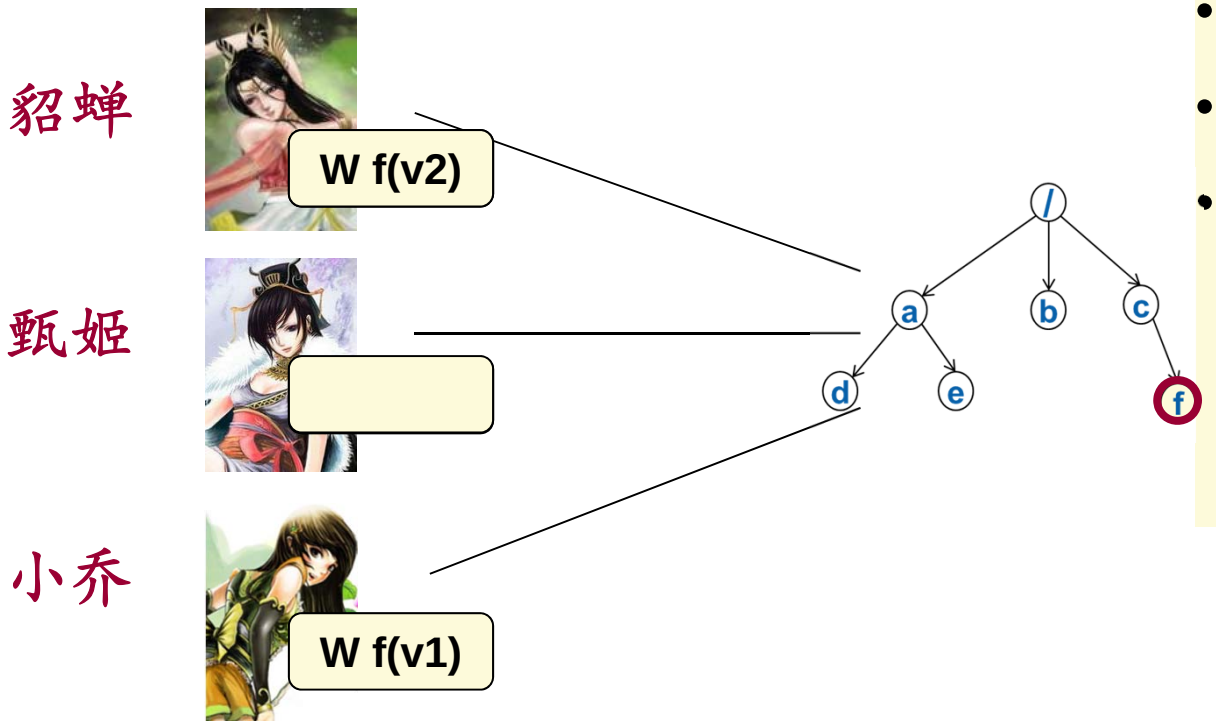
读请求的处理



- 每个节点都可以处理读请求，因为所有的Replicated DB都是一致的
- 读请求将直接由节点本地的replicated database回答
  - 写的全局顺序有Leader决定
  - 但读是分布的，如果写还没有广播完成，读可能看到旧数据

ZooKeeper保证： Linearizable writes

- 写操作串行化
- 因为写操作是Leader带领Follower按照相同顺序完成的



- Client小乔写f(v1)
- Client貂蝉写f(v2)
- 所有client都只按照这个顺序看到的写
  - 如果读到了f(v2)那么下面的读不会看到f(v1)

ZooKeeper保证： FIFO client order

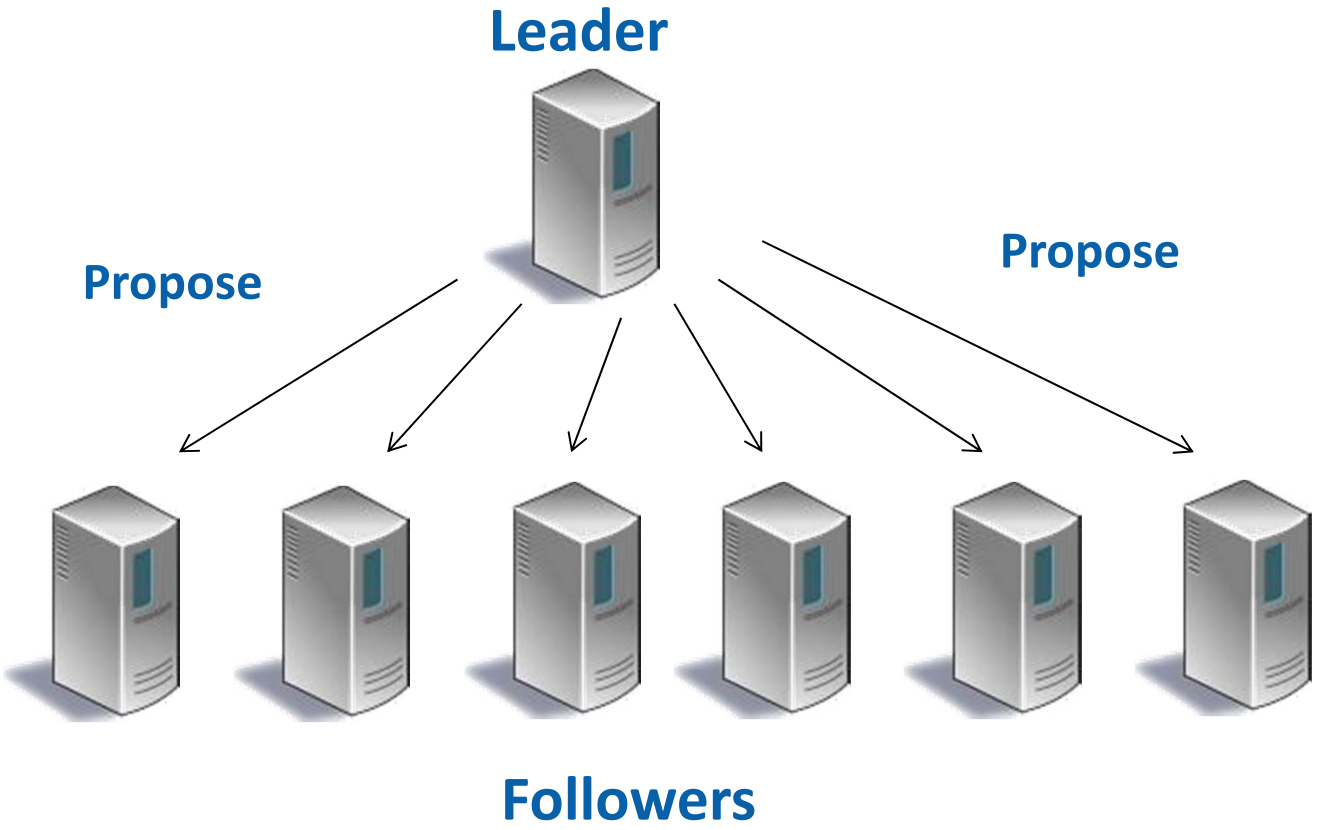
- 每个Client自己的读写操作是按照FIFO的顺序发生的

- 但是，不同Client之间的读写顺序没有任何保证
  - 一个Client读的可能是另一个Client的写前或写后的数据
  - 因为可能访问到了follower更新前或更新后的数据
  - 如果一定要读最新数据，那么调用sync

ZAB

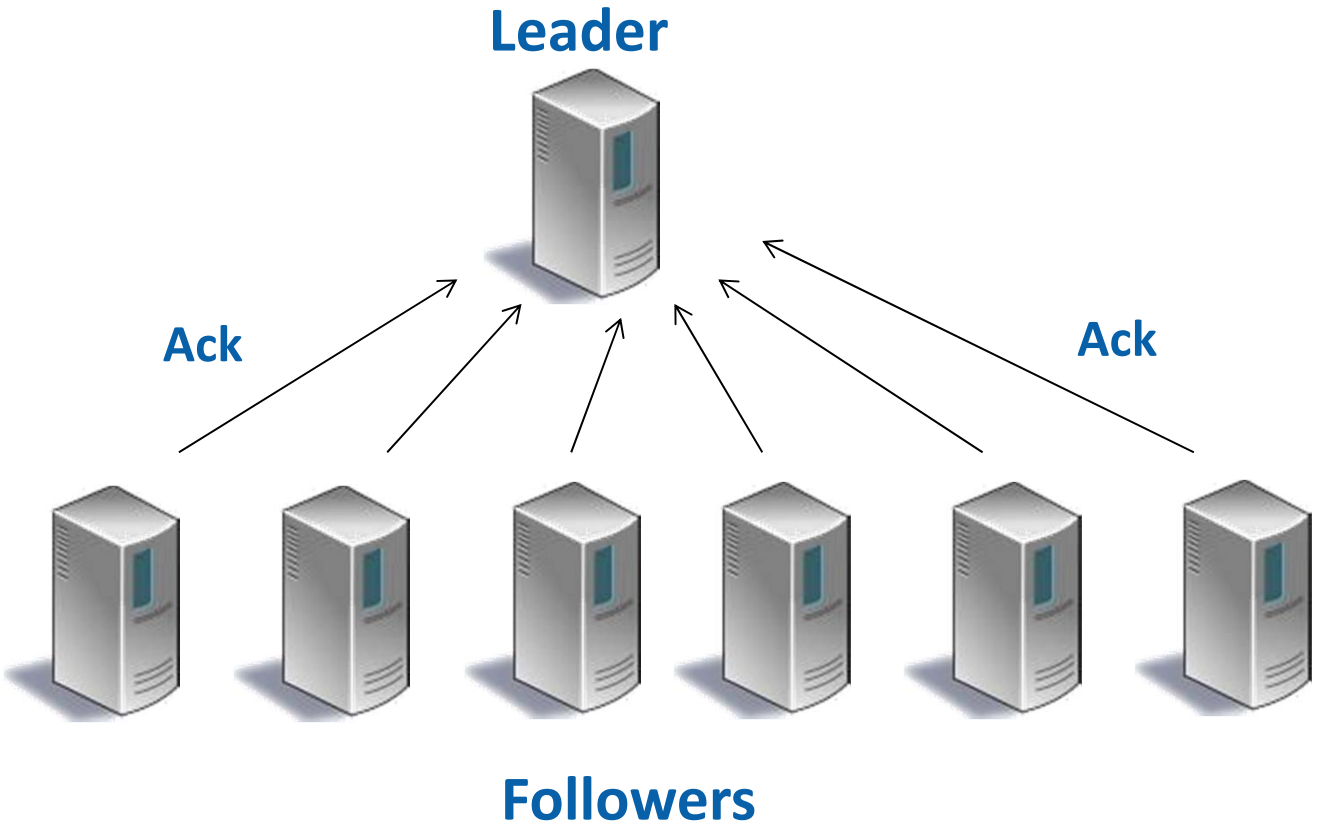
- 两个主要工作模式
  - 正常Broadcast
    - Leader向Follower广播新的写操作
  - 异常Recovery
    - 竞争新的Leader
    - 新的Leader进行恢复

ZAB Broadcast： phase 1 (propose)



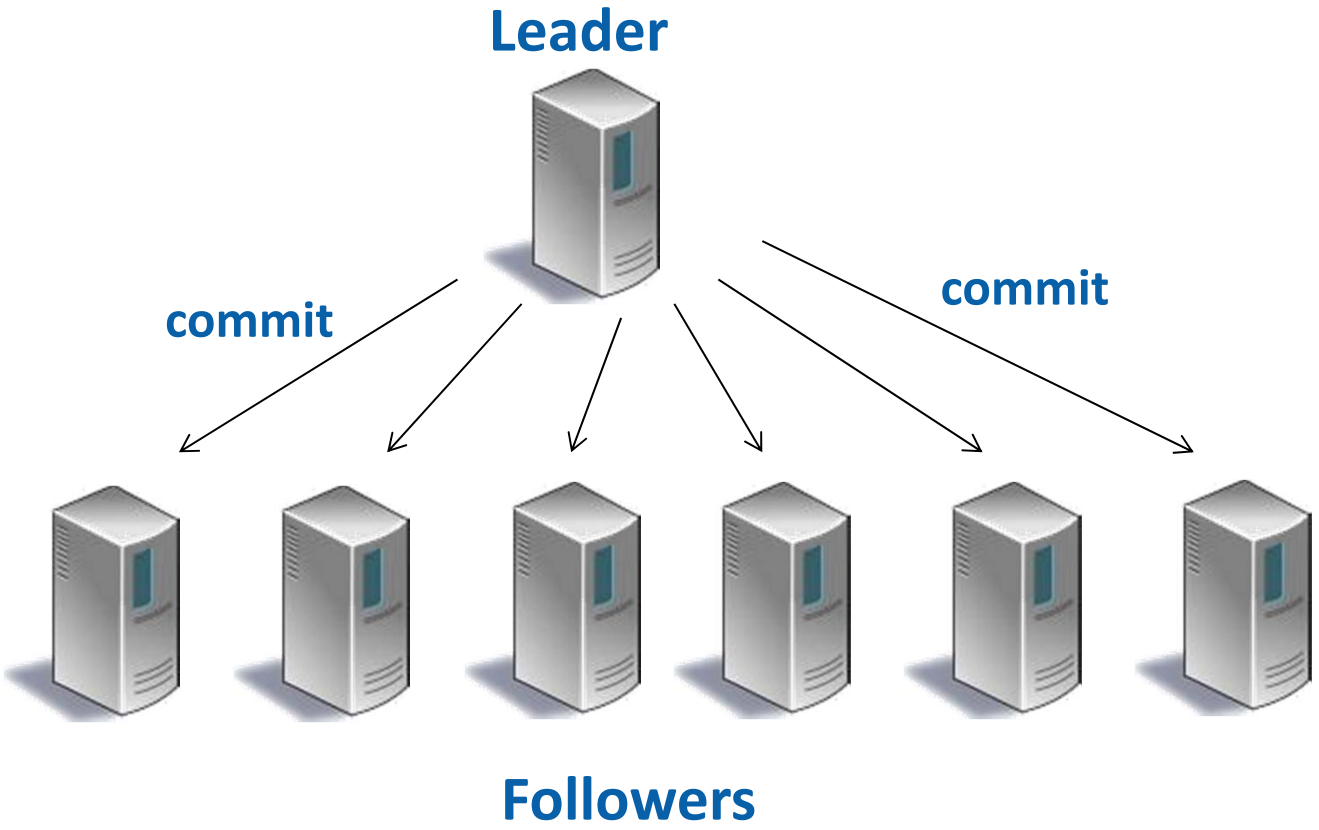
- Leader把一个新的txn写入本地log， 广播Propose这个txn

ZAB Broadcast： phase 1 (propose)



- 每个Follower收到Propose后，写入本地log，向Leader发回Ack
- 一定可以commit（因为进行相同的写），所以不需要yes/no，只要Ack

ZAB Broadcast： phase 2 (commit)



- Leader收到 f个Ack后(在所有2f+1个节点中，共有f个followers和自己=f+1节点记住了这个txn)，写Commit到log, 广播Commit，修改ZooKeeper树
- Follower收到Commit消息，写Commit到log，然后修改ZooKeeper树

ZAB Broadcast

- 2PC的简化
  - 原因：通知新的Transaction发生，所有节点的写操作是一样的
  - Propose阶段
    - Leader把一个新的txn写入本地log，广播Propose这个txn
    - 每个Follower收到Propose后，写入本地log，向Leader发回Ack
  - Commit阶段
    - Leader收到 f个Ack后，写Commit到log, 广播Commit，然后修改自己的ZooKeeper树
    - Follower收到Commit消息，写Commit到log，然后修改ZooKeeper树
- 注意
  - 可以异步发送多个Propose，从而可以批量写入log
  - Commit阶段不需要Ack
  - 如果Leader未收到f个Ack(timeout了)或Follower长时间未收到Leader的消息，那么就发现了故障，需要进入Recovery

ZAB Recovery

- 竞选Leader
  - 每个节点察看自己看到的最大Txn ID
  - 选择Leader为看到max(TxnID)为最大的节点
  - 可以最大限度地保护Client写操作
- TxnID共64位：高32位代表epoch，低32位为in-epoch id
  - 每次选Leader, epoch ++
  - 在一个Leader内部，新的txn增计低32位
  - 于是，每次Recovery后，一定使用了更高的txn id
- 新的Leader
  - 把所有正确执行的Txn都确保正确执行（idempotent，再广播一次）
  - 其它已经提交但是还没有执行的Client操作，都丢弃
  - Client会重试

应用举例（1）

- Configuration Management
  - 把配置信息存储在一个确定路径的Znode中
    - 例如：/app1/conf
  - 利用Watch机制获得配置信息的变化
    - getData(“/app1/conf”, watch)
    - 获得当前配置信息
    - 当配置信息更新时，会收到watch的通知

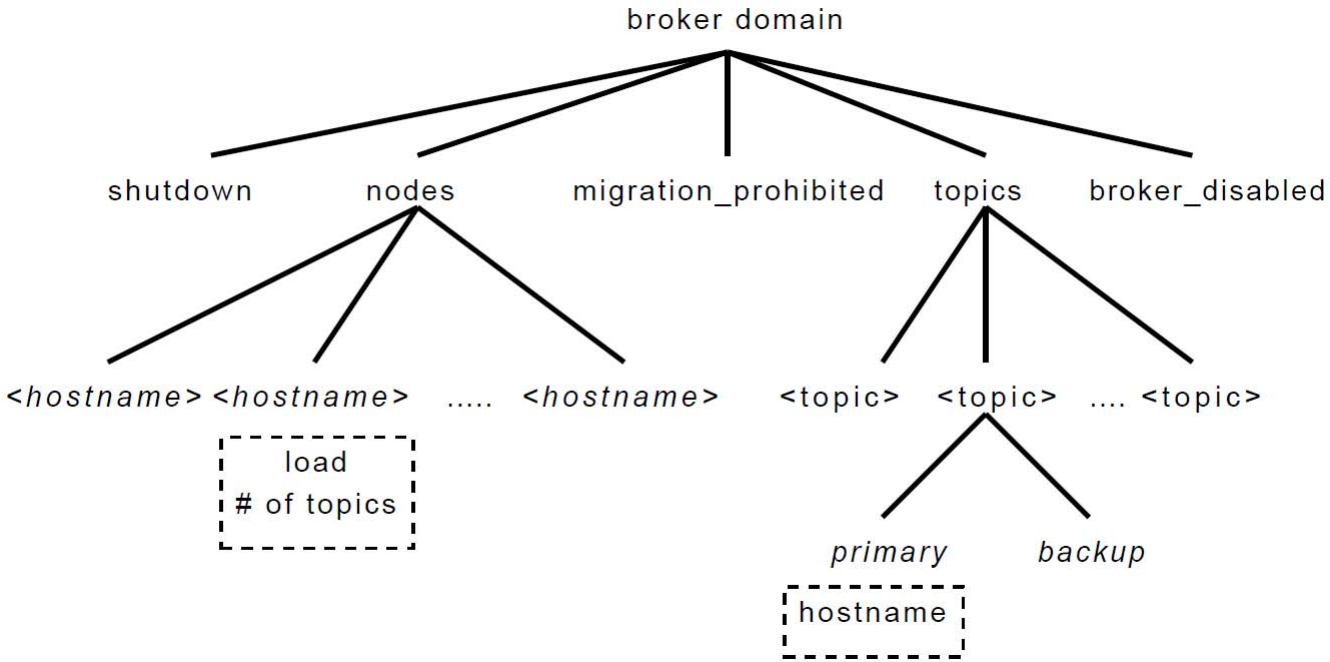
应用举例（2）

- Group Membership
  - 用一个Znode代表节点组
    - 例如：/app1/group
  - 每个成员都在group下面创建一个ephemeral 的孩子
    - 例如：/app1/group/machine1, /app1/group/machine2, …
  - 当某个成员崩溃了，那么它对应的孩子就被删除
    - 在一定Timeout时间后，相应的Client session被终止，这个ephemeral节点被删除
  - 从而可以读group的孩子来确定组的成员
    - Group下的节点与仍然工作的成员一一对应

应用举例（3）

- Simple Lock
  - 可以把lock对应为一个Znode
  - 加锁=创建Znode，解锁=删除Znode
  - 加锁不成功，可以用watch，当Znode被删除时，可以得到通知

应用举例：Yahoo Message Broker



- Publish/subscribe系统
- 看一下
  - Group membership：/broker domain/nodes
  - 配置：topic在哪台机器上：/broker domain/topics
  - 重要事件：/broker domain/shutdown, /broker domain/migration_prohibited等

Distributed Coordination

- 我们今天介绍了Zookeeper
- 经典的协议有Paxos和Raft
  - 达成类似功能：使多个参与者看到一致的操作
  - Paxos：参与者是平等的
  - Raft：有leader

小结

- Key-Value Store
  - Dynamo
  - Bigtable / Hbase
  - Cassandra
  - RocksDB
- Distributed Coordination: ZooKeeper
  - 概念
  - 数据模型和API
  - 基本原理
  - 应用举例