Static Optimization of Conjunctive Queries with Sliding Windows Over Infinite Streams

Ahmed M. Ayad Jeffrey F.Naughton
Department of Computer Sciences
University of Wisconsin - Madison
1210 W. Dayton, Madison, WI 53706
{ahmed, naughton}@cs.wisc.edu

ABSTRACT

We define a framework for static optimization of sliding window conjunctive queries over infinite streams. When computational resources are sufficient, we propose that the goal of optimization should be to find an execution plan that minimizes resource usage within the available resource constraints. When resources are insufficient, on the other hand, we propose that the goal should be to find an execution plan that sheds some of the input load (by randomly dropping tuples) to keep resource usage within bounds while maximizing the output rate. An intuitive approach to load shedding suggests starting with the plan that would be optimal if resources were sufficient and adding "drop boxes" to this plan. We find this to be often times suboptimal – in many instances the optimal partial answer plan results from adding drop boxes to plans that are not optimal in the unlimited resource case. In view of this, we use our framework to investigate an approach to optimization that unifies the placement of drop boxes and the choice of the query plan from which to drop tuples. The effectiveness of our optimizer is experimentally validated and the results show the promise of this approach.

1. INTRODUCTION

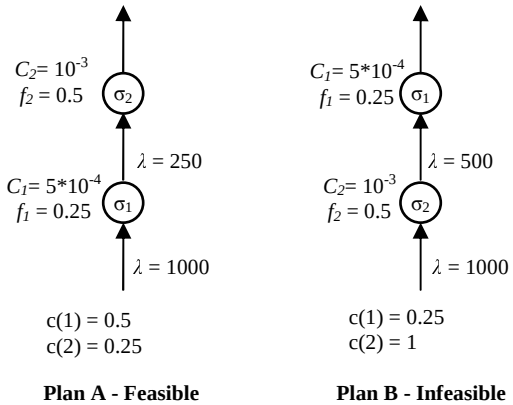
The focus of research on data and information processing has recently shifted towards an emerging type of applications in which the data is streaming from its sources. Such applications include monitoring network traffic, intrusion detection, telecommunications, sensor networks, financial services, and 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) so 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 [29] team which extended the SQL query language with constructs to pose queries on any combination of relations and continuous streaming sources [3]. This approach opens the problem of query optimization for continuous queries.

In the context of data streaming systems, the optimization problem is distinguished by the necessity to adopt the push model. The system has no choice but to keep up with the incoming data. Given a continuous query in steady state, each execution plan can be viewed as a queuing network system in which arriving tuples from the input streams are the clients and query operators are the servers. From basic queuing theory [20], 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.

To illustrate the above, Figure 1 shows an example of a simple query composed of two selections, σ_1 and σ_2, on a single data stream. The cost per tuple for the first selection, C_1, is half a millisecond and its selectivity, f_1, is 0.25. The cost per tuple for the second, C_2, is one millisecond and its selectivity, f_2, is 0.5. The rate of the input data stream is 1000 tuples/sec. For each selection operator, assuming computational resources are available (see section 3), the output rate of a selection is computed as simply its input rate multiplied by its selectivity. Two alternative plans exist for the query. In the first alternative (plan A), it takes 500



Plan A - Feasible **Plan B - Infeasible**

Figure 1. Feasible and infeasible alternatives to a query. c(i) is the cost per second of σ_i.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

***SIGMOD 2004*, June 13–18, 2004, Paris, France.**
Copyright 2004 ACM 1-58113-859-8/04/06 …$5.00.

milliseconds, on average, for selection σ_1 to process its input in one second, and 250 milliseconds for σ_2. This means that there is 75% average resource utilization and 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 hand, dictates that σ_1 needs 250 milliseconds, and σ_2 needs one second to handle the arrivals in a unit time. Hence, it is *infeasible*. Since a feasible plan exists for the query, the query itself is *feasible*.

It can be observed that, if an execution plan is feasible, its final output rate is only determined by the rates of the input streams. Since the input is the same for all plans of a query, this leads directly to the observation that all feasible plans of the same query have the same final output rate (see Section 3.1.3).

If no feasible plans exist for the query, load shedding becomes a necessity to bring down the demand on the system to within the available resources. Load shedding can be done by several methods (e.g., random or semantic dropping of tuples) and can have several objectives, see [1][23] for a discussion. In this work, we choose random dropping of tuples as the method of load shedding. This is achieved by inserting random drop boxes at several points in the query plan. When tuples are being dropped from a plan, the final output rate becomes dependent on the amount of shed load. Since, as demonstrated above, plans differ in their resource usage, different plans will need different amounts of load shedding. Therefore, the final output rates of plans with drop boxes inserted will differ. In light of this, we choose the goal of load shedding to be the plan that maximizes the output rate of the partial answer query. In this context, there are two different problems that need to be addressed. The first is the optimal placement of drop boxes in an execution plan and the optimal setting of their sampling rate. The second is concerned with the choice of plan to shed load from. Notice that in this case, all the plans considered should finally have the same resource utilization, the maximum possible, while differing on the final output rate. Recent research on load shedding (e.g., [7][25]) focused on examining the best method to shed load from a given plan. The plan used is usually assumed to be the plan selected when resources where sufficient. We are unaware of any attempts to address the issue of selecting the best plan for load shedding.

Given the above discussion, a static query optimizer for continuous queries faces a number of challenges. In case the query is feasible, the optimizer has to find the feasible plan for the query that has the lowest resource utilization, or at least avoid the infeasible plans to avoid unnecessary load shedding. In case the query is infeasible, the goal becomes to search for the plan that, when tuples are dropped from it, yields the maximum output rate. We present a framework for static query optimization that tackles these challenges. In particular, our main contributions are:

- We develop a model for estimating the final output rate and resource utilization of an execution plan of a continuous query.
- We use the model to investigate the best way to shed load from a plan by inserting random drop boxes.
- We show that the approach of shedding load from the plan that was running when resources were sufficient is often times suboptimal. Significant gains can be achieved if the query is re-optimized with load shedding in mind.
- We develop an optimizer that integrates load shedding into the optimization process by taking resource constraints into account.
- We experimentally validate the effectiveness of our optimization framework.

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 [4][17][28]. This is built on the earlier idea of mid-query re-optimization [18]. 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 environments, static optimization will be best. At the other extreme, very dynamic environments, 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 the semantics of sliding window conjunctive streaming queries that we study. Section 3 describes the cost model used in the optimization problem. Section 4 tackles the load shedding problem. Section 5 defines the optimization framework and the proposed optimizer. Section 6 discusses the experimental evaluation. Section 7 discusses related work in the literature. Section 8 concludes the paper.

2. THE SEMANTICS OF SLIDING WINDOW CONTINUOUS QUERIES

There has been no agreed upon concrete semantics for queries over data streams. Attempts towards this goal can be found in [1][3]. These attempts differ slightly on the meaning of a timestamp, whether strict ordering of tuples is required, handling out of order tuples, timestamps for generated tuples, querying relational data, and how resulting tuples are streamed out.

For the purpose of this work, we are only concerned with the modeling of a data stream and the precise semantics of the selection and the sliding window join operators. We assume a global, discrete, ordered time domain $\mathcal{T}$ from which timestamp values are derived. For ease of analysis, we also make some simplifying assumptions:

1- For any data stream, the time stamps are unique; there are no ties.
2- Tuples arrive in the stream in a monotonically increasing order by its time stamp; there is no out of order arrival.
3- There are no relational tables involved in the query.

The easiest way to satisfy our assumptions for timestamps is to assume that they are assigned by the system for each tuple upon its arrival.

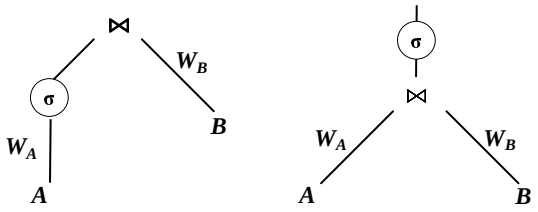


Figure 2. Two plans for the same query. The active window size on the left side of the join is less in the left plan.

2.1 Definitions

We adopt the definition of data streams in [3].

Definition 1: Data Stream. A streams S is a bag of elements <s, t>, where s is a tuple belonging to the schema of the stream and $t \in \mathcal{T}$ is the timestamp of the element. ❑

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 predicates exist; each has its own modeling requirements. A discussion of the different types can be found in [1][3][8]. For the purpose of this paper, we will only consider tuple-based and time-based sliding windows. Again, we adopt the definitions in [3].

Definition 2: Time-based Window. At any time instant *t*, a time-based window of size *T* on a stream *S* defines a subset of *S* containing all elements of *S* with timestamp *t'* such that $t\text{-}t' \leq T$. ❑

Definition 3: Tuple-based Window. At any time instant *t*, a tuple-based window of size *W* on a stream *S* defines a subset of *S* with the largest *W* timestamps not exceeding *t*. If the size of *S* at time *t* is less than *W*, the window includes all elements of the stream. ❑

Note that the number of tuples satisfying the window predicate is affected by the tuple arrival rate in the case of time-based windows only. Tuples satisfying the window predicate can become *stale* by the passage of time and the window size can eventually be zero if no new tuples arrive. This is in contrast with tuple-based windows, in which, once the window is full, the number of tuples satisfying the predicate remains constant regardless of the rate of new arrivals.

2.2 Selection and Join Semantics

A selection operator, also called a filter, takes a stream *S* as input and outputs a stream *O* whose elements are the subset of *S* that satisfy the selection predicate. Elements in the output stream of a selection have the same timestamps and relative order they had in the input stream.

As for joins, since we are only considering streaming sources, we are only interested in sliding window joins. Also, without loss of generality, we will only consider equality predicates. From [2], equi-joins on infinite streaming sources result in unbounded memory requirements, hence the necessity of sliding window predicates.

The sliding window join is a symmetric operator that takes two input streams, *S* and *R*. For every arriving tuple on any of the two input streams, the operator joins it with the current window contents on the other input stream. The operator then streams out resulting tuples that satisfy the join predicate. The timestamp of a resulting element from the join is the greater of the two timestamps of its components. The resulting stream is ordered on the timestamps of its elements.

3. THE COST MODEL

In this section, we provide the necessary calculations to estimate the expected processing constraints for providing an answer to continuous queries. First, we derive the necessary equations to estimate the output rate for the different operators assuming there are no constraints (i.e., assuming the plan is feasible.) Second, we estimate the size of the *active window*. By that we mean the average number of output elements that are eligible for participation as input if the output of the operator is fed into the input of a second one. Consider the example in Figure 2. It shows two plans for the same query that joins streams *A* and *B* and has a selection on *A*. Both streams have tuple-based window predicates, W_A and W_B respectively. For the plan on the right, each element arriving from stream *B* joins with the latest W_A elements of stream *A*. This can't be true for the left plan or else we would be joining elements from *B* with the latest W_A elements that pass the selection, instead of the latest W_A arriving. Instead, for the left plan, elements from stream *B* should join only with the active elements in the window on *A*, the size of which is W_A multiplied by the selectivity of the selection operator. A similar argument can be made for the size of the active window if the windows were time-based instead. Lastly, we move to estimate average processing 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. We also assume that there is enough memory to hold the buffering requirements for any query plan. Table 1 defines the notation used throughout the paper. All costs are in time units.

We will develop the cost model for tuple-based windows only. 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.

The development of the results concerning the output rates and costs of single operators resembles the one in [19].

3.1 Rate and Window Calculations

3.1.1 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

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

For a selection operator with a window W_i defined on its input, the active window size is (see discussion of Figure 2 above)

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

3.1.2 Joins and Cartesian Products

A Cartesian product can be viewed as a special case of a join with the selectivity factor equal to 1. We define the selectivity factor of a sliding window join to be the percentage of tuples satisfying the join predicate relative to a simple Cartesian product.

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

We assume, without loss of generality, that the selectivity is symmetric relative to the two inputs.

Now, the number of tuples arriving from the left side of the operator in a unit time is equal to λ_L. From the join semantics, each of which is expected to join with $f \cdot W_R$ tuples from the right side window. Hence, the number of tuples produced as a result of tuples arriving from the left side is $f \cdot W_R \cdot \lambda_L$ per unit time. Similarly, the number of tuples resulting from right side arrivals is $f \cdot W_L \cdot \lambda_R$. So, the total output rate for a window join is

$$\lambda_o = f\,(W_R \cdot \lambda_L + W_L \cdot \lambda_R) \tag{3}$$

To compute the active window size, we need to estimate the average number of valid tuples coming out of the join. A joined tuple is considered valid (not expired) only if all the original tuples it is comprised from are still valid. Consider the join operator at steady state. There are W_L and W_R active tuples in the windows on the left and the right sides respectively. Each of which must have already joined with the other active tuples in the opposite window. The resulting size of this join is $f \cdot W_L \cdot W_R$. Now, consider arrivals on the left(right) side of the join. Each arriving tuple that is inserted into the window on the left(right) side produces $f \cdot W_R (f \cdot W_L)$ new tuples. While at the same time, the arriving tuple invalidates the earliest one in the window, causing the same number of tuples to become invalid. Hence, on average, the number of resulting active tuples stays the same. So

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

3.1.3 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-way 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 3.1.1, suggest that the *steady state* output rate of a conjunctive continuous query, given enough resources, is independent of the execution plan and that it should not be the goal of query optimization.

3.1.4 Discussion

We pause to discuss some issues relating to the previous observation. We have proved that all feasible plans of a continuous query have the same output rate. From the semantics discussed in Section 2, all feasible plans must produce the same tuples in the same order, and with the same timestamps. This does not mean, however, that all feasible plans produce the same output at exactly the same time. To understand this, it may be helpful again to regard a query execution plan as an open queuing system. From queuing theory, the utilization and response times of two stable systems 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 its timestamp. The response time of a plan is the average response time of all its resulting tuples. Feasible plans differ in their response times, meaning that they produce the same result tuples with each shifted in time, from its timestamp, by an average amount equal to the average response time of the plan.

3.2 Processing Constraints

We move to derive the necessary computational resource requirements for the different types of operators given their inputs. We also compute the constraints on these resources.

3.2.1 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 holds

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

3.2.2 Joins and Cartesian Products

In the case of joining infinite streams, only non-blocking algorithms can be used, like the symmetric hash join [33]. Kang et al. made the observation in [19] 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

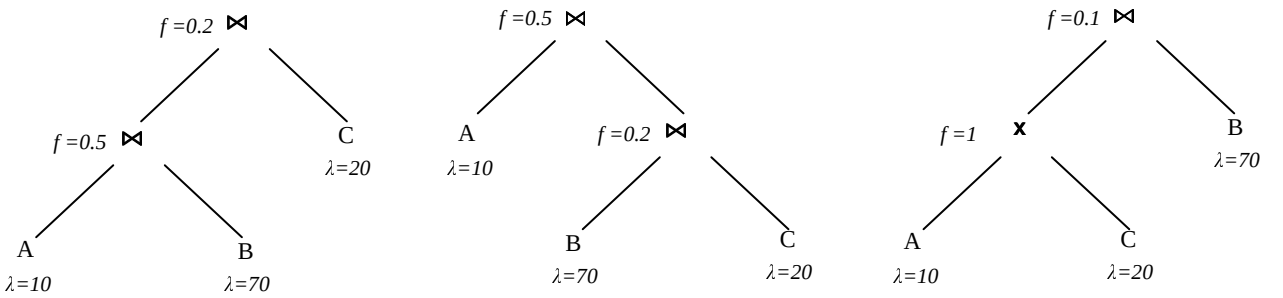


Figure 3. Possible plans to evaluate the join. Assuming enough resources, all three plans have the same final output rate.

window join which we will use here. The cost of the join per unit time is

$$C_L = \lambda_R \cdot C_P(L) + \lambda_L \cdot (C_I(R) + C_V(R))$$
$$C_R = \lambda_L \cdot C_P(R) + \lambda_R \cdot (C_I(L) + C_V(L)) \quad (8)$$
$$C_{L \bowtie R} = C_L + C_R$$

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

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

In both cases, the constraint is

$$C_{L \bowtie R} < 1 \quad (10)$$

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).

It is worth mentioning that the cost of the join is dependent on the join algorithm used. The model presented in [19] can be used to choose the best possible algorithm for each binary join.

3.2.3 Notes on the Processing 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 another 9 identical operators, then the cost should increase ten fold to 10 milliseconds.

Example 1

We end this section with a concrete example on the application of the cost model. Consider the following simple SQL-like query (the window constraint syntax is modeled after [23]):

```
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, 70, and 20 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. Since the cost of the plan is the summation of the individual costs of its operators (in this case the two joins) the previous assumption makes the cost of the plan directly proportional to the summation of the input streams rates and the output rate of the intermediate join. It is not hard to also show that the utilization of every plan is the multiplication of this sum by the join cost. Figure 3 shows the possible plans to evaluate the query. Note that using the model and assuming each plan has enough computational resources to execute, all three have the same final output rate.

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. The plans differ dramatically, however in terms of their resource utilization. While plan (a) keeps the system 25% utilized (500*.0005), plan (c) has only 20% utilization (400*.0005), and plan (b) has 14% utilization (280*.0005). Plan (b) is therefore the best choice. Choosing plan (a) results in a 170% increase in the necessary resources to answer the query.

Now, assume that a join requires 3 milliseconds to handle an incoming tuple, meaning that the system capacity is about 334 tuples/second. In this case, plan (a) will have 150% utilization, plan (c) will have 120% utilization, and plan (b) will have 84% utilization meaning that only plan (b) is feasible. Choosing either of the other two plans will unnecessarily require load shedding.

If a join requires 5 milliseconds per incoming tuple (i.e., maximum system capacity of 200 tuples/second), all plans become infeasible and some load must be shed. 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 load shedding 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 [11]). We discuss in the next section how to arrive at this choice.

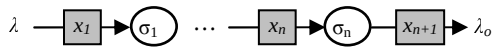


Figure 4. A plan with *n* filtering operators with drop boxes in all possible places.

4. LOAD SHEDDING

We now turn to the case when all the plans are infeasible and a partial answer is inevitable. *Load shedding* [1] 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 or quality of service), see [1][23] for a discussion. In this section, we consider random dropping of tuples as the method of load shedding and the goal is to maximize the output rate of the approximated query. We consider the best way to place random filters[1], and the optimal setting of the amount that each filter should drop. As mentioned in the introduction, there are two basic questions the optimizer needs to answer. The first is, given a plan to shed load from, where do we place the random filters, and how much should we drop in each? The second, which plan do we choose for load shedding? Intuition suggests that we should choose the best plan when resources were sufficient. We test the validity of this intuition here.

We assume for convenience that the random filter has negligible cost compared to other operators. Since the drop boxes are artificial operators, we will also assume that they are semantically invisible (i.e., the query operators will not differentiate between a drop in arrival rate at the source and one resulting from a drop box.) We start by handling the case of only selection operators and then extend the problem to include joins.

4.1 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. Now, assume that the plan is infeasible and drop boxes should be used to shed load. There are *n+1* possible places to put drop boxes (see Figure 4.) We will assume that the *selectivity* of drop box *i* is x_i (i.e., the filter randomly drops *100*(1-*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 partial answer plan will be

$$\lambda_{approx} = \lambda_o \cdot \prod_{i=1}^{n+1} x_i \tag{11}$$

and its total cost will be

$$C(p) = \sum_{i=1}^{n} \left(c(i) \cdot \prod_{j=1}^{i} x_j \right) \tag{12}$$

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

$$\lambda_o = \prod_{j=1}^{n} \sigma_j \cdot \lambda \tag{13}$$

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

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

$$c(i) = \lambda \cdot c_i \cdot \prod_{j=1}^{i-1} \sigma_j \tag{14}$$

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

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

Subject to

$$C(p) = 1$$
$$0 \le x_i \le 1, \quad i = 1...n+1 \tag{15}$$

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

The easiest way to prove the above is to consider the analogy between the problem at hand and the one concerning the optimum way to order a number of expensive predicates over a relational table, replacing the input relation cardinality by the input stream rate[2]. From rank optimization [14], all random filters have zero cost and selectivities less than one, which means they will all have infinite rank. Hence they should all be pushed to the left to be applied the earliest. Since a combination of random filters amounts to a single one, we deduce that the optimum solution is to have a single filter at the beginning. This proves the first part of the observation and leads to the following solution of equation (15):

$$x_1^* = \frac{1}{\sum_{i=1}^{n} c(i)}$$
$$x_j^* = 1, \quad j = 2..n+1 \tag{16}$$

The optimum value of the objective function becomes

$$\lambda_{approx}^* = x_1^* \cdot \lambda_o \tag{17}$$

To prove that the load shedding should be performed on the plan with the least cost, two observations are necessary. First, the solution above 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 load shedding. Combining these two, the lower the cost of the plan, the higher x_1^* is (i.e., the fewer the number of tuples dropped.) Since the optimum approximate rate is directly proportional to x_1^*, we deduce that the plan with the lowest cost yields the highest rate. ❑

[2] Another method is to directly solve the constrained optimization problem.

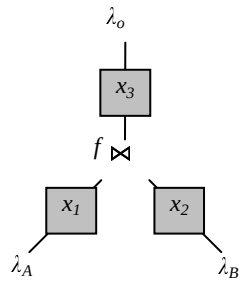


Figure 5. A join operator with drop boxes placed at all three possible locations.

The first part of this observation provides a rigorous validation of a rule of thumb reported in [25].

4.2 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. We first investigate the optimum method to drop tuples from a given query plan, and then we move to investigate the choice of the plan to shed load from.

4.2.1 Shedding Load from a Specific Plan

We first look at the where to put the drop boxes. For a query plan joining *n* streams, using binary joins, 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.

Proof Sketch

Figure 5 shows an arbitrary join operator in an approximated plan for the n-way join with drop boxes in all possible locations. As a first step to prove the observation, we need to show that, given any 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.

The operator's effect on the rest of the plan is through its output rate, and the resulting active window size. If we prove the manipulations of the filter values will not affect both values, we can guarantee they will not affect the rest of the plan.

In the case of tuple-based windows, it can be easily shown by examining equations (4) and (6) that in steady state, the size of the resulting active window size of any join in a query plan is always independent of the values of the input stream rates. Hence, manipulating the settings for the drop box won't affect the active window size. This leaves the resulting output rate.

For the output rate, it suffices to show that, to keep the output rate the same after increasing x_3, the values of x_1 and x_2 must decrease. Since x_3 does not contribute to the cost of this join, the final effect of this manipulation would be a decrease in the join cost.

Now, assume the arbitrary join in Figure 5 is the top most join in the query plan. We can then consider it to be a base case and recursively apply the previous observation to the joins feeding its inputs until reaching the original input streams. This completes the proof. ❑

Despite the difference in characteristics between time-based and tuple-based windows (the number of active tuples in a time-based window is dependant on the input rates), a similar reasoning can be applied to prove the previous observation for time-based windows.

We now turn to determining the selectivity of 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 output rate after load shedding for a query with n input streams to be

$$\lambda_{approx} = \prod_{\substack{all \\ selectivities}} f \cdot \sum_{i=1}^{n} \left(\left(\prod_{\substack{k=1 \\ k \neq i}}^{n} W_k \right) \cdot \lambda_i \right) \cdot x_i \qquad (18)$$

It can also be easily verified that the cost function is linear in the values of the x_i's. We can therefore express it as

$$C(p) = \sum_{i=1}^{n} a_i \cdot x_i \qquad (19)$$

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

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

Subject to

$$C(p) = 1$$
$$0 \le x_i \le 1, \quad i = 1...n \qquad (20)$$

The solution of the problem can be obtained by observing that the problem has a linear objective function, one linear constraint in all the variables, and a set of limiting constraints on each variable. This problem is then an instance of the continuous knapsack problem. Thus, the solution is by the following algorithm:

1- Set all values of the variables to 0.
2- For every variable x_i, compute the ratio

$$\frac{\left(\prod_{\substack{k=1 \\ k \neq i}}^{n} W_k \right) \cdot \lambda_i}{a_i}$$

which is the ratio between its coefficients in the objective function (less the multiplication of all selectivities since it is constant for all variables), and the equality constraint respectively.
3- Sort the ratios in descending order.
4- If all ratios have been considered, then stop.
5- Set the value of the variable that corresponds to the current highest ratio to the maximum possible; 100%.
6- If setting the latest variable causes $C(p)$ to exceed 1 then decrease it until $C(p)$ reaches 1 and stop. Else, remove the variable and its ratio from the list.
7- Repeat step 4. ❑

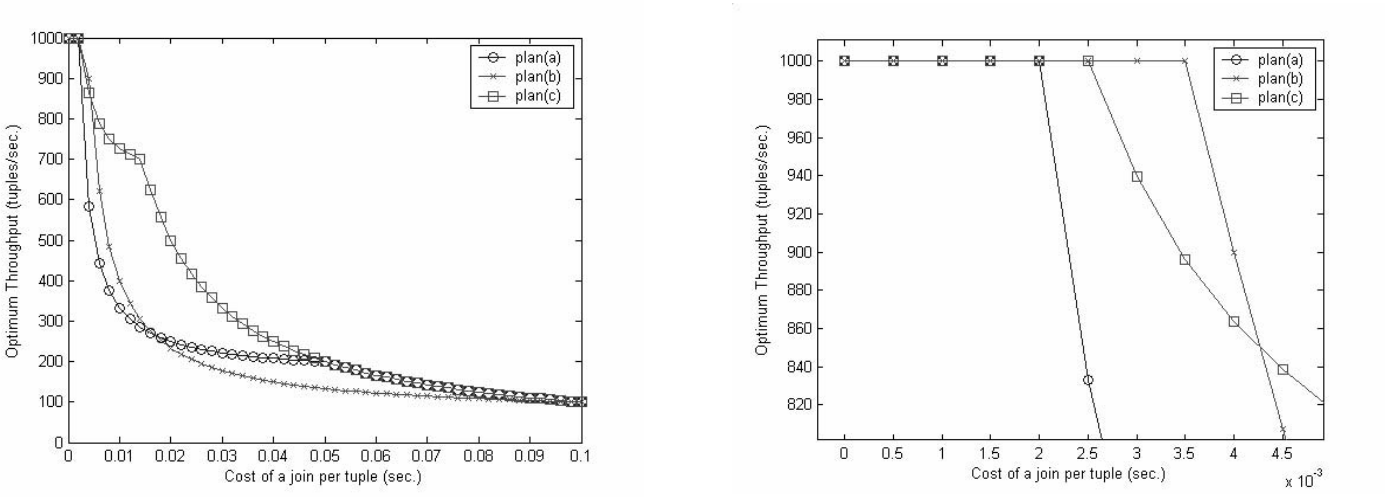


Figure 6. Optimum throughput for the query of Example 1.
The figure on the right magnifies the upper left corner of the left figure, with more data points.

4.2.2 Choice of Plan for Load Shedding

We now move to investigate the second question concerning load shedding; what is the best plan to select? In Section 4.1 we have confirmed for selection queries the intuition that the plan to select is the one with the lowest resource utilization. Interestingly, this intuition does not carry over in the case of join queries. Depending on the available resources, a plan that would have been suboptimal when resources were abundant can be a better choice for load shedding. We show this using a simple example. Consider the query of Example 1 presented in Section 3. We tested the behavior of each of the three alternative plans for the query when the join cost per tuple increases from 0 (infinitely fast processor) to 100 milliseconds. For each plan, we computed the optimum output rate at each join cost. When a plan is infeasible, the optimum output rate is the one obtained after drop boxes have been optimally inserted into the plan. The left side of Figure 6 shows the behavior of the three plans. All three plans start by delivering 1000 tuples/sec., which is the maximum possible rate. As resources become scarcer, the throughputs of the plans start to drop as they are forced to shed load. The plans start shedding load in the order of their average utlization, starting with the worst, plan (a), followed by plan (c) then finally, when the join cost exceeds 3.5 milliseconds, plan (b) starts to shed load. Somewhere between the join cost of 4 and 4.5 milliseconds per tuple, a switch over occurs (see the right side of Figure 6.) For join costs starting 4.5 milliseconds and higher, plan (c) becomes the best choice, delivering the maximum throughput. This trend continues until all plans deliver the same throughput at join cost of 100 milliseconds.

A number of interesting observations can be made on the previous example:

1- The plan with the lowest utilization is not always the best choice for load shedding.
2- The gap between the lowest utilization plan and the best plan to shed load from keeps increasing until the point when the join cost is approximately 14 milliseconds. At this point, the throughput of the best plan is more than twice the throughput of the lowest utilization plan.
3- It may be the case that the lowest utilization plan is actually the worst choice, as it is in the example when the join cost exceeds 17 milliseconds.

From the previous demonstration, it is evident that load shedding has to be integrated in the process of optimization, as opposed to being treated as an afterthought. When searching for the best plan, the optimizer must take into account the resource constraints in addition to the input stream rates, window sizes and selectivities. This is the focus of the rest of the paper.

5. THE OPTIMIZATION FRAMEWORK

We are now ready to formulate the optimization problem for conjunctive queries over infinite streaming sources. We start by defining a query plan, and then we move on to the objective of the optimization. Finally, we discuss a heuristic based dynamic programming optimizer developed to approximate the best left deep tree for a tuple-based sliding window conjunctive query. Although, as in the load shedding section above, we only develop the problem for tuple-based sliding windows, many aspects of the solution carry to time-based windows. A complete treatment of time-based windows is left for future work.

5.1 The Optimization Problem

Given a conjunctive query Q on streaming sources, we can define two functions on any execution plan p for Q. The first is $\lambda(p)$, which is the throughput of the plan, and the second is $C(p)$, which is the utilization cost of the plan. $\lambda(p)$ is bounded by the maximum output rate of the query, and $C(p)$ is bounded from above by 100%.

From the previous discussions it is now clear that there are two distinct modes of operation. The first is when the query is feasible, and the second is when it is not. For the first mode, the goal of optimization is to minimize $C(p)$. While in this mode, $\lambda(p)$ is fixed at its maximum value for all feasible plans p of the query. In the second mode, the goal is to maximize $\lambda(p)$. In this mode, the value of $C(p)$ is fixed at its maximum value for all p.

To tackle the problem in a uniform manner, we will assume that the search space of alternative plans for Q is always equipped with drop boxes for load shedding, if necessary. This way, all

plans in the search space will be feasible, and we can treat the problem as an unconstrained one.

Now, we can define the objective of the optimization of a query Q as

$$\textbf{Max}\ R(p) = \frac{\lambda(p)}{C(p)}, \text{ where } p \text{ is a plan for } Q \qquad (21)$$

To see why this works, consider a feasible query. For all plans p of the query, either p has no drop boxes, which means that the numerator of R is fixed at the maximum query throughput while the denominator is less than 1, or p has drop boxes, which means that the denominator is now 1, while the numerator is less than the maximum throughput. It is then obvious that all plans with no drop boxes have a higher value of R than any one with. Among all plans with no drop boxes, the one with the least cost has the highest value. If the query itself is infeasible, all plans will have drop boxes and the one with maximum throughput will have the highest R value.

Using equation (21), the simplest optimization algorithm is now as follows:

1- Generate the set $\mathcal{P}$ of all plans of the query.
2- For all $p \in \mathcal{P}$, compute $C(p)$.
3- If $C(p) > 1$, insert drop boxes in p using the algorithm of Section 4.2.1.
4- Compute $R(p)$.
5- Return p^* that maximizes $R(p)$. ❑

The complexity of the above algorithm is obviously combinatorial in the number of input streams being joined. Since the algorithm to determine the optimum settings of the drop boxes is linear in the number of input streams, the problem has, in essence, not changed a lot from the traditional optimization problem of conjunctive queries for relational data. Some techniques should be directly applicable here (e.g., randomized algorithms, as in [15].) In the next section, we propose a bottom up dynamic programming optimizer, similar to the approach in [24], which searches the space of left deep plans.

5.2 A Heuristic Optimizer

One technique used in relational optimization to reduce the size of the search space is to confine the search to only left deep plans. This was used by the original System R optimizer [24]. In this section, we adapt the dynamic programming optimizer of System R to search, bottom up, for the best left deep plan for a continuous query. The optimizer uses equation (21) as its objective function.

At first glance, the problem looks trivial. The optimizer should treat the drop boxes as regular selection operators and proceed with optimization normally. The catch is, unlike normal selections, the selectivity of the drop boxes are not known beforehand. In fact, the selectivity of the drop boxes is one output of the optimization procedure.

The way our proposed algorithm works is by proceeding as the original System R optimizer, building the plan bottom up by storing the best plans for successively larger subsets of the input streams. When computing the best plan for any subset, the algorithm tests whether this subplan is actually feasible given the resource constraints. If the plan is infeasible, the algorithm tunes the values of the drop boxes placed at its input streams using the load shedding algorithm. The subplan is then stored with the settings of its drop boxes. At the next stage when it is reconsidered, the stored settings of the drop boxes are taken into account as if the drop boxes were normal filters. If at any stage the algorithm places a drop box in front of a stream which had another one from a previous round, the two are combined into one drop box whose selectivity is the product of the original two.

The astute reader will notice that we have relied on the same optimality principal employed by the System R optimizer; namely that the best plan to join a subset of the streams of size k+1 in which stream k+1 is the last one to join is the plan that joins stream k+1 with the best subplan joining the other k streams. It can be shown that the optimality principal holds if the query has a feasible left deep plan. The algorithm is guaranteed to arrive at the best feasible left deep plan for the query if any exists. If the query is infeasible, however, this is not necessarily true. This is why we call it a heuristic, since the algorithm is not guaranteed to arrive at the best plan which maximizes throughput if no feasible plan exists. We can intuitively argue though, that the heuristic will perform well in most cases. The reason is, it postpones the decision for dropping tuples until the latest possible round and progressively adjusts the values of the drop boxes only when needed.

We test the performance of our optimizer in the next section.

6. EXPERIMENTAL EVALUATION

In this section we discuss a number of experiments designed to study the following points:

1- We have shown by example that reoptimization when load shedding is necessary can be better than sticking with the lowest utilization plan. The question is: Was this just an artificially constructed pathological case, or is this something that occurs often enough that it is worth paying attention to?
2- We study the benefits of reoptimization when load shedding is necessary. In particular, we answer the question: How much do we lose if we shed load from the lowest utilization plan and ignore reoptimization?
3- We validate the effectiveness of the heuristic optimizer developed in the previous section.

We limited the search space throughout the study to that of left deep plans.

Table 2. Fixed parameters for the randomized queries.

$f_{A\bowtie B}$	0.2	W_A	100
$f_{A\bowtie C}$	0.5	W_B	300
$f_{B\bowtie D}$	0.1	W_C	500
$f_{D\bowtie E}$	0.001	W_D	400
		W_E	1000

6.1 Setup

For all experiments, we generated 1000 random continuous queries with tuple-based sliding window joins. Each query represents a join of five input streaming sources A, B, C, D, and E. For all queries, the window sizes and join selectivities were fixed, while the rates of the input streams were randomly picked uniformly from 10 to 1000 tuples/sec. We tried the same set of experiments with different values of the join selectivities and

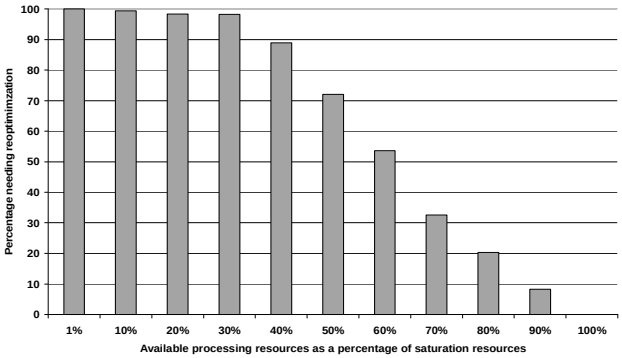


Figure 7. Percentage of queries benefiting from reoptimization.

window sizes but found the general trends in our results to be relatively insensitive to these changes. Table 2 contains the settings used for the fixed parameters.

As in Example 1 above, we assumed that the join cost per tuple is fixed. This enables the characterization of system resources to be represented only by this single value. Using an exhaustive optimizer that searches the space of left deep plans, we determined for each query the plan with the lowest resource utilization. Then, we found the join cost per tuple at which the query becomes infeasible. We call the inverse of this value (measured in tuples/sec) the *saturation resources*, which means that at this capacity, the system becomes saturated. We then gradually increased the join cost and took measurements at 1% decrements of the saturation resources (e.g., if the saturation resources are 1000 tuples/sec, we measured at resources decreasing by 10 tuples/sec.) At each of these points, we optimized each query using the exhaustive optimizer, then again with our heuristic optimizer. We report our findings at 100% of saturation resources then decreasing by 10% until 10% of saturation resources, then finally at the 1% level. All the reported results are based on the predictions of our cost model of the performance of the query plans.

6.2 The Need for Reoptimization

In this experiment we measured for every examined level of system resources, the percentage of queries which benefited from reoptimization (i.e. the lowest utilization plan is not the best choice for load shedding.)

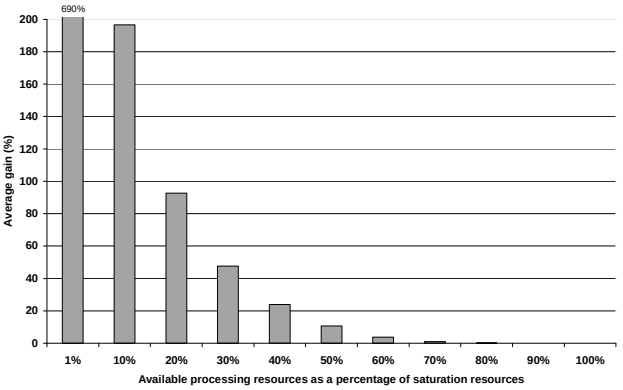


Figure 8. Average gain in throughput over using the lowest utilization plan.

Figure 7 shows the results of the experiment. The probability that a plan would need reoptimization rises almost linearly with decreasing system resources. At about 60% of saturation resources, a plan will more likely than not need reoptimization. The curve flattens at around 30% of saturation resources, after which, it is almost certain that reoptimization is better.

We then measure the tangible benefits of reoptimization. At each examined resource level, we compute for each query the throughput after load shedding for the best left deep plan with drop boxes. We then compute the throughput of the lowest utilization plan after load shedding. We compute the gain as the ratio between the difference of the two throughputs and the lowest utilization throughput (i.e., a gain of 100% means that the best plan delivers twice the throughput of the lowest utilization one.) We then compute the average gain for all queries.

From Figure 8, at very low resources, the gain is very significant (almost 8 folds at the 1% mark.) Significance drops, however, as more resources become available. At the 60% level, when there is more than a 50% chance of having improvement, the average gain is about 4%.

To check the effect of dilution from the queries that didn't need reoptimization, we repeated the previous experiment but computed the average gain only among the queries that benefited from reoptimization. We also report, at every resource level, the max gain attained. The results are shown in Figure 9. As expected, the most notable difference between the two averages was when available resources are near the saturation level. Starting from the 50% level, most of the plans benefit from reoptimization and there is no notable dilution effect. The figure also depicts the maximum gain measured at each level of resources. As resources decrease, the ratio between the maximum and the average gain also decreases. Near the saturation resources level, the gain is negligible for most queries needing reoptimization, but it makes a huge difference for the worst case. With decreasing resources, the benefit of reoptimization becomes more distributed among all queries.

6.3 Testing the Heuristic Optimizer

Our final experiment is designed to gauge the effectiveness of the heuristic optimizer developed in Section 5.2. To accomplish this, at every examined level of resources, we optimized every query using our optimizer. Then, for every query, we computed the difference in the value of the objective function between the plan found by the optimizer, and the best one acquired through the exhaustive search. The ratio between this difference and the best value is the relative error of the optimizer for this query.

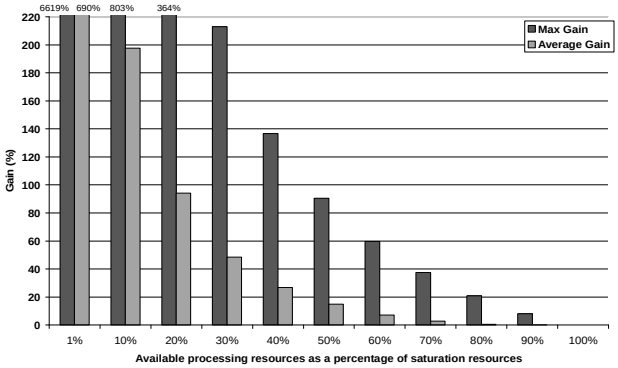


Figure 9. Average and maximum gain.

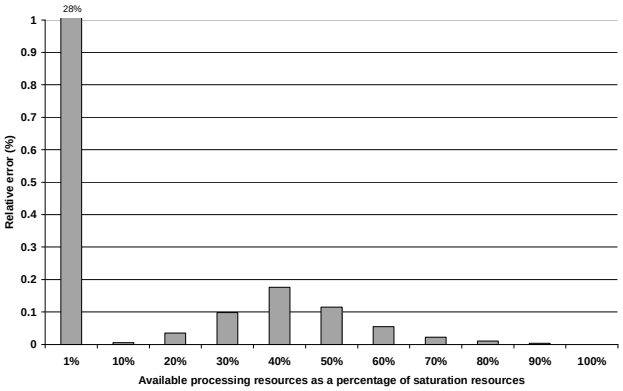


Figure 10. Average relative error for the optimizer.

From Figure 10, it is clear that, except at very low resources, the performance of the heuristic optimizer is quite impressive. At the 1% mark, the worst case, the average error was 28%. For all other scenarios, the average error never exceeded 0.2%. For the experiments that we have conducted, the error was never more than 100%, which means that the best throughput for the query was always within a factor of 2 of the optimizer's pick.

7. RELATED WORK

The existence of applications built on streaming information motivated building specialized systems to manage streaming data. Among the recent examples are: Niagara [9], STREAM [29], Aurora [1], and Telegraph [28]. The survey in [5] 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. Sensor networks and databases (e.g., TinyDB [21] and the Cougar [34] project) are also closely related.

The seminal work of [24] introduced a framework for optimization of relational queries aimed at minimizing query completion time. NiagaraCQ [9] aims at addressing the scalability of a system supporting a large number of continuous queries by grouping predicates and queries together. The work in [8][10][22] uses similar techniques by extending the earlier work on eddies [4] 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 [1] 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, and then the system manages them with little, if any, modification. Similarly, the work on scheduling operators in [6][12] 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 [23] together with a brief discussion of solutions. The assumption in such work is that a query optimizer has already arrived at a *best* plan.

The work in [31] advocates moving from cardinality-based optimization to rate-based optimization and provides a model for a rate-based optimizer. Such work is geared towards optimizing queries over finite streaming sources, or short lived queries on infinite streams. It does not model the effect of sliding windows for continuous queries over infinite sources. The work in [32] provides a symmetric multi-join operator for multiple joined streams to minimize memory usage as opposed to using multiple binary join operators. Also close is [26] 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 it 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 if operators are allowed to be duplicated on the different processors.

A lot of work dealt with providing partial answers to continuous queries. In [23], the authors survey a number of methods to arrive at a partial answer, among which is random sampling (i.e., random dropping of tuples) discussed here. The work in [25] provides algorithms for placing drop boxes to reduce resource usage. It explores both random and semantic filtering. The work in [7] also deals with the optimum placement of random filters for multiple aggregation queries sharing operators and resources over data streams. The difference between this body of work and ours is that it does not explore the effect of modifying the query plan to achieve better results. The work in [19] 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 [11], 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 [1].) 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.

8. CONCLUSION AND FUTURE WORK

In this paper, we proposed a framework for static optimization of sliding window conjunctive queries over infinite streams. We illustrated the constrained nature of the optimization problem and proposed different goals for the optimization when computational resources are sufficient and when they are tight. We developed a cost model to estimate the average resource utilization and output rate of a query plan. Using the model, we studied the problem of how to optimally shed load from a query by randomly dropping tuples such that the final output rate is maximized. We demonstrated that the intuition suggesting that the plan to shed load from is the same plan that is selected when resources are sufficient is often times incorrect. We then proposed an optimization algorithm that integrates resource constraints into the optimization process. Finally, we analyzed the need for reoptimization when resources are insufficient. We also analyzed the effectiveness of the proposed optimization algorithm.

In developing a solution for the problem, we have made some simplifying assumptions. There are a number of future directions

to be explored by relaxing those assumptions. We have considered the optimization of single isolated queries. In reality, streaming systems are envisioned to handle multiple concurrent queries, often with significant overlap in their requirements. In this scenario resource sharing between queries is a must. This makes multi-query optimization an immediate extension to our work.

We have also focused on modeling the average steady state rate of arrival for data streams. It might be more interesting to consider the effect of the variance of the rate around its average on the different query plans.

It is an interesting extension to this work to investigate semantic load shedding, in which tuples are smartly dropped based on their data values. It is not clear if the results presented here will hold for the semantic load shedding case.

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.

Finally, a feasible plan that is close to 100% utilization can still have a large response time and buffer requirements. While there is a quick fix to this situation in our model by restricting the actual resources 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 and buffering requirements.

REFERENCES

[1] D. Abadi, D. Carney, et al. Aurora: a new model and architecture for data stream management. The VLDB Journal, Vol.12(2), pp. 120 – 139, 2003.

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

[3] A. Arasu, S. Babu, J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. Technical Report, Department of Computer Sciences, Stanford University, October 2003.

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

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

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

[7] B. Babcock, M. Datar, R. Motwani. Load Shedding for Aggregation Queries over Data Streams. ICDE 2004.

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

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

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

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

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

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

[14] J. M. Hellerstein. Optimization Techniques for Queries with Expensive Methods. TODS 23(2), pp. 113-157, 1998.

[15] Y. Ioannidis and Y. Kang. Randomized Algorithms for Optimizing Large Join Queries. SIGMOD, May 1990.

[16] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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