NoSQL Databases

金培权（jpq@ustc.edu.cn）

主要内容

- NoSQL简介
- NoSQL主要的类型
- NoSQL的分布式系统基础
- LSM-tree

一、NoSQL简介

- **Definition (from http://nosql-database.org)**
  - Next Generation Databases mostly addressing some of the points: being non-relational, distributed, open-source and horizontal scalable.
  - The original intention has been modern Web-scale databases. The movement began early 2009 and is growing rapidly. Often more characteristics apply as: schema-free, easy replication support, simple API, eventually consistent /BASE (not ACID), a huge data amount, and more.
  - So the misleading term *"nosql"* (the community now translates it mostly with "not only sql") should be seen as an alias to something like the definition above.

一、NoSQL简介

- **NoSQL特点：**
  - **Non relational**
  - **Scalability**
  - **No pre-defined schema**
  - **CAP not ACID**

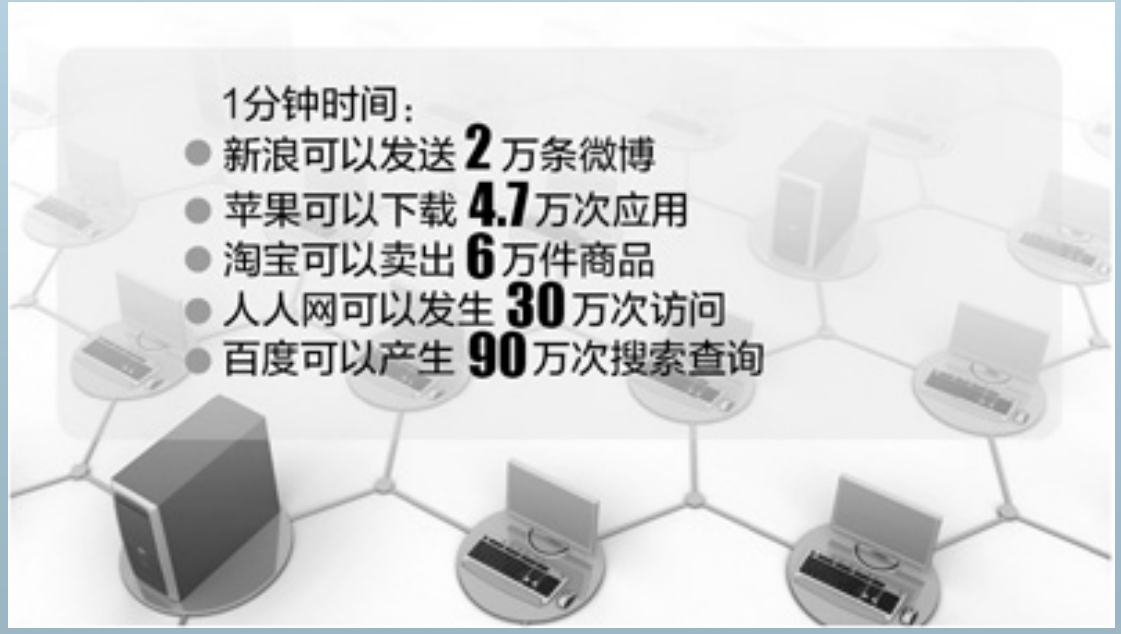
一、NoSQL简介

- 现在已有很多公司使用了NoSQL数据库：
  - Google
  - Facebook
  - Adobe
  - Foursquare
  - LinkedIn
  - 字节、腾讯、阿里、新浪、华为.....

1、NoSQL兴起的原因

（1）RDBMS无法满足Web 2.0的需求：

- 无法满足海量数据的管理需求　TB→PB →ZB
- 无法满足数据高并发的需求　1K→1M→10M 并发
- 无法满足高可扩展性和高可用性的需求



1、NoSQL兴起的原因

（2）“One size fits all”模式很难适用于截然不同的业务场景

- 关系模型作为统一的数据模型既被用于数据分析（OLAP），也被用于在线业务（OLTP）。但这两者一个强调高吞吐，一个强调低延时，已经演化出完全不同的架构。用同一套模型来抽象显然是不合适的
  - Hadoop就是针对数据分析
  - MongoDB、Redis等针对在线业务，两者都抛弃了关系模型

1、NoSQL兴起的原因

（3）关系数据库的关键特性包括完善的事务机制和高效的查询机制。这些关键特性在Web 2.0时代出现了变化：

- Web 2.0网站系统通常不要求严格的数据库事务
- Web 2.0并不要求严格的读写一致性
- Web 2.0通常不包含大量复杂的SQL查询（去结构化，存储空间换取更好的查询性能）

2、NoSQL vs. RDBMS

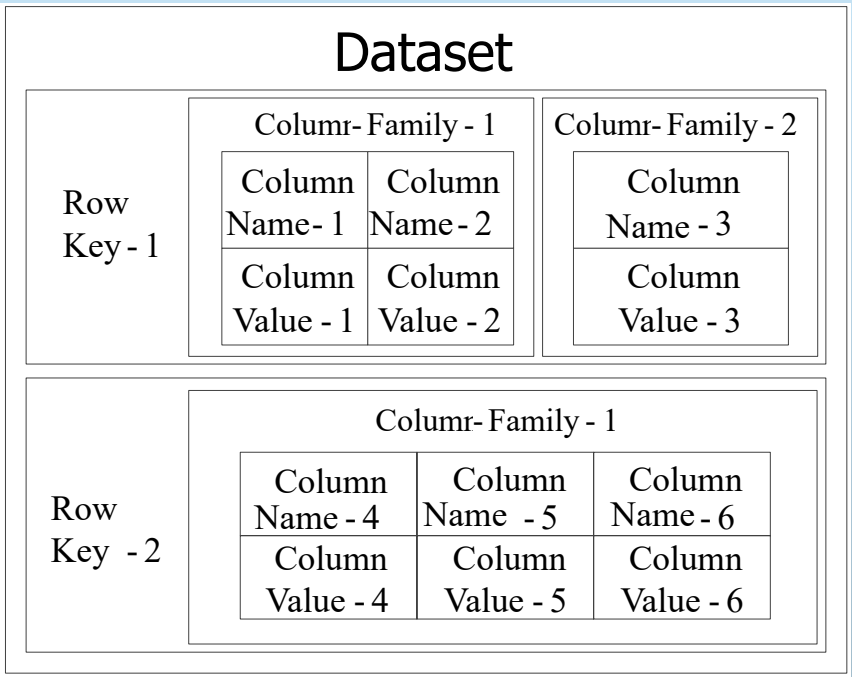
- **RDBMS**
  - 优势：以完善的关系代数理论作为基础，有严格的标准，支持事务ACID，提供严格的数据一致性，借助索引机制可以实现高效的查询，技术成熟，有专业公司的技术支持
  - 劣势：可扩展性较差，无法较好支持海量数据存储，采用固定的数据库模式，无法较好支持Web 2.0应用，事务机制影响系统的整体性能等
- **NoSQL**
  - 优势：可以支持超大规模数据存储，数据分布和复制容易，灵活的数据模型可以很好地支持Web 2.0应用，具有强大的横向扩展能力等
  - 劣势：缺乏数学理论基础，复杂查询性能不高，大都不能实现事务强一致性，很难实现数据完整性，技术尚不成熟，缺乏专业团队的技术支持，维护较困难，目前处于百花齐放的状态，用户难以选择（120+产品 listed in http://nosql-database.org ）等

二、NoSQL的主要类型

- 键值数据库、列存储数据库、文档数据库和图数据库

Key_1	Value _1
Key_2	Value _2
Key_3	Value _1
Key_4	Value _3
Key_5	Value _2
Key_6	Value _1
Key_7	Value _4
Key_8	Value _3

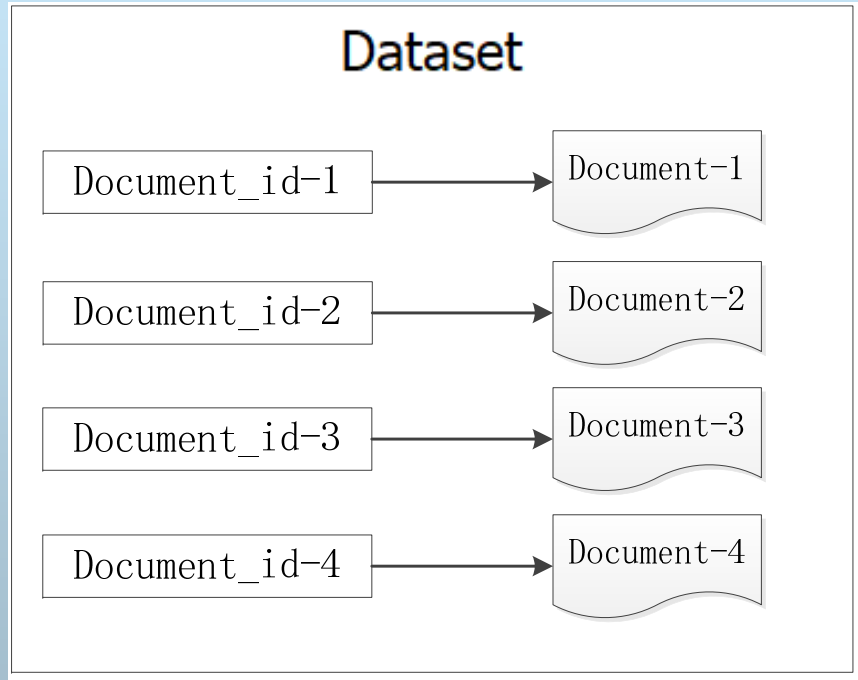
键值数据库



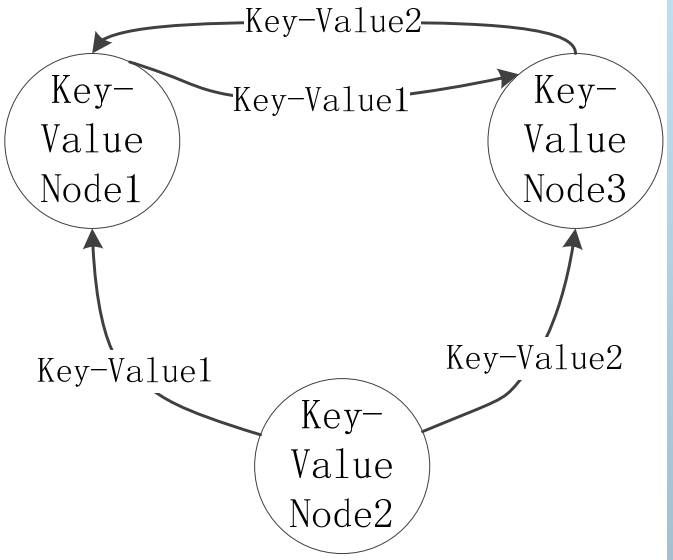
列存储数据库

二、NoSQL的主要类型

- 键值数据库、列存储数据库、文档数据库和图数据库

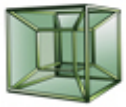


文档数据库



图数据库

二、NoSQL的主要类型

文档数据库	图数据库
 Couchbase MarkLogic mongoDB	 InfiniteGraph The Distributed Graph Database
键值数据库	**列存储数据库**
 amazon DynamoDB  levelDB riak	 accumulo HYPERTABLE INC  Cassandra APACHE HBASE Amazon SimpleDB

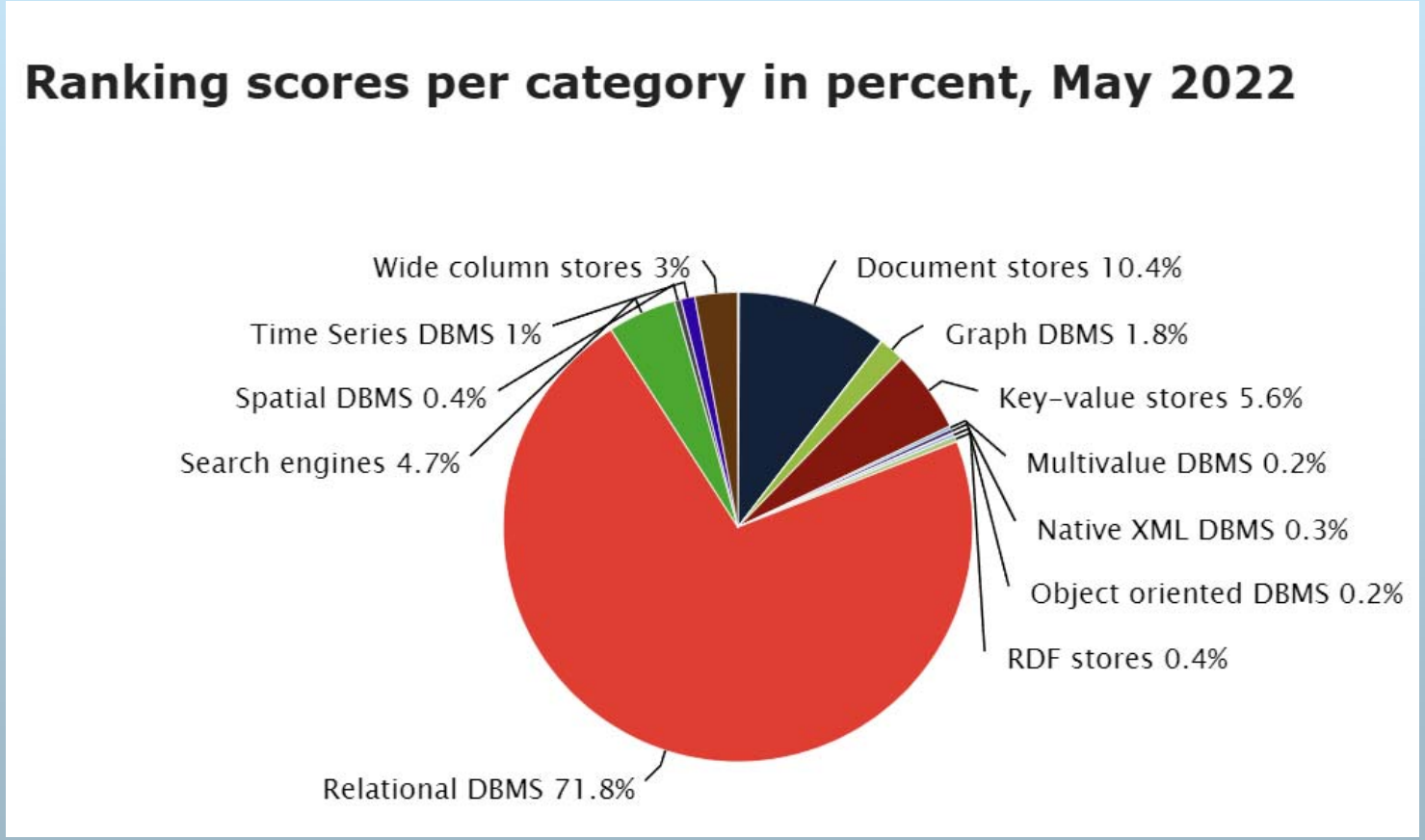
Overall Rank

394 systems in ranking, May 2022

Rank May 2022	Rank Apr 2022	Rank May 2021	DBMS	Database Model	Score May 2022	Score Apr 2022	Score May 2021
1.	1.	1.	Oracle	Relational, Multi-model	1262.82	+8.00	-7.12
2.	2.	2.	MySQL	Relational, Multi-model	1202.10	-2.06	-34.28
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	941.20	+2.74	-51.46
4.	4.	4.	PostgreSQL	Relational, Multi-model	615.29	+0.83	+56.04
5.	5.	5.	MongoDB	Document, Multi-model	478.24	-5.14	-2.78
6.	6.	↑ 7.	Redis	Key-value, Multi-model	179.02	+1.41	+16.85
7.	↑ 8.	↓ 6.	IBM Db2	Relational, Multi-model	160.32	-0.13	-6.34
8.	↓ 7.	8.	Elasticsearch	Search engine, Multi-model	157.69	-3.14	+2.34
9.	9.	↑ 10.	Microsoft Access	Relational	143.44	+0.66	+28.04
10.	10.	↓ 9.	SQLite	Relational	134.73	+1.94	+8.04
11.	11.	11.	Cassandra	Wide column	118.01	-3.98	+7.08
12.	12.	12.	MariaDB	Relational, Multi-model	111.13	+0.81	+14.44
13.	13.	13.	Splunk	Search engine	96.35	+1.11	+4.24
14.	14.	↑ 27.	Snowflake	Relational	93.51	+4.06	+63.46
15.	15.	15.	Microsoft Azure SQL Database	Relational, Multi-model	85.33	-0.45	+14.88
16.	16.	16.	Amazon DynamoDB	Multi-model	84.46	+1.55	+14.39
17.	17.	↓ 14.	Hive	Relational	81.61	+0.18	+5.42
18.	18.	↓ 17.	Teradata	Relational, Multi-model	68.39	+0.82	-1.59
19.	19.	19.	Neo4j	Graph	60.14	+0.62	+7.91
20.	20.	20.	Solr	Search engine, Multi-model	57.26	-0.48	+6.07

Overall Rank (cont.)

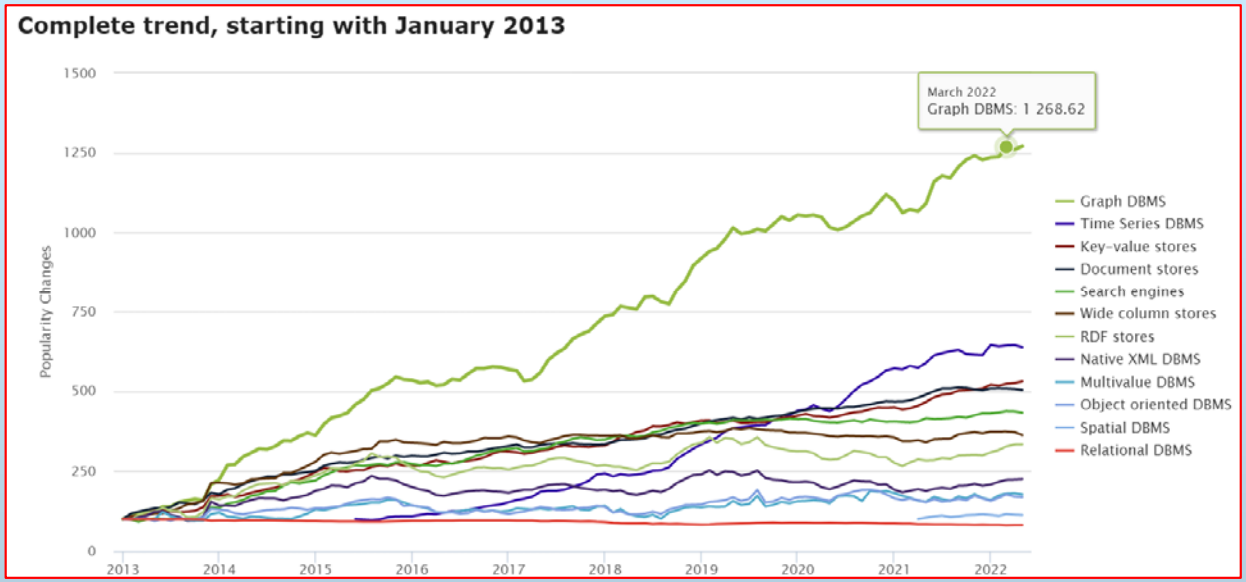
- 关系数据库仍是主流，但NoSQL比例在不断增长

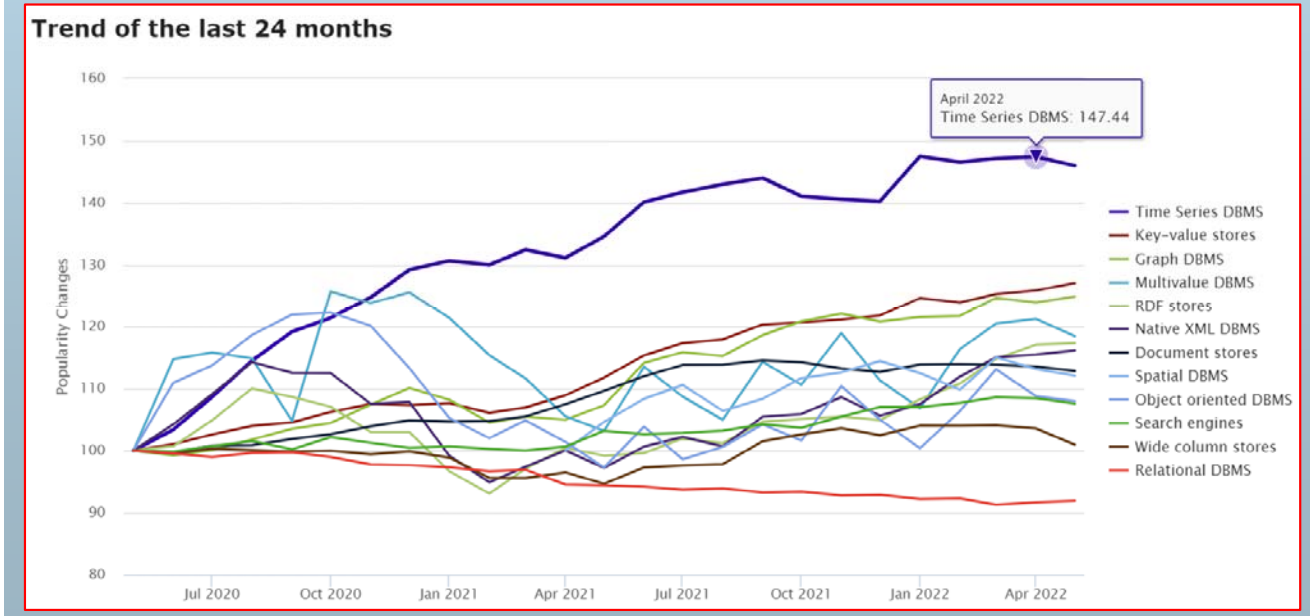


Overall Rank (cont.)

- 发展趋势

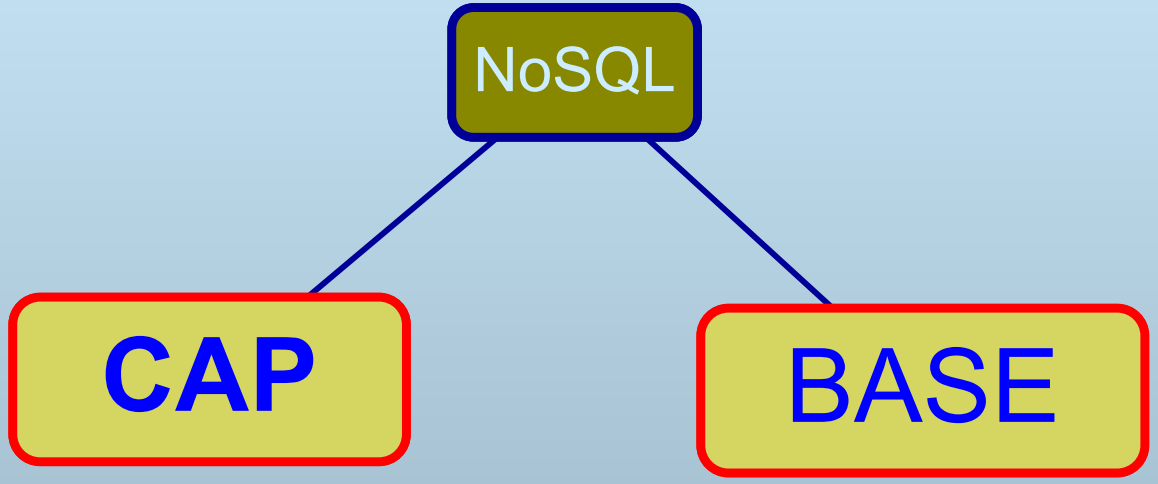
长期趋势上，图数据库
Graph DBMS一枝独秀





短期趋势上，时序数据库
Time-Series DBMS是热点

三、NoSQL的分布式系统基础



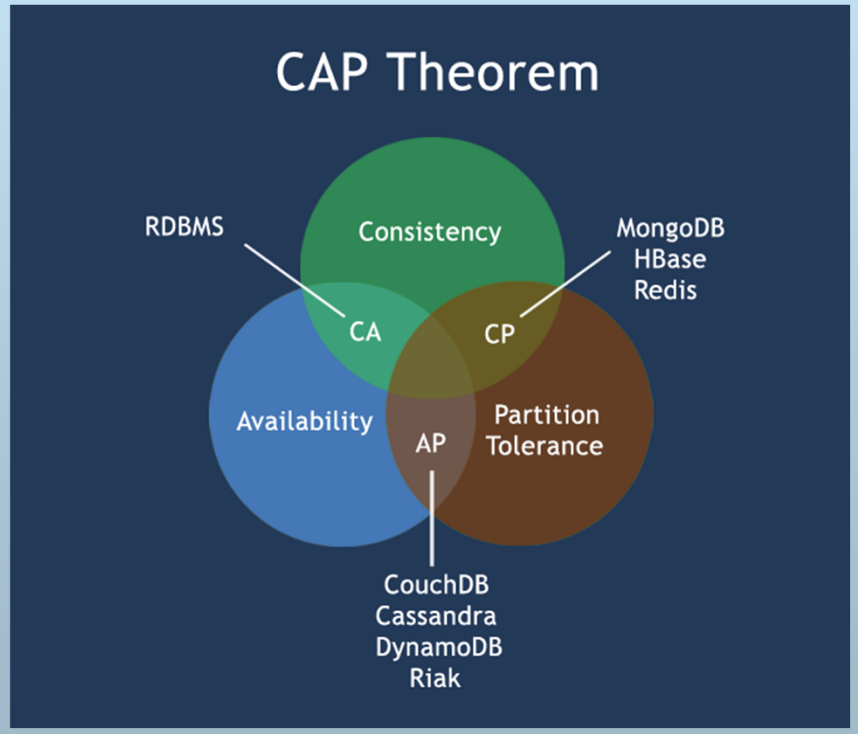
Note： 基本都是分布式系统中的技术，跟数据库系统关系不大

1、CAP

- **C（Consistency）：一致性——** ***all nodes see the same data at the same time***
  - 是指任何一个读操作总是能够读到之前完成的写操作的结果，也就是在分布式环境中，多点的数据是一致的，或者说，所有节点在同一时间具有相同的数据
- **A:（Availability）：可用性——*reads and writes always succeed***
  - 是指快速获取数据，可以在确定的时间内返回操作结果，保证每个请求不管成功或者失败都有响应
- **P（Tolerance of Network Partition）：分区容忍性——*the system continues to operate despite arbitrary message loss or failure of part of the system***
  - 是指当出现网络分区的情况时（即系统中的一部分节点无法和其他节点进行通信），分离的系统也能够正常运行，也就是说，系统中任意信息的丢失或失败不会影响系统的继续运作。

1、CAP

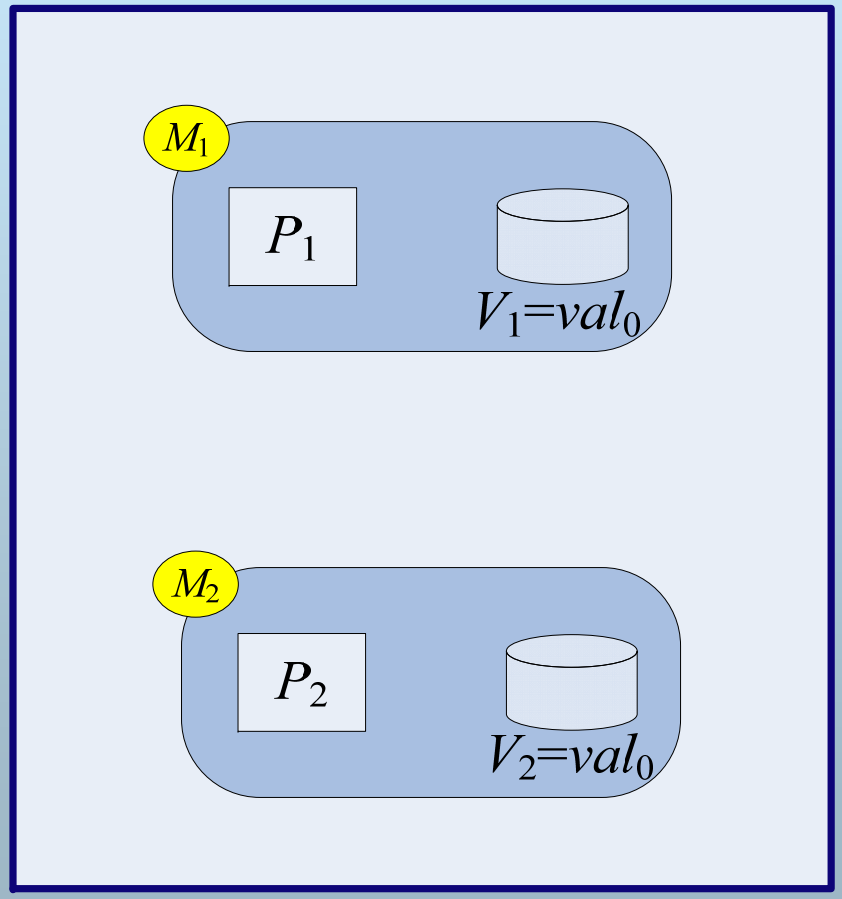
- **Brewer's Theorem (CAP Theorem):** 一个分布式系统不可能同时满足一致性、可用性和分区容忍性这三个需求，最多只能同时满足其中两个 **(Brewer, 2000; Gilbert, 2002)**



Brewer, Eric A. (2000): *Towards Robust Distributed Systems*. Keynote at the ACM Symposium on Principles of Distributed Computing (PODC).
Gilbert, S., & Lynch, N. (2002): *Brewers Conjunction and the Feasibility of Consistent, Available, Partition-Tolerant Web Services*. ACM SIGACT News, p. 33(2).

1、CAP

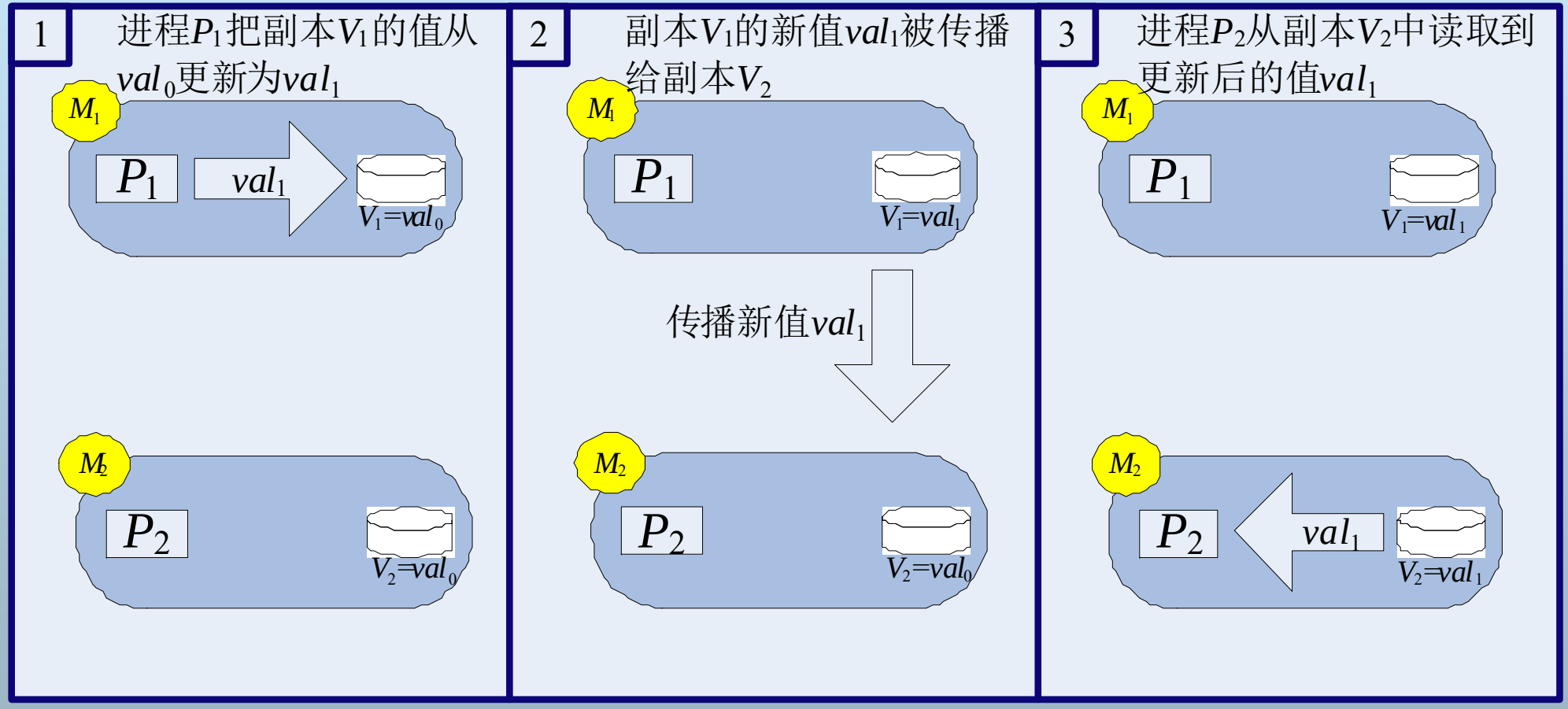
- 一个牺牲一致性来换取可用性的实例



（a）初始状态

1、CAP

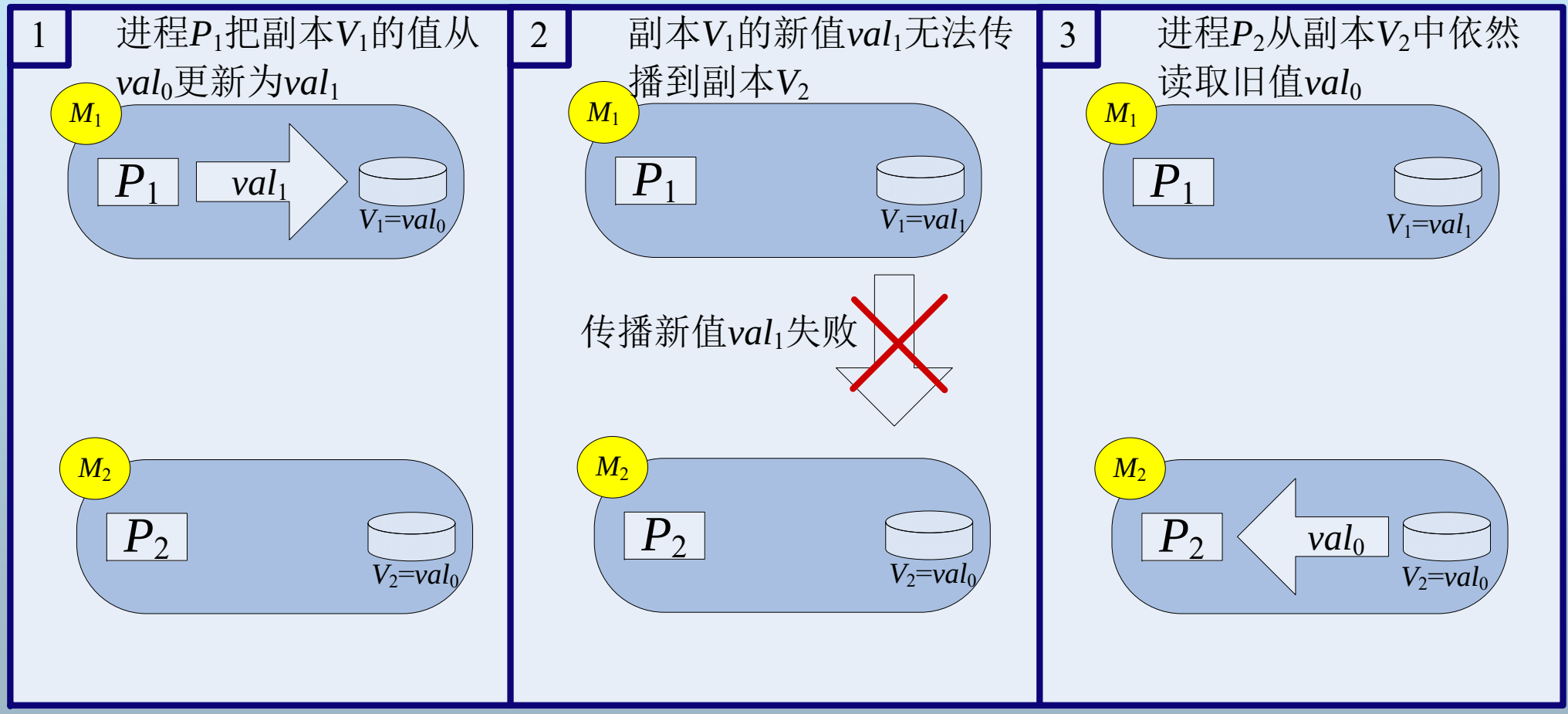
- 一个牺牲一致性来换取可用性的实例



（b）正常执行过程

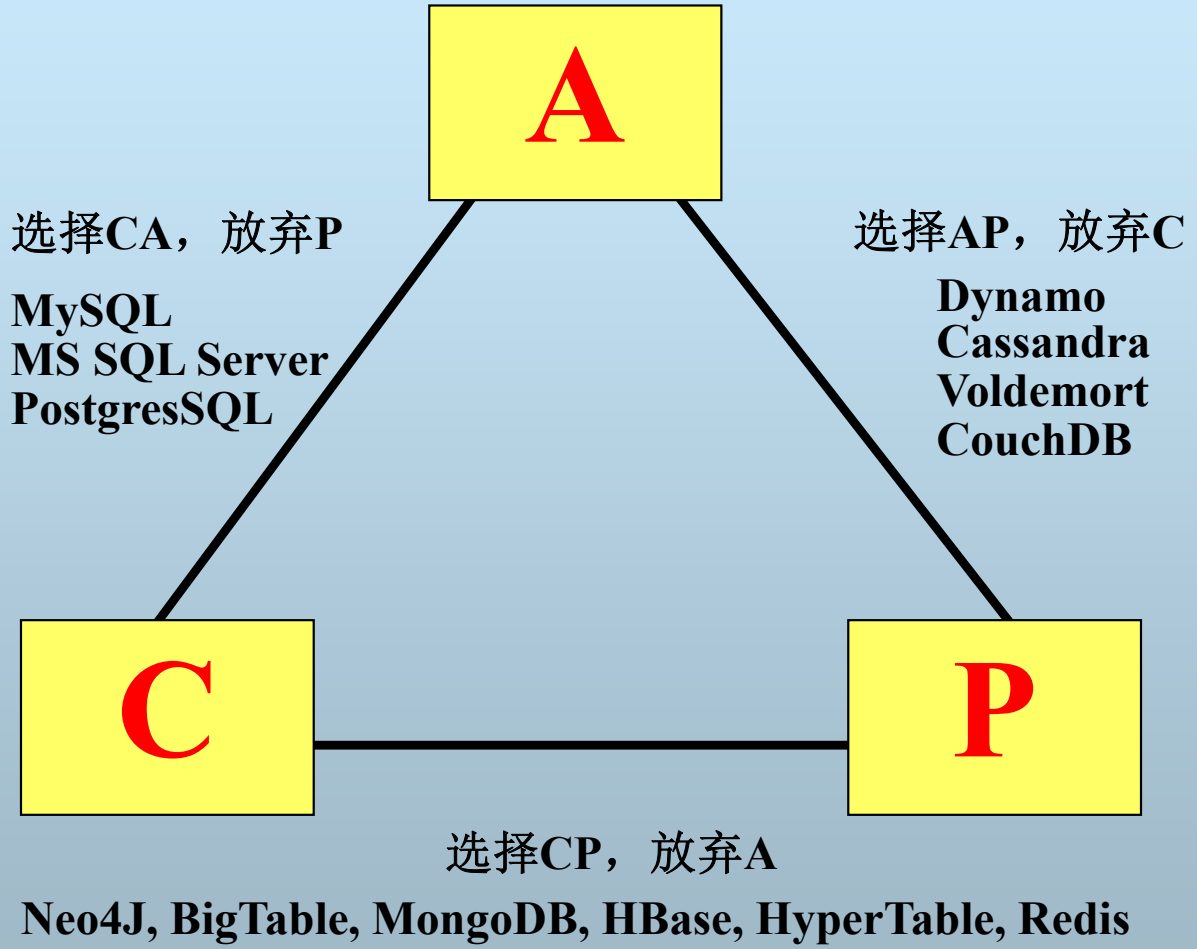
1、CAP

- 一个牺牲一致性来换取可用性的实例



(c) 更新传播失败时的执行过程

1、CAP



不同产品在CAP理论下的不同设计原则

2、BASE

- **BASE（Basically Available, Soft-state, Eventual consistency (Pritchett, 2008)**
  - **是对CAP理论的延伸**

ACID	BASE
原子性(**A**tomicity)	基本可用(**B**asically **A**vailable)
一致性(**C**onsistency)	软状态/柔性事务(**S**oft state)
隔离性(**I**solation)	最终一致性 (**E**ventual consistency)
持久性 (**D**urable)	

BASE vs. ACID

Dan Pritchett. (2008): *BASE: An ACID Alternative*. ACM *Queue*, Vol.6(3): 48-55

2、BASE

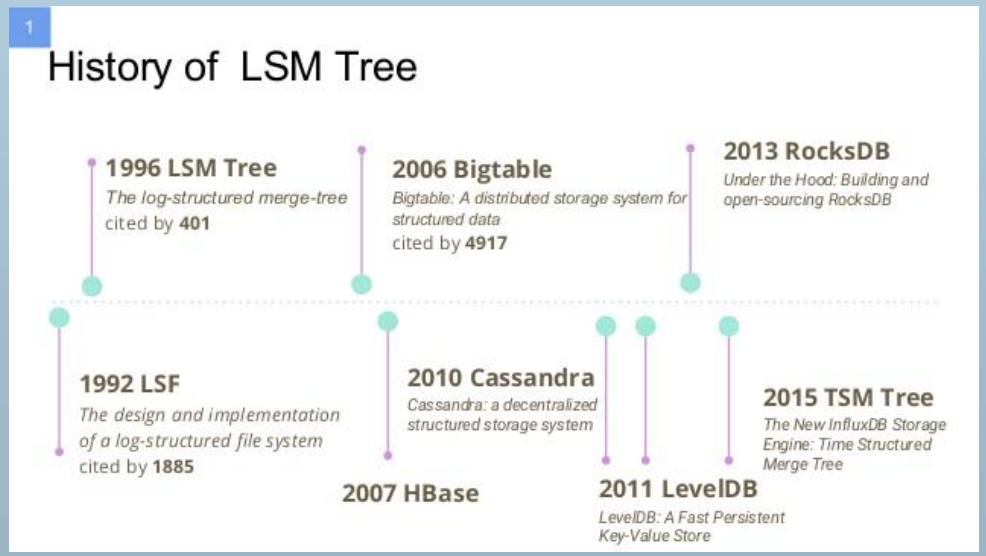
- **Basically Available**
  - 基本可用，是指一个分布式系统的一部分发生问题变得不可用时，其他部分仍然可以正常使用。也即允许损失部分可用性。
- **Soft-state**
  - “软状态（Soft-state）”是与“硬状态（Hard-state）”相对应的一种提法。数据库保存的数据是“硬状态”时，可以保证数据一致性，即保证数据一直是正确的。“软状态”是指状态可以有一段时间不同步，具有一定的滞后性

2、BASE

- **Eventual consistency**
  - 一致性的类型包括强一致性和弱一致性。对于强一致性而言，当执行完一次更新操作后，后续的其他读操作就可以保证读到更新后的最新数据。如果不能保证后续访问读到的都是更新后的最新数据，那么就是弱一致性。
  - 最终一致性是弱一致性的一种特例，允许后续的访问操作可以暂时读不到更新后的数据，但是经过一段时间之后，必须最终读到更新后的数据。
    - 最常见的实现最终一致性的系统是**DNS**（域名系统）。一个域名更新操作根据配置的形式被分发出去，并结合有过期机制的缓存；最终所有的客户端可以看到最新的值。

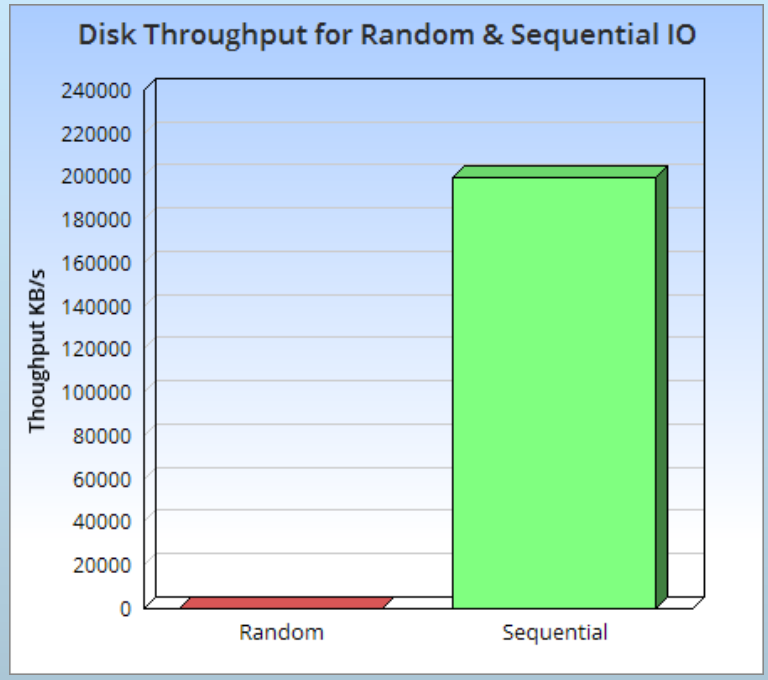
三、LSM-tree

- **LSM-tree是许多NoSQL数据库采用的存储引擎**
  - **1996年提出，借鉴了Log-Structured 文件系统的思想（1992）。**
  - **2006年，Google的BigTable采用LSM-tree作为存储引擎**
  - **被很多NoSQL数据库采用：HBase (2007)，Cassandra (2010), LevelDB (2011, Google)，RocksDB (2013, Facebook), InfluxDB(2015)等，以及国内的OceanBase、TiDB、PolarDB等等**



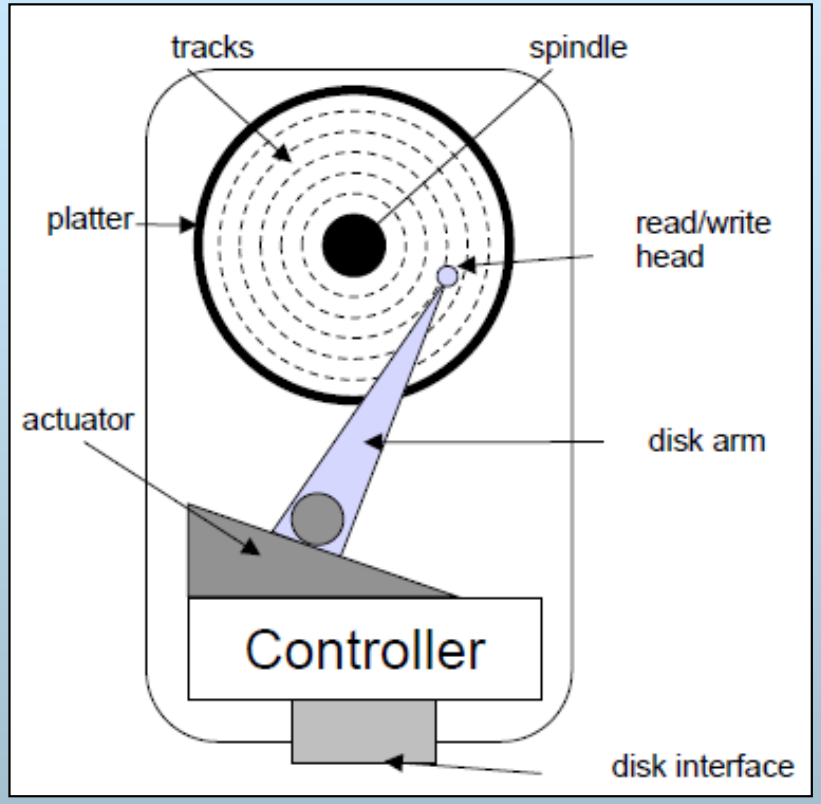
1、LSM-tree提出的背景

- **B+-tree**
  - 有序、平衡的多级存储结构
  - 面向磁盘设计，节点都是磁盘块
  - 查找性能好，适合读密集的负载
- **B+-tree的问题**
  - 原位更新（**In-Place Update**）
  - 写代价高，写性能差
    - 对叶节点的写基本都是随机写
    - 级联分裂、合并等**SMO**操作带来大量的随机写



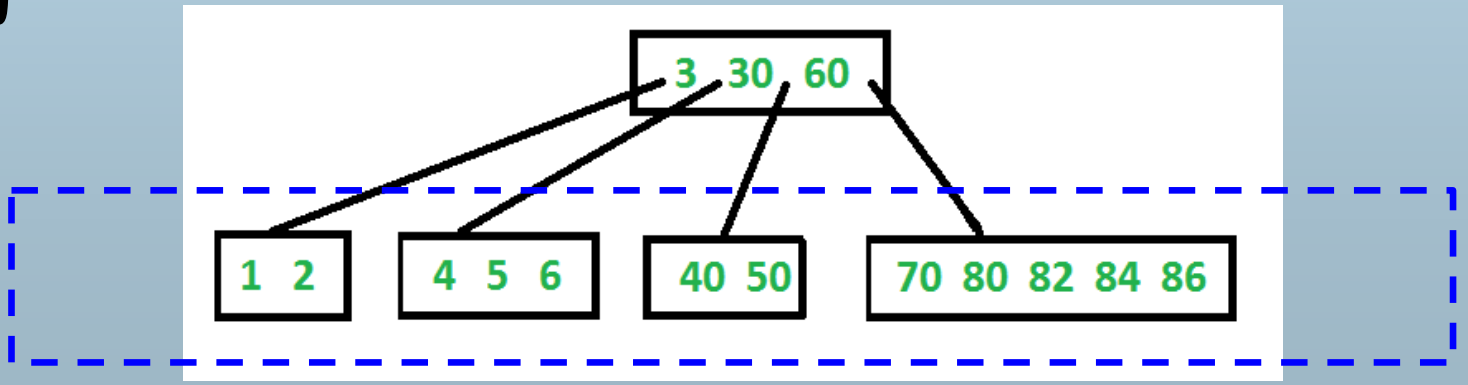
1、LSM-tree提出的背景

- 如何避免B+-tree随机写？
  - 采用log-structured的Append-only写
  - Log-structured update
    - 类似事务日志的写策略
    - 日志项不允许修改，只能Append
  - 写日志一般可视作是顺序写，写性能高



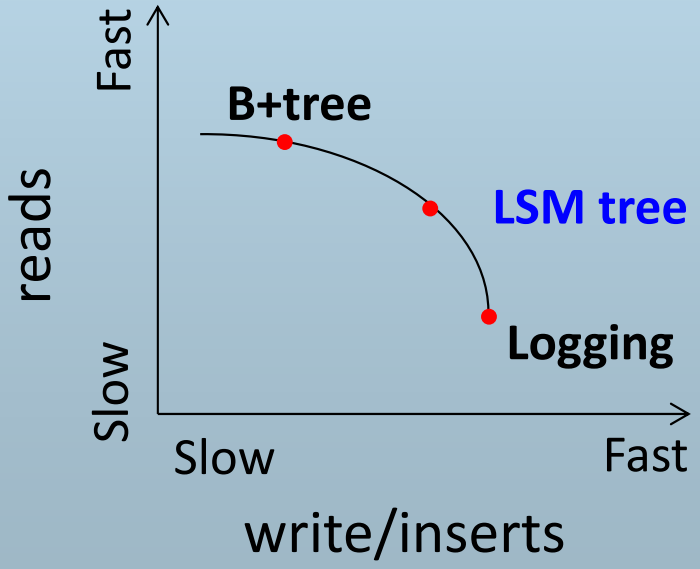
1、LSM-tree提出的背景

- 但是直接把B-tree的节点改成Log structured的Append-Only不能解决随机写的问题
  - 叶节点需要有序
    - 把新的键值Append到叶节点最后不行
  - 不同节点的更新依然是随机写
- 另一种思路：把所有叶节点的更新合并起来一起顺序写



2、LSM-tree的设计思路

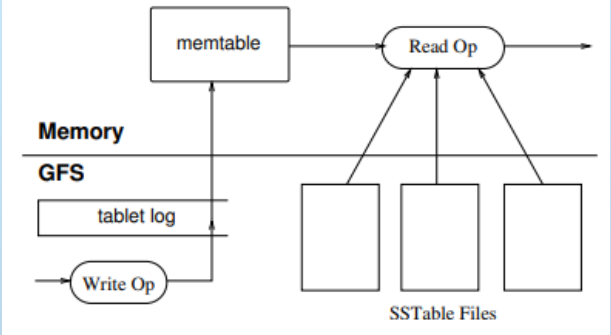
- **B-tree: 写慢读快**
- **Logging：写快读慢**
- **LSM-tree：先保证写快，同时读也较快**



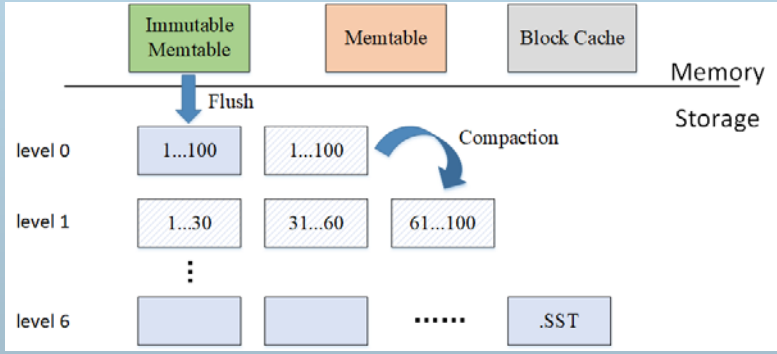
2、LSM-tree的设计思路

- **LSM (Log-Structured Merge) Tree**
  - 同时结合内存结构和磁盘结构（page-based）
    - 数据先写到内存结构，然后再写入磁盘
  - **Log-Structured**
    - 采用Append方式顺序写磁盘数据
  - **Merge Write**
    - 内存数据批量合并写入磁盘
    - 将多个小的随机写转换为顺序写
  - 数据分层写入磁盘
    - 避免一次批量写的数据量过大：内存压力过大、批量写时IO太多
  - 每一层的数据均有序

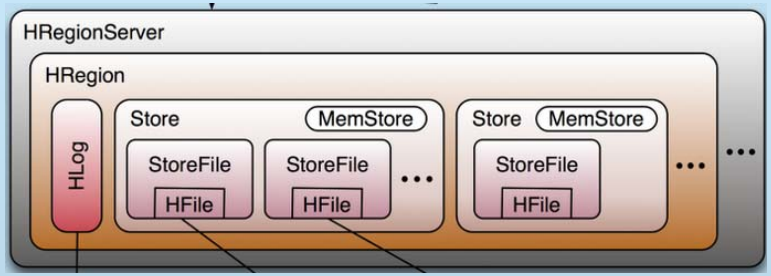
2、LSM-tree的设计思路



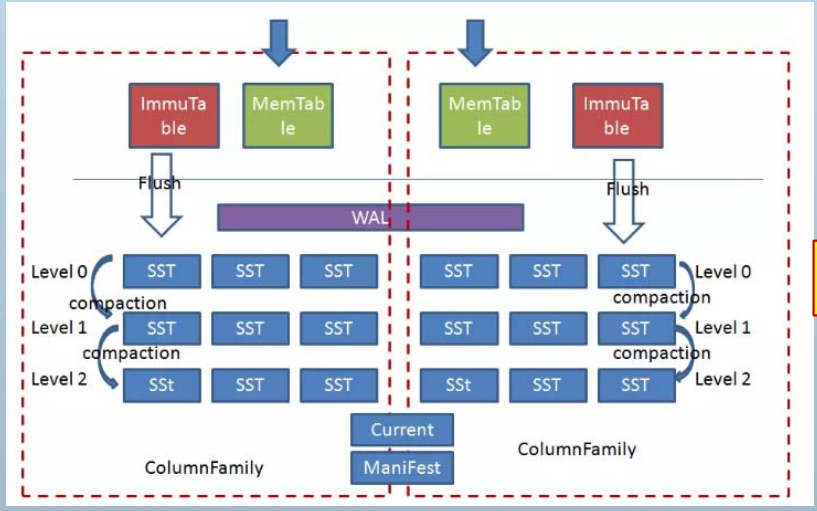
BigTable



LevelDB



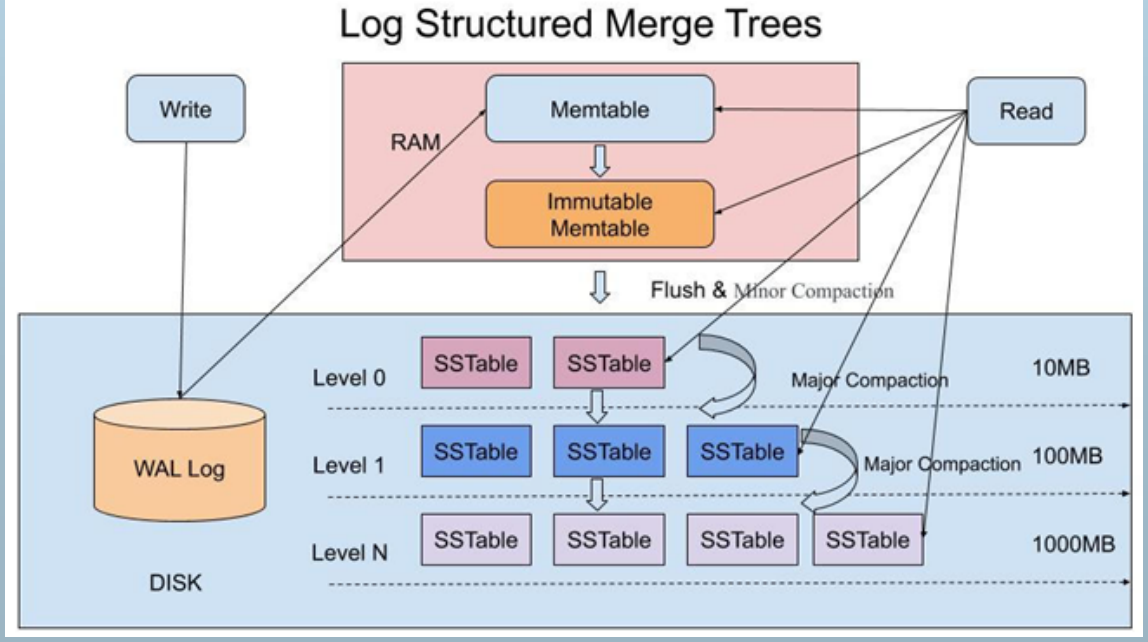
HBase

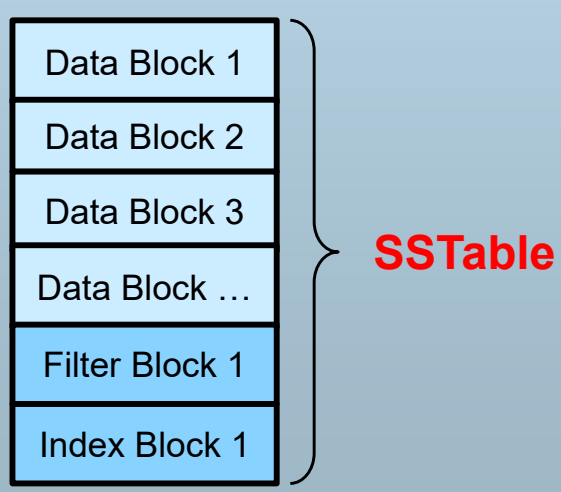


RocksDB

3、LSM-tree的实现

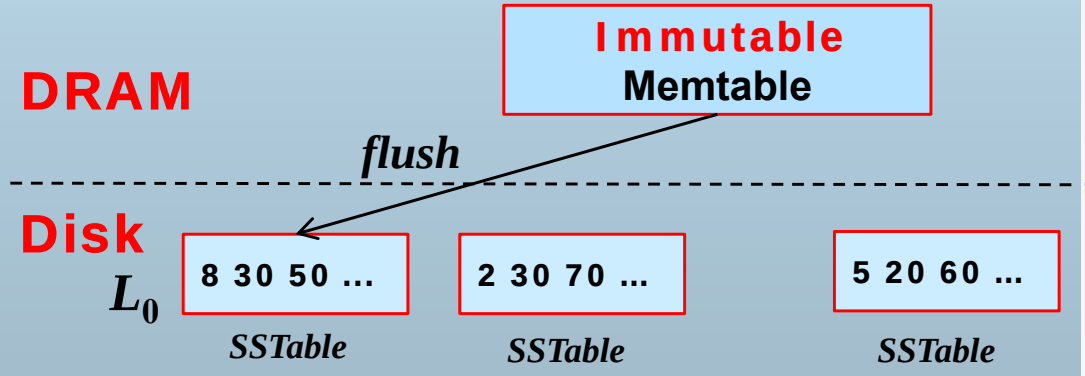
- 写数据的过程
  - 直接写入WAL和Memtable即返回，很快
  - 数据落盘：Flush & Compaction
- 读数据的过程
  - 先查DRAM结构，依次下沉查磁盘
  - 现有的一些读优化：Skiplist、Block Cache、Index Block、Bloom Filter

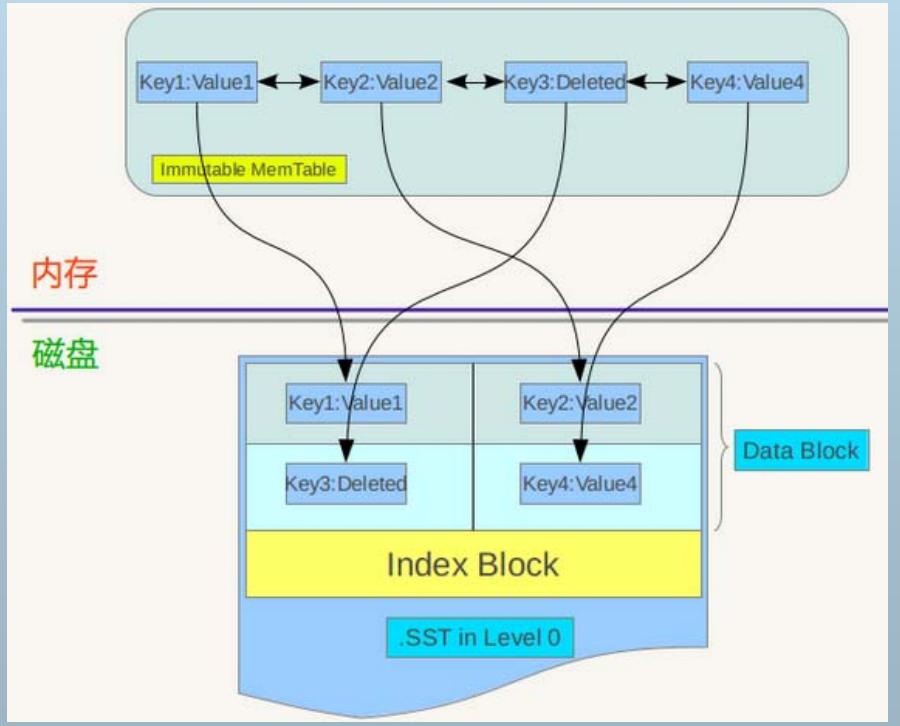




3、LSM-tree的实现

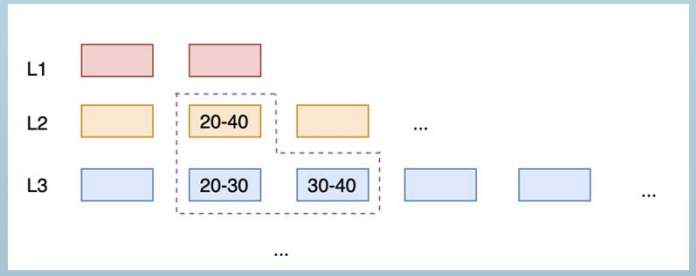
- **Compaction操作**
  - **Minor Compaction （Flush）**
    - **Immutable MemTable ---> SSTable (L0)**
    - **L0中的每个SSTable内部有序，但SSTable之间可能会存在重复的key，key的范围也可能overlap**





3、LSM-tree的实现

- **Compaction操作**
  - **Major Compaction**
    - **L0 --> L1， …， SSTable (L_i) --> SSTable (L_{i+1})**
    - **当Li层的score超过阈值时（基于文件数和大小计算）**
    - **Compaction时执行垃圾回收，抛弃掉已经被删除的KV（物理的删除操作只有在Major Compaction时才会执行），减小SSTable数量和大小**
  - **Compaction (Merge) 过程**
    - **选择Li层的一个SSTable文件**
    - **选择Li+1层中所有与Li层的SSTable的key有重叠的所有文件**
    - **执行merge sort：读入DRAM，merge，write out to SSTs**



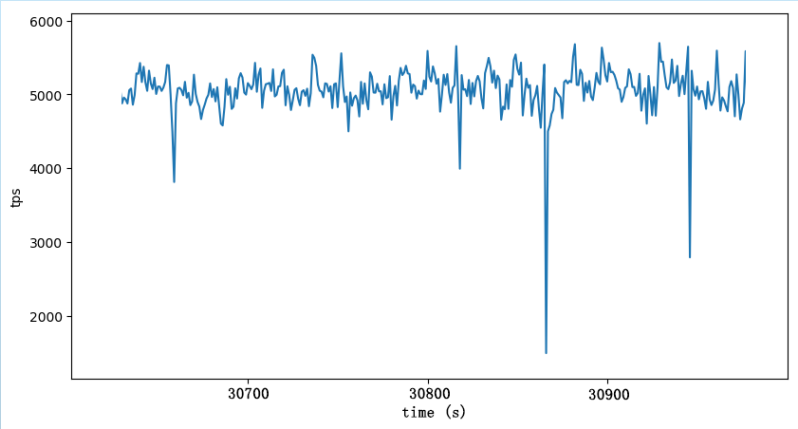
  - **不同的Major Compaction策略**
    - **Leveled Compaction (one sorted run at each level)**
    - **Tiered Compaction (multiple sorted overlapping runs at each level)**

4、LSM-tree的总结

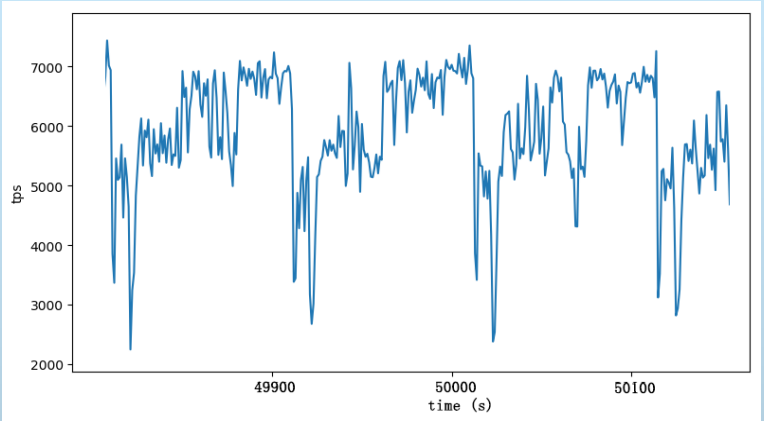
- 优点
  - 把随机写操作转化为顺序写，支持高吞吐的写（尤其适合分布式大数据应用场景）
  - 采用Append方式写数据，读写操作相互独立，可以支持高并发应用
  - 适合写多读少的应用
- 缺点
  - 读性能较差
  - 空间放大严重，需要Compaction才能回收空间
  - Compaction操作导致系统性能抖动
    - 系统资源消耗高
      - I/O代价（写放大、I/O带宽消耗）
      - CPU和内存消耗
    - Block Cache失效

Compaction对系统性能的影响

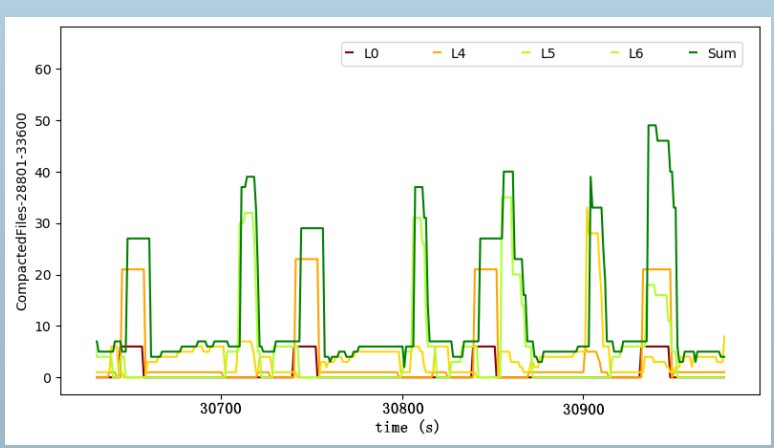
TPC-C@sysbench

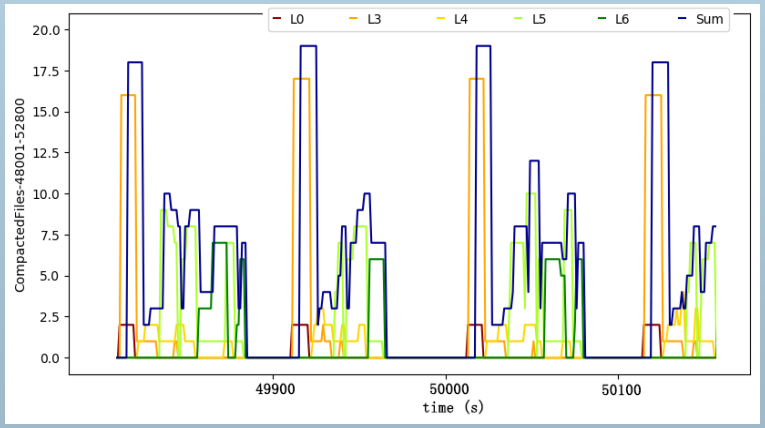


OLTP@sysbench



Throughput



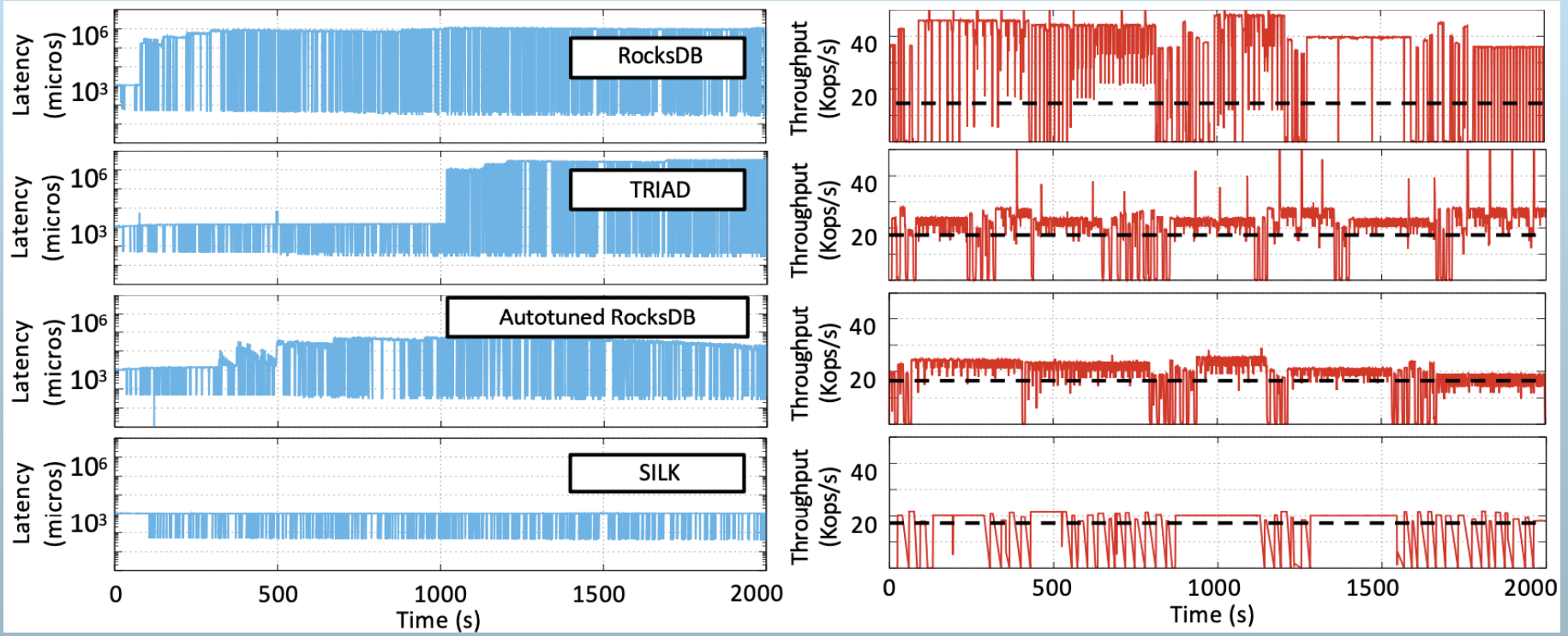


正在Compact的SSTable文件数

针对LSM-tree的一些优化方向

最新的一些优化工作

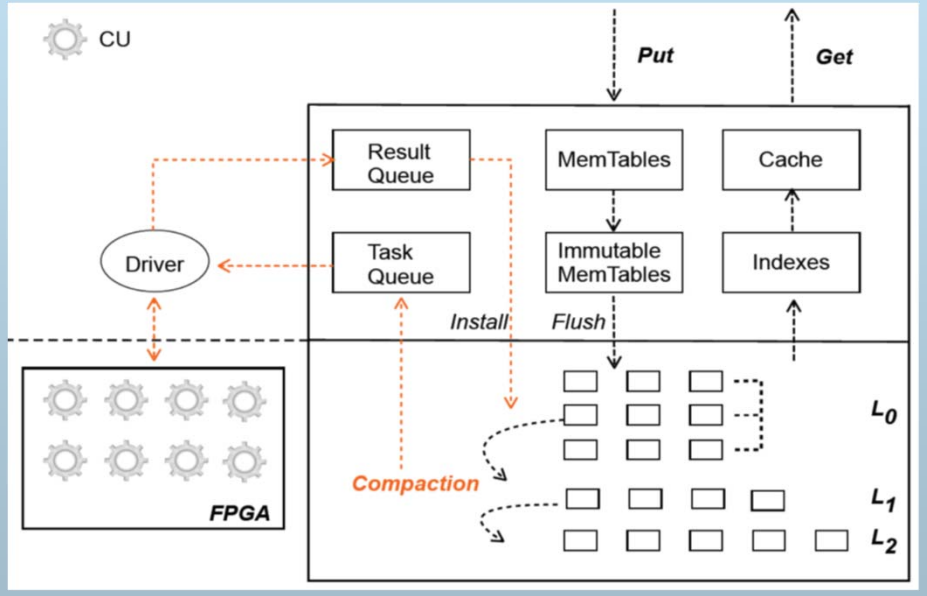
- **SILK （ATC 2019, Best Paper）**
  - 针对周期性有峰值的负载，高负载时延迟下层合并但继续上层合并，低负载时再执行下层合并

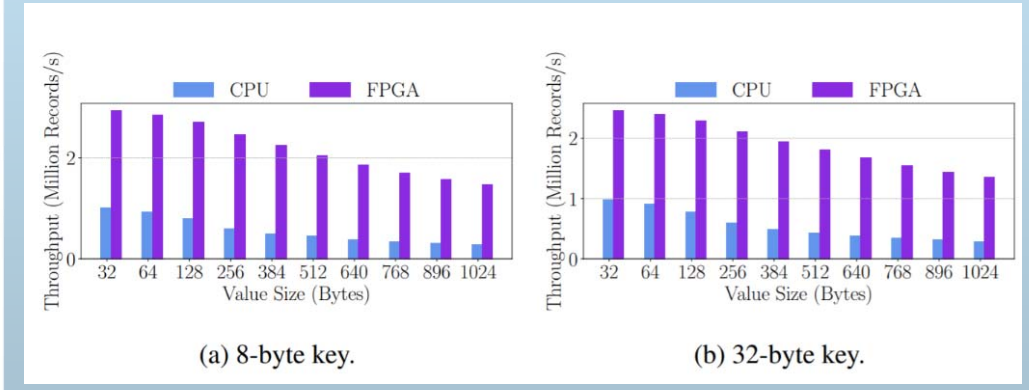


Oana Balmau et al. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. ATC 2019

最新的一些优化工作

- **FPGA-Accelerated Compaction (FAST'20)**
  - 理论上有助于平滑LSM-tree性能的抖动
  - 需要专用的FGPA及驱动支持

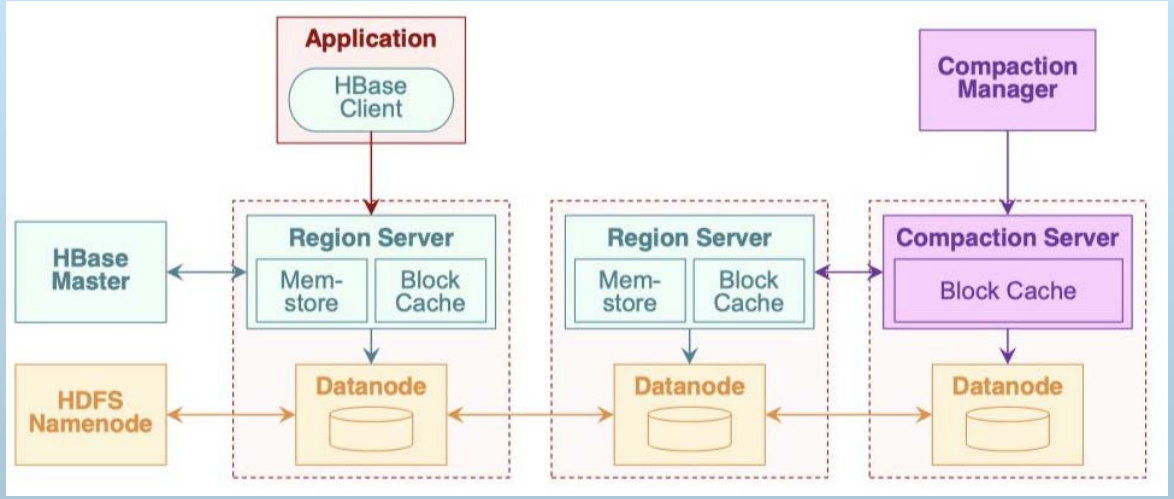


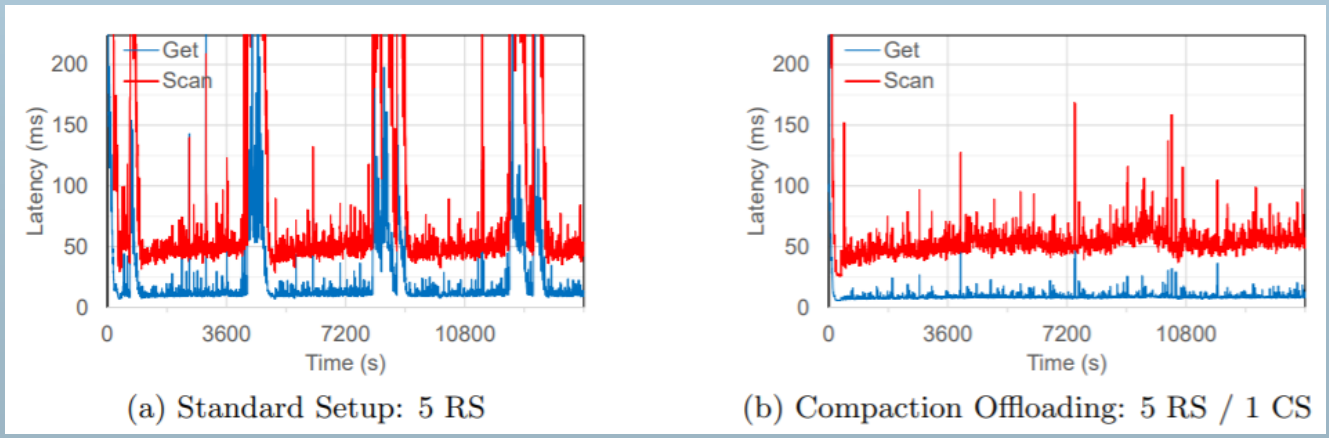


Teng Zhang et al. FPGA-Accelerated Compactions for LSM-based Key-Value Store. FAST 2020

最新的一些优化工作

- **Offload Compaction (VLDB'15)**
  - 借助独立的Compaction Server来完成合并，从而不消耗本地Server的IO和CPU资源

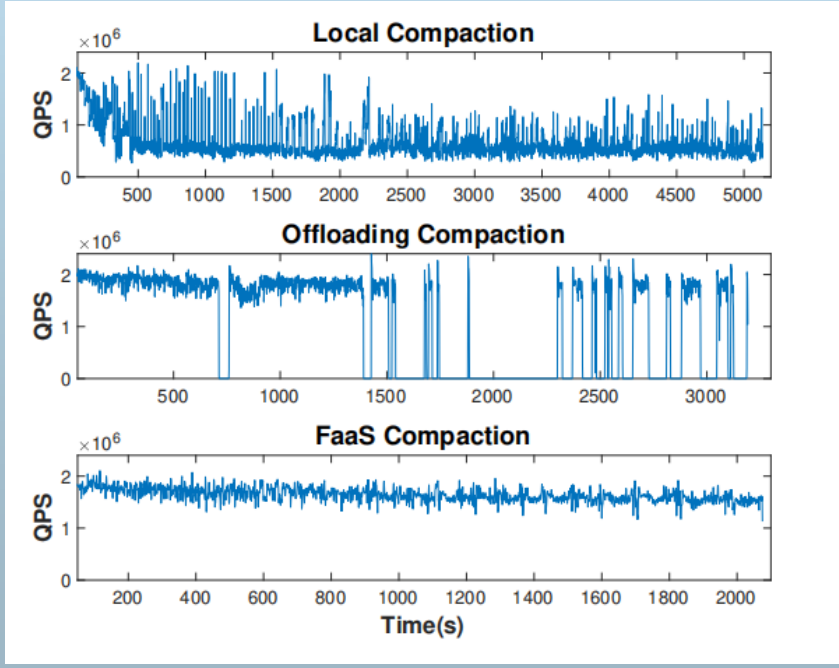




Muhammad Yousuf Ahmad et al. Compaction management in distributed key-value datastores. VLDB 2015

最新的一些优化工作

- **FaaS Compaction (CIKM'21)**
  - **FaaS: Function as a Service**
    - 函数即服务，新型Elastic Computing
    - 根据需求自动扩/缩容，按需计算成本
    - 2014年提出，大厂迅速跟进
      - Amazon AWS Lamda (2014)
      - Google Cloud (2016)
      - 阿里、腾讯、字节 (2017-2020)
  - 思路
    - 当需要执行Compaction时，将任务发送给FaaS实例执行
    - 从而不占用本地服务器的资源，保证系统性能的稳定性



Jianchuan Li et al Elastic and Stable Compaction for LSM-tree: A FaaS-Based Approach on TerarkDB. CIKM 2021

本章小结

- NoSQL概念
- NoSQL的类型
- CAP & BASE
- LSM-tree