MySQL查询优化

作者：苏坡

优化目的与目标

（一）为什么要优化

优化的目的主要可分为以下四个：

1）提高资源利用率；

2）避免短板效应；

3）提高系统吞吐量；

4）同时满足更多用户的在线需求。

简单来说，优化的目的是为了提高资源的利用率，让资源充分发挥价值。常见场景下，一台服务器有4大资源：CPU、内存、网络和磁盘，一旦其中某个资源出现问题，整个服务器提供服务的能力就会变差。优化的最终目的是为了同时满足更多用户的在线需求。

（二）MySQL优化目标

MySQL优化目标主要有3个：

第一，减少磁盘IO，在数据库中主要是来自于像全表扫描这种扫描大量数据块的场景，然后就是日志以及数据块的写入所带来的压力。

第二，减少网络带宽，主要是包括两个方面，第一，SQL查询时，返回太多数据；第二，插入场景下，交互次数过多。

第三，降低CPU的消耗，主要包括三个方面，第一，MySQL本身的逻辑读，第二，额外的计算操作，比如排序分组（order by group by），第三，是聚合函数（max,min,sum...）。

总结如下：

减少磁盘IO

· 全表扫描

· 磁盘临时表

· 日志、数据块fsync

减少网络带宽

· 返回太多数据

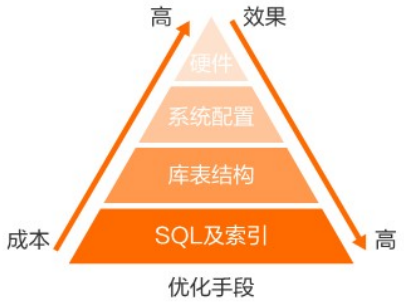
· 交互次数过多

降低CPU消耗

· 排序分组。order by, group by

· 聚合函数。max,min,sum...

· 逻辑读



上图所示的金字塔，从下往上列了4个查询的优化手段，依次是SQL及索引优化、库表结构优化、系统配置优化、硬件优化。对于单个MySQL来讲，从下往上优化，成本是逐步提升的，但效果反而越来越差。通常来说，SQL及索引调优往往不需要花费过多成本，却可以取到显著效果。

优化流程及思路

（一）关注的指标

SQL优化常规流程及思路。需要关注以下六个指标：

第一：**CPU使用率**，是SQL查询关键资源指标，CPU的消耗一般来自于数据扫描与显式计算。

第二：**IOPS**，是衡量磁盘压力的指标，它指的是每秒IO请求的次数，对数据库来说，IOPS是物理读写的关键资源指标。

第三：**QPS/TPS**，指MySQL数据库的吞吐量，也能在一定程度上反映应用系统的业务压力。

第四：**会话数/活跃会话数**，一般在应用配置问题，没有合理使用到连接池，或者SQL执行效率较差的时候出现这类的指标

异常问题，这些情况会导致数据库的Server端产生大量的会话，甚至会积压大量的活跃会话。

第五：**Innodb逻辑读/物理读**，这是主要用于反映数据库实例整体查询效率的引擎指标。

第六：**临时表**，通常来说，产生临时表往往意味着SQL执行效率的下降

总结如下：

· **CPU使用率**

SQL查询关键资源指标

数据扫描、显式计算

· **IOPS**

每秒IO请求次数

物理读写关键资源指标

· **QPS/TPS**

吞吐量

业务压力

· **会话数/活跃会话数**

应用配置

执行效率

· **Innodb逻辑读/物理读**

反映整体查询效率的引擎指标

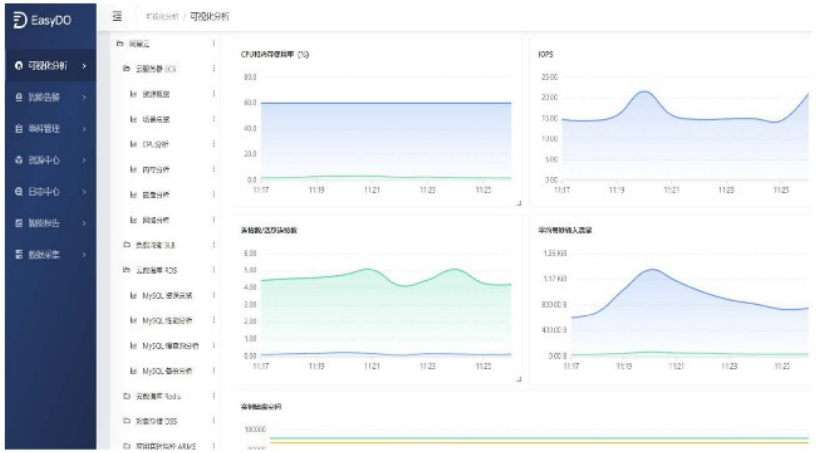
· **临时表**

导致SQL执行效率下降的特殊行为

（二）合理监控

事实上我们实际分析问题的时候，可能还会涉及到很多其他的资源指标，而这些指标数据都需要通过一个合理的方式来获取，在比较传统的时代，是通过Top、Iostat、Sar、Dstat、show status等命令去看。

下图所示是袋鼠云EasyDO智能运维平台，可以看到整个业务系统里各个数据库示例，包括CPU、IOPS等我们所关心的指标。



（三）MySQL优化流程

第一步，**构建完备的监控体系**。为了获取性能数据，分析及诊断问题，需要建立一套相对完备的监控体系。对于这块，首先需要有细致合理的告警，其次有多维度图形化指标，只有做到这两点，才可以暴露整个系统的性能缺陷，从而掌握大规模资源。

第二步，当出现问题，或者当我们发现资源指标趋势跟预想不一致的时候，需要**分析定位问题**，这个过程就是性能诊断。一般关注5点，第一，发生异常时间区间；第二，系统日志以及数据库的错误日志；第三，Slow Log日志；第四，通过合理手段对SQL执行统计；第五，Session会话分析。诊断分析之后，定位到某些会话或者某些SQL语句，可以看到异常行为。

第三步，**分析业务逻辑**，包括3点，第一，读写需求，请求量是不是正常；第二，事务精简，事务是不是有设计上的缺陷；第三，资源调用关系，比如SQL执行本身不慢，但是因为资源调用关系，出现锁等待的问题。

以上问题分析清楚之后，接下来才是要对真正有性能问题的SQL进行优化。

第四步，**SQL优化**，关于这块主要包括4点，第一，Explain查看SQL执行计划；第二，SQL改写；第三，索引调整；第四，参数调整。

总结如下：

· **构建完备的监控体系**

细致合理的告警

多维度图形化指标

暴露性能缺陷，掌控大规模资源

· **分析定位问题**

异常时间区间

System log、DB Error Log

Slow Log

SQL执行统计

session

· **分析业务逻辑**

读写需求

事务精简

资源调用关系

· **SQL优化**

explain

SQL改写

索引调整

参数调整

（四）SQL优化原则与方法

1.优化原则

SQL优化原则主要有两点：减少数据访问量与减少计算操作。

减少访问量：数据存取是数据库系统最核心功能，所以IO是数据库系统中最容易出现性能瓶颈，减少SQL访问IO量是SQL优化的第一步；数据块的逻辑读也是产生CPU开销的因素之一。

· 减少访问量的方法：创建合适的索引、减少不必访问的列、使用索引覆盖、语句改写。

减少计算操作：计算操作进行优化也是SQL优化的重要方向。SQL中排序、分组、多表连接操作等计算操作，都是CPU消耗的大户。

· 减少SQL计算操作的方法：排序列加入索引、适当的列冗余、SQL拆分、计算功能拆分。

关于SQL优化方法，包括5点

1）创建索引减少扫描量；

2）调整索引减少计算量；

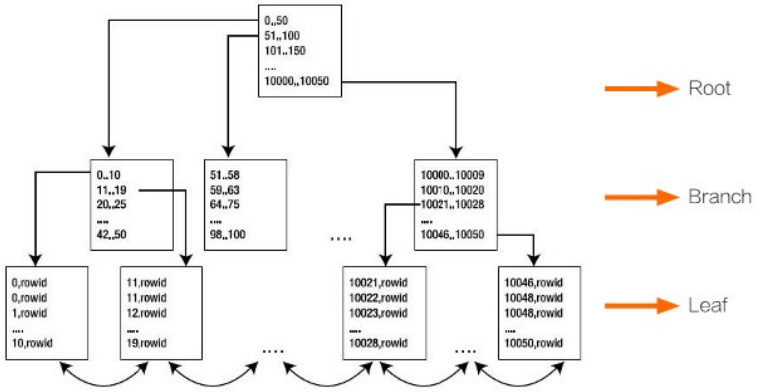
3）索引覆盖（减少不必访问的列，避免回表查询）；

4）SQL改写；

5）干预执行计划；

原理剖析

（一）B+ Tree index



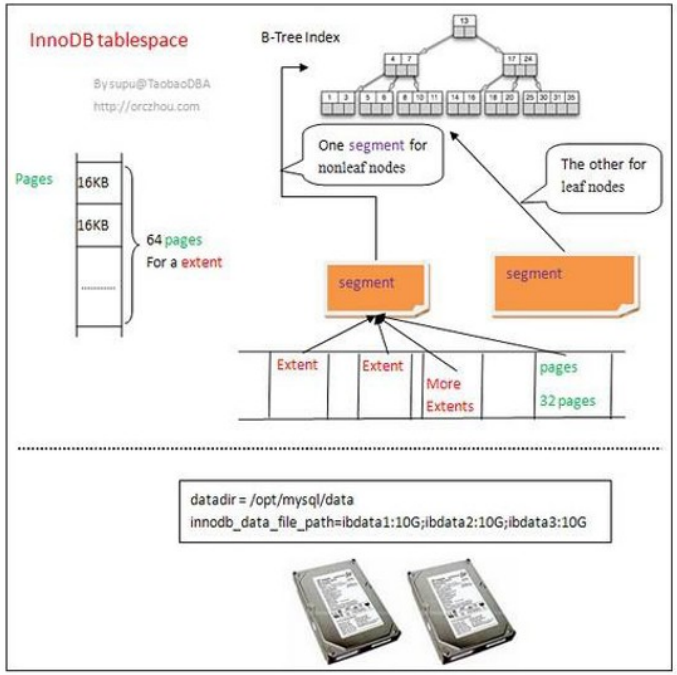
如上图所示，B+ Tree Index索引分为3个部分：根、枝、叶。

核心特点是根和枝不存储数据，行高比较固定。通过“B+Tree”索引取数据，必然经过根枝叶三个节点路径，取数据的代价比较稳定；另外一点，叶子节点上的数据是有序存储的。

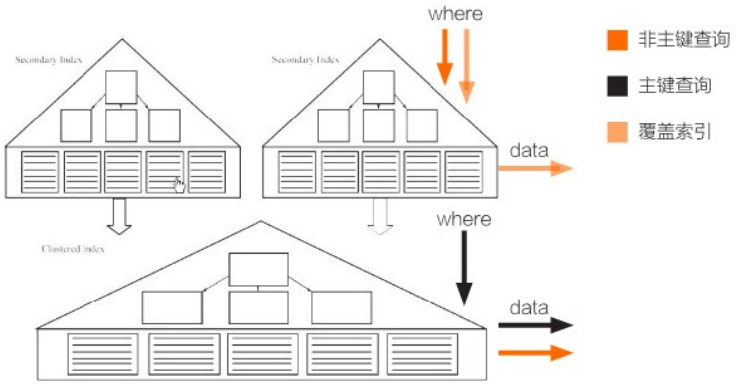
（二）Innodb Table

Innodb 是MySQL的核心存储引擎，Innodb Table是IOT有序存储，核心概念为：Innodb的表数据按照“B+ Tree”的结构进行组织，表数据本身是“B+ Tree”索引的叶子节点。

如下图所示，每张表，也就是每个存储段，实际是在MySQL里构建了一个“B+ Tree”索引的树状结构，段的物理存储跟其他关系数据库的存储方式一样分区和块。



（三）索引检索过程



如上图所示有三个流程，上面两块是二级索引，下面是属于主键索引，也叫聚集索引，是Innodb表的数据本身，依次看这三个流程：

第一，**非主键查询**，入口是从二级索引，通过二级索引，第一个过程返回聚集索引的ID；第二个过程是回表，相当于再做一次数据检索，然后从聚集索引中获取数据。

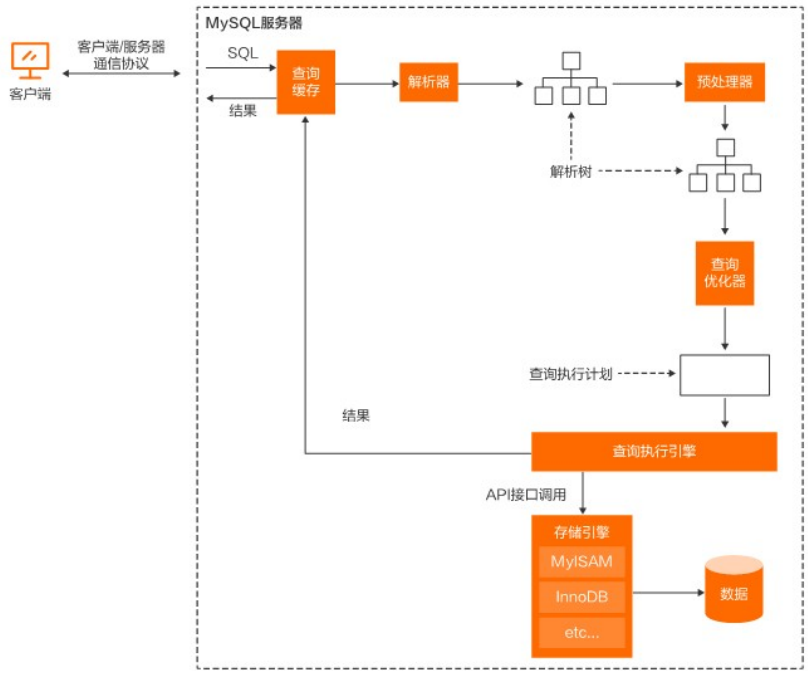
第二，**主键查询**，入口是直接通过聚集索引的ID，可以在聚集索引中获取数据。

第三，**覆盖索引**，入口是二级索引，直接从二级索引当中获取数据。

MySQL的行为

（一）MySQL SQL执行过程

1. 执行过程示例



如上图所示，MySQL的执行的过程包括：

1）客户端提交一条语句；

2）先在查询缓存查看是否存在对应的缓存数据，如有则直接返回(一般有的可能性极小，因此一般建议关闭查询缓存)；

3）交给解析器处理，解析器会将提交的语句生成一个解析树；

4）预处理器会处理解析树，形成新的解析树。这一阶段存在一些SQL改写的过程；

5）改写后的解析树提交给查询优化器。查询优化器生成执行计划；

6）执行计划交由执行引擎调用存储引擎接口，完成执行过程。这里要注意，MySQL的Server层和Engine层是分离的；

7）最终的结果由执行引擎返回给客户端，如果开启查询缓存的话，则会缓存。

2.SQL执行顺序

(8) SELECT (9) DISTINCT <select_list>

(1) FROM <left_table>

(3) <join_type> JOIN <right_table>

(2) ON <join_condition>

(4) WHERE <where_condition>

(5) GROUP <group_by_list>

(6) WITH {CUBE|ROLLUP}

(7) HAVING <having_condition>

(10) ORDER BY <order_by_list>

(11) LIMIT <limit_number>

关于SQL的执行顺序，在某些时候也可以给我们一些指导性建议。比如Where条件和Order by，在通常情况下，SQL语句先获取数据，再做Select操作，先获取数据再返回到Server端结果集的存储区之后进行排序，从这里我们可以假设如果通过索引获取数据，那么在取数据时，数据排序就已经完成，相当于MySQL存储引擎的层面已经做了优化，而不需要再增加额外的排序计算操作。

（二）MySQL优化器与执行计划

1.查询优化器

查询优化器的主要作用是用来负责生成SQL语句的执行计划。优化器是数据库的核心价值所在，它是数据库的“大脑”，优化SQL某种意义上就是理解优化器的行为。

在MySQL里面，优化的依据是执行成本，它的本质是CBO，也就是说执行计划的生成是基于成本的。目前MySQL优化器没有那么完善，执行成本主要基于行数而定。优化器工作的前提是了解数据，工作的目的是解析SQL，生成执行计划。

总结如下：

负责生成 SQL 语句的有效执行计划的数据库组件；

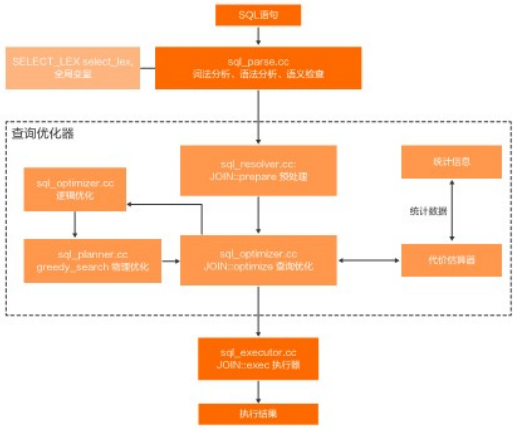
优化器是数据库的核心价值所在，它是数据库的“大脑”；

优化SQL，某种意义上就是理解优化器的行为；

优化的依据是执行成本（CBO）；

优化器工作的前提是了解数据，工作的目的是解析SQL，生成执行计划。

2.查询优化器工作过程



如上图所示，查询优化器工作过程包括：

1）词法分析、语法分析、语义检查；

2）预处理阶段(查询改写等)；

3）查询优化阶段，可详细划分为逻辑优化、物理优化两部分

逻辑优化：把SQL交给查询优化器之后，会去做相应的改写动作。

物理优化：过程是优化器生成获取数据去扫描数据的路径。

4）查询优化器优化依据，来自于代价估算器估算结果(它会调用统计信息作为计算依据);

5）交由执行器执行。

（三）查看和干预执行计划

在MySQL里查看SQL的执行计划直接通过Explain关键词就可以了，或者我们可以添加Extended关键字，它会展示MySQL优化器的逻辑优化改写过程。

1.执行计划

· explain [extended] SQL_Statement

```
mysql> explain select * from payment where rental_id > 1000 order by payment_date;
+----+-------------+---------+------+-----------------+------+---------+------+-------+----------------------------------------------+
| id | select_type | table   | type | possible_keys   | key  | key_len | ref  | rows  | Extra                                        |
+----+-------------+---------+------+-----------------+------+---------+------+-------+----------------------------------------------+
|  1 | SIMPLE      | payment | ALL  | fk_payment_rental | NULL | NULL    | NULL | 15511 | Using where; Using filesort |
+----+-------------+---------+------+-----------------+------+---------+------+-------+----------------------------------------------+
1 row in set (0.00 sec)
```

当我们认为SQL的执行计划不合理时，可以通过适当的手段，强制加索引或者强制驱动表的顺序，通过这种hints方式干预SQL的执行计划。另外MySQL查询优化器的一些关键特性，我们也可以通过控制优化器开关的参数，从而控制优化器相关的行为。

2.优化器开关

· show variables like 'optimizer_switch'

```
mysql> show variables like 'optimizer_switch'\G
*************************** 1. row ***************************
Variable_name: optimizer_switch
        Value: index_merge=on,index_merge_union=on,index_merge_sort_union=on,index_merge_intersection=on,engine
_condition_pushdown=on,index_condition_pushdown=on,mrr=on,mrr_cost_based=on,block_nested_loop=on,batched_key_ac
cess=off,materialization=on,semijoin=on,loosescan=on,firstmatch=on,duplicateweedout=on,subquery_materialization
_cost_based=on,use_index_extensions=on,condition_fanout_filter=on,derived_merge=on
1 row in set (0.01 sec)
```

3.Processlist

另一种观测MySQL行为的常用手段就是Processlist。通过Processlist，我们可以看到当前在MySQL中执行的所有SQL语句，有没有异常的会话或比较特殊的SQL状态。查看会话操作可以通过2种途径：

第一，show [full] processlist;

第二，information_schema.processlist。

这里列出了几种常见的异常行为：

1）Copy to tmp table

出现在某些Alter Table语句的Copy Table操作。

2）Copying to tmp table on disk

由于临时结果集大于tmp_table_size，正在将临时表从内存存储转为磁盘存储以此节省内存。

3）Converting HEAP to MyISAM

线程正在转换内部Memory临时表到磁盘MyISAM临时表。

4）Creating sort index

正在使用内部临时表处理Select查询。

5）Sorting index

磁盘排序操作的一个过程。

6）Sending data

正在处理Select查询的记录，同时正在把结果发送给客户端。

7）Waiting for table metadata lock

等待元数据锁。

常规优化策略

（一）Select优化

1.Order by

Order by查询的两种情况：

1）Using index，是针对查询优化器的两种行为来去区分的。Using index就是说MySQL它可以直接通过索引去返回有序的记录，而不需要去经过额外的排序的操作；

2）Using filesort需要去做额外的排序，在某些特殊的情况下，可能还会出现临时表排序的情况。

优化目标：尽量通过索引来避免额外的排序，减少CPU资源的消耗。

· 主要优化策略：

1）Where条件和Order by使用相同的索引；

2）Order by的顺序和索引顺序相同；

3）Order by 的字段同为升序或降序。

注：当Where条件中的过滤字段为覆盖索引的前缀列，而Order by字段是第二个索引列时，只有Where条件是Const匹配时，才可以通过索引消除排序，而between…and或>?、<?这种Range匹配都无法避免Filesort操作。

当无法避免Filesort操作时，优化思路就是让Filesort的操作更快。

· 排序算法

1）两次扫描算法。两次访问数据，第一步获取排序字段的行指针信息，在内存中排序，第二步根据行指针获取记录。

2）一次扫描算法。一次性取出满足条件的所有记录，在排序区中排序后输出结果集。是采用空间换时间的方式。

注：需要排序的字段总长度越小，越趋向于第二种扫描算法，MySQL通过max_length_for_sort_data参数的值来进行参考选择。

· 优化策略

1）适当调大max_length_for_sort_data这个参数的值，让优化器更倾向于选择第二种扫描算法；

2）只使用必要的字段，不要使用Select *的写法；

3）适当加大sort_buffer_size这个参数的值，避免磁盘排序的出现（线程参数，不要设置过大）。

2.Subquery

对于子查询，一般的优化策略是做等价改写，在MySQL查询优化器中也叫反嵌套。在MySQL里，查询优化器本身也可以做一些简单查询的反嵌套操作，但在绝大部分情况下还是需要去做一些人为的干预。

· Subquery优化总结：

1）子查询会用到临时表，需尽量避免；

2）可以使用效率更高的Join查询来替代。

· 优化策略

等价改写、反嵌套。

如下SQL：

```
select * from customer where customer_id not in (select customer_id from payment)
```

改写形式：

```
select * from customer a left join payment b on a.customer_id=b.customer_id where b.customer_id is null
```


如上图所示，SQL语句用Not In的这样的方式，在子查询里执行Select语句。对于这个SQL语句，直接把Not In改写成Left Join，从而提升它的执行效率。在MySQL里，一般情况下Join的效率比子查询要高。

3.Limit

分页查询，就是将过多的结果在有限的界面上分多页来显示。

其实质是每次查询只返回有限行，翻页一次执行一次。

· 优化目标

1）消除排序；

2）避免扫描到大量不需要的记录。

SQL场景（film_id为主键）：

select film_id,description from film order by title limit 10000,20

此时MySQL排序出前10020条记录后仅仅需要返回第10001到10020条记录，前 10000条记录造成额外的代价消耗。

对于分页查询的优化的策略

· 优化策略一

“覆盖索引”

Alter table film add index idx_lmtest(title,description);

记录直接从索引中获取，效率最高。

仅适合查询字段较少的情况。

· 优化策略二

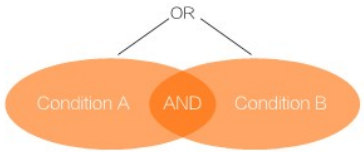
“SQL改写”

select a.film_id,a.description from film a inner join (select film_id from film order by title limit 1000,20) b on a.film_id=b.film_id;

优化的前提是Title字段有索引。

思路是从索引中取出20条满足条件记录的主键值，然后回表获取记录。

4.Or/And Condition



And结果集为关键字前后过滤结果的交集；

Or结果集为关键字前后分别查询的并集；

And条件可以在前一个条件过滤基础上过滤；

Or条件被处理为UNION，相当于两个单独条件的查询；

复合索引对于Or条件相当于一个单列索引。

· 处理策略

1）And子句多个条件中拥有一个过滤性较高的索引即可；

2）Or条件前后字段均要创建索引；

3）为最常用的And组合条件创建复合索引。

（二）Join优化

1.Nested-Loop Join算法

```
for each row in t1 matching range {
  for each row in t2 matching reference key {
    for each row in t3 {
      if row satisfies join conditions, send toclient
    }
  }
}
```

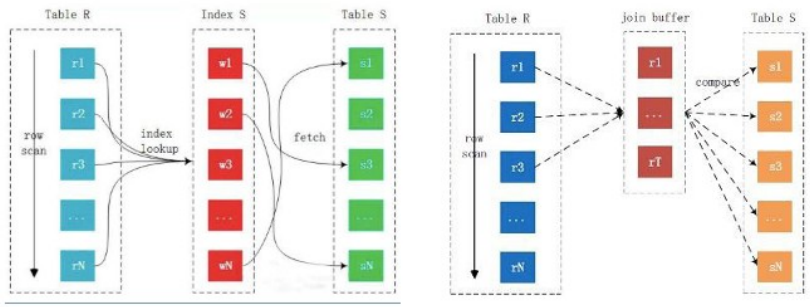
在MySQL里，关于join的典型算法就是Nested-Loop Join，也就是嵌套循环。

如上图所示，T1、T2、T3三张表Join，首先扫描T1表找到匹配条件的行，然后根据T1、T2的关联条件，再扫描 T2表找到匹配条件的的行，T2、T3也做同样的操作。

既然本质是嵌套循环，那么我们主要需要注意两点：

1）关联字段索引：每层内部循环仅获取需要关心的数据。

引申算法：Bloack Nested-Loop。



如上图所示，下方是Bloack Nested-Loop，在MySQL里有一个特性叫join_buffer，当两张表关联，如果不能够通过索引去做关联条件的匹配，这时候就会产生join_buffer的使用。

当SQL的Join语句，执行计划里出现Bloack Nested-Loop时，通常情况下，需要看关联条件是否有索引，或者是其他原因而导致关联条件的匹配没有正常使用到索引。一旦SQL语句执行计划出现Bloack Nested-Loop，绝大部分场景下都意味着SQL执行效率会大幅下降。

2）小表驱动原则：外层循环的结果集尽量小，目的是为了减少循环的次数。

2.关联字段索引的必要性

案例：

```
mysql> select count(*) from film2 a  join film_category2 b on a.film_id=b.film_id;
+----------+
| count(*) |
+----------+
|   512000 |
+----------+
1 row in set (1 min 50.26 sec)

mysql>
mysql> explain  select count(*) from film2 a  join film_category2 b on a.film_id=b.film_id;
+----+-------------+-------+------------+------+---------------+------+---------+------+--------+----------+-------------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows   | filtered | Extra                                                 |
+----+-------------+-------+------------+------+---------------+------+---------+------+--------+----------+-------------------------------------------------------+
|  1 | SIMPLE      | b     | NULL       | ALL  | NULL          | NULL | NULL    | NULL |   4000 |   100.00 | NULL                                                  |
|  1 | SIMPLE      | a     | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 127062 |    10.00 | Using where; Using join buffer (Block Nested Loop)    |
+----+-------------+-------+------------+------+---------------+------+---------+------+--------+----------+-------------------------------------------------------+
2 rows in set, 1 warning (0.00 sec)

mysql>
```

如上图所示，对于这一条Select语句，是两个表Join关联。SQL语句执行计划时，出现了join_buffer，执行计划extra部分就是前面所说的Bloack Nested-Loop。

我们可以看到，通过b表关联访问a表时，Rows是127062，整个访问过程的代价特别大，对于这种场景，优化策略是给关联条件添加索引。如下图所示：

```
mysql>
mysql> alter table film2 add index(film_id);
Query OK, 0 rows affected (0.32 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> explain  select count(*) from film2 a  join film_category2 b on a.film_id=b.film_id;
+----+-------------+-------+------------+------+---------------+---------+---------+-------------------+--------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key     | key_len | ref               | rows   | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+---------+---------+-------------------+--------+----------+-------------+
|  1 | SIMPLE      | b     | NULL       | ALL  | NULL          | NULL    | NULL    | NULL              |   4000 |   100.00 | NULL        |
|  1 | SIMPLE      | a     | NULL       | ref  | film_id       | film_id | 2       | sakila.b.film_id  |    125 |   100.00 | Using index |
+----+-------------+-------+------------+------+---------------+---------+---------+-------------------+--------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)

mysql>  select count(*) from film2 a  join film_category2 b on a.film_id=b.film_id;
+----------+
| count(*) |
+----------+
|   512000 |
+----------+
1 row in set (0.31 sec)
```

后面可以看到，通过b表访问a表时，执行计划里Key使用到了刚才所添加的索引，Rows从127042下降到125。前者执行时间接近两分钟，后者只需要0.31秒，执行效率大幅提升。

3.小表驱动原则

忽略b表的索引，使b表作为驱动表，如下图所示：

```
mysql> explain  select count(*) from film2 a  join film_category2 b ignore index(film_id) on a.film_id=b.film_id;
+----+-------------+-------+------------+------+---------------+---------+---------+-------------------+--------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key     | key_len | ref               | rows   | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+---------+---------+-------------------+--------+----------+-------------+
|  1 | SIMPLE      | b     | NULL       | ALL  | NULL          | NULL    | NULL    | NULL              |   4000 |   100.00 | NULL        |
|  1 | SIMPLE      | a     | NULL       | ref  | film_id       | film_id | 2       | sakila.b.film_id  |    125 |   100.00 | Using index |
+----+-------------+-------+------------+------+---------------+---------+---------+-------------------+--------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)

mysql>
mysql>  select count(*) from film2 a  join film_category2 b ignore index(film_id) on a.film_id=b.film_id;
+----------+
| count(*) |
+----------+
|   512000 |
+----------+
1 row in set (0.31 sec)
```

同样的SQL语句，这里增加忽略索引的hints，目的是为了通过b表做驱动表，可以看到Rows是4000×125，执行时间是0.31秒。

忽略a表的索引，使a表作为驱动表：

```
mysql>
mysql> explain  select count(*) from film2 a ignore index(film_id)  join film_category2 b on a.film_id=b.film_id;
+----+-------------+-------+------------+------+---------------+---------+---------+-------------------+--------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key     | key_len | ref               | rows   | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+---------+---------+-------------------+--------+----------+-------------+
|  1 | SIMPLE      | a     | NULL       | ALL  | NULL          | NULL    | NULL    | NULL              | 127042 |   100.00 | NULL        |
|  1 | SIMPLE      | b     | NULL       | ref  | film_id       | film_id | 2       | sakila.a.film_id  |      4 |   100.00 | Using index |
+----+-------------+-------+------------+------+---------------+---------+---------+-------------------+--------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)

mysql>
mysql>  select count(*) from film2 a ignore index(film_id)  join film_category2 b on a.film_id=b.film_id;
+----------+
| count(*) |
+----------+
|   512000 |
+----------+
1 row in set (0.53 sec)
```



```
mysql>  select count(*) from film2;
+----------+
| count(*) |
+----------+
|   128000 |
+----------+
1 row in set (0.05 sec)

mysql>  select count(*) from film_category2;
+----------+
| count(*) |
+----------+
|     4000 |
+----------+
1 row in set (0.00 sec)
```

这时b表成了被驱动表，Rows为127042×4，总行数接近的情况下后者驱动表行数有明显增加。

这两个SQL语句做关联时，无论通过a表还是b表驱动，最后关联时都通过索引进行数据检索。但是由于驱动表的大小问题，导致了执行效率的不同，后面一条语句执行的时间是0.53秒，比前者慢了一倍左右。

（三）Insert优化

关于Insert的插入优化策略主要有2种：

1）优化策略一：

“减少交互次数”

如批量插入语句：

```
insert into test values(1,2,3);
insert into test values(4,5,6);
insert into test values(7,8,9);
...
```

可改写为如下形式：

```
insert into test values(1,2,3),(4,5,6),(7,8,9) ···
```

2）优化策略二：

“文本装载方式”

通过LOAD DATA INFILE句式从文本装载数据，通常比Insert语句快20倍。

常规优化策略

MySQL查询优化

1.关于MySQL的查询优化目的和目标：

优化的目的是让资源发挥价值；

SQL和索引是调优的关键，往往可以起到“四两拨千斤”的效果。

2.关于优化的流程和思路：

充分了解核心指标，并构建完备的监控体系，这是优化工作的前提；

SQL优化的原则是减少数据访问及计算；

常用的优化方法主要是调整索引、改写SQL、干预执行计划。

3.关于MySQL的核心概念及原理：

Innodb的表是典型的IOT，数据本身是B+ tree索引的叶节点；

扫描二级索引可以直接获取数据，或者返回主键ID；

优化器是数据库的大脑，我们要了解优化器，并观测以及干预MySQL的行为。