

MySQL性能

最大数据量

抛开数据量和并发数，谈性能都是耍流氓。MySQL没有限制单表最大记录数，它取决于操作系统对文件大小的限制。

分类	数据库	特性
键值型	Memcache	用于内容缓存，大量数据的高访问负载
键值型	Redis	用于内容缓存，比Memcache支持更多的数据类型，并能持久化数据
列式存储	HBase	Hadoop体系的核心数据库，海量结构化数据存储，大数据必备。
文档型	MongoDb	知名文档型数据库，也可以用于缓存
文档型	CouchDB	Apache的开源项目，专注于易用性，支持REST API
文档型	SequoiaDB	国内知名文档型数据库
图形	Neo4J	用于社交网络构建关系图谱，推荐系统等  Java之间

《阿里巴巴Java开发手册》提出单表行数超过500万行或者单表容量超过2GB，才推荐分库分表。性能由综合因素决定，抛开业务复杂度，影响程度依次是硬件配置、MySQL配置、数据表设计、索引优化。500万这个值仅供参考，并非铁律。

博主曾经操作过超过4亿行数据的单表，分页查询最新的20条记录耗时0.6秒，SQL语句大致是

分类	数据库	特性
键值型	Memcache	用于内容缓存，大量数据的高访问负载
键值型	Redis	用于内容缓存，比Memcache支持更多的数据类型，并能持久化数据
列式存储	HBase	Hadoop体系的核心数据库，海量结构化数据存储，大数据必备。
文档型	MongoDb	知名文档型数据库，也可以用于缓存
文档型	CouchDB	Apache的开源项目，专注于易用性，支持REST API
文档型	SequoiaDB	国内知名文档型数据库
图形	Neo4J	用于社交网络构建关系图谱，推荐系统等  Java之间

prePageMinId是上一页数据记录的最小ID。

虽然当时查询速度还凑合，随着数据不断增长，有朝一日必定不堪重负。分库分表是个周期长而风险高的大活儿，应该尽可能在当前结构上优化，比如升级硬件、迁移历史数据等等，实在没辙了再分。对分库分表感兴趣的同学可以阅读分库分表的基本思想。

最大并发数

并发数是指同一时刻数据库能处理多少个请求，由max_connections和max_user_connections决定。max_connections是指MySQL实例的最大连接数，上限值是16384，max_user_connections是指每个数据库用户的最大连接数。

MySQL会为每个连接提供缓冲区，意味着消耗更多的内存。如果连接数设置太高硬件吃不消，太低又不能充分利用硬件。一般要求两者比值超过10%，计算方法如下：

分类	数据库	特性
键值型	Memcache	用于内容缓存，大量数据的高访问负载
键值型	Redis	用于内容缓存，比Memcache支持更多的数据类型，并能持久化数据
列式存储	HBase	Hadoop体系的核心数据库，海量结构化数据存储，大数据必备。
文档型	MongoDb	知名文档型数据库，也可以用于缓存
文档型	CouchDB	Apache的开源项目，专注于易用性，支持REST API
文档型	SequoiaDB	国内知名文档型数据库
图形	Neo4J	用于社交网络构建关系图谱，推荐系统等  Java之间

查看最大连接数与响应最大连接数：

```
1 show variables like '%max_connections%';
2 show variables like '%max_user_connections%';
```

在配置文件my.cnf中修改最大连接数

```
1 [mysqld]
2 max_connections = 100
3 max_used_connections = 20
```

查询耗时0.5秒

建议将单次查询耗时控制在0.5秒以内，0.5秒是个经验值，源于用户体验的3秒原则。如果用户的操作3秒内没有响应，将会厌烦甚至退出。响应时间=客户端UI渲染耗时+网络请求耗时+应用程序处理耗时+查询数据库耗时，0.5秒就是留给数据库1/6的处理时间。

实施原则

相比NoSQL数据库，MySQL是个娇气脆弱的家伙。它就像体育课上的女同学，一点纠纷就和同学闹别扭(扩容难)，跑两步就气喘吁吁(容量小并发低)，常常身体不适要请假(SQL约束太多)。如今大家都会搞点分布式，应用程序扩容比数据库要容易得多，所以实施原则是数据库少干活，应用程序多干活。

- 充分利用但不滥用索引，须知索引也消耗磁盘和CPU。

- 不推荐使用数据库函数格式化数据，交给应用程序处理。
- 不推荐使用外键约束，用应用程序保证数据准确性。
- 写多读少的场景，不推荐使用唯一索引，用应用程序保证唯一性。
- 适当冗余字段，尝试创建中间表，用应用程序计算中间结果，用空间换时间。
- 不允许执行极度耗时的事务，配合应用程序拆分成更小的事务。
- 预估重要数据表（比如订单表）的负载和数据增长态势，提前优化。

数据表设计

数据类型

数据类型的选择原则：更简单或者占用空间更小。

- 如果长度能够满足，整型尽量使用tinyint、smallint、medium_int而非int。
- 如果字符串长度确定，采用char类型。
- 如果varchar能够满足，不采用text类型。
- 精度要求较高的使用decimal类型，也可以使用BIGINT，比如精确两位小数就乘以100后保存。
- 尽量采用timestamp而非datetime。

分类	数据库	特性
键值型	Memcache	用于内容缓存，大量数据的高访问负载
键值型	Redis	用于内容缓存，比Memcache支持更多的数据类型，并能持久化数据
列式存储	HBase	Hadoop体系的核心数据库，海量结构化数据存储，大数据必备。
文档型	MongoDB	知名文档型数据库，也可以用于缓存
文档型	CouchDB	Apache的开源项目，专注于易用性，支持REST API
文档型	SequoiaDB	国内知名文档型数据库
图形	Neo4J	用于社交网络构建关系图谱，推荐系统等

相比datetime，timestamp占用更少的空间，以UTC的格式储存自动转换时区。

避免空值

MySQL中字段为NULL时依然占用空间，会使索引、索引统计更加复杂。从NULL值更新到非NULL无法做到原地更新，容易发生索引分裂影响性能。尽可能将NULL值用有意义的值代替，也能避免SQL语句里面包含is not null的判断。

text类型优化

由于text字段储存大量数据，表容量会很早涨上去，影响其他字段的查询性能。建议抽取出来放在子表里，用业务主键关联。

索引优化

索引分类

- 普通索引：最基本的索引。

- 空间索引:
- 唯一索引: 与普通索引类似, 但索引列的值必须唯一, 允许有空值。
- 主键索引: 特殊的唯一索引, 用于唯一标识数据表中的某一条记录, 不允许有空值, 一般用 primary key 约束。
- 全文索引: 用于海量文本的查询, MySQL 5.6 之后的 InnoDB 和 MyISAM 均支持全文索引。由于查询精度以及扩展性不佳, 更多的企业选择 Elasticsearch。

索引优化

- 分页查询很重要, 如果查询数据量超过 30%, MySQL 不会使用索引。
- 单表索引数不超过 5 个、单个索引字段数不超过 5 个。
- 字符串可使用前缀索引, 前缀长度控制在 5-8 个字符。
- 字段唯一性太低, 增加索引没有意义, 如: 是否删除、性别。

合理使用覆盖索引, 如下所示:

```
1 select login_name, nick_name from member where login_name = ?
```

login_name, nick_name 两个字段建立组合索引, 比 login_name 简单索引要更快。

SQL 优化

分批处理

博主小时候看到鱼塘挖开小口子放水, 水面有各种漂浮物。浮萍和树叶总能顺利通过出水口, 而树枝会挡住其他物体通过, 有时还会卡住, 需要人工清理。MySQL 就是鱼塘, 最大并发数和网络带宽就是出水口, 用户 SQL 就是漂浮物。

不带分页参数的查询或者影响大量数据的 update 和 delete 操作, 都是树枝, 我们要把它打散分批处理, 举例说明:

业务描述: 更新用户所有已过期的优惠券为不可用状态。

SQL 语句:

```
1 update status=0 FROM `coupon` WHERE expire_date <= #{currentDate} and status=1;
```

如果大量优惠券需要更新为不可用状态, 执行这条 SQL 可能会堵死其他 SQL, 分批处理伪代码如下:

```
1 int pageNo = 1;
2 int PAGE_SIZE = 100;
3 while(true) {
4     List<Integer> batchIdList = queryList
5         ('select id FROM `coupon` WHERE expire_date <= #{currentDate}
6          and status = 1 limit #{(pageNo-1) * PAGE_SIZE},#{PAGE_SIZE}');
7     if (CollectionUtils.isEmpty(batchIdList)) {
8         return;
9     }
10    update('update status = 0 FROM `coupon` where status = 1 and id in #{batchIdList}')
11    pageNo ++;
12 }
```

 Java 之间

操作符 <> 优化

通常 <> 操作符无法使用索引, 举例如下, 查询金额不为 100 元的订单:

```
1 select id from orders where amount != 100;
```



如果金额为100的订单极少，这种数据分布严重不均的情况下，有可能使用索引。鉴于这种不确定性，采用union聚合搜索结果，改写方法如下：

```
1 (select id from orders where amount > 100)
```

```
2 union all
```

```
3 (select id from orders where amount < 100 and amount > 0)
```



OR优化

在Innodb引擎下or无法使用组合索引，比如：

```
1 select id, product_name from orders where mobile_no = '13421800407' or user_id = 100;
```

OR无法命中mobile_no + user_id的组合索引，可采用union，如下所示：

```
1 (select id, product_name from orders where mobile_no = '13421800407')
```

```
2 union
```

```
3 (select id, product_name from orders where user_id = 100);
```



此时id和product_name字段都有索引，查询才最高效。

IN优化

IN适合主表大子表小，EXIST适合主表小子表大。由于查询优化器的不断升级，很多场景这两者性能差不多一样了。

尝试改为join查询，举例如下：

```
1 select id from orders where user_id in (select id from user where level = 'VIP');
```

采用JOIN如下所示：

```
1 select o.id from orders o left join user u on o.user_id = u.id where u.level = 'VIP';
```

不做列运算

通常在查询条件列运算会导致索引失效，如下所示：

查询当日订单

```
1 select id from order where date_format(create_time, '%Y-%m-%d') = '2019-07-01';
```

date_format函数会导致这个查询无法使用索引，改写后：

```
1 select id from order where create_time between
```

```
2 '2019-07-01 00:00:00' and '2019-07-01 23:59:59';
```



避免Select all

如果不查询表中所有的列，避免使用SELECT *，它会进行全表扫描，不能有效利用索引。

Like优化

like用于模糊查询，举个例子（field已建立索引）：

```
1 SELECT column FROM table WHERE field like '%keyword%';
```



这个查询未命中索引，换成下面的写法：

```
1 SELECT column FROM table WHERE field like 'keyword%';
```



去除了前面的%查询将会命中索引，但是产品经理一定要前后模糊匹配呢？全文索引|fulltext可以尝试一下，但Elasticsearch才是终极武器。

Join优化


join的实现是采用Nested Loop Join算法，就是通过驱动表的结果集作为基础数据，通过该结果集作为过滤条件到下一个表中循环查询数据，然后合并结果。如果有多个join，则将前面的结果集作为循环数据，再次到后一个表中查询数据。

驱动表和被驱动表尽可能增加查询条件，满足ON的条件而少用Where，用小结果集驱动大结果集。被驱动表的join字段上加上索引，无法建立索引的时候，设置足够的Join Buffer Size。禁止join连接三个以上的表，尝试增加冗余字段。

Limit优化


limit用于分页查询时越往后翻性能越差，解决的原则：缩小扫描范围，如下所示：

```
1 select * from orders order by id desc limit 100000,10
2 耗时0.4秒
3 <
3 select * from orders order by id desc limit 1000000,10
4 耗时5.2秒
```



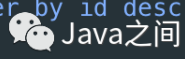
先筛选出ID缩小查询范围，写法如下：

```
1 select * from orders where id > (select id from orders order by id desc limit 1000000, 1)
2   order by id desc limit 0,10
3 耗时0.5秒
```



如果查询条件仅有主键ID，写法如下：

```
1 select id from orders where id between 1000000 and 1000010 order by id desc
2 耗时0.3秒
```



如果以上方案依然很慢呢？只好用游标了，感兴趣的朋友阅读JDBC使用游标实现分页查询的方法

其他数据库

作为一名后端开发人员，务必精通作为存储核心的MySQL或SQL Server，也要积极关注NoSQL数据库，他们已经足够成熟并被广泛采用，能解决特定场景下的性能瓶颈。

分类	数据库	特性
键值型	<u>Memcache</u>	用于内容缓存，大量数据的高访问负载
键值型	<u>Redis</u>	用于内容缓存，比Memcache支持更多的数据类型，并能持久化数据
列式存储	<u>HBase</u>	Hadoop体系的核心数据库，海量结构化数据存储，大数据必备。
文档型	<u>MongoDb</u>	知名文档型数据库，也可以用于缓存
文档型	<u>CouchDB</u>	Apache的开源项目，专注于易用性，支持REST API
文档型	<u>SequoiaDB</u>	国内知名文档型数据库
图形	<u>Neo4J</u>	用于社交网络构建关系图谱，推荐系统等