Providing Resiliency to Load Variations in Distributed Stream Processing *

Ying Xing, Jeong-Hyon Hwang, Uğur Çetintemel, and Stan Zdonik
Department of Computer Science, Brown University
{yx, jhhwang, ugur, sbz}@cs.brown.edu

ABSTRACT

Scalability in stream processing systems can be achieved by using a cluster of computing devices. The processing burden can, thus, be distributed among the nodes by partitioning the query graph. The specific operator placement plan can have a huge impact on performance. Previous work has focused on how to move query operators dynamically in reaction to load changes in order to keep the load balanced. Operator movement is too expensive to alleviate short-term bursts; moreover, some systems do not support the ability to move operators dynamically. In this paper, we develop algorithms for selecting an operator placement plan that is resilient to changes in load. In other words, we assume that operators cannot move, therefore, we try to place them in such a way that the resulting system will be able to withstand the largest set of input rate combinations. We call this a *resilient* placement.

This paper first formalizes the problem for operators that exhibit linear load characteristics (e.g., filter, aggregate), and introduces a resilient placement algorithm. We then show how we can extend our algorithm to take advantage of additional workload information (such as known minimum input stream rates). We further show how this approach can be extended to operators that exhibit non-linear load characteristics (e.g., join). Finally, we present prototype- and simulation-based experiments that quantify the benefits of our approach over existing techniques using real network traffic traces.

1. INTRODUCTION

Recently, a new class of applications has emerged in which high-speed streaming data must be processed with very low latency. Financial data analysis, network traffic monitoring and intrusion detection are prime examples of such applications. In these domains, one observes increasing stream rates as more and more data is captured electronically putting stress on the processing ability of stream processing systems. At the same time, the utility of results decays quickly demanding shorter and shorter latencies. Clusters of inexpensive processors allow us to bring distributed processing techniques to bear on these problems, enabling the scalability and availability that these applications demand [7, 17, 4, 23].

Modern stream processing systems [3, 13, 6] often support a data flow architecture in which streams of data pass through specialized operators that process and refine the input to produce results for waiting applications. These operators are typically modifications of the familiar operators of the relational algebra (e.g., filter, join, union). Figure 1 illustrates a typical configuration in which a query network is distributed across multiple machines (nodes). The specific operator distribution pattern has an enormous impact on the performance of the resulting system.

Distributed stream processing systems have two fundamental characteristics that differentiate them from traditional parallel database systems. First, stream processing tasks are long-running continuous queries rather than short-lived one-time queries. In traditional parallel systems, the optimization goal is often minimizing the completion time of a finite task. In contrast, a continuous query has no completion time; therefore, we are more concerned with the latency of individual results.

Second, the data in stream processing systems is pushed from external data sources. Load information needed by task allocation algorithms is often not available in advance or varies significantly and over all time-scales. Medium and long term variations arise typically due to application-specific behaviour; e.g., flash-crowds reacting to breaking news, closing of a stock market at the end of a business day, temperature dropping during night time, etc. Short-term variations, on the other hand, happen primarily because of the event-based aperiodic nature of stream sources as well as the influence of the network interconnecting data sources. Figure 2 illustrates such variations using three real-world traces [1]: a wide-area packet traffic trace (PKT), a wide-area TCP connection trace (TCP), and an HTTP request trace (HTTP). The figure plots the normalized stream rates as a function of time and indicates their standard deviation. Note that similar behaviour is observed at other time-scales due to the self-similar nature of these workloads [9].

A common way to deal with time-varying, unpredictable load variations in a distributed setting is dynamic load distribution. This

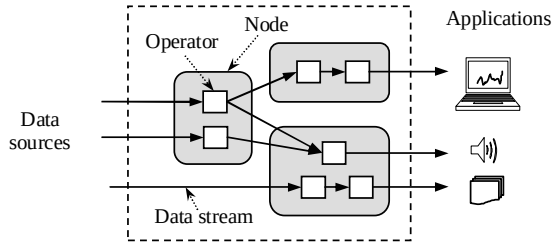


Figure 1: Distributed Stream Processing.

*This work has been supported by the NSF under grants IIS-0086057 and IIS-0325838.

VLDB '06, September 12-15, 2006, Seoul, Korea.

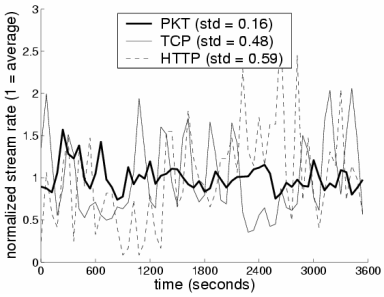


Figure 2: Stream rates exhibit significant variation over time.

approach is suitable for medium-to-long term variations since they persist for relatively long periods of time and are thus rather easy to capture. Furthermore, the overhead of load redistribution is amortized over time. Neither of these properties holds in the presence of short-term load variations. Capturing such transient variations requires frequent statistics gathering and analysis across distributed machines. Moreover, reactive load distribution requires costly operator state migration and multi-node synchronization. In our stream processing prototype, the base overhead of run-time operator migration is on the order of a few hundred milliseconds. Operators with large states will have longer migration times depending on the amount of state transferred. Also, some systems do not provide support for dynamic operator migration. As a result, dealing with short-term load fluctuations by frequent operator re-distribution is typically prohibitive.

In this paper, we explore a novel approach to operator distribution, namely that of identifying operator distributions that are *resilient* to unpredictable load variations. Informally, a resilient distribution is one that does not become overloaded easily in the presence of bursty and fluctuating input rates. Standard load distribution algorithms optimize system performance with respect to a single load point, which is typically the load perceived by the system in some recent time period. The effectiveness of such an approach can become arbitrarily poor and even infeasible when the observed load characteristics are different from what the system was originally optimized for. Resilient distribution, on the other hand, does not try to optimize for a single load point. Instead, it enables the system to "tolerate" a large set of load points without operator migration.

It should be noted that static, resilient operator distribution is not in conflict with dynamic operator distribution. For a system that supports dynamic operator migration, the techniques presented here can be used to place operators with large state size. Lighter-weight operators can be moved more frequently using a dynamic algorithm (e.g., the correlation-based scheme that we proposed earlier [23]). Moreover, resilient operator distribution can be used to provide a good initial plan.

We focus on static operator distribution algorithms. More specifically, we model the load of each operator as a function of the system input stream rates. For given input stream rates and a given operator distribution plan, the system is either feasible (none of the nodes are overloaded) or overloaded. The set of all feasible input rate combinations defines a *feasible set*. Figure 3 illustrates an example of a feasible set for two input streams. For unpredictable workloads, we want to make the system feasible for as many input rate points as possible. Thus, the optimization goal of resilient operator distribution is to maximize the size of the feasible set.

In general, finding the optimal operator distribution plan requires comparing the feasible set size of different operator distribution plans. This problem is intractable for a large number of operators or a large number of input streams. In this paper, we present

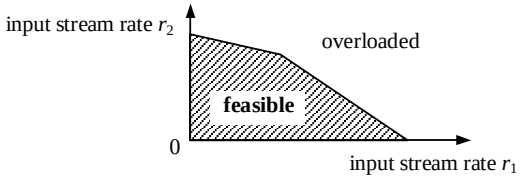


Figure 3: Feasible set on input stream rate space.

a greedy operator distribution algorithm that can find suboptimal solutions without actually computing the feasible set size of any operator distribution plan. The contributions of this work can be summarized as follows:

1. We formalize the resilient operator distribution problem for systems with linear load models, where the load of each operator can be expressed as a linear function of system input stream rates. We identify a tight superset of all possible feasible sets called the *ideal feasible set*. When this set is achieved, the load from each input stream is perfectly balanced across all nodes (in proportional to the nodes' CPU capacity).
2. The ideal feasible set is in general unachievable. We propose two novel operator distribution heuristics to make the achieved feasible set as close to the ideal feasible set as possible. The first heuristic tries to balance the load of each input stream across all nodes. The second heuristic focuses on the combination of the "impact" of different input streams on each node to avoid creating bottlenecks. We then present a resilient operator distribution algorithm that seamlessly combines both heuristics.
3. We present a generalization of our approach that can transform a nonlinear load model into a linear load model. Using this transformation, our resilient algorithm can be applied to any system.
4. We present algorithm extensions that take into account the communications costs and knowledge of specific workload characteristics (i.e., lower bound on input stream rates) to optimize system performance.

Our study is based on extensive experiments that evaluate the relative performance of our algorithm against several other load distribution techniques. We conduct these experiments using both a simulator and the Borealis distributed stream processing prototype [2] on real-world network traffic traces. The results demonstrate that our algorithm is much more robust to unpredictable or bursty workloads than traditional load distribution algorithms.

The rest of the paper is organized as follows: In Section 2, we introduce our distributed stream processing model and formalize the problem. Section 3 presents our optimization approach. We discuss the operator distribution heuristics in detail in Section 4 and present the resilient operator distribution algorithm in Section 5. Section 6 discusses the extensions of this algorithm. Section 7 examines the performance of our algorithm. We discuss related work in Section 8 and present concluding remarks in Section 9.

2. MODEL & PROBLEM STATEMENT

2.1 System Model

We assume a computing cluster that consists of loosely coupled, shared-nothing computers because this is widely recognized as the most cost-effective, incrementally scalable parallel architecture today. We assume the available CPU cycles on each machine for stream data processing are fixed and known. We further assume that the cluster is interconnected by a high-bandwidth local area network, thus bandwidth is not a bottleneck. For simplicity, we

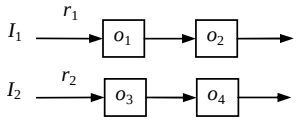


Figure 4: Example query graph.

initially assume that the CPU overhead for data communication is negligible compared to that of data processing. We relax this assumption in Section 6.3.

The tasks to be distributed on the machines are data-flow-style acyclic query graphs (e.g., in Figure 1), which are commonly used for stream processing (e.g., [3, 13, 6]). In this paper, we consider each continuous query operator as the minimum task allocation unit.

2.2 Load Model

We assume there are n nodes ($N_i, i = 1, \cdots, n$), m operators ($o_j, j = 1, \cdots, m$), and d input streams ($I_k, k = 1, \cdots, d$) in the system. In general, an operator may have multiple input streams and multiple output streams. The *rate* of a stream is defined as the number of data items (tuples) that arrive at the stream per unit time. We define the *cost* of an operator (with respect to an input stream) as the average number of CPU cycles needed to process an input tuple from that input stream per unit time. The *selectivity* of an operator (with respect to an input and an output stream) is defined as the ratio of the output stream rate to the input stream rate. We define the *load* of an operator per unit time as the CPU cycles needed by the operator per unit time to process its input tuples. We can thus write the load of each operator as a function of operator costs, selectivities and system input stream rates ($r_k, k = 1, \cdots, d$).

Example 1: Consider the simple query graph shown in Figure 4. Assume the rate of input stream I_k is r_k for $k = 1, 2$, and operator o_j has cost c_j and selectivity s_j for $j = 1, \cdots, 4$. The load of these operators is then computed as

$$\begin{aligned} load(o_1) &= c_1 r_1 \\ load(o_2) &= c_2 s_1 r_1 \\ load(o_3) &= c_3 r_2 \\ load(o_4) &= c_4 s_3 r_2 \ . \end{aligned}$$

Our operator distribution algorithm is based on a *linear load model* where the load of each operator can be written as a linear function, i.e.

$$load(o_j) = l_{j1} x_1 + \cdots + l_{jd} x_d, \quad j = 1, \cdots, m,$$

where $x_1, \cdots, x_d$ are variables and l_{jk} are constants. For simplicity of exposition, we first assume that the system input stream rates are variables and the operator costs and selectivities are constant. Under this assumption, all operator load functions are linear functions of system input stream rates. Assuming stable selectivity, operators that satisfy this assumption include union, map, aggregate, filter etc. In Section 6.2, we relax this assumption and discuss systems with operators whose load cannot be written as linear functions of input stream rates (e.g., time-window based joins).

2.3 Definitions and Notations

We now introduce the key notations and definitions that are used in the remainder of the paper. We also summarize them in Table 1.

We represent the distribution of operators on the nodes of the system by the *operator allocation matrix*:

$$A = \{a_{ij}\}_{n \times m},$$

where $a_{ij} = 1$ if operator o_j is assigned to node N_i and $a_{ij} = 0$ otherwise.

Given an operator distribution plan, the load of a node is defined as the aggregate load of the operators allocated at that node. We express the load functions of the operators and nodes as:

$$\begin{aligned} load(o_j) &= l^o_{j1} r_1 + \cdots + l^o_{jd} r_d \ , \quad j = 1, \cdots, m, \\ load(N_i) &= l^n_{i1} r_1 + \cdots + l^n_{id} r_d \ , \quad i = 1, \cdots, n, \end{aligned}$$

where l^o_{jk} is the *load coefficient* of operator o_j for input stream I_k and l^n_{ik} is the *load coefficient* of node N_i for input stream I_k. As shown in Example 1 above, the load coefficients can be computed using the costs and selectivities of the operators and are assumed to be constant unless otherwise specified. Putting the load coefficients together, we get the *load coefficient matrices*:

$$L^o = \{l^o_{jk}\}_{m \times d}, \quad L^n = \{l^n_{ik}\}_{n \times d}.$$

It follows from the definition of the operator allocation matrix that

$$\begin{aligned} L^n &= A L^o, \\ \sum_{i=1}^{n} l^n_{ik} &= \sum_{j=1}^{m} l^o_{jk} \ , \quad k = 1, \cdots d. \end{aligned}$$

Example 2: We now present a simple example of these definitions using the query graph shown in Figure 4. Assume the following operator costs and selectivities: c_1= 14, c_2= 6, c_3= 9, c_4= 14 and s_1= 1, s_3= 0.5. Further assume that there are two nodes in the system, N_1 and N_2, with capacities C_1 and C_2, respectively. In Table 2, we show the corresponding operator load coefficient matrix L^o and, for three different operator allocation plans (Plan (a), Plan (b), and Plan(c)), the resulting operator allocation matrices and node load coefficient matrices.

Next, we introduce further notations to provide a formal definition for the feasible set of an operator distribution plan. Let $R = (r_1, \cdots, r_d)^T$ be the vector of system input stream rates. The load of node N_i can then be written as $L^n_i R$, where L^n_i is the ith row of matrix L^n. Let $C = (C_1, \cdots, C_n)^T$ be the vector of available CPU cycles (i.e., CPU capacity) of the nodes. Then N_i is

Table 1: Notation.

n	number of nodes
m	number of operators
d	number of system input streams
N_i	the ith node
o_j	the jth operator
I_k	the kth input stream
$C = (C_1, \cdots, C_n)^T$	available CPU capacity vector
$R = (r_1, \cdots, r_d)^T$	system input stream rate vector
l^n_{ik}	load coefficient of N_i for I_k
l^o_{jk}	load coefficient of o_j for I_k
$L^n = \{l^n_{ik}\}_{n \times d}$	node load coefficient matrix
$L^o = \{l^o_{jk}\}_{m \times d}$	operator load coefficient matrix
$A = \{a_{ij}\}_{n \times m}$	operator allocation matrix
D	workload set
$F(A)$	feasible set of A
C_T	total CPU capacity of all nodes
l_k	sum of load coefficients of I_k
w_{ik}	$(l^n_{ik}/l_k) \ / \ (C_i/C_T)$, weight of I_k on N_i
$W = \{w_{ik}\}_{n \times d}$	weight matrix

Table 2: Three example operator distribution plans.

L^o	Plan	A	L^n
$\begin{pmatrix} 14 & 0 \\ 6 & 0 \\ 0 & 9 \\ 0 & 7 \end{pmatrix}$	(a)	$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 20 & 0 \\ 0 & 16 \end{pmatrix}$
	(b)	$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 14 & 9 \\ 6 & 7 \end{pmatrix}$
	(c)	$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$	$\begin{pmatrix} 14 & 7 \\ 6 & 9 \end{pmatrix}$

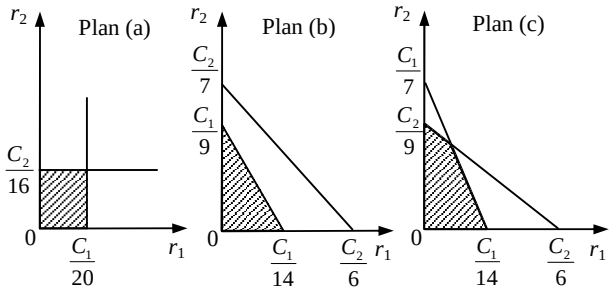


Figure 5: Feasible sets for various distribution plans.

feasible if and only if $L_i^n R \leq C_i$. Therefore, the system is feasible if and only if $L^n R \leq C$. The set of all possible input stream rate points is called the *workload set* and is referred to by D. For example, if there are no constraints on the input stream rates, then $D = \{R : R \geq 0\}$.

Feasible Set Definition: *Given a CPU capacity vector C, an operator load coefficient matrix L^o, and a workload set D, the feasible set of the system under operator distribution plan A (denoted by $F(A)$) is defined as the set of all points in the workload set D for which the system is feasible, i.e.,*

$$F(A) = \{R : R \in D,\ AL^o R \leq C\}.$$

In Figure 5, we show the feasible sets (the shaded regions) of the distribution plans of Example 2. We can see that different operator distribution plans can result in very different feasible sets.

2.4 Problem Statement

In order to be resilient to time-varying, unpredictable workloads and maintain quality of service (i.e., consistently produce low latency results), we aim to maximize the size of the feasible set of the system through intelligent operator distribution. We formally state the corresponding optimization problem as follows:

The Resilient Operator Distribution (ROD) problem: *Given a CPU capacity vector C, an operator load coefficient matrix L^o, and a workload set D, find an operator allocation matrix A^* that achieves the largest feasible set size among all operator allocation plans, i.e., find*

$$A^* = \arg\max_A \int \cdots \int_{F(A)} 1\ dr_1 \cdots dr_d.$$

In the equation above, the multiple integral over $F(A)$ represents the size of the feasible set of A. Note that A^*may not be unique.

ROD is different from the canonical linear programming and nonlinear programming problems with linear constraints on feasible sets. The latter two problems aim to maximize or minimize a (linear or nonlinear) objective function on a fixed feasible set (with fixed linear constraints) [10, 15], whereas in our problem, we attempt to maximize the size of the feasible set by appropriately *constructing* the linear constraints through operator distribution. To the best of our knowledge, our work is the first to study this problem in the context of load distribution.

A straightforward solution to ROD requires enumerating all possible allocation plans and comparing their feasible set sizes. Unfortunately, the number of different distribution plans is $n^m/n!$. Moreover, even computing the feasible set size of a single plan (i.e., a d dimensional multiple integral) is expensive since the Monte Carlo integration method, which is commonly used in high dimensional integration, requires at least $O(2^d)$ sample points [19]. As a result, finding the optimal solution for this problem is intractable for a large d or large m.

3. OPTIMIZATION FUNDAMENTALS

Given the intractability of ROD, we explore a heuristic-driven strategy. We first explore the characteristics of an "ideal" plan using a linear algebraic model and its corresponding geometrical interpretation. We then use this insight to derive our solution.

3.1 Feasible Set and Node Hyperplanes

We here examine the relationship between the feasible set size and the node load coefficient matrix. Initially, we assume no knowledge about the expected workload and thus let $D = \{R : R \geq 0\}$ (we relax this assumption in Section 6.1). The feasible set that results from the node load coefficient matrix L^n is defined by

$$F'(L^n) = \{R : R \in D, L^n R \leq C\}.$$

This is a convex set in the nonnegative space below n hyperplanes, where the hyperplanes are defined by

$$l_{i1}^n r_1 + \cdots + l_{id}^n r_d = C_i, \quad i = 1, \cdots, n.$$

Note that the ith hyperplane consists of all points that render node N_i fully loaded. In other words, if a point is above this hyperplane, then N_i is overloaded at that point. The system is thus feasible at a point if and only if the point is on or below all of the n hyperplanes defined by $L^n R = C$. We refer to these hyperplanes as *node hyperplanes*.

For instance, in Figure 5, the node hyperplanes correspond to the lines above the feasible sets. Because the node hyperplanes collectively determine the shape and size of the feasible set, the feasible set size can be optimized by constructing "good" node hyperplanes or, equivalently, by constructing a "good" node load coefficient matrix.

3.2 Ideal Node Load Coefficient Matrix

We now present and prove a theorem that characterizes an *ideal* node load coefficient matrix.

THEOREM 1. *Given load coefficient matrix $L^o = \{l_{jk}^o\}_{m\times d}$ and node capacity vector $C = (C_1, \cdots, C_n)^T$, among all n by d matrices $L^n = \{l_{ik}^n\}_{n\times d}$ that satisfy the constraint*

$$\sum_{i=1}^{n} l_{ik}^n = \sum_{j=1}^{m} l_{jk}^o, \tag{1}$$

the matrix $L^{n*} = \{l_{ik}^{n*}\}_{n\times d}$ with

$$l_{ik}^{n*} = l_k \frac{C_i}{C_T},\ \text{where } l_k = \sum_{j=1}^{m} l_{jk}^o,\ C_T = \sum_{i=1}^{n} C_i,$$

achieves the maximum feasible set size, i.e.,

$$L^{n*} = \arg\max_{L^n} \int \cdots \int_{F'(L^n)} 1\ dr_1 \cdots dr_d,$$

PROOF. All node load coefficient matrices must satisfy constraint 1. It is easy to verify that L^{n*} also satisfies this constraint. Now, it suffices to show that L^{n*} has the largest feasible set size among all L^n that satisfy constraint 1.

From $L^n R \leq C$, we have that

$$(1 \cdots 1) \begin{pmatrix} l_{11}^n & \cdots & l_{1d}^n \\ \vdots & \ddots & \vdots \\ l_{n1}^n & \cdots & l_{nd}^n \end{pmatrix} \begin{pmatrix} r_1 \\ \vdots \\ r_d \end{pmatrix} \leq (1 \cdots 1) \begin{pmatrix} C_1 \\ \vdots \\ C_n \end{pmatrix},$$

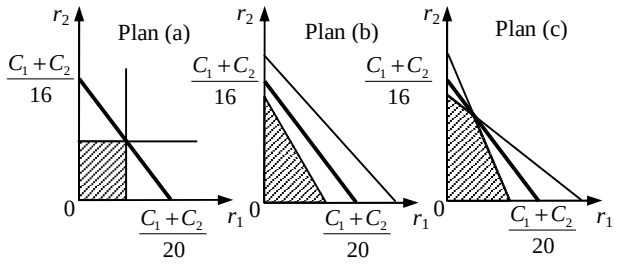


Figure 6: Ideal hyperplanes and feasible sets.

which can be written as

$$l_1 r_1 + \cdots + l_d r_d \leq C_T. \quad (2)$$

Thus, any feasible point must belong to the set

$$F^* = \{R : R \in D,\ l_1 r_1 + \cdots + l_d r_d \leq C_T\}.$$

In other words, $F^*$is the superset of any feasible set. It then suffices to show that $F'(L^{n*}) = F^*$.

There are n constraints in $L^{n*}R \leq C$ (each row is one constraint). For the ith row, we have that

$$l_1 \frac{C_i}{C_T} r_1 + \cdots + l_d \frac{C_i}{C_T} r_d \leq C_i\ ,$$

which is equivalent to inequality 2. Since all n constraints are the same, we have that $F'(L^{n*}) = F^*$. □

Intuitively, Theorem 1 says that the load coefficient matrix that balances the load of each stream *perfectly* across all nodes (in proportion to the relative CPU capacity of each node) achieves the maximum feasible set size. Such a load coefficient matrix may not be realizable by operator distribution, i.e., there may not exist an operator allocation matrix A such that $AL^o = L^{n*}$ (the reason why L^{n*} is referred to as "ideal"). Note that the ideal coefficient matrix is independent of the workload set D.

When the ideal node load coefficient matrix is obtained, all node hyperplanes overlap with the ideal hyperplane. The largest feasible set achieved by the ideal load coefficient matrix is called the *ideal feasible set* (denoted by F^*). It consists of all points that fall below the *ideal hyperplane* defined by

$$l_1 r_1 + \cdots + l_d r_d = C_T.$$

We can compute the size of the ideal feasible set as:

$$V(F^*) = \int_{F^*} \cdots \int 1\ dr_1 \cdots dr_d = \frac{C_T^d}{d!} \cdot \prod_{k=1}^{d} \frac{1}{l_k}.$$

Figure 6 illustrates the ideal hyperplane (represented by the thick lines) and the feasible sets of Plan (a), (b) and (c) in Example 2. It is easy to see that none of the shown distribution plans are ideal. In fact, no distribution plan for Example 2 can achieve the ideal feasible set.

3.3 Optimization Guidelines

The key high-level guideline that we will rely on to maximize feasible set size is to make the node hyperplanes as close to the ideal hyperplane as possible.

To accomplish this, we first normalize the ideal feasible set by changing the coordinate system. The normalization step is necessary to smooth out high variations in the values of load coefficients of different input streams, which may adversely bias the optimization.

Let $x_k = l_k r_k / C_T$. In the new coordinate system with axis x_1 to x_k, the corresponding node hyperplanes are defined by

$$\frac{l_{i1}^n}{l_1} x_1 + \cdots + \frac{l_{id}^n}{l_d} x_d = \frac{C_i}{C_T}\ , \quad i = 1, \cdots, n.$$

The corresponding ideal hyperplane is defined by

$$x_1 + \cdots + x_d = 1.$$

By the change of variable theorem for multiple integrals [21], we have that the size of the original feasible set equals the size of the normalized feasible set multiplied by a constant c, where $c = C_T^d \Big/ \prod_{k=1}^{d} l_k$. Therefore, the goal of maximizing the original feasible set size can be achieved by maximizing the normalized feasible set size.

We now define our goal more formally using our algebraic model: Let matrix

$$W = \{w_{ik}\}_{n \times d} = \{l_{ik}^n / l_{ik}^{n*}\}_{n \times d}.$$

$w_{ik} = (l_{ik}^n / l_k) / (C_i / C_T)$ is the percentage of the load from stream I_k on node N_i divided by the normalized CPU capacity of N_i. Thus, we can view w_{ik} as the "weight" of stream I_k on node N_i and view matrix W as a normalized form of a load distribution plan. Matrix W is also called the *weight matrix*.

Note that the equations of the node hyperplanes in the normalized space is equivalent to

$$w_{i1} x_1 + \cdots + w_{id} x_d = 1, \quad i = 1, \cdots, n.$$

Our goal is then to make the normalized node hyperplanes close to the normalized ideal hyperplane, i.e. make

$$W_i = (w_{i1}, \cdots, w_{id}) \text{ close to } (1, \cdots, 1),$$

for $i = 1, \cdots, n$.

For brevity, in the rest of this paper, we assume that all terms, such as hyperplane and feasible set, refer to the ones in the normalized space, unless specified otherwise.

4. HEURISTICS

We now present two heuristics that are guided by the formal analysis presented in the previous section. For simplicity of exposition, we motivate and describe the heuristics from a geometrical point of view. We also formally present the pertinent algebraic foundations as appropriate.

4.1 Heuristic 1: MaxMin Axis Distance

Recall that we aim to make the node hyperplanes converge to the ideal hyperplane as much as possible. In the first heuristic, we try to push the intersection points of the node hyperplanes (along each axis) to the intersection point of the ideal hyperplane as much as possible. In other words, we would like to make the *axis distance* of each node hyperplane as close to that of the ideal hyperplane as possible. We define the axis distance of hyperplane h on axis a as the distance from the origin to the intersection point of h and a. For example, this heuristic prefers the plan in Figure 7(b) to the one in Figure 7(a).

Note that the axis distance of the ith node hyperplane on the kth axis is $1/w_{ik}$, and the axis distance of the ideal hyperplane is one on all axes (e.g. Figure 7(a)). Thus, from the algebraic point of view, this heuristic strives to make each entry of W_i as close to 1 as possible.

Because $\sum_i l_{ik}^n$ is fixed for each k, the optimization goal of making w_{ik} close to one for all k is equivalent to balancing the load of each input stream across the nodes in proportion to the nodes' CPU

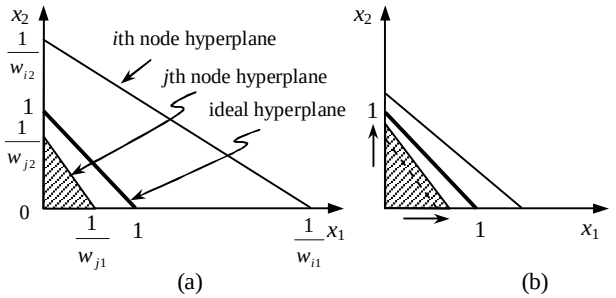


Figure 7: Illustrating MaxMin Axis Distance.

capacities. This goal can be achieved by maximizing the minimum axis distance of the node hyperplanes on each axis, i.e., we want to maximize

$$\min_i \frac{1}{w_{ik}}, \quad \text{for } k = 1, \cdots, d.$$

We therefore refer to this heuristic as *MaxMin Axis Distance* (MMAD). The arrows in Figure 7(b) illustrate how MMAD pushes the intersection points of the node hyperplanes that are closest to the origin to that of the ideal hyperplane.

It can be proven that when the minimum axis distance is maximized for axis k, all the node hyperplane intersection points along axis k converge to that of the ideal hyperplane. In addition, the achieved feasible set size is bounded by

$$V(F^*) \cdot \prod_{k=1}^{d} \min_i \frac{1}{w_{ik}}$$

from below, where $V(F^*)$ is the ideal feasible set size (we omit the proof for brevity). Therefore, MMAD tries to maximize a lower bound for feasible set size, and this lower bound is close to the ideal value when all axis distances are close to 1.

On the downside, the key limitation of MMAD is that it does not take into consideration how to combine the weights of different input streams at each node. This is best illustrated by a simple example as depicted by Figure 8. Both plans in Figure 8 are deemed equivalent by MMAD, since their axis intersection points are exactly the same. They do, however, have significantly different feasible sets. Obviously, if the largest weights for each input stream are placed on the same node (e.g., the one with the lowest hyperplane in Figure 8(a)), the corresponding node becomes the bottleneck of the system because it always has more load than the other node. Next, we will describe another heuristic that addresses this limitation.

4.2 Heuristic 2: MaxMin Plane Distance

Intuitively, MMAD pushes the intersection points of the node hyperplanes closer to those of the ideal hyperplane using the axis distance metric. Our second heuristic, on the other hand, pushes the node hyperplanes directly towards the ideal hyperplane using the *plane distance* metric. The plane distance of an hyperplane h is the distance from the origin to h. Our goal is thus to maximize the minimum plane distance of all node hyperplanes. We refer to this heuristic as *MaxMin Plane Distance* (MMPD).

Another way to think about this heuristic is to imagine a partial hypersphere that has its center at the origin and its radius r equal the minimum plane distance (e.g., Figure 8). Obviously, MMAD prefers Figure 8(b) to Figure 8(a) because the former has a larger r. The small arrow in Figure 8(b) illustrates how MMPD pushes the hyperplane that is the closest to the origin in terms of plane distance towards the ideal hyperplane.

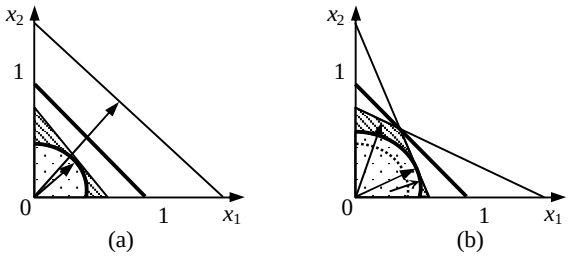


Figure 8: Illustrating MaxMin Plane Distance.

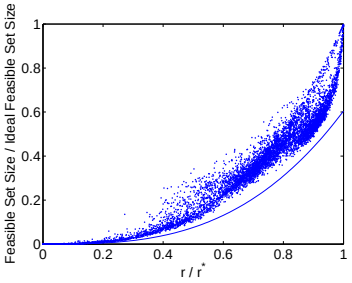


Figure 9: Relationship between r and the feasible set size.

The plane distance of the ith hyperplane is computed as:

$$\frac{1}{\sqrt{w_{i1}^2 + \cdots + w_{id}^2}} = \frac{1}{\|W_i\|},$$

where $\|W_i\|$ denotes the second norm of the ith row vector of W. Thus, the value we would like maximize is:

$$r = \min_i \frac{1}{\|W_i\|}.$$

By maximizing r, we maximize the size of the partial hypersphere, which is a lower bound on the feasible set size. To further examine the relationship between r and the feasible set size, we generated random node load coefficient matrices and plotted the ratios of their feasible-set-size / ideal-feasible-set-size vs. the ratio of $r/\,r^*$ ($r^*$is the distance from the origin to the ideal hyperplane). Figure 9 shows the results of 10000 random load coefficient matrices with 10 nodes and 3 input streams. We see a trend that both the upper bound and lower bound of the feasible-set-size ratio increase when $r/\,r^*$ increases. The curve in the graph is the computed lower bound using the volume function of hyperspheres, which is a constant times r^d [22]. For different n and d, the upper bound and lower bound differs from each other; however, the trend remains intact. This trend is an important ground for the effectiveness of MMPD.

Intuitively, by maximizing r, i.e., minimizing the largest weight vector norm of the nodes, we avoid having nodes with large weights arising from multiple input streams. Nodes with relatively larger weights often have larger load/capacity ratios than other nodes at many stream rate points. Therefore, MMPD can also be said to balance the load of the nodes (in proportion to the nodes' capacity) *for multiple workload points*. Notice that this approach sharply contrasts with traditional load balancing schemes that optimize for single workload points.

5. THE ROD ALGORITHM

We now present a greedy operator distribution algorithm that seamlessly combines the heuristics discussed in the previous section. The algorithm consists of two phases: the first phase orders the operators and the second one greedily places them on the available nodes. The pseudocode of the algorithm is shown in Figure 10.

Initialization

$C_T \leftarrow C_1 + \cdots + C_n$

for $k = 1, \cdots, d$, $l_k \leftarrow l^o_{1k} + \cdots + l^o_{mk}$

for $i = 1, \cdots, n$, $j = 1, \cdots, m$, $a_{ij} \leftarrow 0$

for $i = 1, \cdots, n$, $k = 1, \cdots, d$, $l^n_{ik} \leftarrow 0$

Operator Ordering

Sort operators by $\sqrt{{l^o_{j1}}^2 + \cdots + {l^o_{jd}}^2}$ in descending order (let $h_1, \cdots, h_m$ be the sorted operator indices)

Operator Assignment

for $j = h_1, \cdots, h_m$ (assign operator o_j)

  class I nodes $\leftarrow \phi$, class II nodes $\leftarrow \phi$

  for $i = 1, \cdots, n$ (classify nodes)

    for $k = 1, \cdots, d$, $w'_{ik} \leftarrow \left(\left(l^n_{ik} + l^o_{jk}\right)/l_k\right) / (C_i/C_T)$

    if $w'_{ik} \leq 1$ for all $k = 1, \cdots, d$

      add N_i to class I nodes

    else

      add N_i to class II nodes

  if class I is not empty

    select a destination node from class I

  else

    select the node with $\min_i 1 \Big/ \sqrt{{w'_{i1}}^2 + \cdots + {w'_{id}}^2}$

  (Assume N_s is the selected node. Assign o_j to N_s)

    $a_{sj} \leftarrow 1$;

    for $k = 1, \cdots, d$, $l^n_{sk} \leftarrow 1^n_{sk} + l^o_{jk}$

Figure 10: The ROD algorithm pseudocode.

5.1 Phase 1: Operator Ordering

This phase sorts the operators in descending order based on the second norm of their load coefficient vectors. The reason for this sorting order is to enable the second phase to place "high impact" operators (i.e., those with large norms) early on in the process, since dealing with such operators late may cause the system to significantly deviate from the optimal results. Similar sorting orders are commonly used in greedy load balancing and bin packing algorithms [8].

5.2 Phase 2: Operator Assignment

The second phase goes through the ordered list of operators and iteratively assigns each to one of the n candidate nodes. Our basic destination node selection policy is greedy: at each step, the operator assignment that *minimally* reduces the final feasible set size is chosen.

At each step, we separate nodes into two classes. Class I nodes consist of those that, if chosen for assignment, will not lead to a reduction in the final feasible set. Class II nodes are the remaining ones. If Class I nodes exist, one of them is chosen as the destination node using a goodness function (more on this choice below). Otherwise, the operator is assigned to the Class II node with the maximum *candidate* plane distance (i.e., the distance after the assignment).

Let us now describe the algorithm in more detail while providing geometrical intuition. Initially, all the nodes are empty. Thus, all the node hyperplanes are at infinity. The node hyperplanes move closer to the origin as operators get assigned. The feasible set size at each step is given by the space that is below all the node hyperplanes. Class I nodes are those whose candidate hyperplanes are above the ideal hyperplane, whereas the candidate hyperplanes of Class II nodes are either entirely below, or intersect with, the ideal hyperplane. Figure 11(a) and 11(b) show, respectively, the current and candidate hyperplanes of three nodes, as well as the ideal hyperplane.

Since we know that the feasible set size is bounded by the ideal hyperplane, at a given step, choosing a node in Class I will not reduce the possible space for the final feasible set. Figure 11(c) shows an example of the hyperplanes of a Class I node. Notice that as we assign operators to Class I nodes, we push the axis intersection points closer to those of the ideal hyperplane, thus follow the MMAD heuristic. If no Class I nodes exist, then we have to use a Class II node and, as a result, we inevitably reduce the feasible set. An example of two Class II nodes and the resulting decrease in the feasible set size are shown in Figure 11(d). In this case, we follow the MMPD heuristic and use the plane distance to make our decision by selecting the node that has the largest candidate plane distance.

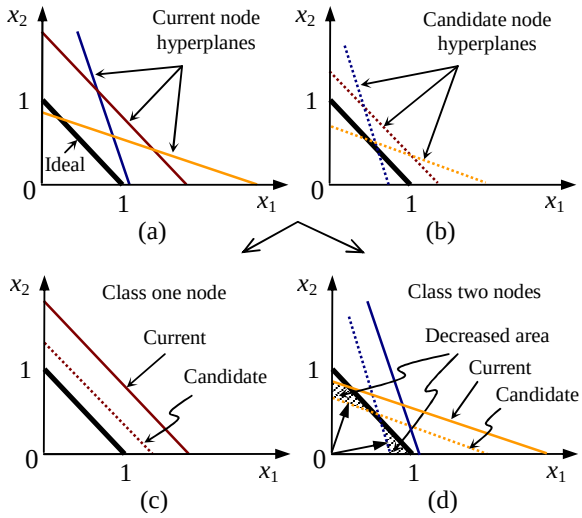


Figure 11: Node selection policy example.

As described above, choosing any node from Class I does not affect the final feasible set size in this step. Therefore, a random node can be selected or we can choose the destination node using some other criteria. For example, we can choose the node that results in the minimum number of inter-node streams to reduce data communication overheads for scenarios where this is a concern.

6. ALGORITHM EXTENSIONS

6.1 General Lower Bounds on Input Rates

We have so far leveraged no knowledge about the expected workload, assuming $D = \{R : R \geq 0\}$. We now present an extension where we allow more general, non-zero lower bound values for the stream rates, assuming:

$$D = \{R : R \geq B,\ B = (b_1, \cdots, b_d)^T,\ b_k \geq 0 \text{ for } k = 1, \cdots, d\}.$$

This general lower bound extension is useful in cases where it is known that the input stream rates are strictly, or likely, larger than a workload point B. Using point B as the lower bound is equivalent to ignoring those workload points that never or seldom happen; i.e., we optimize the system for workloads that are more likely to happen.

The operator distribution algorithm for the general lower bound is similar to the base algorithm discussed before. Recall that the ideal node load coefficient matrix does not depend on D. Therefore, our first heuristic, MMAD, remains the same. In the second heuristic, MMPD, because the lower bound of the feasible set size changes, the center of the partial hypersphere should also change. In the normalized space, the lower bound corresponds to the point $B' = (b_1 l_1/C_T, \cdots, b_d l_d/C_T)^T$. In this case, we want to maximize the radius of the partial hypersphere centered at B' within the

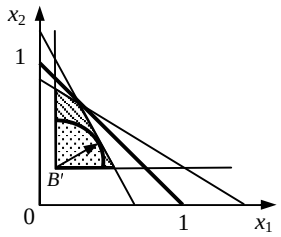


Figure 12: Feasible set with lower bound B'.

normalized feasible set (e.g., Figure 12). The formula of its radius r is

$$r = \min_i \frac{1 - W_i B'}{\|W_i\|}.$$

In the ROD algorithm, we simply replace the distance from the origin to the node hyperplanes with the distance from the lower bound to the node hyperplanes.

6.2 Nonlinear Load Models

Our discussion so far has assumed linear systems. In this section, we generalize our discussion to deal with nonlinear systems.

Our key technique is to introduce new variables such that the load functions of all operators can be expressed as linear functions of the actual system input stream rates as well as the newly introduced variables. Our linearization technique is best illustrated with a simple example.

Example 3: Consider the query graph in Figure 13. Assume that the selectivities of all operators except o_1 are constant. Because of this, the load function of o_2 is not a linear function of r_1. So we introduce the output stream rate of o_1 as a new variable r_3. Thereby, the load functions of o_1 to o_4 are all linear with respect to r_1 to r_3.

Assume operator o_5 is a time-window-based join operator that joins tuples whose timestamps are within a give time window w [1]. Let o_5's input stream rates be r_u and r_v, its selectivity (per tuple pair) be s_5, and its processing cost be c_5 CPU cycles per tuple pair. The number of tuple pairs to be processed in unit time is wr_ur_v. The load of o_5 is thus $c_5wr_ur_v$ and the output stream rate of this operator is $s_5wr_ur_v$. As a result, it is easy to see that the load function of o_5 cannot be expressed as a linear function of r_1 to r_3. In addition, the input to o_6 cannot be expressed as a linear function of r_1 to r_3 either. The solution is to introduce the output stream rate of operator o_5 as a new variable r_4. It is easy to see that the load of operator o_6 can be written as a linear function of r_4. Less apparent is the fact that the load of operator o_5 can also be written as $(c_5/s_5)r_4$, which is a linear function of r_4 (assuming c_5 and s_5 are constant). Therefore, the load functions of the entire query graph can be expressed as linear functions of four variables r_1 to r_4. This approach can also be considered as "cutting" a nonlinear query graph into linear pieces (as in Figure 13).

This linearization technique is general; i.e., it can transform any nonlinear load model into a linear load model by introducing additional input variables. Once the system is linear, the analysis and techniques presented earlier apply. However, because the performance of ROD depends on whether the *weights* of each variable can be well balanced across the nodes, we aim to introduce as few additional variables as possible.

6.3 Operator Clustering

So far, we have ignored the CPU cost of communication. We now address this issue by introducing *operator clustering* as a preprocessing step to be applied before ROD. The key idea is to identify costly arcs and ensure that they do not cross the network by placing the end operators on the same machine.

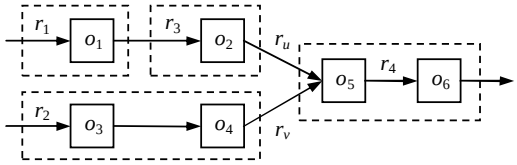


Figure 13: Linear cut of a non-linear query graph.

We studied two greedy clustering approaches. The first approach (i) computes a *clustering ratio* (i.e., the ratio of the per-tuple data transfer overhead of the arc over the minimum data processing overhead of the two end-operators) for each arc; (ii) clusters the end-operators of the arc with the largest clustering ratio; and (iii) repeats the previous step until the clustering ratios of all arcs are less than a given clustering threshold. A potential problem with this method is that it may create operator clusters with very large weights. A second approach is, thus, to choose the two connected operators with the minimum total weight in step (ii) of the approach above. In both cases, we set an upper bound on the maximum weight of the resulting clusters. It is easy to see that varying clustering thresholds and weight upper bounds leads to different clustering plans.

Our experimental analysis of these approaches did not yield a clear winner when considering various query graphs. Our current practical solution is to generate a small number of clustering plans for each of these approaches by systematically varying the threshold values, obtain the resulting operator distribution plans using ROD, and pick the one with the maximum plane distance.

7. PERFORMANCE STUDY

In this section, we study the performance of ROD by comparing it with several alternative schemes using the Borealis distributed stream processing system [2] and a custom-built simulator. We use real network traffic data and an aggregation-heavy traffic monitoring workload, and report results on feasible set size as well as processing latencies.

7.1 Experimental Setup

Unless otherwise stated, we assume the system has 10 homogeneous nodes. In addition to the aggregation-based traffic monitoring queries, we used random query graphs generated as a collection of operator trees rooted at input operators. We randomly generate with equal probability from one to three downstream operators for each node of the tree. Because the maximum achievable feasible set size is determined by how well the weight of each input stream can be balanced, we let each operator tree consists of the same number of operators and vary this number in the experiments. For ease of experimentation, we also implemented a "delay" operator whose per-tuple processing cost and selectivity can be adjusted. The delay times of the operators are uniformly distributed between 0.1 ms to 1 ms. Half of these operators are randomly selected and assigned a selectivity of one. The selectivities of other operators are uniformly distributed from 0.5 to 1. To measure the operator costs and selectivities in the prototype implementation, we randomly distribute the operators and run the system for a sufficiently long time to gather stable statistics.

In Borealis, we compute the feasible set size by randomly generating 100 workload points, all within the ideal feasible set. We compute the ideal feasible set based on operator cost and selectivity statistics collected from trial runs. For each workload point, we run

the system for a sufficiently long period and monitor the CPU utilization of all the nodes. The system is deemed feasible if none of the nodes experience 100% utilization. The ratio of the number of feasible points to the number of runs is the ratio of the achievable feasible set size to the ideal one.

In the simulator, the feasible set sizes of the load distribution plans are computed using Quasi Monte Carlo integration [14]. Due to the computational complexity of computing multiple integrals, most of our experiments are based on query graphs with five input streams (unless otherwise specified). However, the observable trends in experiments with different numbers of input streams suggest that our conclusions are general.

7.2 Algorithms Studied

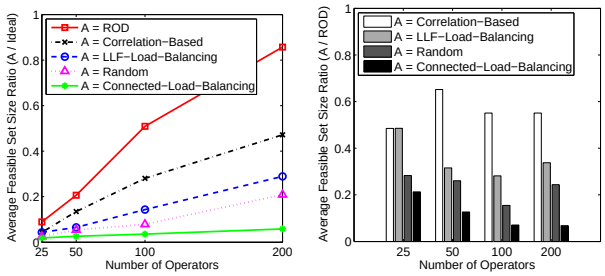
We compared ROD with four alternative load distribution approaches. Three of these algorithms attempt to balance the load while the fourth produces a random placement while maintaining an equal number of operators on each node. Each of the three load balancing techniques tries to balance the load of the nodes according to the average input stream rates. The first one, called Largest-Load First (LLF) Load Balancing, orders the operators by their average load-level and assigns operators in descending order to the currently least loaded node. The second algorithm, called Connected-Load-Balancing, prefers to put connected operators on the same node to minimize data communication overhead. It assigns operators in three steps: (1) Assign the most loaded candidate operator to the currently least loaded node (denoted by N_s). (2) Assign operators that are connected to operators already on N_s to N_s as long as the load of N_s (after assignment) is less than the average load of all operators. (3) Repeat step (1) and (2) until all operators are assigned. The third algorithm, called Correlation-based Load Balancing, assigns operators to nodes such that operators with high load correlation are separated onto different nodes. This algorithm was designed in our previous work [23] for dynamic operator distribution.

7.3 Experimental Results

7.3.1 Resiliency Results

First, we compare the feasible set size achieved by different operator distribution algorithms in Borealis. We repeat each algorithm except ROD ten times. For the Random algorithm, we use different random seeds for each run. For the load balancing algorithms, we use random input stream rates, and for the Correlation-based algorithm, we generate random stream-rate time series. ROD does not need to be repeated because it does not depend on the input stream rates and produces only one operator distribution plan. Figure 14(a) shows the average feasible set size achieved by each algorithm divided by the ideal feasible set size on query graphs with different numbers of operators.

It is obvious that the performance of ROD is significantly better than the average performance of all other algorithms. The Connected algorithm fares the worst because it tries to keep all connected operators on the same node. This is a bad choice because a spike in an input rate cannot be shared (i.e., collectively absorbed) among multiple processors. The Correlation-Based algorithm does fairly well compared to the other load balancing algorithms, because it tends to do the opposite of the Connected algorithm. That is, operators that are downstream from a given input have high load correlation and thus tend to be separated onto different nodes. The Random algorithm and the LLF Load Balancing algorithm lie between the previous two algorithms because, although they do not explicitly try to separate operators from the same input stream, the



(a) (b)

Figure 14: Base resiliency results.

inherent randomness in these algorithms tends to spread operators out to some extent. ROD is superior because it not only separates operators from each input stream, but also tries to avoid placing "heavy" operators from different input streams on the same node, thus avoiding bottlenecks.

As the number of operators increases, ROD approaches the ideal case and most of the other algorithms improve because there is a greater chance that the load of a given input stream will be spread across multiple nodes. On the other hand, even for fewer operators, our method retains roughly the same relative performance improvement (Figure 14(b)).

Notice that the two hundred operators case is not unrealistic. In our experience with the financial services domain, applications often consist of related queries with common sub-expressions, so query graphs tend to get very wide (but not necessarily as deep). For example, a real-time proof-of-concept compliance application we built for 3 compliance rules required 25 operators. A full-blown compliance application might have hundreds of rules, thus requiring very large query graphs. Even in cases where the user-specified query graph is rather small, parallelization techniques (e.g., range-based data partitioning) significantly increase the number of operator instances, thus creating much wider, larger graphs.

We also ran the same experiments in our distributed stream-processing simulator. We observed that the simulator results tracked the results in Borealis very closely, thus allowing us to trust the simulator for experiments in which the total running time in Borealis would be prohibitive.

In the simulator, we compared the feasible set size of ROD with the optimal solution on small query graphs (no more than 20 operators and 2 to 5 input streams) on two nodes. The average feasible set size ratio of ROD to the optimal is 0.95 and the minimum ratio is 0.82. Thus, for cases that are computationally tractable, we can see that ROD's performance is quite close to the optimal.

7.3.2 Varying the Number of Inputs

Our previous results are based on a fixed number of input streams (i.e., dimensions). We now examine the relative performance of different algorithms for different numbers of dimensions using the simulator.

Figure 15 shows the ratio of the feasible set size of the competing approaches to that of ROD, averaged over multiple independent runs. We observe that as additional inputs are used, the relative performance of ROD gets increasingly better. In fact, each additional dimension seems to bring to ROD a constant relative percentage improvement, as implied by the linearity of the tails of the curves. Notice that the case with two inputs exhibits a higher ratio than that estimated by the tail, as the relatively few operators per node in this case significantly limits the possible load distribution choices. As a result, all approaches make more or less the same distribu-

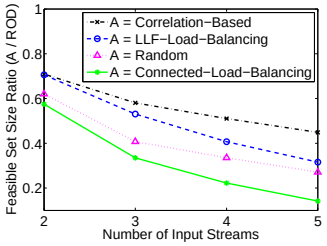
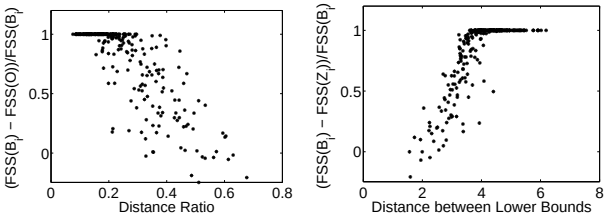


Figure 15: Impact of number of input streams.



(a) (b)

Figure 16: (a) Penalty for not using the lower bound. (b) Penalty for using a wrong lower bound.

tion decisions. For example, when the number of operators equals the number of nodes, all algorithms produce practically equivalent operator distribution plans.

7.3.3 *Using a Known Lower-bound*

As discussed in Section 6, having knowledge of a lower bound on one or more input rates can produce results that are closer to the ideal. We verify this analysis in this next set of experiments in the simulator.

We generate random points B_i in the ideal feasible space anchored at the origin to use as the lower bounds of each experiment. For each B_i, we generate two operator distribution plans, one that uses B_i as the lower bound and one that uses the origin. We then compute the feasible set size for these two plans relative to B_i. Let us call the feasible set size for the former plan FFS(B_i) and the feasible set size for the later FSS(O). We compute the penalty for not knowing the lower bound as $(\text{FFS}(B_i) - \text{FSS}(O))\,/\text{FSS}(B_i)$

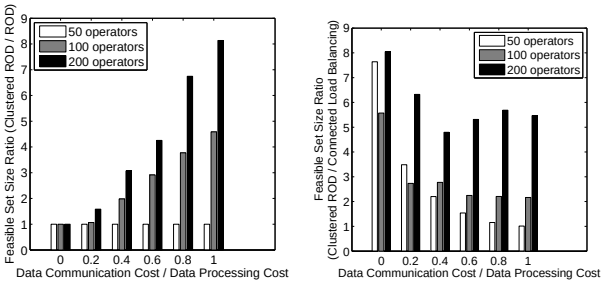
We now run our experiment on a network of 50 operators. We plot the penalty in Figure 16(a) with the x-axis as the ratio of the distance from B_i to the ideal hyperplane to the distance from the origin to the ideal hyperplane. Notice that when this ratio is small (B_i is very close to the ideal hyperplane), the penalty is large because without knowing the lower bound it is likely that we will sacrifice the small actual feasible set in order to satisfy points that will not occur. As B_i approaches the origin (i.e., the ratio gets bigger), the penalty drops off as expected.

The next experiment quantifies the impact of inaccurate knowledge of the lower bound values. In Figure 16(b), we run the same experiment as above except that, instead of using the origin as the assumed lower bound, we use another randomly generated point. As in the above experiment, we compute a penalty for being wrong. In this case, the penalty is computed as $(\text{FFS}(B_i) - \text{FSS}(Z_i))\,/\text{FSS}(B_i)$ where B_i is the real lower bound, as before, and Z_i is the assumed lower bound. The x axis is the distance between B_i and Z_i in the normalized space. As one might expect, when the real and the assumed lower bounds are close to each other, the penalty is low. As the distance increases, the penalty also increases (Figure 16(b)).

The penalty is also dependent on the number of operators in our

Table 3: Average penalty

number of operators	25	50	100	200
origin as the lower bound	0.90	0.79	0.56	0.35
Z_i as the lower bound	0.89	0.83	0.55	0.32



(a) (b)

Figure 17: Performance with operator clustering.

query network. We redo our experiments for different networks with 25 to 200 operators. Looking at Table 3, we see that the average penalty drops as we increase the number of operators. For very large numbers of operators, the penalty will converge to zero since, at that point, all the hyperplanes can be very close to the ideal case given the greater opportunity for load balancing.

7.3.4 *Operator Clustering*

In this section, we address data communication overheads and study the impact of operator clustering. For simplicity, we let each arc have the same per-tuple data communication cost and each operator have the same per tuple data processing cost. We vary the ratio of data communication cost over data processing cost (from 0 to 1) and compare ROD with and without operator clustering. The results shown in Figure 17(a) are consistent with the intuition that operator clustering becomes more important when the relative communication cost increases.

We also compare the performance of clustered ROD with Connected Load Balancing in Figure 17(b). Our first observation is that clustered ROD consistently performs better than Connected Load Balancing regardless of the data communication overhead. Secondly, we observe that clustered ROD can do increasingly better as the number of operators per input stream increases—more operators means more clustering alternatives and more flexibility in balancing the weight of each input stream across machines.

7.3.5 *Latency Results*

While the abstract optimization goal of this paper is to maximize the feasible set size (or minimize the probability of an overload situation), stream processing systems must, in general, produce low latency results. In this section, we evaluate the latency performance of ROD against the alternative approaches. The results are based on the Borealis prototype with five machines for aggregation-based network traffic monitoring queries on real-world network traces.

As input streams, we use an hour's worth of TCP packet traces (obtained from the Internet Traffic Archive [5]). For query graph, we use 16 aggregation operators that compute the number of packets and the average packet size for each second and each minute (using non-overlapping time windows), and for the most recent 10 seconds and most recent one minute (using overlapping sliding windows), grouped by the source IP address or source-destination address pairs. Such multi-resolution aggregation queries are com-

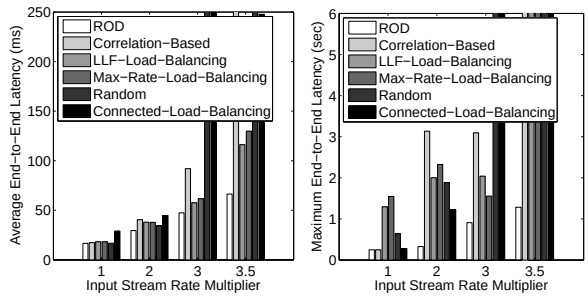


Figure 18: Latency results (prototype based).

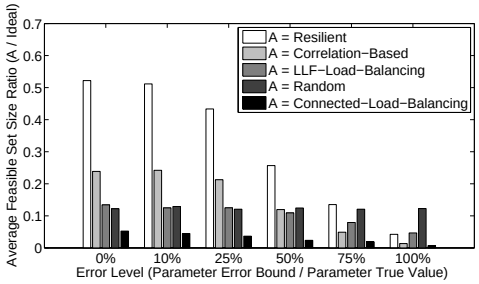


Figure 19: Sensitivity to statistics errors.

monly used for various network monitoring tasks including denial of service (DoS) attack detection.

To give more flexibility to the load distributor and enable higher data parallelism, we partition the input traces into 10 sub-streams based on the IP addresses, with each sub-stream having roughly one tenth of the source IP addresses. We then apply the aggregation operators to each sub-stream and thus end up with 160 "physical" operators. Note that this approach does not yield perfectly uniform parallelism, as the rates of the sub-streams are non-uniform and independent. It is therefore not possible to assign equal numbers of sub-streams along with their corresponding query graphs to different nodes and expect to have load balanced across the nodes (i.e. expect that the ideal feasible set can be achieved by pure data-partitioning based parallelism).

In addition to the algorithms described earlier, we introduce yet another alternative, Max-Rate-Load-Balancing, that operates similar to LLF-Load-Balancing, but differs from it in that the new algorithm balances the maximum load of the nodes using the maximum stream rate (as observed during the statistics collection period).

In order to test the algorithms with different stream rates, we scale the rate of the inputs by a constant. Figure 18 shows the average end-to-end latency and the maximum end-to-end latency results for the algorithms when the input rate multiplier is 1, 2, 3, and 3.5, respectively. These multipliers correspond to 26%, 48%, 69% and 79% average CPU utilization for ROD. Overall, ROD performs better than all others not only because it produces the largest feasible set size (i.e., it is the least likely to be overloaded), but also because it tends to balance the load of the nodes under multiple input rate combinations. When we further increase the input rate multiplier to 4, all approaches except ROD fail due to overload (i.e., the machines run out of memory as input tuples queue up and overflow the system memory). At this point, ROD operates with approximately 91% average CPU utilization.

The results demonstrate that, for a representative workload and data set, ROD (1) sustains longer and is more resilient than the alternatives, and (2) despite its high resiliency, it does not sacrifice latency performance.

7.3.6 Sensitivity to Statistics Errors

In the following experiments, we test, in the simulator, how sensitive ROD is to the accuracy of the cost and selectivity statistics. Suppose that the true value of a statistic is v. We generate a random error factor f uniformly distributed in the interval $[1-e, 1+e]$. The measured value of v is then set as $f \times v$. We call e the *error level*. In each experiment, we generate all measured costs and selectivities according to a fixed e. Figure 19 shows the performance of different algorithms with different error levels on a query graph of 100 operators. The feasible set size of all algorithms, except for Random, decreases when the error level increases. The feasible set size of ROD does not change much when the error level is 10%. ROD's performance remains much better than the others even when the error level is as large as 50%.

8. RELATED WORK

Task allocation algorithms have been widely studied for traditional distributed and parallel computing systems [11, 18]. In principle, these algorithms are categorized into static algorithms and dynamic algorithms. In static task allocation algorithms, the distribution of a task graph is performed only once before the tasks are executed on the processing nodes. Because those tasks can be finished in a relatively short time, these algorithms do not consider time-varying workload as we did.

In this paper, we try to balance the load of each input stream for bursty and unpredictable workloads. Our work is different from the work on multi-dimensional resource scheduling [12]. This work considers each resource (e.g. CPU, memory) as a single dimension, while we balance the load of different input streams that share the same resource (CPU). Moreover, balancing the load of different input streams is only part of our contribution. Our final optimization goal is to maximize the feasible set size, which is substantially different from previous work.

Dynamic task migration received attention for systems with long running tasks, such as large scientific computations or stream data processing. Dynamic graph partitioning is a good example [16, 20]. This problem involves partitioning a connected task graph into uniformly loaded subgraphs, while minimizing the total "weight" (often the data communication cost) of the cutting edges among the subgraphs. If changes in load lead to unbalanced subgraphs, the boundary vertices of the subgraphs are moved to re-balance the overall system load. Our work differs in that we aim to keep the system feasible under unpredictable workloads without operator migration.

Our work is done in the context of distributed stream processing. Early work on stream processing (e.g., Aurora [3] , STREAM [13], and TelegraphCQ [6]) focused on efficiently running queries over continuous data streams on a single machine. The requirement for scalable and highly-available stream processing services led to the research on distributed stream processing systems [7]. Load management in these systems has recently started to receive attention [4, 17, 23].

Shah *et al.* presented a dynamic load distribution approach for a parallel continuous query processing system, called Flux, where multiple shared-nothing servers cooperatively process a single continuous query operator [17]. Flux performs dynamic "intra-operator" load balancing in which the input streams to a single operator are partitioned into sub-streams and the assignment of the sub-streams to servers is determined on the fly. Our work is orthogonal to Flux, as we address the "inter-operator" load distribution problem.

Medusa [4] explores dynamic load management in a federated

environment. Medusa relies on an economical model based on pair-wise contracts to incrementally converge to a balanced configuration. Medusa is an interesting example of load balancing, as it focuses on a decentralized dynamic approach, whereas our work attempts to keep the system feasible *without* relying on dynamic load distribution.

Our previous work [23] presented another dynamic load balancing approach for distributed stream processing. This approach continually tracks load variances and correlations among nodes (captured based on a short recent time window) and dynamically distributes load to minimize the former metric and maximize the latter across all node pairs. Among other differences, our work is different in that it does not rely on history information and is thus more resilient to unpredictable load variations that are not captured in the recent past.

9. CONCLUSIONS

We have demonstrated that significant benefit can be derived by carefully considering the initial operator placement in a distributed stream processing system. We have introduced the notion of a resilient operator placement plan that optimizes the size of the input workload space that will not overload the system. In this way, the system will be able to better withstand short input bursts.

Our model is based on reducing the query processing graph to segments that are linear in the sense that the load functions can be expressed as a set of linear constraints. In this context, we present a resilient load distribution algorithm that places operators based on two heuristics. The first balances the load of each input stream across all nodes, and the second tries to keep the load on each node evenly distributed.

We have shown experimentally that there is much to be gained with this approach. It is possible to increase the size of the allowable input set over standard approaches. We also show that the average latency of our resilient distribution plans is reasonable. Thus, this technique is well-suited to any modern distributed stream processor. Initial operator placement is useful whether or not dynamic operator movement is available. Even if operator movement is supported, this technique can be thought of as a way to minimize its use.

An open issue of resilient operator distribution is how to use extra information, such as upper bounds on input stream rates, variations of input stream rates, or input stream rate distributions, to further optimize the operator distribution plan. Due to the complexity of computing multiple integrals and the large number of possible operator distribution plans, incorporating extra information in the operator distribution algorithm is not trivial. For each kind of new information, new heuristics need to be explored and integrated into the operator distribution algorithm.

Recall that we deal with systems with non-linear operators by transforming their load models into linear ones. We would like to investigate alternatives to this that would not ignore the relationships between the contiguous linear pieces. We believe that in so doing, we would end up with a larger feasible region.

10. REFERENCES

[1] The internet traffic archive. http://ita.ee.lbl.gov/.

[2] 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. In *Proc. of CIDR*, 2005.

[3] D. J. Abadi, D. Carney, U. Çetintemel, 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 Journal*, 2003.

[4] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In *Proc. of the 1st NSDI*, 2004.

[5] L. Bottomley. Dec-pkt, the internet traffic archive. http://ita.ee.lbl.gov/html/contrib/DEC-PKT.html.

[6] S. Chandrasekaran, A. Deshpande, M. Franklin, and J. Hellerstein. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of CIDR*, 2003.

[7] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. of CIDR*, 2003.

[8] E. G. Coffman, M. R. G. Jr., and D. S. Johnson. Approximation algorithms for binpacking - an updated survey. *Algorithm Design for Computer Systems Design*, pages 49–106, 1984.

[9] M. Crovella and A. Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 1997.

[10] G. Dantzig. *Linear programming and extensions*. Princeton University Press, Princeton, 1963.

[11] R. Diekmann, B. Monien, , and R. Preis. Load balancing strategies for distributed memory machines. *Multi-Scale Phenomena and Their Simulation*, pages 255–266, 1997.

[12] M. N. Garofalakis and Y. E. Ioannidis. Multi-dimensional resource scheduling for parallel queries. In *Proc. of the 1996 ACM SIGMOD*, pages 365–376, 1996.

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

[14] H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*. Society for Industrial and Applied Mathematics, Philadelphia, 1992.

[15] A. L. Peressini, F. E. Sullivan, and J. J. Uhl. *The Mathematics of Nonlinear Programming*. 1988.

[16] K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high performance scientific simulations. *CRPC Parallel Computing Handbook*, 2000.

[17] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proc. of the 19th ICDE*, 2003.

[18] B. A. Shirazi, A. R. Hurson, and K. M. Kavi. Scheduling and load balancing in parallel and distributed systems. *IEEE Comp. Sci. Press*, 1995.

[19] I. H. Sloan and H. Wozniakowski. Multiple integrals in many dimensions. In *Advances in Computational Mathematics: Proc. of the Guangzhou International Symposium*, 1997.

[20] C. Walshaw, M. Cross, , and M. G. Everett. Dynamic load balancing for parallel adaptive unstructured meshes. *Parallel Processing for Scientific Computing*, Oct. 1997.

[21] E. W. Weisstein. Change of variables theorem. http://mathworld.wolfram.com/Hypersphere.html.

[22] E. W. Weisstein. Hypersphere. http://mathworld.wolfram.com/Hypersphere.html.

[23] Y. Xing, , S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *Proc. of the 19th ICDE*, Mar. 2005.