mysql课件PPT


MySQL数据库高级
安大叔
besttest
1. MySQL架构
2. MySQL复制
3. MySQL优化
4. 事务与锁
内容提要
第一章 MySQL架构

- Why MySQL ?

为什么用MySQL？

- MySQL随互联网成长，自身也在飞速成长。其广泛使用还基于以下原因：
- 1.简单易用
- 2.成本低
- 3.易扩展
- 4.开源
- 5.高性能
- 6.复制功能领先

为什么用MySQL？

- MySQL用来做什么？

MySQL数据库用途

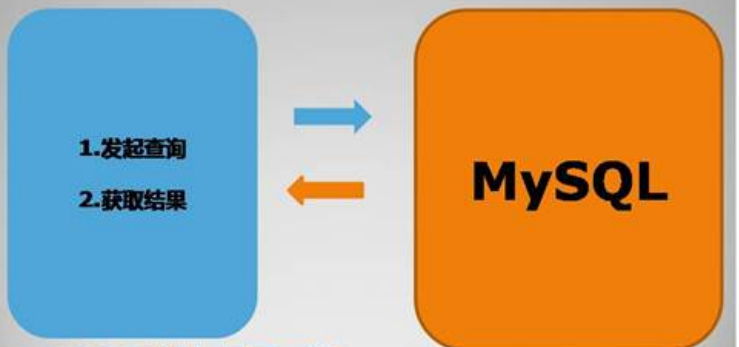
- 写配置，记录用户信息，记录广告资料信息，记录客户消耗的信息
- 读配置，读用户信息，读广告信息，读客户消耗信息
- 所有行为可以归结为写数据，读数据。

MySQL数据库用途

- MySQL是如何帮我们读数据？

MySQL数据库用途

- 1个查询sql在生命周期内的都做了什么？
- select name from table where id=1 ;



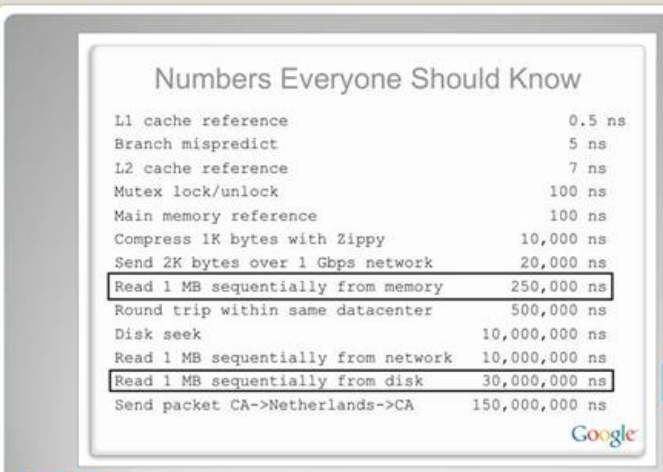
MySQL数据库用途

1.MySQL服务器监听3306端口
2.验证访问用户
3.创建MySQL线程
4.检查内存（Qcache）
5.解析SQL
6.生成查询计划
7.打开表
8.检查内存（Buffer Pool）
9.到磁盘取数据
10.写入内存
11.返回数据给客户端
12.关闭表
13.关闭线程
14.关闭连接

MySQL数据库用途

掀起你的盖头来 -- MySQL什么样？

MySQL架构



0.03秒

缓存

缓存1（MySQL层）：查询缓存 Query Cache
缓存2（存储引擎层）：InnoDB_Buffer_Pool

缓存

- MySQL查询缓存（qcache）
- 缓存完整的SELECT查询结果，当查询命中缓存，MySQL会立刻返回结果，跳过解析、优化和执行阶段。
- 查询缓存会跟踪系统中的每张表，如果这些表发生变化，那么和这张表相关的所有查询缓存全部失效。
- 查询缓存对应用程序是完全透明的。应用程序无须关心MySQL是通过查询缓存返回的结果还是实际执行返回的结果。事实上，这2种结果是完全相同的。

查询缓存-Qcache

- 在检查查询缓存的时候，MySQL不会对SQL进行任何处理，它精确的使用客户端传来的查询（select），只要字符大小写或者注释有一点点不同，查询缓存就认为是不同的查询。
- 任何一个包含不确定的函数(比如now(),current_date())的查询不会被缓存。

查询缓存-Qcache

- MySQL查询缓存可以改善性能，但是在使用的时候也有一些问题需要注意：
- 开启查询缓存对于读写都增加了额外的开销。对于读，在查询开始前需要先检查缓存；对于写，在写入后需要更新缓存。
- 一般情况这些开销相对较小，所以查询缓存一般还是有好处的。但也要根据业务特征权衡是否需要开启查询缓存。

查询缓存-Qcache

- 查询缓存参数：

1.query_cache_type
是否开启查询缓存，具体选项是off，on
2.query_cache_size
分配给查询缓存的总内存，一般建议不超过256M
3.query_cache_limit
这个选项限制了MySQL存储的最大结果。如果查询的结果比这个大，那么就不会被缓存。

查询缓存参数

- 实际案例word文档

查询缓存缓存案例

- InnoDB有buffer pool（简称bp）是数据库页面的缓存，对InnoDB的任何修改操作都会首先在bp的page上进行，然后这样的页面将被标记为dirty并被放到专门的flush list上，后续将由master thread或专门的刷脏线程阶段性的将这些页面写入磁盘（disk or ssd）。这样的好处是避免每次写操作都操作磁盘导致大量的随机IO，阶段性的刷脏可以将多次对页面的修改merge成一次IO操作，同时异步写入也降低了访问的时延。

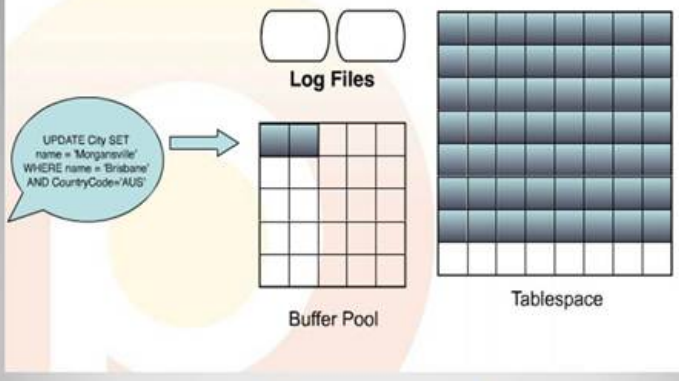
缓存-InnoDB-bp

- 如果在dirty page还未刷入磁盘时，server非正常关闭，这些修改操作将会丢失，如果写入操作正在进行，甚至会由于损坏数据文件导致数据库不可用。为了避免上述问题的发生，InnoDB将所有对页面的修改操作写入一个专门的文件，并在数据库启动时从此文件进行恢复操作，这个文件就是redo log file。这样的技术推迟了bp页面的刷新，从而提升了数据库的吞吐，有效的降低了访问时延。带来的问题是额外的写redo log操作的开销（顺序IO速度快），以及数据库启动时恢复操作所需的时间。

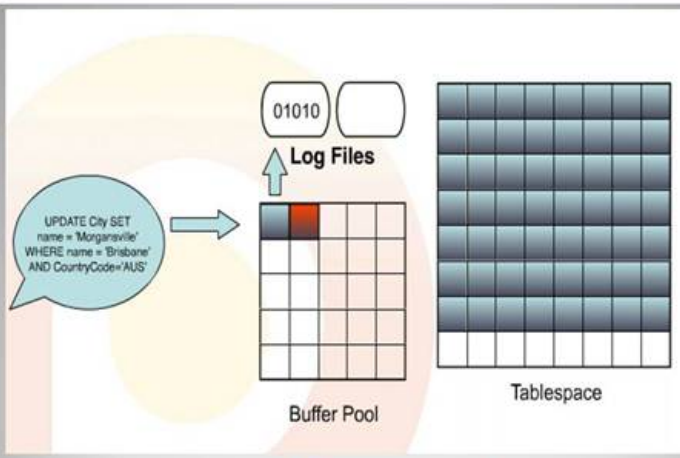
缓存-InnoDB-bp

- 存储引擎层缓存（InnoDB-bp）的设置
- 同qcache不同，InnoDB-bp一定是有好处的，而且设置越大，MySQL性能越好。考虑到MySQL的其他模块的内存需求，一般一台专用的MySQL服务器的InnoDB-bp大小设置为物理内存的70%。

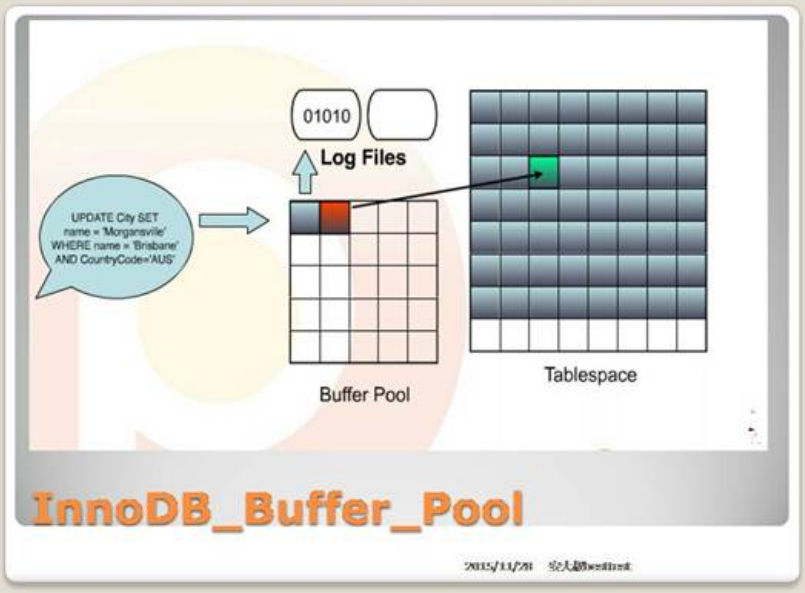
缓存-InnoDB-bp



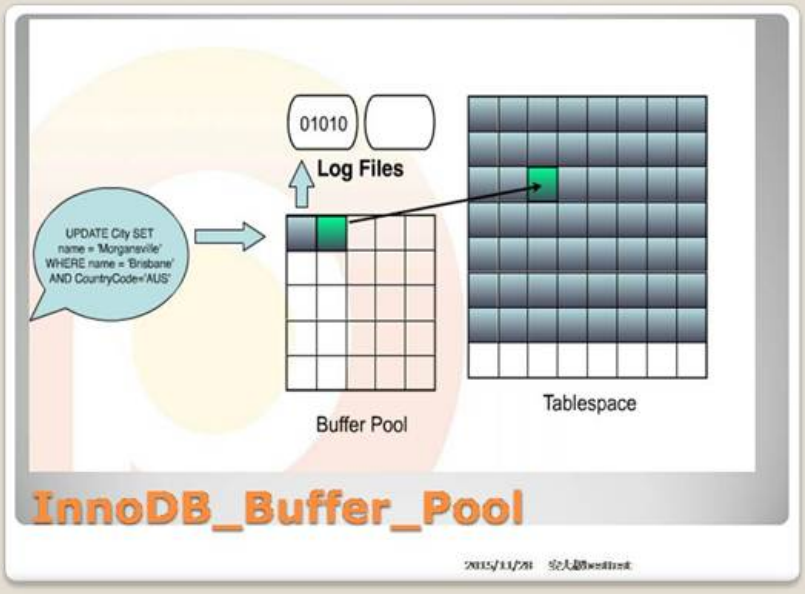
InnoDB_Buffer_Pool



InnoDB_Buffer_Pool



InnoDB_Buffer_Pool



InnoDB_Buffer_Pool

- 小结：
- InnoDB把常用数据加载到内存中，这样零散操作在内存中快速完成。随后再把更新批量写回磁盘，大大提高了写入效率。
- 为避免突然宕机造成的数据丢失，通过顺序记录redo log来确保数据安全。

InnoDB_Buffer_Pool

第二章 MySQL复制

- MySQL内建的复制功能是构建大型、高性能应用程序的基础。
- 复制解决的基本问题是让一台服务器的数据和其他的服务器保持同步。一台主服务器的数据可以同步到多台从服务器上。并且从服务器也可以被配置为另外一台服务器的主库。主库和从库之间可以有多种不同的组合方式。
- 此外复制的方式还能被配置，可以对整台服务器，特定数据库甚至特定表进行复制。

复制概述

- MySQL支持两种复制方案：基于语句的复制（statement-based replication）和基于行的复制（Row-based replication）。基于语句的复制在MySQL3.23就已经存在，它是使用较多的复制方式。基于行的复制是MySQL5.1引入的。这2种复制方式都是通过记录主服务器的二进制日志，并在从服务器进行重放（replay）完成复制。它们都是异步的。也就是说，从服务器上的数据并非都是最新的。

复制概述

- 复制通常不会增加主库的开销，主要是启用二进制日志带来的开销，但出于备份或及时从崩溃中恢复的目的，这些开销是必要的。
- 通过复制可以将读操作指向从库来获得更好的读扩展，但对于写操作，并不适合通过复制来扩展。

复制概述

- 分布数据

MySQL通常不会对带宽造成很大的压力。因此可以在不同的地理位置来分布数据，实现跨机房跨地域的数据分布。

- 负载均衡

通过MySQL复制可以将读操作分布到多个服务器上，实现对读密集型应用的优化

复制用途

- 备份：

复制对备份很有帮助，但是从服务器并不是备份。

- 高可用性和故障转移

复制可以避免在应用程序中出现MySQL失效。好的故障转移能显著的减少停机时间，甚至让用户无感知。

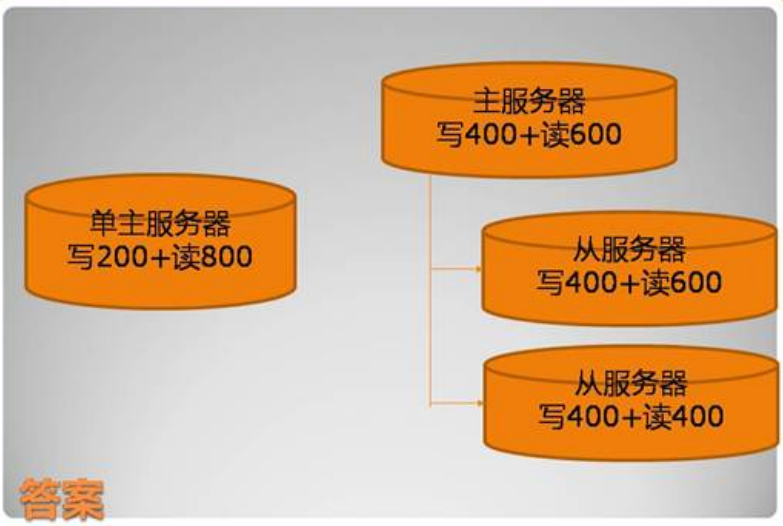
- 测试MySQL版本升级

一个常见的方法是先把从服务器升级到MySQL新版本，然后用它来测试查询，确保无异常后再升级主服务器。

复制用途

- 某业务读写比为1:4，此前数据库没有使用从库。只有一台主库。恰使主库在理论满载下运行。现在业务访问量翻倍，希望通过配置从库来解决，需要几台从库。
- 假设条件：
- 1.读写查询包含同样的工作量
- 2.所有服务器等同，每秒能进行1000次查询
- 3.业务访问是均匀而稳定的
- 4.从库和主库有同样的性能特征。

复制思考

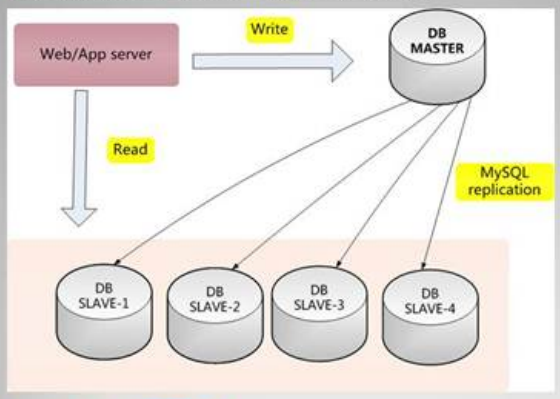


答案

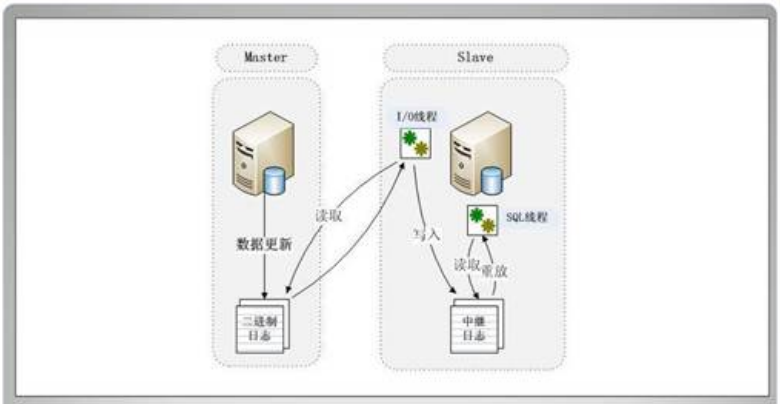
- 案例：
- 客户投诉：修改了xx信息，提示修改成功。
- 再查看还是旧数据。怎么回事？

MySQL复制

- 很可能是数据库主从延迟造成



MySQL复制



MySQL复制

- MySQL复制小结：
- 3个线程完成复制：
- 主库1个线程负责记录数据库变更日志
- 从库1个线程负责拉取主库的变更日志
- 从库1个线程负责执行主库的变更日志

- 实现了获取事件和重放事件的解耦，允许异步进行。
- 复制的瓶颈：主库并行(多线程)写入和从库串行（单线程）写入，会造成主从延迟。

MySQL复制

MySQL主从延时延时问题如何处理?
1.偶发性延时：
控制写入速度，削峰填谷。
2.频发性延时：
拆分数据库实现多点写入
最后一招：从库磁盘硬件升级为ssd

MySQL复制

- 基于语句的复制（statement-based replication）：
- MySQL5.0之前只支持这种复制方式。这种复制方式实现简单，简单记录并重放sql语句就可以使从服务器和主服务器同步。另一个优点是二进制日志紧凑，使得复制占用的带宽较少（更改上GB数据的查询在日志里可能只有几十个字节）
- 但是在实际使用中也有一些问题，例如命令在主从服务器上执行的时间可能有些或非常不同，这样就需要在MySQL二进制日志中包含除文本之外的信息，比如当前的时间戳。尽管如此还是有一些命令不能被正确的复制，比如使用了current_user()函数的查询。

更深入的理解复制

- 基于行复制（row-based replication）：
- 从MySQL5.1开始增加了对基于行的复制的支持。这种方式把实际的数据更改记录到二进制日志中。
- 最大的优点是它保证了主从服务器的强一致性。并且一些命令可以更有效的复制（？）
- 主要缺点是二进制日志会变的很大，增加了写盘和网络带宽占用。并且更新数据的可见度降低，也增加了基于时间点恢复数据的难度。

更深入的理解复制

- mysql-bin.index

这个文件中每一行包含了二进制日志文件的文件名

- Mysql-relay-bin.index

同上，这个文件记录了中继日志文件的文件名。

- Master.info

这个文件包含了从服务器连接主服务器的所有信息。它的格式是纯文本的。如果删除它，从服务器重启后就不知道如何连接主服务器。

- Relay-log.info

该文件包含了从服务器的当前二进制日志和中继日志的偏移量。如果删除它，从服务器在重启后就不知道复制进行到的位置，有可能重复复制。（有故事！）

复制文件

第三章 MySQL优化

- 数据库优化？

优化要从整个业务逻辑上进行。针对数据库问题进行的优化，首先要考虑不查或少查数据库。

请看案例：《数据库业务优化的实例》

如果查询不可避免，可以考虑两种优化方式：

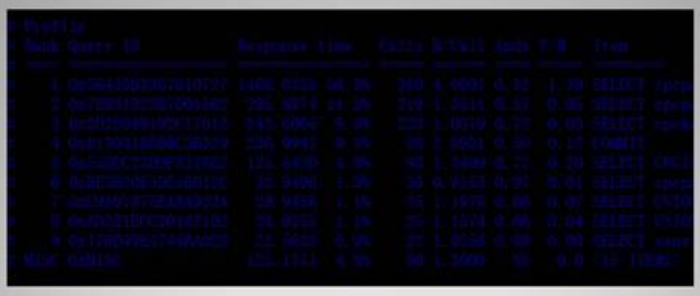
1.避免磁盘IO,也就是让查询在内存中完成。

2.通过sql和索引的调整，让MySQL用更高效率的方式查询。

优化概述

- 现象：
- Cpu占用高，I/O使用高，负载高，MySQL连接线程高，MySQL运行线程高。主从延时不断增加。
- 问题在哪里？
- 数据库的问题肯定是sql带来的，那么是哪些sql呢？最常用的方式是使用pt-query-digist分析慢查询日志。定位到最严重的sql。
- 命令：
- pt-query-digest --since='2015-11-21 09:57:01' slow_my3306.log > slow.out

如何诊断和定位问题



如何诊断和定位问题

- 针对占用资源最严重的sql进一步的分析是什么？
- Explain

Explain

- Explain :该命令是查看查询优化器如何决定执行查询的主要方法，这个功能有局限性，只是一个近似结果，有时它是一个很好的近似，有时可能相差甚远。但它的输出是可以获取的最准确信息，值得仔细学习。
- 一个简单的执行计划

```
mysql>explain select * from students
where scode in (2,3,4);
```

explain


```
mysql> explain select * from students where scode IN (2,3,4)\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: students
   partitions: NULL
         type: range
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: NULL
         rows: 3
     filtered: 100.00
        Extra: Using where
1 row in set, 1 warning (0.00 sec)
```

explain

- 1.explain不会考虑触发器、存储过程或函数对查询的影响。
- 2.explain不会考虑缓存对查询结果的影响
- 3.不会考虑mysql执行查询所做的特定优化
- 4.是基于统计信息的估算，并非精确值。
- 5.不能区分同名事物。比如，对内存排序和临时文件都使用“filesort”；对磁盘上和内存中的临时表都显示“using temporary”
- 6.mysql5.6之前只支持对select进行explain

explain的局限性

- ID列
- 这一列就是1个编号。如果语句中没有子查询或联合查询，那么每一行都会显示1.否则内层的select语句会顺序编号。
- id如果相同，可以认为是一组，从上往下顺序执行，这个顺序不总是与其原始SQL中的相一致；在所有组中，id值越大，优先级越高，越先执行。

explain中的列

- Select_type列：
- 这一列显示了对应行是简单还是复杂select（如果是后者，那么是3种复杂类型中的哪一种）。
- SIMPLE值意味着查询不包括子查询和UNION。如果查询有任何复杂的子部分，则最外层部分标记为PRIMARY，其他部分标记如下：

explain中的列

- SUBQUERY
- 包含在select列表中的子查询中的select，也就是不在from字句中。

```
mysql> explain select (select 1 from test) from test;
+----+-------------+-------+------+---------------+--
| id | select_type | table | type | possible_keys | k
+----+-------------+-------+------+---------------+--
|  1 | PRIMARY     | test  | ALL  | NULL          | N
|  2 | SUBQUERY    | test  | ALL  | NULL          | N
+----+-------------+-------+------+---------------+--
2 rows in set (0.00 sec)
```

Explain中的列（select_type）

- DERIVED:
- DERIVED值表示包含在FROM字句的子查询中的SELECT，MySQL会递归执行并将结果放到一个临时表中。服务器内部称其“派生表”，因为该临时表是从子查询中派生来的。

```
mysql> explain select * from (select * from test) as t;
+----+-------------+------------+------+---------------+
| id | select_type | table      | type | possible_keys |
+----+-------------+------------+------+---------------+
|  1 | PRIMARY     | <derived2> | ALL  | NULL          |
|  2 | DERIVED     | test       | ALL  | NULL          |
+----+-------------+------------+------+---------------+
2 rows in set (0.03 sec)
```

Explain中的列（select_type）

- UNION
- 在UNION中的第二个和随后的SELECT被标记为UNION。第一个SELECT被标记就好像它以部分外查询来执行。如果UNION被FROM字句包含，那么它的第一个SELECT会被标记为DERIVED
- UNION RESULT
- 用来从UNION的匿名临时表检索结果的SELECT被标记为此。

Explain中的列（select_type）


```
mysql> explain select 1 union select 2;
+------+--------------+------------+------+---------------+
| id   | select_type  | table      | type | possible_keys |
+------+--------------+------------+------+---------------+
|    1 | PRIMARY      | NULL       | NULL | NULL          |
|    2 | UNION        | NULL       | NULL | NULL          |
| NULL | UNION RESULT | <union1,2> | ALL  | NULL          |
+------+--------------+------------+------+---------------+
3 rows in set (0.01 sec)
```

```
mysql> explain select 1 from (select 1 union select 2);
ERROR 1248 (42000): Every derived table must have its own alias
mysql> explain select 1 from (select 1 union select 2) as tb1;
+------+--------------+------------+------+---------------+------+
| id   | select_type  | table      | type | possible_keys | key  |
+------+--------------+------------+------+---------------+------+
|    1 | PRIMARY      | <derived2> | ALL  | NULL          | NULL |
|    2 | DERIVED      | NULL       | NULL | NULL          | NULL |
|    3 | UNION        | NULL       | NULL | NULL          | NULL |
| NULL | UNION RESULT | <union2,3> | ALL  | NULL          | NULL |
+------+--------------+------------+------+---------------+------+
4 rows in set (0.00 sec)
```

Explain中的列（select_type）

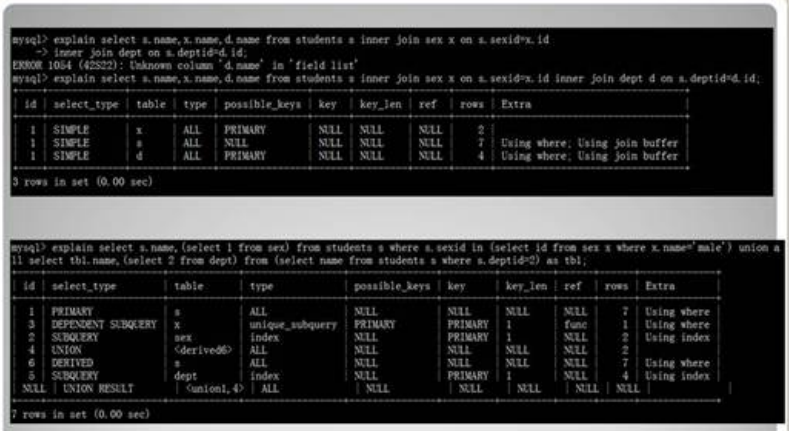
- Table列

这一列显示了对应行正在访问那个表。通常情况，它就是那个表或是该表的别名。

可以从这一列中从上往下观察MySQL的关联优化器为查询选择的关联顺序。

当在FROM字句有子查询时，table列是<derived N>的形式

Explain中的列（table列）



```
mysql> explain select s.name,x.name,d.name from students s inner join sex x on s.sexid=x.id
    -> inner join dept on s.deptid=d.id;
ERROR 1054 (42S22): Unknown column 'd.name' in 'field list'
mysql> explain select s.name,x.name,d.name from students s inner join sex x on s.sexid=x.id inner join dept d on s.deptid=d.id;
+----+-------------+-------+------+---------------+------+---------+------+------+----------------------------------------------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra                                              |
+----+-------------+-------+------+---------------+------+---------+------+------+----------------------------------------------------+
|  1 | SIMPLE      | x     | ALL  | PRIMARY       | NULL | NULL    | NULL |    2 |                                                    |
|  1 | SIMPLE      | s     | ALL  | NULL          | NULL | NULL    | NULL |    7 | Using where; Using join buffer                     |
|  1 | SIMPLE      | d     | ALL  | PRIMARY       | NULL | NULL    | NULL |    4 | Using where; Using join buffer                     |
+----+-------------+-------+------+---------------+------+---------+------+------+----------------------------------------------------+
3 rows in set (0.00 sec)
```



```
mysql> explain select s.name,(select 1 from sex) from students s where s.sexid in (select id from sex x where x.name='male') union a
ll select tbl.name,(select 2 from dept) from (select name from students s where s.deptid=2) as tbl;
+----+--------------------+------------+-----------------+---------------+---------+---------+------+------+-----------------+
| id | select_type        | table      | type            | possible_keys | key     | key_len | ref  | rows | Extra           |
+----+--------------------+------------+-----------------+---------------+---------+---------+------+------+-----------------+
|  1 | PRIMARY            | s          | ALL             | NULL          | NULL    | NULL    | NULL |    7 | Using where     |
|  3 | DEPENDENT SUBQUERY | x          | unique_subquery | PRIMARY       | PRIMARY | 1       | func |    1 | Using where     |
|  2 | SUBQUERY           | sex        | index           | NULL          | PRIMARY | 1       | NULL |    2 | Using index     |
|  4 | UNION              | <derived6> | ALL             | NULL          | NULL    | NULL    | NULL |    2 |                 |
|  6 | DERIVED            | s          | ALL             | NULL          | NULL    | NULL    | NULL |    7 | Using where     |
|  5 | SUBQUERY           | dept       | index           | NULL          | PRIMARY | 1       | NULL |    4 | Using index     |
| NULL | UNION RESULT     | <union1,4> | ALL             | NULL          | NULL    | NULL    | NULL | NULL |                 |
+----+--------------------+------------+-----------------+---------------+---------+---------+------+------+-----------------+
7 rows in set (0.00 sec)
```

Explain中的列（table列）

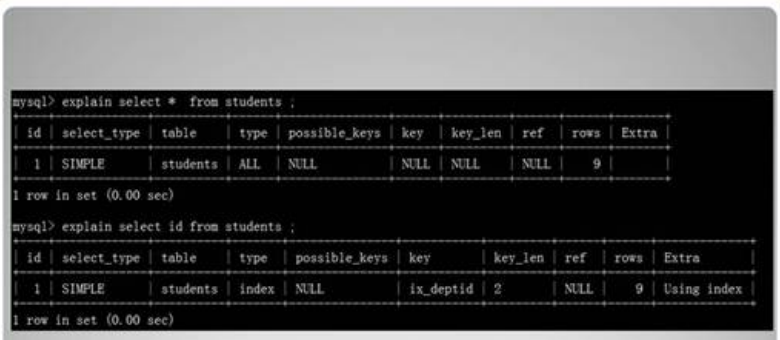
- Type列
- 指MySQL的访问类型，也就是如何查找表中的行。下面是最重要的访问方法，依次从最差到最优。

all<index<range<ref<eq_ref<const,system<null

Explain中的列（type列）

- ALL
- 就是全表扫描，通常意味着MySQL必须扫描整张表，从头到尾，去找需要的行。对innodb表就是按主键顺序。
- Index
- 跟全表扫描一样，只是MySQL扫描表时按索引的次序进行而不是行。它的优点是避免了排序，缺点是要承担按索引次序读取整个表的开销。
- 如果在Extra列中看到“using index”说明MySQL正在使用覆盖索引，它只扫描索引的数据，而不用回表查询。

Explain中的列（type列）



```
mysql> explain select *  from students ;
+----+-------------+----------+------+---------------+------+---------+------+------+-------+
| id | select_type | table    | type | possible_keys | key  | key_len | ref  | rows | Extra |
+----+-------------+----------+------+---------------+------+---------+------+------+-------+
|  1 | SIMPLE      | students | ALL  | NULL          | NULL | NULL    | NULL |    9 |       |
+----+-------------+----------+------+---------------+------+---------+------+------+-------+
1 row in set (0.00 sec)

mysql> explain select id from students ;
+----+-------------+----------+-------+---------------+-----------+---------+------+------+-------------+
| id | select_type | table    | type  | possible_keys | key       | key_len | ref  | rows | Extra       |
+----+-------------+----------+-------+---------------+-----------+---------+------+------+-------------+
|  1 | SIMPLE      | students | index | NULL          | ix_deptid | 2       | NULL |    9 | Using index |
+----+-------------+----------+-------+---------------+-----------+---------+------+------+-------------+
1 row in set (0.00 sec)
```

Explain中的列（type列）

- Range
- 范围扫描就是一个有限制的索引扫描，它开始于索引里面的一个点，返回匹配整个值域的行。这比全索引扫描好一些，因为它用不着遍历全部索引。显而易见的是between或在where字句带有“>”或“<”的查询。
- Ref
- 这是一种索引访问，它返回所有匹配某个单个值的行，然而它可能找到多个符合条件的行，因此它是查找和扫描的混合体。此类索引访问只有当使用非唯一索引或唯一索引的非唯一性前缀时才会发生。

Explain中的列（type列）

```
mysql> explain select id,name from students where deptid >4;
+----+-------------+----------+-------+---------------+-----------+---------+------+------+-------------+
| id | select_type | table    | type  | possible_keys | key       | key_len | ref  | rows | Extra       |
+----+-------------+----------+-------+---------------+-----------+---------+------+------+-------------+
|  1 | SIMPLE      | students | range | ix_deptid     | ix_deptid | 2       | NULL |    1 | Using where |
+----+-------------+----------+-------+---------------+-----------+---------+------+------+-------------+
1 row in set (0.00 sec)

mysql> explain select id,name from students where deptid between 4 and 6;
+----+-------------+----------+-------+---------------+-----------+---------+------+------+-------------+
| id | select_type | table    | type  | possible_keys | key       | key_len | ref  | rows | Extra       |
+----+-------------+----------+-------+---------------+-----------+---------+------+------+-------------+
|  1 | SIMPLE      | students | range | ix_deptid     | ix_deptid | 2       | NULL |    3 | Using where |
+----+-------------+----------+-------+---------------+-----------+---------+------+------+-------------+
1 row in set (0.00 sec)

mysql> explain select id,name from students where deptid between 3 and 6;
+----+-------------+----------+------+---------------+------+---------+------+------+-------------+
| id | select_type | table    | type | possible_keys | key  | key_len | ref  | rows | Extra       |
+----+-------------+----------+------+---------------+------+---------+------+------+-------------+
|  1 | SIMPLE      | students | ALL  | ix_deptid     | NULL | NULL    | NULL |    9 | Using where |
+----+-------------+----------+------+---------------+------+---------+------+------+-------------+
1 row in set (0.00 sec)
```

Explain中的列（type列）


```
mysql> explain select id from students where deptid=5;
+----+-------------+----------+------+---------------+-----------+---------+-------+------+--------------------------+
| id | select_type | table    | type | possible_keys | key       | key_len | ref   | rows | Extra                    |
+----+-------------+----------+------+---------------+-----------+---------+-------+------+--------------------------+
|  1 | SIMPLE      | students | ref  | ix_deptid     | ix_deptid | 2       | const |    1 | Using where; Using index |
+----+-------------+----------+------+---------------+-----------+---------+-------+------+--------------------------+
1 row in set (0.00 sec)

mysql> explain select * from students where deptid=5;
+----+-------------+----------+------+---------------+-----------+---------+-------+------+-------------+
| id | select_type | table    | type | possible_keys | key       | key_len | ref   | rows | Extra       |
+----+-------------+----------+------+---------------+-----------+---------+-------+------+-------------+
|  1 | SIMPLE      | students | ref  | ix_deptid     | ix_deptid | 2       | const |    1 | Using where |
+----+-------------+----------+------+---------------+-----------+---------+-------+------+-------------+
1 row in set (0.00 sec)
```

Explain中的列（type列）

- Eq_ref
- 使用这种索引查找，MySQL知道最多只返回一条符合条件的记录。这种访问方式在MySQL使用主键或唯一索引时可以看到。MySQL对于这类访问类型优化的非常好。
- Const，system
- 当MySQL能对查询的某部分进行优化并将其转化成一个常量时，它就会使用这些类型访问。

Explain中的列（type列）

```
mysql> explain select *  from students where name='test3';
+----+-------------+----------+-------+---------------+---------+---------+-------+------+-------+
| id | select_type | table    | type  | possible_keys | key     | key_len | ref   | rows | Extra |
+----+-------------+----------+-------+---------------+---------+---------+-------+------+-------+
|  1 | SIMPLE      | students | const | uq_name       | uq_name | 99      | const |    1 |       |
+----+-------------+----------+-------+---------------+---------+---------+-------+------+-------+
1 row in set (0.00 sec)

mysql> explain select *  from students s left join sex x on s.sexid=x.id ;
+----+-------------+-------+--------+---------------+---------+---------+--------------+------+-------+
| id | select_type | table | type   | possible_keys | key     | key_len | ref          | rows | Extra |
+----+-------------+-------+--------+---------------+---------+---------+--------------+------+-------+
|  1 | SIMPLE      | s     | ALL    | NULL          | NULL    | NULL    | NULL         |    9 |       |
|  1 | SIMPLE      | x     | eq_ref | PRIMARY       | PRIMARY | 1       | test.s.sexid |    1 |       |
+----+-------------+-------+--------+---------------+---------+---------+--------------+------+-------+
2 rows in set (0.00 sec)
```

Explain中的列（type列）

- NULL
- 这种访问方式意味着MySQL能在优化阶段分解查询语句，在执行阶段甚至用不着再访问表或者索引。

Explain中的列（type列）


```
mysql> explain select max(id) from students;
+----+-------------+-------+------+---------------+------+---------+------+------+------------------------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra                        |
+----+-------------+-------+------+---------------+------+---------+------+------+------------------------------+
|  1 | SIMPLE      | NULL  | NULL | NULL          | NULL | NULL    | NULL | NULL | Select tables optimized away |
+----+-------------+-------+------+---------------+------+---------+------+------+------------------------------+
1 row in set (0.00 sec)

mysql> explain select max(sexid) from students;
+----+-------------+----------+------+---------------+------+---------+------+------+-------+
| id | select_type | table    | type | possible_keys | key  | key_len | ref  | rows | Extra |
+----+-------------+----------+------+---------------+------+---------+------+------+-------+
|  1 | SIMPLE      | students | ALL  | NULL          | NULL | NULL    | NULL |    7 |       |
+----+-------------+----------+------+---------------+------+---------+------+------+-------+
1 row in set (0.00 sec)

mysql> explain select max(deptid) from students;
+----+-------------+-------+------+---------------+------+---------+------+------+------------------------------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra                        |
+----+-------------+-------+------+---------------+------+---------+------+------+------------------------------+
|  1 | SIMPLE      | NULL  | NULL | NULL          | NULL | NULL    | NULL | NULL | Select tables optimized away |
+----+-------------+-------+------+---------------+------+---------+------+------+------------------------------+
1 row in set (0.00 sec)
```

Explain中的列（type列）

- Possible_keys列
- 这一列显示了查询可以使用哪些索引，这是基于查询访问的列和使用的比较操作符来判断的。这个列是在优化过程早期创建的，因此有些罗列出来的索引可能对于后续优化过程无用。
- Key列
- 这一列显示了MySQL决定采用哪个索引来优化对该表的访问。如果该索引没有出现在possible_keys中，那么MySQL选用它是出于另外的原因，比如覆盖索引。

Explain中的列

- Key_len列：
- 该列显示了MySQL在索引里使用的字节数，如果MySQL正在使用的只是索引里的某些列，那么就可以用这个值来计算出具体是哪些列。
- Ref列：
- 这一列显示了之前的表在key列记录的索引中查找值所有的列或常量。

Explain中的列

- Rows列
- 这一列是MySQL估计为了找到所需的行而要读取的行数。这个数字是内嵌循环关联计划里的循环数目。它不是MySQL认为它最终读取的行数，而是为了找到符合查询的每一点上标准的那些行而必须读取的行的平均数。
- FILTERED列
- 这一列是MySQL5.1新加的列，在使用explain extended时出现。它显示的是针对表里符合某个条件的记录数的百分比所做的一个悲观估算。

Explain中的列

- Extra列：

这一列包含的是不适合在其他列显示的额外信息。

- 一些比较重要的如下：

“using index”

此值表示MySQL将使用覆盖索引，以避免访问表。

“using where”

表示MySQL服务器将在存储引擎检索行后再进行过滤。

Explain中的列

- “using temporary”

这意味着MySQL在对查询结果排序时会使用一个临时表。

- “using filesort”

这意味着MySQL会对结果使用一个外部的索引排序，而不是按索引顺序从表里读取行。

Explain中的列

- Explain能做什么？
- Explain各列的含义是什么？

Explain小结

- 性能优化案例：

1. sql优化-sysdate函数问题

Explain应用-性能优化案例

- MySQL开发和使用规范

MySQL开发和使用规范

- 使用原则

让数据库做她擅长的事
- 尽量不在数据库做运算
- 复杂运算移动到程序端cpu
- 尽可能简单应用MySQL

拒绝3B
- BIG SQL
- BIG Transaction
- BIG Batch

平衡范式与冗余
- 严格遵循三大范式？
- 效率优先，提升性能
- 适当时牺牲范式，加入冗余

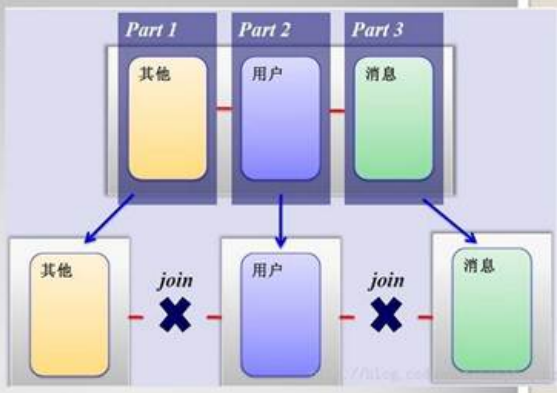
通用原则

性能阈值

- 纯int不超过1kw 含char不超过0.5kw
- 索引不超过5个
- 单库300-400张表
- 并发数

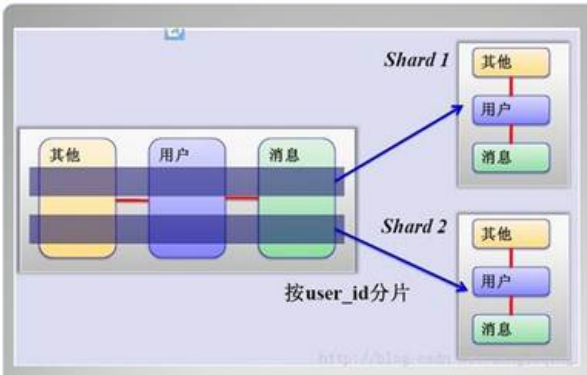
分库分表

- 垂直分区
  - 业务分离
- 水平分区
  - 数据分离
- 其他
  - 时间转储
  - 热冷分离



通用建议

2015/11/28 安大霸weiliwei



水平分区

2015/11/28 安大霸weiliwei

table_option:

```
ENGINE [=] engine_name                          = INNODB
| AUTO_INCREMENT [=] value                      =1
| AVG_ROW_LENGTH [=] value
| [DEFAULT] CHARACTER SET [=]                   = UTF8
| CHECKSUM [=] {0 | 1}
| [DEFAULT] COLLATE [=] collation_name           = utf8_general_ci
| COMMENT [=]   'string'
...
| ROW_FORMAT [=]
| TABLESPACE tablespace_name [STORAGE {DISK|MEMORY|DEFAULT}]
| UNION [=] (tbl_name[,tbl_name]...)
```

表设计

2015/11/28 安大霸weiliwei

- col_name column_definition
  - 类型精确（更小，简单）
  - Varchar()类型使用
  - 避免text/blob,不要存储图片
  - 时间类型(year,date,timestamp,datetime)
  - Int()的含义
  - 尽量NOT NULL
  - 使用int unsigned 存储IP 地址（效率更好）
  - int类型替代枚举
  - 建议comment

表设计（列）

- Key_definition：
  - Primary K(必用，自增)
  - Unique K
  - Foreign K(禁用)
  - Normal K

表设计（索引）

Innodb存储引擎支持两种常见的索引。

- 哈希索引 是innodb自维护的
- B+树是为磁盘及其他存储辅助设备而设计一种平衡查找树（不是二叉树）

B+树

- 插入操作
- 删除操作
- 旋转操作（分页合页耗资源）

索引设计

- 索引类型
  - 聚集索引（避免对此进行更新操作）
  - 辅助索引
- 使用索引原则
  - 谨慎合理添加索引
    - 改善查询
    - 影响更新性能
    - 索引非越多越好
  - 针对核心SQL，设计索引

B+tree索引

适用条件：
- 匹配全名
- 匹配最左原则
- 匹配列前缀
- 匹配范围值
- 精确匹配一部分且匹配某个范围的另一部分
- 只访问索引的查询

局限：
- 不能跳过最左原则
- 不能跳过索引中的列
- 存储引擎不能优化访问任何在第一个范围条件右边的列

B+tree索引

- 隔离列
  - id + 1=5
  - to_days(createdate)=xx
- 前缀索引和索引选择性
  - varchar()或者text/blob
  - 预估前缀索引的选择性
    - count distinct(left(col，n))/count(*) 越大越好
    - 评估n的取值长度
- 优先使用主键
  - 索引+行数据
  - 省去二次IO
  - 选择性高

高性能索引策略

- 覆盖索引
  - 只走索引树，不取行数据
- 为排序使用索引扫描
  - Order by
  - Group by
- 多余和重复索引
  - 联合索引

高性能索引策略

- 优化count
  - count迷思
  - 使用memcache或汇总表
- 优化联接
  - 确保on或using使用的列有索引（联接顺序）
  - 确保group by或order by 只引用一个表中的列
  - 谨慎的升级MySQL
- 优化子查询
  - 尽可能的使用连接
  - 优先使用独立子查询

索引案例

- 优化group by和distinct
  - 分组+排序
  - Order by null
- 优化limit 和offset
  - 偏移量越大越慢
  - 推荐使用位置查询
    - Select * from table WHERE id>=23423 limit 11;
  - 使用覆盖索引
    - SELECT * FROM table INNER JOIN (SELECT id FROM table LIMIT 10000,10) USING (id) ;
- 优化union
  - union一定会使用临时表，mysql对临时表的优化不大
  - ‘下推’条件过滤语句（where，limit，order by）
  - 始终使用union all，避免mysql临时表处理重复的行

索引案例

- 避免大SQL
  - 一个SQL只能在一个cpu上运行
  - 高并发环境中，大SQL容易影响性能问题
  - 可能一个大SQL把数据库搞死
  - 拆分SQL
- 保持事务短小精悍
  - 即开即用，用完即关
  - 无关操作踢出事务，减少资源占用
  - 保持一致性的前提下，拆分事务
- 避免大批量更新
  - 避开高峰
  - 白天限制速度
  - 加sleep

SQL使用建议

- 避免使用select * ,只取有关列
- 避免取过量数据，建议使用limit
- 避免OR
  - 同一字段，推荐in
  - 不同字段，推荐union
- 避免在SQL 语句中进行数学运算、函数计算、逻辑判断等操作
- update，delete 语句避免使用limit
- 尽量避免负向查询，如NOT、!=、<>、!<、!>、NOT EXISTS、NOT IN、NOT LIKE 等
- 对同一个表的alter操作，要合并语句

SQL使用建议

- 优化案例：
- 1.子查询优化
- 2.单条sql问题-优化复杂sql解决
- 3.数据库业务优化的实例
- 4.单条sql问题-根据业务逻辑优化sql

优化案例集锦

第四章 锁与事务

- 什么是锁？
- 锁是数据库系统区别于文件系统的一个关键特性。锁机制用于管理对共享资源的并发访问，一方面最大程度的提供并发访问，另一方面要确保每个用户能以一致性的方式读取和修改数据。
- InnoDB存储引擎会在行级别上对表数据加锁，也会在数据库内部其他地方使用锁，比如bp中的LRU列表。

什么是锁

- 对应MyISAM存储引擎来说，其锁是表锁。并发情况下的读没有问题，但是并发写入性能比较差。
- InnoDB存储引擎是行锁，提供一致性的非锁定读。它的行级锁没有开销，可以同时得到并发性和一致性。

不同存储引擎的锁

- latch
  latch称为闩锁(轻量级的锁)，因为其要求锁定的时间必须非常短。若持续的时间长，则应用的性能会非常差。在InnoDB存储引擎中，又可以分为mutex(互斥量)和rwlock(读写锁)。其目的是用来保证并发线程操作临界资源的正确性，并且通常没有死锁检测的机制。

Lock与latch

- Show engine innodb mutex

```
mysql> show engine innodb mutex;
+---------+-----------------------------+----------+
| Type    | Name                        | Status   |
+---------+-----------------------------+----------+
| InnoDB | rwlock: log0log.cc:785       | waits=2 |
| InnoDB | sum rwlock: buf0buf.cc:1379 | waits=9 |
+---------+-----------------------------+----------+
2 rows in set (0.00 sec)
```

- 列Type显示的总是InnoDB，列Name显示latch的信息以及所在源码的行数，列Status中显示的os_waits表示操作系统等待的次数。

Lock与latch

- lock的对象是事务，用来锁定的是数据库中的对象，如表、页、行。并且一般lock的对象仅在事务commit或者rollback后释放(不同事务隔离级别释放的时间可能不一样)。
- 特点：
  InnoDB是通过对索引上的索引项加锁来实现行锁。这种特点也就意味着，只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁。

Lock

- latch
  - 对象：线程
  - 保护：内存数据结构
  - 持续时间：临界资源
  - 模式：读写锁，互斥量
  - 死锁：程序保证
  - 存在于：每个数据结构对象中

Latch与lock

- lock
  - 对象：事务
  - 保护：数据库内容
  - 持续时间：整个事务
  - 模式：行锁
  - 死锁：waiting-for graph，time out
  - 存在于：lock manager的哈希表

Latch与lock

- Record lock:单个行记录上的锁
- Gap lock：间隙锁，锁定一个范围，但不包括记录本身
- Next-key lock：锁定一个范围，包括记录本身。

InnoDB的3种行锁的设计

- 事务是数据库区别于文件系统的重要特性之一，它会把数据库从一种一致性状态转换到另一种一致性状态。在数据库提交工作时，可以确保其要么所有修改都已经保存了，要么所有修改都不保存，InnoDB存储引擎完全符合ACID特性
- ACID：
  - 原子性（atomicity）
  - 一致性（consistency）
  - 隔离性（isolation）
  - 持久性（durability）

事务

- 原子性（A）：
- 指整个数据库事务是不可分割的工作单位。只有使事务中的所有数据库操作执行都成功，才算整个事务成功。如果任何一个SQL语句执行失败，那么已经执行成功的SQL语句也必须撤销，数据库状态退回到执行事务前的状态。
- 一致性（C）：
- 指事务将数据库从一种状态转变为下一种一致的状态。在事务开始之前和事务结束之后，数据库的完整性约束没有被破坏。

事务

- 隔离性（I）
- 一个事务的影响在该事务提交前对其他事务都不可见，这通过锁（lock）来实现。
- 持久性（D）
- 事务一旦提交，其结果就是永久性的。即使发生宕机等故障，数据库也能将数据恢复。

事务

Automatic & Duration：

- redo log （顺序读写，日志先行）
- LSN

Consistent：一致性

- Undo log
- MVCC

事务的实现

- 在InnoDB存储引擎中，事务日志通过重做（redo）日志文件和InnoDB存储引擎的日志缓冲（InnoDB log buffer)来实现。当开始一个事务时，会记录该事务的一个LSN（日志序列号），当事务执行时，会往日志缓冲里插入事务日志；当事务提交时，必须将日志缓冲写入磁盘（需要参数inodb_flush_log_at_trx_commit=1），也就是在写数据前，需要先写日志。
- InnoDB通过这种方式保证事务的完整性。

事务的实现

- 重做日志记录了事务的行为，可以很好的通过其进行“重做”。但是事务有时还需要撤销，这时就需要undo。Undo与redo相反，对于数据库进行修改时，数据库不但会产生redo，而且还会产生一定量的undo。这些undo信息可以用来将数据回滚到修改之前的样子。Undo存放在数据库内部的共享表空间中。

事务的实现

- MVCC(Multiversion Concurrency control)
- 多版本并发控制技术
- 它是通过及时保存在某些时刻的数据快照得以实现的。InnoDB通过为每个数据行增加2个隐含值的方式实现。这2个隐含值记录了行的创建时间和删除时间。每一行都存储了事件发生时的系统版本号。每一次开始一个新事物，版本号都会自动递增。每个查询都会根据事物的版本号，检查每行数据的版本号。

事务的实现（mvcc）

- 在repeatable read隔离级别下，mvcc的实际操作：
- Select
- 1. 只查找版本早于当前事物的行。这确保了当前事物读取的行都是在事务开始前已经存在的，或者是由当前事物创建或修改的。
- 2. 数据的删除版本未定义或大于事务版本。这确保了事务读取的行，在事务开始时是未被删除的。
- Update
- InnoDB为每一个更新的行建立一个新的行拷贝，并且为新的行拷贝记录当前的系统版本号。同时也为更新前的旧行记录系统的版本号作为其删除版本标示。
- Insert
- 为每个新增行记录当前系统版本号
- Delete
- 为每个删除行记录当前系统版本号，作为行删除标示。

事物的实现（mvcc）

Isolation:

四种隔离级别

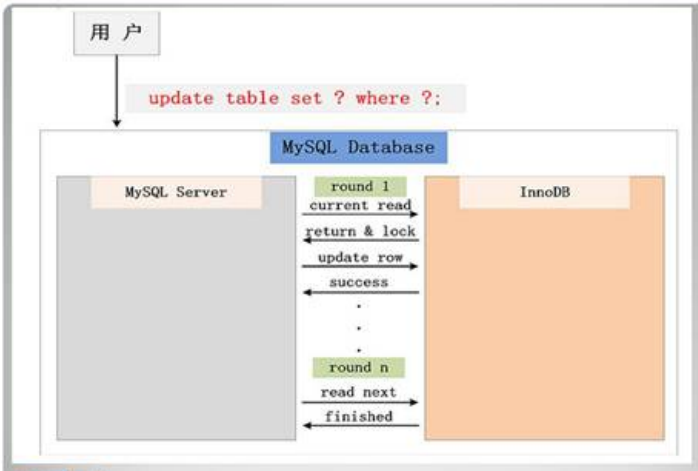
- Read uncommitted
- Read committed
- Repeateable read
- serializable

锁（lock）

事务

- 背景知识：
  - 快照读：简单的select操作，属于快照读，不加锁
    - select * from table where ?;
  - 当前读：特殊的读操作，插入/更新/删除操作，属于当前读，需要加锁
    - select * from table where ? lock in share mode;
    - select * from table where ? for update;
    - insert into table values (…);
    - update table set ? where ?;
    - delete from table where ?;

案例



案例

- 从图中，可以看到，一个Update操作的具体流程。当Update SQL被发给MySQL后，MySQL Server会根据where条件，读取第一条满足条件的记录，然后InnoDB引擎会将第一条记录返回，并加锁 (current read)。待MySQL Server收到这条加锁的记录之后，会再发起一个Update请求，更新这条记录。一条记录操作完成，再读取下一条记录，直至没有满足条件的记录为止。因此，Update操作内部，就包含了一个当前读。同理，Delete操作也一样。Insert操作会稍微有些不同，简单来说，就是Insert操作可能会触发Unique Key的冲突检查，也会进行一个当前读。
- **注**：根据上图的交互，针对一条当前读的SQL语句，InnoDB与MySQL Server的交互，是一条一条进行的，因此，加锁也是一条一条进行的。先对一条满足条件的记录加锁，返回给MySQL Server，做一些DML操作；然后在读取下一条加锁，直至读取完毕。

案例说明

SQL1：
select * from t1 where id = 10;
- 不加锁

SQL2：
delete from t1 where id = 10;
- 对id=10的记录加锁？

案例

RR的隔离级别下：

case1：id是主键（pk）

case2：id是唯一键（uk）

case3：id是普通索引（key）

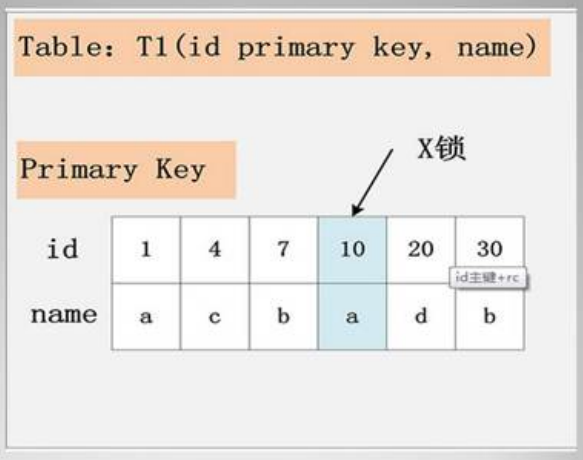
case4：id普通字段

案例

case 1:

Table：T1(id primary key, name)

Primary Key

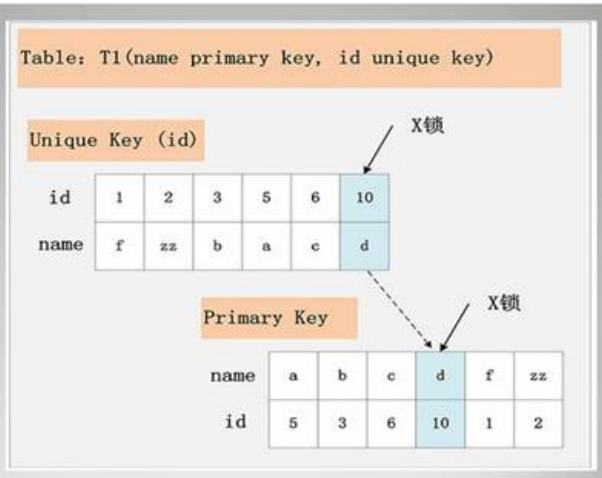


案例

- 这个组合，是最简单，最容易分析的组合。id是主键，Read Committed隔离级别，给定SQL：delete from t1 where id = 10; 只需要将主键上，id = 10的记录加上X锁即可。

Case1说明

case 2:



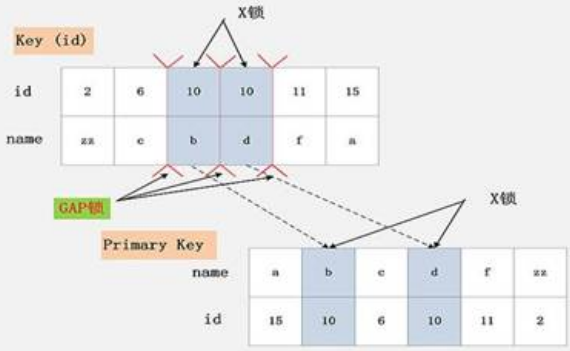
案例

- 此组合中，id是unique索引，而主键是name列。此时，加锁的情况同组合一不同了。由于id是unique索引，因此delete语句会选择走id列的索引进行where条件的过滤，在找到id=10的记录后，首先会将unique索引上的id=10索引记录加上X锁，同时，会根据读取到的name列，回主键索引(聚簇索引)，然后将聚簇索引上的name = 'd'对应的主键索引项加X锁。为什么聚簇索引上的记录也要加锁？试想一下，如果并发的一个SQL，是通过主键索引来更新：update t1 set id = 100 where name = 'a'; 此时，如果delete语句没有将主键索引上的记录加锁，那么并发的update就会感知不到delete语句的存在，违背了同一记录上的更新/删除需要串行执行的约束。
- **结论**：若id列是unique列，其上有unique索引。那么SQL需要加两个X锁，一个对应于id unique索引上的id = 10的记录，另一把锁对应于聚簇索引上的[name='d',id=10]的记录。

Case2说明

case 3:

Table: T1(name primary key, id key)



案例

- 根据此图，可以看到，首先，id列索引上，满足id = 10查询条件的记录，均已加锁。同时，这些记录对应的主键索引上的记录也都加上了锁。与组合二唯一索引的区别在于，组合二最多只有一个满足等值查询的记录，而组合三会将所有满足查询条件的记录都加锁。
- **结论**：若id列上有非唯一索引，那么对应的所有满足SQL查询条件的记录，都会被加锁。同时，这些记录在主键索引上的记录，也会被加锁。

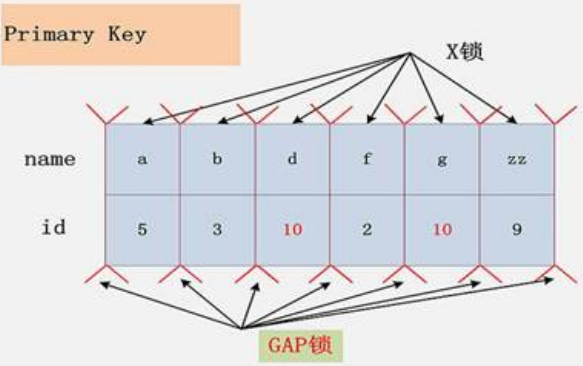
Case3说明

- 相对于前面三个组合，这是一个比较特殊的情况。id列上没有索引，where id = 10;这个过滤条件，没法通过索引进行过滤，那么只能走全表扫描做过滤。对应于这个组合，SQL会加什么锁？或者是换句话说，全表扫描时，会加什么锁？这个答案也有很多：有人说会在表上加X锁；有人说会将聚簇索引上，选择出来的id = 10;的记录加上X锁。那么实际情况呢？请看下图：

case4

case 4:

Table: T1(name primary key, id)



案例

- **结论：**若id列上没有索引，SQL会走聚簇索引的全扫描进行过滤，由于过滤是由MySQL Server层面进行的。因此每条记录，无论是否满足条件，都会被加上X锁。

Case4说明

SQL：

```
DELETE
FROM t1
WHERE pubtime > 1
   AND puptime < 20
   AND userid = 'hbc'
   AND COMMENT IS NOT null
```

一条复杂的SQL

表分布

Table: t1(id primary key, userid, blogid, pubtime, comment)
Index: idx_t1_pu(puptime,userid)

idx_t1_pu

pubtime	1	3	5	10	20	100
userid	hdc	yyy	hdc	hdc	bbb	hdc
id	10	4	8	1	100	6

Primary Key

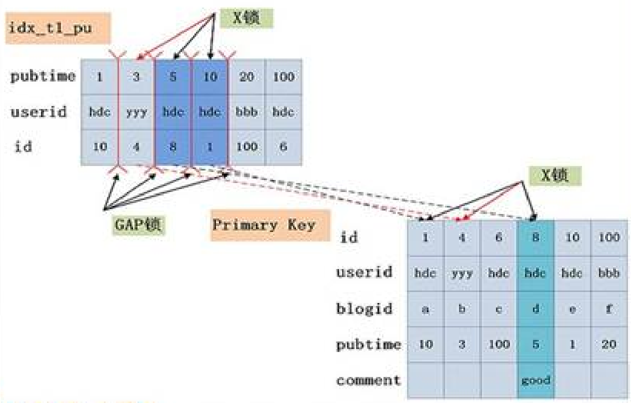
id	1	4	6	8	10	100
userid	hdc	yyy	hdc	hdc	hdc	bbb
blogid	a	b	c	d	e	f
pubtime	10	3	100	5	1	20
comment				good		

一条复杂的SQL

- **Index key：** pubtime > 1 and puptime < 20。此条件，用于确定SQL在idx_t1_pu索引上的查询范围。
- **Index Filter：** userid = 'hdc' 。此条件，可以在idx_t1_pu索引上进行过滤，但不属于Index Key。
- **Table Filter：** comment is not NULL。此条件，在idx_t1_pu索引上无法过滤，只能在聚簇索引上过滤。

锁分布：

Table: t1(id primary key, userid, blogid, pubtime, comment)
Index: idx_t1_pu(puptime,userid)



一条复杂的SQL

- 从图中可以看出，在RR隔离级别下，由Index Key所确定的范围，被加上了GAP锁；Index Filter锁给定的条件(userid = 'hdc')何时过滤，视MySQL的版本而定，在MySQL 5.6版本之前，不支持Index Condition Pushdown(ICP)，因此Index Filter在MySQL Server层过滤，在5.6后支持了Index Condition Pushdown，则在index上过滤。若不支持ICP，不满足Index Filter的记录，也需要加上记录X锁，若支持ICP，则不满足Index Filter的记录，无需加记录X锁 (图中，用红色箭头标出的X锁，是否要加，视是否支持ICP而定)；而Table Filter对应的过滤条件，则在聚簇索引中读取后，在MySQL Server层面过滤，因此聚簇索引上也需要X锁。最后，选取出了一条满足条件的记录[8,hdc,d,5,good]，但是加锁的数量，要远远大于满足条件的记录数量。

加锁分析

- **结论：**在Repeatable Read隔离级别下，针对一个复杂的SQL，首先需要提取其where条件。Index Key确定的范围，需要加上GAP锁；Index Filter过滤条件，视MySQL版本是否支持ICP，若支持ICP，则不满足Index Filter的记录，不加X锁，否则需要X锁；Table Filter过滤条件，无论是否满足，都需要加X锁。

加锁分析结论

学习加锁分析，作用有二：
To 研发和测试：可以根据MySQL的加锁规则，写出或测出不会发生死锁的SQL；

To dba：可以根据MySQL的加锁规则，定位出线上产生死锁的原因；

死锁

主要原因：

- ✓系统资源不足
- ✓进程推进顺序不合适
- ✓资源分配不当等

死锁必要条件：

- 互斥条件　　（锁排斥）
- 请求与保持
- 不可剥夺
- 循环等待

MySQL直接回滚已认为回滚成本小的事务

死锁

经典死锁模型：

Table：T1(id primary key, name)

```
session 1                                    session 2
begin;                                       begin;
select * from t1 where id = 1 for update;
                                             delete from t1 where id = 5;
update t1 set name=' qqq'  where id = 5;
                                             delete from t1 where id = 1;
```

Lock id=1　wait id=5　Lock id=5　wait id=1

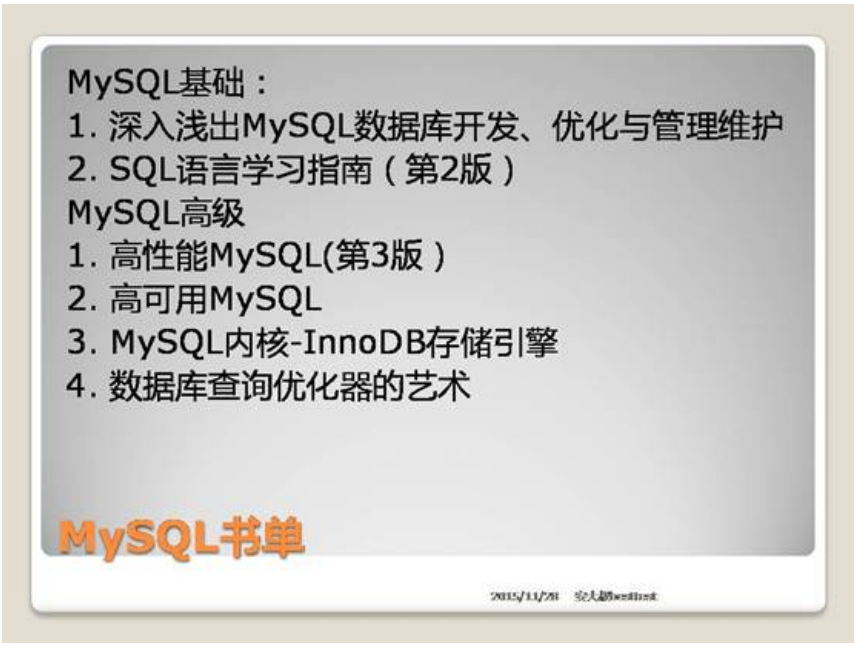
死锁发生！！！

id	1	2	3	4	5	6
name	aaa	ccc	aaa	bbb	ccc	zzz

小结：

- 多线程并发才有可能死锁
- 避免交叉加锁
- 减少涉及的表，表联接会大大增加锁范围
- 避免全表更新
- 控制更新行数
- where优先使用区分度高的索引

死锁


MySQL基础：
1. 深入浅出MySQL数据库开发、优化与管理维护
2. SQL语言学习指南（第2版）
MySQL高级
1. 高性能MySQL(第3版）
2. 高可用MySQL
3. MySQL内核-InnoDB存储引擎
4. 数据库查询优化器的艺术
MySQL书单