Streaming for Dummies

Stan Zdonik Peter Sibley Alexander Rasin Victoria Sweetser
Philip Montgomery Jenine Turner John Wicks Alexander Zgolinski
Derek Snyder Mark Humphrey Charles Williamson

May 19, 2004

1 Introduction

Despite over thirty years of continuing development and a rising prominence in the public eye, today's databases continue to be outpaced by the human appetite for new ways to process information. Although databases continue to serve as a reliable platform for transaction-based operations, traditional relational database management systems have come up short against a new class of data management problems. This paper provides an overview of these problems, examines why traditional relational systems are inadequate to deal with them, and identifies a new class of data processing currently known as *Data Stream Management Systems* (DSMS) designed to handle them. We will provide an overview of the current status of the field, as well as focus on some prototypical systems.

1.1 Motivating Examples

There are several commercial examples that cause difficulties in traditional database systems. Network security monitoring, intrusion detection, traffic monitoring, stock tickers and assembly lines are all sources of streaming data.

Commercial LANs and WANs are essential to a business' operation, so intrusion detection in network monitoring has become a critical application. A single compromised node could corrupt the entire system, and while there are systems to collect information about network traffic, there is no efficient way to process the information. Scanning network logs would be an onerous and useless task, as would loading those logs into a traditional database.

Many United States and European highways suffer from congestion. Urban planners are seeking to efficiently route traffic. One solution that has gained popularity is to use a variable rate toll based on the current volume of traffic. Implementing such a system requires continuous feedback about the state of traffic on particular road segments.

1.2 Running Examples

Examples in streaming data range in their needs and implementations. We will consider two main examples as common threads throughout this tutorial. They illustrate different challenges faced by DSMS researchers.

1.2.1 Stock Market

Monitoring stocks is an example of a class of problem where the data *can not be dropped.* Data arrival can vary from a steady stream to a large influx of data, as occurs when the Tokyo stock market closes. Tuples can come from the stock ticker in the form `<time, symbol, price, exchange>` or they can be of trades of the form `<time, symbol, quantity, buy/sell>`.

In addition to the streaming data, a system must be able to interface with persistent storage that holds the status of the market. Queries can be made on the database itself, or from the streaming trade information, e.g. get me all the sells of IBM in the last 10 minutes. In Section 2.3, we will discuss precisely what such a query might mean and see how to express this query more formally.

1.3 Military Sensors

The second example we consider is one with more technological bottlenecks. Consider a scenario where soldiers are each equipped with a number of wireless sensors monitoring heart rate, temperature, respiratory rate, pedometry, and location. These sensors transmit to a hub, which speaks to intermediate stations on vehicles, which each transmit to a medic on call. Bandwidth is a serious consideration since the vehicle-to-medic rate of transfer is high, the hub-to-vehicle rate moderate, but the sensors-to-hub rate low. Battery power is another issue, considering that sensors used naively can run out of power in just a few hours.

In the military scenario, individual sensors have readings consisting of tuples of the form `<sensor type, sensor id, time, value>`. Each sensor transmits a summary and time interval to the hub, and the hub sends these summaries along with the solider id to the vehicle. A medic's query in this scenario might be to get the location of soldiers in trouble.

1.4 Challenges

We see a few common challenges in the examples discussed above. Our data sources are unbounded, and do not have an obvious division into blocks. Our database must gracefully handle unreliable data sources. The database must also cope with transient spikes of activity. In the examples above, there tends to be a time-critical nature to the queries, so system latency must also be managed.

Unbounded Input with out an obvious partition The streams of tuples are unbounded in the above applications. For instance, a soldier's pulse monitor should return data as long as the soldier has a pulse. Furthermore, there is no obvious division of the pulse monitor data stream. Should we divide it into 8 hour chunks or fifteen minute chunks? This unboundedness is problematic because the relational algebra that traditional DBMSs use assumes finite sets of tuples. For example, given two unbounded sequences of tuples, it makes no sense to do a join because the join can never be completed.

Uncontrolled input rates and unreliable data sources Periods of high traffic are critical times when we want to use our database. However during these times, tuples from data sources will be late or may be dropped. In sensor networks such as military remote triage applications, a sensor's batteries might fail, in which case the sensor would cease to send tuples. We need the database to deal with such cases gracefully, and not report that the soldier has died because his pulse-rate sensor failed.

In financial applications there are extreme surges in activity, and hence data, when certain events happen. When the NYSE opens in the morning there is always a high volume of trading for the first few minutes. Also, if policy makers e.g. Alan Greenspan, announce cuts or increases in the interest rate, trading activity will spike immediately. When the Tokyo stock exchange closes for the day, the exchange makes available a history of all the trades executed that day, which also causes a surge of trades based on information in that trade history.

In general, the database cannot make assumptions regarding regularity of the data sources. This in turn makes defining the behavior of our database difficult. We have to choose a tradeoff between exact *blocking* operations, which would stall while waiting for tuples that arrive late, and approximate *non-blocking* operations.

Latency A frequent assumption in these monitor-oriented applications is that tuples that report about the current status are the most valuable, while older tuples have little value. Thus latency of our answer stream from a continuous query is something for which we want to optimize, although other application-specific metrics of Quality of Service (*QoS*) are useful. Suppose an application is monitoring stock ticker feeds and monitoring trades for compliance. That is, a trade must respect certain constraints on the contents on a portfolio, e.g. the percent of holdings tied up in options. It is in an investment firm's best interest to have this application operate with very low latency since the quicker the trade is executed, the more likely it will be to have the intended effect.

1.5 Conventional Database Systems

Conventional databases are quite good at certain applications, but the have been designed around fixed, semi-permanent data. Applications such as banking, business ledgers, and traditional inventory management are some of the more common uses of a traditional database, where data mirrors real world items, and changes its state relatively infrequently when compared with the processing speed of a computer. In recent years, this distinction has blurred as databases deal with *continuous* queries and modifications rather than *frequent* queries and modifications.

In the emerging class of problems in data management, streaming data is nothing like the data that is being used in traditional databases. It is infinite, chaotic, of varying size and reliability. Traditional databases do not deal well with this new type of data. Historical and recent research shows that current implementations of traditional relational systems do not scale well with continuously expanding data, continuously expanding queries, or large numbers of triggers active on any one table. The problem is not one of hardware engineering, it is the way the system thinks about data.

1.6 Data Stream Management System Requirements

Any viable DSMS has to deal with unbounded streams and to return meaningful results while only having seen part of the stream. As mentioned above, the volume and unboundedness of the streams limits persistent storage. In addition, many of these monitor-oriented applications do not need information from old tuples. In this case, processing continuous queries and then disposing the source data is a reasonable solution, and with high volumes of data, the only solution.

The DSMS must deal with unreliable data sources due to, for instance, delay, failing sensors, or mis-ordered tuples. The operators need to have some timeout mechanism or be non-blocking in order to perform well. If tuples arrive late, the DSMS must have a policy to deal with them. One strategy, acceptable in some cases, is to drop late tuples.

1.7 Three Major Systems

We consider three main DSMS groups, and over the course of this tutorial we discuss and compare their various features in addressing problems facing DSMSs. The three groups are Aurora from Brown University, STREAM from Stanford and TelegraphCQ from Berkeley,

1.7.1 Aurora

Brown University's Aurora[1] project allows for stream processing. Aurora provides applications with a means of specifying *QoS* functions. Application designers are given a GUI with a "boxes and arrows" representation of the system. Aurora has a number of operators, or boxes, namely filtering, mapping, windowed aggregate, and join. To help with scheduling and ordering constraints, windows in Aurora have a timeout, and there is a slack parameter on each window to allow waiting for tuples. Aurora also allows user-defined aggregates, filters, and mapping operators. Another recent system, Aurora* from the same group, provides a distributed DSMS.

1.7.2 Telegraph

Berkeley's TelegraphCQ[6] consists of "a set of composable data flow modules that produce and consume records in a manner analogous to operators in traditional dbms." There are three types of modules. The first is ingress and caching to interface with external data sources. The second is query processing by routing tuples through query modules which are pipelined, nonblocking versions of standard relational operators. TelegraphCQ uses what is called a State Module, which is temporary storage for tuples. The third is adaptive routing: TelegraphCQ's routing plan is able to "re-optimize" the plan while the query is running. They use Eddies which decide the routing plan on a tuple-by-tuple basis.

1.7.3 STREAM

Stanford's STREAM[9] system uses traditional relational operators in a streaming setting. The system considers a stream to be an unbounded append-only bag of `<tuple, timestamp>` and a relation. STREAM defines an abstract semantics for streams and relations, in which there are three classes of operators: stream-to-relation, relation-to-relation, and stream-to-relation. They use CQL, a continuous query language for streams, which is an extension to the SQL standard.

2 Language and Operators

2.1 Introduction

A fundamental notion in a stream is the time stamped onto each tuple. This ordering of tuples differentiates a stream from a relational set of data. As a result, the notion of a timestamp is

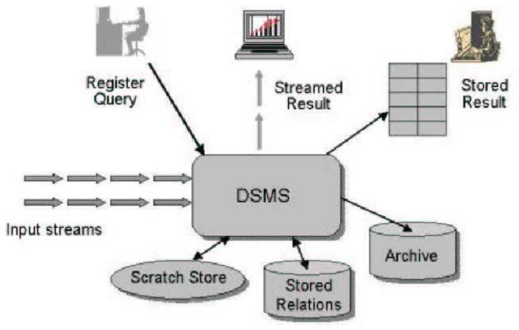


Figure 1: A graphical representation of STREAM's system.

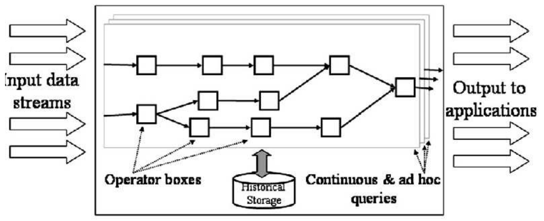


Figure 2: Aurora

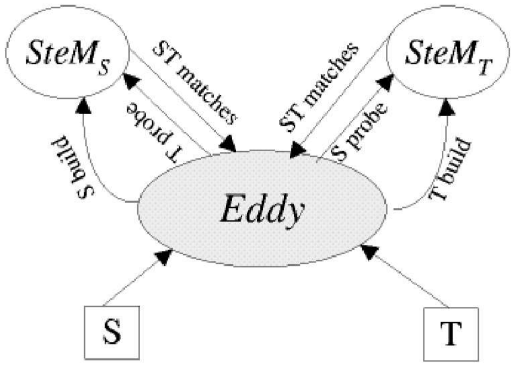


Figure 3: Telegraph

Timestamp Classification		Based on:	
		Clock time	Logical time
Assigned by:	Source	*Explicit*	▨
	DSMS	*Implicit*	*Logical*

Figure 4: Classification of timestamps

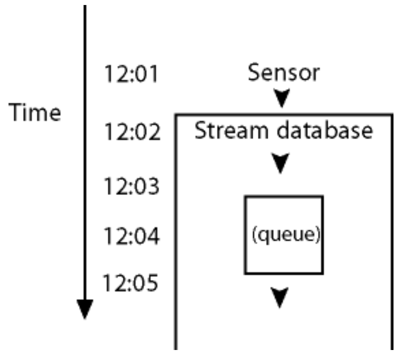


Figure 5: Creation of a tuple and injection into a streaming database

fundamental, and influences the definitions of operators and the languages used to interact with these systems.

For the following discussion, we must define the different types of timestamps to which we will be referring. They may be classified by whether they are assigned by the source or by the DSMS, and whether they are based on clock time or on logical time. We will focus on three of the four cases and refer to them as in Figure 4.

An implicit timestamp is defined as one which has been provided by the system, such as an arrival time. An explicit timestamp is one which is an ordinary attribute of a tuple embedded in the data stream itself, which we assume to be the clock time at the data source (although it need not be). However, notice that since clock time may be measured with a greater granularity than the process which generates the stream, any number of tuples could be assigned the same timestamp. Moreover, transmission delays or lack of clock synchronization imply that, even if it originated from its source(s) as ordered, by the time that it arrives at the DSMS a stream may no longer be ordered by explicit timestamp.

In addition, we assume that tuples are effectively stamped by a "logical" clock (i.e. tuples are assigned consecutive integer values) with distinct value, either on receipt by the system, or possibly by each operator's input queue. That is, tuples are either globally ordered explicitly by assigning a unique sequence number to each tuple, or locally ordered implicitly by the manner in which tuples are placed in various input queues within the system. Notice that in either case, from the user's perspective this ordering is based on a non-deterministic choice.

For example, data generated at a sensor contributing to the stream S may be processed as follows:

1. Data items A and B are generated at 12:00:00 and 12:00:30, respectively.

2. Because the sensor's clock has a resolution of minutes, they are both given the same explicit timestamp of 12:00 and sent as tuples in S.

3. The tuples (A, 12:00) and (B, 12:00) arrive at the database at 12:01:30 and 12:01:00, respectively.

4. Because the system's clock also has a resolution of minutes, they both receive an implicit timestamp of 12:01 as tuples in S. That is, the system logically adds [12:01, (A, 12:00)] and [12:01, (B, 12:00)] to S.

5. Internally, the system represents S as an ordered list of tuples, so that these data items give rise to stream tuples [12:01, (B, 12:00), 1] and [12:01, (A, 12:00), 2] on a queue representing S.

Notice that implicit and logical timestamps are metadata and are not considered part of the data tuples. Also, the logical timestamps may not be stored explicitly with the stream tuples, but may only be implied by their relative position in the queue. As we shall see, one or more of these notions of timestamps may be used when processing a given stream.

2.2 Key Technologies

2.2.1 Stream sources

Since streaming database systems are comprised of streams instead of persistent data stored in tables, the data model is slightly different. Semantically, streams can be thought of as a sequence of timestamped insertions of a tuple. These stream database systems consume streams and typically produce new streams as a result.

Instead of being able to rely on data preexisting in some internal format available upon request, streaming systems have to cope with a more heterogeneous environment. Data streams may be originating from unintelligent sensors which lack the resources to implement a sophisticated protocol, leaving it up to the server to handle the burden of interpreting the data.

For example, in TelegraphCQ, there are no restrictions on data format, as custom wrappers handle the processing of the arriving data. There may be multiple devices pushing in data for the same stream. Also, streams are created and dropped using DDL statements in a similar fashion to how relational databases allow the user to interact with schema definitions. However, one detail is noteworthy: in the declaration there must exist exactly one column that will act as a timestamp for that stream.

Timestamps are a reoccurring issue that different systems have different approaches for handling. When considering a single stream of data, the situation appears unambiguous, however, complications arise when multiple streams are combined. Which timestamp should the resulting stream assume? In combining streams, what should the join do when the tuple with the corresponding timestamp is not immediately available?

Implicit timestamps provide some useful information when exact times are not required. By providing just an ordering, they allow notions of "recent" or "older" to be represented but lack information for correlating data between streams. Explicit timestamps however can be more semantically meaningful because these represent the true time of the data recorded. As a result, these can be used to correlate across different streams in a meaningful way. If one were to use some implicit timestamp for correlation, streams may be inaccurately joined, depending on the delays in receiving the data from the different sources. However, using explicit timestamps introduces complications. When relinquishing control of the timestamp to the data source itself, there is no guarantee that the timestamps will arrive in order.

2.2.2 Ordering requirements

Many operations, such as computing aggregates, require some method of dividing a stream into finite sequences so that the computation can complete in a finite amount of time, and utilize a finite amount of memory. As we shall see in Section 2.3, this is commonly accomplished via "windowing" operators, which require the use of a time-stamp or some other attribute upon which the tuples are ordered. When the data is unordered, the number of tuples that must be stored and the length of time needed to accurately compute the window can grow without bound. This means that we must make some assumptions about the data and/or be willing to tolerate a certain level of approximation in our results.

Two problems in particular are:

1. If ordering on a timestamp, will we use the time at which a tuple was *generated* (i.e., an explicit timestamp), or the time at which it *arrived* (i.e. an implicit timestamp)?
2. How will we react if a tuple arrives out of order, that is, the ordering attribute is not consistent with the logical timestamp?

For example, we might have a query which operates on all tuples between 2:00 and 3:00. If we use the arrival time, this imposes the assumption that all tuples arrived in order, and that they took a negligible amount of time to arrive. As soon as the system clock reaches 3:00, that operation can be performed, safe in the knowledge that no further tuples will be relevant to the computation.

Often, however, we are interested in the time-stamp marked when a tuple was generated. We can still assume that all tuples will arrive in perfect order. As soon as a tuple with a time-stamp after 3:00 is received, the operation can be performed. But under this strict assumption, if a tuple generated at 2:58 does not arrive until 3:05, the only option is to ignore it.

It is instructive to compare how ordering and timestamps are handled differently in TelegraphCQ, Aurora, and STREAM. TelegraphCQ can define a stream in terms of logical or implicit timestamps. It also provides operators for converting a stream from one notion of time to the other. Although this is a limitation, it has the useful consequence that its streams are automatically ordered by timestamp. In contrast, Aurora supports all three types. As we will see in Section 2.3, when using explicit timestamps, it assumes that a stream is *approximately* ordered. STREAM takes a middle road. Although it supports all three types of timestamps, it preprocesses a stream entering the system to insure that it is ordered by timestamp.[1]

2.2.3 K-Constraints

In the previous section, we mentioned that Aurora may assume that stream is "approximately" ordered on explicit timestamp. Specifically, it assumes that there is a parameter k which bounds the maximum number of tuples, or time, between two out-of-order tuples. We say that the stream

[1]However, this may require the system to approximate the stream by dropping tuples.

adheres to an *ordered arrival constraint* with parameter k[4]. More precisely, a stream S is ordered on some attribute $S.A$ with adherence to k if, for any tuples s in stream S, all S tuples that arrive at least $k+1$ tuples after s have an A value $\geq s.A$. That is, any two tuples that arrive out of order are within k tuples of each other. Notice that $k = 0$ implies strict ordering.

Computing a window is just one type of stateful operation which we must constrain and "ordered arrival" is just one of several corresponding parameterized constraints, commonly referred to as *k-constraints*. For example, grouping and join operations lead to similar problems of unbounded space and time requirements. The corresponding constraints are *clustered arrival* and *referential integrity*.

A clustered-arrival constraint on a stream with parameter k specifies the maximum number of tuples with differing attribute values that can arrive between two tuples with the same value of a particular attribute. Specifically, a stream S is clustered on attribute $S.A$ with adherence to k if whenever two tuples in stream S have the same value v for A, then at most k tuples with non-v values for A occur on S between them. In other words, a cluster with value v can be interrupted by at most k non-v tuples.

The last type of k-constraint, referential integrity, bounds the delay between the arrival of a tuple in one stream and the arrival of the unique tuple to which it is to be joined in another stream. Here, the parameter k can either represent the number of tuples or the time between the arrivals of the tuples to be joined. For example, if tuples of `S(TIME, SYMBOL, QUANTITY, BUY/SELL)` are being joined on `SYMBOL` in order to find matching buy and sell orders, one could define a referential integrity constraint of `5` minutes. This would essentially put a `5` minute window on the length of a trade query. This could be desirable in instances where stock prices fluctuate up and down quickly.

2.2.4 Punctuation

Instead of simply making assumptions about the contents of a stream, the technique of *punctuating* a stream assumes that the data source includes additional information in the stream to enable efficient processing. A punctuation is a predicate which tells the system that there will be no more input tuples which match that predicate. For example, a punctuation received with a predicate matching all tuples that have been received for a particular date would indicate to the DSMS that there will be no more tuples for that date.

There are multiple ways to implement punctuation. For example, in the Niagara Query Engine[14], punctuation is represented as data consisting of a series of patterns each corresponding to a tuple attribute. Below are the five patterns defined in Niagara.

NAME	PATTERN	MATCH CONDITION
wildcard	$*$	$true$
constant	c	$i == c$
range	$[c2, c2]$	$i \geq c1 \wedge i \leq c2$
list	$\{c1, c2, ...\}$	$i \in \{c1, c2, ...\}$
empty	$\emptyset$	$false$

To illustrate, consider the stock market example, but represent the stream as XML tuples of the form:[2]

```
<TICKER>
```

[2]This example is similar to one published at http://www.cse.ogi.edu/ ptucker/PStream/pscheme.html.

```
      <TIME></TIME>
      <SYMBOL></SYMBOL>
      <PRICE></PRICE>
      <EXCHANGE></EXCHANGE>
</TICKER>
```

The following XML data would then represent punctuation which matches all quotes with the symbol IBM and the exchange NYSE with a `TIME` between `5/18/2004` at `10:00` and `5/18/2004` at `12:00`:

```
<TICKER>
      <TIME> [ 5/18/2001:10:00, 5/18/2001:12:00 ] </TIME>
      <SYMBOL> IBM </SYMBOL>
      <PRICE> * </PRICE>
      <EXCHANGE> NYSE </EXCHANGE>
</TICKER>
```

When a DSMS system receives this punctuation, it can assume that it will receive no more matching tuples. This would signal it to calculate average prices for that time period and output those results.

In general, the notion of punctuation gives us a very robust language with which to describe a stream. It provides a way for the data source to assert properties of the stream, rather than requiring the system to make assumptions through observations. The disadvantage, of course, is that it relies on external sources to generate the punctuations.

2.3 Windowing Operators

Recall the query from Section 1.2.1: "return all the sells of IBM in the last 10 minutes." Because a stream is infinite, to be computable, we must qualify a query to limit its scope in some way. Such a limitation of a stream to a finite subset is commonly referred to as a "window". The modifier *in the last 10 minutes* serves this purpose. Notice, however, that since this query would be executed *continuously*, the the meaning of this phrase *changes over time*. Thus, we have really specified a *series* of windows, one for each moment in time, t. Applying each window to the stream, $Trades$, would lead to a sequence of tables, W_t, and the results of such a query should reflect the sequence of SQL queries:

```
SELECT *
        FROM $W_t$ as T
        WHERE T.symbol = "IBM" and T.buy/sell = "sell"
```

Conceptually, the sequence, W_t is the result of applying a *windowing operator*, to the stream $Trades$. One may view the sequence W_t as a series of "snapshots" of a finite, time-varying buffer of tuples.

Notice that when discussing window operators we should be clear to distinguish between the window *operator*, any particular window, and the *contents* of the window when applied to a particular stream. In this example, the window operator is *in the last 10 minutes.* This specifies a series of individual windows, such as *from 11:50 to 12:00*. Applying this window to the stream of $Trades$ would give a set of tuples W_{12} corresponding to all the trades between 11:50 and 12:00.

In general, a window operator is specified by three predicates which determine:

- which tuples should be added to the buffer,
- which tuples should be removed from the buffer, and
- at what times, t, a specific "snapshot" W_t is passed on for subsequent processing.

In addition, in order to produce a semantically coherent sequence of tuples as output, an operator should be parameterized in an intuitive manner.

In this section, we will discuss windowing operators in general, as well as some of the specific windowing operators which are commonly implemented. They fall into three general categories: tuple-based, time-based, and value-based windows.

2.3.1 Tuple-based Windows

The simplest case is a window of "the 10 most recent tuples". The window at time t includes all tuples with timestamp $<= t$, and it excludes those with logical timestamp $< s - 9$, where s is the maximum logical timestamp in the set. A snapshot, W_t, of the window's contents is passed on at each clock tick. Note: The choice of timestamp (i.e., logical, implicit, or explicit) is ambiguous; different systems allow for different choices here.

Such a "tuple-based" window is provided by many existing stream-processing systems. For example, STREAM uses the Continuous Query Language (CQL)[2] to express this window operator applied to the stream S by "S[*Rows* 10]". TelegraphCQ would express the same window by:

$$for(t = ST; true; t++)\{WindowIs(S, t-9, t);\}$$

where S is assumed to be defined in terms of logical time[3]. In contrast, Aurora does not have independent window operators; rather they are implicitly specified in the arguments to SQuAL operators[1], which require windowed input, such as Aggregate, Join, and Resample. The "last 10 tuples" window applied to the input stream of an Aggregate would correspond to the arguments: *Order* (*on Tup#*), *Size* 10, *Advance* 1. Notice that Aurora allows the query to access the logical timestamp as the "virtual" attribute *Tup#*.

We should observe that the meanings of these tuple-based windows are slightly different in STREAM, TelegraphCQ, and Aurora. That is because the STREAM version is based on implicit timestamps (i.e., system time), whereas TelegraphCQ and Aurora require that the extent and advance of a window be specified in terms of a single notion of time, in this case logical timestamp. In particular, such a STREAM window cannot be specified in those systems and vice versa.

This difference would be observable in the following scenario. Assume a series of trades of decreasing quantity occur every 0.1 sec. for 1 sec., slowing down to every 0.25 sec. for the following second, yielding the stream in Figure 6, where we have omitted the irrelevant attributes. Assuming an implicit time granularity of seconds, the CQL query:

```
SELECT Max(T.quantity)
FROM TRADES [Rows 2] as T
```

[3]ST refers to the time at which the query was registered.

Input		Output	
Implicit Timestamp	Quantity	STREAM	Aurora TelegraphCQ
1	10	-	-
1	9	10	10
2	8	-	9
2	7	8	8
3	-	8	-
4	6	7	7
5	-	7	-
6	5	6	6

Figure 6: A tuple-based windowed query

initially only gives a subset of the results of the corresponding Aurora or TelegraphCQ queries, as shown. This is because the STREAM window only "advances" (i.e., a new window is defined and applied to the stream to create the next "snapshot") each second in STREAM, while it advances on each tuple in the other two systems.

Notice, however, that if tuples are guaranteed to have unique implicit timestamps, the snapshots of both types of windows will be comparable. That is because each time the contents of the STREAM window changes (i.e., a new tuple has arrived), the contents of the corresponding TelegraphCQ/Aurora window will change as well. However, as the latter part of the previous example demonstrates, we may have the reverse problem. That is, unlike the other two systems, STREAM will pass along (identical) snapshots of the window at each clock tick, even when the contents have not changed (i.e., no new tuples have arrived).

In this example, as with most common window operators used in practice, the defining predicates of the window are parameterized by t. Since continuous queries are concerned with time-varying data, this is quite natural. However, if we do not assume that tuples arrive in timestamp order, it becomes more difficult to specify at what points to take snapshots. We would like to wait until we are guaranteed to have seen all tuples with timestamp $\leq t$ before we take the tth snapshot. While we may not be able to guarantee this in general, if the tuples are approximately ordered, taking the tth snapshot at time $t + T$ should give a reasonable approximation. T is sometimes called a "timeout" parameter, and represents the amount of time we wait for "late" tuples to arrive. For example, Aurora allows us to specify a timeout value as an optional argument to our previous example.

We should observe that, since the previous example is based on logical timestamp, the contents of the window are non-deterministic. For example, if 7 tuples arrive in the DSMS simultaneously, the "last" 5 are determined by the logical timestamp. Since this is assigned internally by the DSMS, from the user's perspective, they will be chosen "randomly".

2.3.2 Time-based Windows

Alternatively, we might wish to specify a window of "all tuples in the last 10 minutes". Assume that implicit time is measured (by the DSMS's clock) with a granularity of seconds. The window at time t should include all tuples with timestamp $> t - 600$ and exclude all tuples with timestamp $> t$. Since this only depends on system time, it is called a "time-based" window. Note: As before, in general, this requires the use of a timeout parameter.

This type of window is widely supported. Similar to a "tuple-based" window, this would be expressed in CQL by $S[Range\ 300]$. The specification in TelegraphCQ would be exactly the same as before (after conversion from minutes to seconds), as long as we insure that S is now defined in terms of implicit time. In Aurora, we would simply change the order attribute; that is, the arguments would now be: $Order\ (on\ Time), Size\ 10\ MIN, Advance\ 1\ SEC$. Notice how Aurora provides access to the implicit timestamp and can specify units of time measurement. Since the extent and the advance of the window are both measured in the same time units, these windows generate comparable results in all systems. Although STREAM still may generate duplicate tuples, as we will see, there is a natural way to eliminate duplication so that all three systems generate identical streams.

While Aurora only allows windows of fixed size, STREAM allows one special type of window of dynamic extent, sometimes referred to as a "landmark" window. The operator $[Range\ Unbounded]$ specifies a time-based window which "opens" at the instant the query is registered, and which "closes" at each time instant t. This is more clearly expressed in TelegraphCQ as:

$$for(t = ST; true; t++)\{WindowIs(S, ST, t);\}$$

It is apparent, however, that TelegraphCQ can specify windows quite generally, as long as they open and close at times which are computable functions of t and ST.

2.3.3 Value-based Windows

Additionally, we might wish to specify a window of "all tuples with A values between $5t$ and $5t+5$". More generally, we might specify "values between a_i and $a_i + s$" for some increasing sequence, a_i. Note: If the stream includes an explicit timestamp (or the system allows the window operators access to an implicit timestamp), this includes all "time-based" windows. In this way, we obtain a "value-based" window. As before, we need to have some mechanism for knowing when we may compute the value of the window on the stream, W_t. Intuitively, we would want to do so when we are guaranteed that the tuples of interest no longer have A values in the interval $[a_i, a_i + s]$, but rather in the interval $[a_{i+1}, a_{i+1} + s]$ While a timeout parameter is sufficient, if tuples are approximately ordered on A, we can specify what is essentially a dynamic timeout parameter based on the values of A by specifying a "slack" parameter, k. This means that the window includes all tuples with A value $\geq a_i$, excludes all tuples with A value $\geq a_i + s$. We take a snapshot when we see a tuple with A value $\geq a_i + s + k$, advancing the counter, i. Note: In order to avoid storing the entire stream, a_i should be an increasing function of t.

While neither STREAM[4] nor TelegraphCQ provide such "value-based" windows, Aurora does provide limited support in this direction. For example, by passing Aggregate the arguments: $Order\ (on\ A, Slack\ 4), Size\ 5, Advance\ 3$, we are effectively computing over such a sliding, value-based window with $a_i = 3i$, $s = 5$, and $k = 4$.

Both STREAM and Aurora provide an additional value-based wrinkle in their windowing constructs. They both allow the window to be defined by treating sets of tuples as a single tuple. By

[4]STREAM does employ a dynamic, system-generated slack parameter when ordering a stream on explicit timestamp, as described in Section 2.2.2

using an optional "Partition by $A_1, \dots, A_n$" specification in a tuple-based window, STREAM allows one to count a set of tuples with equal values for $A_1, \dots, A_n$ as a single, logical tuple. An optional "GroupBy" argument in Aurora will have a similar effect when creating Aggregate windows.

2.3.4 Windows and Joins

While Aurora can implicitly define many of the window specifications provided by STREAM and TelegraphCQ, unlike these two systems any such window is explicitly tied to its Aggregate operator. The window operation implicitly specified on the inputs to a Join is value-based, but its precise semantics is quite unlike any of the previous examples. That is because the window specification on one input stream depends on the values of an attribute of the other input stream.

Specifically, a Join between two streams X and Y in Aurora requires that each stream identify an attribute in which it is approximately ordered (with a corresponding slack specification), say A and B, respectively. These attributes should be measured in comparable units because the window on each stream is effectively computed in terms of the magnitude of the difference $X.A - Y.B$. That is, to view Aurora's stream Join in terms of applying an ordinary Join to the output of windowed snapshots of its input streams, the corresponding windowing operator must take two streams as input and generate two sequences of "snapshots" in a coordinated fashion. In contrast, STREAM and TelegraphCQ can only apply window operators independently to each input stream.

2.3.5 Windows and Punctuation

If we may assume that the stream includes punctuation, we may define "punctuation-based" windows. That is, we may open and close windows based on seeing a certain punctuation in the stream. This approach is considerably less succinct than the others mentioned so far. A single window operator automatically gives rise to a sequence of snapshots. To achieve the same effect using punctuation would require a sequence of punctuations in the stream to generate each each snapshot. As we will see in Section 3.3, punctuation is used at a lower level than windows to directly notify the DSMS when it may purge a buffer of tuples.

2.3.6 Window to Stream Operators

Related to windowing is the reverse problem: how do we convert the results of applying aggregate or join operators to one or more sequences of snapshots back into a stream? There are a variety of possible approaches. Assuming that the timestamp values of the individual results of an aggregate and join are well-defined, for each t the snapshot W_t gives rise to a set of timestamped tuples, O_t. We may simply concatenate the results in system time order $(\dots, O_t, O_{t+1}, \dots)$ to obtain the output stream. Because of the large number of duplicate tuples produced by this method, it may be more useful to consider the differences $(\dots, O_t - O_{t-1}, O_{t+1} - O_t, \dots)$. It is also possible to take the differences in the reverse order: $(\dots, O_{t-1} - O_t, O_t - O_{t+1}, \dots)$. Notice that since the tuples in any O_t are unordered, they will be "randomly" ordered by the system as they are placed on the output queue of the operator (cf., logical timestamps). In STREAM, these "reverse windowing" operators are called *RStream*, *IStream*, and *DStream*, respectively. Aurora and TelegraphCQ implicitly use the *IStream* operator.

2.4 SQL Compatibility

As suggested by the example at the beginning of Section 2.3, STREAM's query language, CQL, is simply an extension of SQL to include the windowing operators, along with the "reverse windowing" operators, *RStream*, *IStream*, and *DStream*. TelegraphCQ likewise uses SQL-like language augmented by window specifications.

In contrast, Aurora uses a novel query language called SQuAL. As was mentioned in the previous section, its windowing operators are tightly integrated into its Aggregate and Join operators. While it provides much of the same basic functionality as SQL, it packages this functionality quite differently. It also provides some novel features. We will highlight a few of the similarities and differences.

For example, SQL allows functions to be applied to the resulting attributes of a query, such as:

```
SELECT floor(T.quantity/10)
FROM Trades as T
Where T.symbol = ``IBM'' and T.buy/sell = ``sell''
```

While the support for different functional expressions is large, it is still limited. In contrast, SQuAL provides a flexible Map operator to transform the attributes of a stream that supports arbitrary user-defined functions.

SQL provides an *Order By* clause to sort the results of a query with respect to some attribute. In contrast, SQuAL provides a BSort operator which only *improves* the sortedness of the stream by some degree k. It can only guarantee completely sorted results if the stream adheres to an ordered arrival constraint with parameter k. Note: BSort implicitly uses a tuple-based window of size k.

Finally, SQL provides a *Where* clause which acts like a relational select operator. SQuAL goes one step further with its Filter operator. Filter will split a single stream into several streams based on the values of any number of predicates.

2.5 Stateless processing

All operators naturally fall into one of the two categories: *stateless* or *stateful*. Stateful operators are discussed in the next section. Stateless operators are the operators that maintain no internal state. A stateless operator consists of a function that operates on one tuple at a time and a position in the query network. Moving a stateless operator to a remote location is a matter of moving that function across the network.

2.5.1 Filter

Filter is an operator that is similar to a WHERE clause in SQL. It allows specifying a boolean predicate to filter the incoming stream. A tuple that satisfies the predicate will be passed downstream, while a tuple that fails the predicate will be dropped or sent to an alternative output. In Aurora, a filter can have multiple outputs, each with its own predicate. Thus a filter can split a stream into multiple substreams.

To put filter in the context of our running example, consider the military sensors example. In the interest of saving power, we might decide to transmit only abnormal data, i.e. a heart rate that is too high or too low. Therefore, a filter operator would check every heart rate tuple and only forward the ones that fall outside the specified normal range.

2.5.2 Map

Map provides functionality similar to that of the SELECT clause in SQL. It maps the input tuples into output tuples according to some specified function. For every tuple, a separate function is applied to every field resulting in a new tuple which is sent downstream.

Continuing with the military sensors example, consider a soldier reporting his position on the map. The position is sent in a compressed, binary form. A map function might convert this position report into a human-readable form for the medics.

2.5.3 Union

Union merges multiple input streams into a single stream. Tuples are forwarded downstream immediately as they are received. Thus, the resulting output order depends on the precise arrival times and is nondeterministic.

2.6 Stateful processing

Stateful operators are considerably more complicated as compared to stateless. At runtime, they maintain an internal state that consists of some tuples and meta-data about them. The functions that stateful operators have to execute apply to a window of tuples, thus incomplete windows have to be stored until they are filled. We also need to store window management data (for example, what part of a window has been processed). Having internal state also creates opportunities for sharing the results of intermediate computations.

2.6.1 Aggregates

Aggregate operator computes a composite of a data stream based on a window partition. An aggregate function takes a window of tuples and produces a single tuple for every window it sees. Some of the most typical aggregate functions are average, min and max. Using the stock market example, an aggregate function would be used in order to compute the average price of IBM over every 10 minute window. Note that the output of an aggregate box can have very different characteristics from the input stream. The output schema might not include any of the fields from the input stream. Also, the output stream rate can be either proportional to the input rate if the window is specified as a number of tuples, or independent of the input rate if the window is specified as a time period. The internal state of the aggregate is the exact opposite. An aggregate with a tuple-based window can compute a hard bound on the internal state that it might need to store, whereas a time-based window results in a practically unbounded potential state (limited only by input's throughput capacity).

Both the expected output rate to input rate ratio (a.k.a. selectivity) and the amount of stored internal state are important issues for predicting system behavior and optimizing performance. The expected amount of internal state stored in an operator also determines the amount of space required by operator serialization. This issue might come up when moving an operator in a distributed system or recovering a network after a crash. Depending on the particular aggregate function, the internal state may be represented in a more compact state. For example, if we are computing a maximum over a window, we might chose to store only the largest value number of tuples seen up to this point. This drastically reduces the stored state, but limits us in several ways. The data processed by the operator is eliminated, thus no changes can be applied to the already processed stream. For the same reason, there is not much meta-data to share with other operators. Both of these issues will be covered in more detail later in the paper. Finally, such optimizations are only possible with predefined aggregate functions. There is no easy way to reduce the internal state if the aggregate function has been custom built.

2.6.2 Joins

Join correlates two input streams over some field, producing a single output stream. Two tuples can join only if they satisfy the join predicate and arrive within a window bound of each other. The output stream type in a join is more restricted as compared to aggregate. The output is a subset of all fields present in the input streams, as join cannot introduce new fields. As in the case of aggregate, the window specification type allows us to bound either the output rate or the internal state size. An interesting point to note is that the join output rate can easily exceed the combined rate of the input streams. Any tuple can potentially join with every tuple in the window, producing an output tuple each time it joins.

Join has a very limited ability to eliminate the internal state that it needs to store. Once a tuple becomes too old to join with newly arriving ones (i.e. falls outside of join window) it may be eliminated. Otherwise, it must be stored, since any arriving tuple may potentially join with every tuple populating its window. In order to speed up the matching process, we will need to compute a hash-based index over the tuples that are stored in the join operator. Otherwise, every time a single new tuple arrives, we might have to access every tuple in the respective join window.

2.6.3 Sharing

Stateful operators can compute meta-data or arrange stored data in a way that makes access more efficient. A hash index over join inputs for probing into stored tuples is one example of such meta-data. Once the indexing jobs have been performed, we would like to allow other operators to benefit from this work. Such capability does not have widespread support in the existing streaming systems. However, a few instances of state sharing are used or are being considered for implementation.

Although Aurora does not implement any indexing on tuple queues, it does allow common queue sharing. Any queue that feeds more than one operator will only be stored once. It is the responsibility of the Aurora engine to maintain the minimum necessary set of tuples in that queue. STREAM goes a step further and supports more complex meta-data sharing. For example, if the same queue is simultaneously used by two operators that can benefit from tuple indexing, such indexing would be shared. The query engine builds an index over the queue and makes it available to all operators. Operators that can benefit from hash index include join, and some of aggregate functions.

2.7 Integrating with persistent storage

2.7.1 Operations

The data streams continuously flow with newer data replacing the old. Thus by its nature streaming data is transient and does not need to be permanently stored. Storing data in memory instead of on disk allows for much faster access. As a result, none of the three streaming engines that we discuss in this paper utilize persistent storage during most of their operation. However, persistent storage cannot be eliminated completely. For example, some queries can utilize both streaming data and persistent data on disk. Also, the ability to analyze historical data, and similarly, the ability to recover a failed query network often requires using persistent storage. Finally, the amount of memory tends to be two orders of magnitude smaller than disk space. Therefore any system that does high data rate processing might overflow available memory and spill to disk.

2.7.2 Rollback and Logging

As of now, streaming system environment does not seem to have a well defined notion of a transaction. Many of the applications are content with current data even if there was some gap due to a failure. A query is expected to represent the current state of the system and meet the latency constraints. For example, if a there is a temporary outage in the military sensor example, we would only want the current results once connection is restored. Medics are primarily concerned with the current needs of the soldiers rather than the stale data pertaining to the outage period.

In critical systems (such as financial analysis systems), backup is done via hot swap server. The primary server that performs the operations is duplicated by a secondary server which takes over if the primary fails. There are also other, more involved ways of achieving reliability as discussed in [10]. However, the best we can do is to fall back to some checkpoint and recreate the state. The reason we cannot perform rollbacks is because many of the operators are non-deterministic or irreversible. In order to rollback such operators, we would have to record every tuple as it is processed.

2.8 Correctness

Stream databases do not focus on traditional ACID semantics, nor define a clear idea of what a transaction is. As a result, it is worth asking how valid are the results that come from the database? Or in particular, how is correctness defined? What differentiates a good answer from a poor answer?

The answer is different for different scenarios. For example, in the scenario where soldiers are deployed in the field, a medic may want to monitor for sudden drops in blood pressure. Here, historical information doesn't matter as much as current information. Also, since the user wants soldiers who have low blood pressure, lower values are more interesting than high values.

Both a relational database and a streaming database could be designed to pass information on and fulfil the users' queries. However, in the resource-constrained environment of the deployed soldiers, the system must be efficient. Additionally, the system must be prepared to not be able to keep up with the amount of data collected. Since current values are the most useful, the system in use cannot afford to get backlogged.

As a result, under heavy load, streaming systems may be forced to produce results with some amount of approximation. Along the lines of the earlier question, since an exact answer might not be feasible, what constitutes the "best" answer?

Ideally, one would like to define what the end user cares the most about, ensuring that the results are in line with the user's interest. The STREAM group proposed assigning weights to different queries. This prioritizing could then be used by the scheduler to make decisions to ensure that interesting queries produce more accurate answers than less interesting queries.

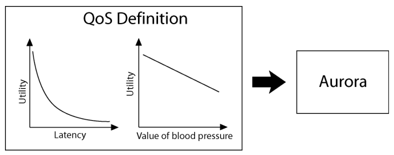


Figure 7: Sample specification of QoS as a function relating a metric to utility

However, in Aurora, a more flexible approach is taken where the user provides a definition of quality of service. This quality of service (QoS) is defined as a set of relationships between parameters of the outputs and its usefulness, or utility. A QoS definition may rate some outputs as more useful than others, thus allowing the system to make decisions to encourage those results to reach the output. In this manner, one can simulate assigning static weights on each query by assigning constant weights to all values of each of the outputs. Additionally, one can be more expressive and say that larger values are more interesting than smaller ones, or express that high latency is acceptable for some queries, but not so for others.

The QoS is formally defined as a function mapping metrics to utility. Returning to the example of the medic asking about low blood pressure, we can define the following two QoS definitions that can be provided to Aurora.

Shedding load will obviously impact the accuracy of the output of any queries. However, not shedding load can result in high latency as bursts of data must be queued and processed to completion. This results in an inherent trade off between accuracy and latency. Then by allowing the user to define utility as a function of latency, the user can then tune that tradeoff to be aligned with his interest.

This notion of QoS is extremely powerful and descriptive, and defining it on parameters of output is the most direct and useful approach but it is not easily utilized. The earlier in an execution plan changes are made to shed load, the more effective those changes will be. However, it is very difficult to reason about data backwards through an operator. One needs to be able to make decisions early in the stream. However, the the decision is made, the harder it is to reason about the impact the changes will have on the final result. As a result, it has been proposed that QoS would be better defined on the incoming streams, making it much easier to determine which data is more valuable.

The actual mechanics of how data is selected to be sampled and dropped is covered later under resource management.

3 Resource Management

3.1 Resources

An important challenge in database management systems has always been how to optimally utilize resources in order to maximize performance, while at the same time balancing other factors such as recoverability and reliability. This remains true in streaming databases, but often with a different emphasis.

Streaming databases deal with push-based sources that often feed streams in through a continuous query registered with the system. Often, the usefulness of a result depends upon how quickly it was produced. This means that minimizing latency and maximizing throughput is typically very important, making it highly desirable to minimize cpu and memory usage. Techniques to accomplish this range from shedding tuples in order to reduce load on the system, to scheduling operator queues in order to optimally reduce the amount of tuples needed by the system. Many of these techniques, such as load shedding, fundamentally affect the accuracy of the query by, in essence, changing it. Therefore, it is necessary to develop approximation techniques and measures of performance to balance performance versus accuracy and to give some guarantee of a certain level of accuracy.

By nature of its applications a DSMS lends itself to a distributed implementation in which other resources, such as power and network bandwidth, must be considered. For example, think of a DSMS composed of thousands of small sensors which have a relatively small battery life, such as in the military example. To reduce the amount of bandwidth used by the streams from all these sensors, a distributed DSMS may slide an operator down to the sensor level. While this may conserve bandwidth and additionally reduce load on the DSMS above the sensor level, it may also put an additional power strain on the sensors. This introduces an additional level of complexity above that of the non-distributed case as the various needs of each component of the system must be considered when managing resources.

3.2 Operator Scheduling

In a conventional DBMS, queries are run against datasets which are finite and which do not change while the queries are executing. The goal of the query optimizer in these systems is usually to minimize the average time required to execute each query.

In a streaming data system, however, queries run indefinitely, and they must process datasets that are constantly growing. If the system does not schedule the execution of query operators intelligently, the backlog of tuples at some of the operators may exhaust the available memory. As mentioned above, conserving memory is important in a stream system because it reduces the demands on other system resources. Therefore, in a DSMS, intelligent operator scheduling is critical to effective resource management.

3.2.1 Why Not Just Create a Bunch of Threads?

A naïve approach to operator scheduling would be to create a single thread of execution for each operator in the stream manager. An advantage of this approach is that implementation is simpler — most modern operating systems already have sophisticated mechanisms for scheduling the execution of threads. By creating a thread for each operator, the problem of scheduling is essentially delegated to the operating system.

There are, however, problems with this approach. The first problem is that it doesn't scale well. A data stream application could contain thousands of query operators. Yet many operating systems can't handle an arbitrarily large number of threads. [5]

Even if an operating system can handle a large number of threads, it might not be able to switch between them efficiently. The thread scheduler may have to scan all active threads whenever it performs a context switch. If this is the case, the time complexity of the thread scheduler is linear in the number of threads. With a large number of threads, the time overhead required for scheduling becomes unwieldy. [8]

The second problem with the naïve approach is that the operating system isn't aware of the quality of service (QoS) requirements for the application. [5] As an example, consider two stream queries running in the military sensors application. The queries, in plain English, are as follows:

```
Query A: Every 5 minutes, report the average foot speed of the soldiers in the
unit.

Query B: Report the ID number of any soldier whose health is at risk.
```

Obviously, the latency requirements are much stricter for Query B than for Query A. However, if the operating system were responsible for scheduling, it would tend to allocate roughly equal amounts of processor time to each operator. What is needed is a centralized scheduler that has information about the QoS requirements for all of the queries in the system. [8]

The third problem with the naïve approach is that the operating system doesn't know enough about how the stream operators work to manage resources effectively. A custom stream scheduler, on the other hand, can make use of certain types of information to minimize the total resource usage. The techniques for doing this are described in the next section.

3.2.2 Scheduling Algorithms

The scheduler in a DSMS can take advantage of two types of information to minimize resource usage. The first type of information is the non-linearity of the processing time for an operator. An operator may be able to execute more efficiently by processing input tuples in batches, or "trains". [5] The reason for this is that, in order to execute an operator, the system must perform certain tasks that are not directly related to the function of the operator itself. One such task is fetching tuples for processing. Another task is switching from one operator to another. The amount of CPU time required for these tasks tends not to be a linear function of the number of tuples to be processed. So by processing tuples in trains, the system can reduce the time cost per tuple. This in turn reduces memory usage because there is less time for input stream tuples to accumulate in the operator queues.

The second type of information that a scheduling algorithm can exploit is the selectivity of the operators. An operator's selectivity is the number of tuples it outputs in a given period of time, divided by the number of tuples it receives over the same period of time. If the selectivity is less than one, the operator produces fewer tuples than it consumes. This is usually the case with selection (filter) operators. If the selectivity is greater than one, the operator produces more tuples than it consumes. This is often true for join operators. In practice, the selectivity is almost always an estimated value, based on various heuristic techniques.

By taking selectivity into account, the scheduling algorithm can reduce the memory requirements of the stream application. As an example, consider a query in a streaming data application 8. The query consists of two operators. Operator 1 has selectivity 2.0, and Operator 2 has selectivity 0.25. Assume that the time cost per tuple is the same for both operators. If the scheduling algorithm didn't consider selectivity, it might process several tuples for Operator 1 and then move on to an operator in a different query. In the meantime, several tuples would be left waiting in the queue between Operator 1 and Operator 2. The number of tuples in the queue would be twice the number of input tuples that had been processed by Operator 1. So by executing Operator 1, the stream manager would have increased the total number of tuples present in the system.

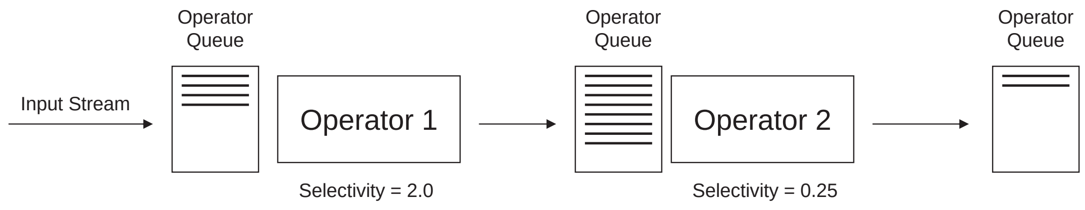


Figure 8: A simple query. The scheduler can optimize the use of resources by taking into account the selectivity of the operators.

A better approach would be to treat Operator 1 and Operator 2 as a single unit, by pipelining the output from Operator 1 into Operator 2. The combination of the two operators effectively creates a new operator with selectivity $2.0 \times 0.25 = 0.5$. The advantage of using this combined operator is that, since its selectivity is less than one, it produces fewer tuples than it consumes. In other words, whenever the combined operator is given a time slice, it will tend to reduce the total number of tuples present in the system. The technique of combining operators based on selectivity is called *chain scheduling*. [3] [5]

3.2.3 Dynamic Tuple Routing

So far we have assumed that query plans are static. This is not a requirement. It's possible for a stream manager to reorder operations on the fly, as the queries are executing. This can be advantageous for a streaming data application, where the queries run indefinitely and stream properties can change over time.

In the Telegraph system, dynamic query optimization is accomplished using modules called Eddies. The role of an Eddy is to route tuples among various processing modules, according to some routing policy. For each processing module and for each tuple, the system estimates the benefit and the cost of sending the tuple to that module. The Eddy then decides where to send each tuple based on this cost/benefit analysis. The current progress of each tuple — specifically, the operations that have been performed on it so far — must be stored in the tuple itself. [12]

One operation which can benefit significantly from dynamic routing is a join. Typically, a join between two data sources, R and S, is implemented as follows: for each tuple in R, probe S for a matching tuple; if a match is found, output both tuples. With dynamic routing, however, the join operation can be decoupled into two symmetric operations called *half-joins*. Each half-join operates on a single stream.

In Telegraph, half-joins are implemented as modules called State Modules, or *SteMs*. Each SteM accepts two types of request tuples: *build* tuples and *probe* tuples. When the SteM receives a build tuple, it adds the tuple to its lookup table. When the SteM receives a probe tuple, it outputs any tuples in its lookup table that match the probe tuple. A join can be implemented by using an Eddy to route input tuples first to one SteM, and then to another. Figure 9 shows a join operation as implemented in the Telegraph system. [12]

The advantage of the Telegraph approach is flexible scheduling — input tuples can be routed to the SteMs in any order. If one SteM becomes too busy, the system can decide at run time to route tuples to another SteM instead.

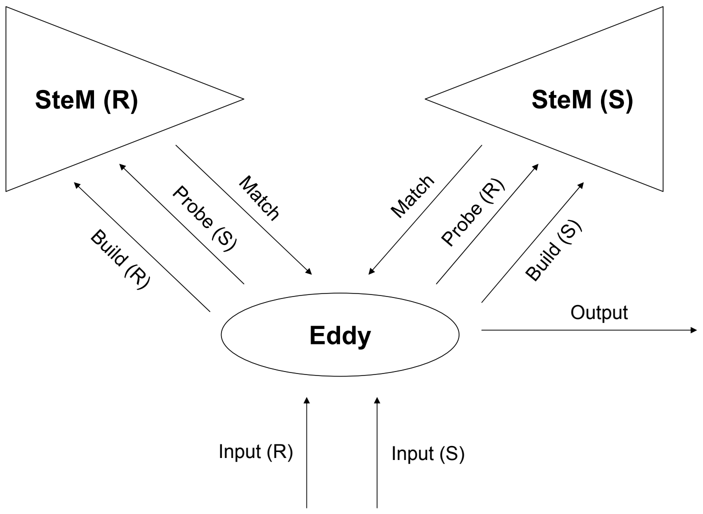


Figure 9: A join as implemented in the Telegraph system.

3.3 K-Constraints and Punctuation

The performing of continuous aggregate or join queries over such streams can require a prohibitive amount of state to be stored and/or may require the system to block until it can compute the results of the query. Although it is possible to store this state on disk, for performance reasons, it is desirable to do this in memory instead. However, as memory is far more limited than disk, techniques must be employed to limit the amount of state stored. Also, since a DSMS should produce results in a timely fashion, blocking is undesirable. Two devices, k-constraints and punctuation, have been employed to handle these time and space issues.

As mentioned in Section 2.2.2, k-constraints can be used to measure the degree to which the data is ordered or clustered on an attribute in single stream and the degree to which two streams are correlated for joining. These constraints may be enforced by the user within a query itself or may be inferred by the system by observing the stream over time.

For example, the slack parameter, k, in an Aurora value-based window may be viewed as the parameter of an ordered arrival constraint. Any tuples that do not adhere to this constraint are simply dropped. That is, if the stream satisfies this constraint, it will give exact results. However, to the degree to which it fails to adhere to this constraint, the results will only be approximate. As was pointed out in Section 2.3, this constraint serves to allow an aggregate to be computed over the window without blocking.

As was mentioned in Section 2.2.2, STREAM timestamp-orders its streams on entry into the system. To do so in a non-blocking manner and with limited memory, it must likewise utilize an ordering constraint. Unlike Aurora, however, it collects statistics on the stream to dynamically estimate an appropriate value for k. While this may still lead to some degree of approximation,

Input	1	3	3	5	2	4	8	6	6	10	5	9	12	11
Constraint	-	-	-	> 1	> 3	> 3	> 5	> 5	> 5	> 8	> 8	> 8	> 10	> 10
k = 2														
Buffer		1	1	3	3	3	4	5	6	6	8	8	9	10
			3	3	5	5	5	8	8	8	10	10	10	12
Output	-	-	-	1	3	-	3	4	5	6	6	-	8	9

Figure 10: Ordering a Stream with $k = 2$ Ordered Arrival Constraint.

this is controlled by the system based on its space limitations. For example, Figure 10 shows how a stream which is approximately ordered with $k = 2$ could be ordered using a buffer of size 2.

Similarly, a clustered-arrival constraint limits how much we must buffer when processing a grouped aggregate. For example, assume that stream S is pushing tuples of the schema `S(TIME, SYMBOL, PRICE, EXCHANGE)` which adheres to a clustered-arrival constraint on `S.TIME` with a $k = 2$. Assume that the average stock prices for each company are being calculated on a daily basis and currently tuples with an `S.TIME = 5/18/2001` are being streamed in. Once $k + 1$ tuples have arrived with a `S.TIME = 5/19/2001` it can be assumed that there will be no more tuples with a `S.TIME = 5/18/2001` and the average can be computed and state for `5/18/2001` flushed.

In Section 2.2.4, we indicated how punctuation could also be used to limit a computation. Since punctuation is part of the stream, all query operators must be extended to specify how to handle punctuation. In particular, each operator must know when it is time for a state purge or to pass on partial results. Niagara implements this by allowing definition of pass, purge and propagation rules. Specifically pass defines when results can be passed on, purge discards state and propagation determines how and when punctuation can be passed on to additional operators.

3.4 Approximation Techniques

Consider a DSMS that has a total of 10 memory units and accepts incoming tuples at the rate of 1 per second. While being processed by the DSMS, a tuple consumes $\frac{1}{10}$ of a memory unit. This means that the DSMS can hold at most 100 tuples in memory at any one time. If it takes 2 seconds to process each tuple from start to finish, we can see that a problem will definitely arise. Namely, tuples arrive twice as fast as they are processed. Given this condition, memory will eventually saturate, and the DSMS will no longer be able to handle incoming tuples.

In addition to input overload, we might also imagine query processing as a resource bottleneck. As an example, take the following query which selects distinct ticker symbols from the `Trades` stream that involved more than $10,000$ shares.

```
SELECT DISTINCT
FROM trades
WHERE qty > 10000
```

What's interesting about the query above is that in order to select distinct `Trade` tuples, the DSMS will need to remember every tuple that it has seen. Tuples will accumulate on the operator queues until available memory is exhausted. Again, finite system resources saturate in the face of an infinite data stream.

To mitigate the potential for harm that the above, or similar situations may cause, a DSMS will make use of **approximation** techniques. It will degrade system performance in some respect while still providing the end user or application with results that are close to, but not completely, accurate. That is, it will provide approximate answers to queries. The following subsections delve into the most prevalent approximation techniques and describe the details of their use.

3.4.1 Load Shedding

Perhaps the easiest way for a DSMS to deal with a resource bottleneck is to simply disregard certain tuples. This is referred to as **load shedding**. For example, given the tuple overload in the first scenario described above, the DSMS could choose to ignore every other incoming tuple. This should guarantee that tuple load is in line with system resources.

When implementing a load shedding strategy, there are four main questions that should be answered. The first, and probably most important, is how to shed load. In this regard there are only two options. A DSMS can either shed load deterministically or at random.

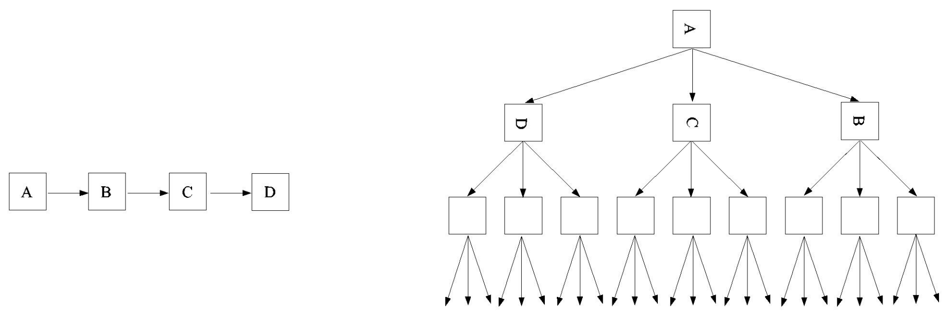
If the input tuples are relatively uniform, then simply dropping tuples at random should suffice as a load shedding solution. This is sometimes also referred to as **sampling** the data stream. However, if there are wide variations in the data values, then dropping tuples at random could significantly skew an approximation.

Let's illustrate this distinction using the military sensor example application. Suppose the body temperature sensor sends tuples to the hub worn by a soldier once a minute. Again, tuples are of the form: `<sensor type, sensor id, time, value>`. It is then the job of the hub to forward tuples on to an intermediate station so that they can then be forwarded on to the medic. In this case, there are 100 soldiers in a squad all sending tuples to the same intermediate station. The intermediate station is located on a Humvee but it is running out of battery power. As body temperature is generally very constant, the station on the Humvee decides to sample the body temperature tuples it receives. It randomly selects 75 of the 100 tuples per minute and forwards them on the the medic at the base station.

Now, imagine one or several soldiers were to be wounded. Body temperature is suddenly a vital statistic, and the medic on call will want to observe all possible body temperature values for all the soldiers in the squad. The danger is that the wounded soldiers body temperature tuples could potentially be sampled out of the stream by the station on the Humvee. The situation might worsen if tuples from the same wounded soldier were sampled out over the course of two consecutive minutes. Potentially life-saving information is disregarded. Clearly, sampling is very sensitive to extreme and sudden changes in data stream values.

In contrast to dropping tuples at random, **deterministic** load shedding filters out tuples whose values fall outside a specified range. Continuing with the body temperature example, to save battery power the Humvee could decide to only forward tuples whose values fall outside the range $97-100$. The idea is the medic only really needs to receive information when things go wrong. The hub on the Humvee can then drop any body temperature tuples on arrival whose value falls between $97-100$ and save on battery power.

While the above example may be a bit contrived, it illustrates the basic point. The nature of the streaming data must be considered to determine an effective load shedding strategy.



(a) (b)

Figure 11: (a) A single chain of operators (b) a balanced query processing tree

The next issue to address is deciding when to shed load. A DSMS should be able to detect system overload and react accordingly. A straightforward way to do this is to use statistics. During a test run, the DSMS might gather a profile of system resource statistics under heavy load. It could then define a **headroom** value which represents a buffer between maximum operational conditions and complete resource saturation. The DSMS would then use the headroom value in a live environment to determine how close it is to being overloaded.

As an abstract example, assume that given tuple load, processing times, available memory and cpu resources a DSMS can define a cost function f. It then uses that function to identify that optimal operating conditions occur at value x and complete system saturation occurs at value y. The headroom, h, is then the range of values between x and y. In a live environment, if the current operational costs, c, approach or breach the headroom range, then the DSMS can proactively shed load to degrade performance according to QoS expectations.

The third aspect of a load shedding policy that must be decided upon is finding the best location in the query processing tree to shed load. This involves a tradeoff between processing cost and query accuracy. Shedding tuples early avoids wasted processing. It is undesirable for a tuple to be operated on early in a query processing tree only to be dropped farther down the line. On the other hand, dropping a tuple early could adversely effect later processing. This is especially true if operator **fan-out** is high. Fan-out refers to the number of operators that immediately follow an operator in a query processing tree.

As always, this tradeoff is best illustrated with an example. Consider Figure 11(a). A tuple arrives at operator A, is processed, and is then sent to B. It is then processed at B and sent to C and similarly from C to D. In this situation, if a tuple is to be dropped it is best to do so before it enters operator A, saving the cost of the work performed at each of the operators. Now consider Figure 11(b). Instead of a single operator chain, B, C and D each immediately follow A and are the root of a separate query processing tree. The fan-out of operator A is 3. Dropping a tuple after it leaves A will adversely impact the answer provided by the trees rooted at B, C and D. As a general rule, the farther up a query processing tree tuples can be dropped while still taking into account the impact on the subtrees, the better.

The final component of a load shedding policy is determining how many tuples to drop. In the case of random drops, this means deciding what percentage of the tuples a DSMS sees it should disregard. In the case of deterministic load shedding, this amounts to linking a percentage of tuples to a filter predicate. In either case the answer usually is application specific and depends on stream rates, query processing costs and available system resources.

3.4.2 Histogram

Another means of approximation available to a DSMS is a **histogram**. A histogram is a summarization technique used to capture data value distribution. Formally, given a stream S and a property of that stream A, a histogram of S would be the distribution $(v_1, f_1), \ldots, (v_n, f_n)$, where v_n represents a value of A, and f_n represents the frequency of that value. Put another way, the domain $S.A$ is partitioned into a number of buckets, where each bucket stores a summary of value distribution. Varying the number of buckets will vary the granularity of the approximation.

While there are many types of histograms, this paper considers only a specific type and demonstrates how it might be used by a DSMS. In an **equi-width** histogram, the number of possible values are partitioned into buckets, where each bucket has the same value range.

For example, assume the stream S reports the pedometry rate of a squad of 100 soldiers over the course of 2 hours in which they are involved in combat. A pedometry tuple is sent by the hub worn by each soldier to the intermediate station once every 15 seconds. Whereas pedometry rate is not as vital as heart rate or blood pressure, it might be approximated during peak load.

To make it easy, we will also assume that the range of reported pedometry rates is $1 - 100$. We then might divide this range into 10 buckets where each bucket contains 10 values. The first bucket will store the reported frequencies of values $1 - 10$, the second $11 - 20$ and so on. Each bucket also stores its boundary values (1 and 10, 11 and 20, etc.), as well as the sum of the frequencies for each of its values. Consider the following list as a bucket:

```
[91,6], [92,3], [93,5], [94,4], [95,2], [96, 7], [97,2], [98,1], [99,4], [100,4]
```

The number on the left represents a pedometry rate and the number on the right its reported frequency over the course of the two hours. The boundaries of this bucket are 91 and 100. Given the following query:

```
SELECT COUNT(S.value)
FROM S
WHERE S.value > 90 AND S.value <= 100
```

an overloaded DSMS could use the bucket to provide an approximate answer. To do so, it first computes the total for all the frequencies in the appropriate bucket, in this case 38. It then assumes an equal distribution of frequencies among the bucket members. From the values above we have, $38/10 = 3.8$. While this is clearly not the correct answer, it does serve as a rough estimate.

It is worth mentioning that in addition to equi-width histograms, there are also *equi-height*, *Optimal-V* and *Max-diff* histograms. For space considerations, the details of the algorithms are omitted

3.4.3 Timeout

Imagine a query that computes average pedometry rate for a squad of soldiers, each with a hundred members. The query that is executed must wait for each member of the squad to report a pedometry rate before executing. In optimal conditions, the sensors worn by the tuples emit a pedometry rate once every 30 seconds. At that rate, the DSMS can execute the query twice a minute. But, what happens if one of the sensors worn by a soldier fails? The DSMS will have 99 tuples, but will be blocked waiting for the 100th. It DSMS should not block forever though, as the final tuple may never arrive. It is precisely for this reason that the need for a **timeout** mechanism becomes apparent.

As described earlier in the paper, a **timeout** parameter is used in a time-based windowing operation as a maximum period for which to wait for late tuples to arrive. If the window operation has not emitted a result tuple after time t, the DSMS will execute the query on the tuples that it currently has and disregard any tuple belonging to that computation arriving thereafter.

The use of timeouts in query processing clearly provides the DSMS with a tunable approximation parameter. To see this, consider a query registered at an intermediate station which computes the average pedometry rate for one soldier every half hour.

```
SELECT AVG(S.value)
FROM (S [Range 30 Min]) as S
WHERE TIMEOUT=5 min
```

Again, the stream S sends a pedometry rate tuple for an individual soldier to an intermediate station once every 15 seconds. Assume one of the soldiers becomes separated from his unit for a period of 6 minutes. This separation causes all of the soldier's pedometry tuples for that duration to be lost. With the `TIMEOUT=5 min` clause in the query above, the pedometry tuples that have already arrived at the intermediate station will remain there for an additional 5 minutes. They will consume space in memory waiting for lost tuples that will never arrive.

Now, imagine the `TIMEOUT` value in the query above was set to 0. Once the 30 minute window had expired, the DSMS would immediately execute the query, releasing the tuples to their next destination. Thus, to counteract system overload and save memory a DSMS might decrease `TIMEOUT` values of window based operations.

3.4.4 Slack

Related to timeout is the notion of **slack**. Slack is a bound on the disorder expected in the stream. It measures how far out of place a data item might be from its correct position.

For instance, suppose 10 tuples of a fictional stream `S` were to arrive at a DSMS and each tuple has an integer field, `A`. If the slack value for `A` stream were set to 2, then any two values of `S.A` will be at most two places out of order. This is very similar to the *ordered arrival k-constraint* described earlier in the paper.

As with timeout, the higher the slack value, the more resources are needed for its accommodation. Conversely, if no slack whatsoever is allowed, resources might be conserved: any tuple that arrives out of order will simply be disregarded.
Imagine the following series of `Trade` tuples arriving at a DSMS

```
1.<1:00, IBM, 105, NASDAQ>
2.<1:05, IBM, 112, NASDAQ>
3.<1:10, IBM, 114, NASDAQ>
6.<1:25, IBM, 119, NASDAQ>
7.<1:30, IBM, 117, NASDAQ>
4.<1:15, IBM, 115, NASDAQ>
5.<1:20, IBM, 118, NASDAQ>
```

Where each tuple is of the familiar schema `<time, symbol, price, exchange>`. In addition, consider the following query:

```
SELECT AVG(T.price)
FROM (Trades [RANGE 30 MIN]) as T
WHERE T.SYMBOL='IBM' AND SLACK=0
```

With a slack value of 0, once the tuple for `1:30` arrives, the DSMS is free to compute the average and stream the result. With the tuple arrivals as listed above, the result would be an approximation as the tuples for `1:15` and `1:20` were not included in the computation. If the clause in the query above were changed to `SLACK=2` though, the `1:15` and `1:20` tuples would be included in computation. But again this means they would be held on the operator queue, and hence in main memory, for a longer period of time.

3.4.5 Reductions

Simply reducing the size of any window operation is also a form of approximation. Instead of computing an average over a window of 100 tuples, or time span of 30 seconds, we might reduce the window size to 50 tuples or the time to 15 seconds. This allows the DSMS to hold fewer tuples in memory, but obviously represents a less comprehensive and precise calculation.

Although window size, timeout and slack query constructs are generally specified by end-users or applications, they also represent stress relief points for a DSMS. Taking into consideration QoS requirements, an overloaded system might proactively manage these constructs as a first step toward graceful performance degradation.

3.5 Performance Measurements

Attempting to develop a set of performance evaluation criteria for a Data Stream Management System is problematic. Not many metrics are valued across all systems, but rather are application dependent. This subsection explores the few criteria common to most DSMSs and discusses the specific situations where they might be applied.

3.5.1 Linear Road

The most basic measure of performance for any system is whether it actually exhibits the behavior and characteristics of a DSMS. If a software system does not support continuous queries or windowing operations, it is not really a DSMS. This is exactly the role of the *Linear Road* benchmark. Defined by two researchers at MIT, Linear Road is a hypothetical road traffic management system used to measure the performance of data stream management systems and to illustrate how that performance cannot be matched by a traditional DBMS re-engineered to handle streaming data.

Very briefly, the Linear Road application uses variable tolling – adaptive, real-time computation of vehicle tolls based on traffic conditions – to regulate vehicular traffic on a system of highways. Each vehicle is equipped with a sensor that continuously relays its position and speed to a central server. The server aggregates the information received from all vehicles on the highway system, computes tolls in real-time, and transmits tolls back to vehicles using the sensor network [2].

To evaluate the performance and responsiveness of implementing a software system, Linear-Road specifications state that it should be run on a single processor Linux box. The number of highways in the scenario should then be increased until the system can no longer keep up with the input data stream. The output for any software system is the number of highways that it can run for each scenario.

In terms of performance, Linear Road specifies two metrics to which an implementing software system must adhere. The first is a set of acceptable outputs. Given a set of queries defined by the benchmark and various tuple loads, the system must produce the correct output, in the correct format, within some error bound. The second metric is a set of quality of service requirements for the queries themselves. The QoS requirements are simply response times which must not be exceeded.
For more information on Linear Road visit:

`http://www.cs.brown.edu/research/aurora/Linear_Road_Benchmark_Homepage.html`

3.5.2 Quality of Service

Another good way to measure DSMS performance is by considering **Quality of Service**. As stated previously, Quality of Service specifications can be measured in any number of ways, including response times, tuple drops, and accuracy. Moreover, QoS can be viewed from both a system-design perspective as well as from an application or end-user perspective. As we have just seen, the Linear Road benchmark is system-design QoS: a set of standards to which aspects of the DSMS must apply.

When considering QoS from an end-user or application point of view, things are a bit less concrete. The nature of the application really defines QoS requirements. For example, the stream-based stock market example application is likely to have the most strict requirements regarding tuple drops, accuracy and latency. Specifically, tuple drops are intolerable, accuracy must be as close to 100% as possible, with little or no latency. On the other hand, due to the uncertain nature of the streaming data in the remote military triage application, QoS requirements must be relaxed and redefined. Tuples will definitely be lost, load will vary, sensor batteries will die, and bandwidth can be insufficient. In this type of unpredictable environment, it is left to the users of the application to define reasonable performance requirements.

Regardless of the application however, it is easy to see how the performance of the DSMS can be judged against user-defined QoS expectations.

3.5.3 Throughput & Latency

Throughput in a DSMS refers to the end-to-end speed with which tuples are processed. In general, throughput makes a good comparison metric as given the same tuple load, queries and data, one may compare the relative throughput of two DSMSs.

Related to the idea of throughput is **latency**. The latency of a DSMS is its responsiveness. Given tuple arrivals at a DSMS operator, latency is a measure of how long it will take to produce a result.

While both latency and throughput are a measure of speed, it is important to note the difference between the two. Latency is a measure of the amount of time between the start of an action and its completion, whereas throughput is the total number of such actions in a given amount of time.

To illustrate this difference, suppose we have two different operators that compute an average. Each of the operators resides in a separate DSMS. The first operator, on system A, computes the average of 1000 tuples, and takes 10 seconds to do so. The operator on system B, computes the average of 50 tuples but only takes 2.5 seconds. Effectively, the operator on system A computes the average of 100 tuples per second (*throughput*), but every tuple stays in the operator for 10 seconds (*latency*). In contrast, the operator on system B computes the of average of 20 tuples per second, but those tuples leave the operator much sooner, in 2.5 seconds.

Given the fluid, non-blocking nature of a DSMS, latency is often a better gauge of system performance than throughput. With infinite, streaming data sources, the responsiveness of a system is key. Often it is better to know the answer, or even an approximate answer, to a query as soon as possible, rather than processing a greater number of tuples per time period.

3.5.4 Accuracy

A final performance metric to discuss is **accuracy**. As was described in section 3.5.1, the DSMS system should produce results consistent with its current load. If the system has ample resources to process the incoming tuples, and tuples arrive at anticipated rates, then it is reasonable to expect perfect answers. However, as described above, this is not always the case. The DSMS will use approximation techniques to conserve system resources which will distort the accuracy of query results. Thus, accuracy of query answers should be considered in the context of system load.

If the answers are not accurate, then this is the first indication of a problem with the system. As was just stated, the DSMS might be overloaded. If so, then there must be a bottleneck somewhere. Perhaps system resources are being overtaxed. Or, a query processing module could be holding things up. On the other hand, there might be a problem with the data sources. One or more of the streams might have failed or outside interference might have increased transmission latency from the source. In short, query accuracy serves as the first and most prominent metric for an end-user or application to gauge DSMS performance.

4 Distributed operation

4.1 Settings

To this point, this paper has detailed many of the design and architecture aspects of a data stream management system. However, it has done so under the assumption that all system components will reside on the same machine. While for most small scale streaming applications this may suffice, larger applications will require more resources than one machine can provide. For example, they might need a group of machines acting in concert to handle the query processing load.

We might also want to remotely deploy aspects of a DSMS to perform a specific task or small group of related tasks. We can imagine a wireless PC or PDA that allows users to query the system or filter tuples. In general, there are three types of environments for a distributed DSMS architecture: *clustered server*, *wide-area* and *wireless*. Of course, a distributed architecture might combine two, or even all three of these environments. Each of these environments is examined in more detail below.

That is, it could incorporate diverse participants such as physically connected stream processing nodes, as well as the aforementioned wireless devices, to networks of sensors that provide input streams.

4.1.1 Clustered Server

A **clustered server** environment is a group of machines operating within a single domain. While this may take any number of forms, the assumption is of a traditional shared-nothing architecture: duplicated high performance disks and processors connected by a high speed communication link. Essentially, it is a distributed architecture with no single point of failure.

There are several characteristics of a clustered server environment that are worth noting. First, as with a single node DSMS, the system administrator has access to and control over the entire distributed environment. He is free to configure and optimize the system in any way he chooses. This is not case with *wide-area* and *wireless* environments. Secondly, a clustered server environment uses a distributed query plan. To make use of distributed resources to process queries, the query plan must be partitioned amongst nodes. This gives rise to interesting challenges with regard to load sharing, operator splitting and shifting, and tuple routing. Finally, every clustered server environment has a coordinator node that maintains a global view of the system state. This allows it to act as an intelligent load balancing mechanism, routing tuples between machines given the load and query processing requirements of the system.

4.1.2 Wide Area

While the distributed resources provided by a clustered server environment address the needs of larger streaming applications, there is still a class of applications for which it is not sufficient. Streaming services such as weather forecasting, or road traffic management that gather and process data from different autonomous domains would not be handled well by a clustered-server DSMS. Specifically, a clustered sever DSMS is not suited to coordinate resources and query processing across several organizational administrative boundaries.

In general, we can imagine a federated DSMS in which each participating organization contributes a modest amount of computing, communication, and storage resources. This would allow for a high degree of resource multiplexing and sharing, enabling large time-varying load spikes to be handled. It also helps improve fault-tolerance and resilience against denial-of-service attacks [7].

This is what is meant by a **wide area** environment. Instead of distributing stream processing amongst nodes within a single domain, it is distributed across many domains.

Distributing processing in this fashion can pose several problems. Chief among these is coordination. A wide-area DSMS has to incorporate a heterogeneity of systems, resources and motivations to process a query. Note the inclusion of motivations in the previous sentence. Not only should performance and system resources of each participant be acceptable to all participants, but also participants might need sufficient motivation to process portions of queries. Put another way, a participating system or organization has no reason to process a query, or portion thereof, if they receive no benefit as a result.

Additionally, for performance reasons some applications may want their queries processed within a specified set of acceptable domains. We can see this in the example stock market application. As stated in section 3.5.2, this application would likely have the most strict performance requirements. For this reason, it might not want any system whose requirements are lower than its own to process parts of its queries. At the very least, they could increase latency times. However, they also might introduce tuple loss or bursty data flow, both of which are completely unacceptable given the nature of the stock market application.

4.1.3 Wireless

Stepping away from the physically connected world, a DSMS might also be deployed remotely. This is exactly the situation in the military sensors example. Soldiers wear sensors that monitor vital statistics and hubs that communicate with stations on vehicles. The vehicles might have a modified PDA of PC that allows them to receive tuples and run queries against them. For instance, they might want to filter out tuples that report "normal" vital statistics and forward tuples for squad members whose vital statistics fall outside some specified range.

Perhaps the single most limiting resource when working in wireless environments is power consumption. As nodes are deployed remotely, they run on battery power. By definition, a battery has a limited lifespan and storage capacity. Simply by transmitting and receiving bytes of information, nodes diminish their power supply. Thus, to conserve battery power, nodes only want to send and receive information as little as is absolutely necessary. Power conservation techniques are addressed to a greater extent later in this section.

4.1.4 Challenges

With the benefits of a distributed environment come a whole new set of challenges. First and foremost is the issue of scalability. Under continual heavy load, it should be easy to integrate new resources into the system. Also, a distributed DSMS should also be to accommodate geographically disparate systems. Finally, there is the issue of administrative scalability. A distributed DSMS should be administratively scalable as well.

If the DSMS does in fact scale, questions as to how it will use distributed resources immediately arise. One concern in particular is load balancing. Given a distributed DSMS, it should define a policy of how it will balance load amongst nodes. In this respect, it should take into account whether load will be balanced across the entire system, or locally amongst a group of nodes.

Next, the distributed DSMS should define load balancing semantics. One option is to move operators from overloaded machines to idle ones. However, this will incur a high communication cost and does not translate well to a wide-area environment. Alternatively, the distributed DSMS might employ something similar to RPC, providing nodes with an API so that they might instantiate remote operators to help with query processing. Finally the DSMS might want to replicate functionality across nodes, split streams and re-route tuples.

While QoS specifications are difficult enough to define for a single node, they pose a significant challenge in a distributed setting. Specifically, it is difficult to ensure QoS at the output streams when interior or intermediate operators are unaware of QoS guidelines. This difficulty increases when trying to reason about QoS across administrative domains.

When considering distributed architectures, the issues of correctness and reliability must be addressed. Whereas transactions define correctness for a DBMS it should be determined if this

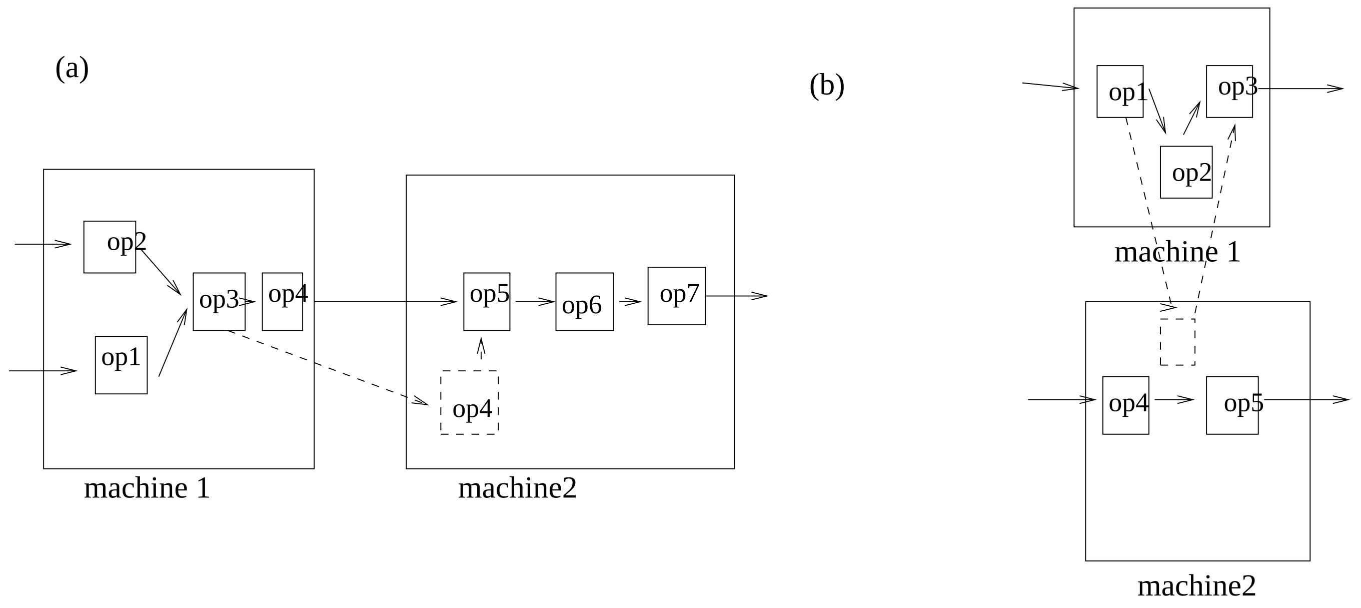


Figure 12: Two examples of moving operators. The dashed lines indicate where the operator will be moved.

is also the case for a DSMS. Given the transient nature of a DSMS, the blocking and time requirements of transactions seem to preclude its inclusion.

Eventually one or more machines will fail. A policy should established to handle this eventuality. This includes exploring recovery and re-incorporation mechanisms for the failed machine.

Deploying a DSMS in a wireless environment presents several of its own challenges. As previously mentioned, processing algorithms will have to be modified to account for power conservation. In a related issue, sensor failure or temporary disconnection will obviously impact tuple routing and query processing.

The sections that follow delve into more detail as to specific approaches to addressing many, if not all, of the aforementioned challenges. The first of which, *Load Sharing*, describes several techniques a distributed DSMS might employ to balance load across nodes in the system.

4.2 Load Sharing

4.2.1 Mechanics of Moving Operators

Several common implementation decisions have to be made in order to perform any kind of operator migration. First, we need some sort of protocol which specifies how to perform operator migration between two nodes. We must also develop a heuristic to determine when to move operators. Should this heuristic be based on global information or only local information? A related issue is whether we are moving a single operator to a more lightly loaded machine or whether we want both machines to operate in parallel.

A DSMS has to have a heuristic to decide when to move an operator. We define a *boundary operator* as an operator that lies directly attached to an input stream on a node or directly linked to an output stream of a node. For example, in figure 12(a), on machine 1 operators 1,2,4. One simple strategy is have each node ask its neighbors to take a boundary operator if the node is overloaded. This is *local* since it only uses neighborhood information and not information about the whole system of operators and load. This has the appeal that it is truly distributed but will rarely be optimal. The other strategy is to employ some planner, a node that collects statistics about all the nodes in the network, and then makes decisions to relocate operators accordingly. This is more centralized and introduces a single point of failure, but it is likely to yield better plans since it has more information available.

To move an operator, the DSMS has to perform the following steps. First two nodes have to agree to move the operator from one node to the other. The first node should start buffering the stream. Second the node should pack the operator, into some interchange format, such as XML in Aurora, then send it to the receiving node. If the operator has state then this can be more complicated; this issue will be discussed later. Next, the receiving node has to set up the operator, and then the sending node should start sending the contents of the buffer to be processed. The upstream node should be notified to start sending tuples to the receiving node instead of the original node. This step is needed if the operator occurs on a boundary. In figure 12 we show two different operator movement scenarios. One scenario is where you move a boundary operator to the adjacent machine, and the second is where you move an interior operator to another machine, which incurs higher bandwidth costs since you have to send queries to the machine and receive the results back from the same machine.

4.2.2 Stateful versus stateless operators

The moving of operators which maintain state requires additional consideration than in the stateless case. In the simplest situation, a state maintaining operator would be moved in its entirety to another node, requiring all the state to be transferred as well. The cost in terms of bandwidth and memory in moving this state must be factored into the rebalancing decision. A more complex situation would be the splitting of a stateful operator onto multiple nodes in the middle of a continuous query. This would require moving only a specific portion of the total state. A natural application of this would be for the computation of an aggregate in combination with a groupby clause. An example of this would be the computation of the average `price` of each stock grouped by `symbol`. To balance load, the average computation for a particular `symbol`, such as IBM, could be split off onto another node. This would require that the state associated with the average aggregate for that particular group be moved.

One current implementation of a special operator which allows for rebalancing of stateful operators is called Flux, Fault-tolerant Load-balancing eXchange. This operator provides data partitioning as well as buffering and reordering to deal with short-term, local imbalances and online partitioning of lookup-based operator state to handle long term, system-wide imbalances. Central to the functioning of the Flux operator is its repartitioning mechanism, which divides the state for each operator into numerous smaller partitions. This allows for finer control of rebalancing and state movement, helping to reduce the chance of thrashing.

4.2.3 Operator Splitting

To do load sharing we must run operations in parallel. In particular we need to be able to take an operator on one machine and split it into two copies with some means to divide the stream into two smaller streams to feed into the two operators. After this, we have to merge the output of

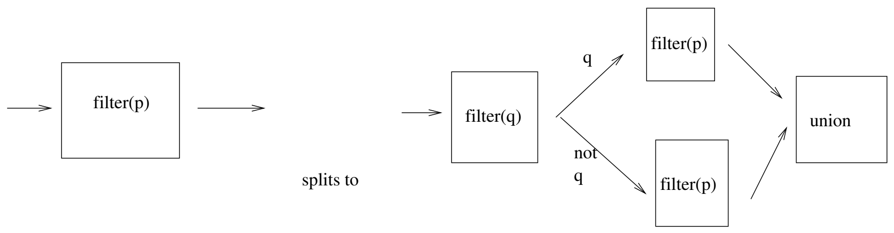


Figure 13: An example of a filter being split.

the two operators. This process as implemented in Aurora* is called *box splitting*. The difficulty in doing this lies in modifying the operator network without altering the intended meaning. This is further complicated if the operator has state associated with it. Ideally, the output of a pair of split operators when merged should be identical to the output of a single operator.

The first issue to address is how to split the stream into two smaller disjoint streams. It is a start to add a filter to output in one stream tuples that satisfy a predicate and on a second stream tuples that don't satisfy the predicate. What predicate to use is application specific. One predicate that might work is `TupleSequenceId % 2 == 0`, provided the system time-stamps or places sequence numbers on tuples. This predicate would be fine if each machine should be computing half query to by split. However, different machine loads and the upstream operators will affect the traffic pattern and data coming into this filter. Because of this, designing an optimal filter is non-trivial. The second issue is how to merge the two output streams of the split operator. For a simple filter, a union is sufficient. However for aggregates, more complicated operations have to be performed. Furthermore these merge operations vary depending on the operator we are splitting. The same issues of deciding when and where to move operators apply for box splitting as well.

4.2.4 Shared State

In a distributed environment, the sharing of data between operators introduces additional complexity. It is not uncommon for several operators to be working on the same input stream. In a single system environment it is possible to encapsulate the stream in a table-like structure and share the data between all operators which require it. When operators can be balanced among multiple nodes in a distributed system, the input tuples and state stored locally in one node may need to be accessed by other nodes. This could mean replicating the tuples or remotely accessing the tuples on other nodes as required, depending on resource management needs. This would require extensions of Aurora's shared queues and STREAM's synopses to deal with the complexity that this distributed environment introduces.

4.2.5 Bandwidth Considerations

When transferring any state between machines, bandwidth limitations must be taken into consideration. A node might be overloaded due to insufficient outgoing bandwidth. In this case, any remedy that requires significant bandwidth will only make the situation worse. Instead, we might want to wait for the load shedding mechanisms to kick in and resolve the situation. If the optimizer decides that state movement is necessary, the streaming engine will take up the bandwidth. Again, we rely on the load shedder to maintain reasonable latencies by getting rid of some traffic.

The bandwidth link between the individual nodes serves as a single union box. Multiple streams merge in and are spread out again once they cross the link. If we know the exact link capacity we can figure out the cost of sending each individual tuple, based on its size.

4.3 Power management and TinyDB

Consider a streaming sensor model such as the wireless sensors that soldiers wear. In this model, problems with power consumption arise. A remote wireless sensor network system such as this has limited power with which to perform its tasks, and is located where it can be sometimes infeasible to frequently change batteries. These sensors can have their power drained in a few days with constant use, but with smart techniques for power management, the sensors can last for months or even years. Berkeley's TinyDB[11] handles streaming data in a different way than previously discussed DBMSs. It utilizes acquisitional query processing (ACQP) in order to minimize power consumption.

TinyDB sensor nodes have four modes to their life: Snooze mode where power consumption is minimal and the sensor is just waiting for a timer or external event, processing mode, where query results are generated locally, processing and receiving mode, where results are collected from the node's neighbors, and transmitting mode, where the results from the query are delivered via the node's parent.

TinyDB has many concepts in common with DSMSs; it allows for user defined aggregates, landmark queries, and sliding window queries. It too disallows blocking operations except over windows. However, its four mode system gives rise to a different set of challenges.

Rather than having constantly streaming data, sensors will only append new tuples at sample intervals, or epochs, as in the following query.

```
SELECT nodeid, temp, heartrate
      FROM sensors
      SAMPLE INTERVAL 1s for 10s
```

This says that each sensor should report its own id, temperature, and heartrate readings once per second for 10 seconds.

4.3.1 Event-based Queries

TinyDB allows for event based queries, where the system starts processing a query, or ends a query when a certain event occurs. This is well within the bounds of microprocessor technology and allows a sensor to be dormant whenever it is not needed.

4.3.2 Lifetime-based Queries

Another feature of TinyDB stems from the desire to get as much information as possible while still requiring a certain sensor life, as in the following query, which specifies that the network should run for at least 30 days, sampling as quickly as possible while fulfilling that lifetime requirement. TinyDB performs lifetime estimations on a sensor network, based on knowledge of power consumption. Knowing how much time a node can be on and transmitting, the node will be in snooze mode as much as it needs to be to still be able to transmit at the end of the user required lifetime.

```
SELECT nodeid, accel
       FROM sensors
       LIFETIME 30 days
```

4.3.3 Semantic Routing Trees

To limit required processing time, sensors in the TinyDB system are ordered in semantic routing trees, based on semantic similarity rather than by the most reliable connection. To see exactly how a semantic routing tree is built, see [11]. Nodes are time-synchronized so that a parent will never be asleep while its child is is transmitting data.

4.3.4 Prioritizing Data Delivery

Whenever the network is ready to deliver a tuple, TinyDB desires that the result sent is the one that will most improve the "quality" of the answer that the user sees, an obvious correlation with previously mentioned Quality of Service. TinyDB considered three prioritization schemes, but decided on the "delta" scheme, where tuples are assigned scores and at each point in time, the tuple with the highest score is delivered. When the queue overflows, the tuple with the lowest score is dropped.

4.4 Global vs. Local

As in any distributed system, the load balancing policy may be either *global* or *local*. In a global policy, the load balancer uses the status of all available nodes. In a local policy, the nodes are partitioned into smaller groups, often consisting of a node and its group of nearest neighbors.

In calculating and implementing a redistribution of workload, factors to consider are: *cost of communication*: How much additional messaging is required to gather the necessary statistics? *cost of synchronization*: What is the delay necessary to synchronize the entire system? How many operations will be frozen in order to migrate to another node? *optimality*: How much can we increase the efficiency of the system?

Local Optimization. The local optimizer may shift operators between neighboring machines. If one machine is experiencing an overload, it can communicate with neighboring upstream or downstream sites and attempt to migrate operations. This is least costly in terms of communication and synchronization. However, if the immediate neighborhood does not have sufficient resources to resolve a bottleneck, the overload will persist.

Global Optimization. A global policy is able to consider the status of all nodes in the network to produce an optimal re-distribution of work. The cost of communication with every machine in the system, however, can be very high. Additionally, a significant migration of operators causes a greater delay in implementing a new policy.

In practice, a system may employ several load balancers, resorting to the more costly global balancing only when a bottleneck cannot be resolved though local balancing.

4.5 QoS management

QoS management in distributed systems provides several challenges for the DSMS. QoS is generally specified with respect to the output streams, since they are the ones that the user is receiving.

Many of the operators are either irreversible or non-deterministic and thus reasoning about QoS throughput the query network is difficult [13]. For example, a change applied to the input stream can be reversed or cancelled by downstream operators.

Reasoning about QoS on a single node is difficult, and in distributed system it becomes a lot more complicated. In a distributed system QoS is still specified only at the output. Thus intermediate nodes have no QoS specifications at all. Therefore, in addition to trying to *push* QoS through to the input within a single node, we must also provide the derived QoS to the upstream nodes as a guideline. Every such derivation will introduce uncertainties when performed.

There are several different QoS measurements that user might care about: latency, throughput and value-based. These measurements present increasing challenges to the DSMS. Latency is an additive measurement. A change in latency in any of the nodes will be most likely reflected by the same change in the overall latency. Throughput can be reasoned about in the same way as latency, but changes in different places in the network will have different effect. Any throughput change will cascade all the way through to the output. Finally, value-based changes will have unpredictable effects downstream depending on the operators. There is no easy way to tell what the result from a value change will be on a downstream node.

4.6 Reliability

4.6.1 Correctness Criteria

Because of the time-critical nature of most streaming data applications, reliability is usually a key concern. This is especially true for a distributed stream manager, where any computing node or inter-node network path can be a point of failure.

There are two considerations for reliability. The first is correctness. For a distributed stream manager, this means that the system must not allow any hardware failure or latency to cause incorrect answers to be returned. Unfortunately, this requirement is difficult to define precisely because it's not always clear whether a particular answer is correct or not. The second consideration is availability. Ensuring high availability means maximizing the percentage of time that the system is operational.

In the world of traditional databases, correctness is usually defined in terms of transactions. A transaction is a sequence of update operations that the user wants to treat as a single operation. It is the responsibility of the DBMS to make it appear to the user that each transaction is atomic, meaning that the effects of the transaction are either applied or they aren't.

Using the transactional model, the correctness criteria for a DBMS can be defined as follows. The results of any query executed by the DBMS must not show the effects of any uncommitted transaction. Conversely, any query executed by the DBMS must reflect the effects of all transactions that been committed so far. Furthermore, the effects of committed transactions must be the same as if the transactions had been executed in some serial order.

Unfortunately, the notion of transactions can not be easily adapted to the world of streaming data systems. One problem is that the semantics of the input data are different. With a traditional database application, it typically makes sense for the user to group together a sequence of operations to form a single operation. With a streaming data application, however, this semantic model usually doesn't make sense. For example, consider an application whose input stream is a sequence of stock prices. There isn't really a natural way to group together multiple stock prices into a single unit. An alternative would be to treat each tuple as its own transaction. However, this approach creates its own set of problems. If each input tuple is a transaction, the stream manager could process the tuples in any order without violating the serializability constraint imposed by the transactional model. Since the order of tuples in an input stream is usually significant, this approach does not impose a sufficiently strong correctness constraint.

Another problem with using transactions in a streaming data application is that the queries must operate on data that is constantly changing. In a traditional database application, the underlying data used by a query does not change while the query is running. However, since the queries in a stream application run indefinitely, the system must be able to process updates while the query is still running. For this reason, the transactional model can not be used in a streaming data application.

In general, a major obstacle to defining correctness criteria for a streaming data application is the temporal variability of the input. In a transactional DBMS, the system guarantees only that the transactions will be executed in some serial order. This is generally sufficient, since relatively few orderings are possible. For example, if two transactions are submitted at about the same time, only two orderings are possible — either A and then B, or else B and then A.

In a streaming data application, however, there are many tuples entering the system on many different streams, with no assurance that related tuples will arrive at the same time. This causes many stream operators to be non-deterministic. One example is the "union" or "merge" operator, which outputs any tuple that is received on any of its input streams. The order of tuples outputted by this operator depends entirely on the arrival times of the tuples on the input streams. Because the operators are non-deterministic, it is difficult to define what the correct answer to a stream query should be.

The problem of non-determinism is further complicated by the fact that an application's QoS constraints usually require that query results be returned in a timely manner. These constraints can cause load shedding to occur, which in turn can change the answers to queries. Because of this, the correctness criteria must allow for the possibility of approximate answers.

4.6.2 High Availability

Traditionally, high availability and fault tolerance has been achieved with a process-pairs model. A redundant server is kept synchronized, and in the event of a failure, the second server can continue from the last synchronization point. However, given the non ACID nature of some of these systems, one may be able to take advantage of a simpler recovery scheme.

The simplest of these, which requires no synchronization, is to follow a policy of amnesia. For systems in which older values are not important, such as many sensor systems, one doesn't need to worry so much about reconstructing state information that was on the lost server. For some applications, it is sufficient if eventually correct values are generated within a bounded amount of time after the failure.

Alternatively, one can take advantage of the streaming nature of these systems. In Aurora, for example, queries can distribute across multiple servers, each working on a piece of the stream that is then feed as a new stream into a subsequent server. By buffering data as it is streamed to the latter server, one can utilize that information to resume seamlessly. If the downstream server fails, the operations on that server can be migrated to a new one, and the buffered tuples replayed to reconstruct any lost state.

In a similar way, TelegraphCQ achieves fault tolerance through the additions of "Flux" operators between servers. Flux has two responsibilities, load balancing and fault recovery. It is a partitioning dataflow operator which includes a state movement protocol. For both migrating load, and recovery of a fault, a flux operator can migrate the state at a server to another. In the case of a failure, the partitioning will be reconfigured such that all data for the failed node is redirected to one that is still available.

4.6.3 Reliability and Recovery

Fortunately, there has already been a significant amount of work in analyzing the reliability of the cousin of SDMSs, Main Memory Databases (MMDBs). MMDBs, sometimes referred to as real-time systems, operate the exact same way as Disk Resident Databases (DRDBs), i.e. the standard database management system, but because one of the qualifications of an MMDB is that most or all of the database reside in main memory, it is able to achieve significantly higher throughput. MMDBs therefore suffer from the same problems that SDMSs do, which is that most of the data is held in volatile storage and is therefore susceptible to loss in a way that data held in stable storage systems is not.

There are three elements in providing a reliable system in the event of failure: logging, checkpointing, and recovery. Logging commits the workings of the database to disk. Checkpointing periodically stores parts of the database to stable storage. Recovery is the procedure to use the log and the checkpoint to bring the database back up online. For MMDBs, most of the recovery schemes are built around transactions, and there would need to be significant conceptual reworking of the theory to apply to data streaming. Practically, the feasibility of such a modification depends very much on the specifics of an SDMS system. If the throughput rate is incredibly high, the overhead induced by all forms of reliability control might become too significant. MMDBs have a very tightly integrated interface with persistent storage due to their transactional nature; more research is needed into the interface between SDMSs and persistent storage before it can be concluded whether the techniques for MMDBs can be safely copied to SDMSs.

References

[1] Daniel J. Abadi, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2), August 2003.

[2] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: Semantic foundations and query execution. Technical report, Stanford University, October 2003.

[3] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Dilys Thomas. Operator scheduling in data stream systems. Technical report, Stanford University, October 2003.

[4] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. Exploiting k-constraints to reduce memory overhead in continous queries over data streams. *ACM Transactions on Database Systems*, September 2004.

[5] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Operator Scheduling in a Data Stream Manager. In *VLDB Conference*, Berlin, Germany, September 2003.

[6] S. Chandrasekaran, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR Conference*, January 2003.

[7] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Ying Xing, and Stan Zdonik. Scalable Distributed Stream Processing. In *CIDR Conference*, Asilomar, CA, Jan 2003.

[8] Personal Communication with Don Carney, May 3, 2004.

[9] The STREAM Group. Stream: The stanford stream data manager. *IEEE Data Engineering Bulletin*, 26(1), 2003.

[10] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Uğur Çetintemel, Michael Stonebraker, and Stan Zdonik. A Comparison of Stream-Oriented High-Availability Algorithms. Technical Report CS-03-17, Department of Computer Science, Brown University, October 2003.

[11] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. The Design of an Acquisitional Query Processor for Sensor Networks. In *ACM SIGMOD Conference*, June 2003.

[12] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. Using state modules for adaptive query processing. Technical Report UCB/CSD-03-1231, University of California, February 2003.

[13] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load Shedding in a Data Stream Manager. In *VLDB Conference*, Berlin, Germany, September 2003.

[14] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, 2003.