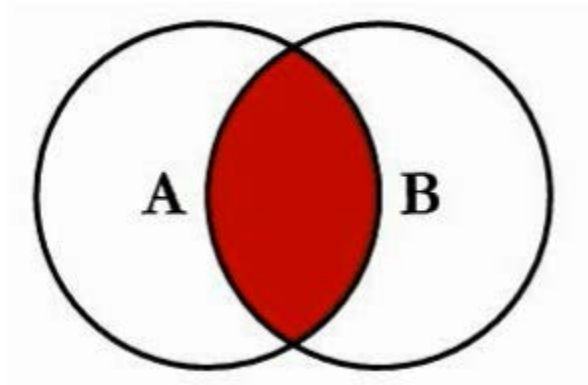


Join

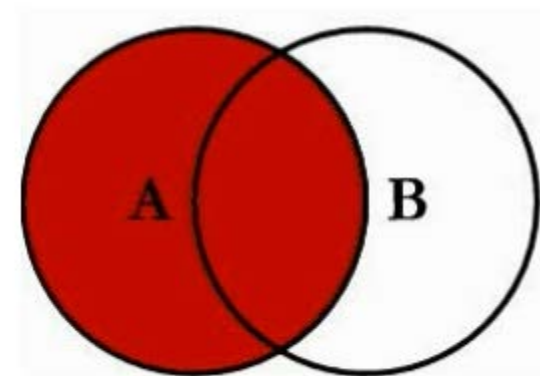
A. inner join

A、B 共有，也就是交集。



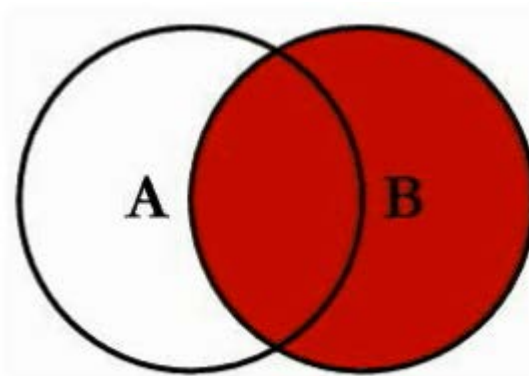
B. left join

A 独有+AB 共有（交集）

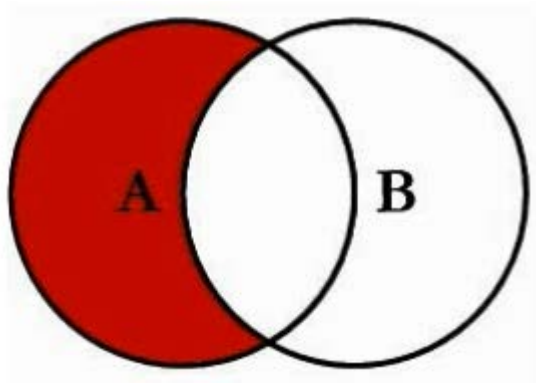


C. right join

B 独有+AB 共有（交集）

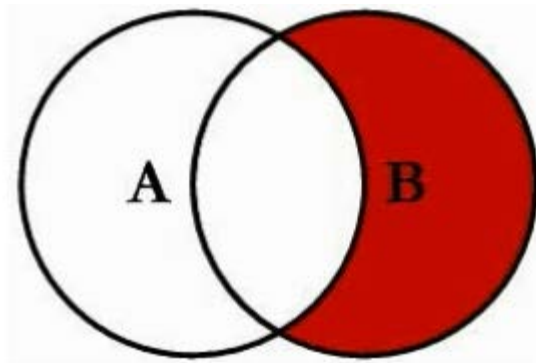


D. A 独有



注：参照 left join, **A 独有**只是将 AB 交集部分去掉。

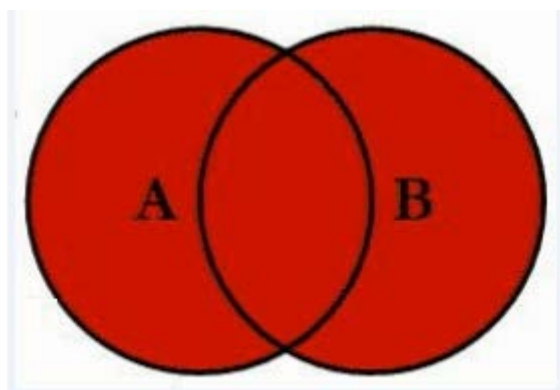
E. B 独有



注：参照 right join, **B 独有**只是将 AB 交集部分去掉。

F. AB 全有（并集）

由于 mysql 中不支持 full outer join, 所以这里通过 union 进行转换。AB 并集: **AB 交集** + **A 独有** + **B 独有**



```
mysql> select *from tb_emp a left join tb_dept b on a.deptid=b.id
-> union
-> select *from tb_emp a right join tb_dept b on a.deptid=b.id;
```

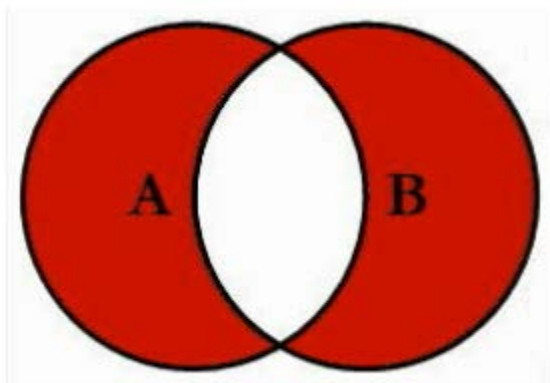
id	name	deptid	id	deptname
1	jack	1	1	研发
2	tom	1	1	研发
3	tonny	1	1	研发
4	mary	2	2	测试
5	rose	2	2	测试
6	luffy	3	3	运维
7	outman	14	NULL	NULL
NULL	NULL	NULL	4	经理

8 rows in set (0.01 sec)

AB共有

A、B独有

G. A、B 独有并集



A、B 独有并集，相当于 A、B 全有去掉 AB 的共有（交集）。

```
mysql>
mysql> select *from tb_emp a left join tb_dept b on a.deptid=b.id where b.id is null
-> union
-> select *from tb_emp a right join tb_dept b on a.deptid=b.id where a.deptid is null;
```

id	name	deptid	id	deptname
7	outman	14	NULL	NULL
NULL	NULL	NULL	4	经理

2 rows in set (0.00 sec)

A、B独有并集

总结：这里主要对 MySQL 中 join 语句的 7 中用法进行了总结，主要注意 MySQL 不支持 full outer join，所以需要对其进行转换变形，最终达到效果。

索引

(一) 索引优缺点

优点

- 1、类似大学图书馆的书目索引，提高数据的检索效率，降低数据库的 IO 成本。
- 2、通过索引列对数据进行排序，降低数据的排序成本，从而降低 CPU 的消耗。

缺点

- 1、索引实际上也是一张表，该表保存了主键与索引字段，并指向实体表的记录，所以索引列也要占用空间。
- 2、虽然索引大大提高了查询效率，但是降低了更新表的速度，如 insert、update 和 delete 操作。因为更新表时，MySQL 不仅要保存数据，还要保存索引文件每次更新的索引列字段，并且在更新操作后，会更新相应字段索引的信息。
- 3、索引只是提高查询效率的一个因素，如果你的 MySQL 有大量的数据表，就需要花时间研究建立最优秀的索引或优化查询语句。

(二) 索引分类

- 1、单值索引：一个索引只包含单个列，一个表可以有多个单值索引。
- 2、唯一索引：索引列的值必须唯一，但允许有空值，主键就是唯一索引。
- 3、复合索引：一个索引包含多个列。

(三) 索引结构

- 1、BTree 索引；
- 2、Hash 索引；
- 3、Full-Text 索引；
- 4、R-Tree 索引。

(四) 基本语法

1、创建索引

```
create [unique] index indexname on tablename(columnname(length));  
alter table tablename add index indexname (columnname(length));
```

注：如果是 char、varchar 类型的字段，length 可以小于字段实际长度；如果是 blob、text 类型，必须指定 length。

2、删除索引

```
drop index indexname on tablename;
```

3、查看索引

```
show index from tablename;
```

4、其他创建索引的方式

a、添加主键索引

```
ALTER TABLE `table_name` ADD PRIMARY KEY (`column`)
```

b、添加唯一索引

```
ALTER TABLE `table_name` ADD UNIQUE (`column`)
```

c、添加全文索引

```
ALTER TABLE `table_name` ADD FULLTEXT (`column`)
```

d、添加普通索引

```
ALTER TABLE `table_name` ADD INDEX index_name (`column`)
```

e、添加组合索引

```
ALTER TABLE `table_name` ADD INDEX index_name (`column1`, `column2`,  
`column3`)
```

(五) 建立索引的一般情景

1、需建立索引的情况

- a、主键自动建立唯一索引；
- b、频繁作为查询条件的字段；
- c、查询中与其他表关联的字段，外键关系建立索引；
- d、高并发下趋向创建组合索引；
- e、查询中排序的字段，排序字段若通过索引去访问将大大提高排序速度；
- f、查询中统计或分组字段；

2、不需要创建索引的情况

- a、表记录太少。（数据量太少 MySQL 自己就可以搞定了）；
- b、经常增删改的表；
- c、数据重复且平均分配的字段，如国籍、性别，不适合创建索引；
- d、频繁更新的字段不适合建立索引；
- e、Where 条件里用不到的字段不创建索引；

执行计划（Explain）

explain（执行计划），使用explain关键字可以模拟优化器执行sql查询语句，从而知道MySQL是如何处理sql语句。explain主要用于分析查询语句或表结构的性能瓶颈。

(一) Explain 的作用

通过 explain+sql 语句可以知道如下内容：

- 1、表的读取顺序。（对应 id）
- 2、数据读取操作的操作类型。（对应 select_type）
- 3、哪些索引可以使用。（对应 possible_keys）
- 4、哪些索引被实际使用。（对应 key）
- 5、表直接的引用。（对应 ref）
- 6、每张表有多少行被优化器查询。（对应 rows）

(二) Explain 包含的信息

explain 使用：explain+sql 语句，通过执行 explain 可以获得 sql 语句执行的相关信息。

```
mysql> explain select * from tb_emp;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tb_emp	NULL	ALL	NULL	NULL	NULL	NULL	2	100.00	NULL

1、id

select 查询的序列号，包含一组数字，表示查询中执行 select 子句或操作表的顺序，该字段通常与 table 字段搭配来分析。

a、id 相同，执行顺序从上到下

```
mysql> explain select t2.*
-> from t1, t2, t3
-> where t1.id = t2.id and t1.id = t3.id
-> and t1.other_column = '';
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t1	ref	PRIMARY, idx_t1	idx_t1	92	const	1	Using where
1	SIMPLE	t3	eq_ref	PRIMARY	PRIMARY	4	test.t1.ID	1	Using index
1	SIMPLE	t2	eq_ref	PRIMARY	PRIMARY	4	test.t1.ID	1	

3 rows in set (0.00 sec)

id 相同，执行顺序从上到下，搭配 table 列进行观察可知，执行顺序为 t1->t3->t2。

b、id 不同，如果是子查询，id 的序号会递增，id 值越大执行优先级越高

```
mysql> explain SELECT t2.*
-> FROM t2
-> WHERE id = (SELECT id
-> FROM t1
-> WHERE id = (SELECT t3.id
-> FROM t3
-> WHERE t3.other_column = ''));
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	t2	const	PRIMARY	PRIMARY	4	const	1	
2	SUBQUERY	t1	const	PRIMARY	PRIMARY	4		1	Using index
3	SUBQUERY	t3	ALL	NULL	NULL	NULL	NULL	1	Using where

3 rows in set (0.00 sec)

如果是子查询 id 的序号会递增，id 值越大执行优先级越高，搭配 table 列可知，执行顺序为 t3->t1->t2。

c、id 相同和不同的情况同时存在

```
mysql> explain select t2.* from (
-> select t3.id
-> from t3
-> where t3.other_column = '') s1, t2
-> where s1.id = t2.id;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	system	NULL	NULL	NULL	NULL	1	
1	PRIMARY	t2	const	PRIMARY	PRIMARY	4	const	1	
2	DERIVED	t3	ALL	NULL	NULL	NULL	NULL	1	Using where

3 rows in set (0.00 sec)

id 如果相同，可认为是同一组，执行顺序从上到下。在所有组中，id 值越大执行优先级越高。所以执行顺序为 t3->derived2(衍生表，也可以说临时表)->t2。

总结：id 的值表示 select 子句或表的执行顺序，id 相同，执行顺序从上到下，id 不同，值越大的执行优先级越高。

2、select_type

查询的类型，主要用于区别普通查询、联合查询、子查询等复杂的查询。其值主要有六个：

(1)、SIMPLE

简单的 select 查询，查询中不包含子查询或 union 查询。

(2)、PRIMARY

查询中若包含任何复杂的子部分，最外层查询为 PRIMARY，也就是最后加载的就是 PRIMARY。

(3)、SUBQUERY

在 select 或 where 列表中包含了子查询，就被标记为 SUBQUERY。

(4)、DERIVED

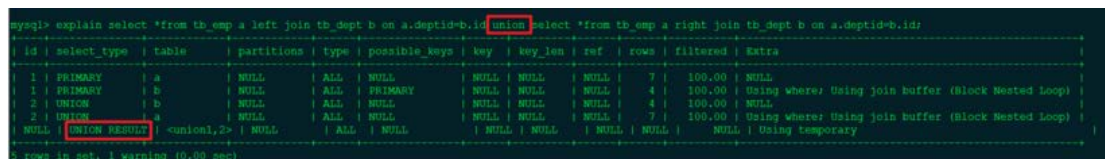
在 from 列表中包含的子查询会被标记为 DERIVED(衍生)，MySQL 会递归执行这些子查询，将结果放在临时表中。

(5)、UNION

若第二个 select 出现在 union 后，则被标记为 UNION，若 union 包含在 from 子句的子查询中，外层 select 将被标记为 DERIVED。

(6)、UNION RESULT

从 union 表获取结果的 select。



```
mysql> explain select *from tb_emp a left join tb_dept b on a.deptid=b.id union select *from tb_emp a right join tb_dept b on a.deptid=b.id;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	a		ALL					7	100.00	Using where; Using join buffer (Block Nested Loop)
1	PRIMARY	b		ALL	PRIMARY				4	100.00	Using where; Using join buffer (Block Nested Loop)
2	UNION	b		ALL					4	100.00	Using where; Using join buffer (Block Nested Loop)
2	UNION	a		ALL					7	100.00	Using where; Using join buffer (Block Nested Loop)
NULL	UNION RESULT	<union1,2>		ALL					NULL	NULL	Using temporary

rows in set: 1 warning: 10.00 sec

3、table

显示 sql 操作属于哪张表的。

4、partitions

官方定义为 The matching partitions (匹配的分区)，该字段应该是看 table 所在的分区吧（不晓得理解错误没）。值为 NULL 表示表未被分区。

5、type

表示查询所使用的访问类型，type 的值主要有八种，该值表示查询的 sql 语句好坏，

从最好到最差依次为：system>const>eq_ref>ref>range>index>ALL。

要详细了解 type 取值的作用，需要用数据说话。创建 tb_emp (员工表) 和 tb_dept (部门表)。

a、tb_emp 表

```
DROP TABLE IF EXISTS `tb_emp`;  
CREATE TABLE `tb_emp` (  
  `id` int(11) NOT NULL AUTO_INCREMENT,  
  `name` varchar(20) NOT NULL,  
  `deptid` int(11) NOT NULL,  
  PRIMARY KEY (`id`),
```



```

KEY `idx_tb_emp_name` (`name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO `tb_emp` (name,deptid) VALUES ('jack', '1');
INSERT INTO `tb_emp` (name,deptid) VALUES ('tom', '1');
INSERT INTO `tb_emp` (name,deptid) VALUES ('tonny', '1');
INSERT INTO `tb_emp` (name,deptid) VALUES ('mary', '2');
INSERT INTO `tb_emp` (name,deptid) VALUES ('rose', '2');
INSERT INTO `tb_emp` (name,deptid) VALUES ('luffy', '3');
INSERT INTO `tb_emp` (name,deptid) VALUES ('outman', '4');

```

b、tb_dept 表

```

DROP TABLE IF EXISTS `tb_dept`;
CREATE TABLE `tb_dept` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `deptname` varchar(20) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO `tb_dept` (deptname) VALUES ('研发');
INSERT INTO `tb_dept` (deptname) VALUES ('测试');
INSERT INTO `tb_dept` (deptname) VALUES ('运维');
INSERT INTO `tb_dept` (deptname) VALUES ('经理');

```

(1)、system

表只有一行记录（等于系统表），是 const 的特例类型，平时不会出现，可以忽略不计。

但是笔者发现在 MySQL5.7.22 时，不会出现该字段值，只能出现 const，但是在 MySQL5.7 版本以下可以出现该情况。猜测 MySQL5.7 版本是不是进行了优化，因为 system 官网的解释：

• system

The table has only one row (= system table). This is a special case of the const join type.

5.5.48:

查询创建工具 查询编辑器

```
1 explain select *from (select *from tb_emp where id=1) emp;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	system	(Null)	(Null)	(Null)	(Null)	1	
2	DERIVED	tb_emp	const	PRIMARY	PRIMARY	4		1	

5.7.19:


```
20 EXPLAIN SELECT * FROM (SELECT * FROM tb_emp where id=1) emp;
```

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

注：两个引擎的执行信息不一样，5.5.48 执行过程中产生了临时表(DERIVED)，5.7.19 为简单查询。

(2)、const

表示通过一次索引就找到了结果，常出现于 primary key 或 unique 索引。因为只匹配一行数据，所以查询非常快。如将主键置于 where 条件中，MySQL 就能将查询转换为一个常量。

```
21 EXPLAIN SELECT * FROM tb_emp WHERE id = 1;
```

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

(3)、eq_ref

唯一索引扫描，对于每个索引键，表中只有一条记录与之匹配。常见主键或唯一索引扫描。

```
22 EXPLAIN SELECT * FROM tb_emp, tb_dept WHERE tb_dept.id=tb_emp.deptid and tb_emp.name='outman';
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tb_emp	(Null)	ref	idx_tb_emp_name	idx_tb_emp_name	62	const	1	100	(Null)
1	SIMPLE	tb_dept	(Null)	eq_ref	PRIMARY	PRIMARY	4	tb_emp.deptid	1	100	(Null)

(4)、ref

非唯一性索引扫描，返回匹配某个单独值的所有行。本质上也是一种索引访问，返回匹配某值(某条件)的多行值，属于查找和扫描的混合体。由于是非唯一性索引扫描，所以对 tb_emp 表的 deptid 字段创建索引：

```
create index idx_tb_emp_deptid on tb_emp(deptid);
```

```
24 EXPLAIN SELECT * FROM tb_emp WHERE deptid=2;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tb_emp	(Null)	ref	idx_tb_emp_deptid	idx_tb_emp_deptid	4	const	2	100	(Null)

(5)、range

只检索给定范围的行，使用一个索引来检索行，可以在 key 列中查看使用的索引，一般出现在 where 语句的条件中，如使用 between、>、<、in 等查询。

这种索引的范围扫描比全表扫描要好，因为索引的开始点和结束点都固定，不用扫描全索引。

```
25 EXPLAIN SELECT * FROM tb_emp WHERE deptid>2;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tb_emp	(Null)	range	idx_tb_emp_deptid	idx_tb_emp_deptid	4	(Null)	2	100	Using index condition

虽然我们为 deptid 字段创建了索引并在 where 中使用了 between 等，但在如下情况 type 仍为 **ALL**。

26 EXPLAIN SELECT * FROM tb_emp WHERE deptid BETWEEN 1 AND 3;

信息	结果1	概况	状态								
id	select_type	table	partitio	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tb_emp	(Null)	ALL	idx_tb_emp_deptid	(Null)	(Null)	(Null)	7	85.71	Using where

27 EXPLAIN SELECT * FROM tb_emp WHERE id BETWEEN 1 AND 3;

信息	结果1	概况	状态								
id	select_type	table	partitio	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tb_emp	(Null)	range	PRIMARY	PRIMARY	4	(Null)	3	100	Using where

对比两图，可以看到使用 deptid 和 id 进行操作，其 type 的值一个是 ALL 也就是进行了全表扫描，一个是 range 进行了指定索引范围值检索。可能原因 deptid 并不是唯一索引。

对于以上问题，**需要具体问题具体分析**，并不能一概而论。

(6)、index

全索引扫描，index 和 ALL 的区别：index 只遍历**索引树**，通常比 ALL 快，因为索引文件通常比数据文件小。虽说 index 和 ALL 都是全表扫描，但是 index 是从**索引**中读取，ALL 是从**磁盘**中读取。

28 EXPLAIN SELECT deptid FROM tb_emp;																							
信息		结果1		概况		状态																	
id		select_type		table		partitio		type		possible_keys		key		key_len		ref		rows		filtered		Extra	
1		SIMPLE		tb_emp		(Null)		index		(Null)		idx_tb_emp_deptid		4		(Null)		7		100		Using index	

(7)、all

全表扫描

注：一般来说，需保证查询至少达到 **range** 级别，最好能达到 **ref**。

6、possible_keys 和 key、key_len

possible_keys: 显示**可能**应用在表中的索引，**可能一个或多个**。查询涉及到的字段若存在索引，则该索引将被列出，**但不一定被查询实际使用**。

key: **实际中使用的索引**，如为 **NULL**，则表示未使用索引。若查询中使用了**覆盖索引**，则该索引和查询的 select 字段重叠。

key_len: 表示索引中所使用的**字节数**，可通过该列计算查询中使用的索引长度。在不损失精确性的情况下，**长度越短越好**。key_len 显示的值为索引字段的最大可能长度，并非实际使用长度，即 key_len 是根据表定义计算而得，并不是通过表内检索出的。

简单理解：possible_keys 表示**理论上**可能用到的索引，key 表示**实际中**使用的索引。

```
mysql> explain select id,deptid from tb_emp;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_emp | NULL | index | NULL | idx_deptid | 4 | NULL | 7 | 100.00 | Using index |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

possible_keys 为 NULL 表示可能未用到索引，但 key=idx_deptid 表示在实际查询的过程中进行了索引的全扫描。

通过下面的例子来理解 key_len，首先为 name 字段创建索引：

```
create index idx_name on tb_emp(name);
```

```
mysql> explain select * from tb_emp where deptid =2;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_emp | NULL | ref | idx_deptid | idx_deptid | 4 | const | 2 | 100.00 | NULL |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)

mysql> explain select * from tb_emp where deptid =2 and name='rose';
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_emp | NULL | ref | idx_deptid,idx_name | idx_name | 62 | const | 1 | 28.57 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

注：在使用索引查询时，当条件越精确，key_len 的长度可能会越长，所以在不影响结果的情况下，key_len 的值越短越好。

7、 ref

显示关联的字段。如果使用常数等值查询，则显示 const，如果是连接查询，则会显示关联的字段。

```
mysql> explain select * from tb_emp,tb_dept where tb_emp.deptid=tb_dept.id and tb_emp.name='rose';
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | tb_emp | NULL | ref | idx_deptid,idx_name | idx_name | 62 | const | 1 | 100.00 | NULL |
| 2 | SIMPLE | tb_dept | NULL | eq_ref | PRIMARY | PRIMARY | 4 | db01.tb_emp.deptid | 1 | 100.00 | NULL |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set, 1 warning (0.00 sec)
```

注：由于 id 相同，因此从上到下执行：

#1. tb_emp 表为非唯一性索引扫描，实际使用的索引列为 idx_name，由于 tb_emp.name='rose' 为一个常量，所以 ref=const。

#2. tb_dept 为唯一索引扫描，从 sql 语句可以看出，实际使用了 PRIMARY 主键索引，ref=db01.tb_emp.deptid 表示关联了 db01 数据库中 tb_emp 表的 deptid 字段。

8、 rows

根据表统计信息及索引选用情况大致估算出找到所需记录所要读取的行数。当然该值越小越好。

9、 filtered

百分比值，表示存储引擎返回的数据经过过滤后，剩下多少满足查询条件记录数量的比例。

10、 extra

显示十分重要的额外信息。其取值有以下几个：

(1)、 Using filesort

Using filesort 表明 mysql 会对数据使用一个外部的索引排序，而不是按照

表内的索引顺序进行读取。

mysql 中无法利用索引完成的排序操作称为“文件排序”。

出现 Using filesort 就**非常危险**了,在数据量非常大的时候几乎“**九死一生**”。

出现 Using filesort 尽快优化 sql 语句。

deptname 字段未建索引的情况。

```
mysql> explain select *from tb_dept order by deptname;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE      | tb_dept | NULL       | ALL | NULL          | NULL | NULL    | NULL | 4 | 100.00 | Using filesort |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

为 deptname 字段创建索引后。

(2)、Using temporary

使用了临时表保存中间结果,常见于排序 order by 和分组查询 group by。非常危险,“**十死无生**”,急需优化。

将 tb_emp 中 name 的索引先删除,出现如下图结果,非常烂,Using filesort 和 Using temporary,“**十死无生**”。

```
mysql> explain select *from tb_emp group by name;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE      | tb_emp | NULL       | ALL | NULL          | NULL | NULL    | NULL | 7 | 100.00 | Using temporary; Using filesort |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

为 name 字段创建索引后。

```
mysql> create index idx_name on tb_emp(name);
Query OK, 0 rows affected (0.03 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> explain select *from tb_emp group by name;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE      | tb_emp | NULL       | index | idx_name      | idx_name | 62      | NULL | 7 | 100.00 | NULL |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.00 sec)
```

(3)、Using index

表明相应的 select 操作中使用了覆盖索引,避免访问表的额外数据行,效率不错。

如果同时出现了 Using where,表明索引被用来执行索引键值的查找。(where deptid=1)

如果没有同时出现 Using where,表明索引用来读取数据而非执行查找动作。

删除 tb_emp 表中 name 和 deptid 字段的单独索引,创建复合索引。

```
mysql> create index idx_name_deptid on tb_emp(name,deptid);
Query OK, 0 rows affected (0.03 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> explain select name from tb_emp where deptid=1;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE      | tb_emp | NULL       | index | NULL          | idx_name_deptid | 66      | NULL | 7 | 14.29 | Using where; Using index |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set, 1 warning (0.01 sec)
```

从这里给出覆盖索引的定义: select 的数据列只从索引中就能取得数据,不必读取数据行。通过上面的例子理解:创建了 (name, deptid) 的复合索引,查询的时候也使用复合索引或部分,这就形成了覆盖索引。简记: 查询使用复合索引,并且查询的列

就是索引列，不能多，个数需对应。

使用优先级 `Using index>Using filesort (九死一生)>Using temporary (十死无生)`。也就说出现后面两项表明 sql 语句是非常烂的，急需优化!!!

(三) 总结

explain (执行计划) 包含的信息十分的丰富，着重关注以下几个字段信息。

- 1、 id, select 子句或表执行顺序, id 相同, 从上到下执行, id 不同, id 值越大, 执行优先级越高。
- 2、 type, type 主要取值及其表示 sql 的好坏程度 (由好到差排序):
`system > const > eq_ref > ref > range > index > ALL`。
保证 range, 最好到 ref。
- 3、 key, 实际被使用的索引列。
- 4、 ref, 关联的字段, 常量等值查询, 显示为 `const`, 如果为连接查询, 显示关联的字段。
- 5、 Extra, 额外信息, 使用优先级 `Using index>Using filesort (九死一生)>Using temporary (十死无生)`。

着重关注上述五个字段信息, 对日常生产过程中调优十分有用。