

**内存数据库项目报告文档**

|  |
| --- |
| Boying——博影票务系统 |

指导教师*:* **袁时金**

朱明志 (1854116)

石稼晟 (1851632)

谭梓煊 (1853434)

李翠琪 (1751022)

沈钰姣 (1853972)

In-Memory Database

Tongji University

School of Software Engineering

# 目录

[Boying——博影票务系统 I](#_Toc61825296)

[目录 2](#_Toc61825297)

[一、 SQL语句优化实验 5](#_Toc61825298)

[1.1 实验背景 5](#_Toc61825299)

[1.2 实验方案 5](#_Toc61825300)

[1.2.1 调整 WHERE 子句中的连接顺序 6](#_Toc61825301)

[1.2.2 SELECT 子句中不使用‘\*’ 7](#_Toc61825302)

[1.2.3 用 EXISTS 替换 DISTINCT 8](#_Toc61825303)

[1.2.4 使用 Where 替代 Having 9](#_Toc61825304)

[二、 索引优化实验 11](#_Toc61825305)

[2.1 实验目的 11](#_Toc61825306)

[2.2 实验方案 11](#_Toc61825307)

[2.3 实验结果 11](#_Toc61825308)

[三、 预编译语句实验 13](#_Toc61825309)

[3.1 实验背景 13](#_Toc61825310)

[3.2 对应功能点 13](#_Toc61825311)

[3.3 实验代码 13](#_Toc61825312)

[3.4 实验结果 14](#_Toc61825313)

[3.5 实验结论 14](#_Toc61825314)

[四、 存储过程实验 15](#_Toc61825315)

[4.1 实验背景 15](#_Toc61825316)

[4.2 对应功能点 15](#_Toc61825317)

[4.3 SQL语句 15](#_Toc61825318)

[4.4 实验结果 16](#_Toc61825319)

[4.5 实验结论 16](#_Toc61825320)

[五、 物化视图实验 17](#_Toc61825321)

[5.1 实验背景 17](#_Toc61825322)

[5.2 对应功能点 17](#_Toc61825323)

[5.3 实验步骤 17](#_Toc61825324)

[5.4 SQL语句 17](#_Toc61825325)

[5.5 实验过程 18](#_Toc61825326)

[5.6 实验结论与分析 18](#_Toc61825327)

[5.7 实验心得 19](#_Toc61825328)

[六、 数据库连接模式实验 21](#_Toc61825329)

[6.1 实验背景 21](#_Toc61825330)

[6.2 对应功能点 21](#_Toc61825331)

[6.3 实验步骤 22](#_Toc61825332)

[6.4 实验代码 22](#_Toc61825333)

[6.5 实验结果 23](#_Toc61825334)

[6.6 实验结论 24](#_Toc61825335)

[6.7 实验心得 24](#_Toc61825336)

[七、 SGA调整实验 26](#_Toc61825337)

[7.1 实验背景 26](#_Toc61825338)

[7.2 对应功能点 26](#_Toc61825339)

[7.3 实验步骤 26](#_Toc61825340)

[7.4 SQL语句 26](#_Toc61825341)

[7.5 实验结果 27](#_Toc61825342)

[7.6 实验结论 27](#_Toc61825343)

[八、 批量读写实验 28](#_Toc61825344)

[8.1 写入速度实验 28](#_Toc61825345)

[8.1.1 实验背景 28](#_Toc61825346)

[8.1.2 功能对应点 28](#_Toc61825347)

[8.1.3 实验步骤 28](#_Toc61825348)

[8.1.4 SQL 语句（其中一例） 28](#_Toc61825349)

[8.1.5 实验结果 29](#_Toc61825350)

[8.1.6 实验结论 29](#_Toc61825351)

[8.2 读取速度实验 29](#_Toc61825352)

[8.2.1 实验背景 29](#_Toc61825353)

[8.2.2 对应功能点 29](#_Toc61825354)

[8.2.3 实验步骤 30](#_Toc61825355)

[8.2.4 SQL语句 30](#_Toc61825356)

[8.2.5 实验结果 30](#_Toc61825357)

[九、 数据类型实验 31](#_Toc61825358)

[9.1 实验背景 31](#_Toc61825359)

[9.2 功能对应点 31](#_Toc61825360)

[9.3 实验步骤 32](#_Toc61825361)

[9.4 SQL语句 32](#_Toc61825362)

[9.5 实验结果 32](#_Toc61825363)

[9.5.1 账号存储类型的影响 32](#_Toc61825364)

[9.5.2 演出存储类型的影响 33](#_Toc61825365)

[9.6 实验结论 33](#_Toc61825366)

[成员贡献 34](#_Toc61825367)

[附录A. 图目录 35](#_Toc61825368)

[附录B. 表目录 36](#_Toc61825369)

# SQL语句优化实验

## 实验背景

用户在使用博影系统时，查看订单为主要使用功能之一，其中包含多种方式的订单的查看，因而就包含多种 SQL 查询语句。因此这一实验，我们将主要探究 SQL 查询语句优化在性能优化中的应用和影响。

## 实验方案

对于相同的数据库，有不同的查询方法可以获取想要的数据，对于不同的查询语句数据库也会采用不同的扫描策略。因此我们将通过不断修改 SQL 语句，对比各个形式下的查询执行时间，形成对比图表。

### 调整 WHERE 子句中的连接顺序

ORACLE 采用自下而上的顺序解析 WHERE 子句，根据这个原理，表之间的连接必须写在其他 WHERE 条件之前，那些可以过滤掉最大数量记录的条件必须写在 WHERE 子句的末尾。

#### 对应功能点

查询具有特定ID和超过一定数量订单数量的用户

#### 实验步骤

我们对年龄大小（条件 A）、订单数量（条件 B）两个参数进行控制，使得在前后两次实验中两个条件的过滤能力大小关系相反。比较交换 where 子句后的查询时间：

#### SQL 语句

select u.\*

from boying\_user u

where u.AGE <20 and 1 < (

select count(\*)

from boying\_order o

where o.USER\_ID = u.ID

);

select u.\*

from boying\_user u

where 1 < (

select count(\*)

from boying\_order o

where o.USER\_ID = u.ID

) and u.AGE <20;

#### 实验结果

|  |  |  |
| --- | --- | --- |
| 过滤条件顺序 | 条件A在前 | 条件B在前 |
| 时间 | 3.728s | 3.761s |

#### 实验结论

实验结果表明，调整过滤能力强的句子先后顺序，仅可以起到略微的优化效果，与我们的预期不一致。条件A的过滤能力比较强，我们对比了条件 A在后和条件 B 在后时 oracle 的执行计划后。查阅资料后了解到，对于 where 子句中的多个条件连接，oracle 的优化器可以像调整表连接顺序一样自动进行优化，使得连接顺序自动调整为最优后执行，因此时间差仅仅来自于 oracle 优化器所使用的优化时间。

### SELECT 子句中不使用‘\*’

ORACLE 在解析的过程中，会将‘\*’依次转换成所有的列名，这个工作是通过查询数据字典完成的，这意味着将耗费更多的时间。

#### 对应功能点

查询用户关注的所有人发的订单。

#### SQL 语句

select u.\* from boying\_user u

where u.ID like '%1';select u.ID, u.USERNAME, u.PASSWORD, u.PHONE, u.EMAIL, u.ICON, u.REAL\_NAME, u.IDENTITY\_NUMBER, u.AGE, u.CREATE\_TIME, u.GENDER, u.ADMIN\_DELETE from boying\_user u

where u.ID like '%1';

#### 实验结果

|  |  |  |
| --- | --- | --- |
| 时间 | 优化前 | 优化后 |
| Oracle | 58.2s | 52.6s |

#### 实验结论

实验结果表明不使用\*并没有明显优势。猜想原因在于我们通常使用的数据库表列数量通常不会太多，查询一次数据字典的耗时不会太长。我们查看执行计划的关键部分发现在磁盘读写以及扫描快数量上面，仅存在细微差别，最主要的差别来源于数据字典的一次调用，增加了 recursive calls 的次数，这也是时间差的主要来源。

### 用 EXISTS 替换 DISTINCT

当提交一个包含一对多表信息（对用户粉丝）的查询时，避免在SELECT子句中使用 DISTINCT. 一般可以考虑用 EXIST 替换，EXISTS 使查询更为迅速，因为 RDBMS 核心模块将在子查询的条件一旦满足后，立刻返回结果。

#### 对应功能点

查询用户粉丝列表

#### SQL 语句

select distinct o.ID, o.USER\_ID

from boying\_user u, boying\_order o

where u.ID = o.USER\_ID;select o.ID, o.USER\_ID

from boying\_order o

where exists (

select null from boying\_user u

where u.ID = o.USER\_ID

);

#### 实验结果

|  |  |  |
| --- | --- | --- |
| 类别 | DISTINCT | EXISTS |
| 时间 | 2.134s | 0.058s |

#### 实验结论

首先说明一下 exists 的工作原理，exists 语句用来判断()内的表达式是否存在返回值，如果存在就返回 true，如果不存在就返回 false，所以在上面语句中我们使用 select null，因为我们不关注返回值。另外 exists 的优点是，它只要括号中的表达式有一个值存在，就立刻返回 true，而不用遍历表中所有的数据。所以在 USER\_ID 同样多的情况下，用 distinct 需要每次都遍历 USER 表进行比对，而使用 exists 只需要比对 USER 表的一部分，在 USER 表数据十分庞大时，这种性能差别就能更好的体现出来。

### 使用 Where 替代 Having

where是用于过滤行的，而having是用来过滤组的，因为行被分组后，having才能过滤组，所以使用 where 的过滤速度更快。

#### 对应功能点

查询用户的粉丝数量

#### SQL 语句

select u.REAL\_NAME, count(u.ID)

from boying\_order o join boying\_user u on o.USER\_ID = u.ID

having o.MONEY < 200

group by u.REAL\_NAME order by count(u.ID) desc;

select u.REAL\_NAME, count(u.ID)

from boying\_order o, boying\_user u

where o.MONEY < 200 and o.USER\_ID = u.ID

group by u.REAL\_NAME order by count(u.ID) desc;

#### 实验结果

|  |  |  |
| --- | --- | --- |
| 语句 | having | where |
| 时间 | 0.938s | 0.208s |

#### 实验结论

where是用于过滤行的，而having是用来过滤组的，因为行被分组后，having才能过滤组，所以使用 where 的过滤速度更快。在 USER 表数据十分庞大时，这种性能差别就能更好的体现出来。

# 索引优化实验

索引可以大大提高数据库在某方面的性能，但不可避免地会在另一方面降低性能，而且索引种类很多，不同索引适用的场景也有所不同。

## 实验目的

通过针对不同字段及其组合，在Oracle和TimesTen数据库中分别建立不同类型的索引，并进行查询和插入两种操作检验性能的变化，最终找到适用于不同数据库的最合理的索引方案，达到提高系统性能的目的。

## 实验方案

针对日期建立索引，日期是用户进行演出场次挑选时的重要依据，用户可以指定演出的开始日期范围，之后系

统返回符合条件的演出结果。针对这点，我们计划对SESSION表的用户名称字段建立索引，比较性能。现设计实验记录表格如下:

|  |  |
| --- | --- |
| 索引类型 | 查询操作 |
| 无索引 | 56ms |
| B树索引 | 51ms |
| 散列索引 | 50ms |

注: TimesTen中使用的索引与Oracle中使用的索引有所不同

## 实验结果

预计复合索引的总体效率和查询操作效率高于单字段索引，但是插入操作效率有所不如，这是因为复合索引针对设计多字段的查询有更好的优化，但是由于涉及字段较多，较为复杂，所以在插入数据之后重建索引时会消耗更多时间；对于单字段索引，情况较为复杂，因为 SHOW表中包含大量数据，而不论是日期还是类型键值数目都与数据量有着巨大的数量级差距，故难以推测实验结果。

# 预编译语句实验

用户在使用博影票务系统时，查看演出，购票为主要使用功能之一，其中包含大量插入（购票）操作。这些操作重复性极强，而系统却需要每一步都重新解析，这浪费了大量时间，而预编译可以避免这种情况。因此这一实验中，我们将主要探究在后端使用预编译时的使用在性能优化中的应用和影响。

## 实验背景

SQL 预编译与 JDBC 有关，预编译语句 PreparedStatement 是 java.sql 中的一个接口，它是 Statement 的子接口。通过 Statement 对象执行SQL 语句时， 需要将 SQL 语句发送给 DBMS，由 DBMS 首先进行编译后再执行。预编译语句和 Statement 不同，在创建 PreparedStatement 对象时就指定了 SQL 语句，该语句立即发送给 DBMS 进行编译。当该编译语句被执行时，DBMS 直接运行编译后的 SQL 语句，而不需要像其他 SQL 语句那样首先将其编译。预编译的 SQL 语句处理性能稍微高于普通的传递变量的办法。

## 对应功能点

购票下单

## 实验代码

在实践中，我们考虑购票下单时，有大量 insert 相似操作，如果采用预编译的方式，可能会减少执行时间，示例代码如下图所示：

PreparedStatement psql = conn.prepareStatement(

"INSERT INTO BOYING\_ORDER\_TEST(USER\_ID, SHOW\_ID, STATUS, " + "TICKET\_COUNT, MONEY) VALUES (?,?,?,?,?)");

long start = System.currentTimeMillis();

for (int i = 1; i <= count; i++) {

psql.setInt(1, random.nextInt(1000)); // user\_id

psql.setInt(2, random.nextInt(10000000)); // show\_id

psql.setInt(3, 1 + random.nextInt(3)); // status

psql.setInt(4, 1 + random.nextInt(3)); // ticket\_count

psql.setInt(5, 0); // money

psql.execute();

}

System.out.println("使用预编译语句:" + (System.currentTimeMillis() - start));

start = System.currentTimeMillis();

for (int i = 1; i <= count; i++) {

Statement statement = conn.createStatement();

statement.execute("INSERT INTO BOYING\_ORDER\_TEST(USER\_ID, SHOW\_ID, STATUS, "

+ "TICKET\_COUNT, MONEY) VALUES (" + random.nextInt(1000) + "," + random.nextInt(10000000) + ","

+ (1 + random.nextInt(3)) + "," + (1 + random.nextInt(3)) + ",0)");

statement.close();

}

System.out.println("不使用预编译语句:" + (System.currentTimeMillis() - start));

## 实验结果

|  |  |  |
| --- | --- | --- |
| 执行次数 | 不使用预编译(ms) | 使用预编译(ms) |
| 100 | 3485 | 3405 |
| 1000 | 35842 | 35083 |

## 实验结论

我们可以看到使用预编译语句在大量重复操作时，效率略微高于不使用预编译语句。

使用预编译的好处：

使用 PreparedStatement 最重要的一点好处是它拥有更佳的性能优势， SQL 语句会预编译在数据库系统中。执行计划同样会被缓存起来，它允许数据库做参数化查询。使用预处理语句比普通的查询更快，因为它做的工作更少（数据库对 SQL 语句的分析，编译，优化已经在第一次查询前完成了）。

其次，使用 PreparedStatement 对象可以大大提高代码的可读性和可维护性。

# 存储过程实验

## 实验背景

存储过程只在创造时进行编译，以后每次执行存储过程都不需再重新编译，而一般SQL语句每执行一次就编译一次,所以使用存储过程可提高数据库执行速度。当对数据库进行复杂操作时（如对多个表进行Update、Insert、Query、Delete时），可将此复杂操作用存储过程封装起来与数据库提供的事务处理结合一起使用。

## 对应功能点

查询用户的订单记录

## SQL语句

方案一、存储过程

create procedure tttrelationlist

(

p\_in\_userid in number,

p\_out\_results out sys\_refcursor

)

AS

begin

open p\_out\_results for select o.ID, o.MONEY, o.STATUS

from BOYING\_ORDER o join BOYING\_USER u

on u.id = o.user\_id where u.id = p\_in\_userid;

end;

/

调用函数测试语句：

declare

p\_results sys\_refcursor;

v\_value1 number;

v\_value2 varchar2(3000);

begin

TTTRELATIONLIST(1235, p\_results);

fetch p\_results into v\_value1, v\_value2;

while p\_results%found

loop

dbms\_output.put\_line(v\_value1 || '' || v\_value2);

fetch p\_results into v\_value1, v\_value2;

end loop;

close p\_results;

end;

方案二、单纯 SQL 语句

select o.ID, o.MONEY, o.STATUS

from BOYING\_ORDER o join BOYING\_USER u

on u.id = o.user\_id

where u.id = 1;

## 实验结果

|  |  |  |
| --- | --- | --- |
| 数据库\查询时间 | 使用存储过程(ms) | 直接 SQL 查询(ms) |
| Oracle | 34 | 76 |
| TimesTen | - | 84 |

## 实验结论

在首次执行存储过程的时候，此时存储过程内容尚未编译，执行时间与直接sql查询基本没有区别；在经过编译之后存储过程中的内容如再次被调用，可以直接执行，效率会高很多。但是 PLSQL 和 SQL 之间的切换，这两者在Oracle内部是不同的执行引擎在负责，频繁切换的开销应该比较客观。

# 物化视图实验

## 实验背景

用户在使用博影系统时，最主要使用的功能为查看已下单的订单。因而订单的查询速度直接影响到响应速度和用户的主观使用感受。我们使用物化视图的方式将首页显示的订单存放在物化视图中，这将避免每次查询时的跨表查找， 然而，用户频繁的发帖等操作也无时无刻不在更新这些物化视图。而物化视图的更新则对用户的使用也会产生影响。因此这一实验，我们将主要探究物化视图在性能优化中的应用和影响。

## 对应功能点

检索订单功能

## 实验步骤

Oracle 的物化视图提供了强大的功能，可以用于预先计算并保存表连接、聚集等耗时较多的操作的结果。这个实验我们将在相同数据量级，同种数据库的条件下，对是否使用物化视图的检索功能记录查询时间，生成性能对比图表，观察物化视图对于性能是否有提升，提升有多大。

同时，因为物化视图具有不同的刷新方式，而这些方式又会影响物化视图的性能，我们打算把物化视图以不同的刷新方式，执行不同量级的 DML 语句进行对比，详细探究物化视图对性能的影响。

## SQL语句

create materialized view ad1

refresh fast

as

select u.REAL\_NAME

from boying\_order o, boying\_user u

where o.MONEY < 200 and o.USER\_ID = u.ID;create materialized view ad2

refresh complete

as

select u.REAL\_NAME

from boying\_order o, boying\_user u

where o.MONEY < 200 and o.USER\_ID = u.ID;create materialized view ad3

refresh force

as

select u.REAL\_NAME

from boying\_order o, boying\_user u

where o.MONEY < 200 and o.USER\_ID = u.ID;

## 实验过程

具体实验中，由于查询时间不易控制，我们使用刷新时间进行对比。

实验结果：（修改后看查询事件）

|  |  |  |  |
| --- | --- | --- | --- |
|  | 使用物化视图Force refresh | 使用物化视图Complete refresh | 使用物化视图Fast refresh |
| 时间 | 0.592s | 0.766s | 0.712s |

## 实验结论与分析

实验的结论符合我们的预期，由于刷新方式的不同导致刷新时间呈现出不同的变化趋势，我们分析得到的数据，可以看到使用强制刷新和完全刷新的时间基本不随 DML 操作数据量的改变而改变，基本保持在一个确定的时间范围内；而快速刷新的时间则是会随操作数据量的增大而增大。这主要是他们不同的刷新方式的执行原理造成的。

完全刷新：先把物化视图的数据全部删除，然后把基表的数据插入到物化视图中。当数据达到百万级别时，若原表更新了一条数据，完全刷新就得插入全部数据。这也就使得影响完全刷新执行速度的主要影响因素是基表的数据量，而并非 DML 操作的数据量。

快速刷新：保留物化视图的数据，然后基表的所有数据的变更记录到物化视图日志中。这就需要我们开启物化视图日志。由于是更具改变的记录的数量确定哪些数据需要刷新，因此该种刷新方式会受到的 DML 操作数量的影响。

## 实验心得

在具体实验中，我们发现过于复杂的查询语句是无法使用快速刷新且无法on commit时刷新，发现如果要创建一个复杂查询的物化视图有如下限制：

必须满足所有快速刷新物化视图都满足的条件；

物化视图查询的所有表必须建立物化视图日志，且物化视图日志必须满足下列限制：

包含物化视图查询语句中的所有列，包括 SELECT 列表中的列和WHERE 语句中的列；

必须指明 ROWID 和 INCLUDING NEW VALUES；

如果对基本的操作同时包括 INSERT、UPDATE 和 DELETE 操作（即不是只包含 INSERT 操作），那么物化视图日志应该包括 SEQUENCE。

允许的聚集函数包括：SUM、COUNT、AVG、STDDEV、VARIANCE、MIN 和 MAX；

必须指定 COUNT()；不加 COUNT()，建立快速刷新的物化视图会成功，但通过执行 EXPLAIN\_MVIEW 过程，可以发现，COUNT(\*)对于 INSERT 操作是可以快速刷新的，不过对于 UPDATE 和 DELETE 则会造成快速刷新的失败。

如果指明了除 COUNT 之外的聚集函数，则 COUNT(expr)也必须存在；

比如：包含 SUM(a)，则必须同时包含 COUNT(a)。

如果指明了 VARIANCE(expr)或 STDDEV(expr)，除了 COUNT(expr)外， SUM(expr)也必须指明；Oracle 推荐同时包括 SUM(expr\*expr)。

SELECT 列表中必须包括所有的 GROUP BY 列；

当物化视图属于下面的某种情况，则快速刷新只支持常规 DML 插入和直接装载，这种类型的物化视图又称为 INSERT-ONLY 物化视图；

物化视图包含 MIN 或 MAX 聚集函数；物化视图包含 SUM(expr)，但是没有包括 COUNT(expr)；物化视图没有包含 COUNT(\*)。

注意：如果建立了这种物化视图且刷新机制是 ON COMMIT 的，则会存在潜在的问题。当出现了 UPDATE 或 DELETE 语句，除非手工完全刷新解决这个问题，否则物化视图至此以后都不再自动刷新，且不会报任何错误。

如果包含 inline views、outer joins、self joins 或 grouping set，则兼容性的设置必须在 9.0 以上；

如果物化视图建立在视图或子查询上，则要求视图必须可以完全合并的。

如果没有外连接，则对 WHERE 语句没有限制。如果包含外连接，则要求 WHERE 语句只能包括 AND 连接和“=”操作。对于包含外连接的聚集物化视图，快速刷新支持 outer 表的修改。且 inter 表的连接列上必须存在唯一约束。

对于包含了 ROLLUP、CUBE、GROUPINGSET 的物化视图必须满足下列限制条件：

SELECT 语句列表中应该包含 GROUPING 标识符：可以是 GROUP BY 表达式中所有列的 GROUPING\_ID 函数，也可以是 GROUP BY 表达式中每一列的 GROUPING 函数；GROUPBY 不能产生重复的 GROUPING。

# 数据库连接模式实验

## 实验背景

Oracle 数据库提供了两种不同的连接模式，分别应对不同的使用场景。

专用服务器模式：是一个一对一的模式，即一个客户端产生一个服务器进程。该类服务器进程通过 TCP 或 TCPS 等直接建立连接，且此类服务器进程不为实例所有该类服务器进程一旦建立，直到退出和关闭该会话相关的资源才被释放。

共享服务器模式：配置共享模式后，可以使用一个或多个调度进程也可以使用一个或多个共享服务器进程，同时会在 SGA 共享池内开辟部分空间来用作队列的存储，包含请求队列、响应队列，当使用共享模式的连接时，服务器上的所有本地连接都会得到一个专用服务器，使用 IPC 进行连接。

二者各有优劣，使用共享服务器模式的优点：减少了实例中的进程数；增加了更多并发用户的数量；实现动态负载均衡；减少了空闲服务器进程数量；降低了对内存的使用。适用专用服务器模式的场景主要有：提交事务的为批处理；启动关闭或执行恢复使用 sysdba 时；在层架构中，专用模式具有更好的性能。适用共享服务器模式的场景主要有：多用于管理许多完成短事务会话的 OLTP 系统。

本项目需要支持多用户同时适用，对并发性有一定要求，因此我们将通过比较不同并发数下，两种数据库连接模式的响应速度，得出更适合此项目的数据库连接方式。

## 对应功能点

项目并发度支持

## 实验步骤

通过比较不同并发数下，两种数据库连接模式的响应速度，得出更适合此项目的数据库连接方式。

## 实验代码

我们通过 Java 程序开启多个线程，并使用锁，使得他们在同一时刻开始访问数据库，来模拟多个用户并发访问。

public class JavaTest {

final static SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

static final String DB\_URL = "jdbc:oracle:thin:@59.110.162.32:1521:orcl";

static final String USER = "sjs";

static final String PASS = "sjs@2020";public static void main(String[] args) {

CountDownLatch latch = new CountDownLatch(1);

// 模拟 n 个用户

List<Thread> threads = new ArrayList<>();

for (int i = 1; i <= 999; i++) {

Thread thread = new AnalogUser("user" + i, i, latch);

thread.start();

threads.add(thread);

}long start = System.currentTimeMillis();

// 计数器減一 所有线程释放 并发访问。

latch.countDown();threads.forEach(th -> {

try {

th.join();

} catch (Exception e) {

}

});

System.out.println("所有模拟请求结束: " + (System.currentTimeMillis() - start));

}public static class AnalogUser extends Thread {

String workerName;

int user\_id;

CountDownLatch latch;public AnalogUser(String workerName, int user\_id, CountDownLatch latch) {

super();

this.workerName = workerName;

this.user\_id = user\_id;

this.latch = latch;

}@Override

public void run() {

try {

latch.await(); // 一直阻塞当前线程，直到计时器的值为 0

} catch (InterruptedException e) {

e.printStackTrace();

}

request();// 发送请求

}public void request() {

String result = "";

System.out.println("模拟用户： " + workerName + " 开始发送模拟请at " + sdf.format(new Date()));

try {

Class.forName("oracle.jdbc.driver.OracleDriver");

Connection conn = DriverManager.getConnection(DB\_URL, USER, PASS);

PreparedStatement statement = conn.prepareStatement("select ID, REAL\_NAME from BOYING\_USER where ID = ?");

statement.setInt(1, user\_id);

ResultSet resultSet = statement.executeQuery();

resultSet.next();

result = resultSet.getString(2);

statement.close();

conn.close();

} catch (Exception e) {

e.printStackTrace();

}

System.out.println("模拟用户： " + workerName + " 模拟请求结束 at " + sdf.format(new Date()) + " 操作结果：" + result);

}

}

}

## 实验结果

|  |  |  |  |
| --- | --- | --- | --- |
| 并行线程数量 | 10(ms) | 100(ms) | 1000(ms) |
| 专业服务器模式 | 1085 | 1755 | 16358 |
| 共享服务器模式  (max\_shared\_servers=5) | 1067 | 1747 | 22103 |

## 实验结论

在连接 Oracle 数据库时，使用直连模式下，当会话树超过 1000 时，就会报错：java.sql.SQLException: Listener refused the connection with the following error:ORA-12519, TNS:no appropriate service handler found。导致服务器不能响应更多连接请求。

因而除非继续增大设置，否则无法继续实验。而在共享服务器模式下则不会出现此问题。

## 实验心得

数据库连接池负责分配、管理和释放数据库连接，它允许应用程序重复使用一个现有的数据库连接，而不是再重新建立一个；释放空闲时间超过最大空闲时间的数据库连接来避免因为没有释放数据库连接而引起的数据库连接遗漏。这项技术能明显提高对数据库操作的性能。

数据库连接池在初始化时将创建一定数量的数据库连接放到连接池中，这些数据库连接的数量是由最小数据库连接数制约。无论这些数据库连接是否被使用，连接池都将一直保证至少拥有这么多的连接数量。连接池的最大数据库连接数量限定了这个连接池能占有的最大连接数，当应用程序向连接池请求的连接数超过最大连接数量时，这些请求将被加入到等待队列中。数据库连接池的最小连接数和最大连接数的设置要考虑到下列几个因素：

1. 最小连接数

是连接池一直保持的数据库连接，所以如果应用程序对数据库连接的使用量不大，将会有大量的数据库连接资源被浪费。

2. 最大连接数

是连接池能申请的最大连接数，如果数据库连接请求超过此数，后面的数据库连接请求将被加入到等待队列中，这会影响之后的数据库操作。

3. 最小连接数与最大连接数差距

最小连接数与最大连接数相差太大，那么最先的连接请求将会获利，之后超过最小连接数量的连接请求等价于建立一个新的数据库连接。不过，这些大于最小连接数的数据库连接在使用完不会马上被释放，它将被放到连接池中等待重复使用或是空闲超时后被释放。

我们在使用连接池时应该注意不要超过 Oracle 服务器的设置。

# SGA调整实验

## 实验背景

系统全局区中主要影响性能的调整参数包括：结果缓存、共享池缓存大小。

过大过小的共享池都会严重影响系统性能。

共享池包括：数据字典高速缓存（用户资料、表空间等等）、库缓存（SQL和PL/SQL代码的已解析或已编译形式）。应主要关心库缓存，因为共享池中的内存分配算法倾向与把字典数据放在内存的时间比较长，而库缓存中的数据放在内存中的时间要短。

## 对应功能点

无

## 实验步骤

在配置新的数据库实例时，很难知道共享池缓存的正确大小。首先估算缓存大小，然后将我们所有的业务功能全部运行一遍，然后检查相关统计信息Library Cache Statistics以查看缓存配置不足还是配置过度。

查看共享SQL区的重用率，最好在95％以上，否则需要增加共享池的大小。

查看数据字典缓冲区的命中率，最好在95％以上，否则需要增加共享池的大小。

## SQL语句

select (sum(pins - reloads)) / sum(pins) "Lib Cache" from v$librarycache;

select (sum(gets - getmisses - usage - fixED)) / sum(gets) "Row Cache" from v$rowcache;

## 实验结果

12个PL/SQL情况下：

|  |  |  |
| --- | --- | --- |
| 共享池缓存大小 | 共享SQL区的复用率 | 数据字典缓冲区的命中率 |
| 50M | 99.6898538% | 75.2807136% |
| 400M | 99.6316464% | 98.6237841% |
| 800M | 99.6971681% | 98. 7507137% |

自动分配SGA情况下：虚拟机32G内存分配SGA大小为1408M，缓冲池大小为400M

|  |  |  |
| --- | --- | --- |
| PL/SQL个数 | 共享SQL区的复用率 | 数据字典缓冲区的命中率 |
| 12 | 99.6316464% | 98.6237841% |
| 48 | 98. 9315692% | 95. 2334566% |
| 120 | 98. 2376945% | 84. 1264321% |

## 实验结论

系统自动分配的SGA和缓冲池已经能够满足我们的需求，然而，对于较小内存的电脑，手动SGA和缓冲池大小还是有他的优势，因为系统默认分配为10%内存左右动态调整，可以看到当共享池只有50M 时，库缓存和数据字典缓冲区的命中率还是有明显偏低的，因为对于这样的机器可以手动调整SGA。

对于较大内存机器，完全不必手动SGA。但是如果我们系统中PL/SQL 非常多，又有许多预编译的语句， 共享SQL 区重用率小幅回升后下降，但还是可以接受的，但是数据字典缓冲区命中率迅速下降到84%。结论是当内存较小，又有很多存储的过程函数等等需要预编译的语句，那么需要手动增大SGA 大小。

# 批量读写实验

## 写入速度实验

### 实验背景

我们的插入实验针对 TICKET 表，可以模拟用户的买票行为，插入新的演出票数据。这里我们的数据从千级逐渐增加到千万级，对每一个数量级进行插入，得出统计时间结果。

### 功能对应点

购票

### 实验步骤

我们首先在 oracle 数据库中，建立好所有的表，我们添加 3000 余条数据到 TICKET 表，（此时进行了一次插入时间计时，为了之后的插入实验）接下来我们进行了一次查询，查询成功后记录时间，再插入 30000 余条数据……直到所有的千万级数据插入完毕，进行最后一次查询。

### SQL 语句（其中一例）

直接逐条插入

INSERT INTO BOYING\_ TICKET (ID, ORDER\_ID，SEAT\_ID, QR\_CODE\_URL)

VALUES (1923, 1690, 90815, 'https://tongji4m3.oss-cn-beijing.aliyuncs.com/1608898790.jpg');

无需回滚操作的优化插入

INSERT /\*+APPEND\*/ INTO BOYING\_ TICKET (ID, ORDER\_ID，SEAT\_ID, QR\_CODE\_URL)

VALUES (1923, 1690, 90815, 'https://tongji4m3.oss-cn-beijing.aliyuncs.com/1608898790.jpg');

### 实验结果

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 数据插入方式 | 千条 | 万条 | 十万条 | 百万条 | 千万条 |
| Oracle（使用 csv 导入） | 0.395s | 4.010s | 21.132s | 2min24s | 25min19s |
| Oracle（逐条  insert） | 2.091s | 22.625s | 3min47s | 32min33s | 5h11min |
| TimesTen（逐条  insert） | 2.465s | 22.773s | 3min43s | 33min17s | ------ |
| TimesTen（优化  insert） | 2.227s | 20.801s | 3min1s | 28min55s | 4h19min |

### 实验结论

数据库的性能对于简单的数据插入影响并不大，数据插入的主要性能关键在于磁盘 IO 效率。由于 TimesTen 本身兼容性不如 Oracle 好等原因，并不能支持 csv 数据批量导入，只能通过 insert 的方式插入数据，因此在大批量插入上 oracle 有优势，但在逐条插入的实际应用场景下两者性能是相近的。同时， 优化插入虽然一定程度上优化了数据写入，但是并不能体现出明显优势，在性能上依然比不上数据源批量导入。

## 读取速度实验

### 实验背景

我们的查询实验同样针对TICKET 表，我们这样来模拟用户根据票号查找指定演出票的场景，这个场景日常都在发生。尤其在演出开始前检票时，会很容易出现大量的二维码查询，这时我们就会一直收到向TICKET表中进行查询的请求。我们设计这个实验查看各种数据库应对此场景的能力。

### 对应功能点

查看票二维码

### 实验步骤

与插入实验步骤一致，该实验是插入实验的前置步骤，我们分批次插入各个数量级的数据，随即进行查询，然后分别进行计时。然后再次插入下一批数据并执行查询直到结束。

### SQL语句

select QR\_CODE\_URL from BOYING\_ TICKET

where id=370707

### 实验结果

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 数据库 | 千条 | 万条 | 十万条 | 百万条 | 千万条 |
| Oracle | <1ms | <1ms | 0.004s | 0.038s | 0.429s |
| TimesTen | <1ms | <1ms | 0.002s | 0.024s | 0.232s |

### 实验结论

在查询方面 timesten 由于内存读取速度更快等原因会更有优势，但是由于timesten 的查询优化的服务并不如 oracle 做的完善（例如后面即将讲到的缓存池，预编译等机制），oracle的缓存机制可以使得其在面对批量相同的服务时（重复的多次查询等）可能会有更好的性能表现。

# 数据类型实验

## 实验背景

在数据库中创建数据表的时候，我们需要定义表中所有字段的类型。ORACLE提供多种数据类型来需要。经常使用的数据类型如下表所示：

|  |  |  |  |
| --- | --- | --- | --- |
| 类型 | 含义 | 存储描述 | 备注 |
| CHAR | 固定长度字符串 | 最大长度2000bytes |  |
| VARCHAR2 | 可变长度的字符串, | 最大长度4000bytes | 可做索引的最大长度749 |
| NCHAR | 根据字符集而定的固定长度字符串 | 最大长度2000bytes |  |
| NVARCHAR2 | 根据字符集而定的可变长度字符串 | 最大长度4000bytes |  |
| DATE | 日期（日-月-年） | DD-MM-YY(HH-MI-SS),经过严格测试，无千虫问题 |  |
| FLOAT | 浮点数类型 | NUMBER(38)，双精度 |  |
| NUMBER(P,S) | 数字类型 | P为整数位，S为小数位 |  |

注：本实验主要是探究对于同样的数据，采用不同的数据类型存储会对数据库造成何种影响。

## 功能对应点

有关用户账号、演出的操作，包括存储。

## 实验步骤

1. 对于本项目而言，关于用户账号涉及到了ID 这个字段表示用户的ID，这是用户的唯一标识符，ID 的存储方式便十分重要，对于该字段有多种存储方式，主要VARCHAR 和NUMBER。我们将针对这一问题，进行如下实验，找出适合 ID 的存储类型。我们进行了两组实验，分别使用单表查询以及多表连接查询。
2. 演出在本项目中是非常关键的信息，演出表中的信息存储也涉及到了大量的文本字符串存储，存储类型可以是CHAR、VARCHAR2和NVARCHAR，CHAR的长度是固定的，而VARCHAR2的长度是可以变化的，nvarchar中，中文字符按两个字节计算，英文字符按一个字符计算；nvarchar2中，所有字符都按两个字节计算。不同的存储类型会对查询时间、占用存储空间等性能造成影响。我们将针对这一问题，进行如下实验，找出不同存储方式的差异和找到最适合的存储方式

## SQL语句

--1.1

SELECT \* FROM BOYING\_USER WHERE GENDER = 1;

--1.2

SELECT \* FROM BOYING\_USER c,BOYING\_ORDER d WHERE (c.ID=d.USER\_ID)AND(PAYMENT='PALPAY')

--2.1

SELECT \* FROM BOYING\_SHOW;

## 实验结果

### 账号存储类型的影响

|  |  |  |
| --- | --- | --- |
| ID类型 | 单表查询时间 | 多表查询时间 |
| NUMBER | 1.217s | 2.695s |
| VARCHAR(9) | 1.071s | 2.597s |
| VARCHAR(255) | 1.555s | 3.015s |

### 演出存储类型的影响

|  |  |  |  |
| --- | --- | --- | --- |
| 存储类型 | 查询时间 | 存储大小 | 数据导入时间 |
| CHAR | 43ms | 80mb | 3min20s |
| VARCHAR(255) | 34ms | 11mb | 32s |
| NVARCHAR(255) | 35ms | 15mb | 29s |

## 实验结论

使用 number 进行存储在查询、插入、存储空间大小的效果均优于char 进行存储。在Oracle 中，当字符长度较短时，应尽量使用VARCHAR2替代CHAR存储，以节约空间和时间。

# 成员贡献

# 附录A. 图目录

# 附录B. 表目录