Big Data Summer School

Spark SQL

Challenges and Solutions

Challenges

- Perform ETL to and from various (semi- or unstructured) data sources
- Perform advanced analytics (e.g. machine learning, graph processing) that are hard to express in relational systems.

Solutions

- A *DataFrame* API that can perform relational operations on both external data sources and Spark's built-in RDDs.
- A highly extensible optimizer, *Catalyst*, that uses features of Scala to add compostable rule, control code gen., and define extensions.

What is Apache Spark?

- Fast and general cluster computing system, interoperable with Hadoop

- Improves efficiency through: → **Up to 100× faster (2-10× on disk)**
  - In-memory computing primitives
  - General computation graphs
- Improves usability through: → **2-5× less code**
  - Rich APIs in Scala, Java, Python
  - Interactive shell

Spark Model

- *Write programs in terms of transformations on distributed datasets*
- Resilient Distributed Datasets (RDDs)
  - Collections of objects that can be stored in memory or disk across a cluster
  - Parallel functional transformations (map, filter, …)
  - Automatically rebuilt on failure

On-Disk Sort Record:

Time to sort 100TB

2013 Record: Hadoop

2100 machines

72 minutes

2014 Record: Spark

207 machines

23 minutes

Also sorted 1PB in 4 hours

Source: Daytona GraySort benchmark, sortbenchmark.org

Spark SQL

- Spark SQL
  - Part of the core distribution since Spark 1.0 (April 2014)
  - Runs SQL / HiveQL queries, optionally alongside or replacing existing Hive deployments

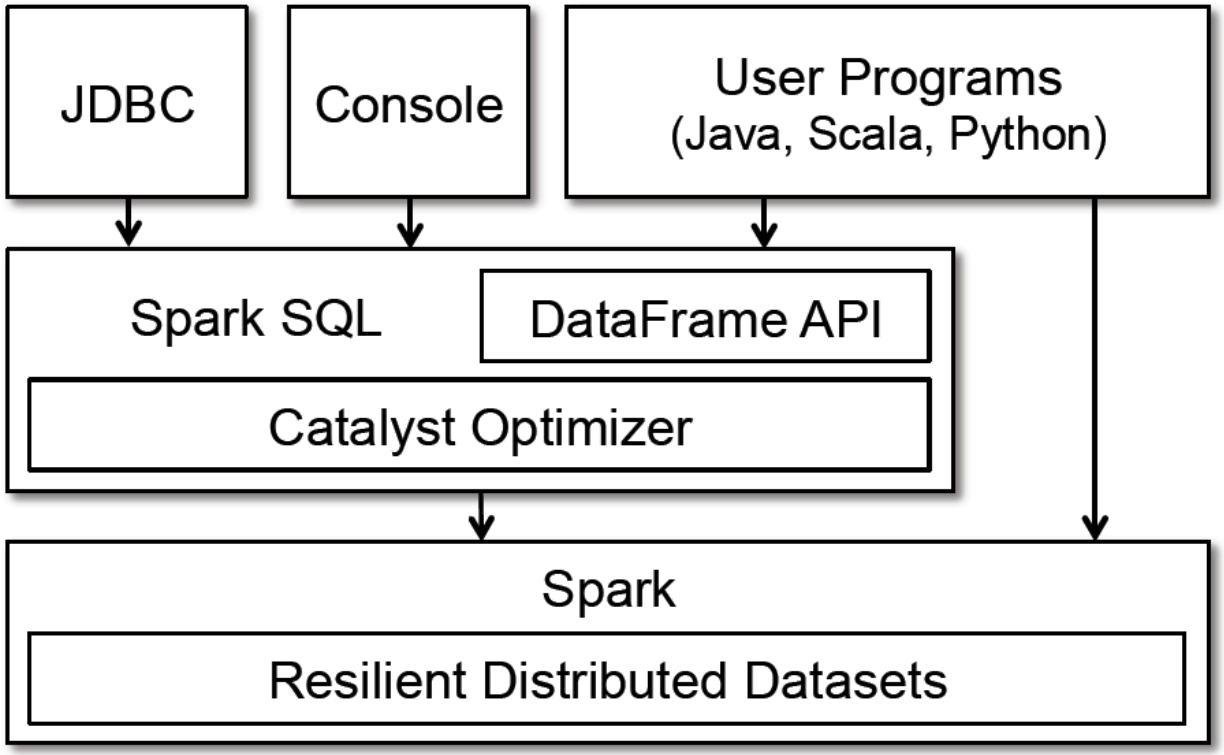

```
SELECT COUNT(*)
FROM hiveTable
WHERE hive_udf(data)
```

Improvement upon Existing Art

- Engine does not understand the structure of the data in RDDs or the semantics of user functions → limited optimization.

- Can only be used to query external data in Hive catalog → limited data sources
- Can only be invoked via SQL string from Spark→ error prone
- Hive optimizer tailored for MapReduce → difficult to extend

Programming Interface



- DataFrame

1. A distributed collection of rows organized into named columns.
2. An abstraction for selecting, filtering, aggregating and plotting structured data (*cf. R, Pandas*).
3. Archaic: Previously SchemaRDD (*cf. Spark < 1.3*).

DataFrame

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

- A distributed collection of rows with the same schema (RDDs suffer from type erasure)
- Can be constructed from external data sources or RDDs into essentially an RDD of Row objects (SchemaRDDs as of Spark < 1.3)
- Supports relational operators (e.g. *where*, *groupby*) as well as Spark operations.
- Evaluated lazily → unmaterialized *logical* plan

DataFrame Operations

- Relational operations (select, where, join, groupBy) Operators take *expression* objects
- Operators build up an abstract syntax tree (AST), which is then optimized by *Catalyst*.

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

- Alternatively, register as temp SQL table and perform traditional SQL query strings

```
users.where(users("age") < 21)
     .registerTempTable("young")
ctx.sql("SELECT count(*), avg(age) FROM young")
```

Querying Native Datasets

- Infer column names and types directly from data objects (via reflection in Java and Scala and data sampling in Python, which is dynamically typed)

```
case class User(name: String, age: Int)
```

- Native objects accessed in-place to avoid expensive data format transformation.
- Benefits:
  - Run relational operations on existing Spark programs.
  - Combine RDDs with external structured data

Columnar storage with *hot* columns cached in memory

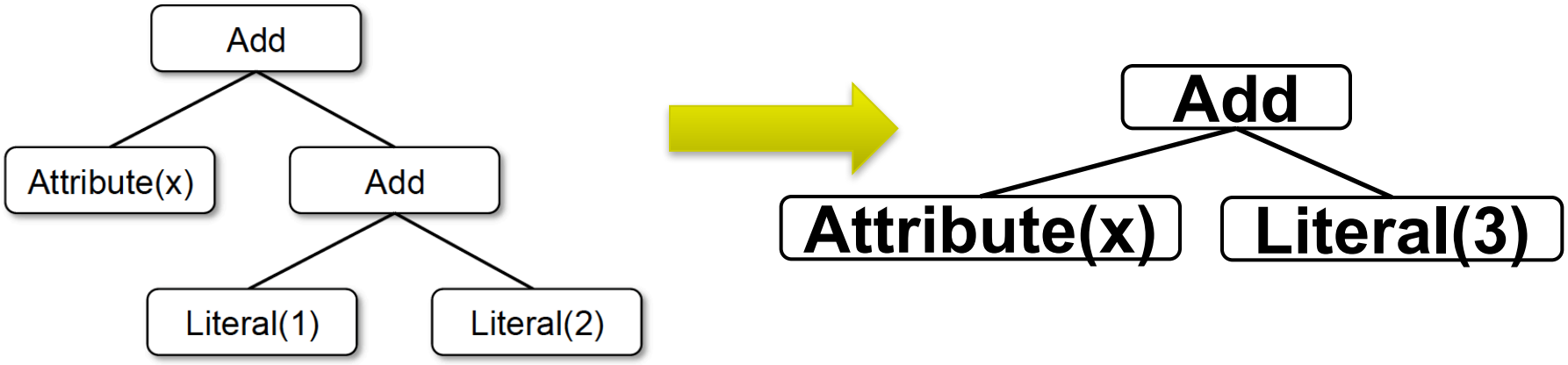
User-Defined Functions (UDFs)

- Easy extension of limited operations supported.
- Allows inline registration of UDFs
  - Compare with Pig, which requires the UDF to be written in a Java package that's loaded into the Pig script.
- Can be defined on simple data types or entire tables.
- UDFs available to other interfaces after registration

```
val model: LogisticRegressionModel = ...

ctx.udf.register("predict",
  (x: Float, y: Float) => model.predict(Vector(x, y)))

ctx.sql("SELECT predict(age, weight) FROM users")
```

Catalyst

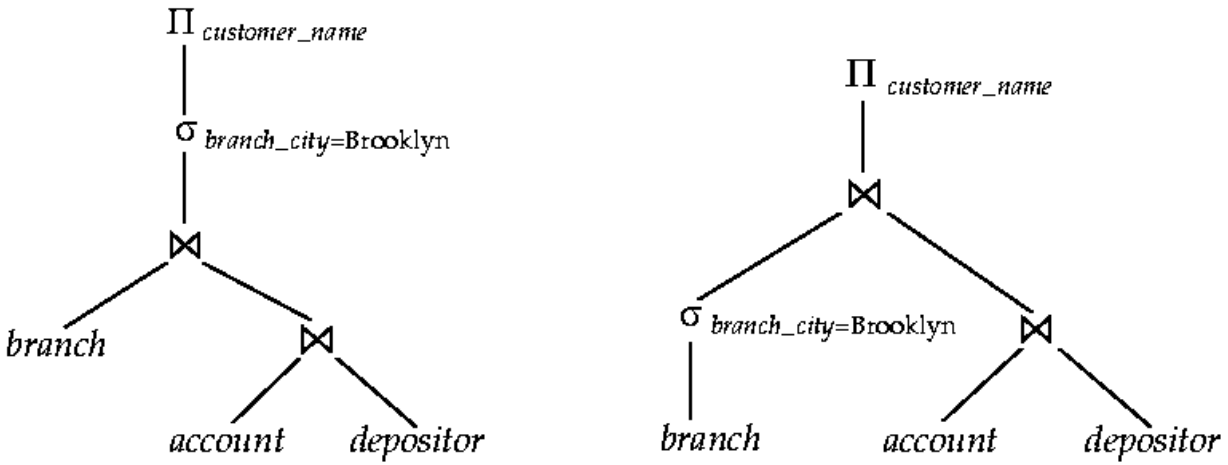
```
tree.transform {
  case Add(Literal(c1), Literal(c2)) => Literal(c1+c2)
}
```



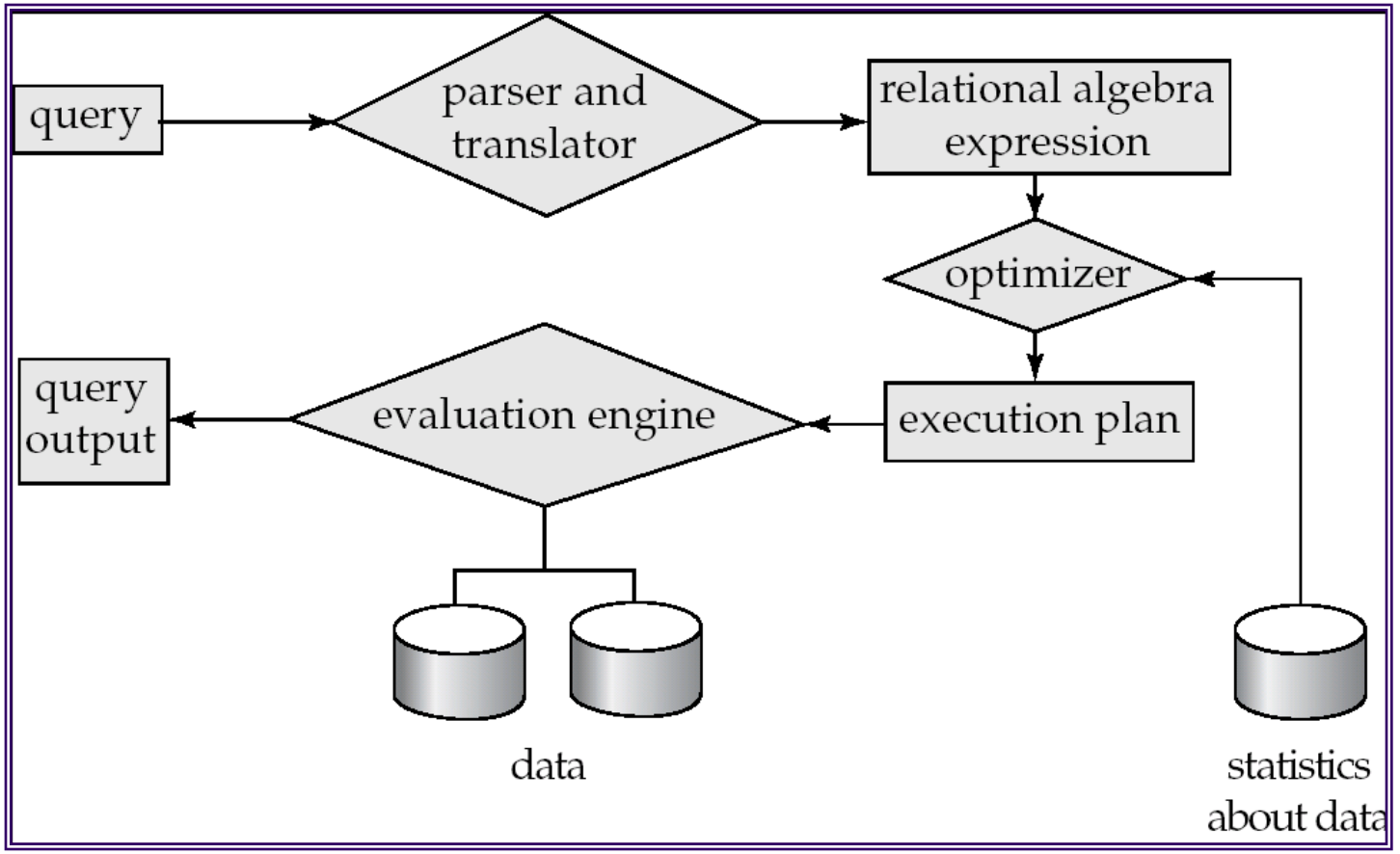
x + (1 + 2) **x + 3**

Prior Work: Optimizer Generators

- Cascades:
- Create a custom language for expressing rules that rewrite trees of relational operators.
- Build a compiler that generates executable code for these rules.



Query Processing

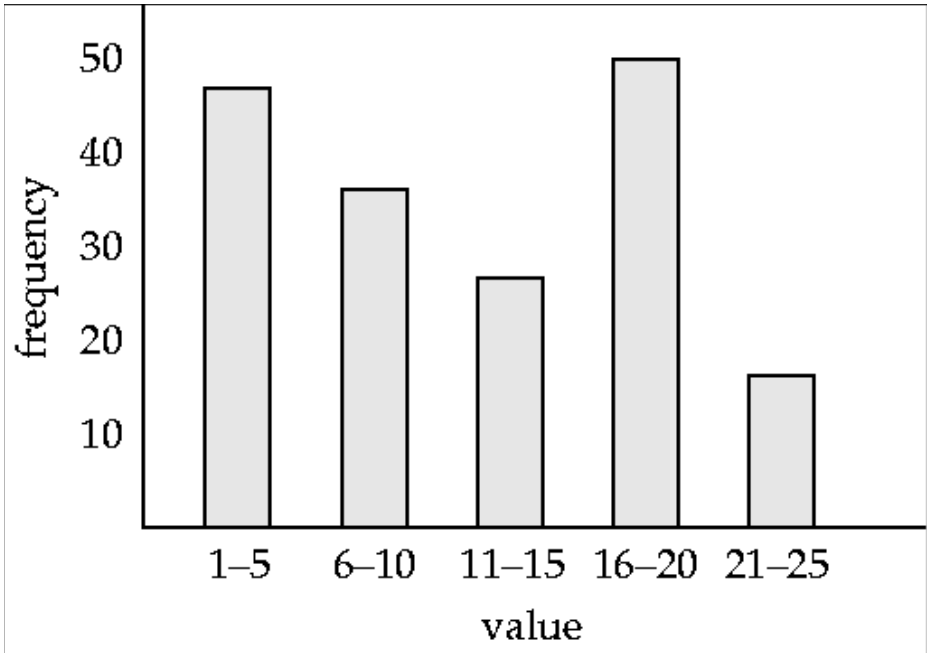


Query Optimization

- Selection
- Join
- Cost Estimation
- Join Order

Selection

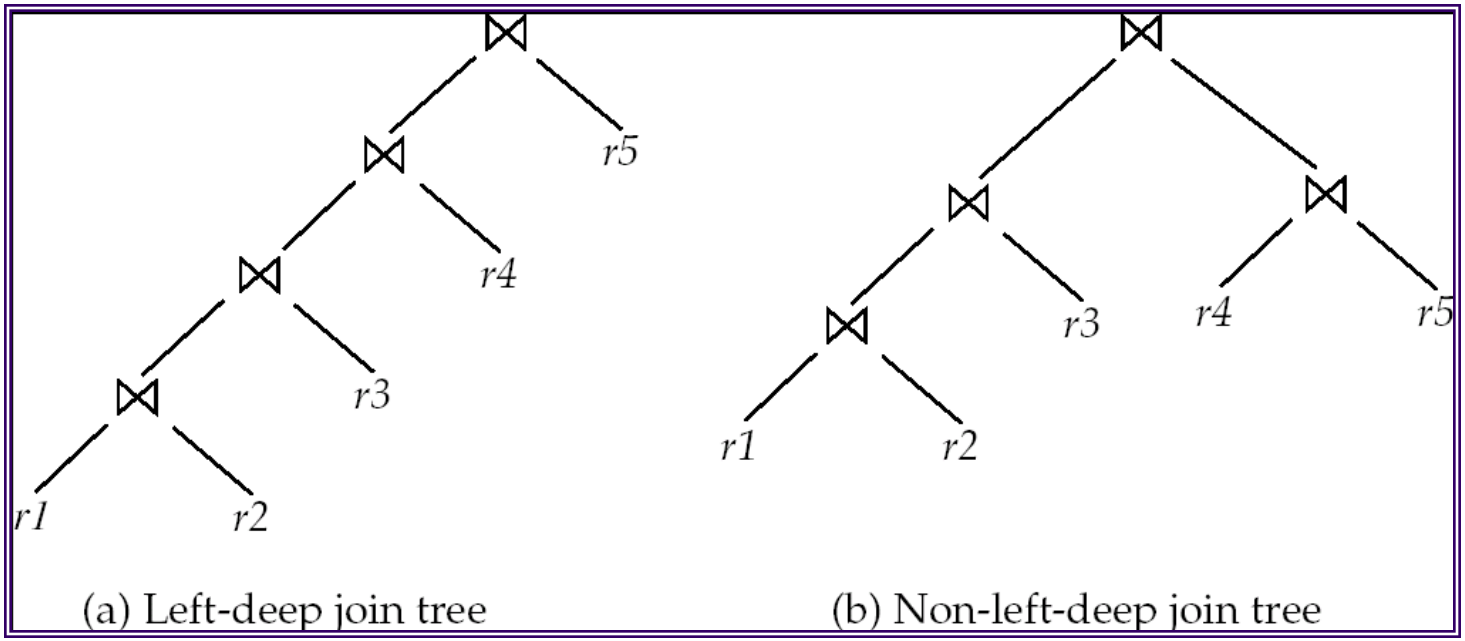
- B-tree Index
- Hash Index
- Histogram



Join

- Nested-loop join
- Block nested-loop join
- Indexed nested-loop join
- Merge-join
- Hash-join

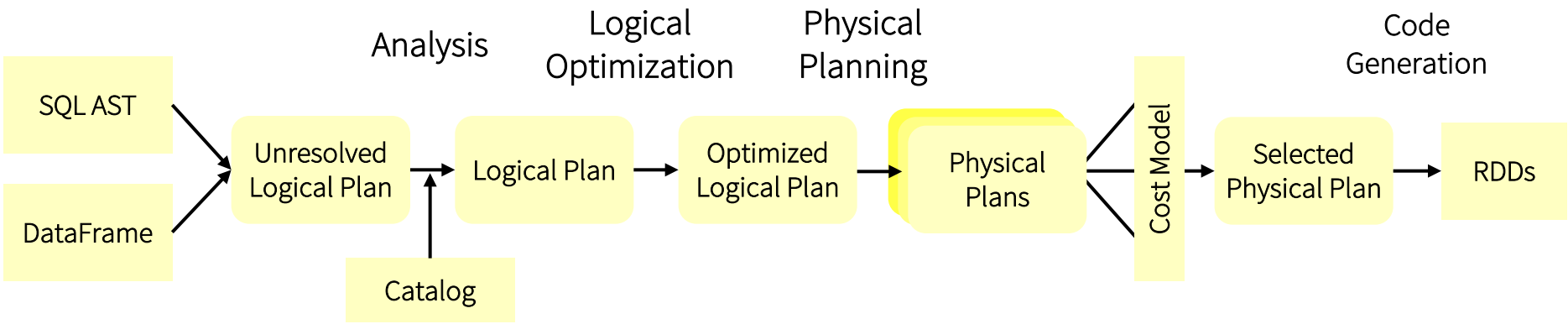
Join Order


r5
r4
r3
r1
r2
(a) Left-deep join tree
r3
r4
r5
r1
r2
(b) Non-left-deep join tree

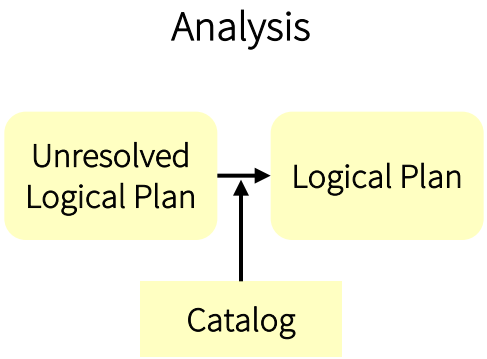
Catalyst Rules

- *Pattern matching* functions that transform subtrees into specific structures.
  - *Partial function*—skip over subtrees that do not match → no need to modify existing rules when adding new types of operators.
- Multiple patterns in the same *transform* call.
- May take multiple *batches* to reach a *fixed point*.
- *transform* can contain arbitrary Scala code.

Plan Optimization & Execution



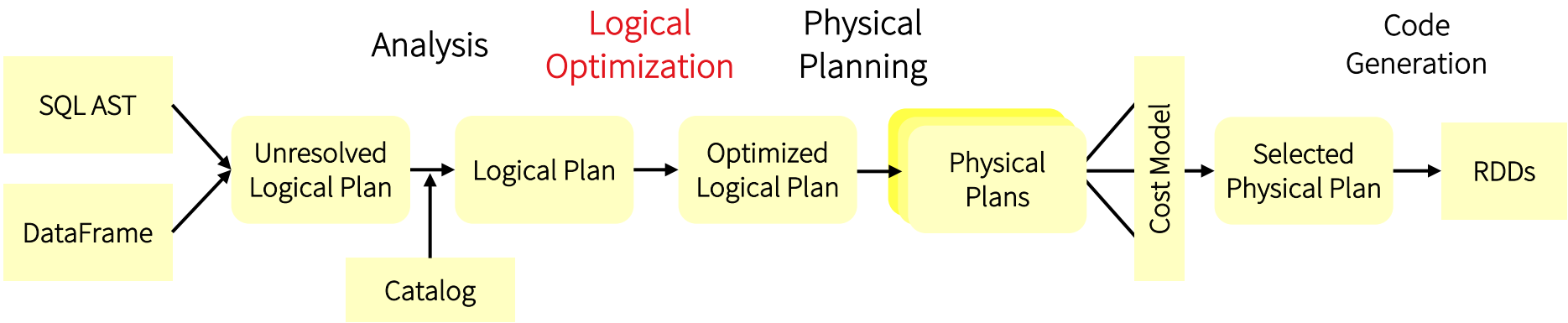
DataFrames and SQL share the same optimization/execution pipeline



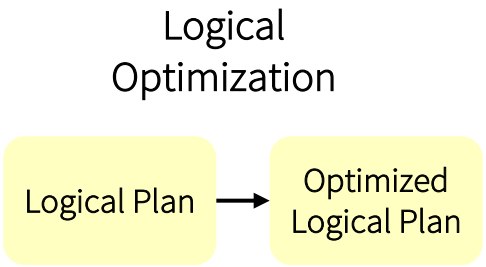
SELECT *col* FROM *sales*

- 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*)

Plan Optimization & Execution



DataFrames and SQL share the same optimization/execution pipeline

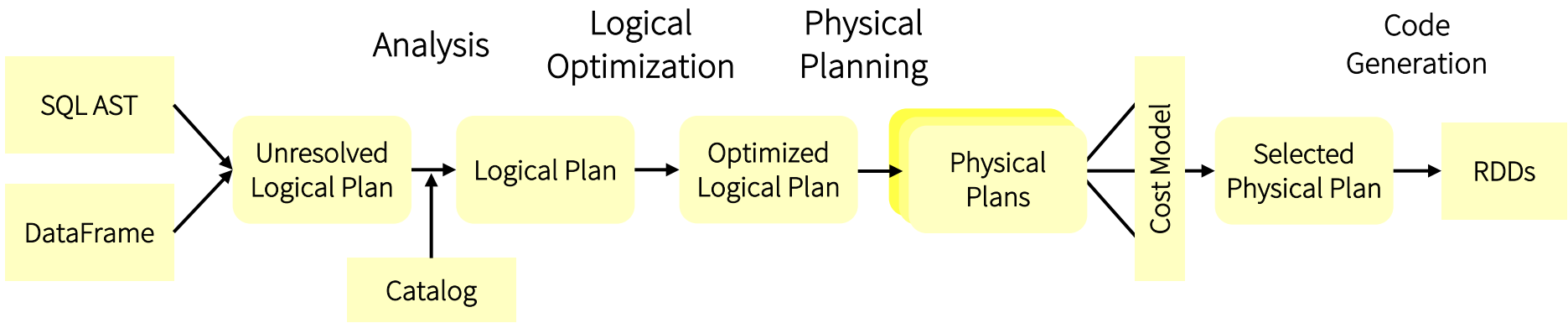


- Applies standard rule-based optimization (constant folding, predicate-pushdown, projection pruning, null propagation, boolean expression simplification, etc)
- 800LOC

```
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)
    }
}
```

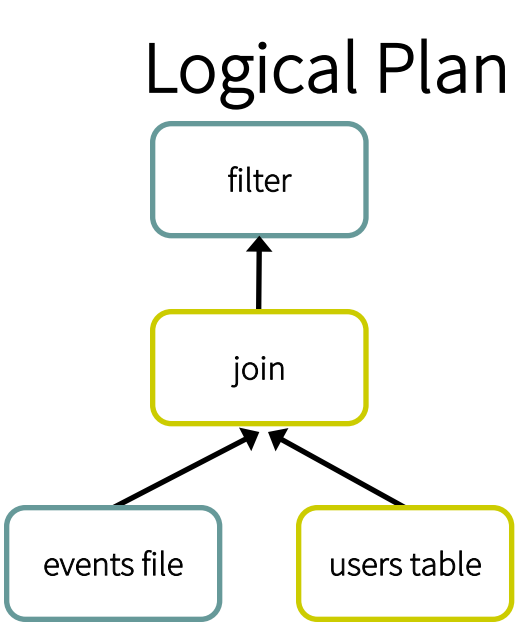
Plan Optimization & Execution

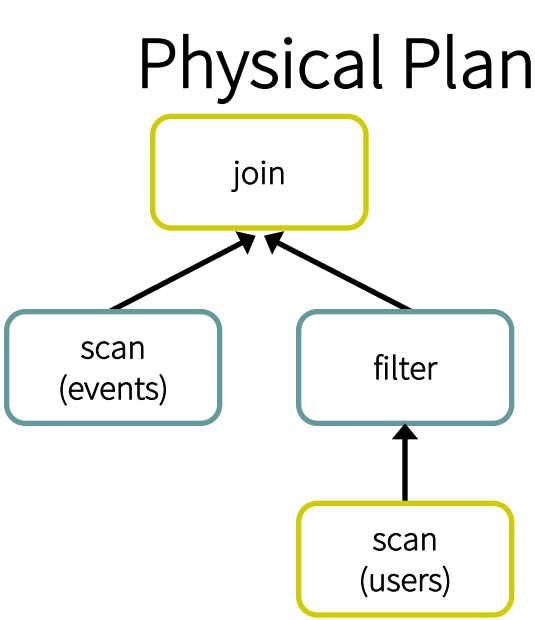
e.g. Pipeline projections and filters into a single *map*

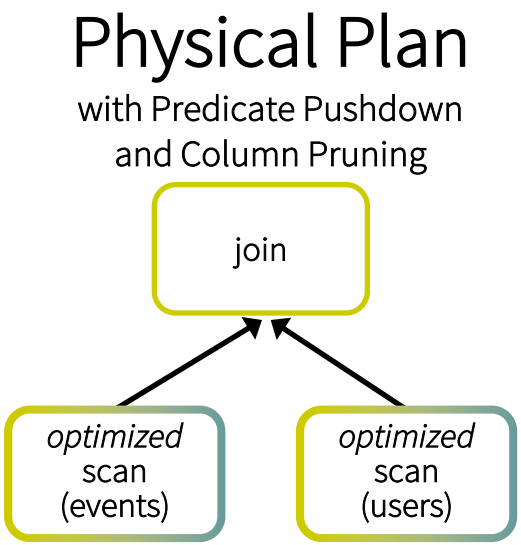


DataFrames and SQL share the same optimization/execution pipeline

```
def add_demographics(events):
    u = sqlCtx.table("users")                          # Load partitioned Hive table
    events \
      .join(u, events.user_id == u.user_id) \          # Join on user_id
      .withColumn("city", zipToCity(df.zip))           # Run udf to add city column

events = add_demographics(sqlCtx.load("/data/events", "parquet"))
training_data = events.where(events.city == "Melbourne").select(events.timestamp).collect()
```

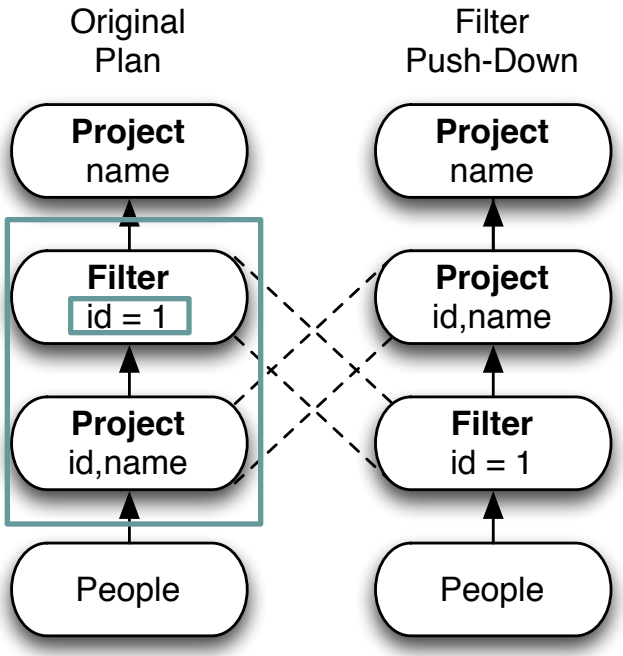




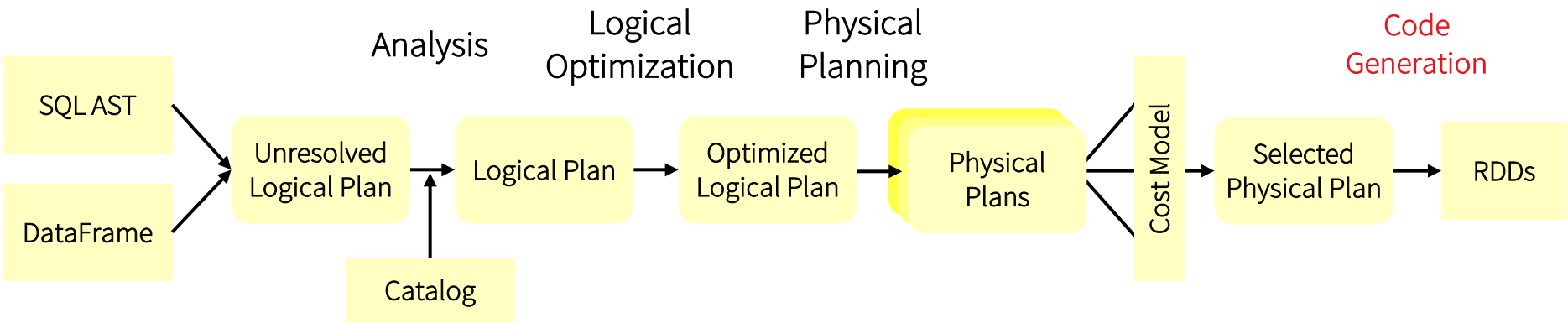


An Example Catalyst Transformation

1. Find filters on top of projections.
2. Check that the filter can be evaluated without the result of the project.
3. If so, switch the operators.



Plan Optimization & Execution



DataFrames and SQL share the same optimization/execution pipeline

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)}"
}
```

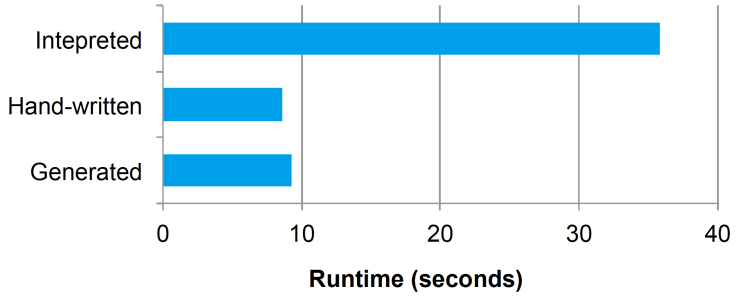


Figure 4: A comparision of the performance evaluating the expresion x+x+x, where x is an integer, 1 billion times.

- Relies on Scala's *quasiquotes* to simplify code gen.
- Catalyst transforms a SQL tree into an abstract syntax tree (AST) for Scala code to eval expr and generate code
- 700LOC

Extensions

- **Data Sources**
  - must implement a *createRelation* function that takes a set of key-value params and returns a *BaseRelation* object.
  - E.g. CSV, Avro, Parquet, JDBC
- **User-Defined Types (UDTs)**
  - 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))
}
```

Advanced Analytics Features

Schema Inference for Semistructured Data

- JSON
- Automatically infers schema from a set of records, in one pass or sample
- A tree of STRUCT types, each of which may contain atoms, arrays, or other STRUCTs.
- Find the most appropriate type for a field based on all data observed in that column. Determine array element types in the same way.
- Merge schemata of single records in one *reduce* operation.

```
{
  "text": "This is a tweet about #Spark",
  "tags": ["#Spark"],
  "loc": {"lat": 45.1, "long": 90}
}

{
  "text": "This is another tweet",
  "tags": [],
  "loc": {"lat": 39, "long": 88.5}
}

{
  "text": "A #tweet without #location",
  "tags": ["#tweet", "#location"]
}
```

```
text STRING NOT NULL,
tags ARRAY<STRING NOT NULL> NOT NULL,
loc STRUCT<lat FLOAT NOT NULL, long FLOAT NOT NULL>
```

Write Less Code: Compute an Average

- 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 Pig

```
P = load '/people' as (name, name);
G = group P by name;
R = foreach G generate … AVG(G.age);
```

Using DataFrames

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

Seamlessly Integrated: RDDs

- Internally, DataFrame execution is done with Spark RDDs making interoperation with outside sources and custom algorithms easy.

External Input

```
def buildScan(
    requiredColumns: Array[String],
    filters: Array[Filter]):
RDD[Row]
```

Custom Processing

```
queryResult.rdd.mapPartitions { iter =>

  … Your code here …

}
```

Extensible Input & Output

- Spark's Data Source API allows optimizations like column pruning and filter pushdown into custom data sources.

Built-In

External

Seamlessly Integrated

- Embedding in a full programming language makes UDFs trivial and allows composition using functions.

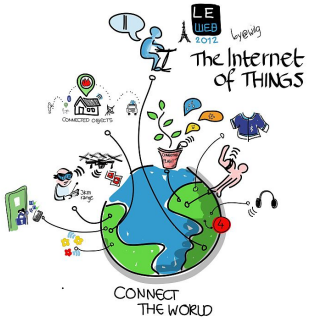
```
zipToCity = udf(lambda city: <custom logic
here>)

def add_demographics(events):
    u = sqlCtx.table("users")
    events \
      .join(u, events.user_id == u.user_id) \
      .withColumn("city", zipToCity(df.zip))
```

Takes and returns a DataFrame

Spatial data is ubiquitous!

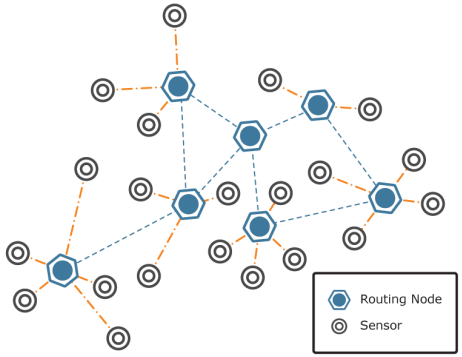
Location-based Services

Google Maps



UBER

yelp



Social Networks

twitter

facebook

Google+

Existing solutions

- Spatial Databases?
  - Oracle Spatial & Graph
  - PostGIS
  - ArchGIS
  - …
- Big Data Analysis Framework?
  - Apache Hadoop
  - Apache Spark
  - …
- Distributed Spatial Analysis Systems/Libraries?
  - SpatialHadoop
  - GeoSpark
  - GeoMesa
  - …

Spatial databases

- Spatial extension of traditional single-node RDBMS
- Support geographic objects allowing location queries to be run in SQL.

```
SELECT superhero.name
FROM city, superhero
WHERE ST_Contains(city.geom, superhero.geom)
AND city.name = 'Gotham';
```

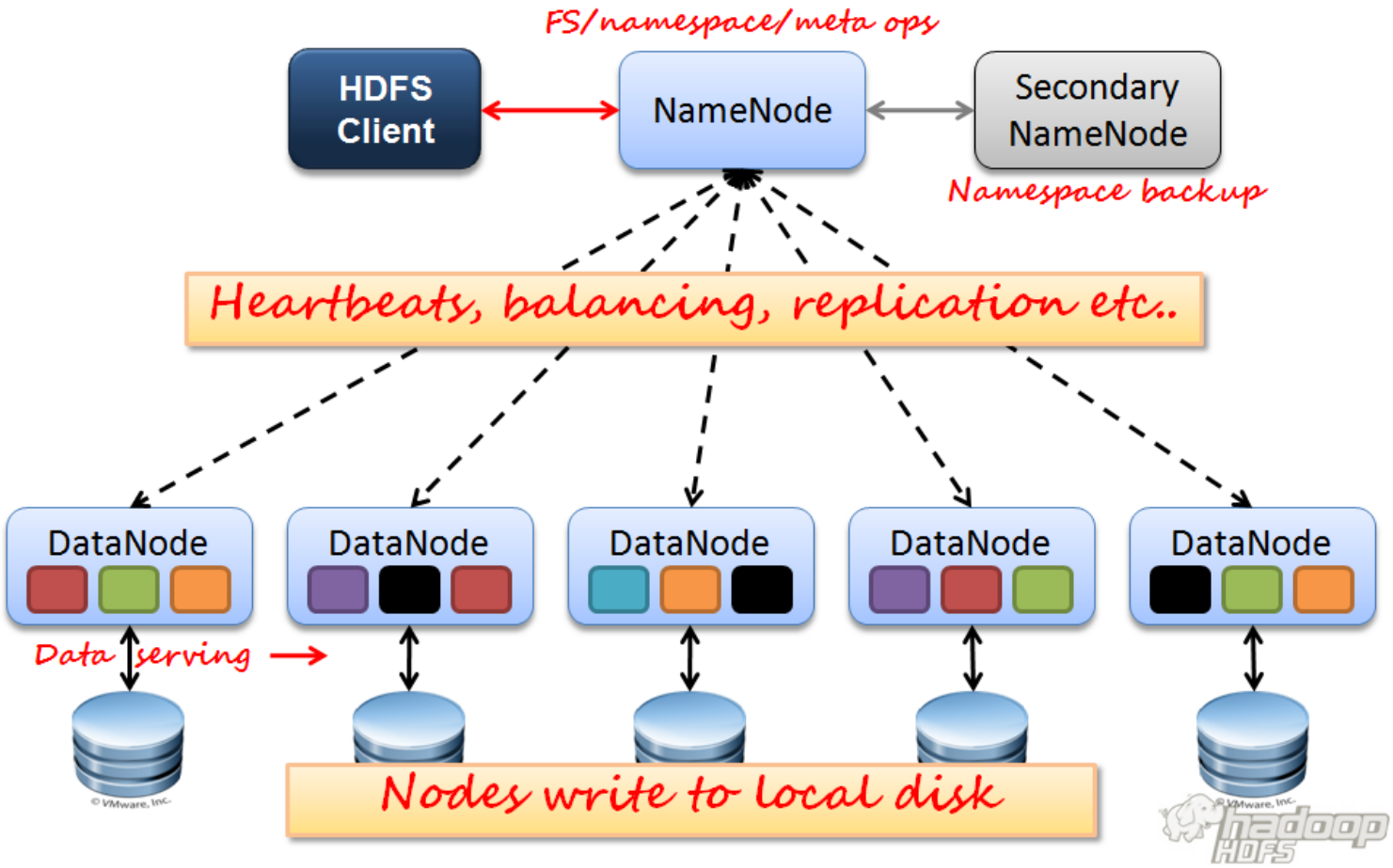
- Spatial Indexing for fast query processing.

Apache Hadoop

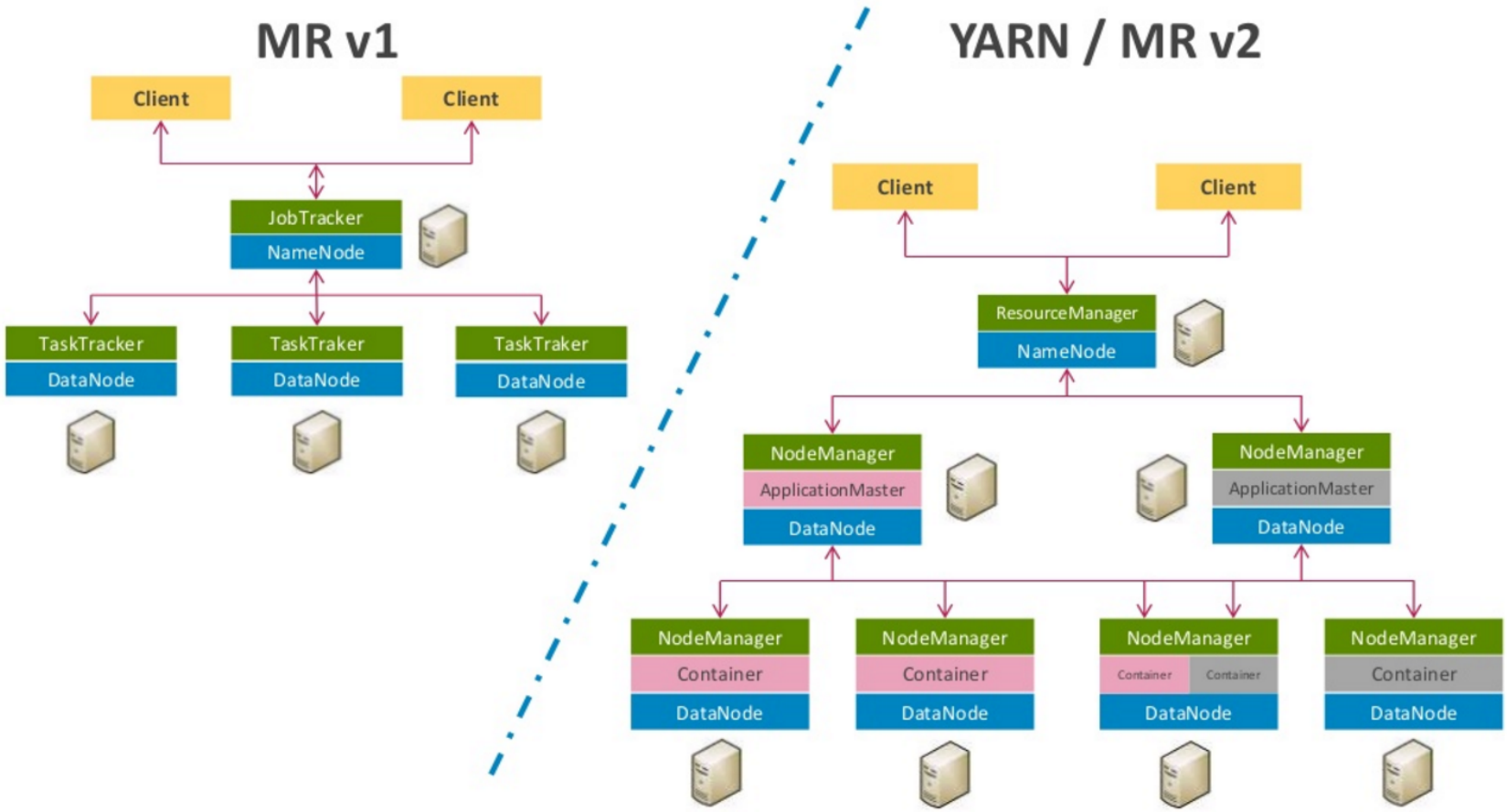
"Flexible and available architecture for large scale computation and data processing on a cluster of commodity hardware"

- **HDFS :** A distributed file system that provides high-throughput access to application data.
- **MapReduce :** A general computation framework for parallel processing of large data sets.
- **YARN** (since Hadoop 2.0) **:** A framework for job scheduling and cluster resource management.

Hadoop Distributed File System (HDFS)

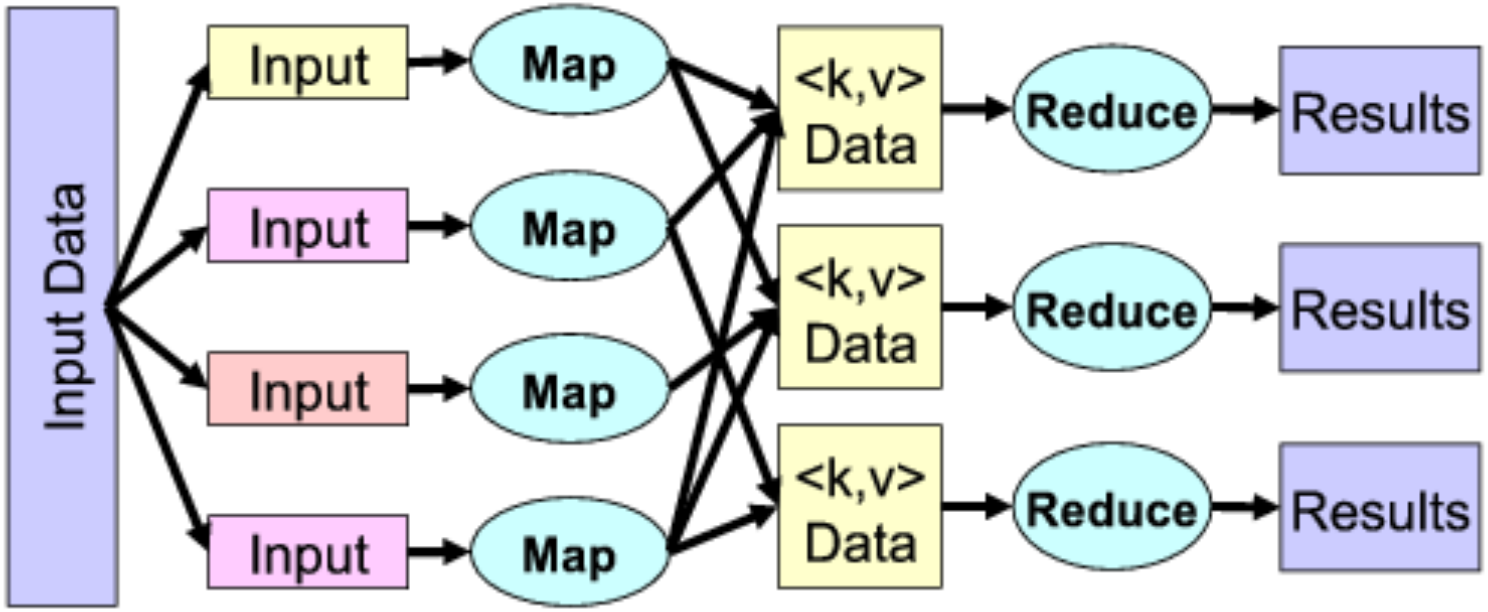


MapReduce / YARN Architecture

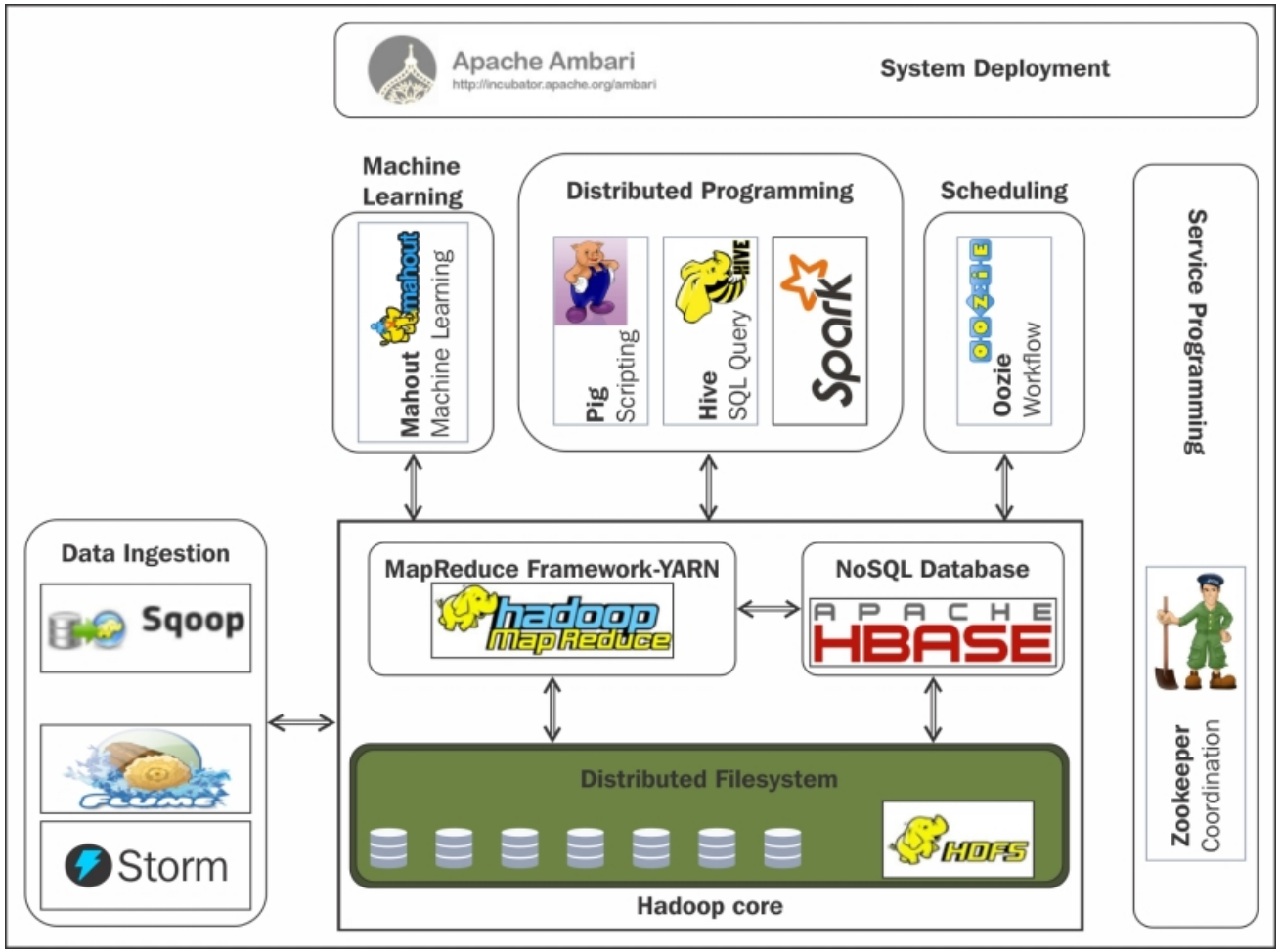


- **YARN** : **Y**et **A**nother **R**esource **N**egotiator
- **MR** : MapReduce

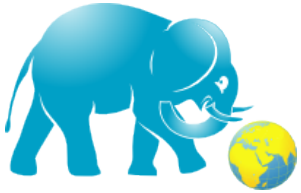
MapReduce computation framework



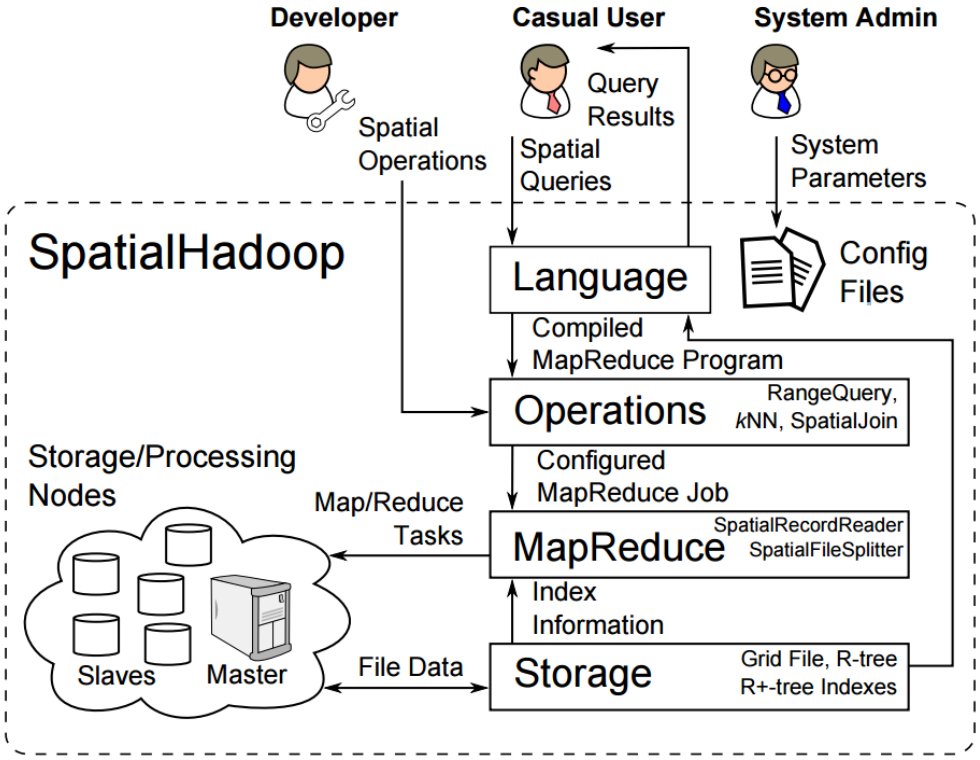
Hadoop ecosystem



SpatialHadoop



"SpatialHadoop is a MapReduce extension to Apache Hadoop designed specially to work with spatial data."



Key Features:

- **Language: Pig**
- **Built-in Data Types**
- **Distributed Spatial Index**
- **Pre-defined spatial operations**

Apache Spark 

"Fast and General engine for large-scale data processing."

- **Speed** : By exploiting **in-memory computing** and other optimizations, Spark can be 100x faster than Hadoop for large scale data processing.

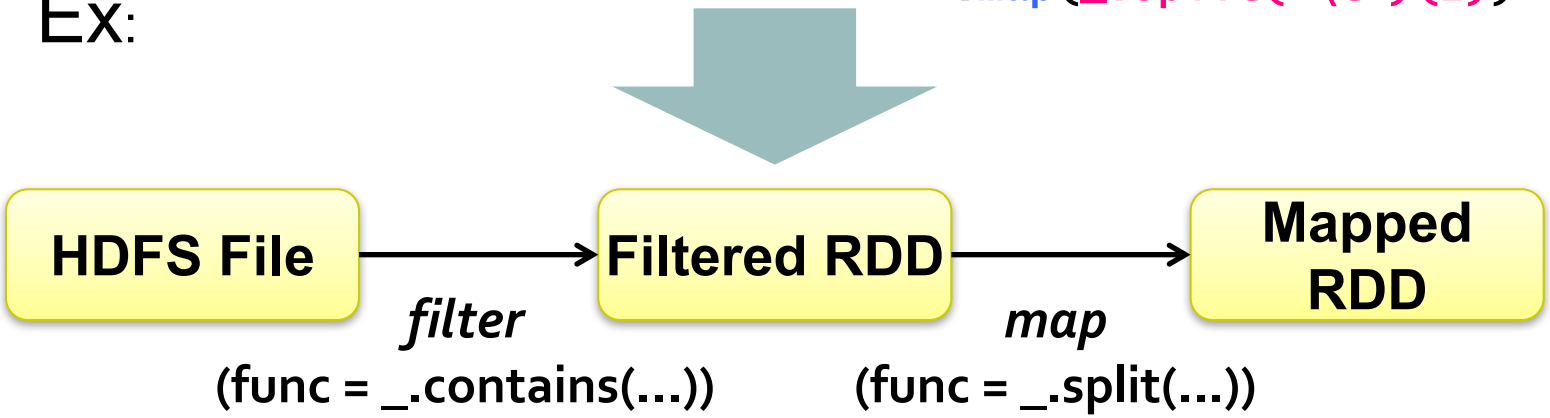
- **Ease of Use** : Spark has easy-to-use **language integrated APIs** for operating on large datasets.

- **A Unified Engine** : Spark comes packed with **higher-level libraries**, including support for ***SQL queries***, ***streaming data***, ***machine learning*** and ***graph processing***.

Resilient Distributed Datasets (RDDs)

- Immutable, partitioned collections of objects
- Created through parallel ***transformations*** (map, filter, groupBy, join…) on data in stable storage.
- Can be ***cached in memory*** for efficient reuse.
- Retain the attractive properties of MapReduce:
  - Fault tolerance, data locality, scalability…
- Maintain linage information that can be used to reconstruct lost partitions.
- Ex:

```
messages = textFile(...).filter(_.startsWith("ERROR"))
                        .map(_.split('\t')(2))
```



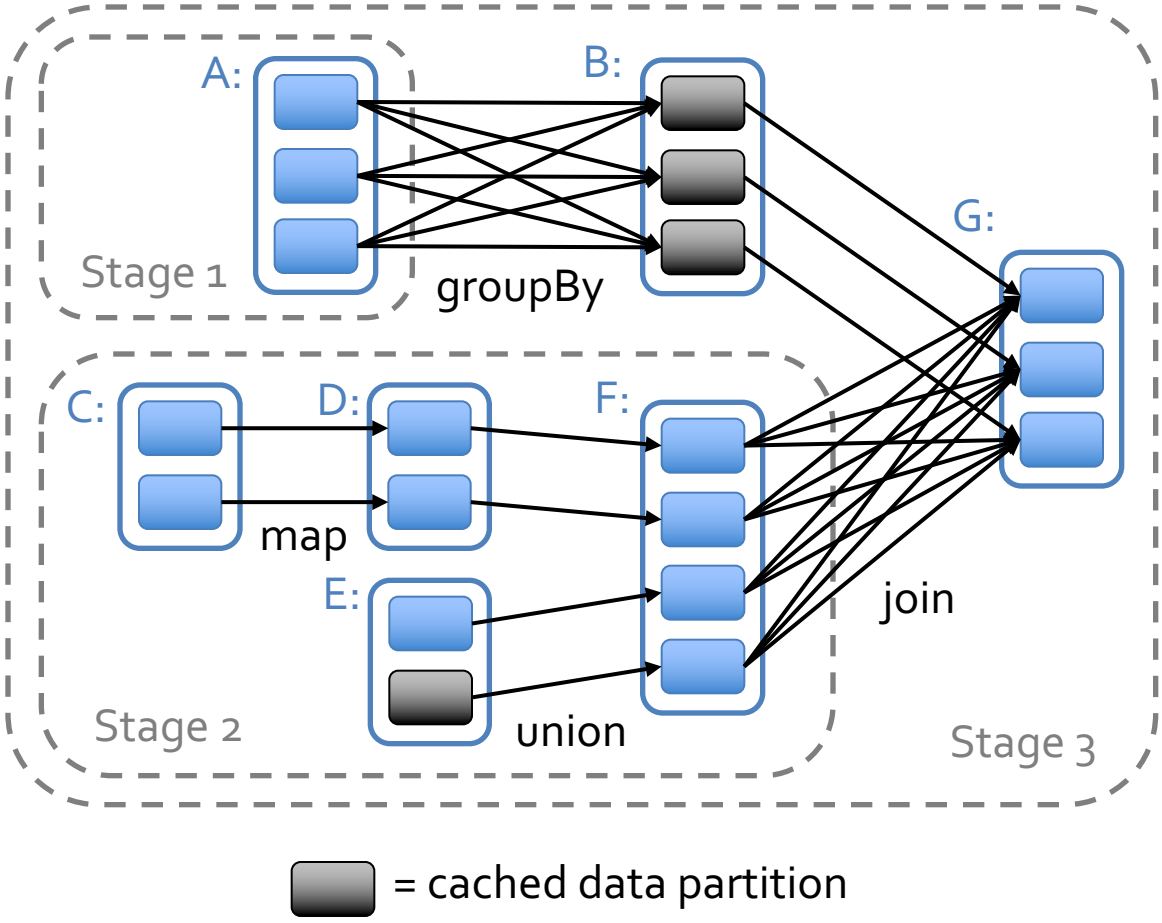
Spark scheduler

DAG based scheduler

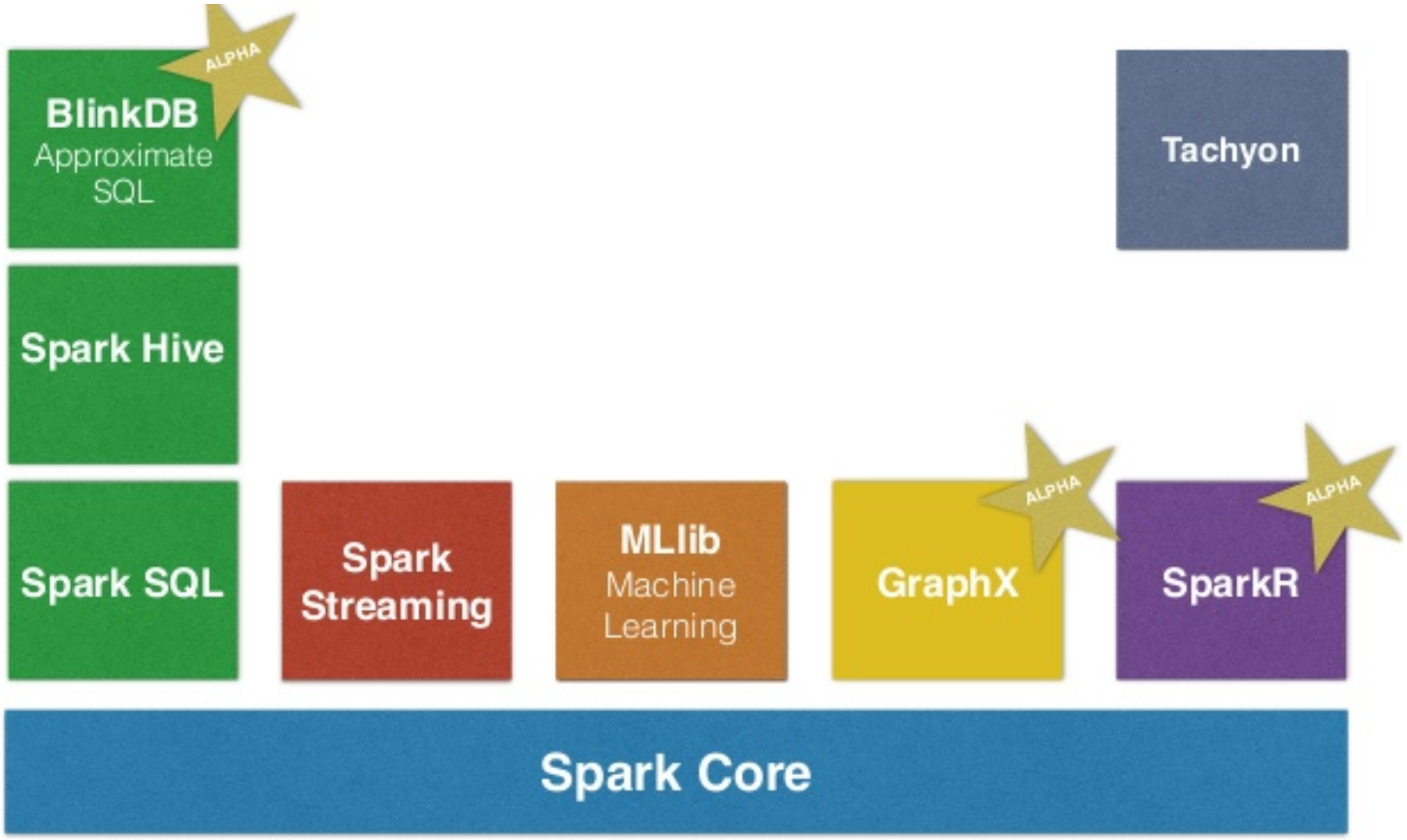
Pipeline functions within a stage

Cache-aware work reuse & locality

Partitioning-aware to avoid shuffles

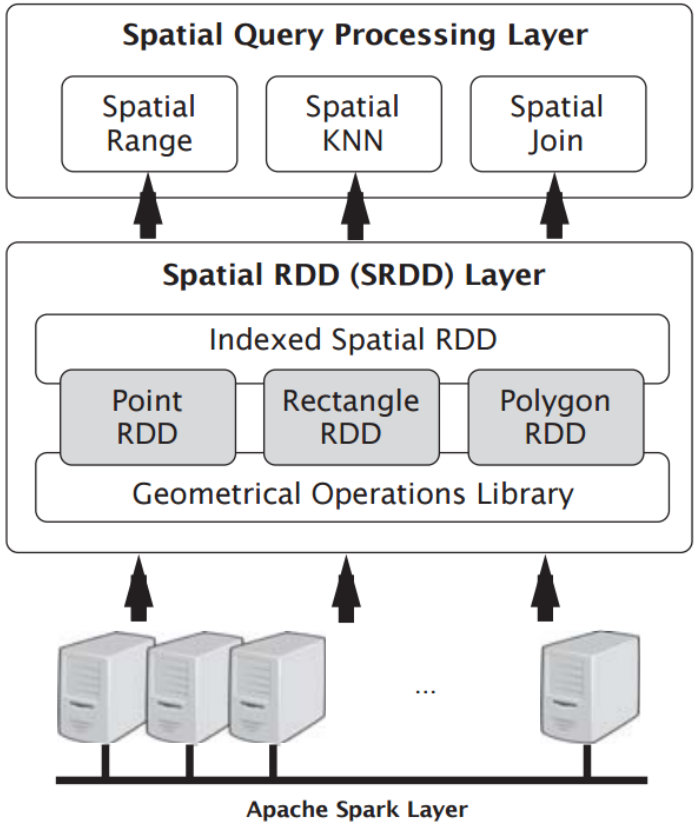


Spark components



GeoSpark GeoSpark

"A spatial library based on Spark for large-scale spatial data processing."

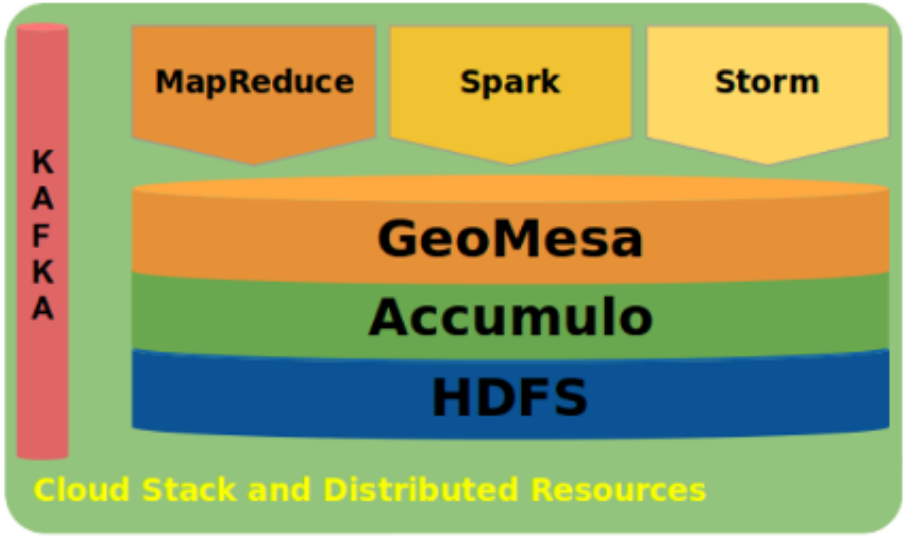


Key Features:

- **Spatial RDD (SRDD) API**
- **Out-of-box spatial functions**
- **Spatial Partitioning**
- **Local Indexing**

GeoMesa

"A distributed, spatial-temporal database built on top of the Apache Accumulo column family store."



KEY						VALUE
ROW			COLUMN FAMILY	COLUMN QUALIFIER	TIMESTAMP	VIZ	Byte-encoded **SimpleFeature**
Epoch Week *2 bytes*	**Z3(x,y,t)** *8 bytes*	Unique ID (such as UUID)	"F"	-	-	Security tags	

Key Features:

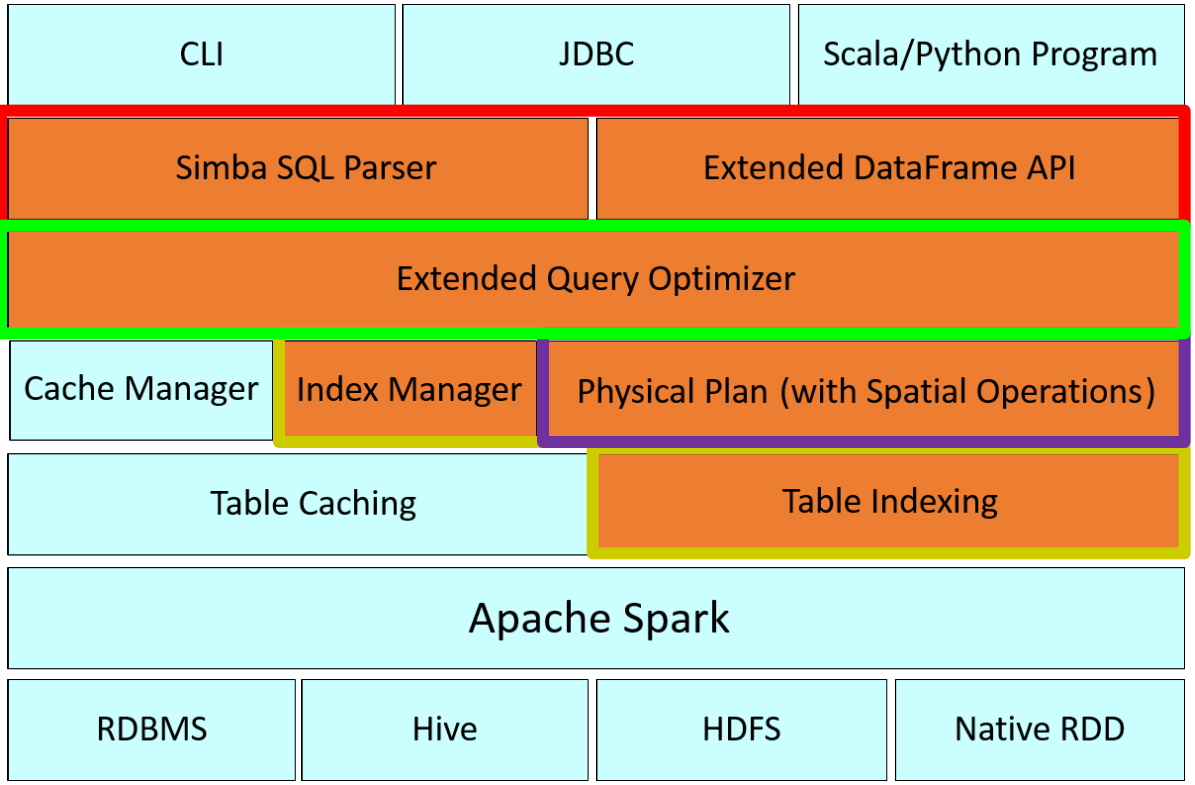
- **GeoHash spatial indexing**
- **OGC standard data access**
- **Built-in Geotools interface**
- **Plugin to GeoServer**

Problems of existing systems

- **Single Node Database -> low scalability**
  - ArchGIS, PostGIS, Oracle Spatial
- **Disk-oriented -> low performance**
  - GeoMesa, Hadoop-GIS, SpatialHadoop
- **No native support for spatial operators**
  - Spark SQL, MemSQL
- **No sophisticated query planner & optimizer**
  - SpatialSpark, GeoSpark

Simba

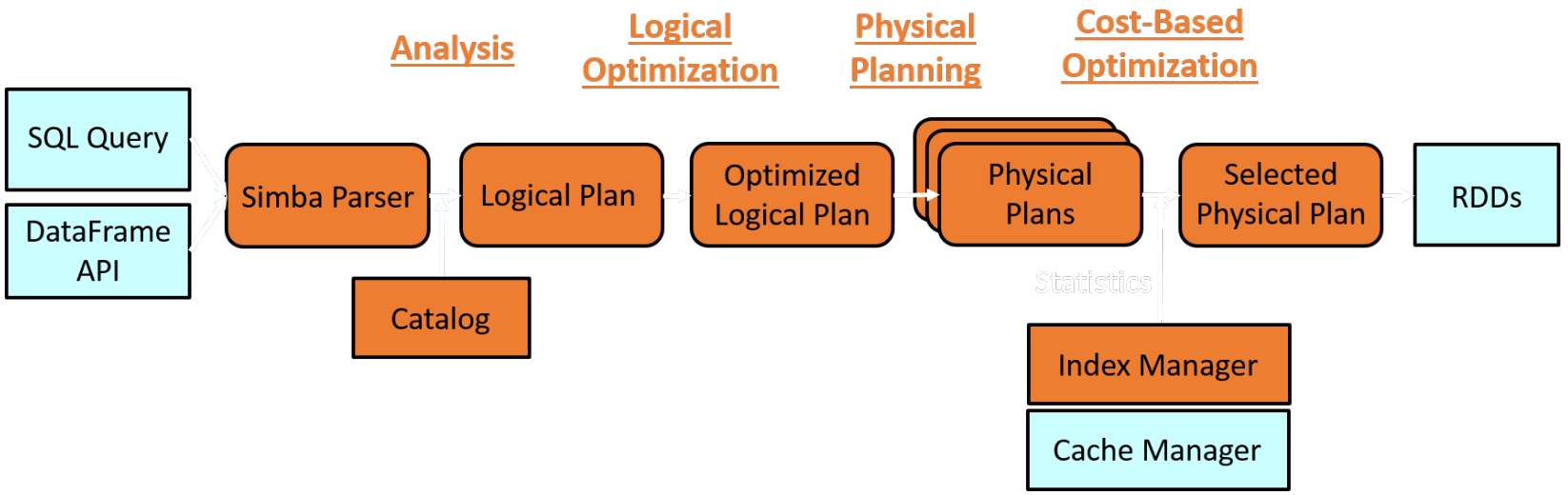
Simba is an extension of Spark SQL across the system stack!



1. ***Programming Interface***
2. ***Table Indexing***
3. ***Efficient Spatial Operators***
4. ***New Query Optimizations***

Introducing Simba (cont'd)

Life of a query in Simba



Programming interface

- Extends both SQL Parser and DataFrame API of Spark SQL
- Make spatial queries more natural

```
SELECT *
FROM points
SORT BY (x - 2)*(x - 2)
      + (y - 3)*(y - 3)
LIMIT 5
```

```
SELECT *
FROM points
WHERE POINT(x, y)
   IN KNN(POINT(2, 3), 5)
```

- Achieve something that is impossible in Spark SQL.

```
SELECT *
FROM queries q KNN JOIN pois p
     ON POINT(p.x, p.y) IN KNN(POINT(q.x, q.y), 3)
```

Programming interface (cont'd)

- **Fully compatible with original Spark SQL operators.**

```
SELECT poi.id, count(*) as c
FROM poi DISTANCE JOIN data
     ON POINT(data.lat, data.long)
        IN CIRCLERANGE(POINT(poi.lat, poi.long), 3.0)
WHERE POINT(data.lat, data.long)
      IN RANGE(POINT(24.39, 66.88), POINT(49.38, 124.84))
GROUP BY poi.id
ORDER BY poi.id
```

- **Same level of flexibility for DataFrame**

```
poi.distanceJoin(data, Point(poi("lat"), poi("long")),
                       Point(data("lat"), data("long")), 3.0)
   .range(Point(data("lat"), data("long")),
          Point(24.39, 66.88), Point(49.38, 124.84))
   .groupBy(poi("id"))
   .agg(count("*").as("c")).sort(poi("id")).show()
```

Supported SQL & DataFrame API

- **Point wrapper**

```
SQL: POINT(pois.x + 2, pois.y * 3)
DataFrame: Point(pois("x") + 2, pois("y") * 3)
```

- **Box range query**

```
SQL: p IN RANGE(low, high)
DataFrame: range(base: Point, low: Point, high: Point)
```

- **Circle range query**

```
SQL: p IN CIRCLERANGE(c, rd)
DataFrame: circleRange(base: Point, c: Point, rd: Double)
```

- ***k* nearest neighbor query**

```
SQL: p IN KNN(q, k)
DataFrame: knn(base: Point, k: Int)
```

Supported SQL & DataFrame API (cont'd)

- **Distance join**

```
SQL: R DISTANCE JOIN S ON s IN CIRCLERANGE(r, τ)
DataFrame: distanceJoin(target: DataFrame, left_key: Point,
                        right_key: Point, τ: Double)
```

- ***k*NN join**

```
SQL: R KNN JOIN S ON s IN KNN(r, k)
DataFrame: knnJoin(target: DataFrame, left_key: Point,
                   right_key: Point, k: Int)
```

- **Index management**

```
SQL: CREATE INDEX idx_name ON R(x_1, ..., x_m) USE idx_type
     DROP INDEX idx_name ON table_name
     DROP INDEX ON table_name
     SHOW INDEX ON R
DataFrame: index(idx_type: IndexType, idx_name: String,
                 attrs:Seq[Attribute])
           dropIndex()
           showIndex()
```

Zeppelin integration.

"A web-based notebook that enables interactive data analytics."

```
%sql
SELECT poi.id, count(*) as count
FROM poi DISTANCE JOIN data
ON POINT(data.lat, data.long) IN CIRCLERANGE(POINT(poi.lat, poi.long), 3)
WHERE POINT(data.lat, data.long) IN RANGE(POINT(24.39, 66.88), POINT(49.38, 124.84))
GROUP BY poi.id
ORDER BY poi.id
```

FINISHED

settings

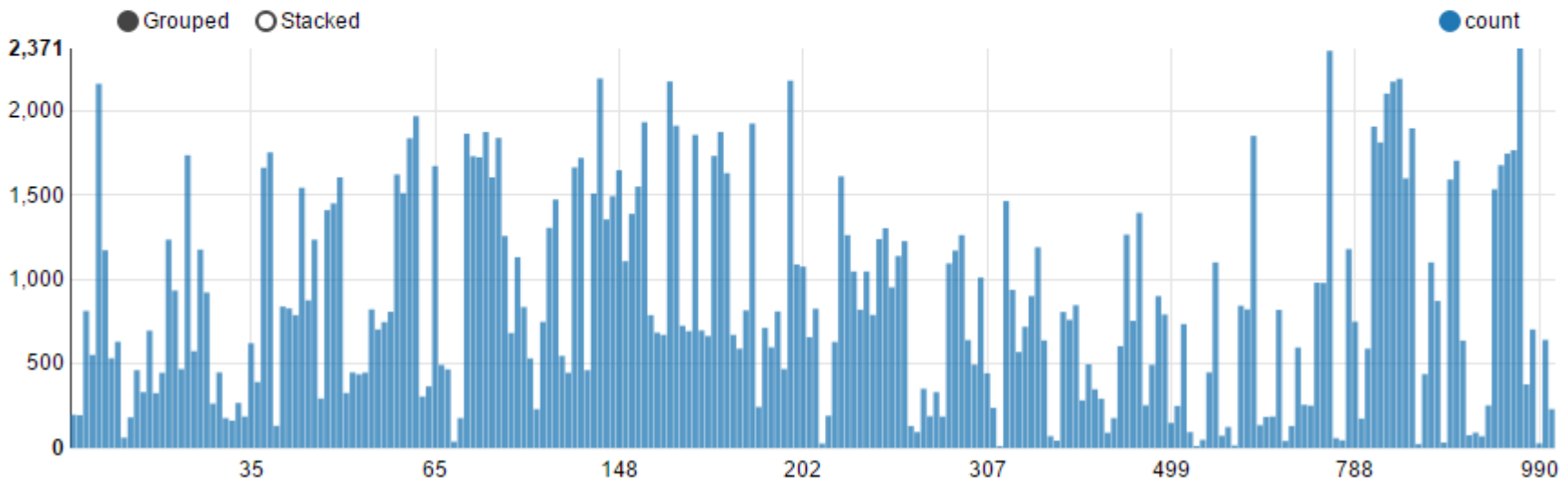


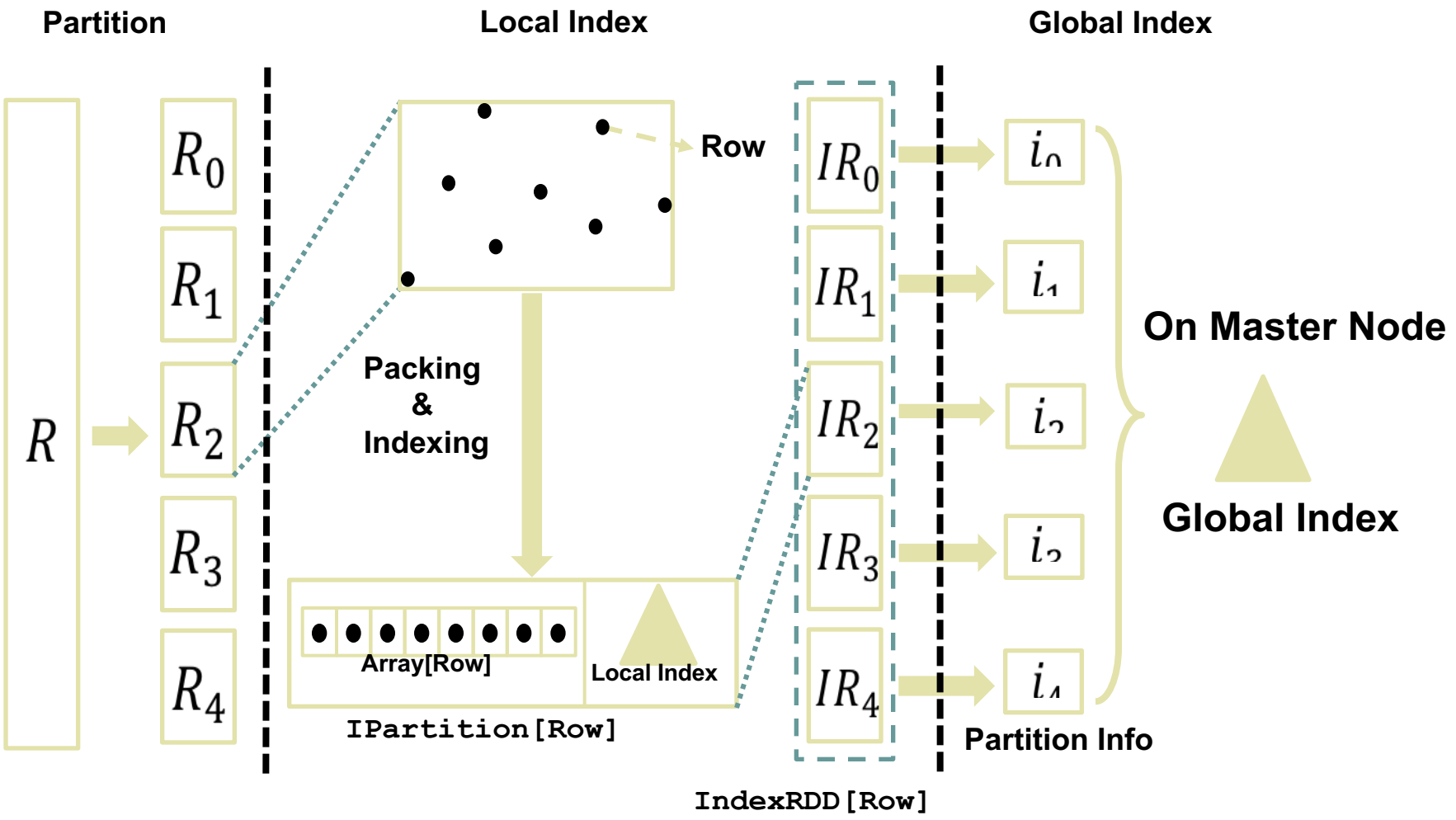
Table Indexing

- All Spark SQL operations are based on RDD scanning.
- Inefficient for selective spatial queries!

- In Spark SQL:
  - **Record -> Row**
  - **Table -> RDD[Row]**
- Solution in Simba: ***native two-level indexing over RDDs***
- Challenges:
  - RDD is not designed for ***random access***
  - **Achieve this *without hurting Spark kernel and RDD abstraction***

Table Indexing (cont'd)

Two-level Indexing Framework: *local* + *global indexing*



Global Index

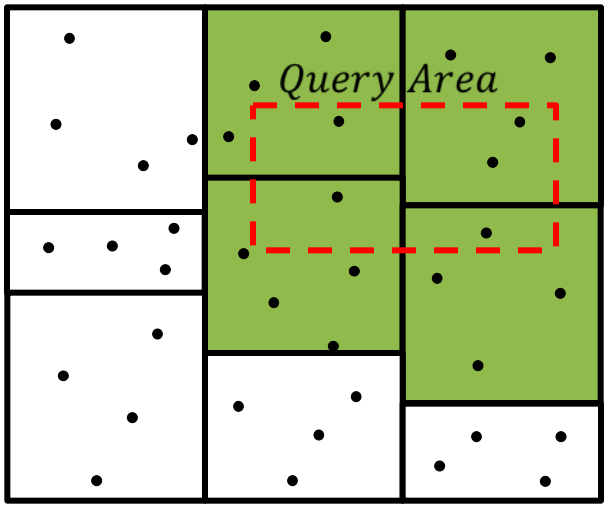
- Partition boundary information is collected by the master node
- A global index is built over these data and stored in the driver program on the master node.
- Even for big data, the number of partitions is not very large (from several hundreds to tens of thousands). Thus, global index can easily fit in the memory of the master node.

e.g. the global index only consumes less than 700KB for our largest data set with 4.4 billion records.

Spatial operations

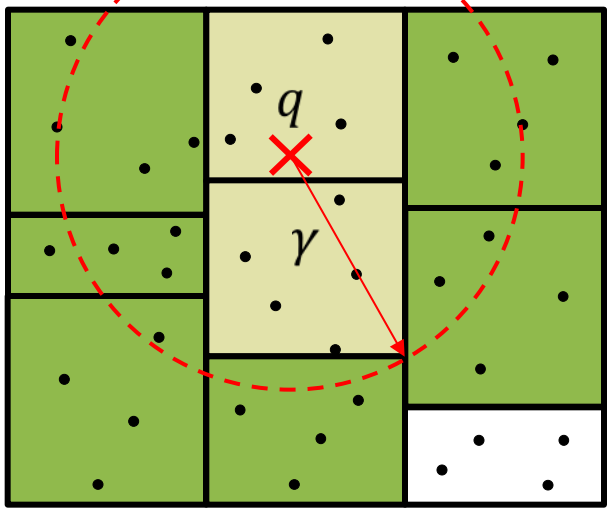
- Indexing support -> efficient algorithms

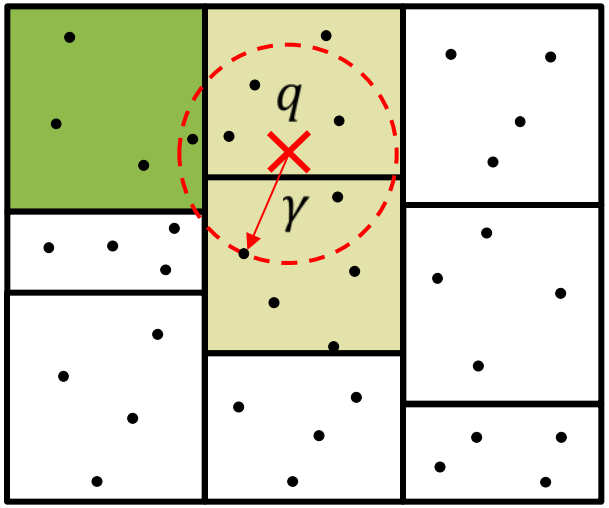
- Range Query : $range(Q, R)$
- Two steps: global Filtering + local processing



Spatial operations (cont'd)

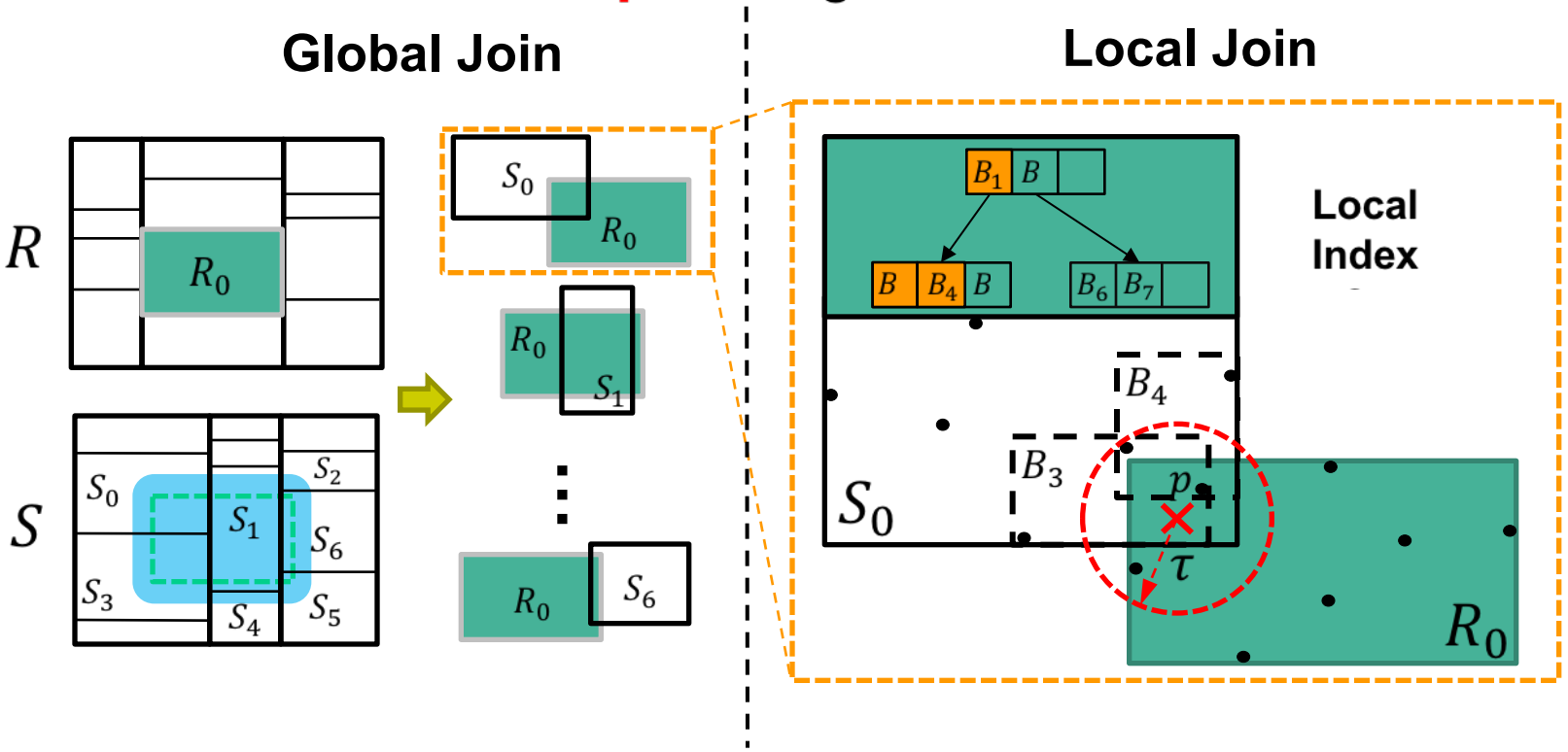
- k nearest neighbor query : $knn(q,\ R)$
- Use global index to retrieve a radius r such that the partitions within r cover at least k points (global index stores the count from each partition)
- Key to achieve good performance:
  - **Local indexes**
  - **Pruning bound** that is sufficient to cover global kNN results.





Spatial operations (cont'd)

- Distance Join : $R \bowtie_\tau S$
- General theta-join in Spark SQL -> ***Cartesian product!!!***
- Our solution: the **DJSpark** Algorithm



Spatial operations (cont'd)

- kNN join : $R \ltimes_{kNN} S$
- Solutions in Simba:
  - Block Nested Loop kNN join (***BKJSpark-N***)
  - Block Nested Loop kNN join with local R-Trees (***BKJSpark-R***)
  - Voronoi kNN join* (***VKJSpark***)
  - z-value kNN join+ (***ZKJSpark***) -> approximate kNN join
  - **R-Tree kNN join (*RKJSpark*)**

Query optimizer

- Spatial predicates merging.
- Selectivity estimation + Cost-based Optimization
  - Selectivity estimation over local indexes
  - Choose a proper plan: **scan** or **use index**.
- Broadcast join optimization: ***small table*** **joins large table**
- Logical partitioning optimization for RKJSpark
  - **provides *tighter pruning bounds*** γ_i

Comparison with existing systems

Core Features	Simba	GeoSpark	SpatialSpark	SpatialHadoop	Hadoop GIS
Data dimensions	multiple	$d \leq 2$	$d \leq 2$	$d \leq 2$	$d \leq 2$
SQL	✓	×	×	Pigeon	×
DataFrame API	✓	×	×	×	×
Spatial indexing	R-tree	R-/quad-tree	grid/kd-tree	grid/R-tree	SATO
In-memory	✓	✓	✓	×	×
Query planner	✓	×	×	✓	×
Query optimizer	✓	×	×	×	×
Concurrent query execution	thread pool in query engine	user-level process	user-level process	user-level process	user-level process
query operation support					
Box range query	✓	✓	✓	✓	✓
Circle range query	✓	✓	✓	×	×
k nearest neighbor	✓	✓	only 1NN	✓	×
Distance join	✓	✓	✓	via spatial join	✓
kNN join	✓	×	×	×	×
Geometric object	×[1]	✓	✓	✓	✓
Compound query	✓	×	×	✓	×