MySQL 索引应用实践

1 MySQL 索引简介

1.1索引诞生的背景是怎样的？

假如数据库表中只有 10 条记录，我们可以一条条的进行查询。假如有 500 万条记录呢，从假如还是一条条去查询可能需要的时间就会比较长，此时索引就诞生了。

1.2我们所说的索引是什么？

官方给出的解释是，索引(Index)是一种数据结构。通俗的说，假如我们将一本书看成是一张表，那么索引就相当于书的目录，为了方便查找书中的内容，可以通过对内容建立索引形成目录。通过目录可以快速定位要查找的内容。

对于数据库而言，可以维护一个满足特定查找算法的数据结构，这些结构以某种方式引用数据，这样就可以在这些数据结构上实现数据的快速查找。这就是索引。

1.3索引有什么特点呢？

索引(Index)的优点是可以加大数据的检索速度。缺点是索引需要占用物理空间（默认是以数据页进行存储），而且对表中的数据进行增加、删除和修改时，索引也要动态维护，这样会降低增/改/删的执行效率。

如果没有索引，我们从数据库表中查询记录时，必须从第一行开始，读取整个表进行查找，表越大，成本就越高。如果表中有相关列的索引，就可以快速确定要在数据文件中间查找的位置，而不必查看所有数据，比按顺序读取每一行快得多。

MySQL 索引对表中指定列进行排序后可以另外保存，保存的内容中包含着对数据表里所有记录的引用指针，用于快速查找具有特定值的行。但不是所有表上都可以建索引，要根据表使用的存储引擎来看，有的存储引擎支持建索引，有的不支持。

1.4索引的应用原则有哪些？

索引使用总有一些原则：

1） 对经常在 where、连接条件中出现的列考虑创建索引。
2） 对选择性比较好的列必要时建索引。

比如用户表中，身份证的列具有不同值，选择性很好，索引被使用时特别高效；姓名列，选择性较好，索引被使用时也比较高效；性别列，只含有男和女，选择性很差，对此列建索引就没有多大用处。

3） 不要过度创建索引。

索引不是越多越好，每个索引都要占用磁盘空间，并会降低 DML 操作的性能。另外 MySQL 在生成执行计划时，过多的索引也会加重优化器的工作，甚至可能干扰优化器选择不到最好的索引。

2 MySQL 索引类型分析

2.1索引是怎样分类的？

说到索引的分类，可从三个维度进行分析：

- **逻辑应用维度**

1） 主键索引：主键索引是一种特殊的唯一索引，不允许有空值（唯一索引允许值为空）。
2） 普通索引或者单列索引：每个索引只包含单个列，一个表可以有多个单列索引。
3） 多列索引（复合索引、联合索引）：复合索引指多个字段上创建的索引，只有在查询条件中使用了创建索引时的第一个字段，索引才会被使用。使用复合索引时遵循最左前缀集合
4） 唯一索引：表中字段的值不能出现重复的。
5） 空间索引：空间索引是对空间数据类型的字段建立的索引，空间索引只能在存储引擎为 MYISAM 的表中创建。

- **物理存储维度**

1） 聚簇索引(clustered index)：在 InnoDB 中一张表只有一个聚簇索引(一般可以理解为主键索引)，索引和数据是在一起的。
2） 非聚簇索引(non-clustered index)：也叫辅助索引（secondary index），非聚簇索引是索引和数据分离的。

- **数据存储结构维度**

  1） B+树索引。（最重要）
  2） Hash 索引。
  3） Full-Text 全文索引。
  4） R-Tree 索引。

2.2如何创建索引？

2.2.1 普通索引

普通索引为 MySQL 中的基本索引类型，没有什么限制，允许在定义索引的列中插入重复值和空值，纯粹为了查询数据更快一点。

创建普通索引的方式如下：

```
CREATE TABLE tablename (…,INDEX[索引名字](字段名));
CREATE INDEX 索引的名字 ON tablename(字段名);
ALTER TABLE table_name ADD INDEX index_name (column_name);
```

2.2.2 唯一索引

唯一索引与普通索引类似，只是索引列中的值必须是唯一的，但是允许为空值。

```
CREATE TABLE  tablename (…,UNIQUE[索引名字](字段名));
CREATE UNIQUE INDEX 索引的名字 ON tablename(字段名);
ALTER TABLE table_name ADD UNIQUE INDEX index_name (column_name);
```

2.2.3 主键索引

主键索引是一种特殊的唯一索引，索引列中的值必须是唯一的，并且不允许有空值。

```
CREATE TABLE table_name (…,PRIMARY KEY (字段名))
ALTER TABLE table_name ADD PRIMARY KEY (column_name);
```

2.2.4 组合索引

组合索引又称为复合索引，在表中的多个字段组合上创建的索引,组合索引的使用，需要遵循最左前缀原则（最左匹配原则）。一般情况下，建议使用组合索引代替单列索引（主键索引除外）

```
CREATE TABLE table_name (…,INDEX index_name (column1,column2,…))
CREATE INDEX index_name ON table_name  (column1,column2,…))
ALTER TABLE table_name ADD INDEX index_name (column1,column2,…);
```

说明：
1) 复合索引的字段是有顺序的，在查询使用时要按照定义时索引字段的顺序使用，例如 select * from test where name=xxx and phone=xxx;匹配(name,age)组合索引，不匹配(age,name)索引。
2) 如果表已经建立了(col1,col2)，就没有必要再单独建立(col1);

2.2.5 全文索引

只能在文本类型 CHAR,VARCHAR,TEXT 类型字段上创建全文索引。字段长度比较大时，如果创建普通索引，在进行 like 模糊查询时效率比较低，这时可以创建全文索引。
全文搜索时候，全文索引一般很少使用，数据量比较少或者并发度低的时候可以用。但是数据量大或者并发度高的时候一般是用专业的工具 lucene,es，solr。

```
CREATE TABLE table_name (…,FULLTEXT KEY index_name (column));
CREATE FULLTEXT INDEX index_name ON table_name (column);
ALTER TABLE `t_fulltext` ADD FULLTEXT INDEX
`idx_content`(`content`);
```

和常用的 like 模糊查询不同，全文索引有自己的语法格式，可以使用 MATCH() … AGAINST 语法执行全文搜索，例如：

```
SELECT * FROM t_fulltext WHERE MATCH(content) AGAINST(“tedu”);//默认等值
```

```
SELECT * FROM t_fulltext WHERE match(name) against (‘aaa* in Boolean
mode’)
```

说明：
1）在 MySQL5.6 以前的版本，只有 MyISAM 存储引擎支持全文索引，从 MySQL5.6 开始，MyISAM 和 InnoDB 存储引擎都已经支持。
2）全文索引必须在字符串、文本字段上建立。
3）全文索引的字段值必须在最小和最大字符之间时才会生效。
4）全文索引的字段值会进行切词处理，例如 b+aaa，切分成 b 和 aaa。
5）全文索引查询默认使用的是等值匹配，例如 a 匹配 a，不会匹配 ab,ac。如果希望匹配可以在布尔模式下搜索 a*;

如何查询全文索引的规则？(innodb 存储引擎默认是大于 3 个字符小于 84 个字符使用全文)

```
show variables like ‘%ft%’
```

2.2.6 前缀索引

在文本类型如 CHAR,VARCHAR,TEXT 类列上创建索引时，可以指定索引列的长度，但是数值类型不能指定。

```
ALTER TABLE table_name ADD INDEX index_name (column1(length));
```

2.3如何查看索引？

删除索引之前我们可以基于如下方式查看索引

```
SHOW INDEX FROM table_name \G
```

2.4如何删除指定索引？

删除索引，例如：

```
DROP INDEX index_name ON table
```

3 MySQL 索引结构分析

3.1需求分析

我们知道正确应用索引、可以提高数据查询的性能。那我们日常工作的的数据查询可以将其分为两大类：

1． 等值查询：根据某个值查找数据。例如：

```
select * from t_user where age=76;
```

2． 范围查询：根据某个范围区间查找数据。例如

```
select * from t_user where age>=76 and age<=86;
```

我们在基于查询设计索引时，**需要考虑时间和空间两个因素：**
1） 在执行时间方面，我们希望通过索引，查询数据的时间尽可能小。
2） 在存储空间方面，我们希望索引不要消耗太多的内存空间和磁盘空间。

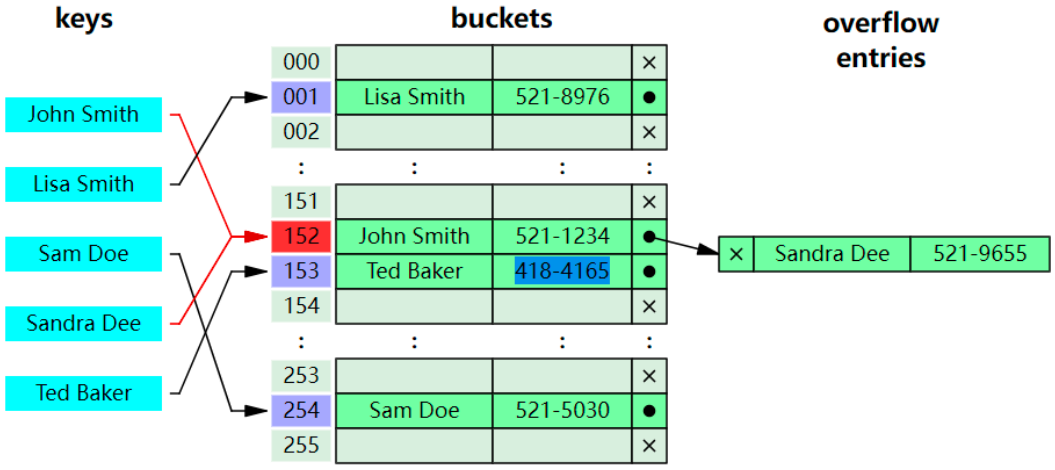
3.2数据结构分析

我们首先要知道一点，索引(Index)是在存储引擎(storage engine)层面实现的，不是所有的存储引擎，都支持所有的索引类型，即使是不同存储引擎支持同一索引类型，她们的实现和行为也可能会有差异。

常用的索引数据结构：Hash 表，二叉树，平衡二叉查找树（红黑树是一个近似平衡二叉树），B 树，B+树。

3.3Hash 表结构分析

Java 中的 HashMap 就是 Hash 表结构，以键值对的方式存储数据。例如：



我们使用 Hash 表存储表数据，Key 可以存储索引列，Value 可以存储行记录或者行磁盘地址。Hash 表在等值查询时效率很高，时间复杂度为 O(1)；但是不支持范围快速查找，范围查找时还是只能通过扫描全表方式。

数据库的 InnoDB 引擎提供了自适应索引，为了提高查询效率，InnoDB 存储引擎会监控表上各个索引页的查询，当 InnoDB 注意到某些索引值访问频繁时，会在内存中基于 B+Tree 索引再创建一个哈希索引，使得内存中的 B+Tree 索引具备哈希索引的功能，即能够快速定值访问频繁访问的索引页。

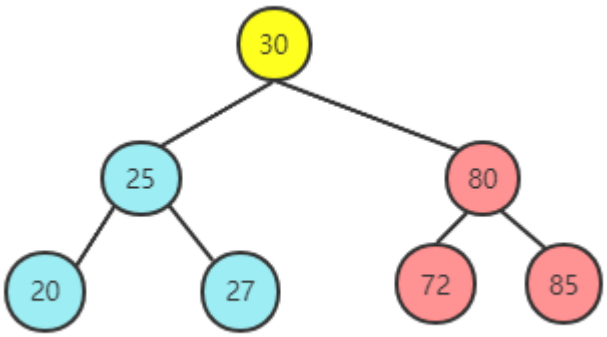
自适应哈希索引的建立使得 InnoDB 存储引擎自动根据索引页访问频率和模式自动的为某些热点页建立哈希索引来加速访问，另外 InnoDB 自适应 Hash 索引的功能，用户只能选择开启或关闭，服务进行人工干预

3.4Tree 结构索引分析

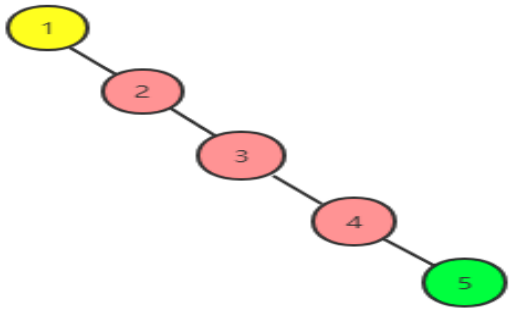
MySQL 中的 InnoDB 和 MyISAM，都使用 B+Tree 的数据结构，但如如何理解 B+Tree 结构，我们可以从二叉查找树进行分析。

3.4.1 二叉查找树

二叉查找树（Binary Search Trees），每个节点最多有 2 个分叉，左子树和右子树，数据顺序左小右大，也就是左子树的值小于根的值，右子树的值大于根的值。例如：

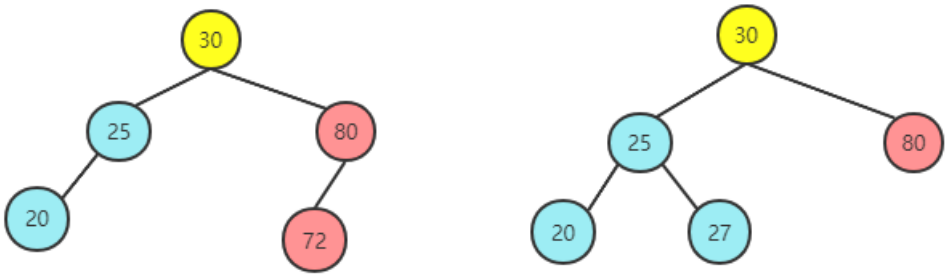


在上面的图中，假如查找 72，则只需要三次查找即可，查询效率可以得到明显提升。但问题来了，是不是任何数据都可以基于二叉查找树结构提高查询效率呢？答案是否定的，例如，基于表中的 id 自增主键值，构建的二叉查找树，这个树就会退化为了一个单项链表，其结构如图所示。假如此时我们查找 5 这个节点，就进行全表扫描，并不会提高查询效率。



3.4.2 平衡二叉树

平衡二叉树（Balanced binary search trees）是采用二分法思维，平衡二叉查找树除了具备二叉树的特点，最主要的特征是树的左右两个子树的层级最多相差 1。



平衡二叉树在插入、删除数据时，通过左旋/右旋操作保持二叉树的平衡，不会出现左子树很高、右子树很矮的情况。这样我们查询数据的时间复杂度就稳定了。

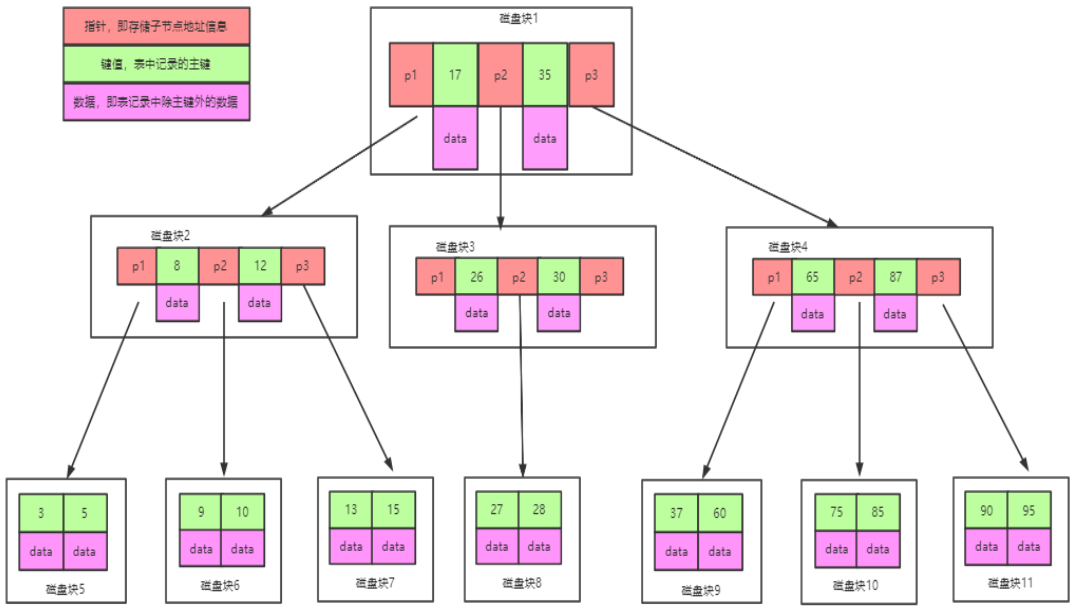
使用平衡二叉查找树，查询的性能接近于二分查找法，时间复杂度是 O(log2n)。查询

id=25，只需要两次 IO。

那我们使用 AVL 树作为索引是不是就可以了呢，答案是否定的。我们知道 MySQL 中数据是存储在磁盘上的，每次进行数据查询会将磁盘里的数据读取到内存中，对磁盘 io 是非常耗时的。在表数据量大时，查询性能就会很差。所以我们优化的重点就是尽量减少磁盘 IO 操作。访问二叉树的每个节点就会发生一次 IO，如果想要减少磁盘 IO 操作，就需要尽量降低树的高度。那如何降低树的高度呢？

3.4.3 B-树

为了解决平衡二叉树浪费磁盘空间以及 IO 次数过多的问题，我们在一个节点中多存储一些数据，之前我们放一个，现在我们放多个。由此，B 树就诞生了。注意这里的 B 树中的 B 代表平衡（balance），而不是二叉（binary）。例如：



B 树是一种多叉平衡查找树，一颗 m 阶 B-树，要么为空树，要么满足如下特性：

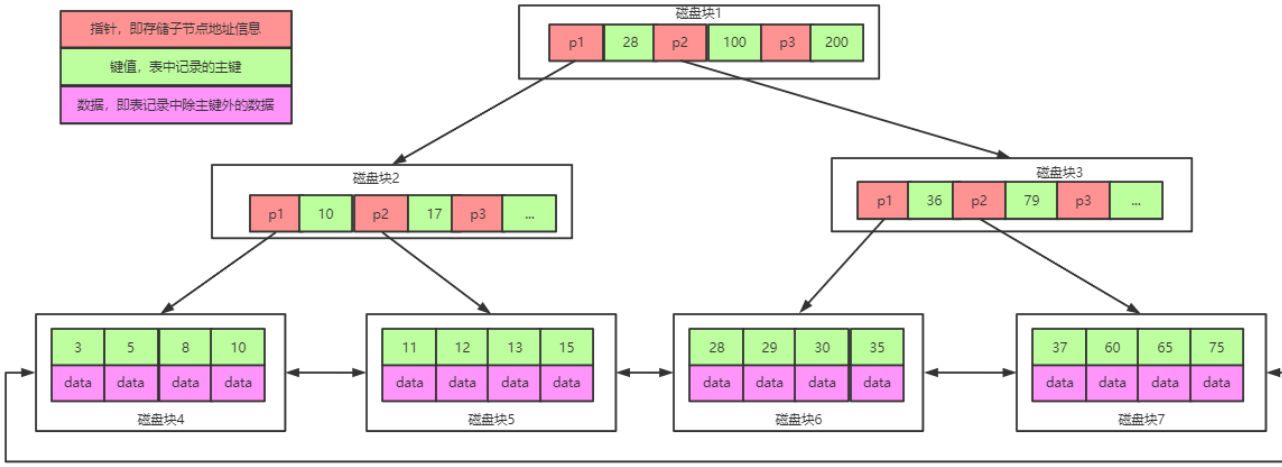
1) 树中每个节点最多有 m 棵子树(m 阶为树中节点的最大分支数)。
2) 若根节点不是叶子，则至少有两棵子树。
3) 所有非叶子节点包含 n 个关键字和 n+1 棵子树。
4) 所有叶子节点都在同一层。

说明，B 树不支持范围查询的快速查找，每次查询都需要从根节点进行多次遍历，查询效率有待提高。还有，如果 data 存储的是行记录，行的大小随着列数的增多，占空间会变大。这时，一个页中可存储的数据量就会变少，树相应就会变高，磁盘 IO 次数就会变大。

3.4.4 B+树(B-树 Plus)

在 B 树基础上，MySQL 在 B 树的基础上继续改造，使用 B+树构建索引。B+树和 B-树最主要的区别在于非叶子节点是否存储数据的问题。

1) 一棵 m 阶的 B+树每个节点最多可以有 m 个 key；
2) B+树的所有 key 都在叶子节点中有序排列；
3) B+树的所有非叶子节点都是为了查找到叶子节点。（非叶子节点不存储数据）
4) B+树的叶子节点处于同一层，相邻的叶子节点通过双向链表进行连接。（与 B-树不同）



为什么 Mysql 使用 B+树而不是 B 树作为索引？

1) B 树的叶子节点和非叶子节点都会存储数据，这样会导致非叶子节点可以保存的指针数量减少，数据量比较大时，树的高度可能就会比较大，树越高，导致磁盘 IO 的次数就会变多，查询的性能就会变低。
2) B+树非叶子节点不存储数据，只存储索引 key，这样可以存储 key 就会比较多，key 越多可以分的叉就越多，树的高度就不会太高，这样磁盘 io 就会比较少，性能就会比较好。
3) B+树对于新增与修改节点的效率也是比较高的，这与 B 树相同；
4) 在叶子节点引入了链表，增加范围查询的效率。

3.5聚簇索引和非聚簇索引

聚簇索引和非聚簇索引：

B+Tree 的叶子节点存放主键索引值和行记录就属于聚簇索引；如果索引值和行记录分开存放就属于非聚簇索引。

主键索引和辅助索引：

B+Tree 的叶子节点存放的是主键字段值就属于主键索引。如果存放的是非主键值就属于辅助索引（二级索引）。

在 InnoDB 引擎中，主键索引采用的就是聚簇索引结构存储。

1）聚簇索引（聚集索引）

聚簇索引是一种数据存储方式， InnoDB 的聚簇索引就是按照主键顺序构建 B+Tree 结构。 B+Tree 的叶子节点就是行记录，行记录和主键值紧凑地存储在一起。 这也意味着 InnoDB 的主键索引就是数据表本身，它按主键顺序存放了整张表的数据，占用的空间就是整个表数据量的大小。通常说的主键索引就是聚集索引。

InnoDB 的表要求必须要有聚簇索引。如果表定义了主键，则主键索引就是聚簇索引。
如果表没有定义主键，则第一个非空 unique 列作为聚簇索引，否则 InnoDB 会建一个隐藏的 row-id 作为聚簇索引。

2）辅助索引

InnoDB 辅助索引，也叫作二级索引，是根据索引列构建 B+Tree 结构。但在 B+Tree 的叶子节点中只存了索引列和主键的信息。二级索引占用的空间会比聚簇索引小很多， 通常创建辅助索引就是为了提升查询效率。一个表 InnoDB 只能创建一个聚簇索引，但可以创建多个辅助索引。

与 InnoDB 表存储不同，MyISAM 数据表的索引文件和数据文件是分开的，被称为非聚簇索引结构。

4 索引分析与优化

4.1 Explain 是什么？

MySQL 提供了一个 Explain 命令，它可以对 select 语句进行分析，并输出 select 执行时的详细信息，开发人员可以基于这些信息进行有针对性的优化，例如：

```
mysql> explain select * from employees where employee_id<100 \G;
```

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: employees
   partitions: NULL
         type: range
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: NULL
         rows: 1
     filtered: 100.00
        Extra: Using where
1 row in set, 1 warning (0.00 sec)
```

其中：

- select_type 表示查询的类型。常用的值如下：

1) SIMPLE ： 表示查询语句不包含子查询或 union
2) PRIMARY：表示此查询是最外层的查询
3) UNION：表示此查询是 union 的第二个或后续的查询
4) DEPENDENT UNION：union 中的第二个或后续的查询语句，使用了外面查询结果
5) UNION RESULT：union 的结果
6) SUBQUERY：SELECT 子查询语句
7) DEPENDENT SUBQUERY：SELECT 子查询语句依赖外层查询的结果。

- type 表示查询数据的方式。

type 是一个比较重要的一个属性，通过它可以判断出查询是全表扫描还是基于索引的部分扫描。常用属性值如下，从上至下效率依次增强。

1) ALL：表示全表扫描，性能最差。
2) index：表示基于索引的全表扫描，先扫描索引再扫描全表数据。
3) range：表示使用索引范围查询。使用 >、>=、<、<=、in 等等。
4) ref：表示使用非唯一索引进行单值查询。
5) eq_ref：一般情况下出现在多表 join 查询，表示前面表的每一个记录，都只能匹配后面表的一行结果。
6) const：表示使用主键或唯一索引做等值查询，常量查询。
7) NULL：表示不用访问表，速度最快。

- possible_keys

表示查询时能够使用到的索引。注意并不一定会真正使用，显示的是索引名称。

- key

表示查询时真正使用到的索引，显示的是索引名称。

- rows

MySQL 查询优化器会根据统计信息，估算 SQL 要查询到结果需要扫描多少行记录。原则上 rows 是越少效率越高，可以直观的了解到 SQL 效率高低。

- key_len

表示查询使用了索引的字节数量。可以判断是否全部使用了组合索引。

key_len 的计算规则如下：

1） 字符串类型

字符串长度跟字符集有关：latin1=1、gbk=2、utf8=3、utf8mb4=4
char(n)：n*字符集长度
varchar(n)：n * 字符集长度 + 2 字节

2） 数值类型

TINYINT：1 个字节
SMALLINT：2 个字节
MEDIUMINT：3 个字节
INT、FLOAT：4 个字节
BIGINT、DOUBLE：8 个字节

3） 时间类型

DATE：3 个字节
TIMESTAMP：4 个字节
DATETIME：8 个字节

4） 字段属性

NULL 属性占用 1 个字节，如果一个字段设置了 NOT NULL ，则没有此项。

- Extra

Extra 表示很多额外的信息，各种操作会在 Extra 提示相关信息，常见几种如下：

1) Using where

表示查询需要通过索引回表查询数据。

2) Using index

表示查询需要通过索引，索引就可以满足所需数据。

3) Using filesort

表示查询出来的结果需要额外排序，数据量小在内存，大的话在磁盘，因此有 Using filesort 建议优化。

4) Using temprorary

查询使用到了临时表，一般出现于去重、分组等操作。

4.2 什么是回表查询？

我们知道 InnoDB 索引有聚簇索引和辅助索引。聚簇索引的叶子节点存储行记录，InnoDB 必须要有，而且只有一个。辅助索引的叶子节点存储的是主键值和索引字段值，通过辅助索引无法直接定位行记录，通常情况下，需要扫码两遍索引树。先通过辅助索引定位主键值，然后再通过聚簇索引定位行记录，这就叫做回表查询，它的性能比扫一遍索引树低。

4.3 什么是覆盖索引？

覆盖索引指的是在一棵索引树上就可以获取 SQL 所需要的所有数据，而不需要执行回表查询。在 MySQL 中， explain 输出结果的 Extra 字段为 Using index 时，能够触发索引覆盖。假如我们需要实现索引覆盖，可以考虑将被查询的字段，建立到组合索引。

4.4 什么是最左前缀匹配？

复合索引使用时遵循最左前缀原则，也就是最左优先，如果查询条件中最左边的列和复合索引最左边的列的顺序匹配，查询就会使用到索引，如果不匹配则索引将失效。例如你在（username,phone,email）三个字段上建立了索引，查询条件中也按此顺序进行定义的则会应用索引，假如查询条件中写的是 phone='1111' and email='t@tedu.cn'则会索引失效。

4.5有哪些常见问题？

- MySQL 在使用 like 模糊查询时，索引能不能起作用？

MySQL 在使用 Like 模糊查询时，索引是可以被使用的，只有把 % 字符写在后面才会使用到索引。

```
select * from tb_user where name like '%d%';    //不起作用
select * from tb_user where name like 'd%';     //起作用
select * from tb_user where name like '%d';     //不起作用
```

- 如果 MySQL 表的某一列含有 NULL 值，那么包含该列的索引是否有效？

对 null 值是否走索引要结合具体版本以及使用的引擎来确定，例如在 mysql5.7 存储引擎为 InnoDB 中， 我们看到两个 sql 都是走索引的，并不会因为 is null 而不走索引，所以字段为 null 并不会影响 sql 走索引。但是，还是建议无论索引列还是其他列，都设置成非 null，通过设置默认值解决 null 值问题。如果把索引列一些值设置为 null，也是允许的，但是写 sql 时候就要用到 is null 和 is not null 来进行筛选数据，这仅仅是语法的需要适配，但并不影响正确的走索引。

5 总结(Summary)

5.1重难点分析

- XXXXXXXXX

5.2FAQ 分析

- XXXXX