Part 10 索引优化与查询优化

都有哪些维度可以进行数据库调优?简言之:

- 索引失效、没有充分利用到索引——索引建立
- 关联查询太多JOIN (设计缺陷或不得已的需求)——SQL优化
- 服务器调优及各个参数设置(缓冲、线程数等)————调整my.cnf
- 数据过多——分库分表

虽然SQL查询优化的技术有很多，但是大方向上完全可以分成`物理查询优化`和`逻辑查询优化`两大块。

- 物理查询优化是通过`索引`和`表连接方式`等技术来进行优化，这里重点需要掌握索引的使用。
- 逻辑查询优化就是通过SQL`等价变换`提升查询效率，直白一点就是说，换一种查询写法执行效率可能更高。

1. 索引失效案例

大多数情况下都（默认）采用`B+树`来构建索引。只是空间列类型的索引使用`R-树`，并且MEMORY表还支持`hash索引`。其实，用不用索引，最终都是优化器说了算。优化器是基于`cost开销(CostBaseOptimizer)`，它不是基于`规则(Rule-BasedOptimizer)`，也不是基于`语义`。怎么样开销小就怎么来。另外，**SQL语句是否使用索引，跟数据库版本、数据量、数据选择度都有关系。**开销也不止是基于时间。

1.1 全值匹配我最爱

意思是创建联合索引多个索引同时生效。索引帮助我们极大的提高了查询效率。

1.2 最佳左前缀法则

在MySQL建立联合索引时会遵守最佳左前缀匹配原则，即最左优先，在检索数据时从联合索引的最左边开始匹配。

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。

MySQL可以为多个字段创建索引，一个索引可以包括16个字段。对于多列索引，**过滤条件要使用索引必须按照索引建立时的顺序，依次满足，一旦跳过某个字段，索引后面的字段都无法被使用。**如果查询条件中没有使用这些字段中第1个字段时，多列(或联合）索引不会被使用。

1.3 主键插入顺序

如果按主键值从小到大排序的记录连续存储在一个数据页中，并且该数据页存储的记录已经满了，则希望插入主键值在该页范围内的记录时，只能将当前页面分裂成两个页面。而**页面分裂**和**记录移位**就导致**性能损耗**，为避免这样的性能损耗，最好让插入的记录的**主键值依次递增。**建议让主键具有`AUTO_INCREMENT`，让存储引擎自己为表生成主键，而不是我们手动插入。这样的主键占用空间小，顺序写入，减少页分裂。

1.4 计算、函数、类型转换(自动或手动)导致索引失效

1.4.1 使用函数导致的索引失效

举例

```
EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE student.name LIKE 'abc%';
# 时间为0.01秒
# 索引有效

EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE LEFT(student.name,3) =
'abc';
# 时间为3.62秒
# type为“ALL”，表示没有使用到索引。这个索引失效，因为用上函数了。
```

1.4.2 计算导致的索引失效

举例

```
EXPLAIN SELECT SQL_NO_CACHE id, stuno, NAME FROM student WHERE stuno + 1 =
900001;
# 计算导致索引失效
# type为“ALL”，表示没有使用到索引。
```

```
mysql> EXPLAIN SELECT SQL_NO_CACHE id, stuno, NAME FROM student WHERE stuno+1 = 900001;
+----+-------------+---------+------------+------+---------------+------+---------+------+---------+----------+-------------+
| id | select_type | table   | partitions | type | possible_keys | key  | key_len | ref  | rows    | filtered | Extra       |
+----+-------------+---------+------------+------+---------------+------+---------+------+---------+----------+-------------+
|  1 | SIMPLE      | student | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 1069943 |   100.00 | Using where |
+----+-------------+---------+------------+------+---------------+------+---------+------+---------+----------+-------------+
1 row in set, 2 warnings (0.05 sec)
```

```
EXPLAIN SELECT SQL_NO_CACHE id, stuno, NAME FROM student WHERE stuno =
900000;
# 索引有效
```

```
mysql> EXPLAIN SELECT SQL_NO_CACHE id, stuno, NAME FROM student WHERE stuno = 900001+1;
+----+-------------+---------+------------+------+---------------+---------+---------+-------+------+----------+-------+
| id | select_type | table   | partitions | type | possible_keys | key     | key_len | ref   | rows | filtered | Extra |
+----+-------------+---------+------------+------+---------------+---------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | student | NULL       | ref  | idx_sno       | idx_sno | 4       | const |    1 |   100.00 | NULL  |
+----+-------------+---------+------------+------+---------------+---------+---------+-------+------+----------+-------+
1 row in set, 2 warnings (0.01 sec)
```

1.4.3 类型转换导致索引失效

举例

```
# name字段类型为VARCHAR(20)
# 未发生类型转换，索引有效
EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE name='123';

# 未使用到索引
# name=123发生类型转换，索引失效
EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE name=123;
```

1.5 范围条件右边的列索引失效

哪些属于范围?

1. 大于等于，大于，小于等于，小于
2. `between`

```
# 联合索引顺序为age，classId，name
# classId使用了范围查找，范围查找后面的索引就失效了。
EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE student.age=30 AND
student.classId>20 AND student.name = 'abc' ;
```

> tips： 因为范围条件导致的索引失效，可以考虑把确定的索引放在前面。
>
> ```
> create index idx_age_name_cid on student(age, name, classId);
> ```
>
> 将name放在范围查找classId前面，索引就能生效了。

> 应用开发中范围查询，例如： 金额查询，日期查询往往都是范围查询。创建联合索引时考虑放在后面。

1.6 不等于(!= 或者<>)索引失效

索引只能查到知道的东西

举例

```
EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE student.name !='abc';
```

```
mysql> EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE student.name !='abc';
+----+-------------+---------+------------+------+---------------+------+---------+------+---------+
| id | select_type | table   | partitions | type | possible_keys | key  | key_len | ref  | rows    |
+----+-------------+---------+------------+------+---------------+------+---------+------+---------+
|  1 | SIMPLE      | student | NULL       | ALL  | idx_name      | NULL | NULL    | NULL | 1069943 |
+----+-------------+---------+------------+------+---------------+------+---------+------+---------+
1 row in set, 2 warnings (0.00 sec)
```

1.7 is null可以使用索引，is not null无法使用索引

这里不举例了。

结论：最好在设计数据表的时候就将`字段设置为 NOT NULL 约束`，比如你可以将INT类型的字段，默认值设置为0。将字符类型的默认值设置为空字符串。

拓展：同理，在查询中使用`not like`也无法使用索引，导致全表扫描。

1.8 like以通配符%开头索引失效

在使用LIKE关键字进行查询的查询语句中，如果匹配字符串的第一个字符为“%”，索引就不会起作用。只有“%"不在第一个位置，索引才会起作用。

1.9 OR 前后存在非索引的列，索引失效

在WHERE子句中，如果在OR前的条件列进行了索引，而在OR后的条件列没有进行索引，那么索引会失效。也就是说，**OR前后的两个条件中的列都是索引时，查询中才使用索引。**

因为OR的含义就是两个只要满足一个即可，`因此只有一个条件列进行了索引是没有意义的`，只要有条件列没有进行索引，就会进行全表扫描，因此索引的条件列也会失效。

> 只要一个需要全表扫描，那就全部都全表扫描，因为全表扫描的代价可能比全表扫描加索引更小。

1.10 数据库和表的字符集统一使用utf8mb4

统一使用utf8mb4(5.5.3版本以上支持)兼容性更好，统一字符集可以避免由于字符集转换产生的乱码。不同的 字符集 进行比较前需要进行 `转换` 会造成索引失效。

1.11 练习及一般性建议

练习:

假设index(a,b,c)

Where语句	索引是否被使用
where a = 3	Y,使用到a
where a = 3 and b = 5	Y,使用到a，b
where a = 3 and b = 5 and c = 4	Y,使用到a,b,c
where b = 3 或者 where b = 3 and c = 4 或者 where c = 4	N
where a = 3 and c = 5	使用到a， 但是c不可以，b中间断了
where a = 3 and b > 4 and c = 5	使用到a和b， c不能用在范围之后，b断了
where a is null and b is not null	is null 支持索引 但是is not null 不支持。所以 a 可以使用索引,但是 b不可以使用
where a <> 3	不能使用索引
where abs(a) =3	不能使用 索引
where a = 3 and b like 'kk%' and c = 4	Y,使用到a,b,c
where a = 3 and b like '%kk' and c = 4	Y,只用到a
where a = 3 and b like '%kk%' and c = 4	Y,只用到a
where a = 3 and b like 'k%kk%' and c = 4	Y,使用到a,b,c

一般性建议:

- 对于单列索引，尽量选择针对当前query过滤性更好的索引
- 在选择组合索引的时候，当前query中过滤性最好的字段在索引字段顺序中，位置越靠前越好。
- 索引的时候，尽量选择能够包含当前query中的where子句中更多字段的索引。

- 在选择组合索引的时候，如果某个字段可能出现范围查询时，尽量把这个字段放在索引次序的最后面。

总之，书写SQL语句时，尽量避免造成索引失效的情况。

2. 关联查询优化

外连接和内连接

- 内连接，也被称为自然连接，只有两个表相匹配的行才能在结果集中出现。
- 外连接，不仅包含符合连接条件的行，还包含左表(左连接时)、右表(右连接时)或两个边接表(全外连接)中的所有数据行。

2.1 数据准备

```
#分类
CREATE TABLE IF NOT EXISTS `type`(
`id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
`card` INT(10) UNSIGNED NOT NULL,
PRIMARY KEY ( `id` )
);

#图书
CREATE TABLE IF NOT EXISTS `book`(
    `bookid` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
    `card`INT(10) UNSIGNED NOT NULL,
    PRIMARY KEY (`bookid`)
);


#向分类表中添加20条记录
INSERT INTO type (card) VALUES (FLOOR(1 +(RAND() * 20)));

#向图书表中添加20条记录
INSERT INTO book(card) VALUES (FLOOR(1 +(RAND() * 20)) );
```

2.2 采用左外连接

- 未建立索引时

```
EXPLAIN SELECT SQL_NO_CACHE * FROM `type` LEFT JOIN book ON type.card = book.card;
```

```
mysql> EXPLAIN SELECT SQL_NO_CACHE * FROM `type` LEFT JOIN book ON type.card = book.card;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                               |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------------------+
|  1 | SIMPLE      | type  | NULL       | ALL  | NULL          | NULL | NULL    | NULL |   20 |   100.00 | NULL                                                |
|  1 | SIMPLE      | book  | NULL       | ALL  | NULL          | NULL | NULL    | NULL |   20 |   100.00 | Using where; Using join buffer (hash join) |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------------------+
```

- 被驱动表添加索引优化

```sql
# 添加索引,【被驱动表】，可以避免全表扫描
ALTER TABLE book ADD INDEX Y(card);

EXPLAIN SELECT SQL_NO_CACHE * FROM `type` LEFT JOIN book ON type.card =
book.card;
```

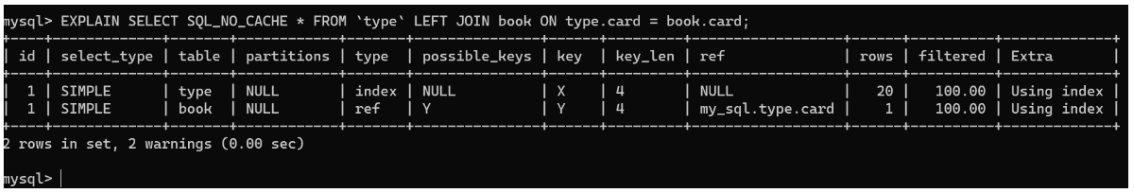
可以看到第二行的 type 变为了 ref，rows 也变成了优化比较明显。这是由左连接特性决定的。LEFT JOIN条件用于确定如何从右表搜索行，左边一定都有，所以 右边是我们的关键点,一定需要建立索引 。

> 如果只能添加一边的索引，，那就给被驱动表添加上索引。

- 驱动表添加索引优化（仅限出现索引覆盖的情况）

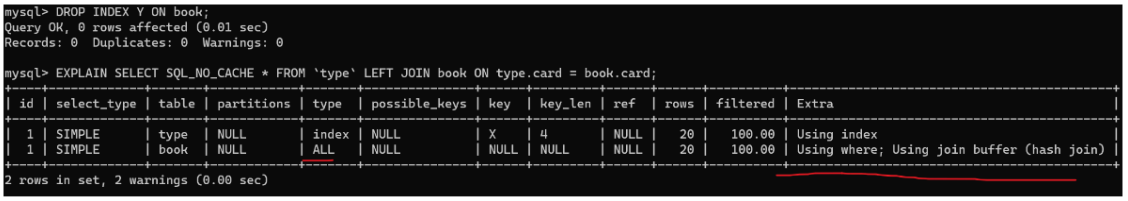
```sql
ALTER TABLE `type` ADD INDEX X (card); #【驱动表】，无法避免全表扫描

EXPLAIN SELECT SQL_NO_CACHE * FROM `type` LEFT JOIN book ON type.card =
book.card;
```



这里表`type`一共就两个字段，主键已有索引，给card建立索引，索引处依然会存有主键。使用这个索引，会出现索引覆盖的情况，无需回表，也比全表扫描更快。

- 若去掉被驱动索引，又变成了 join buffer



2.3 采用内连接

内连接时，MySQL自动选择驱动表。并不是谁在前谁是驱动表。

- 未建立索引时

```sql
EXPLAIN SELECT SQL_NO_CACHE * FROM type INNER JOIN book ON
type.card=book.card;
```

```
mysql> EXPLAIN SELECT SQL_NO_CACHE * FROM type INNER JOIN book ON type.card=book.card;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                              |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
|  1 | SIMPLE      | type  | NULL       | ALL  | NULL          | NULL | NULL    | NULL |   20 |   100.00 | NULL                                               |
|  1 | SIMPLE      | book  | NULL       | ALL  | NULL          | NULL | NULL    | NULL |   20 |    10.00 | Using where; Using join buffer (hash join)         |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
2 rows in set, 2 warnings (0.00 sec)
```

没有用到索引。

- 添加索引优化

```sql
# 加上索引的表变成被驱动表
ALTER TABLE book ADD INDEX Y (card);
EXPLAIN SELECT SQL_NO_CACHE * FROM type INNER JOIN book ON
type.card=book.card;
```

```
rows in set, 2 warnings (0.00 sec)

ysql> ALTER TABLE book ADD INDEX Y (card);
uery OK, 0 rows affected (0.07 sec)
ecords: 0  Duplicates: 0  Warnings: 0

ysql>
ysql> EXPLAIN SELECT SQL_NO_CACHE * FROM type INNER JOIN book ON type.card=book.card;
+----+-------------+-------+------------+------+---------------+------+---------+-------------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref         | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+-------------+------+----------+-------------+
|  1 | SIMPLE      | type  | NULL       | ALL  | NULL          | NULL | NULL    | NULL        |   20 |   100.00 | NULL        |
|  1 | SIMPLE      | book  | NULL       | ref  | Y             | Y    | 4       | my_sql.type.card |    1 |   100.00 | Using index |
+----+-------------+-------+------------+------+---------------+------+---------+-------------+------+----------+-------------+
2 rows in set, 2 warnings (0.00 sec)
```

添加索引后，book 自动变成被驱动表

- 给另一个表添加索引优化

```sql
# type 加索引
ALTER TABLE type ADD INDEX X (card);
# 观察执行情况
EXPLAIN SELECT SQL_NO_CACHE * FROM type INNER JOIN book ON
type.card=book.card;
```

```
mysql> # type 加索引
mysql> ALTER TABLE type ADD INDEX X (card);
Query OK, 0 rows affected (0.08 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> # 观察执行情况
mysql> EXPLAIN SELECT SQL_NO_CACHE * FROM type INNER JOIN book ON type.card=book.card;
+----+-------------+-------+------------+-------+---------------+------+---------+-------------+------+----------+-------------+
| id | select_type | table | partitions | type  | possible_keys | key  | key_len | ref         | rows | filtered | Extra       |
+----+-------------+-------+------------+-------+---------------+------+---------+-------------+------+----------+-------------+
|  1 | SIMPLE      | type  | NULL       | index | X             | X    | 4       | NULL        |   20 |   100.00 | Using index |
|  1 | SIMPLE      | book  | NULL       | ref   | Y             | Y    | 4       | my_sql.type.card |    1 |   100.00 | Using index |
+----+-------------+-------+------------+-------+---------------+------+---------+-------------+------+----------+-------------+
2 rows in set, 2 warnings (0.00 sec)
```

给 type 添加数据后，驱动表关系就变。

```
mysql> EXPLAIN SELECT SQL_NO_CACHE * FROM type INNER JOIN book ON type.card=book.card;
+----+-------------+-------+------------+-------+---------------+------+---------+-------------+------+----------+-------------+
| id | select_type | table | partitions | type  | possible_keys | key  | key_len | ref         | rows | filtered | Extra       |
+----+-------------+-------+------------+-------+---------------+------+---------+-------------+------+----------+-------------+
|  1 | SIMPLE      | book  | NULL       | index | Y             | Y    | 4       | NULL        |   20 |   100.00 | Using index |
|  1 | SIMPLE      | type  | NULL       | ref   | X             | X    | 4       | my_sql.book.card |    1 |   100.00 | Using index |
+----+-------------+-------+------------+-------+---------------+------+---------+-------------+------+----------+-------------+
2 rows in set, 2 warnings (0.00 sec)
```

驱动表关系变化，这里是优化器判断的，哪个数据少，就作为驱动表。

结论：

- `内连接` 主被驱动表是由优化器决定的。优化器认为哪个成本比较小，就采用哪种作为驱动表。
- 如果两张表只有一个有索引，那有索引的表作为 `被驱动表`。
  - 原因：驱动表要全查出来。有没有索引你都得全查出来。
- 两个索引都存在的情况下， 数据量大的 作为 `被驱动表` （小表驱动大表）
  - 原因：驱动表要全部查出来，而大表可以通过索引加快查找

2.4 join语句原理

join方式连接多个表，本质就是各个表之间数据的循环匹配。MySQL5.5欣本之刖，MySQL只文持一种表间关联方式，就是嵌套循环(`Nested Loop Join`)。如果关联表的数据量很大，则join关联的执行时间会非常长。在MySQL 5.5以后的版本中，MySQL通过引入`BNLJ`算法来优化嵌套执行。

2.4.1 驱动表和被驱动表

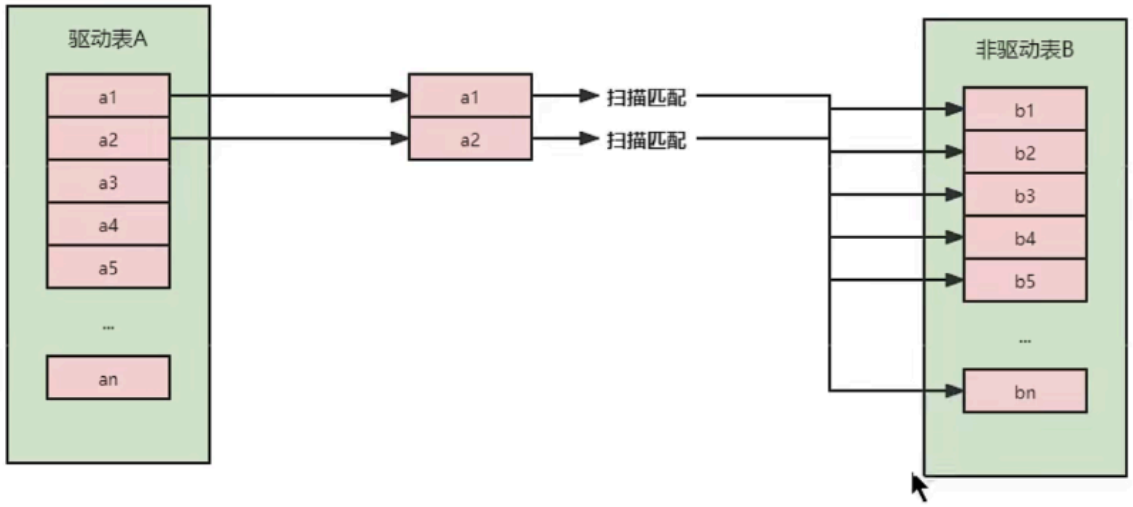
驱动表就是主表，被驱动表就是从表、非驱动表。

- 内连接的驱动表关系是由优化器决定的。
- 外连接

```
SELECT * FROM A LEFT JOIN B ON ...
#或
SELECT *FROM B RIGHT JOIN A ON ...
```

通常会认为A是驱动表，B是被驱动表，但也不一定。有时优化器可能会将外连接转化成内连接执行。

2.4.2 Simple Nested-Loop Join(简单嵌套循环连接)



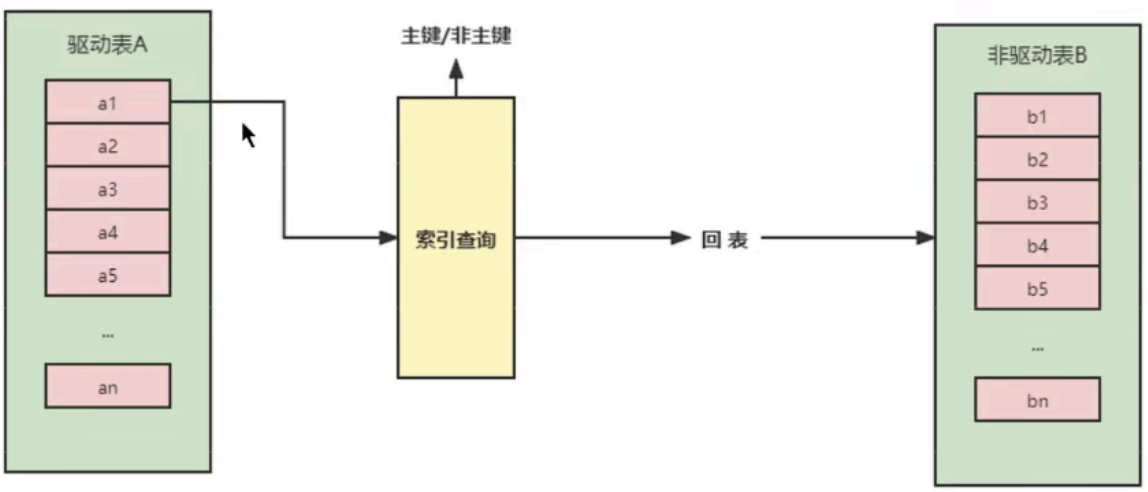
在没有索引的情况下，做全表扫描。效率极低，下面有两种优化。

2.4.3 Index Nested-Loop Join(索引嵌套循环连接)

优化思路：为了**减少内层表数据的匹配次数**，所以要求**被驱动表**上必须**有索引**才行。会极大的减少对内层表的匹配次数。

驱动表中的每条记录通过被驱动表的索引进行访问，因为索引查询的成本是比较固定的，故mysql优化器都倾向于使用**记录数少的表**作为驱动表(外表)。

当索引不是主键索引时，还要回表查询，因此，主键索引效率会更高。

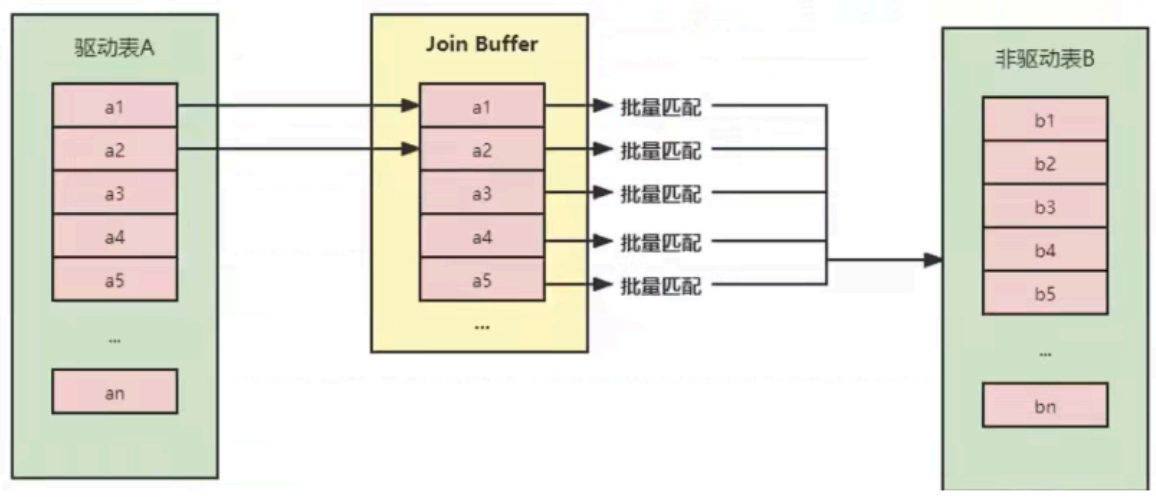


2.4.4 Block Nested-Loop Join(块嵌套循环连接)

若没有索引，在驱动表的每条数据匹配被驱动表时，被驱动表中的数据可能会不断读入读出，增加了IO次数。块嵌套循环连接可以减少被驱动表的IO次数。

不再是逐条获取驱动表的数据，而是一块一块的获取，引入了`join buffer缓冲区`，将`驱动表join`相关的部分数据列(大小受join buffer的限制)缓存到join buffer中，然后全表扫描被驱动表被驱动表的每一条记录一次性和**join buffer**中的所有驱动表记录进行匹配（`内存中操作`），将简单嵌套循环中的多次比较合并成一次，降低了被驱动表的访问频率。

> 注意：
>
> 这里缓存的不只是关联表的列, select后面的列也会缓存起来。**（存的是驱动表）**。在一个有N个join关联的sql中会分配N-1个join buffer。所以查询的时候尽量减少不必要的字段，可以让joinbuffer中可以存放更多的列。



- `block_nested_loop`

  该算法的是否开启。可以通过`show variables like '%optimizer_switch%'`查看`block_nested_loop`状态。block_nested_loop为on，默认是开启的。

- `join_buffer_size`

  驱动表能不能一次加载完，要看join buffer能不能存储所有的数据，默认情况下`join_buffer_size=256k`。

join_buffer_size的最大值在32位系统可以电请4G，而在64位操做系统下可以申请大于4G的Join Buffer空间(64位Windows除外，其大值会被截断为4GB并发出警告)。

2.4.5 Join小结

- 整体效率比较:INLJ > BNLJ > SNLJ
- 永远用小结果集驱动大结果集(其本质就是减少外层循环的数据数量)(小的度量单位指的是表行数*每行大小)
- 使用straight_join连接时，永远是前面的驱动后面的。因此，这里尽量将列少的放前面。
- 为被驱动表匹配的条件增加索引(减少内层表的循环匹配次数)
- 增大join buffer size的大小(一次缓存的数据越多，那么内层包的扫表次数就越少)
- 减少`驱动表`不必要的字段查询（字段越少，join buffer 所缓存的数据就越多)
- 在决定哪个表做驱动表的时候，应该是两个表按照各自的条件过滤，过滤完成之后，计算参与join的各个字段的总数据量，数据量小的那个表，就是“小表”，应该作为驱动表。

2.5 小结

- 保证被驱动表的JOIN字段已经创建了索引
- 需要JOIN的字段，数据类型保持绝对一致。（这样才能用索引）
- LEFT JOIN 时，选择小表作为驱动表，`大表作为被驱动表`。减少外层循环的次数。
- INNER JOIN 时，MySQL会自动将`小结果集的表选为驱动表`。选择相信MySQL优化策略。
- 能够直接多表关联的尽量直接关联，不用子查询。(减少查询的趟数)
- 不建议使用子查询，建议将子查询SQL拆开结合程序多次查询，或使用 JOIN 来代替子查询。
- 衍生表建不了索引

2.6 Hash Join

从MySQL的8.0.20版本开始将废弃BNLJ，因为从MySQL8.0.18版本开始就加入了hash join默认都会使用hash join。

- Nested Loop:
  对于被连接的数据子集较小的情况，Nested Loop是个较好的选择。
- Hash Join是做`大数据集连接`时的常用方式，优化器使用两个表中较小(相对较小)的表利用Join Key在内存中建立`散列表`，然后扫描较大的表并探测散列表，找出与Hash表匹配的行。
  - 这种方式适用于较小的表完全可以放于内存中的情况，这样总成本就是访问两个表的成本之和。
  - 在表很大的情况下并不能完全放入内存，这时优化器会将它分割成`若干不同的分区`，不能放入内存的部分就把该分区写入磁盘的临时段，此时要求有较大的临时段从而尽量提高I/O的性能。
  - 它能够很好的工作于没有索引的大表和并行查询的环境中，并提供最好的性能。大多数人都说它是Join的重型升降机。Hash Join只能应用于等值连接(如WHERE A.COL1=B.COL2)，这是由Hash的特点决定的。

类别	Nested Loop	Hash Join
使用条件	任何条件	等值连接（=）
相关资源	CPU、磁盘I/O	内存、临时空间
特点	当有高选择性索引或进行限制性搜索时效率比较高，能够快速返回第一次的搜索结果。	当缺乏索引或者索引条件模糊时，Hash Join比Nested Loop有效。在数据仓库环境下，如果表的纪录数多，效率高。
缺点	当索引丢失或者查询条件限制不够时，效率很低；当表的纪录数多时，效率低。	为建立哈希表，需要大量内存。第一次的结果返回较慢。

3. 子查询优化

MySQL从4.1版本开始支持子查询，使用子查询可以进行SELECT语句的嵌套查询，即一个SELECT查询的结果作为另一个SELECT语句的条件。`子查询可以一次性完成很多逻辑上需要多个步骤才能完成的SQL操作`。

子查询是MySQL的一项重要的功能，可以帮助我们通过一个SQL语句实现比较复杂的查询。但是，子查询的执行效率不高。原因:

①执行子查询时MySQL需要为内层查询语句的查询结果`建立一个临时表`，然后外层查询语句从临时表中查询记录。查询完毕后，再`撤销这些临时表`。这样会消耗过多的CPU和IO资源，产生大量的慢查询。

②子查询的结果集存储的临时表，不论是内存临时表还是磁盘临时表都`不会存在索引`，所以查询性能会受到一定的影响。

③对于返回结果集比较大的子查询，其对查询性能的影响也就越大。

在MySQL中，可以使用连接（JOIN）查询来替代子查询。连接查询不需要`建立临时表`，其`速度比子查询要快`，如果查询中使用索引的话，性能就会更好。

举例1:查询学生表中是班长的学生信息

- 使用子查询

```
#创建班级表中班长的索引
CREATE INDEX idx_monitor ON class ( monitor ) ;
EXPLAIN SELECT *FROM student stu1
WHERE stu1 . 'stuno`IN(
SELECT monitor
FROM class c
WHERE monitor IS NOT NULL);
```

- 推荐:使用多表查询

```
EXPLAIN SELECT stu1.* FROM student stu1 JOIN class c
ON stu1 . 'stuno` = c. 'monitor'
WHERE c. 'monitor` IS NOT NULL;
```

举例2:取所有不为班长的同学·不推荐

- 子查询

```
EXPLAIN SELECT SQL_NO_CACHE a.* FROM student a
WHERE a.stuno NOT IN (
SELECT monitor FROM class bWHERE monitor IS NOT NULL);
```

- 修改成多表查询

```
EXPLAIN SELECT SQL_NO_CACHE a.*
FROM student a LEFT OUTER JOIN class b ON a. stuno =b.monitor
WHERE b.monitor IS NULL;
```

> 结论: 尽量不要使用NOT IN或者NOT EXISTS，用LEFT JOIN Xxx ON xx WHERE xx IS NULL替代。

4. 排序优化

4.1 排序优化

问题: 在WHERE 条件字段上加索引但是为什么在ORDER BY字段上还要加索引呢?

回答:

在MySQL中，支持两种排序方式，分别是`FileSort`和`Index`排序。

- Index排序中，索引可以保证数据的有序性，不需要再进行排序，`效率更高`。
- FileSort排序则一般在`内存中`进行排序，占用`CPU较多`。如果待排结果较大，会产生临时文件I/O到磁盘进行排序的情况，效率较低。

优化建议:

1. 在WHERE子句中`避免全表扫描`，在ORDER BY子句`避免使用FileSort排序`。当然，某些情况下全表扫描，或者FileSort排序不一定比索引慢。但总的来说，我们还是要避免，以提高查询效率。
2. 尽量使用Index完成ORDER BY排序。如果WHERE和ORDER BY后面是相同的列就使用单索引列；如果不同就使用联合索引。
3. 无法使用Index时，需要对FileSort方式进行调优。

4.1.1 order by对索引的使用情况

- 情况一
  - 没有索引时，排序只能`using filesort`
- 情况二
  - 有索引时，如果没有limit
    - 索引失效：因为可能优化器认为还要回表，开销更大，不如直接`using filesort`
    - 索引覆盖，不失效：如果索引覆盖，不用回表，则优化器会认为，索引更快，使用索引。

    - 有索引，有limit
        - 增加limit 减少回表的数量，优化器觉得走索引快，会使用索引
- 情况三
    - 符合最左前缀原则才能用索引
- 情况四
    - 与索引的排序方向相同才能用索引
- 情况五
    - 没有limit过滤，不用索引？？？（这不是和情况二一样吗）

4.1.2 小结

```
INDEX a_b_c( a, b,c)

# order by 能使用索引最左前缀
- ORDER BY a
- ORDER BY a, b
- ORDER BY a , b, c
- ORDER BY a DESC, b DESC,c DESC

# 如果WHERE使用索引的最左前缀定义为常量，则order by 能使用索引
- WHERE a = const ORDER BY b, c
- WHERE a = const AND b = const ORDER BY c
- WHERE a = const ORDER BY b, c
- WHERE a = const AND b > const ORDER BY b , c

# 不能使用索引进行排序
- ORDER BY a ASC, b DESC, c DESC/*排序不一致*/
- WHERE g = const ORDER BY b,c/*丢失a索引*/
- WHERE a = const ORDER BY c/*丢失b索引*/
- WHERE a = const ORDER BY a, d /*d不是索引的一部分*/
- WHERE a in (...) ORDER BY b,c /*对于排序来说，多个相等条件也是范围查询*/
```

> 索引只会用到一个，没办法一个索引用来where 一个索引用来 order by。但是可以建立联合索引。

4.2 案例实战

如果出现最坏情况，比如type是ALL，还有Using filesort，则优化方案：

- 方案一：为了去掉filesort我们可以把索引建成

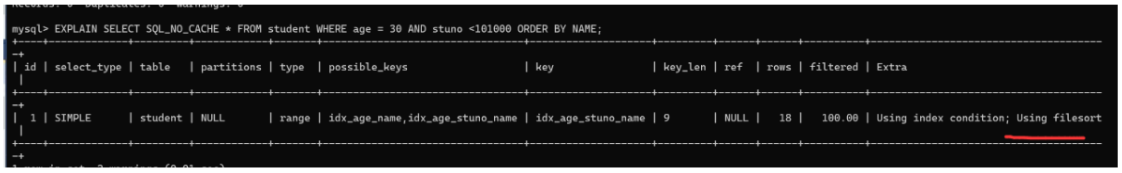
```
#创建新索引
CREATE INDEX idx_age_name ON student(age , NAME);

EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE age = 30 AND stuno <101000 ORDER BY NAME;
```

```
mysql>
mysql> EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE age = 30 AND stuno <101000 ORDER BY NAME;
+----+-------------+---------+------------+------+---------------+---------------+---------+-------+-------+----------+-------------+
| id | select_type | table   | partitions | type | possible_keys | key           | key_len | ref   | rows  | filtered | Extra       |
+----+-------------+---------+------------+------+---------------+---------------+---------+-------+-------+----------+-------------+
|  1 | SIMPLE      | student | NULL       | ref  | idx_age_name  | idx_age_name | 5       | const | 40558 |    33.33 | Using where |
+----+-------------+---------+------------+------+---------------+---------------+---------+-------+-------+----------+-------------+
1 row in set, 2 warnings (0.00 sec)
```

- 方案二：尽量让where的过滤条件和排序使用上索引

```
create index idx_age_stuno_name on student(age,stuno,name);
EXPLAIN SELECT SQL_NO_CACHE * FROM student WHERE age = 30 AND stuno <101000 ORDER BY NAME;
```



该方案使用了`Using filesort`但是速度反而更快了。

因为排序是在条件过滤之后执行的，如果大部分数据已经过滤掉，则后面排序也不是很消耗性能。而这里如果存在索引优化，其实际提升性能也有限。

其实就是考虑将索引给谁效果更好的问题。

> 结论:
>
> - 两个索引同时存在，mysql自动选择最优的方案。(对于这个例子mysql选择idx_age_stuno_name)。但是，`随着数据量的变化，选择的索引也会随之变化的。`
> - **当【范围条件】和【group by或者order by】的字段出现二选一时，优先观察条件字段的过滤数量，如果过滤的数据足够多，而需要排序的数据并不多时，优先把索引放在范围字段上。反之，亦然。**

4.3 filesort算法:双路排序和单路排序

4.3.1 双路排序

`MySQL 4.1之前是使用双路排序`，字面意思就是两次扫描磁盘。第一次扫描磁盘读取行指针和order by列排好序，第二次根据排好序的列表读出整条记录。

类似回表。

4.3.2 单路排序

只读一次磁盘，从磁盘中读出所有需要的列，按照order by列在buffer对它们进行排序。效率更高，但使用了更大的空间。

单路排序的问题：单路排序需要要求取出的数据的总大小不超过`sort_buffer`的容量，如果超出，需要创建tmp文件，将已排好的存储，继续排剩下的，会造成多次IO，得不偿失。

4.3.3 对单路排序的优化策略

- 尝试提高sort_buffer_size
- 尝试提高max_length_for_sort_data
  - 该参数是决定用单路还是用双路，超过则用双路。但该值也不能过大，过大会出现单路排序的问题。
- Order by时select*是一个大忌。最好只Query需要的字段。
  - 多余的无用字段会占用缓冲区的空间，使得本可以使用单路排序解决的问题使用了双路排序。

- 两种算法都可能会有超出sort_buffer_size的风险，而单路的超出代价比双路的高，索引还是尽量提高sort_buffer_size。

5. GROUP BY优化

- group by使用索引的原则几乎跟order by一致，group by即使没有过滤条件用到索引，也可以直接使用索引。
- group by先排序再分组，遵照索引建的最佳左前缀法则。
- 当无法使用索引列，增大`max_length_for_sort_data`和`sort_buffer_size`参数的设置
- where效率高于having，能写在where限定的条件就不要写在having中了
- 减少使用order by，和业务沟通能不排序就不排序，或将排序放到程序端去做
- Order by、group by、distinct这些语句较为耗费CPU，数据库的CPU资源是极其宝贵的。如果包含了order by、group by、distinct这些查询，where条件过滤出来的结果集尽量保持在1000行以内，否则SQL会很慢。

6. 优化分页查询

一般分页查询时，通过创建覆盖索引能够比较好地提高性能。但若在分页查询中取靠后一小段的**完整记录**，则前大半部分无用，代价很大。下面给出两个优化思路。

6.1 主键索引

在索引上完成排序分页操作，最后根据主键关联回原表查询所需要的其他列内容。

6.2 主键自增

该方案适用于主键自增的表，可以把Limit查询转换成某个位置的查询。直接去主键索引中找到对应位置取记录即可。

> 不靠谱，生产中id可能会删除，查询的条件也不可能这么简单。

7. 优先考虑覆盖索引

一个索引包含了满足查询结果的数据就叫做覆盖索引。非聚簇复合索引的一种形式，`索引列 + 主键`包含`SELECT 到 FROM之间查询的列`。

7.1 举例

这里就不举例了，上网搜一下就明白了。

7.2 覆盖索引的利弊

7.2.1 好处

- 避免Innodb表进行索引的二次查询（回表）
- 可以把随机IO变成顺序IO加快查询效率（还是说的回表）
- 数据在索引里面数据量少更紧凑

由于覆盖索引可以减少树的搜索次数，显著提升查询性能，所以使用覆盖索引是一个常用的性能优化手段。

7.2.2 弊端

`索引字段的维护` 总是有代价的。因此，在建立冗余索引来支持覆盖索引时就需要权衡考虑了。

8. 如何给字符串添加索引

8.1 前缀索引

MySQL是支持前缀索引的。默认地，如果你创建索引的语句不指定前缀长度，那么索引就会包含整个字符串。

索引长度越高，越能针对性寻找对应字符串；索引长度越低，需要对比的候选字符串就越多。**前缀索引，定义好长度，就可以做到既节省空间，又不用额外增加太多的查询成本。**区分度越高越好，意味着重复的键值越少。

8.2 前缀索引对覆盖索引的影响

使用前缀索引就用不上覆盖索引对查询性能的优化了，这也是你在选择是否使用前缀索引时需要考虑的一个因素。

9. 索引下推Index Condition Pushdown

Index Condition Pushdown(ICP)是MySQL 5.6中新特性，是一种在存储引擎层使用索引过滤数据的一种优化方式。

9.1 使用前后对比

- 使用前：如果出现LIKE '%aaa'，则不满足使用联合索引的条件，需要读取之后才能进行判断。
- 使用后：如果部分`WHERE`条件可以仅使用索引中的列进行筛选，则MySQL服务器会把这部分`WHERE`条件放到存储引擎筛选。然后，存储引擎通过使用索引条目来筛选数据，并且只有在满足这一条件时才从表中读取行。
  - 好处：ICP可以减少存储引擎必须访问基表的次数和MySQL服务器必须访问存储引擎的次数。
  - 但是，ICP的`加速效果`取决于在存储引擎内通过`ICP筛选`掉的数据的比例。
- 多次测试效率对比来看，使用ICP优化的查询效率会好一些。这里建议多存储一些数据效果更明显。

9.2 举例

```
CREATE INDEX zip_lastname_address ON people (zipcode, lastname, firstname);

EXPLAIN SELECT * FROM people WHERE zipcode = '000001' AND lastname LIKE '%章%'
AND address LIKE '%北京市%';
```

- 这里如果没有索引下推，则只有zipcode使用了索引。如果有索引下推，则zipcode和lastname都可以使用，减少了初步筛选出的记录，效率更高。

9.3 ICP的开启/关闭

- 默认情况下启用索引条件下推。可以通过设置系统变量`optimizer_switch`控制:`index_condition_pushdown`

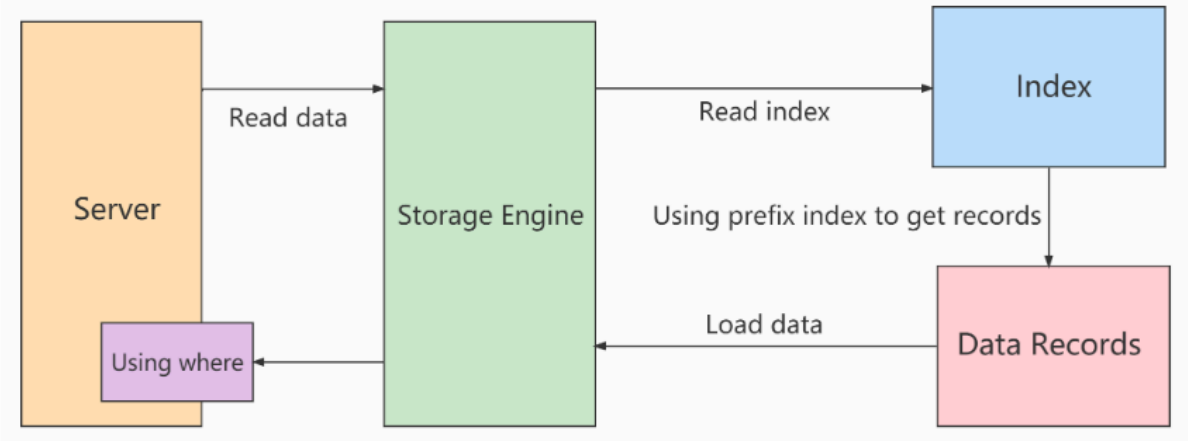
```
#打开索引下推
SET optimizer_switch = 'index_condition_pushdown=off';
#关闭索引下推
SET optimizer_switch = 'index_condition_pushdown=on';
```

- 当使用索引条件下推时，`EXPLAIN`语句输出结果中Extra列内容显示为`Using index condition`。

9.4 使用前后的扫描过程

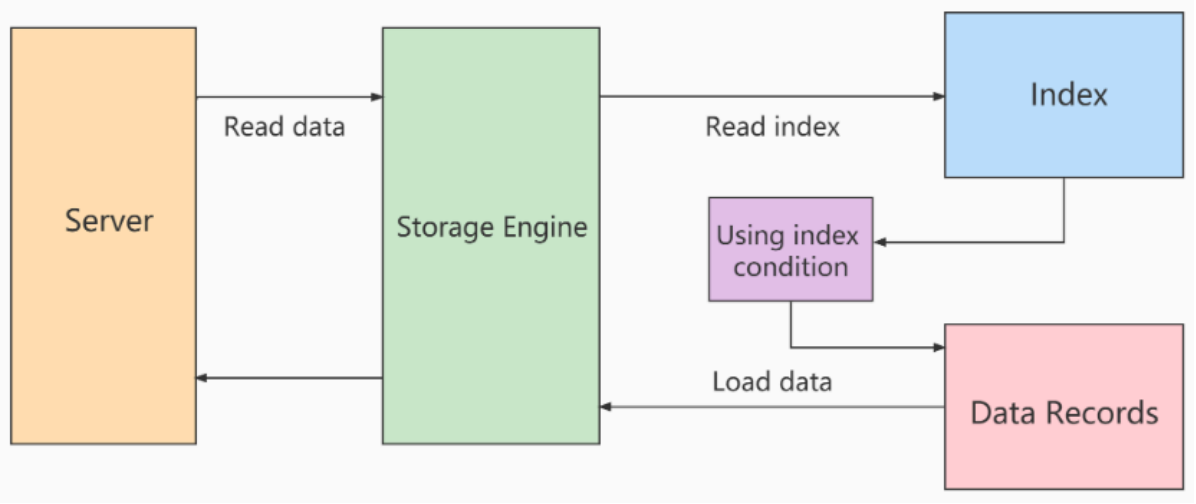
9.4.1 使用前

- storage层：只将满足index key条件的索引记录对应的整行记录取出，返回给server层
- server层：对返回的数据，使用后面的where条件过滤，直至返回最后一行。



9.4.2 使用后

- storage层：首先将index key条件满足的索引记录区间确定，然后在索引上使用index filter进行过滤。将满足的index filter条件的索引记录才去回表取出整行记录返回server层。不满足index filter条件的索引记录丢弃，不回表、也不会返回server层。
- server层：对返回的数据，使用table filter条件做最后的过滤。



9.4.3 使用前后的成本差别

使用前，存储层多返回了需要被index filter过滤掉的整行记录。使用ICP后，直接就去掉了不满足index filter条件的记录，省去了他们回表和传递到server层的成本。ICP的 加速效果 取决于在存储引擎内通过ICP筛选 掉的数据的比例。

9.5 ICP的使用条件

1. 如果表访问的类型为range、ref、eq_ref和ref_or_null可以使用ICP
2. ICP可以用于`InnoDB`和`MyISAM`表，包括分区表`InnoDB`和`MyISAM`表
3. 对于`InnoDB`表，`ICP`仅用于二级索引。ICP的目标是减少全行读取次数，从而减少I/o操作。
4. 当SQL使用覆盖索引时，不支持ICP。因为这种情况下使用ICP不会减少I/O。索引覆盖不能使用，一个原因是，索引覆盖，不需要回表。ICP作用是减小回表，ICP需要回表。
5. 相关子查询的条件不能使用ICP

10. 普通索引 vs 唯一索引

从性能的角度考虑，你选择唯一索引还是普通索引呢？选择的依据是什么呢？

10.1 查询过程

- 普通索引：没有唯一性，所以查找到第一个之后需要继续往后直到不符合查找要求。
- 唯一索引：因为只有一个，所以查找到就结束。

但这个性能差距很小很小。如果字段上每个值都不重复，普通索引查询两个，唯一索引查询一个，差距很小。

10.2 更新过程

change buffer对普通索引更新有优化。

当需要更新一个数据页时

- 如果数据页在内存中，直接更新内存中的页

- 如果不在内存中，在不影响一致性的前提下，将更新操作缓存在change buffer中，等该数据页写入内存再进行修改，该过程称为merge。merge发生的情况如下
    - 需要访问该数据页时，要将该数据页读入
    - 系统有后台线程定期处理
    - 数据库正常关闭时

优点：减少读磁盘，避免占用内存，提高内存利用率。

唯一索引的更新不能使用change buffer，实际上也只有普通索引可以使用。

10.3 change buffer的使用场景

- 考虑到更新性能，尽量选择普通索引，配合change buffer使用。
- 如果更新后立即查询，则change buffer用处不大，应该关闭，在其他情况下，change buffer都能提升更新性能。
- 如果业务可以接受，优先考虑非唯一索引，这样才能利用change buffer这一优化，如果业务不接受。
    - 首先业务正确性优先，如果保证了一定不会写入重复数据，则可以进一步考虑性能问题，否则还是需要使用唯一索引。
    - 其次，一些归档库场景可以考虑普通索引，因为这些数据一般不会更新，保证唯一性之后就可以将唯一索引改为普通索引了。

11. 其它查询优化策略

11.1 EXISTS 和 IN 的区分

不太理解哪种情况下应该使用 EXISTS，哪种情况应该用 IN。选择的标准是看能否使用表的索引吗?

回答:

索引是个前提，其实选择与否还是要看表的大小。exists和in都是循环嵌套。你可以将选择的标准理解为`小表驱动大表`。在这种方式下效率是最高的。

```
A in B # A在循环内层，B在循环外层(驱动)
A exists B # A在循环外层(驱动)，B在循环内层
```

哪个表小就用哪个表来驱动，A表小就用EXISTS，B表小就用IN。

11.2 COUNT(*)与COUNT(具体字段)效率

在MySQL中统计数据表的行数，可以使用三种方式: `SELECT COUNT(*)`、`SELECT COUNT(1)` 和 `SELECT COUNT(具体字段)`，使用这三者之间的查询效率是怎样的?

回答:

- `COUNT(*)` 和 `COUNT(1)` 都是对所有结果进行 `COUNT`，效率差不多。
- 如果是MyISAM存储引擎，统计数据表的行数只需要 `o(1)` 的复杂度，这是因为每张 MyISAM的数据表都有一个meta 信息存储了 `row_count` 值，而一致性则由表级锁来保证。

- 如果是InnoDB存储引擎，因为InnoDB支持事务，采用行级锁和MVCC机制，没有row_count变量，因此需要采用`扫描全表`，是o(n) 复杂度，进行循环 + 计数的方式来完成统计。
- 在InnoDB引擎中，如果采用`COUNT(具体字段)`来统计数据行数，要尽量采用二级索引。因为主键采用的索引是聚簇索引，聚簇索引包含的信息多，明显会大于二级索引(非聚簇索引)。对于`COUNT(*)`和`COUNT(1)`来说，它们不需要查找具体的行，只是统计行数，系统会`自动`采用占用空间更小的二级索引来进行统计。当没有二级索引的时候，才会采用主键索引来进行统计。

11.3 关于SELECT(*)

尽量不要用

- MySQL 在解析的过程中，会通过`查询数据字典`将"*"按序转换成所有列名，这会大大的耗费资源和时间。
- 无法使用`覆盖索引`

11.4 LIMIT 1 对优化的影响

针对的是会扫描全表的 SQL 语句，并且可以确定结果集只有一条。

- 如果是非唯一索引，加上这一条就不会作出多于扫描。
- 如果是唯一索引，不需要加这一条，就不会作出扫描。

11.5 多使用COMMIT

只要有可能，在程序中尽量多使用 COMMIT，这样程序的性能得到提高，需求也会因为 COMMIT 所释放的资源而减少。

COMMIT 所释放的资源：

- 回滚段上用于恢复数据的信息
- 被程序语句获得的锁
- redo / undo log buffer 中的空间
- 管理上述 3 种资源中的内部花费

12. 淘宝数据库，主键如何设计的?

12.1 自增ID的问题

除了ID简单，其他都是缺点。

- **可靠性不高**：存在自增ID回溯的问题，这个问题直到最新版本的MySQL 8.0才修复。
- **安全性不高**：对外暴露的接口可以非常容易猜测对应的信息。
- **性能差**：自增ID的性能较差，需要在数据库服务器端生成。
- **交互多**：业务还需要额外执行一次类似 last_insert_id() 的函数才能知道刚才插入的自增值，这需要多一次的网络交互。在海量并发的系统中，多1条SQL，就多一次性能上的开销。

- **局部唯一性**：最重要的一点，自增ID是局部唯一，只在当前数据库实例中唯一，而不是全局唯一，在任意服务器间都是唯一的。对分布式来说不现实。

12.2 业务字段做主键

为了能够唯一地标识一个会员的信息，需要为 `会员信息表` 设置一个主键。那么，怎么为这个表设置主键，才能达到我们理想的目标呢？ 这里我们考虑业务字段做主键。

cardno（卡号）	membername（名称）	memberphone（电话）	memberpid（身份证号）	address（地址）	sex（性别）	birthday（生日）
10000001	张三	13812345678	110123200001017890	北京	男	2000-01-01
10000002	李四	13512312312	123123199001012356	上海	女	1990-01-01

选择哪个字段合适呢?

- **选择卡号（cardno）**：会员卡号（cardno）看起来比较合适，因为会员卡号不能为空，而且有唯一性，可以用来标识一条会员记录。
- 但实际情况是，`会员卡号可能存在重复使用` 的情况。比如一个人退会员，另一个人进会员，可能会把退会员的身份给进会员的用。
  - 从信息系统层面上看没问题，可以将有关前会员的信息换成后一个会员的信息。
  - 从系统的业务层面看有问题，如果将前一个会员的消费信息中的个人信息改为后一个会员的，造成错误。
- 因此不能把会员卡号做主键。
- **选择会员电话 或 身份证号**：电话可能会被运营商收回再发给别人使用，不合适。身份证号属于隐私信息，不合适。

所以，建议尽量不要用跟业务有关的字段做主键。毕竟，作为项目设计的技术人员，我们谁也无法预测在项目的整个生命周期中，哪个业务字段会因为项目的业务需求而有重复，或者重用之类的情况出现。

> 经验：
>
> 刚开始使用 MySQL 时，很多人都很容易犯的错误是喜欢用业务字段做主键，想当然地认为了解业务需求，但实际情况往往出乎意料，而更改主键设置的成本非常高。

12.3 淘宝的主键设计

`订单表的主键` 淘宝是如何设计的呢？是自增ID吗?

我们详细看下4个订单号：

```
1550672064762308113
1481195847180308113
1431156171142308113
1431146631521308113
```

订单号是19位的长度，且订单的最后5位都是一样的，都是08113。且订单号的前面14位部分是单调递增的。大胆猜测，淘宝的订单ID设计应该是：

```
订单ID = 时间 + 去重字段 + 用户ID后6位尾号
```

这样的设计能做到全局唯一，且对分布式系统查询及其友好。

12.4 推荐的主键设计

`非核心业务`：对应表的主键自增ID，如告警、日志、监控等信息。

`核心业务`：主键设计至少应该是全局唯一且是单调递增。全局唯一保证在各系统之间都是唯一的，单调递增是希望插入时不影响数据库性能。

这里推荐最简单的一种主键设计：UUID。

UUID的特点：

全局唯一，占用36字节，数据无序，插入性能差。

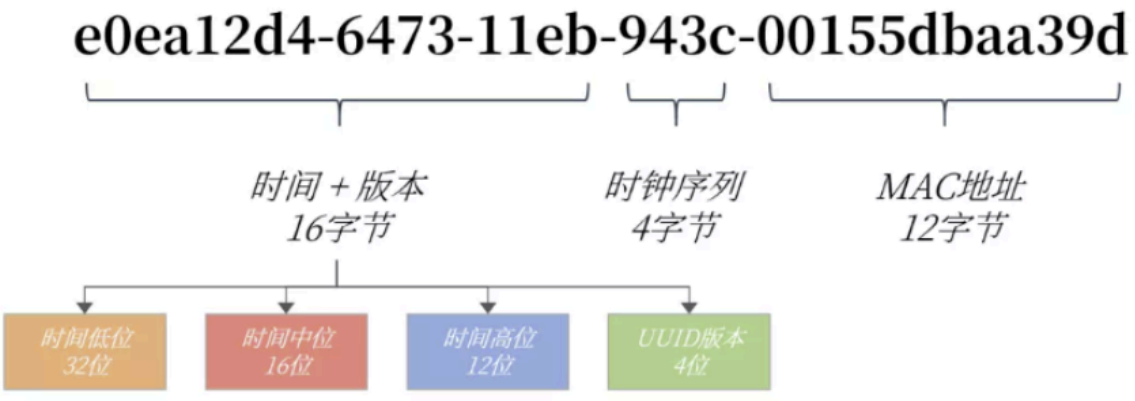
认识UUID：

- 为什么UUID是全局唯一的?
- 为什么UUID占用36个字节?
- 为什么UUID是无序的?

MySQL数据库的UUID组成如下所示：

```
UUID = 时间+UUID版本（16字节）- 时钟序列（4字节） - MAC地址（12字节）
```

我们以UUID值e0ea12d4-6473-11eb-943c-00155dbaa39d举例：



- 为什么UUID是全局唯一的?
  - 在UUID中时间部分占用60位，存储的类似TIMESTAMP的时间戳，但表示的是从1582-10-15 00：00：00.00到现在的100ns的计数。可以看到UUID存储的时间精度比TIMESTAMPE更高，时间维度发生重复的概率降低到1/100ns。
  - 时钟序列是为了避免时钟被回拨导致产生时间重复的可能性。MAC地址用于全局唯一。
- 为什么UUID占用36个字节?
  - UUID根据字符串进行存储，设计时还带有无用"-"字符串，因此总共需要36个字节。
- 为什么UUID是随机无序的呢?
  - 因为UUID的设计中，将时间低位放在最前面，而这部分的数据是一直在变化的，并且是无序。

- 改造UUID：若将时间高低位互换，则时间就是单调递增的了，也就变得单调递增了。MySQL 8.0可以更换时间低位和时间高位的存储方式，这样UUID就是有序的UUID了。MySQL 8.0还解决了UUID存在的空间占用的问题，除去了UUID字符串中无意义的"-"字符串，并且将字符串用二进制类型保存，这样存储空间降低为了16字节。

可以通过MySQL8.0提供的uuid_to_bin函数实现上述功能，同样的，MySQL也提供了bin_to_uuid函数进行转化：

```
SET @uuid = UUID();
SELECT @uuid,uuid_to_bin(@uuid),uuid_to_bin(@uuid,TRUE);
# uuid_to_bin(@uuid) 转成16进制存储
# uuid_to_bin(@uuid,TRUE); 修改成先高位 中位 地位，就可以保证uuid递增了
```

```
mysql> SET @uuid = UUID();
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @uuid,uuid_to_bin(@uuid),uuid_to_bin(@uuid,TRUE);
+--------------------------------------+------------------------------------+-------------------------------------------+
| @uuid                                | uuid_to_bin(@uuid)                 | uuid_to_bin(@uuid,TRUE)                   |
+--------------------------------------+------------------------------------+-------------------------------------------+
| 71c8dc8a-6533-11ec-a652-000c2923a5e8 | 0x71C8DC8A653311ECA652000C2923A5E8 | 0x11EC653371C8DC8AA652000C2923A5E8        |
+--------------------------------------+------------------------------------+-------------------------------------------+
1 row in set (0.00 sec)
```

有序UUID性能测试

	时间（秒）	表大小（G）
自增ID	2712	240
UUID	3396	250
有序UUID	2624	243

从上图可以看到插入1亿条数据有序UUID是最快的，而且在实际业务使用中有序UUID在 业务端就可以生成 。还可以进一步减少SQL的交互次数。

另外，虽然有序UUID相比自增ID多了8个字节，但实际只增大了3G的存储空间，还可以接受。

> 在当今的互联网环境中，非常不推荐自增ID作为主键的数据库设计。更推荐类似有序UUID的全局唯一的实现。
>
> 另外在真实的业务系统中，主键还可以加入业务和系统属性，如用户的尾号，机房的信息等。这样的主键设计就更为考验架构师的水平了。

如果不是MySQL8.0 肿么办?

手动赋值字段做主键！（分布式中的每一处有一段和别人不一样的内容做字段的固定部分）

比如，设计各个分店的会员表的主键，因为如果每台机器各自产生的数据需要合并，就可能会出现主键重复的问题。

可以在总部 MySQL 数据库中，有一个管理信息表，在这个表中添加一个字段，专门用来记录当前会员编号的最大值。

门店在添加会员的时候，先到总部 MySQL 数据库中获取这个最大值，在这个基础上加 1，然后用这个值作为新会员的“id”，同时，更新总部 MySQL 数据库管理信息表中的当 前会员编号的最大值。

这样一来，各个门店添加会员的时候，都对同一个总部 MySQL 数据库中的数据表字段进 行操作，就解决了各门店添加会员时会员编号冲突的问题。