Spark SQL: A compiler from queries to RDDs

Wenchen Fan

SDCC 2016

databricks™

Agenda

- **Why Spark SQL?**
- **The Frontend: Catalyst**
- **The Backend**
- **The Tungsten Project**
- **Benchmark**
- **What’s next**

Background: What is an RDD?

- Dependencies
- Partitions
- Compute function: Partition => Iterator[T]

databricks

Background: What is an RDD?

- Dependencies
- Partitions
- Compute function: $\underbrace{\text{Partition => Iterator[T]}}_{}$

Opaque Computation

databricks

Background: What is an RDD?

- Dependencies
- Partitions
- Compute function: Partition => $\underbrace{\text{Iterator}}_{\text{Opaque Data}}$[T]

RDD Programming Model

Construct execution DAG using low level RDD operators.

```
pdata.map(lambda x: (x.dept, [x.age, 1])) \
   .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
   .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
   .collect()
```

Spark SQL Come to Rescue

- **More efficient**: Only process structural data, this limits what can be expressed but enables optimization.

Spark SQL Come to Rescue

- **More efficient**: Only process structural data, this limits what can be expressed but enables optimization.
- **High-level API**: SQL, DataFrame/Dataset

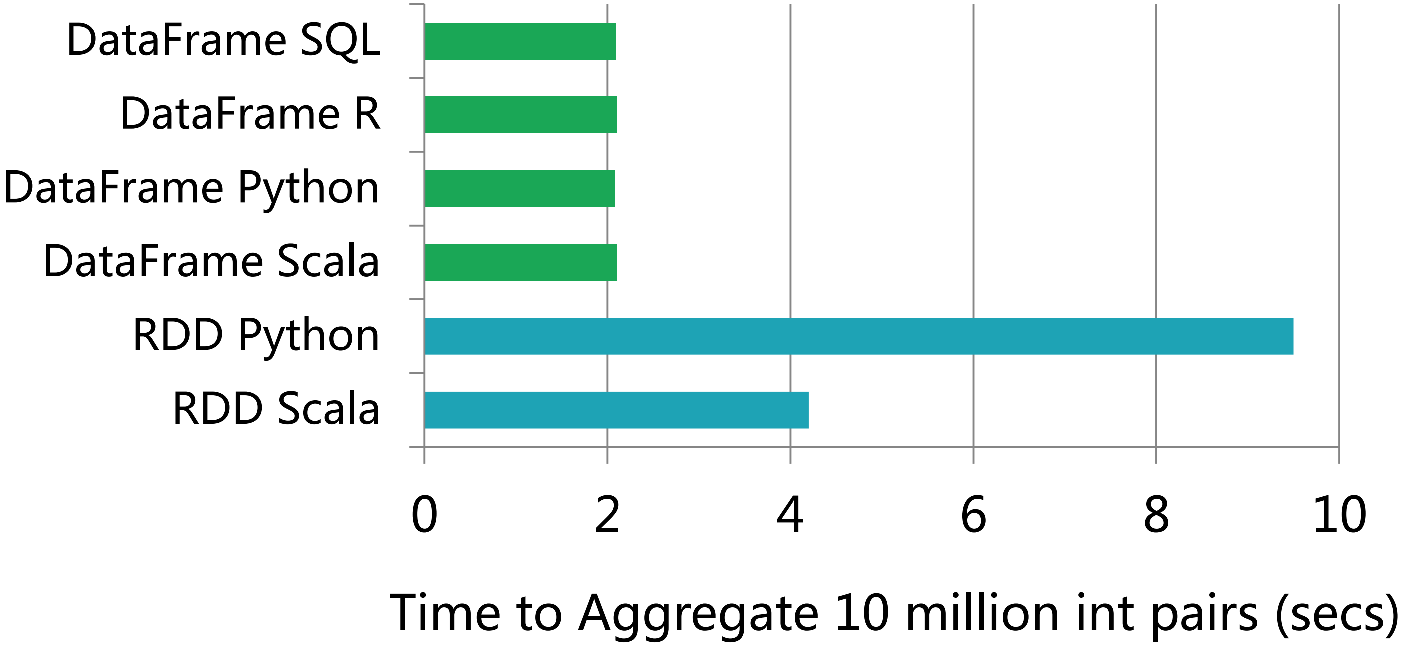
Write less code

```
pdata.map(lambda x: (x.dept, [x.age, 1])) \
   .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
   .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
   .collect()
```

```
SELECT dept, AVG(age) FROM pdata GROUP BY dept
```

```
pData.groupBy("dept").agg(avg("age"))
```

Not Just Less Code, Faster Too!



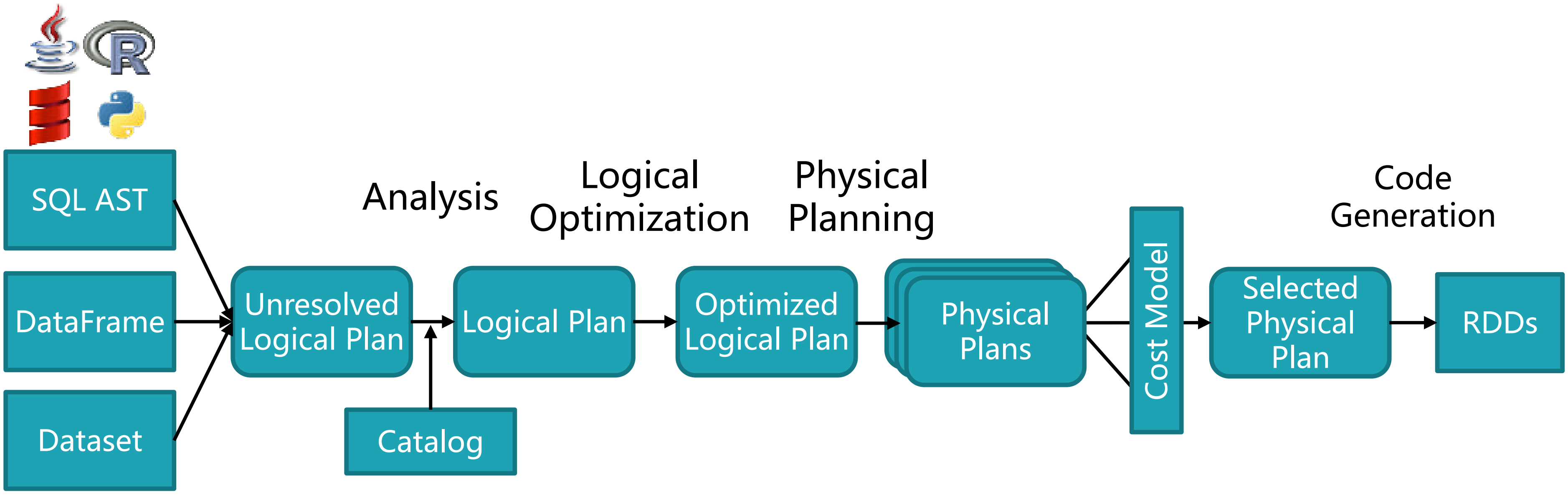
databricks

The not-so-secret truth…

Spark SQL

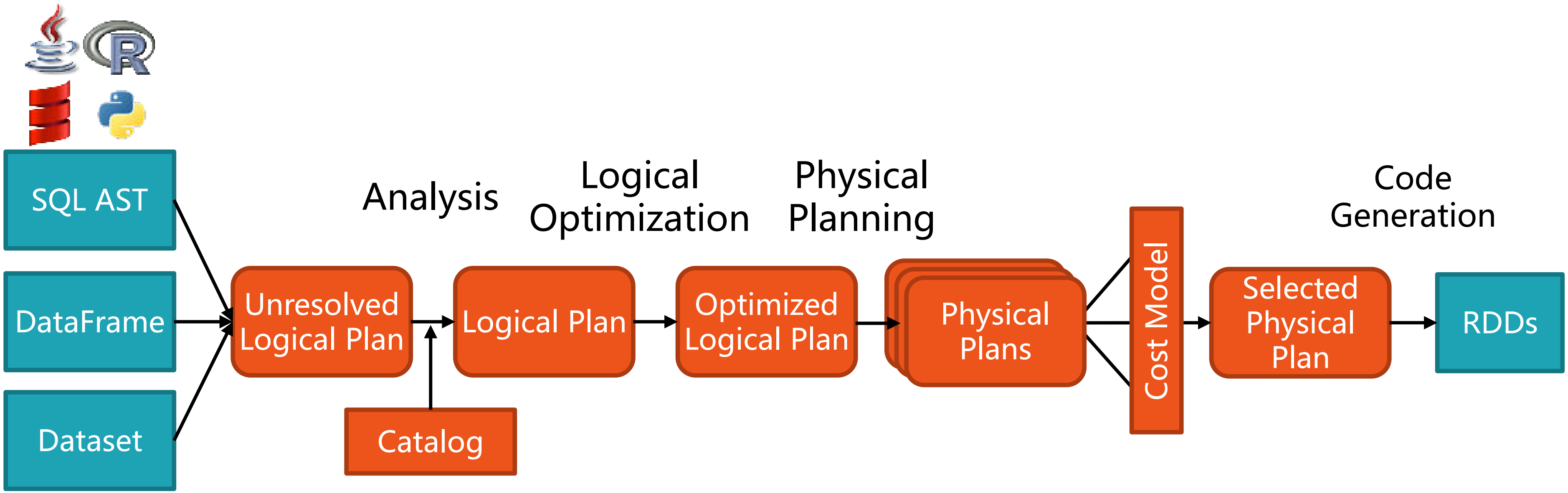
is about more than SQL.

Spark SQL Overview

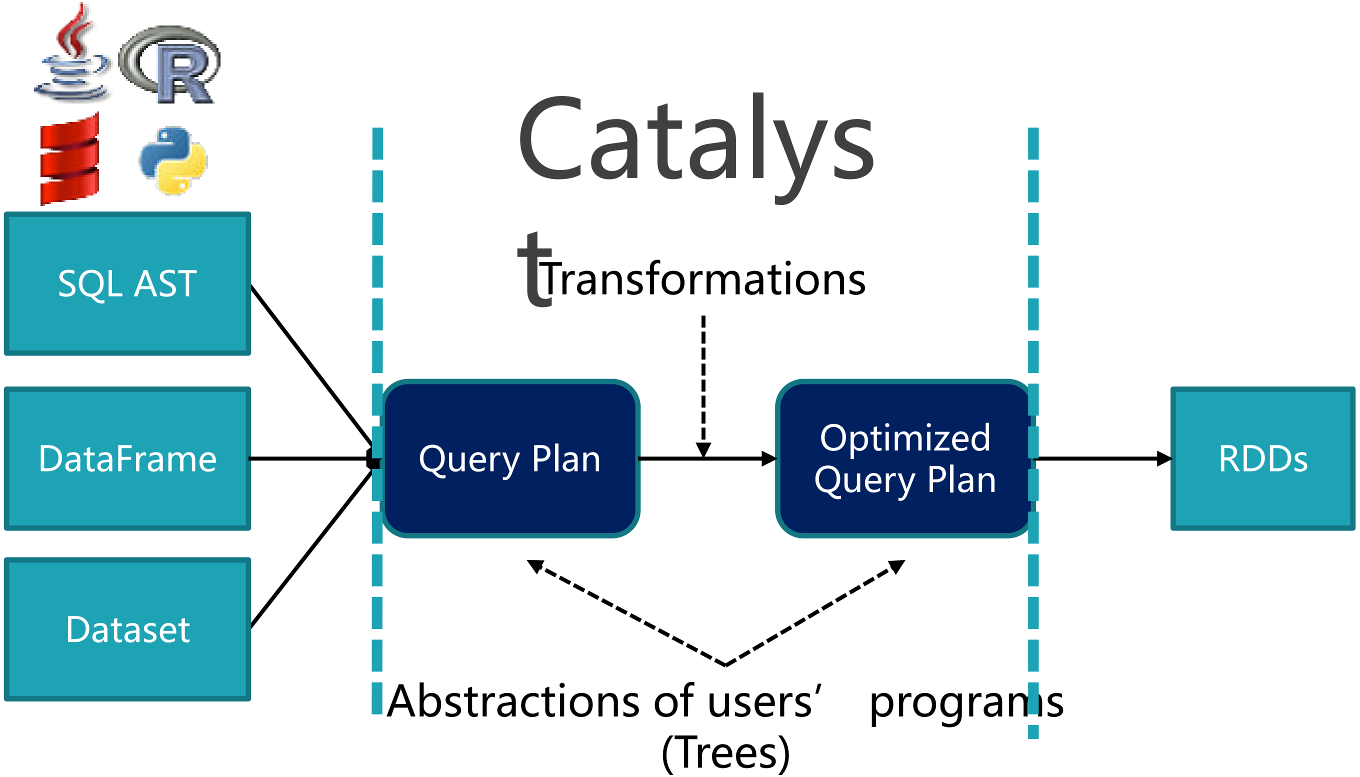


DataFrames, Datasets and SQL
share the same optimization/execution pipeline

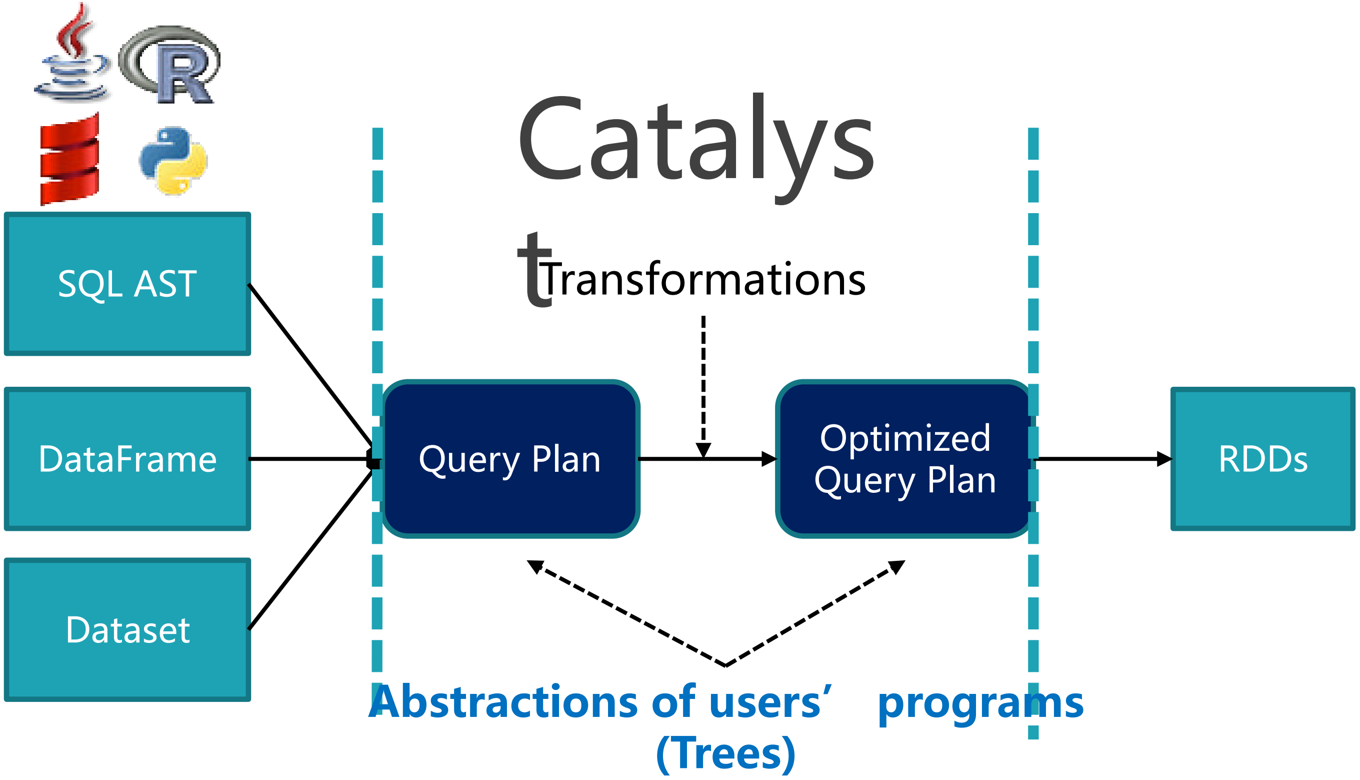
Catalyst: The frontend



How Catalyst Works: An Overview



How Catalyst Works: An Overview



Trees: Abstractions of Users' Programs

```
SELECT sum(v)
FROM (
  SELECT
    t1.id,
    1 + 2 + t1.value AS v
  FROM t1 JOIN t2
  WHERE
    t1.id = t2.id AND
    t2.id > 50 * 1000) tmp
```

Trees: Abstractions of Users' Programs

Expression

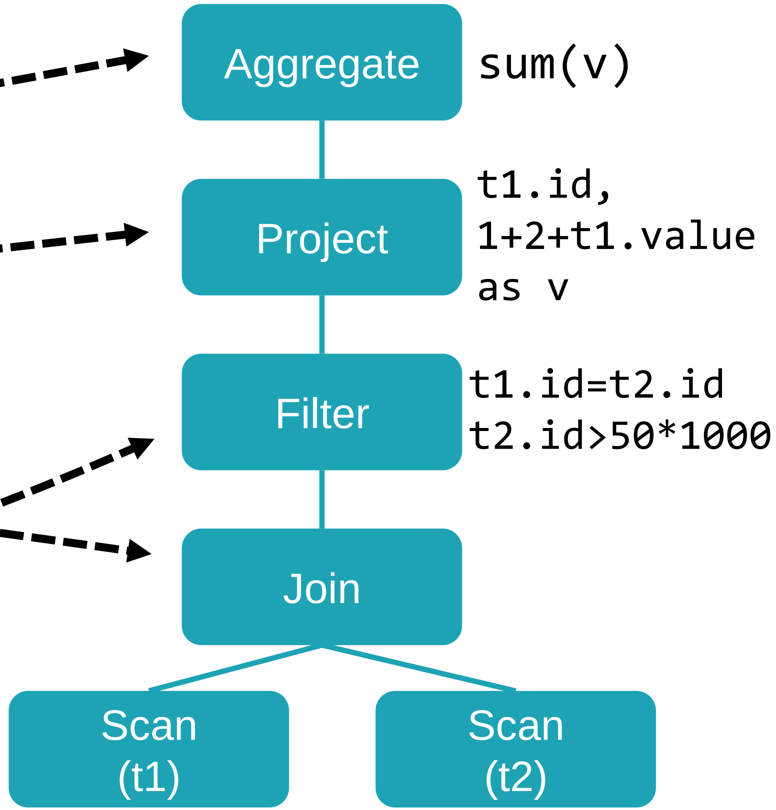
```
SELECT sum(v)
FROM (
  SELECT
    t1.id,
    1 + 2 + t1.value AS v
  FROM t1 JOIN t2
  WHERE
    t1.id = t2.id AND
    t2.id > 50 * 1000) tmp
```

- An expression represents a new value, computed based on input values
  - e.g. `1 + 2 + t1.value`

Trees: Abstractions of Users' Programs

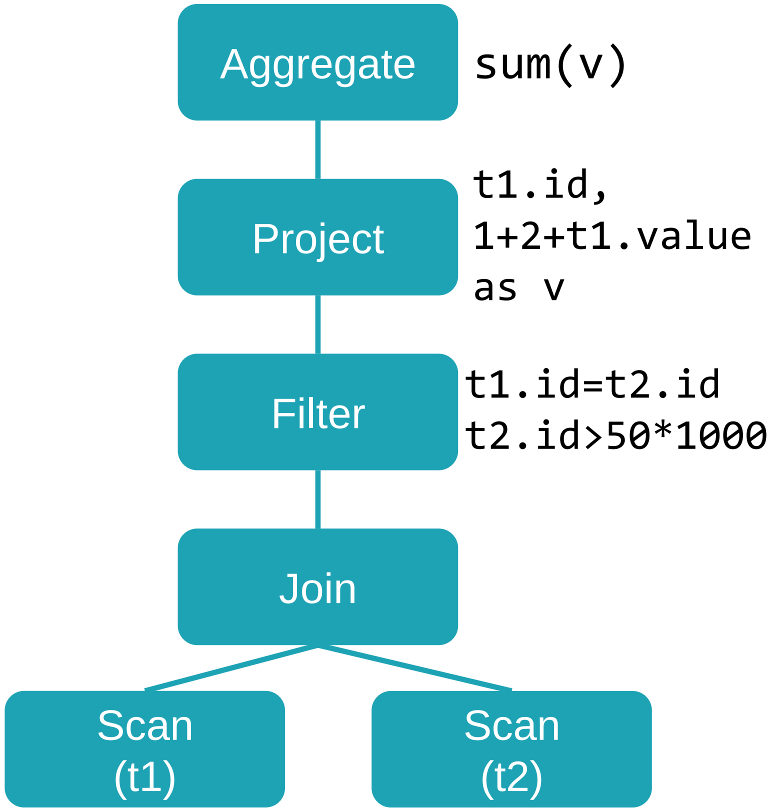
Query Plan

```
SELECT sum(v)
FROM (
  SELECT
    t1.id,
    1 + 2 + t1.value AS v
  FROM t1 JOIN t2
  WHERE
    t1.id = t2.id AND
    t2.id > 50 * 1000) tmp
```



Logical Plan

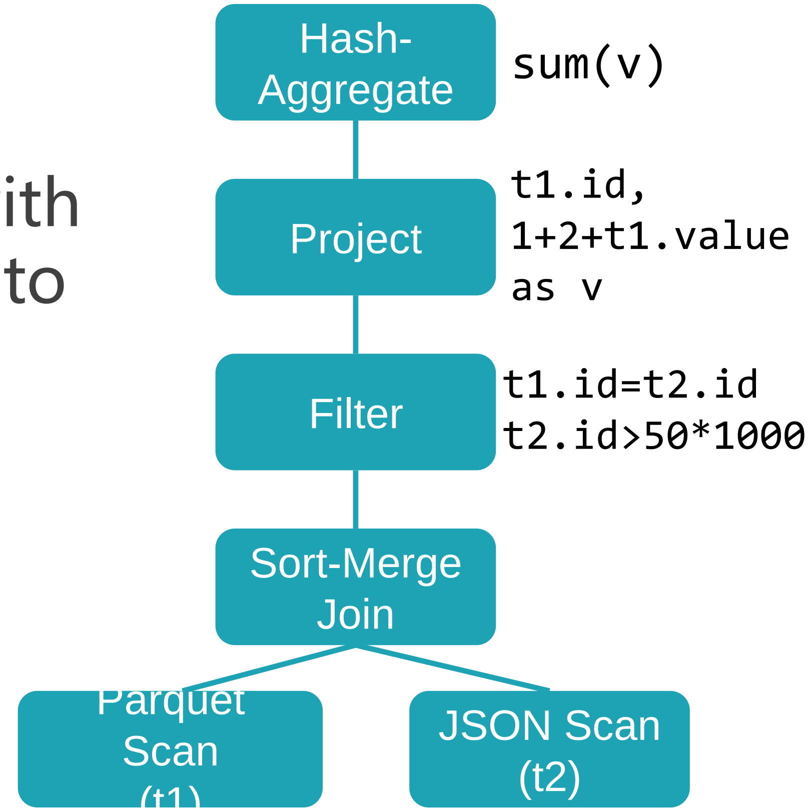
- A Logical Plan describes computation on datasets **without** defining how to conduct the computation



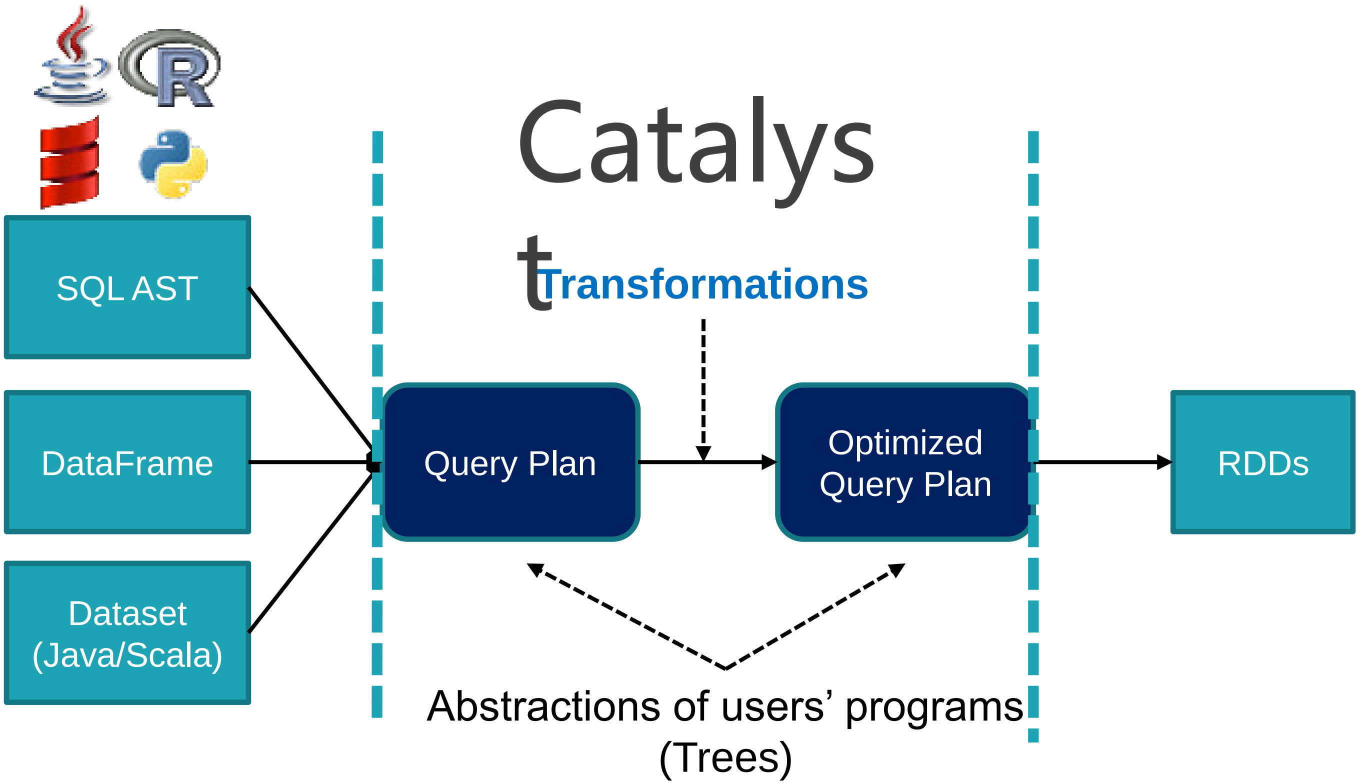
databricks

Physical Plan

- A Physical Plan describes computation on datasets with specific definitions on how to conduct the computation



How Catalyst Works: An Overview

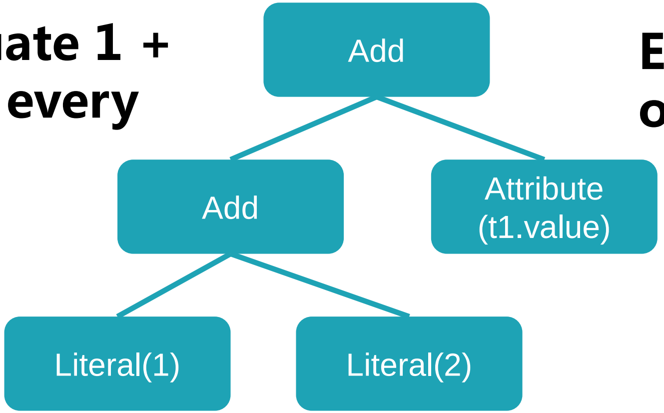


Transform

- A function associated with every tree used to implement a single rule

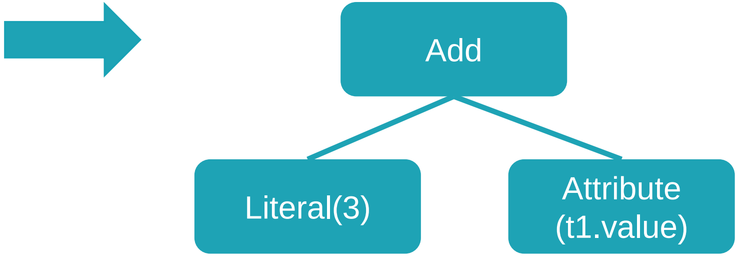
1 + 2 + t1.value

Evaluate 1 + 2 for every row



Evaluate 1 + 2 once

3+ t1.value



Transform

- A transform is defined as a Partial Function
- Partial Function: A function that is defined for a subset of its possible arguments

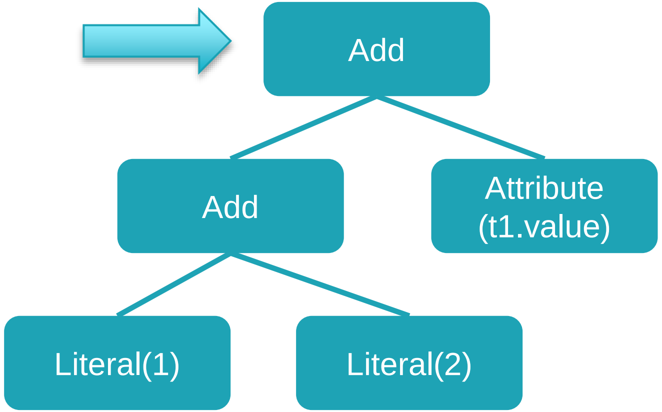
```
val expression: Expression = ...
expression.transform {
  case Add(Literal(x, IntegerType), Literal(y, IntegerType)) =>
    Literal(x + y)
}
```

Case statement determine if the partial function is defined for a given input

Transform

```
val expression: Expression = ...
expression.transform {
  case Add(Literal(x, IntegerType), Literal(y, IntegerType)) =>
    Literal(x + y)
}
```

1 + 2 + t1.value

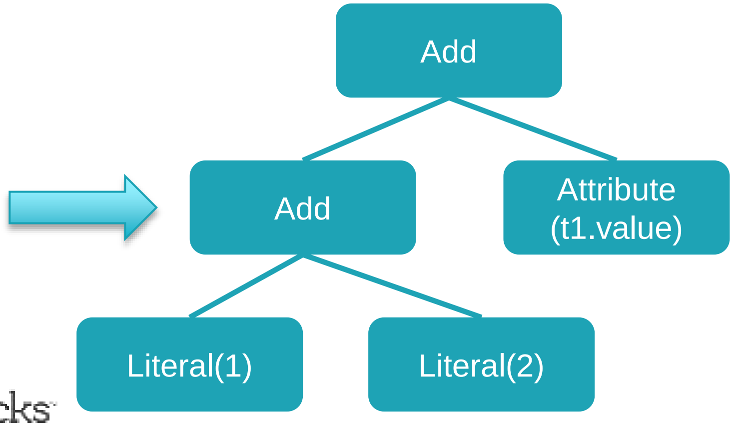


databricks

Transform

```
val expression: Expression = ...
expression.transform {
  case Add(Literal(x, IntegerType), Literal(y, IntegerType)) =>
    Literal(x + y)
}
```

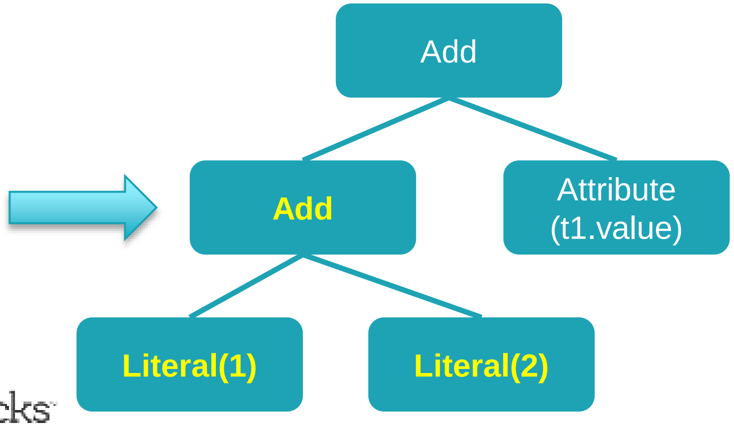
1 + 2 + t1.value



Transform

```
val expression: Expression = ...
expression.transform {
  case Add(Literal(x, IntegerType), Literal(y, IntegerType)) =>
    Literal(x + y)
}
```

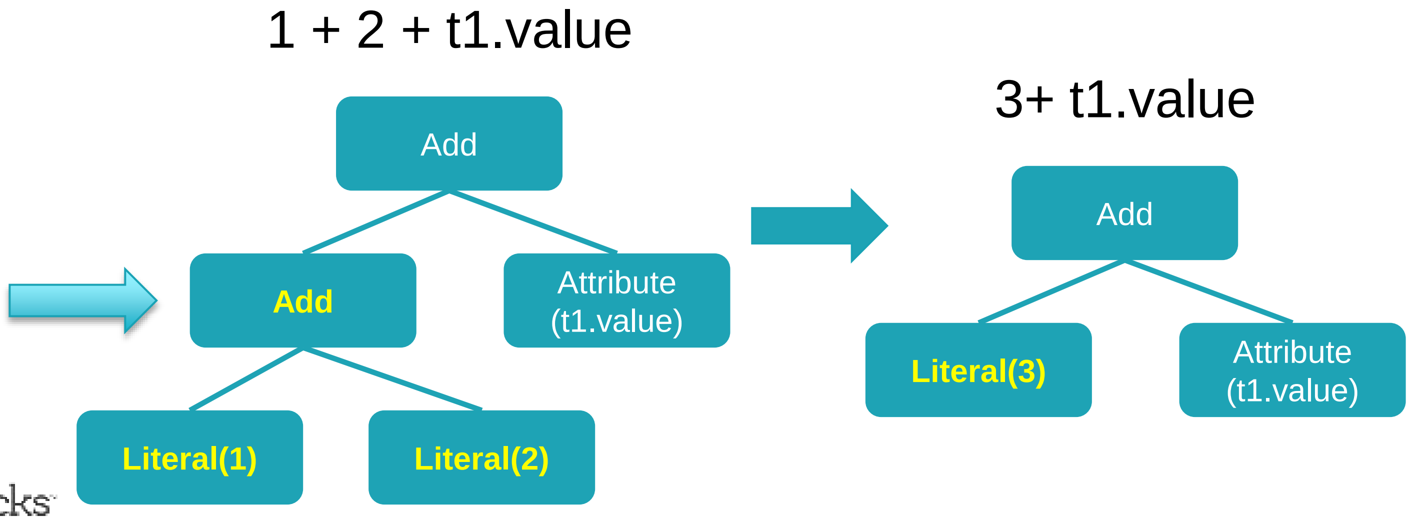
1 + 2 + t1.value



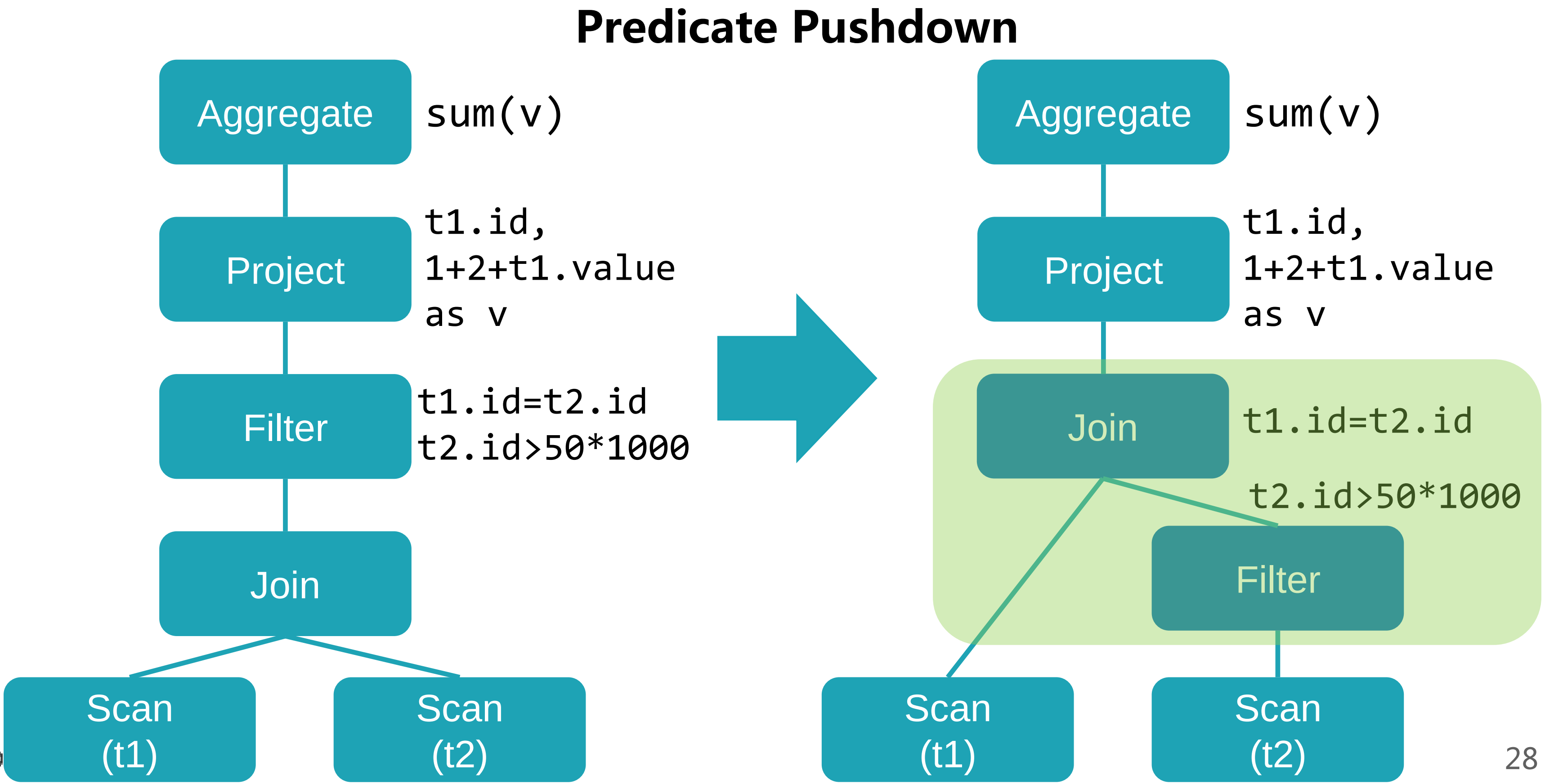
databricks

Transform

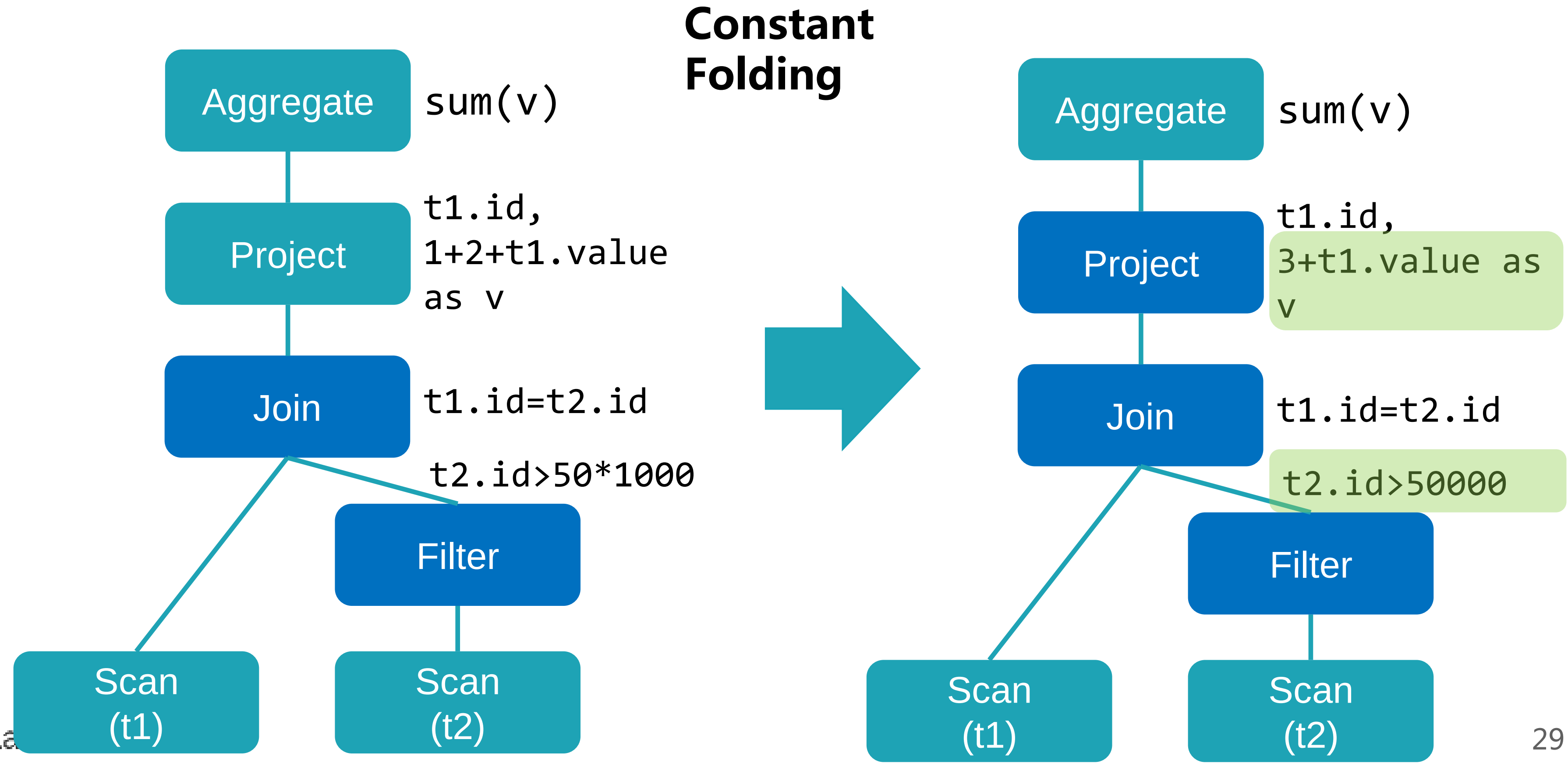
```
val expression: Expression = ...
expression.transform {
  case Add(Literal(x, IntegerType), Literal(y, IntegerType)) =>
    Literal(x + y)
}
```



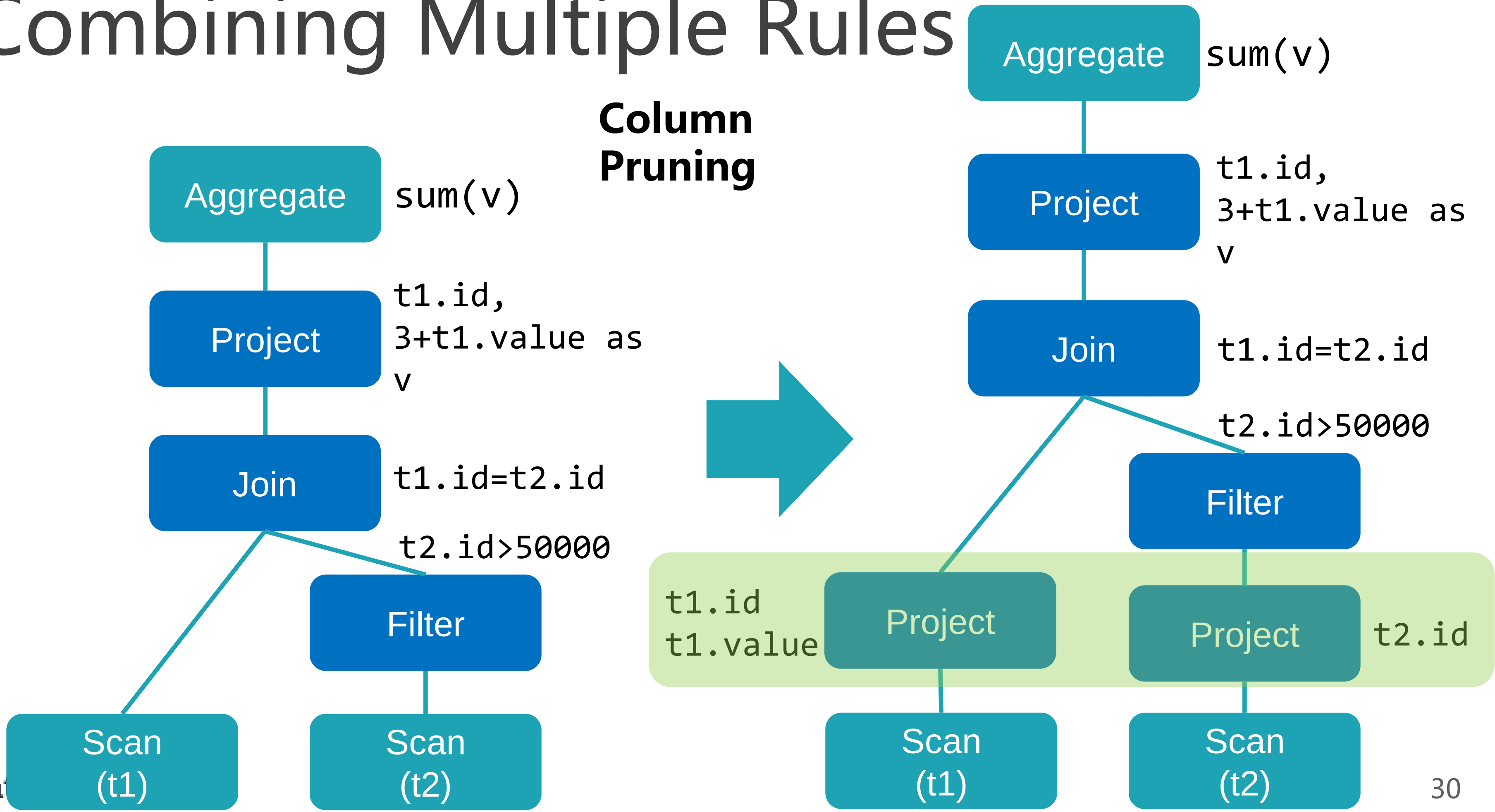
Combining Multiple Rules



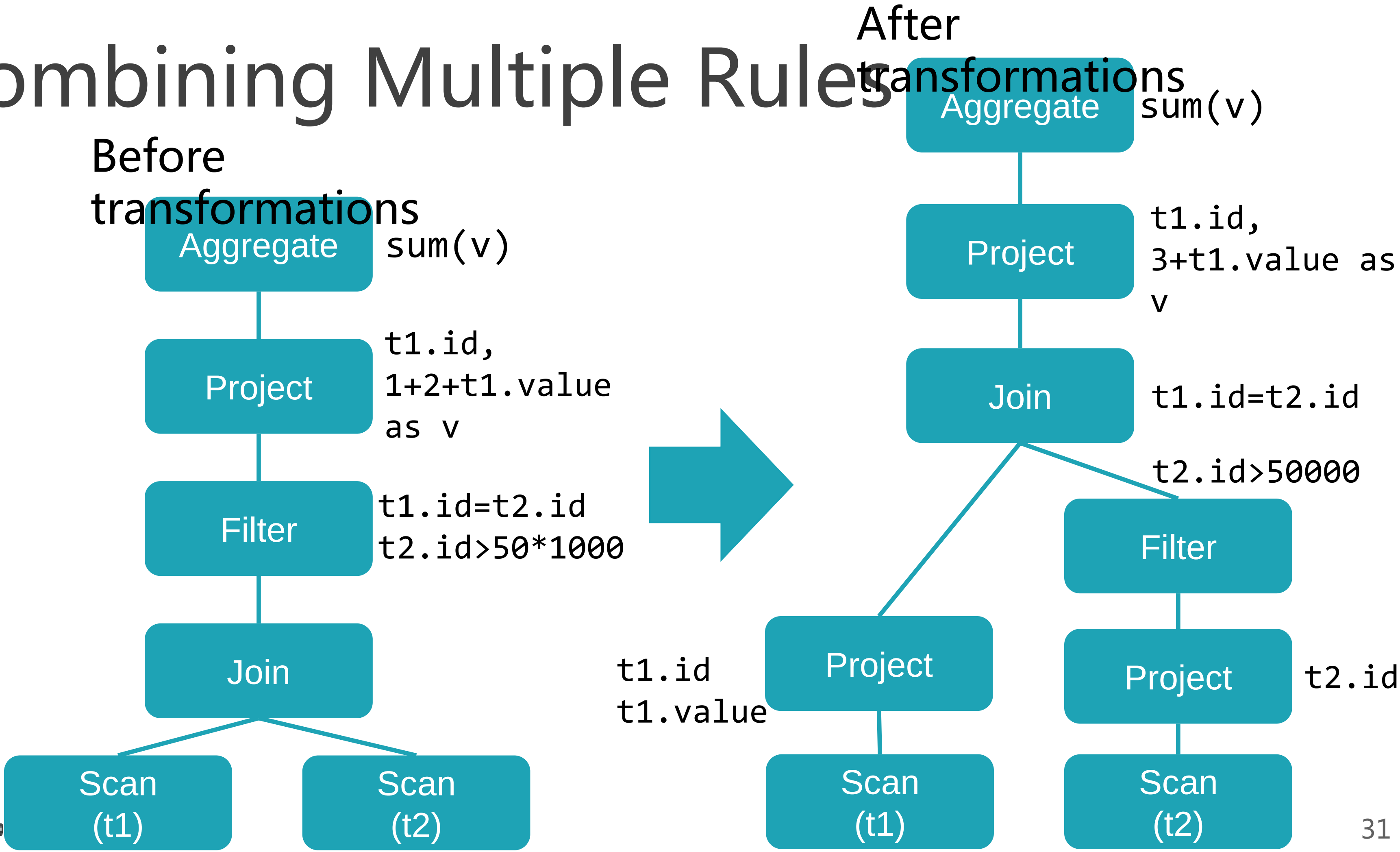
Combining Multiple Rules

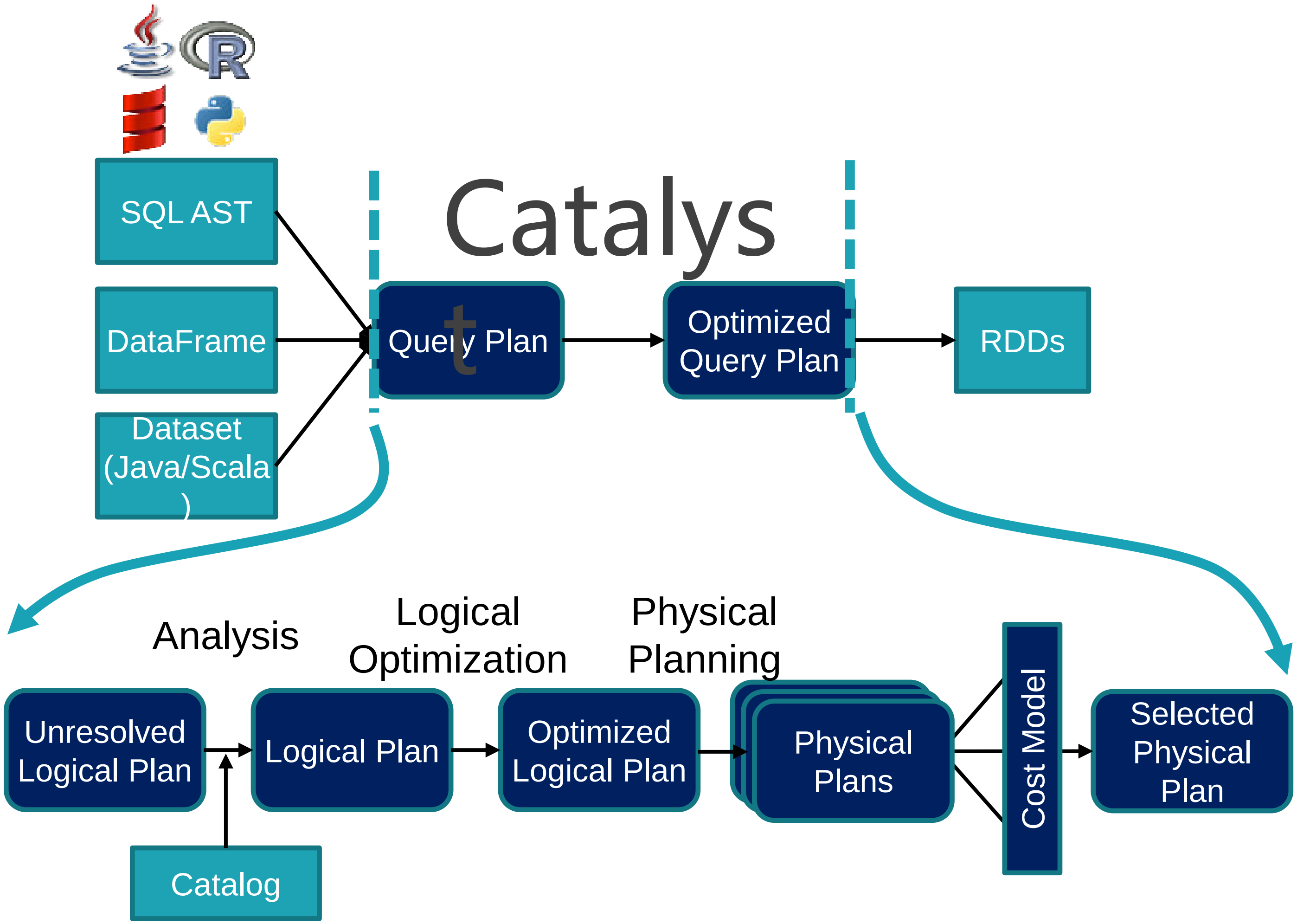


Combining Multiple Rules

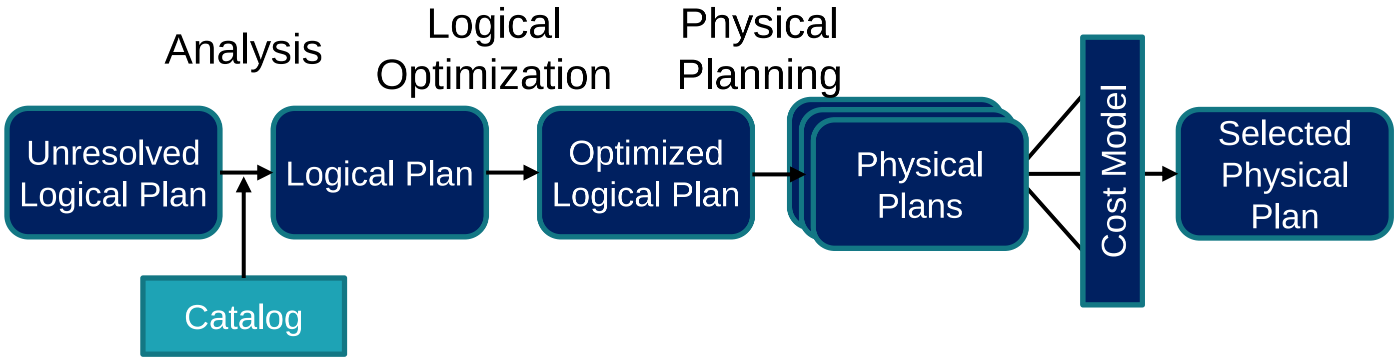


Combining Multiple Rules



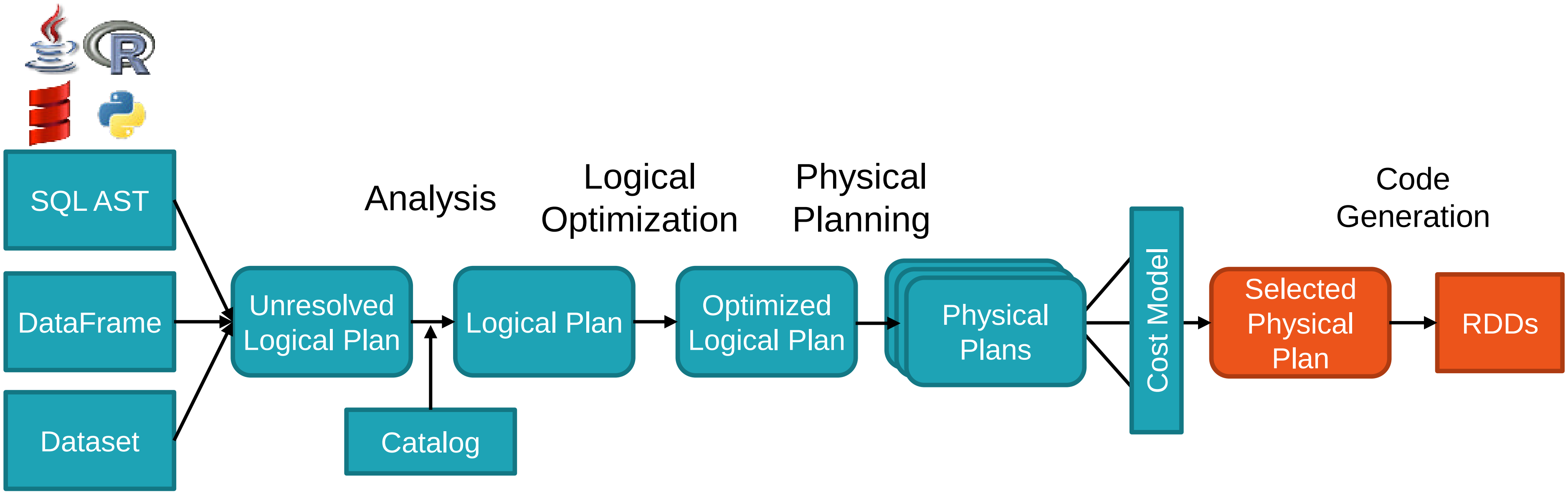


SQL AST
DataFrame
Dataset (Java/Scala)
Catalyst
Query Plan
Optimized Query Plan
RDDs
Analysis
Logical Optimization
Physical Planning
Unresolved Logical Plan
Logical Plan
Optimized Logical Plan
Physical Plans
Cost Model
Selected Physical Plan
Catalog



- **Analysis:** Transforms an Unresolved Logical Plan to a Resolved Logical Plan
  - Unresolved => Resolved: Use Catalog to find where datasets and columns are coming from and types of columns
- **Logical Optimization**: Transforms a Resolved Logical Plan to an Optimized Logical Plan
- **Physical Planning**: Transforms a Optimized Logical Plan to a Physical Plan

The Backend Execution Engine



Volcano—An Extensible and Parallel Query Evaluation System

Goetz Graefe

Abstract—To investigate the interactions of extensibility and parallelism in database query processing, we have developed a new dataflow query execution system called Volcano. The Volcano effort provides a rich environment for research and education in database systems design, heuristics for query optimization, parallel query execution, and resource allocation.

Volcano uses a standard interface between algebra operators, allowing easy addition of new operators and operator implementations. Operations on individual items, e.g., predicates, are imported into the query processing operators using *support functions*. The semantics of support functions is not prescribed; any data type including complex objects and any operation can be realized. Thus, Volcano is *extensible* with new operators, algorithms, data types, and type-specific methods.

Volcano includes two novel *meta-operators*. The *choose-plan*

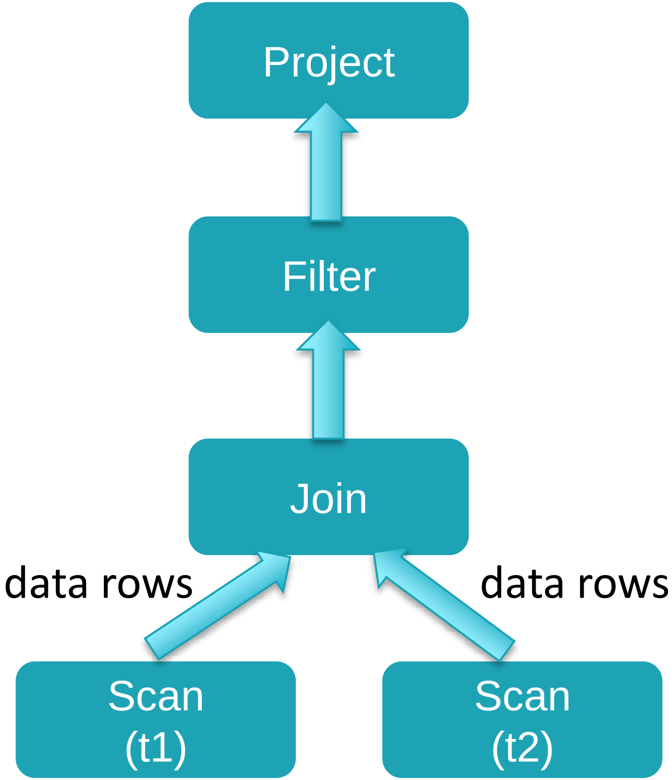
tem as it lacks features such as a user-friendly query language, a type system for instances (record definitions), a query optimizer, and catalogs. Because of this focus, Volcano is able to serve as an experimental vehicle for a multitude of purposes, all of them open-ended, which results in a combination of requirements that have not been integrated in a single system before. First, it is modular and extensible to enable future research, e.g., on algorithms, data models, resource allocation, parallel execution, load balancing, and query optimization heuristics. Thus, Volcano provides an infrastructure for experimental research rather than a final research prototype in itself. Second, it

G. Graefe, **Volcano— An Extensible and Parallel Query Evaluation System**,
In IEEE Transactions on Knowledge and Data Engineering 1994

Volcano Iterator Model

- Standard for 30 years: almost all databases do it

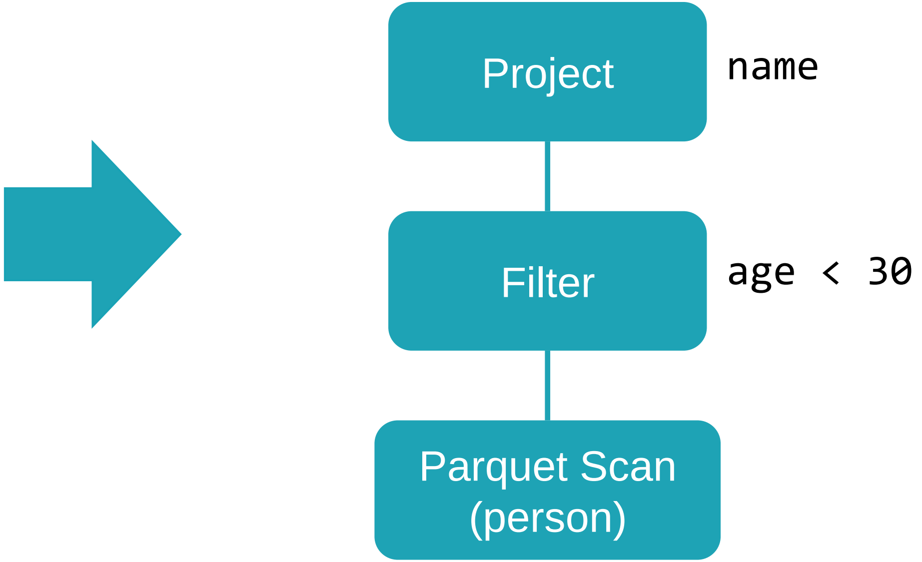
- Each operator is an "iterator" that consumes records from its input operator



databricks

How Spark SQL Run Queries

```
SELECT name
FROM person
WHERE age < 30
```



How Spark SQL Run Queries

```
class ParquetScan {
  def execute(): RDD[Row] = {
    ...
  }
}
```

How Spark SQL Run Queries

```scala
class FilterExec(condition: Expression) {
  def execute(): RDD[Row] = {
    child.execute().mapPartitions { input =>
      val predicate: Row => Boolean = row => {
        condition.eval(row)
      }
      input.filter(predicate)
    }
  }
}
```

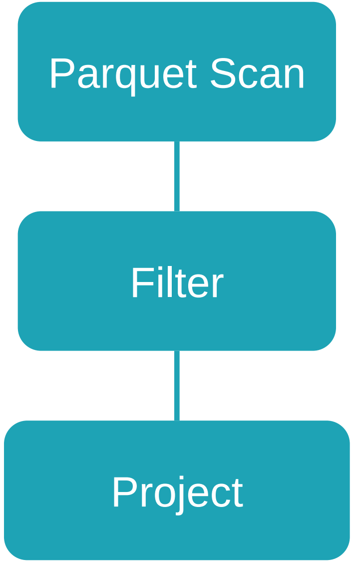
How Spark SQL Run Queries

```
class ProjectExec(projectList: Seq[Expression]) {
  def execute(): RDD[Row] = {
    child.execute().mapPartitions { input =>
      val project: Row => Row = ...
      input.map(project)
    }
  }
}
```

How Spark SQL Run Queries

```
val tableScan: RDD[Row] = ...
tableScan.mapPartitions { input =>
  val predicate: Row => Boolean = ...
  input.filter(predicate)
}.mapPartitions { input =>
  val project: Row => Row = ...
  input.map(project)
}
```

How Spark SQL Run Queries




```scala
val tableScan: RDD[Row] = ...
tableScan.mapPartitions { input =>
  val predicate: Row => Boolean = ...
  input.filter(predicate)
}.mapPartitions { input =>
  val project: Row => Row = ...
  input.map(project)
}
```

Data Exchange

```
SELECT dept, AVG(age) FROM pdata GROUP BY dept
```


```
group by: dept
output: AVG(age)
```

Data Exchange

Partition 1

(dept="a", age=20)
(dept="a", age=22)
(dept="b", age=18)

...

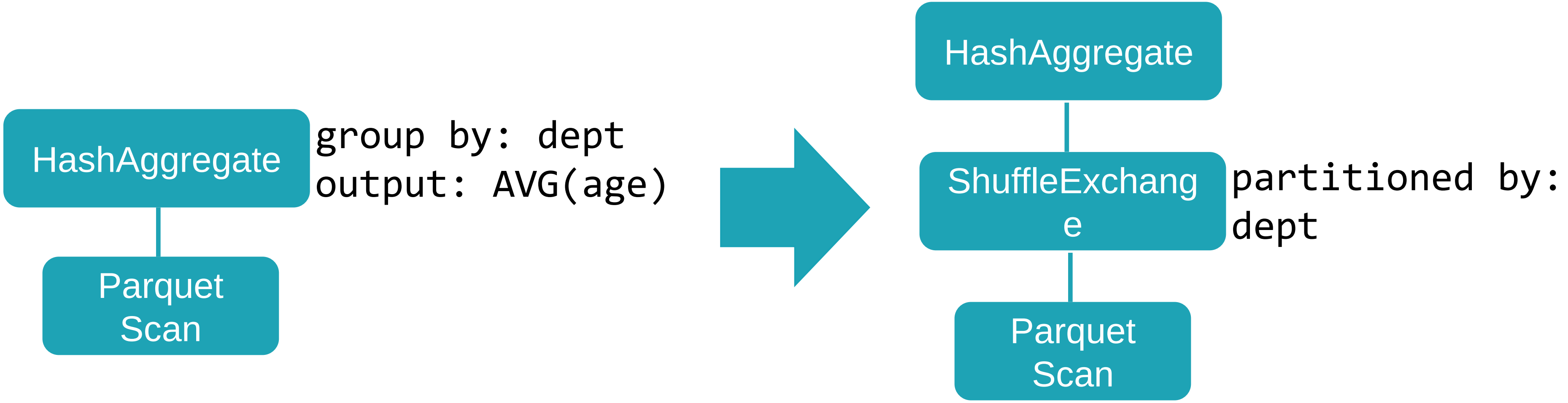
Partition n

(dept="b", age=26)
(dept="b", age=22)
(dept="c", age=24)

mapPartitions

dept=a	ageCount=123, ageSum=2312
dept=b	ageCount=45, ageSum=912
…	…

Data Exchange



Optimized Execution with Project Tungsten

Binary encoding of row object

Expression code generation

Whole stage code generation

Vectorization

The overheads of JVM objects

"abcd"

- Native: 4 bytes with UTF-8 encoding
- Java: 48 bytes

java.lang.String object internals:

OFFSET	SIZE	TYPE DESCRIPTION	VALUE
0	4	(object header)	...
4	4	(object header)	...
8	4	(object header)	...
12	4	char[] String.value	[]
16	4	int String.hash	0
20	4	int String.hash32	0

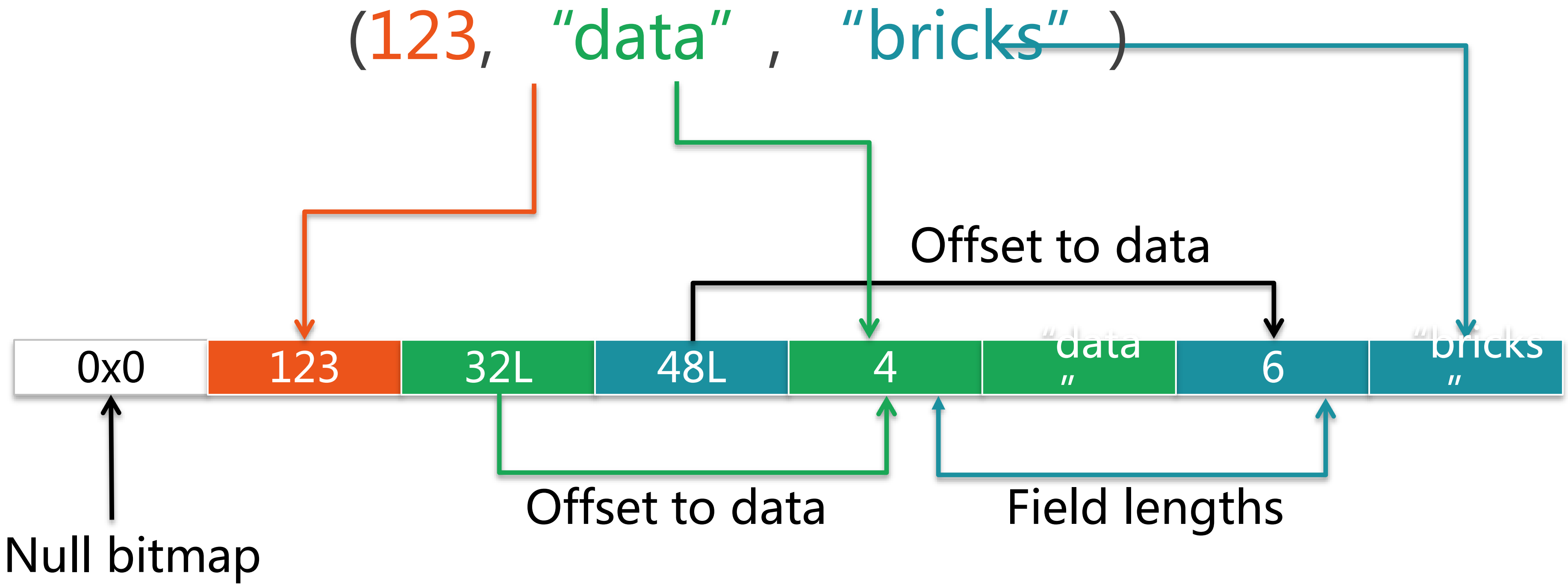
Instance size: 48 bytes (reported by Instrumentation API)

12 byte object header

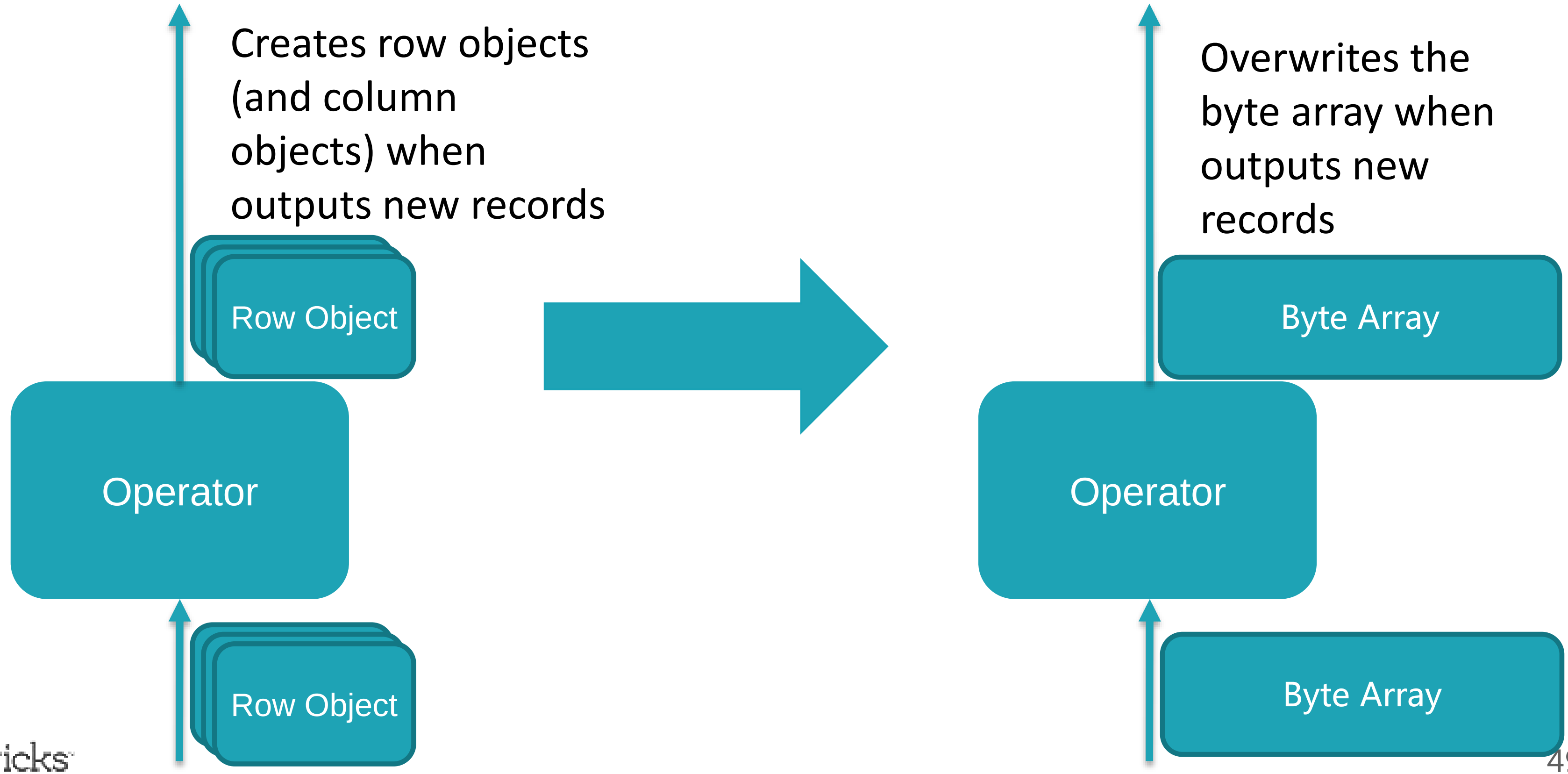
20 bytes data + overh

8 byte hashcode

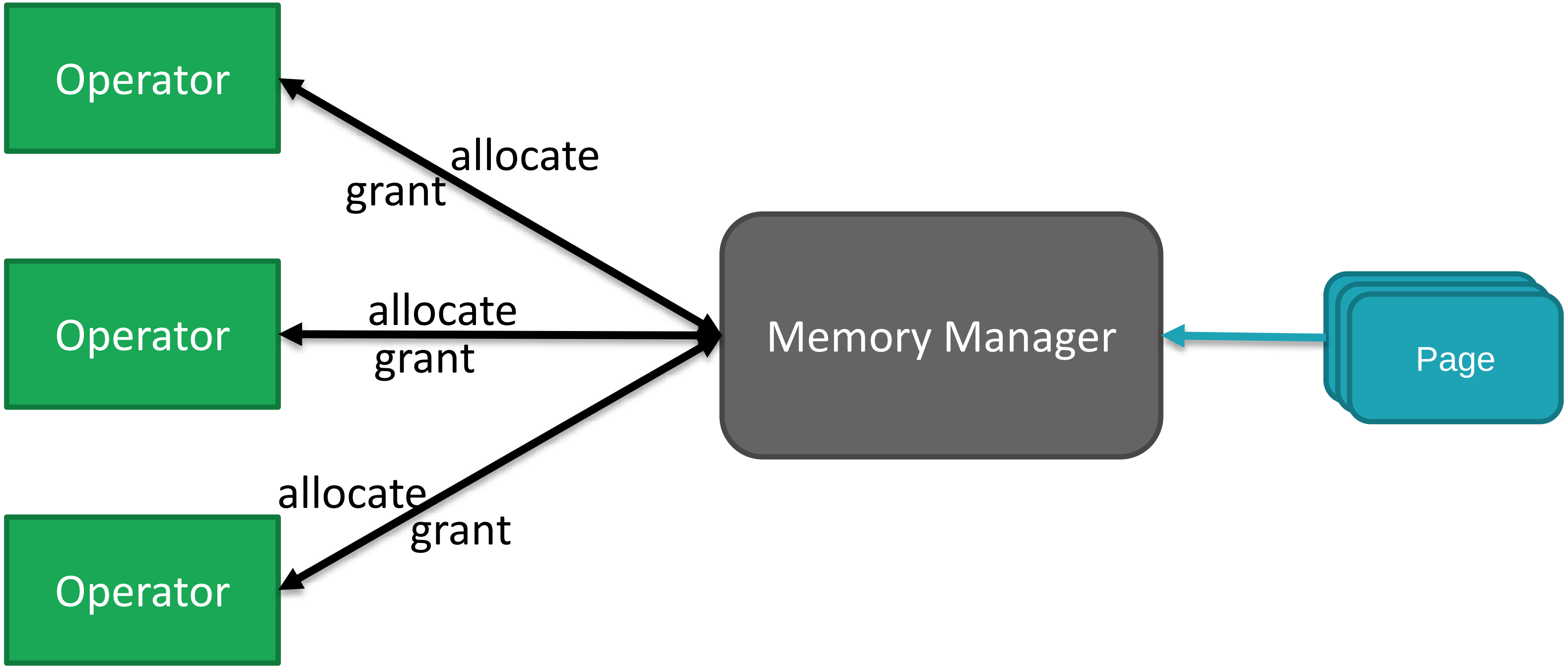
Tungsten's Compact Encoding



Less Objects Creation

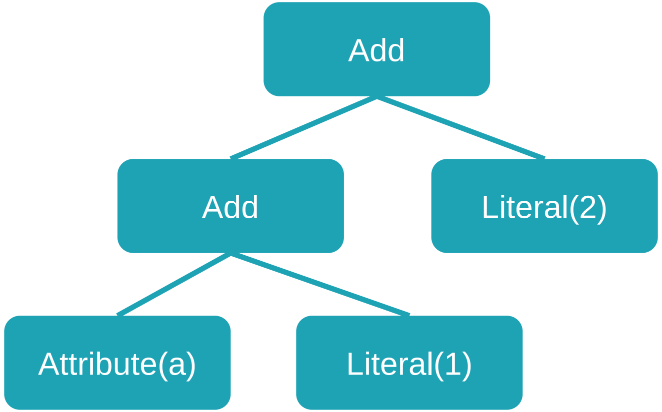


Manual Memory Management

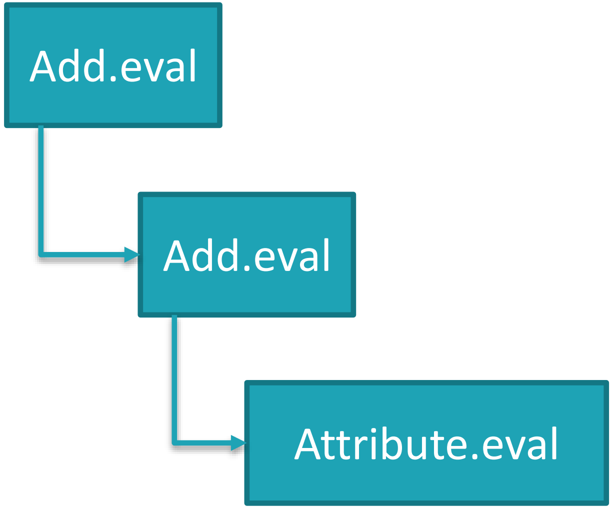


How to Evaluate Expression

a + 1 + 2



Function calls



databricks

Expression Code Generation

DataFrame Code / SQL

```
df.where(df("year") > 2015)
```

Catalyst Expressions

```
GreaterThan(year#234, Literal(2015))
```

Low-level Java code

```
boolean filter(Object baseObject) {
   int offset = baseOffset + bitSetWidthInBytes + 3*8L;
   int value = Platform.getInt(baseObject, offset);
   return value34 > 2015;
}
```

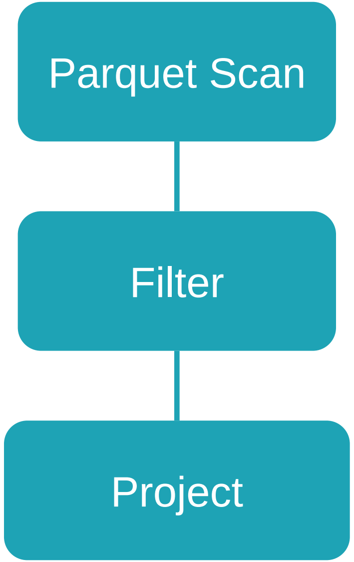
JVM *intrinsic* JIT-ed to pointer arithmetic

Expression Code Generation

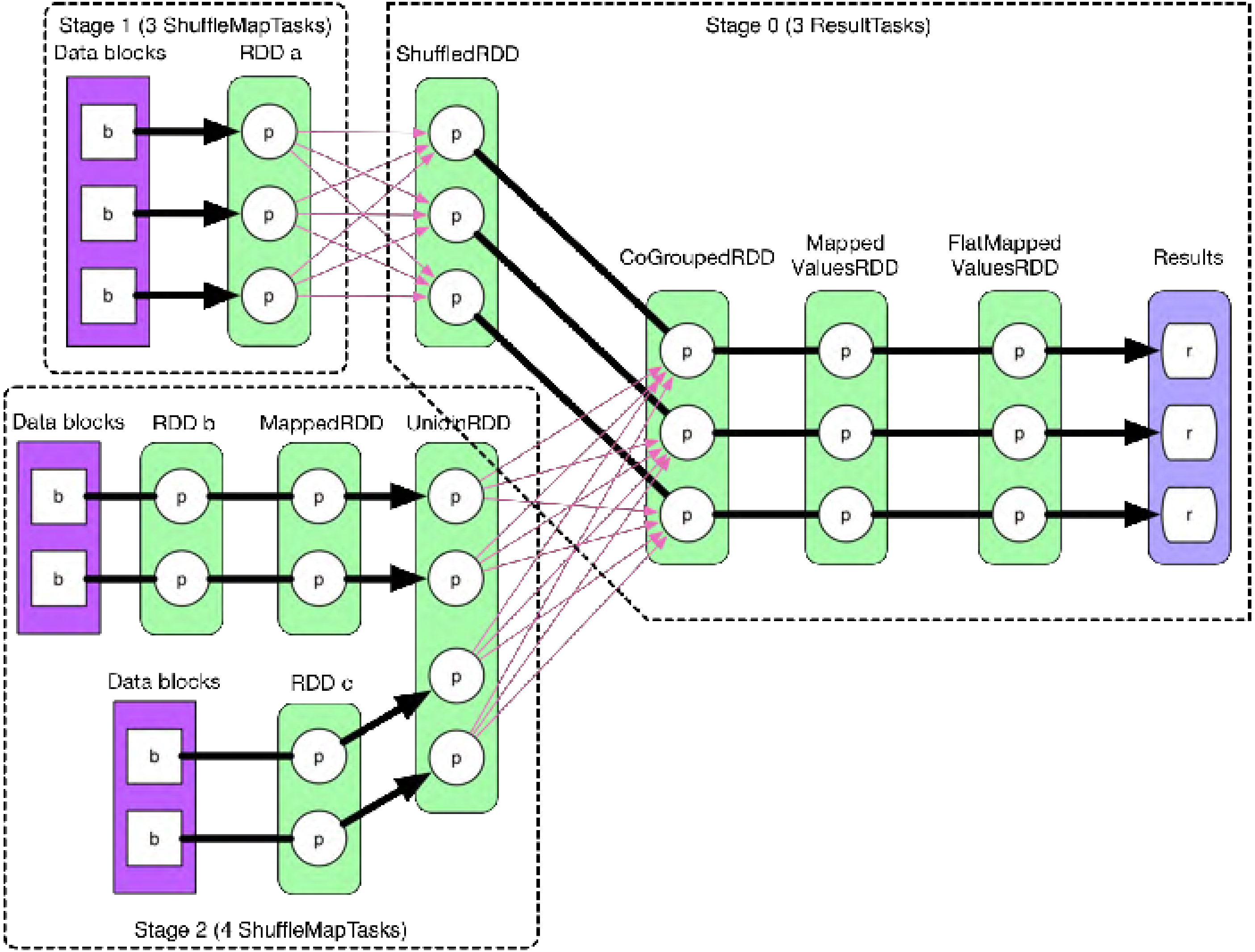
Saves a lot of virtual function calls and boxing costs!

```
class FilterExec(condition: Expression) {
  def execute(): RDD[Row] = {
    child.execute().mapPartitions { input =>
      val predicate: Row => Boolean =
        PredicateGenerator.generate(condition)
      input.filter(predicate)
    }
  }
}
```

After Expression Code Generation



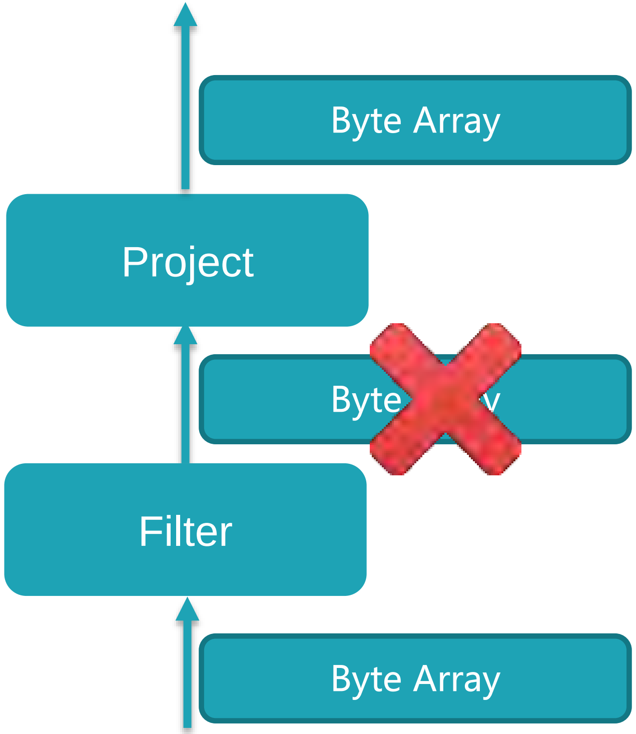

```scala
val tableScan: RDD[Row] = ...
tableScan.mapPartitions { input =>
  val predicate: Row => Boolean = ...
  input.filter(predicate)
}.mapPartitions { input =>
  val project: Row => Row = ...
  input.map(project)
}
```


Stage 1 (3 ShuffleMapTasks)
Data blocks
RDD a
b
p
Stage 0 (3 ResultTasks)
ShuffledRDD
CoGroupedRDD
Mapped ValuesRDD
FlatMapped ValuesRDD
Results
r
Data blocks
RDD b
MappedRDD
UnionRDD
Data blocks
RDD c
Stage 2 (4 ShuffleMapTasks)

What We Really Run

```
val predicate = ... // generated code
val project = ... // generated code
input.filter(predicate).map(project)
```

What We Really Run



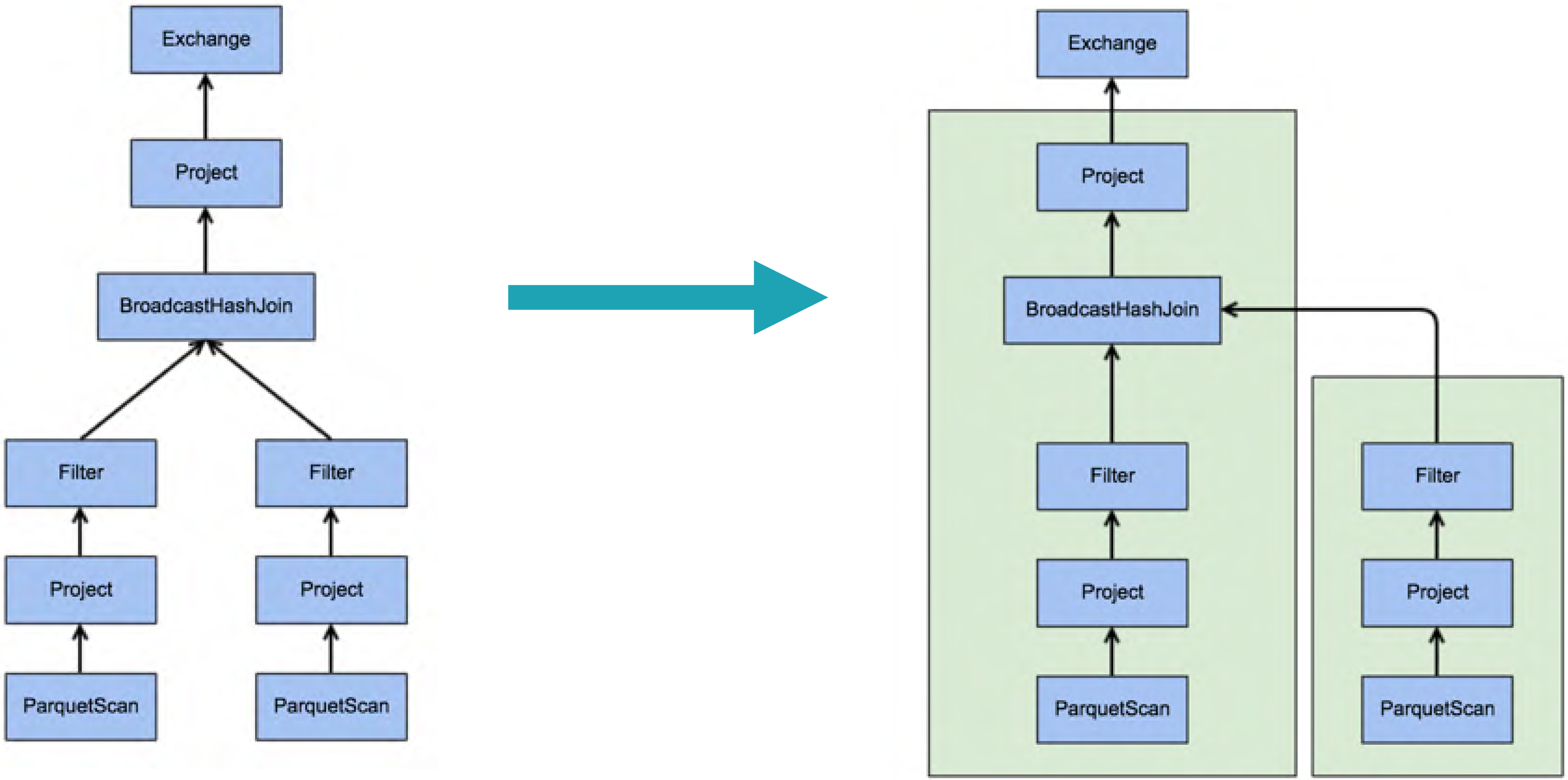
databricks

Whole Stage Code Generation

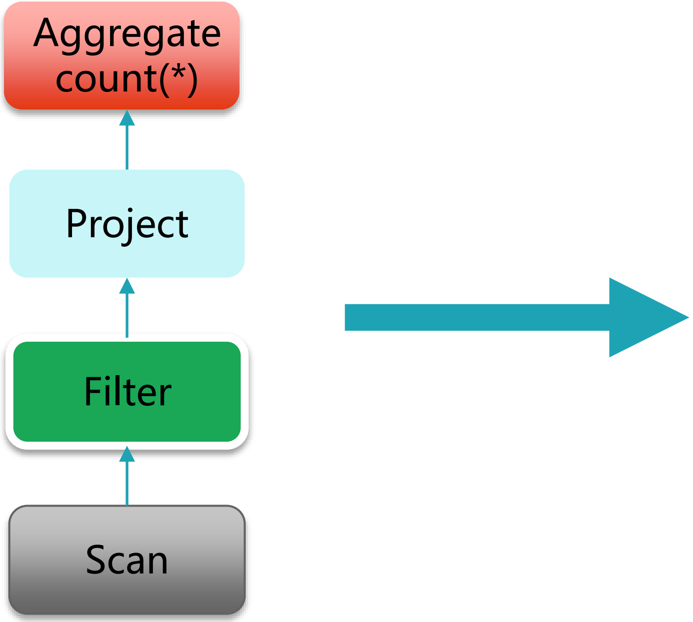
Fusing operators together:

- Identify chains of operators (“stages”)
- Compile each stage into a single function

Whole Stage Codegen: Planner

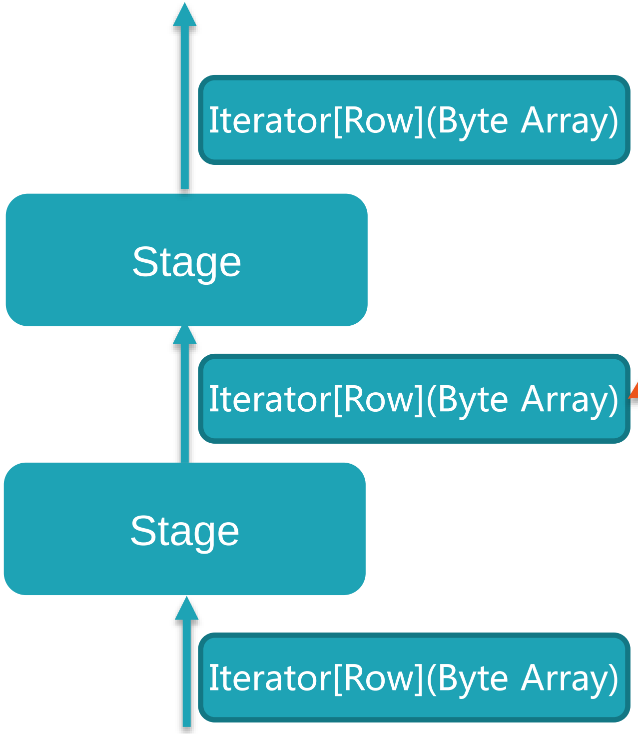


Whole Stage Codegen: Generate code like handwritten



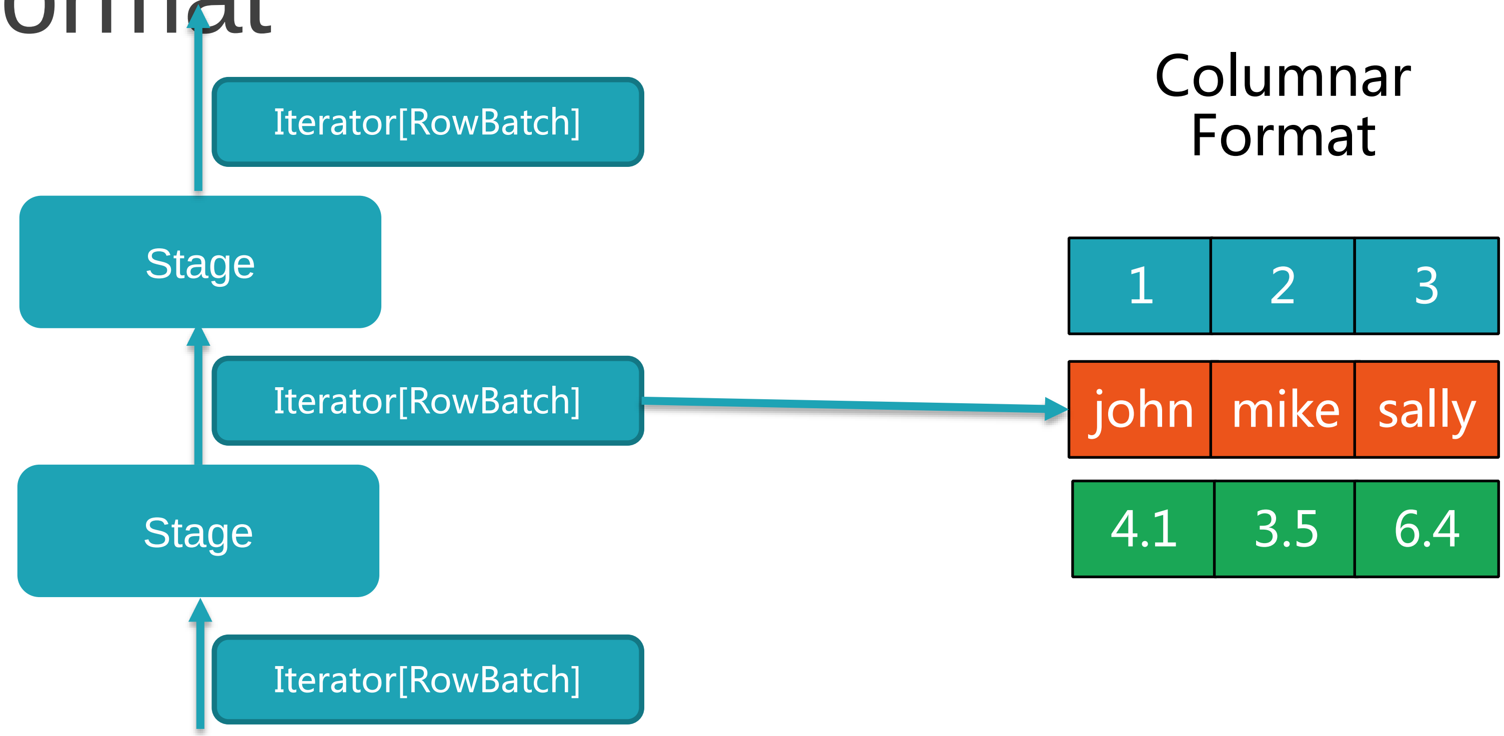

```
long count = 0;
for (ss_item_sk in store_sales) {
  if (ss_item_sk == 1000) {
    count += 1;
  }
}
```

Where Can We Push Further?



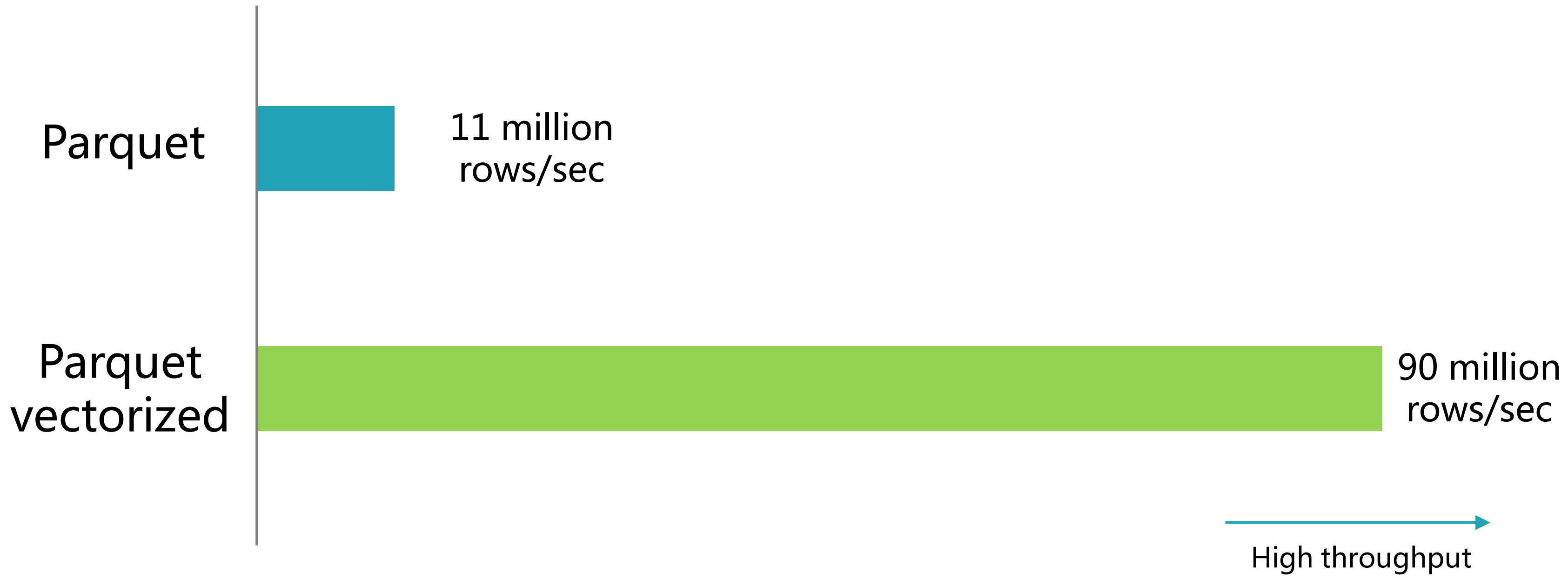
Iterator.next wastes a lot performance!

Vectorization: Batch + Columnar Format



Why columnar?

1. More efficient: denser storage, regular data access, easier to index into
2. More compatible: Most high-performance external systems are already columnar (numpy, TensorFlow, Parquet); zero serialization/copy to work with them
3. Easier to extend: process encoded data



Note: End-to-end, single thread, single column, and data originated in Parquet on disk

Putting it All Together

Operator Benchmarks: Cost/Row (ns)

primitive	Spark 1.6	Spark 2.0
filter	15 ns	1.1 ns
sum w/o group	14 ns	0.9 ns
sum w/ group	79 ns	10.7 ns
hash join	115 ns	4.0 ns
sort (8-bit entropy)	620 ns	5.3 ns
sort (64-bit entropy)	620 ns	40 ns
sort-merge join	750 ns	700 ns
Parquet decoding (single int column)	120 ns	13 ns

5-30x
Speedups

Operator Benchmarks: Cost/Row (ns)

primitive	Spark 1.6	Spark 2.0
filter	15 ns	1.1 ns
sum w/o group	14 ns	0.9 ns
sum w/ group	79 ns	10.7 ns
hash join	115 ns	4.0 ns
sort (8-bit entropy)	620 ns	5.3 ns
sort (64-bit entropy)	620 ns	40 ns
sort-merge join	750 ns	700 ns
Parquet decoding (single int column)	120 ns	13 ns

Radix Sort
10-100x
Speedups

Operator Benchmarks: Cost/Row (ns)

primitive	Spark 1.6	Spark 2.0
filter	15 ns	1.1 ns
sum w/o group	14 ns	0.9 ns
sum w/ group	79 ns	10.7 ns
hash join	115 ns	4.0 ns
sort (8-bit entropy)	620 ns	5.3 ns
sort (64-bit entropy)	620 ns	40 ns
sort-merge join	750 ns	700 ns
Parquet decoding (single int column)	120 ns	13 ns

Shuffling still the bottleneck

databricks

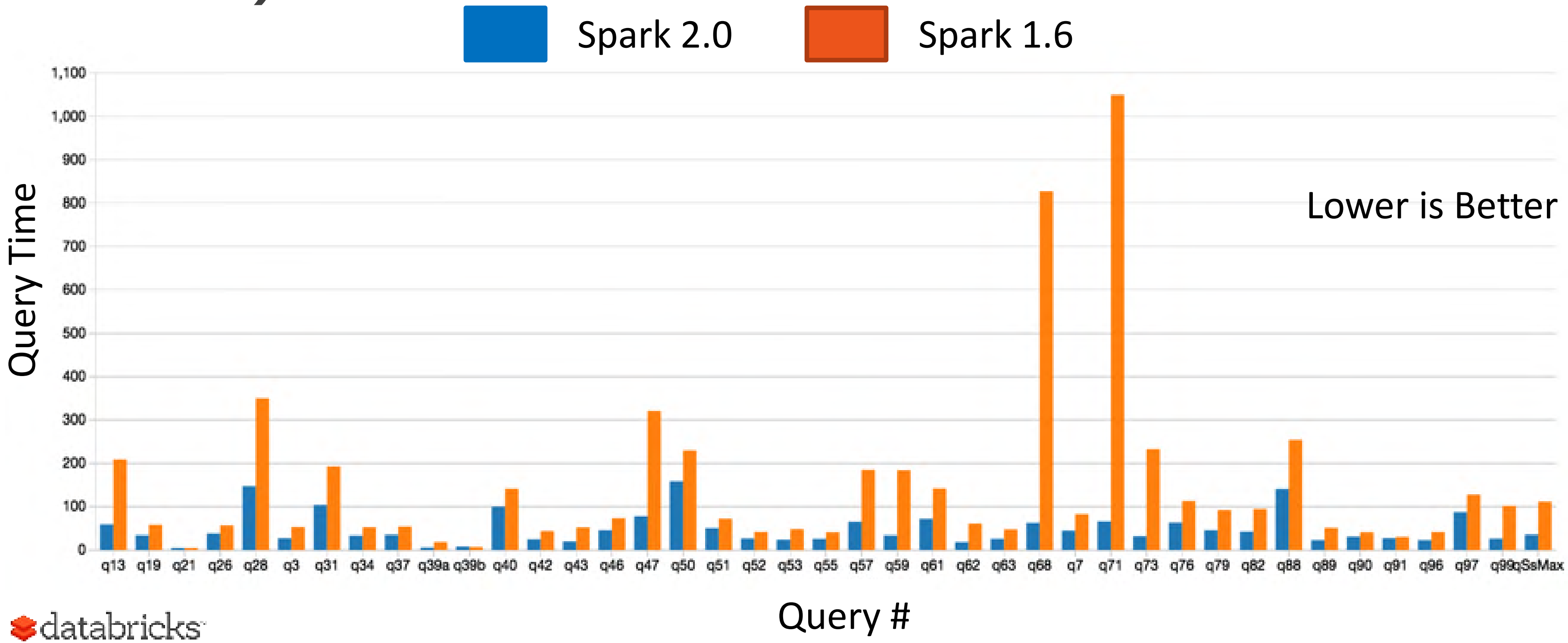
Operator Benchmarks: Cost/Row (ns)

primitive	Spark 1.6	Spark 2.0
filter	15 ns	1.1 ns
sum w/o group	14 ns	0.9 ns
sum w/ group	79 ns	10.7 ns
hash join	115 ns	4.0 ns
sort (8-bit entropy)	620 ns	5.3 ns
sort (64-bit entropy)	620 ns	40 ns
sort-merge join	750 ns	700 ns
Parquet decoding (single int column)	120 ns	13 ns

10x Speedup

databricks

TPC-DS (Scale Factor 1500, 100 cores)



databricks

What’s Next?

Spark 2.1, 2.2 and beyond

1. SPARK-16026: Cost Based Optimizer
   - Leverage table/column level statistics to optimize joins and aggregates
   - Statistics Collection Framework (Spark 2.1)
   - Cost Based Optimizer (Spark 2.2)
2. Boosting Spark’s Performance on Many-Core Machines
   - In-memory/ single node shuffle
3. Improving quality of generated code and better integration with the in-memory column format in Spark

Further Reading

Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop

Deep dive into the new Tungsten execution engine

by Sameer Agarwal, Davies Liu and Reynold Xin

Posted in **ENGINEERING BLOG** | May 23, 2016

http://tinyurl.com/project-tungsten

Thank you.

databricks™