SQL to Stream with S2S: An Automatic Benchmark Generator for the Java Stream API

Filippo Schiavio
filippo.schiavio@usi.ch
Università della Svizzera italiana
Lugano, Switzerland

Andrea Rosà
andrea.rosa@usi.ch
Università della Svizzera italiana
Lugano, Switzerland

Walter Binder
walter.binder@usi.ch
Università della Svizzera italiana
Lugano, Switzerland

Abstract

The Java Stream API was introduced in Java 8, allowing developers to express computations in a functional style by defining a pipeline of data-processing operations. Despite the growing importance of this API, there is a lack of benchmarks specifically targeting stream-based applications. Instead of designing and implementing new ad-hoc workloads for the Java Stream API, we propose to automatically translate existing data-processing workloads. To this end, we present S2S, an automatic benchmark generator for the Java Stream API. S2S is a SQL query compiler that converts existing workloads designed for relational databases to stream-based code. We use S2S to generate BSS, the first benchmark suite for the Java Stream API.

CCS Concepts: • **Software and its engineering** → **Source code generation**.

Keywords: SQL Query Compilation, Code Generation, Java Stream API, Automatic Benchmark Generation

ACM Reference Format:
Filippo Schiavio, Andrea Rosà, and Walter Binder. 2022. SQL to Stream with S2S: An Automatic Benchmark Generator for the Java Stream API. In *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '22), December 06–07, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3564719.3568699

1 Introduction

The Java Stream API [26] was introduced in Java 8 to allow developers to express a pipeline of data-processing operations on collections of elements called *streams*. The API eases the implementation of data transformations expressed with a functional, concise and declarative style while keeping the advantages of Java as an object-oriented language with an imperative paradigm. Moreover, the API allows one to easily and automatically parallelize stream processing by calling just a single operation. For these reasons, the Java Stream API is increasingly used by Java applications [30].

Despite the growing importance of the Java Stream API, after ten years since its release, there is a lack of benchmarks specifically targeting stream-based applications. To the best of our knowledge, the only established benchmark suite including workloads that use the Java Stream API is Renaissance [27], which however contains only three stream-based applications that are not representative of a realistic use of streams[1] and are therefore little suitable for evaluation needs. Other well-known benchmark suites for Java, such as DaCapo [4] or SPECjvm2008 [7], were released before the availability of the Java Stream API, while others (such as SPECjbb2015 [6]) target runtimes (Java 7) not supporting streams. Such a lack of benchmarks makes it very hard for researchers and language developers to analyze and improve the performance of the Java Stream API. Indeed, many techniques [1, 3, 15, 17, 20, 29] have been proposed to improve the performance of the Java Stream API, but all of them have been evaluated on a simple set of micro-benchmarks that have been manually implemented or crawled by the authors, which is a non-effortless process. We believe that researchers and Java developers working on optimizing the Stream API would benefit from benchmark suite composed of workloads fully dedicated to the Java Stream API.

Our work aims at mitigating the long-standing absence of benchmarks specific to the Java Stream API. Instead of designing a new benchmark suite from scratch—which is known to be challenging and extremely time consuming [40]—we resort to a different approach. The idea behind of our work is that the relational database community has already designed many data-processing workloads. Examples include TPC-H [36], TPC-DS [35], TPC-C [34], TPCx-BB [37], Start Schema Benchmark (SSB) [23], CH-benCHmark [5] and Join Order Benchmark (JOB) [18], all of which are composed of a multitude of queries expressed in the SQL language. Since

GPCE '22, December 06–07, 2022, Auckland, New Zealand
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9920-3/22/12…$15.00
https://doi.org/10.1145/3564719.3568699

[1]The Renaissance benchmarks use the Java Stream API to simulate games (scrabble) or solve combinatorial problems (mnemonics and par-mnemonics), which do not reflect a realistic use of streams.

the Java Stream API has been specifically designed to execute data-processing operations, such workloads seem a natural fit to be used as benchmarks for the API, if they are translated from SQL to Java code making use of streams.

To this end, we present S2S, the first automatic benchmark generator for the Java Stream API (Section 2). [2] At its core, S2S is a SQL query compiler that produces Java source code that makes use of the Java Stream API. S2S takes a SQL query as input and converts it into an equivalent Java code using streams via a template-based code-generation technique (Section 3). As output, S2S produces a Java application that enables the translated code to be easily and readily used as benchmark thanks to the integration with the Java Microbenchmark Harness (JMH) [11]. We used S2S to generate BSS, the first benchmark suite for the Java Stream API, obtained by applying our tool to all the SQL queries used as benchmarks by the stream-fusion engine [32] (Section 4). We complement the paper with a review of related work (Section 5) and our concluding remarks (Section 6).

2 S2S Architecture

S2S is a SQL query compiler that generates Java source code making use of the Java Stream API to express query computation. In this section, we describe the architecture of S2S and discuss the details of its components. Figure 1 illustrates the execution flow of the benchmark generation process used by our tool. From a user prospective, using S2S is straightforward. To generate a benchmark, S2S needs very little user input, which is a database schema definition, a dataset for that schema and a SQL query to be converted into Java code. The schema and the dataset can be automatically retrieved from a connection URL defined with any JDBC [28] driver.

Once invoked, S2S first connects to the database with the provided connection and starts executing queries on the database to retrieve its table names and schemas, which will be converted to Java classes responsible of holding the data and retrieving the rows from the database, forming an in-memory database (step 1 in Figure 1). Then, before translating a SQL query into Java code, the query needs to be parsed, validated, and converted into a *query plan*, i.e., a representation of the query specifying the order and the implementation of each operator composing the query. In this way, query translation can be performed by traversing the query plan (step 2). Then, S2S translates the query plan into a Java class by converting the query operators into Java source code that makes use of the Stream API (step 3). Finally, S2S generates a JUnit [33] test for the translated query, as well as a benchmark leveraging JMH [11] as a harness (step 4).

We now provide additional details for each step involved in the described process.

In-memory DB generation. This step creates a Java class to retrieve and store the database tables into Java object arrays, acting as an in-memory database, and mostly involves a data-type translation schema from SQL types to Java ones. To create the in-memory database, S2S retrieves a list of tables from the original database using the JDBC connection provided as input. For each table, S2S determines its name and schema. Similarly to many object-relational mapping systems, S2S uses such information to create a Java class C_t for each SQL table t.[3]

When generating the Java classes modeling the database schema, S2S needs to map SQL types to Java ones. Currently, S2S supports the SQL types `INT`, `BIGINT`, `DOUBLE`, `CHAR` (fixed-size string), `VARCHAR` (variable-size string), and `DATE`, which are converted into the Java types `int`, `long`, `double`, `String` (for both `CHAR` and `VARCHAR`) and `java.sql.Date`, respectively.

Then, S2S generates a class (here called `DB`) to model a simple in-memory database. The `DB` class contains a field for each database table t typed as array of C_t. Moreover, the `DB` class generates code to retrieve the records for each table in the database.

Query planning. A query plan is usually represented as a tree, where internal nodes represent query operators and the leaves represents the tables involved in the query. The query plan specifies the order in which commutative operators should be executed (e.g., joins). All the query operators are physical ones, i.e., their implementation is specified in the query plan (e.g., a join can be implemented as nested-loop join or hash join).

Similarly to many existing query engines [10, 31], S2S uses Apache Calcite [2], a state-of-the-art open-source query planner, for query parsing and planning. The query plan generated by Calcite is composed of the following SQL operators (discussed in more details in Section 3): table scan, projection, predicate, aggregate, sort, limit, nested-loop join, and hash join. Table scan operators are the leaves of the query plan, while other operators are internal nodes. In particular, join operators (both nested-loop join and hash join) have exactly two children, all remaining operators have exactly one child.

Query compilation. After the creation of all the Java classes handling the data schema, S2S begins the process of translating SQL queries into Java code. Currently, S2S supports the SQL operators listed in Table 1, arithmetic and logical expressions, as well as the aggregation functions `COUNT`, `SUM`, `MIN`, `MAX` and `AVG`. For each input query q, S2S generates a Java class with a single `exec` method M_q that takes as input an instance of the class `DB`. All the code emitted for a given query will be included by S2S in this method.

Moreover, S2S generates the code so that all the streams created within such method do not escape it. In this way, the benchmarks generated by our tool can be used to evaluate

[2]S2S is available at https://github.com/usi-dag/S2S/releases/tag/v0.0.1.

[3]To simplify source-code generation, we leverage Java records [25].

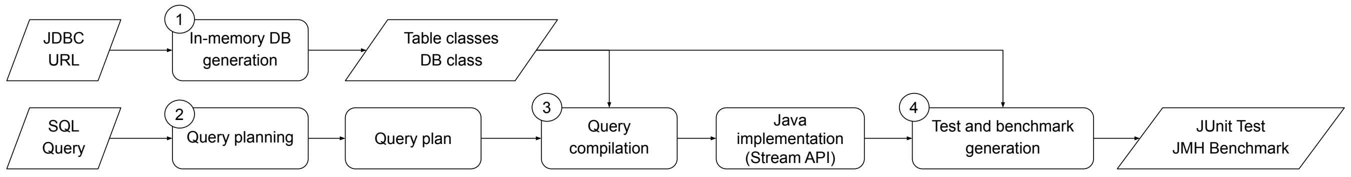

Figure 1. Execution flow of S2S. Parallelogram shape denote inputs and outputs while boxes denote the internal steps of S2S.

Table 1. Mapping between SQL operators and Java Stream API methods.

SQL operator	Stream method
Table scan	`.of`
Projection	`.map`
Predicate	`.filter`
Limit	`.limit`
Sort	`.sorted`
Aggregate	`.collect(custom)`
Nested-loop join	`.{toList, flatMap}`
Hash join	`.{collect(groupingBy), flatMap}`

optimizations on the Stream API based on static analyses [1, 20], which often cannot deal with streams that are not created and executed in a single method, though there are static analyses of streams that do not have such a limitation [15, 16], being based on interprocedural analyses.

Additional details about the query compilation will be discussed in Section 3.

Test and benchmark generation. After query compilation, we aim at ensuring that the translated code is semantically equivalent to the provided query through testing. To this end, S2S generates a JUnit [33] test for each translated query, which verifies that the result of a query q is equivalent to the original one. To do so, S2S executes q on the generated in-memory database using the Calcite engine and retrieves back its result set, which we use as expected result. Then, the correspondent method M_q is executed by passing as parameter an instance of the `DB` class and the result is compared to the expected result. Finally, S2S generates a benchmark leveraging JMH [11] as a harness.

3 Query Compilation

Here we describe the technical details of the translation from SQL queries to Java code done by S2S. Similarly to existing SQL query compilers [2, 22, 31, 39], S2S translates a query by visiting its query plan, which is done by the *visitor* component of S2S, i.e., an instance of the visitor pattern. The visitor maintains a state composed of three components:

- `v.body`: a string that contains the partially-generated method body implementing the given query.
- `v.elemType`: a string containing the type name of the stream elements.
- `v.decl`: a list of strings representing variable declarations referred by the generated method body.

When visiting each operator (excluding table scans), the visitor can assume that `v.body` ends with a piece of code representing an expression of type `Stream<T>` and that the name of the stream elements type `T` is stored in the field `v.elemType`. From now on, we will refer to such an invariant as state assumption.

In general, once the visitor visits a node in the query plan, it first recursively visits its children in depth-first order. Then it generates the code implementing the current operator and appends it to `v.body`. Thanks to the state assumption, each generated stream operation can be simply appended to `v.body` forming a chain of method calls. Finally, it sets `v.elemType` as the type name of the stream elements, resulting from the generated stream operation so that the state assumption holds. Some query operators (e.g., predicate) do not change the element type. In this case, the visitor leaves unchanged `v.elemType`. For the remaining query operators, such stream element type is a class generated by the visitor.

Figure 2 shows a pseudo-code algorithm describing the S2S' query compiler. We now provide more details on the template-based code-generation approach used by S2S which implements the mapping between SQL query operators and stream operations shown in Table 1, focusing on how the visit of each operator mutates `v.body`, `v.elemType` and `v.decl`.

Table Scan (Figure 2, lines 2 – 5). Since the visit is performed in depth-first order, and table scans are always the leaves of a query plan, table scan is always the first visited operator. During the visit to a table scan for a table t, the state is modified for the first time.

We note that during the first step of our approach (in-memory DB generation), a class `Ct` has been created as a representation of t and the *DB* class contains a field named t and typed as `Ct[]`. `v.elemType` is set to the name of the class `Ct` generated during the first phase, and `v.body` is set to the value `Arrays.stream(db.t)`. This state mutation respects the state assumption mentioned above, since the value of `v.body` is an expression of type `Stream<T>` where `T` is the class `Ct`.

Projection (Figure 2, lines 7 – 13). Since a projection p defines a mapping between tuples through a sequence of expressions, the visitor must generate a new class as a type

```
1  visit(Operator op) = op match {
2    case TableScan scan {
3      elemType = scan.type
4      body = "Arrays.stream(db.{scan.name})"
5    }
6
7    case Projection proj {
8      visit(proj.child)
9      cls = genDataClass(proj.type)
10     es = join(',', map(genExpr, proj.exprs))
11     elemType = cls.name;
12     body += ".map(row -> new {cls.name}({es}))"
13   }
14
15   case Predicate pred {
16     visit(pred.child)
17     expr = genExpr(pred.expr)
18     body += ".filter(row -> {expr})"
19   }
20
21   case Limit limit {
22     visit(limit.child)
23     body += ".limit({limit.value})"
24   }
25
26   case Sort sort {
27     visit(sort.child)
28     comparator = genComparatorExpr(sort)
29     decl.append("comp", comparator)
30     body += ".sorted(comp)"
31   }
32
33   case Aggregate agg {
34     visit(agg.child)
35     aggClass = genAggregatorClass(agg)
36     elemType = aggClass.name
37     if(!agg.hasGrouping) {
38       body += ".collect({aggClass.collector})"
39       body = "Stream.of({body})"
40     } else {
41       keysClass = genDataClass(agg.keyType)
42       body += ".collect(Collectors.groupingBy(
43         {keyClass.name}::new,
44         aggClass.collector
45       )).values().stream()"
46     }
47   }
48
49   case NestedLoopJoin join {
50     bVisitor = new Visitor()
51     bVisit = bVisitor.visit(join.left)
52     buildBody = bVisit.body + ".toList();"
53
54     visit(join.right)
55     joinedClass = genDataClass(join.type)
56     elemType = joinedClass.name
57     decl.append("left", buildBody)
58     body += // probe side as shown in Figure 3
59   }
60
61   case HashJoin join {
62     bVisitor = new Visitor()
63     bVisit = bVisitor.visit(join.left)
64     buildBody = bVisit.body + ".collect(
           Collectors.groupingBy{join.leftKey});"
65
66     visit(join.right)
67     joinedClass = genDataClass(join.type)
68     elemType = joinedClass.name
69     decl.append("empty", "emptyList()")
70     decl.append("left", buildBody)
71     body += // probe side as shown in Figure 4
72   }
73 }
```

Figure 2. Pseudo-code algorithm to convert SQL queries into stream-based Java source code.

for the mapped tuples. To this end, the visitor infers the types of the given projection expressions and generates a Java record `Cp` to hold these types, setting `v.elemType` to `Cp`. Then, the visitor converts each SQL expression in the projection into a Java expression. Finally, it appends to `v.body` a call to the `map` operations as `.map(row -> new Cp(e1,e2,...,en))` where `e1,e2,...,en` are the generated expressions.

Predicate (Figure 2, lines 15 – 19). Since filtering does not alter the element type in a given stream, visiting a predicate does not require mutating `v.elemType` or generating a new class. Similarly to a projection, a predicate holds a SQL expression which is converted into a Java expression. However, the SQL expression hold by a predicate cannot have an arbitrary return type as in the case of projections, but it must return a boolean value. This requirement is checked at query-validation time. Once the boolean expression *e* is converted, the visitor appends to `v.body` the code `.filter(row -> e)`.

Limit (Figure 2, lines 21 – 24). The visit to the limit operator is straightforward since it does not require to generate a class, mutate `v.elemType` and convert any expression. The visitor simply appends to `v.body` to code `.limit(n)`.

Sort (Figure 2, lines 26 – 31). Also the visit to the sort operator does not require to generate a new class or to mutate `v.elemType`. The sort operator holds a sequence of expressions which define the ordering constraints. The visitor converts these SQL expressions into Java lambda expressions and uses them to form a `java.util.Comparator` instance. Finally, the visitor appends to `v.body` the code `.sorted(c)` where `c` is a reference to the generated comparator.

Aggregate (Figure 2, lines 33 – 47). Aggregate nodes represent both grouped and not-grouped aggregations, depending on whether the group-by clause is used in the query. Grouped aggregations can project the result of aggregation functions and the grouped fields, whilst not-grouped aggregations can project only through aggregation functions. Translating an aggregate operator into methods of the Java Stream API is more challenging than the previously mentioned operators. This is motivated by the lack of a declarative interface in the Java Stream API which allows projecting the result of multiple aggregation functions. Indeed, while the Java Stream API offers methods which evaluate single aggregations, such as `Stream.count`, `Stream.min`, `Stream.max`, there is no direct way to express a sequence of aggregations, as one can simply obtain with a SQL query like the following (not-grouped) aggregation: `SELECT min(x), avg(y) FROM T`.

Due to such missing feature, S2S converts aggregations using the method `Stream.collect(Collector)`, as shown in Table 1. A collector [24] in Java is a result container specified by four functions that together allows accumulating entries. In particular, the functions allow to create a new container (`supplier()`), to update the container by incorporating a new entry (`accumulator()`), to combine two containers into one (`combiner()`) and to perform a final transformation on the container once all entries have been processed (`finisher()`). To translate an aggregate operator, S2S generates a new class with a method for each of the abovementioned function and derives a collector from the generated class.

Once the class implementing the aggregation has been generated, the visitor performs a state mutation. In particular, `v.elemType` is set to the name of the generated class (which here we call `Agg`), while the mutation of `v.body` depends on whether the aggregation is grouped or not. In both cases, since `Stream.collect` is a terminal operator (i.e., it does not return a stream, but the result container of the aggregation), in order to respect the state assumption, the result of the `collect` method call needs to be wrapped into a new stream.

For not-grouped aggregations, the visitor simply appends to `v.body` the code `.collect(Agg.collector())`, where the method `Agg.collector()` refers to the mentioned collector derived from the class `Agg`. Then, it wraps the stream result into a new stream with a single element by mutating `v.body` to `Stream.of(v.body)`. In case of grouped aggregations, the visitor first generates a class `KeysRecord` for the keys (i.e., the fields used to perform the grouping), and then it appends to `v.body` the following code:

```
.collect(Collectors.groupingBy(
    row -> new KeysRecord(row),
    Agg.collector()
)).values().stream()
```

To preserve the state assumption, S2S wraps the map values into a new stream by adding the code `.values().stream()`.

Joins (Figure 2, lines 49 – 72). In contrast to the other SQL operators, joins have two children in a query plan, hence the visitor needs to handle such a different structure. S2S supports two join implementations, namely hash join (commonly used when the join condition is an equality among fields) and nested-loop join (used for other arbitrary join conditions). For both join implementations, the left child is called *build side*, while the right child is called *probe side*. To avoid executing the pipeline of operators which compose each join side more than once, the implementation of both joins materializes the tuples from the build side into an intermediate data structure B. Then, the implementation scans the probe side and, for each element p, finds all the matching pairs $(p, b), b \in B$ for which the join condition holds.

When visiting both join implementations, S2S first creates a new visitor $v2$ which is used to visit the build side. Then, the original visitor v takes `v2.body` appending to it the code for materializing that side into B. We denote C_{v2} the resulting code. Then, `v.decl` is set to `v2.decl` with appended a new declaration for B, initialized with the code C_{v2}. Finally, v visits the probe side of the join, and appends to `v.body` a call to the `flatMap` method defined in the Java Stream API, which takes care of selecting the join matching pairs (p, b) by accessing

```
List<B> list = left.toList(); // build side

// probe side
right.flatMap(r -> list
  .stream()
  .filter(l -> joinCondition(l, r))
  .map(l -> mapping(l, r)));
```

Figure 3. Template for the nested-loop join operator.

```
List<B> empty = Collections.emptyList();

// build hashmap for build side
Map<K, List<B>> map = left.collect(
    Collectors.groupingBy(leftKey));

return right.flatMap(r -> // probe side
  map.getOrDefault(rightKey(r), empty)
    .stream()
    .map(l -> mapping(l, r)));
```

Figure 4. Template for the hash join operator.

B, as further discussed in the next paragraphs, which discuss the difference w.r.t. the two join implementations.

Nested-loop join (Figure 2, lines 49 – 59). The build side of a nested-loop materializes its tuple into a list L_b. Then, the operator pipeline on the probe side is executed and, for each element p, the whole list L_b is scanned. For each element b in L_b a pair of tuple (p, b) is created and if such pair passes the join condition, then it will be an output row of the nested-loop join.

Such a behavior is implemented with the Java Stream API as shown in the template reported in Figure 3. In particular, the placeholder `left` represents the code generated by visiting the build side with the visitor $v2$, the assignment to the `list` variable is added mutating `v.decl`. Placeholder `right` represents the code generated by visiting the build side with v, and the call to `flatMap` is the mutation of `v.body` performed by the visitor. Also, the parametric class `B` in the declaration is set to the value `v2.elemType`. We note that the mapping function, which is here omitted for brevity, creates elements of a generated class T which contains the fields from both join children. `v.elemType` is then set to T.

Hash join (Figure 2, lines 61 – 72). Generating the code for the hash-join operator is very similar to the case of nested-loop joins. The algorithm differs from the previous one in finding the matching pairs. In case of nested-loop joins, the build side is materialized into a list which is fully scanned for each element in the probe side, while in case of hash joins the build side is materialized into an hash map and matching pairs are found by performing a hash lookup on that map. In particular, the code-generation template for hash join is depicted in Figure 4. Placeholders `left`, `right`, `mapping` have the same meaning as in the case of nested-loop joins. Hash joins also require to generate code for extracting

```
WITH top_orders AS (
  SELECT * FROM orders
  WHERE o_orderdate >= DATE '1995-12-01'
  ORDER BY o_totalprice DESC
  LIMIT 1000)

SELECT SUM(o_totalprice)
FROM lineitem, top_orders
WHERE l_shipdate >= DATE '1995-12-01'
AND o_orderkey = l_orderkey
```

Figure 5. Original definition of Q8 in SQL.

the map keys used for finding matching pairs. To this end, the visitor creates a Java record (C) with the fields used in the key. Placeholders `leftKey` and `rightKey` in the template represent the generated code which creates an instance of C from an element of the build side stream and probe side stream, respectively.

4 Use Case: Generating BSS with S2S

In this section, we present a use case to demonstrate the capabilities of S2S in generating benchmarks from SQL queries. With S2S, we generated BSS, which to the best of our knowledge is the first benchmark suite for the Java Stream API. BSS has been obtained by running S2S on a set of queries that have been proposed as benchmarks by the authors of the stream-fusion engine [32]. The queries are based on the dataset of TPC-H [36]. The original benchmark suite is composed of 7 queries; however, there is no query that makes use of all SQL operators. With the goal of presenting a complete example of query compilation that involves the translation of each SQL operator, we created another query (here called Q8) listed in Figure 5. The query is inspired by the combination of two original queries, Q5 and Q7. In the following text, we describe the query-compilation process performed by S2S on Q8, which exemplifies the code-generation approach described in Section 3. [4]

The class generated by S2S taking Q8 as input is depicted in Figure 6. The query plan is composed as follows. The root is an aggregate (`sum(o_totalprice)`) node, followed by a join operator with a join condition (`o_orderkey = l_orderkey`). The build side of the join is a projection (`l_orderkey`), followed by a predicate (on `l_shipdate`) and a table scan (on `lineitem`). The probe side of the join is a limit followed by a sort (by field `o_totalprice` in descending order) that is followed by a projection (of fields `o_orderkey`, `o_totalprice`).

The query-plan visit first goes down to the join operator, then a new visitor (`v2`) is created to handle the build side, generating code that creates the join hash map `hm` (lines 26–28 in Figure 6). Then, the first visitor (`v`) appends to `v.decl` the declarations of `E` (defined in `v2`) and `hm`, the latter is obtained by appending the grouping collector to `v2.body` (line 29). We note that the method `Collectors.groupingBy` (used in line 29) returns a collector that groups the stream elements according to the given classification function, accumulating the result into a map (hash map by default). Now, `v` can visit the probe side, going down to the table scan (on `order`), where it sets `v.body` to `Arrays.stream(db.orders)`. Then, it visits the predicate operator, generating a call to `filter` (line 32), and the projection operator, which generates a call to `map` (line 33). Then, by visiting the sort operator, `v` generates the declaration for the `Comparator` instance (line 3) and appends to `v.body` a call to `sorted` (line 34). Then, the limit operator is visited, appending to `v.body` a call to `limit` (line 35).

At this point, the probe side has been completely visited, and `v` appends to `v.body` the call to `flatMap` to join the probe side with the build side (lines 36–39). Finally, the aggregate node is visited, generating the class that takes care of evaluating the aggregation (class `Agg` in the code, lines 9–22), and appending to `v.body` the call to `collect` (line 40).

5 Related Work

The lack of suitable benchmarks for specific evaluation needs is a long-standing problem in the research community, as demonstrated by the rich presence of research work proposing new techniques to synthetize custom benchmarks automatically. Joshi et al. [13] propose a framework that automatically generate benchmarks starting from CPU-level workload characteristics provided as input. Zheng et al. [40] locate real-world applications hosted on GitHub and exhibiting user-defined behavior, using their unit tests to drive benchmarking workloads. Van Ertvelde et al. [38] generate benchmarks based on the CPU behavior of existing closed-source applications without revealing proprietary information. Bell et al. [12] aim at creating short running workloads from actual applications. Automatic benchmark generation has also been proposed for specific domains, such as I/O-intensive parallel computations [9], bug detection [14], predictive machine-learning models [8], and matrix operations [19].

Our tool differs from the above work as it is the first one to allow automatic benchmark generation for the Java Stream API, which related work does not target. Moreover, while the above techniques typically require users to provide a specification of the desired workload characterics as input, S2S is based on a different approach, i.e., translating existing SQL queries to stream-based Java applications, which makes it possible to generate a large number of benchmarks for the Java Stream API thanks to the vast availability of data-processing workloads designed for relational databases.

6 Concluding Remarks

This paper presents S2S, an automatic benchmark generator for the Java Stream API. S2S is a SQL query compiler able to convert SQL queries into Java stream-based workloads using a template-based code-generation approach. We use

[4]BSS is composed of all the 8 translated queries and is publicly available at https://github.com/usi-dag/BSS/releases/tag/v0.0.1.

```java
public class Query_8 {
  Date const_0 = Date.valueOf("1995-12-01");
  Comparator<P1> comp = Comparator.comparing( (
      P1 x) -> x.o_totalprice()).reversed();


  record P0(...) {} // fields omitted
  record J0(...) {} // fields omitted
  record P1(...) {} // fields omitted
  class Agg {
    int sum_total;
    void accumulate(J0 row) {
      sum_total += row.o_totalprice;
    }
    Agg combine(Agg other) {
      sum_total += other.sum_total;
      return this;
    }
    static Collector<J0, Agg, Agg> collector(){
      return Collectors.of(
        Agg::new,Agg::accumulate,Agg::combine);
    }
  }

  public List<Agg> exec(DB db) {
    List<P0> E = Collections.emptyList();
    var hm = Arrays.stream(db.lineitem)
        .filter(row -> (row.l_shipdate.
            compareTo(const_0) >= 0))
        .map(row -> new P0(...))
        .collect(Collectors.groupingBy(row ->
            row.l_orderkey()));

    var stream = Arrays.stream(db.orders)
        .filter(row -> (row.o_orderdate().
            compareTo(const_0) >= 0))
        .map(row -> new P1(...))
        .sorted(comp)
        .limit(1000)
        .flatMap(r ->
          hm.getOrDefault(r.o_orderkey(), E)
           .stream()
           .map(l -> new J0(...)))
        .collect(Agg.collector());

    return Stream.of(stream).toList();
  }
}
```

Figure 6. Code generated by S2S from query Q8 (Figure 5).

S2S to generate BSS, the first benchmark suite for the Java Stream API, obtained by applying S2S on all the SQL queries used as benchmarks by the stream-fusion engine. Thanks to S2S, numerous benchmarks for the Stream API can be easily generated by converting existing data-processing workloads designed for relational databases.

Limitations. One limitation of our approach is that S2S uses a pre-determined set of stream operations (see Table 1) in the translation. While these operations are the most relevant for data-processing benchmarks obtained from SQL queries, an implication of our approach is that some operations of the Java Stream API will never be used by any workload resulting from S2S. Hence, the resulting benchmarks cannot exercise all methods defined by the Java Stream API.

Future Work. We plan to address the abovementioned limitation as future work, by exploring alternative conversions of SQL operators to Java streams with the goal of covering more operations defined by the Stream API. Such an extension would also allow measuring the performance difference among equivalent stream operations. Simple examples of equivalent conversions are, e.g., implementing joins with `mapMulti` instead of `flatMap`, using `findFirst` and `findAny` in place of the `limit(1)` operator. Moreover, the exploration of alternative conversions would immediately enable a performance evaluation of the parallelism implemented in the Stream API. Indeed, our approach conceptually already supports parallel streams, since all generated operations support parallel evaluation. Thus, generating benchmarks for parallel streams would only require adding `.parallel()` in the generated source code, that we will implement in a future release.

While S2S was implemented as a compiler for relational queries, it could be extended to support NoSQL queries (e.g., MongoDB [21]). Such an extension could also help in extending the coverage of the Java Stream API within the generated benchmarks. Moreover, while S2S focuses on the Java Stream API, the same approach could be reused to generate benchmarks for LINQ and Scala Collections. In this setting, part of the existing infrastructure could be reused, in particular the in-memory DB population as well as the tests and benchmarks generation. However, the query compiler should be extended with a new backend that takes care of generating code for a different language library.

As future work, we plan to generate a large and comprehensive benchmark suite for the Java Stream API by converting other established data-processing workloads. We plan to use the new suite to conduct a performance analysis on the Java Stream API; in particular, we plan to evaluate the performance of the Java Stream API among different JDKs.

Acknowledgments

The work presented in this paper has been supported by Oracle (ERO project 1332) and the Swiss National Science Foundation (project 200020_188688).

References

[1] Matteo Basso, Filippo Schiavio, Andrea Rosà, and Walter Binder. 2022. Optimizing Parallel Java Streams. In *ICECCS*. 23–32. https://doi.org/10.1109/iceccs54210.2022.00012

[2] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *SIGMOD*. 221–230. https://doi.org/10.1145/3183713.3190662

[3] Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2014. Clash of the Lambdas. 1–11. https://doi.org/10.48550/arXiv.1406.6631

[4] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko

Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*. 169–190. https://doi.org/10.1145/1167515.1167488

[5] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. 2011. The Mixed Workload CH-BenCHmark. In *DBTest*. 1–6. https://doi.org/10.1145/1988842.1988850

[6] Standard Performance Evaluation Corporation. 2022. SPECjbb2015. https://www.spec.org/jbb2015/.

[7] Standard Performance Evaluation Corporation. 2022. SPECjvm2008. https://www.spec.org/jvm2008.

[8] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Synthesizing Benchmarks for Predictive Modeling. In *CGO*. 86–99. https://doi.org/10.1109/cgo.2017.7863731

[9] Meng Hao, Weizhe Zhang, You Zhang, Marc Snir, and Laurence T. Yang. 2019. Automatic Generation of Benchmarks for I/O-intensive Parallel Applications. *J. Parallel and Distrib. Comput.* 124 (2019), 1–13. https://doi.org/10.1016/j.jpdc.2018.10.004

[10] Michael Hausenblas and Jacques Nadeau. 2013. Apache Drill: Interactive Ad-hoc Analysis at Scale. *Big data* 1, 2 (2013), 100–104. https://doi.org/10.1089/big.2013.0011

[11] JMH Team. 2022. JMH. online. https://github.com/openjdk/jmh

[12] Lizy Kurian John. 2005. The Case for Automatic Synthesis of Miniature Benchmarks. In *MoBS*. 4–8. https://doi.org/10.1145/1958746.1958748

[13] Ajay Joshi, Lieven Eeckhout, and Lizy K John. 2008. The Return of Synthetic Benchmarks. In *SPEC Benchmark Workshop*.

[14] Vineeth Kashyap, Jason Ruchti, Lucja Kot, Emma Turetsky, Rebecca Swords, Shih An Pan, Julien Henry, David Melski, and Eric Schulte. 2019. Automated Customized Bug-Benchmark Generation. In *SCAM*. 103–114. https://doi.org/10.1109/SCAM.2019.00020

[15] Raffi Khatchadourian, Yiming Tang, and Mehdi Bagherzadeh. 2020. Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams. *Science of Computer Programming* 195 (2020), 1–24. https://doi.org/10.1016/j.scico.2020.102476

[16] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Syed Ahmed. 2018. A Tool for Optimizing Java 8 Stream Software via Automated Refactoring. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 34–39. https://doi.org/10.1109/SCAM.2018.00011

[17] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream Fusion, to Completeness. In *POPL*. 285–299. https://doi.org/10.1145/3093333.3009880

[18] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[19] John McCalpin and Mark Smotherman. 1995. Automatic Benchmark Generation for Cache Optimization of Matrix Operations. In *ACM-SE*. 195–204. https://doi.org/10.1145/1122018.1122054

[20] Anders Møller and Oskar Haarklou Veileborg. 2020. Eliminating Abstraction Overhead of Java Stream Pipelines Using Ahead-of-Time Program Optimization. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 1–29. https://doi.org/10.1145/3428236

[21] MongoDB Team. 2020. The most popular database for modern apps |MongoDB. https://www.mongodb.com/.

[22] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550. https://doi.org/10.14778/2002938.2002940

[23] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. *The Star Schema Benchmark and Augmented Fact Table Indexing*. Springer-Verlag, 237–252. https://doi.org/10.1007/978-3-642-10424-4_17

[24] Oracle. 2022. Class Collector. https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/stream/Collector.html.

[25] Oracle. 2022. Class Record. https://docs.oracle.com/javase/specs/jls/se18/html/jls-8.html#jls-8.10.

[26] Oracle. 2022. Package java.util.stream. https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/util/stream/Stream.html.

[27] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *PLDI*. 31–47. https://doi.org/10.1145/3314221.3314637

[28] George Reese. 2000. *Database Programming with JDBC and JAVA*. "O'Reilly Media, Inc.".

[29] Francisco Ribeiro, João Saraiva, and Alberto Pardo. 2019. Java Stream Fusion: Adapting FP Mechanisms for an OO Setting. In *SBLP*. 30–37. https://doi.org/10.1145/3355378.3355386

[30] Eduardo Rosales, Andrea Rosà, Matteo Basso, Alex Villazón, Adriana Orellana, Ángel Zenteno, Jhon Rivero, and Walter Binder. 2022. Characterizing Java Streams in the Wild. In *2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 143–152. https://doi.org/10.1109/iceccs54210.2022.00025

[31] Filippo Schiavio, Daniele Bonetta, and Walter Binder. 2021. Language-Agnostic Integrated Queries in a Managed Polyglot Runtime. *Proc. VLDB Endow.* 14, 8 (2021), 1414–1426. https://doi.org/10.14778/3457390.3457405

[32] A. Shaikhha, M. Dashti, and C. Koch. 2018. Push vs. Pull-Based Loop Fusion in Query Engines. *Journal of Functional Programming* 28 (2018), 539–550. https://doi.org/10.1017/s0956796818000102

[33] Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory. 2011. *JUnit in Action, 2nd Edition*. Manning Publications Company. https://doi.org/10.21019/9781582121994.ch9

[34] TPC. 2022. TPC-C. https://www.tpc.org/tpcc/default5.asp.

[35] TPC. 2022. TPC-DS. https://www.tpc.org/tpcds/default5.asp.

[36] TPC. 2022. TPC-H. https://www.tpc.org/tpch/default5.asp.

[37] TPC. 2022. TPCx-BB. https://www.tpc.org/tpcx-bb/default5.asp.

[38] L. Van Ertvelde and L. Eeckhout. 2010. Benchmark Synthesis for Architecture and Compiler Exploration. In *IEEE IISWC*. 1–11. https://doi.org/10.1109/iiswc.2010.5650208

[39] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 15–28. https://doi.org/10.21236/ada575859

[40] Yudi Zheng, Andrea Rosà, Luca Salucci, Yao Li, Haiyang Sun, Omar Javed, Lubomír Bulej, Lydia Y. Chen, Zhengwei Qi, and Walter Binder. 2016. AutoBench: Finding Workloads That You Need Using Pluggable Hybrid Analyses. In *SANER*, Vol. 1. 639–643. https://doi.org/10.1109/saner.2016.70