OM: A Tunable Framework for Optimizing Continuous Queries over Data Streams

Dhananjay Kulkarni[1] **and Chinya V. Ravishankar**[1]

[1] Department of Computer Science and Engineering
University of California, Riverside, CA 92507, U.S.A

{kulkarni,ravi}@cs.ucr.edu

Abstract. *Continuous query (CQ) is an important class of queries in Data Stream Management Systems. While much work has been done on algorithms for processing CQs, less attention has been paid to the issue of optimizing such queries. In this paper, we first argue that parameters such as output rate and main memory utilization are important cost objectives for CQ performance, than disk I/O. We propose a novel framework, called OM to optimize the memory utilization and output rate of CQs. Our technique monitors input stream and query characteristics, and switches plans only at certain boundary conditions. Our approach is tunable to application requirements and enables a user to make the query performance versus optimization overhead trade-off. Experimental analysis shows that our approach has high fidelity in predicting correct plans and is promising in terms of minimizing query scheduling overhead.*

Resumo. *Consulta contínua (CQ, Continuous query) é uma classe importante em Sistemas de Gerenciamento de Streams. Enquanto vários trabalhos propõem algoritmos para processar CQs, pouca atenção tem sido dada à otimização de tais consultas. Nesse artigo, argumentamos que parâmetros tais como a taxa de saída e a utilização da memória principal são métricas mais importantes para o desempenho de CQ do que o acesso ao disco. Dessa forma, nós propomos um novo framework, chamado OM, para otimizar a utilização de memória e a taxa de saída de CQs. Nossa técnica monitora o stream de entrada e as características da consulta, e troca planos somente em certas condições de limite. Nossa proposta é ajustável aos requisitos da aplicação e permite que um usuário compare os benefícios entre o desempenho da consulta e a sobrecarga imposta pela otimização. A análise experimental mostra que a nossa proposta tem alta fidelidade ao prever os planos corretos e é promissora em minimizar a sobrecarga do agendamento de consultas.*

1. Introduction

A sequence of data elements that is continuous, unbounded, and time-varying is called a *data stream* [Babcock et al. 2002]. For example, data transmitted by sensors, stock market data and network monitoring data. Users issue a *continuous*

query (CQ) [Madden et al. 2002] over such data streams and when new data arrives, the Data Stream Management System (DSMS) [Chandrasekaran et al. 2003, Carney et al. 2002, Motwani et al. 2003, Chen et al. 2000, Sullivan and Heybey 1998, Madden and Franklin 2002, Seshadri et al. 1994, Engine 2005] executes the CQ and returns the result. The same query is re-executed when new data elements arrive into the system. This query processing model is significantly different from relational database management systems (RDBMS). A DSMS is also likely to be used in environments where multiple users and queries exist. For example, online bidding systems such as eBay [eBay 2003] and real-time financial search engines such as TraderBot [Traderbot 2003].

We focus on the query optimization problem over data streams. While much work has been done in processing data streams, query optimization is still an important research issue. Earlier works have considered either output rate or memory utilization as the only optimization objective. Optimizing CQs for a single objective is not reasonable in practice, because there could be inter-play of various objectives that affect query performance. Moreover, a DSMS application could have it's own requirements to maximize (or minimize) a particular objective. Thus, we see a need for an optimization technique that addresses multiple cost objectives, particularly when the data stream characteristics are not known apriori. Moreover, the technique should be general enough to be used in practice.

1.1. Data Stream Management Systems

Figure 1 shows the schematic model of a typical DSMS. Each CQ *submitted* by the user remains in the system, until it is explicitly removed by the user. The optimizer (QO) first optimizes the query and finds a minimum cost *plan*. Next, the scheduler (QS) finds a *schedule* for the plan and prepares it's operators for execution. When new tuples arrive, the tuple dispenser (TD) dispatches the enqueued tuples to the ready operators. We assume there is a limited main memory M_{sys} available for executing a CQ. Each query execution produces a stream of tuples as output. Output rate of a query refers to the number of result tuples produced per unit time. The unprocessed and intermediate data tuples are buffered in-memory for further processing. As we will show later, both, memory utilization and output rate are important parameters for CQ processing.

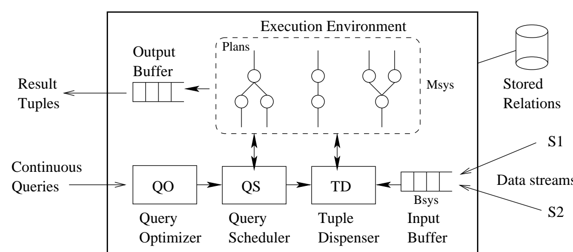


Figure 1. Schematic Model of a DSMS

1.2. Motivation and Problem

There exist significant differences between optimizing queries over relation data versus optimizing CQs over data streams. Since relational queries are executed over stored rations, most RDBMS optimizers [Astrahan et al. 1976] estimate the query execution cost in terms of disk I/O. By contrast, a CQ is executed multiple times over a sequence of tuples that arrive continuously. Moreover, a CQ is evaluated by directly accessing the tuples (or their summaries) from main memory. Thus, disk I/O is not a dominant cost factor in a DSMS, unless the DSMS supports queries over stored relations as well. In this work we only consider CQs over data streams.

Example: Consider a CQ with operators o_1 and o_2. Let selectivity of o_1, o_2 be 0.5, 0.1, respectively. Let the processing rate (measured in tuples per sec) of operators o_1, o_2 be 500, 50, respectively. As shown in Figure 2 consider the two candidate plans for the CQ. Since the memory utimization of of plan 2 is less than that of plan 1, a RDBMS heuristic (to push down highly selective operations) would choose plan 2. By contrast, a rate based optimization technique [Viglas and Naughton 2002] (to maximize output rate) would choose plan 1. Thus, we observe that a DSMS optimizer has other costs to worry about, than only disk I/O.

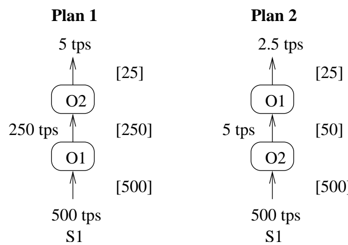


Figure 2. Cost parameters associated with a CQ

Next, we discuss various parameters (cost objectives) that affect CQ performance.

- C1 **Input buffering**: Input buffering is associated with tuples that have already arrived, but not been consumed by any query. If input buffer becomes full, some arriving tuples would have to be dropped, thus affecting *result accuracy*.
- C2 **Output rate**: Output rate of a query plan refers to the number of tuples produced by the CQ per unit time.
- C3 **Intermediate queue length**: Every query operator produces intermediate tuples at a certain rate. However, if these tuples are not consumed fast enough, then the intermediate queues eventually overflow and some tuples have to be dropped.
- C4 **Re-computation overhead**: If two or more queries perform a common operation, the DSMS can save significant processor cycles by sharing the computation. Moreover, saving on re-computation means faster overall tuple consumption and lower input buffer requirement.
- C5 **Latency of query evaluation**: Latency is time between the the query submission and the time when the output tuples are produced.

Most DSMS applications have their own requirements for query optimization. For example, an application to monitor temperature and pressure, may require a high result accuracy. On the contrary, a stock market application may require a higher output rate. This represents a multi-objective optimization problem [Fonseca and Fleming 1995] and motivates us to combine the two objectives in some way. Earlier papers [Viglas and Naughton 2002, Babcock et al. 2003b, Avnur and Hellerstein 2000] have either studied only 1 cost objective or studied each one in isolation. We wish to address the problem of optimizing mutiple cost objectives, particularly output rate and memory utilization.

Static query plans become sub-optimal when the data stream characteristics change as a result of newly arriving data. We identify two extremes to addressing this problem. First, the approach presented in [Babcock et al. 2003b] has low complexity, but assumes that operator selectivity remains static. Second, we believe that per-tuple optimization proposed in [Avnur and Hellerstein 2000] suffers very high computational and storage overhead. We propose a optimization framework that strikes a balance between the two extremes stated above.

1.3. Approach

We propose a utility function-based technique to combine multiple cost objectives, specifically output rate and memory utilization. Since frequent re-optimization of CQs is a significant overhead, we propose spending extra time during the initial optimization step itself. Our optimization technique generates multiple plans and switches plans only at certain boundary conditions. Thus our technique avoids re-optimization when the query is in-flight and is able to qucikly adapt to changing stream characteristics. Since the overhead due updating query plans and query performance are two conflicting goals, we make our approach tunable to the application requirements. This helps make the appropriate trade-off between query performance and optimization overheard.

2. Related Work

[Avnur and Hellerstein 2000, Chandrasekaran et al. 2003] provides an alternative to query optimization by introducing an adaptive query operator, called Eddy. The ticket-based policy used for Eddy is limited to optimizing a single objective and suffers high overhead. Work presented in [Deshpande 2004] batches tuples to reduce the Eddy overhead, but it does not address multiple cost objectives. [Kang et al. 2003, Viglas et al. 2002, Viglas and Naughton 2002] proposes a rate based optimization technique for CQs, but assumes that operator selectivity remains static and that stream arrival rates are known apriori. [Babcock et al. 2003b, Babcock et al. 2003a] provides efficient techniques to handle bursty data streams, however, this approach is limited to minimizing only memory utilization and suffers from starvation. Recent work [Ayad and Naughton 2004] is very much applicable to our problem, however it assumes that arrival rates are constant and query can be updated while it is already in-flight.

Our work is novel because it avoids frequent re-optimization of plans, addresses multiple cost objectives, and provides a framework to address an application's requirements for query performance.

3. Problem Formulation

Let R be the schema for a relation with one more more attributes. Let T be the time domain. We represent a data stream as a growing sequence of tuples, which are time-stamped on arrival. Tuples that have already arrived can be represented as a *bag* $s = \{a_{t_0}, a_{t_1}, \ldots, a_{t_n}\}$, where each element a is consistent with relational schema R, $t_i \in T$, and t_n is the current time instant. In our work we assume that the arrival rate and data distribution of the data stream are not known apriori.

We abstract a query using $Q[W]$, where Q is a query composed of one or more selection (σ), projection (π), join ($\bowtie$) and aggregation (Ω) operators. W is the *window* (or the time-interval) that defines the subset of tuples to be processed per execution. During execution, each query consumes data tuples that arrive within the time interval [t_n-W, t_n]. For each successive execution of the query we *slide* this window over the data stream. Let the query Q be composed of a set of k *operators* $\{o_1, o_2, \ldots, o_k\}$, where each $o_i \in \{\sigma, \pi, \bowtie, \Omega\}$. A *plan* p for a query Q is a directed acyclic graph consisting of k nodes that correspond to the k operators. Let P be the set of all possible candidate plans for the query Q. The goal of the optimizer then is to choose a plan $p_i \in P$, based on a cost function. We explain our cost functions in Section 4.1.

Some statistical information (or meta-data) is required to estimate query costs. We define *selectivity* as the fraction of the number input tuples produced as output of the operator. We define *tuple processing rate* as number of tuples that the operator can process per unit time. We define the *input arrival rate of an operator* as the average arrival rate of the input streams. The operator selectivity (*sel*), the operator tuple processing rate (*tpr*) and the input arrival rate (λ) are maintained as meta-data for each operator. The optimizer uses this meta-data to estimate the cost of executing the query. The *tpr* is a constant for each operator type. The *sel* and λ values are maintained online as described in Section 4.3.

3.1. Our Query Processing Model

Let P be the set of candidate plans for a query Q. Consider a plan $p \in P$ chosen for execution. Our processing model assumes that the query execution is atomic and the operators $\{o_1, o_2, \ldots, o_k\}$ are pipelined during execution. In such a model, the input tuples and intermediate tuples are required to buffered in-memory.

When an operator o_i is scheduled for execution it performs the following steps. First, it *reads* the input tuples (of size 1 window) waiting at the input queues. Second, it *processes* the tuples by executing the operator functions (or routines). Third, it *produces* the result tuples by writing them to the output queue. This *read-process-produce* sequence

is repeated when more data arrives into the system. A *round* refers to one complete execution of the read-process-produce steps. We use the notation $round_j$ to denote the jth execution of the query. As each CQ remains in the system for a long time, the total number of rounds for a query are expected to be high. For example, a query with $W = 5$ mins will make 288 rounds in 1 day.

Example: Let Q be a query consisting of a selection σ=$s.a_1 \geq 10$ and a projection $\pi = s.a_1$, where the stream s has integer attributes (a_1, a_2). Let the window W be 20 seconds. Assume that the data stream tuples arrive with the following time-stamps, $(10,1)_1$, $(7,2)_5$, $(8,3)_{10}$, $(25,4)_{15}$, $(10,5)_{20}$, $(8,6)_{25}$, $(9,7)_{30}$, $(26,8)_{35}$, $(11,9)_{40}$, $(7,10)_{45}$, $(20,11)_{50}$. Let the 2 operators be executed in the sequence: $\sigma \rightarrow \pi$. Thus, the following outputs are seen at the end of each round, $round_1 = \{10, 25\}$, $round_2 = \{26, 11\}$, $round_3 = \{20\}$.

3.2. Our Cost Model

We address 2 cost objectives, namely the output rate (C_r) and the memory utilization (C_m). The query execution cost is a function of C_r and C_m. We measure the query cost (in terms of *utils*) using a utility function [Neumann and Morgenstern 1944]. This utility function, written as $Cost(C_r, C_m)$ is the weighted cost of the utility of the output rate and memory utilization of the query. Thus,
$Cost(C_r, C_m) = w_r \times \frac{1}{C_r} + w_m \times C_m$, where w_r, w_m are user-specified weights.

Our cost model is based on the policy: *the query performance improves when the output rate is higher and when the memory utilization is lower*. From an application perspective, a higher w_r helps in choosing a plan with higher output rate and a higher w_m helps in choosing a plan with lower memory utilization. Once the weights are specified, the goal of our optimizer is to choose a plan with minimum $Cost(C_r, C_m)$. For brevity, we use $Cost_p$, to denote the weighted cost of plan p.

Example: We use the query in Figure 2 and show that different weight assignments can infuence the choice of plans. We first find the individual costs for output rate and memory utilization. For p_1, we have $C_r = 5.0$ and $C_m = 775$. For p_2, we have $C_r = 2.5$ and $C_m = 575$. Consider two weight assignments as shown below.
Assign 1:$\{w_r = 0.1, w_m = 0.9\}$, Assign 2:$\{w_r = 20000, w_m = 2\}$
The weighted query cost for the two assignment are as follows.
Assign 1: $Cost_{p_1} = 697.52$, $Cost_{p_2} = 517.54$
Assign 2: $Cost_{p_1} = 5550$, $Cost_{p_2} = 9150$
Thus, The optimizer chooses plan 2 in assignment 1. On the contrary, the optimizer chooses plan 1 in assignment 2.

3.3. Problem Statement

Given a continuous query Q over a set of data streams S, find the query plan p from it's set of candidate plans P, such that $Cost_p$ is minimum.

4. Our Approach

We first describe our technique to generate plans. Next, we present our approach to monitor and switch plans in an enviroment where the data stream charactertics are dynamic.

4.1. Cost Functions for Output Rate and Memory Utilization

A selection operation selects all the tuples that satisfy the given condition. Projections throw away the unwanted attributes and retain the attributes which appear in the projection list. Since we do not use any special access methods, the selection, projection and aggregate operations require 1 sequential scan of tuples in the current window. We consider only 2-way joins and use L and R to indicate the left and right input streams of a join. The cost of processing a sliding window join is determined in two parts, namely joining the new incoming tuples in left window L_W with all the tuples in the right window R_W, and vice-versa. Thus, the join requires 1 scan of the tuples in L_W and R_W window.

Based on the above processing model, we derive the cost functions. Let C_σ and C_π be the time needed to do selection and projection operation, respectively. Let $C_L \bowtie$ be the time needed for joining 1 tuple with left window, $C_R \bowtie$ be the time needed for joining 1 tuple with right window. Let C_Ω be the time needed for aggregating 1 tuples.

The **output rate** of the various operators can be estimated as follows. For selection, $sel \times \lambda$. For Projection, λ. For Aggregates, $\frac{1}{C_\Omega \times \lambda \times W}$. For Join, $\frac{sel \times \lambda_L \times \lambda_R \times W}{\{C_L \bowtie \times \lambda_L\} + \{C_R \bowtie \times \lambda_R\}}$.

While an operator is processing the current window, the number of tuples that get buffered at **input buffer** is $c_{inp} = \lambda \times \{\frac{\lambda \times W}{tpr}\}$. The number of tuple buffered at the **intermediate queue** (output of operator) is calculated as $c_{int} = c_r \times W$. Since both, input buffering and the intermediate queue buffering contribute toward the operator's **memory utilization** c_m, we have $c_m = c_{inp} + c_{int}$.

Let Q be a query composed of k operators $\{o_1, o_2, \ldots, o_k\}$. We assume that the operators in the plan are labeled in increasing order from 1 to k, using a breadth-first traversal of the query tree. Hence, the root operator is labeled o_1. Let $c_r[o_i]$ and $c_m[o_i]$ be the output rate and memory utilization of o_i, respectively. Since the query output rate is same output rate of the root operator, we have $C_r = c_r[o_1]$. Since the query memory utilization is the sum of the memory utilization of it's operators, we have $C_m = \sum_{i=0}^{k} c_m[o_i]$. The **weighted cost of the query** is determined by substituting for C_r and C_m in the function $Cost(C_r, C_m)$, described in section 3.2.

4.2. Optimization Heuristics

Given a query Q, we apply steps 1-4 for generating the query plan.
(1) Group selections: We group all selections in Q that have predicates over the same stream. For example, $S.a > 5$ OR $S.a < 10$, is grouped as $5 < S.a < 10$.
(2) Push down projections and selections: We push down the π and σ operators to the bottom of the query plan after examining the cost of the two sub-plans, namely $\pi \rightarrow \sigma$ and $\sigma \rightarrow \pi$.

(3) Left-deep join ordering: We perform a local search to generate a left-deep ordering of the joining streams. For example, if the query joins s_1, s_2, s_3, we first choose 2 streams to be joined, say $J1 \leftarrow (s_1 \bowtie s_2)$. The output of $J1$ is considered as left input for the next join $J2 \leftarrow (J1 \bowtie s_3)$. The output of sub-plans generated for s_1, s_2 and s_3 in step 2, serve is inputs to the $J1$ and $J2$. The cost of the join query is the total cost of performing $((s_1 \bowtie s_2) \bowtie s_3)$.
(4) Aggregate last: We apply the aggregate operations at the top of the query tree.

4.3. Maintaining Online Statistics

Since the arrival rate and selectivity of operators are not known apriori, we estimate them by finding their weighted sample mean. Consider selection operator's selectivity. First, we record the previous d values of the operator selectivity as samples. These can be observed at the end of each round of query execution. Next, we estimate the selectivity at some round R as $sel_e(R) = \frac{\sum_{r=0}^{d} F(R-r)G(r)}{\sum_{r=0}^{d} G(r)}$, where $F(r)$ is the observed selectivity at round r and $G(r) = e^{-ar}$ is a *decay* function used to weight the sample values.

Join selectivity is determined using a histogram based approach [Poosala 1997]. We monitor the join selectivity at each round and re-estimate the selectivity only when observed and estimated values differ by a certain threshold.

Similarly, we estimate arrival rate as $\lambda_e(R) = \frac{\sum_{r=0}^{d} F(R-r)G(r)}{\sum_{r=0}^{d} G(r)}$, where $F(r)$ is the the observed arrival rate of the stream at round r. Next section describes how these estimations are useful in scheduling the query plans.

4.4. Generating and Indexing Plans using Charts

Frequent re-optimization of a query is cost prohibitive, especially, if the data stream statistics change very often. We propose a technique that spends extra time during the initial optimization step, rather than worrying about re-optimization when the query is in-flight. We first divide the space defined by the optimization parameters into *regions*. Next, we generate 1 *optimal* plan p_i per region g_i, which represents the minimum cost plan in that region. As long as the data stream statistics lie within the region g_i, we schedule plan p_i for execution. If the system state moves to a new region g' due to a change in the stream statistics, we schedule plan p', where p' is the plan determined to be optimal within region g'. As the plans in each region are generated apriori, the complexity of switching plans and scheduling them is very low.

4.4.1. sel-λ chart

A sel-λ chart is a 2 dimensional chart, with dimensions selectivity sel and arrival rate λ. We build the chart by first performing a k-regular partitioning in each dimension. The space enclosed by consecutive partitions is called a region and is denoted by $G(i, j)$,

where indexes i, j refer to the ith and j^{th} partitions in the space defined by sel, λ, respectively. The centroid $C(i, j)$ of region $G(i, j)$ represents the mean value of the sel and λ within region $G(i, j)$. The (sel_i,λ_j) values at point $C(i, j)$ are used as the parameter values when determining the cost of the plans in region $G(i, j)$. Let the set P be set of candidate plans for the query. The plan $p(i, j) \in P$ denotes the plan that is optimal within region $G(i, j)$. The algorithm to construct the sel-λ chart is shown in Algorithm 1. We maintain 1 chart per stream, hence the storage overhead is of the order $O(nk^2)$, where k is the number of partitions and n is the number of streams.

Algorithm 1 Create sel-λ chart

Require: $[sel_{min}, sel_{max}], [\lambda_{min}, \lambda_{max}], k, P$
Ensure: $p(i, j), \forall 1 \leq (i, j) \leq k$
1: Perform k regular partition of dimensions sel and λ, with partitions in each dimension at $(i \times (sel_{max} - sel_{min})/k)$, and $(j \times (\lambda_{max} - \lambda_{min})/k)$, where $1 \leq i, j \leq k$.
2: Let $G(i, j)$ be the region enclosed by i^{th}, $(i - 1)^{th}$ partition in dimension sel and j^{th}, $(j - 1)^{th}$ partition in dimension λ.
3: **for all** i, j such that $1 \leq (i, j) \leq k$ **do**
4: Let $C(i, j)$ be the centroid of region $G(i, j)$, with the coordinates (sel_m, λ_m).
5: Using sel_m and λ_m determine the $Cost_p$ of each candidate plan $p \in P$.
6: $p(i, j) \leftarrow$ Minimum cost plan $p \in P$.
7: **end for**

4.4.2. λ-λ chart

For join queries, we construct a l dimensional λ-λ, where l is the number of joining streams. The ith dimension refers to the space defined by arrival rate λ_i of stream s_i. The procedure to create this chart is similar to the algorithm presented for the sel-λ chart, except the fact that the centroid represents a point in l dimensional space. Thus, a plan $p(i_1, \ldots, i_l)$ denotes the plan that is optimal in the region $G(i_1, \ldots, i_l)$. Here we assume that the selectivity remains constant within each region.

4.5. Dynamic Query Scheduling using Charts

The charts are constructed during the initial optimization of the query and used during query scheduling. At the beginning of each round r_i the optimizer refers to the charts and determines which plan to schedule. Let sel_e^r and λ_e^r be the estimated values for selectivity and arrival rate at round r. Let the sets OBS_s^{r-1}, OBS_λ^{r-1} represents an ordered set of $d - 1$ observed values of selectivity and arrival rate. The values sel_o and λ_o refer to the observed values in round r_{i-1}. The Algorithm 2 shows how a a plan is picked based on the region that bounds the point (sel_e^r, λ_e^r). After plan p is scheduled and executed, the d observed values are used to estimate the selectivity and λ for round r_{i+1}. The $Estimate()$ function in the algorithm follows the procedure described in Section 4.1. It is easy to see

that, if the stream statistics remain within the region $G(i,j)$, the same plan p can be scheduled for the successive execution of the query.

Algorithm 2 Schedule Plan for Execution

Require: Q[W], sel-λ chart, λ-λ chart, sel_e^r, λ_e^r, d, OBS_s^{r-1}, OBS_λ^{r-1}
Ensure: Results of executing Q, sel_e^{r+1}, λ_e^{r+1}
1: Using $g = (sel_e^r, \lambda_e^r)$ find the region $G(i,j)$ that forms a bounding box for the point g.
2: Let $C(i,j)$ be the centroid of the region $G(i,j)$.
3: Let $p \leftarrow p(i,j)$ the plan indexed at centroid $C(i,j)$
4: Schedule and execute p for one round, over [W] tuples.
5: $(sel_o^r, \lambda_o^r) \leftarrow$ Observed values of selectivity and arrival rate at the end of the executing plan p.
6: $sel_e^{r+1} \leftarrow$ Estimate (sel_o^r, OBS_s^{r-1}).
7: $\lambda_e^{r+1} \leftarrow$ Estimate (λ_o^r, OBS_λ^{r-1}).
8: Update #moves, #switches and fidelity (%hits) for round r

5. Experimental Analysis

Table 1 compares the features of our work **OM** with some recent techniques; **Eddy** refers to the work described in [Avnur and Hellerstein 2000], **Rate** is the approach presented in [Viglas and Naughton 2002], and **Chain** is scheduling algorithm described in [Babcock et al. 2003b]. Our work is able to maintain fairness, because we do not prioritize the query schedule based on memory requirements. Our approach is starvation-free because we process multiple queries in a round-robin fashion. Moreover, our approach is tunable because it enables the user to choose the granularity (of regions in chart) used during scheduling algorithm.

Table 1. Comparison with recent work

	OM	**Eddy**	**Chain**	**Rate**
Optimization technique	Dynamic	Dynamic	Dynamic	Static
Optimization granularity	Query	Tuple	Operator	Query
Addresses Memory utilization	Yes	Yes	Yes	No
Addresses Output rate	Yes	No	No	Yes
Tunable to user-requirement	Yes	No	No	No
Fairness in query scheduling	Yes	Yes	No	n/a
Addresses starvation problem	Yes	Yes	No	n/a

5.1. Setup

We used the DEC-PKT dataset [Archive 2005] as real-world dataset. For single stream queries we worked with the TCP trace and for join queries we used the TCP, UDP and SF

traces, with suitable equi-join conditions. As the DEC-PKT data is quite erratic, we applied a filtering step to generate a series of datasets called REAL, each having a different coefficient of variation (for the arrival rate). The synthetic dataset URND is the uncorrelated random data, which was generated using functions available in GNU compilers.

The experiments were conducted using the OM prototype that is developed as part of our work. Currently, we can optimize queries and maintain various query and data statistics. The queries were executed as per the processing model described in section 3.1.

Our main focus was to study *fidelity*, defined as the fraction of times we correctly predict the optimal plan. We consider it a *hit* if the plan (est_plan) chosen by our optimizer is same as the optimal plan (opt_plan) determined in an offline procedure. We consider it a *move* whenever the data stream characteristics change and the stream statistics crosses a region boundary (defined for the charts discussed earlier). A move does not necessarily imply scheduling a new query plan, because two adjacent regions may index the same plan. We consider it a *switch*, if the plan executed in $round_i$ if different from the plan executed in $round_{i+1}$.

We executed the common query operators in isolation and found the tuple processing rates (measured in tuples per sec) to be the following: Selection Equality=400 $\times$ 10^3 tps, Selection Range=200 $\times$ 10^3 tps, Projection=500 $\times$ 10^3 tps, Join Equi=3 $\times$ 10^3 tps. We use the following default values, unless stated otherwise. Weights ($w_r = 0.4, w_m = 0.6$), number partitions in each dimension $k = 10$. Each continuous query is run for 10 mins.

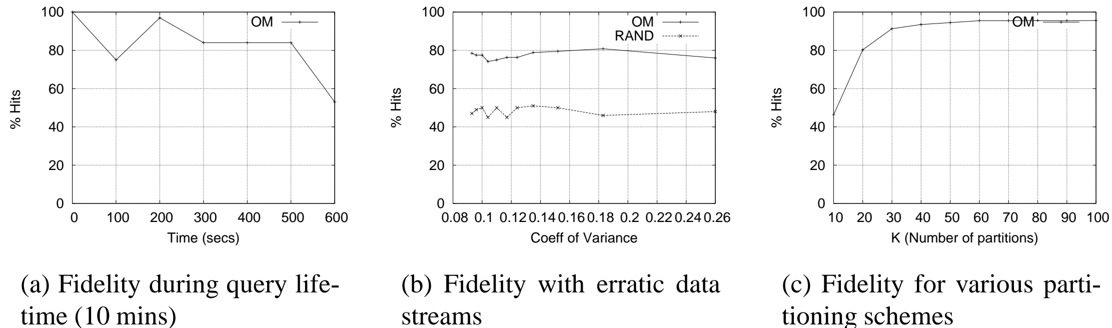
Table 2. Queries Used in our Experiments

Single-stream query Q1	**Multi-stream query Q2**
SELECT TCP.src_host, TCP.dest_host FROM TCP WHERE TCP.dest_host=2 WINDOW [1]	SELECT * FROM TCP, UDP, SF WHERE TCP.src_host=UDP.dest_host and UDP.dest_host=SF.dest_host WINDOW [1]

5.2. Single Stream Queries

We used the query $Q1$ shown in Table 2 to model single stream experiments. Since this plan has one selection and one projection operation, there are only 2 possible candidate plans. Next, we describe the results for single stream queries.

Comparison with Optimal plans: We ran $Q1$ over the DEC-PKT dataset and monitored the number of times we were able to choose the correct plan. Figure 3(a) shows the fidelity, averaged per 100 seconds. Our estimations worked very well, except when there was a sudden burst of inputs. During such a burst (for example between 500-600 seconds), the optimizer scored wrong hits ($opt_plan \neq est_plan$). Our optimizer



(a) Fidelity during query lifetime (10 mins)

(b) Fidelity with erratic data streams

(c) Fidelity for various partitioning schemes

Figure 3. Experimental Results for Single-stream Queries

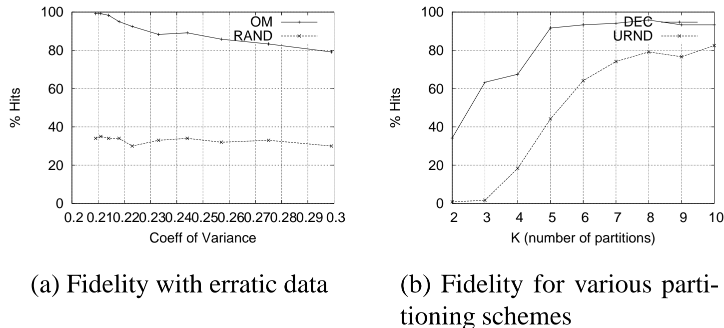
performed with an average fidelity of 80% in this experiment, in which there were 84 moves and 23 switches.

Querying Erratic Data Streams: We ran the query $Q1$ over data streams that had coefficient of variation (for λ) ranging from 0.26 to 0.09. As shown in Figure 3(b), the fidelity is slightly lower for data streams that are more erratic. The fidelity is in the range [76 %, 81 %], with most experiments having a fidelity of 80%. In the random approach (RAND), a random plan is picked from the set of possible candidate plans, in this case just 2 plans. As expected, the RAND approach cannot do better than 50% for single stream queries. The RATE approach uses static values for selectivity and arrival rate, hence does not switch plans when required. In fact, the RATE approach has a lower fidelity, because it suffers a wrong hit every-time there is a switch.

Performance v/s Optimization Overhead: We measure the optimization (storage) overhead in terms of number of plans generated while constructing the charts. The parameter k abstracts our notion of overhead, because a chart has $k \times k$ partitions, hence k^2 plans are indexed in the chart. As shown in Figure 3(c), the fidelity increase as we go from k=10 to 40, and then stabilizes. The observation is that, when we have more partitions, we have better control over when to switch between query plans. For lower k, multiple plans could be optimal within the same region, hence the optimizer might miss some plans. The fidelity saturates around 95 %, which implies that the performance gain is negligible after this point. Since we check for optimal plans only at the end of each round, we make only 600 *optimization decisions*. Since the average the arrival rate is 500 tuple/sec, the per-tuple EDDY approach would make 300000 decisions, and the modified EDDY approach [Deshpande 2004] would make 1500 decisions, both of which are significantly higher than our approach.

5.3. Multi-Stream Queries

We used the join query $Q2$ shown in Table 2 for the multiple stream experiments. The 3 streams in $Q2$ can be joined in 6 ways. However, we consider only left-deep ordering, so



(a) Fidelity with erratic data

(b) Fidelity for various partitioning schemes

Figure 4. Experimental Results for Multi-stream Queries

we have to deal with only 3 candidate plans.

Querying Erratic Data Streams: As shown in Figure 4(a), we do very well with 99 % fidelity when the data is not very erratic. This is also intuitively correct, because as the data becomes smoother we should be able to predict the stream properties (and hence the plans) more accurately. In fact, even when the data is erratic we are not below 80% fidelity. This shows that our estimation and optimization approach is consistent and always does better than the RAND approach. The RATE approach is not dynamic, hence the initial plan quickly becomes sub-optimal. The RAND plans perform close to 30% fidelity, because 1 plan is randomly picked from 3 candidate plans.

Performance v/s Optimization Overhead: We use DEC and URND dataset in this experiment. We vary k from 2 to 10. In case of the λ-λ there are 3 dimensions and hence the optimization overhead is of the order $O(k^3)$. As shown in figure 4(b), we always get a higher fidelity for the DEC dataset than the URND dataset. This is mostly because the tuples in URND are completely uncorrelated. The fidelity stabilizes at a saturation point (for DEC:k=5 and for URND:k=8) beyond which we are not likely to see much performance benefit. Since both datasets had arrival rate $\geq$ 500 tuples per second, we see that the per-tuple EDDY approach would make at least $500 \times 60 \times 10$ optimization decisions in the query lifetime of 10 mins.

5.4. Scheduling Multiple Queries

Let N_q be the number of queries registered with the DSMS. We maintain a queue of ready CQs and process them in a round-robin fashion. The scheduling is fair because the queue is not sorted after each round. Moreover, this approach is starvation-free because each query is serviced in FIFO fashion and reinserted at the back of the queue. The query has to wait in the queue till the next round of execution. We experimented with 10 queries, with a mix of single stream and join queries. We ran the simulation over the REAL dataset and found that our approach showed consistent performance. Single stream queries experienced a fidelity of 95 % and the join queries had a fidelity of 90 %.

6. Conclusions

We have provided a novel framework to perform continuous query (CQ) optimization in data stream environments. We have shown that CQ performance is influenced by parameters such as output rate and memory utilization of queries, rather than just disk I/O. Our important contribution has been to propose a cost model that is tunable to the application requirements. Moreover, our technique to generate multiple plans and schedule them using charts is quick and low in complexity. A chart with higher granularity provides better query performance, because the optimizer can monitor and switch plans more frequently.

Acknowledgments: This project was supported by a grant from Tata Consultancy Services, Inc. The authors would like to thank Mirella Moro for the abstract translation.

References

Archive, I. T. (2005). Ita homepage, http://www.acm.org/sigcomm/ita.

Astrahan, M. M., Blasgen, M. W., Chamberlin, D. D., Eswaran, K. P., Gray, J., Griffiths, P. P., III, W. F. K., Lorie, R. A., McJones, P. R., Mehl, J. W., Putzolu, G. R., Traiger, I. L., Wade, B. W., and Watson, V. (1976). System r: Relational approach to database management. *TODS*, 1(2):97–137.

Avnur, R. and Hellerstein, J. M. (2000). Eddies: Continuously adaptive query processing. In *Proceedings of ACM SIGMOD*, volume 29, pages 261–272. ACM.

Ayad, A. M. and Naughton, J. F. (2004). Static optimization of conjunctive queries with sliding windows over infinite streams. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 419–430, New York, NY, USA. ACM Press.

Babcock, B., Babu, S., Datar, M., Motwani, R., and Thomas, D. (2003a). Operator scheduling in data stream systems.

Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. (2002). Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16.

Babcock, B., Babu, S., Motwani, R., and Datar, M. (2003b). Chain: operator scheduling for memory minimization in data stream systems. In *Proceedings of ACM SIGMOD*, pages 253–264. ACM Press.

Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., and Zdonik, S. (2002). Monitoring streams - A new class of data management applications. Technical Report CS-02-04, Department of Computer Science, Brown University.

Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., and Shah, M. (2003). Tele-

graphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of CIDR*.

Chen, J., DeWitt, D. J., Tian, F., and Wang, Y. (2000). NiagaraCQ: a scalable continuous query system for Internet databases. In *Proceedings of ACM SIGMOD*, pages 379–390.

Deshpande, A. (2004). An initial study of overheads of eddies. *SIGMOD Rec.*, 33(1):44–49.

eBay (2003). ebay homepage, http://pages.ebay.com.

Engine, S. S. P. (2005). Streambase systems inc. homepage, http://www.streambase.com/.

Fonseca, C. M. and Fleming, P. J. (1995). An overview of evolutionary algorithms in multiobjective optimization. *Evolutionary Computation*, 3(1):1–16.

Kang, J., Naughton, J. F., and Viglas, S. D. (2003). Evaluating window joins over unbounded streams. In *Proceedings of ICDE*.

Madden, S. and Franklin, M. J. (2002). Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of ICDE*.

Madden, S., Shah, M. A., Hellerstein, J. M., and Raman, V. (2002). Continuously adaptive continuous queries over streams. In *Proceedings of SIGMOD*.

Motwani, R., Widom, J., Arasu, A., Babcock, B., Babu, S., Datar, M., Manku, G., Olston, C., Rosenstein, J., and Varma, R. (2003). Query processing, resource management, and approximation in a data stream management system. In *Proceedings of CIDR*.

Neumann, J. V. and Morgenstern, O. (1944). Theory of games and economic behavior.

Poosala, V. (1997). *Histogram-based selectivity estimation for query optimization*. PhD thesis, University of Wisconsin.

Seshadri, P., Livny, M., and Ramakrishnan, R. (1994). Sequence query processing. In *Proceedings of ACM SIGMOD*, pages 430–441.

Sullivan, M. and Heybey, A. (1998). Tribeca: A system for managing large databases of network traffic. In *Proceedings of USENIX Annual Technical Conference*, pages 13–24.

Traderbot (2003). Traderbot homepage, http://traderbot.com.

Viglas, S. and Naughton, J. F. (2002). Rate-based query optimization for streaming information sources. In *Proceedings of SIGMOD*.

Viglas, S., Naughton, J. F., and Burger, J. (2002). Maximizing the output rate of multi-join queries over streaming information sources. In *Proceedings of VLDB*.