Static Optimization of Conjunctive Queries with Sliding Windows over Unbounded Streaming Information Sources

Ahmed M. Ayad Jeffrey F. Naughton
University of Wisconsin-Madison
Computer Sciences Department
{ahmed, naughton}@cs.wisc.edu

Abstract

We study the problem of static optimization of conjunctive queries with sliding window joins over unbounded streaming information sources. While previous work has suggested focusing on maximizing the output rate of queries over streaming information sources, we show that in steady-state, for conjunctive queries with sliding windows over unbounded streams, all feasible plans have the same output rate. For this reason, we suggest that the goal of optimization for such queries should be to minimize resource utilization in steady state. We formulate a cost model for the problem and introduce a framework for optimization based on feasibility and resource management. We then use the model to study where to optimally place random drop boxes and how much to drop in each box in case no feasible plan exists and approximation is necessary. The goal of the approximation is to achieve the maximum throughput of the resulting tuples given the resource constraints. We find that, given a plan to approximate, the optimum placement and values are easily obtained. However, finding the best plan to approximate is not necessarily an obvious task.

1. Introduction

The focus of research on data and information processing has recently shifted towards an emerging type of applications in which the data is streaming from its sources. Such applications include monitoring network traffic, intrusion detection, telecommunications, sensor networks, financial services, and e-business applications.

Some major assumptions made by traditional data management systems do not hold in the context of streaming applications. In these applications, the system has no control over the arrival time of the data. Hence, the adoption of a push model of computation is mandatory. Also, in such applications, monitoring queries can run for a long time (e.g., on the order of days or months) that they can be assumed for all practical purposes to be running continuously, hence the name *continuous queries*.

An important goal in systems designed for such applications is to provide an easy framework for users to express their queries. A good approach is to provide users with a declarative method to do so, leaving the decision on arranging how the query is executed to the system. Such approach is taken by the STREAM [33] team which extended the SQL query language with constructs to pose queries on any combination of relations and continuous streaming sources [2]. This approach reopens the problem of query optimization for continuous queries.

The goal of query optimization has always matched the framework in which the queries are executed. Query completion time was the goal in early efforts of query optimization for traditional relational database systems [28]. Then it became the fastest response [34], or an approximate response with statistical guarantees [18]. In the case of streaming sources, the goal differs depending on the type of application and the query posed. In terms of the data sources, it can either be: a) finite streaming sources (e.g., documents flowing over the internet) or b) infinite streaming sources (e.g., call logs, network traffic logs, sensor readings.) In terms of the posed query, it can either be: a) short running (i.e., the user is interested in asking about the current state of the system over a short period of time, or is interested in a small prefix of the answer,) or b) continuous, meaning the user would like to monitor the system's behavior for a considerable length of time. Using the previous classification, Figure 1 highlights the different combinations and the goal of optimization in each.

One combination does not make sense, since if the streaming source is finite, a continuous query degenerates into a short running one. In case a query is short running, the best plan for the query is the one giving the fastest response of the whole answer (or a specific prefix of it,) or the one giving the most answers in the specified lifetime of the query. The work in [35] presented a framework for optimization of such queries.

	Stream	
Query	Finite	Infinite
Short running	Throughput / Response time	Throughput / Response time
Continuous		Feasibility / Resource Management

Figure 1. Goal of query optimization for streaming information sources

The fourth quadrant, in which the sources are infinite and the query is continuous, is the one called continuous queries. Since the execution engine has no control over data availability, it has no choice but to keep up. From basic queuing theory [23], if the system capacity exceeds the requirements for the input rate (utilization < 100%,) the system is stable. Otherwise, the system is said to be saturated or unstable. In the context of continuous queries, an execution plan for the query is feasible if the system it will execute on will be stable. A feasible query is one for which at least one feasible plan exists.

With this setting in mind, three scenarios are possible for any given query:

1. Resources are abundant, which means that all plans will be feasible. However, a poor choice can still tie-up unnecessary resources.
2. Resources are tight, meaning the choice of a good execution plan can prove to be essential for feasibility.
3. Resources are insufficient, meaning that the query is infeasible and an approximate result is inevitable. The choice of plan that carefully uses available resources can improve the quality of such an approximation.

To arrive at the best decision at each situation, we present a framework for static optimization of continuous queries based on feasibility and resource management. In particular, our main contributions are:

- We develop a model for estimating the resource utilization of an execution plan of a continuous query.
- We use the model to show that for all feasible plans, the output rate will always be the same, making output rate or throughput a non-distinguishing factor. This is why the goal of optimization in such case should rather be feasibility and resource management.
- We then introduce our framework for optimization based on the above goal. The optimizer in this framework should decide on the best plan for situations (1) and (2) above, or state that no feasible plan exists.
- In case no feasible plan exists, situation (3), we use the model to examine the problem of finding an approximate answer by randomly dropping tuples from the query plan. The goal of the approximation is to achieve the least amount of lost tuples from the final answer of the query.

Table 1. Variables used in estimating resource requirements.

C_σ	Cost of performing a selection on a single tuple
C_P	Cost to probe an active window for a matching tuple just arriving
C_I	Cost to insert an arriving tuple into the sliding window
C_V	Cost to invalidate an expired tuple from the sliding window
σ	Selectivity factor of a selection predicate
f	Join selectivity factor
λ_i	Rate of arrival of tuples from source i
W	Size of a tuple-based window
T	Size of a time-base window
M	Least amount of memory needed for an operator/query

- We show that interestingly, when approximation is necessary, the best plan to choose for approximation is not necessarily the best feasible plan.

Much of the recent work on systems for streaming information sources is built on being able to dynamically adapt to the changing characteristics of the data as it flows by. The paradigm is: start with a plan, and then continuously change it as you know more about the data. Examples are the work in [3][19][32]. This is built on the earlier idea of mid-query re-optimization [21]. It is important to note that, by introducing a static optimization framework, we are not effectively stating that it is a better way to approach the problem. Static optimizations can be useful in cases where the rates of the input streams are slow changing, and the pattern of change is predictable (e.g., network/transportation traffic loads, building sensors.) It suffers from its rigidity and inadaptability to rapid changes of basic assumptions about the data characteristics. The adaptive approach solves these problems, but it is not without its overhead. The question of which is better depends upon several things, including the exact amount of overhead, and how volatile the environment is. At one extreme, very static environment, static optimization will be best. At the other extreme, very dynamic environment, adaptive may be superior. In between the two are a number of tradeoffs (e.g., optimize and monitor then re-optimize when necessary, or optimize every k number of seconds.) Our goal is not to answer the question of which is better or when to use which. To be able to answer such questions, we need first to know what it means to do static optimization for continuous queries, which is the goal of this paper.

The rest of the paper is organized as follows: Section 2 discusses related work in the literature. Section 3 describes the cost model used in the optimization problem. Section 4 defines the problem and discusses related issues. Section 5 tackles the problem of how to arrive at a good approximation of a query if no feasible plan exists. Section 6 concludes the paper.

2. Related Work

The existence of applications built on streaming information motivated building specialized systems to manage streaming data. Among the recent examples are: Niagara [11], which queries streaming XML sources over the internet; STREAM [33], which has as its goal extending traditional DBMS technology to also manage streaming sources; Aurora [8], which is specifically designed to accommodate a large number of continuous queries; Telegraph [32], which is a highly adaptive dataflow management system. The survey in [4] contains a good documentation of earlier models and systems that are also targeted at such applications, together with a number of issues related to building a data stream management system. Another closely related type of system is sensor networks and databases, examples of which are TinyDB [24][25] and the Cougar [38] projects.

The problem of query optimization is almost as old as relational databases. The seminal work of [28] introduced a framework for optimization of relational queries aimed at minimizing query completion time. In the context of continuous queries, such a goal is inappropriate. NiagaraCQ [10][11] aims at addressing the scalability of a system supporting a large number of continuous queries by grouping predicates and queries together. The work in [9][12][26] uses similar techniques by extending the earlier work on eddies [3] to support multiple concurrent continuous queries. The difference between this body of work and ours is that they are all dynamic optimization methods that adapt at run time to changing data and query characteristics; they do not deal with static optimization.

The Aurora system [8] treats multiple streaming sources and multiple output queries as data flows between operators (boxes) that are input by the user. The queries in Aurora are composed by the user through an interface, then the system manages them without modification. This is the equivalent of adapting to an already given static plan of multiple queries with shared resources. Similarly, the work on scheduling operators in [5][17] deals with scheduling operators of a static plan to minimize resource usage or response time. Different problems related to scheduling and static resource allocation are reported in [27] together with a brief discussion of solutions. The assumption in such work is that a query optimizer has already arrived at a “best” plan.

Close to our work is that presented in [35] and [36]. The former advocates moving from cardinality-based optimization to rate-based optimization and provides a model for a rate-based optimizer. Such work is geared towards queries in the first and second quadrants of Figure 1. It does not model the effect of sliding windows for continuous queries over infinite sources. The later provides a symmetric multi-join operator for multiple joined streams to minimize memory usage as opposed to using multiple binary join operators. Although it does not model sliding windows, it can be easily adapted and incorporated as an option for the optimization framework provided here. Also close is [30] in which the authors provide a queuing model for distributed eddies. One interesting result provided is that sometimes no single plan is the best if the goal is to achieve the maximum input rate before the system saturates. A combination of plans running concurrently, each with some share of the input load is proven to be better. The subtle difference between this work and ours is that this work assumes the operators are running on different processors, hence each has its fixed resources. Our work assumes all operators share a pool of resources. In this case, one plan is always better, the one our framework optimizes for. An interesting direction would be to look at how an optimum plan can be distributed over multiple processors.

A lot of work dealt with providing approximate answers to continuous queries. In [27], the authors survey a number of methods to arrive at an approximate answer, among which is random sampling (i.e. random dropping of tuples) discussed here. In the context of Aurora, the authors in [29] provide algorithms for placing drop filters to reduce resource usage. They explore both random and semantic filtering. The difference between this work and ours is that they don not explore the effect of modifying the query plan to achieve better approximation. Plus, they deal with multiple queries, while we only consider single query optimization. Extending our work to multi-query optimization is an interesting direction. The work in [22] discusses single join approximation using random drops in case of either memory or computational resource shortages or both. This work extends that by studying the problem of insufficient computational resources for multiple joins. Also close to our work is [15], in which the authors study the problem of maximizing the result size of a single sliding window join in case of memory constraints by smartly selecting tuples to drop (semantic load shedding [8].) There is a brief discussion about extending the work to multiple joins and to deal with resource constraints. In this work, we deal with computational resource constraints, and multiple window joins. A comparison between our technique extended to handle smart load shedding and theirs after extension to multiple joins and resource constraints is another interesting direction.

Another related area is developing compact statistical structures or synopsis for estimating different characteristics about the input streams (e.g., selectivities, running aggregates, … etc.) Examples of such work are [6][7][13][14][16]. Such techniques are essential for the success of an optimizer built on the framework presented here.

Finally, theoretical work examining different aspects of data stream systems exist. In [1], the authors examine the memory requirements for different types of conjunctive continuous queries with arithmetic comparison, and without window semantics. They find some instances of such queries to require only bounded memory. It is worth noting that conjunctive queries with only equality joins require unbounded memory and that is why window semantics are essential. The work in [2] is a first attempt at developing a SQL like streaming query language.

3. The Cost Model

In this section, we provide the necessary calculations to estimate the expected processing and memory constraints for providing an answer for continuous queries. First, we derive the necessary equations to estimate the output rate and active window sizes for different operators assuming there are no constraints (i.e., assuming the plan is feasible.) Then, we move on to estimate processing and memory requirements for these operators together with the constraints on such requirements.

We assume steady state conditions and use the average rate to characterize the rate of arrivals of incoming tuples from external sources. This implicitly assumes a stable arrival rate. Table 1 defines the notation used throughout the paper. All costs are in time units.

3.1 Window Predicates

Before delving into the derivations of the cost model, a brief discussion on window predicates is warranted. Besides their semantic usages, window predicates are a means to restrict an infinite stream for operations like stream joins to become feasible. Many types of window semantics exist, each has its own modeling requirements. A discussion of the different types can be found in [2][8][9]. For the purpose of this paper, we will only consider tuple-based and time-based sliding windows as described in [2]. We will develop the cost model for the tuple-based type. However, since we are concerned with steady state conditions and are using average rate, it is easy to adapt the model for time-based windows using the following argument. On average, the number of active tuples in a window *i* of size *T* is $\lambda_i \cdot T$. So, by replacing the size W_i of a tuple-based window with $\lambda_i \cdot T$, the equations will be applicable to time-based windows as well.

3.2 Rate and Window Calculations

3.2.1 Selections and Projections

We will consider projections as a special case of selections in which the selectivity factor is equal to 1. The number of tuples a selection/projection operator handles in a unit time is λ_i. Of those, only $f \cdot \lambda_i$ qualify for the selection. Hence, the output rate is

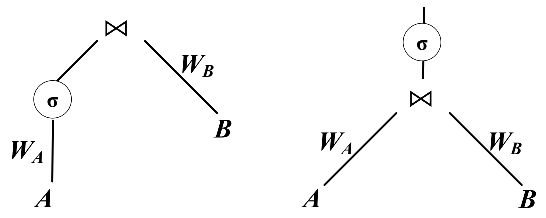


Figure 2. Two plans for the same query. The window size on the left side of the join is less in the left plan.

$$\lambda_o = f \cdot \lambda_i \qquad (1)$$

Since selections and projections are treated as filters on incoming streams, there are no buffer requirements above what is necessary to store and inspect the current tuple.

If the input stream to the selection or projection operator has a window predicate defined on it and then the output of the selection operator is fed into a join operator, we need to estimate the output window size resulting from the selection. To see the reason for this, consider a continuous query on two streams *A* and *B*. The query is composed of a selection on *A* followed by a sliding window join on the two streams with W_A and W_B being the values of the sliding windows defined on the two streams respectively. Two plans for the query are possible (Figure 2.) The window size on the left side of the join in the two plans cannot be equal, otherwise, for the left plan, we would be joining tuples from *B* with the last W_A tuples that have passed the selection on *A*, instead of the last W_A tuples that arrived (as done in the right plan), and the two plans will not produce the same results. To resolve this, the input window size should be reduced by a factor matching the selective capability of the selection operator.

Hence, for a selection operator with a window W_i defined on its input, the size of the output window is

$$W_o = f \cdot W_i \qquad (2)$$

3.2.2 Joins and Cartesian Products

Similar to the previous section, a Cartesian product can be viewed as a special case of a join with the selectivity factor equal to 1. We assume a deterministic timestamp ordering (e.g., arrival time) on the tuples from external sources.

Consider sliding windows of sizes W_L and W_R tuples placed on the left and right inputs of the join respectively. The number of tuples arriving from the left-hand side of the operator in a unit time is equal to λ_L, each of which is expected to join with $f \cdot W_R$ tuples from the right-hand side window. Hence, the number of tuples produced as a result of tuples arriving from the left-hand side is $f \cdot W_R \cdot \lambda_L$ per unit time. Similarly, the number of tuples resulting from right-hand side arrivals is $f \cdot W_L \cdot \lambda_R$. So, the total output rate for a window join is

$$\lambda_o = f\,(W_R \cdot \lambda_L + W_L \cdot \lambda_R) \qquad (3)$$

Similar to the filter-only case above, if the output of the join is fed into another join operator, we need to calculate the size of the resulting window W_o to use it as an input value for the other join. Any joined tuple is considered valid (not expired) only if all the original tuples it is comprised from are still valid. Consider arrivals on the left side of the join. Each arriving tuple that is inserted into the window on the left side causes the earliest tuple in the window to expire. The arriving tuple produces an average of $f \cdot W_L$ tuples, while the same number of tuples resulting from the expired tuple becomes invalidated. The same scenario occurs for arrivals on the right hand side. So, on average, the number of resulting active tuples stays the same, which is the expected size of joining the active tuples on the left hand side with the ones on the right hand side. So

$$W_o = f \cdot W_L \cdot W_R \tag{4}$$

3.2.3 The general case

The above equations are all derived for binary joins. Using these derivations, it is possible to generalize them for the case of n-ary joins. In doing so, we arrive at the following observation.

Observation 1

The output rate of an n-ary join of *n* streams is constant and is estimated by

$$\lambda_o^n = \prod_{\substack{all \\ selectivities}} f \cdot \sum_{k=1}^{n} \left(\lambda_k \cdot \prod_{\substack{i=1 \\ i \neq k}}^{n} W_i \right) \tag{5}$$

where λ_k is the arrival rate of stream *k*, and W_i is the size of the tuple-based window predicate on stream *i*.

The size of the resulting active window for an n-ary join can also be estimated by

$$W_o^n = \prod_{\substack{all \\ selectivities}} f \cdot \prod_{i=1}^{n} W_i \tag{6}$$

Proof

The proof is simply by induction on the number of streams involved in the join and using equations (3) and (4) for the base case. ❑

It is clear from the above that the final output rate and active window size resulting from joining *n* streams are independent of how the join operation is performed. This is intuitively equivalent to the fact that, for a traditional relational query, the size of the final result is independent of the execution plan.

The previous observation, coupled with the equations in Section 3.2.1, suggest that the output rate of a conjunctive continuous query is independent of the execution plan and that it should not be the goal of query optimization.

3.3 Processing and Memory Constraints

The problem of optimizing a conjunctive continuous query is similar to the traditional problem of optimizing conjunctive queries in relational databases. In both problems, we need to search the space of query plans to find the minimum cost tree. The current problem differs, however, in that it is a constrained one. In traditional query optimization, even the worst plan is supposed to be feasible given enough time. When dealing with continuous queries, if the system cannot handle the load, the plan is simply infeasible and resorting to approximations is inevitable. To correctly qualify the feasibility of a plan, we need to derive the necessary requirements for the different types of operators, which we do in the current section.

3.3.1 Selections and Projections

The cost of handling a tuple for a selection or a projection operator, C_σ, includes reading, inspecting the condition, and writing out the result, if necessary. For a selection or a projection operator to be able to correctly handle an arriving tuple, C_σ must be, on average, less than the average time until the next arrival. Hence, the following constraint

$$C_\sigma \cdot \lambda_i < 1 \tag{7}$$

There are no memory constraints on selection and projection operators since no buffers are required.

3.3.2 Joins and Cartesian Products

In the case of joining infinite streams, only non-blocking algorithms can be used, like the symmetric hash join [37]. Kang et al. made the observation in [22] that the join cost can be divided into the cost of performing the left and the right parts of the join, and that the method of performing the two parts are completely independent. They derived a general cost model for the sliding window join which we will use here. The cost of the join per unit time is

$$\begin{aligned} C_L &= \lambda_R \cdot C_P(L) + \lambda_L \cdot (C_I(R) + C_V(R)) \\ C_R &= \lambda_L \cdot C_P(R) + \lambda_R \cdot (C_I(L) + C_V(L)) \\ C_{L \bowtie R} &= C_L + C_R \end{aligned} \tag{8}$$

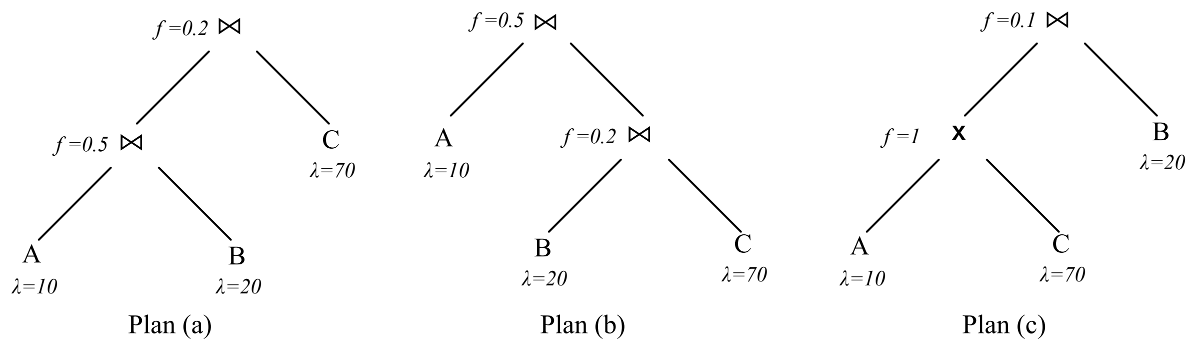
The previous calculations are necessary if asymmetric operators will be used on the left and right side of the join. If, on the other hand, the traditional symmetric operator is used, the cost functions can be simplified to

$$C_{L \bowtie R} = (\lambda_R + \lambda_L) \cdot (C_I + C_V + C_P) \tag{9}$$

In both cases, the constraint is

$$C_{L \bowtie R} < 1 \tag{10}$$

In the later case, the operator can be seen as having an arrival rate of $(\lambda_R + \lambda_L)$ and a service rate of $(C_L + C_R + C_P)$, analogous to equation (7).



$\lambda_{A\bowtie B}$= 0.5*10*(10+20) = 150 $\quad$ $\lambda_{B\bowtie C}$= 0.2*10*(20+70) = 180 $\quad$ $\lambda_{A\bowtie C}$= 1*10*(10+70) = 800

$W_{A\bowtie B}$= 0.5*10*10 = 50 $\quad$ $W_{B\bowtie C}$= 0.2*10*10 = 20 $\quad$ $W_{A\bowtie C}$= 1*10*10 = 100

Figure 3. Possible plans to evaluate the join.

It is worth mentioning that the cost of the join is dependent on the join algorithm used. The model presented in [22] can be used to choose the best possible algorithm for each join.

In terms of memory requirements, the join operator needs at least enough memory to hold the active tuples on both sides of the join[1], so the minimum memory requirement is

$$M = W_L + W_R \tag{11}$$

3.3.3 Notes on Processing and Memory Constraints

The constraints derived in this section have the subtle assumption that the operator will be the only running process in the system. In case a host of operators are sharing processing resources, the previous bounds are not tight. For the constraints to become tight in this case, the cost values of each operator should be dilated by the inverse of the fraction of time the operator is scheduled to run on the system. For example, if it takes 1 millisecond to process a tuple for selection, but the operator is sharing the processor fairly with 9 other operators, then the cost should increase ten fold to 10 milliseconds.

Another note on the memory constraints is that the ones developed here do not take into account the queuing requirements for each operator. The estimation of such constraints requires a queuing model of the execution plan, which is part of our future work.

Example 1

To motivate the classification we had in the introduction and the cost model computations, consider the following simple SQL-like query (the window constraint syntax is modeled after [27]):

```
SELECT A.a, B.b, C.c
FROM   A [ROWS 10]
       B [ROWS 10]
       C [ROWS 10]
WHERE  A.a = B.a
AND B.b = C.b
```

This is a simple three-way tuple-based window join between the streams A, B, and C with the window being the latest 10 rows in each stream. Assume 0.5 is the selectivity of A⋈B and 0.2 is the selectivity of B⋈C. Also assume that 10, 20, and 70 are the rates of arrival of streams A, B, and C respectively in tuples/second. Further assume, for ease of exposition, that any join operator takes a constant amount of time to handle an incoming tuple from either side of the join, and that memory is measured in tuples. Figure 3 shows the possible plans to evaluate the query.

First, assume that a join operator takes 0.5 milliseconds to join an incoming tuple, which means that the system can handle at most 2000 tuples/second. In this case, it is obvious that any of the plans is feasible. In terms of memory consumption, all three plans require three queues capable of holding 10 tuples each for every stream. The plans differ dramatically, however, in terms of the active window size for the intermediate result that is input to the second join. Buffering requirements for plans (a), (b), and (c) are 80, 50, and 130 tuples respectively. A poor choice (plan (c) in this case) can result is in an expected 160% increase in memory consumption. The plans also differ dramatically it terms of their resource utilization. While plan (c) keeps the system 50% utilized, plan (b) has only 14% utilization, and plan (a) has 12.5% utilization. Choosing plan (c) results in a 400% increase in the necessary resources to answer the query.

Now, assume that a join requires 4 milliseconds to handle an incoming tuple, meaning that the system capacity is 250 tuples/second. In this case, the tuple

[1] The amount of memory needed depends on the join algorithm. In case of a symmetric hash join, for example, the size of the hash table may slightly exceed the size of the window.

arrival rate should be taken into consideration. A join operator should be able to handle an arrival rate at least equal to the sum of the arrival rates of its input streams. Since all operators are run on the same system, the sum of the service rates of all operators cannot exceed the total system capacity. To test whether a plan is feasible, we need to sum the minimum service rates required for all joins and compare this to the system capacity. For plan (a), the first join must handle at least 30 tup/sec., and the second join 220 tup/sec resulting in total system requirement of 250 tup/sec. Similarly, plan (b) requires 280 tup/sec, and plan (c) requires 900 tup/sec. This means that plan (a) is the only feasible solution and choosing either (b) or (c) requires the system to resort to approximation unnecessarily.

If a join requires 5 milliseconds per incoming tuple (i.e., maximum system capacity of 200 tuples/second,) all plans become infeasible and some sort of approximation must take place. One way to approximate the result of a query is to randomly drop tuples from the input queues of the different operators. A heuristic measure of the quality of approximation can be the final plan throughput; the plan that drops the least number of tuples might be the best choice (the *MAX-subset* measure in [15]). We later discuss (in Section 5) some of the issues related to how to arrive at such a plan.

3.4 Discussion

In the previous sections we have proved that all *feasible* plans of a continuous query have the same output rate. This does not mean that all feasible plans produce the same result at the same time. To understand this, it may be helpful to regard a query execution plan as being analogous to an open queuing system, with the input being the data streams and output being the query answer. From queuing theory, the throughput of any two stable systems seeing the same input is fixed and is equal to the input throughput (for separable networks [23],) while the utilization and response times may vary between the two depending on the characteristics of each. In our context, the response time of a result tuple is the time difference between the production time of the tuple and the arrival time of the latest input tuple involved in generating it. The response time of a plan is the average response time of all resulting tuples. Feasible plans differ in their response times, meaning that they produce the same result tuples with each shifted in time by an average amount equal to the average response time. The response time of a plan can increase arbitrarily which is not desirable for streaming applications with real-time requirements.

In the next section, we formalize the optimization problem to use the model above for obtaining the plan with the least resource usage (i.e., utilization.) Although the model does not directly model the response time of the plan, it can be easily argued that the plan produced will be the one with the least response time. From queuing theory, to use the analogy again, utilization is directly proportional to response time (i.e., the system with the least utilization has the least response time.)

4. The Optimization Problem Definition

Definition 1: Execution Plans. Given a certain conjunctive query Q on streaming sources, an execution plan p is a tree, whose leaves represent streaming sources, and internal nodes represent query operators each of which is a selection; a projection; or a join. The plan p represents a possible execution order of the operations in Q. ❑

We are now ready to formulate the optimization problem of conjunctive queries over infinite streams. Depending on the situation, we may choose to optimize for processing or memory resources. Two cost functions, $C(p)$ and $M(p)$, can be associated with each execution plan, defined as follows

$$C(p) = \sum_{all\ operators\ i\ in\ p} c(i) \tag{12}$$

$$M(p) = \sum_{all\ operators\ i\ in\ p} m(i) \tag{13}$$

where $c(i)$ and $m(i)$ are the processing per unit time and memory cost, respectively, of operator i. The objective of optimization is to

$$\textbf{Min}\ \{C(p) \mid p \text{ is an execution plan for } Q\} \tag{14}$$

subject to

$$C(p) < 1 \tag{15}$$

when optimizing for processing resources, or

$$\textbf{Min}\ \{M(p) \mid p \text{ is an execution plan for } Q\} \tag{16}$$

subject to

$$M(p) < \mathcal{M}, \text{ where } \mathcal{M} \text{ is the available memory budget} \tag{17}$$

when optimizing for memory resources.

Equation (15) needs some further explanation. Since the cost function of each operator in the execution plan is defined in terms of the cost required per unit time, and since the assumption is that all the operators will be competing for the same computational resources, the whole plan is feasible only if the total cost of all operators per unit time is less than one unit time (i.e., utilization < 100%.) Figure 4 shows an example of a simple plan with two selections. Assuming the time unit is a second, the first selection takes half a millisecond to process a tuple, the second takes a millisecond. In the first alternative (plan A,) it takes 500 milliseconds – on average – for selection σ_1 to take care of arrivals in one second, and 250 milliseconds for σ_2. This means that there is enough time for both operators to handle the load coming their way in a unit time. Hence, both operators can share the same processing resources and the plan is feasible. Plan B, on the other hands, dictates that σ_1 needs 250 milliseconds,

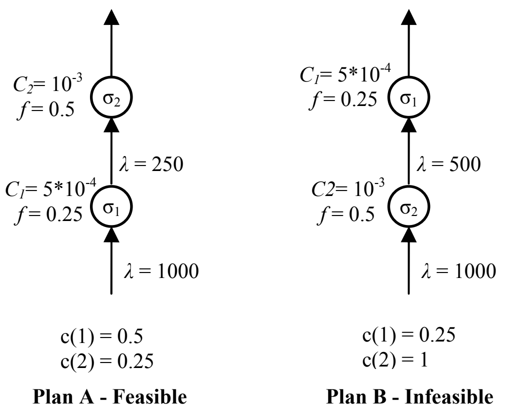


Figure 4. Feasible and infeasible alternatives.

and σ_2 needs one second to handle the arrivals in a unit time. Hence, it is infeasible and should be discarded.

We end the section with a discussion on several related issues:

The Single Processor Assumption. The above assumption, that the whole plan will be executed on one machine, is not necessary for the correctness of the model. Sometimes it is desirable to allow for some *head room* for the operators to avoid congestion in case of burst arrivals [29], other times, the plan can be distributed on more than one processor. The model can easily accommodate both cases. In the first case, if it is desired to leave 20% of the resources as head room, the bound of equation (15) should be 0.8 instead of 1. In the later case, if *n* processors are available, the bound should be *n*.

Querying Relational Tables. A continuous query can combine relational tables with continuous streams [2]. Our model can easily integrate this scenario. Two cases arise. The first is when an operator, or a sequence of operators in a plan operate only on relational tables. In such case, the optimizer will treat this sub-query in a traditional way coming up with the best possible arrangement of the operators and estimating the result size. The option of materializing the result in memory or on disk should then be taken into account. The computational cost of the sub-query, however, is zero since it should be only a one-time computation that can be done before registering the query (notwithstanding updates to the relational tables while the query is running.) The second is when a relational table is joined with a stream. In this case, we can consider the join to be a filter on the stream with a variable cost-per-tuple that is dependent on the join selectivity, the different access paths available for the relation, and the placement of the relation (i.e., in memory, partially/completely on disk.) The optimizer should consider all possibilities and choose the one with the least cost. The estimated size of the resulting window is calculated in the same way as streaming joins if we consider the relational table to have a window size equal to its cardinality.

The Relation to Scheduling. The best plan, or any feasible plan, found using the previous model does not dictate the best method by which its operators should be scheduled for execution. In developing the model, we have assumed a steady state in which the arrivals are stable and can be represented using the average rate. In reality, arrival rates can exhibit bursts of high loads that require adapting. Such adaptations are the function of the scheduling strategies used [5]. The static plan, however, can guide the scheduler by dictating that the scheduling strategy allows execution time to each operator proportional to its cost per unit time. For example, plan A of Figure 4 suggests that a good scheduling strategy would, on average, allot selection σ_1 twice the time allotted for selection σ_2.

Complexity of Optimization. It can be proven, in a manner similar to [20], that the problem of finding the best join order for a deep tree of a conjunctive continuous query is NP-complete. An optimizer would have to resort to heuristics to reach a good plan. A dynamic programming approach, combined with heuristics similar to the ones used for traditional relational query optimization, would be applicable here too.

5. Approximation Methods

We now turn to the case when all the plans are infeasible and approximation is inevitable. *Load shedding* [8] is one form of approximation which reduces load by dropping tuples from the incoming streams. Load shedding can be done by several methods (e.g., random or semantic dropping of tuples) and can have several objectives (e.g., maximize throughput,) see [8][27] for a discussion. In this section, we consider random dropping of tuples as the method of approximation and the goal is to maximize the output rate of the approximated query. We consider the best way to place random filters[2], and the optimal setting of the amount that each filter should drop. We also attempt to answer another interesting question. If approximation is inevitable, does the goal of optimization change? In other words, is it better to approximate the best plan, or can we achieve the same, or higher, throughput working on a suboptimal one.

We start by handling the case of only selection operators and then extend the problem to include joins.

5.1 Selection Only Queries

Consider a query consisting of *n* consecutive filters, and an execution plan for it that orders the filters in ascending order by their designated numbers. The cost per tuple for filter *i* is c_i time units, and its selectivity is σ_i.

[2] We use random filter and drop box interchangeably.

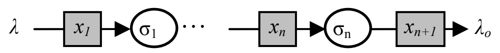


Figure 5. A plan with *n* filtering operators with drop boxes in all possible places.

Now, assume that the plan is infeasible and drop boxes should be used to approximate the result. There are *n+1* possible places to put drop boxes (see Figure 5.) We will assume that the *selectivity* of drop box *i* is x_i (i.e., the filter randomly drops $100\text{-}x_i$ percent of the tuples it sees.) Notice that the filter becomes unnecessary if its parameter is equal to 1. The problem is to determine the optimum values of the x_i*'s* such that the output rate is maximized. Using the model, the output rate of the approximated plan will be

$$\lambda_{approx} = \lambda_o \cdot \prod_{i=1}^{n+1} x_i \tag{18}$$

and the total cost of the approximated plan will be

$$C(p) = \sum_{i=1}^{n} \left(c(i) \cdot \prod_{j=1}^{i} x_j \right) \tag{19}$$

where λ_o is the output rate of un-approximated plan, calculated as

$$\lambda_o = \prod_{j=1}^{n} \sigma_j \cdot \lambda \tag{20}$$

and *c(i)* is the cost per unit time of filter *i*, *i=1..n,* calculated as

$$c(i) = \lambda \cdot c_i \cdot \prod_{j=1}^{i-1} \sigma_j \tag{21}$$

Using the previous equations, and noticing that we will only need to drop tuples in case the plan is infeasible (i.e., *C(p) > 1*,) we can formulate the problem as a constrained optimization one as follows

$$\textbf{Max } \lambda_{approx}$$

Subject to

$$C(p) = 1 \tag{22}$$
$$0 \le x_i \le 1, \quad i = 1 \ldots n+1$$

The above formulation leads to the following observation.

Observation 2

To approximate a plan for a filtering-only continuous query, we only need to drop tuples directly from the streaming source before they are processed by any of the filters. Furthermore, the approximation should be performed on the plan with the least cost in order to maximize the output rate given certain computational resources.

Proof Sketch

The proof is by inspecting the solution of equation (22). The optimum solution is:

$$x_1^* = \frac{1}{\sum_{i=1}^{n} c(i)} \tag{23}$$
$$x_j^* = 1, \; j = 2..n+1$$

The optimum value of the objective function is

$$\lambda_{approx}^* = x_1^* \cdot \lambda_o \tag{24}$$

From the solution, only the first drop filter is necessary with all the others having 100% selectivity. This proves that we only need to drop tuples directly from the streaming source before being processed by any filter.

To prove that the approximation should be performed on the plan with the least cost, two observations are necessary. First, the solution is applicable for any given plan for the query. Second, given a certain plan, *c(i)* is the cost per unit time for filter *i*, making the summation in the denominator of x_1^* the cost of running the plan without approximation. Combining these two, the lower the cost of the plan, the higher x_1^* is (i.e., the less the number of tuples dropped.) Since the optimum approximate rate is directly proportional to x_1^*, we can see that the plan with the lowest cost yields the highest approximated rate. ❑

The first part of this observation provides a rigorous validation of a rule of thumb reported in [29].

5.2 Join Queries

We now turn to the case where the query contains window joins. For ease of analysis, we will only consider tuple-based windows in this section. If the query is infeasible, the goal is again to find the optimum places to put drop boxes and the values of each drop box so that the throughput of the final plan is maximized. We would like to also confirm the intuition that the best approximate plan results from approximating the plan with the least cost as was shown in the filter-only case.

We first look at the where to put the drop boxes. For a query joining *n* streams, a drop box can be put before each of the two inputs to the *n-1* join operators, plus a box right after the last join is performed, resulting in *2n+1* possible places. We can show, however, that similar to the filter-only case, we need to drop tuples only from the input streams before they are processed by any join operator.

Observation 3

To approximate a plan for a continuous query joining *n* streams, it is sufficient to drop tuples only from the input sources before they are processed by any join operator.

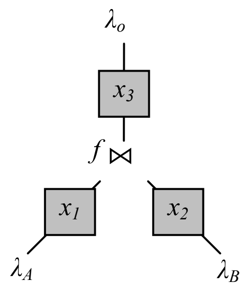


Figure 6. A join operator with drop boxes placed at all three possible locations.

Proof

Figure 6 shows an arbitrary join operator in an approximated plan for the n-ary join with drop boxes in all possible locations. Assume that W_A and W_B are the sizes of the left and right windows of the join. We will show that, given certain values of the parameters x_1; x_2; and x_3 of the drop boxes, we can always arbitrarily increase x_3 without affecting the rest of the plan while decreasing the cost of the join. Note that the operator's effect on the rest of the plan is through its output rate, and the resulting active window size.

In the case of tuple based windows, the active window size is obtained from equation (4). To prove that it is independent of the values of the drop boxes consider two cases: a) the two inputs to the join are directly coming from the input streams, and b) at least one input is a result of joining i streams, where $2 \leq i \leq n-1$. In case (a), the output window size obtained from equation (4) is independent of the input rates and hence the drop boxes. In case (b), let W_A be the window resulting from previous join(s), the value of W_A obtained from equation (6) is only dependent on the sizes of the windows on its i input streams, which are also independent of the input rates and the drop boxes. From the previous two cases we can see that the resulting window size is independent of the values of the drop boxes. This leaves the resulting output rate of the join.

To prove the observation for the output rate, consider its value. From the model, the output rate is

$$\lambda_o = x_3 \cdot f \cdot (\lambda_A \cdot W_B \cdot x_1 + \lambda_B \cdot W_A \cdot x_2) \tag{25}$$

Now, assume that x_3 is increased by Δx_3 to 1 (its maximum value.) In order for λ_o to remain constant, we need to decrease x_1 and x_2 by Δx_1 and Δx_2 respectively. After simplification, the relation between the increments is

$$\begin{aligned}\lambda_A \cdot W_B \cdot \Delta x_1 + \lambda_B \cdot W_A \cdot \Delta x_2 = \\ \lambda_A \cdot W_B \cdot \Delta x_3 \cdot x_1 + \lambda_B \cdot W_A \cdot \Delta x_3 \cdot x_2\end{aligned} \tag{26}$$

From the above, there are many values to set Δx_1 and Δx_2 to compensate for the increase in x_3[3]. The cost of the join, however, is directly proportional only to the input rates to the join, dropping tuples after the join has no effect on the cost. This means that decreasing the values of x_1 and x_2 will decrease the join cost. From the previous we can conclude that, for an arbitrary join operator and for any output rate, it is always preferable to drop tuples only from the inputs to the join.

Now, assume the arbitrary join in Figure 6 is the top most join in the query plan. Recursively applying the previous observation to the joins feeding its inputs until reaching the original input streams will complete the proof. ❑

We now turn to determining the amount of tuples dropped by each box. As in the previous section, we can formulate the problem as an optimization one. Placing drop boxes only at the leaves of a query plan decreases the complexity of the problem significantly. For every input stream i to the query with rate λ_i, there exists an associated drop box with the parameter x_i. Using equation (5), we can estimate the approximate output rate for a query with n input streams to be

$$\lambda_{approx} = \prod_{\substack{all \\ selectivities}} f \cdot \sum_{k=1}^{n} \left(\prod_{\substack{i=1 \\ i \neq k}}^{n} W_i \cdot \lambda_k \cdot x_k \right) \tag{27}$$

Contrary to the filtering only case, there is no closed form for the total cost of the approximated plan since the cost depends on its shape. However, it can be easily verified that the cost function is linear in the values of the x_i's. We can therefore express it as

$$C(p) = \sum_{i=1}^{n} a_i \cdot x_i \tag{28}$$

where the a_i's are constants. The problem can then be formulated as

$$\textbf{Max } \lambda_{approx}$$

Subject to

$$\begin{gathered} C(p) = 1 \\ 0 \leq x_i \leq 1, \quad i = 1 \ldots n \end{gathered} \tag{29}$$

The solution of the problem is easily calculated by noticing that it is an instance of the continuous knapsack problem. The solution is obtained by sorting the drop boxes parameters in descending order by the ratio of their coefficients in the objective function to their coefficients in the constraint equation (28). Then, we assign as much as possible to the parameter with the highest ratio, then if

[3] Note that the increments Δx_1 and Δx_2 are also bound by 0 and x_1 and x_2 respectively. Setting Δx_1 to Δx_3. x_1 and Δx_1 to Δx_3. x_2 still satisfies such bounds.

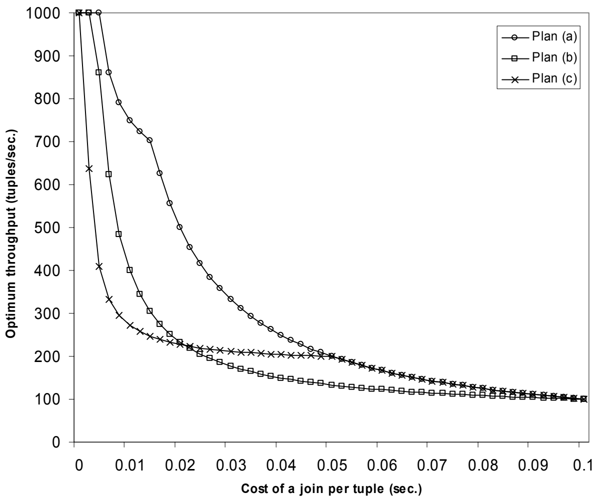


Figure 7. Optimum throughput for the query of example 1.

there is still room left to the next one, and so on until the constraint cannot be satisfied.

Interestingly, the intuition confirmed in the previous section about the best approximate plan does not carry over in the current case. The approximation of the plan with the least cost is not guaranteed to result in the best throughput. Figure 7 shows the optimum throughput for the three plans of example 1 as the cost per tuple is increased (i.e., system resources decrease.) All three plans start at the maximum throughput of 1000 tuples/sec. Up until the cost reaches 0.004 sec/tuple (i.e., system capacity of 250 tuples/sec,) plan (a) can sustain the maximum throughput without any approximation. Afterwards, all plans are infeasible and they start dropping tuples, which brings down the optimum throughput value. Plan (a), the one with least cost, remains the best choice until the cost is about 0.05 sec/tuple (system capacity of 20 tuples/sec,) at which time plan (c) (the worst feasible plan) catches up and the throughputs of the two become identical. If the cost increases to 0.097 sec/tuple (system capacity of about 10 tuples/sec,) the throughputs of all plans become identical, and the choice between all three becomes irrelevant. We can also see that when the cost is in the interval of 0.022 to 0.095 sec/tuple (system capacity of 45 to 10 tuples/sec,) the throughput of plan (b) is worse than that of plan (c) even though plan (b) has a lower cost if both are feasible. This shows that having a lower cost with no approximation does not guarantee that the plan will have a better throughput if approximated. This suggests that in case approximation is necessary, we need some different types of heuristics to guide the search than the ones used to find the optimum feasible plan. Such methods are still an open problem.

There are a number of issues regarding the solution presented above. In the following we will discuss these issues and suggest methods to deal with them.

1. The solution of the optimization problem may dictate that certain streams be completely shut off to produce the maximum throughput. This is similar to the result obtained in [22] for the single join case. This represents an undesirable semantic problem since for these streams, only the first window's worth of tuples will be represented in the join result. The problem is inherent in the objective of the approximation; to maximize throughput. The solution suggested in [22] is still applicable. A minimum rate can be required for each stream so that the answer is not completely biased. This can be done by raising the lower bounds on the parameters of the drop boxes from 0 to the required lower percentage for each stream. Such restriction may render a problem completely infeasible. However, if there is a feasible solution, it can still be obtained by the above algorithm, except for an additional initialization step that assigns to each parameter its minimum possible value.
2. The solution favors streams with higher value per cost ratio which may be different from the relative importance of the streams from the application's point of view (e.g., in a road monitoring situation, the system may prefer to get as much as possible from the road sensor readings than accurately tracking continuous polling from cars requesting update on their toll for the day in a system like the one simulated in [31].) This can be solved by assigning numeric weights to each input stream. These weights can be multiplied by the coefficient of each stream in the objective function to increase the solution's bias toward more important streams.
3. The solution views the selectivity of a join as symmetric for both of its inputs, which lead to them being considered as equals. In reality this may not be true. In a one-to-many join, for example, dropping a tuple from the one side can drop more than one tuple from the result while dropping a tuple from the many side will drop at most one tuple from the result. This can be solved by modeling the selectivity of the join by two numbers representing the join selectivities of incoming tuples on each side of the join into the windows of the other.

6. Conclusion and Future Work

In this paper, we developed a cost model for optimizing conjunctive continuous queries with sliding window joins. Using the cost model, we proved that all feasible plans of a continuous query produce the same rate, hence we introduced a framework for static optimization based on feasibility and resource management as opposed to rate-based optimization.

We used the cost model to study the problem of approximating a continuous query with tuple-based sliding windows using random filters. Our results show that given a plan, finding the optimum places for the random filters and their values is an easy task. However, finding a plan to approximate, is not that obvious. Furthermore, we found that the best feasible plan and the best approximate plan are not necessarily the same.

There are a number of possible extensions to this work, we discuss them in the following paragraphs.

The model can be extended in two directions; to handle multi-query optimizations, and to model disjunctions, partitions, and aggregations.

To answer the question of when to use static or dynamic optimization, models for both the overhead of adaptability and the change in data characteristics are needed to determine which situations each technique is more beneficial at, and when it would be better to use a hybrid scheme of the two.

From the discussion in Section 3.4, a feasible plan can still have a large response time. For applications with real-time requirements, a feasible plan may still be unacceptable. The problem can be solved by guiding the optimizer using our model to leave more head-room for the system to avoid approaching saturation. A better approach would be to use a queuing model to optimize directly for response time. Such a model would also be useful for modeling the average memory requirements of a plan.

References

[1] A. Arasu, B. Babcock, et al. Characterizing Memory Requirements for Queries over Continuous Data Streams. ACM PODS, June 2002.

[2] A. Arasu, S. Babu, J. Widom. An Abstract Semantics and Concrete Language for Continuous Queries over Streams and Relations. Technical Report, Department of Computer Sciences, Stanford University, November 2002.

[3] R. Avnur, J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. SIGMOD, May 2000.

[4] B. Babcock, S. Babu, et al. Models and Issues in Data Stream Systems. PODS, June 2002.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani. Chain: Operator Scheduling for Memory Minimization in Data Stream Systems. SIGMOD, June 2003.

[6] B. Babcock, M. Datar, et al. Maintaining Variance and k-Medians over Data Stream Windows. PODS, June 2003.

[7] B. Babcock, M. Datar, R. Motwani. Sampling From a Moving Window Over Streaming Data. SODA 2002.

[8] D. Carney, U. Cetintemel, et al. Monitoring Streams: A New Class of Data Management Applications. VLDB 2002.

[9] S. Chandrasekaran, A. Deshpande, et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. CIDR, January 2003.

[10] J. Chen, D. J. DeWitt, J. F. Naughton. Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries. ICDE 2002.

[11] J. Chen, D. J. DeWitt, F. Tian, Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. SIGMOD, May 2000.

[12] S. Chandrasekaran, M. J. Franklin. Streaming Queries over Streaming Data. VLDB 2002.

[13] A. Dobra, M. Garofalakis, J. E. Gehrke, R. Rastogi. Processing Complex Aggregate Queries over Data Streams. SIGMOD, June 2002.

[14] M. Datar, A. Gionis, P. Indyk, R. Motwani. Maintaining Stream Statistics over Sliding Windows. SODA 2002.

[15] A. Das, J. Gehrke, M. Riedewald, Approximate Join Processing Over Data Streams. SIGMOD, June 2003.

[16] J. Gehrke, F. Korn, et al. On Computing Correlated Aggregates Over Continual Data Streams. SIGMOD, June 2001.

[17] M. A. Hammad, M. J. Franklin, et al. Scheduling for shared window joins over data streams. VLDB 2003.

[18] P. J. Haas and J. M. Hellerstein. Ripple Joins for Online Aggregation, ACM SIGMOD, June 1999.

[19] Z. Ives, D. Florescu, et al. An Adaptive Query Execution System for Data Integration. SIGMOD, June 1999.

[20] T. Ibaraki, T. Kameda. On the Optimal Nesting Order for Computing N-Relational Joins. ACM Transactions on Database Systems, Vol. 9, No. 3, September 1984.

[21] N. Kabra, J. DeWitt. Efficient Mid-Query Reoptimization of Sub-Optimal Query Execution Plans. SIGMOD, June 1998.

[22] J. Kang, J. F. Naughton, S. D. Viglas. Evaluating Window Joins over Unbounded Streams. ICDE 2003.

[23] E. D. Lazowska, J. Zahorjan, G. S. Graham, K. C. Sevcik, Quantitative System Performance, Prentice Hall, 1984.

[24] S. R. Madden and M. J. Franklin. Fjording the Stream: An Architecture for Queries over Streaming Sensor Data. ICDE 2002.

[25] S. Madden, M. Franklin, J. Hellerstein, W. Hong. The Design of an Acquisitional Query Processor for Sensor Networks. SIGMOD, June 2003.

[26] S. Madden, M. Shah, et al. Continuously Adaptive Continuous Queries over Streams. SIGMOD, June 2002.

[27] R. Motwani, J. Widom, et al. Query Processing, Approximation, and Resource Management, in a Data Stream Management System. CIDR, January 2003.

[28] P. Selinger, M. Astrahan, et al. Access Path Selection in a Relational Database Management System. SIGMOD, May 1979.

[29] N. Tatbul, U. Çetintemel, et al. Load Shedding in a Data Stream Manager. VLDB 2003.

[30] F. Tian, D. J. DeWitt. Tuple Routing Strategies for Distributed Eddies. VLDB 2003.

[31] The Linear Road Benchmark. http://www.cs.brown.edu/research/aurora/linear-road.pdf.

[32] The Telegraph Project. http://telegraph.cs.berkeley.edu

[33] The Stanford Stream Data Manager. http://www-db.stanford.edu/stream.

[34] T. Urhan, M. J. Franklin, L. Amsaleg. Cost Based Query Scrambling for Initial Delays, SIGMOD, May 1998.

[35] S. D. Viglas, J. F. Naughton. Rate-Based Query Optimization for Streaming Information Sources, SIGMOD, June 2002.

[36] S. Viglas, J. F. Naughton, J. Burger. Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources. VLDB 2003.

[37] A. N. Wilschut, P. M. G. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. PDIS 1991.

[38] Y. Yao, J. E. Gehrke, Query Processing in Sensor Networks, CIDR 2003.