May 2008

NON-DUPLICATIVE APPROACH TO SHARING BETWEEN STREAMED QUERIES

Bart Shappee
Worcester Polytechnic Institute

Christopher Adam Bass
Worcester Polytechnic Institute

James Albert Roumeliotis
Worcester Polytechnic Institute

Project Number: EAR-0708

NON-DUPLICATIVE APPROACH TO SHARING BETWEEN STREAMED QUERIES

A Major Qualifying Project Report:

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

_____

James Roumeliotis

_____

Bart Shappee

_____

Christopher Bass

Date: April 24th, 2008

Approved:

_____

Professor Elke Rundensteiner, Major Advisor

1. sharing
2. streamed
3. queries

Abstract
The push for streaming database systems to handle massive amounts of data and multiple queries necessitates the development of efficient yet adaptive query sharing technology. This project designed an effective solution to this problem poised as NASSQ, an elegant hybrid between static and dynamic routing alternatives. Utilizing the adaptive architecture of dynamic routing systems, NASSQ supports adaptive sharing of operators among different queries while refraining from duplicating intermediate data tuples. However like static routing, NASSQ constructs optimized routes using statistics.

Acknowledgements
The NASSQ system and this entire Major Qualifying Project could not have been completed without the help and assistance of a number of people. First and foremost, the guidance provided by Professor Elke Rundensteiner insured that our project was on track consistently throughout the school year. We would like to then thank our representatives and collaborators at MITRE, Jennifer Casper and Peter Leveille for providing us with the facilities as well as the practicality information on SDAF for our project and logistics with badging and lab setup. With the MITRE Corporation we would like to thank Jing Hu for providing the data generator. When struggling to switch over to the CAPE engine, we would like to give a special thanks to Karen Works for dealing with our countless questions regarding CAPE as well as Rimma Nehme, Venkatesh Raghavan, and the rest of the DSRG staff for all of the assistance.

Finally we would like to thank the NSF Funding agency for the $100,000 to purchase the computer equipment for testing. This material is based in part upon equipment supported by the National Science Foundation under Grant No. 0551584. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Table of Contents

1 Introduction....8
1.1 Background and Motivation....8
1.2 State-of-art in Streaming Systems....9
1.3 Problem Statement....10
1.4 Roadmap of the Document....11
2 Background....13
2.1 Streaming Database Systems....13
2.2 Static Routes....14
2.3 Dynamic Routes....17
3 SPE Evaluation....19
3.1 Requirements of System Processing Engine (SPE)....19
3.2 Requirements: In-depth Analysis....20
3.2.1 Custom Operators....20
3.2.2 Sharing Data among Queries....21
3.2.3 Dynamic Query Plans....21
3.2.4 Saving State of Query Plans....22
3.2.5 Comparing Metrics....23
3.2.6 Reordering....24
3.2.7 Performance Test....24
3.3 Stream Engine Evaluation....25
3.3.1 Calder....25
3.3.2 CAPE....25
3.3.3 Coral8....25
3.3.4 Esper....27
3.3.5 RiverGlass....29
3.3.6 STREAM....30
3.4 The Chosen Engine....32
4 The Approach: Non-duplicative Approach to Sharing (NASSQ)....33
4.1 Description....33
4.1.1 NASSQ Routing Tree: Definition and Construction....34

4.1.2 NASSQ Operator Process....35
4.1.3 NASSQ Process Layout....36
4.1.4 NASSQ Benefits....37
4.2 The Data Flow....38
4.2.1 Tuple Batch....38
4.2.2 Routing Tree Generation....39
4.2.3 Operator Processing Logic....41
4.3 Design of NASSQ....41
5 Implementing NASSQ on CAPE....44
5.1 Insights into CAPE....44
5.2 QueryMesh....44
5.3 Key Packages....45
5.4 Java Files Implemented....46
5.4.1 Routing Tree Generation....46
5.4.2 Data Preparation....46
5.4.3 Execution....47
5.5 XML Files Implemented....47
6 Testing and Results....50
6.1 Testing....50
6.1.1 Benchmarking....50
6.1.2 Stream Files....51
6.1.3 Procedure....51
6.2 Testing Decisions....52
6.2.1 Time Stamping....52
6.2.2 25 Query Measurements....53
6.3 Testing Issues....53
6.3.1 Academic Code Base....53
6.3.2 Stream Generator....53
6.3.3 Implementation Limitations....55
6.4 Data Analysis....56
6.4.1 Set Up....56
1.1.1 Throughput....57
1.1.2 System Utilization....57

1.1.3 Latency.....63
1.1.4 Memory Usage.....64
2 Conclusions.....65
2.1 Future works.....65
3 Bibliography.....67
4 APPENDICIES.....69
4.1 Appendix A: SPE Feature List.....69
4.2 Appendix B: Test Result Data.....71
4.3 Appendix C: Generated NASSQ Batch.....77

Table of Figures

Figure 1: Static Route Unshared.....14
Figure 2: Static Route Shared.....15
Figure 3: The Eddies System.....17
Figure 4: Query Plan.....18
Figure 5: Sharing.....21
Figure 6: NASSQ Operator Diagram.....35
Figure 7: NASSQ Routing Tree.....36
Figure 8: Static/NASSQ/Dynamic Overview.....37
Figure 9: Tuple Batch.....38
Figure 10: Routing Tree Generation.....39
Figure 11: Pass/Fail List Structure.....40
Figure 12: Operator Process on the Tuple.....41
Figure 13: Key Packages.....45
Figure 14: Java Files Implemented.....46
Figure 15: Tuple Throughput Chart.....57
Figure 16: Approximate CPU Usage Chart.....58
Figure 17: Latency Chart.....59
Figure 18: Approximate Memory Usage Chart.....60

1 Introduction

1.1 Background and Motivation

The purpose of a standard relational database is to store large quantities of information, and provide a technology for efficiently retrieving and displaying relevant subsets of information of the possibly huge data store [1]. Relational databases are useful for static or mostly-static data, but are not optimized for situations where data constantly updates and changes more often than the queries run upon it. A relatively new class of systems has emerged in recent years, known as streaming databases that target to handle more dynamic data scenarios. Instead of storing static data in a persistent store first and then running one-time queries on it, these systems receive and operate over large quantities of dynamically incoming data on streams through continuously standing queries [2].

For example, say the military had sensor arrays, or satellites or any other form of vehicular tracking, laid out in Baghdad to keep track of troop movements. They would like to quickly receive updates about any changes of the military situation, such as certain vehicles showing up, or large groups of enemy or American troops disappearing or relocating. These sorts of checks are clearly time-sensitive, as actions may need to be taken in real time to react to the situation. Multiple military leaders may need access to the same incoming data, without having to wait unpredictably long to receive those updates. In this case, a streaming database has many advantages over a relational database to support these types of real-time data monitoring applications, as explained

further below.

Streaming systems offer users several advantages over their static counterparts. For example, they can receive new data at any time during the execution and process it in real-time with recent data residing in main memory (without first writing it to disk), respond to these updates, and feed the results to any actually hooked-up applications. They can quickly report new results derived from the incoming data, which enables them to handle time-sensitive data with minimal delay. Results must be bound to only a portion of the more recent data, referred to as windowing, rather than running over all data that has ever passed into the database. In other words, streaming databases tend to operate on moving windows. Relational databases, on the other hand, are by nature finite in size, and queries are logically specified in the complete snapshot of the data store. Given the volume of persistent data stored, static databases do not tend to preside real-time opportunities, instead they optimize for the overall throughput of the system. Streaming databases on the other hand aim to maximize for maximal continuous output rate given data is potentially infinite.

1.2 State-of-art in Streaming Systems

In previously implemented streaming database systems, there exist both static and dynamic methods of choosing the order to process data in [3] also called query plans. In static route generation, a complete path is generated at compile time when registering the query before the processing of the data begins. There are many algorithms that have been used to generate these paths, but they typically are complex and sophisticated. In static

databases, some data is fully known before the query commences, paths can be completed and then utilized for optimal route design. Hence, the routes, calculated using these statistics, are expected to remain optimal during query execution [4].

A new paradigm has emerged to generate routes dynamically for streaming systems. In this paradigm, a complete path is never generated; instead a central processor directs tuples one by one through the query network from operator to operator using local calculations, such as observed variability or speed of operators [5]. Routing in this type of system tries to be highly adaptive.

However, the path generation algorithms in these dynamic systems are calculated at the individual tuple level, and therefore tend to be simple heuristics and at best locally optimal. Simpler routing algorithms keep the overhead of route generation small, which is critical because this cost is accrued for each tuple during run time.

While both methodologies have their respective scopes of applicability, the question arises if alternate efficient routing decisions other than purely static or dynamic systems could be derived that may be more high-performance.

1.3 Problem Statement

In this project, we seek to improve the efficiency of stream processing engines by designing a more routing paradigm mechanism that borrows the benefits of both already-existing statically and dynamically routed stream processing engines.

We began this project by researching current stream processing engines to learn what has already been accomplished. Once we have established a working knowledge of these existing systems, we began piecing together plans for a new routing method, one that would keep the positives of currently existing systems while mitigating as many of the negatives as possible. Once our new approach was developed, we revisited the stream processing engines we had researched previously to select a platform for our system development. We chose the WPI's Constraint-exploiting Adaptive Processing Engine (CAPE) [9] as the system to use as our core platform explained in Section 5. We build our project on top of one of CAPE's subsystems, Query Mesh [14]. This sub-system's design idea is similar in some ways to our system, and has all the necessary features to use to implement our vision as we will explain in Section 4.1. After implementing our method on the CAPE Query Mesh system, we performed various experiments that we designed to gauge the success of our proposed paradigm and its implementation.

1.4 Roadmap of the Document

Following this introduction, this paper goes into more depth to describe our research into streaming systems, our system, and the testing thereof.

In Section 2, we review static and dynamic routing as they apply to streaming database systems. After that, in Section 3, we record the results of our research into various currently-existing stream processing engines. We also describe our choice of the stream processing engine to use as the platform for the development of our project in Section 3, leading into Section 4 where we explain the methodology behind our core work, the

NASSQ system. In Section 5, we revisit our chosen stream processing engine, explaining why this system was a good choice for implementing NASSQ. We continue by describing our actual implementation of NASSQ; this is followed by our testing and conclusions in Sections 6 and 7. The end of our paper, Section 8, contains various appendices that we reference throughout the paper to showcase certain points that would not have fit well in the paper itself.

2 Background

2.1 Streaming Database Systems

A traditional static database conforms to the relational model of how the data is structured [1]. The data is typically stored on persistent storage devices. A real-time database, also known as a streaming database, is designed to handle data constantly being updated [2]. Unlike a traditional database system, where the data is input through a query language, the data travels through a stream and is constantly flowing. A streaming database system is set up as a dam interrupting the flow of data. The streams of data pass through the system and are queried in real-time. Tuples of data that proceed to pass through the entire query have succeeded and are output by the system as results. Streaming database systems have been for instance applied to monitoring applications such as the stock market, where the current values of markets are constantly changing [6].

The desire for optimization is where the battle lies for current streaming systems. In a relational database, the queries can be optimized prior to calling upon the database; however, with streaming data, the queries on the data may seem optimal at first yet later may quickly become ineffective. A change of the initially optimal query execution structure called the query plan may be necessary to again process the data efficiently. The two main alternative paradigms for processing of queries with streaming systems revolve around the concepts of static versus dynamic routing. A static routing system is one that does not change; the plan is established and set for the entirety of the run. A dynamic

routing system focuses on the adaptation of the system in regards to new information with the goal of being continuously optimized to best meet the experiments of the ever changing environment. Both approaches have respective slopes of applicability; however, through the combination of ideas from both static and dynamic routing systems, in this work we defuse a novel more effective paradigm.

2.2 Static Routes

A static route does not change over time. It is created at compile-time and then stays unchanged throughout its execution. Relative to streaming database systems, tuples are processed through the queries of the system. When a tuple arrives, a copy of the data is created for each individual query of the system. In Figure 1, there are three queries in the

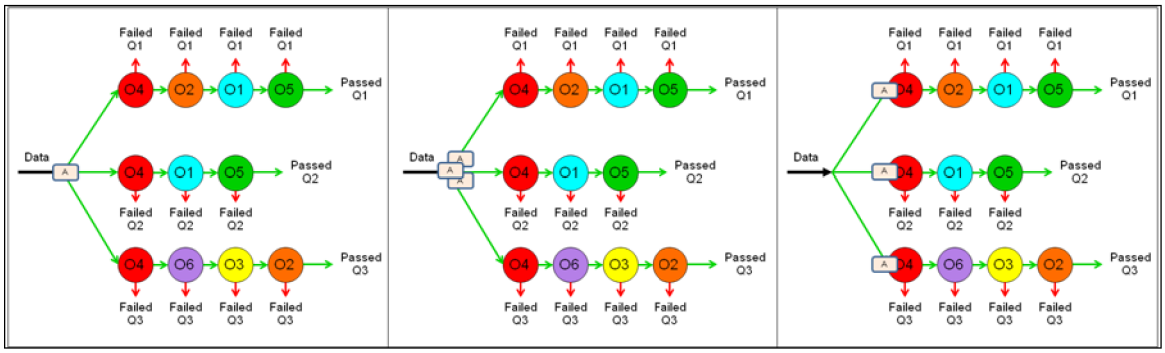


Figure 1: Static Route Unshared

system. When the data arrives on the left side of the figure, tuples are copied three times, one for each query. The reasoning behind this idea is that when a tuple fails at a particular operator it is dropped from the system. When a tuple hits operator 2 of query 1 and fails, that tuple is dropped and not processed any further of query 1. If only one copy of the tuple were to exist then the other two queries would never get handled. With tuple replication, while logically correct, there is now a performance issue. Normally, a large

amount of unnecessary data may be created wasting memory as well as CPU resources. Yet, because of this replication, there is a low overhead for each tuple. Since the path is established and the processing can now proceed simply along the pipeline without any further decision making each tuple does not concern itself on where to go.

While the tuples may be duplicated for each query, the previous method discussed the scenario without considering query sharing. When introducing sharing into the system, not only can routing heuristics be more sophisticated without additional overhead due to being amortized, but the tuple duplication can also be limited. Figure 2 illustrates the same database system as seen in Figure 1 except now with sharing. The operators distinct in the system among the queries are now identified and established as possible shared operators. Many heuristics for operator sharing can be employed. For a simple example, let us focus on a scenario where operator 4 is found in all three queries. Then the algorithm may choose to share this operator among all three queries. The difference in the system now is that when a tuple arrives, it is not duplicated at first. With sharing introduced, if a tuple fails at operator 4 it is impossible to pass the other three queries and the tuple is therefore dropped from the system. This eliminates that particular tuple from being copied. If the tuple passes operator 4, tuple duplication still exists. The tuple is copied and passed to both operators 1 and 6. In this particular example, operator 1 is also

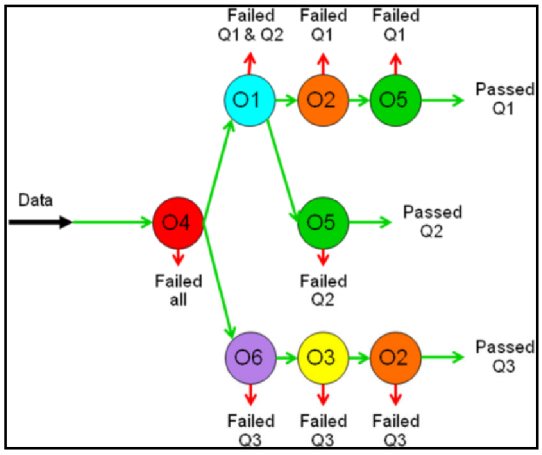


Figure 2: Static Route Shared

shared, this time between query 1 and query 2. The same benefits apply when a tuple arrives at operator 1. However, just as before, if a tuple passes, the tuple is copied yet again. Clearly, tuple replication still exists. The main difference is that it is now limited rather than coping each tuple initially blindly. Another important note to make is that sharing is not necessarily the best solution. Analyzing query 1 and query 2, operator 5 seems to be in each query. Clearly sharing operator 5 is possible. However, let us assume in this particular example that query 2 is a very simple filter while operator 5 consists of a more complex and thus expensive operation. The processing time of the operator must be taken into consideration to allow for the optimal structure. With query 2 being a simple filter yet having a low throughput (fewer tuples pass this operator), it is more efficient to have tuples carry to operator 2 and fail rather than share operator 5. There are many ways to look at this particular example considering sharing resources and CPU processing. It is however important to realize that sharing is not always the best method.

As for the main concept of static routes being unable to adhere to change, the path of the tuples (the query layout as in Figure 1) must be rebuilt. This modification is costly and must be worked in appropriately so not to lose any tuples at runtime during query plan manipulation. Since data is constantly flowing, all of the data currently being managed must finish before the new route is established. This puts a strain on the system since the flow is temporarily stalled. While static routes are not easily adaptable, the efficiency of execution due to lack of overhead as well as the possibility to apply sophisticated routing heuristics results contributes to a practical streaming database system. For this reason, most prototype streams so far have adopted these static routing methods.

2.3 Dynamic Routes

The principles introduced with a dynamic routing strategy incorporate adaptive execution plans. With static routes, the execution plan was established initially and kept fixed for the remainder of the system; that is, to change the plan a rebuild of the execution plan is required. Adaptive execution plans focus on updating and changing the execution plan immediately without having to rebuild each time. The environment and stream characteristics changes since the data is constantly streaming. While tuples coming in from different streams may be different, the dynamic route allows for the astonished path to be executed for each individual tuple.

As an example, a variation of the Eddies system (Figure 3) will be explained [7]. It is important to remember that while this example reflects the design of Eddies, not all dynamic routes function this way. With any dynamic routing, the principle revolves around adaptive execution plans. With Eddies, the idea is to involve an additional operator that functions as a router rather than a traditional query operator. When a tuple arrives into the system, the router suggests the next operator to travel to during runtime, not prior to it. An important note to make is that the entire path is not determined, only the next operator for a particular tuple to go to next to execute is chosen. When the operator is finished processing the tuple, information is marked on the tuple to record whether it passed or failed the operator. Once the tuple is

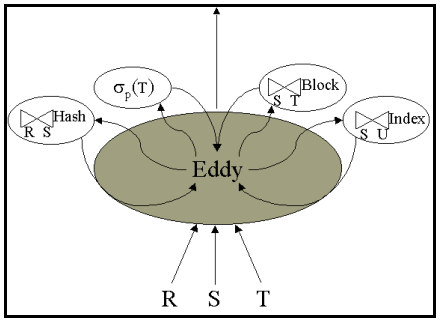


Figure 3: The Eddies System

finished, it returns to the router. The router uses the information from the tuple to determine the next operator to route the tuple to if more processing is needed.

If a tuple fails at a particular operator, the adaptivity of the system becomes clearer. To understand this benefit Figure 4 will be used to establish the main queries in a given example. Figure 4 shows three different queries that consist of a number of operators. Query 3 in particular incorporates operators 2, 3, 4, and 6. If the router decides to pass the tuple towards operator 6 and fails at that operator, the router's execution plan changes. Query 3 is the only query that uses operator 3. With the tuple fails at operator 6, the entirety of query 3 is noted as a failure. The router decides that there is no reason to ever send the tuple towards operator 3 since query 3 is the only query to access that operator and that query has failed. This adaptivity prevents the tuple from traveling to unnecessary operators.

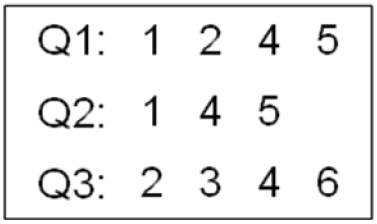


Figure 4: Query Plan

While this is a benefit in reducing the number of operations per tuple, minimal overhead is now placed at the tuple level. With each tuple traveling back and forth from the router and having the route calculated at the end of each operation, the cost can be rather excessive. This process is also expensive when trying complex routing decisions, and for this reason within said adaptive routing systems [7] the routing logic itself tends to be rather simplistic and purely heuristic.

3 SPE Evaluation

3.1 Requirements of System Processing Engine (SPE)

The overall project goal is to optimize the execution of multiple simultaneous queries over streaming data in order to allow for faster real-time data analysis. Given the nature of the SDAF system, a number of queries have the potential to be composed of similar operators over the same data. This is the area in which our research will focus; the ability to share operators between queries instead of processing queries separately. The hypothesis is that this will reduce execution cost and the aim is that these improvements will be noticeable when a number of queries are being simultaneously executed.

A few important parts of this problem are comparing queries and recognizing similar sub-queries to share, adding and removing queries (dynamic query plans), collecting all the data while combining queries (saving state), to optimize the flow of data. The overall requirements for query optimization for this project are as follows:

- ability to add custom operators for personalized monitoring applications
- ability to share data among query plans
- dynamic query plans
    - saving state during query plan changes
        - to prevent data loss
    - develop metrics to discover and evaluate sharing potential
- handle the ability for reordering of operators

The challenges for each individual requirement and a more in-depth understanding of each concept are given in the next section.

3.2 Requirements: In-depth Analysis

Referring back to the overview of the requirements, there are a number of concepts to implement with this project. Each concept is fully explored and possible problems and their solutions are looked into in this section.

3.2.1 Custom Operators

By using either existing available operators or coding new methods, the system needs the support to create personalized operators that can be plugged in any query. For example, an operator, called BlueTankSelect, which will select a tuple in the stream that is a blue tank located in a desired location. This would be a custom operator assuming it does not already exist as functionality within the given system.

Custom operators have a number of very practical applications. First, it will be easier to call one simple operator rather than a series of operators to do the same thing, similar to macros. The code itself will have a series of operators executing, but when searching for patterns and query plans it will be easier to have a simple call of BlueTankSelect rather than all of the other background calls.

Possible problems will be the ease of editing the SPE and adding new operators. The problem seems very simple and straightforward, but if the code is too scrambled and unstructured it can be difficult to understand the proper steps needed to correctly create and implement an operator into the system. Other complications revolve around what expensive power is at our disposal to construct these custom operators. It is simple to use

a series of predefined operators to create a new operator but the SPE needs to be explored to see if new useful custom operators can be created with some ease.

3.2.2 Sharing Data among Queries

The goal is to take data used in one query plan and be able to use this data in other query plans. Then, if possible, reuse the same stream for multiple queries. When multiple query windows are created, some of the windows may have overlapping parts. In order to capture all of the data properly, the stream would need to be shared among the customer query plans. Sharing data would allow the query plans (query plan A' and query plan B') to incorporate the same stream data with AB being the overlapping data of the two query plans (Figure 5).

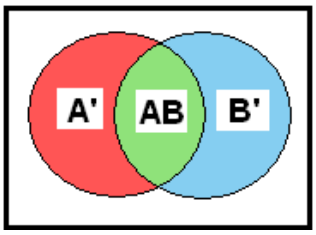


Figure 5: Sharing

If data is being shared, it is important that the data is passing through all of the query plans simultaneously. If any of the data is not synchronized, this can cause further complications as we may need to store some historical data and refer to it later.

3.2.3 Dynamic Query Plans

Query plans correspond to a pipeline of operations to perform on a particular stream. However, if several query plans are combined to work together, they may work more efficiently. Clearly, one would need the technology to change this set of query plans to a new and better shared plan. Aside from creating a new shared query plan and changing the current ones, a query may also need to be removed and the still existing queries must reflect the change.

The idea of dynamic query plans allows the plans to be adjusted on the spot. If the query plans could not be adjusted, there would not be any way to optimize the overall streaming system efficiently. Overall, the SPE needs to be able to identify patterns between queries and, if there are any similarities, a new query plan needs to be created and established. Adapting to the changes for adding, removing, and modifying queries allows the system to be much more efficient.

One of the big problems with adjusting the query plans is to make sure that no data is lost. While in the process of modifying the current query plans to create a new plan, the current data streaming through must also be tracked. Otherwise, there is going to be a gap in the data that can be crucial depending on the circumstances. If a streaming query is removed, the original state of the previous query plans must now be changed back to their original form before sharing. This issue needs to be addressed since the query plan is now ineffective with one of the streaming queries removed.

3.2.4 Saving State of Query Plans

While the query plans are adjusted to make new shared query plans, the original unshared state of each query plan must be retained somewhere. The time to construct the new query plan based off the other query plans would allow for data loss if none of the original query plans are executing correctly. Therefore, a query plan state must be recorded to prevent data loss. The saving states of these queries must allow for query modifications to take place without worrying about removing the original unshared query plan from existence.

The benefit with saving the states of query plans is that no tuple data will be lost. By having the original unshared query plans backlogged, as the new query plan is constructed the original query plans can continue to execute. Also, if a stream query is deleted and the combined query needs to revert back to the original unshared query plans, saving the original states allows for this process to be a bit easier.

The main difficulty with implementing a saving state revolves around the idea of where to store these query states. On top of that, the SPE needs to allow for any number of query states to be stored at one time.

3.2.5 Comparing Metrics

A cost model is based on a structured set of metrics used to help determine if a particular query plan is optimized. This estimator can compare new constructed query plans to see if they are effective or not. If a new query plan is structured based off of a previous query plan, it would be helpful to know if the new plan is indeed more efficient. An optimizer should have a set of metrics on which to judge the new query plan. This will prevent time loss if a new query plan is determined to be ineffective.

A simple question to ask is, “what types of metrics will be used to determine if a query is optimal?” There needs to be some sort of cost model which will be constructed carefully to account for all of the variables. This model can be used to determine if it is worthwhile or not to construct a new query plan.

3.2.6 Reordering

The goal is to change the query plan based on reordering operators and switching parts of the plan around. Rather than going through the effort of completely changing the query plan, a change of some local ordering of operators may sometimes be effective. On top of that, we may need to establish a different ordering to enable sharing. New queries coming into the SPE can be reordered into a universal structure that can be altered and modified easier. If all of the filters are moved to the front of the plan, for example, sharing query plans and operators can be easier.

Determining what order the query plan should be takes time. There are reasons for having the query plans structured in many different ways. Aside from that, the basic principle of searching through and properly rearranging the plan in a way can be worthwhile.

3.2.7 Performance Test

A valid test set provides the ability to evaluate the performance of the system, measuring its effectiveness. The performance test allows us to compare multiple approaches in the system and determine their effectiveness. Most basically, these tests should allow the justification of running the queries in a shared operator system over a standard one, if this is the case. Secondly, these tests should allow us to compare incremental changes in the system and algorithms used. Getting a good coverage in simulating real world conditions is a non-trivial exercise. Although this process is aided by data sets to be provided by MITRE.

3.3 Stream Engine Evaluation

3.3.1 Calder

Calder is a continuous query grid service that brings data streams together to merge them as a single coherent data resource [8]. Calder extends OGSA-DAI and dQUOB and was created by the Distributed Data Everywhere Lab out of the Indiana University Computer Science Department. The Calder architecture has two subsystems of Data Management and Query Processing. The Calder system uses stream loading and provides a framework for timely research issues in stream processing.

3.3.2 CAPE

CAPE is WPI's streaming database system entitled Constraint-exploiting Adaptive Processing Engine. The aim of CAPE is "to provide novel techniques for processing large numbers of concurrent continuous queries with required Quality of Service" [9]. CAPE beyond the core stream engine infrastructure, the CAPE project explores a number of projects that exploits dynamic metadata at many different levels. The engine is written in Java and features the ability to develop reactive operators with configurable execution logic. CAPE also incorporates adaptive operator scheduling. CAPE is explained further in Section 5.

3.3.3 Coral8

Coral8 is a commercial but free for development purposes Complex Event Processing engine and development studio, available as an installable program for Windows and Linux [10]. This program can handle processing requiring filtering, aggregation, multi-stream correlation, event pattern matching, and other complex processing.

Coral8 is comprised of the Coral8 Server, which is the software engine that actually processes and correlates data streams at runtime, and the Coral8 Studio, described by Coral8 as "the graphical environment for defining streams, adapters, CCL queries and CCL modules, as well as for managing CCL projects at runtime". This includes the compiler for CCL and debugging tools. Along with this comes a host of adapters for input and output from the server, and a Software Development Kit for creating your own adapters.

As for the specifics of implementation, sliding, counting, and jumping windows are available, which may be time-based or row-based are harder to define, with the ability to retain, uniquely identify and remove rows based on one or more column values. There are basic operators such as grouping, aggregation, sorting, stream filtering, rate limiting, etc. The syntax is an SQL-like query language called CCL.

The join between two windows is similar to a join between two tables, except that it is executing continuously. Joiners also exist for stream-to-window correlations and stream-to-historical data correlations. The join condition can be arbitrarily complex. Inner joins as well as left, right, and full outer joins are all also supported. There is also a GroupBy clause that "allows applications to distinguish states by individual column definitions".

Coral8 is an event pattern matching system. Queries and subqueries can also be made to historical databases to check data against current streams. Dynamic queries are also available. As for adapters, both input and output adapters can be written through the built

in SDK, and run either as in-process or as separate processes to the engine. Many adapters are already written, the relevant ones are: JMS, RFC/RPC (SOAP and custom plug-ins), SNMP, SOAP/XML, Files (CSV, XML, Binary), Sockets (CSV, XML, Binary), E-mail, and RSS/ATOM

The Coral8 home page describes many of the useful features that it has, with extensive documentation for users. The documentation provided by this website shows tutorials for setting up the engine on various platforms, and information on running, monitoring, and administering the software. There are even documents showcasing various features of Coral8 and how they might be best used. There is an Eclipse plugin available which is open-source, but the Coral8 software itself is not. Nevertheless, Coral8 seems to be freely accessible, and has documentation on all of its features.

3.3.4 Esper

Esper is an open-source SPE project and a component for CEP and ESP applications, available for Java as Esper, and for .NET as NEsper. Esper and NEsper enable rapid development of applications that process large volumes of incoming messages or events [11]. Esper and NEsper filter and analyze events in various ways, and respond to conditions of interest in real-time.

Esper breaks down the main features into a handful of categories ranging from Event Stream Processing, Event Pattern Matching, Event Representations, etc. Under the Event Stream Processing, the queue can be time, length, interval, and even window based. There are basic operators such as grouping, aggregation, sorting, etc. The syntax is an

SQL-like query language but there are no specifics as to what it is called. Inner and outer joins are possible on an unlimited number of streams or windows as well, which can be useful for combining query windows. The Event Pattern Matching provides logical and temporal correlation. Specific events can have listeners provided to see if a certain pattern is being executed. This can allow for common patterns to be stated and the system can automatically detect and prepare for these patterns. More specifically towards the representation, Esper supports event-type inheritance as well as polymorphism provided by the Java Language. It features an event-driven architecture which supports reactions to event creation, detection, and so forth. As for adapters, Esper features CSV input adapters and reads comma-separated value formats. The JMS input/output adapters are based on the Spring JMS templates. Another useful feature is that as of release 1.5, Esper is multithread-safe and there can by safe multithreaded sends of events to the Esper engine. The pluggable architecture of this SPE allows for event pattern, event stream analysis, and other forms of plug-ins.

The Esper home page describes many of the useful features that it has. The best thing going for Esper is the documentation. Each SPE has been carefully analyzed by our team; however, Esper seems worthwhile to pursue to at least see if it is practical for our situation. The documentation provided by this website shows tutorials for setting up the engine for both Java and .NET. There are even case studies provided which showcase some of the nicer features of Esper. The fact that Esper is open-source allows for future additions including customizable operators depending on how difficult the

implementation is. The biggest question with any of these SPE's is if they are manageable and relatively easy to work with.

While Esper is noted as being an open-source project, there are many hurdles to cross. The user help through the e-mail aliases provide 24/7 help with relatively immediate responses. This is very useful for working with Esper; however, not all of the questions can be answered. While trying to modify the existing code to reorder the filters as we desire, the Esper team had already adapted one fixed method to sort all of the query filters into a predetermined path. The problem is that the provided help from the Esper team did not understand where this modification in the code was and it could not be turned off. Our goal was to implement our own form of filter optimization and while Esper already had a working sorting method, the practicality of not being able to alter the engine to our needs was deterring. Esper's documentation and tech support seems to allow for end users on a high level to get what they need. When the goal is to modify the engine specifically, the documentation is irrelevant and the tech support becomes quickly puzzled with the objective.

3.3.5 RiverGlass

It is difficult, because of a lack of documentation or elaboration on their website, to determine exactly what RiverGlass's products are meant to do. However, their site describes their primary product as one designed to crawl the web and perform autonomous searches and prepare reports on this data [12]. Their other advertised product appears to add onto this functionality, enabling the data coming in from their Recon

program to be viewed in stream format and for data to be processed from this particular stream. It does not appear to be a generalized stream processing engine.

It is impossible to describe the architectural features of Riverglass's software, as they provide no online documentation except for their advertisement pages. Nevertheless, the scope of this software appears to be outside of the scope of this overview, so it is unnecessary. RiverGlass's analytic tools might be useful for web searching and analysis, but not for our needs. They don't give enough information to make the purchase of their commercial software appetizing.

3.3.6 STREAM

STREAM is a research project on SPEs from Stanford University [13]. The STREAM project officially wrapped up on January 2006, however the source code and other information is still available on the STREAM project website. STREAM's core system is implemented in C++.

STREAM focuses on five main issues that arrive with streaming systems as compared with traditional relational databases; streaming semantics and language, scheduling query plans to reduce resource usage, adapting to changing nature of data, quality of service, and the ability to monitor long running queries. To address the first issue, like other SPEs STREAM created an extension to the SQL language, called the Continuous Query Language (CQL). CQL is designed around the idea of three types of operations going from stream to relational semantics and relational to stream and the standard relation to relational. Windowing (using a sliding window) in CQL is a stream to relation style

operation with the ability to be specified by time intervals, tuple count, through a partitioning manner. CQL provides three operations to transition from relational semantics to streaming; creating an insert stream, delete stream, or relation stream.

STREAM handles the issues of scheduling and adaptivity through the use of internal techniques and optimizations. Queues as with many SPEs are utilized in stream to store the tuples for operators in a query plan. In addition STREAM also uses synopsis with advanced sharing algorithms, to reduced number of synopsis needed along with amount of resources consumed to facilitate temporary data storage. In reference to query plan optimization STREAM utilizes a scheduler, which executes operators following a chain-scheduling algorithm. STREAM also uses constraints.

Quality of service features offered by STREAM are load shedding. STREAM also supplies the user with a GUI for system administration and monitoring. This GUI allows for dynamic adjustment of different attributes of the system such as different operators queue sizes.

STREAM allows for custom operators. STREAM also has a small set of advanced features, which include support for distributed systems, real-time output, and tools for handling data revision. STREAM is an adequate SPE, providing all the necessary basic functionalities with some interesting additions and differences from other standard SPEs. One of the main issues STREAM was not chosen is that the project was discontinued in January 2006.

3.4 The Chosen Engine

With a number of Engines to choose from, the initial decision was to narrow down to three systems normally. From the three selected systems, each one was researched. One was chosen to work with. At first, Esper was the clear winner. While Esper had a tremendous amount of documentation and a surprisingly quick responding tech support team, the inability to alter and modify the engine to our needs was the largest issue. With Esper as a failure, CAPE was the next in line to work with.

The appeal towards CAPE involved a number of logical reasons. With the developers of the system being from WPI, problems and technical issues can be dealt with in person with clear and concise feedback from several individuals. The structure of CAPE is overwhelming at first; however, sitting down with some of the developers helped to explain where all of the pertinent classes are located. The biggest benefit with CAPE is the ability to actually modify the code to adhere to our projects needs. Since the NASSQ system incorporates dynamic as well as static routing principles, our project required a flexible engine with user support; CAPE offered both. CAPE already had both a sub-system with both static routing and dynamic routing.

4 The Approach: Non-duplicative Approach to Sharing (NASSQ)

Section 4 covers the description and implementation of our system. The Non-duplicative Approach to Sharing between Streamed Queries (NASSQ) is the name of our project which blends both static and adaptive routing strategies.

4.1 Description

NASSQ is the technique designed to take advantage of the benefits offered in static/dynamic routing strategies. First, the optimizer generates full routing instructions for the execution plan. The routing instructions are referred to as a routing tree. They indicate how the tuples pass through the execution plan. Rather than the system allowing tuples to travel individually, batches of tuples[1] are accumulated before being sent off to the routing tree assigner. Once full, the batch proceeds towards the tree generation assembler who associates the path with the batch. At the operator level (we assume Boolean operators here), each batch is broken down into two separate batches: one for tuples that pass the operator and one for tuples that fail. This divide is encoded into the routing tree execution plan, indicating the conditional path for each type of batch. A promising attribute of the NASSQ system is that the tuples in the system are typically not duplicated thanks to the routing tree structure. Along with this, only one instance of each operator exists while references to that operator allow the batches to travel through the operators. The NASSQ system idea of combining the overall benefits from two different strategies results in a new approach for streaming databases.

[1] This idea was inspired by QueryMesh, where a batch accumulates a number of tuples before it is sent through the system [14].

4.1.1 NASSQ Routing Tree: Definition and Construction

The routing tree provides the batch of tuples with an optimized path determined by three different statistics. First a list of the operators for each query is created. With this list, data can be collected to see how to devise an optimal tree structure. With the list of operators constructed, it is passed into a recursive tree building function. There are particular weights added to three statistics on each operator throughout the building of the tree. Those three statistics are the per-tuple execution cost, the selectivity, and the sharing between the queries. Each statistic can be gathered through the course of testing the needs of the system. With MITRE and the SDAF system, the selectivity and other statistics would reflect the needs of MITRE. Each of the three statistics carries a particular weight. The weight of each operator is calculated and the list of operators and their weights is evaluated to determine which operator to process first.

The construction of the Routing Tree list begins by using the structured list of operators. The first operator of the structured list is placed in a separate routing tree list while the rest of the list carries through a different process. The operator that was chosen is then removed from the list. The new operator list is duplicated; one to proceed as the pass list and one to proceed as the fail list. The fail list is evaluated more to determine which queries have failed due to the failure of that particular operator. Any operators remaining in the fail list that have no bearing any more are then removed from the fail list. In essence, the fail list of operators is stripped down to only the necessary operators that should remain. The recursive functionality of the tree building function is called on both of the newly constructed lists and the process continues. This process carries onward

until the entire tree is constructed. With the routing tree generated, two fake tuples are created; a head tuple and a tail tuple. The head contains the vital information on which operator to proceed towards next, by associating the routing tree with each batch, while the tail tuple simply marks the end of the batch. The head tuple is placed at the beginning of a tuple batch and the batch is then filled with the desired amount of tuples. When the tuple batch is full, the tail is placed at the end of the tuple batch and proceeds towards the first operator.

4.1.2 NASSQ Operator Process

When a batch of tuples arrives at an operator, the data in the input queue is processed by the operator (Figure 6). Two buffers that function like queues are created in the operator to separate the tuples that have passed and failed this operator respectively. The head tuple of the batch is analyzed to determine which path to follow depending on the pass or failure of the operator. The pass path head meta tuple is placed in the pass queue while the failure path head meta tuple is placed in the fail queue. Once the head tuples are in place, the processing of the actual tuples commences.

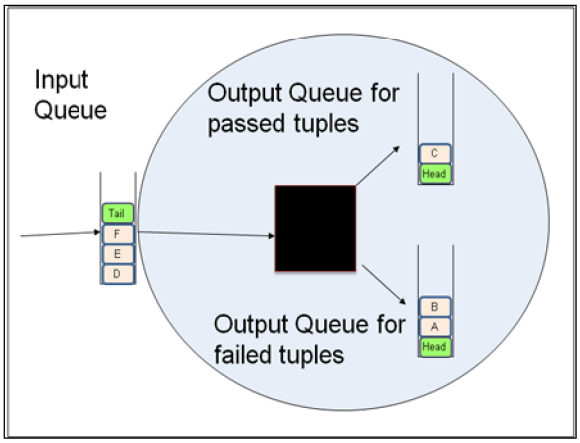


Figure 6: NASSQ Operator Diagram

Each tuple is analyzed by the operator and placed in their respective output queues. The process continues until the entire input batch is emptied; determined by when the tail tuple has been reached. When the tail tuple is reached, it is passed into both queues to

mark the end of those particular queues. Each of the two queues now has a separate path to travel. The content of the two queries is placed into the input queries of the respective operators.

4.1.3 NASSQ Process Layout

As discussed previously, the layout of the system is a tree-like structure. At each operator that a batch of tuples arrives at, a branching occurs separating the batch of tuples by a pass and fail procedure. No tuples are ever duplicated since the tree expands fully to encompass all possible paths[2]. The left branches in Figure 7 represent the pass path while the right relates to failure. In this particular example, operator 4 is found in all three of the queries. Because of this, if any tuple fails operator 4 it fails all of the queries and is dropped from the system. The Ø symbol represents the tuples being dropped from the system. While there may be a number of duplicate operators in a system, it is important to note that all of these are references to the actual operator objects and not replications of genuine operators.

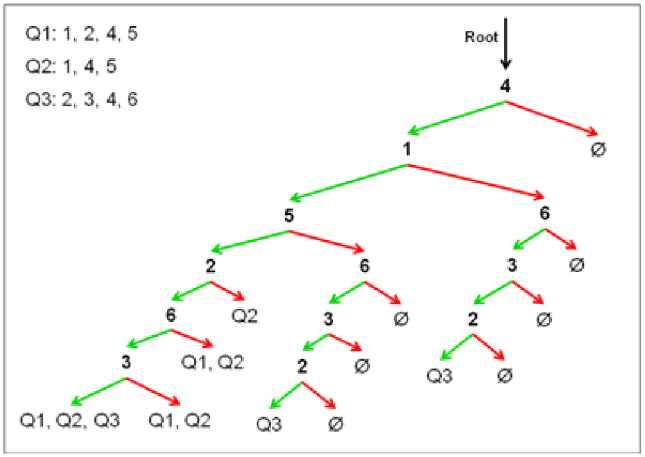


Figure 7: NASSQ Routing Tree

Throughout the system layout, while a batch of tuples may fail a particular operator, unlike the static strategy, a failure does not necessarily mean the tuples are dropped from

[2] The duplication of tuples can still be done if desired

the system. If the fail path continues to have operators, then the tuple will continue to be processed. This approach prevents the tuple duplication and thus reduces memory usage.

4.1.4 NASSQ Benefits

Static Routing	NASSQ	Dynamic Routing
•One precoded tree of routes for whole runtime	•Path is dynamically decided during runtime	•Path is dynamically decided during runtime
•Routing heuristics can be sophisticated	•Routing heuristics can be sophisticated	•Expensive to do complex routing decisions
•Must rebuild path if queries change	•Handles dynamic query changes quickly	•Handles dynamic query changes quickly
•Efficient execution structure	•Efficient execution structure	•Centralized router to return to after each operator
•Low overhead per tuple	•Low overhead per tuple	•High overhead per tuple

Figure 8: Static/NASSQ/Dynamic Overview

In comparison to the static and dynamic routing strategies, the NASSQ approach offers a number of benefits. Figure 8 illustrates some of the pros and cons for static and dynamic routing while showcasing how the NASSQ system takes advantages of the benefits from both strategies. When a new tuple batch is created, the routing tree execution plan is attached to it. The path could also be determined at runtime, if deemed necessary, thus this method is highly adaptive. With the help of the recursive tree generation function, the routing heuristics and tree generation algorithms can be sophisticated. A tuple batch can be large or small depending on the application which allows for an even better flow of control.

When queries are added or removed from the system, the routing tree generator is able to construct paths for the incoming tuple batches to reflect those changes. With a static system, the execution plan must be rebuilt; however, the NASSQ system benefits like the dynamic strategy by adapting quickly to query changes. The routing tree allows for a controlled execution structure. The problem with the dynamic strategy is the constant travel back and forth from the router to an operator. This travel causes the dynamic system to incorporate a higher overhead since this happens at the tuple level. The NASSQ system's instead encompasses a defined and structured execution plan, so no decision making needs to be done at run-time.

4.2 The Data Flow

To understand the process of tuples traveling through the NASSQ system, there are a series of flow charts that highlight the particular areas of the NASSQ system discussed earlier. The flow charts help to explain a bit more detail.

4.2.1 Tuple Batch

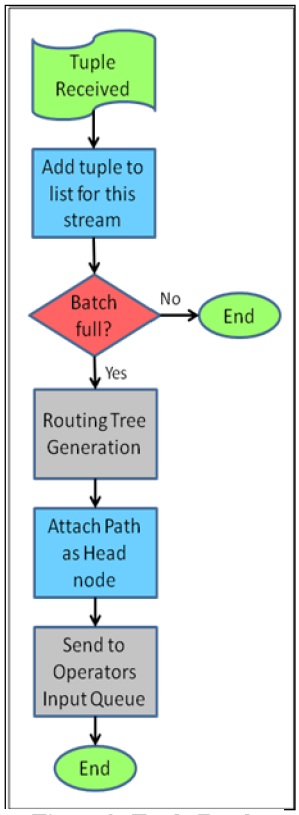


Figure 9: Tuple Batch

The tuple batch construction begins by filling the batch with tuples (Figure 9). As a tuple is introduced into the system, it is added to the batch. A check is made to see if the batch is full whether it is the tuple capacity or a time limit. If the batch is full, the routing tree is attached and the batches are released into the system. The process continues to receive tuples from the streams and places them inside of the next batch.

The entire tree route is placed inside of a head node which is attached to the front of the tuple batch queue. With the head node attached, the batch is sent into the system for processing. The first operator to hit is determined by the tree processing that is by its head tuple.

4.2.2 Routing Tree Generation

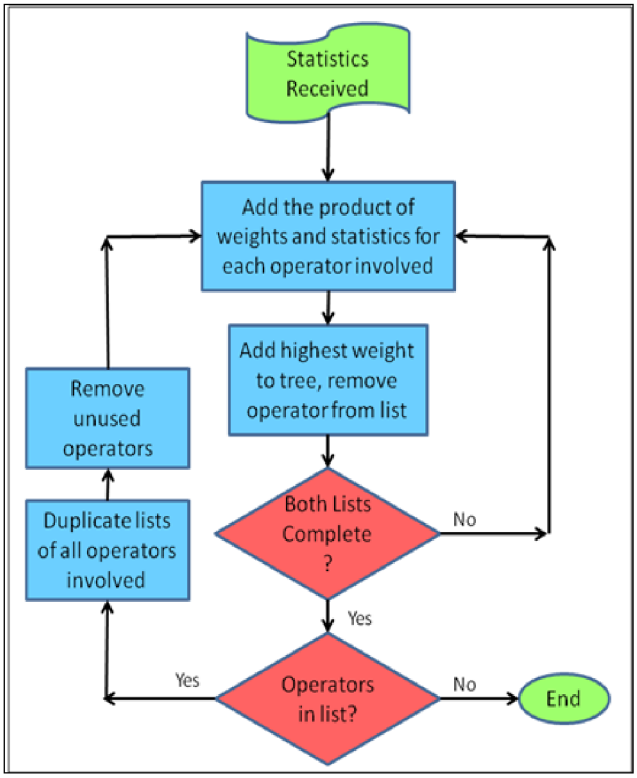


Figure 10: Routing Tree Generation

The routing tree generation process produces a path for the batch to travel. The routing tree generation process begins by receiving the statistics to be analyzed. The weights of the heuristics are calculated and summed together for each individual operator in the list. The operator with the highest weight is added to the tree and removed from the operator list. A check is made to see if both the pass and fail lists are complete.

If there are no more operators, the tree generation is finished; otherwise, the process continues by duplicating the list. This is necessary since the structure of the tree focuses on a pass and fail path. Now that one operator has already been decided, the breakdown has a pass and fail path. The two lists created now cater towards that objective. Unused operators are removed from both the pass and fail lists. This more specifically influences the fail list since the previous operator failed. The query that contains that operator has now failed as well, so the remaining operators in that query that are not used in other queries are now unnecessary and must be removed from the fail list. In the previous examples operator 4 was chosen as the first operator. Since operator 4 is found in all three of the queries, all of the operators in the fail list would be removed; therefore, causing the tree generation to drop the tuple batch that fails operator 4 (Figure 11).

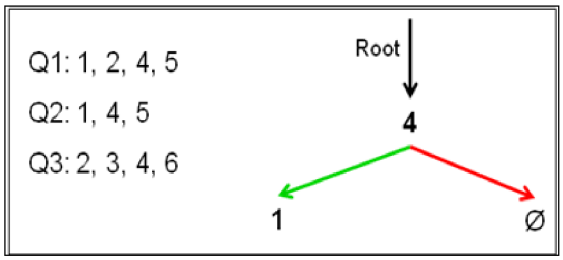


Figure 11: Pass/Fail List Structure

Each of the two lists (pass and fail) is then processed individually. The operators are weighed again and the next best operator is decided upon. The following operators are evaluated individually for each list (both pass and fail lists), so the weights, which are calculated prior to entering into the Routing Tree Generation process, can be different in each case. When all of the operators are placed in both lists, the routing tree generation is complete and a tree is created.

4.2.3 Operator Processing Logic

The operator flow begins with the input queue which is the batch of tuples. The tuples are processed upon arrival. Two queues, a pass and a fail queue, are constructed to handle the output. The first tuple of the batch is de-queued and checked. First, we determine if the tuple is a head tuple. If this is the case, the path information from the head tuple is added to each of the output queues as seen in Section 4.1.2. The current operator being processed is removed from the path and the pass path is placed in the pass output queue and vice versa for the fail path.

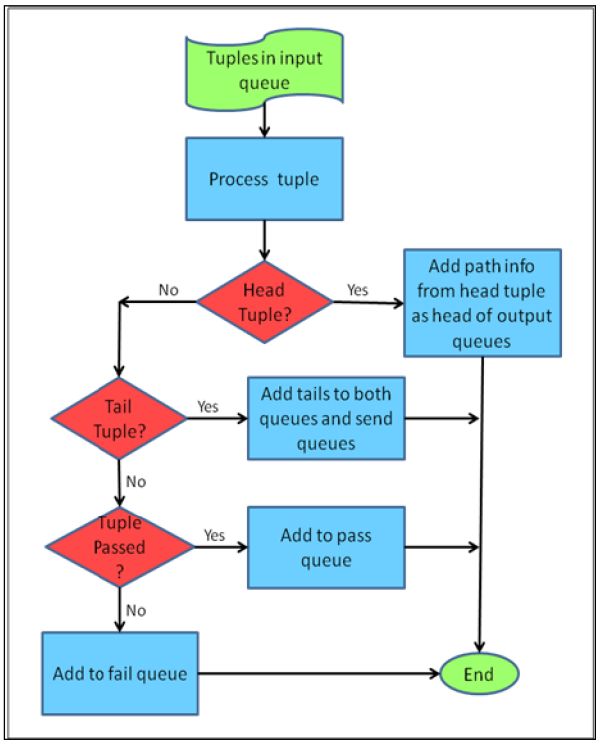


Figure 12: Operator Process on the Tuple

If the processed tuple is not a head tuple, the next check is to see if the tuple is a tail tuple. A tail tuple causes the output queues to be capped off with a tail tuple. The two queues (pass and fail) are then sent on their prospective paths to proceed to their respective next operator. If the processed tuple is not a tail, the tuple is then in fact a regular valid data tuple and is processed to determine if it passes or fails the query logic. A tuple that passes the operator is added to the pass queue while a failed tuple gets sent to the fail queue.

This process happens for each tuple in the input queue of the current operator. The tail tuple signifies an end to the process.

4.3 Design of NASSQ

The structure of the system is built to encompass adaptivity. The logic behind batches instead of dealing with tuples individually is to allow for the adaptive structure of a dynamic plan without having the complete overhead relate to each tuple itself. The cost to have a customizable plan for each tuple as seen in the Eddies example is high; however, having a set strategy plan for all tuples, as seen in the static strategy, will not be optimal for all tuples. With a batch, the convenience of altering the execution plan dynamically is available without the overhead in Eddies. The developer can set the size of each batch which can be custom tuned to fit the desired application.

As for the tree generation, a path calculated based on the current operators in the queries allows for adaptivity when adding and removing queries is involved. If a query is removed from the system, particular operators may not be needed. The incoming tuples that are organized into a new batch conversely incorporate this change because the routing tree generation function can simply be re-executed. Along with this, creating head and tail tuples helps us to isolate each batch; thereby limiting the amount of work an operator must do. Similar to a network routing issue, it is easier to have lightweight clients and a heavy server with most of the computation. In terms of NASSQ, the operators can be viewed as clients in a way that they do not worry about where to send the output queues or how to determine the next path. All the operators are concerned

about is whether or not a tuple passes or fails and then places the tuple into the appropriate output queue. Less overhead on the operator level allows for faster processing at each individual operator.

A final aspect to the design is where the output ends up. A simple fail queue that is dropped from the system does just that; it *drops* from the system. There is no concern with what to do with it. When an output queue reaches a point where the tuples have passed some queries, the tuples are written to files that reflect that particular query. Each output file is designed to be named appropriately with the particular query it has passed. As the tree continues downward, some of the tuples pass multiple queries (as seen in Figure 7). In these scenarios, the tuples are written to all of the output files that it is associated with. The example in Figure 7 where a batch of tuples passes all of the operators (travels down the entire green path) will end up being written to the files associated with all three queries.

5 Implementing NASSQ on CAPE

5.1 Insights into CAPE

CAPE is a customizable framework for generating, processing, and outputting streaming data. It is the centerpiece of many projects at WPI, including a shared-operator static system, an implementation of a dynamic system known as Eddies, multiple projects surrounding different types of operators (FireStream, STeM, etc.), and clustered systems [15, 16]. Depending on the project involved, different parts may be utilized, but in general, the CAPE system provides an infrastructure for receiving tuples, some method of path generation, and data structures for representing operators and query plans. XML files are used to define the system configuration settings, incoming stream schemas, queries to run on the incoming data, and the operators these queries contain.

5.2 QueryMesh

One project in particular that made CAPE useful to our project's purposes was QueryMesh [14]. QueryMesh is a system design for stream processors that work by analyzing the incoming streaming data and generating a path that takes into account specifics about the current data and its content.

QueryMesh consists in part of a tuple receiver, called the Classifier, that reads in incoming stream data, analyses the tuples, and boxes up tuples that are expected to pass similar sets of operators. For example, scanning for vehicles in a set of tuples taken from a barren wasteland might be better done by scanning first for radio signals, then checking for motion, whereas for ones from a military compound perhaps the radio signals would

be less selective so that the motion tracking should be done first. The Classifier then treats the boxed tuples as a dynamic routing system and generates a path for the batches. During the operator running, statistics are being taken on the various types of tuples and what data they're passing and failing to use in the algorithms for future tuples.

QueryMesh works well for our system because the concept of tuple packaging is already fully implemented, so we can focus on coding the adaptation to multiple queries, the routing tree for static path generation.

5.3 Key Packages

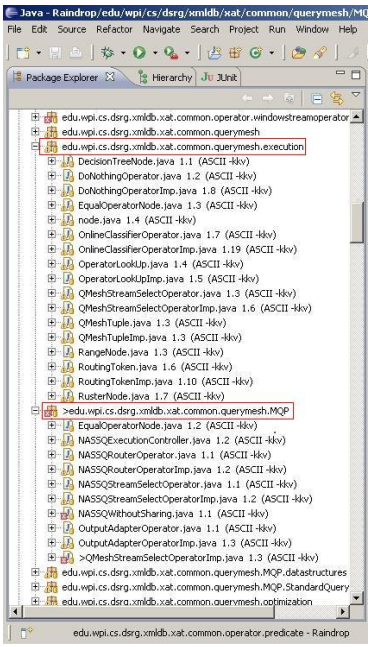


Figure 13: Key Packages

Originally, the package edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.execution enclosed key files such as the ExecutionController, the Classifier, and the StreamSelectOperator. In order to incorporate these files, a separate location was created to isolate the MQP project located at edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP. This particular folder contains the NASSQ ExecutionController, Router, and Operators. The other folder (edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.datastructures) holds the files for the routing tree as well as the batch of tuples called RusterNode. These two packages contained the implemented java files of the NASSQ system. The XML data files for the

NASSQ system were created in the resources.QueryMesh.MQP folder. The XML files were used for various tests we ran on the system, as well as the configuration XML files, and the stream files, which are pipe-separated value tuples imported via CAPE's stream creator.

5.4 Java Files Implemented

5.4.1 Routing Tree Generation

A RoutingTree is generated via the algorithm in NASSQRouterOperator.buildRoute(). Currently, we plugged in a very simple algorithm, namely to simply get the next unchecked operator with the most sharing. In the future, this could be changed by rewriting the buildRoute() method. Different algorithmic approaches can be implemented in this file if the developer chooses to do so.

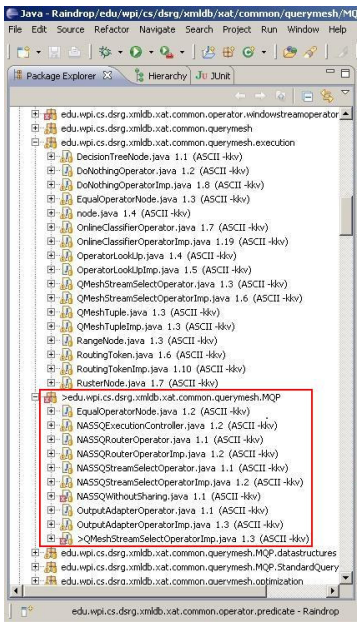


Figure 14: Java Files Implemented

5.4.2 Data Preparation

When a tuple comes into the input stream, NASSQExecutionController forwards it to NASSQRouterOperator, which collects tuples until a suitably large group of tuples arrives. The philosophy to determine the appropriate number of tuples was not studied and so a batch consisting of several hundred tuples was used. At this point it places a RoutingTree token at the head of the list of tuples, and an EndNode is placed at the rear. Then it sends the produced batch of tuples to the first operator on the routing tree.

5.4.3 Execution

When a RoutingTree token (treated as a tuple) comes into an operator, it's placed in the input buffer. The tuple is processed and extracted sub-trees off the pass and fail paths respectively are placed into the two pass/fail output queue. Then, as the actual tuples that were batched with it arrive, each will be processed by the operator, and placed in the appropriate output queue. Lastly, an EndTuple will arrive, which simply marks that the batch of tuples has ended and that those tuples can be sent off from the output queues.

After a given tuple batch has traversed through all the applicable operators in its route (as determined by the associated routing tree), the tuple batch proceeds to the java file OutputAdapterOperator. This operator reads from the batches routing token what queries the tuple batch has passed, and reports it to the correct outputs (in our system we choice to utilize a simple text files recording all tuples passed by those queries when testing accuracy of the system or dropping them during performance testing).

5.5 XML Files Implemented

Details of the system configuration file are either computer-specific (as far as where to do logging, data flow speeds, etc), and others are related more to other parts of CAPE that we didn't modify. There was no need to change the format of this file, except to replace the values, such as the query plan file name with those of the tests we were running.

Along with that, there is a configuration file to define what streams exist and what port to listen to, a configuration file to define the stream schemas, and a configuration file to define the operators and queries that will be running.

Of these, the first two were only modified to represent our incoming streams. We did not need to change the format of these files at all, except to replace the values with those of the test we were running.

The last, the Query Plan configuration file, was the only one that received any formatting changes. The change was to add a variable near the end, called “sharing”, that allowed us to toggle whether operators and tuples were reused during execution. This allowed us to test the sharing versus non-sharing alternatives in an otherwise identical system.

As well, in this file, we changed the meaning of a section. In standard QueryMesh, there is only one query, but it can have multiple paths and so the Query Plan allows for multiple operator sets to be recorded for the query. In our system, there are multiple queries, but we don’t define the path, only the set of operators to run.

Due to having many tests and computers on which we did our project, there are many versions of the configuration files; here are some of the examples:

System Configuration Files:

These files elaborate which java files to use and the overall structure of the other resource files used during execution.

Resources.QueryMesh.MQP.systemfiles.BFTSystemConfig.xml
Resources.QueryMesh.MQP.SystemConfigQueryMeshPlan.xml

Stream Reader Configuration Files:

These files host the location of the stream data files as well as how many streams there are going to be in the system and the attributes of the streams.

Resources.QueryMesh.MQP.BFTStreamFeederConfig.xml
Resources.QueryMesh.MQP.CLUSTER_BFTStreamFeederConfig.xml

Stream Schema Configuration Files:

These files detail out the attributes of each individual stream. Each column and property of the stream is listed here by what type of data it represents and the location of the data as far as which column.

Resources.QueryMesh.MQP.BFTStreamGeneratorConfig.xml
Resources.QueryMesh.MQP.queryplans.SingleStreamsConfig.xml
Resources.QueryMesh.MQP.StreamsConfig.xml

Query/Operator Configuration Files:

These files are where the operators in the query plan are outlined and created. Operators such as selects and filters are established and initialized for use during execution.

Resources.QueryMesh.MQP.queryplans.BFTQueryPlan.xml
Resources.QueryMesh.MQP.queryplans.5OP_Low_BFTQueryPlan.xml
Resources.QueryMesh.MQP.queryplans.7OP_High_BFTQueryPlan.xml

6 Testing and Results

6.1 Testing

6.1.1 Benchmarking

In order to properly analyze our results on our system's throughput we need to collect benchmarks for comparison. Proper benchmarking requires an environment in which two conditions can be met; first that processes separate from the SPE have a negligible impact on the results and second a fine time granularity. These two conditions were satisfied using WPI's Linux computing cluster at cs-master.wpi.edu. In this cluster we were able to run each component of our SPE on separate machines, with minimal additional processes running, helping us to ensure that other processes did not impact the results. Also running java on the computing cluster increase the time granularity compared to using a more low-end laptop, providing the accuracy we need to more effectively run the stream generator.

However, using a cluster, some additional amount of set up work is required, as each process has to communicate over a local network, instead of just a local machine. Switching from the windows environment to the Linux environment also required additional work to ensure compatibility, primarily with file work such as reformatting the files based on a different file system structure. We also had to create a set of execution scripts to run on each machine, the different process (see Appendix C for the readme on how to run).

6.1.2 Stream Files

The data that we used to simulate incoming streams was Ground Movement Tracking Indicator (GMTI) data provided by MITRE. This was a 50,000 piece sample data set, which include attributes (dimensions) per tuple, including coordinates, height, and sensor ID. This data was designed by MITRE Corporation to simulate real types of data that could be acquired.

6.1.3 Procedure

One primary measure of the effectiveness of our system was an attempt to measure throughput on the system. In order to due this we cranked up the stream processor to send out tuples at as fast of a rate as possible. We increased the generating speed as we required the input rate to be higher than the output rate, in order to accurately measure the max throughput of a system. We encountered a number of other concerns with our approach, primarily with running out of available main memory accessible by the java virtual machine. When we increased the stream generator to the first speed that was faster than output, on the Standard query system, the system would constantly crash due to memory heap issues, as there would be a buildup in system overhead. In order to remedy this, given that that we could not go to an intermediate speed, we increased the Java heap size to 3 gigabytes of the 4 available on the cluster machines and shortened the length of the experiments depending on the streaming speed, for example when the speed was set to max, we could not stream longer then 7 minutes.

6.2 Testing Decisions

6.2.1 Time Stamping

In order to properly measure the latency of the tuple handling in the system, we had to choose a time-stamping technique to as accurately as possible measure the time. For this we chose to insert the timestamps at the stream generator, before sending them out to the processor. The alternative solution of applying the time stamp once the tuple had entered the system introduced a significant time error due to the systems nature of execution. The single threaded process will just let the tuples sit in the input queue until it is available, before applying a timestamp, which does not allow for an accurate measurement of latency. Even though the timestamps are coming from two separate systems, in the WPI cluster the systems clocks are synchronized and the potential error introduced is negligible in regards to utilizing system side time stamping.

We encountered a problem in our initial attempts at calculating latency, due to the overall numerical size of the measured latency, which cause a precision error, resulting in an unusable value. To reduce the numerical size of this overall latency, we implemented a sampling technique, in which every hundredth (when many operators reported statistics – Standard Cape) or thousandth (when only the output operator reported statistics – NASSQ) tuple was recorded. This allowed us to reduce the numerical size that is associated with summing all recorded values, while continuing to retrieve unbiased measurement. Overall there were still cases where occasionally the latency resulted in an unusable number, typically in a high throughput high latency case, but the technique was overall implemented successfully.

6.2.2 25 Query Measurements

The initial idea behind NASSQ was to develop a system which could handle high tuple counts and high query counts. In order to properly assess whether or not our system functioned to that end, we develop a Query Plan containing 25 queries. This number allows us the ability to test whether or not our system provided improvements to streaming systems that handle a larger number of filtering queries.

6.3 Testing Issues

6.3.1 Academic Code Base

While working with an academic code base we ran into a number of issues.

- Lack of documentation. This made the startup process take significantly longer then we originally planned as well as the process of working through the code. This was partially mitigated by the presence of WPI graduate students familiar with the code base.

6.3.2 Stream Generator

Failure to Stream – When we shifted our system to generate routes before executing the system, an error was introduced in NASSQ, in which after setup was completed, a 20 second process, the stream generator would fail to connect. This problem was corrected how the timeout process was handled, allowing for a delayed startup among the components running on distributed machine.

DistibutionManager.java Line 352

```
NASSQ Version
```

```
for(Iterator it = config.getMachines(); it.hasNext();){
      Machine m = (Machine) it.next();
      Socket s = null;
       while(s==null){
            try {
            s = new Socket(m.getIPAddress(),
                           m.getConnectionListenerPort());
            } catch (SocketException se) {
                  s=null;
                  Thread.sleep(1000);
            }
      }
```

```
Cape Version

for(Iterator it = config.getMachines(); it.hasNext();){
      Machine m = (Machine) it.next();
      Socket s = null;
       long sockettimeout = System.currentTimeMillis() + 15000;
       while(s==null){
            try {
            s = new Socket(m.getIPAddress(),
                           m.getConnectionListenerPort());
            } catch (SocketException se) {
               if(sockettimeout < System.currentTimeMillis()) {
                   System.err.println(
                        "System could not open socket to " +
                         m.getMachineName());
                   System.exit(-53);
               Thread.sleep(1000);
            }
      }
```

Inconsistent Output Generation - After working through a number of issues throughout the testing cycle, we have been unable to overcome the issue of "Inconsistent Streaming Speed" to NASSQ test. In initial rounds of testing there was no noticeable delay when working with the stream generator, however in the final rounds of testing, given the same speed setting, the stream generator sent tuples to the 4 different systems at significantly different speeds. For example in a 10 minute execution, it was observed that the

generator, sent out 5 times as many tuples to NASSQ-NS as it did to NASSQ and 2-4 times as many to standard CAPE implementations as it did to NASSQ-NS. This significantly hindered our ability to measure high throughput rates effectively and will be reflected in the results analysis.

6.3.3 Implementation Limitations

Standard Cape Memory Issues - We observed a number of other concerns with our approach, primarily with memory. When we went to test the maximum throughput rates of the individual systems, we increased the stream generator to the first speed that was faster than output, on the standard CAPE system, the system would constantly crash due to memory heap issues, as there would be a buildup in system overhead. In order to partially alleviate this, given that we could not go to an intermediate speed, we increased the Java heap size and shortened the length of the experiment. This was not a complete fix, however the system still had limitations based on execution length and streaming speed. These concerns however were not an issue that we encountered while running equivalent test in any of the NASSQ variants.

NASSQ Tree Generation Time - When conducting our final test on NASSQ, we ran into a number of issues in regards to tree size. We found that the implementation was not as scalable as originally anticipated, as the overhead, specifically in regards to execution cost, required for generating the routing tree as well as for traversing the tree grew exponentially with the number of queries in a tree. With only two to four queries, we found that the generation time was negligible. However, when working with 29 queries, we found that the tree generation (processed as a recursive function), took up a

significant amount of memory (similar to the CAPE issue, however after generation completed the main bulk of memory for handling the recursive overhead was released and posed no long term execution concerns), required a larger heap size, and took approximately 2 minutes to complete on our testing systems. From this we found tthat the tree needed to be generated prior to system execution. The following is a table of time requirements for tree generation based on subsets of our original 29 Query plan.

10 Queries	20 Queries	25 Queries	29 Queries
< 1 sec	2 secs	9-11 secs	150+ secs

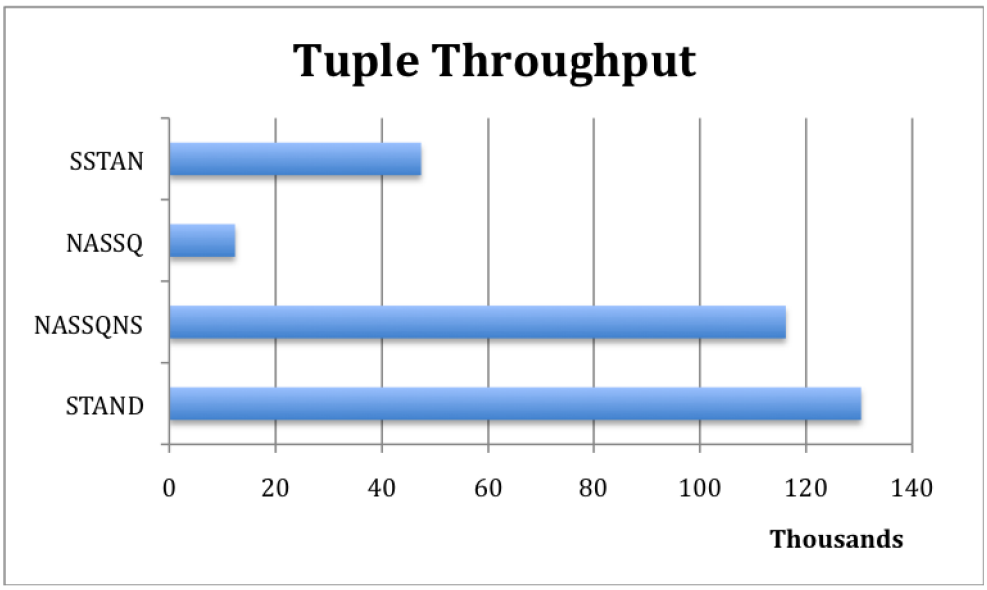
6.4 Data Analysis

6.4.1 Set Up

Our primary testing utilized a 25-query plan, as our system was intended to address both high volumes of data and a high number of queries. In addition, we executed the system for 10 min cycles.

Due to the nature of the issue we experienced with the stream generator, our results analysis will focus primarily on a 25 query implementations of NASSQ, NASSQNS, standard CAPE, and shared standard CAPE with a fast, but not overbearing tuple generation speed (timing variable in the stream.config file was seed = 20, other test that we conducted and looked at used seeds of 4(faster) and 200(slower)) The memory usage and CPU utilization statistics were gathered from the system controlling the WPI cluster.

1.1.1 Throughput

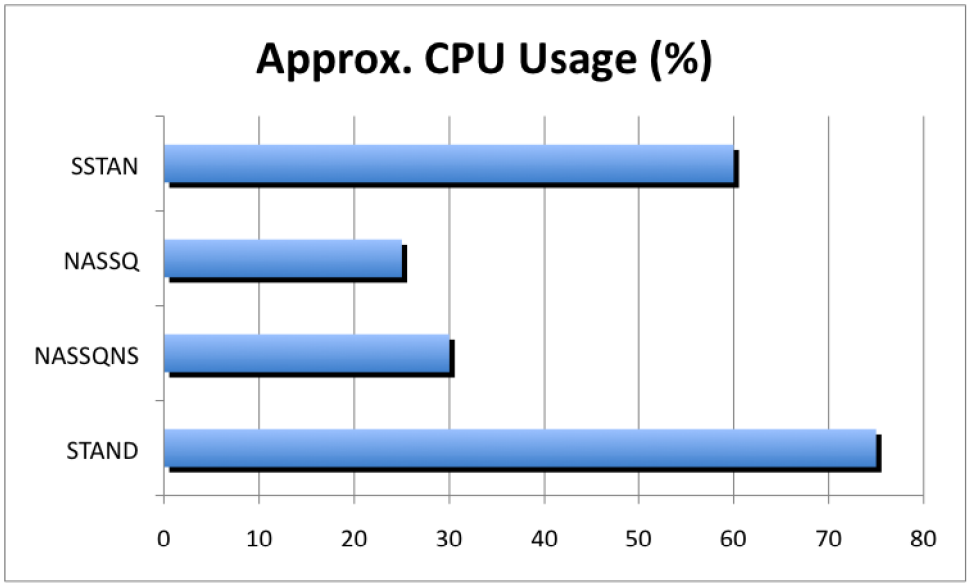


SSTAN – Shared Standard CAPE
NASSQ – Non-Duplicative Approach to Sharing between Streamed Queries
NASSQ – Non-Duplicative Approach to Sharing between Streamed Queries (Unshared)
STAND – Standard CAPE (Unshared)

Figure 15: Tuple Throughput Chart

From this test throughput results we can not conclude that NASSQ provides any benefits to query execution, rather a significantly reduced throughput rate can be observed. However, two things are note-worthy; first that while NASSQ didn't perform neither did shared standard CAPE, which should have preformed better then standard cape, indicating another under lying issue, potentially some additional overhead that we have not yet been able to identify. Second, NASSQNS preformed almost as well as standard cape, indicating a promising area to look at.

1.1.2 System Utilization

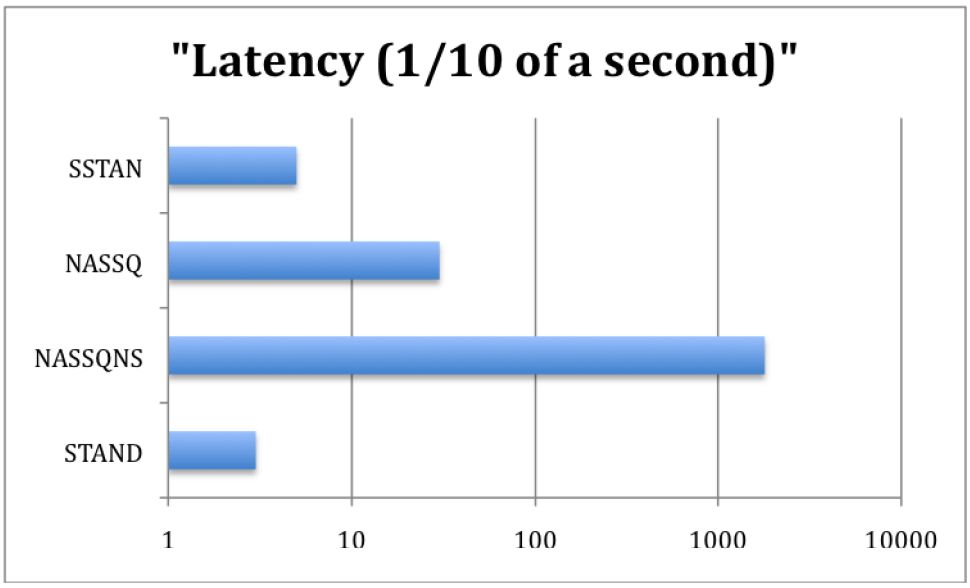


SSTAN – Shared Standard CAPE
NASSQ – Non-Duplicative Approach to Sharing between Streamed Queries
NASSQ – Non-Duplicative Approach to Sharing between Streamed Queries (Unshared)
STAND – Standard CAPE (Unshared)

Figure 16: Approximate CPU Usage Chart

A much more promising result is that NASSQ had a lower CPU overhead in comparision to the two alternatives. You would expect both NASSQ and shared CAPE to have a reduced overhead than their counterparts because of their reduced throughput rates and reduced operator counts. The fact that NASSQ-NS with a throughput almost equivalent to CAPE, but a significantly reduced CPU Overhead is promising, indicating a much more effective handling of the data. We attribute these results to the reduced number of operators and the benefits that come with the associated reductions in the amount of switching between operators.

1.1.3 Latency

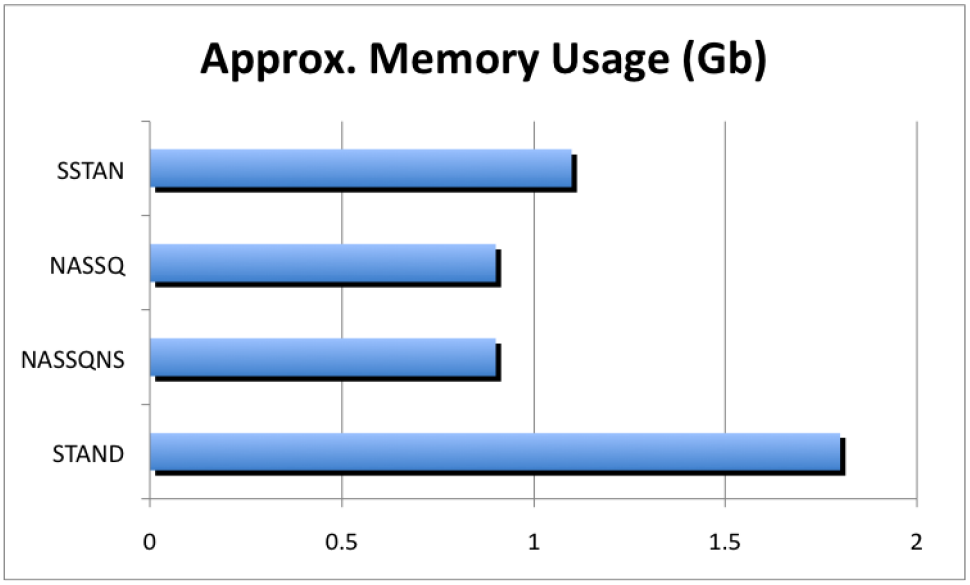


SSTAN – Shared Standard CAPE
NASSQ – Non-Duplicative Approach to Sharing between Streamed Queries
NASSQ – Non-Duplicative Approach to Sharing between Streamed Queries (Unshared)
STAND – Standard CAPE (Unshared)

Figure 17: Latency Chart

A less promising but expected result was increased latency times in NASSQ. These results lead to our later suggestion to change the scheduling algorithm, as even though there is an expected increase in Latency, the value should not increase by 200 fold times.

1.1.4 Memory Usage



SSTAN – Shared Standard CAPE
NASSQ – Non-Duplicative Approach to Sharing between Streamed Queries
NASSQ – Non-Duplicative Approach to Sharing between Streamed Queries (Unshared)
STAND – Standard CAPE (Unshared)

Figure 18: Approximate Memory Usage Chart

These measuremnts are approximations and were taken from the WPI's clusters monitoring toolkit and while not as strong of a measurement as the others, the results in Figure 18 show one clear advantage to the NASSQ systems, namely, reduced memory usage.

2 Conclusions

Throughout this project we have learned a lot about stream processing engines through our initial rounds of research and work with ESPER to are development work and testing within CAPE. We have produced two variations on the CAPE system, to include a batching and distribution system in which tuples are received, grouped, and sent out through predetermined routing structure (individual or shared), while sharing operators in an attempt to reduce execution costs. The primary new ideas that this project brings to the forefront, is that of the non-duplicative shared routing tree, to reduce tuple copies and memory usage overhead.

In conclusion we think that the concept of the tree execution path can be useful in the low query count system, especially with further testing. However, after our poor results in testing NASSQ on high numbers of queries, we realize that the algorithm for generating the tree and other components of our system need to be improved in several ways, in order to reduce generation cost, tree complexity, and execution cost of the system.

2.1 Future works

Needs of the System:

- A system of non-binary operators, those that can handle similiar test on the same element in order to reduce the tree complexity, and save significant generation and execution time, enabling a non-binary tree structure (i.e. If three operators are looking for greater then 1, 2, or 3, tuples greater than 3 necessarily pass the other

2, and thus a four-way (the fourth being less then 1 and thus a failure) branch could be made, significantly cutting down on the total number of branches that would need to be made).

- A more efficient tree generation algorithm would improve tree generation performance and reduce the associated overhead.
- A better scheduler. Looking at the system load while executing the NASSQ systems in comparison to the standard cape systems, it showed the system is not being utilized to its full potential. Only achieving a 25-30% processor utilization shows that there is potential for a significantly greater number of operations to occur. We believe that the source of this problem may be that the Routing Operator is visited as often as the regular operators, and that a system that spends a longer amount of time in the standard operators, between visiting the router, which takes time to look at things, may increase system utilization.
- When building the tree a hybrid approach, between individual trees for each query and a single routing tree needs to be looked into, utilizing a maximum number of queries per tree. This should provide improvements in generation time and latency, while maintaining many of the other benefits of NASSQ.

When running ad-hoc tests on NASSQ for few queries and large data flow, the results looked promising as memory usage and execution were significantly lower than both NASSQ-Unshared and Standard CAPE, however testing has not been significantly completed on these areas, as our primary focus was high query count.

3 Bibliography

[1] E. Codd. "A relational model of data for large shared data banks." Comm. ACM IS, 6 (June 1970), 377-387.

[2] H. Kostowski. "Stream Database Systems." November 8, 2005. http://www.cs.uml.edu/~hkostows/stream/streams.pdf.

[3]. K. Munagala, U. Srivastava, and J. Widom. "Optimization of Continuous Queries with Shared Expensive Filters." Technical Report, Nov. 2005

[4] R. Ramakrishnan and J. Gehrke. "Database Management Systems." McGraw-Hill Higher Education, 2000.

[5] A. Deshpande, Z. Ives, and V. Raman. "Adaptive query processing." Foundations and Trends in Databases, 1(1), 2007

[6] Crosman, P. "IBM Previews Ultra-Powerful Stream Processing System" Wall Street & Technology. 2007. Wall Street Journal. 22 June 2007, Retrieved 28 April 2008. http:// www.wallstreetandtech.com/blog/archives/2007/06/ibm_previews_ul.html.

[7] R. Avnur and J. M. Hellerstein. "Eddies: continuously adaptive query processing." In SIGMOD,2000.

[8] N. Vijayakumar, Y. Liu, and B. Plale, “Calder query grid service: Insights and experimental evaluations.” in CCGrid Conference, 2006.

[9] E. Rundensteiner et al. "CAPE: Continuous query engine with heterogeneous-grained adaptivity." In VLDB Demo, pages 1353–1356, 2004.

[10] Coral8, http://www.coral8.com, 2004-2008.

[11] Esper, http://esper.codehaus.org, 2008-04-13.

[12] RiverGlass, http://www.riverglassinc.com, 2008-04-08.

[13] A. Arasu et al. "STREAM: The Stanford Stream Data Manager." IEEE Data Engineering Bulletin, 26(1)

[14] R. Nehme, K. Works, E. Rundensteiner, and E. Bertino. "Query Mesh: An Efficient Multi-Route Approach to Query Optimization." CSD TR #08-009, Purdue University, West Lafayette, IN, April 2008.

[15] V. Raghavan, E. Rundensteiner, J. Woycheese, and A. Mukherji. "FireStream: Sensor Stream Processing for Monitoring Fire Spread." 2007 IEEE 23rd International Conference on Data Engineering, April 2007, pp. 1507-1508

[16] V. Raman, A. Deshpande, and J. Hellerstein. "Using State Modules for Adaptive Query Processing," 19th International Conference on Data Engineering (ICDE'03), 2003

4 APPENDICIES

4.1 Appendix A: SPE Feature List

		Institute Research				Open Source		Commercial	
	Feature List	CAPE	Borelias	STREAM		Esper		STREAMBase	Coral 8
Querries	Multiple	yes	yes	yes		yes		yes	yes
	Utilizes Plans		yes	yes				yes	yes
	Dynamic Adjustment		yes	yes					yes
	Plan Manip. (Devolper)								
	Windowing	yes	yes	yes		yes		yes	yes
	Time		?			yes			yes
	Interval		?			yes			yes
	Length		?			yes			yes
	Macro Definitions		yes						
Data Sources	Add/Remove	yes	yes	yes				yes	yes
	Relation Databases	yes	yes	yes				yes	yes
	Complex Datatypes	yes	yes	yes		yes		yes	yes
	Dynamic Event Props.					yes			
	Simulator	yes				yes			
	Data Recorder					?			
	Adapters		yes			yes		yes	yes
Operators	Add New			yes				yes	yes
	Dynamic Adjustment		*	yes					
	Sharing			yes		yes			
	Stream to Stream		yes	no				yes	
	Stream to Relation		yes	yes				yes	
	Relation to Stream		yes	yes				yes	
	Relational to Relational	yes	yes	yes		yes		yes	
Interesting Ops.	Metrenome							yes	
	Split		yes			yes		yes	
	Merge		yes			yes		yes	
Data Handeling	Queing	yes	yes	yes				yes	yes
	Synopsoies			yes					
	Synopsoies Sharing			yes					
	Overflow Handeling			yes					

Interface	Text Authouring		part	yes				part	yes
	Authoring GUI		yes	?				yes	yes
	System Monitor			GUI				GUI	yes
Syntax	Language		Graphical	CQL		?		StreamSQL	CCL
	SQL derivitive		no	yes		yes		yes	yes
Dynamic Ft.	Puncuations		no					no	
	Scheduling		yes	yes				yes	yes
	Advanced Scheduling		yes	yes				yes	maybe?
	Adaptive		yes	yes				yes	maybe?
QoS	Load Shedding		yes						
Analytics	Pattern Recognition					yes			yes
	Listener Tools					yes			
	Life Cycle Control					yes			
	Event Corelation					yes			yes
	Sensor Data Fusion								yes
Advaced	Distributed Support	yes	yes	yes		yes		yes	yes
	Pub-Sub Ability			*					yes
	Support Bounding								
	Fault Tolerence		yes						
	Crash Recovery			*					Enterprise
	Transactions			*					
	Heart Beat Tools							yes	
	MultiThreading					yes		yes	yes
	Persistent State Tools							yes	Enterprise
	"Out of Order" handeling		yes					yes	yes
	Real-Time Output			yes				yes	
	Plugable Architecture					yes			no
	Revision Handeling			yes					
Operating Systems	Windows	yes				yes		yes	yes
	XP	yes				yes		yes	yes
	Vista	yes						yes	yes
	Server	?						yes	yes
	Mac					yes		?	no
	Linux					yes		yes	yes
	Red Hat					yes		yes	yes
	SUSE					yes		yes	yes
	Other					yes		?	yes
	Solaris					?		yes	yes

Languages	Java	yes				yes		yes	yes
	C++	no				no		yes	yes
	C	no				no		no	yes
	.Net	no				yes		yes	yes
		*	mentioned as future work						

4.2 Appendix B: Test Result Data

Standard Run								
FILE	Seed	Throughput	Start	Stop	Time (min)	Rate/min		
280235	2	77970	290722	450000	2.654633333	29371.28794		
280305	1	12959	285404	450032	2.7438	4723.011881		
280402	1	89675	301042	450000	2.482633333	36120.91999		
280412	1	245448	0	450000	7.5	32726.4		
280422	1	268236	0	450000	7.5	35764.8	1629.323326	
280432	1	250157	0	450000	7.5	33354.26667	34491.59666	0.047238269

NASSQNS Rs-200						
FILE	Seed	Throughput	Start	Stop	Time (min)	Rate/min
280350	1	50736	308319	450000	2.36135	21486.01436
280425	1	85345	278941	450000	2.850983333	29935.28549
280505	1	89122	280091	450000	2.831816667	31471.67013

Execution Count	NASSQStreamSelectOperatorImp 1515 ACTIVE	1969

Tuple Through-Put	NASSQStreamSelectOperatorImp 1515 ACTIVE	1969
Tupple Deque	NASSQStreamSelectOperatorImp 1515 ACTIVE	1969
Selectivity	NASSQStreamSelectOperatorImp 1515 ACTIVE	1
Execution_Time	NASSQStreamSelectOperatorImp 1515 ACTIVE	1.09E+07
Execution Count	NASSQStreamSelectOperatorImp 44 ACTIVE	425
Tuple Through-Put	NASSQStreamSelectOperatorImp 44 ACTIVE	0
Tupple Deque	NASSQStreamSelectOperatorImp 44 ACTIVE	425
Selectivity	NASSQStreamSelectOperatorImp 44 ACTIVE	0
Execution_Time	NASSQStreamSelectOperatorImp 44 ACTIVE	2635000
Execution Count	NASSQStreamSelectOperatorImp 2020 ACTIVE	2073
Tuple Through-Put	NASSQStreamSelectOperatorImp 2020 ACTIVE	2073
Tupple Deque	NASSQStreamSelectOperatorImp 2020 ACTIVE	2073
Selectivity	NASSQStreamSelectOperatorImp 2020 ACTIVE	1
Execution_Time	NASSQStreamSelectOperatorImp 2020 ACTIVE	3.38E+07
Execution Count	NASSQStreamSelectOperatorImp 1818 ACTIVE	2050
Tuple Through-Put	NASSQStreamSelectOperatorImp 1818 ACTIVE	2050
Tupple Deque	NASSQStreamSelectOperatorImp 1818 ACTIVE	2050
Selectivity	NASSQStreamSelectOperatorImp 1818 ACTIVE	1
Execution_Time	NASSQStreamSelectOperatorImp 1818 ACTIVE	1.56E+07
Execution Count	NASSQStreamSelectOperatorImp 33 ACTIVE	0
Tuple Through-Put	NASSQStreamSelectOperatorImp 33 ACTIVE	0
Tupple Deque	NASSQStreamSelectOperatorImp 33 ACTIVE	0
Selectivity	NASSQStreamSelectOperatorImp 33 ACTIVE	0
Execution_Time	NASSQStreamSelectOperatorImp 33 ACTIVE	0
Execution Count	NASSQStreamSelectOperatorImp 1313 ACTIVE	2009
Tuple Through-Put	NASSQStreamSelectOperatorImp 1313 ACTIVE	2009
Tupple Deque	NASSQStreamSelectOperatorImp 1313 ACTIVE	2009
Selectivity	NASSQStreamSelectOperatorImp 1313 ACTIVE	1
Execution_Time	NASSQStreamSelectOperatorImp 1313 ACTIVE	1.34E+07
Execution	NASSQStreamSelectOperatorImp 2727 AC-	2050

Count	TIVE	
Tuple Through-Put	NASSQStreamSelectOperatorImp 2727 AC-TIVE	1172
Tupple Deque	NASSQStreamSelectOperatorImp 2727 AC-TIVE	2050
Selectivity	NASSQStreamSelectOperatorImp 2727 AC-TIVE	0.571707317
Execution_Time	NASSQStreamSelectOperatorImp 2727 AC-TIVE	5.12E+07
Execution Count	NASSQStreamSelectOperatorImp 3030 AC-TIVE	0
Tuple Through-Put	NASSQStreamSelectOperatorImp 3030 AC-TIVE	0
Tupple Deque	NASSQStreamSelectOperatorImp 3030 AC-TIVE	0
Selectivity	NASSQStreamSelectOperatorImp 3030 AC-TIVE	0
Execution_Time	NASSQStreamSelectOperatorImp 3030 AC-TIVE	0
Throughput	OutputAdapterOperatorImp 3232 ACTIVE	10115
Latency	OutputAdapterOperatorImp 3232 ACTIVE	1839020
Execution Count	NASSQStreamSelectOperatorImp 11 ACTIVE	2000
Tuple Through-Put	NASSQStreamSelectOperatorImp 11 ACTIVE	0
Tupple Deque	NASSQStreamSelectOperatorImp 11 ACTIVE	2000
Selectivity	NASSQStreamSelectOperatorImp 11 ACTIVE	0
Execution_Time	NASSQStreamSelectOperatorImp 11 ACTIVE	1.11E+07
Execution Count	NASSQStreamSelectOperatorImp 2929 AC-TIVE	0
Tuple Through-Put	NASSQStreamSelectOperatorImp 2929 AC-TIVE	0
Tupple Deque	NASSQStreamSelectOperatorImp 2929 AC-TIVE	0
Selectivity	NASSQStreamSelectOperatorImp 2929 AC-TIVE	0
Execution_Time	NASSQStreamSelectOperatorImp 2929 AC-TIVE	0
Execution Count	NASSQStreamSelectOperatorImp 66 ACTIVE	878
Tuple Through-Put	NASSQStreamSelectOperatorImp 66 ACTIVE	435
Tupple Deque	NASSQStreamSelectOperatorImp 66 ACTIVE	878
Selectivity	NASSQStreamSelectOperatorImp 66 ACTIVE	0.495444191
Execution_Time	NASSQStreamSelectOperatorImp 66 ACTIVE	5846000
Execution Count	NASSQStreamSelectOperatorImp 1010 AC-TIVE	2050
Tuple Through-Put	NASSQStreamSelectOperatorImp 1010 AC-TIVE	2050
Tupple Deque	NASSQStreamSelectOperatorImp 1010 AC-TIVE	2050
Selectivity	NASSQStreamSelectOperatorImp 1010 AC-	1

	TIVE	
Execution_Time	NASSQStreamSelectOperatorImp 1010 ACTIVE	1.48E+07
Execution Count	NASSQStreamSelectOperatorImp 1212 ACTIVE	2050
Tuple Through-Put	NASSQStreamSelectOperatorImp 1212 ACTIVE	0
Tupple Deque	NASSQStreamSelectOperatorImp 1212 ACTIVE	2050
Selectivity	NASSQStreamSelectOperatorImp 1212 ACTIVE	0
Execution_Time	NASSQStreamSelectOperatorImp 1212 ACTIVE	1.67E+07
Execution Count	NASSQStreamSelectOperatorImp 2828 ACTIVE	858
Tuple Through-Put	NASSQStreamSelectOperatorImp 2828 ACTIVE	433
Tupple Deque	NASSQStreamSelectOperatorImp 2828 ACTIVE	858
Selectivity	NASSQStreamSelectOperatorImp 2828 ACTIVE	0.504662005
Execution_Time	NASSQStreamSelectOperatorImp 2828 ACTIVE	7981000
Execution_Time	NASSQRouterOperatorImp 00 ACTIVE	3.17E+11
Execution Count	NASSQStreamSelectOperatorImp 22 ACTIVE	858
Tuple Through-Put	NASSQStreamSelectOperatorImp 22 ACTIVE	0
Tupple Deque	NASSQStreamSelectOperatorImp 22 ACTIVE	858
Selectivity	NASSQStreamSelectOperatorImp 22 ACTIVE	0
Execution_Time	NASSQStreamSelectOperatorImp 22 ACTIVE	4945000
Execution Count	NASSQStreamSelectOperatorImp 1616 ACTIVE	1999
Tuple Through-Put	NASSQStreamSelectOperatorImp 1616 ACTIVE	1999
Tupple Deque	NASSQStreamSelectOperatorImp 1616 ACTIVE	1999
Selectivity	NASSQStreamSelectOperatorImp 1616 ACTIVE	1
Execution_Time	NASSQStreamSelectOperatorImp 1616 ACTIVE	1.04E+07
Execution Count	NASSQStreamSelectOperatorImp 2626 ACTIVE	1928
Tuple Through-Put	NASSQStreamSelectOperatorImp 2626 ACTIVE	559
Tupple Deque	NASSQStreamSelectOperatorImp 2626 ACTIVE	1928
Selectivity	NASSQStreamSelectOperatorImp 2626 ACTIVE	0.289937759
Execution_Time	NASSQStreamSelectOperatorImp 2626 ACTIVE	9815000
Execution Count	NASSQStreamSelectOperatorImp 77 ACTIVE	2100

Tuple Through-Put	NASSQStreamSelectOperatorImp 77 ACTIVE	901
Tupple Deque	NASSQStreamSelectOperatorImp 77 ACTIVE	2100
Selectivity	NASSQStreamSelectOperatorImp 77 ACTIVE	0.429047619
Execution_Time	NASSQStreamSelectOperatorImp 77 ACTIVE	1.73E+07
Execution Count	NASSQStreamSelectOperatorImp 1111 AC-TIVE	568
Tuple Through-Put	NASSQStreamSelectOperatorImp 1111 AC-TIVE	568
Tupple Deque	NASSQStreamSelectOperatorImp 1111 AC-TIVE	568
Selectivity	NASSQStreamSelectOperatorImp 1111 AC-TIVE	1
Execution_Time	NASSQStreamSelectOperatorImp 1111 AC-TIVE	3082000
Execution Count	NASSQStreamSelectOperatorImp 1919 AC-TIVE	1142
Tuple Through-Put	NASSQStreamSelectOperatorImp 1919 AC-TIVE	1142
Tupple Deque	NASSQStreamSelectOperatorImp 1919 AC-TIVE	1142
Selectivity	NASSQStreamSelectOperatorImp 1919 AC-TIVE	1
Execution_Time	NASSQStreamSelectOperatorImp 1919 AC-TIVE	5507000
Execution Count	NASSQStreamSelectOperatorImp 2323 AC-TIVE	1999
Tuple Through-Put	NASSQStreamSelectOperatorImp 2323 AC-TIVE	1410
Tupple Deque	NASSQStreamSelectOperatorImp 2323 AC-TIVE	1999
Selectivity	NASSQStreamSelectOperatorImp 2323 AC-TIVE	0.705352676
Execution_Time	NASSQStreamSelectOperatorImp 2323 AC-TIVE	5.00E+07
Execution Count	NASSQStreamSelectOperatorImp 1414 AC-TIVE	0
Tuple Through-Put	NASSQStreamSelectOperatorImp 1414 AC-TIVE	0
Tupple Deque	NASSQStreamSelectOperatorImp 1414 AC-TIVE	0
Selectivity	NASSQStreamSelectOperatorImp 1414 AC-TIVE	0
Execution_Time	NASSQStreamSelectOperatorImp 1414 AC-TIVE	0
Execution Count	NASSQStreamSelectOperatorImp 2222 AC-TIVE	0
Tuple Through-Put	NASSQStreamSelectOperatorImp 2222 AC-TIVE	0
Tupple Deque	NASSQStreamSelectOperatorImp 2222 AC-TIVE	0
Selectivity	NASSQStreamSelectOperatorImp 2222 AC-TIVE	0

Execution_Time	NASSQStreamSelectOperatorImp 2222 ACTIVE	0
Execution Count	NASSQStreamSelectOperatorImp 3131 ACTIVE	0
Tuple Through-Put	NASSQStreamSelectOperatorImp 3131 ACTIVE	0
Tupple Deque	NASSQStreamSelectOperatorImp 3131 ACTIVE	0
Selectivity	NASSQStreamSelectOperatorImp 3131 ACTIVE	0
Execution_Time	NASSQStreamSelectOperatorImp 3131 ACTIVE	0
Execution Count	NASSQStreamSelectOperatorImp 2424 ACTIVE	2030
Tuple Through-Put	NASSQStreamSelectOperatorImp 2424 ACTIVE	858
Tupple Deque	NASSQStreamSelectOperatorImp 2424 ACTIVE	2030
Selectivity	NASSQStreamSelectOperatorImp 2424 ACTIVE	0.422660099
Execution_Time	NASSQStreamSelectOperatorImp 2424 ACTIVE	1.23E+07
Execution Count	NASSQStreamSelectOperatorImp 99 ACTIVE	0
Tuple Through-Put	NASSQStreamSelectOperatorImp 99 ACTIVE	0
Tupple Deque	NASSQStreamSelectOperatorImp 99 ACTIVE	0
Selectivity	NASSQStreamSelectOperatorImp 99 ACTIVE	0
Execution_Time	NASSQStreamSelectOperatorImp 99 ACTIVE	0
Execution Count	NASSQStreamSelectOperatorImp 88 ACTIVE	2077
Tuple Through-Put	NASSQStreamSelectOperatorImp 88 ACTIVE	2077
Tupple Deque	NASSQStreamSelectOperatorImp 88 ACTIVE	2077
Selectivity	NASSQStreamSelectOperatorImp 88 ACTIVE	1
Execution_Time	NASSQStreamSelectOperatorImp 88 ACTIVE	1.59E+07
Execution Count	NASSQStreamSelectOperatorImp 2525 ACTIVE	2015
Tuple Through-Put	NASSQStreamSelectOperatorImp 2525 ACTIVE	425
Tupple Deque	NASSQStreamSelectOperatorImp 2525 ACTIVE	2015
Selectivity	NASSQStreamSelectOperatorImp 2525 ACTIVE	0.210918114
Execution_Time	NASSQStreamSelectOperatorImp 2525 ACTIVE	1.15E+07
Execution Count	NASSQStreamSelectOperatorImp 55 ACTIVE	847
Tuple Through-Put	NASSQStreamSelectOperatorImp 55 ACTIVE	0
Tupple Deque	NASSQStreamSelectOperatorImp 55 ACTIVE	847
Selectivity	NASSQStreamSelectOperatorImp 55 ACTIVE	0

Execution_Time	NASSQStreamSelectOperatorImp 55 ACTIVE	4429000
Execution Count	NASSQStreamSelectOperatorImp 2121 ACTIVE	1539
Tuple Through-Put	NASSQStreamSelectOperatorImp 2121 ACTIVE	0
Tupple Deque	NASSQStreamSelectOperatorImp 2121 ACTIVE	1539
Selectivity	NASSQStreamSelectOperatorImp 2121 ACTIVE	0
Execution_Time	NASSQStreamSelectOperatorImp 2121 ACTIVE	8166000
Execution Count	NASSQStreamSelectOperatorImp 1717 ACTIVE	415
Tuple Through-Put	NASSQStreamSelectOperatorImp 1717 ACTIVE	415
Tupple Deque	NASSQStreamSelectOperatorImp 1717 ACTIVE	415
Selectivity	NASSQStreamSelectOperatorImp 1717 ACTIVE	1
Execution_Time	NASSQStreamSelectOperatorImp 1717 ACTIVE	2376000

4.3 Appendix C: Generated NASSQ Batch

```
<queryplan>
	<operator root="true" id="32"

	className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.OutputAdapterOperatorImp"
		numberOfOutputQueue="1">
		<classVariables>
			<variable name="IsEddyOp" value="false" />
		</classVariables>
		<properties></properties>
		<schema />
		<parents></parents>
		<children>
<child type="operator" id = "1"/>
<child type="operator" id = "2"/>
<child type="operator" id = "3"/>
<child type="operator" id = "4"/>
<child type="operator" id = "5"/>
<child type="operator" id = "6"/>
<child type="operator" id = "7"/>
<child type="operator" id = "8"/>
<child type="operator" id = "9"/>
<child type="operator" id = "10"/>
<child type="operator" id = "11"/>
<child type="operator" id = "12"/>
<child type="operator" id = "13"/>
<child type="operator" id = "14"/>
<child type="operator" id = "15"/>
<child type="operator" id = "16"/>
```

```
<child type="operator" id = "17"/>
<child type="operator" id = "18"/>
<child type="operator" id = "19"/>
<child type="operator" id = "20"/>
<child type="operator" id = "21"/>
<child type="operator" id = "22"/>
<child type="operator" id = "23"/>
<child type="operator" id = "24"/>
<child type="operator" id = "25"/>
<child type="operator" id = "26"/>
<child type="operator" id = "27"/>
<child type="operator" id = "28"/>
<child type="operator" id = "29"/>
<child type="operator" id = "30"/>
<child type="operator" id = "31"/>
                </children>
        </operator>

        <operator root="false" id="31"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="31" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="3" />
                        <expressions>
                                <expr id="1" type="TerminalExpressionStrImp"
                                        qPosition="0" cPosition="11" value="" valtype="column"
lid=""
                                        rid="" />
                                <expr id="2" type="TerminalConstantImp" qPosition=""
                                        cPosition="" value="41.5" valtype="double" lid="" rid="" />
                                <expr id="3" type="BinCOMPExpressionStrImp" qPosition=""
                                        cPosition="" value="" valtype="GT" lid="1" rid="2" />
                        </expressions>
                </classVariables>
                <properties />
                <schema />
                <parents>
                        <parent id="32" />
                </parents>
                <children>
                        <child type="operator" id="0" queueId="30" />
                </children>
        </operator>
        <operator root="false" id="30"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="30" />
```

```
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="6" />
                        <expressions>
                                <expr id="4" type="TerminalExpressionStrImp"
                                        qPosition="0" cPosition="8" value="" valtype="column"
lid=""
                                        rid="" />
                                <expr id="5" type="TerminalConstantImp" qPosition=""
                                        cPosition="" value="h-e" valtype="String" lid="" rid="" />
                                <expr id="6" type="BinCOMPExpressionStrImp" qPosition=""
                                        cPosition="" value="" valtype="EQ" lid="4" rid="5" />
                        </expressions>
                </classVariables>
                <properties />
                <schema />
                <parents>
                        <parent id="32" />
                </parents>
                <children>
                        <child type="operator" id="0" queueId="29" />
                </children>
        </operator>
        <operator root="false" id="29"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="29" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="9" />
                        <expressions>
                                <expr id="7" type="TerminalExpressionStrImp"
                                        qPosition="0" cPosition="7" value="" valtype="column"
lid=""
                                        rid="" />
                                <expr id="8" type="TerminalConstantImp" qPosition=""
                                        cPosition="" value="a-f-g" valtype="String" lid="" rid="" />
                                <expr id="9" type="BinCOMPExpressionStrImp" qPosition=""
                                        cPosition="" value="" valtype="EQ" lid="7" rid="8" />
                        </expressions>
                </classVariables>
                <properties />
                <schema />
                <parents>
                        <parent id="32" />
                </parents>
                <children>
                        <child type="operator" id="0" queueId="28" />
                </children>
        </operator>
        <operator root="false" id="28"
```

```
className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="28" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="12" />
                        <expressions>
                                <expr id="10" type="TerminalExpressionStrImp"
                                        qPosition="0" cPosition="1" value="" valtype="column"
lid=""
                                        rid="" />
                                <expr id="11" type="TerminalConstantImp" qPosition=""
                                        cPosition="" value="300" valtype="integer" lid="" rid="" />
                                <expr id="12" type="BinCOMPExpressionStrImp"
                                        qPosition="" cPosition="" value="" valtype="LT" lid="10"
rid="11" />
                        </expressions>
                </classVariables>
                <properties />
                <schema />
                <parents>
                        <parent id="32" />
                </parents>
                <children>
                        <child type="operator" id="0" queueId="27" />
                </children>
        </operator>
        <operator root="false" id="27"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="27" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="15" />
                        <expressions>
                                <expr id="13" type="TerminalExpressionStrImp"
                                        qPosition="0" cPosition="1" value="" valtype="column"
lid=""
                                        rid="" />
                                <expr id="14" type="TerminalConstantImp" qPosition=""
                                        cPosition="" value="200" valtype="integer" lid="" rid="" />
                                <expr id="15" type="BinCOMPExpressionStrImp"
                                        qPosition="" cPosition="" value="" valtype="LT" lid="13"
rid="14" />
                        </expressions>
                </classVariables>
                <properties />
                <schema />
                <parents>
```

```
                <parent id="32" />
            </parents>
            <children>
                <child type="operator" id="0" queueId="26" />
            </children>
        </operator>
        <operator root="false" id="26"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
            <classVariables>
                <variable name="QMeshOperatorID" value="26" />
                <variable name="IsEddyOp" value="false" />
                <variable name="StreamID" value="0" />
                <variable name="expression_id" value="18" />
                <expressions>
                    <expr id="16" type="TerminalExpressionStrImp"
                        qPosition="0" cPosition="1" value="" valtype="column"
lid=""
                        rid="" />
                    <expr id="17" type="TerminalConstantImp" qPosition=""
                        cPosition="" value="115" valtype="integer" lid="" rid="" />
                    <expr id="18" type="BinCOMPExpressionStrImp"
                        qPosition="" cPosition="" value="" valtype="LT" lid="16"
rid="17" />
                </expressions>
            </classVariables>
            <properties />
            <schema />
            <parents>
                <parent id="32" />
            </parents>
            <children>
                <child type="operator" id="0" queueId="25" />
            </children>
        </operator>
        <operator root="false" id="25"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
            <classVariables>
                <variable name="QMeshOperatorID" value="25" />
                <variable name="IsEddyOp" value="false" />
                <variable name="StreamID" value="0" />
                <variable name="expression_id" value="21" />
                <expressions>
                    <expr id="19" type="TerminalExpressionStrImp"
                        qPosition="0" cPosition="1" value="" valtype="column"
lid=""
                        rid="" />
                    <expr id="20" type="TerminalConstantImp" qPosition=""
                        cPosition="" value="300" valtype="integer" lid="" rid="" />
                    <expr id="21" type="BinCOMPExpressionStrImp"
```

```
                                        qPosition="" cPosition="" value="" valtype="GT" lid="19"
rid="20" />
                        </expressions>
                </classVariables>
                <properties />
                <schema />
                <parents>
                        <parent id="32" />
                </parents>
                <children>
                        <child type="operator" id="0" queueId="24" />
                </children>
        </operator>
        <operator root="false" id="24"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="24" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="24" />
                        <expressions>
                                <expr id="22" type="TerminalExpressionStrImp"
                                        qPosition="0" cPosition="1" value="" valtype="column"
lid=""
                                        rid="" />
                                <expr id="23" type="TerminalConstantImp" qPosition=""
                                        cPosition="" value="200" valtype="integer" lid="" rid="" />
                                <expr id="24" type="BinCOMPExpressionStrImp"
                                        qPosition="" cPosition="" value="" valtype="GT" lid="22"
rid="23" />
                        </expressions>
                </classVariables>
                <properties />
                <schema />
                <parents>
                        <parent id="32" />
                </parents>
                <children>
                        <child type="operator" id="0" queueId="23" />
                </children>
        </operator>
        <operator root="false" id="23"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="23" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="27" />
                        <expressions>
```

```
                                        <expr id="25" type="TerminalExpressionStrImp"
                                                qPosition="0" cPosition="1" value="" valtype="column"
lid=""
                                                rid="" />
                                        <expr id="26" type="TerminalConstantImp" qPosition=""
                                                cPosition="" value="114" valtype="integer" lid="" rid="" />
                                        <expr id="27" type="BinCOMPExpressionStrImp"
                                                qPosition="" cPosition="" value="" valtype="GT" lid="25"
rid="26" />
                                </expressions>
                        </classVariables>
                        <properties />
                        <schema />
                        <parents>
                                <parent id="32" />
                        </parents>
                        <children>
                                <child type="operator" id="0" queueId="22" />
                        </children>
                </operator>
                <operator root="false" id="22"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                        <classVariables>
                                <variable name="QMeshOperatorID" value="22" />
                                <variable name="IsEddyOp" value="false" />
                                <variable name="StreamID" value="0" />
                                <variable name="expression_id" value="30" />
                                <expressions>
                                        <expr id="28" type="TerminalExpressionStrImp"
                                                qPosition="0" cPosition="12" value="" valtype="column"
lid=""
                                                rid="" />
                                        <expr id="29" type="TerminalConstantImp" qPosition=""
                                                cPosition="" value="300" valtype="double" lid="" rid="" />
                                        <expr id="30" type="BinCOMPExpressionStrImp"
                                                qPosition="" cPosition="" value="" valtype="GT" lid="28"
rid="29" />
                                </expressions>
                        </classVariables>
                        <properties />
                        <schema />
                        <parents>
                                <parent id="32" />
                        </parents>
                        <children>
                                <child type="operator" id="0" queueId="21" />
                        </children>
                </operator>
                <operator root="false" id="21"
```

```
className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="21" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="33" />
                        <expressions>
                                <expr id="31" type="TerminalExpressionStrImp"
                                        qPosition="0" cPosition="12" value="" valtype="column"
lid=""
                                        rid="" />
                                <expr id="32" type="TerminalConstantImp" qPosition=""
                                        cPosition="" value="200" valtype="double" lid="" rid="" />
                                <expr id="33" type="BinCOMPExpressionStrImp"
                                        qPosition="" cPosition="" value="" valtype="GT" lid="31"
rid="32" />
                        </expressions>
                </classVariables>
                <properties />
                <schema />
                <parents>
                        <parent id="32" />
                </parents>
                <children>
                        <child type="operator" id="0" queueId="20" />
                </children>
        </operator>
        <operator root="false" id="20"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="20" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="36" />
                        <expressions>
                                <expr id="34" type="TerminalExpressionStrImp"
                                        qPosition="0" cPosition="12" value="" valtype="column"
lid=""
                                        rid="" />
                                <expr id="35" type="TerminalConstantImp" qPosition=""
                                        cPosition="" value="100" valtype="double" lid="" rid="" />
                                <expr id="36" type="BinCOMPExpressionStrImp"
                                        qPosition="" cPosition="" value="" valtype="GT" lid="34"
rid="35" />
                        </expressions>
                </classVariables>
                <properties />
                <schema />
                <parents>
                        <parent id="32" />
```

```
                </parents>
                <children>
                        <child type="operator" id="0" queueId="19" />
                </children>
        </operator>
        <operator root="false" id="19"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="19" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="39" />
                        <expressions>
                                <expr id="37" type="TerminalExpressionStrImp"
                                        qPosition="0" cPosition="12" value="" valtype="column"
lid=""
                                        rid="" />
                                <expr id="38" type="TerminalConstantImp" qPosition=""
                                        cPosition="" value="0" valtype="double" lid="" rid="" />
                                <expr id="39" type="BinCOMPExpressionStrImp"
                                        qPosition="" cPosition="" value="" valtype="GT" lid="37"
rid="38" />
                        </expressions>
                </classVariables>
                <properties />
                <schema />
                <parents>
                        <parent id="32" />
                </parents>
                <children>
                        <child type="operator" id="0" queueId="18" />
                </children>
        </operator>
        <operator root="false" id="18"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="18" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="42" />
                        <expressions>
                                <expr id="40" type="TerminalExpressionStrImp"
                                        qPosition="0" cPosition="11" value="" valtype="column"
lid=""
                                        rid="" />
                                <expr id="41" type="TerminalConstantImp" qPosition=""
                                        cPosition="" value="43.0" valtype="double" lid="" rid="" />
                                <expr id="42" type="BinCOMPExpressionStrImp"
```

```
                                        qPosition="" cPosition="" value="" valtype="LT" lid="40"
rid="41" />
                        </expressions>
                </classVariables>
                <properties />
                <schema />
                <parents>
                        <parent id="32" />
                </parents>
                <children>
                        <child type="operator" id="0" queueId="17" />
                </children>
        </operator>
        <operator root="false" id="17"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="17" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="45" />
                        <expressions>
                                <expr id="43" type="TerminalExpressionStrImp"
                                        qPosition="0" cPosition="10" value="" valtype="column"
lid=""
                                        rid="" />
                                <expr id="44" type="TerminalConstantImp" qPosition=""
                                        cPosition="" value="33.3" valtype="double" lid="" rid="" />
                                <expr id="45" type="BinCOMPExpressionStrImp"
                                        qPosition="" cPosition="" value="" valtype="GT" lid="43"
rid="44" />
                        </expressions>
                </classVariables>
                <properties />
                <schema />
                <parents>
                        <parent id="32" />
                </parents>
                <children>
                        <child type="operator" id="0" queueId="16" />
                </children>
        </operator>
        <operator root="false" id="16"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="16" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="48" />
                        <expressions>
```

```
                                        <expr id="46" type="TerminalExpressionStrImp"
                                                qPosition="0" cPosition="10" value="" valtype="column"
lid=""
                                                rid="" />
                                        <expr id="47" type="TerminalConstantImp" qPosition=""
                                                cPosition="" value="33.2" valtype="double" lid="" rid="" />
                                        <expr id="48" type="BinCOMPExpressionStrImp"
                                                qPosition="" cPosition="" value="" valtype="GT" lid="46"
rid="47" />
                                </expressions>
                        </classVariables>
                        <properties />
                        <schema />
                        <parents>
                                <parent id="32" />
                        </parents>
                        <children>
                                <child type="operator" id="0" queueId="15" />
                        </children>
                </operator>
                <operator root="false" id="15"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                        <classVariables>
                                <variable name="QMeshOperatorID" value="15" />
                                <variable name="IsEddyOp" value="false" />
                                <variable name="StreamID" value="0" />
                                <variable name="expression_id" value="51" />
                                <expressions>
                                        <expr id="49" type="TerminalExpressionStrImp"
                                                qPosition="0" cPosition="10" value="" valtype="column"
lid=""
                                                rid="" />
                                        <expr id="50" type="TerminalConstantImp" qPosition=""
                                                cPosition="" value="33.1" valtype="double" lid="" rid="" />
                                        <expr id="51" type="BinCOMPExpressionStrImp"
                                                qPosition="" cPosition="" value="" valtype="GT" lid="49"
rid="50" />
                                </expressions>
                        </classVariables>
                        <properties />
                        <schema />
                        <parents>
                                <parent id="32" />
                        </parents>
                        <children>
                                <child type="operator" id="0" queueId="14" />
                        </children>
                </operator>
                <operator root="false" id="14"
```

```
className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="14" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="54" />
                        <expressions>
                                <expr id="52" type="TerminalExpressionStrImp"
                                        qPosition="0" cPosition="10" value="" valtype="column"
lid=""
                                        rid="" />
                                <expr id="53" type="TerminalConstantImp" qPosition=""
                                        cPosition="" value="33.0" valtype="double" lid="" rid="" />
                                <expr id="54" type="BinCOMPExpressionStrImp"
                                        qPosition="" cPosition="" value="" valtype="GT" lid="52"
rid="53" />
                        </expressions>
                </classVariables>
                <properties />
                <schema />
                <parents>
                        <parent id="32" />
                </parents>
                <children>
                        <child type="operator" id="0" queueId="13" />
                </children>
        </operator>
        <operator root="false" id="13"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="13" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="57" />
                        <expressions>
                                <expr id="55" type="TerminalExpressionStrImp"
                                        qPosition="0" cPosition="10" value="" valtype="column"
lid=""
                                        rid="" />
                                <expr id="56" type="TerminalConstantImp" qPosition=""
                                        cPosition="" value="33.7" valtype="double" lid="" rid="" />
                                <expr id="57" type="BinCOMPExpressionStrImp"
                                        qPosition="" cPosition="" value="" valtype="LT" lid="55"
rid="56" />
                        </expressions>
                </classVariables>
                <properties />
                <schema />
                <parents>
                        <parent id="32" />
```

```
                </parents>
                <children>
                        <child type="operator" id="0" queueId="12" />
                </children>
        </operator>
        <operator root="false" id="12"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="12" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="60" />
                        <expressions>
                                <expr id="58" type="TerminalExpressionStrImp"
                                        qPosition="0" cPosition="10" value="" valtype="column"
lid=""
                                        rid="" />
                                <expr id="59" type="TerminalConstantImp" qPosition=""
                                        cPosition="" value="33.3" valtype="double" lid="" rid="" />
                                <expr id="60" type="BinCOMPExpressionStrImp"
                                        qPosition="" cPosition="" value="" valtype="LT" lid="58"
rid="59" />
                        </expressions>
                </classVariables>
                <properties />
                <schema />
                <parents>
                        <parent id="32" />
                </parents>
                <children>
                        <child type="operator" id="0" queueId="11" />
                </children>
        </operator>
        <operator root="false" id="11"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="11" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="63" />
                        <expressions>
                                <expr id="61" type="TerminalExpressionStrImp"
                                        qPosition="0" cPosition="11" value="" valtype="column"
lid=""
                                        rid="" />
                                <expr id="62" type="TerminalConstantImp" qPosition=""
                                        cPosition="" value="40.8" valtype="double" lid="" rid="" />
                                <expr id="63" type="BinCOMPExpressionStrImp"
```

```
                                        qPosition="" cPosition="" value="" valtype="GT" lid="61"
rid="62" />
                        </expressions>
                </classVariables>
                <properties />
                <schema />
                <parents>
                        <parent id="32" />
                </parents>
                <children>
                        <child type="operator" id="0" queueId="10" />
                </children>
        </operator>
        <operator root="false" id="10"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="10" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="66" />
                        <expressions>
                                <expr id="64" type="TerminalExpressionStrImp"
                                        qPosition="0" cPosition="11" value="" valtype="column"
lid=""
                                        rid="" />
                                <expr id="65" type="TerminalConstantImp" qPosition=""
                                        cPosition="" value="40.5" valtype="double" lid="" rid="" />
                                <expr id="66" type="BinCOMPExpressionStrImp"
                                        qPosition="" cPosition="" value="" valtype="GT" lid="64"
rid="65" />
                        </expressions>
                </classVariables>
                <properties />
                <schema />
                <parents>
                        <parent id="32" />
                </parents>
                <children>
                        <child type="operator" id="0" queueId="9" />
                </children>
        </operator>
        <operator root="false" id="9"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="9" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="69" />
                        <expressions>
```

```
                                        <expr id="67" type="TerminalExpressionStrImp"
                                                qPosition="0" cPosition="11" value="" valtype="column"
lid=""
                                                rid="" />
                                        <expr id="68" type="TerminalConstantImp" qPosition=""
                                                cPosition="" value="40.3" valtype="double" lid="" rid="" />
                                        <expr id="69" type="BinCOMPExpressionStrImp"
                                                qPosition="" cPosition="" value="" valtype="GT" lid="67"
rid="68" />
                                </expressions>
                        </classVariables>
                        <properties />
                        <schema />
                        <parents>
                                <parent id="32" />
                        </parents>
                        <children>
                                <child type="operator" id="0" queueId="8" />
                        </children>
                </operator>
                <operator root="false" id="8"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                        <classVariables>
                                <variable name="QMeshOperatorID" value="8" />
                                <variable name="IsEddyOp" value="false" />
                                <variable name="StreamID" value="0" />
                                <variable name="expression_id" value="72" />
                                <expressions>
                                        <expr id="70" type="TerminalExpressionStrImp"
                                                qPosition="0" cPosition="11" value="" valtype="column"
lid=""
                                                rid="" />
                                        <expr id="71" type="TerminalConstantImp" qPosition=""
                                                cPosition="" value="40.0" valtype="double" lid="" rid="" />
                                        <expr id="72" type="BinCOMPExpressionStrImp"
                                                qPosition="" cPosition="" value="" valtype="GT" lid="70"
rid="71" />
                                </expressions>
                        </classVariables>
                        <properties />
                        <schema />
                        <parents>
                                <parent id="32" />
                        </parents>
                        <children>
                                <child type="operator" id="0" queueId="7" />
                        </children>
                </operator>
                <operator root="false" id="7"
```

```
className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="7" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="75" />
                        <expressions>
                                <expr id="73" type="TerminalExpressionStrImp"
                                        qPosition="0" cPosition="11" value="" valtype="column"
lid=""
                                        rid="" />
                                <expr id="74" type="TerminalConstantImp" qPosition=""
                                        cPosition="" value="42.8" valtype="double" lid="" rid="" />
                                <expr id="75" type="BinCOMPExpressionStrImp"
                                        qPosition="" cPosition="" value="" valtype="LT" lid="73"
rid="74" />
                        </expressions>
                </classVariables>
                <properties />
                <schema />
                <parents>
                        <parent id="32" />
                </parents>
                <children>
                        <child type="operator" id="0" queueId="6" />
                </children>
        </operator>
        <operator root="false" id="6"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="6" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="78" />
                        <expressions>
                                <expr id="76" type="TerminalExpressionStrImp"
                                        qPosition="0" cPosition="11" value="" valtype="column"
lid=""
                                        rid="" />
                                <expr id="77" type="TerminalConstantImp" qPosition=""
                                        cPosition="" value="42.7" valtype="double" lid="" rid="" />
                                <expr id="78" type="BinCOMPExpressionStrImp"
                                        qPosition="" cPosition="" value="" valtype="LT" lid="76"
rid="77" />
                        </expressions>
                </classVariables>
                <properties />
                <schema />
                <parents>
                        <parent id="32" />
```

```
                </parents>
                <children>
                        <child type="operator" id="0" queueId="5" />
                </children>
        </operator>
        <operator root="false" id="5"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="5" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="81" />
                        <expressions>
                                <expr id="79" type="TerminalExpressionStrImp"
                                        qPosition="0" cPosition="11" value="" valtype="column"
lid=""
                                        rid="" />
                                <expr id="80" type="TerminalConstantImp" qPosition=""
                                        cPosition="" value="42.2" valtype="double" lid="" rid="" />
                                <expr id="81" type="BinCOMPExpressionStrImp"
                                        qPosition="" cPosition="" value="" valtype="LT" lid="79"
rid="80" />
                        </expressions>
                </classVariables>
                <properties />
                <schema />
                <parents>
                        <parent id="32" />
                </parents>
                <children>
                        <child type="operator" id="0" queueId="4" />
                </children>
        </operator>
        <operator root="false" id="4"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="4" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="84" />
                        <expressions>
                                <expr id="82" type="TerminalExpressionStrImp"
                                        qPosition="0" cPosition="11" value="" valtype="column"
lid=""
                                        rid="" />
                                <expr id="83" type="TerminalConstantImp" qPosition=""
                                        cPosition="" value="42.1" valtype="double" lid="" rid="" />
                                <expr id="84" type="BinCOMPExpressionStrImp"
```

```
                                        qPosition="" cPosition="" value="" valtype="LT" lid="82"
rid="83" />
                        </expressions>
                </classVariables>
                <properties />
                <schema />
                <parents>
                        <parent id="32" />
                </parents>
                <children>
                        <child type="operator" id="0" queueId="3" />
                </children>
        </operator>
        <operator root="false" id="3"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="3" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="87" />
                        <expressions>
                                <expr id="85" type="TerminalExpressionStrImp"
                                        qPosition="0" cPosition="11" value="" valtype="column"
lid=""
                                        rid="" />
                                <expr id="86" type="TerminalConstantImp" qPosition=""
                                        cPosition="" value="40.8" valtype="double" lid="" rid="" />
                                <expr id="87" type="BinCOMPExpressionStrImp"
                                        qPosition="" cPosition="" value="" valtype="LT" lid="85"
rid="86" />
                        </expressions>
                </classVariables>
                <properties />
                <schema />
                <parents>
                        <parent id="32" />
                </parents>
                <children>
                        <child type="operator" id="0" queueId="2" />
                </children>
        </operator>
        <operator root="false" id="2"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                <classVariables>
                        <variable name="QMeshOperatorID" value="2" />
                        <variable name="IsEddyOp" value="false" />
                        <variable name="StreamID" value="0" />
                        <variable name="expression_id" value="90" />
                        <expressions>
```

```
                                        <expr id="88" type="TerminalExpressionStrImp"
                                                qPosition="0" cPosition="11" value="" valtype="column"
lid=""
                                                rid="" />
                                        <expr id="89" type="TerminalConstantImp" qPosition=""
                                                cPosition="" value="40.7" valtype="double" lid="" rid="" />
                                        <expr id="90" type="BinCOMPExpressionStrImp"
                                                qPosition="" cPosition="" value="" valtype="LT" lid="88"
rid="89" />
                                </expressions>
                        </classVariables>
                        <properties />
                        <schema />
                        <parents>
                                <parent id="32" />
                        </parents>
                        <children>
                                <child type="operator" id="0" queueId="1" />
                        </children>
                </operator>
                <operator root="false" id="1"


className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQStreamSelectOperatorImp">
                        <classVariables>
                                <variable name="QMeshOperatorID" value="1" />
                                <variable name="IsEddyOp" value="false" />
                                <variable name="StreamID" value="0" />
                                <variable name="expression_id" value="93" />
                                <expressions>
                                        <expr id="91" type="TerminalExpressionStrImp"
                                                qPosition="0" cPosition="11" value="" valtype="column"
lid=""
                                                rid="" />
                                        <expr id="92" type="TerminalConstantImp" qPosition=""
                                                cPosition="" value="40.1" valtype="double" lid="" rid="" />
                                        <expr id="93" type="BinCOMPExpressionStrImp"
                                                qPosition="" cPosition="" value="" valtype="LT" lid="91"
rid="92" />
                                </expressions>
                        </classVariables>
                        <properties />
                        <schema />
                        <parents>
                                <parent id="32" />
                        </parents>
                        <children>
                                <child type="operator" id="0" queueId="0" />
                        </children>
                </operator>

                <operator root="false" id="0"
```

```
className="edu.wpi.cs.dsrg.xmldb.xat.common.querymesh.MQP.NASSQRouterOperatorImp"
        numberOfOutputQueue="32">
        <classVariables>
                <variable name="Num_Streams" value="1" />
                <variable name="Num_Operators" value="32" />
                <variable name="Num_SendOff" value="500" />
                <variable name="TupleCountThreshold" value="2000" />
                <variable name="Sharing" value="true" />
                <variable name="Stream0" QueueId="0"
                        window_type="CountBased" window_size="1500" />
                <globalDecisionTree>
                        <localQM id="1" stream_id="0">
                                <localDecisionTree id="1" stream_id="0"
                                        is_empty="true" />
                                <allRoutes>
                                        <route id="1" is_default="true"
                                                path="15|13|8|18" />
                                        <route id="2" is_default="false"
                                                path="15|12|19|29" />
                                        <route id="3" is_default="false" path="8|18|24" />
                                        <route id="4" is_default="false" path="8|18|26" />
                                        <route id="5" is_default="false" path="8|18|27" />
                                        <route id="6" is_default="false"
                                                path="13|8|19|27" />
                                        <route id="7" is_default="false"
                                                path="14|12|8|30" />
                                        <route id="8" is_default="false"
                                                path="16|12|31|9|20" />
                                        <route id="9" is_default="false"
                                                path="10|18|21|28" />
                                        <route id="10" is_default="false"
                                                path="17|10|25" />
                                        <route id="11" is_default="false" path="27" />
                                        <route id="12" is_default="false" path="24|28" />
                                        <route id="13" is_default="false" path="25" />
                                        <route id="14" is_default="false"
                                                path="20|7|27|23" />
                                        <route id="15" is_default="false"
                                                path="11|16|27|23" />
                                        <route id="16" is_default="false"
                                                path="20|16|23" />
                                        <route id="17" is_default="false"
                                                path="20|7|27" />
                                        <route id="18" is_default="false" path="4|7|27" />
                                        <route id="19" is_default="false" path="1|7|27" />
                                        <route id="20" is_default="false"
                                                path="20|1|27" />
                                        <route id="21" is_default="false" path="20|7|5" />
                                        <route id="22" is_default="false" path="20|7|6" />
                                        <route id="23" is_default="false" path="20|7|1" />
                                        <route id="24" is_default="false" path="20|7|2" />
                                        <route id="25" is_default="false" path="4|7|6" />
```

```
                                        </allRoutes>
                                </localQM>
                        </globalDecisionTree>
                </classVariables>
                <properties />
                <schema />
                <parents>
                        <parent id="31" queueId="30" />
                        <parent id="30" queueId="29" />
                        <parent id="29" queueId="28" />
                        <parent id="28" queueId="27" />
                        <parent id="27" queueId="26" />
                        <parent id="26" queueId="25" />
                        <parent id="25" queueId="24" />
                        <parent id="24" queueId="23" />
                        <parent id="23" queueId="22" />
                        <parent id="22" queueId="21" />
                        <parent id="21" queueId="20" />
                        <parent id="20" queueId="19" />
                        <parent id="19" queueId="18" />
                        <parent id="18" queueId="17" />
                        <parent id="17" queueId="16" />
                        <parent id="16" queueId="15" />
                        <parent id="15" queueId="14" />
                        <parent id="14" queueId="13" />
                        <parent id="13" queueId="12" />
                        <parent id="12" queueId="11" />
                        <parent id="11" queueId="10" />
                        <parent id="10" queueId="9" />
                        <parent id="9" queueId="8" />
                        <parent id="8" queueId="7" />
                        <parent id="7" queueId="6" />
                        <parent id="6" queueId="5" />
                        <parent id="5" queueId="4" />
                        <parent id="4" queueId="3" />
                        <parent id="3" queueId="2" />
                        <parent id="2" queueId="1" />
                        <parent id="1" queueId="0" />
                </parents>
                <children>
                        <child type="stream" id="0" name="Stream0" />
                </children>
        </operator>
</queryplan>
```