# NoSQL Databases I: Introduction



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
- 从NoSQL到NewSQL

# 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

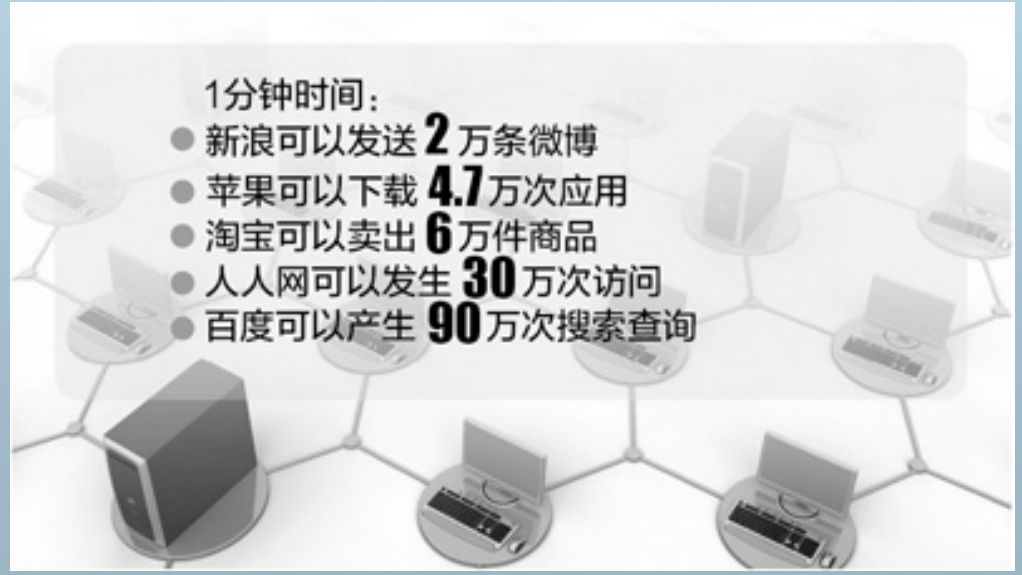
# 一、NoSQL简介

- 现在已有很多公司使用了NoSQL数据库：
  - Google
  - Facebook
  - Adobe
  - Foursquare
  - LinkedIn
  - 百度、腾讯、阿里、新浪、华为……

# 1、NoSQL兴起的原因

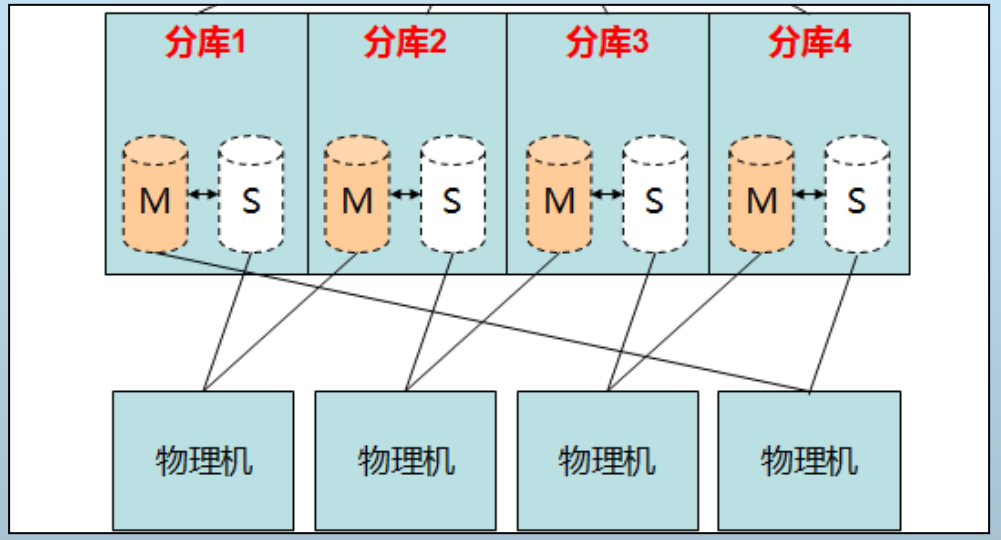
## （1）RDBMS无法满足Web 2.0的需求：

- 无法满足海量数据的管理需求
- 无法满足数据高并发的需求
- 无法满足高可扩展性和高可用性的需求



# MySQL集群能否解决问题?

- **复杂性：**部署、管理、配置复杂
- **数据库复制：**MySQL主备之间采用异步复制方式，当主库压力较大时将产生较大延迟，主备切换可能会丢失最后一部分更新事务，往往需要人工介入，备份和恢复不方便
- **扩容问题：**如果系统压力过大需要增加新的机器，这个过程涉及数据重新划分，整个过程比较复杂，且容易出错
- **动态数据迁移问题：**如果某个数据库组压力过大，需要将其中部分数据迁移出去，迁移过程需要总控节点整体协调，以及数据库节点的配合。这个过程很难做到自动化



# 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等是针对在线业务，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实时性
- 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

| 比较标准 | RDBMS | NoSQL | 备注 |
| --- | --- | --- | --- |
| **数据库原理** | 完全支持 | 部分支持 | • RDBMS有关系代数理论作为基础<br>• NoSQL没有统一的理论基础 |
| **数据规模** | 大 | 超大 | • RDBMS很难实现横向扩展，纵向扩展的空间也有限，性能会随着数据规模的增大而降低<br>• NoSQL可以很容易通过添加更多设备来支持更大规模的数据 |
| **数据库模式** | 固定 | 灵活 | • RDBMS需要定义数据库模式，严格遵守数据定义和相关约束条件<br>• NoSQL不存在数据库模式，可以自由灵活定义并存储各种不同类型的数据 |
| **查询效率** | 快 | 可以实现高效的简单查询，但是不具备高度结构化查询等特性，复杂查询的性能不尽人意 | • RDBMS借助于索引机制可以实现快速查询（包括记录查询和范围查询）<br>• 很多NoSQL数据库没有面向复杂查询的索引，虽然NoSQL可以使用MapReduce来加速查询，但是，在复杂查询方面的性能仍然不如RDBMS |

# 2、NoSQL vs. RDBMS (cont.)

| 比较标准 | RDBMS | NoSQL | 备注 |
| --- | --- | --- | --- |
| 一致性 | 强一致性 | 弱一致性 | • RDBMS严格遵守事务ACID模型，可以保证事务强一致性<br>• NoSQL数据库放松了对事务ACID的要求，而是遵守BASE模型，只能保证最终一致性 |
| 数据完整性 | 容易实现 | 很难实现 | • RDBMS可以很容易实现数据完整性，比如通过主键来实现实体完整性，通过外键来实现参照完整性，通过check约束或者触发器来实现用户自定义完整性<br>• 但是，在NoSQL数据库却无法实现 |
| 扩展性 | 一般 | 好 | • RDBMS很难实现横向扩展，纵向扩展的空间也比较有限<br>• NoSQL在设计之初就考虑了横向扩展的需求，可以很容易通过添加廉价设备实现扩展 |
| 可用性 | 好 | 很好 | • RDBMS为了保证严格的数据一致性，只能提供相对较弱的可用性<br>• 大多数NoSQL都能提供较高的可用性 |

# 2、NoSQL vs. RDBMS (cont.)

| 比较标准 | RDBMS | NoSQL | 备注 |
| --- | --- | --- | --- |
| **标准化** | 是 | 否 | • RDBMS已经标准化（SQL）<br>• NoSQL还没有行业标准，不同的NoSQL数据库都有自己的查询语言和应用程序接口。NoSQL缺乏统一查询语言，将会拖慢NoSQL发展 |
| **技术支持** | 高 | 低 | • RDBMS经过几十年的发展，已经非常成熟，Oracle等大型厂商都可以提供很好的技术支持<br>• NoSQL在技术支持方面仍然处于起步阶段，还不成熟，缺乏有力的技术支持 |
| **可维护性** | 复杂 | 复杂 | • RDBMS需要专门的数据库管理员(DBA)维护<br>• NoSQL数据库虽然没有RDBMS复杂，也难以维护 |

# 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

# 2、NoSQL vs. RDBMS

- RDBMS和NoSQL各有优缺点，彼此无法取代
- RDBMS应用场景
  - 电信、银行等领域的关键业务系统，需要保证强事务一致性
- NoSQL数据库应用场景
  - 互联网企业、传统企业的非关键业务（比如数据分析）
- 采用混合架构
  - 案例：亚马逊公司就使用不同类型的数据库来支撑它的电子商务应用
    - 对于“购物篮”这种临时性数据，采用键值存储会更加高效
    - 当前的产品和订单信息则适合存放在关系数据库中
    - 大量的历史订单信息则适合保存在类似MongoDB的文档数据库中

# 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 |

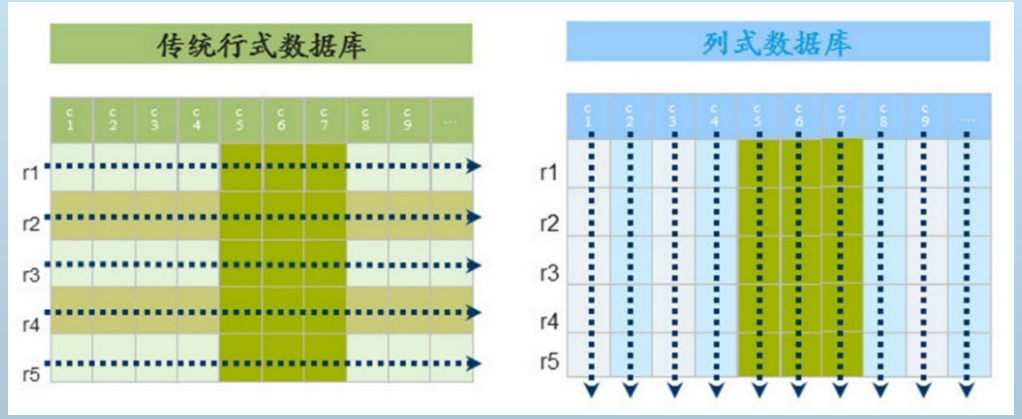
键值数据库

Dataset

| Row Key-1 | Colum-Family-1 | | Colum-Family-2 |
|---|---|---|---|
| | Column Name-1 | Column Name-2 | Column Name-3 |
| | Column Value-1 | Column Value-2 | Column Value-3 |

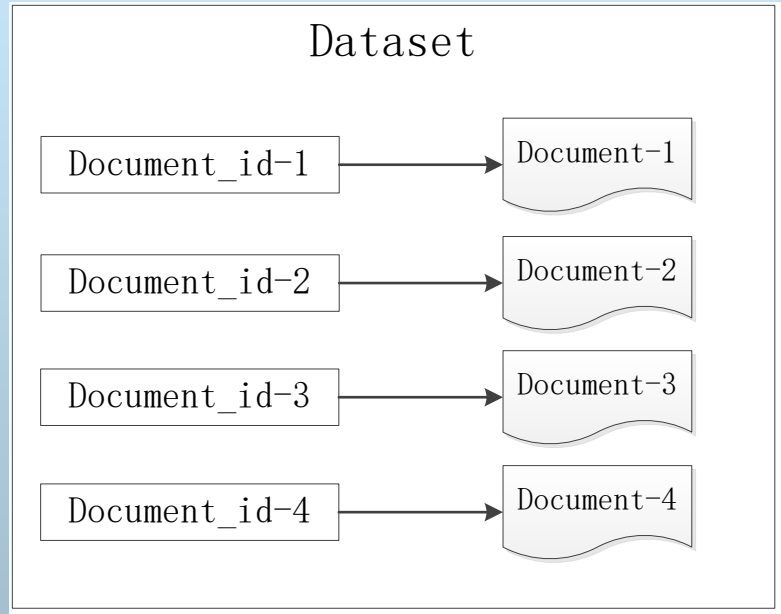
| Row Key-2 | Colum-Family-1 | | |
|---|---|---|---|
| | Column Name-4 | Column Name-5 | Column Name-6 |
| | Column Value-4 | Column Value-5 | Column Value-6 |

列存储数据库

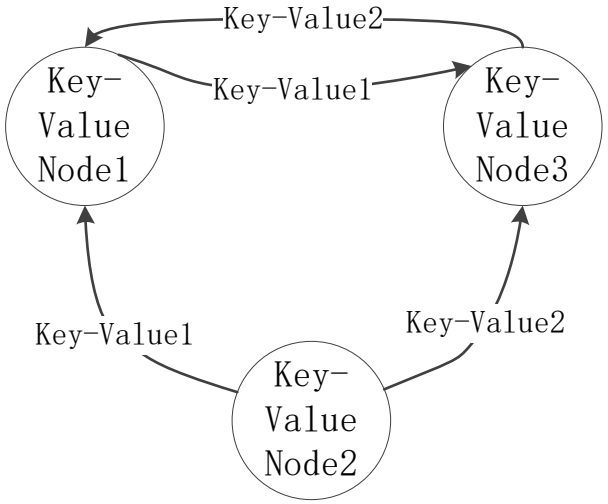


# 二、NoSQL的主要类型

- 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

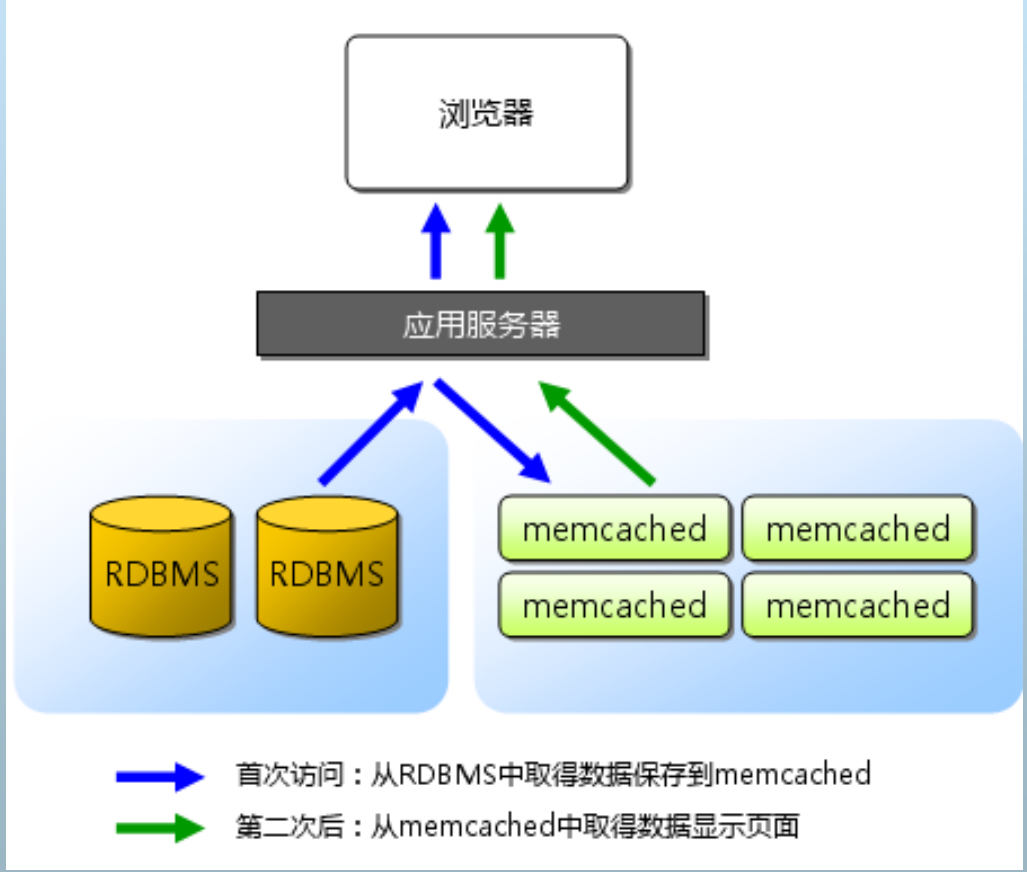
| 文档数据库 | 图数据库 |
| --- | --- |
|  Couchbase MarkLogic mongoDB | Neo4j InfiniteGraph The Distributed Graph Database |
| **键值数据库** | **列存储数据库** |
| redis amazon DynamoDB AEROSPIKE riak |  accumulo HYPERTABLE INC Cassandra APACHE HBASE Amazon SimpleDB |

# 1、键值数据库 （Key-Value Store）

| 相关产品 | Redis、Riak、SimpleDB、Chordless、Scalaris、Memcached |
|---|---|
| 数据模型 | • 键/值对<br>• 键是一个字符串对象<br>• 值可以是任意类型的数据，比如整型、字符型、数组、列表、集合等 |
| 典型应用 | • 涉及频繁读写、拥有简单数据模型的应用<br>• 内容缓存，比如会话、配置文件、参数、购物车等<br>• 存储配置和用户数据信息的移动应用 |
| 优点 | 扩展性好，灵活性好，大量写操作时性能高 |
| 缺点 | 无法存储结构化信息，条件查询效率较低 |
| 使用者 | 百度云数据库（Redis）、GitHub（Riak）、BestBuy（Riak）、Twitter（Redis和Memcached）、StackOverFlow（Redis）、Youtube（Memcached）、Wikipedia（Memcached） |
| 不适用情形 | • 不是通过键而是通过值来查：键值数据库根本没有通过值查询的途径<br>• 需要存储数据之间的关系：在键值数据库中，不能通过两个或两个以上的键来关联数据<br>• 需要事务的支持：在一些键值数据库中，产生故障时，不可以回滚 |

# 1、键值数据库 （Key-Value Store）

- 键值数据库成为理想的缓冲层解决方案
- 支持持久化、数据恢复、更多数据类型



# 2、列存储数据库（Column Store）

| 相关产品 | BigTable、HBase、Cassandra、HadoopDB、GreenPlum、PNUTS |
|---|---|
| 数据模型 | 列族 （Column Family） |
| 典型应用 | • 分布式数据存储与管理<br>• 数据在地理上分布于多个数据中心的应用程序<br>• 可以容忍副本中存在短期不一致情况的应用程序<br>• 拥有动态字段的应用程序<br>• 拥有潜在大量数据的应用程序，大到几百TB的数据 |
| 优点 | 查找速度快，可扩展性强，容易进行分布式扩展，复杂性低 |
| 缺点 | 功能较少，大都不支持强事务一致性 |
| 使用者 | EBay（Cassandra）、Instagram（Cassandra）、NASA（Cassandra）、Twitter（Cassandra and HBase）、Facebook（HBase）、Yahoo!（HBase） |
| 不适用情形 | 需要ACID事务支持的情形，Cassandra等产品就不适用 |

# 3、文档数据库（Document Store）

| | |
|---|---|
| 相关产品 | MongoDB、CouchDB、Terrastore、ThruDB、RavenDB、SisoDB、RaptorDB、CloudKit、Perservere、Jackrabbit、**SequoiaDB** |
| 数据模型 | • 键/值<br>• 值（value）是版本化的文档 |
| 典型应用 | • 存储、索引并管理面向文档的数据或者类似的半结构化数据<br>• 比如，用于后台具有大量读写操作的网站、使用JSON数据结构的应用、使用嵌套结构等非规范化数据的应用程序 |
| 优点 | • 性能好（高并发），灵活性高，复杂性低，数据结构灵活<br>• 提供嵌入式文档功能，将经常查询的数据存储在同一个文档中<br>• 既可以根据键来构建索引，也可以根据内容构建索引 |
| 缺点 | 缺乏统一的查询语法 |
| 使用者 | 百度云数据库（MongoDB）、SAP （MongoDB）、Codecademy（MongoDB）、Foursquare （MongoDB）、NBC News （RavenDB） |
| 不适用情形 | 在不同的文档上添加事务。文档数据库并不支持文档间的事务，如果对这方面有需求则不应该选用这个解决方案 |

# 3、文档数据库（Document Store）

“文档”其实是一个数据记录，这个记录能够对包含的数据类型和内容进行“自我描述”。XML文档和JSON文档就属于这一类。SequoiaDB(巨杉)就是使用JSON格式的文档数据库，它的存储的数据是这样的：

```
{
“ID”:1,
“NAME”: “SequoiaDB”,
“Tel”: {
        “Office”: “123123”, “Mobile”: “132132132”
      }
“Addr”: “China,GZ”
}
```

# 3、文档数据库（Document Store）

```
{
“ID”:1,
“NAME”: “SequoiaDB”,
“Tel”: {
        “Office”: “123123”, “Mobile”: “132132132”
      }
“Addr”: “China,GZ”
}
```

- 数据是不规则的，每一条记录包含了所有的有关“SequoiaDB”的信息而没有任何外部的引用，这条记录就是“自包含”的
- 这使得记录很容易完全移动到其他服务器，因为这条记录的所有信息都包含在里面了，不需要考虑还有信息在别的表没有一起迁移走
- 同时，因为在移动过程中，只有被移动的那一条记录（文档）需要操作，而不像关系型中每个有关联的表都需要锁住来保证一致性

# 4、图数据库（Graph Store）

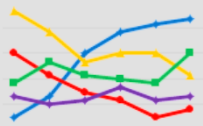
| 相关产品 | Neo4J、OrientDB、InfoGrid、InfiniteGraph、GraphDB |
| --- | --- |
| 数据模型 | 图结构 |
| 典型应用 | 专门用于处理具有高度相互关联关系的数据，比较适合于社交网络、模式识别、依赖分析、推荐系统以及路径寻找等问题 |
| 优点 | 灵活性高，支持复杂的图算法，可用于构建复杂的关系图谱 |
| 缺点 | 复杂性高，只能支持一定的数据规模 |
| 使用者 | Adobe（Neo4J）、Cisco（Neo4J）、T-Mobile（Neo4J） |

# Overall Rank

## DB-Engines Ranking

The DB-Engines Ranking ranks database management systems according to their popularity. The ranking is updated monthly.

Read more about the method of calculating the scores.



trend chart

423 systems in ranking, December 2024

| Rank Dec 2024 | Rank Nov 2024 | Rank Dec 2023 | DBMS | Database Model | Score Dec 2024 | Score Nov 2024 | Score Dec 2023 |
|---|---|---|---|---|---|---|---|
| 1. | 1. | 1. | Oracle | Relational, Multi-model | 1263.79 | -53.22 | +6.38 |
| 2. | 2. | 2. | MySQL | Relational, Multi-model | 1003.76 | -14.04 | -122.88 |
| 3. | 3. | 3. | Microsoft SQL Server | Relational, Multi-model | 805.69 | +5.88 | -98.14 |
| 4. | 4. | 4. | PostgreSQL | Relational, Multi-model | 666.37 | +12.04 | +15.47 |
| 5. | 5. | 5. | MongoDB | Document, Multi-model | 400.39 | -0.54 | -18.76 |
| 6. | 6. | 6. | Redis | Key-value, Multi-model | 150.27 | +1.63 | -8.08 |
| 7. | 7. | ↑ 10. | Snowflake | Relational | 147.36 | +4.87 | +27.48 |
| 8. | 8. | ↓ 7. | Elasticsearch | Multi-model | 132.32 | +0.68 | -5.43 |
| 9. | 9. | ↓ 8. | IBM Db2 | Relational, Multi-model | 122.78 | +1.04 | -11.81 |
| 10. | 10. | ↑ 11. | SQLite | Relational | 101.72 | +2.24 | -16.23 |

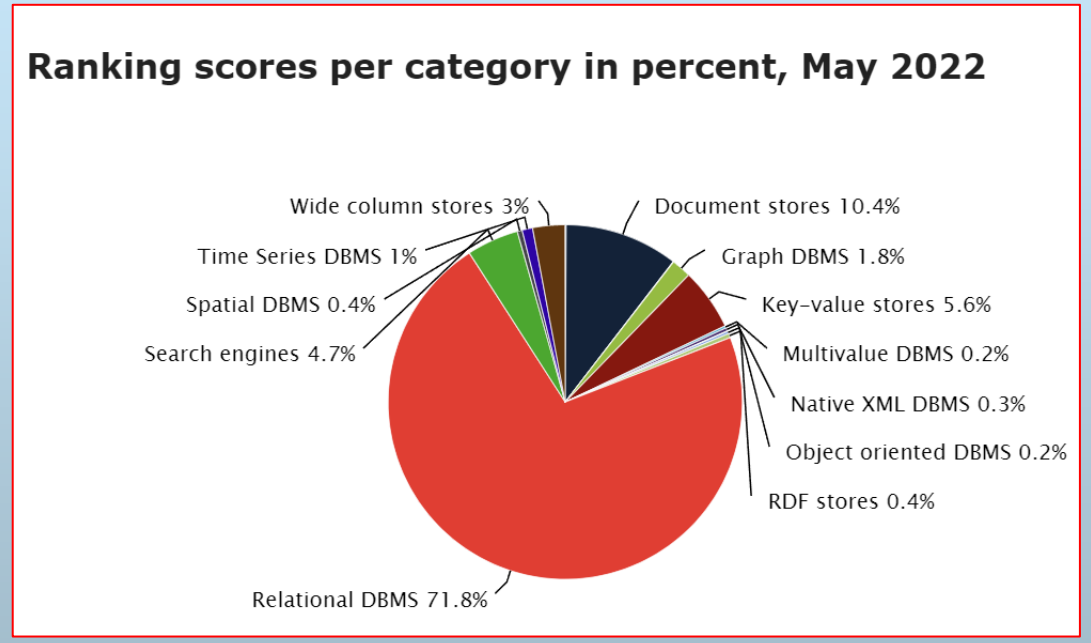
## Method of calculating the scores of the DB-Engines Ranking

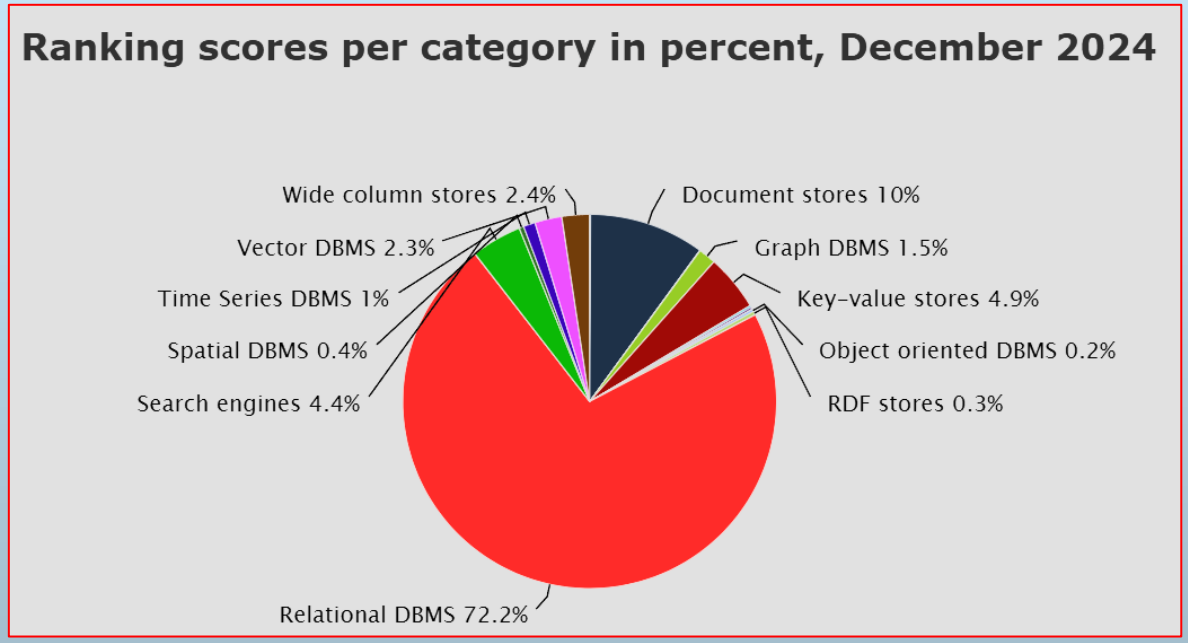
The DB-Engines Ranking is a list of database management systems ranked by their current popularity. We measure the popularity of a system by using the following parameters:

- **Number of mentions of the system on websites**, measured as number of results in search engines queries. At the moment, we use Google and Bing for this measurement. In order to count only relevant results, we are searching for <system name> together with the term database, e.g. "Oracle" and "database".
- **General interest in the system.** For this measurement, we use the frequency of searches in Google Trends.
- **Frequency of technical discussions about the system.** We use the number of related questions and the number of interested users on the well-known IT-related Q&A sites Stack Overflow and DBA Stack Exchange.
- **Number of job offers, in which the system is mentioned.** We use the number of offers on the leading job search engines Indeed and Simply Hired.
- **Number of profiles in professional networks, in which the system is mentioned.** We use the internationally most popular professional network LinkedIn.
- **Relevance in social networks.** We count the number of Twitter (X) tweets, in which the system is mentioned.

# Overall Rank (cont.)

- 关系数据库仍是主流，但NoSQL比例在不断增长

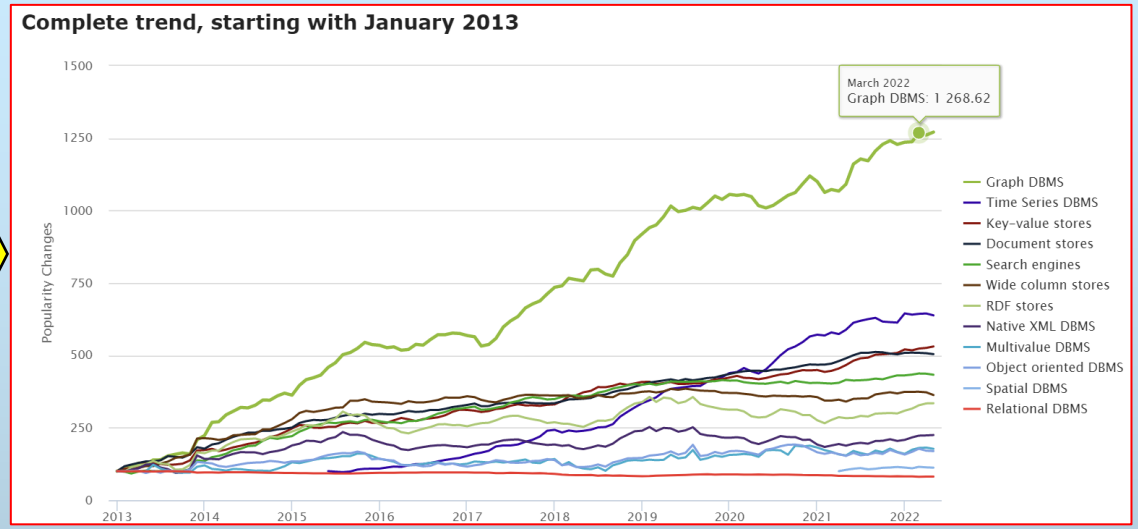


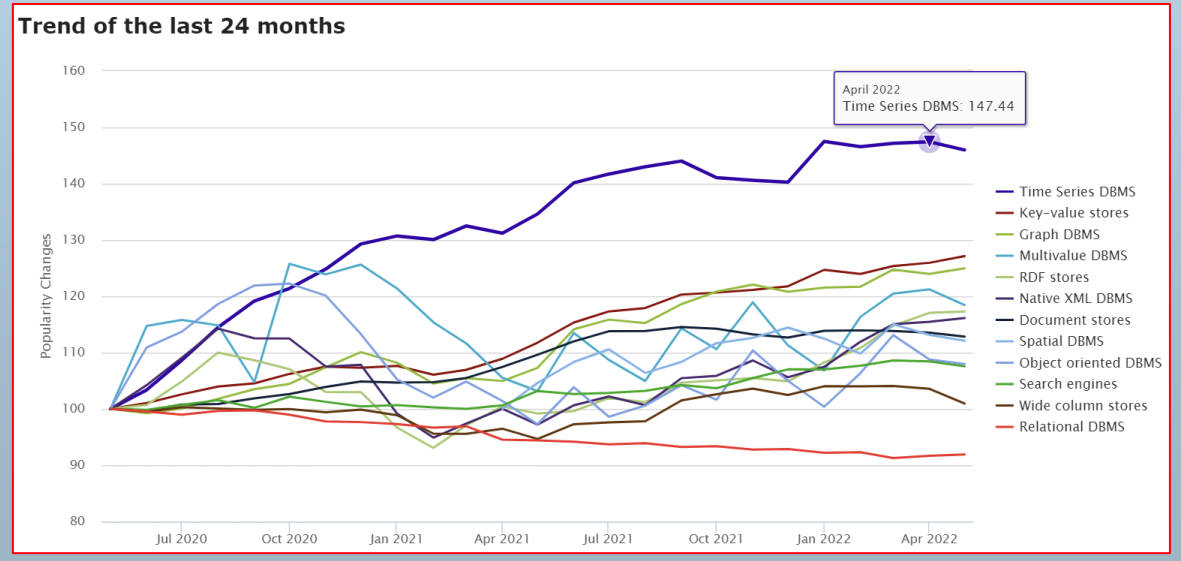
# Overall Rank (cont.)

- 发展趋势

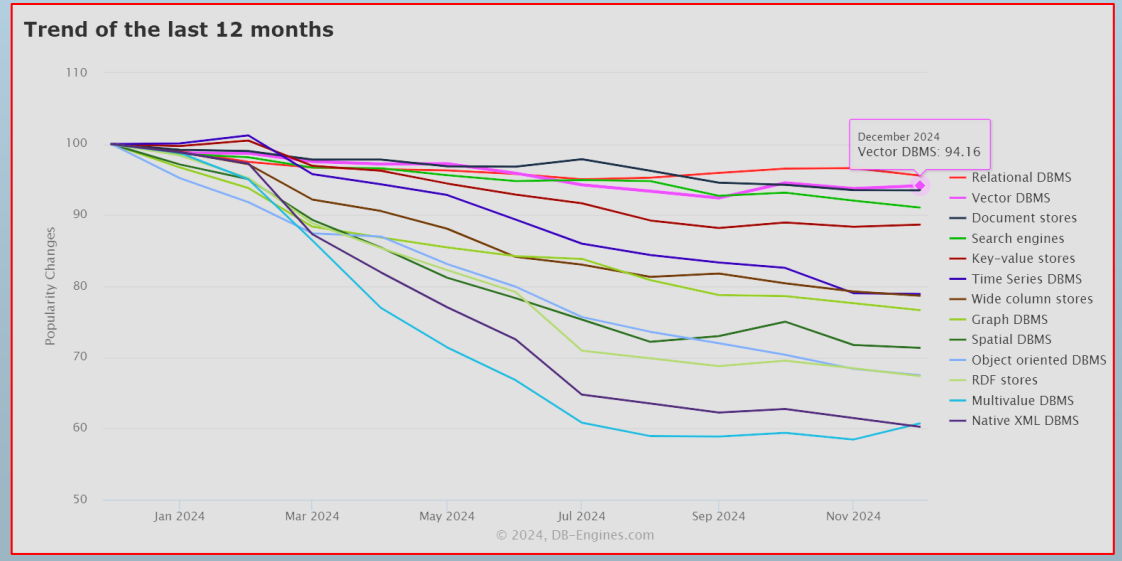
图数据库



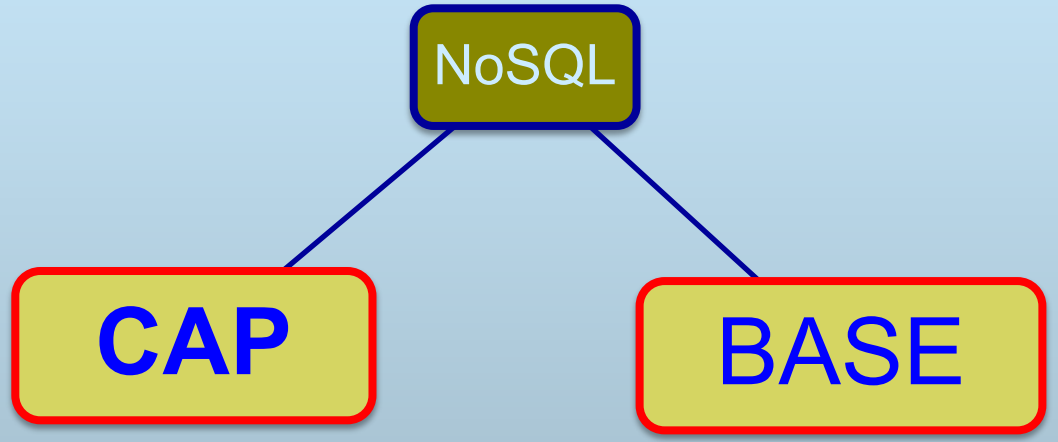

时序数据库




向量数据库



# 三、NoSQL的分布式系统基础



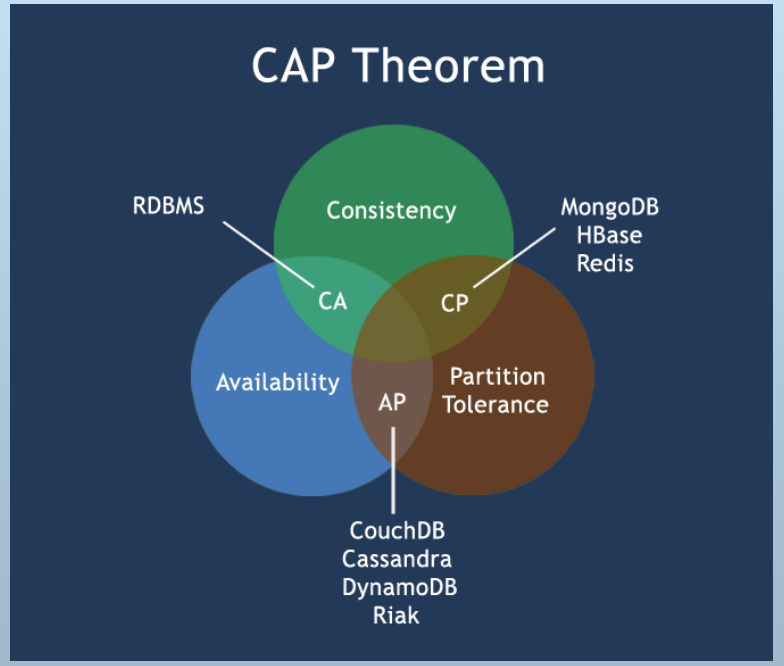

**Note：基本都是分布式系统中的技术，跟数据库系统关系不大**

# 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

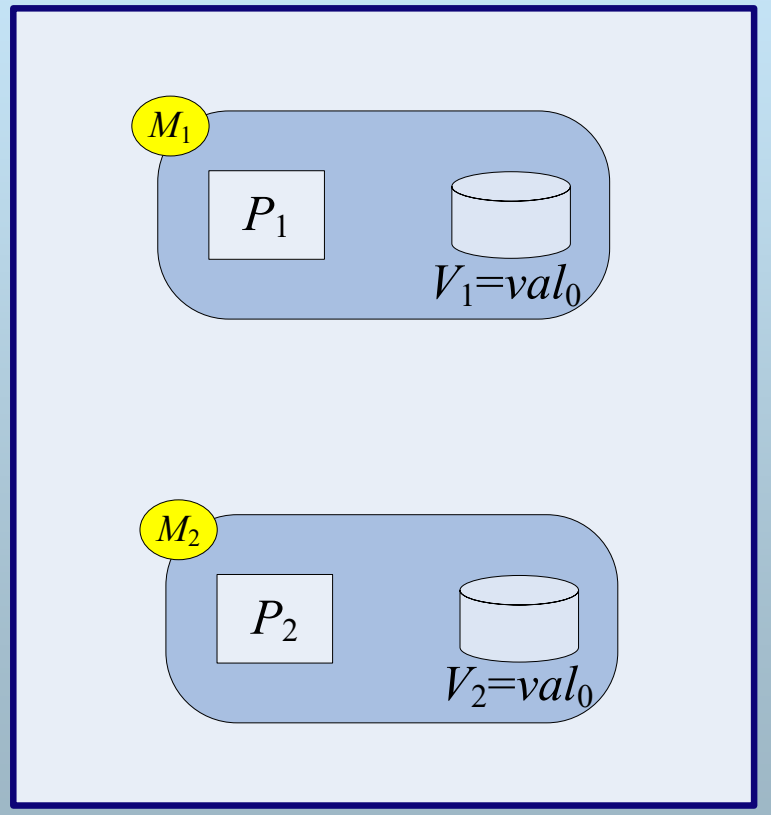
- **Brewer's Theorem (CAP Theorem):** 一个分布式系统不可能同时满足一致性、可用性和分区容忍性这三个需求，最多只能同时满足其中两个 **(Brewer, 2000; Gilbert, 2002 )**




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

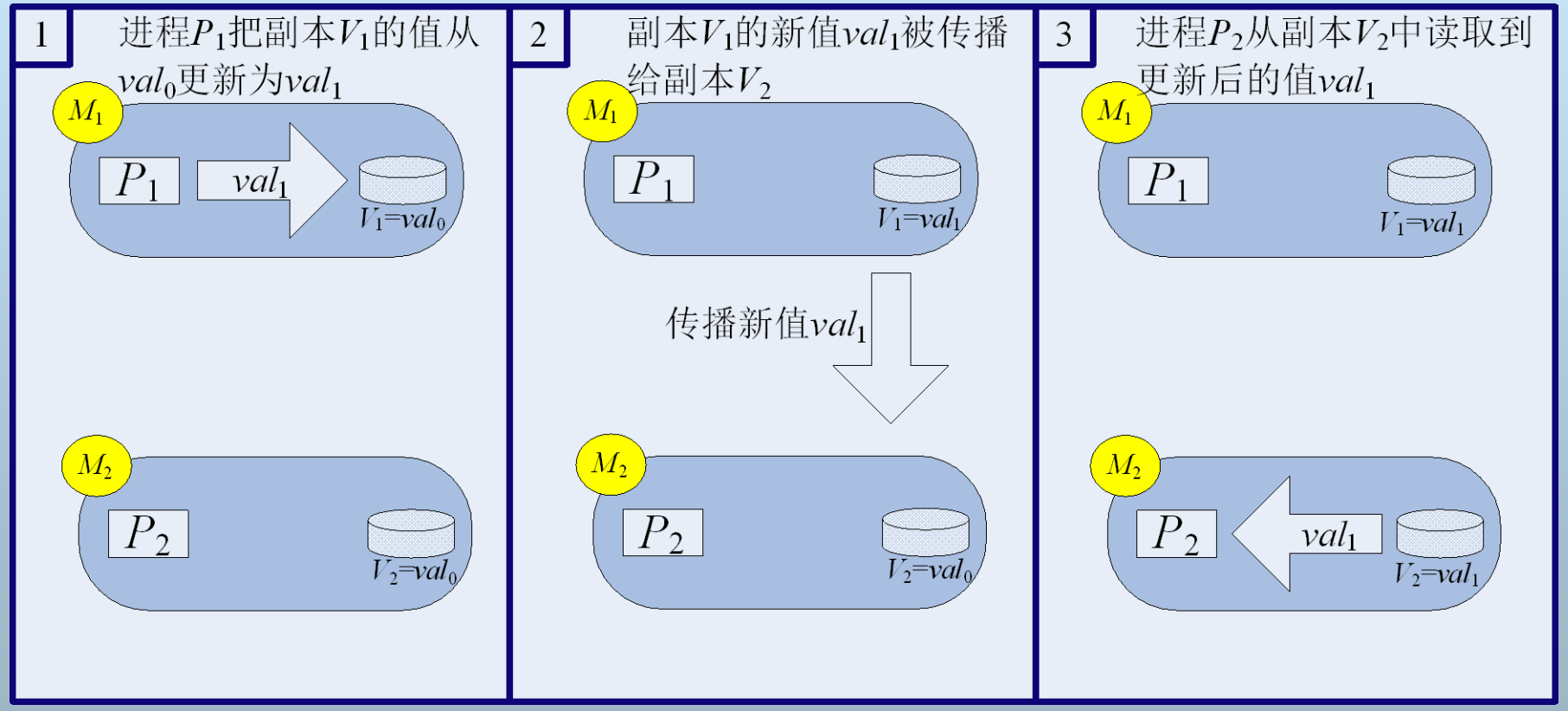
- 一个牺牲一致性来换取可用性的实例




（a）初始状态

# 1、CAP

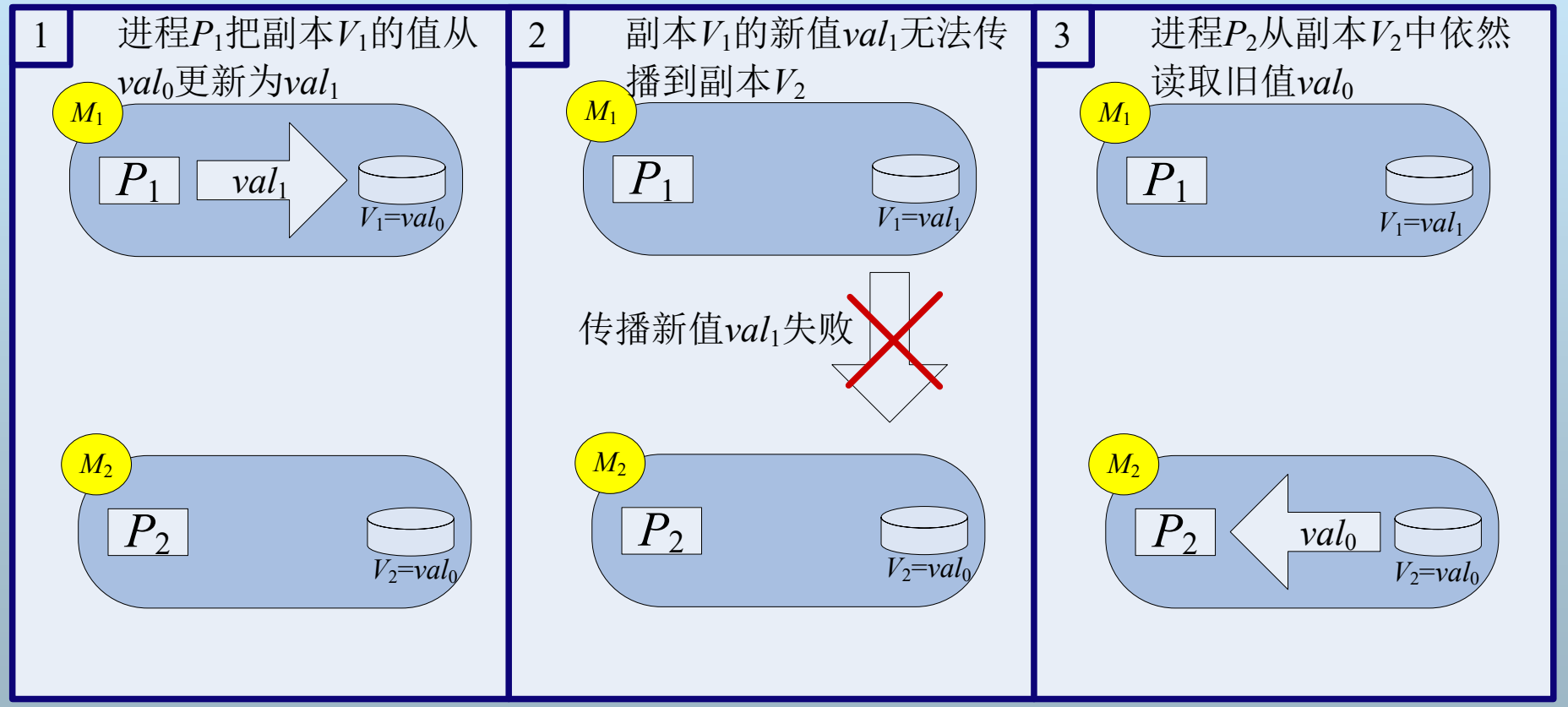
- 一个牺牲一致性来换取可用性的实例




（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

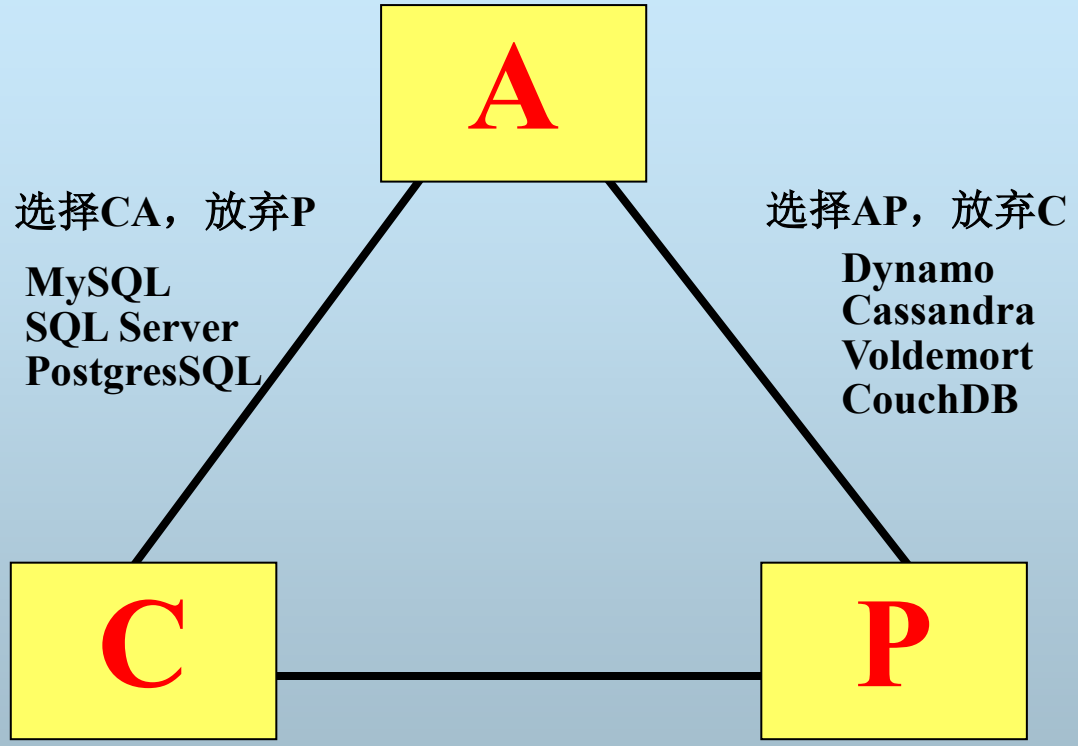



(c) 更新传播失败时的执行过程

# 1、CAP

- 当处理CAP的问题时，有几个明显的选择：
  - **CA**：也就是强调一致性（C）和可用性（A），放弃分区容忍性（P），最简单的做法是把所有与事务相关的内容都放到同一台机器上。很显然，这种做法会严重影响系统的可扩展性。传统的关系数据库（MySQL、SQL Server和PostgreSQL），都采用了这种设计原则，因此，扩展性都比较差
  - **CP**：也就是强调一致性（C）和分区容忍性（P），放弃可用性（A），当出现网络分区的情况时，受影响的服务需要等待数据一致，因此在等待期间就无法对外提供服务
  - **AP**：也就是强调可用性（A）和分区容忍性（P），放弃一致性（C），允许系统返回不一致的数据

# 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

- 事务的ACID回顾
  - A（Atomicity）：原子性，是指事务中的操作要么全都执行，要么全都不执行
  - C（Consistency）：一致性，是指事务在完成时，必须使所有的数据都保持一致状态
  - I（Isolation）：隔离性，是指由并发事务所做的修改必须与任何其它并发事务所做的修改隔离
  - D（Durability）：持久性，是指事务完成之后，它对于系统的影响是持久性的

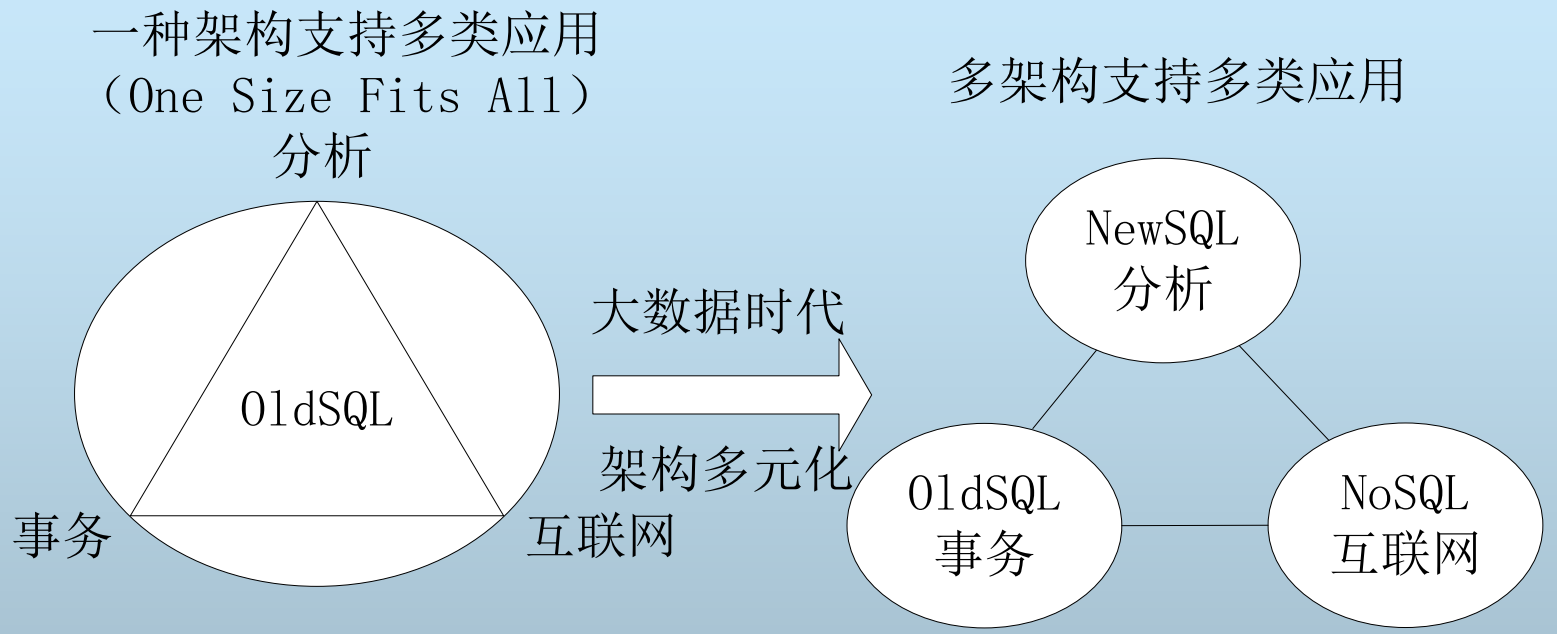
# 2、BASE

- **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

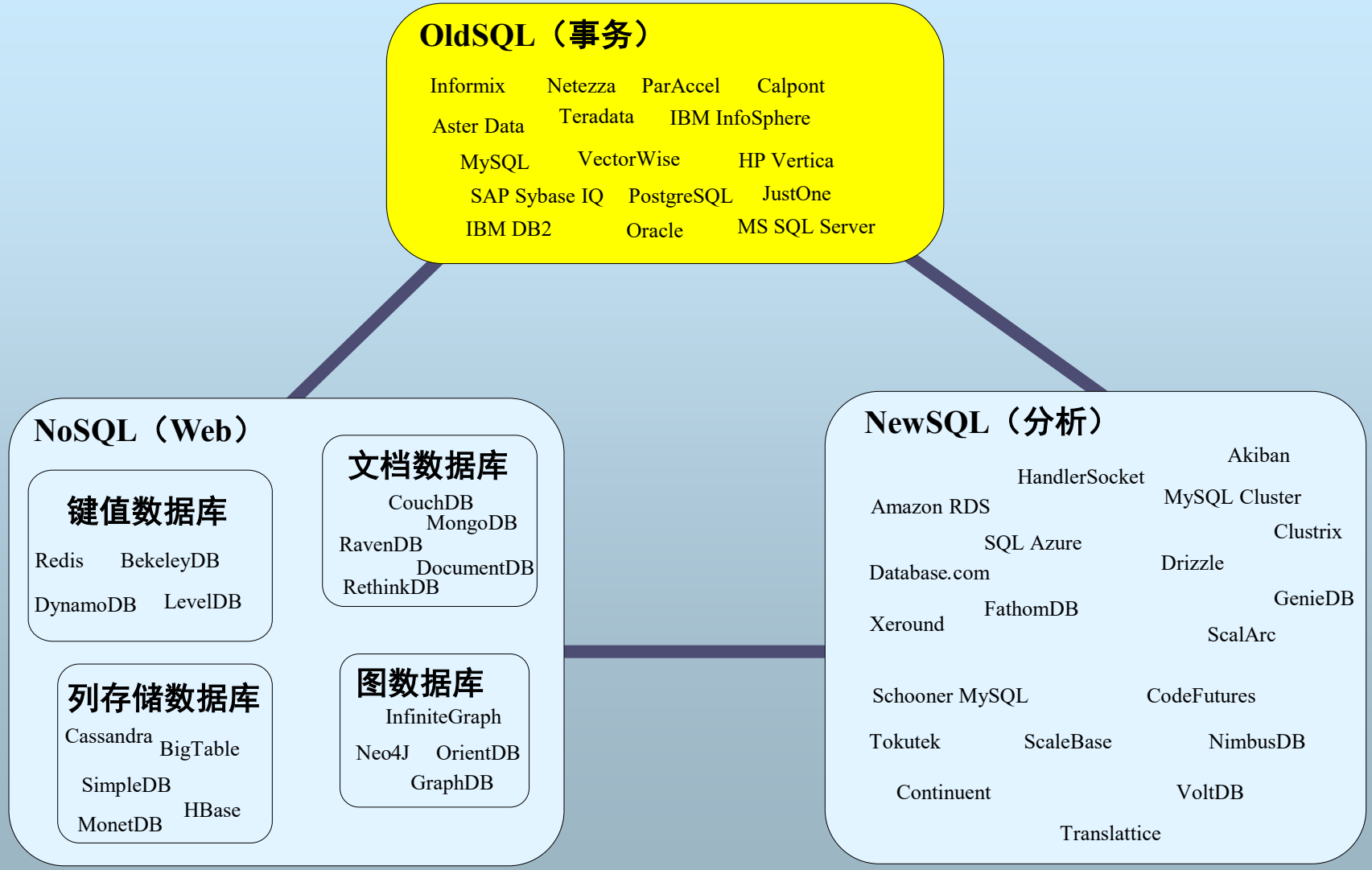
- **Eventual consistency**
  - 一致性的类型包括强一致性和弱一致性。对于强一致性而言，当执行完一次更新操作后，后续的其他读操作就可以保证读到更新后的最新数据。如果不能保证后续访问读到的都是更新后的最新数据，那么就是弱一致性。
  - 最终一致性是弱一致性的一种特例，允许后续的访问操作可以暂时读不到更新后的数据，但是经过一段时间之后，必须最终读到更新后的数据。
    - 最常见的实现最终一致性的系统是DNS（域名系统）。一个域名更新操作根据配置的形式被分发出去，并结合有过期机制的缓存；最终所有的客户端可以看到最新的值。

# 三、从NoSQL到NewSQL




**大数据引发数据处理架构变革**

# 三、从NoSQL到NewSQL



# 1、NewSQL概念

“A relational database that seeks to provide the same scalable performance of NoSQL system while still maintaining the ACID guarantees of a traditional database system”

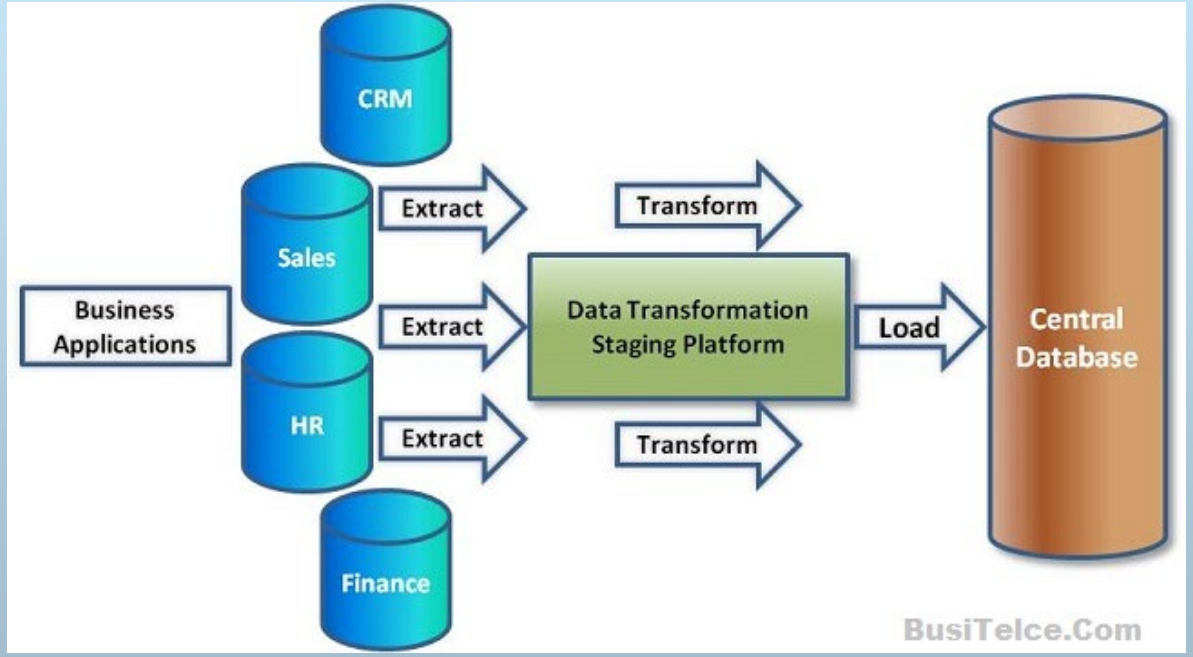
*-- Wikipedia*

# 2、NewSQL Motivation

- **Old OLTP requirements:**
  - **a large number of short on-line transactions (INSERT, UPDATE, DELETE)**
    - **very fast query processing**
    - **data integrity in multi-access environments**
    - **effectiveness measured by number of transactions per second （tps）.**
    - **In OLTP databases there is detailed and current data, and schema used to store transactional databases is the entity model (usually 3NF).**
  - **Enterprises used ETL products to convert OLTP data to a common format and load into data warehouse for performing business analysis.**

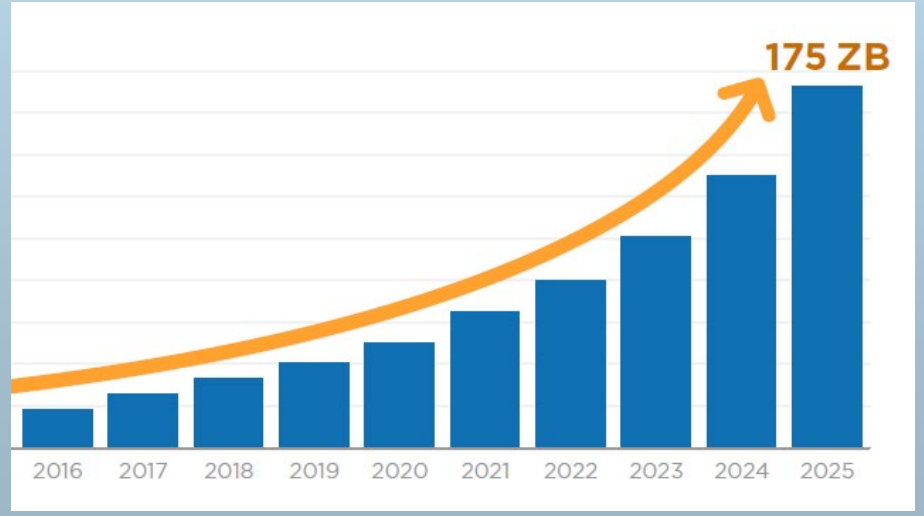
# 2、NewSQL Motivation

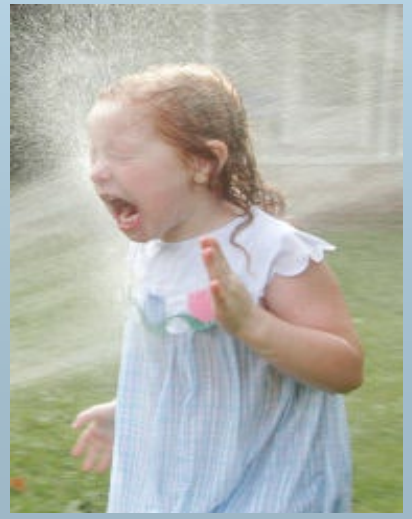
- **Old OLTP requirements:**



# 2、NewSQL Motivation

- **New OLTP requirements:**
  - The transactions are increasing day by day. Good throughput has to be maintained even with the increase in transactions (e.g., Web and smart phones)
  - Need for real-time analytics. (e.g., a Web property wants to know the no. of current users playing its game)



高吞吐、实时

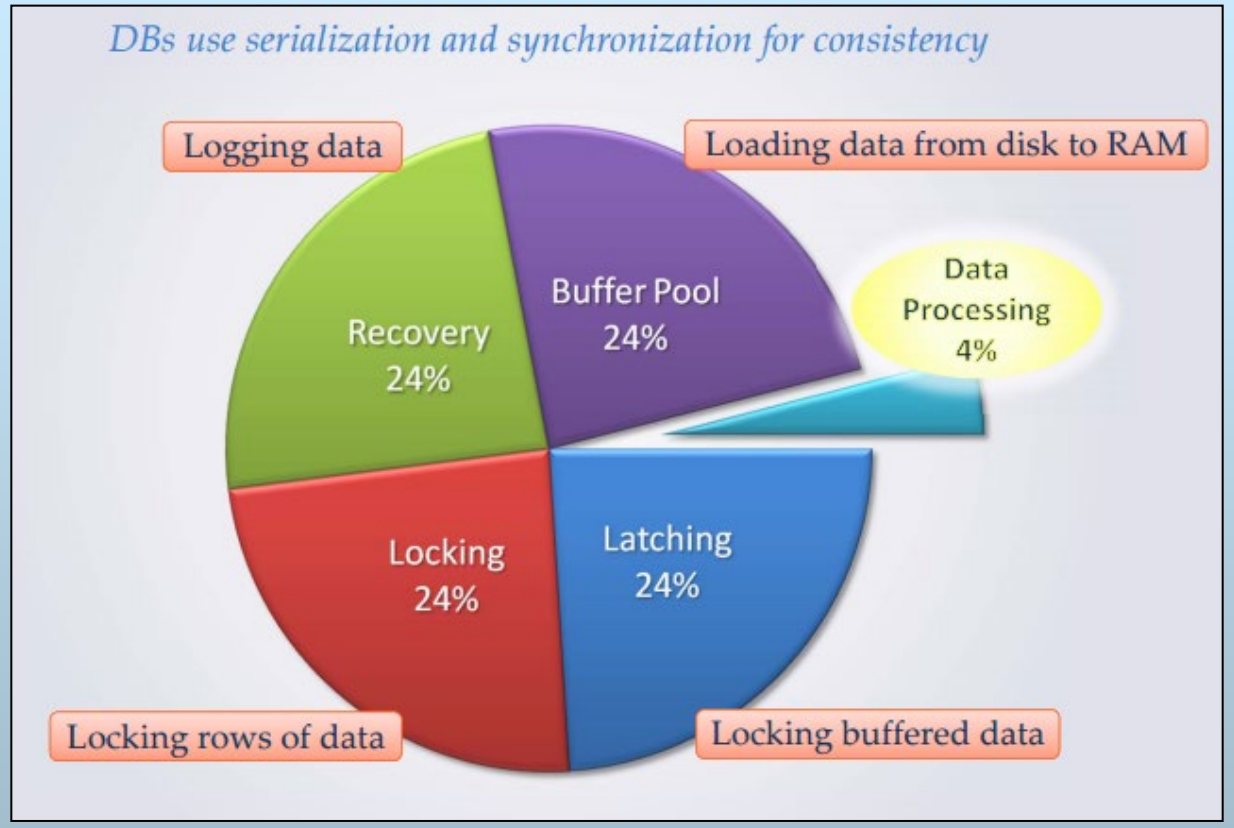
*New OLTP and you*

# 2、NewSQL Motivation

- **Traditional SQL:**
  - The workload experienced by New OLTP may exceed the capabilities of Old SQL solutions.
  - Data warehouses are typically stale by tens of minutes to hours. So real-time analytics are very difficult with Old SQL.

→ Not ideal for New OLTP requirements.

# Traditional SQL Overheads




**"Removing those overheads and running the database in main memory would yield orders of magnitude improvements in database performance"**

*-- White paper of VoltDB*

# 2、NewSQL Motivation

- **NoSQL:**
  - Overcomes workload problems in Old SQL
  - Provides scalability and high performance
  - Achieved through relaxing or eliminating transaction support and moving back to a low-level DBMS interface.
- **Downside:**
  - Give up SQL and ACID for performance

# Give Up SQL?

- Compiler translates SQL at compile time into a sequence of low level operations
- Similar to what the NoSQL products make you program in your application
- 40+ years of RDBMS experience
  - Hard to beat the compiler
  - High level languages are good (data independence, less code, …)
  - Stored procedures are good!
    - One round trip from app to DBMS rather than one round trip per record
    - Move the code to the data, not the other way around

# Give Up ACID?

- If you need data consistency, giving up ACID is a decision to tear your hair out by doing database "heavy lifting" in user code
  - Pushes ACID properties to applications where they are far harder to solve.
- Can you guarantee you won't need ACID tomorrow?

# NoSQL Summary

- Appropriate for non-transactional systems
- Appropriate for single record transactions that are commutative
- Not a good fit for New OLTP
- Use the right tool for the job

Interesting …

Two recently-proposed NoSQL language standards – CQL and UnQL – are amazingly similar to (you guessed it!) SQL

# 2、NewSQL Motivation

- **NewSQL:**
  - Preserve SQL
  - Preserve ACID
  - Performance and scalability through modern innovative software architecture
- **Challenges**
  - Needs something other than traditional record level locking
  - Needs a solution to buffer pool overhead
  - Needs a solution to latching for shared data structures
  - Needs a solution to write-ahead logging

# 3、NewSQL Principles

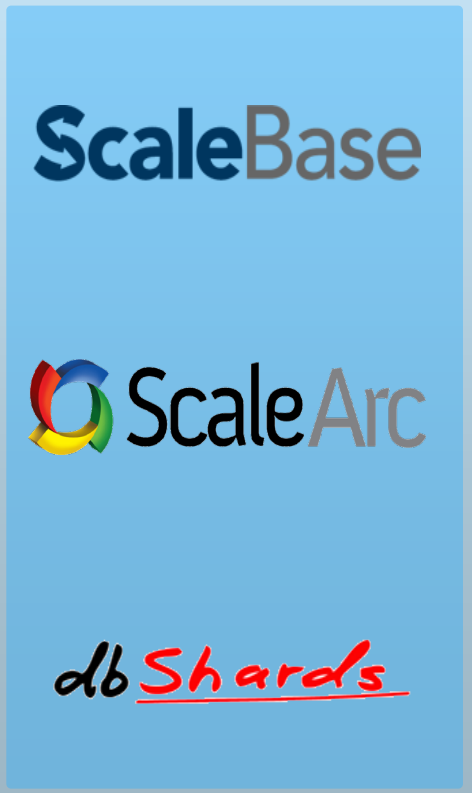
- **Challenges**
  - Needs something other than traditional record level locking
  - ⇨ *Principle 1: minimizing or stay away from locking*
  - Needs a solution to buffer pool overhead
  - ⇨ *Principle 2: rely on main memory*
  - Needs a solution to latching for shared data structures
  - ⇨ *Principle 3: try to avoid latching*
  - Needs a solution to write-ahead logging
  - ⇨ *Principle 4: cheaper solutions for high availability*

# 4、NewSQL的实现类型

## New Design


## MySQL Engines


## Middleware



# 4、NewSQL Summary

- **Pros.**
  - SQL & ACID support
  - Performance: Hundreds of thousands to millions of transactions per second
- **Cons.**
  - Very new
  - In-memory architecture inappropriate for volumes exceeding a few terabytes

# 本章小结

- NoSQL概念
- NoSQL的类型
- CAP & BASE
- NewSQL概念