Università della Svizzera italiana

SQL to Stream with S2S

An Automatic Benchmark Generator for the Java Stream API

Filippo Schiavio Andrea Rosà Walter Binder

Java Stream API - Introduction

- The Java Stream API allows manipulating sequences of elements in a function style
- Computation is expressed as a pipeline of operations
  - Stream creation (e.g., creating a stream from an array or a collection)
  - Intermediate operations (e.g., mapping a function or filtering by a predicate)
  - Terminal operation (e.g., applying an aggregation function)
- Concise style improves readability and maintainability
- Facilitates parallelization: automatically splits computation using fork-join tasks

Java Stream API - Performance

- The Stream API often leads to performance degradation [1]
- Existing tools optimize streams by transformation them to loops [2], [3]
  - Lack of benchmarks: these tools have been evaluated on (manually selected or implemented) micro-benchmarks

[1] O. Kiselyov et al., "Stream Fusion, to Completeness". POPL 2017.

[2] A. Møller et al., "Eliminating Abstraction Overhead of Java Stream Pipelines Using Ahead-of-Time Program Optimization". OOPSLA 2020

[3] M. Basso et al., “Optimizing Parallel Java Streams”. ICECCS 2022

The Problem

- Lack of benchmarks dedicated to the Java Stream API
  - Most of existing Java benchmarks (e.g., DaCapo) do not evaluate streams
  - Only Renaissance [1] benchmark suite have (only three) workloads for stream
- Researchers and language implementors need benchmarks
- Designing benchmarks is tedious and requires a lot of engineering effort [2]

[1] A. Prokopec et al., "Renaissance: Benchmarking Suite for Parallel Applications on the JVM". PLDI 2019

[2] Y. Zheng et al., “AutoBench: Finding Workloads That You Need Using Pluggable Hybrid Analyses”. SANER 2016

The Solution

- The Stream API is basically a data-processing library
- Many data-processing benchmarks already exist!
- The database community created many benchmarks for SQL
  - I.e., a dataset and a set of SQL queries, e.g., TPC-H

```java
int sumOfSquaresEven(int[] source) {
    return IntStream.of(source)
        .filter(x -> x % 2 == 0)
        .map(x -> x * x)
        .sum();
}
```

```sql
SELECT SUM(x * x)
FROM T
WHERE x % 2 == 0
```

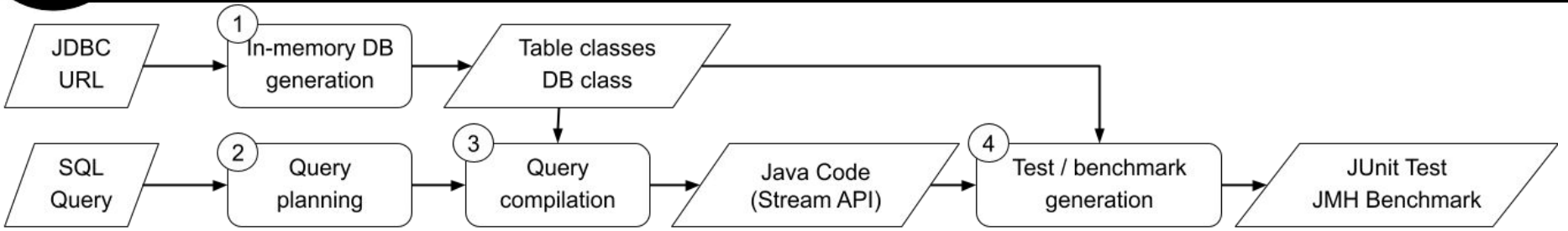
- Reusing SQL benchmarks for the Java Stream API seems a natural solution
- Automatic conversion of SQL query into Java code that uses the Stream API

SQL to Stream with S2S

- Before execution, a SQL query is parsed and converted into a query plan
  - An AST-like representation of the query
- The query plan is then executed by interpretation or compilation
- S2S compiles a query plan into Java source code that makes use of the Stream API
- Queries generated with S2S are then:
  - Tested by comparing the results with an existing SQL engine (Apache Calcite [1])
  - Wrapped into (JMH) benchmarks
- S2S is open-source: github.com/usi-dag/S2S

[1] E. Begoli et al., “Apache Calcite: A foundational framework for optimized query processing over heterogeneous data sources”. SIGMOD 2018

S2S Overview



0. Input: JDBC url for a DB connection (e.g., a SQLite local dataset) and a SQL query
1. Connects to a database and reads table schemas
2. Converts the input SQL query into a query plan (leveraging Apache Calcite)
3. Compiles the query plan into a Java implementation that uses the Stream API
4. Generates test (JUnit) and benchmarks (JMH) source classes

S2S Query Compiler - Conversion Table

SQL Query Plan Node (operator)	Stream API Method
Table Scan	Stream.of
Projection	Stream.map
Predicate	Stream.filter
Limit	Stream.limit
Sort	Stream.sorted
Aggregate	Stream.collect (custom collector)
Nested-Loop Join	Stream.toList / Stream.flatMap
Hash Join	Stream.collect(grupingByCollector) / Stream.flatMap

S2S Compiler: Aggregations

- Aggregation is the most challenging operator for our conversion
  - A SQL aggregation is defined by a list of aggregation functions
    - E.g., **`SELECT MIN(x), AVG(y) from T`**
  - There is no interface in the Java Stream API that allows defining multiple aggregation functions
- S2S compiles aggregations with the terminal operator **`.collect`**, a mutable reduction
  - The generated code representing the collector is imperative
  - See the paper for details

BSS: Benchmark Suite for the Stream API

- BSS is the first publicly available benchmark suite for the Java Stream API
- It is composed on a set of 8 simple queries
  - Dataset from the TPC-H benchmark
  - 7 queries from stream-fusion engine [1]
  - One query added to show most of the supported operators in a single query

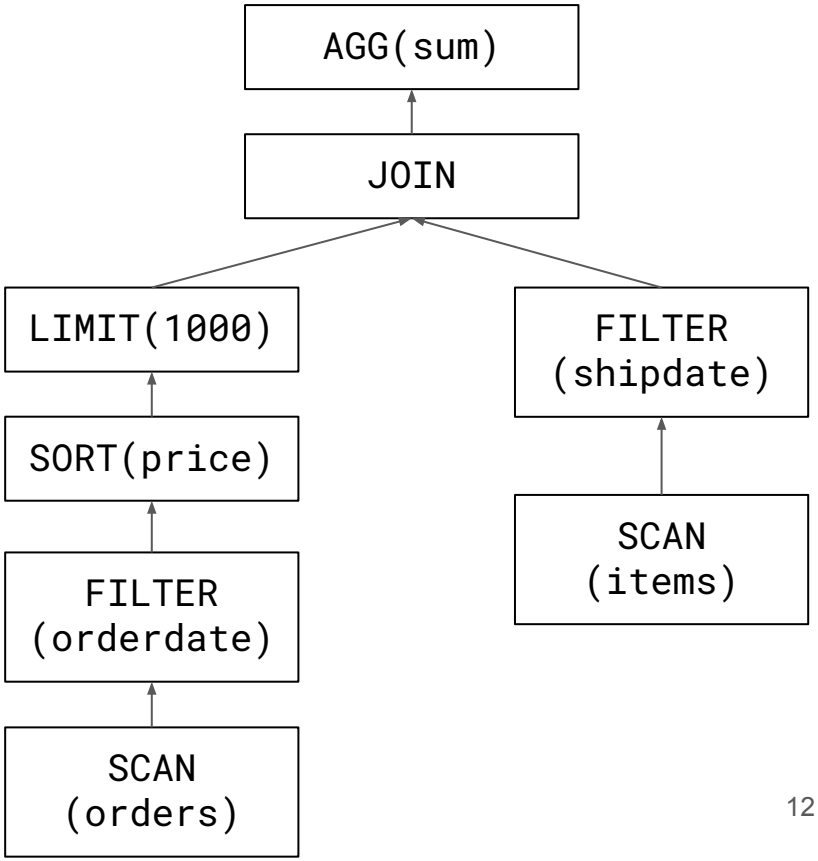
[1] A. Shaikhha et al., “Push vs. Pull-Based Loop Fusion in Query Engines”. Journal of Functional Programming. 2016

S2S Compilation Example (query)

```
WITH top_orders AS (
    SELECT * FROM orders
    WHERE orderdate >= '1995-12-01'
    ORDER BY price DESC
    LIMIT 1000)
WITH fastship_items AS (
    SELECT * FROM items
    WHERE shipdate <= '1995-12-02')
SELECT SUM(I.price) as total
FROM top_orders O, fastship_items I
WHERE O.orderkey = I.orderkey
```

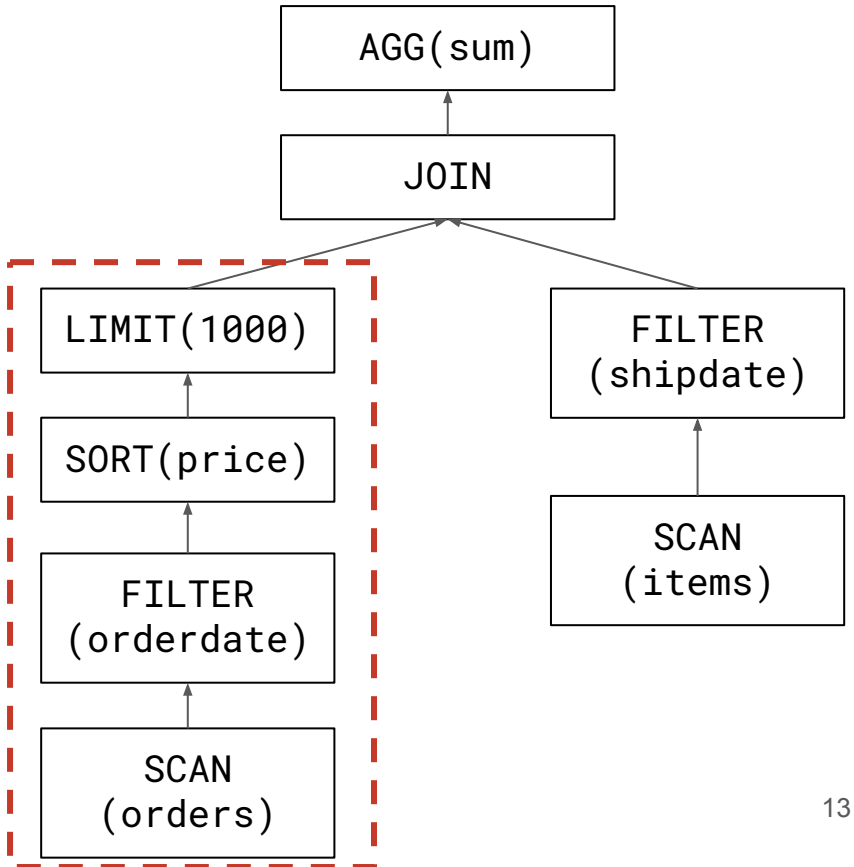
S2S Compilation Example (plan)

```
WITH top_orders AS (
    SELECT * FROM orders
    WHERE orderdate >= '1995-12-01'
    ORDER BY price DESC
    LIMIT 1000)
WITH fastship_items AS (
    SELECT * FROM items
    WHERE shipdate <= '1995-12-02')
SELECT SUM(I.price) as total
FROM top_orders O, fastship_items I
WHERE O.orderkey = I.orderkey
```



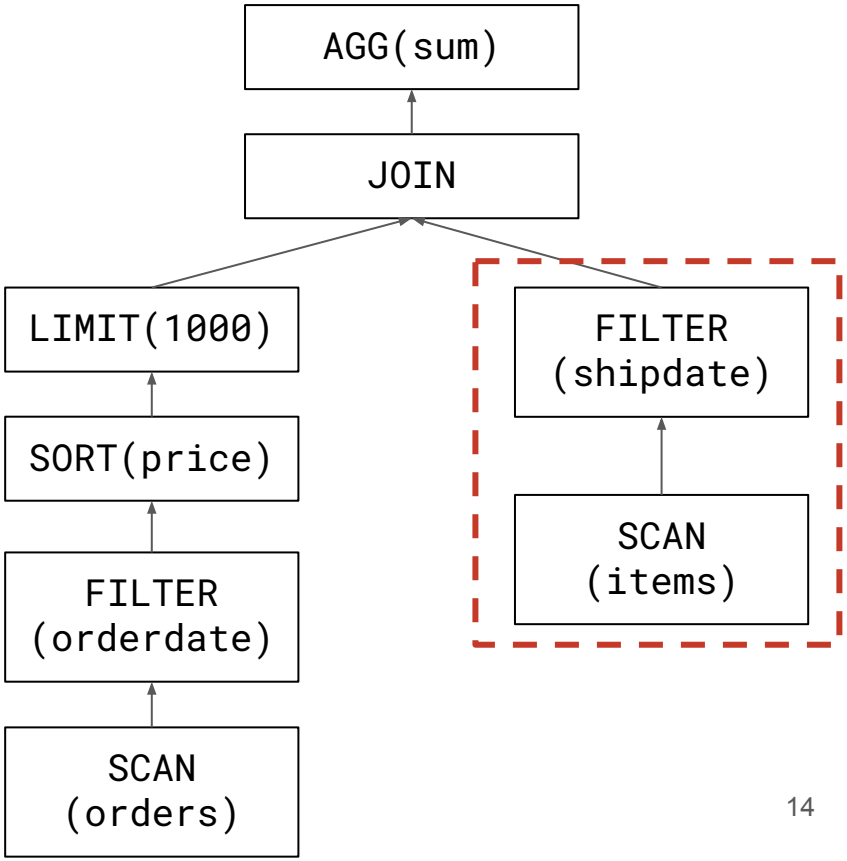
S2S Compilation Example (plan)

```
WITH top_orders AS (
    SELECT * FROM orders
    WHERE orderdate >= '1995-12-01'
    ORDER BY price DESC
    LIMIT 1000)
WITH fastship_items AS (
    SELECT * FROM items
    WHERE shipdate <= '1995-12-02')
SELECT SUM(I.price) as total
FROM top_orders O, fastship_items I
WHERE O.orderkey = I.orderkey
```



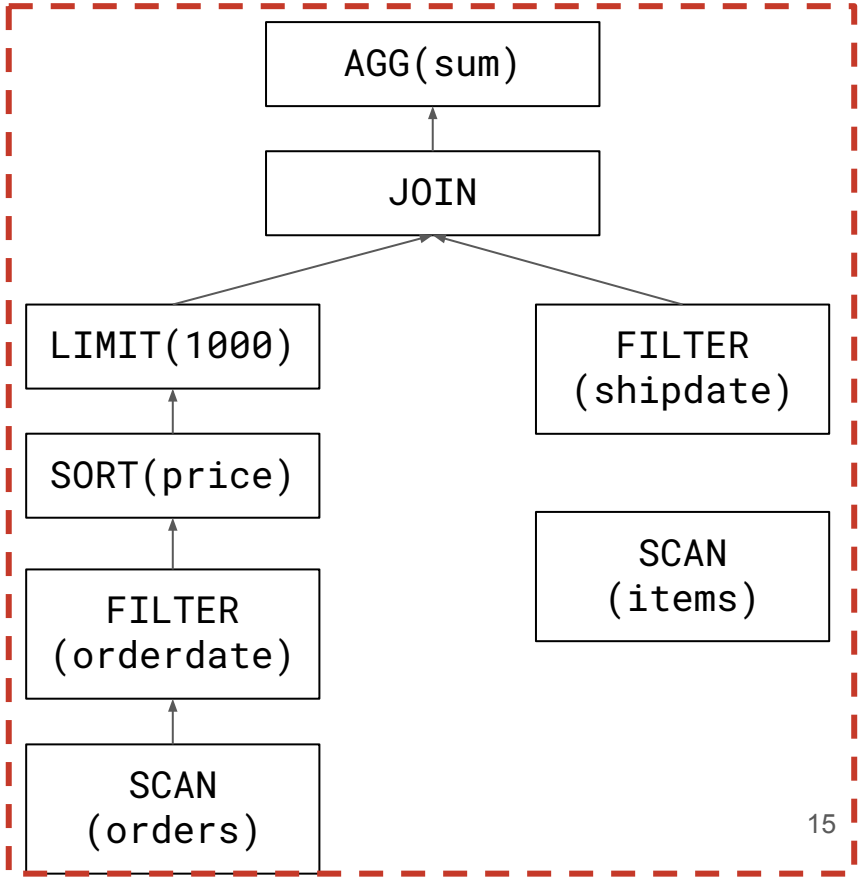
S2S Compilation Example (plan)

```
WITH top_orders AS (
    SELECT * FROM orders
    WHERE orderdate >= '1995-12-01'
    ORDER BY price DESC
    LIMIT 1000)
WITH fastship_items AS (
    SELECT * FROM items
    WHERE shipdate <= '1995-12-02')
SELECT SUM(I.price) as total
FROM top_orders O, fastship_items I
WHERE O.orderkey = I.orderkey
```



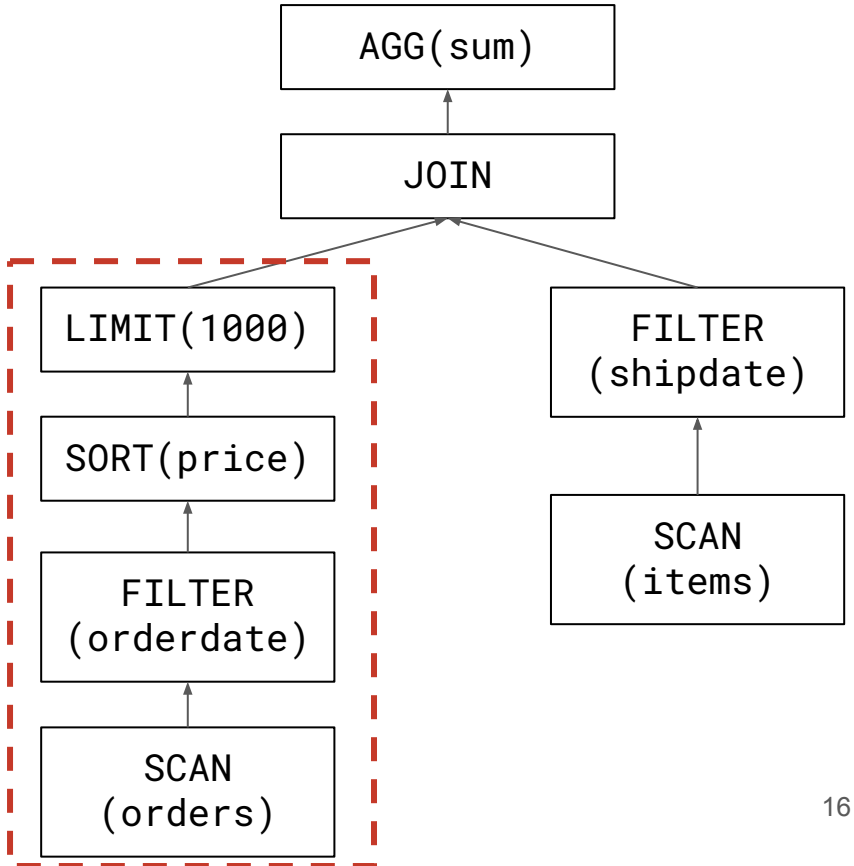
S2S Compilation Example (plan)

```
WITH top_orders AS (
    SELECT * FROM orders
    WHERE orderdate >= '1995-12-01'
    ORDER BY price DESC
    LIMIT 1000)
WITH fastship_items AS (
    SELECT * FROM items
    WHERE shipdate <= '1995-12-02')
SELECT SUM(I.price) as total
FROM top_orders O, fastship_items I
WHERE O.orderkey = I.orderkey
```



S2S Compilation Example (codegen)

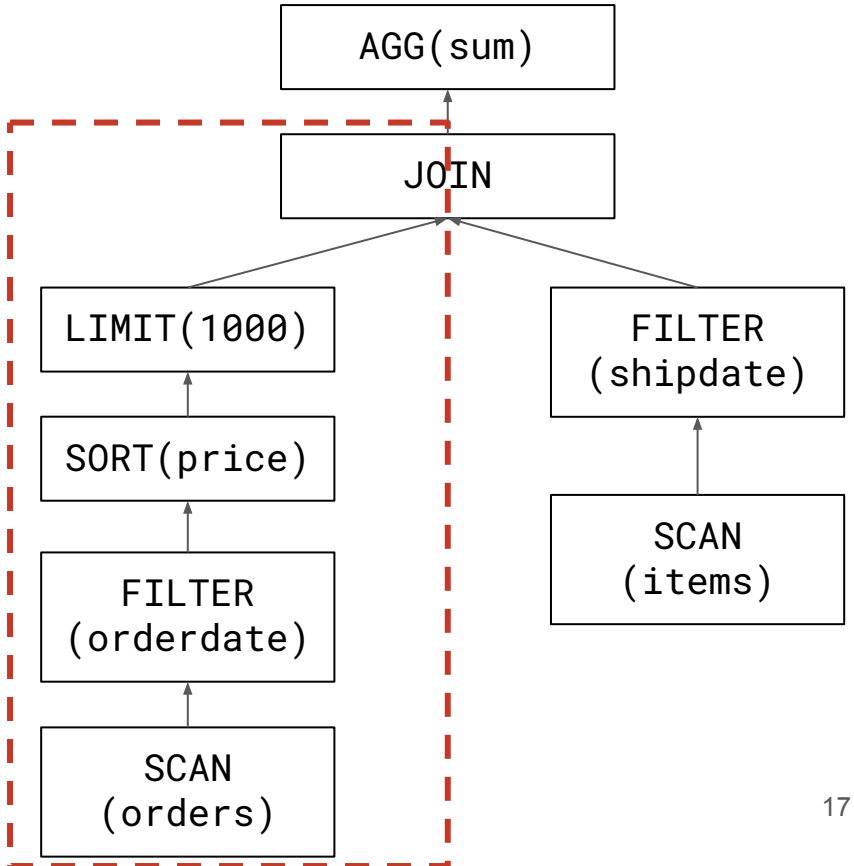
```
HashMap<JoinKey, List<OrderRow>> hm =
  Stream.of(DB.orders)
        .filter(o -> o.orderdate > DATE_FROM)
        .sorted(o -> o.price)
        .limit(1000)
        .collect(groupByCollector(orderkey));

int result =
  Stream.of(DB.items)
        .filter(i -> i.shipdate < DATE_TO)
        .flatMap(i -> hm.get(i.orderkey)
                        .map(joinRows(i, o)))
        .sum(joined -> joined.price)
```



S2S Compilation Example (codegen)

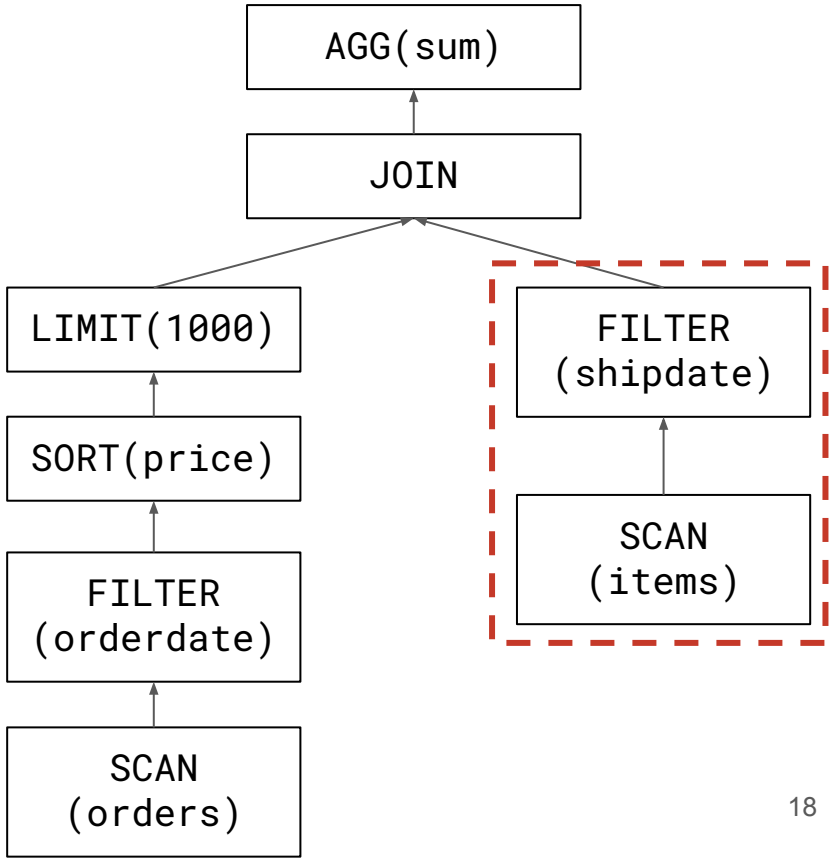
```
HashMap<JoinKey, List<OrderRow>> hm =
  Stream.of(DB.orders)
        .filter(o -> o.orderdate > DATE_FROM)
        .sorted(o -> o.price)
        .limit(1000)
        .collect(groupByCollector(orderkey));
```

```
int result =
  Stream.of(DB.items)
        .filter(i -> i.shipdate < DATE_TO)
        .flatMap(i -> hm.get(i.orderkey)
                        .map(joinRows(i, o)))
        .sum(joined -> joined.price)
```



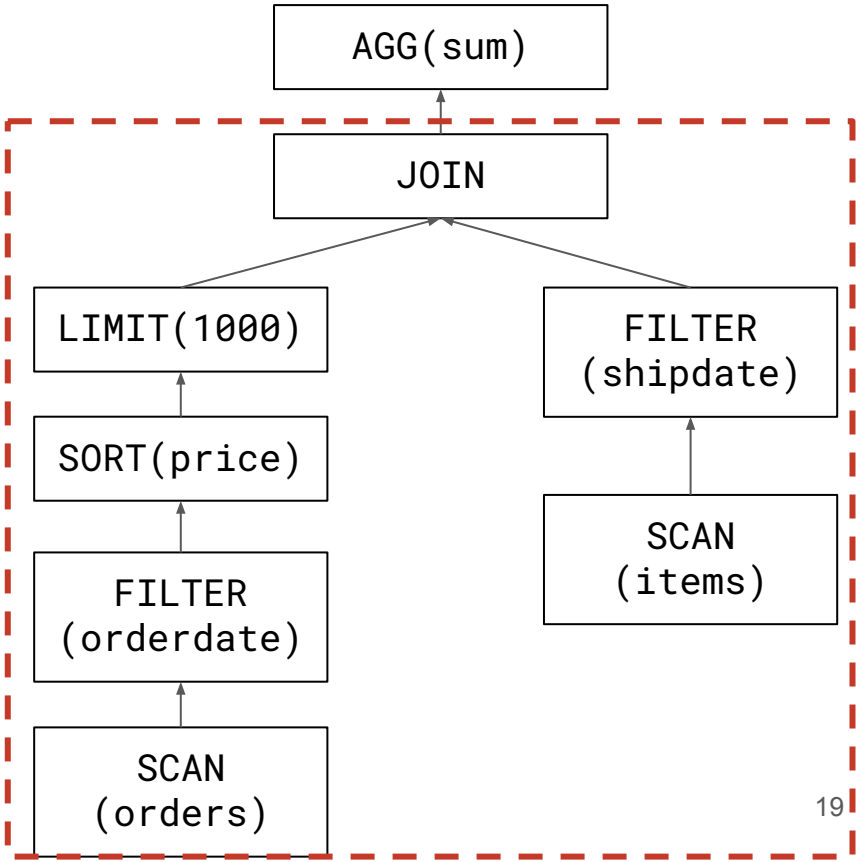
S2S Compilation Example (codegen)

```
HashMap<JoinKey, List<OrderRow>> hm =
  Stream.of(DB.orders)
        .filter(o -> o.orderdate > DATE_FROM)
        .sorted(o -> o.price)
        .limit(1000)
        .collect(groupByCollector(orderkey));

int result =
  Stream.of(DB.items)
        .filter(i -> i.shipdate < DATE_TO)
        .flatMap(i -> hm.get(i.orderkey)
                        .map(joinRows(i, o)))
        .sum(joined -> joined.price)
```



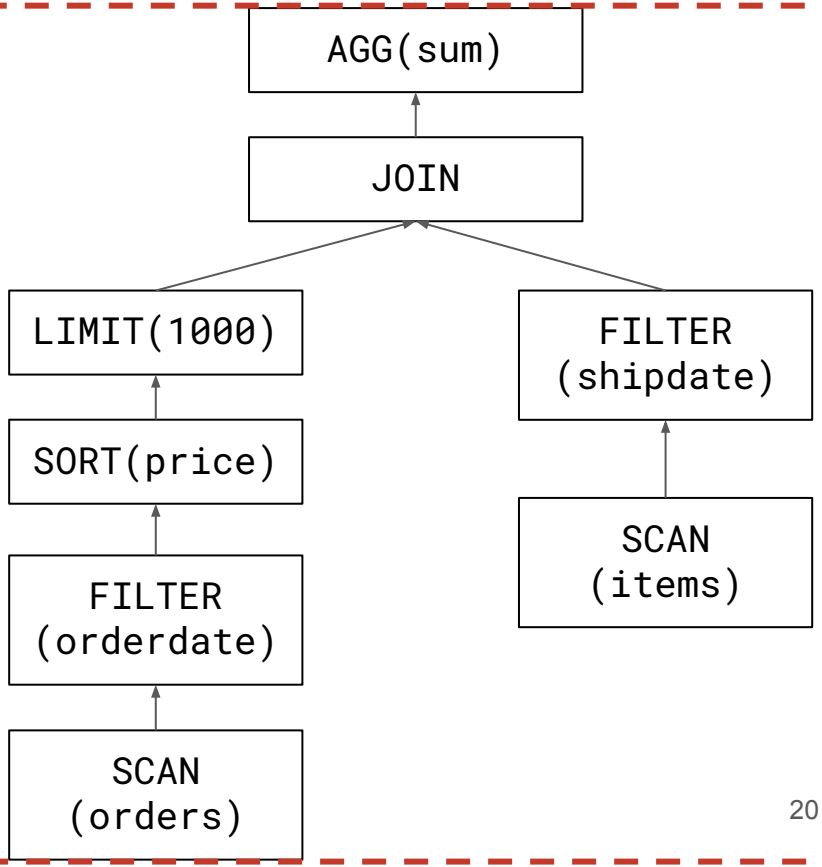
S2S Compilation Example (codegen)

```
HashMap<JoinKey, List<OrderRow>> hm =
  Stream.of(DB.orders)
        .filter(o -> o.orderdate > DATE_FROM)
        .sorted(o -> o.price)
        .limit(1000)
        .collect(groupByCollector(orderkey));

int result =
  Stream.of(DB.items)
        .filter(i -> i.shipdate < DATE_TO)
        .flatMap(i -> hm.get(i.orderkey)
                        .map(joinRows(i, o)))
        .sum(joined -> joined.price)
```



S2S Compilation Example (codegen)

```
HashMap<JoinKey, List<OrderRow>> hm =
  Stream.of(DB.orders)
        .filter(o -> o.orderdate > DATE_FROM)
        .sorted(o -> o.price)
        .limit(1000)
        .collect(groupByCollector(orderkey));

int result =
  Stream.of(DB.items)
        .filter(i -> i.shipdate < DATE_TO)
        .flatMap(i -> hm.get(i.orderkey)
                        .map(joinRows(i, o)))
        .sum(joined -> joined.price)
```



Limitations

- Main limitation: partial coverage of stream operations
  - Even if a set of queries covers all the query plan operators supported by S2S, some methods of the Java Stream API would never be called
  - Operations covered by S2S are the most relevant for data processing
  - Only few stream operations make little sense in SQL: takeWhile/dropWhile
    - Designed for ordered sequences but SQL tables are multisets

Conclusion

- We proposed a generative programming technique to create benchmarks for the Java Stream API
- We implemented S2S, a tool to convert SQL queries into Java source code
- With S2S we generated BSS, the first benchmark fully dedicated to the Java Stream API
- Thanks to S2S, many existing SQL benchmarks can be easily converted into benchmarks for the Stream API

Future Work

- Address mentioned limitation by exploring alternative conversions of SQL operators
  - E.g., mapMulti instead of flatMap
  - Performance comparison of different (equivalent) conversions
- Performance evaluation of parallel streams
- Supporting different input query languages and datasets (e.g. NoSQL/MongoDB)
- Supporting different output data-processing libraries (e.g., LINQ or Scala Collections)
- Large benchmark suite composed of many complex queries (e.g., TPC-H)

Università
della
Svizzera
italiana

Thanks!

filippo.schiavio@usi.ch

S2S (SQL compiler) github.com/usi-dag/S2S

BSS (Benchmark Suite) github.com/usi-dag/BSS

BACKUP

Java Stream API - Example

```
int sumOfSquaresEven(int[] source) {
    return IntStream.of(source)
        .filter(x -> x % 2 == 0)
        .map(x -> x * x)
        .sum();
}
```

S2S Query Compiler

- Currently supported query plan nodes (operators):
  - Table scans
  - Projections
  - Predicates
  - Limit
  - Sort
  - Joins (nested-loop-joins and hash-joins)
  - Aggregations (count, sum, avg, min/max), and grouping

S2S Query Compiler

- Current implementation based generates code by visiting the query plan:
- During the visit of each node a state is updated
  - **`type`**: The concrete type of the elements in the currently generated stream
  - **`body`**: The Java source code generated for the current stream
  - **`decl`**: A list of variables declaration and initialization
- At the end of the visit S2S emits a method body implementing the input SQL query
- The code is composed of a list of variable assignments as defined in the list **`decl`** and then the code of the last pipeline which is stored in **`body`**

S2S Compiler: Table scans

- Table scans are always the leaves in a query plan (first visited nodes)
- Each table-scan node models a loop over a single input table **T**
- Tables are implemented as arrays of classes generated with the in-memory DB
  - **`type`** is the name of the Java class for table **T**
  - **`body`** is set to the code **`Stream.of(DB.T)`**

S2S Compiler: Projections

- Projections are commonly defined by a list of SQL expressions in the SELECT clause
- By visiting a projection, S2S generates a Java class **`C`** to wrap the projected fields
- Each SQL expressions **`Si`** is converted into a Java expression **`Ji`**
  - **`type`** is set to **`C`**
  - **`body`** is concatenated with the code **`.map(row -> new C(J1, …, Jn))`**

S2S Compiler: Predicates

- Predicates are defined by the WHERE and the HAVING clauses
- The predicate is converted into a Java expression **`expr`**
  - **`type`** is unchanged (the output type is the input type)
  - **`body`** is concatenated with the code **`.filter(row -> expr)`**

S2S compiler: Limit & Sort

- The limit operator selects only a given number of rows **`n`**
  - **`body`** is concatenated with the code **`.limit(n)`**
- Visiting the sort operator, S2S generates code **`C`** defining a Java **`Comparator`** instance
  - **`body`** is concatenated with the code **`.sortec(row -> C)`**

S2S Compiler: Joins

- Join nodes have two children and are implemented with two stream pipelines
- A second visitor (**v2**) is created and used to visit the left child
- The visitor (**v**) visits the right child
- **`v.type`** is set to a freshly generated class to hold the fields of the joined tables
- **`v.decl`** is set to **`v2.decl`** and a new variable declaration that defines a data-structure to store the elements visited on the left side is appended to **`v.decl`**
- **`v.body`** update depends on the join type (nested-loop-join or hash-joins)
  - See the paper for details

S2S Compiler: Aggregations

- Aggregation is the most challenging operator for our conversion
  - A SQL aggregation is defined by a list of aggregation functions
    - E.g., **`SELECT MIN(x), AVG(y) from T`**
  - There is no interface in the Java Stream API that allows defining multiple aggregation functions
- S2S compiles aggregations with the terminal operator **`.collect`**, a mutable reduction
  - The generated code representing the collector is imperative
  - See the paper for details