# 1 数据库基础知识

## 数据库范式

数据库的设计范式是数据库设计所需要满足的规范。只有理解数据库的设计范式，才能设计出高效率、优雅的数据库，否则可能会设计出错误的数据库。

范式可以避免数据冗余，减少数据库的空间，减轻维护数据完整性的麻烦，但是操作困难，因为需要联系多个表才能得到所需要数据，而且范式越高性能就会越差。

关系的描述称为关系模式（Relation Schema）它可以形式化地表示为：

R（U，D，dom，F）

其中R为关系名，U为组成该关系的属性名集合，D为属性组U中属性所来自的域，dom为属性向域的[映象](http://baike.baidu.com/view/540438.htm" \t "_blank)集合，F为属性间数据的依赖关系集合。

通常简记为：

R(U)或R(A1，A2，…,An)

其中R为关系名，U为属性名集合，A1，A2，…,An为各属性名。

第一范式：如果关系模式R的每个关系R的属性都是不可分的数据项，那么就称R是第一范式的模式。每个属性原子项不可分割。

案例：student table

学生编号 姓名 性别 联系方式 地址

0001 chen 1 [xxxx@qq.com,188888888888](mailto:xxxx@qq.com,188888888888) 中国四川省成都市武侯区

第二范式：如果关系模式R是1NF，且每个非主键属性都完全依赖主键，那么R就是第二范式。

案例：student table 主键（学生、课程）

学生 课程 教师 教师职称 教材 教室 上课时间

Zhang java li 高级 corejava 303 15:00

第三范式：如果关系模式R是2NF，且每个非主键属性都独立其他非关键字列，并依赖于关键字，不能存在传递依赖。

案例：student table 主键（学生、课程）

学生 课程 教师 教师职称 教室 上课时间

Zhang java li 高级 303 15:00

总结：1NF原子性，2NF消除部分依赖，3NF消除传递依赖

## 事务特性

A---atomicity（原子性）: 一个事务（transaction）中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。

C---Consistency(一致性)：在事务开始之前和事务结束以后，数据库的完整性没有被破坏。

I---Isolation（隔离性）：当两个或者多个事务并发访问（此处访问指查询和修改的操作）数据库的同一数据时所表现出的相互关系。也就是说，事务内部的操作是不会被外部所看到的。事务隔离分为不同级别，包括读未提交（Read uncommitted）、读提交（read committed）、可重复读（repeatable read）和串行化（Serializable）。

D---durability（持久性）：在事务完成以后，该事务对数据库所作的更改便持久地保存在数据库之中，并且是完全的。

## 1.3 事务隔离与加锁

[并发控制](https://zh.wikipedia.org/w/index.php?title=Concurrency_control&action=edit&redlink=1)描述了数据库处理隔离以保证数据正确性的机制，以事务为单元，并发控制通常使用封锁机制，锁与多版本机制。

### 1.3.1数据库加锁方式

共享锁：如果事务T获得了数据A的共享锁，T对A可读不可写，其他事务只能对A加共享锁，不能加排他锁。

排他锁：如果事务T获得了数据A的排他锁，则T对A可写可读，其他事事务不能再对A进行加锁。

意向共享锁

意向排他锁

#### 1.3.1.1 一次封锁

一次封锁：在方法开始时预先知道会使用哪些数据，然后全部锁住，方法执行完毕再解锁。这种加锁方式可以避免循环死锁，但是在数据库不适用，事务开始时数据库不知道使用哪些数据。

两阶段锁：加锁阶段与解锁阶段，在加锁阶段，对读操作加共享锁，在进行写操作之前申请获取排他锁，加锁不成功事物进行等待，直到加锁成功。加锁阶段不能进行解锁，事务释放第一个锁后，事务进入解锁阶段，在该阶段只能进行解锁不能加锁。容易出现死锁，加锁阶段多次申请同一个锁。

死锁：T1 slockB xlockA

T2 slockA xlockA

#### 1.3.1.2 树形协议

树形协议：数据集合T{t1,t2,t3…..tn}满足一个偏序关系，访问数据按照偏序关系先后进行,如t1🡪t2，要访问t2得先访问t1。可以对任何数据项进行加锁，加锁顺序进行。解锁倒序进行。

规则：事务T第一次加锁对任何数据进行，此后事务T对数据A进行加锁需要先持有A的父亲数据项的锁。对数据解锁可以随时进行，事务T对数据A进行加锁解锁之后不能再被事务T加锁。

#### 1.3.1.3 时间戳排序协议

时间戳排序协议：每个事务都有一个唯一的时间戳，时间戳有大小，以此来保证事务可串行化。对于每个数据项A，有两个时间戳与之绑定，一个是W-TS(A)最近写事务时间戳,一个是R-TS(A)最近读时间戳。Thomas规则如下：

若事务T1发起一个write(A)，则

如果TS(T1)<R-TS(A)表明事务T1准备写入的值来不及写入A就被提前读取，拒绝TS的写入，操作回滚。

如果TS(T1)<W-TS(A)表明事务T1准备写入过期数据，其他数据已经写回到A上，拒绝T1的写入，操作回滚。

若事务T2发出read(B)：

如果TS(T2)<W-TS(B)，则T2需要读取的数据B已被覆盖，read操作拒绝，T2回滚

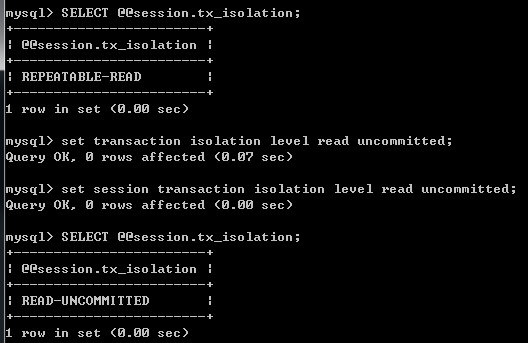
如果TS(T2)>=W-TS(B)，则执行read操作。

其他情况操作被允许。

#### 1.3.1.4 MVCC

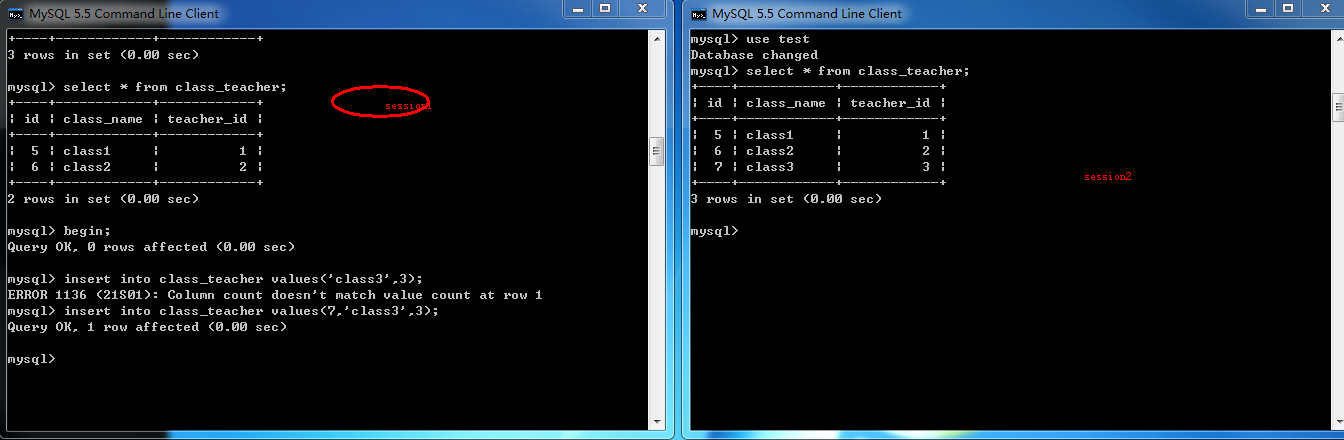
### 1.3.2事务隔离

在数据库事务ACID四个属性中，隔离性往往容易被忽略。隔离级别会影响数据库系统的锁机制或者多版本并发控制，数据库系统定义了不同的隔离级别来控制锁的程度，多数数据库事务都避免使用高等级的隔离级别（可序列化）从而减少系统开销。



未提交读：允许脏读，事务可以看到其他事务尚未提交的修改。

案例：



提交读：提交读这种隔离级别保证了读到的任何数据都是提交的数据，避免读到中间的未提交的数据。但是不保证事务重新读的时候能读到相同的数据，因为在每次数据读完之后其他事务可以修改刚才读到的数据。

如何避免不可重复读？

可重复读：同一事务的多个实例在并发读取数据时，会看到同样的数据行。对读锁和写锁会保持到事务结束。但是不要求范围锁，所以可能会造成幻影读。在事务执行过程中，当两个完全相同的查询语句执行得到不同的结果集称为幻影读。

可序列化：最高的隔离级别，在可重复读级别增加了一个范围锁。

**隔离级别vs读现象**

|  |  |  |  |
| --- | --- | --- | --- |
| **隔离级别** | **脏读** | **不可重复读** | **幻影读** |
| 未提交读 | 可能发生 | 可能发生 | 可能发生 |
| 提交读 | - | 可能发生 | 可能发生 |
| 可重复读 | - | - | 可能发生 |
| 可序列化 | - | - | - |

**隔离级别vs 锁持续时间**

在基于锁的并发控制中，隔离级别决定了锁的持有时间。**"C"**-表示锁会持续到事务提交。 **"S"** –表示锁持续到当前语句执行完毕。如果锁在语句执行完毕就释放则另外一个事务就可以在这个事务提交前修改锁定的数据，从而造成混乱。

|  |  |  |  |
| --- | --- | --- | --- |
| **隔离级别l** | **写操作** | **读操作** | **范围操作 (...where...)** |
| 未提交读 | S | S | S |
| 提交读 | C | S | S |
| 可重复读 | C | C | S |
| 可序列化 | C | C | C |

## 1.4 SQL主要组成部分

1.**DDL**（[**Data**](javascript:;)**Definition Language**）[**数据库**](javascript:;)**定义语言。**用于定义数据库的三级结构，包括外模式、概念模式、内模式及其相互之间的映像，定义数据的完整性、安全控制等约束。

CREATE、ALTER、DROP、COMMENT、RENAME

2.**DML**（**Data Manipulation Language**）**数据操纵语言。由DBMS提供，用于让用户或程序员使用，实现对数据库中数据的操作。**

SELECT、INSERT、UPDATE、DELETE

3.**DCL**（**Data Control Language**）**数据库控制语言**  授权，角色控制等。

4.**TCL**（**Transaction Control Language**）**事务控制语言**

## 1.5 数据库原理

### 1.5.1 数据库结构

### 1.5.2 数据存储

数据库数据存储最终是落在磁盘上。数据如何进行组织呢？最终涉及到核心概念映射，映射就是：使得对A中的每个元素a，按法则f，在B中有唯一确定的元素b与之对应，则称f为从A到B的映射，记作f：A→B。如何实现映射就涉及到相关数据结构。应证了：程序=算法+数据结构。

扩展知识点：磁盘结构特性。

**局部性原理**：**当一个数据被用到时，其附近的数据也通常会马上被使用。程序运行期间所需要的数据通常比较集中。所以操作系统会进行预读，磁盘与内存是以页为单位进行数据交换，所以尽可能预读到有效数据，合理组织数据结构减少IO开销**

### 1.5.1 数据库索引

**数据库索引**，是[数据库管理系统](https://zh.wikipedia.org/wiki/%E6%95%B0%E6%8D%AE%E5%BA%93%E7%AE%A1%E7%90%86%E7%B3%BB%E7%BB%9F)中一个排序的[数据结构](https://zh.wikipedia.org/wiki/%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84)，以协助快速查询、更新[数据库表](https://zh.wikipedia.org/wiki/%E6%95%B0%E6%8D%AE%E5%BA%93%E8%A1%A8)中数据。存储在磁盘。

索引优点：

第一，通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。

第二，可以大大加快数据的检索速度，这也是创建索引的最主要的原因。

索引缺点：

第一，创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

第二，索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

第三，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

不具备创建索引列特点：

第一，对于那些在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。

第二，对于那些只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。

第三，对于那些定义为text, image和bit数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少。

第四，当修改性能远远大于检索性能时，不应该创建索引。这是因为，**修改性能和检索性能是互相矛盾的**。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

索引分类：唯一索引、主键索引、聚集索引、非聚集索引

### 1.5.3 数据映射数据结构

有序数组、散列表、二叉查找树、平衡二叉查找树、红黑树、B树、B+树、**LSM**

。

### 1.5.4 如何选择数据结构

##### 1.5.4.4.1衡量指标

1、是否支持范围查找

2、是否自动扩展

3、读写性能指标

4、并行指标

5、是否面向磁盘结构

6、CPU占用

二叉查找树的结构不适合数据库，因为它的查找效率与层数相关。越处在下层的数据，就需要越多次比较。极端情况下，n个数据需要n次比较才能找到目标值。对于数据库来说，每进入一层，就要从硬盘读取一次数据，这非常致命，因为硬盘的读取时间远远大于数据处理时间，数据库读取硬盘的次数越少越好。

### 1.5.5 数据存储模型

行存储模型

列存储模型

行列结合类存储模型

文档类存储模型

json格式存储模型

column family类存储模型

图形存储模型

# 2 MySQL数据库

## 2.1 mysql 日志

### 2.1.1 binlog

MySQL binlog二进制数据库改变事件日志，表与数据的变化，查询日志不记录。主要用途用于用户数据同步与恢复。

### 3.1.2 relay-log

## 2.2 加锁分析

MySQL InnoDB存储引擎，实现的是基于多版本的并发控制协议——MVCC ，MVCC最大的好处读不加锁，读写不冲突。

在MVCC并发控制中，读操作可以分成两类：快照读 (snapshot read)与当前读 (current read)。快照读，读取的是记录的可见版本 (有可能是历史版本)，不用加锁。当前读，读取的是记录的最新版本，并且，当前读返回的记录，都会加上锁，保证其他事务不会再并发修改这条记录。

**快照读：**简单的select操作，属于快照读，不加锁。

select \* from table where ?;

**当前读：**特殊的读操作，插入/更新/删除操作，属于当前读，需要加锁。

select \* from table where ? lock in share mode;

select \* from table where ? for update;

insert into table values (…);

update table set ? where ?;

delete from table where ?;

如何加锁的？

**SQL1：**select \* from t1 where id = 10;

**SQL2：**delete from t1 where id = 10;

* **组合一：**id列是主键，RC隔离级别
* **组合二：**id列是二级唯一索引，RC隔离级别
* **组合三：**id列是二级非唯一索引，RC隔离级别
* **组合四：**id列上没有索引，RC隔离级别
* **组合五：**id列是主键，RR隔离级别
* **组合六：**id列是二级唯一索引，RR隔离级别
* **组合七：**id列是二级非唯一索引，RR隔离级别
* **组合八：**id列上没有索引，RR隔离级别
* **组合九：**Serializable隔离级别

锁表

## 2.3 并行度控制

MySQL并发控制Patch

## 2.4 数据复制

并行复制

# 3 数据库分区

# 4 数据架构方案

## 4.1 分库

分库：将不同的数据划分到不同数据库，解决数据库压力、数据容量问题。

分库方案：

1. 基于业务逻辑拆分，按功能模块，关系紧密的进行拆分，如将用户信息、订单、商品拆分为不同的业务数据库
2. 基于负载压力拆分：可能拆分后的数据库包含不同业务类型的数据表，日常维护会有一定的烦恼。
3. 安全拆分：高安全性数据与低安全性数据分库，这样的好处第一是便于维护，第二是高安全性数据的数据库参数配置可以以安全优先，而低安全性数据的参数配置以性能优先。
4. 混合拆分组合

基于安全拆分出A数据库实例，基于业务拆分出B,C数据库实例，C数据库存在较高负载，基于负载拆分为C1,C2,C3,C4等实例。

总结：去关联化，去除表之间的关联，反范式设计冗余字段。去关联带来显著的问题就是对一个数据库表修改，有影响的表进行修改。跨库事务控制？

## 4.2 分表

分表：数据量过大或者访问压力过大的数据表需要切分。先确定切分规则，再选择切分算法。

### 4.2.1 切分方式

1. 横向切分表：按数据分散到不同的表

* 等分切分表，按照HASH、取余、奇偶切分算法。等分切分方式数据分布均匀，无法对达到上限的节点不再写入数据。当数据容量达到限制的时候扩容非常麻烦，需要重新切分数据和转表。
* 递增切分表，比如每1KW数据新增一个表，可以适应数据的增长趋势而来，但是带来问题就是热点数据问题，最新数据负载过高，压力分配不均匀。
* 日期切分表，按时间段进行数据划分，相当于递增。
* 映射切分：切分数据ID与分表分库建立映射关系，查询数据首先查询映射关系再路由到真正数据库表进行查询。涉及多一次查询性能开销，可以针对映射关系进行缓存。
* 混合切分：一级时间增量切分+二级HASH切分

1. 纵向切分表：忙闲切分，单表数据字典过多，将更新频繁的数据字典与非更新频繁的数据字段进行拆分。
2. 热点数据分表：将数据量较大的读写频繁的数据抽取出来，单独存放形成热点数据表(缓存)，减少整体系统压力负载。

## 4.3 会带来哪些难以解决问题？

1. 跨表跨库JOIN、分组、排序
2. 分区主键全局唯一
3. 事务
4. 扩容

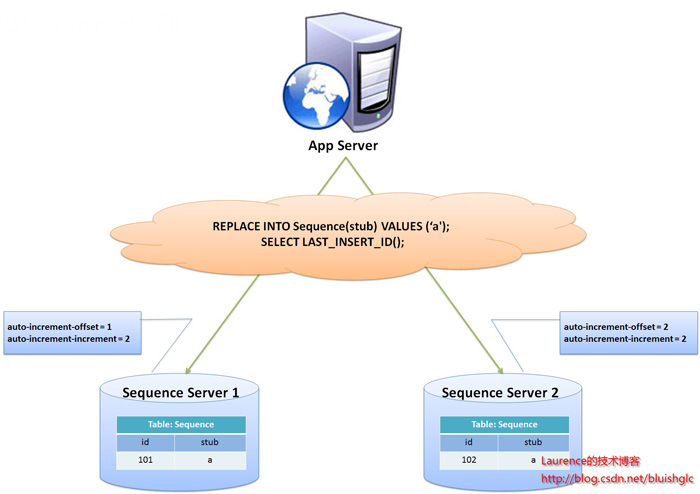
如何支持JOIN？

1. SQL最好不写JOIN，通过程序关联查询聚合
2. 建立冗余表，小表广播

全局ID方案：

1. UUID
2. 时间戳+机器ID+PID+计数
3. **Flickr:**巧妙地使用了mysql的自增ID，及replace into语法

[**http://code.flickr.net/2010/02/08/ticket-servers-distributed-unique-primary-keys-on-the-cheap/**](http://code.flickr.net/2010/02/08/ticket-servers-distributed-unique-primary-keys-on-the-cheap/)



1. snowflake：时间+机器+序列数(ZK)

分库分表需要满足哪些要求才是好的全局ID？

1. 不能单点故障
2. 有序减少一个索引
3. 可以控制分片
4. 不能太长
5. 是否具有业务含义

如何重新设计ID？

系统编号+时间戳+分表位+序列数+校验位，分表位要预留扩容需求。

序列数采取批量生成按步长一次取一段缓存到客户端。

如何扩容，其实扩容与路由算法息息相关？

何为理想扩容：满足数据少迁移，负载均匀分布读写、硬件动态扩容、已达到上限节点不再写入数据。

Hash算法扩容？

直接扩展库表，无法避免再次写入数据量已经达到极限的表

增量算法扩容？

扩展数据库，再扩展表，无数据迁移。

支持动态扩容切分算法

基于散列的路由能均匀地分布数据，但却需要数据迁移，同时也无法避免对达到上限的节点不再写入新数据；基于增量区间的路由天然不存在数据迁移和向某一节点无上限写入数据的问题，但却存在“热点”困扰。我们设计方案的初衷就是希望能结合两种路由规则的优势，摒弃各自的劣势，创造出一种接近“理想”状态的扩容方式，而这种方式简单概括起来就是：全局按增量区间分布数据，使用增量扩容，无数据迁移，局部使用散列方式分散数据读写，解决“热点”问题。

## 4.4 高可用切分实现

### 4.4.1一致性Hash

一致性Hash：是一种特殊的哈希算法。在使用一致哈希算法后，哈希表槽位数（大小）的改变平均只需要对K/n 个关键字重新映射，其中 K是关键字的数量，n是槽位数量。然而在传统的哈希表中，添加或删除一个槽位的几乎需要对所有关键字进行重新映射。主要用于负载均衡。

判断哈希好坏标准：

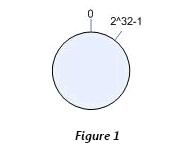
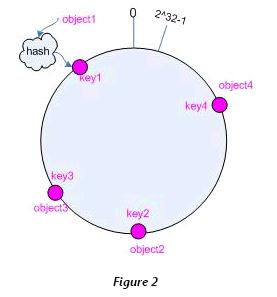
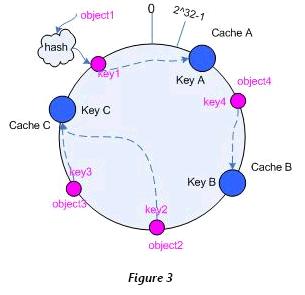
1、平衡性(Balance)：平衡性是指哈希的结果能够尽可能分布到所有的缓冲中去，这样可以使得所有的缓冲空间都得到利用。很多哈希算法都能够满足这一条件。

2、单调性(Monotonicity)：单调性是指如果已经有一些内容通过哈希分派到了相应的缓冲中，又有新的缓冲加入到系统中。哈希的结果应能够保证原有已分配的内容可以被映射到原有的或者新的缓冲中去，而不会被映射到旧的缓冲集合中的其他缓冲区。

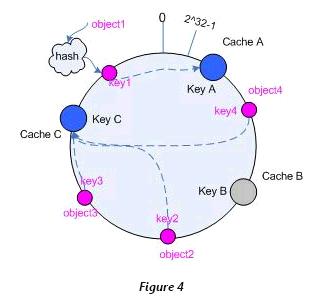
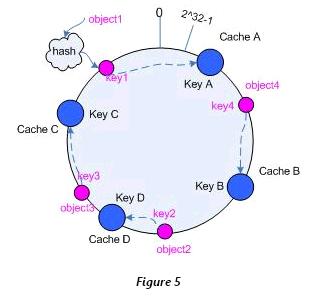
3、分散性(Spread)：在分布式环境中，终端有可能看不到所有的缓冲，而是只能看到其中的一部分。当终端希望通过哈希过程将内容映射到缓冲上时，由于不同终端所见的缓冲范围有可能不同，从而导致哈希的结果不一致，最终的结果是相同的内容被不同的终端映射到不同的缓冲区中。这种情况显然是应该避免的，因为它导致相同内容被存储到不同缓冲中去，降低了系统存储的效率。分散性的定义就是上述情况发生的严重程度。好的哈希算法应能够尽量避免不一致的情况发生，也就是尽量降低分散性。

4、负载(Load)：负载问题实际上是从另一个角度看待分散性问题。既然不同的终端可能将相同的内容映射到不同的缓冲区中，那么对于一个特定的缓冲区而言，也可能被不同的用户映射为不同的内容。与分散性一样，这种情况也是应当避免的，因此好的哈希算法应能够尽量降低缓冲的负荷。

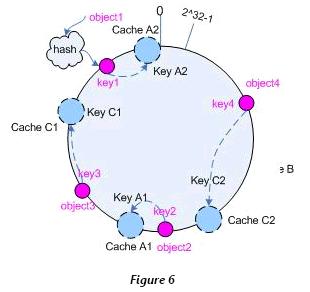
实现原理：

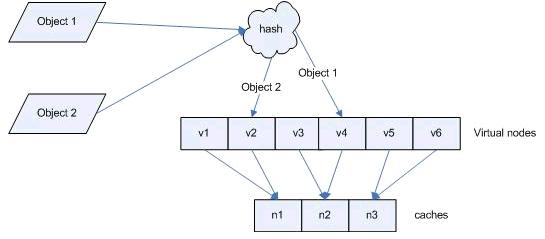
  

添加或者删除节点

虚拟节点





案例分析redis java客户端jedis

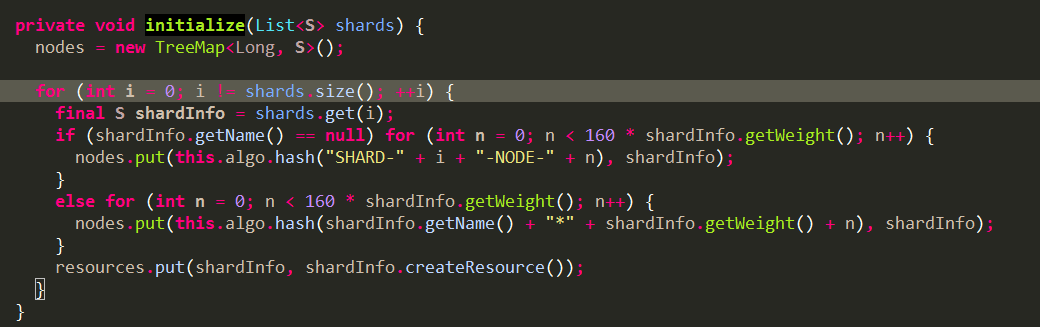
Sharded类

数据结构TreeMap（红黑树）+ LinkedHashMap,TreeMap存储虚拟节点信息

LinkedHashMap存储虚拟节点对应的真实节点信息。

**定位的流程**先在TreeMap中找到对应key所对应的ShardInfo，然后通过ShardInfo在LinkedHashMap中找到对应的Jedis实例。





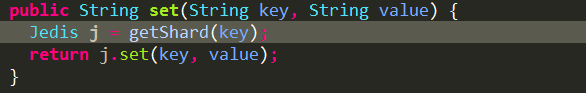
权重越高被命中率就越高

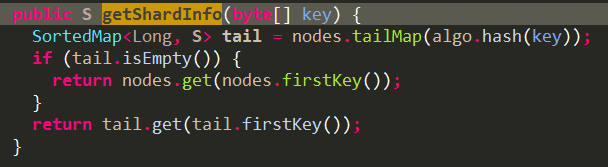
虚拟节点个数=逻辑区间\*权重

使用虚拟节点hash

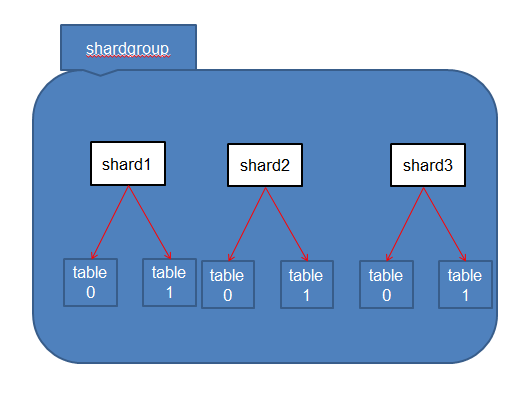
Hash采用md5

数据存放流程



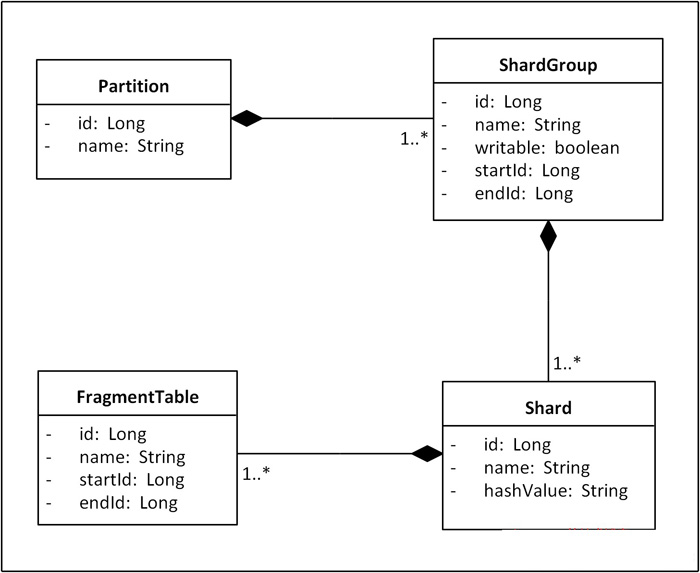


### 4.4.2 混合切分路由模型



实现逻辑：全局上，数据按增量区间分布，但是每个增量区间并不是按照某个Shard的存储规模划分的，而是根据一组Shard的存储总量来确定的，我们把这样的一组Shard称为一个ShardGroup，局部上，也就是一个ShardGroup内，记录会再按散列方式均匀分布到组内各Shard上。

数据库实体抽象关系：



分区：主要是垂直拆分的相对独立的库。

分片组：全局按增量存储的，一个分组存储多少数据量

分片：每个shard按存储总量进行划分的单个存储增量区间。

分段：每个表进行存储，达到单表上线再以递增编号方式进行创建

## 4.5 反范式设计

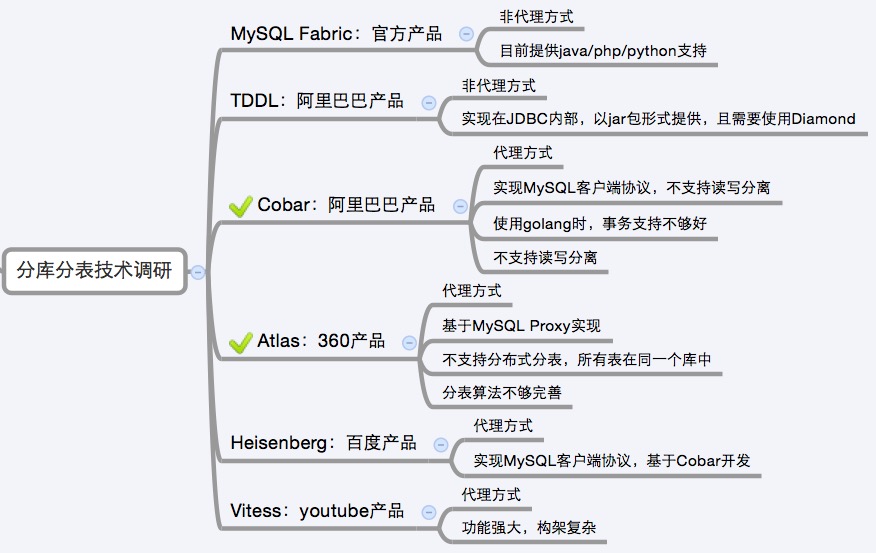
反范式设计实际就是冗余结构设计，核心概念无外键设计、无需遵守范式设计。

## 4.6 分库分表方案对比

Mysql 中间代理分库路由

TDDL

Cobar



## 4.7 在哪里实现Sharing

DAO、JDBC、Spring事务封装

# 5 分布式设计

## 5.1 分布式理论

### 5.1.1 CAP理论

* 一致性 ( **C**onsistency) ：任何一个读操作总是能读取到之前完成的写操作结果；
* 可用性 ( **A**vailability) ： 每一个操作总是能够在确定的时间内返回；
* 分区可容忍性 (Tolerance of network **P**artition) ：在出现网络分裂的情况下，仍然能够满足一致性和可用性；

如何证明CAP只能满足其中两者？

### 5.1.2 一致性模型

1. **弱一致性（Weak）**：当你写入一个新值后，读操作在各个数据副本上不保证能读出最新值。比如：某些cache系统，网络游戏其它玩家的数据和你没什么关系。
2. **最终一致性（Eventually）**：Eventually 是 Weak 的一种特殊情况。当你写入一个新值后，有可能读不出来，但在某个时间窗口之后保证最终能读出来。比如：DNS，电子邮件、Amazon S3，Google搜索引擎这样的系统。
3. **强一致性（Strong）**：新的数据一旦写入，在任意副本任意时刻都能读到新值。比如：文件系统，RDBMS，Azure Table都是强一致性的。

### 5.1.3 最终一致性模型变体

* **因果一致性**。如果进程A通知进程B它已更新了一个数据项，那么进程B的后续访问将返回更新后的值，且一次写入将保证取代前一次写入。与进程A无因果关系的进程C的访问遵守一般的最终一致性规则。
* **“读己之所写（read-your-writes）”一致性**。这是一个重要的模型。当进程A自己更新一个数据项之后，它总是访问到更新过的值，绝不会看到旧值。这是因果一致性模型的一个特例。
* **会话（Session）一致性**。这是上一个模型的实用版本，它把访问存储系统的进程放到会话的上下文中。只要会话还存在，系统就保证“读己之所写”一致性。如果由于某些失败情形令会话终止，就要建立新的会话，而且系统的保证不会延续到新的会话。
* **单调（Monotonic）读一致性**。如果进程已经看到过数据对象的某个值，那么任何后续访问都不会返回在那个值之前的值。
* **单调写一致性**。系统保证来自同一个进程的写操作顺序执行。要是系统不能保证这种程度的一致性，就非常难以编程了。

### 5.1.4 一致性权衡策略

一致性不是系统最优先考虑的事情，不一致是可以容忍的，这有两个理由：一是可以在高并发条件下提高读写性能；二是处理一些分区状况——多数决模型（majority model）有可能使系统的一部分表现为不可用，虽然那些节点正运行良好。

不一致是否可接受取决于客户应用程序。真正重要的是“用户感知到的一致性”，也就是让不一致窗口——即“更新发生时刻到任何观察者都一定能观察到更新后数据的时刻之间的时间段”——“小于顾客对下一页面加载时间的期待”，这样更新就可在预期发生下一次读取的时刻之前传播到整个系统。

看待一致性可以从两个角度：一种是从开发者/客户端的角度；他们如何观察数据更新。第二种是从服务器的角度；更新如何流经整个系统，系统对更新有何保证。

### 5.1.3 BASE模型

BASE模型反ACID模型，完全不同ACID模型，牺牲高一致性，获得可用性或可靠性.主要包括以下几点：  
 1、Basically Available基本可用。支持分区失败(e.g. sharding碎片划分数据库)  
 2、Soft state软状态 状态可以有一段时间不同步，异步。  
 3、Eventually consistent最终一致，最终数据是一致的就可以了，而不是时时高一致。

### 5.1.4 短板理论

一个集群如果出现了负载不均衡问题，那么负载最大的机器往往将成为影响系统整体表现的瓶颈和短板。为了避免这种情况的发生，需要动态负载均衡机制，以达到实时的最大化资源利用率，从而提升系统整体的吞吐。

### 5.1.5 CAC

[**Consistency、Availability、Convergence**](http://www.cs.utexas.edu/users/dahlin/papers/cac-tr.pdf)

收敛性是指一种实现能力，它能确保被一个节点写入的数据被另外一个读取，收敛性的定义是：描述的是一个节点能够读取到其他节点的写入时的一系列环境条件(如网络，本地时钟等)。

### 5.1.6 COPS

<https://www.cs.cmu.edu/~dga/papers/cops-sosp2011.pdf>

## 5.2 一致性策略

### 5.2.1 Paxos

### 5.2.2 Quorum协议

**Quorom** 机制，是一种分布式系统中常用的，用来保证数据冗余和最终一致性的投票算法，其主要数学思想来源于[鸽巢原理](https://zh.wikipedia.org/wiki/%E9%B8%BD%E5%B7%A2%E5%8E%9F%E7%90%86" \o "鸽巢原理)。也被称为NWR策略。

该协议有三个参数，W是更新一个数据对象时需要确保成功更新的票数；R代表读取一个数据需要读取的票数，如果系统有N票，那么读写票必须满足以下规则：

公式1 W+R>N

公式2 W>N/2

公式1保证同一条数据不会被同时读写，当写的请求为W时，R<N-W不满足读需要的最低票数，所以不能进行读取操作。

公司2保证数据串行化修改，一份数据的冗余拷贝不可能同时被两个写请求修改。多个进程针对同一份数据若满足了以上模型设置，那么必然两个进程对同一个冗余数据必然存在竞争，竞争失败的写不可用。

读写效率要求不同，WR参数配置不同，可以调节。

案例推敲

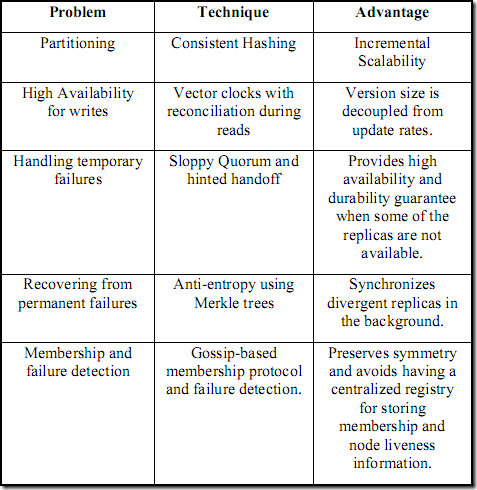
当W = 1，R = N时，系统对写操作有较高的要求，但读操作会比较慢，若N个节点中有节点发生故障，那么读操作将不能完成。   
　　 当R = 1，W = N时，系统要求读操作高性能、高可用，但写操作性能较低，用于需要大量读操作的系统，若N个节点中有节点发生故障，那么写操作将无法完成。   
　　 当R = W = N / 2 + 1时，系统在读写性能之间取得了平衡，兼顾了性能和可用性， 采用的就是平衡策略即N=3，W=2，R=2

存在哪些问题？

1. 若节点挂掉此时N变小，怎么办？挂了节点恢复之后再加入，数据一致么？
2. 多IDC策略又该如何？
3. 有无中心？无法判断节点数据是否同步？数据冲突怎么解决？
4. 无中心节点间如何通讯？

如何实现该协议算法呢？Dynamo

Dynamo设计的时候遇到的问题和解决问题采用的技术如下图：



第一个是分区，采用一致性hash

第二个是向量时钟来做版本控制：用一个向量（比如说[a,1]表示这个数据在a节点第一次写入）来标记数据的版本，这样在有版本冲突的时候，可以追溯到出现问题的地方。这可以使数据的最终一致成为可能。（Cassandra未用vector clock，而只用client timestamps也达到了同样效果。）

第三个是引入Quorum协议

第四个是Merkle tree来提速数据变动时的查找：使用Merkle tree为数据建立索引，只要任意数据有变动，都将快速反馈出来。

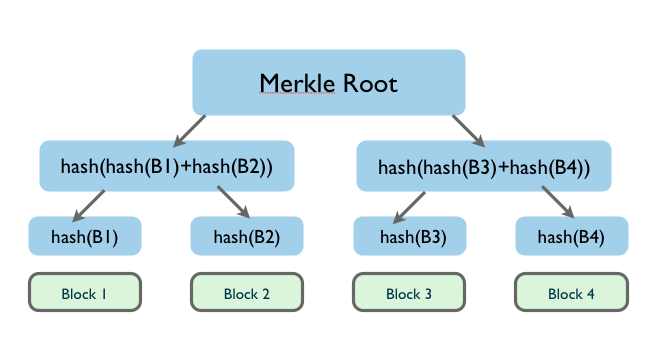
第五个是Gossip协议：一种通讯协议，目标是让节点与节点之间通信，省略中心节点的存在，使网络达到去中心化。提高系统的可用性。

第六个是hinted handoff数据的加入：在一个节点出现临时性故障时，数据会自动进入列表中的下一个节点进行写操作，并标记为handoff数据，在收到通知需要原节点恢复时重新把数据推回去。这能使系统的写入成功大大提升。

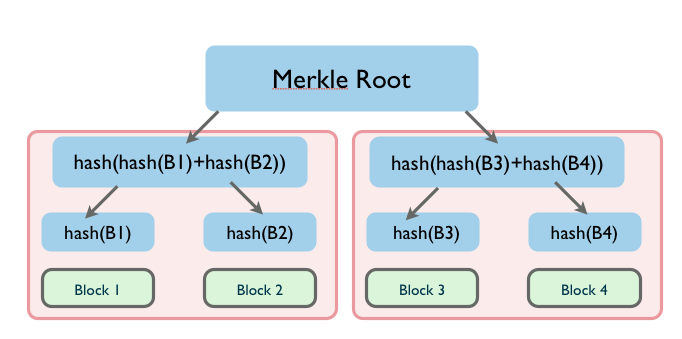
#### 5.2.2.1 merkle Tree

Merkle Tree它是一种树，而且是存储哈希值的树，树中只有叶子节点才是真正的存储值的，其它的非叶子节点都是根据孩子的值进行哈希计算得来的。说说用途，Merkle tree可以用来进行大数据的比对，可以快速定位(O(logn))到哪一部分数据不一致，在分布式环境下可以减少数据的传输量。

原理：我们把数据分成小的数据块，有相应地哈希和它对应。但是往上走，并不是直接去运算根哈希，而是把相邻的两个哈希合并成一个字符串，然后运算这个字符串的哈希，这样每两个哈希就结婚生子，得到了一个”子哈希“。如果最底层的哈希总数是单数，那到最后必然出现一个单身哈希，这种情况就直接对它进行哈希运算，所以也能得到它的子哈希。于是往上推，依然是一样的方式，可以得到数目更少的新一级哈希，最终必然形成一棵倒挂的树，到了树根的这个位置，这一代就剩下一个根哈希了，我们把它叫做 Merkle root.



算法优点：可以单独拿出一个分支来（作为一个小树）对部分数据进行校验，这个很多使用场合就带来了哈希列表所不能比拟的方便和高效。



#### 5.2.2.2 hinted handoff（提示移交）

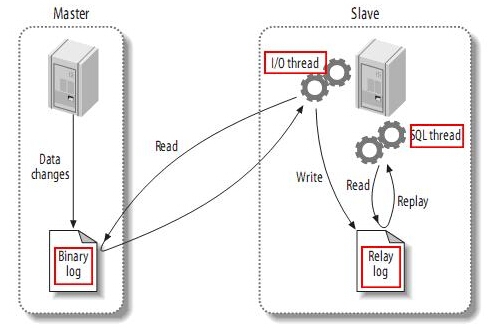
如果这些节点中的一个宕机了,会发生什么呢?写操作稍后将如何传递到此节点呢?有一个叫提示移交(Hinted Handoff)的技术来解决此问题,其中数据会被写入并保存到另一个随机节点X,并提示这些数据需要被保存到节点Y,并在节点重新在线时进行重放(记住,当节点Y变成在线时,快速通知X节点).提示移交可以确保节点Y可以快速的匹配上集群中的其他节点.注意,如果提示移交由于某种原因没有起作用,读修复最终仍然会“修复”这些过期数据，不过只有当客户端访问这些数据时才会进行读修复.

### 5.2.3 Raft

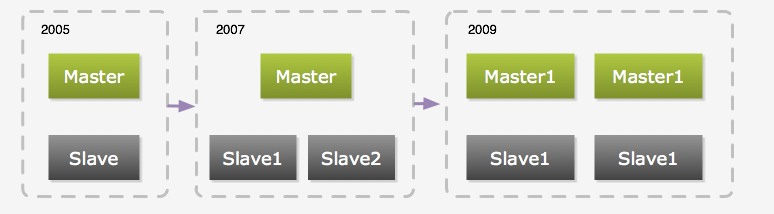
## 5.3 数据复制策略

数据复制主要包括两类：数据分区与数据镜像

#### 5.3.1 主从

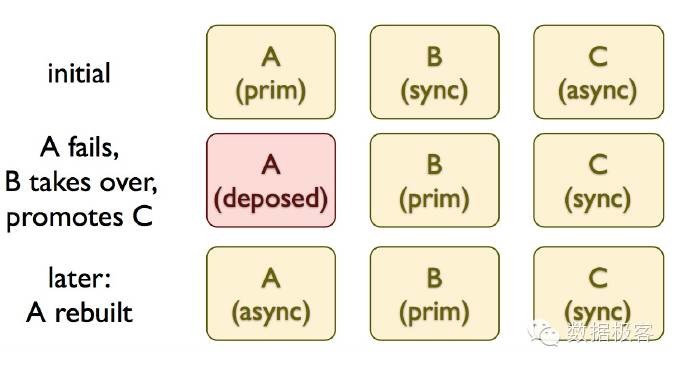


#### 5.3.2 主主



#### 5.3.3 manatee

Joyent的PostgreSQL高可用方案manatee，一个主节点带一个同步复制的从节点，以及若干异步复制从节点，当主节点挂了之后，同步复制节点被选举为主节点，异步复制节点选举一个提升为同步复制，而此前挂掉的节点恢复之后首先加入异步复制集群。



#### 5.3.4 MaxScale

#### 5.3.5 复制状态机

#### 5.3.6 链式复制

## 5.4 数据冲突解决策略

#### 5.4.1 last-writer-win

#### 5.4.2 vector clock（失量时钟）算法

矢量时钟实际上是一个(node,counter)对列表(即(节点，计数器)列表)。矢量时钟是与每个对象的每个版本相关联。通过审查其向量时钟，我们可以判断一个对象的两个版本是平行分枝或有因果顺序。如果第一个时钟对象上的计数器在第二个时钟对象上小于或等于其他所有节点的计数器，那么第一个是第二个的祖先，可以被人忽略。否则，这两个变化被认为是冲突，并要求协调。

下面，我们来看一个操作序列：



1）一个写请求，第一次被节点A处理了。节点A会增加一个版本信息(A，1)。我们把这个时候的数据记做D1(A，1)。 然后另外一个对同样key的请求还是被A处理了于是有D2(A，2)。这个时候，D2是可以覆盖D1的，不会有冲突产生。

2）现在我们假设D2传播到了所有节点(B和C)，B和C收到的数据不是从客户产生的，而是别人复制给他们的，所以他们不产生新的版本信息，所以现在B和C所持有的数据还是D2(A，2)。于是A，B，C上的数据及其版本号都是一样的。

3）如果我们有一个新的写请求到了B结点上，于是B结点生成数据D3(A,2; B,1)，意思是：数据D全局版本号为3，A升了两新，B升了一次。这不就是所谓的代码版本的log么？

4）如果D3没有传播到C的时候又一个请求被C处理了，于是，以C结点上的数据是D4(A,2; C,1)。

5）好，最精彩的事情来了：如果这个时候来了一个读请求，我们要记得，我们的W=1 那么R=N=3，所以R会从所有三个节点上读，此时，他会读到三个版本：

* + A结点：D2(A,2)
  + B结点：D3(A,2;  B,1);
  + C结点：D4(A,2;  C,1)

6）这个时候可以判断出，D2已经是旧版本（已经包含在D3/D4中），可以舍弃。

7）但是D3和D4是明显的版本冲突。于是，交给调用方自己去做版本冲突处理。就像源代码版本管理一样。

Vector Clocks 存储留一定的空间，造成性能和资源的消耗加剧，实际使用效果并不佳。

#### 5.4.3 Rev Tree

#### 5.4.4 区块链

#### 5.4.5 读修复

#### 5.4.6 CRDT

CRDT是Conflict-Free Replicated Data Types的缩写，直译的话即“无冲突可复制数据类型”。

通俗的讲是指能够避免冲突的可复制的数据结构。

论文

<http://hal.upmc.fr/file/index/docid/555588/filename/techreport.pdf>

实现该数据结构的项目：Akka、Riak

<https://github.com/jboner/akka-crdt>

<http://docs.basho.com/riak/latest/theory/concepts/crdts>

国内文章

<http://www.jdon.com/artichect/how-to-build-a-distributed-counter.html>

<http://www.jdon.com/artichect/crdt.html>

## 5.5 分布式事务

平面分布式事务和嵌套分布式事务

分布式事务的并发控制

时间戳并发控制

乐观并发控制

事务恢复

数据一致性

一致性策略分析

弱一致性读写

强一致性写

两段提交改协议的问题

三段提交改协议的分析

PAXOS

少数服从多数

无主机活锁的避免

自动master选择策略

强一致性读(W+R>N)

## 5.6 典型系统分析

### 5.6.1 Yahoo! PNUTS

<http://blog.sina.com.cn/s/blog_5374d6e30100sl13.html>

<http://timyang.net/architecture/yahoo-pnuts/>

### 5.6.2 cassandra

<http://dongxicheng.org/nosql/cassandra-strategy/>

### 5.6.3 MongoDb

### 5.6.4 GFS与BigTabe

### 5.6.5 Hbase

## 5.8 负载均衡算法

# 6 实现分布式数据库中间件

## 6.1 路由模型

## 6.2 容灾模块

并行复制

## 6.3 SQL解析

### 6.3.1 解析过程

### 6.3.2 SQL TopN过滤审查

### 6.3.3 超慢SQL拦截

## 6.4 JDBC

## 6.5 执行计划优化

* [基于规则的优化]
* [基于开销分析的优化]
* [分布式/并行优化]
* [延迟计算]
* [案例分析：FoundationDB, Apache Phoenix]
* [案例分析：Google F1, GreenPlum]

## 6.6 反向索引

## 6.7 并发度流控

## 6.8 读写负载均衡

## 6.9 切分规则变更推送

## 6.10 数据合并

小表广播

分布式Join算法

合并排序

Hash分片Join

# 7 中间件监控

## 7.1 连接池管理

## 7.2 JVM监控

## 7.3 慢SQL监控

## 7.4 SQL报警