

分布式计算

09 - Cloud Storage

Weixiong Rao 饶卫雄

Tongji University 同济大学软件学院

2023 秋季

wxrao@tongji.edu.cn

目录

- 传统方法
- Key-value 存储 (KVS)
 - ◆ 基本概念; 操作
 - ◆ KVS举例
 - ◆ KVS 及并发
 - ◆ Key-multi-value 存储; 光标
- 云平台的Key-value存储
 - ◆ 难点
 - ◆ 专用的KVS存储
- 3个具体实现
 - ◆ Amazon's Dynamo
 - ◆ Google's BigTable
 - ◆ Facebook's Cassandra



传统方法

■ 什么是数据库？

- ◆ “...结构化和有组织的数据仓储” (Abramova & Bernardino, 2013-07)
- ◆ 可以通过数据库管理系统DBMS (DataBase Management System)访问数据

■ 什么是数据库关系系统？

- ◆ “DBMS 定义为数据存储在、编辑、抽取的使能方法集合” (Abramova & Bernardino, 2013-07)

■ SQL: Structured Query Language

◆ 标准语言

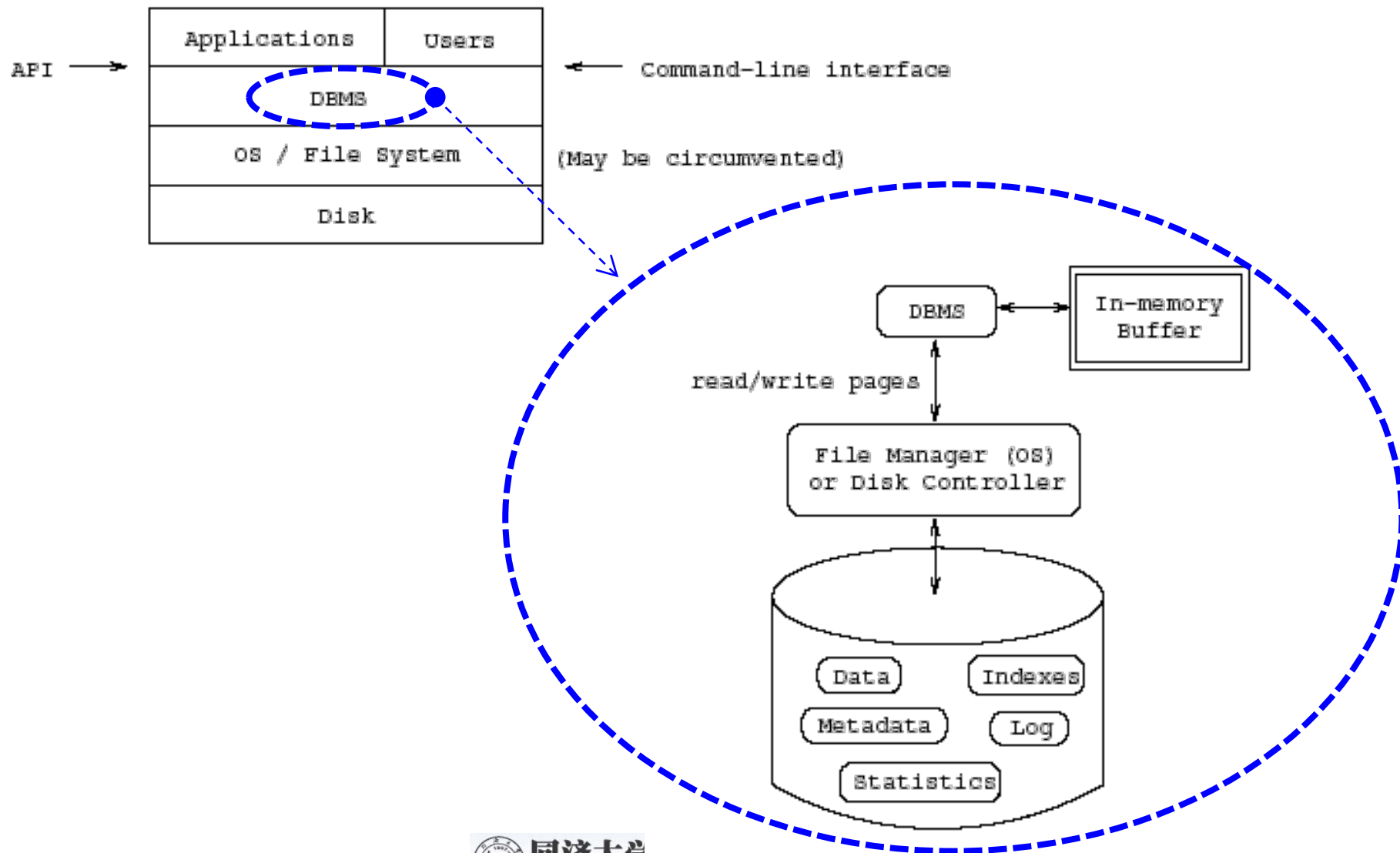
- 数据交互

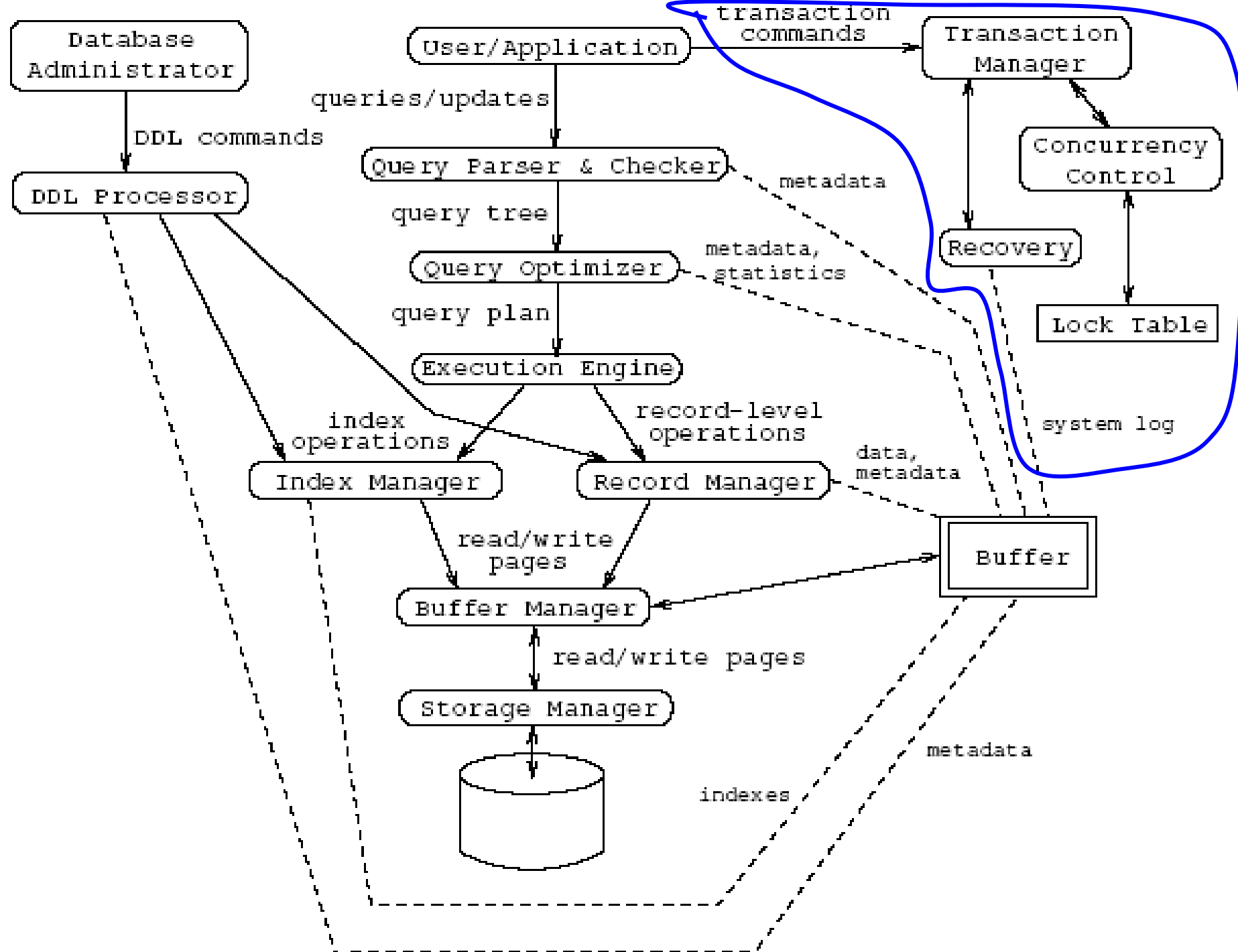
- 数据操纵

◆ 数据以表**tables**的形式存储

◆ 可同时访问多个表中的数据

DBMS的位置





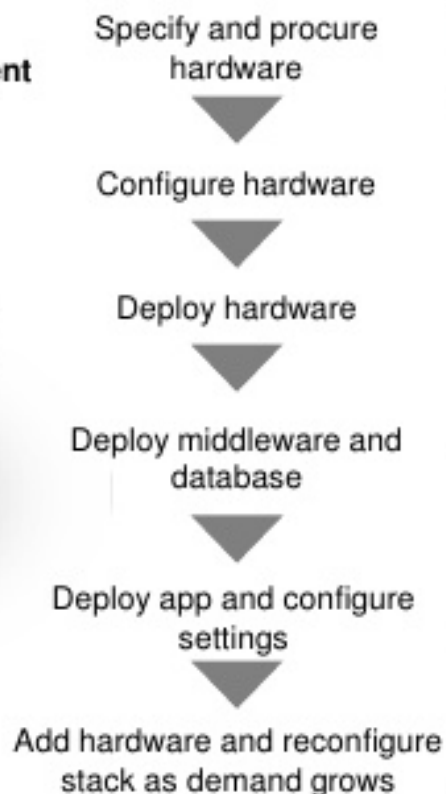
传统方法...

- 云数据服务(Cloud Data Services)相当多方法是在关系数据库系统之上进行延伸
 - ◆ Oracle, Microsoft SQL Server 甚至 MySQL 传统上就可以用于企业计算和在线数据云平台
 - ◆ 集群数据库- 传统的企业级就可以提供多服务器的集群计算和数据复制能力 – 提供可靠性
 - ◆ 高可用性 – 提供同步能力 (“Always Consistent”), 负载均衡和高可用, 以提供接近 100% 在线服务能力
 - ◆ 复杂的数据模型和结构化查询 – 能够把数据处理迁移到后台的数据库服务器

Oracle方案

Evolution of IT Operations

Traditional App Deployment (Admin driven)



DEPLOYMENT PORTAL

Request Deployment
via Cloud

Adjust capacity as
demand changes

Retire app when
not needed

User unaware of
underlying infra

Platform-as-a-Service Deployment (End-user driven)



Self-Service Provisioning

传统方法...

- 可是, 传统的关系数据云:

EXPENSIVE! 贵! 贵! 贵!

需要进行系统维护、许可证、存储大量的数据

- 传统企业关系数据库如Oracle所达到的服务保证, 需要云端较高的开销成本
- 复杂的数据模型需要更加昂贵的云端成本: 包括数据的维护、更新和同步
- 负载平衡通常需要昂贵的网络设备
- 云平台的高回弹性, 亦需要开销较高的网络升级

解决方法

- 降低传统的RDBMS的服务保证
 - ◆ 将Oracle和SQL Server的复杂数据模型替换为一个简单的模型
 - ◆ 将“Always Consistent”模型替换为 “Eventually Consistent” – 涉及到数据的更新策略
- 根据更为简单的数据模型和降低一致性的设计原则，重新设计与传统RDBMS不同的方法

RDBMS优点

- 传统上看RDBMS的优点:
 - ◆ 表达数据关系
 - ◆ 关系模型/SQL易于理解
 - ◆ 以磁盘为中心的数据存储
 - ◆ 索引结构
 - ◆ 数据的一致性 (上锁机制)

不足

- 数据模型过于严苛
 - ◆ 自由点，简单点？
- 吞吐量有限
 - ◆ 希望更高的吞吐量
- 向上扩展成本难以接受 (昂贵的服务器)
 - ◆ 向外横向扩展 (低成本服务器)
- 从数据本身到关系模型映射所需开销
 - ◆ 我们希望数据长成什么样子就保存为什么样子
- 如何有效进行数据的切分
 - ◆ 我们希望自由的进行数据切分
- 数据库供应商对云服务总体看来身段迟钝缓慢
 - ◆ 我们每个人都需要使用云服务



今天的数据是怎样的...

- 万事皆变
- 数据非得符合关系模型?
- 不同的信息数据, 约束未必相同
- 举例:
 - ◆ 购物篮添加商品
 - ◆ Wikipedia 搜寻答案
 - ◆ Web页面信息检索
 - ◆ 数据量巨大!!!



SQL并不适合于:

- Text
- 数据分析
- 流时数据处理Stream processing
- 科学和智能数据库
- ...

今天的数据

- 数据类型:
 - ◆ 结构化, 半结构化, 非结构化
- 数据库中的结构化信息
 - ◆ Data → chunks
 - 本质上是把类似的实体**entities**进行分组
 - ◆ 同分组的实体数据描述 – 数据格式、长度等均一致

今天的数据

- 半结构化的数据包含一些结构信息，但是未必所有的数据格式都一样
 - ◆ 相似实体的分组 – 可能包含不同的属性
 - ◆ Schema信息可能和数据属性值混合在一起(如：XML)
 - ◆ 亦可能展示为一张图

今天的数据

■ 非结构化数据

- ◆ 数据可能是任何类型,无特定的结构或序列
- ◆ 难以表达为一种特定的schema结构
 - HTML书写的Web页面
 - Video, sound, images
- ◆ Big data – 多数是非结构化、其中有些是半结构化

Big Data - 什么是大数据?

- 增量速度非常快:
 - ◆ 智能收集广播位置信息 (few secs)
 - ◆ 汽车芯片记录诊断测试数据 (1000s per sec)
 - ◆ 相机记录公开/私人空间的图片和视频数据
 - ◆ 物流RFID 标签扫描读数据

大数据的特点

- 非结构化
- 异构
- 增长速度
- 多样性
- 无法进行形式化建模
- 数据价值 (仅仅是因为数据量大, 就重要吗?)
- 标准数据库和数据仓库难以涵括多样性异构性
- 难以取得满意的性能

解决方法

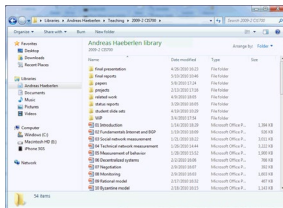
- **Amazon's Dynamo** – 用于Amazon's EC2 Cloud Hosting Service. 使能高回弹存储服务S2平台及其电子商务平台
提供一个简单的Primary-key的数据模型. 在分布式的低成本节点上存储大量数据
- **Google's BigTable** – Google的主要数据云平台, 使用较Amazon's Dynamo更为复杂的 column-family的数据模型, 不过比传统的 RMDBS更为简单
Google'底层数据服务系统, 在低成本节点上提供分布式的数据服务系统
- **Facebook's Cassandra** – Facebook的主要数据云服务平台
近期作为开源项目, 提供与Google's BigTable 类型的数据模型, 并采用Amazon's Dynamo 的分布式特性

目录

- 传统方法
- Key-value 存储 (KVS)
 - ◆ 基本概念; 操作
 - ◆ KVS举例
 - ◆ KVS 及并发
 - ◆ Key-multi-value 存储; 光标
- 云平台的Key-value存储
 - ◆ 难点
 - ◆ 专用的KVS存储
- 3个具体实现
 - ◆ Amazon's Dynamo
 - ◆ Google's BigTable
 - ◆ Facebook's Cassandra



云服务应用越来越复杂, 存储设施越来越简单



文件大小不一

- 读、写、追加
- 移动, 重命名
- 加锁, 解锁
- ...

操作系统

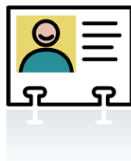
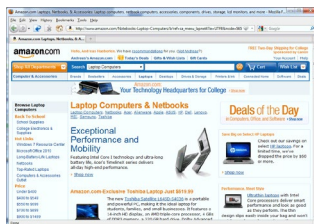


数据库大小一致

- 读
- 写

- PC用户习惯了功能丰富的界面
 - ◆ 树状的名字空间 (目录); 可以方便的进行文件移动、重命名、追加、压缩、解压、查看、删除 ...
- 但是实际的存储设备还是非常简单
 - ◆ HDD 只需知道如何读、写固定小的数据块
- 交易处理由操作系统来完成

与云存储相类似



购物车
好友列表
用户账户
个人画像

...


Web 服务

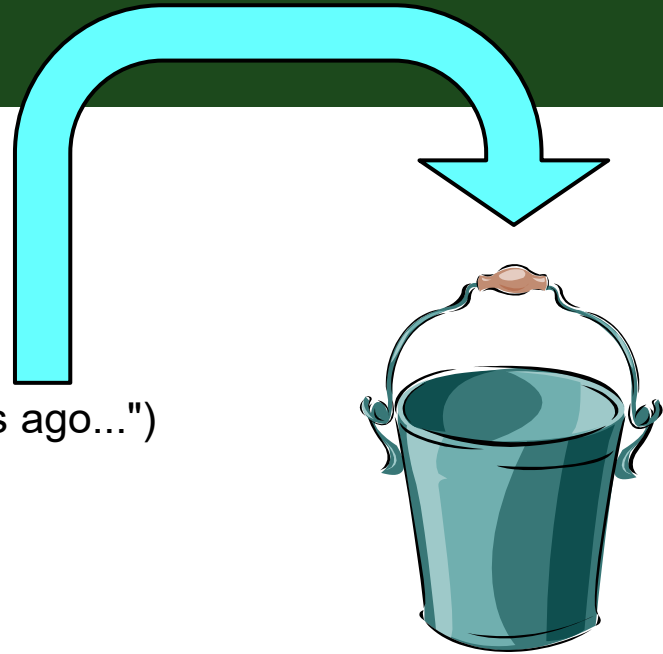


Key/value 存储
- read, write
- delete

- 很多云服务均由相类似的结构
 - ◆ 用户使用功能丰富的接口 (购物车, 商品列表, 可搜索索引 推荐功能, ...)
- 但是实际的存储服务非常简单
 - ◆ 数据库读写, 类似于一个巨型的硬盘
- 交易处理由Web服务来完成

Key-value 存储

Keys Values
↓ ↓
(bob, bschmitt@foo.com)
(gettysburg, "Four score and seven years ago...")
(29ck2dxa1, 0128ckso1\$9#*!!8349e)
(windows, )



- **key-value store (KVS)**是管理可持久化状态的简单抽象
 - ◆ 数据已 (key,value) (键, 值)对组成
 - ◆ 只有3个基本运算:
 - PUT(key, value)
 - GET(key) → value
 - Delete(key)

KVS举例

- 见过这个概念吗？
- 经典的例子：
 - ◆ 主内存的哈希表结构
 - ◆ 磁盘文件索引结构 (如BerkeleyDB)
 - ◆ “倒排表索引” – 搜索引擎
 - ◆ 数据库管理系统 – 多个KVSs
 - ◆ 分布式哈希表 (Chord/Pastry)

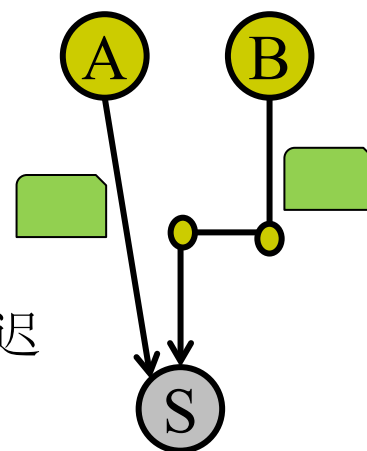
目录

- 传统方法
- Key-value 存储 (KVS)
 - ◆ 基本概念; 操作
 - ◆ KVS 举例
 - ◆ KVS 及并发
 - ◆ Key-multi-value 存储; 光标
- 云平台的Key-value存储
 - ◆ 难点
 - ◆ 专用的KVS存储
- 3个具体实现
 - ◆ Amazon's Dynamo
 - ◆ Google's BigTable
 - ◆ Facebook's Cassandra



通过KVS支持Internet服务

- 通过一个中心服务器，如Web或者应用服务器，来完成
- 两个主要的问题:
 1. 来自于不同客户端的多个并发请求
 - GETs, PUTs, DELETEs 等
 2. 这些请求来自于网络的不同部分, 有消息传递延迟
 - 需要时间把请求发送至服务器
 - 将按照接受的顺序请求服务 (为什么呢?)



KVS的并发管理

- 如果同时收到多个GET操作?
 - ◆ ... 是不同的keys?
 - ◆ ...是相同的keys?
- 如果同时收到多个PUT操作? 或者是一个GET 和一个PUT?
- 保护的单元粒度是多少 (并发控制)?

并发控制

- 多数系统在每个数据项上使用locks锁
 - ◆ 每个请求者申请上锁
 - ◆ Lock manager锁管理器处理请求 (通常使用FIFO顺序) :<.....
 - 锁管理器授权申请者的请求锁
 - 申请者修改数据
 - 完成之后释放锁
- There are several kinds of locks, and several other alternatives
 - ◆ Example: S/X lock
 - ◆ See CIS 455 for more details

目录

- 传统方法
- Key-value 存储 (KVS)
 - ◆ 基本概念; 操作
 - ◆ KVS 举例
 - ◆ KVS 及并发
 - ◆ Key-multi-value 存储; 光标
- 云平台的Key-value存储
 - ◆ 难点
 - ◆ 专用的KVS存储
- 3个具体实现
 - ◆ Amazon's Dynamo
 - ◆ Google's BigTable
 - ◆ Facebook's Cassandra



◁键-多值▷存储系统

- 如果一个key对应多个值，怎么办？
 - ◆ 例如：一个人可以有多个头像
- 选项1: 多值→一个集合
 - ◆ PUT操作等于先GET → add → PUT
- 选项 2: KVS可以直接保存一键多值
 - ◆ 通过光标 进行滑动
 - ◆ 类似于Java的Iterator

- 如何查找一键对应的所有值呢？

- ◆ 有好多方法

- 思路: 使用光标

- ◆ 使用下面的编程模式:

```
cursor = kvs.getFirstMatch(key);  
  
while (cursor != null) {  
    value = cursor.getValue();  
    cursor = kvs.getNextMatch(key, cursor);  
}
```


小结: KVS

- **KVS: 管理持久化数据的抽象**
 - ◆ 接口只有 **PUT** 及 **GET** (+可能还有 **DELETE**)
 - ◆ 不同的变种版本, 如允许一键多值
 - ◆ 举例: 分布式哈希表, 管理数组, ...
 - ◆ 深度可扩展的真实系统
- **问题: 并发控制**
 - ◆ 从**KVS**的角度上看, 不同的键值是独立的
 - ◆ 难以原子性的改变多值
 - ◆ 有些应用可能会取得更高级的锁

目录

- 传统方法
- Key-value 存储 (KVS)
 - ◆ 基本概念; 操作
 - ◆ KVS举例
 - ◆ KVS 及并发
 - ◆ Key-multi-value 存储; 光标
- 云平台的Key-value存储
 - ◆ 难点
 - ◆ 专用的KVS存储
- 3个具体实现
 - ◆ Amazon's Dynamo
 - ◆ Google's BigTable
 - ◆ Facebook's Cassandra



云端的KVS系统

- 托管大型的数据集

- ◆ 例如: Amazon 产品列表, eBay 产品列表, Facebook 页面, ...

- 思路: 云端大硬盘的抽象:

- ◆ 耐用性Durability– 即使宕机, 数据亦不会消失
- ◆ 100% 可用性Availability – 总是能访问云服务
- ◆ 零 延时: 世界任何一个地方均是如此!
- ◆ 最小化的带宽使用– 仅当绝对需要的时候才发生网络数据包
- ◆ 并发更新时满足隔离性Isolation – 确保数据一致性

现实情况呢？

- 可行吗？
- 云平台是存在于物理网络之上
 - ◆ 通信需要时间开销
 - ◆ 网络带宽永远是有限的，不管是骨干网还是用户端
- 云平台还存在有缺陷的硬件
 - ◆ 硬盘宕机
 - ◆ 服务器宕机
 - ◆ 软件还需有缺陷、漏洞
- 如何考虑到这些实际情况，云平台会怎样呢？

本质上各方的妥协和平衡

- 现实情况下，我们其实不需要满足所有的这些目标
- 一些观察:
 1. 只读 (或者多数读)数据更加易于实现
 - 多副本! 没有并发问题!
 - 很显然只有某些情况符合这个场景- 举例?
 2. 粒度问题: “几个大的数据对象” 往往比“很多个小的数据对象”更加容忍长延时
 - 请求数少、但是更多是在客户端处理
 - 但是在数据复制或者更新时候代价更高!
 3. 为大的多数读操作数据对象和小的读写数据对象分别设计不同的系统，可行吗？行!
 - 不同的需求 → 不同的技术手段

专用的KVS系统

- 云端KVS系统往往在特定需要或者场景下进行妥协
- 举例: Amazon的技术方案
 - ◆ Simple Storage Service (S3):
 - 大的数据对象 – 文件、虚拟机等.
 - 数据对象变化频率低
 - 数据对象对存储系统是透明的
 - ◆ SimpleDB/DynamoDB:
 - 小数据对象 – Java objects, 记录等
 - 更新频率高; 往往需要一致性
 - 存在多属性, 均需要暴露给存储系统

小结：云端KVS系统

- 我们偏向于简单、干净的方式
 - ◆ PUT, GET
- Practical constraints require compromises
 - ◆ 理想情况：耐久性, 可用性, 一致性, 吞吐量, ...
 - ◆ 现实情况：传播延时、不可靠的硬件/软件, ...
- 因此, 需要各种妥协和平衡
 - ◆ 例如, 针对不同的任务负载, 设计专用的KVSs系统
 - ◆ 没有可能一刀切(No one-size-fits-all)方法
 - ◆ 不同的情况设计不同的技术方法

目录

- 传统方法



- Key-value 存储 (KVS)



- ◆ 基本概念; 操作



- ◆ KVS举例



- ◆ KVS 及并发



- ◆ Key-multi-value 存储; 光标

- 云平台的Key-value存储



- ◆ 难点



- ◆ 专用的KVS存储



- 3个具体实现

- ◆ Amazon's Dynamo



- ◆ Google's BigTable

- ◆ Facebook's Cassandra



Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall and Werner Vogels

SOSP 2007

Dynamo

- An infrastructure to host services
- Reliability and fault-tolerance at massive scale
- Availability providing an "always-on" experience
- Cost-effectiveness
- Performance

Context

- Amazon's e-commerce platform
 - ◆ Shopping cart served tens of millions requests, over 3 million checkouts in a single day
- Unavailability == \$\$\$
- No complex queries → 与DBMS SQL有明显区别
- Managed system
 - ◆ Add/remove nodes
 - ◆ Security
 - ◆ Byzantine nodes

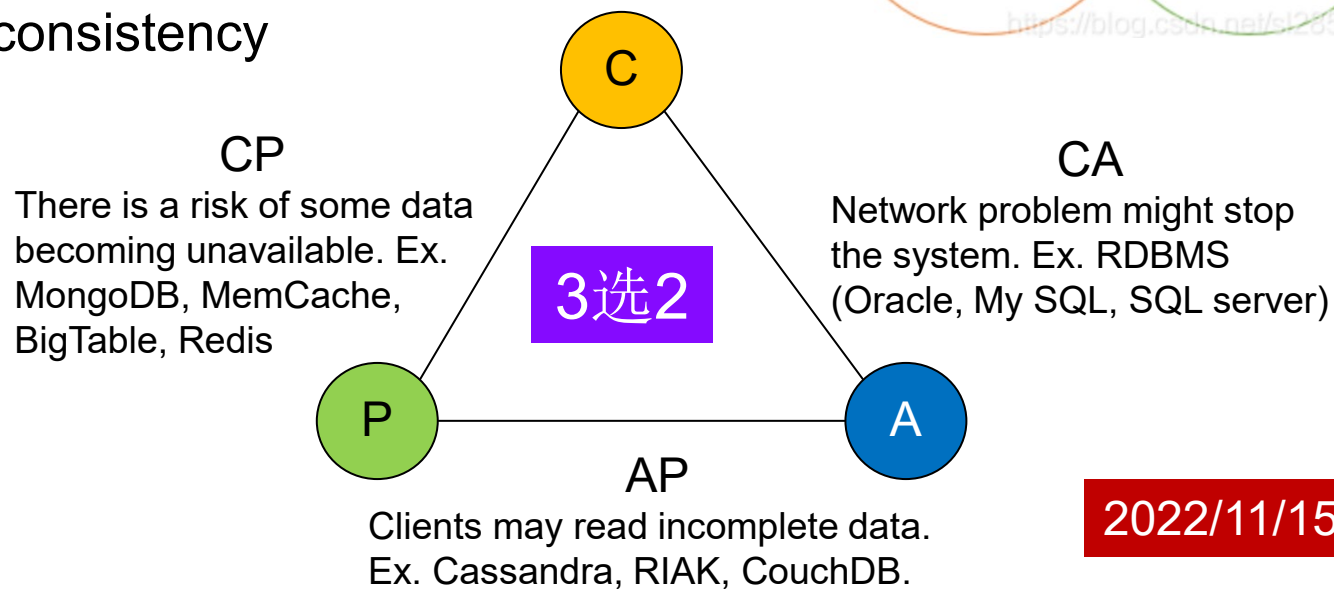
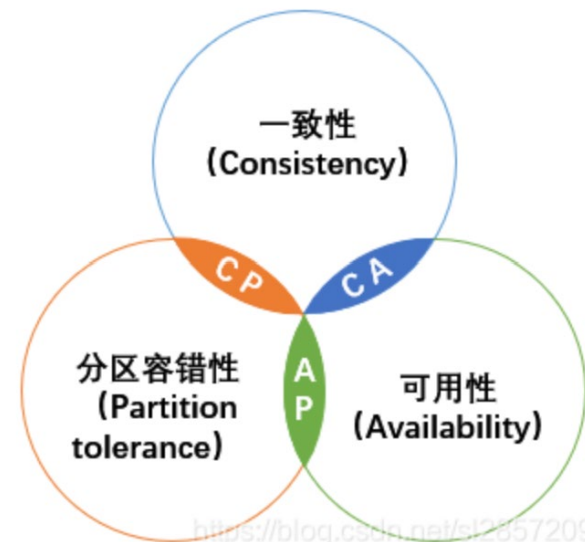
CAP Theorem

■ Only two possible at the same time

- ◆ Consistency
- ◆ Availability
- ◆ Partition-tolerance

■ Dynamo, target applications:

- ◆ Availability and Partition-tolerance
- ◆ Eventual consistency



2022/11/15

Clients view on Consistency

- Strong consistency.

- ◆ Single storage image. After the update completes, any subsequent access will return the updated value.

- Weak consistency.

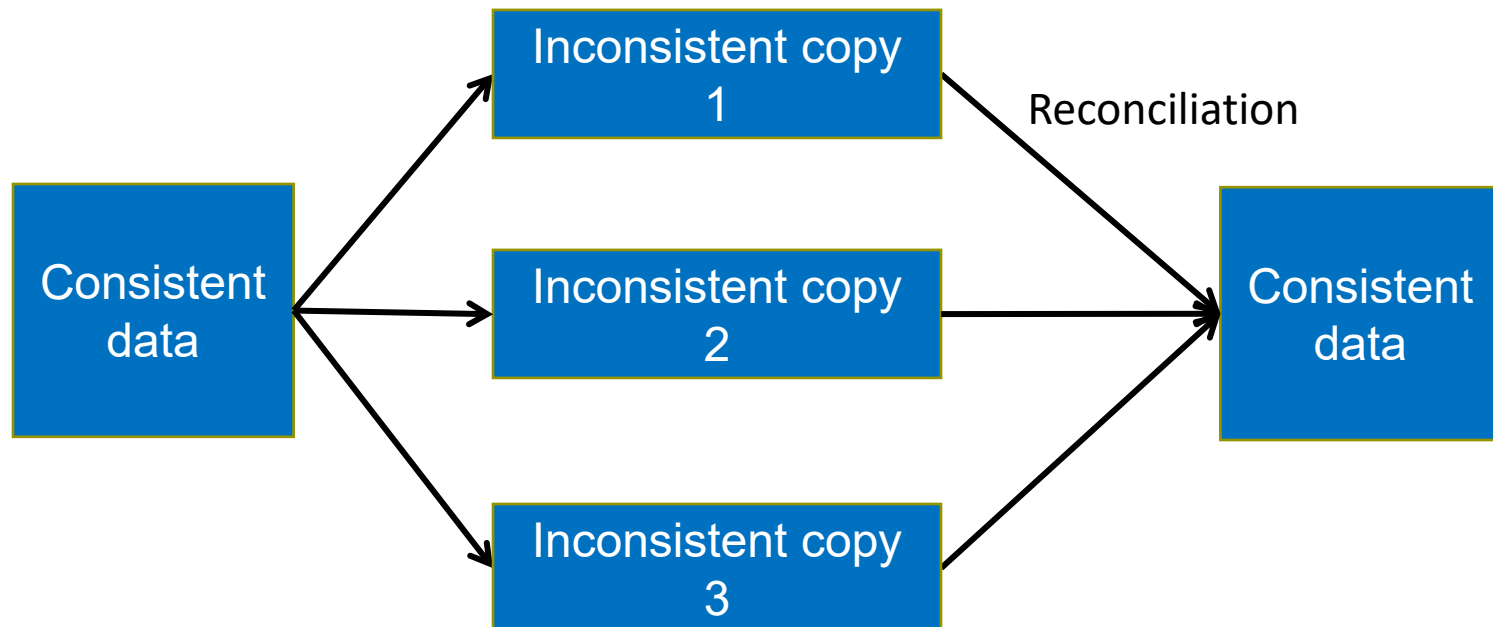
- ◆ The system does not guarantee that subsequent accesses will return the updated value.
- ◆ Inconsistency window.

- Eventual consistency.

- ◆ Form of weak consistency
- ◆ If no new updates are made to the object, eventually all accesses will return the last updated value.

Eventual consistency

- Causal consistency
- Read-your-writes consistency
- ...



Requirements

- Query model

- ◆ Simple read/write operations on small data items

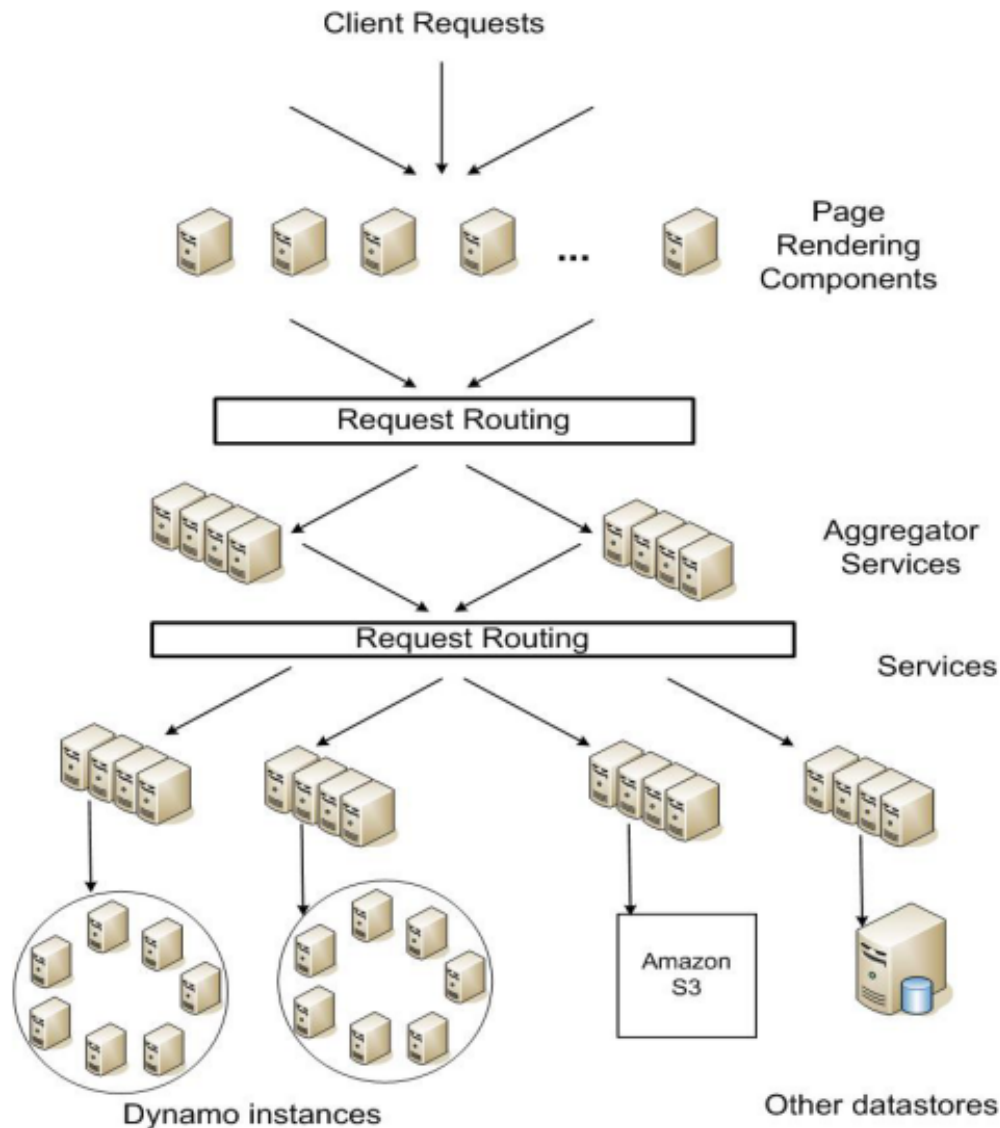
- ACID properties

- ◆ Weaker consistency model
- ◆ No isolation, only single key updates

- Efficiency

- ◆ Tradeoff between performance, cost efficiency, availability and durability guarantees

Amazon Store Architecture



Design considerations

- Conflict resolution
 - ◆ When
 - ◆ Who
- Scalability
- Symmetry
- Decentralization
- Heterogeneity

The big picture

Easy usage

Load-balancing

Replication

Eventual
consistency

High
availability

Easy
management

Failure-
detection

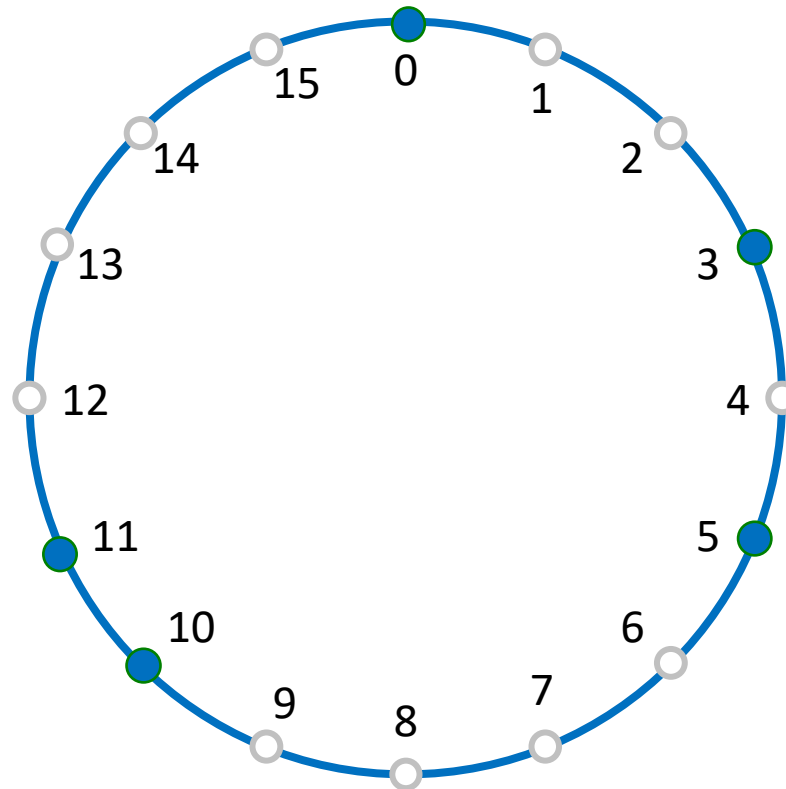
Scalability

Easy usage: Interface

- `get(key)`
 - ◆ return single object or list of objects with conflicting version and context
- `put(key, context, object)`
 - ◆ store object and context under key
- Context encodes system meta-data, e.g. version number

Data partitioning

- Based on consistent hashing
- Hash key and put on responsible node



Load balancing

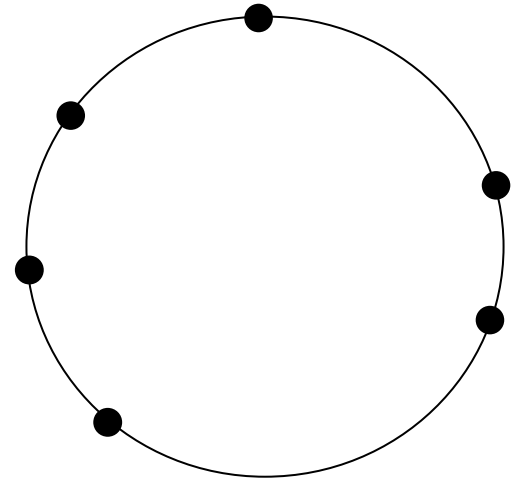
■ Load

- ◆ Storage bits
- ◆ Popularity of the item
- ◆ Processing required to serve the item
- ◆ ...

■ Consistent hashing may lead to imbalance

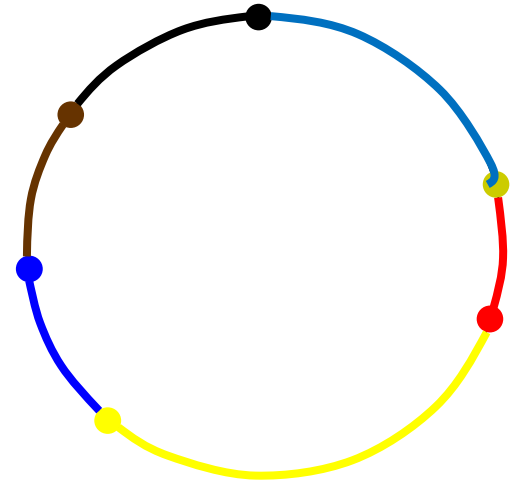
Load imbalance (1/5)

- Node identifiers may not be balanced



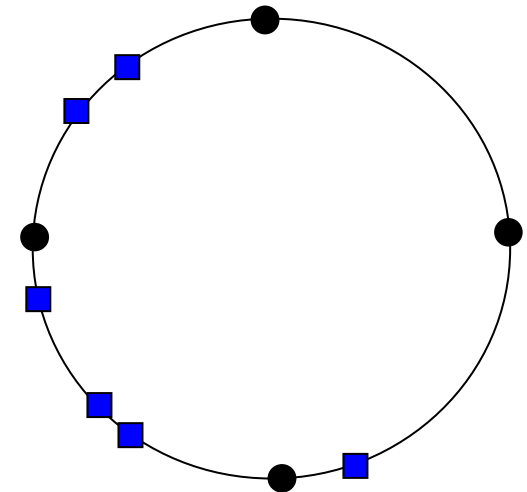
Load imbalance (2/5)

- Node identifiers may not be balanced



Load imbalance (3/5)

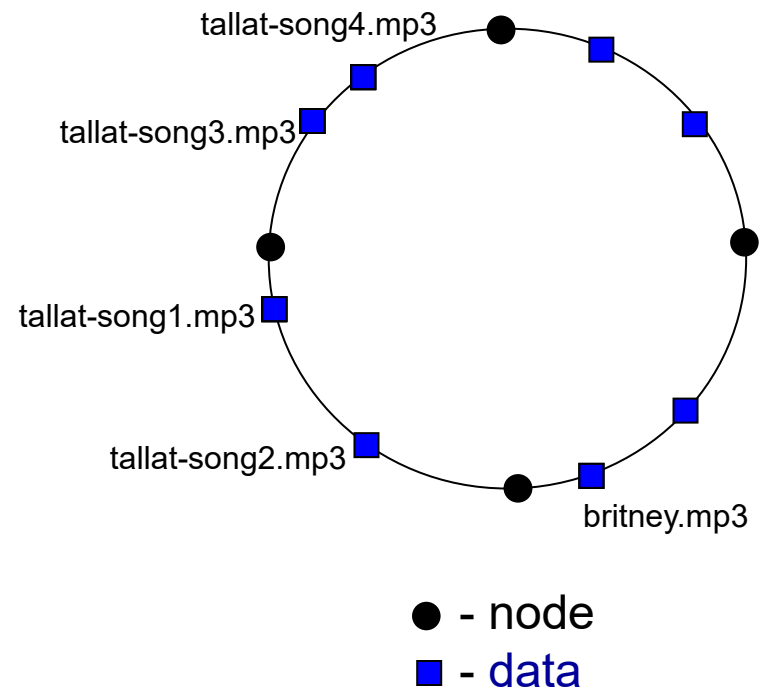
- Node identifiers may not be balanced
- Data identifiers may not be balanced



● - node
■ - data

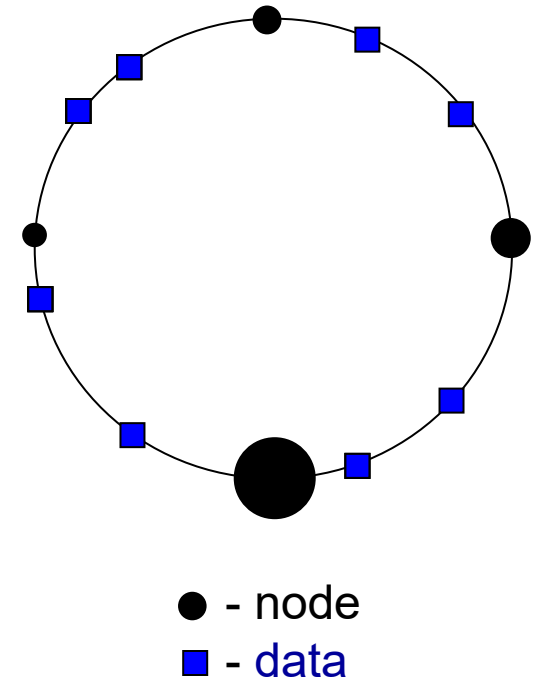
Load imbalance (4/5)

- Node identifiers may not be balanced
- Data identifiers may not be balanced
- Hot spots



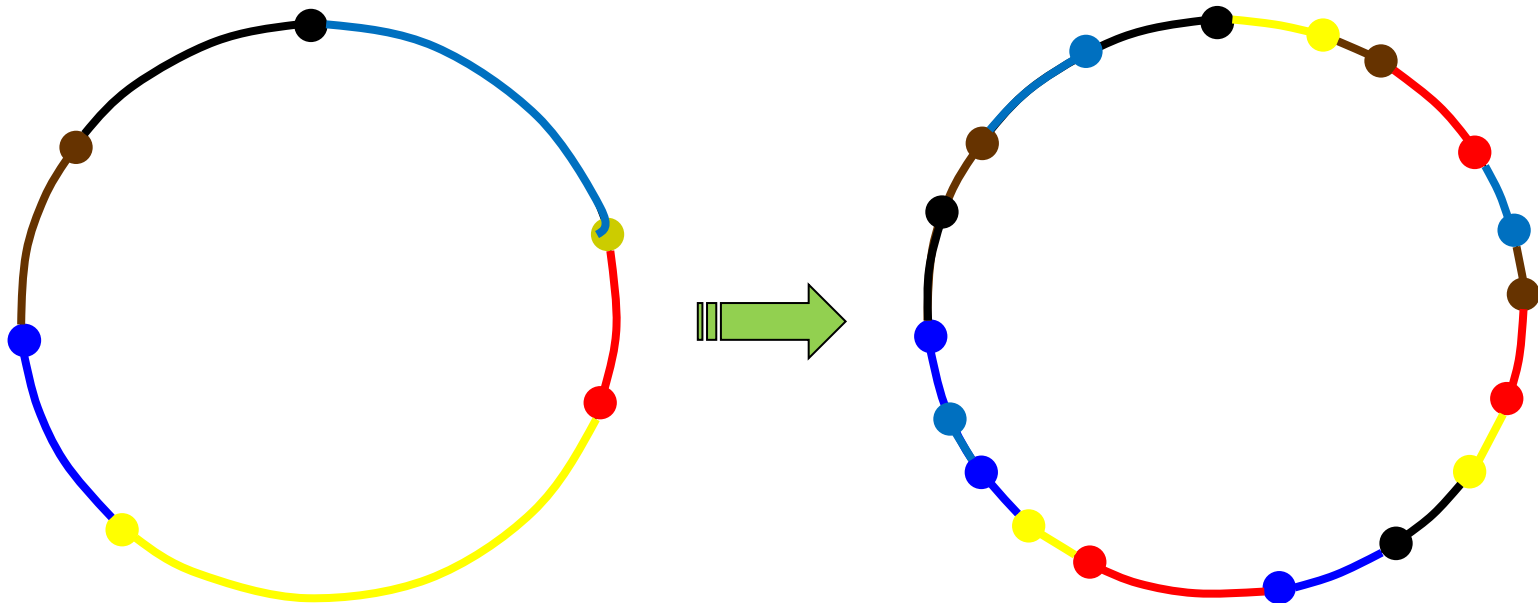
Load imbalance (5/5)

- Node identifiers may not be balanced
- Data identifiers may not be balanced
- Hot spots
- Heterogeneous nodes



Load balancing via Virtual Servers

- Each physical node picks multiple random identifiers
 - ◆ Each identifier represents a virtual server
 - ◆ Each node runs multiple virtual servers
- Each node responsible for noncontiguous regions

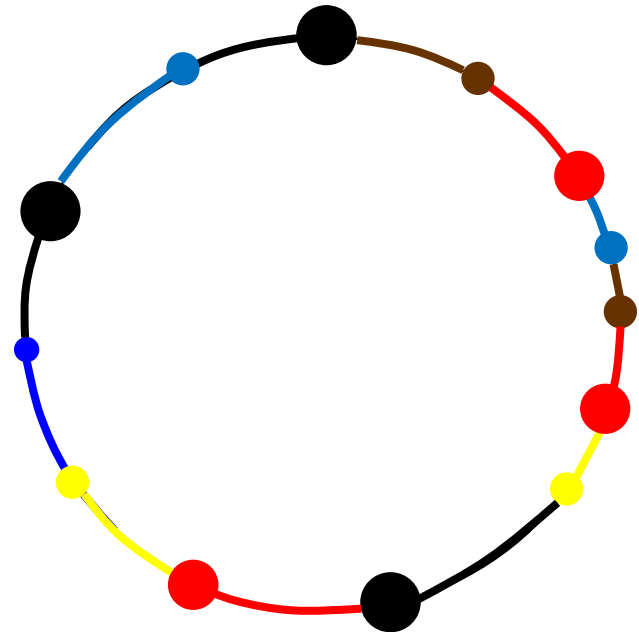


Virtual Servers

■ How many virtual servers?

- ◆ For homogeneous, all nodes run **$\log N$** virtual servers
- ◆ For heterogeneous, nodes run **$c \log N$** virtual servers, where 'c' is
 - small for weak nodes
 - large for powerful nodes

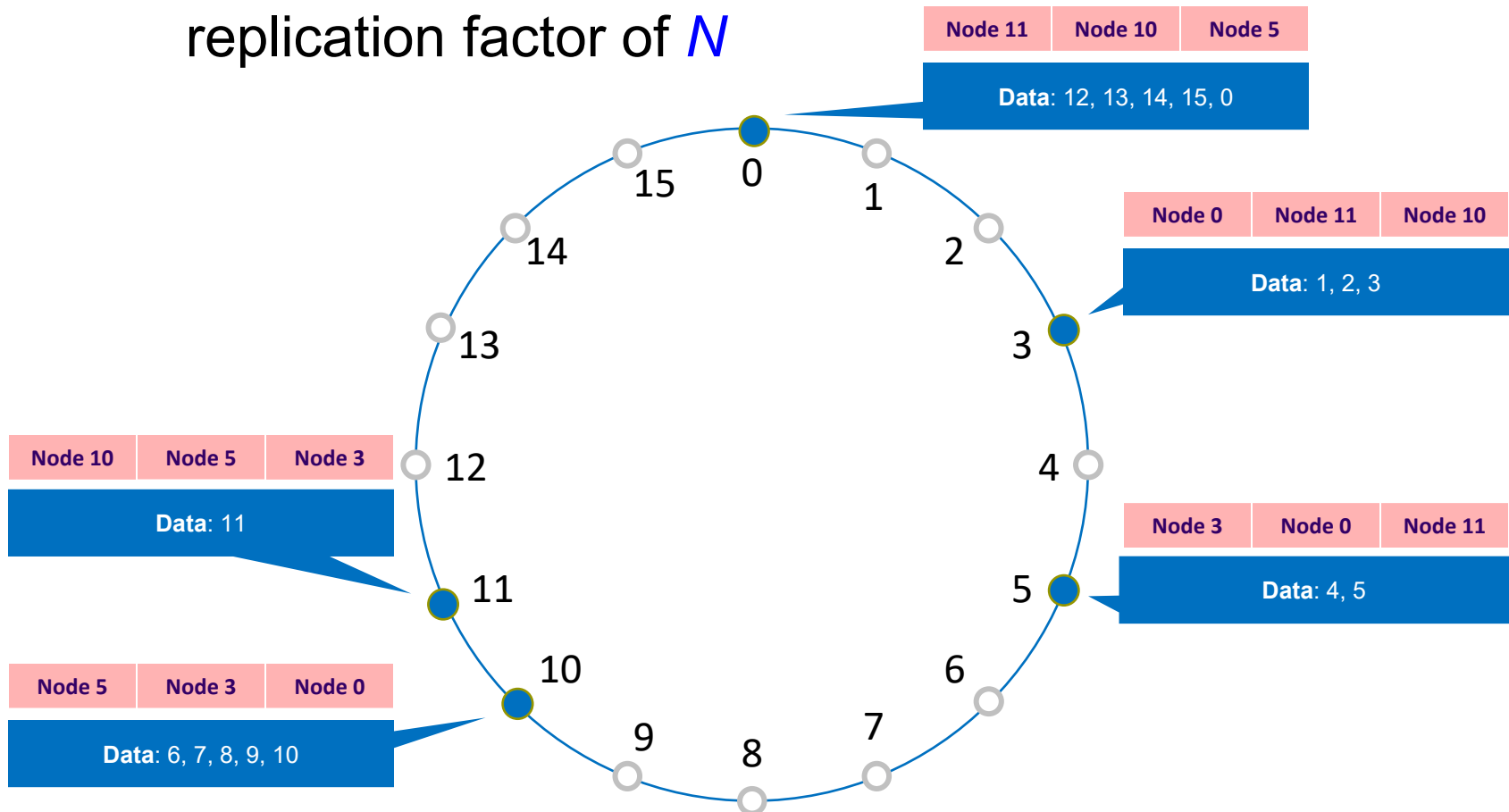
- Move virtual servers from heavily loaded physical nodes to lightly loaded physical nodes



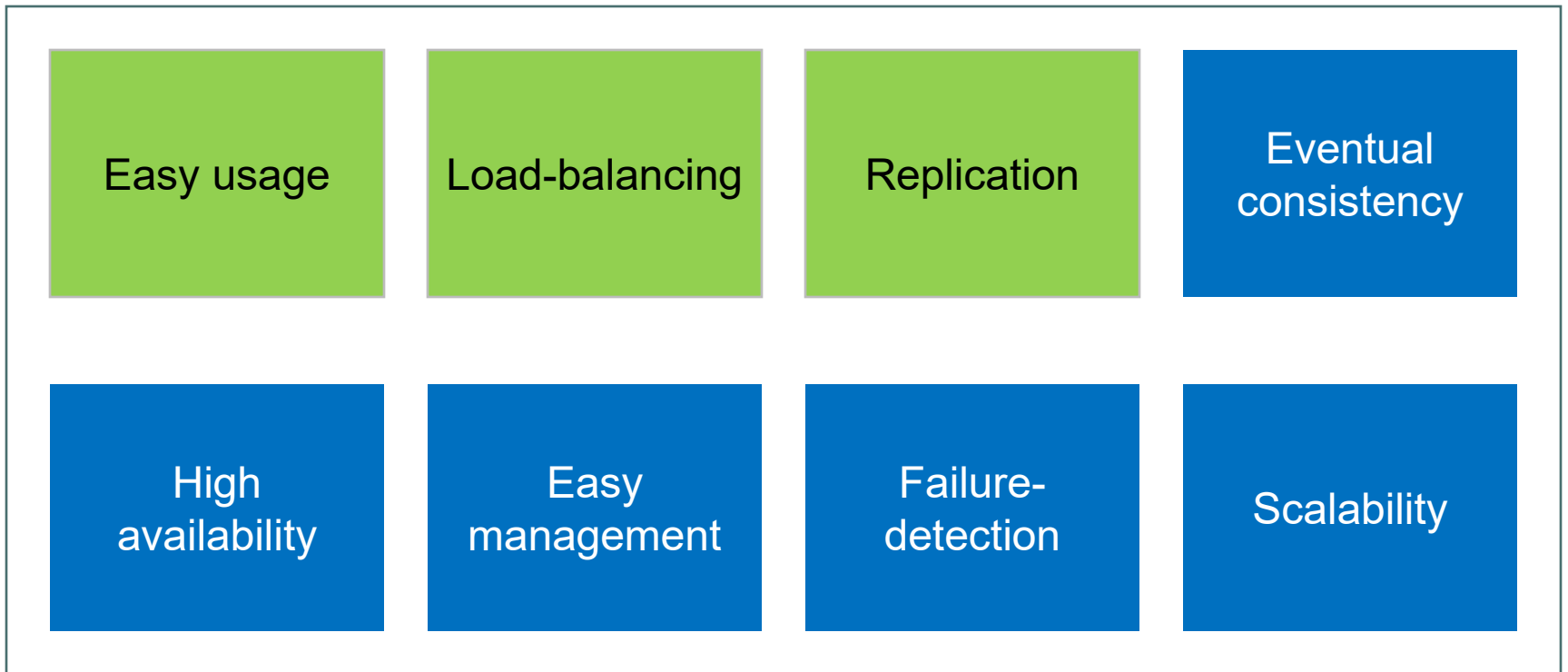
Replication

■ Successor list replication

- ◆ Replicate the data of your N closest neighbors for a replication factor of N



The big picture



Data versioning (1/3)

- Eventual consistency, updates propagated asynchronously
- Each modification is a new and immutable version of the data
 - ◆ Multiple versions of an object
- New versions can subsume older versions
 - ◆ Syntactic reconciliation
 - ◆ Semantic reconciliation

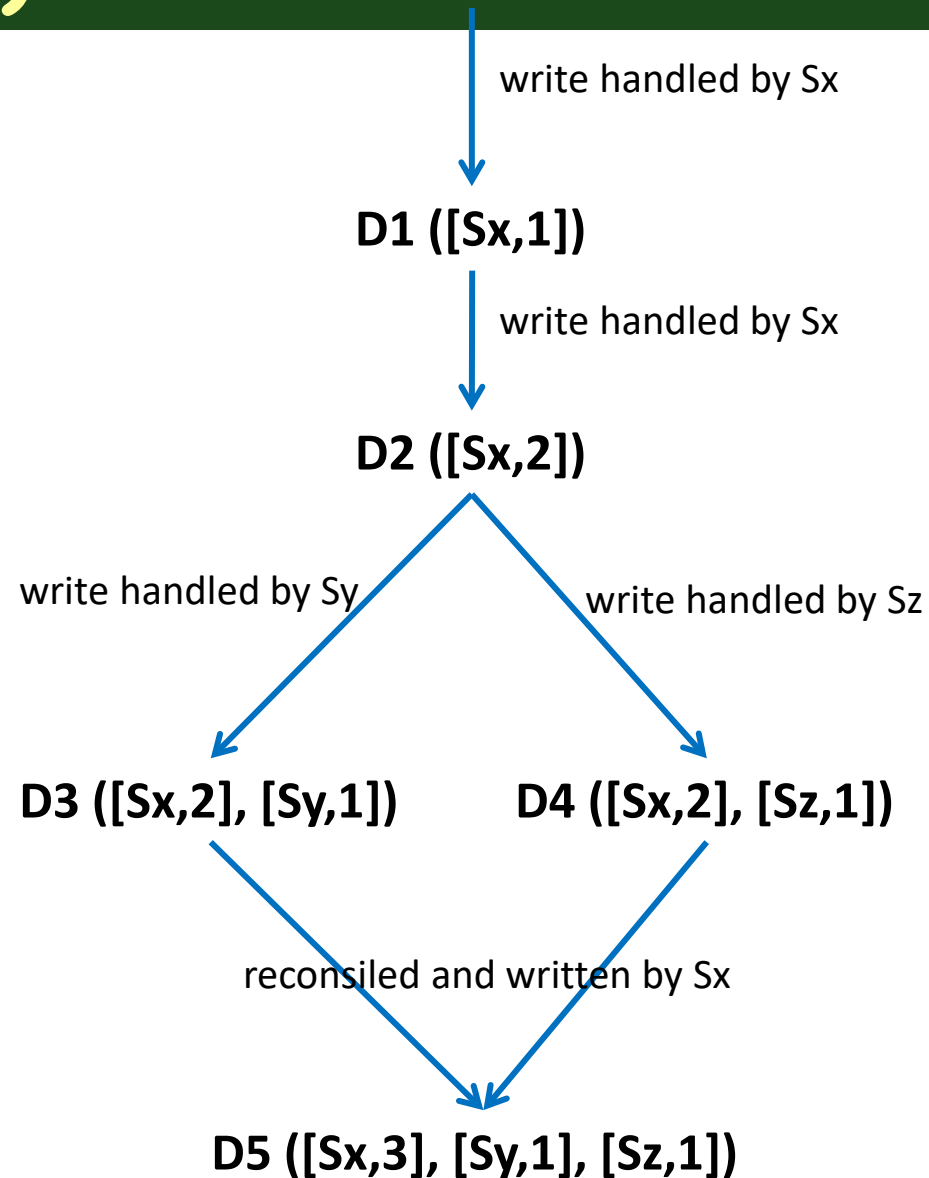
2022/11/22

Data versioning (2/3)

- Version branching due to failures, network partitions, etc.
- Target applications aware of multiple versions
- Use vector clocks for capturing causality
 - ◆ If causal, older version can be forgotten
 - ◆ If concurrent, conflict exists requiring reconciliation
- A put requires a context, i.e. which version to update

Data versioning (3/3)

- Client C1 writes new object
 - ◆ say via Sx
- C1 updates the object
 - ◆ say via Sx
- C1 updates the object
 - ◆ say via Sy
- C2 reads D2 and updates the object
 - ◆ Say via Sz
- Reconciliation

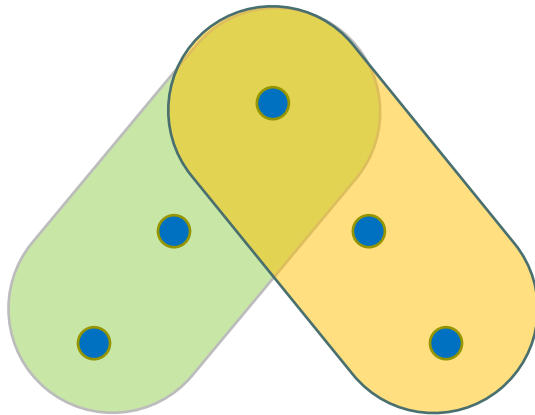


Execution of operations

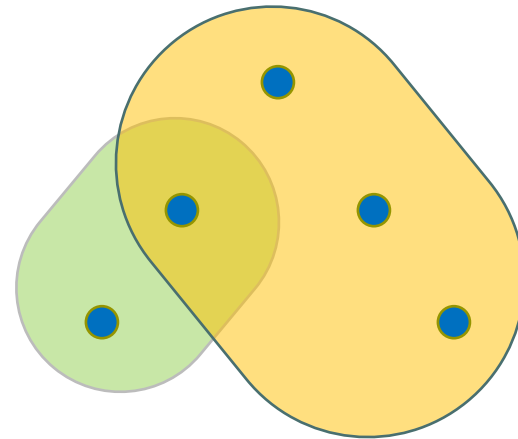
- put and get operations
- Client can send the request
 - ◆ to the node responsible for the data
 - Save on latency, code on client
 - ◆ to a generic load balancer
 - Extra hop

Quorum systems

- R / W : minimum number of nodes that must participate in a successful read / write
- $R + W > N$ (overlap)



$R=3, W=3, N=5$



$R=4, W=2, N=5$

put (key, value, context)

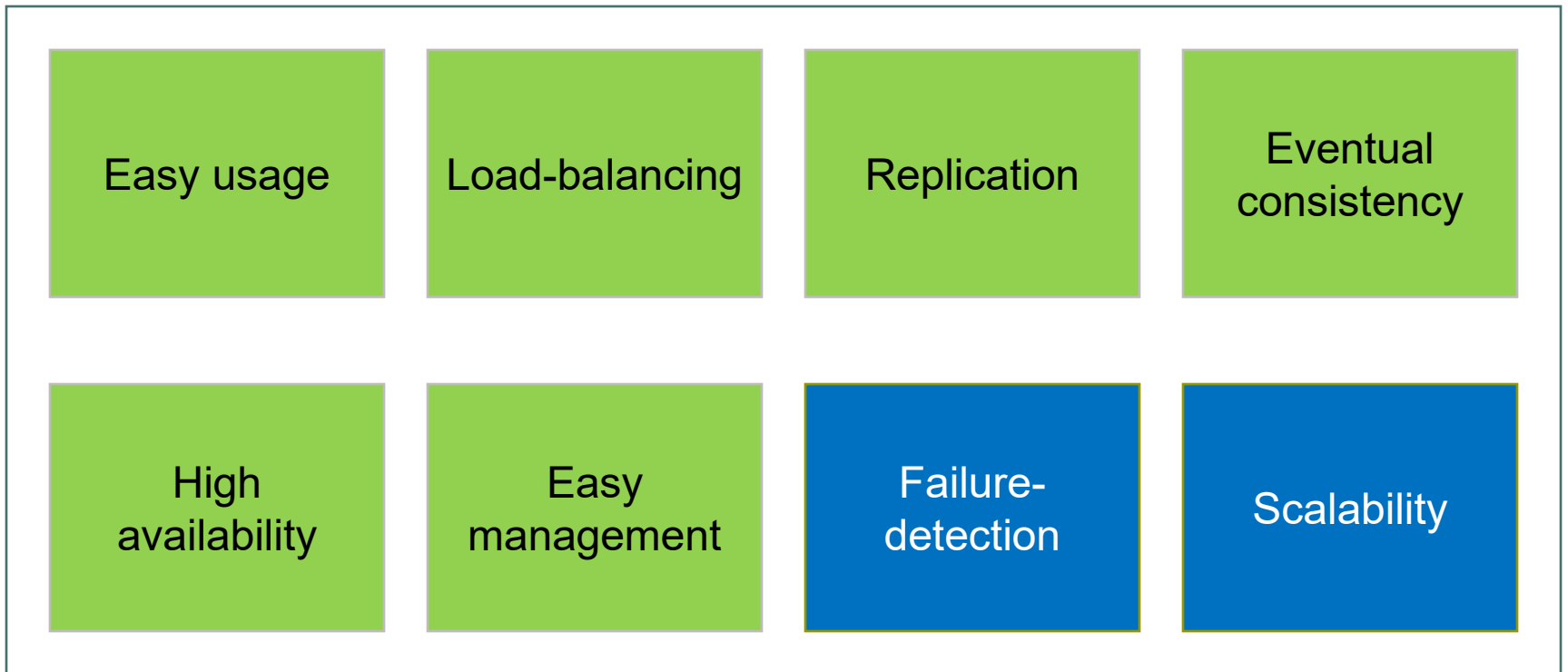
- Coordinator generates new vector clock and writes the new version locally
- Send to N nodes
- Wait for response from W-1 nodes
- Using W=1
 - ◆ High availability for writes
 - ◆ Low durability

$(\text{value}, \text{context}) \leftarrow \text{get}(\text{key})$

- Coordinator requests existing versions from N
- Wait for response from R nodes
- If multiple versions, return all versions that are causally unrelated
- Divergent versions are then reconciled
- Reconciled version written back

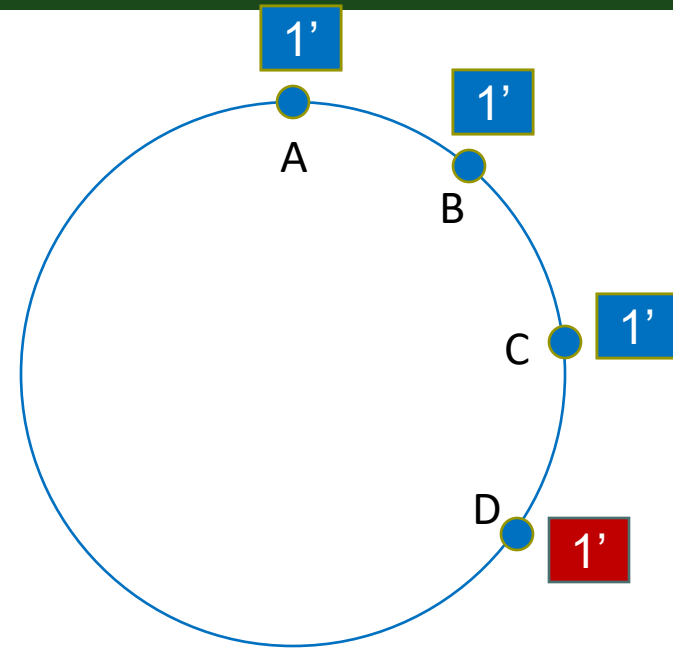
- Using $R=1$
 - ◆ High performance read engine

The big picture



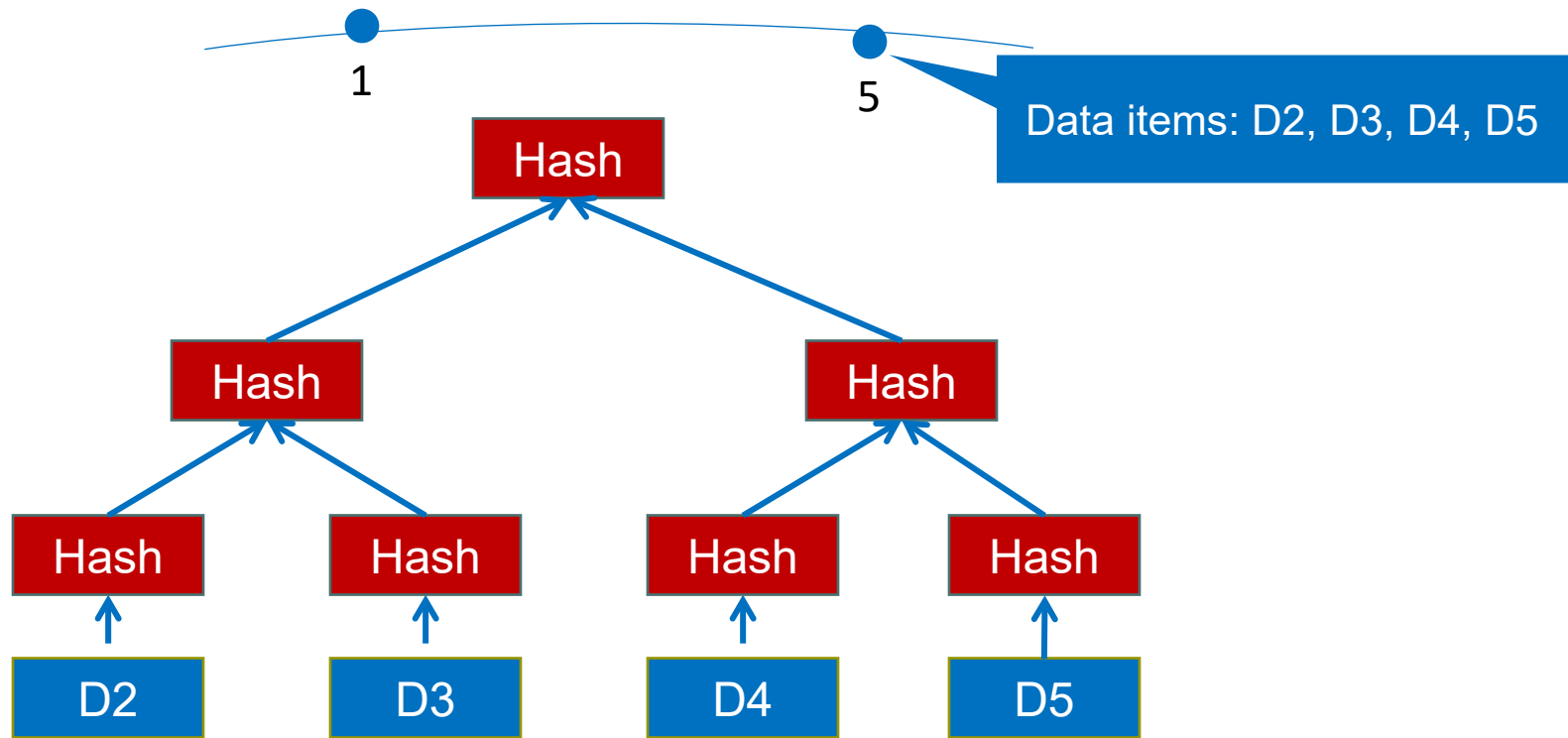
Handling transient failures

- A managed system
- Which N nodes to update?
- Say A is unreachable
- 'put' will use D
- Later, D detects A is alive
 - ◆ send the replica to A
 - ◆ remove the replica
- Tolerate failure of a data center
 - ◆ Each object replicated across multiple data centers



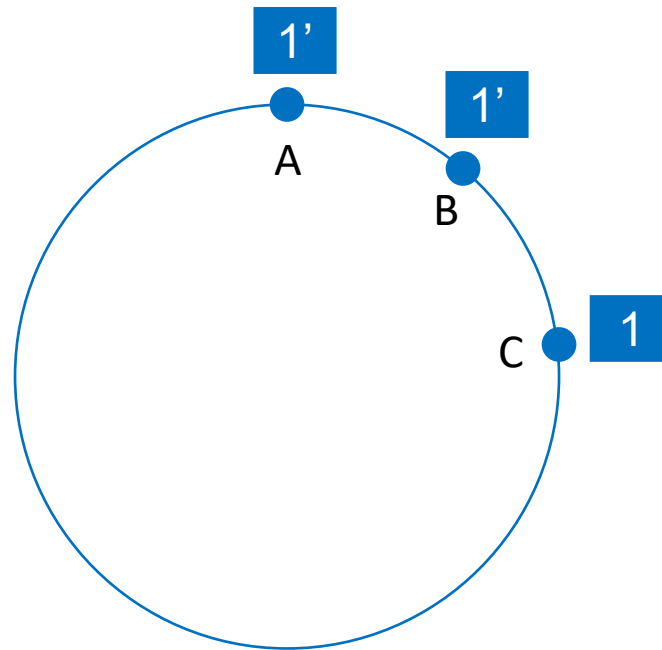
Handling permanent failures (1/2)

- Anti-entropy for replica synchronization
- Use Merkle trees for fast inconsistency detection and minimum transfer of data



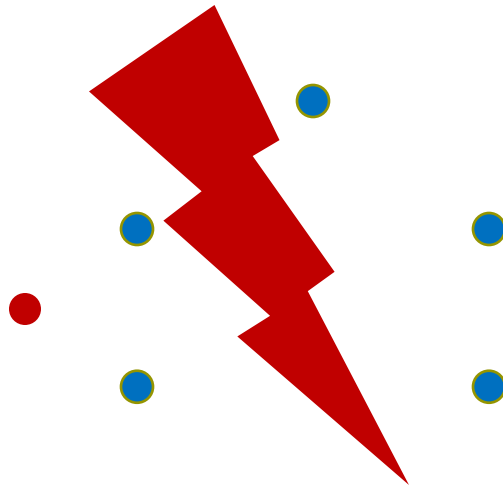
Handling permanent failures (2/2)

- Nodes maintain Merkle tree of each key range
- Exchange root of Merkle tree to check if the key ranges are up-to-date



Quorums under failures systems

- Due to partitions, quorums might not exist
- Create transient replicas
- Reconcile after partition heals



$R=3, W=3, N=5$

Membership

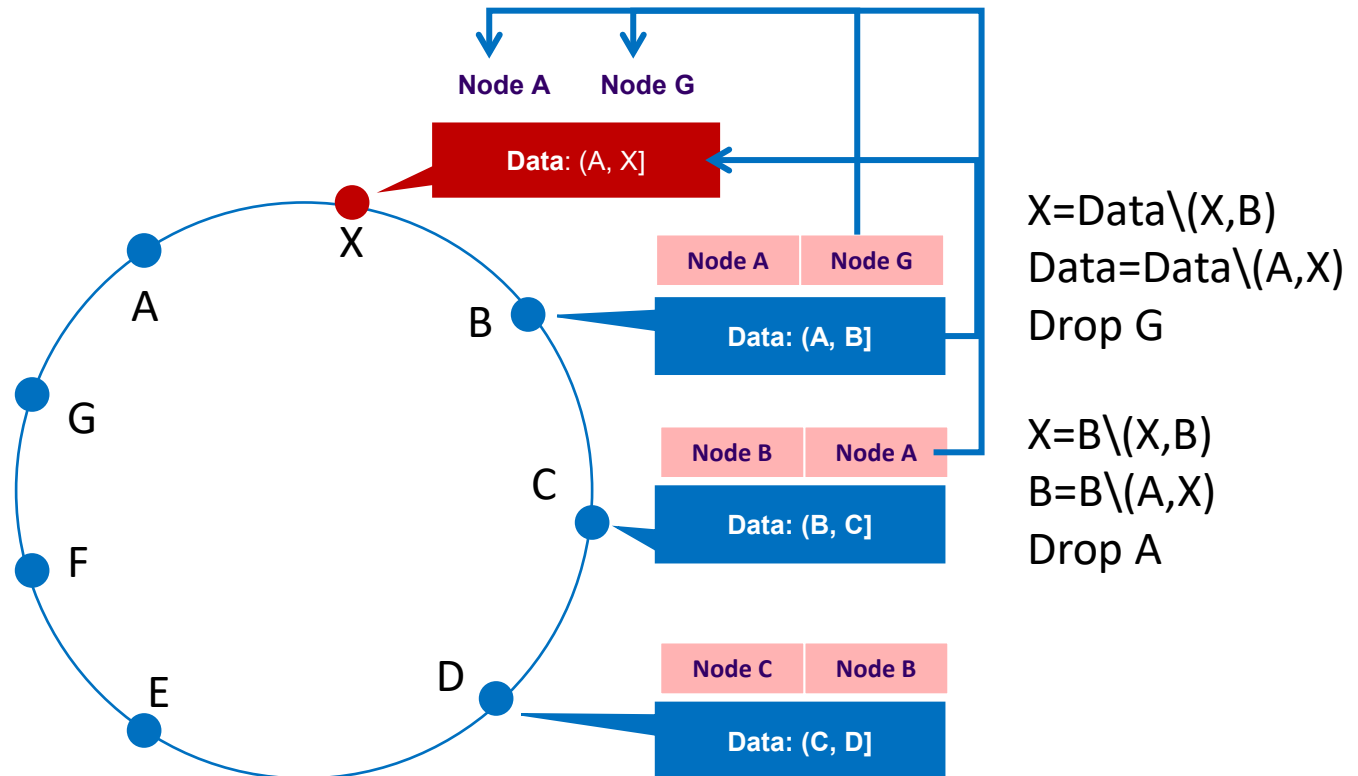
- A managed system
 - ◆ Administrator explicitly adds and removes nodes
- Receiving node stores changes with time stamp
- Gossiping to propagate membership changes
 - ◆ Eventually consistent view
 - ◆ $O(1)$ hop overlay
 - $\log(n)$ hops, e.g. $n=1024$, 10 hops, 50ms/hop, 500ms

Failure detection

- Passive failure detection
 - ◆ Use pings only for detection from failed to alive
 - ◆ A detects B as failed if it doesn't respond to a message
 - ◆ A periodically checks if B is alive again
- In the absence of client requests, A doesn't need to know if B is alive
 - ◆ Permanent node additions and removals are explicit

Adding nodes

- A new node X added to system
- X is assigned key ranges w.r.t. its virtual servers
- For each key range, it transfers the data items



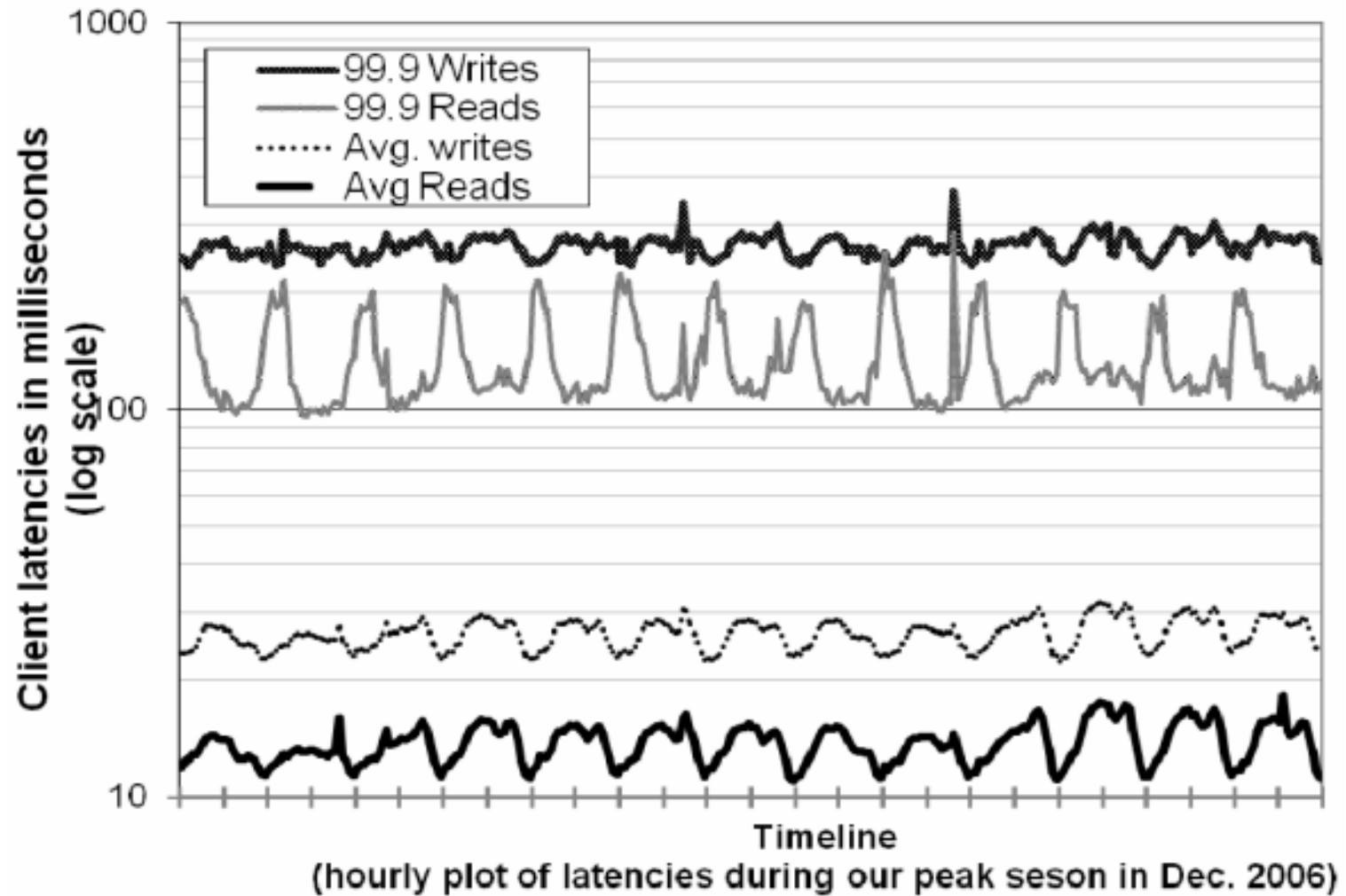
Removing nodes

- Reallocation of keys is a reverse process of adding nodes

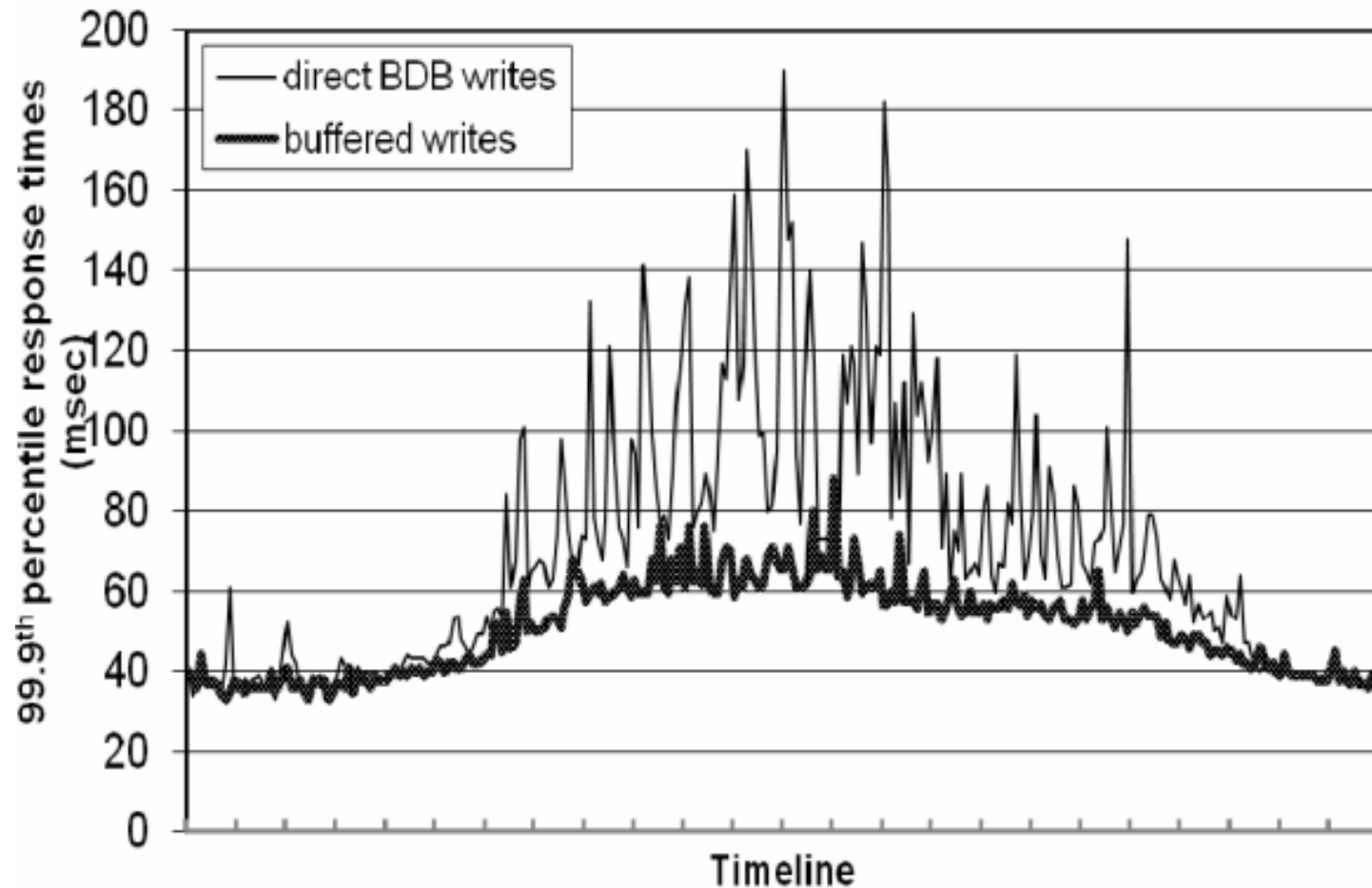
Implementation details

- Local persistence
 - ◆ BDS, MySQL, etc.
- Request coordination
 - ◆ Read operation
 - Create context
 - Syntactic reconciliation
 - Read repair
 - ◆ Write operation
 - Read-your-writes

Evaluation

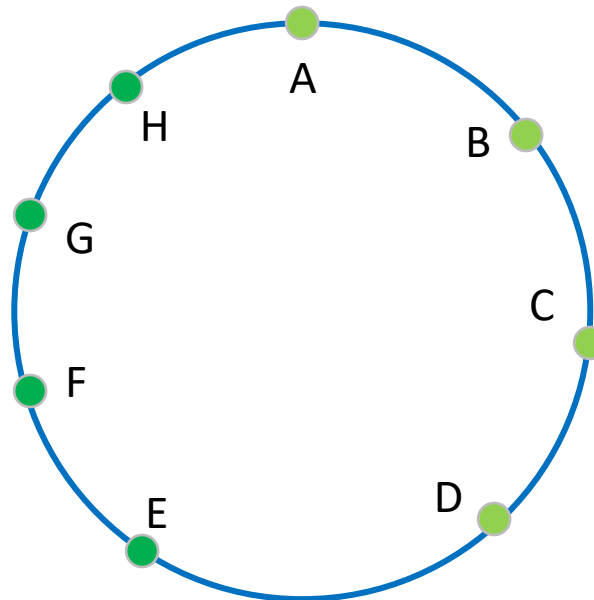


Evaluation



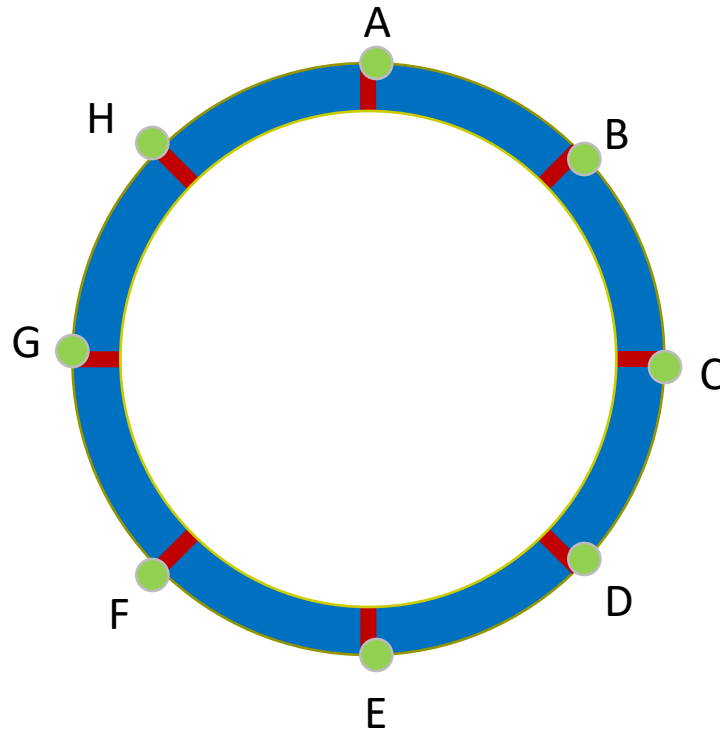
Partitioning and placement (1/2)

- Data ranges are not fixed
 - ◆ More time spend to locate items
 - ◆ More data storage needed for indexing
- Inefficient bootstrapping
- Difficult to archive the whole data



Partitioning and placement (2/2)

- Divide data space into equally sized ranges
- Assign ranges to nodes



Versions of an item

■ Reason

- ◆ Node failures, data center failures, network partitions
- ◆ Large number of concurrent writes to an item

■ Occurrence

- ◆ 99.94 % one version
- ◆ 0.00057 % two versions
- ◆ 0.00047 % three versions
- ◆ 0.00009 % four versions

■ Evaluation: versioning due to concurrent writes

Client vs Server coordination

- Read requests coordinated by any Dynamo node
- Write requests coordinated by a node replicating the data item
- Request coordination can be moved to client
 - ◆ Use libraries
 - ◆ Reduces latency by saving one hop
 - ◆ Client library updates view of membership periodically

End notes

Peer-to-peer techniques have been the key enablers for building Dynamo:

- "... decentralized techniques can be combined to provide a single highly-available system."

References

- **Dynamo: amazon's highly available key-value store**, Giuseppe DeCandia et. al., SOSP 2007.
- **Bigtable: A Distributed Storage System for Structured Data**, Fay Chang et. al., OSDI 2006.
- **Cassandra** - <http://cassandra.apache.org/>
- **Eventual consistency** - http://www.allthingsdistributed.com/2008/12/eventually_consistent.html
- Key values stores, No SQL