一、存储引擎

InnoDB

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

采用 MVCC 来支持高并发，并且实现了四个标准的隔离级别，默认级别是可重复读（REPEATABLE READ），并且通过间隙锁（next-key locking）策略防止幻读的出现。间隙锁使得 InnoDB 不仅仅锁定查询涉及的行，还会对索引中的间隙进行锁定，以防止幻影行的插入。

表是基于聚簇索引建立的，它对主键的查询性能有很高的提升。

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、能够自动在内存中创建哈希索引以加速读操作的自适应哈希索引、能够加速插入操作的插入缓冲区等。

通过一些机制和工具支持真正的热备份。其它存储引擎不支持热备份，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。

MyISAM

MyISAM 提供了大量的特性，包括全文索引、压缩表、空间数据索引等。应该注意的是，MySQL 5.6.4 也添加了对 InnoDB 存储引擎的全文索引支持。

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。但在表有读取查询的同时，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。

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，则依然可以继续使用 MyISAM。

比较

- 事务：InnoDB 是事务型的。
- 备份：InnoDB 支持在线热备份。
- 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。
- 并发：MyISAM 只支持表级锁，而 InnoDB 还支持行级锁。
- 其它特性：MyISAM 支持压缩表和空间数据索引。

二、数据类型

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。

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。

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。

VARCHAR 会保留字符串末尾的空格，而 CHAR 会删除。

时间和日期

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

1. DATATIME

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 DATATIME 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。

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。

三、索引

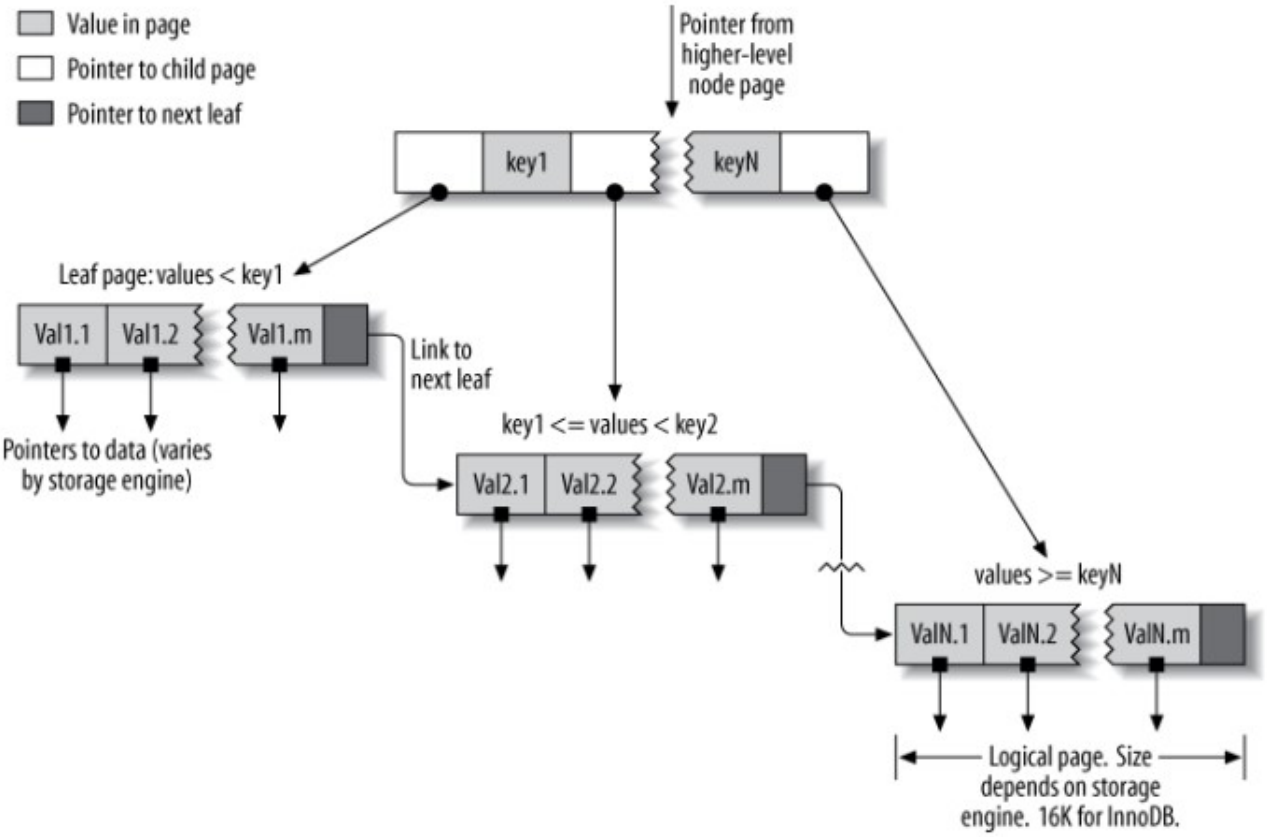
索引是在存储引擎层实现的，而不是在服务器层实现的，所以不同存储引擎具有不同的索引类型和实现。

索引能够轻易将查询性能提升几个数量级。

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，建立和使用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。

索引分类

1. B+Tree 索引



《高性能 MySQL》一书使用 B-Tree 进行描述，其实从技术上来说这种索引是 B+Tree，因为只有叶子节点存储数据值。

B+Tree 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，因此查找速度快很多。

可以指定多个列作为索引列，多个索引列共同组成键。B+Tree 索引适用于全键值、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。

除了用于查找，还可以用于排序和分组。

如果不是按照索引列的顺序进行查找，则无法使用索引。

2. 哈希索引

基于哈希表实现，优点是查找非常快。

在 MySQL 中只有 Memory 引擎显式支持哈希索引。

InnoDB 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+Tree 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。

限制：

- 哈希索引只包含哈希值和行指针，而不存储字段值，所以不能使用索引中的值来避免读取行。不过，访问内存中的行的速度很快，所以大部分情况下这一点对性能影响并不明显；
- 无法用于排序与分组；
- 只支持精确查找，无法用于部分查找和范围查找；
- 如果哈希冲突很多，查找速度会变得很慢。

3. 空间数据索引（R-Tree）

MyISAM 存储引擎支持空间数据索引，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。

4. 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，而不是直接比较索引中的值。

查找条件使用 MATCH AGAINST，而不是普通的 WHERE。

索引的优点

- 大大减少了服务器需要扫描的数据量；
- 帮助服务器避免进行排序和创建临时表（B+Tree 索引是有序的，可以用来做 ORDER BY 和 GROUP BY 操作）；
- 将随机 I/O 变为顺序 I/O（B+Tree 索引是有序的，也就将相邻的数据都存储在一起）。

索引优化

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;
```

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

3. 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;
```

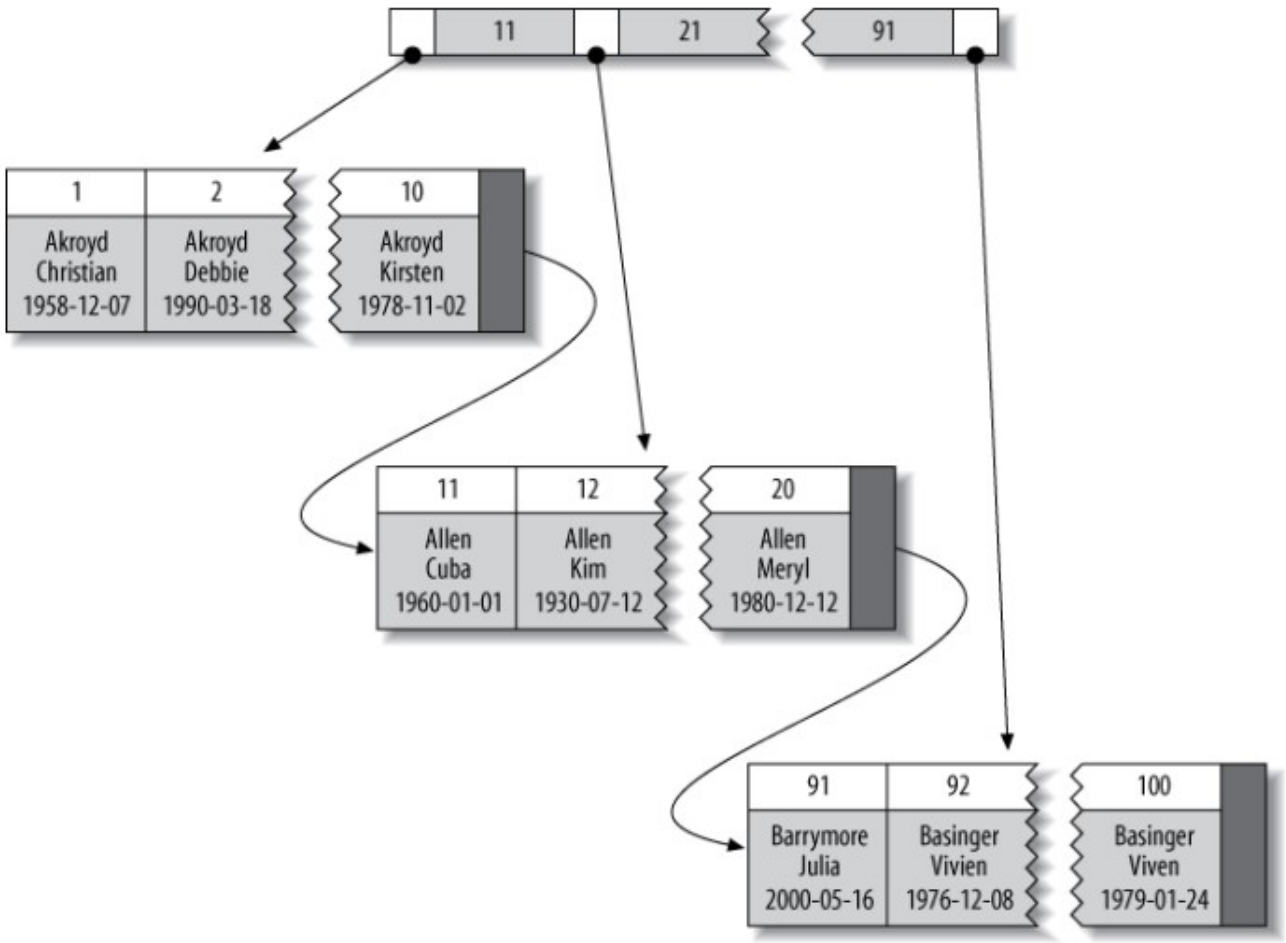
4. 索引列的顺序

让选择性最强的索引列放在前面，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
```

5. 聚簇索引



聚簇索引并不是一种索引类型，而是一种数据存储方式。

术语“聚簇”表示数据行和相邻的键值紧密地存储在一起，InnoDB 的聚簇索引在同一个结构中保存了 B+Tree 索引和数据行。

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。

优点

1. 可以把相关数据保存在一起，减少 I/O 操作。例如电子邮件表可以根据用户 ID 来聚集数据，这样只需要从磁盘读取少数的数据也就能获取某个用户的全部邮件，如果没有使用聚聚簇索引，则每封邮件都可能导致一次磁盘 I/O。
2. 数据访问更快。

缺点

1. 聚簇索引最大限度提高了 I/O 密集型应用的性能，但是如果数据全部放在内存，就没必要用聚簇索引。
2. 插入速度严重依赖于插入顺序，按主键的顺序插入是最快的。
3. 更新操作代价很高，因为每个被更新的行都会移动到新的位置。
4. 当插入到某个已满的页中，存储引擎会将该页分裂成两个页面来容纳该行，页分裂会导致表占用更多的磁盘空间。
5. 如果行比较稀疏，或者由于页分裂导致数据存储不连续时，聚簇索引可能导致全表扫描速度变慢。

6. 覆盖索引

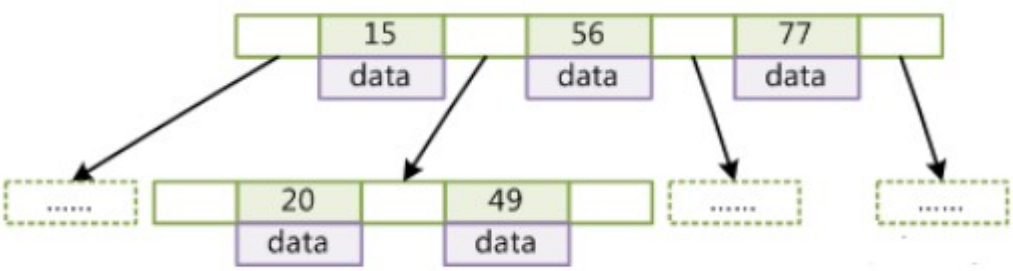
索引包含所有需要查询的字段的值。

优点

1. 因为索引条目通常远小于数据行的大小，所以若只读取索引，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，若二级索引能够覆盖查询，则无需访问聚簇索引。

B-Tree 和 B+Tree 原理

1. B-Tree



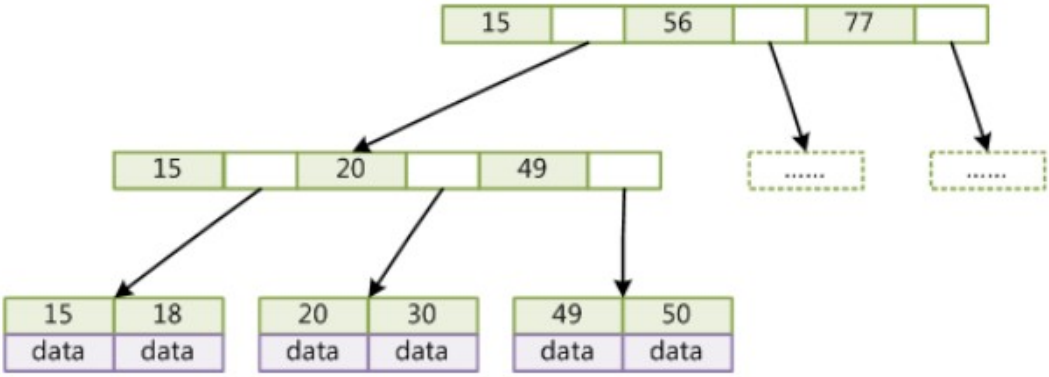
定义一条数据记录为一个二元组 [key, data]，B-Tree 是满足下列条件的数据结构：

- 所有叶节点具有相同的深度，也就是说 B-Tree 是平衡的；
- 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}。

在 B-Tree 中按 key 检索数据的算法非常直观：首先在根节点进行二分查找，如果找到则返回对应节点的 data，否则在相应区间的指针指向的节点递归进行查找。

由于插入删除新的数据记录会破坏 B-Tree 的性质，因此在插入删除时，需要对树进行一个分裂、合并、转移等操作以保持 B-Tree 性质。

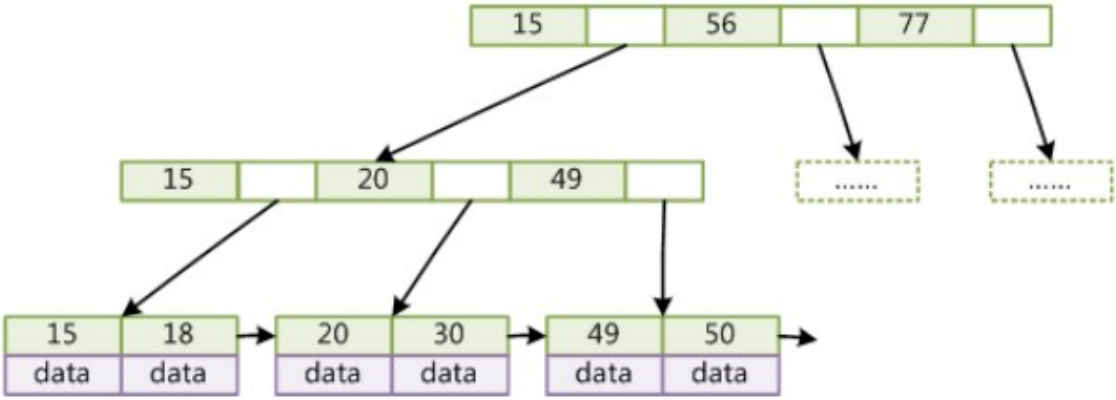
2. B+Tree



与 B-Tree 相比，B+Tree 有以下不同点：

- 每个节点的指针上限为 2d 而不是 2d+1；
- 内节点不存储 data，只存储 key，叶子节点不存储指针。

3. 带有顺序访问指针的 B+Tree



一般在数据库系统或文件系统中使用的 B+Tree 结构都在经典 B+Tree 基础上进行了优化，在叶子节点增加了顺序访问指针，做这个优化的目的是为了提高区间访问的性能。

4. 为什么使用 B+Tree 和 B-Tree

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+Tree B-Tree 作为索引结构，主要有以下两个原因：

（一）更少的检索次数

红黑树和 B+Tree B-Tree 检索数据的时间复杂度等于树高 h，而树高大致为 $O(h)=O(\log_d N)$，其中 d 为每个节点的出度。

红黑树的出度为 2，而 B+Tree 与 B-Tree 的出度一般都非常大。红黑树的树高 h 很明显比 B+Tree B-Tree 大非常多，因此检索的次数也就更多。

B+Tree 相比于 B-Tree 更适合外存索引，因为 B+Tree 内节点去掉了 data 域，因此可以拥有更大的出度，检索效率会更高。

（二）利用计算机预读特性

为了减少磁盘 I/O，磁盘往往不是严格按需读取，而是每次都会预读。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。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，因此速度会非常快。

操作系统一般将内存和磁盘分割成固态大小的块，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，临近的节点也能够被预先载入。

更多内容请参考：MySQL 索引背后的数据结构及算法原理

四、查询性能优化

使用 Explain 进行分析

Explain 用来分析 SELECT 查询语句，开发人员可以通过分析结果来优化查询语句。

比较重要的字段有：

- select_type : 查询类型，有简单查询、联合查询、子查询等
- key : 使用的索引
- rows : 扫描的行数

更多内容请参考：MySQL 性能优化神器 Explain 使用分析

优化数据访问

1. 减少请求的数据量

（一）只返回必要的列

最好不要使用 SELECT * 语句。

（二）只返回必要的行

使用 WHERE 语句进行查询过滤，有时候也需要使用 LIMIT 语句来限制返回的数据。

（三）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，特别要查询的数据经常被重复查询，缓存可以带来的查询性能提升将会是非常明显的。

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。

重构查询方式

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。

```
DELEFT 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
```

2. 分解关联查询

将一个关联查询分解成对每一个表进行一次单表查询，然后将结果在应用程序中进行关联，这样做的好处有：

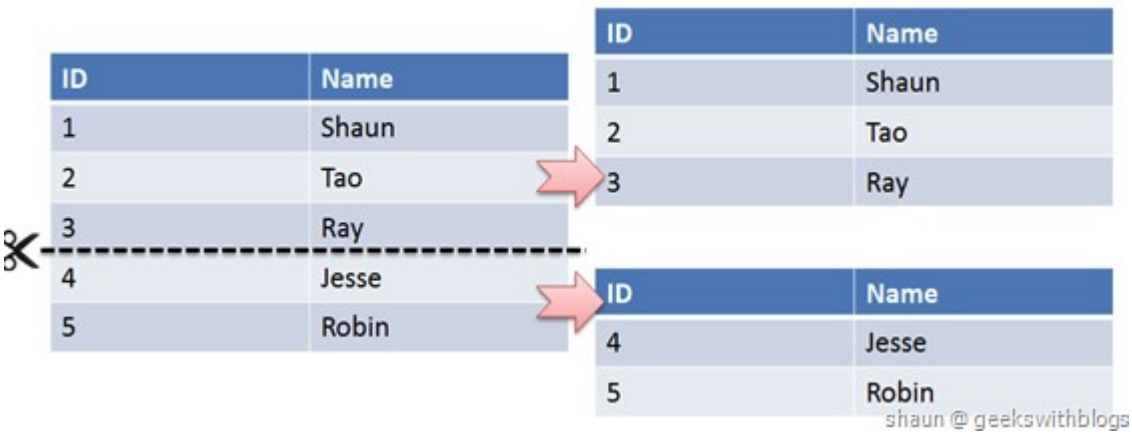
- 让缓存更高效。对于关联查询，如果其中一个表发生变化，那么整个查询缓存就无法使用。而分解后的多个查询，即使其中一个表发生变化，对其它表的查询缓存依然可以使用。
- 减少锁竞争；
- 在应用层进行关联，可以更容易对数据库进行拆分，从而更容易做到高性能和可扩展。
- 查询本身效率也可能会有所提升。例如下面的例子中，使用 IN() 代替关联查询，可以让 MySQL 按照 ID 顺序进行查询，这可能比随机的关联要更高效。
- 分解成多个单表查询，这些单表查询的缓存结果更可能被其它查询使用到，从而减少冗余记录的查询。

```
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);
```

五、切分

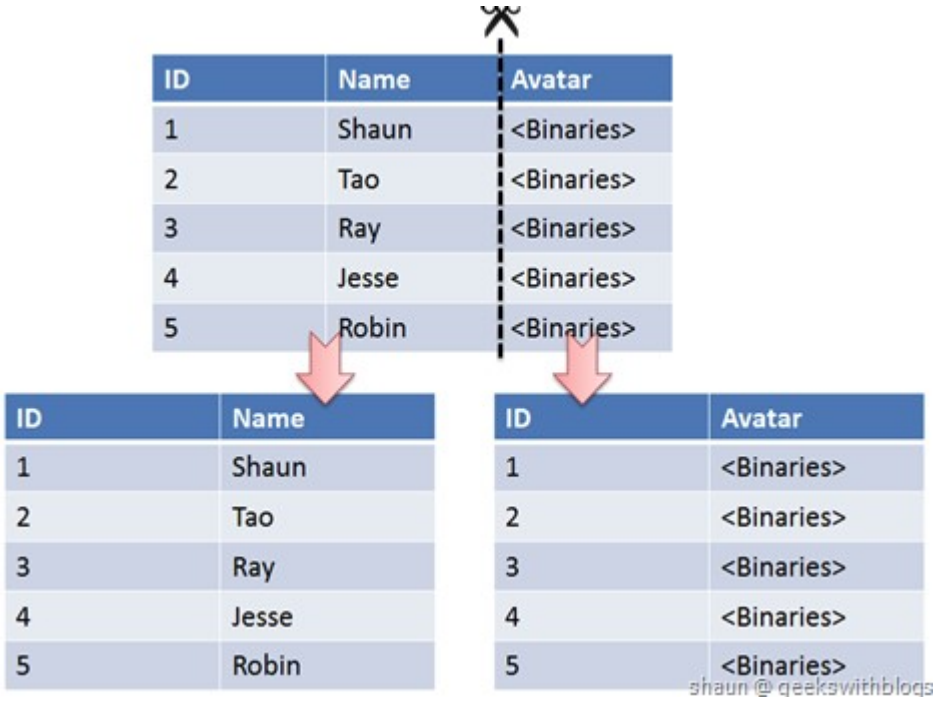
水平切分



水平切分就是就是常见的 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。

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。

也可以在数据库的层面使用垂直切分，它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 payDB、用户数据库 userBD 等。

Sharding 策略

- 哈希取模：hash(key) % NUM_DB
- 范围：可以是 ID 范围也可以是时间范围
- 映射表：使用单独的一个数据库来存储映射关系

Sharding 存在的问题

1. 事务问题

使用分布式事务。

2. JOIN

将原来的 JOIN 查询分解成多个单表查询，然后在用户程序中进行 JOIN。

3. ID 唯一性

- 使用全局唯一 ID：GUID。
- 为每个分片指定一个 ID 范围。

- 分布式 ID 生成器 (如 Twitter 的 Snowflake 算法)。

更多内容请参考：

- How Sharding Works
- 大众点评订单系统分库分表实践

参考资料

- BaronScbwartz, PeterZaitsev, VadimTkacbenko, 等. 高性能 MySQL[M]. 电子工业出版社, 2013.
- 20+ 条 MySQL 性能优化的最佳经验
- 服务端指南 数据存储篇 | MySQL（09） 分库与分表带来的分布式困境与应对之策
- How to create unique row ID in sharded databases?
- SQL Azure Federation – Introduction