# Writing better spark using Spark Execution plans

Michael Johnson

Cobalt Analytics

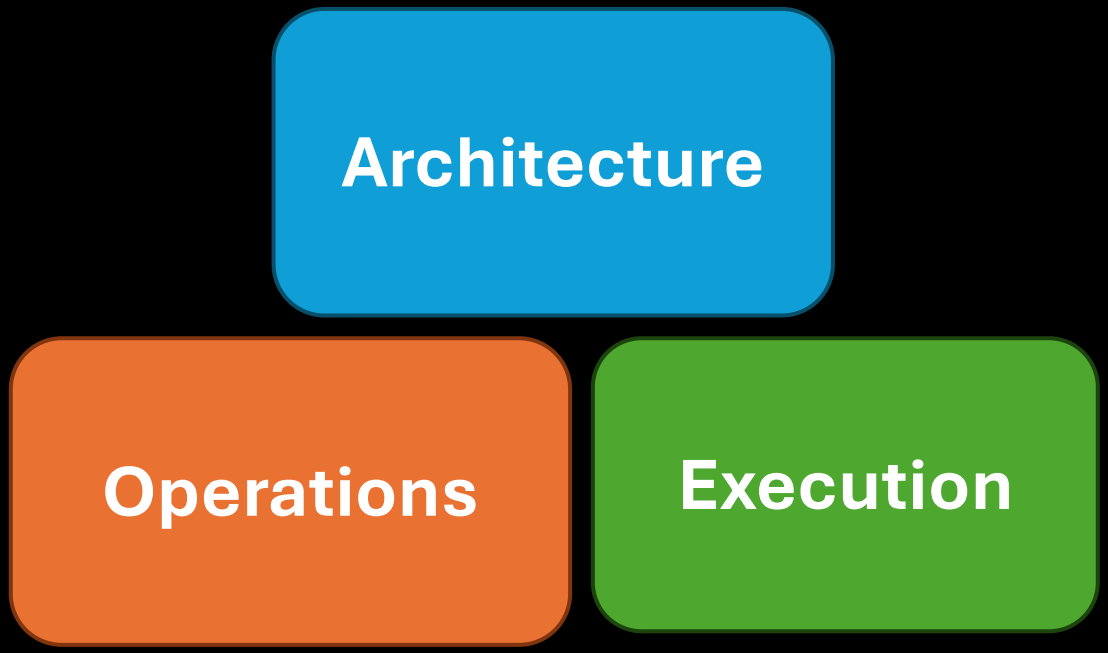
Michael@CobatlAnalytics.io

# Agenda

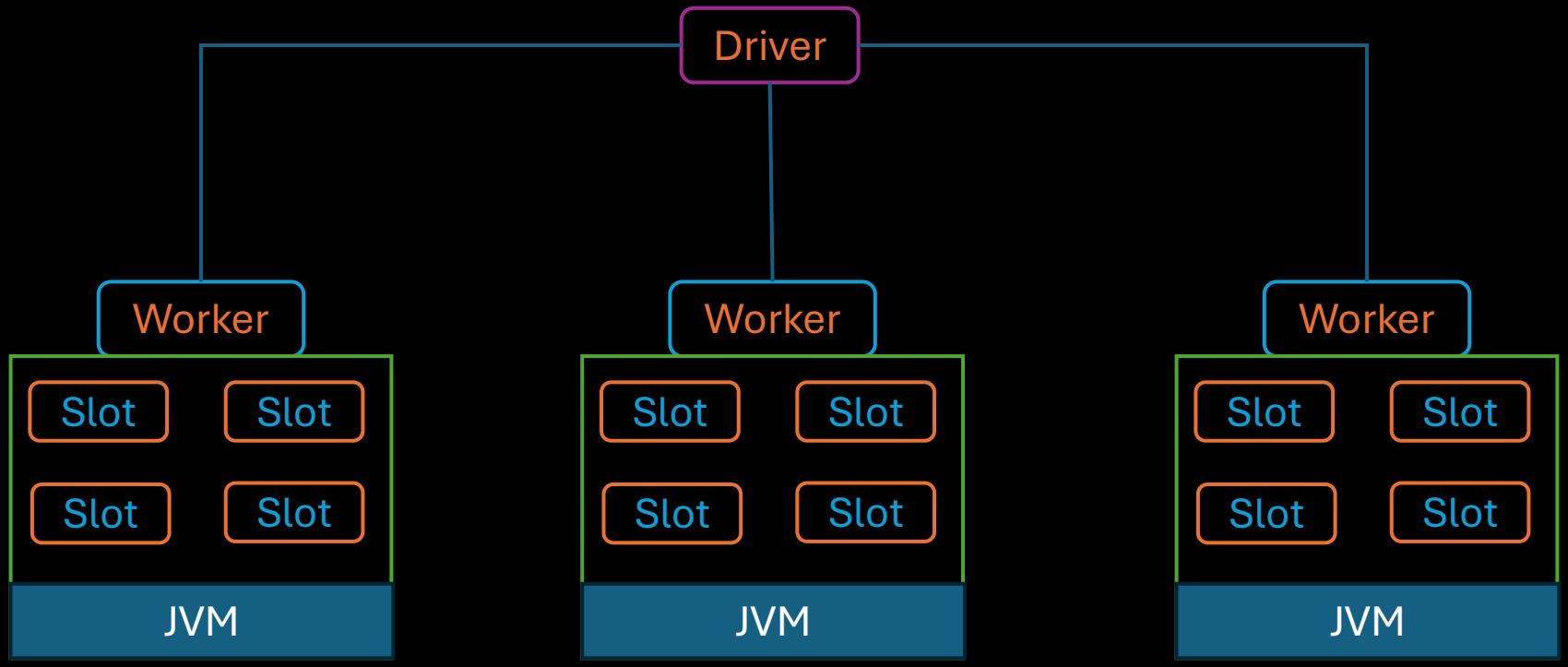
1. Spark fundamentals recap
2. Spark execution plans
3. Engine Enhancements
4. Best Practices and what can you control

# Spark fundamentals recap

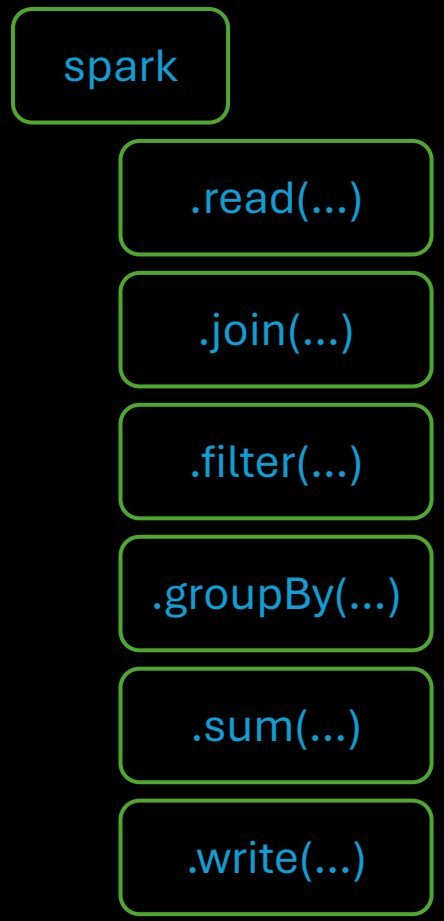
# Spark Fundamentals Recap

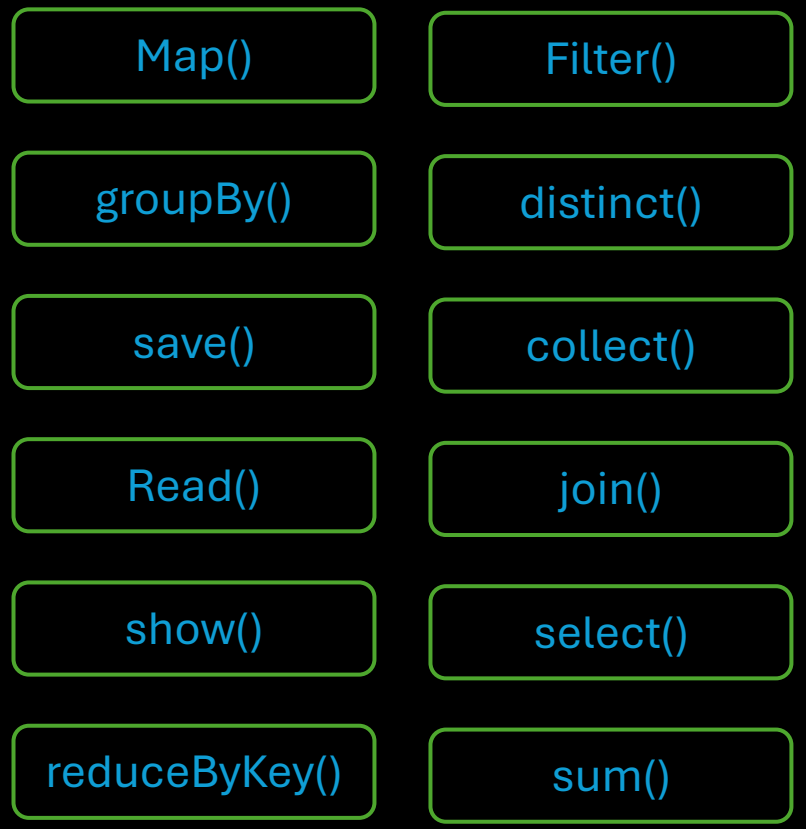


# Spark Architecture



# Spark Operations



# Spark Operations

Map()

Filter()

groupBy()

distinct()

save()

collect()

Read()

join()

show()

select()

reduceByKey()

sum()

Actions

Transformations

# Spark Operations

Map()

Filter()

groupBy()

distinct()

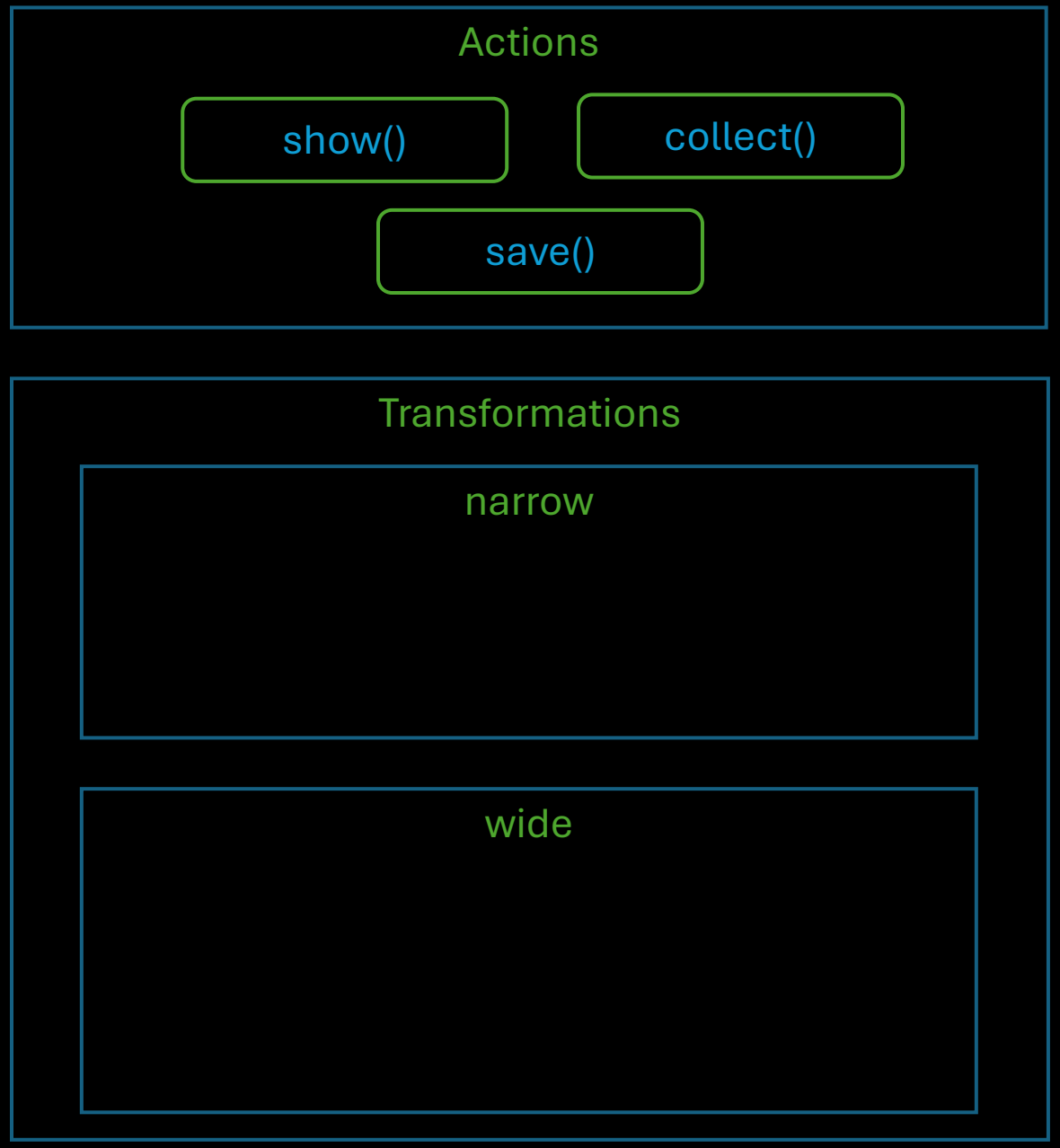
Read()

join()

select()

reduceByKey()

sum()



# Spark Operations

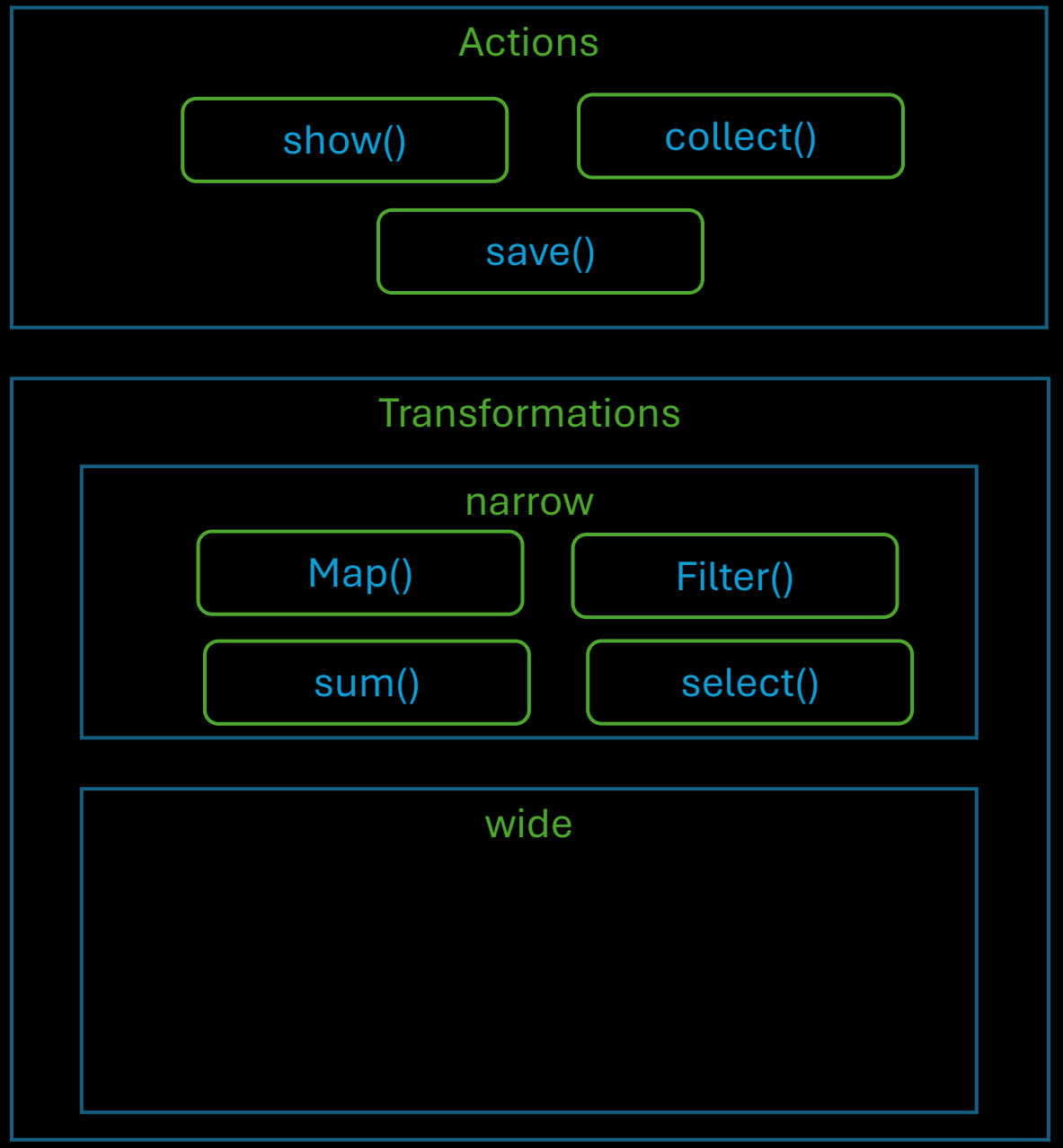
groupBy()

distinct()

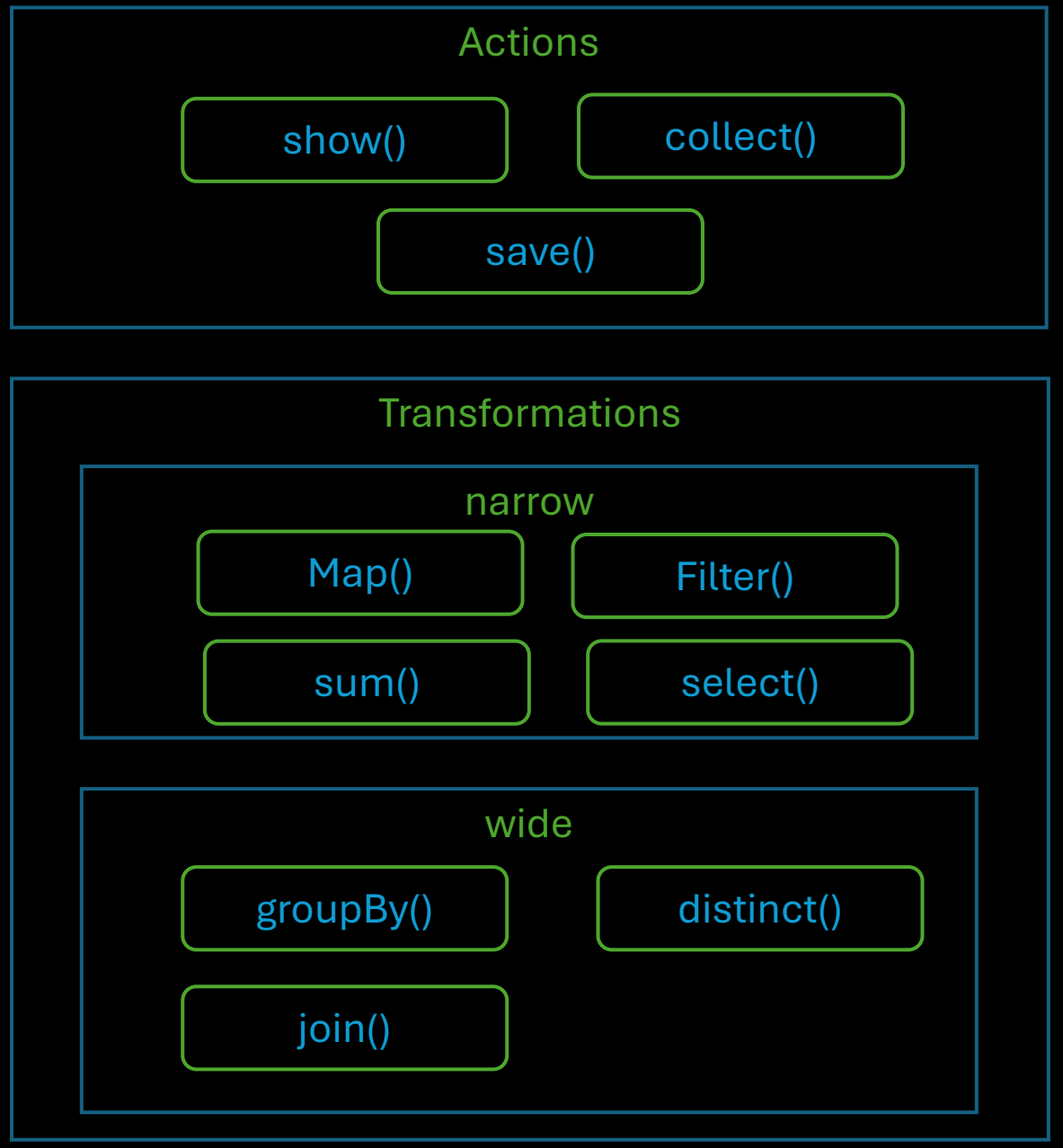
Read()

join()

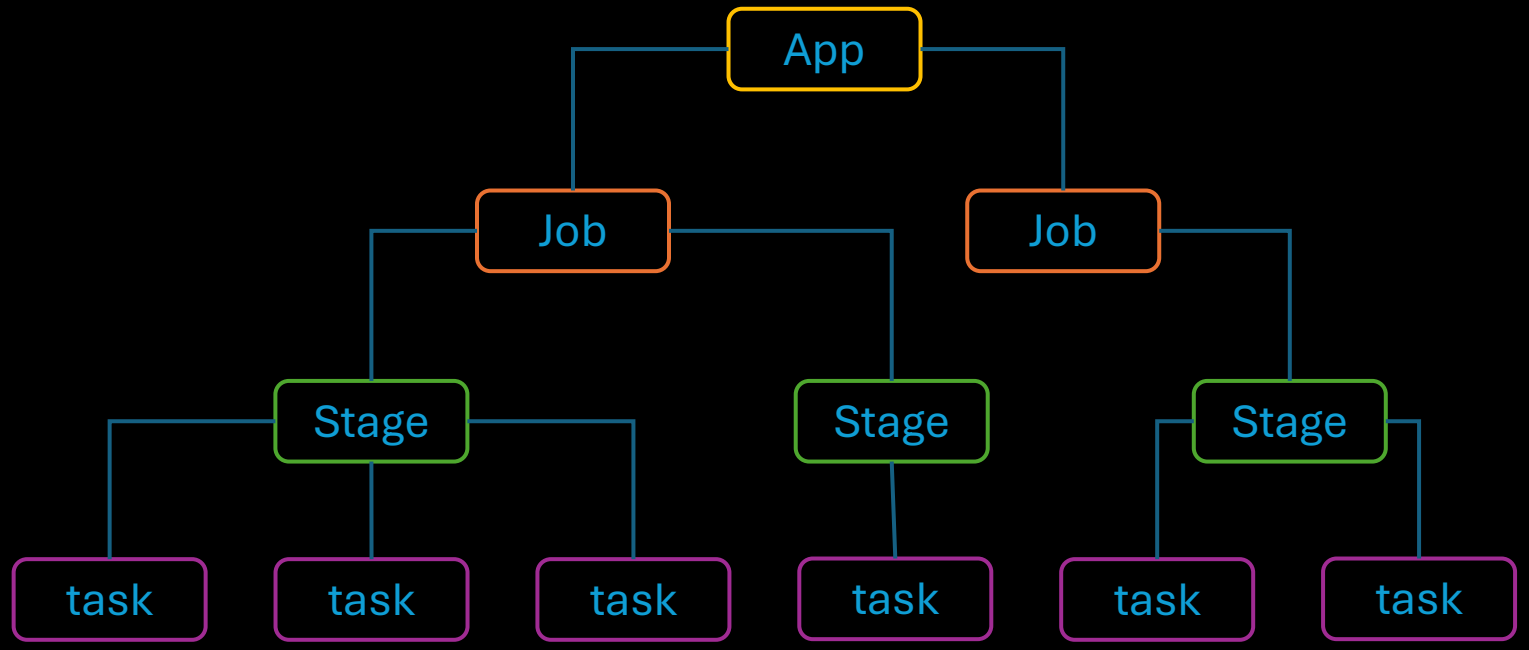
reduceByKey()



# Spark Operations

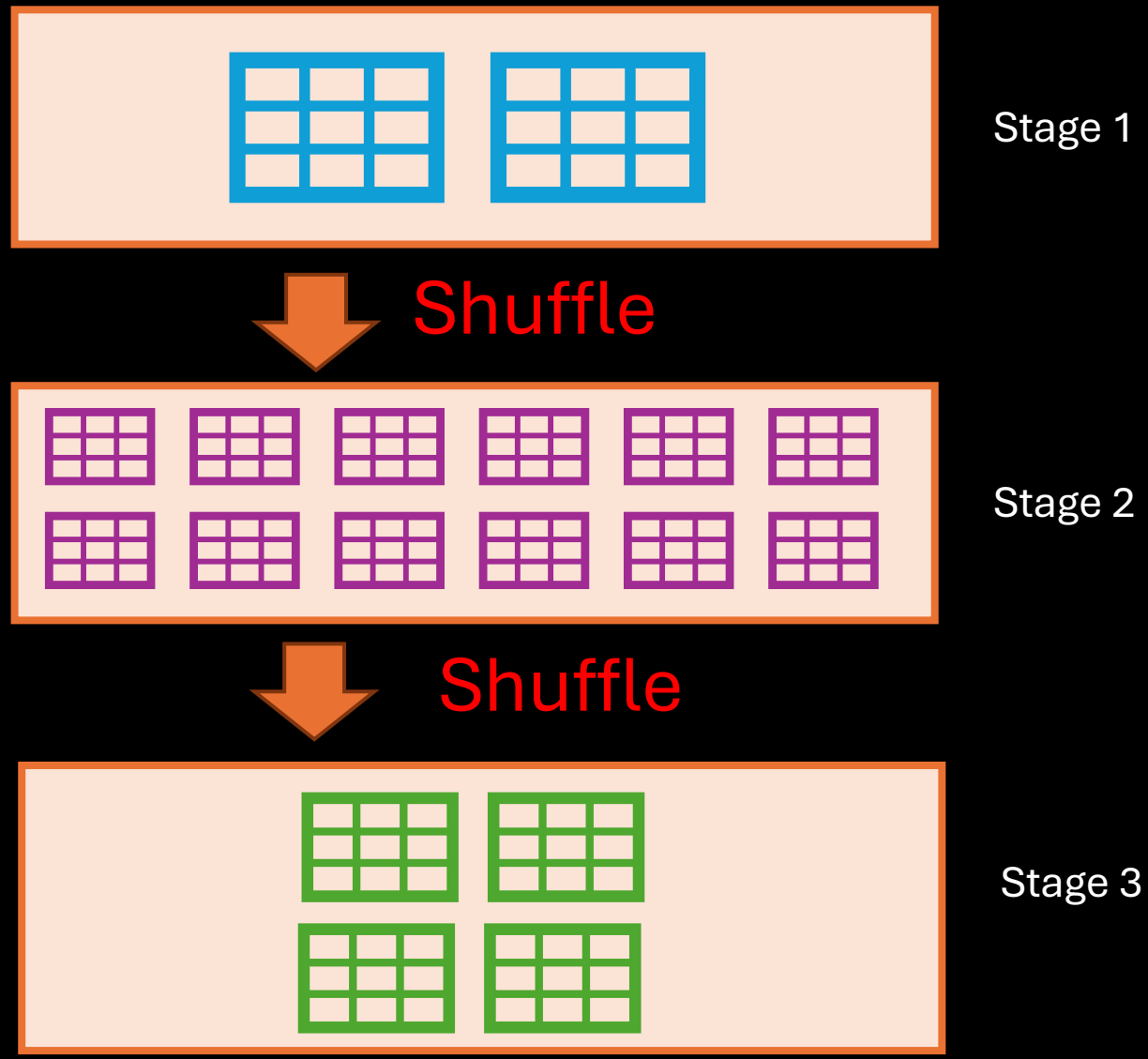


# Spark Execution



# Execution Flow

```
Spark
        .read(...)
        .join(...)
        .filter(...)
        .groupBy(...)
        .sum(...)
        .write(...)
```



# Why Are Shuffles bad

- Network Overhead
- Disk Spills
- Data Skew

# Spark Execution Plans

# How to view the pre-execution plan

DataFrame API

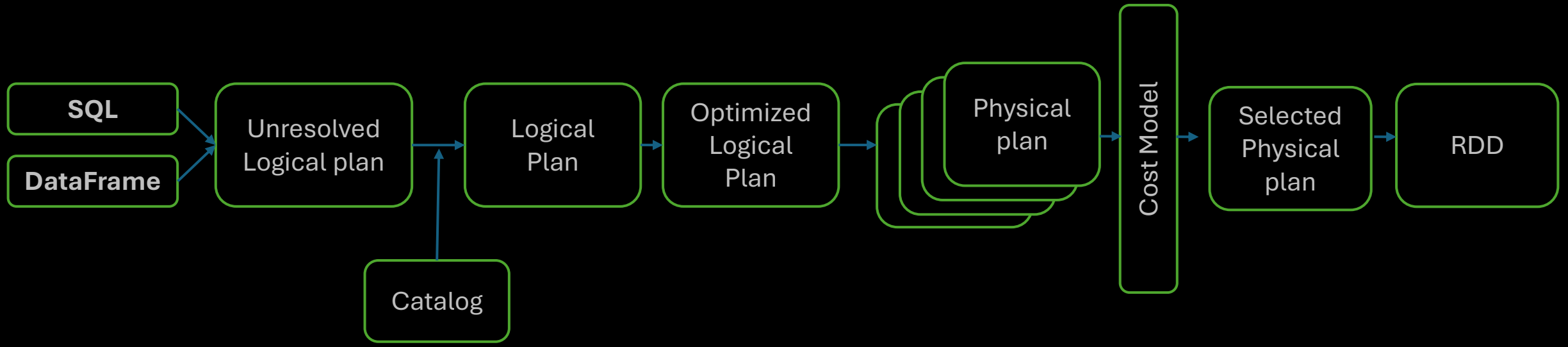
df = spark.(.....)
df.explain(mode=‘’)

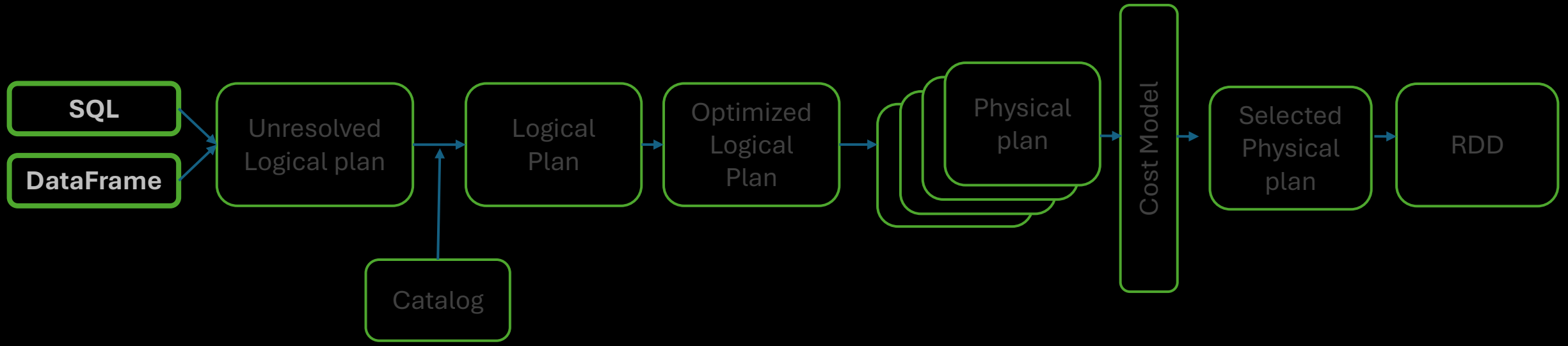
SQL

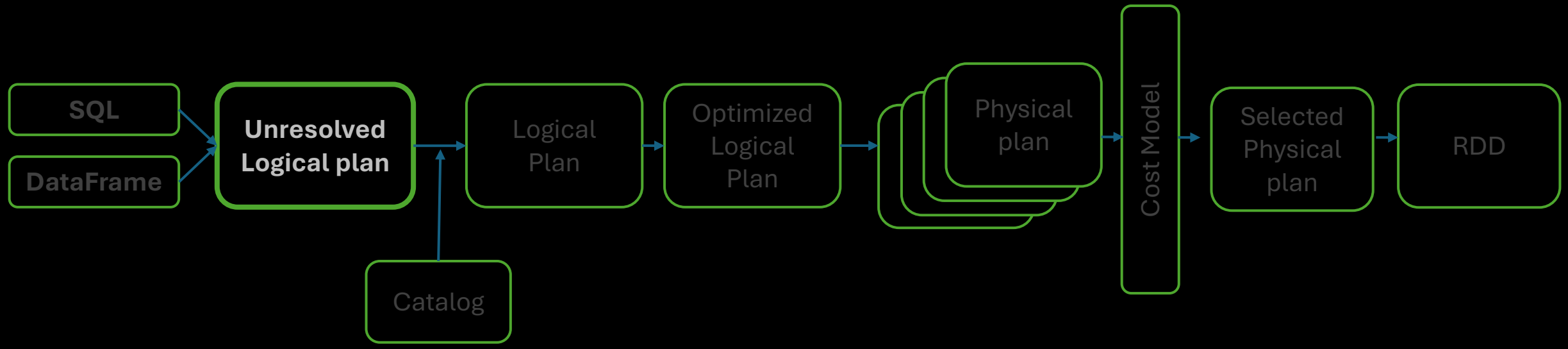
EXPLAIN []
Select ....
From .....

# Pre execution plan (Explain) settings

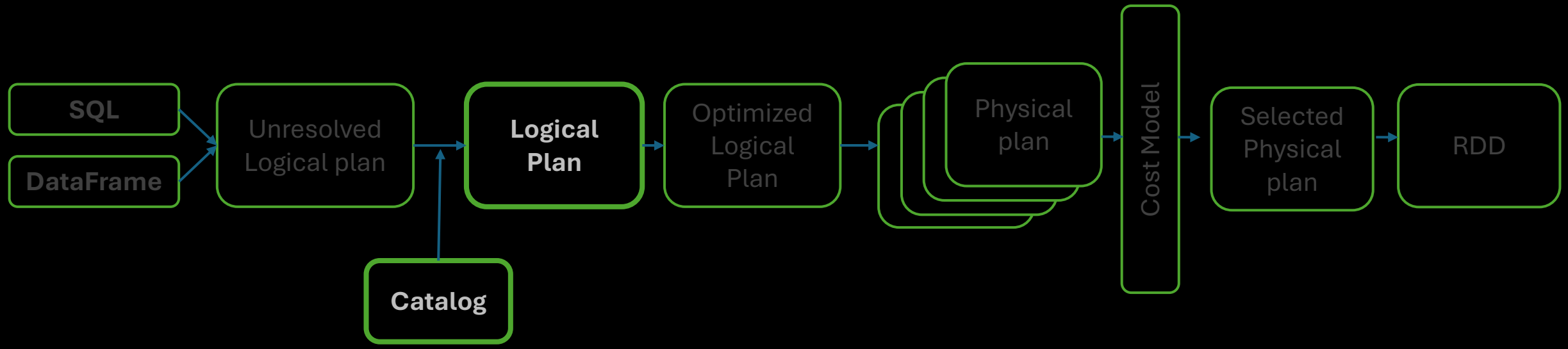
| Command | What it shows |
| --- | --- |
| Simple | Physical execution plan only |
| Extended | Parsed, analyzed, optimized logical plans + physical plan |
| Codegen | Generated Java code for whole-stage code generation |
| Cost | Optimized logical plan with estimated statistics/costs |
| Formatted | Nicely structured physical plan with operator details |


SQL
DataFrame
Unresolved Logical plan
Logical Plan
Optimized Logical Plan
Physical plan
Cost Model
Selected Physical plan
RDD
Catalog

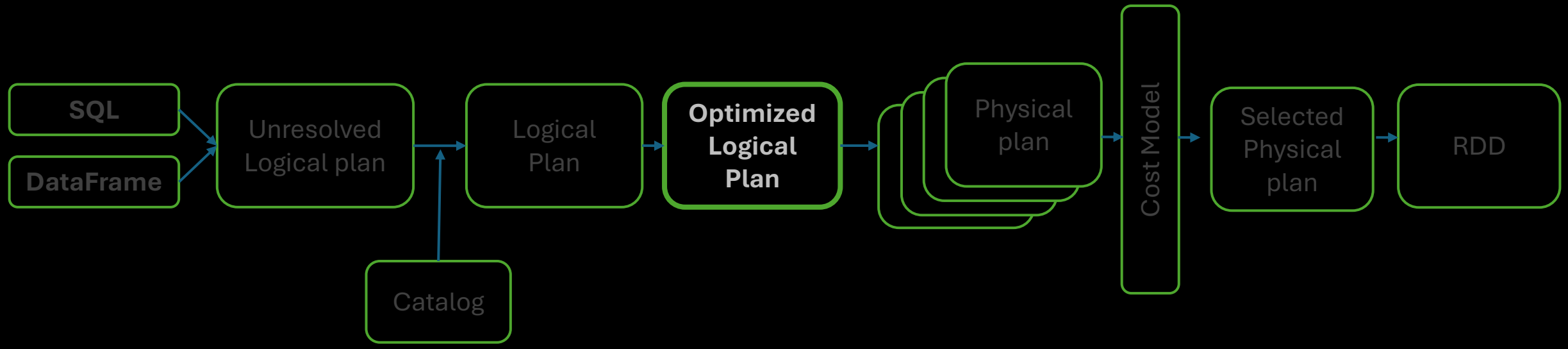

SQL
DataFrame
Unresolved Logical plan
Logical Plan
Optimized Logical Plan
Physical plan
Cost Model
Selected Physical plan
RDD
Catalog



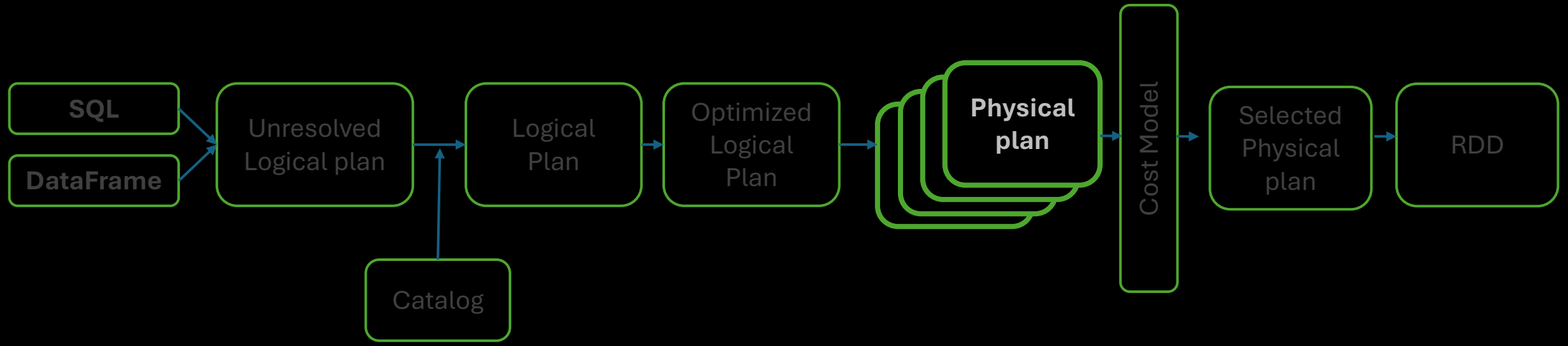

```
== Parsed Logical Plan ==
'SubqueryAlias total_fares
+- 'Aggregate ['home_area], ['home_area, unresolvedalias('sum('fare_amount))]
  +- 'Filter '`==` ('trip_date, 2024-06-02)
    +- 'Join Inner, '`==` ('driver_id, 'driver_id)
      :- 'UnresolvedRelation [demo, trips, trips], [], false
      +- 'UnresolvedRelation [demo, trips, drivers], [], false
```

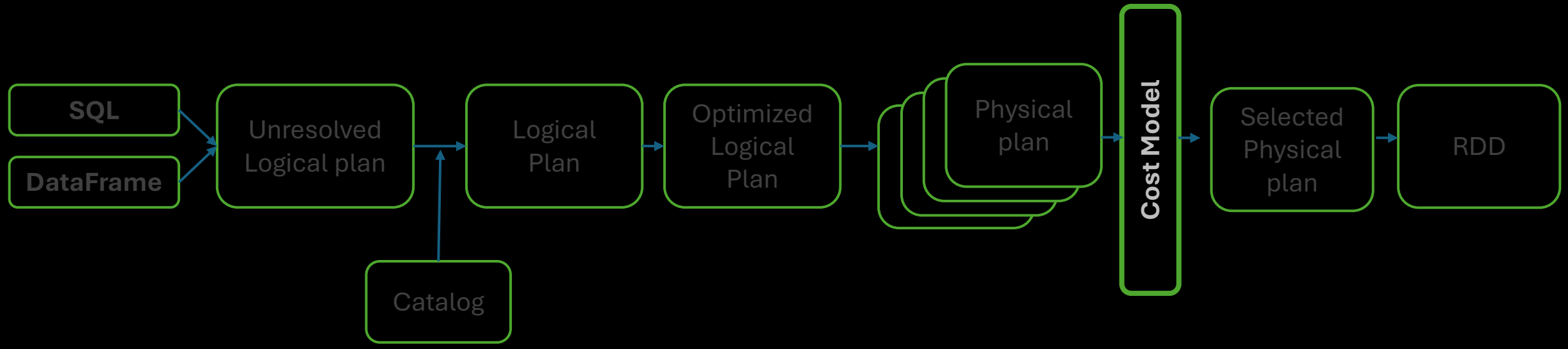



```
== Analyzed Logical Plan ==
home_area: string, sum(fare_amount): double
SubqueryAlias total_fares
+- Aggregate [home_area#2861], [home_area#2861, sum(fare_amount#2851) AS sum(fare_amount)#2866]
   +- Filter (trip_date#2847 = 2024-06-02)
      +- Join Inner, (driver_id#2848 = driver_id#2857)
         :- SubqueryAlias demo.trips.trips
         :  +- Relation demo.trips.trips[trip_id#2846L,trip_date#2847,driver_id#2848, pickup_zone_id#2849, dropoff_zone_id#285
         +- SubqueryAlias demo.trips.drivers
            +- Relation demo.trips.drivers[driver_id#2857,driver_name#2858,tier#2859,start_date#2860,home_area#2861,license
```



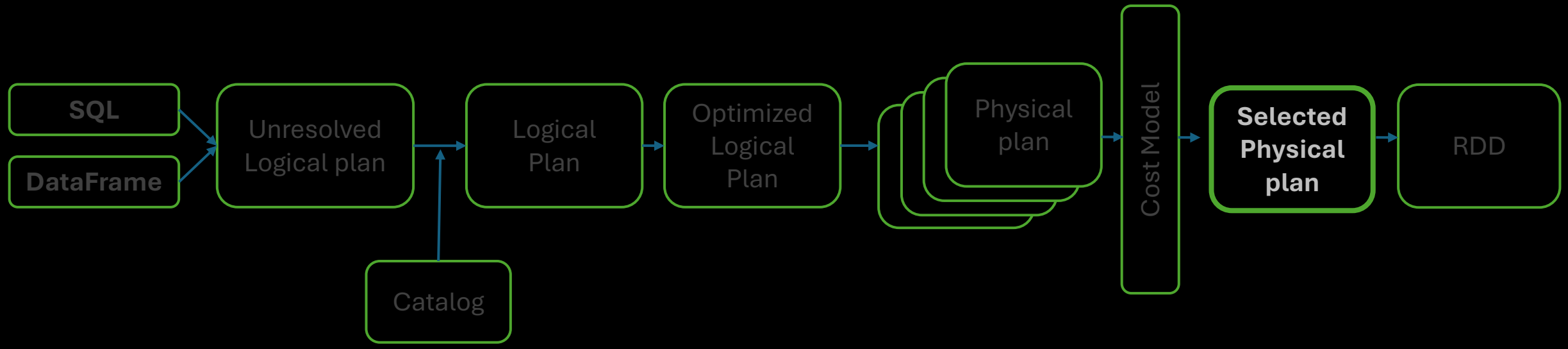

```
== Optimized Logical Plan ==
Aggregate [home_area#2861], [home_area#2861, sum(fare_amount#2851) AS sum(fare_amount)#2866]
+- Project [fare_amount#2851, home_area#2861]
  +- Join Inner, (driver_id#2848 = driver_id#2857)
    :- Project [driver_id#2848, fare_amount#2851]
    :  +- Filter ((isnotnull(trip_date#2847) AND (trip_date#2847 = 2024-06-02)) AND isnotnull(driver_id#2848))
    :     +- Relation demo.trips.trips[trip_id#2846L,trip_date#2847,driver_id#2848,pickup_zone_id#2849,dropoff_zone_id#285
    +- Project [driver_id#2857, home_area#2861]
      +- Filter isnotnull(driver_id#2857)
        +- Relation demo.trips.drivers[driver_id#2857,driver_name#2858,tier#2859,start_date#2860,home_area#2861,license
```


SQL
DataFrame
Unresolved Logical plan
Logical Plan
Optimized Logical Plan
Physical plan
Cost Model
Selected Physical plan
RDD
Catalog

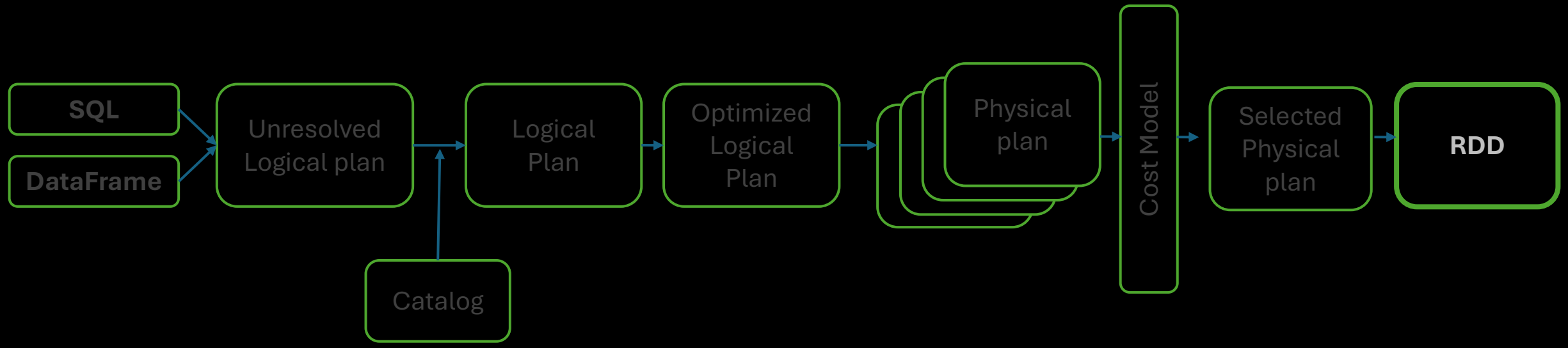



```
== Optimized Logical Plan ==
Aggregate [home_area#3687], [home_area#3687, sum(fare_amount#3677) AS sum(fare_amount)#3692], Statistics(sizeInBy
+- Project [fare_amount#3677, home_area#3687], Statistics(sizeInBytes=3.3 MiB, rowCount=9.48E+4, [fare_amount#3677 ->
   +- Join Inner, (driver_id#3674 = driver_id#3683), Statistics(sizeInBytes=4.0 MiB, rowCount=9.48E+4, [driver_id#3674 -> Colu
      :- Project [driver_id#3674, fare_amount#3677], Statistics(sizeInBytes=1698.4 KiB, rowCount=8.70E+4, [driver_id#3674 ->
      :  +- Filter ((isnotnull(trip_date#3673) AND (trip_date#3673 = 2024-06-02)) AND isnotnull(driver_id#3674)), Statistics(sizeIn
      :     +- Relation demo.trips.trips[trip_id#3672L,trip_date#3673,driver_id#3674,pickup_zone_id#3675,dropoff_zone_id#3676
      +- Project [driver_id#3683, home_area#3687], Statistics(sizeInBytes=1562.5 KiB, rowCount=5.00E+4, [driver_id#3683 -> C
         +- Filter isnotnull(driver_id#3683), Statistics(sizeInBytes=5.2 MiB, rowCount=5.00E+4, [license_type#3688 -> ColumnSta
            +- Relation demo.trips.drivers[driver_id#3683,driver_name#3684,tier#3685,start_date#3686,home_area#3687,license
```

```
Statistics(
sizeInBytes=9.4 MiB,
rowCount=8.70E+4,
[dropoff_zone_id#3676 -> ColumnStat(
                        distinctCount: 3,
                        min: 1,
                        max: 1000,
                        nullCount: 0,
                        avgLen: 4,
                        maxLen: 4,
                        histogram: N/A,
                        version: 2),
passenger_count#3679 -> ColumnStat(distinctCount: 1, min: 1, max: 6, nullCount: 0, avgLen: 4, maxLen: 4, histogram: N/A, version: 2),
trip_status#3682 -> ColumnStat(distinctCount: 1, min: N/A, max: N/A, nullCount: 0, avgLen: 9, maxLen: 9, histogram: N/A, version: 2),
pickup_zone_id#3675 -> ColumnStat(distinctCount: 3, min: 1, max: 1000, nullCount: 0, avgLen: 4, maxLen: 4, histogram: N/A, version: 2),
trip_distance#3678 -> ColumnStat(distinctCount: 15, min: 0.1, max: 50.0, nullCount: 0, avgLen: 8, maxLen: 8, histogram: N/A, version: 2),
driver_id#3674 -> ColumnStat(distinctCount: 134, min: 1, max: 50000, nullCount: 0, avgLen: 4, maxLen: 4, histogram: N/A, version: 2),
trip_id#3672L -> ColumnStat(distinctCount: 5395, min: 0, max: 1999999, nullCount: 0, avgLen: 8, maxLen: 8, histogram: N/A, version: 2),
fare_amount#3677 -> ColumnStat(distinctCount: 40, min: 2.5, max: 150.0, nullCount: 0, avgLen: 8, maxLen: 8, histogram: N/A, version: 2),
trip_date#3673 -> ColumnStat(distinctCount: 1, min: N/A, max: N/A, nullCount: 0, avgLen: 10, maxLen: 10, histogram: N/A, version: 2),
payment_method#3680 -> ColumnStat(distinctCount: 1, min: N/A, max: N/A, nullCount: 0, avgLen: 6, maxLen: 6, histogram: N/A, version: 2
duration_seconds#3681 -> ColumnStat(distinctCount: 20, min: 300, max: 7200, nullCount: 0, avgLen: 4, maxLen: 4, histogram: N/A, version
)
```




```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- == Initial Plan ==
   HashAggregate(keys=[home_area#2861], functions=[finalmerge_sum(merge sum#2870) AS sum(fare_amount#2851)#2865
   +- Exchange hashpartitioning(home_area#2861, 200), ENSURE_REQUIREMENTS, [plan_id=3072]
      +- HashAggregate(keys=[home_area#2861], functions=[partial_sum(fare_amount#2851) AS sum#2870], output=[home_a
         +- Project [fare_amount#2851, home_area#2861]
            +- BroadcastHashJoin [driver_id#2848], [driver_id#2857], Inner, BuildRight, false, true
               :- Project [driver_id#2848, fare_amount#2851]
               :  +- Filter (((if (isnotnull(_databricks_internal_edge_computed_column_skip_row#2900)) (_databricks_internal_edge_
               :     +- FileScan parquet demo.trips.trips[trip_date#2847,driver_id#2848,fare_amount#2851,_databricks_internal_edge
               +- Exchange SinglePartition, EXECUTOR_BROADCAST, [plan_id=3067]
                  +- Project [driver_id#2857, home_area#2861]
                     +- Filter (if (isnotnull(_databricks_internal_edge_computed_column_skip_row#2901)) (_databricks_internal_edge_
                        +- FileScan parquet demo.trips.drivers[driver_id#2857,home_area#2861,_databricks_internal_edge_computed_c
```


SQL
DataFrame
Unresolved Logical plan
Logical Plan
Optimized Logical Plan
Physical plan
Cost Model
Selected Physical plan
RDD
Catalog

# Demo

# Engine Enhancements

# Catalyst

Spark SQL's query optimizer that analyzes and rewrites queries into more efficient execution plans before they run.

- Reduces query execution time.
- Automatically optimizes joins, filters, and projections.
- Improves resource utilization without manual tuning

# Tungsten

Spark's execution engine that improves performance through efficient memory management, CPU optimization, and code generation.

- Faster data processing.
- Lower memory usage.
- Reduced garbage collection overhead.
- Whole-Stage Code Generation (WSCG)

# Adaptive Query Execution (AQE)

A runtime optimization framework that modifies the execution plan based on actual data characteristics observed during execution.

- Selects more efficient join strategies dynamically.
- Mitigates data skew issues.
- Combines small partitions to improve performance and reduce task overhead.

```
spark.conf.get("spark.sql.adaptive.enabled")
```

# Best Practices for writing spark queries

# 1) Number of Partitions

# 1) Number of Partitions

- Spark will use 200 as a default value
  - spark.conf.get("spark.sql.shuffle.partitions")
  - spark.conf.set("spark.sql.shuffle.partitions", 200)

- How to chose a better default value:
  - Should be a multiple of the number of slots that you have
  - Spark is optimised for partitions sizes of roughly 128Mb
    - Total Data size(mb) \ 128(mb)
  - EG)
    - 10 GB dataset = (12 cores) x (10,000 / 128) = 84 partitions

# 1) Minimize Shuffles

# 2) Minimize Shuffles

| |
|---|
| 1) Avoid unnecessary groupBy, distinct, and joins |
| 2) Broadcast small tables |
| 3) Repartition only when needed |
| 4) Partition data by join/group keys |

# 2) Avoid Unnessisary actions