辛星笔记之MySQL优化篇

----------于2014年12月----------

**********说明**********

1.百度搜索“辛星 笔记”可以找到更多更全更优秀的资料奥。

2.辛星笔记，进击的捷径，致胜的法宝。

**********致敬原著**********

1.本书原名为《深入浅出 MySQL：数据库开发、优化与管理维护》，出版时间为2010年，是网易技术部的唐汉明、翟振兴、兰丽华、关宝军、申宝柱编著的。

2.祝愿各位作者工作顺利，事业顺心，万事如意。

**********辛星笔记**********

去除废话，提炼精华。　　辛星笔记，恪守此道。

传播知识，传递温情。　　我心永恒，始终如一。

**********目录**********

**********与君共勉**********

因寂寞而优秀　　因努力而充实

****

第一节：SQL 技巧

**********正则表达式的使用**********

1.正则表达式(Regular Expression)，是指一个用来描述或者匹配一系列复合某个句法规则的字符串的单个字符串。

2.很多文本编辑器或者其他工具里，正则表达式通常被用来检索或替换复合某个模式的文本内容。

3.正则表达式这个概念最初是由 Unix 中的工具软件普及开的，通常缩写为 regex 或者 regexp。

4.MySQL 利用 regexp 命令提供给用户扩展的正则表达式功能，regexp 实现的功能，类似 Unix 上 grep 和 sed 的功能，并且 regexp 在进行模式匹配的时候是区分大小写的。

**********正则表达式中的模式**********

1.^ 表示在字符串的开始处进行匹配。

2.$表示在字符串的末尾处进行匹配。

3..表示匹配任意单个字符，包括换行符。

4.[...]表示匹配出括号内的任意字符

5.[^...]表示 匹配不出括号内的任意字符。

6.a*表示匹配零个或者多个 a(包括空格)

7.a+表示匹配一个或者多个 a(不包括空格)

8.a?表示匹配 1 个或者零个 a

9.a1|a2 表示匹配 a1 或者 a2

10.a(m)表示匹配 m 个 a。

11.a(m,)表示匹配 m 个 a 或者更多个 a。

12.a(m,n)表示匹配 m 到 n 个 a。

13.a(,n)表示匹配 0 到 n 个 a。

14.(....)将模式元素组成单一元素。

**********操作范例**********

1.其中^在字符串的开始表示匹配，返回 1 表示匹配，返回 0 表示不匹配，下面检测 abcdefg 是否以 a 开头，范例：

```
select 'abcdefg' regexp '^a';
```

2.而[...]匹配括号内出现的任意字符，范例：

```
select 'efg' regexp '[ef]';
```

3.提取以'@163.com'结尾的范例：

```
select name,email from t where email regexp '@163[.,]com$';
```

4.当然上例中使用 email like '@163%.com' or email like '@163%.com'也可以。

**********巧用 rand()提取随机行**********

1.大多数数据库都会提供产生随机数的包或函数，通过这些包或者函数可以产生用户需要的随机数，也可以用来从数据表中抽取随机产生的记录，这对一些抽样统计是很有用的。

2.在 MySQL 中，产生随机数的方法是 rand()函数。

3.我们可以利用这个函数与 order by 子句一起完成随机抽取某些行的功能。

4.它的原理其实就是 order by rand()能够把数据随机排序。

5.比如随机抽取一部分样本：

```
select * from t1 order by rand() limit 5;
```

**********利用 group by 的 with rollup 子句做统计**********

1.在 SQL 语句中，使用 group by 的 with rollup 子句可以检索出更多的分组聚合信息。

2.它不仅仅能像一般的 group by 语句那样检索出各组的聚合信息，还能检索出本组类的整体聚合信息。

3.其实 with rollup 反应的是一种 olap 思想，也就是说这一个 group by 语句执行完后可以得到任何一个分组的聚合信息值。

4.当使用 rollup 时，不能使用 order by 子句进行结果排序，也就是说，rollup 和 order by 是互斥的。

5.另外，limit 用来 rollup 后面。

6.范例操作：

select year，country，product，sum(profit) from sales group by year,country,product with rollup。

**********用 bit group functions 做统计**********

1.我们可以使用 group by 语句和 bit_and、bit_or 函数完成统计工作。

2.这两个函数的一般用途就是做数值之间的逻辑位运算。

3.当把它们与 group by 子句联合使用的时候可以做一些其他的任务。

4.操作范例：

select customer_id,bit_or(kind) from order_rab group by customer_id;

5.由于它是通过把数据转化为二进制做位运算的，但是现实的时候还是 10 进制。

6.这种方法能够大大节省存储空间，而且能够提高部分统计计算的速度。

**********数据库名、表名大小写问题**********

1.在 mysql 中，数据库对应操作系统下的数据目录。

2.数据库中的每个表至少对应数据库目录中的一个文件,也可能是多个，这取决于引擎。

3.使用操作系统的大小写敏感性决定了数据库名和表名的大小写敏感性。

4.在大多数 Unix 操作系统，由于操作系统对大小写的敏感性导致了数据库命名和表名对大小写的敏感性。

5.在 win 中，由于操作系统本身对大小写不敏感，以你 win 下 mysql 对数据库名和表名对大小写也不敏感。

6.列、索引、存储子程序和触发器在任何平台上对大小写不敏感。

7.默认情况下，表别名在 Unix 中对大小写敏感，但是在 windows 或者 mac os x 中对大小写不敏感。

**********lower_case_tables_names**********

1.在 mysql 中如何在硬盘上保存、使用表名和数据库名由 lower_case_tables_names 系统变量决定。

2.它可以取值为 0 或者 1 或者 2。

3.取值为 0 表示使用 create table 或者 create database 语句指定的大写和小写在硬盘上保存表名和数据库名，名称对大小写敏感，在 Unix 系统中的默认设置就是这个值。

4.取值为 1 表示表名在硬盘上以小写保存，名称对大小写敏感，mysql 将所有表名转换为小写以便存储和查找，该值为 win 和 mac 系统中的默认值。

5.取值为 2 表示表名和数据库名在硬盘上使用 create table 或 create database 语句指定的大小写进行保存，但 mysql 将它们转换为小写以便查找。此值只在对大小写不敏感的文件系统上使用。

6.要查看它，可以使用：

```
show variables like '%lower%';
```

**********使用外键需要注意的问题**********

1.在 mysql 中，InnoDB 存储引擎支持对外部键字约束条件的查询。

2.对于其他类型存储引擎的表，当使用 references t1(c1)子句定义列是可以使用外部关键字，但是该子句没有实际的效果，只作为备忘录或者注释来提醒用户目前正定义的列指向另一个表中的一个列。

3.下面是 InnoDB 引擎的范例：

```
>create table t1(id int,name varchar(10),primary key(id))
engine=innodb;
>create table t2(
->id int,
```

->bookname varchar(10),

->userid int,

->primary key(id),

->constraint fk_userid_id foreign key(userid) references t1(id))

->engine = innodb;

4.而且当我们用 show create table 命令查看建表语句的时候，发现 MyISAM 存储引擎的并不显示外键的语句，而 InnoDB 存储引擎的就显示外键语句。

**********小结**********

1.应用正则表达式可以使用户能够在一段很大的字符串中找到符合自己规则的一些语句，这个功能在进行匹配查找的时候非常有用。

2.rand()函数与 order by 子句的配合能够实现随机抽取样本的功能，这个期缴在进行数据统计的时候很方便。

3.group by 的 with rollup 子句能够实现类似于 olap 的查询。

4.比特函数与 group by 子句的联合使用在某些应用场合可以大大降低存储量，提高统计查询效率。

5.mysql 数据库名和表名的大小写与操作系统对大小写的敏感性关系密切，因此需要格外引起用户的注意。

6.mysql 的外键功能仅仅对 InnoDB 存储引擎的表有作用，其他类型存储引擎的表虽然可以建立外键，但是并不能起到外键的作用。

第二节：SQL 优化

**********前言**********

1.在应用的开发过程中，由于初期数据量小，开发人员写 SQL 语句更重视功能上的实现。

2.当应用系统上线之后，随着生产数据量的急剧增长，很多 SQL 语句就开始逐渐显露出性能问题，对生产的影响也越来越大。

3.此时有些 SQL 语句就成为整个系统性能的瓶颈，因此我们必须对其进行优化。

**********优化 SQL 语句的一般步骤**********

1.通过 show status 命令了解各种 SQL 的执行频率

2.定位执行执行效率较低的 SQL 语句

3.通过 explain 分析低效 SQL 的执行计划

4.确定问题并采取相应的优化措施

**********了解 SQL 执行频率**********

1.我们可以通过 show [session|global] status 命令可以提供服务器状态信息。

2.也可以在操作系统上使用 mysqladmin extended-status 命令获取这些信息。

3.show status 如果使用 session 修饰表示当前连接，使用 global 修饰表示自数据库上次启动至今的统计结果，如果不写，默认是 session。

4.操作范例：

```
show status like 'Com_%';
```

5.其中 Com_xxx 表示每个 xxx 语句执行的次数，我们比较关心的是下面几个统计参数，它们对所有存储引擎的表操作都会进行累计：

(1)Com_select：表示执行 select 操作的次数，一次查询累加 1.

(2)Com_insert：执行 insert 操作的次数，对于批量插入累加 1.

(3) Com_update:执行 update 操作的次数

(4)Com_delete:执行 delete 操作的次数。

6.对于 InnoDB 存储引擎来说，我们关注下面几个数据：

(1)Innodb_rows_read:select 查询返回的行数

(2)Innodb_rows_inserted:执行 insert 操作插入的行数

(3)Innodb_rows_updated:执行 update 操作更新的行数

(4)Innodb_rows_deleted:执行 delete 操作删除的行数

7.通过上面这些参数，我们很容易的了解到当前数据库的应用是以插入更新为主还是以查询操作为主，以及各种类型的 SQL 大致的执行比例是多少。

8.对于更新操作的计数，是对执行次数的计数，不论提交还是回滚都会进行增加。

9.对于事务型的应用，通过 Com_commit 和 Com_rollback 可以了解事务提交和回滚的情况，对于回滚操作非常频繁的数据库，可能意味着应用编写存在问题。

10.我们可以查询如下几个参数来了解数据库的基本情况：

(1)Connections :试图连接 MySQL 服务器的次数

(2)Uptime:服务器工作时间

(3)Slow_queries:慢查询的次数

**********定位执行效率较低的 SQL 语句**********

1.可以通过慢查询日志的方式来定位哪些执行效率较低的 SQL 语句，用--log-slow-queries[=file_name]启动选项时，mysqld 写一个包含所有执行时间超过 long_query_time 秒的 SQL 语句的日志文件。

2.慢查询日志在查询结束之后才能记录，所以在应用反应执行效率出现问题的时候查询慢日志并不能定位问题。

3.可以使用 show processlist 命令查看当前 mysql 在进行的线程，包括线程的状态、是否锁表等等，可以实时地查看 SQL 的执行情况，同时对一些锁表操作进行优化。

**********分析低效 SQL 的执行计划**********

1.我们可以通过 explain 或者 desc 命令获取 MySQL 如何执行 select 语句的信息，包括在 select 语句执行过程中表如何连接和连接的顺序。

2.使用格式如下：

explain select 语句 \G

3.其中在结果中 select_type 表示 select 的类型，常见的取值有：

simpe----简单表，即不使用表连接或者子查询

primary---主查询，即外层的查询

union------union 中的第二个或者后面的查询语句

subquery--子查询中的第一个 select

4.在结果中的 type 表示表的连接类型，性能由好到差为：

system------表中仅有一行，即常量表

const-------单表中最多只有一个匹配行，比如 unique index

eq_ref------在表中只查询一条记录，多表连接使用唯一索引

ref----------使用普通索引

ref_or_null--条件中包含对 null 的查询

index_merge--索引合并优化

unique_subquery-in 后面是一个查询主键字段的子查询

index_subquery--in 的后面是查询非唯一索引字段的子查询

range----------单表查询中的范围查询

index----------通过查询索引来得到数据

all----------通过全表扫描来得到数据

5.在结果中 possible_keys 表示查询时，可能使用的索引。

6.在结果中 key 表示实际使用的索引。

7.在结果中的 key_len 表示索引字段的长度。

8.在结果中的 rows 表示扫描行的数量。

9.在结果中的 extra 表示执行情况的说明和描述。

**********索引的存储分类**********

1.MyISAM 存储引擎的表的数据和索引是自动分开存储的，各自是独立的一个文件，InnoDB 存储引擎的表的数据和索引是存储在同一个表空间中，但是可以有多个文件组成。

2.MySQL 中索引的存储类型目前有两种，即 BTREE 和 HASH，具体和表的存储引擎有关。

3.MyISAM 和 InnoDB 存储引擎都只支持 btree 索引，memory/heap 存储引擎都可以支持 hash 和 btree 索引。

4.MySQL 目前不支持函数索引，但是能对列的前面某一部分索引，比如 name 字段，可以只取 name 的前 4 个字符进行索引，这个特性可以大大缩小索引文件的大小，用户在设计表结构的时候也可以对文本列根据此特性进行灵活设计。

5.创建前缀索引的范例：

```
create index ind_t1 on t1(name(4));
```

**********索引说明**********

1.索引用于快速找出在某个列中有一特定值的行，对相关列使用索引是提高 select 操作性能的最佳途径。

2.查询要使用索引最主要的条件是查询条件中需要使用索引关键字。

3.如果是多列索引，那么只有查询条件使用了多列关键字最左边的前缀时，才可以使用索引，否则将不能使用索引。

**********使用索引的情况**********

1.对于创建的多列索引，只要查询的条件中用到了最左边的列，索引一般就会被使用。

2.对于使用 like 的查询，后面如果是常量并且只有%号不在第一个字符，索引才可能会被使用。

3.如果 like 后面跟的是一个列的名字，那么索引也不会被使用。

4.如果对大量的文本进行索引，使用全文索引而不是使用 like '%...%';

5.如果列名是索引，使用 c1 is null 将使用索引。

**********存在索引但不使用索引的情况**********

1.如果 mysql 估计使用索引比全表扫描更慢，则不使用索引。

2.如果使用 memory/heap 表并且 where 条件中不使用"="进行索引列，那么不会用到索引。

3.heap 表只有在"="的条件下才会使用索引。

4.用 or 分隔开的条件，如果 or 前的条件中的列有索引，而后面的列中没有索引，那么涉及的索引都不会被用到。

5.如果不是索引列的第一部分，这里指的是复合索引。

6.如果 like 以%开始。

7.如果列类型是字符串，那么一定记得在 where 条件中把字符串常量值用引号引起来，否则的话即使这个列上有索引，mysql 也不会用。因为 mysql 默认把输入的常量值进行转换后才进行检索。比如：

```
explain select *from t1 where name=333\G
```

**********查看索引使用情况**********

1.如果索引正在工作，那么 Handler_read_key 的值将会很高。

2.这个值代表了一个行被索引值读的次数。

3.很低的值表明增加索引得到的性能改善不高，因为索引并不经常使用。

4.Handler_read_rnd_next 的值高意味着查询运行低效，并且应该建立索引进行补救。

5.这个值的含义是在数据文件中读下一行的请求数。

6.如果正在进行大量的表扫描，Handler_read_rnd_next 的值较高，则通常说明索引不正确或者写入的查询没有利用索引。

7.查询它们的使用为：

show status like 'Handler_read%';

********两个简单有效的优化方法**********

1.对于大多数开发人员来说，可能只希望掌握一些简单实用的优化方法。对于更多更复杂的优化，更倾向于交给专业的 DBA 去做。

2.第一个优化方法就是定期分析表和检查表。

3.第二个优化方法就是定期优化表。

**********分析表**********

1.分析表的语法如下：

analyze [local|no_write_to_binlog] table 表名 1 [,表名 2].....

2.本语句用于分析和存储表的关键字分布，分析的结果将可以使得系统得到准确的统计信息，使得 SQL 能够生成正确的执行计划。

3.如果用户感觉实际执行计划并不是预期的执行计划，执行一次分析表可能会解决问题。

4.在分析期间，使用一个读取锁定对表进行锁定。这对于 MyISAM, BDB 和 InnoDB 表有作用。

5.对于 MyISAM 表，本语句与使用 myisamchk -a 相当。

6.对表进行分析的范例：

analyze table t1;

**********检查表**********

1.检查表的语法如下：

check table 表名 1 [,表名 2].... [option].... option={QUICK | FAST |MEDIUM | EXTENDED |CHANGED}

2.检查表的作用是检查一个或多个表是否有错误。

3.check table 对 MyISAM 和 InnoDB 表有作用。

4.对 MyISAM 表，关键字数据数据被更新，就可以 check table。

5.check table 也可以检查视图是否有错误，比如在视图定义中被引用的表已经不存在，范例如下：

(1)首先我们创建一个视图。

```
create view v1 as select * from t1;
```

(2)然后 check 一下该视图，发现没有问题：

```
check table v1;
```

(3)现在删除掉该视图依赖的表：

```
drop table t1;
```

(4)再来 check 一下刚才的视图，发现报错了：

```
check table v1;
```

**********优化表**********

1.优化表的语法如下：

```
optimize [local | no_write_to_binlog] table 表名 1 [,表名 2]....
```

2.如果已经删除了表的一大部分，或者如果已经对含有可变长度行的表(含有 varchar，blob 或者 text 列的表)进行了很多更改，则应该使用 optimize table 命令来进行表优化。

3.这个命令可以将表中的空间碎片进行合并，并且可以消除由于删除或者更新造成的空间浪费。

4.但是 optimize table 命令只对 MyISAM、BDB 和 InnoDB 表起作用。

5.操作范例：

```
optimize table t1;
```

6.注意：analyze、check、optimize 执行期间将会对表进行锁定，因此一定要注意在数据库不繁忙的时候执行相关的操作。

**********常用 SQL 的优化**********

1.大批量插入数据

2.优化 insert 语句

3.优化 group by 语句

4.优化 order by 语句

5.优化嵌套查询

6.优化 or 条件

7.使用 SQL 提示

**********提高 MyISAM 表的导入效率**********

1.当我们使用 load 命令导入数据的时候，适当的设置可以提高导入的速度。

2.对于 MyISAM 存储引擎的表，可以通过下面的方式来快速的导入大量的数据：

```
alter table t1 disable keys;
  loading the data
alter table t1 enable keys;
```

3.其中 disable keys 和 enable keys 用来打开或者关闭 MyISAM 表非唯一索引的更新。

4.在导入大量的数据到一个非空的 MyISAM 表的时候，通过设置这两个命令，可以提高导入的效率。

5.对于导入大量数据到一个空的 MyISAM 表，默认就是先导入数据然后才创建索引的，所以不用进行设置。

6.上面是对 MyISAM 表进行数据导入时的优化措施，对于 InnoDB 类型的表，这种方式并不能提高导入数据的效率。

**********提高 InnoDB 表的导入效率**********

1.因为 InnoDB 类型的表是按照主键的顺序保存的，所以将导入的数据按照主键的顺序排列，可以有效的提高导入数据的效率。

2.在导入数据前执行 set unique_checks =0 关闭唯一性校验，在导入结束后执行 set unique_checks=1 来恢复唯一性校验，提高效率。

3.如果应用使用自动提交的方式，建议在导入前执行 set autocommit=0 来关闭自动提交，导入结束后再执行 set autocommit=1 来打开自动提交，它也可以提高导入的效率。

**********优化 insert 语句**********

1.当进行数据 insert 的时候，可以考虑采用以下几种优化方式。

2.如果同时从同一客户插入很多行，尽量使用多个值表的 insert 语句，这种方式将大大缩减客户端与数据库之间的连接、关闭等消耗，使得效率比分开执行的单个 insert 语句块。

3.如果从不同客户插入很多行，能通过使用 insert delayed 语句得到更高的速度，delayed 的含义是让 insert 语句马上执行，其实数据都被存放在内存的队列中，并没有真正写入磁盘，这比每条数据分别插入要快得多。

4.而 low_priority 刚好相反，在所有其他用户对象的读写完后才进行插入。

5.将索引文件和数据文件分在不同的磁盘上存放,此处需要使用建表中的选项。

6.如果进行批量插入，可以增加 bulk_insert_buffer_size 变量值的方式来提高速度，但是，这只能对 MyISAM 表使用。

7.当从一个文本文件装载一个表时，使用 load data infile，这通常比使用很多 insert 语句快 20 倍。

**********优化 group by 语句**********

1.默认情况下，MySQL 对所有 group by c1，c2....的字段进行排序，这与在查询中指定 order by c1，c2 类似。

2.因此，如果显式包含相同的列的 order by 子句，则对 MySQL 的实际执行性能没有什么影响。

3.如果查询包括 group by，但是用户想要避免排序结果的消耗，可以指定 order by null 来禁止排序。

4.比如：

```
explain select id，sum(money) from t1 group by id\G
```

5.在 4 被执行的时候，发现有一个 filesort 的过程。

6.禁止排序范例：

explain select id，sum(tt) from t1 group by id order by null

**********优化 order by 语句**********

1.在某些情况中，MySQL 可以使用一个索引来满足 order by 子句，而不需要额外的排序。

2.where 条件和 order by 使用相同的索引，并且 order by 的顺序和索引顺序相同，并且 order by 的字段都是升序或者都是降序。

3.比如下列 SQL 就可以使用索引：

select * from t1 order by key_part1 ,key_part2....;

select * from t1 where key_part = 1 order by key_part1 desc, key_part2 desc;

select * from t1 order by key_part1 desc,key_part2 desc;

4.但是在下面几种情况下则不使用索引：

(1)order by 的字段混合 asc 和 desc：

select * from t1 order by key_part1 desc，key_part2 asc；

(2)用于查询行的关键字与 order by 中所使用的不同：

select * from t1 where key2= constant order by key1；

(3)对不同的关键字使用 order by：

select * from t1 order by key1，key2；

**********优化嵌套子查询**********

1.MySQL 4.1 开始支持 SQL 子查询，这个技术可以使用 select 语句来创建一个单列的查询结果，然后把这个结果作为过滤条件用在另一个查询中。

2.使用子查询可以一次性的完成很多逻辑上需要多个步骤才能完成的 SQL 操作，同时也可以避免失误或者表锁死，而且写起来很容易。

3.在很多情况下，子查询可以被更有效率的 join 替代。

4.比如在 t1 表中找到那些在 t2 表中不存在的所有公司信息：

select * from t1 where comany_id not in(select id from t2);

5.我们可以使用连接来优化它，尤其当 t2 表中对 id 建有索引的话，范例：

select *from t1 left join t2 on t1.compary_id = t2.id where t1.company_id is null;

6.连接之所以会更有效率，是因为 MySQL 不需要在内存中创建临时表来完成这个逻辑上需要两个步骤的查询工作。

**********优化 or 条件**********

1.对于含有 or 的查询子句，如果要利用索引，则 or 之间的每个条件列都必须用到索引，如果没有索引，则应该考虑增加索引。

2.MySQL 在处理含有 or 子句的时候，实际上是对 or 的各个字段分别查询后的结果进行了 union。

**********使用 SQL 提示**********

1.SQL 提示(SQL HINT)是优化数据库的一个重要手段，简单来说就是在 SQL 语句中加入一些人为的提示来达到优化操作的目的。

2.下面是一些常用的 SQL 提示：

use index 、ignore index、force index、

3.比如 select sql_buffer_results * from....这个语句强制 MySQL 生成一个临时结果集，只要临时结果集生成后，所有表上的锁定均被释放。

4.上述操作能在遇到表锁问题时或者要花很长时间将结果传给客户端时有所帮助，因为可以尽快释放锁资源。

**********use index**********

1.在查询语句中表名的后面，添加 use index 来提供希望 mysql 去参考的索引列表，就可以让 mysql 不再考虑其他可用的索引。

2.范例：

explain select *from t1 use index (ind_t1) where id = 3 \G

**********ignore index**********

1.如果用户只是单纯地想让 MySQL 忽略一个或者多个索引，则可以使用 ignore index 作为 hint。

2.范例：

explain select * from t1 ignore index (ind_t1) where id = 3 \G

**********force index**********

1.为 mysql 使用一个特定的索引，可以在查询中使用 force index 作为 hint。

2.比如，当不强制使用索引的时候，因为 id 的值都是大于 0 的，因此 MySQL 会默认进行全表扫描，而不使用索引。即：

explain select * from t1 where id > 0 \G

3.当使用 force index 进行提示时，即便使用索引的效率不是最高，MySQL 还是选择了使用索引，这是 MySQL 留给用户的一个自行选择执行计划的权利。即：

explain select * from t1 force index (id_xx) where id >0\G

**********小结**********

1.SQL 优化问题是数据库性能优化最基础也是最重要的一个问题。

2.很多数据库性能问题都是由不合适的 SQL 语句造成的。

第三节：数据库对象

**********一些问题**********

1.是否应该把所有表都按照第三范式来设计？

2.表里面的字段到底设置为多大长度合适？

3.很多问题虽然很小，但是如果设计不当可能会应用带来很多性能问题。

**********解决方案**********

1.优化表的数据类型

2.通过拆分提高表的访问效率

3.逆规范化

4.使用中间表来提高统计查询速度

**********优化表的数据类型**********

1.表需要使用何种数据类型，是需要根据应用来判断的。

2.虽然应用设计的时候需要考虑字段的长度留有一定的冗余，但是不推荐让很多字段都留有大量的冗余。这样既浪费磁盘存储空间，同时在应用程序操作时也浪费物理内存。

3.在 mysql 中，可以使用函数 procedure analyse()对当前应用的表进行分析，该函数可以对数据表中列的数据类型提出优化建议，用户可以根据应用的实际情况酌情考虑是否实施优化。

4.范例：

```
select * from t1 procedure analyse();

select * from t1 procedure analyse(16,256);
```

5.以上的第二个语句告诉 procedure analyse()不要为那些包含的值多于 16 个或者 256 字节的 enum 类型提出建议。如果没有这样的限制，输出信息可能会很长。

6.输出的每一列信息都会对数据表中的列的数据类型提出优化建议。

**********通过拆分提高表的访问效率**********

1.这里所说的“拆分”，是值对数据表进行拆分。

2.如果是针对 MyISAM 类型的表进行拆分，有两种方法：

(1)垂直拆分，即把主码和一些列放到一个表，然后把主码和另外的列放到另一个表中。

(2)水平拆分，即根据一列或者多列数据的值把数据行放到两个独立的表中。

3.水平拆分在以下几种情况下使用：

(1)表很大，分割后可以降低在查询时需要读的数据和索引的页数，同时也降低了索引的层数，提高查询速度。

(2)表中的数据本来就有独立性，比如，表中分别记录各个地区的数据或不同时期的数据，特别是有些数据常用，有些数据不常用。

(3)需要把数据存放到多个介质上。

4.水平拆分会给应用增加复杂度，它通常在查询时需要多个表名，查询所有数据需要 union 操作。

5.在许多数据库应用中，这种复杂性会超过它带来的优点，因为只要索引关键字不大，则在索引用于查询时，表中增加 2 至 3 倍的数据量，查询时也就增加读一个索引层的磁盘次数。

6.水平拆分要考虑数据量的增长速度，根据实际情况决定是否需要对表进行水平拆分。

**********逆规范化**********

1.数据库设计时要满足规范化，但是并不是数据的规范化程度越高就越好。

2.规范化越高，关系过多的直接结果就是导致表之间的连接操作越频繁，表之间的连接操作是性能较低的操作，直接影响到查询的速度。

3.对于查询较多的应用就需要根据实际情况运用规范化对数据进行设计，通过逆规范化来提高查询的性能。

4.反规范的好处是降低连接操作的需求、降低外码和索引的数目，还可能减少表的数目。

5.反规范化带来的问题是可能出现数据的完整性问题。它可以加快查询速度，但是对降低修改速度。

6.在进行反规范操作之前，要充分考虑数据的存取需求、常用表的大小、一些特殊的计算、数据的物理存储位置。

**********反规范技术**********

1.常用的反规范技术有：

增加冗余列、增加派生列、重新组表和分割表

2.所谓增加冗余列，就是在多个表中有相同的列，它常用来在查询时避免连接操作。

3.所谓增加派生列，就是增加的列来自其他表中的数据，由其他表中的数据经过计算生成。增加的派生列其作用就是在查询时减少连接操作，避免使用集函数。

4.所谓重新组表，就是如果许多用户需要查看两个表连接出来的结果数据，则把这两个表重新组成一个表来减少连接而提高性能。

5.分割表：也就是对表进行垂直和水平拆分。

**********注意点**********

1.逆规范化技术需要维护数据的完整性。

2.无论使用何种反规范化技术，都需要一定的管理来维护数据的完整性，常用的方法就是批处理维护、应用逻辑和触发器。

3.批处理维护是指对复制列或者派生列的修改积累一定的时间后，运行一批处理作业或存储过程或派生列进行修改，这只能在对实时性要求不高的情况下使用。

4.数据的完整性也可以有应用逻辑来实现，这就要求必须在同一事务中对所有涉及的表进行增、删、改操作。用应用逻辑来实现数据的完整性风险较大，因为同一逻辑必须在所有的应用中使用和维护，容易遗漏，特别是在需求变化时不易于维护。

5.另一种方式就是使用触发器，对数据的任何修改立即触发对复制或派生列的相应修改。触发器是实时的，而且相应的处理逻辑只在一个地方出现，易于维护。一般来说，是解决这类问题比较好的办法。

**********使用中间表提高统计查询速度**********

1.对于数据量较大的表，在其上进行统计查询通常会效率很低，并且还要考虑统计查询是否会对在线的应用产生负面影响。

2.在这种情况下，使用中间表可以提高统计查询的效率。

3.比如 session 表记录了客户每天的消费记录，那么当我们想分析最近一周客户的消费情况的时候，可以创建一个 tmp_session 表，它只拥有 session 表的部分数据，在该中间表上的操作会快很多。

4.中间表在统计查询中经常会用到，其优点如下：

(1)中间表复制源表部分数据，并且与源表相“隔离”，在中间表上做统计查询不会对在线应用产生负面影响。

(2)中间表上可以灵活的添加索引或增加临时用的新字段，从而达到提高统计查询效率和辅助统计查询的作用。

**********小结**********

1.数据库对象设计的好坏是一个数据库设计的基础。

2.一旦数据库对象设计完毕并投入使用，将来再进行修改就比较麻烦，因此在进行数据库设计的时候一定要尽可能地考虑周到。

第四节：锁问题

**********说明**********

1.锁是计算机协调多个进程或线程并发访问某一资源的机制。

2.在数据库中，除了传统的计算资源(比如 CPU、RAM、IO 等)的争用外，数据也是一种供许多用户共享的资源。

3.如何保证数据并发访问的一致性、有效性是所有数据库必须解决的一个问题。

4.锁冲突也是影响数据库并发访问性能的一个重要因素。

5.从这个角度来说，锁对数据而言尤其重要，也更加复杂。

**********mysql 锁概述**********

1.相比其他数据库而言，mysql 的锁机制比较简单。

2.它最显著的特点就是不同的存储引擎支持不同的锁机制。

3.MyISAM 和 memory 存储引擎采用的是表级锁(table -level locking)。

4.BDB 存储引擎采用的是页面锁(page-level locking),但是也支持表级锁。

5.InnoDB 存储引擎既支持行级锁(row-level locking),也支持表级锁，但是默认情况下使用行级锁。

6.表级锁：开销小，加锁块，不会出现死锁，锁定粒度大，发生锁冲突的概率最高，并发度最低。

7,.行级锁：开销大，加锁慢，会出现死锁，锁定粒度小，发送锁冲突的概率最低，并发度也最高。

8.页面锁：开销和加锁时间介于表锁和行锁中间，会出现死锁，锁定粒度介于表锁和行锁之间，并发度一般。

9.因此，很难说哪种锁更好，只能根据具体应用来判断哪种锁更合适。

10.仅仅锁的角度来说，表级锁更适合以查询为主，只有少量按索引条件更新数据的应用，比如 web 应用。

11.行级锁则更适合有大量按索引条件并发更新少量不同数据，同时又有并发查询的应用，比如一些在线事务处理系统(oltp).

12.由于 BDB 已经被 InnoDB 取代，成为历史，因此就不介绍了。

**********MyISAM 表锁**********

1.MyISAM 存储引擎只支持表锁，这也是 mysql 开始几个版本中唯一支持的锁类型。

2.随着应用对事务完整性和并发性要求的不断提高，mysql 才开始开发基于事务的存储引擎。后来慢慢出现了支持页所的 BDB 引擎和支持行级锁的 InnoDB 引擎(实际 InnoDB 是单独的一个公司，后被 oracle 收购).

3.但是 MyISAM 的表锁依然是使用最为广泛的锁类型。

4.可以通过检查 table_locks_waited 和 table_locks_immediate 状态变量来分析系统上的表锁定争夺，语句：

```
show status like 'table%';
```

5.如果 Table_locks_waited 的值比较高，说明存在着比较严重的表级锁争用情况。

**********表级锁的锁模式**********

1.mysql 的表级锁模式有两种：

(1)表共享读锁(Table Read Lock)

(2)表独占写锁(Table Write Lock)

2.对 MyISAM 表的读操作，不会阻塞其他用户对同一表的读请求，但是会阻塞对同一个表的写请求。

3.对 MyISAM 表的写操作，会阻塞其他用户对同一表的读和写操作。

4.MyISAM 表的读操作和写操作之间，以及写操作之间，是串行的。

5.当一个线程获得对一个表的写锁后，只有持有锁的线程可以对表进行更新，其他线程的读、写操作都会等待，直到锁被释放为止。

**********MyISAM 写阻塞范例**********

1.这里开启了两个 session，我们不妨简单记为 a 和 b。

2.在 a 中获得表 t1 的 write 锁定：

lock table t1 write；

3.在 a 中可以对锁定表的查询、更新、插入操作都可以执行，但是在 b 中对锁定表的查询会被阻塞，需要等待锁被释放。

4.在 a 中释放锁：

unlock tables；

5.在 b 中会获得资源，查询返回。

**********如何加表锁**********

1.MyISAM 在执行查询语句(select)前，会自动给涉及的所有表加读锁，在执行更新操作(update、delete、insert 等)前，会自动给涉及的表加写锁。

2.这个过程不需要用户干预，因此，用户一般不需要直接用 lock table 命令给 MyISAM 表显式加锁。

3.上述例子中的显式加锁是为了方便查看。

4.给 MyISAM 表显式加锁，一般是为了在一定程度上模拟事务操作，实现对某一个时间点内多个表的一致性读取。

5.在 lock table 的时候加了“local”选项，其作用就是在满足 MyISAM 表并发插入条件的情况下，允许其他用户在表尾并发插入记录。

6.使用 local 选项的范例：

lock tables t1 read local，t2 read local；

7.在用 lock tables 给表显式加锁时，必须同时取得所有涉及表的锁，并且 mysql 不支持表升级。也就是说，在执行 lock tables 之后，只能访问显式加锁的这些表，不能访问为加锁的表。

8.如果加的是读锁，那么只能执行查询操作，不能执行更新操作。在自动加锁的情况下也基本如此。

9.MyISAM 总是一次获得 SQL 语句所需要的全部锁。这也正是 MyISAM 不会出现死锁(Deadlock Free)的原因。

**********MyISAM 读阻塞范例**********

1.这里取得两个 session，分别记为 a 和 b。

2.在 a 中获得 t1 的 read 锁定：

lock table t1 read；

3.可以在 a 中查询该表记录，也可以在其他 session 中查询该表的记录。

4.但是在 a 中不能查询没有锁定的表，在 b 中可以查询或者更新未锁定的表。

5.在 a 中插入或者更新锁定的表会提示错误，在 b 中更新锁定表会等待获得锁。

6.在 a 中使用 unlock tables；来释放锁，在 b 中的更新操作完成。

**********操作说明**********

1.当使用 lock tables 时，不仅需要一次锁定用到的所有表，而且，同一个表在 SQL 语句中出现所少次，就要通过与 SQL 语句中相同的别名锁定多少次，否则也会出错。

2.对 actor 表获得读锁：

>lock table actor read;

3.但是通过别名访问会提示错误：

>select a.name from actor a ;

4.需要对别名分别锁定：

>lock table actor as a read,actor as b read;

5.之后的操作就正常了。

**********并发插入**********

1.由于 MyISAM 表的读和写是串行的，但是它是就总体而言。

2.在一定条件下，MyISAM 表也支持查询和插入操作的并发进行。

3.MyISAM 存储引擎有一个系统变量 concurrent_insert，专门用以控制其并发插入的行为，它的取值可以为 0、1、2。

4.当 concurrent_insert 为 0 时，不允许并发插入。

5.当 concurrent_insert 为 1 时，如果 MyISAM 表中没有空洞(即表的中间没有被删除的行),MyISAM 允许在一个进程中读表的同时，另一个进程从表尾插入记录。这也是 mysql 的默认设置。

6.当 concurrent_insert 为 2 时，无论 MyISAM 表中有没有空洞，都允许在表尾并发插入记录。

7.也就是说当某个 session 对表执行了 lock table t1 read；的时候，另一个表是可以执行 insert 的，但是 update 会被阻塞。

8.可以利用 MyISAM 存储引擎的并发插入特性，来解决应用中对同一表查询和插入的锁争用。

9.比如我们可以将 concurrent_insert 系统变量设置为 2，总是允许并发插入。同时，通过定期在系统空闲时间执行 optimize table 语句来整理空间碎片，收回因删除记录而产生的中间空洞。

**********MyISAM 的锁调度**********

1.MyISAM 存储引擎的读锁和写锁是互斥的，读写操作是串行的。

2.一个进程请求某个 MyISAM 表的读锁，同时另一个进程也请求同一表的写锁，MySQL 会如何处理呢?答案是写进程先获得锁。

3.即使是读请求先到锁等待队列，写请求后到，写锁也会被插入到读锁请求之前。

4.因为 MySQL 会认为写请求一般比读请求更重要。

5.这也正是 MyISAM 表不太适合有大量更新操作和查询操作应用的原因，因为，大量的更新操作会造成查询操作很难获得读锁，从而可能永远阻塞。

6.不过我们可以通过一些设置细节来调节 MyISAM 的调度行为，通常由下面几种方法：

(1)通过指定启动参数：low-priority-updates，使得 MyISAM 引擎默认给予读请求以有限的权利。

(2)通过执行命令 set low_priority_updates=1，使该连接发出的更新请求优先级降低。

(3)通过指定 insert、update、delete 语句的 low_priority 属性，降低改语句的优先级。

7.虽然上面三种方法都要么更新优先，要么查询优先的方法，但还是可以用其来解决查询相对重要的应用中，读锁等待严重的问题。

8.另外，mysql 也提供了一种折中的办法来调节读写冲突，即给系统参数 max_write_lock_count 设置一个合适的值，当一个表的读锁达到这个值后，mysql 就暂时将写请求的优先级降低，给读进程一定获得所的机会。

9.不过必须要避免长时间运行的查询操作，使得写进程“饿死”，因此，应该避免出现长时间运行的查询操作，不要总想用一条 select 语句解决问题，分解往往能够减少锁冲突。

**********InnoDB 锁问题**********

1.InnoDB 与 MyISAM 的最大不同在于两点：

(1)支持事务(transaction)

(2)采用行级锁。

2.行级锁与表级锁有很多的不同之处，而且事务的引入也带来了很多新问题。

**********事务**********

1.事务由一组 SQL 语句组成的逻辑处理单元，事务具有四个属性，也就是 acid 属性。

2.原子性(atomicity):事务是一个原子操作单元，其对数据的修改，要么全都执行，要么全都不执行。

3.一致性(onsistent):在事务开始和完成时，数据都必须保持一致状态。这意味着所有相关的数据规则都必须应用于事务的修改，以保

持数据的完整性，事务结束时，所有的内部数据结构(比如 B 树或双向链表)也都必须时正确的。

4.隔离性(isolation):数据库系统提供一定的隔离机制，保证事务在不受外部并发操作影响的“独立”环境执行。这意味着事务处理过程中的中间状态对外部是不可见的，反之亦然。

5.持久性(durable):事务完成后，它对于数据的修改是永久性的，即使出现系统故障也能够保持。

**********并发事务带来的问题**********

1.相对于串行处理来说，并发事务处理能大大增加数据库资源的利用率，提高数据库系统的事务吞吐量，从而可以支持更多的用户。

2.并发事务可能会出现的问题如下：

更新丢失、脏读、不可重复读、幻读

3.所谓更新丢失(lost update)，也就是当两个或多个事务选择同一行，然后基于最初选定的值更新该行时，由于每个事务都不知道其他事务的存在，就会发生丢失更新问题---最后的更新覆盖了由其他事务所做的更新。

4.脏读(dirty reads):一个事务正在对一条记录做修改，在这个事务完成并提交前，这条记录的数据就处于不一致状态。这时，另一个事务也来读取同一条记录，如果不加控制，第二个事务读取了这些“脏”数据，并据此做进一步的处理，就会产生未提交的数据依赖关系。这种现象被形象的叫做“脏读”。

5.不可重复读(non-repeatable reads):一个事务在读取某些数据后的某个时间，再次读取以前读过的数据，却发现其读出的数据已经发生了改变、或某些记录已经被删除了，这种现象叫做“不可重复读”。

6.幻读(phantom reads):一个事务按相同的查询条件重新读取以前检索过的数据，却发现其他事务插入了满足其查询条件的新数据，这种现象称为“幻读”。

7.这四种问题两种因为更新、一种因为删除、一种因为插入。

**********事务隔离级别**********

1.上面提到的并发事务处理带来的问题中，更新丢失通常是应该完全避免的。但防止更新丢失，并不能单靠数据库事务控制器来解决，需要应用程序对要更新的数据加必要的锁来解决，因此，防止更新丢失应该是应用的责任。

2.对于脏读、不可重复读、幻读，都是数据库读一致性问题，必须由数据库提供一定的事务隔离机制来解决。

3.数据库实现事务隔离的方式，基本可以分为两种：

(1)在读数据之前，对其加锁，防止其他事务对数据修改。

(2)不加任何锁，通过一定能够机制生成一个数据请求时间点的一致性数据快照(snapshot),并用这个快照来提供一定级别(语句级或事务级)的一致性读取。

从用户的角度来看，好像是数据库可以提供同一个数据的多个版本，因此，这种技术叫做数据多版本并发控制(MultiVersion Concurrency Control,简称 MVCC 或 MCC),也经常称为多版本数据库。

4.数据库的事务隔离越严格，并发副作用就越小，但付出的代价也就越大，因为事务隔离机制实质上是使事务在一定程度上“串行化”进行，这显然与“并发”是矛盾的。

5.因此，不同的应用对读一致性和事务隔离程度的要求也是不同的。比如许多应用对“不可重复读”和“幻读”并不敏感，可能更关心数据并发访问的能力。

6.为了解决“隔离”与“并发”的矛盾，ISO/ANSI SQL92 定义了 4 个事务隔离级别，每个级别的隔离程度不同，允许出现的副作用也不同，应用可以根据自己的业务逻辑要求，通过选择不同的隔离级来平衡“隔离”与“并发”的矛盾。

7.这四个隔离级为：

(1)未提交读(read uncommitted):

---------最低级别，只能保证不读取物理上损坏的数据。

---------可能出现：脏读、不可重复读、幻读

(2)以提交读(read committed):

------- 语句级

---------不会出现脏读、但是可能出现：不可重复读、幻读

(3)可重复读(repeated read)

---------事务级

---------不会出现脏读、不可重复读，可能幻读

(4)可序列化(serializable)

--------最高级别，事务级

---------不可能出现脏读、不可重复读、幻读

8.各具体数据库并不一定完全实现了上述 4 个隔离级别，比如 oracle 只提供 read committed 和 serializable 两个标准隔离级别，还提供自定义的 read only 级别。

9.SQL Server 除了支持上述定义的 4 个级别之外，还支持一个“快照”的隔离级别，但严格来说它是一个用 MVCC 实现的 Serializable 隔离级别。

10.MySQL 支持全部的四个隔离级别，但是在具体实现时，有一些特点，比如在一些隔离级别下面采用 MVCC 一致性读，有些情况下不是。

**********InnoDB 行锁争用情况**********

1.检查 Innodb_row_lock 状态变量来分析系统上的行锁的争夺情况：

```
show status like 'innodb_row_lock%';
```

2.如果发现锁争用比较严重，比如 Innodb_row_lock_waits 和 Innodb_row_lock_time_avg 的值比较高，还可以通过设置 InnoDB Monitors 来进一步观察发生锁冲突的表、数据行等等，并分析锁争用的原因。

3.具体操作：

```
>create table innodb_monitor(a int) engine = innodb;
```

>show innodb status\G

#当停止查看时，可以：

>drop table innodb_monitor;

4.在设置监视器后，在 show innodb status 的显示内容中，会有详细的当前锁等待的信息，包括表名、锁类型、锁定记录的情况等等，便于进一步的分析和问题的确定。

5.在打开监视器后，默认情况下每 15 秒会向日志中记录监控的内容，如果长时间打开会导致.err 文件变得跟大，所以用户在确认问题原因后，要记得删除监控表以关闭监视器，或者通过使用"--console"选项来启动服务器以关闭写日志文件。

**********InnoDB 锁分析**********

1.InnoDB 实现了以下两种类型的锁：

(1)共享锁(S):允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。

(2)排他锁(X):允许获得排他锁的事务更新数据，阻止其他事务取得相同数据集的共享读锁和排他写锁。

2.为了允许行锁和表锁共存，实现多粒度锁机制，InnoDB 还有两种内部使用的意向锁(Intention Locks)，这两种锁都是表锁。

3.意向共享锁(IS):事务打算给数据行加共享锁，事务在给一个数据行加共享锁前必须先取得该表的 IS 锁。

4.意向排他锁(IX):事务打算给数据行加排他锁，事务在给一个数据行加排他锁之前必须先取得该表的 IX 锁。

5.各个锁的兼容冲突情况如下：

(1) X 和所有锁都冲突，

(2) IX 兼容 IX 和 IS

(3) S 兼容 S 和 IS

(4) IS 兼容 IS IX 和 S。

6.如果一个事务请求的锁模式与当前的锁兼容，InnoDB 就将请求的锁授予该事务，如果两者不兼容，该事务就要等待锁释放。

7.意向锁是 InnoDB 自动加的，不用用户干预。

8.对于 update、delete、insert 语句，InnoDB 会自动给涉及数据集加排他锁(X)。

9.对于 select 语句，InnoDB 不会加任何所。

10.事务可以根据通过如下语句显示给记录集加共享锁或排他锁：

(1)共享锁(S):

select * from t1 where ... lock in share mode;

(2)排他锁(X):

select * from t1 where ... for update;

11.用 select..in share mode 获得共享锁，主要用在数据依存关系时来确认某行记录是否存在，并确认没有人对这个记录进行 update 或者 delete 操作。如果当前事务也需要对该记录进行更新操作，很可能会造成死锁，对于锁定记录后需要进行更新操作的应用，应该使用 select ....for update 方式获得排他锁。

**********实例操作 InnoDB 存储引擎共享锁**********

1.这里开启了两个 session，分别记为 a 和 b。

2.在 a 和 b 中分别设置 set autocommit =0 来关闭自动提交，此时是可以在两个事务中对同一行记录进行 select 操作的。

3.在 a 中对某条记录加共享锁：

select ..... lock in share mode;

4.此时在其他 session 仍然可以查询记录，也可以对该记录加 share mode 的共享锁。

5.在 a 中对锁定的记录进行更新操作，会等待锁。

6.在 b 中对该记录进行更新操作，则会死锁退出。

7.在 a 中获得锁后，更新成功。

**********InnoDB 排他锁示例**********

1.这里开启了两个 session，分别记为 a 和 b。

2.在 a 和 b 中都设置 set autocommit=0 来关闭自动提交，此时两个表中都可以检索同一条记录。

3.在 a 中对某条记录添加共享锁：

select ... for update;

4.在 b 中可以查询该记录，但是不能对该记录加共享锁，加锁的时候会等待。

5.在 a 中可以对锁定的记录进行更新操作，更新后释放锁(即 commit 操作)。

6.此时 b 中获得锁，也就是在第 4 步中的加排他锁会成功。

**********InnoDB 行锁实现方式**********

1.InnoDB 行锁是通过给索引项加锁来实现的，而 oracle 则是通过在数据块中对相应数据行加锁来实现的。

2.InnoDB 这种行锁意味着：

(1)只有通过索引条件检索数据，InnoDB 才使用行级锁。

(2)否则，InnoDB 使用表锁。

*********InnoDB 在不使用索引时使用表锁范例*********

1.这里使用两个 session，记为 a 和 b。

2.并且在 a 和 b 中设置 set autocommit=0 来关闭自动提交，注意这里的 t1 没有索引。

3.当我们在 a 中加共享锁：

select * from t1 where id = 1 for update;

4.此时在 b 中加共享锁会等待：

select * from t1 where id = 2 for update;

5.虽然在上例中我们对 t1 只加了一行排他锁，但是却锁定了整个表。

6.其原因就是因为没有索引的情况下，InnoDB 只能使用表锁。

**********使用相同的索引键会出现冲突**********

1.由于 MySQL 的行锁是针对索引加的锁，不是针对记录加的锁，所以虽然是访问不同行的记录，但是如果是使用相同的索引键，是会出现锁冲突的。

2.比如在开启了事务的两个 session 中，在 session1 操作如下：

```
select * from t1 where id = 1 and name='1' for update;
```

3.在 session2 中访问不同的记录：

```
select * from t1 where id = 1 and name ='4' for update;
```

4.虽然 session2 中访问的是不同的记录，但是因为使用了相同的索引，所以需要等待锁。

**********多个索引的阻塞情况**********

1.当表有多个索引的时候，不同的事务可以使用不同的索引锁定不同的行。

2.不论是使用主键索引、唯一索引或普通索引，InnoDB 都会使用行锁来对数据加锁。

3.比如在两个 session 中都开启事务，并且在 session1 中操作：

```
select * from t1 where id = 1 for update;
```

4.此时在 session2 中使用如下：

```
select * from t1 where name='4' for update;
```

5.由于此记录已被 session1 锁定，所以 session2 会被阻塞，来等待获得锁。

**********说明**********

1.即便在条件中使用了索引字段，但是否使用索引来检索数据是由 MySQL 通过判断不同执行计划的代价来决定的。

2.如果 MySQL 认为全表扫描效率更高，比如对一些很小的表，它就不会使用索引。

3.在不使用索引的情况下，InnoDB 就会使用表锁，而不是行锁。

4.因此，在分析锁冲突的时候，别忘记检查 SQL 的执行计划，以确认是否真正使用了索引。

**********间隙锁**********

1.当我们用范围条件而不是相等条件来检索数据，并且请求共享或排他锁时，InnoDB 会给符合条件的已有数据记录的索引项加锁。

2.对于键值在条件范围内但是并不存在的记录，叫做“间隙(GAP)”，InnoDB 也会对这个间隙加锁，这种锁机制就是所谓的间隙锁(Next-Key 锁)。

3.InnoDB 使用间隙锁的目的：

(1)为了防止幻读，以满足相关隔离级别的要求。

(2)为了满足其恢复和复制的需要

4.InnoDB 这种加锁机制会阻塞符合条件范围内键值的并发插入，这往往会造成严重的锁等待。

5.在实际开发中，尤其是并发插入比较多的应用，我们要尽量优化业务逻辑，尽量使用相等条件来访问更新数据，避免使用范围条件。

6.InnoDB 除了通过范围条件加锁时使用间隙锁，如果使用相等条件请求给一个不存在的记录加锁，也会使用间隙锁。

7.比如我们在两个开启了事务的 session 中，其中一个对不存在的记录加 for update 锁：

```
select * from t1 where empid=102 for update;
```

8.在另一个 session 中，我们插入 empid 为 202 的记录，也会出现锁等待。(这条记录并不存在)

9.只有在原 session 进行 rollback 之后，释放了 Next-Key 锁，其他 session 才可以获得锁并且成功插入记录。

******恢复和复制对 InnoDB 锁机制的影响**********

1.MySQL 通过 binlog 记录执行成功的 insert、update、delete 等更新数据的 SQL 语句，并由此实现 MySQL 数据库的恢复和主从复制。

2.复制其实就是在 slave mysql 不断做基于 binlog 的恢复。

3.MySQL 的恢复机制有如下特点：

(1)MySQL 的恢复是 SQL 语句级的，也就是重新执行 binlog 中的 SQL 语句，而 oracle 是基于数据库文件块的。

(2)MySQL 的 binlog 是按照事务提交的先后顺序记录的，恢复也是按这个顺序进行的。这点与 oracle 不同，oracle 是按照系统更新号(system change number，SCN)来恢复数据的，每个事务开始时，oracle 都会分配一个全局唯一的 SCN，SCN 的顺序与事务开始的时间顺序是一致的。

4.MySQL 的恢复机制要求：在一个事务提交前，其他并发事务不能插入满足其锁定条件的任何记录，也就是不允许出现幻读。实际上就是要求事务串行化。

5.这也是许多情况下，InnoDB 要用到间隙锁的原因，比如在用范围条件更新记录时，无论时在 read commited 还是 repeatable read 隔离级别下，InnoDB 都要使用间隙锁，这并不是隔离级别要求的。

**********CTAS**********

1.对于“insert into t1 select * from t2 where ...”和“create table t1 from t2...”这种 SQL 语句，用户并没有对 t2 做任何操作，但是 MySQL 对这种 SQL 语句做了特殊处理。

2.当我们在两个 session 中开启事务，在一个 session 中进行 CTAS 语句的时候，在另一个 session 是无法对 t2 表加排他锁，也就是说无法进行写、更新等操作的。

3.虽然说对 t2 表只是一个简单的 select 操作，用一致性读就可以了，oracle 也是这么做的，它通过 MVCC 技术实现的多版本数据来实现一致性读，不需要给 t2 加任何所，但是 InnoDB 却给 t2 加了共享锁，并没有使用多版本数据一致性技术。

4.MySQL 之所以这么做，就是为了保证恢复和复制的正确性，因为不加锁的话，如果其他事务对 t2 做了更新操作，可能导致数据恢复的错误结果。

5.我们用下面的操作范例来展示可能出现的问题。

*********CTAS 操作不给原表加锁带来的安全问题**********

1.这里开启两个 session，分别记为 a 和 b，我们在 a 和 b 中都设置 set autocommit=0，但是在 a 中进行 set innodb_locks_unsafe_for_binlog='on';

2.在 a 中操作如下：

insert into t1 select * from t2 where name = '1';

之后可以在 b 中进行进行操作：

update t2 set name='8' where name = '1';

3.首先在 b 中 commit，然后在 a 中 commit。

4.此时在两个表中查看数据，结果也是符合逻辑的。

5.但是，在 binlog 中，更新操作的位置在 insert.select 之前，如果使用整个 binlog 进行数据恢复，恢复的结果就会与实际的应用逻辑不符，如果进行复制，就会导致主从数据库不一致。

6.如果 CTAS 语句的 select 是范围条件，InnoDB 还会给原表添加“间隙锁(Next-Lock)”。

7.因此，insert.....select....和 create table ....select...语句，坑你会阻止对原表的并发更新，造成对原表锁的等待，如果查询比较复杂的话，会造成严重的性能问题，我们在应用中应该尽量避免使用。

8.实际上，MySQL 将这种 SQL 叫做不确定(non-deterministic)的 SQL，不推荐使用。

9.如果应用中一定要用这种 SQL 来实现业务逻辑，又不希望对源表的并发更新产生影响，可以采取如下措施：

(1)将 innodb_locks_unsafe_for_binlog 的值设置为 on，强制 MySQL 使用多版本数据一致性读，它的代价就是无法用 binlog 正确地恢复或者复制数据，不推荐。

(2)使用 select * from source_tab ....into outfile 和 load data infile....语句组来间接实现，采用这种方式 MySQL 不会给 source_tab 加锁。

**********不同隔离级别下的一致性读及锁的差异**********

1.锁和多版本数据是 InnoDB 实现一致性读和 ISO/ANSI SQL92 隔离级别的手段。

2.在不同的隔离级别下，InnoDB 处理 SQL 时采用的一致性读策略和需要的锁是不同的。

3.数据恢复和复制机制的特点，也对一些 SQL 的一致性读策略和锁策略有很大影响。

4.隔离级别越高，InnoDB 给记录集加的锁就越严格，产生锁冲突的可能性也就越高，从而对并发性事务处理性能的影响也就越大。

5.因此我们在应用中，应该尽量使用较低级别的隔离级别，以减少锁争用的几率。

6.通过优化事务逻辑，大部分应用使用 read commited 隔离级就够了，对于一些确实需要更高隔离级别的事务，可以通过在程序中执行 set session transaction isolation level repeatable read 或者 set session transaction isolation level serializable 动态改变事务隔离级别的方式满足需求。

**********什么时候使用表锁**********

1.对于 InnoDB 表，在绝大部分情况下都应该使用行级锁，因为事务和行锁往往是我们选择 InnoDB 表的理由。

2.个别事务中可以考虑使用表级锁：

(1)事务需要更新大部分或者全部数据，表又比较大，如果使用默认的行锁，会使得事务执行效率低，而且可能造成其他事务长时间锁等待和锁冲突，这种情况下可以考虑使用表锁来提高该事务的执行速度。

(2)事务涉及多个表，比较复杂，很可能引起死锁，造成大量事务回滚，这种情况也可以考虑一次性锁定事务涉及的表，从而避免死锁、减少数据库因事务回滚带来的开销。

3.这种事务不能太多，否则，就应该考虑使用 MyISAM 表了。

**********InnoDB 的表锁**********

1.虽然使用 lock tables 虽然可以给 InnoDB 加表级锁，但是表锁并不是由 InnoDB 存储引擎管理的，而是由 MySQL server 负责的，仅仅当 autocommit=0,、innodb_table_locks=1(默认设置)时，InnoDB 层才能知道 MySQL 加的表锁，MySQL Server 也才能感知 InnoDB 加的行锁，这种情况下，InnoDB 才能自动识别涉及表级锁的死锁，否则 InnoDB 将无法自动检测并处理这种死锁。

2.使用 lock tables 对 InnoDB 加表锁时要注意，要将 autocommit 设置为 0，否则 MySQL 不会给表加锁，事务结束前，不要用 unlock tables 释放表锁，因为它会隐含的提交事务，commit 或者 rollback 并不能释放用 lock tables 加的表级锁，必须用 unlock tables 释放表锁。

3.正确操作范例：

```
 set autocommit =0;
 lock tables t1 write , t2 read, .....;
 .....
 commit ;
 unlock tables;
```

**********死锁**********

1.MyISAM 是 deadlock free 的，这是因为 MyISAM 总是一次获得所需的全部锁，要么全部满足，要么当代，因此不会出现死锁。

2.在 InnoDB 中，除了单个 SQL 组成的事务外，锁是逐步获得的，这就决定了在 InnoDB 中发生死锁的可能。

3.比如在两个 session 中都设置 autocommit 为 0，在第二个中 select * from t1 where id=1 for update，第一个中 select * from t1 where id=1 for update；然后做一些其他处理。

4.此时第一个 session 中进行 select * from t2 where id = 1 for update；此时会进入等待状态，而第二个再进行 select * from t1 where id =1 for update；的时候，就进入了死锁。

5.当两个事务都需要获得对方持有的排他锁才能继续完成事务，这种循环锁等待就是典型的死锁。

6.发生死锁后，InnoDB 一般都能自动检测到，并使一个事务释放锁并回退，另一个事务获得锁，继续完成事务。

7.在设计到外部锁、表锁的情况下，InnoDB 并不能完全自动检测到锁，这需要设置锁等待超时参数 innodb_lock_wait_timeout 来解决。

8.当然这个参数并不只是用来解决死锁问题，在并发访问量比较高的情况下，如果大量事务因为无法立即获得所需的锁而挂起，会占用大量计算机资源，造成严重性能问题，甚至拖垮数据库。

**********死锁的解决**********

1.通常来说，死锁都是应用设计的问题，通过调整业务流程、数据库对象设计、事务大小、访问数据库的 SQL 语句，绝大部分死锁都可以避免。

2.如果不同的程序会并发存取多个表，应尽量约定以相同的顺序来访问表，这样可以大大降低产生死锁的机会。当两个 session 访问表的顺序不同的时候，发生死锁的几率会增加，以相同的顺序来访问，死锁就会降低很多。

3.在程序批量方式处理数据的时候，如果事先对数据排序，保证每个线程按固定的顺序来处理记录，也可以大大降低出现死锁的可能。

4.在事务中，如果要更新记录，应该直接申请足够级别的锁，即排他锁，而不应该先申请共享锁，更新时再申请排他锁，因为当用户申请排他锁的时候，其他事务可能又已经获取了相同记录的共享锁，从而造成锁冲突，甚至死锁。

5.在 repeatable-read 隔离级别下，如果两个线程同时对相同条件记录用 select ...for update 加排他锁，在没有符合该条件记录情况下，两个线程都会加锁成功。程序发现记录尚不存在，就试图插入一条新记录，如果两个线程都这么做，就会出现死锁。这种情况下，将隔离级别改为 read commited 就可以避免问题。

6.当隔离级为 read committed 时，如果两个线程都先执行 select...for update，判断是否存在符合条件的记录，如果没有，就插入记录，此时，只有一个线程能插入成功，另一个线程会出现锁等待，当第一个线程提交后，第二个线程会因主键重排出错，虽然

这个线程出错了，但是会获得一个排他锁。此时如果有第三个线程又来申请排他锁，就会出现死锁。对于这种情况，可以直接做插入操作，然后再捕获主键重异常，或者在遇到主键重错误时，总是执行 rollback 释放获得的排他锁。

**********说明**********

1.如果出现死锁，可以用 show innodb status 命令来确定最后一个死锁产生的原因。

2.返回结果中包括死锁相关事务的详细信息，如引发死锁的 SQL 语句，事务已经获得的锁，正在等待什么锁，以及被回滚的事务等等。

**********小结**********

1.在 MyISAM 中，共享读锁(S)之间是兼容的，但共享读锁(S)与排他写锁(X)之间，以及排他写锁(X)之间是互斥的，也就是说读写是串行的。

2.在一定条件下，MyISAM 允许查询和插入并发执行，我们可以利用这一点来解决应用中对同一表查询和插入的锁争用问题。

3.MyISAM 默认的锁调度机制是写优先，这并不一定适合所有应用，用户可以通过设置 low-priority_updates 参数，或者在 insert、update、delete 语句中指定 low_priority 选项来调节读写锁的争用。

4.由于表锁的锁定粒度大，读写之间又是串行的，因此，如果更新操作较多，MyISAM 表可能会出现严重的锁等待，可以考虑用 InnoDB 表来减少锁冲突。

5.对于 InnoDB 表，主要有几项内容：

(1)InnoDB 的行锁是基于索引实现的，如果不通过索引来访问数据，InnoDB 会使用表锁。

(2)InnoDB 的间隙锁及其原因。

(3)在不同的隔离级别下，InnoDB 的锁机制和一致性读策略不同。

(4)MySQL 的恢复和赋值对 InnoDB 锁机制和一致性读策略也有较大影响。

(5)锁冲突甚至死锁很难完全避免。

**********减少锁冲突和死锁的方案**********

1.尽量使用较低的隔离级别。

2.精心设计索引，并尽量使用索引访问数据，使加锁更精确，从而减少锁冲突的机会。

3.选择合理的事务大小，小事务发生锁冲突的几率更小。

4.给记录集显示加锁时，最好易行请求足够级别的锁，比如要修改数据的话，最好直接申请排他锁，而不是先申请共享锁，修改时再请求排他锁，这样容易产生死锁。

5.不同的程序访问一组表时，应尽量约定以相同的顺序访问各表，对一个表而言，尽可能以固定的顺序存取表中的行，这样可以大大减少死锁的机会。

6.尽量用相等条件来访问数据，这样可以避免间隙锁对并发插入的影响。

7.不要申请超过实际需要的锁级别，除非必须，查询时不要显式加锁。

8.对于一些特定的事务，可以使用表锁来提高处理速度或减少死锁的可能性。

第五节：优化 server

**********说明**********

1.当服务第一次启动的时候，所有的启动参数都是系统默认的。

2.这些参数在很多生产环境下并不能完全满足实际的应用需求，所以用户就需要按照实际情况进行调整。

**********查看 MySQL Server 参数**********

1.MySQL 服务启动后，我们可以用 show variables 和 show status 命令来查看静态参数值和动态运行状态信息。

2.show variables 是在数据库启动后不会动态更改的值，比如缓冲区大小、字符集、数据文件名等。

3.show status 是数据库运行期间会动态变化的信息，比如锁等待、当前连接数。

4.也可以在操作系统下直接查看数据库参数或者数据库状态信息，命令：

mysqladmin -uroot vaiables

mysqladmin -uroot status

5.由于 MySQL 服务器的参数很多，如果需要了解某个参数的详细定义，可以使用：

mysqld --verbose --help|more

6.比如查看当前服务器字符集的设置，可以使用：

mysqld --verbose --help|grep character-set-server

**********重要参数**********

1.key_buffer_size

2.table_cache

3.innodb_buffer_pool_size

4.innodb_flush_log_at_trx_commit

5.innodb_additional_mem_pool_size

6.innodb_lock_wait_timeout

7.innodb_support_xa

8.innodb_log_buffer_size

9.innodb_log_file_size

**********key_buffer_size**********

1.官方对 key_buffer_size 的定义是"the size of the buffer used for index blocks for MyISAM"。

2.该参数是用来设置索引块(Index Blocks)索引的大小，它被所有线程共享，此参数只适用于 MyISAM 存储引擎。

3.MySQL5.1 之前只允许使用一个系统默认的 key_buffer，而 MySQL5.1 以后提供了多个 key_buffer，可以将制定的表索引缓存入制定的 key_buffer，这样可以更小地降低线程之间的竞争。

4.建立一个索引缓存的范例：

set global hot_cahe2.key_buffer_size = 128*1024；

其中 global 表示对每一个新的连接，此参数都生效，而 hot_cache2 是新的 key_buffer 名称，可以重新用 set 语句来对它进行修改数值。

5.然后可以把相关表的索引放到指定的索引缓存中，比如：

cache index t1，t2 in hot_cache2;

6.要想索引预装到默认 key_buffer 中，可以使用 load index into cache 语句，范例：

load index into cache t1；

7.如果要删除索引缓存，可以使用如下命令：

set global hot_cache2.key_buffer_size = 0;

8.查看默认的 key_buffer 的大小，使用：

show variables like 'key_buffer_size';

9.如果是删除默认的 key_buffer，虽然可以使用下面语句删除：

set global key_buffer_size = 0;

但是我们使用 show warnings 发现并没有删除。

10.cache index 命令在一个表和 key_buffer 之间建立一种联系，但每次服务器重启时 key_buffer 中的数据将清空。

11.如果想要每次服务器重启时相应表的索引能自动放到 key_buffer 中，可以在配置文件中设置 init-file 选项来指定包含 cache index 语句的文件路径，然后在对应的文件中写入 cache index 语句。

12.比如在 my.cnf 中设置如下：

key_buffer_size = 4G

hot_cache.key_buffer_size = 2G

cold_cache.key_buffer_size = 2G

init_file = /path/to/data-directory/mysqld_init.sql

13.然后在 mysqld_init.sql 中写入：

cache index a.t1,a.t2,b.t3 in hot_cache;

cache index a.t4,b.t4,b.t5 in cold_cache;

**********table_cache**********

1.MySQL 官方对 table_cache 的定义如下"The number of open tables for all threads".

2.这个参数表示数据库用户打开表的缓存数量。

3.每个连接进来，都会至少打开一个表缓存，因此，table_cache 与 max_connections 有关。

4.比如，对于 200 个并行运行的连接，应该让表的缓存至少有 200xN 个，这里 N 是可以执行的查询的一个连接中表的最大数量。

5.此外，还需要为临时表和文件保留一些额外的文件描述符。

6.可以通过检查 mysqld 的状态变量 open_tables 和 opened_tables 确定这个参数是否过小。

7.这两个参数的区别是前者表示当前打开的表缓存数，如果执行 flush tables 操作，则此系统会关闭一些当前没有使用的表缓存而使得此状态值减小。

8.而后者表示曾经打开的表缓存数，会一直进行累加，如果执行 flush tables 操作，值不会减小。

9.查看操作如下：

```
show global status like 'open_tables';
show global status like 'opened_tables';
```

**********innodb_buffer_pool_size**********

1.官方对它的定义如下"the size of the memory buffer innodb uses to cache data and indexes of its tables"。

2.这个参数定义了 InnoDB 存储引擎的表数据和索引数据的最大内存缓冲区大小。

3.和 MyISAM 引擎不同的是，MyISAM 的 key_buffer_size 只缓存索引键，而 innodb_buffer_pool_size 同时为数据块和索引块做缓存。

4.这个特性和 oracle 是一样的，这个值设置的越高，访问表中数据需要的磁盘 IO 就越小。

5.在一个专用的数据库服务器上，可以设置这个参数达机器物理内存大小的 80%。

**********innodb_flush_log_at_trx_commit**********

1.这个参数用来控制缓冲区中的数据写入到日志文件以及日志文件数据刷新到磁盘的操作时机。

2.对这个参数的设置可以对数据库在性能与数据安全之间进行折中。

3.当这个参数是 0 的时候，日志缓冲每秒一次地被写到日志文件，并且对日志文件做向磁盘刷新的操作，但是在一个事务提交不做任何操作。

4.当这个参数是 1 的时候，在每个事务提交时，日志缓冲被写到日志文件，并且对日志文件做向磁盘刷新的操作。

5.当这个操作是 2 的时候，在每个事务提交时，日志缓冲被写到日志文件，但不对日志文件做向磁盘刷新的操作，对日志文件每秒向磁盘做一次刷新操作。

6.它的默认设置为 1，也是最安全的设置，也就是每个事务提交的时候都会从 log buffer 写到日志文件，并且会实际刷新磁盘，但是这样性能有一定能够损失。

7.如果可以容忍在数据库崩溃的时候损失一部分数据，那么设置为 0 或者 2 都会有所改善。

8.设置为 0，则在数据库崩溃的时候会丢失那些没有被写入日志文件的事务，最多丢失 1 秒钟的事务，这种方式是最不安全的，也是效率最高的。

9.设置为 2，因为只是没有刷新到磁盘，但是已经写入日志文件，所以只要操作系统没有崩溃，那么并没有丢失数据，比设置为 0 更安全。

10.建议设置为 1.

**********innodb_additional_mem_pool_size**********

1.这个参数是 InnoDB 存储引擎用来存储数据库结构和其他内部数据结构的内存池的大小，其默认值是 1MB。

2.应用程序里的表越多，则需要在这里分配越多的内存。

3.如果 InnoDB 用光了这个池内的内存，则 InnoDB 开始从操作系统分配内存，并且往 MySQL 错误日志写警告信息。

4.没有必要给这个缓冲池分配非常大的空间，在应用相对稳定的情况下，这个缓冲池的大小也相对稳定，系统默认值为 1MB。

**********innodb_lock_wait_timeout**********

1.官方的定义为"Timeout in seconds an InnoDB transaction may wait for a lock before being rolled back"。

2.系统默认值为 50 秒。

3.MySQL 可以自动地监测行锁导致的死锁并进行相应的处理，但是对于表锁的死锁不能自动的检测。

4.所以该参数主要被用于在出现类似情况的时候等待指定的时间后回滚。

**********innodb_support_xa**********

1.官方给出的解释是“Enable InnoDB support for the xa two-phase commit”。

2.通过该参数设置是否支持分布式事务，默认值为 on 或者 1，表示支持分布式事务。

3.如果确认应用中不需要使用分布式事务，则可以关闭这个参数，减少磁盘刷新的次数并获得更好的 InnoDB 性能。

**********innodb_log_buffer_size**********

1.官方给出的定义如下”The size of the buffer which InnoDB uses to write log to the log files on disk”。

2.含义就是日志缓存的大小。

3.默认的设置在中等强度写入负载以及较短事务的情况下，一般都可以满足服务器的性能要求。

4.默认值为 1MB。

5.如果存在更新操作峰值或者负载较大，应该考虑加大它的值了。

6.如果它的值设置太高，可能会浪费内存，因为它每秒都会刷新一次，因此无需设置超过 1 秒所需的内存空间。

7.通常它通常设置为 8-16MB 就够了，越小的系统它的值越小。

**********innodb_log_file_size**********

1.官方的定义是“size of each log file in a log group”。

2.该参数是一个日志组中每个日志文件的大小。

3.此参数在高写入负载尤其是大数据集的情况下很重要。

4.系统默认值为 5MB。

5.这个值越大则性能相对越高，但是带来的副作用是，当系统灾难时恢复时间会加大。

**********小结**********

1.本小节的题目是优化 MySQL Server，实际上更多地介绍了一些服务器性能相关的参数。

2.对服务器的优化，实际也就是对这些参数的不断调整。

第六节：磁盘 IO

**********说明**********

1.作为应用系统的持久化层，不管数据库采取了什么样的 cache 层，但数据库最终要将数据储存到可以长久保存的 IO 设备--磁盘上。

2.当然磁盘的存取速度比 CPU、RAM 的速度慢很多。

3.因此，对于较大的数据库，磁盘 IO 一般会成为数据库的一个性能瓶颈。

4.当然我们的优化大部分都是想通过减少或延缓磁盘读写来减轻磁盘 IO 的压力及其对性能的影响。

5.解决磁盘 IO 问题，减少或延缓磁盘操作肯定是一个重要方面。

6.增强磁盘 IO 本身的性能和吞吐量也是一个重要方面。

7.下面我们从磁盘阵列、符号连接、裸设备等底层的方面来介绍。

**********磁盘阵列**********

1.RAID 是 Redundant Array of Inexpensive Disks 的缩写，翻译成中文就是“廉价磁盘冗余阵列”，通常叫做磁盘阵列。

2.RAID 就是按照一定策略将数据分布到若干物理磁盘上。

3.这样不仅增强了数据存储的可靠性，而且可以提高数据读写的整体性能。

4.因为通过分布实现了数据的“并行”读写。

5.raid 最早是用来取代大型计算机上高档存储设备的，相对于那些高档存储设备而言，raid 的价格很便宜，这也是其名称中带有“廉价”一次的原因。

6.但是很长的一段时间内，相对于 pc 而言，其价格绝对谈不上廉价。

7.不过最近几年，随着存储技术的发展，raid 开始真正廉价了。

**********raid 级别**********

1.根据数据分布和冗余方式，raid 分为许多级别。

2.不同的存储厂商提供的 raid 卡或设备支持的 raid 级别也不尽相同。

3.raid 0 也叫条带化，即 stripe，按照一定的条带大小(chunk size)将数据依次分布到各个磁盘，没有数据冗余。优点是数据并发读写速度快，无额外磁盘空间开销，投资省。缺点是数据无冗余保护，可靠性差。

4.raid 1 也叫磁盘镜像，即 mirror，两个磁盘一组，所有数据都同时写入两个磁盘，但读时从任一磁盘都可以。优点时数据有完全冗余保护，只要不出现两块镜像磁盘同时损坏，不会影响使用，可以提高并发读性能。缺点是容量一定的话，需要 2 倍的磁盘，投资比较大。

5.raid 10 是 raid1 和 raid0 的结合，也叫 raid 1+0，先对磁盘做镜像，再条带化，其实兼具 raid1 的可靠性和 raid0 的优良并发读写性能。优点是可靠性高，并发读写性能优良。缺点就是容量一定的话，需要 2 倍的磁盘，投资比较大。

6.raid 4 像 raid 0 一样对磁盘组条带化，不同的是：需要额外增加一个磁盘，用来写各 stripe 的校验纠错数据。优点就是 raid 中的一个磁盘损坏，其数据可以通过校验纠错数据计算出来，具有一定容错保护能力，读数据速度快。缺点就是每个 stripe 上数据的修改都要写校验纠错块，写性能受影响，所有纠错数据都在同一磁盘上，风险大，也会形成一个性能 ing 经，在出现坏盘时，读性能会下降。

7.raid 5 是对 raid 4 的改进，将每一个条带的校验纠错数据块也分布写到各个磁盘，而不是写到一个特定的磁盘。优点基本同 raid4，只是其写性能和数据保护能力要强一点。缺点就是写性能不及 raid0、raid1 和 raid10，容错能力也不及 raid1，在出现坏盘时，读性能下降。

**********如何选择 raid 级别**********

1.我们可以通过数据读写的特点、可靠性要求、投资预算等来选择合适的 raid 级别。

2.数据读写都很频繁，可靠性要求也较高的，最好选择 raid 10；

3.数据读很频繁，写相对较少，对可靠性有一定要求，可以选择 raid5.

4.数据读写都很频繁，但可靠性要求不高，可以选择 raid0.

******虚拟文件卷或软 raid**********

1.最初，raid 都是由硬件来实现的，要使用 raid，至少需要一个 raid 卡。

2.现在的操作系统中提供一些软件包，也模拟实现了一些 raid 的特性。

3.虽然它们性能上不如硬 raid，但是相比单个磁盘，性能和可靠性都有所改善。

4.比如 Linux 下的逻辑卷(Logical Volume)系统 lvm2，支持条带化。

5.Linux 下的 MD(Multiple Device)驱动，支持 raid0、raid1、raid4、raid5、raid6 等。

6.在不具备硬件条件的环境下，可以考虑使用虚拟文件卷或软 raid 技术。

**********symbolic links**********

1.MySQL 的数据库名和表名是与文件系统的目录名和文件名对应的。

2.默认情况下，创建的数据库和表都存放在参数 datadir 定义的目录下。

3.如果不使用 raid 或逻辑卷，所有的表都存放在一个磁盘设备上，无法发挥出多磁盘并行读写的优势。

4.在这种情况下，我们就可以利用操作系统的符号连接(symbolic links)将不同的数据库或表、索引指向不同的物理磁盘，从而达到分布磁盘 IO 的目的。

5.常用方式有：

(1)将一个数据库指向其他物理磁盘。

(2)将 MyISAM 表的数据文件或索引文件指向其他物理磁盘。

(3)windows 下使用符号连接。

6.所谓将一个数据库指向其他物理磁盘，方法就是先在目标磁盘上创建目录，然后在创建从 MySQL 数据目录到目标目录的符号连接。操作范例：

shell>mkdir /otherdisk/databases/test

shell>ln -s /otherdisk/databases/test /path/to/datadir

7.将 MyISAM 表的数据文件或索引文件指向其他物理磁盘，不过值得注意的是其他存储引擎不支持。操作如下：

(1)对于新建的表，可以通过在 create table 语句中增加 data directory 和 index directory 选项来完成，比如：

>create table test(id int primary key,

->name varchar(20))

->type = myisam

->data directory = '/disk2/data'

->index directory = '/disk3/index';

(2)对于已有的表，可以先将其数据文件.MYD 或索引文件.MYI 转移到目标磁盘，然后再建立符号连接即可。需要注意的是表定义文件.frm 必须位于 MySQL 数据文件目录下，不能使用符号连接。

8.在 win 下使用符号连接，在 win 下，是通过在 MySQL 数据文件目录下创建包含目标路径并以".sym"结尾的文本文件来实现的。

9.比如假设 MySQL 的数据文件是 C:\mysql\data，要把数据库 foo 存放到 D:\data\foo,则步骤如下：

(1)创建目录 D:\data\foo

(2)创建文件 C:\mysql\data\foo.sym,在其中输入 D:\data\foo

这样在数据库 foo 创建的表都会存储到 D:\data\foo 目录下。

10.使用 Symbolic Links 存在一定的安全风险，如果不使用 Symbolic Links，应该通过启动参数 skip-symbolic-links 禁用该功能。

**********禁用 atime 属性**********

1.atime 是*nix 系统下的一个文件属性。

2.每当读取文件时，操作系统都会将读操作发生的时间写到磁盘上。

3.对于读写频繁的数据库文件来说，记录文件的访问时间一般没有任何用处，却会增加磁盘系统的负担，影响 IO 的性能。

4.可以通过设置文件系统的 mount 属性，阻止操作系统写 atime 信息，以减轻磁盘 IO 的负担。

5.在 Linux 下的操作是：

(1)修改文件系统配置文件/etc/fstab,制定 noatime 选项：

LABEL=/home　　/home　　ext3　　noatime　　1 2

(2)然后重新 mount 文件系统：

#mount　-oremount　　/home

**********用裸设备存放 InnoDB 的共享表空间**********

1.裸设备即 Raw Device。

2.MyISAM 存储引擎有自己的索引缓存机制，但是数据文件的读写完全依赖于操作系统，操作系统磁盘 IO 缓存对 MyISAM 表的存取很重要。

3.但 InnoDB 存储引擎与 MyISAM 不同，它采用类似 oracle 的数据缓存机制来 cache 索引和数据，操作系统的磁盘 IO 缓存对其性能不仅没有帮助，甚至还有反作用。

4.因此，在 InnoDB 缓存充足的情况下，可以考虑使用 Raw Device 来存放 InnoDB 共享表空间。

5.操作如下：

(1)修改 MySQL 配置文件，在 innodb_data_file_path 参数中增加裸设备文件名并制定 newraw 属性：

[mysqld]

innodb_data_home_dir =

innodb_data_file_path=

/dev/hdd1:3Gnewraw;/dev/hdd2:2Gnewraw

(2)启动 MySQL，使其完成分区初始化工作，然后关闭 MySQL，此时还不能创建或修改 InnoDB 表。

(3)将 innodb_data_file_path 中的 newraw 改成 raw:

[mysqld]

innodb_data_home_dir=

innodb_data_file_path=/dev/hdd1:3Graw;/dev/hdd2:2Graw

(4)重启即可。

**********小结**********

1.在大多数的数据库系统中，磁盘 IO 都是影响系统性能的瓶颈。

2.我们主要讨论了 IO 的优化问题、文件系统分布的优化问题。

第七节：应用优化

**********解决方案**********

1.使用连接池

2.减少对 MySQL 的访问

3.负载均衡

4.其他方法

**********连接池**********

1.对于访问数据库来说，建立连接的代价比较昂贵。

2.连接池就是一个存放连接的池子。

3.我们可以把连接当做对象或者设备，统一放在一个池子里面，以前需要直接访问数据库的地方，现在都改为从这个池子里面获取连接来使用。

4.因为池子中的连接都已经预先创建好，可以直接分配给应用使用，因此大大减少了创建新连接所耗费的资源。

5.连接返回后，本次访问将连接交还给连接池，以供新的访问使用。

**********减少对 MySQL 的访问**********

1.避免对统一数据做重复检索

2.使用查询缓存

3.增加 cache 层

**********避免对同一数据做重复检索**********

1.应用中需要理清对数据库的访问逻辑，能一次连接能取出所有结果的，绝对不用两次连接。

2.这样可以大大减少对数据库无谓的重复访问。

**********使用查询缓存**********

1.MySQL 的查询缓存也就是 MySQL Query Cache。

2.它是在 4.1 版本以后新增的功能。

3.它的作用是存储 select 查询的文本以及相应结果。如果随后收到一个相同的查询，服务器会从查询缓存中重新得到查询结果，而不需要再解析和执行查询。

4.查询缓存的适用对象是更新不频繁的表，当表更改(包括表结构和表数据)后，查询缓存值的相关条目被清空。

5.要查看它的情况可以使用如下语句：

show variables like '%query_cache%';

6.它的返回结果中解释如下：

(1)have_query_cache 表明服务器是否已经配置了高速缓存。

(2)query_cache_size 表明缓存区大小，单位是 MB。

(3)query_cache_type 的变量值从 0 到 2，含义为：

0 或者 off 表示缓存关闭，

1 或者 on 表示缓存打开，即使用 sql_no_cache 提示的 select 除外，
2 或者 demand 表示只有带 sql_cache 的 select 使用高速缓存

7.通过 show status 命令，可以监视查询缓存的使用情况，比如：

(1)Qcache_queries_in_cache 在缓存中已注册的查询数目

(2)Qcache_inserts 被就到缓存中的查询数目

(3)Qcache_hits 缓存采样数书目

(4)Qcache_lowmem_prunes 因缺少内存被从缓存中删除的查询数目

(5)Qcache_not_cached 没有被缓存的查询数目(不能被缓存的，或由于 query_cache_tye)

(6)Qcache_free_memory 查询缓存的空闲内存总数

(7)Qcache_free_blocks 查询缓存中的空闲内存块的数目

(8)Qcache_total_blocks 查询缓存中的块的总数目

**********增加 cache 层**********

1.我们可以在应用中增加 cache 层来达到减轻数据库的负担的目的。

2.cache 层有很多，也有很多种实现的方式，只要能达到降低数据库的负担，又能满足应用就可以，这就需要根据应用的实际情况进行特殊处理。

3.比如可以使用文件缓存等，不过这里要注意数据更新的问题，多长时间刷新一次 cache，需要根据具体应用环境来进行处理。

4.再比如用户可以在应用端建立一个二级数据库，把访问频率高的数据放到二级数据库上，在设定一个机制与主数据库同步，我们的主要操作就可以在该二级数据库上进行，减少主数据库的压力。

**********负载均衡**********

1.负载均衡(load balance)是实际应用中使用比较普遍的一种优化方法。

2.它的机制就是使用某种均衡算法，将固定的负载量分布到不同的服务器上，以此来减轻单台服务器的负载，达到优化的目的。

3.负载均衡可以用在系统中的各个层面中，从前台的 web 服务器到中间层的应用服务器，最后到数据层的数据库服务器，都可以使用。

4.对于数据层基本的方式为：

(1)利用 MySQL 复制分流查询

(2)采用分布式数据库架构

**********利用 MySQL 复制分流查询**********

1.利用 MySQL 的主从复制可以有效地分流更新操作和查询操作。

2.具体的实现是一个主服务器承担更新操作，多台从服务器承担查询操作，主从之间通过复制实现数据的同步。

3.多台从服务器的优势：

(1)确保可用性

(2)创建不同的索引以满足不同查询的需要。

4.对于主从之间不需要复制全部表的情况，可以通过在主服务器上搭建一个虚拟的从服务器，将需要复制到从服务器的表设置成 BlackHole 引擎，然后定义 replicate-do-table 参数只复制这些表，这样就可以过滤出需要复制的 binlog，减少了传输 binlog 的宽带。

5.因为搭建的虚拟从服务器只起到过滤 binlog 的作用，并没有实际记录任何数据，所以对主数据库服务器的性能影响也非常有限。

6.通过复制来分流查询是减少主数据库负载的一个常用方法，但是这种方法也存在一些问题，最主要的问题就是当主数据库上更新频繁或者网络出现问题的时候，主从之间的数据可能存在有比较大的延迟更新，从而造成查询结果和主数据库上有所差异。

**********分布式数据库**********

1.分布式数据库架构适合大数据量、负载高的情况。

2.它具有良好的扩展性和高可用性。

3.通过在多台服务器之间分布数据，可以实现在多态服务器之间的负载平均，提高了访问的执行效率。

4.具体的实现上，可以使用 MySQL 的 Cluster 功能或者用户自己编写的程序来实现全局事务。

5.需要注意的是当前分布式事务只支持 InnoDB 存储引擎，因此如果自己编写程序来实现分布式架构数据库的话，那么就必须采用 InnoDB 存储引擎。

**********其他优化措施**********

1.对于没有删除行操作的 MyISAM 表，插入操作和查询操作可以并行进行，因为没有删除操作的表查询期间不会阻塞插入操作。

2.对于确实需要执行删除操作的表，尽量在空闲时间进行批量删除操作，并且在进行删除操作之后进行 optimize 操作来消除由于删除操作带来的空洞，以避免将来的更新操作阻塞其他操作。

3.充分利用列有默认值的事实，只有当插入的值不同于默认值的时候，才明确的插入值，这会减少 MySQL 需要做的语法分析从而提高插入速度。

4.表的字段尽量不使用自增长变量，在高并发的情况下该字段的自增可能对效率有比较大的影响，推荐通过应用来实现字段的自增长。

**********小结**********

1.应用的优化也是数据库优化的主要组成部分，数据库本身的优化是有一定的局限性的，到了一定程度很难有再大的提升。

2.本节从连接池、减少数据库访问、负载均衡以及 cache 层等方面进行了讨论。

**********辛星推荐**********

1.推荐一首歌，带泪的鱼的《爱释枷锁》。

2.第一次听就被忧伤的旋律，流畅的语速，朦胧的意境吸引了。

3.辛星与您，共同进步。