SPARK SQL: RELATIONAL DATA PROCESSING IN SPARK

Michael Armbrust, Reynold Xin, Cheng Lian, Yin Huai,
Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, Matei Zaharia

Presented by
Mohamad Dolatshah

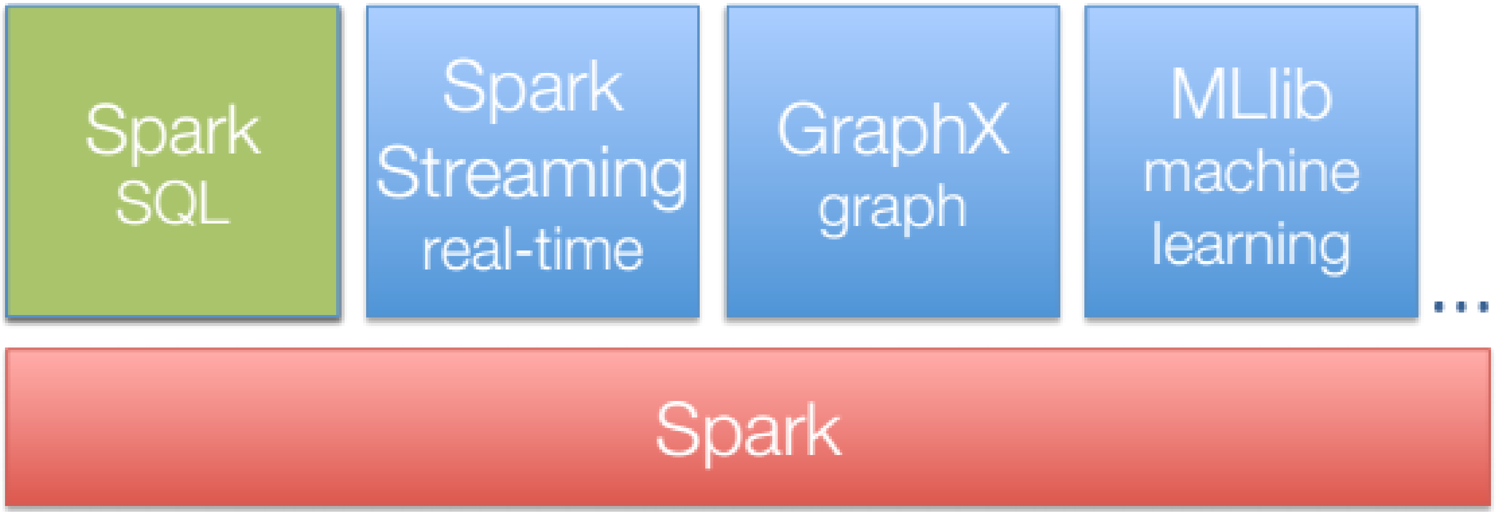
Spark SQL

Background

Apache Spark

- Fast and general cluster computing system
- Improves efficiency through: → Up to 100× faster (2-10× on disk)
  - *In-memory computing primitives*
- Improves usability through: → 2-5× less code
  - *Rich APIs in Scala, Java, Python*
  - *Interactive shell*

A General Stack



Spark SQL

About Spark SQL

- Spark SQL
  - *Part of the core distribution since April 2014.*
  - *Runs SQL / HiveQL queries, optionally* ***alongside*** *or* ***replacing*** *existing* ***Hive deployments****.*

```
SELECT COUNT(*)
FROM hiveTable
WHERE hive_udf(data)
```

Improvement upon Existing Art

Spark

- Engine does not understand the structure of the data in RDDs or the semantics of user functions → limited optimization.

SHARK

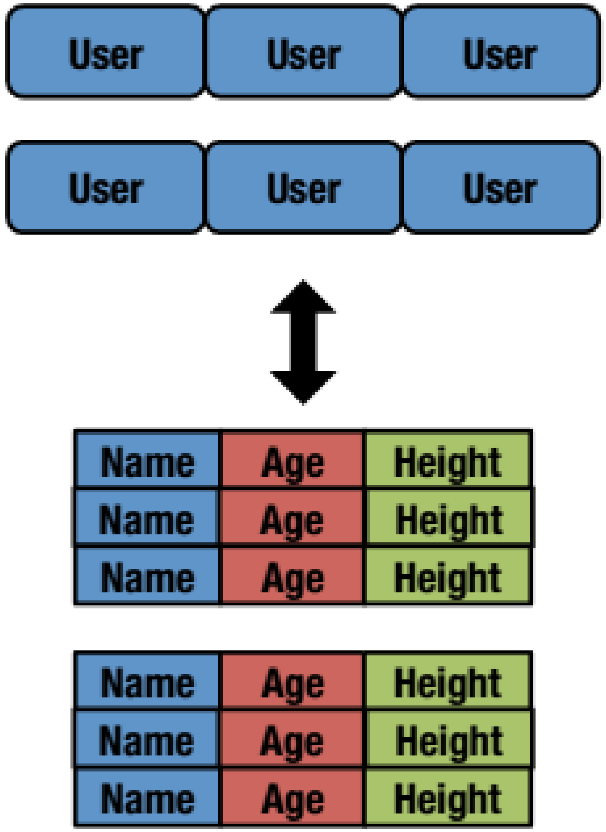
- Can only be used to query external data in Hive catalog → limited data sources
- Can only be invoked via SQL string from Spark→ error prone
- Hive optimizer tailored for MapReduce → difficult

Data Model

- Nested data model
- Supports
    - *Primitive SQL types (boolean, integer, double, decimal, string, data, timestamp)*
    - *Complex types (structs, arrays, maps, and unions)*
    - *User defined types.*

SchemaRDD’s as a Key Concept

- RDD: “Immutable partitioned collection of elements”.
- SchemaRDD: “An RDD of Row objects that has an associated schema”.



Advantages over RDDs

- They express the *how* of a solution better than the *what.*
- They cannot be optimized by Spark.
- They're *slow* on non-JVM languages like Python.
- It's too easy to build an inefficient RDD transformation chain.

```
parsedRDD.reduceByKey(_ + _).                                    <--- INEFFICIENT
          filter { case (project, numRequest) => project == "en" }. <--- ORDERING
          take(100).
          foreach { case (project, requests) => println(s"project: $requests") }
```

Advantages over Relational Query Languages

- Holistic optimization across functions composed in different languages.
- Control structures (e.g. *if*, *for*)
- Logical plan analyzed *eagerly*
  - *Identify code errors associated with data schema issues on the fly.*

DataFrame Operations

- Relational operations (select, where, join, groupBy) via a DSL.
- Operators take *expression* objects.
- Operators build up an abstract syntax tree (AST), which is then optimized by *Catalyst*.

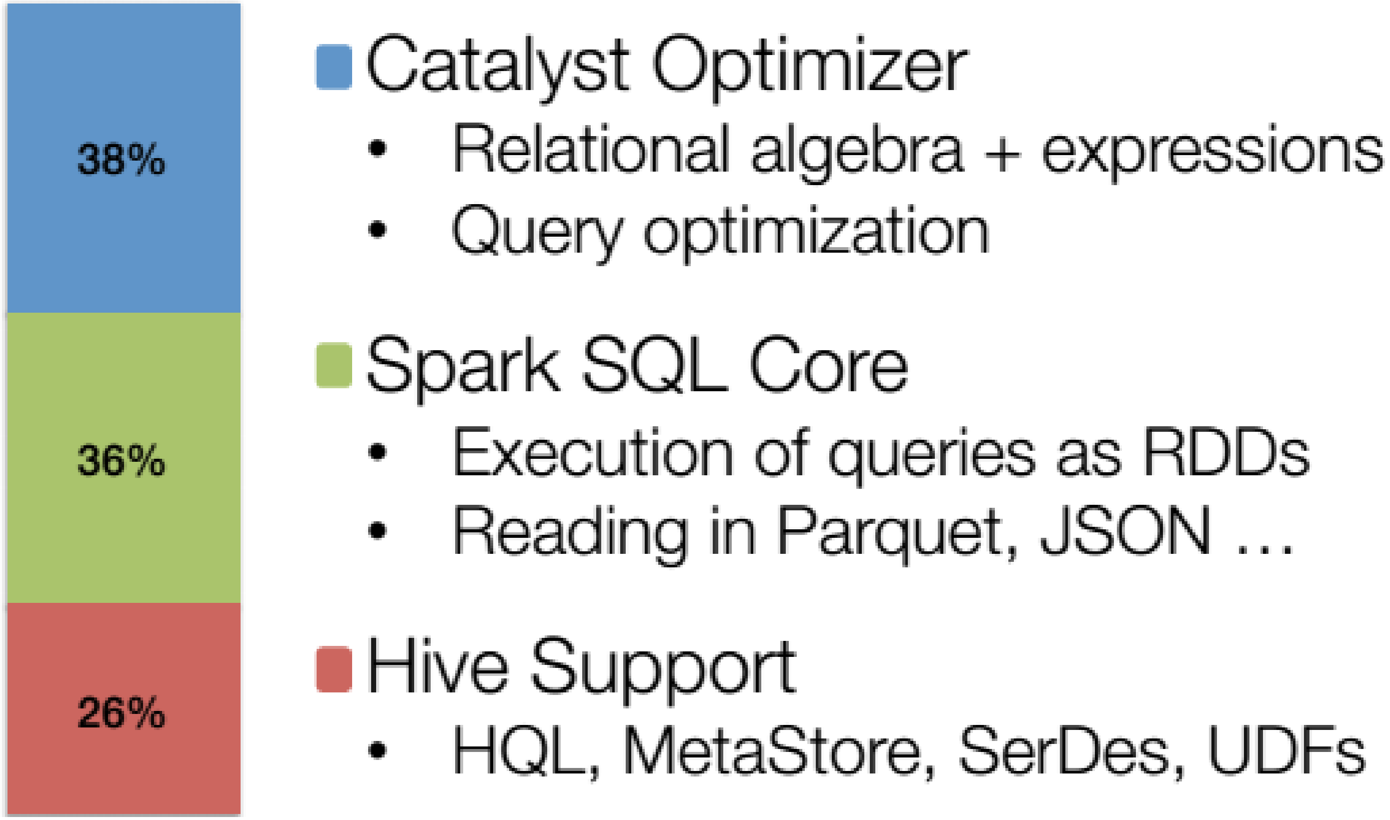
```
employees
  .join(dept, employees("deptId") === dept("id"))
  .where(employees("gender") === "female")
  .groupBy(dept("id"), dept("name"))
  .agg(count("name"))
```

User-Defined Functions (UDFs)

- Easy extension of limited operations supported.
- Allows inline registration of UDFs.
- Can be defined on simple data types or entire tables.
- UDFs available to other interfaces after registration

```
val model: LogisticRegressionModel = ...

ctx.udf.register("predict",
  (x: Float, y: Float) => model.predict(Vector(x, y)))

ctx.sql("SELECT predict(age, weight) FROM users")
```

Spark SQL Components



Optimizing Queries with Catalyst

What is Query Optimization?

- SQL is a declarative language:
  - *Queries express what data to retrieve.*
  - *Not how to retrieve it.*

- The database must pick the ‘best’ execution strategy through a process known as optimization.

Prior Work: Optimizer Generators

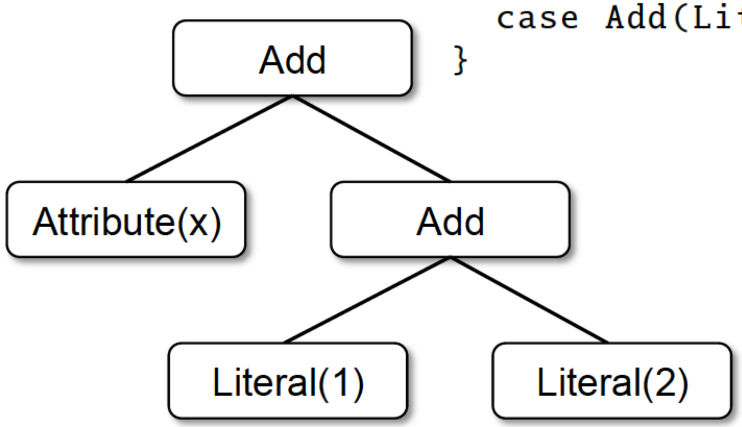
Volcano / Cascades:

- Create a **custom language** for expressing **rules** that rewrite **trees of relational operators**.
- Build a **compiler** that generates **executable** code for these rules.

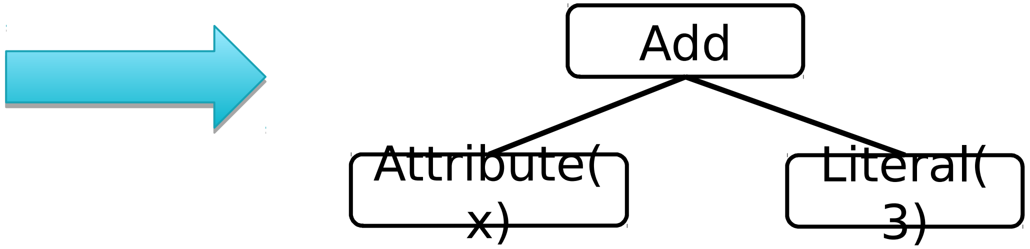
Cons: Developers need to learn this custom language. Language might not be powerful enough.

Catalyst

```
tree.transform {
  case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
}
```



x + (1 + 2)



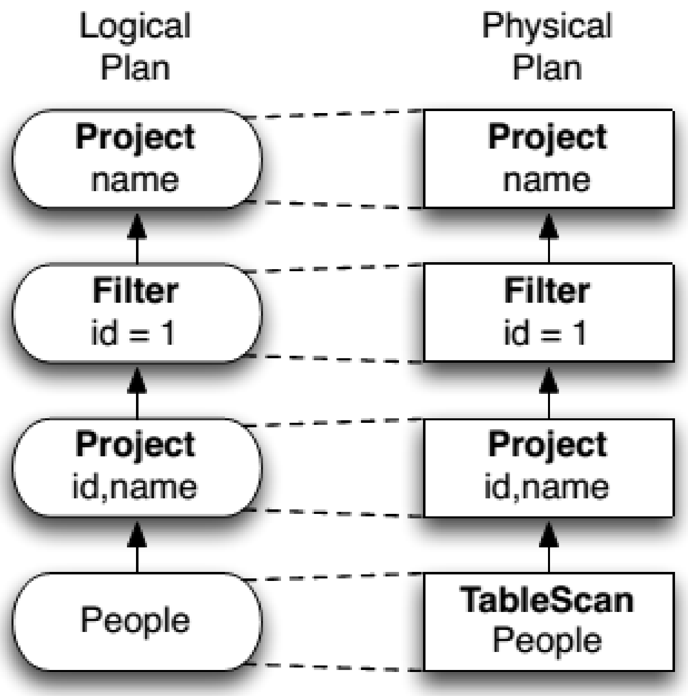
x + 3

Catalyst Rules

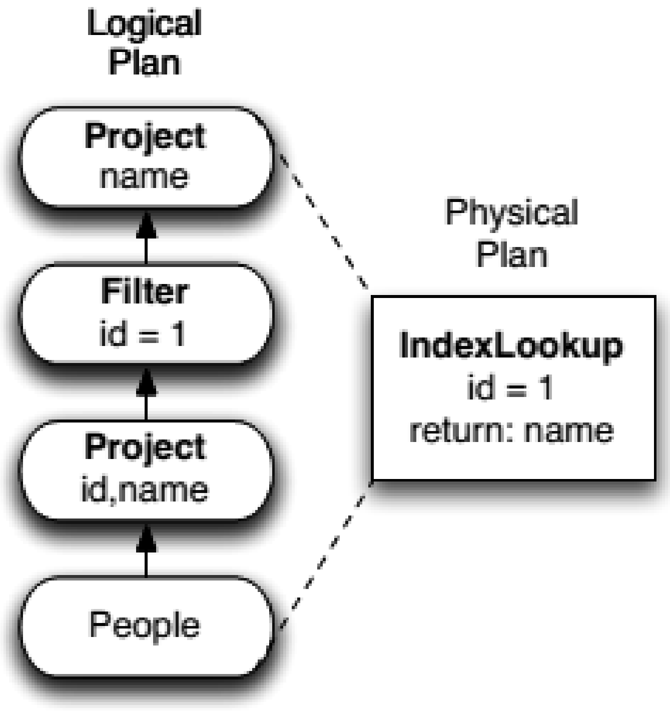
- *Pattern matching* **functions** that transform subtrees into specific structures.
  - *Partial function - skip over subtrees that do not match.*
  - *No need to modify existing rules when adding new types of operators.*
- Multiple patterns in the same *transform* call.
- May take multiple *batches* to reach a *fixed point*.

Naïve Query Planning

```
SELECT name
FROM (
     SELECT id, name
     FROM People) p
WHERE p.id = 1
```



Optimized Execution

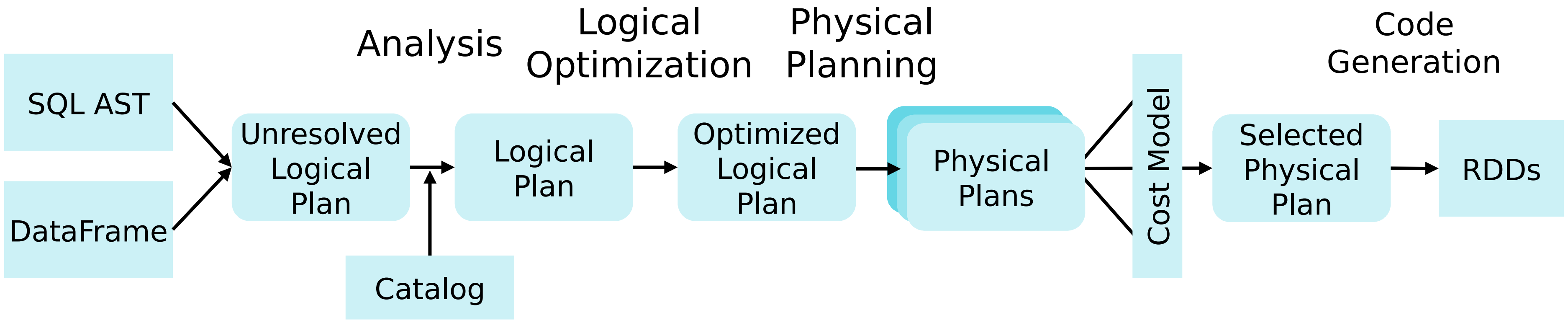


Writing imperative code to optimize all possible Project patterns is hard.

Instead write simple rules:

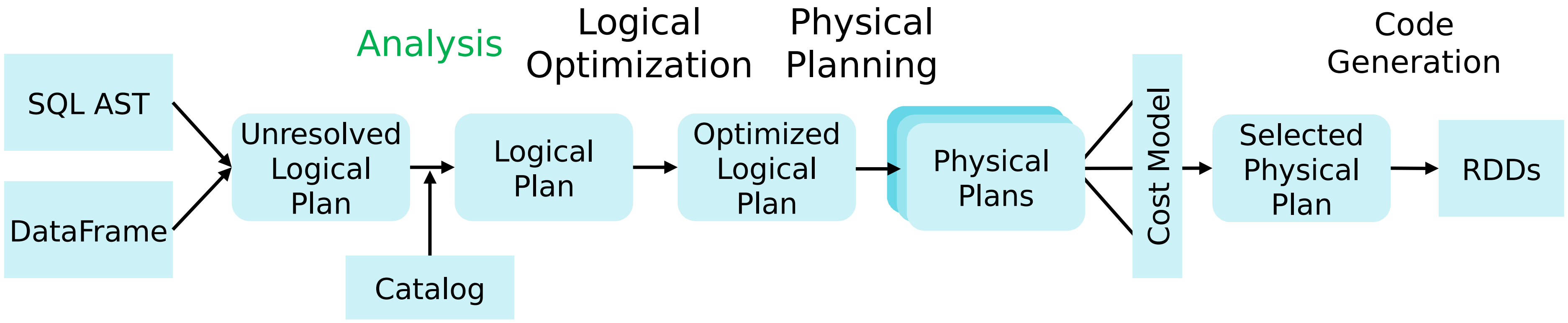
- Each rule makes one change
- Run many rules together to fixed point.

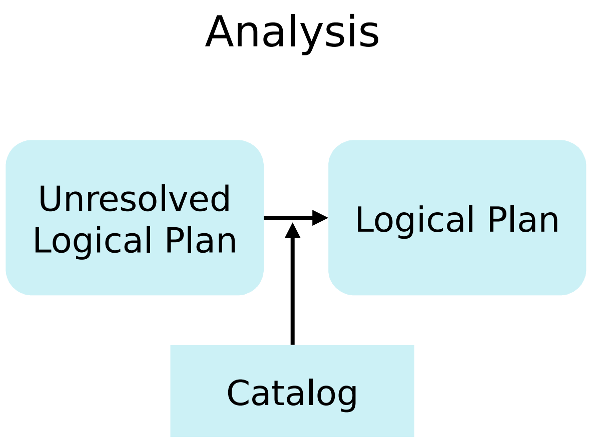
Plan Optimization & Execution



DataFrames and SQL share the same optimization/execution pipeline

Plan Optimization & Execution



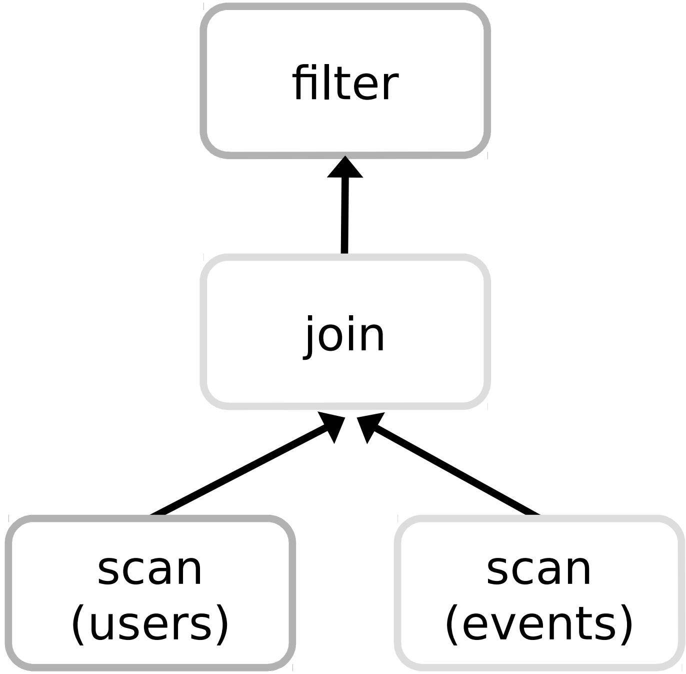


SELECT *col* FROM *sales*

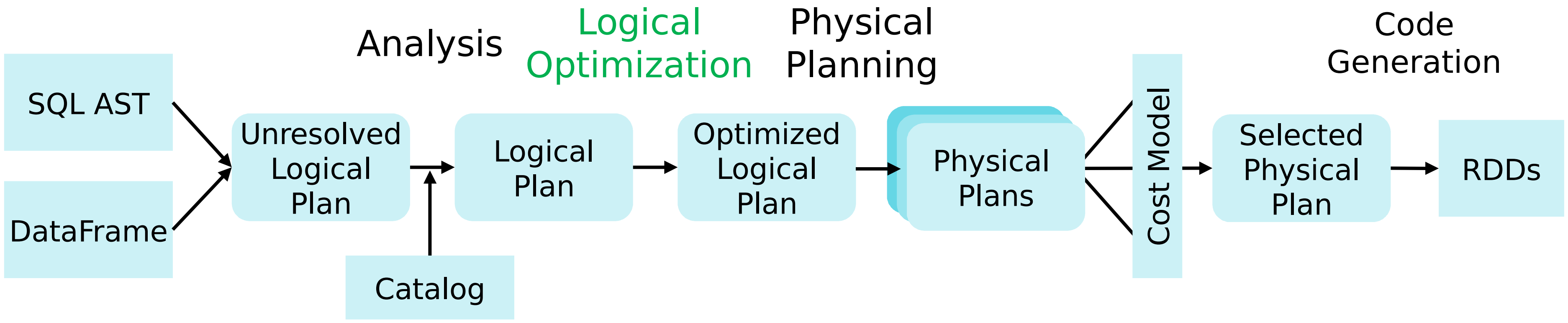
- An attribute is *unresolved* if its type is not known or it's not matched to an input table.
- To resolve attributes:
  - *Look up relations by name from the catalog.*
  - *Map named attributes (col) to the input provided given operator's children.*
  - *UID for references to the same value.*
  - *Propagate and coerce types through expressions (e.g. 1 + col).*

```
users.join(events, users("id") === events("uid"))
     .filter(events("date") > "2015-01-01")
```

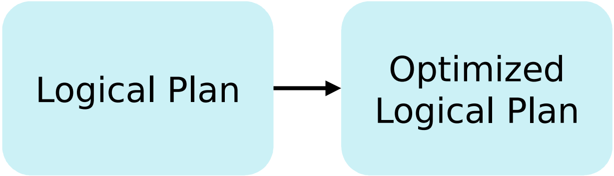
Logical Plan



Plan Optimization & Execution



Logical Optimization

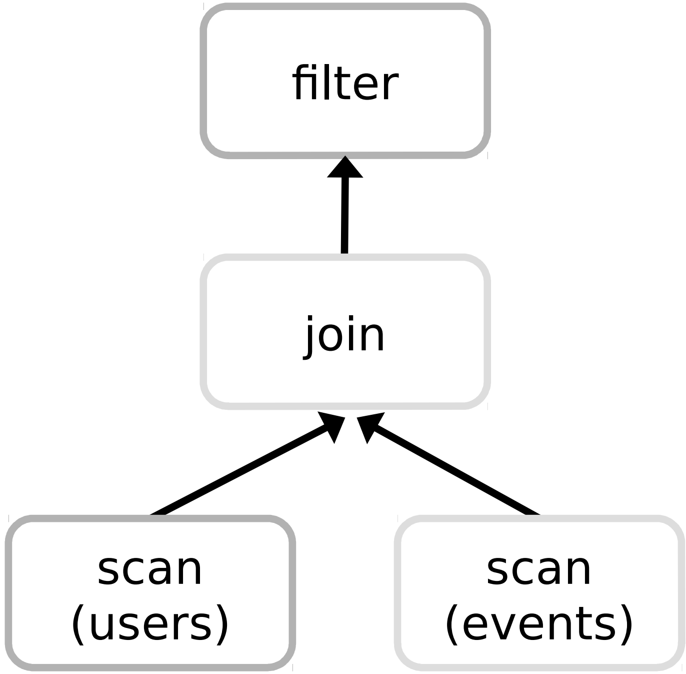


- Applies standard **rule-based optimization** (constant folding, predicate-pushdown, boolean expression simplification, etc)
- 800LOC

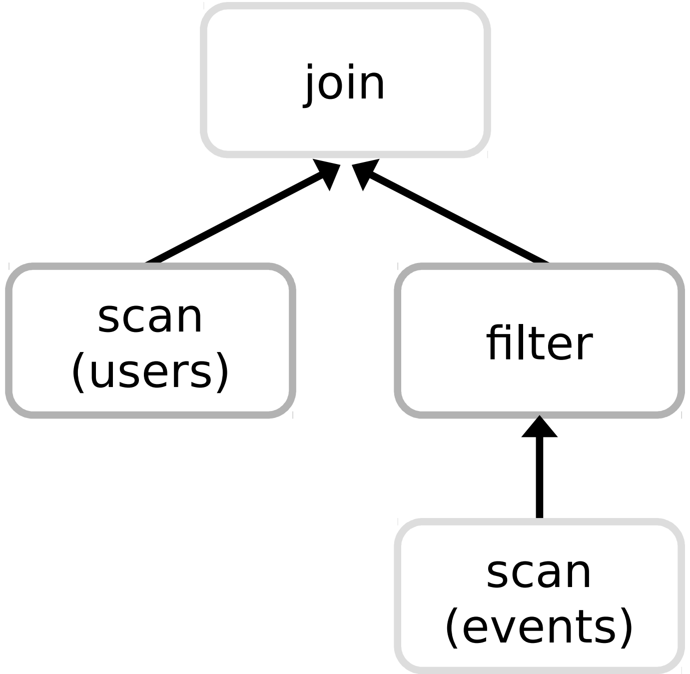
```
object DecimalAggregates extends Rule[LogicalPlan] {
  /** Maximum number of decimal digits in a Long */
  val MAX_LONG_DIGITS = 18

  def apply(plan: LogicalPlan): LogicalPlan = {
    plan transformAllExpressions {
      case Sum(e @ DecimalType.Expression(prec, scale))
              if prec + 10 <= MAX_LONG_DIGITS =>
        MakeDecimal(Sum(LongValue(e)), prec + 10, scale)
    }
}
```

```
users.join(events, users("id") === events("uid"))
     .filter(events("date") > "2015-01-01")
```

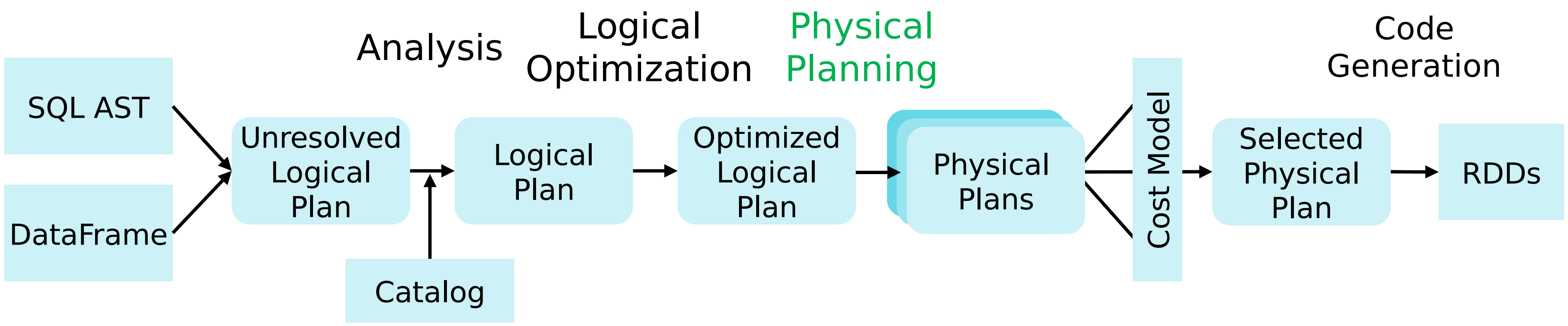
Logical Plan



Optimized plan

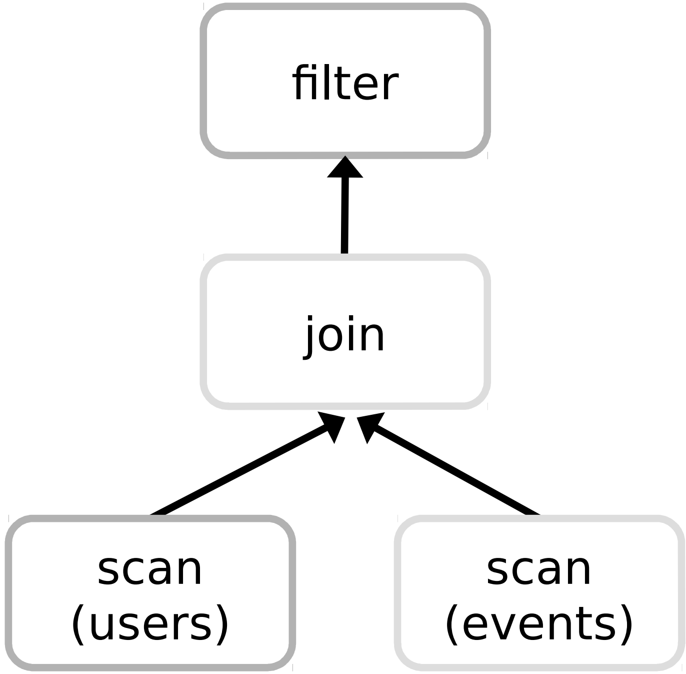


Plan Optimization & Execution

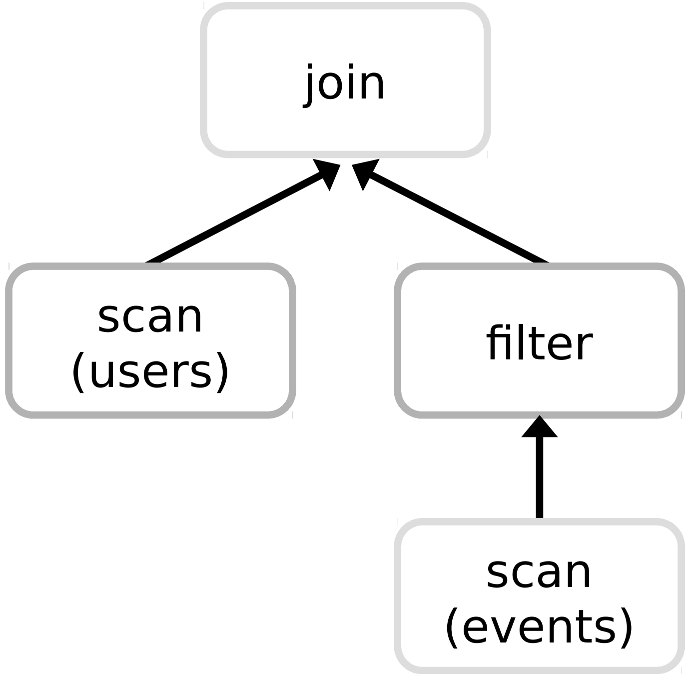


```
users.join(events, users("id") === events("uid"))
     .filter(events("date") > "2015-01-01")
```

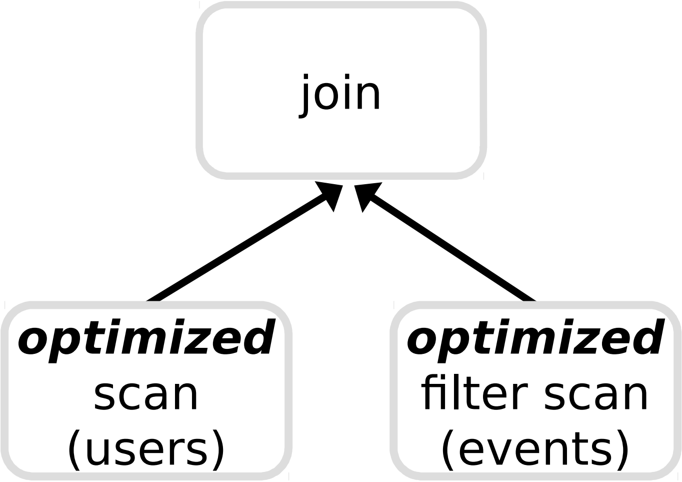
Logical Plan



Optimized plan



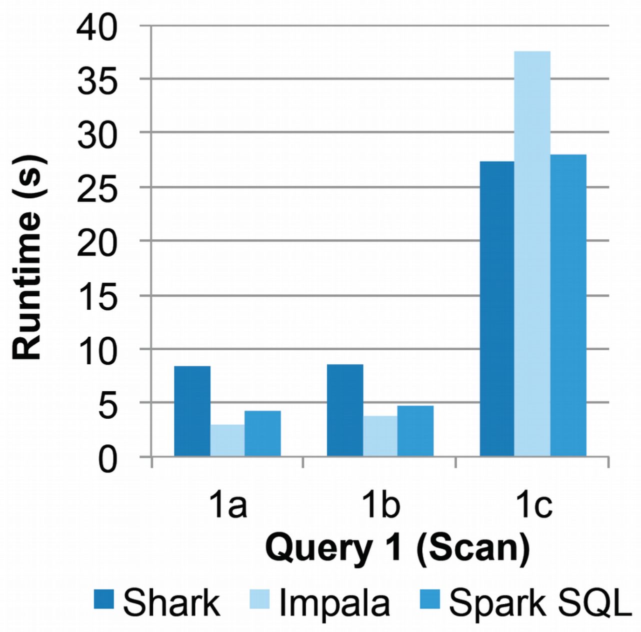
Physical Plan
with Predicate Pushdown and Column Pruning

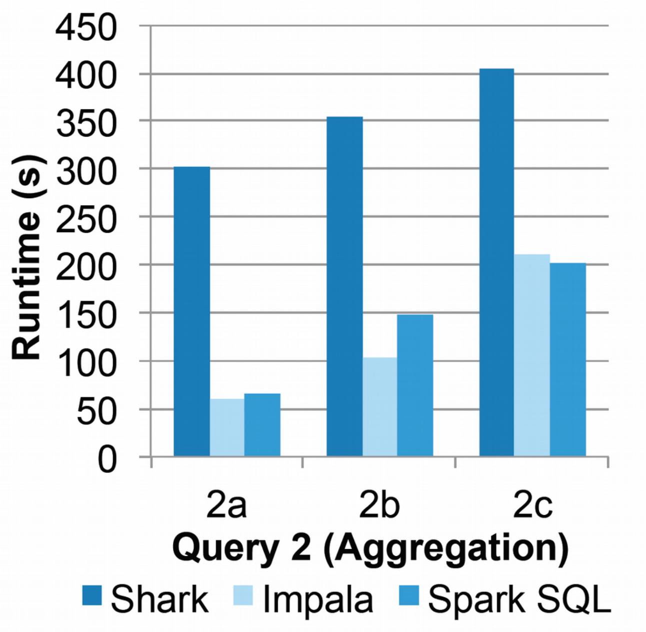


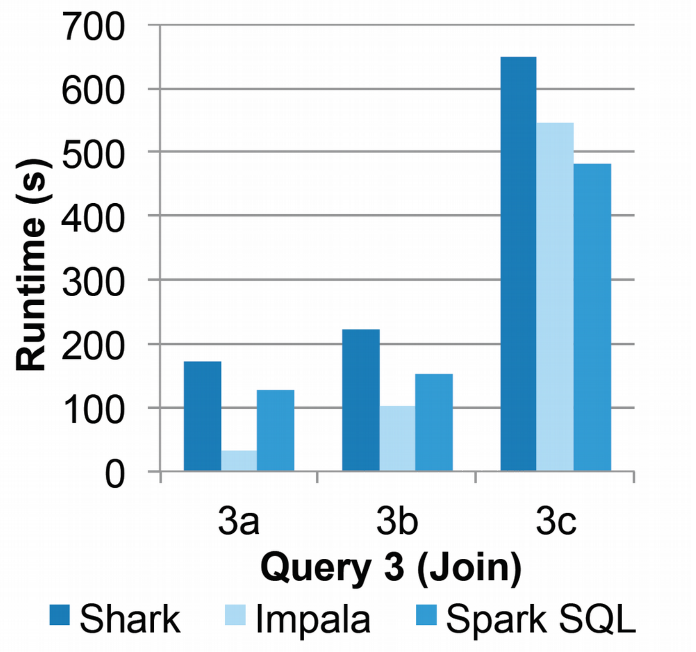
Performance Comparison

- 110GB of data after columnar compression with Parquet
- Cluster of six machines each with 4 cores, 30 GB memory
- 800 GB SSD, running HDFS 2.4, Spark 1.3, Shark 0.9.1 and Impala

- Query 1a, 2a, 3a: The most selective
- Query 1c, 2c, 3c: The least selective and processing more data.







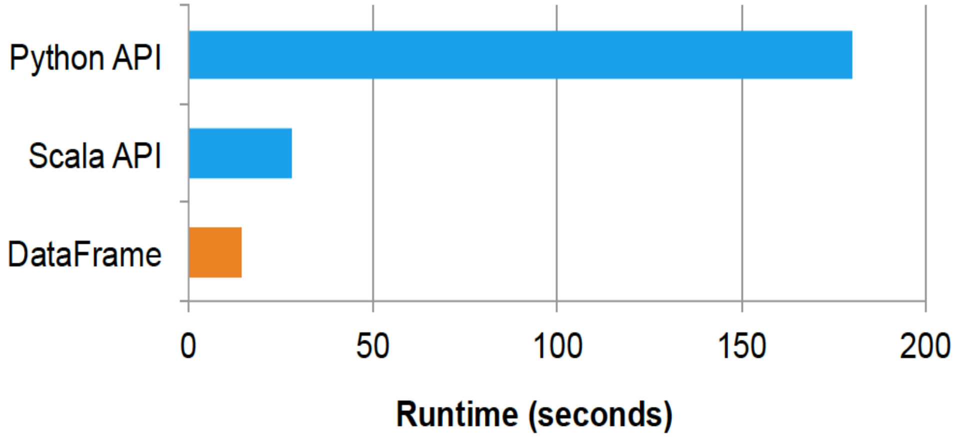
Performance Comparison (cont'd)

The dataset consists of:

- 1 billion integer pairs, *(a, b)* with 100,000 distinct values of *a* on the same five-worker cluster.
- Measure the time taken to compute the average of b for each value of a.

```
sum_and_count = \
  data.map(lambda x: (x.a, (x.b, 1))) \
      .reduceByKey(lambda x, y: (x[0]+y[0], x[1]+y[1])) \
      .collect()
[(x[0], x[1][0] / x[1][1]) for x in sum_and_count]
```

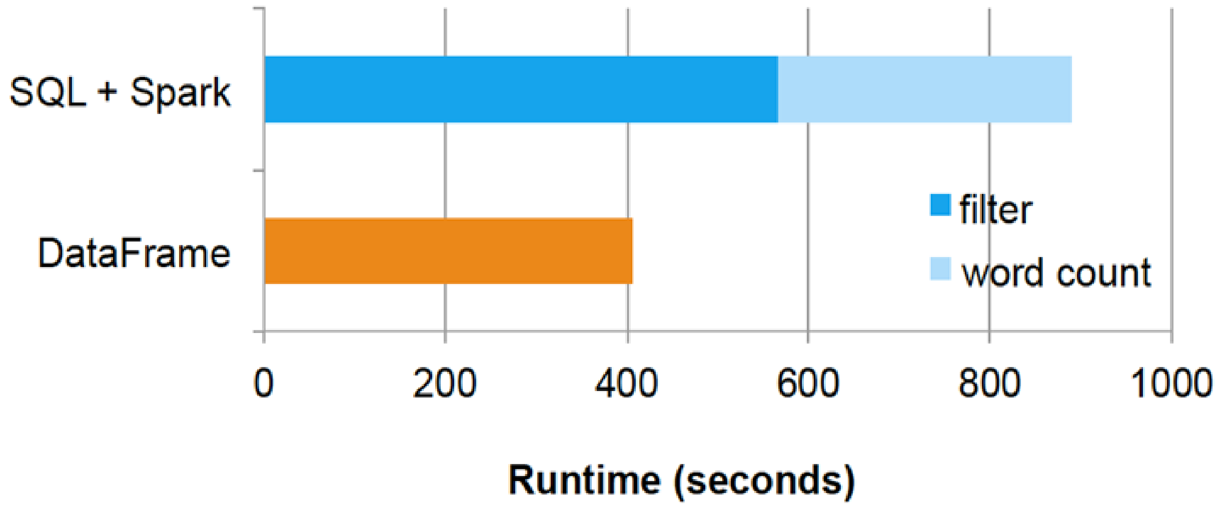
```
df.groupBy("a").avg("b")
```



Performance Comparison (cont’d)

Synthetic dataset of 10 billion messages in HDFS
Each message contained 10 words

- The first stage of the pipeline uses a relational filter to select 90% of the messages.
- The second stage computes the word count.



Summary

■Challenges

- Perform ETL to and from various (semi- or unstructured) data sources.
- Perform advanced analytics (e.g. machine learning, graph processing) that are hard to express in relational systems.

■Solutions

- A *DataFrame* API that can perform relational operations on both external data sources and Spark's built-in RDDs.
- A highly extensible optimizer, *Catalyst*, that uses features of Scala to add composable rule, control code gen., and define extensions.

Future Work: Project Tungsten

- Improving the efficiency of *memory and CPU* for Spark applications, to push performance closer to the limits of modern hardware.

Project Tungsten: Bringing Apache Spark Closer to Bare Metal

■ Overcome JVM limitations:

- **Memory Management and Binary Processing**: leveraging application semantics to manage memory explicitly and eliminate the overhead of JVM object model and garbage collection.
- **Cache-aware computation**: algorithms and data structures to exploit memory hierarchy.
- **Code generation**: using code generation to exploit modern compilers and CPUs.

Apache Spark is already pretty fast, but can we make it 10x faster?

- Rethink about Spark’s physical execution layer.
- Majority of the CPU cycles are spent in useless work.
  - *Making virtual function calls.*
  - *Reading or writing intermediate data to CPU cache or memory.*
- Collapses the entire query into a single function.
  - *Eliminating virtual function calls.*
  - *Leveraging CPU registers for intermediate data .*
  - *“whole-stage code generation”*

“whole-stage code generation”

- In the past
  - *Spark only applied code generation to **expression evaluation** and was limited to a **small number of operators** (e.g. Project, Filter).*

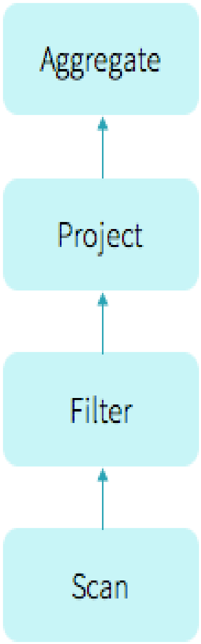
x + (1 + 2)

- Today
  - *Generate code for the entire query plan.*

The Past: Volcano Iterator Model

Spark leveraged a popular classic query evaluation strategy based on an iterator model (commonly referred to as the Volcano model).

```
select count(*) from store_sales
where ss_item_sk = 1000
```




```
class Filter(child: Operator, predicate: (Row => Boolean))
  extends Operator {
  def next(): Row = {
    var current = child.next()
    while (current == null || predicate(current)) {
      current = child.next()
    }
    return current
  }
}
```

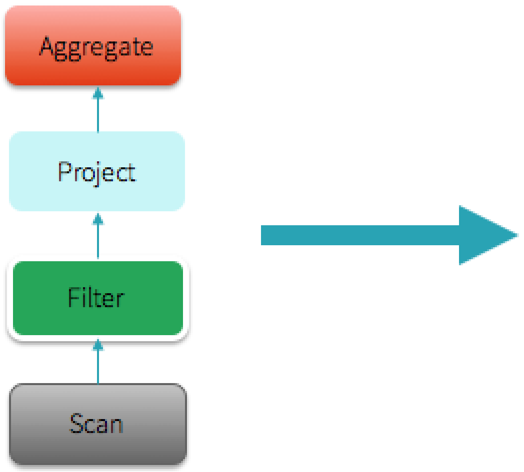
Hand-written code is faster than the Volcano model

- No virtual function dispatches.
- Intermediate data in memory vs CPU registers.
- Loop unrolling and SIMD.

```
var count = 0
for (ss_item_sk in store_sales) {
  if (ss_item_sk == 1000) {
    count += 1
  }
}
```

Whole-stage code generation

- The engine can achieve the performance of hand-written code, yet provide the functionality of a general purpose engine.
- Rather than relying on operators for processing data at runtime:
    - *Collapse each fragment of the query into a single function and execute that generated code instead.*




```
long count = 0;
for (ss_item_sk in store_sales) {
  if (ss_item_sk == 1000) {
    count += 1;
  }
}
```

Thanks for your attention

Questions?