Relational Operators

Ricardo Vilaça
HASLab
University of Minho
rmvilaca@di.uminho.pt

Roadmap

- Relational Operators in MapReduce
- SparkSQL Catalyst

MapReduce Implementations of relational operators

- Select and Project can be easily implemented in the map function
- Aggregation is not difficult
- Join requires more work

Select

- Selections do not need a full-blown MapReduce implementation
  - They can be implemented in the map phase alone
  - Or could also be implemented in the reduce portion
- Map
  - For each tuple t in R, check if t satisfies C
  - If so, emit a key/value pair (t; t)
- Reduce
  - Identity reducer

Select B <= 3

Map Worker 1

File 1	
A	***B***
1	2
2	3
5	6

File 2	
A	***B***
2	8
4	4
6	1

Map Worker 2

File 1	
A	***B***
6	2
6	3
7	6

File 2	
A	***B***
9	8
3	3
0	1

Select B <= 3

Map Worker 1

key	***value***
(1,2)	(1,2)
(2,3)	(2,3)
(6,1)	(6,1)

Map Worker 2

key	***value***
(6,2)	(6,2)
(6,3)	(6,3)
(3,3)	(3,3)
(0,1)	(0,1)

Select B <= 3

Map Worker 1

RW1	
key	***value***
(1,2)	(1,2)
(2,3)	(2,3)

RW2	
key	***value***
(6,1)	(6,1)

Map Worker 2

RW1	
key	***value***
(3,3)	(3,3)
(0,1)	(0,1)

RW2	
key	***value***
(6,2)	(6,2)
(6,3)	(6,3)

Select B <= 3

Reduce Worker 1

RW1	
key	**value**
(1,2)	(1,2)
(2,3)	(2,3)

RW1	
key	**value**
(3,3)	(3,3)
(0,1)	(0,1)

Reduce Worker 2

RW2	
key	**value**
(6,1)	(6,1)

RW2	
key	**value**
(6,2)	(6,2)
(6,3)	(6,3)

Select B <= 3

Reduce Worker 1

File 1	
A	***B***
0	1
1	2
2	3
3	3

Reduce Worker 2

File 1	
A	***B***
6	1
6	2
6	3

Projection

- Similar process to selection
- Projection may cause same tuple to appear several times
- Map
  - For each tuple t in R, construct a tuple t0 by eliminating those components whose attributes are not in S
  - Emit a key/value pair (t0; t0)
- Reduce
  - For each key t0 produced by any of the Map tasks, fetch t0; [t0; … ; t0]
  - Emit a key/value pair (t0; t0)
- NOTE: the reduce operation is duplicate elimination

Projection (A,B)

Map Worker 1

File 1		
A	**B**	**C**
1	2	3
2	2	2
1	2	1

File 2		
A	**B**	**C**
4	2	1
6	8	4
3	2	2

Map Worker 2

File 1		
A	**B**	**C**
1	2	5
2	3	2
1	3	1

File 2		
A	**B**	**C**
3	2	1
6	8	9
3	4	2

Projection (A,B)

Map Worker 1

key	***value***
(1,2)	[(1,2),(1,2)]
(2,2)	[(2,2)]
(4,2)	[(4,2)]
(6,8)	[(6,8)]
(3,2)	[(3,2)]

Map Worker 2

key	***value***
(1,2)	[(1,2)]
(2,3)	[(2,3)]
(1,3)	[(1,3)]
(3,2)	[(3,2)]
(6,8)	[(6,8)]
(3,4)	[(3,4)]

Projection (A,B)

Map Worker 1

RW1	
key	***value***
(1,2)	[(1,2),(1,2)]
(2,2)	[(2,2)]
(4,2)	[(4,2)]

RW2	
key	***value***
(6,8)	[(6,8)]
(3,2)	[(3,2)]

Map Worker 2

RW1	
key	***value***
(1,2)	[(1,2)]
(2,3)	[(2,3)]
(1,3)	[(1,3)]

RW2	
key	***value***
(3,2)	[(3,2)]
(6,8)	[(6,8)]
(3,4)	[(3,4)]

Projection (A,B)

Reduce Worker 1

RW1	
key	***value***
(1,2)	[(1,2),(1,2)]
(2,2)	[(2,2)]
(4,2)	[(4,2)]

RW1	
key	***value***
(1,2)	[(1,2)]
(2,3)	[(2,3)]
(1,3)	[(1,3)]

Reduce Worker 2

RW2	
key	***value***
(6,8)	[(6,8)]
(3,2)	[(3,2)]

RW2	
key	***value***
(3,2)	[(3,2)]
(6,8)	[(6,8)]
(3,4)	[(3,4)]

Projection (A,B)

Reduce Worker 1

key	***value***
(1,2)	[(1,2),(1,2),(1,2)]
(1,3)	[(1,3)]
(2,2)	[(2,2)]
(2,3)	[(2,3)]
(4,2)	[(4,2)]

Reduce Worker 2

key	***value***
(3,2)	[(3,2),(3,2)]
(3,4)	[(3,4)]
(6,8)	[(6,8),(6,8)]

Projection (A,B)

Reduce Worker 1

File 1	
A	***B***
1	2
1	3
2	2
2	3
4	2

Reduce Worker 2

File 1	
A	***B***
3	2
3	4
6	8

Union

- Suppose relations R and S have the same schema
  - Map tasks will be assigned chunks from either R or S
  - Mappers don’t do much, just pass by to reducers
  - Reducers do duplicate elimination
- A MapReduce implementation of union
  - Map
    - For each tuple t in R or S, emit a key/value pair (t; t)
  - Reduce
    - For each key t there will be either one or two values
    - Emit (t; t) in either case

Intersection

- Very similar to computing unions
  - Suppose relations R and S have the same schema
  - The map function is the same (an identity mapper) as for union
  - The reduce function must produce a tuple only if both relations have that tuple
- A MapReduce implementation of intersection
  - Map
    - For each tuple t in R or S, emit a key/value pair (t; t)
  - Reduce
    - If key t has value list [t; t] then emit the key/value pair (t; t)
    - Otherwise, emit the key/value pair (t; NULL)

GroupBy A AGG(B)

- Let R(A;B;C)
  - The map operation prepares the grouping
  - The grouping is done by the framework
  - The reducer computes the aggregation
  - Simplifying assumptions: one grouping attribute and one aggregation function
- Map
  - For each tuple (a; b; c) emit the key/value pair (a; b)
- Reduce
  - Each key a represents a group
  - Apply AGG to the list [b1; b2; …; bn]
  - Emit the key/value pair (a; x) where x = AGG([b1; b2;…; bn])

GroupBy (A,B) Sum(C)

Map Worker 1

File 1			
A	**B**	**C**	**D**
1	2	3	1
2	2	3	2
1	2	1	3

File 2			
A	**B**	**C**	**D**
4	2	1	3
6	8	4	4
3	2	2	4

Map Worker 2

File 1			
A	**B**	**C**	**D**
1	2	5	2
2	3	2	4
1	3	1	3

File 2			
A	**B**	**C**	**D**
3	2	1	3
2	3	9	2
3	4	2	1

GroupBy (A,B) Sum(C)

Map Worker 1

key	***value***
(1,2)	[3,1]
(2,2)	[3]
(4,2)	[1]
(6,8)	[4]
(3,2)	[2]

Map Worker 2

key	***value***
(1,2)	[5]
(2,3)	[2,9]
(1,3)	[1]
(3,2)	[1]
(3,4)	[2]

GroupBy (A,B) Sum(C)

Map Worker 1

RW1	
key	***value***
(1,2)	[3,1]
(2,2)	[3]
(4,2)	[1]

RW2	
key	***value***
(6,8)	[4]
(3,2)	[2]

Map Worker 2

RW1	
key	***value***
(1,2)	[5]
(2,3)	[2,9]

RW2	
key	***value***
(3,2)	[1]
(3,4)	[2]
(1,3)	[1]

GroupBy (A,B) Sum(C)

Reduce Worker 1

RW1	
key	***value***
(1,2)	[3,1]
(2,2)	[3]
(4,2)	[1]

RW1	
key	***value***
(1,2)	[5]
(2,3)	[2,9]

Reduce Worker 2

RW2	
key	***value***
(6,8)	[4]
(3,2)	[2]

RW2	
key	***value***
(3,2)	[1]
(3,4)	[2]
(1,3)	[1]

GroupBy (A,B) Sum(C)

Reduce Worker 1

key	***value***
(1,2)	[3,1,5]
(2,2)	[3]
(2,3)	[2,9]
(4,2)	[1]

Reduce Worker 2

key	***value***
(1,3)	[1]
(3,2)	[1,2]
(3,4)	[2]
(6,8)	[4]

GroupBy (A,B) Sum(C)

Reduce Worker 1

A	*B*	*Sum*
1	2	9
2	2	3
2	3	11
4	2	1

Reduce Worker 2

A	*B*	*Sum*
1	3	1
3	2	3
3	4	2
6	8	4

Natural Join

- Let's look at two relations R(A;B) and S(B;C)
  - We must find tuples that agree on their B components
  - We shall use the B-value of tuples from either relation as the key
  - The value will be the other component and the name of the relation
  - That way the reducer knows from which relation each tuple is coming from
- A MapReduce implementation of Natural Join
  - Map
    - For each tuple (a; b) of R emit the key/value pair (b; (R; a))
    - For each tuple (b; c) of S emit the key/value pair (b; (S; c))
  - Reduce
    - Each key b will be associated to a list of pairs that are either (R; a) or (S; c)
    - Emit key/value pairs of all possible combinations for the values where one value is from table R and the other value is from table S

Natural Join

Map Worker 1

Table 1	
A	***B***
1	2
2	3
5	6

Table 2	
B	***C***
2	3
4	4
6	1

Map Worker 2

Table 1	
A	***B***
6	1
6	3
7	6

Table 2	
B	***C***
9	8
3	4
2	1

Natural Join

Map Worker 1

key	***value***
2	[(T1,1), (T2,3)]
3	[(T1,2)]
6	[(T1,5), (T2,1)]
4	[(T1,4)]

Map Worker 2

key	***value***
1	[(T1,6)]
3	[(T1,6),(T2,4)]
6	[(T1,7)]
9	[(T2,8)]
2	[(T2,1)]

Natural Join

Map Worker 1

RW1	
key	***value***
2	[(T1,1),(T2,3)]
3	[(T1,2)]

RW2	
key	***value***
6	[(T1,5), (T2,1)]
4	[(T1,4)]

Map Worker 2

RW1	
key	***value***
1	[(T1,6)]
3	[(T1,6),(T2,4)]
2	[(T2,1)]

RW2	
key	***value***
6	[(T1,7)]
9	[(T2,8)]

Natural Join

Reduce Worker 1

RW1	
key	***value***
2	[(T1,1),(T2,3)]
3	[(T1,2)]

RW1	
key	***value***
1	[(T1,6)]
3	[(T1,6),(T2,4)]
2	[(T2,1)]

Reduce Worker 2

RW2	
key	***value***
6	[(T1,5),(T2,1)]
4	[(T1,4)]

RW2	
key	***value***
6	[(T1,7)]
9	[(T2,8)]

Natural Join

Reduce Worker 1

RW1	
key	***value***
1	[(T1,6)]
2	[(T1,1), (**T2**,3), (**T2**,1)]
3	[(T1,2), (T1,6),(**T2**,4)]

Reduce Worker 2

RW2	
key	***value***
6	[(T1,5), (**T2**,1), (T1,7)]
4	[(T1,4)]
9	[(**T2**,8)]

Natural Join

Reduce Worker 1

B	**A**	**C**
2	1	3
2	1	1
3	2	4
3	6	4

Reduce Worker 2

B	**A**	**C**
6	5	1
6	7	1

Other joins

- Applied to other complex operators such as duplicate elimination, union, intersection, etc. with minor adaptation

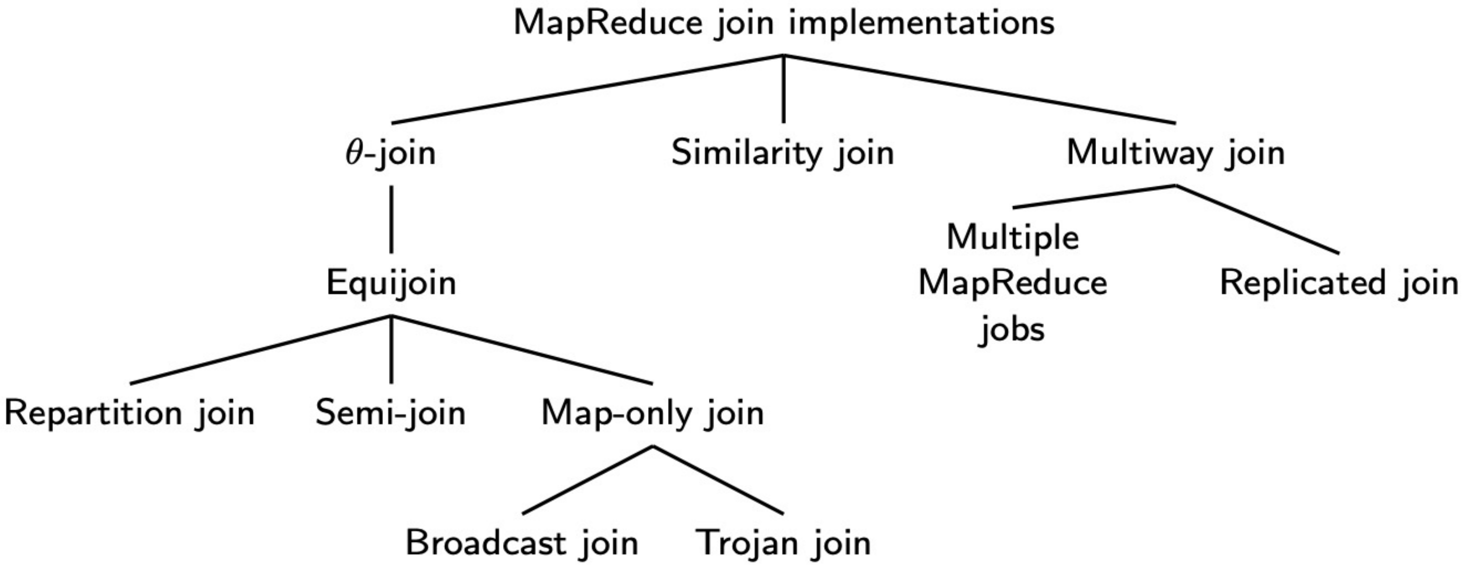


Image from M.T. Özsu & P. Valduriez (2020). Principles of Distributed Database Systems

Spark Joins

- **SortMergeJoin**, ShuffleHashJoin, and BroadcastHashJoin
- SortMergejoin is composed of 2 steps
  - Sort the datasets
  - Merge the sorted data in the partition by iterating over the elements and according to the join key join the rows having the same value.
- BroadcastHashJoin
  - Optimum performance can be achieved
  - Strict limitations with the size of data frames
    - spark.sql.autoBroadcastJoinThreshold=10MB
  - Solves uneven sharding and limited parallelism
- ShuffleHashJoin
  - MapReduce based, similar to natural join

Catalyst

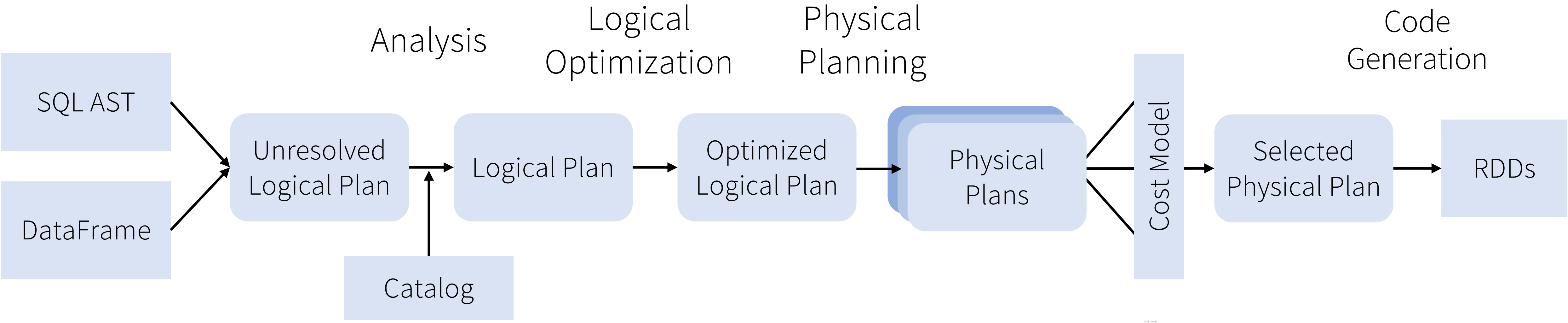
- Goals
  - Optimize logical plan
  - Convert logical to physical plan
  - Optimize physical plan
  - Code generation
- Scala language features
  - Pattern matching
  - Quasiquotes
  - Abstract syntax tree
  - Tree manipulation library
  - Optimizations rules implemented as tree transformations

Extensibility

- Easily add new optimization techniques and features
- Enable external developers to extend the optimizer
  - e.g. adding data source specific rules, support for new data types, etc.
  - Data sources, E.g. CSV, Avro, Parquet, JDBC
  - Map user-defined types to structures composed of Catalyst’s built-in types.

```
class PointUDT extends UserDefinedType[Point] {
  def dataType = StructType(Seq( // Our native structure
    StructField("x", DoubleType),
    StructField("y", DoubleType)
  ))
  def serialize(p: Point) = Row(p.x, p.y)
  def deserialize(r: Row) =
    Point(r.getDouble(0), r.getDouble(1))
}
```

Plan Optimization & Execution



DataFrames and SQL share the same optimization/execution pipeline

Image from DataBricks SparkSQL presentation@SIGMOD 2015

Analysis

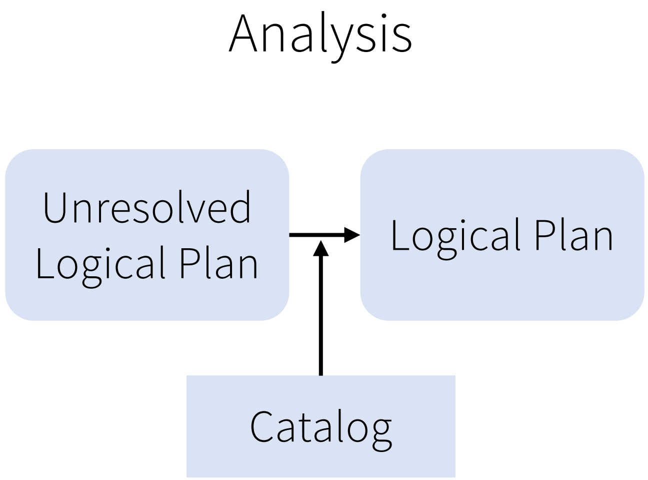


Image from DataBricks SparkSQL presentation@SIGMOD 2015

- 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 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*)

Logical Optimization

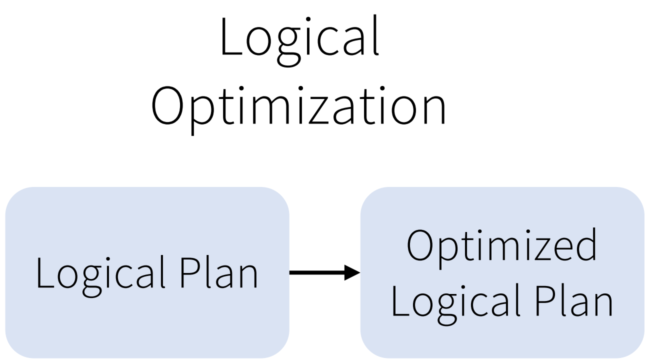


Image from DataBricks SparkSQL presentation@SIGMOD 2015

- Applies standard rule-based optimization (constant folding, predicate-pushdown, projection pruning, null propagation, boolean expression simplification, etc)

```
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)
    }
}
```

Physical Planning

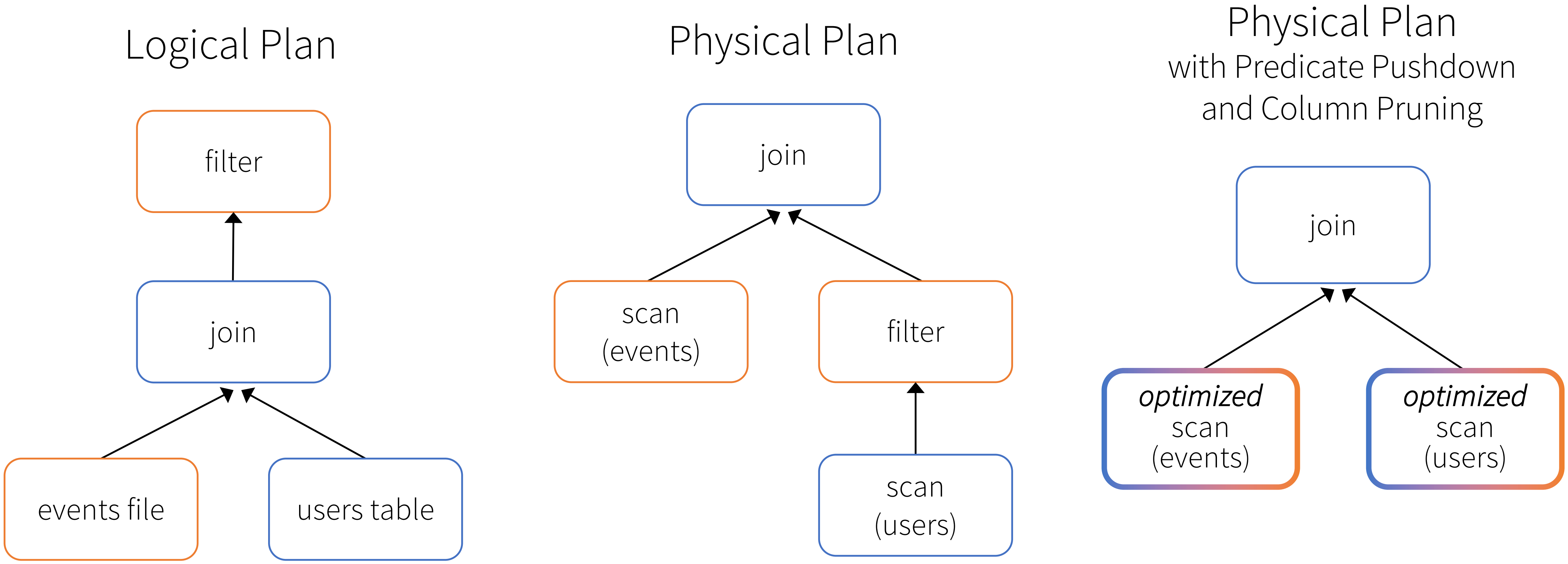


Image from DataBricks SparkSQL presentation@SIGMOD 2015

Code Generation

```
def compile(node: Node): AST = node match {
  case Literal(value) => q"$value"
  case Attribute(name) => q"row.get($name)"
  case Add(left, right) =>
    q"${compile(left)} + ${compile(right)}"
}
```

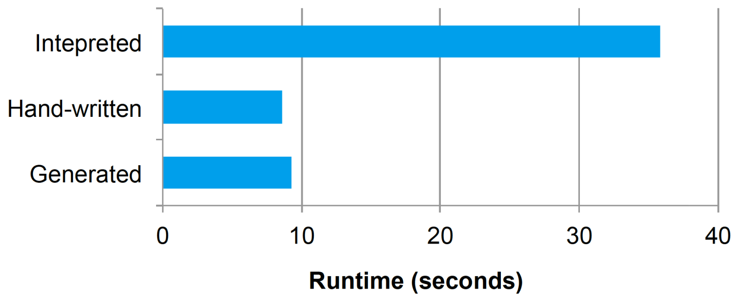


Figure 4: A comparision of the performance evaluating the expresion x+x+x, where x is an integer, 1 billion times.

Images from DataBricks SparkSQL presentation@SIGMOD 2015

- Relies on Scala's quasiquotes to simplify code gen.
- Generating Java bytecode to run on each machine
- Whole-Stage CodeGen
  - Joins multiple physical operations together to form a single Java function
  - Leverages CPU registers for intermediate data

More information

- Mining of Massive Datasets, Anand Rajaraman and Jeff Ullman, Cambridge University Press, Section 2.3

- SparkSQL: Relational Data Processing in Spark, M. Armbrust, et al., SIGMOD, 2015