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

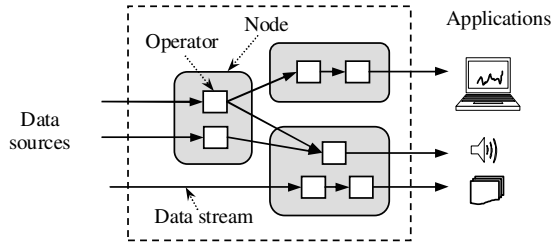

Figure 1: Distributed Stream Processing.

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

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

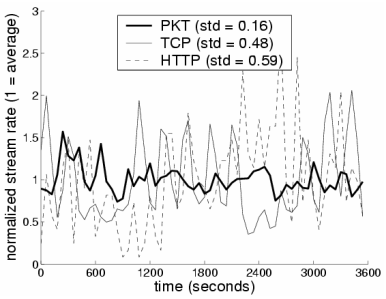

Figure 2: Stream rates exhibit significant variation over time.

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

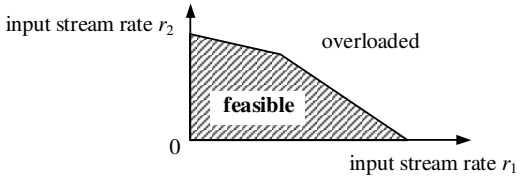

Figure 3: Feasible set on input stream rate space.

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

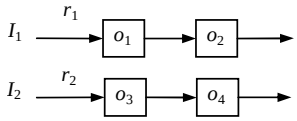

Figure 4: Example query graph.

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

$$L^n = A L^o,$$

$$\sum_{i=1}^{n} l^n_{ik} = \sum_{j=1}^{m} l^o_{jk} \ , \quad k = 1, \cdots d.$$

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}	$\left(l^n_{ik}/l_k\right) / \left(C_i/C_T\right)$, 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)	$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$	$\begin{pmatrix} 14 & 7 \\ 6 & 9 \end{pmatrix}$

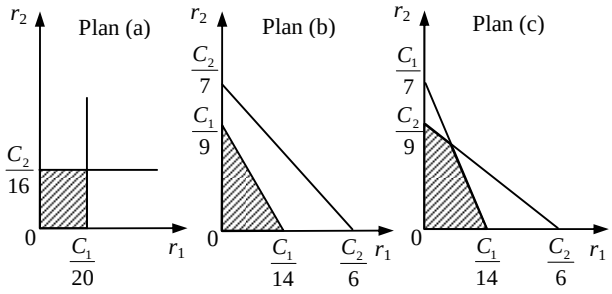

Figure 5: Feasible sets for various distribution plans.

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

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

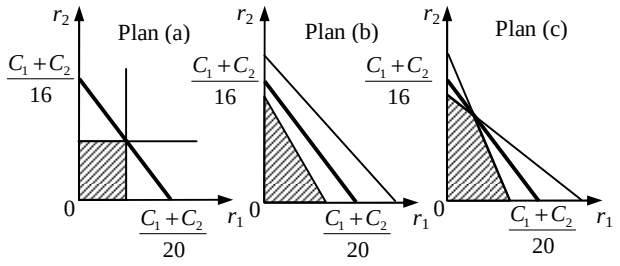

Figure 6: Ideal hyperplanes and feasible sets.

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