day05-MySQL性能优化

学习目标

- [ ] 了解MySQL优化
- [ ] 了解常见的优化思路
- [ ] 了解查询优化
- [ ] 了解索引优化
- [ ] 了解存储优化
- [ ] 了解数据库结构优化
- [ ] 了解查询缓存等缓存优化

1 优化介绍

在进行优化讲解之前，先请大家记住**不要听信你看到的关于优化的"绝对真理"，而应该是在==实际的业务场景==下通过测试来验证你关于执行计划以及响应时间的假设。**本课程只是给大家提供一些优化方面的方向和思路，而具体业务场景的不同，使用的MySQL服务版本不同，都会使得优化方案的制定也不同。

1.1 MySQL介绍

MySQL凭借着出色的性能、低廉的成本、丰富的资源，已经成为绝大多数互联网公司的首选关系型数据库。可以看到Google，Facebook，Twitter，百度，新浪，腾讯，淘宝，网易，久游等绝大多数互联网公司数据库都是用的MySQL数据库，甚至将其作为核心应用的数据库系统。

虽然性能出色，但所谓"好马配好鞍"，如何能够更好的使用它，已经成为开发工程师的必修课，我们经常会从职位描述上看到诸如"精通MySQL"、"SQL语句优化"、"了解数据库原理"等要求。我们知道一般的应用系统，读写比例在10:1左右，而且插入操作和一般的更新操作很少出现性能问题，遇到最多的，也是最容易出问题的，还是一些复杂的查询操作，所以查询语句的优化显然是重中之重。

我们将这里进行一个较为全面的分析，让大家了解到MySQL的性能到底与哪些地方有关，以便于让大家寻找出其性能问题的根本原因，而尽可能清楚的知道该如何去优化自己的数据库。

##1.2 优化要考虑的问题

注意：优化有风险，涉足需谨慎！

1.2.1 优化可能带来的问题

- 优化不总是对一个单纯的环境进行，还很可能是一个复杂的已投产的系统！
- 优化手段有很大的风险，一定要意识到和预见到！
- 任何的技术可以解决一个问题，但必然存在带来一个问题的风险！
- 对于优化来说调优而带来的问题,控制在可接受的范围内才是有成果。
- 保持现状或出现更差的情况都是失败！

###1.2.2 优化的需求

- 稳定性和业务可持续性,通常比性能更重要！

- 优化不可避免涉及到变更，变更就有风险！
- 优化使性能变好，维持和变差是等概率事件！
- 优化应该是各部门协同，共同参与的工作，任何单一部门都不能对数据库进行优化！

所以优化工作,是由业务需要驱使的！！！

###1.2.3 优化由谁参与

在进行数据库优化时，应由数据库管理员、业务部门代表、应用程序架构师、应用程序设计人员、应用程序开发人员、硬件及系统管理员、存储管理员等，业务相关人员共同参与。

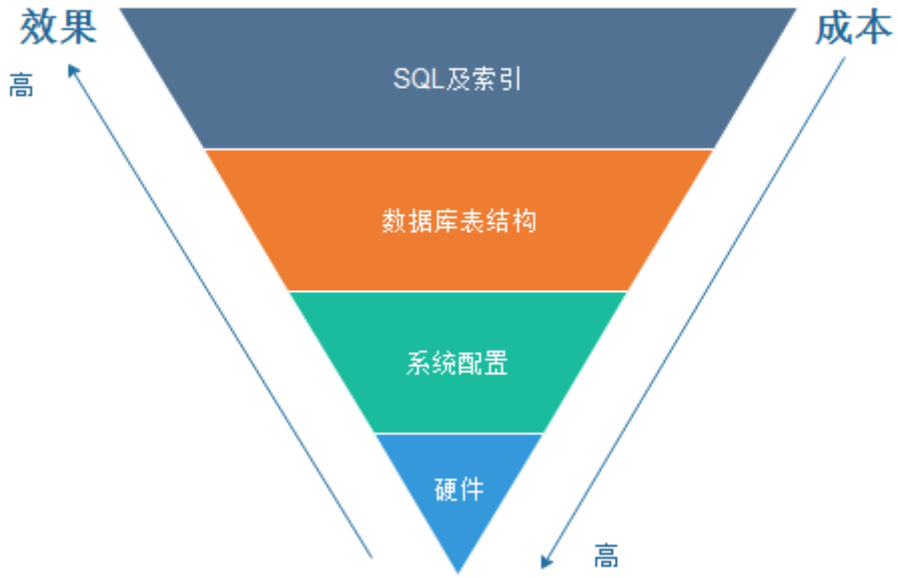
1.2.4 优化的方向

在数据库优化上有两个主要方向：即安全与性能。

- 安全 : 数据安全性
- 性能 : 数据的高性能访问

本课程主要是在**性能优化**方向进行介绍

1.2.5 优化的维度



从上图中可以看出，我们把数据库优化分为四个纬度：硬件，系统配置，数据库表结构，SQL及索引

硬件： CPU、内存、存储、网络设备等

系统配置： 服务器系统、数据库服务参数等

数据库表结构： 高可用、分库分表、读写分离、存储引擎、表设计等

Sql及索引： sql语句、索引使用等

- 从优化成本进行考虑：硬件>系统配置>数据库表结构>SQL及索引
- 从优化效果进行考虑：硬件<系统配置<数据库表结构<SQL及索引

1.3 数据库使用优化思路

本课程尽可能的全面介绍数据库的调优思路，但是在多数时候，我们进行调优不需要进行这么全面、大范围的调优，一般情况下，我们进行数据库层面的优化就可以了，那我们该如何调优的呢?

应急调优的思路：

针对突然的业务办理卡顿，无法进行正常的业务处理！需要立马解决的场景！

1. show processlist（查看连接session状态）
2. explain(分析查询计划)，show index from tableName（分析索引）
3. show status like '%lock%'; # 查询锁状态

常规调优的思路：

针对业务周期性的卡顿，例如在每天10-11点业务特别慢，但是还能够使用，过了这段时间就好了。

1. 开启**慢查询日志**，运行一天
2. 查看slowlog，分析slowlog，分析出查询慢的语句。
3. 按照一定优先级，进行一个一个的排查所有慢语句。
4. 分析top sql，进行explain调试，查看语句执行时间。
5. 调整索引或语句本身。

2 优化实践

2.1 查询优化

###2.1.1 MySQL查询流程

我们该如何进行sql优化呢， 首先我们需要知道，sql优化其实主要是解决查询的优化问题，所以我们先从数据库的查询开始入手，下面这幅图显示了查询的执行路径：

① 客户端将查询发送到服务器；

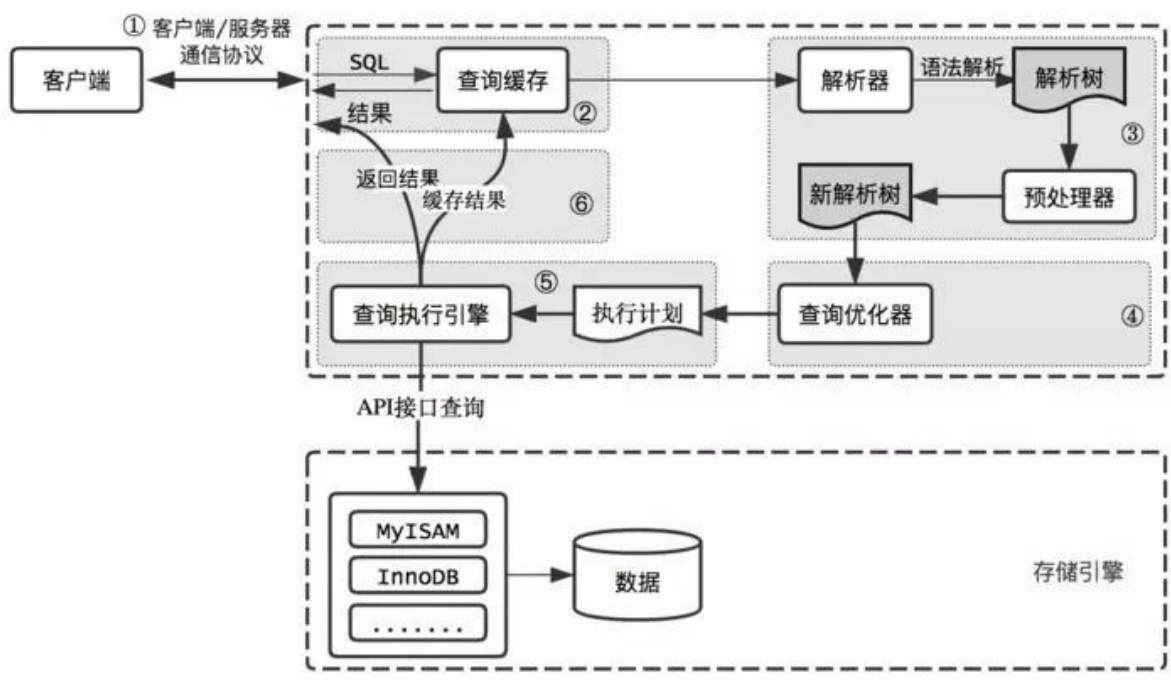
② 服务器检查查询缓存，如果找到了，就从缓存中返回结果，否则进行下一步。

③ 服务器解析，预处理。

④ 查询优化器优化查询

⑤ 生成执行计划，执行引擎调用存储引擎API执行查询

⑥服务器将结果发送回客户端。



查询缓存

在解析一个查询语句之前，如果查询缓存是打开的，那么MySQL会优先检查这个查询是否命中查询缓存中的数据，如果命中缓存直接从缓存中拿到结果并返回给客户端。这种情况下，查询不会被解析，不用生成执行计划，不会被执行。

语法解析和预处理器

MySQL通过关键字将SQL语句进行解析，并生成一棵对应的“解析树”。MySQL解析器将使用MySQL语法规则验证和解析查询。

查询优化器

语法树被校验合法后由优化器转成查询计划，一条语句可以有很多种执行方式，最后返回相同的结果。优化器的作用就是找到这其中最好的执行计划。

查询执行引擎

在解析和优化阶段，MySQL将生成查询对应的执行计划，MySQL的查询执行引擎则根据这个执行计划来完成整个查询。最常使用的也是比较最多的引擎是MyISAM引擎和InnoDB引擎。mysql5.5开始的默认存储引擎已经变更为innodb了。

前面的查询流程分析，我们大概了解了MySQL是如何执行的。现在我们先从查询优化部分开始。

sql是我们和数据库交流最重要的部分，所以我们在调优的时候，需要花费的大量时间就在sql调优上面。常见的分析手段有==慢查询日志，EXPLAIN 分析查询==，通过定位分析性能的瓶颈，才能更好的优化数据库系统的性能。

2.1.2 慢查询日志(重要)

默认情况下慢日志查询是禁用的。通过`show variables like '%slow_query_log%'`查看慢查询日志的开启情况

```
mysql> show variables like '%slow_query_log%'
    -> ;
+---------------------+-----------------------------------------+
| Variable_name       | Value                                   |
+---------------------+-----------------------------------------+
| slow_query_log      | OFF   关闭                              |
| slow_query_log_file | /var/lib/mysql/1de7994592cd-slow.log |
+---------------------+-----------------------------------------+
2 rows in set (0.13 sec)
```

2.1.2.1 慢查询日志开启

如要开启慢查询日志，可以使用命令`set global slow_query_log=1;`。再次查看慢查询日志，可以发现已经开启。

```
mysql> set global slow_query_log=1;
Query OK, 0 rows affected (0.37 sec)

mysql> show variables like '%slow_query_log%';
+---------------------+--------------------------------+
| Variable_name       | Value                          |
+---------------------+--------------------------------+
| slow_query_log      | ON                             |
| slow_query_log_file | /var/lib/mysql/slow-query.log |
+---------------------+--------------------------------+
2 rows in set (0.00 sec)
```

但是当重启MySQL后，则又会关闭。如果需要长期开启的话，需要在配置文件**/etc/my.cnf**或my.ini中在[mysqld]一行下面加入三个配置参数

```
slow_query_log=ON
slow-query-log-file=/var/lib/mysql/slow-query.log
long_query_time=0
```

修改完成后，重启mysql：service mysqld restart

- **慢查询分析**

如果慢查询日志中记录内容很多，可以使用**mysqldumpslow**工具（MySQL客户端安装自带）来对慢查询日志进行分类汇总。mysqldumpslow对日志文件进行了分类汇总，显示汇总后摘要结果。

进入log的存放目录，运行：

```
[root@mysql_data]# mysqldumpslow slow-query.log
Reading mysql slow query log fromslow-query.log
Count: 2 Time=11.00s (22s) Lock=0.00s (0s)Rows=1.0 (2), root[root]@mysql
select count(N) from t_user;
```

mysqldumpslow命令

`mysqldumpslow -s c -t 10 slow-query.log`

这会输出记录次数最多的10条SQL语句，其中：

-s, 是表示按照何种方式排序，c、t、l、r分别是按照记录次数、时间、查询时间、返回的记录数来排序，ac、at、al、ar，表示相应的倒叙

-t, 是top n的意思，即为返回前面多少条的数据；

例如：

`mysqldumpslow -s r -t 10 slow-query.log`：得到返回记录集最多的10个查询。

`mysqldumpslow -s t -t 10 -g “leftjoin” slow-query.log`：得到按照时间排序的前10条里面含有左连接的查询语句。

使用mysqldumpslow命令可以非常明确的得到各种我们需要的查询语句，对MySQL查询语句的监控、分析、优化是MySQL优化非常重要的一步。开启慢查询日志后，由于日志记录操作，在一定程度上会占用CPU资源影响mysql的性能，但是可以阶段性开启来定位性能瓶颈。

2.1.3 执行计划Explain

2.1.3.1 Explain概述

==使用explain关键字可以模拟优化器执行SQL查询语句,==从而知道MYSQL是如何处理SQL语句的。我们可以用执行计划来分析查询语句或者表结构的性能瓶颈

2.1.3.2 Explain作用

1. 查看表的读取顺序
2. 查看数据库读取操作的操作类型
3. 查看哪些索引有可能被用到
4. 查看哪些索引真正被用到
5. 查看表之间的引用
6. 查看表中有多少行记录被优化器查询

2.1.3.3 语法

- 语法

```
explain  sql语句
```

- 示例

```
explain select * from tb_user;
```

2.1.3.4 各字段解释

- 准备工作

```
create table t1(
  id int primary key,
  name varchar(20),
  col1 varchar(20),
  col2 varchar(20),
  col3 varchar(20)
);

create table t2(
  id int primary key,
  name varchar(20),
  col1 varchar(20),
  col2 varchar(20),
  col3 varchar(20)
);
```

```
create table t3(
  id int primary key,
  name varchar(20),
  col1 varchar(20),
  col2 varchar(20),
  col3 varchar(20)
);

insert into t1 values(1,'zs1','col1','col2','col3');
insert into t2 values(1,'zs2','col2','col2','col3');
insert into t3 values(1,'zs3','col3','col2','col3');

create index ind_t1_c1 on t1(col1);
create index ind_t2_c1 on t2(col1);
create index ind_t3_c1 on t3(col1);

create index ind_t1_c12 on t1(col1,col2);
create index ind_t2_c12 on t2(col1,col2);
create index ind_t3_c12 on t3(col1,col2);
```

==2.1.3.4.1 id==

- select 查询的序列号,包含一组数字,表示**查询中执行Select子句或操作表的顺序**
- 两种情况:

1. id值相同：执行顺序由上而下。

```
explain select t2.* from t1,t2,t3 where t1.id = t2.id and t1.id= t3.id and
t1.name = 'zs';
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t1	ALL	PRIMARY	(Null)	(Null)	(Null)	1	Using where
1	SIMPLE	t2	eq_ref	PRIMARY	PRIMAI	4	mysql_optimization.t1.id	1	
1	SIMPLE	t3	eq_ref	PRIMARY	PRIMAI	4	mysql_optimization.t1.id	1	Using index

2. id值不同：id值越大优先级越高。

```
explain select t2.* from t2 where id = (select id from t1 where id = (select
t3.id from t3 where t3.name='zs3'));
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	t2	const	PRIMARY	PRIMAR\	4	const	1	
2	SUBQUERY	t1	const	PRIMARY	PRIMAR\	4		1	Using index
3	SUBQUERY	t3	ALL	(Null)	(Null)	(Null)	(Null)	1	Using where

2.1.3.4.2 select_type

```
- SIMPLE : 简单的select查询,查询中不包含子查询或者UNION

- PRIMARY: 查询中若包含复杂的子查询,最外层的查询则标记为PRIMARY

- SUBQUERY : 在SELECT或者WHERE列表中包含子查询

- DERIVED : 在from列表中包含子查询被标记为DRIVED衍生,MYSQL会递归执行这些子查询,把结果放到临时表中

- UNION: 若第二个SELECT出现在union之后,则被标记为UNION, 若union包含在from子句的子查询中,外层select被标记为:derived

- UNION RESULT: 从union表获取结果的select
```

执行sql

```
explain select col1,col2 from t1 union select col1,col2 from t2;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	t1	index	(Null)	ind_t1_c	126	(Null)	1	Using index
2	UNION	t2	index	(Null)	ind_t2_c	126	(Null)	1	Using index
(Null)	UNION RESULT	<union1,2>	ALL	(Null)	(Null)	(Null)	(Null)	(Null)	

2.1.3.4.3 table

显示这一行的数据是和哪张表相关

==2.1.3.4.4 type==

type显示的是访问类型，是较为重要的一个指标，结果值从最好到最坏依次是

`system > const > eq_ref > ref > fulltext > ref_or_null > index_merge > unique_subquery > index_subquery > range(尽量保证) > index > ALL`

一般来说，得保证查询至少达到range级别，最好能达到ref。

- **system**：表中只有一行记录(系统表), 这是const类型的特例, 基本上不会出现
- **const**：通过索引一次查询就找到了，const用于比较primary key或者unique索引，该表最多有一个匹配行, 在查询开始时读取。由于只有一行, 因此该行中列的值可以被优化器的其余部分视为常量。const 表非常快, 因为它们只读一次。

```
explain select * from t1 where id=1
```

- **eq_ref**：读取本表中和关联表表中的每行组合成的一行。除 了 system 和 const 类型之外, 这是最好的联接类型。当连接使用索引的所有部分时, 索引是主键或唯一非 NULL 索引时, 将使用该值。

```
explain select * from t1,t2 where t1.id = t2.id;
```

- ==ref==：非唯一性索引扫描，返回匹配某个单独值的所有行，本质上也是一种索引访问，它返回所有符合条件的行。

```
explain select * from t1 where col1='zs1';
```

- ==range== : 只检索给定范围的行, 使用一个索引来选择行.key列显示的是真正使用了哪个索引,一般就是在where条件中使用between,>,<,in 等范围的条件,这种在索引范围内的扫描比全表扫描要好,因为它只在某个范围中扫描,不需要扫描全部的索引

```
explain select * from t1 where id between 1 and 10;
```

- ==index== : ==扫描整个索引表==, index 和all的区别为index类型只遍历索引树. 这通常比all快,因为索引文件通常比数据文件小,虽然index和all都是读全表,但是index是从索引中读取,而all是从硬盘中读取数据

```
explain select id from t1;
```

- ==all== : 全表扫描 ,将遍历全表以找到匹配的行

```
explain select * from t1;
```

> 注意: 开发中,我们得保证查询至少达到range级别,最好能达到ref.
>
> 如果百万条数据出现all, 一般情况下就需要考虑使用索引优化了

==2.1.3.4.5 key==

查询过程中真正使用的索引，如果为null，则表示没有使用索引

```
explain select t2.* from t1,t2,t3 where t1.col1 = ' ' and t1.id = t2.id and
t1.id= t3.id;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t1	ref	PRIMARY,ind_t1_c1,i	ind_t1_c1	63	const	1	Using where; Using index
1	SIMPLE	t2	eq_ref	PRIMARY	PRIMARY	4	mysql_opt	1	
1	SIMPLE	t3	eq_ref	PRIMARY	PRIMARY	4	mysql_opt	1	Using index

2.1.3.4.6 key_len

索引中使用的字节数，可通过该列计算查询中使用的索引的长度，长度越短越好。

2.1.3.4.7 ref

显示索引的哪一列被使用了,如果可能的话,是一个常数.哪些列或者常量被用于查找索引列上的值

```
explain select t2.* from t1,t2,t3 where t1.col1 = ' ' and t1.id = t2.id and
t1.id= t3.id;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t1	ref	PRIMARY,ind_t1_c1,i	ind_t1_c1	63	const	1	Using where; Using index
1	SIMPLE	t2	eq_ref	PRIMARY	PRIMARY	4	mysql_opt	1	
1	SIMPLE	t3	eq_ref	PRIMARY	PRIMARY	4	mysql_opt	1	Using index

==2.1.3.4.8 rows==

根据表统计信息及索引选用的情况,估算找出所需记录要读取的行数 (有多少行记录被优化器读取) ,越少越好

==2.1.3.4.9 extra==

包含不适合在其他列中显示但十分重要的额外信息。

2.2 索引优化

2.2.1 准备工作

创建以下表：

```
create database mysql_optimization;
use mysql_optimization;

CREATE TABLE `tb_table` (
  `id` int(11) NOT NULL AUTO_INCREMENT COMMENT '主键',
  `name` varchar(20) DEFAULT NULL COMMENT '姓名',
  `number` int(11) DEFAULT NULL COMMENT '编号',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

执行以下sql，批量添加1000000条数据：

```
drop procedure if exists tb_insert;
CREATE PROCEDURE tb_insert()
BEGIN
DECLARE i INT;
SET i = 0;
START TRANSACTION;
WHILE i < 1000000 DO -- 1000000即插入1000000条数据
    INSERT INTO tb_table (`name`,`number`) VALUES (concat("张三",i),i);
    SET i = i+1;
END WHILE;
COMMIT;
END;

call tb_insert();
```

在表没有添加索引的时候，都执行以下查询：

```
SELECT * FROM tb_table WHERE number = 500000
```

2.2.2 索引介绍

2.2.2.1 什么是索引

==索引（Index）是帮助MySQL高效获取数据的数据结构。==在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式指向数据，这样就可以在这些数据结构上实现高效的查找算法，这种数据结构就是索引。

一般来说索引本身也很大，不可能全部存储在内存中，因此往往以索引文件的形式存放在磁盘中.。我们平常所说的索引，==如果没有特别说明都是指BTree索引(平衡多路搜索树).==

2.2.2.2 索引的优缺点

- 优点
  - 类似大学图书馆建书目索引，提高数据检索的效率，降低数据库的IO成本.
  - 通过索引列对数据进行排序, 降低数据排序的成本，降低了CPU的消耗
- 缺点

- 实际上索引也是一张表，该表保存了主键与索引字段，并指向实体表的记录，所以索引列也是要占用空间的
- ==虽然索引大大提高了查询速度，同时却会降低更新表的速度==，如对表进行INSERT、UPDATE和DELETE。因为更新表时，MySQL不仅要保存数据，还要保存一下索引文件每次更新添加了索引列的字段都会调整。因为更新所带来的键值变化后的索引信息
- 索引只是提高效率的一个因素，如果你的MySQL有大数据量的表，就需要花时间研究建立最优秀的索引，或优化查询语句

2.2.2.3 索引的类别

2.2.2.3.1 普通索引

最基本的索引，它没有任何限制.

```
CREATE index 索引名 on 表名(列名)
```

2.2.2.3.2 唯一索引

与前面的普通索引类似，不同的就是：==索引列的值必须唯一，但允许有空值==。如果是组合索引，则列值的组合必须唯一。

```
CREATE UNIQUE index 索引名 on 表名(列名)
```

2.2.2.3.3 主键索引

是一种特殊的唯一索引，这个时候需要一个表只能有一个主键，不允许有空值。一般是在建表的时候同时创建主键索引.也就是说==主键约束默认索引==

2.2.2.3.4 复合索引

指多个字段上创建的索引，只有在查询条件中使用了创建索引时的第一个字段，索引才会被使用。使用组合索引时遵循最左前缀规格

```
CREATE index 索引名 on 表名(列名,列名...)
```

2.2.2.3.5 全文索引

主要用来查找文本中的关键字，而不是直接与索引中的值相比较。fulltext索引跟其它索引大不相同，它更像是一个搜索引擎，而不是简单的where语句的参数匹配。目前只有**char、varchar，text** 列上可以创建全文索引。值得一提的是，在数据量较大时候，先将数据放入一个没有全局索引的表中，然后再用CREATE index创建fulltext索引，要比先为一张表建立fulltext然后再将数据写入的速度快很多。

```
CREATE TABLE `info` (
    `id` int(11) NOT NULL AUTO_INCREMENT ,
    `title` char(255) CHARACTER NOT NULL ,
    `content` text CHARACTER NULL ,
    `time` int(10) NULL DEFAULT NULL ,
    PRIMARY KEY (`id`),
    FULLTEXT (content)
);
```

2.2.2.4 索引的基本语法

- 创建

```
ALTER mytable ADD  [UNIQUE]  INDEX [indexName] ON 表名(列名)
```

- 删除

```
DROP INDEX [indexName] ON 表名;
```

- 查看

```
SHOW INDEX FROM 表名
```

- alter命令

```
-- 有四种方式来添加数据表的索引：
ALTER TABLE tbl_name ADD PRIMARY KEY (column_list): 该语句添加一个主键，这意味着索引值必须是唯一的，且不能为NULL。

ALTER TABLE tbl_name ADD UNIQUE index_name (column_list): 这条语句创建索引的值必须是唯一的（除了NULL外，NULL可能会出现多次）。

ALTER TABLE tbl_name ADD INDEX index_name (column_list): 添加普通索引，索引值可出现多次。

ALTER TABLE tbl_name ADD FULLTEXT index_name (column_list):该语句指定了索引为 FULLTEXT ，用于全文索引。
```

2.2.2.5 索引的存储结构

#####2.2.2.5.1 BTree索引

在前面的例子中我们看见有USING BTREE，这个是什么呢？这个就是MySQL所使用的索引方案，MySQL中普遍使用B+Tree做索引，也就是BTREE。

为什么使用B+树：

B+树是一个**多路平衡查找树**，它和B树的主要区别在于:

- B树中每个节点（叶子节点和非叶子节点）都存储真实数据。而B+树这种叶子节点存储值，非叶子节点存储键。
- **B树中一条记录只会出现一次，不会出现重复。而B+树的键可能出现重复。**
- B+树的叶子节点使用双向链表连接。

基于上述特点，B+树具有如下优势：

- **更少的IO次数：**B+树的非叶节点只包含键，而不包含真实数据，因此每个节点存储的记录个数比B数多很多（即阶m更大），因此B+树的高度更低，访问时所需要的IO次数更少。此外，由于每个节点存储的记录数更多，所以对访问局部性原理的利用更好，缓存命中率更高。
- **更适于范围查询：**在B树中进行范围查询时，首先找到要查找的下限，然后对**B树进行中序遍历**，直到找到查找的上限；而B+树的范围查询，只需要对**链表进行遍历**即可。
- **更稳定的查询效率：**B树的查询时间复杂度在1到树高之间(分别对应记录在根节点和叶节点)，而B+树的查询复杂度则稳定为树高，因为所有数据都在叶节点。

但是B+树也有其自身的缺点，因为键有可能出现重复，所以会占用更多的空间。但对于现代服务器对比性能来说，空间劣势基本都是可以接受的。

2.2.2.5.2 哈希索引

Hash索引在MySQL中使用的并不是很多，目前主要是Memory存储引擎使用，在Memory存储引擎中将Hash索引作为默认的索引类型。所谓Hash索引，实际上就是通过一定的Hash算法，将需要索引的键值进行Hash运算，然后将得到的Hash值存入一个Hash表中。然后每次需要检索的时候，都会将检索条件进行相同算法的Hash运算，然后再和Hash表中的Hash值进行比较并得出相应的信息。

特点：

- Hash索引仅仅只能满足"="，"IN"和"<=>"查询，不能使用范围查询；
- Hash索引无法被利用来避免数据的排序操作；
- Hash索引不能利用部分索引键查询；
- Hash索引在任何时候都不能避免表扫描；
- Hash索引遇到大量Hash值相等的情况后性能并不一定就会比B+Tree索引高；

2.2.2.6 索引失效情况（重点）

环境准备

```
CREATE TABLE staffs (
  id INT PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR (24)  NULL DEFAULT '' COMMENT '姓名',
  age INT NOT NULL DEFAULT 0 COMMENT '年龄',
  pos VARCHAR (20) NOT NULL DEFAULT '' COMMENT '职位',
  add_time TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '入职时间'
) CHARSET utf8 COMMENT '员工记录表' ;


INSERT INTO staffs(name,age,pos,add_time) VALUES('zhangsan',18,'manager',NOW());
INSERT INTO staffs(name,age,pos,add_time) VALUES('lisi',19,'dev',NOW());
INSERT INTO staffs(name,age,pos,add_time) VALUES('wangwu',20,'dev',NOW());

SELECT * FROM staffs;

ALTER TABLE staffs ADD INDEX idx_staffs_nameAgePos(name, age, pos);
```

1. 全值匹配 (索引idx_staffs_nameAgePos 建立索引时以 name，age，pos 的顺序建立的。全值匹配表示 **按顺序匹配的**

```
EXPLAIN SELECT * FROM staffs WHERE name = 'July';
EXPLAIN SELECT * FROM staffs WHERE name = 'July' AND age = 25;
EXPLAIN SELECT * FROM staffs WHERE age = 25 AND name = 'July' AND pos = 'dev';

EXPLAIN SELECT * FROM staffs WHERE age = 25;
EXPLAIN SELECT add_time FROM staffs WHERE age = 25 AND pos = 'dev';
```

2. 最左前缀法则(如果索引了多列，要遵守**最左前缀法则**。指的是查询从索引的最左前列开始并且不跳过索引中的列。)

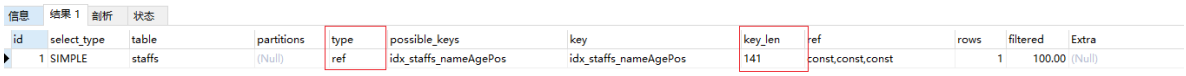
```
-- 【注意】
-- and 忽略左右关系。既即使没有没有按顺序 由于优化器的存在，会自动优化。
-- 除开上述条件 才满足最左前缀法则。
EXPLAIN SELECT * FROM staffs WHERE age = 25 AND pos = 'dev'; -- 索引失效
EXPLAIN SELECT * FROM staffs WHERE pos = 'dev';-- 索引失效
```

3. 不在索引列上做任何操作（计算、函数、(自动or手动)类型转换），如果做的话，会导致索引失效而转向**全表扫描**

```
EXPLAIN SELECT * FROM staffs WHERE left(NAME,4) = 'July';
```

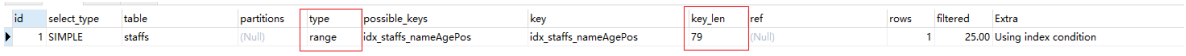
4. 存储引擎不能使用索引中范围条件(**bettween、<、>、in等**)右边的列(范围条件右边与范围条件使用的同一个组合索引，右边的才会失效。若是不同索引则不会失效)。

```
EXPLAIN SELECT * FROM staffs WHERE  name = 'July' AND age = 25 AND pos = 'dev';
```



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	staffs	(Null)	ref	idx_staffs_nameAgePos	idx_staffs_nameAgePos	141	const,const,const	1	100.00	(Null)

```
EXPLAIN SELECT * FROM staffs WHERE  name = 'July' AND age > 25 AND pos = 'dev';
-- 索引失效
```



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	staffs	(Null)	range	idx_staffs_nameAgePos	idx_staffs_nameAgePos	79	(Null)	1	25.00	Using index condition

5. 减少select *，使用哪些字段查哪些字段。
6. mysql5.7 在使用不等于(!= 或者<>)的时候无法使用索引会导致全表扫描。但8.0不会。

```
EXPLAIN SELECT * FROM staffs WHERE  name = 'July';
EXPLAIN SELECT * FROM staffs WHERE  name <> 'July';
EXPLAIN SELECT * FROM staffs WHERE  name != 'July';
```

7. mysql5.7 is not null 也无法使用索引，但是is null是可以使用索引的。但8.0不会

```
EXPLAIN SELECT * FROM staffs WHERE  NAME IS NOT NULL; -- 索引失效
EXPLAIN SELECT * FROM staffs WHERE  NAME IS NULL;
```

8. like以%开头('%abc...')mysql索引失效会变成全表扫描的操作

```
EXPLAIN SELECT * FROM staffs WHERE  NAME LIKE '%J'; -- ALL(索引失效)
EXPLAIN SELECT * FROM staffs WHERE  NAME LIKE 'J%'; -- range
```

9. 字符串不加单引号索引失效 (底层进行转换使索引失效，使用了函数造成索引失效) (**隐式类型转换**)

```
EXPLAIN SELECT * FROM staffs WHERE  NAME = 987
```

一般在开发中，当要进行调优时，需要有一定的依赖信息，可以通过`show status like 'Handler_read%';`查看索引的使用情况。

Variable_name	Value
Handler_read_first	3
Handler_read_key	12352
Handler_read_last	0
Handler_read_next	18
Handler_read_prev	0
Handler_read_rnd	4
Handler_read_rnd_next	25852

handler_read_key：这个值越大说明使用索引查询到的次数越多。

handler_read_rnd_next：这个值越高，说明查询低效。

3 存储优化

MySQL中索引是在存储引擎层实现的，这里我们会讲解存储引擎。

执行查询引擎的命令**show engines**，可以看到MySQL支持的存储引擎结果如下：

```
1  show engines
```

信息 | 结果1 | 概况 | 状态

Engine	Support	Comment	Transactions	XA	Savepoints
FEDERATED	NO	Federated MySQL storag	(Null)	(Null)	(Null)
MRG_MYISAM	YES	Collection of identical My	NO	NO	NO
MyISAM	YES	MyISAM storage engine	NO	NO	NO
BLACKHOLE	YES	/dev/null storage engine	NO	NO	NO
CSV	YES	CSV storage engine	NO	NO	NO
MEMORY	YES	Hash based, stored in me	NO	NO	NO
ARCHIVE	YES	Archive storage engine	NO	NO	NO
InnoDB	DEFAULT	Supports transactions, ro	YES	YES	YES
PERFORMANCE_SCHEMA	YES	Performance Schema	NO	NO	NO

mysql支持存储引擎有好几种，咱们这里主要讨论一下常用的Innodb，MyISAM存储引擎。

3.1 存储引擎介绍

3.1.1 InnoDB存储引擎

特点：

1. **InnoDB存储引擎提供了具有提交、回滚和崩溃恢复能力的事务安全。**相比较MyISAM存储引擎，InnoDB写的处理效率差一点并且会占用更多的磁盘空间保留数据和索引。
2. 提供了对数据库事务ACID（原子性Atomicity、一致性Consistency、隔离性Isolation、持久性Durability）的支持，实现了SQL标准的四种隔离级别。
3. 设计目标就是处理大容量的数据库系统，MySQL运行时InnoDB会在内存中建立缓冲池，用于缓冲数据和索引。
4. 执行“select count(*) from table”语句时需要扫描全表，因为使用innodb引擎的表不会保存表的具体行数，所以需要扫描整个表才能计算多少行。
5. **InnoDB引擎是行锁，粒度更小，所以写操作不会锁定全表，在并发较高时，使用InnoDB会提升效率。即存在大量UPDATE/INSERT操作时，效率较高。**

使用场景：

1. 经常UPDETE/INSERT的表，使用处理多并发的写请求
2. **支持事务，只能选出InnoDB。**
3. 可以从灾难中恢复（日志+事务回滚）
4. 外键约束、列属性AUTO_INCREMENT支持

3.1.2 MyISAM存储引擎

特点：

1. **MyISAM不支持事务，不支持外键，SELECT/INSERT为主的应用可以使用该引擎。**
2. 每个MyISAM在存储成3个文件，扩展名分别是：

1) frm：存储表定义（表结构等信息）

2) MYD(MYData)，存储数据

3) MYI(MYIndex)，存储索引

3. 不同MyISAM表的索引文件和数据文件可以放置到不同的路径下。
4. MyISAM类型的表提供修复的工具，可以用CHECK TABLE语句来检查MyISAM表健康，并用REPAIR TABLE语句修复一个损坏的MyISAM表。
5. 在MySQL5.6以前，只有MyISAM支持Full-text全文索引

使用场景：

1. 经常SELECT的表，插入不频繁，查询非常频繁。
2. 不支持事务。
3. 做很多count 的计算。

3.1.3 MyISAM和Innodb区别

InnoDB和MyISAM是许多人在使用MySQL时最常用的两个存储引擎，这两个存储引擎各有优劣，视具体应用而定。基本的差别为：MyISAM类型不支持事务处理，而InnoDB类型支持。MyISAM类型强调的是性能，其执行速度比InnoDB类型更快，而InnoDB提供事务支持已经外部键等高级数据库功能。

具体实现的差别：

- MyISAM是非事务安全型的，而InnoDB是事务安全型的。
- MyISAM锁的粒度是表级，而InnoDB支持行级锁定。
- MyISAM不支持外键，而InnoDB支持外键
- MyISAM相对简单，所以在效率上要优于InnoDB，小型应用可以考虑使用MyISAM。
- InnoDB表比MyISAM表更安全。

##3.2 存储优化

###3.2.1 禁用索引

对于使用索引的表，插入记录时，MySQL会对插入的记录建立索引。如果插入大量数据，建立索引会降低插入数据速度。为了解决这个问题，==可以在批量插入数据之前禁用索引，数据插入完成后再开启索引。==

禁用索引的语句：

```
ALTER TABLE table_name DISABLE KEYS
```

开启索引语句：

```
ALTER TABLE table_name ENABLE KEYS
```

MyISAM对于空表批量插入数据，则不需要进行操作，因为MyISAM引擎的表是在导入数据后才建立索引。

###3.2.2 禁用唯一性检查

唯一性校验会降低插入记录的速度，==可以在插入记录之前禁用唯一性检查，插入数据完成后再开启。(保证插入的数据没有重复的)==

禁用唯一性检查的语句：SET UNIQUE_CHECKS = 0;
开启唯一性检查的语句：SET UNIQUE_CHECKS = 1;

###3.2.3 禁用外键检查

==插入数据之前执行禁止对外键的检查，数据插入完成后再恢复，可以提供插入速度。==

禁用：SET foreign_key_checks = 0;
开启：SET foreign_key_checks = 1;

###3.2.4 禁止自动提交

插入数据之前执行禁止事务的自动提交，数据插入完成后再恢复，可以提高插入速度。

禁用：SET autocommit = 0;
开启：SET autocommit = 1;

4 数据库结构优化

4.1 优化表结构

- 尽量将表字段定义为NOT NULL约束，这时由于在MySQL中含有空值的列很难进行查询优化，NULL值会使索引以及索引的统计信息变得很复杂。
- 对于只包含特定类型的字段，可以使用enum、set 等数据类型。
- 数值型字段的比较比字符串的比较效率高得多，字段类型尽量使用最小、最简单的数据类型。例如IP地址可以使用int类型。
- 尽量使用TINYINT（4）、SMALLINT（6）、MEDIUM_INT（8）作为整数类型而非INT，如果非负则加上UNSIGNED。
- VARCHAR的长度只分配真正需要的空间
- 尽量使用TIMESTAMP而非DATETIME，但TIMESTAMP只能表示1970 - 2038年，比DATETIME表示的范围小得多，而且TIMESTAMP的值因时区不同而不同。
- 单表不要有太多字段，建议在20以内
- 合理的加入冗余字段可以提高查询速度。

4.2 表拆分

###4.2.1 垂直拆分

垂直拆分**按照字段进行拆分**，其实就是把组成一行的多个列分开放到不同的表中，这些表具有不同的结构，拆分后的表具有更少的列。例如用户表中的一些字段可能经常访问，可以把这些字段放进一张表里。另外一些不经常使用的信息就可以放进另外一张表里。

插入的时候使用事务，也可以保证两表的数据一致。缺点也很明显，由于拆分出来的两张表存在一对一的关系，需要使用冗余字段，而且需要join操作。但是我们可以在使用的时候可以分别取两次，这样的来说既可以避免join操作，又可以提高效率。

###4.2.2 水平拆分

水平拆分**按照行进行拆分**，常见的就是==分库分表==。以用户表为例，可以取用户ID，然后对ID取10的余数，将用户均匀的分配进这 0-9这10个表中。查找的时候也按照这种规则，又快又方便。

有些表业务关联比较强，那么可以使用按时间划分的。例如每天的数据量很大，需要每天新建一张表。这种业务类型就是需要高速插入，但是对于查询的效率不太关心。表越大，插入数据所需要索引维护的时间也就越长。

4.3 写分离

大型网站会有大量的并发访问，如果还是传统的数据存储方案，只是靠一台服务器处理，如此多的数据库连接、读写操作，数据库必然会崩溃，数据丢失的话，后果更是不堪设想。这时候，我们需要考虑如何降低单台服务器的使用压力，提升整个数据库服务的承载能力。

我们发现一般情况对数据库而言都是“读多写少”，也就说对数据库读取数据的压力比较大，这样分析可以采用数据库集群的方案。其中一个是主库，负责写入数据，我们称为写库；其它都是从库，负责读取数据，我们称为读库。这样可以缓解一台服务器的访问压力。

MySql自带主从复制功能，我们可以使用主从复制的主库作为写库，从库和主库进行数据同步，那么可以使用多个从库作为读库，已完成读写分离的效果。

##4.4 数据库集群

如果访问量非常大，虽然使用读写分离能够缓解压力，但是一旦写操作一台服务器都不能承受了，这个时候我们就需要考虑使用多台服务器实现写操作。

例如可以使用MyCat搭建MySql集群，对ID求3的余数，这样可以把数据分别存放到3台不同的服务器上，由MyCat负责维护集群节点的使用。

5 硬件优化

服务器硬件的性能瓶颈，直接决定MySQL数据库的运行速度和效率。可以从以下几个方面考虑：

##5.1 内存

足够大的内存，是提高MySQL数据库性能的方法之一。内存的IO比硬盘快的多，可以增加系统的缓冲区容量，使数据在内存停留的时间更长，以减少磁盘的IO。服务器内存建议不要小于2GB，推荐使用4GB以上的物理内存。

##5.2 磁盘

MySQL每秒钟都在进行大量、复杂的查询操作，对磁盘的读写量可想而知。所以，通常认为磁盘I/O是制约MySQL性能的最大因素之一，对于日均访问量在100万PV以上的系统，由于磁盘I/O的制约，MySQL的性能会非常低下 考虑以下几种解决方案：

- 使用SSD或者PCIe SSD设备，至少获得数百倍甚至万倍的IOPS提升；
- 购置阵列卡，可明显提升IOPS
- 尽可能选用RAID-10，而非RAID-5
- 使用机械盘的话，尽可能选择高转速的，例如选用15000RPM，而不是7200RPM的盘

##5.3 CPU

CPU仅仅只能决定运算速度，及时是运算速度都还取决于与内存之间的总线带宽以及内存本身的速度。但是一般情况下，我们都需要选择计算速度较快的CPU。

关闭节能模式。操作系统和CPU硬件配合，系统不繁忙的时候，为了节约电能和降低温度，它会将CPU降频。这对环保人士和抵制地球变暖来说是一个福音，但是对MySQL来说，可能是一个灾难。为了保证MySQL能够充分利用CPU的资源，建议设置CPU为最大性能模式。

5.4 网络

应该尽可能选择网络延时低，吞吐量高的设备。

- 网络延时：不同的网络设备其延时会有差异，延时自然是越小越好。
- 吞吐量：对于数据库集群来说，各个节点之间的网络吞吐量可能直接决定集群的处理能力。

6 缓存优化

##6.1 查询缓存

query_cache_size：作用于整个 MySQL，主要用来缓存MySQL中的ResultSet，也就是一条SQL语句执行的结果集，所以仅仅只能针对select语句。查询缓存从MySQL 5.7.20开始已被弃用，并在MySQL 8.0中被删除。

当我们打开了 Query Cache功能，MySQL在接受到一条select语句的请求后，如果该语句满足Query Cache的要求，MySQL会直接根据预先设定好的HASH算法将接受到的select语句以字符串方式进行hash，然后到Query Cache中直接查找是否已经缓存。如果已经在缓存中，该select请求就会直接将数据返回，从而省略了后面所有的步骤(如SQL语句的解析，优化器优化以及向存储引擎请求数据等)，极大的提高性能。

当然，Query Cache也有一个致命的缺陷，那就是当某个表的数据有任何任何变化，都会导致所有引用了该表的select语句在Query Cache中的缓存数据失效。所以，==当我们的数据变化非常频繁的情况下，使用Query Cache可能会得不偿失。==

如果缓存命中率非常高的话，有测试表明在极端情况下可以提高效率238%，而在糟糕时，QC会降低系统13%的处理能力。

通过以下命令查看缓存相关变量

```
show variables like '%query_cache%';
```

- have_query_cache：表示此版本mysql是否支持缓存
- query_cache_limit ： 缓存最大值
- query_cache_size：缓存大小
- query_cache_type：off 表示不缓存，on表示缓存所有结果。

6.2 全局缓存

数据库属于IO密集型的应用程序，其主职责就是数据的管理及存储工作。而我们知道，从内存中读取一个数据库的时间是微秒级别，而从一块普通硬盘上读取一个 IO是在毫秒级别，二者相差3个数量级。所以，要优化数据库，首先第一步需要优化的就是IO，尽可能将**磁盘IO转化为内存IO**,也就是使用缓存

启动MySQL时就要分配并且总是存在的全局缓存，可以在MySQL的my.conf或者my.ini文件的[mysqld]组中配置。查询缓存属于全局缓存。

目前有：

key_buffer_size(默认值：402653184,即384M)、

innodb_buffer_pool_size(默认值：134217728即：128M)、

innodb_additional_mem_pool_size（默认值：8388608即：8M）、

innodb_log_buffer_size(默认值：8388608即：8M)、

query_cache_size(默认值：33554432即：32M)

- **key_buffer_size**

用于索引块的缓冲区大小，增加它可得到更好处理的索引(对所有读和多重写)，对MyISAM表性能影响最大的一个参数。如果你使它太大，系统将开始换页并且真的变慢了。

严格说是它决定了数据库索引处理的速度，尤其是索引读的速度。对于内存在4GB左右的服务器该参数可设置为256M或384M.

- **innodb_buffer_pool_size**

主要针对InnoDB表性能影响最大的一个参数。功能与Key_buffer_size一样。InnoDB占用的内存，除innodb_buffer_pool_size用于存储页面缓存数据外，另外正常情况下还有大约8%的开销，主要用在每个缓存页帧的描述、adaptive hash等数据结构，如果不是安全关闭，启动时还要恢复的话，还要另开大约12%的内存用于恢复，两者相加就有差不多21%的开销。

- **innodb_additional_mem_pool_size**

设置了InnoDB存储引擎用来存放数据字典信息以及一些内部数据结构的内存空间大小，所以当我们一个MySQL Instance中的数据库对象非常多的时候，是需要适当调整该参数的大小以确保所有数据都能存放在内存中提高访问效率的。

- **innodb_log_buffer_size**

这是InnoDB存储引擎的事务日志所使用的缓冲区。类似于Binlog Buffer。InnoDB在写事务日志的时候，为了提高性能，也是先将信息写入Innofb Log Buffer中，当满足innodb_flush_log_trx_commit参数所设置的相应条件(或者日志缓冲区写满)之后，才会将日志写到文件(或者同步到磁盘)中。可以通过innodb_log_buffer_size 参数设置其可以使用的最大内存空间。

InnoDB 将日志写入日志磁盘文件前的缓冲大小。理想值为 1M 至 8M。大的日志缓冲允许事务运行时不需要将日志保存入磁盘而只到事务被提交(commit)。因此，如果有大的事务处理，设置大的日志缓冲可以减少磁盘I/O。这个参数实际上还和另外的flush参数相关。一般来说不建议超过32MB。

6.3 局部缓存

除了全局缓冲，**MySql还会为每个连接发放连接缓冲**。个连接到MySQL服务器的线程都需要有自己的缓冲。大概需要立刻分配256K，甚至在线程空闲时，它们使用默认的线程堆栈，网络缓存等。事务开始之后，则需要增加更多的空间。运行较小的查询可能仅给指定的线程增加少量的内存消耗，然而如果对数据表做复杂的操作例如扫描、排序或者需要临时表，则需分配大约read_buffer_size，

sort_buffer_size，read_rnd_buffer_size，tmp_table_size大小的内存空间. 不过它们只是在需要的时候才分配，并且在那些操作做完之后就释放了。

- **read_buffer_size**

是MySql读入缓冲区大小。对表进行顺序扫描的请求将分配一个读入缓冲区，MySql会为它分配一段内存缓冲区。read_buffer_size变量控制这一缓冲区的大小。如果对表的顺序扫描请求非常频繁，并且你认为频繁扫描进行得太慢，可以通过增加该变量值以及内存缓冲区大小提高其性能.

- **sort_buffer_size**

是MySql执行排序使用的缓冲大小。如果想要增加ORDER BY的速度，首先看是否可以让MySQL使用索引而不是额外的排序阶段。如果不能，可以尝试增加sort_buffer_size变量的大小

- **read_rnd_buffer_size**

是MySql的随机读缓冲区大小。当按任意顺序读取行时(例如，按照排序顺序)，将分配一个随机读缓存区。进行排序查询时，MySql会首先扫描一遍该缓冲，以避免磁盘搜索，提高查询速度，如果需要排序大量数据，可适当调高该值。但MySql会为每个客户连接发放该缓冲空间，所以应尽量适当设置该值，以避免内存开销过大。

- **tmp_table_size**

是MySql的heap （堆积）表缓冲大小。所有联合在一个DML指令内完成，并且大多数联合甚至可以不用临时表即可以完成。大多数临时表是基于内存的(HEAP)表。具有大的记录长度的临时表 (所有列的长度的和)或包含BLOB列的表存储在硬盘上。

如果某个内部heap（堆积）表大小超过tmp_table_size，MySQL可以根据需要自动将内存中的heap表改为基于硬盘的MyISAM表。还可以通过设置tmp_table_size选项来增加临时表的大小。也就是说，如果调高该值，MySql同时将增加heap表的大小，可达到提高联接查询速度的效果。

- **record_buffer**

record_buffer每个进行一个顺序扫描的线程为其扫描的每张表分配这个大小的一个缓冲区。如果你做很多顺序扫描，你可能想要增加该值。

6.4 其它缓存

- **table_cache**

TABLE_CACHE(5.1.3及以后版本又名TABLE_OPEN_CACHE)，table_cache指定表高速缓存的大小。每当MySQL访问一个表时，如果在表缓冲区中还有空间，该表就被打开并放入其中，这样可以更快地访问表内容。

不能盲目地把table_cache设置成很大的值。如果设置得太高，可能会造成文件描述符不足，从而造成性能不稳定或者连接失败。

- **thread_cache_size**

服务器线程缓存，默认的thread_cache_size=8，,这个值表示可以重新利用保存在缓存中线程的数量,当断开连接时如果缓存中还有空间,那么客户端的线程将被放到缓存中,如果线程重新被请求，那么请求将从缓存中读取,如果缓存中是空的或者是新的请求，那么这个线程将被重新创建,如果有很多新的线程，

增加这个值可以改善系统性能.通过比较Connections 和 Threads_created 状态的变量，可以看到这个变量的作用。

7 MySQL服务器优化

##7.1 MySQL参数

通过优化MySQL的参数可以提高资源利用率，从而达到提高MySQL服务器性能的目的。MySQL的配置参数都在my.conf或者my.ini文件的[mysqld]组中，常用的参数如下：

- **back_log**

在MySQL暂时停止回答新请求之前的短时间内多少个请求可以被存在堆栈中（每个连接256kb，占用：125M）。也就是说，如果MySql的连接数据达到max_connections时，新来的请求将会被存在堆栈中，以等待某一连接释放资源，该堆栈的数量即back_log，如果等待连接的数量超过back_log，将不被授予连接资源。

- **wait_timeout**

当MySQL连接闲置，超过一定时间后将会被强行关闭。MySQL默认的wait-timeout值为8个小时。

设置这个值是非常有意义的，比如你的网站有大量的MySQL链接请求（每个MySQL连接都是要内存资源开销的），由于你的程序的原因有大量的连接请求空闲啥事也不干，白白占用内存资源，或者导致MySQL超过最大连接数从来无法新建连接导致“Too many connections”的错误。在设置之前你可以查看一下你的MYSQL的状态（可用showprocesslist)，如果经常发现MYSQL中有大量的Sleep进程，则需要修改wait-timeout值了。

- **max_connections**

是指MySql的最大连接数，如果服务器的并发连接请求量比较大，建议调高此值，以增加并行连接数量，当然这建立在机器能支撑的情况下，因为如果连接数越多，介于MySql会为每个连接提供连接缓冲区，就会开销越多的内存，所以要适当调整该值，不能盲目提高设值。

MySQL服务器允许的最大连接数16384

- **max_user_connections**

是指每个数据库用户的最大连接针对某一个账号的所有客户端并行连接到MYSQL服务的最大并行连接数。简单说是指同一个账号能够同时连接到mysql服务的最大连接数。设置为0表示不限制。

- **thread_concurrency**

的值的正确与否, 对mysql的性能影响很大, 在多个cpu(或多核)的情况下，错误设置了thread_concurrency的值, 会导致mysql不能充分利用多cpu(或多核), 出现同一时刻只能一个cpu(或核)在工作的情况。**thread_concurrency应设为CPU核数的2倍。**

- **skip-name-resolve**

禁止MySQL对外部连接进行DNS解析，使用这一选项可以消除MySQL进行DNS解析的时间。但需要注意，如果开启该选项，则所有远程主机连接授权都要使用IP地址方式，否则MySQL将无法正常处理连接请求!

- **default-storage-engine**

default-storage-engine=InnoDB(设置InnoDB类型，另外还可以设置MyISAM类型)设置创建数据库及表默认存储类型

7.2 Linux系统优化

一般情况，我们都会使用Linux来进行MySQL的安装和部署，Linux系统在使用的时候，也需要进行相关的配置，以提高MySQL的使用性能，这里列举以下几点:

- 避免使用Swap交换分区，因为交换时是从硬盘读取的，速度很慢。
- 将操作系统和数据分区分开，不仅仅是逻辑上，还包括物理上，因为操作系统的读写会影响数据库的性能。
- 把MySQL临时空间和复制日志与数据放到不同的分区，数据库后台从磁盘进行读写时会影响数据库的性能。
- 避免使用软件磁盘阵列，使用硬件磁盘阵列。
- 在Linux中设置swappiness的值为0，因为在数据库服务器中不需要缓存文件。
- 使用 noatime 和 nodirtime 挂载文件系统，因为不需要对数据库文件修改时间。
- 使用 XFS 文件系统，一种比ext3更快、更小的文件系统。
- 调整 XFS 文件系统日志和缓冲变量 – 为了最高性能标准。
- 使用64位的操作系统，这会支持更大的内存。
- 删除服务器上未使用的安装包和守护进程，节省系统的资源占用。
- 把使用MySQL的host和你的MySQL host放到一个hosts文件中。