本文档截取自电子工业出版社出版的图书《千金良方：MySQL 性能优化金字塔法则》的附录部分，该图书由来自沃趣科技的李春、罗小波、董红禹三位作者共同撰写。更多精彩内容可关注下方的微信公众号 "**沃趣技术**"与"**知数堂**"


附录 D explain 执行计划详解

1、explain 语法解析

- 要使用 explain 语句来查看执行计划，我们需要先知道 explain 语句的语法，如下：

```
mysql > help explain ;
Name: 'EXPLAIN'
Description:
Syntax:
{EXPLAIN | DESCRIBE | DESC}
    tbl_name [col_name | wild]

{EXPLAIN | DESCRIBE | DESC}
    [explain_type]
    explainable_stmt

explain_type: {
```

```
    EXTENDED
  | PARTITIONS
  | FORMAT = format_name
}

format_name: {
    TRADITIONAL
  | JSON
}

explainable_stmt: {
    SELECT statement
  | DELETE statement
  | INSERT statement
  | REPLACE statement
  | UPDATE statement
}
```

- 从上面的语法中可以看到执行计划三个关键字等值，使用三个任意一个即可（{EXPLAIN | DESCRIBE | DESC}），通常来讲查看执行计划都是使用 explain
    - 另外，三个关键字后面可以跟上某表、某表某字段，表示查看表、字段的定义信息，通常来讲查看表定义属性都是使用 desc、describe，或者使用 show create 语句来查看完整的建表语句
- explain_type 有三个类型，EXTENDED、PARTITIONS、FORMAT = format_name，三个只能同时使用一个，否则报语法错误，分别表示含义如下（注意：5.7 开始默认启用 EXTENDED 和 PARTITIONS，即只需要选择用与不用 FORMAT =json 即可，默认 FORMAT =TRADITIONAL。如果要显式使用 EXTENDED 和 PARTITIONS 关键字，那么与 5.6 及其之前的版本一样，仍然只能选其一）
    - EXTENDED：表示查看扩展的执行计划信息，会把执行计划内部改写的 SQL 语句打印出来，放到 warnings 信息中，另外，

EXTENDED 类型还会多输出一个列 filtered（注意：5.7 版本开始默认启用了扩展的格式，不需要再显式使用这个关键字了）

- PARTITIONS：表示查看分区表的执行计划信息，执行计划中可以看到 mysql 扫描了哪些分区，会多出一个 partitions 输出列（注意：5.7 版本开始默认使启用了 PARTITIONS 关键字的功能，不需要再显式使用这个关键字了）
- FORMAT = format_name：表示使用指定的格式来输出执行计划信息，format_name 有 TRADITIONAL 和 JSON 两种输出格式，分别表示的含义如下：
  * TRADITIONAL：表示使用传统的输出格式，像查询一行数据横排那样输出，当然你也可以使用\G 来竖排输出
  * JSON：表示使用 json 输出格式，json 格式只会打印值不为 null 的列，json 格式中打印了更多的执行计划内部信息，5.7 中更是打印了详细的 cost 信息、行扫描信息以及使用到的列字段信息等

- explainable_stmt 有 5 种类型：select、delete、insert、replace、update（注意：虽然语法支持 insert、replace 语句，但是实际上执行计划是不支持的，输出结果在 5.6 版本中除了 id,select_type,Extra 里诶之外，其他的列全部都是 Null，因为并没有 where 条件可指定，在 5.7 版本中 table、possible_keys 和 type 可以显示内容了，但是 Extra 列反而变 null 了），5.6 之前的版本只支持查看 select 的执行计划，从 5.6 开始支持查看 DML 的执行计划
- 下面是几个使用 explain 语句查看执行计划的使用示例（以下以 5.7.17 版本做演示）

```
# 查看 select 执行计划
mysql > explain select * from test where id=1;
+----+-----------+-------+--------------+---------+---------+-------+------+----------+-------+
| id | select_type | table | partitions |type| possible_keys | key | key_len | re | rows | filtered | Extra |
+----+-----------+-------+--------------+---------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | test  | NULL   | const | PRIMARY| PRIMARY | 4   | const | 1 | 100.00 | NULL  |
+----+---------+--------+-------+--------+---------+---------+-------+------+----------+-------+
```

```
1 row in set, 1 warning (0.00 sec)

# 查看 update 执行计划
mysql > explain update test set id=2 where id=1;
+----+-------------+-------+-------+---------+---------+-------+------+----------+-------------+
| id | select_type| table | partitions | type | possible_keys | key| key_len | ref| rows | filtered | Extra |
+----+--------------+-------+-------+---------+---------+-------+------+----------+-------------+
|  1 | UPDATE | test | NULL| range | PRIMARY    | PRIMARY | 4| const | 1 |   100.00 | Using where |
+----+-------------+-------+--------+---------+---------+-------+------+----------+-------------+
1 row in set (0.00 sec)

# 查看 delete 执行计划
mysql > explain delete from test where id=1;
+----+------+------+------+----------+---------+---------+-------+------+----------+-------------+
| id | select_type |table | partitions| type| possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-------+-------+-------+-------+---------+---------+-------+------+----------+-------------+
|  1 | DELETE| test | NULL | range | PRIMARY | PRIMARY | 4 | const | 1 |    100.00 | Using where |
+----+--------+-------+-------+------+---------+---------+-------+------+----------+-------------+
1 row in set (0.00 sec)

# 查看 replace 执行计划
mysql > explain replace into test(test1) values(1);
+----+-------+-----------+------+-----------+------+---------+------+------+----------+-------+
| id | select_type | table| partitions | type| possible_keys | key| key_len | ref | rows | filtered | Extra |
+----+----------+---------+------+-----------+------+---------+------+------+----------+-------+
|  1 | REPLACE | test | NULL | ALL | PRIMARY | NULL | NULL   | NULL | NULL |       NULL | NULL   |
+----+----------+-----------+------+-----------+------+---------+------+------+----------+-------+
1 row in set (0.00 sec)

# 查看 insert 执行计划
mysql > explain insert into test(test1) values(1);
+----+---------+----------+------+-------------+------+---------+------+------+----------+-------+
```

```
| id | select_type | table | partitions |type | possible_keys | key| key_len | ref| rows | filtered | Extra |
+----+---------+----------+------+-----------+------+---------+------+------+----------+-------+
|  1 | INSERT | test| NULL | ALL | PRIMARY    | NULL | NULL     | NULL | NULL |      NULL | NULL  |
+----+---------+----------+------+-----------+------+---------+------+------+----------+-------+
1 row in set (0.00 sec)

# 查看分区表执行计划(5.7 中使用 partitions 关键字时，会出现两个 warnings 信息，其中一个就是告诉你这个关键字已经弃用了，另外一个是告诉你执行计划内部改写的语句是怎样的)
mysql > explain partitions select * from test where id=1;
+----+---------+--------+-------+----------+---------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type| possible_keys | key| key_len | ref| rows | filtered | Extra |
+----+----------+--------+-------+---------+---------+---------+-------+------+----------+-------+
|  1 | SIMPLE | test  | NULL| const | PRIMARY    | PRIMARY | 4 | const |    1 |    100.00 | NULL  |
+----+---------+---------+-------+---------+---------+---------+-------+------+----------+-------+
1 row in set, 2 warnings (0.00 sec)

mysql > show warnings;
+---------+------+-----------------------------------------------------------------------------------------+
| Level    | Code | Message                                                                                 |
+---------+------+-----------------------------------------------------------------------------------------+
| Warning | 1681 | 'PARTITIONS' is deprecated and will be removed in a future
release.                                                                                                |
| Note     | 1003 | /* select#1 */ select '1' AS `id`,'1' AS `test1`,'0' AS `test2`,NULL AS `test3`,NULL
AS `test4`,NULL AS `test5` from `xiaoboluo`.`test` where 1 |
+---------+------+-----------------------------------------------------------------------------------------+
2 rows in set (0.00 sec)

# 查看扩展的执行计划(5.7 中使用 extended 关键字时，会出现两个 warnings 信息，其中一个就是告诉你这个关键字已经弃用了，另外一个是告诉你执行计划内部改写的语句是怎样的)
mysql > explain extended select * from test where id=1;
+----+----------+---------+-------+-------+---------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type| possible_keys | key| key_len | ref| rows | filtered | Extra |
```

```
+----+-----------+-----+-------+----------+---------+---------+-------+------+----------+-------+
|  1 | SIMPLE | test   | NULL| const | PRIMARY | PRIMARY | 4   | const |     1 |    100.00 | NULL   |
+----+-----------+-------+-------+--------+---------+---------+-------+------+----------+-------+
1 row in set, 2 warnings (0.00 sec)

mysql > show warnings;
+---------+------+-----------------------------------------------------------------------------------------+
| Level    | Code | Message                |
+---------+------+-----------------------------------------------------------------------------------------+
| Warning | 1681 | 'EXTENDED' is deprecated and will be removed in a future
release.                                                                                                |
| Note     | 1003 | /* select#1 */ select '1' AS `id`,'1' AS `test1`,'0' AS `test2`,NULL AS `test3`,NULL
AS `test4`,NULL AS `test5` from `xiaoboluo`.`test` where 1 |
+---------+------+-----------------------------------------------------------------------------------------+
2 rows in set (0.00 sec)

# 以 json 格式输出执行计划
mysql > explain format=json select * from test where id=1;
| EXPLAIN   |
...
| {
  "query_block": {
    "select_id": 1,
    "cost_info": {
      "query_cost": "1.00"
    },
    "table": {
      "table_name": "test",
      "access_type": "const",
      "possible_keys": [
        "PRIMARY"
      ],
```

```
      "key": "PRIMARY",
      "used_key_parts": [
        "id"
      ],
      "key_length": "4",
      "ref": [
        "const"
      ],
      "rows_examined_per_scan": 1,
      "rows_produced_per_join": 1,
      "filtered": "100.00",
      "cost_info": {
        "read_cost": "0.00",
        "eval_cost": "0.20",
        "prefix_cost": "0.00",
        "data_read_per_join": "1K"
      },
      "used_columns": [
        "id",
        "test1",
        "test2",
        "test3",
        "test4",
        "test5"
      ]
    }
  }
} |
...
1 row in set, 1 warning (0.00 sec)
```

PS：以上查看执行计划步骤中也可以把 explain 关键字换做 desc 和 describe，执行结果相同，大家自行尝试，这里不再赘述

- 下面是使用 EXPLAIN | DESCRIBE | DESC 几个关键字查看表定义信息的示例

```
# 使用 explain 关键字查看表定义信息
mysql > explain test;
+-------+------------------+------+-----+---------+----------------+
| Field | Type             | Null | Key | Default | Extra          |
+-------+------------------+------+-----+---------+----------------+
| id    | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| test1 | varchar(100)     | NO   |     | 0       |                |
| test2 | varchar(100)     | NO   |     | 0       |                |
| test3 | varchar(100)     | YES  | MUL | NULL    |                |
| test4 | varchar(100)     | YES  | MUL | NULL    |                |
| test5 | varchar(100)     | YES  | MUL | NULL    |                |
+-------+------------------+------+-----+---------+----------------+
6 rows in set (0.00 sec)

# 使用 desc 关键字查看表定义信息
mysql > desc test;
+-------+------------------+------+-----+---------+----------------+
| Field | Type             | Null | Key | Default | Extra          |
+-------+------------------+------+-----+---------+----------------+
| id    | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| test1 | varchar(100)     | NO   |     | 0       |                |
| test2 | varchar(100)     | NO   |     | 0       |                |
| test3 | varchar(100)     | YES  | MUL | NULL    |                |
| test4 | varchar(100)     | YES  | MUL | NULL    |                |
| test5 | varchar(100)     | YES  | MUL | NULL    |                |
```

```
+-------+-----------------+------+-----+---------+----------------+
6 rows in set (0.00 sec)

# 使用 describe 语句查看表定义信息
mysql > describe test;
+-------+-----------------+------+-----+---------+----------------+
| Field | Type            | Null | Key | Default | Extra          |
+-------+-----------------+------+-----+---------+----------------+
| id    | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
| test1 | varchar(100)    | NO   |     | 0       |                |
| test2 | varchar(100)    | NO   |     | 0       |                |
| test3 | varchar(100)    | YES  | MUL | NULL    |                |
| test4 | varchar(100)    | YES  | MUL | NULL    |                |
| test5 | varchar(100)    | YES  | MUL | NULL    |                |
+-------+-----------------+------+-----+---------+----------------+
6 rows in set (0.00 sec)
```

2、执行计划输出格式、结果详解

- 以下统一以使用 explain 关键字为例查看执行计划进行说明

2.1. 执行计划输出列信息

2.1.1. 执行计划输出列含义简介

- EXPLAIN 的每个输出行只表示一个表的相关信息。 每个输出行包含如表附 D-1 列出的字段信息（表格中的第一列 column 表示 explain 输出中的列名， 第二列表示 FORMAT = JSON 格式输出中显示对应 explain 列名的等效名称，注意：json 格式不显示输出为 null 值的列）

表附 D-1

Column	JSON Name	含义
id	select_id	每个输出行的 SELECT 标识符
select_type	None	每个输出行的查询执行的查询类型
table	table_name	每个输出行的查询的表名
partitions	partitions	每个输出行的查询扫描了哪些分区
type	access_type	每个输出行的查询执行联结查询的类型
possible_keys	possible_keys	每个输出行的查询可能使用到的索引
key	key	每个输出行的查询最后真正使用到的索引
key_len	key_length	每个输出行的查询用于检索数据时实际使用到的索引长度，单位为字节
ref	ref	每个输出行的查询使用索引列进行值比较的类型，不同类型会显示不同的类型值
rows	rows	每个输出行的查询扫描的估算行数
filtered	filtered	每个输出行的查询中 where 条件过滤掉不满足条件之后剩下真正需要的数据与存储引擎实际返回行数的百分比
Extra	None	每个输出行的查询的一些附加的执行计划信息

2.1.2. 执行计划输出列含义详解

- id(JSON name: select_id)：SELECT 标识符。 这是查询中 SELECT 的序列号。 如果行引用了其他输出行的 union 查询结果（即这是其他表的查询的结果集，不需要使用它来进行查询），则该值可能为 NULL。在这种情况下，table 列显示类似的值，表示该行引用输出行 id 值为 M

和 N 的行对应查询结果的并集，id 列数字越大越先执行，如果说数字一样大，那么就从上往下依次执行

- select_type(JSON name: none)：SELECT 的类型，可以如表附 D-2 中显示的任何一种。 JSON 格式的 EXPLAIN 中没有等效的 SELECT select_type 值与之对应，而是在不同查询类型中使用一个 query_block 集合的属性来显示更详细的信息（注：以下表格是 select_type 字段值及其含义）

表附 D-2

select_type Value	JSON Name	含义
SIMPLE	None	表示不需要 union 操作或者不包含子查询的简单 select 查询。有连接查询时，外层的查询为 simple，且只有一个
PRIMARY	None	一个需要 union 操作或者含有子查询的 select，位于最外层的单位查询的 select_type 即为 primary。且只有一个
UNION	None	union 连接的两个 select 查询，第一个查询是 dervied 派生表，除了第一个表外，第二个以后的表 select_type 都是 union

DEPENDENT UNION	dependent (true)	与 union 一样，出现在 union 或 union all 语句中，但是这个查询要受到外部查询的影响
UNION RESULT	union_result	包含 union 的结果集，在 union 和 union all 语句中,因为它不需要参与查询，所以 id 字段为 null
SUBQUERY	None	除了 from 字句中包含的子查询外，其他地方出现的子查询都可能是 subquery
DEPENDENT SUBQUERY	dependent (true)	与 dependent union 类似，表示这个 subquery 的查询要受到外部表查询的影响
DERIVED	None	from 字句中出现的子查询，也叫做派生表，其他数据库中可能叫做内联视图或嵌套 select
MATERIALIZED	materialized_from_subquery	物化子查询
UNCACHEABLE SUBQUERY	cacheable (false)	一个子查询，其结果无法缓存，必须对外部查询的每一行重新扫描进行值匹配

UNCACHEABLE UNION	cacheable (false)	第二个或以后的 UNION 查询是一个 UNCACHEABLE SUBQUERY 类型（见 UNCACHEABLE SUBQUERY 解释部分）

- table (JSON name: table_name)：输出行引用的表名称，可能的值有如下几种（显示的查询表名，如果查询使用了别名，那么这里显示的是别名，如果不涉及对数据表的操作，那么这显示为 null，如果显示为尖括号括起来的就表示这个是临时表，后边的 N 就是执行计划中的 id，表示结果来自于这个查询产生。如果显示<unionM,N>，则表示这个结果来自于 union 查询的 id 为 M,N 的结果集）

tb_name or null：显示查询具体使用的表名称字符串，或者查询使用了别名，显示别名字符串，或者不涉及对数据的操作，显示为 null

<unionM，N>：显示该行引用的是 id 值为 M 和 N 的行的查询结果集，表示 M 和 N 两个查询执行了 union 查询

<derivedN>：显示该行引用的是 id 值为 N 的行的查询结果集。表示 N 使用了派生表查询，派生表可能由如 FROM 子句中的子查询产生的

<subqueryN>：显示该行引用的是 id 值为 N 的行的查询的结果集。表示 N 使用了物化子查询

表附 D-3

连接类型名称	含义
system	表中只有一行数据或者是空表，且只能用于 myisam 和 memory 表。如果是 Innodb 引擎表，type 列在这个情况通常都是 all 或者 index

const	使用唯一索引或者主键，返回记录一定是 1 行记录的等值 where 条件时，通常 type 是 const。其他数据库也叫做唯一索引扫描
eq_ref	出现在要连接过个表的查询计划中，驱动表只返回一行数据，且这行数据是第二个表的主键或者唯一索引，且必须为 not null，唯一索引和主键是多列时，只有所有的列都用作比较时才会出现 eq_ref
ref	不像 eq_ref 那样要求连接顺序，也没有主键和唯一索引的要求，只要使用相等条件检索时就可能出现，常见与辅助索引的等值查找。或者多列主键、唯一索引中，使用第一个列之外的列作为等值查找也会出现，总之，返回数据不唯一的等值查找就可能出现。
fulltext	全文索引检索，要注意，全文索引的优先级很高，若全文索引和普通索引同时存在时，mysql 不管代价，优先选择使用全文索引
ref_or_null	与 ref 方法类似，只是增加了 null 值的比较。实际用的不多。
unique_subquery	用于 where 中的 in 形式子查询，子查询返回唯一值
index_subquery	用于 in 形式子查询使用到了辅助索引或者 in 常数列表，子查询可能返回重复值，可以使用索引将子查询去重。
range	索引范围扫描，常见于使用>,<,is null,between ,in ,like 等运算符的查询中。
index_merge	表示查询使用了两个以上的索引，最后取交集或者并集，常见 and ，or 的条件使用了不同的索引，官方排序这个在 ref_or_null 之后，但是实际上由于要读取所有索引，性能可能大部分时间都不如 range
index	索引全表扫描，把索引从头到尾扫一遍，常见于使用索引列就可以处理不需要读取数据文件的查询、可以使用索引排序或者分组的查询。
all	这个就是全表扫描数据文件，然后再在 server 层进行过滤返回符合要求的记录。

- partitions(JSON name: partitions)：查询匹配到的分区名称（表示该查询给定的 where 条件将要扫描到的分区），如果是非分区表，则显示的值为 NULL
- type (JSON name: access_type)：查询使用的连接类型，下面简单列出常见类型值含义(依次从好到差：system，const，eq_ref，ref，fulltext，ref_or_null，unique_subquery，index_subquery，range，index_merge，index，all，除了 all 之外，其他的 type 都可以使用到索引，除了 index_merge 之外，其他的 type 只可以用到一个索引)

- possible_keys (JSON name: possible_keys)：possible_keys 列显示了 MySQL 可以选择哪些索引来查找此表中的数据行。如果此列为 NULL（或在 JSON 格式中显示为 undefined），则表示该查询没有相关的可用的索引。 在这种情况下，您可以通过检查 WHERE 子句来检查其是否引用含有索引的一些列，如果 where 中的所有列都不能使用到索引，则创建一个适当的索引（适合本查询 where 条件的索引）并再次使用 EXPLAIN 检查查询。要查看表有哪些索引，可以使用 SHOW INDEX FROM tbl_name 语句查看
- key(JSON name: key)：
    - key 列显示了 MySQL 实际使用的 key（索引）。如果 MySQL 决定使用一个 possible_keys 中列出的索引来查找行，那么该索引列的 key 和 value 将被使用来进行数据检索和过滤
    - key 可能会指定一个在 possible_keys 中不存在的索引。如果查询的 select 中列出的所有列被某个索引的列涵盖（即查询优化器指定的索引的列覆盖到了查询语句的 select 列）而 possible_keys 中又没有合适的索引，就可能发生这种情况，这个时候，尽管查询优化器不能使用指定的这个索引来确定要检索哪些行，但可以使用指定的这个索引进行索引覆盖查询。因为索引扫描比数据行扫描更有效。即就是通过索引列值直接返回 select 列数据来避免全表扫描。

    - 对于 InnoDB，如果查询的 select 列中包含了主键，此时如果某个辅助索引的索引列覆盖到了查询语句 select 列时，也会选择使用辅助索引进行覆盖索引查询，因为 InnoDB 的辅助索引数据结构中会将主键值与每个辅助索引列值一起存储。如果执行计划中 key 为 NULL，则表示 MySQL 并没有发现有索引可用于更有效地执行查询（此时只能走全表扫描了）
    - 对于 MyISAM 表，运行 ANALYZE TABLE 有助于优化器选择更好的索引。也可以使用 myisamchk --analyze 命令
    - 要强制 MySQL 使用或忽略 possible_keys 列中列出的索引，可以在查询中使用 FORCE INDEX，USE INDEX 或 IGNORE INDEX 关键字
    - select_type 为 index_merge 时，这里可能出现两个以上的索引，其他的 select_type 这里只会出现一个
- key_len(JSON name: key_length)：key_len 列表示 MySQL 使用到的索引 key 的长度(单位为字节)。 key_len 的值使您能够确定在多列索引中 MySQL 实际使用的索引部分有哪些。 如果 key 列显示 NULL，则 key_len 列也会显示为 NULL（key_len 只计算 where 条件用到的索引长度，而排序和分组就算用到了索引，也不会计算到 key_len 中），key_len 计算及其影响因素如下：
    - key_len 只计算用于数据检索的索引列，用于 order by ，group by 不会算在其中（如果是单列索引，那就整个索引长度算进去，如果是多列索引，那么查询不一定都能使用到所有的列，具体使用到了多少个列的索引，这里就会计算进去，没有使用到的列，这里不会计算进去，5.7 的 explain 的 json 格式中的 used_key_parts 属性可以看到检索使用到了哪些索引列）
    - key_len 计算会受到数据类型及其定义属性影响，varchar 变长数据类型需要+2，如果字段值允许为 null 的则需要+1
    - key_len 计算受到字符集和数据类型定义长度影响，lantin1 字符集按数据类型定义长度 1 个字节计算，gbk 字符集按数据类型定义长度 2 个字节计算，utf8 字符集按数据类型定义长度 3 个字节计算，utf8mb4 字符集按数据类型定义长度 4 个字节计算

    - 如果使用到的索引是前缀索引，则 key_len 只会计算索引定义的前缀长度，不会计算整个索引列的长度
    - mysql 的 ICP 特性使用到的索引不会计入其中
- ref (JSON name: ref)：如果是使用的常数等值查询，这里会显示 const，如果是连接查询，被驱动表的执行计划这里会显示驱动表的关联字段，如果是条件使用了表达式或者函数，或者条件列发生了内部隐式转换，这里可能显示为 func。要查看更详细的信息，可以使用 EXPLAIN 之后查看扩展 EXPLAIN 输出 SHOW 警告。
- rows (JSON name: rows)：rows 列显示 MySQL 认为查询必须扫描的行数。对于 InnoDB 表，此数字是一个估计值，可能不一定准确。
- filtered (JSON name: filtered)：filtered 列显示了在 server 层按照表过滤条件过滤之后剩下数据行数与表数据总行数的估算百分比。 不是精确值（使用 explain extended 时会出现这个列，5.7 之后的版本默认就有这个字段，不需要使用 explain extended 了）
- Extra (JSON name: none)：此列包含有关 MySQL 如何解析查询的附加信息。在 json 输出格式中，没有与 Extra 列对应的单个 JSON 属性; 但是，此列中可能出现的值将作为 JSON 格式中的 message 属性文本显示。表附 D-4 中列出常用的十几种附加信息的含义。

表附 D-4

Extra 值名称	**含义**
distinct	在 select 部分使用了 distinc 关键字
no tables used	不带 from 字句的查询或者 From dual 查询
not exitsts	使用 not in()形式子查询或 not exists 运算符的连接查询，这种叫做反连接。即，一般连接查询是先查询内表，再查询外表，反连接就是先查询外表，再查询内表。
using filesort	排序时无法使用到索引时，就会出现这个。常见于 order by 和 group by 语句中
using index	查询时不需要回表查询，直接通过索引就可以获取查询的数据。

using join buffer（block nested loop），using join buffer（batched key accss）	5.6.x 之后的版本优化关联查询的 BNL，BKA 特性。主要是减少内表的循环数量以及比较顺序地扫描查询。
using sort_union，using_union，using intersect，using sort_intersection	using intersect：表示使用 and 的各个索引的条件时，该信息表示是从处理结果获取交集,using union：表示使用 or 连接各个使用索引的条件时，该信息表示从处理结果获取并集,using sort_union 和 using sort_intersection：与前面两个对应的类似，只是他们是出现在用 and 和 or 查询信息量大时，先查询主键，然后进行排序合并后，才能读取记录并返回。
using temporary	表示使用了临时表存储中间结果。临时表可以是内存临时表和磁盘临时表，执行计划中看不出来，需要查看 status 变量 Created_tmp_tables、Created_tmp_disk_tables，或者 percona、mariadb 的慢查询日志中 used_tmp_table，used_tmp_disk_table 标记才能看出来。
using where	表示存储引擎返回的记录并不是所有的都满足查询条件，需要在 server 层进行过滤。查询条件中分为限制条件和检查条件，5.6 之前，存储引擎只能根据限制条件扫描数据并返回，然后 server 层根据检查条件进行过滤再返回真正符合查询的数据。5.6.x 之后支持 ICP 特性，可以把检查条件也下推到存储引擎层，不符合检查条件和限制条件的数据，直接不读取，这样就大大减少了存储引擎扫描的记录数量。extra 列显示 using index condition
firstmatch(tb_name)	5.6.x 开始引入的优化子查询的新特性之一，常见于 where 字句含有 in()类型的子查询。如果内表的数据量比较大，就可能出现这个

loosescan(m..n)	5.6.x 之后引入的优化子查询的新特性之一，在 in() 类型的子查询中，子查询返回的可能有重复记录时，就可能出现这个

- PS：
  - DEPENDENT 通常使用于相关子查询
  - DEPENDENT SUBQUERY 与 UNCACHEABLE SUBQUERY 在查询计划中的评估算法区别：对于 DEPENDENT SUBQUERY，对于来自其外部上下文的变量的每个集合，子查询仅评估一次。对于 UNCACHEABLE SUBQUERY，对外部上下文的每一行，子查询都要重新评估计算
  - 子查询的可缓存性不同于查询缓存中的查询结果集的缓存，子查询缓存在查询执行期间发生，而查询缓存仅在查询执行完成后才用于存储查询的结果集
  - 当使用 EXPLAIN 指定 FORMAT = JSON 时，输出结果中没有直接等同于 select_type 值的单个属性
  - 非 SELECT 语句的 select_type 值受表的语句类型影响。例如，对于 DELETE 语句，select_type 为 DELETE，以此类推

2.1.4. type 列值详解

- EXPLAIN 输出的类型列描述查询表如何连接。 在 JSON 格式的输出中，type 列值对应 access_type 属性的值。 以下列出 type 列可能出现的连接类型值，顺序从最佳到最差
  - system：查询的表中只有一行数据。 这是 const 连接类型的一种特殊情况（表中只有一行数据或者是空表，且只能用于 myisam 和 memory 表，如果是 innodb 表，则在 5.6 之前的版本，多数时候可能为 all 或者 index，在 5.6 之后的版本中，有一行数据时显示为 const，没有数据时显示为 null）

- const：查询的表最多只有一个匹配的行， 因为只有一行，所以该行中的列的值可以被优化程序视为常量（使用唯一索引或者主键，返回记录一定是 1 行记录的等值 where 条件时，通常 type 是 const。其他数据库也叫做唯一索引扫描）。const 类型查询非常快，当使用 PRIMARY KEY 或 UNIQUE 索引的所有列进行查询时(与常量值进行比较时)，将使用 const(因为主键和唯一索引能够保证值唯一)。tbl_name 可以用作 const 表的示例：SELECT * FROM tbl_name WHERE primary_key = 1；SELECT * FROM tbl_name WHERE primary_key_part1 = 1 AND primary_key_part2 = 2;
- eq_ref：出现在要连接多个表的查询计划中，被驱动表（或者说连接嵌套中的前一个表）中的每一行在与驱动表（或者说连接嵌套中的后一个表）的数据查询结果集匹配中只有一行数据匹配时，且这行数据是第二个表的主键或者唯一索引，且必须为 not null，该表查询类型为 eq_ref。如果使用到的索引是多列索引，那么只有当索引的所有列都被连接查询使用到且索引为 PRIMARY KEY 或 UNIQUE NOT NULL 索引时才会出现 eq_ref。 使用 column=比较运算符的形式也可以使用到索引列， MySQL 可以使用 eq_ref 连接来处理 ref_table 的示例：SELECT * FROM ref_table，other_table WHERE ref_table.key_column = other_table.column; SELECT * FROM ref_table，other_table WHERE ref_table.key_column_part1 = other_table.column AND ef_table.key_column_part2 = 1;
- ref：不像 eq_ref 那样要求连接顺序，也没有主键和唯一索引的要求，只要使用相等条件检索时就可能出现，常见与辅助索引的等值查找，或者多列主键、唯一索引中，使用第一个列之外的列作为等值查找也会出现，或者使用最左前缀索引查找中也可能会出现。总之，返回数据不唯一的等值查找就可能出现，换句话说，如果连接不能基于 key 匹配单个行，则使用 ref
  * ref 可以用于使用=或<=>运算符进行比较的索引列。 MySQL 可以使用 ref 连接来处理 ref_table 的示例：SELECT * FROM

ref_table WHERE key_column=expr; SELECT * FROM ref_table,other_table WHERE ref_table.key_column=other_table.column; SELECT * FROM ref_table,other_table WHERE ref_table.key_column_part1=other_table.column AND ref_table.key_column_part2=1;

- fulltext：使用 FULLTEXT 索引执行连接查询（要注意，全文索引的优先级很高，若全文索引和普通索引同时存在时，mysql 不管代价，优先选择使用全文索引）
- ref_or_null：这种连接类型类似于 ref，但 MySQL 还会额外搜索包含 NULL 值的行。此连接类型优化最常用于解析子查询（与 ref 方法类似，只是增加了 null 值的比较。实际用的不多）。MySQL 可以使用 ref_or_null 连接来处理 ref_table 的示例：SELECT * FROM ref_table WHERE key_column = expr 或 key_column IS NULL;
- index_merge：此连接类型表示使用索引合并优化。 在这种情况下，explain 输出行中的 key 列包含 index_merge 优化所使用的索引列表（表示查询使用了两个以上的索引，最后取交集或者并集，常见 and ，or 的条件使用了不同的索引），而 key_len 使用最长索引部分来计算（注意：实际查询中由于可能要读取多个索引，性能可能大部分时间还不如 range，有些业内人士建议排在 range 之后）
- unique_subquery：用于 where 中的 in 形式子查询，子查询返回唯一值，如： value IN（SELECT primary_key FROM single_table WHERE some_expr），其中 IN 子句中的 select 字段是为 IN 子句中的查询的主键或唯一索引。
- index_subquery：此连接类型与 unique_subquery 类似。 用于 in 形式子查询使用到了辅助索引或者 in 常数列表，子查询可能返回重复值，可以使用索引将子查询去重：value IN（SELECT key_column FROM single_table WHERE some_expr）,其中 IN 子句中的 select 字段为 IN 子句中的查询的一个非唯一索引的 Key

- range：使用索引且只检索指定范围内的行，explain 输出行中的 key 列显示了使用哪个索引， key_len 列按照所使用的索引最长部分进行计算。range 类型查询的 explain 输出行中的 type 的值为 NULL，当使用=，<>，>，> =，<，<=，IS NULL，<=>，like，BETWEEN 或 IN()运算符+一个常量值或表达式返回一个常量值时，会使用 range 类型查询，示例：SELECT * FROM tbl_name WHERE key_column = 10; SELECT * FROM tbl_name WHERE key_column BETWEEN 10 和 20; SELECT * FROM tbl_name WHERE key_column IN（10,20,30）; SELECT * FROM tbl_name WHERE key_part1 = 10 AND key_part2 IN（10,20,30）;
- index：除了扫描索引树这种情况之外，其他的 index 连接类型与 ALL 类型相同。index 有如下两种方式：
  * 如果索引是查询的覆盖索引，并且可以用于满足查询所需的所有数据，则查询只扫描索引树。在这种情况下，Extra 列显示 Using index。index 扫描通常比 ALL 快，因为索引的大小通常小于全表数据
  * 使用索引读取来执行全表扫描，以按索引顺序查找数据行。在这种情况下，Extra 列不会出现 Using index
  * 当查询仅使用单个索引的部分索引时，MySQL 可以使用此连接类型
- ALL：这个就是全表扫描数据文件，然后再在 server 层进行过滤返回符合要求的记录。通常，可以通过添加索引来避免 ALL，

2.1.5. Extra 列值详解

- EXPLAIN 输出的 Extra 列包含有关 MySQL 如何解析查询的附加信息。以下列出了在 explain 的 Extra 列中可能出现的值。 每个值同时还列出了 JSON 格式的输出对应的属性值。其中一些在 json 格式中有一个特定的属性。一些显示为 message 属性。如果你希望你的查询尽可能快地完成，请注意 Using filesort 和 Using temporary 的值，与之对应的在

JSON 格式的 EXPLAIN 输出中是 Using_filesort 和 use_temporary_table 属性。注：以下解释部分出现的 SQL 示例的 employees 和 salaries 表使用的是 mysql 的样例数据库 employees 库，下载链接：https://github.com/datacharmer/test_db，安装方法参考地址：https://dev.mysql.com/doc/employee/en/employees-installation.html

- Child of 'table' pushed join@1 (JSON: message text)：表示此表在连接中被当作子表引用，可以向下推送到 NDB 内核，仅仅在 pushed-down joins 功能开启的 MySQL cluster 中使用，详情参见 ndb_join_pushdown 系统变量描述部分
- const row not found (JSON property: const_row_not_found)：对于使用类似 SELECT ... FROM tbl_name 的查询一张空表时会出现这个值，即表示虽然在执行计划中，显示使用 const 方式访问读取了数据表，但是实际上表中没有行符合条件的记录。查看一下是不是你忘记了插入测试数据。
- Deleting all rows (JSON property: message)：对于 DELETE，一些存储引擎（例如 MyISAM）支持一种以简单快速的方式删除所有表行的处理方法。 如果引擎使用到了此优化方法，将出现这个值。这个是从存储引擎的 Handler 角度来看的。在不带 where 条件的 delete 语句的执行计划中，经常会发现这个提示信息。它表示调用一次删除数据表所有记录的 Handler 功能（API），以前删除数据表记录时，要一句记录条数反复调用各存储引擎的 Handler 函数，而采用了 Deleting all rows 优化处理方式则只需要调用一次 Handler 函数，处理起来更加快速（注意：实测 5.6.35，5.7.17，mariadb10.1.22 的 myisam，innodb,aria，memory 引擎都不会出现这个提示，估计在早期版本中才会出现）
- Distinct (JSON property: distinct)：查询 select 字段使用了 distinct 关键字。表示 MySQL 正在寻找去重的值,所以在发现第一个匹配的行之后停止寻找更多行以便对当前行进行合并
  * 举个例子：explain select distinct d.dept_no from

departments d,dept_emp de where de.dept_no=d.dept_no; 该语句实际上是需要访问 d 表的 dept_no 字段，而 dept_no 在 d 表与 de 表数据表中都存在，执行查询时会先将两个表的数据连接起来，然后再在结果集中进行 distinct 处理，这样 dept_no 就不会有重复的了

- FirstMatch(tbl_name) (JSON property: first_match)：表示对于 tbl_name 表（tbl_name 表是外表）的连接查询使用了简单的半连接的 FirstMatch 策略，该策略是 mariadb 5.3 与 mysql5.6 之后引入的针对子查询的优化策略，在这之前的版本中经常将子查询(IN subquery)转换为 exists subquery 类型的查询（即，IN-TO-EXISTS 优化策略）。策略算法是先从外表数据表读取一行记录，然后检索内表的记录，直到在内表中检索到匹配外表条件值的记录为止。如果在子查询中 1 次即可查找到符合条件的记录，那么执行代价会非常低，如果子查询中不存在任何记录与外表的 where 条件匹配，那么执行代价可能会非常高，因为外表的每一行记录都会对内表进行全表扫描，直到找到一行匹配记录为止。 举个例子：
  * explain select * from departments d where exists(select 1 from dept_emp de where de.dept_no=d.dept_no); 在 5.7 中该语句秒执行，但是并没有看到 FirstMatch 的提示信息，在 select_type 列显示的是 DEPENDENT SUBQUERY。在 mariadb 10.1.22 中，会使用 MATERIALIZED 查询类型来先把子查询（de 表）的内容具体化（物化）成一个临时表，然后依次连接 d 表与临时表，再执行查询。这种效率要比使用 FirstMatch()优化策略要低，使用 MATERIALIZED 查询类型时，时间大约为 0.3S
  * explain select * from departments where dept_no in(select dept_no from dept_emp); 该语句中，在 mysql 5.6.x，mysql5.7.x 版本的执行计划中会出现这个提示信息，但是在 mariadb 的执行计划中，仍然使用子查询 MATERIALIZED 的方式。

- Full scan on NULL key (JSON property: message)：这发生在如果优化器无法使用索引查找访问方法，那么就把子查询优化作为一个后备策略。经常出现在类似 col1 in (select col2 from ...) 的 WHERE 条件查询中，若 col1 值为 null，则条件最终变为 null in (select col2 from ...)，col1 如果定义为允许为 null 时，必须进行全表扫描才能查询出结果（也可以改写语句加一个 col1 is not null and col1 in (...)，这样就等于显式告诉优化器 col1 不可能为 null，就不会进行全表扫描，也不会在 Extra 中显示 Full scan on NULL key 信息，要注意，如果 col1 可能为 null 时，子查询中一定要指定 where 条件，否则会导致严重的性能问题），若 col1 不可能为 null(col1 定义为 not null)，则不会使用全表扫描。SQL 标准将 null 定义为未知值，也定义了 null 的运算规则，根据规则定义，执行运算时需要对条件进行如下比较
  * 若子查询拥有至少一条结果记录，则最终比较结果为 null
  * 若子查询未拥有任何结果记录，则最终比较结果为 FALSE
- Impossible HAVING (JSON property: message)：HAVING 子句始返回 false（不实际执行，而是通过数据表结构判断条件为不可能），表示不存在满足查询 HAVING 子句的记录，举个例子
  * explain select e.emp_no,count(*) from employees e where e.emp_no=10001 group by e.emp_no having e.emp_no is null; 在该查询语句中，因为 e.emp_no 是 e 表的主键，所以不可能出现 e.emp_no is null 的情况，那么执行计划中就会出现这个提示，在实际应用中如果出现这个提示，很可能是查询语句编写错误。
- Impossible WHERE (JSON property: message)：WHERE 子句始返回 false（不实际执行，而是通过数据表结构判断条件为不可能），表示不存在满足查询 WHERE 子句的记录，举个例子
  * explain select * from employees where emp_no is null; 该语句中，emp_no 列为表的主键，不可能为 null，所以执行计划中出现 Impossible WHERE 提示。跟 Impossible HAVING 类似，很可能是因为查询语句编写错误导致

- Impossible WHERE noticed after reading const tables (JSON property: message)：前面说了，Impossible 是不实际执行，而是通过数据表结构判断条件为不可能，但是有时候如果不实际执行就无法判断记录是否存在（如：explain select * from emp where emp_no=0），事实上只需要优化器执行一下查询计划，就知道该记录是否存在。那么实际执行过程中是怎样的呢？只需要执行查询的一部分即可，执行查询记录是否存在使用的是 const 类型查询，在一些 join 查询中还会先使用部分执行结果进行内部值替换（如：select * from emp as e where oe.first_name=(select ie.first_name from emp as ie where ie.emp_no=1)在执行计划查看时优化器会先把 ie.emp_no=1 对应的记录的 oe.first_name 值查询出来，然后把语句改写为 select * from emp as oe where oe.first_name='xxx'，然后如果 xxx 值不存在，可能就显示 Impossible WHERE noticed after reading const tables 值）。注意：在 5.7 之前单表简单查询中，如果值不存在会显示这个值，但是在 5.7 之后的版本，如果单表简单查询值不存在，Extra 列显示的也是 no matching row in const table，而不是像 5.7 之前的版本需要在 const 类型的 join 查询中值不存在才显示 no matching row in const table
- LooseScan(m..n) (JSON property: message)：使用半连接 LooseScan 策略。 m 和 n 是 key 部分数字。mysql5.6 和 mariadb5.3 开始引入的策略。用于优化子查询的众多策略之一。在 in subquery 类型的查询中，子查询的结果可能产生重复记录时，使用该优化方法。子查询的结果集去重后被用作驱动表，与外表进行连接查询。使用到该优化策略时，不需要使用临时表来存放结果集。举个例子（在 mysql 和 mariadb 中都需要关闭 firstmatch 和 materialization 才能引导查询优化器使用该策略：set optimizer_switch=default,optimizer_switch='firstmatch=off,materialization=off';）：
  * explain select * from departments where dept_no

in(select dept_no from dept_emp); 该语句在 mysql5.6.35，mysql5.7.17，mariadb 10.1.22 版本中测试时，如果不关闭 firstmatch 和 materialization 策略，mysql 使用的是 FirstMatch 策略，mariadb 使用的是 MATERIALIZED 查询类型。如果都关闭 firstmatch 和 materialization 策略，mysql 使用的是 Start temporary; End temporary，mariadb 使用的 LooseScan 策略。

- No matching min/max row (JSON property: message)：没有满足查询条件的记录，例如 SELECT MIN（...）FROM ... WHERE xx=value。前面说了，如果表中没有记录满足 where 条件，则应该出现类似 Impossible xxx 的 Extra 提示，但是如果查询使用了聚合函数 max()/min()，且表中没有任何符合条件的记录时，则 Extra 显示这个提示信息。举个例子
  * explain select min(dept_no),max(dept_no) from dept_emp where dept_no=''; 该语句中，使用到了 min()、max()聚合函数，且无任何符合查询条件的记录。
- no matching row in const table (JSON property: message)：对于联接查询，连接表中存在有空表或没有表满足唯一索引条件的记录。即，使用了 type 为 const 类型的连接查询的数据表中，若不存在符合条件的记录，则执行计划中的 Extra 列显示这个提示信息。注：前面 Impossible WHERE noticed after reading const tables (JSON property: message)解释部分提到过，5.7 之后的的 const 类型的连接查询中若没有匹配值则提示这个信息，除此之外，5.7 之后的单表 const 类型查询或者 5.7 之前的版本单表 const 类型或连接表 const 类型查询没有匹配记录返回的，则提示 Impossible WHERE noticed after reading const tables (JSON property: message)信息。举个例子：
  * explain select * from dept_emp de,(select emp_no from employees where emp_no=0) tb1 where tb1.emp_no=de.emp_no and de.dept_no='d005'; 注意：该语句在 5.6.x 版本以及 mariadb10.0.x 版本的执行计划中会出现

两行记录，id=2 的行为派生表，派生表的 Extra 列会出现该提示信息，但在 5.7.x 以及之后的版本的执行计划中只有一行记录，且 Extra 列输出为“no matching row in const table”提示，在 mariadb 10.1.x 及其之后的版本的执行计划中也只有一行记录，但是 EXtra 输出列为“Impossible WHERE noticed after reading const tables”提示，原因是两者查询优化器内部改写 SQL 时产生的 employees 表的条件值不一样了。

- No matching rows after partition pruning (JSON property: message)：对于 DELETE 或 UPDATE，在使用分区表的查询中，执行计划经过分区修剪之后（mysql 检索分区表时，会从操作目录中删除不需要使用分区健列检索的分区，这称为分区修剪）没有发现任何匹配记录（mysql 的查询优化器在制定查询的执行计划时，若不存在符合 WHERE 条件的分区，则 Extra 列中出现这个提示信息），它的意义类似于 SELECT 语句 Impossible WHERE 提示。如果出现时，检查查询语句是否编写错误。
- No tables used (JSON property: message)：查询没有 FROM 子句，或有一个 FROM DUAL 子句。对于 INSERT 或 REPLACE 语句，EXPLAIN 在没有 SELECT 部分时显示此值。 例如，在 EXPLAIN INSERT INTO t VALUES（10）中的执行计划会显示这个提示信息，因为这个语句相当于是 EXPLAIN INSERT INTO t SELECT 10 FROM DUAL。注：前面提到过，5.6 之后的版本可以查看 DML 的执行计划，但是 INSERT 和 REPLACE 除外，另外在 5.7 之前的版本 INSERT 和 REPLACE 的执行计划中会显示这个提示信息，但在 5.7 之后的版本中执行计划中直接显示 null
- Not exists (JSON property: message)：MySQL 对查询执行 LEFT JOIN 优化，在 LEFT JOIN 查询中如果左表中的记录在右表中匹配到了一行，那么就不会在右表中匹配更多行。此时一般会使用 not in subquery 或 not exists 运算符，这种连接称为反连接(anti-join)。同理：也可以用于 left join 查询。先查询外表中的数据，然后再在 where 子句中检查并获取外部表中的关联字段不为 null 的记录（注意：这里说的外表，内表是相对于外连接查询（left outer join、right outer join）而言，不是子查询类型

的连接）。也就是说，反连接方法常常用于普通连接(inner join)不会出现的结果。

* 举个例子：SELECT * FROM t1 LEFT JOIN t2 ON t1.id = t2.id WHERE t2.id IS NULL; 假定 t2.id 列属性被为 NOT NULL。在这种情况下，MySQL 扫描 t1 表并使用 t1.id 列的值查找 t2 中的行。 如果 MySQL 在 t2 中找到了匹配的行，那么因为 t2.id 列属性定义 not null，那么查询优化器知道 t2.id 永远不可能为 NULL，那么就不会继续扫描 t2 中具有相同 id 值的其余行。换句话说，对于 t1 中的每一行，MySQL 需要在 t2 中只进行一次查找，只要匹配到了一行 t2.id 不为 null 则跳过 t2 表的其余行的扫描，而不管 t2 中实际匹配多少行。转而使用 t1 表中的下一行记录进行扫描 t2 表。

- Plan isn't ready yet (JSON property: none)：在命名连接中使用 EXPLAIN FOR CONNECTION 语句，查询优化器未完成对查询语句创建执行计划时，会出现这个提示信息。 如果执行计划输出包含多个行，则它们中的其中一个或全部输出行的 Extra 列都可能出现这个提示信息，这取决于优化器执行查询语句的执行计划的进度进展到哪里了。
- Range checked for each record (index map: N) (JSON property: message)：MySQL 没有发现有合适的索引可以使用，但发现一些索引的列值可能包了原表中的查询数据。对于原表中的每个行组合，MySQL 会检查是否可以使用 range 或 index_merge 访问方法来检索行。 这不是很快，但是比执行没有索引的连接查询快。index map: N 从 1 开始，按照表的 SHOW INDEX 显示的顺序。 index map 值 N 是显示候选索引的位掩码值。 例如，值 0x19（二进制 11001）意味着将考虑使用 index map 为 1,4 和 5 的索引
- Scanned N databases (JSON property: message)：这表示在对 INFORMATION_SCHEMA 库下的表的查询时，服务器执行会扫描多少个目录，N 的值可以是 0,1 或 all。

  * 注：mysql 从 5.0 开始提供 information_schema 数据库，information_schema 中存放着 mysql 服务器内部的元数据信

（数据表、数据列、索引等 schema 信息），整个 information_schema 库下的信息只支持查询。因为该数据库中所查询到的信息并不是真正存放在磁盘上的数据，而是每当使用 SQL 访问时，就从 mysql 服务器内部读取并显示出这些元数据来，这样，一次要访问大量的数据表时，就要花费很长的时间。mysql5.1 开始大大提升了访问性能，使用户可以快速访问 information_schema 数据库，通过改善后的查询检索元数据库信息时，查询执行计划的 Extra 列会显示此提示信息，Scanned N databases 中的 N 表示读取几个数据库中的信息，取值为 0，1，all，分别代表如下含义（0：只请求某个特定表的信息，并不需要读取整个库的元数据信息，1：请求特定数据库内的与该数据库相关的所有 schema 信息，all：读取 mysql 服务器内所有数据库的所有 schema 信息。示例：explain select table_name from information_schema.tables where table_schema='employees' and table_name='employees';，执行计划中 Extra 列回出现包含 Scanned 0 databases 的提示信息，因为只读取了一个表的信息，所以只显示 N 为 0。要注意：正常情况下的业务 SQL 几乎不会查询 information_schema 库，所以几乎不会在 Extra 列中出现该提示信息）

- Select tables optimized away (JSON property: message)：使用 max()和 min()函数聚合函数产生的隐式分组(非 group by 语句分组)的查询中，有两种情况可能会出现该提示：1、优化器最多返回一行记录，2、为了产生该返回行的记录，查询优化器使用索引读取一些可以确定隐式分组需要扫描的行范围结果集进而使用这个结果集就可以进行确认最终 max()或 min()需要返回的行记录值。注：只有在优化阶段通过读取索引行来直接读取要返回行记录的查询中，才会出现该提示信息，这样表示在查询真正执行期间不需要再读取任何表直接通过索引就可以返回需要的数据了。举个例子：
  * create table test(id int,id2 int,key i_id(id),key i_id2(id2),key i_id_id2(id,id2));insert into test(id,id2) values(1,2),(2,1),(2,3),(3,2),(10,9),(11,100);

* 以上建表语句中可以看到在 id 和 id2 列上有 i_id 和 i_id2 两个独立索引，id 和 id2 两个列有一个组合索引，那么使用类似语句：explain select max(id) from test;explain select min(id) from test; explain select max(id2) from test;explain select min(id2) from test; explain select min(id),min(id2) from test; explain select max(id),min(id2) from test; explain select max(id),max(id2) from test; explain select min(id),max(id2) from test; 等语句可以直接使用 id 和 id2 两个列上的独立索引直接返回最大值和最小值，不需要扫描表。而对于加 where 条件的（例如把前面理出的 SQL 加上 id 或者 id2 或者 id,id2 的 where 条件之后），如果查询优化器能够使用索引（注意：前提是要能够使用到索引，不能使用到索引的不行）直接进行评估并确定出有满足条件的记录时，那么前面会出现这个提示。但是如果必须要去扫描表而不能直接通过索引返回数据的查询，那就不会出现这个提示。

* 对于 myisam 表，因为 myisam 表有内部行计数器，所以使用 count(*)这种聚合函数的查询中，可以通过内部计数器直接返回查询结果而不需要去扫描表也不需要通过索引来判断，执行计划中也会有这个提示信息出现。但 innodb 表由于没有这种内部计数器，所以需要依赖索引才行。

- Skip_open_table, Open_frm_only, Open_full_table (JSON property: message)：与“Scanned N databases (JSON property: message)”提示信息类似，这些值表示查询可以使用 INFORMATION_SCHEMA 库下的表的查询来直接返回查询结果时，才会出现这个提示信息：

  * Skip_open_table：不 需要扫描表。只通过读取 INFORMATION_SCHEMA 库下的表直接可以返回查询需要的数据

  * Open_frm_only：只有表的.frm 文件需要打开

  * Open_full_table：未能使用到直接查询 INFORMATION_SCHEMA 库下的表优化的查询，必须打开.frm，.MYD 和.MYI 文件

* 注意：以上内容涉及到需要打开.frm、.MYI、.MYD 的情况仅针对 myisam 表，对于 innodb 表不适用

- Start temporary, End temporary (JSON property: message)：这表示半连接查询中使用临时表进行去重的优化策略（Duplicate Weedout 特性，5.7 版本中引入），Duplicate Weedout 这也是优化子查询的另外一种优化方法，使用该优化时会先访问 in subquery 中的子查询，然后与外部表进行连接查询后将得到的结果保存到临时表中，之后再在临时表中进行去重操作（执行连接操作时，先读取的数据表叫做驱动表，驱动表的执行计划中显示：start temporary 提示信息，后读取的数据表叫做被驱动表，被驱动表的执行计划中显示：end temporary 提示信息）。注：Duplicate Weedout 优化过程与连接查询中的 in subquery 查询+group by 去重的过程是一样的
  * 注意：可能需要类似这种 in (subquery in ())格式的才会出现，示例：explain select * from employees e where e.emp_no in (select de.emp_no from dept_emp de where dept_no in('d001','d003'));
  * 另外：在 mysql5.6.35 和 mysql5.7.17 最新版本上测试时，发现并没有使用这个 Duplicate Weedout 特性特性，而是直接使用了 MATERIALIZED 物化子查询（默认情况下 firstmatch 和 materialation 策略开启，会优先使用这两个特性，把 firstmatch 和 materialation 策略关闭之后，再使用上述语句就会出现这个提示信息）。
- unique row not found (JSON property: message)：对于诸如 SELECT ... FROM tbl_name 的查询，两个表连接查询，都使用到了两个表的主键或者唯一索引进行关联查询，并且 where 条件也是唯一索引或者主键列的，但是连接查询的 where 条件值在其中一个表中有记录而另外一个表中没有时，没有对应主键或唯一索引匹配记录的在执行计划输出 Extra 列就可能会显示这个信息（谁出现这个提示信息就看这条记录在哪一边以及使用 left join 还是 right join，如果记录在左边的表，则使用 left join 并且 where 条件是使用右表不存在的记录，则右表的 explain 输出行

显示这个提示信息，如果是使用 inner join，则查询优化器直接判定不可能有结果，出现 Impossible WHERE noticed after reading const tables 提示信息。反之亦然）。表示某表中没有记录满足 UNIQUE 索引或 PRIMARY KEY 的条件。举个例子：
* 建表和造数语句：create table tb_test1(id int unsigned not null primary key,test1 varchar(100));create table tb_test2(id int unsigned not null primary key,test2 varchar(100));insert into tb_test1 values(1,'1'),(2,'2');insert into tb_test2 values(1,'1');
* 查询语句：explain select t1.* from tb_test1 t1 left join tb_test2 t2 on t1.id=t2.id where t1.id=2; explain select t1.id from tb_test1 t1 left join tb_test2 t2 on t1.id=t2.id where t1.id=2;

- Using filesort (JSON property: using_filesort)：MySQL 必须做一个额外的排序。首先根据连接类型遍历所有行，并存储排序 key 和 WHERE 子句匹配的所有行的行指针完成排序。 然后，按照 key 排序，并按排序之后的顺序检索行。举个例子（last_name 列无索引）：explain select * from employees order by last_name; 由于 last_name 列无索引，所以 mysql 先将所有需要排序的数据行读取到排序缓冲区，在排序缓冲区进行排序，然后将排序后的结果发送给客户端。
- Using index (JSON property: using_index)：仅使用索引就可以返回查询数据的查询，不需要进行额外的回表来读取实际数据行时。可以使用此策略。对于有聚簇索引的 InnoDB 表，如果“EXtra”列中没有显示“Using index”，但是 type 是 index，key 是 PRIMARY，表示也可以使用主键索引来直接返回查询数据。有这个提示的查询就叫做覆盖索引查询。
- Using index condition (JSON property: using_index_condition)：通过访问索引元组并首先测试读取表以确定是否需要进行全表扫描还是使用索引条件过滤后再读取表。 举个例子：explain select * from employees where hire_date between '1986-06-26' and '1986-07-26';

- Using index for group-by (JSON property: using_index_for_group_by)：与“Using index”表访问方法类似，Using index for group-by 表示 MySQL 发现了一个索引，可用于来处理 GROUP BY 或 DISTINCT 列，而不需要在处理 GROUP BY 或 DISTINCT 列的时候对实际表进行任何额外的磁盘访问。以最有效的方式使用索引，因此对于每个分组，只读取少数索引条目。如果无法使用索引，那么 mysql 需要先使用分组基准列进行排序，然后将排序结果进行分组，若使用索引（仅限 BTREE 索引）处理 group by，就会按索引已经排序好的顺序依次读取索引列，不需要额外的排序直接进行分组处理即可。即像这样使用索引处理 group by 的查询，处理 group by 的速度相当快且非常高效，执行计划的 Extra 列就会显示 Using index for group-by (JSON property: using_index_for_group_by)信息。 * 处理 group by 时，常见的可以使用到索引进行松散索引扫描处理 group by 的情况：要使用松散索引扫描来处理 group by，有两种情况：第一种情况是，where 条件列和 group by 列是组合索引，且 select 字段可以使用这个组合索引进行覆盖，那么执行计划中会出现这个提示信息(如：explain select emp_no,min(from_date),max(from_date) from salaries where from_date>='2002-05-01' group by emp_no;explain select emp_no,min(from_date),max(from_date) from salaries where emp_no>=12000 group by emp_no;)。第二种情况是，无 where 条件，但是 group by 中的列和 select 中的列可以完全使用索引（组合索引和单列索引都可以）进行覆盖，那么执行计划中会出现这个提示信息(如：explain select emp_no,min(from_date),max(from_date) from salaries group by emp_no;)。注意：如果执行计划分析 where 条件能够返回一个较小的范围，那么就算 select 字段、where 字段和 group by 字段能够被组合索引覆盖，Extra 列也可以不会出现 Using index for group-by (JSON property: using_index_for_group_by)提示信息(如：explain select emp_no from salaries where emp_no between 10001 and

10099 group by emp_no;)

* 处理 group by 时，常见的不能使用到索引进行松散索引扫描处理 group by 的情况：简单的顺序读取索引的方式（紧凑索引扫描）与只读取索引必要的部分的松散索引扫描的方式是不同的。例如：第一种情况：group by 子句的查询中含有 avg()，sum()，count()等聚合函数时，处理查询时需要读取所有的索引列，此时即便可以使用索引来处理 group by 子句，也无法只松散地读取所需的记录，处理这种查询时，虽然可以使用索引处理 group by，但是也不能叫做松散索引扫描，这种查询的执行计划中不会出现 Using index for group-by (JSON property: using_index_for_group_by)提示信息（如：explain select first_name, count() from employees group by first_name;explain select first_name, avg(emp_no) from employees group by first_name;explain select first_name, sum(emp_no) from employees group by first_name;)。第二种情况：group by 子句可以使用到索引，但是 where 条件无法使用索引，那么 mysql 会先使用索引进行 group by，然后读取数据记录处理 where 子句中的比较，处理这种查询时，虽然可以使用索引处理 group by，但是也不能叫做松散索引扫描，这种查询的执行计划中不会出现 Using index for group-by (JSON property: using_index_for_group_by)提示信息(如：explain select first_name from employees where birth_date>'1994-01-01' group by first_name;)

- Using join buffer (Block Nested Loop), Using join buffer (Batched Key Access) (JSON property: using_join_buffer)：早期连接表查询会将驱动表的连接字段读入连接缓冲区，然后在缓冲区中使用驱动表的连接字段值来逐行执行与被驱动表进行匹配。 （Block Nested Loop）表示使用了块嵌套循环算法。（Batched Key Access）表示使用批量健访问算法， 也就是说，在 EXPLAIN 输出的驱动表中的 Key 将被缓冲，使用缓冲区中的 Key 与被驱动表关联字段进行匹配，然后从被驱动表中批量获取行数据。

* explain select * from dept_emp de,employees e where de.from_date>'2005-01-01' and e.emp_no<10904;

- Using MRR (JSON property: message)：表示查询使用到了多范围读优化策略。
  * MRR(multi range read)策略是 mysql5.6 和 mariadb5.3 引入的优化功能，在没有这个特性之前的多列条件查询中，有时候需要先通过索引范围查找符合 where 条件的记录，然后再根据需要读取数据文件的其余记录。通过索引查找到的记录如果相当多时，对相关记录对应的数据文件的读取每次都是采用随机访问方式。引入了 MRR 就是为了解决这个问题，有 MRR 时，先通过索引读取一定量的符合 where 条件的记录，然后使用主键值进行排序，然后使用主键顺序从数据文件中读取行数据（如果在优化器层可以把条件拆分成元组对，则还可以在这一层就把不符合条件的给过滤掉，且还可以按照拆分好的元组对批量进行访问），然后再根据需要继续前面的步骤来查询其余数据，通过这种方式可以有效减少随机读取数据文件的次数。
  * 在 5.6 及其以上的版本中，使用到多范围读的查询在执行计划中显示 Using MRR，在 mariadb 中，使用到了多范围读的查询的执行计划中显示 Rowid-ordered scan 或者 Key-ordered scan 提示信息。对于 MRR 对应的就是 mariadb 中的 Rowid-ordered scan（通过语句 explain select * from employees where first_name>='A' and first_name<'B';可以看到），mysql 的 BKA 对应的就是 mariadb 的 Key-ordered scan，不过，mariadb 还可以使用 hash join 的方式来进行连接查询（通过语句 explain select * from dept_emp de,employees e where e.emp_no=de.emp_no and de.dept_no in('d001','d002')；可以看到 BKAH join）。
  * mariadb 的 Key-ordered scan 不用于无连接的简单 select 查询，而是用于存储引擎（xtradb，innodb）数据表的主键连接查询。上述 explain select * from dept_emp de,employees e where e.emp_no=de.emp_no and de.dept_no in('d001','d002')；语句中，两个表的 emp_no 都是主键列。从

执行计划中来看，先执行 de 表的读取，再执行与 e 表的连接，de 表在使用 emp_no 字段与 e 表连接之前，先对符合 de.dept_no in ('d001','d002')条件的 de 表的 emp_no 列进行排序，然后再读取 e 数据表。由于 e 数据表的 emp_no 列充当了 e 表的记录地址，所以需要在读取 e 表之前先对 e 表的 emp_no 字段进行排序，然后根据排序字段读取 e 表的记录，因此，在 mariadb 的执行计划中，也会看到 BKAH join 提示信息。

* 注意：mysql 中需要修改 optimizer_switch 变量为 optimizer_switch='mrr=on,mrr_cost_based=off,batched_key_access=on'，mariadb 中除了修改这个变量值为 optimizer_switch='mrr=on,mrr_cost_based=off,mrr_sort_keys=on,optimize_join_buffer_size=on'之外，还需要修改 join_cache_level=8，如果 join_cache_level=4 则只能使用到 BKAH，无法使用 Key-ordered scan

- Using sort_union(...), Using union(...), Using intersect(...) (JSON property: message)：这些表示特定的算法，显示如何为 index_merge 连接类型选择合并索引扫描策略。在执行计划的 type 列为 Index_merge 时，mysql 可以使用两个以上的索引执行查询，此时为了进一步说明如何进行索引合并来读取查询结果的。会在执行计划的 Extra 列显示这几个值：

  * Using intersect(...)：使用 and 连接各个使用索引的条件列时，该信息表示从各个处理结果集获取交集

  * Using union(...)：使用 or 连接各个使用索引的条件列时，该信息表示从各个处理结果中获取并集

  * Using sort_union(...)：执行的处理与 Using union 相同，但在使用 or 连接查询的数据量比较大的 range 条件列时，才能使用该方式进行处理，Using sort_union(..)要先读取主键，进行排序合并之后，才能读取记录并返回。

  * Using sort_intersection(...)：在 5.6 中，sort_intersection(...)只能用于等值比较条件，但从 mariadb5.3 开始，使用 between 或 in 等范围比较运算符的条件中也可以使用 sort_intersection，跟 sort_union 类似，也是需要先排序，但是是取交集。

* 注：Using union()和 Using sort_union()都可以在 or 连接查询中使用，但是 Using union()是用于匹配记录数不多的情况，而使用大于，小于等于这类有很多记录的范围条件时，常常使用 Using sort_union()。但是，无论有多少记录，若各 where 条件中使用的比较条件完全相等时，则使用 Using union()，否则使用 Using sort_union()。另外，在 mariadb 内部，Using union()与 Using sort_union()使用的排序算法不同，前者使用的是单路排序算法，后者使用的是双路排序算法。

- Using temporary (JSON property: using_temporary_table)：表示 MySQL 需要创建一个临时表来保存中间结果（临时表可以在内存中创建，也可以在磁盘上创建，若查询执行计划中显示该提示信息，则表示使用了临时表，该提示信息是无法看出来是使用的内存临时表还是磁盘临时表。但可以通过状态变量 Created_tmp_tables 和 Created_tmp_disk_tables 来查看使用了哪种临时表）。 如果查询包含不同列顺序的 GROUP BY 和 ORDER BY 子句，则通常会发生这种情况。举个例子：
  * explain select * from employees group by gender order by min(emp_no)；该查询中由于 group by 与 order by 使用到了不同的数据列，无法同时使用两个索引一个用于排序，一个用于分组。所以需要使用临时 表。其中，group by 无法使用索引时在执行计划的 Extra 列显示 Using temporary，order by 无法使用索引时，执行计划中的 Extra 列显示 Using filesort(注意：5.7 中由于 sql_mode 默认添加了 sql_mode=only_full_group_by，则 group by 字段必须出现在 select 列中，否则会无法执行。使用*号也不行)。
  * 关于临时表的注意事项（并不是只有执行计划中出现了 Using temporary 才表示使用到了临时表）：1、from 子句中的子查询必定使用临时表，也就是常常说的派生表(Derived table)，但其实就是临时表，2、含有 count(distinct column)的查询如果无法使用索引时，也会创建临时表，3、使用 union 或 unionall 的查询总是使用临时表来合并查询结果，4、不能使用索引排序的操作也会使用临时表来做缓冲空间，要排序的记录不断增多时，

最后还可能使用到磁盘临时表，排序缓冲本质上就是临时表，使用到排序缓冲进行排序时，执行计划中显示 Using filesort。

- Using where (JSON property: attached_condition)：WHERE 子句用于限制哪些行与下一个表匹配或按照过滤条件过滤之后发送到客户端。举个例子：

  * explain select * from employees where emp_no between 10001 and 10100 and gender='F'; 该语句的限制条件（检索条件）为 emp_no between 10001 and 10100，gender 为检查条件（过滤条件），虽然满足第一个条件 emp_no 记录显示有 100 条，但是同时满足两个条件的记录实际上只有 37 条。而存储引擎会从磁盘读取 100 条件记录，然后交给 mysql server，在 server 层从 100 条记录中过滤掉不满足条件的 63 条记录。而 Using where 就表示执行了过滤掉符合条件的 63 条记录的动作。

  * 不能使用索引而全表扫描的查询，执行计划中都可能出现 Using where。另外，模糊匹配 like 查询中，通配符不能放在关键字的前面，因为 BTREE 索引结构原因，只能放到关键字的后边，如：'%abc'、'abc'不能使用索引，只有'abc%'、'abc'才能使用索引。
- Using where with pushed condition (JSON property: message)：此项仅适用于 NDB 表。 表示 MySQL 集群正在使用条件下推优化来提高非索引列和常量之间直接比较的效率。 在这种情况下，条件被"下推"到集群的数据节点，并在所有数据节点上同时进行评估。 这样就不需要节点之间通过网络发送不匹配的行，使用这种优化策略的查询与不使用条件下推的查询相比可以加速 5 到 10 倍。
- Zero limit (JSON property: message)：该查询使用了 LIMIT 0 子句，导致不能选择任何行。在 5.7 之前的版本中，这种查询语句的 Extra 列会显示 Impossible WHERE，在 5.7 之后的版本中才显示该提示信息。

2.1.6. explain extended 输出格式详解

- 该小节主要讲解 explain extended 语句输出执行计划之后，使用 show warnings 查看到的查询优化器内部改写过后的 SQL 语句，其中包含了一些注释符号的含义解释
    - extended 扩展关键字在 5.7 中默认开启，后续该关键字可能被移除，目前为了向后兼容，该语法仍然保留，但 show warnings 语句可以查看到一个提示性的警告语句告知这个是一个废弃的功能。
    - extended 扩展功能目前只支持查看 select 语句，对于 DML 和 replace 语句，目前不支持查看相关内部改写的信息
    - 因为 SHOW WARNINGS 显示的语句可能包含特殊标记以提供有关查询重写或优化程序操作的信息，所以该语句可能不是有效的 SQL，并且不会把该改写 SQL 直接用于内部执行。 输出信息还可以包括关于优化器采取的动作的附加非 SQL 解释性注释的 Message 值的行
- 示例：

```
mysql> EXPLAIN
       SELECT t1.a, t1.a IN (SELECT t2.a FROM t2) FROM t1\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: t1
         type: index
possible_keys: NULL
          key: PRIMARY
      key_len: 4
          ref: NULL
         rows: 4
     filtered: 100.00
        Extra: Using index
*************************** 2. row ***************************
           id: 2
```

```
  select_type: SUBQUERY
        table: t2
         type: index
possible_keys: a
          key: a
      key_len: 5
          ref: NULL
         rows: 3
     filtered: 100.00
        Extra: Using index
2 rows in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: /* select#1 */ select `test`.`t1`.`a` AS `a`,
          <in_optimizer>(`test`.`t1`.`a`,`test`.`t1`.`a` in
          ( <materialize> (/* select#2 */ select `test`.`t2`.`a`
          from `test`.`t2` where 1 having 1 ),
          <primary_index_lookup>(`test`.`t1`.`a` in
          <temporary table> on <auto_key>
          where ((`test`.`t1`.`a` = `materialized-subquery`.`a`))))) AS `t1.a
          IN (SELECT t2.a FROM t2)` from `test`.`t1`
1 row in set (0.00 sec)
```

- explain extended 的 show warnings 查看结果相关信息详解

```
<auto_key> : 临时表自动创建的索引
```

<cache>（expr）：表示执行了一次表达式（例如标量子查询），结果值保存在内存中以供以后使用。对于由多个值组成的结果集，可能会创建一个临时表来代替，如果被创建临时表来代替临时保存在内存中的话，那么您看到的就是<temporary table>而不是<cache>（expr）

<exists>（query fragmen）：subquery 谓词将转换为 EXISTS 谓词，并且 subquery 语句将进行转换，以便可以与 EXISTS 谓词一起使用

<in_optimizer>（query fragment）：这是一个没有用户意义的内部优化器对象

<index_lookup>（query fragment）：使用索引查找处理查询片段以查找匹配行

<if>（condition，expr1，expr2）：如果条件为真，则求值为 expr1，否则为 expr2

<is_not_null_test>（expr）：测试验证表达式返回值是否为 NULL

<materialize>（query fragment）：使用了物化子查询实现

`materialized-subquery`.col_name：对内部临时表中列 col_name 的引用，用于保存评估子查询的结果

<primary_index_lookup>（query fragment）：使用主键查找来处理查询片段以查找匹配行

<ref_null_helper>（expr）：这是一个没有用户意义的内部优化器对象

/ * select # N * / select_stmt：SELECT 与具有 id 值为 N 的非 extended EXPLAIN 输出中的行相关联(注意，指的是 explain 输出结果中的行，N 代表行的 id 号，不是数据行)

outer_tables semi join（inner_tables）：半连接操作。 inner_tables 表示嵌套循环中的驱动表

<temporary table>：这表示为缓存中间结果而创建的内部临时表

- 当一些表是 const 或 system 查询类型时，涉及这些表中的列的表达式由优化器提前评估，所以可能执行 extended 之后，show warnings 结果显示不出来相关的内部信息。但是，如果使用 FORMAT = JSON，在输出结果中可以显示一些内部信息，但是一些使用 const 值的 const 表访问类型可能被显示为 ref 访问类型