第2课 MySQL常见优化

内容

```
1.数据库优化原则
2.MySQL执行计划
```

一.优化原则

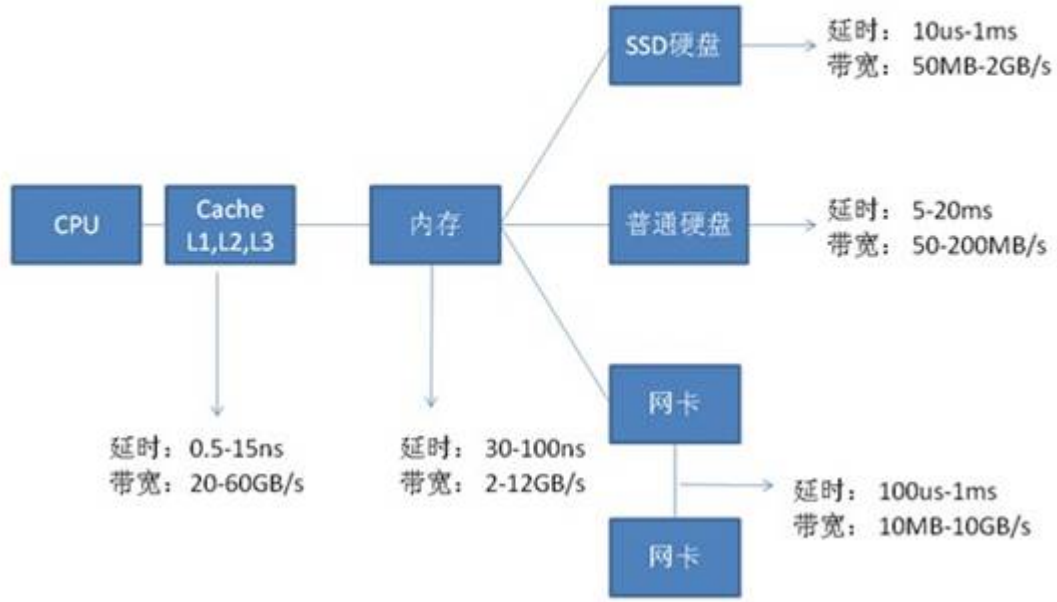
在网上有很多文章介绍数据库优化知识，但是大部份文章只是对某个一个方面进行说明，而对于我们程序员来说这种介绍并不能很好的掌握优化知识，因为很多介绍只是对一些特定的场景优化的，所以反而有时会产生误导或让程序员感觉不明白其中的奥妙而对数据库优化感觉很神秘。

很多程序员总是问如何学习数据库优化，有没有好的教材之类的问题。在书店也看到了许多数据库优化的专业书籍，但是感觉更多是面向DBA或者是PL/SQL开发方面的知识，个人感觉不太适合普通程序员。而要想做到数据库优化的高手，不是花几周，几个月就能达到的，这并不是因为数据库优化有多高深，而是因为要做好优化一方面需要有非常好的技术功底，对操作系统、存储硬件网络、数据库原理等方面有比较扎实的基础知识，另一方面是需要花大量时间对特定的数据库进行实践测试与总结。

作为一个程序员，我们也许不清楚线上正式的服务器硬件配置，我们不可能像DBA那样专业的对数据库进行各种实践测试与总结，但我们都应该非常了解我们SQL的业务逻辑，我们清楚SQL中访问表及字段的数据情况，我们其实只关心我们的SQL是否能尽快返回结果。那程序员如何利用已知的知识进行数据库优化？如何能快速定位SQL性能问题并找到正确的优化方向？

要正确的优化SQL，我们需要快速定位能性的瓶颈点，也就是说快速找到我们SQL主要的开销在哪里？而大多数情况性能最慢的设备会是瓶颈点，如下载时网络速度可能会是瓶颈点，本地复制文件时硬盘可能会是瓶颈点，为什么这些一般的工作我们能快速确认瓶颈点呢，因为我们对这些慢速设备的性能数据有一些基本的认识，如网络带宽是2Mbps，硬盘是每分钟7200转等等。因此，为了快速找到SQL的性能瓶颈点，我们也需要了解我们计算机系统的硬件基本性能指标，下图展示的当前主流计算机性能指标数据。

IO各层次性能汇总

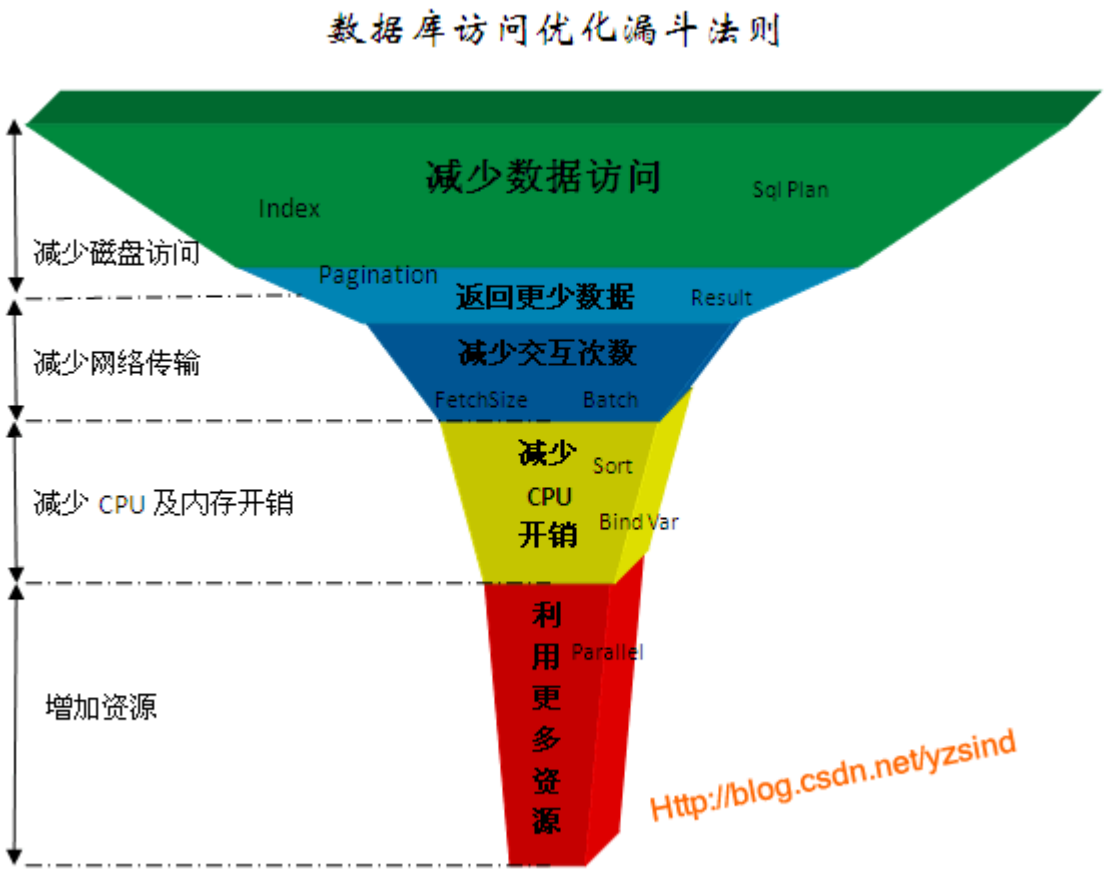


从图上可以看到基本上每种设备都有两个指标：
延时（响应时间）：表示硬件的突发处理能力；
带宽（吞吐量）：代表硬件持续处理能力。

从上图可以看出，计算机系统硬件性能从高到代依次为：
CPU—Cache(L1-L2-L3)—内存—SSD硬盘—网络—硬盘

由于SSD硬盘还处于快速发展阶段，所以这里暂不涉及SSD相关应用系统。
根据数据库知识，我们可以列出每种硬件主要的工作内容：
1.CPU及内存：缓存数据访问、比较、排序、事务检测、SQL解析、函数或逻辑运算；
2.网络：结果数据传输、SQL请求、远程数据库访问（dblink）；
3.硬盘：数据访问、数据写入、日志记录、大数据量排序、大表连接。

根据当前计算机硬件的基本性能指标及其在数据库中主要操作内容，可以整理出如下图所示的性能基本优化法则：



这个优化法则归纳为5个层次：

1、 减少数据访问（减少磁盘访问）
2、 返回更少数据（减少网络传输或磁盘访问）
3、 减少交互次数（减少网络传输）
4、 减少服务器CPU开销（减少CPU及内存开销）
5、 利用更多资源（增加资源）

由于每一层优化法则都是解决其对应硬件的性能问题，所以带来的性能提升比例也不一样。传统数据库系统设计是也是尽可能对低速设备提供优化方法，因此针对低速设备问题的可优化手段也更多，优化成本也更低。我们任何一个SQL的性能优化都应该按这个规则由上到下来诊断问题并提出解决方案，而不应该首先想到的是增加资源解决问题。

以下是每个优化法则层级对应优化效果及成本经验参考：

优化法则	性能提升效果	优化成本
减少数据访问	1~1000	低
返回更少数据	1~100	低
减少交互次数	1~20	低
减少服务器CPU开销	1~5	低
利用更多资源	@~10	高

1.减少数据访问

最常见的是使用索引，减少对数据的访问。

1.1 创建并使用正确的索引

数据库索引的原理非常简单，但在复杂的表中真正能正确使用索引的人很少，即使是专业的DBA也不一定能完全做到最优。

索引会大大增加表记录的DML(INSERT,UPDATE,DELETE)开销，正确的索引可以让性能提升100，1000倍以上，不合理的索引也可能会让性能下降100倍，因此在一个表中创建什么样的索引需要平衡各种业务需求。

但索引是快速搜索的关键，MySQL索引的建立对于MySQL的高效运行是很重要的。在数据库表中，对字段建立索引可以大大提高查询速度。

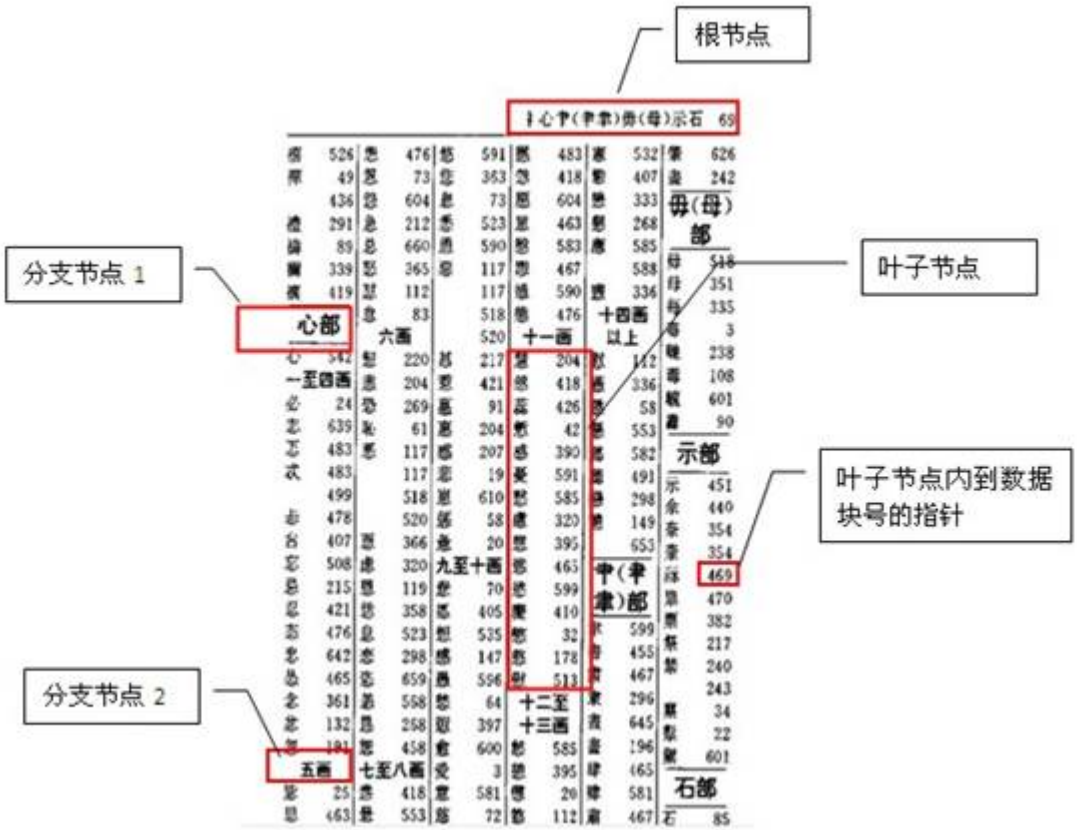
如我们创建一个USER表：

```
CREATE TABLE USER
(
    ID INT NOT NULL,
    username VARCHAR(16) NOT NULL
);
```

我们随机向里面插入了10000条记录，假如其中有一条记录：5555, admin。
如果进行以下查询时： `SELECT * FROM mytable WHERE username='admin';`
如果在username上已经建立了索引，MySQL无须任何扫描，即准确可找到该记录。
相反，MySQL会扫描所有记录，即要查询10000条记录。

1.2 索引认识

如果我们把一个表的内容认为是一本字典，那索引就相当于字典的目录，如下图所示：



图中是一个字典按部首+笔划数的目录，相当于给字典建了一个按部首+笔划的组合索引。
一个表中可以建多个索引，就如一本字典可以建多个目录一样（按拼音、笔划、部首等等）。
一个索引也可以由多个字段组成，称为组合索引，如上图就是一个按部首+笔划的组合目录。

1.3 MySQL常见索引

索引一：普通索引

这是最基本的索引，它没有任何限制。普通索引(由关键字KEY或INDEX定义的索引)的唯一任务是加快对数据的访问速度。因此，应该只为那些最经常出现在
查询条件(`WHERE column = `)或排序条件(`ORDER BY column`)中的数据列创建索引。
只要有可能，就应该选择一个数据最整齐、最紧凑的数据列(如一个整数类型的数据列)来创建索引。

创建索引方式：

```
#a.直接创建
CREATE INDEX 索引名 ON 表(表字段(length));
如果是CHAR，VARCHAR类型，length可以小于字段实际长度；
如果是BLOB和TEXT类型，必须指定length，下同。
```

```
#b.修改表结构创建
ALTER 表 ADD INDEX [索引名] ON (表字段(length))
```

```
#c.创建表的时候直接指定
CREATE TABLE 表
(
    ID INT NOT NULL,
    字段2 VARCHAR(16) NOT NULL,
    INDEX [索引名] (字段2(length))
);
```

删除索引的方式：

```
DROP INDEX [索引名] ON 表;
```

索引二：唯一索引（unique）

它与前面的普通索引类似，不同的就是：
普通索引允许被索引的数据列包含重复的值。而唯一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。

创建索引方式：

```
#a.直接创建
CREATE UNIQUE INDEX 索引名 ON 表(字段(length))
```

```
#b.修改表结构
ALTER 表 ADD UNIQUE [索引名] ON (字段(length))
```

```
#c.创建表的时候直接指定
CREATE TABLE 表
(
    ID INT NOT NULL,
    字段2 VARCHAR(16) NOT NULL,
    UNIQUE [索引名] (字段2(length))
);
```

索引三：主键索引

它是一种特殊的唯一索引，不允许有空值。一般是在建表的时候同时创建主键索引。

创建方式：

```
CREATE TABLE 表
(
    ID INT NOT NULL,
    字段2 VARCHAR(16) NOT NULL,
    PRIMARY KEY(ID)  #声明主键索引
);

#当然也可以用ALTER命令。记住：一个表只能有一个主键。
```

索引四：外键索引

如果为某个外键字段定义了一个外键约束条件，MySQL就会定义一个内部索引来帮助自己以最有效率的方式去管理和使用外键约束条件。

索引五：组合索引

以上是单列索引。为了形象地对比单列索引和组合索引，为表添加多个字段：

```
CREATE TABLE USER(
    ID INT NOT NULL,
    name VARCHAR(16) NOT NULL,
    city VARCHAR(50) NOT NULL,
    age INT NOT NULL
);
```

为了进一步榨取MySQL的效率，就要考虑建立组合索引。
可以将 name,city,age 建到一个索引里。
则组合索引如下：

```
ALTER TABLE 表 ADD INDEX 索引名 (name(10),city,age);
```

建表时，name长度为16，这里用10。这是因为一般情况下名字的长度不会超过10，这样会加速索引查询速度，还会减少索引文件的大小，提高INSERT的更新速度。

如果分别在 name，city，age上建立单列索引，让该表有3个单列索引，查询时和上述的组合索引效率也会大不一样，远远低于我们的组合索引。虽然此时有了三个索引，但MySQL只能用到其中的那个它认为似乎是最有效率的单列索引。

建立这样的组合索引，其实是相当于分别建立了下面三组组合索引：

```
name,city,age
name,city
name
```

为什么没有city，age这样的组合索引呢？
这是因为MySQL组合索引依据“最左前缀”的结果。简单的理解就是只从最左面的开始组合。并不是只要包含这三列的查询都会用到该组合索引。

下面两条会用到组合索引：

```
SELECT * FROM USER WHREE name="admin" AND city="郑州"

SELECT * FROM USER WHREE name="admin"
```

而下面两个则不会用到：

```
SELECT * FROM USER WHREE age=20 AND city="郑州"
SELECT * FROM USER WHREE city="郑州"
```

1.4 建立索引的时机

我们需要在什么情况下建立索引呢？

一般来说，在WHERE和JOIN中出现的列需要建立索引.

当字段上建有索引时，通常以下情况会使用该索引：

```
INDEX_COLUMN = ?
INDEX_COLUMN > ?
INDEX_COLUMN >= ?
INDEX_COLUMN < ?
INDEX_COLUMN <= ?
INDEX_COLUMN between ? and ?
INDEX_COLUMN in (?,?,...,?)
#like操作一般在全文索引中会用到（InnoDB数据表不支持全文索引）,且必须是后导模糊查询
INDEX_COLUMN like 'XXX%'（后导模糊查询）
T1.INDEX_COLUMN=T2.COLUMN（两个表通过索引字段关联）
```

当字段上建有索引时，但以下情况不会使用该索引：

```
#不等操作不能使用索引
INDEX_COLUMN <>?
INDEX_COLUMN not in (?,?,?)
#经过普通运算或函数处理的字段，不能使用索引
function(INDEX_COLUMN)=?
INDEX_COLUMN+1=?
INDEX_COLUMN || 'a'=?
#前导模糊查询不能使用索引
INDEX_COLUMN like '%XXX'
INDEX_COLUMN like '%XXX%'
INDEX_COLUMN like '_XXX'
#B-Tree索引不保存null值，因此有null值字段，不能使用索引
INDEX_COLUMN is null
#MySQL查询只使用一个索引，因此如果where子句中已经使用了索引的话，那么order by中的列是不会使用索引的。因此数据库默认排序可以符合要求的情况下不要使用排序操作；尽量不要包含多个列的排序，如果需要最好给这些列创建复合索引
where INDEX_COLUMN = ? order by INDEX_COLUMN desc
```

1.5 索引的不足之处

面都在说使用索引的好处，但过多的使用索引将会造成滥用。因此索引也会有它的缺点：

1.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，还要保存索引文件。

索引对DML(INSERT,UPDATE,DELETE)附加的开销有多少？

这个没有固定的比例，与每个表记录的大小及索引字段大小密切相关，以下是一个普通表测试数据，仅供参考：

索引对于Insert性能降低56%
索引对于Update性能降低47%
索引对于Delete性能降低29%
因此对于写IO压力比较大的系统，表的索引需要仔细评估必要性。

2.建立索引会占用磁盘空间的索引文件。一般情况这个问题不太严重，但如果你在一个大表上创建了多种组合索引，索引文件的会膨胀很快。

3.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。
因此应该只为最经常查询和最经常排序的数据列建立索引。注意，如果某个数据列包含许多重复的内容，为它建立索引就没有太大的实际效果，如sex列。

4.从理论上讲，完全可以为数据表里的每个字段分别建一个索引，但MySQL把同一个数据表里的索引总数限制为16个。

1.6 总结

只有当数据库里已经有了足够多的测试数据时，它的性能测试结果才有实际参考价值。如果在测试数据库里只有几百条数据记录，它们往往在执行完第一条查询命令之后就被全部加载到内存里，这将使后续的查询命令都执行得非常快–不管有没有使用索引。

只有当数据库里的记录超过了1000条、数据总量也超过了 MySQL服务器上的内存总量时，数据库的性能测试结果才有意义。

在不确定应该在哪些数据列上创建索引的时候，人们从EXPLAIN SELECT（执行计划，后面会提到）命令那里往往可以获得一些帮助。这其实只是简单地给一条普通的SELECT命令加一个EXPLAIN关键字作为前缀而已。
有了这个关键字，MySQL将不是去执行那条SELECT命令，而是去对它进行分析。MySQL将以表格的形式把查询的执行过程和用到的索引(如果有的话)等信息列出来。

2.返回更少的数据

2.1数据分页处理

客户端(应用程序或浏览器)分页：

将数据从应用服务器全部下载到本地应用程序或浏览器，在应用程序或浏览器内部通过本地代码进行分页处理
优点：编码简单，减少客户端与应用服务器网络交互次数
缺点：首次交互时间长，占用客户端内存
适应场景：客户端与应用服务器网络延时较大，但要求后续操作流畅，如手机GPRS，超远程访问（跨国）等等。

应用服务器分页：

将数据从数据库服务器全部下载到应用服务器，在应用服务器内部再进行数据筛选。以下是一个应用服务器端Java程序分页的示例：

```
List list=executeQuery(“select * from employee order by id”);
Int count= list.size();
List subList= list.subList(10, 20);
```

优点：编码简单，只需要一次SQL交互，总数据与分页数据差不多时性能较好。
缺点：总数据量较多时性能较差。
适应场景：数据库系统不支持分页处理，数据量较小并且可控。

数据库SQL分页：

采用数据库SQL分页需要两次SQL完成
一个SQL计算总数量
一个SQL返回分页后的数据
优点：性能好
缺点：编码复杂，各种数据库语法不同，需要两次SQL交互（查询总记录数SQL，分页SQL）。

数据访问开销=索引IO+索引分页结果对应的表数据IO.

2.2 只返回需要的字段

通过去除不必要的返回字段可以提高性能，例：
调整前： `select * from product where company_id=?;`
调整后： `select id,name from product where company_id=?;`

优点：
1、减少数据在网络上传输开销
2、减少服务器数据处理开销
3、减少客户端内存占用
4、字段变更时提前发现问题，减少程序BUG
5、如果访问的所有字段刚好在一个索引里面，则可以使用纯索引访问提高性能。
缺点：
增加编码工作量。
由于会增加一些编码工作量，所以一般需求通过开发规范来要求程序员这么做，否则等项目上线后再整改工作量更大。
如果你的查询表中有大字段或内容较多的字段，如备注信息、文件内容等等，那在查询表时一定要注意这方面的问题，否则可能会带来严重的性能问题。

如果表经常要查询并且请求大内容字段的概率很低，我们可以采用分表处理，将一个大表分拆成两个一对一的关系表，将不常用的大内容字段放在一张单独的表中。
如一张存储上传文件的表：
T_FILE (ID,FILE_NAME,FILE_SIZE,FILE_TYPE,FILE_CONTENT)

我们可以分拆成两张一对一的关系表：
T_FILE (ID,FILE_NAME,FILE_SIZE,FILE_TYPE)
T_FILECONTENT (ID, FILE_CONTENT)

通过这种分拆，可以大大提少T_FILE表的单条记录及总大小，这样在查询T_FILE时性能会更好，当需要查询FILE_CONTENT字段内容时再访问T_FILECONTENT表。

3.减少交互次数

3.1 批处理（batch DML）

数据库访问框架一般都提供了批量提交的接口，jdbc支持batch的提交处理方法，当你一次性要往一个表中插入1000万条数据时，如果采用普通的executeUpdate处理，
那么和服务器交互次数为1000万次，
按每秒钟可以向数据库服务器提交10000次估算，
要完成所有工作需要1000秒。

如果采用批量提交模式，1000条提交一次，那么和服务器交互次数为1万次，交互次数大大减少。
采用batch操作一般不会减少很多数据库服务器的物理IO，但是会大大减少客户端与服务端的交互次数，从而减少了多次发起的网络延时开销，同时也会降低数据库的CPU开销。

假设要向一个普通表插入1000万数据，每条记录大小为1K字节，表上没有任何索引，客户端与数据库服务器网络是100Mbps，以下是根据现在一般计算机能力估算的各种batch大小性能对比值如下图：

单位：ms	No batch	Batch=10	Batch=100	Batch=1000	Batch=10000
服务器事务处理时间	0.1	0.1	0.1	0.1	0.1
服务器IO处理时间	0.02	0.2	2	20	200
网络交互发起时间	0.1	0.1	0.1	0.1	0.1
网络数据传输时间	0.01	0.1	1	10	100
小计	0.23	0.5	3.2	30.2	300.2
平均每条记录处理时间	0.23	0.05	0.032	0.0302	0.03002

从上可以看出，Insert操作加大Batch可以对性能提高近8倍性能，一般根据主键的Update或Delete操作也可能提高2-3倍性能，但不如Insert明显，因为Update及Delete操作可能有比较大的开销在物理IO访问。以上仅是理论计算值，实际情况需要根据具体环境测量。

3.2 In List方式

很多时候我们需要按一些ID查询数据库记录，我们可以采用一个ID一个请求发给数据库，如下所示：

```
for :var in ids[] do begin
  select * from mytable where id=:var;
end;
```

我们也可以做一个小的优化， 如下所示，用ID INLIST的这种方式写SQL：

```
select * from mytable where id in(:id1,id2,...,idn);
```

1.通过这样处理可以大大减少SQL请求的数量，从而提高性能。那如果有10000个ID，那是不是全部放在一条SQL里处理呢？答案肯定是否定的。首先大部份数据库都会有SQL长度和IN里个数的限制，如ORACLE的IN里就不允许超过1000个值。
mysql中，in语句中参数个数是不限制的。不过对整段sql语句的长度有了限制.

2.另外当前数据库一般都是采用基于成本的优化规则，当IN数量达到一定值时有可能改变SQL执行计划，从索引访问变成全表访问，这将使性能急剧变化。随着SQL中IN的里面的值个数增加，SQL的执行计划会更复杂，占用的内存将会变大，这将会增加服务器CPU及内存成本。

3.评估在IN里面一次放多少个值还需要考虑应用服务器本地内存的开销，有并发访问时要计算本地数据使用周期内的并发上限，否则可能会导致内存溢出。

4.综合考虑，一般IN里面的值个数超过20个以后性能基本没什么太大变化，也特别说明不要超过100，超过后可能会引起执行计划的不稳定性及增加数据库CPU及内存成本，这个需要专业DBA评估。

3.3设置FetchSize

当我们采用select从数据库查询数据时，数据默认并不是一条一条返回给客户端的，也不是一次全部返回客户端的，而是根据客户端fetch_size参数处理，每次只返回fetch_size条记录，当客户端游标遍历到尾部时再从服务端取数据，直到最后全部传送完成。

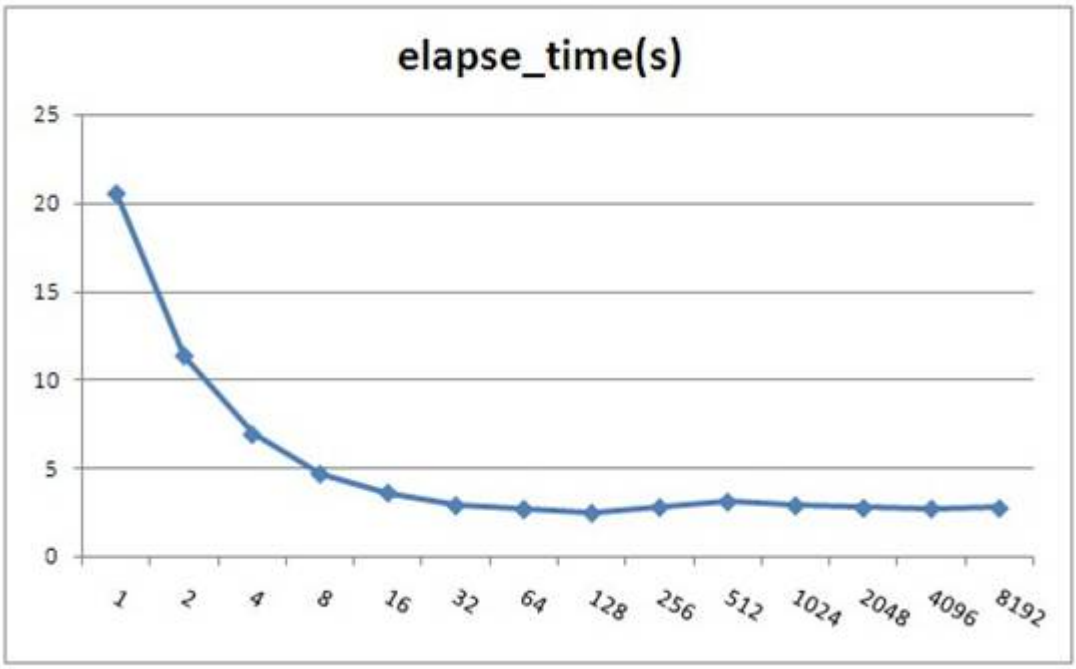
所以如果我们要从服务端一次取大量数据时，可以加大fetch_size，这样可以减少结果数据传输的交互次数及服务器数据准备时间，提高性能。

以下是jdbc测试的代码，采用本地数据库，表缓存在数据库CACHE中，因此没有网络连接及磁盘IO开销，客户端只遍历游标，不做任何处理，这样更能体现fetch参数的影响：

```
String vsql ="select * from t_employee";
PreparedStatement pstmt =
conn.prepareStatement(vsql,ResultSet.TYPE_FORWARD_ONLY,ResultSet.CONCUR_READ_ONLY);
//设置批处理
pstmt.setFetchSize(1000);
ResultSet rs = pstmt.executeQuery(vsql);
//每条记录的列数
int cnt = rs.getMetaData().getColumnCount();
Object o;
while (rs.next()) {
    for (int i = 1; i <= cnt; i++) {
       o = rs.getObject(i);
    }
}
```

测试示例中的employee表有100000条记录，每条记录平均长度135字节
以下是测试结果，对每种fetchsize测试5次再取平均值：

fetchsize	elapse_time（s）
1	20.516
2	11.34
4	6.894
8	4.65
16	3.584
32	2.865
64	2.656
128	2.44
256	2.765
512	3.075
1024	2.862
2048	2.722
4096	2.681
8192	2.715



由上测试可以看出fetchsize对性能影响还是比较大的，但是当fetchsize大于100时就基本上没有影响了。

fetchsize并不会存在一个最优的固定值，因为整体性能与记录集大小及硬件平台有关。根据测试结果建议当一次性要取大量数据时这个值设置为100左右，不要小于40。注意，fetchsize不能设置太大，如果一次取出的数据大于JVM的内存会导致内存溢出，所以建议不要超过1000，太大了也没什么性能提高，反而可能会增加内存溢出的危险。

注意：oracle一般默认设置为10
mysql5以上，可以在连接设置jdbc:mysql://127.0.0.1/：3306/test?useCursorFetch=true;即可以批量读取。

3.4使用存储过程

大型数据库一般都支持存储过程，合理的利用存储过程也可以提高系统性能。如你有一个业务需要将A表的数据做一些加工然后更新到B表中，但是又不可能一条SQL完成，这时你需要如下3步操作：
- a：将A表数据全部取出到客户端；
- b：计算出要更新的数据；
- c：将计算结果更新到B表。
如果采用存储过程你可以将整个业务逻辑封装在存储过程里，然后在客户端直接调用存储过程处理，这样可以减少网络交互的成本。

缺点：
a、不可移植性，每种数据库的内部编程语法都不太相同，当你的系统需要兼容多种数据库时最好不要用存储过程。
b、学习成本高，DBA一般都擅长写存储过程，但并不是每个程序员都能写好存储过程，除非你的团队有较多的开发人员熟悉写存储过程，否则后期系统维护会产生问题。
c、业务逻辑多处存在，采用存储过程后也就意味着你的系统有一些业务逻辑不是在应用程序里处理，这种架构会增加一些系统维护和调试成本。
d、存储过程和常用应用程序语言不一样，它支持的函数及语法有可能不能满足需求，有些逻辑就只能通过应用程序处理。
e、如果存储过程中有复杂运算的话，会增加一些数据库服务端的处理成本，对于集中式数据库可能会导致系统可扩展性问题。
f、为了提高性能，数据库会把存储过程代码编译成中间运行代码(类似于java的class文件)，所以更像静态语言。当存储过程引用的对像(表、视图等等)结构改变后，存储过程需要重新编译才能生效，在24*7高并发应用场景，一般都是在线变更结构的，所以在变更的瞬间要同时编译存储过程，这可能会导致数据库瞬间压力上升引起故障(Oracle数据库就存在这样的问题)。

个人观点：普通业务逻辑尽量不要使用存储过程，定时性的ETL任务或报表统计函数可以根据团队资源情况采用存储过程处理。

3.5优化业务逻辑

要通过优化业务逻辑来提高性能是比较困难的，这需要程序员对所访问的数据及业务流程非常清楚。

举一个案例：
某移动公司推出优惠套参，活动对像为VIP会员并且2010年1，2，3月平均话费20元以上的客户。
那我们的检测逻辑为：

```
#查寻该用户平均话费

select avg(money) as avg_money from bill
```

```
where phone_no='13988888888' and date between '201001' and '201003';
#查询该用户是否是VIP
select vip_flag from member where phone_no='13988888888';
#判断是否给于优惠套参
if avg_money>20 and vip_flag=true then
begin
  执行套参();
end;

#如果我们修改业务逻辑为:

select avg(money) as  avg_money from bill where phone_no='13988888888' and date between '201001'
and '201003';
#先判断平均话费是否满足条件
if avg_money>20 then
begin
  select vip_flag from member where phone_no='13988888888';
  #判断是否是VIP
  if vip_flag=true then
  begin
    执行套参();
  end;
end;

#通过这样可以减少一些判断vip_flag的开销，平均话费20元以下的用户就不需要再检测是否VIP了。
#如果程序员分析业务，VIP会员比例为1%，平均话费20元以上的用户比例为90%，那我们改成如下：

#先过滤记录数最少的情况
select vip_flag from member where phone_no='13988888888';
#判断是否是VIP
if vip_flag=true then
begin
  select avg(money) as avg_money from bill where phone_no='13988888888' and date between
'201001' and '201003';
  #判断平均话费
  if avg_money>20 then
  begin
    执行套参();
  end;
end;

#这样就只有1%的VIP会员才会做检测平均话费，最终大大减少了SQL的交互次数。
```

以上只是一个简单的示例，实际的业务总是比这复杂得多，所以一般只是高级程序员更容易做出优化的逻辑，但是我们需要有这样一种成本优化的意识。

4.减少数据库服务器CPU运算

4.1合量使用排序

普通OLTP（联机事务处理，传统的关系型数据库的主要应用）系统排序操作一般都是在内存里进行的，对于数据库来说是一种CPU的消耗，曾在PC机做过测试，单核普通CPU在1秒钟可以完成100万条记录的全内存排序操作，所以说由于现在CPU的性能增强，对于普通的几十条或上百条记录排序对系统的影响也不会很大。但是当你的记录集增加到上万条以上时，你需要注意是否一定要这么做了，大记录集排序不仅增加了CPU开销，而且可能会由于内存不足发生硬盘排序的现象，当发生硬盘排序时性能会急剧下降，这种需求需要与DBA沟通再决定，取决于你的需求和数据，所以只有你自己最清楚，而不要被别人说排序很慢就吓倒。

以下列出了可能会发生排序操作的SQL语法：
Order by
Group by
Distinct
Exists子查询
Not Exists子查询
In子查询
Not In子查询
Union（并集），Union All也是一种并集操作，但是不会发生排序，如果你确认两个数据集不需要执行去除重复数据操作，那请使用Union All 代替Union。
Create Index

4.2减少比较操作

我们SQL的业务逻辑经常会包含一些比较操作，如a=b，a<b之类的操作，对于这些比较操作数据库都体现得很好，但是如果有以下操作，我们需要保持警惕：
Like模糊查询，如下所示：
a like ‘%abc%’

Like模糊查询对于数据库来说不是很擅长，特别是你需要模糊检查的记录有上万条以上时，性能比较糟糕，这种情况一般可以采用专用Search或者采用全文索引方案来提高性能。

没使用索引的列作为in的条件时，如下所示：
a in (:1,:2,:3,…,:n) ----n>20
如果这里的a字段不能通过索引比较，那数据库会将字段与in里面的每个值都进行比较运算，如果记录数有上万以上，会明显感觉到SQL的CPU开销加大

4.3大量复杂运算在客户端处理

1.什么是复杂运算，一般我认为是一秒钟CPU只能做10万次以内的运算。如含小数的对数及指数运算、三角函数、3DES及BASE64数据加密算法等等。

2.如果有大量这类函数运算，尽量放在客户端处理，一般CPU每秒中也只能处理1万-10万次这样的函数运算，放在数据库内不利于高并发处理。

5.利用更多的资源

多进程并行访问是指在客户端创建多个进程(线程)，每个进程建立一个与数据库的连接，然后同时向数据库提交访问请求。当数据库主机资源有空闲时，我们可以采用客户端多进程并行访问的方法来提高性能。

如果数据库主机已经很忙时，采用多进程并行访问性能不会提高，反而可能会更慢。所以使用这种方式最好与DBA或系统管理员进行沟通后再决定是否采用。

例如：
我们有10000个产品ID，现在需要根据ID取出产品的详细信息，如果单线程访问，按每个IO要5ms计算，忽略主机CPU运算及网络传输时间，我们需要50s才能完成任务。如果采用5个并行访问，每个进程访问2000个ID，那么10s就有可能完成任务。

那是不是并行数越多越好呢，开1000个并行是否只要50ms就搞定，答案肯定是否定的，当并行数超过服务器主机资源的上限时性能就不会再提高，如果再增加反而会增加主机的进程间调度成本和进程冲突机率。

以下是一些如何设置并行数的基本建议：
如果瓶颈在服务器主机，但是主机还有空闲资源，那么最大并行数取主机CPU核数和主机提供数据服务的磁盘数两个参数中的最小值，同时要保证主机有资源做其它任务。
如果瓶颈在客户端处理，但是客户端还有空闲资源，那建议不要增加SQL的并行，而是用一个进程取回数据后在客户端起多个进程处理即可，进程数根据客户端CPU核数计算。
如果瓶颈在客户端网络，那建议做数据压缩或者增加多个客户端，采用map reduce的架构处理。
如果瓶颈在服务器网络，那需要增加服务器的网络带宽或者在服务端将数据压缩后再处理了。

二.MySQL执行计划

1.explain用法详解

1.1语法

```
#可以得出一个表的字段结构等
1.EXPLAIN tbl_name;
#可以得到相关的一些索引信息
2.EXPLAIN [EXTENDED] SELECT select_options;
```

重点讲第二种用法：

```
EXPLAIN SELECT * FROM USER WHERE id = 1;
```

分析结果：

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	user	const	PRIMARY	PRIMARY	4	const	1	*(NULL)*

字段含意：

```
1. id
select查询的序列号


2. select_type
```

select查询的类型，主要是区别普通查询和联合查询、子查询之类的复杂查询。取值如下：
    a.SIMPLE：查询中不包含子查询或者UNION
    b.查询中若包含任何复杂的子部分，最外层查询则被标记为：PRIMARY
    c.在SELECT或WHERE列表中包含了子查询，该子查询被标记为：SUBQUERY
    d.在FROM列表中包含的子查询被标记为：DERIVED（衍生）
    e.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将被标记为：DERIVED
    f.从UNION表获取结果的SELECT被标记为：UNION RESULT

3.table
输出的行所引用的表。

4.type
联合查询所使用的类型，表示MySQL在表中找到所需行的方式，又称“访问类型”。
type显示的是访问类型，是较为重要的一个指标，结果值从好到坏依次是：
    system > const > eq_ref > ref > fulltext > ref_or_null > index_merge >
    unique_subquery > index_subquery > range > index > ALL
一般来说，得保证查询至少达到range级别，最好能达到ref。
    ALL: 扫描全表
    index: 扫描全部索引树
    range: 扫描部分索引，索引范围扫描，对索引的扫描开始于某一点，返回匹配值域的行，
    常见于between、<、>等的查询
    ref: 非唯一性索引扫描，返回匹配某个单独值的所有行。常见于使用非唯一索引即唯一索引的非唯一前缀进行的查找
    eq_ref：唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配。常见于主键或唯一索引扫描
    const, system: 当MySQL对查询某部分进行优化，并转换为一个常量时，使用这些类型访问。
    如将主键置于where列表中，MySQL就能将该查询转换为一个常量。system是const类型的特例，当查询的表只有一行的情况下， 使用system。

5.possible_keys
指出MySQL能使用哪个索引在该表中找到行。查询涉及到的字段上若存在索引，则该索引将被列出，但不一定被查询使用。
如果是空的，没有相关的索引。这时要提高性能，可通过检验WHERE子句，看是否引用某些字段，或者检查字段不是适合索引。

6.key
显示MySQL实际决定使用的键。如果没有索引被选择，键是NULL。

7.key_len
显示MySQL决定使用的键长度。表示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。
如果键是NULL，长度就是NULL。文档提示特别注意这个值可以得出一个多重主键里mysql实际使用了哪一部分。
注：
key_len显示的值为索引字段的最大可能长度，并非实际使用长度，即key_len是根据表定义计算而得，不是通过表内检索出的。

8.ref
显示哪个字段或常数与key一起被使用。

9.rows
这个数表示mysql要遍历多少数据才能找到，表示MySQL根据表统计信息及索引选用情况，估算的找到所需的记录所需要读取的行数，在innodb上可能是不准确的。

10.Extra

包含不适合在其他列中显示但十分重要的额外信息。

Only index，这意味着信息只用索引树中的信息检索出的，这比扫描整个表要快。

using where是使用上了where限制，表示MySQL服务器在存储引擎受到记录后进行“后过滤”（Post-filter），如果查询未能使用索引，Using where的作用只是提醒我们MySQL将用where子句来过滤结果集。

impossible where 表示用不着where，一般就是没查出来啥。

Using filesort（MySQL中无法利用索引完成的排序操作称为“文件排序”）当我们试图对一个没有索引的字段进行排序时，就是filesort。它跟文件没有任何关系，实际上是内部的一个快速排序。

Using temporary（表示MySQL需要使用临时表来存储结果集，常见于排序和分组查询），使用filesort和temporary的话会很吃力，WHERE和ORDER BY的索引经常无法兼顾，
如果按照WHERE来确定索引，那么在ORDER BY时，就必然会引起Using filesort，这就要看是先过滤再排序划算，还是先排序再过滤划算。

1.2示例

```
mysql> explain select d1.name, (select id from t3) d2
    -> from (select id, name from t1 where other_column = '') d1
    -> union
    -> (select name, id from t2);
+------+--------------+------------+--------+---------------+---------+---------+------+------+-------------+
| id   | select_type  | table      | type   | possible_keys | key     | key_len | ref  | rows | Extra       |
+------+--------------+------------+--------+---------------+---------+---------+------+------+-------------+
|  1   | PRIMARY      | <derived3> | system | NULL          | NULL    | NULL    | NULL |    1 |             |
|  3   | DERIVED      | t1         | ALL    | NULL          | NULL    | NULL    | NULL |    1 | Using where |
|  2   | SUBQUERY     | t3         | index  | NULL          | PRIMARY | 4       | NULL |    1 | Using index |
|  4   | UNION        | t2         | ALL    | NULL          | NULL    | NULL    | NULL |    1 |             |
| NULL | UNION RESULT | <union1,4> | ALL    | NULL          | NULL    | NULL    | NULL | NULL |             |
+------+--------------+------------+--------+---------------+---------+---------+------+------+-------------+
5 rows in set (0.01 sec)
```


第一行：id列为1，表示第一个select，select_type列的primary表示该查询为外层查询，table列被标记为<derived3>，表示查询结果来自一个衍生表，其中3代表该查询衍生自第三个select查询，即id为3的select。
sql:[select d1.name.....]

第二行：id为3，表示该查询的执行次序为2（4→3），是整个查询中第三个select的一部分。因查询包含在from中，所以为derived。
sql：[select id,name from t1 where other_column='']

第三行：select列表中的子查询，select_type为subquery，为整个查询中的第二个select。
sql:[select id from t3]

第四行：select_type为union，说明第四个select是union里的第二个select，最先执行。
sql:[select name,id from t2]

第五行：代表从union的临时表中读取行的阶段，table列的<union1,4>表示用第一个和第四个select的结果进行union操作。[两个结果union操作]

1.3执行计划的局限性

1.EXPLAIN不会告诉你关于触发器、存储过程的信息或用户自定义函数对查询的影响情况
2.EXPLAIN不考虑各种Cache
3.EXPLAIN不能显示MySQL在执行查询时所作的优化工作
4.部分统计信息是估算的，并非精确值
5.EXPALIN只能解释SELECT操作，其他操作要重写为SELECT后查看。