MySQL架构设计

小書匠

目录

一、MySQL架构设计

1. 大表和大事务

1-1. 大表带来的问题

- 记录行数巨大，单表超过千万行
- 表数据文件巨大，表数据文件超过10G

1.大表对查询的影响，很难在一定的时间内过滤出所需要的数据
2.建立索引需要很长时间，MySQL版本<5.5时，建立索引会锁表；MySQL版本>=5.5，建立索引时虽然不会锁表但会引起主从延迟
3.修改表结构需要长时间锁表，造成长时间的主从延迟

1-1-1. 如何处理数据库中的大表

1. 分库分表把一张大表分成多个小表(影响前后端业务)
   难点：
   - 分表主键的选择
   - 分表后跨分区数据的查询和统计
2. 大表的历史数据归档(减少对前后端业务的影响)
   难点：
   - 归档时间点的选择
   - 如何进行归档操作

1-2. 大事务带来的功能

运行时间比较长，操作的数据比较多的事务，锁定太多数据，造成大量的阻塞和锁超时；回滚时所需的时间比较长；执行时间长，容易造成主从延迟

1-2-1. 如何处理大事务

1. 避免一次处理太多数据
2. 移出不必要在事务中的SELECT操作(事务中基本是写操作)

1-2-2. 事务的原子性(A

整个事务中的所有操作要么全部提交成功，要么全部回滚失败

1-2-3. 事务的一致性(C

一致性是指事务将数据库从一种一致性状态转换到另一种一致性状态，在事务开始之前和事务结束之后数据库中数

据的完整性没有被破坏。
举个栗子，小A从有2000元的A账户转移到B账户，A账户没了2000元，B账户多了2000元，但是始终小A的账户下都只有2000元，这也就是数据的完整性。

1-2-4. 事务的隔离性(I

隔离性要求一个事务对数据库中的数据修改，在未提交完成前对于其他事务是不可见的
SQL标准中定义的四种隔离级别：

- 未提交读--脏读
- 读已提交--又称为不可重复读--虚读
- 重复读--幻读
- 序列化

读提交和可重复读的区别：读已提交是事务A先读取了数据，事务B紧接着更新了数据并提交了事务，而事务A再次读取该数据时，数据已经发生改变，造成了不可重复读，又称为虚读;可重复读是事务A读取数据，事务B以插入或删除行等方式更新数据来修改事务A的结果集，然后提交，事务A读到的数据仍然是原来事务B修改前的数据，实际上就是事务A在读取数据时，事务B不能修改其数据，又称为幻读

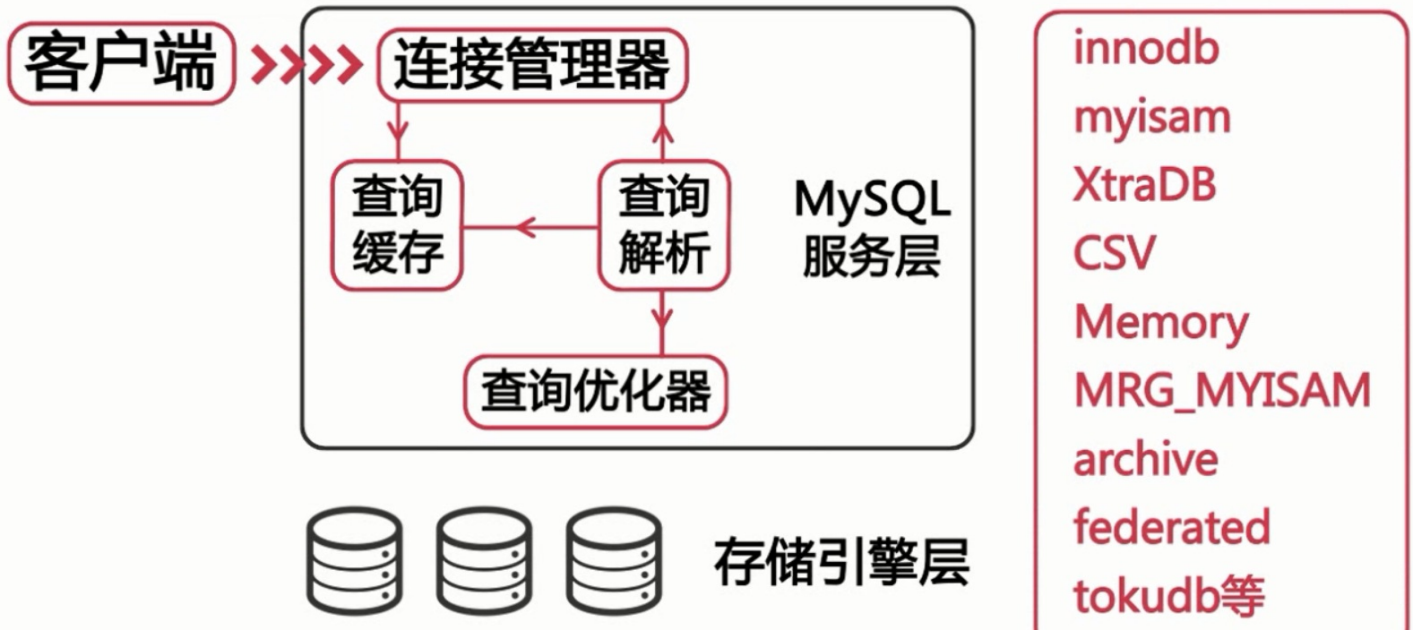
1-2-5. 事务的持久性(D

一旦事务提交，则其所做的修改就会永久保存到数据库中

---

2. MySQL体系结构

MySQL最特别之处在于插件式存储引擎
存储引擎是针对于表的，而不是针对于库的，不同表可以有不同的存储引擎



2-1. 存储引擎之MyISAM

MyISAM是MySQL5.5之前版本默认存储引擎

MyISAM仍然用于MySQL中的系统表和临时表等

`临时表:在排序、分组等操作中，当数量超过一定的大小之后，由查询优化器建立的临时表`

(ps:frm是MySQL所有引擎都有的文件，用于存储表结构)

2-1-1. 特性

- MyISAM使用的是表级锁，对于读写混合的并发性支持不够
- 由于MyISAM不支持事务
- MyISAM表支持数据压缩-myisampack(压缩之后的表只能进行读操作)

2-2. 存储引擎之Innodb

Innodb是MySQL5.5之后版本默认存储引擎

Innodb使用表空间进行数据存储

2-2-1. 特性

- Innodb是一种事务性存储引擎
- 完全支持事务的ACID特性
- Redo Log(支持事务持久性) 和Undo Log(支持事务回滚)
- 支持行级锁(行级锁是由存储引擎层实现的)

2-2-1-1. MySQL中锁的类型

Innodb中读锁和写锁都是行锁

- 共享锁(读锁)
- 独占锁(写锁)

	写锁	读锁
写锁	不兼容	不兼容
读锁	不兼容	兼容

2-2-1-2. MySQL中锁的粒度

锁的粒度实际上就是锁的最小单位

- 表级锁 `lock table 表名; /unlock table 表名`
- 行级锁

3. MySQL基准测试

基准测试可以简单认为是针对系统设置的一种压力测试

- 压力测试需要针对不同主题，所使用的数据和查询也是真实用到的
- 基准测试不关心业务逻辑，所使用的查询和业务的真实性可以和业务环境没关系

3-1. 基准测试工具-mysqlslap

3-1-1. 常用参数说明

```
--auto-generate-sql 由系统自动生成SQL脚本进行测试
--auto-generate-sql-add-autoincrement 在生成的表中增加自增ID
--auto-generate-sql-load-type 指定测试中使用的查询类型
--auto-generate-sql-write-number 指定初始化数据时生成的数据量
--concurrency 指定并发线程的数量
--engine 指定要测试表的存储引擎，可以用逗号分隔多个存储引擎
--no-drop 指定不清理测试数据
--iterations 指定测试运行的次数
--number-of-queries 指定每一个线程执行的查询数量
--debug-info 输出CPU和内存信息
--number-int-cols 指定测试表中包含INT类型列的数量
--number-char-cols 指定测试表中包含varchar类型列的数量
--create-schema 指定了用于执行测试的数据库的名字
--query 指定自定义SQL的脚本
--only-print 并不运行测试脚本，而是把生成的脚本打印出来
--help 查询可用命令
```

4. 数据库结构优化

- 减少数据冗余，数据冗余就是相同的数据在多处存在
- 尽量避免数据维护中出现更新、插入和删除异常
  - 插入异常：如果表中的某个实体随着另一个实体而存在
  - 更新异常：如果更改表中的某个实体的单独属性时，需要对多行进行更新
  - 删除异常：如果删除表中的某一实体则会导致其他实体的消失

4-1. 数据库设计范式

数据库的设计范式是数据库设计所需要满足的规范，满足这些规范的数据库是简洁的、结构明晰的，同时，不会发生插入（insert）、删除（delete）和更新（update）操作异常。反之则是乱七八糟，不仅给数据库的编程人员制造麻烦，而且面目可憎，可能存储了大量不需要的冗余信息

4-1-1. 第一范式

简而言之，第一范式就是无重复的列

- 数据库表中所有字段都只具有单一属性
- 单一属性的列是由基本的数据类型所构成的
- 设计出来的表是简单的二维表

如下的数据库表是符合第一范式的：

字段1	字段2	字段3	字段4

而这样的数据库是不符合第一范式的：

字段1	字段2	字段3	字段4
		字段3.1	字段4.1

4-1-2. 第二范式

第二范式要求数据库表中的每个实例或行必须可以被唯一地区分，为实现区分通常需要为表加上一个列，以存储每个实例的惟一标识，这个惟一属性称为主键

例如员工信息表中加上了员工编号（emp_id）列，因为每个员工的员工编号是惟一的，因此每个员工可以被惟一区分

简而言之，第二范式（2NF）就是非主属性完全依赖于主关键字

- 所谓完全依赖是指不能存在仅依赖主关键字一部分的属性（设有函数依赖W→A，若存在XW，有X→A成立，那么称W→A是局部依赖，否则就称W→A是完全函数依赖）。如果存在，那么这个属性和主关键字的这一部分应该分离出来形成一个新的实体，新实体与原实体之间是一对多的关系
- 要求一个表中只具有一个业务主键，也就是说符合第二范式的表中不能存在非主键列只对部分主键的依赖，**简单理解就是不能有复合主键**

4-1-2-1. 举个栗子

假定选课关系表为SelectCourse(学号，姓名，年龄，课程名称，成绩，学分)，关键字为组合关键字(学号，课程名称)，因为存在如下决定关系：
(学号，课程名称) → (姓名，年龄，成绩，学分)

这个数据库表不满足第二范式，因为存在如下决定关系：即存在组合关键字中的字段决定非关键字的情况

- (课程名称) → (学分)
- (学号) → (姓名，年龄)

由于不符合2NF，这个选课关系表会存在如下问题：

1. 数据冗余:
   - 同一门课程由n个学生选修，"学分"就重复n-1次；同一个学生选修了m门课程，姓名和年龄就重复了m-1次
2. 更新异常：
   - 若调整了某门课程的学分，数据表中所有行的"学分"值都要更新，否则会出现同一门课程学分不同的情况
3. 插入异常：
   - 假设要开设一门新的课程，暂时还没有人选修。这样，由于还没有"学号"关键字，课程名称和学分也无法记录入数据库
4. 删除异常：
   - 假设一批学生已经完成课程的选修，这些选修记录就应该从数据库表中删除。但是，与此同时，课程名称和学分信息也被删除了。很显然，这也会导致插入异常

4-1-3. 第三范式

第三范式要求一个数据库表中不包含已在其他表中已包含的非关键字信息

例如，存在一个部门信息表，其中每个部门有部门编号（dept_id）、部门名称、部门简介等信息。那么在的员工信息表中列出部门编号后就不能再将部门名称、部门简介等与部门有关的信息再加入员工信息表中。如果不存在部门信息表，则根据第三范式（3NF）也应该构建它，否则就会有大量的数据冗余

4-1-3-1. 举个栗子

所谓传递函数依赖，指的是如果存在"A → B → C"的决定关系，则C传递函数依赖于A
因此，满足第三范式的数据库表应该不存在如下依赖关系：
关键字段 → 非关键字段x → 非关键字段y(可以依赖关键字段)
假定学生关系表为Student(学号，姓名，年龄，所在学院，学院地点，学院电话)，关键字为单一关键字"学号"，因为存在如下决定关系：
(学号) → (姓名，年龄，所在学院，学院地点，学院电话)
这个数据库是符合2NF的，但是不符合3NF，因为存在如下决定关系：
(学号) → (所在学院) → (学院地点，学院电话)
即存在非关键字段"学院地点"、"学院电话"对关键字段"学号"的传递函数依赖
它也会存在数据冗余、更新异常、插入异常和删除异常的情况

把学生关系表分为如下两个表：
学生：(学号，姓名，年龄，所在学院)
学院：(学院，地点，电话)
这样的数据库表是符合第三范式的，消除了数据冗余、更新异常、插入异常和删除异常

4-1-4. 反范式化设计

反范式化允许存在少量的数据冗余，相当于用空间换时间

4-2. 物理设计

4-2-1. 数据类型的选择

- 当一个列可以选择多种数据类型·时，应该优先考虑数字类型，其次是日期或二进制类型，最后是字符类型。对于相同级别的数据类型，应该优先选择占用空间小的数据类型

4-2-1-1. 如何正确选择整数类型

列类型	存储空间	取值范围	
		SIGNED	UNSIGNED
tinyint	1字节	-128~127	0~255
smallint	2字节	-32768~32767	0~65535
mediumint	3字节	-8388608~8388607	0~16777215
int	4字节	-2147483648~2147483647	0~4294967295
bigint	8字节	-9223372036854775808	0
		~9223372036854775807	~18446744073709551615

enter description here

4-2-1-2. 如何正确选择实数类型

- DECIMAL用法--DECIMAL(10,2)，共10位长度，其中保留两位小数位

列类型	存储空间	是否精确类型
FLOAT	4个字节	否
DOUBLE	8个字节	否
DECIMAL	每4个字节存9个数字，小数点占一个字节	是

4-2-1-3. 如何选择VARCHAR和CHAR类型

VARCHAR和CHAR中宽度是以字符为单位

VARCHAR类型的存储特点

- VARCHAR用于存储变长字符串，只占用必要的存储空间
- 列的最大长度小于255则只占用一个额外字节用于记录字符串长度
- 列的最大长度大于255则要占用两个额外字节用于记录字符串长度

CHAR类型的存储特点

- CHAR类型是定长的
- 字符串存储在CHAR类型的列中会删除末尾的空格
- CHAR类型的最大长度为255

4-2-1-4. 如何正确选择日期类型

4-2-1-4-1. DATATIME

DATATIME = YYYY-MM-DD HH:MM:SS
DATATIME类型与时区无关，占用8个字节的存储空间
时间范围1000-01-01 00:00:00到9999-12-31 23:59:59

4-2-1-4-2. TIMESTAMP(时间戳)

存储了由格林尼治时间1970年1月1日到当前时间的秒数
TIMESTAMP = YYYY-MM-DD HH:MM:SS
TIMESTAMP占用4个字节

时间范围1970-01-01到2038奶奶-01-19

TIMESTAMP类型显示依赖于所指定的时区
在行的数据修改时可以自动更新timestamp列的值

4-2-1-4-3. date类型和time类型

date类型用于存储日期
DATA = YYYY - MM -DD
存储用户生日时，只需存储日期类型

time类型用于存储时间数据，格式为HH:MM:SS

4-2-1-4-4. 存储日期时间数据的注意事项

- 不要使用字符串类型来存储日期时间数据
  - 日期时间通常比字符串占用的存储空间小
  - 日期时间类型在查找过滤时可以利用日期来进行对比
  - 日期时间类型有着丰富的处理函数，可以方便地对时期类型进行日期计算

5. MySQL高可用架构

5-1. MySQL复制

5-1-1. MySQL二进制日志

二进制日志记录了所有对MySQL数据库的修改事件，包括了CURD事件和对表结构的修改事件


```
#  查看二进制日志是否启动
show  variables  like  'log_bin'
```

若未开启需要修改my.cnf配置文件

```
log_bin = /var/log/mysql/mariadb-bin
```

如何开启二进制日志

二进制日志的格式

- 基于段的格式 binlog_format = STATEMENT
  优点：
  - 日志记录量相对较小，节约磁盘及网络IO
    缺点：
  - 可能造成MySQL复制的主备服务器数据不一致
- 基于行的格式 binlog_format = ROW
  同一SQL语句修改了10000条数据的情况下
  基于段的日志格式只会记录这个SQL语句
  基于行的日志会由10000条记录分别记录每一行的数据修改
  优点：
  - 使MySQL主从复制更加安全
  - 对每一行数据的修改比基于段的复制高效
    缺点：
  - 记录日志量较大--binlog_row_image=[FULL][MINIMAL][NOBlOB]//三个可选值
- 混合日志格式 binlog_format = MIXED
  特点
  - 根据SQL语句由系统在基于段和基于行的日志格式选择
  - 数据量的大小由所执行的SQL语句决定

5-1-2. MySQL二进制格式对复制的影响

- 基于SQL语名的复制(SBR),二进制日志格式使用的是statement格式(只记录SQL语句，不记录数据)
- 基于行的复制(RBR),二进制日志格式使用的是基于行的日志格式

5-1-2-1. 基于SQL语名的复制(SBR)

优点

- 生成的日志量少，节约网络传输IO
- 并不强制要求主从数据库的表定义完全正确

缺点：

- 对于非确定性事件，无法保证主从复制数据的一致性，比如UUID()
- 对于存储过程，触发器和自定义函数进行的修改也可能造成数据不一致

5-1-2-2. 基于行的复制(RBR)

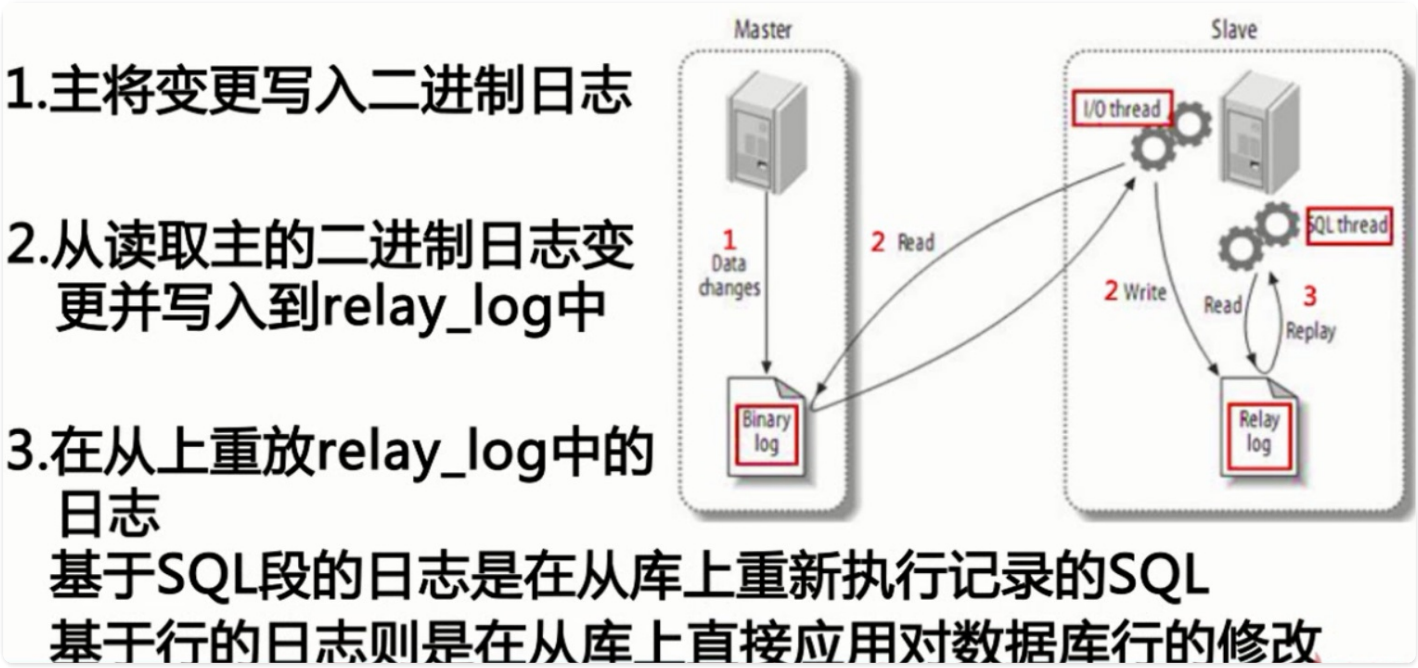
优点

- 可以应用任何SQL的复制包括非确定函数，存储过程等
- 可以减少数据库锁的使用

缺点：

- 要求主从数据库的表结构相同，否则可能会中断复制

5-1-3. MySQL是如何进行复制的?



enter description here

5-1-3-1. 基于日志点的复制

优点

- 是MySQL最早支持的复制技术，Bug相对较少
- 对SQL查询没有任何限制
- 故障处理比较容易

缺点:

- 故障转移时重新获取新主的日志点信息比较困难

基于日志点的复制配置步骤

1. 在主DB服务器上建立复制账号`CREATE USER 'repl' @ 'IP段' identified by 'PassW0rd'`
2. 授权--`GRANT REPLICATION SLAVE ON *.* TO 'repl' @ 'IP段'`

配置主服务器

1. `bin_log = mysql-bin`启动二进制日志
2. 配置server_id = 100(或者其他数字，只要在集群中唯一)

配置从服务器

1. `bin_log = mysql-bin`
2. `server_id = 101`(其他数字也可以)
3. `relay_log = mysql-relay-bin`
4. log_slave_updatec = on [可选]
5. read_only = on [可选]

初始化从服务器数据(热备份)

备份服务器数据--`mysqldump --master-data=2 -single-transaction`

启动复制链路

```
CHANGE MASTER TO MASTER_HOST = 'master_host_ip',
MASTER_USER = 'repl',
MASTER_PASSWORD = 'PassW0rd',
MASTER_LOG_FILE = 'mysql_log_file_name',
MASTER_LOG_POS=4;
```

整体流程:

```
# 主服务器配置
create user repl@'192.168.1.%' identified by '123456';# 配置账户
grant replication slave on *.* to repl@'192.168.1.%'; # 授权
log-bin=/var/lib/mysql/mysql-bin
server-id=1

# 从服务器配置
log-bin=/var/lib/mysql/mysql-bin
server-id=2
relay_log=/var/lib/mysql/mysqld-relay-bin

# 备份数据库
mysqldump --single-transaction --master-data --triggers --all-databases >> all.sql
# 拷贝备份数据到从机
scp all.sql root@192.168.1.123:/root
# 从数据库数据初始化
mysql -uroot -p <all.sql

# 进行主从连接
change master to master host='192.168.1.111',
master_user='repl',
master_password='123456',
MASTER_LOG_FILE='mysql-bin.000002', MASTER_LOG_POS=154; # 这个可以从之前备份的all.sql里面找

# 启动
show slave status \G
start slave;
```

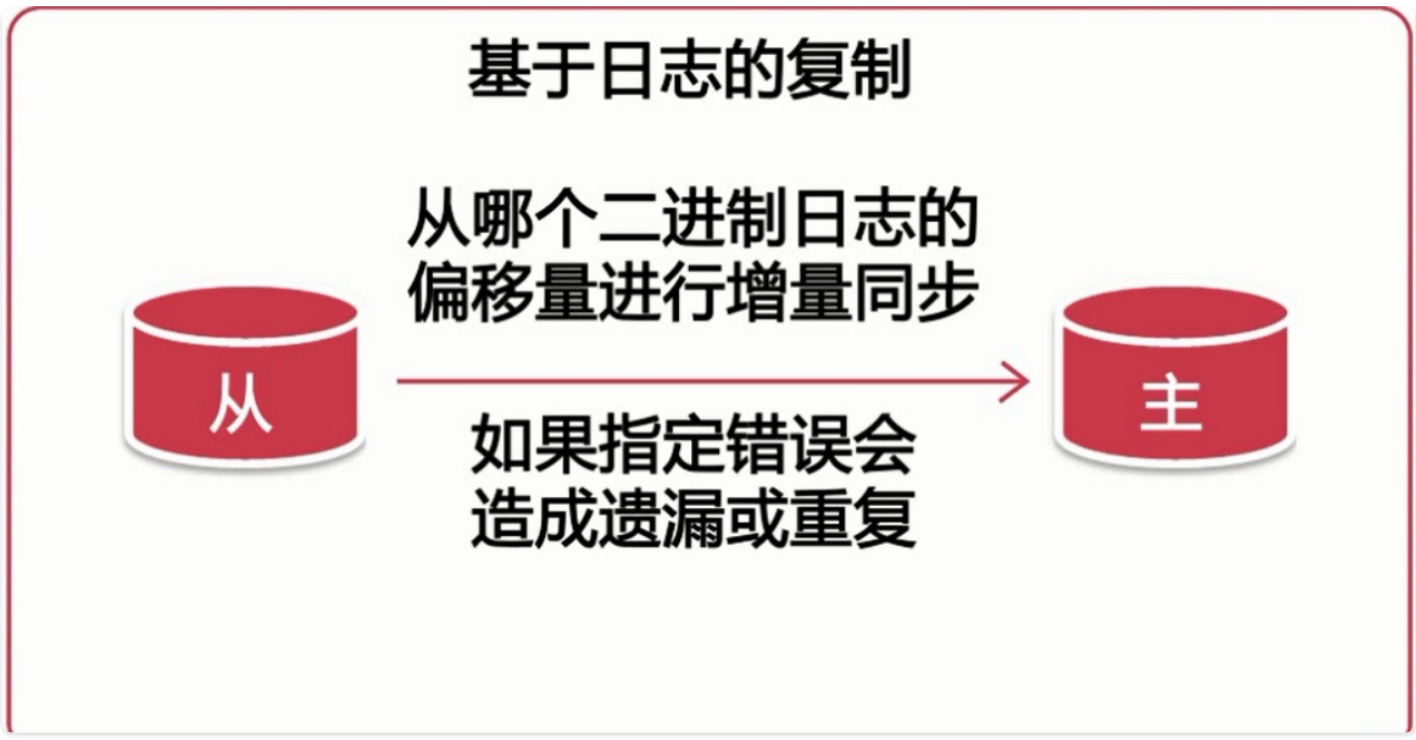
检查两个状态

- Slave_IO_RUNNING
- Slave_SQL_RUNNING

如果Slave_SQL_RUNNING = NO
执行

```
stop slave;
set GLOBAL SQL_SLAVE_SKIP_COUNTER=1;
start slave ;
```

5-1-3-2. 基于GTID的复制



enter description here

5-1-3-2-1. 什么是GTID?

GTID即全局事务ID，其保证为每一个在主上提交的事务在复制集群中可以生成一个惟一的ID

`GTID=source_id:transaction_id`

5-1-4. MySQL复制性能优化

影响主从延迟的因素

- 主库写入二进制日志的时间(控制主库中事务的大小，分割大事务)

- 二进制日志传输时间，传输的日志量(使用MIXED日志格式或设置`set binlog_row_image = minimal;`)
- 默认情况下从只有一个SQL线程，主上并发的修改在从上变成了串行(使用多线程复制)

如何配置多线程复制

```
stop slave;
set global slave_parallel_type = 'logical_clock';
set global slave_parallel_workers = 4;# 配置线程数量
start slave;
```

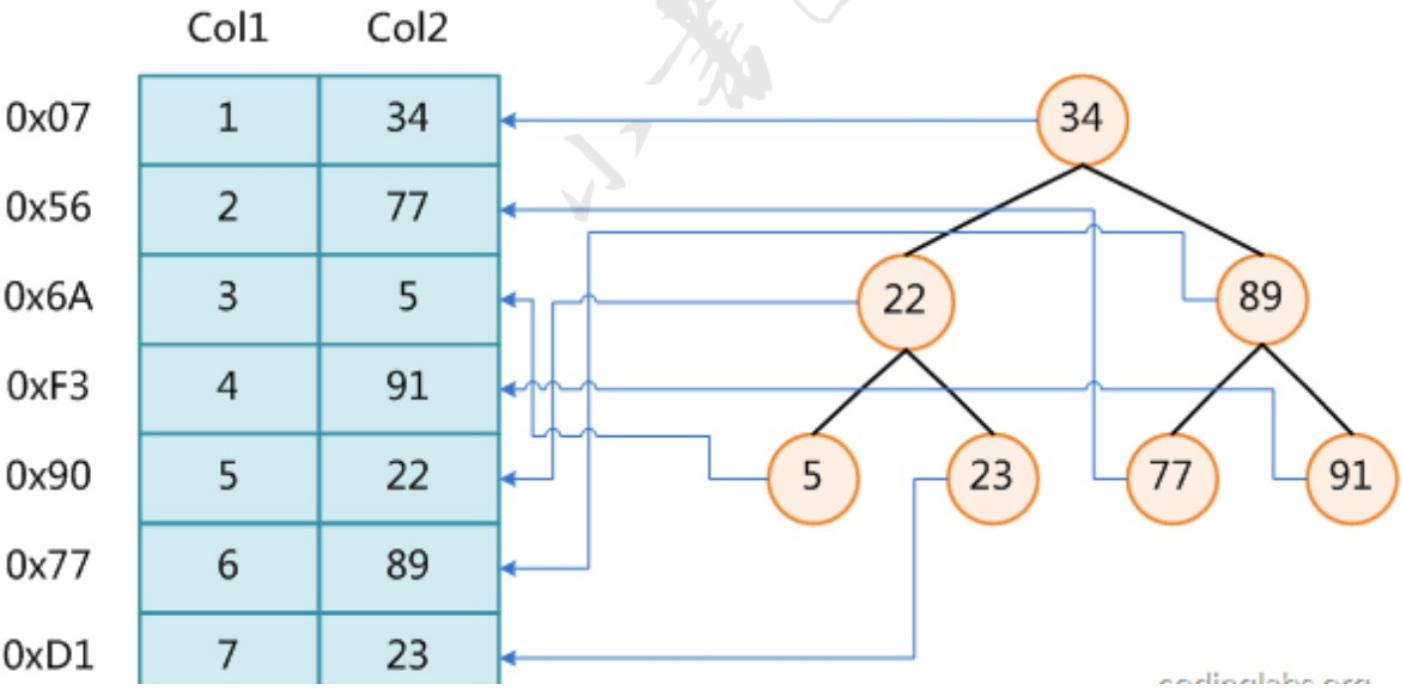
5-1-5. MySQL复制常见问题

- 由于数据损坏或丢失所引起的主从复制错误
  - 主库或从库意外宕机引起的错误
  - 主库上的二进制日志损坏(通过change master重新指定)
  - 备库上的重击日志损坏
- 在从库上进行修改造成的主从复制错误(对从库设置read_only)

6. 索引

索引好文章

索引是帮助MySQL高效获取数据的数据结构。



左边是数据表，最左边是数据记录的物理地址。

为了加快Col2的查找，可以维护右边所示的二叉查找树，每个节点分别包含索引键值和一个指向对应记录物理地址的指针

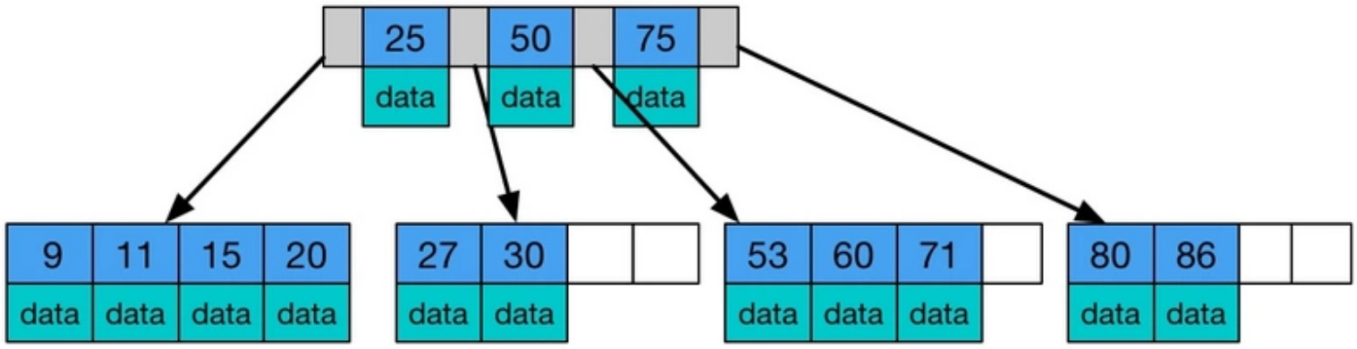
我对索引的理解：被设置为索引的字段，比如主键ID，会被设置为索引值并形成树，树中节点包含索引值和data(data的具体实现内容取决于不同的存储引擎)

6-1. Btree索引

6-1-1. B-/B+树

6-1-1-1. B-树

B-树，这里的B代表balance，B-树是一种多路平衡搜索树，类似普通的平衡二叉树，不同的是B-树允许每个节点有更多的子节点。它的每一个节点最多包含M个孩子，M就是B树的阶，M的大小取决磁盘页的大小。
ps(B-树就是B树，不要念成B减树)



特点

- 所有的键值分布在整棵树中
- 任何关键字(data)出现且只出现在一个节点中
- 搜索有可能在非叶子节点就结束了
- 性能逼近二分查找

6-1-1-2. B+树

B+树是B-树的变体，也是一种多路搜索树，与B-树的不同之处在于：

- 所有关键字(data)存储在叶子节点
- 为所有叶子节点增加了一个链指针，形成了有序的链表
- 每个父节点的元素都同时存在于子节点中，是子节点中最大或最小的元素
- 由于父节点的信息都包含在子节点中，因此所有的叶子节点包含了全部的元素信息

6-1-1-3. 为什么使用B-/B+树?

红黑树等数据结构也可以用来实现索引，但是文件系统及数据库系统普遍采用B-/+Tree作为索引结构
MySQL 是基于磁盘的数据库系统,索引往往以索引文件的形式存储的磁盘上,索引查找过程中就要产生磁盘I/O消耗,相对于内存存取，I/O存取的消耗要高几个数量级,索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数。为什么使用B-/+Tree，还跟磁盘存取原理有关。

局部性原理与磁盘预读

由于磁盘的存取速度与内存之间鸿沟,为了提高效率,要尽量减少磁盘I/O.磁盘往往不是严格按需读取，而是每次都会预读,磁盘读取完需要的数据,会顺序向后读一定长度的数据放入内存。而这样做的理论依据是计算机科学中著名的局部性原理：

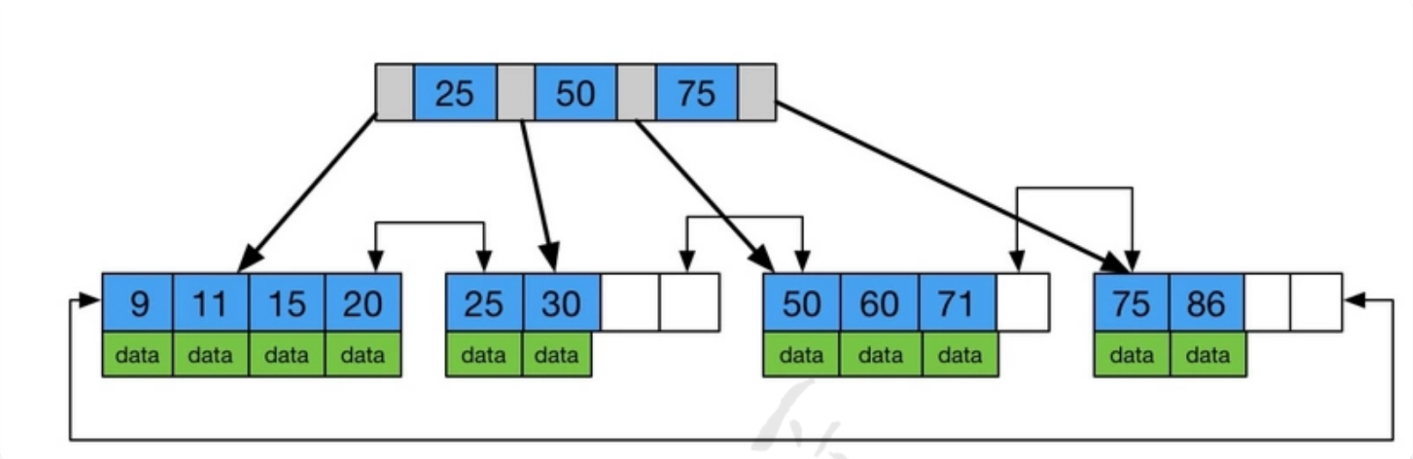
> 当一个数据被用到时，其附近的数据也通常会马上被使用
> 程序运行期间所需要的数据通常比较集中

由于磁盘顺序读取的效率很高(不需要寻道时间，只需很少的旋转时间)，因此对于具有局部性的程序来说，预读可以提高I/O效率.预读的长度一般为页(page)的整倍数
`MySQL(默认使用InnoDB引擎),将记录按照页的方式进行管理,每页大小默认为16K(这个值可以修改).linux 默认页大小为4K`

B-Tree树每次新建节点时，直接新建一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配是按页对齐的，就实现了一个结点只需一次IO，逻辑上相近的结点，物理地址也接近；
而红黑树这种结构，逻辑上相近的结点物理上可能很远，无法利用局部性。

6-1-1-4. 为什么使用B+树?

1. B+树更适合外部存储，由于内节点无data，一个节点可以存储更多的内节点，每个节点能索引的范围更大，同样大小的磁盘页可以容纳更多的节点元素，因此查询时IO会更少
2. B-树想要范围查询只能依靠繁琐的中序遍历，而B+树只需要在链表上遍历即可，B+树范围查询方便
3. B+树的查询必须是最终找到叶子节点，而B-树只需要找到匹配的元素，无论匹配元素是中间节点还是叶子节点，因此B-树的查找性能不稳定



enter description here

6-2. 哈希索引

- Hash索引是基于Hash表实现的，只有查询条件精确匹配Hash索引中的所有列时，才能够使用到Hash索引
- 对于Hash索引中的所有行，存储引擎都会为每一行计算一个Hash码，Hash索引中存储的就是Hash码

6-2-1. Hash索引的限制

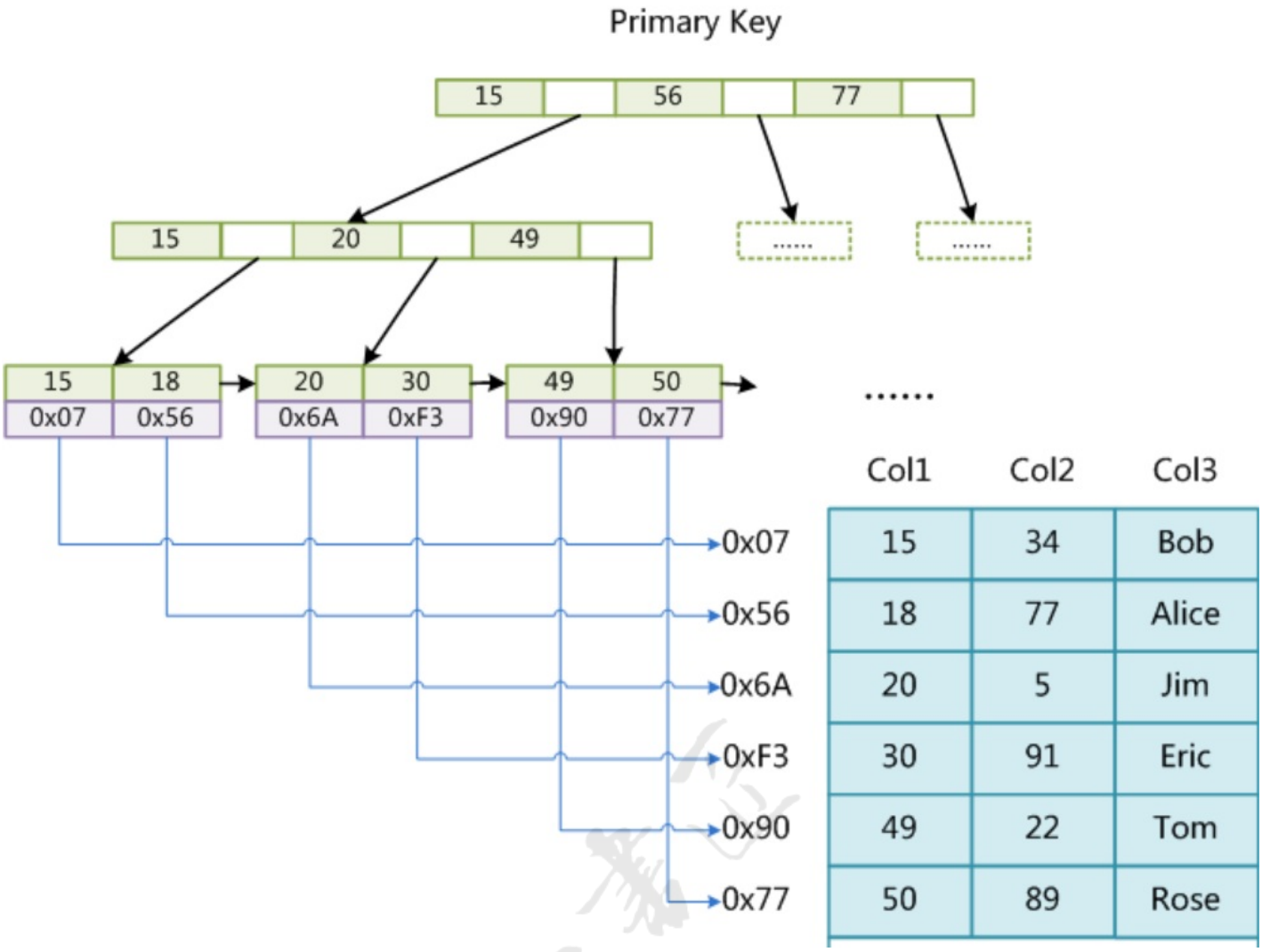
- Hash查找数据需要进行两次读取，因为Hash索引中保存的是哈希码，必须先通过哈希码找到对应的行，再对行的记录进行读取
- Hash索引无法用于排序，由于Hash索引中存放的是经过Hash计算之后的Hash值，而且Hash值的大小关系不一定和Hash运算前的键值一样，所以数据库无法利用哈希索引的数据进行任何排序
- Hash索引不支持部分索引查找，只能进行全值匹配，对于组合索引，Hash索引在计算哈希码的时候是组合索引键合并后再在一起计算Hash值，而不是单独计算Hash值，所以没办法进行部分索引
- Hash索引不支持范围查找，如果键值不是唯一的，就需要先找到该键所在的位置，然后再根据链表往后扫描，直到找到相应的数据
- 哈希索引存在哈希冲突(不适合用在键值对重复值很多的字段，比如性别)

6-3. 不同引擎下索引的实现

在MySQL中，索引属于存储引擎级别的概念，不同存储引擎对索引的实现方式是不同的。

6-3-1. MyISAM索引实现

MyISAM引擎使用B+Tree作为索引结构，叶节点的data域存放的是数据记录的地址



MyISAM的索引文件仅仅记录数据记录的地址，在MyISAM中，主索引和辅助索引在结构上没有任何差别，只是主索引要求key是唯一的

因此，MyISAM中索引检索的算法为首先按照B+Tree搜索算法搜索索引，如果指定的Key存在，则取出其data域的值，然后以data域的值为地址，读取相应数据记录

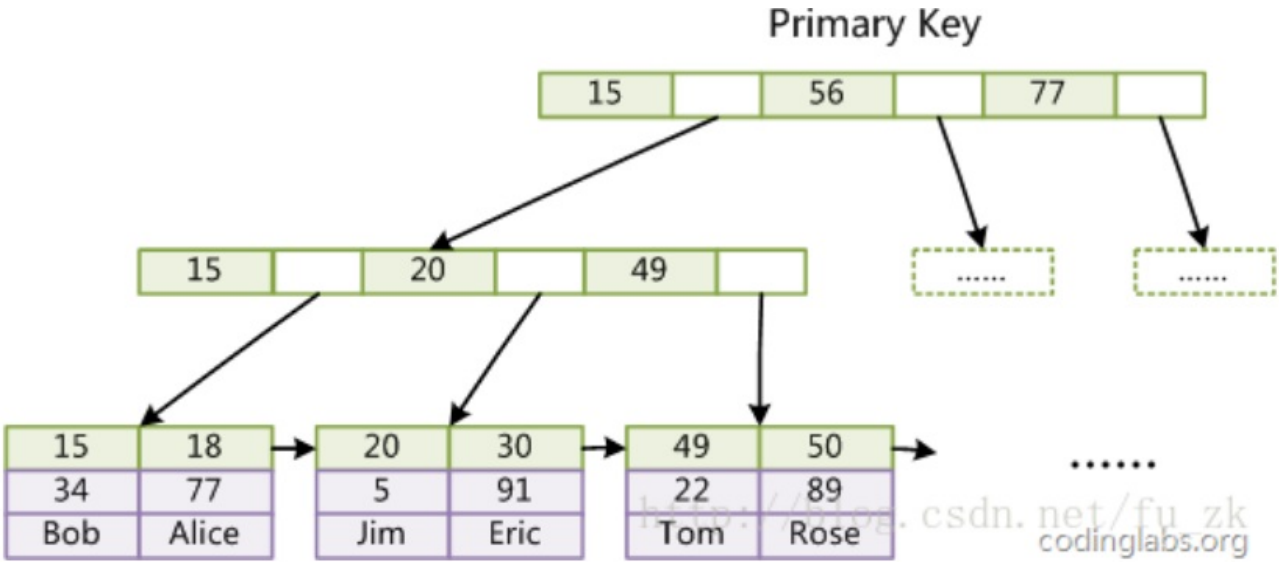
MyISAM的索引方式也叫做“非聚集”的，之所以这么称呼是为了与InnoDB的聚集索引区分。

ps：(主索引和辅助索引都是B+树，叶子节点都存储的是数据记录的地址，索引文件和数据文件是分离的，主索引和辅助索引都不会影响数据文件)

6-3-2. Innodb索引实现

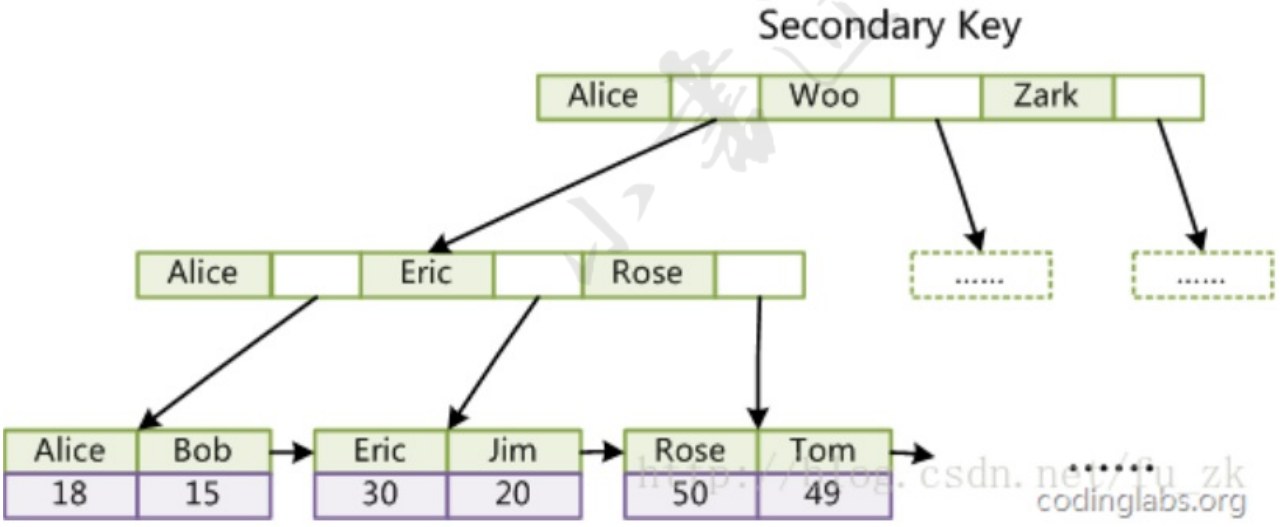
虽然InnoDB也使用B+Tree作为索引结构，但具体实现方式却与MyISAM截然不同。

第一个重大区别是InnoDB的数据文件本身就是索引文件。MyISAM索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。而在InnoDB中，表数据文件本身就是按B+Tree组织的一个索引结构，这棵树的叶结点data域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。



上图是InnoDB主索引（同时也是数据文件）的示意图，可以看到**叶结点包含了完整的数据记录(从上往下代表(字段1、字段2、字段3：15，34，Bob))，这种索引叫做聚集索引**。因为InnoDB的数据文件本身要按主键聚集，所以InnoDB要求表必须有主键（MyISAM可以没有），如果没有显式指定，则MySQL系统会自动选择一个可以唯一标识数据记录的列作为主键，如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。

第二个与MyISAM索引的不同是 InnoDB的辅助索引data域存储相应记录主键的值而不是地址
换句话说，InnoDB的所有辅助索引都引用主键作为data域。例如，下图为定义在Col3上的一个辅助索引：



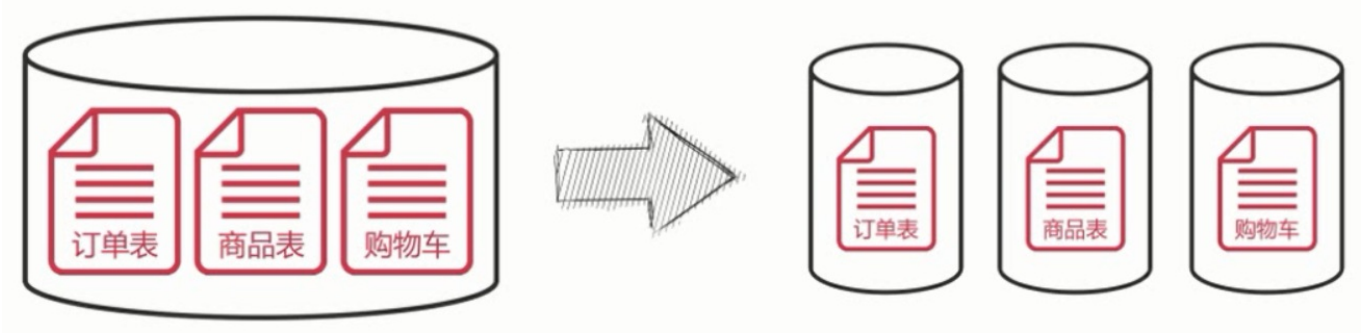
聚集索引这种实现方式使得按主键的搜索十分高效，但是辅助索引搜索需要检索两遍索引：**首先检索辅助索引获得主键，然后用主键到主索引中检索获得记录**

了解不同存储引擎的索引实现方式对于正确使用和优化索引都非常有帮助，例如知道了InnoDB的索引实现后，就很容易明白为什么不建议使用过长的字段作为主键，因为所有辅助索引都引用主索引，过长的主索引会令辅助索引变得过大。再例如，用非单调的字段(如UUID或者字符类型)作为主键在InnoDB中不是个好主意，因为InnoDB数据文件本身是一颗B+Tree，非单调的主键会造成在插入新记录时数据文件为了维持B+Tree的特性而频繁的分裂调整，十分低效，而使用自增字段作为主键则是一个很好的选择

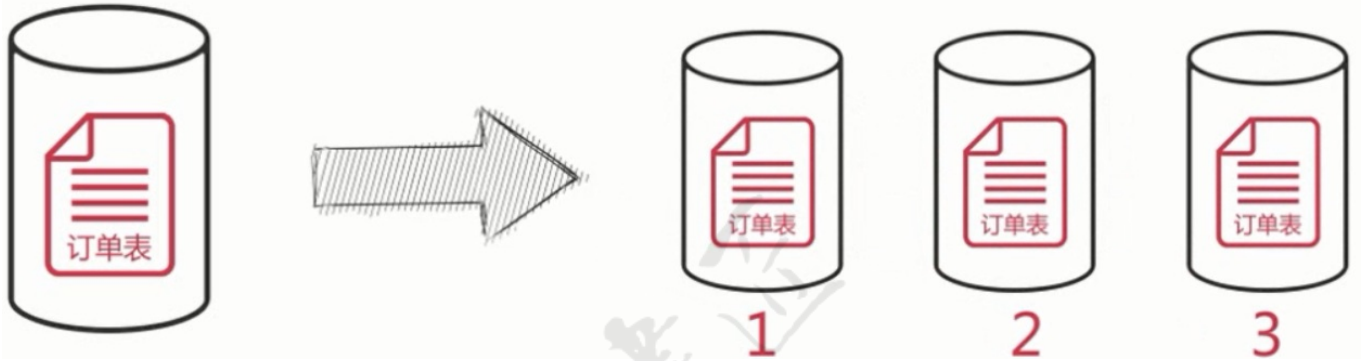
7. 数据库的分库分表

- 把一个实例的多个数据库拆分到不同的实例

- 把一个库中的表分离到不同的数据库中



- 库中的相关表进行水平分片



7-1. 数据库分片

7-1-1. 如何选择分区键

- 分区键要尽量跨分片查询的发生
- 分区键要能尽量使各个分片中的数据平均

7-1-2. 如何存储无需分片的表

- 每个分片中存储一份相同的数据，进行数据冗余

7-1-3. 如何在节点上部署分片

- 每个分片使用单一数据库，并且数据库名也相同
- 将多个分片表存储在一个数据库中，并在表名上加上分片号后缀
- 在一个节点中部署多个数据库，每个数据库包含一个分片

7-1-4. 如何分配分片中的数据

- 按分区键的Hash值取模来分配分片数据
- 按分区键的范围来分配分片数据
- 利用分区键和分片的映射表来分配分片数据

7-1-5. 如何生成全局唯一ID

- 使用`auto_increment_increment`和`auto_increment_offest`参数2(offest设置为分片节点的数量，比如6个分片则设置offest为6，而auto_increment_increment设置为0-6的值，这样唯一ID就不会冲突)
- 使用全局节点来生成ID
- 在Redis等缓存服务器中创建全局ID

8. 数据库监控

- 对数据库服务可用性进行监控
  - 数据库进程或是端口存在并不一定数据库就可以对外提供服务
- 对数据库性能进行监控，QPT、TPS以及并发线程数
- 对主从复制进行监控
  - 主从复制链路状态的监控
  - 主从复制延迟的监控
  - 定期确认主从复制的数据是否一致
- 对服务器资源的监控

8-1. 数据库可用性监控

8-1-1. 如何确认数据库是否可以通过网络连接

```
#  ping被连接的服务器数据库用户
mysqladmin  -umonitor_user  -p  -h  ping
#  telnet
telnet  ip  db_port
```

8-1-2. 如何确认数据库是否读写?

- 定期检查数据库的read_only参数是否为off
- 执行简单的查询 select @@version

8-1-3. 监控数据库的连接数量

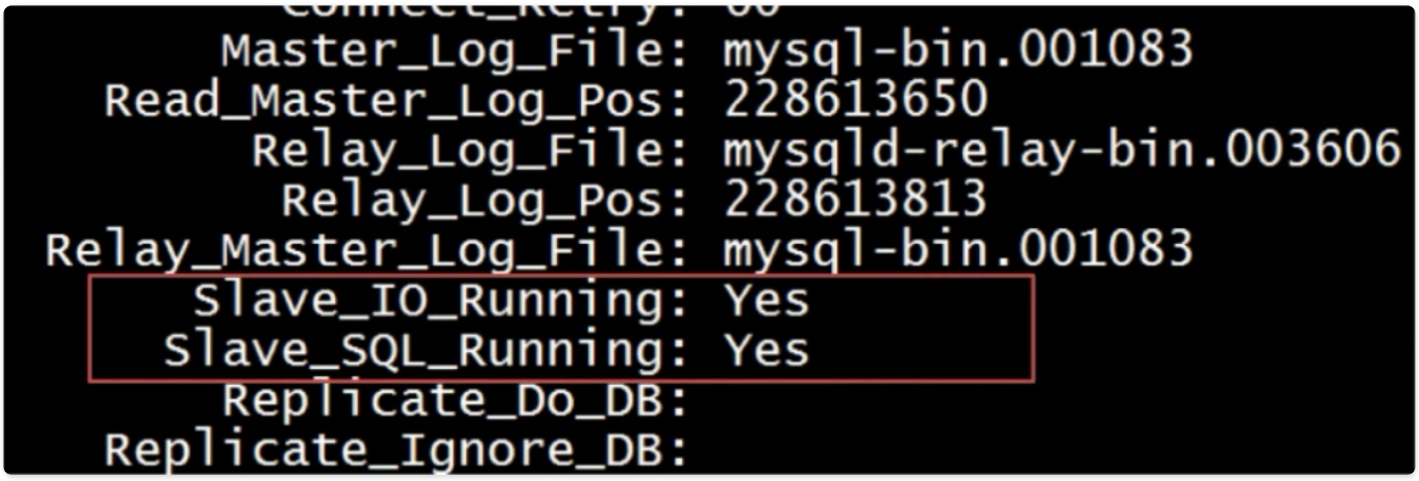
```
show  variables  like  'max_connections';
shwo  global  status  like  'Threads_connected'
```

8-2. 数据库性能监控

8-3. 主从复制监控

8-3-1. 如何监控主从复制链路的状态

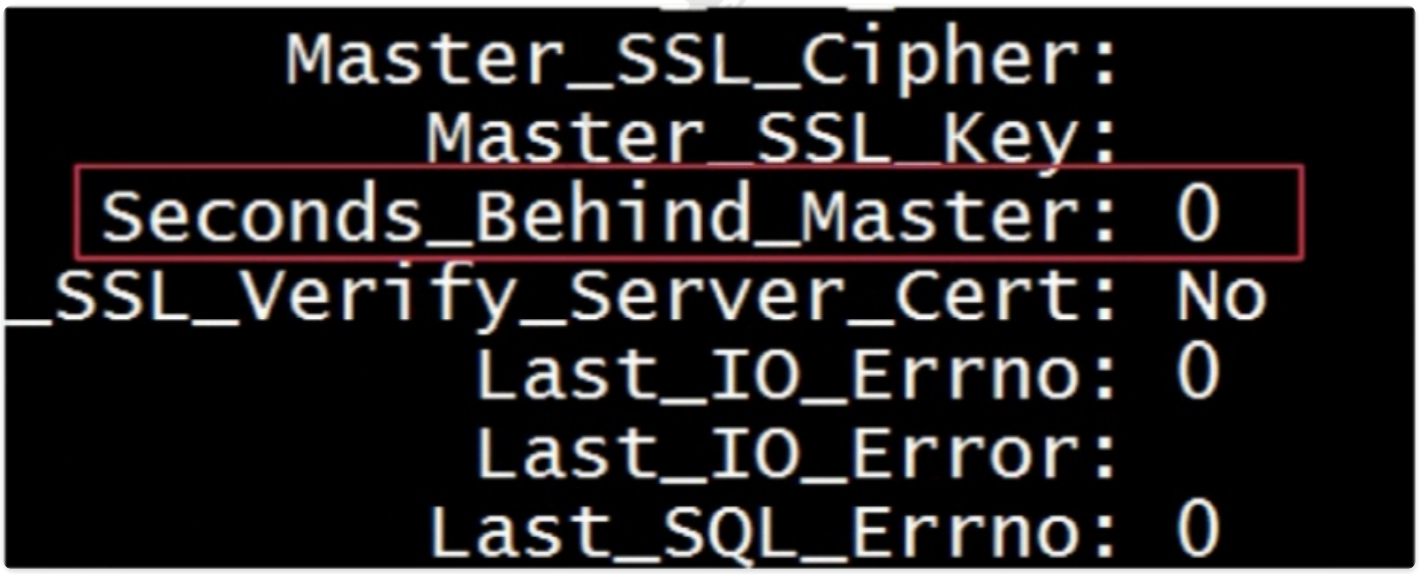
```
#  监测Slave上负责主从复制的两个线程是否开启
show  processlist;
show  slave  status  \G
```



enter description here

8-3-2. 如何监控主从复制延迟

```
show  slave  status  \G
```



enter description here

8-3-3. 如何检查主从复制的数据是否一致

```
#  主库上运行即可，会自动检测从库信息
pt-table-checksum  u  =  '账户',p='密码'
--databases  mysql  \
--replicate  test.checksums
```