CIT650 Introduction to Big Data

Spark SQL

Project Tungesten

Spark SQL[7]

- RDD APIs although richer and more concise than MapReduce, still are considered low-level
- We still need to benefit from the in-memory execution model of Spark but make it accessible to more people
  - Similar to Hive and Impala for MapReduce/HDFS, Spark SQL wraps RDD API calls with an SQL-like shell
- Spark SQL uses DataFrames and DataSet abstractions
- It has an advanced query optimizer, called Catalyst

[7]Armbrust, Michael, et al. "Spark sql: Relational data processing in spark." Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, 2015.

Programming Interface

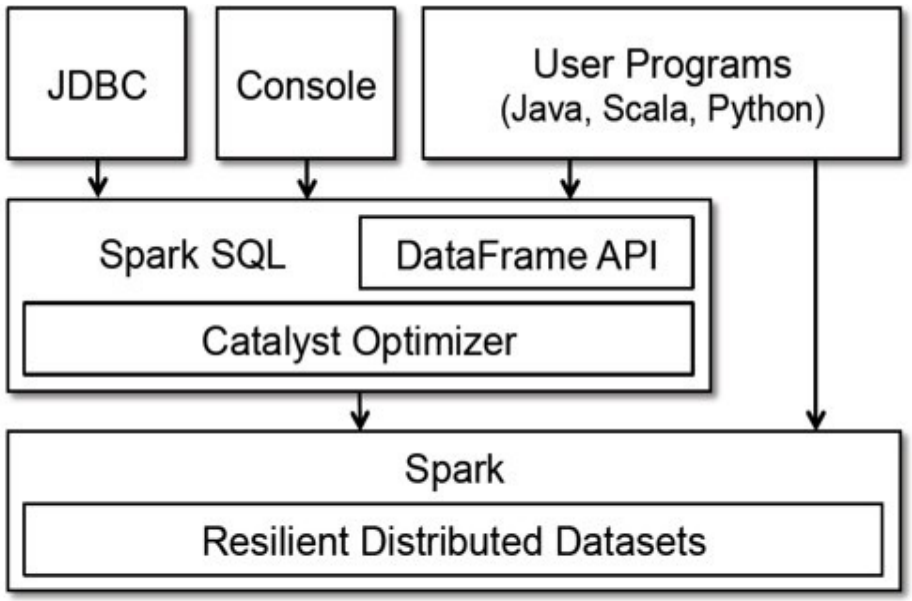


Figure 1: Interfaces to Spark SQL, and interaction with Spark.

source: Armbrust, Michael, et al. “Spark SQL: Relational data processing in spark.” Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, 2015.

DataFrame

- A distributed collection of rows organized into named columns
- A DataFrame is similar to Table (Relation) in relational databases
- Supports complex types
  - Structs, maps, unions
- Data can be selected, projected, joined, and aggregated
  - Similar to Python's Pandas

Creating DataFrames

- From a structured file source
  - JSON or Parquet files
  - From an existing RDD
  - By Performing an operation on another DataFrame
  - By programmatically defining a schema
- Example

```
peopleDF = sqlCtx.jsonFile("people.json")
```

File: people.json

```
{"name":"Alice", "pcode":"94304"}
{"name":"Brayden", "age":30, "pcode":"94304"}
{"name":"Carla", "age":19, "pcode":"10036"}
{"name":"Diana", "age":46}
{"name":"Étienne", "pcode":"94104"}
```

name	pcode	age
Alice	94304	Null
Brayden	94304	30
Diana	Null	46
…		

Basic DataFrame Operatrions

- Operations to deal with DataFrame metadata
  - schema ? returns a schema object describing the data
  - printSchema ? displays the schema as a tree
  - cache/persist ? persists the DataFrame to disk or memory
  - columns ? returns an array of column names
  - Dtypes ? retuns an array of pairs of column names and types
  - Explain ? prints debug information about the DataFrame to the console

```
> peopleDF = sqlCtx.jsonFile("people.json")
> for item in peopleDF.dtypes(): print(item)
('age', 'bigint')
('name', 'string')
('pcode', 'string')
```

Manipulating Data in DataFrames

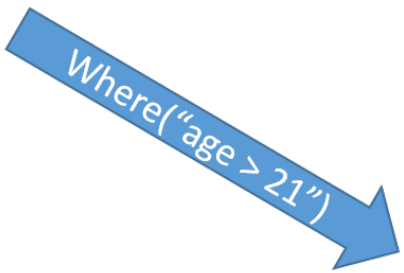
- Queries create new DataFrames
  - DataFrames are immutable
  - Analogous to RDD transformations
  - Some query methods
    - Select: returns a new DataFrame with the selected columns only from base DataFrame
    - Join: joins the base DataFrame with the other DataFrame
    - where: keeps only the records that match the condition in the new DataFrame
  - Actions return data
    - Lazy execution as with RDDs
    - Some actions: take(n), collect(), count()

DataFrame Query String

- You can pass column names as String
  - peopleDF.select(“name”,”age”)
  - peopleDF.where(“age >21”)

name	pcode	age
Alice	94304	Null
Brayden	94304	30
Diana	Null	46
Carla	Null	19

name	age
Alice	Null
Brayden	30
Diana	46
Carla	19



name	pcode	age
Brayden	94304	30
Diana	Null	46

Querying DataFrames Using Columns

- You can refer to the column object. In Python:
  - peopleDF.select(peopleDF.age,peopleDF.name)
  - peopleDF.select(peopleDF.age+10, peopleDF.name.toUpperCase())
  - peopleDF.select(peopleDF("age"),peopleDF("name"))

  - peopleDF.select(peopleDF("age")+10,peopleDF("name").toUpperCase())
  - peopleDF.sort(peopleDF.age.desc())

SQL Queries

- It is possible to query a DataFrame using SQL
  - First, register the DataFrame as a temp table

```
peopleDF.registerTempTable("people")
sqlCtx.sql("SELECT * FROM people where name like 'A%' ")
```

RDD Vrs. DataFrames Vrs. Spark SQL

Using RDDs

```
data = sc.textFile(...).split("\t")
data.map(lambda x: (x[0], [int(x[1]), 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()
```

Statement	Operation	Example Output
sc.textFile(...)	Read data into RDD	["John\t29", "John\t31", "Jane\t21"]
.split("\t")	Split lines into arrays	[["John", "29"], ["John", "31"], ["Jane", "21"]]
.map(lambda x: (x[0], [int(x[1]), 1]))	Map to key-value pairs	[("John", [29, 1]), ("John", [31, 1]), ("Jane", [21, 1])]
.reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]])	Aggregate values by key	[("John", [60, 2]), ("Jane", [21, 1])]
.map(lambda x: [x[0], x[1][0] / x[1][1]])	Calculate average age	[("John", 30), ("Jane", 21)]
.collect()	Collect results to driver	[("John", 30), ("Jane", 21)]

RDD Vrs. DataFrames Vrs. Spark SQL

Using RDDs

```
data = sc.textFile(...).split("\t")
data.map(lambda x: (x[0], [int(x[1]), 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()
```

Using SQL

```
SELECT name, avg(age)
FROM people
GROUP BY name
```

RDD Vrs. DataFrames Vrs. Spark SQL

Using RDDs

```
data = sc.textFile(...).split("\t")
data.map(lambda x: (x[0], [int(x[1]), 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()
```

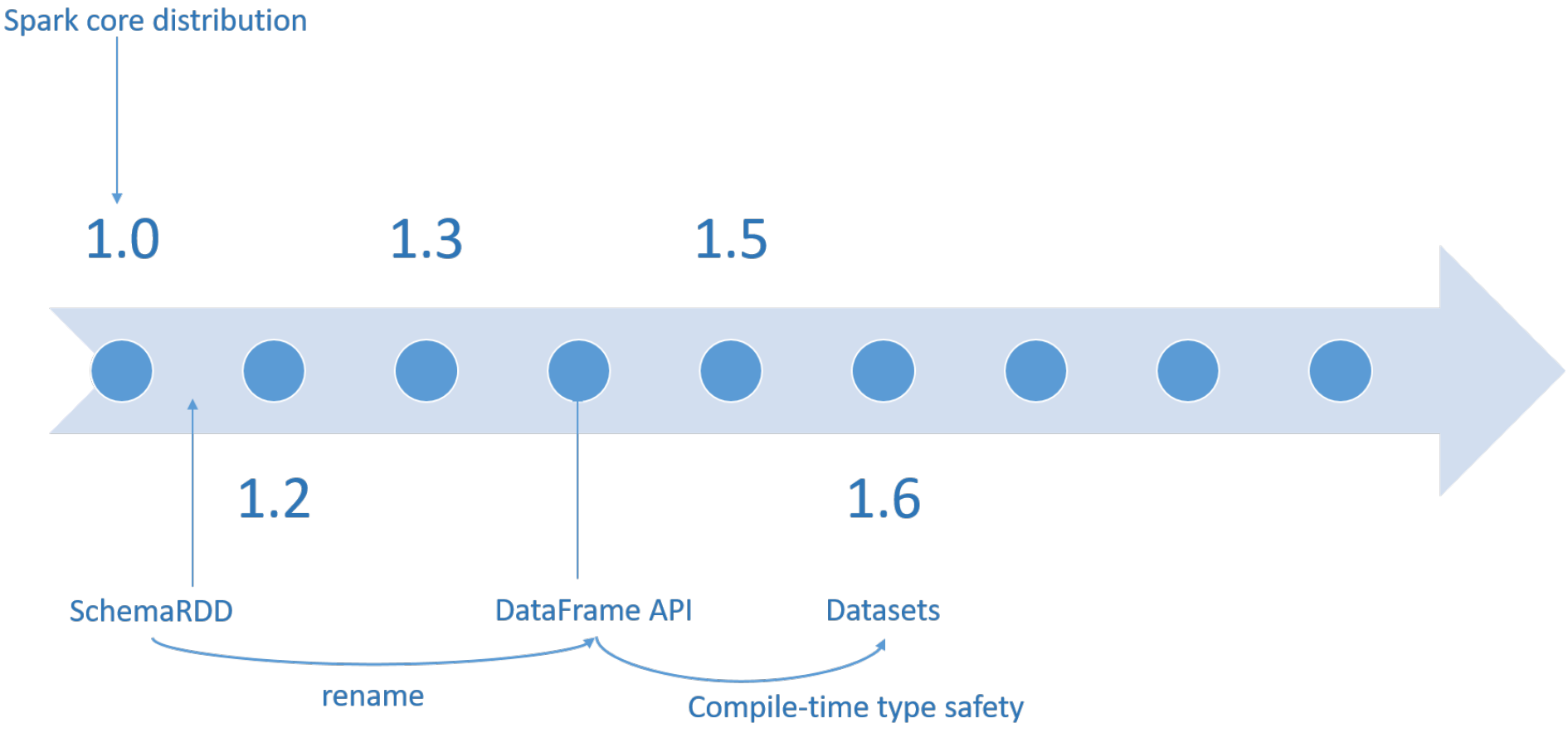
Using SQL

```
SELECT name, avg(age)
FROM people
GROUP BY name
```

Using DataFrames

```
sqlCtx.table("people") \
    .groupBy("name") \
    .agg("name", avg("age")) \
    .collect()
```

Major Milestones in Spark SQL



DataFrames Vs. Datasets

Feature	DataFrame API Example	Dataset API Example
Type Safety	DataFrames are not type-safe; errors in column names or data types are caught at runtime.	Datasets are type-safe; errors in column names or data types are caught at compile-time.
API Usage	Operates with untyped API (Dataset<Row>), which uses column names as strings.	Operates with a typed API, which uses Java classes to represent rows and compile-time checked lambda functions.
Data Representation	Represents data as rows without any compile-time type information.	Represents data as objects of a specified class, providing compile-time type information.
Lambda Functions	Uses lambda functions that work with Row objects, which provide no compile-time type checking.	Uses lambda functions that work with typed objects (e.g., Person), allowing for compile-time type checking.
Serialization/Deserialization	Implicitly converts data to and from Row objects.	Uses Encoders to convert data to and from Java objects, which can be more efficient.
Data Source	Reads JSON directly into a DataFrame (Dataset<Row>).	Reads JSON and converts it into a Dataset of Java objects using Encoders and the Java Bean class.

Spark RDD API Example

```
public class SparkRDDExample {
    public static void main(String[] args) {

        SparkConf conf = new SparkConf().setAppName("FilterByAge").setMaster("local[2]");

        JavaSparkContext sc = new JavaSparkContext(conf);

        JavaRDD<String> lines = sc.textFile("person.txt");

        JavaRDD<String> filteredLines = lines
                .filter((Function<String, Boolean>)
                        s -> Integer.parseInt(s.split(",")[1].trim()) > 26);

        // Now we can collect and print our filtered RDD
        List<String> peopleList = filteredLines.collect();
        peopleList.forEach(System.out::println);

        sc.close();
    }
}
```

Spark DataFrame Example - SQL

```
public class SparkSQLExample {
    public static void main(String[] args) {
        // Initialize Spark Session
        SparkSession spark = SparkSession
                .builder()
                .appName("Java Spark SQL Example")
                .config("spark.master", "local")
                .getOrCreate();

        // Read JSON file and infer schema
        Dataset<Row> df = spark.read()
                                .option("mode", "DROPMALFORMED")
                                .json("person.json");


        // Show the contents of the Dataset, and Print the schema
        df.show();
        df.printSchema();

        // Register the DataFrame as a SQL temporary view
        df.createOrReplaceTempView("person");

        // Perform SQL Query
        Dataset<Row> sqlDF = spark.sql("SELECT * FROM person WHERE age > 21");
        sqlDF.show();

        // Stop Spark Session
        spark.stop();
    }
}
```

Spark DataFrame Example

```
public class SparkDataFrameExample {

        public static void main(String[] args) {
            // Initialize Spark Session
            SparkSession spark = SparkSession
                    .builder()
                    .appName("DataFrameExample")
                    .master("local")
                    .getOrCreate();

            // Read JSON file into a DataFrame
            Dataset<Row> df = spark.read().json("person.json");

            // Show the contents of the Dataset, and Print the schema
            df.show();
            df.printSchema();

            // Select "name" and "age" columns and filter rows where "age" is greater than 21
            Dataset<Row> filteredDF = df.select("name", "age").where("age > 21");

            // Show the result
            filteredDF.show();

            // Stop the Spark Session
            spark.stop();
        }
}
```

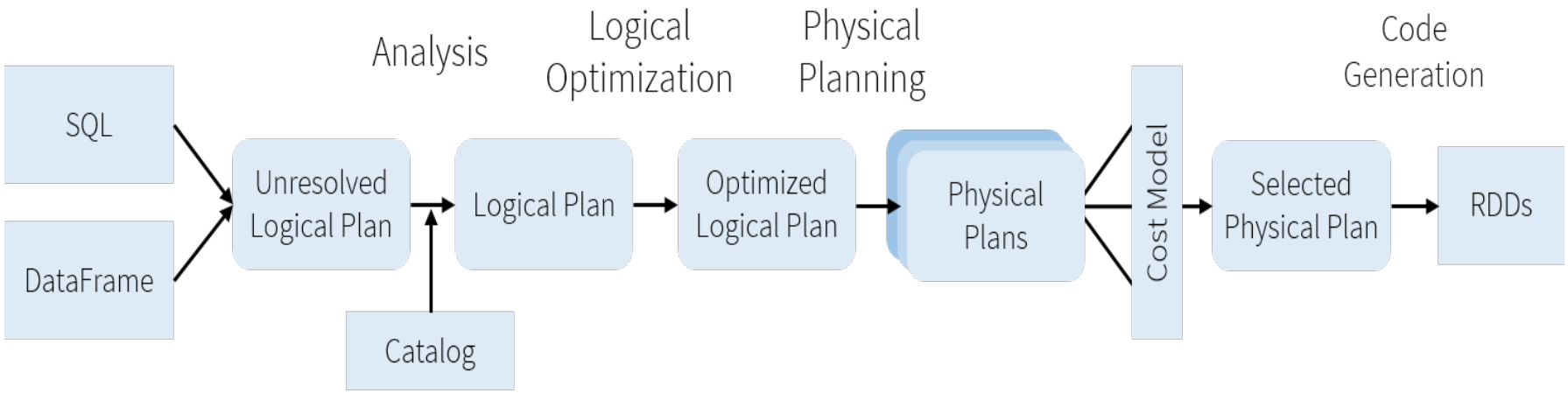
Spark Dataset Example

```java
public class SparkDatasetExample {
	public static void main(String[] args) {
		// Initialize Spark Session
		SparkSession spark = SparkSession
			.builder()
			.appName("DatasetExample")
			.master("local")
			.getOrCreate();

		// Read JSON file into a Dataset
		Dataset<Person> peopleDS = spark.read()
					.option("multiline", true)
					.json("person.json")
					.as(Encoders.bean(Person.class));

		// Show the contents of the Dataset, and Print the schema
		peopleDS.show();
		peopleDS.printSchema();

		// Filter rows where "age" is greater than 21 and select the "name" and "age" columns
		Dataset<Person> filteredDS = peopleDS.filter((Person person) -> person.getAge() > 21)
					.select(col("name"), col("age"))
					.as(Encoders.bean(Person.class));

		// Show the result, and Stop the Spark Session
		filteredDS.show();

		// Stop the Spark Session
		spark.stop();
	}
}
```

```java
public class Person {
    private String name;
    private Long age; // Use Long instead of long

    // Getters and setters
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public Long getAge() { return age; }
    public void setAge(Long age) { this.age = age; }
}
```

Catalyst: Plan Optimization and Execution[8]

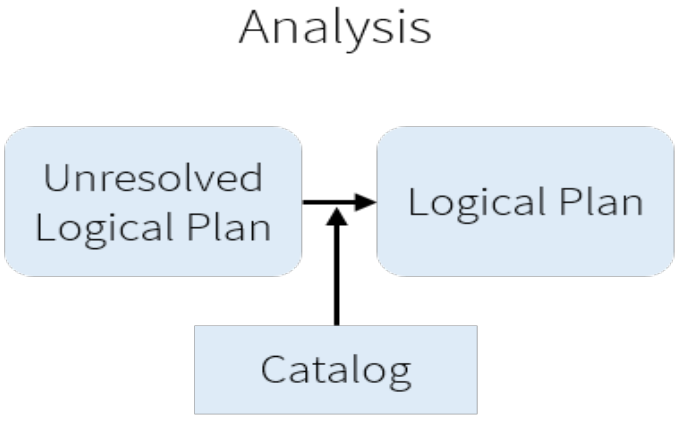
- An extensible query optimizer
- Builds on Scala's pattern matching capabilities
- Plans are represented as trees
- Optimization rules transform trees



[8]Armbrust, Michael, et al. "Spark sql: Relational data processing in spark."Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, 2015.

Analysis

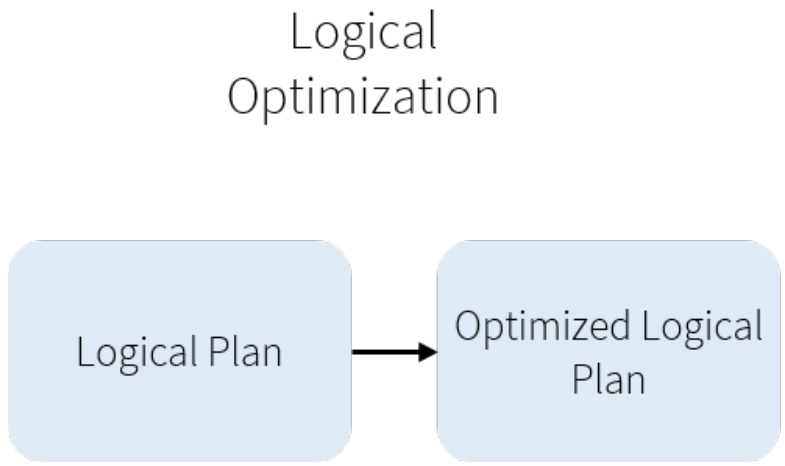
- An attribute is unresolved if it's type is not known or it's not matched to an input table.
- To resolve attributes:
  - Look up relations by name from the catalog.
  - Map named attributes to the input provided given operator's children.
  - UID for references to the same value
  - Propagate and coerce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



SELECT *col* FROM *sales*

Logical Optimization

- Applies standard rule-based optimization
  - constant folding,
  - predicate-pushdown,
  - projection pruning,
  - null propagation,
  - Boolean expression simplification
  - Subquery elimination, etc



Logical Optimization

Technique	Description	Example
Constant Folding	Pre-evaluates constant expressions during planning.	SELECT 1+2; becomes SELECT 3;.
Predicate Pushdown	Applies filters early, close to the data source.	Filters data during read operation, not afterwards.
Projection Pruning	Retrieves only the columns needed by the query.	Selects specific columns from a table.
Null Propagation	Simplifies expressions when null values are involved, based on nullability rules.	Parts of expressions with null become null.
Boolean Expression Simplification	Optimizes boolean logic by eliminating redundancies.	Simplifies col = TRUE AND TRUE to col = TRUE.
Subquery Elimination	Replaces complex subqueries with more efficient constructs like joins.	Converts correlated subqueries to joins.

Trees

- A tree is the main data type in the catalyst optimizer.
- A tree contains node objects.
- A node can have one or more children.
- New nodes are defined as subclasses of TreeNode class.
- These objects are immutable in nature.
- The objects can be manipulated using functional transformation

Tree Abstractions

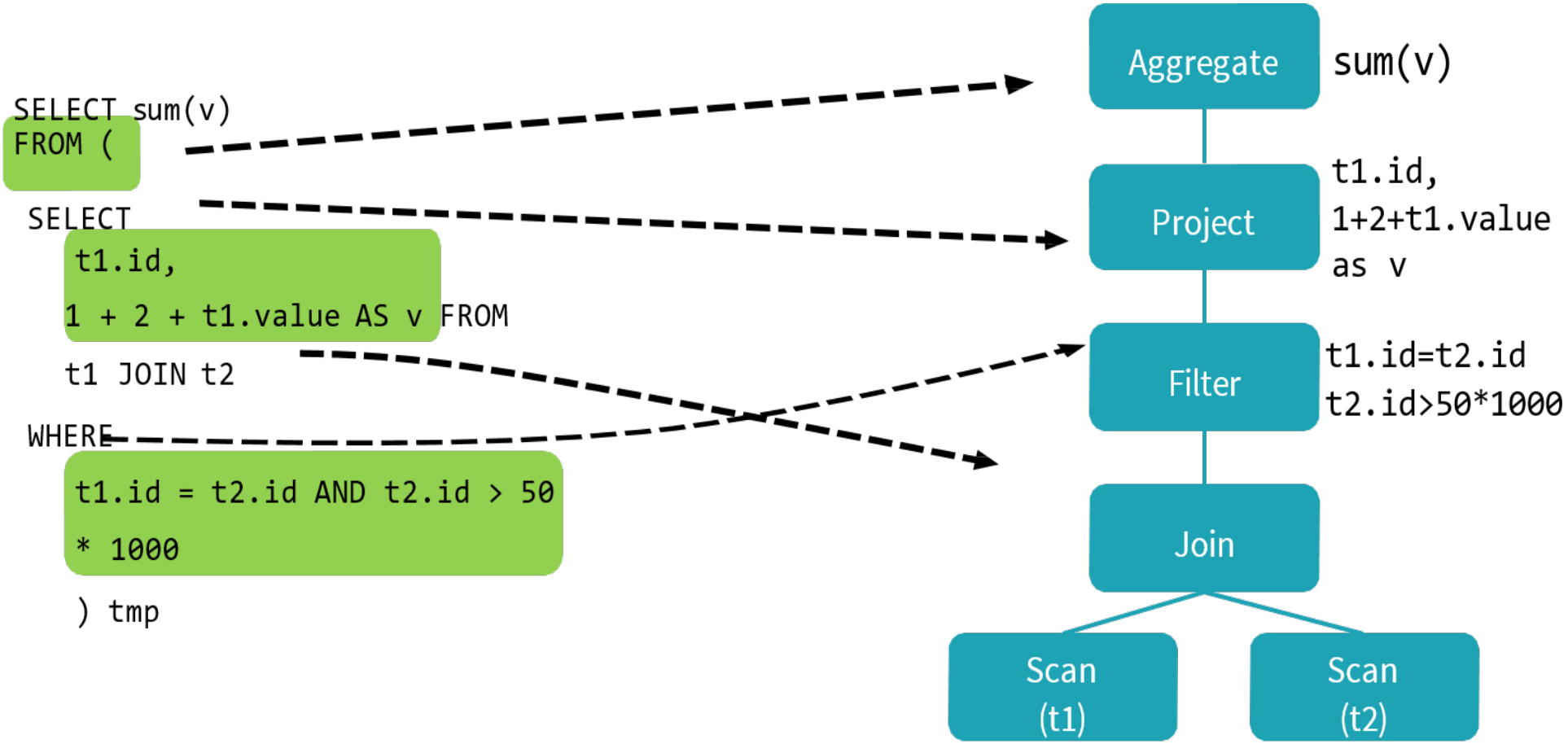
Expression

```
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
- Attribute: A column of a dataset (e.g. t1.id) or a column generated by a specific data operation (e.g. v)

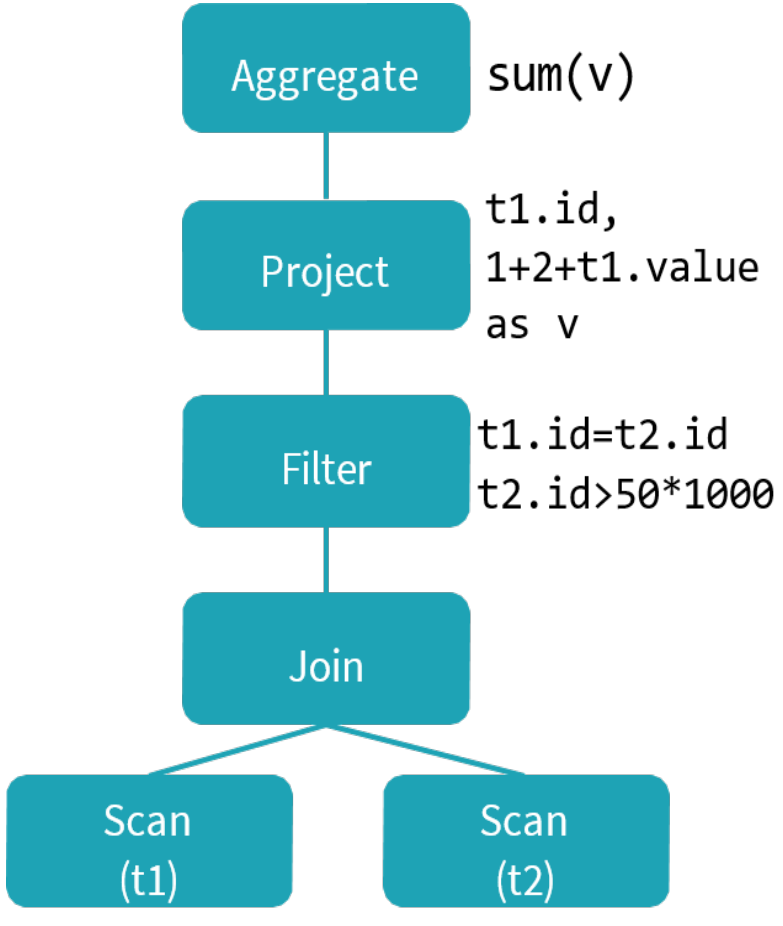
SQL to Logical plan

Expression



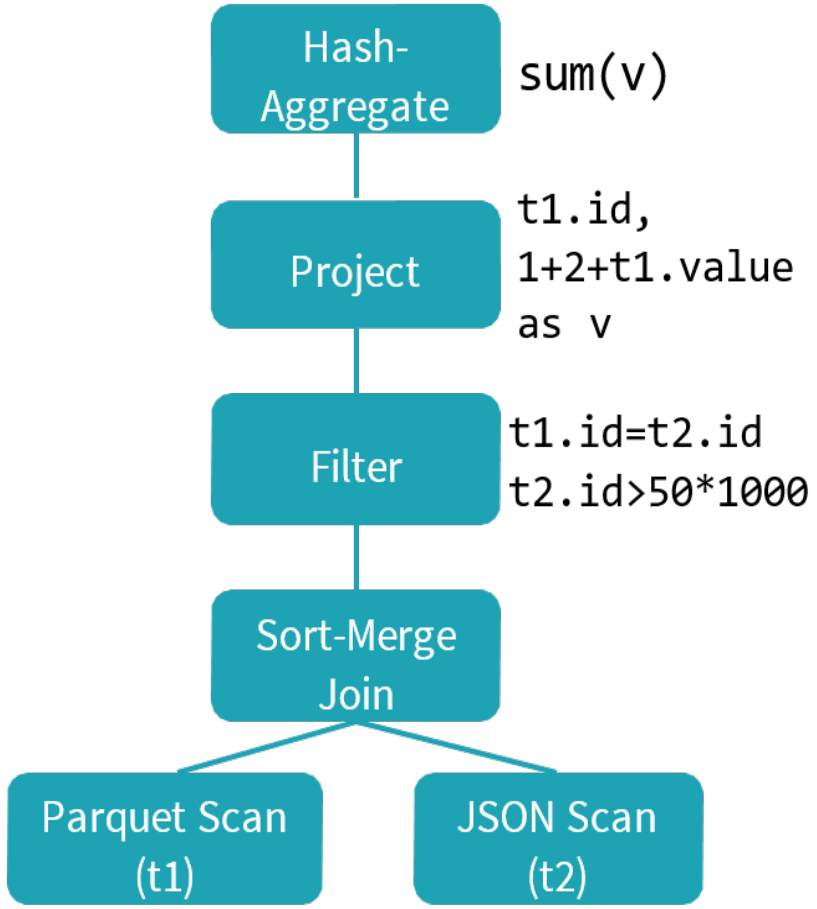
Logical Plan

- A Logical Plan describes computation on datasets without defining how to conduct the computation
- output: a list of attributes generated by this Logical Plan, e.g. [id, v]
- constraints: a set of invariants about the rows generated by this plan, e.g., t2.id > 50 * 1000



Physical Plan

- A Physical Plan describes computation on datasets with specific definitions on how to conduct the computation
- A physical plan is executable



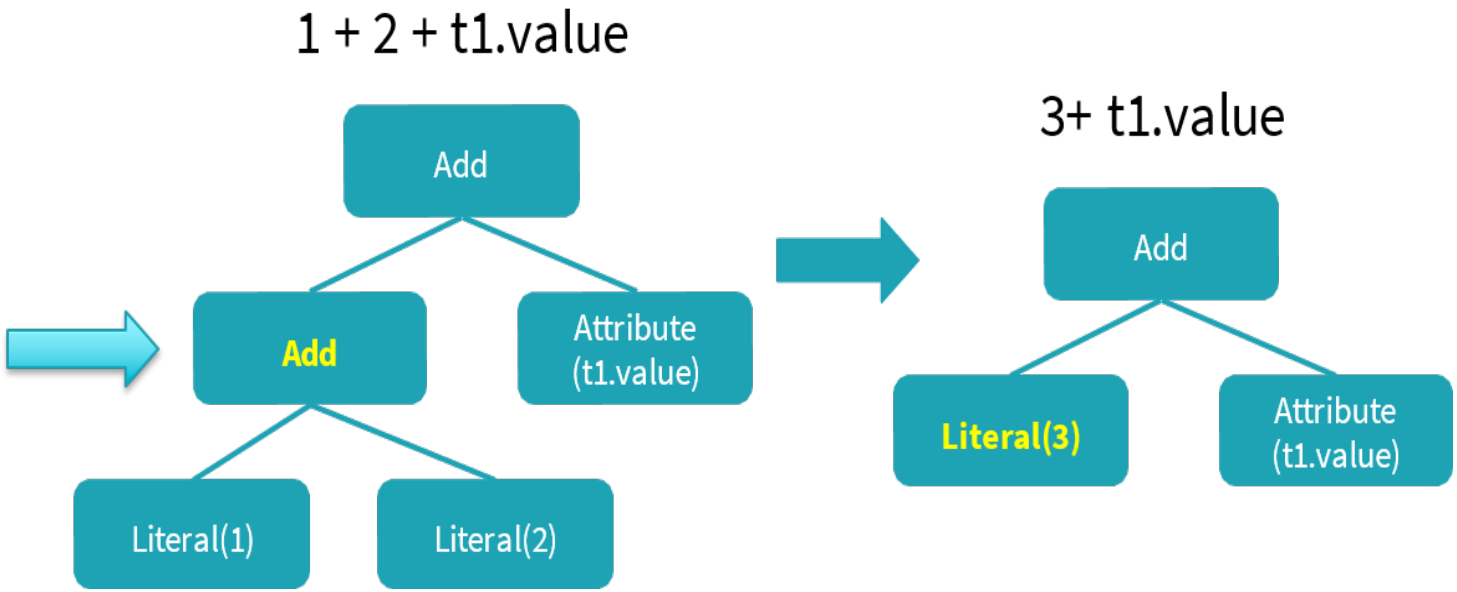
Optimization

- Transformations are used to optimize plans
- Plans should be logically equivalent
- Transformation is done via rules
  - Tree type preserving
    - Expression to Expression
    - Logical Plan to Logical Plan
    - Physical Plan to Physical Plan
  - Non-type preserving
    - Logical Plan to Physical Plan

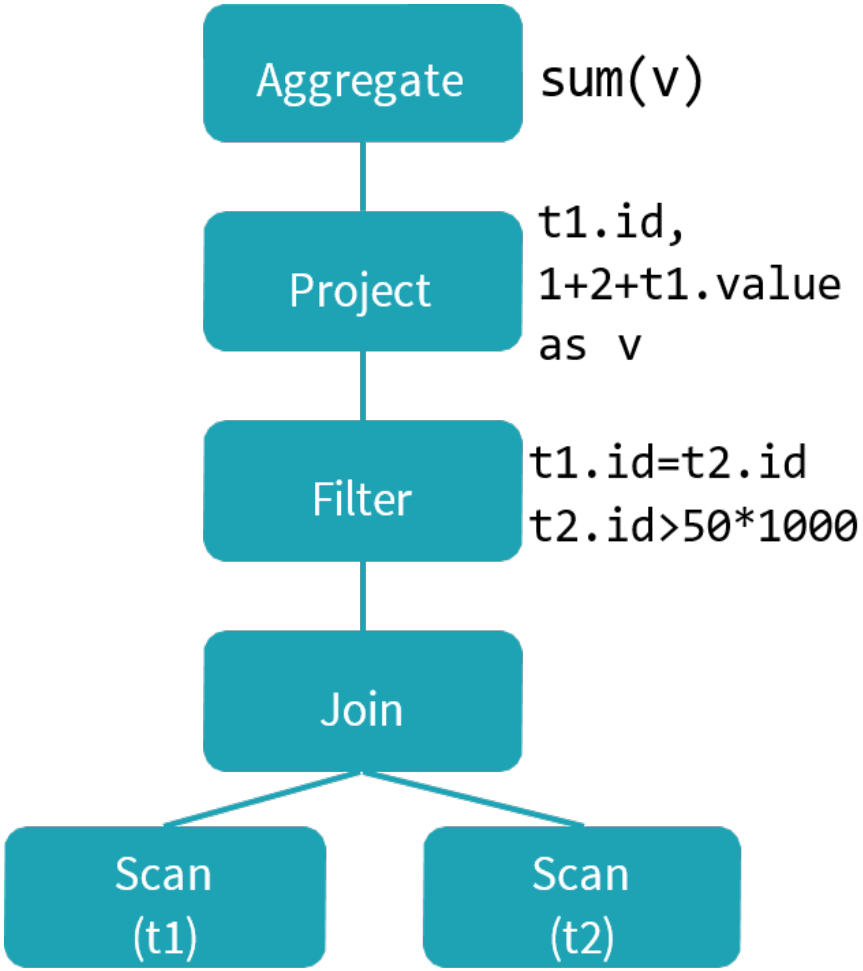
Transforms

- A transform is defined as a partial function
  - A partial function is a function that is defined for a subset of its arguments

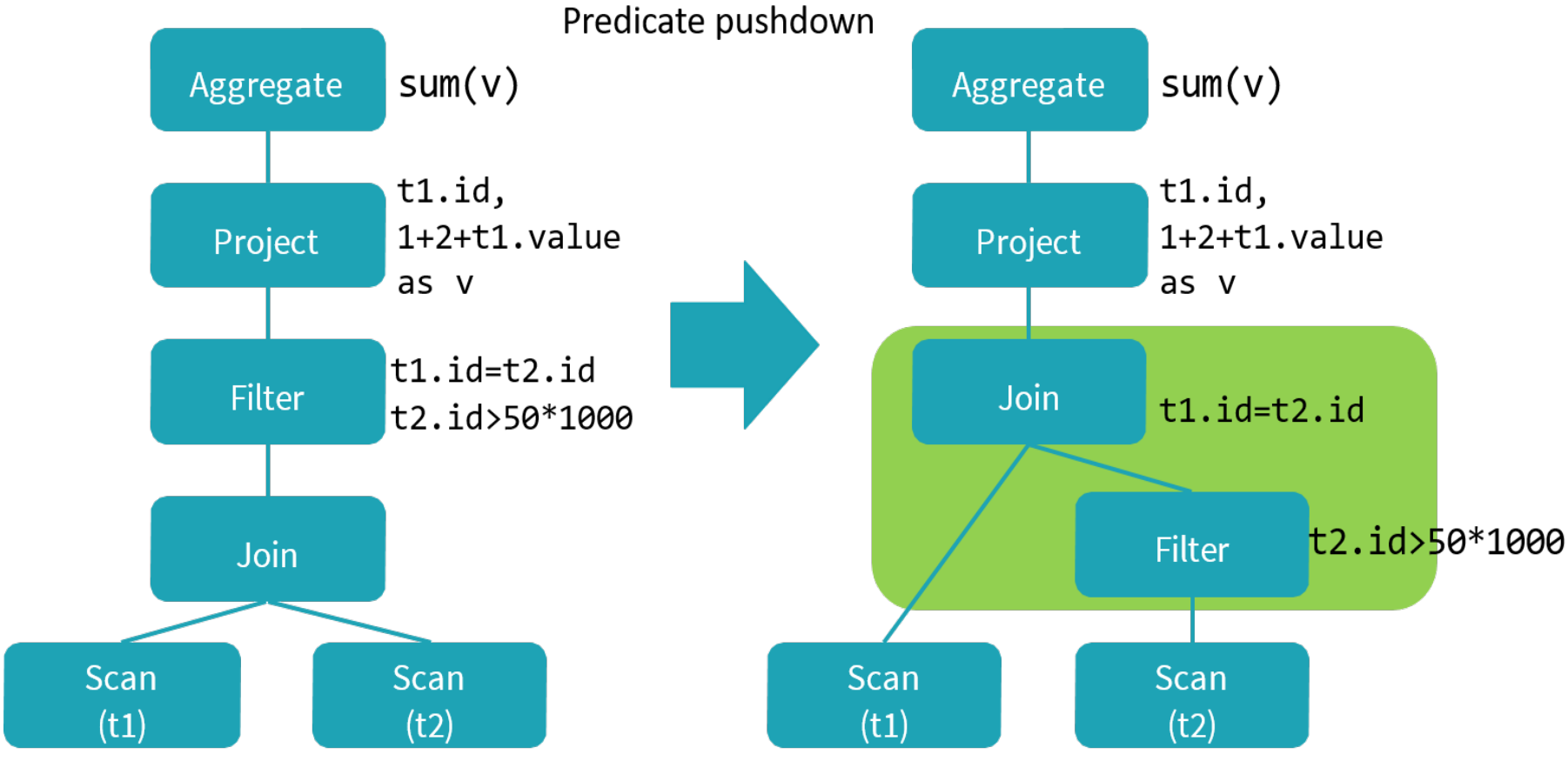
```
val expression: Expression =
... expression.transform {
case Add(Literal(x, IntegerType), Literal(y, IntegerType)) => L
iteral(x + y)
}
```



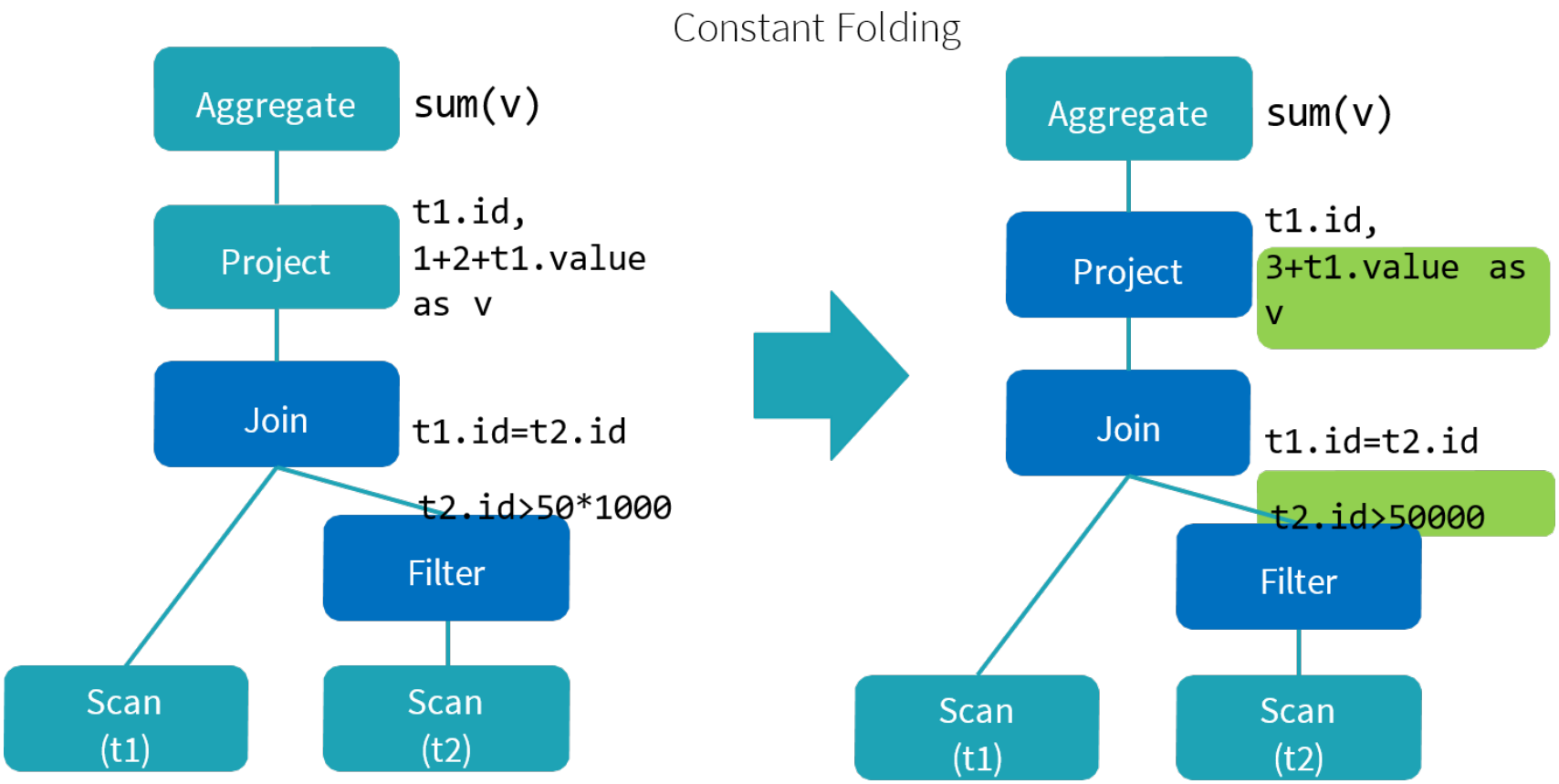
Combining Multiple Rules



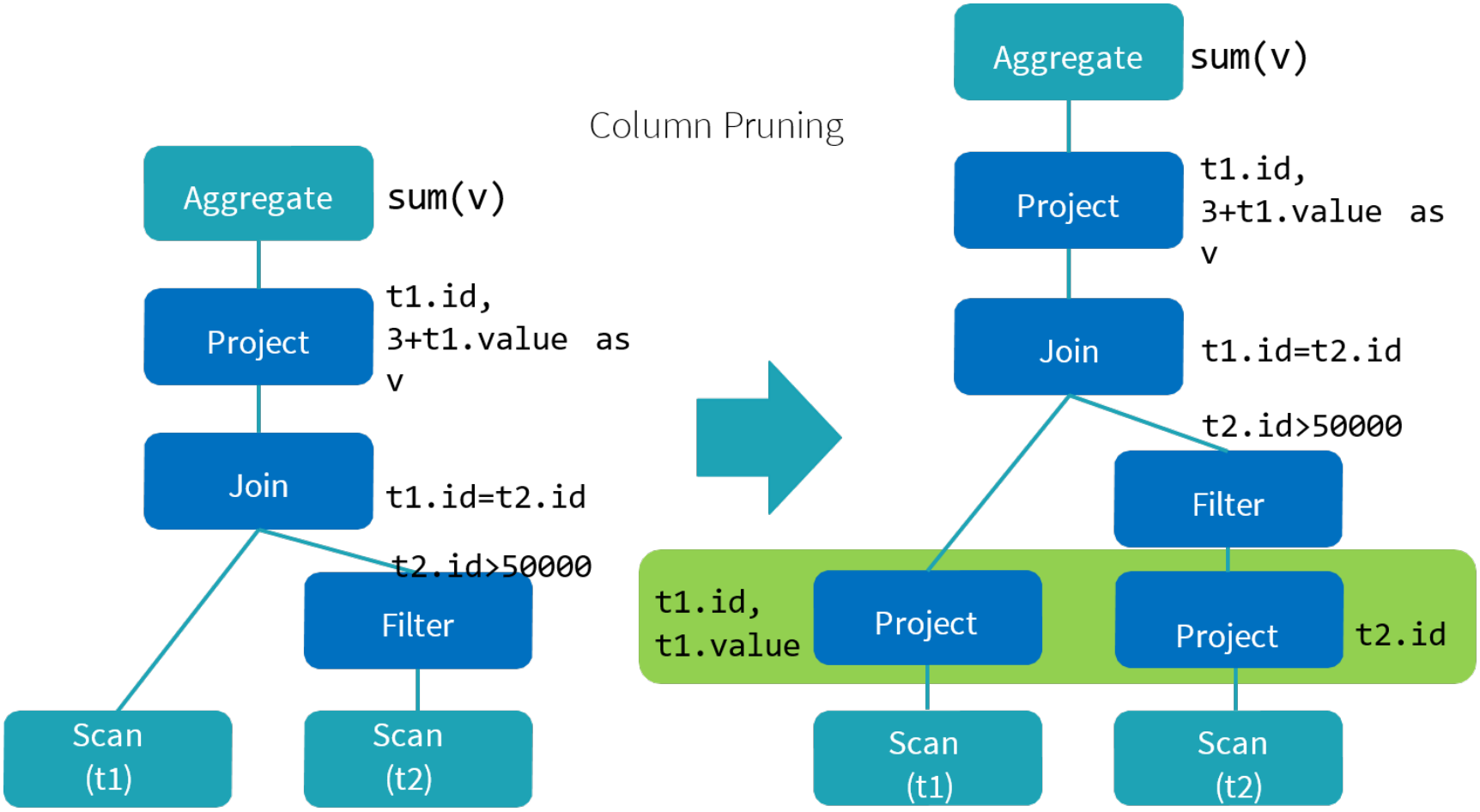
Combining Multiple Rules



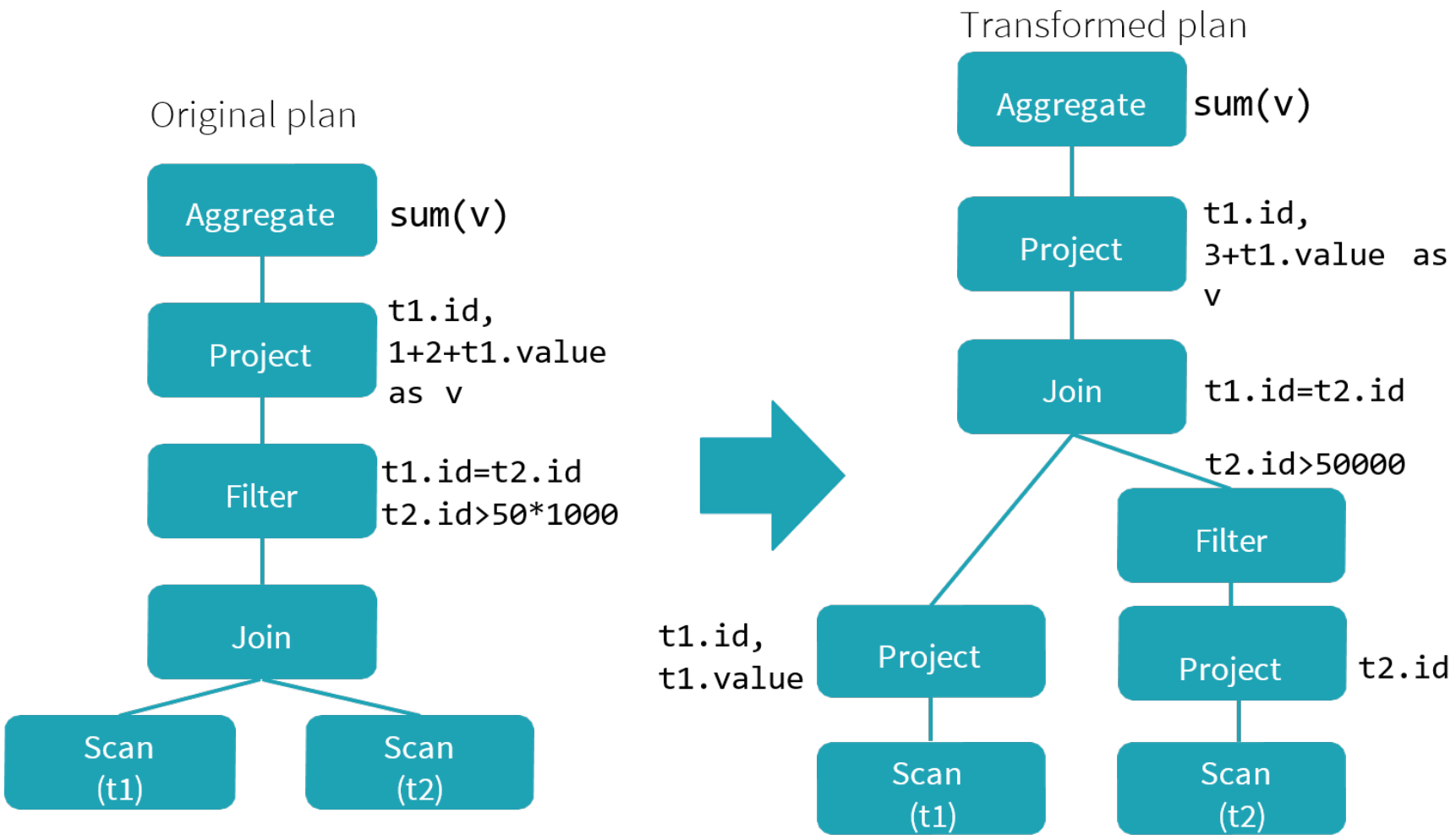
Combining Multiple Rules



Combining Multiple Rules



Combining Multiple Rules



Project Tungesten

- Explicit memory management: Leverage application semantics and data schema to eliminate JVM GC overhead,
- Cache-aware computation: exploit memory hierarchy,
- Code generation: using code generation to exploit modern CPU: It has been observed that Spark workloads are more constrained by CPU and memory rather than network or I/O.

HARDWARE TRENDS Over ~7years

Both Disk and Network I/O has improved 10x while throughput of CPU hardly changed

	2010	2016	Growth%
Storage	50Mbps (HDD)	500Mbps (SSD)	10x
Network	1Gbps	10Gbps	10x
CPU	~3GHz	~3GHz	—

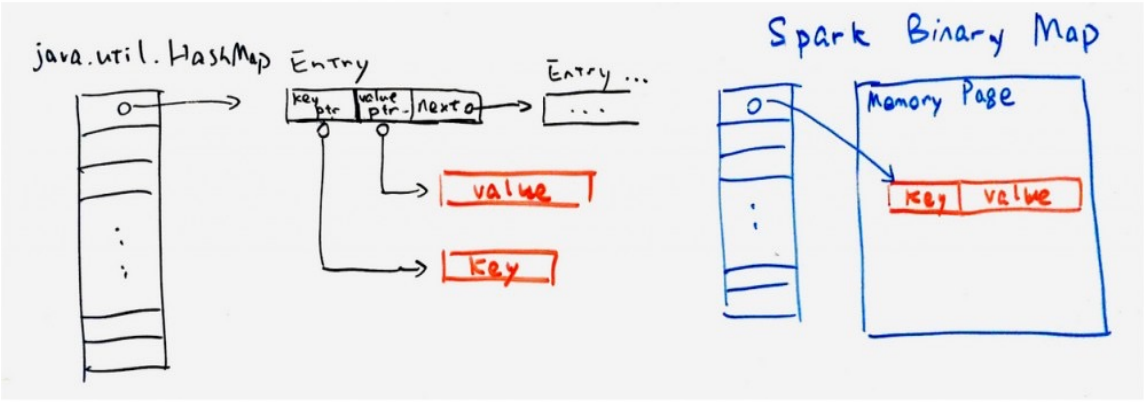
Off-Heap Memory Management

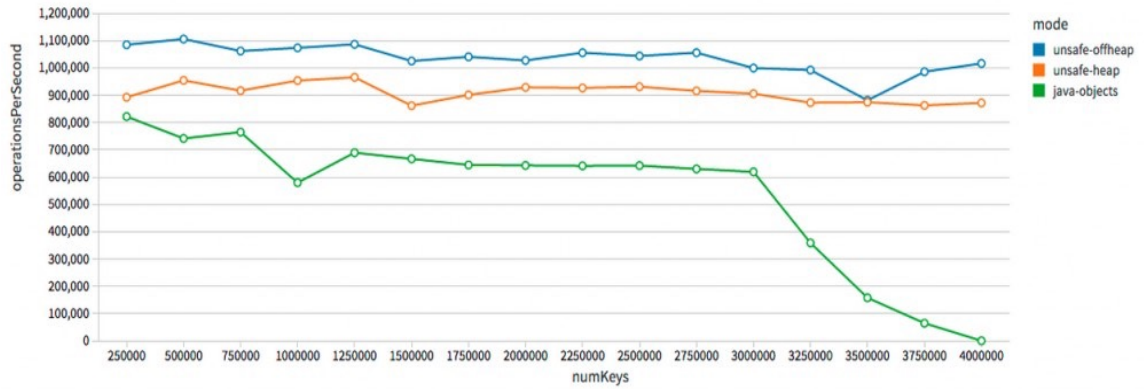
- JVM objects and GC overhead is non-negligible.
- Java objects have large inherent memory overhead
- Example string “abcd” that would take 4 bytes using UT-8 encoding takes 48 bytes when stored using JVM native String data type.
- String object (24 bytes) wrapping around the character array (24 bytes)

```
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: 24 bytes (reported by Instrumentation API)
```

Off-Heap Memory Management

- Spark to have its data stored in binary format
- Spark serializes/deserializes its own data
- Exploit data schema to reduce the overhead
- build on `sun.misc.unsafe` to give C-like memory access

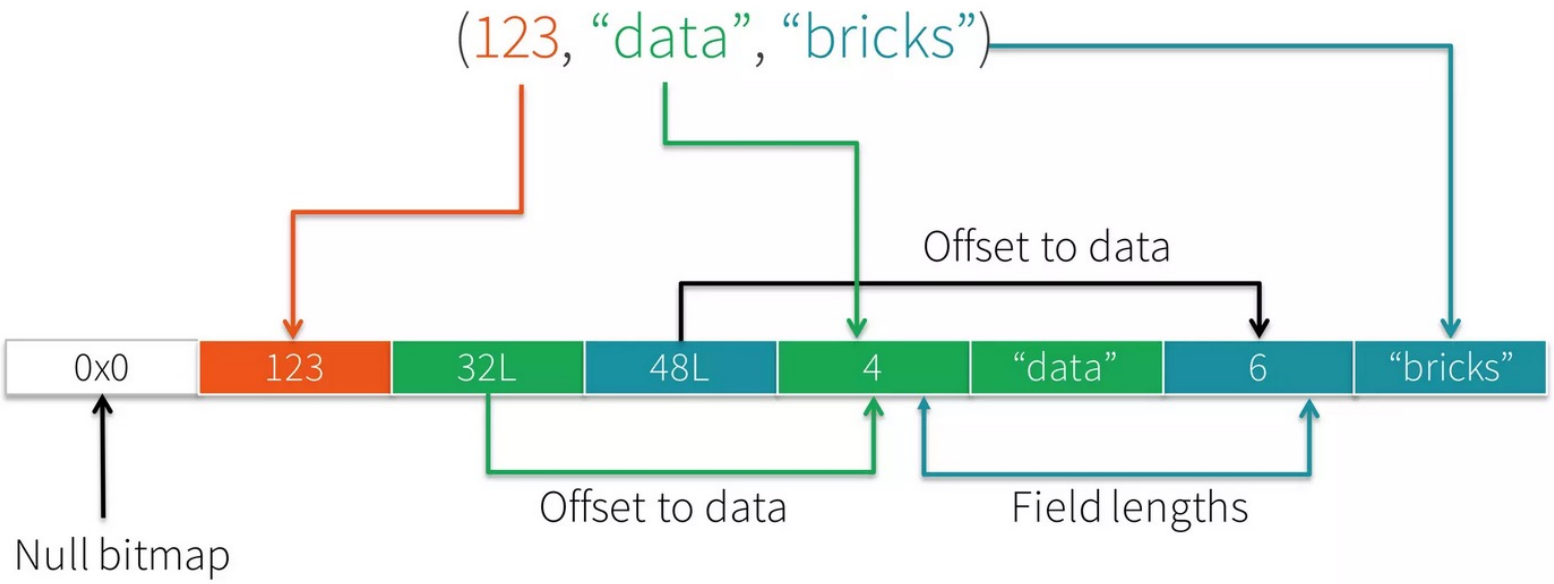




Off-Heap Memory Management

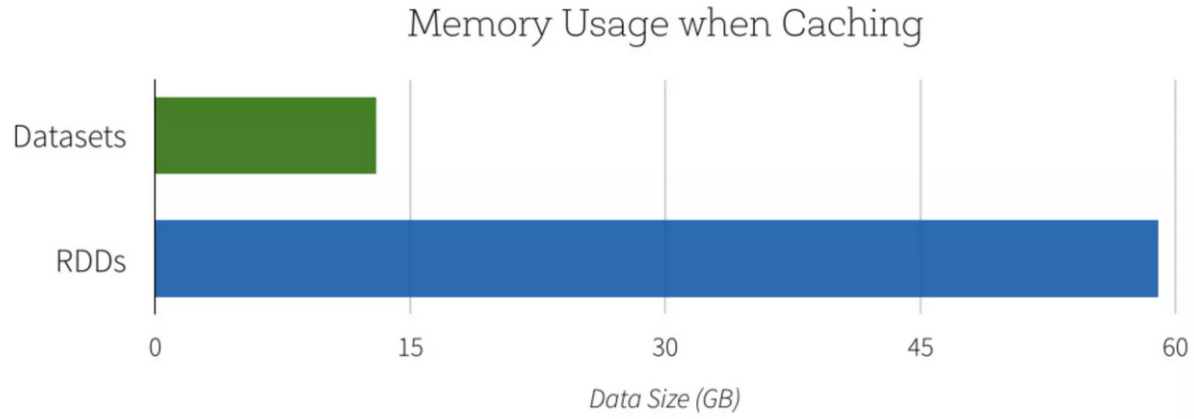
How is an object stored in the new Binary-Row-Format?

If the field is a primitive	With fixed length	Its stored in place
If the field is an Object	With variable length	1. Its offset is stored in place. 2. At the specified offset, we store: *length of the variable + followed by its data*

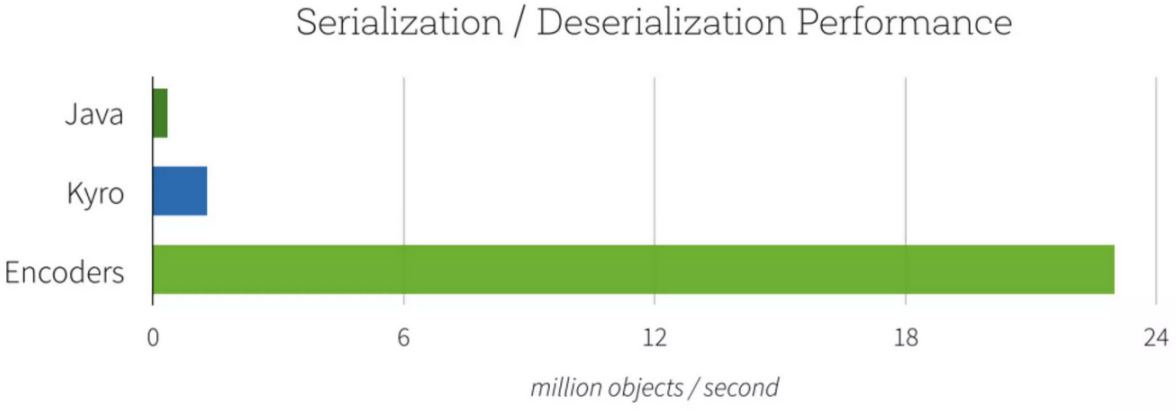


Off-Heap Memory Management

Space Efficiency

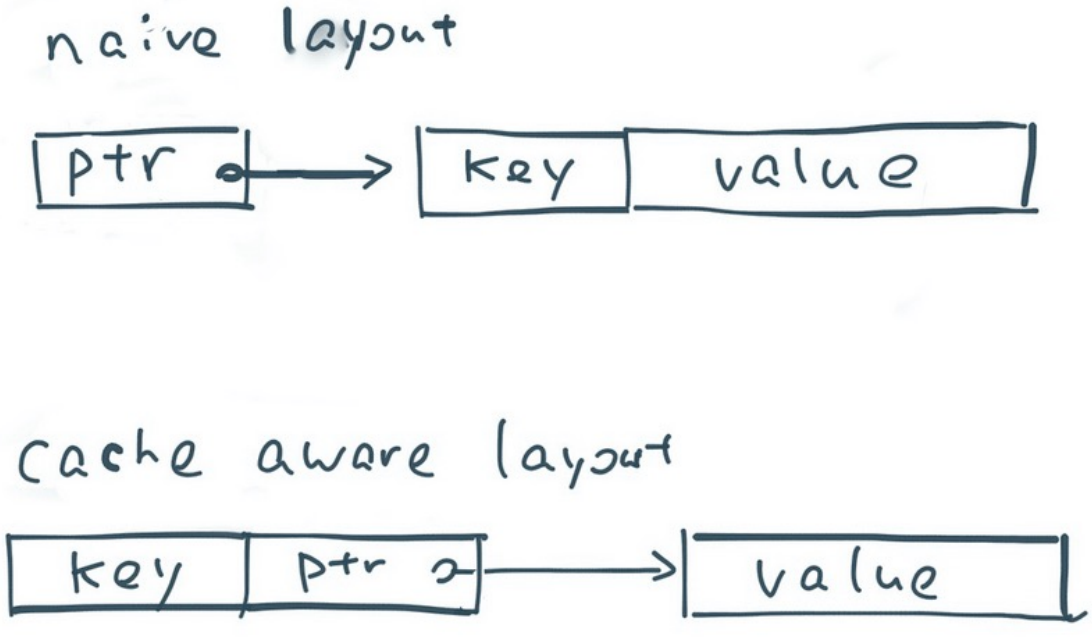


Serialization performance



Cache-aware Computation

- It was observed that Spark applications spend several CPU cycles waiting for data to be fetched from main memory
- Why not benefit from the memory hierarchy and pre-fetch data?
- 3X faster

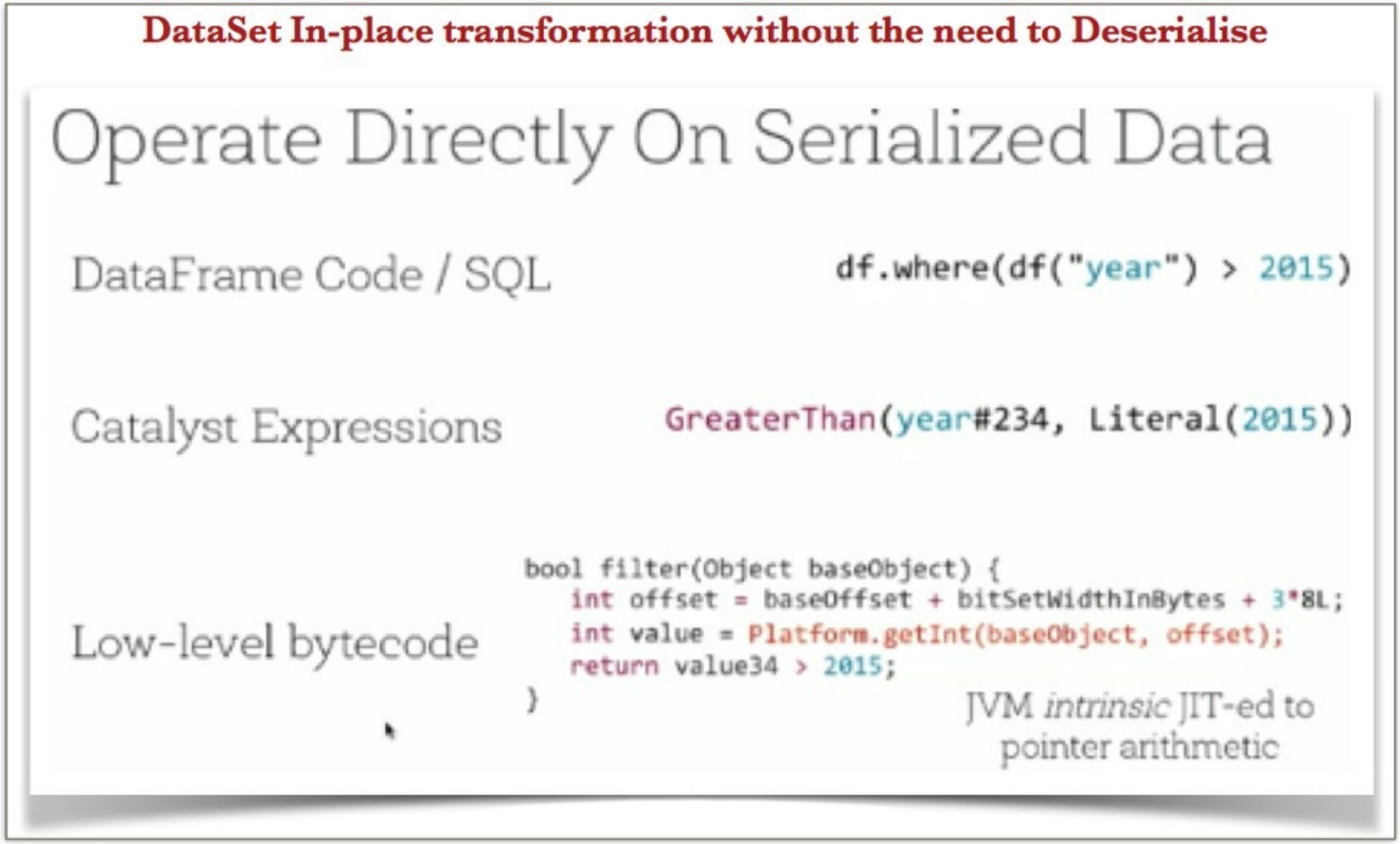


Code Generation

- Avoid parsing row-by-row at runtime,
- Imagine an expression (x+y)+1. Without code generation, lots of virtual function calls, which result in huge overhead in total,
- with code generation, at compile time and utilizing the data types, Spark can generate byte-code optimized for the specific data types

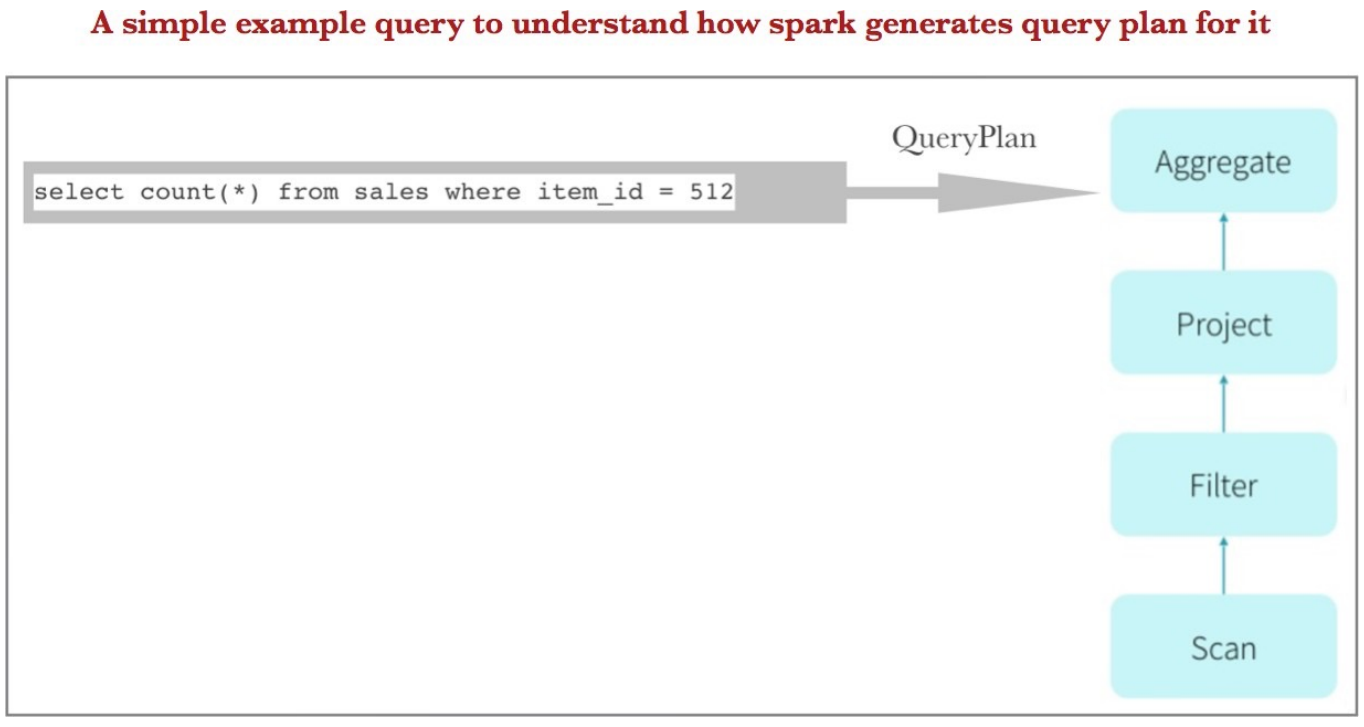
Example

Consider the case where we need to filter a dataframe by the column year: year >2015



Whole Stage Code Generation

Traditionally, Spark followed a volcano model



Volcano Iterator Model

- All operators, scan, filter, project, ..., etc. implement an `Iterator` interface
- A physical query plan is a chaining of operator where each operator iterates over the output of its parent operator, hence, data move up like lava in a volcano
- Operator-specific logic is applied and results are emitted to children operators
- Every handshake between 2 operators causes one virtual function call + reading parent output from memory + write the final output in memory

Issues with Volcano Model

- Too many virtual function calls
- Extensive memory access
- Unable to leverage lots of modern techniques: pipelining, pre-fetching, SIMD, loop unrolling, ..., etc.

How would a dedicated code look like?

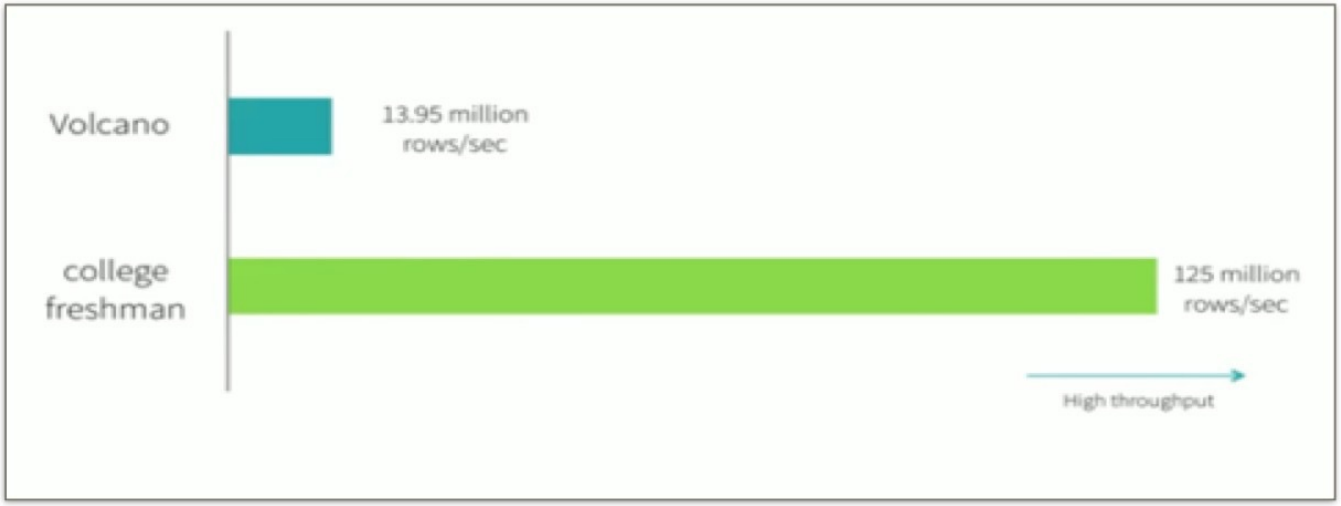
A College freshman would write the following code to do implement the same query

What if we hire a college freshman to implement this query in Java in 10 mins?

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

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

Dedicated versus Volcano



Why?

How does a student beat 30 years of research?

Volcano

1. Many virtual function calls
2. Data in memory (or cache)
3. No loop unrolling, SIMD, pipelining

hand-written code

1. No virtual function calls
2. Data in CPU registers
3. Compiler loop unrolling, SIMD, pipelining

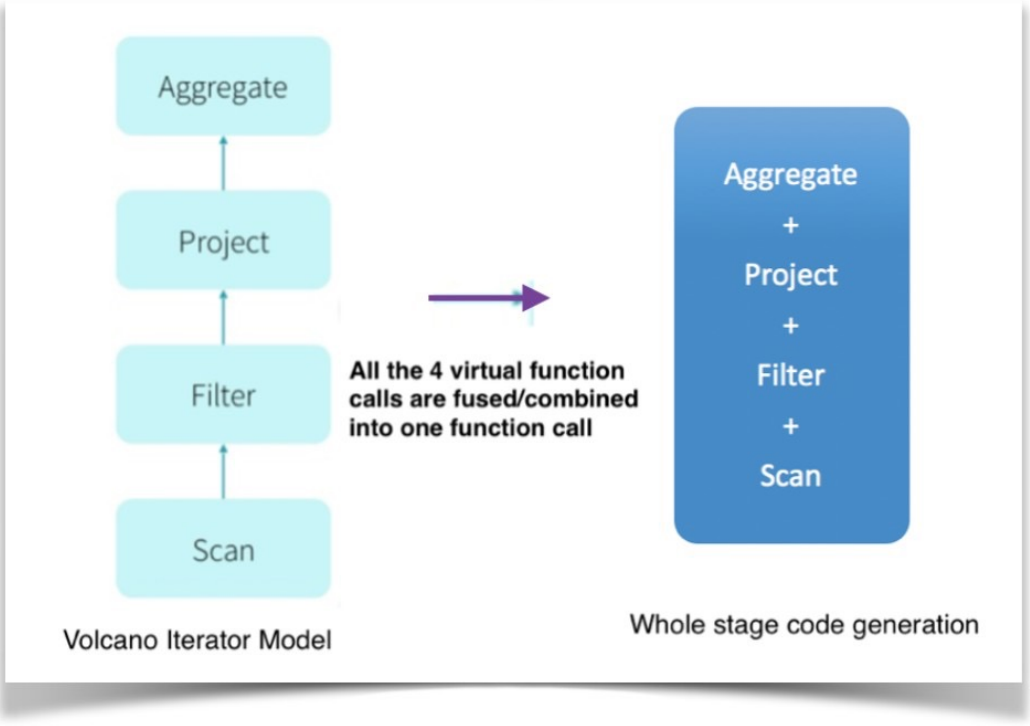
Take advantage of all the information that is known **after** query compilation

Whole-stage Code Generation

Target was to reach a functionality of a general-purpose execution engine like volcano model and Perform just like a hand built system that does exactly what user wants to do

- A new technique now popular in DB literature,
- Simply fuse together the operators so the generated code looks like hand-optimized

WholeStageCodeGeneration (**Vs**) VolcanoIteratorModel

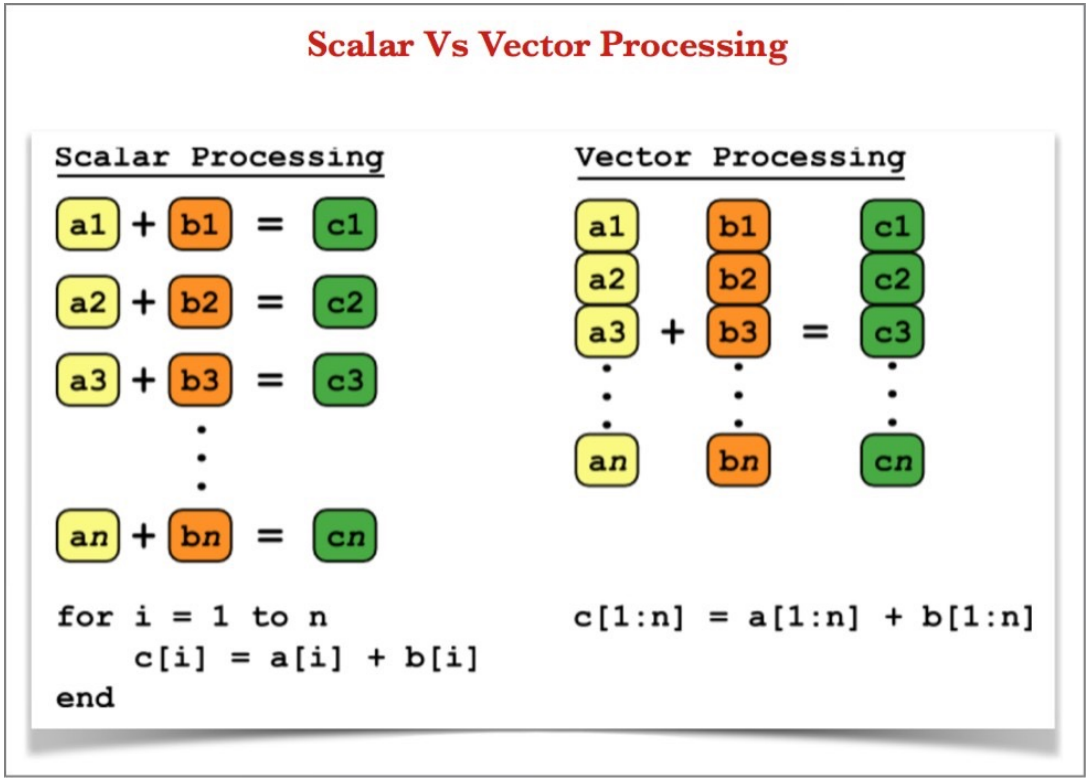


Vectorized Execution: in-memory columnar storage

After WSCG, we can still speedup the execution of the generated code. How? Vectorization.

- The idea is to take advantage of data level parallelism (DLP) in modern CPUs. That is, to process data in batches of rows rather than one row at a time.
- Shift from row-based to columnar-based storage

Scalar versus Vector Processing



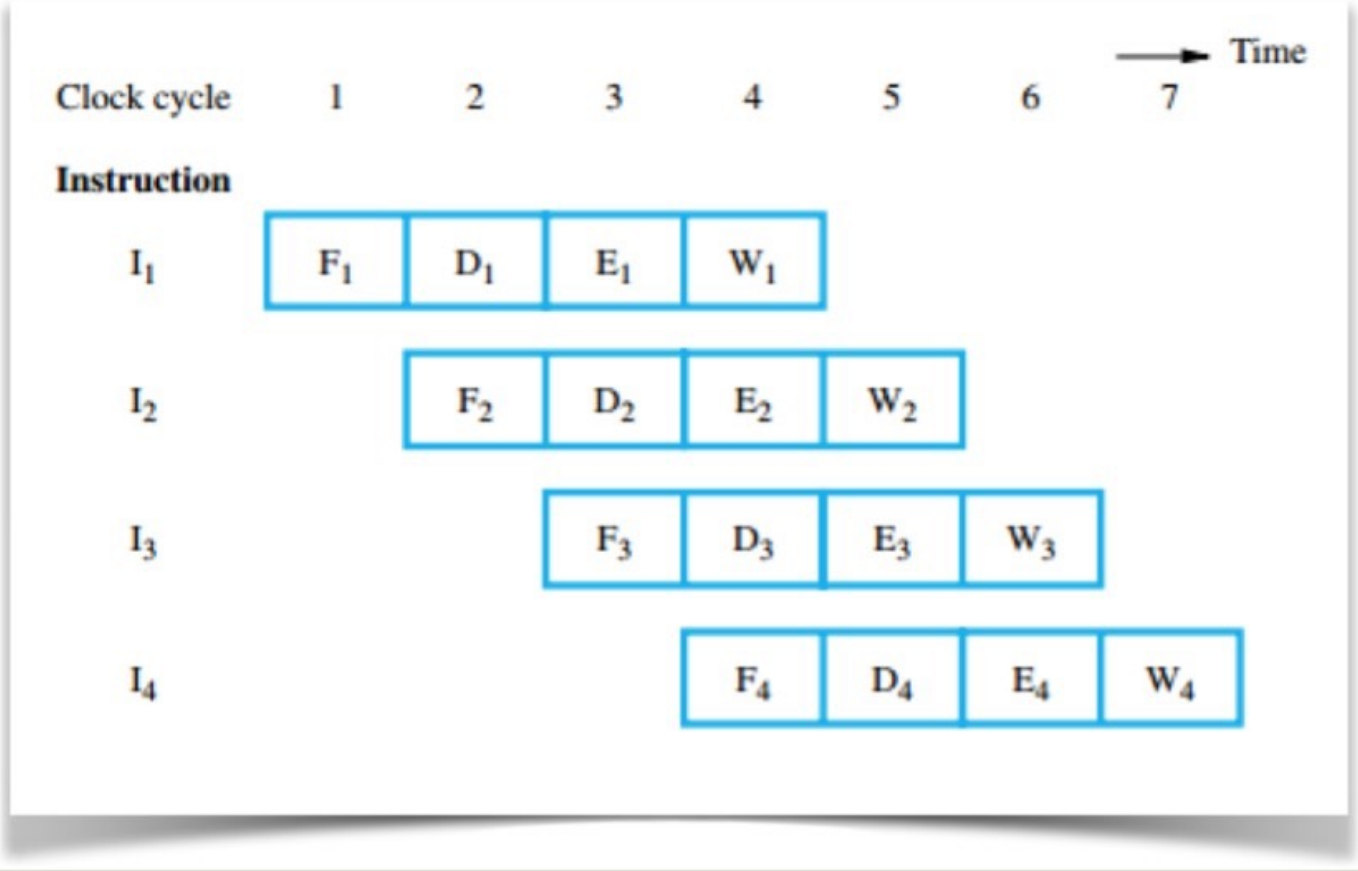
Data availability for vectorized processing

- Columnar data storage. Benefits
  - Simple access versus complex off-set in row-based formats
  - Denser storage
  - Compatibility with in-memory cache
  - Enables harnessing more benefits from hardware, e.g. GPU
- Avoiding CPU stalls
  - Keep data as close in CPU registers to avoid idle cycles
  - We have 4 operations F=fetch, D=decode, E=execute, W=write, if data is missing, it has to be fetched from lower (slower) memory .

Ideal execution without CPU stalls

Ideal pipeline without any CPU stalling

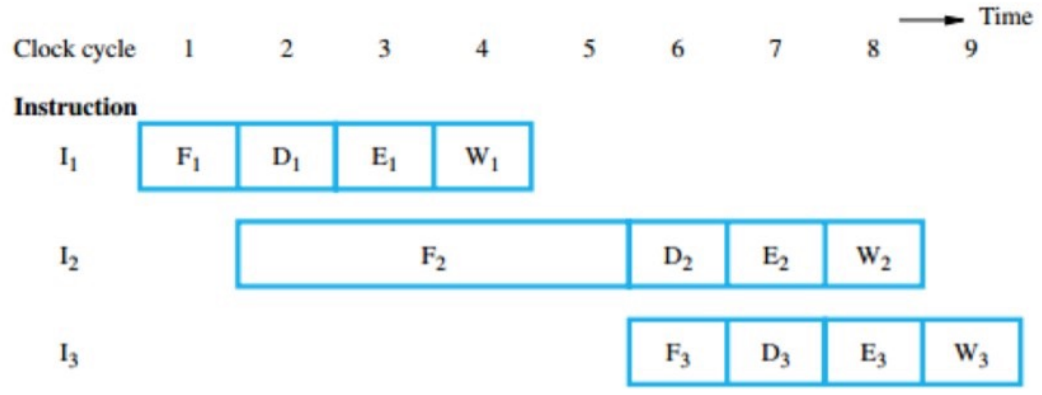
[F - Fetch, D - Decode, E - Execute, W - Write are 4 stages of InstructionCycle]



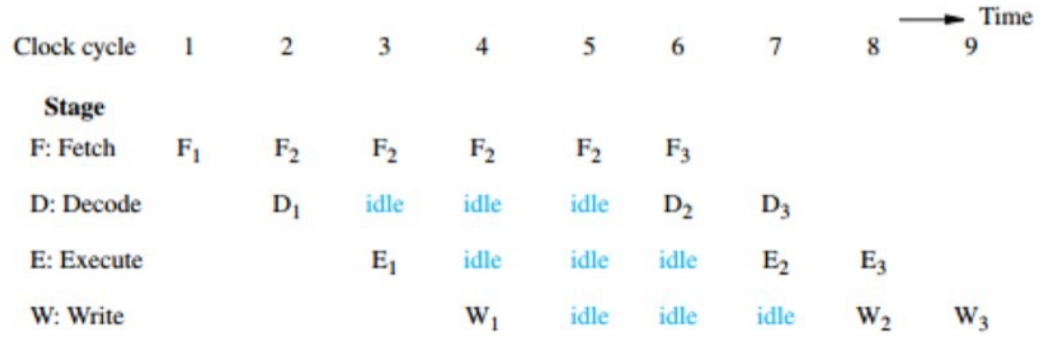
Execution with CPU stalls

Impact of Cache miss a.k.a Data Unavailability on pipeline

[F - Fetch, D - Decode, E - Execute, W - Write are 4 stages of InstructionCycle]



(a) Instruction execution steps in successive clock cycles



(b) Function performed by each processor stage in successive clock cycles

Benchmarking Big SQL Systems

- Several systems provide SQL-like access to big data
- How do they perform?
- What aspects affect their performance?
- Benchmarking studies are important to give researchers and practitioners insight about the capabilities of the different systems
  - One such study was conducted in Victor Aluko and Sherif Sakr : Big SQL Systems: An Experimental Evaluation , 2018

Benchmark Scope

- Systems: Hive, Impala, Spark SQL, and PrestoDB
- Benchmarks: TPC-H, TPC-DS
- Hardware setup: A cluster of 5 nodes, each node with Intel Xeon 2.4GHz with 16 cores, 64 GB of RAM, 1.2 TB SSD with disk speed 300MB/s
- Metrics: Response time, CPU, memory, disk and network utilization
- Data Formats: text, ORC, Paruet

TPC-H Results