InnoDB、MyISAM差异

- 事务支持：InnoDB支持事务处理，这意味着它支持ACID事务特性（原子性、一致性、隔离性和持久性）。而MyISAM不支持事务。
- 行级锁定与表级锁定：InnoDB支持行级锁定，这意味着在多用户并发访问时，它只锁定被访问的行，而不是整个表，这可以提高并发性能。而MyISAM使用表级锁定，当一个线程访问表时，其他线程必须等待。
- 外键支持：InnoDB支持外键约束，这是保证数据完整性的重要手段。MyISAM则不支持外键。
- 存储结构：MyISAM总共有三个文件***.frm（表结构）、***.myd（表数据）***、.myi（表索引），而InnoDB只要两个文件***.frm（表结构）、***.ibd（表索引加数据）。此外，MyISAM的索引是基于B+树的，但叶子节点存储的是数据地址，而InnoDB的叶子节点直接存储数据。

索引的分类

- 主键索引：针对表中主键所建索引，只能有一个。
- 唯一索引
- 普通索引
- 全文索引

慢查询

- 查看慢查询日志是否开启：show variables like 'slow_query_log';
- MySQL的慢查询日志默认是没有开启的，需要在MySQL的配置文件（etc/my.cnf）中配置如下信息：
  - slow_query_log=1 开启慢查询日志开关
  - long_query_time=2 设置慢查询日志的时长，SQL语句执行时间超过2秒则会被视为慢查询，从而被记录。
  - 慢查询日志地址在/var/lib/mysql/localhost-slow.log。

profile

- 查看MySQL是否支持profile操作 select @@have_profiling;
- 默认profile是关闭的，select @@profiling;（查看是否开启）set (global/session) profiling=1;（设置全局/会话开启）。
- show profiles; 查看开启后每条SQL语句的执行耗时。

最左前缀法则

- 使用联合索引时，最左边的列必须存在。比如联合索引（A、B、C），使用（A、B、C）、（A、B）、(A)会使用联合索引，使用（A、C）也会使用联合索引但其索引长度跟（A）一样，也就是说字段C的索引并没有用，也就是部分失效。
- order by使用联合索引时，必须完全按照联合索引的顺序。
- explain实例如下所示：

- ```
mysql> show index from test_1;
+--------+------------+-----------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment | Visible | Expression |
+--------+------------+-----------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+
test_1	0	PRIMARY	1	ID	A	0	NULL	NULL		BTREE			YES	NULL
test_1	1	idx_age_score	1	Age	A	0	NULL	NULL	YES	BTREE			YES	NULL
test_1	1	idx_age_score	2	Score	A	0	NULL	NULL	YES	BTREE			YES	NULL
+--------+------------+-----------------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+---------+------------+
3 rows in set (0.00 sec)

mysql> explain select Age,Score,ID from test_1 where age=1 and score=1;
+----+-------------+--------+------------+------+---------------+---------------+---------+-------------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-------------+--------+------------+------+---------------+---------------+---------+-------------+------+----------+-------------+
| 1 | SIMPLE | test_1 | NULL | ref | idx_age_score | idx_age_score | 10 | const,const | 1 | 100.00 | Using index |
+----+-------------+--------+------------+------+---------------+---------------+---------+-------------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)

mysql> explain select Age,Score,ID from test_1 where score=1 and age=1;
+----+-------------+--------+------------+------+---------------+---------------+---------+-------------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-------------+--------+------------+------+---------------+---------------+---------+-------------+------+----------+-------------+
| 1 | SIMPLE | test_1 | NULL | ref | idx_age_score | idx_age_score | 10 | const,const | 1 | 100.00 | Using index |
+----+-------------+--------+------------+------+---------------+---------------+---------+-------------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)

mysql> explain select Age,Score,ID from test_1 order by age,score;
+----+-------------+--------+------------+-------+---------------+---------------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-------------+--------+------------+-------+---------------+---------------+---------+------+------+----------+-------------+
| 1 | SIMPLE | test_1 | NULL | index | NULL | idx_age_score | 10 | NULL | 1 | 100.00 | Using index |
+----+-------------+--------+------------+-------+---------------+---------------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)

mysql> explain select Age,Score,ID from test_1 order by score,age;
+----+-------------+--------+------------+-------+---------------+---------------+---------+------+------+----------+--+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-------------+--------+------------+-------+---------------+---------------+---------+------+------+----------+--+
| 1 | SIMPLE | test_1 | NULL | index | NULL | idx_age_score | 10 | NULL | 1 | 100.00 | Using index; Using filesort |
+----+-------------+--------+------------+-------+---------------+---------------+---------+------+------+----------+--+
1 row in set, 1 warning (0.00 sec)
```

# 索引失效

- 对索引进行运算。
- 使用字符串索引时没加单引号。
- 模糊查询前面加百分号。
- 使用or对条件进行组合时，只要有一个条件没有索引，那另外一个就算有索引也不会使用。

# 前缀索引

- 当字段类型为字符串时，有时候需要索引很长的字符串，这会让索引变得很大，查询时，浪费大量的磁盘IO，影响查询性能，此时只对字符串的一部分前缀，建立索引，这样可以大大节约索引空间，从而提升性能。
- create index idx_xxx on table_name(column(n)); n表示几个字符。

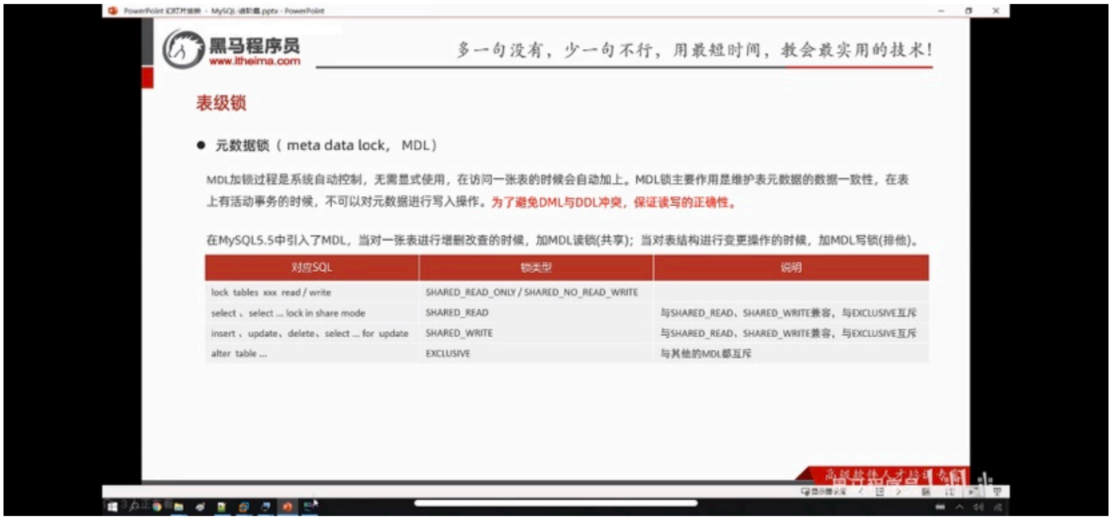
# 全局锁

- 应用场景：数据备份（锁住后写入操作将会阻塞）。
- 全局锁数据备份示例如下：
  - flush tables with read lock;
  - mysqldump -hxxx -uxxx -pxxx databasename > databasename.sql;（非SQL语句）
  - unlock tables;

# 表锁

- lock tables xxx read/write;
- unlock tables;
```

元数据锁（表级锁）

- 

- 系统自动控制，无需主动显示调用，当一张表涉及到未提交的事务时，元数据锁会阻止对该表的结构进行修改。如下图所示，此时如果修改表结构则会阻塞。
-
```
mysql> update test_1 set age = 10 where id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>
mysql> select * from metadata_locks;
| OBJECT_TYPE | OBJECT_SCHEMA      | OBJECT_NAME    | COLUMN_NAME | OBJECT_INSTANCE_BEGIN | LOCK_TYPE    | LOCK_DURATION | LOCK_STATUS | SOURCE            | OWNER_THREAD_ID | OWNER_EVENT_ID |
| TABLE       | test               | test_1         | NULL        |         2173716551504 | SHARED_WRITE | TRANSACTION   | GRANTED     | sql_parse.cc:6142 |              55 |             44 |
| TABLE       | performance_schema | metadata_locks | NULL        |         2173716553136 | SHARED_READ  | TRANSACTION   | GRANTED     | sql_parse.cc:6142 |              56 |             23 |
```

行锁、间隙锁（行级锁）、临键锁（行级锁）

- 可以通过”$performance_schema$”数据库的”$data_lock$”查看当前所持锁的信息。
- 临键锁本质上其实就是行锁加间隙锁。只有在可重复读隔离级别下，如下所示情况下才会加间隙锁：
- 1、字段是唯一索引时，只有更新某个范围记录或者一条不存在的记录的时候，才会加间隙锁，指定某条存在的记录时，只会加行锁，不会加间隙锁。
- 2、字段是普通索引时，不管是单条还是多条都会加间隙锁。
- 间隙锁的目的是为了防止其他事务在间隙中塞入数据，间隙锁之间是不互斥的，不同事务可以对相同的间隙加锁。

```
mysql> select * from test_1;
+----+------+-----+-------+
| ID | Name | Age | Score |
+----+------+-----+-------+
|  1 |    1 |  10 |     1 |
|  2 |    2 |   3 |     2 |
|  3 |    3 |   3 |     3 |
|  5 |    5 |   5 |   100 |
|  7 |    7 |   7 |   100 |
|  8 |    8 |   8 |   100 |
|  9 |    9 |   9 |     9 |
| 10 |   10 |  10 |   100 |
| 20 |   20 |  20 |    20 |
+----+------+-----+-------+
```

- 示例1：字段是唯一索引，更新不存在的记录。(update test_1 set score=100 where id=15;)

```
| ENGINE | ENGINE_LOCK_ID                      | ENGINE_TRANSACTION_ID | THREAD_ID | EVENT_ID | OBJECT_SCHEMA | OBJECT_NAME | PARTITION_NAME | SUBPARTITION_NAME | INDEX_NAME | OBJECT_INSTANCE_BEGIN | LOCK_TYPE | LOCK_MODE | LOCK_STATUS |
LOCK_DATA |
| INNODB | 3085157338688:1064:3085162974232    |                  2882 |        50 |       72 | test          | test_1      | NULL           | NULL              | NULL       |         3085162974232 | TABLE     | IX        | GRANTED     |
NULL      |
| INNODB | 3085157338688:2:4:10:3085146849304  |                  2882 |        50 |       72 | test          | test_1      | NULL           | NULL              | PRIMARY    |         3085146849304 | RECORD    | X,GAP     | GRANTED     |
20        |
2 rows in set (0.00 sec)
```

- 示例2：字段是唯一索引，更新范围。(update test_1 set score=100 where id>=9 and id<15;)

```
| ENGINE | ENGINE_LOCK_ID                      | ENGINE_TRANSACTION_ID | THREAD_ID | EVENT_ID | OBJECT_SCHEMA | OBJECT_NAME | PARTITION_NAME | SUBPARTITION_NAME | INDEX_NAME | OBJECT_INSTANCE_BEGIN | LOCK_TYPE | LOCK_MODE     | LOCK_STATU
S | LOCK_DATA |
| INNODB | 3085157338688:1064:3085162974232    |                  2883 |        50 |       76 | test          | test_1      | NULL           | NULL              | NULL       |         3085162974232 | TABLE     | IX            | GRANTED
  | NULL      |
| INNODB | 3085157338688:2:4:9:3085146849304   |                  2883 |        50 |       76 | test          | test_1      | NULL           | NULL              | PRIMARY    |         3085146849304 | RECORD    | X,REC_NOT_GAP | GRANTED
  | 9         |
| INNODB | 3085157338688:2:4:6:3085146849648   |                  2883 |        50 |       76 | test          | test_1      | NULL           | NULL              | PRIMARY    |         3085146849648 | RECORD    | X             | GRANTED
  | 10        |
| INNODB | 3085157338688:2:4:10:3085146849992  |                  2883 |        50 |       76 | test          | test_1      | NULL           | NULL              | PRIMARY    |         3085146849992 | RECORD    | X,GAP         | GRANTED
  | 20        |
4 rows in set (0.00 sec)
```

- 示例3：字段是普通索引，更新单行。 （update test_1 set score=100 where name=10;）

```
+--------+--------------------------------+--------------------+-----------+----------+---------------+-------------+----------------+------------+-----------------------+-----------+---------------+-------------+
| ENGINE | ENGINE_LOCK_ID                 | ENGINE_TRANSACTION_ID | THREAD_ID | EVENT_ID | OBJECT_SCHEMA | OBJECT_NAME | PARTITION_NAME | SUBPARTITION_NAME | INDEX_NAME | OBJECT_INSTANCE_BEGIN | LOCK_TYPE | LOCK_MODE     | LOCK_STATUS | LOCK_DATA |
+--------+--------------------------------+--------------------+-----------+----------+---------------+-------------+----------------+------------+-----------------------+-----------+---------------+-------------+
| INNODB | 3085157338688:1064:3085162974232 |               2897 |        50 |       85 | test          | test_1      | NULL           | NULL       | NULL       |          3085162974232 | TABLE     | IX            | GRANTED     | NULL      |
| INNODB | 3085157338688:2:6:9:3085146849304 |               2897 |        50 |       85 | test          | test_1      | NULL           | NULL       | idx_name   |          3085146849304 | RECORD    | X             | GRANTED     | 10, 10    |
| INNODB | 3085157338688:2:4:6:3085146849648 |               2897 |        50 |       85 | test          | test_1      | NULL           | NULL       | PRIMARY    |          3085146849648 | RECORD    | X,REC_NOT_GAP | GRANTED     | 10        |
| INNODB | 3085157338688:2:6:10:3085146849992 |               2897 |        50 |       85 | test          | test_1      | NULL           | NULL       | idx_name   |          3085146849992 | RECORD    | X,GAP         | GRANTED     | 20, 20    |
+--------+--------------------------------+--------------------+-----------+----------+---------------+-------------+----------------+------------+-----------------------+-----------+---------------+-------------+
4 rows in set (0.00 sec)
```

每个表的数据量

- 这并没有一个绝对值，网上不少说法都是说不能超过2000W条数据，这个数据的来源是：根节点的页可以存1200多个索引，那么第二层的索引个数是1200*1200=144W，如果每条数据的大小为1K，那么第三层每一页（16K）可以存16条数据，那么层高为3的B+树，可以存144W*16≈2300W条数据。其实每个表的最优数据条数得从多个方面来考量，比如索引效率，如果表的索引较多较复杂的话，数据条数过多的话，那么维护索引的性能就会越来越低。还有就是设置的buff poll大小，如下面介绍所示，数据库专属服务器buff poll建议设置为物理内存的百分之八十，其实简单的考量的话，buff poll的大小应该就等于单表数据量的大小，因为单表的数据可以都加载到内存中，那么性能也不会差。2000W条数据的来源可能就是基于这个，2000W条数据内存占用大概就是20G。

主键

buff pool

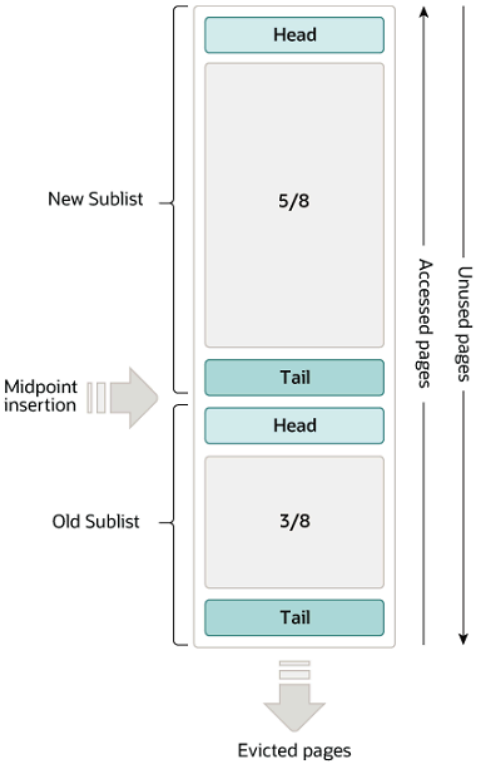
- 官网介绍地址：https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool.html
- On dedicated servers, up to 80% of physical memory is often assigned to the buffer pool.数据库专属服务器，建议分配百分之八十的物理内存作为buff pool。

Buffer Pool LRU Algorithm

The buffer pool is managed as a list using a variation of the LRU algorithm. When room is needed to add a new page to the buffer pool, the least recently used page is evicted and a new page is added to the middle of the list. This midpoint insertion strategy treats the list as two sublists:

- At the head, a sublist of new ("young") pages that were accessed recently
- At the tail, a sublist of old pages that were accessed less recently

Figure 17.2 Buffer Pool List



- When InnoDB reads a page into the buffer pool, it initially inserts it at the midpoint (the head of the old sublist). A page can be read because it is required for a user-initiated operation such as an SQL query, or as part of a read-ahead operation performed automatically by InnoDB.
  （当innodb读取数据到buff pool,先将其放到old sublist的头部，数据页被加载到缓存，那必然是玩家执行操作或者被innodb的预读机制加载。）
- You can control the insertion point in the LRU list and choose whether InnoDB applies the same optimization to blocks brought into the buffer pool by table or index scans. The configuration parameter innodb_old_blocks_pct controls the percentage of “old” blocks in the LRU list. The default value of innodb_old_blocks_pct is 37, corresponding to the original fixed ratio of 3/8. The value range is 5 (new pages in the buffer pool age out very quickly) to 95 (only 5% of the buffer pool is reserved for hot pages, making the algorithm close to the familiar LRU strategy).
  （主要是介绍innodb_old_blocks_pct参数，new sublist和old sublist的比例，默认是37。）
- A data page is typically accessed a few times in quick succession and is never touched again. The configuration parameter innodb_old_blocks_time specifies the time window (in milliseconds) after the first access to a page during which it can be accessed without being moved to the front (most-recently used end) of the LRU list. The default value of innodb_old_blocks_time is 1000. Increasing this value makes more and more blocks likely to age out faster from the buffer pool.
  （上面的意思是大部分时候数据被加载后可能只是短时间内访问，如果访问了就从old sublist挪到new sublist是不合理的，此时可以设置innodb_old_blocks_time，默认为1000毫秒，这个参数的意思是超过这个时间old sublist里的页还被访问就把它挪到new sublist。）

undolog、redolog、binlog

- undolog
  undolog的作用是保证事务的原子性以及实现多版本并发控制（MVCC）。undolog的数据格式是对应更新操作的相反语句，比如用户输入insert，那么在undolog中记录一条对应的delete。
- redolog
  1、redolog它能保证对于已经COMMIT的事务产生的数据变更，即使是系统宕机崩溃也可以通过它来进行数据重做，达到数据的持久性，一旦事务成功提交后，不会因为异常、宕机而造成数据错误或丢失。
  2、当redolog空间满了之后又会从头开始以循环的方式进行覆盖式的写入。MySQL支持三种将redo log buffer写入redo log file的时机，可以通过innodb_flush_log_at_trx_commit参数配置，各参数含义如下：
  0（延迟写）：表示每次事务提交时都只是把redolog留在redolog buffer中，开启一个后台线程，每1s刷新一次到磁盘中;
  1（实时写，实时刷）：表示每次事务提交时都将redolog直接持久化到磁盘，真正保证数据的持久性；
  2（实时写，延迟刷）：表示每次事务提交时都只是把redolog写到page cache，具体的刷盘时机不确定。
  除了上面几种机制外，还有其它两种情况会把redo log buffer中的日志刷到磁盘。
  定时处理：有线程会定时(每隔 1 秒)把redo log buffer中的数据刷盘。
  根据空间处理：redo log buffer占用到了一定程度(innodb_log_buffer_size设置的值一半)占，这个时候也会把redo log buffer中的数据刷盘。
  3、redolog有两个阶段prepare和commit，在prepare阶段，事务的更改首先被写入redolog，并标记为“准备(prepare)”状态。这意味着更改尚未真正提交，只是暂时保存在redolog中。当事务提交后，进入commit阶段，MySQL会确保这些更改被持久化到磁盘中。
  无论是用上面的何种写入配置，$redolog$的写入都是先从内存开始的。这通常涉及到将日志条目写入到$InnoDB$的$redolog$缓冲区
- binlog
  1、binlog记录了对MySQL数据库执行更改的所有的写操作，包括所有对数据库的数据、表结构、索引等等变更的操作。binlog的主要应用场景分别是主从复制和数据恢复，主从复制：在Master端开启binlog，然后将binlog发送到各个 Slave端，Slave端重放binlog来达到主从数据一致。数据恢复：通过使用mysqlbinlog工具来恢复数据。
  2、binlog日志有三种格式，分别为STATMENT、ROW和MIXED。
  STATMENT格式：保存执行的sql语句，但是如果语句中有随机数，会造成主从同步数据不一致。
  ROW格式：每次操作保存受影响的行，以及怎么变化。但是会产生大量的日志，比较浪费存储空间。
  MIXED格式：如果语句中没有随机数等，就保存sql语句，否则保存为row格式。
  3、通过 sync_binlog 参数控制 biglog 的刷盘时机，取值范围是 0-N：
  0：每次提交事务binlog不会马上写入到磁盘，而是先写到page cache。不去强制要求，由系统自行判断何时写入磁盘，在Mysql 崩溃的时候会有丢失日志的风险；
  1：每次提交事务都会执行fsync将binlog写入到磁盘；
  N：每次提交事务都先写到page cach，只有等到积累了N个事务之后才fsync将binlog写入到磁盘，在MySQL崩溃的时候会有丢失N个事务日志的风险。

MVCC

不同隔离级别的执行效率，读未提交>读已提交>可重复度>可串行化，为什么MVCC不针对读未提交和可串行化呢?
读未提交虽然效率高，但本身就破坏了数据库的ACID原则。
可串行化底层是通过加读写锁的方式来约束事务的并发，执行效率太低，而且违背了MVCC的原则。

- undolog、read view以及三个隐藏列（DB_ROW_ID：记录的主键id（没有指定主键时才会有）。DB_TRX_ID：事务ID，当对某条记录发生修改时，就会将这个事务的Id记录其中。DB_ROLL_PTR：回滚指针，版本链中的指针）
- 在读已提交隔离级别下每次查询都会生成read view，即在该隔离级别下是当前读。
- 在可重复读隔离级别下如果是普通的select，则只有在第一次执行select会生成read view，之后每次普通查询都不会再生成read view（即是当前读）。如果是加锁读（即select ... for update;或者select ... in share model;）则会生成read view。