- Yogesh Simmhan
- simmhan@iisc.ac.in

- Department of Computational and Data Sciences
- Indian Institute of Science, Bangalore

DS256 (3:1)

Scalable Systems for Data Science

Module 2

Processing Large Volumes of Big Data

- Programming Assignment 1 on Spark Data Frames posted on 7 Feb.
- Due on 28 Feb.
- Teams of two
- 20% weightage
- No external help, GenAI coding, etc.

“

- *Quiz 1 on Modules 1 & 2 on Thu, 12 Feb*
- *All topics till today’s class*

Spark DataFrames & SQL

Spark SQL: Relational Data Processing in Spark, Michael Armbrust, et al., *ACM SIGMOD* 2015

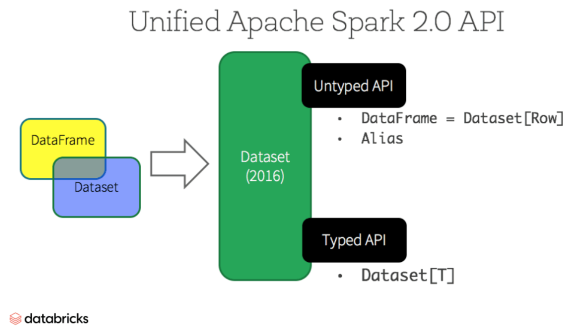
Limitations of Spark RDD

- Spark only offers high-level constructs on iterations over RDD items, invocation pattern
- Lambda expressions or functions are opaque to Spark
  - No optimizations of execution
- Types are opaque (other than being homogeneous)
  - No type-specific behavior
- Limitations on efficient execution
  - **Imperative programming**: Users tell *how* to execute
  - Relies on users to optimize code

DataFrames

- DataFrame: Inspired by pandas
  - RDD: Inspired by Python native operators like map
- More expressive and simpler
  - Compose a SQL-like query…
  - Using high-level DSL operators and APIs
- Tell Spark *what* to do
  - Spark can parse query, understand our intention
  - Optimize/arrange operations for efficient execution
- Even more uniformity across language bindings
  - *Avoid user-defined code!*
- *Allows you to drop down to RDD also!*

RDD vs. Datasets/DataFrames



- **RDD**: Immutable distributed collection
  - Transformations, actions, partitions
  - Lower-level abstraction. Better control. More coding required
- **DataFrame**: Immutable distributed collection (like RDD)
  - Data is organized into *named columns*
    - Imposes a structure, easier abstraction
    - Makes processing large data sets easier
  - Domain specific language API
- **Dataset**: *Strongly-typed* flavor of DataFrame
  - Only Java or Scala

https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

RDD vs. Datasets/Data Frames

- Static-typing and runtime type-safety
  - Dataset easier to debug due to strong typing
  - RDD on Java/Scala is strongly typed
- High-level abstraction due to typing
- High-level abstraction due to columnar
  - RDD is row-based, Dataset/DF is both row & column based

	SparkSQL	DataFrames
Syntax Errors	Runtime	Compile Time
Analysis Errors	Runtime	Runtime

https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

RDD vs. Datasets/Data Frames

- Performance
  - Dataset and DataFrame use SparkSQL Catalyst optimizer…speed and space efficiency
  - Dataset efficient fast for *batch* execution. Specialized Tungsten serialization/deserialization and compact bytecode
  - DataFrame is untyped and faster for *interactive*
  - RDD gives fine-grained control
- You can move across them
  - dataset.rdd.take(10)

Declarative Programming

- Tell ***what*** to do, not *how* to do it
  - System determines the best plan
  - E.g., SQL and RDBMS query planning

  - `SELECT Part,Items FROM Widget WHERE Part='Bolts'`

- Uses schema for the data/table to plan the execution

RDD vs. DataFrames

```
# Create an RDD of tuples (name, age)
dataRDD = sc.parallelize([("Brooke", 20), ("Denny", 31), ("Jules", 30),
  ("TD", 35), ("Brooke", 25)])
# Use map and reduceByKey transformations with their lambda
# expressions to aggregate and then compute average

agesRDD = (dataRDD
  .map(lambda x: (x[0], (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])))
```

```
+------+--------+
|  name|avg(age)|
+------+--------+
|Brooke|    22.5|
| Jules|    30.0|
|    TD|    35.0|
| Denny|    31.0|
+------+--------+
```

```
# Create a DataFrame using SparkSession
spark = (SparkSession
  .builder
  .appName("AuthorsAges")
  .getOrCreate())
# Create a DataFrame
data_df = spark.createDataFrame([("Brooke", 20), ("Denny", 31), ("Jules", 30),
  ("TD", 35), ("Brooke", 25)], ["name", "age"])
# Group the same names together, aggregate their ages, and compute an average
avg_df = data_df.groupBy("name").agg(avg("age"))
# Show the results of the final execution
avg_df.show()
```

DataFrames

- Distributed in-memory **tables** with named columns and schemas
  - Define *schema* explicitly by user
  - But **immutable** … Spark keeps a lineage of all transformations
  - Add or change column names/types, creates new DataFrames while the previous versions are preserved

Table 3-1. The table-like format of a DataFrame

Id (Int)	First (String)	Last (String)	Url (String)	Published (Date)	Hits (Int)	Campaigns (List[Strings])
1	Jules	Damji	https://tinyurl.1	1/4/2016	4535	[twitter, LinkedIn]
2	Brooke	Wenig	https://tinyurl.2	5/5/2018	8908	[twitter, LinkedIn]
3	Denny	Lee	https://tinyurl.3	6/7/2019	7659	[web, twitter, FB, LinkedIn]
4	Tathagata	Das	https://tinyurl.4	5/12/2018	10568	[twitter, FB]

DataFrames: Define Schema using DDL

```
# Define schema for our data using DDL
schema = "`Id` INT, `First` STRING, `Last` STRING, `Url` STRING,
  `Published` STRING, `Hits` INT, `Campaigns` ARRAY<STRING>"

# Create our static data
data = [[1, "Jules", "Damji", "https://tinyurl.1", "1/4/2016", 4535, ["twitter",
"LinkedIn"]],
       [2, "Brooke","Wenig", "https://tinyurl.2", "5/5/2018", 8908, ["twitter",
"LinkedIn"]],
       [3, "Denny", "Lee", "https://tinyurl.3", "6/7/2019", 7659, ["web",
"twitter", "FB", "LinkedIn"]],
       [4, "Tathagata", "Das", "https://tinyurl.4", "5/12/2018", 10568,
["twitter", "FB"]],
       [5, "Matei","Zaharia", "https://tinyurl.5", "5/14/2014", 40578, ["web",
"twitter", "FB", "LinkedIn"]],
       [6, "Reynold", "Xin", "https://tinyurl.6", "3/2/2015", 25568,
["twitter", "LinkedIn"]]
      ]
# Create a DataFrame using the schema defined above
blogs_df = spark.createDataFrame(data, schema)
# Show the DataFrame; it should reflect our table above
blogs_df.show()
# Print the schema used by Spark to process the DataFrame
print(blogs_df.printSchema())
```

```
+-------+---------+-------+-----------------+---------+-----+------------------+
|Id     |First    |Last   |Url              |Published|Hits |Campaigns         |
+-------+---------+-------+-----------------+---------+-----+------------------+
|1      |Jules    |Damji  |https://tinyurl.1|1/4/2016 |4535 |[twitter,...]     |
|2      |Brooke   |Wenig  |https://tinyurl.2|5/5/2018 |8908 |[twitter,...]     |
|3      |Denny    |Lee    |https://tinyurl.3|6/7/2019 |7659 |[web, twitter...] |
|4      |Tathagata|Das    |https://tinyurl.4|5/12/2018|10568|[twitter, FB]     |
|5      |Matei    |Zaharia|https://tinyurl.5|5/14/2014|40578|[web, twitter,...]|
|6      |Reynold  |Xin    |https://tinyurl.6|3/2/2015 |25568|[twitter,...]     |
+-------+---------+-------+-----------------+---------+-----+------------------+
```

Projections & Filters

```
+--------------+----------------------+-------------+
|IncidentNumber|AvailableDtTm         |CallType     |
+--------------+----------------------+-------------+
|2003235       |01/11/2002 01:47:00 AM|Structure Fire|
|2003235       |01/11/2002 01:51:54 AM|Structure Fire|
|2003235       |01/11/2002 01:47:00 AM|Structure Fire|
```

```python
# In Python
few_fire_df = (fire_df
  .select("IncidentNumber", "AvailableDtTm", "CallType")
  .where(col("CallType") != "Medical Incident"))
few_fire_df.show(5, truncate=False)
```

```python
# In Python, filter for only distinct non-null CallTypes from all the rows
(fire_df
  .select("CallType")
  .where(col("CallType").isNotNull())
  .distinct()
  .show(10, False))
```

```
+-----------------------------------+
|CallType                           |
+-----------------------------------+
|Elevator / Escalator Rescue        |
|Marine Fire                        |
|Aircraft Emergency                 |
|Confined Space / Structure Collapse|
```

Aggregation & Join

```
# In Python
(fire_ts_df
  .select("CallType")
  .where(col("CallType").isNotNull())
  .groupBy("CallType")
  .count()
  .orderBy("count", ascending=False)
  .show(n=10, truncate=False))
```

```
+-----------------------------+-------+
|CallType                     |count  |
+-----------------------------+-------+
|Medical Incident             |2843475|
|Structure Fire               |578998 |
|Alarms                       |483518 |
|Traffic Collision            |175507 |
|Citizen Assist / Service Call|65360  |
```

Here, count() is the aggregator for groupBy() & not the action df.count()

```
# In Python
# Join departure delays data (foo) with airport info
foo.join(
  airports,
  airports.IATA == foo.origin
).select("City", "State", "date", "delay", "distance", "destination").show()
```

City	State	Country	IATA
Seattle	WA	USA	SEA
New York	NY	USA	JFK
Bangalore	KA	India	BLR

Origin	Destination	Date	Delay	Distance	Airline
SEA	SFO	01010710	31	590	AA
SEA	SFO	01010955	104	590	UA
SEA	SFO	01010730	5	590	AA
LAX	SFO	01010600	15	400	DL

```
+-------+-----+--------+-----+--------+-----------+
|   City|State|    date|delay|distance|destination|
+-------+-----+--------+-----+--------+-----------+
|Seattle|   WA|01010710|   31|     590|        SFO|
|Seattle|   WA|01010955|  104|     590|        SFO|
|Seattle|   WA|01010730|    5|     590|        SFO|
+-------+-----+--------+-----+--------+-----------+
```

Lazy Evaluation in DF

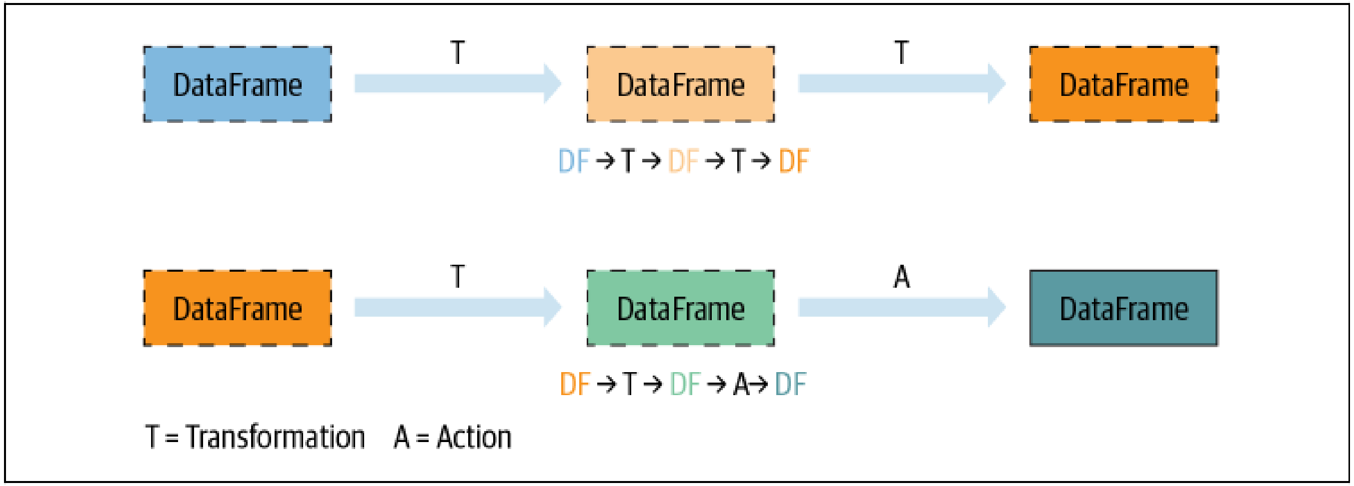


Figure 2-6. Lazy transformations and eager actions

Transformations	Actions
`orderBy()`	`show()`
`groupBy()`	`take()`
`filter()`	`count()`
`select()`	`collect()`
`join()`	`save()`

Immutability vs Modifications in DF

- DataFrames themselves are immutable
  - Backed by RDDs
- You can modify them to create new, different DataFrames
  - **Add Columns**: Original *foo* DataFrame plus the additional *status* derived column

```python
# In Python
from pyspark.sql.functions import expr
foo2 = (foo.withColumn(
          "status",
          expr("CASE WHEN delay <= 10 THEN 'On-time' ELSE 'Delayed' END")
        ))
```

  - Drop Cols: `foo3 = foo2.drop("delay")`
  - Rename Cols: `foo4 = foo3.withColumnRenamed("status", "flight_status")`
- ***ACID does not apply since immutable***

Spark SQL

- ANSI SQL:2003–compatible queries on structured data with a schema
- Permits abstraction to DataFrames/Datasets
- Connects to Apache Hive, JSON, CSV, Parquet,
- JDBC/ODBC and SQL Shell
- Optimized query plans

```python
# In Python
count_mnm_df = (mnm_df
  .select("State", "Color", "Count")
  .groupBy("State", "Color")
  .agg(count("Count")
  .alias("Total"))
  .orderBy("Total", ascending=False))
```

```sql
-- In SQL
                                    count
SELECT State, Color, ~~Count~~, ~~sum~~(Count) AS Total
FROM MNM_TABLE_NAME
GROUP BY State, Color, ~~Count~~
ORDER BY Total DESC
```

Joins

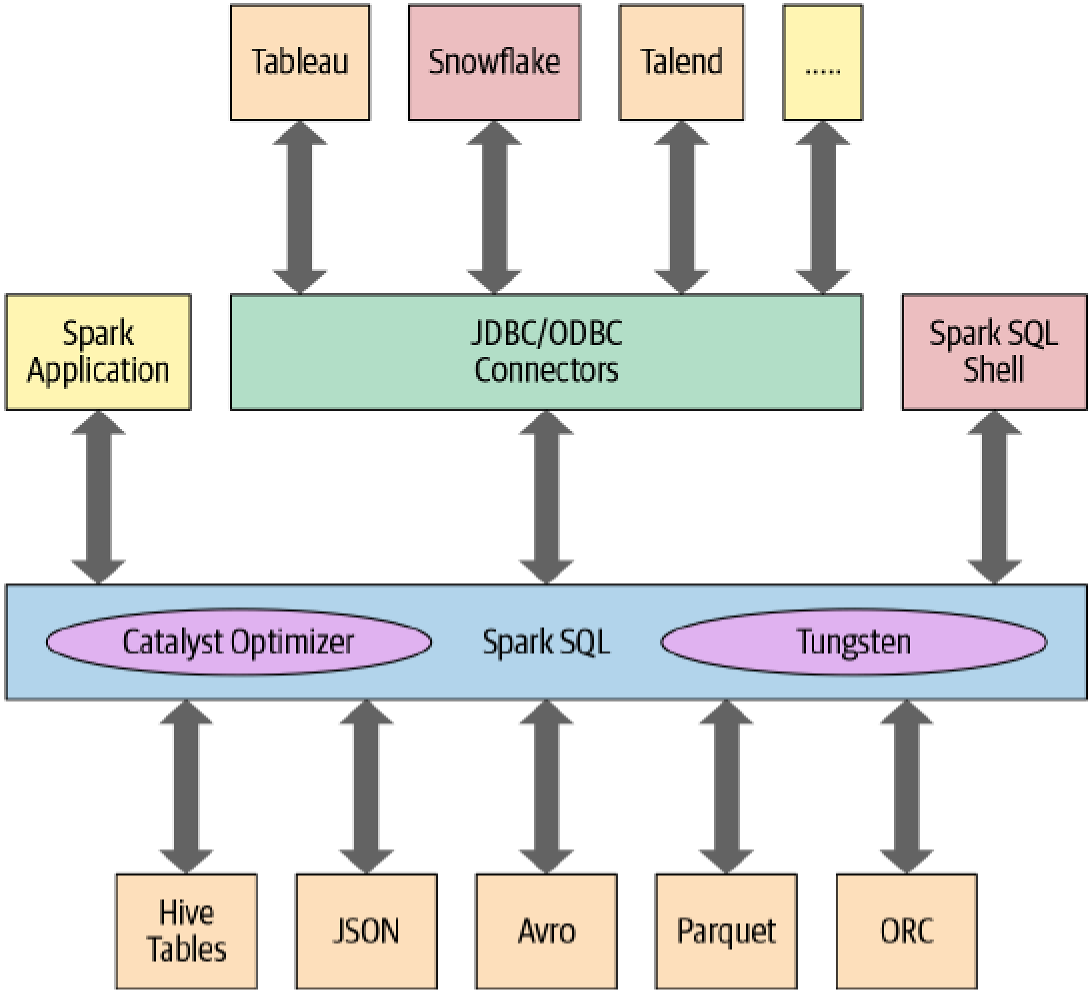
```
# In Python
# Join departure delays data (foo) with airport info
foo.join(
  airports,
  airports.IATA == foo.origin
).select("City", "State", "date", "delay", "distance", "destination").show()

-- In SQL
spark.sql("""
SELECT a.City, a.State, f.date, f.delay, f.distance, f.destination
  FROM foo f
  JOIN airports_na a
    ON a.IATA = f.origin
""").show()
```

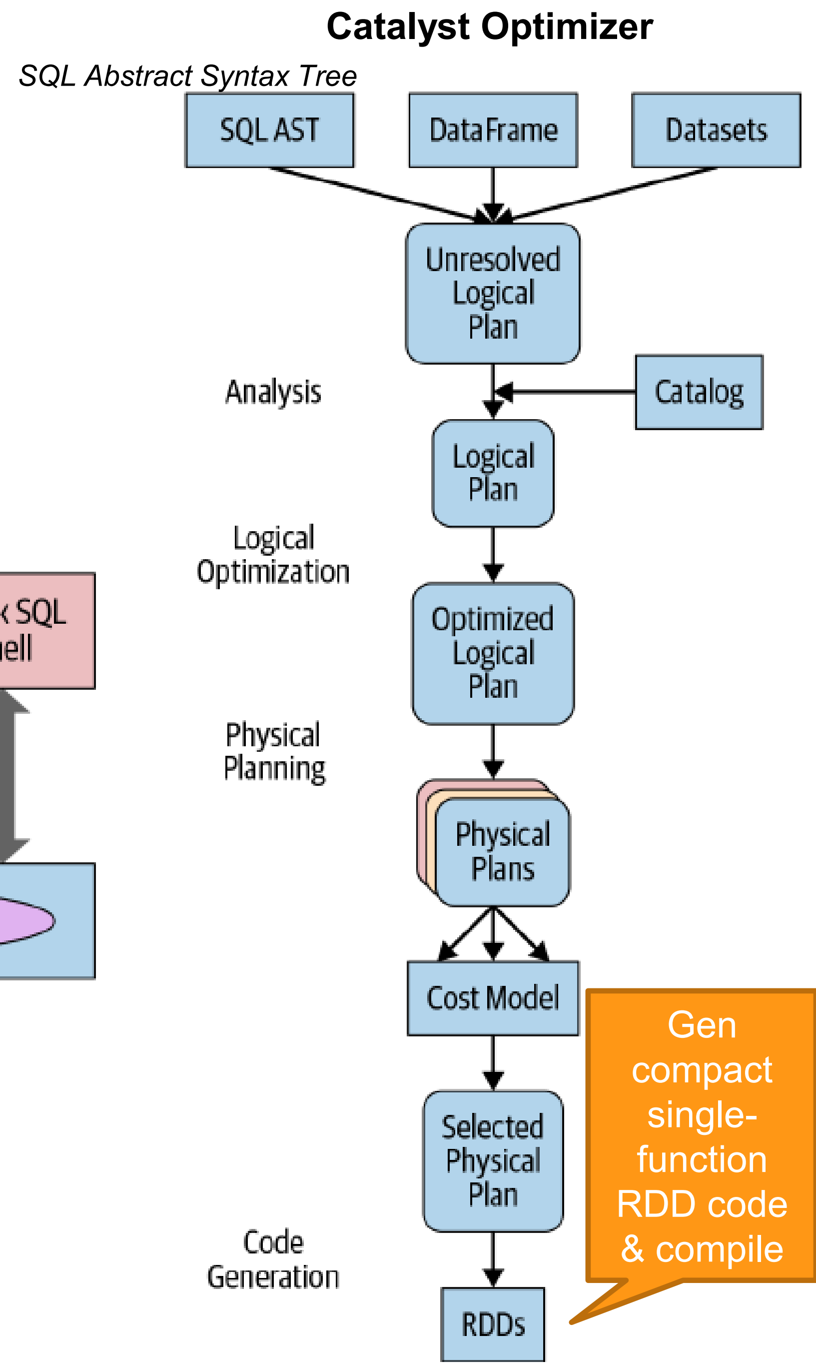
```
+-------+-----+--------+-----+--------+-----------+
|   City|State|    date|delay|distance|destination|
+-------+-----+--------+-----+--------+-----------+
|Seattle|   WA|01010710|   31|     590|        SFO|
|Seattle|   WA|01010955|  104|     590|        SFO|
|Seattle|   WA|01010730|    5|     590|        SFO|
+-------+-----+--------+-----+--------+-----------+
```

Execution Model

- Computation expressed in high-level DataFrame or SQL APIs is decomposed into low-level optimized and generated RDD operations
  - Then converted into Scala bytecode for the executors' JVMs
  - Generated RDD operation code is not accessible to users
  - RDD ops are NOT the same as the user-facing RDD APIs

SQL Stack & Query Plan

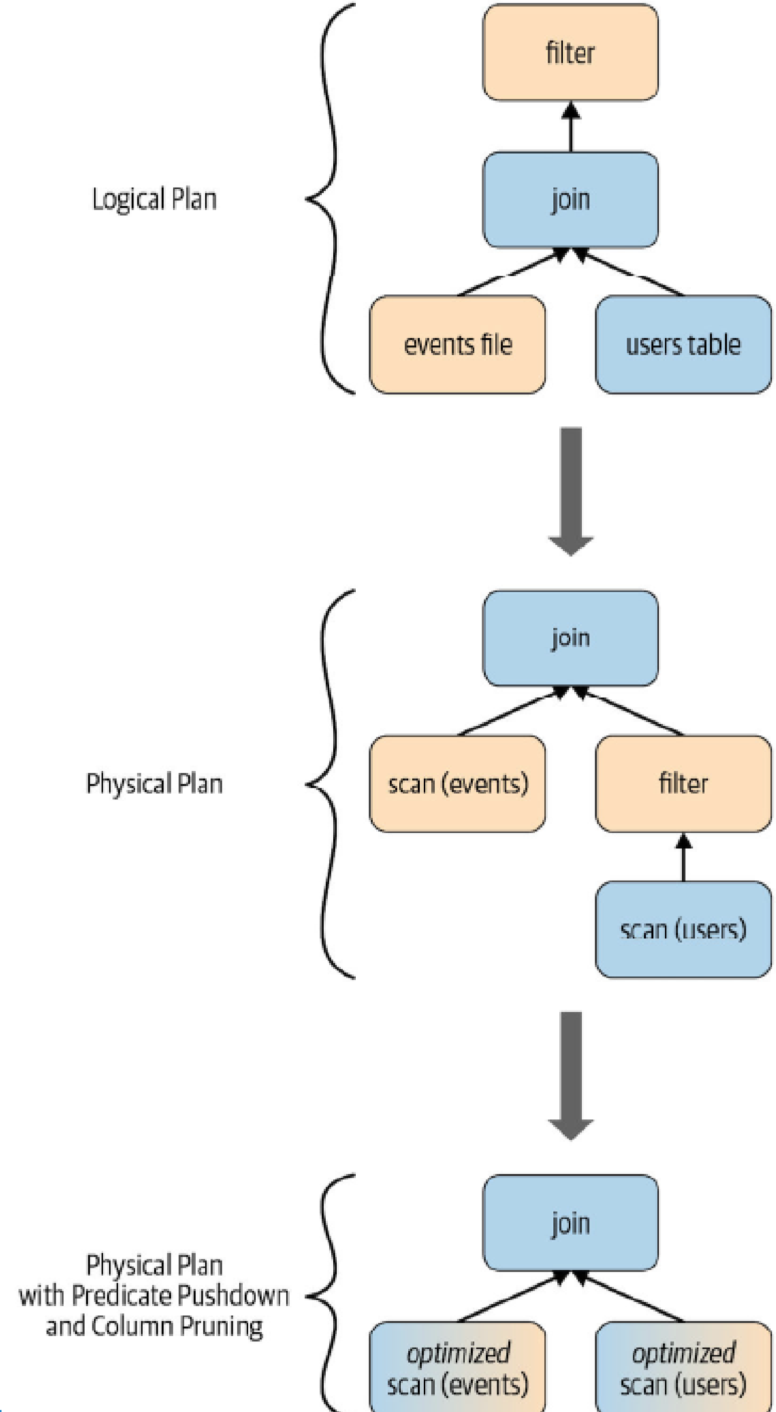




Query Planning

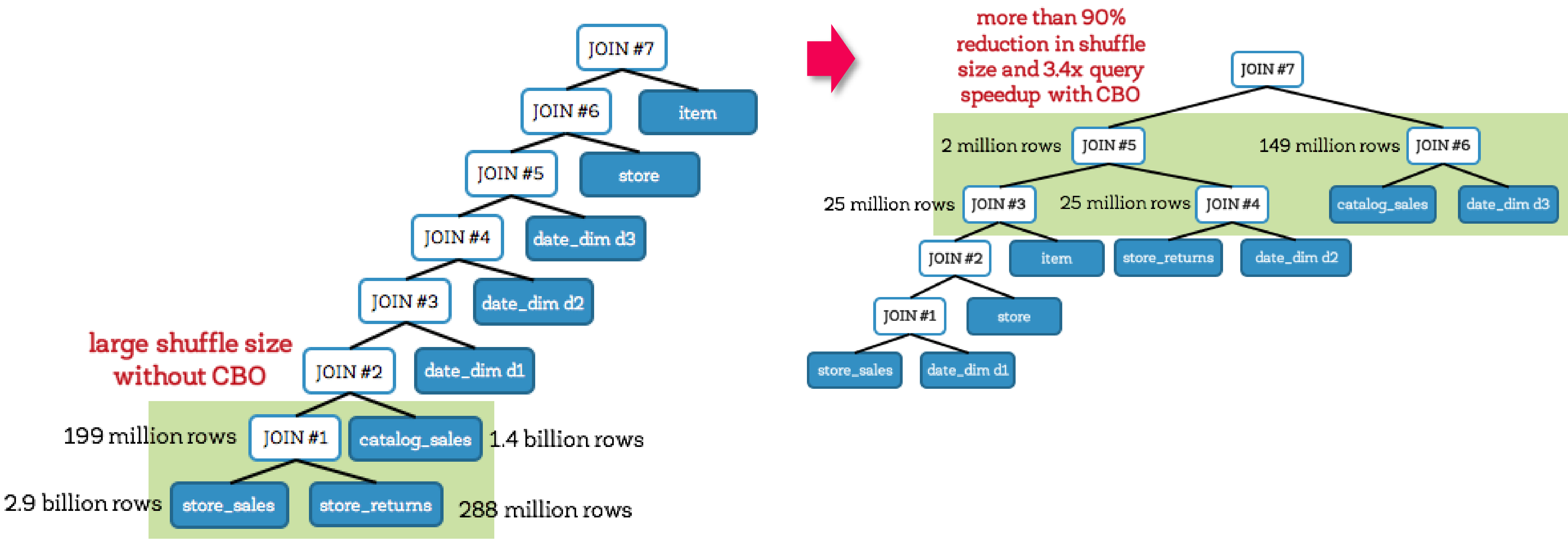
```
// Join two DataFrames
val joinedDF = users
  .join(events, users("id") === events("uid"))
```

.filter(~~events~~ users("date") > "2015-01-01")



https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html
Catalyst Query Optimizer, SIGMOD 2015,
http://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf

Query Planning: Cost Based Optimizer



Catalyst Optimizer for Spark DF/SQL Execution

Spark SQL: Relational Data Processing in Spark, Michael Armbrust, et al., *SIGMOD* 2015.

(Advanced Topic)

Catalyst Optimizer

- DataFrames keep track of their schema
- DataFrames support various relational operations
  - DataFrame represents a *logical plan* to compute a dataset
  - No execution occurs (materialization of dataset) until an "action" output operation is called, such as save
- Enables rich optimization across all operations that were used to build the DataFrame
- User operations captured using abstract syntax tree rather than opaque Python/Scala functions

```
ctx = new HiveContext()
users = ctx.table("users")
young = users.where(users("age") < 21)
println(young.count())
```

Expression parsed by Spark DF

Triggers Physical Plan

DataFrame Domain Specific Language (DSL)

- **Relational Operators:** Projection (select), filter (where), join & aggregations (groupBy)
- Operators all take **expression** objects

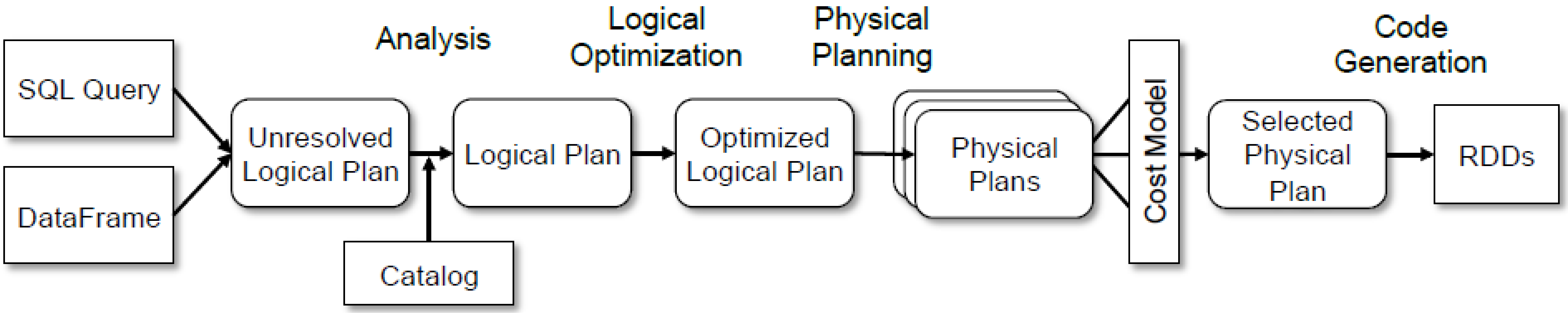
```
employees
  .join(dept, employees("deptId") === dept("id"))
  .where(employees("gender") === "female")
  .groupBy(dept("id"), dept("name"))
  .agg(count("name"))
```

- Similar to SQL, but lets uses to split into logical steps, separate DFs, control flows, UDFs
  - Schema evaluation is eager to catch type errors
  - Execution is lazy when an action is seen

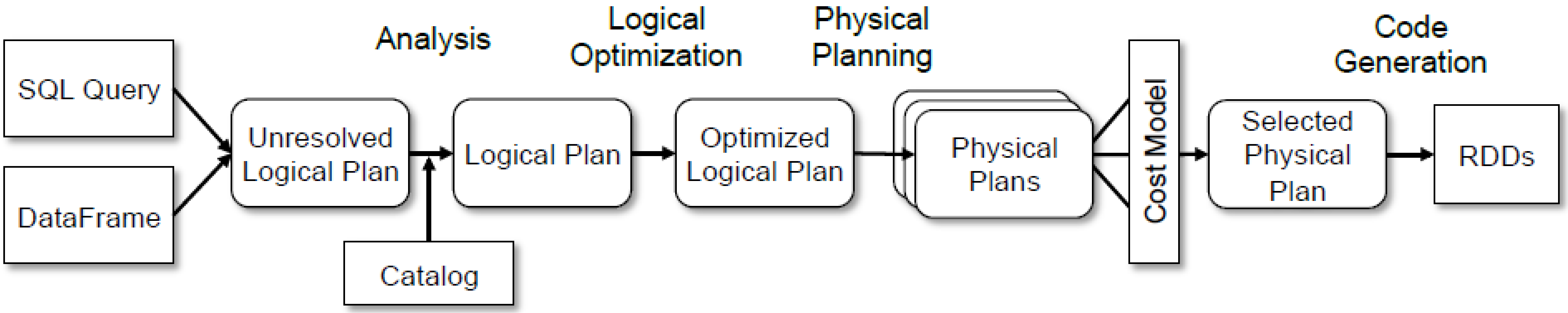
Catalyst Query Optimizer

- Modular and easy to add new optimization techniques and features to Spark SQL
- Enable external developers to extend the optimizer

- Cost-based Optimizer
  - Represent queries as **trees**
  - Applying **rules** to manipulate them
  - Generate different **plans** based on manipulation
  - Estimate the **execution cost** of each plan
  - Select the cheapest plan for actual execution



- Analyzing a logical plan to resolve references
  - AST from SQL or DF operations
  - Use *catalog* and inferencing to resolve attribute types, e.g., "SELECT (col+1) FROM sales"
  - Schema inference for semi-structured data



- Logical plan optimization
  - **Rule-based optimizations**: Constant folding, Predicate pushdown, Projection pruning, Null propagation, Boolean expression simplification
  - Trees and Rules for pattern matching/replacement
  - Recursively apply rules till tree is simplified/optimized

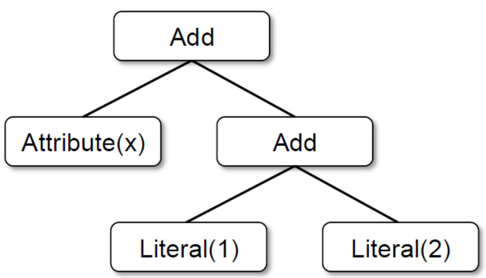
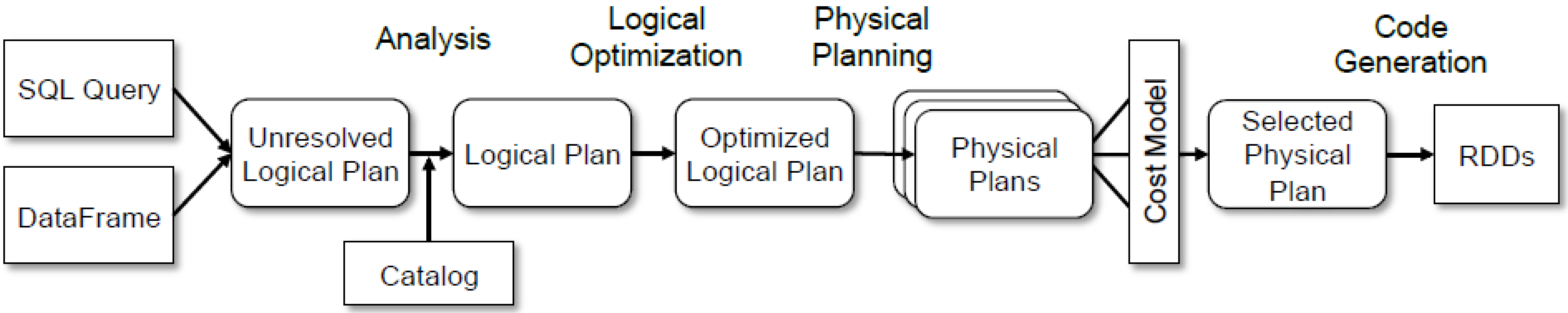


Figure 2: Catalyst tree for the expression x+(1+2).

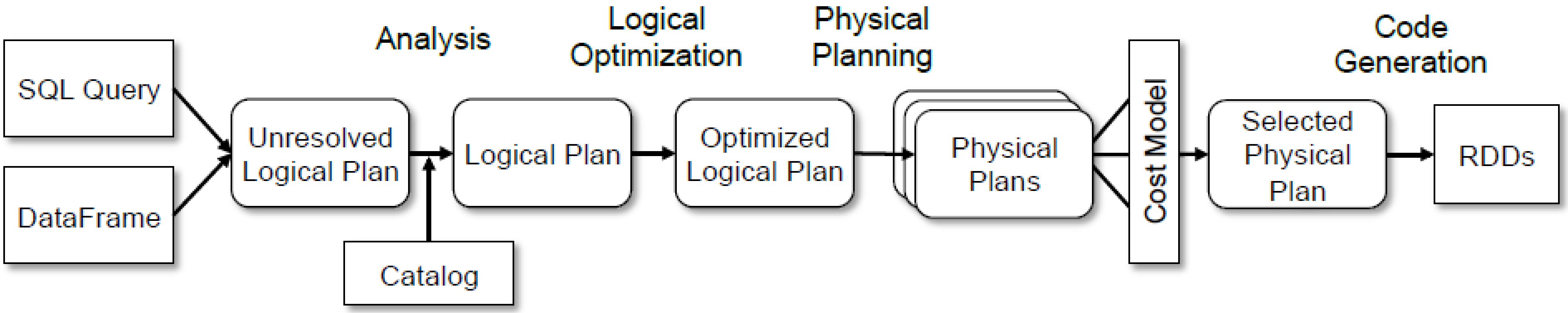
```
tree.transform {
  case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
  case Add(left, Literal(0)) => left
  case Add(Literal(0), right) => right
}
```

```
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)
    }
}
```

Float to Long for low precision math operations



- Physical planning
  - Takes a logical plan and generates one or more physical plans
  - Uses physical operators present in the Spark (RDD) execution engine
  - Rule-based optimizations: Pipelining projections or filters into one Spark RDD **map** operation
  - Cost based optimization: Estimates table sizes using in-memory cache, external file sizing, result of a subquery with a LIMIT



- Code generation to compile parts of the query to Java bytecode
  - Operates on in-memory datasets, so processing is *CPU-bound*
  - Code generation speeds up execution
    - Interpreting and evaluating each expression using if/else/switch is costly
    - AST Converted to Scala code, compiled into byte-code

“

- *Until here for Test #1 on 12 Feb on Module 1 and 2*

“

End of Lecture 11

Design Patterns
HeartBeat

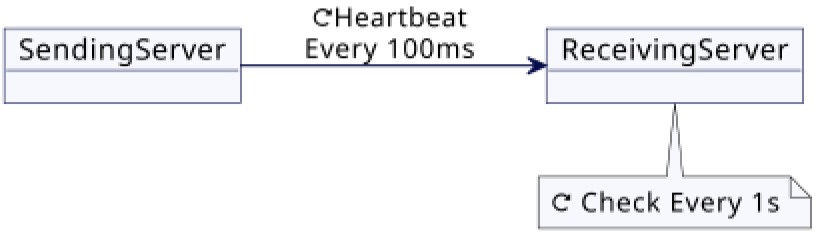


Figure 7.1 *Heartbeat*

- Timely detection of server failures
- timeout interval > request interval > network round trip time between the servers

From Patterns of Distributed Systems, Unmesh Joshi, Martin Fowler, 2023

A Consistent Core Can Manage the Membership of a Data Cluster

▷ HDFS Name Node, later Kafka

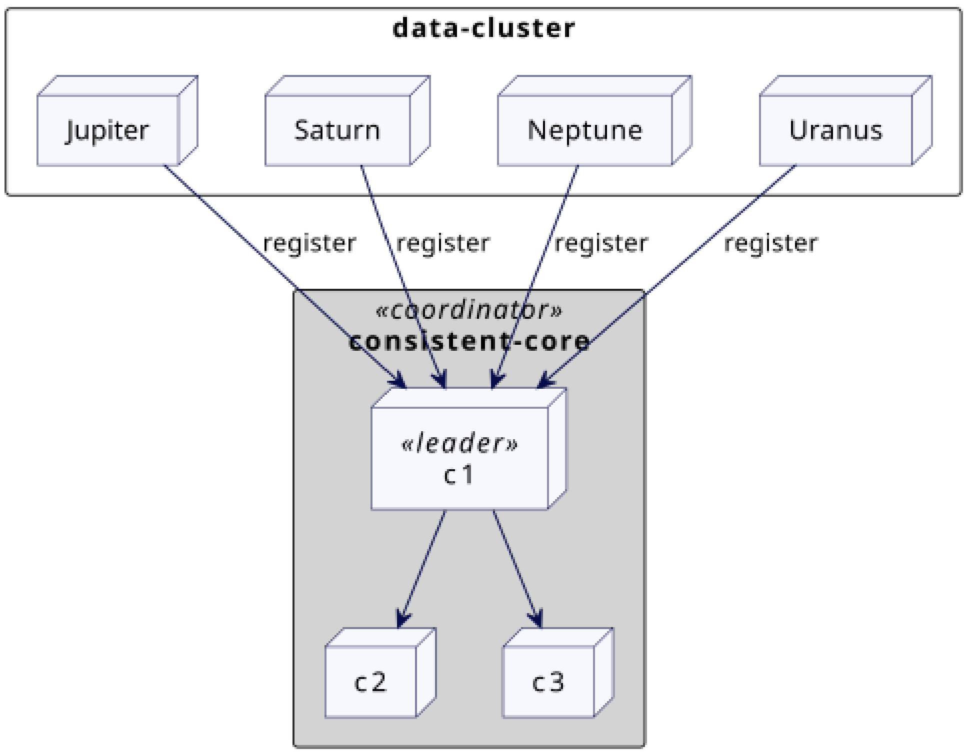


Figure 2.53 *Consistent Core tracks cluster membership.*

Design Pattens
Leasing

- A node can ask for a lease for a limited period of time, after which it expires.
- The node can renew the lease before it expires if it wants to extend the access.
- Implement the lease mechanism with *Consistent Core* to provide fault tolerance and consistency.
- Have a time-to-live value associated with the lease.
- Leader node tracks the lease timeouts, using its own monotonic clock…*why not wallclock?*
- *HDFS Name node and leases to Block leaders*

Design Patterns
Keeping Data Resilient on a Single Server

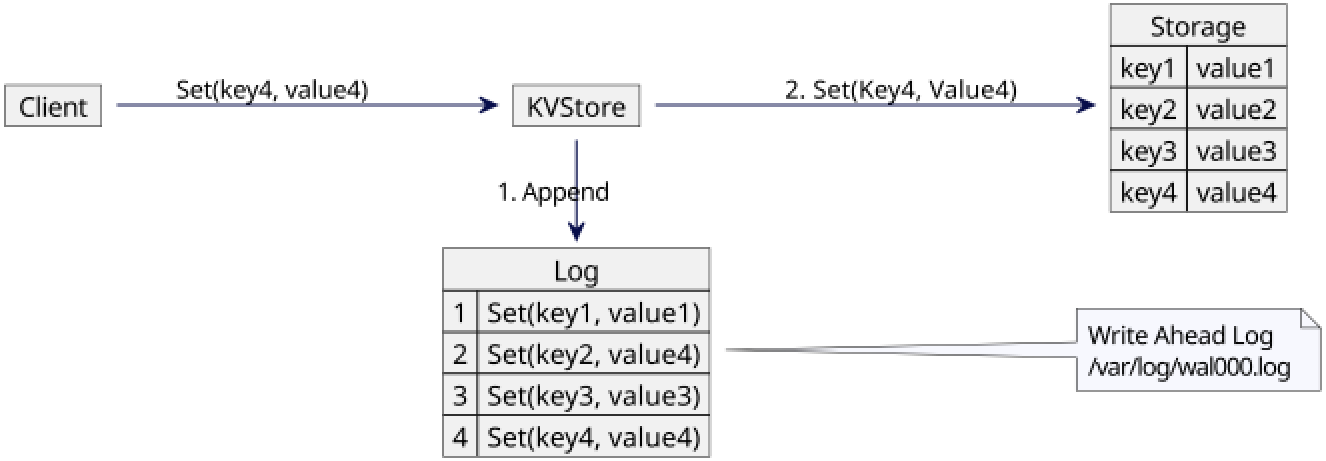


Figure 3.1 *Write-Ahead Log*

A variation used in Name Node of HDFS

Design Patterns
Replicating Client Requests

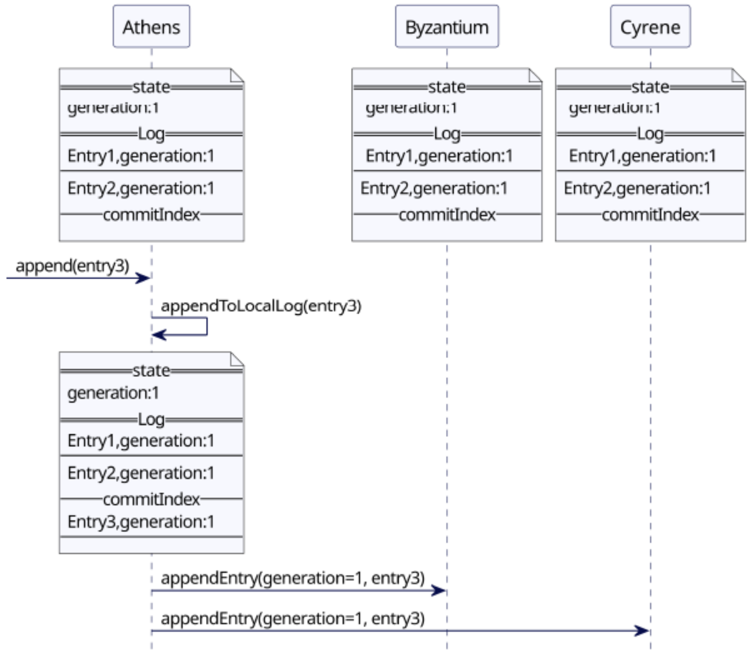


Figure 12.1 *Leader appends to its own log.*

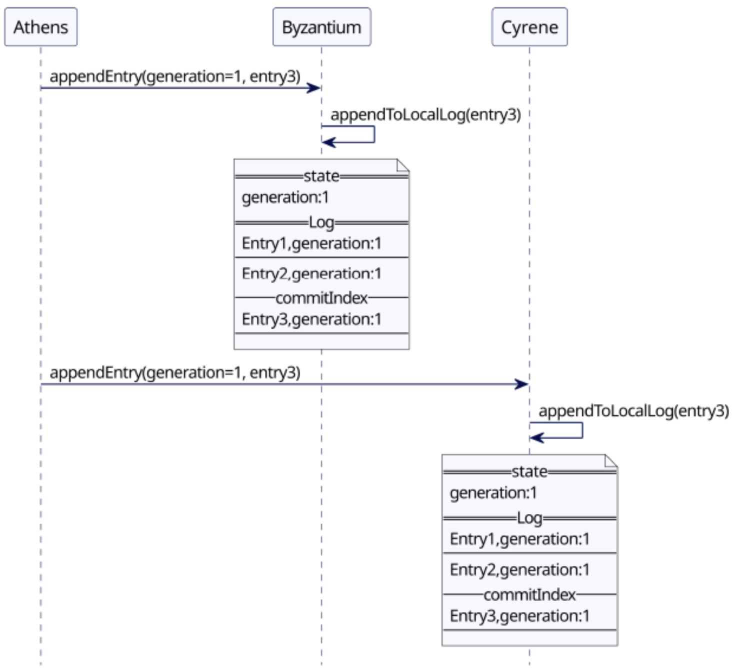


Figure 12.2 *Followers append to their logs.*

A Large Amount of Data Can Be Partitioned over Multiple Nodes

- Data should be evenly distributed across all the cluster nodes.
  - Fixed block size (HDFS) vs Variable block size (RDD)
- It should be possible to know which cluster node stores a particular data record, without making a request to all the nodes.
  - Costly Lookups (HDFS) vs Static Hashing (RDD shuffle)
- It should be quick and easy to move part of the data to the new nodes.
  - Logical (Dyanamo) vs Physical Partitioning (HDFS)

Design Patterns

Partitions Can Be Replicated for Resilience

- All Replication (HDFS Data Node) vs. Majority Quorum Replication
  - *Leader election*

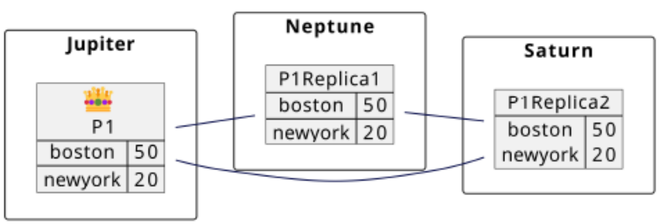


Figure 2.40 *Partitions are replicated for resilience.*

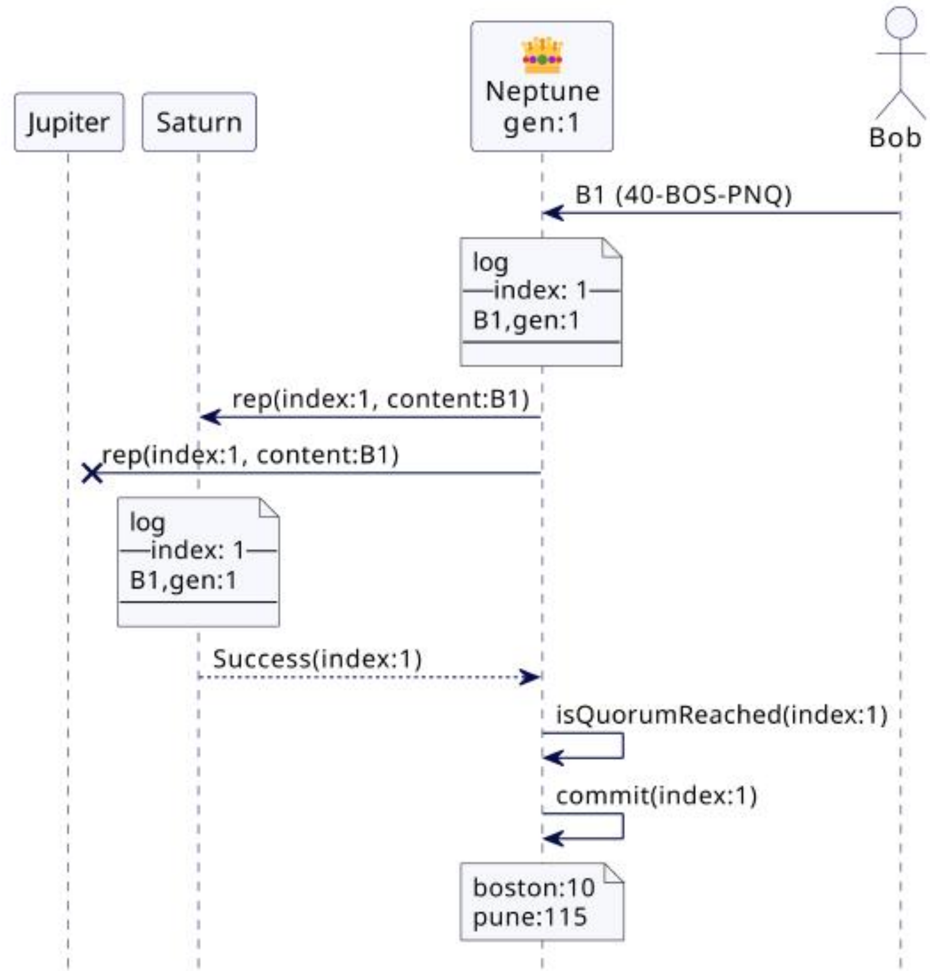


Figure 2.20 *Log entries are committed once they are accepted by a Majority Quorum.*

Design Patterns
Versioning

- HDFS uses block versioning
  - Prevent DataNodes from serving outdated or corrupted block replicas
- <BlockID, GenerationStamp> on NameNode
  - GenerationStamp (GS): Monotonically increasing version number assigned by NameNode upon Write or Append
  - Higher GS → newer version
- Any DataNode replica with a lower GS is stale
  - During the write, replicas must all have the same GS

Design Patterns

Request Pipeline

- Send next request without waiting for previous request to be acked.
- Reduces waiting time latency.
- Queue to buffer pending requests.
- *Data node write pipeline in HDFS...*

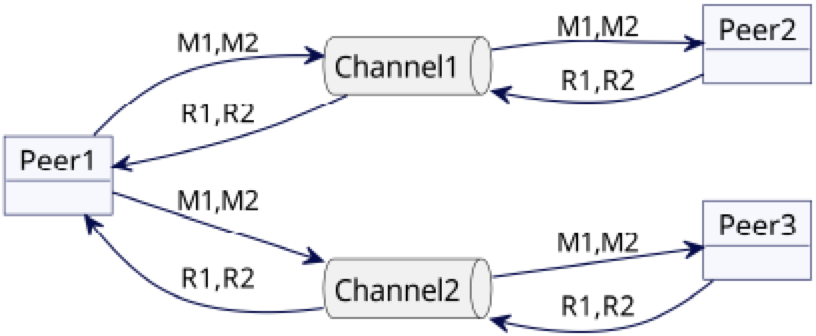


Figure 32.1 *Request pipeline*

Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics

Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao Chen Wang, Berthold Reinwald, and Fatma Ozcan

VLDB 2015

- Hadoop version 2.4.0
- Spark version 1.3.0

Architectural Diffs

- Evaluation of *batch* and *iterative* jobs
- Shuffle stage
  - MR: Shuffle+Sort, combiner/aggregation, external merge
  - Spark: Hash shuffle
- Execution
  - Task parallelism, Overlapping computation, Data Pipelining among stages
- Caching
  - Reuse of intermediate data. OS, HDFS, Tachyon, RDD

Workload

Ratio of Map output size to Job input size

Ratio of Reduce output size to Job input size

Table 1: Characteristics of Selected Workloads

		Word Count	Sort	K-Means (LR)	Page-Rank
Type	One Pass	√	√		
	Iterative			√	√
Shuffle Sel.	High		√		
	Medium				√
	Low	√		√	
Job/Iter. Sel.	High		√		
	Medium				√
	Low	√		√	

Workload

Incl. Combiner

Merge sort for large shuffle data

Compute+ NW xfr

Reuse w/o HDFS write

Table 2: Key Architectural Components of Interest

		Word Count	Sort	K-Means (LR)	Page-Rank
Shuffle	Aggregation	√		√	√
	External sort		√		
	Data transfer		√		√
Execution	Task parallelism	√	√	√	√
	Stage overlap		√		
	Data pipelining				√
Caching	Input			√	√
	Intermediate data				√

Setup

- 4 servers @ 32 CPU cores, 2.9GHz,
  - 9 disk drives at 7.2k RPM with 1 TB each
  - Hard disks deliver an aggregate bandwidth of about 125 GB/sec for reads and 45 GB/sec for writes
  - 190GB RAM
- 1 Gbps Ethernet switch
- We use Hadoop version 2.4.0 to run MapReduce on YARN
  - 32 containers per node
- We use Spark version 1.3.0 running in the standalone mode on HDFS 2.4.0
  - 8 Spark workers per node with 4 threads each

Summary: Diffs

- Performance differences due to components architectures in the two frameworks.
- Spark is about 2.5x, 5x, and 5x faster than MapReduce, for Word Count, k-means, and PageRank
  - + efficiency of the **hash-based aggregation** component for **combine** (Map-side reduction), 40% improvement
  - + **reduced CPU and disk overheads** due to RDD caching in Spark (PR, k-Means), 90%
  - + Data pipelining, **avoid materialization**
  - + Task loading , **context switch** is 10x faster

Summary: Diffs

- MapReduce is 2x faster than Spark for **Sort** workload
  - + MapReduce is **more efficient for shuffling data** than Spark, overlap Shuffle with Map, hide network overhead
  - Map stage in Spark is slower with more Reducers due to more open files
  - Increasing JVM for Spark has GC overheads

Summary: Common

- For one pass jobs, **Map is CPU bound, Reduce is network bound**
  - So **disk I/O not a bottleneck** (NW is), so spills often do not have a lot of penalty
- Input **parsing** is often an overhead
  - RDD caching helps. OS/HDFS caching does not.
- **GC overhead** bottleneck if heap size per task drops to 64MB with 128MB split
- Disk caching is bottleneck for RDD if CPU and disk I/O capacities unbalanced