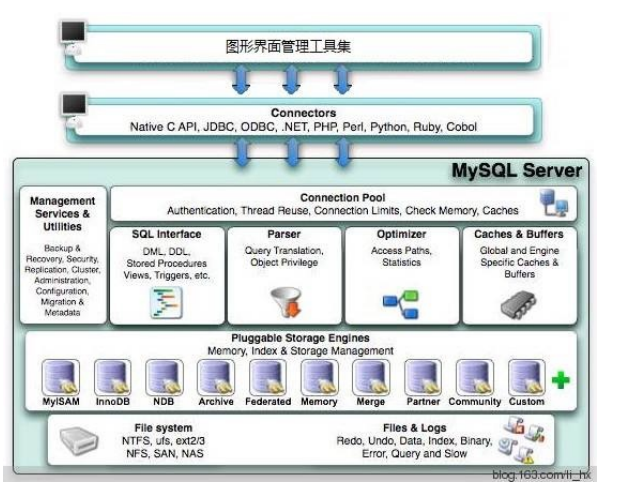
InnoDb行格式、数据页结构以及索引底层原理分析

Mysql架构图



存储引擎负责对表中的数据的进行读取和写入，常用的存储引擎有InnoDB、MylSAM、Memory等，不同的存储引擎有自己的特性，数据在不同存储引擎中存放的格式也是不同的，比如Memory都不用磁盘来存储数据。

在InnoDB中，数据会存储到磁盘上，在真正处理数据时需要先将数据加载到内存，表中读取某些记录时， InnoDB存储引擎不需要一条一条的把记录从磁盘上读出来，InnoDB采取的方式是：将数据划分为若干个页，以页作为磁盘和内存之间交互的基本单位，InnoDB中页的大小一般为16 KB ,也就是说，当需要从磁盘中读数据时 每一次最少将从磁盘中读取16KB的内容到内存中，每一次最少也会把内存中的16KB内容写到磁盘中。

InnoDB数据页结构

页是InnoDB管理存储空间的基本单位，一页的大小默认是16KB。

SHOW GLOBAL STATUS like 'Innodb\_page\_size';

页结构：

File Header

Page Header（数据页）

Infimun + Supremum Records

User Records

Free Space

Page Directory

File Trailer

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** |  | **占用空间** | **简单描述** |
| File Header | 文件头部 | 38字节 | 页的一些通用信息 |
| Page Header | 页面头部 | 56字节 | 数据页专有的一些信息 |
| Infimum + Supremum | 最小记录和最大记录 | 26字节 | 两个虚拟的行记录 |
| User Records | 用户记录 | 不确定 | 实际存储的行记录内容 |
| Free Space | 空闲空间 | 不确定 | 页中尚未使用的空间 |
| Page Directory | 页面目录 | 不确定 | 页中的某些记录的相对位置 |
| File Trailer | 文件尾部 | 8字节 | 校验页是否完整 |

InnoDB行格式

—行记录可以以不同的格式存在InnoDB中，行格式分别是Compact、Redundant、Dynamic和Compressed行格式。

我们可以在创建或修改表的语句中指定行格式：

CREATE TABLE表名(列的信息)ROW\_FORMAT=行格式名称

ALTER TABLE 表名 ROW\_FORMAT=行格式名称

COMPACT行格式

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **变长字段长度列表** | **NULL标志位** | **记录头信息** | **列1数据** | **列2数据** |  |

**记录的额外信息**

这部分信息是服务器为了描述这条记录而不得不额外添加的一些信息，这些额外信息分为3类，分别是：

•变长字段长度列表

•NULL值列表

•记录头信息

**变长字段长度列表**

MySQL支持一些变长的数据类型，比如VARCHAR(M)、VARBINARY(M)、TEXT类型，BLOB类型，这些数据类型修饰列称为变长字段，变长字段中存储多少字节的数据不是固定的，所以我们在存储真实数据的时候需要顺便把这些数据占用的字节数也存起来。在C ompact行格式中，把所有变长字段的真实数据占用的字节长度都存放在记录的开头部位，从而形成一个变长字段长度列表。

I

CHAR是一种固定长度的类型，VARCHAR则是一种可变长度的类型。

VARCHAR(M)，M代表最大能存多少个字符。(MySQL5.0.3以前是字节，以后就是字符)

**NULL值列表**

Compact行格式会把可以为NULL的列统一管理起来，存一个标记为在NULL值列表中，如果表中没有允许存储 NULL的列，则NULL值列表也不存在了。

・二进制位的值为1时，代表该列的值为NULL。

・二进制位的值为0时，代表该列的值不为NULL。

**记录头信息**

除了变长字段长度列表、NULL值列表之外，还有一个用于描述记录的记录头信息，它是由固定的5个字节组成。 5个字节也就是40个二进制位，不同的位代表不同的意思，如图：

|  |  |  |
| --- | --- | --- |
| **名称** | **大小（单**  **位: bit）** | **描述** |
| 预留位1 | 1 | 没有使用 |
| 预留位2 | 1 | 没有使用 |
| delete\_mask | 1 | 标记该记录是否被删除 |
| min\_rec\_mask | 1 | B+树的每层非叶子节点中的最小记录都会添加该标记 |
| n\_owned | 4 | 表示当前记录拥有的记录数 |
| heap\_no | 13 | 表示当前记录在记录堆的位置信息 |
| record\_type | 3 | 表示当前记录的类型，0表示普通记录，1表示B+树非叶子节点 记录， 2表示最小记录， 3表示最大记录 |
| next\_record | 16 | **表示下一条记录的相对位置** |

**记录的真实数据**

记录的真实数据除了我们自己定义的列的数据以外，还会有三个隐藏列：

|  |  |  |  |
| --- | --- | --- | --- |
| **列名** | **是否必须** | **占用空间** | **-IMHt-XJp**  **描述** |
| row\_id | 否 | 6字节 | 行ID，唯一标识一条记录 |
| transaction\_id | 是 | 6字节 | 事务ID |
| roll\_pointer | 是 | 7字节 | 回滚指针 |

实际上这几个列的真正名称其实是：DB\_ROW\_ID、DB\_TRX\_ID、DB\_ROLL\_PTR。

—个表没有手动定义主键，则会选取一个Unique键作为主键，如果连Unique键都没有定义的话，则会为表默 认添加一个名为row\_id的隐藏列作为主键。所以row\_id是在没有自定义主键以及Unique键的情况下才会存在的。

**行溢出数据**

VARCHAR(M)类型的列最多可以占用65535个字节。其中的M代表该类型最多存储的字符数量，如果我们使用 ascii字符集的话，一个字符就代表一个字节，我们看看VARCHAR(65535)是否可用：

mysql> **CREATE TABLE** varchar\_size\_demo(

-> **c** VARCHAR(65535)

-> ) CHARSET=ascii ROW\_FORMAT=Compact;

ERROR 1118 (42000): **Row size** too **large**. The maximum **row size for** the used **table type**, **not** counting BLOBs, **is** 65535. This includes **storage** overhead, **check** the manual. You have **to** change **some** columns **to** TEXT **or** BLOBs mysql>

报错信息表达的意思是：MySQL对一条记录占用的最大存储空间是有限制的，除BLOB或者TEXT类型的列之外， 其他所有的列(不包括隐藏列和记录头信息)占用的字节长度加起来不能超过65535个字节。这个65535个字节除了列本身的数据之外，还包括一些其他的数据，比如说我们为了存储一个VARCHAR(M)类型的列，其实需要占用3部分存储空间：

1. 真实数据
2. 变长字段真实数据的长度
3. NULL值标识

如果该VARCHAR类型的列没有NOT NULL属性，那最多只能存储65532个字节的数据，因为变长字段的长度占用 2个字节，NULL值标识需要占用1个字节。

mysql> **CREATE TABLE** varchar\_size\_demo(

-> **c** VARCHAR(65532)

-> ) CHARSET=ascii ROW\_FORMAT=Compact;

Query OK, 0 **rows** affected (0.02 sec)

**CREATE TABLE** varchar\_size\_demo( **c** VARCHAR(65533) **not null**

) CHARSET=ascii ROW\_FORMAT=Compact; Query OK, 0 **rows** affected (0.02 sec)

**记录中的数据太多产生的溢出**

—个页的大小一般是16KB，也就是16384字节，而一个VARCHAR（M）类型的列就最多可以存储65533个字节，这 样就可能出现一个页存放不了一条记录。

在Compact和Reduntant行格式中，对于占用存储空间非常大的列，在记录的真实数据处只会存储该列的一部分数据，把剩余的数据分散存储在几个其他的页中，然后记录的真实数据处用20个字节存储指向这些页的地址（当然这20个字节中还包括这些分散在其他页面中的数据的占用的字节数），从而可以找到剩余数据所在的页。

Dynamic 和Compressed 行格式

这两种行格式类似于COMPACT行格式，只不过在处理**行溢出数据时有点儿分歧**，它们不会在记录的真实数据处存储—部分数据，而是把所有的数据都存储到其他页面中，只在记录的真实数据处存储其他页面的地址。另外， Compressed行格式会采用压缩算法对页面进行压缩。

索引

...

待定

聚簇索引（B+树）

聚簇索引的特点：

按主键值的大小进行记录和页的排序：

。数据页（叶子节点）里的记录是按照主键值从小到大排序的一个单向链表。

。数据页（叶子节点）之间也是是按照主键值从小到大排序的一个双向链表。

。B+树中同一个层的页目录也是按照主键值从小到大排序的一个双向链表。

B+树的叶子节点存储的是完整的用户记录，就是指这个记录中存储了所有列的值（包括隐藏列）。

具有这两种特性的B+树称为聚簇索引，所有完整的用户记录都存放在这个聚簇索引的叶子节点处。这种聚簇索引并不需要我们在MySQL语句中显式的使用INDEX语句去创建。InnoDB存储引擎会自动的为我们创建聚簇索引。 在InnoDB存储引擎中，聚簇索引就是数据的存储方式（所有的用户记录都存储在了叶子节点），也就是所谓的索引即数据，数据即索引。

二级索引（复制索引）

聚簇索引只能在搜索条件是主键值时才能发挥作用，因为B+树中的数据都是按照主键进行排序的。当我们想以别的列作为搜索条件时我们可以多建几棵B+树，不同的B+树中的数据采用不同的排序规则。

二级索引与聚簇索引有几处不同：

按指定的索引列的值来进行排序。

叶子节点存储的不是完整的用户记录，而只是**索引列+主键**。

目录项记录中不是主键+页号，变成了索引列+页号。

在对二级索引进行查找数据时，需要根据主键值去聚簇索引中再查找一遍完整的用户记录，这个过程叫做**回表**。

联合索引

以多个列的大小为排序规则建立的B+树称为联合索引，本质上也是一个二级索引。

目录项记录的唯一性

我们需要保证在B+树的同一层内节点的目录项记录除页号这个字段以外是唯一的。所以对于二级索引的内节点的目录项记录的内容实际上是由三个部分构成的：

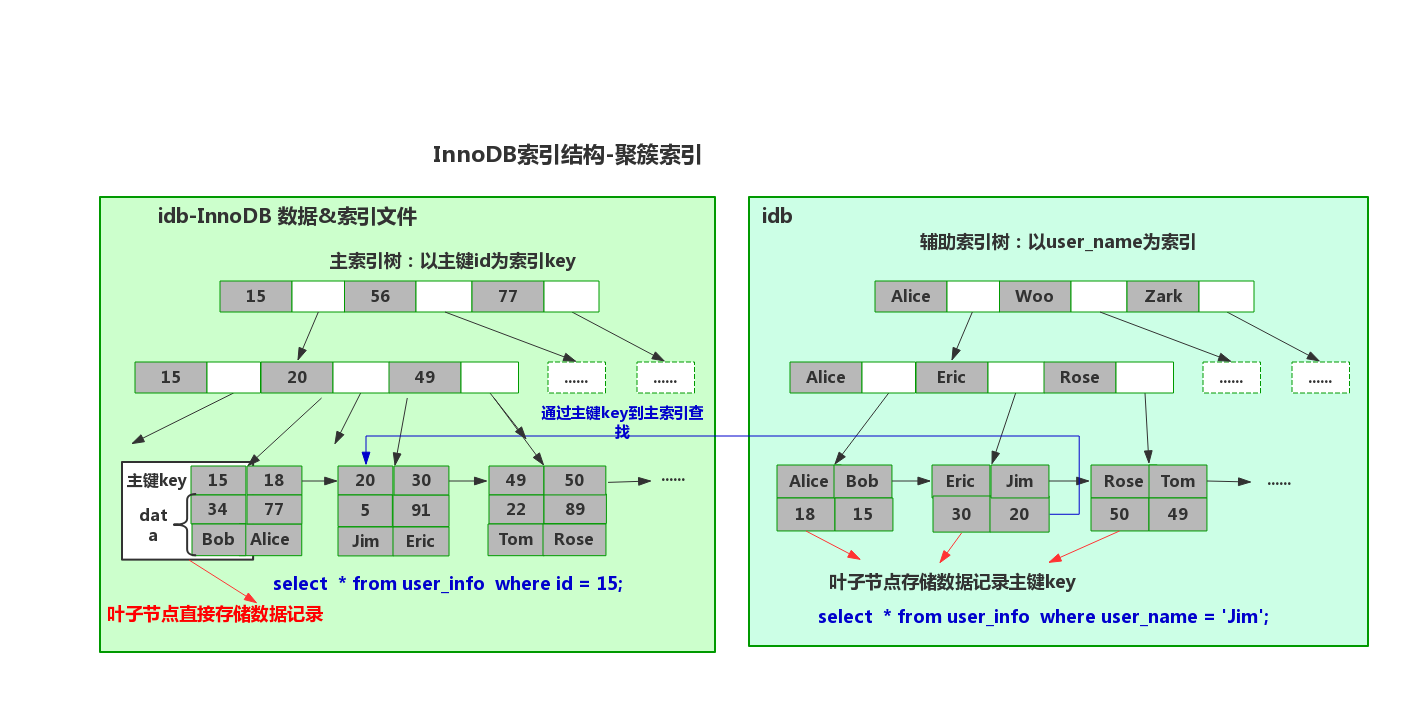
•索引列的值

•主键值

•页号

B+树索引总结

1. 每个索引都对应一棵B+树。用户记录都存储在B+树的叶子节点，所有目录记录都存储在非叶子节点。
2. InnoDB存储引擎会自动为主键（如果没有它会自动帮我们添加）建立聚簇索引，聚簇索引的叶子节点包含完整的用户记录。
3. 可以为指定的列建立二级索引，二级索引的叶子节点包含的用户记录由索引列 + 主键组成，所以如果想通过二级索引来查找完整的用户记录的话，需要通过回表操作，也就是在通过二级索引找到主键值之后再到聚簇索引中查找完整的用户记录。
4. B+树中每层节点都是按照索引列值从小到大的顺序排序而组成了双向链表，而且每个页内的记录（不论是用户记录还是目录项记录）都是按照索引列的值从小到大的顺序而形成了一个单链表。如果是联合索引的话，则页面和记录先按照联合索引前边的列排序，如果该列值相同，再按照联合索引后边的列排序。
5. 通过索引查找记录是从B+树的根节点开始，一层一层向下搜索。由于每个页面都按照索引列的值建立了页目录，所以在这些页面中的查找非常快。



B+树索引实战

索引的代价

空间上的代价

—个索引都为对应一棵B+树，树中每一个节点都是一个数据页，一个页默认会占用16KB的存储空间，所以一个索引也是会占用磁盘空间的。

时间上的代价

索引是对数据的排序，那么当对表中的数据进行增、删、改操作时，都需要去维护修改内容涉及到的B+树索引。 所以在进行增、删、改操作时可能需要额外的时间进行—些记录移动，页面分裂、页面回收等操作来维护好排序。

B+树索引实战

bcd建立联合索引index\_bcd

全值匹配

**select** \* **from** t1 **where** b = 1 **and c** = 1 **and** d = 1;

查询优化器会分析这些查询条件并且按照可以使用的索引中列的顺序来决定先使用哪个查询条件。

匹配左边的列

**select \* from t1 where b = 1;//b\*\***

**select \* from t1 where b = 1 and c = 1;**

下面这个sql是用不到索引的

**select \* from t1 where c = 1;**

因为B+树先是按照b列的值排序的，在b列的值相同的情况下才使用c列进行排序，也就是说b列的值不同的记录中c的值可能是无序的。而现在你跳过b列直接根据c的值去查找，这是做不到的。

匹配列前缀

如果只给出后缀或者中间的某个字符串，比如：

**select \* from t1 where b like '%101%';**

这种是用不到索引的，因为字符串中间有'101'的字符串并没有排好序，所以只能全表扫描了。有时候我们有一些匹配某些字符串后缀的需求，比方说某个表有一个url列，该列中存储了许多url :

[www.baidu.com](http://www.baidu.com)

[www.google.com](http://www.google.com)

[www.qq.com](http://www.qq.com)

假设已经对该url列创建了索引，如果我们想查询以com为后缀的网址的话可以这样写查询条件：WHERE url LIKE '%com'，但是这样的话无法使用该url列的索引。为了在查询时用到这个索引而不至于全表扫描，我们可以把后缀查询改写成前缀查询，不过我们就得把表中的数据全部逆序存储一下，也就是说我们可以这样保存url列中的数据：

moc.udiab.www

moc.elgoog.www

moc.qq.www

这样再查找以com为后缀的网址时搜索条件便可以这么写：WHERE url LIKE 'moc%'，这样就可以用到索引了。

匹配范围值

**select \* from t1 where b > 1 and b < 20000;**

由于B+树中的数据页和记录是先按b列排序的，所以我们上边的查询过程其实是这样的：

・找到b值为1的记录。

・找到b值为20000的记录。

・由于所有记录都是由链表连起来的（记录之间用单链表，数据页之间用双链表），所以他们之间的记录都可以很容易的取出来

・找到这些记录的主键值，再到聚簇索引中回表查找完整的记录。

不过在使用联合进行范围查找的时候需要注意，如果对多个列同时进行范围查找的话，只有对索引最左边的那个列进行范围查找的时候才能用至0B+树索引，比如：

**select \* from t1 where b > 1 and c > 1;**

上边这个查询可以分成两个部分：

1. 通过条件b > 1来对b进行范围，查找的结果可能有多条b直不同的记录，
2. 对这些b直不同的记录继续通过C > 1继续过滤。

这样子对于联合索引来说，只能用到b列的部分，而用不到c列的部分，因为只有b值相同的情况下才能用c列的值进行排序，而这个查询中通过b进行范围查找的记录中可能并不是按照c列进行排序的，所以在搜索条件中继续以 c列进行查找时是用不到这个B+树索引的。

精确匹配某一列并范围匹配另外一列

对于同一个联合索引来说，虽然对多个列都进行范围查找时只能用到最左边那个索引列，但是如果左边的列是精确查找，则右边的列可以进行范围查找，比方说这样：

|  |
| --- |
| **select \* from t1 where b = 1 and c > 1;** |
| 排序 |
| **select \* from t1 order by b, c, d;** |

这个查询的结果集需要先按照b值排序，如果记录的b值相同，则需要按照c来排序，如果c的值相同，则需要按照 d排序。因为这个B+树索引本身就是按照上述规则排好序的，所以直接从索引中提取数据，然后进行回表操作取出该索引中不包含的列就好了。

分组

**select b, c, d, count(\*) from t1 group by b, c, d;**

这个查询语句相当于做了3次分组操作：

1. 先把记录按照b值进行分组，所有b值相同的记录划分为一组。
2. 将每个b值相同的分组里的记录再按照c的值进行分组，将title值相同的记录放到一个分组里。
3. 再将上一步中产生的分组按照d的值分成更小的分组。

如果没有索引的话，这个分组过程全部需要在内存里实现，而如果有索引的话，正好这个分组顺序又和B+树中的索引列的顺序是一致的，所以可以直接使用B+树索引进行分组。

使用联合索引进行排序或分组的注意事项

对于联合索引有个问题需要注意，ORDER By的子句后边的列的顺序也必须按照索引列的顺序给出，如果给出 order by c, b, d的顺序，那也是用不了B+树索引的。

同理，order by b, order by b, c这种匹配索引左边的列的形式可以使用部分的B+树索引。当联合索引左边列的值为常量，也可以使用后边的列进行排序，比如这样：

**select \* from t1 where b = 1 order by c, d;**

这个查询能使用联合索引进行排序是因为b列的值相同的记录是按照c, d排序的。

不可以使用索引进行排序或分组的几种情况

ASC、DESC 混用

对于使用联合索引进行排序的场景，我们要求各个排序列的排序顺序是一致的，也就是要么各个列都是ASC规则排序，要么都是DESC规则排序。

| ORDER BY子句后的列如果不加ASC或者DESC默认是按照ASC排序规则排序的，也就是升序排序的。

**select** \* **from** t1 **order by** b **ASC**, **c DESC**;

这个查询是用不到索引的。

如何建立索引

考虑索引选择性

索引的选择性（Selectivity ），是指不重复的索引值（也叫基数，Cardinality ）与表记录数的比值：

选择性=基数/记录数

选择性的取值范围为（0, 1］，选择性越高的索引价值越大。如果选择性等于1，就代表这个列的不重复值和表记录数是一样的，那么对这个列建立索引是非常合适的，如果选择性非常小，那么就代表这个列的重复值是很多的，不适合建立索引。

考虑前缀索引 用列的前缀代替整个列作为索引key ,当前缀长度合适时，可以做到既使得前缀索引的选择性接近全列索引，同时因为索引key变短而减少了索引文件的大小和维护开销。

I

使用mysql官网提供的示例数据库：<https://dev.mysql.com/doc/employee/en/employees-installation.html> github地址：<https://github.com/datacharmer/test_db.git>

employees表只有一个索引，那么如果我们想按名字搜索一个人，就只能全表扫描了：

**EXPLAIN SELECT** \* **FROM** employees.employees **WHERE** first\_name='Eric' **AND** last\_name='Anido';

那么可以对或建立索引，看下两个索引的选择性：

**SELECT count**(**DISTINCT**(first\_name))/**count**(\*) **AS** Selectivity **FROM** employees.employees; - - 0.0042

**SELECT count**(**DISTINCT**(concat(first\_name, last\_name)))/**count**(\*) **AS** Selectivity **FROM** employees.employees; -- 0.9313

first\_name显然选择性太低，first\_name, last\_name选择性很好，但是first\_name和last\_name加起来长度为30，有没有兼顾长度和选择性的办法？ 可以考虑用first\_name和last\_name的前几个字符建立索引，例如，看看其选择性：

**SELECT count**(**DISTINCT**(concat(first\_name, **left**(last\_name, 3))))/**count**(\*) **AS** Selectivity **FROM** employees.employees; -- 0.7879

选择性还不错，但离0.9313还是有点距离，那么把last\_name前缀加到4 :

**SELECT count**(**DISTINCT**(concat(first\_name, **left**(last\_name, 4))))/**count**(\*) **AS** Selectivity **FROM** employees.employees; -- 0.9007

这时选择性已经很理想了，而这个索引的长度只有18，比短了接近一半，建立前缀索引的方式为：

**ALTER TABLE** employees ・ employees **ADD INDEX** 'first\_name\_last\_name4' (first\_name, last\_name(4));

前缀索引兼顾索引大小和查询速度，但是其缺点是不能用于ORDER BY和GROUP BY操作，也不能用于覆盖索引。

总结

・索引列的类型尽量小

・利用索引字符串值的前缀

・主键自增（排序）

・定位并删除表中的重复和冗余索引

・尽量使用覆盖索引进行查询，避免回表带来的性能损耗。

**Explain**关键字详解、查询优化原理分析与实战

对于一个SQL语句，查询优化器先看是不是能转换成JOIN，再将JOIN进行优化

优化分为：1. 条件优化， 2.计算全表扫描成本， 3. 找出所有能用到的索引， 4. 针对每个索引计算不同的访问方式的成本， 5. 选出成本最小的索引以及访问方式

开启查询优化器日志

--开启

**set** optimizer\_trace="enabled=on";

\_\_执行sql

—查看日志信息

**select** \* **from** information\_schema.OPTIMIZER\_TRACE;

\_\_关闭

**set** optimizer\_trace="enabled=off";

常量传递(constant\_propagation)

a = 1 **AND** b > a

上面这个sql可以转换为：

|  |  |
| --- | --- |
| a = | 1 **AND** b > 1 |

等值传递(equality\_propagation)

|  |  |
| --- | --- |
| a = | b **and** b = **c and c** = 5 |

上面这个sql可以转换为：

|  |  |
| --- | --- |
| a = | 5 **and** b = 5 **and c** = 5 |

移除没用的条件(trivial\_condition\_rejoval)

|  |  |
| --- | --- |
| a = | 1 **AND** 1 = 1 |

上面这个sql可以转换为：

|  |  |
| --- | --- |
| a = | 1 |

基于成本

一个查询可以有不同的执行方案，可以选择某个索引进行查询，也可以选择全表扫描，查询优化器会选择其中成本最低的方案去执行查询。

I/O成本

InnoDB存储引擎都是将数据和索引都存储到磁盘上的，当我们想查询表中的记录时，需要先把数据或者索引加载到内存中然后再操作。这个从磁盘到内存这个加载的过程损耗的时间称之为I/O成本。

CPU成本

读取以及检测记录是否满足对应的搜索条件、对结果集进行排序等这些操作损耗的时间称之为CPU成本。

InnoDB存储引擎规定读取一个页面花费的成本默认是1.0，读取以及检测一条记录是否符合搜索条件的成本默认是0.2。

基于成本的优化步骤

在一条单表查询语句真正执行之前，MySQL的查询优化器会找出执行该语句所有可能使用的方案，对比之后找出成本最低的方案，这个成本最低的方案就是所谓的执行计划，之后才会调用存储引擎提供的接口真正的执行查询。

下边我们就以一个实例来分析一下这些步骤，单表查询语句如下：

**select** \* **from** employees.titles **where** emp\_no > '10101' **and** emp\_no < '20000' **and** to\_date = '1991-10-10';

1•根据搜索条件，找出所有可能使用的索引

* emp\_no > '10101'，这个搜索条件可以使用主键索引PRIMARY。
* to\_date=’1991-10-10'，这个搜索条件可以使用二级索引idx\_titles\_to\_date。

**2**•**计算全表扫描的代价**

对于InnoDB存储引擎来说，全表扫描的意思就是把聚簇索引中的记录都依次和给定的搜索条件做一下比较，把符合搜索条件的记录加入到结果集，所以需要将聚簇索引对应的页面加载到内存中，然后再检测记录是否符合搜索条件。由于查询成本=I/0成本+CPU成本，所以计算全表扫描的代价需要两个信息：

1. 聚簇索引占用的页面数
2. 该表中的记录数

MySQL为每个表维护了一系列的统计信息，show table status语句来查看表的统计信息。

**SHOW TABLE** STATUS **LIKE** 'titles';

Rows

表示表中的记录条数。对于使用MyISAM存储引擎的表来说，该值是准确的，对于使用InnoDB存储引擎的表来说，该值是一个估计值。

Data\_length

表示表占用的存储空间字节数。使用MyISAM存储引擎的表来说，该值就是数据文件的大小，对于使用InnoDB存储引擎的表来说，该值就相当于聚簇索引占用的存储空间大小，也就是说可以这样计算该值的大小：

Data\_length = 聚簇索引的页面数量x每个页面的大小

我们的titles使用默认16KB的页面大小，而上边查询结果显示Data\_length的值是20512768，所以我们可以反向来推导出聚簇索引的页面数量：

聚簇索引的页面数量=Data\_length/16/1024 = 20512768/16/1024 = 1252

我们现在已经得到了聚簇索引占用的页面数量以及该表记录数的估计值，所以就可以计算全表扫描成本了。但是 MySQL在真实计算成本时会进行一些微调。

I/O成本：1252\*1 = 1252。1252指的是聚簇索引占用的页面数，1.0指的是加载一个页面的成本常数。

CPU成本：442070\*0.2=88414。442070指的是统计数据中表的记录数，对于InnoDB存储引擎来说是一个估计值， 0.2指的是访问一条记录所需的成本常数

总成本： 1252+88414 = 89666。

综上所述，对于titles的全表扫描所需的总成本就是89666。

我们前边说过表中的记录其实都存储在聚簇索引对应B+树的叶子节点中，所以只要我们通过根节点获得了最左边的叶子节点，就可以沿着叶子节点组成的双向链表把所有记录都查看一遍。也就是说全表扫描这个过程其实有的B+树内节点是不需要访问的，但是M ySQL在计算全表扫描成本时直接使用聚簇索引占用的页面数作为计算I/O成本的依据，是不区分内节点和叶子节点的。

**3**•**计算PRIMARY需要成本**

计算PRIMARY需要多少成本的关键问题是：需要预估出根据对应的where条件在主键索引B+树中存在多少条符合条件的记录。

范围区间数

当我们从索引中查询记录时，不管是=、n >、<这些操作都需要从索引中确定一个范围，不论这个范围区间的索引到底占用了多少页面，查询优化器粗暴的认为读取索引的一个范围区间的I/O成本和读取一个页面是相同的。

本例中使用PRIMARY的范围区间只有一个：(10101,20000)，所以相当于访问这个范围区间的索引付出的I/O成本就是：

1 x 1.0 = 1.0

预估范围内的记录数

优化器需要计算索引的某个范围区间到底包含多少条记录，对于本例来说就是要计算PRIMARY在(10101,20000) 这个范围区间中包含多少条数据记录，计算过程是这样的：

•步骤1 :先根据emp\_no > 10101这个条件访问一下PRIMARY对应的B+树索引，找到满足emp\_no

> 10101这个条件的第一条记录，我们把这条记录称之为区间最左记录。

**・步骤2 :然后再根据emp\_no < 20000这个条件继续从PRIMARY对应的B+树索引中找出第一条满足 这个条件的记录，我们把这条记录称之为区间最右记录。**

**・步骤3 :如果区间最左记录和区间最右记录相隔不太远(只要相隔不大于10个页面即可)，那就可以精确统计出满足emp\_no > '10101' and emp\_no < '20000'条件的记录条数。**否则只沿着区间最左记录向右读10个页面，计算平均每个页面中包含多少记录，然后用这个平均值乘以区间最左记录和区间最右记录之间的页面数量就可以了。那么问题又来了，怎么估计区间最左记录和区间最右记录之间有多少个页面呢？计算它们父节点中对应的目录项记录之间隔着几条记录就可以了。

根据上面的步骤可以算出来PRIMARY索引的记录条数，所以读取记录的CPU成本为：26808\*0.2=5361.6，其中 26808是预估的需要读取的数据记录条数， 0.2是读取一条记录成本常数。

PRIMARY的总成本

确定访问的IO成本+过滤数据的CPU成本=1+5361.6=5362.6

1. 计算idx\_titles\_to\_date需要成本

因为通过二级索引查询需要回表，所以在计算二级索引需要成本时还要加上回表的成本，而回表的成本就相当于

下面这个SQL执行：

**select** \* **from** employees.titles **where** 主键字段 **in** （主键值 1,主键值2,。。。，主键值3）;

所以idx\_titles\_to\_date的成本=辅助索引的查询成本+回表查询的成本

5.比较各成本选出最优者

选择成本最小的索引

基于索引统计数据的成本计算

有时候使用索引执行查询时会有许多单点区间，比如使用IN语句就很容易产生非常多的单点区间，比如下边这个查询：

**select** \* **from** employees.titles **where** to\_date **in** （'a','b','c','d', ..., 'e'）;

很显然，这个查询可能使用到的索引就是idx\_titles\_to\_date，由于这个索引并不是唯一二级索引，所以并不能确 定一个单点区间对应的二级索引记录的条数有多少，需要我们去计算。计算方式我们上边已经介绍过了，就是先获取索引对应的B+树的区间最左记录和区间最右记录，然后再计算这两条记录之间有多少记录（记录条数少的时候可以做到精确计算，多的时候只能估算）。这种通过直接访问索引对应的B+树来计算某个范围区间对应的索引记录条数的方式称之为index dive。

如果只有几个单点区间的话，使用index dive的方式去计算这些单点区间对应的记录数也不是什么问题，可是如 果很多呢，比如有20000次，MySQL的查询优化器为了计算这些单点区间对应的索引记录条数，要进行20000次 index dive操作，那么这种情况下是很耗性能的，所以MySQL提供了一 系统变量eq\_range\_index\_dive\_limit， 我们看一下这个系统变量的默认值：SHOW VARIABLES LIKE '%dive%';为200。

也就是说如果我们的IN语句中的参数个数小于200个的话，将使用index dive的方式计算各个单点区间对应的记录 条数，如果大于或等于200个的话，可就不能使用index dive 了，要使用所谓的索引统计数据来进行估算。像会为 每个表维护一份统计数据一样，MySQL也会为表中的每一个索引维护一份统计数据，查看某个表中索引的统计数据可以使用SHOW INDEX FROM表名的语法。

Cardinality属性表示索引列中不重复值的个数。比如对于一个一万行记录的表来说，某个索引列的Cardinality属性是10000 ,那意味着该列中没有重复的值，如果Cardinality属性是1的话，就意味着该列的值全部是重复的。不过需要注意的是，对于InnoDB存储引擎来说，使用SHOW INDEX语句展示出来的某个索引列的Cardinality属性是—个估计值，并不是精确的。可以根据这个属性来估算IN语句中的参数所对应的记录数：

•使用SHOW TABLE STATUS展示出的Rows值，也就是一个表中有多少条记录。

•使用SHOW INDEX语句展示出的Cardinality属性。

•根据上面两个值可以算出idx\_key1索引对于的key1列平均单个值的重复次数：Rows/Cardinality

•所以总共需要回表的记录数就是：IN语句中的参数个数\*Rows/Cardinality

NULL值处理

上面知道在统计列不重复值的时候，会影响到查询优化器。

对于NULL，有三种理解方式：

1. NULL值代表一个未确定的值，每一个NULL值都是独一无二的，在统计列不重复值的时候应该都当作独立的。
2. NULL值在业务上就是代表没有，所有的NULL值代表的意义是一样的，所以所有的NULL值都一样，在统计列不重复值的时候应该只算—个。
3. NULL完全没有意义，在统计列不重复值的时候应该忽略NULL。

innodb提供了一个系统变量：

**show global** variables **like** '%innodb\_stats\_method%';

这个变量有三个值：

1. nulls\_equal :认为所有NULL值都是相等的。这个值也是innodb\_stats\_method的默认值。如果某个索引列中NULL值特别多的话，这种统计方式会让优化器认为某个列中平均一个值重复次数特别多，所以倾向于不使用索引进行访问。
2. nulls\_unequal :认为所有NULL值都是不相等的。如果某个索引列中NULL值特别多的话，这种统计方式 会让优化器认为某个列中平均一个值重复次数特别少，所以倾向于使用索引进行访问。
3. nulls\_ignored :直接把NULL值忽略掉。

|最好不在索引列中存放NULL值才是正解

统计数据

InnoDB提供了两种存储统计数据的方式：

・统计数据存储在磁盘上。

・统计数据存储在内存中，当服务器关闭时这些这些统计数据就都被清除掉了。

MySQL给我们提供了系统变量innodb\_stats\_persistent来控制到底采用哪种方式去存储统计数据。在MySQL 566之前，innodb\_stats\_persistent的值默认是OFF，也就是说InnoDB的统计数据默认是存储到内存的，之后的版本中innodb\_stats\_persistent的值默认是ON，也就是统计数据默认被存储到磁盘中。

不过InnoDB默认是以表为单位来收集和存储统计数据的，也就是说我们可以把某些表的统计数据（以及该表的索引统计数据）存储在磁盘上，把另一些表的统计数据存储在内存中。我们可以在创建和修改表的时候通过指定 **STATS\_PERSISTENT**属性来指明该表的统计数据存储方式。

**基于磁盘的永久性统计数据**

当我们选择把某个表以及该表索引的统计数据存放到磁盘上时，实际上是把这些统计数据存储到了两个表里：

innodb\_table\_stats存储了关于表的统计数据，每一条记录对应着一个表的统计数据

innodb\_index\_stats存储了关于索引的统计数据，每一条记录对应着一个索引的一个统计项的统计数据

定期更新统计数据

• 系统变量**innodb\_stats\_auto\_recalc**决定着服务器是否自动重新计算统计数据，它的默认值是ON， 也就是该功能默认是开启的。每个表都维护了一个变量，该变量记录着对该表进行增删改的记录条数，如果发生变动的记录数量超过了表大小的10%，并且自动重新计算统计数据的功能是打开的，那么服务器会重新进行一次统计数据的计算，并且更新innodb\_table\_stat和innodb\_index\_stats表。不过自动重新计算统计数据的过程是异步发生的，也就是即使表中变动的记录数超过了10%，自动重新计算统计数据也不会立即发生，可能会延迟几秒才会进行计算。

• 如果innodb\_stats\_auto\_recalc系统变量的值为OFF的话，我们也可以手动调用ANALYZE TABLE语句来重新计算统计数据。

ANALYZE TABLE single\_table;

**控制执行计划**

Index Hints（索引限制）

・USE INDEX :限制索引的使用范围，我们在数据表里建立了很多索引，当MySQL对索引进行选择时，这些索引都在考虑的范围内。但有时我们希望M ySQL只考虑几个索引，而不是全部的索引， 这就需要用到USE INDEX对查询语句进行设置。

・IGNORE INDEX :限制不使用索引的范围

・FORCE INDEX :我们希望MySQL必须要使用某一个索引(由于MySQL在查询时只能使用一个索索引,因此只能强迫MySQL使用一个索引)。这就需要使用FORCE INDEX来完成这个功能。

基本语法格式:

**SELECT** \* **FROM** table1 USE|**IGNORE**|**FORCE INDEX** (col1\_index,col2\_index) **WHERE** col1=1 **AND** col2=2 **AND** col3=3;

关于**JOIN**的优化

内连接

以下三种写法都是内连接:

mysql> **select** \* **from** t1 **join** t2 **on** t1.a = t2.a;

mysql> **select** \* **from** t1 **inner join** t2 **on** t1.a = t2.a;

mysql> **select** \* **from** t1 **cross join** t2 **on** t1.a = t2.a;

左连接、右连接

左连接:

|  |
| --- |
| mysql> **select** \* **from** t1 **left join** t2 **on** t1.a = t2.a; |
| 右连接: |
| mysql> **select** \* **from** t1 **right join** t2 **on** t1.a = t2.a; |

连接的原理

不管是内连接还是左右连接,都需要一个驱动表和一个被驱动表,对于内连接来说,选取哪个表为驱动表都没关系,而外连接的驱动表是固定的,也就是说左连接的驱动表就是左边的那个表,右连接的驱动表就是右边的那个表。

连接的大致原理是:

1. 选取驱动表,使用与驱动表相关的过滤条件,选取代价最低的访问形式来执行对驱动表的单表查询。
2. 对上一步骤中查询驱动表得到的结果集中每一条记录,都分别到被驱动表中查找匹配的记录。

对应伪代码就是:

**for** each row in t1 { //此处表示遍历满足对t1单表查询结果集中的每一条记录

**for** each row in t2 { //此处表示对于某条t1表的记录来说，遍历满足对t2单表查询结果集中的每一条记录

//判断是否符合join条件

}

}

嵌套循环连接(Nested-Loop Join)

上面的过程就像是一个嵌套的循环,所以这种驱动表只访问一次,但被驱动表却可能被多次访问,访问次数取决于对驱动表执行单表查询后的结果集中的记录条数的连接执行方式称之为嵌套循环连接(Nested-Loop Join)，这是最简单,也是最笨拙的一种连接查询算法。

比如对于下面这个sql:

mysql> **select** \* **from** t1 **join** t2 **on** t1.a = t2.a **where** t1.b **in** (1,2);

先会执行：

mysql> **select** \* **from** t1 **where** t1.b **in** (1,2);

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| + | ---+------ | +------ | +------ | +------ | + |
| | a | b | | | **c** | | d | | e | | |
| + | ---+------ | +------ | +------ | +------ | + |
| | 1 | | | 1 | | 1 | | 1 | a | | |
| | 2 | | | 2 | | 2 | | 2 | b | | |
| | 5 | | | 2 | | 3 | | 5 | e | | |
| + | ---+------ | +------ | +------ | +------ | + |
| 3 | **rows in set** (0.00sec) | | | |  |

得到三条记录。

然后分别执行：

mysql> **select** \* **from** t2 **where** t2.a = 1;

mysql> **select** \* **from** t2 **where** t2.a = 2;

mysql> **select** \* **from** t2 **where** t2.a = 5;

所以实际上对于上面的步骤，实际上都是针对单表的查询，所以都可以使用索引来帮助查询。

基于块的嵌套循环连接(Block Nested-Loop Join)

扫描一个表的过程其实是先把这个表从磁盘上加载到内存中，然后从内存中比较匹配条件是否满足。现实生活中的表可不像t1、t2这种只有几条记录，可能会有成千上万的数据。内存里可能并不能完全存放的下表中所有的记录，所以在扫描表前边记录的时候后边的记录可能还在磁盘上，等扫描到后边记录的时候可能内存不足，所以需要把前边的记录从内存中释放掉。我们前边又说过，采用**嵌套循环连接算法**的两表连接过程中，被驱动表可是要被访问好多次的，如果这个被驱动表中的数据特别多而且不能使用索引进行访问，那就相当于要从磁盘上读好几次这个表，这个I/O代价就非常大了，所以我们得想办法：**尽量减少访问被驱动表的次数。**

当被驱动表中的数据非常多时，每次访问被驱动表，被驱动表的记录会被加载到内存中，在内存中的每一条记录只会和驱动表结果集的一条记录做匹配，之后就会被从内存中清除掉。然后再从驱动表结果集中拿出另一条记录，再一次把被驱动表的记录加载到内存中一遍，周而复始，驱动表结果集中有多少条记录，就得把被驱动表从 磁盘上加载到内存中多少次。所以我们可不可以在把被驱动表的记录加载到内存的时候，一次性和多条驱动表中的记录做匹配，这样就可以大大减少重复从磁盘上加载被驱动表的代价了。

Mysql中有一叫做**join buffer**的概念，join buffer就是执行连接查询前申请的一块固定大小的内存，先把若干条驱动表结果集中的记录装在这个join buffer中，然后开始扫描被驱动表，每一条被驱动表的记录一次性和join buffer中的多条驱动表记录做匹配，因为匹配的过程都是在内存中完成的，所以这样可以显著减少被驱动表的I/O 代价。

最好的情况是join buffer足够大，能容纳驱动表结果集中的所有记录，这样只需要访问一次被驱动表就可以完成连接操作了。这种加入了join buffer的嵌套循环连接算法称之为基于块的嵌套连接(**Block Nested-Loop Join** )算法。

这个join buffer的大小是可以通过**启动参数或者系统变量**join\_buffer\_size进行配置，默认大小为262144字节(也 就是256KB )，最小可以设置为128字节。当然，对于优化被驱动表的查询来说，最好是为被驱动表加上效率高的索引，如果实在不能使用索引，并且自己的机器的内存也比较大可以尝试调大**join\_buffer\_size**的值来对连接查询进行优化。

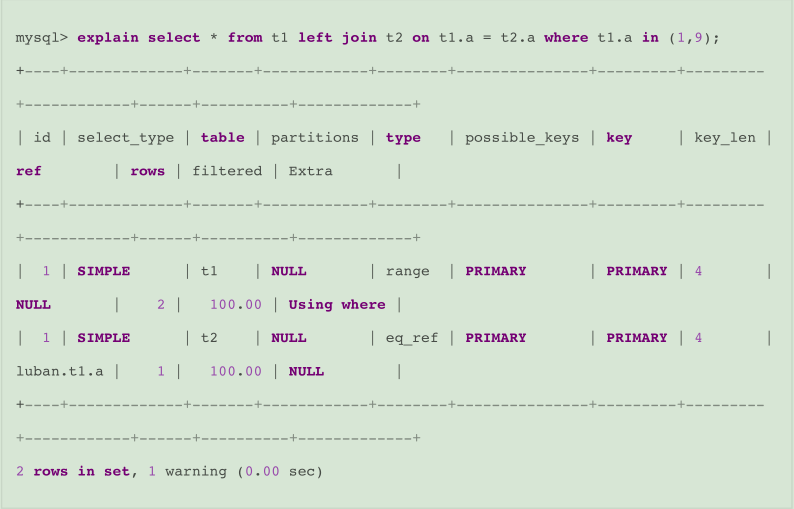
另外需要注意的是，驱动表的记录并不是所有列都会被放到join buffer中，只有查询列表中的列和过滤条件中的列才会被放到join buffer中，所以再次提醒我们，最好不要把\*作为查询列表，只需要把我们关心的列放到查询列表就好了，这样可以在join buffer中放置更多的记录。

外连接消除

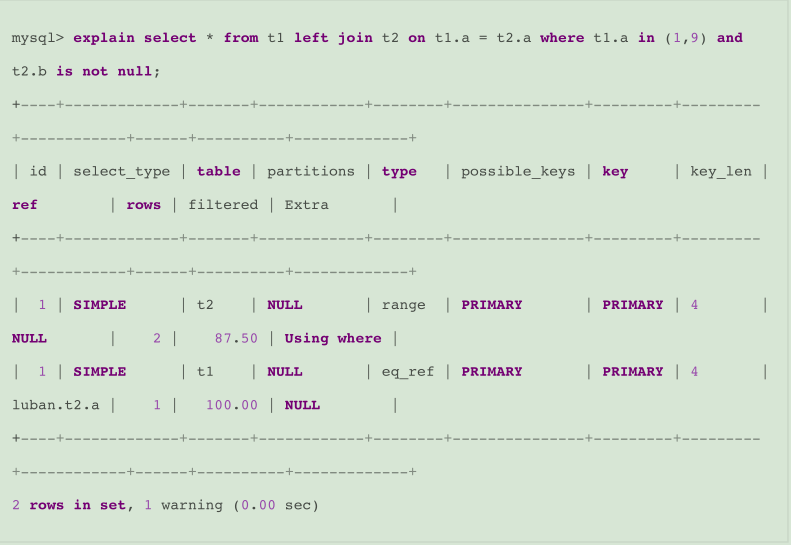
内连接的驱动表和被驱动表的位置可以相互转换，而左连接和右连接的驱动表和被驱动表是固定的。这就导致内连接可能通过优化表的连接顺序来降低整体的查询成本，而外连接却无法优化表的连接顺序。

外连接和内连接的本质区别就是：**对于外连接的驱动表的记录来说”如果无法在被驱动表中找到匹配ON子句中的过滤条件的记录，那么该记录仍然会被加入到结果集中，对应的被驱动表记录的各个字段使用 NULL值填充；而内连接的驱动表的记录如果无法在被驱动表中找到匹配ON子句中的过滤条件的记录” 那么该记录会被舍弃.**

比如下面这个sql :



这个左连接的驱动表和被驱动表是固定好了的，而如果将上面的sql语句改成：



我们可以发现驱动表和被驱动表发生了变化，实际上加上了 is not nul l之后被优化成了内连接，就可以利用查询优化器选择最优的连接顺序了。

关于子查询的优化

下面这些sql都含有子查询：

mysql> **select** \* **from** t1 **where** a **in** (**select** a **from** t2);

mysql> **select** \* **from** (**select** \* **from** t1) **as** t;

**按返回的结果集区分子查询**

标量子查询

那些只返回一个单一值的子查询称之为标量子查询。比如:

mysql> **select** \* **from** t1 **where** a **in** (**select max**(a) **from** t2);

行子查询

返回一条记录的子查询，不过这条记录需要包含多个列。比如：

mysql> **select** \* **from** t1 **where** (a, b) = (**select** a, b **from** t2 **limit** 1);

列子查询

返回一个列的数据的子查询，包含多条记录。比如:

mysql> **select** \* **from** t1 **where** a **in** (**select** a **from** t2);

表子查询

子查询的结果既包含很多条记录，又包含很多个列。比如：

mysql> **select** \* **from** t1 **where** (a, b) **in** (**select** a,b **from** t2);

按与外层查询关系来区分子查询

相关子查询

如果子查询的执行需要依赖于外层查询的值，我们就可以把这个子查询称之为相关子查询。比如：

mysql> **select** \* **from** t1 **where** a **in** (**select** a **from** t2 **where** t1.a = t2.a);

不相关子查询

如果子查询可以单独运行出结果，而不依赖于外层查询的值，我们就可以把这个子查询称之为不相关子查询。前边介绍的那些子查询全部都可以看作不相关子查询。

子查询在MySQL中是怎么执行的

对于不相关标量子查询或者行子查询

比如：

mysql> **select** \* **from** t1 **where** a = (**select** a **from** t2 **limit** 1);

它的执行步骤是:

1. 执行 select a from t2 limit 1 这个子查询。
2. 然后在将上一步子查询得到的结果当作外层查询的参数再执行外层查询select \* from t1 where a =…；

对于相关标量子查询或者行子查询

比如:

mysql> **select** \* **from** t1 **where** b = (**select** b **from** t2 **where** t1.a = t2.a **limit** 1);

它的执行步骤是

1. 先从外层查询中获取一条记录，本例中也就是先从t1表中获取一条记录。
2. 然后从上一步骤中获取的那条记录中找出子查询中涉及到的值，本例中就是从t1表中获取的那条记录中找出t1.a列的值，然后执行子查询。
3. 最后根据子查询的查询结果来检测外层查询WHERE子句的条件是否成立，如果成立，就把外层查询的那条记录加入到结果集，否则就丢弃。
4. 再次执行第一步，获取第二条外层查询中的记录，依次类推。。。

**IN**子查询优化

mysql对IN子查询进行了优化。

比如:

mysql> **select** \* **from** t1 **where** a **in** (**select** a **from** t2);

对于不相关的IN子查询来说，如果子查询的结果集中的记录条数很少，那么把子查询和外层查询分别看成两个单独的单表查询效率还是蛮高的，但是如果单独执行子查询后的结果集太多的话，就会导致这些问题:

•结果集太多，可能内存中都放不下

•对于外层查询来说，如果子查询的结果集太多，那就意味着IN子句中的参数特别多，这会导致无法有效的使用索引，只能对外层查询进行全表扫描。在对外层查询执行全表扫描时，由于IN子句中的参数太多，这会导致检测一条记录是否符合和IN子句中的参数匹配花费的时间太长。 在mysql中，不直接将不相关子查询的结果集当作外层查询的参数，而是将该结果集写入一个**临时表**里。写入临时表的过程是这样的：

1、该临时表的列就是子查询结果集中的列。

2、**写入临时表的记录会被去重**。IN语句是判断某个操作数在不在某个集合中，集合中的值重不重复对整IN语句的结果并不影响，所以我们在将结果集写入临时表时对记录进行去重可以让临时表变得更小。临时表也是个表，只要为表中记录的所有列建立主键或者唯一索引就可以进行去重。

3、一般情况下子查询结果集不会特别大，所以会为它建立基于内存的使用**Memory**存储引擎的临时表，而且会为该表建立哈希索引。IN语句的本质就是判断某个操作数在不在某个集合里，如果集合中的数据建立了哈希索引，那么这个匹配的过程就是很快的。

•如果子查询的结果集非常大，超过了系统变量tmp\_table\_size或者max\_heap\_table\_size，临时表会转而使用基于磁盘的存储引擎来保存结果集中的记录，索引类型也对应转变为B+树索引。

**这个将子查询结果集中的记录保存到临时表的过程称之为物化(Materialize )。**那个存储子查询结果集的临时表称之为物化表。正因为物化表中的记录都建立了索引(基于内存的物化表有哈希索引，基于磁盘的B+树 索引)，通过索引执行IN语句判断某个操作数在不在子查询结果集中变得非常快，从而提升了子查询语句的性能。

还是对于上面的那个sql :

mysql> **select** \* **from** t1 **where** a **in** (**select** a **from** t2);

当我们把子查询进行物化之后，假设子查询物化表的名称为materialized\_table，该物化表存储的子查询结果集的列为m\_val，那么这个查询其实可以从下边两种角度来看待：

・从表t1的角度来看待，整个查询的意思其实是：对于t1表中的每条记录来说，如果该记录的a列的值在子查询对应的物化表中，则该记录会被加入最终的结果集。

・从子查询物化表的角度来看待，整个查询的意思其实是：对于子查询物化表的每个值来说，如果能在t1表中找到对应的a列的值与该值相等的记录，那么就把这些记录加入到最终的结果集。

也就是说其实上边的查询就相当于表s1和子查询物化表materialized\_table进行内连接：

**select** \* **from** t1 **inner join** materialized\_table **on** t1.a = m\_val;

转化成内连接之后，查询优化器就可以评估不同连接顺序需要的成本是多少，选取成本最低的那种查询方式执行查询。

虽然将子查询进行物化之后再执行查询会有建立临时表的成本，但是可以将子查询转换为JOIN还是会更有效率一点的。那能不能不进行物化操作直接把子查询转换为连接呢。

我们对比下面两个sql :

mysql> **select** \* **from** t1 **where** a **in** (**select** a **from** t2);

mysql> **select** t1.\* **from** t1 **inner join** t2 **on** t1.a = t2.a;

这两个sql的查询结果其实很像，只是说对于第二个sql的结果集没有去重，所以IN子查询和两表连接之间并不完全等价，但是将子查询转换为连接又真的可以充分发挥优化器的作用，所以MySQL提出了一个新概念**半连接(semi-join)**,将t1表和t2表进行半连接的意思就是：对于t1表的某条记录来说，**我们只关心在t2表中是否存在与之匹配的记录是否存在，而不关心具体有多少条记录与之匹配，**最终的结果集中只保留t1表的记录。semi-join 只是在MySQL内部采用的一种执行子查询的方式，MySQL并没有提供面向用户的semi-join语法。

那么怎么实现semi-join呢？

Table pullout (子查询中的表上拉)

当子查询的查询列表处**只有主键或者唯一索引列**时，可以直接把子查询中的表上拉到外层查询的FROM子句中， 并把子查询中的搜索条件合并到外层查询的搜索条件中。

比如：

mysql> **select** \* **from** t1 **where** a **in** (**select** a **from** t2 **where** t2・b = 1); -- a 是主键

我们可以直接把t2表上拉到外层查询的FRO M子句中，并且把子查询中的搜索条件合并到外层查询的搜索条件 中，上拉之后的查询就是这样的：

mysql> **select** \* **from** t1 **inner join** t2 **on** t1・a = t2・a **where** t2・b = 1; -- a 是主键

DuplicateWeedout execution strategy (重复值消除)

对于这个查询来说：

mysql> **select** \* **from** t1 **where** a **in** (**select** e **from** t2 **where** t2・b = 1); -一 e 只是一个普通字段

转换为半连接查询后，t1表中的某条记录可能在t2表中有多条匹配的记录，所以该条记录可能多次被添加到最后的结果集中，为了消除重复，我们可以建立一个临时表，比方说这个临时表长这样：

**CREATE TABLE** tmp (

id **PRIMARY KEY**

);

这样在执行连接查询的过程中，每当某条t1表中的记录要加入结果集时，就首先把这条记录的主键值加入到这个临时表里，如果添加成功，说明之前这条t1表中的记录并没有加入最终的结果集，现在把该记录添加到最终的结果集；如果添加失败，说明这条之前这条t1表中的记录已经加入过最终的结果集，这里直接把它丢弃就好了，这 种使用临时表消除semi-join结果集中的重复值的方式称之为DuplicateWeedout。

FirstMatch execution strategy (首次匹配)

FirstMatch是一种最原始的半连接执行方式，就是我们最开始的思路，先取一条外层查询的中的记录，然后到子查询的表中寻找符合匹配条件的记录，如果能找到一条，则将该外层查询的记录放入最终的结果集并且停止查找更多匹配的记录，如果找不到则把该外层查询的记录丢弃掉；然后再开始取下一条外层查询中的记录，重复上边这个过程。

LooseScan(松散索引扫描)

子查询扫描了非唯一索引，因为是非唯一索引，所以可能有相同的值，可以利用索引去重

对于某些使用IN语句的**相关**子查询，比方这个查询：

**select** \* **from** t1 **where** a **in** (**select** b **from** t2 **where** t1.b = t2.b);

它可以转换为半连接：

**select** \* **from** t1 semi **join** t2 **on** t1.a = t2.a **and** t1.b = t2.b;

然后再使用上面介绍的几种semi-join实现方式来进行实现。

**但是注意：由于相关子查询并不是一个独立的查询，所以不能转换为物化表来执行查询。**

不是所有包含IN子查询的查询语句都可以转换为semi-join，只有形如下面这样的查询才可以被转换为semi-join :

**SELECT** ... **FROM** outer\_tables **WHERE** expr **IN** (**SELECT** ... **FROM** inner\_tables ...) **AND** ...

或者

**SELECT** ... **FROM** outer\_tables **WHERE** (oe1, oe2, ...) **IN** (**SELECT** ie1, ie2, ... **FROM** inner\_tables ...) **AND** ...

如以下几种情况就不能转换为semi-join :

・外层查询的WHERE条件中有其他搜索条件与IN子查询组成的布尔表达式使用OR连接起来

・使用NOT IN而不是IN的情况

・子查询中包含GROUP BY、HAVING或者聚集函数的情况

・子查询中包含UNION的情况

那么对于不能转为semi-join查询的子查询，有其他方式来进行优化：

・对于不相关子查询来说，可以尝试把它们物化之后再参与查询

比如对于使用了 NOT IN下面这个sql:

**select** \* **from** t1 **where** a **not in** (**select** a **from** t2 **where** t2.a = 1);

请注意这里将子查询物化之后不能转为和外层查询的表的连接，因为用的是not in只能是先扫描t1表，然后对t1表的某条记录来说，判断该记录的a值在不在物化表中。

不管子查询是相关的还是不相关的，都可以把IN子查询尝试专为**EXISTS**子查询

其实对于任意一个IN子查询来说，都可以被转为EXISTS子查询，通用的例子如下：

outer\_expr **IN** (**SELECT** inner\_expr **FROM** ... **WHERE** subquery\_where)

可以被转换为：

**EXISTS** (**SELECT** inner\_expr **FROM** ... **WHERE** subquery\_where **AND** outer\_expr=inner\_expr)

这样转换的好处是，转换前本来不能用到索引，但是转换后可能就能用到索引了，比如：

**select** \* **from** t1 **where** a **in** (**select** a **from** t2 **where** t2.e = t1.e);

这个sql里面的子查询时用不到索引的，转换后变为：

**select** \* **from** t1 **where exists** (**select** 1 **from** t2 **where** t2・e = t1・e **and** t1・a = t2・a);

转换之后t2表就能用到a字段的索引了。

I

所以，如果IN子查询不满足转换为semi-join的条件，又不能转换为物化表或者转换为物化表的成本太大，那 么它就会被转换为EXISTS查询。

对于派生表的优化

mysql> **select** \* **from** (**select** a, b **from** t1) **as** t;

上面这个sql，子查询是放在from后面的，这个子查询的结果相当于一个**派生表**，表的名称是t，有a，b两个字段。 对于派生表，有两种执行方式：

把派生表物化

我们可以将派生表的结果集写到一个内部的临时表中，然后就把这个物化表当作普通表一样参与查询。当然，在对派生表进行物化时，使用了一种称为**延迟物化**的策略，也就是在查询中真正使用到派生表时才回去尝试物化派生表，而不是还没开始执行查询就把派生表物化掉。比如：

**select** \* **from** (**select** \* **from** t1 **where** a = 1) **as** derived1 **inner join** t2 **on** derived1・a = t2・a **where** t2・a =10;

如果采用物化派生表的方式来执行这个查询的话，那么执行时首先会到t1表中找出满足t1.a = 1的记录，如果找不到，说明参与连接的t1表记录就是空的，所以整个查询的结果集就是空的，所以也就没有必要去物化查询中的派生表了。

将派生表和外层的表合并，也就是将查询重写为没有派生表的形式

比如下面这个sql:

mysql> **select** \* **from** (**select** \* **from** t1 **where** a = 1) **as** t;

和下面的sql是等价的：

mysql> **select** \* **from** t1 **where** a = 1;

再看一些复杂一点的sql :

mysql> **select** \* **from** (**select** \* **from** t1 **where** a = 1) **as** t **inner join** t2 **on** t.a = t2.a **where** t2.b = 1;

我们可以将派生表与外层查询的表合并，然后将派生表中的搜索条件放到外层查询的搜索条件中，就像下面这样:

mysql> **select** \* **from** t1 **inner join** t2 **on** t1.a = t2.a **where** t1.a = 1 **and** t2.b = 1;

这样通过将外层查询和派生表合并的方式成功的消除了派生表，也就意味着我们没必要再付出创建和访问临时表的成本了。可是并不是所有带有派生表的查询都能被成功的和外层查询合并，当派生表中有这些语句就不可以和外层查询合并:

聚集函数，比如MAX。、MIN。、SUM啥的

DISTINCT

GROUP BY

HAVING

LIMIT

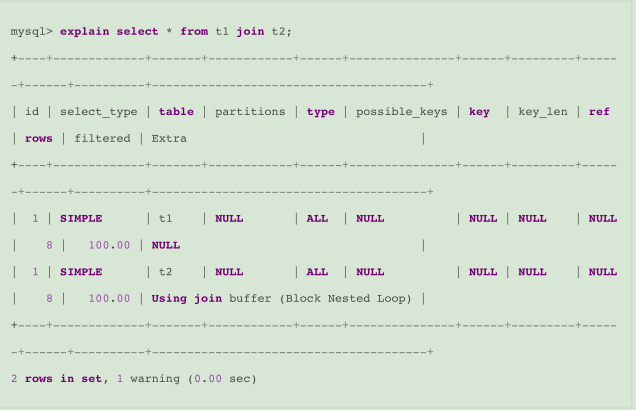
UNION 或者 UNION ALL

派生表对应的子查询的SELECT子句中含有另一个子查询

所以MySQL在执行带有派生表的时候，优先尝试把派生表和外层查询合并掉，如果不行的话，再把派生表物化掉执行查询。

Explain关键字解析

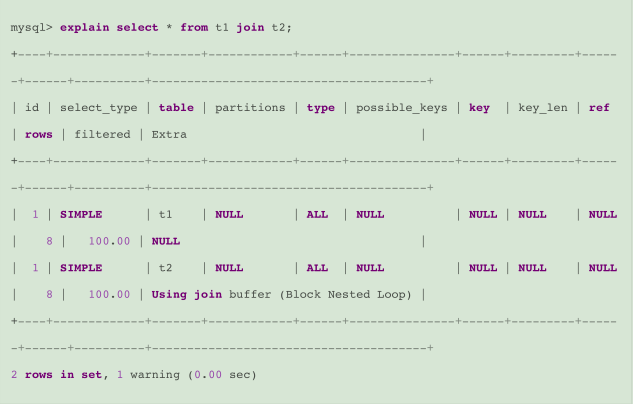
|  |  |
| --- | --- |
| **列名** | **描述** |
| id | 在一个大的查询语句中每个SELECT关键字都对应一个唯一的id |
| select\_type | SELECT关键字对应的那个查询的类型 |
| table | 表名 |
| partitions | 匹配的分区信息 |
| **type** | 针对单表的访问方法 |
| **possible\_keys** | 可能用到的索引 |
| **key** | 实际上使用的索引 |
| **key\_len** | 实际使用到的索引长度 |
| ref | 当使用索引列等值查询时，与索引列进行等值匹配的对象信息 |
| rows | 预估的需要读取的记录条数 |
| filtered | 某个表经过搜索条件过滤后剩余记录条数的百分比 |
| **Extra** | 一些额外的信息 |

Id：我们写的查询语句一般都以SELECT关键字开头，比较简单的查询语句里只有一个SELECT关键字，但是下边两种情况下在一条查询语句中会出现多个SELECT关键字：

・查询中包含子查询的情况

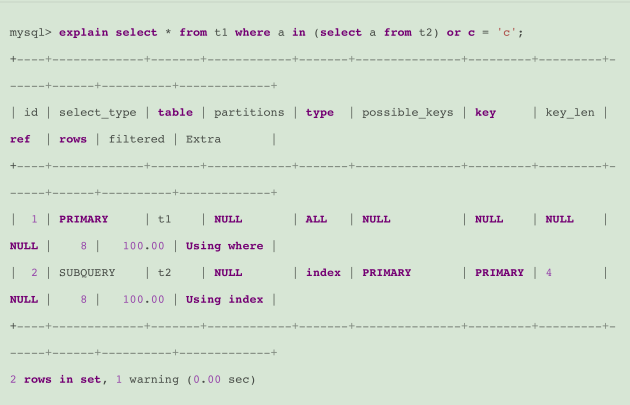
・查询中包含UNION语句的情况

查询语句中每出现一个SELECT关键字，MySQL就会为它分配一个唯一的id值。这个id值就是EXPLAIN语句的第—个列。对于连接查询来说，一个SELECT关键字后边的FROM子句中可以跟随多个表，所以在连接查询的执行计划中，每个表都会对应一条记录，但是这些记录的id值都是相同的。



在连接查询的执行计划中，每个表都会对应一条记录，这些记录的id列的值是相同的，出现在前边的表是驱动表，出现在后边的表是被驱动表

对于包含子查询的查询语句来说，就可能涉及多个SELECT关键字，所以在包含子查询的查询语句的执行计划中，每个SELECT关键字都会对应一个唯一的id值，比如这样：



但是这里大家需要特别注意，查询优化器可能对涉及子查询的查询语句进行重写，从而转换为连接查询。所以如果我们想知道查询优化器对某个包含子查询的语句是否进行了重写，直接查看执行计划就好了，比如说：

mysql> **explain select** \* **from** t1 **where** a **in** (**select** a **from** t2);

+ + + + + + + +

+ + + + +

| id | select\_type | **table** | partitions | **type** | possible\_keys | **key** | key\_len | **ref** | **rows** | filtered | Extra |

+ + + + + + + +

+ + + + +

| 1 | **SIMPLE** | t1 | **NULL** | **ALL** | **PRIMARY** | **NULL** | **NULL** |

**NULL** | 8 | 100.00 | **NULL** |

| 1 | **SIMPLE** | t2 | **NULL** | eq\_ref | **PRIMARY** | **PRIMARY** | 4 |

luban.t1.a | 1 | 100.00 | **Using index** |

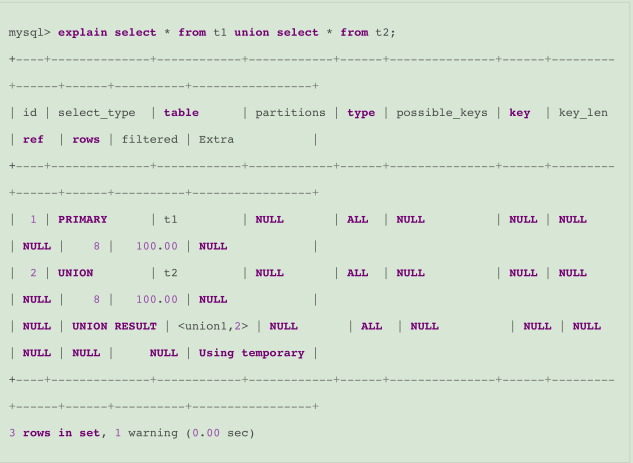
+ + + + + + + +

+ + + + +

2 **rows in set**, 1 warning (0.00 sec)

可以看到，虽然我们的查询语句是一个子查询，但是执行计划中t1和t2表对应的记录的id值全部是1，这就表明了查询优化器将子查询转换为了连接查询。

对于包含UNION子句的查询语句来说，每个SELECT关键字对应一个id值也是没错的，不过还是有点儿特别的东西，比方说下边这个查询：



这个语句的执行计划的第三条记录是什么？为什么id值是NULL ? UNION会把多个查询的结果集合并起来并对结果集中的记录进行去重，怎么去重呢？ MySQL使用的是内部的临时表。正如上边的查询计划中所示，UNION子句是为了把id为1的查询和id为2的查询的结果集合并起来并去重，所以在内部创建了一个名为的临时表（就是执行计划第三条记录的table列的名称），id为NULL表明这个临时表是为了合并两个查询的结果集而创建的。

跟UNION对比起来，UNION ALL就不需要为最终的结果集进行去重，它只是单纯的把多个查询的结果集中的记录 合并成一个并返回给用户，所以也就不需要使用临时表。所以在包含UNION ALL子句的查询的执行计划中，就没有那个id为NULL的记录，如下所示：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| mysql> **explain select** \* **from** t1 **union all select** \* **from** t2; | | | | | |
| + +- |  | + | +- | +- | + + + + |
| -+ | -+ | + | + |  |  |
| | id | | select\_type | | **table** | | | partitions | | **type** | possible\_keys | **key** | key\_len | **ref** |
| | **rows** | | filtered | | Extra | |  |  |  |
| + +- |  | + | +- | +- | + + + + |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| -+ | | -+ | -+ | -+ | | | | | | |
| | 1 | | | **PRIMARY** | | t1 | | | **NULL** | | **ALL** | | **NULL** | | **NULL** | | **NULL** | | **NULL** |
| | | 8 | | 100.00 | | **NULL** | | |  |  |  |  |  |  |
| | 2 | | | **UNION** | | t2 | | | **NULL** | | **ALL** | | **NULL** | | **NULL** | | **NULL** | | **NULL** |
| | | 8 | | 100.00 | | **NULL** | | |  |  |  |  |  |  |
| + | +- |  | -+ | -+- |  | ---+ | -+ | + | + | --+ |
| -+--- |  | -+ | -+ | -+ |  |  |  |  |  |  |

2 rows in set, 1 warning (0.00 sec)

select\_type：每一个SELECT关键字代表的小查询都定义了一个称之为select\_type的属性，意思是我们只要知道了某个小查询的select\_type属性，就知道了这个小查询在整个大查询中扮演了一个什么角色。

SIMPLE

查询语句中不包含UNION或者子查询的查询都算作是SIMPLE类型。

mysql> explain select \* from t1;

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| +- | ---+- | | + | | + | | +- | +- | | + | | | + | | + | |
| -+ |  |  | -+ | + | + |  |  |  |  |  |  |  |  |  |  |  |
| | | id | | | select\_type | | **table** | | | partitions | | | **type** | | | possible\_keys | | | **key** | | | key\_len | | | **ref** |
| | | **rows** | | | filtered | | Extra | |  |  |  |  |  |  |  |  |  |  |  |  |
| +- |  | +- |  | + | +- |  | +- |  | +- |  | +- |  | +- |  | +- |  |
| -+ |  |  | -+ | + | + |  |  |  |  |  |  |  |  |  |  |  |
| | | 1 | | | **SIMPLE** | | t1 | | | **NULL** | | | **ALL** | | | **NULL** | | | **NULL** | | | **NULL** | | | **NULL** |
| | |  | 8 | | 100.00 | | **NULL** | |  |  |  |  |  |  |  |  |  |  |  |  |
| +- |  | +- |  | + | +- |  | +- |  | +- |  | +- |  | +- |  | +- |  |

-+ + + +

连接查询也算是SIMPLE类型

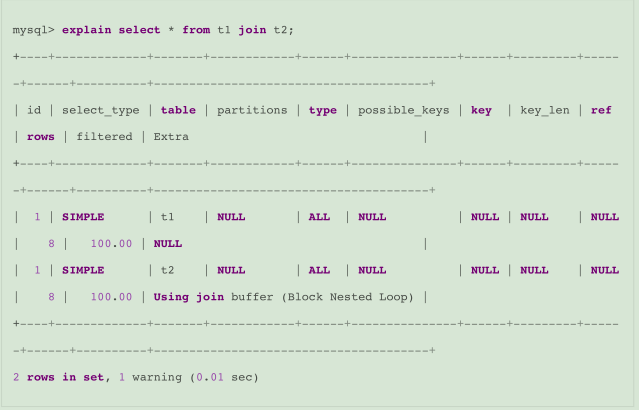
| -+--- |  | -+ | -+ |  |
| --- | --- | --- | --- | --- |
| | 1 | | | **SIMPLE** | | t1 | | **NULL** |
| | | 8 | | 100.00 | | **NULL** |  |
| | 1 | | | **SIMPLE** | | t2 | | **NULL** |
| | | 8 | | 100.00 | | **Using** | **join** buffer |
| + | +- |  | -+ | -+ |

| | **ALL** | | **NULL**  | | | **NULL** | | **NULL** |
| --- | --- | --- | --- |
| | **ALL** | |  | **NULL** | | **NULL** | | **NULL** |
| (Block Nested Loop) | | |  |  |
| --+ | --+ | ---+ | + |

+

| **NULL**

| **NULL**



....

详情看PDF文档说明.