Nov 9, 2021

版本: 1.0

同济大学 TONGJI UNIVERSITY

内存数据库项目报告

归宿——一个民宿预定平台

指导教授：**袁时金**

1851231 王立友

1852612 周涵嵩

1952560 安江涛

1953068 高天宸

1954098 香宁雨

In-memory Database

Tongji University

School of Software Engineering

目录

实验一 数据库方案对比实验 5

1.1 实验目的 5

1.2 实验方案 5

1.2.1 订单信息查询 5

1.2.2 用户信息查询 6

1.3 实验设计思考 6

1.4 实验设计总结 6

实验二 查询优化实验 7

2.1 实验目的 7

2.2 实验方案 7

2.2.1 房东订单数据查询 7

2.2.2 房源订单数据查询 9

2.2.3 房东订单信息查询 10

2.3 实验设计思考 10

2.4 实验设计总结 11

实验三 索引优化实验 12

3.1 实验目的 12

3.2 实验方案 12

3.2.1 针对评论等级建立索引 12

3.3 实验设计思考 13

3.4 实验设计总结 15

实验四 批量写入实验 16

4.1实验目的 16

4.2实验方案 16
4.2.1 单表批量写入实验 16
4.2.2 多表批量写入实验 17
4.3实验设计思考 18
4.4实验设计总结 18
实验五 数据报表实验 20
5.1 实验目的 20
5.2 实验方案 20
5.2.1 用户查看民宿与订单统计数据报表 20
5.3 实验设计思考 22
5.4 实验设计总结 23
实验六 SQL脚本与过程对比实验 24
6.1 实验目的 24
6.2 实验方案 24
6.2.1 生成订单统计报表 24
6.3 实验设计思考 25
6.4 实验设计总结 26
实验七 添加冗余表 27
7.1实验目的 27
7.2实验方案 27
7.3实验设计思考 28
7.4实验设计总结 28
实验八 增加冗余操作 29
8.1实验目的 29
8.2实验方案 29
8.2.1 增加冗余列 29
8.3实验设计思考 31

8.4实验设计总结 31
实验九 冷热对比实验 32
9.1 实验目的 32
9.2 实验方案 32
9.2.1 根据容量进行冷热数据区分 32
9.2.2 对冷热数据比例进行划分 32
9.3 实验设计思考 32
9.4 实验设计总结 32
实验十 事务对插入的影响实验 33
10.1 实验目的 33
10.2 实验方案 33
10.3 实验设计思考 34
10.4 实验设计总结 34
实验十一 数据分区实验 35
11.1 实验目的 35
11.2 实验方案 35
11.2.1 范围分区：查询时间范围内的订单信息 36
11.2.2 列表分区：目的性对评论数据进行筛选查询 37
11.3 实验设计思考 37
11.4 实验小结 37

实验一 数据库方案对比实验 1

在不同数据量时，对于不同数据库的选择可能影响数据库查询的速度以及数据库的性能。因此可以对不同数据库进行其性能对比实验。

1.1 实验目的

对于TimesTen和Oracle in Memory而言，TimesTen的所有数据都会被加载到内存中，Oracle in Memory会将指定表加载进内存中，而对于Oracle而言，数据的处理可能是先从辅存读入内存再进行处理。在小数据量和大数据量不同时，TimesTen、Oracle in Memory和Oracle的表现可能不同。所以我们对TimesTen、Oracle in Memory和Oracle分别进行查询操作，分析三者在相同数据量时性能的差异。在归宿中存在大量对数据的查询业务，而首要任务是对数据库进行选择。我们将对比Oracle和TimesTen、Oracle in Memory在千万级表和万级表之间的查询速度来进行对数据库的选择。

1.2 实验方案

1.2.1 订单信息查询

数据表：Order

Order表为本项目中千万级数据表，我们通过查询所有订单的数量来进行Oracle和TimesTen在千万级数据表上性能的比较。

查询SQL语句如下：

```
select count(*)
from t_order;
```

实现结果：

数据库	查询时间
ORACLE	0. 538s
TIMESTEN	0. 324s
ORACLE IN MEMORY	0. 252s

Tab. 1.1: 数据库对比查询实验实验记录表格

1.2.2 房源信息查询

数据表：Stay

Stay表为本项目中百万级数据表，我们通过查询所有订单的数量来进行Oracle、Oracle in Memory和TimesTen在百万级数据表上性能的比较。

查询SQL语句如下：

```
select count(*)
from t_stay;
```

实现结果如下：

数据库	查询时间
ORACLE	0.340s
TIMESTEN	0.183s
ORACLE IN MEMORY	0.151s

Tab. 1.2: 数据库对比查询实验记录表格

1.3 实验设计思考

对于数据库查询而言，数据加载进内存和没有加载进内存查询速度会有很大差别，加载进内存后会减少读写磁盘的开销，但同时会对内存的要求比较高，也就是对硬件要求较高。

1.4 实验设计总结

通过实验结果可以看出TimesTen和Oracle in Memory的查询效率确实高于Oracle，说明当数据加载入内存后，查询时间会大幅减少。对于千万级表和百万级表而言，千万级表的查询速度要慢于百万级表，符合实验设计的思路。

同时，TimesTen和Oracle in Memory相对于Oracle的查询效率在千万级表上的优化效率没有在百万级表上的优化效率高，是通过将数据存储于内存中从而减少了对磁盘读写操作的次数实现的优化过程。

实验二 查询优化实验

2

通常对于同一个目标，我们会有多种SQL查询语句的写法，但是各种写法的实际执行效率并不相同。为此，我们针对本系统的主要的查询业务，来进行SQL语句级别的优化实验。

2.1 实验目的

订单管理系统连接的是卖家用户与买家用户，对于这两方用户的使用来说，都存在大量的查询业务，这也是整个系统并发量最大的业务。所以我们需要对查询过程进行详细的优化设计，来提升用户的使用体验。

2.2 实验方案

2.2.1 房东订单数据查询

数据表：order, order_room, room, stay

优化策略：房东订单信息查询涉及到多表查询，很多时候我们都需要用join来拼接多个表。在join操作中的优化规则是：数据行较少的表在左边，数据行较多的表在右边。在join当中，是使用左边表的每一个数据行去扫描右边的整个表的所有数据行，所以虽然总的匹配次数是相同的，但是如果左边表数据行很多，则需要加载右边的整个表很多次，使用小表驱动大表主要是减少这个次数，来提高性能，在本实验中，我们将在Oracle数据库中分别对于 Hash Join 和 Nested Loop 两种表连接方式，分别按照不同的顺序进行表的连接并进行查询。

在Hash join中，优化器使用两个表中较小的表（通常是小一点的那个表或数据源）利用连接键在内存中建立散列表，将列数据存储到hash列表中，然后扫描较大的表，同样对JOIN KEY进行HASH后探测散列表，找出与散列表匹配的行。

在使用Hash Join的情况下，优化前查询SQL语句如下：

```
select count(distinct(order_id))
from t_order_room natural join t_order natural join t_room  natural join t_stay
where host_id = 634225;
```

在使用Hash Join的情况下，优化后查询SQL语句如下：

```
select count(distinct(order_id))
from t_stay natural join t_room natural join t_order  natural join t_order_room
where host_id = 634225;
```

Nested loop的工作方式是循环从一张表中读取数据(驱动表outer table)，然后访问另一张表（被查找表 inner table ,通常有索引）。驱动表中的每一行与inner表中的相应记录JOIN。类似一个嵌套的循环。

在使用Nested Loop的情况下，优化前查询SQL语句如下：

```
select count(distinct(t_order.order_id))
from  t_order_room, t_order, t_room, t_stay
where t_order_room.order_id = t_order.order_id and
      t_order_room.room_id = t_room.room_id and
      t_order_room.stay_id = t_room.stay_id and
      t_room.stay_id = t_stay.stay_id and
      t_stay.host_id = 634225;
```

在使用Nested Loop的情况下，优化后查询SQL语句如下：

```
select count(distinct(t_order.order_id))
from  t_stay, t_room, t_order, t_order_room
where t_order_room.order_id = t_order.order_id and
      t_order_room.room_id = t_room.room_id and
      t_order_room.stay_id = t_room.stay_id and
      t_room.stay_id = t_stay.stay_id and
      t_stay.host_id = 634225;
```

由于TIMESTEN中不支持natural join操作，因此不进行TIMESTEN的Hash Join查询演示，实验结果如下：

实验结果如下：

数据库	优化前	优化后
ORACLE	3.866s	3.855s
TIMESTEN	–	–
ORACLE IN MEMORY	0.294s	0.282s

Tab. 2.1: 房东订单数据查询优化实验(Hash Join)部分

数据库	优化前	优化后
ORACLE	3.864s	3.858s
TIMESTEN	25.826s	25.44s
ORACLE IN MEMROY	0.279s	0.267s

Tab. 2.2: 房东订单数据查询优化实验(Nested Loop)部分

2.2.2 房源订单数据查询

数据表：order, order_room, room, stay

优化策略： Oracle 采用自下而上或自右向左的顺序解析 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。

优化前查询SQL语句如下：

```
select count(distinct(t_order.order_id))
from  t_stay, t_room, t_order, t_order_room
where t_stay.host_id = 634225 and
      t_order_room.stay_id = t_room.stay_id and
      t_order_room.room_id = t_room.room_id and
      t_room.stay_id = t_stay.stay_id and
      t_order_room.order_id = t_order.order_id;
```

优化后查询SQL语句如下：

```
select count(distinct(t_order.order_id))
from  t_stay, t_room, t_order, t_order_room
where t_order_room.order_id = t_order.order_id and
      t_order_room.room_id = t_room.room_id and
      t_order_room.stay_id = t_room.stay_id and
      t_room.stay_id = t_stay.stay_id and
      t_stay.host_id = 634225;
```

实验结果如下：

数据库	优化前	优化后
ORACLE	3.857s	3.854s
TIMESTEN	26.451s	26.206s
ORACLE IN MEMORY	0.283s	0.282s

Tab. 2.3: 房源订单数据查询优化实验

2.2.3 房东订单信息查询

数据表： order, order_room, room, stay

优化策略：查询房东订单的信息，查询结果需要对订单ID字段去重。去重查询可以有两种查询方案：通过 select distinct 进行查询；先进行 group by，再进行 select。通过比较两种查询方案的实验结果，分析性能上的差异。

使用 distinct 的SQL语句如下：

```
select distinct order_id
from t_stay natural join t_room natural join t_order  natural join t_order_room
where host_id = 634225;
```

使用 group by 的查询SQL语句如下：

```
select order_id
from t_stay natural join t_room natural join t_order  natural join t_order_room
where host_id = 634225
group by order_id;
```

由于TIMESTEN中不支持natural join操作，因此不进行TIMESTEN的去重查询演示，实验结果如下：

数据库	优化前	优化后
ORACLE	3.855s	3.847s
TIMESTEN	–	–
ORACLE IN MEMORY	0.281s	0.277s

Tab. 2.4: 房东订单信息查询优化实验

2.3 实验设计思考

在join当中，是使用左边表的每一个数据行去扫描右边的整个表的所有数据行，所以虽然总的匹配次数是相同的，但是如果左边表数据行很多，则需要加载右边的整个表很多次，使用小表驱动大表主要是减少这个次数，来提高性能。

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。

ORACLE8到11G的在线文档，关于SQL优化相关章节，没有任何文档说过where子句中的条件对SQL性能有影响，到底哪种观点是对的，还是需要实验来证明。

2.4 实验设计总结

通过实验结果可以看出，join表顺序实验和Where条件顺序实验优化前与优化后的sql查询时间几乎没有差距，猜测ORACLE已经对此做了自动优化。

使用 distinct 和 group by 都可以实现去重操作，从实验结果来看，在 1000w 级的数据表上进行查询，二者的查询性能差别较小，group by 会比 distinct更快。 distinct, group by两种方式本质就是时间与空间的权衡。

distinct需要将colA中的所有内容都加载到内存中，大致可以理解为一个hash结构，key自然就是colA的所有值。因为是hash结构，那运算速度自然就快。最后计算hash中有多少key就是最终的结果。但是在海量数据环境下，需要将所有不同的值都存起来，内存消耗大，会导致jvm更加频繁的进行内存交换，进而拖慢查询速度。

group by的实现方式是先将colA排序。排序大家都不陌生，拿最常见的快排来说，时间复杂度为O(nlogn),而空间复杂度只有O(1)。即使数据量很大，group by的内存占用也基本不会受到影响，在空间和时间的权衡下，于当前的实验环境中，group by 会稍优于 distinct。

在本次实验中，TIMESTEN数据库的总查询时间均比同样情况下Oracle数据库长。在查阅资料后，分析原因是SQL语句较为复杂，涉及多个大型数据表的联表查询，内存占用已经超出服务器内存容量，内存数据库的优势发挥不出来，而得益于Oracle数据库强大的优化器，其执行速度要显著快于TIMESTEN数据库。

实验三 索引优化实验

3

3.1 实验目的

根据实际的应用场景，通过对不同字段进行组合，在 Oracle 、Oracle in Memory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。

3.2 实验方案

3.2.1 针对评论等级建立索引

查询SQL语句：

```
select count(*)
from T_COMMENT
where comment_grade = 1
```

插入SQL语句：

```
insert into T_COMMENT
values (COMMENT_INC.nextval, null, 1, to_date('2021/11/08', 'yyyy-mm-dd'), 1)
```

TimesTen部分：

无索引

无需额外操作

范围索引

```
drop index comment_index;
create index comment_index
on T_COMMENT(comment_grade)
```

位图索引

```
drop index comment_index;
create bitmap index comment_index
on T_COMMENT(comment_grade)
```

哈希索引

```
drop index comment_index;
create hash index comment_index
on T_COMMENT(comment_grade)
```

在创建索引之后分别进行查询和插入操作。

实验结果如下：

索引类型	查询操作	插入操作
无索引	0.895s	0.094s
范围索引	0.399s	0.049s
位图索引	0.066s	0.009s
哈希索引	0.270s	0.044s

Tab. 3.1: 针对评论等级建立索引的实验记录（TimesTen）

Oracle部分：

在创建索引之后分别进行查询和插入操作。

实验结果如下：

索引类型	查询操作	插入操作
无索引	2.919s	0.305s
范围索引	1.352s	0.044s
位图索引	0.197s	0.163s

Tab. 3.2: 针对评论等级建立索引的实验记录（Oracle）

Oracle in Memory部分：

在创建索引之后分别进行查询和插入操作。

实验结果如下：

索引类型	查询操作	插入操作
无索引	0.035s	0.039s
范围索引	0.034s	0.038s
位图索引	0.034s	0.037s

Tab. 3.3: 针对评论等级建立索引的实验记录（Oracle in Memory）

3.2.2 针对评论时间建立索引

查询SQL语句：

```
select count(*)
from T_COMMENT
where COMMENT_TIME > to_date('2021-01-01 00:00:00','yyyy-MM-dd HH24:MI:SS');
```

无索引

无需额外操作

范围索引

```
drop index comment_time_index;
create index comment_time_index
on T_COMMENT(COMMENT_TIME);
```

位图索引

```
drop index comment_time_index;
create bitmap index comment_time_index
on T_COMMENT(COMMENT_TIME);
```

Oracle部分：

在创建索引之后进行查询操作。

实验结果如下：

索引类型	查询操作
无索引	2.658s
范围索引	1.545s
位图索引	2.652s

Tab. 3.2: 针对评论时间建立索引的实验记录（Oracle）

Oracle in Memory部分：

在创建索引之后进行查询操作。

实验结果如下：

索引类型	查询操作
无索引	0.040s
范围索引	0.038s
位图索引	0.043s

Tab. 3.3: 针对评论时间建立索引的实验记录（Oracle in Memory）

3.1 实验设计思考

不同的索引对于不同的查询和插入操作以及不同的数据格式有不同的优化效果，面对不同的需求以及数据格式应该设计合适的索引等优化策略。另外在使用索引时应考虑其适合的应用场景，使用得恰当会增加应用的效率，反之可能会适得其反。

3.2 实验设计总结

通过实验结果可以看出，面对评论等级的数据格式的查询和插入需求时，位图索引更为合适，范围索引和哈希索引在减少执行时间上效果并不如位图索引好，这是因为对评论等级进行查询优化实验中仅对 comment_grade 为某一特定值进行查询统计，而数据中 comment_grade 仅有五个取值，范围很小，因此范围索引和哈希索引并不能发挥出它的优势，而这样的数据格式十分适合位图索引。但位图索引仍有缺点，在更新操作时会将对应值的向量加锁，导致在大量插入和更新操作时效率变慢，但是根据实验结果，单次插入操作效果很好，因此根据应用场景可以选择不同的索引来进行优化。而对于对评论时间进行查询优化实验中，评论时间这一字段本身具有多个范围值，通过范围索引可以很好的对时间信息进行筛选(B+树)，实验结果也印证了范围索引的优势。

Oracle in Memory的优化效果也很理想，可以看出预先将数据加载放入内存的操作可以行之有效的提高数据查询速度，另一方面，从结果角度而言，不同索引实现的效果并无太大的区分度，可以看出对于内存中查找速度而言，索引优化的程度是很有限的，在硬盘中进行索引优化是通过极大减少了I/O操作的次数，从而减少查询操作时间。

实验四 批量写入实验

4

本系统会有集中性的高并发场景。因此会需要进行大规模批量写入实验，便于开展对多线程效果的测试，其实验结果也是有效评估数据库性能的重要指标。

4.1 实验目的

针对用户收藏房源这一业务情景，进行大规模单表写入实验，测试系统在单线程和多线程环境下的操作用时，从而评估系统的单表批量写入能力。

针对用户下单房源这一业务情景，进行大规模多表插入实时更新实验，测试系统分别在单线程和多线程环境下的操作用时，从而评估系统的多表批量插入更新能力。

利用多线程实现数据库查询时，不同的线程数会带来不同的查询性能。通过进行线程数的对比实验，探究不同业务情景下线程数对数据库性能的影响。

4.2 实验方案

4.2.1 单表批量写入实验

业务情景：用户收藏房源时，需要向收藏夹单表导入大量数据。

实验说明：设计不同的线程数进行实验，对比不同线程方案的实现。

数据表：Favorites

实验代码如下：

```
CREATE OR REPLACE PROCEDURE INSERT_INTO_FAVORITE(customer_id IN NUMBER,
favorite_name IN VARCHAR2)
IS
BEGIN
if customer_id IS NULL then
  return;
end if;
INSERT INTO T_FAVORITE VALUES(FAVORITE_INC.nextval, customer_id, favorite_name);
commit;
END;
CREATE OR REPLACE PROCEDURE MULTI_INSERT_INTO_FAVORITE(job_count IN NUMBER)
IS
tmp_jobno NUMBER;
BEGIN
FOR I in 1..job_count LOOP
```

```
  dbms_job.submit(tmp_jobno, 'begin INSERT_INTO_FAVORITE(I, test_name_'||I||');
end; ');
END LOOP;
END;
```

实验结果：

线程数	1,000条	10,000条	100,000条
1（单线程）	3.114s	32.077s	329.901s
2（多线程）	1.991s	20.260s	201.554s
3（多线程）	1.294s	13.309s	129.710s
4（多线程）	1.010s	10.971s	103.048s
5（多线程）	0.891s	9.338s	95.144s

Tab. 4.1: 单表批量写入实验记录表格

4.2.2 多表批量写入实验

业务情景：用户下单房源时，需要向订单表、房间表、房源表级连写入更新大量数据。

实验说明：设计不同的线程数进行实验，对比不同线程方案的实现。

数据表：Order、Room、Stay

实验代码如下：

```
CREATE OR REPLACE PROCEDURE BUY_INSERT_TEST(
customer_id IN NUMBER,
order_time IN DATE,
customer_num IN NUMBER,
total_price IN NUMBER,
order_status IN NUMBER,
room_id IN NUMBER,
start_time IN DATE,
end_time IN DATE,
price IN NUMBER,
)
IS
order_id NUMBER;
stay_id NUMBER;
stay_cap NUMBER:
BEGIN
INSERT INTO T_ORDER VALUES(ORDER_INC.nextval, customer_id, order_time,
customer_num, total_price, order_status);
SELECT o.order_id INTO order_id FROM T_ORDER o WHERE o.order_time = order_time;
SELECT s.stay_id, s.stay_capacity INTO stay_id, stay_cap FROM T_STAY s WHERE
s.room_id = room_id;
stay_cap = stay_cap - 1;
```

```
INSERT INTO T_ORDER_ROOM VALUES(order_id, stay_id, room_id, start_time, end_time,
price);
UPDATE T_STAY s SET s.stay_capacity = stay_cap WHERE s.room_id = room_id;
commit;
END;

CREATE OR REPLACE PROCEDURE MULTI_BUY_INSERT_TEST(job_count IN NUMBER)
IS
tmp_jobno NUMBER;
BEGIN
FOR I in 1..job_count LOOP
  dbms_job.submit(tmp_jobno, 'begin BUY_INSERT_TEST(I, TEST_CUSTOMER_ID,
TEST_ORDER_TIME, TEST_CUSTOMER_NUM, TEST_TOTAL_PRICE, TEST_ORDER_STATUS,
TEST_ROOM_ID, TEST_START_TIME, TEST_END_TIME, TEST_PRICE); end; ');
END LOOP;
END;
```

实验结果：

线程数	1,000条	10,000条	100,000条
1（单线程）	5.914s	60.611s	595.370s
2（多线程）	3.718s	38.900s	392.742s
3（多线程）	2.718s	27.090s	266.504s
4（多线程）	2.196s	21.759s	223.129s
5（多线程）	1.961s	19.477s	207.814s

Tab. 4.2: 多表批量写入实验记录表格

4.3 实验设计思考

有时系统性能的瓶颈在于某一效率特别慢的功能，此时如果将该功能改为多线程进行，将极大提高系统的性能。例如在购物网站“秒杀”时刻，大量用户的购买行为将导致大量订单的产生，而订单的生成与许多表都相关，因此订单相关操作的效率相较于只涉及一个表的操作来说是相对较慢的，本实验通过测试进行多线程插入订单表及更新和插入相关表来探索多线程操作对于性能提升的效果。

4.4 实验设计总结

通过实验可以看出，无论是对于单表的插入还是多表联合插入操作，使用多线程都能有效提升操作的效率。

细致来看，对于单表操作，由于插入操作不涉及与其它表的交互，因此可以认为各个操作是独立互不影响的，从效率提升上来看也是如此，随着线程数的增加，单表效率提升的百分比要大于多表联合插入效率提升的百分比。

对于多表操作，由于一些插入操作涉及同一表的操作，因此可能会存在冲突导致效率降低，另一方面多表联合插入操作单个操作的时间较长，因此使用多线程可以避免某一操作时间显著长于其它操作导致整体操作效率的降低。从实验结果也可以看出，随着线程数的增加，多线程操作效率提升的百分比在逐渐降低。

同时通过观察实验结果可以看出，虽然增加线程可以减少总体操作时间，但是单位时间内单个线程完成的操作数在减少，由此可以看出优化的时间幅度有所下降。

实验五 数据报表实验

5

数据报表实验涉及多表联合查询，并且需要配合使用聚合函数对表字段进行统计分析。因此大数据量下数据报表查询操作可以很好地反映数据库查询与优化性能。

5.1 实验目的

一方面民宿预定平台具有用户查看个人行程住宿统计信息的功能，提供用户订单交易统计信息、与出行住宿等统计信息，另一方面民宿预定平台提供给民宿房主查看统计个人民宿出租收益统计数据的信息，使老板根据民宿的出租情况与利益对民宿的基础设施与价格进行调整。这两方面功能都是以数据报表为基础实现，因此通过对两类功能实验进行性能对比分析。

5.2 实验方案

5.2.1 用户查看民宿与订单统计数据报表

业务情景：根据用户的标识信息，筛选出给定时间范围(以月份为基本单位)内该用户的总订单数量、总订单金额、最大交易金额订单信息、最大交易金额相关民宿信息、旅居民宿次数最多的省份等信息，根据以上筛选与统计结果生成数据报表返回用户。

返回数据格式如下：

```
{
    "total_order_num":100,
    "total_order_amount":10045.8,
    "maximum_amount_order":{
        "order_id":100055,
        "order_time":"2021-10-11",
        "order_total_amount":2000
    },
    "maximum_amount_stay":[{
        "stay_id":100054,
        "room_id":201,
        "room_price":199,
        "room_photo":"xxx.png",
        "start_time":"2021-11-11",
        "end_time":"2021-11-14"
    },{
        "stay_id":100054,
        "room_id":202,
        "room_price":399,
        "room_photo":"xxx.png",
        "start_time":"2021-11-11",
```

```
        "end_time":"2021-11-14"
    }],
    "maximum_time_province":"上海"
}
```

核心实现代码为：

```
CREATE OR REPLACE FUNCTION CUSTOMER_ORDER_REPORT(c_id NUMBER)
RETURN t_type_order_report
AS
    a_type_order_report t_type_order_report:=t_type_order_report();
    i number :=0;
BEGIN
    for v_result in (
        SELECT T_ORDER.customer_id,COUNT(T_ORDER.customer_id) as ORDER_COUNT,
SUM(T_ORDER.total_price) as order_total_price, MAX(T_ORDER.total_price) as
order_max_price
        FROM T_ORDER
        WHERE T_ORDER.customer_id = c_id
        GROUP BY T_ORDER.customer_id) loop
        a_type_order_report.extend;
        i := i+1;
        a_type_order_report(i) :=
type_order_report(v_result.customer_id,v_result.order_count,v_result.order_total_
price,v_result.order_max_price);
    end loop;
    return a_type_order_report;
END;
```

```
CREATE OR REPLACE FUNCTION CUSTOMER_PROVINCE_REPORT(c_id NUMBER)
RETURN t_type_province_report
AS
    a_type_province_report t_type_province_report:=t_type_province_report();
    i number :=0;
BEGIN
    for v_result in (
    SELECT T_ORDER.customer_id, T_STAY.stay_province, COUNT(T_STAY.stay_province)
as province_count
    FROM T_ORDER,T_ORDER_ROOM,T_STAY
    WHERE T_ORDER.customer_id = c_id AND T_ORDER.order_id = T_ORDER_ROOM.order_id
and T_ORDER_ROOM.stay_id = T_STAY.stay_id
    HAVING COUNT(T_STAY.stay_province)=
    (SELECT MAX(COUNT(T_STAY.stay_province)) FROM T_ORDER,T_ORDER_ROOM,T_STAY
WHERE T_ORDER.customer_id = 1 AND T_ORDER.order_id = T_ORDER_ROOM.order_id and
T_ORDER_ROOM.stay_id = T_STAY.stay_id GROUP BY T_STAY.stay_province)
    GROUP BY T_ORDER.customer_id,T_STAY.stay_province
    ) loop
        a_type_province_report.extend;
        i := i+1;
```

```
        a_type_province_report(i) :=
type_province_report(v_result.customer_id,v_result.stay_province,v_result.provinc
e_count);
    end loop;
    return a_type_province_report;
END;
```

```
CREATE OR REPLACE FUNCTION CUSTOMER_MAX_ORDER_REPORT(c_id NUMBER, m_price
NUMERIC)
RETURN t_type_max_order_report
AS
    a_type_max_order_report t_type_max_order_report:=t_type_max_order_report();
    i number :=0;
BEGIN
    for v_result in (
    SELECT T_STAY.stay_name, T_ROOM.room_id, T_ROOM.room_price,
T_ROOM.room_photo, T_ORDER_ROOM.start_time, T_ORDER_ROOM.end_time
    FROM T_ORDER_ROOM,T_ORDER,T_ROOM,T_STAY
    WHERE T_ORDER.customer_id = c_id and t_order.total_price = m_price and
T_ORDER_ROOM.order_id = T_ORDER.order_id
    and T_ORDER_ROOM.room_id = T_ROOM.room_id and T_ORDER_ROOM.stay_id =
T_STAY.stay_id and T_ROOM.stay_id = T_ORDER_ROOM.stay_id
    ) loop
        a_type_max_order_report.extend;
        i := i+1;
        a_type_max_order_report(i) :=
type_max_order_report(v_result.stay_name,v_result.room_id,v_result.room_price,v_r
esult.room_photo,v_result.start_time,v_result.end_time);
    end loop;
    return a_type_max_order_report;
END;
```

实现结果如下：

数据库	查询时间
ORACLE 封装查询	13.089s
ORACLE	7.862s
TIMESTEN	1.481s
ORACLE IN MEMORY 封装查询	0.0543s

Tab. 5.1: 用户查看民宿与订单统计数据报表实验记录表格

5.3 实验设计思考

数据报表实验设计目的在于处理复杂的业务逻辑需求，其中需要多方面对数据存储、数据读取的优化与设计。另一方面，数据报表实验中可能遇到的问题在于多个统计量的查询与运算的过程会对效率有很大的损耗，需要利用存储过程、函数以及对底层表优化实现查询目标。

5.4 实验设计总结

通过数据报表实验结果可以看出，TimesTen很适合于这种对于复杂情况下的查询工作，数据报表的查询在本实验案例中是通过多种不同的查询组合拼接到最后的数据报表，相对应来看，对于每个单元的查询,TimesTen具有良好的表现能力，但是对于复杂的类型声明，TimesTen不支持自定义类型返回结果，因此对于报表的封装能力比较差。

虽然TimesTen对这种复杂情况下的查询工作很合适，但是Oracle in Memory对这种复杂情况下的查询工作优化程度更高，同时，因为Oracle in Memory和Oracle的语法相同，所以Oracle in Memory也支持对于复杂类型的声明以及查询，同时这种封装查询远远快于普通的Oracle数据封装查询。

综上所述，TimeTen适合于对复杂情况下的查询，但是在语法上存在一定程度的限制，例如，不支持自定义类型，不支持复杂的嵌套聚合函数查询，但是相对应查询速度卓越。另一方面，Oracle数据库查询速度相对较慢，但是支持PL/SQL的良好封装，可以通过整体查询提供良好的服务，Oracle in Memory支持复杂的嵌套聚合函数查询，查询速度极其优越，在项目中采用Oracle in Memory无疑大大加快了执行的效率。

因此可以看出，Oracle in Memory在复杂的，多表联合的过程查询可以极大的提高SQL语句执行效率，从结果而看具有显著的提升效果。

实验六 SQL 脚本与过程对比实验 6

SQL脚本是点对点运行的，且未被预编译；存储过程是预编译并存储在服务器上以供重用的一组SQL命令集，相比之下，存储过程的运行性能会优于SQL脚本。

6.1 实验目的

通过分别比较Oracle、Oracle in Memory和TimesTen数据库下SQL脚本和存储过程带来的处理时间差异，分析两种方法的性能表现。

6.2 实验方案

6.2.1 生成订单统计报表

业务情景:统计平台订单的平均成交金额、热门地区

数据表：t_order, t_order_room, t_ stay

优化策略：使用存储过程代替单纯的SQL脚本

优化前：

```
select avg(total_price)
from t_order;

select stay_province, count(*) as order_num
from t_order join t_order_room on t_order.order_id=t_order_room.order_id
join t_stay on t_order_room.stay_id=t_stay.stay_id
group by stay_province
having count(*)>10000;
```

优化后：

```
CREATE OR REPLACE PROCEDURE get_price_report
IS
v_sql VARCHAR2(300);
v_price NUMBER(20,10);
csr SYS_REFCURSOR;
BEGIN
v_sql:='select avg(total_price) as v_price
from t_order';
OPEN csr FOR v_sql;
```

```
  LOOP
    FETCH csr INTO v_price;
    EXIT WHEN csr%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('v_price''' || v_price || '''  ');
  END LOOP;
  CLOSE csr;
END;

CREATE OR REPLACE PROCEDURE get_order_report
IS
v_sql VARCHAR2(300);
v_province VARCHAR2(100);
v_order_num VARCHAR2(100);
csr SYS_REFCURSOR;
BEGIN
v_sql:='select stay_province, count(*) as order_num
from t_order join t_order_room on t_order.order_id=t_order_room.order_id
join t_stay on t_order_room.stay_id=t_stay.stay_id
group by stay_province
having count(*)>10000';
OPEN csr FOR v_sql;
  LOOP
    FETCH csr INTO v_province, v_order_num;
    EXIT WHEN csr%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('v_province=''' || v_province || '''  ' ||
                         'v_order_num=''' || v_order_num || '''');
  END LOOP;
  CLOSE csr;
END;

call get_price_report();

call get_order_report();
```

实验结果如下：

数据库	SQL语句	存储过程
ORACLE	12.17s(2.727s+9.443s)	10.184s(2.652s+7.532s)
TIMESTEN	29.428s(1.286s+28.142s)	29.344s(0.837s+28.507s)
ORACLE IN MEMORY	0.714s(0.382s+0.332s)	0.733s(0.391s+0.342s)

Tab. 6.3: 生成统计报表实验记录

6.3 实验设计思考

SQL脚本是即开即用，用完即删的一次性过程，而存储过程在第一次使用时被载入内存，下次使用会大大加快执行速度。SQL脚本只能执行单一操作，而存储过程可以互相调用，引用其他存储过程，更加灵活，使用更加方便，但修改起来更加困难。

6.4 实验设计总结

通过数据报表实验结果可以看出，存储过程的执行时间要略短于SQL语句，这也验证了存储过程需要编译，且编译后执行速度比SQL语句快。

在本次实验中，TIMESTEN数据库的总查询时间均比同样情况下Oracle数据库长。在查阅资料后，分析原因是第二个SQL语句较为复杂，涉及三个大型数据表的联表查询，得益于Oracle数据库强大的优化器，其执行速度要显著快于TIMESTEN数据库。同时， 对于第一个简单SQL语句的执行，TIMESTEN的速度则比Oracle数据库要快。TIMESTEN是为响应时间极快的小事务而设计的内存数据库，因此从应用上看，应该避免批量操作，减少单次事务读取、更新的记录数。

但Oracle in Memory不仅对于响应时间快的小事务具有较高程度的优化，同时对于复杂语句的查询其优化程度也很高，可以看到越复杂的查询其优化效果越好，这不仅因为Oracle本身的优化，同时也因为放入内存后读取的便捷。

实验七 添加冗余表

7

通过增加冗余表，可以降低外码和索引的数目，减少计算量，提高数据库查询性能，但同时也会破坏数据库范式，带来数据冗余。

7.1 实验目的

本实验探究添加冗余表相较于普通方法所带来的性能提升率，并分析添加冗余的代价，可能产生的问题。

7.2 实验方案

数据表： order, order_room, room, stay

优化策略：房东经常需要查看自己的订单数据，但是查询房东订单时需要order, order_room, room, stay四张表一起join，开销非常大。我们可以增加一个房东订单数据表，这样在查询房东订单数据时，只需要在本表里查询即可，避免了join的开销。

添加冗余表前查询SQL语句如下：

```
select count(distinct(order_id))
from t_order_room natural join t_order natural join t_room  natural join t_stay
where host_id = 634225;
```

添加冗余表后查询SQL语句如下：

```
select count(distinct(order_id))
from t_host_order
where host_id = 634225;
```

由于TIMESTEN中不支持natural join操作，因此不进行TIMESTEN添加冗余表前的查询演示，实验结果如下：

数据库	优化前	优化后
ORACLE	3. 866s	0. 24s
TIMESTEN		0. 032s
ORACLE IN MEMORY	0. 066s	0. 032s

Tab. 7.1: 增加冗余表实验

7.3 实验设计思考

增加冗余表可以使得部分差分操作速度加快，但是其不符合第三范式，过多添加会使得数据库数据冗余，一致性维护困难，本质上是一种空间换时间的做法，因此，在实际数据库设计中，需要谨慎考虑，适当采取该措施，可以优化查询性能。

7.4 实验设计总结

在本次实验中，添加了一个冗余表，从而减少了一次联表查询操作，查询时间缩短，与预期相符，查询速度直接提升了一个数量级，优化效果显著。

但由于Oracle in Memory的数据本身就在内存中，所以查询所消耗的时间并不是很久，在优化前其查询速度就已经达到0.01s的量级了。但在添加冗余后仍然提高了将近一倍的速度。

进行数据库表设计时，若有些查询需要联合多张表，例如本次查询如果没有冗余表则需要联合四张表，可以考虑建立冗余表加快查询速度。一味的服从范式可能会使数据表看起来更规范，但不一定有利于满足实际的数据库需求，设计最终还是需要服务于功能，但是冗余表的建立还是要谨慎考虑过后再做决定。

实验八 增加冗余操作

8

当进行联表查询时，如果频繁通过外键查询某一字段，且该字段在两表中表示的意义完全相同，此时需要进行冗余设计，将部分字段冗余到关联表中，避免大表之间的关联查询，提高响应速度。

8.1 实验目的

探究冗余操作相较于未使用时所带来的性能变化，以及分析该方法的优劣。

8.2 实验方案

8.2.1 增加冗余列

业务场景：顾客搜索房源时，结果列表展示房东信息（包括用户名），需要以房源表host_id为外键查询host表中的host_username字段。

数据表：stay, host

优化策略：在房源表中加入房东用户名作为冗余列，这样可以避免在查询时对stay和host表进行join操作，从而优化性能。

优化前SQL：

SQL1:

```
select stay_id, t_host.host_username
from t_stay,t_host
where t_stay.host_id=t_host.host_id
and stay_province='上海市'
```

SQL2:

```
select stay_id, t_host.host_username
from t_stay, t_host
where t_stay.host_id=t_host.host_id
and stay_province='上海市'
and stay_capacity<>3
and room_num<>1
```

优化后SQL：

SQL1:

```
select stay_id, host_username
from t_stay
where stay_province='上海市'
```

SQL2:

```
select stay_id, host_username
from t_stay
where stay_province='上海市'
and stay_capacity<>3
and room_num<>1
```

实验结果如下：

查询策略	花费时间
不增加冗余列	7.595s， 3.198s
增加冗余列	6.15s， 3.519s

Tab. 8.1: 增加冗余列实验（Oracle）

实验结果如下：

查询策略	花费时间
不增加冗余列	6.53s， 3.06s
增加冗余列	5.661s， 2.934s

Tab. 8.2: 增加冗余列实验（Oracle in Memory）

业务场景：顾客搜索订单时，结果列表展示订单信息（包括房源名），需要以stay_id为外键查询stay表中的stay_name字段。

数据表：stay, order,order_room

优化策略：在order中加入房源名称作为冗余列，这样可以避免在查询时对order、order_room和stay表进行join操作，从而优化性能。

优化前SQL：

SQL:

```
select distinct stay_name
from t_order_room natural join t_stay
where order_id = '2';
```

优化后SQL：

SQL:

```
select stay_name
from t_order_redundancy
where order_id='2';
```

实验结果如下：

查询策略	花费时间
不增加冗余列	0.262s
增加冗余列	8.441s

Tab. 8.3: 增加冗余列实验（Oracle）

实验结果如下：

查询策略	花费时间
不增加冗余列	0.239s
增加冗余列	0.235s

Tab. 8.3: 增加冗余列实验（Oracle in Memory）

8.3 实验设计思考

适当增加冗余列会使得部分查询操作时速度加快，但其不符合第三范式，过多添加会使得数据库冗余数据增多，反而降低某些查询操作效率。因此，在实际数据库设计中，需要谨慎考虑，适当采取该措施，优化查询性能。

8.4 实验设计总结

在本次实验中，只增加了一列冗余列，从而直接减少了一次联表查询操作，查询时间缩短，与预期相符。但查询速度在数量级上并没有显著提升，这是因为Oracle的优化器实现非常强大，对许多查询都进行了内部优化，在后续更复杂的SQL执行结果中也可以验证该结论。

进行数据库表字段设计时，若有些查询需要联表，但其中一个表对另一个表的字段依赖较少时，如只需要一行，此时可以考虑将该行设置为数据表的冗余字段，可以加快查询速度。一味的服从范式可能会使数据表看起来更规范，但不一定有利于满足实际的数据库需求，设计最终还是需要服务于功能。

但是我们发现实验二本意是减少联表操作达到优化的效果，但是结果却不尽人意。通过对操作与结果的分析，我们发现由于冗余列的字段所占空间太大(stay_name)，导致在每一个块内的记录数大量减少，导致了频繁的I/O操作出现，大大增大了时间开销。而且当我们利用Oracle in Memory加载到内存时，我们发现两者的速度就并无太大的差别了，进一步印证了对于结果的猜想。

通过这个实验，我们发现需要合理的设计表字段的长度，联表操作的消耗与字段过长带来的I/O交换的消耗需要达到一个平衡点才能提高整体的性能。

实验九 冷热对比实验

9

对于房源筛选过程来看，用户青睐于选择价格实惠的房源去入住，在实际情况房源数据总量较多的情况下，可以适当的提前预加载数据，将价格亲民实惠的房源数据提前加载入内存表中，如果提前在内存中命中查询，则可以有效的提高查询效率。

9.1 实验目的

对于不同价格的房源来说，其数据访问频次的差距是很明显的，而对于实际情况下的大数据量内存数据库而言，所有数据都存储在内存中，硬件设备无法承受那么大的数据量。价格贴近实际出行人需求的房源会被经常访问，而价格不贴近实际出行需求的房源访问频率会变得很低，经常访问的热数据因为访问频次需求大，效率要求高，所以可以就近计算和部署，而访问频率低的冷数据对效率要求并不如热数据，所以冷热数据可以进行分开存储，来提高系统的整体性能。

9.2 实验方案

9.2.1 根据价格进行冷热数据区分

数据表：Stay

将价格贴合实际出行需求的房源数据放入内存中，将其余的房源数据放入Oracle中，实现冷热数据在房源价格方面的区分，理论而言会因为数据表大小的减小而对系统的性能有所提升，但其数据分配的比例点需要通过实验进行测试。

9.2.2 对冷热数据比例进行划分

基于给定的查询次数并且对房源价格贴合出行需求的房源可能占全部查询的比例对冷热数据进行比例划分，并且根据不同比例将冷热数据存储在Oracle和Oracle In Memory中，对不同比例数据的查询速度进行测试，寻找最佳的冷热数据比例。

```
create or replace procedure searchForStayInMemory
as
    price NUMBER(10,2);
    s_id NUMBER(10,0);
    s_name VARCHAR2(4000);
    s_province VARCHAR2(1000);
    csr SYS_REFCURSOR;
    sq VARCHAR2(4000) :='select stay_id,stay_name,stay_province from t_stay_c
            where stay_id in (select stay_id from t_room group by stay_id having
min(room_price) < 500)';
    sq2 VARCHAR2(4000) :='select stay_id,s_name,stay_province from t_stay
            where stay_id in (select stay_id from t_room group by stay_id having
min(room_price) < 1000 and min(room_price)>500)';
begin
```

```
    for i in 1 .. 7
        loop
            OPEN csr FOR sq;
              LOOP
                FETCH csr INTO s_id,s_name,s_province;
                EXIT WHEN csr%NOTFOUND;
              END LOOP;
            CLOSE csr;
        end loop;
    for i in 1 .. 3
        loop
            OPEN csr FOR sq2;
              LOOP
            FETCH csr INTO s_id,s_name,s_province;
                EXIT WHEN csr%NOTFOUND;
              END LOOP;
            CLOSE csr;
        end loop;
end;



create or replace procedure searchForStayInMemory
as
    s_id NUMBER(10,0);
    s_name VARCHAR2(4000);
    s_province VARCHAR2(100);
    csr SYS_REFCURSOR;
    sq VARCHAR2(4000) :='select stay_id,stay_name,stay_province  from t_stay
            where stay_id in (select stay_id from t_room group by stay_id having
min(room_price) < 500)';
    sq2 VARCHAR2(4000) :='select stay_id,stay_name,stay_province from t_stay
            where stay_id in (select stay_id from t_room group by stay_id having
min(room_price) < 1000 and min(room_price)>500)';
begin
    for i in 1 .. 7000
        loop
            OPEN csr FOR sq;
              LOOP
            FETCH csr INTO s_id, s_name, s_province;
                EXIT WHEN csr%NOTFOUND;
              END LOOP;
            CLOSE csr;
        end loop;
    for i in 1 .. 3000
        loop
            OPEN csr FOR sq2;
              LOOP
            FETCH csr INTO s_id, s_name, s_province;
                EXIT WHEN csr%NOTFOUND;
              END LOOP;
            CLOSE csr;
        end loop;
end;
```

实验结果：

查询价格小于500占比	总查询时间
50%	4.383s
60%	3.561s
70%	2.938s

Tab. 9.1: 冷热数据查询实验记录

9.3 实验设计思考

本实验的难点在于如何对冷热数据的比例进行划分，当热数据表过大时，热数据表查询的时间会因为表的增大而增加，可能会减缓对于整个房源表的查询速度，而当冷数据表过大时，冷数据表中可能会包含很多本应存放在热数据表中的内容，在查询时可能会多次调用冷数据表，反而降低查询的效率。所以本次实验需要通过多次对比试验对冷热数据的划分进行对比，来确保划分冷热数据块确实能增快数据的查询速度。

9.4 实验设计总结

当房源价格小于500时进行划分时，价格小于等于500的数据表都被加载进内存中，同时，由于房源表为百万级表，所以数据表可以完整加载入内存中，在这种条件下，加载进Oracle In Memory中的数据越多，查询的速度越快。在实际情况下，如果大数据量无法全部加载进内存，可以通过将热门数据加入到内存中，根据数据访问的局部性原理以及对访问数据的频率分析，使得大多数查询命中内存中，减少对硬盘数据的读入与读出的开销，提高系统运行效率。

在查询过程中，Oracle的第一次查询时间会慢于之后的查询时间，而之后的查询时间相近，这是由于第一次查询时可能有很多数据不在内存中，会增加磁盘读写的压力，导致时间变慢。而在TimesTen中不管是第几次查询，其查询时间均相近，这是由于数据全部在内存中，不需要对磁盘进行读入读出，所以查询时间相近。

实验十 事务对插入的影响实验 10

事务可以将多条插入语句并成一次进行commit，可以对插入的速度进行优化，在引入事务后可以对数据库的插入效率进行比较。

10.1 实验目的

对数据库而言，默认每次插入数据都会commit一次，也就是写入一次磁盘，开启事务相当于多次插入后进行一次commit，减少了对磁盘的读写，会加快数据库插入的速度和效率。为此，我们针对数据库的插入效率，对事务对其速率的影响进行实验，来判断怎样才能达到提高系统性能的目的。

10.2 实验方案

我们使用sql中的produre对事务进行使用，sql语句如下。

```
create or replace procedure proc
as
begin
  for i in 1 .. 1000000
    loop
      insert into order values(i,54045, to_date('2007-12-20 18:31:34', 'YYYY-MM-
DD HH24:MI:SS'),1,127.28,0);
    end loop;
  commit;
end;
```

数据表：Order

通过对千万级数据表插入时所进行的时间测算来推定事务对插入时间的影响，将插入数据按照量级分为1w、10w、100w，分别进行使用事务和不使用事务时间的判断。

实验结果：

数据量级	使用事务插入时间	不使用事务插入时间
1w	3 s 762 ms	13 s 378 ms
10w	1 m 21 s 344 ms	2 m 35 s 503 ms
100w	16 m 40 s 629 ms	30 m 57 s 306 ms

Tab. 10.1: 插入数据对比（Oracle）

数据量级	使用事务插入时间	不使用事务插入时间
100w	14 m 3 s 357 ms	26 m 8 s 407 ms

Tab. 10.2: 插入数据对比（Oracle In Memory）

10.3 实验设计思考

一个事务相当于一个原子操作，所以事务的使用有利有弊。当使用事务时，可以降低对磁盘的读写操作，加快速度，但同样如果无法commit时也会导致大部分数据的遗失。

10.4 实验设计总结

从实验结果可以看出，使用事务的插入时间是不使用事务插入时间的百分之五十左右，减少了磁盘的读写会大幅度提高插入的效率，但是使用事务也可能会导致数据的不准确性，比如一个事务读取了另外一个事务未提交的数据等问题，所以事务的使用要在具体情况下具体分析。通过Oracle in Memory一定程度上提高了插入速度，但是从实验结果来看，通过事务插入对实验整体性能提升较大。

本次实验的实验环境是非常简单的环境，所以可以直接使用事务进行优化，没有其他操作者同时操作，在这种情况下使用事务的插入时间远远快于不使用事务的插入时间。

实验十一 数据分区实验

11

数据库中所有数据对象都放在指定的表空间中，当表中数据量不断增大时，查询速度相应会变慢。通过考虑对表进行分区操作，将表中数据在物理上存放到多个表空间中，提高查询指定数据时的性能，一定程度上减少全表扫描开销。

11.1 实验目的

民宿预定平台中存在如对某一时间范围内订单数据查询的功能，即根据下单时间选择订单信息。另一方面，也存在对民宿房源评论信息进行筛选，筛选出有实际评论信息的数据返回展示。可以采用索引对上述需求进行优化，但是另一方面可以采用数据分区的方式减少扫描次数，提高数据库的查询性能。

11.2 实验方案

11.2.1 范围分区：查询时间范围内的订单信息

业务情景：民宿预定平台中部分复杂业务逻辑实现是基于基础功能的优化，例如在生成数据报表时需要根据时间范围统计订单信息，在此处可以采用根据订单表中下单时间列字段的取值范围对订单表进行分区，通过测试相关SQL脚本完成性能实验对比。

查询下单时间在2020-01-01与2021-12-10之间，由标识码为10002的用户所下的订单信息，SQL执行语句如下：

```
select *
from T_ORDER
where customer_id = '10002' and order_time >= to_date('2020-01-01','yyyy-mm-dd')
and order_time <= to_date('2021-12-10','yyyy-mm-dd');
```

由于，我们并不可以在已经创建好的数据表进行数据分区，我们需要额外创建一张分区表进行实验对比，根据实验设计，我们可以依据订单的下单时间范围对订单进行分区，代码如下所示：

```
CREATE TABLE T_ORDER_PARTITION(
    ORDER_ID NUMBER(10,0) PRIMARY KEY,
    CUSTOMER_ID NUMBER(10,0),
    ORDER_TIME DATE,
    CUSTOMER_NUM NUMBER,
    TOTAL_PRICE NUMERIC(10,2),
    ORDER_STATUS NUMBER(2,0)
)PARTITION BY RANGE(ORDER_TIME)
(
```

```
    PARTITION T_ORDER_PART1 VALUES LESS THAN (TO_DATE('01-01-2019','DD-MM-
YYYY')),
    PARTITION T_ORDER_PART2 VALUES LESS THAN (TO_DATE('01-01-2021','DD-MM-
YYYY')) ,
    PARTITION T_ORDER_PART3 VALUES LESS THAN (TO_DATE('01-01-2023','DD-MM-
YYYY')),
    PARTITION T_ORDER_PART4 VALUES LESS THAN (MAXVALUE)
);
```

根据上述分区后的订单表，由于每一个分区的数据量小于总计的订单表，因此条件查询在分区内执行速度会有提升。

但是由于TIMESTEN中并不支持分区操作，因此不进行TIMESTEN的数据分区查询演示，实验结果如下：

数据库	不使用数据分区	使用数据分区
ORACLE	2.762s	1.413s
TIMESTEN	0.642s	
ORACLE IN MEMORY	0.038S	0.035S

Tab. 11.1: 范围分区实验结果表格

11.2.2 列表分区：目的性对评论数据进行筛选查询

业务情景：民宿预定平台在实际生活中会遇到评价中仅有评价得分但没有评价内容的情况，在对房源进行展示时需要筛选出有实际内涵的评价信息，如果对全表进行扫描会比较产生比较大的开销，因此通过对列表分区的变种，根据评论内容的有无进行区分，将评论信息分区为两部分表，分别为有实际评论内容的表与无实际评论内容的表，进行性能实验对比测试。

通过查询房源标识码为1000的房源评价进行对分区实验的测试，其中SQL执行语句如下所示：

```
select customer_id,comment_content,comment_grade,comment_time
from t_comment,t_order,t_order_room
where t_comment.order_id = t_order.order_id and t_order.order_id =
t_order_room.order_id and stay_id = '1000' and comment_content is not null;
```

通过对上述过程的理解，与范围分区不同的是，可以通过对评论内容是否是空值进行分区，这样做的目的在于尽可能的少浏览过多的无用信息，设计分区如下：

```
CREATE TABLE T_COMMENT_PARTITION(
    COMMENT_ID NUMBER PRIMARY KEY,
    COMMENT_CONTENT VARCHAR2(100),
    COMMENT_GRADE NUMBER,
    COMMENT_TIME DATE,
    ORDER_ID NUMBER
)PARTITION BY LIST(COMMENT_CONTENT)
(
```

```
    PARTITION T_COMMENT_PART1 VALUES (NULL),
    PARTITION T_COMMENT_PART2 VALUES (DEFAULT)
);
```

同样，TIMESTEN中无法实现数据分区不进行相关的结果查询，其他情况实验结果如下：

数据库	不使用数据分区	使用数据分区
ORACLE	4.972s	3.553s
TIMESTEN	3.82s	
ORACLE IN MEMORY	0.083s	0.081s

Tab. 11.2: 列表分区实验结果表格

11.3 实验设计思考

数据分区实验会对查找性能上有所优化，但无论选择范围分区优化或者列表分区优化，都是选择具有代表性的特殊的表字段进行筛选。相对于索引而言，没有额外的增删改开销，但是相对来说可能灵活性较差。实验可能遇到的困难在于分区带来的收益是否较大以及如何确定分区的数目以及分区适应的场景。

11.4 实验小结

数据分区对大数据量的表单进行规则的分块处理，使得可以达到一个根据目的定点查询的效果。结合实验过程来看，数据量越大，分区后查询的提升效果也是越明显的，通过分区可以减少不必要的查询开销，达到提升查询速度的效果。在将数据加入内存后，我们发现查询效果提升了很多，相对于分区带来的效果而言也有显著的性能优化。根据官方文档提示，数据分区的表最好大小超过2G，对于本实验来说，数据量大小远没有达到该标准，数据分区实验的效果并没有利用到最大值。另一方面，数据分区理想的应用场景是在查询落在指定分区的情况下，如果是复杂的跨多分区的查询，一定程度上会导致效果的退化。因此选择是否使用数据分区应当结合查询场景以及表的数据量大小与可扩充性，避免出现因为分区后的查询跨区开销反而降低查询效果。