MySQL执行计划及SQL优化

题目标签

学习时常：30分钟

题目难度：初级

知识点标签：explain、sql优化、索引、sql性能问题

题目描述

MySQL执行计划及SQL优化

1.SQL语句表头运行一个explain时，执行后所显示的表头字段如下：

```
1 | id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra
```

- id : select查询的序列号，包含一组数字，表示查询中执行select子句或操作表的顺序
  - id相同：执行顺序由上至下
  - id不同：如果是子查询，id的序号会递增，id值越大优先级越高，越先被执行
  - id相同又不同（两种情况同时存在）：id如果相同，可以认为是一组，从上往下顺序执行；在所有组中，id值越大，优先级越高，越先执行
- select_type : 查询的类型，主要是用于区分普通查询、联合查询、子查询等复杂的查询
  - SIMPLE：简单的select查询，查询中不包含子查询或者union
  - PRIMARY：查询中包含任何复杂的子部分，最外层查询则被标记为primary
  - SUBQUERY：在select 或 where列表中包含了子查询
  - DERIVED：在from列表中包含的子查询被标记为derived（衍生），mysql或递归执行这些子查询，把结果放在零时表里
  - UNION：若第二个select出现在union之后，则被标记为union；若union包含在from子句的子查询中，外层select将被标记为derived
  - UNION RESULT：从union表获取结果的select
- type : 访问类型，sql查询优化中一个很重要的指标，结果值从好到坏依次是
  - system > const > eq_ref > ref > fulltext > ref_or_null > index_merge > unique_subquery > index_subquery > range > index > All

    > 一般来说，好的sql查询至少达到range级别，最好能达到ref

    - system：表只有一行记录（等于系统表），这是const类型的特例，平时不会出现，可以忽略不计
    - const：表示通过索引一次就找到了，const用于比较primary key 或者 unique索引。因为只需匹配一行数据，所有很快。如果将主键置于where列表中，mysql就能将该查询转换为一个const
    - eq_ref：唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配。常见于主键或 唯一索引扫描。

    > 注意：ALL全表扫描的表记录最少的表

    - ref：非唯一性索引扫描，返回匹配某个单独值的所有行。本质是也是一种索引访问，它返回所有匹配某个单独值的行，然而他可能会找到多个符合条件的行，所以它应该属于查找和扫描的混合体

        - range：只检索给定范围的行，使用一个索引来选择行。key列显示使用了那个索引。一般就是在where语句中出现了bettween、<、>、in等的查询。这种索引列上的范围扫描比全索引扫描要好。只需要开始于某个点，结束于另一个点，不用扫描全部索引
        - index：Full Index Scan，index与ALL区别为index类型只遍历索引树。这通常为ALL块，应为索引文件通常比数据文件小。（Index与ALL虽然都是读全表，但index是从索引中读取，而ALL是从硬盘读取）
        - ALL：Full Table Scan，遍历全表以找到匹配的行
- possible_keys : 查询涉及到的字段上存在索引，则该索引将被列出，但不一定被查询实际使用
- key : 实际使用的索引，如果为NULL，则没有使用索引。查询中如果使用了覆盖索引，则该索引仅出现在key列表中
- key_len : 表示索引中使用的字节数，查询中使用的索引的长度（最大可能长度），并非实际使用长度，理论上长度越短越好。key_len是根据表定义计算而得的，不是通过表内检索出的
- ref : 显示索引的那一列被使用了，如果可能，是一个常量const。
- rows : 根据表统计信息及索引选用情况，大致估算出找到所需的记录所需要读取的行数
- Extra : 不适合在其他字段中显示，但是十分重要的额外信息
    - Using filesort： mysql对数据使用一个外部的索引排序，而不是按照表内的索引进行排序读取。也就是说mysql无法利用索引完成的排序操作成为“文件排序”
    - Using temporary：使用临时表保存中间结果，也就是说mysql在对查询结果排序时使用了临时表，常见于order by 和 group by
    - Using index：表示相应的select操作中使用了覆盖索引（Covering Index），避免了访问表的数据行，效率高。如果同时出现Using where，表明索引被用来执行索引键值的查找如果没用同时出现Using where，表明索引用来读取数据而非执行查找动作。
      覆盖索引（Covering Index）：也叫索引覆盖。就是select列表中的字段，只用从索引中就能获取，不必根据索引再次读取数据文件，换句话说查询列要被所建的索引覆盖。

    > 如需使用覆盖索引，select列表中的字段只取出需要的列，不要使用select *
    > 如果将所有字段都建索引会导致索引文件过大，反而降低crud性能

    - Using where： 使用了where过滤
    - Using join buffer： 使用了链接缓存
    - Impossible WHERE：where子句的值总是false，不能用来获取任何元祖
    - select tables optimized away：在没有group by子句的情况下，基于索引优化MIN/MAX操作或者对于MyISAM存储引擎优化COUNT（*）操作，不必等到执行阶段在进行计算，查询执行计划生成的阶段即可完成优化
    - distinct：优化distinct操作，在找到第一个匹配的元祖后即停止找同样值得动作

2.利用执行计划优化SQL语句常见的表访问方式

- TABLE ACCESS FULL(全表扫描)
    1. 读取表中所有的行，并检查每一行是否满足SQL语句中的Where限制条件；
    2. 可以使用多块读（即一次I/O读取多块数据块）操作，提升吞吐量

    > 使用建议：数据量太大的表不建议使用全表扫描，除非本身需要取出的数据较多，占到表数据总量的5%-10%或以上。

- TABLE ACCESS BY ROWID(通过ROWID存储)
    1. ROWID是由Oracle自动加在表中每行最后的一列伪列，既然是伪列，就说明表中并不会物理存储ROWID的值。
    2. 我们可以像使用其他列一样使用它，只是不能对该列的值进行增删改操作。

3. 一旦一行数据插入后，则其对应的ROWID在该行的生命周期也是唯一的，及时发生行迁移，该行的ROWID值也不变。
4. 行的ROWID指出了该行坐在的数据文件、数据块以及行在改块中的位置，所以通过ROWID可以快速定位到目标数据上，这也是Oracle中存取当行数据最快的方法。

- TABLE ACCESS BY INDEX SCAN（索引扫描）
  1. 在索引块中，既存储每个索引的键值，也存储具有该键值行的ROWID。
  2. 索引扫描分为两步：
     - 扫描索引得到对应的ROWID
     - 通过ROWID定位到具体的行读取数据

3.相关基本知识

1. 索引扫描（以ORACLE进行分析）
   - INDEX UNIQUE SCAN(索引唯一扫描)
     - 针对唯一性索引的扫描，每次之多只返回一条记录
     - 表中某字段勋在UNIQUE、PRIMARY KEY约束时，ORACLE常实现唯一性扫描
   - INDEX RANGE SCAN(索引范围扫描)
     - 使用一个索引存取多行数据
     - 发生索引范围扫描的三种情况
       - 在唯一索引列上使用范围操作符(如:>,<,<>,>=,<=,between)
       - 在组合索引上，只使用部分列进行查询（查询时必须包含前导列，否则会进行全表扫描）
       - 对非唯一索引列上进行的任何查询
   - INDEX FULL SCAN(索引全扫描)
     - 进行全索引扫描时，查出的数据都必须从索引中可以直接得到（注意全索引扫描只有在CBO模式下才有效)
   - INDEX FAST FULL SCAN(索引快速扫描)
     - 扫描索引中的所有的数据块，与INDEX FULL SCAN类似，但是一个显著的区别是它不对查询出的数据进行排序（即数据不是以排序顺序被返回）
   - INDEX SKIP SCAN(索引跳跃扫描)

2.总结

- 利用执行计划对SQL语句进行优化的过程，就是提高索引的命中率过程，其主要目的是降低字节开销和基数，节约耗费和CPU耗费，提升整体执行效率。
  - 针对优化方法不一定，要视具体情况而定，尽可能提高执行结果的返回速度。

SQL优化

sql需要优化的原因

性能低，执行时间长，等待时间太长，SQL语句欠佳（连接查询），索引失败，服务器参数设置不合理（缓冲、线程数）

sql优化，主要就是优化索引

- 索引：相当于书的目录。
- 索引：index是帮助MYSQL高效获取数据的数据结构。索引是数据结构（树：B树(默认)）
- 索引的弊端
  - 索引本身很大，可以存放在内存/硬盘（通常为硬盘）
  - 索引不是所有情况均适用：

        - 数据量少
        - 频繁更新的字段
        - 很少使用的字段
    - 所有索引会降低增删改的效率
- 优势
    - 提高查询效率（降低IO的使用率）
    - 降低CPU使用率 （因为B树索引本身就是一个排好序的结构，因此在排序时可以直接使用）

索引

- 分类：
    - 主键索引：不能重复 不能为Null
    - 单值索引：单列；一个表可以有多个单值索引
    - 唯一索引：不能重复 可以是Null
    - 复合索引：多个列构成的索引
- 创建索引：
    - 创建方式一：create 索引类型 索引名 on 表（字段）
        - 单值：create index 索引名 on 表(字段)
        - 唯一：create unique index 索引名 on 表(字段)
        - 复合：create index 索引名 on 表(字段1，字段2)
    - 创建方式二：alert table 表名 add index 索引名(字段)
        - 单值：alert table 表名 add index 索引名(字段)
        - 唯一：alert table 表名 add unique index 索引名(字段)
        - 复合：alert table 表名 add index 索引名(字段1，字段2)

    > 注意：如果一个字段是primary key,则该字段默认是主键索引

- 删除索引：drop index 索引名 on 表名
- 查询索引：show index from 表名

SQL性能问题

- 分析SQL执行计划：explain 可以模拟SQL优化器执行SQL语句，从而让开发人员知道自己编写的SQL状况
    - id
        - id值相同，从上往下顺序执行
        - 表的执行顺序，因数量的个数改变而改变的原因：笛卡尔积

        > 数据小的表优先查询

        - id值不相同，id值越大越优先查询

        > 本质：在嵌套子查询时，先查内层 再查外层

        - id值有相同，有不相同：id值越大越优先查询，id值相同，从上往下顺序执行
    - select_type
        - PRIMARY：包含子查询SQL中的主查询（最外层）
        - SUBQUERY：包含子查询SQL中的子查询（非最外层）
        - simple:简单查询（不包含子查询，union）
        - derived:衍生查询(使用到了临时表)
            - 在from子查询中只有一张表
            - 在from子查询中，如果有table union table2,则table1就是衍生表

        - UNION:若第二个select出现在union之后，则被标记为union；若union包含在from子句的子查询中，外层select将被标记为derived
        - UNION RESULT:告知开发人员，哪些表直接存在union查询
    - type:索引类型，类型
        - system>const>eq_ref>ref>range>index>all,要对type进行优化的前提：有索引
            - 其中system,const只是理性状态；实际能达到ref,range
            - system（忽略）:只有1条数据的系统表；或衍生表只有一条数据的主查询
            - const:仅仅能查到一条数据的SQL，用于PRIMARY KEY 或 UNIQUE索引（const类型与索引类型有关）
            - eq_ref:唯一性索引：低于每个索引键的查询，返回匹配唯一行数据（有且只有1个，不能多，不能为0）常见于唯一索引和主键索引
            - ref: 非唯一性索引，对于每个索引键的查询，返回匹配的所有行（0，多）
            - range:检索指定范围的行，where后面是一个范围查询（between,in,> < >=）

            > in 有时候会失效，从而转换为无索引ALL

            - index:查询全部索引中数据（只需要扫描索引表）
            - all:查询全部表中的数据（需要全表扫描）
    - possible_keys:可能遇到的索引，是一种预测，不准
    - key:实际使用到的索引
    - key_len:索引的长度
        - 作用：用于判断复合索引是否被完全使用
    - ref:注意与type中的ref值区分
        - 作用： 指明当前表所参照的字段
            - 常量 const
    - rows:被索引优化查询的数据个数
    - Extra:
        - using filesort:性能消耗大，需要“额外”的一次排序（查询）

        > 小结：对于单索引，如果排序和查找是同一个字段，则不会出现using filesort；否则则出现。
        > 小结：对于复合索引出现using filesort的避免方法：where和order by 按照复合索引的顺序使用，不要跨列或无序使用。

        - using temporary:性能损耗大，用到了临时表，经常出现在geoup by语句中。(原因：已经有表了，但不使用，必须再来一张表)
            - 解析过程：from...on...join...where...group by...having...select dinstinct...order bu limit...

        > 避免：查询那些列，就根据哪些列group by

        - using index:性能提升，索引覆盖（覆盖索引）。原因：不读取原文件，只从索引文件中获取数据（不需要回表查询）只要使用到的列全部在索引中，就叫做索引覆盖
            - 如果用到了索引覆盖（using index）,会对possible_keys和key造成影响；
                - 如果没有where,则索引只出现在key中
                - 若果有where,则索引出现在key和possible_keys中。
        - using where:需要回表查询
        - impossible_where:where子句永远为false
- MySQL查询优化器会干扰我们的优化

子查询 关联查询 效率问题

- 子查询就是查询中有嵌套的查询，表连接都可以使用子查询，但不是所有子查询都能用表连接替换，子查询比较灵活，方便，形态多样，适合用于作为查询的筛选条件，而表连接更适合于查看多表的数据。
- 子查询不一定需要两个表有关联字段，而连接查询必须有字段关联（所谓的主外键关系）
    - 表关联的效率要高于子查询，因为子查询走的是笛卡尔积。
    - 表关联可能有多条记录，子查询只有一条记录，如果需要唯一的列，最好使用子查询
    - 对于数据多的使用连接查询快一些，原因是：因为子查询会多次遍历所有的数据，而连接查询只会遍历一次。
    - 数据量少的话也就无所谓是连接查询还是子查询，视自己的习惯而定。一般情况下还是用子查询来的好，容易控制
    - 执行子查询时，MYSQL需要创建临时表，查询完毕后再删除这些临时表，所以，子查询的速度会受到一定的影响，这里多了一个创建和销毁临时表的过程。