SQL速成

DBMS：MySQL

- Retrieving Data
- Inserting Data
- Updating Data
- Deleting Data
- Summarizing Data
- Writing Complex Queries
- Built-in Functions
- Views
- Stored Procedures

查询一个数据库：USE语句或者直接双击

> USE sql_store （SQL不区分大小写，但关键词最好用大写）

用；来结束一个操作命令

```
USE sql_store;
SELECT * (*代表所有列)
FROM customers
-- WHERE customer_id = 1 '--'表示注释
ORDER BY first_name
```

```
SELECT
      first_name,
      last_name,
      points + 10 AS discount_factor/'discount factor'
FROM customers
-- 在customers的数据库中提取列名为first_name和last_name的列，其书写顺序即为提取顺序

SELECT DISTINCT state (DISTINCT：删除重复项)
FROM customers
```

```
WHERE (用于筛选数据，类似条件句)
SELECT *
FROM customers
WHERE points > 3000 (!=/<>都表示不等于)
WHERE state = 'VA' ("VA"是字符串故要加引号)
```

```
WHERE birth_date > '1990-01-01' (晚于1990年生的人，XXXX-XX-XX是mysql日期的默认格式) AND points > 1000 (用AND表示交，OR表示并，NOT表示否定，AND有最优先级)

WHERE birth_date > '1990-01-01' OR points > 1000 AND state = 'VA' (筛选条件是同时满足points>1000且在VA的人，然后或者是生于1990年之后的人)
-- 括号可以改变逻辑顺序
WHERE NOT (birth_date > '1990-01-01' OR points > 1000) (表示筛选出来的人既早于1990年出生又积分小于等于1000)
WHERE state IN ('VA', 'FL', 'GA') =
WHERE state = 'VA' OR state = 'FL' OR state = 'GA'
WHERE points BETWEEN 1000 AND 3000
WHERE birth_date BETWEEN '1990-01-01' AND '2000-01-01'

LIKE (用于检索遵循特定字符串模式的行)
WHERE last_name LIKE 'b%' (检索所有姓以b开头的顾客)
WHERE last_name LIKE '_____y' (%用于表示任何长度的字符串，_用于表示一个字符串)

REGEXP (which is short for regular expression)
WHERE last_name REGEXP 'field' =
WHERE last_name LIKE '%field%'
WHERE last_name REGEXP '^field' (^表示字符串开头)
'field$' ($表示字符串结尾)
WHERE last_name REGEXP 'field|mac|rose' (|表示或者)
WHERE last_name REGEXP '[gim]e' (表示姓里有ge/ie/me的顾客)
WHERE last_name REGEXP '[a-h]e' ([]匹配任意在括号里列举的单字符)

IS NULL (用于查找缺失值)
WHERE phone IS NULL
WHERE phone IS NOT NULL
```

```
ORDER BY (用于改变数据排列顺序)
SELECT *
FROM customers
ORDER BY first_name DESC (short for descending降序排列)
ORDER BY state, first_name (依照参考顺序排序：先按洲际排序，相同洲际按名字排序)
ORDER BY state DESC, first_name DESC
```

```
LIMIT (用于限制搜索，LIMIT语句永远放在最后)
SELECT *
FROM customers
LIMIT 3 (只取前三行)
LIMIT 6, 3 (跳过前6项，然后取3项)
```

```
综上语句的排列顺序如下：
SELECT
FROM
WHERE
ORDER BY
LIMIT
```

内连接：通过唯一指代名找到不同表的内在联系(类比：vlookup)

```
SELECT order_id, order.customer_id, first_name, last_name
-- order.customer_id有前缀order.是因为orders表和customers表中都有customer_id这一列，需要明确到底提取哪一列
FROM orders
JOIN customers
    ON order.customer_id = customers.customer_id （INNER JOIN：INNER可省）
简化版：可定义表的缩写，以减少重复引用的工作量
SELECT order_id, o.customer_id, first_name, last_name
FROM orders o
JOIN customers c
    ON o.customer_id = c.customer_id

USE sql_store;
SELECT *
FROM order_items oi
JOIN sql_inventory.products p （连接不同数据库的表要加数据库前缀）
    ON oi.product_id = p.product_id
=   USING(product_id) （用于两表列名相同时）
```

自连接：可用于寻找组织架构

```
USE sql_hr;

SELECT *
FROM employees e
JOIN employees m
  ON e.reports_to = m.employee_id
```

连接多个表

```sql
SELECT order_id, order_date, first_name, last_name, name AS status
FROM orders o
JOIN order_statuses
  ON status = order_status_id
JOIN customers c
  ON o.customer_id = c.customer_id
```

复合连接条件：有时会出现复合主键，需要两列甚至多列才能唯一确定一行数据

```sql
-- 在order_items表中，order_id有重复是因为同一个顾客订购了不同的产品，product_id有重复是因为同一种产品有不同顾客订购，该表的逻辑在于：一个订单对应一个或多个产品，一对多时订单需要分列明细科目。故要两个id才能确定唯一一条订单
SELECT *
FROM order_items oi
JOIN order_item_notes oin
  ON oi.order_id = oin.order_d
  AND oi.product_id = oi.product_id
```

隐式连接语法(但尽量用显示连接语法)

```sql
SELECT *
FROM orders o, customers c
WHERE o.customer_id = c.customer_id
=
SELECT *
FROM orders o
JOIN customers c
  ON o.customer_id = c.customer_id
```

外连接

```sql
LEFT JOIN 左边的表的记录都会被返回(尽量用左连接)
RIGHT JOIN 右边同理(左：FROM的表；右：JOIN的表)

SELECT
  c.customer_id,
    c.first_name,
    o.order_id
FROM customers c
LEFT JOIN orders o
  ON c.customer_id = o.customer_id
-- 上述语句产生的结果就是：显示所有顾客就算其现在没有订购东西
```

自然连接：数据库引擎会自己看着办，把相同列名连接到一起（不推荐使用）

```sql
SELECT *
FROM orders o
NATURAL JOIN customers c
```

交叉连接

```sql
SELECT
  c.first_name,
  p.name
FROM customers c
CROSS JOIN products p
ORDER BY 1（指的是第一列）
-- 上述语句将每个顾客的名字（10个）和每个产品（10个）的名字都连接了一遍，生成了10*10的数据。常用于得到所有可能的组合。

-- 隐式表示法
SELECT
  c.first_name,
  p.name
FROM customers c, products p
ORDER BY 1
```

联合：合并多个查询结果（列数要相同!!!）

```sql
SELECT
  order_id,
  order_date,
  'Active' AS status
FROM orders
WHERE order_date >= '2019-01-01'
UNION
SELECT
  order_id,
  order_date,
  'Archived' AS status
FROM orders
WHERE order_date < '2019-01-01'
```

列属性

- int 整数
- varchar for variable character, (50) is the limit of the character
- PK for primary key; NN for not null;

插入单行(结合每张表的列属性查看)

```sql
INSERT INTO customers ()
VALUES (
  DEFAULT,
  'John',
  'Smith',
  '1990-01-01',
  NULL,
  'address',
  'city',
  'CA',
  DEFAULT)
-- values子句用于新增数据，切记数据要与列顺序一致，数量也要相同，default在这里是因为customer_id是主键，系统会自动生成一个唯一值（AI for auto_increment），故直接填默认最好（也可以自己赋值，但不推荐）
OR
INSERT INTO customers (
  first_name,
  last_name,
  birth_date,
  address,
  city,
  state)
-- 想要赋值的列(顺序可以改变，只要和下面的值一一对应即可)
VALUES (
  'John',
  'Smith',
  '1990-01-01',
  'address',
  'city',
  'CA')
```

插入多行

```sql
INSERT INTO shippers (name)
VALUES ('shipper1'),
       ('shipper2'),
       ('shipper3')
```

往多张表中插入数据

```sql
INSERT INTO orders (customer_id, order_date, status)
VALUES (1, '2019-02-03', 1);

INSERT INTO order_items ()
VALUES (LAST_INSERT_ID(), 1, 1, 2.97),
       (LAST_INSERT_ID(), 2, 1, 5.97)

SELECT LAST_INSERT_ID()
-- 得到新加数据的id
```

复制表内容

```sql
-- 创建新表
CREATE TABLE orders_archived AS
SELECT * FROM orders （属于子查询）
-- 上述语句将orders表中的所有数据都快速复制到新表中，但mysql会忽略原表中每列数据的属性，故新表中没有主键，也不会自动增列，所以在新表中增加数据时，原主键列需要手动赋值

-- 复制原表中一部分数据，可以在INSERT下加入子查询
INSERT INTO orders_archived
SELECT *
FROM orders
WHERE order_date < '2019-01-01'
```

更新表中数据

```sql
UPDATE invoices （定位哪张表）
SET payment_total = 10, payment_date = '2019-03-01'
-- SET用于指定一列或多列的新值（表里的哪些数据）
WHERE invoice_id = 1
-- 用于定位需要更新哪一条或多条数据

-- 在UPDATE中运用子查询
UPDATE invoices
SET
  payment_total = invoice_total * 0.5
  payment_date = due_date
WHERE client_id =/IN    （如果子查询返回的数据大于1，则用IN代替=）
  (SELECT client_id
   FROM clients
   WHERE name = 'Myworks')
-- 上述语句的逻辑是，先找出用户名为“Myworks”的cliemt_id，然后将这位顾客的所有订单都更新数据
```

删除数据

```
DELETE FROM invoices
WHERE invoice_id = 1
-- 不加限制条件会删除整张表(同样可以使用子查询)
```

恢复数据库(回到初始状态)

> File----Open SQL Script----create-datebases.sql----run it----done

聚合函数

```
SELECT
  MAX(invoice_total) AS highest,
  MIN(invoice_total) AS lowest,
  AVG(invoice_total) AS average,
  SUM(invoice_total) AS total,
  COUNT(invoice_total) AS number_of_invoices
FROM invoices
-- 这些函数也可以应用于日期和字符串上，且它们只运行非空值，会取重复值；可在括号内运用公式
WHERE invoice_date > '2019-07-01'
-- 加入限制后只会计算限制后的数据（相当于限制程序优先被执行）
COUNT(*) AS total_records
-- 计算包括非空的所有记录
COUNT(DISTINCT client_id)
-- 去掉重复值要使用DISTINCT
```

数据分组

```
SELECT
  client_id,
  SUM(invoice_total) AS total_sales
FROM invoices
WHERE invoice_date >= '2019-07-01'
GROUP BY client_id (默认排序是按照分组的列进行的)
ORDER BY total_sales DESC
-- 注意语句逻辑顺序

-- 多列分组
SELECT
  state,
  city,
  SUM(invoice_total) AS total_sales
FROM invoices i
JOIN clients USING (client_id)
```

```
GROUP BY state, city
```

分组之后筛选数据

```
HAVING语句用于分组之后（引用的列必须是SELECT中的!!!），同样可用逻辑语句设置条件限制
WHERE语句用于分组之前

SELECT
  client_id,
  SUM(invoice_total) AS total_sales
FROM invoices
GROUP BY client_id
HAVING total_sales > 500
```

汇总运用聚合函数的列的值

```
SELECT
  client_id,
  SUM(invoice_total) AS total_sales
FROM invoices
GROUP BY client_id WITH ROLLUP  （WITH ROLLUP相当于汇总求和）
```

⚠️ 在使用WITH ROLLUP 时，GROUP BY 的列不能使用列别名，而要使用其实际名称

更复杂的查询

```
-- 在WHERE下添加子查询
SELECT *
FROM employees
WHERE salary > (
      SELECT AVG(salary)
      FROM employees
)

-- Find clients without invoices
SELECT *
FROM clients
WHERE client_id NOT IN (
    SELECT DISTINCT client_id
    FROM invoices
)
```

子查询和连接的比较：在不同情况下，两种算法执行效率有差异，且要比较可读性

```

```

```sql
-- Find customers who have ordered lettuce(product_id = 3), and
select customer_id, first_name, last_name

-- TO use Joins
SELECT
  DISTINCT customer_id,
    first_name,
    last_name
FROM customers
JOIN orders USING (customer_id)
JOIN order_items USING (order_id)
WHERE product_id = 3

-- TO use subqueries
SELECT
  DISTINCT customer_id,
    first_name,
    last_name
FROM customers
JOIN orders USING (customer_id)
WHERE order_id IN (
    SELECT DISTINCT order_id
    FROM order_items
    WHERE product_id = 3
)
```

- ALL

```sql
SELECT *
FROM invoices
WHERE invoice_total > ALL (
    SELECT invoice_total
    FROM invoices
    WHERE client_id = 3
        )
=
SELECT *
FROM invoices
WHERE invoice_total > (
    SELECT MAX(invoice_total)
    FROM invoices
    WHERE client_id = 3
        )
```

- ANY

```sql
-- Select clients with at least two invoices
SELECT *
FROM clients
WHERE client_id = ANY (
  SELECT client_id
  FROM invoices
  GROUP BY client_id
  HAVING COUNT(*) >= 2
)
=
SELECT *
FROM clients
WHERE client_id IN (
  SELECT client_id
  FROM invoices
  GROUP BY client_id
  HAVING COUNT(*) >= 2
)
```

相关子查询

```sql
-- Select the employees whose salary are larger than the average salary of their offices
SELECT *
FROM employees e
WHERE salary > (
  SELECT AVG(salary)
  FROM employees
  WHERE office_id = e.office_id （这里体现相关性）
)
-- 上述语句的逻辑：对employees的每一条数据的salary比较和它拥有相同office_id的部分的平均薪资，若比部门平均水平高则提取相关人员出来
-- 对每一条数据程序都会对子查询做一次处理，所以相关子查询随着数据量的增大会耗时更久
```

- EXISTS

```sql
Select clients that have an invoice
SELECT *
FROM clients c
WHERE EXISTS(
  SELECT client_id
  FROM invoices
  WHERE client_id = c.client_id
)
```

```sql
-- 这个子查询会返回True/False，而不是一个结果（与IN的区别，如果数据量很大的话，IN会占用很多空间）
=
SELECT *
FROM clients
WHERE client_id IN (
  SELECT DISTINCT client_id
  FROM invoices
)
```

SELECT下的子查询

```sql
SELECT
  invoice_id,
    invoice_total,
    (SELECT AVG(invoice_total)
    FROM invoices) AS invoice_average
    invoice_total - (SELECT invoice_average) AS difference
FROM invoices
-- 直接AVG()只会返回一个数值，故要先用一遍SELECT查询
-- (SELECT invoice_average)也可以写成(SELECT AVG(invoice_total) FROM invoices)，但不能直接引用invoice_average这个别名
```

第六章第九节练习题 （bilibili：BV1UE41147KC）

```sql
SELECT
  client_id,
  name,
  (SELECT SUM(invoice_total)
      FROM invoices
      WHERE client_id = c.client_id) AS total_sales,
  (SELECT AVG(invoice_total) FROM invoices) AS average,
  (SELECT total_sales - average) AS difference
FROM clients c
-- 两个结果相同
SELECT
  client_id,
    name,
  SUM(invoice_total) AS total_sales,
    (SELECT AVG(invoice_total) FROM invoices) AS average,
     SUM(invoice_total) - (SELECT average) AS difference
FROM clients
LEFT JOIN invoices USING (client_id)
GROUP BY client_id
```

FROM下的子查询(仅限于简单的查询，复杂的要借助视图)

```sql
SELECT *
FROM (
  SELECT
    client_id,
    name,
    (SELECT SUM(invoice_total)
        FROM invoices
        WHERE client_id = c.client_id) AS total_sales,
    (SELECT AVG(invoice_total) FROM invoices) AS average,
    (SELECT total_sales - average) AS difference
  FROM clients c
) AS sales_summary
-- 引用新表必须赋予名字
WHERE total_sales IS NOT NULL
```

内置函数

- 数值函数：Numeric Functions

```sql
SELECT ROUND(5.73, 1)
-- 四舍五入原理；第二位参数设置小数点后几位的位数
SELECT TRUNCATE(5.7345, 2)
-- 只是单纯截断
SELECT CEILING(5.2)
-- 返回大于或等于该值的最小整数
SELECT FLOOR(5.2)
-- 取整函数
SELECT ABS(-5.3)
-- 绝对值函数
SELECT RAND()
-- 随机生成0-1的数
```

完整的数值函数见这里

- 字符串函数：String Functions----用于处理字符串值的函数

```sql
SELECT LENGTH('sky')
-- 字符长度
SELECT UPPER('sky')
-- 大写字母
SELECT LOWER('Sky')
-- 小写字母
-- 以下为删除字符串中不必要的空格的函数
SELECT LTRIM('    sky')
-- which is short for left trim 左修整
SELECT RTRIM('sky   ')
```

```
-- 右修整
SELECT TRIM('  sky  ')
-- 前后都修整
SELECT LEFT('Kindergarten', 4)
-- 取左边四位
SELECT RIGHT('Kindergarten', 6)
-- 取右边六位
SELECT SUBSTRING('Kindergarten', 3, 5)
-- 字符截取函数，起始点（包括这个点），截取长度（可选参数）
SELECT LOCATE('n', 'Kindergarten')
-- 定位n在Kindergarten中是第几位（无关大小写），如果没有会返回0值，若是单词如garten在其中的位置，会取第一个字母开始的值
SELECT REPLACE('Kindergarten', 't', 'd')
SELECT REPLACE('Kindergarten', 'garten', 'garden')
-- 替换的字符，字符中要替换的字符，想要替换成的字符
SELECT CONCAT('hello', 'world', '!')
-- 串联多个字符
```

完整的字符串函数见这里

- 日期函数：Date Functions

```
SELECT NOW()
-- 调用当前时间
SELECT CURDATE()
-- which is short for current date 仅返回当前日期
SELECT CURTIME()
SELECT YEAR(NOW())
SELECT MONTH(NOW())
SELECT DAY(NOW())
SELECT HOUR(NOW())
SELECT MINUTE(NOW())
SELECT SECOND(NOW())
-- 上述函数均返回整数
-- 以下两个返回字符串
SELECT DAYNAME(NOW())
SELECT MONTHNAME(NOW())
-- EXTRACT是SQL的标准函数，适用于其他DBMS
SELECT EXTRACT(DAY/YEAR/MONTH... FROM NOW())
```

格式化日期和时间

```
SELECT DATE_FORMAT(NOW(), '%M %Y')
-- 第二位参数是需要转换成的格式，字母大小写有意义
SELECT TIME_FORMAT(NOW(), '%H:%i %p')
```

完成函数在这里

计算日期和时间

```
SELECT DATE_ADD(NOW(), INTERVAL 1/-1 DAY/YEAR)
-- 给日期时间添加日期成分，第一个参数为添加对象，第二个参数为增量大小
SELECT DATE_SUB(NOW(), INTERVAL -1/1 DAY/TEAR)
-- 与上面的语句一一对应
SELECT DATEDIFF('2019-01-05', '2019-01-01')
-- 计算两个日期的间隔时间（单位只能为天），前-后（非绝对值）
SELECT TIME_TO_SEC('09:00')
SELECT TIME_TO_SEC('09:00') - TIME_TO_SEC('09:02')
-- 计算从零时开始到输入值的秒数，以此可计算两个时间相隔的秒数
```

- IFNULL and COALESC Functions

```
SELECT
  order_id,
  IFNULL(shipper_id, 'Not assigned') AS shipper
-- shipper_id有些是空值，将空值部分换成"Not assigned"
FROM orders

SELECT
  order_id,
  COALESCE(shipper_id, comments 'Not assigned') AS shipper
-- shipper_id若是空值，则提取comments列中的内容，若还是空值则填写"Not assigned"
-- COALESCE可以输入一系列值，函数会返回第一个非空值
FROM orders
```

- IF Functions

```
IF(expression, first_value, secon_value)
-- 表达式为真，返回第一个值，否则返回第二个值（值可以是空值/数字/日期/字符串等）

SELECT
  order_id,
  order_date,
  IF(YEAR(order_date) = YEAR(NOW()),
    'Active',
     'Archived'
    ) AS category
FROM orders
=
SELECT
  order_id,
```

```
  order_date,
  'Active' AS category
FROM orders
WHERE YEAR(order_date) = YEAR(NOW())
UNION
SELECT
  order_id,
  order_date,
  'Archived' AS category
FROM orders
WHERE YEAR(order_date) != YEAR(NOW())
```

第七章第七节练习题（bilibili：BV1UE41147KC）

```
SELECT
  product_id,
    name,
    (SELECT
        COUNT(*)
     FROM order_items
     WHERE product_id = p.product_id) AS orders,
    IF((SELECT orders) > 1, 'Many times', 'Once') AS frequency
FROM products p
-- 注意两点
-- 在SELECT之外可以用COUNT+GROUP BY的语句，可以在SELECT语句下用相关子连接+WHERE的格式，谨记!!!
-- 使用别名会报错，用SELECT+别名
=
SELECT
  product_id,
  name,
  COUNT(*) AS orders,
  IF(COUNT(*) > 1, 'Many times', 'Once') AS frequency
FROM products
JOIN order_items USING (product_id)
GROUP BY product_id
```

CASE(弥补IF函数只允许单一测试表达式的缺陷，允许多个表达式)

```sql
SELECT
  order_id,
  CASE
    WHEN YEAR(order_date) = YEAR(NOW()) THEN 'Active'
    WHEN YEAR(order_date) = YEAR(NOW()) - 1 THEN 'Last Year'
    WHEN YEAR(order_date) < YEAR(NOW()) - 1 THEN 'Archived'
    ELSE 'Future'
  END AS category
FROM
-- 一个WHEN表示一个表达式
```

第七章第八节练习题（bilibili：BV1UE41147KC）

```sql
SELECT
  CONCAT(first_name, ' ', last_name) AS customer,
  points,
  CASE
    WHEN points > 3000 THEN 'Gold'
    WHEN points >= 2000 THEN 'Silver'
    ELSE 'Bronze'
  END AS category
FROM customers
=
-- 之前用UNION的语句
```

创建视图：简化复杂查询语句（本质是创建一个虚拟表，以便后续重复使用）

其他优点（第八章第五节没太懂）：

1. 简化变动

> 若原表中的列A改名为B，则之前与列A相关的已存在的查询就都需要改为B，但可在列A变为B后新建一个VIEW，将VIEW中的列B改名为A，再将与列A相关的原查询语句运用在新建的视图中即可

2. 没听懂 再说吧

```sql
CREATE VIEW sales_by_client AS
SELECT
  c.client_id,
  c.name,
  SUM(invoice_total) AS total_sales
FROM clients c
JOIN invoices i USING (client_id)
GROUP BY client_id, name
-- 可以将其当作Table来使用，和CREATE TABLE的区别在于表储存数据，视图不储存数据，看作虚拟表
```

更改或删除视图

```sql
-- 删除视图（想要修改必须先删除或建立一个名字不同的视图）/再重建视图（运行上述语句即可）
DROP VIEW sales_by_client
-- 另法：可以不用先删除再修改重建，执行多少次都可
CREATE OR REPLACE VIEW sales_by_client AS
SELECT
  c.client_id,
  c.name,
  SUM(invoice_total) AS total_sales
FROM clients c
JOIN invoices i USING (client_id)
GROUP BY client_id, name
-- 另另法：保存源文件在sql里，以便他人修改
```

可更新视图

定义：没有以下语句的视图

> - DISTINCT
> - Aggregate Functions (MIN/MAX/SUM...)
> - GROUP BY / HAVING
> - UNION

```sql
CREATE OR REPLACE VIEW invoices_wiith_balance AS
SELECT
  invoice_id,
  number,
  client_id,
  invoice_total,
  payment_total,
  invoice_total - payment_total AS balance,
  invoice_date,
  due_date,
```

```
  payment_date
FROM invoices
WHERE (invoice_total - payment_total) > 0
-- 可像普通表一样进行删除或增加等后续操作
DELETE FROM invoices_with_balance
WHERE invoice_id = 1

UPDATE invoices_with_balance
SET due_date = DATE_ADD(due_date, INTERVAL 2 DAY)
WHERE invoice_id = 2
```

WITH OPTION CHECK

> 有时候更新或删除可更新视图时，一些行会从视图中消失，为防止这一现象，可在创造可更新视图的最后加上WITH OPTION CHECK，则当修改一行会使其消失时，系统会报错以提醒你

```
CREATE OR REPLACE VIEW invoices_wiith_balance AS
SELECT
  invoice_id,
  number,
  client_id,
  invoice_total,
  payment_total,
  invoice_total - payment_total AS balance,
  invoice_date,
  due_date,
  payment_date
FROM invoices
WHERE (invoice_total - payment_total) > 0
WITH CHECK OPTION
```

存储过程(Stored Procedure)：能够返回多行多列的结果集

> 一个包含一堆SQL代码的数据库对象，用于存储和管理SQL代码
>
> 存储过程里的SQL代码有时执行效率更高
>
> 可加强数据安全性

```
-- 创建存储过程
DELIMITER $$
CREATE PROCEDURE get_clients()
BEGIN
  SELECT * FROM clients;
END$$

```

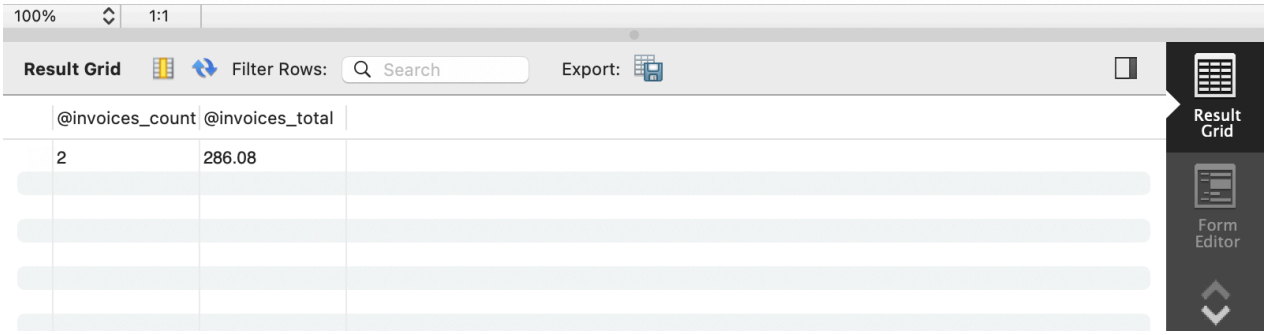
```sql
DELIMITER ; (将分隔符改为默认的';')
-- BEGIN和END之间的内容为存储过程的主体，且每条语句需要用';'隔开
-- DELIMITER means 分隔符，告诉系统设置了一个新的分隔符，$$之间的语句作为一个整体处理

-- 简化版：右键Stored Procedures----Create XXX----该页面下不用修改分隔符----Apply

-- 调用存储过程
CALL get_clients()

-- 删除存储过程
DROP PROCEDURE get_clients
DROP PROCEDURE IF EXISTS get_clients (如果已经删除，再执行该语句系统也不会报错)

-- 与视图相同，可储存在sql里，以便他人recreate

-- 添加参数：为存储过程传递值
DELIMITER $$
CREATE PROCEDURE get_clients_by_state(state CHAR(2))
-- 参数为state，并将其类型设置为CHAR(2)
BEGIN
  SELECT * FROM clients c
  WHERE c.state = state;
END $$

DELIMITER ;
-- 调用赋参存储过程
CALL get_clients_by_state('VA') (如设置了参数，那么参数为必填项)

-- 设置带默认值的参数
DELIMITER $$
CREATE PROCEDURE get_clients_by_state(state CHAR(2))
BEGIN
  IF state IS NULL THEN
    SET state = 'CA'; (如果输入值为空，则返回默认值"CA")
  END IF;

  SELECT * FROM clients c
  WHERE c.state = state;
END $$

DELIMITER ;

-- 参数验证：以确保存储过程不会王数据库存储错误数据
-- 下面是一个更新发票的程序
```

```sql
DELIMITER $$
CREATE PROCEDURE make_payment
(
  invoice_id INT,
  payment_amount DECIMAL(9, 2),
  payment_date DATE
)
BEGIN
  IF payment_amount <= 0 THEN
    SIGNAL SQLSTATE '22003' SET MESSAGE_TEXT = 'Invalid payment amount'; （这里就是参数验证的过程，确保输入金额为正）
  END IF;
  UPDATE invoices i
  SET
    i.payment_total = payment_amount,
    i.payment_date = payment_date
  WHERE i.invoice_id = invoice_id;
END $$

DELIMITER ;
-- 22003 stands for "A numeric value is out of range"
```

完整的SQL错误代码在这里

```sql
-- 输出参数
-- 下面是一个计算未支付发票数量和总金额的存储过程
DELIMITER $$
CREATE PROCEDURE get_unpaid_invoices_for_client
(
  client_id INT,
  OUT invoices_count INT,
  OUT invoices_total DECIMAL(9, 2)
)
BEGIN
  SELECT COUNT(*), SUM(invoice_total)
  INTO invoices_count, invoices_total
  FROM invoices i
  WHERE i.client_id = client_id AND i.payment_total = 0;
END $$

DELIMITER ;
-- OUT前缀代表输出变量，从调用带输出参数的过程在读取数据上更繁琐（如下图），应尽量避免使用
-- 下图中，@前缀表示这是用户定义变量，会先设定初始值，调动后再select（这就是繁琐的原因）
```

```sql
set @invoices_count = 0;
set @invoices_total = 0;
call sql_invoicing.get_unpaid_invoices_for_client(3, @invoices_count, @invoices_total);
select @invoices_count, @invoices_total;
```




```sql
-- 变量种类
-- User or session variables：会随着下线而被清空
SET @invoices_count = 0

-- Local variables：随着程序执行完毕就被清空，常用于执行计算任务
DELIMITER $$
CREATE PROCEDURE get_risk_factor()
-- 定义这样一个公式：risk_factor = invoices_total / invoices_count * 5
BEGIN
  DECLARE risk_factor DECIMAL(9, 2) DEFAULT 0;
  DECLARE invoices_total DECIMAL(9, 2);
  DECLARE invoices_count INT;
-- 首先声明一下定义风险因子需要用到变量
  SELECT COUNT(*), SUM(invoice_total)
  INTO invoices_count, invoices_total
  FROM invoices i;
-- 设置这两个变量
  SET risk_factor = invoices_total / invoices_count * 5;
-- 计算风险因子
  SELECT risk_factor
END $$

DELIMITER ;
-- 上述语句中，invoices_total和invoices_count是Local variables，这些变量只有在存储过程中才有意义，一旦结束程序即会被清空
```

函数：只能返回单一值（打开方式：在工具栏找到Functions右键create即可）

```sql
-- 计算每个顾客风险因子
CREATE FUNCTION get_risk_factor_for_client
(
  client_id INT
)
RETURNS INTEGER(与存储过程最大的区别，明确了返回值的类型，可以是其他数据类型)
READS SQL DATA
-- 设置函数属性
BEGIN
  DECLARE risk_factor DECIMAL(9, 2) DEFAULT 0;
  DECLARE invoices_total DECIMAL(9, 2);
  DECLARE invoices_count INT;

  SELECT COUNT(*), SUM(invoice_total)
  INTO invoices_count, invoices_total
  FROM invoices i
  WHERE i.client_id = client_id;

  SET risk_factor = invoices_total / invoices_count * 5;

  RETURN IFNULL(risk_factor, 0); (要考虑invoice_total和invoice_count为0导致返回值为NULL的情况)
END

-- 应用
SELECT
  client_id,
  name,
  get_risk_factor_for_client(client_id) AS risk_factor
FROM clients

-- 删除函数
DROP FUNCTION IF EXISTS get_risk_factor_for_client

-- 保存函数的方法同存储过程
```

函数属性

> - DETERMINISTIC：保证同一组数据返回同一值
> - READS SQL DATA：函数中会配置选择语句，用以读取一些数据
> - MODIFIES SQL DATA：函数中有插入、更新或删除函数

第九章第六节练习题（bilibili：BV1UE41147KC）

> 要求：写一个get_payments的存储过程，设置两个非必填参数，若两个参数均为空值则返回全部发票，若client_id不是空值，则返回该顾客的发票，若client_id和payment_method_id都不是空值，就返回相应发票

```sql
DELIMITER $$
CREATE PROCEDURE get_payments(
client_id INT,
payment_method_id TINYINT
)
BEGIN
  SELECT *
    FROM payments p
    WHERE
      p.client_id = IFNULL(client_id, p.client_id) AND
      p.payment_method = IFNULL(payment_method_id,
p.payment_method);
END $$

DELIMITER ;

-- p.client_id = IFNULL(client_id, p.client_id)的逻辑在于：如果client_id是空值，则返回p.client_id，有p.client_id = p.client_id，始终成立，相当于无限制条件，即返回所有数据；如果client_id不是空值，则有p.client_id = client_id，返回使限制条件成立的数据
```

arguments 和 parameters 的区别（以下为CHATGPT的回答）

> In coding, "arguments" and "parameters" are related concepts but they serve different roles.
>
> 1. **Parameters**:
>   - Parameters are the placeholders or variables defined in the function definition.
>   - They act as local variables within the function, representing the data that the function expects to receive when it's called.
>   - Parameters define the signature of the function and specify what kind of data the function needs to work with.
>   - Parameters are part of the function declaration or definition.
> 2. **Arguments**:
>   - Arguments are the actual values that are passed to a function when it is called.
>   - They provide the data that the function will operate on.
>   - When you call a function, you supply arguments for each parameter the function expects, matching the order and data types specified by the parameters.
>   - Arguments are part of the function call.

触发器：TRIGGER

> Definition: A block of SQL code that automatically gets executed before or after an insert, update or delete statement.
>
> 通常使用触发器增强数据一致性

```sql
-- 创建触发器
-- 在payments表中，几个payments可能来自同一个invoice，故当新增一个payment的时候，invoices表中的payment_total就需要相应发生变化，故可以建立一个触发器来实现这一目标
DELIMITER $$
CREATE TRIGGER payments_after_insert
  AFTER INSERT ON payments
-- AFTER/BEFORE 触发事件可选
-- INSERT/UPDATE/DELETE statement可选
  FOR EACH ROW (表示触发器会作用到每一个受影响的行)
BEGIN
  UPDATE invoices i
  SET payment_total = payment_total + NEW.amount (NEW用于指代新输入的行，amount定位具体数据；OLD会返回更新或删除前的相应数据，具体问题具体分析)
  WHERE i.invoice_id = NEW.invoice_id;
END $$
-- BEGIN和END之间为主体(可以是sql代码，也可以调用存储过程)
DELIMITER ;

-- 应用
INSERT INTO payments
VALUES (DEFAULT, 5, 3, '2019-01-01', 10, 1)
```

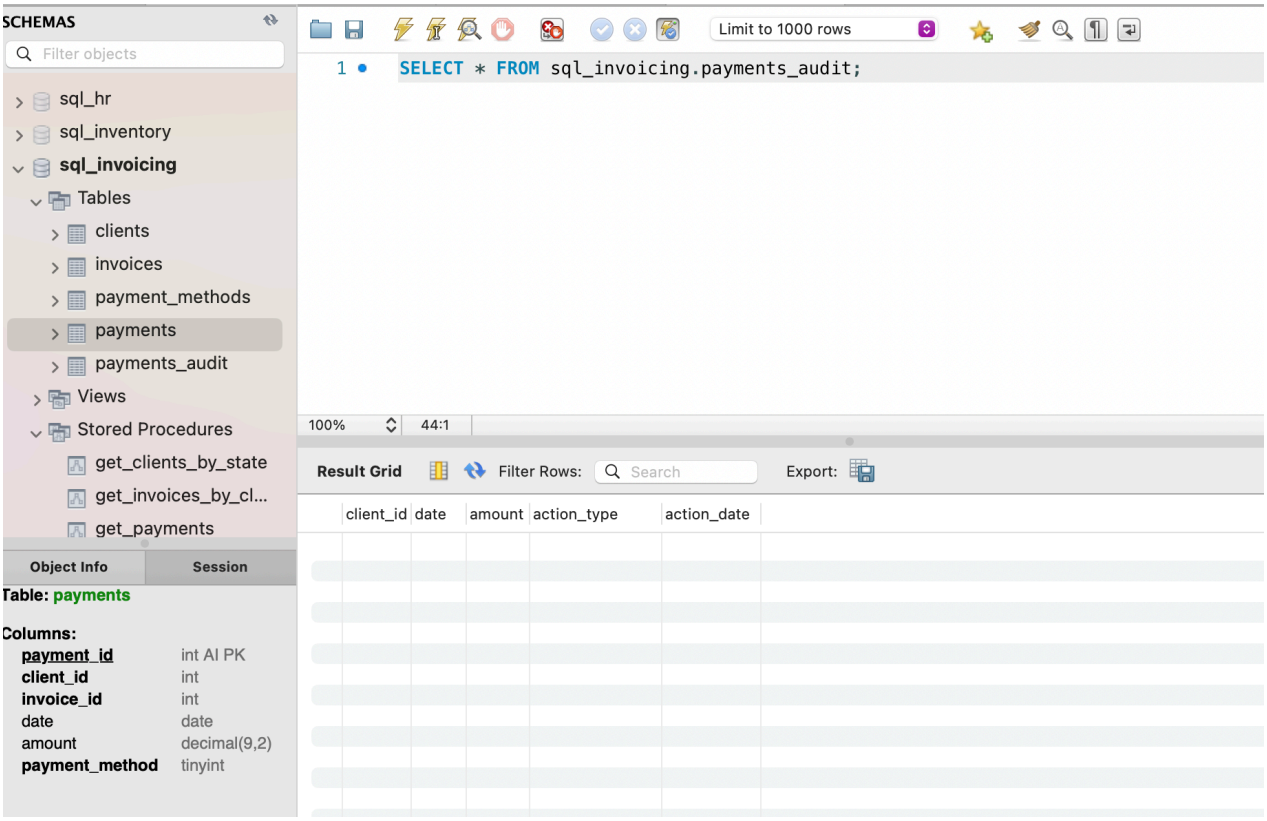
⚠️不可以修改触发器所在表中的数据（**原理不清楚，待补充实际例子**）

> The reason why you're typically not allowed to modify data in the same table that the trigger is defined on is to prevent potential conflicts, infinite loops, or unintended consequences. Here are a few key reasons:
>
> 1. **Avoiding Recursive Triggers**: If you were allowed to modify data in the same table that the trigger is defined on, it could potentially lead to recursive triggers. For example, an UPDATE trigger that modifies the same table could trigger itself, resulting in an infinite loop of trigger activations.
> 2. **Maintaining Data Integrity**: Modifying data in the same table that triggered the action could lead to unexpected or inconsistent results, especially if the trigger modifies the same rows that initiated the trigger action.
> 3. **Performance Considerations**: Allowing triggers to modify data in the same table could lead to performance issues, as each modification could trigger additional trigger activations, leading to cascading effects and potentially slowing down database operations.
> 4. **Clarity and Maintainability**: By restricting triggers from modifying the same

> table, it helps maintain clarity and makes it easier to understand the flow of data modifications within the database. It also helps prevent unintended side effects that could arise from trigger actions.

```
-- 查看触发器（创建好的触发器无法非常直观地看到）
SHOW TRIGGERS （查看所有触发器）
SHOW TRIGGERS LIKE 'payments%' （查看以payment开头命名的触发器，故命名时最好遵守命名原则"Table_Timing_Event"，方便以后查找）

-- 删除触发器
DROP TRIGGER IF EXISTS payments_after_insert
```

应用触发器进行审计：用于追踪信息系统修改过程（文件：create-payments-table.sql）




```
-- 目的在于：当payments表发生修改后，操作的内容和时间都会被记录在payments_audit这张表中
DELIMITER $$
CREATE TRIGGER payments_after_insert
  AFTER INSERT ON payments
  FOR EACH ROW
BEGIN
  UPDATE invoices i
  SET payment_total = payment_total + NEW.amount
  WHERE i.invoice_id = NEW.invoice_id;

  INSERT INTO payments_audit
```

```
  VALUES (NEW.client_id, NEW.date, NEW.amount, 'Insert', NOW());
END $$
DELIMITER ;

DELIMITER $$
CREATE TRIGGER payments_after_delete
  AFTER DELETE ON payments
  FOR EACH ROW
BEGIN
  UPDATE invoices i
  SET payment_total = payment_total - OLD.amount
  WHERE i.invoice_id = OLD.invoice_id;

  INSERT INTO payments_audit
  VALUES (OLD.client_id, OLD.date, OLD.amount, 'Delete', NOW());
END $$
DELIMITER ;
```

事件：EVENT

> Definition: A task (or block of SQL code) that gets executed according to a schedule.
>
> 可以是一次性的，也可以是规律性的，用于自动化维护数据库，如删除过期数据、复制表中数据、汇总数据生成报告等

```
-- 首先打开MySQL事件调度器
SHOW VARIABLES LIKE 'event%';
SET GLOBAL event_scheduler = ON/OFF (打开或关掉)

-- 创建一个事件：这里是定期删除过期的审计记录
DELIMITER $$

CREATE EVENT yearly_delete_stale_audit_rows
ON SCHEDULE
  -- AT '2019-05-01' (只执行一次)
  EVERY 1 YEAR STARTS '2019-01-01' ENDS '2029-01-01'(定期执行，这里是一年一次; STARTS/ENDS是可选项)
DO BEGIN
  DELETE FROM payments_audit
  WHERE action_date < NOW() - INTERVAL 1 YEAR
END $$

DELIMITER ;
-- 'NOW() - INTERVAL 1 YEAR' = DATESUB(NOW(), INTERVAL 1 YEAR)

-- 查看事件
SHOW EVENTS
```

```sql
SHOW EVENTS LIKE 'yearly%'

-- 删除事件
DROP EVENT IF EXISTS yearly_delete_stale_audit_rows

-- 修改事件(不用删除后再重建)
ALTER EVENT yearly_delete_stale_audit_rows
ON SCHEDULE
  EVERY 1 YEAR STARTS '2019-01-01' ENDS '2029-01-01'
DO BEGIN
  DELETE FROM payments_audit
  WHERE action_date < NOW() - INTERVAL 1 YEAR
END $$

DELIMITER ;
-- 还可用ALTER来暂时启用或禁用一个事件
ALTER EVENT yearly_delete_stale_audit_rows ENABLE/DISABLE
```

事务：TRANSACTION

> Definition: A group of SQL statements that represent a single unit of work

- Atomicity 强调整体性
- Consistency 强调协同性
- Isolation 强调事件之间的互斥性
- Durability 强调变更的永久性

```sql
-- 创建事务：下面是一个带有order_items的orders
START TRANSACTION;

INSERT INTO orders (customer_id, order_date, status)
VALUES(1, '2019-01-01', 1);

INSERT INTO order_items
VALUES (LAST_INSERT_ID(), 1,  1,  1);
-- LAST_INSERT_ID()返回最新插入的订单号
COMMIT; (用于关闭此事务)
ROLLBACK; (用于退回事务并撤消所有更改)
```

> 并发问题：多个用户同时修改统一数据
>
> 1. 数据丢失：晚提交的事务会覆盖早一些提交事务的修改
>
> - 解决方案：使用锁，即一个用户修改时，会锁住受影响的行，直到一个事务完全提交后，另一个事务才能进入同一行进行修改
>
> 2. 脏读：当一个事务读取了尚未被提交的数据

> - 解决方案：为事务建立隔离级别——READ COMMITTED即只能读取已提交的数据（一共有4个隔离级别）
>
> 3. 不一致读取：同一事务对数据在不同时间读取出不同的值
>
> - 解决方案：将这一事务与其他事务隔离——REPEATABLE READ以确保数据更改对此事务不可见
>
> 4. 幻读：一事务因其他事务正在修改但未提交而未完整提取数据
>
> - 解决方案：建立隔离——SERIALIZABLE以确保当有别的事务在更新数据时，此事务能知晓变动，如果有其他事务修改了可能影响查询结果的数据，此事务必须等它们完成以后再执行

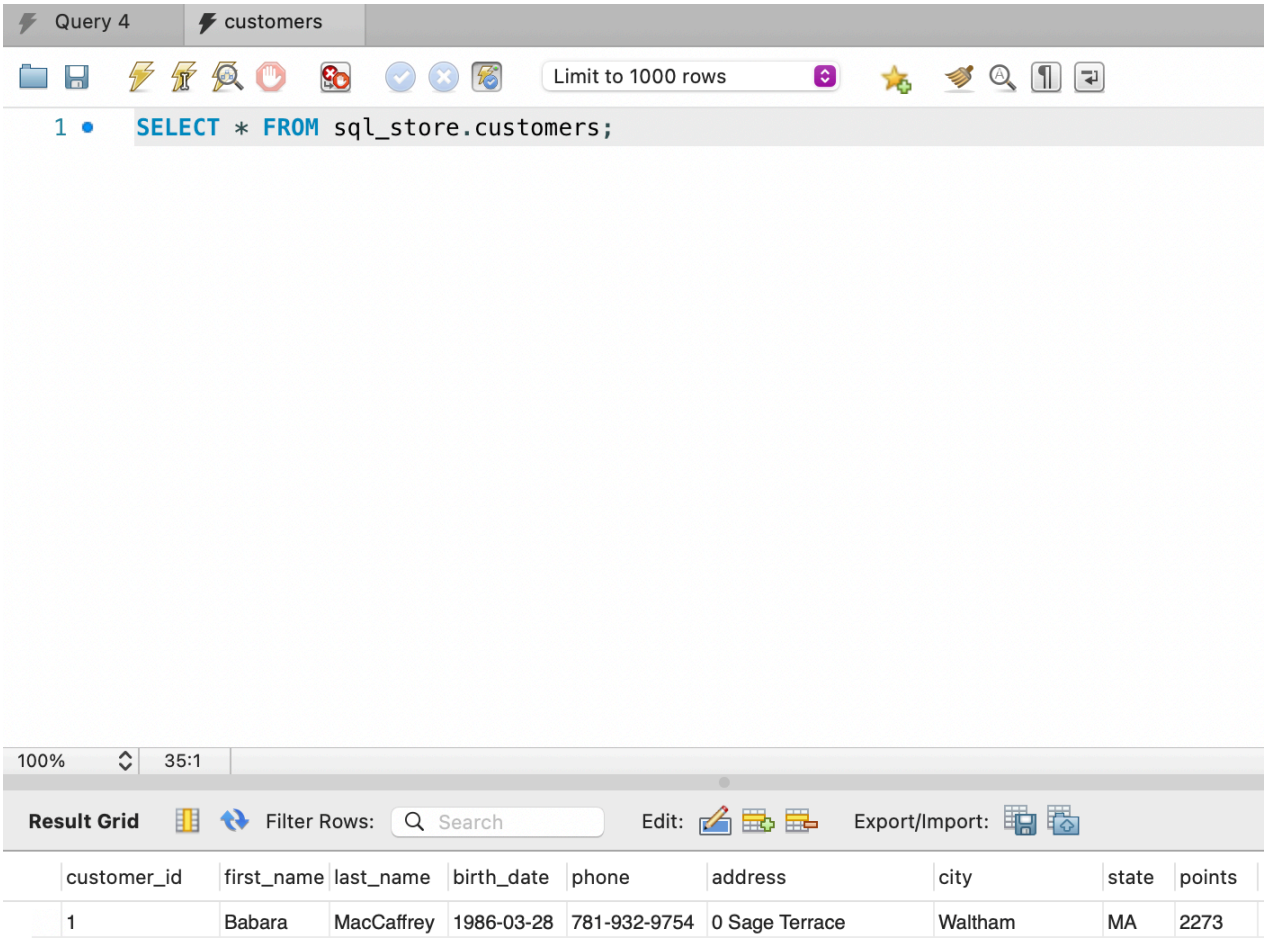
	Lost Updates	Dirty Reads	Non-repeating Reads	Phantom Reads
READ UNCOMMITTED				
READ COMMITTED		✅		
REPEATABLE READ	✅	✅	✅	
SERIALIZABLE	✅	✅	✅	✅

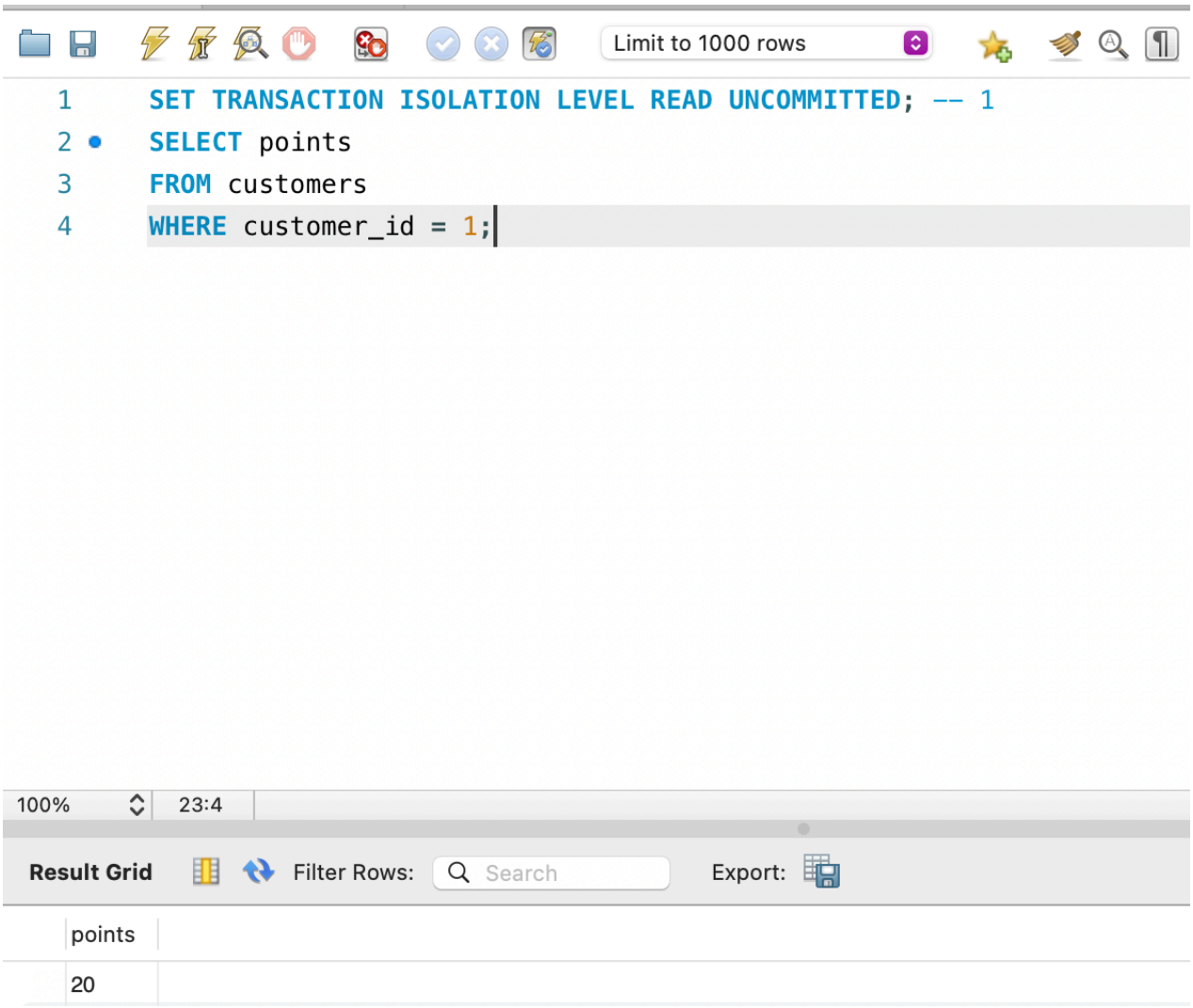
```
-- 查看当前事务隔离级别
SHOW VARIABLES LIKE 'transaction_isolation';
-- 更改当前事务隔离级别
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
-- 加了SESSION表示所有未来事务都是这个隔离级别
SET GLOBAL TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

- READ UNCOMMITTED

```sql
-- 用户A
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED; -- 1
SELECT points
FROM customers
WHERE customer_id = 1; -- 4

-- 用户B
START TRANSACTION; -- 2
UPDATE customers
SET points = 20
WHERE customer_id = 1; -- 3
ROLLBACK; -- 5
```

> 按上述顺序执行语句就会造成脏读，即用户B退回了对customer1积分的修改，即积分仍是2273，而用户A在用户B提交之前就提取了积分，故A得到的结果为20

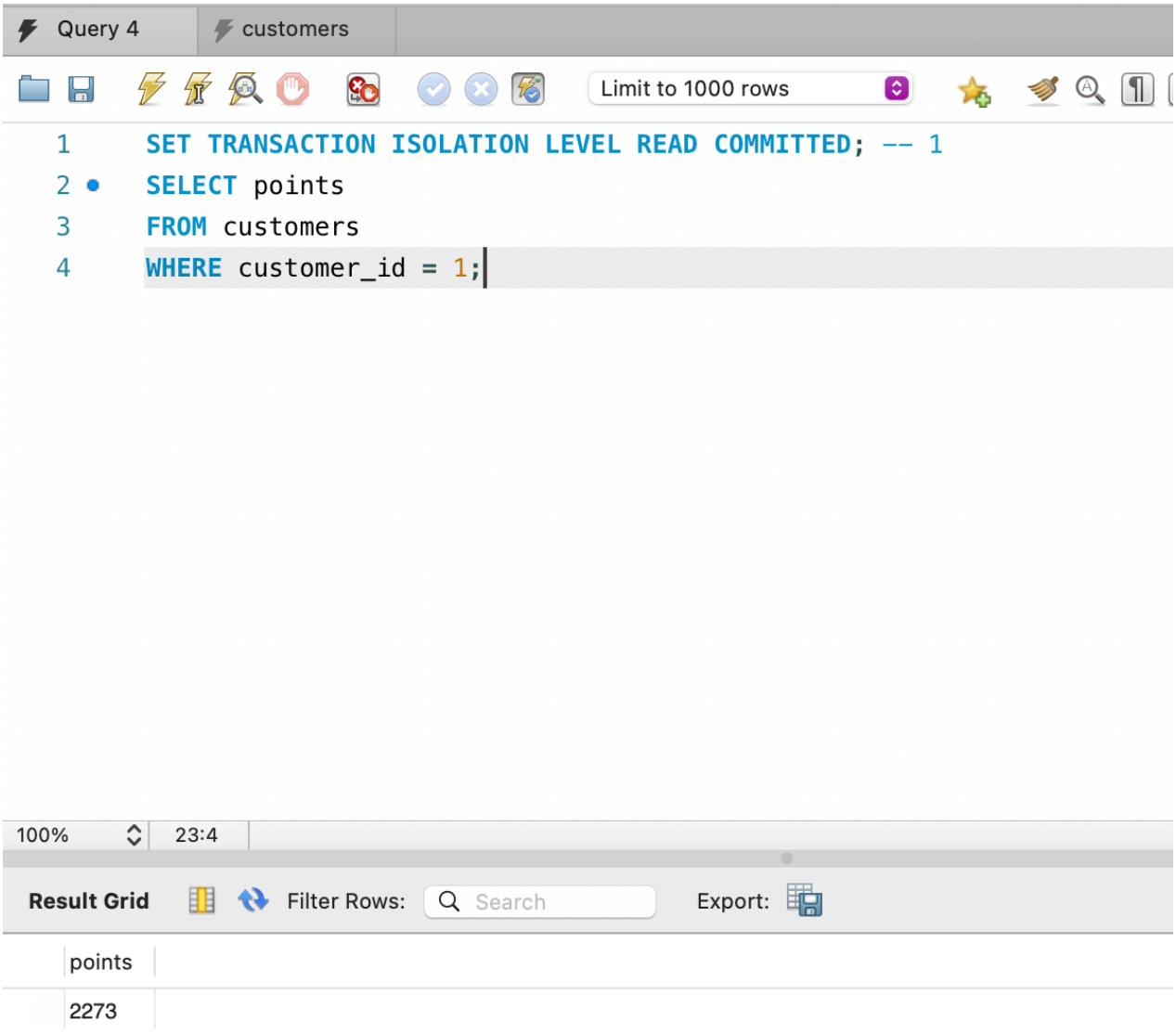




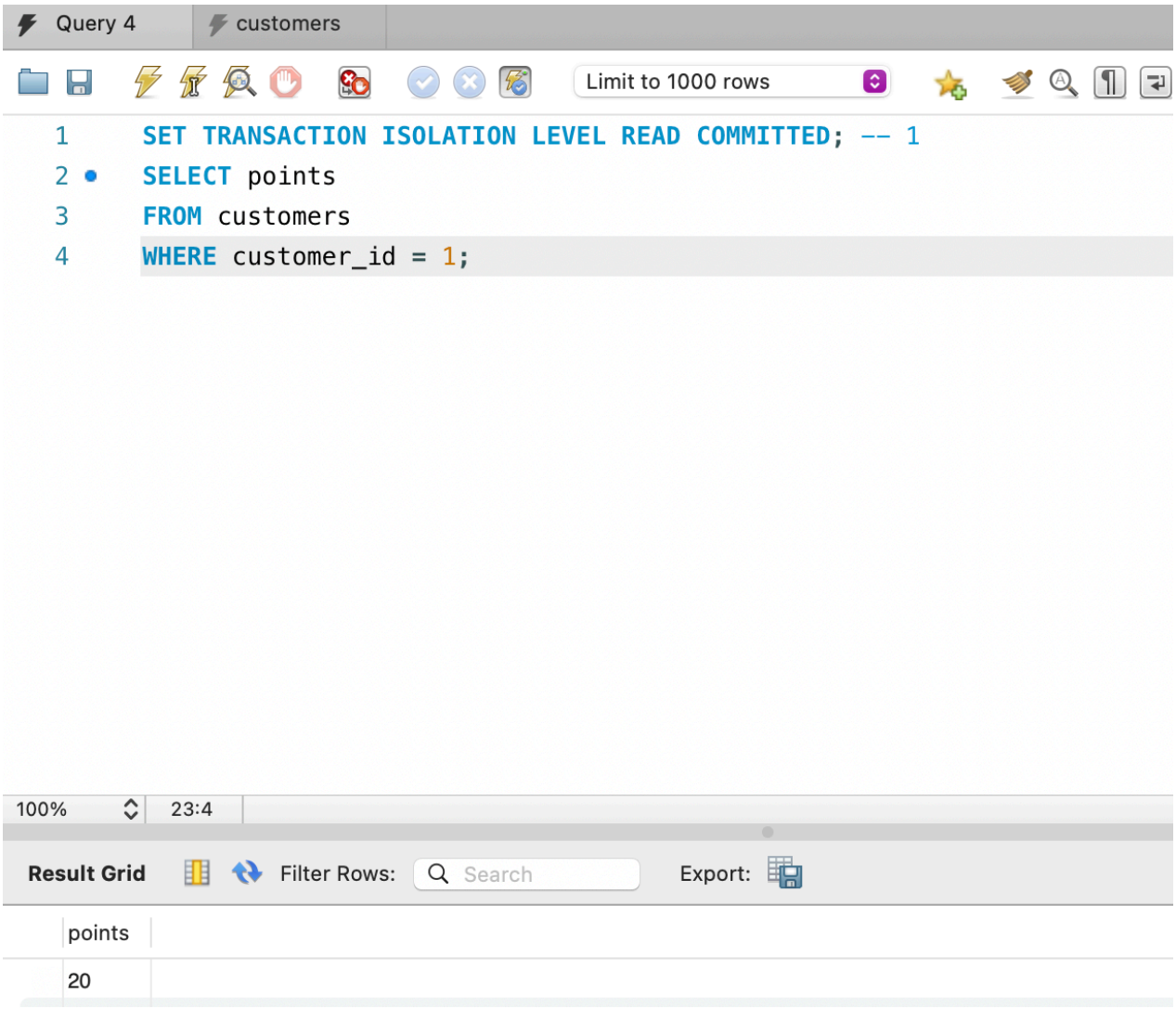
- READ COMMITTED

```sql
-- 用户A
SET TRANSACTION ISOLATION LEVEL READ COMMITTED; -- 1
SELECT points
FROM customers
WHERE customer_id = 1; -- 4 and 6

-- 用户B
START TRANSACTION; -- 2
UPDATE customers
SET points = 20
WHERE customer_id = 1; -- 3
COMMIT; -- 5
```

> 因为用户B未提交，故用户A提取时(第4步)仍是修改前的积分数2273

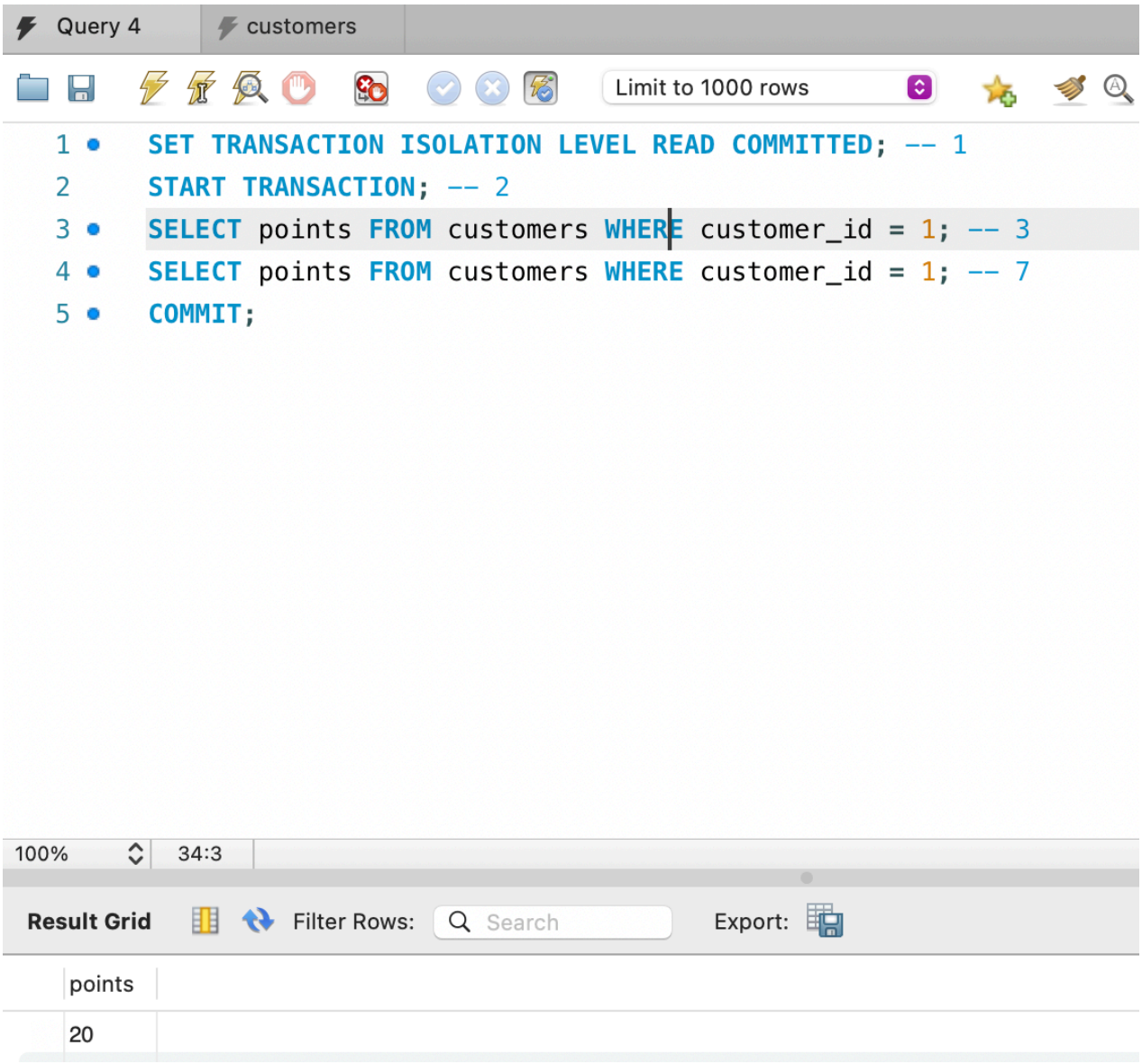


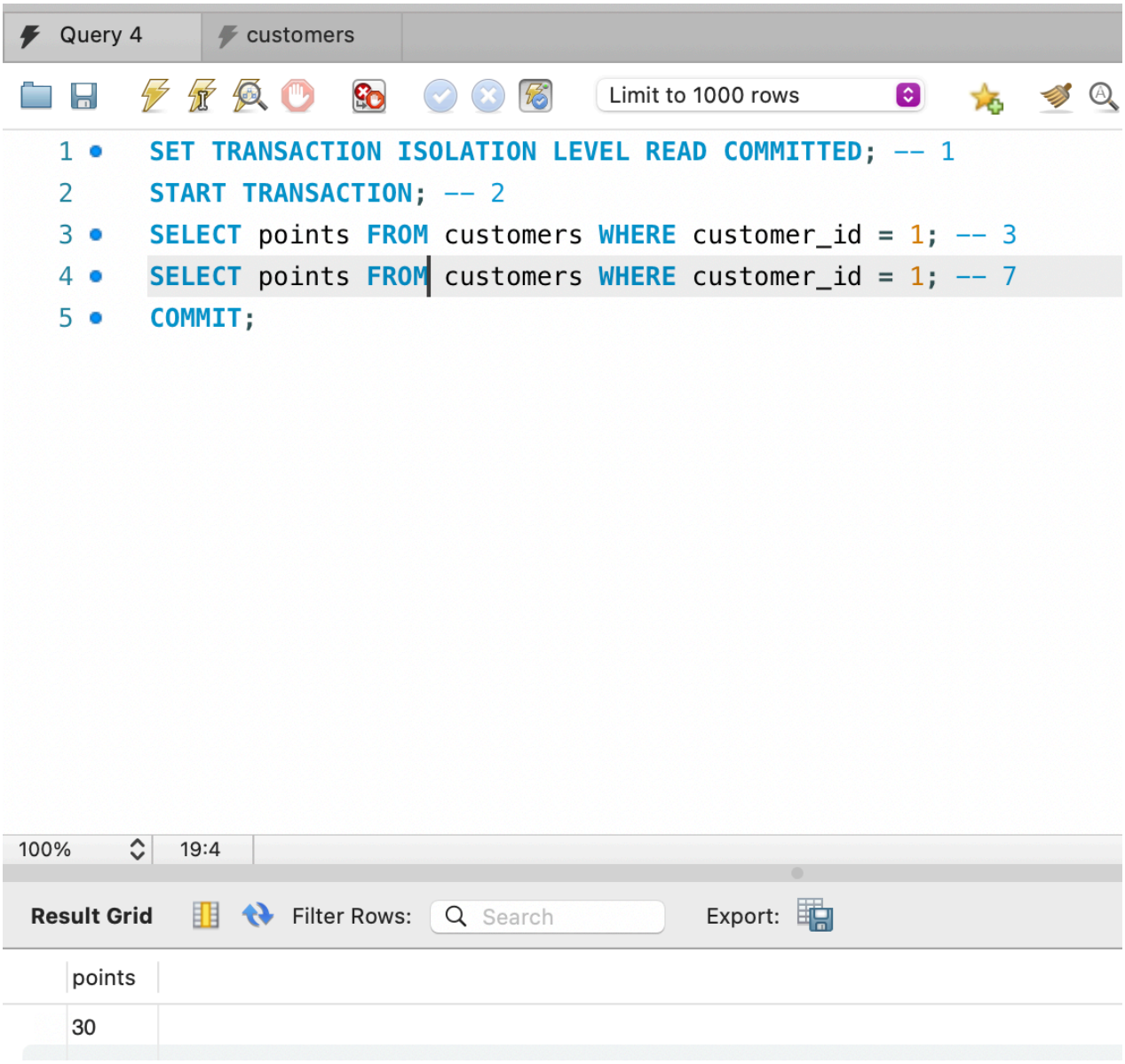
> 用户B进行第5步后，用户A进行第6步时，积分值更改为20




```sql
-- 用户A
SET TRANSACTION ISOLATION LEVEL READ COMMITTED; -- 1
START TRANSACTION; -- 2
SELECT points FROM customers WHERE customer_id = 1; -- 3
SELECT points FROM customers WHERE customer_id = 1; -- 7
COMMIT; -- 8

-- 用户B
START TRANSACTION; -- 4
UPDATE customers
SET points = 30
WHERE customer_id = 1; -- 5
COMMIT; -- 6
```

> 用户A第3步和第7步结果如下，产生不一致读取问题

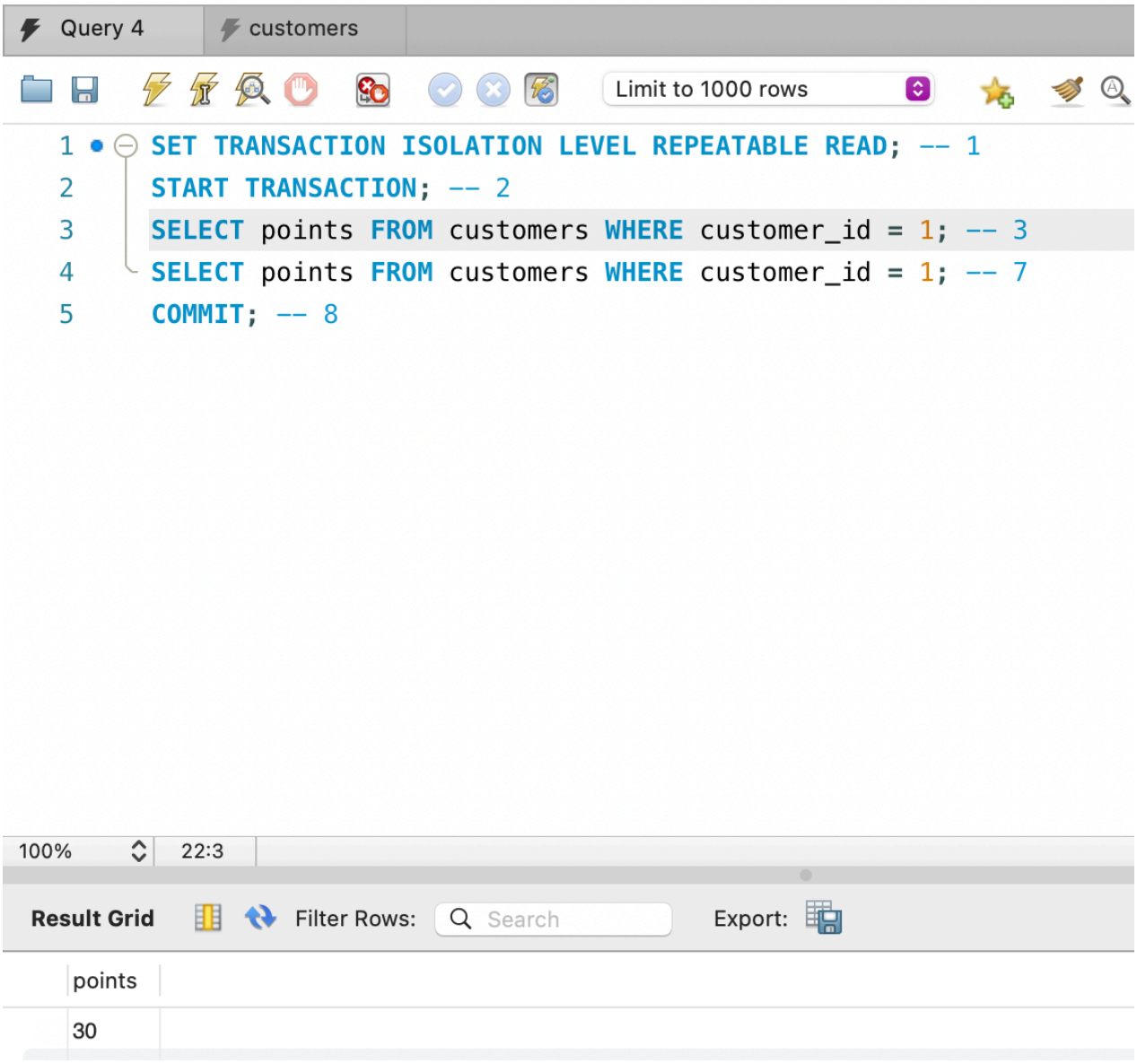

Query 4
customers
Limit to 1000 rows
1 SET TRANSACTION ISOLATION LEVEL READ COMMITTED; -- 1
2 START TRANSACTION; -- 2
3 SELECT points FROM customers WHERE customer_id = 1; -- 3
4 SELECT points FROM customers WHERE customer_id = 1; -- 7
5 COMMIT;
100%
34:3
Result Grid
Filter Rows:
Search
Export:
points
20

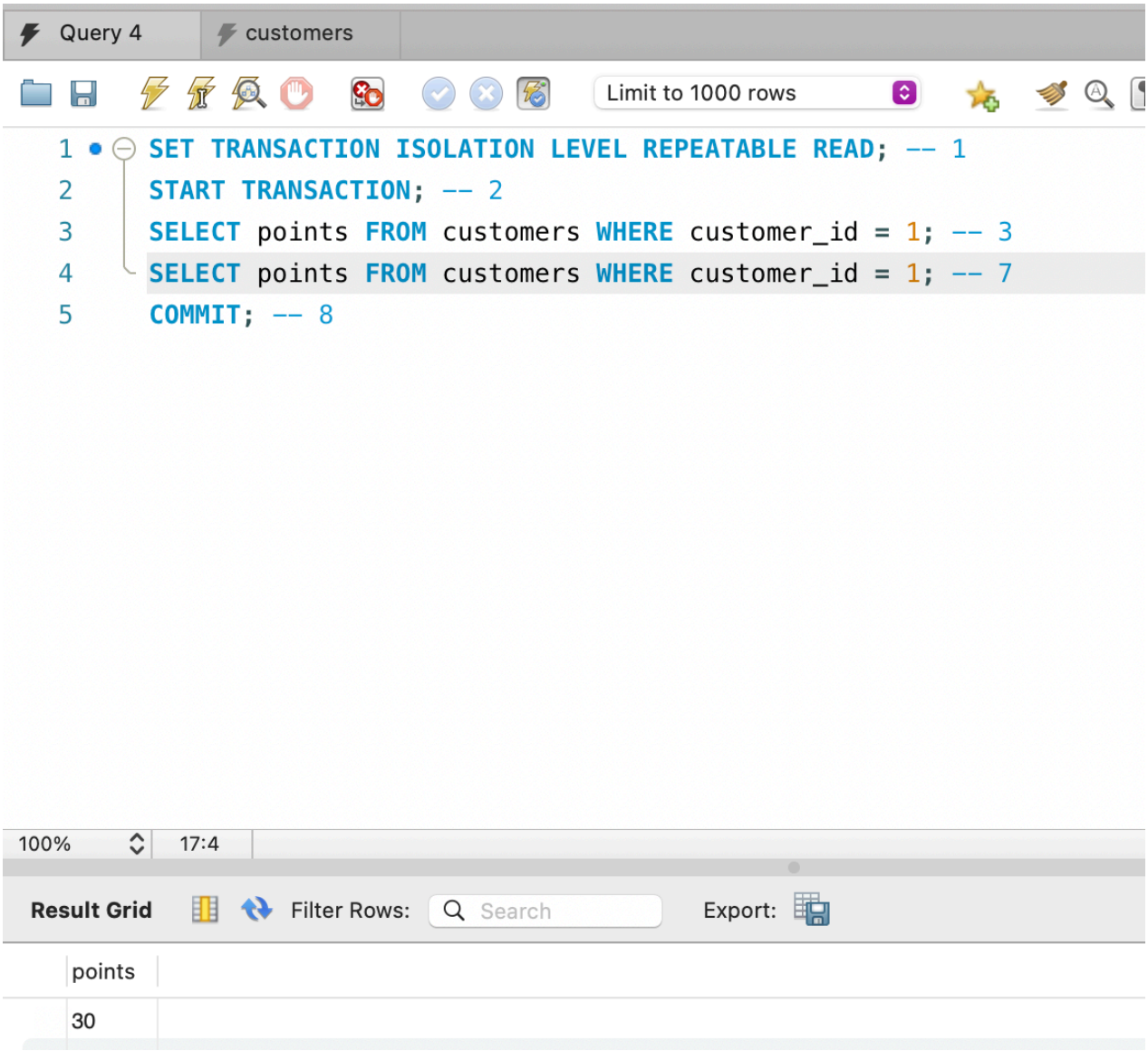


- REPEATABLE READ

```sql
-- 用户A
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ; -- 1
START TRANSACTION; -- 2
SELECT points FROM customers WHERE customer_id = 1; -- 3
SELECT points FROM customers WHERE customer_id = 1; -- 7
COMMIT; -- 8

-- 用户B
START TRANSACTION; -- 4
UPDATE customers
SET points = 30
WHERE customer_id = 1; -- 5
COMMIT; -- 6
```

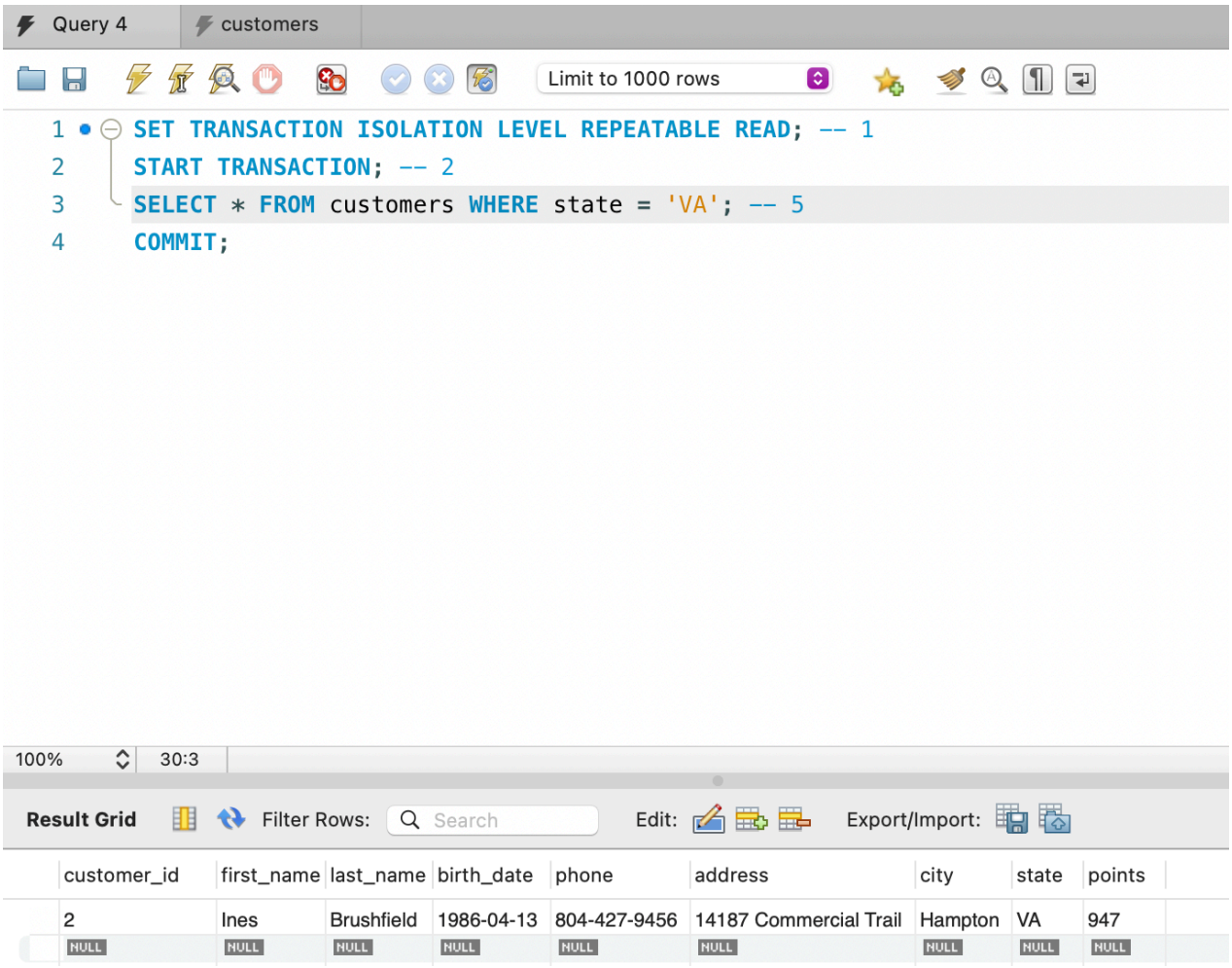
> 用户A第3步和第7步结果如下，结果一致解决不可重复读问题

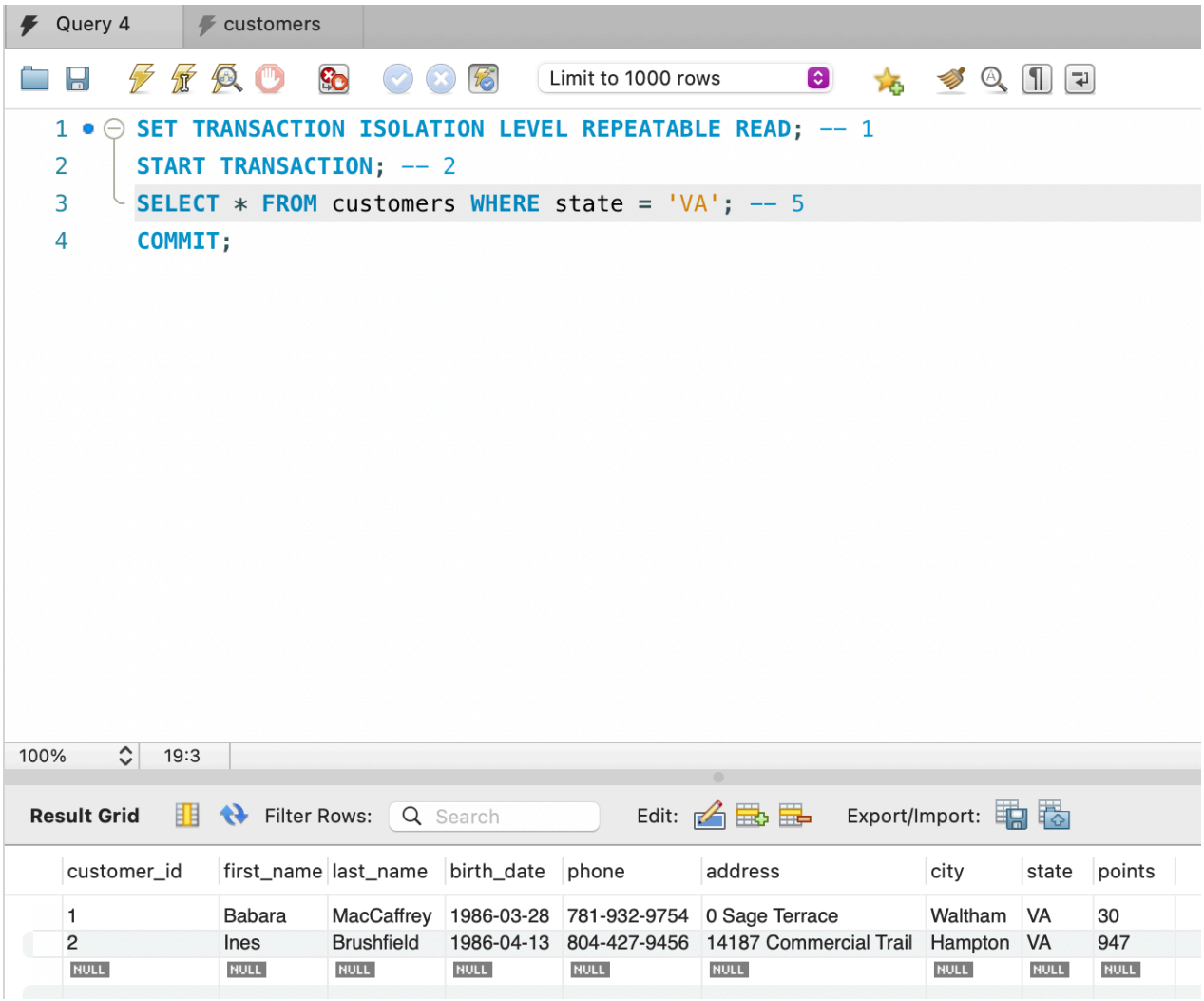

Query 4
customers
Limit to 1000 rows
1 SET TRANSACTION ISOLATION LEVEL REPEATABLE READ; -- 1
2 START TRANSACTION; -- 2
3 SELECT points FROM customers WHERE customer_id = 1; -- 3
4 SELECT points FROM customers WHERE customer_id = 1; -- 7
5 COMMIT; -- 8
100%
22:3
Result Grid
Filter Rows:
Search
Export:
points
30




```sql
-- 用户A
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ; -- 1
START TRANSACTION; -- 2
SELECT * FROM customers WHERE state = 'VA'; -- 5 and 8
COMMIT; -- 7

-- 用户B
START TRANSACTION; -- 3
UPDATE customers
SET state = 'VA'
WHERE customer_id = 1; -- 4
COMMIT; -- 6
```

> 用户A第5步提取的数据只有一条，而当用户B进行了第6步之后，实际上在"VA"的顾客有两名，即执行第8步得到如下结果

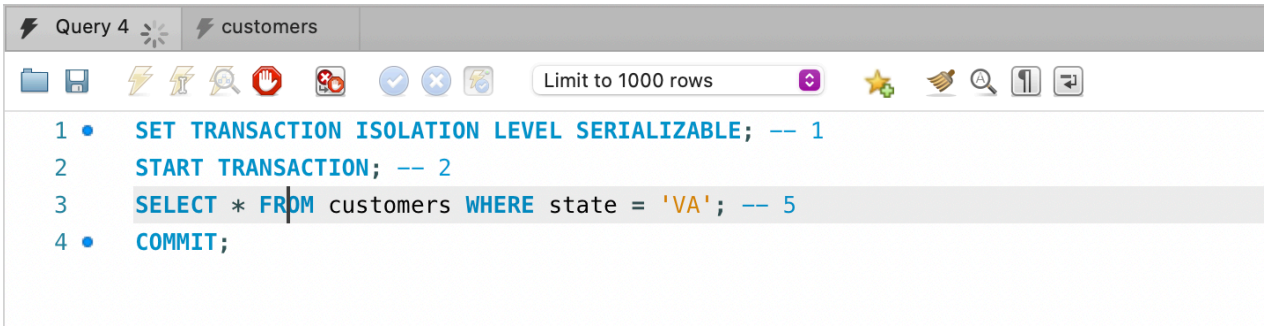

Query 4
customers
Limit to 1000 rows
1 SET TRANSACTION ISOLATION LEVEL REPEATABLE READ; -- 1
2 START TRANSACTION; -- 2
3 SELECT * FROM customers WHERE state = 'VA'; -- 5
4 COMMIT;
100%
30:3
Result Grid
Filter Rows:
Search
Edit:
Export/Import:
customer_id
first_name
last_name
birth_date
phone
address
city
state
points
2
Ines
Brushfield
1986-04-13
804-427-9456
14187 Commercial Trail
Hampton
VA
947
NULL

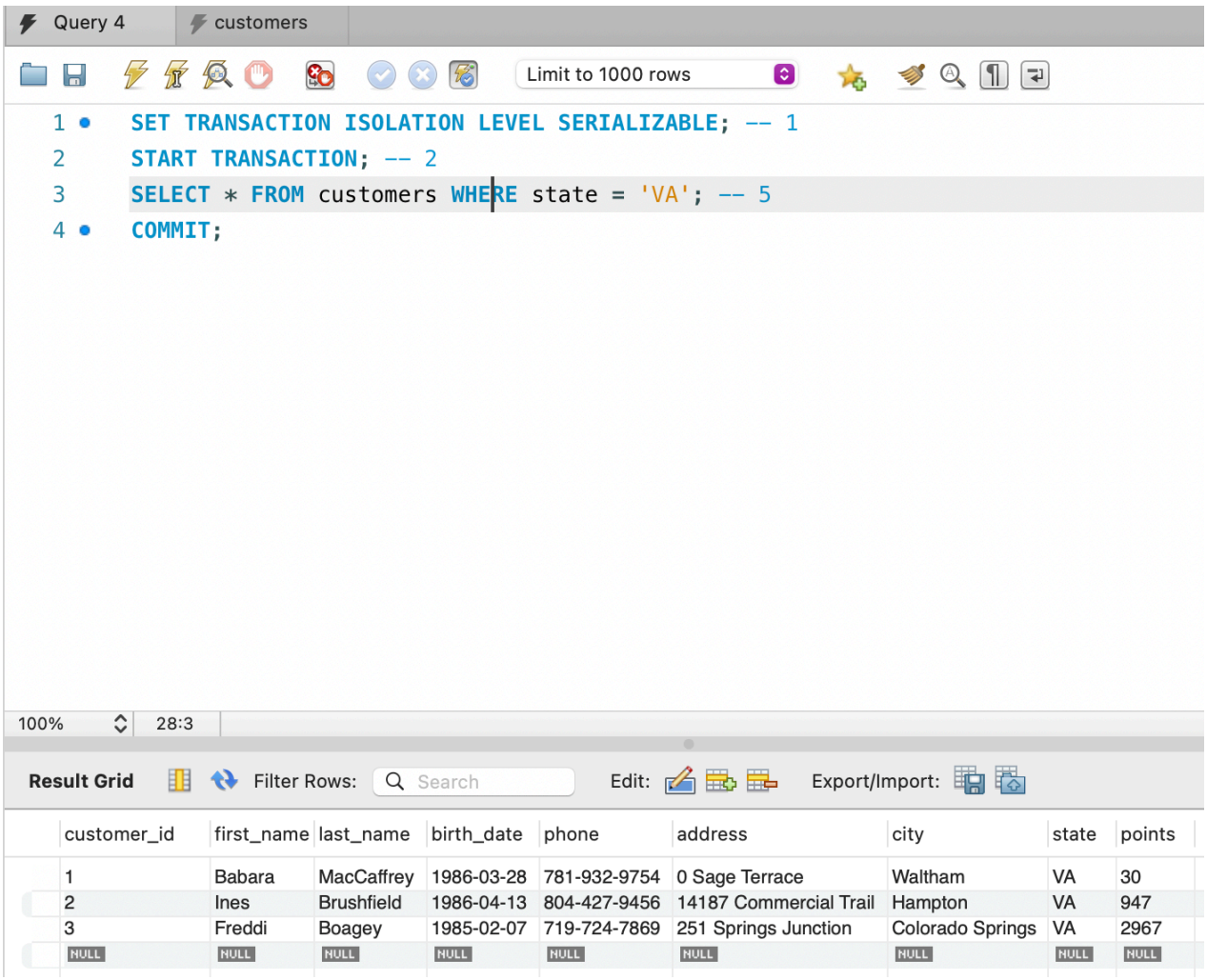


- SERIALIZABLE

```
-- 用户A
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE; -- 1
START TRANSACTION; -- 2
SELECT * FROM customers WHERE state = 'VA'; -- 5
COMMIT; -- 7

-- 用户B
START TRANSACTION; -- 3
UPDATE customers
SET state = 'VA'
WHERE customer_id = 3; -- 4
COMMIT; -- 6
```

> 用户A执行第5步后并没有返回结果，它在等待另一个事务的结束，当用户B执行了第6步后，用户A第5步的结果才出来



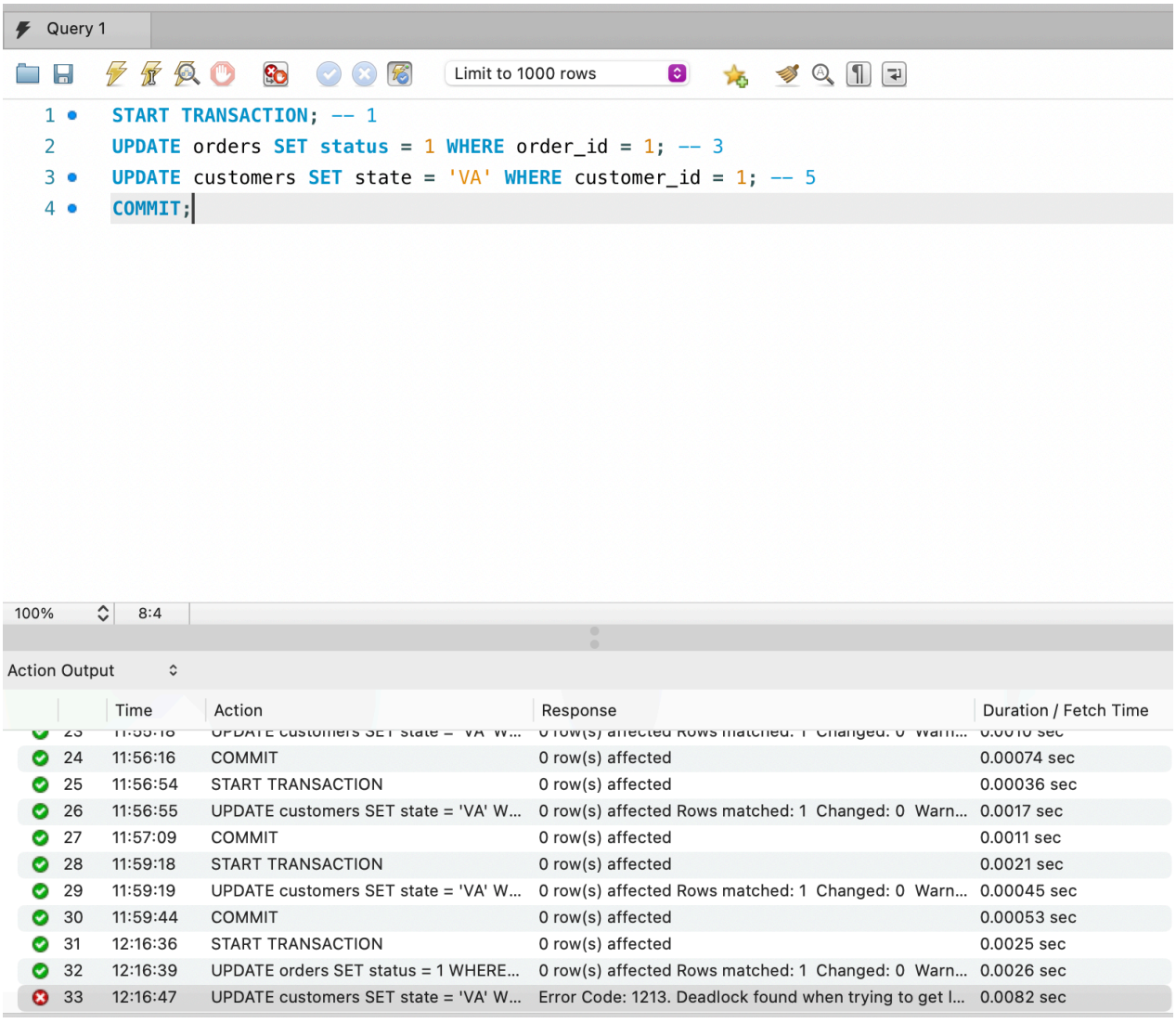


死锁：DEADLOCK

> 当不同事务均因握住了别的事务需要的“锁”而无法完成的情况

```sql
-- 用户A
START TRANSACTION; -- 1
UPDATE customers SET state = 'VA' WHERE customer_id = 1; -- 2
UPDATE orders SET status = 1 WHERE order_id = 1; -- 4
COMMIT;

-- 用户B
START TRANSACTION; -- 1
UPDATE orders SET status = 1 WHERE order_id = 1; -- 3
UPDATE customers SET state = 'VA' WHERE customer_id = 1; -- 5
COMMIT;
```

> 用户A执行第2步时，会锁住受影响的行；用户B执行他的第二步(即3)时，同样会锁住受影响的行，而用户A想要进行第4步必须等用户B，而用户B想要执行第5步必须等用户A，因此形成死锁（系统会报错），死锁无法彻底避免，只能减小其出现的可能性，一个解决方案为遵循相同的修改顺序。



数据类型

- **String Types 字符串类型**

```
CHAR for fixed-length
VARCHAR(50) for short strings
VARCHAR(225) for medium-length strings
VARCHAR max:64KB (长度范围内推荐使用，理由：可被编入索引)
MEDIUMTEXT max:16MB (适用于存储JSON对象/SCV字符串/短中长度的书)
LONGTEXT max:4GB (适用于存储textbooks和日志等)
TINYTEXT max:255 bytes
TEXT max:64KB
-- English占用1 byte；European/Middle-eastern占用2 bytes；Asian占用3 bytes
```

- **Numeric Types 数值类型**
  - Integer Types 整数类型

Types	Storage(Bytes)	Range of Values
TINYINT	1	[-128, 127]
UNSIGNED TINYINT	1	[0, 225]
SMALLINT	2	[-32K, 32K]
MEDIUMINT	3	[-8M, 8M]
INT	4	[-2B, 2B]
BIGINT	8	[-9Z, 9Z]

完整的整数类型在这里

> Tips: Use the smallest data type that suits your needs

  - Fixed-point and Floating-point Types 定点数和浮点数类型

```
DECIMAL(9, 2): 9位数 包括小数点后两位 = DEC/NUMERIC/FIXED
-- 下面两个取近似值
FLOAT 4b
DOUBLE 8b
```

  - Boolean Types 布尔类型

```
BOOL/BOOLEAN: TRUE = 1 / FALSE = 0

UPDATE posts
SET is_published = 1/TRUE
```

- Enum and Set Types 枚举和集合类型

```
ENUM('small', 'medium', 'large') 将输入值限制在某个范围内
SET(...)
```

- **Data and Time Types 日期和时间类型**

```
DATE
TIME
DATETIME 8b
TIMESTAMP 4b (up to 2038)
YEAR
```

- **Blob Types 二进制长对象类型**

> 用于存储大型二进制数据，如图像/视频/PDF/Word文件等

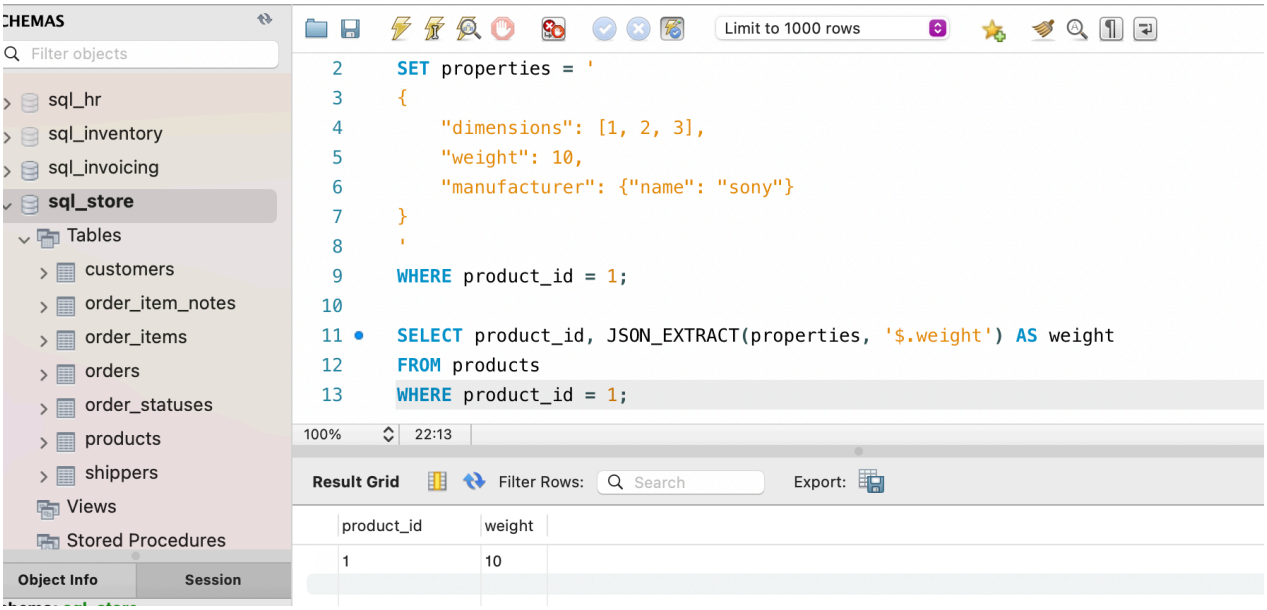
Categoty	Max Storage
TINYBLOB	255b
BLOB	65KB
MEDIUMBLOB	16MB
LONGBLOB	4GB

- **Spatial Types**
- **JSON Types**

```
-- products表添加一列名为'properties'类型为'JSON'的列，用于对每个种类的产品标明其不同的属性
UPDATE products
SET properties = '
{
  "dimensions": [1, 2, 3],
  "weight": 10,
  "manufacturer": {"name": "sony"}
}
'
WHERE product_id = 1;
=
UPDATE products
SET properties = JSON_OBJECT(
  'weight', 10,
  'dimensions', JSON_ARRAY(1, 2, 3),
  'manufacturer', JSON_OBJECT('name', 'sony')
```

```sql
)
WHERE product_id = 1;

-- 应用
SELECT product_id, JSON_EXTRACT(properties, '$.weight') AS weight
FROM products
WHERE product_id = 1;
=
SELECT product_id, properties -> '$.weight'
FROM products
WHERE product_id = 1;
-- '->'为列路径运算符
```

应用结果如下：



设计数据库

- 数据建模：为要存储在数据库中的数据创建模型
  - 理解和分析业务需求
  - 构建业务的概念模型：识别业务中的实体、事物或概念以及它们之间的关系
  - 构建逻辑模型：生成一个数据模型或数据结构用以存储数据
  - 构建实体模型：围绕特定数据库技术对逻辑模型的实现

概念模型：Conceptual Models

> Represents the entities and their relationships
>
> 需要可视化方法：实体关系图(Entity Relationship)/UML图(标准建模语言图)
>
> 建模工具：draw.io/LucidCharts(在线网页版)

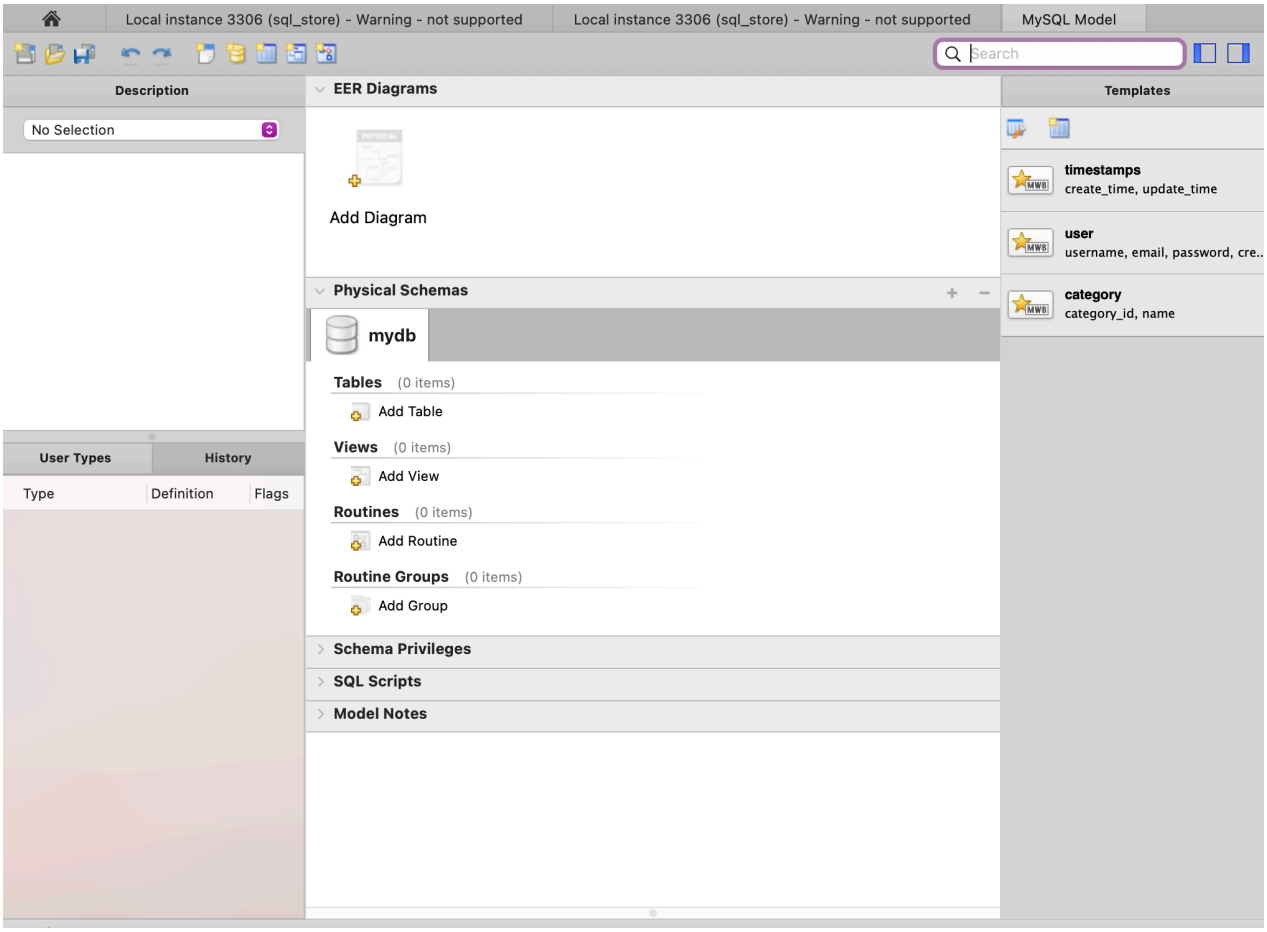
逻辑模型：Logical Models

> 独立于数据库技术，如定义name的属性为字符串，而不是VARCHAR这种具体实现
>
> 定义实体之间的关系：一对一/一对多/多对多

概念模型是更宏观的概念，着眼于业务，大致列举出业务涉及实体以及每个实体的属性。逻辑模型需要对概念模型进行进一步拆解，首先是每个实体属性的拆解，大拆小并确定数据类型；其次是厘清每个实体之间的关系，这个关系也需要数据化（即能用数据表达，每个表的关联点在哪里）

实体模型：Physical Models

> 强调实操性
>
> 创建路径：File----New model



> EER: enhance entity relationship 增强实体关系（也可用来创建实体关系图）创建好后：Database----Forward Engineer----默认情况下一直continue
>
> mydb: 默认的数据库名，右键Edit Schema可改

实体模型中要思考：

1. 列名的具体数据类型，在全面的情况下尽量小以节省存储空间
2. 每个表的主键，以连接其他表（主键的选择有讲究：唯一性/稳定性/简洁性）
3. 选择表间关系：

1. 一对多：先选择外键表（外键为在一张表中引用了另一张表主键的那列），再选择参考表。下方有外键限制选项，可根据具体情况选择是否根据参考表的变化而变化
2. 多对多：关系型数据库中没有“多对多”关系，只有“一对一”和“一对多”，故需要引入“链接表”，将“多对多”关系拆解成两个“一对多”关系

标准化：Normalization

> 审查设计并确保它遵循一些防止数据重复的预定义规则的过程
>
> 有七范式，掌握前三够用了（每条规则都假设已采用前面几条规则）
>
> 1. 第一范式要求一行中的每个单元格都应该有单一值，且不能出现重复列
>
> > 有多个值的列最好新建一张表单独储值
>
> 2. 第二范式要求每张表都应该有一个单一目的，即它只能代表一种且仅有一种实体类型，而那张表中的每一列都应该用来描述那个实体（表中不能有不属于它的属性存在）
>
> > 有不属于(本质上是易重复，不方便后续修改)该表的属性应该单独建表
>
> 3. 第三范式表示，表中的列不应派生自其他列(即线性无关)
>
> > 可以由公式生成新的一列，方便相关数据的联动

模型的逆向工程

> 已知数据库求模型
>
> 路径：Database----Reverse Engineer----选择想要知道模型的数据库（可以知道别人如何建造模型的）

```sql
-- 创建数据库(这里指没有任何表的库)
CREATE DATABASE IF NOT EXISTS sql_store2;

-- 删除数据库
DROP DATABASE IF EXISTS sql_store2;

-- 创建表
CREATE TABLE customers
(
  customer_id INT PRIMARY KEY AUTO_INCREMENT,
  first_name  VARCHAR(50) NOT NULL,
  points      INT NOT NULL DEFAULT 0, (默认值为0)
  email       VARCHAR(255) NOT NULL UNIQUE
);
-- "列名 数据类型 属性"排列顺序

-- 更改表
ALTER TABLE customers
```

```
  ADD last_name VARCHAR(50) NOT NULL AFTER first_name,
  MODIFY COLUMN first_name VARCHAR(55) DEFAULT '',
  DROP points;
-- COLUMN可写可不写
-- 最好在数据库中修改表!!!

-- 创建表间关系（接创建表语句）
DROP TABLE IF EXISTS orders;
DROP TABLE IF EXISTS customers;

CREATE TABLE customers
(
  customer_id INT PRIMARY KEY AUTO_INCREMENT,
  first_name  VARCHAR(50) NOT NULL,
  points      INT NOT NULL DEFAULT 0, (默认值为0)
  email       VARCHAR(255) NOT NULL UNIQUE
);
CREATE TABLE orders
(
  order_id    INT PRIMARY KEY,
  customer_id INT NOT NULL
  FOREIGN KEY fk_orders_customers (customer_id)
    REFERENCES customers (customer_id)
    ON UPDATE CASCADE/SET NULL/NO ACTION/RESTRICT(定义更新和删除行为：级联or限制or不响应)
    ON DELETE NO ACTION
);
-- 外键命名原则："fk_子表名_父表名(被应用外键的列名)"
-- 因为建立了表间关系，故不能轻易删掉customers这张表，需要先删除orders表才能删除customers表，所以注意调换DROP TABLE的顺序

-- 更改主键/外键约束
ALTER TABLE orders
  ADD PRIMARY KEY (order_id),
  DROP PRIMARY KEY,             (DROP时不用明确列名)
  DROP FOREIGN KEY fk_orders_customers,
  ADD FOREIGN KEY fk_orders_customers(customer_id)
    REFERENCES customers (customer_id)
    ON UPDATE CASCADE
    ON DELETE NO ACTION;
```

字符集和排序规则(可以优化内存，暂时用不到)

> 字符集中的每个字符都有其对应的数值表示，字符集有很多类

```
SHOW CHARSET
```

存储引擎

> 决定数据如何被存储，以及哪些功能可供使用
>
> 常用引擎：MyISAM/InnoDB(better)

```
SHOW ENGINES
```

索引：Indexes

> 数据库引擎用来快速查找数据的数据结构，其最终目的是加快运行较慢的查询

```
EXPLAIN SELECT customer_id FROM customers WHERE state = 'CA'
```

Result Grid Filter Rows: Search Export:

id	select_ty...	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	customers	NULL	ALL	NULL	NULL	NULL	NULL	1010	10.00	Using where

> type为ALL表示SQL进行了全表扫描和读取（rows = 1010）----没有加索引

```
-- 创建索引
CREATE INDEX idx_state ON customers (state); (将索引放置在customers表的state列)
EXPLAIN SELECT customer_id FROM customers WHERE state = 'CA'
```

Result Grid Filter Rows: Search Export:

id	select_ty...	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	customers	NULL	ref	idx_state	idx_state	8	const	112	100.00	Using where

> SQL只扫描了112行

```
-- 查看索引
SHOW INDEXES IN customers;
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Expression
customers	0	PRIMARY	1	customer_id	A	10	NULL	NULL		BTREE			YES	NULL
customers	1	idx_state	1	state	A	48	NULL	NULL		BTREE			YES	NULL
customers	1	idx_point	1	points	A	788	NULL	NULL		BTREE			YES	NULL

> Key_name： PRIMARY Mysql会给主键自动生成一个索引（也会为外键自动创建索引）
>
> Collation：排序规则 A代表升序/D代表降序
>
> Index_type：二叉树

- 前缀索引：Prefix Indexes

```
CREATE INDEX idx_lastname ON customers (last_name(20))
-- 因为last_name是字符串，可以指定索引中要包含的字符数（减少索引占用空间），对于CHAR和VARCHAR是选填项，对于TEXT和BLOB是必填项
-- 必须通过观察数据找到最佳字符数，标准为"足以唯一识别"
-- 如何选择最佳字符数：最大化索引中唯一值的数量
SELECT
  COUNT(DISTINCT LEFT(last_name, 1)),
  COUNT(DISTINCT LEFT(last_name, 5)),
  COUNT(DISTINCT LEFT(last_name, 10))
FROM customers;
-- 该过程有点像用岭迹找到最佳超参数
```

- 全文索引：Full-text Indexes（模糊搜索）

```
CREATE FULLTEXT IINDEX idx_title_body ON posts(title, body);

SELECT
  *,
  MATCH(title, body) AGAINST ('react redux')
FROM posts
WHERE MATCH(title, body) AGAINST ('react redux'); (全文搜索有两种模式，这里是默认模式即自然语言模式)
-- 会返回所有标题或正文中包含一个或两个关键字的文章，这些单词可以按照任何顺序排列，也可以被一个或多个单词分割
-- MATCH(title, body) AGAINST ('react redux')会返回相关性
-- AGAINST ('react redux' IN BOOLEAN MODE)是另一种模式即布尔模式，该模式下可以包括或排除某些单词
```

- 复合索引：Composite Indexes

```
EXPLAIN SELECT customer_id FROM customers
WHERE state = 'CA' AND points > 1000;
```

Result Grid Filter Rows: Search Export:

id	select_ty...	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	customers	NULL	ref	idx_state,idx_point	idx_state	8	const	112	52.28	Using index condition; U

> 首先可以看到可供选择的索引有两个，但不管有多少索引，MySQL最多只会选择一个。
>
> 在这个语句下，选择的索引能够缩小state的搜索范围，但必须对所有(112行)的数据进行遍历才能筛选出points大于1000的顾客，因为state索引中没有每位顾客的积分点，这是就需要复合索引——允许对多列建立索引(MySQL最多允许16列)

```sql
CREATE INDEX idx_state_points ON customers (state, points);
-- 括号中列的顺序是重要的
EXPLAIN SELECT customer_id FROM customers
WHERE state = 'CA' AND points > 1000;
```

Result Grid Filter Rows: Search Export:

id	select_ty...	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered
1	SIMPLE	customers	NULL	range	idx_state,idx_point,idx_state_points	idx_state_points	12	NULL	58	100.00

> 上图可见，扫描行数从112减少为58

复合索引中的列顺序原则(不一定始终正确，需要考虑数据结构)

> 把更频繁使用的列排在最前面
>
> 把基数更高的列排在最前面
>
> > 基数(Cardinality)：索引中唯一值的数量

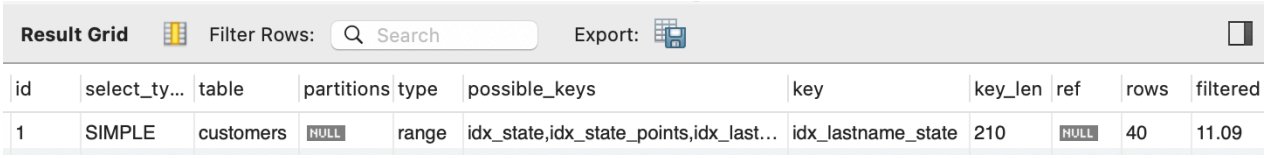
```sql
SELECT customer_id
FROM customers
WHERE state = 'CA' AND last_name LIKE 'A%';
-- 如何选择复合索引的列顺序，操作如下：
SELECT
  COUNT(DISTINCT state),
  COUNT(DISTINCT last_name)
FROM customers
```

Result Grid Filter Rows: Search

COUNT(DISTINCT stat...	COUNT(DISTINCT last_nam...
48	996

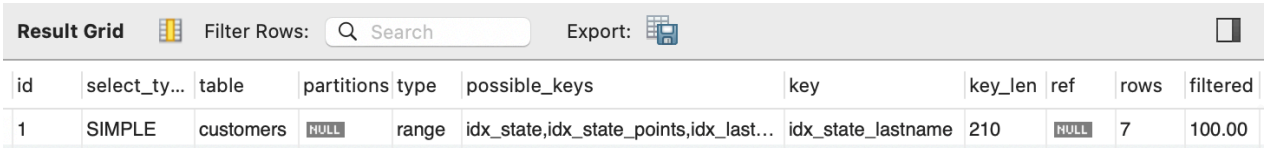
> 看结果应该选择last_name在前，因为它能把数据分成更小份

```sql
-- last_name在前
CREATE INDEX idx_lastname_state ON customers(last_name, state);
EXPLAIN SELECT customer_id
FROM customers
WHERE state = 'CA' AND last_name LIKE 'A%';

-- state在前
CREATE INDEX idx_state_lastname ON customers(state, last_name);
EXPLAIN SELECT customer_id
FROM customers
WHERE state = 'CA' AND last_name LIKE 'A%';
```



Result Grid Filter Rows: Search Export:

id	select_ty...	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered
1	SIMPLE	customers	NULL	range	idx_state,idx_state_points,idx_last...	idx_lastname_state	210	NULL	40	11.09



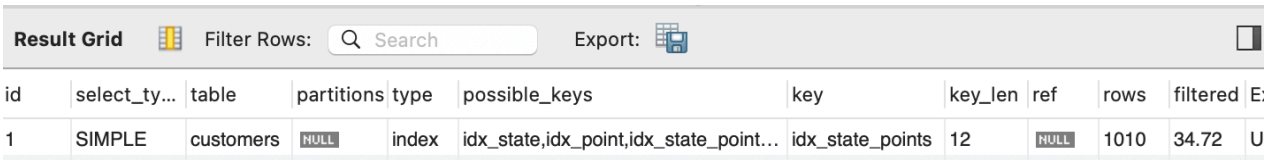
Result Grid Filter Rows: Search Export:

id	select_ty...	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered
1	SIMPLE	customers	NULL	range	idx_state,idx_state_points,idx_last...	idx_state_lastname	210	NULL	7	100.00

> 从上图结果可以看到，尽管选择基数大的列排在前面，但其效果不如后者(原因在于'='的约束力强于'LIKE')，所以最优顺序和所要查询的东西以及逻辑思维有关，上述原则不必奉为圭臬

```sql
-- 可以命令SQL使用指定索引
EXPLAIN SELECT customer_id
FROM customers
USE INDEX (idx_lastname_state)
WHERE state = 'CA' AND last_name LIKE 'A%';
```

```sql
-- 有时索引会失效
EXPLAIN SELECT customer_id FROM customers
WHERE state = 'CA' OR points > 1000;
```



Result Grid Filter Rows: Search Export:

id	select_ty...	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	E
1	SIMPLE	customers	NULL	index	idx_state,idx_point,idx_state_point...	idx_state_points	12	NULL	1010	34.72	U

> 做了全索引扫描(rows=数据总量)，它比表扫描快，因为它不涉及从磁盘读取每个记录。但这种时候需要考虑重新编写查询，已尽可能最好的方式利用索引，这里把OR语句拆成两个语句联合

```sql
CREATE INDEX idx_points ON customers(points);
-- 针对后一个查询建立一个单独的pionts索引会更高效
EXPLAIN
  SELECT customer_id FROM customers
  WHERE state = 'CA'
  UNION
  SELECT customer_id FROM customers
  WHERE points > 1000
-- UNION会自动去重
```

Result Grid Filter Rows: Search Export:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	customers	NULL	ref	idx_state	idx_state	8	const	112	100.00	Using
2	UNION	customers	NULL	range	idx_point,idx_points	idx_point	4	NULL	528	100.00	Using
3	UNION RESULT	<union1,2>	NULL	ALL	NULL	NULL	NULL	NULL	NULL	NULL	Using

> 一共扫描了640行(112+528)

```sql
-- 使用索引排序
EXPLAIN SELECT customer_id FROM customers
ORDER BY state;

-- 未使用索引排序
EXPLAIN SELECT customer_id FROM customers
ORDER BY first_name;
```

Result Grid Filter Rows: Search Export:

id	select_ty...	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	customers	NULL	index	NULL	idx_state	8	NULL	1010	100.00	Using index

Result Grid Filter Rows: Search Export:

id	select_ty...	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	customers	NULL	ALL	NULL	NULL	NULL	NULL	1010	100.00	Using filesort

> 上面的使用了索引：Using index
>
> 下面的使用外部排序：Using filesort（很费时的操作：1112...）

```sql
SHOW STATUS LIKE 'last_query_cost'
```

Result Grid Filter Rows:

Variable_name	Value
Last_query_cost	102.749000

Result Grid | Filter Rows:

Variable_name	Value
Last_query_cost	1112.749000

覆盖索引：Covering Indexes

> 一个包含所有满足查询需要的数据的索引

```
EXPLAIN SELECT * FROM customers
ORDER BY state;

EXPLAIN SELECT customer_id, state FROM customers
ORDER BY state;
```

Result Grid | Filter Rows: Search | Export:

id	select_ty...	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	customers	NULL	ALL	NULL	NULL	NULL	NULL	1010	100.00	Using filesort

Result Grid | Filter Rows: Search | Export:

id	select_ty...	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	customers	NULL	index	NULL	idx_state	8	NULL	1010	100.00	Using index

> 因为现有索引没有包含所有列，而包含了主键、state和points，所以前一个会使用外部排序，后一个会使用索引排序

维护索引

> 要多加注意"重复索引"和"多余索引"
>
> 重复索引：同一组列且顺序一致的索引 eg: (A, B, C) and (A, B, C)
>
> 多余索引：建立了有包含关系的索引 eg: (A, B) and (A)