MySQL知识整理

# 前言介绍

MySQL其实就是一个文件管理系统，将我们的数据按照一定的规则存储在计算机硬盘上。这里简单介绍一下在数据库中相关名词的含义。后文中遇到将不再做解释。

## Schema

直译为概要，计划，图标。用数据库术语来说是指模式  ，是数据库的组织和结构模式包含模式对象，可以是 表， 列，数据类型，视图，存储过程，关系，主键，外键等。是数据库对象的集合。在MySQL中*Schema*和*数据库*是同一件事。

## DDL

Data Definition Language，常见的创建语句有：

CREATE DATABASE

### 创建数据库

CREATE {DATABASE | SCHEMA} db\_name

[create\_specification [, create\_specification] ...]

create\_specification:

[DEFAULT] CHARACTER SET charset\_name

| [DEFAULT] COLLATE collation\_name

### 创建数据库表格

CREATE TABLE

创建数据库表格

CREATE [TEMPORARY] TABLE tbl\_name

[(create\_definition,...)]

[table\_options] [select\_statement]

### 修改数据库表名

ALTER TABLE tbl\_name

alter\_specification [, alter\_specification] ...

alter\_specification:

ADD [COLUMN] column\_definition [FIRST | AFTER col\_name ]

| ADD [COLUMN] (column\_definition,...)

| ADD INDEX (index\_col\_name,...)

| ADD [CONSTRAINT [symbol]]

PRIMARY KEY (index\_col\_name,...)

| ADD [CONSTRAINT [symbol]]

UNIQUE (index\_col\_name,...)

| ADD (index\_col\_name,...)

| ADD [CONSTRAINT [symbol]]

FOREIGN KEY (index\_col\_name,...)

[reference\_definition]

| ALTER [COLUMN] col\_name {SET DEFAULT literal | DROP DEFAULT}

| CHANGE [COLUMN] old\_col\_name column\_definition

[FIRST|AFTER col\_name]

| MODIFY [COLUMN] column\_definition [FIRST | AFTER col\_name]

| DROP [COLUMN] col\_name

| DROP PRIMARY KEY

| DROP INDEX index\_name

| DROP FOREIGN KEY fk\_symbol

| DISABLE KEYS

| ENABLE KEYS

| RENAME [TO] new\_tbl\_name

| ORDER BY col\_name

| CONVERT TO CHARACTER SET charset\_name [COLLATE collation\_name]

| [DEFAULT] CHARACTER SET charset\_name [COLLATE collation\_name]

| DISCARD TABLESPACE

| IMPORT TABLESPACE

| table\_options

### 删除数据库表格

DROP TABLE

删除数据库表格

DROP [TEMPORARY] TABLE

tbl\_name [, tbl\_name] ...

[RESTRICT | CASCADE]

### 创建视图

CREATE VIEW

创建查询命令

CREATE[ORREPLACE][ALGORITHM={UNDEFINED|MERGE|TEMPTABLE}]

VIEW view\_name [(column\_list)]

AS select\_statement

[WITH [CASCADED | LOCAL] CHECK OPTION]

### 修改视图

ALTER VIEW

修改查询命令

ALTER [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]

VIEW view\_name [(column\_list)]

AS select\_statement

[WITH [CASCADED | LOCAL] CHECK OPTION]

### 删除视图

DROP VIEW

删除查询命令

DROP VIEW

view\_name [, view\_name] ...

[RESTRICT | CASCADE]

### 删除表格内容

TRUNCATE TABLE

删除数据表内容

TRUNCATE TABLE name [DROP/REUSE STORAGE]

DROP STORAGE：显式指明释放数据表和索引的空间

REUSE STORAGE：显式指明不释放数据表和索引的空间

## DML

DML 操作是指对数据库中表记录的操作，主要包括表记录的插入（insert）、更新（update）、删除（delete）和查询（select）。

## ACID

* Atomicity（原子性）：一个事务（transaction）中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被恢复（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。
* Consistency（一致性）：在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的预设规则，这包含资料的精确度、串联性以及后续数据库可以自发性地完成预定的工作。
* Isolation（隔离性）：数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。事务隔离分为不同级别，包括读未提交（Read uncommitted）、读提交（read committed）、可重复读（repeatable read）和串行化（Serializable）。
* Durability（持久性）：事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

# Schema与数据类型

## 数据类型要合适

一般情况下，应该尽量使用可以正确存储数据类型的最小数据类型。更小的数据类型占用的空间更少，意味着占用更少的磁盘，CPU，内存和缓存，CPU处理周期更少

## 简单就好

简单数据类型的操作通常需要更少的CPU，例如：整型比字符串的代价更低，因为字符集和校对规则（排序规则）使字符串的比较比整型比较更复杂。

## 尽量避免null

很多表都包含null的列，即使应用程序并不需要保存null也是如此，这可能是因为null是列的默认属性。通常情况下最好指定列为 not null，除非真的需要存储null。如果查询中包含null，对MySQL来说更难优化，因为null列将使索引，索引统计和值的比较变得更复杂。可为null的列会使用更多的存储空间，在MySQL里也需要特殊处理。当可为null的列被索引时，每个索引记录需要一个额外的字节。在MyISAM里甚至可能导致固定大小的索引。

通常把可为null的列改为not null带来的性能提升比较小。调优时没有必要首先在现有的schema中查找并修改这些情况，除非确定是这个导致的性能问题。

但是计划在建索引和创建表的时候，应为避免设计成可为null的列。

## 数据类型

### 整数类型

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 占用位 | 范围 | 备注 |
| TINYINT | 8 | -2N-1~2N-1-1 | N等于存储空间位数 |
| SMALLINT | 16 |
| MEDIUMINT | 24 |
| INT | 32 |
| BIGINT | 64 |

整数类型有unsigned的可选属性，表示不允许负数。这大致可以使正数的上线提高一倍。

注意：整数计算一般使用64位的BIGINT整数，即使这是在32位中的环境也是如此。（部分聚合函数例外。他们使用decimal或者double）

声明的时候可以指定宽度，比如int（11），对于大多数应用没有意义：这不会限制值的合法范围，只是规定MySQL的一些交互工具用来显示的字符个数。存于存储和计算来说int（1）和int（20）相等。

### 实数类型

实数是带有小数部分的数字。不只是为了存储小数部分，也可以使用decimal存储比bigint还大的整数。

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 区别 | 声明 | 含义 |
| Float | 单精度浮点数值 | FLOAT(L, D) | length长度表示全部位数（小数点后都要算进去），Decimals表示小数点位数，当存储的数值小数部分>D,则进行四舍五入 |
| Double | 双精度浮点数值 | DOUBLE(L, D) |
| Decimal | 精确的数值，数字总长度最大为65 | DECIMAL(L, D) |

单精度存储浮点类型的话 就是 ==4x8=32位的长度== , 所以float单精度浮点数在内存中占 4 个字节，并且用 32 位二进制进行描述

双精度存储浮点类型就是 ==8x8 =64位的长度==, 所以double双精度浮点数在内存中占 8 个字节，并且用 64 位二进制进行描述 通过计算、那么64位就可以获得更多的尾数!

实数在进行保存的时候会进行四舍五入的运算。对于**浮点数**来说，小数位超过设定的值，按四舍五入保存，小数位全为0则舍弃。比如定义一列为float(5,2)，

|  |  |
| --- | --- |
| 插入值 | 结果 |
| 90.012 | 90.01 |
| 90.018 | 90.02 |
| 90.0 | 90 |

**注意：**定点数与浮点数的区别是能存储的值的小数点位置是否是固定的。

对于定点数来说，定义数据库列为DECIMAL(5,2)

|  |  |
| --- | --- |
| 插入值 | 结果 |
| 1.2345 | 小数点后最多2位，所以保存可以，自动四舍五入数据截断,但会报waning |
| 12.34 | OK |
| 1234.5 | 因为小数部分未满2位，要补0.所以保存应该1234.50。所以整个位数超出了5，保存报错。 |
| 1.2 | 小数未满部分补0。按照1.20保存。 |

### 字符串类型

|  |  |  |
| --- | --- | --- |
| 类型 | 作用 | 备注 |
| Varchar | 用于存储可变长的字符串，最常见的字符串数据类型 | 比char更节省空间，仅使用必需的空间；  需要使用1或2个额外字节记录字符串长度，取决于列的最大长度是否＞255 |
| Char | 定长 | 对于非常短的列，效果比varchar好 |
| blob | 二进制数据，没有排序规则或者字符集 | 有自己的数据类型家族，tinyXX，smallXX，XX，mediumXX，longXX；  MySQL把这两种类型的的值当成一个独立的对象处理，值太大时会专门使用外部的存储区域进行存储。 |
| Text | 有字符集和排序规则 |

有时候可以使用枚举来代替常用的字符串类型。枚举列可以把一些不重复的字符串存储成一个预定义的集合。MySQL存储枚举时非常紧凑，会根据列表值的数量压缩到一个或两个字节中。

内部使用的是整型来存储枚举

### 日期和时间类型

|  |  |  |
| --- | --- | --- |
| 类型 | 范围 | 备注 |
| Datetime | 1001~9999年，格式YYYYMMDDHHMMSS | 使用8个字节存储 |
| Timestamp |  | 使用从1970年1月1日午夜到现在的秒数，只使用4个字节 |

### 位数据类型

|  |  |  |
| --- | --- | --- |
| 类型 | 使用 | 备注 |
| Bit | 用来存储bit值，BIT(M)代表可以存储M个bit，M的取值范围为1到64 | 谨慎使用 |
| Set | 设定set的格式：  字段名称 SET("选项1","选项2",...,'选项n')，最多64个选项 | 如果需要保存很多的true/false值，可以考虑合并这些列到一个set数据类型；  缺点是如果表很大，改变表的结构代价较高； |

### 选择标识符（identifier）

为标识列（identifier column）选择合适的数据类型非常重要。一般来说更有可能用标识列与其他值进行比较，或者通过此列来寻找其他列。所以选择合适的类型将影响精确匹配的效率

推荐使用整数类型，整数类型非常快而且可以使用自增。

不推荐枚举和set类型，这两种数据类型只适合存储固定的信息。

尽量避免字符串类型，很耗费空间，比数字慢

## Schema设计中陷阱

这里列举一些MySQL的schema问题

|  |  |  |
| --- | --- | --- |
| 问题描述 | 备注 | 建议 |
| 太多的列 | MySQL存储引擎API工作时需要在服务层和存储引擎层之间通过行缓冲格式拷贝数据，然后在服务层将缓存内容解码成列，这个过程操作代价非常高 | 垂直切割，拆分表 |
| 太多的关联 | 实体-属性-值 这种设计模式可能会让表超过关联限制 | 限制表的关联，最好单个查询限制在12个表内关联查询 |
| 全能枚举 | 枚举的类型太多 | 应该用整数作为外键关联字典表或者查找表 |
| 变相枚举 | 数据类型定义为set，实际使用的情况下集合的值是互斥的 | 改为枚举 |
| 替代null | 为了避免使用null，使用了常数替代，但是可能造成业务代码复杂度 | 根据实际情况使用null |

## 范式与反范式

### 第一范式，无重复列

数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。如果出现重复的属性，就可能需要定义一个新的实体，新的实体由重复的属性构成，新实体与原实体之间为一对多关系。在第一范式（1NF）中表的每一行只包含一个实例的信息。简而言之，第一范式就是无重复的列。

### 第二范式，属性完全依赖于主键

如果关系模式R为第一范式，并且R中每一个非主属性完全函数依赖于R的某个候选键， 则称为第二范式模式。

### 第三范式，属性不依赖于其它非主属性

如果关系模式R是第二范式，且每个非主属性都不传递依赖于R的候选键，则称R为第三范式模式。

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

### 范式的优点和缺点

因为以下原因，范式化通常能够带来好处：

* 范式化的更新操作通常比反范式更快
* 当数据较好的范式化的时候，就只有很少的数据或者没有重复数据，所以修改只有更少的数据。
* 范式化的表通常更少。可以更好的放在内存中。查询会更快
* 很少的多余数据意味着检索列表数据的时候更少需要distinct或者group by语句。

范式化设计的schema的缺点通常需要关联。稍微复杂的查询可能使得索引策略无效。

### 反范式的优点和缺点

反范式化的schema因为数据在一张表中，可以很好的避免关联查询。单独的表也能更好的使用索引策略。

### 总结

混用范式与反范式设计。完全遵守范式化和反范式化的设计只有实验室里面才有的东西。

## 缓存表与汇总表

上文提到，有时候提升性能最好的方法是在同一张表中保存衍生的冗余数据。但是有时候也需要创建一张完全独立的汇总表或者缓存表（特别是满足检索需求时）。

比如：统计某个文件的下载次数，网站的访问人数等等。可以创建一个表来存储文件下载次数、网站访问人数统计等信息。而不是每次去实时计算 count（列）

# 高性能索引

## 索引基础

去一栋楼里面的某个公司面试，怎么快速找到公司位置呢？当然先去大厅的楼层索引位置查询公司在哪个楼层。

在MySQL中，存储引擎用类似的方法使用索引。首先在索引中找到对应的值，然后根据匹配的索引记录找到对应的数据行。假设执行以下查询：SELECT description FROM main\_order WHERE order\_no = 1;

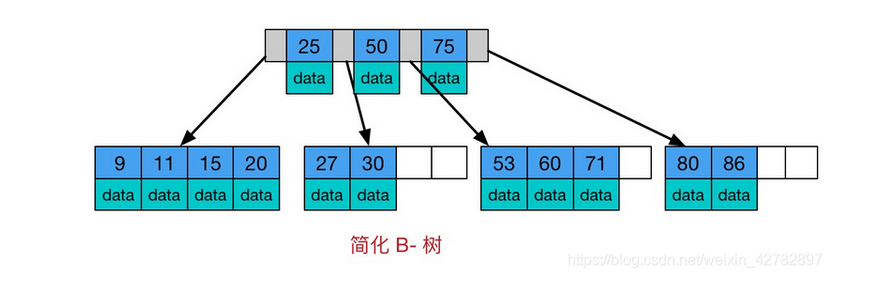
如果在order\_no列上创建有索引，则MySQL使用该索引找到order\_no为5的行。简而言之MySQL现在索引上按值查询，然后返回包含该值的数据行。

索引可以包含一个或多个列。如果索引包含多个列，则列的顺序也非常重要。MySQL只会高效的使用最左前缀列。

### MySQL支持的索引类型

#### B-tree

如果没有特别的指明类型，多半说的就是B-tree索引。它使用B-tree数据结构来存储数据。这是一种多路查找树。



**B-tree中，每个结点包含：**

1、本结点所含关键字的个数；

2、指向父结点的指针；

3、关键字；

4、指向子结点的指针；

**B-tree有以下特性：**

1、关键字集合分布在整棵树中；

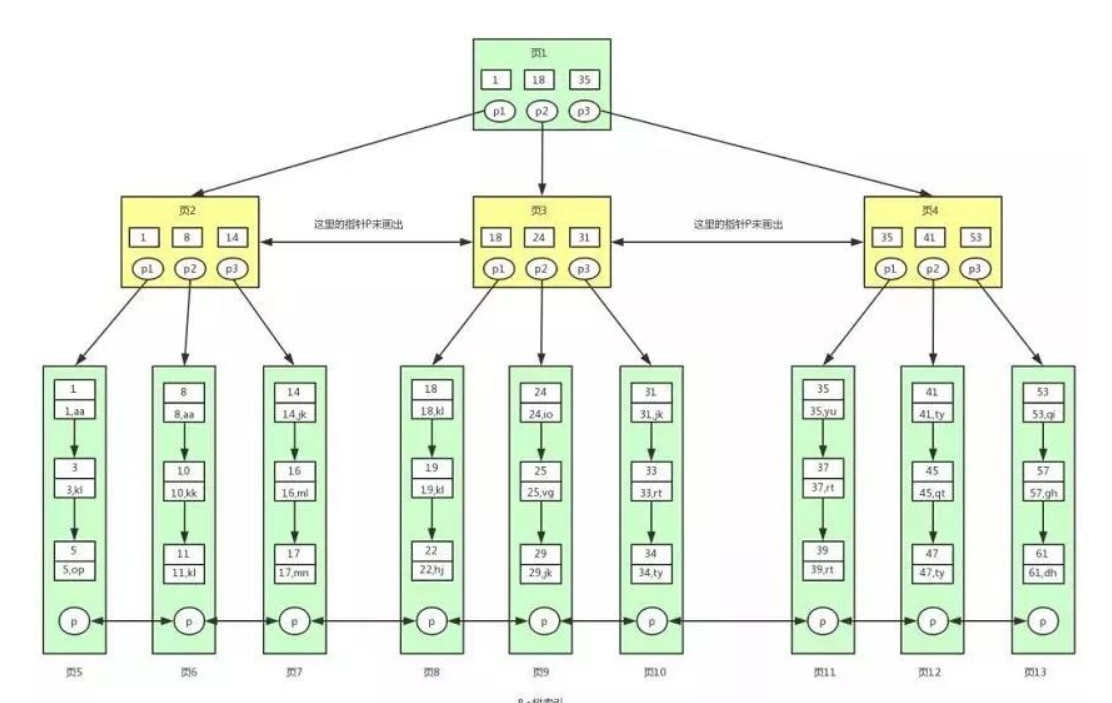
2、任何一个关键字出现且只出现在一个结点中；

3、搜索有可能在非叶子结点结束；

4、其搜索性能等价于在关键字全集内做一次二分查找；

5、自动层次控制；

**注意：**B+tree是B-tree的一个变种，不同点是数据放在了叶子节点，叶子节点之间有指针可以进行范围访问。



#### 哈希索引

哈希索引基于哈希表实现。只有精确匹配索引所有列的查询才有效。对于每一行数据，存储引擎会对所有的索引计算出一个哈希码。

哈希索引将所有的哈希码存储在索引中，同时哈希表中保存指向每个数据行的指针。由此可见，哈希索引也有了自己的限制：

* 哈希索引只包含哈希值和行指针，不存储字段值，所以不能用哈希索引来避免行读取。
* 哈希索引并不是按照顺序来存储，所以不能用来排序
* 不支持部分索引查询
* 只支持等值的比较，比如 = ，in（），<=>，也不支持范围查询
* 访问哈希索引的数据比较快，除非哈希冲突非常多
* 如果哈希冲突特别多，索引的维护代价也非常高。

在MySQL中，只有memory引擎显示的支持哈希索引。同时也支持Btree索引。

## 索引的优点

* 索引极大的减少了服务器需要扫描的数据量
* 索引可以帮助服务器避免排序和临时表
* 索引可以将随机I/O变为顺序I/O

## 高性能索引策略

### 独立的列

指索引列不能是表达式的一部分，也不能是函数的参数

比如 where id+1 = 5；即使id上建了索引查询也不会走索引。

### 前缀索引和索引选择性

索引的选择性是指：不重复的索引值和数据表的记录总数的比值。越高则查询效率越高。

前缀索引是指：索引列的字符串太长，这会让索引非常大且慢。优化方式就是索引开始的部分字段。

创建前缀索引的方式：alter table table\_name add key(column\_name(legnth));

Length是指前缀索引的长度。

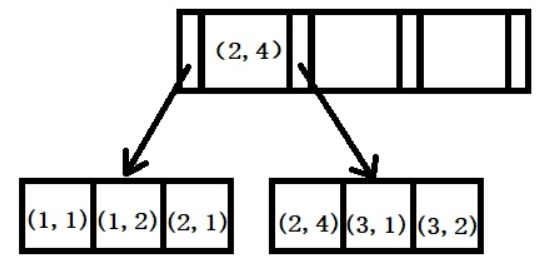
前缀索引是一种能让索引变得更小，更快的有效方法。但是不能用前缀索引进行order by和group by的操作，也无法用前缀索引进行覆盖扫描。

### 选择合适的索引顺序

在一个多列的Btree索引中，索引列的顺序表示索引首先按照最左列进行排序，其次是第二列，等等。所以索引可以按照升序/降序进行扫描，以满足order by、group by、distinct等查询要求。

MySQL建立联合索引时，参考索引底层原理，索引的底层是一颗Btree，联合索引也是一样，由于构建一颗Btree只能根据一个值来确定索引关系，所以数据库依赖联合索引最左的字段来匹配。

举例：创建一个（a,b）的联合索引，那么它的索引树就是下图的样子



可以看到a的值是有顺序的，1，1，2，2，3，3，而b的值是没有顺序的1，2，1，4，1，2。但是我们又可发现a在等值的情况下，b值又是按顺序排列的，但是这种顺序是相对的。

创建这个联合索引的时候，相当于创建了索引列(a),(a,b)。

同时应当注意索引的列的顺序，

### 聚簇索引

聚簇索引并不是一种单独的索引类型，而是一种数据存储方式。具体依赖于实现方式。

在InnoDb上，聚簇索引实际上是在同一个结构中保存了Btree索引和数据行。当表有聚簇索引时，数据行实际上放在了索引的叶子节点中。

对比非聚簇索引，只需要查询一次。这是因为辅助索引的保存的“行指针”保存的是行的主键值。

### 覆盖索引

索引包含需要的查询的字段值，就称之为覆盖索引。

### 使用索引扫描做排序

只有当索引的列顺序和order by字句的顺序完全一致，并且所有的列的排序方向都一样时，MySQL就可以使用索引对结果进行排序。

如果查询需要关联多张表，只有order by字句引用的字段全部为第一张表时，才能使用索引排序。

Order by字句和查询的限制是一样的：需要满足索引最左匹配要求

### 冗余和重复的索引

重复索引是指相同的列上按照相同的顺序创建相同类型的索引。应该避免这种情况

冗余索引是指：创建了索引(a,b),再去创建索引(a)就是冗余索引。这只是索引（a,b）的前缀索引。

当扩展索引会让索引变得非常大，影响性能时，需要冗余索引。比如；扩展一个varchar的列到索引中

### 未使用的索引

完全用不到的索引叫未使用的索引，建议直接删除

### 索引和锁

索引可以让查询锁定更少的行。

## 维护索引和表

常用方法：找到并修复损坏的表；更新索引统计信息；减少索引和数据的碎片

# 查询性能优化

## 查询为什么变慢

需要搞清楚的一点是：真正重要的是响应时间。如果把查询看做是一个任务，那么他是由一系列子任务构成的，每个任务都需要消耗一点时间。优化查询其实就是优化其子任务。要么消除其中一些子任务，要么减少子任务执行次数，要么让子任务执行更快。

需要了解查询的生命周期，清楚查询的时间消耗情况对于查询优化有很大的意义。

## 慢查询基础

查询性能低下的最基础原因就是访问的数据太多：

* 分析应用程序是否检索了大量超过需要的数据（比如多表关联返回全部的列；总是取出全部列，重复查询相同的数据）
* 分析MySQL服务层是否存在分析大量超过需要的行，在确定只返回需要的列后，需要查看MySQL是否扫描了过多的数据，大致从三方面分析：响应时间、扫描的行数、返回的行数。

## 重构查询方式

设计查询的时候需要考虑的一个问题是：是否需要将一个复杂的查询分成多个简单的查询。在目前的计算机/网络带宽能力下，运行多个小查询已经不是问题。

MySQL内部每秒能扫描内存中上百万行数据，响应数据给客户端要慢得多。

### 切分查询

有时候需要将一个大的查询切分为好几次查询，比如删除。删除旧数据，一个大的语句一次执行可能需要锁住很多数据，占满整个事务日志，阻塞很多小的但是非常重要的查询。这时候需要将一个删除语句拆分为多个小删除

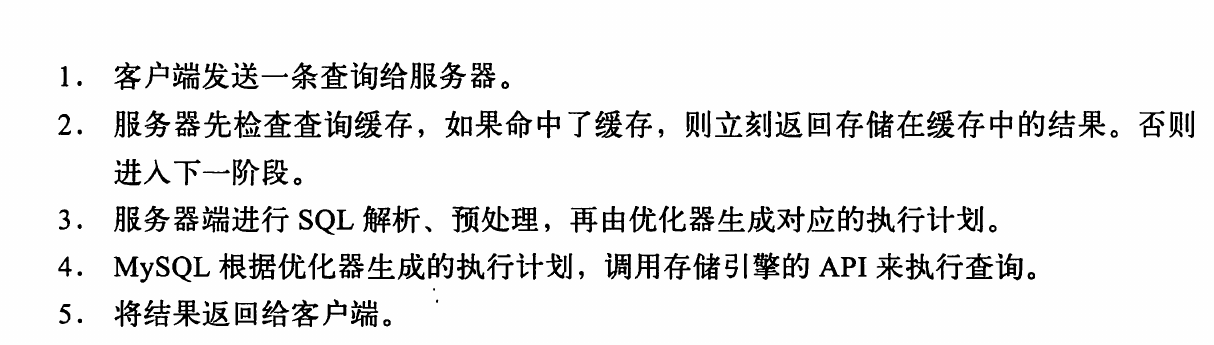
### 分解关联语句

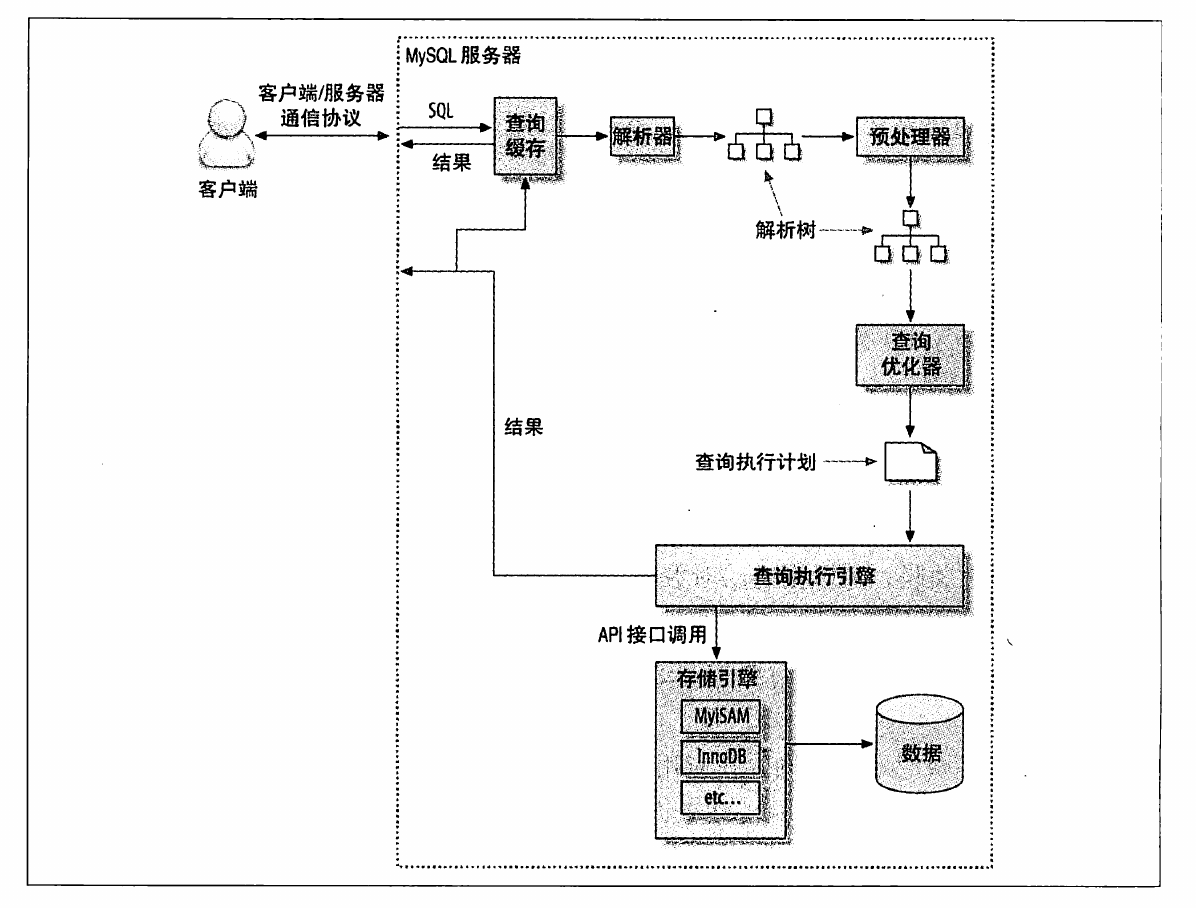
很多高性能应用都会对关联查询进行分解。简单的，可以对每一个表进行一次单表查询，然后在应用程序中进行结果关联。

这样的一个好处是让缓存效率更高；执行单个查询减少锁的竞争；应用层做关联，更容易对数据库进行拆分；减少冗余记录查询（数据库中关联查询，可能重复的访问一部分数据）

## 查询执行基础

执行一个查询时，MySQL的动作：



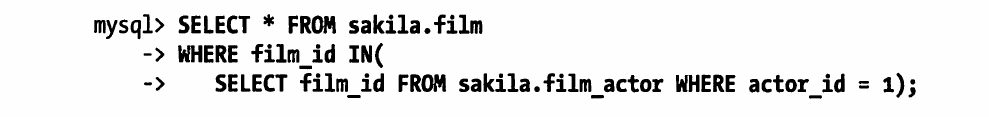


## 查询优化器的局限性

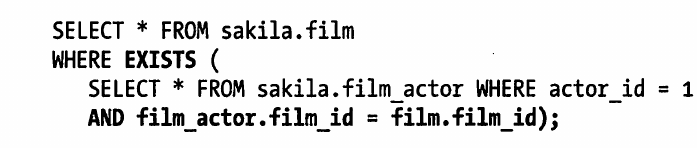
在以下情况下，优化器可能负优化

### 子关联查询

MySQL子查询实现比较糟糕。最坏的一类查询时where 条件中包含in的子查询。比如：

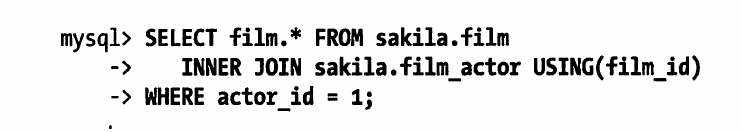


Mysql会将相关联的外层表压到子查询中



如果外表的数据比较大，性能就比较糟糕了。

修改为：



### Union的限制

如果希望union的各个字句能够根据limit只取部分结果，或者希望先排序再合并，需要在各个字句中分别使用。

### 等值传递

等值传递又叫等值连接。比如4.5.1中的查询例子就是等值传递，会将in（）列表复制应用到各个关联的表中，如果表特别大，就会很慢了。

### 并行执行

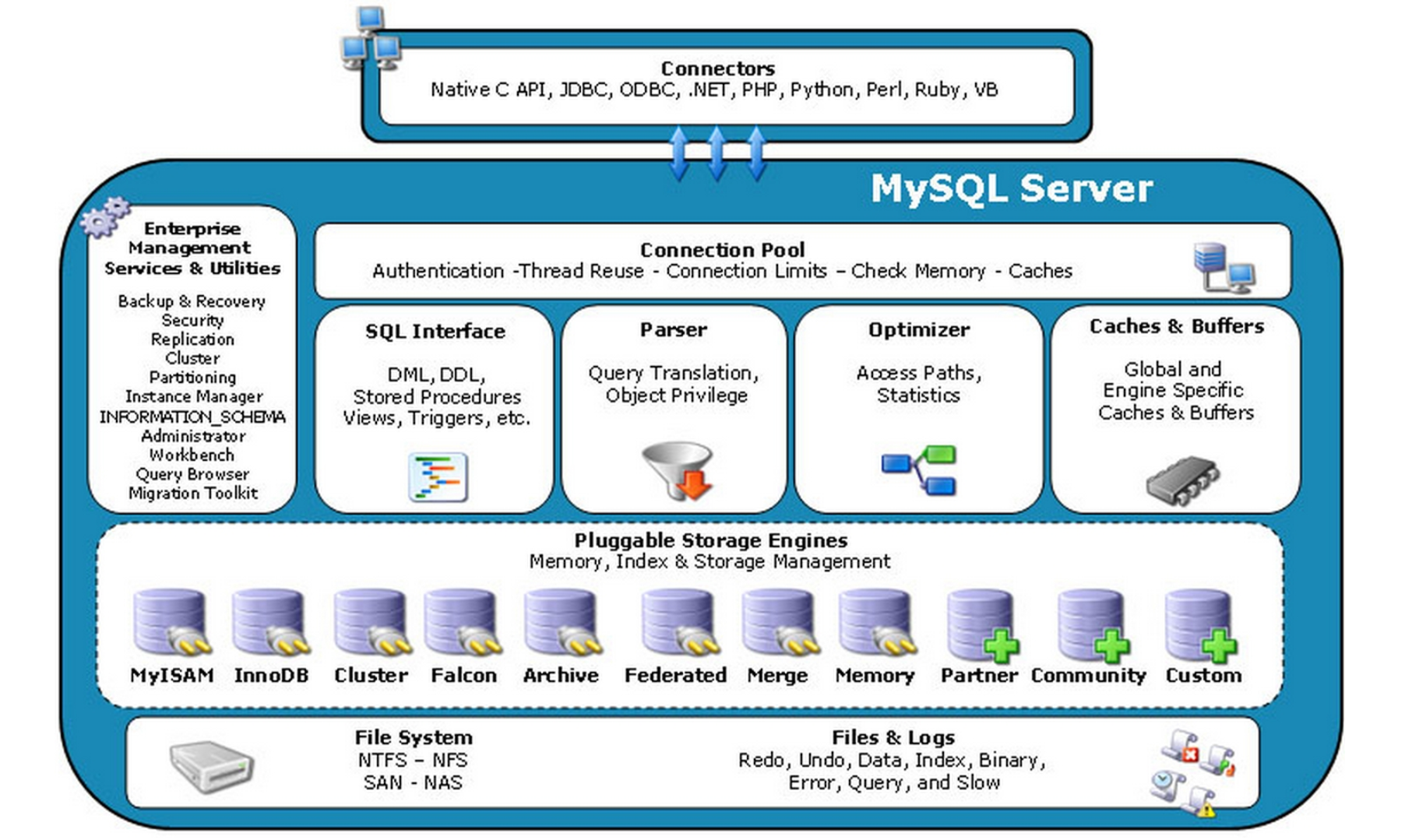
MySQL无法利用多核特性来并行查询。

## 优化特定类型查询

### 优化count()

如果精度不要求那么高，可以使用近似值；高精度要求需要改变应用的架构，增加汇总表或者外部缓存系统等等

# MySQL架构



MySQL 在大体上可以分为 Server 层和存储引擎层两部分

## Server 层

Server 层包括连接器、查询缓存、分析器、优化器、执行器等，涵盖 MySQL 的大多数核心服务功能。以及所有的内置函数（如日期、时间、数学和加密函数等）

所有跨存储引擎的功能都在这一层实现，比如存储过程、触发器、视图等。

## 存储引擎层

存储引擎层负责数据的存储和提取。

其架构模式是插件式的，支持 InnoDB、MyISAM、Memory 等多个存储引擎。

现在最常用的存储引擎是 InnoDB，它从 MySQL 5.5.5 版本开始成为了默认存储引擎。