# 百度 MySQL 数据库 SQL 编写指南

2011.5

# 版 本

| 版本 | 发布日期 | 修订内容 | 主要修订人 |
|---|---|---|---|
| v 0.1 | 2011.1.17 | 创建 SQL 编写指南 | 李京生、尹博学、李娜娜、朱金清 |
| v 0.2 | 2011.1.28 | 通过 DBA 内部评审 | 王龙、李京生、尹博学、李娜娜、朱金清 |
| v0.3 | 2011.2.28 | 通过 DBA、OPTC 评审 | 王龙、李京生、尹博学、李娜娜、朱金清、李鹏 |
| v1.0 | 2011.5.18 | 通过总 TC 评审后终定稿 | 王龙、李京生、尹博学、李娜娜 |
| | | | |

# 目 录

# 第 1 章. 序言

## 1.1. 背景与目标

随着百度业务的发展，使用数据库的产品线数量及应用规模不断扩大，由于缺乏相关 SQL 准则，线上对 SQL 应用存在诸多不规范不统一，导致 SQL 性能、稳定、安全性问题，目前已成数据库性能低下首要素，直接影响用户体验。

鉴于此，DBA 为指导百度范围内数据库 SQL 编写制定此指南，以减少因 SQL 应用导致的数据库性能、稳定、安全性问题，形成统一规范标准，为数据库据准入建立基础，定名为《百度 MySQL 数据库 SQL 编写指南》。

同时为更简单通俗介绍 SQL 编写，在本指南中有大量的例子，所有的 SQL 例子涉及的库表见附录中的例子库表。

## 1.2. 适用范围

本指南适用于百度所有 MySQL 数据库的开发设计、SQL 编写等。

# 第 2 章. 库表设计

## 2.1. 高效的设计模型

数据库设计是指对于一个给定的应用环境，构造合理的数据库模式，建立数据库及其应用系统，有效存储数据，满足用户信息要求和处理要求。

数据库设计在开发过程中处于一个非常重要的地位。一个高效的数据库模型是非常重要和必要的。

### 2.1.1. 设计原则说明

#### 2.1.1.1. 完整性

数据库完整性是指数据库中数据的正确性和相容性。数据库完整性是由完整性约束来保证的。数据库完整性约束通过 DBMS 或者应用程序来实现的。

数据库完整性对于数据库应用系统非常关键，其作用主要体现在以下几个方面：

1. 数据库完整性约束能够防止合法用户使用数据库时向数据库中添加不合语义的数据。
2. 利用基于 DBMS 的完整性控制机制来实现业务规则，易于定义，容易理解，而且可以降低应用程序的复杂性，提高应用程序的运行效率。同时，基于 DBMS 的完整性控制机制是集中管理的，因此比应用程序更容易实现数据库的完整性。
3. 合理的数据库完整性设计，能够同时兼顾数据库的完整性和系统的效能。比如装载大量数据时，只要在装载之前临时使基于 DBMS 的数据库完整性约束失效，此后再使其生效，就能保证既不影响数据装载的效率又能保证数据库的完整性。
4. 在应用软件的功能测试中，完善的数据库完整性有助于尽早发现应用软件的错误。

完整性主要包括实体完整性，参照完整性以及用户定义完整性。它们的实现机制如下：

1. 实体完整性：主键
2. 参照完整性：
   - 父表中删除数据：级联删除；受限删除；置空值
   - 表中插入数据：受限插入；递归插入
   - 父表中更新数据：级联更新；受限更新；置空值
   - DBMS 对参照完整性可以有两种方法实现：外键实现机制(约束规则)和触发器实现机制
3. 用户定义完整性： 指针对某一具体关系数据库的约束条件，它反映某一具体应用所涉及的数据必须满足的语义要求，比如表中利用 check 关键字定义 age 的取值范围。

了解如上的基本知识外，在设计数据库完整性时，有一些原则需要注意：

1. 根据数据库完整性约束的类型确定其实现的系统层次和方式，并提前考虑对系统性能的影响。一般情况下，静态约束应尽量包含在数据库模式中，而动态约束由应用程序实现。其中静态约束主要是指数据库确定状态时的数据对象所应满足的约束条件，它是反映数据库状态合理性的约束，这是最重要的一类完整性约束，比如列取值，列类型，空值等的约束。而动态约束主要是指数据库从一种状态转变为另一种状态时，新、旧值之间所应满足的约束条件，它是反映数据库状态变迁的约束，比如一些用户自定义类的完整性约束。
2. 实体完整性约束、参照完整性约束是关系数据库最重要的完整性约束，在不影响系统关键性能的前提下可考虑使用。

3. 要慎用目前主流 DBMS 都支持的触发器功能，一方面由于触发器的性能开销较大，另一方面，触发器的多级触发不好控制，容易发生错误。在不支持事务的引擎下建议使用 before 型触发器。
4. 要根据业务规则对数据库完整性进行细致的测试，以尽早排除隐含的完整性约束间的冲突和对性能的影响。
5. 为了在数据库和应用程序代码之间提供另一层抽象，可以为应用程序建立专门的视图而不必非要应用程序直接访问数据表。这样做还等于在处理数据库变更时给你提供了更多的自由。
6. 在 MySQL 中由于一些约束和存储引擎有关，比如外键约束的需求，因此设计时需要注意该表存储引擎的选择。
7. 目前 MySQL5.0 版本语法支持 check，经过测试，可以执行，但是约束不生效。

#### 2.1.1.2. 性能

性能是衡量一个系统的关键因素，在设计阶段就在性能方面就应该多关注，尽量减少后期的烦恼。

在数据库设计阶段，性能上的考虑时需要注意：不能以范式作为唯一标准或者指导，在设计过程中，需要从实际需求出发，以性能提升为根本目标来展开设计工作，一些时候为了提升性能，甚至会做反范式设计。

另外还有一些设计上的方法和技巧：

1. 设置合理的字段类型和长度。字段类型在满足需求后应尽量短，比如，能用 int 就尽量不要用 bigint。另外不同数据库在 varchar 和 text 类型在长度和性能上也是不同的，选择时要谨慎。
2. 选择高效的主键和索引。由于对表记录的读取都是直接或者间接地通过主键或索引来获取，因此应该该根据具体应用特性来设计合理的主键或索引。同时索引长度的也应该关注，尽量减少索引长度。有关索引方面的详细介绍请见第三章索引。
3. 减少数据量。除了减少字段长度外，可考虑数据压缩。比如 MyISAM 表的压缩操作，压缩后对磁盘利用率，IO 缓存等都有明显的性能提升。
4. 适度冗余。适度的冗余可以避免关联查询，减少 join 查询。
5. 精简表结构。表结构如果太过复杂，会引起业务上处理复杂，同时也可能会引起并发问题。如果根据业务特性拆分成多个表，可以避免高并发下的锁表现象。

#### 2.1.1.3. 扩展性

在大规模系统中，除了性能，可扩展性也是设计的关键点，而数据库表扩展性主要包含表逻辑结构、功能字段的增加、分表等。在扩展性上要把握的原则如下：

1. 一表一实体。如果不同实体之间有关联时，可增加一个单独的表，不会影响以前的功能。
2. 扩展字段。在表数据较小时增加一个字段可以很快完成，但是在表很大时，增加字段会比较困难。因此在设计时可考虑选择预留扩展字段。
3. 分表设计。也就是水平切分。在设计阶段应该考虑数据的增长情况，并根据数据特性以及数据之间关系选择合适的切分策略。有关分表的更详细介绍具体可详见章节 2.1.4 的介绍。

### 2.1.2. 正则化与非正则化的选择

#### 2.1.2.1. 正则化和非正则化

数据库正则化是指消除冗余、有效组织数据、减少在数据操作期间潜在的不规则和提高数据一致性。在正则化数据库中，每个元素只会被存储一次，而在非正则化数据库中，信息是重复的或者保存在多个地方的。

比如在设计消费者与账号信息表时，如果消费者的信息比如姓名、联系方式等都存储在这张消费者与账号信息表中时，当“张三”有多个账号时，“张三”的基本信息就被多次存储，这样设计是不符合正则化的。

#### 2.1.2.2. 正则化的利弊

正则化包括根据设计规则创建表并在这些表间建立关系；通过取消冗余度与不一致相关性，该设计规则可以同时保护数据并提高数据的灵活性。

一般情况下正则化有如下优点：

1. 正则化更新通常比非正则化更新快。
2. 当数据被很好正则化后，数据量很少，重复数据小，更新量也会变少。
3. 正则化表通常较小，更容易装入内存性能因此也会更好。

当然正则化也是存在一些缺点：在正则化结构上的非一般性查询，一般都需要至少一个联接。

#### 2.1.2.3. 非正则化的利弊

非正则化由于数据都在一个表中，避免了联接，同时也能运行更有效率的索引策略，因此性能不错。

当然非正则化在数据冗余，数据独立性,相关性以及提高数据一致性方面稍差。

#### 2.1.2.4. 正则化和非正则化的结合

正则化和非正则化各有利弊，在具体应用中通常是结合两种方案。

### 2.1.3. 存储引擎的选择

MySQL 支持数个存储引擎作为对不同表的类型的处理器，MySQL 中的插件式存储引擎架构是非常有特色的亮点。如下先列出常见存储引擎的部分说明，后面各小节将对最常用的存储引擎进行更详细介绍。

| MySQL 引擎 | 说明 |
|---|---|
| MyISAM | 内存中只缓存索引，不缓存数据；<br>不支持事务；<br>只支持表级锁，支持 insert 操作和 select 操作并发进行；<br>适合读多写少的应用，如 web；<br>支持全文索引； |
| InnoDB | 索引和数据都可以缓存到内存中；<br>支持事务；<br>支持行级锁，可实现更高的并发度；<br>支持故障恢复；<br>支持外键约束；<br>支持 4 种不同的事务隔离级别； |
| Memory | 所有数据均存在内存中，可提供高速的查询效率；<br>一旦重启数据库(包括死机)，将丢失所有数据；<br>不支持 blob 和 text 类型；<br>数据大小必须小于内存容量；<br>其它特性类似于 MyISAM； |
| Merge | 将一系列等同的 MyISAM 表以逻辑方式组合在一起，并作为 1 个对象引用它们。<br>高并发时，会导致适用的文件句柄数剧增；<br>适用于数据仓储等环境。 |

| | |
|---|---|
| Archive | 可将大量的数据压缩存储；<br>不支持索引；<br>支持 insert 和 select，不支持 delete，replace 和 update；<br>支持行级锁； |
| Federated | 能够将多个分离的 MySQL 服务器链接起来，从多个物理服务器创建一个逻辑数据库；<br>不支持事务；<br>不支持 alter table 等直接修改表结构的 DDL 操作；<br>不支持 prepare 语句；<br>不支持查询缓存； |
| Blackhole | 不会存储任何数据，仅记录日志；<br>适用于日志统计、过滤、同步中间层等； |
| CSV | 将数据以逗号分隔的格式存储在文本文件中；<br>不支持索引； |

## 2.1.3.1. MyISAM 引擎

MyISAM 引擎是 MySQL 的默认存储引擎。 MyISAM 提供的特征包括全文检索，压缩，空间函数。但是不支持事务和行级锁。

MyISAM 将每个表存储为两个文件：数据文件和索引文件。两个文件的扩展名分别为.MYD 和.MYI。MyISAM 的格式是平台通用的，所以用户可以在不同架构上相互拷贝数据文件和索引文件。

MyISAM 表可以包含动态行和静态行。MySQL 会根据表定义决定选用何种行格式。MyISAM 表的可容纳的行总数，一般只受限于数据库的可用磁盘空间大小，以及操作系统允许创建的最大文件大小。

MyISAM 表的特征：

1. 加锁和并发：MyISAM 表对整张表加锁，读操作会加共享锁，写操作会加排他锁。但是支持查询时在该表内插入新行。
2. 修复：用户可以运用 check table mytable 和 repair table mytable 来检查并修复错误。当服务离线时，也可以使用 MyISAMchk 命令行工具检查和修复表。
3. 索引特性：MyISAM 表支持 B-Tree,R-Tree,以及 full-text 索引。值得注意的是 MyISAM 表索引长度不能超过 1000 字节。
4. 延迟更新索引：当使用 delay_key_write 创建的 MyISAM 表时，在查询结束后，不会将索引的改变数据立即写入磁盘，而是先存放在键缓冲区中缓存。在清理缓冲区或者关闭表时才会将索引块刷到磁盘。对于数据经常改变，并且使用频繁的表，该模式很大程度上提高了表的处理性能。不过，如果服务器或者系统崩溃时，索引肯定损坏，并需要修复。这时可以使用如上的修复工具进行修复。延迟特性可以被全局配置也可以为个别表单独配置。
5. 压缩操作：压缩表是不能被改变的(除非是解压后，修改然后再压缩表)。压缩表占用的磁盘空间很小,降低磁盘 I/O，提高表的处理性能。同时压缩也支持索引，但注意索引也是只读的。

## 2.1.3.2. Merge 引擎

Merge 引擎是 MyISAM 表的变种。Merge 引擎类型允许你把许多结构相同的表合并为一个表。然后，你可以执行查询，从多个表返回的结果就像从一个表返回的结果一样。前提是每一个合并的表必须有同样的表定义。

另外还有几点说明：

1. 此表类似于 SQL 中的 union 机制。
2. 此表结构必须与基本表完全一致，包括列名、顺序。union 表必须同属一个 database。
3. 基本表类型必须是 MyISAM。

4. 可以通过修改.MRG 文件来修改 Merge 表，每个基本表的名字占一行。注意：修改后要通过 flush tables 刷新表缓存。
5. 对基本表的更改可以直接反映在此表上。
6. insert_method 的取值可以是： 针对 MySQL4.0 以后的版本，no 表示不允许插入； first 表示插入到 union 中的第一个表； last 表示插入到 union 中的最后一个表。
7. 定义在 Merge 表上面的约束没有任何作用，约束是由基本表控制的，例如两个基本表中存在着同样的一个 key 值，那么在 Merge 表中会有两个一样的 key 值。

## 2.1.3.3. InnoDB 引擎

MySQL 中使用最为广泛的除了 MyISAM 表之外就是 InnoDB 引擎了。InnoDB 引擎是专为事务处理设计的。

InnoDB 将所有的数据共同存储在一个或几个数据文件中，这种数据文件一般称为表空间。在 MySQL4.1 及更新版本中 InnoDB 支持每个表和相关索引存储为单独的分离的文件。

InnoDB 之所以这么受宠和它如下的特点有很大关系：

1. 支持事务：并且支持实现了四种标准隔离级别，默认的级别是 repeatable read,同时在这个级别上使用间隙锁防止“幻读”：不仅对查询中读取的行加锁，还对索引结构中的间隙进行加锁，以防止幻影插入。
2. 数据多版本读取：InnoDB 采用 MVCC 机制获取高并发性能。
3. 锁定机制改进：InnoDB 改变了 MyISAM 表的锁机制，实现行锁。InnoDB 的行锁机制是通过索引来完成的，同时数据库中绝大部分查询都是通过索引来检索的。
4. 外键：InnoDB 实现了外键，使在数据库端控制部分数据的完整性成为可能。

## 2.1.3.4. Memory 引擎

Memory 引擎，顾名思义就是一个将数据存储在内存中的存储引擎。所以一旦 MySQL 或者主机崩溃后，Memory 的表只有表结构仍然会保留，相关数据将全部丢失。

Memory 引擎有以下特点

1. 支持 hash 索引(Memory 引擎的默认索引)，它使得查询非常快速。
2. 使用表级锁，只支持较低的并发。
3. 不支持 text 或 blob 类型，也只支持固定大小行。
4. 重启后，如果打开 binlog 会写一条 truncate table 又保证主从数据一致由于不写盘，写入速度很可观。

## 2.1.3.5. 选择合适的引擎

在设计时需要决定选择何种存储引擎来存储数据。如果不在设计阶段就考虑这个问题，在后续工作中可能就会面临复杂的问题。用户可能会发现默认配置的引擎不能满足自己的某种需要。引擎的选择需要用户从整体上正确理解应用，并且预测应用的增长需求。如下列出选择引擎时的主要考虑因素：

1. 事务：如果应用有事务需求，那么 InnoDB 会是一个好的选择。
2. 并发：如何更好地满足并发需求，需要根据应用的工作负载来定。如果只是需要并发的插入操作或读操作，MyISAM 会是一个正确的选择。如果需要的是混合并发操作，并且操作之间要互不干扰，那么支持行级锁的引擎将是更好的选择。
3. 其他特有特性：比如全文检索只有 MyISAM 引擎支持等。

在具体应用中，比如在商务应用的一些计费服务中，需要对事务的支持，那么 innodb 引擎将是首选。总之在具体应用中需要多结合业务以及存储引擎的特点来做出选择。

# 2.1.4. 表容量设计_数据切分

## 2.1.4.1. 使用场景

随着数据库表中数据日积月累越来越多，数据库会越来越大，表记录数也会达到千万甚至亿级别，数据库表的访问效率下降明显，导致外层应用的访问效率非常差，访问时间急剧上升，用户体验下降。此时就必须使用数据切分来解决这个瓶颈了。

数据切分就是指通过某种特定的条件，将存放在同一个数据库中的数据分散存放到多个数据库上面，以达到分散单台设备负载的效果。数据切分根据切分规则的类型，可以分为两种模式。一种是按照不同的表切分到不同的数据库上或不同的表上，这种切分称为垂直切分；另外一种是根据表中数据的逻辑关系，将同一个表的数据按照某种规则切分到多台数据库上或不同的表上，这种称为水平切分。

## 2.1.4.2. 垂直切分

垂直切分就是要把表按模块划分到不同数据库或不同表中。这种切分在大型网站的演变过程中是很常见的。当一个网站还在很小的时候，只有少量的人来开发和维护，各模块和表都在一起，当网站不断丰富和壮大的时候，也会变成多个子系统来支撑，这时就有按模块和功能把表划分出来的需求。

一个架构设计较好的应用系统，其总体功能肯定是由多个功能模块所组成的，而每一个功能模块所需要的数据对应数据库中的就是一张表或者多个表。在架构设计中，各个功能模块互相之间的交互点越统一、越少，系统的耦合度就越低，系统各个模块的维护性及扩展性也就越好。这样的系统，实现数据的垂直切分也就越容易。比如公司的轩辕系统中，直销模块和工作流模块数据库原本在一起的，但随着推广工作进行，导致该数据库的压力较大，因此对直销模块和工作流模块进行了拆分工作。由于前期设计时这两个模块之间的耦合度较低，在拆分过程中也很顺利。

功能模块越清晰，耦合度越低，数据垂直切分的规则定义也就越容易。完全可以根据功能模块来进行数据切分，不同功能模块的数据存放在不同的数据库或不同表上，可以很容易就避免跨库的 join 存在，同时系统架构也是非常清晰的。

当然，很难有系统能够做到所有功能模块使用的表完全独立，根本不需要访问对方的表，或者不需要将两个模块的表进行 join 操作。这种情况下，就必须根据实际的应用场景来进行评估权衡。迁就应用程序就需要将待 join 的表的相关模块存放在同一个数据库表中，或者让应用程序做更多的事情——完全通过模块接口取得不同数据库表中的数据，然后在程序中完成 join 操作。

如果让多个模块集中共用数据源，实际上也是间接默认了各模块架构耦合度增大的发展，可能会恶化以后的架构。

所以，在数据库进行垂直切分的时候，如何切分、切分到什么样的程度，是一个比较考验人的难题，这只能在实际应用场景中通过平衡各方面的成本和收益，才能分析出一个真正合适自己的切分方案。

垂直切分的优缺点：

优点：

- 数据库表的切分简单明了，切分规则明确；
- 应用程序模块清晰明确，整合容易；
- 数据维护方便易行，容易定位；

缺点：

- 部分表关联无法在数据库级别完成，需要在程序中进行；
- 对于访问极其频繁且数据量超大的表仍然存在性能瓶颈，不一定能满足要求；

- 事务处理相对复杂；
- 切分达到一定程度之后，扩展性会受到限制；
- 过度切分可能会导致系统过于复杂而难以维护；

#### 2.1.4.3. 水平切分

水平切分可以简单理解为按照数据行的切分，就是将表中的某些行切分到一个数据库或表，而另外的某些行又可以切分到其他的数据库或表中。为了能够比较容易判定各行数据被切分到哪个库或表中，切分需要按照某种特定的规则进行。

水平切分的切分标准可以按照数据范围分，比如 1-100 万一个表，100 万-200 万又是一个表；也可以按照时间顺序来切分，比如一年的数据归到一张表中等；也可以按照地域范围来分，比如按照地市来分，每个或多个地市一个库等；也可以按照某种计算公式来切分，比较简单的比如取模的方式，如根据用户 id 进行水平切分，可通过对 ID 被 2 取模，然后分别存放在不同的表中，这样关联时也非常方便。公司著名的凤巢拆库就是采用取模方式进行的拆分。

水平切分的优缺点：

优点：

- 表关联基本能够在数据库端全部完成；
- 不会存在某些超大型数据量和高负载的表遇到瓶颈的问题；
- 应用程序端整体架构改动相对较少；
- 事务处理相对简单；
- 只要切分规则能够定义好，扩展性一般不会受到限制；

缺点：

- 切分规则相对复杂，很难抽象出一个能满足整个数据库的切分规则；
- 后期的维护难度有所增加，人为手工定位数据较困难。
- 应用系统各模块耦合度非常高，可能会对后面数据的迁移切分造成一定的困难。
- 若切分不合理，会造成数据表的冷热不均现象。

#### 2.1.4.4. 垂直与水平联合切分

由上面可知垂直切分能更清晰化模块划分，区分治理，水平切分能解决大数据量性能瓶颈问题。本节将结合垂直切分和水平切分的优缺点，进一步完善整体架构，并提高系统的扩展性。

在实际的应用场景中，除了那些负载并不是太大、业务逻辑相对简单的系统可以通过上面两种切分方法之一来解决扩展性问题，但是大部分的业务逻辑复杂、系统负载大的系统，都无法通过上面任何一种数据的切分方法来实现较好的扩展性。这就需要将上述两种方法结合使用，不同的场景使用不同的切分方法。

每一个应用系统的负载都是一步一步增长起来的，在开始遇到性能瓶颈时，大多架构师和 DBA 都会选择数据的垂直切分。然而随着业务的不断扩张，系统负载的持续增长，在系统稳定一段时间之后，经过垂直切分的数据库集群可能再次不堪重负，遇到性能瓶颈。

这时再进一步细分模块？随着模块的不断细化，应用系统的架构会越来越复杂，整个系统很可能会出现失控的局面。这时就必须利用数据水平切分的优势来解决问题。在垂直切分的基础上利用水平切分来避开垂直切分的弊端，解决系统不断扩大的问题。而水平切分的弊端也已经被之前的垂直切分解决了。

在大型的应用系统上，垂直和水平切分基本上是并存的，而且是不断的交替进行的，以增加系统的扩展能力。我们在应对不同的应用场景时就要充分考虑这两种切分方法的局限及优势，在不同的时期使用不同的方法。

联合切分的优缺点：

优点

1. 可以充分利用垂直和水平切分各自的优势而避免各自的缺陷；
2. 让系统扩展性得到最大化提升。

缺点

1. 数据库系统架构会比较复杂，维护难度更大；
2. 应用程序架构也更复杂。

# 2.1.5. 字符集选择

## 2.1.5.1. 字符集介绍

MySQL 支持 30 多种字符集的 70 多种校对规则。字符集和它们的默认校对规则可以通过 show character set 语句显示。

## 2.1.5.2. 字符集的选择

MySQL5.0 目前支持几十种字符集，utf-8 是 MySQL5.0 支持的唯一 unicode 字符集，但版本是 3.0，不支持 4 字节的扩展部分。面对众多的字符集我们该如何选择呢？

在选择数据库字符集时，主要考虑因素包括：

1. 满足应用支持语言的要求，如果应用要处理各种各样的文字，或者将发布到使用不同语言的国家或地区，就应该选择 unicode 字符集。对 MySQL 来说，就是 utf-8。
2. 如果应用中涉及已有数据的导入，就要充分考虑数据库字符集对已有数据的兼容性。
3. 如果数据库只需要支持一般的中文，数据量很大，性能要求很高，那就应该选择双字节定长编码的中文字符集，比如 gbk。
4. 如果数据库需要做大量的字符运算，如比较、排序等，选择定长字符集可能更好，因为定长字符集的处理速度要比变长字符集处理的速度快些。

如果所有客户端程序都支持相同的字符集，应该优先选择该字符集作为数据库字符集。这样可以避免因字符集转换带来的性能开销和数据损失。

# 2.2. 合适的数据类型

# 2.2.1. 数据类型的选择原则

## 2.2.1.1. 使用正确的数据类型

使用正确的数据类型的第一步是大致决定需要存储的数据类型的大类：数字、字符串、时间等。这通常很直观，但也有特殊情况(后续会有特殊情况的介绍)。

第二步是确定特定的数据类型。许多 MySQL 类型同属一类，它们能够保存同类的数据，但是在存储范围、精度或物理空间(磁盘上或内存中)却不相同。一些数据类型还有特殊的行为或属性。例如 datetime 和 timestamp 能保存同样类型的数据：日期和时间，精度为秒。然而，timestamp 使用的空间只有 datetime 一半，还能保存

时区，拥有特殊的自动更新能力。而另一方面，它允许的范围要小的多，并且在某些时候，它的特殊功能会成为障碍。

#### 2.2.1.2. 列类型的选择原则

上一节介绍了选择正确数据类型的原则。MySQL 支持多种不同的数据类型，数据库表的设计者在保证正确性的前提下，为列选择合适的数据类型对于获得高性能至关重要。对于“选择何种数据类型合适”这个问题，下面的几个简单的原则会为选择提供很大的帮助：

1. 更小的数据类型更好：一般来说，要试着使用正确地存储和表示数据的最小类型。更小的数据类型通常更快，因为他们使用了更少的磁盘空间、内存和 CPU 缓存，而且需要的 CPU 周期也更少。但是要确保不会低估需要保存的值。如果在设计初期选定一种数据类型后，后期增加数据类型的范围是一件极其费时费力的工作。如果在设计初期不确定需要什么数据类型，就选择你认为不会超出范围的最小类型。
2. 简单更好：越简单的数据类型，需要的 CPU 周期就越少。例如，比较整数的代价小于比较字符，因为字符集选择和相应的排序规则使字符比较更复杂。比如应该使用 MySQL 内建的类型来保存日期和时间，而不是使用字符串；应该使用整数来保存 IP 地址。
3. 尽量避免 null：要尽可能地把字段(field)定义为 not null。需要注意的是可空列(nullable column)是 MySQL 的默认选项。除非真的要保存 null，否则就把；列定义为 not null。MySQL 难以优化引用了可空列的查询。因为 null，不等于任何值。它会使索引，索引统计和值更加复杂。可空列需要更多的存储空间，还需要在 MySQL 内部进行特殊处理。当可空列被索引的时候，每条记录都需要一个额外的字节，还可能导致 MyISAM 中固定大小的索引变成可变大小的索引。即使要在表中存储“没有值”的字段，可以使用特殊值代替。如下面的 SQL 中，如果 consume 表中允许 customer_id 为空，则子查询中有可能存在 null，这样在 customer 表中，需要遍历整张表的 customer_id 值与 null 比对。

```
Select customer_phone from customer where customer_id in (select distinct customer_id from consume where expenditure > 10000);
```

### 2.2.2. 数据类型

#### 2.2.2.1. 整数

MySQL 支持的整数类型包括：tinyint、smallint、mediumint、int 和 bigint，它们分别需要 8、16、24、32 和 64 位存储空间。它们的范围如下表：

| 整数类型 | 范围 |
|---|---|
| tinyint | -128 到 127 |
| smallint | -32768 到 32767 |
| mediumint | -8388608 到 8388607 |
| int | -2147483648 到 2147483647 |
| bigint | -9223372036864775808 到 9223372036854775807 |

整数类型有可选的 unsigned 属性，它不允许负数。范围为最小值为 0，最大值为上表中对应范围的最小值取绝对值后加上最大值。例如，tinyint unsigned 保存的范围为 0 到 255。

有符号(signed)和无符号(unsigned)类型占用的存储空间是一样的，性能也一样。因此可以根据实际情况采用合适的类型。

MySQL 还可以对整数类型定义宽度，比如 int(11)。这对于大多数应用程序都是没有意义的：它不会限制值的范围，只规定了 MySQL 交互工具(如命令行客户端)用来显示字符的个数。对于存储和计算，int(1)和 int(20)是一样的。

同时每个整数类型都有同义词，如下表：

| 整数类型 | 同义词 |
|---|---|
| tinyint | int1 |
| smallint | int2 |
| mediumint | int3，middleint |
| int | int，int4 |
| bigint | int8 |

#### 2.2.2.2. 实数

MySQL 同时支持精确与非精确类型：

float 和 double 类型属于非精确类型(浮点类型)，用于保存非精确的小数，即数据库存储的数值和要保存的数值可能不一致。它们支持使用标准的浮点运算进行近似计算。

decimal 类型是精确类型，用于保存精确的小数。在 MySQL5.0.3 及以上版本，decimal 类型支持精确的数学计算。MySQL4.1 和早期版本对 decimal 值执行浮点计算，它会因为丢失精度而导致奇怪的结果(在这些版本中，decimal 仅仅是存储类型)。在 MySQL5.0.3 及以上版本中，MySQL 支持 decimal 运算，但 CPU 并不支持对它进行直接计算，因此效率相对浮点运算较低。

可以定义浮点类型和 decimal 类型的精度：

1. 对于浮点数类型，可以使用多种方式定义其精度，它会导致 MySQL 悄悄采用不同的数据类型，或者在保存的时候进行调整。比如指定一个浮点数类型的长度 float(1)或者 float(30)，这个长度仅能决定存储占用的字节数(4 字节 float 或者 8 字节 double)。如果长度在 0 到 23 之间，这是一个单精度浮点类型(float)，如果长度在 24 到 53 之间，这是一个双精度浮点类型(double)。对于这种定义方式，小数点前后的数字位数由 MySQL 自己决定。再比如对于 float(9，5)和 double(9，5)，9 规定的是显示宽度，5 规定的是小数点后的数字位数。对于浮点数类型显示宽度必须在 1 到 255 之间，小数点后位数必须在 0 到 30 之间。
2. 对于 decimal 列，可以定义小数点之前和之后的最大位数，这影响了所需的存储空间。MySQL5.0.3 以前的版本把数字保存到了一个二进制字符串中(每 4 个字节保存 9 个数字)。例如，decimal(18,9)将会在小数点前后都保存 9 位数字，总共使用 9 个字节：小数点前 4 个字节，小数点后 4 个字节，小数点占一个字节。MySQL5.0.3 将 decimal 存储为二进制形式而不是字符串形式，存储空间上会出现变化，但 decimal(18,9)依然表示保存小数点前后各 9 位数字。MySQL5.0.5 以上版本 decimal 类型最大支持 65 位的数字(5.0.3 到 5.0.5 是 64；早期的版本是 254)。

### 2.2.3. 字符存储类型

#### 2.2.3.1. varchar 和 char 类型

MySQL 支持很多种字符串类型，它们之间有很多不同。其中两种主要的字符串类型是 varchar 和 char。这两种类型在磁盘或内存的存储方式依赖于存储引擎。本节主要针对 MyISAM 和 InnoDB 两种存储引擎介绍 varchar 和 char 类型的存储方式。

1. varchar

varchar 保存了可变长度的字符串，它能比固定长度类型占用更少的存储空间。因为它只占用了自己需要的空间，较短的值占用的空间就较小。例外情况是使用 row_format=fixed 创建的 MyISAM 表，它为每行使用固定长度的空间，可能会造成浪费。

在 MySQL5.0.3 之前的版本 varchar 以字符数控制存储的最大长度，如 varchar(m)，最大只能存放 255 个字符，占用空间的实际大小与字符集有关。从 MySQL5.0.3 开始，varchar 的最大存储限制改为字节数，最大单条记录不超过 65536 字节。在表定义时 varchar(m)中的 m 表示的是字符数，所以在定义时需要考虑实际存储的数据最大字符长度(最多包含的字符数)和字符集每个字符占用的字节数来决定 m 的最大值。

varchar 使用额外的 1 或 2 个字节来存储值的长度。如果列的最大长度小于或等于 255，则使用 1 字节，否则就使用 2 字节。假设使用 latin1 字符集(一个字符对应存储占一个字节)，varchar(10)将占用 11 字节的存储空间。varchar(1000)则占用 1002 字节。

varchar 能节约空间，所以对性能有帮助。然而，由于行的长度是可变的，它们在更新的时候可能会发生变化，这会引起额外的工作。如果行的长度增加并不再适合于原始的位置时，具体的行为则会和存储引擎有关。例如，MyISAM 会把行拆开，InnoDB 则可能分页。

当最大长度大于平均长度，并且很少发生更新的时候，通常适合使用 varchar。还有当你使用复杂的字符集，比如 utf-8 时，它的每个字符都可能会占用不同的存储空间。

同时对于 varchar(5)和 varchar(200)保存“hello”占用的空间是一样的，但是较短的列会有很大的优势。因为较大的列会使用更多的内存，因为 MySQL 通常会分配固定大小的内存块来保存值。这对排序或使用基于内存的临时表尤其不好。同样的事情也会发生在使用文件排序或基于磁盘的临时表的时候。最好的策略就是只分配真正需要的空间。

2. char

char 是固定长度的。MySQL 总是为特定数量的字符分配足够的空间。当保存 char 值的时候，MySQL 会去掉任何末尾的空格。(在 MySQL4.1 及之前版本，varchar 也是如此。)进行比较的时候，空格会被填充到字符串末尾。如果所要存储的数据末尾确实需要空格则不能使用 char 类型存放。

在 MySQL5.0.3 之前的版本，如果定义 char(m)时，m 表示定义 m 个字符，如果 m 超过 255 时，MySQL 会自动将其转换成可以存放对应数据量的 text 类型。从 MySQL5.0.3 开始，所有超过 255 的定义 MySQL 会直接报错。而 char(m)最终在磁盘和内存中的存储字节数是基于字符集的。如 latin1 最大存储长度是 255 字节，而 gbk 最大存储长度是 510 字节。

char 在存储很短的字符串或长度近似相同的字符串的时候很有用。例如，char 适合用存储用户密码的 md5 哈希值，它的长度总是一样。对于经常改变的值，char 也好于 varchar，因为固定长度的行不容易产生碎片。对于很短的列，char 的效率也高于 varchar。char(1)字符串对于单字节字符集只会占用 1 个字节，但是 varchar(1)则会占用 2 个字节。

数据如何保存取决于存储引擎，并非所有的存储引擎都会按照相同的方式来处理定长和可变长度的字符串。Memory 存储引擎使用了固定长度的行，因此当它面对可变长度字段的时候就会分配可能的最大空间。而 Falcon 引擎即使是对固定长度的 char 字段，也会使用长度可变的列。但是填充和截取空格的行为在各个存储引擎之间都是一样的，因为这是 MySQL SERVER 程序的行为。

3. binary 和 varbinary

char 和 varchar 的兄弟类型为 binary 和 varbinary，它们用于保存二进制字符串。二进制字符串和传统的字符串很类似，但是它们保存的是字节，而不是字符。填充也有不同，MySQL 使用\0 填充 binary 值，而不是空格，并且不会再获取数据时把填充的值截掉。它们在必须要存储二进制数据并且想让 MySQL 按照字节进行比较的时候是有用的。字节比较的优势并不仅仅体现在大小写敏感上。MySQL 按照字节的数值进行比较，比按照字符比较简单的多，效率也更高效。

#### 2.2.3.2. blob 和 text 的类型

blob 和 text 分别以二进制和字符形式保存大量数据。在大多数方面，可以将 blob 类型视为能足够大的 varbinary 类型；同样，可以将 text 类型视为足够大的 varchar 类型。blob 和 text 在以下几个方面不同于 varbinary 和 varchar：当保存或检索 blob 和 text 列的值时不删除尾部空格；对于 blob 和 text 列上的索引，必须指定前缀长度；blob 和 text 列不能有默认值。

blob 和 text 各自有自己的数据类型家族：字符类型有 tinytext、smalltext、text、mediumtext 和 longtext，二进制类型有 tinyblob、smallblob、blob、mediumblob 和 longblob。blob 等用于 smallblob，text 等同于 smalltext。

和其他类型不同，MySQL 把 blob 和 text 当成有实体的对象来处理。存储引擎通常会特别地保存它们。InnoDB 在它们较大的时候会使用单独的“外部”存储区域来进行保存。

blob 和 text 唯一的区别就是 blob 保存的是二进制数据，没有字符集和排序规则，但是 text 有字符集和排序规则。MySQL 对 text 列的排序方式和其他类型不同：它不会按照字符串的完整长度进行排序，而只是按照 max_sort_length 规定的前若干个字节进行排序。如果只按照开始的几个字符排序，就可以减少 max_sort_length 的值或使用 order by substring(column，length)。

MySQL 不能索引这些数据类型的完整长度，也不能为排序使用索引。由于 Memory 存储引擎不支持 blob 和 text 类型，使用了 blob 和 text 列并且需要隐式临时表的查询将不得不使用磁盘上的 MyISAM 临时表，即使只有几列也是这样。这会导致严重的性能开销。即使把 MySQL 配置为使用 RAM 磁盘上的临时表，也需要很多昂贵的操作系统调用。最好的方法是尽可能地避免使用 blob 和 text 类型。如果不能避免，就可以使用 substring(column，length)把这些值转换为字符串，让它们使用内存中的临时表。

### 2.2.4. 日期时间类型

#### 2.2.4.1. 日期类型

datetime 这个类型能保存大范围的值，从 1001 年到 9999 年，精度为秒。它把日期和时间封装到一个格式为 yyyymmddhhmmss 的整数中，与时区无关。它使用了 8 字节存储空间。在默认情况下，MySQL 以一种可排序的、清楚的格式显示 datetime 值。

#### 2.2.4.2. 时间类型

timestamp 类型保持了自 1970 年 1 月 1 日午夜以来的秒数，它和 unix 时间戳相同。timestamp 只使用了 4 字节的存储空间，因此它的范围比 datetime 小得多。它只能表示从 1970 年到 2038 年。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。

较新的 MySQL 版本按照 datetime 格式化 timestamp 的值，但是较老的 MySQL 不会在各个部分之间显示任何标点符号。这仅仅是显示上的区别，timeatamp 存储格式在所有版本中都是一样的。

timestamp 显示的值依赖于时区。MySQL 服务器，操作系统及客户端连接都有时区设置。因此保存 0 值的 timestamp 实现显示为美国东部时间 1969-12-31 19:00:00，与格林尼治标准时间(GMT)相差 5 小时。

timestamp 也有 datetime 没有的特殊性质。在默认情况下，如果插入的行没有定义 timestamp 列的值，MySQL 会把它设置为当前时间。在更新的时候，如果没有显示地定义 timestamp 列的值，MySQL 也会自动更新它。可以配置 timestamp 列的插入和更新行为。最后，timestamp 列默认是 not null，这和其他数据类型都不一样。

## 2.2.5. 其它常用类型

### 2.2.5.1. bit 类型

MySQL 有几种数据类型可以利用值里面的二进制位来紧凑地保存数据。无论是 bit 类型还是后面的 set 类型从技术上来说都是字符串类型。

在 MySQL5.0 之前，bit 仅仅是 tinyint 的同名词。但是在 MySQL5.0 及以上版本中，它成了完全不同的，有特殊性质的新类型。可以使用 bit 列。并且在一列中保存一个或多个 true/false 值。bit(n)定义了含有 n 位二进制位的字段。bit 列的最大长度是 64 位。

bit 列的行为在不同的存储引擎之间是不同的。MyISAM 出于存储的目的，会把列包起来，因此 17 个单独的 bit 列只需要 17 位存储空间(假设所有列都不允许 null)。MyISAM 会把存储空间调整为 3 个字节。

MySQL 把 bit 当成字符串类型，而不是数字类型。当获取 bit(1)值的时候，结果是一个字符串，但内容是二进制的 0 或者 1，而不是 ASCII 值“0”或者“1”。但是，如果以数字的方式把它取出来，结果就是该二进制字符串的数字。例如把值 b‘00111001’保存到 bit(8)列中，并且以字符的方式取出来，这时就会得到包含了字符代码为 57 的字符串，它正好等于 ASCII 字符“9”。但如果用数字的方式取出来，就会得到 57。这个特点使得 bit 类型比较迷惑人。对于大多数程序，最好避免使用该类型。如果只想保存一位的 true/false，另外一个选择是创建一个可空的 char(0)列。它能保存空值(null)或 0 长度的值(空字符串)。

### 2.2.5.2. set 类型

如果要保存许多 true/false 值，可以考虑把许多列合并为 set 数据类型，它在 MySQL 内部是以一系列位表示的。set 是一个字符串对象，可以有零或多个值，其值来自表创建时规定的允许的一列值。指定包括多个 set 成员的 set 列值时各成员之间用逗号隔开。这样 set 成员本身不能含逗号。如 set(‘one’, ‘two’)not null 定义的列可以包含的值：‘’‘one’ ‘two’ ‘one,two’。set 最多可以有 64 个不同的成员。

set 有效地使用了存储空间，并且 MySQL 有诸如 find_in_set()和 field()这样的函数，方便在查询中使用它。它的主要缺点是改变列定义的代价比较高，需要 alter table，这在大表上是很昂贵的操作。通常来说，也不能在 set 列上使用索引。

set 类型的一种代替方式是利用整数包装一系列位。例如可以把 8 位包装到 tinyint 中，并且通过位运算符来操纵他们。可以在应用程序中为每个位定义命名常量来实现替代 set 类型。这样的好处在于不用 alter table 就可以改变字段代表的“枚举”意义；缺点是需要应用程序保存对应位对应值的意义，同时比较查询比较难写。

### 2.2.5.3. enum 类型

enum 是一个字符串对象，其值来自表创建时再列规定中显示枚举的一列值。在某些情况下，enum 值也可以为空字符串( ‘’ )或 null。如果将一个非法值插入 enum(也就是说，允许的值列之外的字符串)，MySQL 将插入空字符串以作为特殊错误值。该字符串与“普通”空字符串不同，该字符串有数值值 0。如果将 enum 列声明为允许 null，null 值则为该列的一个有效值，并且该列的默认值为 null。否则 enum 列被声明为 not null，其默认值为允许的值列的第 1 个元素。

每个枚举值有一个索引：来自列规定的允许值列中的值从 1 开始编号；空字符串错误值的索引为 0；null 的索引为 null。这说明可以使用索引 0 来查找分配了非法 enum 值的行。

枚举最多可以有 65535 个元素。enum 成员值的尾部的空格会被自动删除。

# 第 3 章. 索引

## 3.1. 索引介绍

### 3.1.1. 索引简介

为了满足对数据的快速访问，我们通常需要将数据组织成一种有序的方式， 而原始的情况下数据的物理存储顺序便可代表一种“序”，但是由于物理存储的“序”只能是一种，但我们业务的访问模式是多样的，所以我们有了索引,索引是一种以更小代价来组织数据关系的一种“序”，不同的索引可以满足不同的访问模式。

索引是使用 MySQL 过程中非常重要的一环，良好的索引将大大提高 SQL 的执行效率，提升单机性能，同时索引同锁、排序等都有着密切的关系。

索引类型的选择主要取决于应用的不同需求：

1. hash index 主要用于满足精确匹配；
2. B-Tree index 主要用于满足范围查询、精确匹配；
3. fulltext index 主要用于全文关键字查询；

不同的存储引擎支持不同的索引类型：

1. heap 引擎支持 hash index；
2. MyISAM 、InnoDB 引擎支持 B-Tree index；
3. MyISAM 支持 fulltext index；

索引具体由存储引擎提供支持，而非 MySQL 内核，所以使用的是对同一种索引类型，内部的实现方式与效率都可能不同。

### 3.1.2. B-Tree

当我们谈及索引而没说明其类型的时候，多半是指 B-Tree 索引，它通常使用 B-Tree 数据结构来保存数据。大部分 MySQL 的存储引擎都支持 B-Tree 索引(Archive 例外)。

B-Tree 索引加速了数据访问。从 B-Tree 根开始，借助中间节点页的上界和下界值，可以快速搜寻到叶子页层，最终找到含有需要找的值的叶子页(或者确定无法找到需要的数据)，找到对应的叶子页后可以通过相应的指针直接找到数据表中对应的数据行。这样存储引擎不会扫描整个表得到需要的数据。同时 B-Tree 索引通常意味着索引中数据保存时有序的，可以利用 B-Tree 索引来加速排序。

#### 3.1.2.1. InnoDB B-Tree

存储引擎使用了不同的方式把 B-Tree 索引保存到磁盘上，它们会表现出不同的性能。例如 MyISAM 使用前缀压缩的方式以减小索引；而 InnoDB 不会压缩索引。同时 MyISAM 的 B-Tree 索引按照行存储的物理位置来引用被索引的行，但是 InnoDB 按照主键值引用行。这些不同有各自的优点和缺点。

##### 3.1.2.1.1. InnoDB 聚簇索引(cluster index)

聚簇索引不是一种单独的索引类型，而是一种存储数据的方式。当表有聚簇索引的时候，它的数据行实际保存在索引的叶子页。聚簇是指实际的数据行和相关的键值都保存在一起。每个表只能有一个聚簇索引。

由于是存储引擎负责实现索引，并不是所有的存储引擎都支持聚簇索引。当前只有 SolidDB 和 InnoDB 是唯一支持聚簇索引的存储引擎。

数据与索引在同一个 B-Tree 上，一般数据的存储顺序与索引的顺序一致。InnoDB cluster index 每个叶子节点包含 primary key 和行数据,非叶子节点只包括被索引列的索引信息。

聚簇索引的优缺点：

优点：

1. 相关的数据保存在一起，利于磁盘存取；
2. 数据访问快，因为聚簇索引把索引和数据一起存放；
3. 覆盖索引可以使用叶子节点的 primary key 的值使查询更快；

缺点：

1. 如果访问模式与存储顺序无关，则聚簇索引没有太大的用处；
2. 按主键顺序插入和读取最快，但是如果按主键随机插入(特别是字符串)则读写效率降低；
3. 更新聚簇索引的代价较大，因为它强制 InnoDB 把每个更新的行移到新的位置；
4. 建立在聚簇索引上的表在插入新行，或者在行的主键被更新，该行必须被移动的时候会进行分页。分页发生在行的键值要求行必须被放到一个已经放满了数据的页的时候，此时存储引擎必须分页才能容纳该行，分页会导致表占用更多的磁盘空间。
5. 聚簇表可能会比全表扫描慢，尤其在表存储的比较稀疏或因为分页而没有顺序存储的时候。
6. 非聚簇索引可能会比预想的大，因为它们的叶子节点包含了被引用行的主键列。
7. 非聚簇索引访问需要两次索引查找，而不是一次。

其它需要说明点：

InnoDB 的 primary key 为 cluster index，除此之外，不能通过其他方式指定 cluster index，如果 InnoDB 不指定 primary key,InnoDB 会找一个 unique not null 的 field 做 cluster index，如果还没有这样的字段，则 InnoDB 会建一个非可见的系 统默认的主键---row_id(6 个字节长)作为 cluster_index 。建议使用数字型 auto_increment 的字段作为 cluster index 。不推荐用字符串字段做 cluster index (primary key) ，因为字符串往往都较长， 会导致 secondary index 过大(secondary index 的叶子节点存储了 primary key 的值),而且字符串往往是乱序。 cluster index 乱序插入容易造成插入和查询的效率低下。

### 3.1.2.1.2. InnoDB 辅助索引(secondary index)

InnoDB 中非 cluster index 的所有索引都是 secondary index。

secondary index 的查询代价变大，需要两次 B-Tree 查询，一次 secondary index，一次 cluster index。所以在建立 cluster index 和 secondary index 的时候需要考虑到这点。

当 secondary index 满足 covering index(参见 3.1.2.1.4 章节介绍)时，只需要一次 B-Tree 查询并且直接在 secondary index 便可获取所需数据，不需要再进行数据读取，提高了效率。我们在设计索引和写 SQL 语句的时候就可以考虑利用到 covering index 的优势。

建议尽量减少对 primary key 的更新，因为 secondary index 叶子节点包含 primary key 的 value(这样避免当 row 被移动或 page split 时更新 secondary index)，primary key 的变化会导致所有 secondary index 的更新。

### 3.1.2.1.3. InnoDB 动态哈希(adaptive hash index)

动态哈希索引是 InnoDB 为了加速 B-Tree 上的节点查找而保存的 hash 表 。 B-Tree 上经常被访问的节点将会被放在动态哈希索引中。

注意点：

MySQL 重启后的速度肯定会比重启前慢，因为 InnoDB 的 innodb_buff_pool 和 adaptive hash index 都是内存型的，重启后消失，需要预热(访问一段时间) 后性能才能慢慢上来。

#### 3.1.2.1.4. InnoDB 覆盖索引(covering index)

索引通常是用于找到行的，但也可以用于找到某个字段的值而不需要读取整个行，因为索引中存储了被索引字段的值,只读索引不读数据，这种情况下的索引就叫做覆盖索引。

覆盖索引是很有力的工具，可以极大地提高性能。它主要的优势如下：

1. 索引记录通常远小于全行大小，因此只读索引，MySQL 就能极大的减少数据访问量。这对缓存的负载是非常重要的，它大部分的响应时间都花在拷贝数据上。对于 I/O 密集型的负载也有帮助。因为索引比数据小很多，能很好的装入内存。
2. 索引是按照索引值来进行排序的，因此 I/O 密集型范围访问将会比随机地从磁盘提取每行数据要快的多。
3. 覆盖索引对于 InnoDB 来说非常有用，因为 InnoDB 的聚集缓存。InnoDB 的辅助索引在叶子节点保存了主键值，因此，覆盖了查询的第二索引在主键上避免了另外一次索引查找。

### 3.1.2.2. MyISAM B-Tree

MyISAM B-Tree 索引和 InnoDB B-Tree 索引都是采用 B-Tree 的存储方式，但是在数据布局上是不同的。

与 InnoDB 不同，MyISAM 的主键和其它索引没有结构上的区别。主键只是一个唯一的，名为 primary key 的非空索引。

MyISAM B-Tree 支持前缀压缩，压缩后的索引称为 MyISAM packed index。索引可以压缩为原来的 1/10，其实是 CPU/mem/disk 之间的一个 tradeoff，create table 的时候可以为索引指定 pack_keys (alter table t engine=MyISAM pack_keys=1)，但是 packed index 对反向 order 和 binary-search 效率差。

## 3.1.3. hash

### 3.1.3.1. hash 索引介绍

哈希索引建立在哈希表的基础上，它只对使用了索引的每一列的精确查找有用。对于每一行，存储引擎计算出了被索引列的哈希码，它是一个较小的值(可能对具有相同索引值的不同行计算出的哈希值不同)。索引中包含哈希码和对应指向数据行的指针。

在 MySQL 中,只有 Memory 引擎支持显式的哈希索引。尽管 Memory 引擎也支持 B-Tree 索引,但它是 Memory 表的默认索引类型。

### 3.1.3.2. hash 索引的合理使用

除了 Memory 引擎显式支持 hash 索引，NDB cluster 存储引擎支持唯一的 hash 索引。它的功能是该存储引擎特有的。

同时 InnoDB 存储引擎也有一个特别的功能较自适应 hash 索引。当 InnoDB 注意到一些索引值被很频繁地访问的时候，它就会在 B-Tree 的顶端为这些值建立起内存中的索引。这使 B-Tree 索引有了一些 hash 索引的特性。

同时如果存储引擎不支持 hash 索引，可以按照 InnoDB 使用的方式模拟自己的 hash 索引。

#### 3.1.3.3. hash 索引与 B-Tree 索引的优劣势

B-Tree 索引能很好地用于全键值、键值范围或者键前缀查找。主要对以下类型的查询有用：

1. 匹配全名，全键值匹配指和索引中的所有列匹配。
2. 匹配最左前缀，B-Tree 索引只能匹配索引的最左部分。
3. 匹配前缀索引，可以匹配某列值的开头部分。
4. 匹配范围值。
5. 只访问索引的查询，覆盖索引讨论的就是这种应用。

同时当 B-Tree 能以某种特殊的方式找到某行，那么它能以同样的方式对行进行排序。因此如上的查找方式也可以同等地应用于 order by。但是 B-Tree 也有一些局限：

1. 如果查找没有从索引列的最左开始，它就没有什么用，即最左原则。
2. 不能跳过索引中的列。
3. 存储引擎不能优化访问任何在第一个范围条件右边的列。

而 hash 索引本身只保存简单的 hash 值，hash 索引显得非常紧凑。hash 值的长度不会依赖于索引的列。查找速度是很快的，但是 hash 索引有一些局限：

1. 索引只包含了 hash 值和行指针，而不是值本身，MySQL 不能使用索引中的值来避免读取行。
2. MySQL 不能使用 hash 索引进行排序，因为不是按序包含行。
3. hash 索引不支持部分键匹配，因为它是由被索引的全部值计算出来的。
4. hash 索引只适用于=、in 相等的比较。不能加快范围查询。
5. 访问 hash 索引中的数据非常快，除非是碰撞率很高。当发生碰撞时，存储引擎必须访问链表中的每一个行指针，然后逐行进行数据比较，以确定正确的数据。

### 3.1.4. 前缀索引

#### 3.1.4.1. 前缀索引介绍

在 MySQL 中，索引只能从字段内容的最左端开始建， 查询的时候也只能从索引的最左端开始查， 对字段内容只建从左开始的部分字节的索引，而非全部做索引的这种 index 就叫做前缀索引(prefix index)。

前缀索引的优缺点：

优点

在索引满足一定的区分度的情况下，索引变得更小，更有利于放入或将更多的索引放入内存，减少 I/O 操作，提高效率。

缺点

前缀索引不支持 covering index 和 order by 。举例说明下：假如表 account 上有如下索引(balance,customer_email(50),account_number);其中字段 customer_email 的定义为 varchar(100)，那如下的两个 SQL 并不能完全使用该索引。

```
select account_number
from account
where balance=100.1 and customer_email='1@1.com';
select account_id
from account
where balance=100.1 and customer_email='1@1.com' order by account_number;
```

### 3.1.4.2. 前缀索引适合的字段类型

前缀索引一般可以提供高性能所需的选择。如果索引 blob 和 text 列，或者很长的 varchar 列，就必须定义前缀索引，这样既能节约空间同时能得到好的性能。

int 型的不建议使用 prefix index，虽然可以提升效率，但是却不能使用 order by，covering index 等， 建议使用更小的数字类型如 tinyint,bit 等来满足。

### 3.1.4.3. 前缀索引的合理长度选择

前缀索引涉及索引到底建多长的选择。短的索引可以节约空间。但是前缀又应该足够长，使他的选择性能够接近索引整个列，因此前缀的基数性应该接近于全列的基数性。

设计索引的时候结合记录数、字符集大小、字段长度、字段内容的重复程度、字符之间的相关性等考虑索引长度，索引长度不当将使索引过于庞大， 内存资源利用不高， 造成 IO 较重，程序效率降低。合理的索引长度，可以在满足较好索引区分度的情况下减少索引所占空间，我们的目标就是找到索引空间大小与索引区分度的一个平衡点。

选择索引长度的方法：

1. 首先了解表中记录的总体情况，如果表中数据还不存在或者很少，应该通过了解业务去构造和模拟符合业务和产品特点的数据，使用这些数据来模拟上线后的真实数据。
2. show table status\G；能看到 avg_row_length（每行的平均长度，不准确）、rows(不准确)、data 所占空间、已有索引所占空间等信息。
3. select count(*) from table；查看准确的总体行数。
4. 查看欲建立索引的字段的总体情况
5. 通过 select * from t procedure analyse()\G; 能看到表中所有字段的 min_value、max_value、min_length、max_length、是否为 null、字段平均长度、字段类型优化建议等信息。其中字段长度的相关信息很重要，它给出了字段的大致信息，对索引长度的选择很有帮助， 而字段类型优化则是在已有内容基础上给出的类型优化， 例如：如果你的表中有 1000 万行， 字段 name 为字符串， 但是却只有”a”,”b”,”c”三个值，则会建议优化字段类型为 enum(“a”,”b”,”c”)，这样查询和索引效率都会大大提高。
6. 查看欲建立字段的最佳索引区分度，select count(distinct city)/count(*) from city_demo；是该字段全部内容长度都做索引能达到的最理想的区分度，这个首先可以用来衡量该字段是否适合做索引。
7. 看不同索引长度的区分度，这个是个平均值例如：

```
Select count(distinct left(city, 3))/count(*) as sel3,
Count(distinct left(city, 4))/count(*) as sel4,
Count(distinct left(city, 5))/count(*) as sel5,
Count(distinct left(city, 6))/count(*) as sel6,
Count(distinct left(city, 7))/count(*) as sel7
From city_demo;
```

8. 查看到 city 字段做 3 个字节索引、4 个字节索引、5 个字节索引、6 个字节索引、7 个字节索引的区分度，可以一直增加索引长度来探测结果。
9. 如果随着索引长度的增加， 索引区分度在很明显地增大， 那说明我们应该继续增加索引长度，使当我们增加索引长度时，索引区分度没有明显变化，我们仍然应该继续增加索引长度探测。
10. 那么探测到何时为止呢？当我们发现继续增加很多索引长度但是区分度却没有明显提升而现有区分度接近第 3 条中的最佳区分度时，这个时候的索引长度可能就比较合理了。

11. 截止上面的步骤， 我们找的都是平均分布，有可能出现的是平均区分度很好而少量数据集中出现区分度极差的情况，所以我们还需要查看一下区分度分布是否均匀。
12. 查看区分度是否均匀：

```
        select count(*) as cnt,city
        from city_demo
        group by city
        order by cnt desc limit 100;

  select count(*) as cnt,left(city,3) as pref
        from city_demo
        group by pref
        order by cnt desc limit 100;

        select count(*) as cnt,left(city,4) as pref
        from city_demo
        group by pref
        order by cnt desc limit 100;

        select count(*) as cnt,left(city,5) as pref
        from city_demo
        group by pref
        order by cnt desc limit 100;
```

13. 索引选择的最终长度应该在平均区分度(前 4 条)与区分度是否均匀(第 5 条)之间长度做一个综合的选择。
14. 建完索引后 show table status 查看索引大小。这是一个收尾且非常重要的工作， 我们必须清楚的知道建立这个索引的代价。

## 3.1.5. 全文检索

### 3.1.5.1. 全文检索介绍

大部分的查询都可能有 where 语句，用于比较相等性，过滤数据等。但是有时也需要执行关键字搜索，它基于数据的关联性，而不是相互比较。全文检索就是为这个目的设计的。

MySQL 中只有 MyISAM 存储引擎支持全文索引。可以在上面搜索基于字符的内容(char，varchar 和 text 列)，并且它支持自然语言搜索和布尔搜索。

MyISAM 全文索引操作了一个全文集合，它由单个表中的一个或者多个字符列组成。实际上，MySQL 在集合中通过联接列构造索引，并且把他们当成很长的字符号串进行索引。

MyISAM 全文索引是一种特殊的具有两层结构的 B 树。第一层保存了关键字，然后对每个关键字，第 2 层包含了一个列表，它由相关的文档指针组成，这些指针指向包含该关键字的全文集合。索引不会包含集合中的每一个词。它按照下面的方式进行调整：

一个停用字清单把无意义的词剔除了，这样它们就不会被索引。停用字列表基于常用的英语语法，但是可以使用 ft_stopword_file 选项用一个外部列表替换掉它。

除非一个词长度大于 ft_min_word_len 并且小于 ft_max_word_len，否则它就会被忽略。

全文索引没有存储关键字发生的列信息，所以如果要对组合的列进行搜索，就要创建多个索引。

#### 3.1.5.2. 布尔全文搜索

在布尔搜索中，查询自身定义了匹配单词的相对相关性。布尔搜索使用了停用词表(stopword list)来过滤无用的单词，但是要禁用单词的长度必须大于 ft_min_word_len 且小于 ft_max_word_len 这一选项。

布尔搜索的结果是没有排序的。但是可以使用前缀来修改搜索字符串每个关键词的相对排名。常用的修饰符见下表：

| 示例 | 含义 |
|---|---|
| dinosaur | 含有“ dinosaur”的行排名较高 |
| ~ dinosaur | 含有“dinsaur”的行排名较低 |
| + dinosaur | 行必须含有“dinosaur” |
| - dinosaur | 行不能含有“dinsaur” |
| dino* | 含有以“dino”打头的单词的行排名较高 |

布尔全文搜索实际不需要全文索引。如果有全文索引的话，他就会使用索引，如果没有的话，它就会扫描整个表。甚至可以对多个表使用布尔全文搜索。

### 3.1.6. 空间(R-Tree)索引

R-Tree 索引可能是在其他数据库中很少见的一种索引类型，主要用来解决空间数据检索的问题。

#### 3.1.6.1. R-Tree 索引介绍

在 MySQL 中，支持一种用来存放空间信息的数据类型 geometry，且基于 OpenGIS 规范。在 MySQL 5.0.16 之前的版本中，仅 MyISAM 存储引擎支持该数据类型，但是从 MySQL 5.0.16 版本开始，BDB、InnoDB、NDBCluster 和 Archive 存储引擎也开始支持该数据类型。当然，虽然多种存储引擎都开始支持 geometry 数据类型，但是仅仅之后的 MyISAM 存储引擎支持 R-Tree 索引。

在 MySQL 中采用了具有二次分裂特性的 R-Tree 来索引空间数据信息，然后通过几何对象(MRB)信息来创建索引。

虽然只有 MyISAM 存储引擎支持空间索引(R-Tree Index)，但是如果是精确的等值匹配，创建在空间数据上面的 B-Tree 索引同样可以起到优化检索的效果，空间索引的主要优势在于使用范围查找的时候，可以利用 R-Tree 索引，而 B-Tree 索引就无能为力了。

## 3.2. 索引的合理设计和使用

### 3.2.1. 索引的字段及长度

#### 3.2.1.1. 主键和候选键上的索引

在 create table 语句中，我们可以指定主键和候选键。主键和候选键都有 unique 约束，这些列不会包含重复值。MySQL 自动为主键和每个候选键创建一个唯一索引，以便新值的唯一性可以很快检查，而不必扫描全表。同时加速对于这些列上确定值的查找。主键的索引名为 prmariy，候选键的名为该键包含的第一列的列名。如果存在多个候选键的名字以同一个列名开头，就在该列明后放置一个顺序号码区别。

### 3.2.1.2. 连接列上创建索引

一般会在表的连接列上建立索引，尤其是该表频繁参与连接操作。对于一个比较大的连接操作，如果被驱动表的连接列上没有索引的话，由于 MySQL 的连接算法是 nested loop 算法，会造成多次扫描被驱动表，对数据库造成的压力和开销是巨大的。

### 3.2.1.3. 在高选择度的列上创建索引

属性列上的选择度是指该列所包含的不重复的值和数据表中总行数(T)的比值，它的比值在 1 到 1/T 之间。选择度越大，越适合建索引。因为对于要查找这个列上的一个值的行，通过索引可以过滤掉大部分数据行，剩下的符合要求的行数较少，可以快速在数据表中定位这些行。相反，如果列的选择度比较小，通过索引过滤后的行数依然很大，和全表扫描的开销没有明显的改善，甚至会更大(全表扫描带来的是顺序 I/O，而通过索引过滤后的扫描可能是随机 I/O)。因此，在选择索引列时的首要条件就是候选列的选择度。索引要建立在那些选择度高的索引上，在选择度低的列上尽量避免建索引。

### 3.2.1.4. 创建联合索引的选择

在很多时候，where 子句中的过滤条件并不是只针对某一个字段，经常会有多个字段一起作为查询过滤条件存在于 where 子句中。在这时候，就需要判断是该仅仅为过滤性最好的(选择度最大)的列建立索引，还是在所有过滤条件中所有列上建立组合索引。对于这个问题，需要衡量两种方案各自的优劣。当 where 子句中的这些字段组成的联合索引过滤性远大于其中过滤性最高的单列，就适合建联合索引。这样就意味着 where 子句中对应的每个列过滤性都不高，但是这些单列的过滤性乘在一起后过滤性就高了。

**例如：要从存储着学籍信息的表中查找来自中国，大连的女性学生，使用的 SQL 的 where 子句如下：**
**where country = ‘china’ and city = ‘dalian’ and gender = female；**
**country 和 city 的联合（country，city）的选择性会比 country 和 city 各自的选择性高，同时因为 gender 本身的选择性低，将其加入对于提高总体选择性贡献不大，所以在此情境下适合建立（country，city）的联合索引。**

同时从性能角度讲，MySQL 使用联合索引比使用 index_merge 算法来使用各个单列索引的效率要高，性能要好。因此对于经常一起出现在 where 子句中的过滤条件组合，优先考虑建立这些条件列的联合索引，而不是为每个单列建立索引。

### 3.2.1.5. 通过索引列属性的前缀控制索引的长度

索引占用的空间越小，对于 MySQL 获得高性能越有益。不管是什么类型的索引，在查询中使用都是需要从磁盘中加载到内存中去的，无论是 MyISAM 对应的 key cache，还是 InnoDB 对应的 buffer pool。这些受到程序自身和硬件条件限制都是有大小限制的，如果索引大小比较大的话，会造成这些存放索引的内存区域无法存下整个索引数据，根据 LRU 算法频繁地淘汰索引，加载新的索引进去，这就造成比较大的 I/O 开销。

如果要建索引的列是很长的字符串的话，它会使索引变大。如果大小超过限制的话，可以考虑建前缀索引，即只索引数据列中存储的数据的前几个字符，而不是全部的值，这样可以有效地减小索引的大小。当然这样做的前提是保证索引的选择性。在选择列上要索引的字符长度时，考虑选择性不能只看平均值，还要考虑最坏情况下的选择性。因为使用前缀索引而索引的字符数不足的话，容易造成数据分布不均匀。如果这种情况比较极端，可能会造成索引的作用下降。

## 3.2.2. 如何在操作中利用索引

### 3.2.2.1. 索引与排序操作

MySQL 有两种产生排序结果的方式：使用文件排序(filesort)，或者使用扫描有序的索引。explain 的输出中 type 列的值为“index”，这说明 MySQL 会扫描索引。

MySQL 能为排序和查找行使用同样的索引。如果可能，按照这样一举两得的方式设计索引是个好主意。按照索引对结果进行排序，只有当 order by 子句中的顺序和索引最左前缀顺序完全一致，并且所有列排序的方向(升序或降序)一样才可以。order by 无需定义索引的最左前缀的一种情况是索引中其它前导列在 where 子句中为常量。如果查询联接了多个表，只有在 order by 子句的所有列引用的是第一个表才可以。

### 3.2.2.2. 索引与分组操作

group by 实际上也同样需要进行排序操作，而且与 order by 相比，group by 主要只是多了排序之后的分组操作。当然，如果在分组时还是用了其他一些聚合函数，还需要一些聚合函数的计算。所以在 group by 的实现过程中，与 order by 一样可以使用索引。同时使用索引带来的性能提升是巨大的。

在 MySQL 中 group by 使用索引的方式有两种：使用松散(loose)索引扫描；使用紧凑索引扫描。

松散索引扫描实现 group by 是指 MySQL 完全利用索引扫描来实现 group by 时，并不需要扫描所有满足条件的索引键即可完成操作，得出结果。如果 MySQL 使用了这种方式，则在 explain 的 extra 行会出现“using index for group-by”。要利用到松散索引扫描实现 group by，需要至少满足以下几个条件：

1. group by 条件列必须处在同一个索引的最左连续位置；
2. 在使用 group by 同时，只能使用 max 和 min 这两个聚合函数；
3. 如果引用到了该索引中 group by 条之外的列条件，它就必须以常量形式出现。

紧凑索引扫描实现 group by 是指 MySQL 需要在扫描索引时，读取所有满足条件的索引键值，然后再根据读取到的数据来完成 group by 操作，已得到相应的结果。这时的执行计划中就不会出现“using index for group-by”。使用这种索引扫描方式要满足的条件是：group by 条件列必须是索引中的列，同时索引中位于该条件列左边的列必须以常数的形式出现在 where 子句中。

除了上述两种使用索引扫描来完成 group by 外，还可以使用临时表加 filesort 实现。但是这种方式带来的性能开销比较大，一般也比较费时。所以 group by 最好实现方式是松散索引扫描，其次是紧凑索引扫描，最后是使用临时表和 filesort。

### 3.2.2.3. 索引与求 distinct 查询

distinct 实际上和 group by 操作非常相似，只是在 group by 之后的每组中只取其中一条记录而已。所以，distinct 的实现方式和 group by 也基本相同。同样通过松散索引扫描或者紧凑索引扫描的方式实现要优于使用临时表实现。但在使用临时表时，MySQL 仅是使用临时表缓存数据，而不需要进行排序，也就省了 filesort 操作。

### 3.2.2.4. 索引与带有 limit 子句的查询

含有 limit 子句的查询往往同时含有 order by 子句(如果没有 order by 子句则优化方法和普通查询一样)。这样的查询最好在排序时使用索引扫描进行排序。否则即使 limit 子句中只取排序后起始部分很少的数据都会引

起 MySQL 取出全部符合条件的数据进行排序。如果使用索引扫描的话，则不需要对所有数据排序，只需扫描索引取出满足 limit 限制的数据即可。

同时对于 limit 子句中的大偏移量的 offset，比如 limit 10000,20，它就会产生 10020 行数据，并且丢掉前 10000 行。这个操作的代价太大。一个提高效率的简单技巧是在覆盖索引上进行偏移，而不是对全行数据进行偏移。也可以将从覆盖索引上提取出来的数据和全表数据进行连接，然后取得需要的数据。

#### 3.2.2.5. 索引与连接操作

在 MySQL 中，只有一种连接 join 算法即 nested loop join。该算法就是通过驱动表的结果集作为循环的基础数据，然后将该结果集中的数据作为过滤条件一条条地到下一个表中查询数据，最后合并结果。所以在通过结果集中的数据作为过滤条件到下一个表中地位数据时，最好是通过索引，而不是扫表。因为如果结果集中的数据比较多，要是每次都通过扫描来定位的话，造成的开销和对 MySQL 的压力是巨大的。因此，最好在被驱动表的连接列上建立索引，并且使 MySQL 在连接过程中使用索引。

#### 3.2.2.6. 索引与隔离列

如果在查询中没有隔离索引的列，MySQL 通常不会使用索引。“隔离”列意味着它不是表达式的一部分，也没有位于函数中。

例如，下面的查询不能使用 actor_id 上的索引：

```
    select account_id from account where account_id+1=5;
```

人们能轻易地看出 where 子句中的 actor_id 等于 4，但是 MySQL 却不会帮你求解方程。应该主动去简化 where 子句，把被索引的列单独放在比较运算符的一边。

下面是另外一种常见的问题：

```
    select expenditure from consume where to_days(current_date)-to_days(consume_time)
<=10;
```

这个查询将会查找 date_col 值离今天不超过 10 的所有行，但是它不会使用索引，因为使用了 to_days() 函数。下面是一种比较好的方式：

```
    select expenditure from consume where consume_time>=
date_sub(current_date,interval 10 day) ;
```

这个查询就可以使用索引，但是还可以改进。使用 current_date 将会阻止查询缓存把结果缓存起来，可以使用常量替换掉 current_date 的值：

```
    select expenditure from consume where consume_time>=
date_sub('2010-12-12',interval 10 day);
```

### 3.2.3. 索引创建的建议

#### 3.2.3.1. 对联合索引中包含属性列的选择

对于 where 子句中过滤条件涉及的属性列大致相同的一系列 SQL 建立共同的索引。如果共同涉及的属性列是多个的话，则应建立联合索引。在确定联合索引应该包含这些共同涉及的属性列中的哪些时，应该考察这些 WHERE 子句对于涉及这些列上的过滤条件的形式。对于那些是范围条件对应的列，由于 B-Tree 索引本身的限制，只能选取其中一个选择度比较高的列进入联合索引。而对于那些等值条件对应的列，原则上都可以进入联合索引，但是需

要综合考虑联合索引最后的大小和进入索引的列的选择度。如果属性列的选择度非常低的话，把它放入索引对于联合索引的选择度贡献比较小，但是会增大索引大小，引起其它开销。所以不要把这样的列加入到索引中去。如 3.2.1.4 节中提到的例子，gender 列的选择性较低，加入联合索引对于提高联合索引的选择性没有太大帮助，但却增加了联合索引的大小。

### 3.2.3.2. 正确创建联合索引中各列的顺序

对于 MySQL 普遍使用的 B-Tree 索引，索引列的顺序对于 SQL 使用该索引至关重要。如果索引中列的顺序不合理，在使用过程中往往会使该索引无法被使用或者通过该索引得到的过滤能力大大减弱。

首先由于 B-Tree 索引的数据结构限制，只有当 SQL 的 where 子句使用索引的最左前缀的时候，索引才能被使用、发挥作用。所以在创建索引、决定索引的顺序时，应提取希望使用该索引 SQL 的 where 子句中的过滤条件，提炼出其中的最常出现的条件和其对应的属性列。按照这些列的选择度由高到低排列这些属性列，按照这个顺序创建这个索引。同时相关 SQL 的 where 子句中出现的过滤条件顺序，以尽量让这些 SQL 可以使用建立的索引的最左前缀。

对于联合索引中包含的属性列中，有一列对应在相关 SQL 的 where 子句的过滤条件是以范围条件出现，而索引中其他属性列是以等于条件出现，则应该把这些等值条件对应的列放在索引的前面，把范围条件对应的列放在索引的最后。

```
select account_id from consume where account_payee =72478814 and expenditure>1.00;
```

为上述 SQL 创建对应的联合索引时：如果创建索引(expenditure,account_payee)，由于 expenditure 列上是范围条件，所以索引(expenditure,account_payee)无法使用完全(只能使用索引中的 expenditure 部分)；如创建索引(account_payee, expenditure)，SQL 则可以完全使用此索引。所以针对上述 SQL 应该创建联合索引(account_payee, expenditure)。

### 3.2.3.3. 避免重复索引

MySQL 允许你在同一列上创建多个索引，它不会注意到你的错误，也不会为错误提供保护。MySQL 不得不单独维护每一个索引，并且查询优化器在优化查询的时候会逐个考虑它们，这会严重影响性能。重复索引是类型相同，以同样的顺序在同样的列上创建的索引。应该避免创建重复索引，并且在发现它时把它移除掉。

有时会在不经意间创建重复索引。例如下面的代码：

```
create table test(
    id int not null primary key,
    unique(id),
    index(id)
);
```

对于 id 列，首先它是 primary key，同时 unique(id)使 MySQL 自动为 id 创建了名为 id 的索引，最后 index(id),现在给 id 列创建了三个索引。这通常是不需要的,除非需要为同一列建立不同类型的索引,如 B-Tree,fulltext 等类型索引。

### 3.2.3.4. 避免多余索引

多余索引和重复索引不同。例如列(A，B)上有索引，那么另外一个索引(A)就是多余的。也就是说(A，B)上的索引能被当成索引(A)(这种多余只适合 B-Tree 索引)。多余索引通常发生在向表添加索引的时候，例如，有人也许会在(A,B)上添加索引，而不是对索引(A)进行扩展。

对于 B-Tree 类型索引，有单列索引对应的属性列出现在了某个联合索引的第 1 位置上，那么这个单列索引可能是多余的。

如果某一索引是主键的超集，那么这个索引除非有特殊理由(如希望使用覆盖索引)，否则也是多余索引。因为主键是唯一索引，过滤能力很强，和它建立联合索引意义不大。

在大部分情况下，多余索引都是不好的，为了避免它，应该扩展已有索引，而不是添加新的索引。但是，还有一些情况出于性能考虑需要多余索引。使用多余索引的主要原因是扩展已有索引的时候，它会变得很大。

#### 3.2.3.5. 使用覆盖索引

索引是找到行的高效方式，但是 MySQL 也能使用索引来接收数据，这样就可以不用读取行数据。包含所有满足查询需要的数据的索引叫覆盖索引。覆盖索引和任何一种索引都不一样。覆盖索引必须保存它包含的列的数据。MySQL 只能使用 B-Tree 索引来覆盖查询。

在 SQL 执行中要使用覆盖索引的话，需要相应的索引包含 SQL 中 where 子句中涉及的列都在索引中且满足最左前缀，同时 SQL 的返回列也必须在索引中。同是 where 子句中的过滤条件中不能包含 like 等操作符和函数。MySQL 使用了覆盖索引，会在 explain 输出中出现“using index”字样。

有关使用覆盖索引的案例请参见第 4 章 4.2.3.6 使用 covering index 优化 select 语句。

### 3.2.4. 索引的维护

#### 3.2.4.1. 数据的 optimize 和 analyze 操作

MySQL 查询优化器在决定如何使用索引的时候会调用两个 API，以了解索引如何分布。第一个调用接受范围结束点并且返回该范围内记录的数量；第二个调用返回不同类型的数据，包括数据基数性(每个键值有多少记录)。当存储引擎没有向优化器提供查询检查的行的精确数量的时候，优化器会使用索引统计来估计行的数量，统计可以通过运行 analyze table 重新生成。MySQL 的优化器基于开销，并且主要的开销指标是查询会访问多少数据。如果统计永远没有产生，或者过时了，优化器就会做出不好的决定。解决方案是运行 analyze table。

每个存储引擎实现索引统计的方式不同，由于运行 analyze table 的开销不同，所以运行它的频率也不一样。

1. Memory 存储引擎根本就不保存索引统计。
2. MyISAM 把索引统计保存在磁盘上，并且 analyze table 执行完整的索引扫描以计算基数性。整个表都会在这个过程中被锁住。
3. InnoDB 不会把统计信息保存到磁盘上，同时不会时时去统计更新它们，而是在第一次打开表的时候利用采样的方法进行估计。InnoDB 上的 analyze table 命令就使用了采样方法，因此 InnoDB 统计不够精确，除非让服务器运行很长的时间，否则不要手动更新它们。同样，analyze table 在 InnoDB 上不是阻塞性的，并且相对不那么昂贵，因此可以在不大影响服务器的情况下在线更新统计。

B-Tree 索引能变成碎片，它降低了性能。碎片化的索引可能会以很差或非顺序的方式保存在磁盘上。同是表的数据存储也能变得碎片化。碎片化对于数据的读取，尤其是范围数据的读取，会使读取速度慢很多。为了消除碎片，可以运行 optimize table 解决。

#### 3.2.4.2. 索引的修复——MyISAM

MyISAM 引擎在下列情况下可能会出现数据和索引不一致的情况，出现索引错误或者数据错误：

1. MySQL 进程在写中间被杀掉；
2. 发生未预期的计算机关闭；

3. 硬件故障；
4. 可能同时在正被 server 程序修改的表上使用外部程序(如 myisamcheck)；
5. MySQL 或 MyISAM 代码的缺陷；

一个损坏的表的典型症状如下：

```
incorrect key file for table: '…'.try to repair it.
```

遇到 MyISAM 出现数据和索引不一致的情况时，可以用 check table 语句来检查 MyISAM 表的健康，并用 repair table 修复。当 MySQL 不运行时，也可以使用 myisamcheck 命令检查或修复这个问题。

## 3.2.5. 索引的其他说明

### 3.2.5.1. 索引对插入、更新的影响和避免

索引是独立于基础数据之外的一部分数据。假设在 table t 中的 column c 创建了索引 idx_t_c，那么任何更新 column c 的操作包括插入 insert，update，MySQL 在更新表中 column c 的同时，都必须更新 column c 上的索引 idx_t_c 数据，调整因为更新带来键值变化的索引信息。而如果没有对 column c 建立索引，则仅仅是更新表中 column c 的信息就可以了。这样因调整索引带来的资源消耗是更新带来的 I/O 量和调整索引所致的计算量。

基于以上的分析，在更新非常频繁地字段不适合创建索引。很多时候是通过比较同一时间内被更新的次数和利用该列作为条件的查询次数来判断的，如果通过该列的查询并不多，可能几个小时或者更长时间才会执行一次，更新反而比查询更频繁，那么这样的字段肯定不适合创建索引。

# 第 4 章. SQL 编写规范及其优化

## 4.1. 数据定义语句语法

### 4.1.1. 有关 database 语法

使用 create database 语句，可以创建一个新的数据库。在这个过程中，可以指定默认的字符集和默认的字符比较方法。

```
    <create database statement> = create database [if not exists] <database name>
[<database option>…]
    <database option> = [default] character set <character set name> | [default] collate
<collate name>
```

同时可以使用一条 alter database 语句来修改已有的默认字符集和字符比较方法。这些新的默认设置只适用于在更新以后创建的表和列。

```
    <alter database statement> = alter database <database name> [<database option>…]
    <database option> = [default] character set <character set name> | [default] collate
<collate name>
```

drop database 语句会立刻删除整个数据库。该数据库的所有表也将永久删除，因此要特别小心使用该语句。

```
    <drop database statement> = drop database [if not exists] <database name>
```

### 4.1.2. 有关 table 语法

#### 4.1.2.1. create table 相关语法

可以使用 create table 语句构建一个新的表来存储数据行。这个语句包括列定义、表完整性约束、列完整性约束、数据类型和索引定义等。

```
    <create table statement> = create [temporary] table [if not exists] <table
specification> <table structure> [<table option>…]
    <table specification> = [<database name>.] <table name>
    <table structure> = <table schema>
    <table schema> = (<table element>[,<table element>]…)
    <table element> = <column definition> | <table integrity constraint > | <index
definition>
    < column definition> = <column name> <data type> [<null specification>][<column
integrity constraint>]
    <null specification> = [not] null
    <column integrity constraint> = primary key | unique [key] | <check integrity
constraint>
    <table integrity constraint> = <primary key> | <alternate key> | <foreign key> |
<check integrity constraint>
```

```
    <table option> = engine = <engine name> | type = <engine name> | union (<table name>
[,<table name>]) | insert_method = {no | first | last} | auto_increment = <whole number>
| max_rows = <whole number> | min_rows = <whole number> | [default] character set {<name>
| default} | [default] collate {<name> | default} | data directory = <directory> | index
directory = <directory> | check_sum = {0|1} | delay_key_write = {0|1} | pack_keys =
{0|1|default} | password = <> | raid_type = {1 | striped | raid0} | raid_chunks = <whole
number>| raid_chunksize = <whole number> | row_format = {default | dynamic| fixed |
comperssed}
```

#### 4.1.2.2. create table 中完整性约束

对数据库施加数据完整性约束是数据库服务器的最为重要的功能之一。数据完整性指的是数据的一致性和正确性。如果单个数据项没有彼此冲突，数据就是正确的；如果满足所有的相关规则，数据就是正确的。完整性约束是一个数据库的内容必须随时遵守的规则；它们描述了对数据库的哪一次更新是允许的。如果定义了完整性约束，MySQL 会负责数据完整性。每次更新后，MySQL 都会测试新的数据库内容是否符合相关的完整性约束。

MySQL 中的完整性约束主要包括：主键，候选键，外键，参照动作，check 完整性约束。

主键就是表中的一列或多个列的组合，它们的值总是唯一的。构成一个主键的一部分的列的值不允许为空。可以用两种方式来定义主键：作为列或这表的完整性约束。

候选键像一个主键一样，是一个表的一列或一组列，它们的值在任何时候都是唯一的。候选键是主键的候选键(没有被选作主键)。一个表可以有多个候选键，但是只能有一个主键。

外键定义于其中的表叫作参照表，外键所指向的表叫作被参照表。在定义这种参照性约束后，MySQL 会确保插入到外键中的每一个非空值都是已经在被参照表中作为主键出现。外键约束包括三部分：第一部分表示哪个列是外键，第二部分指定外键参照的表和列，第三部分参照动作。当指定一个外键的时候：

1. 被参照表必须已经用一条 create table 语句创建了，或者必须是当前正在创建的表。在后一种情况下，参照表和被参照表是同一表。
2. 必须为被参照表定义主键。
3. 必须在被参照表的表明后面指定列名(或者列名的组合)。这个列(或者列的组合)必须是这个表的主键。
4. 外键中的列的数目必须和被参照表的主键中的列的数目相同。
5. 外键中列的数据类型必须和被参照表的主键中列的数据类型对应相等。

## 4.2. select 语句语法及其优化

### 4.2.1. select 语法规范介绍

#### 4.2.1.1. select 子句的语法及相关聚合函数

select 子句作用是选取所需的列，它的结果形成了表的一个竖向子集合。select 子句中可以包含有相关的聚合函数及一定的计算表达式等。下面是 select 子句语法的定义：

```
    <select clause> = select <select option> … <select element list>
    <select option> = distinct | distinctrow | all | high_priority | sql_buffer_result
| sql_cahce | sql_no_cache | sql_clac_found_rows | sql_small_result | sql_big_result
| straight_join
    <select element list> = <select element> [,<select element> …]
```

```
<select element> = <column expression> | <table specification>.<column expression>
```

select 子句中可以指定一个星号(*)的子句。这个星号是返回 from 子句中提取到的每行所有列的一种简单表达式。同时 select 子句中可以包含一个或多个表达式，这些一个表达式可以是常量值，一个计算(该计算可以包含表中的列)或者一个聚合函数(函数的输入可以是表中的列)。select 子句可以使用 distinct 移除重复的行。当在 select 子句中的列名、表达式前面加上关键字 distinct 时，MySQL 会从中间结果中移除重复的行。在<select option>中还可以使用 sql_no_cache 等 hint 相关的关键字，详细介绍请见第 4 章 4.5.优化器相关 explain 以及常用 hint 介绍。

select 子句中的表达式可以包含聚合函数：

| 聚合函数名 | 解释 |
|---|---|
| count | 计数 |
| Min | 求最小 |
| Max | 求最大 |
| Sum | 求和 |
| Avg | 求平均 |
| stddev | 标准差 |
| variance | 方差 |
| bit_and | 位与 |
| bit_or | 位或 |
| bit_xor | 位异或 |

```
select account_payee,max(expenditure) from consume where account_id=1 group by account_payee;
```

select 子句中包含了列名(account_payee)和聚合函数 max。

### 4.2.1.2. from 子句的语法

from 子句是一个重要的子句。因为，在这个子句中指定了要在其他子句中“使用”那些表的列。“使用”这个词的意思是，一个列出现在一个条件中或者 select 子句中。简而言之，在 from 子句中，指定了表达式中的结果从哪个表中获取。

from 子句用来指定将要查询的表。这是通过表引用来完成的。一个表引用由一个表指定组成，表指定后面可能还跟着一个假名。

```
from clause = from <table reference> [,<table reference>]
<table reference> = <table specification> [[as] <pseudonym>]
<table specification> = [<database name>.]<table name>
<pseudonym> = <name>
```

表指定通常由一个表的名字组成，但是也可以是指定视图的名字。from 子句可以包含一个或多个表指定。同时每张表都存储在相应的数据库中，可以使用两种方式引用一个表。第一种方式是使用 use 语句，第二种方式是显示地扩展表指定，带上表所属的数据库的名字。

### 4.2.1.3. where 子句的语法

where 子句充当一种过滤器，它移除了所有条件不为 true(为 false 或 unknow)的行。where 子句的定义：

```
<where clause> = where <condition>
```

```
    <condition> = <predicate> | <predicate> or <predicate> | <predicate> and
<predicate> | not <condition>
    <predicate> = <predicate with comparison> | <predicate without comparison> |
<predicate with in> | <predicate with between> | <predicate with like> | <predicate
with regexp> | <predicate with match> | <predicate with null> | <predicate with exists>
| <predicate with any all>
```

上述 where 子句中包含的条件有：比较运算符；使用 and、or 和 not 的条件；带有子查询的比较运算符；带有表达式列表的 in 运算符；带有子查询的 in 运算符；between 运算符；like 运算符；regexp 运算符；match 运算符；null 运算符；any 和 all 运算符；exists 运算符。

### 4.2.1.4. group by，having 子句的语法

group by 子句根据行的相似性对它们分组。通过把聚合函数(例如 sum 和 count)添加到带有一个 group by 子句的一个选择语句块中，数据就可以实现聚合。聚合意味着我们是求一个加和、平均、频次以及子和，而不是单个的值。

```
    <group by clause> = group by <group by specification list> [with rollup]
    <group by specification list> = <group by specification> [,<group by
specification>]…
    <group by specification> = <group by specification> [<sort direction>]
    <group by specification> = <column expression>
    <sort direction> = asc | desc
```

一个选择语句块的 having 子句的目的和 where 子句类似。不同之处在于，where 子句用来在 from 子句处理之后选择一行，而 having 子句用来在 group by 子句执行以后选择一组。一个 having 子句可以单独使用，而不使用 group by 子句。

```
    <having clause> = having <condition>
```

group by 子句对 from 子句的结果中行分组。having 子句能够根据它们特定的组属性来选择(带有行的)组。having 子句中的条件看上去很像是 where 子句中的一个正常的条件。尽管如此，还是存在一个区别：having 子句中的条件中的表达式可以包含一个聚合函数，而 where 子句的条件的表达式则不可以。having 子句中的所有列指定必须出现在一个聚合函数中，或者出现在 group by 子句指定的列的列表中。因为一个聚合函数总是每组由一个值组成。另外，用来对结果进行分组的列指定的结果也总是每组只有一值。下面的 SQL 是不正确的：

```
    select max(expenditure) from consume where account_id=1 group by account_payee
having year(consume_time)>2008;
```

consume_time 没有出现在聚合函数中，也没有出现在 group by 的列表中。可以修改为：

```
    select max(expenditure) from consume where account_id=1 group by account_payee
having min(year(consume_time))>2008;
```

### 4.2.1.5. order by 子句的语法

在一条 select 语句的最末尾添加一个 order by 子句，可以保证最终结果中的行按照一定的顺序排列。

```
    <order by clause> = order by <sort specification> [,<sort specification>]
    <sort specification> = <column name> [sort direction] | <scalar expression> [sort
direction] | <sequence number> [sort direction]
    <sort direction> = asc | desc
```

order by 子句可以按照列名排序，这种排序包含一个列指定。除了根据列名排序，排序也可以由标量表达式组成。order by 子句中的表达式甚至可以包含子查询(关联子查询也可以)。同时在 orderby 子句中，可以使用顺序号码来代替有列名或表达式组成的排序。

```
    select expenditure , account_payee from consume where account_id=1 order by
expenditure;
    select expenditure , account_payee from consume where account_id=1 order by 1;
```

上述两个 SQL 是等价的，但是从代码的可读性方面考虑，推荐使用第一种方式。

### 4.2.1.6. limit 子句的语法

limit 子句是一个选择语句块的最后一个子句，它选取了行的一个子集，由此，中间结果中的行数可以再次减少。

```
    <limit clause> = limit [<fetch offset>, ] <fetch number of rows> | limit <fetch
number of rows> [offset <fetch offset> ]
    <fetch number of rows>, <fetch offset> = <whole number>
```

### 4.2.1.7. join 联接的种类及其语法

将不同表的数据组合到一个表中就叫做表的联接(join)。在其上执行联接的列叫做联接列(join column)。联接分成隐式联接和显示联接。隐式联接是由 from 子句中的几个表指定以及 where 子句中的一个或多个条件构成的，这些条件横跨多个表，都是联接列。对于显示联接就是在 from 子句中增加 join 关键字，构成多表联接。定义如下：

```
    <from clause> = from <table reference> [,<table reference>]…
    <table reference> = <table specification> | <join specification>
    <join specification> = <table reference> <join type><table reference>[<join
condition>]
    <join condition> = on <condition> | using <column list>
    <join type> = [inner] join | left [outer] join | right [outer] join | natural [left|
right] [outer] join | cross join
    <column list> = (<column name> [,<column name>]…)
```

从上述定义上可以看出显示联接分为内连接，外连接，自然联接和交叉联接。设 t1，t2 代表两张不同的表，联接的同属性列是 c1，c2：

1. 如果要求列 t1，t2 在 c1，c2 的交集，则使用内连接。内连接的关键字 inner 可以省略；
2. 如果要以 c1 为基础列，从 c2 中选取和 c1 值相等的行与 c1 对应的行进行联接；对于 c1 中某值，c2 中没有相应的值，则 c1 此值对应的行联接后对应 t2 表中的列值以 null 代替。这种情况下，使用 c1 左连接 c2。右联接定义与左连接对应。外连接的关键字 outer 可以省略；
3. 对于多个表的联接，如果联接列的名字相同且只选取了一个联接列，这种情况可以使用自然连接，自然连接省略了联接条件地指定，默认使用几个表共同的属性列作为连接条件；
4. 在 MySQL 中，从语法上看，交叉联接和内连接等价。

# 4.2.2. select 子句的优化方法

## 4.2.2.1. 去掉 select 子句中非必要的结果列

在 select 子句中使用 select * 会给 MySQL 数据库造成比较大的资源开销和性能影响。比如 select * 会造成覆盖索引(covering index)失效，因为索引极少甚至不可能包含表中所有的属性列；在 MySQL 使用 filesort 时，由于 select * 中包含 blob，text 列，而导致 filesort 使用双路排序算法，从而增加 IO 和 SQL 执行时间；select * 本身返回了一些不需要的列，增加了无用的 IO，网络传输，CPU 等开销。

基于 select * 的缺点，在写 select 子句时，应该仔细分析需要返回的属性列，只把需要的属性列加入 select 子句。对于 select *要保持警惕，除非有确实的需求，一般不写这样的 select 子句。

## 4.2.2.2. select 子句中的聚合函数的优化

select 子句中可以包含 count(),min(),max()等聚合函数。而索引和列的可空性常常帮助 MySQL 优化掉这些聚合函数。比如，为了查找一个位于 B 树最左边的列的最小值，那么直接找索引的第一行就可以了。如果 MySQL 使用了这种优化，那么在 explain 中看到“select tables optimized away”。同样地，没有 where 子句的 count(*)通常也会被一些存储引擎优化掉(比如 MyISAM 总是保留着表行数的精确值)。

但是对于没有索引直接可用的 min()和 max()时，一般做不到很好地优化。可以尝试优化掉 min()和 max()，利用数据的有序性配合 limit 1 将 SQL 等价转化。

应用场景：

```
select min(customer_id)from customer where customer_name= '张三';
```

因为在 customer_name 上没有索引，所以查询会扫描整个表。从理论上说，如果 MySQL 扫描主键，它应该在发现第一个匹配之后就立即停止，因为主键是按照升序排列的，这意味着后续的行会有较大的 customer_id。但是在这个例子中，MySQL 会扫描整个表。一个变通的方式就是去掉 min()，并且使用 limit 来改写这个查询，如下：

```
select customer_id from customer use index(primary) where customer_name ='张三' limit 1;
```

这个通用策略在 MySQL 试图扫描超过需要的行时能很好地工作。

## 4.2.2.3. select 子句中加入控制结构避免重复实现优化

访问同一张表的连续几个查询应该引起注意，即使它们嵌在 if…then…else 结构中，也是如此。看下面的应用场景：它包含了两个查询，但只引用到了一张表。

```
select count(*) from consume where account_id= 6111 and account_payee= 51906734;
select count(*) from consume where account_id= 6111 and account_payee= 95161305;
```

这两个 SQL 访问同一张表(consume)，扫描同一个索引(index_accountid_expenditure_consumetime)两次。这些开销属于重复开销。可以使用汇总技术，不用依次检查不同的条件了，一遍就能收集到多个结果，然后再测试这些结果，而不需要在访问数据库。应该汇总技术改写合并上述两个 SQL 得到新的 SQL：

```
select sum(case account_payee when '51906734' then 1 else 0 end) as p1_sum, sum(case account_payee when '95161305' then 1 else 0 end) as p2_sum from consume where account_id=6111;
```

SQL 只需要扫描索引(index_accountid_expenditure_consumetime)一次及表(consume)就能获得结果，节省了很大的开销。

## 4.2.2.4. from 子句的优化方法

### 4.2.2.4.1. 去掉 from 子句中未涉及的 table 及非必要的联接

在 SQL 时，应该检查一下在 from 子句中出现的那些表的作用。这其中主要会出现 3 种类型的表：

1. 从中返回数据的表，其字段可能用于也可能没有用于 where 子句的过滤条件中；
2. 该表中没有要返回的数据，但在 where 子句中的条件使用了该表的的列；
3. 仅作为另两种表之间的“粘合剂”而出现的表，使 SQL 可以通过表连接将那些 1)，2)类型的表连在一起。
4. 既没有从该表中返回数据，也没有过滤条件涉及该表，同时该表也没有参加联接。

对于第 4)类型的表，属于 from 子句中整个 SQL 未涉及的表，可以从 from 子句中去掉。

对于第 3)类型的表，举例分析：表 A 联接表 B，联接列是 ab ，同时表 B 联接表 C。其中表 B 是第 3)类型的表。需要分析，A 表中的 ab 列是否可以为空值：如果可以空值，则 A 和 B 的联接对最后的结果集是有贡献的，不能去掉表 B 及 A 与 B 的联接；反之，不可以为空，如果表 A 可以直接联接表 C 的话，可以去掉表 B，表 A 直接联接表 C。上述结论是基于分析：空值的值是未知的，不能说它等于任何东西，两个空值也不相等。对于表 A 中的列 ab 可以为空值的情况，如果去掉了表 B 及表 A 与它的联接，那么表 A 中那些其它属性列的值符合过滤条件，但 ab 列对应值为空值的记录有可能进入最后的结果集。因此在这种情况，去掉表 B 及相关联接，可能改变结果集。

如下场景：

```
    select a.account_number from consume c, cust_acct b,account a where
a.account_id=b.account_id and b.account_id=c.account_id and c.expenditure>1.00;
```

分析 SQL 中表 c 的 account_id 不可以为空值，所以可以去掉表 b 及其相关的联接，改写成：

```
    select a.account_number from consume c,account a where a.account_id=c.account_id
and c.expenditure>1.00;
```

## 4.2.2.5. where 子句的优化方法

### 4.2.2.5.1. index 和 full table scan 的权衡

MySQL 获取获取所需要的数据表中数据的主要方式有两种：1 通过索引获得该数据所在行的位置后取得数据(index)；2 通过逐行扫描数据表中的数据行来过滤出所需要的数据(full table scan)。这两种数据获取方式各有利弊。

通过索引获取数据方式的利：

1. 可以通过索引迅速找到需要的数据在数据表中存放的位置，迅速定位。这种方式在通过索引定位获得的数据行数不多的情况下，效率高；
2. 通常情况下，索引数据和数据表相比较小，因此容易被缓存。

通过索引获取数据的弊：

如果通过索引获得的数据行数比较多，因为索引中的数据存放是有序的，而这种顺序如果和数据表中存放这些数据的顺序不一致，则在索引获得数据位置后从数据表中取数据会引起大量的随机读，这对 MySQL 的性能影响比较大。

通过扫描数据表获得所需数据方式的利：

1. 扫描数据表时，是按照数据表中数据存放位置顺序扫描，这是顺序读操作。现在的硬件设备对顺序读性能还是不错的；
2. 同时对于现在的硬件设备和文件系统，去读取某个 block 数据时，往往会将该 block 临近的其他几个 block 数据一起读到内存中，然后通过 virtual I/O 获取该 block。这种模式对于顺序扫表操作比较有利。

通过扫描数据表获得所需数据方式的弊：

如果数据表中的数据行数比较多，则扫表一次，需要耗费比较多的 CPU 时间和多次的 IO 操作。

了解了通过索引获取数据(index)和扫表获取数据(full table scan)两种方式各自的利弊后，在实际情况中如何取舍，需要考虑以下因素：

1. 数据表中的数据是否按照索引顺序存放，如 InnoDB 采用聚簇索引来存放数据(按照 primary key 顺序来存放数据)。这种情况下，通过索引获得的数据位置信息的顺序和数据存放顺序一致，因此不会引起大量的随机读。优先考虑使用这个索引；
2. 考虑通过能利用索引过滤余下的数据行数，如果过滤后余下行数依然很大，那意味着还需要大量的 IO 从数据表中获得需要的数据。如果是随机 IO，对 MySQL 的影响就更大了。而相比之下，顺序扫表，因为硬件和文件系统的特点(一次 cache 多个 block)的特点，而变得有优势。
3. 要考虑索引文件的大小。如果索引文件太大的话，无法在 MySQL 的 cache 中缓存下，则从索引中获得数据位置也会引起大量的 IO。

```
select account_id from cust_acct where customer_id=1;
```

如果 customer_id=1 条件对应的 account_id 太多，从执行时间上看不使用索引 index_customerid 而使用顺序扫表，效果更好；相反对于 customer_id=1，使用索引 index_customerid 效果更好。

## 4.2.2.5.2. where 子句中限制条件的排列与 index 的命中

因为 MySQL 中的 MyISAM，InnoDB 等引擎使用的索引主要都是 B-Tree 数据类型，所以对 where 子句中的限制条件的排列顺序会有一些限制。这些限制包括：

1. where 子句中限制条件排列最好为对应索引的最左前缀；
2. where 子句中的 range(范围)条件后的限制条件无法使用索引(即使这些条件对应的列在索引中)。

对于 1)MySQL 5.0 以上的 MySQL 优化器可以自己调整 where 子句中限制条件的顺序以使用对应的索引。但是这个过程本身会增加优化器的压力，尤其是 where 子句中的限制条件和对应索引包含列较多时。所以建议在写 SQL 的 where 子句中考虑打算使用的索引中列的顺序，相应地调整 where 子句中限制条件的顺序，以满足最左前缀的限制。

对于 2)可以考虑同时调整索引中列的顺序及 where 子句中限制条件的顺序:将 range(范围)限制条件调整到 where 子句的最后(最右)部分；将 range 条件对应的列调整到索引中列顺序的最后(最右)部。

```
select account_id from consume where expenditure>1.00 and account_payee =72478814;
```

如上语句，expenditure>1.00 是一个范围限制条件，可以调整其至 account_payee 条件之后。同时设计索引 index_accountpayee_expenditure 时考虑到这个问题，索引中列顺序为(account_payee,expenditure)。

#### 4.2.2.5.3. range 限制条件的优化(in 与>,<在 index 命中上的区别等)

在 SQL 中经常会出现多个 range(范围)限制条件。这里指的范围限制条件包括>,<,between 等。这些范围限制条件会对相应的索引使用造成影响。正常情况下，即使将这些范围条件调到 where 子句中的最后(最右)部，第一个范围限制条件后的范围条件对应的索引列也都无法在索引中同时被使用。

```
select account_id from consume where account_payee between 51906734 and 51907000 and expenditure>0.00;
```

因为 account_payee between 51906734 and 51907000 这个条件，得到的执行计划中只能使用索引 index_accountpayee_expenditure 中的 account_payee 列，而无法使用 expenditure 列。

在 SQL 中出现范围限制条件后,可以考虑这个范围条件对应到数据时,包含的值是否是有限个(个数不是太多)。如果存在这种特性，可以将范围条件转换为多个等值条件 in()。MySQL 对于 in()条件处理和等值条件一样，不会影响索引中其他列(右边列)的使用。针对上述应用场景，可以改为 in()：

```
select account_id from consume where account_payee in (51907000, 51906734, 51906740) and expenditure>0.00;
```

经过改写的 SQL 就可以使用索引 index_accountpayee_expenditure 中包含的全部列信息。

#### 4.2.2.5.4. in 减轻 optimizer MySQL 优化器的压力

一条 SQL 在 MySQL 中执行的过程中，会经过 SQL 解析，优化器制定执行计划，存储引擎查找过滤数据，返回客户端等步骤。其中优化器制定计划阶段有可能成为 MySQL 的性能瓶颈。优化器为一条 SQL 制定执行计划的过程相对比较复杂，其中包括语法树优化，相关开销信息获取，计算和比较等。如果并发比较大，在优化器处堆积了大量的 SQL 需要优化的话，优化器可能成为性能瓶颈。此时在 MySQL 中通过查看各个线程的状态(show processlist)，可以看到很多线程处在 statistics。

对于这种情况，可以检查一下向 MySQL 发起的 SQL 是否是形式一致，只是变量不同。如果是这种情况，可以由应用层在一个时间段(应用可以接受的)内收集这样的请求，使用 in()将不同变量的 SQL 合并成一个 SQL 发给 MySQL。这样 MySQL 原本需要制定数个执行计划的工作，变成只需要制定一个，减轻了优化器的压力。

应用场景：

```
select expenditure from consume where account_id =1;
```

可以将数个上述的 SQL 合并成下面的 SQL ，从而减轻优化器的压力，并且不会影响到索引的使用。

```
select expenditure from consume where account_id in(1,2,3);
```

#### 4.2.2.5.5. is null 限制条件的优化

对于 is null 限制条件，MySQL 会首先检查 is null 限制条件涉及的列，如果该列声明时是 not null 的，则优化器会直接忽略掉该 is null 限制条件。

对于可以为 null 的列上的 is null 限制条件，MySQL 同样可以使用该列上的索引。使用索引的条件和等值，range 限制使用索引的条件一样。

同时需要注意的是 MySQL 对于一条 SQL 只能使用索引来处理一个 is null 限制条件。比如：

```
select * from consume where account_payee is null and expenditure is null;
```

上述 SQL 只能使用 index_accountpayee_expenditure 中 account_payee 列，索引中 expenditure 列数据无法使用，这和等值限制不同。

## 4.2.2.5.6. index merge 优化方法

index merge(索引合并)方法适用于通过多个等值条件 index 定位或 range 扫描搜索数据行并将结果合并成一个的 SQL 语句。合并会产生并集，交集等。

对于下面的 SQL，MySQL 优化器会考虑使用 index merge 算法。

```
select id from cust_acct where customer_id=1 or account_id=1;
```

通过 explain 观察该 SQL 的执行计划，得到 MySQL 同时使用了 customer_id 列和 account_id 列上的两个索引。在这种情况下使用 index merge 会大大加速 SQL 的执行，可以将上面的 SQL 与下面的 SQL 进行比较。

```
select id from cust_acct ignore index(idx_accountid) where customer_id=1 or account_id=1;
```

同时需要指出如果 or 条件是在同一属性列上的，MySQL 则会考虑使用该属性列上的索引，这和上面所讲的 or 条件在不同属性列上是不同的。

```
select id from cust_acct where customer_id=1 or customer_id=2;
```

index merge 优化对于这种 or 条件的并集操作比较有效。对于 and 条件的交集操作，其效果不如联合索引。如：

```
select id from cust_acct where customer_id=1 and account_id=1;
```

建(customer_id,accout_id)的联合索引，MySQL 使用该联合索引的效果要优于使用 index merge 的效果。

对于 or 条件和 and 条件同时存在的 where 子句，MySQL 优化器会优先使用 and 子句中的索引，而放弃使用 or 条件的 index merge。如果能确定 or 条件的 index merge 优于 and 子句中的索引，可以使用 ignore index(and 子句的索引)的 hint 技术干预 MySQL 优化器制定的执行计划。

## 4.2.2.6. order by 子句的优化

在 MySQL 中，order by 的实现有如下两种类型：

1. 通过有序索引直接获得有序的数据，这样不用任何排序操作即可得到满足要求的有序数据；
2. 须通过 MySQL 的排序算法将存储引擎返回的数据进行排序后得到有序的数据。

order by 子句场景：

```
select consume_time from consume where account_payee=48370945 order by expenditure;
select account_id from consume where consume_time between '2009-09-07' and '2009-09-10' order by expenditure;
```

第一个 SQL 使用索引 index_accountpayee_expenditure。Order by 使用索引需要满足一些条件：where 子句中涉及的列加 order by 子句中涉及的列要满足要使用的索引的最左前缀；同时 where 子句中的条件必须是等值条件。

第二个 SQL 无法借助索引实现 order by，需要通过 MySQL 本身的排序算法完成 order by 操作。

利用索引实现数据排序是 MySQL 中实现结果集排序的最佳方法，可以完全避免因为排序计算带来的资源消耗。所以，在优化 SQL 中的 order by 时，尽可能利用已有的索引来避免实际的排序计算，甚至可以增加索引字段，这可以很大幅度地提升 order by 操作的性能。当然这需要从整体上权衡，不能因此影响其他 SQL 的性能。

### 4.2.2.7. 利用 limit 子句优化相关 SQL

在分页等系统中使用 limit 和 offset 是很常见的，它们通常也会和 order by 一起使用。一个比较常见的问题是偏移量很大，比如查询使用了 limit 10000,20，它就会产生 10020 行数据，并且丢掉前面 10000 行。这个操作代价比较高。

一个提高效率的简单技巧就是在覆盖索引上进行偏移，而不是对全行数据进行偏移。可以将从覆盖索引上提取的数据和全行数据进行联接，然后取得所需的列。这会更有效率。

应用场景：

```
    select * from consume where account_payee=72478814 order by expenditure limit 50,5;
    select * from consume inner join (select account_id from consume where
account_payee=72478814 order by expenditure limit 50,5) as lim using(account_id);
```

下面的 SQL 是上面 SQL 的优化版本，它使用覆盖索引，避免了整个结果集上进行 offset 操作。

### 4.2.2.8. 使用 covering index 优化 select 语句

在前面章节已经介绍了覆盖索引，覆盖索引可以使 MySQL 从索引中获得所需的数据，而不需要再到数据表中去取所需要的数据。通常索引要比数据表小，同时有序。因此使用覆盖索引对提高 SQL 性能会有很大的帮助。

在 MySQL 中，覆盖索引的使用要求 select 子句中涉及的列必须在相应的同一索引中，同时 where 子句中的限制条件应该符合使用这个索引的要求。比如限制条件要满足索引的最左前缀要求。如果有 range 条件，order by 子句等，都要满足索引对这些条件的限制。对于那些不满足这些条件的 SQL，可以通过适当的改造，以使它能使用覆盖索引达到优化的效果。

应用场景：

```
    select account_id from account where account_bank= 'icbc.beijing.fengtai' and
customer_email='email8002';
    select * from account where account_bank= 'icbc.beijing.fengtai' and
customer_email like '%@baidu.com';
```

因为 account 表使用的是 InnoDB 引擎，所以索引 index_accountbank_customeremail 存储有 account_id 的信息(它是 primary key)。第一个 SQL 可以使用覆盖索引;但是第二个 SQL 却不能,因为 select 子句要求返回所有列，超出了索引 index_accountid_expenditure_consumetime 的范围。可以将 SQL 修改一下：

```
    select * from account join (select account_id from account where
account_bank='icbc.beijing.fengtai' and customer_email like '%@baidu.com') t on
t.account_id=account.account_id;
```

通过上述改写，from 子句中的 select account_id from account where account_bank='icbc.beijing.fengtai' and customer_email like '%@baidu.com'可以使用覆盖索引。这样在 account_bank='icbc.beijing.fengtai'过滤出来的数据行比较多,而加上 customer_email like '%@baidu.com'条件后过滤出来的数据不多的情况下，这种改写会有比较大的优化效果。因为未改写前对 customer_email 条件过滤需要去数据表取数据，而改写后直接在索引中进行。同时由于改写后最终到数据表中取的数据较少，所以优化效果明显。

#### 4.2.2.9. 由返回结果推动反向优化技巧

在写完 SQL 后，一定将其在数据库上执行一遍，观察一下输出结果。如果输出结果呈一定的规律性，则可考察一下数据库中相应的数据是否都符合这种规律。如果是的话，则可以重新审视先前的 SQL，其中的限制条件等是否可以修改，来避免其中使用函数等复杂的限制条件。这样有助于 MySQL 指定执行计划时充分使用索引，指定最优的执行计划。同样有助于提升 SQL 的执行速度。

应用场景：

```
select account_id from account where substr(account_bank,6,7)= 'beijing';
```

上述 SQL 由于使用函数 substr，则不能使用索引 index_accountbank_customeremail。但是通过观察 SQL 的输出，发现 account_bank 列的数据都是以 icbc.开头，进而可以 SQL 改写如下：

```
select account_id from account where account_bank like 'icbc.beijing%';
```

因为限制条件改写成了 like 'icbc.beijing%'，符合 MySQL 索引使用规范，该 SQL 就可以使用索引 index_accountbank_customeremail 了。

### 4.2.3. join 联接的优化方法

#### 4.2.3.1. MySQL join 的相关算法

在介绍 join 语句的优化思路之前，首先要理解在 MySQL 中是如何实现 join 的。只要理解了其实现原理，优化就比较简单了。

在 MySQL 中，只有一种 join 算法，就是 nested loop join，它没有很多其他数据库提供的 hash Join，也没有 sort merge join。nested loop join 实际上就是通过驱动表的结果集作为循环基础数据，然后将该结果集中的数据(联接键对应的值)作为过滤条件一条条地到下一个表中查询数据，最后合并结果。如果还有第三个表参与 join，则把前两个表的 join 结果集作为循环基础数据，再一次通过循环查询条件到第三个表中查询数据，如此往复。

比如一个表 t1，t2，t3 的联接，通过 explain 观察到的执行计划中 join 的类型(explain 中 type 行的值)是：

```
table       join type
  t1          range
  t2          ref
  t3          All
```

则相应的这个联接的算法伪代码如下：

```
for each row in t1 matching range {
  for each row in t2 matching reference key {
    for each row in t3 {
      if row satisfies join conditions
      send to client
    }
  }
}
```

#### 4.2.3.2. 小结果集驱动大结果集

针对 MySQL 这种比较简单 join 算法，只能通过嵌套循环来实现。如果驱动结果集越大，所需循环也就越多，那么被驱动表的访问次数自然也就越多，而且每次访问被驱动表，即使所需的 IO 很少，循环次数多了，总量也不可能小，而且每次循环都不能避免消耗 CPU，所以 CPU 运算量也会跟着增加。但是不能仅以表的大小来作为驱动表的判断依据，假如小表过滤后所剩下的结果集比大表过滤得到的结果集还大，结果就会在嵌套循环中带来更多的循环次数。反之，所需要的循环次数就会更小，总体 IO 量和 CPU 运算量也会更少。所以，在优化 join 联接时，最基本的原则是“小结果集驱动大结果集”，通过这个原则来减少循环次数。

如下场景：

通过 explain 观察 MySQL 为该 SQL 制定的执行计划：

```
    select a.expenditure, b.balance from consume a, account b where a.account_id =
b.account_id and a.account_payee=13301087 and b.account_bank='icbc.beijing.fengtai';
```

join 中选取表 consume 为驱动表。

如果通过 explain 观察，发现 MySQL 在 join 过程中选取的驱动表不是很合适的话，建议最先通过索引，再通过 hint 技术 straight_join 来干预 MySQL，使其按照最优的方式选取驱动表，制定最优的执行计划。

#### 4.2.3.3. 被驱动表的 join column 最好能命中 index

对于 nested loop join 算法，内层循环是整个 join 执行过程中执行次数最多的，如果每次内层循环中执行的操作能节省很少的资源，就能在整个 join 循环中节约很多的资源。

而被驱动表的 join column 能使用索引正是基于上述考虑的。只有让被驱动表的 join 条件字段被索引了，才能保证循环中每次查询都能通过索引迅速定位到所需的行，这样可以减少内层一次循环所消耗的资源；否则只能通过扫表等操作来找到所需要的行。

例如

```
    select a.expenditure, b.balance from consume a, account b where a.account_id =
b.account_id and a.account_payee=13301087 and b.account_bank='icbc.beijing.fengtai';
```

join 过程中被驱动表 join 条件列 account_id 是 primary key，通过主键索引每次内层循环定位所需的行非常快且开销非常小。

## 4.3. 数据更新语句语法

### 4.3.1. insert 语法，优化 tips

#### 4.3.1.1. insert 语法

insert 语句，它的基本含义是往数据库中插入新的数据，其语法主要有三类，分别说明如下：

1. insert into tbl_name values …

```
    insert [low_priority | delayed | high_priority] [ignore]
      [into] tbl_name [(col_name,...)]
      {values | value} ({expr | default},...),(...),...
      [ on duplicate key update  col_name=expr[, col_name=expr] ... ]
```

首先[low_priority | delayed | high_priority] [ignore]部分是说明写入操作的优先级，low_priority 选项表示写操作会被延迟直到所有没有任何对 tbl_name 表的读操作，但是如果指定了这个选项，MyISAM 引擎的并发写入就无法生效，即使它原本是可以并发插入的；delayed 选项表示插入操作暂时缓存在服务器端被延迟插入，客户端可以立即返回，当表处于空闲状态时，在进行插入操作。但是这个选项在第三类型的 insert into … select …和 insert … on duplicate key update 语句中无法生效；high_priority 选项表明写入时最高优先级，它的状态级别优先于--low-priority-updates，同样它会也让 MyISAM 引擎的并发写入失效。

特别注意的是，low_priority 和 high_priority 选项只影响表级锁的存储引擎，如 MyISAM、Memory 和 Merge 三种引擎的表。如 InnoDB 引擎，这几个选项就无法生效。

ignore 选项表示将执行 insert 语句返回的错误当成 warning，即可以跳过错误语句，这样可以保证一个线程在批量执行 SQL 时，其中一个 SQL 错误不会导致整个处理都退出；而是跳过错误继续执行。这种错误包括 duplicate-key 的错误等。

接下来那个表名表示被插入的字段和具体的值，特别需要说明的是 on duplicate key update 选项，如果插入的值与原来主键、unique 索引中的键值一样，那么将会执行更新操作，通过返回结果观察，影响的行数(即“affected-rows”)就是 2，相对于插入操作影响的行数为 1。

例子：我们往 customer(customer_id(int)、customer_name(varchar)、customer_contact (varchar)、customer_phone(varchar))插入一条数据，SQL 语句如下：

```
    insert into customer values (20101213, '百度在线', '海淀区上地百度大厦',
'010-59928888');
```

若插入的数据，其 customer_id 字段是自增的，那么插入时则不能指定 customer_id 字段的值，需要在插入字段列表明确列出，如下所示：

```
    insert into customer(customer_name, customer_contact, customer_phone) values ('
百度在线', '海淀区上地百度大厦','010-59928888');
```

若原来可能一个 id 为 20101213 的客户，那么不确定的情况下可以采用 on duplicate key update，方式如下：

```
    insert into customer (customer_id, customer_phone ) values (20101213,
'010-59926666') on duplicate key update customer_phone='010-59926666';
```

2. insert into tbl_name set col_name=expr...

```
    insert [low_priority | delayed | high_priority] [ignore]
        [into] tbl_name
        set col_name={expr | default}, ...
        [ on duplicate key update  col_name=expr[, col_name=expr] ... ]
```

这种语法其实与上面的类似，只不过写法不一样，它把字段的指定挪到 set 后面，表示每个字段的插入值为多少，其它的部分说明与上面的一样。

例子：针对上面的例子，采用此从句的方法如下：

```
    insert into customer set customer_name='百度在线', customer_contact='海淀区上地百
度大厦', customer_phone='010-59928888';
```

3. insert into tbl_name select …

```
    insert [low_priority | high_priority] [ignore]
     [into] tbl_name [(col_name,...)]
     select ...
     [ on duplicate key update  col_name=expr[, col_name=expr] ... ]
```

这种语法与上面的(1)和(2)类似，主要是数据是从已知的另外一个表中获取，如数据转移应用中。

如一个统计表中，有许多统计数据，数据来源于多个数据库和表，很多数据需要使用批量更新方法，(不可能查出所有数据然后一条一条去更新吧)，因此需要使用 insert into table1 select * from table2 这样的语句。

同时，由于需要多次更新 table，table 定义了一个 unique key，因此如果第二次运行或者更新第二批数据的时候，会产生 error code : 1062 duplicate entry ‘xx-yy’ key for 'keyname' 错误，自然我们会想到用 on duplicate key 来解除约束，这个时候就需要用到 values 取值函数 values 取值函数：values(col_name)可以引用被插入的 col_name 的值，注意这个函数只在 insert ... Update 语句中有意义。

```
    insert into table1 (a, b, c) select a, b, c from table2 on duplicate key update
c = values(c);
```

注意 values(c)中的 c 不需要加任何修饰，table 表设有 a,b 联合 unique key。

#### 4.3.1.2. insert 优化 tips

insert 插入一条记录花费的时间由以下几个因素决定：连接、发送查询给服务器、解析查询、插入记录、插入索引和关闭连接。

此处没有考虑初始化时打开数据表的开销，因为每次运行查询只会做一次。如果是 B-tree 索引，随着索引数量的增加，插入记录的速度以 logN 的比例下降。

可以用以下几种方法来提高插入速度：

1. 如果要在同一个客户端在同一时间内插入很多记录，可以使用 insert 语句附带有多个 values 值。这种做法比使用单一值的 insert 语句快多了(在一些情况下比较快)。如果是往一个非空数据表增加记录，可以调整变量 bulk_insert_buffer_size 的值使其更快。
2. 对实时性要求不高情况下，如要从不用的客户端插入大量记录，使用 insert delayed 语句也可以提高速度。
3. 对应 MyISAM，可以在 select 语句正在运行时插入记录，只要表中没有空洞（由删除记录引起）或者打开 MySQL 相关设置。
4. 对于 MyISAM，在进行大批量插入前可以将索引关闭，等全部插入完毕后再开启索引，进行索引更新。
5. 想要将一个文本文件加载到数据表中，可以使用 load data infile。速度上通常是使用大量 insert 语句的 20 倍。
6. 对于 InnoDB 的 insert，若插入上百万，建议分批进行，批量插入 3000~5000 后，sleep 数秒钟之后进行下一次插入，可以避免同步延迟的累积。

### 4.3.2. delete 语法，优化 tips

#### 4.3.2.1. delete 语法

delete 语法的实际操作步骤以及对 MySQL delete 语法的实际应用代码的描述如下：

1. 单表语法

```
    delete [low_priority] [quick] [ignore] from tbl_name
    [where where_definition]
    [order by ...]
    [limit row_count]
```

2. 多表语法

```
    delete [low_priority] [quick] [ignore]
```

```
    tbl_name[.*] [, tbl_name[.*] ...]
    from table_references
    [where where_definition]
```

或：

```
    delete [low_priority] [quick] [ignore]
    from tbl_name[.*] [, tbl_name[.*] ...]
    using table_references
    [where where_definition]
```

tbl_name 中有些行满足由 where_definition 给定的条件，那么 delete 将用于删除这些行，并返回被删除的记录的数目。

#### 4.3.2.2. delete 优化

如果编写的 delete 语句中没有 where 子句，则所有的行都被删除。当不想知道被删除的行的数目时，有一个更快的方法，即使用 truncate table。

如果删除的行中包括用于 auto_increment 列的最大值，对于 MyISAM 表或 InnoDB 表不会被重新用。如果在 autocommit 模式下使用 delete from tbl_name(不含 where 子句)删除表中的所有行，则对于所有的表类型(除 InnoDB 和 MyISAM 外)，序列重新编排。对于 InnoDB 表，此项操作有一些例外。

对于 MyISAM 和 BDB 表，您可以把 auto_increment 次级列指定到一个多列关键字中。在这种情况下，从序列的顶端被删除的值被再次使用，甚至对于 MyISAM 表也如此。delete 语句支持以下修饰符：

如果您指定 low_priority，则 delete 的执行被延迟，直到没有其它客户端读取本表时再执行。

对于 MyISAM 表，如果使用 quick 关键词，则在删除过程中，存储引擎不会合并索引端结点，这样可以加快部分种类的删除操作的速度。

在删除行的过程中，ignore 关键词会使 MySQL 忽略所有的错误。(在分析阶段遇到的错误会以常规方式处理。)由于使用本选项而被忽略的错误会作为警告返回。

在 MyISAM 表中，被删除的记录被保留在一个带链接的清单中，后续的 insert 操作会重新使用旧的记录位置，要想重新使用未使用的空间并减小文件的尺寸，则可以使用 optimize table 语句或 myisamchk 应用程序重新编排表。optimize table 更简便，但是 myisamchk 速度更快。

quick 修饰符会影响到在删除操作中索引端结点是否合并。当用于被删除的行的索引值被来自后插入的行的相近的索引值代替时，delete quick 最为适用。在此情况下，被删除的值留下来的空穴被重新使用。

未充满的索引块跨越某一个范围的索引值，会再次发生新的插入。当被删除的值导致出现未充满的索引块时，delete quick 没有作用。在此情况下，使用 quick 会导致未利用的索引中出现废弃空间。

### 4.3.3. update 语法及优化 tips

#### 4.3.3.1. update 语法

更新语句主要也有两类，单表和多表更新：

1. 单表更新语法

```
    update [low_priority] [ignore]
    tbl_name
    setcol_name1=expr1[,col_name2=expr2...]
    [where where_definition]
    [order by ...]
```

```
    [limit row_count]
```

2. 多表更新语法

```
    update [low_priority] [ignore]
    table_references
    set   col_name1=expr1[,col_name2=expr2...]
    [where where_definition]
```

update 语法可以用新值更新原有表行中的各列。set 子句指示要修改哪些列和要给予哪些值。where 子句指定应更新哪些行。如果没有 where 子句，则更新所有的行。如果指定了 order by 子句，则按照被指定的顺序对行进行更新。limit 子句用于给定一个限值，限制可以被更新的行的数目。

多表更新的例子:

```
    update consume a, account b set b. balance=a. expenditure where
a.account_id=b.account_id;
```

以上的例子显示出了使用逗号操作符的内部联合，但是多表更新语句可以使用在 select 语句中允许的任何类型的联合，比如 left join。

update 语句支持以下修饰符：

1. 如果您使用 low_priority 关键词，则 update 的执行被延迟了，直到没有其它的客户端从表中读取为止。
2. 如果您使用 ignore 关键词，则即使在更新过程中出现错误，更新语句也不会中断。如果出现了重复关键字冲突，则这些行不会被更新。如果列被更新后，新值会导致数据转化错误，则这些行被更新为最接近的合法的值。

#### 4.3.3.2. update 优化

update 更新查询的优化同 select 查询一样，但需要额外的写开销。写的速度依赖更新的数据大小和更新的索引的数量。所以，锁定表，同时做多个更新比一次做一个快得多。

另一个提高更新速度的办法是推迟更新并且把很多次更新放在后面一起做。如果锁表了，那么同时做很多次更新比分别做更新来得快多了。

注意，如果是在 MyISAM 表中使用了动态的记录格式，那么记录被更新为更长之后就可能会被拆分。如果经常做这个，那么偶尔做一次 optimize table 就显得非常重要了。

### 4.3.4. replace 语法，优化 tips

#### 4.3.4.1. replace 语法

replace 是 MySQL 对 SQL 标准的扩展，它会插入记录，或者删除记录再插入记录。如果 replace 使用的主键或者唯一索引列的值在表中能够找到，则删除该记录再插入新的记录，否则只是插入。如果表没有主键或者唯一索引，那么只是插入。

```
    replace [low_priority | delayed]
    [into] tbl_name [(col_name,...)]
    values ({expr | default},...),(...),...
```

或:

```
    replace [low_priority | delayed]
    [into] tbl_name
```

```
    set col_name={expr | default}, ...
```

或：

```
    replace [low_priority | delayed]
    [into] tbl_name [(col_name,...)]
    select ...
```

replace 的运行与 insert 很相像。只有一点除外，如果表中的一个旧记录与一个用于 primary key 或一个 unique 索引的新记录具有相同的值，则在新记录被插入之前，旧记录被删除。

注意，除非表有一个 primary key 或 unique 索引，否则，使用一个 replace 语句没有意义。该语句会与 insert 相同，因为没有索引被用于确定是否新行复制了其它的行。

当然，我们可能想用新记录的值来覆盖原来的记录值。如果使用传统的做法，必须先使用 delete 语句删除原先的记录，然后再使用 insert 插入新的记录。而在 MySQL 为我们提供了一种新的解决方案，这就是 replace 语句。使用 replace 插入一条记录时，如果不重复， replace 就和 insert 的功能一样，如果有重复记录，replace 就使用新记录的值来替换原来的记录值。使用 replace 的最大好处就是可以将 delete 和 insert 合二为一。这样就不必考虑在同时使用 delete 和 insert 时添加事务等复杂操作了。在使用 replace 时，表中必须有唯一索引，而且这个索引所在的字段不能允许空值，否则 replace 就和 insert 完全一样的。

在执行 replace 后，系统返回了所影响的行数，如果返回 1，说明在表中并没有重复的记录，如果返回 2，说明有一条重复记录，系统自动先调用了 delete 删除这条记录，然后再记录用 insert 来插入这条记录。如果返回的值大于 2，那说明有多个唯一索引，有多条记录被删除和插入。

#### 4.3.4.2. replace 语法

由于 replace 是先 delete 再 insert 的操作，有可能会导致系统的空洞无法得到使用。所以采用先 select 判断是否存在记录，然后再考虑是否进行 insert 还是 update 的方式进行数据的更新可能更好。可以考虑采用 insert on duplicate key，replace 和 insert on duplicate key 差别在于前者是 delete-insert，后者是 update。

### 4.3.5. truncate 语法，优化 tips

#### 4.3.5.1. truncate 语法

```
    truncate tbl_name;
```

truncate tbl_name 用于完全清空一个表。从逻辑上说，该语句与用于删除所有行的 delete 语句等同，但是在有些情况下，两者在使用上有所不同。

对于 InnoDB 表，如果有需要引用表的外键限制，则 truncate table 被映射到 delete 上；否则使用快速删减(取消和重新创建表)。使用 truncate table 重新设置 auto_increment 计数器，设置时不考虑是否有外键限制。

对于其它存储引擎，在 MySQL 中，truncate table 与 delete from 有以下几处不同：

1. 删减操作会取消并重新创建表，这比一行一行的删除行要快很多。
2. 删减操作不能保证对事务是安全的；在进行事务处理和表锁定的过程中尝试进行删减，会发生错误。
3. 被删除的行的数目没有被返回。
4. 只要表定义文件 tbl_name.frm 是合法的，则可以使用 truncate table 把表重新创建为一个空表，即使数据或索引文件已经被破坏。

5. 表管理程序不记得最后被使用的 auto_increment 值，但是会从头开始计数。即使对于 MyISAM 和 InnoDB 也是如此。MyISAM 和 InnoDB 通常不再次使用序列值。
6. 当被用于带分区的表时，truncate table 会保留分区；即，数据和索引文件被取消并重新创建，同时分区定义(.par)文件不受影响。

#### 4.3.5.2. truncate 优化 tips

truncate 和不带 where 子句的 delete，以及 drop 都会删除表内的数据；

truncate 和 delete 只删除数据不删除表的结构(定义)；drop 语句将删除表的结构被依赖的约束(constrain)，触发器(trigger)，索引(index)；依赖于该表的存储过程/函数将保留，但是变为 invalid 状态；

delete 语句是 DML 语句，这个操作会放到 rollback segement 中，事务提交之后才生效，如果有相应的 trigger,执行的时候将被触发;truncate,drop 是 DDL 语句,操作立即生效,原数据不放到 rollback segment 中，不能回滚．操作不触发 trigger；

在执行速度方面，一般来说: drop> truncate > delete；

但是在安全性上，小心使用 drop 和 truncate，尤其没有备份的时候；

使用上，想删除部分数据行用 delete，注意带上 where 子句，否则回滚段要足够大；想删除表，当然用 drop；想保留表而将所有数据删除，如果和事务无关，用 truncate 即可；如果和事务有关，或者想触发 trigger，还是用 delete.

## 4.4. 子查询(subquery)

当一个查询语句嵌套在另一个查询的查询条件之中时，称为子查询。子查询总是写在圆括号中，可以用在使用表达式的任何地方。子查询也称为内部查询或者内部选择,而包含子查询的语句也称为外层查询或外层选择。MySQL 是从 4.1 支持子查询功能的，之前可考虑使用联表查询进行替代。

根据子查询出现的位置：位于 select 子句中，位于 from 子句中，位于 where 条件中的子查询来逐个介绍。

### 4.4.1. 位于 select 子句中的子查询

位于 select 子句的子查询是指子查询在外层查询的 select 项。此时子查询返回的值是 n 行一列的表集合(n>=1)。

应用场景：账号名为'a-001'的客户信息。

```
select a.account_id,a.balance,
(select c.customer_name
from customer as c
where c. customer_id=ac.customer_id )as cust_name
from account as a, cust_acct as ac
where a. account_id=ac.account_id and a. account_id= 'a-001';
```

子查询通过与 ac.customer_id 相等的条件引用了主查询的当前记录。而且该子查询只能返回一条记录，如果没有符合条件的，则结果中的 cust_name 值为空。

优化关注点：对于子查询来说，外层查询返回的每条记录都会需要执行这个子查询。如果返回几百条或者更少记录来说，上面语句的性能应该还不错，但是对于返回几百万，上千万行数据的查询来说，这样做就非常致命了。可以替换为外连接，如下：

```
    select a.account_id,a.balance,c.customer_id
    from account as a, cust_acct as ac left join customer as c
    on c.customer_id = ac.customer_id
    where a.account_id=ac.account_id and a.account_id= 'a-001';
```

注意，连接需要是个外连接，这样才能在 customer 表中无对应记录时得到空值。

### 4.4.2. 位于 from 中的子查询

位于 from 中的子查询是指子查询落在外层查询的 from 项。此时子查询返回的值是 n 行 n 列的表集合(n>=1)。

应用场景：一次消费超过 100 元的同时消费是在最近一个月进行的账号信息。

```
    select consume_gt100.account_id
    from
    (
    select c2. account_id,c2. consume_time
    from consume as c2
    where c2. expenditure>100
    ) as consume_gt100
    where consume_gt100. consume_time>'2010-11-17';
```

其中子查询和外层查询没有任何关联。一般情况下，from 中的子查询都是可以独立执行的。

### 4.4.3. 位于 where 中的子查询

位于 whrere 中的子查询是指子查询位于外层查询的 where 条件中。这类查询通常有三种基本的子查询。

1. 通过使用 in 引起的范围查询。
2. 通过由 any,some 或 all 修改的比较运算符引入的列表上操作。
3. 通过 exists 引入的存在测试。

如上三种子查询与外层查询极可能有关联，也可能无关联。第一种和第二种属于无关联的，可以独立执行；exists 一般是和外层查询都是有关联的。

位于 where 中的子查询暂时有个限制，在带 in 或者由 any,some,all 的嵌套子查询中，不能使用 limit 关键字。但是此限制可通过其他方式来变相实现。

关联子查询和非关联子查询的两个较大的区别如下：

1. 关联子查询在来自外层查询的值每次发生变化就会被触发一次；而非关联子查询却永远只需要触发一次。
2. 关联子查询的话，是外层查询在驱动执行过程。如果是无关联查询，那么子查询就有可能驱动外层查询。

#### 4.4.3.1. 带 in 的嵌套查询

应用场景：和张三有在同一银行开户的账号的客户信息，并且按照 customer_id 递减排序，取前 10 个客户信息。

```
    select c. customer_name,c.customer_id
    from customer as c, cust_acct as ac, account as a
    where c.customer_id=ac.customer_id and a.account_id=ac.account_id and a.
account_bank in(
    select a2.account_bank
    from account as a2,cust_acct as ac2,customer as c2
```

```
    where c2.customer_name='张三' and c2.customer_id=ac2.customer_id and
ac2.account_id=a2.account_id
    )
    order by c.customer_id desc limit 0,10;
```

该 SQL 中的子查询为非关联子查询，该子查询也可以移到 from 中。但是由于 in()隐含了 distinct，因此在 from 子查询中需要显式写 distinct。如上可以改写为：

```
    select c.customer_name,c.customer_id
    from customer as c,cust_acct as ac,account as a,
    (
    select distinct a2.account_bank
    from account as a2,cust_acct as ac2,customer as c2
    where c2.customer_name='张三' and c2.customer_id=ac2.customer_id and
ac2.account_id=a2.account_id
    ) as tr
    where tr. account_bank=a. account_bank
    and c.customer_id=ac.customer_id and a.account_id=ac.account_id
    order by c.customer_id desc limit 0,10;
```

这两种写法的对比：

1. from 子句支持 limit 子句，在索引合适的情况下，添加 order by 和 limit 在子查询中可减少文件排序。
2. 带 in 的子查询和联表比起来，MySQL 更喜欢联表。尤其是 in()中子查询为空时，外层查询会逐行扫描对比，而此时联表查询会有较大优势。

### 4.4.3.2. 带 any，some 或者 all 的嵌套查询

some,any 和 all 可对子查询中返回的多行结果进行处理，下面简单介绍下这几个关键字的含义：

some：表示满足其中一个的含义，是由 or 连起来的比较从句。

any：也表示满足其中一个的意义，也是用 or 串起来的比较从句，区别是 any 一般用在非“=”的比较关系中，这也很好理解，英文中的否定句中使用 any，肯定句中使用 some，这一点是一样的。

all：表示满足其中所有的查询结果的含义，使用 and 串起来的比较从句。

应用场景：本月内消费额度超过'a-001'账户的所有账户信息。

```
    select c.customer_name,c.customer_id
    from customer as c ,cust_acct  as ac, consume as c2
    where c.customer_id=ac.customer_id and ac.account_id=c2.account_id
    and c2.expenditure > any (select c3.expenditure
    from consume as c3
    where c3.account_id='a-001');
```

这里改为>some(select c3. expenditure  from consume c3  where c3.account_id='a-001')也是符合语法的，查询结果也是一致的。

### 4.4.3.3. 带 exists 的嵌套查询

应用场景：账户开户以来一直没有消费记录的账户信息。

```
    select c.customer_name,c.customer_id
```

```
from customer as c,cust_acct as ac
where c.customer_id=ac.customer_id and not exists
(
select c2. account_id
from consume as c2
where c2.account_id=ac.account_id
);
```

该子查询是通过 account_id 进行关联，该查询也可以改写为无关联查询，如下：

```
select c.customer_name,c.customer_id
from customer as c,cust_acct as ac
where c.customer_id=ac.customer_id and ac.account_id not in
(
select c2. account_id
from consume as c2
);
```

在关联子查询和非关联子查询之间做选择并不是很困难，有些优化器会帮你做选择，在使用关联子查询和非关联子查询时，对于索引情况的假设不同，因为它不再是其他部分的查询的那个部分了。优化器会尽可能使用可用的索引，但不会创建索引。

# 4.5. 优化器相关 explain 以及常用 hint 介绍

## 4.5.1. 查看 select 语句执行计划的语法 explain

### 4.5.1.1. MySQL explain 输出信息介绍

MySQL Query Optimizer 通过执行 explain 命令告诉我们它将使用一个怎么样的执行计划来优化 query。因此 explain 是在优化 query 中最直接有效的验证我们想法的工具。

要使用 explain，只需把 explain 放在查询语句的关键字 select 前面就可以了。MySQL 会在查询里设置一个标记，当它执行查询时，这个标记会促使 MySQL 返回执行计划里每一步的信息。用不着真正执行。它会返回一行或多行。每行都会显示执行计划的每一个组成部分，以及执行的次序。

MySQL 只能解释 select 查询，无法解释存储过程的调用、insert、update、delete 和其它语句。这时只有通过重写这些非 select 语句为 select，使能够被 explain。

下面来详细解释下 explain 功能中展示的各种信息的解释。

| 项 | 子类型 | 说明 |
|---|---|---|
| id | | MySQL Query Optimizer 选定的执行计划中查询的序列号。表示查询中执行 select 子句或操作表的顺序，id 值越大优先级越高，越先被执行。id 相同，执行顺序由上至下。 |
| select_type（所使用的查询类型） | dependent subquery | 子查询内层的第一个查询，依赖于外部查询的结果集。 |
| | dependent unoin | 子查询中的 unoin，且为 union 中从第二个 select 开始的后面所有 select，同样依赖于外部查询的结果集。 |
| | primary | 子查询中的最外层查询，注意并不是主键查询。 |
| | simple | 除子查询或 union 外的其他查询。 |

| | | |
|---|---|---|
| | derived | 用于 from 子句里有子查询的情况。MySQL 会递归执行这些子查询，把结果放在临时表里。 |
| | subquery | 子查询内层查询的第一个查询，结果不依赖于外部查询结果集。 |
| | uncacheable subquery | 结果集无法缓存的子查询。 |
| | union | union 语句中第二个 select 开始后面的所有 select，第一个 select 为 primary。 |
| | union result | union 中的合并结果集。 |
| table | | 显示这一步中锁访问的数据库中的表名称。 |
| type（表示表的连接类型） | all | 通常意味着必须扫描整张表，从头到尾，去找匹配的行。(这里也有例外，例如查询中使用了 limit，或者在 extra 列里显示使用了 distinct 或 not exists 等限定词)。 |
| | index | 全索引扫描。与全表扫描一样，只是扫描表的时候按照索引次序进行而不是行。主要优点就是避免了排序；最大的缺点就是要承担索引次序读取整张表的开销。这一般意味着若是随机次序访问行，开销将会非常大。 |
| | range | 索引范围扫描。就是有限制的索引扫描，返回匹配某个值域的行。这比全索引扫描好些，因为不用便利遍历全部索引。常见于 between、<、>等的查询。 |
| | ref | 一种索引访问方式，它返回匹配某个单独值的所有行。但是它可能查找到多个符合条件的行，因此，它是查找和扫描的混合体。此类索引访问只有当使用非唯一索引或者唯一性索引的非唯一性前缀才会发生。叫做 ref 是因为索引要跟某个参考值相比较。这个参考值或者是一个常数，或者是来自一个表里的多表查询的结果值。ref_or_null 是 ref 之上的一个变体，它意味着 MySQL 必须进行二次查找，在初次查找的结果里找出 null 条目。 |
| | 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。 |
| | const，system | 当 MySQL 对查询某部分进行优化，并转换为一个常量时，使用这些类型访问。如将主键置于 where 列表中，MySQL 就能将该查询转换为一个常量。system 是 const 类型的特例，当查询的表只有一行的情况下， 使用 system。 |
| | null | MySQL 在优化过程中分解语句，执行时甚至不用访问表或索引。 |
| possible_key | | 这一列显示查询可以使用的索引，如果没有索引可以使用，就会显示为 null。 |
| key | | 这一列显示的是 MySQL Query Optimizer 从 possible_key 选择使用的索引。需要注意的是查询中若使用了覆盖索引，则该索引仅出现在 key 列表中，不会出现在 possible_keys 列。possible_keys 说明哪一个索引能有助于查询，而 key 显示的是优化器采用哪一个索引可以最小化查询成本。 |
| key_len | | 该列显示了使用的索引的索引键长度。由于索引的最左策略，因此通过该值可以计算查询中使用的索引情况。 |

| | | |
|---|---|---|
| ref | | 这一列显示上述表的连接匹配条件，即哪些列或常量被用于查找索引列上的值。 |
| rows | | 这一列显示的表示 MySQL 根据表统计信息及索引选用情况，估算的找到所需的记录所需要读取的行数。 |
| filtered | | 这一列是 5.1 加进去的。当使用 explain extended 时才会出现。它显示的是针对表里符合某个条件的记录数的百分比所作的一个悲观估算。rows 列和这个百分比相乘，就能看到 MySQL 估算的它将和查询计划里前一个表联接的行数 |
| extra（这一列包含的是不适合在其他列显示的额外信息） | using index | 该值表示 MySQL 将使用覆盖索引，以避免访问表。 |
| | using where | 表示 MySQL 服务器在存储引擎收到记录后进行“后过滤”(post-filter),如果查询未能使用索引，using where 的作用只是提醒我们 MySQL 将用 where 子句来过滤结果集。 |
| | using temporary | 表示 MySQL 在对查询结果排序时使用临时表。常见于排序和分组查询。 |
| | using filesort | 表示 MySQL 会对结果使用一个外部索引排序，而不是从表里按索引次序读到相关内容。可能在内存或者磁盘上进行排序。MySQL 中无法利用索引完成的排序操作称为“文件排序” |
| | range checked for each record(index map:n) | 这表示没有好的索引可用，新的索引将在联接的每一行上被重新评估。n 是显示在 possible_keys 列索引的位图。这是一个冗余。 |

## 4.5.1.2. extend explain 输出信息

explain extended 和普通的 explain 很相似，但是它会告知服务器把执行计划“反编译”成 select 语句，然后立即执行 show warnings 就能看到这些生成的语句。这些语句是直接来自执行计划，而不是原始的 SQL 语句。

## 4.5.1.3. explain 的局限

explain 只是一个近似，没有更多的细节。有时它是一个很好的近似，但是有时它会远离真实情况，以下就是它的几个局限性：

1. explain 不会告诉你关于触发器、存储过程的信息或用户自定义函数对查询的影响情况。
2. explain 不考虑各种 cache。
3. explain 不能显示 MySQL 在执行查询时所作的优化工作。
4. 一些显示出来的统计信息是估算的，不是很精确。
5. expalin 只能解释 select 操作，其他操作要重写为 select 后查看执行计划。

#### 4.5.1.4. MySQL 制定的执行计划不一定最优

尽管优化器的作用是将很糟糕的语句转化成高效的处理方式，但即便是在索引建立的很合理，并且优化器得到正确的相关信息后，优化器还是可能会制定出错误方向的执行计划。一般会有两个方面的原因：

1. 优化器的缺陷。
2. 查询太复杂，优化器只能在给它的有限时间内尝试大量可能的组合中很少一部分优化方案，因此可能无法找到最佳执行计划。

针对这种情况，最有效的解决办法就是正确编写查询。以更简单更直接的方式写 SQL 通常能节省优化器很多工作，因为它能很快命中一个很好而且很高效的执行计划。

### 4.5.2. MySQL 干预执行计划的方法(hint)

如果不满意 MySQL 优化器选择的优化方案，可以使用一些优化提示来控制优化器的行为。下面简单介绍下在 MySQL 中常用的提示，以及使用它们的时机。

#### 4.5.2.1. force index、use index、ignore index

这些提示告诉优化器在该表中查询时使用或者忽略该索引。

force index 和 using index 是一样的。但是它告诉优化器，表扫描比起索引来说代价要高很多，即使索引不是非常有效。

在 MySQL5.0 及以前版本中，它们不会影响排序和分组使用的索引。

#### 4.5.2.2. straight_join

这个提示用于 select 语句中 select 关键字的后面，也可以用于联接语句。因此它的第一个用途是强制 MySQL 按照查询中表出现的顺序来联接表，第二个用途是当它出现在两个联表的表中间时，强制这两个表按照顺序联接。

straight_join 在 MySQL 没有选择好的连接顺序，或者当优化器花费很长时间确定连接顺序的时候很有用。在后一种的情况下，线程将会在“统计”状态停留很长时间，添加这个提示将会减少优化器的搜索空间。

可以使用 explain 查看优化器选择的联接顺序，然后按照顺序重写连接，并且加上 straight_join 提示。

#### 4.5.2.3. sql_no_cache、sql_cache

sql_cache 表明查询结果需要进行缓存，结果进行缓存和变量 query_cache_type，have_query_cache 以及 query_cache_limit 的设置有关。

而 sql_no_cache 表明查询结果不需要进行缓存。

#### 4.5.2.4. high_priority、low_priority

这两个提示决定了访问同一个表的 SQL 语句相对其它语句的优先级。

high_priority 提示用于 select 和 insert。是将一个查询语句放在队列的前面，而不是在队列中等待。

low_priority 提示用于 select、insert、update、replace、delete、load data。和 high_priority 相反，如果有其他语句访问数据，它就把当前语句放在队列的最后。

这两个提示不是指在查询上分配较多或者较少资源。它们只是影响服务器对访问表的队列的处理。

#### 4.5.2.5. delayed

这个提示用于 insert 和 update。使用了该提示的语句会立即返回并且将插入的列放在缓冲区中，在表空闲的时候在执行插入。它对于记录日志很有用，对于某些需要插入大量数据，对每一个语句都引发 I/O 操作但是又不希望客户等待的应用程序很有用。但是它有很多限制，比如：延迟插入不能运行于所有的存储引擎上(仅适用于 MyISAM, Memory 和 Archive 表)，并且它也无法使用 last_insert_id()。

#### 4.5.2.6. sql_small_result、sql_big_result

这两个提示用于 select 语句。它们会告诉 MySQL 在 group by 或 distinct 查询中如何并且何时使用临时表。sql_small_result 告诉优化器结果集会比较小，可以放在索引过的临时表中，以避免对分组后的数据排序。sql_big_result 的意思是结果集比较大，最好使用磁盘上的临时表进行排序。

#### 4.5.2.7. sql_buffer_result

这个提示告诉优化器将结果存放在临时表中，并且尽快释放掉表锁。

### 4.6. 触发器

MySQL 从 5.0.2 开始支持触发器的功能。触发器是与表有关的数据库对象，在满足定义条件时触发，并执行触发器中定义的语句集合。触发器的这种特性可以协助应用在数据库端确保数据的完成性。

#### 4.6.1. 触发器的语法规范

创建触发器的语句如下：

```
create trigger trigger_name trigger_time trigger_event
on tbl_name for each row trigger_stmt;
```

触发程序与命名为 tbl_name 的表相关。tbl_name 必须引用永久性表。不能将触发程序与 temporary 表或视图关联起来。

trigger_time 是触发程序的动作时间。它可以是 before 或 after，以指明触发程序是在激活它的语句之前或之后触发。

trigger_event 指明了激活触发程序的语句的类型。trigger_event 可以是下述值之一：

1. insert：将新行插入表时激活触发程序，例如，通过 insert、load data 和 replace 语句。
2. update：更改某一行时激活触发程序，例如，通过 update 语句。
3. delete：从表中删除某一行时激活触发程序，例如，通过 delete 和 replace 语句。

对同一个表相同触发时间的相同触发事件，只能定义一个触发器。

在触发器中使用别名 new 和 old 来引用触发器中发生变化的记录内容。使用 old 和 new 关键字，能够访问受触发程序影响的行中的列(old 和 new 不区分大小写)。在 insert 触发程序中，仅能使用 new.col_name，没有

旧行。在 delete 触发程序中，仅能使用 old.col_name，没有新行。在 update 触发程序中，可以使用 old.col_name 来引用更新前的某一行的列，也能使用 new.col_name 来引用更新后的行中的列。

现在的触发器还只支持行级触发，不支持语句级触发。

另外在触发程序的执行过程中，MySQL 处理错误的方式如下：

1. 如果 before 触发程序失败，不执行相应行上的操作。
2. 仅当 before 触发程序(如果有的话)和行操作均已成功执行，才执行 after 触发程序。
3. 如果在 before 或 after 触发程序的执行过程中出现错误，将导致调用触发程序的整个语句的失败。

对于事务性表，如果触发程序失败(以及由此导致的整个语句的失败)，该语句所执行的所有更改将回滚。对于非事务性表，不能执行这类回滚，因而，即使语句失败，失败之前所作的任何更改依然有效。

应用场景：张三消费时，同时更新账号表的信息。

```
delimiter $$
create trigger tr_upd_acc after insert
on consume
for  each row begin
      update account set balance=balance-new.expenditure;
end $$
delimiter ;
```

创建该触发器后，当账号有消费记录时，即消息记录表 consume 表插入一行后，将触发该触发器，会对账号表的余额进行更新。这时能达到对账号余额的控制。

注：创建触发器需要 super 权限。激活的触发器，对于触发程序引用的所有 old 和 new 列，需要具有 select 权限，对于作为 set 赋值目标的所有 new 列，需要具有 update 权限。其他操作也类似。

在一个命名空间中，触发器的名字必须是唯一的，对于能够创建的触发程序的类型还存在其他限制。尤其是，对于具有相同触发时间和触发事件的表，不能有 2 个触发程序。例如：不能定义两个 after update 的触发器。当然这样也是无意义的。需要考虑总和需求。

删除触发器的语法如下：

```
drop trigger [schema_name.]trigger_name;
```

注：删除触发器也需要 super 权限。

## 4.6.2. 触发器的注意事项

MySQL 的触发器在使用时有许多限制，主要的限制包括：

1. 触发器只能用于永久表，不能用于临时表或者视图；
2. 触发器不能调用存储过程；
3. 触发器不能使用事务控制相关的语句，包括隐含 comimit 或者 rollback 的语句；
4. 在 before 触发程序中，auto_increment 列的 new 值为 0，不是实际插入新记录时将自动生成的序列号；
5. 函数、过程或者触发器无法调用 lock table、load data、load table、check table、flush 等相关语句；
6. 触发器无法使用返回数据集的 select 函数，但可以使用 select…into，也可以使用光标及 fetch 语句；
7. 由于主库上触发器 SQL 不会发送到从库上，主从库均可根据应用具体需求创建触发器。

## 4.6.3. 何时不用触发器

由于触发器的如上的几个限制问题，可能会要求触发器改为使用应用程序来实现。

触发器功能强大，轻松可靠地实现许多复杂的功能。但是写触发器时，应特别小心。因为一个触发器错误会导致引发该触发器的插入/删除/更新语句失败。因此业务上有其它需求的需要考虑在应用程序上去实现。

同时一个触发器的动作可以引发另外一个触发器。数据库系统通常都会限制触发器链的长度，把更长的触发器链看成为一个错误。因此业务上有较多级关系时也需要考虑使用应用程序来实现。而且程序中触发器不可滥用，否则会造成应用程序的维护困难。

## 4.7. 存储过程和自定义函数

MySQL 从 5.0 版本开始支持存储过程。存储程序是被存储在服务器中的一段 SQL 语句的集合，调用存储过程可以简化应用开发人员的很多工作，减少数据在数据库和应用服务器之间的传输，对于提高数据处理的效率是很有好处的。

### 4.7.1. 语法

#### 4.7.1.1. 如何使用存储过程和自定义函数

1. 创建存储过程和自定义函数

```
create procedure sp_name ([proc_parameter[,...]])
    [characteristic ...] routine_body
 create function sp_name ([func_parameter[,...]])
    returns type
    [characteristic ...] routine_body
     proc_parameter:
    [ in | out | inout ] param_name type
     func_parameter:
    param_name type
 type:
    any valid mysql data type
 characteristic:
    language sql
  | [not] deterministic
  | { contains sql | no sql | reads sql data | modifies sql data }
  | sql security { definer | invoker }
  | comment 'string'
 routine_body:
    valid sql procedure statement or statements
```

2. 改存储过程和自定义函数

```
alter {procedure | function} sp_name [characteristic ...]
 characteristic:
    { contains sql | no sql | reads sql data | modifies sql data }
  | sql security { definer | invoker }
  | comment 'string'
```

3. 删除存储过程和自定义函数

```
drop {procedure | function} [if exists] sp_name
```

4. 调用存储过程和自定义函数

```
call sp_name([parameter[,...]])
```

5. call 语句调用一个先前用 create procedure 创建的程序。
6. call 语句可以用 声明为 out 或 inout 参数的参数给它的调用者传回值。它也“返回”受影响的行数，客户端程序可以在 SQL 级别通过调用 row_count()函数获得这个值。
7. 查看存储过程
8. 使用 show procedure status 或者 information_schema 可以查看 procedure 的信息，show create procedure 可以查看 procedure 的定义信息。

```
show create procedure sp_name;
```

9. 它返回一个可用来重新创建已命名存储过程的确切字符串。

```
show procedure status [like 'pattern'];
```

10. 存储过程的特征，如数据库，名字，类型，创建者及创建和修改日期。如果没有指定样式，根据你使用的语句，所有存储程序的信息都被列出。

#### 4.7.1.2. 参数

存储过程的参数分为 in、out、inout 三种。

#### 4.7.1.3. 变量

1. 声明变量

```
declare var_name[,...] type [default value]
```

2. 给变量赋值
3. 方式一：set var_name = expr [, var_name = expr] ...
4. 方式二：select col_name[,...] into var_name[,...] table_expr
5. 注意：用户变量名在 MySQL 5.1 中是对大小写不敏感的。同时 SQL 变量名不能和列名一样。如果 select ... into 这样的 SQL 语句包含一个对列的参考，并包含一个与列相同名字的局部变量，MySQL 当前把参考解释为一个变量的名字。

#### 4.7.1.4. 流控制语句

可以使用 if、case、loop、leave、iterate、repeat 以及 while 语句进行流程控制，下面逐一介绍。

1. if 语句

```
if search_condition then statement_list
    [elseif search_condition then statement_list] ...
    [else statement_list]
end if;
```

2. 如果 *search_condition* 求值为真，相应的 SQL 语句列表被执行。如果没有 *search_condition* 匹配，在 else 子句里的语句列表被执行。
3. case 语句

```
case case_value
    when when_value then statement_list
    [when when_value then statement_list] ...
```

```
        [else statement_list]
    end case;
    or:
    case
       when search_condition then statement_list
       [when search_condition then statement_list] ...
       [else statement_list]
    end case;
```

4. 存储程序的 case 语句实现一个复杂的条件构造。如果 search_condition 求值为真，相应的 SQL 被执行。如果没有搜索条件匹配，在 else 子句里的语句被执行。
5. loop 语句

```
    [begin_label:] loop
    statement_list
    end loop [end_label]
```

6. loop 允许某特定语句或语句群的重复执行，实现一个简单的循环构造。在循环内的语句一直重复直到循环被退出，退出通常伴随着一个 leave 语句。
7. loop 语句可以被标注。除非 begin_label 存在，否则 end_label 不能被给出，并且如果两者都出现，它们必须是同样的。
8. leave 语句

```
    leave label;
```

9. 这个语句被用来退出任何被标注的流程控制构造。它和 begin ... end 或循环一起被使用。
10. iterate 语句

```
    iterate label;
```

11. iterate 只可以出现在 loop，repeat，和 while 语句内。iterate 意思为：“再次循环。”
12. repeate 语句

```
    [begin_label:] repeat
    statement_list
    until search_condition
    end repeat [end_label]
```

13. repeat 语句内的语句或语句群被重复，直至 search_condition 为真。
14. repeat 语句可以被标注。 除非 begin_label 也存在，end_label 才能被用，如果两者都存在，它们必须是一样的。
15. while 语句

```
    [begin_label:] while search_condition do
     statement_list
    end while [end_label]
```

16. while 语句内的语句或语句群被重复，直至 search_condition 为真。
17. while 语句可以被标注。 除非 begin_label 也存在，end_label 才能被用，如果两者都存在，它们必须是一样的。

#### 4.7.1.5. 光标

MySQL 提供了对服务器端光标的支持，在存储过程和函数中可以使用光标对结果集进行循环处理。但光标的支持还又很多限制：只读(read only)、不支持回溯(not scrollable)、asensitive。

1. 声明光标

```
declare cursor_name cursor for select_statement;
```

这个语句声明一个光标。也可以在子程序中定义多个光标，但是一个块中的每一个光标必须有唯一的名字。注意 select 不能有 into 语句。

2. open 光标

```
open cursor_name;
```

3. fentch 光标

```
fetch cursor_name into var_name [, var_name] ...;
```

这个语句用指定的打开光标读取下一行(如果有下一行的话)，并且前进光标指针。

4. 关闭光标

```
close cursor_name;
```

关闭打开的光标。如果未被明确地关闭，光标在它被声明的复合语句的末尾被关闭。

#### 4.7.1.6. 条件和处理程序

```
declare handler_type handler for condition_value[,...] sp_statement
 handler_type:
    continue
  | exit
  | undo
 condition_value:
    sqlstate [value] sqlstate_value
  | condition_name
  | sqlwarning
  | not found
  | sqlexception
  | mysql_error_code
```

这个语句指定每个可以处理一个或多个条件的处理程序。如果产生一个或多个条件，指定的语句被执行。

对一个 continue 处理程序，当前子程序的执行在执行 处理程序语句之后继续。对于 exit 处理程序，当前 begin...end 复合语句的执行被终止。undo 处理程序类型语句还不被支持。

sqlwarning 是对所有以 01 开头的 sqlstate 代码的速记。

not found 是对所有以 02 开头的 sqlstate 代码的速记。

sqlexception 是对所有没有被 sqlwarning 或 not found 捕获的 sqlstate 代码的速记。

除了 sqlstate 值，MySQL 错误代码也不被支持。

### 4.7.2. 存储过程和自定义函数的注意事项

和触发器类似，存储过程和自定义函数在使用中也有一些注意事项，具体如下：

1. MySQL 记录每个发生在存储程序和函数里的 DML 事件，并复制这些单独的行为到从服务器。除 DML 外的存储程序和函数中的语句不会被复制。
2. 注意自定义函数和触发器不是这样的。主服务器不会记录自定义函数和触发器里的 DML 事件，会记录自定义函数的调用，而触发器是在从服务上执行到触发条件时会被触发。
3. 存储过程和自定义函数禁用 check tables、lock tables、unlock tables、load data、 load table、flush 语句以及 SQL 预处理语句(prepare、execute、deallocate prepare)。在 5.0.13 中放宽了限制，可以在存储过程中使用，但是在用户自定义函数中还是不可以使用。

4. 对于自定义函数额外的限制：禁止执行显式或隐式提交或回滚操作的语句以及返回结果集的语句。

### 4.7.3. 存储过程优势

存储过程主要是把一组 SQL 语句和控制语句组成起来，然后封装在一起的过程，它驻留在数据库中，可以被客户应用程序调用，也可以从另一个存储过程或触发器调用。

它实现的功能也可以在应用程序中实现。但是和应用程序相比，存储过程有如下优势：

1. 模块化程序设计：只需一次性编写然后存储在数据库中，以后就可以在应用程序中调用存储过程。
2. 方便的管理与部署： 如果使用应用程序来实现的某业务功能变更时，就需要重新编码，编译，以及重新部署在服务器上。而用存储过程方式实现，就不需要对应用程序进行重编译，直接修改存储过程就可以。
3. 提供了更安全的实现机制：通过对执行某一存储过程的权限进行限制，从而能够实现对相应的数据访问权限的限制，避免非授权用户对数据的访问，保证数据的安全。

## 4.8. 应用与 SQL 设计

人们通常认为改写查询是优化数据库访问的最终办法。尽管如此，它并不是最值得关注和最高效的方法。

### 4.8.1. 应用需求与 SQL 设计

应用需求的合理性在应用设计中很容易被忽略，但是一个不合理的需求可能在整个系统中是画蛇添足的，甚至带来资源的浪费。同时需求是否合理并不是很容易界定，尤其是对纯技术人员来说。因此在应用需求的提出，设计，开发，后期的维护中，需要评估项目的一个投入产出比率。在立项之初需要判断该功能的预期投入产出比率是否合理，判定项目是否有必要进行。同时也可在上线后，在有了实际的投入产出比后，拿真实数据来说话，让一些不合理的功能应该撤下来。

案例分析：

点击到用户的账户页面时，显示该用户的每个账户的消费次数(要求实时更新)。

首先觉得这个功能非常容易实现。执行一条 `select count(*) from consume where account_id=1;` 如果存放消费记录的表已经有上千万的帖子，执行该条 qeury 的成本就比较高，同时当并发量较大时，这个应该不是一个简单的事情。

为实现该功能，可能会想到再添加一张表，在这个表中存放消费次数。每次有新的消费时，就增加 1。这个查询效率暂时能满足查询需求，如果随着账户增多，以及高峰期每秒几十或者上百次的消费记录产生。恐怕这个表也要成为噩梦了。

其实这个问题的关键不是实现这个问题的技术细节，而是这个功能带来的收益。到底有多少人会关注消费次数，尤其是消费次数很大时。

同时随着系统不断升级也很容易造成系统复杂，从而影响系统性能。因此在系统升级中也需要注意投入产出比率。尤其需要注意升级对主要的业务功能带来的影响。

另外还需要注意不要过于重视用户体验，大量非核心的业务消耗了过多的数据库资源。因此除了考虑投入产出比，还多注意核心业务和非核心业务在数据库资源上的消耗情况。

# 第 5 章. 附录

## 5.1. 例子库

### 5.1.1. 例子库的作用

为了更好的学习《百度 MySQL 数据库 SQL 编写指南》，为指南中 SQL 例子提供了网页式的访问平台。通过该平台，大家可以去测试编写指南中的具体的 SQL，加深对编写指南中内容的理解和学习。

### 5.1.2. 例子库地址

例子库的访问链接为 http://dba.baidu.com/topic/liziku/index.php。
访问的账号是：liziku，密码是 baidudba。

### 5.1.3. 例子库表说明

例子库中有一个 database，名称是 sql_exam。
库中有四张表，表结构如下：

1. customer 表：消费者信息表

| 字段名 | 类型 | 索引 | 默认值 | 其它说明 |
|---|---|---|---|---|
| customer_id | int(10) | pri | 自增 | |
| customer_name | varchar(40) | | null | |
| customer_contact | varchar(200) | | null | |
| customer_phone | varchar(20) | | null | |

2. account 表：账户信息表

| 字段名 | 类型 | 索引 | 默认值 | 其它说明 |
|---|---|---|---|---|
| account_id | int(10) | pri | 自增 | |
| account_number | int(10) | | null | |
| account_bank | varchar(100) | | null | (`account_bank`,`customer_email`)的复合索引 |
| Balance | decimal(10,2) | | null | |
| customer_email | varchar(100) | | null | |

3. cust_acct 表：消费者与账号的对应关系表

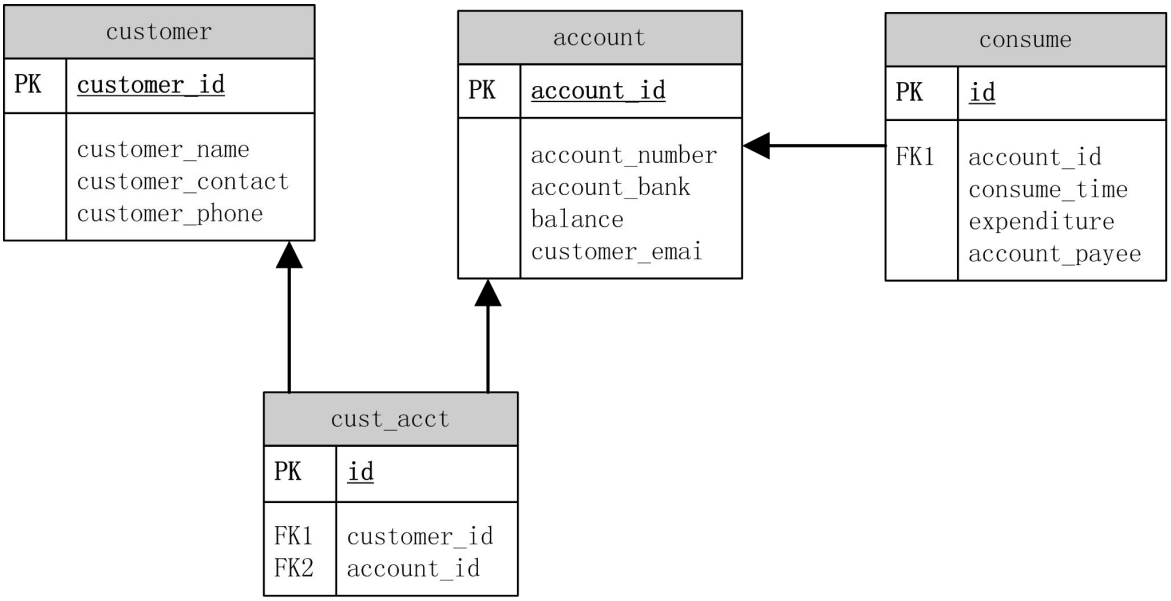
| 字段名 | 类型 | 索引 | 默认值 | 其它说明 |
|---|---|---|---|---|
| id | int(10) | pri | 自增 | |
| customer_id | int(10) | mul | null | |
| account_id | int(10) | mul | null | |

4. consume 表:消费记录表

| 字段名 | 类型 | 索引 | 默认值 | 其它说明 |
|---|---|---|---|---|
| id | int(10) | rri | 自增 | |
| account_id | int(10) | | | (`account_id`,`expenditure`,`consume_time`)的复合索引 |
| consume_time | timestamp | | current_timestamp | |

| | | | | |
|---|---|---|---|---|
| expenditure | decimal(10,2) | | | |
| account_payee | int(11) | | | (`account_payee`,`expenditure`)的复合索引 |

表间的关联关系图如下：




## 5.2. 后续说明

1. 从内核理解 SQL 优化的内容，会在 2011 年 Q4 修订版本中介绍。
2. 数据库设计方面知识，会在《数据库业务设计》文档中介绍，预计 2011 年 Q4 发布。