MySQL高级篇

101、字符集的修改与底层原理说明

- **查看Mysql字符集**：`show variables like 'character%';`
- **修改字符集**：MySQL5.7默认数据库和客户端字符集为Latin1，不支持中文，因此需要修改MySQL配置文件达到支持中文的目的。若数据库已经运行一段时间，部分表中有数据，也可单独修改数据库或表的字符集，使用alert

```
vim /etc/my.cnf

# 在配置文件中修改或添加对应的字符集
# 服务器级别字符集
character_set_server=utf8
# 数据库级别字符集
character_set_database=utf8
# 服务器解码请求时字符集
character_set_client=utf8
# 服务器解码请求时会把字符串从client转为connection
character_set_connection=utf8
# 服务器向客户端返回时字符集
character_set_results=utf8
```

- **字符集级别**：**服务器级别、数据库级别、表级别、列级别。**前两者由数据库配置文件指定，后两者继承上一级，也可以单独指定。

102、比较规则_请求到响应过程中的编码与解码过程

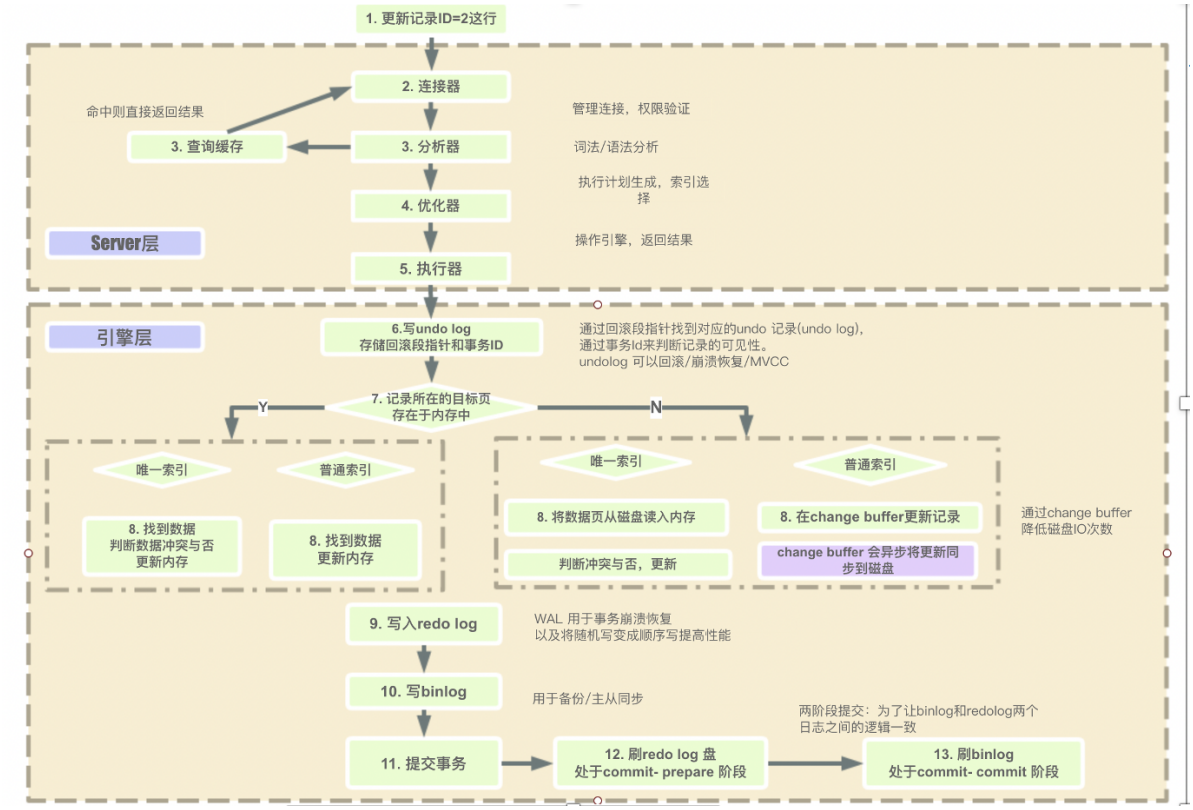
- **字符集**：utf8表示一个字符用1~4个字节表示。MySQL中的utf8其实是utf8mb3，只是用1~3个字符，utfmb4才符合utf8的真正含义。
- **查看数据库支持的字符集和比较规则**：`show charset;`
- **比较规则后缀含义**：ai表示不区分重音，as表示区分重音，ci表示不区分大小写，cs表示区分大小写，bin表示以二进制进行比较。utf8_unicode_ci和utf8_general_ci相对比，前者支持其他语言，后者校对速度块，但准度不高。

103、SQL大小写规范与sql_mode的设置

- **系统大小写区别**：MYSQL中函数名、列名和关键字不区分大小写，Windows数据库名、表名不区分大小写，Linux对此大小写敏感。
- **SQL语法检查规则**：sql_mode模式用于控制对SQL语法的检查，常用的STRICT_TRANS_TABLES严格模式规定，如果一个值不能插入到一个事务表中，则中断当前的操作。

110、SQL执行流程

- 由于缓存命中率很低，8.0中已停用



113、设置表的存储引擎、InnoDB与MyISAM的对比

- **概念**：为了管理方便，将**连接管理、语法解析、查询优化**等不涉及真实数据存储的功能划分为server层，真实存取数据功能划分为存储引擎。`show engines;`查询所支持的存储引擎。
- **查询或修改默认存储引擎**：`show variables like '%storage_engine';`，`set DEFAULT_STORAGE_ENGINE=MyISAM;`或修改`my.cnf`。同时，也可以单独在创建表的时候指定存储引擎或后续修改表的存储引擎。
- **InnoDB引擎**：具备外键支持的事务存储引擎，5.5之后的默认引擎，被设计用于处理大量短期事务，是处理巨大数据量的最大性能设计。相比较于MyISAM，写的处理效率低一点，且缓存索引同时还要缓存真实数据，对内存要求较高。
- **MyISAM引擎**：主要的非事务处理引擎。提供大量特性，包括全文索引、压缩、空间函数GIS等。相比较于InnoDB，不支持事务、行级锁、外键，最重要的缺陷是崩溃后无法安全恢复。优势是访问速度快，且包含常数记录数据总数，只适用于以读为主的业务。
- **InnoDB与MyISAM区别**：
  - InnoDB支持事务，MyISAM不支持，对于InnoDB每一条SQL语言都默认封装成事务，自动提交，这样会影响速度，所以最好把多条SQL语言放在begin和commit之间，组成一个事务；
  - InnoDB支持外键，而MyISAM不支持。对一个包含外键的InnoDB表转为MYISAM会失败；
  - InnoDB是聚集索引，使用B+Tree作为索引结构，数据文件是和（主键）索引绑在一起的（表数据文件本身就是按B+Tree组织的一个索引结构），必须要有主键，通过主键索引效率很高。但是辅助索引需要两次查询，先查询到主键，然后再通过主键查询到数据。因此，主键不应该过大，因为主键太大，其他索引也都会很大。
  - MyISAM是非聚集索引，也是使用B+Tree作为索引结构，索引和数据文件是分离的，索引保存的是数据文件的指针。主键索引和辅助索引是独立的。也就是说：InnoDB的B+树主键索引

的叶子节点就是数据文件，辅助索引的叶子节点是主键的值；而MyISAM的B+树主键索引和辅助索引的叶子节点都是数据文件的地址指针。
    - InnoDB不保存表的具体行数，执行select count(*) from table时需要全表扫描。而MyISAM用一个变量保存了整个表的行数，执行上述语句时只需要读出该变量即可，速度很快（注意不能加有任何WHERE条件）；
    - 5.7以后的InnoDB支持全文索引了
    - MyISAM表格可以被压缩后进行查询操作
    - InnoDB支持表、行(默认)级锁，而MyISAM支持表级锁。InnoDB的行锁是实现在索引上的，而不是锁在物理行记录上。潜台词是，如果访问没有命中索引，也无法使用行锁，将要退化为表锁。
    - InnoDB表必须有唯一索引（如主键）（用户没有指定的话会自己找/生产一个隐藏列Row_id来充当默认主键），而Myisam可以没有

118、聚簇索引、二级索引与联合索引的概念

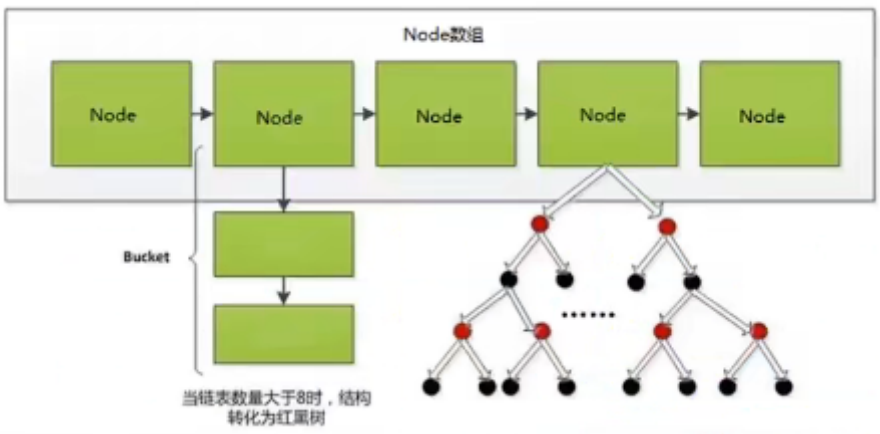
- **聚簇索引**：一种数据存储方式，即所有的用户记录都存在叶子节点，如InnoDB的索引即属于这种，主键索引与数据存在一起。**为了充分利用聚簇索引的特性，innoDB表的主键尽量选用有序的顺序ID，UUID、MD5、HASH、字符串等作为主键将无法保证数据的顺序增长，进而影响索引树生成状态。**

119、InnoDB中B+树注意事项，MyISAM的索引方案

- **B+树的形成过程**：
    - 为一个表创建B+树索引时，会为这个索引创建一个根节点页面，最开始表中没有数据的时候，每个B+树索引对应的根节点也没有用户数据
    - 当插入用户记录时，会把用户记录存储到根节点中
    - 当根节点中的可用空间满了之后，会将根节点的所有记录复制到新的页上，然后对其进行页分裂，也就是形成两个装有记录的新页，然后新插入的记录根据键值大小分配到其中一页里面，而根节点也升级为存储目录项记录的页
- **MyISAM的索引方案**：MyISAM中也是用B+树，但是索引跟数据是分开存储的，索引只存储数据的地址记录

120、Hash索引、AVL树、B树与B+树对比

- **效率标准**：为了减少索引在内存中的占用，数据库索引是存储在外部磁盘上的，当利用索引查询时，不可能一次性将所有索引都加载到内存上，只能逐一加载，那么衡量MYSQL查询效率的标准就是**磁盘IO次数。**
- **Hash索引**：使用hash索引效率极高，平均查找效率为O(1)，基本上一次检索可找到数据，而B+树需要自顶向下依次查找，需要多次IO操作。**hash索引仅适用于memory存储引擎。**



- **Hash索引效率高，但为什么大多数索引还是设计成树形呢?**
    - Hash索引仅能满足=，in，<>查询，如果使用范围查询，将退化为O(n)。而树形由于自身有序，因此仍然可以保持O(log2N)
    - Hash索引数据是没有顺序的，因此在order

by的情况下，使用hash索引还要对数据重新排序
    - 对于联合索引的情况，hash是将联合索引键合并后一起计算的，无法对单独一个键或者几个键进行查询
    - 对于等值查询来说，hash索引一般效率较高。但存在索引列重复值较多时，产生hash冲突，需要遍历bucket中的行指针来比较，效率降低。因此hash索引一般不用于重复值多的列
- **AVL树**：平衡二叉搜索树，左右两子树的高度差不超过1。根据二叉树性质，平均搜索时间复杂度为O(log2N)，因此在此基础上，考虑改成M叉树，使得树变矮
- **B-Tree**：多路平衡查找树，每一个节点最多可以包括M个子节点（即M阶）。每个磁盘块包含关键字和子节点的指针。
    - 根节点的子节点数目范围在[2, M]
    - 每个中间节点包含k-1个关键字和k个子节点，k范围[ceil(M/2)]，即M/2向上取整
    - 叶子节点包括k-1个关键字，k的范围[ceil(M/2), M]，所有叶子节点位于同一层
    - 关键字有序
- **B+Tree**：修改的多路平衡查找树
    - 有k个子树的节点有k个关键字，不同于B-Tree只有k-1个
    - 非叶子节点的关键字会同时存在于子节点中，且是子节点中关键字的最值
    - 非叶子节点仅用于索引，不保存数据
    - 所有关键字都在叶子节点出现，叶子节点形成有序链表
- **B-Tree对比B+Tree**：B+树的查询效率更稳定，且查询效率更高（因为B+树比B树更矮胖），且由于所有关键字都在B+树的叶子节点上，查询范围的效率更高。
- **R树**：仅支持geometry数据类型，用于查找高维空间搜索问题

128、全文索引

- 使用参数FULLTEXT可设置全文索引，只适用于CHAR、VARCHAR、TEXT类型数据上，查询数据量较大的字段时，可以提高效率（不使用like，而是match+against）。mysql5.7之后支持中文分词，但随着大数据时代到来，逐渐被专业的solr、ElasticSearch所取代

130、MySQL8 索引新特性：降序索引、隐藏索引

- **降序索引**：MySQL8之前建立的索引默认为升序索引，在使用order by desc时，还需要额外对数据进行排序（Using Filesort）。MySQL8之后支持声明索引为降序索引，减少外部排序过程。`create table t1 (a bigint, b int, index idx_a_b(a, b desc));`
- **隐藏索引**：MySQL8之后可以将索引隐藏，即使force index也无法使用。常用于验证索引是否对SQL有帮助，或将索引隐藏，确定对数据查询无影响后再删除。`alert table t1 alert index idx_a_b VISIBLE/INVISIBLE;`

131、适合创建索引的11种情况

- 字段的数值有唯一性的限制
- 频繁作为where查询条件的字段
- 经常group by或order by的列，若同时使用group by和order by，可考虑使用联合索引，按照先group by，后order by的顺序建立索引，其中order by部分在MySQL8之后可使用降序索引
- update和delete的where条件字段
- distinct字段需要创建索引
- 多表连接建立索引时，主要对where条件列创建索引，并且可对用于连接的字段建立索引，要求该字段在多个表中类型一致
- 使用列的类型小的创建索引，类型小即占用字节少，一个页可存放的索引会更多

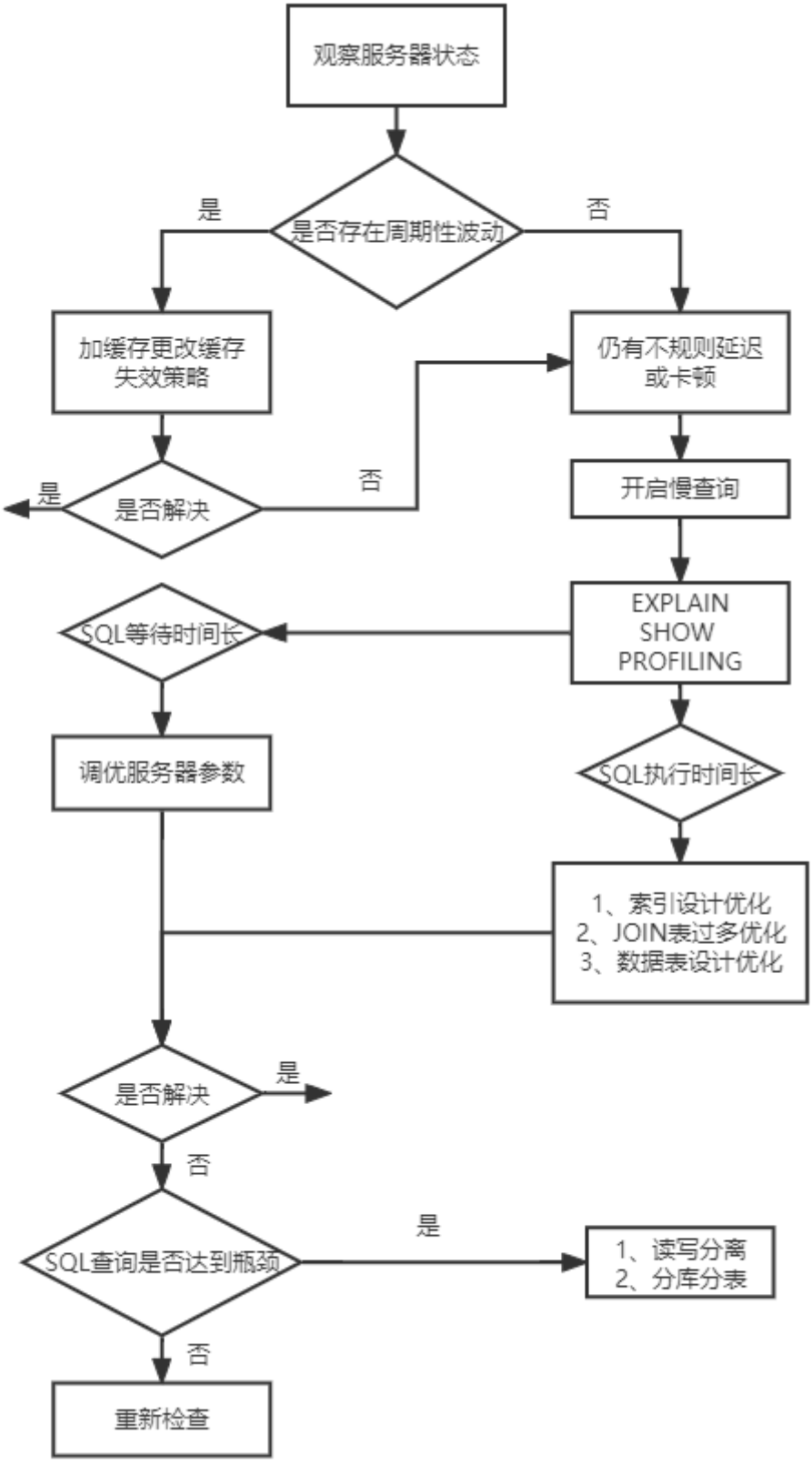
- 使用字符串前缀建立索引（即前缀索引），而不是整个字符串。可以节约空间，同时减少字符串比较时间。
    - 字符串前缀的区分度，`select count(distinct left(字段，长度)) / count(*) from table;`。
    - 由于使用的是前缀索引，在order by该字段时，在指定长度都相同，但是后面不相同的字符串，仍然需要外部排序
- 区分度高的列适合建立索引，将列中不重复数据的个数称为基数，若基数大则表示区分度高
    - 使用`select count(distinct a) / count(*) from table;`计算区分度，一般超过33%则认为较为高效
    - 联合索引时可优先将区分度高的列放在左边
- 使用最频繁的列放在联合索引的左侧，可以减少一些索引的创建，且根据最左前缀原则，联合索引的使用率会达到提升
- 在多个字段都要建立索引时，联合索引由于单列索引

133、不适合创建索引的7种情况

- 不在where条件中的字段
- 数据量小的表不适用索引，全表扫描或许更快
- 有大量重复数据即区分度低的字段不适合创建索引
- 避免对频繁更新的表创建过多索引，频繁更新的字段不一定要创建索引，更新时需要对索引进行维护，会影响效率
- 不建议使用无序的值作为索引，按照B+树的特性，将造成页分裂频繁发生
- 定期删除无用索引
- 不要定义冗余或重复索引，index_abc, index_ab, index_a，后两个都属于前者的冗余索引

134、数据库优化步骤&查看系统性能参数

- **数据库优化步骤**



- **查看系统性能参数 SHOW [GLOBAL|SESSION] STATUS LIKE '参数'：**
  - **Connections**：连接MySQL服务器的次数
  - **Uptime**：MySQL服务器的上线时间
  - **Slow_queries**：慢查询的次数
  - **innodb_rows_read/inserted/updated/deleted**：查询返回/插入/更新/删除的行数
  - **Com_select/insert/update/delete**：查询/插入/更新/删除的操作次数

- **统计SQL的查询成本 last_query_cost**：SQL查询前需要确定执行计划，MySQL会计算每个执行计划的成本， 并选择最低成本的执行。执行完SQL后可以通过查看会话的last_query_cost查看查询成本，`show session status like 'last_query_cost';`。这是评价一个查询的执行效率的指标，对应的是SQL查询需要读取的页的数量。

135、慢查询日志分析、SHOW PROFILE查看SQL执行成本

- MySQL中存在慢查询日志，用于记录响应时间超过阈值（long_query_time，默认10s，可修改）和扫描过的最少记录数超过阈值（min_examined_row_limit， 默认0）的语句。慢查询日志默认关闭，建议仅调休时开启。`show variable like 'slow_query_log%';`查看当前慢查询日志是否开启。`set global slow_query_log='ON';`开启慢查询日志，加global表示全局，不加表示当前会话。
- **查看慢查询次数**：`show status like 'slow_queries%';`
- **慢查询日志分析工具 mysqldumpslow**：-s参数用于确定顺序，-t用于返回前面多少条数据，-g表示正则匹配
- **show profile**：用于分析当前会话中SQL做了什么，执行的消耗资源的情况。默认关闭，`show variable like 'profiling';`查看当前会话是否开启，`set profiling='ON'`开启分析工具。`show profiles;`查看查询的SQL语句以及消耗的时间。`show profile;`查看最近一次查询的消耗，同样作用的有`show profile for query 数字ID;`。除了时间开销之外，还可以查看其他部分的开销，`show profile cpu, block io;`。
- **通过show profile需要注意的状态**：若出现以下四种状态的一种，则SQL语句需要优化
  - converting HEAP to MyISAM：查询结果太大，内存不够，数据往磁盘上搬
  - Creating tmp table：创建临时表，会拷贝数据到临时表上，用完后再删除
  - Copying to tmp table on disk：把内存中的临时表复制到磁盘上
  - locked：锁

137、EXPLAIN中字段剖析

- 定位了慢查询sql后，就可以使用EXPLAIN或者DESCRIBE工具做针对性的分析查询
- **select_type**：一条大的查询语句里面可能包括多个select小查询语句，每个select语句可能包含多张表，每张表对应着执行计划输出的一条记录。通过select_type可以得知小查询在整个查询中扮演的角色。
  - SIMPLE：查询语句中不包含UNION或者子查询
  - PRIMARY：对于包含UNION或者子查询的语句，最左边的查询就是PRIMARY
  - UNION：对于包含UNION或UNION ALL的大查询，非PRIMARY的其他查询
  - UNION RESULT：对于使用临时表来完成UNION查询的去重工作，也可理解为UNION的结果，针对该临时表的type就是UNION RESULT
  - SUBQUERY：包含子查询的语句，且该子查询是不相关子查询
  - DEPENDENT SUBQUERY：包含子查询的语句，且该子查询是相关子查询
  - DFPENDENT UNION：包含UNION或UNION ALL的大查询中，除了最左的小查询，其余小查询若依赖于外部查询，则为DEPENDENT UNION
  - DERIVED：对于包含派生表（即FROM的临时表）的查询，该派生表对应的子查询
  - MATERIALIZED：查询优化器在包含子查询的语句时，选择将子查询物化后与外层查询进行连接查询时，该子查询type为MATERIALIZED

- **type**：执行计划的一条记录代表MySQL对某个表执行查询时的访问方法或者访问类型。一般优化要求至少达到range级别，最好是const。结果值从好到坏为 system > const > eq_ref > ref > fulltext > ref_or_null > index_merge > unqiue_subquery > index_subquery > range > index < ALL
    - system：当表中只有一条记录且该表使用的存储引擎的统计数据是精确的，例如MyISAM，memory
    - const：当根据主键或者唯一二级索引列于常数进行等值匹配时
    - eq_ref：连接查询时，被驱动表是通过主键或者唯一二级索引列等值匹配的方式进行访问的
    - ref：当通过普通的二级索引列与常量进行等值匹配时来查询某个表
    - ref_or_null：当通过普通的二级索引列与常量进行等值匹配时或者该列可为NULL
    - unique_subquery：针对一些包含in子查询的语句，若优化器决定将in子查询转化为exists子查询，并且该子查询可以使用到主键进行等值匹配时
    - range：当使用索引获取某些范围区间的记录时，包括in，<，>
    - index：当使用索引，但是需要扫描全部的索引记录
    - all：全表扫描
- **key_len**：实际使用到的索引长度，值越大越好，尤其对于联合索引
- **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**：预估需要读取的记录条数，越小越好
- **filtered**：某个表经过搜索条件过滤后剩余记录条数的百分比，对于单表而言没太大意义，更重要的在于连接查询中驱动表对应的filtered值，它决定了被驱动表要执行的次数（rows * filtered）

139、EXPLAIN的4种格式与查看优化器重写SQL

- 直接EXPLAIN，EXPLAIN FORMAT=JSON，EXPLAIN FORMAT=TREE，可视化输出（使用MySQL Workbench）
- EXPLAIN之后，执行SHOW WARNINGS语句查看扩展信息，code=1003时查看SQL被重写后的样子

140、trace分析优化器执行计划与Sys schema视图的使用

- OPTIMIZER_TRACE是跟着优化器做出的各种决策，包括访问表的方法、开销计算、转换等等，并且将追踪记录到INFORMATION_SCHEMA.OPTIMIZER_TRACE表中
- 此功能默认关闭，可手动开启，并将格式设置为json格式，同时设置trace最大可使用内存大小，避免展示不完全

```
SET optimizer_trace="enabled=on", end_markers_in_json=on;
SET optimizer_trace_max_mem_size=1000000;
```

- MYSQL监控分析视图sys schema：将performance_schema和information_schema中的数据以更容易理解的方式归纳为视图

141、索引失效的11种情况

- 等值匹配百分百使用索引
- 最左前缀原则：联合索引必须从最左边一列开始出现，且不得跳过中间某一列

- 主键插入顺序：主键最好选择自然增长类型的，否则插入数据时很容易出现页分裂的情况，影响性能
- 计算、函数使索引失效：将会导致全表扫描，将值全部取出进行操作后再过滤
  - name like 'abc%'可以使用索引，但是left(name, 3) = 'abc'不行
  - money = 90000可以使用索引，但是money + 1 = 90001不行
- 类型转换使索引失效：使用错误类型值过滤，如name = 123
- 范围条件右边的列索引失效：
  - age = 30 and classId > 20 and name = 'abc'，若索引按照最左顺序是age, classId, name。则会导致name部分索引内容失效，导致索引真实使用部分极少。应当按照age, name, classId顺序建立索引，并且sql语句应尽量将范围语句放最后
- 不等于!=或者<>使索引失效
- is null可以使用索引，但是is not null不行，同理not like也不行
- like以%开头的无法使用索引
- OR前后的列必须都是索引列，否则会采用全表扫描
- 不同字符集进行转换，导致比较时也会使索引失效

143、外连接与内连接的查询优化

- **左外连接**：根据左外连接的概念，其中驱动表的记录是全部都要的，但是被驱动表记录是有选择性的要，因此**可以在被驱动表的条件字段上建立索引**，减少笛卡儿积。
- **内连接**：对于内连接来说，查询优化器会自主选择其中一个表作为驱动表，另一个作为被驱动表。此时，
  - 若连接表中只有一个表的条件字段有索引，则会将该表作为被驱动表。
  - 若两个表的条件字段都有索引，则会选择数据量小的表作为驱动表，即“小表驱动大表”。
  - **结论**：**当只能给其中一个表建立索引时，应该将数据量大的表作为被驱动表，建立索引**
- **驱动表与被驱动表**：
  - 对于左连接，当不存在where条件时，前置表为驱动表，后置表为被驱动表；当存在where条件时，where中所使用的字段所在表为驱动表
  - 对于内连接，则优先选择小数据量的表作为驱动表
  - 对于不方便分析的情况，建议直接explain

145、子查询优化与排序优化

- **子查询执行效率不高**：
  - 执行子查询时，MySQL会为内层查询语句创建一个临时表，查询完毕后再撤销这些临时表，将消耗过多的CPU和IO资源，产生大量的慢查询
  - 子查询的结果集存储的临时表，是不存在索引的，查询性能受到一定的影响
  - 对于结果集比较大的子查询，对查询性能的影响也越大
- 在MySQL中，**可以用join连接来替代子查询**，不会生成临时表，且可以使用索引。**尽量不要使用not in或者not exists，用left join xx on xx where xx is null。**
- **排序优化**：
  - **SQL中，where子句中使用索引是为了避免全表扫描，在order by子句中使用索引是为了避免使用FileSort排序。**FileSort一般在内存中进行排序，占用CPU较多，若待排序结果较大，会产生临时文件IO到磁盘进行排序的情况，效率较低
  - 若where和order by是相同的列则使用单列索引，若不同则使用联合索引

- 若无法使用索引时，需要对FileSort进行调优，调整例如sort_buffer_size的参数大小

146、GROUP BY优化、分页查询优化

- group by与order by一杨，而也可以使用索引，当无法使用索引时，可调整max_length_for_sort_data和sort_buffer_size的参数大小
- **分页查询优化**：
  - 例如select * from a limit 200000, 10;将会排序前200000条记录，再返回200000-200010记录，前面的记录将被全部舍弃，查询代价十分大
  - **优化一**：在索引上完成排序分页操作，根据主键关联回原表查询所需要的其他列内容。`select * from a, (select id from a order by id limit 200000,10) b where a.id = b.id;`
  - **优化二**：适用于主键自增的表，把limit分页查询转换成条件查询。`select * from a where id > 200000 limit 10;`

147、优先考虑覆盖索引的使用

- **覆盖索引概念**：由于数据库也可以通过索引找到一个列的数据，因此有时候不必读取整个行即可达到需求。当通过读取索引就可以得到想要的数据时，一个索引包含了满足查询结果的数据就叫做覆盖索引。或者说建索引的字段（包括主键）正好是覆盖查询条件中所涉及的字段，如select、join、where中用到的所有列。

149、其他查询优化策略

- **EXISTS和IN的区分**：使用EXISTS还是IN要看表的大小，选择标准即为“小表驱动大表”。比较`select * from A where id in (select id from B);`与`select * from A where exists (select id from B where B.id = A.id);`
  - 当A小于B时，用EXISTS。因为EXISTS相当于外表循环。

```
for i in A
    for j in B
        if j.id == i.id ...
```

  - 当A大于B时，用IN。

```
for i in B
    for j in A
        if j.id == i.id ...
```

- **COUNT(*)与COUNT(字段)效率**：
  - 如果是对所有结果进行统计，则COUNT(1)和COUNT(*)本质上没有差别
  - COUNT(字段)会尽量采用二级索引来统计，对于COUNT(1)和COUNT(*)而言，若不需要查找具体的行，则系统也会采用占用空间更小的二级索引
  - **结论**：执行效率上COUNT(*)和COUNT(1)没区别，COUNT(字段)效率同样。真正的区别在于COUNT(字段)会统计非空字段