Potential-Driven Load Distribution for Distributed Data Stream Processing

Weihan Wang*
University of Toronto
weihan@eecg.toronto.edu

Shimin Guo*
Google, Inc.
sguo@google.com

Mohamed A. Sharaf
University of Toronto
msharaf@eecg.toronto.edu

M. Tamer Özsu
University of Waterloo
tozsu@cs.uwaterloo.ca

ABSTRACT

A large class of applications require real-time processing of continuous stream data resulting in the development of data stream management systems (DSMS). Since many of these applications are distributed, distributed DSMSs are starting to receive attention. In this paper, we focus on an important issue in distributed DSMS operation, namely load distribution to minimize end-to-end latency. We identify the often conflicting requirements of load distribution, and propose a "potential-driven" load distribution approach to mimic the movements of objects in the physical world. Our approach also takes into account heterogeneous machines, different network conditions, and resource constraints. We present experimental results that investigate our algorithms from various aspects, and show that they outperform existing techniques in terms of end-to-end latency.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Distributed databases*

General Terms

Algorithms, Design, Performance

Keywords

Data streams, Distributed systems, Load balancing

1. INTRODUCTION

A large class of applications require real-time processing of continuous streaming data, such as network traffic analysis, sensor data processing, telecommunication accounting and monitoring, financial market analysis, etc. Several general-purpose data stream management systems (DSMSs) [6, 1, 4] have been proposed, with the goal of providing stream applications the ability to query over continuous data streams.

For the purposes of scalability and robustness in data stream processing, prototypes of distributed DSMS have also been proposed (e.g., Borealis [2]). An important issue in designing a distributed DSMS (D-DSMS) is the distribution of *query operators*, thus workload, across the physical machines *with relatively balanced load*. As the query plan and the workload may change over time, a D-DSMS needs to have the ability to dynamically adjust load distribution with the goal of improving the overall performance.

In this paper, we identify the usually conflicting requirements for good load distribution and propose techniques that strike a balance between these requirements which results in a reduced system latency. Our approach follows an analogy to physical systems where we consider each query operator as a physical object which is driven by different types of potentials towards a location where it has lower potential energy. Towards this, we define multiple types of *operator potential energies* corresponding to multiple requirements for load distribution. The definition also addresses different processing capacities and resource constraints. Then, we use the potential energy to guide load distribution.

The contributions of this paper are as follows:

1. We propose a novel framework for load balancing, based on an analogy with the physical world, that addresses different aspects of load balancing in heterogeneous environments.

2. Under this framework, we propose several algorithms to optimize initial load distribution as well as adjusting the distribution periodically.

3. We demonstrate by means of extensive experiments that a realistic implementation of the proposed approach improves the end-to-end latency compared to existing techniques.

The rest of the paper is organized as follows. Section 2 introduces the basics of the load distribution problem. We present the potential-driven approach framework in Section 3, and specific algorithms in Section 4. In Section 5, experimental results are presented. We summarize related work in Section 6 and conclude the paper in Section 7.

*This work has been partially conducted while the first two authors were at the University of Waterloo.

SSPS'08, March 29, 2008, Nantes, France.

2. PRELIMINARIES

2.1 System Model

We represent each query plan by an acyclic directed *query graph* where the nodes represent query operators and the directed edges represent data flow from one operator to another. For our work, it is not important whether the query graph is a traditional one generated by an optimizer or a boxes-and-arrows dataflow diagram as in Aurora [1] and Borealis [2]. For the purposes of exposition and easier comparison with previous work [11, 12], we use the second form in this paper. Figure 1(a) illustrates a sample query graph, where labeled squares represent operators. Two operators are called neighbors if there is an edge between them. In Figure 1(a), the operator labeled B is called the predecessor (or upstream neighbor) of operator D and D is the successor (or downstream neighbor) of B, as an edge is directed from B to D. An operator can have one or more predecessors, and can output tuples to multiple successors to share processing results (e.g. operator A in the figure). Associated with each edge is a finite-length input queue to buffer tuples that cannot be immediately processed (not shown in the figure).

There are two special types of nodes in a query graph, namely data sources and endpoints, and shown in Figure 1(a) as circles labeled DS and EP, respectively. A data source is where tuples are generated, whereas endpoints are where data flow ends and where query results are consumed by interested applications.

In a distributed system, operators are placed on a set of physical machines (hosts) connected by a network. Tuples are sent via the network if two neighboring operators are located on different machines. Operators can freely move from one host to another at runtime. However, such movement comes with cost as operator migration consumes both CPU cycles and network bandwidth.

In this paper, we assume heterogeneous hosts and network links, so that the scale of participating hosts can vary ranging from sensors to mainframes, and network links may have different bandwidth and latencies as well. Given that setting, our goal is to find operator placements which optmize query latency in such a heterogeneous environment.

Example: Figures 1(b) and 1(c) show two possible placements of the sample query graph. The rounded boxes labeled H_1 and H_2 represent two hosts on which the operators are placed. Cross-network data flow is represented by the edges that cut the dashed boxes. The only difference between the two placements is where operator D is located.

2.2 Load Distribution Requirements

In this paper, our objective is finding a good mapping from operators to hosts so that to minimize end-to-end latency of stream queries. End-to-end latency consists of operator processing latency, queueing delay, and network delay for cross-network streams.

In this paper, we identify three factors that affect the quality of load distribution. These factors are: (1) the average load at each host, (2) the load variance at each host, and (3) the utilization of network links.

2.2.1 Average Load

When a host is overloaded, i.e., the operators on this host cannot process tuples as fast as they arrive, tuples have to be queued and thus they will experience large queueing delays. As such, in the event of overloaded conditions, moving some operators from the overloaded host to some lightly-loaded hosts can significantly reduce the latency.

The average load of a host is measured periodically over fixed-length periods, called measurement periods. These should be sufficiently long to smooth out load fluctuation. The load of operator o is defined as the fraction of time needed by operator o to process the tuples that would arrive at operator o over the length of a measurement period. The load ld_h of host h is the sum of loads of all its operators (we summarize all notations used in the paper in Table 1).

To measure the load, let p_o be the *benchmark per-tuple processing cost* of operator o, which is the average time needed for operator o to process an input tuple if operator o is placed on a benchmark machine. (Note that the cost is independent of the machine it is actually being placed on.) During a measurement period, the *benchmark load* contributed by operator o is given by $\lambda_o p_o$, where λ_o is the observed data arrival rate of o during the measurement period.

In practice, some hosts may be overloaded, in which case

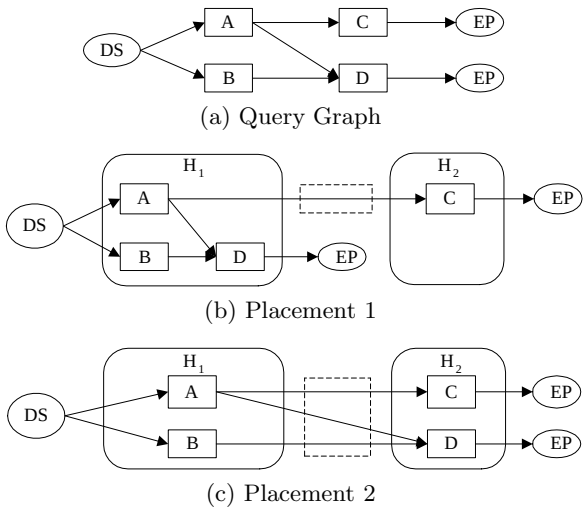

Figure 1: Sample Query Graph and Its Placement

h	a host
o	an operator
l	a network link
P	a placement
n	the number of hosts
m	the number of operators
ld_h	average load of h
α_h	capacity coefficient of h
q_h	load quota of h
p_o	benchmark per-tuple processing cost of o
λ_o	observed data arrival rate of o
λ'_o	ideal data arrival rate of o
ρ	load correlation
d_l	delay of l
u_l	bandwidth utilization of l
w_l	weight of load PE
w_c	weight of correlation PE
w_n	weight of network PE
θ	periodic optimization threshold

Table 1: Notations

operators that are fed by these overloaded hosts will see lower tuple arrival rates than they would otherwise. To overcome such inaccuracy, we can approximate the *ideal arrival rates* λ'_o of operator o using the input rates of data sources and operator *selectivities* in a cascading fashion, from upstream operators to downstream operators [11]. An operator's selectivity can be calculated by measuring the data flow rates of its input and output streams, and the selectivity is the rate of its output stream over the product of the rates of *all* input streams.

We account for different processing capacities of different hosts by introducing a *capacity coefficient* α_h for host h, that describes the relative processing power of host h compared to the benchmark machine. The *actual* per-tuple processing cost of operator o on host h is then $\alpha_h p_o$. Denoting the set of operators on host h as O_h, the load of host h is given by

$$\mathrm{ld}_h = \alpha_h \sum_{o \in O_h} \lambda'_o p_o.$$

Example: Consider host H_2 on Figure 1(c). Assume δ_X is the selectivity of operator X and λ_{DS} is the data source' input rate. The ideal arrival rate of the two operators on that host are computed as

$$\lambda'_C = \delta_A \cdot \lambda_{DS}$$
$$\lambda'_D = (\delta_A + \delta_B) \cdot \lambda_{DS}.$$

The average load of host H_2 is then

$$ld_{H_2} = \alpha_{H_2}(\lambda'_C p_C + \lambda'_D p_D).$$

We define the *load quota* q_h as the maximum allowable load on host h, which is a real number between 0 and 1. Sometimes the users may not want to fully utilize some hosts, either to conserve power for power-constrained devices or because the hosts are shared by other users or applications. For hosts that have no resource constraints and are entirely dedicated to the system, the load quota is simply 1.

2.2.2 Load Variance

Although the long-term average load on a host may not exceed its capacity, short-term load variance can result in temporary overload. During periods of overload, tuples accumulate in the input queues of the operators resulting in high queueing delay. Therefore, to minimize latency, it is necessary to reduce both the average load and the load variance on each host. Xing et al [11] tackle this problem by examining load correlation between operators. The loads of two operators are positively correlated if they tend to go up and down synchronously. They are negatively correlated if they follow opposite trends. In [11], it has been shown that by placing mutually positively correlated operators on separate machines and mutually negatively correlated operators on the same machine, not only will the load variance on each machine be minimized, but also when the load increases, the additional load will likely be distributed evenly across the machines. In this paper, we define load variance and load correlation as in [11]. Specifically, there are defined as follows:

The load of each operator/host measured in the most recent k periods are recorded. These k values form a *load time series*. Given a load time series $S = (s_1, s_2, \ldots, s_k)$, the average load and load variance is given by

$$E(S) = \frac{1}{k}\sum_{i=1}^{k} s_i,$$

$$var(S) = \frac{1}{k}\sum_{i=1}^{k} s_i^2 - \left(\frac{1}{k}\sum_{i=1}^{k} s_i\right)^2.$$

Given two load time series $S_i = (s_{i1}, s_{i2}, \ldots, s_{ik}), i \in \{1, 2\}$, their covariance and correlation coefficient are given by

$$cov(S_1, S_2) = \frac{1}{k}\sum_{j=1}^{k} s_{1j}s_{2j} - \left(\frac{1}{k}\sum_{j=1}^{k} s_{1j}\right)\left(\frac{1}{k}\sum_{j=1}^{k} s_{2j}\right),$$

$$\rho(S_1, S_2) = \frac{Cov(S_1, S_2)}{\sqrt{var(S_1)} \cdot \sqrt{var(S_2)}}.$$

We call the correlation coefficient of two load time series the *load correlation*. Note that load correlation is always between 1 and -1.

2.2.3 Average Link Utilization

The third factor is the utilization of network links. When one operator outputs tuples to another, if they reside on the same host, the transfer can be done in main memory with negligible delay. However, if they are on different hosts, there is a non-negligible network delay, which may become a significant part of the end-to-end latency. In addition, network links have finite bandwidth, which may become a system bottleneck and thus limit the overall throughput. Therefore, we would like to reduce network utilization by minimizing the number of cross-network streams.

Usually, the requirements on these three factors are conflicting. For example, let us suppose that in Figure 1, Placement 2 is better than Placement 1 in terms of average load as the former distributes the operators more evenly. However, if we consider network usage, Placement 1 is more preferable since it has two less cross-network edges than the other. Our algorithms deal with these conflicting requirements in a unified manner, as we explain in the next sections.

3. A POTENTIAL-DRIVEN APPROACH

3.1 Overview

We propose a load distribution framework that is formulated using an analogy to the physical world. In the physical world, a set of objects that exert forces on each other will spontaneously adjust their positions and evolve to an equilibrium where the total potential energy (PE) of the objects are at a local minimum. For example, consider a set of electrically charged particles, some of which are connected by springs, that interact with one another through gravity, electrical force, and the springs. If these particles are let to move almost freely, then each one will be driven by multiple types of potentials and move towards a position where it has lower PE. In such system, the change in the location of one object will change the PEs of all other objects it is interacting with. The end result is usually a well spread set of objects, without many agglomerating in one place.

So how is this related to the load distribution problem? We observe that load distribution exhibits similar dynamics

when we move operators around in the pursuit of reducing end-to-end latency. Specifically, the location (host) which is the most preferable for a particular operator depends on the locations of other operators. Meanwhile, moving an operator to a more preferable location might change the preferable locations of the other operators, which in turn may cause other operators to move.

Example: We would move one or more operators out of host H_1 in Figure 1(b) as the total number of operators (and thus the average load) on that host is relatively high. However, moving operator D from H_1 to H_2 may, in turn, cause A to move in the same direction as an effort to reduce network usage. A's movement may disturb load balance and cause further movement.

We define the PE of operators with respect to the locations of other operators in such a way that *an operator has lower PE if placed at a more preferable location in terms of end-to-end latency*. Then we let operators move in the same fashion as physical objects do: moving towards locations where they would have lower PE. The result is expected to be a distribution of operators to hosts in which the total PE of operators is small, which in turn translates into smaller latency.

We need to answer the following two questions to solve our load distribution problem: (1) how we define the PE function for operators, and (2) how we find the optimal mapping to minimize the total PE of all operators, based on that function. The next subsection addresses the first question, whereas the second question is addressed in Section 4.

3.2 Operator PEs

It is important that the definition of the PE of an operator reflects its preference over possible locations, given the locations of the other operators in the system. As discussed earlier, there exist multiple requirements for selecting preferable placements. Therefore, we define different types of operator PEs, just as there are different types of PEs in the physical world. Our approach is quite flexible in that it can accommodate a wide array of requirements for load distribution by defining a PE function for each.

Next, we define three types of PEs corresponding to the three requirements discussed in the previous section.

(1) Load PE: An operator has higher load PE if it resides on a more heavily loaded host. Therefore, an operator tends to be pushed out of a heavily-loaded host to a lightly-loaded one. The load PE can be defined in a number of ways. For example, it can increase linearly with the load of the host. Alternatively, it can increase slowly with the load when the load is low, but rapidly when the load approaches the quota of the host. The latter can be used when we are not too bothered by the load of a host as long as it does not exceed the quota. In our experiments, we adopt the following definition of load PE of operator o on host h, given the placement P of the other operators

$$PE_l(o,h,P) = \begin{cases} \frac{ld_h}{q_h} & \text{if } \frac{ld_h}{q_h} < 1; \\ 1 + p \cdot (\frac{ld_h}{q_h} - 1) & \text{otherwise.} \end{cases}$$

where p is the overload penalty coefficient, which is normally a value much greater than 1. Under this definition, the load PE of operator o on host h increases steadily as host h becomes more loaded, and increases dramatically faster once the quota is filled. This definition realistically captures the requirement of load balancing.

(2) Correlation PE: This type of PE is concerned with load variance. Xing et al [11] have shown that placing negatively correlated operators on the same host helps reduce the load variance of the host. Intuitively, an operator tends to move to hosts where its load time series fits in nicely with those of the rest of the operators on the same host. Therefore, we define the correlation PE of operator o on host h to be proportional to the correlation between the load time series of operator o and the sum of load time series of all other operators on h. In our experiments, we simply set the correlation PE equal to the correlation coefficient:

$$PE_c(o,h,P) = \rho(o,h),$$

where $\rho(o,h)$ denotes the correlation coefficient between the load time series of operator o and the sum of load time series of all operators on h *except* o.

(3) Network PE: This type of PE accounts for the overhead incurred by network transmission. For operator o, if all its downstream neighbors are on the same host as o, then its network PE is 0 (minimum). Otherwise, suppose L is the set of links that o uses to send tuples to other hosts (only downstream links are included to avoid double counting). Each $l \in L$ will contribute an additive term to the network PE of o

$$PE_n(o,h,P) = \sum_{l \in L} c(l).$$

For each $l \in L$, $c(l)$ is dependent on both link delay d_l and bandwidth utilization u_l of the link. u_l is defined as the ratio of the amount of data to be sent on one link in unit time over the bandwidth of the link. $c(l)$ has the general form of $c(l) = f(d_l) \cdot g(u_l)$, where both $f(\cdot)$ and $g(\cdot)$ are non-decreasing functions. Each link resembles a "rubber band" that connects operators, which pulls the connected operators to the same location. In our experiments, we found that the following definition of $c(l)$ worked quite well

$$c(l) = \begin{cases} d_l & \text{if } u_l \leq 1; \\ d_l \cdot u_l & \text{otherwise.} \end{cases}$$

The underlying idea is that the "tension" of a rubber band only depends on the delay of the link as long as the bandwidth is sufficient. Once the bandwidth utilization reaches 100% forming a bottleneck, the tension will increase with the traffic on that link.

Example: Network PE of operator A in Figure 1(b) is $c(l_{12})$, where l_{12} is the link between H_1 and H_2. Operators C and D in Figure 1(c) should have the same network PE as they share a common link, but the value should be different than $c(l_{12})$ due to different bandwidth utilization of the two placements. Other operators in both placements have zero network PEs.

Finally, the total PE of an operator is the weighted sum of its three individual PEs

$$\begin{aligned} PE(o,h,P) = \; & w_l \cdot PE_l(o,h,P) + w_c \cdot PE_c(o,h,P) \\ & + w_n \cdot PE_n(o,h,P). \end{aligned}$$

The summation must be weighted because the value of individual PEs are not normalized by any means. In the absence of a clear method to normalize these PEs, and in

order to get an unbiased result, we must scale these values so that no single type of PE unduly dominates the others. We will study the effect of different weight assignments through experiments and we will also provide insights towards automated weight assignments.

4. POTENTIAL-DRIVEN ALGORITHMS

Following the definition of PE, we can formally define the goal of load distribution algorithms as finding an optimal placement P^* for all operators that minimizes the total PE:

$$P^* = \arg\min_{P \in \mathcal{P}} \sum_{o \in O} PE(o, h_o^{(P)}, P),$$

where $\mathcal{P}$ is the family of all possible placements, O is the set of all operators in the query graph under consideration, and $h_o^{(P)}$ is the host where o shall reside according to P. As the query graph and the workload often change over time, a practical algorithm should be able to update P^* periodically at runtime to adapt to such changes.

Obviously, there may exist more than one way to find P^* or an approximation of P^*, and finding such an algorithm is independent of the above definition of PE. In this paper, we propose two approximate algorithms, but other approaches that achieve optimality are also possible. Our algorithms are divided into two classes: one that finds an initial mapping and one that periodically adjusts the mapping.

4.1 Initial Mapping

Before the initial mapping algorithm starts, we first randomly distribute the operators to hosts and run the system for a warm-up period to obtain load statistics necessary to compute the PEs for operators. After the warm-up period, we run the initial mapping algorithm which redistributes the operators without considering their original locations.

Given the large search space, the problem of finding the optimal mapping is difficult and computationally expensive. Therefore, we propose two heuristic algorithms that prune the search space by placing operators one-by-one. Specifically, we exploit two standard techniques for solving optimization problems, as we explain next.

4.1.1 Greedy Algorithm

The first algorithm, INIT_GREEDY, greedily finds a host for each operator in some order (ordering is discussed shortly). The algorithm ignores the original operator placement and starts from an empty system, in which no operator has been placed on any host. Then, for each operator o, it finds a host h such that $PE(o, h, P)$ is minimized, and adds the $\langle o, h \rangle$ pair into mapping P. Once a host is assigned to each operator, P is the mapping we get. Since the algorithm is straightforward, we do not show its pseudocode.

4.1.2 Dynamic Programming

A more sophisticated algorithm, INIT_DYN_PROG, uses dynamic programming (see Algorithm 1). Again, starting from an empty system, the algorithm considers operators one by one in some order. It uses an $(m+1) \times n$ matrix A to store intermediate results, where m is the number of operators and n is the number of hosts. Element $a[i, j]$ of A stores the least sum of PE achievable for the first i operators given that the ith operator is placed on the jth host, $0 \le i \le m$, $1 \le j \le n$, and $P_j^{(i)}$ is the mapping that achieves

Algorithm 1 INIT_DYN_PROG(O, H)

```
1: n ← size(H)
2: m ← size(O)
3: for j = 1 to n do
4:    P_j^(0) ← ∅
5:    a[0, j] ← 0
6: end for
7: for i = 1 to m do
8:    for j = 1 to n do
9:       r ← arg min_{1≤k≤n} PE(O[i], H[j], P_k^(i−1))
10:      a[i, j] ← a[i − 1, r] + PE(O[i], H[j], P_r^(i−1))
11:      P_j^(i) ← P_r^(i−1) ∪ ⟨O[i], H[j]⟩
12:   end for
13: end for
14: i ← arg min_{1≤k≤n} a[m, k]
15: return P_i^(m)
```

the least sum of PE. At the beginning, the zero-th row of A is initialized to 0, and the $P_j^{(0)}$'s are set to empty. The matrix is updated according to rules described in lines 9 – 11 of the algorithm. Finally, the smallest is chosen from the last row of A, and returned as the corresponding mapping.

Note that in both algorithms, the sum is computed over the PE of each operator without considering the effects of operators that are added to the mapping later. Normally, adding a new operator to the system will change the PE of many other operators. However, we ignore such effects to reduce the amount of computation at the cost of accuracy and optimality. Alternatively, one may apply some iterative method to gradually approach the "equilibrium" state, and we plan to investigate this in future work.

4.1.3 Operator Ordering

In both algorithms, we need sorted access to the operators. Generally, different orderings will produce different mappings, and it is not clear how to choose one (trying all possible orders is computationally not feasible). Our observation is that, operators with lower load have more flexibility in choosing hosts than those with higher load. For example, if the placement of an operator with high load is left to a later stage, it could happen that although the overall system has some spare capacity to accommodate that operator, there is no single host that has enough room to accommodate it. Based on this observation, for both of the above algorithms, the operators are considered in the descending order of their load.

4.2 Periodic Optimization

While the algorithms for initial mapping do not care where operators are originally placed, algorithms for periodic re-optimization must take that into account that frequent operator migration could incur large overhead (Section 2.1), which sometimes can be even larger than the performance gain from a better mapping. Therefore, for periodic optimization, we only move operators when the migration overhead is well justified by the gain of doing so.

Our solution is in large part based on the same algorithms for initial mapping, with a simple modification to control the number of the operators that can be moved by the algorithm. θ is a user-defined real number in the range of $[0, 1)$. At each time the periodic optimization is performed, we first fix the

last $\lfloor \theta \times m \rfloor$ operators on the ordered operator list to their original hosts, and instruct the algorithms to compute the placement of the rest of operators based on that of the fixed ones. The operators with the lowest load (and so they are at the tail of the list) are chosen to be fixed because they have the least impact on system performance. By tuning the value of θ, it is possible to smoothly tradeoff between the quality of optimization and migration cost as well as computational overhead.

Another possible way to define threshold measurement is in terms of the number of bytes being moved or the amount of time needed by the movement, which has finer granularity than the former. However, these alternatives are not investigated further in this paper.

4.3 Complexity Analysis

Consider the initial mapping algorithms (the analysis applies to periodic optimization algorithms as well). For both algorithms, we need to first sort the operators, which takes $O(m \log m)$. The greedy algorithm has a loop of m iterations, and each iteration needs to evaluate PE on all n hosts. If we ignore the complexity of evaluating PE for the moment, the computational complexity of the greedy algorithm is $O(mn + m \log m)$. The dynamic programming algorithm needs to fill a matrix of size $O(mn)$, and for each entry, it needs to evaluate PE for n times (the 9th line), which gives a total running time of $O(m \log m + mn^2)$.

Now we consider the complexity of evaluating PE. According to the definition of the PE function, it involves evaluating three individual PEs. To evaluate load PE, we need to know the load on the host of interest, which is an operation of $O(1)$ time. To evaluate correlation PE, we need to compute the correlation coefficient of two load time series, which takes $O(k)$ time, where k is the length of the load time series. Finally, to evaluate network PE, we need to account for every data stream that connects the operator to an operator on another host. Since the arity of each operator is usually considered a constant, the worst case running time of evaluating network PE for a single operator is also constant. Combining with the previous analysis, the worst case running time of the greedy algorithm is $O(mnk + m \log m)$, and of the dynamic programming is $O(mn^2k + m \log m)$.

5. EXPERIMENTAL ANALYSIS

In this section, we present the results of our experimental analysis, which tries to answer several questions:

- What is the relationship between system performance and the value of PE?
- How optimal are our algorithms against query graphs with different topologies and configurations?
- How do the weights of individual PEs affect the algorithms' performance, and how can we guide end users to tune those parameters to achieve best performance?
- Do our algorithms outperform existing ones?

For the last question, we compare our algorithms with the one proposed by Xing et al. in [11] (referred to as XZH), which is most relevant to our work.

5.1 The Simulator

To simulate a D-DSMS, we used a custom-built discrete event simulator which is written in Java. The simulator supports filters and binary joins. Since we only consider operators' external behavior and not their actual semantics, other operator types like multi-way joins can always be simulated via combinations of the two. We assume uniprocessor hosts so that all operators on the same host are scheduled sequentially. We choose *FCFS* scheduling policy for all hosts in the following experiments.

Operator input queue length	100 tuples
Operator migration cost ratio	20 ms/tuple
p_o of filters	U(1, 5) ms/tuple
Selectivity of filters	U(0.8, 1.0)
p_o of joins	U(1, 2) ms/tuple
Selectivity of joins	U(0.01, 0.2)
Window size of joins	U(1, 10) tuples
Stable input inter-arrival period	70 ms/tuple
Fluctuating input load period	N(12.0, 0.05) sec
Fluctuating input high rate inter-arrival period	E(50) ms/tuple
Fluctuating ratio ($\frac{\text{high rate}}{\text{low rate}}$)	4
$u_l \cdot d_l$	U(4, 10k) tuples
d_l	U(1, 500) ms
q_h	1
Overload penalty coefficient	20.0
Statistics window size	50 sec
Samples per stat. window (k)	10
Warm-up time	120 sec
w_l	1.0
w_c	1.0
w_n	1.0
θ	0

Table 2: Default Simulation Parameters

Network links are unidirectional channels and a full-duplex link is simulated by bundling two links in different directions. As a common practice, if an operator has more than one downstream neighbour on another host, it will transmit each tuple to that host only once, and duplicate the tuple after arrival. Finally, we use *capacity* and *delay* as two parameters to describe a link. *Capacity* is the maximum number of tuples that can exist on a link at any given time. The bandwidth of the link is calculated as $\frac{capacity}{delay}$.

5.2 Experimental Setup

We test our algorithms under different *schemes*. A scheme is defined by three orthogonal facets: query graph topology, input workload pattern, and the parameters of the query graph and the workload.

Query Graphs: Two types of query graphs are used for experiments. One consists of a number of independent filters *chains*, with the same number of filters on each chain (we call this number *chain length*). Another is *hybrid*, consisting of both filters and binary joins. Operators in both types are initially randomly distributed.

Workload: Workload describes the patterns in which input tuples are generated. There are two types of workloads: *stable* models constant workload where all data sources generate tuples at the same stable input rate. *Fluctuating* is where the average input rate of each data source periodically alternates between a high rate and a low rate. In both periods, tuples' inter-arrival time is drawn from an exponential distribution with the mean set to the current input rate. The duration of high rate and low rate periods follow normal distribution with the same mean and variation. To vary

load correlations between input streams, we align data rate modes of each data source with randomly selected offsets.

Given an instantiation of the other two facets, stable workload leads to deterministic results but the fluctuating one does not. Therefore, for the experiments based on fluctuating workload, we repeat each of them for at least 3 times and report the average result.

Scheme 1: Chain / Stable

n	3
m	9
No. of chains	1
α_h	1
p_o of filters	20 ms/tuple
Selectivity of filters	1.0
$u_l \cdot d_l$	10'000 tuples
d_l	100 ms
Stable input inter-arrival period	100 ms/tuple

Scheme 2: Chain / Stable

n	20
m	200
No. of chains	20
α_h	U(1, 10)

Scheme 3: Chain / Fluctuating

n	10
m	90
No. of chains	3
$alpha_h$	U(1, 2)
p_o of filters	10 ms/tuple
Fluctuating input high rate inter-arrival period	E(100) ms/tuple
Fluctuating ratio ($\frac{\text{high rate}}{\text{low rate}}$)	2.5

Scheme 4: Hybrid / Stable

n	5
m	40
No. of data sources	5
Percentage of filters	50%
α_h	1

Scheme 5: Hybrid / Fluctuating

n	10
m	100
No. of data sources	5
Percentage of filters	85%
α_h	U(1, 10)

Scheme 6: Chain / Fluctuating

n	20
m	200
No. of chains	20
α_h	1
$u_l \cdot d_l$	10'000 tuples
d_l	0
Fluctuating input high rate inter-arrival period	E(15) ms/tuple

Table 3: Scheme-specific Parameter Values

Parameters: We have six different schemes (see the next subsection). Table 2 lists default parameter values across all the schemes, which are used throughout the experiments unless explicitly stated or overridden by scheme-specific values. All values are either deterministic or uniformly (U), normally (N), or exponentially (E) distributed.

Those parameter values were carefully chosen to meet the following criteria: 1) be close to real environments; 2) be close to the parameter values used by [11], for comparable results; and 3) feed the system with medium to very heavy workload (optimization on lightly loaded systems does not produce significant results and are not of interest).

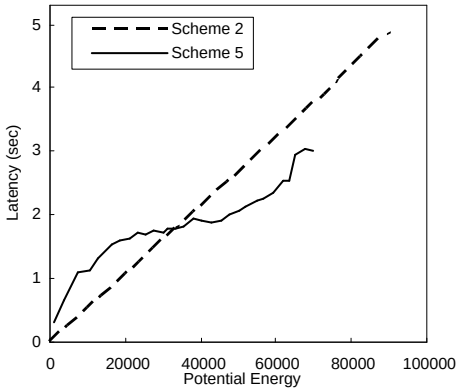

Figure 2: Relationship between PE and Latency

5.2.1 Schemes

To evaluate our algorithms thoroughly, we use six different schemes covering all combinations of query graphs and workloads (see Table 3). Scheme 1 simulates a simple environment, where filters, hosts, and network links are homogeneous, and the query graph only contains one chain with nine filters. This scheme is helpful for intuitive analysis. Schemes 2 to 5 try to cover typical settings of all combinations of query graphs and workloads, from simple to complex. They are expected to simulate real use cases. Scheme 6 is dedicated to a comparison between our algorithm and XZH which will be presented in Section 5.3.7.

5.2.2 Performance Measures

In the following experiments, we measure the average end-to-end latency provided by the D-DSMS. The latency of a tuple t is the difference between its birth time and the time it arrives at an endpoint. Birth time is the time t enters the query graph if it is generated by a data source, or the birth time of the *youngest* tuple that contributes to t's birth if t is generated by an operator. End-to-end latency is then defined as the average latency of all tuples that arrive at endpoints during the period of measurement.

5.3 Experiments and Results

We performed each experiment extensively under all schemes. Since the results are similar among different schemes, we only report interesting ones.

5.3.1 PE vs. Latency

To investigate the relationship of PE to end-to-end latency, we generate 50 variations based on Schemes 2 and 5, and for each of these running schemes, we measure its total PE, and latency. Based on Scheme 2, we generate 20 variations by varying the length of chains from 1 to 20; based on Scheme 5, we generate 30 with the number of hosts ranging from 1 to 30.

Figure 2 shows the relationship between PE and latency.

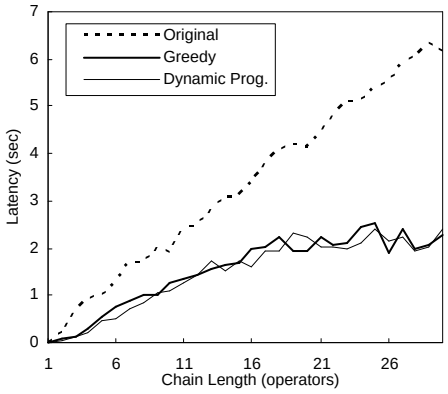

Figure 3: Algorithms' Performance on Latency

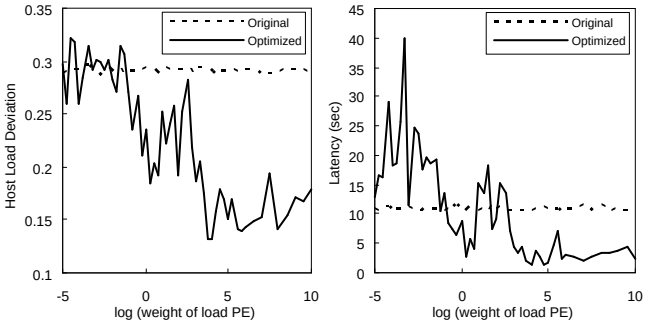

Figure 4: Effect of w_l under Scheme 5

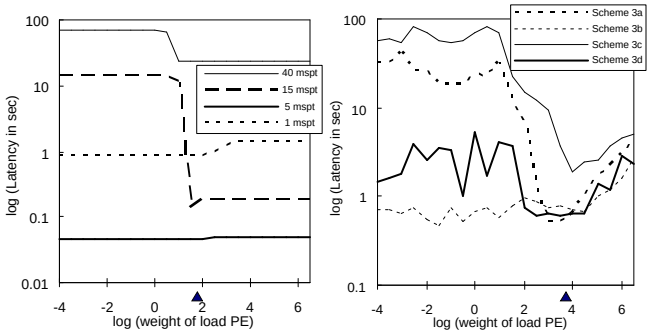

Figure 5: Effect of w_l under Different Schemes

It is interesting that Scheme 2's variations exhibit a perfect linearity, likely caused by the linearity of the chain query graphs. Both curves show positive correlation between PE and the latency, confirming our expectation.

5.3.2 Greedy vs. Dynamic Programming

In this experiment, we compare the two algorithms for solving the optimization problem. That is, the greedy algorithm and dynamic programming (Section 4). The experiment is performed by measuring latency under different chain lengths as shown in Figure 3. The curve marked as "original" is the measurements done before any load distribution algorithm is applied (the remaining plots also follow this convention). The results show no distinguishable difference between the two algorithms. Further investigation on its cause is ongoing. However, our initial theory is that for this problem a simple greedy is sufficient. Hence, in remaining sections we only focus on the greedy algorithm.

5.3.3 Effect of w_l

There are three tunable weights that could noticeably impact the results of our algorithms (see Section 3.2): w_l, w_c, and w_n. Apparently, if we scale them by the same amount, we will only change the value of the PE function but not the behavior of the algorithms. Thus, we can always "transfer" the effect of one weight to the other two, by dividing all of them by the value of that weight. In other words, studying the effects of any two weights is sufficient. To simplify the presentation, we will focus in this section on w_l and w_n.

We first study the effect of w_l under Scheme 5. Under different values of w_l, we record latency as well as load deviation among all hosts. Since load PE reflects the requirement of distributing load among hosts evenly, then it is expected that the load deviation should decrease as w_l increases. The left plot in Figure 4 shows that the trend is as anticipated (x-axes in $\log_{10}$ scale). The latency also decreases as w_l increases, as shown in the right plot in Figure 4. We also notice that improper weight values (e.g. $\log(w_l) < 0$) may lead to a latency even larger than the original one.

Is it true that a larger weight always leads to a lower latency? To verify this, we generate four variants of Scheme 1 and conduct the same experiment on each of them. These variants only differ in their benchmark costs of operators, ranging from 1 to 40 ms/tuple (mspt). The left plot in Figure 5 shows the measured latency with respect to w_l.

The figure illustrates that when the processing cost is high, it makes host load a dominant contributor to the latency. As such, using large w_l values will emphasize the importance of the load PE and make it a dominant term of the PE function which leads to a reduced latency. To the contrary, in cases when the cost is low, a large w_l will instead increase the latency. As a rudimentary conclusion, good performance will be obtained if w_l can properly scale the term of load PE, enabling it to truly reflect the importance of host load compared to the other factors.

A good algorithm should be able to automatically adapt to different situations without frequent adjustment on its parameters. Can we find a "universal" value of w_l that works well in most, if not all, cases? Such a value should usually result in short latency. After extensive experiments, we found that a value around 10^2 satisfies this requirement. For example in Figure 5's left plot, all curves reach or approach their lowest point when the x-coordinate is close to 2. Similarly, the right plot shows the relationship between the latency and w_l under four random variations of Scheme 3. To generate each variation, we randomly distort several parameters that are selected arbitrarily. According to the chart, the algorithm achieves its best result when $\log(w_l)$ is around 4; $\log(w_l) = 2$ also produces acceptable result. In practice, it should be sufficient to initially set w_l to 10^2.

5.3.4 Effect of w_n

We study the effect of w_n in the same way: under Scheme 5, we measure its latency and network utilization with re-

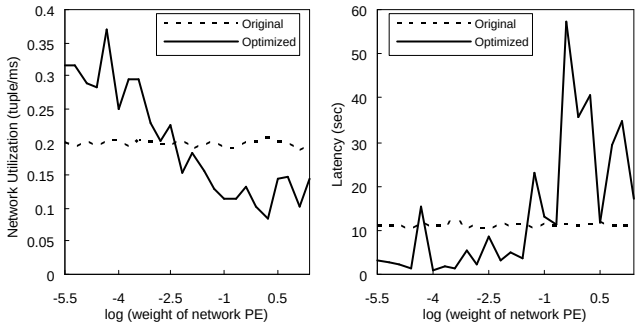

Figure 6: Effect of w_n under Scheme 5

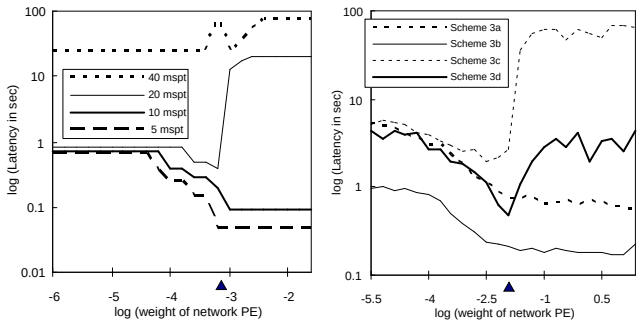

Figure 7: Effect of w_n under Different Schemes

spect to the values of w_n. Given our previous rationale about w_l (Figure 4), host load should be a dominant contributor to the latency of Scheme 5. This explains the increase in latency when network PE is given a high weight by using larger w_n values as shown in the right plot of Figure 6.

Figure 7 shows the relationship between latency and w_n under variants of Scheme 1 with different benchmark costs (left), and under randomly generated variants of Scheme 3 (right). We can see from the left plot that as processing costs increase, the trends for latency shift from decreasing to increasing. This observation confirms our claim that excessive emphasis on one term will cause performance regression when the term's corresponding factor is not dominant.

We found from experiments that a value of w_n around 10^{-3} can achieve good results with high probability. As illustrated in both plots in Figure 7. In summary, $w_l = 10^2$, $w_c = 1$, and $w_n = 10^{-3}$ is our recommended orders of magnitude in practical use. We apply the values of $\langle 100, 1, 0.001 \rangle$ in the subsequent experiments.

5.3.5 *Effect of θ*

Non-zero θ values are used in periodic optimization (Section 4.2) to determine how many operators to move at each period. Figure 8 shows end-to-end latency versus θ under Schemes 3 and 4. It is clear that as θ becomes larger and thus more operators are fixed, the optimized latency becomes worse. When θ reaches 1, no operators are movable and hence the latency becomes equal to the original one.

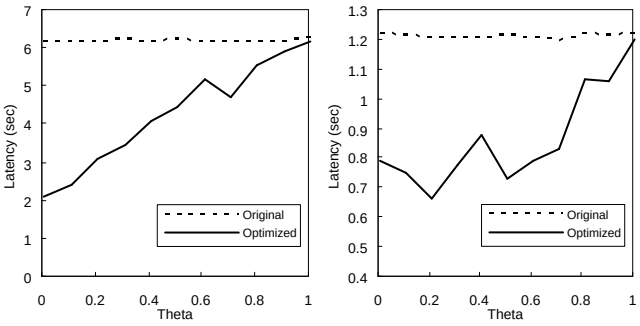

Figure 8: Effect of θ under Scheme 3 and 4

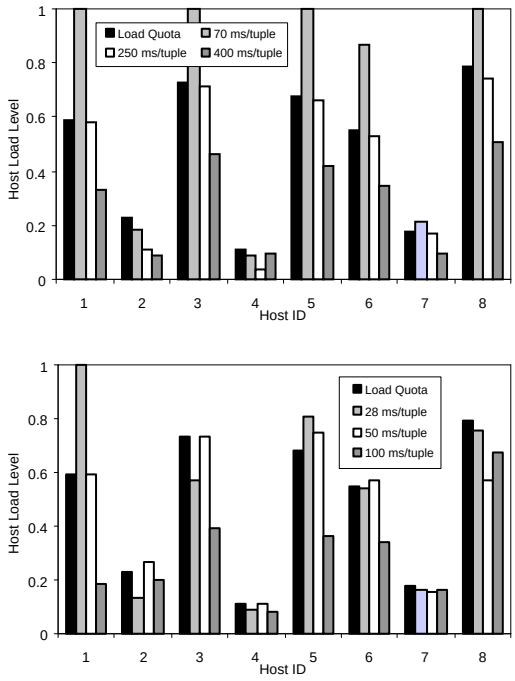

Figure 9: Algorithm's Sensitivity to Load Quota

5.3.6 *Algorithm's Sensitivity to Load Quota*

The default value of host load quotas is set to 1 (Table 2) to enable the algorithms to optimize system performance *unconditionally*. When there exist certain resource constraints, it may be necessary to limit utilization of some hosts.

To test the algorithm's sensitivity to load quotas, we uniformly distribute the quotas between 0 to 1, and compare each host's actual load with its load quota under different system workload. Figure 9 shows the results under Schemes 2 and 5. We reduce the number of hosts on each scheme to 8 for clarity. We observe a good approximation of actual loads to quotas. Note that in the first chart, the inter-arrival period of 70 ms/tuple imposes very heavy workload and causes the four hosts with highest load quotas to be 100% loaded.

5.3.7 *Comparison with XZH Algorithm*

In this experiment, we compare our algorithm with XZH from [11]. Specifically, we implemented the *global load distribution algorithm* [11], which performs load distribution by

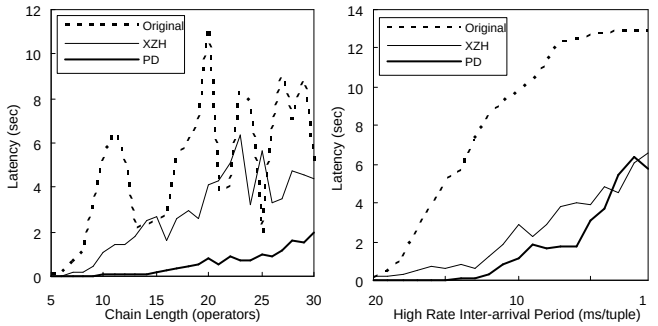

Figure 10: Potential-Driven (PD) vs. XZH in an XZH's Favorable Environment

examining operators' load levels and load correlations with different hosts. In our experiment, all parameters of XZH algorithm are set to the same values as listed in Table 2, and its correlation improvement threshold is set to 0.8. Since XZH algorithm does not consider heterogeneity or network latency, to ensure fairness, we use Scheme 6 for comparison, where we set all capacity coefficients to 1 and network delay to 0. Since only chain shaped graphs were studied in [11], we configure Scheme 6 as a chain graph.

The left plot in Figure 10 is obtained by varying the length of chains from 5 to 30, and the right one is obtained by varying the scheme's high input rates from 20 to 1 ms/tuple. We can see that under this particular homogeneous environment and ideal network conditions, our algorithm still outperforms XZH since it considers operator load correlation (as in [11]) in addition to total host load.

6. RELATED WORK

Xing et al [11] propose a correlation-based approach which, in addition to balancing load across machines, tries to minimize load variance on individual machines, preventing temporary overload. Network limitations are not considered, which could result in frequent transfer of tuples via network links and significant overhead when the network conditions are not perfect. Our work goes further by considering network utilization as well as heterogeneous environments. Also, the PE function that is central to our approach can be extended with more terms, providing further flexibility.

The work in [12] assumes that operator movement is expensive, and try to place them in such a way that the resulting system can withstand large workload variation without overloading or operator migration. Among other differences, our work differs in that we aim to optimize latency. As in [11], [12] assumes homogeneous machines and perfect networks; on the other hand, it uses *CPU capacity* to reflect resource constraint and *operator clustering* as a preprocessing step to prevent costly data crossing the network. [10] attempts to consider the situations where the network transfer delay cannot be ignored by grouping neighboring operators in the initial mapping. When adjustment is necessary, only operators at the boundary are migrated to the neighboring host, so as to avoid creating excessive network traffic. Our work reduces network traffic in a more systematic way, and strikes a balance with other factors.

Other approaches that consider data transmission overhead include [14, 13, 3, 9]. For example, [14, 13] aim to minimize communication cost while maintaining balance of load among hosts. Also, [7] proposes an overlay network between a D-DSMS and the physical network in order to optimize latency and network utilization based on a multi-dimensional cost space. The work in [3] incorporates knowledge of network characteristics such as bandwidth and topology into operator placement algorithms. In the context of sensor networks, [9] considers placing operators along the nodes of a hierarchy to reduce network utilization.

Finally, *intra-operator* load distribution [8, 5] distributes a single operator across multiple machines, as opposed to *inter-operator* load distribution studied in this paper. The former is useful when the granularity of inter-operator load distribution is too coarse to achieve good performance. Integrating intra- and inter-operator techniques remains as a potential future work.

7. CONCLUSIONS

We have studied the problem of efficient load distribution in D-DSMS with the objective of minimizing end-to-end latency. We identify three often conflicting requirements for load distribution that are key to achieving our objective, namely load balancing, load variation minimization, and network utilization reduction. We propose a potential-driven approach to accommodate these conflicting requirements. Under this framework, we propose algorithms for both initial operator mapping and periodic optimization. We report extensive experimental results that analyze the behavior of our approach, and we show that the proposed method outperforms existing ones.

8. REFERENCES

[1] D. J. Abadi et al. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2), 2003.
[2] D. J. Abadi et al. The design of the borealis stream processing engine. In *CIDR*, 2005.
[3] Y. Ahmad and U. Çetintemel. Network-aware query processing for stream-based applications. In *VLDB*, 2004.
[4] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.
[5] X. Gu, P. S. Yu, and H. Wang. Adaptive load diffusion for multiway windowed stream joins. In *ICDE*, 2007.
[6] R. Motwani et al. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.
[7] P. Pietzuch et al. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.
[8] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, 2003.
[9] U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *PODS*, 2005.
[10] Y. Xing. Load distribution for distributed stream processing. In *EDBT Ph.D. Workshop*, 2004.
[11] Y. Xing et al. Dynamic load distribution in the borealis stream processor. In *ICDE*, 2005.
[12] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. Zdonik. Providing resiliency to load variations in distributed stream processing. In *VLDB*, 2006.
[13] Y. Zhou, B. C. Ooi, and K.-L. Tan. Dynamic load management for distributed continuous query systems. In *ICDE*, 2005.
[14] Y. Zhou, B. C. Ooi, K.-L. Tan, and J. Wu. Efficient dynamic operator placement in a locally distributed continuous query system. In *LNCS 4275, OTM*, 2006.