第 8 章　MySQL 数据库 Query 的优化

8.0 引言

在之前的第 6 章“影响 MySQL Server 性能的相关因素”中已经分析了 Query 语句对数据库性能有着非常大的影响，本章将专门针对 MySQL 的 Query 语句优化进行相应的分析。

8.1 理解 MySQL 的 Query Optimizer

8.1.1 MySQL Query Optimizer 是什么？

在第 2 章的 2.2.1 节“逻辑模块组成”中已经了解到，在 MySQL 中有一个专门负责优化 SELECT 语句的优化器模块，这就是本节将要重点分析的 MySQL Query Optimizer，其主要的功能是通过计算分析系统中收集的各种统计信息，为客户端请求的 Query 给出最优的执行计划，也就是最优的数据检索方式。

当 MySQL Query Optimizer 接收到从 Query Parser （解析器）过来的 Query 时，会根据 MySQL Query 语句的相应语法对该 Query 进行分解分析，同时还会做很多其他的计算转化工作，如常量转化，无效内容删除，常量计算等。所有这些工作都是为了 Optimizer 分析出最优的数据检索方式，也就是常说的执行计划。

8.1.2 MySQL Query Optimizer 基本工作原理

在分析 MySQL Query Optimizer 的工作原理之前，先了解一下 MySQL 的 Query Tree。MySQL 的 Query Tree 是通过优化实现 DBXP 的经典数据结构和 Tree 构造器而生成的，是指导完成一个 Query 语句的请求须要处理的工作步骤，我们可以简单地认为就是一个的数据处理流程，只是以 Tree 的数据结构存放而已。通过 Query Tree 可以很清楚地知道一个 Query 的完成须要经过哪些步骤，每一步的数据来源在哪里，处理方式是怎样的。在整个 DBXP 的 Query Tree 生成过程中，MySQL 使用了 LEX 和 YACC 这两个功能非常强大的语法（词法）分析工具。MySQL Query Optimizer 的所有工作都是基于这个 Query Tree 进行的。各位读者朋友如果对 MySQL Query Tree 实现生成的详细信息比较感兴趣，可以参考 Chales A. Bell 的《Expert MySQL》这本书，里面有比较详细的介绍。

MySQL Query Optimizer 并不是一个纯粹的 CBO（Cost Base Optimizer），而是在 CBO 的基础上增加了一个被称为 Heuristic Optimize（启发式优化）的功能。也就是说，MySQL Query

Optimizer 在优化一个 Query 认为的最优执行计划时，并不一定完全按照系数据库的元信息和系统统计信息，而是在此基础上增加了某些特定的规则。其实就是在 CBO 的实现中增加了部分 RBO（Rule Base Optimizer）的功能，以确保在某些特殊场景下控制 Query 按照预定的方式生成执行计划。

当客户端向 MySQL 请求一条 Query，命令解析器模块完成请求分类，区别出是 SELECT 并转发给 MySQL Query Optimizer 时，MySQL Query Optimizer 首先会对整条 Query 进行优化，处理掉一些常量表达式的预算，直接换算成常量值。并对 Query 中的查询条件进行简化和转换，如去掉一些无用或显而易见的条件、结构调整等。然后分析 Query 中的 Hint 信息（如果有），看显示 Hint 信息是否可以完全确定该 Query 的执行计划。如果没有 Hint 或 Hint 信息还不足以完全确定执行计划，则会读取所涉及对象的统计信息，根据 Query 进行写相应的计算分析，然后再得出最后的执行计划。

Query Optimizer 是一个数据库软件非常核心的功能，虽然说起来只是简单的几句话，但在 MySQL 内部，MySQL Query Optimizer 实际上经过了很多复杂的运算分析，才得出最后的执行计划。对于 MySQL Query Optimizer 更多的信息，各位读者可通过 MySQL Internal 文档进行更为全面的了解。

8.2 Query 语句优化基本思路和原则

在分析如何优化 MySQL Query 之前，须要先了解一下 Query 语句优化的基本思路和原则。一般来说，Query 语句的优化思路和原则主要体现在以下几个方面：

（1）优化更需要优化的 Query；

（2）定位优化对象的性能瓶颈；

（3）明确优化目标；

（4）从 Explain 入手；

（5）多使用 Profile；

（6）永远用小结果集驱动大的结果集；

（7）尽可能在索引中完成排序；

（8）只取自己需要的 Columns；

（9）仅仅使用最有效的过滤条件；

（10）尽可能避免复杂的 Join 和子查询。

上面所列的几点信息，前面 4 点可以理解为 Query 优化的一个基本思路，后面部分则是优化的基本原则。

下面先针对 Query 优化的基本思路做一些简单的分析，理解 Query 优化到底该如何进行。

优化更须要优化的 Query

为什么须要优化更须要优化的 Query？我想这个问题不需要过多的解释。那什么样的 Query 更须要优化呢？这个问题须要从对整个系统的影响来考虑。哪个 Query 的优化能给系统整体带来更大的收益，就更须要优化。一般来说，高并发低消耗（相对）的 Query 对整个系统的影响远比低并发高消耗的大。下面可以通过以下一个非常简单的案例分析充分说明问题。

假设有一个 Query 每小时执行 10 000 次，每次需要 20 个 IO，而另外一个 Query 每小时执行 10 次，每次需要 20 000 个 IO。

首先通过 IO 消耗来分析。可以看出，两个 Query 每小时所消耗的 IO 总数目是一样的，都是 200 000 IO/小时。假设优化第一个 Query，从 20 个 IO 降低到 18 个 IO，也就是降低了 2 个 IO，则节省了 2×10 000 = 20 000 （IO/小时）。而如果希望通过优化第二个 Query 达到相同的效果，必须要让每个 Query 减少 20 000 / 10 = 2000 IO。可以看出第一个 Query 节省 2 个 IO 即可达到第二个 Query 节省 2000 个 IO 相同的效果。

其次，通过 CPU 消耗来分析。原理和上面一样，只要让第一个 Query 节省一小块资源，就可以让整个系统节省出一大块资源，尤其是在排序、分组这些对 CPU 消耗比较多的操作中更加明显。

最后，从对整个系统的影响来分析。一个频繁执行的高并发 Query 的危险性比一个低并发的 Query 要大很多。当一个低并发的 Query 执行计划有误时，所带来的影响只是该 Query 请求者的体验会变差，对整体系统的影响并不会特别突出，至少还属于可控范围。但是，如果一个高并发的 Query 执行计划有误，那它带来的后果很可能就是灾难性的，很多时候可能连自救的机会都没有，就会让整个系统崩溃掉。我曾经就遇到过这样一个案例，一个并发度较高的 Query 语句执行计划有误，系统顷刻间崩溃，当重新启动数据库提供服务时，系统负载直线飙升，甚至都来不及登录数据库查看有哪些 Active 的线程在执行哪些 Query。如果是遇到一个并发不太高的 Query 执行计划有误，至少还可以控制整个系统，不至于系统被直接压跨，甚至连问题根源都难以抓到。

定位优化对象的性能瓶颈

当我们拿到一条须要优化的 Query 时，第一件事情是什么？是反问自己这条 Query 有什么问题？我为什么要优化他？只有明白了这些问题，才能知道须要做什么，才能够找到问题的关键。

不能只是觉得某个 Query 好像有点慢，须要优化一下，然后就开始一个一个优化方法去轮番尝试。这样很可能会消耗大量的人力和时间成本，甚至可能到最后还是得不到一个好的优化结果。这就像看病一样，医生必须要清楚病的根源才能对症下药。如果只是知道什么地方不舒服，然后就开始通过各种药物尝试治疗，那后果可能就非常严重了。

所以，在拿到一条须要优化的 Query 之后，首先要判断出这个 Query 的瓶颈到底是 IO 还是 CPU，到底是因为在数据访问上消耗了太多的时间，还是在数据的运算（如分组排序等）方面花费了太多资源。

一般来说，在 MySQL 5.0 系列版本中，可以通过系统自带的 PROFILING 功能清楚地找出一个 Query 的瓶颈。当然，如果读者朋友为了使用 MySQL 的某些在 5.1 版本中才有的新特性（如 Partition，EVENT 等），抑或是早早使用 MySQL 5.1 的预发布版本，可能就没办法使用这个功能了，因为该功能在 MySQL 5.1 系列最初的版本中并不支持，不过让人非常兴奋的是该功能在最新的 MySQL 5.1 正式版（5.1.30）又已经提供了。如果读者朋友正在使用的是 4.x 版本，那就只能通过自行分析 Query 的各个执行步骤，找到性能损失最大的地方了。

明确的优化目标

在定位了一条 Query 的性能瓶颈之后，就须要通过分析该 Query 所完成的功能和 Query 对系统的整体影响制订出一个明确的优化目标。没有一个明确的目标，优化过程将是一个漫无目的而且低效的过程，很难达到一个理想的效果，尤其是对于一些实现应用中较为重要功能点的 Query。

如何设定优化目标？这可能是很多人都非常头疼的问题，对于我自己也一样。要设定一个合理的优化目标，不能过于理想也不能放任自由。一般来说，首先须要清楚数据库目前的整体状态，同时也要清楚数据库中与该 Query 相关的数据库对象的各种信息，而且还要了解该 Query 在整个应用系统中所实现的功能。了解了数据库整体状态，就能知道数据库所能承受的最大压力，也就清楚了我们能够接受的最悲观情况。把握了该 Query 相关数据库对象的信息，就应该知道实现该 Query 最理想情况下须要消耗多少资源，最糟糕又须要消耗多少资源。最后，通过该 Query 所实现的功能点在整个应用系统中的重要地位，可以大概地分析出该 Query 占用的系统资源比例，还能知道该 Query 的效率给客户带来的体验影响到底有多大。

在清楚了这些信息之后，基本可以得出该 Query 应该满足的一个性能范围，这也就是优化目标范围，然后就是寻找相应的优化手段来解决问题了。如果该 Query 实现的应用系统功能比较重要，则必须让目标更偏向于理想值，即使在其他某些方面作出一些让步与牺牲也是需要的，比如调整 schema 设计，调整索引组成等。而如果该 Query 所实现的是一些并不是太关键的功能，那可以让目标偏向悲观值，尽量保证其他更重要的 Query 性能。这种时候，即使须要调整商业需求，减少功能实现，也不得不作出让步。

从 Explain 入手

现在，优化目标已经明确了，到动手时候了。优化该从何处入手呢？答案只有一个，从 Explain 开始入手。为什么？因为只有 Explain 才能告诉你，这个 Query 在数据库中是以一个怎样的执行计划来实现的。

但是，有一点必须清楚，Explain 只是用来获取一个 Query 在当前状态的数据库中的执行计划的，在优化之前，我们必须根据优化目标在头脑中有一个清晰的目标执行计划。只有这样，优化的目标才有意义。一个优秀的 SQL 调优人员（或者成为 SQL Performance Tuner），在优化任何一个 SQL 语句之前，都应该在自己头脑中有一个预定的执行计划，然后不断地调整尝试，再借助 Explain 来验证调整的结果是否满足预定的执行计划。对于不符合预期的执行计划须要不断分析 Query 的写法和数据库对象的信息，继续调整尝试，直至得到预期的结果。

当然，并不一定每次预设的执行计划都是最优的，在不断调整测试的过程中，如果发现 MySQL Query Optimizer 所选择的执行计划实际执行效果确实比自己预设的好，则应该选择使用 MySQL Query Optimizer 所生成的执行计划。

上面的优化思路，只是一个优化的基本方向，实际操作还需要读者结合具体应用场景不断的测试实践。当然也并不一定所有的情况都要严格遵循这样一个思路，规则是死的，人是活的，只有更合理的方法，没有最合理的规则。

在了解了上面这些优化的基本思路之后，再来看看优化的基本原则。

永远用小结果集驱动大的结果集

很多人喜欢在优化 SQL 的时候使用小表驱动大表，个人认为这不太严谨。为什么？因为大表经过 WHERE 条件过滤之后返回的结果集并不一定就比小表所返回的大，也许更小。在这种情况下如果仍然采用小表驱动大表，就会得到相反的性能效果。

其实这也非常容易理解，在 MySQL 中，只有 Nested Loop 一种 Join 方式，也就是说 MySQL 的 Join 都是通过嵌套循环来实现的。驱动结果集越大，所需要循环就越多，那么被驱动表的访问次数自然也就越多，而每次访问被驱动表，即使需要的逻辑 IO 很少，循环次数多了，总量也不可能小，而且每次循环都不能避免消耗 CPU，所以 CPU 运算量也会跟着增加。如果仅仅以表的大小来作为驱动表的判断依据，假若小表过滤后所剩下的结果集比大表多很多，结果就会在嵌套循环中带来更多的循环次数，反之，所需要的循环次数就会更少，总体 IO 量和 CPU 运算量也会更少。在非 Nested Loop 的 Join 算法中，如 Oracle 中的 Hash Join，小结果集驱动大的结果集同样是最优的选择。

所以，在优化 Join Query 的时候，最基本的原则就是“小结果集驱动大结果集”，通过这个原则来减少嵌套循环中的循环次数，以减少 IO 总量及 CPU 运算的次数。

尽可能在索引中完成排序

排序操作是非常消耗 CPU 的操作，当系统设置不当或 Query 取出的字段过多时，还可以造成 MySQL 不得不放弃优化后的排序算法，而使用较为古老的须要两次 IO 读取表数据的排序算法，使排序效率非常低下。

利用索引进行排序操作，主要是利用了索引的有序性。在通过索引进行检索的过程中，就已经得到了有序的数据访问顺序，依次读取结果数据后就不须要进行排序操作，进而避免了此操作，提高了需要有序结果集的 Query 的性能。

只取出自己需要的 Columns

任何时候在 Query 中都只取出需要的 Columns，尤其是在需要排序的 Query 中。为什么？

对于任何 Query，返回的数据都须要通过网络数据包传回给客户端，取出的 Column 越多，须要传输的数据量自然会越大，不论是从网络带宽方面考虑还是从网络传输的缓冲区来看，这都是一个浪费。

如果是须要排序的 Query，其影响就更大了。在 MySQL 中存在两种排序算法，一种是在 MySQL 4.1 之前的算法，实现方式是先将须要排序的字段和可以直接定位到相关行数据的指针信息取出，然后在设定的排序区（通过参数 sort_buffer_size 设定）中进行排序，完成排序之后再次通过行指针信息取出所需的 Columns，也就是说这种算法须要访问两次数据。第二种排序算法是从 MySQL 4.1 版本开始使用的改进算法，一次性将所需的 Columns 全部取出，在排序区排序后直接将数据返回给请求客户端。改行算法只须要访问一次数据，减少了大量的随机 IO，极大地提高了排序 Query 语句的效率。但是，这种改进后的排序算法一次性取出并缓存的数据比第一种算法要多很多，如果我们将并不需要的 Columns 也取出来，就会极大地浪费排序过程所需要的内存。在 MySQL 4.1 之后的版本中，可以通过设置 max_length_for_sort_data 参数来控制 MySQL 选择第一种排序算法还是第二种。当取出的所有大字段总大小大于 max_length_for_sort_data 的设置时，MySQL 就会选择使用第一种排序算法，反之，则会选择第二种。为了尽可能地提高排序性能，我们自然更希望使用第二种排序算法，所以在 Query 中仅仅取出需要的 Columns 是非常有必要的。

仅仅使用最有效的过滤条件

很多人在优化 Query 语句的时候容易进入一个误区，觉得 WHERE 子句中的过滤条件越多越好，实际上这并不是一个非常正确的选择。其实分析 Query 语句的性能优劣最关键的就是要让它选择一条最佳的数据访问路径，做到通过访问最少的数据量完成自己的任务。

为什么说过滤条件多不一定是好事呢？请看下面示例。

需求：查找某个用户在所有 group 中所发的讨论 message 基本信息。

场景：（1）知道用户 ID 和用户 nick_name

（2）信息所在表为 group_message

（3）group_message 中存在用户 ID(user_id)和 nick_name(author)两个索引

方案一：将用户 ID 和用户 nick_name 两者都作为过滤条件放在 WHERE 子句中来查询，Query 的执行计划如代码示例 8-1 所示：

代码 8-1

```
sky@localhost : example 11:29:37> EXPLAIN SELECT * FROM group_message
    ->  WHERE user_id = 1 AND author='1111111111'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
         type: ref
possible_keys: group_message_author_ind,group_message_uid_ind
          key: group_message_author_ind
      key_len: 98
          ref: const
         rows: 1
        Extra: Using where
1 row in set (0.00 sec)
```

方案二：仅仅将用户 ID 作为过滤条件放在 WHERE 子句中来查询，Query 的执行计划如示例代码 8-2 所示：

代码 8-2

```
sky@localhost : example 11:30:45> EXPLAIN SELECT * FROM group_message
    -> WHERE user_id = 1\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
         type: ref
possible_keys: group_message_uid_ind
          key: group_message_uid_ind
      key_len: 4
          ref: const
         rows: 1
        Extra:
1 row in set (0.00 sec)
```

方案三：仅将用户 nick_name 作为过滤条件放在 WHERE 子句中来查询，Query 的执行计划如示例代码 8-3 所示：

代码 8-3

```
sky@localhost : example 11:38:45> EXPLAIN SELECT * FROM group_message
    -> WHERE author = '1111111111'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
         type: ref
possible_keys: group_message_author_ind
          key: group_message_author_ind
      key_len: 98
          ref: const
         rows: 1
        Extra: Using where
1 row in set (0.00 sec)
```

初略一看三个执行计划好像都挺好啊，每一个 Query 的执行类型都用到了索引，而且都是“ref”类型。可是仔细一分析就会发现，group_message_uid_ind 索引的索引键长度为 4（key_len: 4），由于 user_id 字段类型为 int，所以可以判定 Query Optimizer 给出的这个索引键长度是完全准确的。而 group_message_author_ind 索引的索引键长度为 98（key_len: 98），因为 author 字段定义为 varchar(32)，所使用的字符集是 utf8，32×3 + 2 = 98。而且， user_id 与 author（来源于 nick_name）全部是一一对应的，所以同一个 user_id 有哪些记录，所对应的 author 也会有完全相同的记录。这样，同样的数据在 group_message_author_ind 索引中所占用的存储空间要远远大于 group_message_uid_ind 索引所占用的空间。占用空间更大，代表访问该索引须要读取的数据量就会越多。所以，选择 group_message_uid_ind 的执行计划才是最好的。也就是说，上面的方案二才是最好的方案，使用了更多 WHERE 条件的方案反而没有仅仅使用 user_id 一个过滤条件的方案优。

可能有些人会问，如果将 user_id 和 author 两者建立联合索引呢？告诉你，效果可能比没有这个索引更差，因为这个联合索引的索引键更长，占用的空间将会更大。

这个示例并不一定能代表所有场景，仅仅是希望大家明白，并不是任何时候都是过滤条件越多性能越好。在实际应用场景中，肯定会存在更多、更复杂的情形，怎样使 Query 有一个更优化的执行计划，更高效的性能，还须要仔细分析各种执行计划的具体差别，才能选择出更优化的 Query。

尽可能避免复杂的 Join 和子查询

MySQL 在并发这一块并不是太好，当并发量太高的时候，系统整体性能可能会急剧下降，尤其是遇到一些较为复杂的 Query 的时候。这主要与 MySQL 内部资源的争用锁定控制有关，如读写相斥等。InnoDB 存储引擎由于实现了行级锁定可能还要稍微好一些，如果使用的是 MyISAM 存储引擎，并发一旦较高，性能下降非常明显。所以，Query 语句所涉及的表越多，须

要锁定的资源就越多。也就是说，越复杂的 Join 语句，锁定的资源也就越多，所阻塞的其他线程也就越多。相反，如果将比较复杂的 Query 语句分拆成多个较为简单的 Query 语句分步执行，每次锁定的资源也就会少很多，所阻塞的其他线程也要少一些。

可能很多读者会有疑问，将复杂 Join 语句分拆成多个简单的 Query 语句之后，那不是网络交互就会更多了吗？网络延时方面的总体消耗也就更大了啊，完成整个查询的时间不是反而更长了吗？是的，这种情况可能存在，但也并不是肯定就会如此。可以再分析一下，一个复杂的 Join Query 语句在执行的时候，须要锁定的资源比较多，可能被别人阻塞的概率也就更大，如果是一个简单的 Query，由于须要锁定的资源较少，被阻塞的概率也会小很多。所以，较为复杂的 Join Query 有可能在执行之前被阻塞而浪费了更多的时间。而且，数据库所服务的并不单单是这一个 Query 请求，还有很多其他的请求，在高并发的系统中，牺牲单个 Query 的短暂响应时间而提高整体处理能力是非常值得的。优化本身就是一门平衡与取舍的艺术，只有懂得取舍，平衡整体，才能让系统更优。

对于子查询，可能很多人都明白为什么会不被推荐使用。在 MySQL 中，子查询的实现目前还比较差，很难得到一个很好的执行计划，很多时候明明有索引可以利用，可 Query Optimizer 就是不用。MySQL 官方给出的信息说，这一问题将在 MySQL 6.0 中得到较好的解决，将会引入 SemiJoin 的执行计划，可 MySQL 6.0 离我们投入生产环境使用恐怕还有很遥远的一段时间。所以，在 Query 优化的过程中，能不用子查询就尽量不要用。

上面只是一些常用的优化原则，并不是说在 Query 优化中只须要遵循这些原则就可以，更不是说只能通过这些原则来优化。在实际优化过程中，我们还可能会遇到很多带有较为复杂商业逻辑的场景，其优化方法就只能根据不同的应用场景来具体分析，逐步调整。其实，最有效的优化，就是不要用，也就是不要实现这个商业需求。

8.3 充分利用 Explain 和 Profiling

8.3.1 Explain 的使用

说到 Explain，肯定很多读者之前已经用过了，MySQL Query Optimizer 通过执行 EXPLAIN 命令来告诉我们它将使用一个怎样的执行计划来优化 Query。所以，可以说 Explain 是在优化 Query 时最直接有效地验证我们想法的工具。在本章前面已经谈到，一个好的 SQL Performance Tuner 在动手优化一个 Query 之前，头脑中就应该已经有了一个好的执行计划，后面的优化工作只是为实现该执行计划而作出的各种调整。

在对某个 Query 优化过程中，须要不断地使用 Explain 来验证各种调整是否有效。就像前面很多示例都会通过 Explain 来验证和展示结果一样，所有的 Query 优化都应该充分利用它。

下面看一下在 MySQL Explain 功能中展示各种信息的解释。

ID：MySQL Query Optimizer 选定的执行计划中查询的序列号。

Select_type：所使用的查询类型，主要有以下这几种查询类型。

- DEPENDENT SUBQUERY：子查询内层的第一个 SELECT，依赖于外部查询的结果集。
- DEPENDENT UNION：子查询中的 UNION，且为 UNION 中从第二个 SELECT 开始的后面所有 SELECT，同样依赖于外部查询的结果集。
- PRIMARY：子查询中的最外层查询，注意并不是主键查询。
- SIMPLE：除子查询或 UNION 之外的其他查询。
- SUBQUERY：子查询内层查询的第一个 SELECT，结果不依赖于外部查询结果集。
- UNCACHEABLE SUBQUERY：结果集无法缓存的子查询。
- UNION：UNION 语句中第二个 SELECT 开始后面的所有 SELECT，第一个 SELECT 为 PRIMARY。
- UNION RESULT：UNION 中的合并结果。

Table：显示这一步所访问的数据库中的表的名称。

Type：告诉我们对表使用的访问方式，主要包含如下集中类型。

- all：全表扫描。
- const：读常量，最多只会有一条记录匹配，由于是常量，实际上只须要读一次。
- eq_ref：最多只会有一条匹配结果，一般是通过主键或唯一键索引来访问。
- fulltext：进行全文索引检索。
- index：全索引扫描。
- index_merge：查询中同时使用两个（或更多）索引，然后对索引结果进行合并（merge），再读取表数据。
- index_subquery：子查询中的返回结果字段组合是一个索引（或索引组合），但不是一个主键或唯一索引。
- rang：索引范围扫描。
- ref：Join 语句中被驱动表索引引用的查询。
- ref_or_null：与 ref 的唯一区别就是在使用索引引用的查询之外再增加一个空值的查询。
- system：系统表，表中只有一行数据；

- unique_subquery：子查询中的返回结果字段组合是主键或唯一约束。

Possible_keys：该查询可以利用的索引。如果没有任何索引可以使用，就会显示成 null，这项内容对优化索引时的调整非常重要。

Key：MySQL Query Optimizer 从 possible_keys 中所选择使用的索引。

Key_len：被选中使用索引的索引键长度。

Ref：列出是通过常量（const），还是某个表的某个字段（如果是 join）来过滤（通过 key）的。

Rows：MySQL Query Optimizer 通过系统收集的统计信息估算出来的结果集记录条数。

Extra：查询中每一步实现的额外细节信息，主要会是以下内容。

- Distinct：查找 distinct 值，当 mysql 找到了第一条匹配的结果时，将停止该值的查询，转为后面其他值查询。
- Full scan on NULL key：子查询中的一种优化方式，主要在遇到无法通过索引访问 null 值的使用。
- Impossible WHERE noticed after reading const tables：MySQL Query Optimizer 通过收集到的统计信息判断出不可能存在结果。
- No tables：Query 语句中使用 FROM DUAL 或不包含任何 FROM 子句。
- Not exists：在某些左连接中，MySQL Query Optimizer 通过改变原有 Query 的组成而使用的优化方法，可以部分减少数据访问次数。
- Range checked for each record (index map: N)：通过 MySQL 官方手册的描述，当 MySQL Query Optimizer 没有发现好的可以使用的索引时，如果发现前面表的列值已知，部分索引可以使用。对前面表的每个行组合，MySQL 检查是否可以使用 range 或 index_merge 访问方法来索取行。
- SELECT tables optimized away：当我们使用某些聚合函数来访问存在索引的某个字段时，MySQL Query Optimizer 会通过索引直接一次定位到所需的数据行完成整个查询。当然，前提是在 Query 中不能有 GROUP BY 操作。如使用 MIN()或 MAX()的时候。
- Using filesort：当 Query 中包含 ORDER BY 操作，而且无法利用索引完成排序操作的时候，MySQL Query Optimizer 不得不选择相应的排序算法来实现。
- Using index：所需数据只需在 Index 即可全部获得，不须要再到表中取数据。
- Using index for group-by：数据访问和 Using index 一样，所需数据只须要读取索引，当 Query 中使用 GROUP BY 或 DISTINCT 子句时，如果分组字段也在索引中，Extra 中的信息就会是 Using index for group-by。

- Using temporary：当 MySQL 在某些操作中必须使用临时表时，在 Extra 信息中就会出现 Using temporary 。主要常见于 GROUP BY 和 ORDER BY 等操作中。
- Using where：如果不读取表的所有数据，或不是仅仅通过索引就可以获取所有需要的数据，则会出现 Using where 信息。
- Using where with pushed condition：这是一个仅仅在 NDBCluster 存储引擎中才会出现的信息，而且还须要通过打开 Condition Pushdown 优化功能才可能被使用。控制参数为 engine_condition_pushdown 。

这里通过分析示例来看一下不同的 Query 语句通过 Explain 所显示的不同信息。

先看一个简单的单表 Query，如示例代码 8-4 所示：

代码 8-4

```
sky@localhost : example 11:33:18> EXPLAIN SELECT COUNT(*),MAX(id),MIN(id)
    -> FROM user\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: NULL
         type: NULL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
        Extra: SELECT tables optimized away
```

对 user 表的单表查询，查询类型为 SIMPLE，因为既没有 UNION 也不是子查询。聚合函数 MAX、MIN 及 COUNT 三者需要的数据都可以通过索引直接定位得到，所以整个实现的 Extra 信息为 SELECT tables optimized away。

再来看一个稍微复杂一点的 Query，一个子查询，如示例代码 8-5 所示：

代码 8-5

```
sky@localhost : example 11:38:48> EXPLAIN SELECT name FROM groups
    -> WHERE id IN ( SELECT group_id FROM user_group WHERE user_id = 1)\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: groups
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
```

```
          ref: NULL
         rows: 50000
        Extra: Using where
*************************** 2. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: user_group
         type: ref
possible_keys: user_group_gid_ind,user_group_uid_ind
          key: user_group_uid_ind
      key_len: 4
          ref: const
         rows: 1
        Extra: Using where
```

通过 id 信息可以得知 MySQL Query Optimizer 给出的执行计划，首先是对 groups 进行全表扫描，第二步才访问 user_group 表，所使用的查询方式是 DEPENDENT SUBQUERY，对所需数据的访问方式是索引扫描，由于过滤条件是一个整数，所以索引扫描的类型为 ref，过滤条件是 const。可以使用的索引有两个，一个是基于 user_id，另一个则是基于 group_id 的。为什么基于 group_id 的索引 user_group_gid_ind 也被列为可选索引了呢？是因为与子查询的外层查询所关联的条件是基于 group_id 的。当然，最后 MySQL Query Optimizer 还是选择了使用基于 user_id 的索引 user_group_uid_ind。

由于篇幅关系，这里就不再继续举例了，大家可以通过 Explain 功能分析应用环境中的各种 Query，了解它们在 MySQL 中到底是怎么运行的。

8.3.2 Profiling 的使用

在本章第一节中还提到通过 Query Profiler 定位一条 Query 的性能瓶颈，下面再详细介绍一下 Profiling 的用途及使用方法。

要想优化一条 Query，就须要清楚这条 Query 的性能瓶颈到底在哪里，是消耗的 CPU 计算太多，还是需要的 IO 操作太多？要想能够清楚地了解这些信息，在 MySQL 5.0 和 MySQL 5.1 正式版中已经非常容易做到，即通过 Query Profiler 功能。

MySQL 的 Query Profiler 是一个使用非常方便的 Query 诊断分析工具，通过该工具可以获取一条 Query 在整个执行过程中多种资源的消耗情况，如 CPU、IO、IPC、SWAP 等，以及发生的 PAGE FAULTS、CONTEXT SWITCHE 等，同时还能得到该 Query 执行过程中 MySQL 所调用的各个函数在源文件中的位置。下面看看 Query Profiler 的具体用法。

（1）通过执行“set profiling”命令，可以开启关闭 Query Profiler 功能。先开启 profiling 参数，如示例代码 8-6 所示：

代码 8-6

```
root@localhost : (none) 10:53:11> SET profiling=1;
Query OK, 0 rows affected (0.00 sec)
```

（2）在开启 Query Profiler 功能之后，MySQL 就会自动记录所有执行的 Query 的 profile 信息。下面执行 Query，如示例代码 8-7 所示：

代码 8-7

```
    ... ...
root@localhost : test 07:43:18> SELECT status,count(*)
    -> FROM test_profiling GROUP BY status;
+----------------+----------+
| status         | count(*) |
+----------------+----------+
| st_xxx1        |       27 |
| st_xxx2        |     6666 |
| st_xxx3        |   292887 |
| st_xxx4        |       15 |
+----------------+----------+
5 rows in set (1.11 sec)
```

（3）通过执行 “SHOW PROFILE” 命令获取当前系统中保存的多个 Query 的 profile 的概要信息，如示例代码 8-8 所示：

代码 8-8

```
root@localhost : test 07:47:35> show profiles;
+----------+------------+--------------------------------------------------------+
| Query_ID | Duration   |Query                                                   |
+----------+------------+--------------------------------------------------------+
|        1 | 0.00183100 |show databases                                          |
|        2 | 0.00007000 |SELECT DATABASE()                                       |
|        3 | 0.00099300 |desc test                                               |
|        4 | 0.00048800 |show tables                                             |
|        5 | 0.00430400 |desc test_profiling                                     |
|        6 | 1.90115800 |SELECT status,count(*) FROM test_profiling GROUP BY status|
+----------+------------+--------------------------------------------------------+
3 rows in set (0.00 sec)
```

（4）针对单个 Query 获取详细的 profile 信息。

在获取概要信息之后，就可以根据概要信息中的 Query_ID 来获取某个 Query 在执行过程中详细的 profile 信息了，具体操作如示例代码 8-9 所示：

代码 8-9

```
root@localhost : test 07:49:24> show profile cpu, block io for query 6;
+--------------------+----------+----------+-----------+-----------+------------+
| Status             |Duration  |CPU_user  |CPU_system |Block_ops_in|Block_ops_out|
+--------------------+----------+----------+-----------+-----------+------------+
| starting           | 0.000349 | 0.000000 |  0.000000 |         0 |          0 |
```

```
| Opening tables       | 0.000012 | 0.000000 |  0.000000 |         0 |           0 |
| System lock          | 0.000004 | 0.000000 |  0.000000 |         0 |           0 |
| Table lock           | 0.000006 | 0.000000 |  0.000000 |         0 |           0 |
| init                 | 0.000023 | 0.000000 |  0.000000 |         0 |           0 |
| optimizing           | 0.000002 | 0.000000 |  0.000000 |         0 |           0 |
| statistics           | 0.000007 | 0.000000 |  0.000000 |         0 |           0 |
| preparing            | 0.000007 | 0.000000 |  0.000000 |         0 |           0 |
| Creating tmp table   | 0.000035 | 0.000999 |  0.000000 |         0 |           0 |
| executing            | 0.000002 | 0.000000 |  0.000000 |         0 |           0 |
| Copying to tmp table | 1.900619 | 1.030844 |  0.197970 |       347 |         347 |
| Sorting result       | 0.000027 | 0.000000 |  0.000000 |         0 |           0 |
| Sending data         | 0.000017 | 0.000000 |  0.000000 |         0 |           0 |
| end                  | 0.000002 | 0.000000 |  0.000000 |         0 |           0 |
| removing tmp table   | 0.000007 | 0.000000 |  0.000000 |         0 |           0 |
| end                  | 0.000002 | 0.000000 |  0.000000 |         0 |           0 |
| query end            | 0.000003 | 0.000000 |  0.000000 |         0 |           0 |
| freeing items        | 0.000029 | 0.000000 |  0.000000 |         0 |           0 |
| logging slow query   | 0.000001 | 0.000000 |  0.000000 |         0 |           0 |
| logging slow query   | 0.000002 | 0.000000 |  0.000000 |         0 |           0 |
| cleaning up          | 0.000002 | 0.000000 |  0.000000 |         0 |           0 |
+----------------------+----------+----------+-----------+-----------+-------------+
```

上面的例子获取了 CPU 和 Block IO 的消耗，非常清晰，对于定位性能瓶颈非常适用。若希望得到其他的信息，都可以通过执行“SHOW PROFILE *** FOR QUERY n”来获取，各位读者朋友可以自行测试。

8.4 合理设计并利用索引

索引优化，可以说是数据库相关优化，尤其是 Query 优化中最常用的优化手段之一。很多人大部分时候都只是大概了解索引的用途，知道索引能够让 Query 执行得更快，但并不知道为什么会更快。尤其是索引的实现原理、存储方式，以及不同索引之间的区别等就更不清楚了。正因为索引对 Query 的性能影响很大，所以我们更应该深入理解 MySQL 中索引的基本实现，以及不同索引之间的区别，这样才能分析出如何设计最优的索引，最大幅度地提升 Query 的执行效率。

在 MySQL 中，主要有 4 种类型的索引，分别为：B-Tree 索引、Hash 索引、Fulltext 索引和 R-Tree 索引，下面针对这 4 种索引的基本实现方式及存储结构做一个大概的分析。

8.4.1 B-Tree 索引

B-Tree 索引是 MySQL 数据库中使用最为频繁的索引类型，除了 Archive 存储引擎之外的其他所有的存储引擎都支持 B-Tree 索引。不仅在 MySQL 中是如此，在其他的很多数据库管理系统中 B-Tree 索引也同样是作为最主要的索引类型的，这主要是因为 B-Tree 索引的存储结构

在数据库的数据检索中有着非常优异的表现。

一般来说，MySQL 中的 B-Tree 索引的物理文件大多是以 Balance Tree 的结构来存储的，也就是所有实际需要的数据都存放于 Tree 的 Leaf Node，而且到任何一个 Leaf Node 的最短路径的长度都是完全相同的，所以把它称之为 B-Tree 索引。不过，可能各种数据库（或 MySQL 的各种存储引擎）在存放自己的 B-Tree 索引的时候会对存储结构稍作改造。如 InnoDB 存储引擎的 B-Tree 索引使用的存储结构实际上是 B+Tree，在 B-Tree 数据结构的基础上做了很小的改造，在每一个 Leaf Node 上面除了存放索引键的相关信息之外，还存储了指向与该 Leaf Node 相邻的后一个 Leaf Node 的指针信息，这主要是为了加快检索多个相邻 Leaf Node 的效率。

在 InnoDB 存储引擎中，存在两种不同形式的索引，一种是 Cluster 形式的主键索引（Primary Key），另外一种则是和其他存储引擎（如 MyISAM 存储引擎）存放形式基本相同的普通 B-Tree 索引，这种索引在 InnoDB 存储引擎中被称为 Secondary Index。下面通过图 8-1 针对这两种索引的存放形式做一个比较。

图 8-1 左侧为 Clustered 形式存放的 Primary Key，右侧则为普通的 B-Tree 索引。两种索引在 Root Node 和 Branch Nodes 方面完全一样。但它们会在 Leaf Nodes 方面出现差异。在 Primary Key 中，Leaf Nodes 存放的是表的实际数据，不仅仅包括主键字段的数据，还包括其他字段的数据，整个数据以主键值有序的排列。而 Secondary Index 则和其他普通的 B-Tree 索引没有太大的差异，只是在 Leaf Nodes 除了存放索引键的相关信息外，还存放了 InnoDB 的主键值。

所以，在 InnoDB 中如果通过主键来访问数据效率是非常高的，而如果是通过 Secondary Index 来访问数据的话，InnoDB 首先通过 Secondary Index 的相关信息及相应的索引键检索到 Leaf Node，再通过 Leaf Node 中存放的主键值和主键索引来获取相应的数据行。

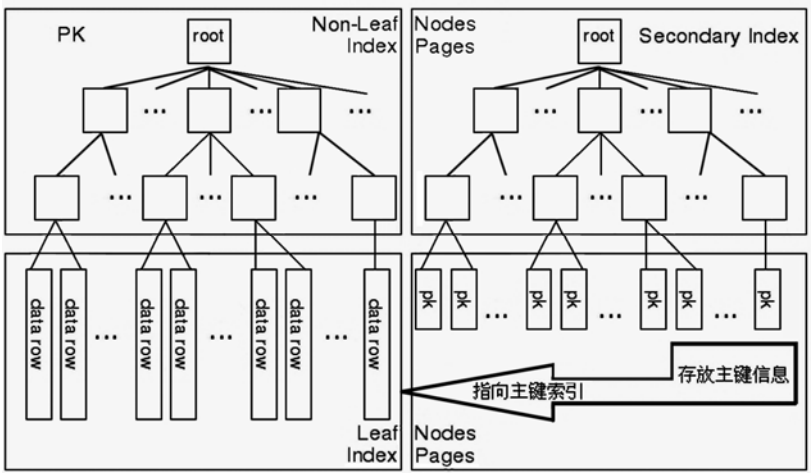

图 8-1

MyISAM 存储引擎的主键索引和非主键索引差别很小，只不过主键索引的索引键是一个唯一且非空的键。而且 MyISAM 存储引擎的索引和 InnoDB 的 Secondary Index 的存储结构基本相同，主要的区别只是 MyISAM 存储引擎在 Leaf Nodes 上除了存放索引键信息之外，还存放能直接定位到 MyISAM 数据文件中相应数据行的信息（如 Row Number），但并不会存放主键的键值信息。

8.4.2 Hash 索引

Hash 索引在 MySQL 中使用的并不是很多，目前主要是 Memory 和 NDB Cluster 存储引擎使用。所谓 Hash 索引，实际上就是通过一定的 Hash 算法，将须要索引的键值进行 Hash 运算，然后将得到的 Hash 值存入一个 Hash 表中。每次须要检索的时候，都会将检索条件进行相同算法的 Hash 运算，再和 Hash 表中的 Hash 值进行比较，并得出相应的信息。

在 Memory 存储引擎中，MySQL 还支持非唯一的 Hash 索引。可能很多人会比较惊讶，如果是非唯一的 Hash 索引，那相同的值该如何处理呢？在 Memory 存储引擎的 Hash 索引中，如果遇到非唯一值，存储引擎会将它们链接到同一个 Hash 键值下，并以一个链表的形式存在，然后在取得实际键值时过滤不符合的键。

由于 Hash 索引结构的特殊性，其检索效率非常高，索引的检索可以一次定位，不像 B-Tree 索引需要从根节点到枝节点，最后才能访问到页节点这样多次的 IO 访问，所以 Hash 索引的查询效率要远高于 B-Tree 索引。

可能很多人又有疑问了，既然 Hash 索引的效率要比 B-Tree 高很多，为什么大家不都用

Hash 索引而还要使用 B-Tree 索引呢？任何事物都是有两面性的，Hash 索引也一样，虽然 Hash 索引效率高，但是 Hash 索引本身由于其特殊性也带来了很多限制和弊端，主要有以下这些。

（1）Hash 索引仅仅能满足“=”,“IN”和“<=>”查询，不能使用范围查询。

由于 Hash 索引比较的是进行 Hash 运算之后的 Hash 值，所以它只能用于等值的过滤，不能用于基于范围的过滤，因为经过相应的 Hash 算法处理之后的 Hash 值的大小关系，并不能保证和 Hash 运算前完全一样。

（2）Hash 索引无法被用来避免数据的排序操作。

由于 Hash 索引中存放的是经过 Hash 计算之后的 Hash 值，而且 Hash 值的大小关系并不一定和 Hash 运算前的键值完全一样，所以数据库无法利用索引的数据来避免任何排序运算；

（3）Hash 索引不能利用部分索引键查询。

对于组合索引，Hash 索引在计算 Hash 值的时候是组合索引键合并后再一起计算 Hash 值，而不是单独计算 Hash 值，所以通过组合索引的前面一个或几个索引键进行查询的时候，Hash 索引也无法被利用。

（4）Hash 索引在任何时候都不能避免表扫描。

前面已经知道，Hash 索引是将索引键通过 Hash 运算之后，将 Hash 运算结果的 Hash 值和所对应的行指针信息存放于一个 Hash 表中，由于不同索引键存在相同 Hash 值，所以即使取满足某个 Hash 键值的数据的记录条数，也无法从 Hash 索引中直接完成查询，还是要通过访问表中的实际数据进行相应的比较，并得到相应的结果。

（5）Hash 索引遇到大量 Hash 值相等的情况后性能并不一定就会比 B-Tree 索引高。

对于选择性比较低的索引键，如果创建 Hash 索引，那么将会存在大量记录指针信息存于同一个 Hash 值相关联。这样要定位某一条记录时就会非常麻烦，会浪费多次表数据的访问，而造成整体性能低下。

8.4.3 Full-text 索引

Full-text 索引也就是全文索引，目前在 MySQL 中仅有 MyISAM 存储引擎支持它，但并不是所有的数据类型都支持。目前，仅有 CHAR、VARCHAR 和 TEXT 这三种数据类型的列可以建 Full-text 索引。

一般来说，Fulltext 索引主要用来替代效率低下的 LIKE '%***%' 操作。实际上，Full-text 索引并不是只能简单地替代传统的全模糊 LIKE 操作，它能通过多字段组合的 Full-text 索引一次全模糊匹配多个字段。

Full-text 索引和普通的 B-Tree 索引实现区别较大，虽然它同样是以 B-Tree 形式来存放索引数据的，但是它并不是通过字段内容的完整匹配，而是通过特定的算法，将字段数据进行分割后再进行的索引。一般来说 MySQL 系统会按照最小 4 个字节来分隔。在整个 Full-text 索引中，存储内容被分为两部分，一部分是分隔前的索引字符串数据集合，另一部分是分隔后的词（或者词组）索引信息。所以，Full-text 索引中，真正在 B-Tree 索引结构的叶节点中的并不是表中的原始数据，而是分词之后的索引数据。在 B-Tree 索引的节点信息中，存放了各个分隔后的词信息，以及指向包含该词的分隔前字符串信息在索引数据集合中的位置信息。

Full-text 索引不仅能实现模糊匹配查找，还能实现基于自然语言的匹配度查找。当然，这个匹配度到底有多准确就需要读者自行验证了。Full-text 通过一些特定的语法信息，针对自然语言做了各种相应规则的匹配，最后给出了非负的匹配值。

此外，有一点须要大家注意，MySQL 目前的 Full-text 索引在中文支持方面还不太好，须要借助第三方的补丁或插件来完成，且 Full-text 的创建所消耗的资源也比较大，所以在应用于实际生产环境之前还是尽量做好评估。

由于 Full-text 的实际使用方法不是本书的重点，感兴趣的读者可以自行参阅 MySQL 关于 Full-text 的使用手册，以了解更为详尽的信息。

8.4.4 R-Tree 索引

R-Tree 索引可能是在其他数据库中很少见的一种索引类型，主要用来解决空间数据检索的问题。

在 MySQL 中，支持一种用来存放空间信息的数据类型 GEOMETRY，且基于 OpenGIS 规范。在 MySQL 5.0.16 之前的版本中，仅 MyISAM 存储引擎支持该数据类型，但是从 MySQL 5.0.16 版本开始，BDB、InnoDB、NDBCluster 和 Archive 存储引擎也开始支持该数据类型。当然，虽然多种存储引擎都开始支持 GEOMETRY 数据类型，但是仅仅之后的 MyISAM 存储引擎支持 R-Tree 索引。

在 MySQL 中采用了具有二次分裂特性的 R-Tree 来索引空间数据信息，然后通过几何对象（MRB）信息来创建索引。

虽然只有 MyISAM 存储引擎支持空间索引（R-Tree Index），但是如果是精确的等值匹配，创建在空间数据上面的 B-Tree 索引同样可以起到优化检索的效果，空间索引的主要优势在于使用范围查找的时候，可以利用 R-Tree 索引，而 B-Tree 索引就无能为力了。

对于 R-Tree 索引的详细介绍和使用信息请参阅 MySQL 使用手册。

8.4.5 索引的利弊与如何判定，是否需要索引

相信读者都知道索引能够极大地提高数据检索的效率，让 Query 执行得更快，但是可能并不是每一位朋友都清楚索引在极大提高检索效率的同时，也给数据库带来了一些负面的影响。下面就分别对 MySQL 中索引的利与弊做一个简单的分析。

索引的好处

索引带来的益处可能很多读者会认为只是“能够提高数据检索的效率，降低数据库的 IO 成本”。

确实，在数据库中表的某个字段创建索引，所带来的最大益处就是将该字段作为检索条件时可以极大地提高检索效率，加快检索时间，降低检索过程中须要读取的数据量。但是索引带来的收益只是提高表数据的检索效率吗？当然不是，索引还有一个非常重要的用途，那就是降低数据的排序成本。

我们知道，每个索引中的数据都是按照索引键键值进行排序后存放的，所以，当 Query 语句中包含排序分组操作时，如果排序字段和索引键字段刚好一致，MySQL Query Optimizer 就会告诉 mysqld 在取得数据后不用排序了，因为根据索引取得的数据已经满足客户的排序要求。

那如果是分组操作呢？分组操作没办法直接利用索引完成。但是分组操作是须要先进行排序然后分组的，所以当 Query 语句中包含分组操作，而且分组字段也刚好和索引键字段一致，那么 mysqld 同样可以利用索引已经排好序的这个特性，省略掉分组中的排序操作。

排序分组操作主要消耗的是内存和 CPU 资源，如果能够在进行排序分组操作中利用好索引，将会极大地降低 CPU 资源的消耗。

索引的弊端

索引的益处已经清楚了，但是我们不能只看到这些益处，并认为索引是解决 Query 优化的圣经，只要发现 Query 运行不够快就将 WHERE 子句中的条件全部放在索引中。

确实，索引能够极大地提高数据检索效率，也能够改善排序分组操作的性能，但有不能忽略的一个问题就是索引是完全独立于基础数据之外的一部分数据。假设在 Table ta 中的 Column ca 创建了索引 idx_ta_ca，那么任何更新 Column ca 的操作，MySQL 在更新表中 Column ca 的同时，都须要更新 Column ca 的索引数据，调整因为更新带来键值变化的索引信息。而如果没有对 Column ca 进行索引，MySQL 要做的仅仅是更新表中 Column ca 的信息。这样，最明显的资源消耗就是增加了更新所带来的 IO 量和调整索引所致的计算量。此外，Column ca 的索引 idx_ta_ca 须要占用存储空间，而且随着 Table ta 数据量的增加，idx_ta_ca 所占用的空间也会不断增加，所以索引还会带来存储空间资源消耗的增加。

如何判定是否须要创建索引

在了解了索引的利与弊之后，那我们到底该如何来判断某个索引是否应该创建呢？

实际上，并没有一个非常明确的定律可以清晰地定义什么字段应该创建索引，什么字段不该创建索引。因为应用场景实在是太复杂，存在太多的差异。当然，还是仍然能够找到几点基本的判定策略来帮助分析的。

1. 较频繁的作为查询条件的字段应该创建索引

提高数据查询检索的效率最有效的办法就是减少须要访问的数据量，从上面索引的益处中我们知道，索引正是减少通过索引键字段作为查询条件的 Query 的 IO 量之最有效手段。所以一般来说应该为较为频繁的查询条件字段创建索引。

2. 唯一性太差的字段不适合单独创建索引，即使频繁作为查询条件

唯一性太差的字段主要是指哪些呢？如状态字段、类型字段等这些字段中存放的数据可能总

共就是那么几个或几十个值重复使用，每个值都会存在于成千上万或更多的记录中。对于这类字段，完全没有必要创建单独的索引。因为即使创建了索引，MySQL Query Optimizer 大多数时候也不会去选择使用，如果什么时候 MySQL Query Optimizer 选择了这种索引，那么非常遗憾地告诉你，这可能会带来极大的性能问题。由于索引字段中每个值都含有大量的记录，那么存储引擎在根据索引访问数据的时候会带来大量的随机 IO，甚至有些时候还会出现大量的重复 IO。

这主要是由于数据基于索引扫描的特点引起的。当我们通过索引访问表中数据时，MySQL 会按照索引键的键值顺序来依序访问。一般来说，每个数据页中大都会存放多条记录，但是这些记录可能大多数都不会和你所使用的索引键的键值顺序一致。

假如有以下场景，我们通过索引查找键值为 A 和 B 的某些数据。在通过 A 键值找到第一条满足要求的记录后，会读取这条记录所在的 X 数据页，然后继续往下查找索引，发现 A 键值所对应的另外一条记录也满足要求，但是这条记录不在 X 数据页上，而在 Y 数据页上，这时候存储引擎就会丢弃 X 数据页，而读取 Y 数据页。如此继续一直到查找完 A 键值所对应的所有记录。然后轮到 B 键值了，这时发现正在查找的记录又在 X 数据页上，可之前读取的 X 数据页已经被丢弃了，只能再次读取 X 数据页。这时候，实际上已经重复读取 X 数据页两次了。在继续往后的查找中，可能还会出现一次又一次的重复读取，这无疑给存储引擎极大地增加了 IO 访问量。

不仅如此，如果一个键值对应了太多的数据记录，也就是说通过该键值会返回占整个表比例很大的记录时，由于根据索引扫描产生的都是随机 IO，其效率比进行全表扫描的顺序 IO 效率低很多，即使不会出现重复 IO 的读取，同样会造成整体 IO 性能的下降。

很多比较有经验的 Query 调优专家经常说，当一条 Query 返回的数据超过了全表的 15%时，就不应该再使用索引扫描来完成这个 Query 了。对于“15%”这个数字我们并不能判定是否很准确，但是至少侧面证明了唯一性太差的字段并不适合创建索引。

3. 更新非常频繁的字段不适合创建索引

上面在索引的弊端中已经分析过了，索引中的字段被更新的时候，不仅要更新表中的数据，还要更新索引数据，以确保索引信息是准确的。这个问题致使 IO 访问量较大增加，不仅仅影响了更新 Query 的响应时间，还影响了整个存储系统的资源消耗，加大了整个存储系统的负载。

当然，并不是存在更新的字段就适合创建索引，从判定策略的用语上也可以看出，是“非常频繁”的字段。到底什么样的更新频率应该算是“非常频繁”呢？每秒？每分钟？还是每小时呢？说实话，还真难定义。很多时候是通过比较同一时间段内被更新的次数和利用该字段作为条件的查询次数来判断的，如果通过该字段的查询并不是很多，可能几个小时或是更长才会执行一次，更新反而比查询更频繁，那这样的字段肯定不适合创建索引。反之，如果我们通过该字段的查询比较频繁，但更新并不是特别多，比如查询几十次或更多才可能会产生一次更新，那我个人觉得

更新所带来的附加成本也是可以接受的。

4. 不会出现在 WHERE 子句中的字段不该创建索引

不会还有人会问为什么吧？自己也觉得这是废话了，哈哈！

8.4.6 单键索引还是组合索引

在大概了解了 MySQL 各种类型的索引，以及索引本身的利弊与判断一个字段是否须要创建索引之后，就要着手创建索引来优化 Query 了。在很多时候，WHERE 子句中的过滤条件并不只是针对于单一的某个字段，经常会有多个字段一起作为查询过滤条件存在于 WHERE 子句中。在这种时候，就必须要判断是该仅仅为过滤性最好的字段建立索引，还是该在所有字段（过滤条件中的）上建立一个组合索引。

对于这种问题，很难有一个绝对的定论，须要从多方面来分析考虑，平衡两种方案各自的优劣，然后选择一种最佳的方案。因为从上一节中已了解到索引在提高某些查询的性能同时，也会让某些更新的效率下降。而组合索引中因为有多个字段存在，理论上被更新的可能性肯定比单键索引要大很多，这样带来的附加成本也就比单键索引要高。但是，当 WHERE 子句中的查询条件含有多个字段时，通过这多个字段共同组成的组合索引的查询效率肯定比只用过滤条件中的某一个字段创建的索引要高。因为通过单键索引过滤的数据并不完整，和组合索引相比，存储引擎须要访问更多的记录数，自然就会访问更多的数据量，也就是说需要更高的 IO 成本。

可能有朋友会说，那可以创建多个单键索引啊。确实可以将 WHERE 子句中的每一个字段都创建一个单键索引。但是这样真的有效吗？在这样的情况下，MySQL Query Optimizer 大多数时候都只会选择其中的一个索引，然后放弃其他的索引。即使他选择了同时利用两个或更多的索引通过 INDEX_MERGE 来优化查询，所收到的效果可能并不会比选择其中某一个单键索引更高效。因为如果选择通过 INDEX_MERGE 来优化查询，就须要访问多个索引，同时还要将几个索引进行 merge 操作，这带来的成本可能反而会比选择其中一个最有效的索引更高。

在一般的应用场景中，只要不是其中某个过滤字段在大多数场景下能过滤 90%以上的数据，而其他的过滤字段会频繁的更新，一般更倾向于创建组合索引，尤其是在并发量较高的场景下。因为当并发量较高的时候，即使只为每个 Query 节省了很少的 IO 消耗，但因为执行量非常大，所节省的资源总量仍然是非常可观的。

当然，创建组合索引并不是说就须要将查询条件中的所有字段都放在一个索引中，还应该尽量让一个索引被多个 Query 语句利用，尽量减少同一个表上的索引数量，减少因为数据更新带来的索引更新成本，同时还可以减少因为索引所消耗的存储空间。

此外，MySQL 还提供了另外一个优化索引的功能，那就是前缀索引。在 MySQL 中，可以

仅仅使用某个字段的前面部分内容做为索引键索引该字段，以达到减小索引占用的存储空间和提高索引访问效率的目的。当然，前缀索引的功能仅仅适用于字段前缀随机重复性很小的字段。如果须要索引的字段前缀内容有较多的重复，索引的过滤性自然也会随之降低，通过索引所访问的数据量就会增加，这时候前缀索引虽然能够减少存储空间消耗，但是可能会造成 Query 访问效率的极大降低，得不偿失。

8.4.7 Query 的索引选择

在有些场景下，Query 由于存在多个过滤条件，而这多个过滤条件可能会存在于两个或更多的索引中。这时，MySQL Query Optimizer 一般都能根据系统的统计信息选择出一个针对该 Query 最优的索引完成查询，但是在有些情况下，可能是由于系统统计的信息不够准确完整，也可能是 MySQL Query Optimizer 自身功能的缺陷，会造成他并没有选择一个真正最优的索引而选择了其他查询效率较低的索引。这时，我们就不得不人为干预，在 Query 中增加 Hint，提示 MySQL Query Optimizer 该使用哪个索引而不该使用哪个索引，或者通过调整查询条件来达到相同的目的。

这里再次使用本章 8.2 节的“仅仅使用最有效的过滤条件”中的示例，将 group_message 表的索引做部分调整，然后进行分析。

在 group_message 上增加索引，如示例代码 8-10 所示：

代码 8-10

```
sky@localhost : example 07:10:38> CREATE index group_message_author_subject
   >on group_message(author,subject(16));
```

调整后的索引信息如示例代码 8-11 所示（出于篇幅考虑省略了主键索引）：

代码 8-11

```
sky@localhost : example 07:13:38> show indexes FROM group_message\G
   ......
*************************** 2. row ***************************
        Table: group_message
   Non_unique: 1
     Key_name: group_message_author_subject
 Seq_in_index: 1
  Column_name: author
    Collation: A
  Cardinality: NULL
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
```

```
     Comment:
*************************** 3. row ***************************
        Table: group_message
   Non_unique: 1
     Key_name: group_message_author_subject
 Seq_in_index: 2
  Column_name: subject
    Collation: A
  Cardinality: NULL
     Sub_part: 16
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
*************************** 4. row ***************************
        Table: group_message
   Non_unique: 1
     Key_name: idx_group_message_uid
 Seq_in_index: 1
  Column_name: user_id
    Collation: A
  Cardinality: NULL
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
*************************** 5. row ***************************
        Table: group_message
   Non_unique: 1
     Key_name: idx_group_message_author
 Seq_in_index: 1
  Column_name: author
    Collation: A
  Cardinality: NULL
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
```

从索引的 Sub_part 中，可以看到 subject 字段是取前 16 个字符的前缀作为索引键的。下面假设我们知道某个用户的 user_id、nick_name 和 subject 字段的部分前缀信息（weiurazs），希望通过这些条件查询出所有满足存在于 group_message 中的信息。存在三个索引可以被利用：idx_group_message_author、idx_group_message_uid 和 group_message_author_ subject，而且每个 user_id 实际上都是分别和一个 author 一一对应的。所以实际上，无论是使用 user_id 和 author（nick_name）中的某一个来作为条件或将两个都作为条件，所得到的数据是完全一样的。当然，还需要 subject LIKE 'weiurazs%' 这个条件来过滤与 subject 相关的信息。

根据三个索引的组成和查询条件，group_message_author_subject 索引可以达到最高的检索效

率，因为只有它索引了与 subject 相关的信息，而 subject 是查询必须包含的过滤条件。下面看看分别使用 user_id、author 和两者共同被使用时的执行计划，如示例代码 8-12 所示：

代码 8-12

```
sky@localhost : example 07:48:45> EXPLAIN SELECT * FROM group_message
    -> WHERE user_id = 3 AND subject LIKE 'weiurazs%'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
         type: ref
possible_keys: idx_group_message_uid
          key: idx_group_message_uid
      key_len: 4
          ref: const
         rows: 8
        Extra: Using where
1 row in set (0.00 sec)
```

很明显，这不是我们所期望的执行计划，然而并不能责怪 MySQL，因为没有使用 author 来进行过滤，所以 Optimizer 当然不会选择 group_message_author_subject 这个索引，这是我们自己的错。稍作调整，再如示例代码 8-13 所示：

代码 8-13

```
sky@localhost : example 07:48:49> EXPLAIN SELECT * FROM group_message
    -> WHERE author = '3' AND subject LIKE 'weiurazs%'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
         type: range
possible_keys: group_message_author_subject,idx_group_message_author
          key: idx_group_message_author
      key_len: 98
          ref: NULL
         rows: 8
        Extra: Using where
1 row in set (0.00 sec)
```

这次改用了 author 作为查询条件，可 MySQL Query Optimizer 仍然没有选择 group_message_author_subject 这个索引，即使通过 analyze 分析也是同样的结果，如示例代码 8-14 所示:

代码 8-14

```
sky@localhost : example 07:48:57> EXPLAIN SELECT * FROM group_message
    -> WHERE user_id = 3 AND author = '3' AND subject LIKE 'weiurazs%'\G
*************************** 1. row ***************************
```

```
           id: 1
  select_type: SIMPLE
        table: group_message
         type: range
possible_keys: group_message_author_subject,idx_group_message_uid,
               idx_group_message_author
          key: idx_group_message_uid
      key_len: 98
          ref: NULL
         rows: 8
        Extra: Using where
1 row in set (0.00 sec)
```

同时使用 user_id 和 author 时，MySQL Query Optimizer 又再次选择了 idx_group_message_uid 这个索引，仍然不是我们期望的结果，如示例代码 8-15 所示:

代码 8-15

```
sky@localhost : example 07:51:11>  EXPLAIN SELECT * FROM group_message
    -> FORCE INDEX(idx_group_message_author_subject)
    -> WHERE user_id = 3 AND author = '3' AND subject LIKE 'weiurazs%'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: group_message
         type: range
possible_keys: group_message_author_subject
          key: group_message_author_subject
      key_len: 148
          ref: NULL
         rows: 8
        Extra: Using where
1 row in set (0.00 sec)
```

最后，不得不利用 MySQL 提供的在优化 Query 时所使用的高级功能，通过显式告诉 MySQL Query Optimizer 我们要使用哪个索引的 Hint 功能。强制 MySQL 使用 group_message_author_subject 这个索引来完成查询，这才达到了需要的效果。

或许有些读者会想，会不会是因为 group_message_author_subject 这个索引本身就不是一个最优的选择呢？大家请看示例代码 8-16 中通过 mysqlslap 来执行各条 Query 的实际测试结果:

代码 8-16

```
sky@sky:~$ mysqlslap  --create-schema=example --query="SELECT * FROM
group_message WHERE user_id = 3 AND subject LIKE 'weiurazs%'" --iterations=10000
Benchmark
    Average number of seconds to run all queries: 0.021 seconds
    Minimum number of seconds to run all queries: 0.010 seconds
    Maximum number of seconds to run all queries: 0.030 seconds
    Number of clients running queries: 1
    Average number of queries per client: 1
```

```
sky@sky:~$ mysqlslap  --create-schema=example --query="SELECT * FROM
group_message WHERE author = '3' AND subject LIKE 'weiurazs%'" --iterations=10000
Benchmark
    Average number of seconds to run all queries: 0.025 seconds
    Minimum number of seconds to run all queries: 0.012 seconds
    Maximum number of seconds to run all queries: 0.031 seconds
    Number of clients running queries: 1
    Average number of queries per client: 1

sky@sky:~$ mysqlslap  --create-schema=example --query="SELECT * FROM
group_message WHERE user_id = 3 AND author = '3' AND subject LIKE 'weiurazs%'"
--iterations=10000
Benchmark
    Average number of seconds to run all queries: 0.026 seconds
    Minimum number of seconds to run all queries: 0.013 seconds
    Maximum number of seconds to run all queries: 0.030 seconds
    Number of clients running queries: 1
    Average number of queries per client: 1

sky@sky:~$ mysqlslap  --create-schema=example --query="SELECT * FROM
group_message force index(group_message_author_subject) WHERE author = '3'
subject LIKE 'weiurazs%'" --iterations=10000
Benchmark
    Average number of seconds to run all queries: 0.017 seconds
    Minimum number of seconds to run all queries: 0.010 seconds
    Maximum number of seconds to run all queries: 0.027 seconds
    Number of clients running queries: 1
    Average number of queries per client: 1
```

可以清晰地看出，通过添加 Hint 之后选择 group_message_author_subject 这个索引的 Query 确实比其他三条要快很多。

通过这个示例，可以看出在优化 Query 的时候，选择合适的索引是非常重要的，同时也证明了 MySQL Query Optimizer 并不是在任何时候都能选择出最佳的执行计划，有时，不得不通过人为的干预来让 MySQL Query Optimizer 改变它的“想法”，按照我们的思路走。

当然，这个示例仅说明了选择合适索引的重要性，并且不能在任何时候都完全相信 MySQL Query Optimizer，却没有说明到底该如何来选择一个更合适的索引。下面是我对于选择合适索引的几点建议，虽然不一定在任何场景下都合适，但在大多数场景下还是适用的。

（1）对于单键索引，尽量选择针对当前 Query 过滤性更好的索引；

（2）在选择组合索引时，当前 Query 中过滤性最好的字段在索引字段顺序中排列越靠前越好；

（3）在选择组合索引时，尽量选择可以包含当前 Query 的 WHERE 子句中更多字段的索引；

（4）尽可能通过分析统计信息和调整 Query 的写法来达到选择合适索引的目的，减少通过使用 Hint 人为控制索引的选择，因为这会使后期的维护成本增加，同时增加维护所带来的潜在风险。

8.4.8 MySQL 中索引的限制

在使用索引的同时，还应该了解 MySQL 中索引存在的限制，以便在索引应用中尽可能地避开限制所带来的问题。下面列出了目前 MySQL 中与索引使用相关的限制。

（1）MyISAM 存储引擎索引键长度的总和不能超过 1000 字节；

（2）BLOB 和 TEXT 类型的列只能创建前缀索引；

（3）MySQL 目前不支持函数索引；

（4）使用不等于（!= 或者 <>）的时候，MySQL 无法使用索引；

（5）过滤字段使用了函数运算（如 abs（column））后，MySQL 无法使用索引；

（6）Join 语句中 Join 条件字段类型不一致的时候，MySQL 无法使用索引；

（7）使用 LIKE 操作的时候如果条件以通配符开始（如'%abc...'）时，MySQL 无法使用索引；

（8）使用非等值查询的时候，MySQL 无法使用 Hash 索引。

在使用索引的时候，须要注意上面的这些限制，尤其是要注意无法使用索引的情况，因为这很容易造成极大的性能隐患。

8.5 Join 的实现原理及优化思路

前面已经了解了 MySQL Query Optimizer 的工作原理，学习了 Query 优化的基本原则和思路，理解了索引选择的技巧，这一节将围绕 Query 语句中使用频繁且随时可能存在性能隐患的 Join 语句，来继续我们的 Query 优化之旅。

8.5.1 Join 的实现原理

在寻找 Join 语句的优化思路之前，首先要理解在 MySQL 中是如何实现 Join 的，只要理解了实现原理，优化就比较简单了。下面先分析一下 MySQL 中 Join 的实现原理。

在 MySQL 中，只有一种 Join 算法，就是大名鼎鼎的 Nested Loop Join，它没有很多其他数据库所提供的 Hash Join，也没有 Sort Merge Join。顾名思义，Nested Loop Join 实际上就是通过驱动表的结果集作为循环基础数据，然后将该结果集中的数据作为过滤条件一条条地到下一个表中查询数据，最后合并结果。如果还有第三个表参与 Join，则把前两个表的 Join 结果集作为循环基础数据，再一次通过循环查询条件到第三个表中查询数据，如此往复。

下面将通过一个三表 Join 语句（如示例代码 8-17）来说明 MySQL 的 Nested Loop Join 的实现方式。

注意：由于要展示 Explain 中一个在 MySQL 5.1.18 才开始出现的输出信息（在之前版本中除了没有输出信息，实际执行过程并没有变化），所以示例环境是 MySQL 5.1.26。

代码 8-17

```
SELECT m.subject msg_subject, c.content msg_content
from user_group g,group_message m,group_message_content c
WHERE g.user_id = 1
and m.group_id = g.group_id
and c.group_msg_id = m.id
```

为了便于说明，我们通过示例代码 8-18 的操作为 group_message 表增加了一个 group_id 的索引：

代码 8-18

```
CREATE index idx_group_message_gid_uid ON group_message(group_id)
```

然后看看 Query 的执行计划，如示例代码 8-19 所示：

代码 8-19

```
sky@localhost : example 11:17:04> EXPLAIN SELECT m.subject msg_subject,
    ->c.content msg_content
    -> FROM user_group g,group_message m,group_message_content c
    -> WHERE g.user_id = 1
    -> and m.group_id = g.group_id
    -> and c.group_msg_id = m.id\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: g
         type: ref
possible_keys: user_group_gid_ind,user_group_uid_ind,user_group_gid_uid_ind
          key: user_group_uid_ind
      key_len: 4
          ref: const
         rows: 2
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: m
         type: ref
possible_keys: PRIMARY,idx_group_message_gid_uid
          key: idx_group_message_gid_uid
      key_len: 4
          ref: example.g.group_id
         rows: 3
```

```
        Extra:
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: c
         type: ref
possible_keys: idx_group_message_content_msg_id
          key: idx_group_message_content_msg_id
      key_len: 4
          ref: example.m.id
         rows: 2
        Extra:
```

可以看出，MySQL Query Optimizer 选择了 user_group 作为驱动表，首先利用传入的条件(user_id)通过该表上的索引(user_group_uid_ind)进行 const 条件的索引 ref 查找，再以 user_group 表过滤出来的结果集中的 group_id 字段作为查询条件，对 group_message 循环查询，然后通过将 user_group 和 group_message 这两个表的结果集中的 group_message 的 id 作为条件，与 group_message_content 的 group_msg_id 比较进行循环查询，才得到最终的结果。

这个过程可以通过如下表达式来表示：

```
for each record g_rec in table user_group that g_rec.user_id=1{
 for each record m_rec in group_message that m_rec.group_id=g_rec.group_id{
  for each record c_rec in group_message_content that c_rec.group_msg_id=m_rec.id
   pass the (g_rec.user_id, m_rec.subject, c_rec.content) row
   combination to output;
 }
}
```

图示 8-2 可以更清晰的标识出实际的执行情况：

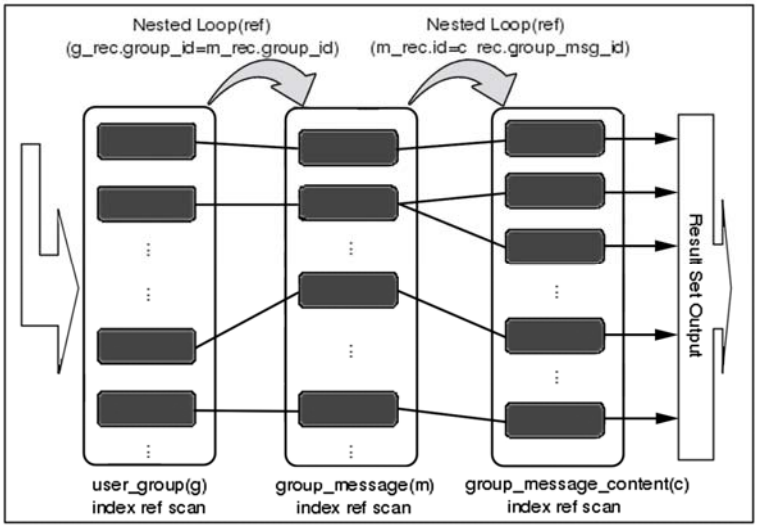

图 8-2

假如去掉 group_message_content 表中 group_msg_id 字段的索引，看看执行计划会变成怎样，如示例代码 8-20 所示：

代码 8-20

```
sky@localhost : example 11:25:36> drop index idx_group_message_content_msg_id
   >on group_message_content;
Query OK, 96 rows affected (0.11 sec)

sky@localhost : example 10:21:06> EXPLAIN
    -> SELECT m.subject msg_subject, c.content msg_content
    -> FROM user_group g,group_message m,group_message_content c
    -> WHERE g.user_id = 1
    -> and m.group_id = g.group_id
    -> and c.group_msg_id = m.id\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: g
         type: ref
possible_keys: idx_user_group_uid
          key: idx_user_group_uid
      key_len: 4
          ref: const
         rows: 2
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: m
         type: ref
possible_keys: PRIMARY,idx_group_message_gid_uid
          key: idx_group_message_gid_uid
      key_len: 4
          ref: example.g.group_id
         rows: 3
        Extra:
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: c
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 96
        Extra: Using where; Using join buffer
```

不仅 user_group 表的访问从 ref 变成了 ALL，而且最后一行的 Extra 信息从没有任何内容变成为 Using where; Using join buffer，也就是说，对于从 ref 变成 ALL 很容易理解——既然没有可以使用的索引了，当然得进行全表扫描了，Using where 也是因为变成全表扫描之后，content

字段只能通过对表中的数据进行 WHERE 过滤才能取得，但是后面出现的 Using join buffer 是什么呢？

实际上，这里的 Join 利用了在第 10 章“MySQL Server 性能优化”中提到的与 Cache 参数相关的内容，也就是通过 join_buffer_size 参数所设置的 Join Buffer。

实际上，Join Buffer 只有当 Join 类型为 ALL（如示例中）、index、rang 或 index_merge 时才能够使用，因此在去掉 group_message_content 表的 group_msg_id 字段的索引前，由于 Join 是 ref 类型的，所以执行计划中并没有使用 Join Buffer。

在使用了 Join Buffer 之后，可以通过下面这个表达式描述出示例中 Join 的完成过程：

```
for each record g_rec in table user_group{
 for each record m_rec in group_message that m_rec.group_id=g_rec.group_id{
  put (g_rec, m_rec) into the buffer
   if (buffer is full)
     flush_buffer();
 }
}

flush_buffer(){
 for each record c_rec in group_message_content that
 c_rec.group_msg_id = c_rec.id{
  for each record in the buffer
   pass (g_rec.user_id, m_rec.subject, c_rec.content) row combination to output;
 }
 empty the buffer;
}
```

当然，通过图片（如图 8-3）来展现或许更易于理解一些，如下：

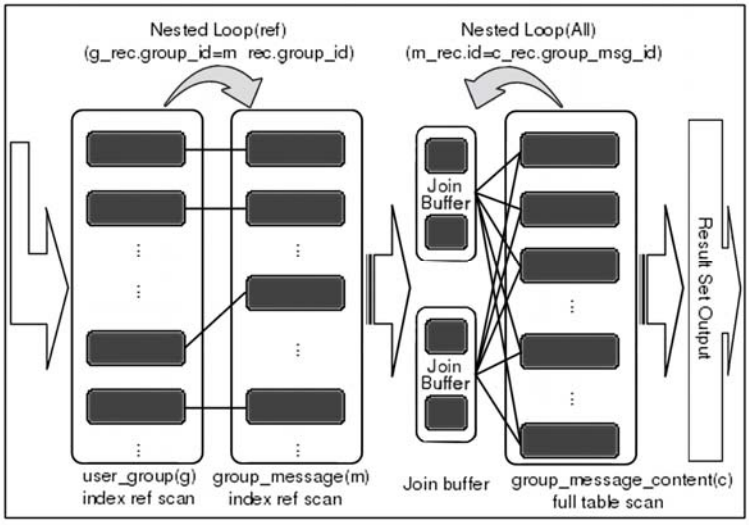

图 8-3

通过上面的示例，我想大家应该对 MySQL 中 Nested Join 的实现原理有了一定的了解，也应该清楚 MySQL 使用 Join Buffer 的方法了。当然，这里并没有涉及外连接的内容，实际对于外连接来说，区别可能主要存在于连接顺序及组合空值记录方面。

8.5.2 Join 语句的优化

在明白了 MySQL 中 Join 的实现原理后，我们就比较清楚该如何去优化 Join 语句了。

1. 尽可能减少 Join 语句中 Nested Loop 的循环总次数

如何减少 Nested Loop 的循环总次数？最有效的办法是让驱动表的结果集尽可能地小，这也正是在本章第二节中所提到的优化基本原则之一——“永远用小结果集驱动大结果集”。

因为驱动结果集越大，意味着需要循环的次数越多，也就是说在被驱动结果集上所需要执行的查询检索次数会越多。比如，当两个表（表 A 和 表 B）Join 时，如果表 A 通过 WHERE 条件过滤后有 10 条记录，而表 B 有 20 条记录。如果选择表 A 作为驱动表，也就是被驱动表的结果集为 20，那么我们通过 Join 条件对被驱动表（表 B）的比较过滤就会进行 10 次。反之，如果选择表 B 作为驱动表，则须要进行 20 次对表 A 的比较过滤。

当然，此优化的前提条件是通过 Join 条件每次对各个表进行访问的资源消耗差别不是太大。如果资源消耗存在较大的差别（一般都是因为索引的区别），就不能简单地通过结果集的大小来判断 Join 语句的驱动顺序，而是要通过比较循环次数和每次循环所须消耗之乘积大小来确定优化方案了。

2. 优先优化 Nested Loop 的内层循环

不仅在数据库的 Join 中应该这样做，实际上在优化程序语言时也有类似的优化原则。内层循环是循环中执行次数最多的，每次循环节约很少的资源，就能在整个循环中节约很多的资源。

3. 保证 Join 语句中被驱动表的 Join 条件字段已经被索引

其目的正是基于上面两点的考虑，只有让被驱动表的 Join 条件字段被索引了，才能保证循环中每次查询都能够消耗较少的资源，这也正是内层循环的实际优化方法。

4. 当无法保证被驱动表的 Join 条件字段被索引且内存资源充足时，不要太吝惜 Join Buffer 的设置

在 Join 是 All、Index、range 或 index_merge 类型的特殊情况下，Join Buffer 才能派上用场。在这种情况下，Join Buffer 的大小将对整个 Join 语句的消耗起到非常关键的作用。

8.6 ORDER BY、GROUP BY 和 DISTINCT 的优化

除了常规的 Join 语句之外，还有一类 Query 语句也是使用比较频繁的，即 ORDER BY、GROUP BY 及 DISTINCT 这三类查询。考虑到这三类查询都涉及数据的排序等操作，所以将它们放在了一起，下面就针对这三类 Query 语句做基本的分析。

8.6.1 ORDER BY 的实现与优化

在 MySQL 中，ORDER BY 的实现有如下两种类型：

- 一种是通过有序索引直接取得有序的数据，这样不用进行任何排序操作即可得到满足客户端要求的有序数据并返回给客户端；
- 另外一种则须通过 MySQL 的排序算法将存储引擎中返回的数据进行排序后，再将排序后的数据返回给客户端。

下面就针对这两种实现方式做一个简单的分析。首先分析一下第一种不用排序的实现方式。如示例代码 8-21 所示：

代码 8-21

```
sky@localhost : example 09:48:41> EXPLAIN
    -> SELECT m.id,m.subject,c.content
    -> FROM group_message m,group_message_content c
    -> WHERE m.group_id = 1 AND m.id = c.group_msg_id
    -> ORDER BY m.user_id\G
*************************** 1. row ***************************
             id:1
     select_type:SIMPLE
           table:m
            type:ref
   possible_keys:PRIMARY,idx_group_message_gid_uid
             key:idx_group_message_gid_uid
         key_len:4
             ref:const
            rows:4
           Extra:Using where
*************************** 2. row ***************************
             id:1
     select_type:SIMPLE
           table:c
            type:ref
   possible_keys:group_message_content_msg_id
             key:group_message_content_msg_id
```

```
key_len:4
    ref:example.m.id
   rows:11
  Extra:
```

该 Query 语句中明明有 ORDER BY user_id，为什么在执行计划中却没有排序操作呢？其实正因为 MySQL Query Optimizer 选择了一个有序的索引来访问表中的数据（idx_group_message_gid_uid），这样，通过 group_id 条件得到的数据已经按照 group_id 和 user_id 进行排序了。虽然排序条件仅有一个 user_id，但是 WHERE 条件决定了返回数据的 group_id 全部一样，也就是说不管有没有根据 group_id 进行排序，返回的结果集都是完全一样的。可以通过图 8-4 来描述整个执行过程：

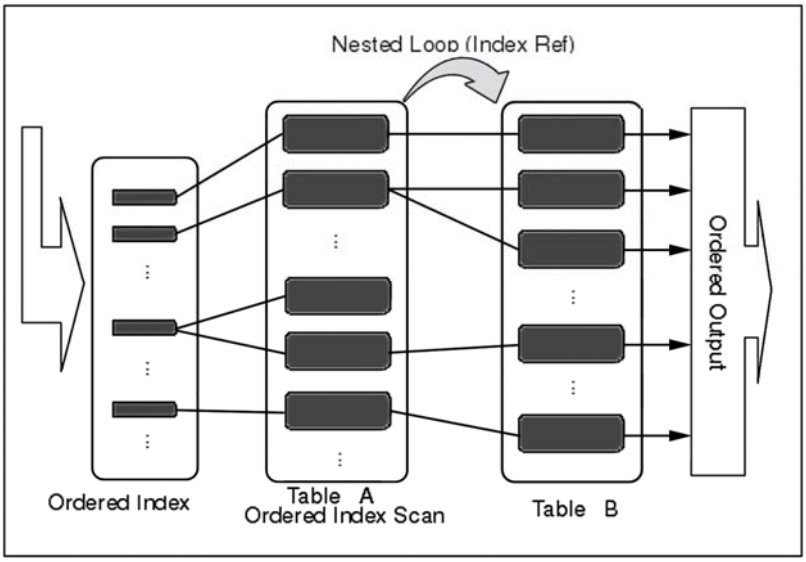

图 8-4

图中的 Table A 和 Table B 分别为上面 Query 中的 group_message 和 gruop_message_content 这两个表。

利用索引实现数据排序是 MySQL 中实现结果集排序的最佳方法，可以完全避免因为排序计算所带来的资源消耗。所以，在优化 Query 语句中的 ORDER BY 时，尽可能利用已有的索引来避免实际的排序计算，甚至可以增加索引字段，这可以很大幅度地提升 ORDER BY 操作的性能。在有些 Query 的优化过程中，即使为了避免实际的排序操作而调整索引字段的顺序，甚至是增加索引字段也是值得的。当然，在调整索引之前，须要评估调整该索引对其他 Query 造成的影响，以平衡整体得失。

如果没有索引可利用时，MySQL 又如何实现排序呢？这时 MySQL 将无法避免通过相关的

排序算法将存储引擎返回的数据进行排序运算。下面针对这种实现方式进行相应的分析。

在 MySQL 第二种排序实现方式中，必须进行相应的排序算法来实现数据的排序。MySQL 目前可以通过两种算法来实现数据的排序操作：

（1）取出满足过滤条件作为排序条件的字段，以及可以直接定位到行数据的行指针信息，在 Sort Buffer 中进行实际的排序操作，然后利用排好序的数据根据行指针信息返回表中取得客户端请求的其他字段的数据，再返回给客户端；

（2）根据过滤条件一次取出排序字段及客户端请求的所有其他字段的数据，并将不须要排序的字段存放在一块内存区域中，然后在 Sort Buffer 中将排序字段和行指针信息进行排序，最后再利用排序后的行指针与存放在内存区域中和其他字段一起的行指针信息进行匹配、合并结果集，再按照顺序返回给客户端。

上述第一种排序算法是 MySQL 一直以来就有的，而第二种则是从 MySQL 4.1 版本才开始增加的改进版排序算法。第二种算法与第一种相比，其主要优势就是减少了数据的二次访问。在排序之后不须要再一次回到表中取数据，节省了 IO 操作。当然，第二种算法会消耗更多的内存，这正是一种典型的通过内存空间换取时间的优化方式。下面同样通过一个实例（如示例代码 8-22）来看看当 MySQL 不得不使用排序算法时的执行计划，仅须更改一下排序字段：

代码 8-22

```
sky@localhost : example 10:09:06> EXPLAIN
    -> SELECT m.id,m.subject,c.content
    -> FROM group_message m,group_message_content c
    -> WHERE m.group_id = 1 AND m.id = c.group_msg_id
    -> ORDER BY m.subject\G
*************************** 1. row ***************************
           id:1
  select_type:SIMPLE
        table:m
         type:ref
possible_keys:PRIMARY,idx_group_message_gid_uid
          key:idx_group_message_gid_uid
      key_len:4
          ref:const
         rows:4
        Extra:Using where; Using filesort
*************************** 2. row ***************************
           id:1
  select_type:SIMPLE
        table:c
         type:ref
possible_keys:group_message_content_msg_id
          key:group_message_content_msg_id
      key_len:4
          ref:example.m.id
         rows:11
```

```
Extra:
```

乍一看，好像整个执行计划并没有什么区别啊？但是细心的读者可能已经发现，在 group_message 表的 Extra 信息中，多了一个“Using filesort”的信息，实际上这就是 MySQL Query Optimizer 在告诉我们，它需要进行排序操作才能按照客户端的要求返回有序的数据。如图 8-5 所示。

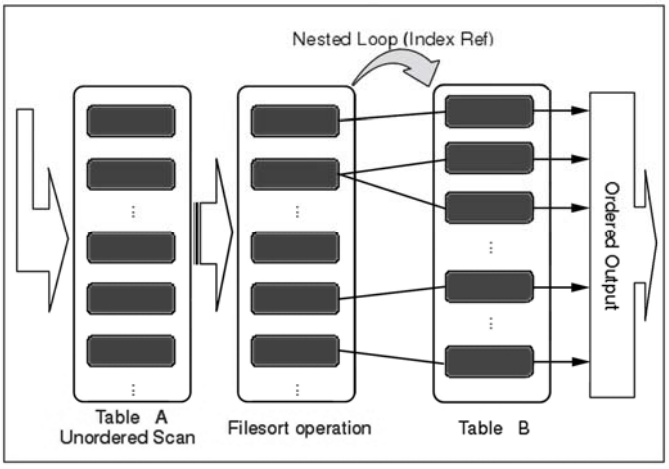

图 8-5

这里可以看到，MySQL 在取得第一个表的数据之后，先根据排序条件将数据进行了一次 filesort，也就是排序操作。然后再将排序后的结果集作为驱动结果集通过 Nested Loop Join 访问第二个表。然而，大家不要误解，这个 filesort 并不是说通过磁盘文件进行排序，而只是告诉我们进行了一个排序操作。

上面看到了排序结果集来源只是单个表的比较简单的 filesort 操作。而在实际应用中，很多时候业务要求并不是这样，须要排序的字段可能同时存在于两个表中，或者 MySQL 在经过一次 Join 之后才进行排序操作。这样的排序在 MySQL 中并不能简单地利用 Sort Buffer 进行排序，而是必须先通过一个临时表将之前 Join 的结果集存放入临时表之后，再将临时表的数据取到 Sort Buffer 中进行操作。下面通过再次更改排序要求来说明这样的执行计划，在选择通过 group_message_content 表上的 content 字段来进行排序之后，结果如示例代码 8-23 所示：

代码 8-23

```
sky@localhost : example 10:22:42> EXPLAIN
    -> SELECT m.id,m.subject,c.content
    -> FROM group_message m,group_message_content c
    -> WHERE m.group_id = 1 AND m.id = c.group_msg_id
    -> ORDER BY c.content\G
*************************** 1. row ***************************
```

```
           id:1
  select_type:SIMPLE
        table:m
         type:ref
possible_keys:PRIMARY,idx_group_message_gid_uid
          key:idx_group_message_gid_uid
      key_len:4
          ref:const
         rows:4
        Extra:Using temporary; Using filesort
*************************** 2. row ***************************
           id:1
  select_type:SIMPLE
        table:c
         type:ref
possible_keys:group_message_content_msg_id
          key:group_message_content_msg_id
      key_len:4
          ref:example.m.id
         rows:11
        Extra:
```

这时执行计划中出现了“Using temporary”，正是因为排序操作须要在两个表 Join 之后才能进行，图 8-6 展示了这个 Query 的执行过程：

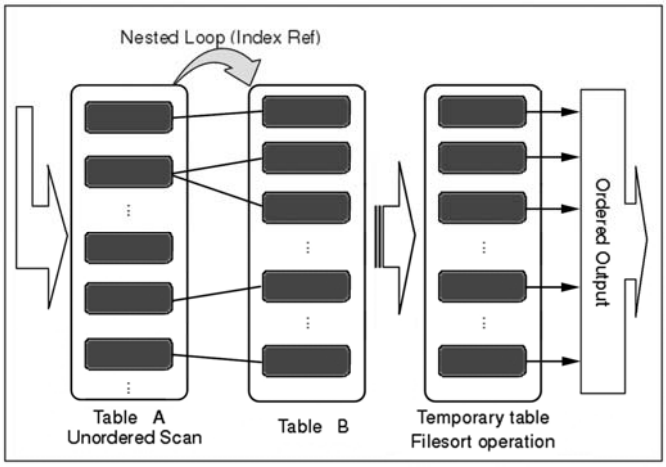

图 8-6

首先是 Table A 和 Table B 进行 Join，然后结果集进入临时表，再进行 filesort，最后得到有序的结果集数据返回给客户端。

上面通过两个不同的示例展示了当 MySQL 无法避免要使用相应的排序算法进行排序操作时的实现原理。虽然在排序过程中所使用的排序算法有两种，但是两种排序的内部实现机制大体上差不多。

当无法避免排序操作时，又该如何来优化呢？很显然，应该尽可能让 MySQL 选择使用第二种算法来进行排序。这样可以减少大量的随机 IO 操作，很大幅度地提高排序工作的效率。

1. 加大 max_length_for_sort_data 参数的设置

在 MySQL 中，决定使用老式排序算法还是改进版排序算法是通过参数 max_length_for_sort_data 来决定的。当所有返回字段的最大长度小于这个参数值时，MySQL 就会选择改进后的排序算法，反之，则选择老式的算法。所以，如果有充足的内存让 MySQL 存放须要返回的非排序字段，就可以加大这个参数的值来让 MySQL 选择使用改进版的排序算法。

2. 去掉不必要的返回字段

当内存不是很充裕时，不能简单地通过强行加大上面的参数来强迫 MySQL 去使用改进版的排序算法，否则可能会造成 MySQL 不得不将数据分成很多段，然后进行排序，这样可能会得不偿失。此时就须要去掉不必要的返回字段，让返回结果长度适应 max_length_for_sort_data 参数的限制。

3. 增大 sort_buffer_size 参数设置

增大 sort_buffer_size 并不是为了让 MySQL 选择改进版的排序算法，而是为了让 MySQL 尽量减少在排序过程中对须要排序的数据进行分段，因为分段会造成 MySQL 不得不使用临时表来进行交换排序。

8.6.2 GROUP BY 的实现与优化

由于 GROUP BY 实际上也同样须要进行排序操作，而且与 ORDER BY 相比，GROUP BY 主要只是多了排序之后的分组操作。当然，如果在分组时还使用了其他一些聚合函数，就还需要一些聚合函数的计算。所以，在 GROUP BY 的实现过程中，与 ORDER BY 一样可以利用索引。

在 MySQL 中，GROUP BY 的实现同样有多种（三种）方式，其中有两种方式会利用现有的索引信息来完成 GROUP BY，另外一种则在完全无法使用索引的场景下使用。下面分别针对这三种实现方式做一个分析。

1. 使用松散（Loose）索引扫描实现 GROUP BY

何谓松散索引扫描实现 GROUP BY 呢？实际上就是当 MySQL 完全利用索引扫描来实现 GROUP BY 时，并不须要扫描所有满足条件的索引键即可完成操作，得出结果。

下面通过一个示例来描述松散索引扫描实现 GROUP BY，在示例之前须要首先调整一下 group_message 表的索引，将 gmt_create 字段添加到 group_id 和 user_id 字段的索引中，如示例代码 8-24 所示：

代码 8-24

```
sky@localhost : example 08:49:45> CREATE index idx_gid_uid_gc
    -> ON group_message(group_id,user_id,gmt_create);
Query OK,  rows affected (0.03 sec)
Records: 96  Duplicates: 0  Warnings: 0

sky@localhost : example 09:07:30> drop index idx_group_message_gid_uid
    -> ON group_message;
Query OK, 96 rows affected (0.02 sec)
Records: 96  Duplicates: 0  Warnings: 0
```

然后再看如下 Query 的执行计划，如示例代码 8-25 所示：

代码 8-25

```
sky@localhost : example 09:26:15> EXPLAIN
    -> SELECT user_id,max(gmt_create)
    -> FROM group_message
    -> WHERE group_id < 10
    -> GROUP BY group_id,user_id\G
*************************** 1. row ***************************
           id:1
  select_type:SIMPLE
        table:group_message
         type:range
possible_keys:idx_gid_uid_gc
          key:idx_gid_uid_gc
      key_len:8
          ref:NULL
         rows:4
        Extra:Using where; Using index for group-by
1 row in set (0.00 sec)
```

可以看到在执行计划的 Extra 信息中显示有“Using index for group-by”，实际上这就是在告诉我们，MySQL Query Optimizer 通过使用松散索引扫描实现了 GROUP BY 操作。

图 8-7 描绘了扫描的主要实现过程：

要利用到松散索引扫描实现 GROUP BY，需要至少满足以下几个条件：

- GROUP BY 条件字段必须处在同一个索引中最前面的连续位置；
- 在使用 GROUP BY 的同时，只能使用 MAX 和 MIN 这两个聚合函数；
- 如果引用到了该索引中 GROUP BY 条件之外的字段条件，它就必须以常量形式存在；

为什么松散索引扫描的效率会很高？

因为在没有 WHERE 子句，也就是必须经过全索引扫描时，松散索引扫描须要读取的键值数量与分组的组数一样多，也就是说比实际存在的键值数目要少很多。而在 WHERE 子句包含范围判断式或等值表达式时，松散索引扫描会查找满足范围条件的每个组的第 1 个关键字，并且再次读取尽可能最少数量的关键字。

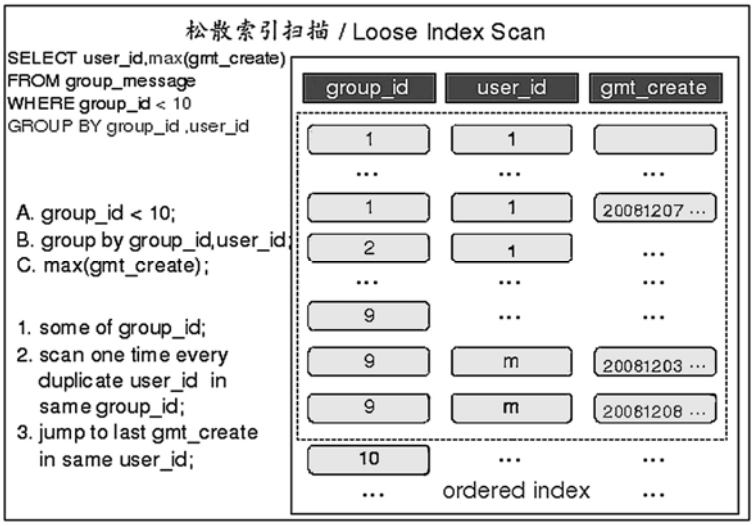

图 8-7

2. 使用紧凑（Tight）索引扫描实现 GROUP BY

紧凑索引扫描和松散索引扫描的区别主要在于前者须要在扫描索引时，读取所有满足条件的索引键，然后再根据读取到的数据来完成 GROUP BY 操作，以得到相应结果，见代码 8-26。

代码 8-26

```
sky@localhost : example 08:55:14> EXPLAIN
    -> SELECT max(gmt_create)
    -> FROM group_message
    -> WHERE group_id = 2
    -> GROUP BY user_id\G
*************************** 1. row ***************************
           id:1
  select_type:SIMPLE
        table:group_message
         type:ref
possible_keys:idx_group_message_gid_uid,idx_gid_uid_gc
          key:idx_gid_uid_gc
      key_len:4
          ref:const
         rows:4
        Extra:Using where; Using index
1 row in set (0.01 sec)
```

这时执行计划的 Extra 信息中已经没有“Using index for group-by”了，但并不是说 MySQL 的 GROUP BY 操作不是通过索引完成的，只不过是须要访问 WHERE 条件所限定的所有索引键信息之后才能得出结果。这就是通过紧凑索引扫描来实现 GROUP BY 的执行计划输出信息。

图 8-8 展示了主要的执行过程。

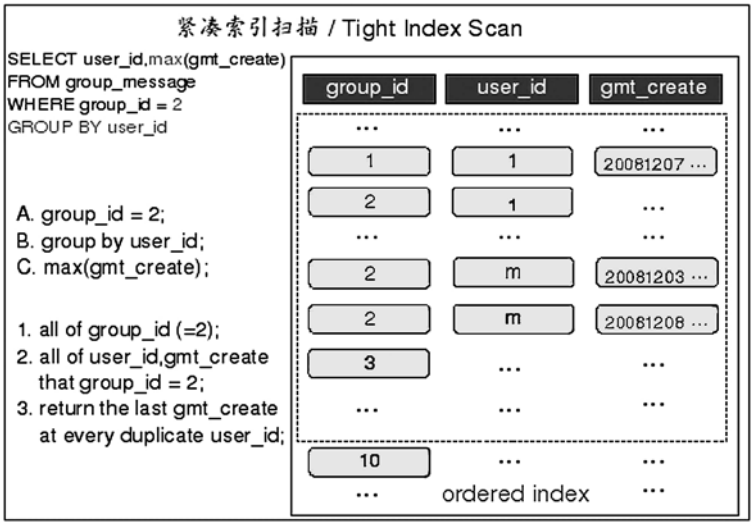

图 8-8

在 MySQL 中，MySQL Query Optimizer 首先会尝试通过松散索引扫描来实现 GROUP BY 操作，当发现某些情况无法满足松散索引扫描实现 GROUP BY 的要求时，会尝试通过紧凑索引扫描来实现。

当 GROUP BY 条件字段并不连续或不是索引前缀部分时，MySQL Query Optimizer 无法使用松散索引扫描，设置无法直接通过索引完成 GROUP BY 操作，因为缺失的索引键信息无法得到。但是，如果 Query 语句中存在一个常量值来引用缺失的索引键，则可以使用紧凑索引扫描完成 GROUP BY 操作，因为常量填充了搜索关键字中的“差距”，能形成完整的索引前缀。这些索引前缀可以用于索引查找。而如果须要排序 GROUP BY 结果，并且能够形成索引前缀的搜索关键字，MySQL 还可以避免额外的排序操作，因为使用有序索引的前缀进行搜索已经按顺序检索到了所有关键字。

3. 使用临时表实现 GROUP BY

MySQL 在进行 GROUP BY 操作时要想利用索引，必须满足 GROUP BY 的字段同时存放于同一个索引中，且该索引是一个有序索引（如 Hash 索引就不能满足要求）。不仅如此，是否能够利用索引来实现 GROUP BY 还与使用的聚合函数有关系。

前面两种 GROUP BY 的实现方式都是在有可以利用的索引时使用的，当 MySQL Query Optimizer 无法找到可以利用的合适索引时，就不得不先读取需要的数据，然后通过临时表来完成 GROUP BY 操作，如示例代码 8-27 所示：

代码 8-27

```
sky@localhost : example 09:02:40> EXPLAIN
    -> SELECT max(gmt_create)
    -> FROM group_message
    -> WHERE group_id > 1 and group_id < 10
    ->  GROUP BY user_id\G
*************************** 1. row ***************************
           id:1
  select_type:SIMPLE
        table:group_message
         type:range
possible_keys:idx_group_message_gid_uid,idx_gid_uid_gc
          key:idx_gid_uid_gc
      key_len:4
          ref:NULL
         rows:32
        Extra:Using where; Using index; Using temporary; Using filesort
```

这次的执行计划非常明显地告诉了我们 MySQL 通过索引找到了所需的数据，然后创建了临时表，又进行了排序操作，才得到所需的 GROUP BY 结果。整个执行过程大概如图 8-9 所示：

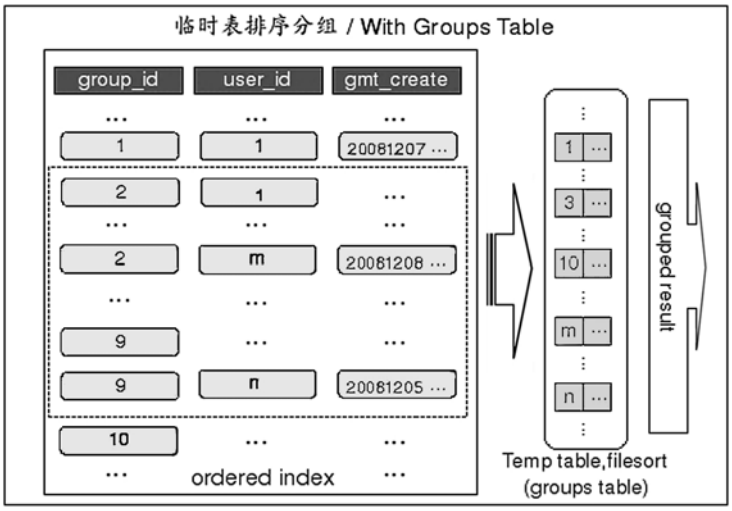

图 8-9

当 MySQL Query Optimizer 发现仅通过索引扫描并不能直接得到 GROUP BY 的结果时，它就不得不选择使用临时表，然后通过再排序的方式来实现 GROUP BY 了。

本示例是这样的情况。group_id 并不是一个常量条件，而是一个范围，而且 GROUP BY 字段为 user_id。所以 MySQL 无法根据索引的顺序来帮助 GROUP BY 的实现，只能先通过索引范围扫描得到需要的数据，将数据存入临时表，然后再进行排序和分组操作来完成 GROUP BY。

对于上面三种 MySQL 处理 GROUP BY 的方式，可以针对性地得出如下两种优化思路：

（1）尽可能让 MySQL 利用索引来完成 GROUP BY 操作，当然最好是松散索引扫描的方式。在系统允许的情况下，可以通过调整索引或调整 Query 这两种方式来达到目的；

（2）当无法使用索引完成 GROUP BY 时，由于要使用到临时表且需要 filesort，所以必须要有足够的 sort_buffer_size 供 MySQL 排序时使用，而且尽量不要进行大结果集的 GROUP BY 操作，因为如果超出系统设置的临时表大小就会出现将临时表数据复制（copy）到磁盘上面再进行操作的情况，这时的排序分组操作性能将成数量级的下降。

至于如何利用好这两种思路，还须要大家在实际应用场景中不断地尝试并测试效果，才能最终得到较佳方案。此外，在优化 GROUP BY 时还有一个小技巧可以让我们在有些无法利用到索引的情况下避免 filesort 操作，即在整个语句最后添加一个以 null 排序（ORDER BY null）的子句，大家可以尝试一下看会有什么效果。

8.6.3 DISTINCT 的实现与优化

DISTINCT 实际上和 GROUP BY 操作非常相似，只不过是在 GROUP BY 之后的每组中只取出一条记录而已。所以，DISTINCT 的实现方式和 GROUP BY 也基本相同。同样可以通过松散索引扫描或是紧凑索引扫描来实现，当然，在仅使用索引无法完成 DISTINCT 时，MySQL 只能通过临时表来完成。但是，和 GROUP BY 不同的是，DISTINCT 并不须要进行排序。也就是说，当只进行 DISTINCT 操作的 Query 仅利用索引无法完成操作时，MySQL 会利用临时表来做一次数据的“缓存”，但不会对临时表中的数据进行 filesort 操作。当然，如果在进行 DISTINCT 操作时还使用了 GROUP BY，并进行了分组，且使用了类似于 MAX 之类的聚合函数操作，就无法避免 filesort 了。

下面就通过几个简单的 Query 示例来展示一下 DISTINCT 的实现。

（1）首先看看通过松散索引扫描完成 DISTINCT 操作的过程，如示例代码 8-28 所示：

代码 8-28

```
sky@localhost : example 11:03:41> EXPLAIN SELECT DISTINCT group_id
    -> FROM group_message\G
*************************** 1. row ***************************
           id:1
  SELECT_type:SIMPLE
        table:group_message
         type:range
possible_keys:NULL
          key:idx_gid_uid_gc
      key_len:4
          ref:NULL
         rows:10
        Extra:Using index for group-by
```

可以很清晰地看到，执行计划中的 Extra 信息为“Using index for group-by”，这代表什么？为什么在没有进行 GROUP BY 操作时，执行计划中会告诉我这里通过索引进行了 GROUP BY 呢？其实它与 DISTINCT 的实现原理相关，在实现 DISTINCT 的过程中，同样也是须要分组的，然后再从每组数据中取出一条返回给客户端。而这里的 Extra 信息就是在告诉我们，MySQL 利用松散索引扫描就完成了整个操作。当然，如果 MySQL Query Optimizer 能够做得再人性化一点，将这里的信息换成“Using index for distinct”那就更容易让人理解了。

（2）再来看看紧凑索引扫描的结果，如示例代码 8-29 所示：

代码 8-29

```
sky@localhost : example 11:03:53>  EXPLAIN SELECT DISTINCT user_id
    -> FROM group_message
    -> WHERE group_id = 2\G
*************************** 1. row ***************************
           id: 1
  SELECT_type: SIMPLE
        table: group_message
         type: ref
possible_keys: idx_gid_uid_gc
          key: idx_gid_uid_gc
      key_len: 4
          ref: const
         rows: 4
        Extra: Using WHERE; Using index
```

该显示和通过紧凑索引扫描实现 GROUP BY 的结果完全一样。实际上，在这个 Query 的实现过程中，MySQL 会让存储引擎扫描 group_id = 2 的所有索引键，得出所有的 user_id，然后利用索引的已排序特性，在每更换一个 user_id 的索引键值时保留一条信息，这样就可在扫描完所有 gruop_id = 2 的索引键时完成整个 DISTINCT 操作。

（3）下面再看看单独使用索引无法完成 DISTINCT 操作时会是怎样，如果如示例代码 8-30 所示：

代码 8-30

```
sky@localhost : example 11:04:40> EXPLAIN SELECT DISTINCT user_id
    -> FROM group_message
    -> WHERE group_id > 1 AND group_id < 10\G
*************************** 1. row ***************************
           id:1
  SELECT_type:SIMPLE
        table:group_message
         type:range
possible_keys:idx_gid_uid_gc
          key:idx_gid_uid_gc
      key_len:4
          ref:NULL
```

```
         rows:32
        Extra:Using WHERE; Using index; Using temporary
1 row in set (0.00 sec)
```

当 MySQL 仅依赖索引无法完成 DISTINCT 操作时，就不得不使用临时表来进行相应的操作了。但是可以看到，当 MySQL 利用临时表来完成 DISTINCT 时，和处理 GROUP BY 有一点区别，就是少了 filesort。实际上，在 MySQL 的分组算法中，并不一定非要排序才能完成分组操作，这一点在上面的 GROUP BY 优化小技巧中已经提到过了。实际上这里 MySQL 正是在没有排序的情况下实现分组，最后完成 DISTINCT 操作的，所以少了 filesort 这个排序操作。

（4）最后再和 GROUP BY 结合试试看，如示例代码 8-31 所示：

代码 8-31

```
sky@localhost : example 11:04:40> EXPLAIN SELECT DISTINCT user_id
    -> FROM group_message
    -> WHERE group_id > 1 AND group_id < 10\G
*************************** 1. row ***************************
           id:1
  SELECT_type:SIMPLE
        table:group_message
         type:range
possible_keys:idx_gid_uid_gc
          key:idx_gid_uid_gc
      key_len:4
          ref:NULL
         rows:32
        Extra:Using WHERE; Using index; Using temporary
1 row in set (0.00 sec)
```

最后再看一下这个和 GROUP BY 一起使用的带有聚合函数的示例，与上面第三个示例相比，可以看到已经多了 filesort 排序操作了，因为我们使用了 MAX 函数。

对于 DISTINCT 的优化，思路和 GROUP BY 基本上一致，关键在于利用好索引，当无法利用索引时，确保尽量不要在大结果集上面进行 DISTINCT 操作，磁盘上面的 IO 操作和内存中的 IO 操作性能的差距完全不是一个数量级的。

8.7 小结

本章重点介绍了与 MySQL Query 语句相关的性能调优的部分思路和方法，也列举了部分的示例，希望能够帮助读者在实际工作中开阔思路。虽然本章涉及的内容包含了从最初的索引设计到编写高效 Query 语句的一些原则，以及最后对语句的调试，但 Query 语句的调优内容远不止这些。很多的调优技巧，只有到实际的调优过程中才会真正体会，真正把握其精髓。所以，希望各位读者能多做实验，以理论为基础，以事实为依据，只有这样，才能不断提升自己对 Query 调优的深入认识。