Proof of DAG Whitepaper

A Highly Scalable decentralized computing protocol for high throughput of transactions processing

[Technical whitepaper — release 2 — revision 2]

June, 2023

By Larry Liu

http://roturalabs.com

1. Introduction

Blockchain technology needs breakthrough. But present blockchain architectures all suffer from a number of issues not least practical means of performance and scalability, which prevents billions of users from adopting cryptos.

The focus of this paper is to formalize the scale out architecture based on new decentralized computing design to allow the network computing power grow linearly.

2. The challenges

The current design of all blockchains is to have all transactions syned to one node and verify those transactions on one node. After verification, the node will broadcast the blockchain throughout the network. In this design, the bottleneck is the network speed to sync the transactions and the computing power on that node to verify the transactions and forge the block. This design is not scalable at all.

In the whitepaper of PoE(Proof of Events), the new protocol of capturing transactions is designed for faster decentralized computing. We need a new architecture to be able to handle the PoE producer data as a consumer. The new design has to be able to scale with PoE.

What is the best algorithm for the scalable computing? The research comes to the DAG. Before understanding the theory about DAG, let's talk about a real life scenario (day to day).

What tasks do I need to complete in order to go to office?

- Wake up.
- Leave the bed.
- Get fresh.

- Take breakfast.
- Get ready.
- Go to office (Walk/Drive, etc.)

If I would need to represent these as a sequence of stages, here’s how it will look:

> *Wake up → Leave the bed → Get fresh → Take breakfast → Get ready → Drive to office*

The above example can be thought of as a DAG (Directed Acyclic Graph) where:

- The sequence of events can be related to different **Stages** of an action which is, “Going to office”.
- Each stage depends on the completion of the previous stage, **in ideal case**. (Well, nothing stops you to wake up and go to office directly. Get a breakfast there, then get fresh and then get ready in the office’s changing room, LOL !!!).
- There is **no cycle** in this uni-directional sequence of stages.

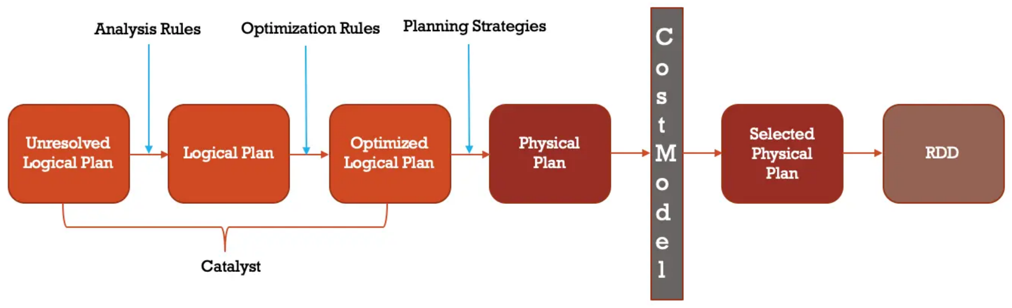
3. PoDAG

PoDAG is a protocol which designs the track of operations applied on RDD from logical to physical. PoDAG uses RDD lineage play a significant role in Rotura’s fault-tolerant and distributed data processing architecture.

3.1 PoDAG Execution Plan

This is the exeuction plan for a dapp:

1. Generate an **Unresolved Logical Plan***.
2. Then it will apply Analysis rules and Schema catalog to convert into a **Resolved Logical Plan.**
3. The Optimization rules will finally create an **Optimized Logical Plan**.
4. The Optimized Logical plan will be transformed to multiple **Physical Plans**** by applying set of Planning strategies.
5. Finally **Cost Model** will choose the **optimal Physical Plan** which will be converted into an RDD.

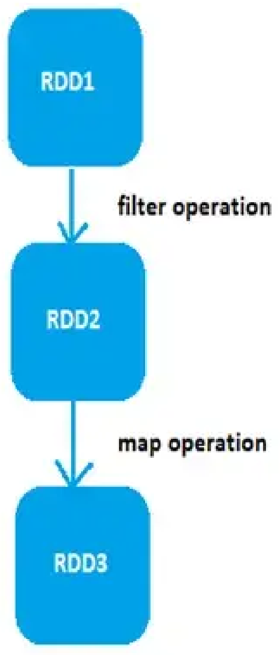


The main reason to convert an Unresolved Logical Plan into an Optimized Physical Plan is to minimize the response time of the query execution.

** A logical plan describes the computation on datasets without defining how to conduct the computation.*

*** A Physical Plan describes computation on datasets with specific definitions on how to conduct the computation. Basically Physical plan is executable.*

3.2 RDD Lineage VS DAG



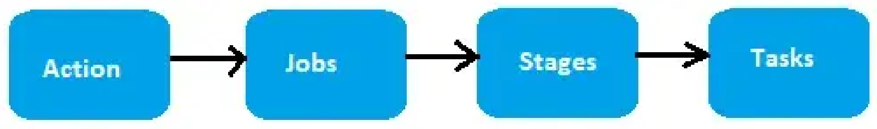
3.2.1 RDD Lineage

Lineage defines the transformation steps which are used to generate an RDD. Basically it is a Logical Plan which tells us how to create an RDD by applying what transformation. The picture shows RDD2 is getting created by applying a filter operation on RDD1. Similarly when we use map operation on RDD2 then RDD3 will be generated.

This information is very useful from dependency point of view and will help to handle any failure, because it knows from the Lineage graph what all transformations are needed to generate the missing RDDs (during any failure).

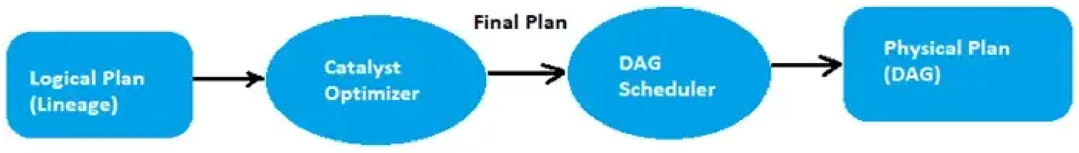
One more point to observe: Lineage doesn't have any action defined. It's a pure transformation tracker.

Before we get into DAG, a quick understanding: whenever encounters an Action it creates a JOB. Based on the transformation types (will discuss later in detail) Stages are getting generated. Each stage contains Tasks to execute.



3.2.2 DAG

DAG (Directed Acyclic Graph) on the other hand is the acyclic graph of Stages. For easy understanding we can say, after applying multiple transformations on RDD when an Action is encountered, the Logical Plan (Lineage) is getting submitted into Catalyst Optimizer.

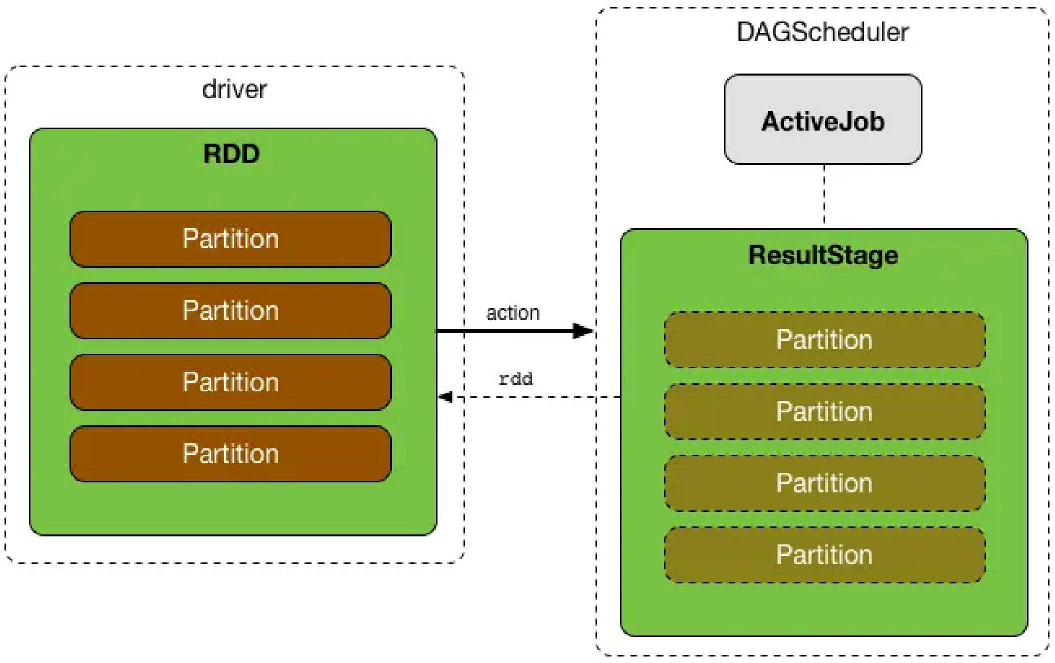


Catalyst Optimizer will apply internal logic and submit the final plan to DAG Scheduler. It's DAG Scheduler responsibility to generate Physical Plan (DAG) and splits the whole execution process into Stages.

3.3 PoDAG Scheduler

`DAGScheduler` is the scheduling layer that implements **stage-oriented scheduling** using Jobs and Stages.

`DAGScheduler` transforms a **logical execution plan** (RDD lineage of dependencies built using RDD transformations) to a **physical execution plan** (using stages).



3.4 Fault Tolerance: Self recovery property

PoDAG fault tolerance property means RDD, has a capability of handling if any loss occurs. It can recover the failure itself, here fault refers to failure. If any bug or loss found, RDD has the capability to recover the loss.

We need a redundant element to redeem the lost data. *Redundant data* plays important role in a self-recovery process. Ultimately, we can recover lost data by redundant data. In addition, we may apply different transformations on RDDs. That creates a logical execution plan for all tasks executed. This logical execution plan is also popular as lineage graph.
In the process, we may lose any RDD as if any fault arises in a machine. By applying the same computation on that node, we can recover our same dataset again. We can apply the same computations by using lineage graph.
If any of the nodes of processing data gets crashed, that results in a fault in a cluster. In other words, RDD is logically partitioned and each node is operating on a partition at any point in time.
Operations which are being performed is a series of scala functions. Those operations are being executed on that partition of RDD. This series of operations are merged together and create a DAG, it refers to Directed Acyclic Graph. That means DAG keeps track of operations performed.
If any node crashes in the middle of an operation, the cluster manager finds out that node. Then, it tries to assign another node to continue the processing at the same place. This node will operate on the same partition of RDD and series of operations. Due to this new node, there is effectively no data loss. Meanwhile, that new node continues the processing smoothly.
There are basically two fault tolerant significance. They are:

- Since RDDs are *immutable* in nature. Hence, to create each RDD we need to memorize the lineage of operations. Thus, it might be used on fault–tolerant input dataset for its creation purpose.
- If any partition of an RDD may be lost due to worker node failure, then using the lineage we can recompute it. This is possible from an original fault tolerant dataset.

To meet fault tolerance for all the RDDs, whole data is being replicated among many nodes in the cluster.
There are two types of data that demand to be recovered in the event of failure:

- Data received and replicated
- Data received but buffered for replication.

To understand better, let's study them in detail:

— Data received and replicated

In this process, data gets replicated on one of the other nodes. So that if any failure occurs we can retrieve the data for further use.

— Data received but buffered for replication

In this process, there is no replicated data. The only way to recover it, by getting it again from the source.

4. Conclusion

PoDAG is a directed acyclic graph that represents the logical execution plan of a dapp.

In summary, let's understand what is PoDAG. On decomposing its name:

- Directed — Means which is directly connected from one node to another. This creates a sequence i.e. each node is in linkage from earlier to later in the appropriate sequence.
- Acyclic — Defines that there is no cycle or loop available. Once a transformation takes place it cannot returns to its earlier position.
- Graph — From graph theory, it is a combination of *vertices* and *edges*. Those pattern of connections together in a sequence is the graph.