Common SQL Clauses and Their Equivalents in Java 8 Streams

Using such APIs, functional programming certainly feels like true declarative programming.

The most popular true declarative programming language is SQL. When you join two tables, you don’t tell the RDBMS how to implement that join. It may decide at its discretion whether a nested loop, merge join, hash join, or some other algorithm is the most suitable in the context of the complete query and of all the available meta information. This is extremely powerful because the performance assumptions that are valid for a simple join may no longer be valid for a complex one, where a different algorithm would outperform the original one. By this abstraction, you can just easily modify a query in 30 seconds, without worrying about low-level details like algorithms or performance.

When an API allows you to combine both (e.g. jOOQ and Streams (http://blog.jooq.org/2014/07/11/java-8-friday-more-functional-relational-transformation/)), you will get the best of both worlds – and those worlds aren’t too different.

In the following sections, we’ll compare common SQL constructs with their equivalent expressions written in Java 8 using Streams and jOOL (https://github.com/jOOQ/jOOL), in case the Stream API doesn’t offer enough functionality (http://blog.jooq.org/2014/09/10/when-the-java-8-streams-api-is-not-enough/).
Tuples

For the sake of this article, we’re going to assume that SQL rows / records have an equivalent representation in Java. For this, we’ll be using jOOQ’s Tuple type (http://www.jooq.org/products/jOOQ/javadoc/latest/org/jooq/lambda/tuple/Tuple.html), which is essentially:

```java
public class Tuple2<T1, T2> {
    public final T1 v1;
    public final T2 v2;
    public Tuple2(T1 v1, T2 v2) {
        this.v1 = v1;
        this.v2 = v2;
    }
}
```

... plus a lot of useful gimmicks like Tuple being Comparable, etc.

Note that we’re assuming the following imports in this and all subsequent examples.

```java
import static org.jooq.lambda.Seq.*;
import static org.jooq.lambda.tuple.Tuple.*;
import java.util.*;
import java.util.function.*;
import java.util.stream.*;
import org.jooq.lambda.*;
```

Much like SQL rows, a tuple is a “value-based” type (http://blog.codefx.org/java/value-based-classes/), meaning that it doesn’t really have an identity. Two tuples (1, 'A') and (1, 'A') can be considered exactly equivalent. Removing identity from the game makes SQL and functional programming with immutable data structures extremely elegant.

FROM = of(), stream(), etc.

In SQL, the FROM clause logically (but not syntactically) precedes all the other clauses. It is used to produce a set of tuples from at least one table, possibly multiple joined tables. A single-table FROM clause can be trivially mapped to Stream.of() (https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#of-T...-), for instance, or to any other method that simply produces a stream:
SELECT *
FROM (VALUES(1, 1),
         (2, 2)) t(v1, v2)

yielding

+------+------+
| v1   | v2   |
+------|------+
| 1    | 1    |
| 2    | 2    |
+------|------+

Java

Stream.of(
tuple(1, 1),
tuple(2, 2)).forEach(System.out::println);

yielding

(1, 1)
(2, 2)

CROSS JOIN = flatMap()

Selecting from multiple tables is already more interesting. The easiest way to combine two tables in SQL is by producing a cartesian product, either via a table list or using a CROSS JOIN. The following two are equivalent SQL statements:

SQL

-- Table list syntax
SELECT *
FROM (VALUES(1, 1), (2, 2)) t1(v1),
       (VALUES('A', 'B')) t2(v2)

-- CROSS JOIN syntax
SELECT *
FROM (VALUES(1, 1), (2, 2)) t1(v1)
     CROSS JOIN (VALUES('A', 'B')) t2(v2)

yielding

In a cross join (or cartesian product), every value from \( t_1 \) is combined with every value from \( t_2 \) producing \( \text{size}(t_1) \times \text{size}(t_2) \) rows in total.

### Java

In functional programming using Java 8’s Stream, the `Stream.flatMap()` method corresponds to SQL `CROSS JOIN` as can be seen in the following example:

```java
List<Integer> s1 = Stream.of(1, 2);
Supplier<Stream<String>> s2 = () -> Stream.of("A", "B");

s1.flatMap(v1 -> s2.get().map(v2 -> tuple(v1, v2))).forEach(System.out::println);
```

yielding

```
(1, A)
(1, B)
(2, A)
(2, B)
```

Note how we have to wrap the second stream in a `Supplier` because streams can be consumed only once (http://blog.jooq.org/2014/06/13/java-8-friday-10-subtle-mistakes-when-using-the-streams-api/), but the above algorithm is really implementing a nested loop, combining all elements of stream \( s_2 \) with each element from stream \( s_1 \). An alternative would be not to use streams but lists (which we will do in subsequent examples, for simplicity):

```java
List<Integer> s1 = Arrays.asList(1, 2);
List<String> s2 = Arrays.asList("A", "B");

s1.stream().flatMap(v1 -> s2.stream().map(v2 -> tuple(v1, v2))).forEach(System.out::println);
```

In fact, `CROSS JOIN` can be chained easily both in SQL and in Java:

SQL

```sql
-- Table list syntax
SELECT *
FROM (VALUES( 1 ), ( 2 )) t1(v1),
    (VALUES('A'), ('B')) t2(v2),
    (VALUES('X'), ('Y')) t3(v3)

-- CROSS JOIN syntax
SELECT *
FROM (VALUES( 1 ), ( 2 )) t1(v1)
CROSS JOIN (VALUES('A'), ('B')) t2(v2)
CROSS JOIN (VALUES('X'), ('Y')) t3(v3)
```

yielding

```
+--------+
| v1 | v2 | v3 |
+--------+
| 1 | A | X |
| 1 | A | Y |
| 1 | B | X |
| 1 | B | Y |
| 2 | A | X |
| 2 | A | Y |
| 2 | B | X |
| 2 | B | Y |
+--------+
```

Java

```java
List<Integer> s1 = Arrays.asList(1, 2);
List<String> s2 = Arrays.asList("A", "B");
List<String> s3 = Arrays.asList("X", "Y");

s1.stream()
    .flatMap(v1 -> s2.stream()
        .map(v2 -> tuple(v1, v2)))
    .flatMap(v12-> s3.stream()
        .map(v3 -> tuple(v12.v1, v12.v2, v3)))
    .forEach(System.out::println);
```

yielding
Note how we explicitly unnested the tuples from the first CROSS JOIN operation to form “flat” tuples in the second operation. This is optional, of course.

Java with jOOλ’s crossJoin()

Us jOOQ (http://www.jooq.org) developers, we’re a very SQL-oriented people, so it is only natural to have added a crossJoin() method for the above use-case. So our triple-cross join can be written like this:

```
Seq<Integer> s1 = Seq.of(1, 2);
Seq<String> s2 = Seq.of("A", "B");
Seq<String> s3 = Seq.of("X", "Y");

s1.crossJoin(s2)
    .crossJoin(s3)
    .forEach(System.out::println);
```

yielding

```
((1, A), X)
((1, A), Y)
((1, B), X)
((1, B), Y)
((2, A), X)
((2, A), Y)
((2, B), X)
((2, B), Y)
```

In this case, we didn’t unnest the tuple produced in the first cross join. From a merely relational perspective, this doesn’t matter either. Nested tuples are the same thing as flat tuples. In SQL, we just don’t see the nesting. Of course, we could still unnest as well by adding a single additional mapping:
INNER JOIN = `flatMap()` with `filter()`

The SQL `INNER JOIN` is essentially just syntactic sugar for a SQL `CROSS JOIN` with a predicate that reduces the tuple set after cross-joining. In SQL, the following two ways of inner joining are equivalent:

**SQL**

```
1    -- Table list syntax
2    SELECT *
3    FROM (VALUES(1), (2)) t1(v1),
4    (VALUES(1), (3)) t2(v2)
5    WHERE t1.v1 = t2.v2

6    -- INNER JOIN syntax
7    SELECT *
8    FROM (VALUES(1), (2)) t1(v1)
9    INNER JOIN (VALUES(1), (3)) t2(v2)
10   ON t1.v1 = t2.v2
```

yielding

```
+-----+-----+
| v1  | v2  |
+-----+-----+
| 1   | 1   |
+-----+-----+
```
(note that the keyword INNER is optional).

So, the values 2 from t1 and the values 3 from t2 are “thrown away”, as they produce any rows for which the join predicate yields true.

The same can be expressed easily, yet more verbosely in Java

Java (inefficient solution!)

```java
List<Integer> s1 = Arrays.asList(1, 2);
List<Integer> s2 = Arrays.asList(1, 3);

s1.stream()
    .flatMap(v1 -> s2.stream()
             .map(v2 -> tuple(v1, v2)))
    .filter(t -> Objects.equals(t.v1, t.v2))
    .forEach(System.out::println);
```

The above correctly yields

```
(1, 1)
```

But beware that you’re attaining this result after producing a cartesian product, the nightmare of every DBA! As mentioned at the beginning of this article, unlike in declarative programming, in functional programming you instruct your program to do exactly the order of operations that you specify. In other words:

*In functional programming, you define the exact “execution plan” of your query.*

*In declarative programming, an optimiser may reorganise your “program”*

There is no optimiser to transform the above into the much more efficient:

Java (more efficient)

```java
List<Integer> s1 = Arrays.asList(1, 2);
List<Integer> s2 = Arrays.asList(1, 3);

s1.stream()
    .flatMap(v1 -> s2.stream()
             .filter(v2 -> Objects.equals(v1, v2))
             .map(v2 -> tuple(v1, v2)))
    .forEach(System.out::println);
```

The above also yields

```
(1, 1)
```
Notice, how the join predicate has moved from the “outer” stream into the “inner” stream, that is produced in the function passed to `flatMap()`.

Java (optimal)

As mentioned previously, functional programming doesn’t necessarily allow you to rewrite algorithms depending on knowledge of the actual data. The above presented implementation for joins always implement nested loop joins going from the first stream to the second. If you join more than two streams, or if the second stream is very large, this approach can be terribly inefficient. A sophisticated RDBMS would never blindly apply nested loop joins like that, but consider constraints, indexes, and histograms on actual data.

Going deeper into that topic would be out of scope for this article, though.

Java with jOOλ’s innerJoin()

Again, inspired by our work on jOOQ (http://www.jooq.org) we’ve also added an `innerJoin()` (http://www.jooq.org/products/jOOλ/javadoc/latest/org/jooq/lambda/Seq.html#innerJoin-java.util.stream.Stream-java.util.function.Predicate-) convenience method for the above use-case:

```
Seq<Integer> s1 = Seq.of(1, 2);
Seq<Integer> s2 = Seq.of(1, 3);

s1.innerJoin(s2, (t, u) -> Objects.equals(t, u))
.forEach(System.out::println);
```

yielding

```
(1, 1)
```

... because after all, when joining two streams, the only really interesting operation is the join Predicate. All else (flatmapping, etc.) is just boilerplate.

LEFT OUTER JOIN = `flatMap()` with filter() and a “default”

SQL’s `OUTER JOIN` works like `INNER JOIN`, except that additional “default” rows are produced in case the `JOIN` predicate yields `false` for a pair of tuples. In terms of set theory / relational algebra (https://en.wikipedia.org/wiki/Relational_algebra#Outer_joins), this can be expressed as such:

```
(R ⋈ S) ∪ ((R − π_{r1,r2,...,rn}(R ⋈ S)) × {(ω, ..., ω)})
```

(https://en.wikipedia.org/wiki/Relational_algebra#Outer_joins)

Or in a SQL-esque dialect:
R LEFT OUTER JOIN S ::= 

R INNER JOIN S
UNION ( 
  (R EXCEPT (SELECT R.* FROM R INNER JOIN S))
  CROSS JOIN
  (null, null, ..., null)
)

This simply means that when left outer joining S to R, there will be at least one row in the result for each row in R, with possibly an empty value for S.

Inversely, when right outer joining S to R, there will be at least one row in the result for each row in S, with possibly an empty value for R.

And finally, when full outer joining S to R, there will be at least one row in the result for each row in R with possibly an empty value for S AND for each row in S with possibly an empty value for R.

Let us look at LEFT OUTER JOIN, which is used most often in SQL.

SQL

```sql
-- Table list, Oracle syntax (don't use this!)
SELECT *
FROM (SELECT 1 v1 FROM DUAL
      UNION ALL
      SELECT 2 v1 FROM DUAL) t1,
     (SELECT 1 v2 FROM DUAL
      UNION ALL
      SELECT 3 v2 FROM DUAL) t2
WHERE t1.v1 = t2.v2 (+)

-- OUTER JOIN syntax
SELECT *
FROM (VALUES(1),(2)) t1(v1)
LEFT OUTER JOIN (VALUES(1),(3)) t2(v2)
ON t1.v1 = t2.v2
```

yielding

```
+----------+
| v1 | v2 |
+----------+
| 1 | 1 |
| 2 | null |
+----------+
```

(note that the keyword OUTER is optional).
Unfortunately, the JDK’s Stream API doesn’t provide us with an easy way to produce “at least” one value from a stream, in case the stream is empty. We could be writing a utility function as explained by Stuart Marks on Stack Overflow (http://stackoverflow.com/a/26659413/521799):

```java
static <T> Stream<T> defaultIfEmpty(
    Stream<T> stream, Supplier<T> supplier) {
  Iterator<T> iterator = stream.iterator();
  if (iterator.hasNext()) {
    return StreamSupport.stream(
      Spliterators.spliteratorUnknownSize(
        iterator, 0
      ), false);
  } else {
    return Stream.of(supplier.get());
  }
}
```

Or, we just use jOOλ’s `Seq.onEmpty()` (http://www.jooq.org/products/jOOλ/javadoc/latest/org/jooq/lambda/Seq.html#onEmpty-T-)

```java
List<Integer> s1 = Arrays.asList(1, 2);
List<Integer> s2 = Arrays.asList(1, 3);

seq(s1)
    .flatMap(v1 -> seq(s2)
      .filter(v2 -> Objects.equals(v1, v2))
      .onEmpty(null)
    .map(v2 -> tuple(v1, v2))
  .forEach(System.out::println);
```

(notice, we’re putting `null` in a stream. This might not always be a good idea. We’ll follow up with that in a future blog post)

The above also yields

```
(1, 1)
(2, null)
```

How to read the implicit left outer join?

- We’ll take each value `v1` from the left stream `s1`
- For each such value `v1`, we flatmap the right stream `s2` to produce a tuple `(v1, v2)` (a cartesian product, cross join)
- We’ll apply the join predicate for each such tuple `(v1, v2)`
- If the join predicate leaves no tuples for any value `v2`, we’ll generate a single tuple containing the value of the left stream `v1` and `null`

Java with jOOλ

For convenience, jOOλ also supports `leftOuterJoin()` (http://www.jooq.org/products/jOOλ/javadoc/latest/org/jooq/lambda/Seq.html#leftOuterJoin) which works as described above:

```java
Seq<Integer> s1 = Seq.of(1, 2);
Seq<Integer> s2 = Seq.of(1, 3);
s1.leftOuterJoin(s2, (t, u) -> Objects.equals(t, u))
    .forEach(System.out::println);
```

yielding

```
(1, 1)
(2, null)
```

**RIGHT OUTER JOIN = inverse LEFT OUTER JOIN**

Trivially, a `RIGHT OUTER JOIN` is just the inverse of the previous `LEFT OUTER JOIN`. The jOOλ implementation of `rightOuterJoin()` (http://www.jooq.org/products/jOOλ/javadoc/latest/org/jooq/lambda/Seq.html#rightOuterJoin) looks like this:

```java
default <U> Seq<Tuple2<T, U>> rightOuterJoin(
    Stream<U> other, BiPredicate<T, U> predicate) {
    return seq(other)
        .leftOuterJoin(this, (u, t) -> predicate.test(t, u))
        .map(t -> tuple(t.v2, t.v1));
}
```

As you can see, the `RIGHT OUTER JOIN` inverses the results of a `LEFT OUTER JOIN`, that's it. For example:

```java
Seq<Integer> s1 = Seq.of(1, 2);
Seq<Integer> s2 = Seq.of(1, 3);
s1.rightOuterJoin(s2, (t, u) -> Objects.equals(t, u))
    .forEach(System.out::println);
```

yielding

```
(1, 1)
(null, 3)
```
WHERE = filter()

The most straight-forward mapping is probably SQL’s WHERE clause having an exact equivalent in the Stream API: `Stream.filter()` (https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#filter-java.util.function.Predicate-).

**SQL**

```
1          SELECT *
2          FROM (VALUES(1), (2), (3)) t(v)
3          WHERE v % 2 = 0
```

yielding

```
+----+
| v  |
+----+
| 2  |
+----+
```

**Java**

```
1   Stream<Integer> s = Stream.of(1, 2, 3);
2   s.filter(v -> v % 2 == 0)
3         .forEach(System.out::println);
```

yielding

```
2
```

The interesting thing with `filter()` and the Stream API in general is that the operation can apply at any place in the call chain, unlike the WHERE clause, which is limited to be placed right after the FROM clause - even if SQL’s JOIN .. ON or HAVING clauses are semantically similar.

GROUP BY = collect()  

The least straight-forward mapping is GROUP BY vs. `Stream.collect()` (https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#collect-java.util.stream.Collector-).
First off, SQL’s GROUP BY may be a bit tricky to fully understand (http://blog.jooq.org/2014/12/04/do-you-really-understand-sqls-group-by-and-having-clauses/). It is really part of the FROM clause, transforming the set of tuples produced by FROM .. JOIN .. WHERE into groups of tuples, where each group has an associated set of aggregatable tuples, which can be aggregated in the HAVING, SELECT, and ORDER BY clauses. Things get even more interesting when you use OLAP features like GROUPING SETS (http://blog.jooq.org/2011/11/26/group-by-rollup-cube/), which allow for duplicating tuples according to several grouping combinations.

In most SQL implementations that don’t support ARRAY or MULTISET, the aggregatable tuples are not available as such (i.e. as nested collections) in the SELECT. Here, the Stream API’s feature set excels. On the other hand, the Stream API can group values only as a terminal operation (https://docs.oracle.com/javase/tutorial/collections/streams/), where in SQL, GROUP BY is applied purely declaratively (and thus, lazily). The execution planner may choose not to execute the GROUP BY at all if it is not needed. For instance:

```java
SELECT * 
FROM some_table 
WHERE EXISTS ( 
    SELECT x, sum(y) 
    FROM other_table 
    GROUP BY x 
)
```

The above query is semantically equivalent to

```java
SELECT * 
FROM some_table 
WHERE EXISTS ( 
    SELECT 1 
    FROM other_table 
)
```

The grouping in the subquery was unnecessary. Someone may have copy-pasted that subquery in there from somewhere else, or refactored the query as a whole. In Java, using the Stream API, each operation is always executed.

For the sake of simplicity, we’ll stick to the most simple examples here

### Aggregation without GROUP BY

A special case is when we do not specify any GROUP BY clause. In that case, we can specify aggregations on all columns of the FROM clause, producing always exactly one record. For instance:

**SQL**

```sql
SELECT sum(v)
FROM (VALUES(1), (2), (3)) t(v)
```
yielding

<table>
<thead>
<tr>
<th></th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

Java

```java
Stream<Integer> s = Stream.of(1, 2, 3);

int sum = s.collect(Collectors.summingInt(i -> i));

System.out.println(sum);
```

yielding

6

### Aggregation with GROUP BY

A more common case of aggregation in SQL is to specify an explicit **GROUP BY** clause as explained before. For instance, we may want to group by even and odd numbers:

**SQL**

```sql
SELECT v % 2, count(v), sum(v)
FROM (VALUES(1), (2), (3)) t(v)
GROUP BY v % 2
```

yielding

```
+-----------------+-------+-------+
| v % 2 | count | sum   |
|-------+-------+-------|
| 0     | 1     | 2     |
| 1     | 2     | 4     |
```

**Java**

```java
Stream<Integer> s = Stream.of(1, 2, 3);

int sum = s.collect(Collectors.summingInt(i -> i));

System.out.println(sum);
```
For this simple grouping / collection use-case, luckily, the JDK offers a utility method called
`Collectors.groupingBy()`
(https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html#groupingBy-java.util.function.Function-), which produces a collector that generates a `Map<K, List<V>>` type like this:

```java
Stream<Integer> s = Stream.of(1, 2, 3);
Map<Integer, List<Integer>> map = s.collect(
    Collectors.groupingBy(v -> v % 2)
);
System.out.println(map);
```

yielding

```
{0=[2], 1=[1, 3]}
```

This certainly takes care of the grouping. Now we want to produce aggregations for each group. The slightly awkward JDK way to do this would be:

```java
Stream<Integer> s = Stream.of(1, 2, 3);
Map<Integer, IntSummaryStatistics> map = s.collect(
    Collectors.groupingBy(
        v -> v % 2,
        Collectors.summarizingInt(i -> i)
    )
);
System.out.println(map);
```

we’ll now get:

```
{0=IntSummaryStatistics{count=1, sum=2, min=2, average=2.000000, max=2},
 1=IntSummaryStatistics{count=2, sum=4, min=1, average=2.000000, max=3}}
```

As you can see, the `count()` and `sum()` values have been calculated somewhere along the lines of the above.

**More sophisticated GROUP BY**

When doing multiple aggregations with Java 8’s Stream API, you will quickly be forced to wrestle low-level API implementing complicated collectors and accumulators yourself. This is tedious and unnecessary. Consider the following SQL statement:
SQL

```sql
CREATE TABLE t (  
  w INT,
  x INT,
  y INT,
  z INT
);

SELECT  
  z, w,
  MIN(x), MAX(x), AVG(x),
  MIN(y), MAX(y), AVG(y)
FROM t
GROUP BY z, w;
```

In one go, we want to:

- Group by several values
- Aggregate from several values

Java

In a previous article, we’ve explained in detail how this can be achieved using convenience API from jOOλ via (http://blog.jooq.org/2015/01/23/how-to-translate-sql-group-by-and-aggregations-to-java-8/) `Seq.groupBy()` (http://www.jooq.org/products/jOOλ/javadoc/latest/org/jooq/lambda/Seq.html#groupBy-java.util.stream.Collector-)

```java
class A {  
  final int w;
  final int x;
  final int y;
  final int z;

  A(int w, int x, int y, int z) {  
    this.w = w;
    this.x = x;
    this.y = y;
    this.z = z;
  }
}

Map<
  Tuple2<Integer, Integer>,
  Tuple2<IntSummaryStatistics, IntSummaryStatistics>>
> map =
Seq.of(  
  new A(1, 1, 1, 1),
  new A(1, 2, 3, 1),
  new A(9, 8, 6, 4),
  new A(9, 9, 7, 4),
  new A(2, 3, 4, 5),
  new A(2, 4, 4, 5),
```
new A(2, 5, 5))

// Seq.groupBy() is just short for
// Stream.collect(Collectors.groupingBy( ... ))
.groupBy(
a -> tuple(a.z, a.w),
   // ... because once you have tuples,
   // why not add tuple-collectors?
   Tuple.collectors(
      Collectors.summarizingInt(a -> a.x),
      Collectors.summarizingInt(a -> a.y)
   )
);

System.out.println(map);

The above yields

{(1, 1)=(IntSummaryStatistics{count=2, sum=3, min=1, average=1.500000, max=2},
   IntSummaryStatistics{count=2, sum=4, min=1, average=2.000000, max=3}),
(4, 9)=(IntSummaryStatistics{count=2, sum=17, min=8, average=8.500000, max=9},
   IntSummaryStatistics{count=2, sum=13, min=6, average=6.500000, max=7})
(5, 2)=(IntSummaryStatistics{count=3, sum=12, min=3, average=4.000000, max=5},
   IntSummaryStatistics{count=3, sum=13, min=4, average=4.333333, max=5})

For more details, read the full article here (http://blog.jooq.org/2015/01/23/how-to-translate-sql-group-by-and-aggregations-to-java-8/).

Notice how using `Stream.collect()` (https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#collect-java.util.stream.Collector-) or `Seq.groupBy()` (http://www.jooq.org/products/jOQ/java/util/stream/Seq.html#groupBy-java.util.stream.Collector-) already makes for an implicit SELECT clause, which we are no longer needed to obtain via `map()` (see below).

**HAVING = filter(), again**

As mentioned before, there aren’t really different ways of applying predicates with the Stream API, there is only `Stream.filter()` (https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#filter-java.util.function.Predicate-). In SQL, HAVING is a “special” predicate clause that is syntactically put after the GROUP BY clause. For instance:

```
SQL
```
Unfortunately, as we have seen before, `collect()` is a terminal operation in the `Stream` API, which means that it eagerly produces a `Map`, instead of transforming the `Stream<T>` into a `Stream<K, Stream<V>>`, which would compose much better in complex `Stream`. This means that any operation that we’d like to implement right after collecting will have to be implemented on a new `stream` produced from the output `Map`:

```java
Stream<Integer> s = Stream.of(1, 2, 3);

s.collect(Collectors.groupingBy(
    v -> v % 2,
    Collectors.summarizingInt(i -> i)
)).entrySet().stream().filter(e -> e.getValue().getCount() > 1)
    .forEach(System.out::println);
```

As you can see, the type transformation that is applied is:

- `Map<Integer, IntSummaryStatistics>`
- `Set<Entry<Integer, IntSummaryStatistics>>`
- `Stream<Entry<Integer, IntSummaryStatistics>>`

```sql
SELECT = map()
```
The SELECT clause in SQL is nothing more than a tuple transformation function that takes the cartesian product of tuples produced by the FROM clause and transforms it into a new tuple expression, which is fed either to the client, or to some higher-level query if this is a nested SELECT. An illustration:

FROM output

```
+-----------------+-----------------+-----------------+-----------------+-----------------+
+-----------------+-----------------+-----------------+-----------------+-----------------+
|   1   |   A   |   a   |   1   |   X   |
|   1   |   B   |   b   |   1   |   Y   |
|   2   |   C   |   c   |   2   |   X   |
|   2   |   D   |   d   |   2   |   Y   |
+-----------------+-----------------+-----------------+-----------------+-----------------+
```

Applying SELECT

```
SELECT t1.a, t1.c, t1.b || t1.d
```

```
+-----------------+-----------------+-----------------+-----------------+-----------------+
+-----------------+-----------------+-----------------+-----------------+
|   1   |   a   |   AX  |
|   1   |   b   |   BY  |
|   2   |   c   |   CX  |
|   2   |   d   |   DY  |
+-----------------+-----------------+-----------------+-----------------+
```

Using Java 8 Streams, SELECT can be achieved very simply by using `Stream.map()` (https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#map-java.util.function.Function-), as we’ve already seen in previous examples, where we unnested tuples using `map()`. The following examples are functionally equivalent:

**SQL**

```sql
1 SELECT t.v1 * 3, t.v2 + 5
2 FROM (3     VALUES(1, 1),
4           (2, 2)
5       ) t(v1, v2)
```

yielding
DISTINCT = distinct()

The DISTINCT keyword that can be supplied with the SELECT clause simply removes duplicate tuples right after they have been produced by the SELECT clause. An illustration:

FROM output

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>a</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>b</td>
<td>1</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>c</td>
<td>2</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
<td>d</td>
<td>2</td>
<td>Y</td>
</tr>
</tbody>
</table>

Applying SELECT DISTINCT

```
Stream.of(
  tuple(1, 1),
  tuple(2, 2)
).map(t -> tuple(t.v1 * 3, t.v2 + 5))
.forEach(System.out::println);
```
SELECT DISTINCT t1.a

+------+
| T1.A  |
+------+
| 1     |
| 2     |
+------+

Using Java 8 Streams, SELECT  DISTINCT can be achieved very simply by using Stream.distinct() (https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#distinct--) right after Stream.map() (https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#map-java.util.function.Function-). The following examples are functionally equivalent:

**SQL**

```
1 | SELECT DISTINCT t.v1 * 3, t.v2 + 5
2  | FROM ( VALUES(1, 1),
3  |     (2, 2),
4  |     (2, 2) ) t(v1, v2)
```

yielding

+------+
| c1    |
| c2    |
+------+
| 3     |
| 6     |
| 6     |
| 7     |
+------+

**Java**

```
1 | Stream.of(
2  |   tuple(1, 1),
3  |   tuple(2, 2),
4  |   tuple(2, 2)
5  | ).map(t -> tuple(t.v1 * 3, t.v2 + 5))
6  | .distinct()
7  | .forEach(System.out::println);
```

yielding

```
(3, 6)
(6, 7)
```
UNION ALL = concat()

Set operations are powerful both in SQL and using the Stream API. The UNION ALL operation maps to Stream.concat()
(https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#concat-java.util.stream.Stream-java.util.stream.Stream-), as can be seen below:

**SQL**

```
SELECT *
FROM (VALUES(1), (2)) t(v)
UNION ALL
SELECT *
FROM (VALUES(1), (3)) t(v)
```

yielding
```
+---+
| v |
+---+
| 1 |
| 2 |
| 1 |
| 3 |
```

**Java**

```
Stream<Integer> s1 = Stream.of(1, 2);
Stream<Integer> s2 = Stream.of(1, 3);
Stream.concat(s1, s2)
    .forEach(System.out::println);
```

yielding
```
1
2
1
3
```

**Java (using jOOλ)**

Unfortunately, `concat()` exists in `Stream` only as a static method, while `Seq.concat()` (http://www.jooq.org/products/jOOλ/javadoc/latest/org/jooq/lambda/Seq.html#concat-java.util.stream.Stream-) also exists on instances when working with jOOλ.

```java
1  Seq<Integer> s1 = Seq.of(1, 2);
2  Seq<Integer> s2 = Seq.of(1, 3);
3  s1.concat(s2)
4     .forEach(System.out::println);
```

**UNION = concat() and distinct()**

In SQL, **UNION** is defined to remove duplicates after concatenating the two sets via **UNION ALL**. The following two statements are equivalent:

```sql
SELECT * FROM t
UNION
SELECT * FROM u;

-- equivalent

SELECT DISTINCT *
FROM (  
    SELECT * FROM t
    UNION ALL
    SELECT * FROM u
  )
;
```

Let’s put this in action:

**SQL**

```sql
1 SELECT *
2 FROM (VALUES(1), (2)) t(v)
3 UNION
4 SELECT *
5 FROM (VALUES(1), (3)) t(v)
```

yielding
Java

```java
Stream<Integer> s1 = Stream.of(1, 2);
Stream<Integer> s2 = Stream.of(1, 3);
Stream.concat(s1, s2)
    .distinct()
    .forEach(System.out::println);
```

ORDER BY = sorted()

The ORDER BY mapping is trivial

SQL

```sql
SELECT *
FROM (VALUES(1), (4), (3)) t(v)
ORDER BY v
```

yielding

```
+----+
| v  |
+----+
| 1  |
| 2  |
| 3  |
+----+
```

Java

```java
Stream<Integer> s = Stream.of(1, 4, 3);
s.sorted()
    .forEach(System.out::println);
```

yielding

```
+----+
| v  |
+----+
| 1  |
| 3  |
| 4  |
+----+
```
LIMIT = limit()

The LIMIT mapping is even more trivial

SQL

```sql
SELECT *  
FROM (VALUES(1), (4), (3)) t(v)  
LIMIT 2
```

yielding

```
+----+
<table>
<thead>
<tr>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>-----</td>
</tr>
</tbody>
</table>
```

Java

```java
Stream<Integer> s = Stream.of(1, 4, 3);

s.limit(2)
.forEach(System.out::println);
```

yielding

```
1
4
```

OFFSET = skip()

The OFFSET mapping is trivial as well
Common SQL Clauses and Their Equivalents in Java 8 Streams | Java, SQL and jOOQ.

SQL

```
1 | SELECT *
2 | FROM (VALUES(1), (4), (3)) t(v)
3 | OFFSET 1
```

yielding

```
+----+
| v   |
+----+
| 4   |
| 3   |
+----+
```

Java

```
1 | Stream<Integer> s = Stream.of(1, 4, 3);
2 | s.skip(1)
3 | .forEach(System.out::println);
```

yielding

```
4
3
```

Conclusion

In the above article, we’ve seen pretty much all the useful SQL SELECT query clauses and how they can be mapped to the Java 8 Stream API, or to jOOQ’s Seq API, in case Stream doesn’t offer sufficient functionality.

The article shows that SQL’s declarative world is not that much different from Java 8’s functional world. SQL clauses can compose ad-hoc queries just as well as Stream methods can be used to compose functional transformation pipelines. But there is a fundamental difference.

While SQL is truly declarative, functional programming is still very instructive. The Stream API does not make optimisation decisions based on constraints, indexes, histograms and other meta information about the data that you’re transforming. Using the Stream API is like using all possible optimisation hints in SQL to force the SQL engine to choose one particular execution plan over another. However, while SQL is a higher level algorithm abstraction, the Stream API may allow you to implement more customisable algorithms.
Tags: java, jOOλ, sql, streams

3 responses to “Common SQL Clauses and Their Equivalents in Java 8 Streams”

stellingsimon says: August 14, 2015 at 19:46

0

0

i
Rate This

Thanks for this awesome read. I’ll show this to everyone of my friends who ever said “SQL? They still use that?” (you know who you are!)

REPLY
lukaseder says: August 14, 2015 at 19:51

0

0

i
Rate This

Thanks for your nice words.

“SQL? They still use that?”

That’s the kind of words that all adolescents use when they try to emancipate from their parents / mentors. When they mature, a couple of years (or decades, at times) later, they will realise “heck, the parents hadn’t been that wrong”.

I tend to quote Isaac Newton:

_If I have seen further, it is by standing on the shoulders of giants._

SQL is built on one of the few unchallenged and solid foundations of our industry. The syntax may be weird and feel COBOL-esque at times, but the ideas are timeless. If anything will ever supersede SQL (it won’t), it will be SQL with a better, even more powerful syntax.

REPLY
lukaseder says: August 14, 2015 at 19:52
Rate This

(I have to ask)

(you know who you are!)

What did you mean by that?

REPLY

Java, SQL and jOOQ.