码农求职小助手：MySQL高频面试题

笔记本： 10-MySQL
创建时间： 2019/9/7 21:48 **更新时间：** 2019/9/7 21:49
作者： pc941206@163.com

一 索引

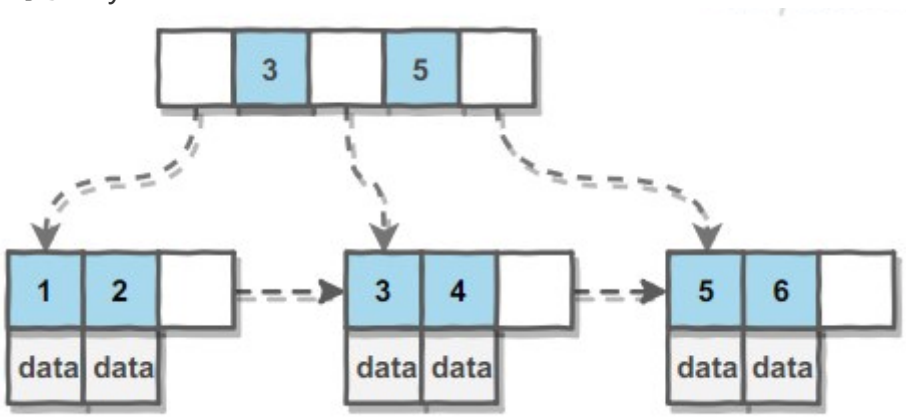
1、B+ 树原理

1.1 数据结构

B Tree 指的是 Balance Tree，也就是平衡树。**平衡树是一颗查找树，并且所有叶子节点位于同一层。**

B+ 树是基于 B 树和叶子节点顺序访问指针进行实现，它具有 B 树的平衡性，并且通过顺序访问指针来提高区间查询的性能。

在 B+ 树中，一个节点中的 key 从左到右非递减排列，如果某个指针的左右相邻 key 分别是 key i 和 key i+1，且不为 null，则该指针指向节点的所有 key 大于等于 key i 且小于等于 key i+1。



1.2 操作

进行查找操作时，首先在根节点进行二分查找，找到一个 key 所在的指针，然后递归地在指针所指向的节点进行查找。直到查找到叶子节点，然后在叶子节点上进行二分查找，找出 key 所对应的 data。

插入、删除操作会破坏平衡树的平衡性，因此在插入删除操作之后，需要对树进行一个分裂、合并、旋转等操作来维护平衡性。

1.3 与红黑树的比较

红黑树等平衡树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+ 树作为索引结构，主要有以下两个原因：

（一）更少的查找次数

平衡树查找操作的时间复杂度和树高 h 相关，O(h)=O(logdN)，其中 d 为每个节点的出度。

红黑树的出度为 2，而 B+ Tree 的出度一般都非常大，所以红黑树的树高 h 很明显比 B+ Tree 大非常多，查找的次数也就更多。

（二）利用磁盘预读特性

为了减少磁盘 I/O 操作，磁盘往往不是严格按需读取，而是每次都会预读。预读过程中，磁盘进行顺序读取，顺序读取不需要进行磁盘寻道，并且只需要很短的旋转时间，速度会非常快。

操作系统一般将内存和磁盘分割成固定大小的块，每一块称为一页，内存与磁盘以页为单位交换数据。数据库系统将索引的一个节点的大小设置为页的大小，使得一次 I/O 就能完全载入一个节点。并且可以利用预读特性，相邻的节点也能够被预先载入。

2、MySQL 索引

索引是在存储引擎层实现的，而不是在服务器层实现的，所以不同存储引擎具有不同的索引类型和实现。

2.1 B+ 树索引

是大多数 MySQL 存储引擎的默认索引类型。

因为不再需要进行全表扫描，只需要对树进行搜索即可，所以查找速度快很多。

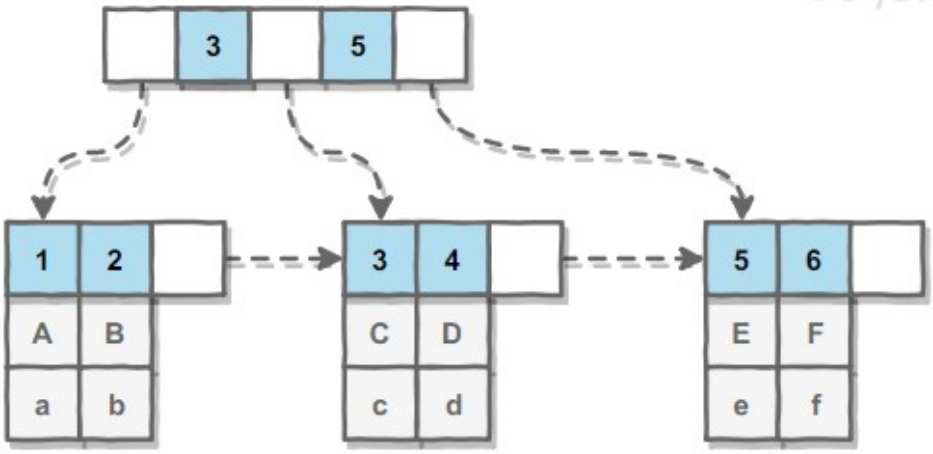
除了用于查找，还可以用于排序和分组。

可以指定多个列作为索引列，多个索引列共同组成键。

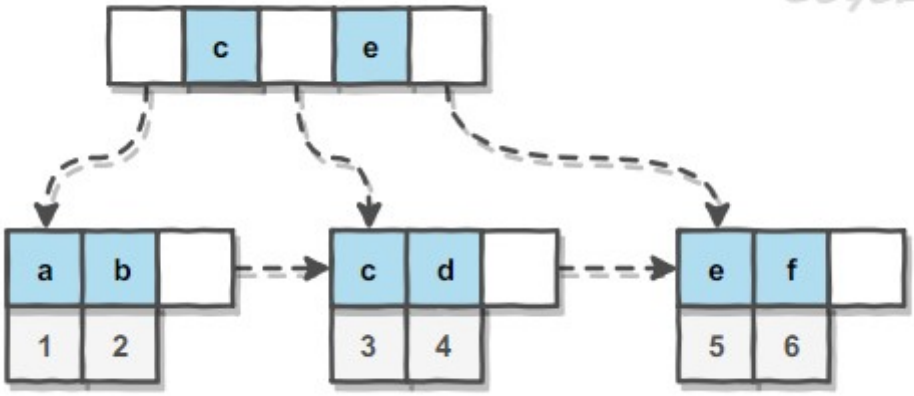
适用于全键值、键值范围和键前缀查找，其中键前缀查找只适用于最左前缀查找。如果不是按照索引列的顺序进行查找，则无法使用索引。

- **聚簇索引**

InnoDB 的 B+ 树索引分为主索引和辅助索引。主索引的叶子节点 data 域记录着完整的数据记录，这种索引方式被称为**聚簇索引。**因为无法把数据行存放在两个不同的地方，所以一个表只能有一个聚簇索引。



辅助索引的叶子节点的 data 域记录着主键的值，因此在使用辅助索引进行查找时，需要先查找到主键值，然后再到主索引中进行查找。



> - 聚簇索引和非聚簇索引：https://blog.csdn.net/alexdamiao/article/details/51934917
>
> **聚簇索引**：是对磁盘上实际数据重新组织以按指定的一个或多个列的值排序的算法。特点是**存储数据的顺序和索引顺序一致。一般情况下主键会默认创建聚簇索引，且一张表只允许存在一个聚簇索引。**
>
> **聚簇索引和非聚簇索引的区别**：**聚簇索引的叶子节点就是数据节点，而非聚簇索引的叶子节点仍然是索引节点，只不过有指向对应数据块的指针。**

因为索引是在存储引擎层实现的，所以不同存储引擎底层索引实现的原理也是不同的：

> MyISAM： **B+ 树叶节点的 data 域存放的是数据记录的地址【物理地址】。**在索引检索的时候，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的数据记录。这被称为“**非聚簇索引**”。

> InnoDB：**其数据文件本身就是索引文件。**相比 **MyISAM的索引文件和数据文件是分离的**，其表数据文件本身就是按 B+Tree 组织的一个索引结构，树的叶节点 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。**这被称为“聚簇索引（或聚集索引）”。而其余的索引都作为辅助索引，**辅助索引的 data 域存储相应记录主键的值而不是地址**，这也是和 MyISAM 不同的地方。在 **根据主索引搜索时，直接找到 key 所在的节点即可取出数据；在根据辅助索引查找时，则需要先取出主键的值，再走一遍主索引。 因此，在设计表的时候，不建议使用过长的字段作为主键，也不建议使用非单调的字段作为主键，这样会造成主索引频繁分裂。**

2.2 哈希索引

哈希索引能以 O(1) 时间进行查找，但是失去了有序性：

> 1.无法用于排序与分组；
>
> 2.只支持精确查找，无法用于部分查找和范围查找。

InnoDB 存储引擎有一个特殊的功能叫“**自适应哈希索引**”，**当某个索引值被使用的非常频繁时，会在 B+ 树索引之上再创建一个哈希索引**，这样就让 B+Tree 索引具有哈希索引的一些优点，比如快速的哈希查找。

2.3 全文索引

MyISAM 存储引擎支持全文索引，用于查找文本中的关键词，而不是直接比较是否相等。

查找条件使用 MATCH AGAINST，而不是普通的 WHERE。

全文索引使用倒排索引实现，它记录着关键词到其所在文档的映射。

InnoDB 存储引擎在 MySQL 5.6.4 版本中也开始支持全文索引。

2.4. 空间数据索引

MyISAM 存储引擎支持空间数据索引（R-Tree），可以用于地理数据存储。空间数据索引会从所有维度来索引数据，可以有效地使用任意维度来进行组合查询。

必须使用 GIS 相关的函数来维护数据。

3、索引优化

3.1 独立的列

在进行查询时，索引列不能是表达式的一部分，也不能是函数的参数，否则无法使用索引。

例如下面的查询不能使用 actor_id 列的索引：

```
SELECT actor_id FROM sakila.actor WHERE actor_id + 1 = 5;
```

3.2 多列索引

在需要使用多个列作为条件进行查询时，使用多列索引比使用多个单列索引性能更好。

例如下面的语句中，最好把 actor_id 和 film_id 设置为多列索引。

```
SELECT film_id, actor_ id FROM sakila.film_actor
WHERE actor_id = 1 AND film_id = 1;
```

2.3 索引列的顺序

让选择性最强的索引列放在前面。

索引的选择性是指：不重复的索引值和记录总数的比值。最大值为 1，此时每个记录都有唯一的索引与其对应。选择性越高，查询效率也越高。

例如下面显示的结果中 customer_id 的选择性比 staff_id 更高，因此最好把 customer_id 列放在多列索引的前面。

```
SELECT COUNT(DISTINCT staff_id)/COUNT(*) AS staff_id_selectivity,
COUNT(DISTINCT customer_id)/COUNT(*) AS customer_id_selectivity,
COUNT(*)
FROM payment;
```

```
staff_id_selectivity: 0.0001
customer_id_selectivity: 0.0373
               COUNT(*): 16049
```

2.4 前缀索引

对于 BLOB、TEXT 和 VARCHAR 类型的列，必须使用前缀索引，只索引开始的部分字符。

对于前缀长度的选取需要根据索引选择性来确定。

2.5 覆盖索引

索引包含所有需要查询的字段的值。

> 当能通过检索索引就可以读取想要的数据，那就不需要再到数据表中读取行了。**如果一个索引包含了（或覆盖了）满足查询语句中字段与条件的数据就叫做覆盖索引。**

具有以下优点：

1.索引通常远小于数据行的大小，只读取索引能大大减少数据访问量。

2.一些存储引擎（例如 MyISAM）在内存中只缓存索引，而数据依赖于操作系统来缓存。因此，只访问索引可以不使用系统调用（通常比较费时）。

3.对于 InnoDB 引擎，若辅助索引能够覆盖查询，则无需访问主索引。

- 覆盖索引：https://www.cnblogs.com/happyflyingpig/p/7662881.html

4、索引的优点

1、大大减少了服务器需要扫描的数据行数。

2、帮助服务器避免进行排序和分组，以及**避免创建临时表**（B+ 树索引是有序的，可以用于 ORDER BY 和 GROUP BY 操作。临时表主要是在排序和分组过程中创建，因为不需要排序和分组，也就不需要创建临时表）。

3、将随机 I/O 变为顺序 I/O（B+ 树索引是有序的，会将相邻的数据都存储在一起）。

5、索引的使用条件

对于非常小的表、大部分情况下简单的全表扫描比建立索引更高效；

对于中到大型的表，索引就非常有效；

但是对于特大型的表，建立和维护索引的代价将会随之增长。这种情况下，需要用到一种技术可以直接区分出需要查询的一组数据，而不是一条记录一条记录地匹配，例如：可以使用**分区技术。**

6、什么情况下索引会失效？即查询不走索引？

下面列举几种不走索引的 SQL 语句：

1、索引列参与表达式计算：

```
SELECT `sname` FROM `stu` WHERE `age`+10=30;
```

2、函数运算：

```
SELECT `sname` FROM `stu` WHERE LEFT(`date`,4) <1990;
```

3、%词语%--模糊查询：

```
SELECT * FROM `houdunwang` WHERE `uname` LIKE '后盾%' -- 走索引

SELECT * FROM `houdunwang` WHERE `uname` LIKE "%后盾%" -- 不走索引
```

4、字符串与数字比较不走索引：

```
CREATE TABLE `a` (`a` char(10));
EXPLAIN SELECT * FROM `a` WHERE `a`="1" -- 走索引
EXPLAIN SELECT * FROM `a` WHERE `a`=1 -- 不走索引，同样也是使用了函数运算
```

5、查询条件中有 or ，即使其中有条件带索引也不会使用。换言之，就是要求使用的所有字段，都必须建立索引：

```
select * from dept where dname='xxx' or loc='xx' or deptno=45
```

6、正则表达式不使用索引。

7、MySQL 内部优化器会对 SQL 语句进行优化，如果优化器估计使用全表扫描要比使用索引快，则不使用索引。

二 查询性能优化

1、使用 Explain 进行分析

Explain 用来分析 SELECT 查询语句，可以模拟优化器执行 SQL 查询语句，从而知道 MySQL 是如何处理你的 SQL 语句的，分析你的查询语句或是表结构的性能瓶颈。

- Explain 执行计划包含的信息：

```
+----+-------------+-------+-------+---------------+---------+---------+------+------+-------------------------------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref  | rows | Extra                         |
+----+-------------+-------+-------+---------------+---------+---------+------+------+-------------------------------+
```


比较重要的字段有：

1、select_type : 查询类型，有简单查询（SIMPLE）、联合查询（UNION）、子查询（SUBQUERY）等；

2、type：访问类型【SQL 查询优化的一个重要指标】结果值从好到坏依次是：

system > const > eq_ref > ref > fulltext > ref_or_null > index_merge > unique_subquery > index_subquery > **range > index> ALL**

- **一般来说，好的 SQL 查询至少达到 RANGE 级别，最好能达到 REF。**

> 挑几个重要的解释下：
>
> **system**：表只有一行记录（等于系统表），这是 const 类型的特例，平时不会出现，可以忽略不计；
>
> **const**：表示通过索引一次就找到了，const 用于 primary key 或者 unique 索引。因为只需匹配一行数据，所有很快。如果将主键置于 where 列表中，mysql 就能将该查询转换为一个 const；
>
> **eq_ref**：唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配。常见于主键 或 唯一索引扫描；
>
> **ref**：非唯一性索引扫描，返回匹配某个单独值的所有行。本质是也是一种索引访问，它返回所有匹配某个单独值的行，然而它可能会找到多个符合条件的行，所以它应该属于查找和扫描的混合体；
>
> **range**：只检索给定范围的行，使用一个索引来选择行。key 列显示使用了那个索引。一般就是在 where 语句中出现了 bettween、<、>、in 等的查询。这种索引列上的范围扫描比全索引扫描要好。只需要开始于某个点，结束于另一个点，不用扫描全部索引。
>
> **index**：Full Index Scan，index 与 ALL 区别为 index 类型只遍历索引树。这通常为 ALL 快，应为索引文件通常比数据文件小。（index 与 ALL 虽然都是读全表，但 index 是从索引中读取，而 ALL 是从硬盘读取）
>
> **ALL**：Full Table Scan，遍历全表以找到匹配的行

2、possible_keys：查询涉及到的字段上存在索引，则该索引将被列出，但不一定被查询实际使用。

3、key : 使用的索引，如果为 NULL，则没有使用索引；

4、rows : 扫描的行数。

5、Extra：不适合在其他字段显示，但是十分重要的额外信息。

- Explain 命令详解：https://www.cnblogs.com/gomysql/p/3720123.html

2、优化数据访问

2.1 减少请求的数据量

1、只返回必要的列：最好不要使用 SELECT * 语句。

2、只返回必要的行：使用 LIMIT 语句来限制返回的数据。

3、缓存重复查询的数据：使用缓存可以避免在数据库中进行查询，特别在要查询的数据经常被重复查询时，缓存带来的查询性能提升将会是非常明显的。

2.2 减少服务器端扫描的行数

最有效的方式是使用索引来覆盖查询。

3、重构查询方式

3.1 切分大查询

一个大查询如果一次性执行的话，可能一次锁住很多数据、占满整个事务日志、耗尽系统资源、阻塞很多小的但重要的查询。

```
DELETE FROM messages WHERE create < DATE_SUB(NOW(), INTERVAL 3 MONTH);
```

```
rows_affected = 0
do {
    rows_affected = do_query(
    "DELETE FROM messages WHERE create  < DATE_SUB(NOW(), INTERVAL 3 MONTH)
LIMIT 10000")
} while rows_affected > 0
```

3.2 分解大连接查询

将一个大连接查询分解成对每一个表进行一次单表查询，然后在应用程序中进行关联，这样做的好处有：

1、让缓存更高效。对于连接查询，如果其中一个表发生变化，那么整个查询缓存就无法使用。而分解后的多个查询，即使其中一个表发生变化，对其它表的查询缓存依然可以使用；

2、分解成多个单表查询，这些单表查询的缓存结果更可能被其它查询使用到，从而减少冗余记录的查询；

3、减少锁竞争；

4、在应用层进行连接，可以更容易对数据库进行拆分，从而更容易做到高性能和可伸缩；

5、查询本身效率也可能会有所提升。例如：下面的例子中，使用 IN() 代替连接查询，可以让 MySQL 按照 ID 顺序进行查询，这可能比随机的连接要更高效。

```
SELECT * FROM tab
JOIN tag_post ON tag_post.tag_id=tag.id
JOIN post ON tag_post.post_id=post.id
WHERE tag.tag='mysql';
```

```
SELECT * FROM tag WHERE tag='mysql';
SELECT * FROM tag_post WHERE tag_id=1234;
SELECT * FROM post WHERE post.id IN (123,456,567,9098,8904);
```

- **面试问题**

- **MySQL 问题排查都有哪些手段?**

1、使用 show processlist 命令查看当前所有连接信息;

2、使用 explain 命令查询 SQL 语句执行计划;

3、开启慢查询日志，查看慢查询的 SQL。

- **如何做 MySQL 的性能优化?**

1、为搜索字段创建索引;

2、避免使用 select * ，列出需要查询的字段;

3、垂直分割分表;

4、选择正确的存储引擎。

三 存储引擎

1、InnoDB

是 MySQL 默认的事务型存储引擎，只有在需要它不支持的特性时，才考虑使用其它存储引擎。

实现了四个标准的隔离级别，默认级别是：可重复读（REPEATABLE READ）。在可重复读隔离级别下，通过多版本并发控制（MVCC）+ 间隙锁（Next-Key Locking）防止幻影读。

主索引是聚簇索引，在索引中保存了数据，从而避免直接读取磁盘，因此对查询性能有很大的提升。

内部做了很多优化，包括从磁盘读取数据时采用的可预测性读、能够加快读操作并且自动创建的自适应哈希索引、能够加速插入操作的插入缓冲区等。

支持真正的在线热备份。其它存储引擎不支持在线热备份，要获取一致性视图需要停止对所有表的写入，而在读写混合场景中，停止写入可能也意味着停止读取。

2、MyISAM

设计简单，数据以紧密格式存储。对于只读数据，或者表比较小、可以容忍修复操作，则依然可以使用它。

提供了大量的特性，包括压缩表、空间数据索引等。

不支持事务。

不支持行级锁，只能对整张表加锁，读取时会对需要读到的所有表加共享锁，写入时则对表加排它锁。但在表有读取操作的同时，也可以往表中插入新的记录，这被称为并发插入（CONCURRENT INSERT）。

可以手工或者自动执行检查和修复操作，但是和事务恢复以及崩溃恢复不同，可能导致一些数据丢失，而且修复操作是非常慢的。

如果指定了 DELAY_KEY_WRITE 选项，在每次修改执行完成时，不会立即将修改的索引数据写入磁盘，而是会写到内存中的键缓冲区，只有在清理键缓冲区或者关闭表的时候才会将对应的索引块写入磁盘。这种方式可以极大的提升写入性能，但是在数据库或者主机崩溃时会造成索引损坏，需要执行修复操作。

3、比较

1、事务：InnoDB 是事务型的，可以使用 Commit 和 Rollback 语句。

2、并发：MyISAM 只支持表级锁，而 InnoDB 还支持行级锁。

3、外键：InnoDB 支持外键。

4、备份：InnoDB 支持在线热备份。

5、崩溃恢复：MyISAM 崩溃后发生损坏的概率比 InnoDB 高很多，而且恢复的速度也更慢。

6、其它特性：MyISAM 支持压缩表和空间数据索引。

- **2 者 selectcount(*) 哪个更快，为什么?**

MyISAM 更快，因为 MyISAM 内部维护了一个计数器，可以直接调取。

四 数据类型

1、整型

TINYINT, SMALLINT, MEDIUMINT, INT, BIGINT 分别使用 8, 16, 24, 32, 64 位存储空间，一般情况下越小的列越好。

INT(11) 中的数字只是规定了交互工具显示字符的个数，对于存储和计算来说是没有意义的。

2、浮点数

FLOAT 和 DOUBLE 为浮点类型，DECIMAL 为高精度小数类型。CPU 原生支持浮点运算，但是不支持 DECIMAl 类型的计算，因此 DECIMAL 的计算比浮点类型需要更高的代价。

FLOAT、DOUBLE 和 DECIMAL 都可以指定列宽，例如 DECIMAL(18, 9) 表示总共 18 位，取 9 位存储小数部分，剩下 9 位存储整数部分。

3、字符串

主要有 CHAR 和 VARCHAR 两种类型，一种是定长的，一种是变长的。

VARCHAR 这种变长类型能够节省空间，因为只需要存储必要的内容。但是在执行 UPDATE 时可能会使行变得比原来长，当超出一个页所能容纳的大小时，就要执行额外的操作。MyISAM 会将行拆成不同的片段存储，而 InnoDB 则需要分裂页来使行放进页内。

在进行存储和检索时，会保留 VARCHAR 末尾的空格，而会删除 CHAR 末尾的空格。

> varchar(50) 中 50 的涵义最多存放 50 个字符，varchar(50) 和 varchar(200) 存储 hello所占空间一样，但后者在排序时会消耗更多内存，因为 order by col 采用 fixed_length 计算 col 长度。

4、时间和日期

MySQL 提供了两种相似的日期时间类型：DATETIME 和 TIMESTAMP。

4.1 DATETIME

能够保存从 1001 年到 9999 年的日期和时间，精度为秒，使用 8 字节的存储空间。

它与时区无关。

默认情况下，MySQL 以一种可排序的、无歧义的格式显示 DATETIME 值，例如“2008-01-16 22:37:08”，这是 ANSI 标准定义的日期和时间表示方法。

4.2 TIMESTAMP

和 UNIX 时间戳相同，保存从 1970 年 1 月 1 日午夜（格林威治时间）以来的秒数，使用 4 个字节，只能表示从 1970 年到 2038 年。

它和时区有关，也就是说一个时间戳在不同的时区所代表的具体时间是不同的。

MySQL 提供了 FROM_UNIXTIME() 函数把 UNIX 时间戳转换为日期，并提供了 UNIX_TIMESTAMP() 函数把日期转换为 UNIX 时间戳。

默认情况下，如果插入时没有指定 TIMESTAMP 列的值，会将这个值设置为当前时间。

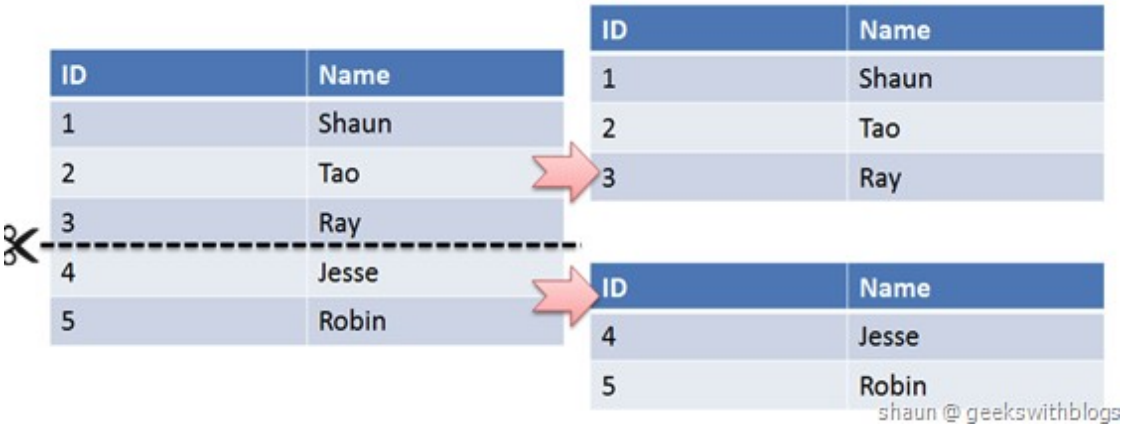
应该尽量使用 TIMESTAMP，因为它比 DATETIME 空间效率更高。

五 切分

1、水平切分

水平切分又称为 Sharding，它是将同一个表中的记录拆分到多个结构相同的表中。

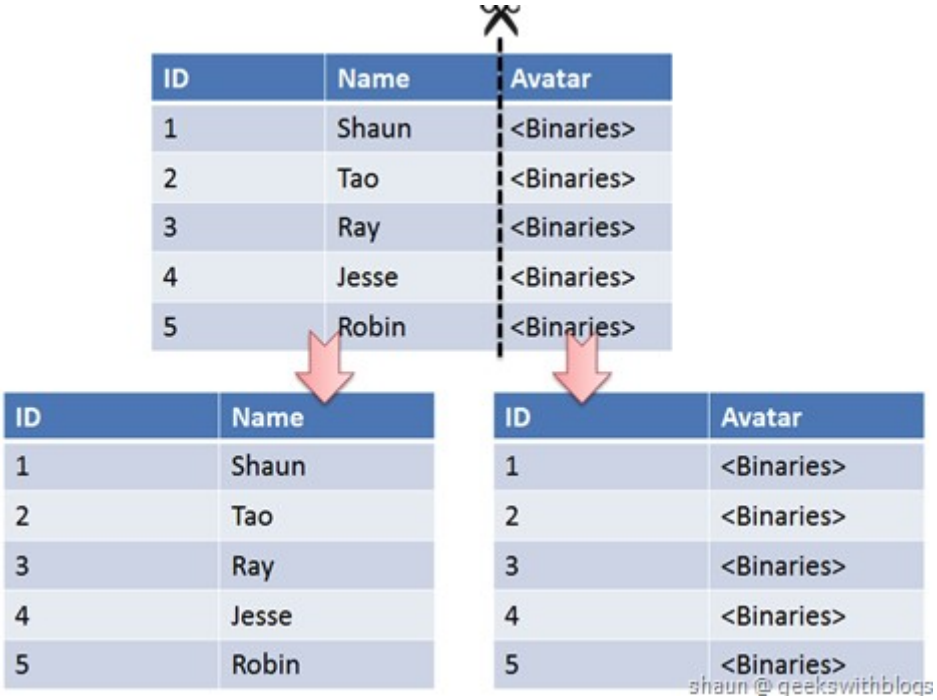
当一个表的数据不断增多时，Sharding 是必然的选择，它可以将数据分布到集群的不同节点上，从而缓存单个数据库的压力。



2、垂直切分

垂直切分是将一张表按列切分成多个表，通常是按照列的关系密集程度进行切分，也可以利用垂直切分将经常被使用的列和不经常被使用的列切分到不同的表中。

在数据库的层面使用垂直切分将按数据库中表的密集程度部署到不同的库中，例如将原来的电商数据库垂直切分成商品数据库、用户数据库等。



3、Sharding 策略

1、哈希取模：hash(key) % N；

2、范围：可以是 ID 范围也可以是时间范围；

3、映射表：使用单独的一个数据库来存储映射关系。

4、Sharding 存在的问题

1. 事务问题

使用分布式事务来解决，比如 XA 接口。

2. 连接

可以将原来的连接分解成多个单表查询，然后在用户程序中进行连接。

3. ID 唯一性

1.使用全局唯一 ID（GUID）

2.为每个分片指定一个 ID 范围

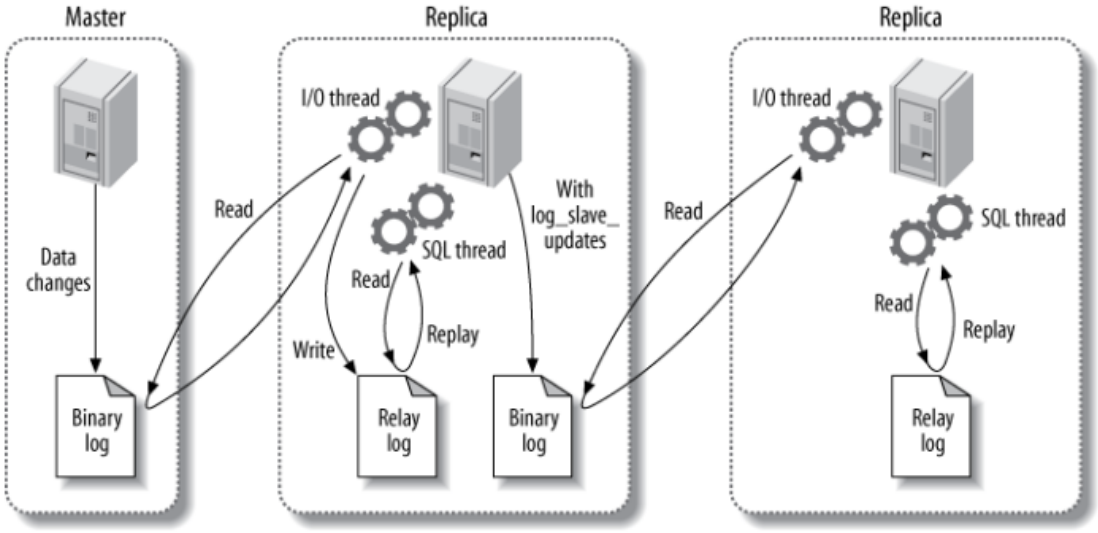
3.分布式 ID 生成器 (如 Twitter 的 Snowflake 算法)

六 复制

1、主从复制

主要涉及三个线程：binlog 线程、I/O 线程和 SQL 线程。

> binlog 线程 ： 负责将主服务器上的数据更改写入二进制日志（Binary log）中。
>
> I/O 线程 ： 负责从主服务器上读取二进制日志，并写入从服务器的重放日志（Relay log）中。
>
> SQL 线程 ： 负责读取重放日志并重放其中的 SQL 语句。



2、读写分离

主服务器处理写操作以及实时性要求比较高的读操作，而从服务器处理读操作。

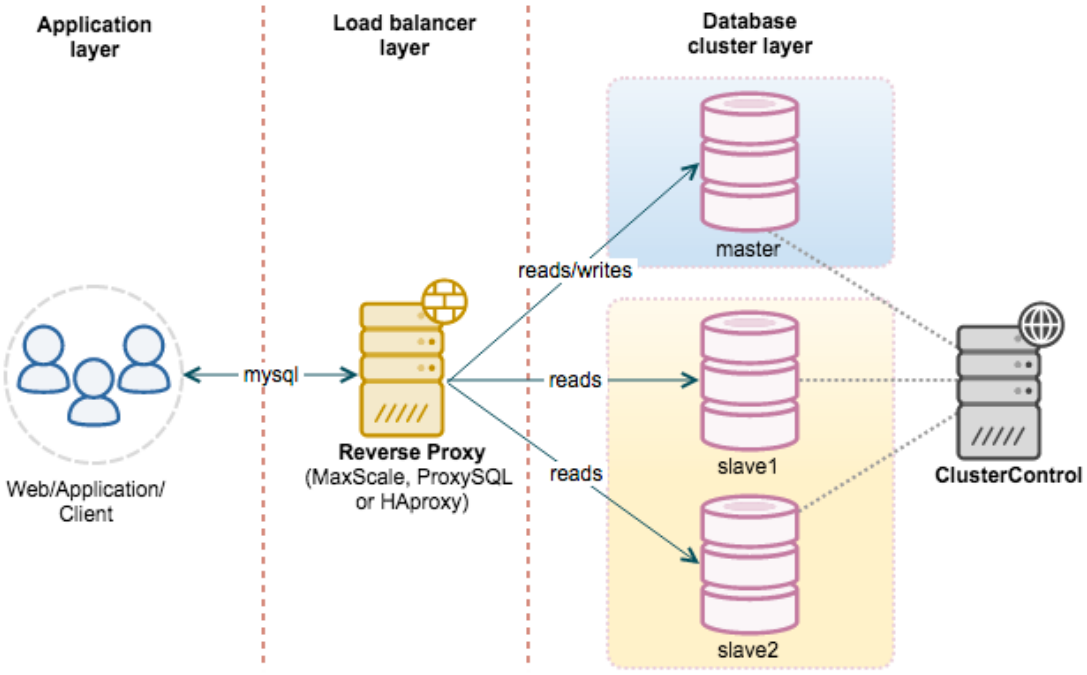
读写分离能提高性能的原因在于：

1、主从服务器负责各自的读和写，极大程度缓解了锁的争用；

2、从服务器可以使用 MyISAM，提升查询性能以及节约系统开销；

3、增加冗余，提高可用性。

读写分离常用代理方式来实现，代理服务器接收应用层传来的读写请求，然后决定转发到哪个服务器。



数据库的分区、分表、分库：
https://blog.csdn.net/qq_28289405/article/details/80576614

七、事务

1、原子性： 事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；

2、一致性： 执行事务前后，数据库从一个一致性状态转换到另一个一致性状态。

3、隔离性： 并发访问数据库时，一个用户的事物不被其他事务所干扰，各并发事务之间数据库是独立的；

4、持久性： 一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库 发生故障也不应该对其有任何影响。

为了达到上述事务特性，数据库定义了几种不同的**事务隔离级别**：

1、READ_UNCOMMITTED（未提交读）：最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读

2、READ_COMMITTED（提交读）：允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生

3、REPEATABLE_READ（可重复读）：对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。

4、SERIALIZABLE（串行化）：最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

> **脏读**： 表示一个事务能够读取另一个事务中还未提交的数据。比如，某个事务尝试插入记录 A，此时该事务还未提交，然后另一个事务尝试读取到了记录 A。
>
> **不可重复读**： 是指在一个事务内，多次读同一数据。
>
> **幻读**： 指同一个事务内多次查询返回的结果集不一样。比如同一个事务 A 第一次查询时候有 n 条记录，但是第二次同等条件下查询却有 n+1 条记录，这就好像产生了幻觉。发生幻读的原因也是另外一个事务新增或者删除或者修改了第一个事务结果集里面的数据，同一个记录的数据内容被修改了，所有数据行的记录就变多或者变少了。

这里需要注意的是：MySQL 默认采用的 REPEATABLE_READ隔离级别 Oracle 默认采用的 READ_COMMITTED 隔离级别。

事务隔离机制的实现基于锁机制和并发调度。其中并发调度使用的是MVCC（多版本并发控制），通过行的创建时间和行的过期时间来支持并发一致性读和回滚等特性。

- **MVCC**

【讲的很好】https://mp.weixin.qq.com/s/Jeg8656gGtkPteYWrG5_Nw

所谓的 MVCC（Multi-Version Concurrency Control ， 多版本并发控制）指的就是在使**用 READ COMMITTD、REPEATABLE READ 这两种隔离级别的事务在执行普通的 SEELCT 操作时访问记录的版本链的过程，这样子可以使不同事务的读-写、写-读操作并发执行，从而提升系统性能。**

READ COMMITTD、REPEATABLE READ这两个隔离级别的一个很大不同就是生成 ReadView 的时机不同，READ COMMITTD 在每一次进行普通 SELECT 操作前都会生成一个 ReadView，而 REPEATABLE READ 只在第一次进行普通 SELECT 操作前生成一个 ReadView，之后的查询操作都重复这个 ReadView 就好了。

八、锁机制

MyISAM 和 InnoDB存储引擎使用的锁

> MyISAM 采用表级锁(table-level locking)。
>
> InnoDB 支持行级锁(row-level locking)和表级锁，默认为行级锁。

表级锁和行级锁对比

表级锁： MySQL 中锁定 **粒度最大** 的一种锁，对当前操作的整张表加锁，实现简单，资源消耗也比较少，加锁快，不会出现死锁。其锁定粒度最大，触发锁冲突的概率最高，并发

度最低，MyISAM 和 InnoDB 引擎都支持表级锁。

行级锁： MySQL 中锁定 **粒度最小** 的一种锁，只针对当前操作的行进行加锁。 行级锁能大大减少数据库操作的冲突。其加锁粒度最小，并发度高，但加锁的开销也最大，加锁慢，会出现死锁。

InnoDB 什么时候使用行级锁？什么时候使用表级锁？

只有通过索引条件检索数据，InnoDB 才使用行级锁，否则，InnoDB 将使用表锁。

1、只有在你增删改查时，匹配的条件字段带有索引时，InnoDB 才会使用行级锁；

2、在你增删改查时，匹配的条件字段不带有索引时，InnoDB 使用的将是表级锁。因为当你匹配条件字段不带有索引时，数据库会全表查询，所以这需要将整张表加锁，才能保证查询匹配的正确性。

在生产环境中我们往往需要满足多人同时对一张表进行增删改查，所以就需要使用行级锁，所以这个时候一定要记住为匹配条件字段加索引。

提到行级锁和表级锁时我们就很容易联想到读锁和写锁，因为只有触发了读写锁，我们才会谈是进行行级锁定还是进行表级锁定。那么什么时候触发读锁，就是在你用 select 命令时触发读锁，什么时候触发写锁，就是在你使用 update、delete、insert 时触发写锁，并且使用 rollback 或 commit 后解除本次锁定。

InnoDB 存储引擎的锁的算法有三种

> 1、Record lock：单个行记录上的锁
>
> 2、Gap lock：间隙锁，锁定一个范围，不包括记录本身
>
> 3、Next-key lock：record+gap 锁定一个范围，包含记录本身

相关知识点：

1、innodb 对于行的查询使用 next-key lock

2、Next-locking keying为了解决Phantom Problem幻读问题

3、当查询的索引含有唯一属性时，将next-key lock降级为record key

4、Gap锁设计的目的是为了阻止多个事务将记录插入到同一范围内，而这会导致幻读问题的产生

5、有两种方式显式关闭gap锁：（除了外键约束和唯一性检查外，其余情况仅使用record lock） A. 将事务隔离级别设置为RC B. 将参数innodb_locks_unsafe_for_binlog设置为1

数据库中锁的详解：
https://blog.csdn.net/qq_35246620/article/details/69943011

九、其他面试题

1、SQL 语句的执行顺序

- **查询语句中的执行顺序**

查询中用到的关键词主要包含六个，并且他们的顺序依次为：select--from--where--group by--having--order by

其中 select 和 from 是必须的，其他关键词是可选的，这六个关键词的执行顺序与 sql 语句的书写顺序并不是一样的，而是按照下面的顺序来执行：

> **from--where--group by--having--select--order by**
>
> from：需要从哪个数据表检索数据
>
> where：过滤表中数据的条件
>
> group by：如何将上面过滤出的数据分组
>
> having：对上面已经分组的数据进行过滤的条件
>
> select：查看结果集中的哪个列，或列的计算结果
>
> order by：按照什么样的顺序来查看返回的数据

2、数据库的三范式是什么?

1、第一范式：强调的是列的原子性，即数据库表的每一列都是不可分割的原子数据项；

2、第二范式：要求实体的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；

3、第三范式：任何非主属性不依赖于其它非主属性。

3、MySQL 数据库 CPU 飙升到 500% 的话他怎么处理?

1、列出所有进程 show processlist，观察所有进程，多秒没有状态变化的(干掉)；

2、查看超时日志或者错误日志 (一般会是查询以及大批量的插入会导致 cpu 与 i/o 上涨，当然不排除网络状态突然断了,导致一个请求服务器只接受到一半，比如 where 子句或分页子句没有发送 ,当然的一次被坑经历)。

十、SQL 语句

连接查询

SQL 中有四种连接查询的方式，但是它们之间其实没有太大的区别，仅仅是查询出来的结果有所不同。

1、**内连接（inner join）**：在两张表进行连接查询的时候，只保留两张表中完全匹配的结果集；

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNoFROM Persons
INNER JOIN Orders
ON Persons.Id_P=Orders.Id_PORDER BY Persons.LastName
```

2、**左外连接（left join）**：在两张表进行连接查询时，会返回左表所有的行，即使在右表中没有匹配的记录；

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNoFROM Persons
LEFT JOIN Orders
ON Persons.Id_P=Orders.Id_PORDER BY Persons.LastName
```

3、**右外连接（right join）**：在两张表进行连接查询时，会返回右表所有的行，即使在左表中没有匹配的记录；

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNoFROM Persons
RIGHT JOIN Orders
ON Persons.Id_P=Orders.Id_PORDER BY Persons.LastName
```

4、**全连接（full join）**：在两张表进行连接查询时，返回左表和右表中所有没有匹配的行。

```
SELECT Persons.LastName, Persons.FirstName, Orders.OrderNoFROM Persons
FULL JOIN Orders
ON Persons.Id_P=Orders.Id_PORDER BY Persons.LastName
```