# Information Engineering 2

## Spark Query Optimization

Prof. Dr. Kurt Stockinger

# Semesterplan

| SW | Datum | Vorlesungsthema | Praktikum |
|---|---|---|---|
| 1 | 23.02.2022 | Data Warehousing Einführung | Praktikum 1: KNIME Tutorial |
| 2 | 02.03.2022 | Dimensionale Datenmodellierung 1 | Praktikum 1: KNIME Tutorial (Vertiefung) |
| 3 | 09.03.2022 | Dimensionale Datenmodellierung 2 | Praktikum 2: Datenmodellierung |
| 4 | 16.03.2022 | Datenqualität und Data Matching | Praktikum 3: Star-Schema, Bonus: Praktikum 4: Slowly Changing Dimensions |
| 5 | 23.03.2022 | Big Data Einführung | DWH Projekt - Teil 1 |
| 6 | 30.03.2022 | Spark - Data Frames | DWH Projekt - Teil 2 (Abgabe: 4.4.2022 23:59:59) |
| 7 | 06.04.2022 | Data Storage: Hadoop Distributed File System & Parquet | Praktikum 1: Data Frames |
| 8 | 13.04.2022 | Query Optimization | Praktikum 2: Data Storage |
| 9 | 20.04.2022 | Spark Best Practices & Applications | Praktikum 3: Query Optimization & Performance Analysis |
| 10 | 27.04.2022 | Machine Learning mit Spark 1 | Praktikum 3: Query Optimization & Performance Analysis (Vertiefung) |
| 11 | 04.05.2022 | Machine Learning mit Spark 2 + Q&A | Praktikum 4: Machine Learning (Regression) |
| 12 | 11.05.2022 | NoSQL Systems | Big Data Projekt - Teil 1 |
| 13 | 18.05.2022 | Keine Vorlesung (Arbeit am Projekt) | Big Data Projekt - Teil 2 |
| 14 | 25.05.2022 | Keine Vorlesung (Arbeit am Projekt) | Big Data Projekt - Teil 3 (Abgabe: 30.5.2022 23:59:59) |

# Educational Objectives for Today

- Learn about internals of query processing
- Understand concepts of logical and physical query plan
- Understand and apply different query plans for joins
- Get better intuition about query performance

# Let Us Analyze a Query

SELECT name, AVG(age)

FROM department

WHERE location = ’Switzerland’

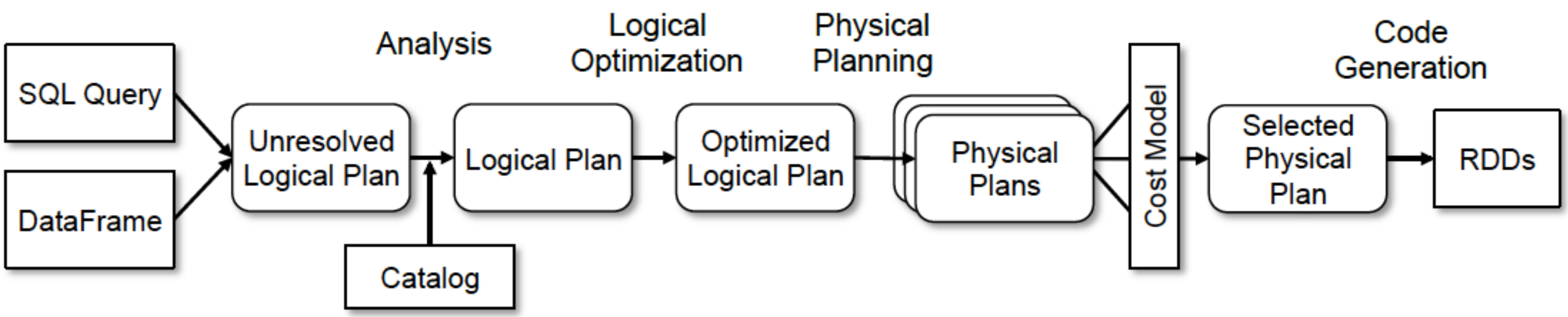
GROUP BY name

- Which steps are required to execute this query?

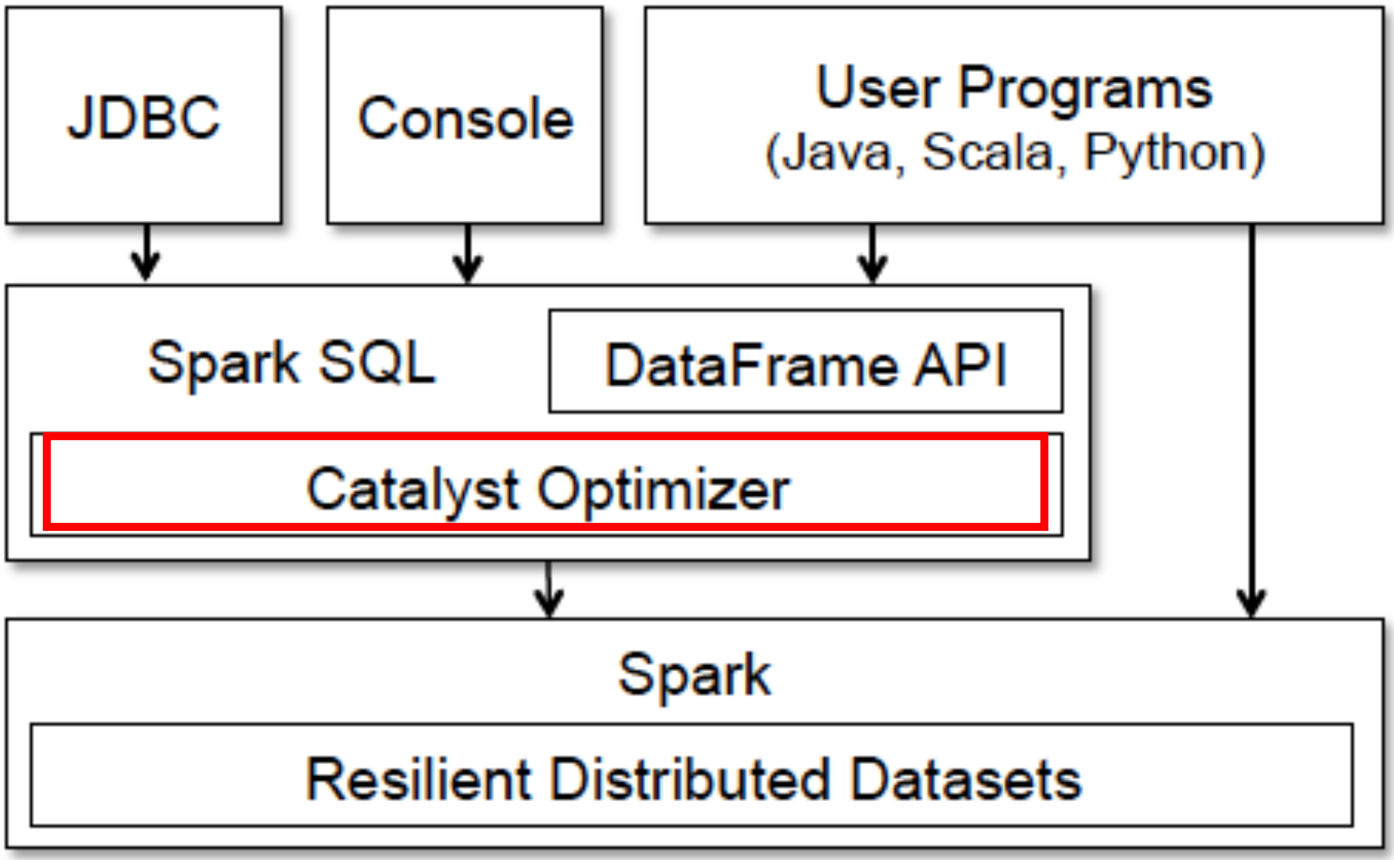
```
root
 |-- name: string (nullable = true)
 |-- age: long (nullable = true)
 |-- department: string (nullable = true)
 |-- location: string (nullable = true)
```

```
+----------+-----------------+
|department|         avg(age)|
+----------+-----------------+
|         B|             28.5|
|         C|             65.0|
|         A|32.66666666666664|
+----------+-----------------+
```

# Phases of Query Planning



# Interfaces to Spark SQL



Zurich University
of Applied Sciences

# Spark DataFrames as SQL
# The Tree Abstraction

## Trees: Abstractions of Users' Programs

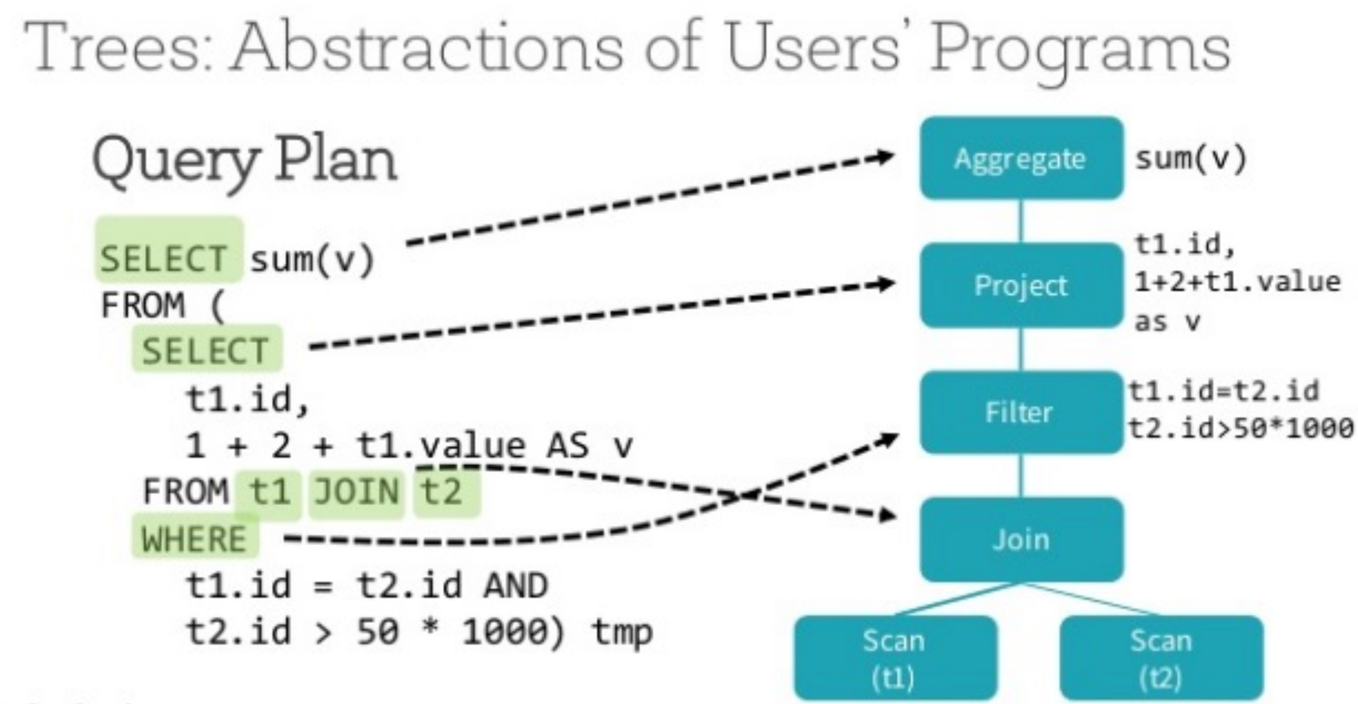
### Expression

```
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
```

Which steps does this query contain?

# Spark DataFrames as SQL
# The Tree Abstraction

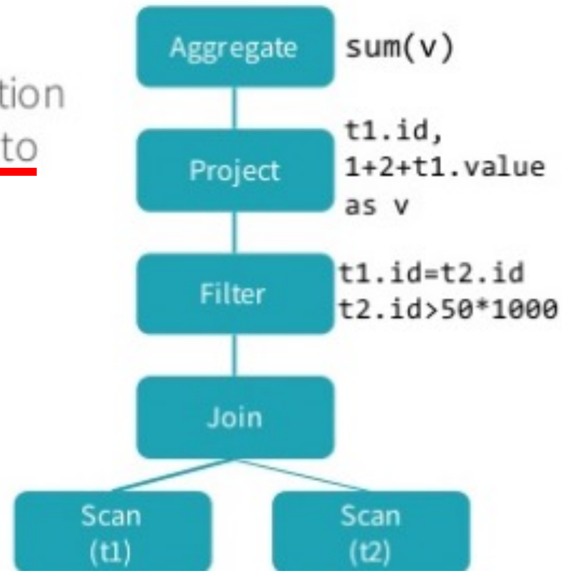




# Spark DataFrames Planning Execution

## Logical Plan

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

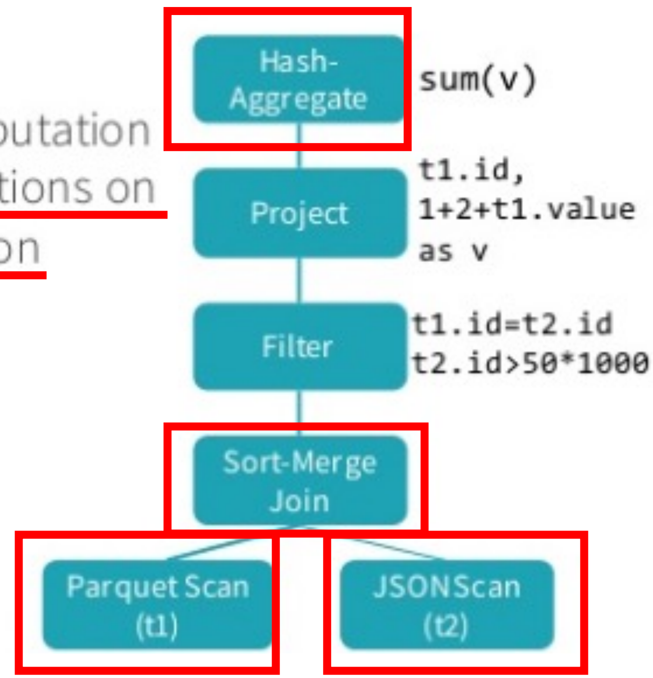



- Output: List of output columns, e.g. id, v
- Constraints: t2.id > 50*1000

# Spark DataFrames
# Planning & Optimizing Execution

## Physical Plan

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




Parquet: a structured, compressed file format

JSON: an unstructured, clear text file format

# Physical Query Optimization of Joins
# Excursion: How Do We Execute a Join?

- Given: relations R(A,B) and S(B,C)
- SELECT *
  FROM R, S
  WHERE R.B = S.B

R

| A | B |
|---|---|
| A1 | 0 |
| A2 | 1 |
| A3 | 2 |
| A4 | 1 |

S

| B | C |
|---|---|
| 1 | C1 |
| 2 | C2 |
| 1 | C3 |
| 3 | C4 |
| 1 | C5 |

$R \bowtie S$

| A | B | C |
|---|---|---|
| A2 | 1 | C1 |
| A2 | 1 | C3 |
| A2 | 1 | C5 |
| A3 | 2 | C2 |
| A4 | 1 | C1 |
| A4 | 1 | C3 |
| A4 | 1 | C5 |

Which physical optimizations (algorithms) exist for performing a join?

# Nested-Loop Join #1

- Super-naïve

```
FOR EACH r IN R DO
   FOR EACH s IN S DO
      IF ( r.B=s.B) THEN OUTPUT (r ⋈ s)
```

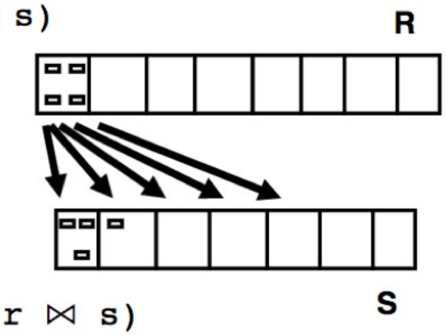
# Nested-Loop Join #2

- Super-naïve

```
FOR EACH r IN R DO
    FOR EACH s IN S DO
        IF ( r.B=s.B) THEN OUTPUT (r ⋈ s)
```

- Slight improvement

```
FOR EACH block x IN R DO
    FOR EACH block y IN S DO
      FOR EACH r in x DO
          FOR EACH s in y DO
             IF ( r.B=s.B) THEN OUTPUT (r ⋈ s)
```




- Cost estimations:
  - b(R), b(S): number of blocks in R and S, respectively
  - Outer relation: each block is read once
  - Inner relation: read once for each block of outer relation
  - Two inner loops are ”free” (only main memory operations)

# Sort-Merge Join

- Approach:
  - Sort both relations on join attributes
  - Merge both sorted relations

# Sort-Merge Join Example #1

R

| A | B |
|---|---|
| A1 | 0 |
| A2 | 1 |
| A3 | 2 |
| A4 | 1 |

S

| B | C |
|---|---|
| 1 | C1 |
| 2 | C2 |
| 1 | C3 |
| 3 | C4 |
| 1 | C5 |

R

| A | B |
|---|---|
| A1 | 0 |
| A2 | 1 |
| A4 | 1 |
| A3 | 2 |

S

| B | C |
|---|---|
| 1 | C1 |
| 1 | C3 |
| 1 | C5 |
| 2 | C2 |
| 3 | C4 |

Partial join

$\{\langle A1,0\rangle,\langle A2,1\rangle\} \bowtie S$

| A | B | C |
|---|---|---|
| A2 | 1 | C1 |
| A2 | 1 | C3 |
| A2 | 1 | C5 |

# Sort-Merge Join Example #2

R

| A | B |
|---|---|
| A1 | 0 |
| A2 | 1 |
| A4 | 1 |
| A3 | 2 |

S

| B | C |
|---|---|
| 1 | C1 |
| 1 | C3 |
| 1 | C5 |
| 2 | C2 |
| 3 | C4 |

Partial join

{<A1,0>,<A2,1>} $\bowtie$ S

| A | B | C |
|---|---|---|
| A2 | 1 | C1 |
| A2 | 1 | C3 |
| A2 | 1 | C5 |

R

| A | B |
|---|---|
| A1 | 0 |
| A2 | 1 |
| A4 | 1 |
| A3 | 2 |

S

| B | C |
|---|---|
| 1 | C1 |
| 1 | C3 |
| 1 | C5 |
| 2 | C2 |
| 3 | C4 |

Partial join

{<A1,0>,<A2,1>,
<A4,1>} $\bowtie$ S

| A | B | C |
|---|---|---|
| A2 | 1 | C1 |
| A2 | 1 | C3 |
| A2 | 1 | C5 |
| A4 | 1 | C1 |
| A4 | 1 | C3 |
| A4 | 1 | C5 |

new

# Hash Join

- Use join attributes as hash keys
- Hash phase:
  - Scan relation S and compute hash table
- Merge phase:
  - Iterate over R tuple-wise
  - Join with S by using hash function
- No sorting is required

# Which Join Algorithm Performs Best?

# Which Join Algorithm Performs Best?

- Hash join is typically faster than sort-merge join (as no sorting is required)
- Sort-merge join is typically faster than nested-loop join for larger tables
- But:
  - Sort-merge can can be faster than hash join, if both tables are already sorted
  - If the join condition is an inequality operator (<, >, <>), hash join can't be used

- Depending on the characteristics of the tables (size, data distribution, indexes, etc.) the optimizer chooses the best join strategy

# What is a DataFrame Really?

- DataFrame consists of:
  - Execution plan
  - Result type schema
  - Underlying RDD
- What is an RDD? (*Resilient Distributed Dataset* [M. Zaharia et al., 2012])
  - Lineage – how was the input data calculated
  - Partition information – where is the input data actually distributed
  - Instructions – code to be executed
- What is a DataFrame NOT?
  - Data

Zurich University
of Applied Sciences

# Declarative Query APIs

- Vague, general definition of declarative programming:
  *Programming where problems are described, or conditions on a solution are described, and the computer finds a solution.*
- For querying data, this can mean:
  "Describing the properties of the requested dataset"
  - SQL as a query language:

    ```
    SELECT dept, AVG(age) FROM pdata GROUP BY dept
    ```
  - Equivalent "Builder" Syntax (Python, Scala, Java)

    ```
    pData.groupBy("dept").agg(avg("age"))
    ```
- No assumptions or indications on *how* to fulfil the query.
- Expression order does not necessarily govern execution order.

# LowLevel RDD Interface

- High – Level APIs: DataFrame (Declarative)

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

or

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

- Lower-Level API: **RDD** (Functional)

```
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()
```

- Generated Intermediate JVM Code: (Imperative)

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

- Generated byte code (executed on machine)

```
00000000    push   ebp
00000001    mov    ebp, esp
00000003    movzx  ecx, [ebp+arg_0]
00000007    pop    ebp
00000008    movzx  dx, cl
0000000C    lea    eax, [edx+edx]
0000000F    add    eax, edx
```

Generates

Abstracts/simplifies

Describes action on column level
Assumes structured, typed data
Executed *somewhere*

Describes action on row level
Assumes homogenous data
Executed by 1..n machines

Describes action on variable level
Assumes typed data
Executed in 1 machine with n CPUs

Describes actions on byte and processor level

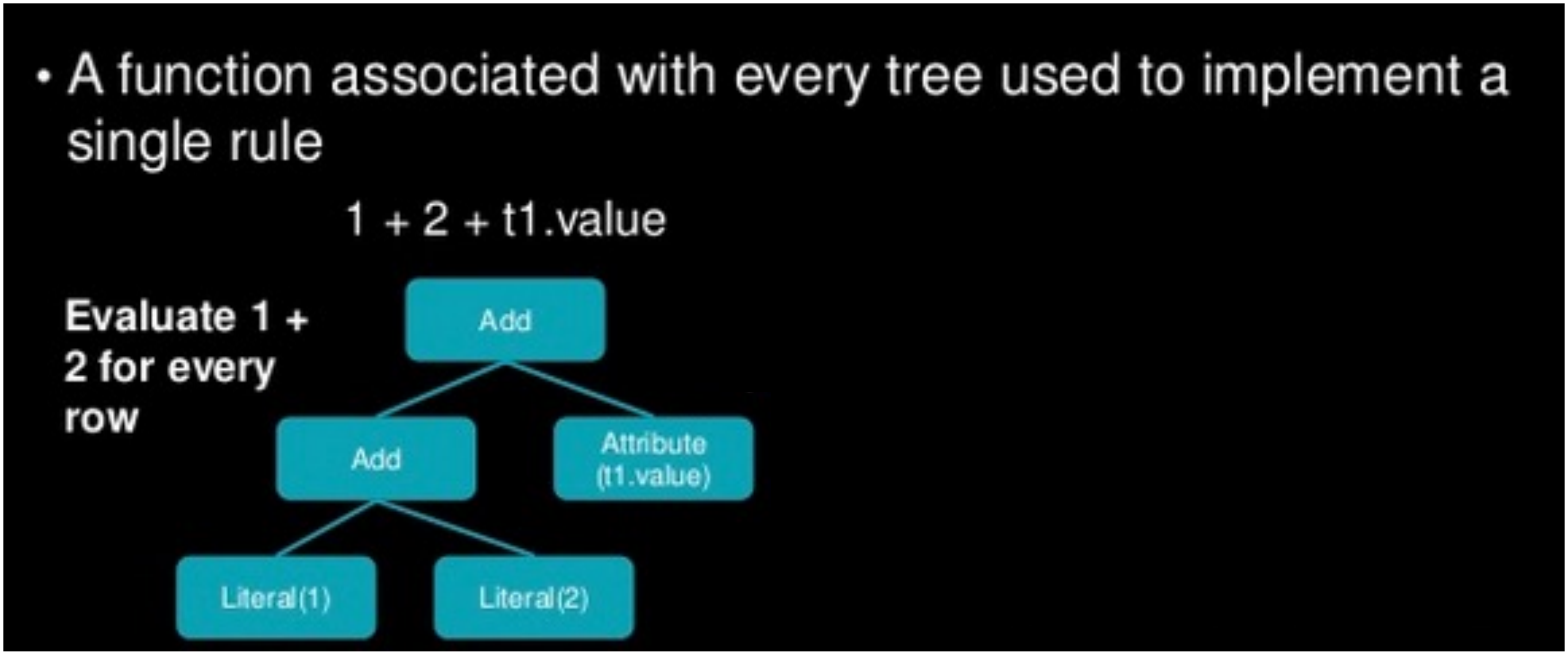
* These code examples are illustrative and almost completely made up,
* don't study them!

# Query Optimization: Transformation

- Functions for converting an un-optimized tree to an optimized tree
  - E.g .Transform tree to logical plan and then to physical plan

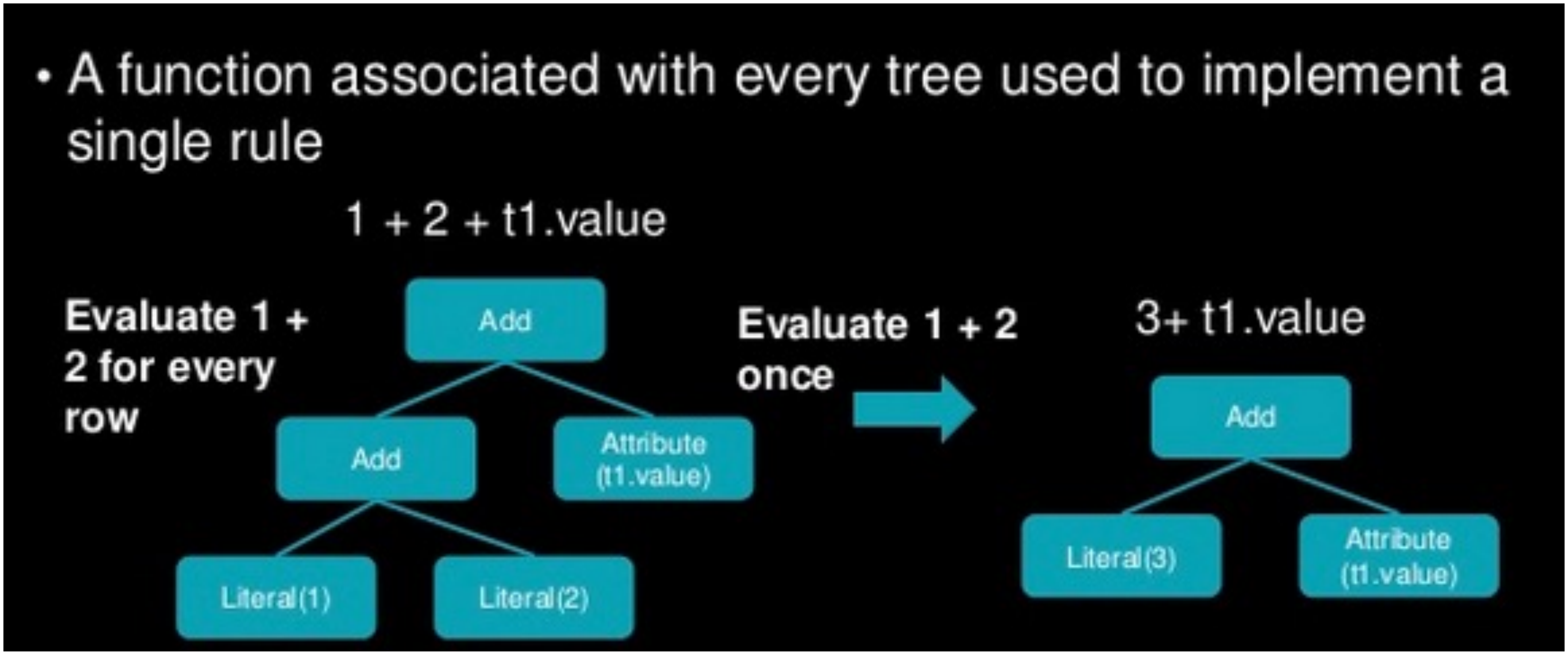
- Assume the function of the previous query:
  - 1 + 2 + t1.value
  - Has to be applied for each row of the table

# Transformation Example #1




Need to evaluate this function for every row. Efficient?

# Transformation Example #2

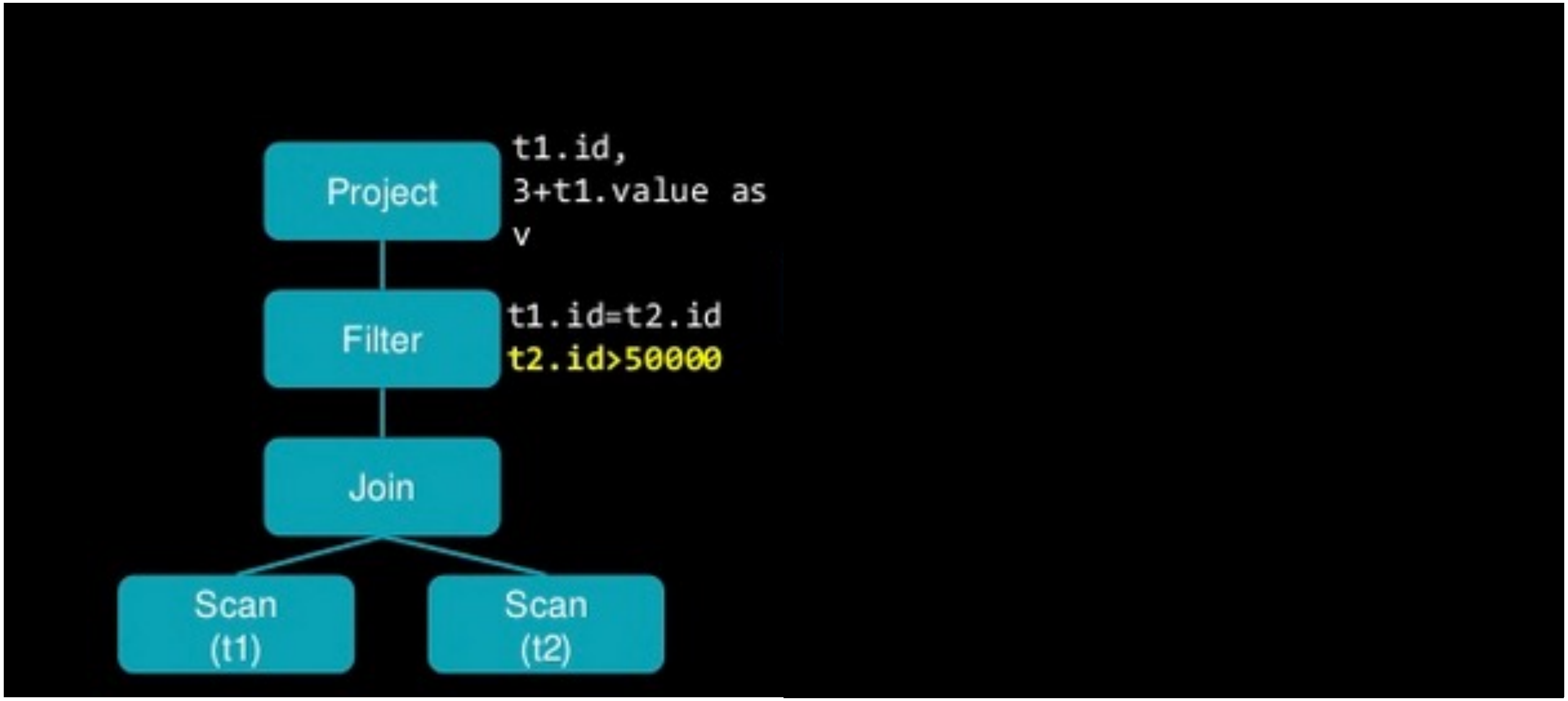


# Catalyst Optimizer Strategies

- Goal: Minimize end-to-end query response time

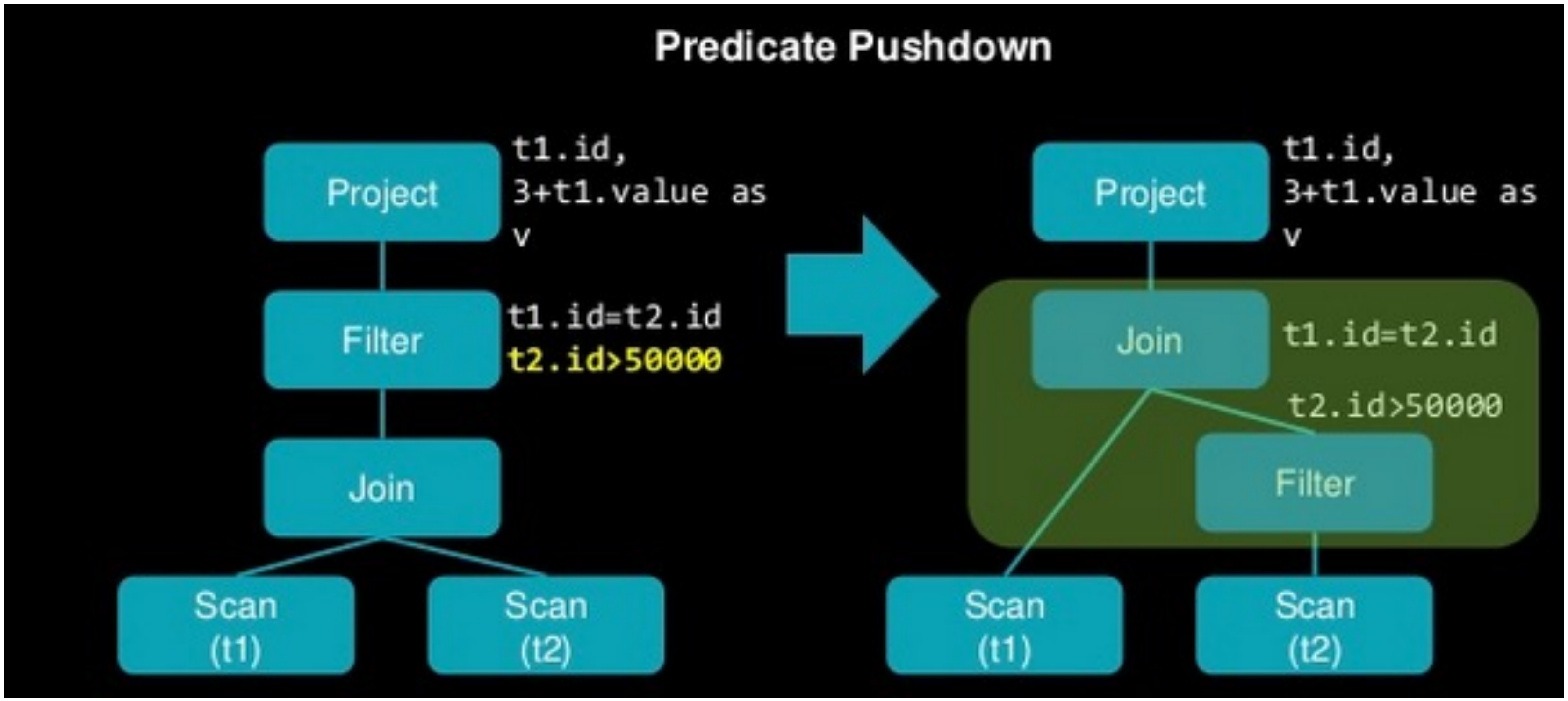
- Two key ideas:
  - Prune unnecessary data as early as possible
    - E.g. filter pushdown, column pruning
  - Minimize per-operator cost
    - E.g. broadcast vs. shuffle, optimal join order

# Logical Query Plan




How do we optimize this query plan?

# Optimizing Logical Query Plan: Predicate Pushdown

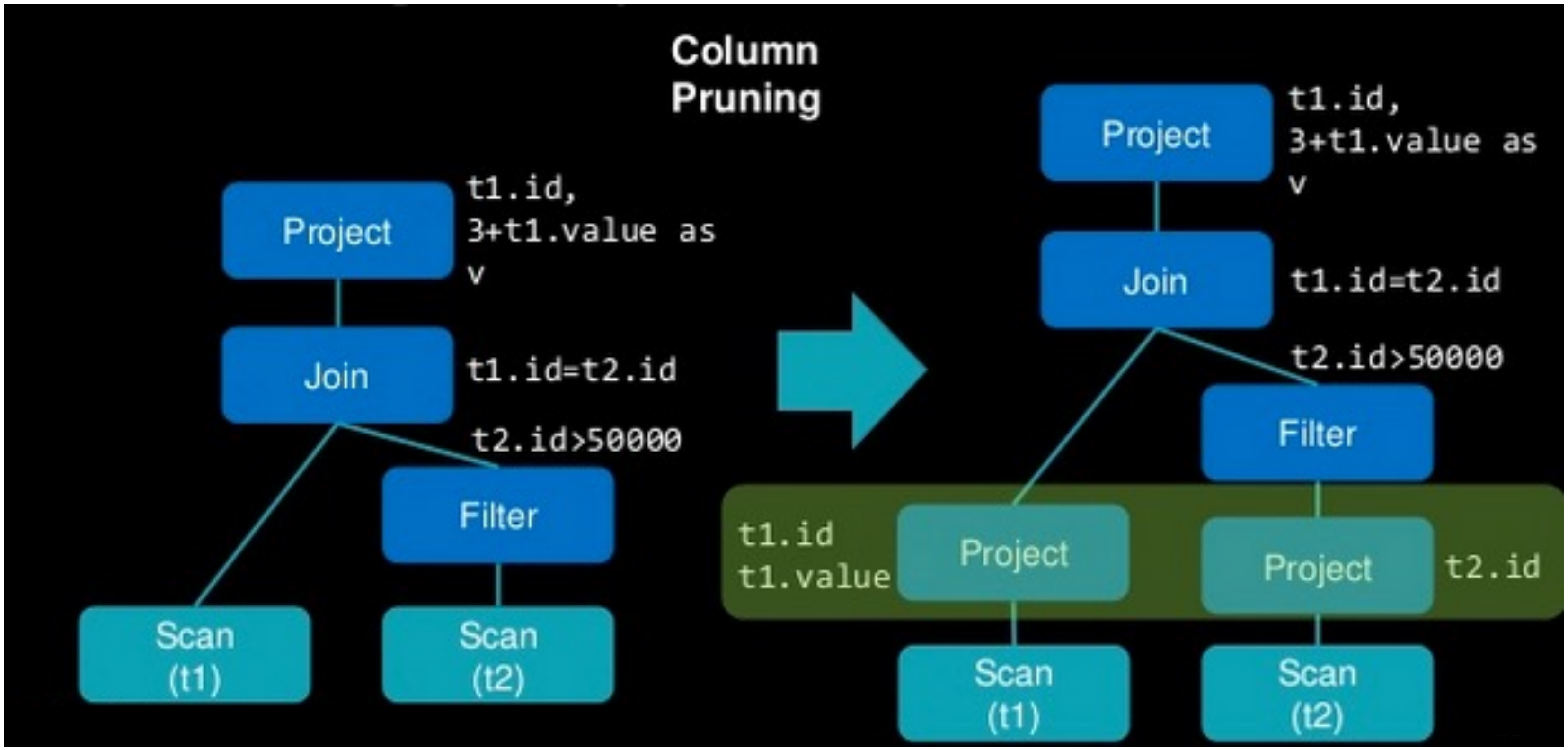


# Logical Query Plan

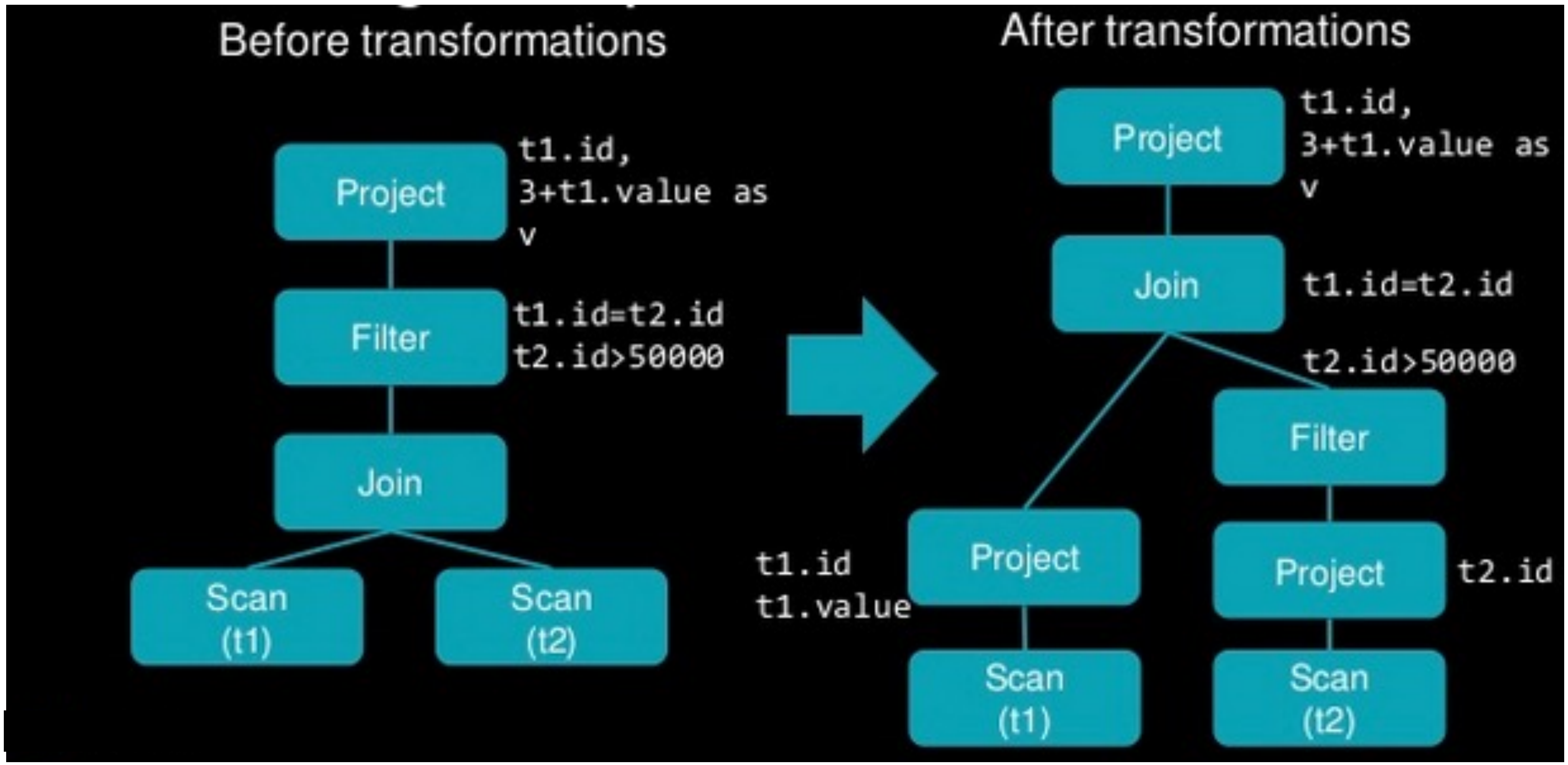

What else can we improve?

Zurich University
of Applied Sciences

# Optimizing Logical Query Plan: Column Pruning



# Optimized Logical Plan



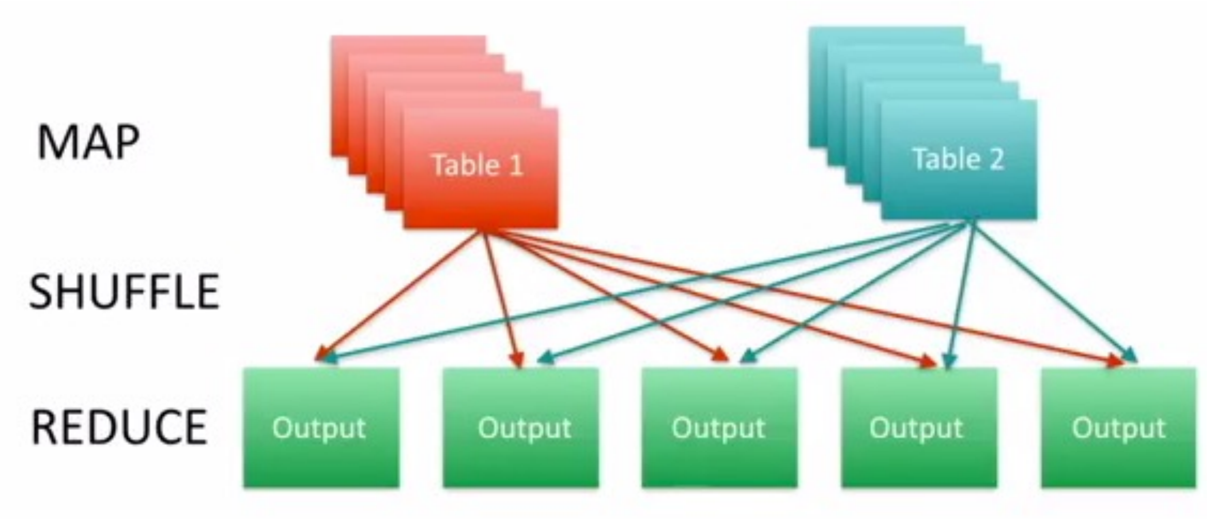
# Takeaway DataFrame API

- A declarative programming API hides complexity
  - How to distribute execution
  - How to treat different data sources
  - How to optimize execution

# Physical Query Optimization: Optimizing Joins for Distributed Data

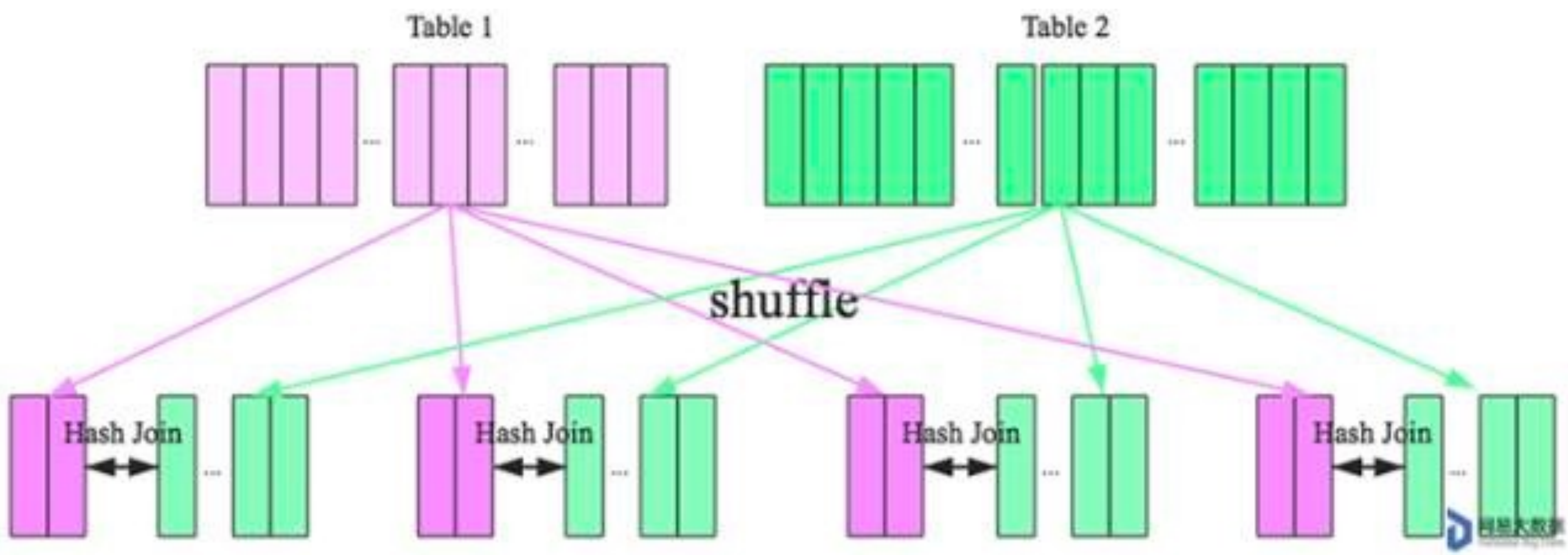
- Shuffle Hash Join
- Broadcast Hash Join

# Shuffle Hash Join




- Map through two different DataFrames
- Use fields in join condition as the output key
- Shuffle both data sets by the output key
- Reduce phase: join the two data sets
  (note: rows of both tables with the same keys are on the same machine and sorted)

# Shuffle Hash Join



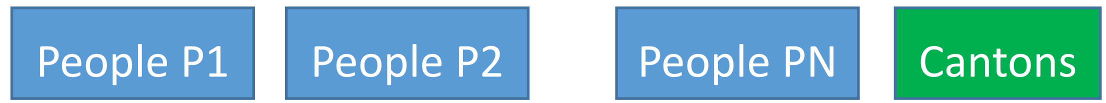

Source: Andreas Weiler

# Shuffle Hash Join Performance

- Works best when
  - Distributed evenly with the key you are joining on
  - Have an adequate number of keys for parallelism
    - E.g. If table A has 1,000,000 rows but only 20 keys, the maximum parallelism is 20

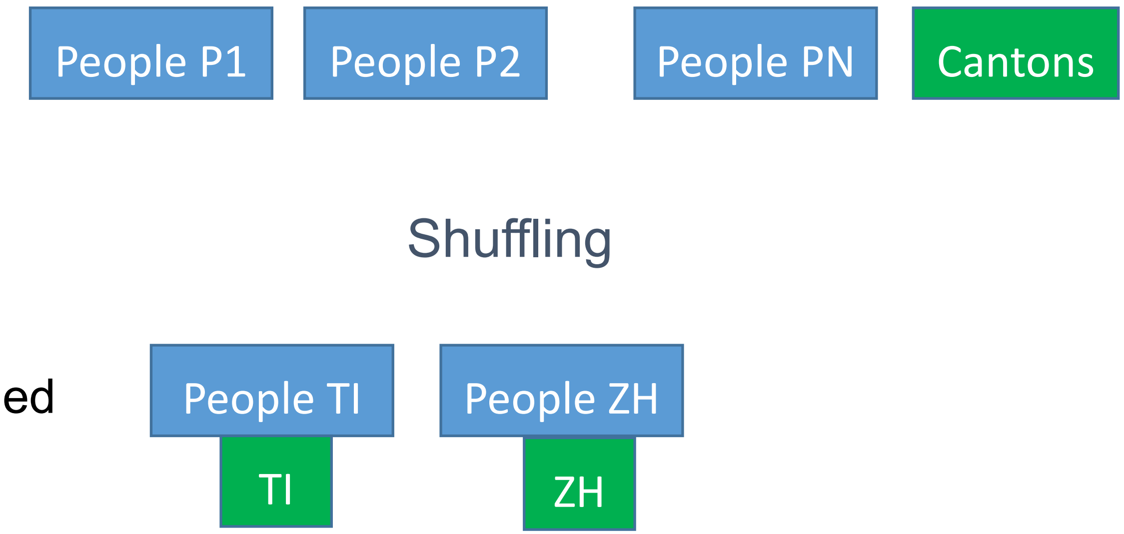
# Uneven Sharding & Limited Parallelism #1

```
SELECT *
FROM PEOPLE_IN_CH p
JOIN CANTONS c
ON p.canton_ID = c.canton_ID
```



# Uneven Sharding & Limited Parallelism #2

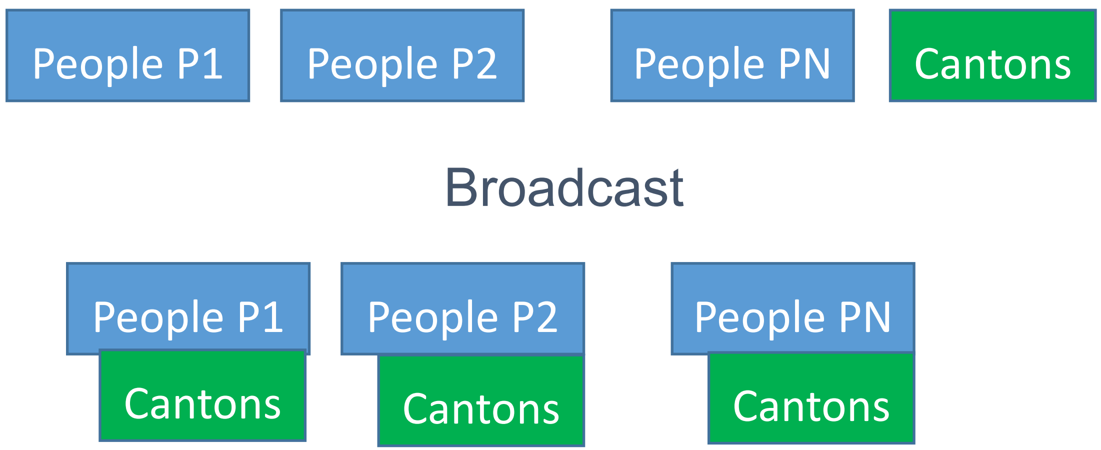
```
SELECT *
FROM PEOPLE_IN_CH p
JOIN CANTONS c
ON p.canton_ID = c.canton_ID
```



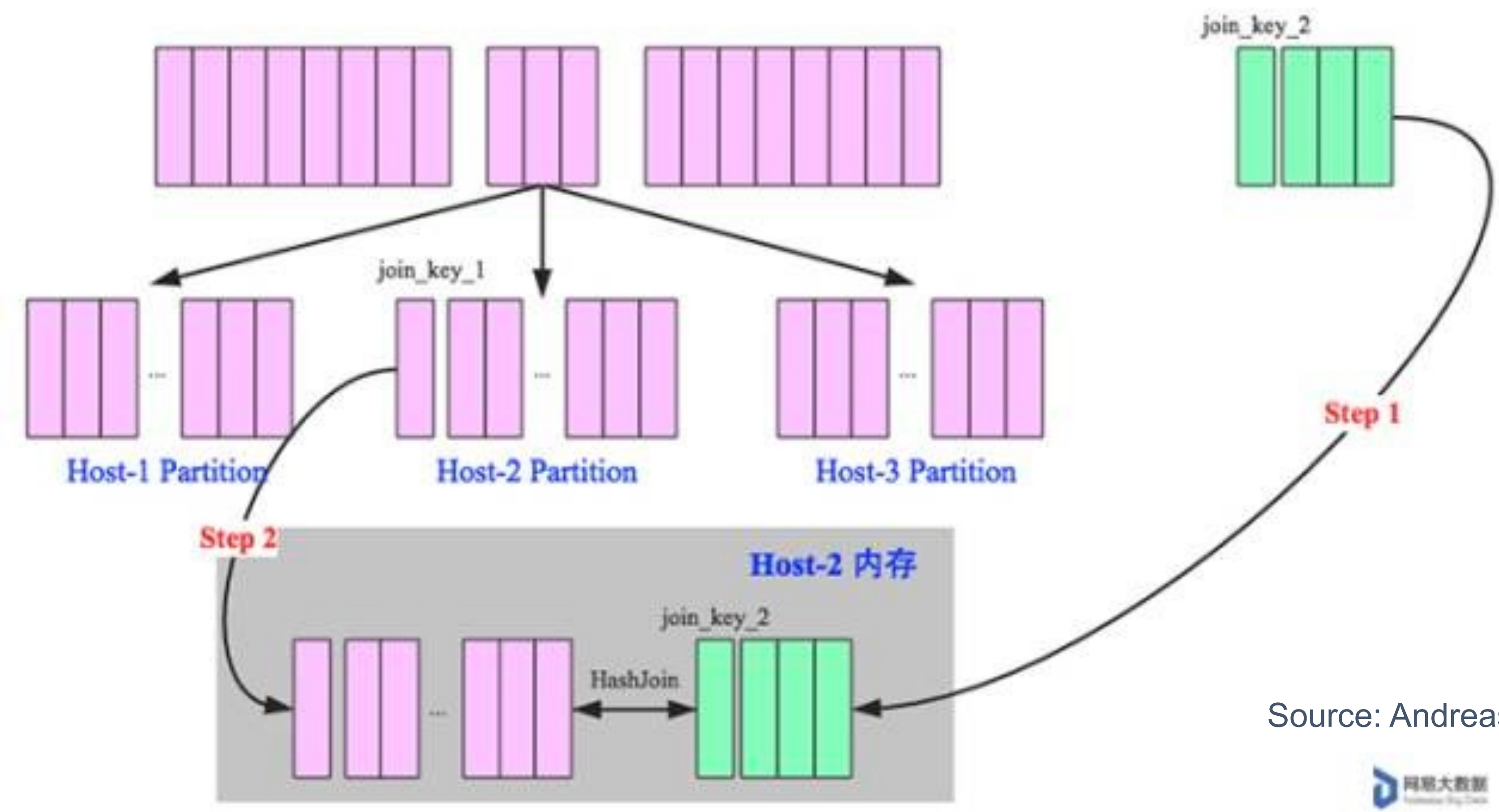

- All the people will only be shuffled into 26 keys for the cantons
- Problem:
    - Uneven sharding
    - Limited parallelism (max. 26)

# Broadcast Hash Join

- When one data frame is small enough to fit into main memory:
  - Broadcast “small” DataFrame to all nodes
- Enables partial local join:
  - No shuffling required
  - No additional communication overhead over network



# Broadcast Hash Join




Source: Andreas Weiler

# Broadcast Hash Join vs. Shuffle Hash Join

- Broadcast Hash Join often better than Shuffle Hash Join (no data transfer over network)
- Should in principle be automatic but might require hints:
  - Spark SQL on parquet does this automatically
  - Not if input file is a text file

# How Efficient is Parquet for Joins with Respect to CSV ?

# Experiment Queries

- Data:
  - Fire1: 122 MB CSV-file, 485,056 rows, 27 columns
  - Fire2: 99 MB CSV-file, 395,658 rows, 27 columns

- Projection: sqlContext.sql("select distinct Postcode_district from fire1").show()

- Join: sqlContext.sql("select distinct f1.Postcode_district from fire1 f1 join fire2 f2 where f1.Postcode_district = f2.Postcode_district").show()

- Self-Join: sqlContext.sql("select distinct f1.Postcode_district from fire1 f1 join fire1 f2 where f1.Postcode_district = f2.Postcode_district").show()

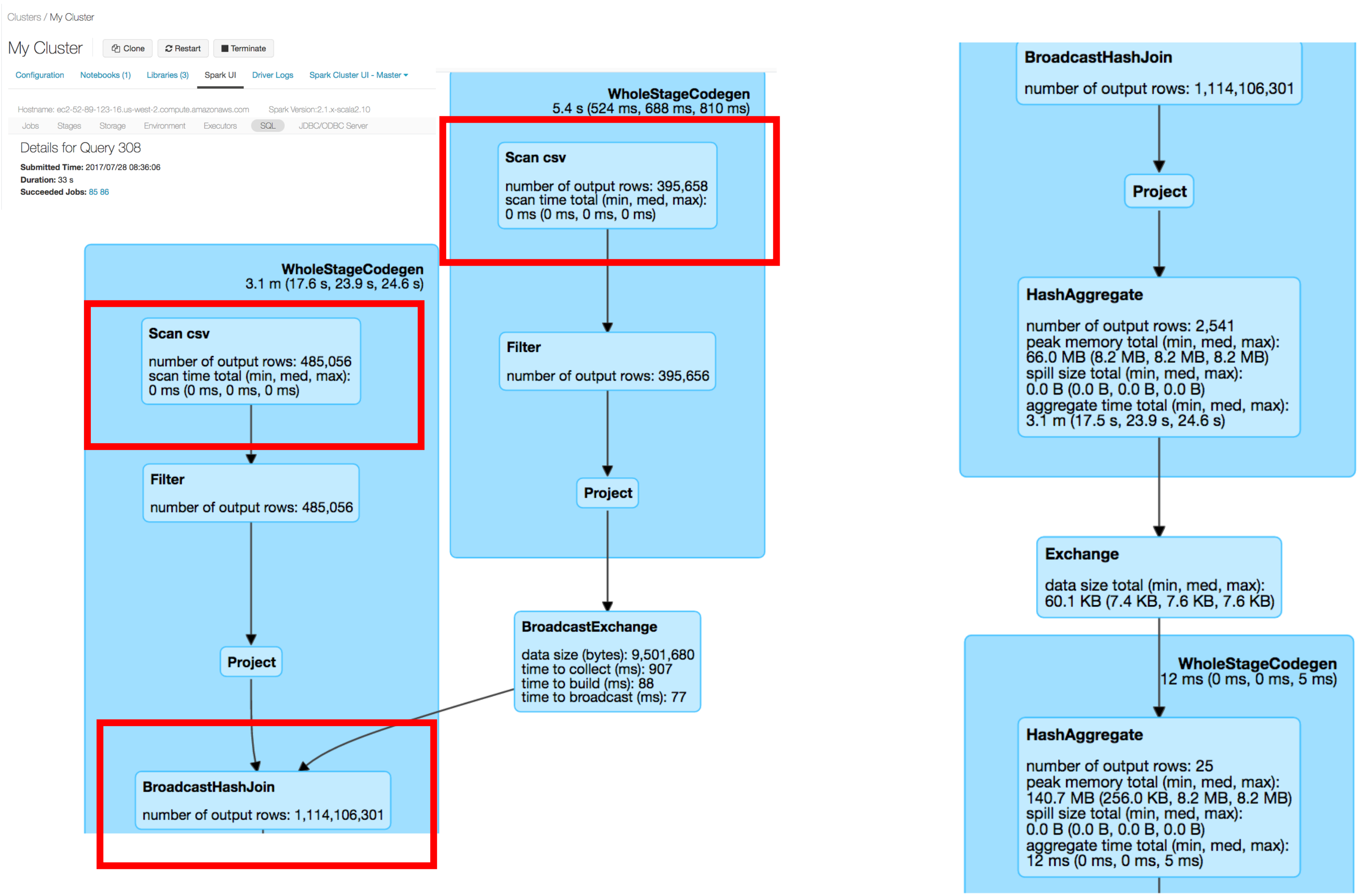
Zurich University of Applied Sciences

# Experiment Results

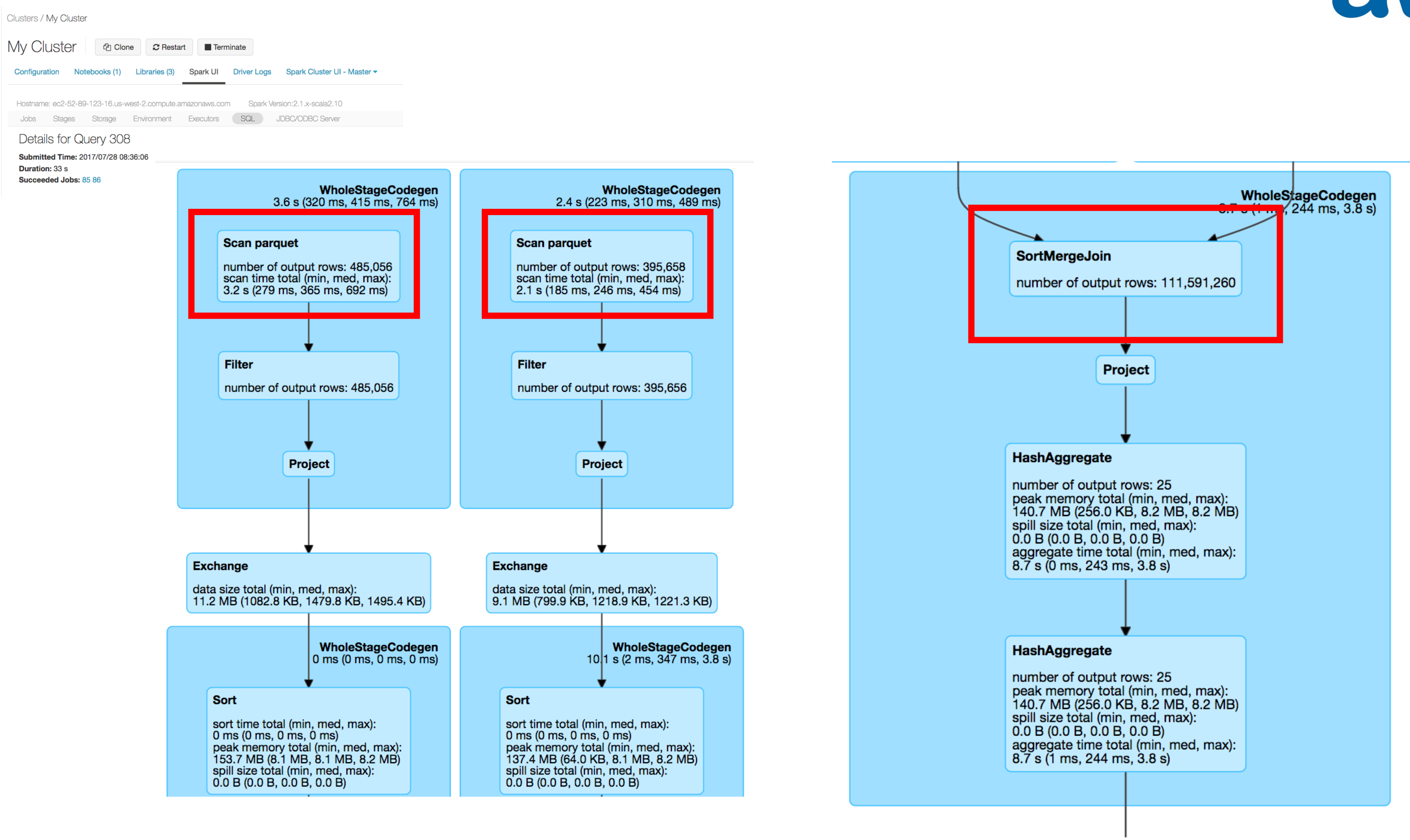
| File Format | Fire 1 Size (MB) | Fire 2 Size (MB) | Join (sec) | Self Join (sec) |
|---|---|---|---|---|
| CSV | 122 | 99 | 26 | 33 |
| Parquet | 6 | ~6 | 6 | 2.5 |

Executed on Databricks Community Edition

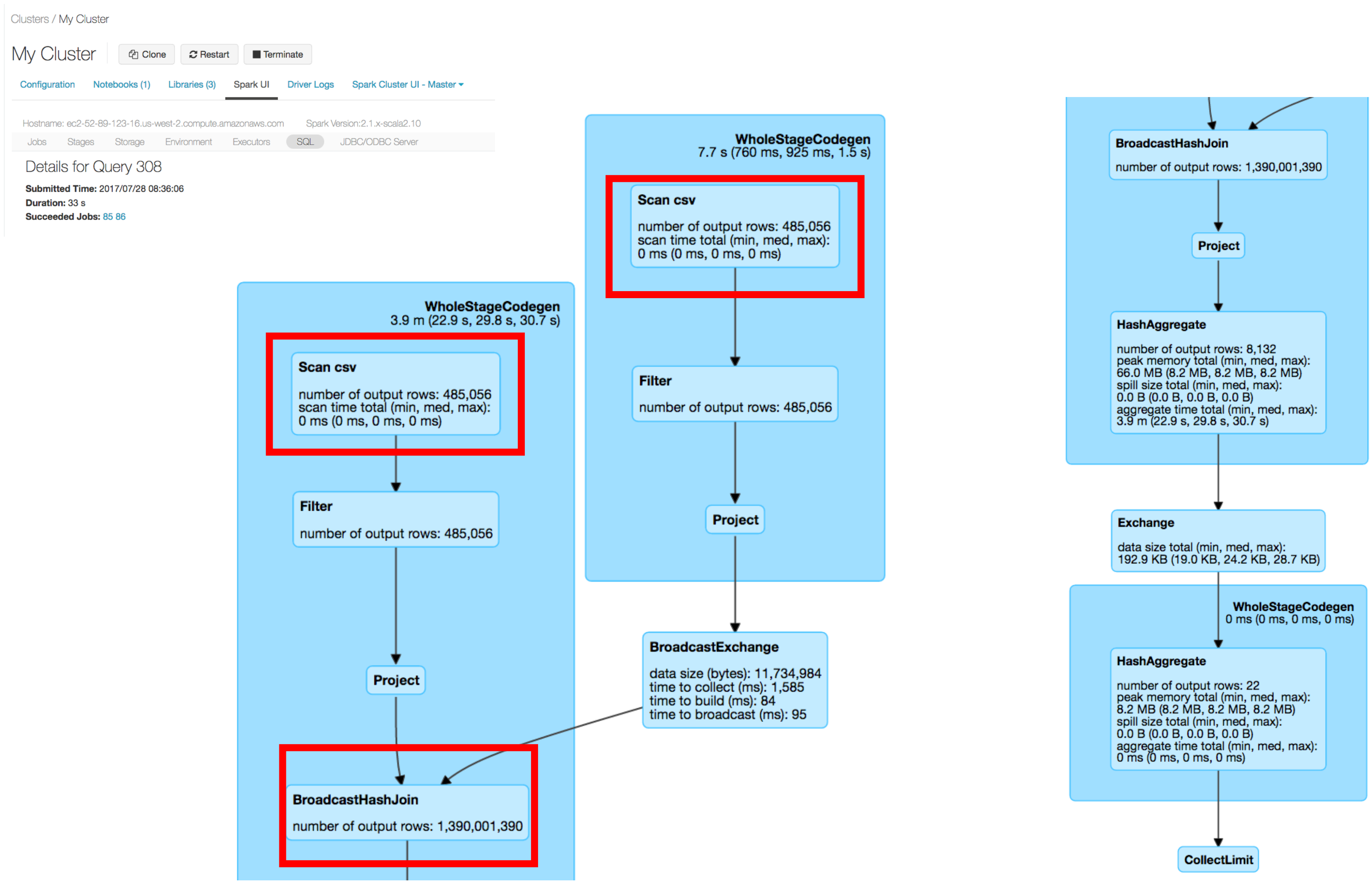
# Spark UI Inspection – Join: CSV



# Spark UI Inspection – Join: Parquet



# Spark UI Inspection – Self-Join: CSV



# Spark UI Inspection – Self-Join Parquet

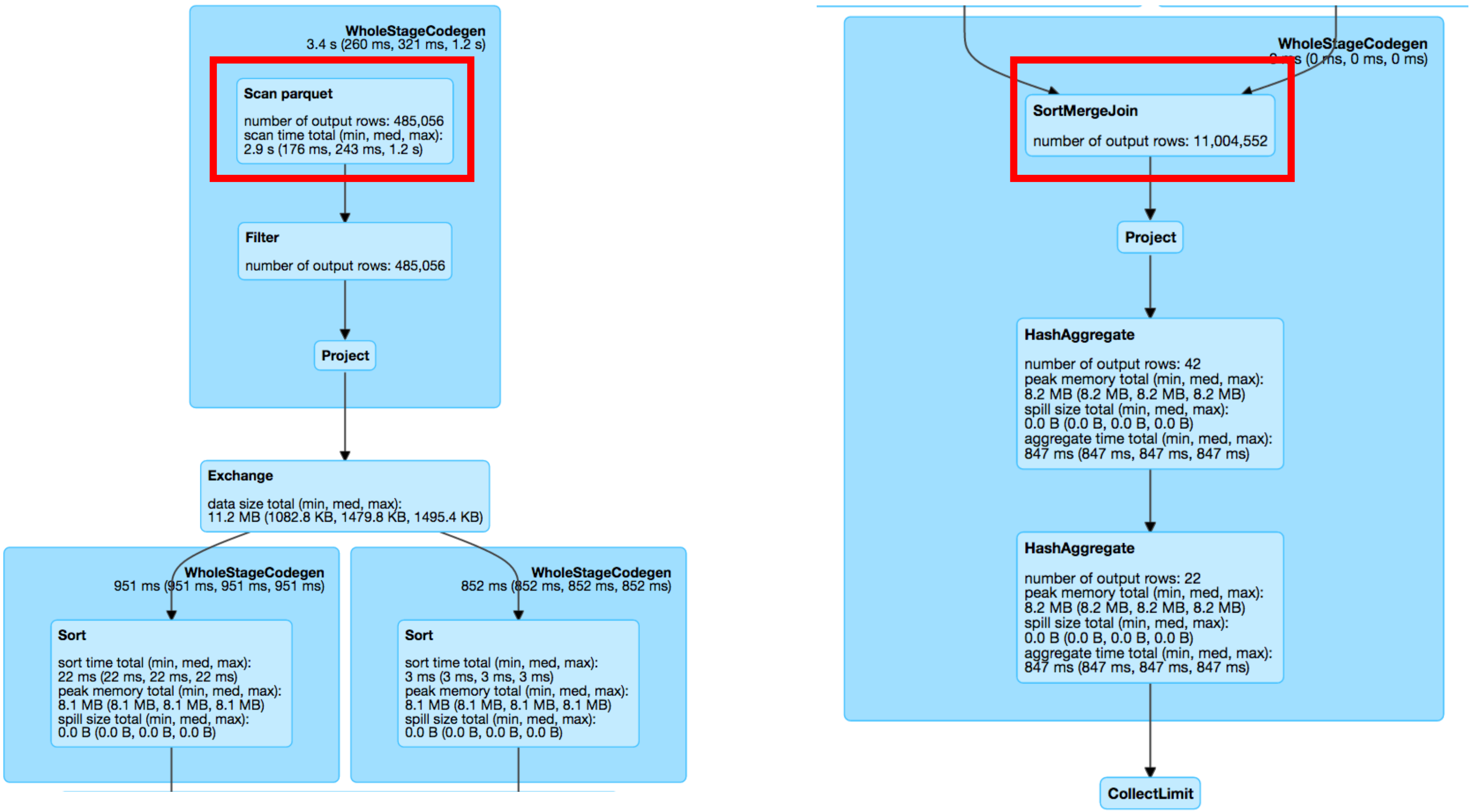



Table is only read once!

# Conclusions

- We learned how a query is analyzed
- Difference between logical and physical query plan
- Performing different types of joins:
  - Nested-loop join
  - Sort-merge join
  - Hash join
- Distributed joins:
  - Shuffle hash join
  - Broadcast hash join
- Join performance on CSV and Parquet file