1.MySQL逻辑分层

MySQL的客户端发送请求,前往MySQL的服务端,服务端分为三层,第一层是**连接层**,第二层是**服务层**,第三层是**引擎层**,最后一程是**存储层**.连接层是提供与客户端连接的服务,服务层首先提供各种用户使用的接口,比如增删改查等操作接口,还提供了一个SQL优化器(MySQL Query Optimizer).引擎层提供了各种存储数据的方式,如InnoDB引擎和MyISAM.存储层用来存储数据.

其中InnoDB和MyISAM的区别:

- InnoDB:事务优先.(适合高并发操作,行锁)
- MyISAM:性能优先.(表锁)

我们可以通过`SHOW ENGINES`来查询MySQL的引擎.

Engine	Support	Comment	Transactions	XA	Savepoints
InnoDB	DEFAULT	Supports transactions, row-level locking, and foreign keys	YES	YES	YES
MRG_MYISAM	YES	Collection of identical MyISAM tables	NO	NO	NO
MEMORY	YES	Hash based, stored in memory, useful for temporary tables	NO	NO	NO
BLACKHOLE	YES	/dev/null storage engine (anything you write to it disappears)	NO	NO	NO
MyISAM	YES	MyISAM storage engine	NO	NO	NO
CSV	YES	CSV storage engine	NO	NO	NO
ARCHIVE	YES	Archive storage engine	NO	NO	NO
PERFORMANCE_SC	YES	Performance Schema	NO	NO	NO
FEDERATED	NO	Federated MySQL storage engine	(Null)	(Null)	(Null)

我们可以通过`SHOW VARIABLES LIKE '%storage_engine%'`来查看MySQL默认使用的引擎.

Variable_name	Value
default_storage_engine	InnoDB
default_tmp_storage_engi	InnoDB
disabled_storage_engines	
internal_tmp_disk_storage	InnoDB

```
CREATE TABLE tb01(
    id int(4) AUTO_INCREMENT,
    `name` VARCHAR(5),
    dept VARCHAR(5),
    PRIMARY KEY(id)
)ENGINE=MyISAM AUTO_INCREMENT=1 DEFAULT CHARSET=UTF8;
```

我们可以在建表的时候规定好引擎为MyISAM

2.SQL优化

2.1.SQL的编写和执行顺序

1.编写过程:

```
SELECT DISTINCT...FROM...JOIN...ON...WHERE...GROUP BY...HAVING...ORDER BY...LIMIT...
```

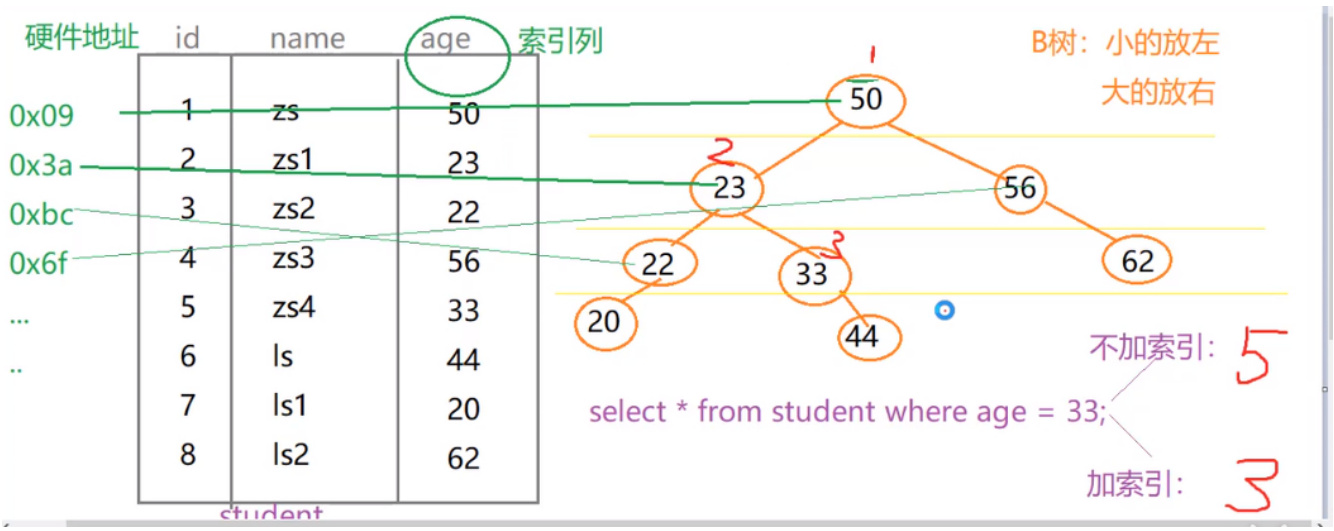
2.执行过程:

```
FROM...ON...JOIN...WHERE...GROUP BY...HAVING...SELECT DISTINCT...ORDER BY...LIMIT...
```

2.2.索引结构

索引相当于书的目录关键字是: `index`.

index是帮助MySQL高效获取数据的数据结构,索引是数据结构(B树).

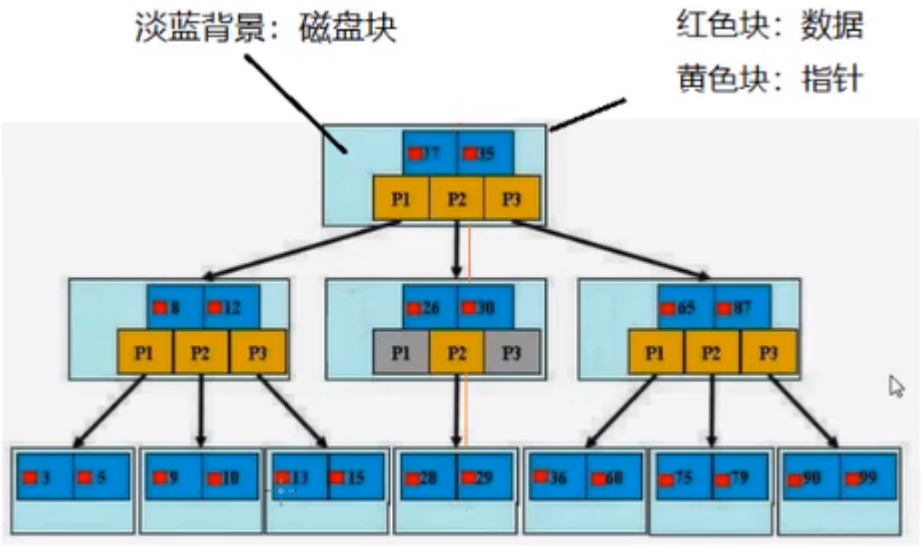


索引的优势:

- 提高查询效率:B树索引,本身就是一个排好序的结构,所以在排序的时候可以直接使用

索引的弊端:

- 索引本身很大,可以存放在硬盘中.
- 索引不是所有情况都适用.
    - 少量数据.
    - 频繁更新的字段
    - 很少适用的字段
- 索引会降低增删改的效率.



2.3.索引分类

单值索引:单列的值,一个表可以有多个单值索引

唯一索引:唯一索引不能重复,唯一索引可以为NULL

复合索引:多个列构成的索引

主键索引:主键索引不能重复,但主键索引不能为NULL

2.4.操作索引

1.创建索引的方式1

```
create 索引类型 索引名 on 表(字段1,字段2);
```

单值索引示例

```
create index dept_index on tb(dept);
```

唯一索引示例

```
create unique index name_index on tb(name);
```

复合索引示例

```
create index dept_name_index on tb(dept,name);
```

2.创建索引的方式2

```
alter table 表名 索引类型 索引名(字段);
```

单值索引示例:

```
alter table tb add index dept_index(dept);
```

唯一索引:

```
alter table tb add unique index name_index(name);
```

复合索引:

```
alter table tb add index dept_name_index(dept,name);
```

注意事项:如果一个字段是primary key,则该字段默认就是主键索引.

3.删除索引

```
drop index 索引名 on 表名;
```

示例:

```
drop index name_index on tb;
```

查询索引

```
show index from 表名;
```

示例

```
show index from tb;
```

2.5.Explain

- 分析SQL的执行计划: `explain`,可以模拟SQL优化器执行SQL语句.`expalin + SQL语句`;
- MySQL查询优化会干扰我们的优化;

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra

- id :编号
  - id 值相同,从上往下顺序执行.这个时候表的执行顺序因表中字段的数量的改变而改变,原因是笛卡尔积.数据小的表优先查询.
  - id值不同,id值越大越优先被查询.体现在嵌套子查询时,先查内层再查外层.
- select_type:查询类型
  - primary:'包含子查询SQL'中的主查询,最外层为primary.
  - subquery:'包含子查询SQL'中的子查询,非最外层为subquery.
  - 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. (示例如下)

```
mysql> explain select  cr.cname from ( select * from course where tid in (1,2) ) cr ;
+----+-------------+------------+------+---------------+------+---------+------+------+-------------+
| id | select_type | table      | type | possible_keys | key  | key_len | ref  | rows | Extra       |
+----+-------------+------------+------+---------------+------+---------+------+------+-------------+
|  1 | PRIMARY     | <derived2> | ALL  | NULL          | NULL | NULL    | NULL |    2 |             |
|  2 | DERIVED     | course     | ALL  | NULL          | NULL | NULL    | NULL |    2 | Using where |
+----+-------------+------------+------+---------------+------+---------+------+------+-------------+
```

        - 在from子查询中,如果有table1 union table2,那么table1就是derived

```
mysql> explain select  cr.cname from ( select * from course where tid = 1  union select * from course where tid = 2 ) c
r ;
+----+--------------+------------+------+---------------+------+---------+------+------+-------------+
| id | select_type  | table      | type | possible_keys | key  | key_len | ref  | rows | Extra       |
+----+--------------+------------+------+---------------+------+---------+------+------+-------------+
|  1 | PRIMARY      | <derived2> | ALL  | NULL          | NULL | NULL    | NULL |    2 |             |
|  2 | DERIVED      | course     | ALL  | NULL          | NULL | NULL    | NULL |    2 | Using where |
|  3 | UNION        | course     | ALL  | NULL          | NULL | NULL    | NULL |    2 | Using where |
| NULL | UNION RESULT | <union2,3> | ALL  | NULL          | NULL | NULL    | NULL | NULL |             |
+----+--------------+------------+------+---------------+------+---------+------+------+-------------+
```

    - union:上例
    - unionresult:上例,告知开发开发人员, 哪些表之间存在union行为.
- table:表
- type:索引类型:system>const>eq_ref>ref>range>index>all
    - 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
    - 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,因为只匹配一行数据,所以很快,如将主键置于where列表中,MySQL就能将该查询转换为一个常量.
    - eq_ref:唯一性索引扫描,对于每个索引键的查询,返回匹配唯一数据,常见于唯一索引和主键索引.
    - ref:非唯一性索引扫描,对于每个索引键的查询,返回匹配的所有行.
    - range:检索指定范围的行,where后面是一个范围查询(between,<,>,in),注意in和数据量有关,大于数据量一半就会全表扫描.
    - index:查询全部索引数据.
    - all:查询全部表的数据.

    注意:const与eq_ref的区别主要是一个是单表一个是多表.
- possible_keys:预测用到的索引
- key:实际使用的索引
- key_len:实际使用的索引的长度
    - 作用:用于判断复合索引是否被完全使用.
    - 注意:UTF-8是一个字符占3个字节,非空判断占1个字节,可变长度标识占2个字节.
- ref:表之间的引用,指明当前表所参照的字段
    -

```
mysql> explain select * from course c,teacher t where c.tid = t.tid  and t.tname ='tw' ;
+----+-------------+-------+------+---------------------+------------+---------+------------+------+-------------+
| id | select_type | table | type | possible_keys       | key        | key_len | ref        | rows | Extra       |
+----+-------------+-------+------+---------------------+------------+---------+------------+------+-------------+
|  1 | SIMPLE      | t     | ref  | index_name,tid_index | index_name | 63      | const      |    1 | Using where |
|  1 | SIMPLE      | c     | ref  | tid_index           | tid_index  | 5       | myDB.t.tid |    1 | Using where |
+----+-------------+-------+------+---------------------+------------+---------+------------+------+-------------+
2 rows in set (0.00 sec)
```

    - 可以是某表的某个字段,也可以是const常量
- rows:通过索引查询的数据量
    - 被索引优化查询的数据个数.
- extra:额外的信息
    - using filesort:性能消耗大.需要额外的一次排序(查询).常见于order by语句中.
        - 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,如果排序和查找的不是同一个字段,则会出现using filesort.
        - 对于复合索引,排序和查找的索引是不能够跨列的(最佳左前缀),否则也会出现using filesort.注意,对于复合索引(id1,id2,id3),如果排序和查找的是id2和id3,也是属于跨列,跨列了一个id1.
    - using temporary:性能消耗大,用到了临时表.常见于group by语句中.

- using index:性能提升,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都在索引中,就是索引覆盖.
- using where:需要回表查询.
- impossible where:where子句永远为false.

2.6.优化示例

1:推荐写法,索引的使用顺序和复合索引的顺序一致.

```
mysql> explain select a1,a2,a3,a4 from test03 where a1=1 and a2=2 and a3=3 and a4 =4 ;
+----+-------------+--------+------+---------------+---------------+---------+-------------------------+------+-------
-------+
| id | select_type | table  | type | possible_keys | key           | key_len | ref                     | rows | Extra
       |
+----+-------------+--------+------+---------------+---------------+---------+-------------------------+------+-------
-------+
|  1 | SIMPLE      | test03 | ref  | idx_a1_a2_a3_4 | idx_a1_a2_a3_4 | 16      | const,const,const,const |    1 | Using
 index |
+----+-------------+--------+------+---------------+---------------+---------+-------------------------+------+-------
```

2.虽然写的SQL顺序不满足于复合索引的顺序,但是MySQL在服务层提供了一个SQL优化器,对SQL进行了优化.出现这种情况是可遇不可求的,不推荐.

```
mysql> explain select a1,a2,a3,a4 from test03 where a4=1 and a3=2 and a2=3 and a1 =4 ;
+----+-------------+--------+------+---------------+---------------+---------+-------------------------+------+-------
-------+
| id | select_type | table  | type | possible_keys | key           | key_len | ref                     | rows | Extra
       |
+----+-------------+--------+------+---------------+---------------+---------+-------------------------+------+-------
-------+
|  1 | SIMPLE      | test03 | ref  | idx_a1_a2_a3_4 | idx_a1_a2_a3_4 | 16      | const,const,const,const |    1 | Using
 index |
+----+-------------+--------+------+---------------+---------------+---------+-------------------------+------+-------
```

3.以下SQL用到了a1,a2两个索引,该两个字段,不需要回表查询using index,而a4因为跨列使用,造成了该索引失效,需要回表查询,可以通过key_len进行校验.

```
mysql> explain select a1,a2,a3,a4 from test03 where a1=1 and a2=2 and a4=3 order by a3;
+----+-------------+--------+------+---------------+---------------+---------+-------------+------+----------------
--------+
| id | select_type | table  | type | possible_keys | key           | key_len | ref         | rows | Extra
       |
+----+-------------+--------+------+---------------+---------------+---------+-------------+------+----------------
--------+
|  1 | SIMPLE      | test03 | ref  | idx_a1_a2_a3_4 | idx_a1_a2_a3_4 | 8       | const,const |    1 | Using where; Usin
g index |
+----+-------------+--------+------+---------------+---------------+---------+-------------+------+----------------
--------+
```

4.这主要看where 和 order拼接起来后能不能满足索引顺序. 上述列是a1,a2,a4(索引失效不管),a3,满足了索引顺序.而下例是a1,a4(索引失效不管),a3.不满足索引的顺序,属于跨列,那么就会出现额外的一次排序.

```
mysql> explain select a1,a2,a3,a4 from test03 where a1=1 and a4=4 order by a3;
+----+-------------+--------+------+---------------+---------------+---------+-------+------+------------------------
------------------+
| id | select_type | table  | type | possible_keys | key           | key_len | ref   | rows | Extra
                  |
+----+-------------+--------+------+---------------+---------------+---------+-------+------+------------------------
------------------+
|  1 | SIMPLE      | test03 | ref  | idx_a1_a2_a3_4 | idx_a1_a2_a3_4 | 4       | const |    1 | Using where; Using inde
x; Using filesort |
+----+-------------+--------+------+---------------+---------------+---------+-------+------+------------------------
------------------+
1 row in set (0.00 sec)
```

总结:复合索引和使用顺序全部一致(且不跨列使用),则复合索引全部使用,如果部分一致,则使用部分索引.

2.7.单表优化

```
mysql> explain select bid from book where typeid in(2,3) and authorid=1  order by typeid desc ;
+----+-------------+-------+------+---------------+------+---------+------+------+----------------------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra                      |
+----+-------------+-------+------+---------------+------+---------+------+------+----------------------------+
|  1 | SIMPLE      | book  | ALL  | NULL          | NULL | NULL    | NULL |    4 | Using where; Using filesort |
+----+-------------+-------+------+---------------+------+---------+------+------+----------------------------+
```

是单表中不加任何索引的查询结果.

优化1:加索引:`alter table book add index idx_bta(bid,typeid,authorid);`

```
mysql> explain select bid from book where typeid in(2,3) and authorid=1  order by typeid desc ;
+----+-------------+-------+-------+---------------+---------+---------+------+------+----------------------------------
---------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref  | rows | Extra
         |
+----+-------------+-------+-------+---------------+---------+---------+------+------+----------------------------------
---------+
|  1 | SIMPLE      | book  | index | NULL          | idx_bta | 12      | NULL |    4 | Using where; Using index; Using
filesort |
+----+-------------+-------+-------+---------------+---------+---------+------+------+----------------------------------
---------+
```

优化2:上述加的索引的顺序问题.按照SQL的执行顺序,我们要对索引进行改变,索引一旦进行升级优化,需要将之前废弃的索引删掉,防止干扰.

`drop index idx_bta on book;`

`alter table book add index idx_tab(typeid,authorid,bid);`

```
mysql> explain select bid from book where typeid in(2,3) and authorid=1  order by typeid desc ;
+----+-------------+-------+-------+---------------+---------+---------+------+------+--------------------------+
| id | select_type | table | type  | possible_keys | key     | key_len | ref  | rows | Extra                    |
+----+-------------+-------+-------+---------------+---------+---------+------+------+--------------------------+
|  1 | SIMPLE      | book  | index | idx_tab       | idx_tab | 12      | NULL |    4 | Using where; Using index |
+----+-------------+-------+-------+---------------+---------+---------+------+------+--------------------------+
1 row in set (0.00 sec)
```

优化3:注意如果上述查询中typeid in(2,3)失效了,那么authorid会跟着失效,所以改变顺序.

`drop index idx_tab on book;`

`alter table book add index idx_atb(authorid,typeid,bid);`

```
mysql> explain select bid from book where  authorid=1 and  typeid in(2,3) order by typeid desc ;
+----+-------------+-------+------+---------------+---------+---------+-------+------+--------------------------+
| id | select_type | table | type | possible_keys | key     | key_len | ref   | rows | Extra                    |
+----+-------------+-------+------+---------------+---------+---------+-------+------+--------------------------+
|  1 | SIMPLE      | book  | ref  | idx_atb       | idx_atb | 4       | const |    2 | Using where; Using index |
+----+-------------+-------+------+---------------+---------+---------+-------+------+--------------------------+
1 row in set (0.00 sec)
```

2.8.多表优化

`select* from teacher2 t left outer join course2 c on t.cid=c.cid where c.cname='java';`

```
mysql> explain select *from teacher2 t left outer join course2 c
    -> on t.cid=c.cid where c.cname='java';
+----+-------------+-------+------+---------------+------+---------+------+------+------------------------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra                        |
+----+-------------+-------+------+---------------+------+---------+------+------+------------------------------+
|  1 | SIMPLE      | t     | ALL  | NULL          | NULL | NULL    | NULL |    3 |                              |
|  1 | SIMPLE      | c     | ALL  | NULL          | NULL | NULL    | NULL |    3 | +Using where; Using join buffer |
+----+-------------+-------+------+---------------+------+---------+------+------+------------------------------+
```

分析:小表驱动大表.如`...where 小表.x = 大表.y`;索引需要建立在经常使用的字段上,所以在上述的SQL中,`t.cid`会经常性的使用,我们需要对`t.cid`添加索引.一般情况下对于左外连接,给左表加索引,右外连接,给右表加索引.

`alter table teacher2 add index index_teacher2_cid(cid);`

```
mysql> explain select *from teacher2 t left outer join course2 c on t.cid=c.cid where c.cname='java';
+----+-------------+-------+-------+-------------------+-------------------+---------+------+------+----------------
---------------+
| id | select_type | table | type  | possible_keys     | key               | key_len | ref  | rows | Extra
               |
+----+-------------+-------+-------+-------------------+-------------------+---------+------+------+----------------
---------------+
|  1 | SIMPLE      | t     | index | index_teacher2_cid | index_teacher2_cid | 4       | NULL |    3 | Using index
               |
|  1 | SIMPLE      | c     | ALL   | NULL              | NULL              | NULL    | NULL |    3 | Using where; Usi
ng join buffer |
+----+-------------+-------+-------+-------------------+-------------------+---------+------+------+----------------
---------------+
```

我们还需要给c.name添加索引,因为是根据c.name进行查询.

```
alter table course2 add index index_course2_cname(cname);
```

```
mysql> explain select *from teacher2 t left outer join course2 c on t.cid=c.cid where c.cname='java';
+----+-------------+-------+------+--------------------+--------------------+---------+------------+------+----------
---+
| id | select_type | table | type | possible_keys      | key                | key_len | ref        | rows | Extra
   |
+----+-------------+-------+------+--------------------+--------------------+---------+------------+------+----------
---+
|  1 | SIMPLE      | c     | ref  | index_course2_cname | index_course2_cname | 63      | const      |    1 | Using whe
re |
|  1 | SIMPLE      | t     | ref  | index_teacher2_cid  | index_teacher2_cid  | 4       | myDB.c.cid |    1 | Using ind
ex |
+----+-------------+-------+------+--------------------+--------------------+---------+------------+------+----------
---+
```

2.9.避免索引失效的原则

SQL优化只是一个概率层面的优化,至于是否实际使用了我们的优化,需要通过explain进行推测.这是由于SQL优化器的原因.

1.复合索引,不要跨列或无序使用(最佳左前缀),尽量使用全索引匹配.如果左侧索引失效,右侧全部失效.

2.复合索引不能使用不等于(!= <>或is null或is not null),否则自身以及右侧全部失效.

3.不要在索引上进行任何操作(计算,函数,类型转换),否则索引都会失效.

4.那么如何才能保证索引绝对不失效呢?尽量去使用索引覆盖(using index);

5.like尽量以'常量'开头,不要以'%'开头,否则索引失效.如果必须要使用'%'开头,可以使用覆盖索引挽回一部分.

6.尽量不要使用类型转换(显示,隐式),如`xplain select * from teacher where tame = 123`,程序底层将123转换成了'123',进行了类型转换,索引失效

7.尽量不要使用or,否则索引失效.

2.10.其他的优化方法

1.exist和in

select ... from table where exist (子查询);

select ... from table where 字段 in (子查询);

如果主查询的数据集大,则使用in,如果子查询的数据集大,则使用exist.

2.oder by

using filesort 有两种算法,单路排序和双路排序(根据IO的次数)

1.MySQL4.1之前,默认使用双路排序;双路:扫描2次磁盘(1.从磁盘读取排序字段,在缓冲区进行排序;2.读取其他字段);

2.MySQL4.1之后,默认使用单路排序;单路:扫描1次磁盘(全部字段),在buffer缓冲区中进行排序.不一定是单次IO,有可能多次IO.如果数据量过大,则无法一次性读取完毕,会进行分片.如果数据量过大,可以考虑调大buffer的容量的大小,`set max_length_for_sort_data = 1024`,单位是字节.

3.如果max_length_for_sort_data值太低,则mysql会自动从单路切换成双路.

提高oder by查询的策略:

1. 选择使用单路,双路,调整buffer的容量大小.
2. 避免select * ...

3. 保证全部的排序字段 排序的一致性(都是升序或降序)

2.11.慢查询日志

MySQL提供的一种日志记录,用于记录MySQL中响应时间超过阈值的SQL语句(long_query_time,默认10s)

慢查询日志默认是关闭的,建议:开发调优是打开,而最终部署时关闭.

1.开启慢查询日志

检查是否开启了慢查询日志: `show variables like '%slow_query_log%'`;

信息 | Result 1 | 概况 | 状态

Variable_name	Value
slow_query_log	ON
slow_query_log_file	FECHIN-THINKPAD-slow.l

临时开启: `set global slow_query_log = 1;`

永久开启:/etc/my.cnf中追加配置.

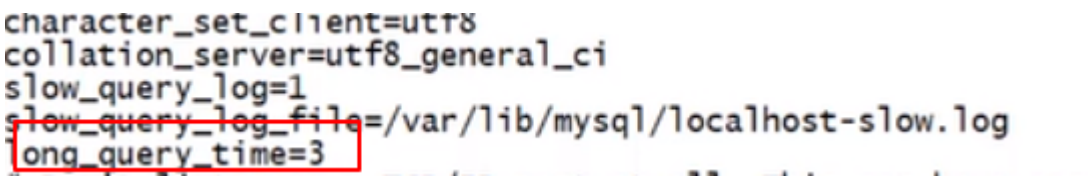
```
# The MySQL server
[mysqld]
port            = 3306
socket          = /var/lib/mysql/mysql.sock
skip-external-locking
key_buffer_size = 384M
max_allowed_packet = 1M
table_open_cache = 512
sort_buffer_size = 2M
read_buffer_size = 2M
read_rnd_buffer_size = 8M
myisam_sort_buffer_size = 64M
thread_cache_size = 8
query_cache_size = 32M
# Try number of CPU's*2 for thread_concurrency
thread_concurrency = 8
character_set_server=utf8
character_set_client=utf8
collation_server=utf8_general_ci
slow_query_log=1
slow_query_log_file=/var/lib/mysql/localhost-slow.log
```

检查慢查询阀值: `show variables like '%long_query_time%'`;

2.设置慢查询的阀值

临时设置阀值: `set global long_query_time = 5` ;设置完毕后,重新登录后起效.

永久设置阀值: 继续在/etc/my.cnf中追加配置.



3.查看慢查询的SQL

查询超过阀值的SQL的个数: show global status like '%slow_queries%';

1.慢查询的sql被记录在了日志中,因此可以通过日志来查看具体的慢SQL.

2.通过mysqldumpslow工具查看慢SQL.

我们可以通过mysqldump --help来获取帮助

列如: `mysqldumpslow -s t -t 10 -g 'left join' /var/lib/mysql/localhost-slow.log`

2.12.分析海量数据

3.锁机制

3.1.锁分类

锁机制:解决因资源共享,而造成的并发问题.

分类:根据操作类型分:**读锁**和**写锁**,根据操作范围分:**表锁**,**行锁**,**页锁**.

- 读锁(共享锁):对同一个数据,多个读操作可以同时进行,互不干扰
- 写锁(互斥锁):如果但是当前写操作没有完毕,则无法进行其他的读操作,写操作.
- 表锁:对一张表整体加锁,如MySIAM存储引擎使用表锁,开销小,加锁快.无死锁,容易发生锁冲突.并发能力差.
- 行锁:对一条数据枷锁.如InnoDB存储引擎使用行锁,开销大,加锁慢,容易出现死锁.不易发生锁冲突,并发度高(很小概率发生高并发问题:脏读,幻读,不可重复读,丢失更新)

3.2.表锁

1.表锁的基本指令

增加锁: `lock table 表1 read/write,表2 read/write ...;`

查看加锁的表: `show open tables;`

释放锁: `unlock tables;` 也可以通过事务释放锁.

2.读锁和写锁

读锁:

- 如果某个会话1,对A表加了read锁,则该会话可以对A表进行读操作,不能进行写操作.对其它表既不能进行读操作也不能进行写操作.
- 会话2,可以对A表进行读操作,进行写操作时会等待会话1将锁释放.对于其它表可以进行读操作,也可以进行写操作.

写锁:

- 会话1对A表加了write锁,可以对A进行任何的增删改查,但是不能操作其他表.
- 会话2对A表可以进行增删改查的前提是等待会话1释放写锁.

3.MySQL表级锁的锁模式

MyISAM在执行查询语句(DQL)前,会自动给涉及的所有表加读锁,在执行增删改操作(DML)前,会自动给涉及的表加写锁.所以对MyISAM表进行操作,会有以下情况:

- 对MyISAM表的读操作(加读锁),不会阻塞其他进程(会话)对同一表的读请求会阻塞对同一表的写请求,只有当读锁释放后,才会执行其它进程的写操作.
- 对MyISAM表进行的写操作(加写锁),会阻塞其他进程(会话)对同一表的读和写操作,只有当写锁释放后,才会执行其它进程的读写操作.

4.分析表锁定

分析表的严重程度:`show status like 'table%'`;操作结果如下:

信息 | Result 1 | 概况 | 状态

Variable_name	Value
Table_locks_immediate	215
Table_locks_waited	0
Table_open_cache_hits	39
Table_open_cache_misses	6
Table_open_cache_overflc	0

Table_locks_immediate:表示立即获取锁的查询次数.

Table_locks_waited:表示需要等待的表锁数.

3.3.行锁

为了研究行锁,暂时将自动commit关闭.

```
set autocommit=0;
```

```
start transaction;
```

```
begin;
```

- 如果会话a对某条数据进行DML操作,则其他会话必须等待会话a结束事务(commit/rollback)后,才能对该条数据进行操作.如果操作不同的数据,那么互不影响,互不干扰.
- 我们也可以对查询数据进行加锁,使用for update.`select * from linelock where id = 2 for update`;

行锁的注意事项:

- 如果没有索引,行锁会转为表锁.
  - 会话1.写操作:update linelock set name = 'a3' where name = '3';
  - 会话2.写操作:update linelock set name = 'a4' where name = '4';
  - 上述操作没有任何问题,因为操作的是不同的记录,互不干扰.

    - 会话1.写操作:update linelock set name = 'a3' where name = 3;
    - 会话2.写操作:updare linelock set name = 'a4' where name = 4;
    - 上述操作,数据被阻塞了(加锁).

  原因:如果索引发生了类型转换,那么索引就会失效,索引失效就会导致行锁转换为表锁.

- 行锁的一种特殊情况:间隙锁:值在范围内,但却不存在.
    - 如`update linelock set name = 'x' where id>1 and id <9;`,即在此where范围中,没有id = 7的数据,那么MySQL会自动给间隙加上锁(间隙锁).

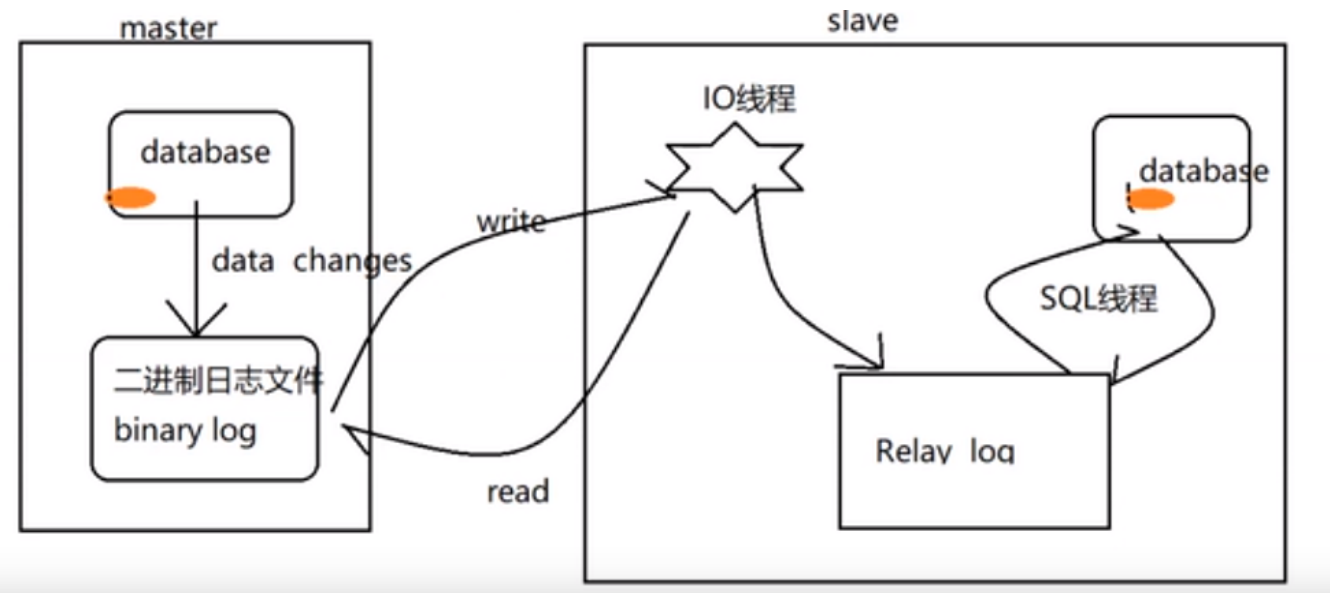
行锁的分析:`show status like '%innodb_row_lock%';`

Variable_name	Value
Innodb_row_lock_current_waits	0
Innodb_row_lock_time	0
Innodb_row_lock_time_avg	0
Innodb_row_lock_time_max	0
Innodb_row_lock_waits	0

1. innodb_row_lock_current_waits:当前正在等待锁的数量
2. innodb_row_lock_time:从系统启动到现在,一共等待的时间
3. innodb_row_lock_time_avg:从系统启动到现在,平均等待的时长
4. innodb_row_lock_time_max:从系统启动到现在,最大等待的时长
5. innodb_row_lock_waits:从系统启动到现在,等待的次数.

4.主从复制

主从复制的原理图:



1. master将改变的数据记录在本地的二进制日志中(binary log),该过程称之为:二进制日志时间.
2. slave将master的binary log拷贝到自己的relay log(中继日志文件)中.
3. 中继日志事件将数据读取到自己的数据库中.

MySQL主从复制是异步的,串行化的,有延迟的.