**DataSamudaya**

**S. Arun**

Contents

[1. Introduction 3](#_Toc162340200)

[1.1 Jgroups scheduler 3](#_Toc162340201)

[1.2 Standalone scheduler 4](#_Toc162340202)

[1.3 Ignite Scheduler 4](#_Toc162340203)

[1.4 Apache MESOS scheduler 5](#_Toc162340204)

[1.5 Apache Hadoop YARN scheduler 5](#_Toc162340205)

[2. Partitioning of file blocks 6](#_Toc162340206)

[2.1 Optimization of DataSamudaya Job pipeline 6](#_Toc162340207)

[3. Data Pipeline tasks to Stage conversion. 7](#_Toc162340208)

[4. Various transformations in DataSamudaya 8](#_Toc162340209)

[5. Actions in DataSamudaya 9](#_Toc162340210)

[6. Three methods of storing the intermediate results 9](#_Toc162340211)

[7. RPC for stream and MR job 9](#_Toc162340212)

[8. Zookeeper Usage in DataSamudaya 10](#_Toc162340213)

[9. SQL server for stream API 10](#_Toc162340214)

[10. Supported SQL functions and operators for Stream API 11](#_Toc162340215)

[11. SQL Server and client for Map Reduce API 12](#_Toc162340216)

[12. Supported SQL functions for Map reduce API 13](#_Toc162340217)

[13. Scripts used in starting and stopping the various server components 13](#_Toc162340218)

[14. Maven goals to build DataSamudaya project 15](#_Toc162340219)

[15. Usage of Apache calcite 15](#_Toc162340220)

# 1. Introduction

The DataSamudaya is a scalable server designed for number crunching and data manipulation in CSV and JSON formats. It follows a master-worker architecture, where the master includes the scheduler, and the workers act as containers. In the DataSamudaya architecture, the workers establish a connection with the master using JGroups, employing multicast protocol for communication. Whenever a job is submitted, the worker launches the task executor to perform the required operations. To keep track of job status, the Heartbeat mechanism is employed, which utilizes JGroups for communication. Task executors send messages to the master, providing updates about the job status. The job status can be one of the following: SUBMITTED, RUNNING, COMPLETED, or FAILED. The DataSamudaya supports multiple schedulers, including JGroups, standalone, Apache Ignite, Apache Mesos, and Hadoop YARN. Each scheduler has its own unique characteristics and functionalities, allowing users to choose the most suitable scheduler for their specific requirements and environment.

## 1.1 Jgroups scheduler

The JGroups scheduler operates autonomously, submitting jobs as stages and tasks according to the topological sorting of the Directed Acyclic Graph (DAG). The task executors are launched using containers. In cases where a stage or task depends on a parent task located in another task executor, the task executors communicate with each other using the JGroups protocol. When a parent task is in the running state, the child task in a different task executor has to wait until the parent task completes. The task executor waiting for the parent task can send a message using the WHOIS command to inquire about the parent task's status. The running task executor will receive the WHOIS request and respond with a WHOIS response message containing the current status of the parent task. To monitor all the tasks being executed by the task executors, the scheduler can send a WHOARE command. This command prompts the task executors to provide their task status information in the WHOARE response. The status information received from the WHOARE response messages of all the task executors is considered to determine whether the job has been completed. Once the final tasks are completed, the scheduler can obtain the results from these tasks.

Jgroups scheduler

Container1

Container2

TE1

TE2

TE3

TE4

In the above diagram the scheduler will determine the number of TEs to launch and communicates to the container to launch 4 Task executors. The number of TEs will decide on the various factors such as total file blocks with common block size, total number of CPUs together with all the containers and memory of each TEs to launch. Each TEs will allocate one CPUs and similar configuration of heap memory. Once all the TEs are allocated and started the scheduler will determine how to launch tasks in each of the TEs.

## 1.2 Standalone scheduler

The standalone scheduler functions as a JGroups cluster for communication, utilizing Remote Procedure Call (RPC). The central DAG scheduler is responsible for executing the DAG graph, following a top-to-bottom and left-to-right order, using the DExecutor utility. As the DAG scheduler progresses through the graph, it obtains the results of the final stage in the DAG, which typically has no successors. This allows the scheduler to determine when the tasks in the DAG have been completed and to retrieve the results. The RPC mechanism in the scheduler enables the exchange of task status information between the launched task executors and the scheduler itself. The task executors send their status updates via RPC, allowing the scheduler to track the progress of the tasks and determine whether they have been completed. This information is crucial for the scheduler to make decisions and obtain the results of the DAG execution.

Standalone scheduler

Container1

Container2

TE1

TE2

TE3

TE4

## 1.3 Ignite Scheduler

The Ignite scheduler is responsible for submitting tasks to an Ignite cluster. The job to be submitted via the Ignite scheduler is converted into stages, and these stages are further grouped into tasks. Each stage is then submitted to the Ignite cluster using the Ignite DExecutor utility. In the Ignite cluster, the text files are cached in compressed form using the LZF compressor, and they are stored in the Ignite cache cluster. The tasks are submitted using the affinity run method, which ensures that the tasks are executed on the server where the cache key and value are available. The cache keys are stored in a partitioned form on the Ignite servers, and the tasks are executed on the server where the corresponding cache key and value reside. In case one or more servers are down, backups are available on other servers to ensure data availability. The backup configuration is set up on each server, where cache keys and values are partitioned for a specific cache. Additionally, the cache can also be replicated across multiple servers in the Ignite cluster to provide redundancy and fault tolerance.

Ignite Server1

Ignite Server2

Ignite Server3

Ignite scheduler

## 1.4 Apache MESOS scheduler

The Apache Mesos scheduler is responsible for submitting jobs to the DataSamudaya scheduler, which are then converted into stages and tasks. The Mesos DataSamudaya framework first registers the scheduler and task executor framework with the Mesos master. It then submits the tasks in the form of a Directed Acyclic Graph (DAG) to the Mesos scheduler. The Mesos scheduler executes the DAG and obtains the results once all the final tasks have been completed. After job completion, the Mesos framework releases the allocated resources. The Mesos master manages resources such as CPU and memory, and it provides resource offers to the Mesos scheduler framework. If the scheduler accepts the allocated resources as offers from Mesos, the tasks are executed by Mesos executors. The final stages of the DAG are considered completed when all the tasks associated with the offers provided by Mesos have been executed and reach the final stage of the DAG.

Mesos

Worker1

Mesos

Worker2

Mesos

Worker3

Mesos Master

Mesos DataSamudaya Scheduler

## 1.5 Apache Hadoop YARN scheduler

When a job is submitted via the YARN scheduler, it involves the DataSamudaya scheduler, YARN resource manager (RM), and node manager (NM). The DataSamudaya scheduler determines the type of scheduler to be used and submits the job accordingly. The DataSamudaya scheduler converts the job into stages and tasks, which are then stored in HDFS. Upon receiving the DataSamudaya job, the YARN resource manager allocates containers. The number of containers is determined by considering factors such as the total file length, available CPUs, and memory resources. Once the containers are allocated, the application master is created and started within one of the containers. The job is then provided to the application master. The application master takes responsibility for executing the tasks, utilizing the YARN scheduler. It continually monitors the progress and status of the tasks. Once all the final stages of the job have been completed, the containers are deallocated and returned to the resource manager. The resource manager then returns the results to the DataSamudaya scheduler, signifying that the job has been completed.

Node Manager 1

Node Manager 2

Node Manager 3

YARN RM

DataSamudaya Scheduler

# 2. Partitioning of file blocks

DataSamudaya pipeline performs partitioning based on the user-defined block size. If the user specifies a true block size, they can determine parameters such as block size, maximum memory, minimum memory, and CPUs. Otherwise, these parameters are automatically set based on the length of file blocks. In HDFS, the default block size for files is 128 MB. However, for user-defined block sizes in any file system, the range can be from 1 to 128 MB. If the user-defined block size is outside this range, an error is thrown. For instance, if a file has a length of 1 GB and the block size is set to 128 MB, the file will be partitioned into approximately 8 blocks (1024/128). The number of partitions, available CPUs, and total memory determine the number of parallel executors. The task executors obtain the blocks with their respective locations. These executors then execute tasks using the blocks as input, retrieving the data from the local node of the HDFS datanode and the task executors themselves. The array of blocks can be either 1 or 2, depending on the presence of a new line in the data. If the last block of the first array ends with a new line, it remains as a single block. However, if it does not end with a new line, it extends to the second block until a new line is encountered.

1 GB file

8

128 MB

7

128 MB

6

128 MB

5

128 MB

4

128 MB

3

128 MB

2

128 MB

1

128 MB

## 2.1 Optimization of DataSamudaya Job pipeline

The DataSamudaya job's data pipeline comprises multiple tasks, also referred to as functional interfaces. These tasks are organized into stages by grouping several tasks together. For instance, if the data pipeline includes Map, filter, MapTuple, ReduceByKey, and coalesce operations, and there are 4 partitions, then there will be a total of 2 stages. Stage 1 will include the tasks Map, filter, MapTuple, and ReduceByKey, while Stage 2 will consist of the coalesce task. These stages will be executed in parallel, with a parallel stage execution count of 4.

Map

Filter

MapTuple

Coalesce

ReduceByKey

Stage1 (Map, filter, MapTuple, ReduceByKey)

Stage2 (Coalesce)

Partition1

Partition2

Partition3

Partition4

# 3. Data Pipeline tasks to Stage conversion.

In the data pipeline, prior to task execution, the task graph is initially transformed into a stage graph. Then, considering the partitions, the logical stages in the stage graph are further transformed into a physical execution graph. In the provided graph, the logical graph is converted into a physical execution graph once the partition inputs are supplied to Stage1, resulting in output that contains lists of lists. This conversion process is shared among all schedulers, with the only variation being the execution scheduler used.

Map

Filter

MapTuple

ReduceByKey

Stage1 (Map, filter, MapTuple, ReduceByKey)

File Block Partition1

File Block Partition2

File Block Partition3

File Block Partition1

Stage1 Partition1

Stage1 Partition2

Stage1 Partition3

Stage1 Partition3

Physical Execution Plan Graph

Logical graph

output

# 4. Various transformations in DataSamudaya

The transformations which produce other transformations in streamed pipeline are

**map** – which accepts function lambda has one input and one typed output

**distinct** – which provides distinct data as output

**filter** – which accepts any type as input and provides only the predicate for an input to true

**flatMap** – which accepts single input and multiple outputs

**flatMapToDouble** – which accepts single input and produces multiple output of Double type

**flatMapToLong** – which accepts single input of any type and produces multiple output of long datatype

**flatMapToTuple** – which accepts input of any type and produces multiple output of Tuple datatype

**flatMapToTuple2** – which accepts input of any type and produces multiple output of Tuple2 datatype

**union** – accepts two similar inputs and produces union of two similar datatype

**intersection** – which accepts two inputs and produces the intersected output

**joins** – inner joins of two similar datatype

**leftOuterJoin** – left outer join of two similar datatype

**maptToInt** – which accepts input of any datatype and single output of Integer datatype in streamed

**mapToPair** – which accepts single input of any datatype and produces single output of Tuple2 datatype

**peek** – which accepts and consumes and no output

**reduce** – reduces to single output for each map input

**sample** – obtains sample from the multiple inputs

**sorted** – sorts the input of any datatype and produces the sorted output either in ascending or descending order.

**coalesce** – confines multiple partition data to specific partition data

**reduceByKey** – reduces each partitioned key value pair data to single key value pair data

**groupByKey** – groups the value based on the key in tuple2

**countByKey** – counts the records based on the key in a partition

**countByValue** – counts the records based on the values in a partition

**cogroup** – combines the values from input1 and input2 based on the key in tuple2

**keyBy** – choose the key based on the user function using the values for tuple2

# 5. Actions in DataSamudaya

The various actions which trigger execution of the tasks are

**count** – counts the number of records

**collect** – triggers the execution of task and produces the output in list of lists which has output for all the partitions.

**forEach** – used for traversing the records

**saveAsTextFile** – dump the output of the DataSamudaya job into the text file in hdfs

# 6. Three methods of storing the intermediate results

**INMEMORY** – stores the intermediate results in memory

**DISK** – stores the intermediate results in disk

**INMEMORYDISK** – stores the intermediate results in memory and spills over disk when memory has reached above 80%

# 7. RPC for stream and MR job

On each machine, the node launcher is responsible for allocating task executors. After the allocation of task executors, they are launched once the blocks required for their execution are allocated. The tasks are then executed by the task executors, which obtain the necessary data from the datanode that is local to them. The scheduler plays a crucial role in the execution process. It assigns tasks to the task executor through rpc (Remote Procedure Call) calls, enabling communication between different components of the system. The scheduler also receives the response from the task executor, which includes the status of task execution. Once the results of the tasks are completed, the node launcher takes care of deallocating the task executors, ensuring efficient utilization of resources.

Task Executor 1

Job

Stream Job Scheduler

Task Executor 2

Node2

Node1

The stream job scheduler operates by converting the Job into stages, where each stage consists of multiple tasks. These stages are distributed to all task schedulers for task execution. Once a task is completed, the task executor sends the results back for each individual task. The task scheduler then proceeds to assign the next available tasks to the task executor, ensuring efficient utilization of resources. Finally, when the final task is completed, the task scheduler retrieves the output produced by the tasks.

# 8. Zookeeper Usage in DataSamudaya

The purpose of zookeeper in DataSamudaya is the node launcher, registers itself to the zookeeper with initial memory, CPU, and disk information. The scheduler which is elected as a leader use this node launchers to allocate the port and starts the executor when a job is submitted. This method of node manager registration and usage is called server grouping. The jobs and tasks also get registered in zookeeper and used across nodes and schedulers and when a job is completed the job information from the zookeeper is deleted.

Stream Job Scheduler/

Standalone Sc

Zookeeper

Node1

Node2

NodeN

# 9. SQL server for stream API

The streaming SQL server is started either in standalone or stream scheduler script. The SQL client connects the SQL server with various parameters like user, number of executors or also called containers, CPUs, memory, and worker mode for initialization. The SQL server can create the table, perform SQL select for querying the table, does join, etc. The various worker mode that supports when SQL client registers to the SQL server are standalone, ignite and JGroups. Standalone and JGroups mode starts with the executors given in specification when SQL client connects to the server. The table is created in h2 database and for each table one extra column named hdfslocation with default value for the path needs to be created along with necessary columns. The various queries executed in SQL client are

**setmode** – sets the worker mode with one parameter. Allowed worker mode values are (standalone, ignite, jgroups).

**getmode** - gets the current worker mode.

**use** – selects the default DB with one parameter in the command.

**getdb** - gets the current DB in use.

Some SQL commands that supported are

**create** – creates table with columns.

**alter** – alters the table.

**drop** – drops the table

**show** – shows the tables.

**describe** – describes the table.

**select** – executes the select query

Once the SQL queries are executed the results are shown in the client.

SQL Client

SQL Server

Scheduler

Executor1

Executor2

# 10. Supported SQL functions and operators for Stream API

The various supported SQL column functions are given below.

**abs** – returns absolute value of a number passed as a parameter to abs function.

**length** – returns length of a string passed as a parameter to length function.

**round** – round the decimal number passed as a parameter to round function.

**ceil** – ceils the decimal number passed as a parameter to round function.

**floor** – calls the floor function in java math package passing decimal number as a parameter to floor function.

**pow** – calculate the power of number with exponent value passed as parameters to the java math pow function.

**sqrt** – calculates the sqrt of a number passed as a parameter to the java math sqrt function.

**exp** – calculates the exponential of a number passed as a parameter to the java exponential function.

**loge** – calculates the log value of a number with base e which is passed as a parameter to the java math log function.

**lowercase** – converts a string to lowercase which is passed as a parameter to lowercase function.

**uppercase** – converts a string to uppercase which is passed as a parameter to uppercase function.

**base64encode** – encodes the string to base64 encoded string which is passed as a parameter to java bas64 encoder library.

**bas64decode** – decodes the base64 encoded string to original value which is passed as a parameter to java base64 decoder library.

**normalizespaces** – removes the multiple whitespaces before, after and in between the given string as a parameter to normalizespaces function.

**substring** – returns substring of a given string passing string, starting position and length of the substring.

The various operator supported are **addition** (+), **subtraction** (-), **multiplication** (\*) and **division** (/).

The various aggregated functions supported by streaming SQL are.

**count** - counts the total number of records.

**sum** – calculates the sum of expression given as parameter to the sum function.

**avg** - calculates the average of expression given as parameter to the sum function.

**grpconcat** – concatenates the column using group concat function.

**min** - finds the min of given column by passing it as a parameter to min function.

**max** - finds the max of given column by passing it as a parameter to max function.

# 11. SQL Server and client for Map Reduce API

The Map Reduce SQL server and client works same as the streaming api except it executes sql query in map reduce mode.

SQL MR Client

SQL MR Server

MR

Scheduler

Executor1

Executor2

# 12. Supported SQL functions for Map reduce API

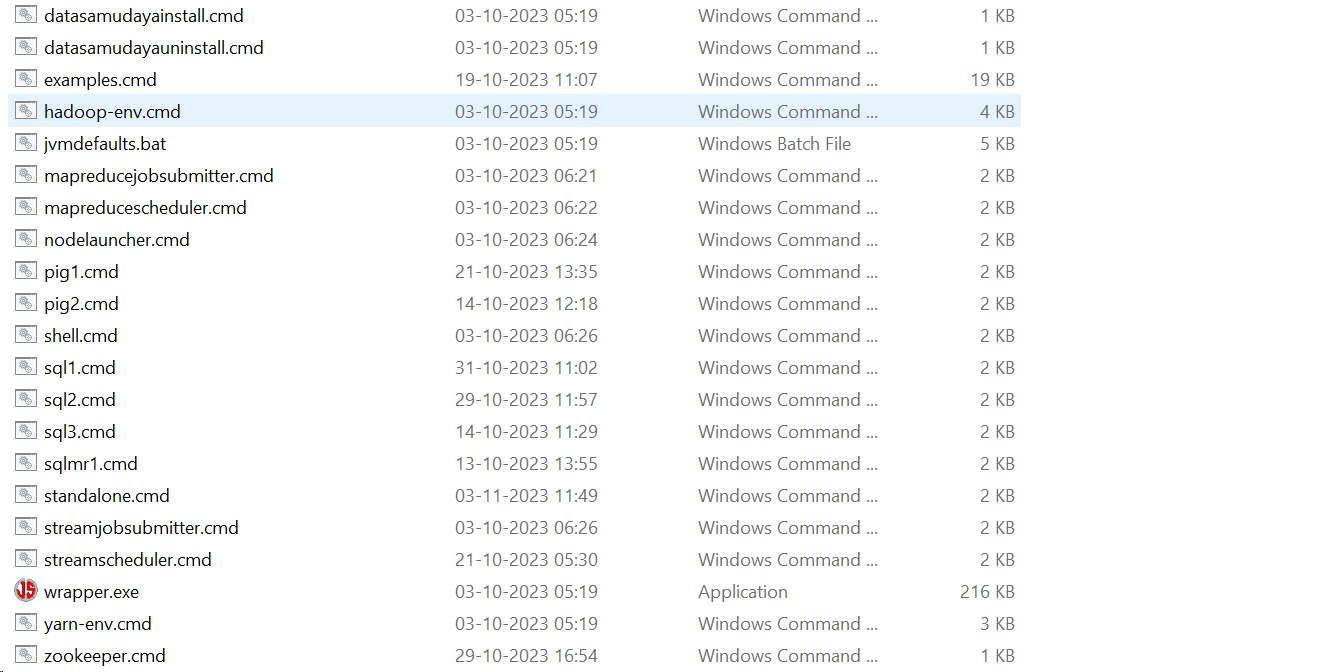
The various aggregated functions supported by Map Reduce SQL are.  
**count** - counts the total number of records.

**sum** – calculates the sum of expression given as parameter to the sum function.

**min** - finds the min of given column by passing it as a parameter to min function.

**max** - finds the max of given column by passing it as a parameter to max function.

# 13. Scripts used in starting and stopping the various server components



The windows **cmd** scripts are available in bin folder of datasamudaya home folder. The description of each script is below,

**examples.cmd** – executes the examples for flight dataset.

**mapreducejobsubmitter.cmd** – submits the MapReduce jar to MapReduce scheduler. An example of submitting map reduce jar is below.

**call mapreducejobsubmitter.cmd -jar ../examples/examples-2.0.jar -arguments "com.github.datasamudaya.mr.examples.join.MrJobArrivalDelayNormal /airline1989 /carriers /examplesdatasamudaya 128 10"**

**mapreducescheduler.cmd** – starts the MapReduce scheduler along with MapReduce SQL server which accepts the SQL commands.

**nodelauncher.cmd** – starts the node to allocate port and creates task executor when a job is submitted.

**pig1.cmd** – starts the pig client for streaming scheduler.

**sql1.cmd** – starts the SQL client with some parameters such as user, number of task executors, CPU per task executor, memory per task executor and worker mode for streaming based scheduler.

**sqlmr1.cmd** – starts the SQL client with user as parameter for allocation of resources such as cpu, memory etc for map reduce mode.

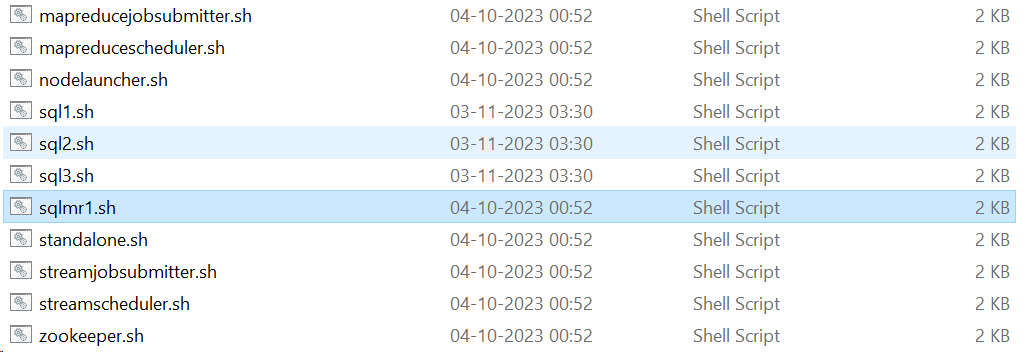
**standalone.cmd** – starts the streaming scheduler, map reduce scheduler and node launcher in a single JVM.

**streamjobsubmitter.cmd** – submits the streaming job implementation to the streaming scheduler which is executed on the task executors.

**streamscheduler.cmd** – starts the streaming scheduler to convert the job to stages to tasks to be executed on the task executor.

**zookeeper.cmd** – starts the zookeeper server. The nodes getting registered to the zookeeper, where the scheduler launches the task executor on the nodes registered in zookeeper. The scheduler registers itself to the zookeeper for electing the master.

The linux scripts are available in **sbin** folder,



**mapreducejobsubmitter.sh** – submits the MapReduce jar to MapReduce scheduler like map reduce job submitter in windows.

**mapreducescheduler.sh** – starts the map reduce scheduler in linux.

**nodelauncher.sh** – starts the nodes registered to the zookeeper.

**sql1.sh** – starts the SQL client for streaming based SQL query execution in linux environment.

**sqlmr1.sh** – starts the SQL client for map reduce based SQL query execution in linux environment.

**standalone.sh** – starts the streaming scheduler, map reduce scheduler and node launcher in linux environment.

**streamjobsubmitter.sh** – submit the streaming based job to the streaming scheduler in linux environment.

**streamscheduler.sh** – starts the streaming based scheduler also starts the SQL server which executes the queries.

**zookeeper.sh** – starts the zookeeper in linux as in windows environment.

# 14. Maven goals to build DataSamudaya project

mvn -Dmaven.antrun.skip=true -Dmaven.test.skip.exec=true -DskipMavenParsing=true -Pmodules clean install package assembly:assembly

The above maven command with options builds two jars in the project target folder

1. **datasamudaya-[version]-jdk17.tar.gz**
2. **datasamudaya-[version]-jdk17.zip**

The version is the current build version of the project.

# 15. Usage of Apache calcite

The Apache calcite libraries are used for validating the SQL and the sql plan sent to volcano optimizer for generating the optimized query plan. The various core rules used are

1. **FILTER\_TO\_CALC** - Rule that converts a LogicalFilter to a LogicalCalc
2. **PROJECT\_TO\_CALC** - Rule that converts a LogicalProject to a LogicalCalc.
3. **FILTER\_MERGE** - Rule that combines two LogicalFilters.
4. **FILTER\_CALC\_MERGE** - Rule that merges a Filter and a LogicalCalc. The result is a LogicalCalc whose filter condition is the logical AND of the two.
5. **PROJECT\_CALC\_MERGE** - Rule that merges a LogicalProject and a LogicalCalc.
6. **AGGREGATE\_PROJECT\_MERGE** - Rule that recognizes an Aggregate on top of a Project and if possible, aggregates through the Project or removes the Project.
7. **PROJECT\_FILTER\_VALUES\_MERGE** - Rule that merges a Project on top of a Filter onto an underlying org.apache.calcite.rel.logical.LogicalValues, resulting in a Values with different columns and potentially fewer rows.

The examples of optimized query plan from the logical plan by applying the above rules are,

**Query 1** - **select \* from airline**

**Logical Plan 1** –

**LogicalProject**(AirlineYear=[$0], MonthOfYear=[$1], DayofMonth=[$2], DayOfWeek=[$3], DepTime=[$4], CRSDepTime=[$5], ArrTime=[$6], CRSArrTime=[$7], UniqueCarrier=[$8], FlightNum=[$9], TailNum=[$10], ActualElapsedTime=[$11], CRSElapsedTime=[$12], AirTime=[$13], ArrDelay=[$14], DepDelay=[$15], Origin=[$16], Dest=[$17], Distance=[$18], TaxiIn=[$19], TaxiOut=[$20], Cancelled=[$21], CancellationCode=[$22], Diverted=[$23], CarrierDelay=[$24], WeatherDelay=[$25], NASDelay=[$26], SecurityDelay=[$27], LateAircraftDelay=[$28]): rowcount = 60000.0, cumulative cost = 120000.0, id = 2

**LogicalTableScan**(table=[[airschema, airline]]): rowcount = 60000.0, cumulative cost = 60000.0, id = 1

**Optimized Plan 1** –

Node ID: 12

Node Description: rel#12:**EnumerableProject**.ENUMERABLE(input=EnumerableTableScan#8,inputs=0..28)

Node ID: 8

Node Description: rel#8:**EnumerableTableScan**.ENUMERABLE(table=[airschema, airline])

**Query 2 - select \* from airline where airlineyear = 2006**

**Logical Plan 2 -**

**LogicalProject**(AirlineYear=[$0], MonthOfYear=[$1], DayofMonth=[$2], DayOfWeek=[$3], DepTime=[$4], CRSDepTime=[$5], ArrTime=[$6], CRSArrTime=[$7], UniqueCarrier=[$8], FlightNum=[$9], TailNum=[$10], ActualElapsedTime=[$11], CRSElapsedTime=[$12], AirTime=[$13], ArrDelay=[$14], DepDelay=[$15], Origin=[$16], Dest=[$17], Distance=[$18], TaxiIn=[$19], TaxiOut=[$20], Cancelled=[$21], CancellationCode=[$22], Diverted=[$23], CarrierDelay=[$24], WeatherDelay=[$25], NASDelay=[$26], SecurityDelay=[$27], LateAircraftDelay=[$28]): rowcount = 9000.0, cumulative cost = 78000.0, id = 3

**LogicalFilter**(condition=[=($0, 2006)]): rowcount = 9000.0, cumulative cost = 69000.0, id = 2

**LogicalTableScan**(table=[[airschema, airline]]): rowcount = 60000.0, cumulative cost = 60000.0, id = 1

**Optimized Plan 2** –

Node ID: 19

Node Description: rel#19:**EnumerableFilter**.ENUMERABLE(input=EnumerableTableScan#11,condition==($0, 2006))

Node ID: 11

Node Description: rel#11:**EnumerableTableScan**.ENUMERABLE(table=[airschema, airline])

**Query 3 - select count(\*) from airline**

**Logical Plan 3 –**

**LogicalAggregate**(group=[{}], EXPR$0=[COUNT()]): rowcount = 1.0, cumulative cost = 120001.125, id = 3

**LogicalProject**($f0=[0]): rowcount = 60000.0, cumulative cost = 120000.0, id = 2

**LogicalTableScan**(table=[[airschema, airline]]): rowcount = 60000.0, cumulative cost = 60000.0, id = 1

**Optimized Plan 3** –

Node ID: 21

Node Description: rel#21:**EnumerableAggregate**.ENUMERABLE(input=EnumerableTableScan#11,group={},EXPR$0=COUNT())

Node ID: 11

Node Description: rel#11:**EnumerableTableScan**.ENUMERABLE(table=[airschema, airline])

**Query 4 - select count(\*) from airline where airlineyear=2009**

**Logical Plan 4 –**

**LogicalAggregate**(group=[{}], EXPR$0=[COUNT()]): rowcount = 1.0, cumulative cost = 78001.125, id = 4

**LogicalProject**($f0=[0]): rowcount = 9000.0, cumulative cost = 78000.0, id = 3

**LogicalFilter**(condition=[=($0, 2009)]): rowcount = 9000.0, cumulative cost = 69000.0, id = 2

**LogicalTableScan**(table=[[airschema, airline]]): rowcount = 60000.0, cumulative cost = 60000.0, id = 1

**Optimized Plan 3** –

Node ID: 29

Node Description: rel#29:**EnumerableAggregate**.ENUMERABLE(input=EnumerableFilter#28,group={},EXPR$0=COUNT())

Node ID: 28

Node Description: rel#28:**EnumerableFilter**.ENUMERABLE(input=EnumerableTableScan#14,condition==($0, 2009))

Node ID: 14

Node Description: rel#14:**EnumerableTableScan**.ENUMERABLE(table=[airschema, airline])

# 16. Sequence Diagram Sql Query Engine

