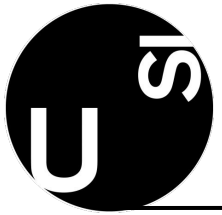
Dynamic Speculative Optimizations for SQL Execution in Apache Spark

Filippo Schiavio
USI Lugano

Daniele Bonetta
Oracle Labs

Walter Binder
USI Lugano



Apache Spark SQL

Apache Spark: de-facto standard for distributed data processing

- Spark-SQL: Spark API for processing structured data
- Can process data stored in multiple formats (e.g. JSON, CSV, …)
- Leverages code generation to optimize query execution

Code Generation in Spark SQL

SQL Query
DataFrame

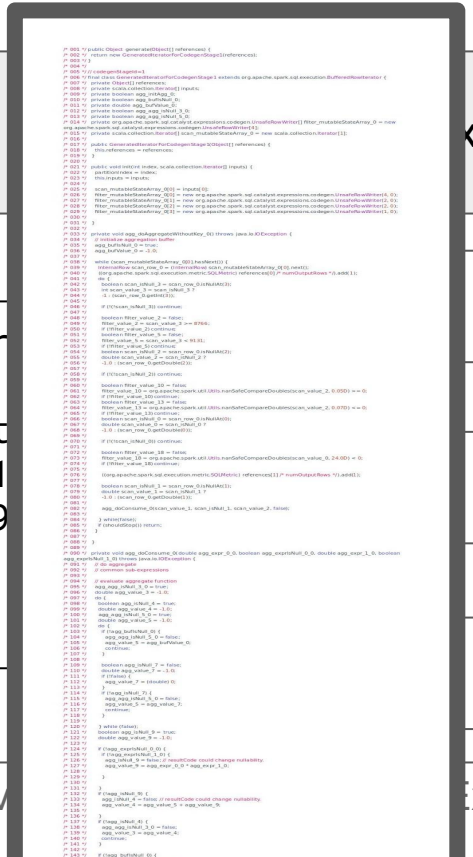
```
SELECT SUM(pr
FROM orders
WHERE shipdat
date '1994-01
AND date '199
```

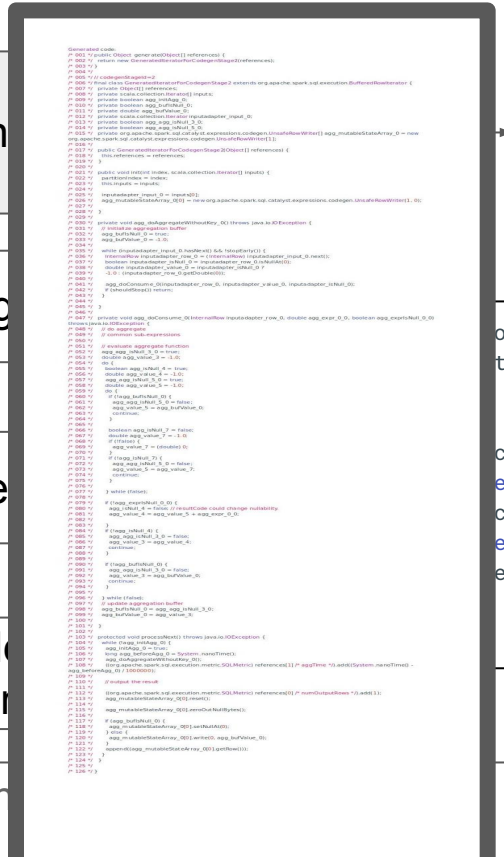
xecution

Aggreg

Filte

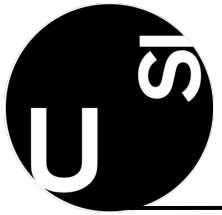
Tabl
Scar





Java Code

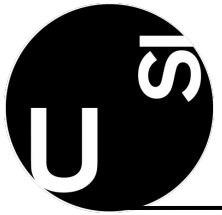
```
ompute(Data input) {
t.hasNext()) {
 input.parseNext();
 = row.getDate("shipdate");
compareTo("1994-01-01") < 0)
e;
compareTo("1994-12-31") > 0)
e;
e(row.getDouble("price"));
```



Code Generation in Spark SQL

Missing optimization opportunities: multiple data formats and modular design

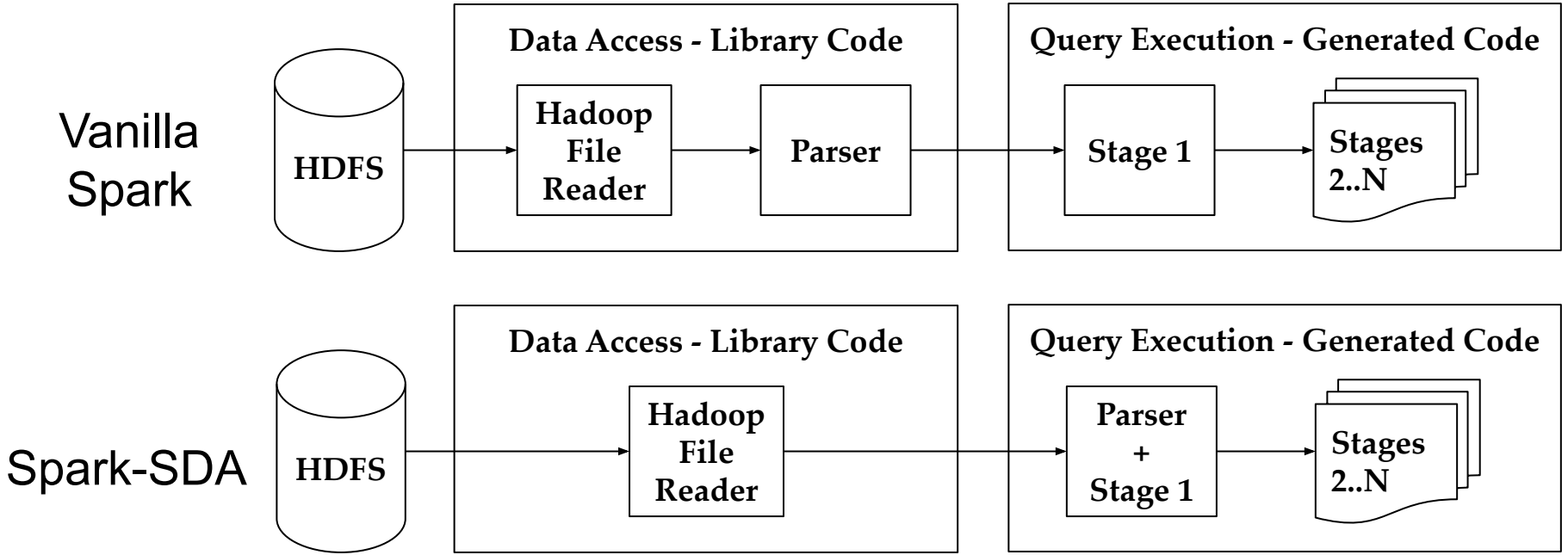
- Spark generates generic, data-format independent code
- Generality in code generation impairs performance
  - Parsing could be part of query execution
  - Predicates could be evaluated without allocating Java objects

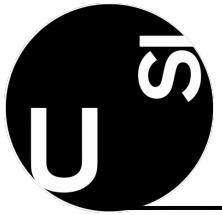


Dynamic Speculative Optimizations

- Generating code that can re-optimize itself depending on runtime conditions
- Two main optimizations:
  - Speculative specializations for data access (Spark-SDA)
  - Speculative specializations for predicate evaluation (Spark-SP)
- TPC-H speedups (up to):
  - Local mode: 8.45x (CSV); 4.9x (JSON)
  - Distributed mode: 4.4x (CSV); 2.6x (JSON)

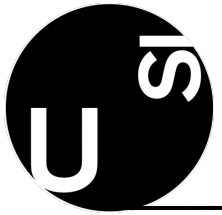
Opt 1: Specialized Data Access (Spark-SDA)





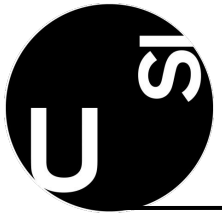
Spark-SDA (Specialized Data Access)

- Integrates a specialized parser for textual data formats: CSV and JSON
- CSV: Incremental parsing (combine parsing and query execution)
  - Skip unused fields
  - Reorder predicate evaluation according to fields' order
- JSON: Speculative incremental parsing
  - JSON values may not be declared in a specified order
  - Practically, in most of the cases they are actually ordered
  - Generated code can assume a stable order (otherwise, fallback to a generic parser)



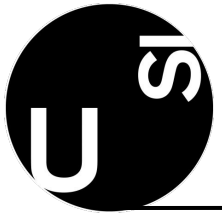
Generating Efficient Speculative Code

- Naive approach for generating speculative code
  - Add conditions that check the speculative assumption (e.g., JSON fields are ordered)
  - May introduce very high overhead if many rows do not meet the assumption
- Our approach: generating Truffle nodes instead of plain Java source code
  - Finer grained control on the compilation
  - Trigger re-compilation through deoptimization if speculative assumptions do not hold



CSV-Parsing Nodes Generation (Spark-SDA)

- Specialized CSV de-serializer can exploit static information of a specific query
  - Used fields
  - Fields declaration order
- Three main categories of generated Truffle nodes
  - Skip nodes:
    - Skip a value without performing any data conversion
  - Lazy data-access nodes:
    - Store the initial position of a field and its length
  - Data-materialization nodes:
    - Materialize a field value from the original byte array, using positions computed during lazy-data-access operation



JSON-Parsing Nodes Generation (Spark-SDA)

- JSON values may be declared in a different order, requiring a speculative approach for generating parser nodes:
  - We use the same categories of nodes described for CSV-parsing, but the generated nodes are wrapped in a new Truffle node
  - We invoke the original Spark code generator and the generated source code is wrapped in a second Truffle node
  - Nodes for lazy data-access and skip operations are extended with a guard which checks that the current field matches the expected one
  - If the matching function fails, the speculatively compiled node is de-optimized and replaced with the general node containing the code generated by Spark

Example of Generated Code (Spark-SDA)

SELECT SUM(price) FROM orders WHERE shipdate BETWEEN date '1994-01-01' AND date '1994-12-31'

CSV Schema: | id:num | price:decimal | shipdate:date | ... other fields … |

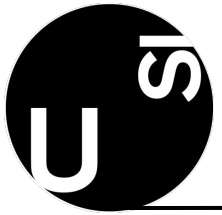
Generated by Spark

Eager Parsing

```
while (input.hasNext()) {
  Row row = input.parseNext();
  Date date = row.getDate("shipdate");
  if (date.compareTo('1994-01-01') < 0)
    continue;
  if (date.compareTo('1994-12-31') > 0)
    continue;
  accumulate(row.getDouble("price"));
}
```

Generated by Spark-SDA

```
while (input.hasNext()) {
  skip();
  int pos_price = lazyAccess();
  Date date = materialize(lazyAccess());
  if (date.compareTo('1994-01-01') < 0)
    continue;
  if (date.compareTo('1994-12-31') > 0)
    continue;
  accumulate(materialize(pos_price));
}
```



Opt 2: Specialized Predicates (Spark-SP)

- Incremental and speculative parsing in generated code allows executing predicates on raw data (e.g., directly on byte arrays)
- Predicate evaluation on raw data can leverage a speculative approach
- E.g., predicates on date fields may speculate on the expected date format

Example of Generated Code (Spark-SP)

SELECT SUM(price) FROM orders WHERE shipdate BETWEEN date '1994-01-01' AND date '1994-12-31'

CSV Schema: | id:num | price:decimal | shipdate:date | ... other fields … |

Generated by Spark-SDA

```
while (input.hasNext()) {
  skip();
  int pos_price = lazyAccess();
  Date date = materialize(lazyAccess());
  if (date.compareTo('1994-01-01') < 0)
    continue;
  if (date.compareTo('1994-12-31') > 0)
    continue;
  accumulate(materialize(pos_price));
}
```

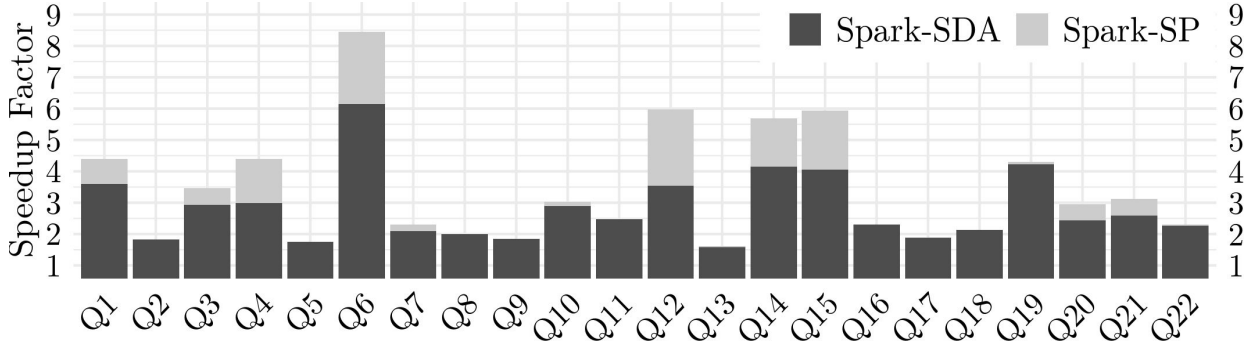
Avoidable Allocation

Generated by Spark-SP

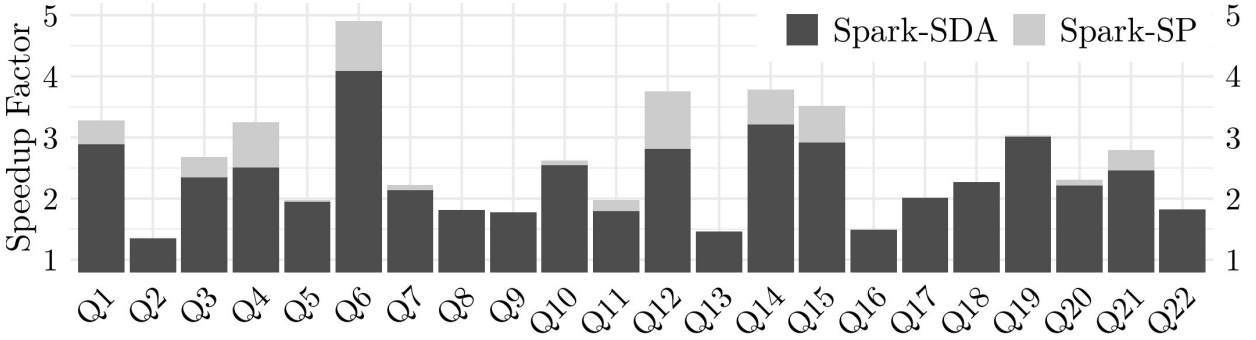
```
while (input.hasNext()) {
  skip();
  int pos_price = lazyAccess();
  int pos_date = lazyAccess();
  cursor = datePredicate(pos_date);
  if(cursor == -1)
    continue;
  accumulate(materialize(pos_price));
}
```

Performance Evaluation (TPC-H)

CSV Dataset
(Scale Factor 30)

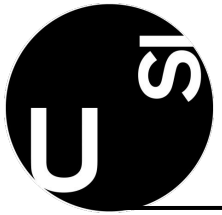


JSON Dataset
(Scale Factor 10)



Setup:

- Spark 2.4 (local mode)
- Machine:
  - 8 cores, 2.7GHz
  - 128GB RAM



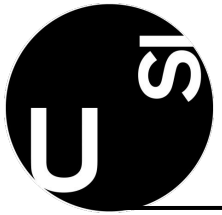
Memory Pressure (TPC-H Q6)

CSV Dataset
(Scale Factor 10)

Implementation	Heap Used Total Memory for TLABs	# Garbage Collection Invocations
Spark	111 GB	255
Spark-SDA	13 GB	12
Spark-SP	600 MB	3

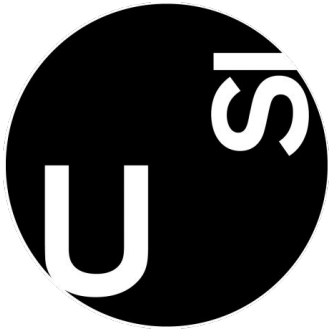
Setup:

- Spark 2.4 (local mode)
- Machine:
  - 8 cores, 2.7GHz
  - 128GB RAM



Limitations and Future Work

- Predicate evaluation order depends on fields declaration order
  - Intuition: parsing is an expensive operation, evaluating predicates ASAP may reduce such cost
  - Depending on predicates evaluation cost, selectivities, and the cost of parsing other fields, postponing a predicate may be more efficient
- Future work
  - Runtime predicate reordering through profiling and re-compilation
  - Applying similar data-processing optimizations to existing Truffle languages



Thanks!

Filippo Schiavio
filippo.schiavio@usi.ch

F. Schiavio, D. Bonetta and W. Binder, Dynamic Speculative Optimizations for SQL Compilation in Apache Spark, PVLDB 2020