# 背景

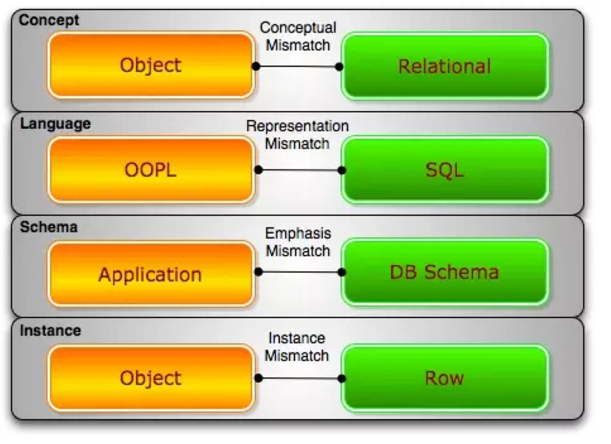
随着互联网网站的兴起，传统的关系型数据库在应对动态网站，特别是超大规模和高并发的纯动态网站已经显得力不从心，暴露了很多难以克服的问题。如商城网站中对商品数据频繁查询、对热搜商品的排名统计、订单超时问题、以及微信朋友圈（音频、视频）存储等相关使用传统的关系型数据库实现就显得非常复杂，虽然能够实现相应功能但是在性能上却不是那么乐观。NoSQL这个技术的出现，更好的解决了这些问题。

在过去几年，关系型数据库一直是数据持久化的唯一选择，数据工作者考虑的也只是在这些传统数据库中做筛选，比如SQL Server、Oracle或者是MySQL。甚至是做一些默认的选择，比如使用.NET的一般会选择SQL Server；使用Java的可能会偏向Oracle，Ruby是MySQL，Python则是PostgreSQL或MySQL等等。

原因很简单，过去很长一段时间内，关系数据库的健壮性已经在多数应用程序中得到证实。我们可以使用这些传统数据库良好的控制并发操作、事务等等。然而如果传统的关系型数据库一直这么可靠，那么还有NoSQL什么事？NoSQL之所以生存并得到发展，是因为它做到了传统关系型数据库做不到的事！

关系型数据库中存在的问题：

1、Impedance Mismatch



我们使用Python、Ruby、Java、.Net等语言编写应用程序，这些语言有一个共同的特性——面向对象。但是我们使用MySQL、PostgreSQL、Oracle以及SQL Server，这些数据库同样有一个共同的特性——关系型数据库。这里就牵扯到了“Impedance Mismatch”这个术语：存储结构是面向对象的，但是数据库却是关系的，所以在每次存储或者查询数据时，我们都需要做转换。类似Hibernate、Entity Framework这样的ORM框架确实可以简化这个过程，但是在对查询有高性能需求时，这些ORM框架就捉襟见肘了。

2、应用程序规模的变大

网络应用程序的规模日渐变大，我们需要储存更多的数据、服务更多的用户以及需求更多的计算能力。为了应对这种情形，我们需要不停的扩展。扩展分为两类：一种是纵向扩展，即购买更好的机器，更多的磁盘、更多的内存等等；另一种是横向扩展，即购买更多的机器组成集群。

在巨大的规模下，纵向扩展发挥的作用并不是很大。首先单机器性能提升需要巨额的开销并且有着性能的上限，在Google和Facebook这种规模下，永远不可能使用一台机器支撑所有的负载。鉴于这种情况，我们需要新的数据库，因为关系数据库并不能很好的运行在集群上。不错你也可能会去搭建关系数据库集群，但是他们使用的是共享存储，这并不是我们想要的类型。于是就有了以Google、Facebook、Amazon这些试图处理更多传输所引领的NoSQL纪元。

# 概述

NoSQL是指那些非关系型的、分布式的、不保证遵循ACID原则的数据存储系统，并分为key-value存储、文档数据库和图数据库这3类。其中，key-value存储备受关注，已成为NoSQL的代名词。

NoSQL数据库使用了不同于关系模型的模型，例如键值模型、文档模型、宽列模型和图模型等。采用这些模型的NoSQL数据库并不提供规范化，本身在设计上是无模式的。大多数NoSQL数据库支持自动分区，无需开发人员干预即可轻松实现水平扩展。

## RDBMS vs NoSQL

NoSQL（Not Only SQL），意即不仅仅是SQL，泛指非关系型数据库。

RDBMS：关系型数据库SQL语句

NoSQL：泛指非关系型数据库

## 特点

### 优点

- 高可扩展性

- 分布式计算

- 低成本

- 架构的灵活性，半结构化数据

- 没有复杂的关系

### 缺点

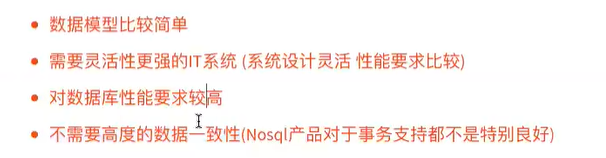
- 没有标准化

- 有限的查询功能（到目前为止）

- 最终一致是不直观的程序

## 应用场景

NoSQL适用于可接受最终一致性的部分应用，例如社交媒体。用户并不关注看到的是否为不一致的数据库视图，并且考虑到数据的状态更新、发推文等，强一致性也并非必要的。但是，NoSQL数据库不宜用于对一致性要求高的系统，例如电子商务平台。



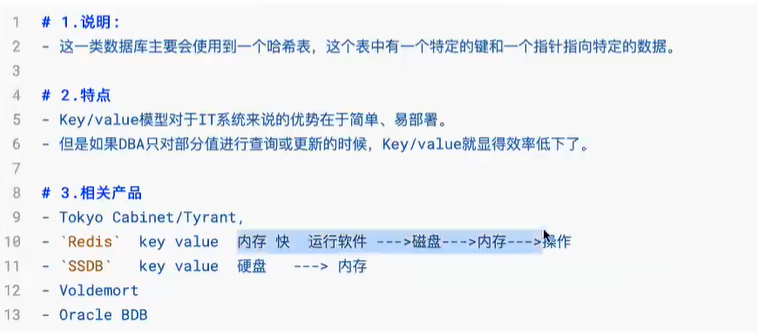
# 分类

|  |  |  |
| --- | --- | --- |
| 类型 | 部分代表 | 特点 |
| 列存储 | Hbase  Cassandra  Hypertable | 顾名思义，是按列存储数据的。最大的特点是方便存储结构化和半结构化数据，方便做数据压缩，对针对某一列或者某几列的查询有非常大的IO优势。 |
| 文档存储 | MongoDB  CouchDB | 文档存储一般用类似json的格式存储，存储的内容是文档型的。这样也就有机会对某些字段建立索引，实现关系数据库的某些功能。 |
| key-value存储 | Tokyo Cabinet/Tyrant  Berkeley DB  MemcacheDB  Redis | 可以通过key快速查询到其value。一般来说，存储不管value的格式，照单全收。（Redis包含了其他功能） |
| 图存储 | Neo4J  FlockDB | 图形关系的最佳存储。使用传统关系数据库来解决的话性能低下，而且设计使用不方便。 |
| 对象存储 | db4o  Versant | 通过类似面向对象语言的语法操作数据库，通过对象的方式存取数据。 |
| xml数据库 | Berkeley DB XML  BaseX | 高效的存储XML数据，并支持XML的内部查询语法，比如XQuery,Xpath。 |

对比传统关系型数据库，NoSQL有着更为复杂的分类——键值（Key-value）、面向文档（Document-Oriented）、列存储（Column-Family Databases）以及图数据库（Graph-Oriented Databases）。

## 键值（key-value）存储数据库

针对key-value数据存储的细微不同，研究者又进一步将key-value存储细分为key-document存储（MongoDB，CouchDB）、 key-column存储（Cassandra，Voldemort，Hbase）和key-value存储（Redis，Tokyo Cabinet）。



键值数据库就像在传统语言中使用的哈希表。你可以通过key来添加、查询或者删除数据，鉴于使用主键访问，所以会获得不错的性能及扩展性。

### 特点

### 产品

产品：Riak、Redis、Memcached

有谁在使用：GitHub（Riak）、BestBuy（Riak）、Twitter（Redis和Memcached）、StackOverFlow（Redis）、Instagram（Redis）、Youtube（Memcached）、Wikipedia（Memcached）

#### Redis

#### Memcached

### 应用

**适用的场景**

储存用户信息，比如会话、配置文件、参数、购物车等等。这些信息一般都和ID（键）挂钩，这种情景下键值数据库是个很好的选择。

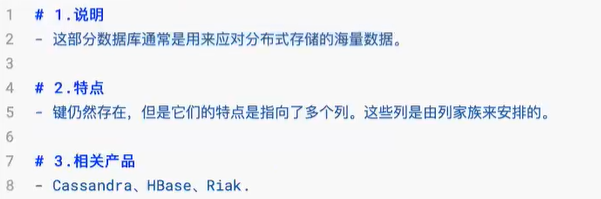
**不适用场景**

1、取代通过键查询，而是通过值来查询。Key-Value数据库中根本没有通过值查询的途径。

2、需要储存数据之间的关系。在Key-Value数据库中不能通过两个或以上的键来关联数据。

3、事务的支持。在Key-Value数据库中故障产生时不可以进行回滚。

## 列存储数据库



 列存储数据库将数据储存在列族（column family）中，一个列族存储经常被一起查询的相关数据。

举个例子，如果我们有一个Person类，我们通常会一起查询他们的姓名和年龄而不是薪资。这种情况下，姓名和年龄就会被放入一个列族中，而薪资则在另一个列族中。

### 特点

### 产品

产品：Cassandra、HBase

有谁在使用：Ebay（Cassandra）、Instagram（Cassandra）、NASA（Cassandra）、Twitter（Cassandra and HBase）、Facebook（HBase）、Yahoo!（HBase）

#### Bigtable

Google的Bigtable，非开源。

#### HBase

基于Hadoop HDFS，开源。

#### Dynamo

Amazon的Dynamo，非开源。

#### Cassandra

Apache的Cassandra，开源。

#### Project Voldemort

### 应用

**适用的场景**

1、日志。因为我们可以将数据储存在不同的列中，每个应用程序可以将信息写入自己的列族中。

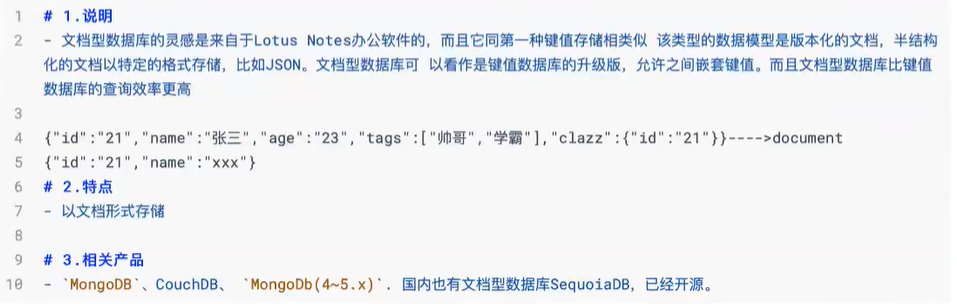
2、博客平台。我们储存每个信息到不同的列族中。举个例子，标签可以储存在一个，类别可以在一个，而文章则在另一个。

**不适用场景**

1、如果我们需要ACID事务。Vassandra就不支持事务。

2、原型设计。如果我们分析Cassandra的数据结构，我们就会发现结构是基于我们期望的数据查询方式而定。在模型设计之初，我们根本不可能去预测它的查询方式，而一旦查询方式改变，我们就必须重新设计列族。

## 文档型数据库



 面向文档数据库会将数据以文档的形式储存。每个文档都是自包含的数据单元，是一系列数据项的集合。每个数据项都有一个名称与对应的值，值既可以是简单的数据类型，如字符串、数字和日期等；也可以是复杂的类型，如有序列表和关联对象。数据存储的最小单位是文档，同一个表中存储的文档属性可以是不同的，数据可以使用XML、JSON或者JSONB等多种形式存储。

### 特点

### 产品

产品：MongoDB、CouchDB、RavenDB

有谁在使用：SAP（MongoDB）、Codecademy（MongoDB）、Foursquare（MongoDB）、NBC News（RavenDB）

#### MongoDB

#### CouchDB

### 应用

**适用的场景**

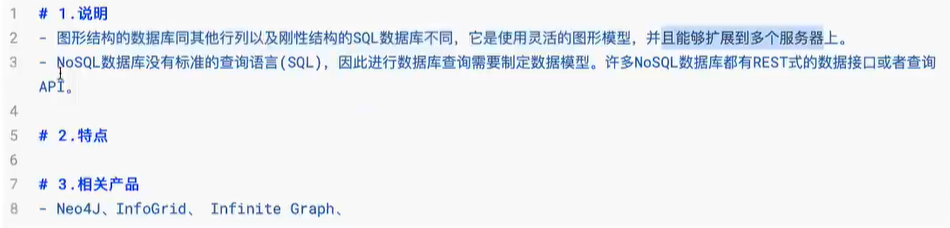
1、日志。企业环境下，每个应用程序都有不同的日志信息。Document-Oriented数据库并没有固定的模式，所以我们可以使用它储存不同的信息。

2、分析。鉴于它的弱模式结构，不改变模式下就可以储存不同的度量方法及添加新的度量。

**不适用场景**

在不同的文档上添加事务。Document-Oriented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。

## 图形（Graph）数据库



图数据库允许我们将数据以图的方式储存。实体会被作为顶点，而实体之间的关系则会被作为边。比如我们有三个实体，Steve Jobs、Apple和Next，则会有两个“Founded by”的边将Apple和Next连接到Steve Jobs。

### 特点

### 产品

产品：Neo4J、Infinite Graph、OrientDB

有谁在使用：Adobe（Neo4J）、Cisco（Neo4J）、T-Mobile（Neo4J）

#### Neo4J

#### Infinite Graph

### 应用

**适用的场景**

1、在一些关系性强的数据中

2、推荐引擎。如果我们将数据以图的形式表现，那么将会非常有益于推荐的制定不适用场景不适合的数据模型。图数据库的适用范围很小，因为很少有操作涉及到整个图。

# 事务

## ACID

NoSQL数据库给出了一种易于实现可扩展性和更好性能的解决方案，解决了CAP理论中的A（可用性）和P（分区容错性）上的设计考虑。但这意味着，**在很多NoSQL设计中实现为最终一致性，摈弃了RDBMS提供的强一致性及事务的ACID属性**。

## CAP

在计算机科学中, CAP定理（CAP theorem）, 又被称作布鲁尔定理（Brewer's theorem）, 它指出对于一个分布式计算系统来说，不可能同时满足以下三点：

一致性(Consistency) (所有节点在同一时间具有相同的数据)

可用性(Availability) (保证每个请求不管成功或者失败都有响应)

分隔容忍(Partition tolerance) (系统中任意信息的丢失或失败不会影响系统的继续运作)

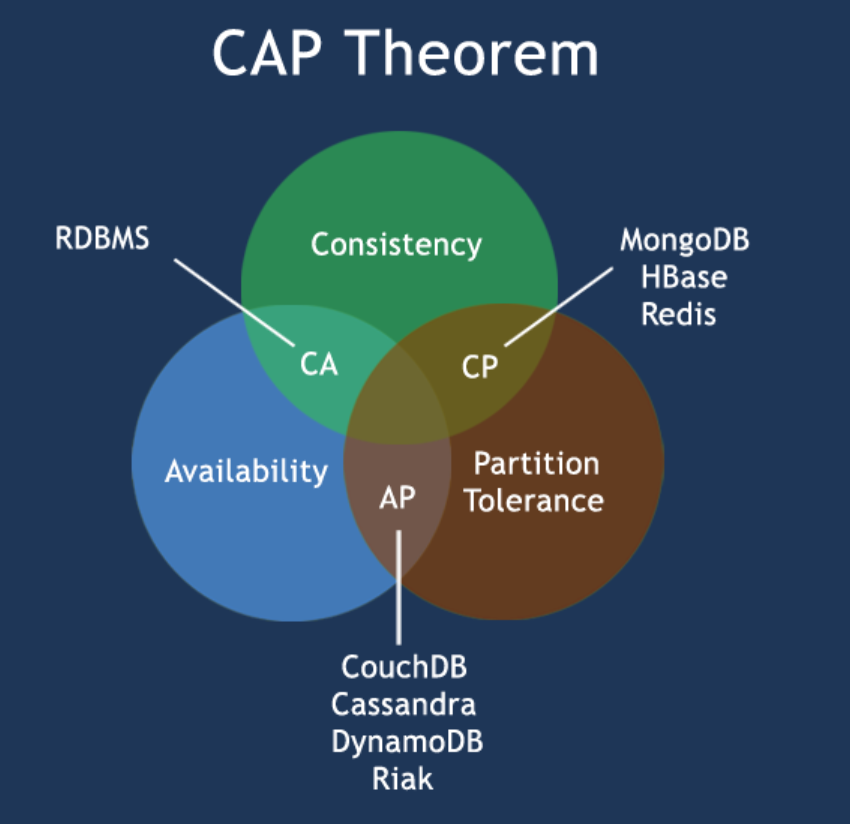
CAP理论的核心是：一个分布式系统不可能同时很好的满足一致性，可用性和分区容错性这三个需求，最多只能同时较好的满足两个。

因此，根据CAP原理将NoSQL数据库分成了满足CA原则、满足CP原则和满足AP原则三大类：

CA - 单点集群，满足一致性，可用性的系统，通常在可扩展性上不太强大。

CP - 满足一致性，分区容忍性的系统，通常性能不是特别高。

AP - 满足可用性，分区容忍性的系统，通常可能对一致性要求低一些。



大部分key-value数据库系统都会根据自己的设计目的进行相应的选择，如Cassandra，Dynamo满足AP；BigTable，MongoDB满足CP；而关系数据库，如Mysql和Postgres满足AC。

## BASE

BASE即Basically Available(基本可用)、Soft state(柔性状态)和 Eventually consistent(最终一致)的缩写。Basically Available是指可以容忍系统的短期不可用，并不强调全天候服务；Soft state是指状态可以有一段时间不同步，存在异步的情况；Eventually consistent是指最终数据一致，而不是严格的时时一致。

BASE是NoSQL数据库通常对可用性及一致性的弱要求原则：

Basically Available --基本可用

Soft-state --软状态/柔性事务。"Soft state"可以理解为"无连接"的, 而"Hard state"是"面向连接"的

Eventually Consistency --最终一致性，也是ACID的最终目的。



## FIT

参考：

<https://www.jdon.com/47671>

像MongoDB, Cassandra, HBase, DynamoDB, 和 Riak这些NoSQL缺乏传统的原子事务机制，所谓原子事务机制是可以保证一系列写操作要么全部完成，要么全部不会完成，不会发生只完成一系列中一两个写操作；因为数据库不提供这种事务机制支持，开发者需要自己编写代码来确保一系列写操作的事务机制，比较复杂和测试。

这些NoSQL数据库不提供事务机制原因在于其分布式特点，一系列写操作中访问的数据可能位于不同的分区服务器，这样的事务就变成分布式事务，在分布式事务中实现原子性需要彼此协调，而协调是耗费时间的，每台机器在一个大事务过程中必须依次确认，这就需要一种协议确保一个事务中没有任何一台机器写操作失败。

这种协调是昂贵的，会增加延迟时间，关键问题是，当协调没有完成时，其他操作是不能读取事务中写操作结果的，这是因为事务的all-or-nothing原理导致，万一协调过程发现某个写操作不能完成，那么需要将其他写操作成功的进行回滚。针对分布式事务的分布式协调对整体数据库性能有严重影响，不只是吞吐量还包括延迟时间，这样大部分NoSQL数据库因为性能问题就选择不提供分布式事务。

MongoDB, Riak, HBase和 Cassandra提供基于单一键的事务，这是因为所有信息都和一个键key有关，这个键是存储在单个服务器上，这样基于单键的事务不会带来复杂的分布式协调。

那么看来扩展性性能和分布式事务是一对矛盾，总要有取舍？实际上是不完全是，现在完全有可能提供高扩展的性能同时提供分布式原子事务。

[FIT](http://cs-www.cs.yale.edu/homes/dna/papers/fit.pdf)是这样一个在分布式系统提供原子事务的策略，在fairness公平性, isolation隔离性, 和throughput吞吐量（简称FIT）可以权衡。

一个支持分布式事务的可伸缩分布式系统能够完成这三个属性中两个，公平是事务之间不会相互影响造成延迟；隔离性提供一种幻觉好像整个数据库只有它自己一个事务，隔离性保证当任何同时发生的事务发生冲突时，能够保证彼此能看到彼此的写操作结果，因此减轻了程序员为避免事务读写冲突的强逻辑推理要求；吞吐量是指每单元时间数据库能够并发处理多少事务。

FIT是如下进行权衡：

1、保证公平性fairness 和隔离性isolation, 但是牺牲吞吐量

2、保证公平性fairness和吞吐量, 牺牲隔离性isolation

3、保证隔离性isolation和吞吐量throughput, 但是牺牲公平性fairness.

牺牲公平性：放弃公平性，数据库能有更多机会降低分布式事务的成本，主要成本是分布式协调带来的，也就是说，不需要在每个事务过程内对每个机器都依次确认事务完成，这样排队式的确认commit事务是很浪费时间的，放弃公平性，意味着可以在事务外面进行协调，这样就只是增加了协调时间，不会增加互相冲突事务因为彼此冲突而不能运行所耽搁的时间，当系统不需要公平性时，需要根据事务的优先级或延迟等标准进行指定先后执行顺序，这样就能够获得很好的吞吐量。

[G-Store](http://www.cs.ucsb.edu/~sudipto/papers/socc10-das.pdf)是一种放弃公平性的 Isolation-Throughput 的分布式key-value存储，支持多键事务(multi-key transactions)，MongoDB 和 HBase在键key在同样分区上也支持多键事务，但是不支持跨分区的事务。

总之：传统分布式事务性能不佳的原因是确保原子性（分布式协调）和隔离性同时重叠，创建一个高吞吐量分布式事务的关键是分离这两种关注，这种分离原子性和隔离性的视角将导致两种类型的系统，第一种选择是弱隔离性能让冲突事务并行执行和确认提交；第二个选择重新排序原子性和隔离性机制保证它们不会某个时间重叠，这是一种放弃公平的事务执行，所谓放弃公平就是不再同时照顾原子性和隔离性了，有所倾斜，放弃高标准道德要求就会带来高自由高效率。

# 关键技术

# 体系结构

尽管目前流行的NoSQL数据存储系统的设计与实现方式各有不同，但是总结起来大体上有两种架构：master-slave结构和P2P环形结构，两者各具特色。

## master-slave结构

在采用master-slave结构的系统中，master节点负责管理整个系统，监视slave节点的运行状态，同时为其下的每一个slave节点分配存储的范围，是查询和写入的入口。

master节点一般全局只有1个，该节点的状态将严重影响整个系统的性能，当master节点宕机时，会引起整个系统的瘫痪。实践中，经常设置多个副本master节点，通过联机热备的方式提高系统的容错性。

slave节点是数据存储节点，通常也维护一张本地数据的索引表。系统通过添加slave节点来实现系统的水平扩展。

在master-slave框架下，master节点一直处于监听状态，而slave节点之间尽量避免直接通信以减少通信代价。在运行过程中，salve节点不断地向master节点报告自身的健康状况和负载情况，当某个节点宕机或负载过高时，由master节点统一调度，或者将此节点的数据重新分摊给其他节点，或者通过加入新节点的方式来调节。BigTable，Hbase是典型的master-slave结构的key-value存储系统。

## P2P环形结构

在P2P环形结构中，系统节点通过分布式哈希算法在逻辑上组成一个环形结构，其中的每个node节点不但存储数据，而且管理自己负责的区域。

P2P环形结构没有master节点，可以灵活地添加节点来实现系统扩充，节点加入时只需与相邻的节点进行数据交换，不会给整个系统带来较大的性能抖动。P2P 环形结构没有中心点，每个节点必须向全局广播自己的状态信息。例如，目前流行的采用P2P环形结构的Cassandra和Dynamo系统采用Gossip机制来进行高效的消息同步。

# 数据存储

## K-V数据模型

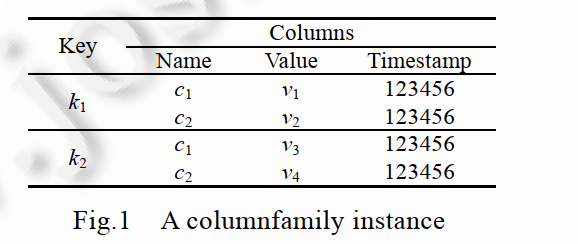
### key-value型

Key-Value键值对数据模型实际上是一个映射，即key是查找每条数据地址的唯一关键字，value是该数据实际存储的内容。例如键值对：(“20091234”,“张三”)，其key:“20091234”是该数据的唯一入口，而value：“张三”是该数据实际存储的内容。

Key-Value数据模型典型的是采用哈希函数实现关键字到值的映射，查询时，基于key的hash值直接定位到数据所在的点，实现快速查询，并支持大数据量和高并发查询。

### key-document型

Key-Column型数据模型主要来自Google的BigTable，目前流行的开源项目Hbase和Cassandra也采用了该种模型。Column型数据模型可以理解成一个多维度的映射，主要包含column，row和columnfamily等概念。



如图1所示，在 key-column型数据模型中，column是数据库中最小的存储单元，它是一个三元组，包括name(如c1,c2)，value(如v1,v2)和timestamp(如 123456)，即一个带有时间戳的key-value键值对。每一个row也是一个key-value对，对于任意一个row，其key是该row下数据的唯一入口(如k1)，value是一个column的集合(如column:c1,c2)。Columnfamily是一个包含了多个row的结构，相当于关系库中表的概念。  
 简单来说，key-column型数据模型是通过多层的映射模拟了传统表的存储格式，实际上类似于key-value数据模型，需要通过key进行查找。因此，key-column型数据模型是key-value数据模型的一种扩展。

### key-column型

数据模型是数据管理所关注的核心问题。Key-Value数据模型因其简单以及具有灵活的可扩展性而广泛被云系统所采用。目前，已有一些key-value数据库产品都是面向特定应用构建的，支持的功能以及采用的关键技术都存在很大差别，并没有形成一套系统化的规范准则。因此，需要规范key-value数据模型及其支持理论，主要包括：

1. 研究key-value数据模型的规范定义和所支持的基本操作；
2. 研究面向应用设计key-value数据组织所遵循的准则，如代价最小化的key-value数据物理组织模型、代价最小化的数据可扩展的启发式准则，为数据最优组织提供遵循准则；
3. 研究key-value数据对间的关联关系以及正确性验证规则，为数据组织的合理性和正确性提供一定的依据。

## 读写方式

## 索引技术

### B Tree

### B+ Tree

### LSM

在单机环境下，类似 \*\*Pebble\*\* 的 LSM（Log-Structured Merge）树实现有一些开源项目，通常设计上更加轻量级、易于嵌入应用程序，并且专注于高效的磁盘存储和快速的数据访问。以下是一些类似 \*\*Pebble\*\* 的单机 LSM 实现：

#### Pebble

**- 简介：**

- Pebble 是一个高性能、嵌入式的键值存储引擎，采用 LSM 树实现，主要由 CockroachDB 团队开发。它是对 LevelDB 和 RocksDB 的轻量级替代品，特别适合用于需要嵌入式存储的场景。

**- 特点：**

1、高效的 LSM 树实现：提供对键值对的高效插入、查询、删除操作。

2、内存和磁盘的高效管理：优化了内存和磁盘之间的数据合并、压缩等操作。

3、支持事务：支持原子性操作和批量操作。

4、并发读写：支持多个读操作与少量写操作的并发执行。

**- 适用场景：**

- 适用于需要嵌入式存储、快速键值存储的应用，如数据库、缓存、日志系统等。

**- 开源地址：**

- [Pebble GitHub](https://github.com/cockroachdb/pebble)

#### LevelDB

**- 简介：**

- LevelDB 是由 Google 开发的嵌入式键值数据库，采用 LSM 树存储结构。它是一个轻量级的库，专为高效的存储和快速的随机访问设计。

**- 特点：**

1、小巧高效：适合嵌入式应用，使用简单，具有较低的资源占用。

2、良好的写入性能：基于 LSM 树的设计，写入操作具有高效性，适合写密集型场景。

3、简单的 API：提供基本的键值存取 API，适合用于轻量级应用。

**- 适用场景：**

- 适用于嵌入式数据库、文件存储、缓存系统等需要高效读写的小型应用。

**- 开源地址：**

- [LevelDB GitHub](https://github.com/google/leveldb)

#### Badger

**- 简介：**

- Badger 是一个高性能的嵌入式键值数据库，采用 LSM 树作为其存储引擎，专为高并发写入和低延迟查询设计。它是 Dgraph（一个分布式图数据库）的底层存储引擎。

- 特点：

1、高并发写入：优化了大量并发写入的性能，适合高负载应用。

2、WAL 支持：写前日志确保数据的一致性。

3、自定义合并和压缩策略：提供对数据合并和压缩的灵活控制，优化存储空间。

4、内存控制：能够在内存和磁盘之间平衡负载，适合高效的存储。

**- 适用场景：**

- 适用于高并发、高吞吐量的应用场景，如实时数据处理、嵌入式存储等。

**- 开源地址：**

- [Badger GitHub](https://github.com/dgraph-io/badger)

#### LMDB (Lightning Memory-Mapped Database)

**- 简介：**

- LMDB 是一个高效的内存映射键值存储数据库，不完全基于 LSM 树，但其高效的 B+ 树设计提供了类似 LSM 的快速读写性能。它是一个嵌入式数据库，适合单机高性能存储。

**- 特点：**

1、内存映射存储：使用内存映射文件（mmap），提供高效的读操作。

2、事务支持：支持 ACID 事务，保证数据的一致性。

3、高性能：特别适用于读取密集型应用，读取性能非常高。

**- 适用场景：**

- 适用于需要快速读取和 ACID 支持的场景，常见于配置存储、嵌入式系统等。

**- 开源地址：**

- [LMDB GitHub](https://github.com/LMDB/lmdb)

#### RocksDB

**- 简介：**

- RocksDB 是由 Facebook 开发的高性能嵌入式键值数据库，基于 LSM 树实现。它的性能在高写入负载和大规模数据存储上表现优秀，广泛用于分布式系统中。尽管它通常作为分布式系统的存储引擎使用，RocksDB 也可以作为单机存储引擎嵌入到应用中。

**- 特点：**

1、高效的写入性能：LSM 树结构使得 RocksDB 适用于高频写操作的场景。

2、压缩和合并：提供多种压缩算法和合并策略，优化存储空间。

3、事务支持：通过 Write-Ahead Log（WAL）和原子写入确保数据一致性。

**- 适用场景：**

- 用于高吞吐量、低延迟的数据存储和日志系统，适用于各种需要高效键值存储的场景。

**- 开源地址：**

- [RocksDB GitHub](https://github.com/facebook/rocksdb)

#### HyperLevelDB

**- 简介：**

- HyperLevelDB 是一个开源的键值存储系统，是 LevelDB 的一个高性能分支，使用 LSM 树结构进行存储。它的设计优化了 LevelDB 的性能，尤其在多核和多线程环境下的表现。

**- 特点：**

1、性能优化：在 LevelDB 基础上进行性能提升，特别在并发写入方面。

2、小巧高效：轻量级设计，适合嵌入式系统。

3、持久化和压缩：支持写前日志（WAL）和自定义的压缩算法。

**- 适用场景：**

- 嵌入式数据库、高性能缓存系统、日志存储等。

**- 开源地址：**

- [HyperLevelDB GitHub](https://github.com/google/leveldb)

#### 总结

对于单机 LSM 实现，以下是一些常见的开源项目：

- Pebble：适用于需要嵌入式存储和高效读写的小型应用，尤其是嵌入式数据库系统。

- LevelDB：轻量级、简单易用的存储引擎，适合小型应用和低资源消耗的嵌入式系统。

- Badger：高并发写入优化，适合高吞吐量、实时数据处理场景。

- LMDB：特别适合读取密集型应用，提供 ACID 支持。

- RocksDB：虽然一般用于分布式系统，但也可作为单机存储引擎，适合高吞吐量和大规模数据存储。

- HyperLevelDB：性能优化的 LevelDB 分支，适用于高性能需求的嵌入式应用。

选择合适的实现取决于你的应用场景、并发需求和资源限制。

### R Tree

## 查询

# 动态平衡技术

# 副本管理

# 基于MapReduce数据分析