第五章 MySQL 索引

索引在MySQL中也叫做 键(key)。是存储引擎用于快速找到记录的一种数据结构。

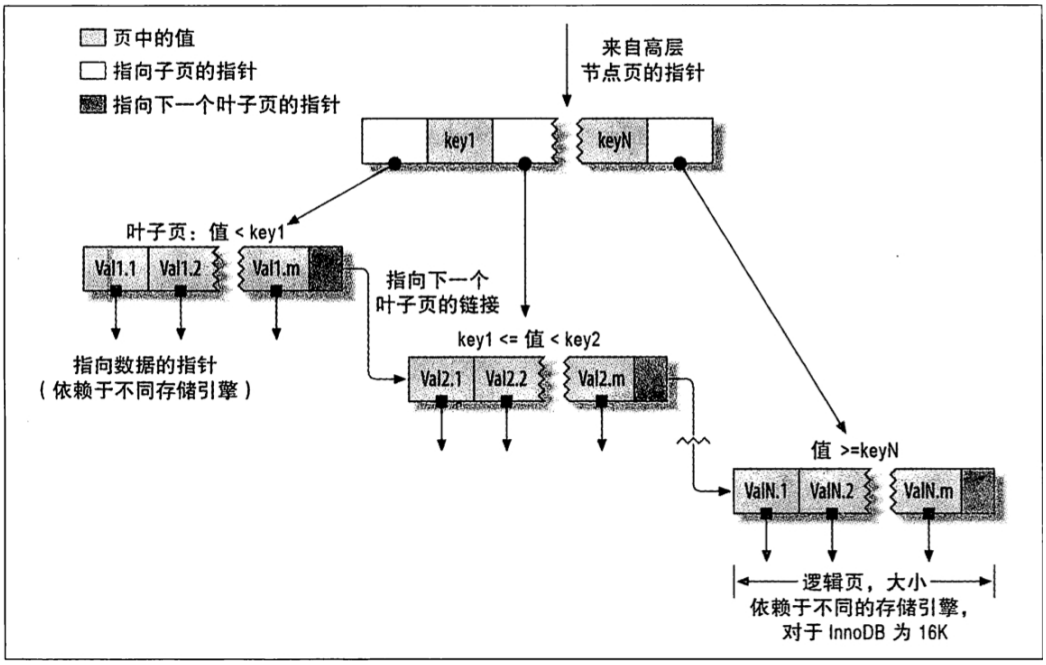
5.1 索引类型

索引有很多类型，可以为不同的场景提供不同的性能。在MySQL中索引是在*存储引擎层*而不是在服务器层实现的。因此并没有统一的标准:

- 不同的存储引擎的索引的工作方式不一样
- 不是所有的存储引擎都支持所有类型的索引
- 不同的存储引擎相同类型的索引的底层实现也不一定相同

5.1.1 B-Tree索引

如果没有特别指明类型，通常所说的索引就是指B-Tree索引，它使用B-Tree数据结构来存储数据[1]。不过底层的存储引擎也可能使用不同的数据结构。例如，NDB集群存储引擎内部实际使用的是T-Tree结构存储这种索引，不过名字用的仍然是BTREE，InnoDB使用的是B+Tree[2]。



B-Tree索引意味着所有的值都是顺序存储的，并且每一个叶子节点到根节点的距离相同。上图展示了InnoDB存储引擎的B-Tree索引的抽象表示(实际使用的是B+Tree结构)。

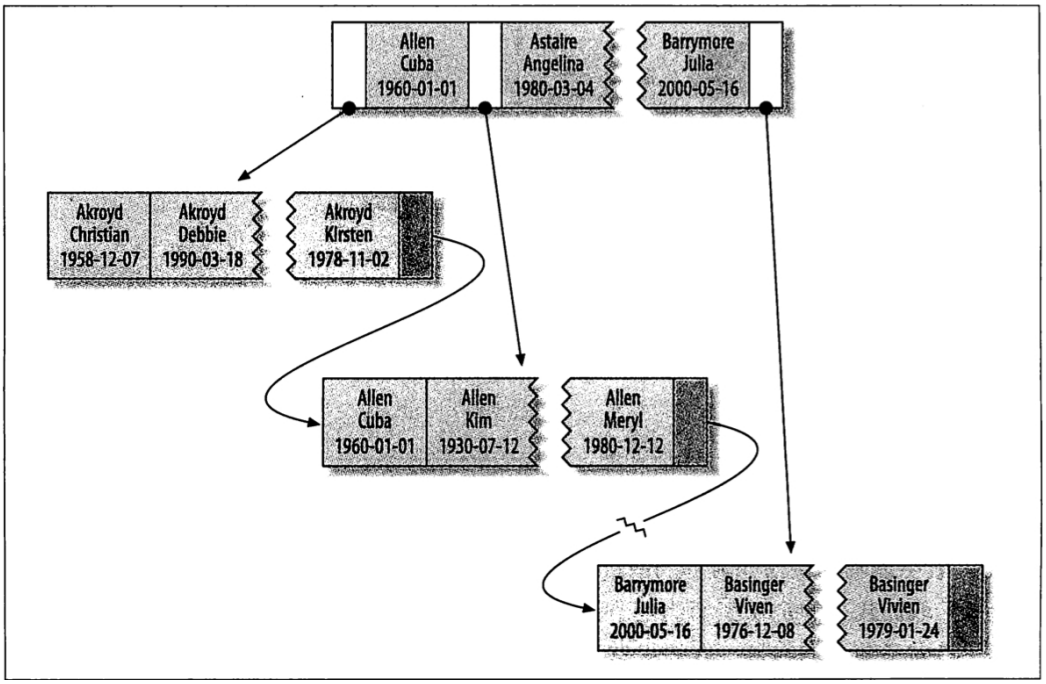
B+Tree的特点是内部节点只存储Key而没有data，叶子节点则没有指针。但是用于数据库索引时都进行了优化，叶子节点添加了指向下一个叶子节点的指针，提高区间查询效率。

B-Tree索引可以加快访问数据的速度，因为存储引擎无需进行全表扫描来获取所需要的数据，取而代之的是从根节点开始进行搜索。

B-Tree对索引列是顺序组织的，所以很适合查找范围数据。例如，在一个基于文本域的索引树上按字母顺序传递连续的值进行查找。假设有如下数据表：

```
CREATE TABLE People (
    last_name  varchar(50) not null,
    first_name varchar(50) not null,
    dob        date        not null,
    gender     enum('m', 'f')not null,
    key(last_name, first_name, dob)
    );
```

其数据示例如下图：



注意：索引对多个值进行排序的依据是**CREATE TABLE**语句中定义索引时列的

顺序。(参考最后一条记录)

可以使用B-Tree索引的查询类型

B-Tree索引适用于：

- 全键值
- 键值范围
- 键前缀查找(只适用于根据最左前缀的查找)

前面的数据表中的索引对如下类型的查询有效：

1. 全值匹配(和索引中所有列比较)
2. 匹配最左前缀 (只使用索引的第一列)
3. 匹配列前缀 (只匹配某一列的值的开头部分，只使用了索引的第一列)
4. 匹配范围值 (例如查找姓在Allen喝Barrymore之间的人， 只使用了索引的第一列)
5. 精确匹配某一列并范围匹配另外一列 (例如查找所有姓为Allen，名字以字母K开头的人，第一列全匹配，第二列范围匹配)
6. 只访问索引的查询 (B-Tree通常支持“只访问索引的查询”，即查询只需访问索引无须访问数据行)

因为索引树中的节点是有序的，因此除了按值查找之外，索引还可以用于查询中的**ORDER BY**操作。

B-Tree索引的限制

B-Tree索引也有其使用限制：

- 如果不是按照索引的最左列开始查找，则无法使用索引。例如上面的例子就无法用于查找名字为Bill的人，也无法查找某个特定生日的人，因为这两列都不是最左数据列。
- 不能跳过索引中的列。也就是说，上面例子中的索引无法用语查找姓为Smith并且出生于某个特定生日的人。即，如果不指明*first_name*则MySQL只能使用索引的第一列。
- 如果查询中有某个列的范围查询，则其右边所有列都无法使用索引优查找。例如：

```
SELECT * from People WHERE last_name='Smith' AND
first_name LIKE 'J%' AND dob ='1976-12-23';
```

这个查询只能使用索引的前两列，因为LIKE是一个范围条件。

由上面的示例可以明白，B-Tree索引的索引列顺序非常重要。所有的限制都和索引列顺序有关。因此，在优化性能的时候可能会需要使用相同的列但是顺序不同的索引来满足不同类型的查询。

5.1.2 哈希索引

哈希索引(Hash Index)基于哈希表实现，只有精确匹配索引所有列的查询才有效。在MySQL中只有**Memory引擎**显式支持哈希索引，这也是Memory引擎表的默认索引类型，Memory引擎同时也支持B-Tree索引。Memory引擎的哈希索引是支持**非唯一哈希索引**的，如果多个列的哈希索值相同，索引会以链表的方式存放多个记录的指针到同一个哈希条目中。

因为哈希索引自身只需存储对应的哈希值，所以索引的结构非常紧凑，这也让哈希索引的查找速度非常快，但是哈希索引也有其缺陷：

- 哈希索引只包含哈希值和行指针，不存储字段值，所以**不能使用索引中的值来避免读取行。**
- 哈希索引数据不是按照索引值顺序存储的，所以无法用于排序。
- 哈希索引不支持部分索引列匹配查找，因为哈希索引始终使用索引列的全部内容来计算哈希值。
- 哈希索引只支持等值比较查询，包括：=, IN(), <=>，也不支持任何范围查找，如WHERE price > 100。
- 可能会出现哈希冲突影响性能，因为出现哈希冲突时引擎需要遍历链表中的所有行指针，逐行比较。
- 如果哈希冲突很多，索引的维护代价会很高。因为当从表中删除一行时，存储引擎需要遍历对应哈希值的链表中的每一行并删除对应行的引用。

自适应哈希索引(Adaptive hash index)

InnoDB引擎中有个特殊功能，当InnoDB中某些索引值被频繁使用时，它会在内存中基于B-Tree索引自动创建一个哈希索引，这样就让B-Tree索引具有了哈希索引的一些特点。

创建自定义哈希索引

模拟InnoDB自动创建哈希索引的方式，用户可以自己创建假的哈希索引。也就是使用哈希值来进行索引查找，需要做的就是在WHERE子句中手动指定使用哈希函数。例如：

```
mysql> SELECT id FROM url WHERE url="http://www.mysql.com"
mysql> SELECT id FROM url WHERE url="http://www.mysql.com"
AND
    -> url_crc=CRC32("http://www.mysql.com")
```

只需删除以前在url上的索引而新增一个url_crc索引列就可以极大的提升速度。

5.1.3 空间数据索引

MyISAM表支持空间索引，可以用作地理位置数据存储。和B-Tree不同，这类索引无须前缀查询。

空间索引会从所有的维度来索引数据，查询时，可以有效的使用任意维度来组合查询。

5.1.4 全文索引

全文索引是一种特殊类型的索引，它查找的是文本中的关键词，而非直接比较索引中的值。

5.2 索引的优点

索引有三个优点：

1. 索引大大减少了服务器需要扫描的数据量
2. 索引可以帮助服务器避免排序和临时表
3. 索引可以将随机I/O变为顺序I/O (B-Tree索引讲相关的列值存储在一起，可以顺序读)

5.3 高性能的索引策略

5.3.1 独立的列

独立的列是指索引列不能是表达式的一部分，也不能是函数的参数。例如下面的查询就无法使用actor_id列的索引：

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

我们应该养成简化WHERE条件的习惯，始终将索引列单独放置在比较符号的一侧。下面是另一个常见错误：

```
mysql> SELECT ... WHERE TO_DAYS(CURRENT_DATE) - TO_DAYS(dat
e_col) <= 10;
```

5.3.2 前缀索引和索引选择性

有时候需要索引很长的字符串列，这会让索引变得大而慢。一个策略是前面提到的模拟哈希索引，但是可能还不够。还可以通过索引开始的部分字符串，这样可以大大节约索引空间，从而提高索引效率。但是这样会降低索引的**选择性**[3]

一般情况下某个列前缀的选择性也是足够高的，足以满足查询性能要求。对于**BLOB**、**TEXT**或者很长的**VARCHAR**类型的列，必须使用前缀索引，因为MySQL不允许索引这些列的完整长度。

如何选择前缀长度

通过选择不同的前缀长度计算其与完整列长度的选择性的差值，找到最小的差值的那个长度即可。

```
mysql> SELECT COUNT(*) AS cnt, city FROM sakila.city_demo G
ROUP BY city
    -> ORDER BY cnt DESC LIMIT 10;
mysql> SELECT COUNT(*) AS cnt, LEFT(city, 3) AS pref FROM s
akila.city_demo
    -> GROUP BY pref ORDER by cnt DESC LIMIT 10;
mysql> SELECT COUNT(*) AS cnt, LEFT(city, 7) AS pref FROM s
akila.city_demo
    -> GROUP BY pref ORDER by cnt DESC LIMIT 10;
```

还有一种方法是计算完整列的选择性，并使前缀的选择性接近于完整列的选择性。

```
mysql> SELECT COUNT(DISTINCT city) / COUNT(*) FROM sakila.c
ity_demo;
+-------------------------------+
| COUNT(DISTINCT city)/COUNT(*) |
+-------------------------------+
|                        0.0312 |
+-------------------------------+
```

通常如果前缀的选择性接近0.0312就可以使用了。

```
mysql> SELECT COUNT(DISTINCT LEFT(city, 3))/COUNT(*) AS sel
3,
    ->     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 sakila.city_demo;
+--------+--------+--------+-------+--------+
| sel3   | sel4   | sel5   | sel6  | sel7   |
+--------+--------+--------+-------+--------+
| 0.0239 | 0.0293 | 0.0305 | 0.0309| 0.0310 |
+--------+--------+--------+-------+--------+
```

当长度达到7时，选择性的增幅已经很小。找到合适的长度之后，下面演示如何创建前缀索引。

```
mysql> ALTER TABLE sakila.city_demo ADD KEY (city(7));
```

前缀索引可以使索引更小、更快，但是也有缺点。MySQL无法使用前缀索引做ORDER BY和GROUP BY，也无法使用前缀索引做覆盖扫描。

5.3.3 多列索引

常见的错误理解：为每个列创建独立的索引，或者按照错误的顺序创建多列索引。

其实，在多个列上创建独立的单列索引大部分情况下并不能提高MySQL的查询

性能。MySQL5.0引入了一种叫做*索引合并(index merge)*的策略，一定程度上可以使用表上的多个单列索引来定位制定的行。更早版本的MySQL职能使用其中某一个列索引，然而这种情况下没有哪一个独立的单列索引是有效的。

可以通过参数*optimizer_switch*来关闭索引合并功能，也可以使用**IGNORE INDEX**提示让优化器忽略掉某些索引列。

5.3.4 选择合适的索引顺序

如何选择索引的列顺序有一个经验法则：将选择性最高的列放在索引最前列。但是这并不是绝对的，避免随机I/O和排序更为重要。

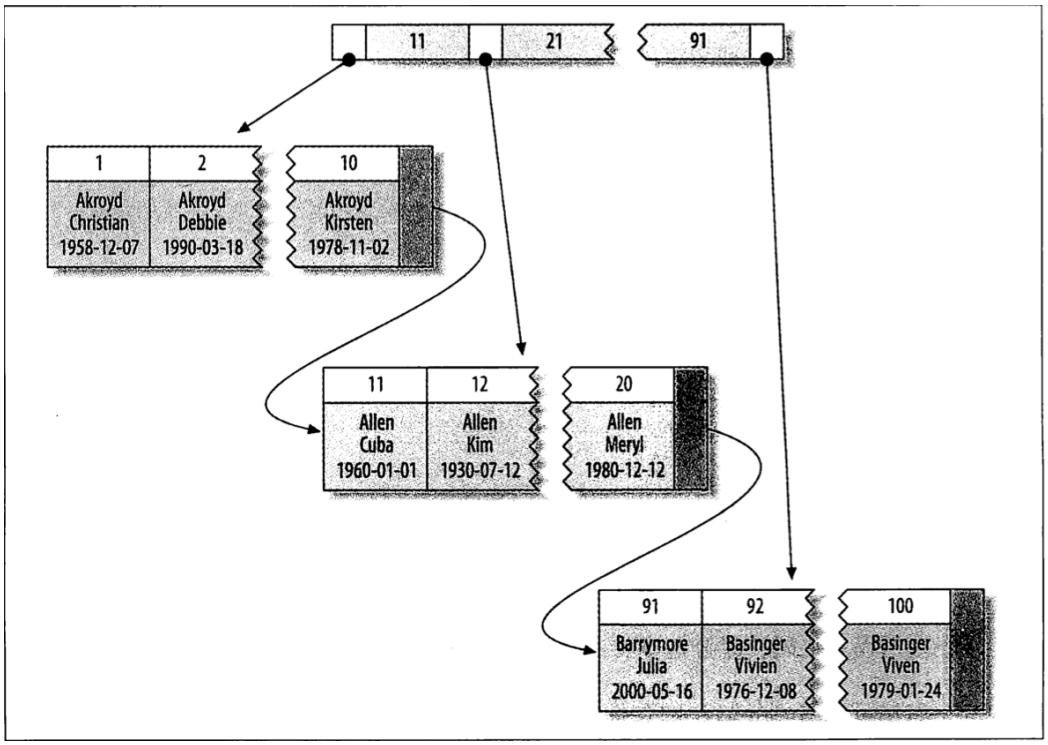
当不需要考虑排序和分组时，将选择性最高的列放前面通常效果很好。

5.3.5 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。具体细节依赖于其实现方式，但InnoDB的聚簇索引实际上在同一个数据结构中存储了B-Tree索引和数据行。

当表有聚簇索引时，它的数据行实际上存储在索引的叶子页(leaf page)中。术语*聚簇*表示数据行和相邻的键值紧凑的存储在一起。 因为无法同时把数据行存放在两个不同的地方，所以一个表只能有一个聚簇索引。

因为存储引擎负责实现索引，因此不是所有的存储引擎都支持聚簇索引。下图展示了聚簇索引中的记录是如何存放的。叶子页包含了行的全部数据，但是节点页则只包含了索引列。图中的索引项是整数值。



InnoDB通过主键聚集数据，如果没有定义主键，InnoDB会选择一个唯一的非空索引代替。如果没有这样的索引，InnoDB会隐式定义一个主键来作为聚簇索引。

InnoDB只聚集在同一个页面中的记录，包含相邻键值的页面可能相距很远。

聚簇主键可能对性能有帮助，也可能导致严重的问题。

聚集数据的优点：

- 可以把相关数据保存在一起，减少磁盘读取次数
- 数据访问更快。聚簇索引将索引和数据保存在同一个B-Tree中，因此从聚簇索引中获取数据通常比在非聚簇索引中查找更快
- 使用覆盖索引扫描的查询可以直接使用页节点中的主键值

聚集数据的缺点：

- 聚簇索引最大限度的提高了I/O密集型应用的性能，但是如果数据都放在内存中则没有什么优势了。
- 插入速度严重依赖于插入顺序。按照主键的顺序插入是加载数据到InnoDB表中速度最快的方式。如果不是，那么在加载完成后最好使用**OPTIMIZE TABLE**命令重现组织一下表。

- 更新聚簇索引列的代价很高，因为会强制InnoDB将每个被更新的行移动到新的位置。
- 基于聚簇索引的表插入新行，或者主键被更新导致需要移动行的时候，可能面临*页分裂(page split)*问题。当插入一行到某个已满的页中时，存储引擎会将该页分裂成两个页面来容纳行，这就是一次页分裂操作，页分裂会导致表占用过多的磁盘空间。
- 聚簇索引可能导致全表扫描变慢，尤其行比较稀疏，或者由于页分裂导致数据存储不连续时。
- 二级索引(非聚簇索引)可能比想象的大，因为在二级索引中叶子节点包含了引用行的主键列。
- 二级索引的访问需要两次索引查找，而不是一次。

5.3.5.1 在InnoDB表中按主键顺序插入

如果正在使用的InnoDB表没有数据需要聚集，可以定义一个代理键(surrogate key)作为主键，这种主键的数据应该和应用无关，最简单的方法是使用**AUTO_INCREAMENT**自增列。这样可以保证数据行是顺序写入，对于根据主键做关联操作的性能会更好。

最好避免随机的(不连续且值的分布范围非常大)聚簇索引，特别是对于I/O密集型的应用。例如，从性能角度考虑，用UUID来作为聚簇索引则会特别糟糕：使得聚簇索引的插入变得完全随机，这是最坏的情况，使得数据没有任何聚集特性。

Tips

顺序的主键什么时候回造成更坏的结果?
答： 对于高并发工作负载，在InnoDB中按主键顺序插入可能会造成明显的竞争。主键的上界会成为*热点*，因为所有的插入都发生在这里，所以并发插入可能导致间隙锁竞争。另一个热点可能是*AUTO_INCREMENT*锁机制，如果遇到这个问题可能需要重新设计表或者应用。

5.3.6 覆盖索引

设计优秀的索引应该考虑整个查询，而不是只考虑**WHERE**条件部分。MySQL不仅可以用索引高效的查找数据，也可以使用索引直接获取列数据。

如果一个索引包含(或者说覆盖)所有需要查询的字段值，我们就称之为**覆盖索引**。

覆盖索引可以极大的提高性能，因为这种情况下查询只需扫描索引而无须再回表扫描。

- 索引条目通常远小于数据行大小，因为如果只读取索引，MySQL会极大地减少数据访问量。
- 索引是按照列值顺序存储的(单页内)，所以对I/O密集型的范围查询会比随机从磁盘读取每一行数据的I/O少得多。
- 一些存储引擎(MyISAM)在内存中只缓存索引值，数据则依赖操作系统来缓存，因此访问数据时需要一次额外的系统调用。这个可能会导致性能问题。
- 由于InnoDB的聚簇索引，覆盖索引对InnoDB表特别有用。

不是所有的索引都可以成为覆盖索引:
覆盖索引必须要存储索引的列值，而*哈希索引*、*空间索引*和*全文索引*等都不存储列值，所以MySQL只能使用B-Tree索引做覆盖索引。

不是所有的存储引擎都支持覆盖索引。

当发起一个被索引覆盖的查询(也叫索引覆盖查询)时，在EXPLAIN的Extra列可以看到“Using Index”的信息。例如表sakila.inventory有一个多列索引(store_id, film_id)，如果只访问这两列则可以使用覆盖索引:

```
mysql> EXPLAIN SELECT store_id, film_id FROM sakila.invento
ry\G;
*****************************1. row***********************
*****
           id: 1
  select_type: SIMPLE
        table: inventory
         type: index
possible_key: null
          key: idx_store_id_film_id
      key_en: 3
          ref: null
         rows: 4673
        Extra: Using index
```

索引覆盖查询也有陷阱可能导致无法优化。MySQL查询优化器在执行查询前会判断是否有一个索引能够覆盖。即使WHERE条件全部都被索引覆盖，如果条件不成立，MySQL5.5及以前版本还是会回表查询。

MySQL5.5及旧版本在索引中只能做简单的比较操作(等于、不等于以及大于)。

MySQL不能在索引中执行LIKE操作，这是底层存储引擎API的限制。(MySQL可以在索引中执行最左前缀匹配的LIKE比较，因为该操作可以转化为简单的比较操作。但是如果是通配符开头的LIKE查询，存储引擎就无法做比较匹配。)

5.3.7 使用索引扫描来排序

MySQL有**2种**方式可以生成有序结果：

- 排序
- 按索引扫描(如果EXPLAIN出来的type为“index”，则说明MySQL用了索引扫描来做排序。)

索引扫描本身是很快的，因为时顺序存储，但是如果索引不能覆盖全部的查询列，就不得不**每扫描一条索引记录就回表查询一次对应的行**，这基本上是随机I/O。因此，按索引顺序读取数据的速度通常比顺序地全表扫描慢，尤其是在I/O密集型的工作负载时。

MySQL能够使用索引来排序的2个条件：

- 索引的列顺序和ORDER BY子句的顺序完全一致
- 所有列的排序方向(升序或降序)都一样
- 如果查询关联多张表，只有ORDER BY子句的字段全部来自第一个表
- ORDER BY子句也要满足最左前缀的要求(例外：前导列为常量)

5.3.8 压缩(前缀压缩)索引

MyISAM使用前缀压缩索引来减少索引的大小。**默认只压缩字符串**，但是通过配置也可以压缩整数。

压缩索引的方式：

完全保存索引块中的第一个值，然后将其他值和第一个值比较得到相同前缀的字节数与剩余的不同后缀部分存储起来。例如，索引块的第一个值为"perform"，第二个值是"performance"，则索引为"7, ance"。

压缩索引减少了索引的存储空间但是查找时无法进行二分查找，只能从头开始。**倒序扫描的性能尤其差**！

可以在CREATE TABLE语句中指定PACK_KEYS参数控制索引压缩方式。

5.3.9 冗余和重复索引

重复索引是指在相同的列上按照相同的顺序创建的相同类型的索引。应该避免创建这样的索引，如果有，立即删除。

冗余索引和重复索引不同，如果创建了索引(A, B)再创建索引(A)就是冗余索引，因为这只是前一个索引的前缀索引。因此(A, B)可以当作(A)来使用。但是如果再创建的索引时(B, A)则不是冗余索引。索引(B)也不是，因为它不是(A, B)的最左前缀列。

冗余索引通常发生在创建新索引的时候发生。例如新增一个(A, B)而不是扩展(A)。

大多数情况不需要冗余索引，尽量扩展索引而不是新增索引。

表中的索引越多插入速度越慢。一般来说，增加新索引会导致**INSERT, UPDATE, DELETE**等操作变慢。

5.3.11 索引和锁

索引可以让查询锁定更少的行。因为**InnodDB只有在访问行的时候才会对其加锁，而索引可以减少InnoDB行的访问数，从而减少锁的数量**。前提是，InnoDB存储引擎层能够过滤掉所有不需要的行时才有效，否则在InnoDB将数据返回给服务器层之后，MySQL服务器才能应用WHERE子句，这时候已经无法避免行锁定了。

第六章 查询优化

为什么查询会慢?

查询是由一些列子过程组成的，每个子过程都会耗费时间，因此优化查询要么减少子过程数，要么减少子过程的执行时间。

6.2 慢查询基础：优化数据访问

查询性能低的最基本原因是**访问的数据太多**。 大部分性能地下的查询都可以通

过减少访问的数据量的方式进行优化。两个分析步骤：

1. 应用程序是否在检索大量超过需要的数据。通常是访问了太多的行，也可能访问过多的列。
2. MySQL服务器层是否在分析大量超过需要的数据行。

6.2.1 是否向数据库请求了不需要的数据

典型案例：

- 查询不需要的记录， 最简单有效的解决方法是加上**LIMIT**
- 多表关联时返回全部列
- 总是用SELECT *返回全部列
- 重复查询相同的数据，比较好的办法是使用缓存

6.2.2 MySQL是否在扫描额外的记录

对于MySQL，最简单的衡量查询开销的三个指标：

- 响应时间
- 扫描的行数
- 返回的行数

响应时间 = 服务时间 + 排队时间

扫描的行数和返回的行数可以通过EXPLAIN来检查。

MySQL使用WHERE条件从好到坏的三种方式：

- 在索引中使用WHERE条件过滤不匹配的记录，在**存储引擎层**完成。
- 使用索引覆盖扫描(Extra列中出现了Using index)来返回记录，直接从索引中过滤不需要的记录并返回命中的结果。在**MySQL服务器层**完成，但是无需再回表查记录。
- 从数据表中返回数据，然后过滤不满足条件的记录(再Extra列中出现Using Where)。在**MySQL服务器层**完成，MySQL需要先从数据表读出记录然后过滤。

如果发现查询需要扫描大量的数据但是只返回少量的行，可以用下面的方式解决：

- 使用索引覆盖扫描， 把所有需要的列都放到索引中。
- 改变库表结构，使用汇总表。
- 重写查询。

6.3 重构查询的方式

- 一个复杂查询还是多个简单查询
- 对一个大的查询分而治之，将其切分为功能相同的小查询，每次只返回一小部分数据集
- 分解关联查询，简单地，对每个表做一次单表查询，然后将结果集在应用程序中进行关联。例如：

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

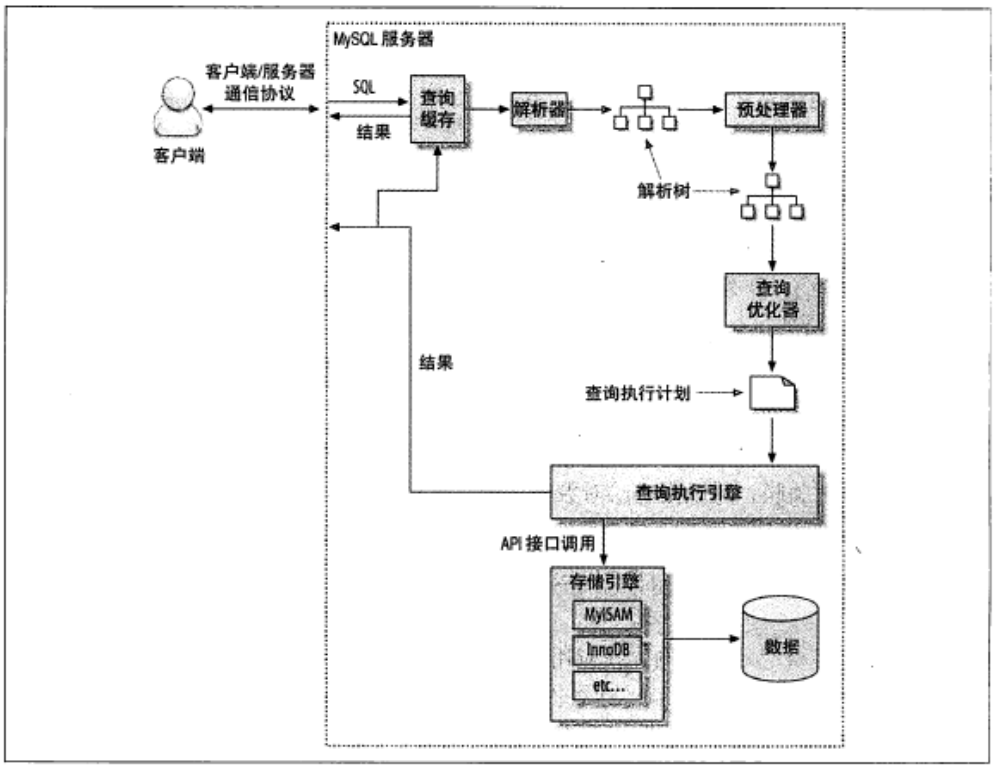
可以分解成下面这些查询替代：

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

分解关联查询的优点：

- 让缓存效率更高
- 查询分解后，执行单个查询减少锁竞争
- 在应用层做关联，可以容易对数据库拆分，更容易做到高性能和可扩展
- 查询本身的效率提高
- 减少冗余记录的查询
- 相当于实现了**哈希关联**而非MySQL的嵌套循环关联

6.4 查询执行的基础



1. 客户端发送一条查询到服务器
2. 服务器先检查缓存，如果命中立即返回，否则进入下一步
3. 服务器进行SQL解析、预处理，再由优化器生成对应的执行计划
4. MySQL根据执行计划调用存储引擎的API执行查询
5. 将结果返回给客户端

6.4.1 MySQL客户端/服务器通信协议

半双工

一个MySQL连接的**查询状态**可以用**SHOW FULL PROCESSLIST**命令查看：

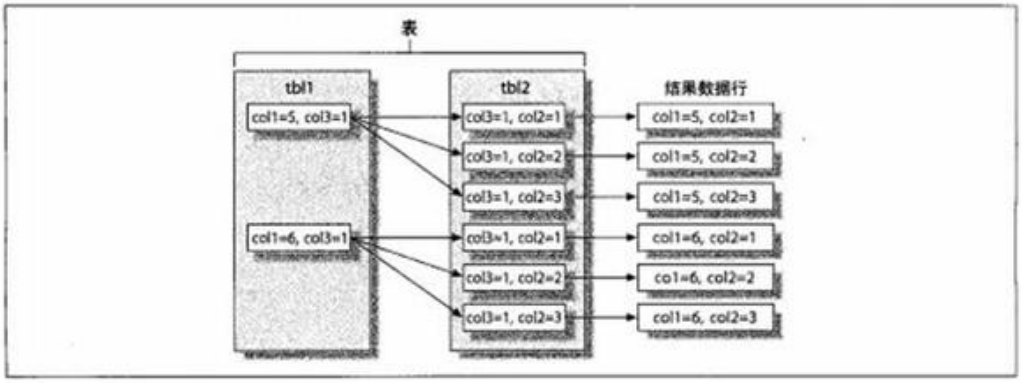
- Sleep
  线程正在等待客户端发送新请求
- Query
  线程正在执行查询或者正在将结果发送给客户端
- Locked
  在MySQL服务器层，线程正在等待表锁。存储引擎级别实现的锁并不会体现在线程状态中
- Analyzing and statistics

正在收集存储引擎的统计信息，并生成执行计划

- Copying to tmp table [on disk]
  线程正在执行查询，并将结果拷贝到一个临时表中，这种状态要么在做GROUP BY操作，要么是文件排序操作，要么是UNION操作。
- Sorting result
  线程正在对结果排序
- Sending data
  线程可能在多个状态之间发送数据，或者生成结果集，或者在向客户端发送数据

MySQL如何完成关联查询

MySQL对任何关联操作都执行**嵌套循环关联**操作，即MySQL先在一个表中循环取出单条数据，然后再嵌套循环到下一个表中寻找匹配的行，依次下去，直到找到的表中匹配的行为止。



6.5 MySQL查询优化器的局限

6.5.1 关联子查询

MySQL的子查询实现的非常糟糕!

最糟糕的一类查询是WHERE条件中包含IN()的子查询。

```
mysql> SELECT * FROM sakila.film
    -> WHERE film_id IN (
    ->     SELECT film_id FROM skill.film_actor WHERE actor
_id = 1);
```

以为是这样查询:

```
-- SELECT GROUP_CONCAT(film_id) FROM sakila.film_actor WHRE
 actor_id = 1;
-- Result: 1, 23, 25, 140, 166, 277, 361, 438
SELECT * FROM skill.film
WHERE film_id
IN (1, 23, 25, 148, 166, 277, 361, 438);
```

实际上MySQL不是这样做的，它会将外层表压到子查询中，它认为这样可以更高效率的查找的数据行。即，MySQL真正会执行下面的查询:

```
SELECT * FROM sakila.film
WHERE EXISTS (
    SELECT * FROM sakila.film_actor WHERE actor_id = 1
    AND film_actor.film_id = film.film_id);
```

优化方法:

- 用INNER JOIN改写

```
mysql> SELECT * FROM sakila.film
    ->     INNER JOIN sakila.film_actor USING (film_id)
    -> WHERE actor_id = 1;
```

- 用GROUP_CONCAT()再IN()中构造一个由逗号分割的列表，这个通常比内部关联更快

Tips

关联子查询不一定性能就一定差，需要跟进需求和试验来确定是否使用关联子查询。

6.5.2 UNION的限制

如果希望UNION各个子句能够根据LIMIT只取部分结果集，或者希望先排好序再合并结果集的话，**就需要在UNION各个子句中分别使用LIMIT**。例如，下面的例子MySQL会将两张表都存放在一个临时表中，然后再取出前20行。

```
(SELECT first_name, last_name FROM sakila.actor ORDER BY la
st_name)
UNION ALL
(SELECT first_name, last_name FROM skill.customer ORDER BY
last_name)
LIMIT 20;
```

可以再每个子句中都加上LIMIT 20限制，减少临时表的数据行。同时因为从临时表中取出的数据顺序可能不一致，所以需要全局的ORDER BY排序。

```
(SELECT first_name, last_name FROM sakila.actor
    ORDER BY last_name LIMIT 20)
UNION ALL
(SELECT first_name, last_name FROM skill.customer
    ORDER BY last_name LIMIT 20)
LIMIT 20;
```

6.5.8 最大值最小值优化

MySQL对MAX()和MIN()查询优化并不好，如果查询条件没有被索引，需要做一次全表扫描。

```
mysql> SELECT MIN(actor_id) FROM sakila.actor WHERE first_n
ame = "PENELOPE";
```

曲线办法可以移除MIN()改用LIMIT 1限制。

```
mysql> SELECT actor_id FROM sakila.actor USE INDEX(PRIMARY)

    -> WHERE first_name = "PENELOPE" LIMIT 1;
```

6.5.9 在同一个表上查询和更新

MySQL不允许对同一张表同时进行查询和更新！

6.7.1 优化关联查询

- 确保ON或者USING子句中的列上有索引
- 确保任何的GROUP BY和ORDER BY中的表达式只涉及到一个表中的列，

这样MySQL才有可能使用索引来优化
- 当升级MySQL的时候要注意：关联语法、运算符优先级等其他可能会发生变化的地方

第七章 MySQL高级特性

7.1 分区表

7.1.1 分区表的原理

对用户来说，分区表是一个独立的**逻辑表**， 但是底层由多个物理子表组成。

实现分区的代码实际上是对一组底层表的句柄对象(Handler Object)的封装。对分区表的请求都会通过句柄对象转化成对存储引擎接口的调用。

因此，分区对于SQL层来说是一个完全封装底层实现的黑盒，对应用是透明的。

由于MySQL分区表的实现是对底层物理表的封装，从而决定索引也是按照分区的子表定义的，**没有全局索引**！

MySQL在创建表时使用**PARTITION BY**子句定义每个分区存放数据。在执行查询的时候优化器会根据分区定义过滤掉没有所需数据的分区，这样查询就无须扫描所有分区了。

分区表上的操作按照下面的操作逻辑进行：

- **SELECT查询**
  分区层先打开并锁住所有的底层表，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**
  分区层先打开并锁住所有的底层表，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**
  分区层先打开并锁住所有的底层表，然后确定数据对应的分区，最后对相应的底层表执行删除操作。

- **UPDATE操作**
  分区层先打开并锁住所有的底层表，MySQL先确定需要更新的数据在哪个分区，然后取出数据并更新；再判断更新后的数据放在哪个分区，最后对底层表进行写入操作，并对原数据所在的底层表进行删除操作。

Note
虽然所有操作都会"打开并锁住所有底层表"，但是如果存储引擎本身能够实现行级锁(如InnoDB)，则会在分区层释放对应的表锁。

分区表的好处:

- 表太大，无法放入内存或者只在表尾部有热点数据，其他全是历史数据
- 便于维护
- 将数据分布在不同的设备上
- 用于避免某些特殊瓶颈，例如InnoDB的单个索引的互斥访问，ext3文件系统的inode锁竞争等
- 可以独立备份、恢复分区，大数据量的使用场景下效果比较好

分区表的限制:

- 一个表最多**1024**个分区
- 在MySQL5.1中分区表达式必须是整数或者返回值为整数的表达式，MySQL5.5中某些场景可以直接用列进行分区
- 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

7.1.2 分区表的类型

MySQL支持多种分区表。

最多的是根据范围进行分区， 每个分区存储落在某个范围内的记录。

分区表达式可以是列，也可以是包含列的表达式。例如:

```
CREATE TABLE sales (
    order_date DATETIME NOT NULL,
    -- Other columns omitted
) ENGINE=InnoDB PARTITION BY RANGE(YEAR(order_date)) (
    PARTITION p_2010 VALUES LESS THAN (2010),
    PARTITION p_2011 VALUES LESS THAN (2011),
    PARTITION p_2012 VALUES LESS THAN (2012),
    PARTITION p_catchall VALUES LESS THAN MAXVALUE);
```

PARTITION分区子句中可以使用任何函数，但是表达式的返回值必须是一个确定的整数，且不能是常数！

MySQL还支持键值、哈希和列表分区。

7.1.3 如何使用分区表

当数据量非常大时，索引的开销和维护成本会非常大，导致大量的随机I/O，**而且当数据量巨大时，B-Tree索引就无法起作用了，除非使用索引覆盖查询，否则服务器需要根据索引扫描的结果回表查询所有符合条件的记录，如果数据量巨大，会产生大量随机I/O**。

为了保证大数据量的扩展性，一般有下面两个策略：

- 全量扫描数据，不要索引。根据分区规则大致定位所需的数据位置。
- 索引数据，并分散热点。如果数据产生热点，其它数据很少被访问，可以将这些数据单独放置在一个分区中。

7.1.4 什么情况会出问题

分区表的使用基于以下两个假设：

- 查询都能过过滤掉很多额外的分区
- 分区本身不会带来很多额外的代价

但是有些情况还是会有问题：

- **NULL值使分区过滤失效**
  因为分区表达式的值可以为**NULL**，第一个分区为特殊分区。这样导致所有表达式值为NULL或者是一个非法值时，所有的记录都会被存放在第一个分区。而实际上查询"**WHERE order_date BETWEEN '2012-01-01 AND**

'2012-12-31'"除了会检查2012年这个分区还会检查第一个特殊分区，如果第一个分区非常大，当使用**全量扫描，不要任何索引**的策略时，代价会非常大。
避免这个问题的办法是创建一个"无效"分区，如果插入表中的数据都是有效的则第一个分区是空分区，即使扫描第一个分区代价也不大。在MySQL5.5中就**不需要这个优化技巧**了，因为MySQL5.5可以直接使用列自身而不是基于列的函数进行分区：**PARTITION BY RANGE COLUMNS(order_date)**。

- 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**a**列上定义了索引，而在**b**列上进行分区。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。 **因此，要避免建立和分区列不匹配的索引，除非查询中还同时包含了可以过滤分区的条件**。

- 选择分区的代价可能比较大
  对于范围分区，在查询属于哪一个分区时，服务器需要扫描所有的分区定义列表来找到正确答案，这种线性搜索的性能并不好，随着分区数的增长，成本会越来越高。可以通过**限制分区数量**来避免这个问题，通常不超过**100**个分区是没什么问题的。
  其它分区，如哈希分区、键分区没有这种问题。

- 打开并锁住所有底层表的成本可能比较高
  可以采用**批量操作**的方式降低单个操作的开销，例如使用批量插入或者**LOAD DATA INFILE**，一次删除多行数据，等等。

- 维护分区的成本可能比较高
  重组分区或者类似ALTER语句的操作需要复制数据从而导致性能较差[4]。

除了上述限制，分区表还有一些其他限制：

- 所有的分区都必须使用相同的存储引擎
- 分区函数中可以使用的函数和表达式也有一些限制
- 某些存储引擎不支持分区
- 对于MyISAM的分区表，不能再使用LOAD INDEX INTO CACHE操作
- 对于MyISAM表，使用分区表时需要打开更多的文件描述符

7.1.5 查询优化

分区的最大优点就是**优化器可以根据分区函数来过滤一些分区**。根据粗粒度索引的优势，通过分区过滤通常可以让查询扫描更少的数据。

因此，对于访问分区表来说，**很重要饿一点是要在WHERE条件中带入分区列，有时即使看似多余也要带上**。

7.1.5 合并表

合并表(Merge table)是一种早起的、简单的分区实现，和分区表相比有一些不同的限制，并且缺乏优化。

分区表是对用户透明的一个逻辑概念，用户无法直接访问底层的各个分区，但是合并表允许用户单独访问各个子表。

合并表相当于一个容器，里面包含了许多真实表，可以在CREATE TABLE中使用一种特别的UNION语法来指定包含哪些真实表。例如：

```
mysql> CREATE TABLE t1(a INT NOT NULL PRIMARY KEY)ENGINE=My
ISAM;
mysql> CREATE TABLE t2(a INT NOT NULL PRIMARY KEY)ENGINE=My
ISAM;
mysql> INSERT INTO t1(a) VALUES(1),(2);
mysql> INSERT INTO t2(a) VALUES(1),(2))
mysql> CREATE TABLE mrg(a INT NOT NULL PRIMARY KEY)
    -> ENGINE=MERGE UNION=(t1, t2) INSERT_METHOD=LAST;
mysql> SELECT a FROM mrg;
+------+
|  a   |
+------+
|  1   |
|  1   |
|  2   |
|  2   |
+------+
```

合并表的特点：

- 合并表和各个真实表的字段完全相同
- 在合并表中有的索引各个真实表也有，这是创建合并表的**前提条件**
- 合并表中每个子表的行为和表定义都相同，但是合并表在全局上并不受此限制。例如上例中，每个子表都有主键约束，但是合并表还是有重复元素

- 例子中的语法INSERT_METHODS=LAST是指将所有INSERT语句都发送给最后一个表。另一个关键字是FIRST，这俩关键字是仅有的用于控制插入行到哪个子表中的唯一关键字。
- INSERT语句的最终结果即可在合并表中看到也可以在真实表中看到:

```
mysql> INSERT INTO mrg(a) VALUES(3);
mysql> SLELECT a FROM t2;
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
+---+
```

- 合并表还有其他诸多限制

7.2 视图

MySQL 5.0以后开始引入视图。

视图是一个虚拟表，不存放任何数据，在使用SQL语句访问视图时，返回的数据是MySQL从其他表中生成的。

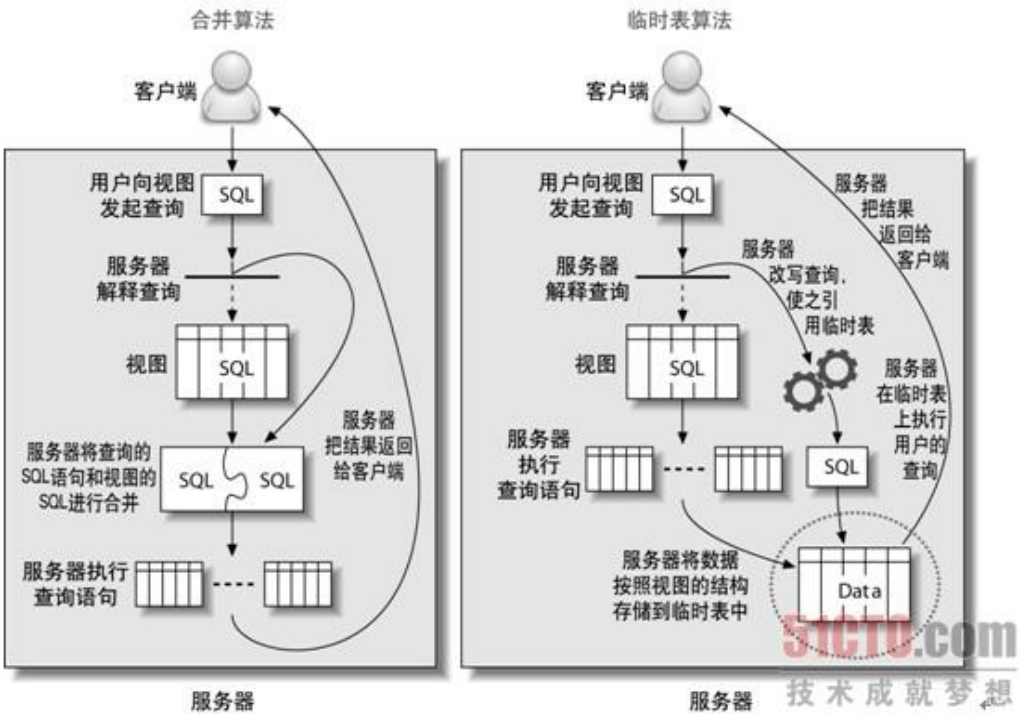
视图和表在同一个命名空间，很多地方对于视图和表是同样对待的。

视图与表也有区别，例如:

- 不能对视图创建触发器
- 不能使用DROP TABLE命令删除视图

视图有**2种**实现方法:

- 合并算法
- 临时表算法



7.2.1 可更新视图

可更新视图(updatable view)是指可以通过更新这个视图来更新视图涉及相关表。

只要指定了合适的条件，就可以更新、删除甚至向视图中写入数据。例如：

```
mysql> UPDATE Oceania SET Population = Population * 1.1 WHE
RE Name = 'Austrlia';
```

Note

如果视图定义中包含了下面几种情况就不能被更新了：

- GROUP BY
- UNION
- 聚合函数
- 其他特殊情况

MySQL不支持在视图上建立触发器。

7.2.3 视图的限制

- MySQL不支持**物化视图**[5]
- MySQL不支持在视图中创建索引
- MySQL不保存视图定义的原始SQL语句，因此SHOW CREATE VIEW的出来的视图创建语句格式很不友好

7.3 外键约束

InnoDB是目前MySQL**唯一**支持外键的内置存储引擎。

使用外键是有成本的，外键通常都要求每次修改数据时都要在另外一张表中多执行一次查找操作。

有时可以通过使用触发器来替代外键。对于相关数据的同时更新，外键更适合；但是如果外键只是用坐数值约束，那么触发器或者显示的限制取值更好。

如果只是使用外键做约束，通常在应用程序中实现约束比较好。因为外键会带来很大的额外消耗。

---

1. 很多存储引擎使用B + Tree来存储数据。↩
2. MyISAM使用前缀压缩技术使得索引更小，但是InnoDB则按照原数据格式进行存储。↩
3. 索引的选择性指，不重复的索引值(基数)和数据的记录总数(#T)的比值，范围从1/#T到1之间。↩
4. 重组分区与ALTER类似，都是先创建一个临时分区，然后将数据复制到临时分区，最后删除原分区。↩
5. 物化视图是指将视图结果数据存放在一个可以查看的表中，并定期从原始表中刷新数据到这个表中。↩