# MySQL Summary

标签（空格分隔）： mysql

---

# 一、索引

## 什么是索引：

索引，类似于书籍的目录，想找到一本书的某个特定的主题，需要先找到书的目录，定位对应的页码。
MySQL 中存储引擎使用类似的方式进行查询，先去索引中查找对应的值，然后根据匹配的索引找到对应的数据行。

## 索引有什么好处?

> 1、提高数据的检索速度，降低数据库IO成本：使用索引的意义就是通过缩小表中需要查询的记录的数目从而加快搜索的速度。

> 2、降低数据排序的成本，降低CPU消耗：索引之所以查的快，是因为先将数据排好序，若该字段正好需要排序，则正好降低了排序的成本。

## 索引有什么坏处?

> 1、占用存储空间：索引实际上也是一张表，记录了主键与索引字段，一般以索引文件的形式存储在磁盘上。

> 2、降低更新表的速度：表的数据发生了变化，对应的索引也需要一起变更，从而减低的更新速度。否则索引指向的物理数据可能不对，这也是索引失效的原因之一。

## 索引的使用场景?

> 1、对非常小的表，大部分情况下全表扫描效率更高。

> 2、对中大型表，索引非常有效。

> 3、特大型的表，建立和使用索引的代价随着增长，可以使用分区技术来解决。

## MySQL索引类型：

索引，都是实现在存储引擎层的。主要有六种类型：

> 1、普通索引：最基本的索引，没有任何约束。

2、唯一索引：与普通索引类似，但具有唯一性约束。

3、主键索引：特殊的唯一索引，不允许有空值。

4、复合索引：将多个列组合在一起创建索引，可以覆盖多个列。

5、外键索引：只有InnoDB类型的表才可以使用外键索引，保证数据的一致性、完整性和实现级联操作。

6、全文索引：MySQL 自带的全文索引只能用于 InnoDB、MyISAM。并且只能对英文进行全文检索，一般使用全文索引引擎。

## MySQL索引的“创建”原则：

1、最适合索引的列是出现在 WHERE 子句中的列，或连接子句中的列，而不是出现在 SELECT 关键字后的列。

2、索引列的基数越大，索引效果越好。

3、根据情况创建复合索引，复合索引可以提高查询效率。

4、避免创建过多的索引，索引会额外占用磁盘空间，降低写操作效率。

5、主键尽可能选择较短的数据类型，可以有效减少索引的磁盘占用提高查询效率。

6、对字符串进行索引，应该定制一个前缀长度，可以节省大量的索引空间。

## MySQL索引的“使用”注意事项：

### 不要在列上使用函数和进行运算

不要在列上使用函数，这将导致索引失效而进行全表扫描。

```
select * from news where year(publish_time) < 2017
```

为了使用索引，防止执行全表扫描，可以进行改造。

```
select * from news where publish_time < '2017-01-01'
```

还有一个建议，不要在列上进行运算，这也将导致索引失效而进行全表扫描。

```
select * from news where id / 100 = 1
```

为了使用索引，防止执行全表扫描，可以进行改造。

```
select * from news where id = 1 * 100
```

### 尽量避免使用 != 或 not in或 <> 等否定操作符

应该尽量避免在 where 子句中使用 != 或 not in 或 <> 操作符，因为这几个操作符都会导致索引失效而进行全表扫描。

### 尽量避免使用 or 来连接条件

应该尽量避免在 where 子句中使用 or 来连接条件，因为这会导致索引失效而进行全表扫描。

```
select * from news where id = 1 or id = 2
```

### 多个单列索引并不是最佳选择

MySQL 只能使用一个索引，会从多个索引中选择一个限制最为严格的索引，因此，为多个列创建单列索引，并不能提高查询性能。

假设，有两个单列索引，分别为 news_year_idx(news_year) 和 news_month_idx(news_month)。现在，有一个场景需要针对资讯的年份和月份进行查询，那么，SQL 语句可以写成：

```
select * from news where news_year = 2017 and news_month = 1
```

事实上，MySQL 只能使用一个单列索引。为了提高性能，可以使用复合索引 news_year_month_idx(news_year, news_month) 保证 news_year 和 news_month 两个列都被索引覆盖。

### 复合索引的最左前缀原则

复合索引遵守“最左前缀”原则，即在查询条件中使用了复合索引的第一个字段，索引才会被使用。因此，在复合索引中索引列的顺序至关重要。如果不是按照索引的最左列开始查找，则无法使用索引。

假设，有一个场景只需要针对资讯的月份进行查询，那么，SQL 语句可以写成：

```
select * from news where news_month = 1
```

此时，无法使用 news_year_month_idx(news_year, news_month) 索引，因为遵守“最左前缀”原则，在查询条件中没有使用复合索引的第一个字段，索引是不会被使用的。

### 覆盖索引的好处

如果一个索引包含所有需要的查询的字段的值，直接根据索引的查询结果返回数据，而无需读表，能够极大的提高性能。因此，可以定义一个让索引包含的额外的列，即使这个列对于索引而言是无用的。

### 范围查询对多列查询的影响

查询中的某个列有范围查询，则其右边所有列都无法使用索引优化查找。

举个例子，假设有一个场景需要查询本周发布的资讯文章，其中的条件是必须是启用状态，且发布时间在这周内。那么，SQL 语句可以写成：

```
select * from news where publish_time >= '2017-01-02' and publish_time <= '2017-01-08' and enable
= 1
```

这种情况下，因为范围查询对多列查询的影响，将导致 news_publish_idx(publish_time, enable) 索引中 publish_time 右边所有列都无法使用索引优化查找。换句话说，news_publish_idx(publish_time, enable) 索引等价于 news_publish_idx(publish_time) 。

对于这种情况，我的建议：对于范围查询，务必要注意它带来的副作用，并且尽量少用范围查询，可以通过曲线救国的方式满足业务场景。

例如，上面案例的需求是查询本周发布的资讯文章，因此可以创建一个news_weekth 字段用来存储资讯文章的周信息，使得范围查询变成普通的查询，SQL 可以改写成：

```
select * from news where news_weekth = 1 and enable = 1
```

然而，并不是所有的范围查询都可以进行改造，对于必须使用范围查询但无法改造的情况，不必试图用 SQL 来解决所有问题，可以使用其他数据存储技术控制时间轴，例如 Redis 的 SortedSet 有序集合保存时间，或者通过缓存方式缓存查询结果从而提高性能。

### 索引不会包含有NULL值的列

只要列中包含有 NULL 值都将不会被包含在索引中，复合索引中只要有一列含有 NULL值，那么这一列对于此复合索引就是无效的。

因此，在数据库设计时，除非有一个很特别的原因使用 NULL 值，不然尽量不要让字段的默认值为 NULL。

### 隐式转换的影响

当查询条件左右两侧类型不匹配的时候会发生隐式转换，隐式转换带来的影响就是可能导致索引失效而进行全表扫描。下面的案例中，date_str 是字符串，然而匹配的是整数类型，从而发生隐式转换。

```
select * from news where date_str = 201701
```

因此，要谨记隐式转换的危害，时刻注意通过同类型进行比较。

### like 语句的索引失效问题

like 的方式进行查询，在 like “value%” 可以使用索引，但是对于 like “%value%” 这样的方式，执行全表查询，这在数据量小的表，不存在性能问题，但是对于海量数据，全表扫描是非常可怕的事情。所以，根据业务需求，考虑使用 ElasticSearch 或 Solr 是个不错的方案。

**Example:** 以下三条 SQL 如何建索引，只建一条怎么建？

```
WHERE a = 1 AND b = 1
WHERE b = 1
WHERE b = 1 ORDER BY time DESC
```

> - 以顺序 b , a, time 建立复合索引，CREATE INDEX table1_b_a_time ON index_test01(b, a, time)。
> - 对于第一条 SQL ， 因为最新 MySQL 版本会优化 WHERE 子句后面的列顺序，以匹配复合索引顺序。

### 用 exists 代替 in 是一个好的选择

```
select num from a where num in(select num from b)
```

用下面语句替换：

```
select num from a where exists(select 1 from b where num = a.num)
```

### 并不是所有索引对查询都有效

SQL是根据表中数据来进行查询优化的，当索引列有大量数据重复时，SQL查询可能不会去利用索引，如一表中有字段sex，male、female几乎各一半，那么即使在sex上建了索引也对查询效率起不了作用。

### 索引并不是越多越好

索引固然可以提高相应的 select 的效率，但同时也降低了insert 及 update 的 效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有 必要。

### 尽可能的避免更新 clustered 索引数据列

因为 clustered 索引数据列的顺序就是表记录的物理存储 顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为非clustered 索引。

### 尽可能的使用 varchar/nvarchar 代替 char/nchar

因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

### 任何地方都不要使用 select * from t，用具体的字段列表代替“*”

### 尽量避免使用游标

因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写。

### 想知道一个查询用到了哪个索引，如何查看?

`EXPLAIN` 显示了 MYSQL 如何使用索引来处理 SELECT 语句以及连接表,可以帮助选择更好的索引和写出更优化的查询语句。
使用方法，在 `SELECT` 语句前加上 `EXPLAIN` 就可以了.

MySQL执行计划详解

## MySQL索引的原理

### 1、B-Tree索引（平衡多路查找树）

B-Tree 指的是 Balance Tree，也就是平衡树。平衡树是一颗查找树，并且所有叶子节点位于同一层。
B-Tree是为磁盘等外存储设备设计的一种平衡查找树。因此在讲B-Tree之前先了解下磁盘的相关知识。

> - 系统从磁盘读取数据到内存时是以磁盘块（block）为基本单位的，位于同一个磁盘块中的数据会被一次性读取出来，而不是需要什么取什么。
> - InnoDB存储引擎中有页（Page）的概念，页是其磁盘管理的最小单位。InnoDB存储引擎中默认每个页的大小为16KB，可通过参数innodb_page_size将页的大小设置为4K、8K、16K，在MySQL中可通过如下命令查看页的大小：

```
mysql> show variables like 'innodb_page_size';
```

> - 而系统一个磁盘块的存储空间往往没有这么大，因此InnoDB每次申请磁盘空间时都会是若干地址连续磁盘块来达到页的大小16KB。

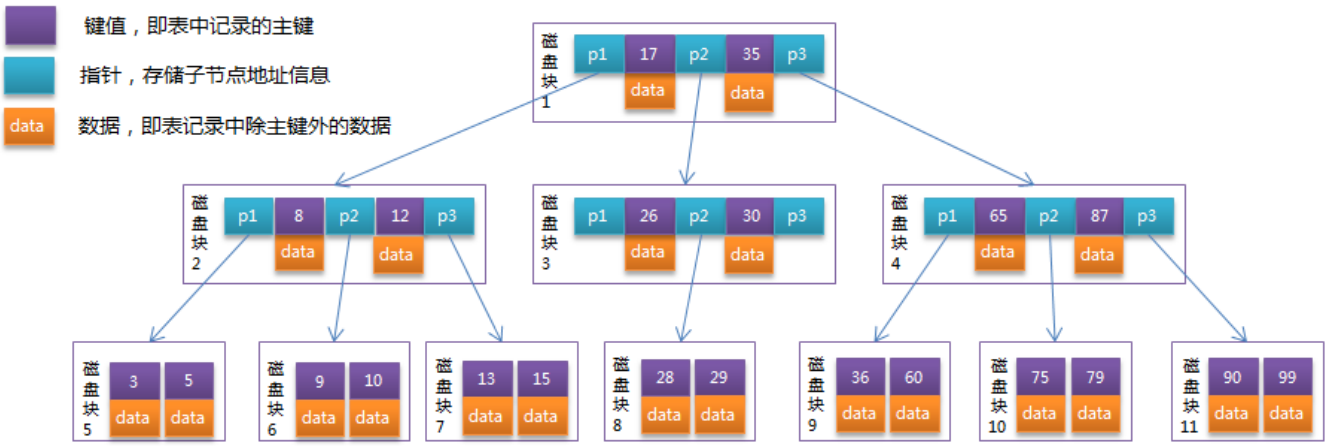
> - InnoDB在把磁盘数据读入到内存时会以页为基本单位，在查询数据时如果一个页中的每条数据都能有助于定位数据记录的位置，这将会减少磁盘I/O次数，提高查询效率。

B-Tree结构的数据可以让系统高效的找到数据所在的磁盘块。为了描述B-Tree，首先定义一条记录为一个二元组[key, data]，key为记录的键值，对应表中的主键值，data为一行记录中除主键外的数据。对于不同的记录，key值互不相同。

一棵m阶的B-Tree有如下特性：

> 1. 每个节点最多有m个孩子。
> 2. 除了根节点和叶子节点外，其它每个节点至少有Ceil(m/2)个孩子。
> 3. 若根节点不是叶子节点，则至少有2个孩子
> 4. 所有叶子节点都在同一层，且不包含其它关键字信息
> 5. 每个非终端节点包含n个关键字信息（P0,P1,…Pn, k1,…kn）
> 6. 关键字的个数n满足：ceil(m/2)-1 <= n <= m-1
> 7. ki(i=1,…n)为关键字，且关键字升序排序。
> 8. Pi(i=1,…n)为指向子树根节点的指针。P(i-1)指向的子树的所有节点关键字均小于ki，但都大于k(i-1)

B-Tree中的每个节点根据实际情况可以包含大量的关键字信息和分支，如下图所示为一个3阶的B-Tree：




每个节点占用一个盘块的磁盘空间，一个节点上有两个 升序排序 的关键字和三个 指向子树根节点的指针 ，指针存储的是子节点所在磁盘块的地址。两个关键词划分成的三个范围域对应三个指针指向的子树的数据的范围域。以根节点为例，关键字为17和35，P1指针指向的子树的数据范围为小于17，P2指针指向的子树的数据范围为17~35，P3指针指向的子树的数据范围为大于35。

**模拟查找关键字29的过程：**

> 1、根据根节点找到磁盘块1，读入内存。【磁盘I/O操作第1次】
> 2、比较关键字29在区间（17,35），找到磁盘块1的指针P2。
> 3、根据P2指针找到磁盘块3，读入内存。【磁盘I/O操作第2次】
> 4、比较关键字29在区间（26,30），找到磁盘块3的指针P2。
> 5、根据P2指针找到磁盘块8，读入内存。【磁盘I/O操作第3次】
> 6、在磁盘块8中的关键字列表中找到关键字29。

分析上面过程，发现需要3次磁盘I/O操作，和3次内存查找操作。由于内存中的关键字是一个有序表结构，可以利用二分法查找提高效率。而3次磁盘I/O操作是影响整个B-Tree查找效率的决定因素。B-Tree相对于AVLTree缩减了节点个数，使每次磁盘I/O取到内存的数据都发挥了作用，从而提高了查询效率。

## 2、B+Tree索引（平衡多路查找树）

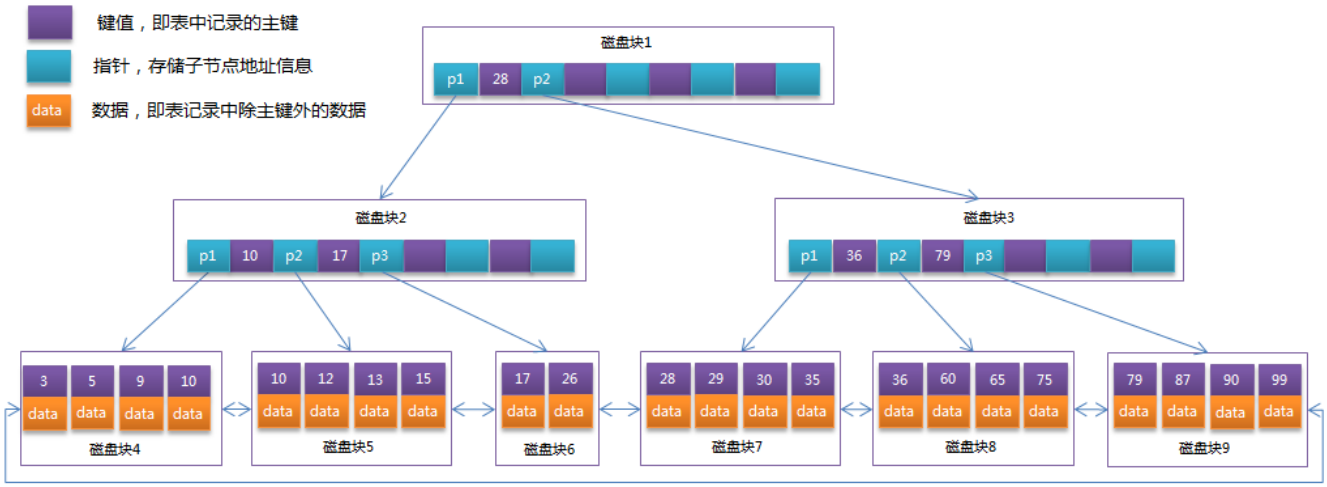
B+ Tree 是基于 B-Tree 和叶子节点顺序访问指针进行实现，它具有 B-Tree 的平衡性，并且通过顺序访问指针来提高区间查询的性能。

从上一节中的B-Tree结构图中可以看到每个节点中不仅包含数据的key值，还有data值。而每一个页的存储空间是有限的，如果data数据较大时将会导致每个节点（即一个页）能存储的key的数量很小，当存储的数据量很大时同样会**导致B-Tree的深度较大，增大查询时的磁盘I/O次数**，进而影响查询效率。在B+Tree中，所有数据记录节点都是按照键值大小顺序存放在同一层的叶子节点上，而**非叶子节点上只存储key值信息**，这样可以大大加大每个节点存储的key值数量，降低B+Tree的高度。

B+Tree相对于B-Tree有几点不同：

> 1、非叶子节点只存储键值信息。
> 2、所有叶子节点之间都有一个链指针。
> 3、数据记录都存放在叶子节点中。

将上一节中的B-Tree优化，由于B+Tree的非叶子节点只存储键值信息，假设每个磁盘块能存储4个键值及指针信息，则变成B+Tree后其结构如下图所示：




通常在B+Tree上有两个头指针，一个指向根节点，另一个指向关键字最小的叶子节点，而且所有叶子节点（即数据节点）之间是一种链式环结构。因此可以对B+Tree进行两种查找运算：一种是对于主键的范围查找和分页查找，另一种是从根节点开始，进行随机查找。

可能上面例子中只有22条数据记录，看不出B+Tree的优点，下面做一个推算：

InnoDB存储引擎中页的大小为16KB，一般表的主键类型为INT（占用4个字节）或BIGINT（占用8个字节），指针类型也一般为4或8个字节，也就是说一个页（B+Tree中的一个节点）中大概存储16KB/(8B+8B)=1K个键值（因为是估值，为方便计算，这里的K取值为10^3）。也就是说一个深度为3的B+Tree索引可以维护10^3 * 10^3 * 10^3 = 10亿 条记录。

实际情况中每个节点可能不能填充满，因此在数据库中，B+Tree的高度一般都在2~4层。mysql的InnoDB存储引擎在设计时是将根节点常驻内存的，也就是说查找某一键值的行记录时最多只需要1~3次磁盘I/O操作。

数据库中的B+Tree索引可以分为**聚集索引（clustered index）**和**辅助索引（secondary index）**。上面的B+Tree示例图在数据库中的实现即为聚集索引，聚集索引的B+Tree中的叶子节点存放的是整张表的行记录数据。辅助索引与聚集索引的区别在于辅助索引的叶子节点并不包含行记录的全部数据，而是存储相应行数据的聚集索引键，即主键。当通过辅助索引来查询数据时，InnoDB存储引擎会遍历辅助索引找到主键，然后再通过主键在聚集索引中找到完整的行记录数据。

B+Tree的特性：

1.所有关键字都出现在叶子结点的链表中（稠密索引），且链表中的关键字恰好是有序的；

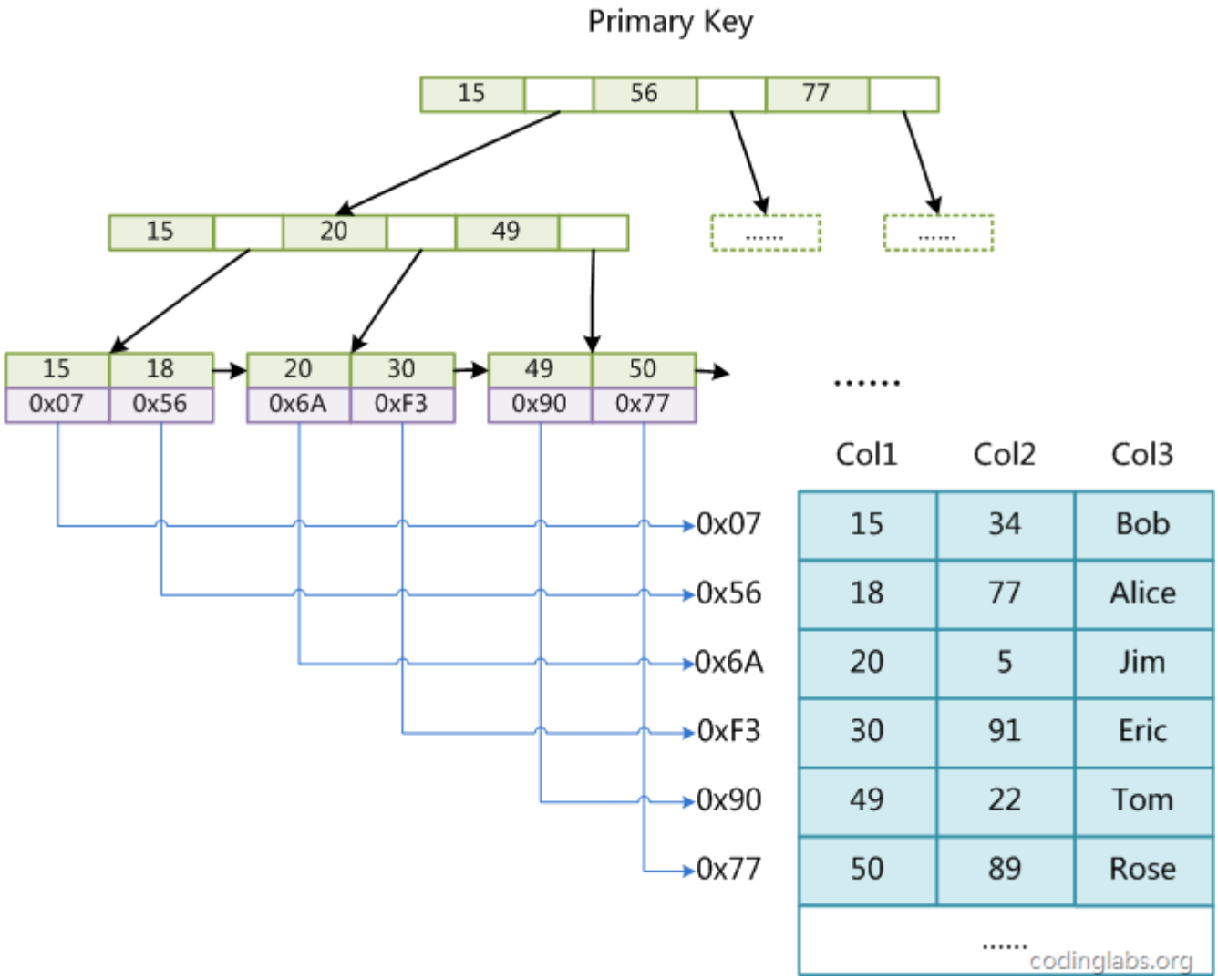
2.不可能在非叶子结点命中；

3.非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储（关键字）数据的数据层；

4.更适合文件索引系统；

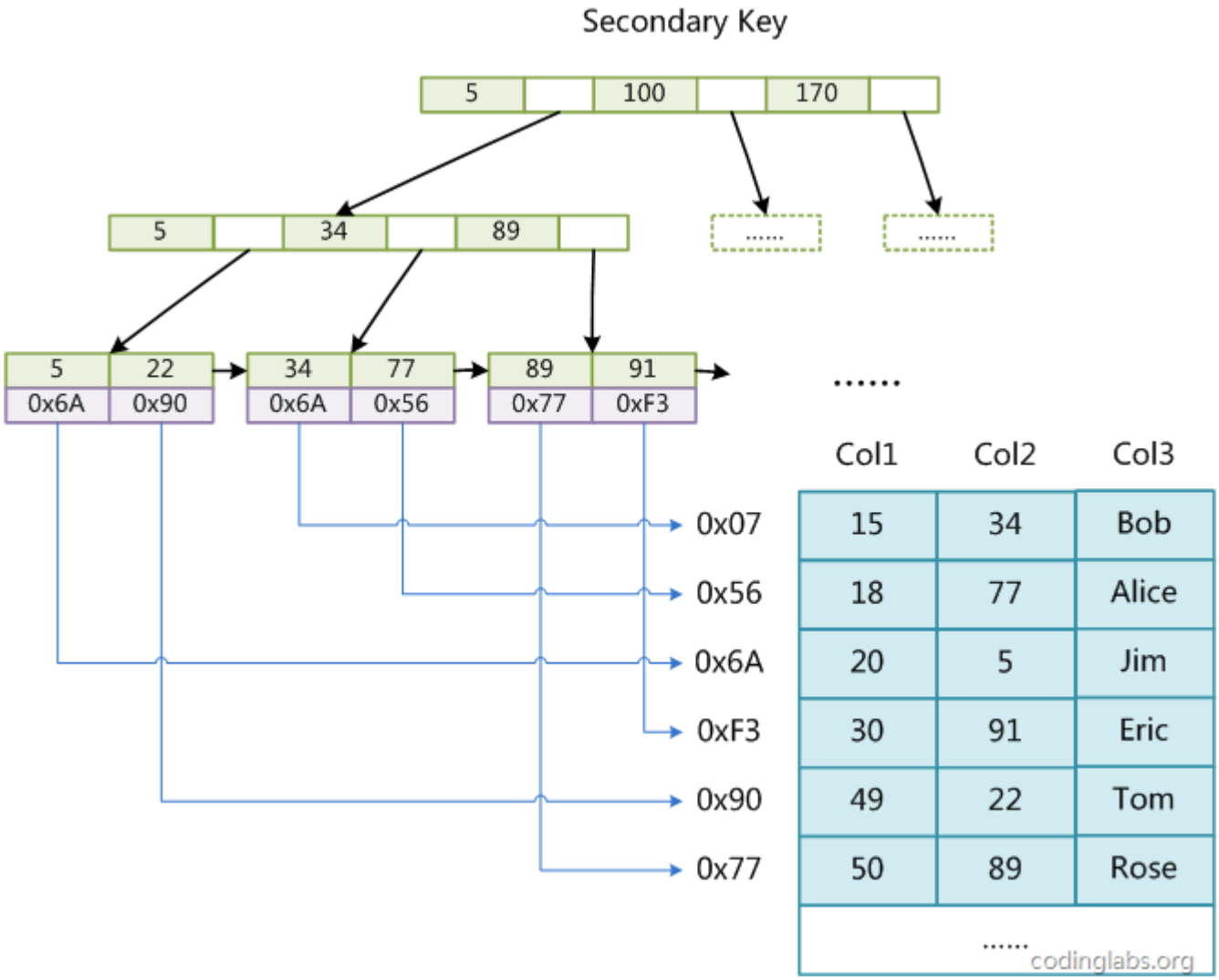
### 3、MyISAM 索引实现

**1）主键索引：**




**1）辅助索引：**

在 MyISAM 中，主索引和辅助索引在结构上没有任何区别，只是主索引要求 key 是唯一的，而辅助索引的 key 可以重复。如果我们在 Col2 上建立一个辅助索引，则此索引的结构如下图所示：




同样也是一颗 B+Tree ， data 域保存数据记录的地址。因此， **MyISAM 中索引检索的算法为首先按照 B+Tree 搜索算法搜索索引，如果指定的 Key 存在，则取出其 data 域的值，然后以 data 域的值为地址，读取相应数据记录。**

### 4、MyISAM 索引与 InnoDB 索引的区别

> - InnoDB 索引是聚簇索引，MyISAM 索引是非聚簇索引。
> - InnoDB 的主键索引的叶子节点存储着行数据，因此主键索引非常高效。
> - MyISAM 索引的叶子节点存储的是行数据地址，需要再寻址一次才能得到数据。
> - InnoDB 非主键索引的叶子节点存储的是主键和其他带索引的列数据，因此查询时做到覆盖索引会非常高效。

## 二、MySQL事务的特性以及四种隔离级别

### 1、事物的特性


- 1、原子性 Atomicity： 一个事务（transaction）中的所有操作，或者全部完成，或者全部不完成，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。即，事务不可分割、不可约简。

- 2、一致性 Consistency： 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约束、触发器)、级联回滚等。

- 3、隔离性 Isolation： 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）。

- 4、持久性 Durability： 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。

## 2、事物的并发问题:

实际场景下，事务并不是串行的，所以会带来如下三个问题：

- 1、脏读：事务 A 读取了事务 B 更新的数据，然后 B 回滚操作，那么 A 读取到的数据是脏数据。
- 2、不可重复读：事务 A 多次读取同一数据，事务 B 在事务 A 多次读取的过程中，对数据作了更新并提交，导致事务 A 多次读取同一数据时，结果不一致。
- 3、幻读：系统管理员 A 将数据库中所有学生的成绩从具体分数改为 ABCDE 等级，但是系统管理员 B 就在这个时候插入了一条具体分数的记录，当系统管理员 A 改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。

小结：不可重复读的和幻读很容易混淆，不可重复读侧重于修改，幻读侧重于新增或删除。解决不可重复读的问题只需锁住满足条件的行，解决幻读需要锁表。

## 3、 MySQL 事务隔离级别会产生的并发问题：

事务定义了四种事务隔离级别，不同数据库在实现时，产生的并发问题是不同的。

> **不同的隔离级别有不同的现象，并有不同的锁定/并发机制，隔离级别越高，数据库的并发性就越差。**

- READ UNCOMMITTED（读未提交）：事务中的修改，即使没有提交，对其他事务也都是可见的。

  > 会导致脏读

- READ COMMITTED（读已提交）：事务从开始直到提交之前，所做的任何修改对其他事务都是不可见的。

  > 会导致不可重复读。 这个隔离级别，也可以叫做“不可重复读”。

- REPEATABLE READ（可重复读）：一个事务按相同的查询条件读取以前检索过的数据，其他事务插入了满足其查询条件的新数据。产生幻行。

  > 会导致幻读。

- SERIALIZABLE（可串行化）：强制事务串行执行。

MySQL InnoDB 采用 MVCC 来支持高并发，实现结果如下表所示:

| 事务隔离级别 | 脏读 | 不可重复读 | 幻读 |
| --- | --- | --- | --- |
| 读未提交（READ UNCOMMITTED） | 是 | 是 | 是 |
| 读已提交（READ COMMITTED） | 否 | 是 | 是 |
| 可重复读（REPEATABLE READ） | 否 | 否 | 是（X） |
| 串行化（SERIALIZABLE） | 否 | 否 | 否 |

> - **MySQL 默认的事务隔离级别为可重复读（repeatable-read）** 。
> - 上图的 处，MySQL 因为其**间隙锁**的特性，导致其在可重复读（repeatable-read）的隔离级别下，不存在幻读问题。也就是说，**上图 处，需要改成“否”！！！！** 其实 REPEATABLE READ 也是可以避免幻读的，**通过对 select 操作手动加 行X锁（SELECT ... FOR UPDATE 这也正是 SERIALIZABLE 隔离级别下会隐式为你做的事情），同时还需要知道，即便当前记录不存在，比如 id = 1 是不存在的，当前事务也会获得一把记录锁（因为InnoDB的行锁锁定的是索引，故记录实体存在与否没关系，存在就加 行X锁，不存在就加 next-key lock间隙X锁）**，其他事务则无法插入此索引的记录，故杜绝了幻读。

# 三、MySQL的锁机制

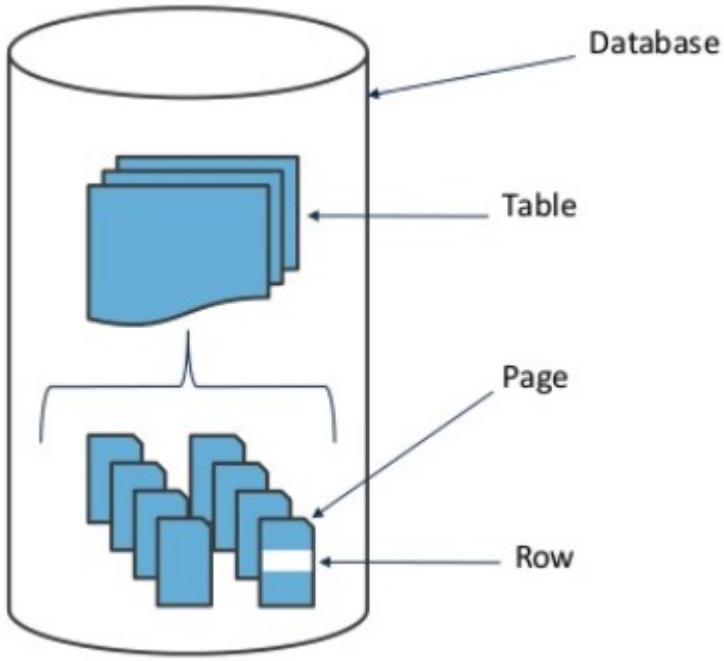
## 1、锁的粒度

MySQL 中提供了两种封锁粒度：**行级锁以及表级锁。**

应该尽量只锁定需要修改的那部分数据，而不是所有的资源。锁定的数据量越少，发生锁争用的可能就越小，系统的并发程度就越高。

但是加锁需要消耗资源，锁的各种操作（包括获取锁、释放锁、以及检查锁状态）都会增加系统开销。因此封锁粒度越小，系统开销就越大。

在选择封锁粒度时，需要在锁开销和并发程度之间做一个权衡。




## 2、锁的类型

### 1、读写锁

> - 排它锁（Exclusive），简写为 X 锁，又称写锁。
> - 共享锁（Shared），简写为 S 锁，又称读锁。

有以下两个规定：

一个事务对数据对象 A 加了 X 锁，就可以对 A 进行读取和更新。加锁期间其它事务不能对 A 加任何锁。 一个事务对数据对象 A 加了 S 锁，可以对 A 进行读取操作，但是不能进行更新操作。加锁期间其它事务能对 A 加 S 锁，但是不能加 X 锁。

锁的兼容关系如下:

| | X锁 | S锁 |
|---|---|---|
| X锁 | No | No |
| S锁 | No | Yes |

### 2、意向锁

使用意向锁（Intention Locks）可以更容易地支持多粒度封锁。

在存在行级锁和表级锁的情况下，事务 T 想要对表 A 加 X 锁，就需要先检测是否有其它事务对表 A 或者表 A 中的任意一行加了锁，那么就需要对表 A 的每一行都检测一次，这是非常耗时的。

意向锁在原来的 X/S 锁之上引入了 IX/IS，IX/IS 都是表锁，用来表示一个事务想要在表中的某个数据行上加 X 锁或 S 锁。有以下两个规定：

> - 一个事务在获得某个数据行对象的 S 锁之前，必须先获得表的 IS 锁或者更强的锁；
> - 一个事务在获得某个数据行对象的 X 锁之前，必须先获得表的 IX 锁。

通过引入意向锁，事务 T 想要对表 A 加 X 锁，只需要先检测是否有其它事务对表 A 加了 X/IX/S/IS 锁，如果加了就表示有其它事务正在使用这个表或者表中某一行的锁，因此事务 T 加 X 锁失败。

各种锁的兼容关系如下：

| | X | IX | S | IS |
|---|---|---|---|---|
| X | No | No | No | No |
| IX | No | Yes | No | Yes |
| S | No | No | Yes | Yes |
| IS | No | Yes | Yes | Yes |

解释如下：

> 1、任意 IS/IX 锁之间都是兼容的，因为它们只是表示想要对表加锁，而不是真正加锁； 2、S 锁只与 S 锁和 IS 锁兼容，也就是说事务 T 想要对数据行加 S 锁，其它事务可以已经获得对表或者表中的行的 S 锁。

### 3、MySQL隐式与显示锁定：

MySQL 的 InnoDB 存储引擎采用两段锁协议，会根据隔离级别在需要的时候自动加锁，并且所有的锁都是在同一时刻被释放，这被称为隐式锁定。 InnoDB 也可以使用特定的语句进行显示锁定：

```
SELECT ... LOCK In SHARE MODE;
SELECT ... FOR UPDATE;
```

### 4、乐观锁与悲观锁

#### 1、悲观锁

它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持**保守态度**，因此，在整个数据处理过程中，将数据处于锁定状态。**悲观锁的实现，往往依靠数据库提供的锁机制**（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

在悲观锁的情况下，为了保证事务的隔离性，就需要一致性锁定读。读取数据时给加锁，其它事务无法修改这些数据。修改删除数据时也要加锁，其它事务无法读取这些数据。

> 悲观锁，就是我们上面看到的共享锁和排他锁。

#### 2、乐观锁

相对悲观锁而言，乐观锁机制采取了更加宽松的加锁机制。悲观锁大多数情况下依靠数据库的锁机制实现，以保证操作最大程度的独占性。但随之而来的就是数据库性能的大量开销，特别是对长事务而言，这样的开销往往无法承受。

而乐观锁机制在一定程度上解决了这个问题。乐观锁，大多是**基于数据版本（ Version ）记录机制实现。**何谓数据版本？即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个 “version” 字段来实现。读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提交数据的版本数据与数据库表对应记录的当前版本信息进行比对，如果提交的数据版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。

> 乐观锁，实际就是通过版本号，从而实现 CAS 原子性更新。

### 5、死锁

多数情况下，可以认为如果一个资源被锁定，它总会在以后某个时间被释放。而死锁发生在当多个进程访问同一数据库时，其中每个进程拥有的锁都是其他进程所需的，由此造成每个进程都无法继续下去。简单的说，进程 A 等待进程 B 释放他的资源，B 又等待 A 释放他的资源，这样就互相等待就形成死锁。

虽然进程在运行过程中，可能发生死锁，但死锁的发生也必须具备一定的条件，死锁的发生必须具备以下四个必要条件：

> - 互斥条件：指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。
> - 请求和保持条件：指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。
> - 不剥夺条件：指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。
> - 环路等待条件：指在发生死锁时，必然存在一个进程——资源的环形链，即进程集合 {P0，P1，P2，•••，Pn} 中的 P0 正在等待一个 P1 占用的资源；P1 正在等待 P2 占用的资源，……，Pn 正在等待已被 P0 占用的资源。

下列方法有助于最大限度地降低死锁：

> - 设置获得锁的超时时间。通过超时，至少保证最差最差最差情况下，可以有退出的口子。
> - 按同一顺序访问对象。
> - 避免事务中的用户交互。
> - 保持事务简短并在一个批处理中。
> - 使用低隔离级别。
> - 使用绑定连接。

### 6、MySQL 中 InnoDB 引擎的行锁的实现

InnoDB 是基于索引来完成行锁。例如：`SELECT * FROM tab_with_index WHERE id = 1 FOR UPDATE` 。`FOR UPDATE` 可以根据条件来完成行锁锁定，并且 id 是有索引键的列,如果 id 不是索引键那么 InnoDB 将完成表锁，并发将无从谈起。

**注意：**

> - 1.**行锁必须有索引才能实现，否则会自动锁全表，那么就不是行锁了。**

> - 2.两个事务不能锁同一个索引，例如：

```
事务A先执行:
select math from zje where math>60 for update;

事务B再执行:
select math from zje where math<60 for update;
```

这样的话，事务B是会阻塞的。如果事务B把math索引换成其他索引就不会阻塞，但注意，**换成其他索引锁住的行不能和math索引锁住的行有重复。**

# 四、MySQL查询执行顺序

以下是常见的SQL语句查询逻辑执行顺序，序号则为实际执行顺序：

```
(7)     SELECT
(8)     DISTINCT <select_list>
(1)     FROM <left_table>
(3)     <join_type> JOIN <right_table>
(2)     ON <join_condition>
(4)     WHERE <where_condition>
(5)     GROUP BY <group_by_list>
(6)     HAVING <having_condition>
(9)     ORDER BY <order_by_condition>
(10)    LIMIT <limit_number>
```

### 执行顺序简介

在SQL语句的执行过程中，每一步都会产生一个虚拟表（VirtualTable，简称VT），用来保存SQL语句的执行结果，以下是上述SQL的执行顺序。

### 执行FROM语句

第一步，执行FROM语句。我们首先需要知道最开始从哪个表开始的，这就是FROM告诉我们的。经过FROM语句对两个表执行笛卡尔积，会得到一个虚拟表，暂且叫VT1。总共有——table1的记录条数 * table2的记录条数——条记录，这就是VT1的结果。

### 执行ON过滤

执行完笛卡尔积以后，接着就进行ON join_condition条件过滤，比如ON a.customer_id = b.customer_id，根据ON中指定的条件，去掉那些不符合条件的数据，得到VT2表。

### 添加外部行（外联结）

这一步只有在连接类型为OUTER JOIN时才发生，如LEFT OUTER JOIN（左连接）、RIGHT OUTER JOIN（右连接）和FULL OUTER JOIN（经过测试Mysql不支持该连接方式）。大多数时候会省略OUTER关键字。 添加外部行的工作就是在VT2表的基础上添加保留表中被过滤条件过滤掉的数据，非保留表中的数据被赋予NULL值，最后生成虚拟表VT3。【比如left join会保留驱动表中未关联到的数据】

### 执行WHERE过滤

对添加外部行得到的VT3进行WHERE过滤，只有符合 where_condition 的记录才会输出到虚拟表VT4中。

### 执行GROUP BY分组

上面得到的虚拟表还没有经过聚合分组，GROU BY子句主要是对使用WHERE子句得到的虚拟表进行分组操作。得到的内容会存入虚拟表VT5中，此时，我们就得到了一个VT5虚拟表，接下来的操作都会在该表上完成。

### 执行HAVING过滤

这里需要注意的是到目前为止已经有了三种过滤，ON、WHERE和HAVING，三者在执行时间段上是有严格区别的，HAVING子句主要和GROUP BY子句配合使用，对分组得到的VT5虚拟表进行条件过滤，然后得到虚拟表VT6。

### SELECT列表

从虚拟表VT6中选择出我们需要的内容，生成虚拟表VT7。

### 执行DISTINCT子句

如果在查询中指定了DISTINCT子句，则会创建一张内存临时表（如果内存放不下，就需要存放在硬盘了）。这张临时表的表结构和上一步产生的虚拟表VT7是一样的，不同的是对进行DISTINCT操作的列增加了一个唯一索引，以此来除重复数据。

### 执行ORDER BY子句

对虚拟表中的内容按照指定的列进行排序，然后返回一个新的虚拟表，上述结果会存储在VT8中。

### 执行LIMIT子句

LIMIT子句从上一步得到的VT8虚拟表中选出从指定位置开始的指定行数据。

# 五、MySQL 优化

## 1、mysql服务器运行缓慢的情况下怎么缓解服务器压力

### a、第一步 检查系统的状态

通过操作系统的一些工具检查系统的状态，比如CPU、内存、交换、磁盘的利用率，根据经验或与系统正常时的状态相比对，有时系统表面上看起来看空闲，这也可能不是一个正常的状态，因为cpu可能正等待IO的完成。除此之外，还应观注那些占用系统资源(cpu、内存)的进程。 1.1 使用sar来检查操作系统是否存在IO问题 1.2 使用vmstat监控内存 cpu资源 1.3 磁盘IO问题，处理方式：做raid10提高性能 1.4 网络问题，telnet一下MySQL对外开放的端口，如果不通的话，看看防火墙是否正确设置了。另外，看看MySQL是不是开启了skip-networking的选项，如果开启请关闭。

### b、第二步 检查mysql参数

```
2.1 max_connect_errors
2.2 connect_timeout
2.3 skip-name-resolve
2.4 slave-net-timeout=seconds
2.5 master-connect-retry
```

### c. 第三步 检查mysql 相关状态值

- 关注连接数
- 关注下系统锁情况
- 关注慢查询（slow query）日志

### 2.数据库的设计：尽量把数据库设计的占更小的磁盘空间.

1) 尽可能使用更小的整数类型.(mediumint就比int更合适).
2) 尽可能的定义字段为not null,除非这个字段需要null.
3) 如果没有用到变长字段的话比如varchar,那就采用固定大小的纪录格式比如char.
4) 表的主索引应该尽可能的短.这样的话每条纪录都有名字标志且更高效.
5) 只创建确实需要的索引。索引有利于检索记录，但是不利于快速保存记录。如果总是要在表的组合字段上做搜索，那么就在这些字段上创建索引。索引的第一部分必须是最常使用的字段.如果总是需要用到很多字段，首先就应该多复制这些字段，使索引更好的压缩。
6) 所有数据都得在保存到数据库前进行处理。
7) 所有字段都得有默认值。
8) 在某些情况下,把一个频繁扫描的表分成两个速度会快好多。在对动态格式表扫描以取得相关记录时，它可能使用更小的静态格式表的情况下更是如此。

### 3.系统的用途

1) 尽量使用长连接.
2) explain复杂的SQL语句。
3) 如果两个关联表要做比较话，做比较的字段必须类型和长度都一致.
4) LIMIT语句尽量要跟order by或者 distinct.这样可以避免做一次full table scan.
5) 如果想要清空表的所有纪录,建议用truncate table tablename而不是delete from tablename.
6) 能使用STORE PROCEDURE 或者 USER FUNCTION的时候.
7) 在一条insert语句中采用多重纪录插入格式.而且使用load data infile来导入大量数据，这比单纯的insert快好多.
8) 经常OPTIMIZE TABLE 来整理碎片.
9) 还有就是date 类型的数据如果频繁要做比较的话尽量保存在unsigned int 类型比较快。

### 4.系统的瓶颈

1) 磁盘搜索。并行搜索,把数据分开存放到多个磁盘中，这样能加快搜索时间.
2) 磁盘读写(IO)。可以从多个媒介中并行的读取数据。
3) CPU周期。数据存放在主内存中.这样就得增加CPU的个数来处理这些数据。
4) 内存带宽。当CPU要将更多的数据存放到CPU的缓存中来的话,内存的带宽就成了瓶颈.

## 六、MySQL -- 多版本并发控制（MVCC）

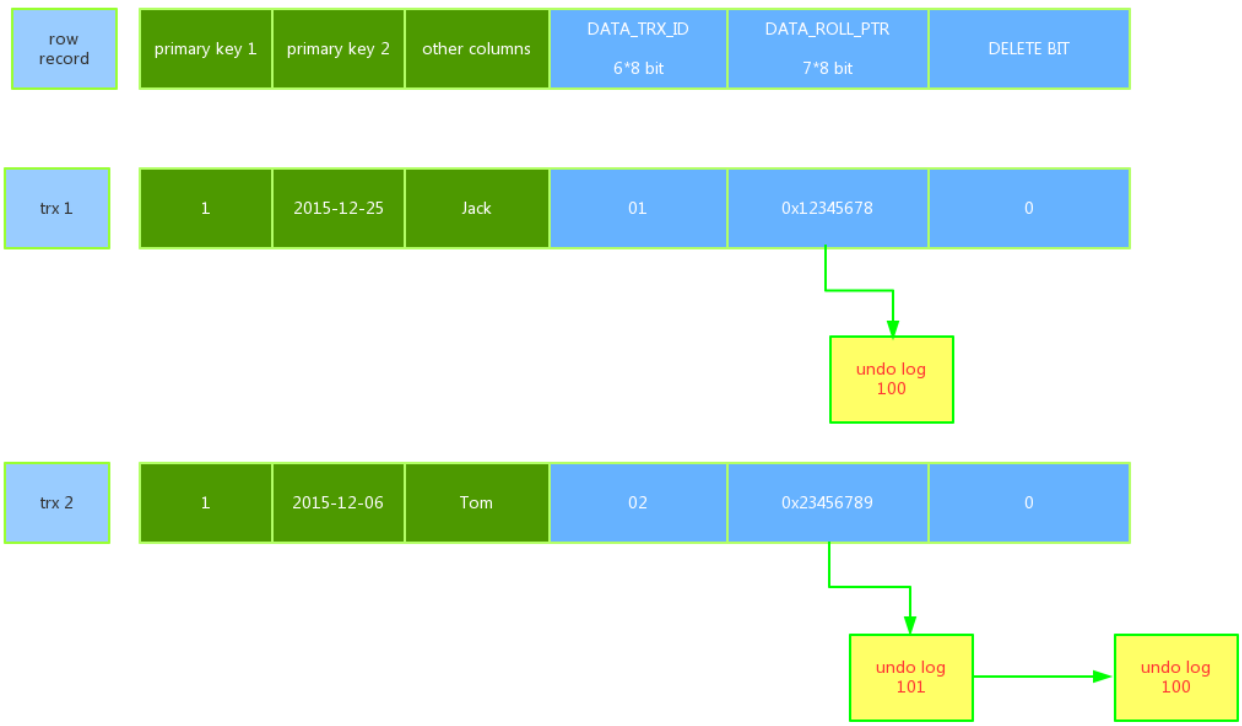
多版本并发控制（MVCC），是一种用来**解决读-写冲突**的无锁并发控制，它使得大部分支持行锁的事务引擎，不再单纯的使用行锁来进行数据库的并发控制，取而代之的是把数据库的行锁与行的多个版本结合起来，只需要很小的开销,就可以实现非锁定读，避免了脏读和不可重复读，从而大大提高数据库系统的并发性能。

Innodb MVCC主要是为Repeatable-Read事务隔离级别做的。在此隔离级别下，A、B客户端所示的数据相互隔离，互相更新不可见

了解Innodb的行结构、Read-View的结构对于理解Innodb mvcc的实现由重要意义

Innodb存储的最基本row中包含一些额外的存储信息 `DATA_TRX_ID` ， `DATA_ROLL_PTR` ， `DB_ROW_ID` ， `DELETE BIT` 。

> - 6字节的DATA_TRX_ID 标记了最新更新这条行记录的transaction id，每处理一个事务，其值自动+1。
> - 7字节的DATA_ROLL_PTR 指向当前记录项的rollback segment的undo log记录，找之前版本的数据就是通过这个指针。
> - 6字节的DB_ROW_ID，当由innodb自动产生聚集索引时，聚集索引包括这个DB_ROW_ID的值，否则聚集索引中不包括这个值.，这个用于索引当中。
> - DELETE BIT位用于标识该记录是否被删除，这里的不是真正的删除数据，而是标志出来的删除。真正意义的删除是在commit的时候




具体的执行过程:

**begin->用排他锁锁定该行->记录redo log->记录undo log->修改当前行的值，写事务编号，回滚指针指向undo log中的修改前的行.**

上述过程确切地说是描述了UPDATE的事务过程，其实undo log分insert和update undo log，因为insert时，原始的数据并不存在，所以回滚时把insert undo log丢弃即可，而update undo log则必须遵守上述过程.

### 分别以select、 delete、 insert、 update语句来说明:

SELECT

Innodb检查每行数据，确保他们符合两个标准：

1、InnoDB只查找版本早于当前事务版本的数据行(也就是数据行的版本必须小于等于事务的版本)，这确保当前事务读取的行都是事务之前已经存在的，或者是由当前事务创建或修改的行。

2、行的删除操作的版本一定是未定义的或者大于当前事务的版本号，确定了当前事务开始之前，行没有**被删除**。

符合了以上两点则返回查询结果。

**INSERT**

InnoDB为每个新增行记录当前系统版本号作为创建ID。

**DELETE**

InnoDB为每个删除行的记录当前系统版本号作为行的删除ID。

**UPDATE**

InnoDB复制了一行。这个新行的版本号使用了系统版本号。它也把系统版本号作为了删除行的版本。

**说明:??????????????????????????????**

> insert操作时 “创建时间”=DB_ROW_ID，这时，“删除时间 ”是未定义的；
>
> update时，复制新增行的“创建时间”=DB_ROW_ID，删除时间未定义，旧数据行“创建时间”不变，删除时间=该事务的DB_ROW_ID；
>
> delete操作，相应数据行的“创建时间”不变，删除时间=该事务的DB_ROW_ID；
>
> select操作对两者都不修改，只读相应的数据

## MVCC的总结:

上述更新前建立undo log，根据各种策略读取时非阻塞就是MVCC，undo log中的行就是MVCC中的多版本，这个可能与我们所理解的MVCC有较大的出入，一般我们认为MVCC有下面几个特点：

- 每行数据都存在一个版本，每次数据更新时都更新该版本
- 修改时Copy出当前版本随意修改，各个事务之间无干扰
- 保存时比较版本号，如果成功（commit），则覆盖原记录；失败则放弃copy（rollback）

就是每行都有版本号，保存时根据版本号决定是否成功，听起来含有乐观锁的味道，而Innodb的实现方式是：

- 事务以排他锁的形式修改原始数据
- 把修改前的数据存放于undo log，通过回滚指针与主数据关联
- 修改成功（commit）啥都不做，失败则恢复undo log中的数据（rollback）

二者最本质的区别是，当修改数据时是否要排他锁定，如果锁定了还算不算是MVCC?

Innodb的实现真算不上MVCC，因为并没有实现核心的多版本共存，undo log中的内容只是串行化的结果，记录了多个事务的过程，不属于多版本共存。但理想的MVCC是难以实现的，当事务仅修改一行记录使用理想的MVCC模式是没有问题的，可以通过比较版本号进行回滚；但当事务影响到多行数据时，理想的MVCC据无能为力了。

比如，如果Transaciton1执行理想的MVCC，修改Row1成功，而修改Row2失败，此时需要回滚Row1，但因为Row1没有被锁定，其数据可能又被Transaction2所修改，如果此时回滚Row1的内容，则会破坏Transaction2的修改结果，导致Transaction2违反ACID。

理想MVCC难以实现的根本原因在于企图通过乐观锁代替二段提交。修改两行数据，但为了保证其一致性，与修改两个分布式系统中的数据并无区别，而二提交是目前这种场景保证一致性的唯一手段。二段提交的本质是锁定，乐观锁的本质是消除锁定，二者矛盾，故理想的MVCC难以真正在实际中被应用，Innodb只是借了MVCC这个名字，提供了读的非阻塞而已。

# 参考资料


http://zouzls.github.io/2017/03/23/SQL%E6%9F%A5%E8%AF%A2%E4%B9%8B%E6%89%A7%E8%A1%8C%E9%A1%BA%E5%BA%8F%E8%A7%A3%E6%9E%90/

https://www.cnblogs.com/chenpingzhao/p/5065316.html