# MySQL架构与历史

## MySQL逻辑架构

MySQL服务器逻辑架构图

第二层架构，大多数MySQL的核心服务功能都在这一层，包括查询解析、分析、优化、缓存以及所有的内置函数(例如，日期、时间、数学和加密函数)，所有跨存储引擎的功能都在这一层实现：存储过程、触发器、视图等。

### 1.1.2 优化与执行

用户可以使用特殊的关键字(hint)优化器，影响它的决策过程。也可以请求优化器解释

## 并发控制

### 读写锁

### 锁粒度

## 事务

事务内的语句，要么全部执行成功，要么全部执行失败。

原子性、一致性、隔离性、持久性

### 隔离级别

未提交读、提交读、可重复读、可串行化

### 死锁

## 1.5 MySQL的存储引擎

创建表时，MySQL会在数据库子目录下创建一个和表同名的.frm文件保存表的定义。

可以使用show table status命令显示表的相关信息。

### 1.5.1 InnoDB存储引擎

**InnoDB概览**

InnoDB的数据存储在表空间中，表空间是由InnoDB管理的一个黑盒子，由一系列的数据文件组成。InnoDB使用MVCC来支持高并发，并且实现了4个标准的隔离级别。其默认级别是REPEATABLE READ(可重复读)，并且通过间隙锁(next-key locking)策略防止幻读的出现。间隙锁使得InnoDB不仅仅锁定查询涉及的行，还会对索引中的间隙进行锁定，以防止幻影行的插入。

InnoDB是基于聚簇索引建立的。聚簇索引对主键查询有很高的性能。不过它的二级索引(非主键索引)中必须包含主键列，所以如果主键列很大的话，其他的所有索引都会很大。

### 1.5.6 转换表的引擎

**ALTER TABLE**

将表从一个引擎修改为另一个引擎最简单的办法是使用ALTER TABLE语句。例如，将myTable表的引擎修改为InnoDB，ALTER TABLE myTable ENGINE = InnoDB；该语法可以适用于任何引擎。但有一个问题：需要执行很长时间。MySQL会按行将数据从原表复制到一张新的表中，在复制期间会消耗系统所有的I/O能力，同时原表上会加上读锁。一个替代的方案，是采用接下来将讨论的导出与导入的方法，手工进行表的复制。

同一个数据库中不能存在相同的表名，即使使用的是不同的存储引擎。

# MySQL基准测试

# 第3章 服务器性能剖析

# 第4章 Schema与数据类型优化

## 4.1 选择优化的数据类型

## 7.1 分区表

MySQL分区意味着索引也是按照分区的子表定义的，而没有全局索引。这和Oracle不同，在Oracle中可以更加灵活的定义索引和表是否进行分区。

分区本身也有一些限制：

\* 一个表最多只能有1024个分区。

\* 在MySQL5.1中，分区表达式必须是整数，或者是返回整数的表达式。在MySQL5.5中，某些场景中可以直接使用列来进行分区。

\* 如果分区字段中有主键或者唯一索引的列，那么所有主键列和唯一索引列都必须包含进来。

\* 分区表中无法使用外键约束

### 7.1.1 分区表的原理

所有的底层表都必须使用相同的存储引擎。分区表的索引只是在各个底层表上加上各自加上一个完全相同的索引。

分区表上的操作按照下面的操作逻辑进行：

select查询

当查询一个分区表的时候，分区层先打开并锁住所有底层表，优化器先判断是否可以过滤部分分区，然后再调用对应的存储引擎接口访问各个分区的数据。

insert操作

当写入一条记录时，分区层先打开并锁住所有底层表，然后确定哪个分区接收这条记录，再将记录写入对应底层表。

delete操作

当删除一条记录时，分区层先打开并锁住底层表，然后确定数据对应的分区，最后对相应底层表进行删除操作。

update操作

当更新一条记录时，分区层先打开并锁住所有底层表，MySQL先确定需要更新的记录在哪个分区，然后取出数据并更新，再判断更新后的数据应该放在哪个分区，最后对底层表进行写入操作，并对源数据所在的底层表进行删除操作。

### 7.1.2 分区表的类型

MySQL支持多种分区表。我们看到最多的是根据范围进行分区，每个分区存储落在某个范围的记录，分区表达式可以是列，也可以是包含列的表达式。例如，下表就可以将每一年的销售额存放在不同的分区里：



PARTITION分区子句中可以使用各种函数。但有一个要求，表达式返回的值要是一个确定的整数，而不能是常数。

MySQL还支持键值、哈希和列表分区，这其中有些还支持子分区，不过在生产环境中很少见到。在MySQL5.5中，还可以使用RANGE COLUMNS类型的分区。

### 7.1.3 如何使用分区表

在数据量超大的时候(比如10T)，B-Tree索引就无法起作用了。除非是索引覆盖查询。

为了保证大数据量的可扩展性，一般有下面两个策略：

**全量扫描数据，不要任何索引。**

可以使用简单的分区方式存放表，不要任何索引，根据分区的规则大致定位需要的数据位置。只要能够使用 where条件，将需要的数据限制在少量分区中，则效率是很高的。

**索引数据，并分离热点**

如果数据有明显的“热点”，并且除了这部分数据，其他数据很少被访问到，那么可以将这部分热点数据单独存放在一个分区中，

### 7.1.4 什么情况下会出问题

上面我们介绍的两个分区策略都基于两个非常重要的假设：查询都能够过滤掉很多额外的分区、分区本身并不会带来很多额外的代价。

**NULL值会使分区过滤无效**

分区表达式的值可以是NULL：第一个分区是一个特殊分区。假设按照PARTITION BY RANGE YEAR(order\_date)分区，那么所有order\_date为NULL或者是一个非法值的时候，记录都会被存放到第一个分区。现在假设有下面的查询：where order\_date between “2012-01-01” and “2012-01-31”。实际上，MySQL会检查两个分区：它会先检查2012这个分区，同时还会检查这个表的第一个分区。检查第一个分区是因为YEAR()函数在接收非法值的时候可能会返回，那么这个值就可能会返回到第一个分区。如果第一个分区非常大，特别是当使用“全量扫描数据，不要任何索引”的策略时，代价会非常大。

**分区列和索引列不匹配**

如果定义的索引列和分区列不匹配，会导致查询无法进行分区过滤。假设在列a上定义了索引，而在列b上进行分区。因为每个分区都有其独立的索引，所以扫描列b上的索引就需要扫描每一个分区内对应的索引。要避免这个问题，应该避免建立和分区列不匹配的索引。

**选择分区的成本可能很高**

**打开并锁住所有底层表的成本可能很高**

维护分区的成本可能很高

### 7.1.5 查询优化

对于访问分区表来说，很重要的一点是要在where条件中带入分区列，有时候即使看似多余的也要带上，这样就可以让优化器能够过滤掉无需访问的分区。

使用EXPLAIN PARTITON是否执行了分区过滤，下面是一个示例：



从上面可以看到，这个查询将访问所有分区。下面我们可以在where条件中加入一个时间限制条件：



但是对于下面的例子却不能过滤分区：



MySQL只能在使用分区函数的列本身进行比较时才能过滤分区，而不能根据表达式的值去过滤分区，即使这个表达式是分区函数也不行。这就和查询中使用独立的列才能使用索引的道理是一样的。索引只需要把上面的查询等价的改成如下形式即可：



一个很重要的原则：即使在创建分区时可以使用表达式，但在查询时却只能根据列来过滤分区。

## 7.2 视图