SparkSQL: A Compiler from Queries to RDDs

Sameer Agarwal

Spark Summit | Boston | February 9th 2017

databricks™

About Me

- Software Engineer at Databricks (Spark Core/SQL)
- PhD in Databases (AMPLab, UC Berkeley)
- Research on BlinkDB (Approximate Queries in Spark)

Background: What is an RDD?

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

Background: What is an RDD?

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

Opaque Computation

Background: What is an RDD?

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

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()
```

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()
```

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

databricks

RDD Programming Model

Construct execution DAG using low level RDD operators.

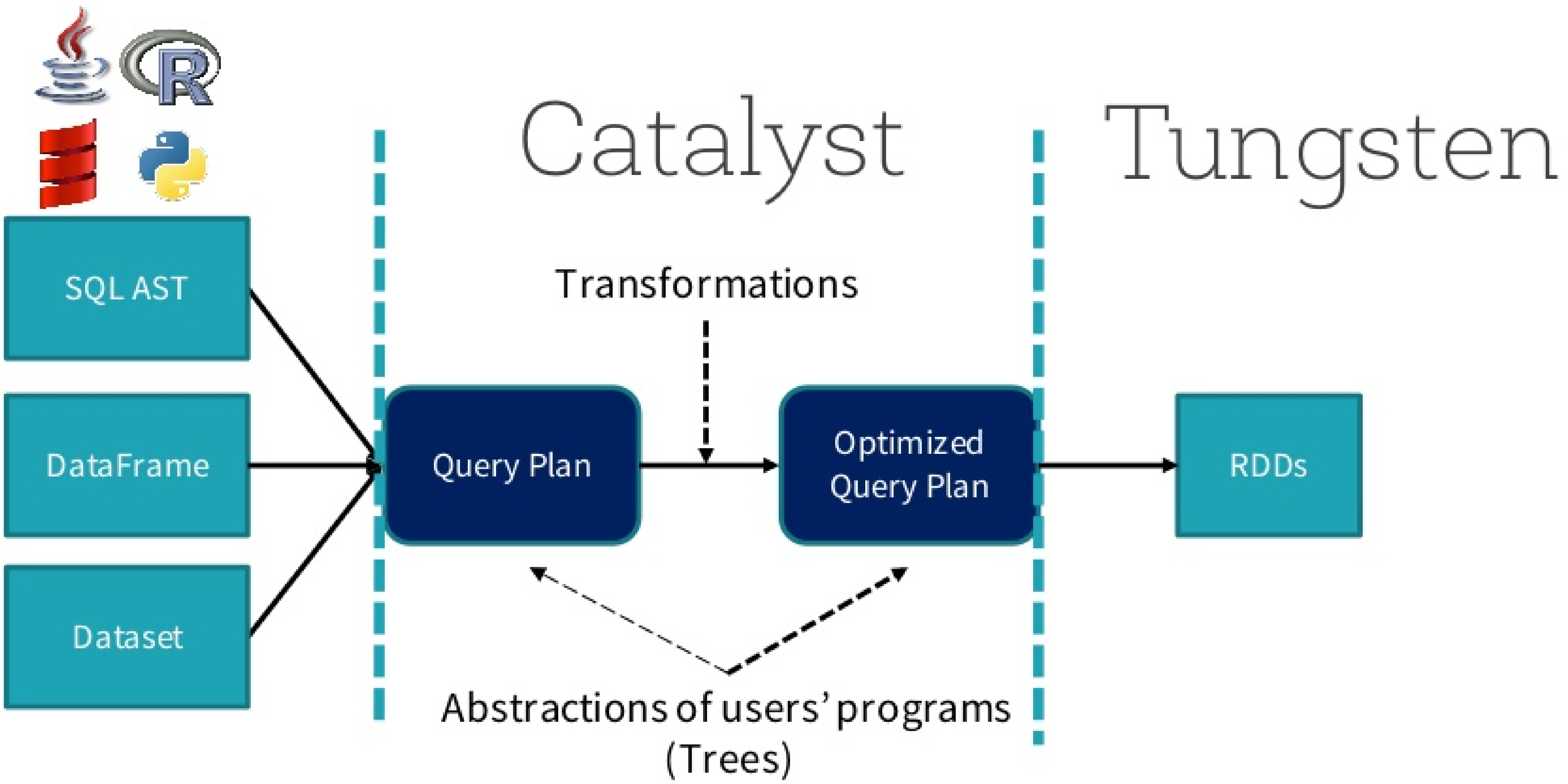
```
pdata.map(lambda x: (x.dept, [x.age, 1])) \
  .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
  .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
  .collect()
```

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

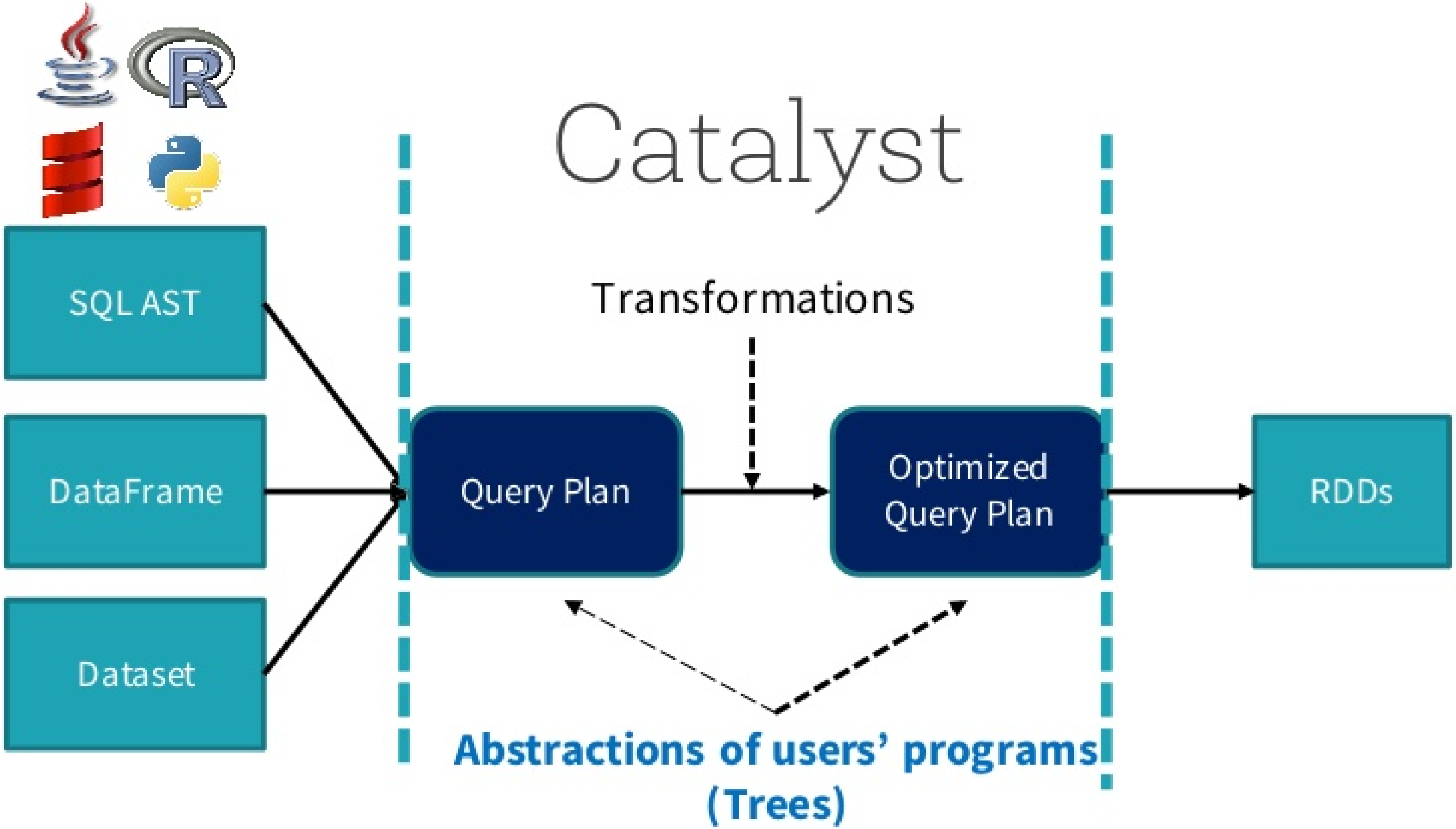
SQL/Structured Programming Model

- **High-level APIs (SQL, DataFrame/Dataset):** Programs describe what data operations are needed without specifying how to execute these operations
- **More efficient**: An optimizer can automatically find out the most efficient plan to execute a query

Spark SQL 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

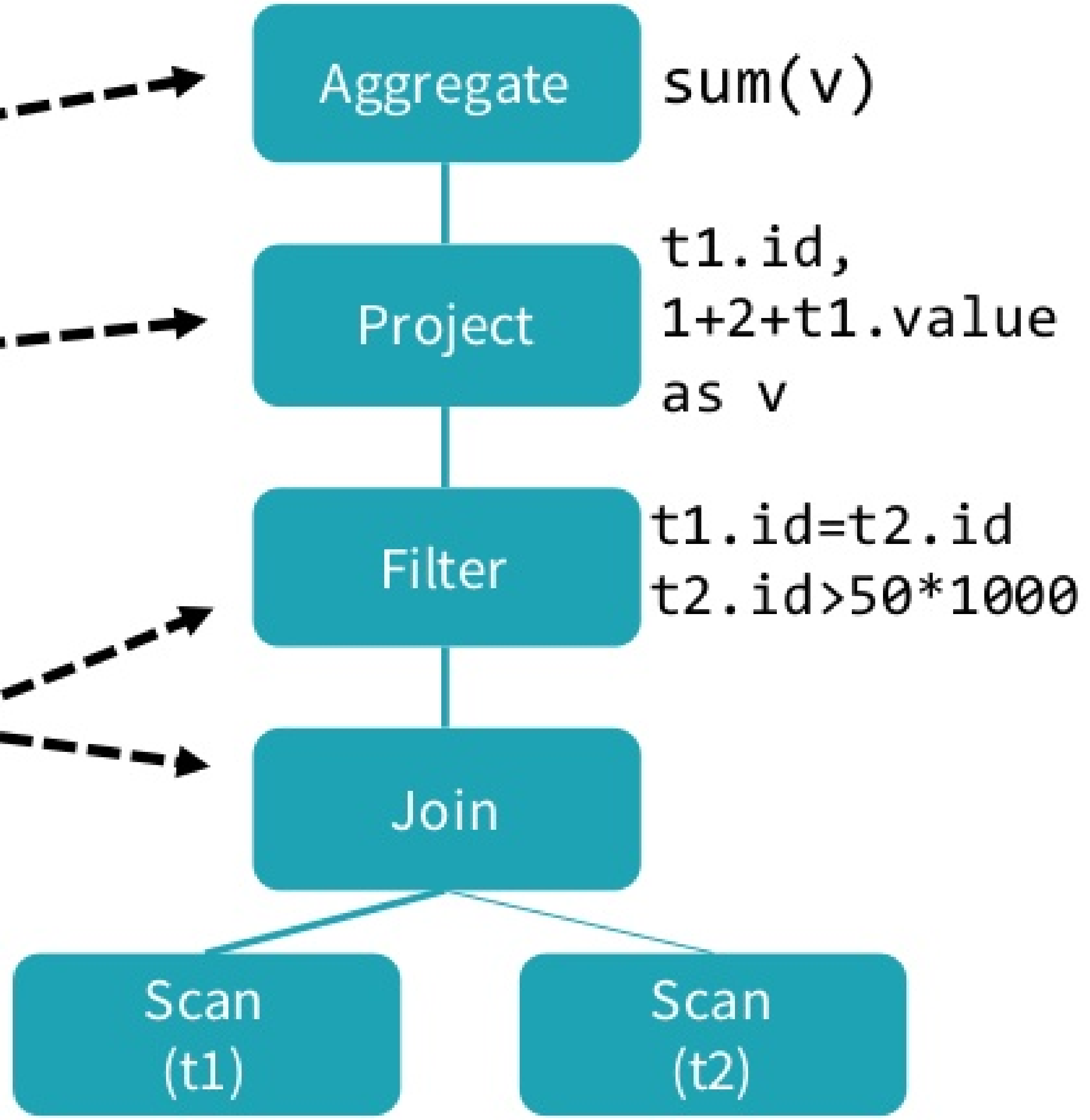
```
SELECT sum(v)
FROM (
  SELECT
    t1.id,
    1 + 2 + t1.value AS v
  FROM t1 JOIN t2
  WHERE
    t1.id = t2.id AND
    t2.id > 50 * 1000) tmp
```

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

Trees: Abstractions of Users' Programs

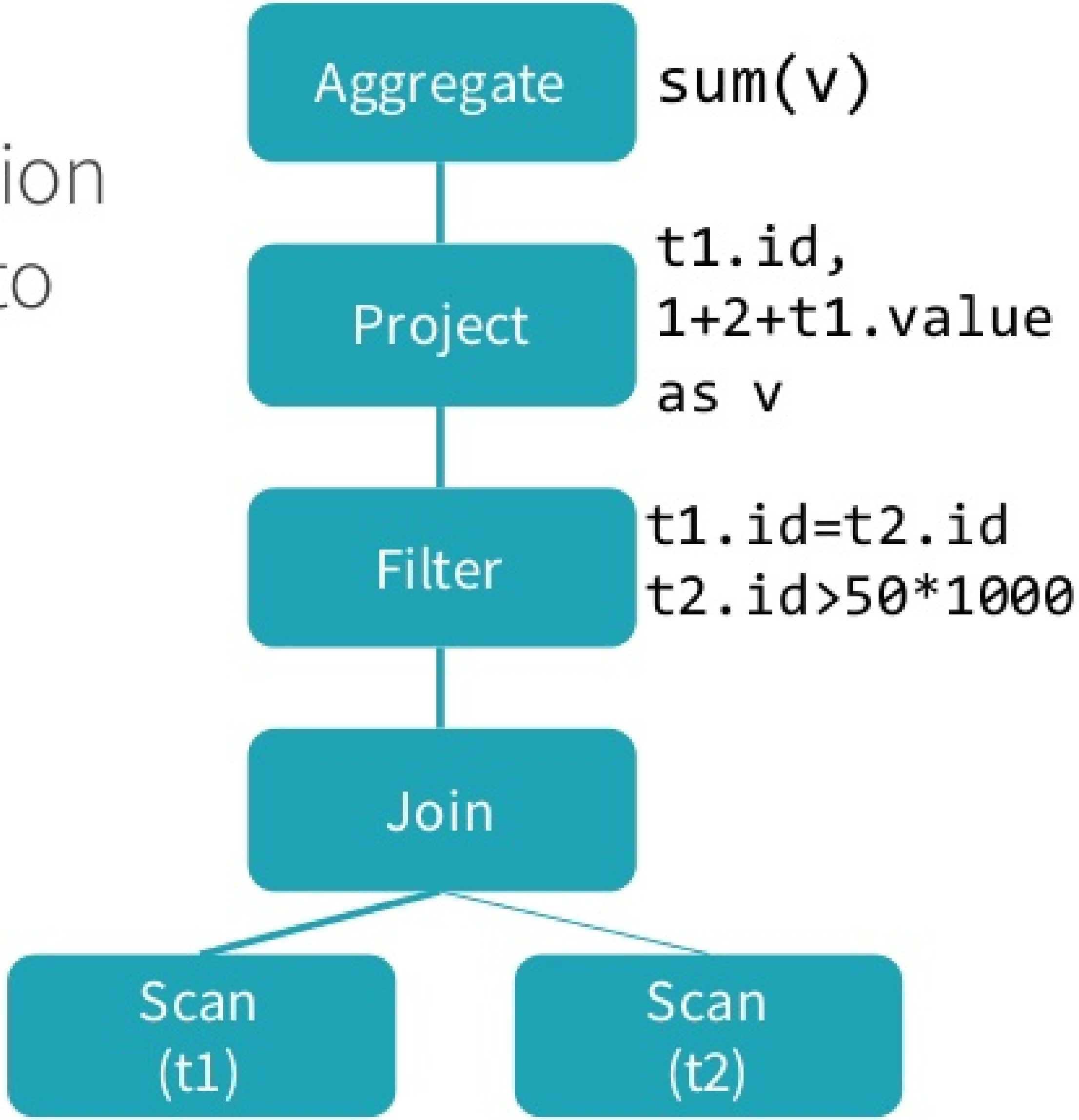
Query Plan

```
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

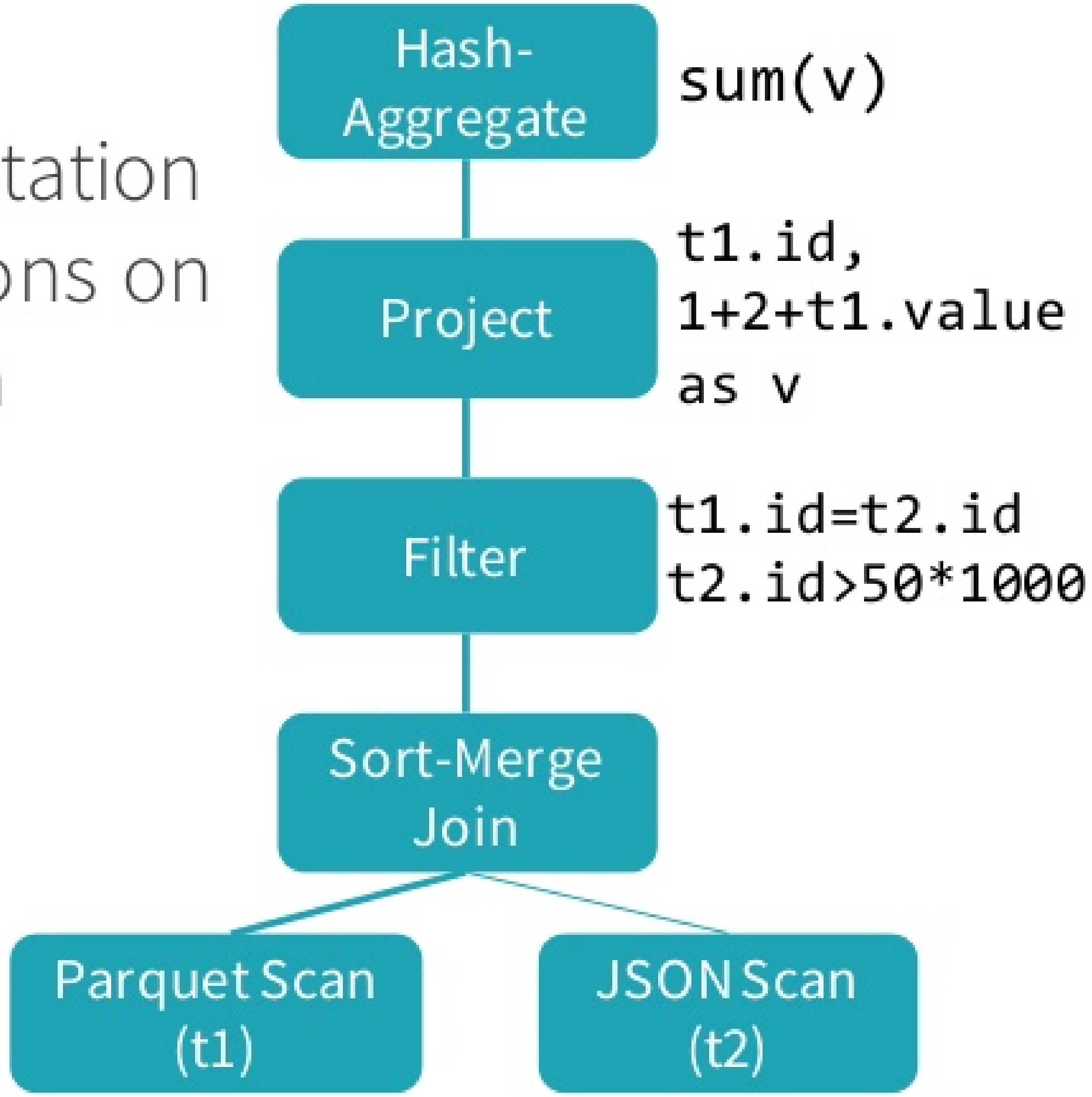
- A Logical Plan describes computation on datasets **without** defining how to conduct the computation



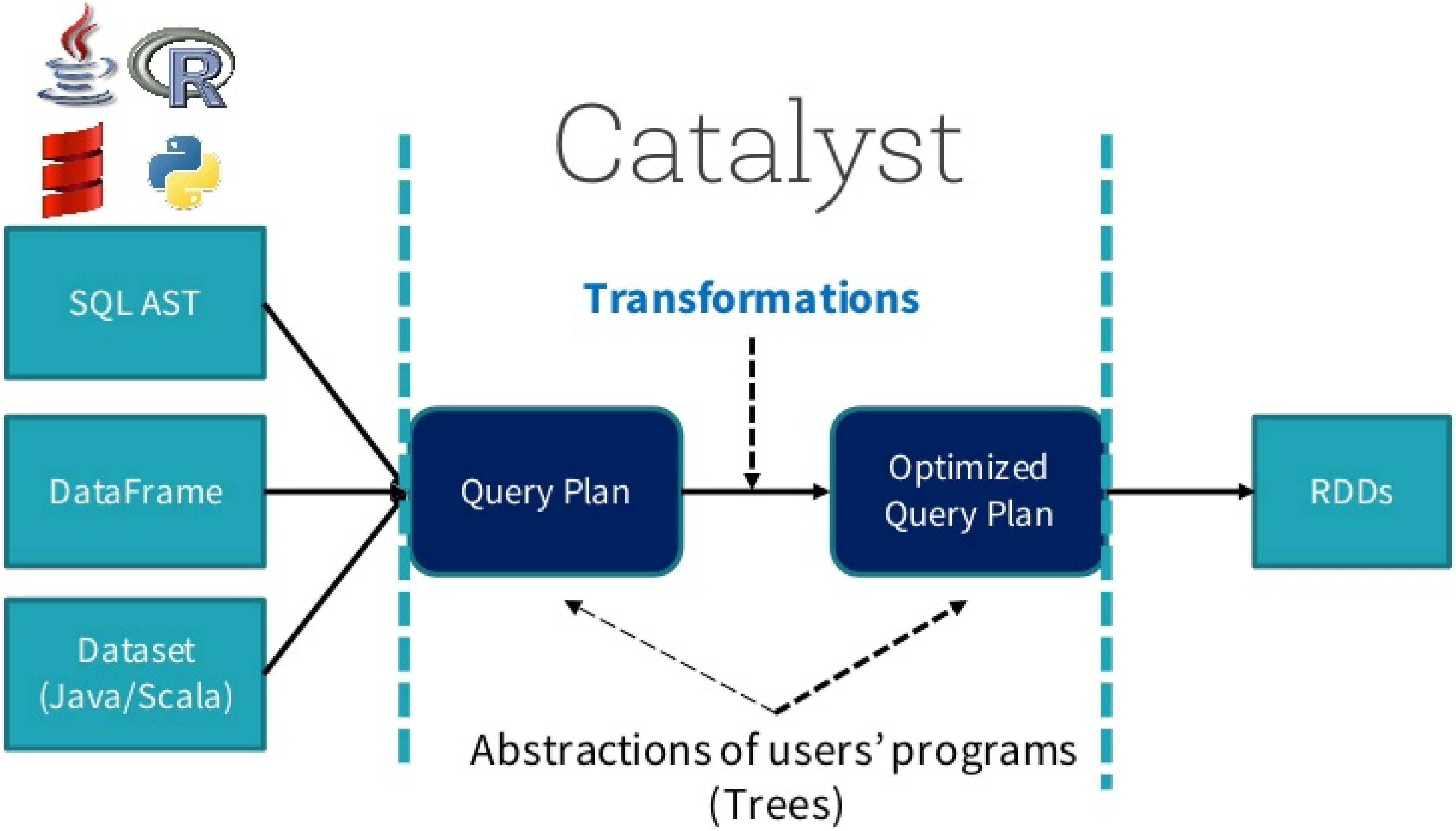
databricks

Physical Plan

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

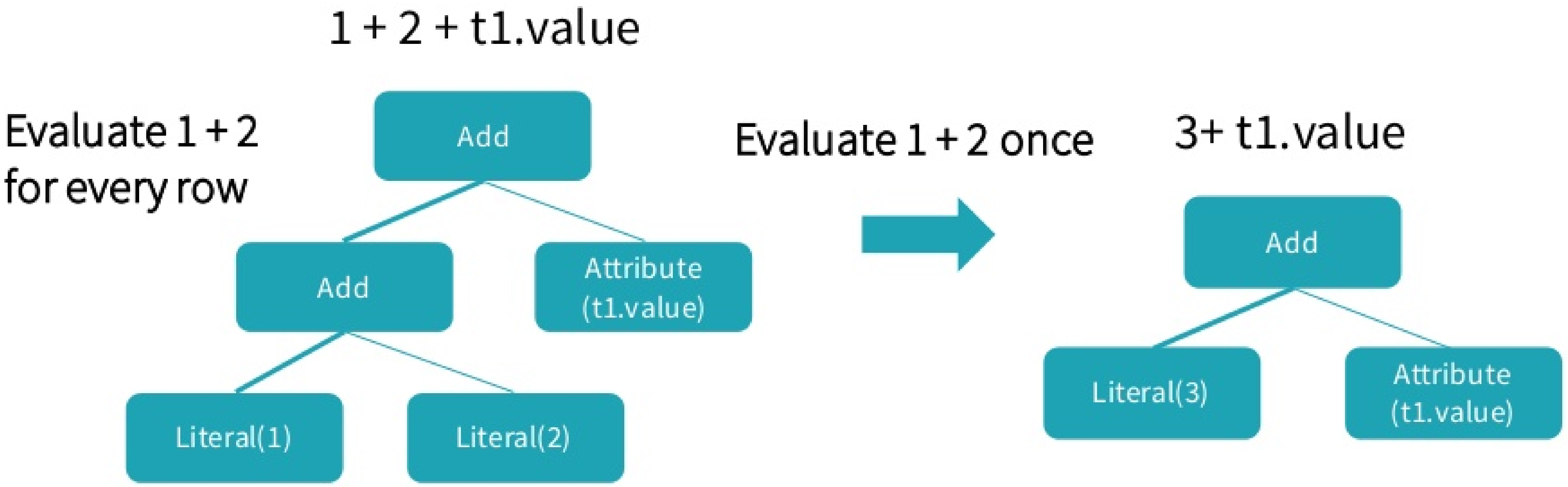


How Catalyst Works: An Overview



Transform

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



Transform

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

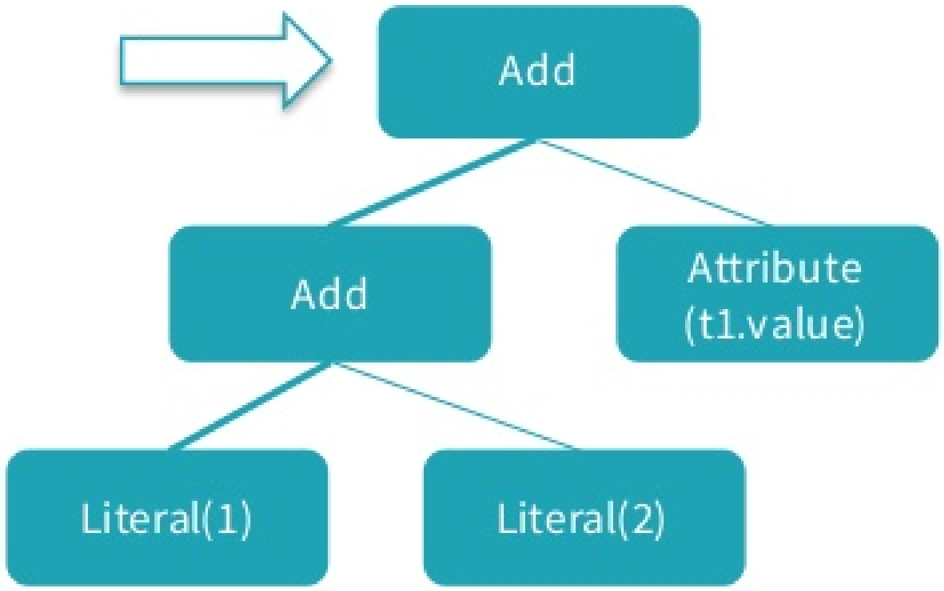
```
val expression: Expression = ...
expression.transform {
  case Add(Literal(x, IntegerType), Literal(y, IntegerType)) =>
    Literal(x + y)
}
```

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)
}
```

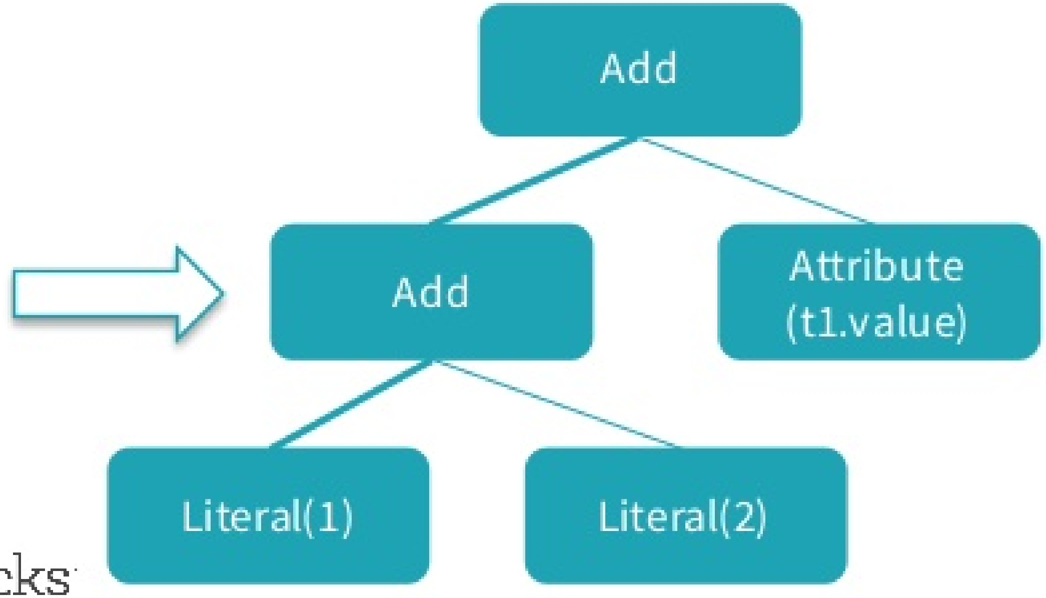
1 + 2 + t1.value



Transform

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

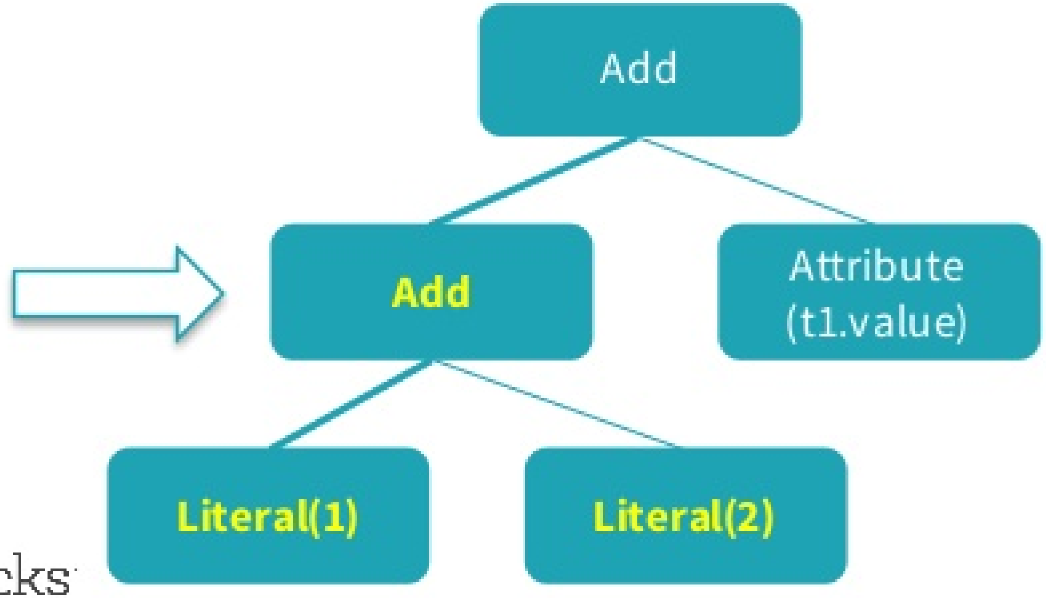
1 + 2 + t1.value



Transform

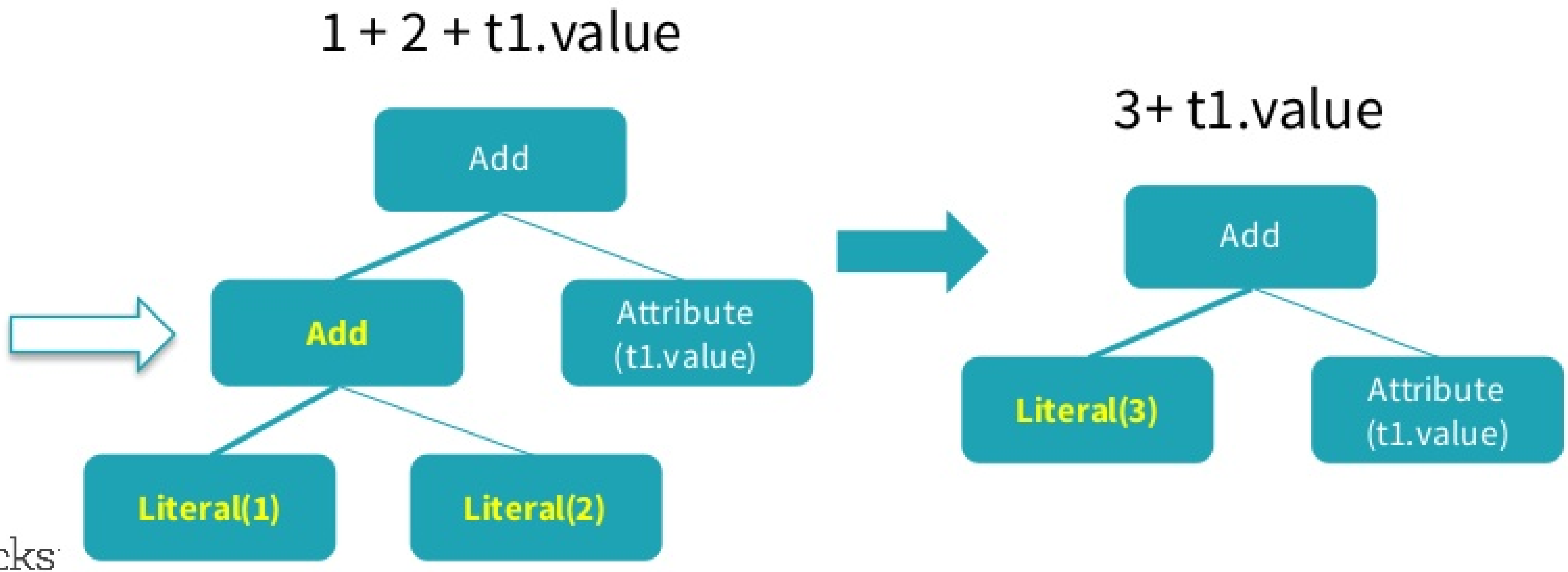
```
val expression: Expression = ...
expression.transform {
  case Add(Literal(x, IntegerType), Literal(y, IntegerType)) =>
    Literal(x + y)
}
```

1 + 2 + t1.value



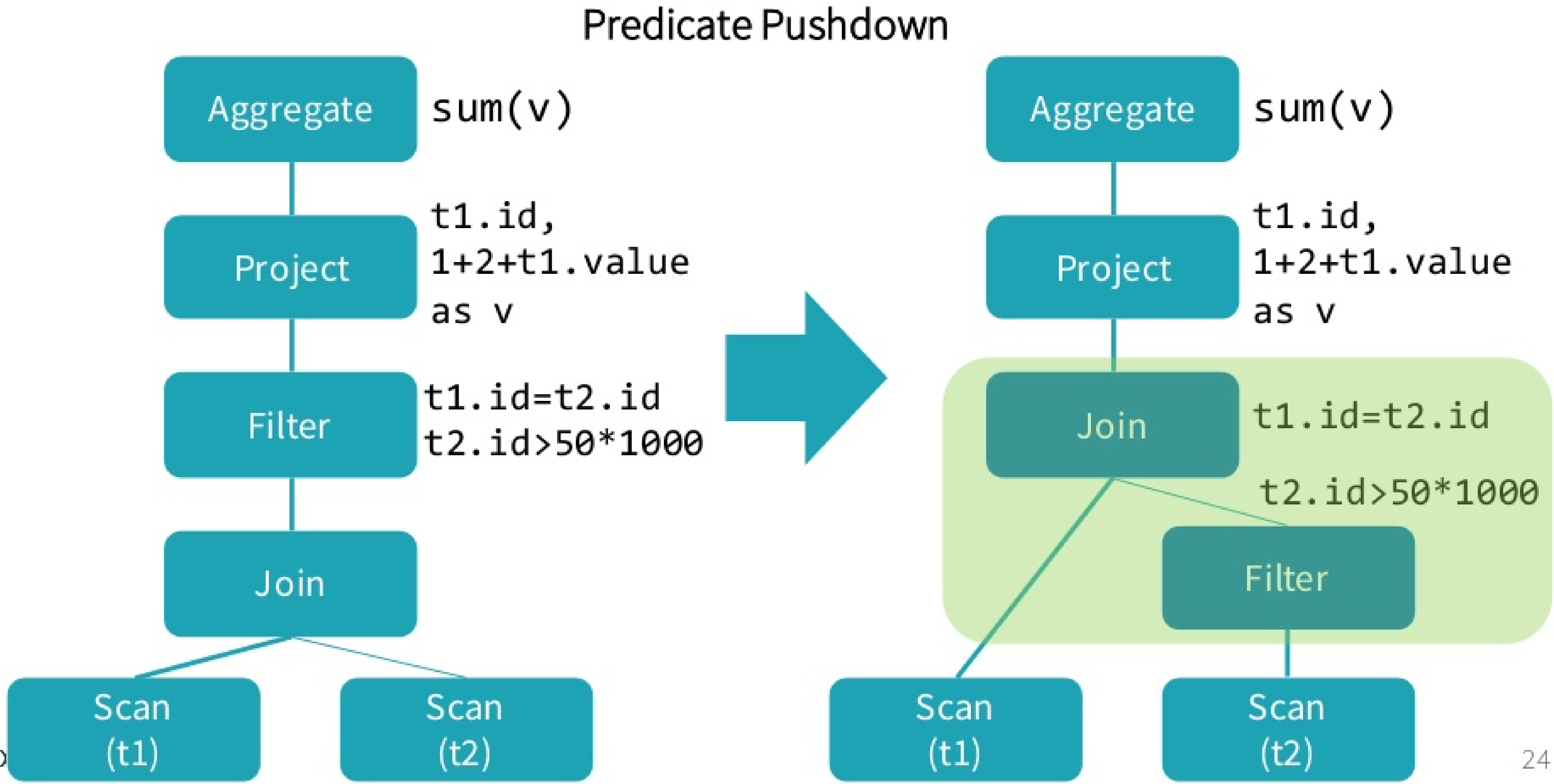
Transform

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

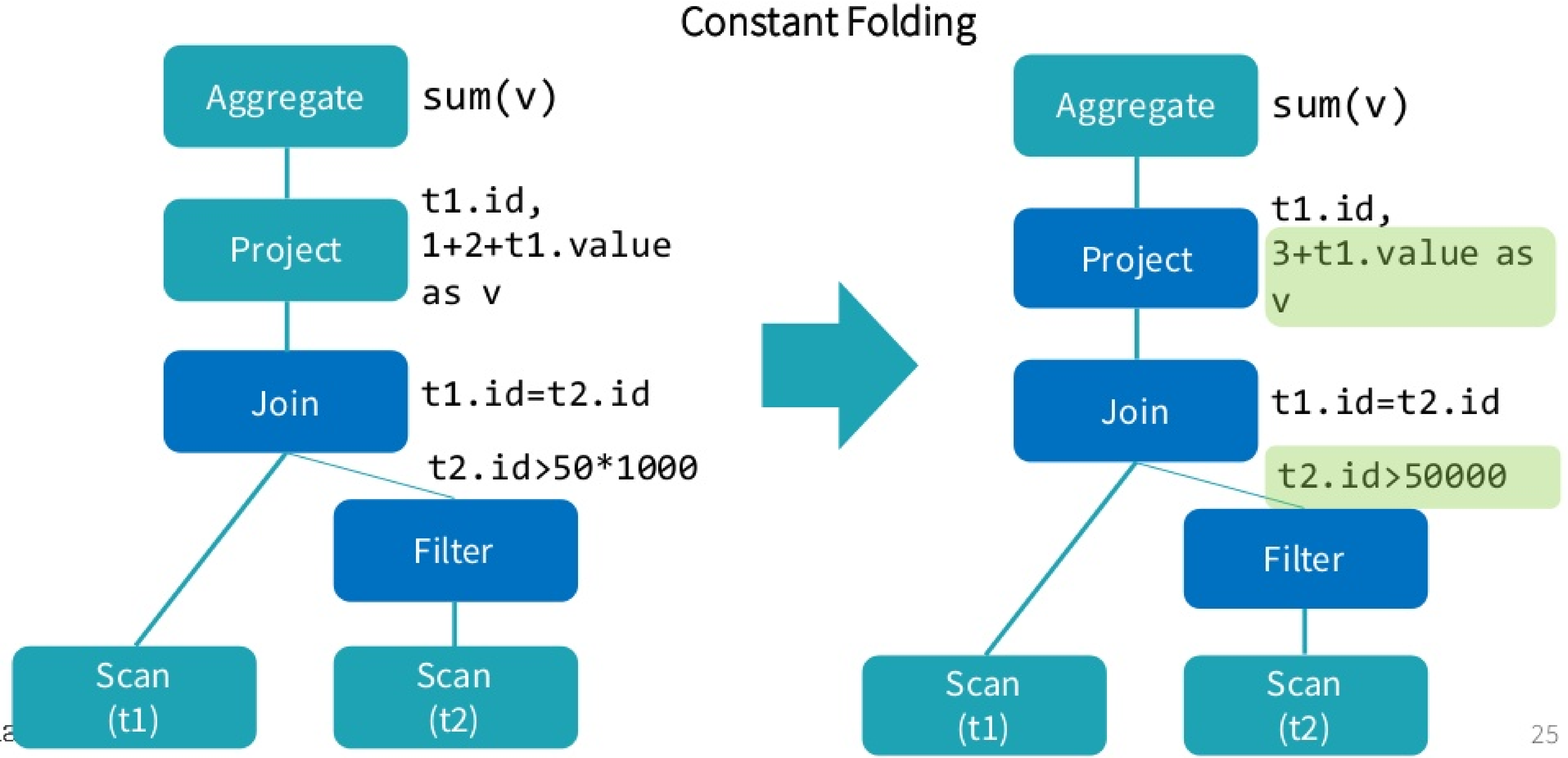


databricks

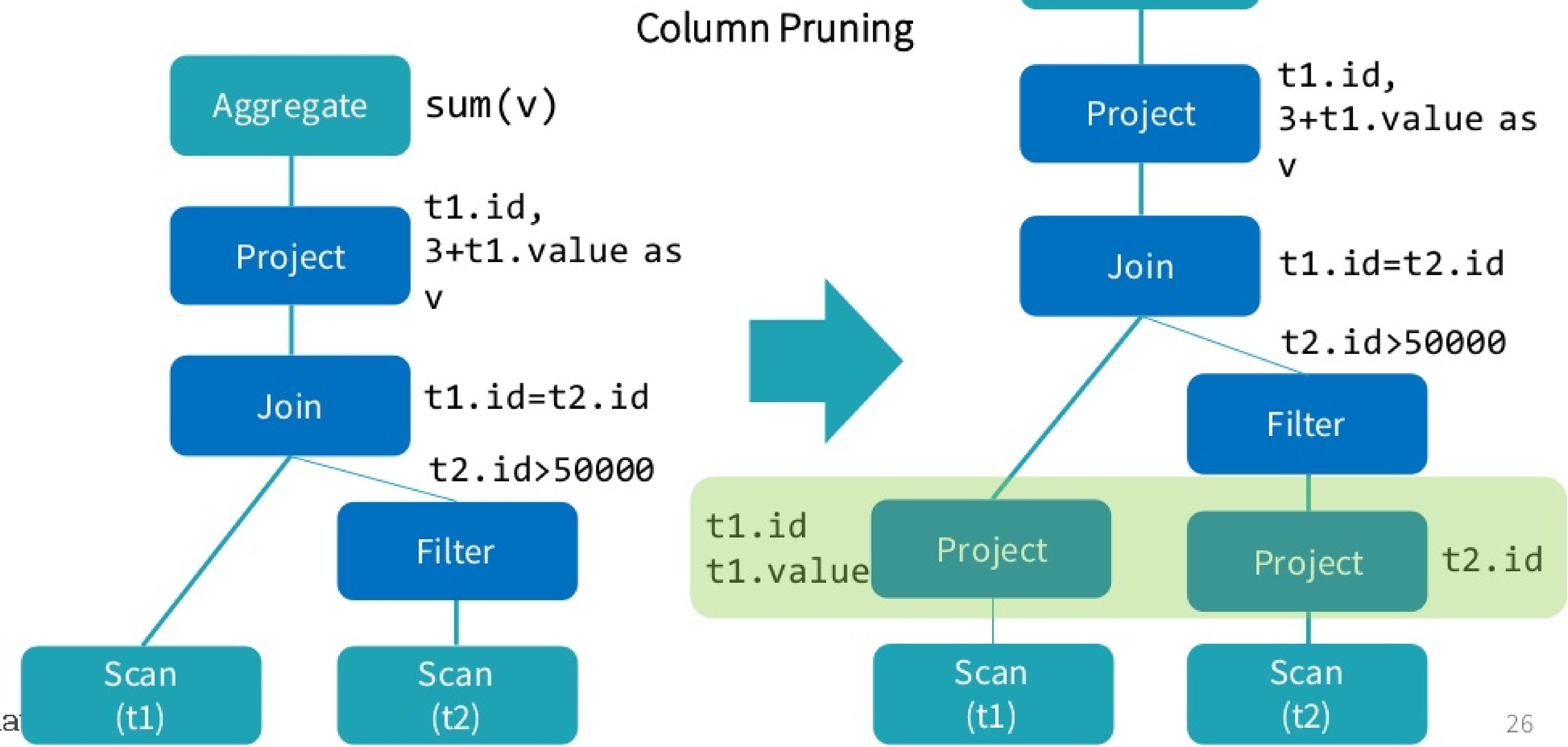
Combining Multiple Rules



Combining Multiple Rules

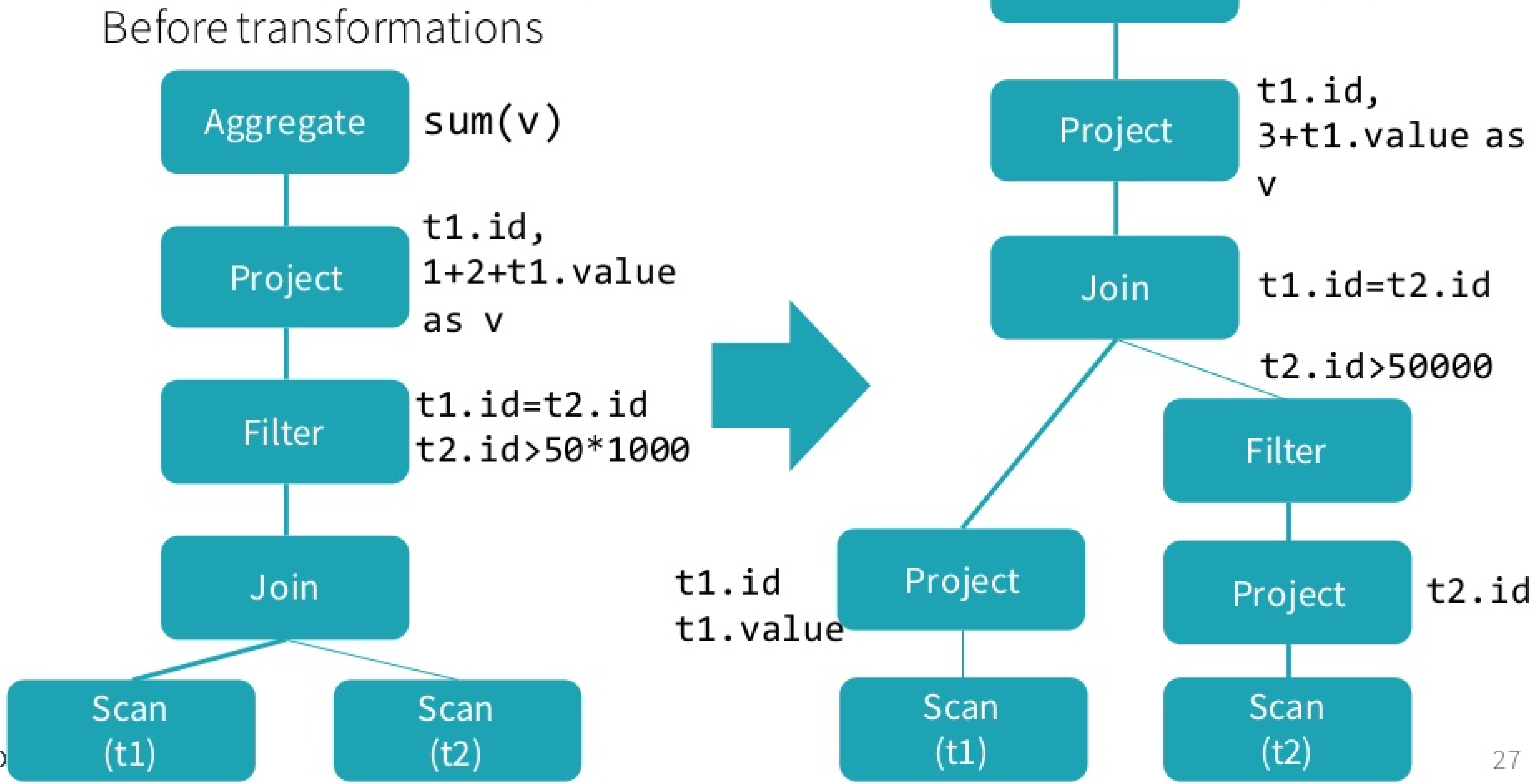


Combining Multiple Rules

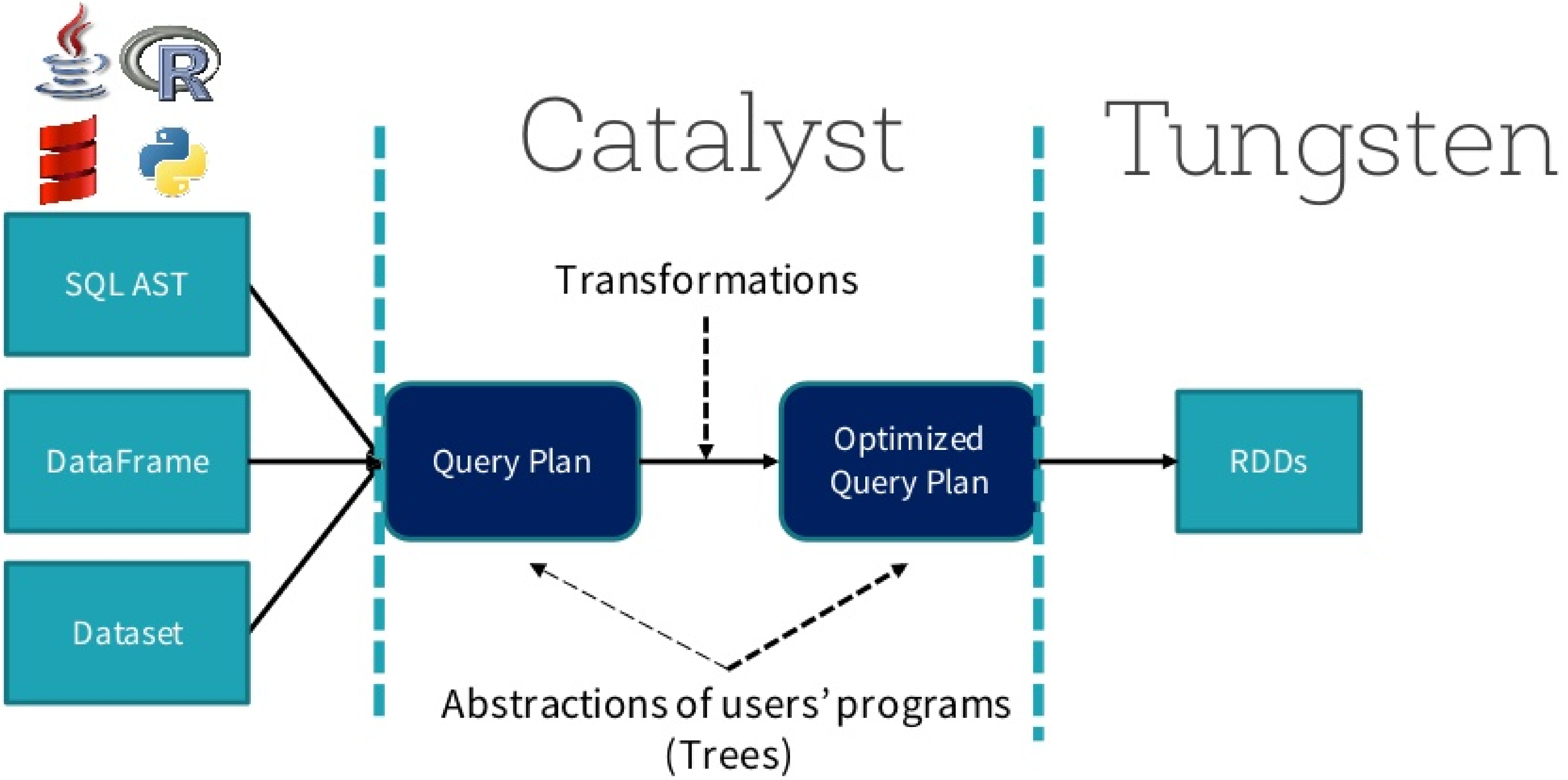


Combining Multiple Rules

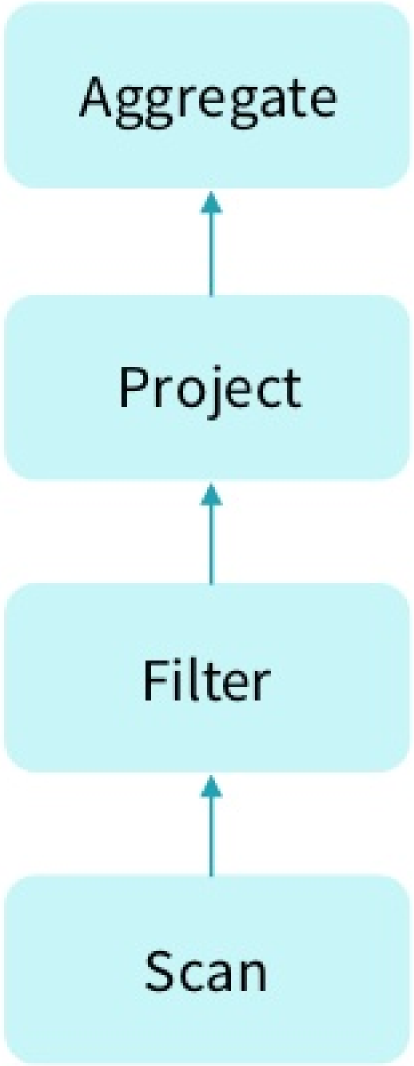
After transformations



Spark SQL Overview



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



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.

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

```
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
  }
}
```

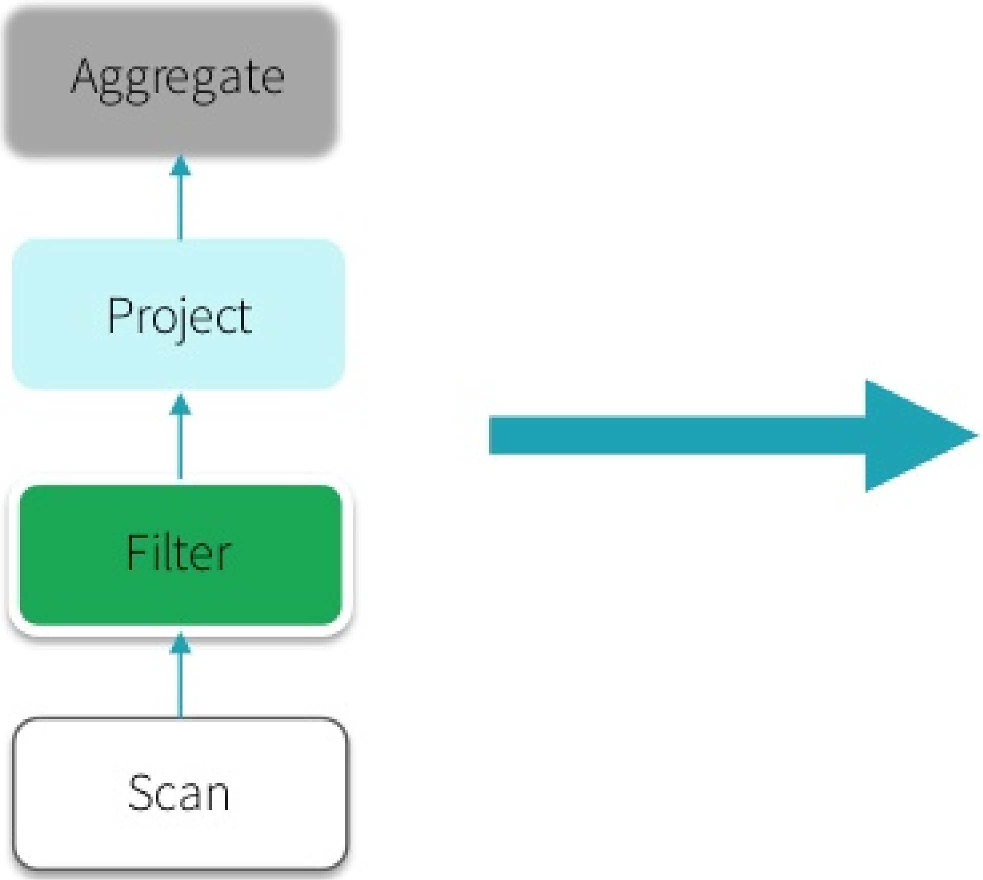
Downside of the Volcano Model

1. Too many virtual function calls
   - at least 3 calls for each row in Aggregate

2. Extensive memory access
   - "row" is a small segment in memory (or in L1/L2/L3 cache)

3. Can't take advantage of modern CPU features
   - SIMD, pipelining, prefetching, branch prediction, ILP, instruction cache, …

Whole-stage Codegen: Spark as a “Compiler”




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

Whole-stage Codegen

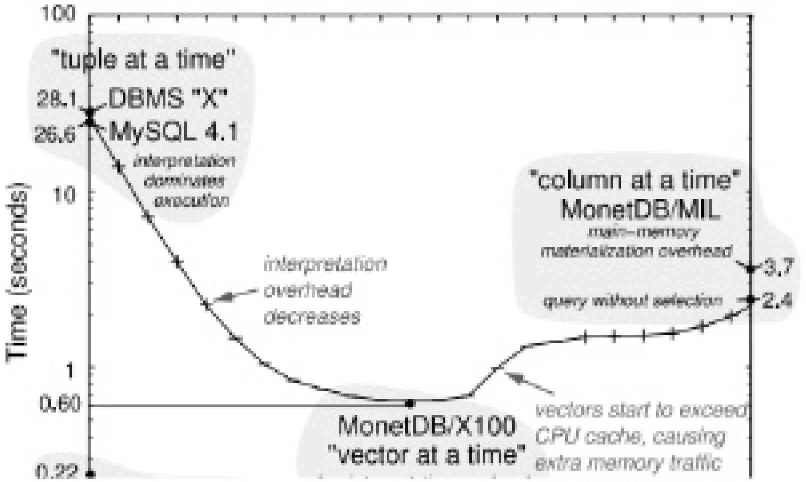
- Fusing operators together so the generated code looks like hand optimized code:
  - Identify chains of operators ("stages")
  - Compile each stage into a single function
  - Functionality of a general purpose execution engine; performance as if hand built system just to run your query

Efficiently Compiling Efficient Query Plans for Modern Hardware

Thomas Neumann
Technische Universität München
Munich, Germany
neumann@in.tum.de

ABSTRACT

As main memory grows, query performance is more and more determined by the raw CPU costs of query processing itself. The classical iterator style query processing technique is very simple and flexible, but shows poor performance on modern CPUs due to lack of locality and frequent instruction mis-predictions. Several techniques like batch oriented processing or vectorized tuple processing have been proposed in the past to improve this situation, but even these techniques are



T Neumann, Efficiently compiling efficient query plans for modern hardware. *In VLDB 2011*

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

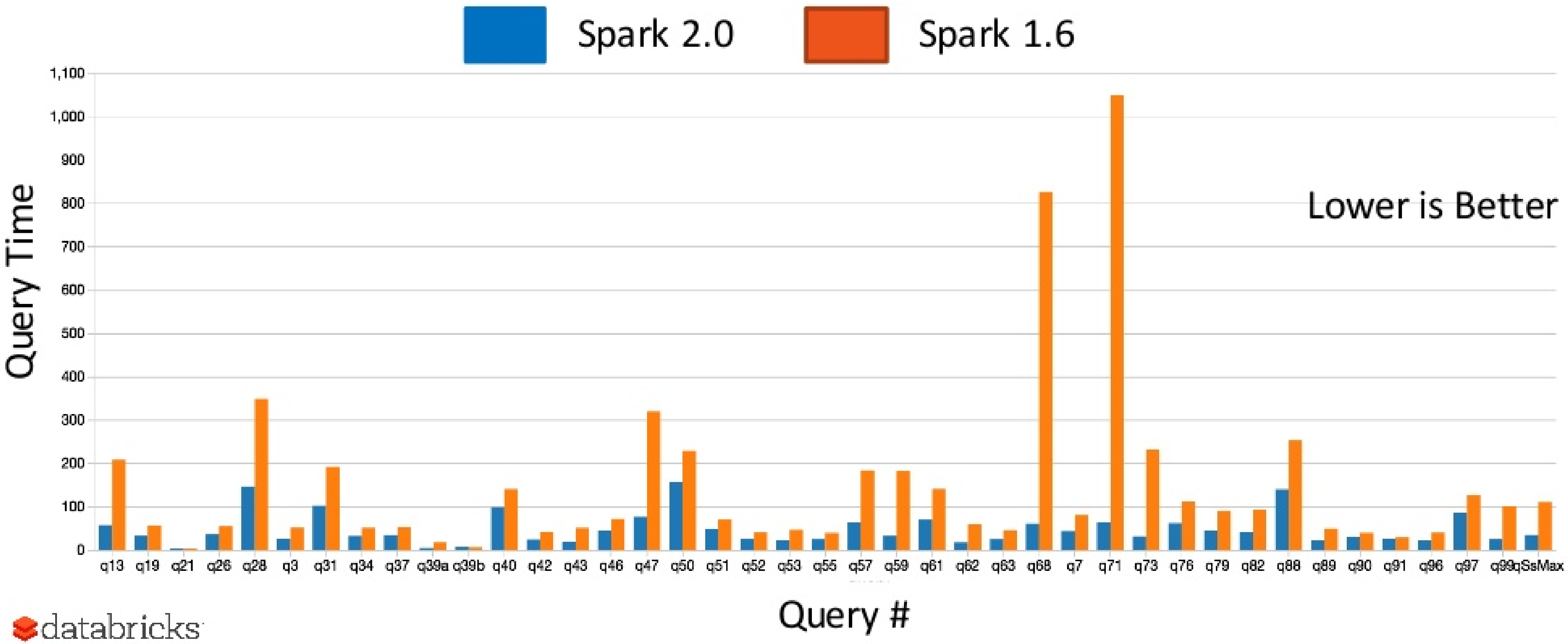
Operator Benchmarks: Cost/Row (ns)

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)



What’s Next?

Spark 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

Thank you.