WPI-CS-TR-04-18

July 2004

D-CAPE: A Self-Tuning Continuous Query Plan Distribution Architecture

by

Timothy Sutherland
Elke A. Rundensteiner

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

D-CAPE: A Self-Tuning Continuous Query Plan Distribution Architecture

Timothy Sutherland, and Elke A. Rundensteiner
Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609
Tel.: (508) 831–5857, Fax: (508) 831–5776
{tims, rundenst}@cs.wpi.edu

Abstract

While practically all reported results on stream query engines are for central systems, it is apparent that due to the finite resources on a single query processor, future Data Stream Management Systems must distribute their workload to multiple query processors to meet the requirements of modern day query workloads and increasing volumes of data streams. This paper discusses a new scalable Distributed Continuous Query System (D-CAPE) that has the ability to distribute query plans over a large cluster of machines. We describe the architecture of the new system and policies and protocols for flexible query plan distribution and redistribution to improve overall performance. We also present techniques for self-tuning query plan re-distribution such as Balance and Degradation redistribution algorithms. D-CAPE's architecture is flexible, allowing different distribution algorithms such as Round Robin and Grouping Distribution and operator reallocation policies to be incorporated with ease. D-CAPE provides an operator reallocation algorithm that is able to seamlessly move an operator(s) across any query processor in our computing cluster.

The core contribution of this work is our extensive experimental evaluation using our software system, not a simulation. We observe that executing a query plan distributed over multiple machines causes no overhead compared to processing it on a single query processor, even for extremely lightly loaded machines. Distributing a query plan among a cluster of query processors can boost performance up to twice that of a centralized stream engine. Our experimental study uncovers that the limitation of each query processor within the distributed network is not primarily in the volume of the data nor the number of query operators, but rather in the number of remote data connections per processor. The overhead of migrating query operators is shown to be very low, allowing for a potentially frequent dynamic redistribution of query plans during execution.

Keywords: Distributed Streaming, Distribution Algorithms,Online Redistribution Algorithms, Experimental Results, D-CAPE.

1 Introduction

Recently, a growing area of research in the database community is the study of persistent queries over streaming data. This core functionality of data stream monitoring is being coined as *Continuous Query Processing*. A new effort is being undertaken by the database community to derive a new general class of continuous query engines called Data Stream Management Systems (DSMS). Data Stream Management Systems execute queries on data that is continuously arriving, and then return the result of the query to the end user in a real-time streaming fashion.

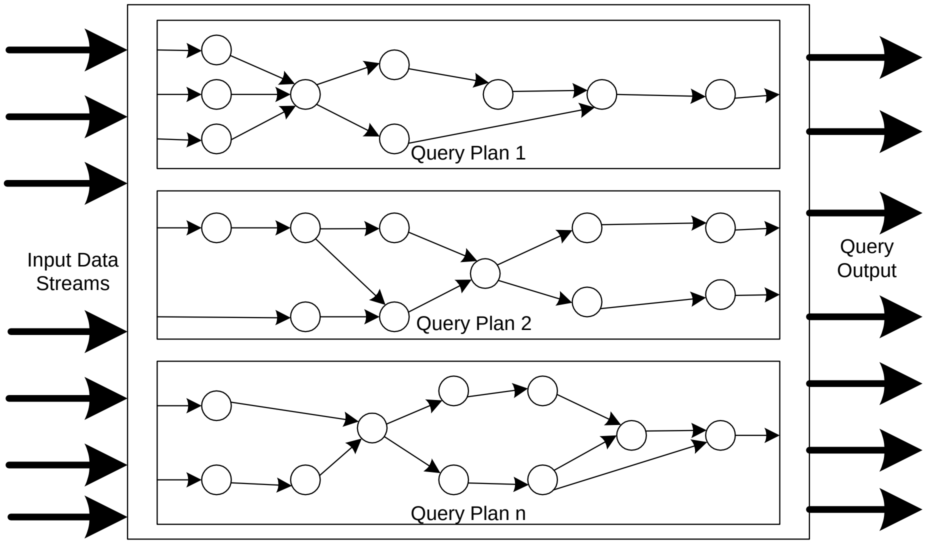


Figure 1. Traditional Continuous Query Processor

A DSMS may need to operate on several thousand queries at once given streams of data for applications such as online auctions, web servers, or to monitor stock market trends. The DSMS typically answers queries about the state of the data over a period of time, and all queries are based on a partial data set, as new data is always arriving. For instance the question *"What is the highest stock price on the New York Stock Exchange over the last two hours?"* can be answered by the DSMS, with the answer of the question always changing over time. This concept is different than the traditional database model where data is already in persistent storage and a query is asked based on this stored data. Traditional databases have the advantage of knowing how much data there is to query over, and that the data will not change (in most cases) during the query. On the other hand, a DSMS must be able to consume a partial data set and give a result based on data seen thus far.

1.1 Motivation

Current Continuous Query Systems such as Stream [6], NiagaraCQ [16], Aurora [1], and our WPI continuous query system, CAPE [33], operate over streams of data on a single processor and output results to the best of their ability. In order for Continuous Query Systems to operate in real-time, it is essential that all data is kept in main memory, as once data is written to persistent storage, the system slows down considerably. Current research that focuses on the issue of minimizing persistent storage use in a continuous query system includes Load Shedding [1], Operator Scheduling [7][37] and Operator-State purging [24].

The potential benefits and applications of data stream processing are becoming more apparent and popular for many applications in different business areas. These applications include monitoring remote sensors [30], and online transaction processing [39]. However, as the popularity of these systems increases, and more queries are registered and data streams grow larger, it will be *essential* to improve the processing power of these applications.

However, even with the current research, single CPU systems are not likely to be sufficient at handling future DSMS as 1) streaming data gets larger and faster as network speed improves, 2) queries get larger, more complicated and plentiful, and

3) more sophisticated and complex operators are incorporated into streaming systems (such as data grouping or statistical summaries).

Given the finite amount of CPU speed and memory on a single system, a distributed architecture will be better suited to handle the load, variability, and complexity of streaming data. Very recently, research is now under way in determining methods to distribute these query plans over a cluster of processing nodes [34][20].

1.2 Our Overall Approach Towards Data Stream Processing

To alleviate this problem, an extension of CAPE called D-CAPE has been developed that exploits a cluster of processors to aid in the processing of continuous queries. This paper discusses the implementation of D-CAPE, that extends our current DSMS, CAPE [33] to work over a cluster of query processors using a centralized controller. D-CAPE is designed to effectively and efficiently distribute query plans and monitor the performance of each query processor with minimal communication between the controller and query processors. We bundle synchronization messages, thus minimizing packets sent between query processors. These messages are also incremental at run-time to aid in minimizing the communication cost of D-CAPE. We process data by creating "pipes" between query processors to allow the data streams to flow, and then filling these pipes with data streams once execution begins. It can also reallocate query operators, or complete sub-plans to a different query processor at runtime during times of heavy load, or if it is determined by D-CAPE, using a cost model, that the reallocation will boost the performance of the DSMS. D-CAPE has a specialized algorithm for reconnecting these pipes during the reallocation process, to ensure no data is lost and that data never stops flowing through the query plans.

1.3 Contributions

This paper contributes to the advancement of Data Stream Management Systems in the following ways:

- A well-designed, distributed architecture called D-CAPE has been created for continuous querying that allows for flexible query allocation and distribution strategies.
- D-CAPE is scalable allowing for distribution of query plans among any number of query processors by using a multi-tiered controller architecture.
- D-CAPE allows for any number of distribution algorithms to be easily plugged into our system. For our current system, we developed two distribution algorithms, Round Robin Distribution and Grouping Distribution, to analyze the ways different query plan distributions affect query processor performance.
- D-CAPE allows for any cost model to be created for monitoring each query processor. These cost models can use the statistical data that is recorded by each query processor about the data, query plan or individual query operator in determining the workload for a query processor.
- D-CAPE has the ability to actively monitor each query processor to determine its workload at runtime and reallocate any number of query operators to *any* query processor in the processing cluster.
- D-CAPE also allows for any operator redistribution policy to be implemented that is independent from the cost model used to determine workload. This gives D-CAPE the flexibility to allow any redistribution policy to operate using *any* cost model.
- D-CAPE implements a *new* operator reallocation algorithm that is able to move operator(s) across any query processors in the computing cluster without interrupting the data flow or query processing on any of the involved processors.
- The original CAPE DSMS was improved by creating new components to boost performance, and also by removing and optimizing other components. The improvements sparked a 10% jump in query processing performance from the original CAPE design.
- Our experimental studies confirm that a DDSMS can effectively parallelize the execution of query operators even during periods when a processing node is not filled to capacity, thus improving performance even for small query plans.

- Experimental studies find that our DDSMS allows for large query plans to be processed efficiently; up to 100% faster than a typical DSMS. In some cases a centralized DSMS fails because of the lack of processing power.
- We show experimentally that the initial distribution algorithm used for distributing query plan workload plays a significant part in the overall performance of the query plan.
- Experimental studies also confirm that the overhead for redistributing an operator is negligible. This allows our D-CAPE architecture to reallocate a query operator or an entire query sub-plan to any query processor in the cluster.
- Experimental studies also show that D-CAPE can effectively monitor each query processor and reallocate query operators to improve the overall performance of the query plan. We find that operator allocation can improve performance over a distribution algorithm alone by up to 100%.

1.4 Outline

First, in section 2 we will discuss the current work in DSMS system and also earlier work in Distributed DBMS systems. Many of these concepts will contribute to our new D-DSMS design. In section 3 we will briefly discuss the background of Data Stream Management Systems. We will show an example query for which a DSMS is used, and outline a DSMS operator, and how it is different from the traditional SQL operator. In section 4 we will outline the design of our new D-DSMS, D-CAPE. We will experimentally show that the network overhead for our design is low, and illustrate steps that were taken to minimize the overhead of our design. section 5 discusses the initial distribution of query operators among a cluster of machines, and the observed costs of distribution over the query processor cluster. We will experimentally show the performance differences in the type of distribution algorithms used in D-CAPE. In section 6 we discuss operator reallocation, and our mechanism for determining the workload of a query processor. We discuss *which* operator to move and *where* to move it. Using experimental results, we find that we can effectively monitor query processors and improve query performance by using our operator reallocation strategies. Finally in section 7 we outline our conclusions and future work.

2 Related Work

In this section we will briefly discuss some areas of related work in both Data Stream Management Systems and also other areas that utilize distribution techniques, such as operating systems and traditional Database Management Systems. This related work serves as a starting point for creating our own D-DSMS.

Current Data Stream Management Systems. Data Stream Management Systems are gaining tremendous popularity in the Database field as remote data streams become available via sensors and monitors, and as the type of query is *persistent*. That is, the query is always running in the system and is returned to the user in real-time. A DSMS also introduces many new and interesting problems [6][12] in current research such as high volumes of input data [14][26], operator scheduling [7][13] and data filtering [31]. There have been many systems proposed, each of which contributes differently to this growing field.

Aurora [1] is a Data Stream Management System that most closely resembles our work. Aurora allows a user to register several continuous queries, and monitor those results through their built-in GUI. They treat query operators as “boxes” which process streams of data. Aurora’s main contribution to this area of research is the ability for its system to schedule its boxes and manage data between memory and disk using Qos-based priority information. A user is able to input a graph representing what the ideal QoS for the query should look like. Aurora is able to adjust its execution (the order of boxes scheduled and which data is stored persistently) based on this QoS. Aurora also introduces *load shedding* to cope with degradations of QoS in periods of bursty data arrival.

NiagaraCQ [16] is a scalable DSMS that aims to scale the number of queries that a DSMS can handle by grouping together common parts of a query plan and also using selection operators to its advantage by reducing the amount of intermediate data in the system. They show that by using this grouping strategy, they achieve scalability in the order of thousands of queries. This work is complementary to ours, as we can take advantage of their query plan grouping strategies to give our system even further processing power.

STREAM [6] is a DSMS whose focus is on effectively processing data streams with bursty arrival rates. If the input rate is high, the system approximates query results after shedding some data. They have developed the Chain [7] operator scheduling algorithm that has been shown to be near-optimal in minimizing the memory footprint of the system. They have also created a Continuous Query Language (CQL) [3] that current DSMS implementations can use when defining continuous queries, more importantly defining clear semantics for continuous queries.

TelegraphCQ [14] is a DSMS whose main contribution is the study of continuous queries with widely varying data rates and sizes. TelegraphCQ brings us the notion that a DSMS must *react* to data arriving into the system, rather than manage data that is already contained within the system. Telegraph utilizes an adaptive processing technique called Eddies [5] that allows a flexible routing technique for tuples between operators. TelegraphCQ also spun off another DSMS called PSoup [15] that has the ability to integrate streaming data with data that has already been captured to disk.

CAPE [33] is being developed here at WPI. Much of our work is very similar to that of Aurora and Stream. We also model the query plans as a dataflow graph where operators are connected by data pipelines. However, instead of improving performance by approximation[6] or load shedding[1], we aim to improve system performance and minimize resources by adapting at different levels of query plan execution. At the lowest level we can adapt within a query operator using punctuations [24]. At the query plan level, we support query plan migration [41] and adaptive scheduling techniques [37].

Distributed Data Stream Management Systems. Flux [34] is a new dataflow operator introduced in TelegraphCQ to allow to adaptively partition an expensive operator such as a Window Join [25][28]. Flux encounters many of the same problems that our D-CAPE system will encounter when reallocating query operators. That is, we have to have a mechanism for moving the *state* of a query operator to ensure that no data is lost or miscalculated by the operator. We move our state in a similar manner to the Flux operator. We first stop the input queue from the operator. We then marshall the state to send it across the network, and then unmarshall it after it is received by the second query processor. Once the state is unmarshalled, we allow the operator to run, which will pick up seamlessly because the state will be the same as the original operator. The Flux operator can complement our D-CAPE system by adaptively partitioning our stateful operators.

Aurora* and Medusa [9][20][26] is the first published work in creating an architectural model for a D-DSMS. Several necessary design challenges are discussed, including such aspects as the Query Model, Run-Time Operation, Routing Rules, Message Transport Protocol and Load Management. They propose a "push pull" architecture where query operators may be reallocated to only neighboring processors so as to not interrupt the data stream. That is, there is no central controller that synchronizes all of the query processors. Instead, each query processor can communicate with their neighbor when they have a high workload and push an operator to that neighbor. They also propose an *operator splitting* strategy where a query operator may be replicated among several machines to improve the performance of the operator, similar to Flux.

By working in this "push pull" architecture, Aurora* limits the options that the DSMS has when there is a very high workload on multiple machines. It is quite possible to have a cluster of machines where one machine is empty, but since Aurora* only considers neighboring processors the machine will not get utilized. They also do not provide a mechanism to move a set of nodes or a query sub-tree at once.

Our system, D-CAPE, is similar in nature to the Aurora* system, however, we do not place any restriction on the location of where a query operator may be reallocated to. Our architecture will allow operator reallocation across any two query processors without a loss in data flow or data contents. Unlike Aurora*, D-CAPE utilizes a centralized controller. The centralized controller allows D-CAPE to monitor *each* query processor and consider the global ramifications of moving query operators. We also show that while the controller is centralized, it is still scalable to hundreds or more query processors. We also allow our controllers to be multi-tiered such that we can have multiple controllers, each controlling a cluster of machines that may have have similar queries or clusters that are all in the same location.

Instead of focusing on operator splitting as Aurora* has done, we aim to first analyze what effects the network has on query plan distribution and how we can exploit advantages in query plan execution. We then plan, as future work, to alter our query model to allow for operator replication while still complimentary to our architecture. There is also other work in pipelined query execution [40] where non-blocking query operators can be pipelined to improve performance. In our work, we will be able to pipeline operators because they are non-blocking, but also process them in parallel across the cluster of query processors.

Distributed Database Systems. Also closely related to this area of research is that of distributed database systems. We are able to use many of the principles [22] used in early research for distributed database systems [22] such as Bubba [2], Gamma [23], and Tandem [38]. There are three main types of distributed database systems: Shared-Disk, Shared-Nothing and Shared-Memory architectures. The main advantage of the Shared-Nothing architecture is scalability. This architecture can be scaled up to hundreds or even thousands [22] of processors. This is possible because they do not interfere with one another. The Shared-Nothing architecture is also most advantageous in environments where the data is partitioned. By having partitioned data, multiple resources need not share the same disk to read the data. Also, by having non-blocking operators we are able to maximize parallelism since operators need not consume an entire dataset before returning output results.

In D-CAPE, we model the architecture after the Shared-Nothing approach in [22]. DSMS systems will need to be scalable, as the number of queries and the amount of stream data grows larger. Since the data streams are *naturally* partitioned, it is easy for D-CAPE to redirect the data to the proper query processor without affecting any other query processors in the cluster.

This Shared-Nothing approach maximizes query execution, since each query processor only manages data that it needs to complete the query. D-CAPE also makes uses of non-blocking operators which will aid in parallelism if a single query plan is distributed among several query processors. Parent operators in the query plan will be able to consume data that was output from the children, even though it is a partial answer. This will improve the performance of our DSMS.

Dynamic Load Balancing. There is also a lot of research in the area of Dynamic Load Balancing from Distributed-DBMS systems [10][11][32] that discuss issues such as: data consistency, reallocation techniques, and communication costs. We find that these issues are similar in the context of our DSMS, and our architecture will have to be designed in such a way to minimize network costs [10][29] and the number of threads our system utilizes [11]. Because of these observed factors, we will create query operator distributions that will aim at minimizing the number of network connections per machine, which will aid in minimizing the volume of data over the network and also the number of threads in D-CAPE, as each network connection will require a thread from the Operating System.

There is also work in the area of Dynamic Load Balancing of Web Servers [17][18][19][21] which use a central controller for communicating with each processing node. These systems typically use a Round-Robin approach [17] for process scheduling or even a QoS-aware approach [18]. The advantages of the Round-Robin approach is that every machine is *guaranteed* to have work to do. The disadvantage of this approach is that each process may have different sizes and thus Round-Robin may not be optimal in cases where many large processes are scheduled on one processor. QoS-aware approaches will typically perform better, however there is more work in determining *how* to determine quality of service and further determine *which* processing node is performing up to a certain QoS level. We use the Round-Robin approach as one of several possible distribution patterns in our work to understand how even a simplistic approach will fare in our DSMS domain. Future work will include designing a QoS-aware distribution algorithm.

The Web Servers in [18][19][21] all use a central controller that may be tiered depending on the number of processing nodes. This is similar to our D-CAPE system where we allow different controllers for a cluster, and using a second-level controller on top of each cluster controller. These works outline steps that can be taken to minimize the communication between a controller and its processing nodes, such that the controller does not become a bottleneck. In D-CAPE we can utilize a similar approach to processor communication, however the type of communication in a DSMS will be quite different. Web Server controllers process *fixed size* jobs for each processor whereas D-CAPE query processors have to execute on queries of *varying size* because of data variability.

3 DSMS Background

In this section we will discuss the background of Data Stream Management Systems. First, we will present a streaming data example. We will then discuss the query plan of a DSMS and how a new class of query operators are necessary for data streams. Finally, we will discuss the basic non-distributed architecture of our DSMS, CAPE, the Constraint-aware Adaptive Processing Engine.

3.1 Example Stream Query

In order to understand the realm of queries that a DSMS is built to answer let us look at an example. Consider a traffic grid as shown in Figure 2. Each sensor, as indicated in Figure 2, collects the data shown in Figure 3.

The sensor data collected is just a sampling of the data streams produced by the sensor. It is important to note that not only may there be many sensors, but there also could be multiple feeds from each sensor, recording different types of data. Our query will make use of the Traffic Flow data. We will use Continuous Query Language (CQL), a query language similar to SQL, that extends traditional query semantics by allowing for time based joins and aggregation along with other features [].

Suppose we ask the query: ”Return all cars and their current MPH that have travelled down Road 2 and taken a LEFT turn onto Road 1 within a 2 minute time period.” Using CQL we would have the following specification:

```
SELECT R1.carID, R1.MPH
FROM Sensor2 as R2 [Range 2min], Sensor 1 as R1 [Range 2min]
WHERE R1.carID = R2.carID AND R1.type = "Car";
```

The corresponding query plan is seen in Figure 4. This query plan consists of a Join, Select, and Project operator, similar to SQL query operators. The functionality of these operators will be discussed in Section 3.2. In Table 3.1 we see example data that may be collected by the two sensors. T represents the *timestamp* associated with the data.

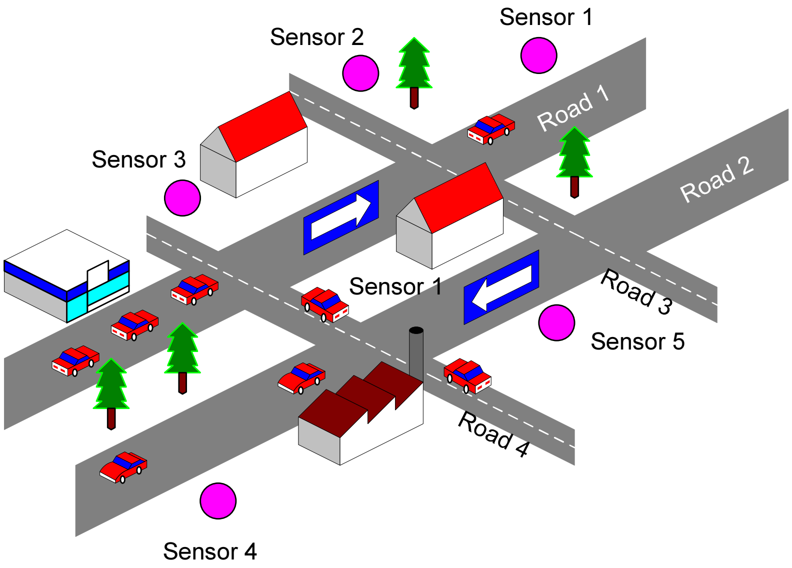


Figure 2. Example Traffic Pattern.

```
Traffic Flow Schema {
Time timestamp,
String carID,
Type type,
int MPH
};
```

Figure 3. Traffic Sensor Schema

Applying the query plan in Figure 4 on the input data in Table 3.1 we will process the data in the following manner.

The Join operator is responsible for joining any two tuples that occured within 2 minutes of each other and also pass the join predicate *R1.carID = R2.carID*. The output of the join operator is then fed into the Select operator, which filters out all tuples that do not pass the predicate *R1.type = "Car"*. Finally, the Project operator projects the columns *R1.carID, R1.type* which can then be returned to the user. Data is continually processed as more data is received from the Sensors, until the query is removed from the system.

3.2 Streaming Query Operators

In our example in Section 3.1 we saw two different types of operators in our query plan. These can have different characteristics than a traditional SQL-type operator. First, some streaming operators have *state* which is maintained by the operator during runtime. This state is the data that must be remembered by the operator to complete its operation. A join operator as in Section 3.1 is an excellent example of such an operator. The join operator is responsible for remembering all tuples that have arrived within the last two minutes. Secondly, since streaming operators have different characteristics than traditional query operators, we have to alter the implementation of traditional operators and add other semantics to allow them to process for real-time continuous data. They must be able to output incremental results as new data arrives from each stream. We can break streaming query operators down into two categories: stateful and stateless. We must give some operators the ability to remember what it has done in the past via an internal state, while other operators do not need this ability.

Every operator o has N_i input queue(s) and O_j output queue(s). The amount of the input can be defined as $n = \sum_{i=1}^{N_i} n_i$,

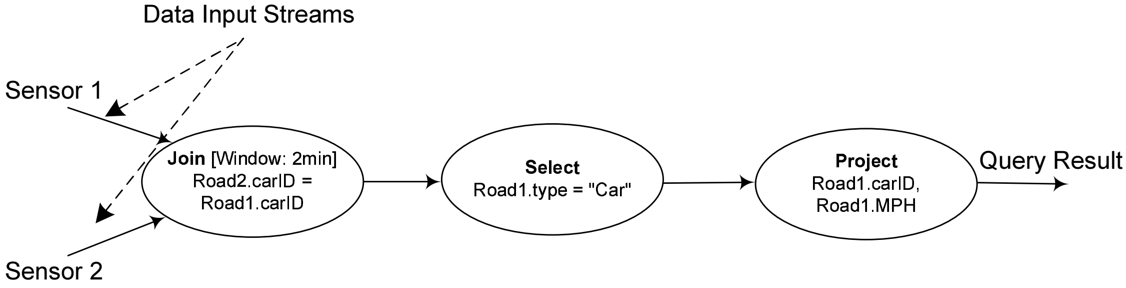


Figure 4. Query Plan Constructed from CQL Statement.

Sensor 1			
T	carID	type	MPH
0	9034 TR	Car	55
0	FED 1	Truck	42
1	SOXFAN4	Car	50
1	8325 DL	Car	35
1	345 DGE	Car	65
1	UMASS1	SUV	45

Sensor 2			
T (min)	carID	type	MPH
0	1345 FD	Car	34
1	MV 1223	Truck	53
2	SOXFAN4	Car	65
2	1492 CC	Car	32
3	UMASS1	SUV	23
4	1353 DW	SUV	56

Table 1. Example Traffic Data.

where n_i is defined as the amount of input at the i^{th} queue. The quantity of the output can be defined as $m = \sum_{j=1}^{O_j} m_j$. Similarly, m_j is defined as the amount of output in the j^{th} queue. The term m/n is known as the *selectivity* (σ) of the operator, more simply known as the probability of a tuple passing the *predicate* (ρ) of the operator. The selectivity is an important attribute of a query operator since it directly controls the number of tuples outputted to its parent. Operators with smaller selectivities tend to improve query plan performance because the number of tuples are reduced, thus reducing the total number of tuples to be processed. There is also fixed cost for reading/writing to queues, which we will define as ω.

3.2.1 Stateless Streaming Query Operators

Stateless operators are similar to traditional DBMS operators, since they have the ability to perform without needing to know what they have done in the past. Typical stateless operators include: Select, Project or XMLTagger.

In Figure 5 we show how a stateless operator processes data. The processing cost associated is linear in the volume of input data. The larger the input data, the longer it will take to process the data. For every n tuples that are dequeued, n tuples are subsequently evaluated and then the m tuples that are evaluated to true are placed in the operators output queue.

Project and XMLTagger operators have a selectivity of 1 while the selectivity of a Select operator varies between zero and one depending on the select predicate. The processing costs associated with a stateless operator are shown in Equations 1 and 2. As you can see, the only variant in the cost is the number of input tuples, since the processing cost is fixed, and determined by the type of the operator and the speed of the query processor doing the work.

Output of Join Operator						
T (min)	R1.carID	R1.type	R1.MPH	R2.carID	R2.type	R2.MPH
(1,2)	SOXFAN4	Car	50	SOXFAN4	Car	65
(1,3)	UMASS1	SUV	45	UMASS1	SUV	23

Output of Select Operator						
T (min)	R1.carID	R1.type	R1.MPH	R2.carID	R2.type	R2.MPH
(1,2)	SOXFAN4	Car	50	SOXFAN4	Car	65

Output of Project Operator		
T (min)	R1.carID	R1.MPH
(1,2)	SOXFAN4	65

Table 2. Output from Query Plan

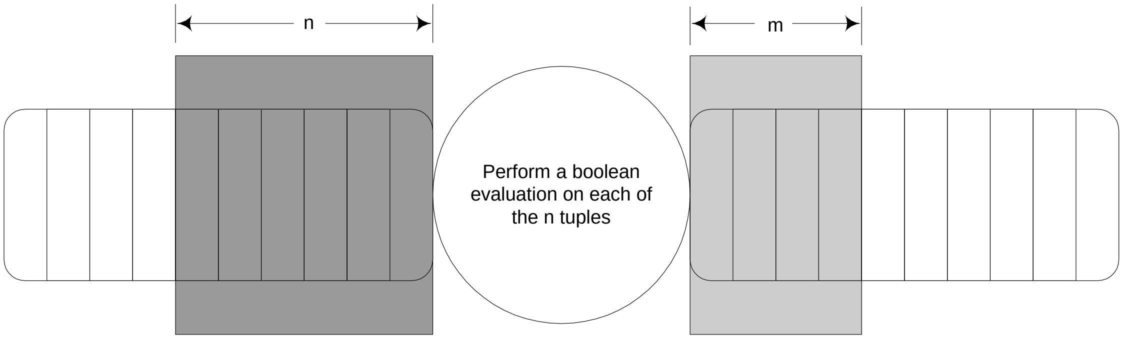


Figure 5. Single Stream Operator.

$$\begin{align} cost &= (n * \omega) + (\rho * n) + (\sigma(n) * \omega) \tag{1} \\ &= n * (\omega + \rho) + (\sigma(n) * \omega) \tag{2} \end{align}$$

3.2.2 Stateful Streaming Query Operators

Stateful operators retain all tuples that are still in the query “window” of acceptance by the user. Using the query in Section 3.1 an example window would be 2 seconds. There are many semantics for determining how to calculate a window for an operator including Moving Window [8] and Sliding Window [28]. For our purpose, we will assume that our operators utilize a sliding window. In a sliding window, all tuples occuring within t time units of each other are in the same window. The window “slides” as new data is read into the operator that have higher timestamps.

Figure 6 illustrates the processing of data in a binary window join operator. Execution proceeds as follows. First we dequeue n_1 tuples from the left queue and n_2 tuples from the right queue. We then purge (ψ) the state of the left (s_1) and right (s_2) states from the total number of states S by looking at the first tuple dequeued from both queues, respectively. If the first tuple in the left queue is out of the window of the first tuple in the right state of the operator, we purge those tuples out of the right state. The tuples can’t possibly be evaluated to true due to being out of the window. We do the same for the right queue and the left state. We then perform a join on all of the tuples from the left queue with the right state. We then move the tuples from the left queue into the left state, since they have been evaluated. We can now join the tuples on the right queue with the left state, and once finished move the tuples from the right queue to the right state.

The processing cost is much higher for a stateful operator. Clearly, the size of the time window has a direct effect on the cost. The larger the window, the larger the cost of the operator, as we will spend more time purging tuples out of the state

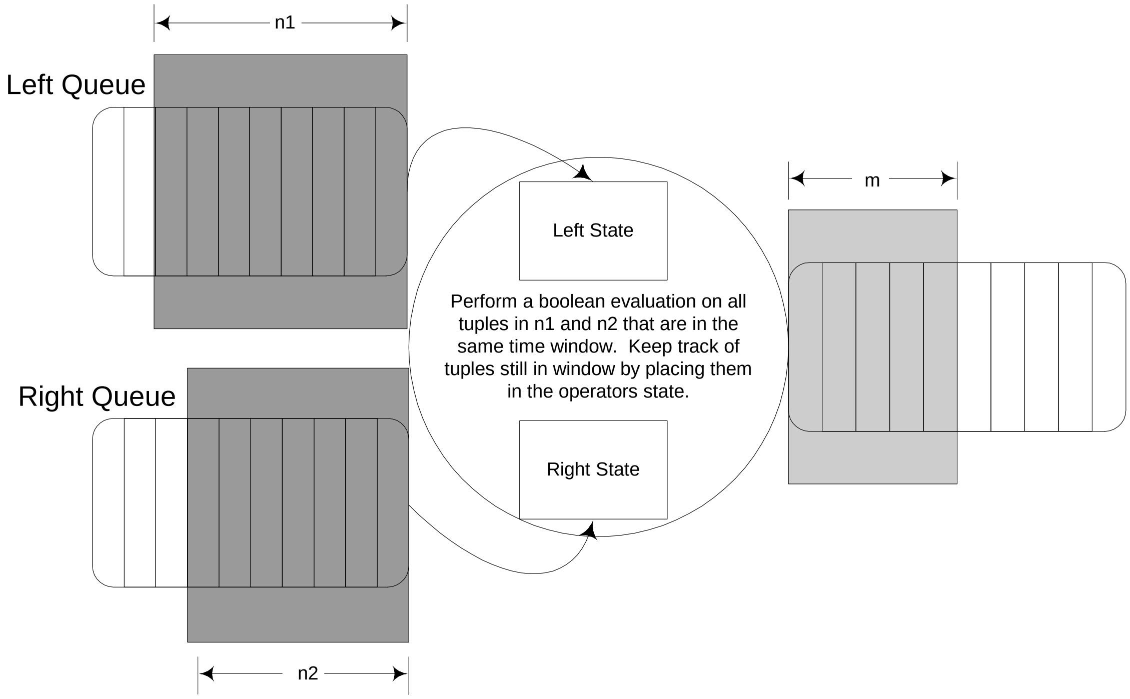

n1
Left Queue
Left State
Perform a boolean evaluation on all tuples in n1 and n2 that are in the same time window. Keep track of tuples still in window by placing them in the operators state.
m
Right Queue
Right State
n2

Figure 6. Multi Stream Operator.

and more time processing the join. The cost of a stateful join operator is shown in Equation 3

$$cost = (n * \omega) + (\psi * \sum_{i=1}^{|S|} s_i) + \rho * (n_1 * s_2 + n_2 * s_1) + (\sigma(n) * \omega) \quad (3)$$

The total cost includes the time it takes to read (ω) the n tuples from the input queues, the time it takes to purge the state, and the cost of evaluating each join predicate and writing those that pass to the output queue. Most of the processing cost of the stateful operator is that of purging the two states and the time it takes to evaluate the join predicate. Many join implementations aim to improve the cost by using hash-based states or hash-joins. Nonetheless the operator still proves to be far more costly than a stateless operator, especially as the state size increases.

3.3 The Data Stream Management System: CAPE

Now we introduce our DSMS, CAPE. CAPE is a continuous query system developed at WPI [33]. It can process any number of user queries on multiple streams and report the resulting data to the user applications. This core architecture is similar to that of [1][6] [16].

Each query is translated into an algebraic query plan as shown in Section 3.1 that then is processed by our runtime engine. The query plan can be thought of as a directed acyclic graph, where the nodes represent query operators and the edges represent queues. The operator(s) that connect directly to the end user application(s) are called the *roots* and those that connect to an input stream are called *leaves*. Each leaf is directly connected to an external data stream where the source data is generated, typically by a remote computer or data sensor. All query operators in CAPE operate in a pipelined, non-blocking manner [40]. That is, every operator is capable of producing results after consuming a partial input data set. Figure 4 illustrates an example of a query plan. Intra-operator data results are stored as tuples in main memory queues. Queues serve as the connections between operators and define the routes that tuples take during execution.

CAPE is made up of four primary components as shown in Figure 7. The *Stream Receiver* is responsible for receiving data from all Stream Sources and placing the tuples in the query plan. *Stream Sender* is responsible for sending the result data to the end user. The *Statistics Gatherer* stores, calculates, and sorts statistics about any part of a query plan, such as operators, queues, and entire query plans. These statistics can be used for many types of calculations in the system, such as deciding how well a particular query plan is running given a cost model, or even simply how many tuples are in main memory at a given time.

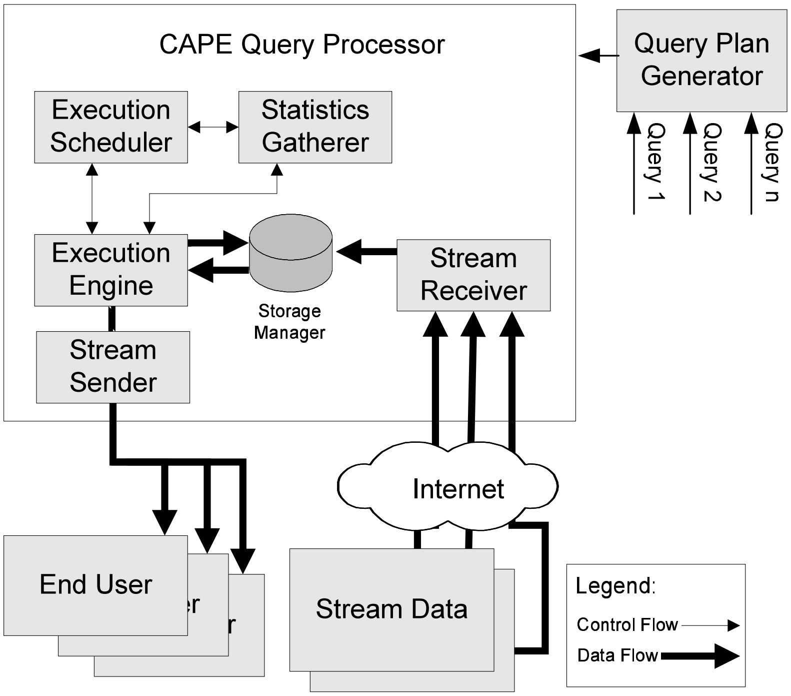


Figure 7. Architecture of CAPE Continuous Query System.

The *Execution Scheduler* is responsible for deciding which operator should be executed at a given time. Several different scheduling algorithms, including Round Robin, First In First Out, and Chain [7] have been incorporated into CAPE. These algorithms use statistics that are gathered from the query plan to determine which operator to schedule next. CAPE has its own novel scheduling strategy, which is referred to as an *Adaptive Scheduler* [37]. The Adaptive Scheduler dynamically selects which scheduling algorithm to run during execution based on how the current scheduling algorithm is performing with respect to the other scheduling algorithms that are available for use. This is CAPE's approach to provide the best possible service on a single machine. The scheduler can improve performance based on various requirements, such as minimizing memory or maximizing the output rate. The *Execution Engine* lies at the heart of CAPE. It is responsible for overseeing the execution of the query plan. The Execution Engine tells the Statistics Gatherer to obtain the latest statistics, and asks the scheduler which scheduling algorithm should be used next. In essence, it is the engine of CAPE that uses information obtained from the other modules to run the system. Here is a brief walkthrough of the Execution Engine's tasks during execution:

1. Ask the current scheduling algorithm to choose the next query operator, Op, to run.
2. If the workload for $Op > 0$, then update the statistics for Op's input and output queues and pass the workload to the operator. If the workload $= 0$, then there is starvation and the strategy will pick another operator.
3. Run the operator. When the operator has processed all of its assigned work, control is returned to the Execution Engine.
4. Ask the Statistics Gatherer to update statistics for various operators and other query plan information.
5. Repeat steps 1-4 for the duration of the query.

4 D-CAPE: The Distribution Stream System Architecture

Next the design of the new D-CAPE system will be introduced. We will first discuss the assumptions and restrictions for this version of implementation of the system. Then we will discuss the general architecture of the system.

4.1 Assumptions and Restrictions

Several assumptions are made in this work so to allow us to focus on the most important concerns of this new system. First, it is assumed that all processors have 100% up-time, and the distributed system will not have to worry about an unresponsive processor. If a query processor is to fail, it is remedied by moving the workload that was on the unresponsive processor to another query processor. Using this assumption, data will be lost, and future work will be needed to come up with ways to recover this lost data, similar to [26][35]. In our experimentation, if a query processor were to fail, we restart the experiment so the experimental results are not tainted with this loss of data. Also, it will be assumed that the query plan is already optimized using query re-write rules, and that each operator is scheduled using the same scheduling strategy. That is, it is not the goal of the new distributed system to achieve better performance on individual machines, but rather to improve overall performance based on the distribution techniques.

4.2 Distributed Architecture Overview

The most important aspect to this paper was to develop a flexible architecture that could be used in future versions of the system. Without a sound architecture, the shelf-life of this system will be short lived. Our main goal was to allow this work to be used for a long period of time as a foundation for improving data stream processing performance.

In 1992 David DeWitt and Jim Gray outlined the architecture necessary to create parallel database systems [22]. They found that query plans can be more efficient if running in a parallel, pipelined manner by using the natural data flow tendencies of a query plan and distributing query operators. Database Management Systems did have a major flaw when it came to pipelined execution: Most of the existing implementations of query operators were blocking. However since Data Stream Systems have developed non-blocking operators [25] that continuously provide output data, we are now in a position to take advantage of paralelized pipelined query execution.

DeWitt and Gray go on to say that another important requirement is that partitioned execution needs partitioned data. Partitioned data allows for easy data transfer, without the need to scan incoming data to determine where it belongs. This is an easy requirement for our DSMS to meet, since the data streams are already generated over multiple machines, and can be redirected to any query processor in our network, without affecting other processors and their execution.

We found that the requirements needed over 10 years ago are still needed today. By parallelizing execution and directing data streams to individual query processors, we are able to improve query execution, as will be seen in experiments throughout this section. We developed a robust, component-based approach to designing this architecture. It is developed in a shared-nothing manner; that is, the only way data is shared is through the Interconnection Network. Each CPU and Memory is private to each query processor. Figure 8 illustrates the basic architecture of the new D-CAPE System.

This is an extended version of the original CAPE system introduced in Section 3.3. Similar to the original system, there is still a Statistics Gatherer, Execution Engine, and Execution Scheduler and Stream Sender/Receiver. We also added two new components to the query processor itself, including the *Connection Manager*, and *Stream Feeder*. These components will be discussed in detail in Section 4.3.

The Distribution Manager resides on a machine separate from all the query processors and is responsible for communicating with each query processor to tell it what data streams and query plan operators it is responsible to process, and where to send it when it is done. This is achieved in four steps and will be discussed in detail in Section 4.3.

1. Send initial configuration information to each query processor

2. Distribute query plans among the query processors using a *distribution pattern*, a way to distribute query operators among a cluster of machines based on query plan properties.

3. Listen to status updates from each query processor by receiving packets of statistical data needed to calculated the workload of a processor.

4. Determine if any of the query processors has too high of a workload and redistribute it, if necessary.

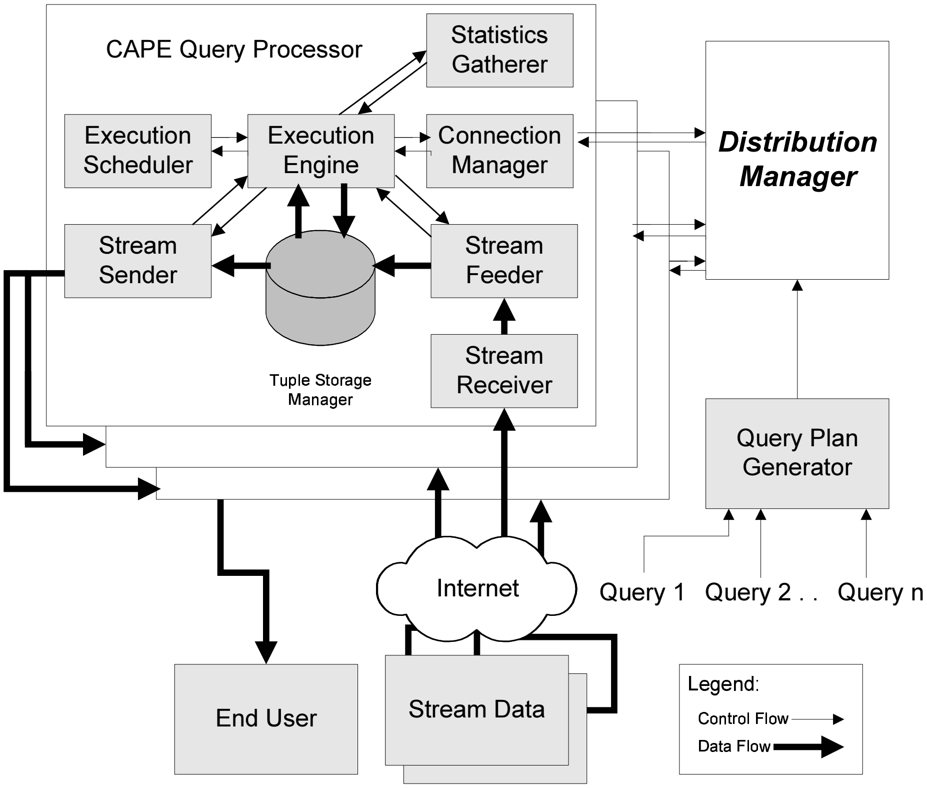


Figure 8. D-CAPE Architecture

Each of these steps will be discussed in detail in sections 5.1 to 6.3.

To increase the potential scalability of a D-CAPE, we have created the Distribution Manager such that it can operate in a tiered environment. Figure 9 illustrates how the Distribution Manager can operate in such an environment. In the future, it may make sense to have clusters of machines in different locations process different workloads. In this case, it may not make sense to have a single distribution manager manage clusters across the Internet. Instead, we can create one distribution managers for each cluster location, and then have a distribution manager on a higher tier that is responsible for allocating query plans to each distribution manager in the lower tier. This way, we have the flexibility of distributing the query plans on any of the query processors available to us, yet we can also eliminate network update costs by localizing distribution managers to work more closely with a particular processing cluster.

4.3 A D-CAPE Query Processor

Before discussing the Distribution Manager in detail, we first go into the details of the query processor, and in particular, the improvements that were made for D-CAPE. At the end of this section, we will show experimental studies that show the limitations of a query processor.

As shown in Figure 8 there are seven main components in the query processor: The Execution Engine, Statistics Gatherer, Execution Scheduler, Stream Receiver, Stream Distributor, Stream Feeder, and Connection Manager. Each of these components are integral to the execution of the query plan. Furthermore, it is important that these components are implemented in such a way as to maximize performance. Each of these components communicates with one another to minimize the cost of context switching between components. We will now discuss each component and how they have been improved for D-CAPE.

Execution Engine. The execution engine was improved from Section 3.3. It now makes fewer calls to the statistics gatherer to maximize processing time. We also implemented the ability to record the statistics upon completion into a

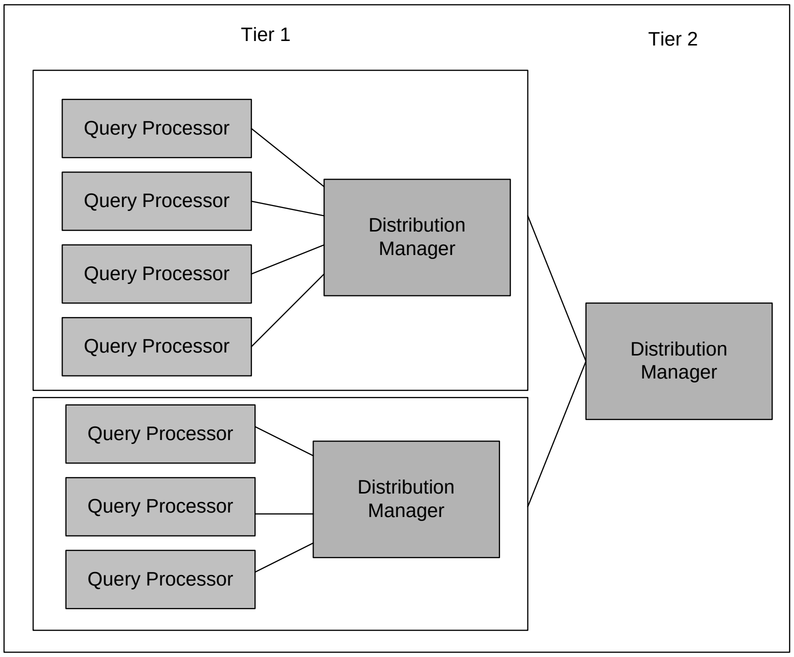


Figure 9. Example of D-CAPE configured to run in a tiered environment.

Microsoft Excel Spreadsheet for quick analysis by the user upon query completion.

Statistics Gatherer. The statistics gatherer was improved from Section 3.3 by reducing the number of calls it would take to insert/remove data to/from the statistics gatherer. Over time this should increase performance to some degree, especially for long running queries.

Stream Feeder. The stream feeder is responsible for taking tuples received by the stream Receiver and placing them in the proper input queue of the operator. This is a hash-based implementation. As a tuple is taken from the pool of received tuples, its corresponding queue is looked up in a hash table to determine where the tuple belongs. Once this tuple is enqueued, the thread will let the Execution Engine know that there is more data to process. The Stream Feeder is a thread designed to ensure that all queues in the query plans have data to process. This way if one stream has a higher data rate, and thus a queue that is more full, we can wait to put that data in an input queue until the operator will actually need it.

Stream Receiver. The stream receiver is implemented in a separate thread. As tuples are received, tells the stream feeder that there is new data waiting to be fed into the input queues. By implementing the tuple receiver in this manner, the stream feeder is only running when there is actually data to process, so CPU cycles are not wasted in doing empty work.

Stream Distributor. The stream distributor is responsible for sending tuples to the next query processor or to an end-user application. This too is hash-based; hence lookup takes a constant time. The distributor waits for a message from the Execution Engine, indicating that there are tuples to be sent across the network. By waiting for a message from the Execution Engine, CPU Cycles are not wasted on the Tuple Distributor when there is no work to do.

Connection Manager. The Connection Manager is the interface between the query processor and the distribution manager. It is responsible for handling requests such as activating operators on the processor, or sending the current status of the machine to the distribution manager. Table 4.3 lists the different connection requests that can be made to the Connection Manager.

The Connection Manager has been designed to allow for an expanded set of commands to be implemented for the future. The most important job of the connection manager is to be available and to respond quickly.

Type	Description
Activate	Activate an Operator.
DeActivate	Turn off an Operator.
SendData	Send data to another QP.
ReceiveData	Receive data for processing.
StopSend	Stop sending data to another QP.
StopReceive	Stop listening for data.
SendStatistics	Send one or more statistics to the DM
Shutdown	Shut down the query processor
Restart	Remove all query plans and data, wait for new plans from the DM.

Table 3. Connection Request Types.

4.4 Calibrating Query Processor Performance Characteristics

It was very important to understand the limitations of a query processor. In particular, there were three questions that needed to be answered:

- How often can the Distribution Manager communicate with a Query Processor?
- How do the new components utilize the CPU? Is it better or worse than the old implementation?
- How many input/output connections can a Query Processor Handle?

To answer these questions, an experimental test-bed was developed. The test-bed consists of a cluster of 10 machines, each with dual 2.4 GHz processors, 2 GB of memory, on a Gigabit Ethernet connection. We utilized two machines to stream data, two machines to listen to query results, one machine to act as the distribution manager, and 5 machines to act as query processors. We use two stream generators so we can send a higher volume of data across the cluster. The data consists of the server logs from the 1998 World Cup website [4]. In our 30 minute experiments, approximately 72,000 tuples are sent *per* stream. Our query plans connect to a minimum of six streams and a maximum of thirteen. Our query plans consist of window join operators and single stream operators in different configurations, ranging from 5 operators to 80. Our join operators have a selectivity of two, that is it outputs twice as much data that is input. Our single streams operators have a selectivity of one, to make its cost as high as possible. The query plans themselves are binary trees (representing many joins linked together) with a height of at least five and a breadth of at least six.

For these experiments, we only utilized one query processor so it could be tested against the original version of CAPE, and also to find the limitations of a single processor machine.

Query Processor Communication Cost. First, it is important to analyze how often a query processor could communicate with the distribution manager before it had a significant impact on query performance. This will indicate how often the distribution manager can communicate with each query processor. A limit needs to be observed, so query operators are not reallocated too often, reducing performance. To study this, we loaded a moderately sized query plan (20 operators, 5 joins) onto a single query processor. We then sent connection requests to it at increasing rates, from 0 per second to 1000 per second, to find out how often a connection request can be handled without degrading performance.

Figure 10 shows the throughput of the query plans with various connection request rates. We can observe that the query processor can easily handle 50 connection requests per second. This is an important number, because it indicates how often the distribution manager may communicate with the query processor. We will see in Section 6.3 that a typical query operator takes 6-10 connect requests to properly activate it on a query processor. Thus we can conclude that we can easily move one operator on a query processor per second.

This is a very high rate, in fact, in our experimentation we will only move operators once per minute, to allow for sufficient time for a distribution to be tested. Thus we see that moving an operator in this environment will not be a bottleneck, as long as we do not communicate with a query processor with more than 50 connections messages per second.

CPU Utilization of the New Query Processor. Another performance test we run for the new D-CAPE query processor is to monitor how one individual query processor utilizes the CPU versus the original CAPE system. For this experiment, we

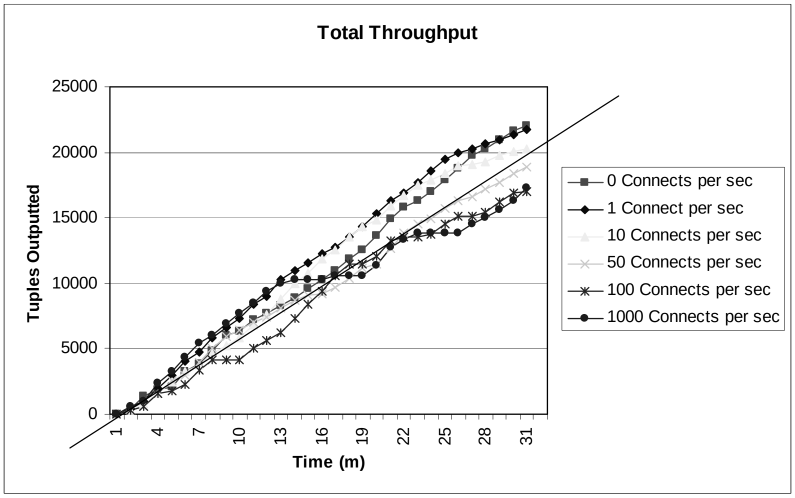
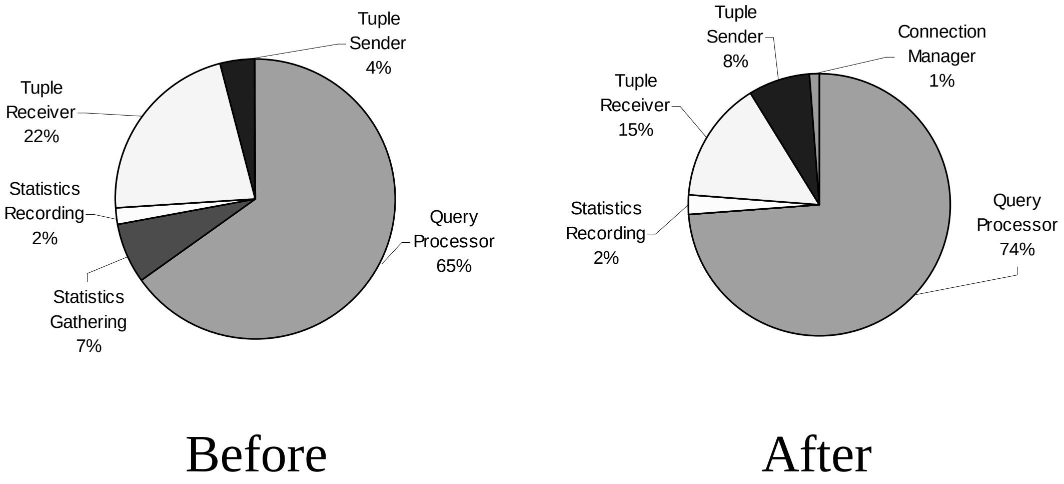


Figure 10. Throughput of Query Processor with Increasing Connection Requests

ran the original CAPE DSMS and the new D-CAPE DSMS on a query plan with 40 operators. This size query plan sends large amounts of data over the network and really tests both the processing of data as well as the way tuples are sent and received. In the D-CAPE DSMS, we ran the query plan utilizing only one query processor so that it could be more fairly compared to the original CAPE DSMS.



Before

After

Figure 11. CPU Utilization Before and after Query Processor optimization.

The primary objective for this experiment was to test if the new query processor utilizes the CPU more efficiently than the original CAPE design. Figure 11 shows the differences in CPU usage between the two query processors. Query execution gets a larger “slice” of the CPU in the new version of CAPE, which is important. Maximizing the amount of time processing the data will provide better performance, rather than spending CPU time performing support operations. Using the hash based functions for the sender and receiver and by having each thread communicate with each other, we see improved CPU utilization over the original CAPE implementation.

Input/Output Connections of a Query Processor. A query processor has two main goals, to process incoming data as fast as possible and to send that data to the next query processor as fast as possible. In order to maximize the performance of these two tasks, it is important to find out what limitation there is (if any) on the number of input and output connections a query processor could handle. We ran experiments with an extremely small query plan (a single operator), as this would provide the best scenario in terms of the number of connections that this one query processor would be able to handle concurrently. We then replicated that query plan several times on the same query processor to increase the total number of connections on the machine.

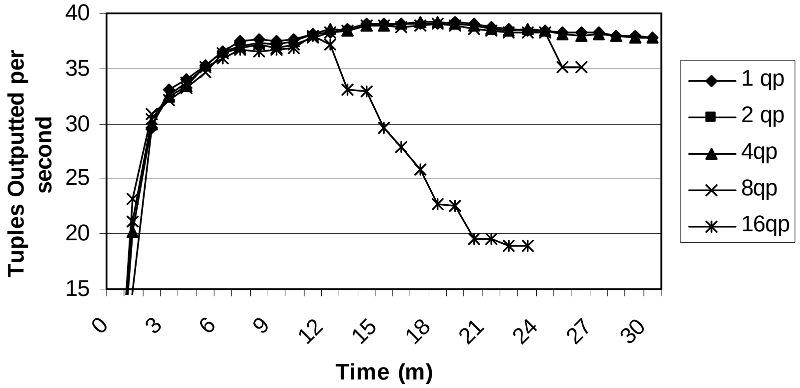


Figure 12. Output Rate of Query Processor with Increasing Connection Requests

Figure 12 illustrates how increasing the number of connections to and from a machine causes a decreased output rate. We find that the cause of this is that the query processor is spending too much time sending and receiving tuples, and not enough time processing them. Figure 13 shows how adding connections decreases the percentage of the CPU that the engine can devote to the actual data processing versus the input/output connection handling. Using these experiments, our machines can safely handle 8-12 input/output connections without significantly degrading performance. This will have to be taken into consideration when we consider *how* to distribute a query plan.

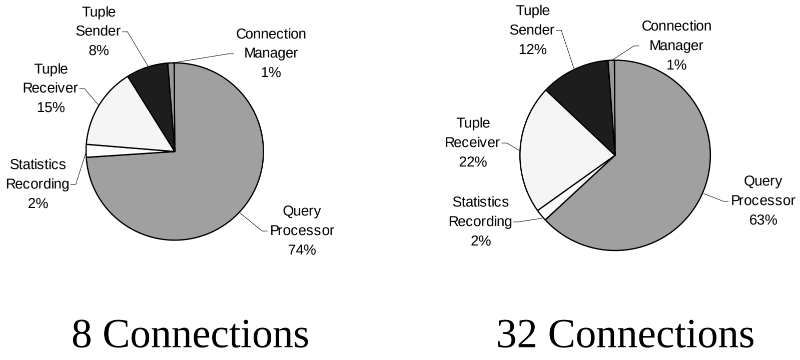


Figure 13. CPU Utilization with a Varying Number of Network Connections

4.5 Distribution Manager

The basic job of the Distribution Manager is to synchronize the management of the query plans and data, and to then respond to situations where a query processor is under heavy load. The Distribution Manager can be thought of as the "brain" of query execution. The Distribution Manager knows about all queries in the system and all available query processors. It is then responsible for *distributing* these operators among the available query processors and for telling each query processor how to work together to process each query plan. It is also responsible for receiving statistical data from each processor to determine the *workload* (how "full" the processor is), and determining *if* operator reallocation is necessary, and then deciding on *how* to reallocate the query operators to improve overall performance.

Figure 14 illustrates the architecture of the Distribution Manager. It is made up of four primary components and three repositories. The four main components are the *Runtime Monitor*, *Connection Manager*, *Query Plan Manager*, and the *Distribution Decision Maker*. Each of these components interact in the following way:

Runtime Monitor. This is the monitor that listens for statistical updates from each query processor. These updates are statistics that are already collected in the query processor, such as the number of tuples in memory or the average output rate. It receives this information, places it into a *Statistics Table* and then gives it to the Distribution Decision Maker.

Connection Manager. The connection manager is responsible for taking the decided distribution, and physically sending a sequence of appropriate connection messages using our redistribution algorithm to establish the distributed plan on its respective query processor. Each of these connection messages derives from a Connection class, guaranteeing a certain packet size and a consistent interface for the query processors and distribution manager to follow. The connection manager typically only sends messages to a query processor, but it can also communicate with the end-user application or the data stream source as well.

Query Plan Manager. The query plan manager is responsible for managing the query plans in the system, and also determining if the query plan distribution is valid. Validity means that all query nodes are represented exactly once on the cluster of query processors, and all of the query processors are up and running.

Distribution Decision Maker. The decision maker is responsible for deciding *how* to distribute the query plans. There are two phases to this decision. First, an initial distribution is created at startup (section 5). Second, it reallocates query operators to other query processors depending on how well the query processors are perceived to be performing (section 6).

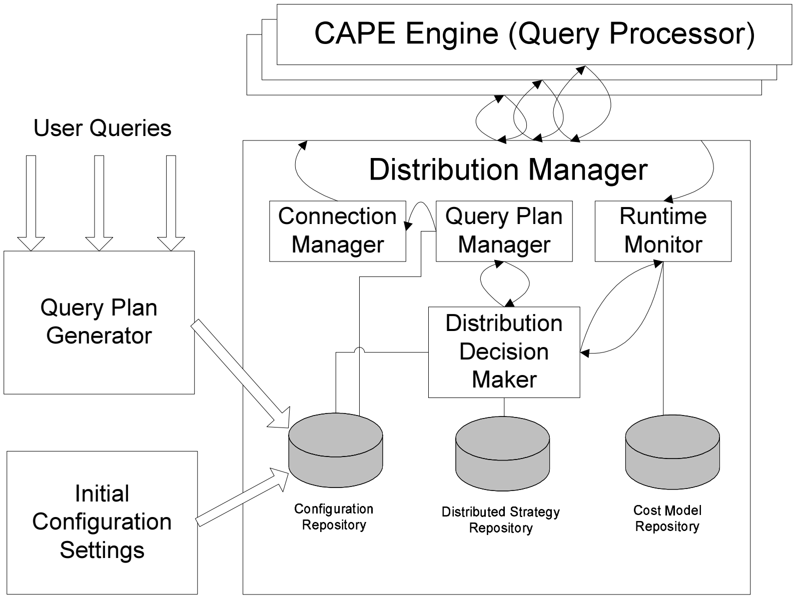


Figure 14. Distribution Manager Architecture

Algorithm 1 describes how the Distribution Manager operates upon initialization. In the following sections we will discuss this algorithm in more depth, including how we distribute, calculate cost and redistribute query processors.

Before processing any data, the Distribution Manager is responsible for configuring each one of the query processors by giving it all query plans that it could potentially process, and other data, such as where streams are located, and statistics that need to be collected. This is done upon initialization to minimize communication cost and only incremental communication messages are sent at run-time such as new query processors or query plans. By doing this before execution, we limit the amount of communications that will need to occur during execution. The *Distribution Decision Maker* then creates an initial distribution of the query operators and then uses the *Connection Manager* to take care of physically *activating* the distribution

Algorithm 1 Distribution Algorithm at a glance.

```
Retrieve configuration information from user.
Retrieve query plans from the query plan generator.
while are more query processors do
    Send all configuration information to query processor
    if Query Processor does not respond then
        Remove query processor from list of active machines.
    end if
end while
if No query processors available then
    EXIT
end if
Load the distribution pattern from Strategy Repository.
Distribute the queries using the pattern.
Load the cost model.
Load the redistribution policy.
Send the statistics to monitor to each query processor.
while still processing do
    Retrieve statistical updates from query processors
    Calculate the workload on each machine
    Redistribute the operators using Algorithm 5.
end while
```

on the remote query processors. A distribution is activated when the query processor that is to run the operator is connected to all data streams and is prepared to process the data. This is discussed in detail in Section 5.1.

During execution, each query processor reports to the *Runtime Monitor* the current statistics of the machine's local state. The Runtime Monitor can use this information, along with its available cost model, to determine the load on each machine. Note that the system can use any one of its available cost models to determine the workload. The specific model is determined by an administrator during the startup of the distribution manager. The *Distribution Decision Maker* then gets the associated costs for each processor from the Runtime Monitor in the form of a *cost table* and uses the table to redistribute query operators. The type of redistribution policy can be any policy found in the Distributed Strategy Repository. After deciding what operators are to be reallocated, the Distribution Decision Maker can then pass this new distribution plan to the *Query Plan Manager*. It is the Query Plan Manager's job to ensure the validity of this new distribution plan. If it is not valid, the Query Plan Manager informs the Distribution Decision Maker to create a new distribution until it is valid. The Query Plan Manager can then tell the *Connection Manager* to make the proper connections between the streams, machines and end user applications. It is the Connection Manager's job to ensure that no data is lost or corrupted.

4.6 Calibrating Distribution Manager Performance

A challenge in implementing the Distribution Manager was ensuring that it was sufficiently lightweight to not render ineffective when redistributing a query plan. Our goal was to make it lightweight enough to process in real time, but also to have the ability to process complex cost models if necessary. Only incremental changes of the set of query plans are sent to the query processors to reduce the amount of time the Distribution Manager spends communicating with each processor at run-time.

Figures 15 to 17 show the Distribution Manager's resources while running a query plan distributed over five query processors. We can see in Figure 15 that the CPU is rarely used. It is primarily used only when calculating new distributions. In Figure 16 we determine that the network traffic the DM creates is minimal. The DM received only an average of 400 bytes per second, and never sends out more than 1kb in a second. We also use memory very sparingly, using less than 20mb of main memory, as shown in Figure 17. In fact, the main cost that we incur is in the very beginning of execution, where the query plans and configuration are sent to all of the query processors. The DM is able to reduce the amount of resources it needs by only providing incremental changes to each query processor, when necessary.

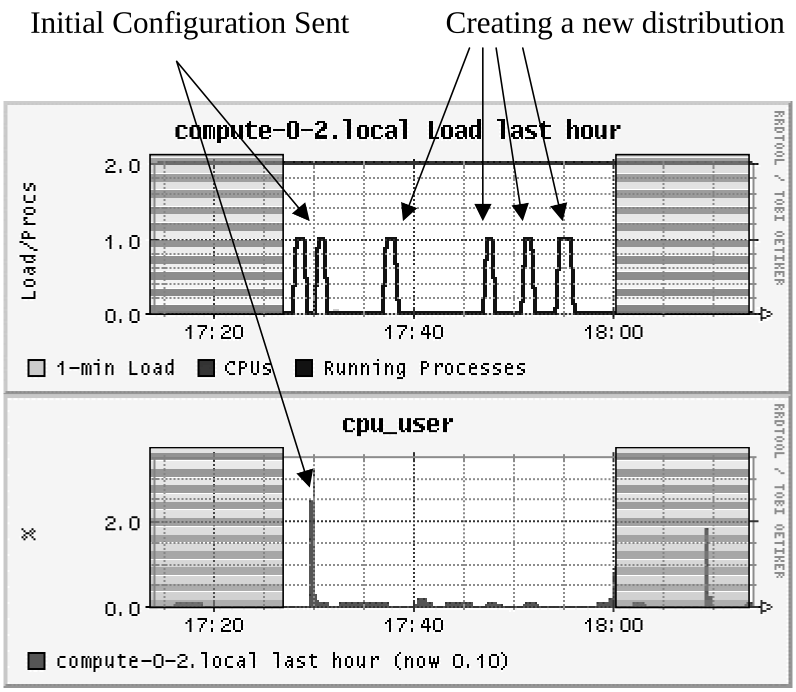


Figure 15. CPU Usage of Distribution Manager

These experimental results indeed now confirm that the overhead of using a single Distribution Manager is minimal. By designing the Distribution Manager carefully, we were able to minimize the system resources used by the DM by limiting the number of communications with the query processors.

5 Query Operator Distribution: Methods, Algorithms and Evaluation

5.1 Initial Distribution

We have found that the initial distribution of a query plan directly influences its performance. Distribution is defined as the physical layout of query operators across a set of query processors. We will later show that we can have performance gains of 100% over a naive distribution algorithm by distributing our query operators using a “connection-aware” approach. We will also show that algorithms that are not carefully designed will not always increase performance beyond that of a single query processor.

The initial distribution depends on the knowledge of two pieces of information: The queries to be processed and the machines that have the potential to do the work. First we will go into detail about query plans and query processors, and how they are composed in our D-CAPE System. We will then discuss how we can take this information to create an initial distribution to begin execution.

5.1.1 Query Processor Description

A query processor is the fundamental component of the DSMS that performs the actual query processing. The query processor can have many properties, and further, each query processors may be heterogeneous. Because of this, we maintain detailed knowledge as shown by Table 5.1.1.

Notice that the processor’s description is simply defined by a Property/Value pair. This way, as new properties of a query processor may arise, we can easily add them to the description without changing our implementation. We can also maintain other properties that change over time, as we will see in Section 6.3. This description is maintained by the Distribution Manager.

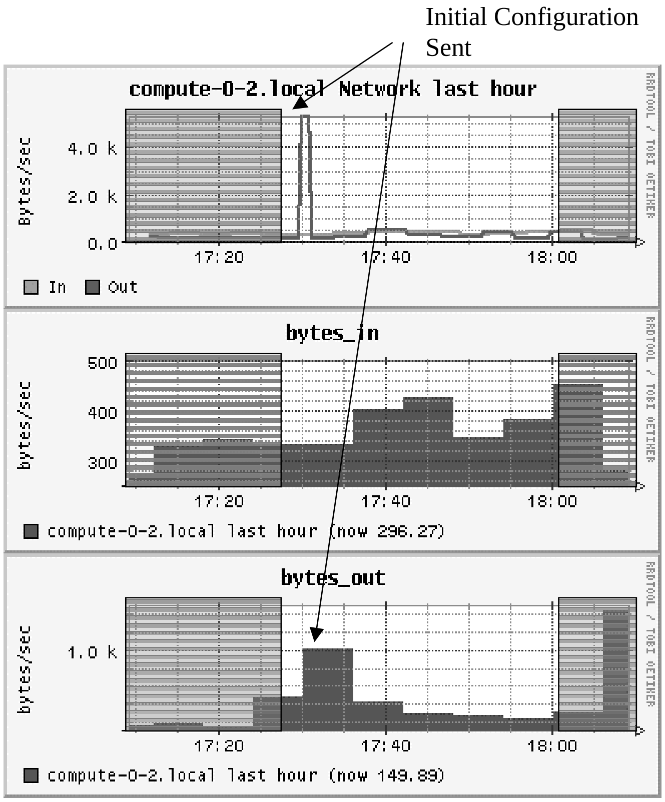

Initial Configuration Sent

Figure 16. Network Usage of Distribution Manager

5.1.2 Query Plan Description

As discussed in section 3, a query plan is a directed acyclic graph with the query operators as the nodes and the queues of data as the edges. The query plan description holds both initial properties of the query plan and also properties that change during execution. D-CAPE represents this query plan as a set of query operators, each containing many properties. These query operators are then connected together using a parent/child relationship, which in turn represents the query plan.

Because of our internal query plan structure, it is easy to capture any property about a query operator or query plan. This is an integral feature of the D-CAPE system; the ability to add properties for future research without modifying the existing data structure. Here are some examples of properties that we may want to capture about a query object.

- Query Plan: Overall priority relative to other operators, number of operators, depth, number of inputs, number of outputs, etc.

- Query Operator: Operator type (Join, Select, etc), number of children, number of parents, etc.

We can also store other properties of the query plans and query operators, just like we could with the query processors. These properties can be dynamic, capturing properties such as Output Rate, Selectivity, Processing Time, and other easily added properties. By combining the properties of each query operator along with its location in the query plan, given the parent/child relationships, we are able to view the query plan in its entirety. We now show how we may use the knowledge about query plans and query processors to come up with an initial distribution of the query plans.

5.1.3 Calculation of Initial Distribution

With knowledge of the processors, plans, and their meta-information, we can create an initial distribution across the cluster of query processors based on the configuration of the query processors and query plans. We create our distribution using a

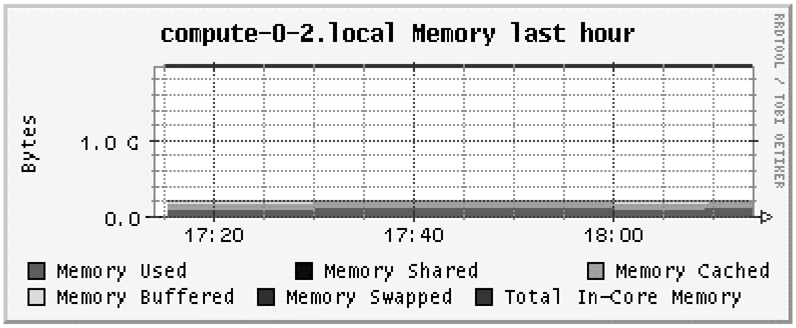


Figure 17. Memory Usage of Distribution Manager

Query Processor Object	
Property	Value
IP Address	davis.wpi.edu
CPU Speed	3.0
Memory	1024MB
OS	Linux
Network Speed	1.5MB/Sec
Location	Worcester, MA
Number of processes	11
Any Property	Any Value

Table 4. Query Processor Object

Distribution Pattern, which is a specific pattern that an algorithm follows to decide how to distribute the query plans. The distribution pattern accepts both the descriptions of the query processors and query plans as inputs and returns a table known as a *Distribution Table* that captures the location of each query plan operator with respect to the query processor that it will be executing on.

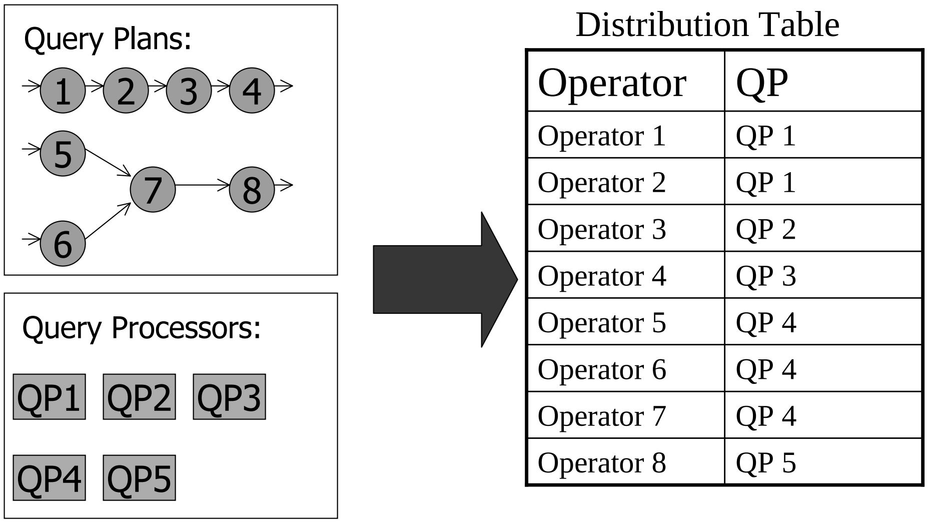
The methodology behind how the table is created depends on the Distribution Pattern. This is important because it allows us the flexibility to implement any Distribution Pattern, and plug it into the system if needed.

Algorithm 2 Round Robin Distribution Pattern.

```
for $qp$ in $queryPlans$ do
    for Operator $o$ in $qp$ do
        Machine $m \leftarrow getMachineWithMinWorkload()$
        $ASSIGN$ $o$ to $m$
    end for
end for
```

In our implementation we introduce two distribution patterns, commonly used in distributed systems in other disciplines [27][32][36]. These algorithms were chosen because of their effectiveness in other disciplines. Round Robin (Algorithm 2) is a common algorithm used in distributed systems such as [23] and Grouping Distribution (Algorithm 3) and various other algorithms are common in distributed database systems such as [29]. Grouping distribution was selected to help reduce the associated network costs [29] of the distribution. We will now define these two algorithms:

- **Round Robin Distribution.** Iteratively take each query operator and place it on the query processor with the fewest number of operators. This will ensure each processor has an equivalent number of operators (i.e, an equal workload). In Algorithm 2 we define workload as the number of operators on the query processor.



Operator	QP
Operator 1	QP 1
Operator 2	QP 1
Operator 3	QP 2
Operator 4	QP 3
Operator 5	QP 4
Operator 6	QP 4
Operator 7	QP 4
Operator 8	QP 5

Figure 18. Distribution Table

- **Grouping Distribution.** Take each query plan and create sub-plans for each query where neighbor operators are grouped together. Then divide these groups among the available query processors. This distribution ensures that few network connections are made, since adjacent operators are for the most part kept on the same processor.

Figure 19 shows how a query plan, in this case Query Plan 2, may be distributed with the Round Robin Pattern. Figure 20 shows how the same plan will be distributed using the Grouping Distribution algorithm. We can see that with this query plan, Grouping Distribution minimizes the number of network connections.

The Round Robin Distribution in contrast distributes in a completely different manner, fragmenting the query plan into 12 pieces, and causing a total of seventeen network connections, nine more connections than the grouping distribution! We also observe that data that flows through a query processor assigned by the Round Robin algorithm may flow back through it for a second (or even third) time for processing. Finally, Round Robin put 3 of the join operators onto one machine! This will create a bottleneck for this query processor due to the expensive join cost.

Our first goal is to create a general framework for managing operator allocation. Second our goal is to then implement a few distribution patterns to compare the trade-off between different properties of distribution. Future work for this project will be in designing novel distribution patterns to maximally boost query performance.

After the distribution table is created, it is then validated by our Query Plan manager for two conditions:

- Every query operator is represented in the table.
- Every machine that is represented responds when asked if it is “still alive”. A processor that is alive is one who has active threads, and is ready to process data.

When a distribution table passes validation, the Connection Manager distributes the query plan among the query processors. The Connection Manager is capable to take any Distribution Table, analyze it and connect the machines accordingly. The Connection Algorithm (Algorithm 4) steps through the process of distributing the query plans according to the distribution table. This algorithm is linear in the number of operators in the table. Once the Connection Algorithm 4 has completed, query execution can begin on the query processors.

5.2 Base Distribution Experiments

Given the different distributions generated by the distribution algorithms, there were four questions that needed to be answered:

Algorithm 3 Grouping Distribution Pattern.

```
$numMachines \leftarrow machines.getCount()$
$totalOperators \leftarrow queryplans.getAll().getSize()$
$avgNumOpsPerMachine \leftarrow totalOperators/numMachines$
$count \leftarrow 0$
$UsedOperatorsTable \leftarrow null$
Operator $o \leftarrow null$
Machine $m \leftarrow getNextQueryProcessor()$
**for** $qp$ in $queryPlans$ **do**
    **if** $count < avgNumOpsPerMachine$ **then**
        **if** $o = null$ **then**
            $o \leftarrow qp.getNextLeaf()$
        **else**
            $o \leftarrow o.getNextOperatorInTree()$
        **end if**
        $UsedOperatorTable.add(o)$
        $ASSIGN$ $o$ to $m$
        $count \leftarrow count + 1$
        **while** $o.hasMoreParents()$ **do**
            **if** $count < avgNumOpsPerMachine$ **then**
                Operator $p \leftarrow o.nextParent()$
                $UsedOperatorTable.add(p)$
                $ASSIGN$ $p$ to $m$
                $count \leftarrow count + 1$
            **end if**
            **if** $p.getDescendantCount() + count < avgNumOpsPerMachine$ **then**
                **while** $p.hasMoreDescendants()$ **do**
                    Operator $c \leftarrow o.nextDescendant()$
                    $UsedOperatorTable.add(c)$
                    $ASSIGN$ $c$ to $m$
                    $count \leftarrow count + 1$
                **end while**
            **end if**
        **end while**
    **else**
        $count \leftarrow 0$
        Machine $m \leftarrow getNextQueryProcessor()$
    **end if**
    $o \leftarrow null$
**end for**
```

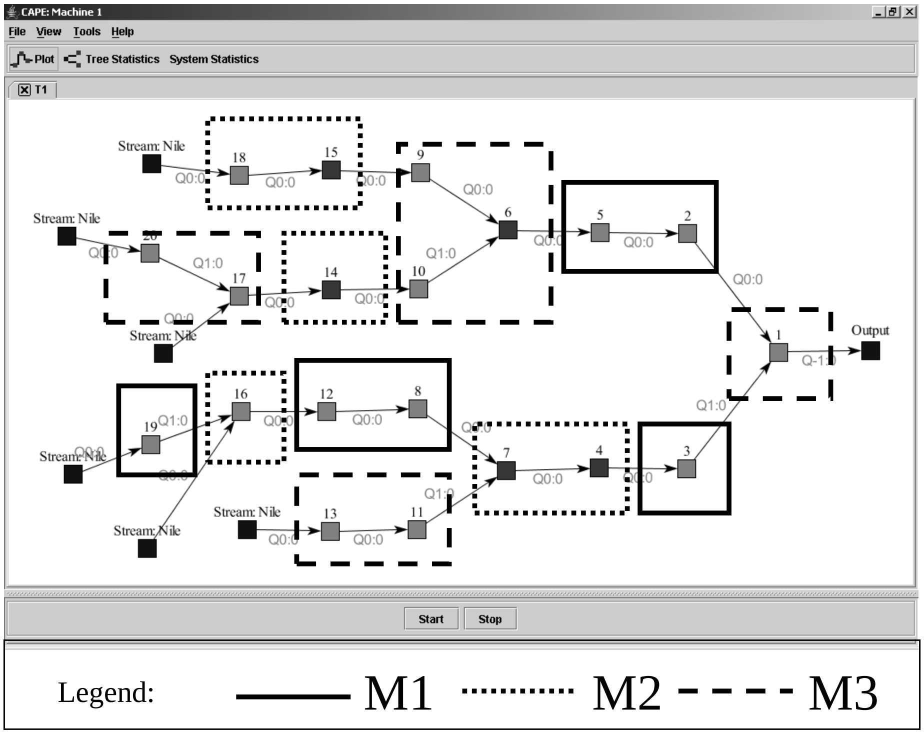


Figure 19. Example of the Round Robin Distribution Pattern using Query Plan 2.

1. What happens when we try to distribute a plan that is small enough to perform well on a single query processor?
2. How much of an improvement can we see over the “traditional” single query processor solution (i.e, a centralized query engine)?
3. Does the type of distribution pattern play a significant part in the performance of the query plan?
4. Based on these experiments, what observable system resources are affected by distributing query plans?

To perform these experiments, we used the same testbed as in Section 4.4 with a variety of query plans, with varying window sizes for 10 seconds to 60 seconds:

- **Query Plan 1:** 5 operators with a depth of 5 and a breadth of 1.
- **Query Plan 2:** 20 operators with a depth of 9 and a breadth of 6.
- **Query Plan 3:** 40 operators with a depth of 14 and a breadth of 8.
- **Query Plan 4:** 80 operators with a depth of 14 and a breadth of 16.

5.2.1 Centralized Processing Versus Distributed Processing

First we want to observe what happens if we distribute a query plan among a cluster of machines, even if the processing could easily be performed on a single machine. Figure 21 shows the throughput of a very small query plan (5 single stream operators) with one single input stream and one output stream for different query distributions from 1 to 5 machines. We can see that even when the query plan is distributed over five machines it still has the same throughput as a centralized (1 machine) processor. At first this may seem surprising because one would assume that the added network cost would slow down the overall query processing and thus throughput, especially as the number of query processors grows larger. We note

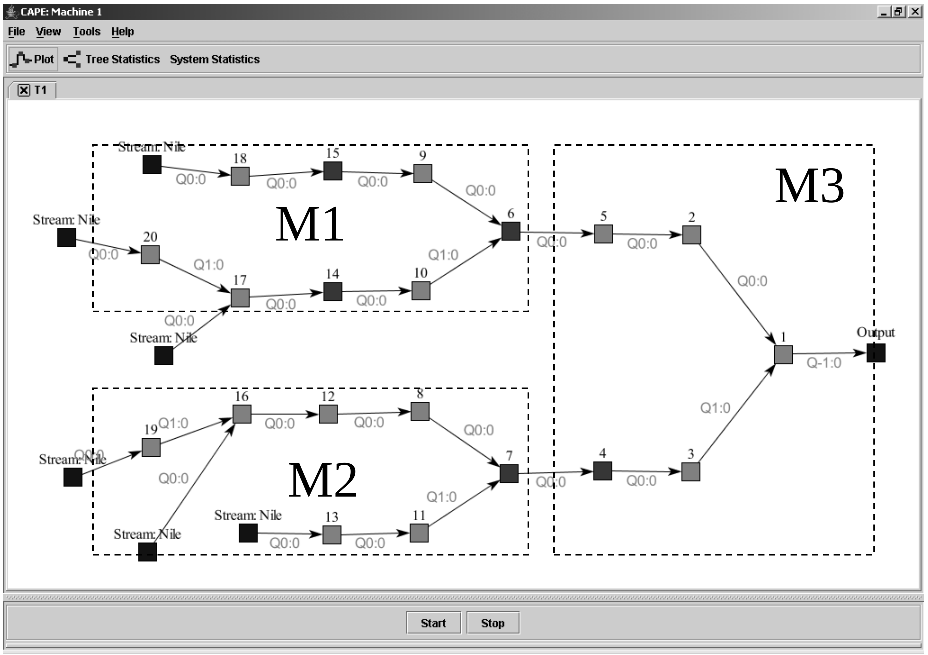


Figure 20. Example of the Grouping Distribution Pattern using Query Plan 2.

that each operator runs in parallel in the distributed processing environment, helping to compensate for the cost of sending tuples across the network.

We observe in Figures 22 and 23 that query plan distribution is just as effective in multi-stream query plans with small windows. We can see that we get similar behavior, that is more query processors will exhibit *at least* the same throughput as a single query processor. This is an important point to make, because we can conclude that it is beneficial to distribute small query plans over the processing cluster. This illustrates that even the smallest plans can be distributed without decreasing performance.

5.2.2 Distribution of Query Plans

In this section we will show experimental studies performed on the query plans described in Section 5.2 to show how a distributed processing environment can improve the performance of the DSMS. We use as a performance measure as the *throughput* of the query plan, or the *total number of tuples outputted over a period of time*. We use this criteria as it indicates how fast we can process the data coming into the system, and producing the result to the user in a more efficient manner. By distributing the query workload over a cluster of machines we are able to improve query execution performance by parallelizing query operators, also giving each operator more time slices to be processing data.

In Figures 24 and 25 we use the Round Robin and Grouping Distribution Patterns, respectively. In both cases, we can see that the total throughput is improved by using multiple query processors. In both cases we can see a 25% performance increase over that of a single query processor. We also observe that the performance increases as we increase the number of machines. This is a logical conclusion since each query operator will have a larger CPU timeslice to run if there are more query processors. This is especially true with operators that tend to take longer amounts of time to process for each incoming tuple. This is especially apparent in a window join operator, where the larger the window and arrival rates of data streams the more the processing time will increase (Equation 3). We find that as more operators (especially join operators) are added to a query plan, the usage of multiple query processors allows for a linear throughput, as shown in Figures 26 and 27.

In Figure 26 we observe the throughput of Query Plan 3. The results are similar to that of the Query Plan 1 (Figure 23),

Algorithm 4 Connection Algorithm.

```
while Operators left in Distribution Table do
    Operator $o \leftarrow DistributionTable.nextOperator()$
    Machine $m \leftarrow DistributionTable.getMachine(o)$
    OperatorArray $parent \leftarrow o.getParents()$
    OperatorArray $children \leftarrow o.getChildren()$
    Send $ACTIVATE$ Connection to $m$ for $o$
    for $p$ in $parentArray$ do
        Machine $m_1 \leftarrow Table.getMachine(p)$
        Send $SENDDATA$ Connection to $m$ to send from $o$ to $m_1$
        Send $RECEIVEDATA$ connection to $m_1$ to receive from $o$
    end for
    for $c$ in $childArray$ do
        Machine $m_2 \leftarrow Table.getMachine(c)$
        Send $SENDDATA$ Connection to $m_2$ to send from $c$ to $m$
        Send $RECEIVEDATA$ connection to $m$ to receive from $c$
    end for
    if $o$ connects to a stream then
        Send $RECEIVEDATA$ connection to $m$ to receive Stream $s$
        Send $STARTSTREAM$ connection to the $Source(s)$ to start sending
    end if
end while
```

except in this case, we can see the single query processor is leveling off in execution, while the multiple processors continue to linearly process data. After 30 minutes we find a 33 percent increase in performance by using five query processors. This is the first example we see where a single processor cannot handle the load of the query plan. It will continue to get worse as we continue to run the query plan over an even longer time period.

Next in Figure 27 we now observe the performance of a large query plan (Query Plan 4). Here we see that the single query processor runs out of memory after executing for 20 minutes. The large amounts of data flowing through the system and the large states of the join operators are filling up memory too quickly in the single CPU system. Figure 28 shows how the single processor memory usage jumps up considerably after approximately 10 minutes of execution as the machine receives larger amounts of data from the children join operators. After a while, the query processor cannot keep up with the large amounts of data flowing into the query processor. Because of the large number of operators per machine, the joins are not getting as much CPU time as they would if there were more machines.

We also note that Query Plan 4 has 16 streams flowing into the processor and 2 streams flowing out for 18 total network connections. As shown in Section 4.4, our query processors can effectively handle 12 connections. The large number of connections into this single query processor would be taking away from the CPU that the query processor can devote to actually query processing tasks. By splitting these 18 connections among more machines, we are able to keep the number of connections per machine small.

5.2.3 Comparison of Distribution Patterns

We observe in this section that different distribution patterns allocate space very differently, sometimes causing many more network connections than others. In this section we compare and contrast the two distribution algorithms in particular: Round Robin (Algorithm 2) and Grouping Distribution (Algorithm 3). We will show in our experimentation that the distribution algorithm that we choose can have a drastic effect on our query processor performance. First, recall that the Round Robin distribution algorithm will distribute query operators in a cyclic fashion, always allocating the next operator to be distributed to the machine with the fewest operators. This balances the total number of operators assigned to each query processor. The Grouping distribution attempts to make large chunks of operators that are adjacent to each other in terms of data flow connections. Then we split those up among the query processors as evenly as possible, such that each processor has a similar number of query operators.

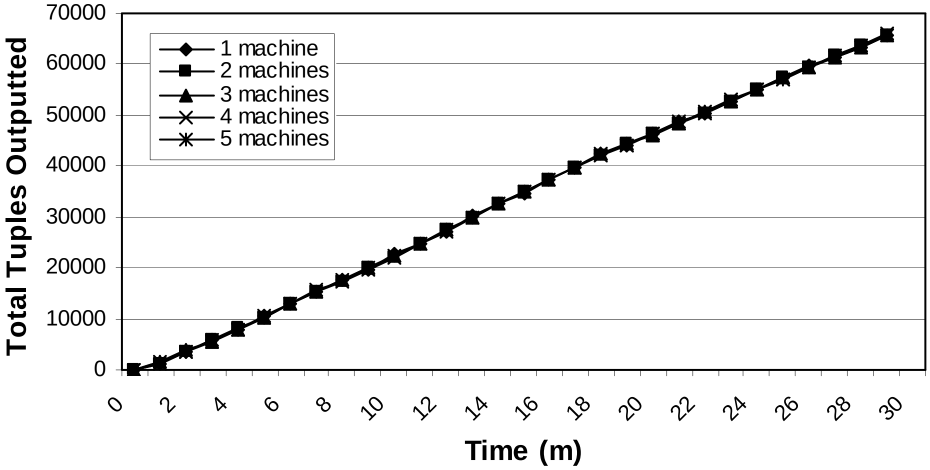


Figure 21. Throughput of Query Plan 1 with a Window of 0 seconds. Round-Robin Distribution.

We will now compare the two algorithms to assess their effectiveness. Our first experiment distributes a query plan of 5 expensive Join Operators (Query Plan 1). The window size for each operator is 60 seconds, and each operator will output twice the data that came in. In Figure 29 we observe that the Round Robin algorithm has a better throughput at first as the query plan begins, but slows down considerably as the query plan executes for longer periods of time. In contrast, we see that the Grouping Distribution algorithm achieves linear throughput, and over time has a better throughput.

Looking at the algorithms and the query plan, this outcome can be explained as follows. The Round Robin algorithm may often allocate two operators onto one machine that are not adjacent. Normally this is not a problem. However in this particular case, the data flows first into the query processor from the stream source and then is output to another machine. That very same data that left the machine is then sent back in at a down stream location of this query plan to operate on the top (root) join operator. This is bad for two reasons: First, the most expensive join operator is the one at the top of the query plan, as it has to process the largest volume of data. Secondly, we are spending time sending data out of the first machine that will later be processed by that same machine again! This distribution exhibits a slower behavior after some time of execution as more and more data is created by the operators over time. The root operator slows down, thus reducing the speed of throughput over time. The Grouping Distribution alleviates this problem by grouping the children joins onto one machine and thus minimizing the total amount of data sent across the network (and number of connections).

We now examine Query Plan 2 in Figure 31 which has a total of 20 operators, 5 of which are joins. Similar to the last experiment, each join is configured to output twice the data that is input, and all of the single stream operators are configured to output 100% of its input, to maximize the cost of the operator. We now analyze different distributions among 1 to 5 query processors (Figure 31)

The first (and most obvious) observation is that the Grouping algorithm always wins. There are two problems that Round Robin introduces that contribute to this outcome. First, in all 3 cases, 3 of the 5 joins were put on one query processor using the Round Robin Strategy. We observe that the join is far more expensive than the single stream operator (Section 3.2), and ideally should be evenly distributed across all query processors. Our Grouping Distribution helped in this situation, partitioning the join operators to all machines. The second problem that we see is that the Round Robin Distribution creates many more network connections than the Grouping Distribution. As discussed in Section 4.4 a single query processor has a limitation in terms of the number of network connections it can handle simultaneously. Grouping distribution never went above 5 connections per query processor in our examples, where Round Robin went as high as 15 connections per machine!

To further illustrate the limitations of the Distribution Patterns we observed execution with even larger query plans: Query Plans 3 and 4. Figure 32 shows the throughput of each algorithm for Query Plan 3, and Figure 33 shows the throughput of Query Plan 4.

Here we observe that as the number of query operators increases, Round Robin and Grouping Distribution tend to drift apart even further in performance. This is for the same reasons stated for the previous experiments. Round Robin makes no guarantee of what operator appears on a query processor, and also tends to create many network connections because of its

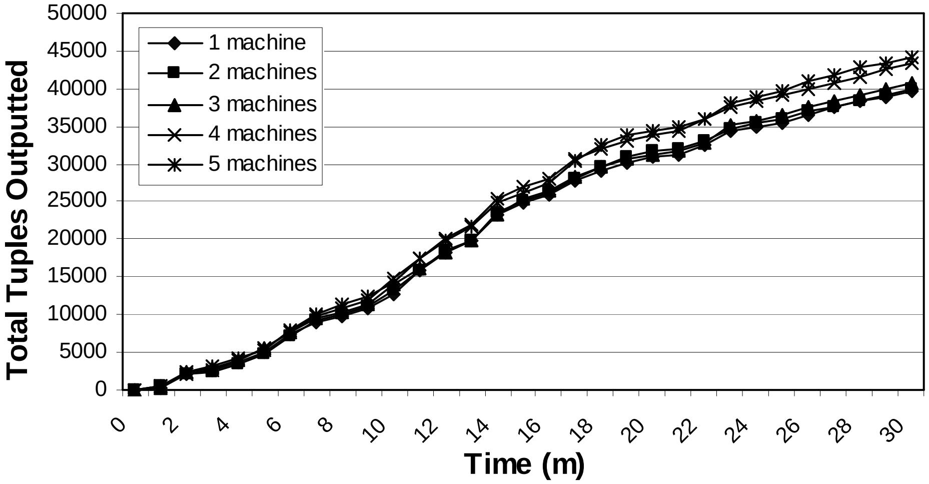


Figure 22. Throughput of Query Plan 1 with a Window of 2 seconds. Round-Robin Distribution.

even-handed nature. We also see a “step” like shape in the graph for Round Robin. Upon looking on the execution, the query processor that outputs the query result to the end user spends much of its CPU time receiving the large amounts of data from the many connections that Round Robin introduces, and then only schedules the root operator periodically, because of the large number of operators on the query processor. In the case of Figure 32 there are 10 operators per machine, 3 of which are a join, leaving little time for the root operator to be scheduled to output result tuples. We thus see a ”step” in throughput, corresponding to each time the root join operator is executed.

We conclude that Grouping Distribution tends to do better than a simple Round Robin algorithm. When we first distribute a query plan, we only know static query plan information: The type of operators, number of machines, number of query plans, window size, etc. We know nothing about the data rate, selectivity of each operator or many other factors that could prove important to execution. Because of this we have to listen for statistical feedback from each query processor to improve upon our initial distribution if need be.

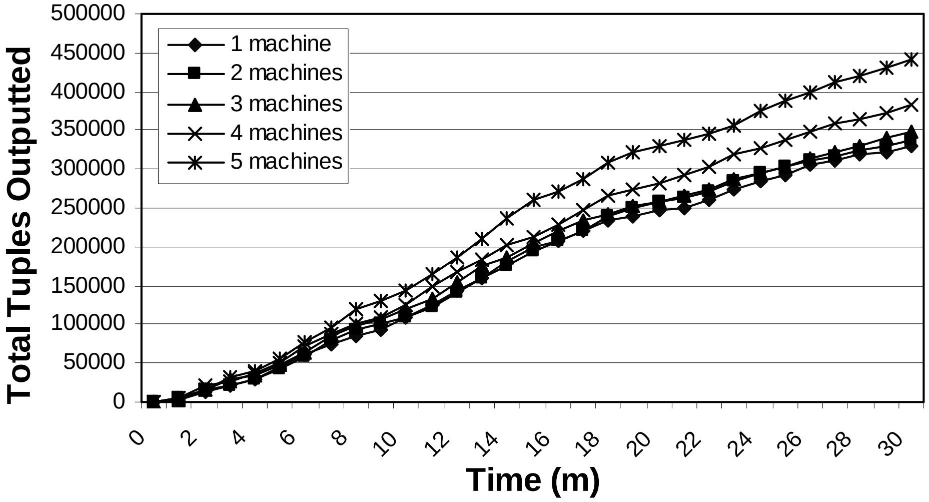


Figure 23. Throughput of Query Plan 1 with a Window of 10 seconds. Round-Robin Distribution.

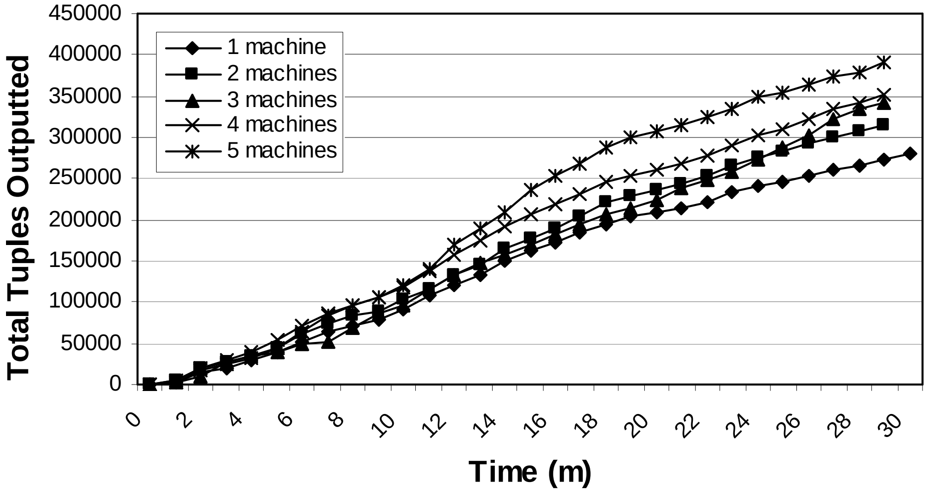


Figure 24. Throughput of Query Plan 2 with a Window of 10 seconds. Round-Robin Distribution.

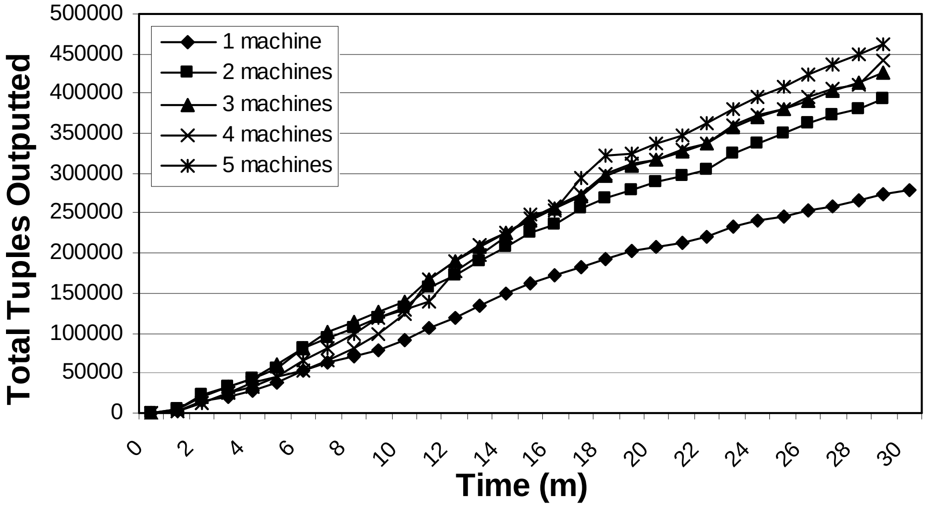


Figure 25. Throughput of Query Plan 2 with a Window of 10 seconds. Grouping Distribution.

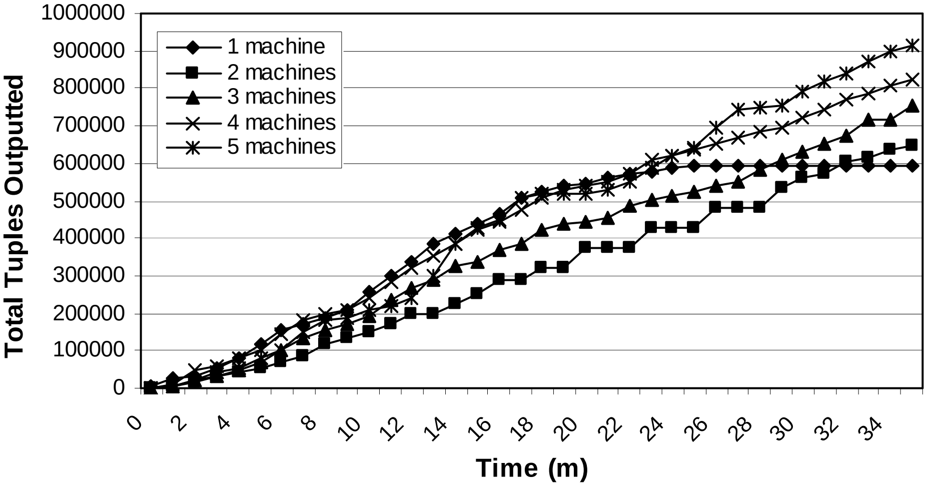


Figure 26. Throughput of Query Plan 3 with a Window of 10 seconds. Grouping Distribution.

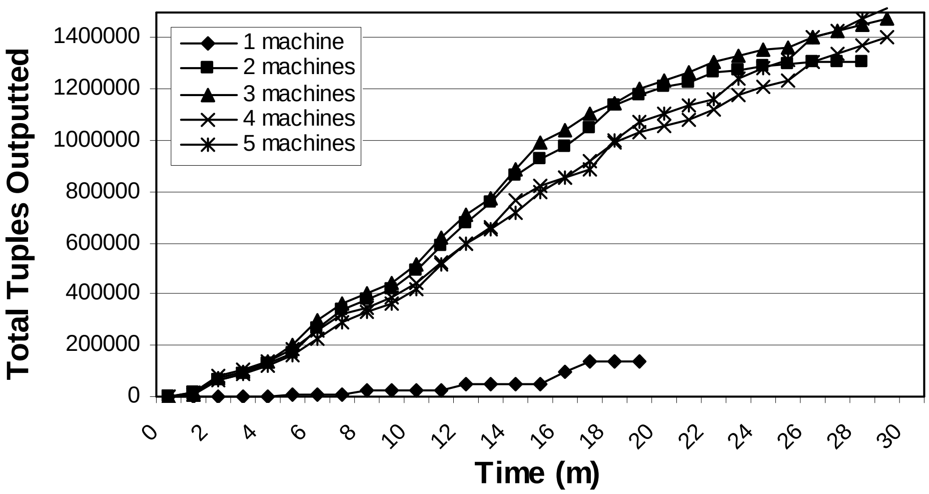


Figure 27. Throughput of Query Plan 4 with a Window of 10 seconds. Grouping Distribution.

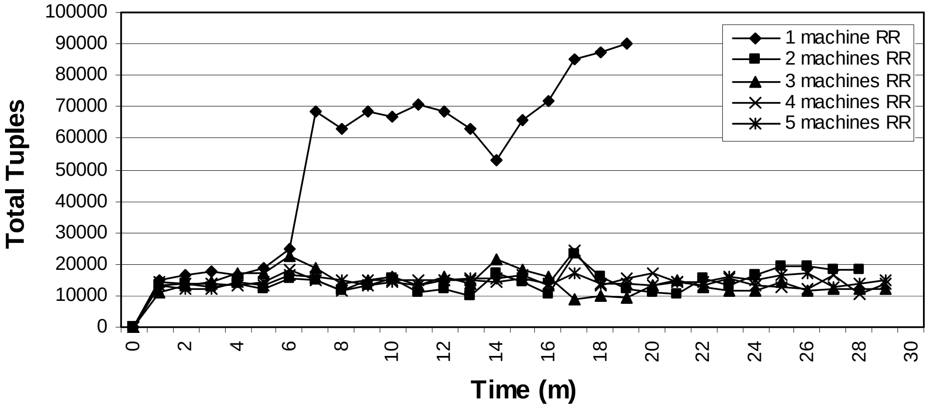


Figure 28. Tuples in memory during execution of Figure 27

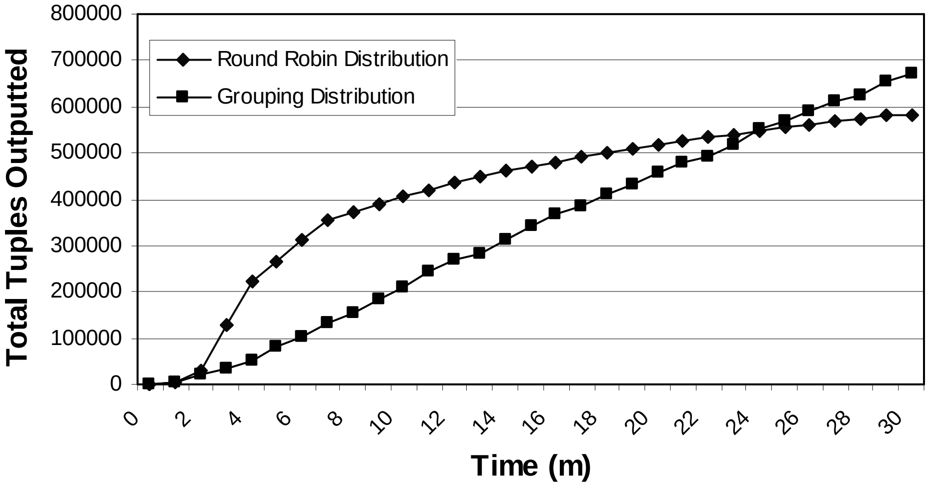


Figure 29. Round Robin vs. Grouping Distribution across 4 query processors.

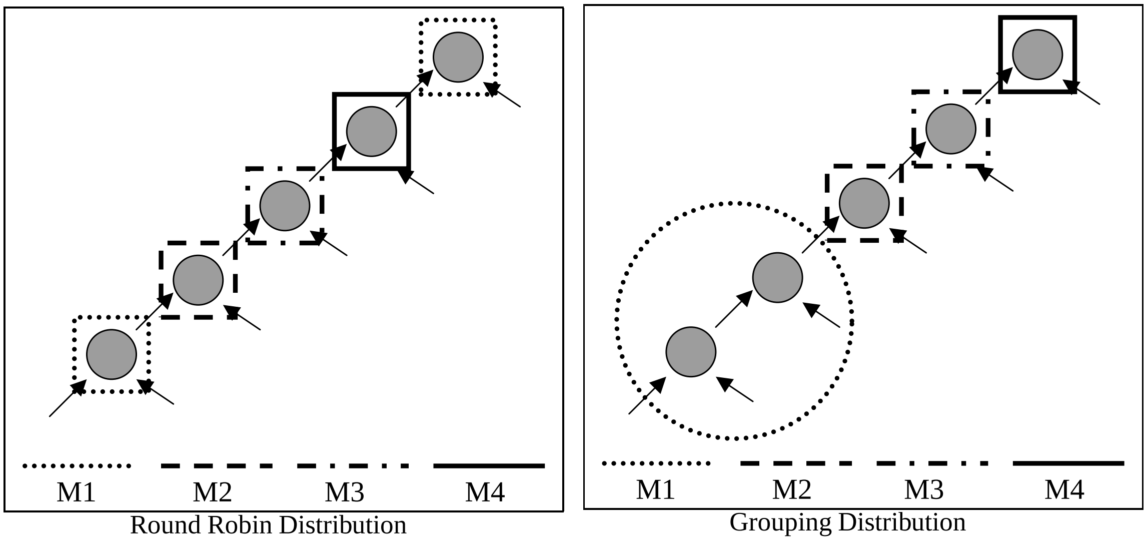
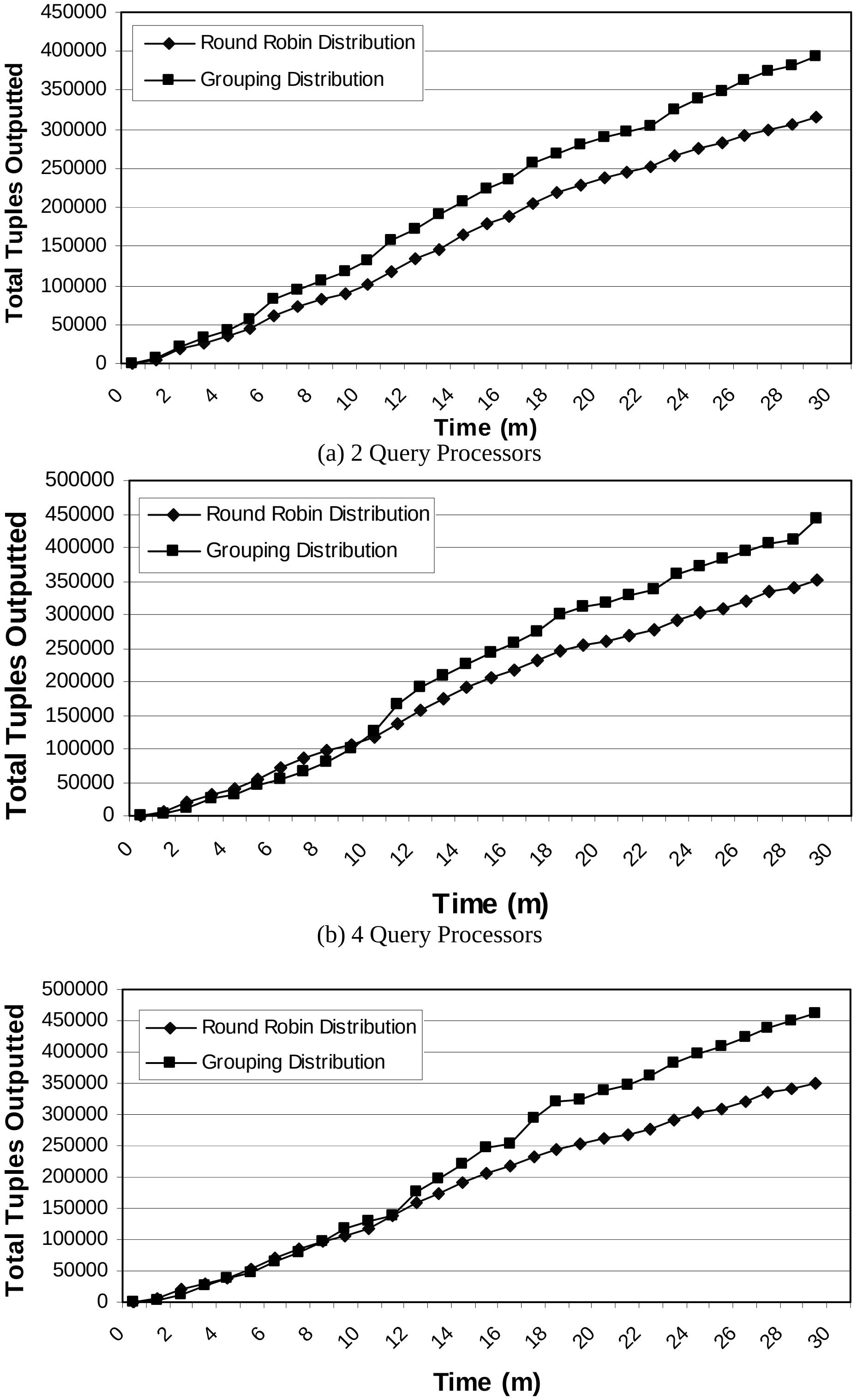


Figure 30. Distributions for Figure 29



(a) 2 Query Processors

(b) 4 Query Processors

(c) 5 Query Processors

Figure 31. Round Robin vs. Grouping Distribution

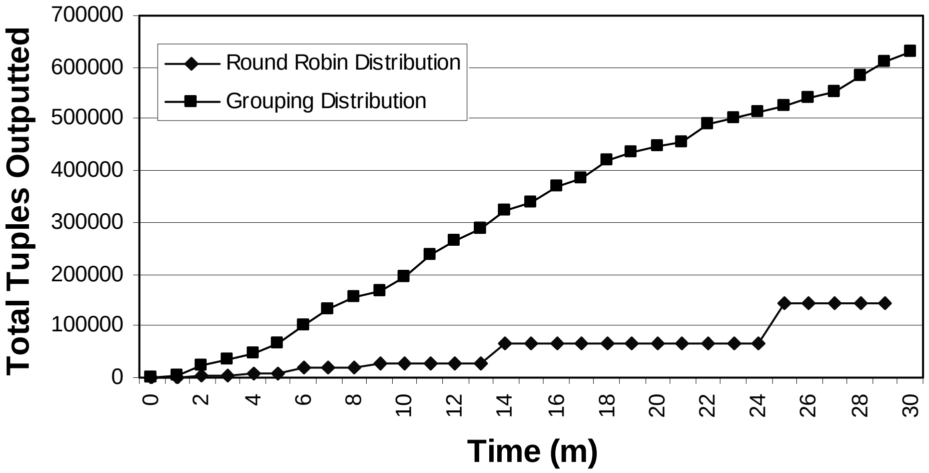


Figure 32. Round Robin vs. Grouping Distribution. Query Plan 3 over 4 Machines.

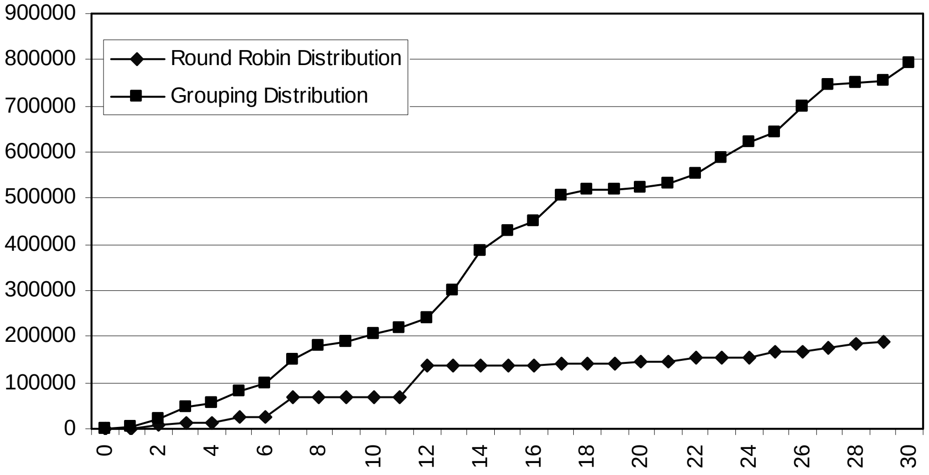


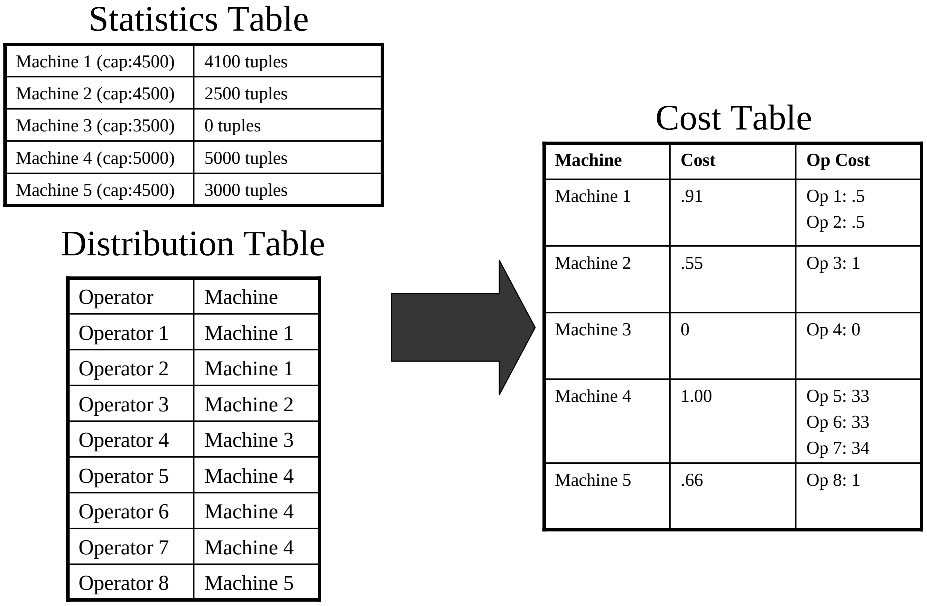
Figure 33. Round Robin vs. Grouping Distribution. Query Plan 4 over 5 Machines.

6 Self-Adaptive Redistribution Strategies: Algorithms & Evaluation

This section discusses and outlines the steps necessary to improve performance by redistributing query operators amongst the cluster of nodes. Static distribution plans can only take into account query and system properties such as shape and size of tree, number of processing nodes, number of input streams, and other data that can be obtained by looking at the layout of the processing cluster and structure of query plan(s). We cannot however count on properties such as state size, selectivity, input data rate, or the expected output rate of the query plan, since this not known until execution. Worse yet is the fact that these runtime properties can change over time depending on many external factors. Even with fluctuating conditions, we can monitor these conditions in D-CAPE to redistribute query operators during runtime. Unlike Aurora* [20], we will allow for redistribution among any of the query processors in our computing cluster. We will also show that the cost of redistribution using our redistribution algorithm is not very costly, even for stateful operators. We will illustrate through experimentation that we are able to monitor the query processing nodes and adaptively redistribute to improve the performance of the query processors over time.

6.1 Cost Measurements

Before deciding how to redistribute operators to improve performance, we first have to define *query processor workload*. That is, we have to define a cost associated with each query processor that tells us how "full" the processor is. There are many potential ways to model cost. Hence, we have built a generic cost model calculator in D-CAPE that allows us to plug in any cost model calculation that we wish. In this section I will first discuss the generic framework for defining a cost model. I will then discuss a specific algorithm that was used for experimentation in this paper.



Statistics Table

Machine 1 (cap:4500)	4100 tuples
Machine 2 (cap:4500)	2500 tuples
Machine 3 (cap:3500)	0 tuples
Machine 4 (cap:5000)	5000 tuples
Machine 5 (cap:4500)	3000 tuples

Distribution Table

Operator	Machine
Operator 1	Machine 1
Operator 2	Machine 1
Operator 3	Machine 2
Operator 4	Machine 3
Operator 5	Machine 4
Operator 6	Machine 4
Operator 7	Machine 4
Operator 8	Machine 5

Cost Table

Machine	**Cost**	**Op Cost**
Machine 1	.91	Op 1: .5 Op 2: .5
Machine 2	.55	Op 3: 1
Machine 3	0	Op 4: 0
Machine 4	1.00	Op 5: 33 Op 6: 33 Op 7: 34
Machine 5	.66	Op 8: 1

Figure 34. Cost Model Creation

In order to determine the cost of each query processor, we need to know three pieces of information. First we must know the current distribution of query operators. The Distribution Table provides this information. We also must know the current statistics on each machine. In D-CAPE, each cost model is based on a set of statistics that each query processor must send back periodically so cost can be calculated. Finally, we must know about every query processor in the cluster so we can retrieve any static properties about the processor that are needed.

Figure 34 illustrates one example of a cost table using a cost model. In this example, we are considering the workload of a query processor to be the percentage of memory filled by tuples. In our implementation, we can use *any* cost model based on statistics collected in the system. This example has been chosen because of its simplicity. In this scenario, we receive the number of tuples each query processor has in its memory at a given time. We can get the capacity of the query processor from its properties. We then create a *cost table* as illustrated on the right side of the arrow that shows each query processor, its related cost, and the cost for every operator running on the processor.

This type of abstraction was chosen for several reasons. First it is easy to break down the workload by operator if necessary. That is, we can determine a fixed cost for every operator on a machine, with respect to how much it is "filling up" a query processor. We also calculate the cost of a machine as a normalized number, typically between zero and one. This is done for generality. In this manner, we are able to give our redistribution policy a table showing costs, but the policy need not know what these costs are. This way we can use any combination of cost models and redistribution policies, as they are orthogonal.

6.2 Experimental Cost Model

In this experimental work, we will measure the cost (workload) of a query processor as the rate at which it is sending tuples across the network. This model was chosen because of the earlier experiments discussed in this paper. By having a large number of input/output connections, performance can degrade significantly. By monitoring how fast a query processor can send tuples across the network, and reallocating operators, we may be able to improve overall performance. By observing the output rate of each query processor, we can move query operators off of the machine so the output rate of the query processor can improve (it will spend more time processing fewer operators on that processor). The algorithm for determining the cost is shown in Equation 4.

$$NetworkOutputRate_i = \sum_{j=1}^{|O|} OutputRate_j \tag{4}$$

For each processor, we determine its total output rate by summing up the output rates of each of the query operators on the processor that produce data to be sent across the network (Equation 4). The output rate for a Query Processor i is the *sum* of the output rate of each Operator j on Query Processor i. The relative cost (in terms of output rate) of each operator k is relative its share of network traffic that it creates versus that of the query processor i it is on (Equation 5). Cost is then input into our redistribution policy to determine how we can re-arrange the query operators to try to increase the output rate.

$$OperatorCost_k = OutputRate_k / NetworkOutputRate_i \tag{5}$$

Based on our empirical evidence in sections 4 and 5 we introduced two examples of cost models based on the number of tuples in memory and the network output rate. Each can be used to determine the workload of a query processor. We will use the network output rate cost model in our Redistribution Policies to observe how query operator reallocation improves performance. There are many alternate cost models that could be created and compared to see which factors most directly influence query processor performance.

6.3 Redistribution Policies

The redistribution policy in D-CAPE is responsible for reallocating query operators across the cluster of query processors based on statistical feedback from each query processor. As discussed in the previous section, this feedback is captured by our Cost Model. Our redistribution policy uses this to determine the re-allocation of operators. In fact, the redistribution is more powerful than the initial distribution by allowing special parameters to be taken into consideration when deciding both *when* and *how* to redistribute. Table 6.3 shows the parameters that our redistribution policies support.

Parameter	**Description**
Cost Table	A table representing the costs of each query processor
Percent Difference	Redistribute if the cost difference exceeds a particular percentage
Eligible Operators	A list of operators that we are allowed to move
Eligible Processors	A list of processors that can get operators to work on
State Size	Operators under a certain size may be moved.

Table 5. Redistribution Parameters.

Besides the absolute cost, we can also specialize redistribution by providing a *Percent Difference* parameter that tells us how far apart the best and worst costs should be before we even consider redistributing. 0% means distribute if there is any cost difference at all and 100% means never distribute. We also pass in a list of *Eligible Operators* that are only considered for reallocation. Along the same lines we can pass in a list of *Eligible Processors* that are available to do work. Finally, we can tell the redistribution policy to only consider operators with a particular state size range, aimed at reducing the time it takes to move an operator. For any particular Redistribution Policy, it may use one or all of these parameters in making its decision. In D-CAPE, we determine these parameters, other than the cost table, by an initial configuration. Using these parameters we are able to use Algorithm 5 to determine how to reallocate the operators (if at all).

Algorithm 5 Redistribution Algorithm

```
$costTable \leftarrow costModel.getTable()$
$maxCost \leftarrow costTable.getMaxCost()$
$minCost \leftarrow costTable.getMinCost()$
**if** $max - min > redistributionPercent$ **then**
    **while** $!valid(newDistribution)$ **do**
        $newDistribution \leftarrow RedistributionPolicy.redistribute()$
    **end while**
    $differenceTable \leftarrow newDistribution - currentTable$
    $connectNewDistribution(differenceTable)$ (Algorithm 4)
    $currentTable \leftarrow newDistribution$
**end if**
```

Here, we use the Redistribution Policy to decide on a new allocation of operators. We then have to reconnect the query processors based on this new distribution. This step is critical, as it has to be fast enough so as not to interrupt query execution, and also correct, in that the data order does not change and no data is lost during the reallocation process. In Figure 35 and Algorithm 6 show how we move a query operator from one query processor to another on the cluster. This is similar to the work in [32], however we have other requirements such as moving the state of the operator and ensuring that the data arrival order is unchanged.

We first find out the new query processor that will be processing the operator, and notify it that it will be doing work on this operator (Step 1). We then create a data flow connection on the new query processor to the query processors that the operator will send its data to (Step 2). We make this connection first so when the operator is activated on the new machine, the data will be able to seamlessly flow from the new machine, causing little to no disruption in data flow. We then create a data flow connection to the output of the children operators to the new machine, so the data will properly flow to the operator on the new processor (Step 3). This also effectively ends any data going into the operator on the old processor, allowing us to terminate its execution (Step 4) after its input queues have "dried up". Since the operator on the old processor may have state information that will be needed, we then send the state from the old processor to the new processor. Execution will then continue as it would if it had never moved (Step 5).

Finally (Step 6), we need to ensure that the data order has not changed. Before allowing the operator to run on the new machine, we will ensure that the data from the old machine is sent to the parent operator and then the connection is terminated [41]. Once all parent connections are terminated on the old machine, we are able to activate the operator on the new machine. As you can see, there is quite a bit of communication involved in moving an operator to ensure correctness. We package this handshake into 6 distinct steps to only communicate between the Distribution Manager and the Query Processors when absolutely needed. This will minimize the cost of moving the operator to the new machine. Our experimental study confirms our hypothesis that the cost of moving a query operator using this algorithm is negligible.

6.3.1 Cost of Redistribution

In our experimental evaluation, we found the cost of moving an operator to be negligible. We ran an experiment using Query Plan 2 with a 20 second window. We moved a stateful window join operator between two machines back and forth, every minute, to see how it would degrade performance (if at all). We were careful to not create more connections when necessary when moving the operator, because this would introduce extra work for the query operators. It was important to isolate the movement so we could measure the cost of moving the data stream connections and sending the state across the network. Figure 36 shows the performance of the query plan when we do not redistribute versus when we move one operator back and

Algorithm 6 Operator Reallocation.

```
for $operator$ in $differenceTable$ do
    $oldProcessor \leftarrow currentTable.getProcessor(operator)$
    $newProcessor \leftarrow newDistribution.getProcessor(operator)$
    tell $newProcessor$ that $operator$ will be activated
    for all $parents$ of $operator$ do
        if $newProcessor! = newDistribution.getProcessor(parent)$ then
            $CONNECT$ $operator$ to $parent$ on $newDistribution.getProcessor(parent)$
        end if
    end for
    for all $children$ of $operator$ do
        if $newProcessor! = newDistribution.getProcessor(children)$ then
            $CONNECT$ $child$ on $newDistribution.getProcessor(children)$ to $operator$
        end if
    end for
    for all $children$ of $operator$ do
        $DISCONNECT$ $child$ on $oldDistribution.getProcessor(children)$
    end for
    $DEACTIVATEoperatoronoldProcessor$
    $SENDSTATEfromoperatoronoldProcessortooperatoronnewProcessor$
    for all $parents$ of $operator$ do
        $DISCONNECT$ $operator$ on $oldProcessor$ to $parent$
    end for
    $ACTIVATE$ $operator$ on $newProcessor$ for processing
end for
```

forth, across machines, every minute.

We see that the cost of moving the operator is negligible, because the throughput of the query plan does not change over the 30 minute runtime. This result was expected because of the way that the operator is moved across machines. Because we create the connections for the data to flow *before* we start sending the data, we are able to “flip a switch” and in the eyes of the query processor, turn off one operator and turn it on another machine. This is especially true for larger query plans such as our 20 operator plan because the probability of the operator being scheduled for execution on its old processor is only 5% (1 in 20). Thus, it is highly unlikely that the query scheduler would even notice that the operator was moved. Instead the scheduler would simply schedule other operators to run.

6.3.2 Redistribution Policies

Now that we have discussed our redistribution algorithm and how we can obtain the information necessary to determine the cost for a query processor, we have to determine how we will interpret these costs to reallocate the query operators. There are many possible ways to decide how to perform reallocation. In our current system, we focus on two of these methods.

Balance. The balance redistribution policy tries to evenly balance the query load across all machines. This strategy is effective when system resources such as memory or CPU usage are at a premium. The policy looks at all query processors in the cost table and aims to balance the table by moving operators from the heaviest loaded processor to the lightest loaded processor. It then continues this process until all machines are as evenly balanced as possible. This policy however will not take operators away from machines that are only moderately loaded, as it may disrupt a set of operators on the processors that were performing well.

Degradation. The degradation redistribution policy does its best effort to alleviate load on machines that have shown a degradation in cost since the last time operators were allocated on the machine. If the cost has degraded beyond a certain percentage we attempt to stop the degradation by moving the most costly operators to other query processors, giving highest preference to those operators that will remove a network connection from the overall distribution of operators.

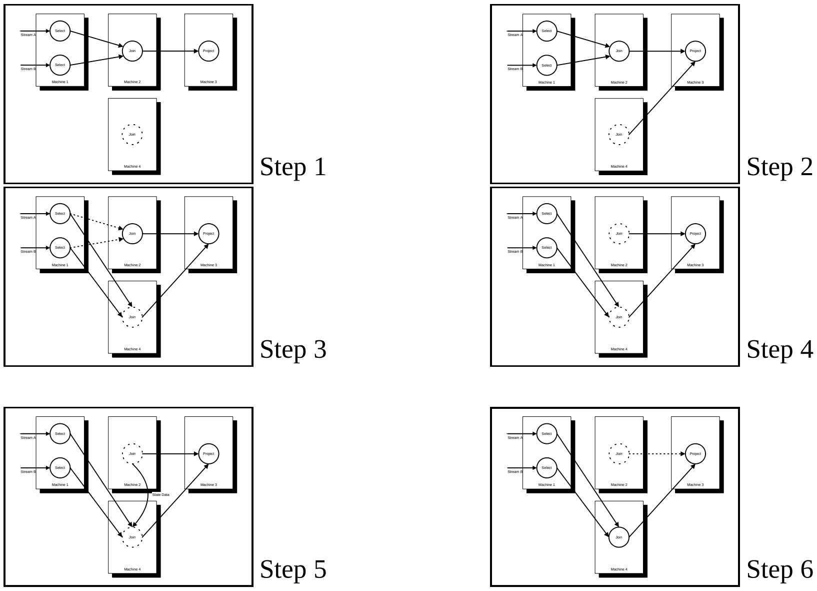


Figure 35. The Six Steps of Redistribution

6.4 Redistribution Experiments

Our goal in redistribution is to *improve* the performance of execution during runtime. Hence, we need to monitor each query processor, and notice when there is an overload and try to correct it. As we saw in section 5, there is no substitution for a good original distribution pattern. However, we can tune the processing if our initial distribution is bad, or turns bad over time. In fact, we can find a speedup of two in some cases, as our experimental studies illustrate.

In this set of experiments, we use the same cluster for our test-bed as in Section 4.4 using our Output Rate cost model explained in Section 6.2 and our Degrading Performance redistribution policy, explained in Section 6.3.2. We use Query Plans 2 and 3 in this section.

In Figure 37 we observe that our redistribution policy is able to improve the performance of the initial Round Robin Distribution by 100%. The redistribution is able to detect the declining output rate for each query processor, and reallocate the operators such that there are fewer network connections per machine. Thus more time to process each operator would be available on the prior overloaded machines rather than spending time sending the data across the network. By observing the output rate, we were able to easily identify bottlenecks in query plan and adjust the output rate before it had degraded too far.

In Figures 38 and 39 we record how redistribution affects a “good” initial distribution pattern such as our Grouping distribution explained in Section 5.1.3. Even though the Grouping Distribution does a great job at grouping operators such that network connections are minimized, we can still see a performance boost of 5 to 10% when moving operators to other query operators by our redistribution policy. In fact during execution with the Grouping Distribution, operators only needed to be moved 4 times in the 30 minute span using our policy, as compared to 17 reallocations for the Round Robin distribution. Regardless, we are able to improve performance by monitoring the performance of each query processor, and then reacting to the costs associate with each processor using our redistribution policy.

Figures 38 and 39 also show us that there is no substitution for a good initial distribution. In both of these experiments we see that the Grouping distribution gives us a big advantage with regard to the throughput regardless of the redistribution policy. However, initial distributions still lack the knowledge of runtime information such as data rates, which can impact the performance as well. Here, a 10% speedup will increase the throughput by almost 800,000 tuples over a 30 minute span, or almost 30,000 tuples per minute.

In this section I have described a framework for reallocating query operators among a cluster of query processors, with the flexibility of adding new reallocation schemes to the system without knowledge of the entire distribution framework. I

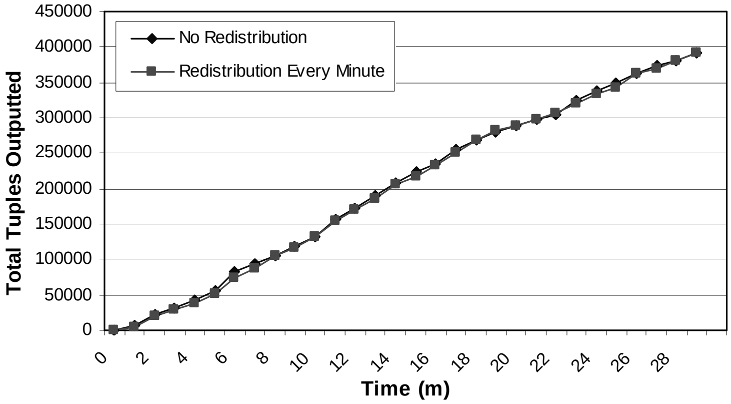


Figure 36. Throughput of Query Plan with Redistribution Every 60 seconds

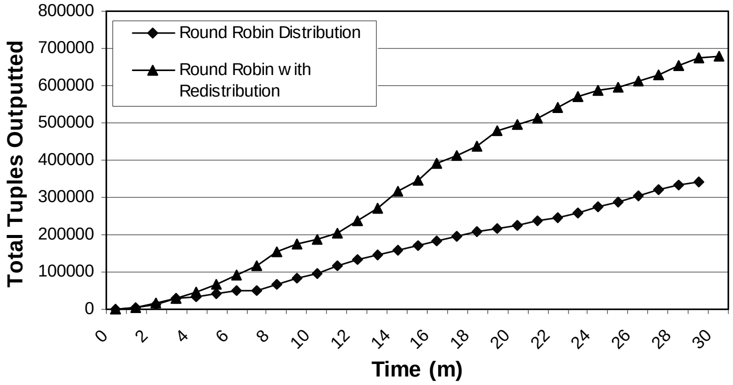


Figure 37. Redistribution of Query Plan 2, with a 10 Second Window over 3 Query Processors.

also show experimentally that reallocation of query operators over a cluster of machines not only increases the performance of the query processors but is also done with little to no overhead, assuming we intelligently move the query operators using our knowledge from section 5, and learned characteristics from a cost model.

7 Conclusions and Future Work

7.1 Conclusions

This paper was able to uncover many of the issues in designing and implementing a D-DSMS. Because of the nature of streaming data and the uniqueness of streaming data operators, new algorithms had to be developed to distribute and reallocate query operators. In addition, we were able to observe the costs associated with query plan distribution and write cost models and redistribution policies that were able to improve query plan performance based on statistical feedback from the cluster of query processors.

This work is a starting point in the area of Distributed Data Stream Management Systems. Because this field is very new, it was first *essential* to come up with an architecture that is both *flexible* and *scalable*. D-CAPE achieves this goal by allowing for individual cost models and distribution algorithms to be “plugged in” to the system, without any special knowledge of the inner workings of the system. By creating a Distribution Manager that was tiered in nature, it allows in the future for clusters of machines to have their own Distribution Manager which is controlled by a higher level manager, thus allowing for a greater number of query plans and the ability to effectively distribute these plans to a cluster of machines that will be

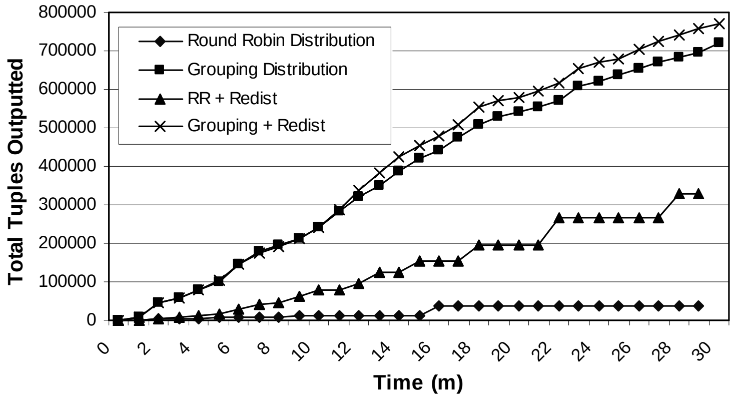


Figure 38. Redistribution with 2 Machines and a 40 Operator Query Plan

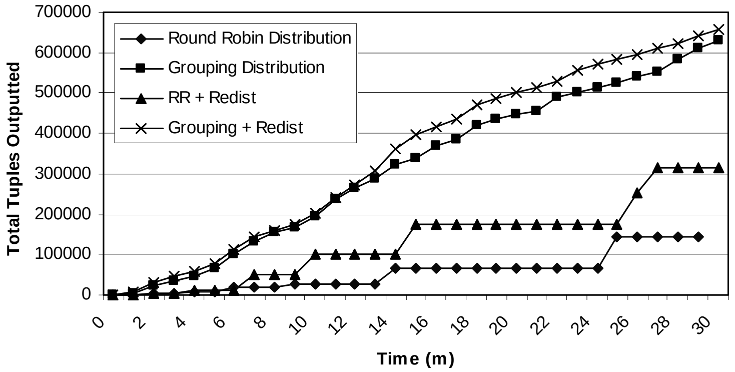


Figure 39. Redistribution with 3 Machines and a 40 Operator Query Plan

effective in processing the query.

Our experimental evidence shows that Query Plan distribution is effective even for small query plans, and becomes very effective as query plans become large. In many cases, we were able to achieve a 100% performance increase over a single query processor, by using a distributed environment. We also find that that the main costs in query plan performance include the number of connections per machine, and the total memory used by the machine. Query processors have better performance when it has to manage fewer connections. We were also able to show that the type of initial distribution algorithm used is essential in how well the query processors will perform overall. Algorithms that tend to create extra network connections, such as Round Robin do not perform as well as algorithms that take network connections into account (Grouping Algorithm).

Redistribution experiments show that we are able to effectively reallocate query operators over time if we observe a degradation in performance at runtime. Reallocating a query operator requires virtually zero overhead as we are able to maintain the flow of data through the cluster using our specialized redistribution protocol.

7.2 Future Work

This paper has opened the door for many potential areas of future work in Distributed Data Stream Management Systems. The flexible architecture of the new D-CAPE system allows for the study of stream processing in many areas.

First this work can be expanded by experimenting with other distribution algorithms and query plans, and studying how they affect overall query plan performance, and what other factors influence performance. Using this knowledge new cost models can be created that can determine the workload of a query processor in different ways using these observed factors.

Redistribution is another key area of future research. There are potentially many other costs associated with query processor performance. Future work can include determining more of these costs and writing redistribution policies that take these costs, or any combination of costs in consideration when deciding how to redistribute a set of query plans.

Research can also be done in using different scheduling algorithms such as Chain [7] or Train [13] scheduling to observe to what degree scheduling algorithms on a single query processor influence performance, and also if particular scheduling algorithms work well with specific query operator distributions.

Finally, work can be done with other data sets of varying volume, such as motion data, traffic data [30] or other forms of streaming data that will be used in future Data Stream Management Systems.

References

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB Journal: Very Large Data Bases*, 12(2):120–139, Aug. 2003.

[2] W. Alexander and G. Copeland. Process and dataflow control in distributed data-intensive systems. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 17(3):90–98, Sept. 1988.

[3] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: Semantic foundations and query execution. Technical Report 2003-67, Department of Computer Science, Stanford University, Oct. 2003.

[4] I. T. Archive. http://www.acm.org/sigcomm/ita/, 2003.

[5] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(2):261–273, 2000.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In ACM, editor, *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems: PODS 2002: Madison, Wisconsin, June 3–5, 2002*, pages 1–16, New York, NY 10036, USA, 2002. ACM Press.

[7] B. Babcock, S. Babu, R. Motwani, and M. Datar. Chain: operator scheduling for memory minimization in data stream systems. In ACM, editor, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003, San Diego, California, June 09–12, 2003*, pages 253–264, New York, NY 10036, USA, 2003. ACM Press.

[8] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *Proceedings of the 13th Annual ACM-SIAM Symposium On Discrete Mathematics (SODA-02)*, pages 633–634, New York, Jan. 6–8 2002. ACM Press.

[9] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In *1st Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[10] P. Bodorik, J. S. Riordon, and C. Jacob. Dynamic distributed query processing techniques. In *Proceedings of the seventeenth annual ACM conference on Computer science : Computing trends in the 1990's*, pages 348–357. ACM Press, 1989.

[11] L. Bouganim, D. Florescu, and P. Valduriez. Dynamic load balancing in hierarchical parallel database systems. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 436–447, 1996.

[12] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - A new class of data management applications. Technical Report CS-02-04, Department of Computer Science, Brown University, Feb. 2002. Fri, 1 Mar 102 16:04:05 GMT.

[13] D. Carney, U. Çetintemel, A. Rasin, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. In J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, editors, *VLDB 2003: Proceedings of 29th International Conference on Very Large Data Bases, September 9–12, 2003, Berlin, Germany*, pages 838–849, Los Altos, CA 94022, USA, 2003. Morgan Kaufmann Publishers.

[14] S. Chandrasekaran. Telegraphcq: Continuous dataflow processing for an uncertain world, 2003.

[15] S. Chandrasekaran and M. J. Franklin. Psoup: a system for streaming queries over streaming data. *VLDB Journal: Very Large Data Bases*, 12(2):140–156, Aug. 2003.

[16] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for Internet databases. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(2):379–391, 2000.

[17] L. Cherkasova. Flex: load balancing and management strategy for scalable web hosting service. In *Computers and Communications, 2000. Proceedings. ISCC 2000. Fifth IEEE Symposium on, Vol., Iss., 2000*, pages 8–13, 2000.

[18] L. Cherkasova and M. Karlsson. Scalable web server cluster design with workload-aware request distribution strategy ward. In *Proceedings of the Third International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS '01)*, page 212. IEEE Computer Society, 2001.

[19] L. Cherkasova and S. Ponnekanti. Optimizing a content–aware load balancing strategy for shared web hosting service. In *MASCOTS*, pages 492–499, 2000.

[20] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR '03)*, 2003.

[21] G. Ciardo, A. Riska, and E. Smirni. EQUILOAD: a load balancing policy for clustered web servers. *Performance Evaluation*, 46(2-3):101–124, 2001.

[22] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[23] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA — A high performance dataflow database machine. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 228–237, 1986.

[24] L. Ding, N. Mehta, E. A. Rundensteiner, and G. T. Heineman. Joining punctuated streams. In *EDBT Conference*, pages 587–604, March 2004.

[25] L. Golab and M. T. Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, September 2003.

[26] J. Hwang, M. Balazinska, A. Rasin, U. Centintemel, M. Stonebraker, and S. Zdonik. A comparison of stream-oriented high availability algorithms. Technical Report CS-03-17, Brown University, Sept. 2003.

[27] D. Jantz, E. A. Unger, R. McBride, and J. Slonim. Query processing in a distributed data base. In *Proceedings of the 1983 ACM SIGSMALL symposium on Personal and small computers*, pages 237–244, 1983.

[28] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *19th International Conference on Data Engineering*, pages 341–353, 2003.

[29] K. A. Lantz, W. I. Nowicki, and M. M. Theimer. Factors affecting the performance of distributed applications. In *Proceedings of the ACM SIGCOMM symposium on Communications architectures and protocols*, pages 116–123. ACM Press, 1984.

[30] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, 2002.

[31] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In ACM, editor, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003, San Diego, California, June 09–12, 2003*, pages 563–574, New York, NY 10036, USA, 2003. ACM Press.

[32] E. Rahm. Dynamic load balancing in parallel database systems. *Lecture Notes in Computer Science*, 1123:37–49, 1996.

[33] E. A. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, and N. Mehta. Cape: Continuous query engine with heterogeneous-grained adaptivity. demonstration paper. 2004.

[34] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: An adaptive partitioning operator for continuous query systems, 2002.

[35] M. A. Shah, J. M. Hellerstein, and E. A. Brewer. Highly-available, fault-tolerant, parallel dataflows. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 827–838, 2004.

[36] A. Silberschatz, P. B. Galvin, and G. Gagne. Operating system concepts, sixth edition. *John Wiley and Sons, Inc*, page 803, 2002.

[37] T. Sutherland, B. Pielech, and E. A. Rundensteiner. Adaptive scheduling framework for a continuous query system. *In Preparation for Submission*, 2003.

[38] C. Tandem Performance Group. A benchmark of non-stop sql on the debit credit transaction. In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 337–341. ACM Press, 1988.

[39] P. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, 2003.

[40] T. Urhan and M. J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In *The VLDB Journal*, pages 501–510, 2001.

[41] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic plan migration for continuous queries over data streams. In *ACM SIGMOD*, June 2004.