MySQL 索引优化

目录

一、Explain 用法

在 select 语句之前增加 explain 关键字
例如：`explain select * from actor;`

各个字段含义：

1. id 列

id 列的编号是 select 的序列号，有几个 select 就有几个 id，并且 id 的顺序是按 select 出现的顺序增长的。

id 列越大执行优先级越高，id 相同则从上往下执行，id 为 NULL 最后执行。

2. select_type 列

select_type 表示对应行是简单还是复杂的查询。

1）simple：

简单查询。查询不包含子查询和 union

explain select * from film where id = 2;

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	film	(Null)	const	PRIMARY	PRIMAR	4	const	1	100	(Null)

2）primary：

复杂查询中最外层的 select

3）subquery：

包含在 select 中的子查询（不在 from 子句中）

4）derived：

包含在 from 子句中的子查询。MySQL 会将结果存放在一个临时表中，也称为派生表（

derived 的英文含

义）

用这个例子来了解 primary、subquery 和 derived 类型

-- 关闭 mysql5.7 新特性对衍生表的合并优化

set session optimizer_switch='derived_merge=off';

explain select (select 1 from actor where id = 1) from (select * from film where

id = 1) der;

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	<derived3>	(Null)	system	(Null)	(Null)	(Null)	(Null)	1	100	(Null)
3	DERIVED	film	(Null)	const	PRIMARY	PRIMAR	4	const	1	100	(Null)
2	SUBQUERY	actor	(Null)	const	PRIMARY	PRIMAR	4	const	1	100	Using index

-- 还原默认配置

set session optimizer_switch='derived_merge=on';

5）union：

在 union 中的第二个和随后的 select

explain select 1 union all select 1;

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	No tables used
2	UNION	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	No tables used

3. table 列

这一列表示 explain 的一行正在访问哪个表。

当 from 子句中有子查询时，table 列是 <derivenN> 格式，表示当前查询依赖 id=N 的查询，于是先执行 id=N 的查

询。

当有 union 时，UNION RESULT 的 table 列的值为<union1,2>，1 和 2 表示参与 union 的 select 行 id。

4. type 列（重要）

这一列表示**关联类型或访问类型**，即 MySQL 决定如何查找表中的行，查找数据行记录的大概范围。

依次从最优到最差分别为：**system > const > eq_ref > ref > range > index > ALL**

一般来说，**得保证查询达到 range 级别，最好达到 ref**

NULL：

mysql 能够在优化阶段分解查询语句，在执行阶段用不着再访问表或索引。例如：在索引列中选取最小值，可

以单独查找索引来完成，不需要在执行时访问表

```
explain select min(id) from film;
```

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	Select tables optimized away

const 和 system：

mysql 能对查询的某部分进行优化并将其转化成一个常量（可以看 show warnings 的结果）。用于 primary key 或 unique key 的所有列与常数比较时，所以表最多有一个匹配行，读取 1 次，速度比较快。

system 是 const 的特例，表里只有一条元组匹配时为 system

set session optimizer_switch='derived_merge=off';

```
explain extended select * from (select * from film where id = 1) tmp;
```

信息 结果1 结果2 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	<derived2>	(Null)	system	(Null)	(Null)	(Null)	(Null)	1	100	(Null)
2	DERIVED	film	(Null)	const	PRIMARY	PRIMAR	4	const	1	100	(Null)

```
show warnings;
```

信息 | 结果1 | 结果2 | 概况 | 状态

Level	Code	Message
Warning	1681	'EXTENDED' is deprecated and will be removed in a future release.
Note	1003	/* select#1 */ select '1' AS `id`,'film1' AS `name` from dual

eq_ref:

primary key 或 unique key 索引的所有部分被连接使用 ，最多只会返回一条符合条件的记录。这可能是在 const 之外最好的联接类型了。

ref:

相比 eq_ref，不使用唯一索引，而是使用普通索引或者唯一性索引的部分前缀，索引要和某个值相比较，可能会找到多个符合条件的行。

1. 简单 select 查询，name 是普通索引（非唯一索引）

```
explain select * from film where name = 'film1';
```

信息 | 结果1 | 概况 | 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	film	(Null)	ref	idx_name	idx_nam	33	const	1	100	Using index

2.关联表查询，idx_film_actor_id 是 film_id 和 actor_id 的联合索引，这里使用到了 film_actor 的左边前缀 film_id 部分。

```
explain select film_id from film left join film_actor on film.id = film_actor.film_id;
```

信息 | 结果1 | 概况 | 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	film	(Null)	index	(Null)	idx_nam	33	(Null)	3	100	Using index
1	SIMPLE	film_actor	(Null)	ref	idx_film_actor_id	idx_film_	4	lxg.film.	1	100	Using index

range:

范围扫描通常出现在 in(), between ,> ,<, >= 等操作中。使用一个索引来检索给定范围的行。

```
explain select * from actor where id > 1;
```

信息 | 结果1 | 概况 | 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	actor	(Null)	range	PRIMARY	PRIMAR	4	(Null)	2	100	Using where

index:

扫描全索引就能拿到结果，一般是扫描某个二级索引，这种扫描不会从索引树根节点开始快速查找，而是直接对二级索引的叶子节点遍历和扫描，速度还是比较慢的，这种查询一般为使用覆盖索引，二级索引一般比较小，所以这 种通常比 ALL 快一些。

```
explain select * from film;
```

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	film	(Null)	index	(Null)	idx_nam	33	(Null)	3	100	Using index

ALL：

即全表扫描，扫描你的聚簇索引的所有叶子节点。通常情况下这需要增加索引来进行优化了。

```
explain select * from actor;
```

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	actor	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	100	(Null)

5. possible_keys 列

这一列显示查询可能使用哪些索引来查找。

explain 时可能出现 possible_keys 有列，而 key 显示 NULL 的情况，这种情况是因为表中数据不多，mysql 认为索引对此查询帮助不大，选择了全表查询。

如果该列是 NULL，则没有相关的索引。在这种情况下，可以通过检查 where 子句看是否可以创造一个适当的索引来提高查询性能，然后用 explain 查看效果。

6. key 列

这一列显示 mysql 实际采用哪个索引来优化对该表的访问。

如果没有使用索引，则该列是 NULL。如果想强制 mysql 使用或忽视 possible_keys 列中的索引，在查询中使用 force index、ignore index。

7. key_len 列

这一列显示了 mysql 在索引里使用的字节数，通过这个值可以算出具体使用了索引中的哪些列。

举例来说，film_actor 的联合索引 idx_film_actor_id 由 film_id 和 actor_id 两个 int 列组成，并且每个 int 是 4 字节。通过结果中的 key_len=4 可推断出查询使用了第一个列：film_id 列来执行索引查找。

```
explain select * from film_actor where film_id = 2;
```

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	film_actor	(Null)	ref	idx_film_actor_id	idx_film_actor_id	4	const	1	100	(Null)

key_len 计算规则如下：

字符串

char(n)和 varchar(n)，5.0.3 以后版本中，**n 均代表字符数，而不是字节数，**如果是 utf-8，一个数字或字母占 1 个字节，一个汉字占 3 个字节

char(n)：如果存汉字长度就是 3n 字节

varchar(n)：如果存汉字则长度是 3n + 2 字节，加的 2 字节用来存储字符串长度，因为 varchar 是变长字符串

数值类型

tinyint：1 字节

smallint：2 字节

int：4 字节

bigint：8 字节

时间类型

date：3 字节

timestamp：4 字节

datetime：8 字节

如果字段允许为 NULL，需要 1 字节记录是否为 NULL

索引最大长度是 768 字节，当字符串过长时，mysql 会做一个类似左前缀索引的处理，将前半部分的字符提取出来做索
引。

计算示例：

```
name` varchar(24)
`age` int(11)
`position` varchar(20)
```

Utf-8

```
EXPLAIN SELECT * FROM employees WHERE name= 'LiLei';
```

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ref	idx_name_age_pos	idx_nam	74	const	1	100	(Null)

```
3 * 24 + 2 = 74
```

```
EXPLAIN SELECT * FROM employees WHERE name= 'LiLei' AND age = 22;
```

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ref	idx_name_age_pos	idx_nam	78	const,cc	1	100	(Null)

```
（3 * 24 + 2） + 4 = 78
```

```
EXPLAIN SELECT * FROM employees WHERE name= 'LiLei' AND age = 22 AND position ='manage
r';
```

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ref	idx_name_age_pos	idx_nam	140	const,const,const	1	100	(Null)

```
（3 * 24 + 2） + 4 + （3 * 20 + 2） = 140
```

非 Utf-8，并且 name 允许为 null

```
EXPLAIN SELECT * FROM employees_copy WHERE name= 'LiLei';
```

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ref	idx_name_age_positio	idx_name_age_position	27	const	1	100	(Null)

```
1 * 24 + 1 + 2 = 27
```

```
EXPLAIN SELECT * FROM employees_copy WHERE name= 'LiLei' AND age = 22;
```

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ref	idx_name_age_positio	idx_name_age_position	31	const,cc	1	100	(Null)

```
（1 * 24 + 1 + 2） + 4 = 31
```

```
EXPLAIN SELECT * FROM employees_copy WHERE name= 'LiLei' AND age = 22 AND position
='manage
r';
（1 * 24 + 1 + 2） + 4 + （1 * 20 + 2） = 53
```

8. ref 列

这一列显示了在 key 列记录的索引中，表查找值所用到的列或常量，常见的有：const（常量），字段名（例：film.id）

9. rows 列

这一列是 mysql 估计要读取并检测的行数，注意这个不是结果集里的行数。

10. Extra 列

这一列展示的是额外信息。常见的重要值如下：

1）**Using index**：使用覆盖索引

覆盖索引定义：mysql 执行计划 explain 结果里的 key 有使用索引，如果 select 后面查询的字段都可以从

这个索引的树中获取，这种情况一般可以说是用到了覆盖索引，extra 里一般都有 using index；覆盖索引一般针对的是辅助索引，整个查询结果只通过辅助索引就能拿到结果，不需要通过辅助索引树找到主键，再通过主键去主键索引树里获取其它字段值

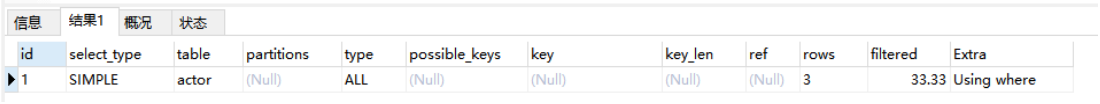
```
explain select film_id from film_actor where film_id = 1;
```

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	film_actor	(Null)	ref	idx_film_actor_id	idx_film_actor_id	4	const	2	100	Using index

2）**Using where**：使用 where 语句来处理结果，并且查询的列未被索引覆盖

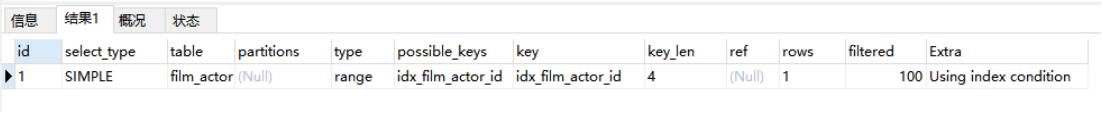
```
explain select * from actor where name = 'a';
```



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	actor	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	33.33	Using where

3）**Using index condition**：查询的列不完全被索引覆盖，where 条件中是一个前导列的范围；

```
explain select * from film_actor where film_id > 1;
```

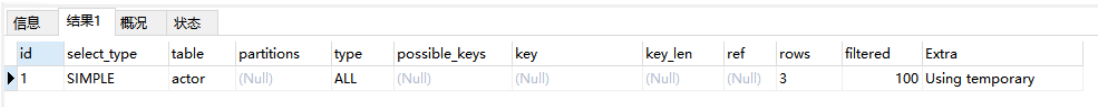


id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	film_actor	(Null)	range	idx_film_actor_id	idx_film_actor_id	4	(Null)	1	100	Using index condition

4）**Using temporary**：mysql 需要创建一张临时表来处理查询。出现这种情况一般是要进行优化的，首先是想到用索引来优化。

1. actor.name 没有索引，此时创建了张临时表来 distinct

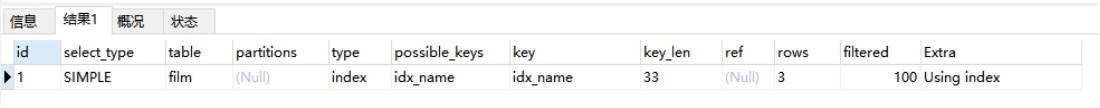
```
explain select distinct name from actor;
```



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	actor	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	100	Using temporary

2. film.name 建立了 idx_name 索引，此时查询时 extra 是 using index,没有用临时表

```
explain select distinct name from film;
```



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	film	(Null)	index	idx_name	idx_name	33	(Null)	3	100	Using index

5）**Using filesort**：将用外部排序而不是索引排序，数据较小时从内存排序，否则需要在磁盘完成排序。

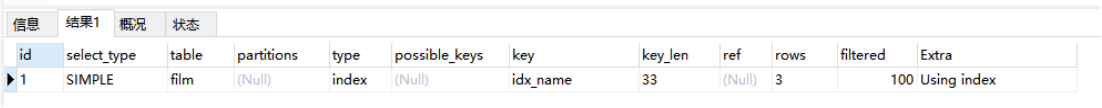
这种情况下一般也是要考虑使用索引来优化的。

1. actor.name 未创建索引，会浏览 actor 整个表，保存排序关键字 name 和对应的 id，然后排序 name 并检索行记录

```
explain select * from actor order by name;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	actor	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	100	Using filesort

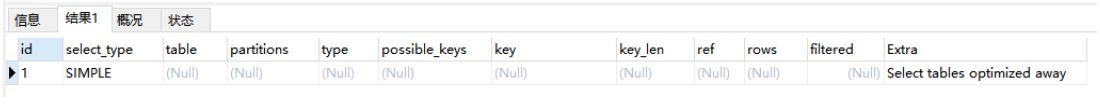
2. film.name 建立了 idx_name 索引,此时查询时 extra 是 using index



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	film	(Null)	index	(Null)	idx_name	33	(Null)	3	100	Using index

6）**Select tables optimized away**：使用某些聚合函数（比如 max、min）来访问存在索引的某个字段是

```
explain select min(id) from film;
```



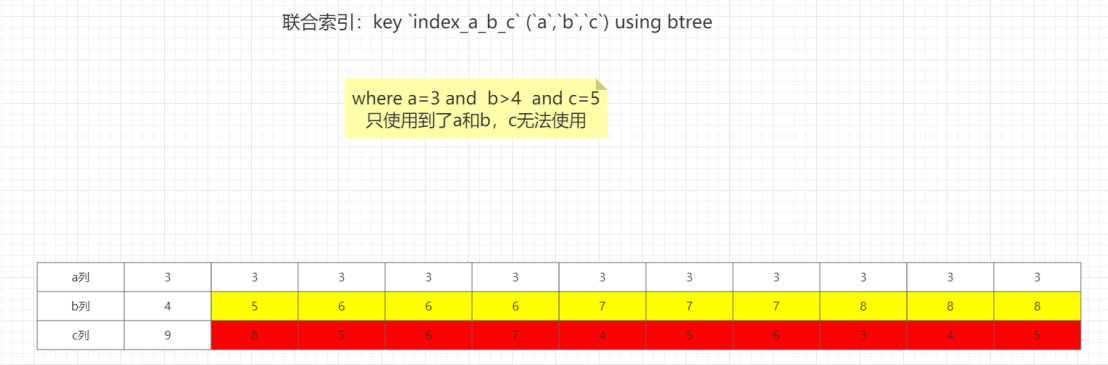
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	Select tables optimized away

二、索引最佳实践

1.全值匹配

```
SELECT * FROM employees WHERE name= 'LiLei';
```

2.最左前缀法则



如果索引了多列，要遵守最左前缀法则。指的是查询从索引的最左前列开始并且**不跳过索引中的列**。

```
EXPLAIN SELECT * FROM employees WHERE name = 'Bill' and age = 31;
```

信息 | 结果1 | 结果2 | 结果3 | 概况 | 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ref	idx_name_age_pos	idx_name_age_position	78	const,const	1	100	(Null)

```
EXPLAIN SELECT * FROM employees WHERE age = 30 AND position = 'dev';
```

信息 | 结果1 | 结果2 | 结果3 | 概况 | 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	33.33	Using where

```
EXPLAIN SELECT * FROM employees WHERE position = 'manager';
```

信息 | 结果1 | 结果2 | 结果3 | 概况 | 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	33.33	Using where

3.不在索引列上做任何操作

计算、函数、类型转换，会导致索引失效而转向全表扫描

```
EXPLAIN SELECT * FROM employees WHERE left(name,3) = 'LiLei';
```

信息 结果1 结果2 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	100	Using where

给 hire_time 增加一个普通索引:

```
ALTER TABLE `employees` ADD INDEX `idx_hire_time` (`hire_time`) USING BTREE ;
EXPLAIN select * from employees where date(hire_time) ='2018-09-30';
```

转化为日期范围查询，有可能会走索引:

EXPLAIN SELECT * FROM employees WHERE hire_time >= '2018-09-30 00:00:00' AND hire_time <= '2018-09-30 23:59:59';

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	range	idx_hire_time	idx_hire_time	4	(Null)	1	100	Using index condition

还原最初索引状态

```
ALTER TABLE `employees` DROP INDEX `idx_hire_time`;
```

4.存储引擎不能使用索引中范围条件右边的列

```
EXPLAIN SELECT * FROM employees WHERE name= 'LiLei' AND age = 22 AND position ='manager';
EXPLAIN SELECT * FROM employees WHERE name= 'LiLei' AND age > 22 AND position ='manager';
```

信息 结果1 结果2 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	range	idx_name_age_pos	idx_nam	78	(Null)	1	33.33	Using index condition

5.尽量使用覆盖索引

只访问索引的查询（索引列包含查询列），减少 `select *` 语句

```
EXPLAIN SELECT name,age FROM employees WHERE name= 'LiLei' AND age = 23 AND position ='manager';
```

信息 结果1 结果2 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ref	idx_name_age_position	idx_name_age_position	140	const,const,const	1	100	Using index

```
EXPLAIN SELECT * FROM employees WHERE name= 'LiLei' AND age = 23 AND position ='manager';
```

信息 结果1 结果2 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ref	idx_name_age_position	idx_name_age_position	140	const,const,const	1	100	(Null)

6. 尽量不使用不等于（！ =或者<>），not in ，not exists

mysql 在使用不等于（！ =或者<>），not in ，not exists 的时候无法使用索引会导致全表扫描 < 小于、 > 大于、 <=、>= 这些，mysql 内部优化器会根据检索比例、表大小等多个因素整体评估是否使用索引。

```
EXPLAIN SELECT * FROM employees WHERE name != 'LiLei';
```

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ALL	idx_name_age_position	(Null)	(Null)	(Null)	3	66.67	Using where

7.is null,is not null 一般情况下也无法使用索引

```
EXPLAIN SELECT * FROM employees WHERE name is null;
```

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	Impossible WHERE

8.like 以通配符开头（'%abc...'）mysql 索引失效会变成全表扫描操作

```
EXPLAIN SELECT * FROM employees WHERE name like '%Lei';
```

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	3	33.33	Using where

```
EXPLAIN SELECT * FROM employees WHERE name like 'Lei%';
```

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	range	idx_name_age_position	idx_name_age_position	74	(Null)	1	100	Using index condition

解决 like'%字符串%'索引不被使用的方法，使用覆盖索引，查询字段必须是建立覆盖索引字段

```
EXPLAIN SELECT name,age,position FROM employees WHERE name like '%Lei%';
```

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	index	(Null)	idx_name_age_position	140	(Null)	3	33.33	Using where; Using index

9.字符串不加单引号索引失效

```
EXPLAIN SELECT * FROM employees WHERE name = 1000;
```

信息 结果1 结果2 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ALL	idx_name_age_position	(Null)	(Null)	(Null)	3	33.33	Using where

10.少用 or 或 in

在使用 **or 或 in** 时，mysql 不一定使用索引，mysql 内部优化器会根据检索比例、表大小等多个因素整体评估是否使用索引。

```
EXPLAIN SELECT * FROM employees WHERE name = 'LiLei' or name = 'HanMeimei';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ALL	idx_name_age_position	(Null)	(Null)	(Null)	3	66.67	Using where

11.范围查询优化

给年龄添加单值索引

```
ALTER TABLE `employees` ADD INDEX `idx_age` (`age`) USING BTREE;
explain select * from employees where age >=1 and age <=2000;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ALL	idx_age	(Null)	(Null)	(Null)	3	100	Using where

没走索引原因：mysql 内部优化器会根据检索比例、表大小等多个因素整体评估是否使用索引。比如这个例子，可能是由于单次数据量查询过大导致优化器最终选择不走索引

优化方法：可以将大的范围拆分成多个小范围

```
explain select * from employees where age >=1 and age <=1000;
explain select * from employees where age >=1001 and age <=2000;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	range	idx_age	idx_age	4	(Null)	1	100	Using index condition

还原最初索引状态

```
ALTER TABLE `employees` DROP INDEX `idx_age`;
```

12.联合索引第一个字段用范围不会走索引

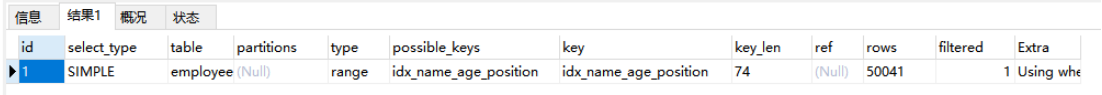
```
EXPLAIN SELECT * FROM employees WHERE name > 'LiLei';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ALL	idx_name_age_position	(Null)	(Null)	(Null)	100082	0.5	Using where

联合索引第一个字段就用范围查找不会走索引，mysql 内部可能觉得第一个字段就用范围，结果集应该很大，回表效率不高，还不如就全表扫描。

优化策略：**覆盖索引优化**

```
EXPLAIN SELECT name,age,position FROM employees WHERE name > 'LiLei' AND age = 22 AND position
='manager';
```



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	range	idx_name_age_position	idx_name_age_position	74	(Null)	50041	1	Using whe

13.强制走索引

```
EXPLAIN SELECT * FROM employees force index(idx_name_age_position) WHERE name > 'LiLei';
```

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	range	idx_name_age_position	idx_name_age_position	74	(Null)	50041	1	Using index condition

```
-- 关闭查询缓存
set global query_cache_size=0;
set global query_cache_type=0;
SELECT * FROM employees WHERE name > 'LiLei';
```

信息 结果1 概况 状态

id	name	age	position	hire_time
6	Lucy	23	dev	2022-01-29 10:39:16
7	zhuge1	1	dev	2022-01-29 13:50:27
8	zhuge2	2	dev	2022-01-29 13:50:27
9	zhuge3	3	dev	2022-01-29 13:50:27
10	zhuge4	4	dev	2022-01-29 13:50:27
11	zhuge5	5	dev	2022-01-29 13:50:27
12	zhuge6	6	dev	2022-01-29 13:50:27
13	zhuge7	7	dev	2022-01-29 13:50:27
14	zhuge8	8	dev	2022-01-29 13:50:27
15	zhuge9	9	dev	2022-01-29 13:50:27
16	zhuge10	10	dev	2022-01-29 13:50:27
17	zhuge11	11	dev	2022-01-29 13:50:27
18	zhuge12	12	dev	2022-01-29 13:50:27
19	zhuge13	13	dev	2022-01-29 13:50:27
20	zhuge14	14	dev	2022-01-29 13:50:27

SELECT * FROM employees WHERE name > 'LiLei'; 查询时间: 0.148s

```
set global query_cache_size=0;
set global query_cache_type=0;
SELECT * FROM employees force index(idx_name_age_position) WHERE name > 'LiLei';
```

信息 结果1 概况 状态

id	name	age	position	hire_time
6	Lucy	23	dev	2022-01-29 10:39:16
7	zhuge1	1	dev	2022-01-29 13:50:27
16	zhuge10	10	dev	2022-01-29 13:50:27
106	zhuge100	100	dev	2022-01-29 13:50:27
1006	zhuge1000	1000	dev	2022-01-29 13:50:29
10006	zhuge10000	10000	dev	2022-01-29 13:50:54
100006	zhuge100000	100000	dev	2022-01-29 13:54:51
10007	zhuge10001	10001	dev	2022-01-29 13:50:54
10008	zhuge10002	10002	dev	2022-01-29 13:50:54
10009	zhuge10003	10003	dev	2022-01-29 13:50:54
10010	zhuge10004	10004	dev	2022-01-29 13:50:54
10011	zhuge10005	10005	dev	2022-01-29 13:50:54
10012	zhuge10006	10006	dev	2022-01-29 13:50:54
10013	zhuge10007	10007	dev	2022-01-29 13:50:54
10014	zhuge10008	10008	dev	2022-01-29 13:50:54

SELECT * FROM employees force index(idx_name_age_position) WHERE name > 'LiLei'; 查询时间: 0.415s

结论：

虽然使用了强制走索引让联合索引第一个字段范围查找也走索引，扫描的行 rows 看上去也少了点，但是最终查找效率不一定比全表扫描高，因为**回表效率不高。**

14.索引下推

```
EXPLAIN SELECT * FROM employees WHERE name like 'LiLei%' AND age = 22 AND position ='manager';
```

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	range	idx_name_age_position	idx_name_age_position	140	(Null)	1	5	Using index condition

MySQL 5.6 引入了索引下推优化，**可以在索引遍历过程中，对索引中包含的所有字段先做判断，过滤掉不符合条件的记录之后再回表，可以有效的减少回表次数。**使用了索引下推优化后，上面那个查询在联合索引里匹配到名字是 **'LiLei' 开头**的索引之后，同时还会继续在索引里过滤 **age** 和 **position** 这两个字段，拿着过滤完剩下的索引对应的主键 id 再回表查整行数据。

为什么范围查找 Mysql 没有用索引下推优化？

估计应该是 Mysql 认为范围查找过滤的结果集过大，like KK% 在绝大多数情况来看，过滤后的结果集比较小，所以这里 Mysql 选择给 like KK% 用了索引下推优化，当然这也不是绝对的，有时 like KK% 也不一定就会走索引下推。

15.范围查找特殊情况

EXPLAIN select * from employees where name > 'aaa';

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ALL	idx_name_age_position	(Null)	(Null)	(Null)	100082	50	Using where

EXPLAIN select * from employees where name > 'zzz';

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	range	idx_name_age_position	idx_name_age_position	74	(Null)	1	100	Using index condition

Mysql 认为 name > 'aaa' 结果集很大，不如直接全表扫描，而 name > 'zzz' 结果集就很小了，所以先走索引树，拿到主键再回表查询。

16.Using filesort 文件排序原理详解

filesort 文件排序方式

单路排序：是一次性取出满足条件行的所有字段，然后在 sort buffer 中进行排序

双路排序（又叫回表排序模式）：是首先根据相应的条件取出相应的排序字段和可以直接定位行数据的行 ID，然后在 sort buffer 中进行排序，排序完后需要回表再次取回其它需要的字段；

MySQL 通过比较系统变量 max_length_for_sort_data(默认 1024 字节）的大小和需要查询的字段总大小来判断使用哪种排序模式。

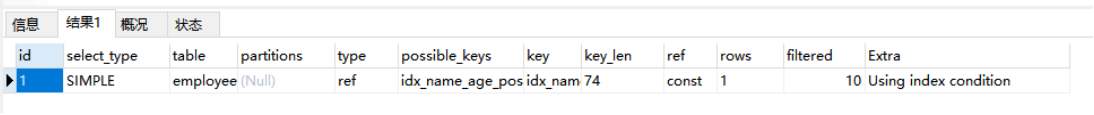
如果字段的总长度小于 max_length_for_sort_data ，那么使用 单路排序模式；

如果字段的总长度大于 max_length_for_sort_data ，那么使用 双路排序模式。

17.Order by 与 Group by 优化

Case1：

EXPLAIN select * from employees where `name` = 'LiLei' and position = 'dev' order by age;



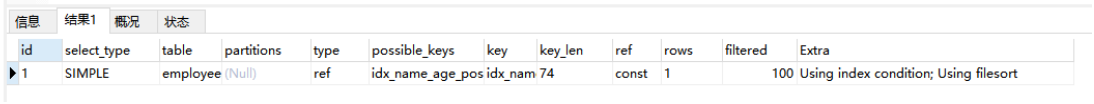
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ref	idx_name_age_pos	idx_nam	74	const	1	10	Using index condition

分析：

利用最左前缀法则：中间字段不能断，因此查询用到了 name 索引，从 key_len=74 也能看出，age 索引列用在排序过程中，因为 Extra 字段里没有 using filesort

Case 2：

EXPLAIN select * from employees where `name` = 'LiLei' order by position;



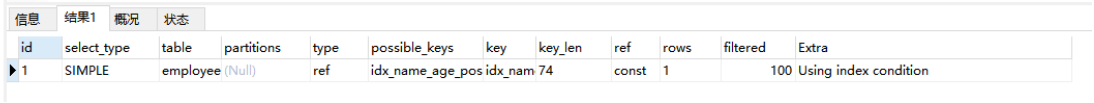
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ref	idx_name_age_pos	idx_nam	74	const	1	100	Using index condition; Using filesort

分析：

从 explain 的执行结果来看：key_len=74，查询使用了 name 索引，由于用了 position 进行排序，跳过了 age，出现了 Using filesort。

Case 3：

EXPLAIN select * from employees where `name` = 'LiLei' order by age,position;



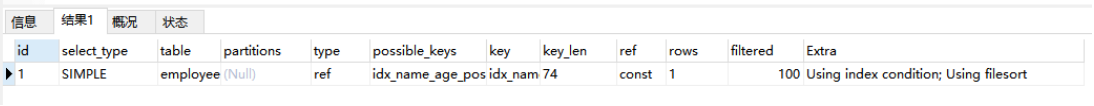
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ref	idx_name_age_pos	idx_nam	74	const	1	100	Using index condition

分析：

查找只用到索引 name，age 和 position 用于排序，无 Using filesort。

Case 4：

EXPLAIN select * from employees where `name` = 'LiLei' order by position,age;



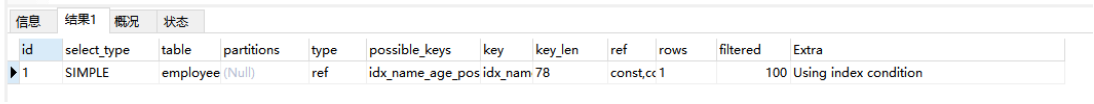
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ref	idx_name_age_pos	idx_nam	74	const	1	100	Using index condition; Using filesort

分析：

和 Case 3 中 explain 的执行结果一样，但是出现了 Using filesort，因为索引的创建顺序为 name,age,position，但是排序的时候 age 和 position 颠倒位置了。

Case 5：

EXPLAIN select * from employees where `name` = 'LiLei' and age = 18 order by position,age;



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ref	idx_name_age_pos	idx_nam	78	const,cc	1	100	Using index condition

分析：

与 Case 4 对比，在 Extra 中并未出现 Using filesort，因为 age 为常量，在排序中被优化，所以索引未颠倒，不会出现 Using filesort。

Case 6：

EXPLAIN select * from employees where `name` = 'LiLei' order by age asc,position desc;

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ref	idx_name_age_pos	idx_nam	74	const	1	100	Using index condition; Using filesort

分析：

虽然排序的字段列与索引顺序一样，且 order by 默认升序，这里 position desc 变成了降序，导致与索引的排序方式不同，从而产生 Using filesort。

age 和 position 改成降序

EXPLAIN select * from employees where `name` = 'LiLei' order by age desc,position desc;

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ref	idx_name_age_pos	idx_nam	74	const	1	100	Using where

分析：

这里 age 和 position 变成了降序，mysql 会采用从后往前取的方式遍历索引树，拿到主键 id 后，依次从主键索引取出的结果也是排好序的，所以不会再有 Using filesort。

Case 7:

EXPLAIN select * from employees where `name` in ('LiLei','zhuge') order by age,position;

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	range	idx_name_age_pos	idx_nam	74	(Null)	2	100	Using index condition; Using filesort

分析：

对于排序来说，多个相等条件也是范围查询

Case 8:

EXPLAIN select * from employees where `name` > 'a' order by name;

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ALL	idx_name_age_pos	(Null)	(Null)	(Null)	100082	50	Using where; Using filesort

可以用覆盖索引优化

EXPLAIN select name,age,position from employees where `name` > 'a' order by name;

信息 结果1 结果2 概况 状态

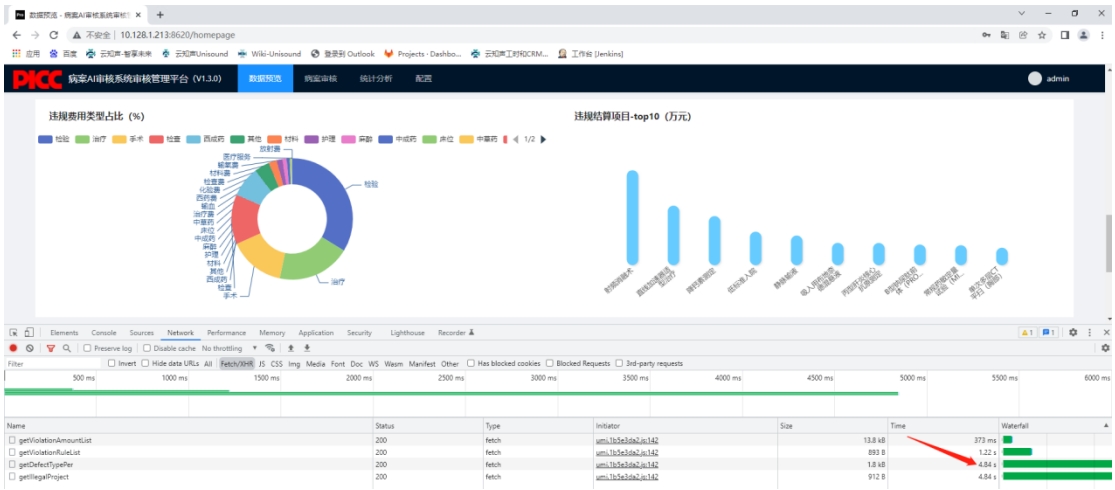
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	range	idx_name_age_position	idx_nam	74	(Null)	50041	100	Using where; Using index

索引树包含了要查询的所有字段，不用回表了，直接利用索引树排序就可以了。

Case 9 :

病案 AI 审核系统审核管理平台，数据预览-违规费用类型占比接口：
请求 url：
http://10.128.1.213:8620/app_wheeljack_manager/rest_api/v1/medicare/getDefectTypePer
请求参数：
{

```
    "hospitalId": "",
    "settlementStartDate": "2020-11-01",
    "settlementEndDate": "2021-11-30"
}
```



可以看到该接口用时 4.84 秒，抓取后台 sql：

```
SELECT sum(dt.reducescroe) AS illegalAmount,dt.illegal_type AS illegalType
FROM 0_detail_tb dt
LEFT JOIN 0_userbase_expand ube ON (
            dt.admissionid = ube.admission_id
            AND ube.hospital_id = dt.hospital_id
)
WHERE dt.illegal_valid = 1
AND date_format(ube.settlement_date,'%Y-%m-%d') >= '2020-11-01 00:00:00.0'
AND date_format(ube.settlement_date,'%Y-%m-%d') <= '2021-11-30 00:00:00.0'
GROUP BY dt.illegal_type ORDER BY illegalAmount DESC;
```

Sql 执行时间：2.974 秒

使用 EXPLAIN 分析：

```
EXPLAIN
SELECT sum(dt.reducescroe) AS illegalAmount,dt.illegal_type AS illegalType
FROM 0_detail_tb dt
LEFT JOIN 0_userbase_expand ube ON (
  dt.admissionid = ube.admission_id
  AND ube.hospital_id = dt.hospital_id
)
WHERE dt.illegal_valid = 1
AND date_format(ube.settlement_date,'%Y-%m-%d') >= '2020-11-01 00:00:00.0'
AND date_format(ube.settlement_date,'%Y-%m-%d') <= '2021-11-30 00:00:00.0'
GROUP BY dt.illegal_type ORDER BY illegalAmount DESC;
```

信息 | 结果1 | 概况 | 状态

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ube	ALL	idx_admission_id	(Null)	(Null)	(Null)	14858	Using where; Using temporary; Using filesort
1	SIMPLE	dt	ref	admissionid,idx_illegal_valid	admissionid	99	app_wheeljack_medic.15		Using where

key_len: 32 * 3 + 2 + 1 = 99

0_userbase_expand 表数据量：14804

0_detail_tb 表数据量：438298

重点优化目标：0_detail_tb 表

（1）可以看到查询使用到了 illegal_type 字段进行分组，所以，先给 illegal_type 字段建个索引

CREATE INDEX idx_test ON 0_detail_tb (illegal_type);

查询创建工具 查询编辑器

```
EXPLAIN
SELECT sum(dt.reducescroe) AS illegalAmount,dt.illegal_type AS illegalType
FROM 0_detail_tb dt
LEFT JOIN 0_userbase_expand ube ON (
  dt.admissionid = ube.admission_id
  AND ube.hospital_id = dt.hospital_id
)
WHERE dt.illegal_valid = 1
AND date_format(ube.settlement_date,'%Y-%m-%d') >= '2020-11-01 00:00:00.0'
AND date_format(ube.settlement_date,'%Y-%m-%d') <= '2021-11-30 00:00:00.0'
GROUP BY dt.illegal_type ORDER BY illegalAmount DESC;
```

信息 结果1 概况 状态

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ube	ALL	idx_admission_id	(Null)	(Null)	(Null)	14858	Using where; Using temporary; Using filesort
1	SIMPLE	dt	ref	admissionid,idx_illegal_valid,idx_test	admissionid	99	app_wheeljack_n	15	Using where

执行计划没有变化

（2）考虑到最左匹配原则，where 条件中还有个 illegal_valid 字段，继续调整索引

DROP INDEX idx_test ON 0_detail_tb;

CREATE INDEX idx_test ON 0_detail_tb (illegal_valid,illegal_type);

查询创建工具 查询编辑器

```
EXPLAIN
SELECT sum(dt.reducescroe) AS illegalAmount,dt.illegal_type AS illegalType
FROM 0_detail_tb dt
LEFT JOIN 0_userbase_expand ube ON (
  dt.admissionid = ube.admission_id
  AND ube.hospital_id = dt.hospital_id
)
WHERE dt.illegal_valid = 1
AND date_format(ube.settlement_date,'%Y-%m-%d') >= '2020-11-01 00:00:00.0'
AND date_format(ube.settlement_date,'%Y-%m-%d') <= '2021-11-30 00:00:00.0'
GROUP BY dt.illegal_type ORDER BY illegalAmount DESC;
```

信息 结果1 概况 状态

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ube	ALL	idx_admission_id	(Null)	(Null)	(Null)	14858	Using where; Using temporary; Using filesort
1	SIMPLE	dt	ref	admissionid,idx_illegal_valid,idx_test	admissionid	99	app_wheeljack_n	15	Using where

执行计划没有变化

（3）因为这个表有 join 查询，所以 admissionid 和 hospital_id 也相当于在过滤条件中了，继续调整索引

DROP INDEX idx_test ON 0_detail_tb;

CREATE INDEX idx_test ON 0_detail_tb (admissionid,hospital_id,illegal_valid,illegal_type);

查询创建工具 查询编辑器

```
EXPLAIN
SELECT sum(dt.reducescroe) AS illegalAmount,dt.illegal_type AS illegalType
FROM 0_detail_tb dt
LEFT JOIN 0_userbase_expand ube ON (
  dt.admissionid = ube.admission_id
  AND ube.hospital_id = dt.hospital_id
)
WHERE dt.illegal_valid = 1
AND date_format(ube.settlement_date,'%Y-%m-%d') >= '2020-11-01 00:00:00.0'
AND date_format(ube.settlement_date,'%Y-%m-%d') <= '2021-11-30 00:00:00.0'
GROUP BY dt.illegal_type ORDER BY illegalAmount DESC;
```

信息 结果1 概况 状态

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ube	ALL	idx_admission_id	(Null)	(Null)	(Null)	14858	Using where; Using temporary; Using filesort
1	SIMPLE	dt	ref	admissionid,idx_illegal_valid,idx_test	idx_test	108	app_wheeljack_n	9	(Null)

可以看到 **key 列变成了 idx_test**，同时 **key_len 变成了 108** (illegal_typ 列用于排序了，不参与 key_len 计算)

key_len: (32 * 3 + 2 + 1) + 4 + (4+1) = 108

但是，sql 执行时间任然需要 2 秒多

（4）虽然联合索引包含了条件中的所有字段，但是在 select 字段中，还有个 reducescroe，想到了使用**覆盖索引**，让所查询数据在索引树就可以全部拿到，**减少回表**性能消耗

DROP INDEX idx_test ON 0_detail_tb;

CREATE INDEX idx_test ON 0_detail_tb (admissionid,hospital_id,illegal_valid,illegal_type,reducescroe);

查询创建工具 查询编辑器

```
EXPLAIN
SELECT sum(dt.reducescroe) AS illegalAmount,dt.illegal_type AS illegalType
FROM 0_detail_tb dt
LEFT JOIN 0_userbase_expand ube ON (
  dt.admissionid = ube.admission_id
  AND ube.hospital_id = dt.hospital_id
)
WHERE dt.illegal_valid = 1
AND date_format(ube.settlement_date,'%Y-%m-%d') >= '2020-11-01 00:00:00.0'
AND date_format(ube.settlement_date,'%Y-%m-%d') <= '2021-11-30 00:00:00.0'
GROUP BY dt.illegal_type ORDER BY illegalAmount DESC;
```

信息 结果1 概况 状态

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ube	ALL	idx_admission_id	(Null)	(Null)	(Null)	14858	Using where; Using temporary; Using filesort
1	SIMPLE	dt	ref	admissionid,idx_illegal_valid,idx_test	idx_test	108	app_wheeljack_	10	Using index

可以看到，Extra 列由 null 变成了 Using index，再次执行 sql 查询

```
SELECT sum(dt.reducescroe) AS illegalAmount,dt.illegal_type AS illegalType
FROM 0_detail_tb dt
LEFT JOIN 0_userbase_expand ube ON (
  dt.admissionid = ube.admission_id
  AND ube.hospital_id = dt.hospital_id
)
WHERE dt.illegal_valid = 1
AND date_format(ube.settlement_date,'%Y-%m-%d') >= '2020-11-01 00:00:00.0'
AND date_format(ube.settlement_date,'%Y-%m-%d') <= '2021-11-30 00:00:00.0'
GROUP BY dt.illegal_type ORDER BY illegalAmount DESC;
```

信息 结果1 概况 状态

illegalAmount	illegalType
4910022.30	检验
2844816.03	治疗
2177067.70	手术
1934472.60	检查
1185354.77	西成药
587016.15	其他
319010.09	材料
216686.00	护理
167906.00	麻醉
114156.09	中成药

SELECT sum(dt.reducescroe) AS illegalAmount,dt.illegal_type AS illegalType FROM 0_detail_tb dt LEFT JOIN 0_userbase_expand ube ON (dt.admissionid = ube.admission_id 只读 查询时间: 0.751s

查询时间变成了 0.751 秒，时间减少了一大半。

深入分析：

情况 1：

DROP INDEX idx_test ON 0_detail_tb;

CREATE INDEX idx_test ON 0_detail_tb (admissionid,hospital_id,illegal_valid,reducescroe,**illegal_type**);

信息 结果1 概况 状态

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ube	ALL	idx_admission_id	(Null)	(Null)	(Null)	14858	Using where; Using temporary; Using filesort
1	SIMPLE	dt	ref	admissionid,idx_illegal_valid,idx_test	idx_test	108	app_wheeljack_me	10	Using index

Sql 执行时间：0.705 秒

情况 2：

DROP INDEX idx_test ON 0_detail_tb;

CREATE INDEX idx_test ON 0_detail_tb (admissionid,hospital_id,**illegal_type**,illegal_valid,reducescroe);

信息 结果1 概况 状态

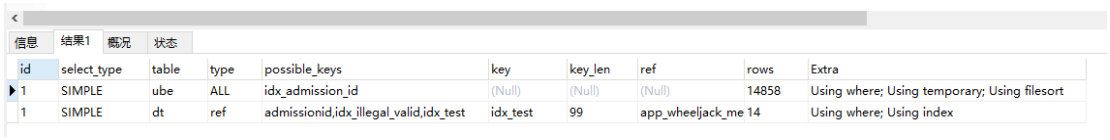
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ube	ALL	idx_admission_id	(Null)	(Null)	(Null)	14858	Using where; Using temporary; Using filesort
1	SIMPLE	dt	ref	admissionid,idx_illegal_valid,idx_test	idx_test	103	app_wheeljack_me	12	Using where; Using index

Sql 执行时间：0.712 秒

情况 3：

DROP INDEX idx_test ON 0_detail_tb;

CREATE INDEX idx_test ON 0_detail_tb (admissionid,**illegal_type**,hospital_id,illegal_valid,reducescroe);



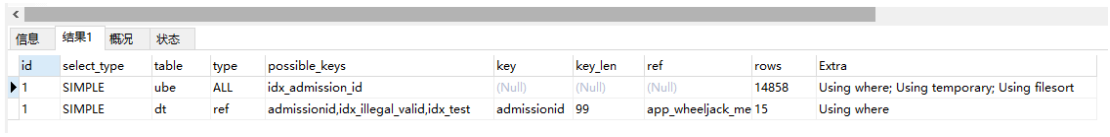
id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ube	ALL	idx_admission_id	(Null)	(Null)	(Null)	14858	Using where; Using temporary; Using filesort
1	SIMPLE	dt	ref	admissionid,idx_illegal_valid,idx_test	idx_test	99	app_wheeljack_me	14	Using where; Using index

Sql 执行时间：0.766 秒

情况 4：

DROP INDEX idx_test ON 0_detail_tb;

CREATE INDEX idx_test ON 0_detail_tb (**illegal_type**,admissionid,hospital_id,illegal_valid,reducescroe);



id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	ube	ALL	idx_admission_id	(Null)	(Null)	(Null)	14858	Using where; Using temporary; Using filesort
1	SIMPLE	dt	ref	admissionid,idx_illegal_valid,idx_test	admissionid	99	app_wheeljack_me	15	Using where

Sql 执行时间：3.467 秒

优化总结：

1、MySQL 支持两种方式的排序 filesort 和 index，Using index 是指 MySQL 扫描索引本身完成排序。index 效率高，filesort 效率低。

2、order by 满足两种情况会使用 Using index。

1) order by 语句使用索引最左前列。

2) 使用 where 子句与 order by 子句条件列组合满足索引最左前列。

3、尽量在索引列上完成排序，遵循索引建立（索引创建的顺序）时的最左前缀法则。

4、如果 order by 的条件不在索引列上，就会产生 Using filesort。

5、能用覆盖索引尽量用覆盖索引

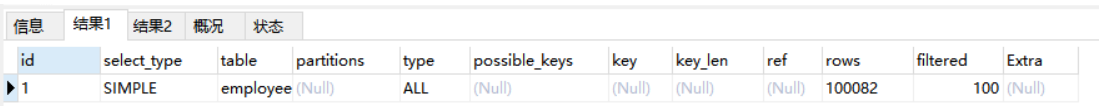
6、group by 与 order by 很类似，其实质是先排序后分组，遵照索引创建顺序的最左前缀法则。对于 group by 的优化如果不需要排序的可以加上 **order by null 禁止排序。**注意，where 高于 having，能写在 where 中的限定条件就不要去 having 限定了。

18.分页查询优化

根据自增且连续的主键排序的分页查询

很多时候我们业务系统实现分页功能可能会用如下 sql 实现

select * from employees limit 90000,10;

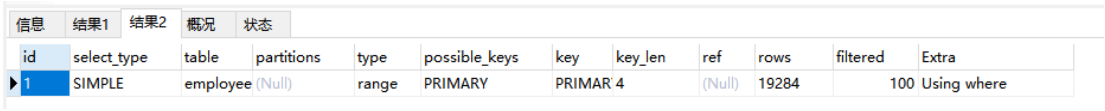


id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	100082	100	(Null)

表示从表 employees 中取出从 90001 行开始的 10 行记录。看似只查询了 10 条记录，实际这条 SQL 是先读取 90010 条记录，然后抛弃前 90000 条记录，然后读到后面 10 条想要的数据。因此要查询一张大表比较靠后的数据，执行效率是非常低的。

可以改成如下写法：

```
EXPLAIN select * from employees where id > 90000 limit 5;
```



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	range	PRIMARY	PRIMAR	4	(Null)	19284	100	Using where

但是，这条改写的 SQL 在很多场景并不实用，这种改写得满足以下两个条件：

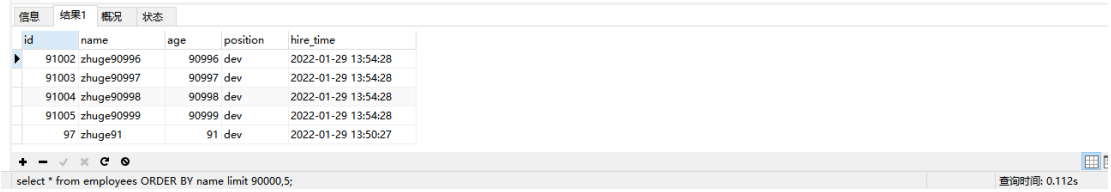
（1）主键自增且连续

（2）结果是按照主键排序的

根据非主键字段排序的分页查询

再看一个根据非主键字段排序的分页查询，SQL 如下：

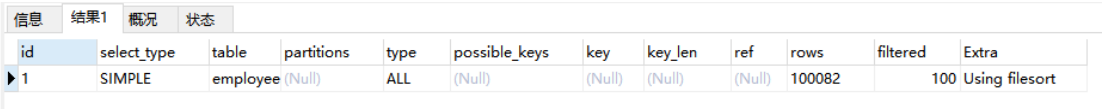
select * from employees ORDER BY name limit 90000,5;



id	name	age	position	hire_time
91002	zhuge90996	90996	dev	2022-01-29 13:54:28
91003	zhuge90997	90997	dev	2022-01-29 13:54:28
91004	zhuge90998	90998	dev	2022-01-29 13:54:28
91005	zhuge90999	90999	dev	2022-01-29 13:54:28
97	zhuge91	91	dev	2022-01-29 13:50:27

select * from employees ORDER BY name limit 90000,5; 查询时间: 0.112s

EXPLAIN select * from employees ORDER BY name limit 90000,5;



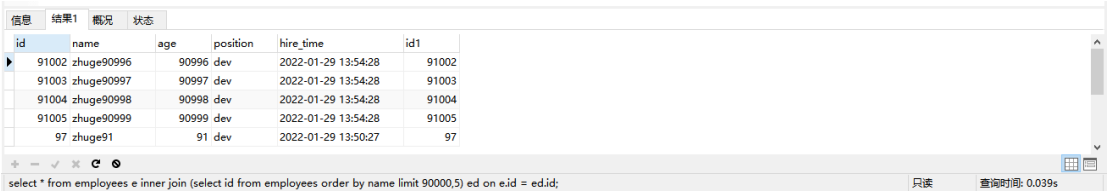
id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	100082	100	Using filesort

发现并没有使用 name 字段的索引（key 字段对应的值为 null），具体原因是：mysql 分析出扫描整个索引的成本比扫描全表的成本更高，所以优化器放弃使用索引。

知道不走索引的原因，那么怎么优化呢？

其实关键是让排序时返回的字段尽可能少，所以可以让排序和分页操作先查出主键，然后根据主键查到对应的记录，SQL 改写如下

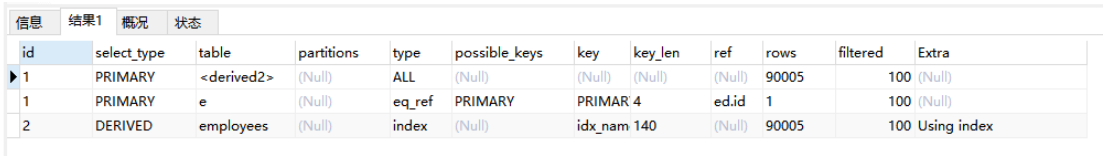
select * from employees e inner join (select id from employees order by name limit 90000,5) ed on e.id=ed.id;



id	name	age	position	hire_time	id1
91002	zhuge90996	90996	dev	2022-01-29 13:54:28	91002
91003	zhuge90997	90997	dev	2022-01-29 13:54:28	91003
91004	zhuge90998	90998	dev	2022-01-29 13:54:28	91004
91005	zhuge90999	90999	dev	2022-01-29 13:54:28	91005
97	zhuge91	91	dev	2022-01-29 13:50:27	97

select * from employees e inner join (select id from employees order by name limit 90000,5) ed on e.id = ed.id; 只读 查询时间: 0.039s

EXPLAIN select * from employees e inner join (select id from employees order by name limit 90000,5) ed on e.id=ed.id;



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	<derived2>	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	90005	100	(Null)
1	PRIMARY	e	(Null)	eq_ref	PRIMARY	PRIMAR	4	ed.id	1	100	(Null)
2	DERIVED	employees	(Null)	index	(Null)	idx_nam	140	(Null)	90005	100	Using index

需要的结果与原 SQL 一致，执行时间减少了一半以上，我们再对比优化前后 sql 的执行计划：

原 SQL 使用的是 filesort 排序，而优化后的 SQL 使用的是索引排序。

19.Join 关联查询优化

mysql 的表关联常见有两种算法

1、 嵌套循环连接 Nested-Loop Join(NLJ) 算法

一次一行循环地从第一张表（称为**驱动表**）中读取行，在这行数据中取到关联字段，根据关联字段在另一张表（**被驱动表**）里取出满足条件的行，然后取出两张表的结果合集。

EXPLAIN select * from t1 inner join t2 on t1.a= t2.a;

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t2	(Null)	ALL	idx_a	(Null)	(Null)	(Null)	100	100	Using where
1	SIMPLE	t1	(Null)	ref	idx_a	idx_a	5	lxg.t2.a	1	100	(Null)

从执行计划中可以看到这些信息：

（1）驱动表是 t2，被驱动表是 t1。先执行的就是驱动表(执行计划结果的 id 如果一样则按从上到下顺序执行 sql)；优化器一般会优先选择**小表做驱动表。所以使用 inner join 时，排在前面的表并不一定就是驱动表。**

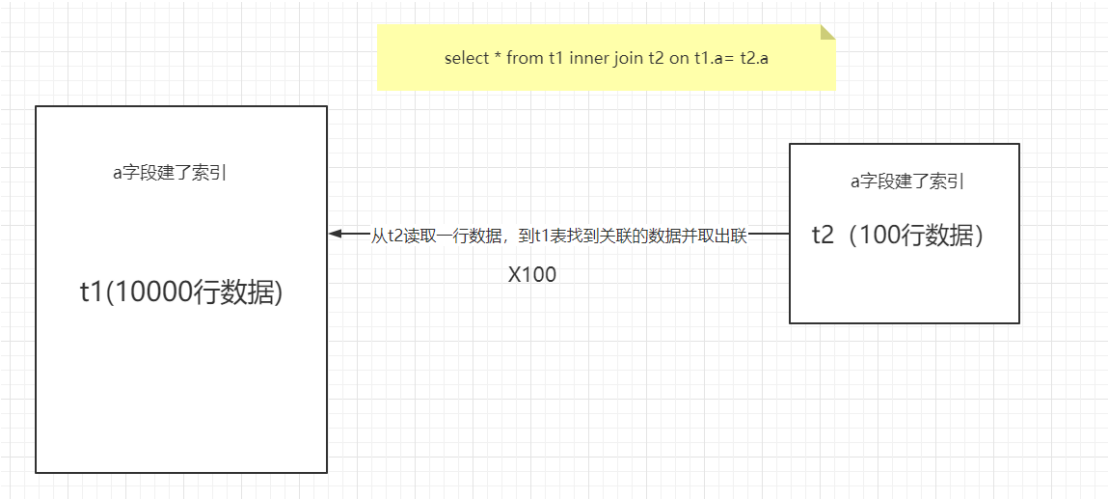
（2）当使用 left join 时，左表是驱动表，右表是被驱动表，当使用 right join 时，右表时驱动表，左表是被驱动表，当使用 join 时，mysql 会选择数据量比较小的表作为驱动表，大表作为被驱动表。

（3）使用了 NLJ 算法。一般 join 语句中，如果执行计划 Extra 中未出现 Using join buffer 则表示使用的 join 算法是 NLJ。

上面 sql 的大致流程如下：

1. 从表 t2 中读取一行数据（如果 t2 表有查询过滤条件的，会从过滤结果里取出一行数据）；
2. 从第 1 步的数据中，取出关联字段 a，到表 t1 中查找；
3. 取出表 t1 中满足条件的行，跟 t2 中获取到的结果合并，作为结果返回给客户端；
4. 重复上面 3 步。

整个过程会读取 t2 表的所有数据(**扫描 100 行**)，然后遍历这每行数据中字段 a 的值，根据 t2 表中 a 的值索引扫描 t1 表中的对应行(**扫描 100 次 t1 表的索引，1 次扫描可以认为最终只扫描 t1 表一行完整数据，也就是总共 t1 表也扫描了 100 行**)。因此整个过程扫描了 **200 行。**



如果被驱动表的关联字段没索引，**使用 NLJ 算法性能会比较低(下面有详细解释)**，mysql 会选择 Block Nested-Loop Join 算法。

2、 基于块的嵌套循环连接 Block Nested-Loop Join(BNL)算法

把**驱动表**的数据读入到 join_buffer 中，然后扫描**被驱动表**，把**被驱动表**每一行取出来跟 join_buffer 中的数据做对比。

EXPLAIN select * from t1 inner join t2 on t1.b= t2.b;

信息 结果1 概况 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t2	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	100	100	(Null)
1	SIMPLE	t1	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	10337	10	Using where; Using join buffer (Block Nested Loop)

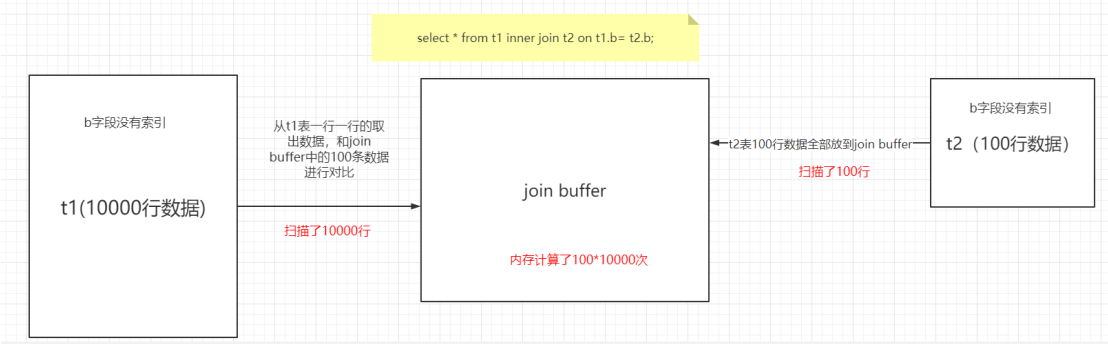
Extra 中 的 Using join buffer (Block Nested Loop)说明该关联查询使用的是 BNL 算法。

上面 sql 的大致流程如下：

1. 把 t2 的所有数据放入到 **join_buffer** 中
2. 把表 t1 中每一行取出来，跟 join_buffer 中的数据做对比
3. 返回满足 join 条件的数据

整个过程对表 t1 和 t2 都做了一次全表扫描，因此扫描的总行数为 10000(表 t1 的数据总量) + 100(表 t2 的数据总量) = **10100**。并且 join_buffer 里的数据是无序的，因此对表 t1 中的每一行，都要做 100 次判断，所以内存中的判断次数是

100 * 10000= **100 万次。**



这个例子里表 t2 才 100 行，要是表 t2 是一个大表，

join_buffer 放不下怎么办呢?

join_buffer 的大小是由参数 join_buffer_size 设定的，默认值是 256k。如果放不下表 t2 的所有数据话，策略很简单，就是**分段放。**

比如 t2 表有 1000 行记录， join_buffer 一次只能放 800 行数据，那么执行过程就是先往 join_buffer 里放 800 行记录，然 后从 t1 表里取数据跟 join_buffer 中数据对比得到部分结果，然后清空 join_buffer ，再放入 t2 表剩余 200 行记录，再次从 t1 表里取数据跟 join_buffer 中数据对比。所以就**多扫了一次 t1 表。**

被驱动表的关联字段没索引为什么要选择使用 BNL 算法而不使用 Nested-Loop Join 呢？

如果上面第二条 sql 使用 Nested-Loop Join，那么扫描行数为 100 * 10000 = 100 万次，这个是**磁盘扫描。**

很显然，用 BNL 磁盘扫描次数少很多，相比于磁盘扫描，**BNL 的内存计算会快得多。**

因此 MySQL 对于被驱动表的关联字段**没索引**的关联查询，一般都会使用 BNL 算法。如果有索引一般选择 NLJ 算法，有索引的情况下 NLJ 算法比 BNL 算法性能更高。

对于关联 sql 的优化

关联字段加索引，让 mysql 做 join 操作时尽量选择 NLJ 算法

小表驱动大表，写多表连接 sql 时如果**明确知道**哪张表是小表可以用 straight_join 写法固定连接驱动方式，省去 mysql 优化器自己判断的时间。

straight_join 解释：straight_join 功能同 join 类似，但能让左边的表来驱动右边的表，能改表优化器对于联表查询的执行顺序。

比如：select * from **t2 straight_join t1** on t2.a = t1.a; 代表指定 mysql 选着 **t2 表作为驱动表。**

straight_join 只适用于 inner join，并不适用于 left join，right join。（因为 left join，right join 已经代表指定了表的执行顺序）

尽可能让优化器去判断，因为大部分情况下 mysql 优化器是比人要聪明的。使用 **straight_join** 一定要慎重，因为部分情况下人为指定的执行顺序并不一定会比优化引擎要靠谱。

20.count 查询优化

```
-- 临时关闭 mysql 查询缓存，为了查看 sql 多次执行的真实时间
set global query_cache_size=0;
set global query_cache_type=0;

EXPLAIN select count(1) from employees;
EXPLAIN select count(id) from employees;
EXPLAIN select count(name) from employees;
EXPLAIN select count(*) from employees;
```

信息 | 结果1 | 结果2 | 结果3 | 结果4 | 概况 | 状态

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	employee	(Null)	index	(Null)	idx_name_age_position	140	(Null)	100082	100	Using index

4 个 sql 执行计划一样说明这四个 sql 执行效率应该差不多

注意：以上 4 条 sql 只有根据某个字段 count 不会统计字段为 null 值的数据行

深究执行效率，理论上：

count(*)≈count(1)>count(字段)>count(主键 id)

（1）count(1)跟 count(字段)执行过程类似，不过 count(1)不需要取出字段统计，就用常量 1 做统计，count(字段)还需要取出字段，所以理论上 count(1)比 count(字段)会快一点。

（2）count(*) 是例外，mysql 并不会把全部字段取出来，而是专门做了优化，不取值，按行累加，效率很高，所以不需要用 count(列名)或 count(常量)来替代 count(*)。

（3）为什么对于 count(id)，mysql 最终选择二级索引而不是主键聚集索引？因为二级索引相对主键索引存储数据更少，检索性能应该更高，mysql 内部做了点优化(应该是在 5.7 版本才优化)。

三、索引设计原则

1、代码先行，索引后上

建完表马上就建立索引吗？

这其实是不对的，一般应该等到主体业务功能开发完毕，把涉及到该表相关 sql 都要拿出来分析之后再建立

索引。

2、联合索引尽量覆盖条件

比如可以设计一个或者两三个联合索引(尽量少建单值索引)，让每一个联合索引都尽量去包含 sql 语句里的 where、order by、group by 的字段，还要确保这些联合索引的字段顺序尽量满足 sql 查询的最左前缀原则。

3、不要在小基数字段上建立索引

索引基数是指这个字段在表里总共有多少个不同的值，比如一张表总共 100 万行记录，其中有个性别字段，其值不是男就是女，那么该字段的基数就是 2。

如果对这种小基数字段建立索引的话，还不如全表扫描了，因为你的索引树里就包含男和女两种值，根本没法进行快速的二分查找，那用索引就没有太大的意义了。

一般建立索引，尽量使用那些基数比较大的字段，就是值比较多的字段，那么才能发挥出 B+树快速二分查找的优势来。

4、长字符串我们可以采用前缀索引

尽量对字段类型较小的列设计索引，比如说什么 tinyint 之类的，因为字段类型较小的话，占用磁盘空间也会比较小，此时你在搜索的时候性能也会比较好一点。

当然，这个所谓的字段类型小一点的列，也不是绝对的，很多时候你就是要针对 varchar(255)这种字段建立索引，哪怕多占用一些磁盘空间也是有必要的。

对于这种 varchar(255)的大字段可能会比较占用磁盘空间，可以稍微优化下，比如针对这个字段的前 20 个字符建立索引，就是说，对这个字段里的每个值的前 20 个字符放在索引树里，类似于 KEY index(name(20),age,position)。

此时你在 where 条件里搜索的时候，如果是根据 name 字段来搜索，那么此时就会先到索引树里根据 name 字段的前 20 个字符去搜索，定位到之后前 20 个字符的前缀匹配的部分数据之后，再回到聚簇索引提取出来完整的 name 字段值进行比对。

但是假如你要是 order by name，那么此时你的 name 因为在索引树里仅仅包含了前 20 个字符，所以这个排序是没法用上索引的， group by 也是同理。所以这里大家要对前缀索引有一个了解。

5、where 与 order by 冲突时优先 where

在 where 和 order by 出现索引设计冲突时，到底是针对 where 去设计索引，还是针对 order by 设计索引？到底是让 where 去用上索引，还是让 order by 用上索引?一般这种时候往往都是让 where 条件去使用索引来快速筛选出来一部分指定的数据，接着再进行排序。

因为大多数情况基于索引进行 where 筛选往往可以最快速度筛选出你要的少部分数据，然后做排序的成本可能会小很多。

四、基于慢 sql 查询做优化

可以根据监控后台的一些慢 sql，针对这些慢 sql 查询做特定的索引优化。

关于慢 sql 查询不清楚的可以参考这篇文章：https://blog.csdn.net/qq_40884473/article/details/89455740

五、MySQL 数据类型选择

1、数值类型

类型	大小	范围（有符号）	范围（无符号）	用途
TINYINT	1 字节	(-128, 127)	(0, 255)	小整数值

SMALLINT	2 字节	(-32 768, 32 767)	(0, 65 535)	大整数值
MEDIUMINT	3 字节	(-8 388 608, 8 388 607)	(0, 16 777 215)	大整数值
INT 或 INTEGER	4 字节	(-2 147 483 648, 2 147 483 647)	(0, 4 294 967 295)	大整数值
BIGINT	8 字节	(-9 233 372 036 854 775 808, 9 223 372 036 854 775 807)	(0, 18 446 744 073 709 551 615)	极大整数值
FLOAT	4 字节	(-3.402 823 466 E+38, 1.175 494 351 E-38)，0，(1.175 494 351 E-38，3.402 823 466 351 E+38)	0, (1.175 494 351 E-38, 3.402 823 466 E+38)	单精度 浮点数值
DOUBLE	8 字节	(1.797 693 134 862 315 7 E+308, 2.225 073 858 507 201 4 E-308), 0, (2.225 073 858 507 201 4 E-308, 1.797 693 134 862 315 7 E+308)	0, (2.225 073 858 507 201 4 E-308, 1.797 693 134 862 315 7 E+308)	双精度 浮点数值
DECIMAL	对 DECIMAL(M,D) ，如果 M>D，为 M+2 否则为 D+2	依赖于 M 和 D 的值	依赖于 M 和 D 的值	小数值

优化建议

1.如果整形数据没有负数，如 ID 号，建议指定为 UNSIGNED 无符号类型，容量可以扩大一倍。

2.建议使用 TINYINT 代替 ENUM、BITENUM、SET。

3.避免使用整数的显示宽度(参看文档最后)，也就是说，不要用 INT(10)类似的方法指定字段显示宽度，直接用 INT。

4.DECIMAL 最适合保存准确度要求高，而且用于计算的数据，比如价格。但是在使用 DECIMAL 类型的时候，注意长度设置。

5.建议使用整形类型来运算和存储实数，方法是，实数乘以相应的倍数后再操作。

6.整数通常是最佳的数据类型，因为它速度快，并且能使用 AUTO_INCREMENT。

2、日期和时间

类型	大小 (字节)	范围	格式	用途
DATE	3	1000-01-01 到 9999-12-31	YYYY-MM-DD	日期值
TIME	3	'-838:59:59' 到 '838:59:59'	HH:MM:SS	时间值或持续时间
YEAR	1	1901 到 2155	YYYY	年份值
DATETIME	8	1000-01-01 00:00:00 到 9999-12-31 23:59:59	YYYY-MM-DD HH:MM:SS	混合日期和时间值
TIMESTAMP	4	1970-01-01 00:00:00 到 2038-01-19 03:14:07	YYYYMMDDhhmmss	混合日期和时间值，时间戳

优化建议

1.MySQL 能存储的最小时间粒度为秒。

2.建议用 DATE 数据类型来保存日期。MySQL 中默认的日期格式是 yyyy-mm-dd。

3.用 MySQL 的内建类型 DATE、TIME、DATETIME 来存储时间，而不是使用字符串。

4.当数据格式为 TIMESTAMP 和 DATETIME 时，可以用 CURRENT_TIMESTAMP 作为默认（MySQL5.6 以后），MySQL 会自动返回记录插入的确切时间。

5.TIMESTAMP 是 UTC 时间戳，与时区相关。

6.DATETIME 的存储格式是一个 YYYYMMDD HH:MM:SS 的整数，与时区无关，你存了什么，读出来就是什么。

7.除非有特殊需求，一般的公司建议使用 TIMESTAMP，它比 DATETIME 更节约空间，但是像阿里这样的公司一般会用 DATETIME，因为不用考虑 TIMESTAMP 将来的时间上限问题。

8.有时人们把 Unix 的时间戳保存为整数值，但是这通常没有任何好处，这种格式处理起来不太方便，我们并不推荐它。

3、字符串

类型	大小	用途
CHAR	0-255 字节	定长字符串，char(n)当插入的字符数不足 n 时(n 代表字符数)，插入空格进行补充保存。在进行检索时，尾部的空格会被去掉。
VARCHAR	0-65535 字节	变长字符串，varchar(n)中的 n 代表最大字符数，插入的字符数不足 n 时不会补充空格
TINYBLOB	0-255 字节	不超过 255 个字符的二进制字符串
TINYTEXT	0-255 字节	短文本字符串
BLOB	0-65 535 字节	二进制形式的长文本数据
TEXT	0-65 535 字节	长文本数据
MEDIUMBLOB	0-16 777 215 字节	二进制形式的中等长度文本数据
MEDIUMTEXT	0-16 777 215 字节	中等长度文本数据
LONGBLOB	0-4 294 967 295 字节	二进制形式的极大文本数据
LONGTEXT	0-4 294 967 295 字节	极大文本数据

优化建议

1.字符串的长度相差较大用 VARCHAR；字符串短，且所有值都接近一个长度用 CHAR。

2.CHAR 和 VARCHAR 适用于包括人名、邮政编码、电话号码和不超过 255 个字符长度的任意字母数字组合。那些要用来计算的数字不要用 VARCHAR 类型保存，因为可能会导致一些与计算相关的问题。换句话说，可能影响到计算的准确性和完整性。

3.尽量少用 BLOB 和 TEXT，如果实在要用可以考虑将 BLOB 和 TEXT 字段单独存一张表，用 id 关联。

4.BLOB 系列存储二进制字符串，与字符集无关。TEXT 系列存储非二进制字符串，与字符集相关。

5.BLOB 和 TEXT 都不能有默认值。

PS：INT 显示宽度

我们经常会使用命令来创建数据表，而且同时会指定一个长度，如下。但是，这里的长度并非是 TINYINT 类型存储的最大长度，而是显示的最大长度。

如下建表 sql 语句：

```
CREATE TABLE `user`( `id` TINYINT(2) UNSIGNED );
```

这里 TINYINT(2)中 2 的作用就是，当需要在查询结果前填充 0 时，命令中加上 ZEROFILL 就可以实现，如：

`id` TINYINT(2) UNSIGNED ZEROFILL

这样，查询结果如果是 5，那输出就是 05。如果指定 TINYINT(5)，那输出就是 00005，其实实际存储的值还是 5，而且存储的数据不会超过 255，只是 MySQL 输出数据时在前面填充了 0。
换句话说，在 MySQL 命令中，字段的类型长度 TINYINT(2)、INT(11)不会影响数据的插入，**只会在使用 ZEROFILL 时有用，让查询结果前填充 0**。

navicat 不显示前边的 0，需要 cmd 命令行查看

```
mysql> select * from user;
+-------+
| id    |
+-------+
| 00001 |
| 00005 |
| 00003 |
+-------+
3 rows in set (0.00 sec)

mysql>
```

Sql 脚本

lxg.sql