WhiteWater: Distributed Processing of Fast Streams

Ioana Stanoi, George A. Mihaila, Themis Palpanas, and Christian A. Lang, *Member*, *IEEE*

Abstract—Monitoring systems today often involve continuous queries over streaming data in a distributed collaborative fashion. The distribution of query operators over a network of processors, as well as their processing sequence, form a query configuration with inherent constraints on the throughput that it can support. In this paper, we discuss the implications of measuring and optimizing for output throughput, as well as its limitations. We propose to use instead the more granular *input throughput* and a version of throughput measure, the *profiled input throughput*, that is focused on matching the expected behavior of the input streams. We show how we can evaluate a query configuration based on profiled input throughput and that the problem of finding the optimal configuration is NP-hard. Furthermore, we describe how we can overcome the complexity limitation by adapting hill-climbing heuristics to reduce the search space of configurations. We show experimentally that the approach used is not only efficient but also effective.

Index Terms—Query processing, optimization, database architectures, distributed applications.

1 Introduction

Monitoring is increasingly used in various applications such as business performance analytics, RFID tracking, and analyzing signals from financial indicators and strategies. A continuous monitoring query can be deployed in various configurations, where some are better than others with respect to optimization criteria such as latency, work, and throughput. In many monitoring applications that we worked with, events are emitted, stored, and processed by different layers. In such a stratified processing topology, we can identify at least two opportunities for optimization. First, we can make use of the inherent distributed topology to reorganize the processing operators in a manner that will improve the overall throughput. Second, each of the processing layers can, in turn, be distributed on to clusters of physical nodes for scalability. In both cases, different orderings of operators and different mappings of operators to physical nodes lead to different upper bounds for the supported throughput. Consider the following examples.

Example 1: Distributed Business Process Monitoring. Oftentimes, a company describes its business actions at various levels of granularity with the use of a workflow concept called a business process. A process is composed of subprocesses such as credit check and, to be more granular, tasks (such as a task within the credit check). Streams of events provide real-time information that is processed, analyzed, and aggregated while crossing different layers of abstraction: from the lower IT layer to the highest business layer. Queries can span more than one such layer, whereas the processing itself is enabled by multiple components such as event bus, various correlation engines, and dedicated monitors. With a high number of concurrent instances of each process, each instance emitting a large numbers of events, the monitoring components can easily become bottlenecks in the system. To relieve their processing load, some filtering and aggregation should be executed at different places in the network, depending on the resources available. A challenge in this case is how we can find a configuration that places operators in the network in such a way that the system can handle the highest input rate possible and avoid processing backlogs.

Example 2: Load Balancing on Clusters of Processors. Granular state reporting of hundreds of instances of business processes (for example, e-commerce orders) can easily generate thousands of events per second. At such high rates, event streams can be efficiently processed by more than one processor, placed in a blade center. Intrablade and interblades load balancing is a major requirement in optimizing the processing of the input flow. Again, this is the same in-network query optimization problem as in the previous example, but in this case, edge latency can vary if chips are on the same blade or on different blades. As in the previous application example, a critical requirement is that the system should be able to support peak-high input rates.

Our work targets a distributed monitoring system of continuous queries processed by a network of components, each capable, to some extent, to execute stream operators over incoming data. Capabilities and resources naturally vary from one component to another. Processing components may already be running other tasks, and, based on local resource limitations and requirements, decide on a resource capacity that they can offer for the processing of

- I. Stanoi is with IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120. E-mail: irs@us.ibm.com.
- G.A. Mihaila and C.A. Lang are with IBM T.J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532. E-mail: {mihaila, langc}@us.ibm.com.
- T. Palpanas is with the Department of Information and Telecommunications, University of Trento, Via Sommarive 14, 38050 Povo, TN, Italy. E-mail: themis@dit.unitn.it.

Manuscript received 3 Aug. 2006; revised 7 Feb. 2007; accepted 2 Apr. 2007; published online 1 May 2007.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0375-0806. Digital Object Identifier no. 10.1109/TC.2007.1056.

1041-4347/07/$25.00 © 2007 IEEE Published by the IEEE Computer Society

monitoring operators. Although the processing components are fairly stable, data can arrive at the system at various rates, and the system is required to support these rates. We have to find an execution plan that spreads the query operators across those layers and makes efficient use of all the available resources. The query configuration should ensure that we can handle the input data rates and avoid the need for load shedding as much as possible. Finding the optimal configuration is not trivial: Exhaustively enumerating and analyzing all the equivalent rewritings and operator placement options for an example query of 15 operators (five joins) and 11 physical nodes (each operator having only two placement options) can take up to 16 hours on a 1.6-GHz machine.

Each operator of a continuous query requires a certain amount of execution time for every incoming data tuple, which leads to an upper bound on the rate at which tuples can be processed. If the input streams exhibit higher rates than the query that the operators can process, then special mechanisms need to be in place to handle them. When high input rates represent only short bursts, buffers can be used to temporarily store the overflow of incoming data. If instead, the high rates have to be supported for a long period of time, then data need to be purged from the input to the operators. This approach cannot avoid the deterioration of the quality of query results. There has been a large body of recent research that focused on which events to shed in order to return a high-quality result. However, some loss of quality is unavoidable when information is discarded. For some applications, any event may contain critical information, and the reduction in the quality of results should be minimized.

In this paper, we focus on a problem complementary to that of load shedding: finding a query configuration that, given resource and quality constraints, can successfully process the highest incoming stream rates. This translates into finding an order of query operators and a placement of the operators on physical nodes that maximize throughput. The benefit is that even with load shedding, the number of tuples that have to be filtered out of the system is reduced if the query is optimized for the throughput. It therefore ensures a more accurate query result. The problem that we address is how we can find and evaluate the configuration that maximizes the sustained throughput in the context of a distributed collaborative system. By contrast with other optimization goals such as work and latency, throughput is affected by the logical ordering of a query's operators, as well as by the physical placement of the operators.

The measure of *throughput* quantifies the number of tuples that can be processed by the system in a unit of time. Referring back to our business performance monitoring example, event streams that represent notifications of state changes of manual tasks (for example, filling out forms) typically have much lower rates than notifications sent from automated tasks (for example, automated transactions). The knowledge about the relative magnitude of the input rates of streams should therefore be taken into account when deciding the allocation of common computing resources. In the WhiteWater project, we capture this knowledge into an *input profile* (or, simply, a *profile*). Since our goal is to maximize the input rate that can be processed without bottlenecks, we express throughput as a vector that quantifies the processing of each input stream. The goal of maximizing throughput is oftentimes relevant *only* if it satisfies the requirements of the input profile. Out of the large space of possible query configurations, our goal is to find the query plans that maximize throughput and adhere to the input profile.

In this work, we make the following contributions:

- We define the notion of *profiled throughput* optimization. We show how we can define a profile that captures the requirements of input streams.
- We propose an algorithm for finding a query configuration that maximizes the profiled throughput. In our solution, we address two challenges: searching through the space of all possible query configurations and efficiently evaluating a configuration. We show that the throughput maximization problem is NP-hard, and we propose to use hill-climbing techniques to traverse the query configuration space. Our model takes into account resource and quality constraints such as CPU, memory, communication bandwidth, and latency.

The novelty of our approach lies in defining the problem of profiled throughput optimization, adapting the known hill-climbing heuristics, and studying their behavior as applied to this specific objective function and constraints.

In the current model, we did not include stream splitting/merging as query operators although they can be very useful for scalability in a cluster environment. Split streams can be processed at different rates on various paths and, in this case, the merging operator would need to buffer in order to synchronize them. Since in this paper, we do not model unbounded queues, we cannot quantify the effects of merging. This raises new challenges that we are currently investigating.

We structured our presentation as follows: In Section 2, we introduce the problem of maximizing the profiled throughput and discuss how it differs from previous work. In order to evaluate a query configuration, in Section 3, we show how we can maximize throughput under a set of constraints that are reviewed in Section 4. Our solution includes an evaluation procedure and a method for traversing the search space. Since finding an optimal configuration is NP-hard, we propose to use hill-climbing techniques, overviewed in Section 5.1. In Section 5.2, we present in detail the search method. The experimental analysis of our techniques is presented in Section 6. Section 7 presents the literature of solutions in different areas related to our work. We conclude with some directions of future work that we find interesting and useful in building a more general system for distributed continuous queries.

2 Problem Definition

In order to avoid the loss of query result quality, our goal is to create a query configuration that can process the highest input rate possible. This naturally translates into finding an order of query operators and a placement of the operators on physical nodes that maximize throughput. However, we

TABLE 1
Notation Used

Notation	Explanation
$o.r_{in}$	Set of input rates into operator o, in terms of tuples per unit of time
$o.r_{out}$	Output rate for operator o, in terms of tuples per unit of time
$o.s$	Predicate selectivity for operator o
$o.w$	The window timespan of operator o
$o.c$	Cost as number of instructions necessary for operator o to evaluate one tuple
$o.c_r$	Cost rate of operator o, as a function of processing cost per tuple and rate of input tuples
$o.col$	Columns associated with operator o
$N.I$	Processing power of physical node N in terms of instructions per unit of time
$N.M$	Memory resource of physical node N
$N.C$	Expression of constraint for node N

show that we cannot use this measure of throughput as the only indicator of how good a query configuration is. In this section, we compare throughput with other optimization goals and redefine throughput optimization to include additional knowledge that we found to be essential. The notation that we will use throughout the paper is summarized in Table 1.

2.1 On Throughput, Work, and Latency

Throughput represents the *number of input tuples that can be processed by the system in a given unit of time.* By contrast, maximum latency measures the maximum time that it takes for all operators on a path to process an input tuple. This notion of latency is often used in practice because it is easily measurable: It is the observed delay between the time that a tuple enters the system and the time that the corresponding result tuple is produced. Note that other definitions of latency are possible (for example, average latency). Work can be defined as the number of instructions needed to process a given input rate per time unit. Work, latency, and throughput are all optimization criteria that can be used to ascertain the quality of a query configuration.

For our purpose, it is critical to maximize throughput, which as we will show, does not coincide with any of the other optimizations. We will illustrate our point with a simple example, as shown in Fig. 1. Consider two SELECT operators o_1 and o_2 with selectivities $o_1.s$ and $o_2.s$, respectively, and costs $o_1.c$ and $o_2.c$ in a number of instructions necessary to process a tuple. We refer to the placement where o_1 is on node N_1 and o_2 is on node N_2 as configuration C_1. Configuration C_2 swaps the physical placement of the operators in C_1. For simplicity, in this example, there are no network, memory, or latency constraints and no queues.

The *maximum latency* of an operator is calculated as the ratio of the number of instructions necessary to process a tuple to the speed of these instructions. In the first configuration C_1, total latency is the sum of the latencies of both operators $o_1.c/N_1.I + o_2.c/N_2.I$. Calculations of latencies of C_2 are similar, and the results are summarized in Table 2. It is important to note that the actual order of operators on a path does not play a role in the total latency.

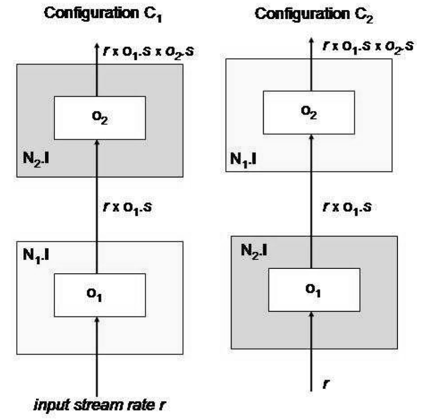

Fig. 1. A query and two of its possible configurations.

By contrast, the total *work* performed by the system only takes into account the logical ordering of operators, whereas the physical placement of the operators onto nodes does not matter. For the first configuration C_1, the work performed by the first operator is measured in the number of instructions/time unit as $r \times o_1.c$. The total work is the work of the two operators $r \times o_1.c + r \times o_2.s \times o_2.c$.

Unlike maximum latency and work, *throughput* is affected by both the physical and the logical placement of the operators.[1] Moreover, instead of considering the input rate r set, throughput is used to calculate the biggest r that the system can cope with. The limit on r is due to at least one of the operators becoming a bottleneck. For the query plan in C_1, operator o_1 can only support an incoming tuple rate bounded by the processing speed $N_1.I$ of the node N_1, that is, $r \times o_1.c \leq N_1.I$. Considering only operator o_1, the input bottleneck occurs when $r = N_1.I/o_1.c$. The second operator is bounded according to $r \times o_1.s \times o_2.c \leq N_1.I$, where $r \times o_1.s$ is the operator's input rate when the input to the query is r. The input limitation of o_2 leads then to a rate of $N_2.I/(o_1.s \times o_2.c)$. Then, the query is only able to cope with the minimum between the possible rates of the two operators:

$$throughput = min\left(\frac{N_1.I}{o_1.c}, \frac{N_2.I}{o_1.s \times o_2.c}\right).$$

Table 2 shows the expression for the throughput of configuration C_2. Previous work in distributed query optimization focused on reducing latency or work, whereas the measurement of throughput was used as an indication of how good a query configuration was. By contrast, our goal is to find a way of optimizing specifically for throughput, whereas latency, for example, can be treated as a constraint.

2.2 Throughput as a Vector

Previous stream work addressed the issue of maximizing or improving the throughput, often measured at the root of the query tree and represented by one number. We refer to the throughput measured by considering the output rate of a query as *output throughput*. This is different from *input*

1. Note that this does not hold in general. For example, in the case of a multijoin query, the latency is affected by the total height of the query plan (for example, left deep versus bushy).

TABLE 2
Optimization Goals

Optimization type	Config. C_1	Config C_2	affected by...
max. latency	$\frac{o_1.c}{N_1.I} + \frac{o_2.c}{N_2.I}$	$\frac{o_1.c}{N_2.I} + \frac{o_2.c}{N_1.I}$	physical plan
work	$r \times (o_1.c + o_1.s \times o_2.c)$	$r \times (o_1.c + o_1.s \times o_2.c)$	logical plan
throughput	$min[\frac{N_1.I}{o_1.c}, \frac{N_2.I}{o_1.s \times o_2.c}]$	$min[\frac{N_2.I}{o_1.c}, \frac{N_1.I}{o_1.s \times o_2.c}]$	physical and logical plan

throughput, which is the rate at which input tuples are processed by the system. As the input throughput increases, the output throughput usually increases as well. If we look in more detail, then one can see that the output throughput depends not only on the input throughput but also on the selectivity of the operators. If selectivities vary, then the ratio of the input to the output throughputs fluctuates as well.

Consider now only the input throughput. In measuring the input throughput as a number, we lose information on how the different input streams contribute to the process. This information is critical if we want to optimize for a system where there are differences in the behavior of the streams. We therefore adopt a vector notation of the throughput as $< r_1, r_2, \cdots, r_i, \cdots r_n >$, where r_i is the number of tuples from input stream i processed in units of time. For some applications, it makes sense to maximize the output throughput because it gives an idea of how much "work" a system is able to do in terms of the tuples processed, and two configurations are easily comparable. This is not the case in our model, since we want to maximize the individual input rates that can be processed by the system.

Using the vector notation for throughput, the comparison between the input throughputs of two query configurations is not straightforward. For example, let two configurations support the input streams r_1, r_2, and r_3 at the maximum rates of $< 10t/s, 40t/s, 20t/s >$ and $< 40t/s, 10t/s, 20t/s >$, respectively. Which throughput should be considered "best?" The sum of all the tuples processed is 70, which is the same for both configurations. We argue that the best configuration is the one that maximizes the throughput AND fits more tightly with the behavior of the input streams. If the observed input rates at one time are $< 20, 5, 10 >$, then the first configuration clearly cannot support them, whereas the second can. The key is to take into account the knowledge about the behavior of input streams (or *profile*), and apply the throughput maximization problem to this profile. Next, we show how we can define a profile and how we can maximize a profiled throughput.

3 Maximizing a Profiled Throughput

A query may receive input from multiple data streams with different rate fluctuations. One stream may come from a source that rarely emits events, whereas another stream may be characterized by long bursts of data at very high rates. If the query optimizer is given even coarse information on the expected input behavior, then it can generate a query plan that is appropriate under these assumptions. Note that without this additional knowledge, the query optimizer will have no way of distinguishing among many feasible solutions and may decide that the best solution is one that accepts a high input rate on the slower stream and a low input rate on the fast stream. In this paper, the profile used during optimization defines the values of the maximum rates that the streams are expected to reach. The profile of the input is then defined as an assignment of values to the input rates that becomes a target for supported throughput $< r_1^p, r_2^p, \cdots r_n^p >$.

A solution $C.S$ is an assignment of values to the input stream rate variables of a given configuration C such that all the constraints are satisfied. The quality $Q^p(C.S)$ of a solution $C.S$ should then quantify how much the solution achieves toward the goal of maximizing the throughput with respect to the profile. Note that the goal can also be surpassed. For a stream r_i where the maximum rate is expected to reach r_i^p, a solution with value r_i^s achieves r_i^s/r_i^p of the goal. The ratio can be greater than 1 if the solution exceeds the goal. We define the "goodness" of a solution as follows:

Definition 1. *The quality $Q^p(C.S)$ of a solution $C.S$ with respect to an input profile vector p is defined as* $Q^p(C.S) = \min_{1 \leq i \leq n} \left(\frac{r_i^s}{r_i^p} \right)$.

The intuition behind the above quality measure is that it captures the highest sensitivity to overload from any input stream because if any of the streams cannot be processed, the system is overloaded, which can lead to backlogs or incorrect results due to shedding. Note also that in the presence of window joins, backlogs in any one stream will require buffering on other streams (in order to guarantee correctness of join results).

Note that a configuration has an infinite number of solutions. Consider one solution $C.S =< r_1^s, r_2^s, \cdots r_n^s >$. Then, all possible $C.S' =< r_1^{s'}, r_2^{s'}, \cdots r_n^{s'} >$ such that $r_i^{s'} \leq r_i^s$ are also solutions for this configuration. The configuration is as good as its best solution.

Definition 2. *The quality $Q^p(C)$ of a configuration C with respect to an input profile p is calculated as*

$$Q^p(C) = \max_{C.S}(Q^p(C.S)) = \max_{C.S}\left(\min_{1 \leq i \leq n} \left(\frac{r_i^s}{r_i^p} \right) \right) Q^p(C.S).$$

Under these definitions, the throughput optimization problem becomes the following nonlinear programming problem: The objective function to maximize is $Q^p(C)$ for all configurations C under the constraints imposed in the distributed system by the physical resources and service quality guarantees. We will discuss these constraints in detail in Section 4. For now, let any constraint be of the form $f(r_1, \cdots, r_n) \leq c$ with the following properties:

- $f()$ is a monotonically increasing function and
- c is a constant that measures the capacity of a resource or a quality of service requirement.

To find a solution, the query optimizer needs to traverse the search space of configurations and compare each visited configuration with the configuration that was the best so

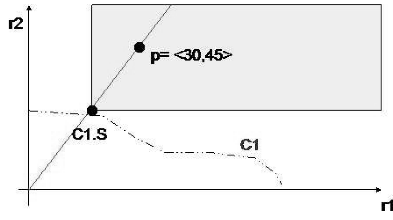

Fig. 2. Example of the best solution for a configuration C_1.

far. Finding an optimal configuration is a hard problem (see Section 5) and, in Section 5.4, we show that hill-climbing techniques can reach very good results efficiently. The challenge that we address next is the evaluation of a configuration, which is equivalent to finding the best solution of the configuration. Recall that each configuration can have an infinite number of solutions that satisfy the given constraints. We show next how we can make use of the properties of the feasible space to quickly identify the best solution for each configuration.

Proposition 1. *Let a query configuration C be restricted by constraints that are of the form $f(r_1, \cdots, r_n) \leq c$, where c is a constant, and $f()$ is monotonically increasing. For a profile $p = < r_1^p, r_2^p, \cdots r_n^p >$, a solution with the greatest $Q^p(C.S)$ lies on the surface bounding the region of feasible solutions and on the line through the origin and p.*

Proof. We will prove by contradiction that Proposition 1 is true. Let the solution that is found at the intersection of the bounding curve with the line between the origin and profile point p be $S = < r_1^s, r_2^s, \cdots r_n^s >$. Then, $r_1^s/r_1^p = r_2^s/r_2^p = \cdots = r_n^s/r_n^p$. Assume now that there is another feasible solution $S' = < r_1^{s'}, r_2^{s'}, \cdots r_n^{s'} >$, $S' \neq S$, such that $Q^p(C.S') > Q^p(C.S)$. In other words,

$$\min_{1 \leq i \leq n} r_i^{s'}/r_i^p > \min_{1 \leq i \leq n} r_i^s/r_i^p.$$

Because $r_1^s/r_1^p = r_2^s/r_2^p = \cdots = r_n^s/r_n^p$, it must be the case that all components of S' are greater than their corresponding components $S : r_i^{s'} > r_i^s,\ \forall r_i^{s'},\ 1 \leq i \leq n$. Without loss of generality, let us rewrite S' as $< r_1^s + \delta_1, r_2^s + \delta_2, \cdots r_n^s + \delta_n >$, with all $\delta_i > 0$. Since S lies on the bounding curve, it satisfies at the limit at least one constraint such that $f(r_1, r_2, \cdots r_n) = c$. For solution S', this constraint will be evaluated as $f(r_1 + \delta_1, r_2 + \delta_2, \cdots r_n + \delta_n) > c$. It follows that at least one constraint is not satisfied, and S' is not a feasible solution. The assumption that S' is a feasible solution is contradicted. □

To illustrate this point in two dimensions, consider the example in Fig. 2. For configuration C_1, the intersection of the constraint boundary with the line through the origin and $< 30, 45 >$ is at the solution $C_1.S$. Any solution with the same or better quality according to the $Q^p(C.S)$ measure increases r_1, r_2, or both. This solution lies in the darker region, where $C_1.S$ is the lower left corner. One can see that due to the shape of the feasible space imposed by the properties of the constraints, no point in the feasible space can also be in the hashed region.

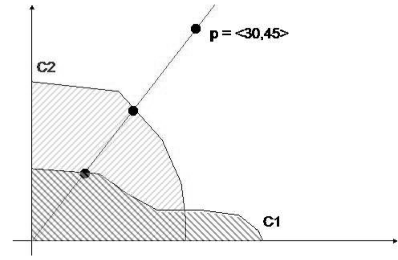

Fig. 3. Boundary of feasible region contains only dominant points.

Note that Proposition 1 allows us to compare two configurations (such as C_1 and C_2 in Fig. 3) by comparing the intersection points of the feasible space boundaries with the line from the origin to the profile point. In this case, the better configuration is C_2 because its intersection point is closer to the profile p.

4 Specifying Constraints

We now describe the details of the constraints and show that they have the property that we used to efficiently find the best solution of a configuration.

4.1 Limitation on Processing Resources

For a processing node N_j with resources $N_j.I$ available for query execution, the combined load of the operators on N_j is limited by $N_j.I$. Typically, the cost $o.c$ of an operator o is characterized by the number of instructions necessary to process one input tuple. Since we are calculating input *rates*, we can define the corresponding *cost rate* $o.c_r$ as a product between input rate and cost, in instructions per second. Note that the resource of a node $N.C$ is also measured in instructions per second. When operators $o_1, o_2, \cdots, o_n$ are placed on N_j, the constraints can be expressed as the sum of the cost rates of all operators:

$$\sum_{i=1}^{n} (o_i.c_r) \leq N_j.I \qquad (N_j.C).$$

For each physical node, there is one such inequality that expresses the constraint on the physical resources of that node. Thus, to obtain the constraint expressions, how do we calculate the cost rates? Let a connected directed graph of operators represent the flow of tuples/processing through operators in the node. Note that there can be multiple such directed graphs. Since the input rate of one operator is the output rate of another, the left-hand side of $N_j.C$ is a nonlinear expression in terms of the input rates into the leaf node of the graph and the cost per tuple of the different operators. Table 3 enumerates the rules for computing the cost rate of operators for SELECT, PROJECT, and JOIN. We

TABLE 3
Rules for Computing $o.c_r$

Operator	$o.r_{in}$	$o.r_{out}$	$o.c_r$
σ	r	$r \times o.s$	$o.c \times r$
π	r	r	$o.c \times r$
$\bowtie$	r_1, r_2	$2 \times o.w \times r_1 \times r_2 \times o.s$	$o.c \times (r_1 + r_2)$

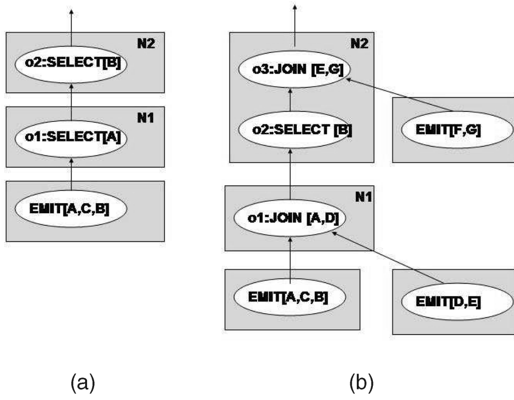

Fig. 4. Example of operator assignments to physical nodes.

assume double hash JOIN and a time-based JOIN window, where the output rate $o.r_{out}$ is therefore the rate on the first stream r_1 multiplied by the number of tuples in the window of the second stream $(o.w \times r_2)$ plus the rate of the second stream multiplied by the number of tuples of the first stream in the JOIN window. Note that the JOIN cost formula does not take into account the cost of purging the expired tuples from their respective windows. More fine-grained cost functions can be considered (for example, as discussed in [17]).

We consider constant input rates because the goal is to analyze how the system behaves at a maximum rate. This is different than modeling the fluctuating behavior of the system at runtime input rates, as in [27].

Example 1. Let a query of two operators be as illustrated in Fig. 4a. Operator o_1 is placed on a node N_1 of capacity $N_1.I$, and operator o_2 is on N_2 of capacity $N_2.I$. Then, the configuration is subject to the following constraints:

$$o_1.c \times r_1 \leq N_1.I \qquad (N_1.C)$$

$$o_2.c \times o_1.r_{out} \leq N_2.I \Leftrightarrow o_2.c \times (r_1 \times o_1.s) \leq N_2.I \qquad (N_2.C).$$

Example 2. For a more complex example, consider the query operators in Fig. 4b. The rate r_1 is the rate of data emitted by EMIT[D, E], r_2 is the rate of tuples emitted by EMIT[A, C, B], and tuples from EMIT[F, G] have a rate r_3. In this case, the constraints are

$$o_1.c \times (r_1 + r_2) \leq N_1.I \qquad (N_1.C)$$

$$\begin{aligned} N_2.I \geq\ & o_2.c \times o_1.r_{out} + o_3.c \times (o_2.r_{out} + r_3) = \\ & o_2.c \times 2 \times o_1.w \times o_1.s \times r_1 \times r_2 + \\ & + o_3.c \times (o_2.s \times 2 \times o_1.w \times o_1.s \times r_1 \times r_2 + r_3) \\ & \qquad (N_2.C). \end{aligned}$$

One can build the constraints by accumulating the terms in a bottom-up traversal of the query graph.

4.2 Memory Limitation

We assume that operators are able to process tuples fast enough that no additional buffers are necessary. In Table 4, we show that the space required by a SELECT and PROJECT is the size of a tuple m_t, whereas the memory requirement for a JOIN is that of storing tuples that fit in the window size $(o.w \times r_1 + o_w \times r_2)$ and two hash tables (of allocated size h).

TABLE 4
Rules for Computing $o.c_r$

Operator	space required o_m
σ	m_t
π	m_t
$\bowtie$	$o.w \times r_1 \times m_t + h + o.w \times r_2 \times m_t + h$

The memory constraints should reflect the fact that the total memory used by all operators in one node should be less than what the node allocates for the execution of the corresponding operators. That is, for each N_j

$$\sum_{i=1}^{n}(o_i.m) \leq N_j.M.$$

Note that Table 4 refers specifically to the SELECT, PROJECT, and JOIN operators. For more general stream operators, the memory requirements will consist of a fixed footprint (such as state for aggregators) and an input-rate-dependent component (such as for a window operator). In order to accommodate additional operators, one would need to provide appropriate formulas for estimating the memory requirements of each operator.

4.3 Bandwidth Requirements

Bottlenecks arise due to operators that process tuples slower than they are received and also due to communication link delays. The constraint on a link $L_{i,j}.C$ from node N_i to node N_j is that the bandwidth $L_{0,1}.B$ cannot be less than (the rate coming out of node N_i) $\times$ (the size m_t of tuples). Consider again the example in Fig. 4a. The bandwidth constraints are

$$L_{0,1}.B \geq r_1 \times m_t \qquad (L_{0,1}.C)$$

$$L_{1,2}.B \geq o_1.r_{out} \times m_t = o_1.s \times r_1 \times m_t \qquad (L_{1,2}.C).$$

4.4 Quality of Service Guarantees: Latency

The maximum latency of a query configuration is given by the total time taken by all operators on the most time-expensive path of the configuration. For an operator o on physical node N, the processing time for one tuple is calculated as $o.c/N.I$. Let $P_1, P_2, \cdots P_m$ in the set P be all the paths from the leaves to the root of a query configuration tree. Then, the requirement that the maximum latency should not exceed a limit L can be written as

$$\max_{P_i \in P}\left(\sum_{N_j \in P_i} \sum_{o_i \in P_i \cap N_j} \frac{o_i.c}{N_j.I}\right) \leq L.$$

Evaluating these constraints efficiently is not straightforward. Finding the values of variables $r_1, \cdots r_n$ that maximize the quality is done through evaluating the set of nonlinear constraints and the additional constraint due to the profile (discussed in Section 2). In our implementation, we use the relationship of the variables imposed by the profile to

rewrite the resource and latency constraints in terms of *only one variable*. Then, to solve the nonlinear equations, we used a binary search approach. Let us rewrite a constraint $N_j.C$ as $\sum_{i=1}^{k}(a_i x^i) \leq N_j.I$. The initial value for the $high$ limit is $\min_{j=1}^{m}[\frac{N_j.I}{a^k}]^{1/k}$, and $low = 0$. In the first iteration, we take the medium mid and plug it in all constraints. If all are satisfied, then the next iteration will continue after setting $low = med$. Otherwise, we set $high = mid$. The algorithm stops when a certain precision is achieved (in our experiments, we used 0.001 for precision).

5 Searching for the Best Configuration

In the previous sections, we introduced the concept of profiled throughput and proposed a way to efficiently evaluate a given configuration by quickly finding its best solution. A challenge that we have not addressed yet is how we can search through the space of possible configurations. Since the problem is NP-hard (as shown in Proposition 2.), we propose to use hill-climbing techniques that are appropriate for the configuration search space. It is essential that the hill-climbing method used emphasize both intensification and diversification.

Proposition 2. *Maximizing the profiled throughput of a distributed continuous query under processing constraints is an NP-hard problem.*

Proof. In order to verify the NP-hardness of this optimization problem, it suffices to prove that the corresponding decision problem is NP-hard. The decision problem can be formulated as follows: For a given continuous query and a given throughput vector, is there a configuration that can sustain the throughput? To prove the NP-hardness of this decision problem, it is enough to consider the special case of queries with only SELECT operators, which was shown to be NP-hard in [26]. Since this special case is NP-hard, the general decision problem is NP-hard as well. Since the original optimization problem is at least as hard as the decision problem, it follows that it is NP-hard as well. □

5.1 Background: Hill Climbing

Large search spaces are often traversed using a greedy local improvement procedure. The procedure starts with an initial configuration and refines it by selecting the best next configuration from the neighborhood of the current configuration until no neighbor is better than the current configuration. This is also called "hill climbing" because the objective function is improved with each iteration (assuming that the goal is maximization). The drawback of a local improvement method is that although it finds the top of a "hill," it may be only a local optimum, dependent on the position of the initial configuration. However, the local optimum found may be different from the global optimum. To increase the chances to find the global optimum, a search method can include steps that escape from the local optimum by jumping to a random position in the search space. Educated decisions on when and where to escape the local optima, as well as when and which inferior intermediate configurations can be accepted, can be based on the information gathered in previous iterations. The class of these methods is metaheuristics. According to [12], a metaheuristic is "a general kind of solution method that orchestrates the interaction between local improvement procedures and higher level strategies to create a process that is capable of escaping from the local optima and performing a robust search of a feasible region." Next, we outline the three metaheuristics that we experimented with: Tabu Search, Reactive Tabu Search, and Simulated Annealing. In the experimental section, we show the trade-offs between these different heuristics for the specific problem of maximizing the throughput in stream networks. In general, these methods differ in the focus that they place on climbing hills toward the local optimum versus exploring several hills to find the tallest ones. Note that in all the algorithms presented next, we assume that the goal is maximization of the objective function.

5.1.1 *Greedy Algorithm*

A basic greedy algorithm can start from an initial configuration C and then iterates to search for a better configuration until a stopping condition becomes true. At each iteration, the neighborhood of the current configuration is explored, and the best neighbor is chosen to become the current configuration. Note that since it only accepts local improvements, it will find the top of the local hill, and it will not explore other hills for a global optimum.

5.1.2 *Tabu Search*

The procedure of the Tabu Search [12] starts from an initial configuration C, and from the neighborhood of C, it only accepts improving configurations C' (that is, such that $f(C') > f(C)$). Through a set of iterations, it finds a local optimum. It then continues to explore the search space by selecting the best nonimproving configuration found in the neighborhood of the local optimum. To avoid cycles back to an already visited local optimum, the procedure uses a limited *Tabu list* of previous moves.

5.1.3 *Reactive Tabu Search*

Improvements to the basic Tabu Search were proposed for intensification and diversification [6]. Intensification is used to explore more the parts of the search space that seem more promising, whereas diversification enables the procedure to consider configurations in parts of the search space that were not explored previously. A method that employs both intensification and diversification is the Reactive Tabu Search. One enhancement is the fully automated way of adjusting the size of the Tabu list that holds the set of prohibited moves, based on the evolution of the search. Another feature that enables better diversification is the escape strategy. Following a threshold number of repetitions of Tabu configurations (notice now that we store configurations instead of moves), the escape movement is enforced. Intuitively, the number of random moves that comprise an escape depends on or is proportional to the moving average of detected cycles because longer cycles can be evidence of a larger basin, and it is likely that more escape steps are required. The Tabu list size increases with

TABLE 5
Additional Notation

Notation	Explanation
$A(C, m(), Q^p(C))$	Algorithm for exploring the search space : place tag for any of the heuristics discussed
$m(C, \alpha)$	Function that produces a new configuration by a 1-step move from configuration C, using parameter α
$Q^p(C)$	Function that evaluates and returns the goodness of a configuration C

every repetition of a Tabu configuration, and it decreases when a number of iterations greater than the moving average passed from the last change of the Tabu list size. To keep the size of the Tabu list within limits, it is reduced when it is so large that all movements become Tabu. Due to its learning-based intensification and diversification characteristics, we were not surprised to find that in our experiments, the Reactive Tabu Search method is the most consistent in finding the global optimum.

5.1.4 Simulated Annealing

Simulated Annealing [12] is a metaheuristic that is especially good at escaping the local minimum. Recall that Tabu Search climbs hills of local maximum very fast and climbs down slowly to search for other local optimum. By contrast, Simulated Annealing focuses first on finding the tall hills and then on climbing them. In the beginning, it has the flexibility of taking steps in random directions, and it increases in time the focus on climbing the hills by reducing the probability to accept a downward move (that leads to an inferior configuration). An interesting aspect of Simulated Annealing is the probabilistic concept of selecting the next trial configuration.

5.2 Solution Outline

For a given query and physical configuration of a system, our goal is to find the configuration with the largest input rates that match the profiled input behavior.

A naive exhaustive approach is to build all possible query configurations by creating all the possible logical query plans and mapping them onto the physical components in all the possible ways. To build the search space, one can start with a feasible solution and explore all possible one-step *moves* to reach the neighborhood of that configuration. Then, the process continues, starting from each of the neighbors of the initial solution, and so on until there are no new configurations. We employ the concept of *one-step move* to build the neighborhood of a configuration. In our notation, the function that implements a one-step move over a given configuration C and returns a neighboring configuration is $m(C, \alpha)$ (see Table 5). In Section 5.3, we explain the implementation of $m(C, \alpha)$ in more detail. Each configuration created by running $m(C, \alpha)$ is evaluated according to an objective (in our case, to maximize the profiled throughput measured by $Q^p(C)$) and is assigned a measure by using $Q^p(C)$. The general structure of a solution is outlined as follows:

Solution Outline

1. Define algorithm A(), moves m(), evaluation function $Q^p(C)$.
2. Initialize a configuration C.
3. Return $C' = \mathrm{A}(C, m(C, \alpha), Q^p(C))$.

Starting from an initial feasible solution C, an algorithm A is employed to generate new configurations, evaluate them, and find a final solution. In order to implement A, we make use of hill-climbing methods, which, as we will show in the experimental section, drastically reduce the number of configurations examined and, in many cases, find the optimum solution. Since hill climbing is a general class of algorithms, they need to be provided with an initial solution, methods for creating the search space, and methods for evaluating a configuration. Therefore, we need to specify how we pick an initial solution, a method for finding neighboring solutions, and an evaluation function.

In this paper, we assume that we are given a logical query plan of the continuous query. In the case of multiple queries, common subexpressions may have been merged in the initial query plan. If this is true, then we do not attempt to separate those subexpressions. Since each operator can be assigned to any node in a set of physical nodes, we randomly select a mapping. This initial query configuration needs to be correct but does not have to be a good solution.

5.3 Neighborhood of a Configuration

Starting from a configuration C, we apply a *one-step move* to build a configuration neighbor to C. The neighborhood of a configuration C is therefore defined as $N(C) = \{C' : C' = m(C, \alpha)\}$.

We did not discuss yet the role of parameter α in method $m(C, \alpha)$. Recall that there are two types of one-step moves that modify a configuration. A logical move is a swap of two operators under the constraints of the operator's semantics. A physical move is a mapping of a query operator to a different physical node. The balance between the two types of moves is quantified by a parameter α. Method $m(C, \alpha)$ selects a physical move with probability α as follows:

$$m(C, \alpha) = \begin{cases} m_{logical}(C) \ , & \text{if } p \geq \alpha, \\ m_{physical}(C), & \text{if } p < \alpha, \end{cases}$$

where p is a random variable uniformly distributed in [0, 1].

We experimented with different values of α, ranging between 0.25 and 0.75, and in our examples, the most consistent was $\alpha = 0.5$, which gives an equal probability for a physical move and a logical move. Note that if the method used requires the examination of the entire neighborhood, then we build the neighborhood exhaustively rather than probabilistically. Physical moves $m_{physical}()$ are straightforward to implement, given knowledge about the topology and resources of the processing components: The system randomly selects an operator and maps it to a choice of a physical node different than the current one. Due to limited resources or limited query capabilities, an operator may be hosted by only a subset of all the physical nodes.

A one-step logical move $m_{logical}()$ is implemented as the swap between an operator ($TopOp$) and its child ($BottomOp$). There are constraints that eliminate some of

TABLE 6
Rules for Swapping Operators

TopOp $o_t \rightarrow$ BottomOp $o_b \downarrow$	$SELECT$	$PROJECT$	$JOIN$
$SELECT$	always	$o_b.col \subseteq o_t.col$	never
$PROJECT$	$o_b.col \supseteq o_t.col$	$o_b.col \supseteq o_t.col$	never
$JOIN$	always	always	always

the logical moves from consideration. In some cases, a swap may never lead to a better solution, or depending on the operator columns, it may lead to an infeasible query plan. Table 6 summarizes the rules for swapping operators, which we do not present in further detail due to lack of space.

The list of logical moves presented here is not exhaustive. There are other logical moves and logical operators such as stream splitting/merging or operator cloning that can be used. Potentially, they can enable a better solution than the subset presented here. Our framework can accommodate any logical operator and logical move, provided that the formulas for deriving the output rates and operator cost are specified. Note that the operators discussed so far are commutative within the conditions specified in Table 5. In order to include more general operators, this table would need to be extended with commutativity rules for all the new operators.

5.4 Traversing the Configuration Space

We choose the heuristics presented in Section 5.1 as the base for the traversal of the configuration space. In general, they all go through a finite number of iterations, climbing toward a local optimum. They differ, however, in their speed of climbing the hills to the local maximum and in their diversification policies to look for a globally best solution. In each iteration, they create one or more configurations in the neighborhood of the current solution and select the next temporary solution. The creation of the candidate configurations is a result of implementing one-step moves with $m(C, \alpha)$, which are evaluated according to our objective of maximizing the most constrained input.

We experimented with two alternatives: employing the metaheuristic search in one and two phases. A one-phase procedure enables any one of the metaheuristics presented (Tabu Search, Reactive Tabu, or Simulated Annealing) using our definition of one-step moves and evaluation function. Note that in this case, each iteration creates new configurations based on either a logical or a physical move. By contrast, a two-phase procedure employs the heuristics twice. First, it searches for a solution by using only logical moves. Then, the solution found in the first phase is used as an initial configuration for the second phase, during which it searches for the best physical placement of this query plan.

PROCEDURE (One-Phase)
1. Initialize a configuration C.
2. $C' = \mathrm{A}(C, m(C, \alpha), Q^p(C))$.
3. Return C'.

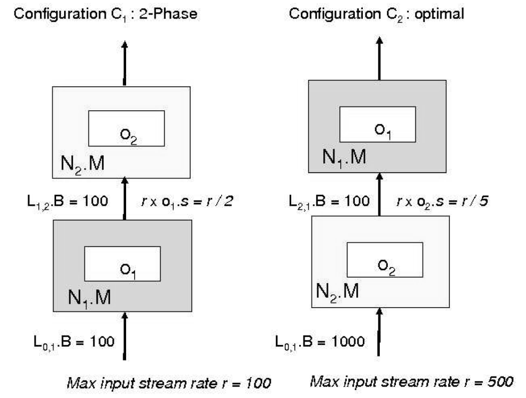

Fig. 5. Two-phase solution example.

PROCEDURE (Two-Phase)
1. Initialize a configuration C.
2. $C' = \mathrm{A}(C, m(C, 0), Q^p(C))$.
3. $C'' = \mathrm{A}(C', m(C, 1), Q^p(C))$.
4. Return C'.

Note again that in our implementation of the one-phase approach, we use $\alpha = 0.5$, but it is a parameter that can be set differently if needed. The advantage of using the two-phase approach is that during the same clock time, it can analyze/evaluate more configurations than the one-phase approach. The disadvantage is that once it finds a logical query plan in the first phase, it cannot change this in the second phase. This may prevent the two-phase algorithm from exploring parts of the search space, which may include the optimal configuration.

To illustrate this problem, consider a query with two operators o_1 and o_2 with selectivities $o_1.s = 1/2$ and $o_2.s = 1/5$. These operators have to be placed onto a physical network of two machines N_1 and N_2. Let us further assume that the memory requirements of the operators and the memory capacity of the nodes are such that they force the placement of o_1 on N_1 and of o_2 on N_2. The exhaustive two-phase approach (first optimize the logical plan and then the physical placement), as shown in Fig. 5, leads to configuration C_1 because on a single machine, the optimal logical plan is the one where most filtering is performed the earliest. By contrast, the exhaustive one-phase integrated approach will find the configuration $C2$, which allows for a higher maximum input rate than C_1 (500 tuples/sec compared to 100 tuples/sec).

6 Performance Evaluation

In this section, we present both a qualitative and a quantitative study of the performance of hill-climbing techniques in the context of maximizing a profiled throughput. We start by examining the search space characteristics and the runtime behavior of each method. Next, we focus on the effectiveness and efficiency of each method. There are two sets of experiments. First, we compare the hill-climbing methods with the optimum solution. In order to be able to compute the optimum solution (by exhaustive search), we are limited to queries for which the search space is relatively small. The results

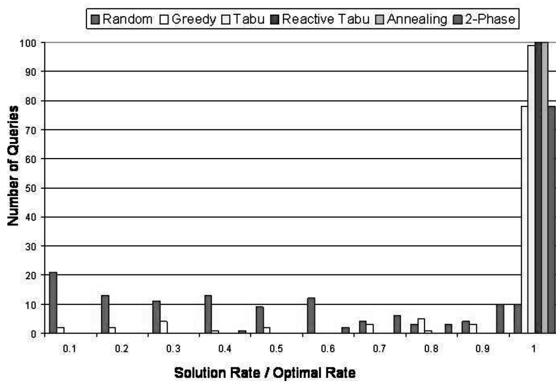

Fig. 6. Distribution of found solutions.

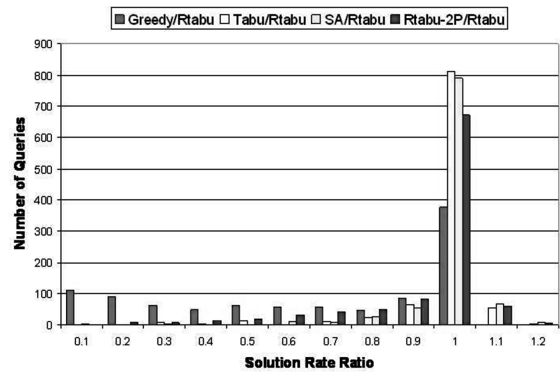

Fig. 7. Distribution of rate ratios.

show that the hill-climbing algorithms perform optimally most of the time. Second, we compare the different methods for efficiency and effectiveness. These are more extensive experiments, with a large number of randomly generated queries with greater search spaces. We show that Simulated Annealing is, by far, the fastest and is consistently one of the most accurate methods.

We implemented all the hill-climbing techniques described in Section 5.1 (greedy, Tabu, Reactive Tabu, and Simulated Annealing) in Java. We used the following parameters:

- Operator costs: EMIT.c = 0, SELECT.c = 20 instr., PROJECT.c = 50 instr., and JOIN.c = 800 instr.,
- Settings: join window size = 5 sec and network latency = 0.06 sec/link,
- Greedy: iterations = 1,000.
- Tabu: iterations = 1,000, and Tabu list size = 1,000,
- Reactive Tabu: iterations = 1,000, initial Tabu list size = 1, maximum cycle size = 50, Tabu list increase factor = 1.1, and Tabu list decrease factor = 0.9, and
- Simulated Annealing: num temperatures = 4, and iterations within a temperature = 400.

We also implemented a two-phase search strategy consisting of a "logical-moves-only" phase followed by a "physical-moves-only" phase on the logical configuration found by the first phase. This two-phase strategy can be used in conjunction with any search algorithm. In our experiments, we used a two-phase exhaustive search and a two-phase reactive Tabu search.

6.1 Effectiveness with Respect to Optimum Solution

In this experiment, we wanted to evaluate the absolute effectiveness of each search technique, that is, how often each technique finds an optimum solution. More generally, we wanted to quantify the distribution of solution quality for each technique. In order to do this, we needed to know the optimum throughput; therefore, we focused on queries that can be exhaustively optimized in a short time. This constrained us to queries with one or two emitters and at most five other logical operators with a selection of two physical nodes each. In order to get a representative measurement, we generated 100 random queries with these characteristics, and we ran each algorithm on all of them. The histogram shown in Fig. 6 summarizes the results.

For this particular batch of queries, we observed the following. The Reactive Tabu algorithm found an optimal solution 100 percent of the time, followed by Simulated Annealing at 92 percent, Tabu at 88 percent, Greedy and two-phase exhaustive, both at 66 percent of the time. The rest of the time, Simulated Annealing found solutions that were around 1.5 times worse (with only a couple of exceptions), the Tabu algorithm found solutions at most two times worse, whereas the Greedy algorithm found 26 solutions up to four times worse, and even two solutions, which were 100 times worse. Interestingly enough, almost all the other solutions found by the two-phase exhaustive algorithm were at most two times worse than the optimum (only one solution was three times worse). We included for reference a random placement, which produces the most suboptimal configurations and finds an optimal solution only 4 percent of the time. According to these results, the most effective algorithms for this class of queries are Reactive Tabu and Simulated Annealing. This is not surprising because both employ aggressive intensification and diversification.

6.2 Relative Effectiveness

In order to study the relative performance of the search algorithms, we generated 1,000 random queries, and we ran all algorithms on each query. In order to get a better idea of the effectiveness of the search algorithms for realistic queries, this time, we generated larger queries with more physical node options. The queries were generated as follows: First, we randomly selected an upper bound on the number of operators $maxOps = 40$. Then, we generated between 3 and $maxOps/3$ emitters. Subsequently, we built a query plan bottom up by placing random operators on top of the current plan up to $maxOps$. As a result, the exhaustive search became prohibitive, and we were able to only measure the relative effectiveness the algorithms. Since Reactive Tabu performed best in the previous experiments on smaller queries, we chose that as a reference, and we plotted for each query the ratio of the throughput found by other techniques over the throughput found by Reactive Tabu (Fig. 7). We found that Reactive Tabu is the most consistent. Rarely, Annealing is better (6.2 percent), sometimes is worse (12.8 percent), but most of the time performs the same (79 percent). Tabu is better only

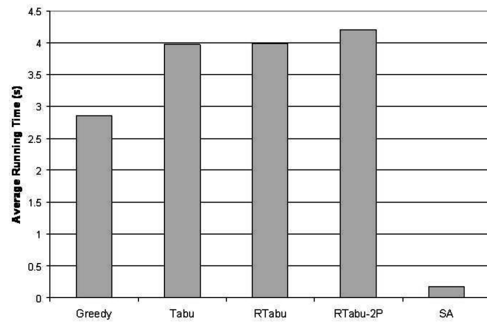

Fig. 8. Average runtimes.

6 percent of the time, worse 12.9 percent of the time, and most of the time the same (81.1 percent). Also, the Greedy algorithm was worse than Reactive Tabu 62 percent of the time and in many times as much as 10 times worse. Finally, the two-phase Reactive Tabu algorithm was worse in 25.4 percent of the cases, rarely better (7.5 percent), and mostly the same (67.1 percent) as the Reactive Tabu.

6.3 Efficiency

In order to study the relative efficiency of the search algorithms, we measured the average runtime for Reactive Tabu, Tabu, and Simulated Annealing on the 1,000 queries from the previous experiment. The results are plotted in Fig. 8. From this graph, we can observe that the Simulated Annealing algorithm is about 23 times faster than the Tabu variants, which have similar times. In fact, it averaged close to 200 ms. The Greedy algorithm is, on the average, about 30 percent faster than the Tabu algorithm, and the two-phase Reactive Tabu algorithm is about 5 percent slower than the Reactive Tabu. The runtime of all these algorithms is clearly dependent on the number of iterations performed, so for a fair comparison, we calibrated the number of iterations in such a way as to produce configurations of similar quality.

In conclusion, for the search space characteristics of our problem, the Simulated Annealing algorithm offers the best trade-off between effectiveness and efficiency.

6.4 Runtime Behavior

Our framework uses the estimated maximal throughput in order to compare the quality of candidate query configurations. In order to validate that the configuration deemed better according to our theoretical analysis indeed supports a higher throughput in practice, we ran the following experiment.

Consider the query

$$Q = \sigma_{D=0}\sigma_{A=0}(Emit_{AB} \bowtie_B (Emit_{BC} \bowtie_C Emit_{CD})).$$

We initialized the Simulated Annealing algorithm with a query configuration C_1 consisting of the same logical plan as shown in Q placed on a single node. After optimization, the resulting configuration C_2 has the logical plan $(\sigma_{A=0}Emit_{AB}) \bowtie_B (Emit_{BC} \bowtie_C (\sigma_{D=0}Emit_{CD}))$ distributed on two nodes (with the internode link being the link between the two joins). We generated synthetic data on the *Emit* operators such that the join on C is 100 times more selective than the JOIN on B, and each of the two SELECT operators have selectivity one in 100. Also, for this experiment, we used a throughput profile in which all the three input rates are equal.

The configuration C_1 was able to process a maximum input throughput of 8,517 tuples/sec/stream, whereas the optimized configuration C_2 processed 37,313 tuples/sec/stream. Therefore, the configuration C_2 performed indeed better in practice, as predicted by the model.

7 Related Work

Examples of data stream management systems are STREAM [29], Aurora [3], TelegraphCQ [8], and Gigascope [9]. The first two are designed to operate in a single physical node and focus on minimizing the end-to-end data processing latency. Subsequent work extends Aurora to distributed systems [7], [1]. The extended framework Borealis includes a load distribution algorithm that aims at minimizing the average end-to-end latencies [30]. TelegraphCQ is based on the Eddy [4] and SteM [22] mechanisms, enabling shared and adaptive processing of multiple queries. The system can also operate in a cluster of machines by using the Flux operator [24]. The main goal of Flux is to equalize the utilization of all the participating physical nodes. Gigascope employs a two-layer architecture tailored for processing high-speed Internet Protocol (IP) traffic data. A new architectural paradigm is presented by Franklin et al. [10] for systems with high fan-in topology. The main goals of the design are to reduce the overall bandwidth consumption and share as much of the processing as possible.

An optimization framework specifically targeted to streaming data management systems is proposed by Viglas and Naughton [27]. The proposed optimization technique is rate based, considering the rates at which the input data streams are coming into the system. In the above case, the focus is on the output rate, whereas in our case, we want to maximize the rates of the input streams that the system can support.

Pietzuch et al. [21] describe a method for query optimization, specifically targeted toward distributed stream-based applications. The proposed method focuses on the problem of query operator placement in the network, aiming at minimizing the end-to-end latency. However, they do not consider operator reordering, which may significantly affect the quality of the solution. The problem of query execution in widely distributed environments is studied by Ahmad and Çetintemel [2]. The study describes algorithms that try to minimize the bandwidth use or meet a certain quality of service requirements. In this case, the query plan is considered to be fixed.

A recent study [26] describes algorithms for the efficient in-network execution of queries on streaming data. The focus of the study is on deciding how a set of filter operators can be placed on a single path of physical nodes. The main algorithm is then extended to handle multiple paths and a special JOIN operator, but there is no straightforward extension that can handle more general query expressions. In addition, the proposed techniques do not take into account the resource constraints of the

physical nodes, thus allowing every node to host any number of operators.

Jin and Strom [16] describe a method for modeling the performance query operators in distributed stream processing. Their model takes the operator placement graph (that maps query operators onto physical nodes) as input, as well reasons about the time latencies to process each event in the system and the throughput that the system can support.

Query optimization has been studied extensively in the traditional database setting [15], as well as in distributed and parallel systems [25], [28], [23], [11]. Nevertheless, in these cases, the focus is on short-lived queries that do not normally depend on each other, which is not the case for the problem that we are studying. The use of randomized algorithms for performing query optimization has been proposed in the past [14], [13], [19]. These studies show that this approach can be effective in finding good-quality solutions for relational query optimization for both centralized and distributed relational database management systems (RDBMSs and nonstream environment).

The problem of query optimization and operator placement in a distributed environment has also been studied in the context of sensor networks [20], [18], [5], where several different optimization metrics have been proposed. In this case, though, the algorithms are restricted by the limited resources of the sensor nodes.

8 Conclusion

In this paper, we explore continuous query optimization, which maximizes the system's runtime capacity with respect to input rates that have an observed profile. An input profile represents the knowledge on input behavior, which is useful in targeting solutions appropriate for the specific runtime requirements of the system. We discuss the profile as the target point of projected maximum values that input rates can take. The notion of a profile can be generalized further to include a set of target points representing characteristics of input streams. Fitting the solution to the line segments described this way by the profile corresponds to finding a solution that can support an evolving set of requirements.

According to our experimental evaluation, the method that we propose for maximizing the profiled throughput is both effective and efficient. The optimization is based on various hill-climbing techniques that perform similarly in our experiments, obtaining the optimal or near-optimal solution in more than 90 percent of the time. Simulated Annealing is orders-of-magnitude faster than the other approaches and, therefore, it may be preferable, especially for large query plans.

There are a few directions for future work that we are planning to look into or currently pursuing. They include optimizing throughput according to various types of profiles, query reconfiguration, and improving parameters through learning. In this work, we only considered one-time configurations and have not yet addressed the dynamic aspects of query reconfiguration. Reconfiguration may be necessary if the user of the monitoring system decides to monitor additional types of events for root-cause analysis. A basic solution for implementing the query modification is to remove the first query and add the new changed query. The disadvantage is that there is an overhead associated with the removal and activation of operators. Instead, operators should be more resilient and change only if they are not necessary for the new query or if the penalty paid for not being part of an optimal configuration is not too great. Another challenge that we did not address in this paper is the maintenance of operator and data statistics. An enhancement that we plan to add to our system is learning about the data and query behavior by using a feedback loop to make use of the more reliable statistics.

References

[1] D.J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A.S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The Design of the Borealis Stream Processing Engine," *Proc. Second Biennial Conf. Innovative Data Systems Research (CIDR '05),* Jan. 2005.

[2] Y. Ahmad and U. Çetintemel, "Networked Query Processing for Distributed Stream-Based Applications," *Proc. 30th Int'l Conf. Very Large Data Bases (VLDB '04),* 2004.

[3] D.J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: A New Model and Architecture for Data Stream Management," *The VLDB J.,* vol. 12, no. 2, pp. 120-139, 2003.

[4] R. Avnur and J.M. Hellerstein, "Eddies: Continuously Adaptive Query Processing," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '00),* W. Chen, J.F. Naughton, and P.A. Bernstein, eds., 2000.

[5] D.J. Abadi, W. Lindner, S. Madden, and J. Schuler, "An Integration Framework for Sensor Networks and Data Stream Management Systems," *Proc. 30th Int'l Conf. Very Large Data Bases (VLDB '04),* 2004.

[6] R. Battiti and G. Tecchiolli, "The Reactive Tabu Search," *ORSA J. Computing.,* pp. 126-140, 1994.

[7] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik, "Scalable Distributed Stream Processing," *Proc. First Biennial Conf. Innovative Data Systems Research (CIDR '03),* 2003.

[8] S. Chandrasekaran, O. Cooper, A. Deshpande, M.J. Franklin, J.M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M.A. Shah, "Telegraphcq: Continuous Dataflow Processing for an Uncertain World," *Proc. First Biennial Conf. Innovative Data Systems Research (CIDR '03),* 2003.

[9] C.D. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk, "Gigascope: A Stream Database for Network Applications," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '03),* 2003.

[10] M.J. Franklin, S.R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong, "Design Considerations for High Fan-In Systems: The HIFI Approach," *Proc. Second Biennial Conf. Innovative Data Systems Research (CIDR '05),* 2005.

[11] M.N. Garofalakis and Y.E. Ioannidis, "Multi-Dimensional Resource Scheduling for Parallel Queries," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '96),* H.V. Jagadish and I.S. Mumick, eds., 1996.

[12] F.S. Hillier and G.J. Lieberman, *Introduction to Operations Research,* nineth ed. McGraw Hill, 2005.

[13] Y.E. Ioannidis and Y. Kang, "Randomized Algorithms for Optimizing Large Join Queries," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '90),* 1990.

[14] Y.E. Ioannidis and E. Wong, "Query Optimization by Simulated Annealing," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '87),* pp. 9-22, 1987.

[15] M. Jarke and J. Koch, "Query Optimization in Database Systems," *ACM Computing Surveys,* vol. 16, no. 2, pp. 111-152, 1984.

[16] Y. Jin and R. Strom, "Relational Subscription Middleware for Internet-Scale Publish-Subscribe," *Proc. Second Int'l Workshop Distributed Event-Based Systems (DEBS '03),* 2003.

[17] J. Kang, J.F. Naughton, and S. Viglas, "Evaluating Window Joins over Unbounded Streams," *Proc. 19th IEEE Int'l Conf. Data Eng. (ICDE '03),* 2003.

[18] R. Kumar, M. Wolenetz, B. Agarwalla, J. Shin, P. Hutto, A. Paul, and U. Ramachandran, "Dfuse: A Framework for Distributed Data Fusion," *Proc. First Int'l Conf. Embedded Networked Sensor Systems (SenSys '03)*, 2003.

[19] R.S.G. Lanzelotte, P. Valduriez, and M. Zaït, "On the Effectiveness of Optimization Search Strategies for Parallel Execution Spaces," *Proc. 19th Int'l Conf. Very Large Data Bases (VLDB '93)*, R. Agrawal, S. Baker, and D.A. Bell, eds., pp. 493-504, 1993.

[20] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong, "Tag: A Tiny Aggregation Service for Ad Hoc Sensor Networks," *SIGOPS Operating Systems Rev.*, vol. 36, no. SI, pp. 131-146, 2002.

[21] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-Aware Operator Placement for Stream-Processing Systems," *Proc. 22nd IEEE Int'l Conf. Data Eng. (ICDE '06)*, 2006.

[22] V. Raman, A. Deshpande, and J.M. Hellerstein, "Using State Modules for Adaptive Query Processing," *Proc. 19th IEEE Int'l Conf. Data Eng. (ICDE '03)*, 2003.

[23] M. Stonebraker, P.M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu, "Mariposa: A Wide-Area Distributed Database System," *The VLDB J.*, vol. 5, no. 1, pp. 048-063, 1996.

[24] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin, "Flux: An Adaptive Partitioning Operator for Continuous Query Systems," *Proc. Int'l Conf. Data Eng.*, 2002.

[25] N.G. Shivaratri, P. Krueger, and M. Singhal, "Load Distributing for Locally Distributed Systems," *Computer*, vol. 25, no. 12, pp. 33-44, 1992.

[26] U. Srivastava, K. Munagala, and J. Widom, "Operator Placement for In-Network Stream Query Processing," *Proc. 24th ACM Symp. Principles of Database Systems (PODS '05)*, 2005.

[27] S. Viglas and J.F. Naughton, "Rate-Based Query Optimization for Streaming Information Sources," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '02)*, 2002.

[28] M.H. Willebeek-LeMair and A.P. Reeves, "Strategies for Dynamic Load Balancing on Highly Parallel Computers," *IEEE Trans. Parallel and Distributed Systems*, vol. 4, no. 9, pp. 979-993, Sept. 1993.

[29] J. Widom and R. Motwani, "Query Processing, Resource Management, and Approximation in a Data Stream Management System," *Proc. First Biennial Conf. Innovative Data Systems Research*, 2003.

[30] Y. Xing, S. Zdonik, and J.-H. Hwang, "Dynamic Load Distribution in the Borealis Stream Processor," *Proc. 21st IEEE Int'l Conf. Data Eng. (ICDE '05)*, 2005.

Ioana Stanoi received the BS degree in computer science, the BA degree in physics, and the PhD degree from the University of California at Santa Barbara. She is a research staff member at IBM Almaden Research Center. She has served as a program committee member of the International Conference on Very Large Data Bases (VLDB), the ACM SIGMOD International Conference on Management of Data (SIGMOD), the IEEE International Conference on Data Engineering (ICDE), the International Conference on Distributed Computing Systems (ICDCS), and numerous other conferences and workshops. She also served on the review panels of the US National Science Foundation (NSF) and of several journals including the *IEEE Transactions on Knowledge and Data Engineering*, *VLDB Journal*, the *ACM Transactions on Database Systems*, and *Information Systems*. Her current research focuses on scalable, correct, and efficient event processing techniques to support event-driven applications. In the past, she worked on a variety of research topics, including data warehouse maintenance, XML indexing, semantic matching, and mobile computing.

George A. Mihaila received the BS degree from the University of Bucharest and the MSc and PhD degrees from the University of Toronto, all in computer science. He is a research staff member at the IBM T.J. Watson Research Center. He also holds an adjunct faculty appointment at Columbia University. His research interests include data integration, data warehousing, data stream processing, and XML storage and processing. His research was published in high-quality journals and conferences, including the *Journal of Digital Libraries*, the ACM Symposium on Principles of Database Systems (PODS), the International Conference on Extending Database Technology (EDBT), the IEEE International Conference on Data Engineering (ICDE), and the W3C International World Wide Web Conference.

Themis Palpanas received the BS degree from the National Technical University of Athens, Greece, and the MSc and PhD degrees from the University of Toronto, Canada. He is a faculty member in the Department of Information and Communication Technology, University of Trento. Before joining the University of Trento, he worked at the IBM T.J. Watson Research Center. He has also worked for the University of California, Riverside, and visited Microsoft Research and the IBM Almaden Research Center. His research interests include data management, data analysis, streaming algorithms, and business process management. He has applied his research solutions to real-world industry problems and is the author of five US patents.

Christian A. Lang received the MS degree from the Munich University of Technology and the PhD degree from the University of California at Santa Barbara, both in computer science. He is a research staff member in the Database Research Group at the Business Informatics Department, IBM T.J. Watson Research Center. He has served on the program committee of the International Workshop on Database and Expert Systems Applications (DEXA) and as a reviewer of most major database conferences and journals. His research interests are in the area of storage and retrieval of structured and unstructured information. This includes query optimization, adaptive database systems, database tuning tools, data analytics, federated repositories, and real-time retrieval over streaming and/or high-dimensional data (for example, multimedia or business intelligence (BI) data). He is currently involved in several projects dealing with the scalability aspects of database management and business process monitoring systems. His findings were published in leading journals and database conferences. He holds and has applied for several US patents related to his work. He is a member of the ACM, the ACM SIGMOD, and the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.