SQL vs. Lambdas

John Kostaras

JCrete 25-29 August 2014

Agenda

* Database schema
  * Product - Supplier
  * SQLite
  * SQL
* Java Data structures
  * Product – Supplier
  * Java 8
  * Lambdas, Streams

SQL

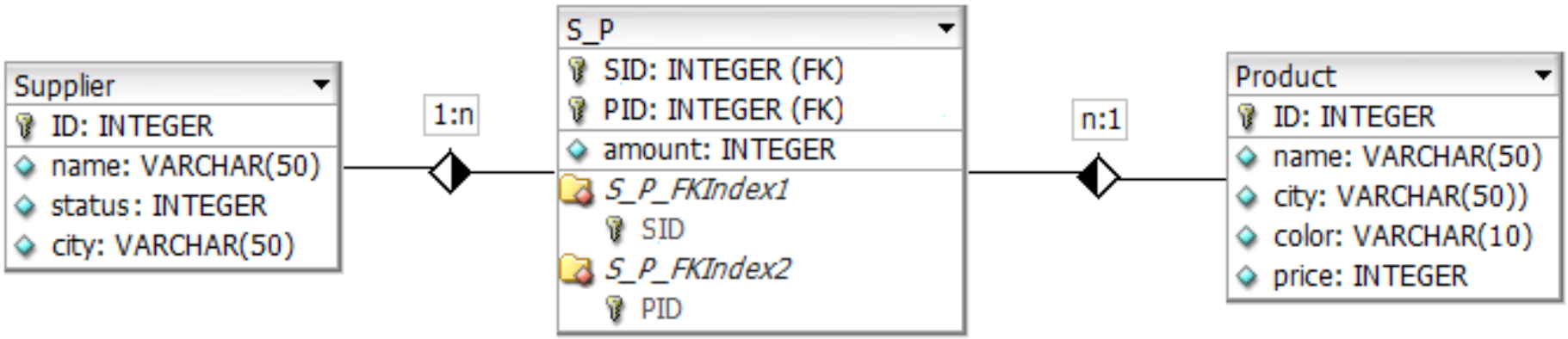
Data Definition Language (DDL)

CREATE (DROP) TABLE
CREATE (DROP) VIEW
CREATE (DROP) INDEX
ALTER TABLE

Data Manipulation Language (DML)

SELECT
INSERT
DELETE
UPDATE

Database schema



id	name	status	city
1	Sony	20	Hania
2	Apple	10	Heraklion
3	Apple	30	Hania
4	Dell	40	Rethymnon

sid	pid	amount
1	6	30
2	1	20
2	2	40
3	7	15
4	3	45
4	4	35
4	5	25

id	name	city	color	price
1	MacBook Pro 17"	Heraklion	Silver	2500
2	MacBook Pro 13"	Heraklion	Blue	1300
3	Alienware 14	Rethymnon	Black	1400
4	Alienware 17	Rethymnon	Red	1700
5	Alienware 18	Rethymnon	Green	1800
6	Vaio Tab 11"	Hania	Black	1200
7	MacBook Pro 13"	Hania	NULL	1250

Data Definition Language

```
CREATE TABLE Product (
    id    INTEGER PRIMARY KEY,
    name  TEXT,
    city  TEXT,
    color TEXT,
    price NUMERIC
);
CREATE TABLE Supplier (
    id     INTEGER PRIMARY KEY,
    name   TEXT,
    status NUMERIC,
    city   TEXT
);
```

Data Definition Language (cont.)

```
CREATE TABLE S_P (
    sid NUMERIC REFERENCES Supplier (id) ON DELETE CASCADE
                                         ON UPDATE CASCADE
                                         MATCH NONE,
    pid NUMERIC REFERENCES Product (id) ON DELETE CASCADE
                                        ON UPDATE CASCADE
                                        MATCH NONE,
    amount NUMERIC
);
```

Data Manipulation Language

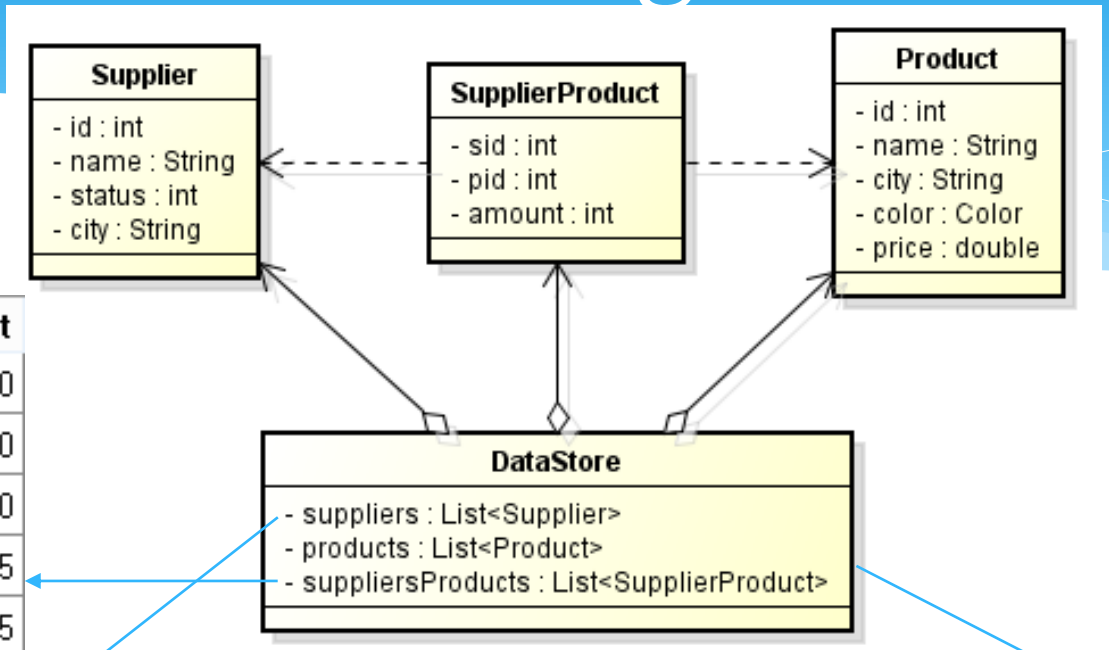
```
INSERT INTO Supplier (id, name, status, city)
VALUES (1, 'Sony', 20, 'Hania');

INSERT INTO Product (id, name, city, color, price)
VALUES (6, 'Vaio Tab 11"', 'Hania', 'Black', 1200);

INSERT INTO S_P (sid, pid, amount)
VALUES (1, 6, 30);
```

Stream API

Class diagram



sid	pid	amount
1	6	30
2	1	20
2	2	40
3	7	15
4	3	45
4	4	35
4	5	25

id	name	status	city
1	Sony	20	Hania
2	Apple	10	Heraklion
3	Apple	30	Hania
4	Dell	40	Rethymnon

id	name	city	color	price
1	MacBook Pro 17"	Heraklion	Silver	2500
2	MacBook Pro 13"	Heraklion	Blue	1300
3	Alienware 14	Rethymnon	Black	1400
4	Alienware 17	Rethymnon	Red	1700
5	Alienware 18	Rethymnon	Green	1800
6	Vaio Tab 11"	Hania	Black	1200
7	MacBook Pro 13"	Hania	*NULL*	1250

Simple Queries

Simple Selection
"Find all suppliers"

SQL

```
SELECT *
FROM Supplier;
-----------------------
1   Sony   20 Hania
2   Apple  10 Heraklion
3   Apple  30 Hania
4   Dell   40 Rethymnon
```

λ

```
suppliers.stream().
collect(toList());
---------------------------
Supplier{id=3, name='Apple',
status=30, city='Hania'}
Supplier{id=4, name='Dell',
status=40, city='Rethymnon'}
Supplier{id=2, name='Apple',
status=10, city='Heraklion'}
Supplier{id=1, name='Sony',
status=20, city='Hania'}
```

Simple Selection
"Find the names of all suppliers"

SQL

```
SELECT name
FROM Supplier;
-----------------------
Sony
Apple
Apple
Dell
```

λ

```
suppliers.stream().
map(Supplier::getName).
collect(toList());
---------------------------
Sony
Apple
Apple
Dell
```

Simple Selection
"Find the unique names of all suppliers"

SQL

```
SELECT DISTINCT name
FROM Supplier;
-----------------------
Sony
Apple
Dell
```

λ

```
suppliers.stream().
map(Supplier::getName).
collect(toSet());
---------------------------
Sony
Dell
Apple
```

Simple Selection
"Calculated values"

SQL

```
SELECT name, price*0.90
FROM Product;
-----------------------
1      2250.0
2      1170.0
3      1260.0
4      1530.0
5      1620.0
6      1080.0
7      1125.0
```

λ

```
products.stream().
map(p ->
{return p.getId() + " " +
 p.getPrice() * 0.9;}).
collect(toList());
---------------------------
1 2250.0
2 1170.0
3 1260.0
4 1530.0
5 1620.0
6 1080.0
7 1125.0
```

Selection with condition
"Find Apple supplier IDs from Hania"

SQL

```
SELECT id
FROM supplier
WHERE name = 'Apple'
AND city = 'Hania';
--------------------
3
```

λ

```
suppliers.stream().
filter(s ->
s.getName().equals("Apple")).
filter(s ->
s.getCity().equals("Hania")).
map(Supplier::getId).
collect(toList());
---------------------------
3
```

Selection between values
“Find products with 1000 ≤ price ≤ 2000”

SQL

```
SELECT name, price
FROM product
WHERE price
BETWEEN 1000 AND 2000;
--------------------
MacBook Pro 13 1300
Alienware 14 1400
Alienware 17 1700
Alienware 18 1800
Vaio Tab 11  1200
MacBook Pro 13 1250
```

λ

```
products.stream().
filter(p -> p.getPrice() >= 1000).
filter(p -> p.getPrice() <= 2000).
map(p -> {return p.getName()+" "+
p.getPrice();}).collect(toList());
--------------------------
MacBook Pro 13 1300.0
Alienware 14 1400.0
Alienware 17 1700.0
Alienware 18 1800.0
Vaio Tab 11 1200.0
MacBook Pro 13 1250.0
```

Selection of values
"Find products with price in (1800,2500)"

SQL

```
SELECT name, price
FROM product
WHERE price
IN (1800, 2500);
--------------------
MacBook Pro 17" 2500
Alienware 18 1800
```

λ

```
products.stream().
filter(p ->
p.getPrice() == 1800 ||
p.getPrice() == 2500).
map(p->{return p.getName() +
" " + p.getPrice();}).
collect(toList());
---------------------------
MacBook Pro 17 2500.0
Alienware 18 1800.0
```

Selection of null
"Find product IDs with null color"

SQL

```
SELECT id
FROM Product
WHERE color IS null;
--------------------
7
```

λ

```
products.stream().
filter(p ->
p.getColor() == null).
map(Product::getId).
collect(toList());
---------------------------
7
```

Selection with like
“Find products with name like ‘macbook’”

SQL

```
SELECT name, color
FROM product
WHERE name like
'%macbook%';
--------------------
MacBook Pro 17" Silver
MacBook Pro 13" Blue
MacBook Pro 13" null
```

λ

```
products.stream().
filter(p ->
p.getName().toLowerCase().
contains("macbook")).
map(p->{return p.getName() +
" " +  (p.getColor() == null ?
"null" : p.getColor());}).
collect(toList());
---------------------------
MacBook Pro 17
java.awt.Color[r=192,g=192,b=192]
MacBook Pro 13
java.awt.Color[r=0,g=0,b=255]
MacBook Pro 13 null
```

Sorted Selection

"Find products sorted descending by price"

SQL

```
SELECT name, price
FROM product
ORDER BY price DESC;
--------------------
MacBook Pro 17" 2500
Alienware 18 1800
Alienware 17 1700
Alienware 14 1400
MacBook Pro 13" 1300
MacBook Pro 13" 1250
Vaio Tab 11" 1200
```

λ

```
products.stream().
sorted(Comparator.comparing(
Product::getPrice).reversed()).
map(p->{return p.getName()+" "
+p.getPrice();}).
collect(toList());
--------------------------
MacBook Pro 17 2500.0
Alienware 18 1800.0
Alienware 17 1700.0
Alienware 14 1400.0
MacBook Pro 13 1300.0
MacBook Pro 13 1250.0
Vaio Tab 11 1200.0
```

Join Queries

Simple Join

“Find products and suppliers from same city”

SQL λ

```
SELECT p.name, s.name, s.city
FROM Product p, Supplier s
WHERE p.city = s.city;
-------------------------
MacBook Pro 17" Apple  Heraklion
MacBook Pro 13" Apple  Heraklion
Alienware 14    Dell   Rethymnon
Alienware 17    Dell   Rethymnon
Alienware 18    Dell   Rethymnon
Vaio Tab 11"    Apple  Hania
Vaio Tab 11"    Sony   Hania
MacBook Pro 13" Apple  Hania
MacBook Pro 13" Sony   Hania
```

Join

“Find products and suppliers’ cities ”

SQL λ

```
SELECT distinct pid, s.city
FROM S_P sp, Supplier s
WHERE sp.sid = s.id;
-------------------------
6       Hania
1       Heraklion
2       Heraklion
7       Hania
3       Rethymnon
4       Rethymnon
5       Rethymnon
```

Multiple Join

“Find total cost of Apple products in stock ”

SQL λ

```
SELECT pid, amount * price
FROM S_P sp, Supplier s,
Product p
WHERE sp.sid = s.id
AND sp.pid = p.id
AND s.name = 'Apple';
-------------------------
1       50000
2       52000
7       18750
```

Self Join

"Find all suppliers from same city"

SQL

```
SELECT sa.id, sb.id,
sa.city
FROM Supplier sa,
Supplier sb
WHERE sa.city = sb.city
AND sa.id < sb.id;
-------------------------
1     3     Hania
```

λ

```
suppliers.stream().
collect(groupingBy(
Supplier::getCity)).
entrySet().stream().filter(
e -> e.getValue().size()>1).
flatMap(e ->
e.getValue().stream()).
distinct().
map(Supplier::getId).
collect(toList());
-----------------------------
1, 3
```

Sub-queries

Simple Sub-query
“Find all suppliers that supply ‘Alienware 17’”

SQL λ

```
SELECT DISTINCT name
FROM Supplier s, S_P sp
WHERE s.id = sp.sid
AND sp.pid = 4;
-----------------------------
Dell
```

Combine Queries

UNION, INTERSECT

Union

"Find all suppliers from Hania or who supply > 40 products"

SQL λ

```
SELECT id
FROM Supplier s
WHERE s.city = 'Hania'
UNION
SELECT sid
FROM S_P sp
WHERE sp.amount > 40;
-----------------------------
1
3
4
```

Intersection

“Find all suppliers from Hania and who supply > 20 products”

SQL λ

```
SELECT id
FROM Supplier s
WHERE s.city = 'Hania'
INTERSECT
SELECT sid
FROM S_P sp
WHERE sp.amount > 20;
------------------------------
1
```

Functions

COUNT, SUM, AVG, MAX, MIN

Count

"Find how many suppliers"

SQL

```
SELECT COUNT (*)
FROM Supplier;
-----------------------------
4
```

λ

```
suppliers.stream().count();
-----------------------------
4
```

Sum

"Find total amount from Supplier 4"

SQL

```
SELECT SUM(amount)
FROM S_P
WHERE sid = 4;
-----------------
105
```

λ

```
suppliersProducts.stream().
filter(sp -> sp.getSupplierId() == 4).
map(SupplierProduct::getAmount).
reduce(0, (a, b) -> a + b);
-----------------------------------
105
```

Average

“Find products with price > avg(price)”

SQL

```
SELECT id, name, price
FROM Product
WHERE price > (SELECT
AVG(price) FROM Product);
-------------------------
1  MacBook Pro 17"  2500
4  Alienware 17     1700
5  Alienware 18     1800
```

λ

```
DoubleSummaryStatistics stats =
products().stream().
mapToDouble(Product::getPrice).
summaryStatistics();
products.stream().
filter(p -> p.getPrice() >
stats.getAverage()).
map(p -> {return p.getId()+" "+
p.getName()+" "+p.getPrice();).
collect(toList());
-------------------------------
1 MacBook Pro 17 2500.0
4 Alienware 17 1700.0
5 Alienware 18 1800.0
```

Group

“Find total amount grouped by supplier id”

SQL λ

```
SELECT sid,
sum(amount)
FROM S_P
GROUP BY sid;
```

```
suppliersProducts.stream().
collect(groupingBy(
SupplierProduct::getSupplierId,
summingInt(SupplierProduct::getAmount));
```

```
------------
1     30
2     60
3     15
4     105
```

```
-------------------------------
1 30
2 60
3 15
4 105
```

Having

"Find suppliers who supply more than 1 products"

SQL

```
SELECT sid, count(*)
FROM S_P
GROUP BY sid
HAVING COUNT(*) > 1;
--------------------
2
4
```

λ

```
suppliersProducts.stream().
collect(groupingBy(
SupplierProduct::getSupplierId)).
entrySet().stream().
filter(entry ->
entry.getValue().size() > 1).
map(Map.Entry::getKey)
.collect(toSet());
------------------------------
2
4
```

INSERT, UPDATE, DELETE

Insert

“Insert a new Supplier”

SQL λ

```
INSERT INTO Supplier
(id, name, status, city)
VALUES (5, 'Sony', 50,
'Rethymnon');
-------------------------
1 row(s) inserted
```

```
suppliers.add(new Supplier
("Sony", 50, "Rethymnon"));
```

Update

“Set status = 50 for Suppliers from Rethymnon”

SQL

```
UPDATE Supplier
SET status = 50
WHERE city ='Rethymnon’;
------------------------
2 row(s) updated
```

λ

```
suppliers.stream().
filter(s -> s.getCity().
equals("Rethymnon")).
forEach(s -> s.setStatus(50));
```

Delete

"Delete a Supplier"

SQL

```
DELETE FROM Supplier
WHERE id = 5;
-----------------------
1 row(s) deleted.
```

λ

```
suppliers.stream().
filter(s -> s.getId() == 5).
forEach(suppliers::remove);
// cascade changes
suppliersProducts.stream().
filter(sp ->
sp.getSupplierId() == 5).
forEach(
suppliersProducts::remove);
```

Delete

“Delete Suppliers from Rethymnon”

SQL

```
DELETE FROM Supplier
WHERE city =
'Rethymnon';
---------------------
2 row(s) deleted.


---------------------
```

λ

```
suppliers.
removeIf(s ->
s.getCity().
equals("Rethymnon"));
```

```
List<Suppliers> toDelete =
suppliers().stream().
filter(s ->
s.getCity().equals("Rethymnon")).
collect(toList());
suppliers.removeAll(toDelete);
// cascade changes
toDelete.forEach(s ->
suppliersProducts.stream().
filter(sp -> s.getId() ==
sp.getSupplierId()).
forEach(
suppliersProducts::remove));
```

Recap

* Compared SQL vs. the new Java 8 Stream API
* Stream API doesn't support joins

* Score: 26 - 20

References

* Naftalin, M. (2014), *Mastering Lambdas*, McGraw-Hill.
* Warburton, R. (2014), *Java 8 Lambdas*, O'Reilly.
* Subramaniam, V. (2014), *Functional Programing in Java*, Pragmatic.
* Darwin I. F. (2014), *Java Cookbook*, 3rd Ed., O' Reilly.
* Urma, R.-G. (2014), "Processing Data with Java SE 8 Streams – Part 1", Java Magazine, Issue 17, March-April, pp. 50-55.
* Urma, R.-G. (2014), "Processing Data with Java SE 8 Streams – Part 2", Java Magazine, Issue 18, May-June, pp. 49-53.
* Naftalin, M. (2013), "Navigating the Stream API", JCrete 2013.
* Κόλλια Γ. (1991), *Βάσεις Δεδομένων*, Τόμος Ι, Συμμετρία.

Questions