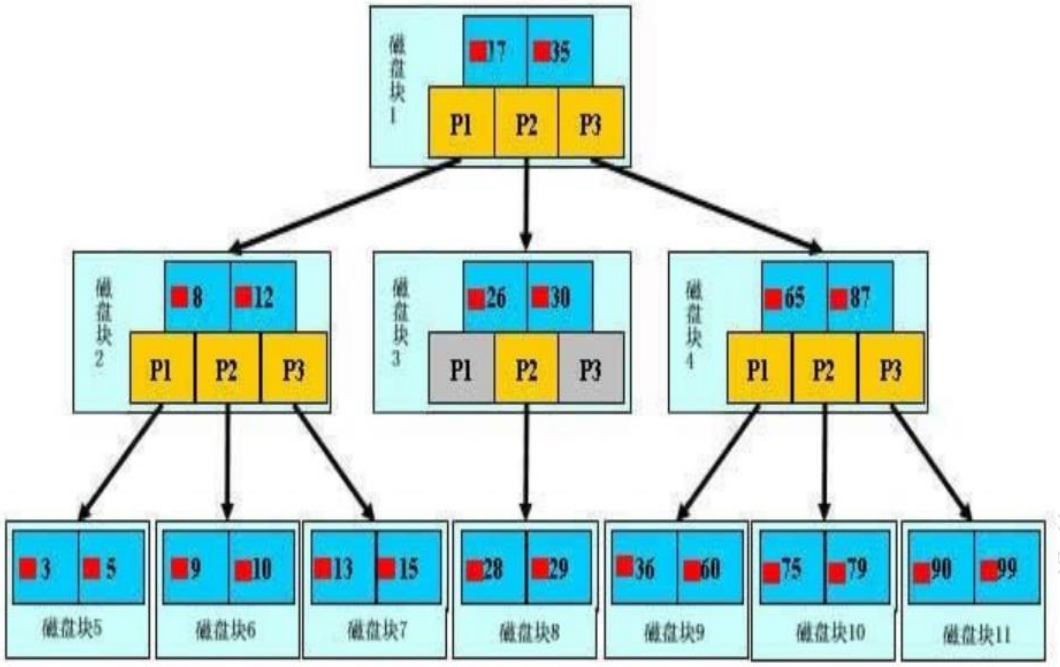
MySQL 优化

一． MySQL 运行缓慢原因分析

二． MySQL 索引

1. 什么索引

索引是 MySql 中一种文件。在索引文件中，根据一列或多列数据进行排序同时指向列所在的数据行或则数据行位置



2. 索引作用

索引可以避免在查询时，对表中所有数据行进行查询。

从而提供查询的效率。是提升查询速度首选方案

3.索引分类

1)MySQL 中根据存储引擎对索引进行分类

2)InnoDB 主键使用的是聚簇索引

3)MyISAM 所有的索引都是非聚簇索引

三．非聚簇索引

1．什么是【非聚簇索引】

使用二叉树将列中数据进行存储。在二叉树最后一级也就是叶子层存储的是数据对应的【主键值】和【数据所在行位置】，这中索引就是【非聚簇索引】

2.【非聚簇索引】工作原理

先创建一张表：

```
CREATE TABLE `user` (
`id` INT NOT NULL ,
`name` VARCHAR NOT NULL ,
`class` VARCHAR NOT NULL);
```

如果我们以 myIsam 引擎创建这个表。此时以 id 作为索引，那么这个索引就是【非聚簇索引】

此时我们可以执行如下操作:

```
select * from user where id = 1
```

会利用索引，先在索引树中快速检索到 id，但是要想取到 id 对应行数据，必须找到改行数据在硬盘中的存储位置，因此 MYISAM 引擎的索引 叶子页面上不仅存储了主键 id 还存储着 数据存储的地址信息。如图：

1	2	3	4
磁盘地址	磁盘地址	磁盘地址	磁盘地址

像这样的索引就称为非聚簇索引

四．聚簇索引

1.什么是【聚簇索引】

对于 非聚簇索引 来说，每次通过索引检索到所需行号后，还需要通过叶子上的磁盘地址去磁盘内取数据（回行）消耗时间。为了优化这部分回行取数据时间，InnoDB 引擎采用了聚簇索引。聚簇索引，即将数据存入索引叶子页面上。对于 InnoDB 引擎来说，叶子页面不再存该行对应的地址，而是直接存储数据：

3	4	5	6
name1	name2	name3	name4
class1	class2	class3	class4

五．聚簇索引与非聚簇索引区别

1.索引类型与存储引擎关系

1)innodb 可以创建【聚簇索引】和【非聚簇索引】

2)myisam 只能创建【非聚簇索引】

2.叶子节点存储内容

1)聚簇索引中叶子节点存储的是主键值以及对应数据行本身

2)非聚簇索引中叶子节点存储的主键值以及对应数据行的引用地址

3.关于自动增长

1)myisam 允许表中可以没有主键和索引

2)innodb 主动将主键作为聚簇索引，如果没有主键则找到一个唯一且非空约束字段上建立聚簇索引。如果没有。**Innodb** 会自动为当前表创建一个 **6** 个字节字段作为主键（用户看不到）。并根据这个主键建立索引。

4.关于自动增长

1) myisam 引擎的自动增长列必须是索引，如果是组合索引，自动增长可以不是第一列，他可以根据前面几列进行排序后递增。

2) innodb 引擎的自动增长列必须是索引，如果是组合索引也必须是组合索引的第一列。

5.关于 count 函数

1) myisam 保存有表的总行数，如果 select count(*) from table;会直接取出出该值

2) innodb 没有保存表的总行数，如果使用 select count(*) from table;就会遍历整个表，消耗相当大，但是在加了 wehre 条件后，myisam 和 innodb 处理的方式都一样。

6.关于全文索引

1) myisam 支持 FULLTEXT 类型的全文索引

2) innodb 不支持 FULLTEXT 类型的全文索引

7. delete from table

1) 使用这条命令时，innodb 不会从新建立表，而是一条一条的删除数据，在 innodb 上如果要清空保存有大量数据的表，最好不要使用这个命令。(推荐使用 truncate table)

2)myisam 会创建一个全新临时表，在临时表进行删除。然后使用临时表覆盖掉原始表

六．聚簇索引与非聚簇索引适用的场合

索引对于提升查询速度有非常大的帮助。但是同时对于数据更新效率以及存储空间和 CPU 使用率影响很大，使用时主要参考如下三个条件

1) 数据频繁被修改的表，不应该设置索引

2)不是经常作为检索字段，分组字段，排序字段的字段不应该设置索引

动作	使用聚簇索引	使用非聚簇索引
列经常被分组排序	应	应
返回某范围内的数据	应	不应
一个或极少不同值	不应	不应
小数目的不同值	应	不应
大数目的不同值	不应	应
频繁更新的列	不应	应
外键列	应	应
主键列	应	应
频繁修改索引列	不应	应

七．索引算法

索引算法决定索引数据存储方式以及定位数据方式。常用算法有 B+Tree 算法，Hash 算法，全文索引算法.

掌握算法，对与我们理解索引和正确使用索引是非常有帮助的。

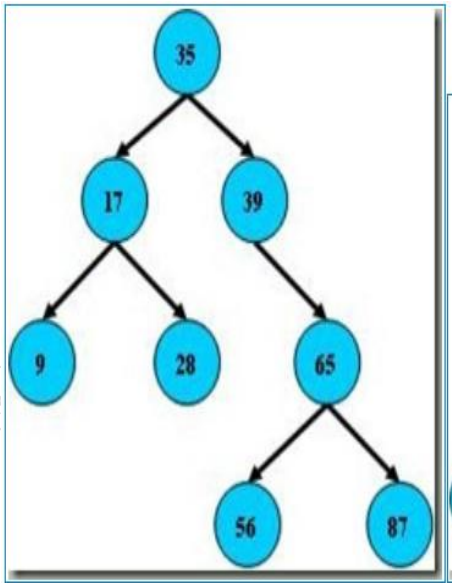
1. 二分查找算法

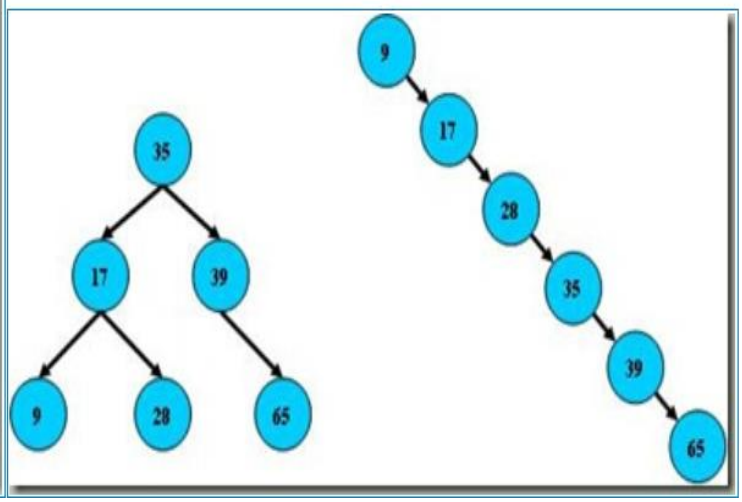
二分查找法的基本思想是，将记录排序（假如从小到大排序），然后采用跳跃式的方式进行查找，以有序数列的中点位置为比较对象，如果要找的元素小于该中点元素，那么查找左半部分，如果要找的元素大于该中点元素，那么久找右半部分。比如一组排好序的数：5 10 19 22 30 55 59 60 90， 如果我要查找 60 这个数字，那么先找 30，发现 30 小于 60，那么找 30 右半部分的中点 59，发现 59 还是小了，那么找 59 右边的数，从而找到了 60，这样通过不断二分把查找需要的时间以指数级进行下降

2. 平衡二叉树算法

平衡二叉树算法，是二分查找算法的升级版. 平衡二叉树，左子树的值总是小于根的值，右子树的值总是大于根的键值，因此可以通过中序遍历（以递归的方式按照左中右的顺序来访问子树），因此遍历以后得到的输出是 9、17、28、35、39、56、65、87。

这样，如果要查找键值为 28 的记录，先找到根，然后发现根大于 28，找左子树，发现左子树的根 17 小于 28，再找下一层右子树，然后找到 28。通过了 3 次查找找到了需要找的节点。但是如果二叉树节点分布非常不均匀，就像第二张图那样，那么如果要查找 39 这个节点的话，查找效率和顺序查找就差不多了，最差的结果就是查找 65，那么二叉搜索树就会完全退化成线性表。因此如果想要最大性能地构造一个二叉查找树，需要这颗二叉查找树是平衡的，平衡二叉树对于查找的性能是比较高的，但是不是最高的，只是接近最高的性能。要达到最好的性能，需要建立一颗最优二叉树，但是最优二叉树的建立和维护需要大量的操作，因此用平衡 二叉树就比较好。同时，平衡二叉树多用于内存结构对象中，因此维护他的开销相对较小。





3. B+Tree 算法

1) MySQL 读取索引原理

在 MySQL 中，将相近的数据保存在同一个存储块中。查询根据 KEY 找到存储块，然后再从存储块中找到对应的数据或则是数据行地址进行定位。

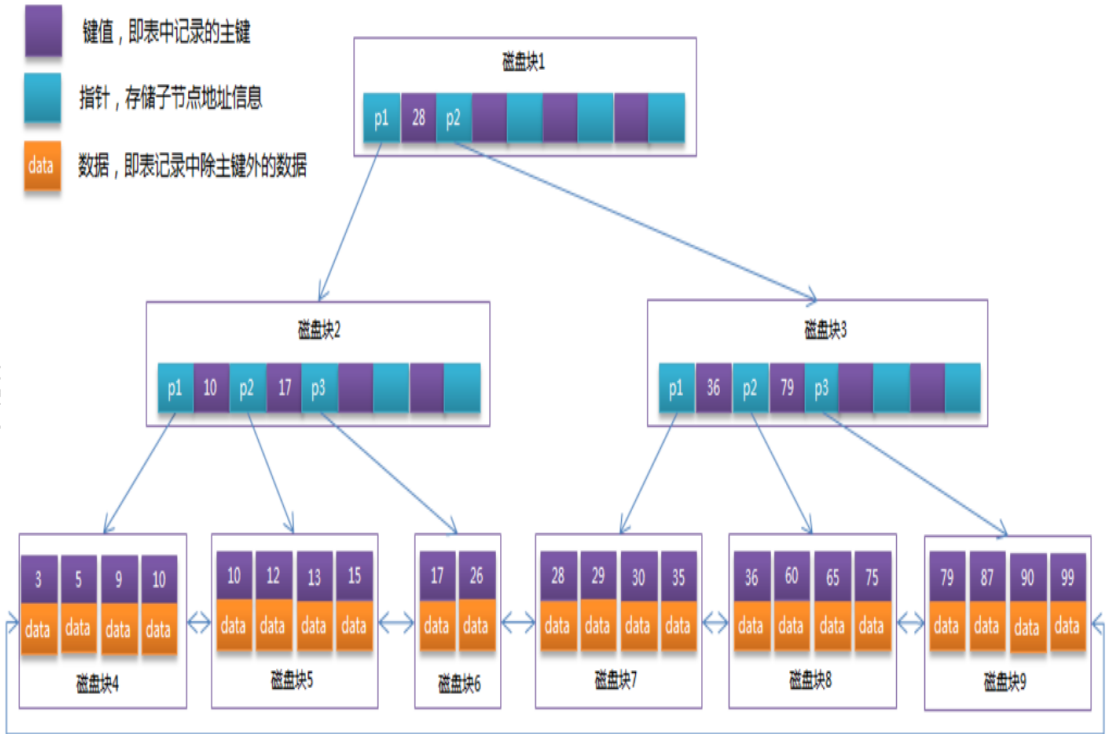
因此这里涉及到一个概念。就是磁盘的 I/O,也就是读取存储块的次数。这个次数越少，那么查询的速度就越快。

那么如何减少这个次数呢？最简单的方式就是让存储块存储的内容增加，这样就可以减少磁盘 I/O 读取次数，从而提高查询速度。而 B+Tree 算法就是为了这个目的而诞生的.

2) 什么是 B+Tree 算法

B-Tree 索引是 MySQL 数据库中使用最为频繁的索引类型，除了 Archive 存储引擎之外的其他所有的存储引擎都支持 B-Tree 索引。不仅仅在 MySQL 中是如此，实际上在其他的很多数据库管理系统中 B-Tree 索引也同样是作为最主要的索引类型，这主要

是因为 B-Tree 索引的存储结构在数据库的数据检索中有非常优异的表现，值得注意的是 mysql 中 innodb 和 myisam 引擎中的 B-tree 索引使用的是 B+tree（即每一个叶子节点都包含指向下一个叶子节点的指针，从而方便叶子节点的范围遍历，并且除叶子节点外其他节点只存储键值和指针）。



B+Tree 根节点和子节点只保存了键值和指针，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，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，并且 B+Tree 中的叶子节点比 B-tree 多存储了指向下一个叶子节点的指针，这样更方便叶子节点的范围遍历。

针对上图，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，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。

然后针对上图模拟下 where id=29 的具体过程：（首先 mysql 读取数据是以块（page）为单位的）。

首先根据根节点找到磁盘块 1，读入内存。【磁盘 I/O 操作第 1 次】

比较关键字 29 在区间（17,35），找到磁盘块 1 的指针 P2。

根据 P2 指针找到磁盘块 3，读入内存。【磁盘 I/O 操作第 2 次】

比较关键字 29 在区间（26,30），找到磁盘块 3 的指针 P2。

根据 P2 指针找到磁盘块 8，读入内存。【磁盘 I/O 操作第 3 次】

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。

4. hash 算法

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。对于每一行数据，存储引擎都会对所有的索引列计算一个哈希码（hash code），哈希码是一个较小的值，并且不同键值的行计算出来的哈希码也不一样。哈希索引将所有的哈希码存储在索引中，同时在哈希表中保存指向每个数据行的指针。

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，Memory 引擎也支持 B-Tree 索引。值的一提的是，Memory 引擎是支持非唯一哈希索引的，这在数据库世界里面是比较与众不同的。如果多个列的哈希值相同，索引会以链表的方式存放多个记录指针到同一个哈希条目中。

以下面的数据库结构为例：

```
CREATE TABLE testhash (

    fname VARCHAR(50) NOT NULL,
    lname VARCHAR(50) NOT NULL,
    KEY USING HASH(fname)

)ENGINE= MEMORY;
```

表中含有如下数据:

```
mysql > select * from testhash;
```

信息 结果1 概况 状态

fname	lname
Arjen	Lentz
Baron	Schwartz
Peter	Zaitsev
Vadim	Tkachenko

假设索引使用假想的哈希函数f()，它返回下面的值（以下为实例数据）：

f('Arjen') = 2323

f('Baron') = 7437

f('Peter') = 8784

f('Vadim') = 2458

则哈希索引的数据结构如下:

则哈希索引的数据结构如下：

槽 (Slot)	值 (Value)
2323	指向第1行的指针
2458	指向第4行的指针
7437	指向第2行的指针
8784	指向第3行的指针

需要注意的是，每个槽的编号是顺序的，但是数据行不是。现在，来看如下查询：

```
1  mysql > select lname from hashtest where fname = 'Peter';
```

MySQL 先计算‘Peter’的哈希值，并使用该值寻找对于的记录指针。因为 f('Peter') = 8784 ，所以 MySQL 在索引中查找 8784，可以找到指向第 3 行的指针，最后一步是比较第三方的值是否为‘Peter’，以确保就是要查找的行。

因为索引自身只需存储对于的哈希值，所以索引的结构十分紧凑，这也让哈希索引查找的速度非常快。然而，哈希索引也有它的限制：

1.哈希索引只包含哈希值和行指针，而不存储字段值，所以不能使用索引中的值来避免读取行。不过，访问内存中的行速度很快，所以对性能的影响并不明显。

2.哈希索引数据并不是按照索引值顺序存储的，所以无法用来进

行排序。

3.哈希索引也不支持部分索引列匹配查找。

4.哈希索引只支持等值比较查询，包括 = 、IN()、 <=>。也不支持范围查询。

因为这些限制，哈希索引只适用于某些特殊的场合。而一旦适合哈希索引，则它带来的性能提升将非常显著。

八． Explain 执行计划详解

1.执行计划介绍

通过 explain 查看查询语句执行的效率。在进行 SQL 优化的主要手段.通过 explain 可以查看如下信息

1) 查看表的加载顺序

2) 查看 sql 的查询类型

3) 哪些索引可能被使用，哪些索引又被实际使用了

4) 表之间的引用关系

5) 一个表中有多少行被优化器查询。

6) 其他额外的辅助信息

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	user_info	ALL	<NULL>	<NULL>	<NULL>	<NULL>	10	

2.执行计划之 id 属性

Id 属性是 mysql 对查询语句中提供查询序号。用于表示本次查询过程中加载表的顺序或则查询子句执行顺序.

Id 属性有三种情况

id 相同表示加载表的顺序是从上到下

id 不同 id 值越大，优先级越高，越先被执行

id 有相同，也有不同，同时存在。id 相同的可以认为是一组，从上往下顺序执行；在所有的组中，id 的值越大，优先级越高，越先执行。

1) id 的属性相同表示加载表的顺序是从上到下

```
EXPLAIN SELECT DNAME ,ENAME
FROM DEPT LEFT JOIN EMP
ON DEPT.DEPTNO = EMP.DEPTNO
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	DEPT	ALL	<NULL>	<NULL>	<NULL>	<NULL>	4	
1	SIMPLE	EMP	ALL	<NULL>	<NULL>	<NULL>	<NULL>	14	

2) id 值越大，优先级越高

```
EXPLAIN SELECT ENAME, JOB, SAL
FROM EMP
WHERE DEPTNO
IN (  SELECT DEPTNO FROM DEPT)
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	EMP	ALL	<NULL>	<NULL>	<NULL>	<NULL>	14	Using where
2	DEPENDENT SU	DEPT	unique_subc	PRIMARY	PRIMARY	4	func	1	Using index

3) id 有相同，也有不同，同时存在

```
explain  select ename,dname
FROM  EMP JOIN ( select deptno,dname from dept group by dname) e
ON emp.DEPTNO = e.deptno
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	ALL	<NULL>	<NULL>	<NULL>	<NULL>	4	
1	PRIMARY	EMP	ALL	<NULL>	<NULL>	<NULL>	<NULL>	14	Using where; Usir
2	DERIVED	dept	ALL	<NULL>	<NULL>	<NULL>	<NULL>	4	Using temporary;

3. 执行计划之 select_type

对当前查询语句中的查询类型进行判断.

1) Simple

表示当前查询语句是一个简单查询语句。不包含子查询，不包含联合查询，不包含连接查询

```
EXPLAIN SELECT ENAME, JOB, SAL
FROM EMP
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	EMP	ALL	<NULL>	<NULL>	<NULL>	<NULL>	14	

2) Primary

如果执行的是一个包含子查询的查询，或则是一个联合查询。Primary 指向的外部查询语句或则是联合查询中的第一个子查询语句

```
EXPLAIN
SELECT empno,ename  FROM EMP
union
select deptno,dname from dept
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	EMP	ALL	<NULL>	<NULL>	<NULL>	<NULL>	14	
2	UNION	dept	ALL	<NULL>	<NULL>	<NULL>	<NULL>	4	
	UNION RESULT	<union1,2>	ALL	<NULL>	<NULL>	<NULL>	<NULL>	<NULL>	

3) DEPENDENT SUBQUERY

表示当前查询语句是一个子查询。并且执行条件依赖与外部查询提供的条件.

```
EXPLAIN SELECT ename,job,sal,
( select max(sal) from emp e2 where e2.job=emp.job)
FROM EMP
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	EMP	ALL	<NULL>	<NULL>	<NULL>	<NULL>	14	
2	DEPENDENT SUBQUERY	e2	ALL	<NULL>	<NULL>	<NULL>	<NULL>	14	Using where

4) SUBQUERY

表示当前查询是一个子查询。并且这个子查询在执行时不需要得到外部查询的帮助.

```
EXPLAIN SELECT ename, job, sal,
(select count(*)  from dept)
FROM EMP
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	EMP	ALL	<NULL>	<NULL>	<NULL>	<NULL>	14	
2	SUBQUERY	dept	index	<NULL>	PRIMARY	4	<NULL>	4	Using index

4. 执行计划之 possible_keys

表示当前查询语句执行时可能用到的索引有哪些，在 possible_keys 可能出现多个索引，但是这些索引未必在本次查询使用到

5. 执行计划之 key

表示当前查询语句真实使用的索引名称.如果这个字段为 null.则有两中可能.一个是当前表中没有索引。二是当前表有索引但是失效了.

6. 执行计划之 key_len

如果本次查询使用了索引。则 key_len 内容不为空。

表示当前索引字段存储内容最大长度。这个长度不是精准值。只是 MySQL 估计的值。这个值越大越精准。在能得到相同结果时，这个值越小那么查询速度越快

7. 执行计划之 type

Type 属性描述 MySQL 对本次查询的评价.是执行计划中的一个重要属性。查询语句执行效率从高到底的顺序依次是.

- *NULL*：无需访问表或者索引，比如获取一个索引列的最大值或最小值。
- *system/const*：当查询最多匹配一行时，常出现于 where 条件是＝的情况。system 是 const 的一种特殊情况，既表本身只有一行数据的情况。
- *eq_ref*：关联查询时，根据唯一非空索引进行查询的情况。
- *ref*：查询时，根据非唯一非空索引进行查询的情况。
- *range*：在一个索引上进行范围查找。
- *index*：遍历索引树查询，通常发生在查询结果只包含索引字段时。
- *ALL*：全表扫描，没有任何索引可以使用时。这是最差的情况，应该避免。

九．索引使用规则

索引创建完毕后，不是就能使用。在使用时必须遵守对应的规则才能生效.

1) 全列匹配原则

在使用多列索引时，由于 MySQL 对于多列索引中字段顺序是敏感的。所以一般要求多列索引字段出现顺序必须与多列索引声明时的字段声明顺序一致.

但是，如果这些查询条件都是使用”=”或则”in”，则 mySql 依然在本次查询使用多列索引.

```
explain
select ename,job,sal
from emp
where
job='clerk' and sal=800  and ename='smith'
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	emp	ref	myIndex	myIndex	72	const,const,const	1	Using where; Usi

2）字符串匹配原则

在模糊查询时,如果是以’%’为开始导致索引失效。

在模糊查询时,如果是以’%’为结束时不会导致索引失效

```
explain
select ename, job
from emp
where ename like 's%'
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	emp	range	enameIndex	enameIndex	33	<NULL>	2	Using where

```
explain
select ename, job
from emp
where ename like '%s'
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	emp	ALL	<NULL>	<NULL>	<NULL>	<NULL>	14	Using where

3)左前缀匹配原则