Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing

Nesime Tatbul
ETH Zurich
tatbul@inf.ethz.ch

Uğur Çetintemel
Brown University
ugur@cs.brown.edu

Stan Zdonik
Brown University
sbz@cs.brown.edu

ABSTRACT

In distributed stream processing environments, large numbers of continuous queries are distributed onto multiple servers. When one or more of these servers become overloaded due to bursty data arrival, excessive load needs to be shed in order to preserve low latency for the query results. Because of the load dependencies among the servers, load shedding decisions on these servers must be well-coordinated to achieve end-to-end control on the output quality. In this paper, we model the distributed load shedding problem as a linear optimization problem, for which we propose two alternative solution approaches: a solver-based centralized approach, and a distributed approach based on metadata aggregation and propagation, whose centralized implementation is also available. Both of our solutions are based on generating a series of load shedding plans in advance, to be used under certain input load conditions. We have implemented our techniques as part of the Borealis distributed stream processing system. We present experimental results from our prototype implementation showing the performance of these techniques under different input and query workloads.

1. INTRODUCTION

Distributed stream processing systems (e.g., [3, 5, 16, 19]) have recently gained importance because distribution is the principle way that we can scale our systems to cope with very high stream rates. Distribution is a crucial issue in applications such as Internet-scale dissemination, in which content from many sources is aggregated and distributed to an audience of many millions of listeners. Also, many streaming applications are naturally distributed, as in the example of distributed sensor networks, where the processing elements are the sensors themselves.

In distributed stream processing systems, large numbers of continuous queries are distributed onto multiple servers. These queries are essentially dataflow diagrams in the form of a collection of operator chains that receive and process continuous streams of data from external push-based data sources. Real-time monitoring applications are especially well-suited to this kind of systems. In this domain, providing low-latency, high-throughput answers to queries is highly important.

Data streams can arrive in bursts which can have a negative effect on result quality (e.g., throughput, latency). Provisioning the system for worst-case load is in general not economically sensible. On the other hand, bursts in data rates may create bottlenecks at some points along the server chain. Bottlenecks may arise due to excessive demand on processing power at the servers, or bandwidth shortage at the shared physical network that connects these servers. Bottlenecks slow down processing and network transmission, and cause delayed outputs.

Load management has been an important challenge for large-scale dynamic systems in which input rates can unexpectedly increase to drive the system into overload. At the same time, some measure of quality of service must be maintained. Common load management techniques include adaptive load distribution, admission control, and load shedding. The choice of a specific technique depends on the characteristics of the workload, resource allocation policies, and application requirements.

This paper studies load shedding. Load shedding aims at dropping tuples at certain points along the server chain to reduce load. Unlike TCP congestion control, there are no retransmissions and dropped tuples are lost forever. This will have a negative effect on the quality of the results delivered at the query outputs. The main goal is to minimize the quality degradation.

Load shedding techniques have been proposed for data stream processing systems for the single-server case (e.g., [7, 17, 22]). In distributed stream processing systems, however, each server node acts like a workload generator for its downstream neighbors. Therefore, resource management decisions at any server node will affect the characteristics of the workload received by its children. Because of this load dependency between nodes, a given node must figure out the effect of its load shedding actions on the load levels of its descendant nodes. Load shedding actions at all nodes along a given server chain will collectively determine the quality degradation at the outputs. This makes the problem more challenging than its centralized counterpart.

1.1 Motivating Example

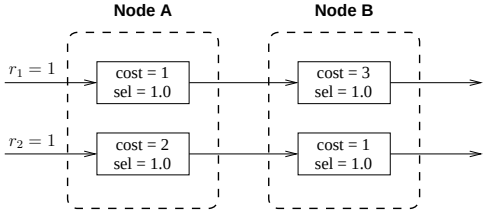


Figure 1: Motivating example

Consider a simple query network with two queries that are distributed onto two processing nodes, A and B (Figure 1). Each small box represents a subquery with a certain cost and selectivity. Cost

VLDB '07, September 23-28, 2007, Vienna, Austria.

Plan	Reduced rates at A	A.load	A.throughput	B.load	B.throughput	Result
0	1, 1	3	1/3, 1/3	4/3	1/4, 1/4	originally, both nodes are overloaded
1	1/3, 1/3	1	1/3, 1/3	4/3	1/4, 1/4	B is still overloaded
2	1, 0	1	1, 0	3	1/3, 0	optimal plan for A, but increases B.load
3	0, 1/2	1	0, 1/2	1/2	0, 1/2	both nodes ok, but not optimal
4	1/5, 2/5	1	1/5, 2/5	1	1/5, 2/5	optimal

Table 1: Alternate load shedding plans for node A of Figure 1

reflects the average CPU time that it takes for one tuple to be processed by the subquery, and selectivity represents the average ratio of the number of output tuples to the number of input tuples. Both inputs arrive at the rate of 1 tuple per second. Potentially each node can reduce load at its inputs by dropping tuples to avoid overload. Let's consider node A. Table 1 shows various ways that A can reduce its input rates and the consequences of this in terms of the load at both A and B, as well as the throughput observed at the query outputs (Note that we are assuming a fair scheduler that allocates CPU cycles among the subqueries in a round-robin fashion). In all of these plans, A can reduce its load to the capacity limit. However, the effect of each plan on B can be very different. In plan 1, B stays at the same overload level. In plan 2, B's load increases to more than twice its original load. In plan 3, B's overload problem is also resolved, but throughput is low. There is a better plan which removes overload from both A and B, while delivering the highest total throughput (plan 4). However, node A can only implement this plan if it knows about the load constraints of B. From A's point of view, the best local plan is plan 2. This simple example clearly shows that nodes must coordinate in their load shedding decisions to be able to deliver high-quality query results.

1.2 Contributions and Outline

A load shedder inserts drop operators on selected arcs in order to reduce the load to a manageable level. A drop operator simply eliminates a given fraction of its inputs probabilistically. We call a set of drop operators with given drop levels at specific arcs a *load shedding plan*. In practice, a load shedder cannot spend large amount of time determining the best plan at runtime, when the system is already under duress. Instead, in this work, we run an off-line algorithm to build a set of plans in advance that can be quickly invoked for different combinations of input load.

For the distributed case, the simplest way to run the off-line algorithm is to have each node send its requirements to a central site at which the coordinated load shedding plans are built. As we shall see, this allows us to formalize distributed load shedding as a linear optimization problem which can be solved with a standard solver.

Unfortunately, the centralized approach does not scale as the number of nodes grows large. Moreover, since the solver can take a long time to run, it is not very useful as a tool for replanning when the environment is highly dynamic. A dynamic environment is one in which the selectivities, processing costs, and network topology are likely to change often. In these cases, the previously computed load shedding plans will likely not be desirable, therefore, a new set of plans must be constructed. For these large and potentially dynamic environments, we describe a distributed algorithm that does not require the high-level of communication that the centralized approach demands. We also show that this distributed algorithm can incrementally compute changes to the previous plan in response to local changes in the environment, thereby making it more responsive than a centralized version.

This paper is organized as follows: In Section 2, we first briefly describe our system model and the underlying assumptions that we make. Then, in Section 3, we present a precise formulation of the distributed load shedding problem. In Section 4, we discuss the architectural aspects of our solution to this problem. Our solver-based centralized approach is detailed in Section 5, while Section 6 provides the details for our distributed solution alternative. In section 8, we present experimental results that show the efficiency of our techniques. We discuss related work in Section 9, and finally conclude with a discussion of future directions in Section 10.

2. MODELS AND ASSUMPTIONS

We study the distributed load shedding problem in the context of our Borealis distributed stream processing system [3]. Borealis accepts a collection of continuous queries, represents them as one large network of query operators, and distributes the processing of these queries across multiple server nodes. Each node runs an instance of the Aurora query processing engine [4] that is responsible for executing its share of the global query network [1].

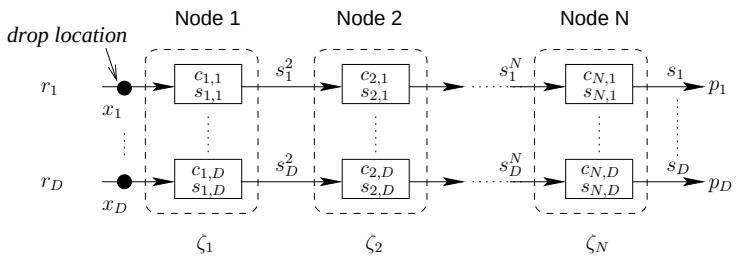
Data streams are modeled as append-only sequences of relational tuples. These data streams are run through the queries which are composed of our well-defined set of operators, including Filter, Map, Aggregate, Join, and Union [4]. Additionally, we have several types of load reducing drop operators that we use for load shedding. In this paper, we focus on drops with probabilistic behavior, namely, Random Drop and Window Drop. Random Drop discards individual tuples based on a drop probability, whereas Window Drop does so in units of whole windows. Window Drop is specifically used for windowed aggregation queries to maintain subset results [23].

In line with our previous work, we adopt a subset-based approximation model [22, 23]. In other words, the output resulting from a load shedding plan only includes tuples from the original query answer. Additionally, we assume that the quality metric (a.k.a., quality score) to maximize is the total weighted query throughput. Throughput-based metrics for scheduling and load shedding have also been commonly used by previous work (e.g., [5, 6]). We allow different weights to be assigned to different queries to enable prioritization and to deal with the potential fairness problem.

For the purposes of this paper, we treat the query operators as black boxes with certain cost and selectivity statistics, which are obtained by observing the running system over time. However, we would like to note here that additional techniques are required to handle complex operators such as Aggregate and Join. We handle aggregation queries using our Window Drop approach [23], which directly complements this work by making sure that any drops placed upstream from aggregates are Window Drops rather than Random Drops. For join queries, we adopt the same cost model earlier proposed by Ayad and Naughton [6]. This work also uses Random Drops for load shedding on join queries with count-based windows, and tries to maximize the query throughput, but for the single server case.

Bottlenecks in a distributed setting may arise both due to the lack of required processing power and also due to bandwidth limitations. In this paper, we limit our scope to the CPU problem. Finally, in this paper, we mainly focus on tree-based server topologies, while

[1]To clarify our terminology, a query network is essentially a horizontal query execution plan on which tuples flow from parents to children.



r_j	rate on arc j
x_j	drop selectivity on arc j
$c_{i,j}$	cost of processing tuples at node i coming from arc j
$s_{i,j}$	selectivity of processing tuples at node i coming from arc j
s_j^i	partial selectivity of processing tuples coming from arc j down to node i
s_j	total selectivity of processing tuples coming from arc j down to the outputs
p_j	weight of output j
ζ_i	fraction of the dedicated CPU capacity at node i

Figure 2: Linear query diagram and notation

the query network itself can have both operator splits and merges.

3. THE DISTRIBUTED LOAD SHEDDING PROBLEM

3.1 Basic Formulation

We define the distributed load shedding problem as a linear optimization problem as follows. Consider a query diagram as shown in Figure 2, that spans N nodes, each with a fixed dedicated CPU capacity ζ_i, $0 < i \leq N$. Assume that we designate D arcs on this diagram as drop locations where drop operators can be inserted. Note that in a linear query diagram without operator splits, drop locations are the input arcs [22]. For a drop location d_j on arc j, $0 < j \leq D$, let $c_{i,j}$ represent the total CPU time required at node i, to process one tuple that is coming from arc j, and similarly, let $s_{i,j}$ represent the overall selectivity of the processing that is performed at node i on tuples that are coming from arc j. Assume that r_j represents the data rate on arc j, and s_j represents the overall selectivity of the query from arc j all the way down to the query outputs (i.e., $s_j = \prod_{i=1}^{N} s_{i,j}$), where each output has a throughput weight of p_j. Lastly, we denote the partial selectivity from arc j down to the inputs of node i by s_j^i (i.e., for $1 < n \leq N, s_j^n = \prod_{i=1}^{n-1} s_{i,j}$, and for $n = 1, s_j^1 = 1.0$).

Our goal is to find x_j, i.e., the fraction of tuples to be kept at drop location d_j (or drop selectivity at d_j), such that for all nodes $i, 0 < i \leq N$:

$$\sum_{j=1}^{D} r_j \times x_j \times s_j^i \times c_{i,j} \leq \zeta_i \quad (1)$$

$$0 \leq x_j \leq 1 \quad (2)$$

$$\sum_{j=1}^{D} r_j \times x_j \times s_j \times p_j \quad \text{is maximized.} \quad (3)$$

This optimization problem can be stated as a linear program (LP) as follows. We have a set of N linear constraints on processing load of the nodes, which we call *load constraints*, as given by (1). We have a set of D variables x_i on *drop selectivities*, which can range in $0 \leq x_i \leq 1$, as given by (2). Our goal is to find assignments to x_i to maximize a linear objective function that represents the *total weighted throughput* as given by (3), subject to the set of constraints (i.e., (1) and (2)).

Next, we will extend this basic formulation to query diagrams with operator splits and operator merges. We will show the formulation for these cases on representative examples. Generalization from these examples to any given query network topology is relatively straightforward.

3.2 Operator Splits

We have operator splits in a query network when output from an operator fans out to multiple downstream operators which further lead to separate query outputs. Note that if split branches merge downstream in the diagram, we do not consider this as a split case. Split is an interesting case because shedding load upstream or downstream from a split may result in different quality degradation at the outputs due to sharing. Therefore, all output arcs of a split constitute potential drop locations [22].

We illustrate the problem formulation for operator splits on a single-node example with two levels of splits shown in Figure 3. Let x_i denote the drop selectivity on a particular drop location, and c_i and s_i denote the processing cost and selectivity of a given operator, respectively. The variables shown with capital letters denote the total drop selectivity at various points in the query network. We can formulate the optimization problem for our example as follows:

$$r(x_1c_1 + As_1c_2 + Bs_1s_2c_3 + Cs_1s_2c_4 + Ds_1c_5 + Es_1c_6) \leq \zeta \quad (4)$$

$$0 \leq x_1 \leq 1 \quad (5)$$

$$0 \leq A, D, E \leq x_1 \quad (6)$$

$$0 \leq B, C \leq A \quad (7)$$

$$\{r(Bs_1s_2s_3p_1 + Cs_1s_2s_4p_2 + Ds_1s_5p_3 + Es_1s_6p_4)\} \text{ is maximized.} \quad (8)$$

We create a variable for each path prefix (e.g., x_1, $A = x_1x_2$, and $B = x_1x_2x_3$), which we call *prefix variables*. We express our load constraints in terms of the prefix variables, as in (4). We define constraints on each prefix variable of length k such that its value is constrained between 0 and the values of its matching prefix variables of length $k-1$ (e.g., $0 \leq x_1x_2x_3 \leq x_1x_2$). We express our objective function in terms of the longest prefix variables on each path (e.g., $x_1x_2x_3$, $x_1x_2x_4$, x_1x_5, and x_1x_6). Then we solve our problem for the prefix variables. Finally, we plug in the values of the prefix variables to obtain values of the original variables. In our example, we would solve for the prefix variables $\{x_1, A, B, C, D, E\}$ to obtain the original variables $\{x_1, x_2, x_3, x_4, x_5, x_6\}$ as follows: $x_1 = x_1, x_2 = \frac{A}{x_1}, x_3 = \frac{B}{x_1x_2}, x_4 = \frac{C}{x_1x_2}, x_5 = \frac{D}{x_1}, x_5 = \frac{E}{x_1}$.

3.3 Operator Merges

Two streams merge on a query diagram via binary operators. We have two binary operators: Union and Join. Although these two operators have very different semantics, they require a common

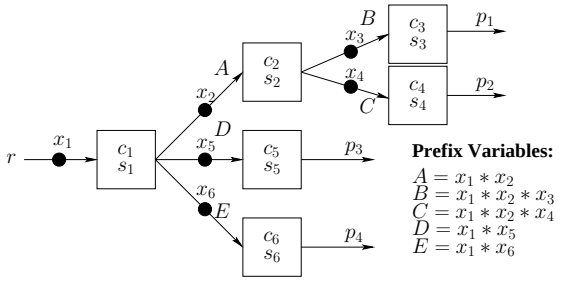


Figure 3: Two levels of operator splits

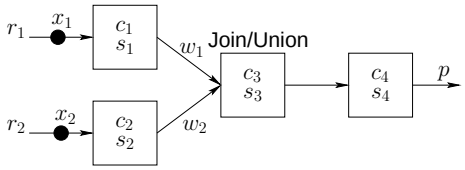


Figure 4: Merging two streams via Join or Union

formulation capturing the fact that the output rate of a binary operator is the sum of contribution from both input branches, and that we may want to shed different amounts from each input branch depending on its relative contribution to the output as well as its relative contribution to the processing load.

Consider the example query segment in Figure 4. Let c_i denote the operator costs, s_i denote the operator selectivities, and w_i denote the average window sizes (in tuple counts) for the inputs. For Union, $w_1 = w_2 = 1$ and $s_3 = 1.0$. For Join, we follow the same cost model as in related work [6]. In this model, join selectivity is the percentage of tuples satisfying the join predicate relative to a cartesian product, and it is symmetric relative to the two inputs. Thus, in Figure 4, each tuple from the top branch of the Join joins with $w_2 * s_3$ tuples and produces $r_1 * x_1 * s_1 * w_2 * s_3$ output tuples per time unit (and similar for the bottom branch). Furthermore, the cost of join for each arriving tuple on the top branch includes the cost of inserting this tuple into window w_1, invalidating any expiring tuples from w_1, and probing w_2 for matching tuples. We use an average cost c_3 to represent these steps for both sides of the join. Based on this cost model, we formulate the optimization problem for the merge scenario shown in Figure 4 as follows:

$$r_1 x_1 (c_1 + s_1 c_3 + s_1 w_2 s_3 c_4) + r_2 x_2 (c_2 + s_2 c_3 + s_2 w_1 s_3 c_4) \leq \zeta \quad (9)$$

$$0 \leq x_1, x_2 \leq 1 \quad (10)$$

$$\{r_1 x_1 (s_1 w_2 s_3 s_4) p + r_2 x_2 (s_2 w_1 s_3 s_4) p\} \quad \text{is maximized.} \quad (11)$$

4. ARCHITECTURAL OVERVIEW

In the previous section, we showed how to formulate one instance of the distributed load shedding problem for a specific observation of the input rates. When we solve such a problem instance, we obtain a load shedding plan. This plan essentially shows where drop operators should be inserted into the query network, and what the drop selectivity should be for each of them. We will describe how we generate a load shedding plan in Sections 5 and 6. In this section, we discuss the architectural aspects of our solution.

During the course of system execution, input rates and hence the load levels on the servers will vary. Therefore, there is a need to continuously monitor the load in the system and react to it using the appropriate load shedding plan. We identified four fundamental phases in the distributed load shedding process:

1. Advance Planning: In this phase, the system prepares itself for potential overload conditions based on available metadata about the system. The idea is to do as much of the work in advance as possible so that the system can react to overload fast and in a light-weight manner. More specifically, in this phase, we generate a series of load shedding plans together with an indication of the conditions under which each of these plans should be used.

2. Load Monitoring: As the system runs, we continuously watch the system load by measuring the input rates and estimating the load level on each server accordingly.

3. Plan Selection: If an important change in system load is detected during the monitoring phase, then we decide what action to take. This is achieved by selecting the right load shedding plan from the many computed during Advance Planning.

Phases	Centralized	Distributed
Advance Planning	coordinator	all
Load Monitoring	coordinator	all
Plan Selection	coordinator	all
Plan Implementation	all	all

Table 2: Four phases of distributed load shedding

4. Plan Implementation: In this final phase, the selected plan is put into effect by inserting drops into the query network.

In a distributed stream processing environment, the Plan Implementation phase will always be performed at multiple servers in a distributed fashion. The first three phases however, can be performed in various ways. In this work, we study two general approaches, based on an architectural distinction regarding where these three phases should be performed:

Centralized Approach. In the centralized approach, Advance Planning, Load Monitoring, and Plan Selection are all performed at one central server. One of the servers in the system is designated as the "coordinator node". It contacts all the other participant nodes in order to collect their local system catalogs and statistics. By doing so, it obtains the global query network topology and the global statistics about various run-time elements in the system (e.g., operator cost and selectivity). Based on the collected global metadata, the coordinator generates a series of load shedding plans for other servers to apply under certain overload conditions. These plans are then uploaded onto the associated servers together with their unique plan id's. Once the plans are precomputed and uploaded onto the nodes, the coordinator starts monitoring the input load. If an overload situation is detected, the coordinator selects the best plan to apply and sends the corresponding plan id to the other servers in order to trigger the distributed implementation of the selected plan.

Distributed Approach. In the distributed approach, all four phases of distributed load shedding are performed at all of the participating nodes in a coordinated fashion. There is no single point of control. Instead, the collective actions of all the servers result in a globally effective load shedding plan. In this paper, we propose a distributed approach in which the needed coordination is achieved through metadata aggregation and propagation between neighboring nodes. More specifically, each node maintains a Feasible Input Table (FIT) as its metadata. This table shows what makes a feasible input load for a node and its server subtree. Using its FIT, a node can shed load for itself and for its descendant nodes.

Table 2 summarizes the four phases of the distributed load shedding process and where each phase takes place for the two general classes of approaches. In this paper, we focus on two specific approaches that fall under these two classes: (i) a solver-based approach, and (ii) a FIT-based approach. The solver-based approach is fundamentally a centralized approach that requires a coordinator, while the FIT-based approach is inherently designed as a distributed approach, but its centralized implementation is also available. Next, we present these two alternative approaches in detail.

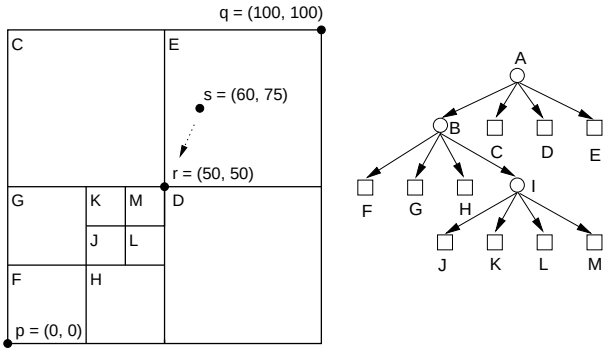
5. ADVANCE PLANNING WITH A SOLVER

As shown in Section 3, the distributed load shedding problem can be formulated as a linear optimization problem. Our solver-based advance planning technique is based on constructing this formulation at a central coordinator and solving it using an off-the-shelf LP solver tool [1].

Given a global query network with statistics on its operator costs and selectivities, the coordinator node first derives the necessary metadata described in Section 3 (including drop locations, path prefixes, partial and total path selectivities, load and rate factors).

Given this metadata, the coordinator can formulate a standard linear program for a specific observation of the input rates as illustrated in Section 3. For each such LP, the coordinator calls the simplex method of the GNU Linear Programming Kit (GLPK) [1]. The solution produced by GLPK consists of value assignments to all the prefix variables and the value of the objective function. From the prefix variables, we can obtain value assignments to all original variables, each representing the drop selectivity on a particular drop location on the global query network.

The final step is to prepare the local load shedding plans. We go through the list of all drop locations. Each such location resides on a specific server node. For each drop location d on node i, we create a drop operator with its drop rate determined by the drop selectivity assignment from the LP solution, and add that operator to the load shedding plan of node i. As a result, we obtain one load shedding plan for each node.



(a) Space division for Solver (b) Space index for Solver

Figure 5: Region-Quadtree-based space division and index

5.1 Region-Quadtree-based Division and Indexing of the Input Rate Space

Given an infeasible (i.e. overloaded) point in the multi-dimensional input rate space, we can generate an optimal plan using the LP solver. However, as part of the Advance Planning phase, we need to generate not only one, but a series of load shedding plans to be able to handle any potential overload condition. In order words, we must map each infeasible point to a load shedding plan that will render that point feasible for all the servers. For a large query network, this space can be very large. Thus, it is not practical to exhaustively consider each possible infeasible point; we certainly do not want to call the solver for too many times. Instead, we carefully pick a subset of infeasible points for which we call the solver to generate optimal plans. Then for the rest of the infeasible points, we try to reuse the generated plans with small modifications. Note that for such points, we certainly can not guarantee optimal plans. Instead, we create plans whose output scores do not deviate from the score of an optimal plan by more than a specific percent error ϵ. For example, given $\epsilon = 10\%$, it would be acceptable to reuse an existing load shedding plan with a score ≥ 45 for an infeasible point p whose actual optimal score would be 50 if we instead called the solver for p.

Our solution is based on dividing the multi-dimensional input rate space into a small number of subspaces such that all infeasible points in a given subspace can be handled using a *similar* load shedding plan. To divide our space into subspaces, we exploit an interesting property of the throughput metric. *The throughput score of an infeasible point q is always greater than or equal to the throughput score of another infeasible point r, when q is larger than or equal to r along all of the dimensions.* This is because for q, the LP solver gets to choose from a larger range of rate values and therefore has more degrees of freedom to find a solution with higher objective value. As a result, given a percent error threshold ϵ, and any infeasible point s such that $r < s < q$: if $(q.score - r.score) \leq \frac{\epsilon}{100} * q.score$, then s can reuse the plan for r with a minor modification. The minor modification simply involves scaling point s to match point r along all dimensions, and is much cheaper to perform than calling the solver for s.

In order to tackle the problem in a systematic way, we use a region-quadtree-based approach to subdivide the space [18]. This approach also gives us the opportunity to build a quadtree-based index on top of our final subspaces which will make the Plan Selection phase much more efficient.

We will now illustrate our approach on the example shown in Figure 5. We assume that the maximum rate along each input dimension is given so that we know the bounds of the space that we are dealing with (e.g., $(100, 100)$ in Figure 5(a)). We start by generating optimal load shedding plans for the two extreme points of our input rate space by calling the solver for each and comparing their scores. Thus, we compare the score of the bottom-most point of this space (e.g., p in Figure 5(a)) with the score of its top-most point (e.g., q in Figure 5(a)). If the percent difference is above the given ϵ value, then we must further divide each dimension of this space into 2 (e.g., giving us 4 subspaces B, C, D, E in Figure 5(a)). Then we repeat the same procedure for each of these 4 subspaces. When we find that the score difference between two extreme points of a subspace is below the ϵ threshold, then we stop dividing that subspace any further. All infeasible points in a given rectangle must reuse the load shedding plan that corresponds to the bottom-most point of that rectangle. For example, in Figure 5(a), assume that the score of point r is within ϵ distance from the score of point q. Then, all infeasible points in the subspace between these two extreme points (i.e., the subspace E) can safely reuse the load shedding plan generated for point r. Assume s is such a point. In order for s to use the plan at r, s must additionally be scaled to match r. This is accomplished by an additional reduction on the input rates. In our example, we need to reduce $s = (60, 75)$ to $r = (50, 50)$ by adding drops of $(\frac{50}{60}, \frac{50}{75})$. With these additional drops, s can now safely use the plan for r without violating ϵ.

Note that, during the space division process, as we get closer to the origin, we may come across some feasible points. If we ever find that the top-most point of a subspace is already a feasible point, it means that all points in that subspace must also be feasible. Therefore, there is no need to generate any load shedding plans for that subspace. Hence, we can stop dividing that subspace any further (e.g., subspace F).

At each iteration of the space division process, we produce a number of new subspaces. To note an important implementation detail, we place these subspaces into a priority queue based on their percent error. Then at each step, we pick the subspace at the top of the queue with the highest error value to divide next. This allows us to stop the space division any time a given error threshold is met.

At the end of the space division process, we obtain a set of disjoint subspaces, each of which is mapped to a load shedding plan at a certain infeasible point in the space. During the Plan Selection phase, we will have to search through these subspaces in order to locate the one that contains a particular infeasible point. In order to make this search process more efficient, we further organize our subspaces into an index. The subspaces can be very conveniently placed into a quadtree-based index during the space division process described above. Figure 5(b) shows the index that corresponds to the space division of Figure 5(a).

5.2 Exploiting Workload Information

We have so far assumed that all of the input rate values in the multi-dimensional space have an even chance of occurrence. In this case, we must guarantee the same ϵ threshold for all of the input rate subspaces. However, if the input rate values are expected to follow an uneven distribution and if this distribution is known in advance, then we could exploit this information to make the Advance Planning phase much more efficient. More specifically, given an input rate subspace with probability p and percent error of ϵ, the expected error for this subspace would be $p * \epsilon$. We must then subdivision the input rate space until the sum of expected errors over all disjoint subspaces meets the ϵ threshold. Thus, instead of strictly satisfying the ϵ threshold for all subspaces, we ensure that on the average the expected maximum error will be below some threshold. Again in this case, we store the subspaces to be divided in a priority queue, but this time we rank them based on their expected errors.

6. ADVANCE PLANNING WITH FIT

Our distributed approach to Advance Planning is based on metadata aggregation and propagation from leaf servers towards the root/input servers. Each leaf server first generates a Feasible Input Table (FIT) which shows the input rate combinations that are feasible (i.e., not causing overload) for that node. This table is then propagated to the parent server. When a parent server receives FITs from its child servers, it maps them from its outputs to its own inputs, merges the mapped tables into a single table, removes the table entries that may be infeasible for itself, and finally propagates the resulting FIT to its own parents. This process continues until the input servers are reached. Using its FIT, a node can then shed load for itself and on behalf of its descendant nodes without any need for further communication.

In this section, we first describe the structure of FIT, followed by a detailed description of how it is generated at the leaf servers, merged and propagated through the non-leaf servers, and finally used for load shedding.

6.1 Feasible Input Table (FIT)

Given a server node with m input streams, the FIT for this node is a table with $m + 2$ columns. The first m columns represent the possible rate combinations for the m inputs; the $(m + 1)$th column shows the "complementary local load shedding plans" that would be needed when a certain input rate combination is observed; and the last column represents the corresponding output quality score (i.e., total weighted throughput). For example, $(0.3, 0.1, null, 0.4)$ is a FIT entry for node B of Figure 1. In this case, since the query network is simple, no complementary local plans are necessary.

Complementary local plans may be needed when the local query network has splits. In this case, each branch of the split may have a different total cost and selectivity. Due to this difference, dropping from some branches may be more desirable than directly dropping from the input. However, this is completely a local issue and need not be exposed to the parent. Instead, we allow a node to create FIT entries for input rate points which are in fact not feasible, and support these input points with complementary local load shedding plans that drop tuples at split branches to provide actual feasibility. We refer such FIT entries as "feasible with a plan". We will present the details of local plan generation in the next section.

6.2 FIT Generation at Leaves

In this section, we describe FIT generation at a leaf server. We first describe how FIT points are chosen, regardless of the query network topology. Then we describe how complementary local plans are generated when there are splits in the query network.

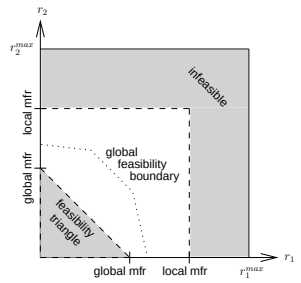


Figure 6: Choosing FIT points (mfr = maximum feasible rate)

6.2.1 Choosing FIT Points

Unlike the solver-based approach which focuses on generating load shedding plans for infeasible points, the FIT-based approach tries to identify the feasible points such that infeasible points can be mapped to one of these feasible points by simply scaling down their rate values (i.e., by inserting drops at the corresponding input arcs). Like in the solver-based case, however, generating and considering all possible input rate combinations would be both inefficient and unnecessary. Therefore, we apply a few tactics in order to choose a reduced number of FIT points, while still providing that we meet a certain error threshold on quality so that we stay close to an optimal load shedding plan.

Exploiting Error Tolerance. In order to meet the ϵ error threshold, we select sample points along each input dimension that are a certain *spread* value apart from each other. For a given dimension, we first compute the maximum feasible rate. For example, the top input of node B of Figure 1 can process a maximum rate of 1/3. Next we generate sample points between 0 and the maximum feasible rate, determining the spread at each step based on the previous sample value. In our example, the next sample point after 1/3 would be $1/3 * (1 - \frac{\epsilon}{100})$. This ensures that the maximum percent error on a given input dimension is at most ϵ. Since the total weighted throughput is a linear summation of all the input rates, the total error would also have a maximum percentage of ϵ.

Excluding Redundant Points. As discussed in Section 5.1, given two points p and q in the multi-dimensional input rate space, if $p \geq q$ along all of the dimensions, then $p.score \geq q.score$. Assume further that p and q are feasible points. Given an infeasible point r where $r > p$ and $r > q$, if we need to scale r down to one of these two feasible points, we must always prefer p over q since it has a larger score. This means that we only need to store in FIT, the feasible points which are on the outer boundary of the global feasibility space. This way, the number of FIT points can be reduced. Unfortunately, at the leaf level, it is not possible to know the exact global feasibility boundary without knowing about the feasibility constraints of the upstream nodes. On the other hand, we can easily identify a region below the global feasibility boundary, whose points can safely be excluded from the leaf FIT. We call this region the "feasibility triangle". This triangle is computed by making a bottom-up pass from our leaf node towards the root, at each node keeping track of the smallest maximum feasible rate along each input dimension. To clarify, in Figure 6, we illustrate a 2-dimensional input rate space. The bottom triangle is the feasibility triangle whereas the top L-shaped region is the infeasible region. The global feasibility boundary (shown with dotted lines) is guaranteed to lie somewhere in the white region between the dashed lines. Thus, at the leaf level, we must generate FIT entries for all the sample points in this white region. To give a concrete example, node B in Figure 1 has maximum feasible rates of 1/3 and 1 locally, and a bottom-up pass from B to A reveals that maximum feasible

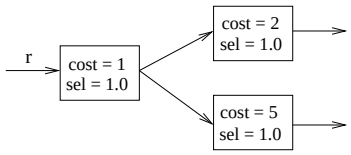


Figure 7: Splits supported by complementary plans

rates are 1/3 and 1/2 globally. As a result, the triangle between (0, 0), (1/3, 0), and (0, 1/2) constitutes the feasibility triangle and can be excluded from B's FIT.

6.2.2 Generating Complementary Local Plans

We next discuss why complementary local plans are needed and how they are generated.

Consider the query network in Figure 7. The input splits into two branches. The top branch saves 2 processing units per dropped tuple, whereas the bottom branch saves 5 processing units per dropped tuple. Also, dropping from the input saves 8 processing units while causing a loss of 2 output tuples. Dropping from the bottom branch is clearly the most beneficial. Therefore, the node should drop completely from the bottom branch before starting to drop from its input. If the input rate is r, then $5{*}r$ out of the total load of $8{*}r$ should be handled locally. Thus, from parent's perspective this node appears as if it could tolerate an input rate up to 1/3. Any excess input rate between 1/8 and 1/3 will be transparently handled with local plans. For example, given $r = 0.2$, we must shed 60% of the load on the bottom branch. By doing so, we end up with a total throughput of 0.28. If we instead shed all of the excess load from the input (37.5%), then our score would be 0.25. Using complementary local plans both increases transparency between the neighboring nodes and also enables us to maximize output quality by always shedding from the branch with the least loss in quality per dropped load.

The local plans at a leaf server are generated as follows:

- We make a list of all drop locations (i.e., split and input arcs).
- For each drop location, we compute the *quality/load* ratio.
- We greedily sort the drop locations in ascending order of their ratios. More specifically, after the drop location with the smallest ratio is included in the sorted list, the ratios of the remaining dependent drop locations are updated. This process continues until all the drop locations are included in the sorted list.
- As before, we determine the maximum feasible rate (after the local plans have been applied) along each input dimension, and choose a sample of FIT points that are properly distanced as explained in Section 6.2.1.
- For each FIT point that is feasible with a plan, we go through the sorted list of drop locations, each time picking the one with the smallest ratio. Each drop location can save up to a certain amount of load based on its load factor. Depending on our excess load amount, we should pick enough drop locations from the list that would in total save us the required amount of load. The local load shedding plan will then consist of drop operators to be placed at the selected drop locations.

Note that error in quality is still kept within the ϵ threshold by choosing FIT points accordingly and by selecting the local drop locations in an optimal order. The formal proof is omitted here due to space limitations.

6.3 FIT Merge and Propagation

In this section, we describe FIT generation at non-leaf nodes based on FIT merge and propagation from their child nodes.

Assume two server nodes like A and B as in Figure 1, where A is upstream from B. After we compute FIT for leaf node B as explained in the previous section, we propagate it upstream to A. The feasible points in B's FIT are expressed in terms of B's inputs, which correspond to the rates at A's outputs. To be able to propagate the FIT further upstream, we have to first express B's FIT in terms of A's inputs. Each input i of A follows a query path to produce a certain output. Along this path, the rate of i changes by a factor determined by the product of the operator selectivities (say sel_i). Therefore, given an output rate r, the corresponding input rate for i is $\frac{r}{sel_i}$. To obtain A's FIT, we first apply this reverse-mapping to each row of B's FIT; the corresponding score for each row stays the same. Then, we eliminate from the resulting FIT the entries which may be violating A's load constraint.

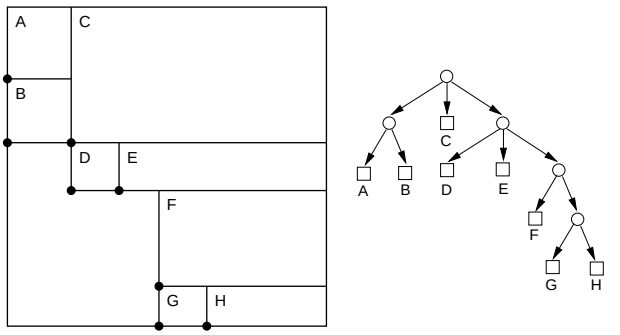
In addition to this simple linear query network/single server chain case, there are three special cases to consider:

- If there is a split along the path from an input i to multiple outputs, and if all child branches of the split map to the same input rate value, then we just propagate that value as described above. Otherwise, we propagate the maximum of all input rates. The assumption here is that any additional reduction will be performed by applying tuple drops at remaining branches of the split. The additional reduction is stored as a complementary local load shedding plan associated with that particular FIT entry, and need not be propagated further upstream.
- If there is a merge of two inputs i and j via a binary operator and if we want to propagate an output rate r upstream from this operator, then we need to generate all (r_i, r_j) rate pairs that can potentially result in r. If the operator is a Union, then all rate pairs that sum up to r needs to be generated. If it is a Join with input windows of w_i and w_j, and selectivity s, then all rate pairs where $r_i * s * w_j + r_j * s * w_i = r$ needs to be generated.
- If node A has multiple child nodes, then the FITs of these children are combined by merging rows from each FIT with the rows from the other FITs. Any new entry violating A's load constraint has to be eliminated. The resulting score is the sum of the children's row scores.

Note that merging multiple FITs at a parent may generate a large number of new FIT entries. To deal with this problem, we employ a number of heuristic tactics. For example, if multiple entries are very similar in their rate and score values, we delete all but the one which would cause the smallest processing load. Similarly, if multiple entries have similar rate values, we only keep the one with the largest score. We omit the rest of the tactics to save space.

6.4 Point-Quadtree-based Division and Indexing of the Input Rate Space

When FITs are propagated all the way from leaves to the roots, we obtain one FIT per node that represents the feasible points for the complete subtree under this node. Next we want to divide the multi-dimensional input rate space for this node into subspaces where each subspace can be mapped to a unique FIT entry. Consider a 2-dimensional space. Let $p(x_1, y_1)$ be a FIT point in this space. Any infeasible point $q(x_2, y_2)$ where $x_2 \geq x_1$ and $y_2 \geq y_1$ could potentially be mapped to p. In fact, if the given bounds of our 2-dimensional space are (x_{max}, y_{max}), then the complete subspace between p and (x_{max}, y_{max}) could be mapped to p. However, we might like to map some portions of this subspace to other FIT entries that might possibly have higher quality scores. In order to come up with the best mapping, we do the following: Assume that t is the top-most point of our multi-dimensional input rate space. For each FIT point p, we first map the subspace between



(a) Space division for FIT (b) Space index for FIT

Figure 8: Point-Quadtree-based space division and index

p and t to p. Then we compare the score of FIT entry p with the scores of the FIT entries for the already existing subspaces. If we find a subspace S whose score is larger than the new subspace N, then we must reduce our new subspace by subtracting S from N. On the other hand, if we find a subspace S whose score is smaller than the new subspace N, then we must reduce S by subtracting N from S. When we do this for each FIT entry, we end up with a disjoint space division where each subspace is mapped to the FIT entry with the highest possible score. Figure 8(a) illustrates how such a division may look like.

As in the solver-based approach, we build a quadtree-based index on top of our space subdivision. However, in the FIT case, instead of dividing the space into regular subspaces and creating plans along the way, we start with a set of plan points and create the subspaces of irregular size based on the existing plan points. Therefore, we end up with a "point-quadtree" [18].

7. PUTTING IT ALL TOGETHER

In this section, we briefly summarize how the other three phases of distributed load shedding (i.e., Load Monitoring, Plan Selection, and Plan Implementation) are performed.

Centralized Case. In the centralized approach, the coordinator periodically measures the input rates at the root nodes. Based on the derived input metadata and load factors, the coordinator can estimate the CPU load at each server for the observed input rates. If the CPU load on one or more of the servers is estimated to be above the capacity, then the coordinator searches the quadtree index to locate the load shedding plan to be applied. Otherwise, no server is overloaded and any existing drops in the query plan are removed.

In the case of overload, the coordinator sends the id of the selected plan to each of the server nodes to trigger the Plan Implementation phase at these servers. Furthermore, inputs may require additional scaling so that the infeasible point exactly matches the plan point. This additional scaling information is also sent to the input servers as additional drops to complement the selected plan.

In the Plan Implementation phase, each server node locates the load shedding plan from its precomputed plan table that was uploaded earlier by the coordinator, and changes the query network by removing redundant drops and adding new drops as necessary.

Distributed Case. In the distributed approach, all nodes periodically measure their local input rates and estimate their local CPU load based on these observed input rates. If an overload is detected, a node uses its local quadtree index (built on top of its local FIT) to locate the appropriate local load shedding plan to be applied. Previously inserted drops, if any, must be removed from the local query plan. Note that in all cases, except when local complementary plans are needed due to splits, parent nodes ensure that all the nodes in their subtree only get feasible input rates.

The quadtree is used in a similar way as described above, except that instead of sending plan id's to others, each node directly applies the selected local plan. Again, inputs that require additional scaling are handled in the same way as in the centralized case.

8. PERFORMANCE STUDY

8.1 Experimental Setup

We implemented our approaches as part of the load shedder component of our Borealis distributed stream processing system. We conducted our experiments on a small cluster of Linux servers, each with an Athlon 64 1.8GHz processor. We created a basic set of benchmark query networks which consisted of "delay" operators, each with a certain delay and selectivity value. A delay operator simply withholds its input tuple for a specific amount of time as indicated by its delay parameter, busy-waiting the CPU. The tuple is then either dropped or released to the next operator based on the selectivity value. As such, a delay operator is a convenient way to represent a query piece with a certain CPU cost and selectivity. We used synthetic data to represent readings from a temperature sensor as (time, value) pairs. For our experiments, the data arrival rates were more important than the actual tuple contents. We used the following workload distributions for the input rates:

- standard exponential probability distribution with a λ parameter which is commonly used to model packet inter-arrival times in the Internet, and
- real network traffic traces from the Internet Traffic Archive [2].

We present results on the following approaches:

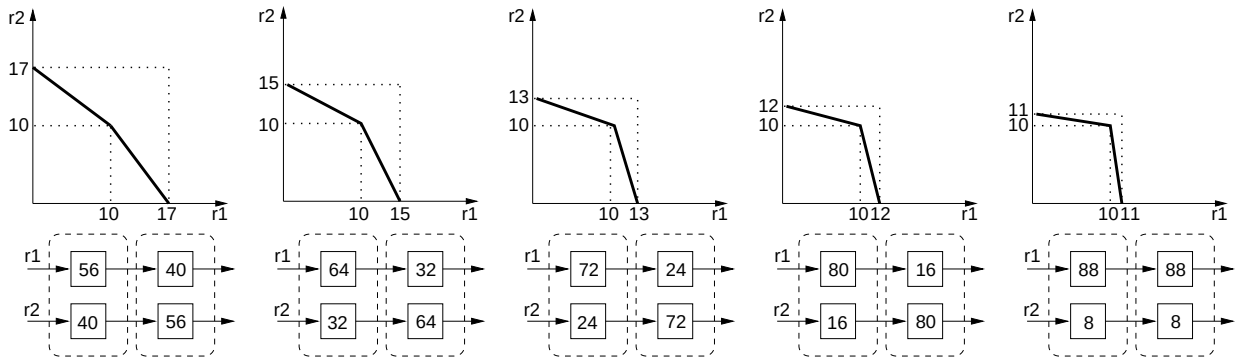
- **Solver.** The centralized solver-based algorithm that is based on the maximum percent error threshold (ϵ_{max}).
- **Solver-W.** A variation of Solver that takes workload information into account and is based on the expected maximum percent error threshold ($E[\epsilon_{max}]$).
- **C-FIT.** The centralized implementation of the FIT-based approach that is based on ϵ_{max}.
- **D-FIT.** The distributed implementation of the FIT-based approach that is based on ϵ_{max}.

8.2 Experimental Results

8.2.1 Effect of Query Load Distribution

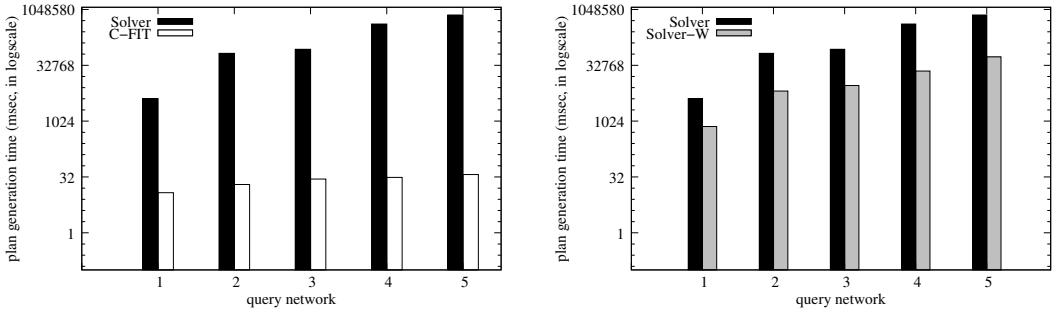
In this experiment, we investigate the effect of query load imbalance on plan generation time. Figure 9 shows the collection of query networks that we used for this experiment. Each query network simply consists of two chain queries. Each chain query is composed of two delay operators that are deployed onto two different servers. The numbers inside the operators indicate the processing cost in milliseconds. We apportioned the processing costs such that the total costs of the query networks on each server are the same, while ensuring that there is some load imbalance between two chain queries, (increasing as we go from Figure 9(a) to Figure 9(e)). In Figure 9, we also show the approximate feasibility boundary for the input rates for each query network. Basically, all input rate combinations below this boundary are feasible for both of the servers, while the points on the opposite side represent overload conditions where load shedding would be needed.

In Figure 10(a), we compare the plan generation time for Solver and C-FIT, fixing ϵ_{max} at 5%. Both of these approaches guarantee that the maximum error in total quality score will not exceed 5%. C-FIT can generate plans with the same guarantee in significantly



(a) Query Network 1 (b) Query Network 2 (c) Query Network 3 (d) Query Network 4 (e) Query Network 5

Figure 9: Query networks with different query load distributions and feasibility boundaries



(a) Solver and C-FIT (ϵ_{max} = 5%) (b) Solver and Solver-W ($\lambda = 5$, exp. ϵ_{max} = 1.44%)

Figure 10: Effect of query load imbalance

shorter time (note the logarithmic y-axis), while Solver turns out to be more sensitive to query load distribution than C-FIT. In Figure 10(b), we compare the plan generation time for Solver and Solver-W. In this case, we assumed an exponential distribution for the input rate workload with $\lambda = 5$. When we ran Solver fixing ϵ_{max} at 5%, this produced plans for which $E[\epsilon_{max}]$ turned out to be around 1.44% on the average. Then we used this value as a threshold for Solver-W. As can be seen on our graph, Solver-W takes the workload distribution into account to guarantee the given expected e_{max} value in much shorter time. Both Solver approaches show similar sensitivity to query load distribution.

8.2.2 *Effect of System Provision Level*

Next, we investigate the effect of expected system provision level on plan generation efficiency for the Solver-W. In order to estimate the provision level, we consider two types of workload: (i) one with a standard exponential workload distribution with parameter λ, and (ii) a real network traffic trace from the Internet Traffic Archive [2]. For the former case, we change the system provision level by varying λ. For the latter case, we use an existing trace from the archive, but we change the query cost in order to create different provision levels.

Figure 11(a) shows how plan generation time for the Solver-W increases with increasing λ for the five query networks of Figure 9. As λ increases, fewer input rate combinations will fall below the feasibility boundary while more will be above it (i.e., the system will appear as if it is less-provisioned). To provide more insight, in Figure 12, we provide a color-map of joint exponential probability distribution for two inputs, where axes correspond to input rates and the brightness of an area indicates the expected probability of occurrence. The high probability area shifts up as we increase λ, affecting that area's contribution to $E[\epsilon_{max}]$. As we increase λ, we thus expect the plan generation to take more time, as the solver has to be called for more points. Also as in the previous section, as the query load imbalance increases, the plan generation time increases.

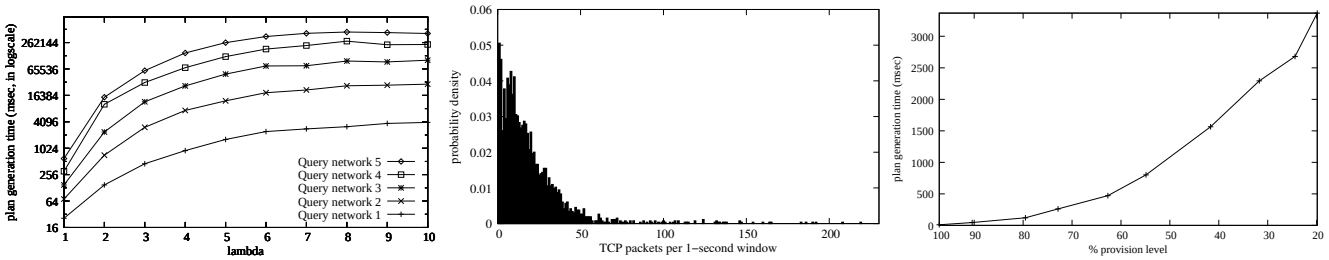
We repeated the same experiment with the TCP traces from the Internet Traffic Archive (ITA). Figure 11(b) shows this workload distribution, which essentially looks a lot like an exponential distribution. We first used the query network in Figure 9(b) (i.e., (64, 32)), which corresponds to a provision level of about 20%. We then reduced the query costs proportionally (e.g., (56, 28), (48, 24), and so on) in order to create increasingly higher provision levels with the same workload distribution. Figure 11(c) presents our result, which clearly shows that as the system is provisioned better, the plan generation time decreases.

8.2.3 *Effect of Operator Fan-out*

To examine the effect of operator fan-out on Solver and C-FIT, we used a single server deployment of a query tree with 2^k query branches that are fed by a single input stream. These queries share one common operator. Thus, load shedding plans would either place a drop at the input arc or at the split arcs downstream from this common operator. As seen in Figure 13, as we increase the degree of sharing in the query network, both approaches spend more time in plan generation. Although Solver can generate plans slightly faster than C-FIT at a fan-out value of 2, C-FIT starts to outperform Solver at higher fan-out values. Thus, C-FIT scales better with increasing operator fan-out.

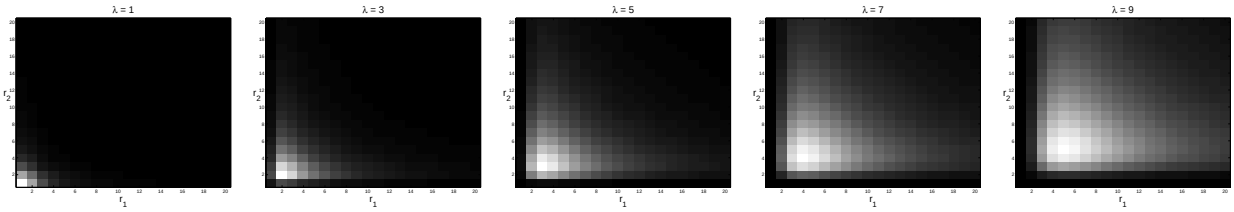
8.2.4 *Effect of Input Dimensionality*

The number of input dimensions is an important factor that can expectedly degrade the performance of plan generation. In this sec-



(a) Effect of λ (exp. ϵ_{max} = 1 %) (b) TCP workload from ITA (c) Effect of provision (exp. ϵ_{max} = 1 %)

Figure 11: Effect of workload distribution and provision level on Solver-W plan generation



(a) $\lambda = 1$ (b) $\lambda = 3$ (c) $\lambda = 5$ (d) $\lambda = 7$ (e) $\lambda = 9$

Figure 12: Exponential workload distribution for different λ values

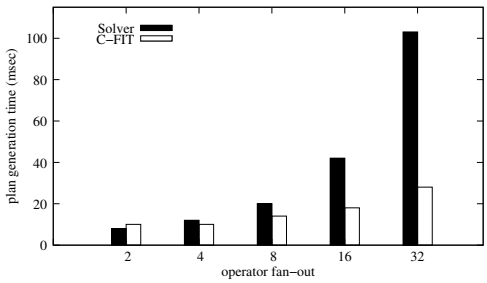


Figure 13: Effect of operator fan-out ($\epsilon_{max} = 1\%$)

# of inputs	# of FIT entries	bytes/entry
2	46	52
4	984	96
8	42472	184

Table 3: Effect of dimensionality ($\epsilon_{max} = 10$%)

tion, we examine how our algorithms are affected from increasing input dimensionality.

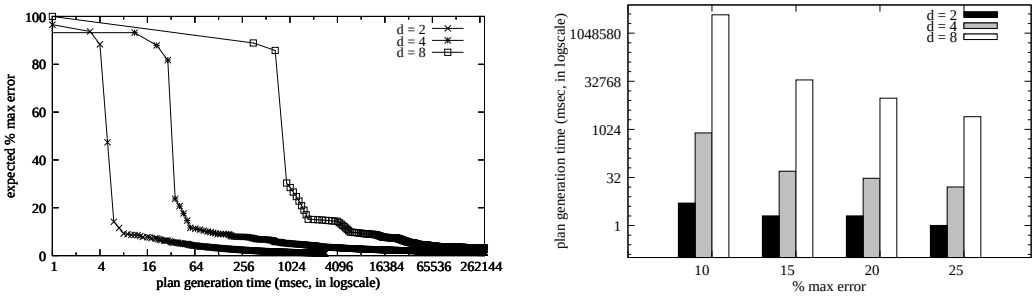
Figure 14(a) shows how $E[\epsilon_{max}]$ converges in time for query networks with 2, 4, and 8 input streams. In this experiment, we used the query network of 9(b) (i.e., (64, 32)) for the two-dimensional case. Then we increased the number of inputs while proportionally decreasing the query costs for the higher-dimensional cases (i.e., (32, 16, 32, 16) and (16, 8, 16, 8, 16, 8, 16, 8)) and assumed an exponential distribution with $\lambda = 3$. Not surprisingly, as we increase the number of input dimensions, the plan generation time significantly increases. As shown in Figure 14(b), the situation is worse for C-FIT (and similarly for the Solver, which is not shown). For example, to ensure a 10% maximum error for 8 inputs, C-FIT needs to run for about an hour. These results demonstrate the "curse of dimensionality". In practice, however, the outlook is better. First, plan generation is an off-line process; it is performed in advance of the overload, potentially using idle cycles. Second, the generated plans are often reusable. Third, if the query network consists of a number of disjoint fragments, then each fragment is handled separately. Therefore, the queries can as well be deployed to the server nodes in such a way that the number of inputs for each connected query network fragment does not exceed a certain threshold. Finally, the number of input dimensions could always be reduced by merging several inputs into one. We should also mention that, in our experience with stream-oriented queries for the financial services domain, the number of input streams are generally few.

8.2.5 Overhead Analysis for Distributed FIT

FIT is a distributed algorithm by design. As such, it can provide all the features of distributed algorithms such as avoiding hot spots and single point of failures. It is hard to provide quantitative evidence for why D-FIT would beat the coordinator-based approaches. However, as mentioned earlier, it is clear that there are certain settings where D-FIT would be preferable (e.g., multi-hop, resource-limited sensor networks) due to its ability to dynamically react to changes. In this section, we analyze overhead issues associated with D-FIT, that would bear importance in such settings.

Our main criteria in the overhead analysis is how much FIT information needs to be communicated between two neighboring nodes. Since complementary local plans associated with FIT entries are not actually being sent to the parent, FIT entry size is simply proportional to the number of input dimensions. Therefore, we will focus on the number of FIT entries rather than the total byte size. For the same reason, operator fan-out, which affects the size of the complementary plan column of FIT, is not very interesting. As a result, we identified three major sources of overhead for FIT size: (i) the number of input dimensions, (ii) magnitude of the ϵ_{max} threshold, and (iii) query load.

As Table 3 clearly shows, for a given ϵ_{max} threshold and a fixed query load, when we have more inputs, we need to represent a higher number of feasible input combinations, which consequently requires a higher number of FIT entries. For 2 inputs, Figure 15(a) details the effect of ϵ_{max} and query load. As we reduce the query load, a larger portion of the input rate space becomes feasible and this increases the number of FIT entries. Another interesting point is that if there is operator fan-out in the query plan, where some por-



(a) Convergence of exp. ϵ_{max} for Solver-W (b) Effect of dimensionality on C-FIT

Figure 14: Effect of input dimensionality

tion of the query can be handled with complementary local plans, then the query appears less costly to the parent node, and therefore, we would have a higher number of feasible point entries in the FIT. Thus, it is not the actual query load, but the load after local plans are applied that determines the needed FIT size. Figure 15(a) also shows that, if the ϵ_{max} threshold is increased, allowing a larger distance from optimal quality plans, then the number of required FIT entries decreases in a dramatic way. This suggests that ϵ_{max} can actually be adaptively adjusted to trade off plan quality for reduced communication overhead. This could also be used as a remedy to the high dimensionality problem.

One major advantage of D-FIT over the centralized approaches is that it enables efficient handling of certain dynamic changes in load conditions. Depending on the nature of the change, there may be cases where it would be sufficient to only update the local FIT, or propagate it only to a small number of upstream nodes. Yet in other cases, it would be possible to send deltas to the parent rather than the complete FIT. We identified two important cases to analyze: (i) changes in operator selectivity, and (ii) changes in query load due to operator movement between neighboring nodes. Figure 15(b) shows the sensitivity of FIT to operator selectivity change. In this experiment, we used two chain queries. One of the operators in one of these queries initially has a selectivity of 1.0. We compute the FIT to be sent to the parent node. Then we decrease the selectivity and recompute the FIT. We then measure what fraction of the new FIT entries must be communicated to the parent in order to stay within some distance from the actual quality score. We find that as the change in selectivity increases, we need to send more entries in order to achieve a certain difference threshold. Similarly, for a given selectivity change, we need to send more entries if we want to stay within a smaller difference from the actual quality score. Figure 15(c) shows the sensitivity of FIT to load movement. In this case, we again use two chain queries similar to the ones in Figure 9. Then we reduce the delay parameter of a parent operator, while adding that same amount to the delay parameter of the downstream child operator (e.g., parent-to-child-8 means that we moved 8 units of load from parent to child). We see that as the amount of load moved gets bigger, it requires more FIT entries to be sent to the parent. In the case for parent-to-child-24, the load movement causes a complete reversal in load balance. Therefore, it is not even possible to to reduce the communication overhead beyond 8%, no matter how large a difference we allow.

9. RELATED WORK

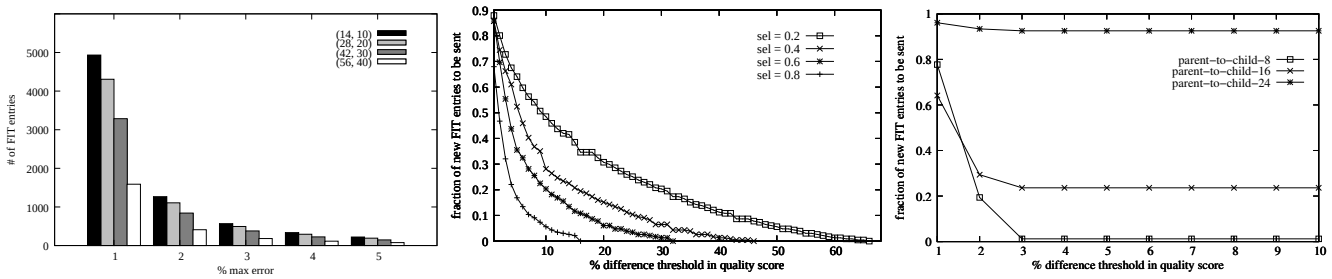
The load shedding problem has been explored extensively for centralized stream processing. In our earlier work with Aurora, drop operators are selectively inserted into running query plans to maximize various QoS-based optimization metrics [22]. This work also utilized load shedding plans that are computed off-line to ensure fast response to overload. STREAM introduced several techniques, one of which provides a statistical approximation for aggregation queries [7]. TelegraphCQ proposed an adaptive load shedding approach, called data triage, which creates summaries of data instead of dropping them [17]. Ayad and Naughton focused on load shedding for join queries in order to maximize the query throughput [6].

The overload management problem has also been studied in the context of push-based data dissemination systems. For example, the Salamander pub-sub system supports application-level QoS policies by allowing clients to plug in their data flow manipulation modules at any point in the data dissemination tree [15]. These modules can prioritize, interleave, or discard certain data objects to adapt to changing workload and network conditions. Unlike our approach, Salamander does not try to coordinate the flow modification actions performed at different points on the data distribution tree. Another example is the data stream dissemination system studied by Shah et al., which selectively disseminates data updates among a network of data repositories to preserve user-defined coherency requirements [20].

A recent, closely related work comes from Amini et al., who proposed a two-tiered approach that combines long-term operator placement and short-term CPU scheduling to maximize throughput in a distributed stream processing system [5]. This is a control-based (closed-loop) solution that continually adjusts the buffer sizes at each node to achieve high throughput and low latency, while ensuring stability in the presence of varying workload and bursts. Besides other differences, our solution uses an open-loop approach to create parametric load shedding plans that can limit the deviation from the optimal plan.

Distributed load shedding is also relevant to the congestion control problem in computer networks [13]. Various IP-layer architectures have been proposed to maintain Internet QoS including IntServ [9] and DiffServ [8]. OverQoS, an overlay-based QoS architecture, uses application-level techniques to prioritize packets that have higher utility for the application [21]. Our upstream metadata propagation technique resembles the pushback mechanism developed for aggregate congestion control [14], where a congested router can request its upstream routers to limit the rate of an aggregate (i.e., a certain collection of packets sharing a common property) primarily to defend against DoS attacks. In our FIT-based approach, nodes also specify their feasible input rates to their parents, but it is the responsibility of the parent to decide how to reduce outgoing stream rates to maximize system throughput.

We also note that our off-line load shedding plan generation approach can be regarded as an instance of parametric query optimization [10, 11, 12], where a set of candidate query plans, each of which is optimal for some region of the parameter space, is precomputed and the appropriate one chosen at run-time based on the



(a) Effect of ϵ_{max} and query load (b) Sensitivity to selectivity change (c) Sensitivity to load movement

Figure 15: D-FIT overhead

observed parameter values. Unlike previous instances of the problem that studied standard query optimization in single-server environments, our solution addresses throughput optimization under overload in distributed settings.

10. SUMMARY AND FUTURE WORK

In this paper, we have studied the problem of load shedding in distributed stream processing. We have shown how it differs from previous centralized solutions, and we have offered several new practical algorithms for addressing the problem. We presented our main solution approach, a distributed algorithm that we call D-FIT that works by transmitting its load requirements locally to its parents. We also investigated several centralized solutions – a linear programming solution (Solver), a variant on Solver that takes a workload history into account (Solver-W), and a centralized version of our distributed algorithm (C-FIT).

As we have said earlier, the purely distributed version of our algorithm D-FIT is especially useful when the underlying query environment is dynamic as in a distributed sensor network. We believe that the distributed algorithm would also scale better in large-scale deployments. Our current results from small-scale cluster deployments look promising. Verifying the practicality and effectiveness of our approaches in larger-scale deployments is left as future work.

The current study restricts the topology of the overlay network to be a tree. In the future, we will try to relax this assumption by looking at other topologies directly, or by reducing a non-tree topology by cutting it into a tree of non-tree clusters. In each cluster, a given node may need to communicate with nodes other than its parent. Solutions here might include using C-FIT for each cluster and D-FIT for communication between the clusters.

In this paper, we restricted our focus on the CPU problem only. We will also investigate the overload problem under bandwidth constraints. This may require pushing the load shedding actions farther upstream to reduce bandwidth usage, even if the upstream node itself is not overloaded.

Acknowledgments. We would like to thank Çağatay Demiralp for his help with MATLAB. This work has been supported in part by the NSF under the grants IIS-0086057 and IIS-0325838, and by ETH under the grant 0-42386-06.

11. REFERENCES

[1] The GNU Linear Programming Kit (GLPK). `http://www.gnu.org/software/glpk/`.

[2] The Internet Traffic Archive. `http://ita.ee.lbl.gov/`.

[3] D. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR Conference*, Asilomar, CA, 2005.

[4] D. 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. *VLDB Journal*, 12(2), August 2003.

[5] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive Control of Extreme-scale Stream Processing Systems. In *IEEE ICDCS Conference*, Lisboa, Portugal, July 2006.

[6] A. Ayad and J. F. Naughton. Static Optimization of Conjunctive Queries with Sliding Windows Over Infinite Streams. In *ACM SIGMOD Conference*, Paris, France, June 2004.

[7] B. Babcock, M. Datar, and R. Motwani. Load Shedding for Aggregation Queries over Data Streams. In *IEEE ICDE Conference*, Boston, MA, March 2004.

[8] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. IETF RFC 2475, December 1998.

[9] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: An Overview. IETF RFC 1633, June 1994.

[10] S. Ganguly. Design and Analysis of Parametric Query Optimization Algorithms. In *VLDB Conference*, New York City, NY, August 1998.

[11] A. Hulgeri and S. Sudarshan. Parametric Query Optimization for Linear and Piecewise Linear Cost Functions. In *VLDB Conference*, Hong Kong, China, August 2002.

[12] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric query optimization. In *VLDB Conference*, Vancouver, Canada, August 1992.

[13] V. Jacobson. Congestion Avoidance and Control. *ACM SIGCOMM Computer Communication Review*, 18(4), August 1988.

[14] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling High Bandwidth Aggregates in the Network. *ACM SIGCOMM Computer Communication Review*, 32(3), July 2002.

[15] G. R. Malan, F. Jahanian, and S. Subramanian. Salamander: A Push-based Distribution Substrate for Internet Applications. In *USENIX USITS Symposium*, Monterey, CA, December 1997.

[16] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. In *IEEE ICDE Conference*, Atlanta, GA, April 2006.

[17] F. Reiss and J. Hellerstein. Data Triage: An Adaptive Architecture for Load Shedding in TelegraphCQ. In *IEEE ICDE Conference*, Tokyo, Japan, April 2005.

[18] H. Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(2), June 1984.

[19] M. A. Shah, J. M. Hellerstein, and E. Brewer. Highly-Available, Fault-Tolerant, Parallel Dataflows. In *ACM SIGMOD Conference*, Paris, France, June 2004.

[20] S. Shah, S. Dharmarajan, and K. Ramamritham. An Efficient and Resilient Approach to Filtering and Disseminating Streaming Data. In *VLDB Conference*, Berlin, Germany, September 2003.

[21] L. Subramanian, I. Stoica, H. Balakrishnan, and R. Katz. OverQoS: An Overlay Based Architecture for Enhancing Internet QoS. In *USENIX NSDI Symposium*, San Francisco, CA, March 2004.

[22] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *VLDB Conference*, Berlin, Germany, September 2003.

[23] N. Tatbul and S. Zdonik. Window-aware Load Shedding for Aggregation Queries over Data Streams. In *VLDB Conference*, Seoul, Korea, September 2006.