Automatic Plan Visualization in Conbench DSPRO Project Report

Benjamin S, Steinbüchel, 396953 Luca Blankenburg, 456835
Leo Raasch, 481987

August 5, 2025

Table of Contents

1 Introduction

Links to GitHub repositories:

- Project fork of NebulaStream
- Project fork of Conbench

1.1 Motivation

In modern database systems, it is vital to understand how queries are executed in order to optimize performance, debug issues, and most importantly, gain insight into internal system behavior. Tools such as SQL's EXPLAIN function offer this transparency by breaking down queries into their execution plans, showing their distinct components, interplay, and data flow. However, for the newly published open source multi-modal edge-application streaming engine NebulaStream, no such automated visual explanation method was yet available.

Although Conbench was already being used to manage NebulaStream benchmarks, it lacked the capability to innately visualize the query plans of the benchmarking queries. This made inspection into how queries were parsed, optimized, and executed, particularly in the context of streaming data flows with multiple processing stages, unclear and required in-depth prior understanding of the project. Furthermore, detecting the reason for benchmark improvements or deteriorations was impossible without observing the latest commits in detail.

Our goal was to close this functional gap by extending Conbench and the NebulaStream source code with a dedicated method for extracting, storing, and visually rendering both logical and pipeline query plans. Inspired by the role of EXPLAIN in traditional SQL systems, this new visualization feature empowers users to better understand internal data flows, operator behavior, and optimization strategies within NebulaStream, ultimately enabling more effective debugging and system tuning whilst running system tests.

1.2 Outset

Our initial understanding was that, in order to display the specific query plans, the server first needs to receive, validate, and then store the query plans. Query plans can be serialized and then sent to the front-end where they will be displayed for visual information extraction.

Specifically, we noticed that NebulaStream sends its benchmarking data in a JSON format to Conbench via the benchmark.py script. To avoid having to bother with a GitHub Workflow, we use this preexisting infrastructure and add the serialized query plans to the JSON file that is sent. Thus, with the –visualizePlan flag set for the systest executable program, in addition to the benchmark result, the query plan will be sent directly to the server.

2 Inital Idea

After our introduction to NebulaStream, our first approach was to directly edit the Conbench files where it was needed. Therefore, we added the query plan validation code to the benchmark result validation code, as well as doing the same for the storage and serialization code. Although our code was working, only later on further into the project did we realize that the delete button disappeared from the website (https://.../benchmark-result/...). This went unnoticed for several days. It was quite tricky to debug this issue, as everything was working seemingly fine, the query plans and the benchmark result information were properly displayed, and all other websites worked as expected. The root cause of this issue was a flawed serialization of the query plan (due to a type error), which was simply stacked onto the benchmark result serialization. This coupled serialization also meant that there was no possibility to decouple the query plan from the benchmark result data. As a result, it could not be guaranteed that the native Conbench functionality remained unimpeded.

As a result of this issue and further reasons, we decided to split the query plan logic from the native Conbench logic as much as possible. By keeping these elements separate, we aim to keep the native Conbench functionality unaffected by issues that may arise from our code in the future. Furthermore, by separating not only the logical aspects, but also the file locations, we are able to keep a clean project structure and a legible overview.

Continuing further from this idea of code separation, we were able to build the necessary fundamental structures presented below inside of the System Architecture Overview. Hereby, each module is individually presented, including its approaches in development and resulting functionalities, which work towards the project goal as a coherent unit.

3 System Architecture Overview

https://github.com/DSPRO-Query-Plan-Visualization/Conbench-Query-Plan-Visualization/tree/main/queryplan

Folder structure:

queryplan/api/__init__.py
queryplan/api/results.py

queryplan/entities/__init__.py
queryplan/entities/validation.py
queryplan/entities/query_plan.py

queryplan/templates/queryplan.html

queryplan/README.md

queryplan/__init__.py

The queryplan folder is the collection of all query plan related code and files which serves to keep the query plan code manageable and easy to overview.

Though, there are a few exceptions:

- Dockerfile includes the queryplan module so it's available at runtime
- conbench/__init__.py registers the queryplan/templates/queryplan.html template
- conbench/api/__init__.py imports queryplan/api/results.py for query plan API functionality and queryplan/entities/validation.py. as a marshmallow schema extension
- /migration/versions/e15e8b226491_benchmark_id_ondelete_cascade_query_.py adds cascade behavior on benchmark_id for query plan records
- benchmark-result.html includes the query plan UI via the shared template

While the query plan code is mostly separated from the conbench code, we still utilize the same structure for consistency.

- queryplan/api/results.py handles API requests (GET, POST).
- queryplan/entities/query_plan.py defines and creates the relevant tables and relationships.
- queryplan/entities/validation.py handles request schema validation using marshmallow.

3.1 Front-End

Let's start with the Front-End.

3.1.1 Goal and Requirements

To reiterate, our goal is to create an automatic visualization of query plans for NebulaStream. Through the development of this project, we found it feasible to focus on two specific plans, the logical- and later pipeline query plan as our end results.

Hereby, we identified some key requirements for the Front-End visualization of the project. From the beginning, it was clear that the visualization needed to integrate into the existing benchmarking system, in this case Conbench. It had to be capable of displaying query plans as visual graphs, be expandable for future features or visual improvements, and be based on pre-existing tested technologies rather than creating a ground-up rendering engine as we are restricted in time and as to not "reinvent the wheel." We wanted to establish a base functionality early on and incrementally expand upon it for these two query plan types. In terms of layout, we wanted the flow of data to be immediately clear. This meant establishing clear parent-child relationships and having well-aligned arrows that clearly indicate the direction of execution. Lastly, the end result should be highly configurable in nature.

These initial goals guided us with the choice of libraries and shaped the development process early on. In hindsight, a number of additions were made to further improve the UI experience and provide better visual clarity.

3.1.2 Development Stages

Initially, we started with the commonly used D3-hierarchical Java Script data library as we had tried to complete a quick working example inspired by the demonstrative image given to us in the on-boarding presentation. However, we quickly realized that the data structures we were working with are not always strictly hierarchical in nature or chronologically ordered. They often consist of multiple layers, like the pipeline plan, and may surface nodes with multiple children, as seen in the possibility of having Join- or Union-nodes with more than one child. Based on this realization, we switched libraries and adopted one that better suited directed acyclic graphs (DAGs). This library was chosen after evaluating the resulting data structure of the query plans to DAGs. This change was supported by the example presented inside of the D3-Dag library, the Sugiyama implementation created by Erik Brinkman.

The D3-Dag library matched our key requirement of being a preexisting and tested interface for us to use. Furthermore, it acts as a strong basis for us to create an informative visual illustration of query plans whilst keeping a small size that offers multiple options in the layout presentation.

As we progressed, from having established our first working solution with the logical plans, we found that pipeline plans introduced another layer of complexity: they required support for nested nodes, since a pipeline contains multiple physical operators. Our first approach was to represent this feature by embedding a query plan inside another query plan as per our approach of extending upon features that were already working. However, this quickly proved to be inadequate. Grouping these nodes into a larger node was not possible as the library is not suited for layering a graph inside of a graph. To be able to present a working solution for the mid-term presentation, we simply decided to present the pipeline plans as a rectangular node to which its belonging operators are added as circular nodes. As this solution did not adhere to our requirement of having the data flow be immediately clear, we recognized the need for a more expressive representation that can offer us not only a DAG visualization, but also nested hierarchically aligned structures to clearly show the data flow through these layered pipeline plans including their nodes and connections across the individual stages.

The resulting solution was for us to move away from the limitations of external graph libraries and instead build the rendering structure manually. That meant creating the entire visualization using plain DOM elements. By doing it in this way, we had full control over how everything looked and behaved. We could style and align the layout exactly how we wanted. So, pipelines were grouped by levels and displayed as .tree-row divs, with each pipeline being its own container. Inside that container, we added all the operators for that pipeline, one below the other. This setup made it easy to show the flow from source to sink in a top-down manner.

A issue that came up was the drawing of edges between query plan nodes. For this, we used the LeaderLine library. It worked well in the sense that it could draw SVG arrows between two HTML elements as well as being extensible in its features, as shapes and sizes can be easily configured. But it also had a downside. The lines are created separately from the nodes, and they don't really "belong" to any group or container. You just pass in two DOM elements, and the library connects them to their bottom or top centers.

Because of that, we ran into an issue: when nodes moved outside the viewport during zooming or panning, the arrows were still being drawn, sometimes even when the nodes weren't visible anymore. To fix this, we had to write our own check function constrainLeaderLineSVGs() to see if both ends of a line were still inside of the visible area. By using the library .show() and .hide() methods that come with LeaderLine we were able to control when the arrows appear or disappear. Without this, the interface got messy, especially with larger graphs. Arrows would be visualized outside of the viewport. This workaround became necessary to keep everything clear and visually aligned.

3.1.3 Feature Overview

The Front-End consists of a main feature set that allows us to display both logical and pipeline query plans. To keep the Conbench UI clean but still show both plan types, we added a button that switches between the pipeline and logical view. When you click it, the current graph gets removed, and the other one is loaded. The label on the button also updates so you know which one you're looking at. Both plans are already passed in from NebulaStream as part of the benchmark data, so we don't need to reload anything.

Each plan gets rendered by its own function; renderPipelinePlan() and renderLogicalPlan(). They both work similarly: we first find the last node in the plan (also called the sink node), and then we go backwards through the structure connecting the next nodes. Based on how far each node is from the sink, we assign it to a level. That is how we decide what goes into which row. The rows are then rendered from top to bottom, meaning sources are at the top and the sinks at the bottom. All nodes are created using regular DOM elements. For pipeline plans, each pipeline is its own container, and inside of it we add the operator nodes. These operators get sorted based on their inputs and outputs so that they show up in the correct order. For logical plans, each node has a type and label, and both are displayed in the container.

The arrows between nodes are drawn with the LeaderLine library. It works well in general, but we did face some issues. The arrows aren't part of the DOM, and the library doesn't keep track of visibility or layout. That means if you zoom, pan, or move things around, you must manually tell the arrows to reposition. On top of that, if a node gets pushed out of the visible area, the line will still show. So, we wrote our own check that goes through each arrow and hides it if its connecting location at the start or end node is no longer inside the viewport as well as updating its location with a separate function.

Zooming and panning are built in. You can scroll to zoom, drag with the mouse to move things, or use the arrow keys to propagate the graph inside of the viewport, and there are keyboard shortcuts for zoom and reset. We also added a small zoom indicator inside of the viewport. It is a helpful indicator to show the current zoom percentage so that one may have a sense of scale when viewing large graphs. Lastly, the graph automatically scales and centers itself to fit the viewport when the page is initially loaded.

To summarize, we used DOM elements, customizing them with CSS and employing the LeaderLine library for the arrows to create the final result. In the end, that gave us full control over structure and styling and made it easier to tweak things along the way or add features later if needed.

3.2 Conbench: Augmentations

3.2.1 queryplan.html

The UI code in queryplan/templates/queryplan.html handles the rendering of the graph. To include the visualization in a page, use:

```
{% include "queryplan.html" %}
```

Make sure a benchmark.id is available in the page context. Without it, no query plan will load. See benchmark-result.html as an example. The detailed explanation of the graph code can be found in the next sections. Hereby, it is important to note that once the page that includes the queryplan.html file is loaded, a fetch request will be automatically triggered.

3.2.2 API

```
GET /api/queryplan/<benchmark_id>
```

This API endpoint returns both the logical and pipeline plan structures for the given benchmark.

Here we can see the desired logic split in action. Rather than using the combined serializer (queryplan + benchmarkresult) we request the query plan data directly. So instead of a faulty combined serializer affecting the whole page, only the query plan part would be affected, keeping the native conbench functionality intact.

Another thing to note is, some pages have different names for the benchmark.id (as in /compare/benchmark-results/abc...xyz - baseline.id and contender.id). If you would like to display the query plan you can use Jinja before including the queryplan.html:

```
{% set benchmark = {"id":  othername.id} %}
```

Meanwhile the fetch requests are handled by queryplan/api/results.py. The GET request functions the same way as the conbench/api/results.py equivalent, but the POST request is handled differently. Since we cannot register another endpoint at the same address (/api/benchmarks/), the only option left is to hook into the transaction using an event-listener. Of course, editing the benchmark result create() function directly is still an option and worked fine before the split, but since we want to keep the code and logic as separate as possible we went with the event-listener option. In the event-listener function, we can then access the query plan data and create the tables. Since the event fires after a successful benchmark result creation, the query plan data will already have been validated, the login requirements checked and the benchmark result entry created.

At the time of writing this, only GET and POST requests are supported (with POST being handled mostly in queryplan/entities/). This aligns with our goal of sending the data to and viewing the data on the server, with dangling query plans not making much sense. Though if a delete or put option is needed in the future, it should be easy to adapt the query plan api.

3.2.3 Tables

The query plan table definition, insertion, and serialization happen in queryplan/entities/query_plan.py. The current relevant table structure looks like this:

```
benchmark_result <- logical_query_plan <- [ logical_query_plan_nodes ]

                  benchmark_result <- pipeline_plan
pipeline_plan <- [ pipeline_node ]  <- operator_plan <- [ operator_node ]
```

The drawn pipeline plan reads like this: The pipeline plan is connected to the benchmark result and holds multiple pipeline nodes. Each pipeline node has a connected operator plan which in turn holds multiple operator nodes. The data (like "id","label",etc.) is stored in the nodes, and the plan tables act as an intermediate, collecting the nodes with the possibility of adding metadata in the future. Lastly, the serialization copies conbench's serialization structure, giving the option to serialize either one or many query plans. The serialized structure is equivalent to the JSON sent to the server.

3.2.4 Migration

To make sure the query plan tables are correctly wiped when the corresponding benchmark result is deleted, we created a new migration: migrations/versions/e15e8b226491_benchmark_id_ondelete_cascade_query_.py adds the ondelete=cascade constraint to the query plan tables. Before the migration, we had to change the deletion order in get_tables_in_cleanup_order() in db.py, to make sure that endpoints like /wipe-db do not crash. But since there probably are a lot of other ways to delete part of the database, it is safer to do it through a new migration.

In case you want to manually apply the migration and are having problems doing so, follow these steps:

1. remove the mounted volume

2. set CREATE_ALL_TABLES:"false" in the docker-compose.yml
3. "docker compose down -v –remove-orphans"
4. "docker compose up -d db"
5. "docker compose run –rm app alembic upgrade head"
6. mount the volume
7. "make run-app" to start conbench

3.2.5 Validation

Now we already talked about how conbench stores and fetches the query plans but we have not yet talked about how conbench validates them. Conbench uses marshmallow schemas to validate incoming payloads. Since we use the same route as conbench when uploading data we also have to use the same validation. To do so we create a schema extension at startup and with it, overwrite the benchmark result marshmallow schema.

1. from conbench.entities.benchmark_result import _BenchmarkResultFacadeSchema, BenchmarkResultFacadeSchema
2. class ExtendedSchema(_BenchmarkResultCreateSchema): ...
3. BenchmarkResultFacadeSchema.create = ExtendedSchema()

This way, all validation will be done using the extended schema.

3.3 NebulaStream

In this section, the changes made to the NebulaStream source code, which were made to enable the query plan visualization on Conbench, will be described in detail. It additionally illustrates our intentions behind these additions, problems that occurred during the development, and ideas that did not make it into the final project. The fork of the NebulaStream repository, which we worked on, can be found here.

3.3.1 Compact documentation

While the following subsections will explain the newly added features in greater detail, the purpose of this section is to provide a compact overview of the most notable new functions, methods, and classes that have been added.

- grpc/SerializablePlanConbench.proto
  - message SerializableLogicalOperatorNode
    Serializes a logical operator using its id, children ids, parent ids, operator type and a label for the details

  - message SerializablePhysicalOperatorNode
    Serializes a physical operator using its id, children ids, parent ids and a label describing the operator type
  - message SerializablePipelineNode
    Serializes a pipeline using its id, predecessor ids, successor ids and a list of the serialized physical operators in this pipeline

- nes-executable/include/CompiledQueryPlan.hpp
  - void getPassingTuplesMap(*stageMap)
    Obtains a pointer to a map between pipeline ids and shared pointers of atomic counters. Iterates through all stages of the CompiledQueryPlan and adds shared pointers of the incomingTuples counters of every CompiledExecutablePipelineStage to the map.

- nes-query-compiler/src/PipelinedQueryPlan.cpp
  - void PipelinedQueryPlan::serializeAsJson(*resultJson)
    Obtains a pointer to a JSON as argument. Iterates through the pipelines of the pipeline query plan and serializes them to SerializablePipelineNode protobuf messages using the serializePipelineRecursive function. Afterwards, the messages are converted into the JSON format and appended to the resultJson.
  - void serializePipelineRecursive(serializedPipelines, pipeline, visitedPipelines)
    Serializes a pipeline to a SerializablePipelineNode protobuf message. Iterates through all physical operators of this pipeline to serialize them to SerializablePhysicalOperatorNode messages. Appends the serialized pipeline to the serializedPipelines vector reference. Will call itself to serialize all successor pipelines of this pipeline, if they have not already been serialized.

- nes-runtime/include/Pipelines/CompiledExecutablePipelineStage.hpp
  - private attribute: incomingTuples
    A shared pointer to an atomic uint64_t. This counter is incremented by the number of incoming tuples in the tuple buffer every time, the execute method of the stage is called.
  - private attribute: countIncomingTuples
    A bool which decides, if the incomingTuples counter is incremented when the execute method is called. Will only be true, if the - -visualizePlan argument was given to systest.

- nes-systests/systest/include/SystestConfiguration.hpp
  - BoolOption visualizePlan
    Configuration option to decide, if the steps to additionally visualize the query plans on Conbench should be performed.

- nes-systests/systest/src/SystestExecutor.cpp
  - program.add_argument("- -visualizePlan")
    Argument for the systest executable. To include the serialized pipeline plan and logical plan in the BenchmarkResults.json file, the systest must be executed with this argument and the -b argument.
- nes-systests/systest/src/SystestRunner.cpp
  - std::vector<RunningQuery> runQueriesAndBenchmark(queries, configuration, resultJson, visualizePlans)
    This function existed already before this project but is now called with an additional bool argument visualizePlans, which prompts the serialization of the query plans and their addition to the resultJson. If visualizePlans is not set, the function will operate like before.
  - void addIncomingTuplesToPipelinePlan(resultJson, incomingTuplesMap, finalTupleCount)
    This function will enrich each pipeline in the JSON pipeline serialization with an "incomingTuples" attribute. The attribute is taken from the incomingTuplesMap, which maps pipeline ids to the corresponding atomic counter. For the sink pipeline, this value is taken from the finalTupleCount argument.
  - void serializeQueryToJson(logicalPlan, resultJson
    Serializes a logical plan. Iterates through every logical operator in the logical plan and serializes them to a list of SerializableLogicalOperatorNode protobuf messages. Afterwards, the messages are transformed into the JSON format and appended to the resultJson.
  - void serializeOperatorToJson(logicalPlan, logicalOp, foundOps, operatorList)
    Serializes a single logical operator to a SerializableLogicalOperatorNode message and appends the message to the operatorList. Afterwards, the function will call itself to serialize all children of this operator, if they have not been serialized before.

Please note that the "benchmark.py" script to send the BenchmarkResults.json to Conbench is included in the Conbench repository of this project, in the systest_mimic directory, and not in the NebulaStream repository of this project. Therefore, sending benchmark results requires moving the JSON to the Conbench repository. A more detailed explanation of this can be found in section 4.

3.3.2 Initial ideas

The purpose of our project topic is to create a visualization of various query plan stages of benchmarking queries on Conbench. The visualizations should be able to explain the logical structure of the query while also facilitating the understanding of the query's performance. Especially, notable changes in a benchmark query's performance should be tracked to either the logical structure or compiled executable version of the query using the plan visualization.

When a system test of NebulaStream is ran with the benchmark flag $-b$ enabled, data about the query's performance will be assembled in a JSON file named "BenchmarkResults.json". This file is stored under the path "nes-systests/working-dir/BenchmarkResults.json" within the build-directory. By running the "benchmark.py" script in "scripts/benchmarking/benchmark.py" with the argument $--systest$, the results are transformed into $BenchmarkResult$ objects, which are then published to the Conbench server specified in the environmental variables. The structure of the "BenchmarkResults.json" file looks like this example:

```
[
    {
        "bytesPerSecond": 65262.91298278269,
        "queryName": "ClusterMonitoring",
        "time": 0.008596,
        "tuplesPerSecond": 6514.657980456026
    }
]
```

Since JSON files are easily extendable and NebulaStream already provides means to create a protobuf serialization of the logical plan, our initial approach was to append protobuf serializations of the different plans to the JSON. The Conbench server itself should then deserialize the plans and transform them into the visualization format. However, this approach was abandoned since it would have required moving the proto files to the Conbench side and implementing a deserialization function on the server side. We did not want to increase the amount of work on the server and therefore decided to append the query plans to the result JSON in a format that could directly be used by the server to generate the visualization.

3.3.3 Serialization of the Logical Plan

The logical plan is the first query plan stage of a NebulaStream query. It is a tree of logical operators, starting with the sink as a root node and ending with the sources of the data as leaves. While the logical plan does not represent the actual, physical operations that the system performs to execute the query, its abstract form facilitates the understanding of the query itself. The Conbench server does not display the query in SQL format but only displays the name of the query. Adding a visualization of the logical plan tree solves this issue.

We serialize the logical plans of the ran systest queries in the $serializeExecutionResults$ function in the "SystestRunner.cpp" file. This function already existed before our project, since its purpose is to create the BenchmarkResult JSON. The logical plans of the queries are stored in the $SystestQuery$ struct, which the

SystestRunner has access to at all times. This is convenient, since it means that the logical plan serialization can be obtained after the queries ran. Therefore, no extra work is added during the queries' execution.

A single logical plan and a reference to a nlohmann::json object are passed as arguments into the *serializeQueryToJson* function, which we added to the "SystestRunner.cpp" file. This function takes every root operator, meaning sink, of the logical plan, traverses the tree top-down, and transforms the found operators into *SerializableLogicalOperatorNode* protobuf serializations. In our serialization, a logical plan graph is represented as a list of homogeneous logical operator nodes. The structure of a single operator node can be observed by looking at the *SerializableLogicalOperatorNode* message, which can be found at "grpc/SerializablePlanConbench.proto":

```
message SerializableLogicalOperatorNode {
  uint32 id = 1;
  repeated uint32 inputs = 2;
  string label = 3;
  string node_type = 4;
  repeated uint32 outputs = 5;
}
```

The id and node type correspond to the logical operator id and the concrete operator type like 'Projection'. The type of an operator can be obtained from the output of its *getName*() method. The inputs of a logical operator are the ids of its children, the operations which are performed before the operation itself. They can be retrieved by using the *getChildren*() method of the logical operator. The outputs, on the other hand, are the ids of the parents of the operator, meaning the operations that are performed after the operator itself. The parent of an operator can be retrieved by the *getParents*(*plan*, *operator*) method in "LogicalPlan.cpp". Both the input and output ids are used on the Conbench server to create the edges between operator nodes. Lastly, the label of a node provides additional information about a logical operator, which varies depending on the operator type. For instance, the label of a selection operator will describe its predicate. The label is obtained by taking the result of the operator's *explain*(*ExplainVerbosity* :: *Short*) method.

The labels of the nodes should be informative, but not too verbose, since they need to be displayed inside of a visualized logical operator node on the server. We therefore decided to use the *Short* verbosity setting instead of the lengthy *Debug* setting. However, we changed some of the explain methods' outputs to make them shorter or more informative. For instance, the short explain output of a window aggregation now names the fields on which the aggregation is performed. On the other hand, the short output of a projection's explain will now only mention the field name and the chosen name via "as" if the field was actually renamed or used in a mapping, to avoid space-consuming explanations like "source1$id as source1$id".

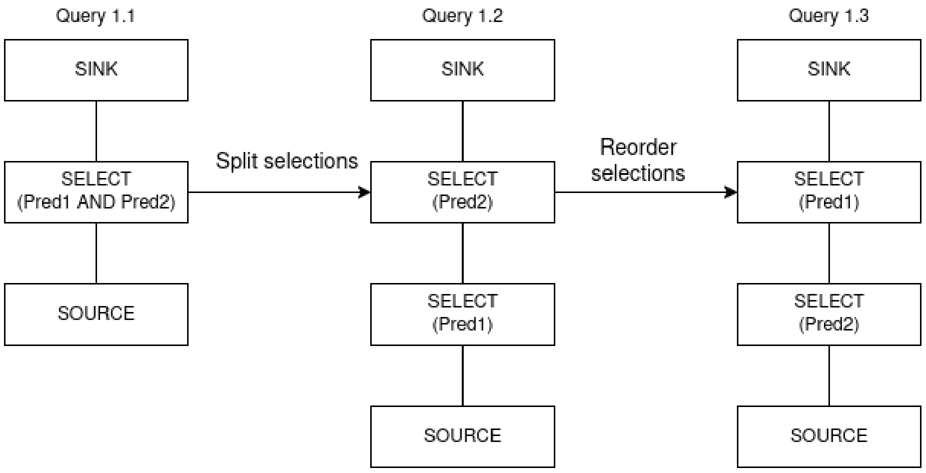

Figure 1: Splitting a logical query into one query per optimization rule. The second predicate of the selection is estimated to eliminate more tuples and is therefore moved before the first predicate.

After obtaining the protobuf serializations of every logical operator of the logical plan, the *serializeQueryToJson* function will then transform every protobuf serialization into a JSON string using *google* :: *protobuf* :: *util* :: *MessageToJsonString*. This string is then parsed into the nlohmann::json format and appended to the JSON reference argument. The *serializeExecutionResults* function can then simply append this JSON as "serializedLogicalPlan" to the BenchmarkResult JSON, which concludes the logical plan's serialization.

Originally, we created the JSON of the logical plan directly, without utilizing protobuf serializations. However, considering that the protobuf message format assists in understanding the expected format of the logical plan and makes the process of the serialization more maintainable, we decided to use protobuf after all. "grpc/SerializableQueryPlan.proto" already contained messages to serialize the logical plan. The purpose of these protobuf messages is to create a serialization of the logical plan with all of its details, for instance to communicate with the NebulaStream UI. This involves specific SerializableVariantDescriptor messages, which encapsulate the details of logical operators. Since our focus was to create a more abstract graph representation with homogeneous nodes, we decided against using the preexisting, more complex serializations and created our own. All newly added protobuf messages can be found in "grpc/SerializablePlanConbench.proto".

In addition to that, we planned to include a mode to create multiple logical plan versions of a single query, one per optional optimization rule. When we planned this feature, "nes-nebuli", the component responsible for creating the logical plan, applied some basic query rewriting rules, like splitting a conjunction predicate of a selection operator into multiple selections with a single predicate.

The purpose of this feature was to visualize how the execution time of a query improves per optimization step. A visualization of it can be observed in figure 1. However, in a commit on the main NebulaStream repository, the optional rewrite rules were removed, which led to the rejection of this idea. While a "legacy-optimizer" with optimization rules from the now private, original NebulaStream repository has been recently re-added to "nes-nebuli/src/LegacyOptimizer", we did not reintroduce this feature. However, if new rules were added in the nes-query-optimizer component, reintroducing such a feature to measure the rules' effectiveness could prove to be useful.

3.3.4 Serialization of the Pipeline Plan

While the logical plan visualization serves the purpose of depicting the logical structure of the query itself, it does not visualize the actual operations that are executed. This is why we decided to additionally visualize the pipeline plan of every query on Conbench.

There are multiple representations of the physical operations throughout the compiling process of a query in NebulaStream. When the SingleNodeWorker's *registerQuery*(*logicalPlan*) method is called, it firstly uses the *optimize*(*logicalPlan*) method of the QueryOptimizer to create a physical plan. In this plan, the logical operators are already lowered to their executable, physical versions, except for the sink and source node, which are abstracted in this plan. The SingleNodeWorker then uses the *compileQuery* method of the QueryCompiler class. In this method, the physical plan is lowered to a pipeline query plan. The pipeline query plan contains pipelines in which multiple physical operators can be executed at once. Only when pipeline-breaking physical operators like an aggregation build operator occur, a pipeline is finished by emitting the data and a new pipeline is created as a successor of this pipeline. After this plan is created, the compiler lowers the plan to a CompiledQueryPlan object, by creating CompiledExecutablePipelineStages, which represent executable pipelines. This plan still includes an abstracted sink and source, but adds an InputFormatterTask for each source.

We decided to visualize the pipeline plan on Conbench since its pipelines represent the actual order of physical operations performed on incoming tuples. At the same time, we did not want to overload the visualizations and therefore decided to keep the abstract source and sink representations. However, because the pipeline plan is an intermediate stage between the physical plan and compiled plan, it only exists during the execution of the query compiler's *compileQuery* method. We therefore needed to create the serialization during the query compiling process.

The query registration method of the worker is called from the *runQueriesAndBenchmark* method of the SystestRunner. If the flag - -visualizePlan is set as an argument of systest, a nlohmann::json object to contain the pipeline plan serialization will be created and a pointer to it will be passed into the *registerQuery* method. We pass a raw pointer because C++ does not allow passing references as std::optional arguments. The SingleNodeWorker passes the JSON pointer as an argument into the *compileQuery* method of the QueryCompiler. If the pointer is not a nullptr, this method will call the newly added $serializeAsJson(resultJson)$ method of the pipeline plan in "nes-query-compiler/src/PipelinedQueryPlan.cpp", before the plan is lowered to the compiled plan.

This method operates similarly to the serialization of the logical plan. It traverses the pipeline plan from top to bottom and recursively serializes all physical operators inside every pipeline as well as the pipeline itself. Afterwards, the protobuf serializations are transformed into the JSON format and appended to the JSON that the resultJson pointer points to. The serialization utilizes two new protobuf messages which we created for this purpose:

```
message SerializablePipelineNode {
  message SerializablePhysicalOperatorNode {
    uint32 id = 1;
    repeated uint32 inputs = 2;
    string label = 3;
    repeated uint32 outputs = 4;
  }

  repeated SerializablePhysicalOperatorNode operators = 1;
  uint32 pipeline_id = 2;
  repeated uint32 predecessors = 3;
  repeated uint32 successors = 4;
}
```

We serialize the pipeline plan as a list of 'SerializablePipelineNode' objects. Each node contains the pipeline's id, the ids of its successors and predecessors, and a list of 'SerializablePhysicalOperatorNode' objects. The 'SerializablePhysicalOperatorNode' message is built similarly to its logical counterpart, but it notably does not include the 'node_type' attribute. This is because the type of the physical operator itself is used as a label. Since the focus of the pipeline plan visualization is the pipelines, it felt unnecessary to include detailed explanations for the operators again, especially since the logical plan provides this information. The operator traversal for a pipeline is easier than for the logical plan, since operators in a pipeline always have one child and one parent (exceptions are the scan operator, which does not have a parent, and the emit operator, which does not have a child).

A notable part in the serialization of the pipelines is the process of obtaining the ids of the predecessors. The predecessors can be obtained via a *getPredecessors* method, which we implemented. However, predecessor pipelines are stored as weak pointers because they are destroyed in the process of lowering the pipeline plan to the compiled plan. This does not pose a threat to us, since we serialize the plan before it is lowered. However, we still need to lock the weak pointers before accessing a predecessor pipeline.

When this feature was implemented, the SystestRunner accessed the SingleNodeWorker directly to register the query. Therefore, the number of functions that the JSON pointer needed to be passed through was relatively low.

However, NebulaStream added the QuerySubmitter class, which is now responsible for submitting the query to the worker. This increases the amount of functions which need to pass the JSON pointer, which makes our approach look a bit "forced". Although, any other way of obtaining the serialized pipeline plan without passing the JSON pointer between the methods would most likely involve the SystestRunner accessing the compiler directly, which did not seem ideal either.

3.3.5 Counting incoming tuples per pipeline stage

The execution time of a query can be strongly dependent on the order of the operators. The order of selection operators or join operators, which decrease or increase the number of passing tuples, is especially crucial. Therefore, if a benchmark query performs well or badly, it would be useful to be able to see the amount of tuples that arrive at each pipeline. This motivated us to implement an incoming tuples counter for the intermediate executable pipeline stages.

At first, we planned to implement a counter for the incoming tuples of every physical operator. However, the *execute* method of a physical operator is always called for exactly one tuple. Since the counter needs to be thread-safe due to the usage of multiple threads during the query execution, millions of accesses to such a counter during the runtime of the query would probably negatively impact the execution time of the query. Additionally, each of the execute functions of the operators of each pipeline is compiled by the Nautilus compiler into executable code. Integrating a tuple counter in this code seemed too complex, since none of us was familiar with Nautilus before this project. This approach was therefore rejected.

Instead, we only count the incoming tuples per CompiledExecutablePipelineStage. For this, we added an atomic uint64_t "incomingTuples" attribute to the CompiledExecutablePipelineStage class in "nes-runtime/include/Pipelines/CompiledExecutablePipelineStage.hpp", as well as a "countIncomingTuples" flag. If this flag is set, the atomic counter will be incremented by the number of tuples in the input tuple buffer when the *execute* method of the stage is called during runtime.

In order to enrich the pipeline plan visualization, the SystestRunner needs access to these counters. To ensure this, we create an "incomingTuplesMap" map for each query in *runQueriesAndBenchmark*. This map maps the id of a pipeline to the shared pointer to the atomic tuple counter of the pipeline stage. Alongside the JSON pointer for the pipeline plan serialization, a pointer to this map is passed into the *registerQuery* method. This will prompt the compiler to set the "countIncomingTuples" flag for the pipeline stages to true when creating the compiled plan. After this plan is created, we call the newly added *getPassingTuplesMap*(*mapPointer*) method of the compiled plan, which is located at "nes-executable/include/CompiledQueryPlan.hpp", from the *registerQuery* method of the SingleNodeWorker. This method inserts shared pointers to all incoming tuple counters into the map. We use shared pointers to ensure that the SystestRunner has access to the counters even after the pipeline stages are

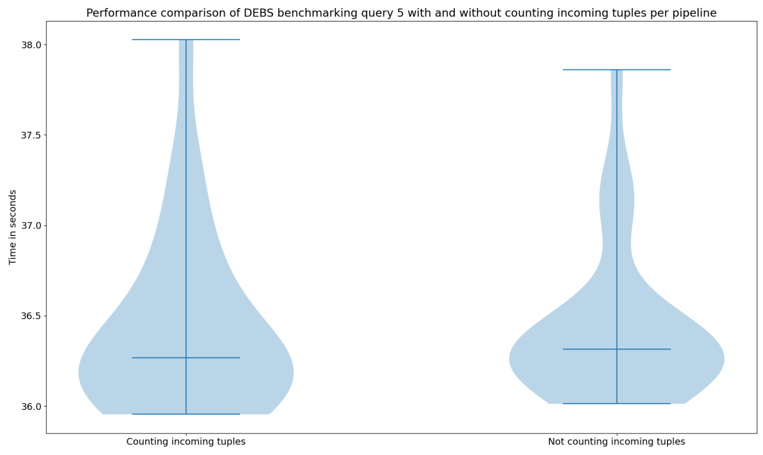

Figure 2: Violin plot of the times of 25 DEBS query 5 runs with and without counting of incoming tuples enabled. The horizontal lines represents the median and the extrema.

destroyed.

After the query finished running, the *runQueriesAndBenchmark* method of the SystestRunner calls the new function *addIncomingTuplesToPipelinePlan*, which is also located in "SystestRunner.cpp". This function obtains the map of the counters and the JSON, which contains the pipeline plan serialization, as arguments. For each pipeline node, we use the pipeline id to access the corresponding incoming tuples counter from the map. The value of the counter is then added as an "incomingTuples" attribute to the node in the JSON.

Since source and sink nodes are still abstract in the compiled plan, we do not obtain a counter for those operators. However, we can obtain the number of incoming tuples for the sink pipeline by counting the number of lines in the result CSV file or taking the value of the very first field in the file if the sink is a checksum sink. Sources use the counter of their successor pipeline for their incoming tuples since the pipelines of the source and the InputFormatterTask do not eliminate any tuples.

The pipeline plan JSON serialization is added to the BenchmarkResult JSON as "serializedPipelinePlan" in the *serializeExecutionResults* function in "SystestRunner.cpp".

In order to test if the tuple counting would heavily impact the benchmarking times, we ran an experiment. The fifth of the DEBS benchmark queries was run 25 times with and 25 times without the counting of incoming tuples per intermediate pipeline stage. The query forms two intermediate pipelines and therefore two atomic counters. It processes over 42 million tuples. The experiment was performed with the standard GitHub-hosted runner for private repositories with 2 CPUs, 7GB RAM, 14GB storage, and x64 architecture. It used 20000 buffers in the global buffer manager and 2 worker threads. The result can be observed in the violin plot in figure 2.

It is evident that the query time was not affected strongly by the atomic counters. As indicated by the median and the peak of the violin plots of both variations, both options seem to usually lead to a time of around 36.3 seconds with this setup. However, the maximum value of the left violin plot, representing the enabled tuple counting option, is slightly higher than that of the right plot. Additionally, the distribution graph indicates that counting incoming tuples might lead more often to a time above 37 seconds for this setup, which could be caused by the counters.

4 How to setup conbench

In our Conbench-Query-Plan-Visualization repository, we feature the systest_mimic folder which includes an example benchmark result (BenchmarkResults.json) and the benchmark.py which actually transfers the data. The benchmark.py is a stripped-down (and slightly changed) copy of what Nebula already uses to send its own data to their conbench server.

4.1 Prerequisites for running locally

For benchmark.py to properly function, you might need to install the conbench package and dotenv first.

- pip install -e benchadapt/python/
- pip install python-dotenv

Keep in mind not to accidentally do this in the root of the project. Also, set up a .env file to set these variables:

- CONBENCH_URL="http://127.0.0.1:5000/"
- CONBENCH_EMAIL="test@test.com"
- CONBENCH_PASSWORD="test"
- CONBENCH_RUN_REASON="test"

This example uses "test" as the values, but you can use whatever you used to register your account on the conbench page. While registering the account, you will have to provide a registration key, which will be "innocent-registration-key" by default.

4.2 Generating a BenchmarkResults.json with NebulaStream (optional)

If you want to try out the functionalities added to NebulaStream, which were described in 3.6, you can run a benchmark query yourself to generate the BenchmarkResults.json file:

- Clone the project's NebulaStream fork from the repository
- Set up the NebulaStream development environment using this guide
- Choose a benchmark .test file in nes-systests/benchmark/ or create a .test file on your own
- Load the cmake project with the options `-DCMAKE_BUILD_TYPE=Debug -DENABLE_LARGE_TESTS=ON`
- Build the project
- Execute the systest executable in the nes-systests/systest/ directory of your build directory with the following arguments:

  ```
  -b
  --visualizePlan
  -t /tmp/nes-public-DSPRO-Plan-Visualization/<path-to-your-test-file>
  -- --worker.numberOfBuffersInGlobalBufferManager=20000
  ```
- In nes-systests/working-dir in your build directory, you will find the BenchmarkResults.json file of your query. Copy it
- In your cloned conbench repository, in the systest_mimic directory, replace the BenchmarkResult.json with the one you copied

4.3 Sending data to conbench:

Use "make run-app" or "make run-app-dev" at the root of the conbench repository, to start the server (if you haven't already registered the account) and once running you can use "python systest_mimic/benchmark.py" in a new console to send the data to conbench. Going to http://127.0.0.1:5000/ should then display the result (you might have to reload the page).

4.4 Running tests

To run the conbench tests use "make tests" or if you haven't added github, use:

```
docker compose down && docker compose build app &&
   docker compose run app  pytest -vv conbench/tests
```

As you should see, all tests except the "docs" test are passing. The doc test fails because of an outdated string comparison.

5 Evaluation and Reflection

At the start of the project, the first goal was to obtain an overview of NebulaStream to find out how the systests operate and where the query plans are created. In this regard, it was helpful to have a project member that was already a bit familiar with NebulaStream. This way, we were swiftly able to determine the core tasks for the project.

Regarding the work on the NebulaStream itself, looking back, it would have been better practice to sync our fork more frequently with the main branch of the NES repository. Especially in the month before the open source release, many refactors of components and clean-ups were pushed to the main branch. These also affected the nes-nebuli component and the systests. As a result, some of our contributions were a bit troublesome to fit into the project after a rebase. Examples of this would be the removal of the logical plan rewrite rules or the addition of the QuerySubmitter for the systest component.

On the other hand, the plan serialization itself remained stable throughout all the syncs with the main repository, even if a plan stage was changed. This can be considered a success, since the homogeneous type of serialization we chose seems to handle changes in operators well, without needing adjustments.

For the Front-End, ultimately it was better to build the core structuring from the group-up with custom elements rather than relying upon preexisting libraries to handle the visual work. This was due to the fact that most of these libraries had certain focus points and we tried to adapt them according to our complex specifications. We could've also opted for a less visually striking version as it generally can be seen with other standard SQL systems such as the "db2explain" function.

In the end, custom code was needed to complete our requirements. This aspect should be taken into consideration for future endeavors as a learning experience. For building initial ideas quickly, it is a good choice to do so with proven and established software. However, when the complexity in our problems starts to outgrow the benefits of adaptation, the key aspect of software evaluation becomes vital. Therefore, before starting such a complex task, the full end-to-end scope should be evaluated including the different requirements, not only of the resulting final product, but also the requirements in architecture and the working basis of technology. By choosing the right programming paradigms one can create a solid foundation for the basis of their work. This then will save a lot of time in the future, by effectively choosing the right frameworks to act as guides and enablers rather than blockers in development.