Load Management and High Availability in the Borealis Distributed Stream Processing Engine

Nesime Tatbul[1], Yanif Ahmad[2], Uğur Çetintemel[2], Jeong-Hyon Hwang[2], Ying Xing[2], and Stan Zdonik[2]

[1] ETH Zürich, Department of Computer Science, Zürich, Switzerland
tatbul@inf.ethz.ch
[2] Brown University, Department of Computer Science, Providence, RI, USA
{yna, ugur, jhhwang, yx, sbz}@cs.brown.edu

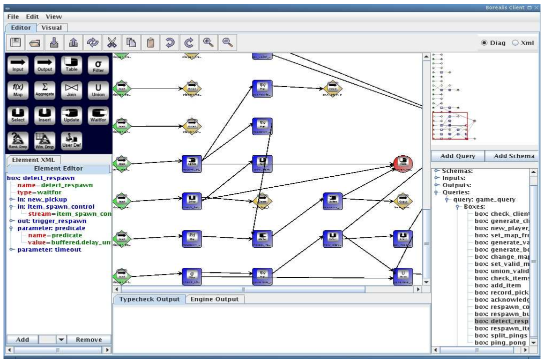
Abstract. Borealis is a distributed stream processing engine that has been developed at Brandeis University, Brown University, and MIT. It extends the first generation of data stream processing systems with advanced capabilities such as distributed operation, scalability with time-varying load, high availability against failures, and dynamic data and query modifications. In this paper, we focus on aspects that are related to load management and high availability in Borealis. We describe our algorithms for balanced and resilient load distribution, scalable distributed load shedding, and cooperative and self-configuring high availability. We also present experimental results from our prototype implementation showing the effectiveness of these algorithms.

1 Introduction

In the past several years, data streaming applications have become very common. The broad range of applications include financial data analysis [1], network traffic monitoring [2], sensor-based environmental monitoring [3], GPS-based location tracking [4], RFID-based asset tracking [5], and so forth. These applications typically monitor real-time events and generate high volumes of continuous data at time-varying rates. Distributed stream processing systems have emerged to address the performance and reliability needs of these applications (e.g., [6], [7], [8], [9]).

Borealis is a distributed stream processing engine that has been developed at Brandeis University, Brown University, and MIT. It builds on our earlier research efforts in the area of stream processing - Aurora and Medusa [10]. Aurora provides the core stream processing functionality for Borealis, whereas Medusa enables inter-node communication. Based on the needs of recently emerging stream processing applications, Borealis extends both of these systems in non-trivial and critical ways to provide a number of advanced capabilities. More specifically, Borealis extends the basic Aurora stream processing system with the ability to:

- operate in a distributed fashion,
- dynamically modify various data and query properties without disrupting the system's run-time operation,



```
box: detect_respawn
    name=detect_respawn
    type=waitfor
  in: new_pickup
  in: item_spawn_control
      stream=item_spawn_con
  out: trigger_respawn
  parameter: predicate
      name=predicate
      value=buffered.delay_un
  parameter: timeout
```

```
Schemas:
Inputs:
Outputs:
Queries:
  query: game_query
    Boxes:
      box: check_client
      box: generate_cli
      box: new_player_
      box: set_map_fro
      box: generate_va
      box: generate_bo
      box: change_map
      box: set_valid_m
      box: union_valid
      box: check_items
      box: add_item
      box: record_pick
      box: acknowledg
      box: respawn_co
      box: respawn_bu
      box: detect_resp
      box: respawn_ite
      box: split_pings
      box: ping_pong
```

Fig. 1. Borealis query editor

- dynamically optimize processing to scale with changing load and resource availability in a heterogeneous environment, and
- tolerate node and network failures for high availability.

In this paper, we focus on two key aspects of distributed operation in Borealis. Distributing stream processing across multiple machines mainly provides the following benefits:

- **Scalability.** The system can scale up and deal with increasing load or time-varying load spikes with the addition of new computational resources.
- **High availability.** Multiple processing nodes can monitor the system health, and can perform fail-over and recovery in the case of node failures.

In the rest of this paper, we first present a brief overview of the Borealis system. Then we summarize our work on three different aspects of distributed operation in Borealis: load distribution (Section 3), distributed load shedding (Section 4), and high availability (Section 5). Finally, we briefly discuss our plans for future research and conclude.

2 Borealis System Overview

Borealis accepts a collection of continuous queries, represents them as one large network of query operators (also known as a query diagram), and distributes the processing of these queries across multiple server nodes. Sensor networks can also participate in query processing behind a sensor proxy interface which acts as another Borealis node [11].

Queries are defined through a graphical editor, while important run-time statistics such as CPU utilizations of the servers, latencies of the system outputs, and percent data delivery at the outputs are visualized through our performance monitor [12]. Figure 1 provides a snapshot from the editor part of our system GUI.

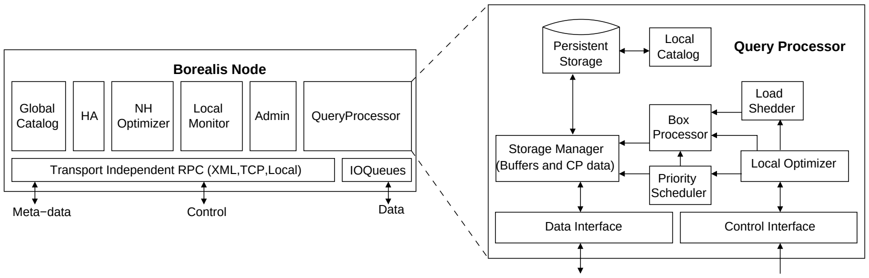


Fig. 2. Borealis system architecture

Each node runs a Borealis server whose major components are shown in Figure 2. The *query processor (QP)* forms the essential piece where local query execution takes place. Most of the core QP functionality is provided by parts inherited from Aurora [13]. *I/O queues* feed input streams into the QP and route tuples between remote Borealis nodes and clients.

The *admin* module is responsible for controlling the local QP, performing tasks such as setting up queries and migrating query diagram fragments. This module also coordinates with the *local optimizer* to provide performance improvements on a running diagram. The local optimizer employs various tactics including, changing local scheduling policies, modifying operator behavior on the fly via special control messages, and locally discarding low-utility tuples via load shedding when the node is overloaded.

The QP also contains the *storage manager*, which is responsible for storage and retrieval of data that flows through the arcs of the local query diagram, including memory buffers and *connection point (CP)* data views. Lastly, the *local catalog* stores query diagram description and metadata, and is accessible by all the local components.

Other than the QP, a Borealis node has modules which communicate with their respective peers on other Borealis nodes to take collaborative actions. The *neighborhood optimizer* uses local load information as well as information from other neighborhood optimizers to improve load balance between nodes or to shed load in a coordinated fashion. The *high availability (HA)* modules on different nodes monitor each other and take over processing for one another in case of failures. The *local monitor* collects performance-related statistics as the local system runs to report to local and neighborhood optimizer modules. The *global catalog* provides access to a single logical representation of the complete query diagram.

In addition to the basic node architecture shown in Figure 2, a certain Borealis server can be designated as the coordinator node to perform global system monitoring and to run various global optimization algorithms, such as global load distribution and global load shedding. Thus, Borealis essentially provides a three-tier monitoring and optimization hierarchy (local, neighborhood, and global) that works in a complementary fashion [7].

3 Load Distribution in Borealis

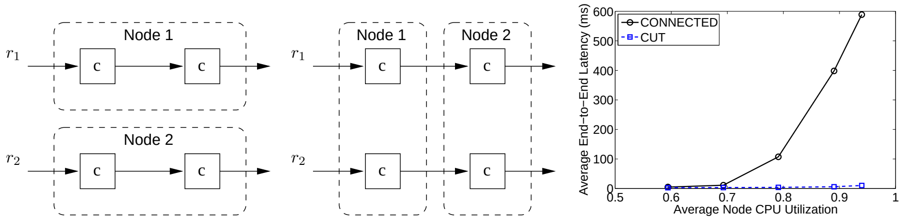
Distributed stream processing engines can process more data at higher speeds by distributing the query load onto multiple servers. The careful mapping of query operators onto available processing nodes is critical in enduring unpredictable load spikes, which otherwise might cause temporary overload and increase in latencies. Thus, the problem involves both coming up with a good initial operator placement as well as dynamically changing this placement as data arrival rates change. Borealis provides two complementary mechanisms to deal with this problem:

- a correlation-based operator distribution algorithm, which exploits the relationship between the load variations of different operators, as well as nodes, in determining and dynamically adjusting the placement of the operators in a balanced way, and
- a resilient operator distribution algorithm, whose primary goal is to provide a static operator placement plan that can withstand the largest possible set of input rate combinations without the need for redistribution.

In this section, we briefly summarize these mechanisms.

3.1 Correlation-based Operator Distribution

To minimize end-to-end latency in a push-based system such as Borealis, it is important, but not enough, to evenly distribute the average load among the servers. The variation of the load is also a key factor in determining the system performance. For example, consider two operator chains. Each chain consists of two identical operators with cost c and selectivity 1. When the average input rates of the two input streams (r_1 and r_2) are the same, the average loads of all operators are the same. Now consider two operator mapping plans on two nodes. In the first plan, we put each of the two connected operator chains on the same node (Figure 3(a)). In the second plan, we place each operator of a chain on a different node (Figure 3(b)). There is no difference between these two plans from the load balancing point of view. However, suppose that the load bursts of the two input streams happen at different times. For example, assume that $r_1 = r$ when $r_2 = 2r$, or $r_1 = 2r$ when $r_2 = r$. Then these two plans result in very different performance. In the connected plan, there is a clear imbalance between the two nodes in both burst scenarios (Node1's load is $2cr$, when Node2's load is $4cr$, and vice versa); whereas in the cut plan, the load balance is maintained for both of the scenarios (Both nodes have the load of $3cr$ in both cases). Since the two bursts are out of phase, the cut plan which groups operators with low load correlation together, ensures that the load variation on each node is kept small. The simulation result presented in Figure 3(c), which shows that the cut plan



(a) Connected plan (b) Cut plan (c) Latencies

Fig. 3. Comparison of different operator mapping plans with fluctuating load

can achieve smaller latency with increasing load, also confirms this observation. This simple example clearly shows that, not only the average load, but also the load variation must be considered to achieve a good operator placement that can withstand bursts.

In Borealis, we try to balance the average load among the processing nodes, but we also try to minimize the load variance on each node. For the latter goal, we exploit the correlation of stream rates across the operators. More specifically, we represent operator load as fixed-length time series. The correlation of two time series is measured by the correlation coefficient, which a real number between -1 and 1. Intuitively, when two time series have a positive correlation coefficient, if the value of one time series at a certain index is relatively large in comparison to its mean, then the value of the other time series at the same index also tends to be relatively large. On the other hand, if the correlation coefficient is negative, then when the value of one time series is relatively large, the value of the other tends to be relatively small. Our algorithm is inspired by the observation that if the correlation coefficient of the load time series of two operators is small, then putting these operators together on the same node helps in minimizing the load variance.

The intuition of correlation is also the foundation of the other idea in our algorithm: when making operator allocation decisions, we try to maximize the correlation coefficient between the load statistics of different nodes. This is because moving operators will result in temporary poor performance due to the execution suspension of those operators. However, if the load time series of two nodes have large correlation coefficient, then their load levels are naturally balanced even when the load changes. By maximizing the average load correlation between all node pairs, we can minimize the number of load migrations needed.

As we showed in an earlier paper [14], minimizing the average load variance in fact also helps in maximizing the average load correlation, and vice versa. Therefore, the main goal of our load distribution algorithms is to produce a balanced operator mapping plan where the average operator load variance is minimized or the average node load correlation is maximized. More formally, assume that there are n nodes in the system. Let X_i denote the load time series of node N_i and ρ_{ij} denote the correlation coefficient of X_i and X_j for $1 \leq i, j \leq n$. We want to find an operator mapping plan with the following properties:

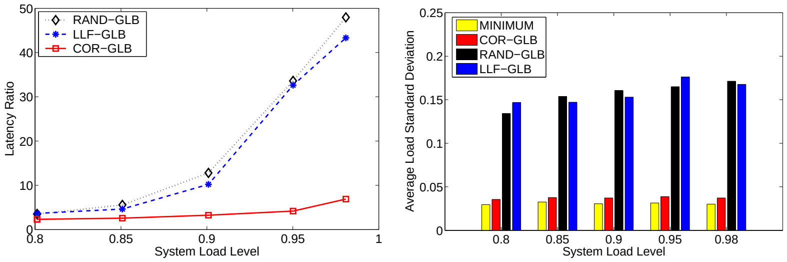
- $EX_1 \approx EX_2 \approx ... \approx EX_k$
- $\frac{1}{n}\sum_{i=1}^{n} varX_i$ is minimized, or
- $\sum_{1 \leq i < j \leq n} \rho_{ij}$ is maximized.

Finding the optimal solution to this problem requires comparison of all possible mapping plans and is NP hard. Instead, we developed a number of greedy heuristics which helps us find sub-optimal solutions in polynomial time, and which can experimentally be shown to perform very close to the optimal.

The Borealis coordinator periodically collects load statistics from all nodes, orders nodes by their average load, and pairs them by grouping the i^{th} node with the $(n-i+1)^{th}$ node in the ordered list. If the load difference between a node pair is above a certain threshold, then operators need to be moved between those nodes to balance their average load in a way that also minimizes their average load variance. Given such a pair, the load movement can be either one-way or two-way:

- In the one-way case, only the more loaded node is allowed to offload half of its excess load to its mate; the purpose is to reduce the load movement overhead. The operators of the more loaded node (say N_1) are ordered based on a score, and the operator with the largest score is moved across to the other node (say N_2) in a greedy fashion until the balance is achieved. The score for an operator o represents the difference between the correlation coefficient between o and the rest of the operators at N_1, and the correlation coefficient between o and the rest of operators at N_2. A larger score makes o a desirable candidate for movement, since this way, the average load variance for the pair can be decreased.
- In the two-way case, all operators on both members of the pair can be moved across freely. Initially, both nodes are treated as empty nodes. At each iteration, we select an operator from the pool of unmapped operators with the largest score and place it at the less loaded node. We continue until all operators have been mapped to one of the nodes. This two-way algorithm can result in a better mapping plan than the one-way algorithm; however, the load movement overhead can be unnecessarily high, especially when the former mapping was relatively good. To address this problem, we add a selective exchange step to our algorithm which would only allow the two-way movement of operators whose score is above certain threshold. By varying this threshold, we can control the tradeoff between the amount of load moved and the quality of the resulting mapping plan.

The above correlation-based load redistribution algorithms can also be modified to handle the case of initial load distribution when all nodes are empty.



(a) Latency ratio

(b) Average load variance

Fig. 4. Performance comparison for correlation-based global algorithm with others

The algorithm is very similar to the two-way case except that the score formula should be generalized to n nodes rather than considering a single pair. The algorithm in this case is global rather than pair-wise.

In Figure 4, we compare our correlation-based load distribution algorithm against two other load balancing alternatives (randomized load balancing (RAND-GLB) and largest load first load balancing (LLF-GLB)). Figure 4(a) shows that our algorithm maintains low latency with increasing load, and Figure 4(b) confirms that the resulting average load variance is also much smaller (and very close to the optimal) for our algorithm. A detailed description of all of our dynamic load distribution algorithms along with their theoretical and experimental performance evaluation can be found in our earlier work [14].

3.2 Resilient Operator Distribution

Dynamic load distribution techniques described in the previous subsection for balancing load and minimizing latency in the face of unpredictable load variations are more suitable for medium-to-long term load variations, since they persist for relatively longer periods of time and are thus rather easy to capture. Furthermore, the overhead of load redistribution is amortized over time. On the other hand, short-term load fluctuations are both difficult to capture due to their transient nature and too heavy-weight to handle through operator redistribution. To give a concrete example, the base overhead of run-time operator migration in Borealis is measured to be on the order of a few hundred milliseconds (higher for operators with larger state) [15]. Thus, for these kinds of scenarios where operator movement is rather prohibitive, Borealis provides a static resilient operator distribution algorithm.

A resilient operator distribution (ROD) is one that does not become overloaded easily in the face of bursty and fluctuating input rates. This is achieved by optimizing the system to handle as many load points as possible so that it can tolerate those load conditions without the need for operator migration. More specifically, we model the load of each operator as a function of operator costs, selectivities, and system input stream rates. For given input stream rates and a given operator distribution plan, the system is either *feasible* (i.e., none of the nodes are overloaded), or *infeasible* (i.e., at least one node is overloaded). The set of all feasible input rate combinations defines a *feasible set*. Thus, our goal is to find an operator distribution plan that maximizes the size of this feasible set.

Our approach to this problem is based on a linear algebraic model. In this model, we consider a multi-dimensional space of input stream rates, where each processing node is represented by a hyperplane that consists of all input rate points that render this node fully loaded. These node hyperplanes collectively determine the shape and size of the feasible set. Thus, our goal is to find the "ideal" hyperplane which gives us the largest feasible set size. We mathematically showed that this "ideal" feasible set can be achieved if all node hyperplanes are identical (i.e., if the load of each stream is perfectly balanced across all nodes) [15]. However, the ideal feasible set may not always be achievable in practice. Therefore, our main goal is to make the node hyperplanes as close to the ideal hyperplane as possible.

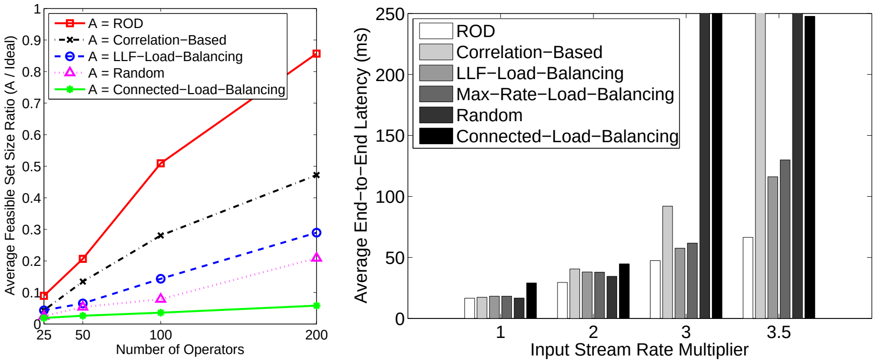
Enumerating all possible operator distribution plans and comparing their feasible set sizes to find an optimal plan is intractable when the number of inputs or the number of operators is large [15]. Therefore, we developed a greedy ROD algorithm which is driven by the following two heuristics:

- **MaxMin Axis Distance (MMAD).** Push the intersection points of the node hyperplanes along each axis, towards those of the ideal hyperplane.
- **MaxMin Plane Distance (MMPD).** Push node hyperplanes directly towards the ideal hyperplane.

Intuitively, MMAD tries to balance the load of each input stream across the nodes in proportion to their CPU capacities, whereas MMPD focuses on the combination of the impact of different input streams on each node to avoid creating bottlenecks at certain nodes. In other words, MMPD tries to balance the load of the nodes in proportion to their CPU capacities for multiple workload points.

The ROD algorithm appropriately combines these heuristics and consists of the following two steps:

- **Operator Ordering.** Sort the operators in descending order of their effect on load.
- **Operator Assignment.** Iteratively assign each operator in the ordered list to a node such that the reduction in the final feasible set size would be minimal. Given an operator *o*, it is assigned to one of the nodes using a combination of our MMAD and MMPD heuristics. More specifically, at each assignment step, we first separate the nodes into two classes. In Class I, we include those nodes that will not lead to a reduction in the final feasible set size, whereas in Class II, we have the remaining ones. If Class I is not empty, then we choose a node from this class (either randomly or based on another orthogonal criteria [15]), and assign *o* to this node. Otherwise, *o* is assigned to the node from Class II which will bear the maximum plane distance. In



(a) Resiliency (b) Average latency

Fig. 5. ROD performance

other words, when we assign operators to Class I nodes, we push the axis intersection points closer to those of the ideal hyperplane as in the MMAD heuristic. On the other hand, when we assign them to Class II nodes, we follow the MMPD heuristic and select the node which has the largest plane distance.

In Figure 5, we show two base results from our performance study on the Borealis prototype running on 10 homogeneous server nodes. We used aggregation-based network traffic monitoring queries. Figure 5(a) compares the feasible set size achieved by different operator distribution algorithms (relative to the ideal). ROD clearly outperforms all of the other alternatives, including our correlation-based load balancing algorithm that was summarized in Section 3.1. As the number of operators increases, ROD approaches to the ideal case and most of the other algorithms also improve because there is a greater chance that the load of a given input stream will be spread across multiple nodes. Figure 5(b) compares average end-to-end latency achieved as a result of applying various load distribution algorithms as the CPU utilization is increased from 26% to 79% (corresponding to input rates multipliers of 1 and 3.5, respectively). Since ROD produces the largest feasible set size and since it balances the node loads considering multiple input rate combinations, it performs and scales better than the other alternatives. Our results demonstrate that, for a representative workload and data set, ROD (i) sustains longer and is more resilient than the alternatives, and (ii) despite its high resiliency, it does not sacrifice latency performance.

We have also extended our ROD algorithm to handle nonlinear load models, to exploit additional workload information, and to consider communication costs. Details of ROD and its extensions together with their detailed performance results can be found in our earlier work [15].

4 Distributed Load Shedding in Borealis

Data streams can arrive in bursts and provisioning the system for the worst-case load (which can be orders of magnitude higher than the average load) is in general not economically sensible. On the other hand, bursts in data rates may create overload on servers which slows down processing and causes delayed outputs. This is unacceptable in terms of quality of service of real-time streaming applications, where low-latency is a major requirement. Borealis provides load shedding techniques to make sure that all servers always operate below their processing capacity limits. This is achieved by inserting load reducing drop operators at selected arcs of the query network. Dropped tuples result in approximate answers. Therefore, the main goal in our load shedding algorithms is to minimize the degradation in answer quality [3].

In a distributed stream processing system, each node acts like a workload generator for its downstream nodes. Therefore, resource management decisions at any 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 the chain will collectively determine the quality degradation at the outputs. This makes the problem more challenging than its centralized counterpart [16].

To illustrate, consider the simple query network in Figure 6, with two queries that are distributed onto two processing nodes A and B. Each small box represents a subquery with a certain cost and selectivity. Cost reflects the CPU time that it takes for one tuple to be processed by the subquery, and selectivity represents the 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

[3] In this work, we focus on total query throughput as the quality metric to maximize.

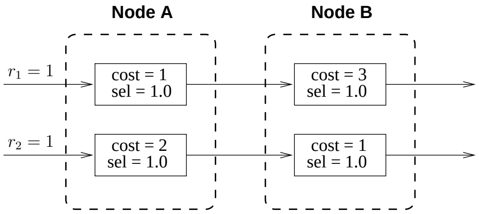


Fig. 6. Motivating example

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 6

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 achieve high-quality query results.

We model the distributed load shedding problem as a linear optimization problem. In our formulation, each server node is represented with a linear load constraint, written in terms of operator costs, selectivities, and input rates. The objective function to maximize is the total output rate at the query end-points, written in terms of operator selectivities and input rates. The drop selectivities (i.e., the fraction of tuples to be kept at the designated drop arcs) appear as the variables in both of these formulas. The goal is to solve the linear program to assign the optimal values to these variables that would satisfy the load constraints on all servers while maximizing the total throughput objective [17].

Our solution to the distributed load shedding problem consists of four steps: (i) advanced planning, (ii) load monitoring, (iii) plan selection, and (iv) plan implementation. In the first step, we precompute a series of load shedding plans for various input rate combinations, each corresponding to an overload condition. The idea is to prepare the system against any potential overload scenario by doing most of the computational work in advance. Next we start periodically measuring the system load. If an overload is detected in one or more of the servers, we select a plan from the previously computed ones and modify the query network according to this plan.

We architect our solution in two alternative ways:

- **Centralized Approach.** In the centralized solution, all load shedding steps are performed at one central server (designated as the "coordinator node") except the plan implementation step. The coordinator contacts all the other servers in order to collect information on their query network topology and run-time statistics (e.g., operator costs and selectivities). Based on the collected global metadata, the coordinator generates a series of load shedding plans for other servers to apply under certain overload conditions. Here, we use the GNU Linear Programming Toolkit (GLPK) [4] to generate the plans. These plans are then uploaded onto the associated servers together with their plan-id's. Then 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.

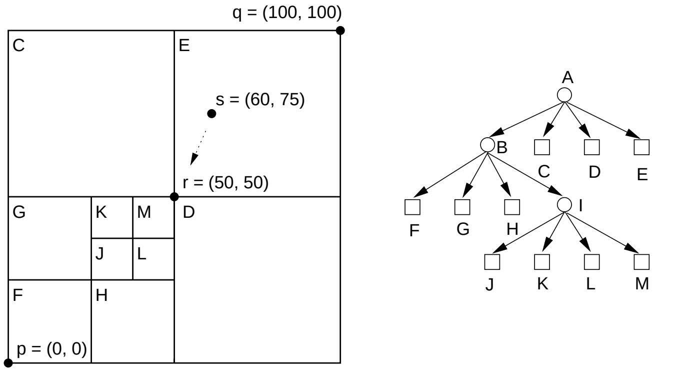
- **Distributed Approach.** In the distributed solution, all four load shedding steps are performed at all of the participating nodes in a cooperative fashion. The collective actions of all the servers result in a globally effective load shedding plan. The neighboring servers coordinate through metadata aggregation and propagation. As a result of this communication, each node identifies what makes a feasible input load for itself and its server subtree, and represents this information in a table that we call the *Feasible Input Table (FIT).* FIT is then propagated to the upstream parent. The parent aggregates the FITs from all of its children, eliminating the table entries that are infeasible for itself. Finally, the parent propagates the resulting FIT to its own parents. This propagation continues until the input nodes receive the FIT for all their downstream nodes. Then using its FIT, a node can shed load for itself and for its descendant nodes.

In the rest of this section, we describe how we perform the advance planning step for the above alternative approaches.

4.1 Solver-based Advance Planning

Our goal in the advance planning step of the centralized, solver-based approach is to produce load shedding plans for a set of infeasible input rate combinations, which will make them feasible for all the servers in the system. The number of such combinations to consider could be potentially very large, and it would be too costly to call the LP solver for each such combination. Instead we use the following, more efficient strategy: We consider a multi-dimensional space of input rates. We systematically search this space to pick a subset of the possible points for which we will call the solver. For the rest of the points, we approximately reuse the solver-generated plans. To be more specific, we assume that an error threshold in quality, ϵ is defined. Given any infeasible point s that lies between two other infeasible points r and q (i.e., $r < s < q$) for which we have already computed the optimal load shedding plan using the solver, if $(q.quality - r.quality) \leq \epsilon * q.quality$, then s can use the plan for r with a minor modification. For example, consider the two-dimensional example in Figure 7(a), each dimension representing an input stream. Assume that $s = (60, 75)$ lies within ϵ-distance from $r = (50, 50)$ and $q = (100, 100)$. Then s can use the plan at r, with the additional modification that input1 and input2 must be reduced by an additional factor of $\frac{50}{60}$ and $\frac{50}{75}$, respectively. Based on this idea, we

[4] http://www.gnu.org/software/glpk/glpk.html



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

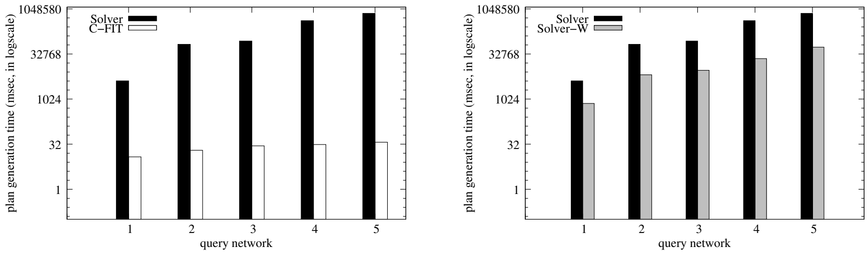
Fig. 7. Quadtree-based space division and index for Solver

take the input rate space and divide it in a region quadtree-like fashion. For each region, the solver is called for the top-most and the bottom-most points. We stop dividing a region either when all points in that region turn out to be within the ϵ bound, or the top-most point is in fact a feasible point. The result of this process is a collection of input rate subspaces with a load shedding plan assigned to each subspace. These subspaces can be very conveniently placed into a quadtree-based index during the space division process described above. For example, Figure 7(b) shows the index that corresponds to the space division of Figure 7(a). At run time, we will use this index to locate the region into which an observed infeasible rate point falls and will use the corresponding load shedding plan.

4.2 FIT-based Advance Planning

Our goal in the advance planning step of the distributed [5], FIT-based approach is to represent a set of feasible input rate combinations (for each server and its subtree) with a table. To briefly summarize, given a node with m inputs, the FIT for this node is a table with $m + 2$ columns. The first m columns represent the rates for the m inputs; the $(m+1)$th column represents the *complementary local load shedding plan* that must be used together with that input entry (this plan may be needed to handle query sharing [17]); and the last column represents the resulting output quality score. Again, for efficiency, we do not want to consider all possible rate combinations. Instead, we use the ϵ threshold as follows: From each input dimension, we pick FIT points that are at most some distance apart, we call this distance "spread" for that dimension. If input dimension i has a maximum feasible rate m_i, then the spread for that dimension can be computed

[5] Although FIT is a distributed algorithm by design, its centralized implementation is also available [17].



(a) Solver and C-FIT

(b) Solver and Solver-W

Fig. 8. Plan generation performance for different query networks

as $\epsilon * m_i$. Next, we must map potential infeasible points to our feasible points in FIT. This means that, if we observe a certain infeasible point q, then we will reduce it to a feasible point p using proper drop values. In general, an infeasible point q that is greater than a feasible point p on all dimensions can be mapped to p. However, we would like to use the best mapping. Our algorithm divides the input rate space accordingly to make sure that this assignment is done to guarantee the highest quality for all infeasible points. The resulting set of subspaces are again placed in a quadtree-based data structure to facilitate search at run-time. Further details about how FIT points are generated and how this table is propagated between neighboring nodes are described in our earlier work [17], [18].

In Figure 8, we show a basic experimental result that compares the load shedding plan generation time for our alternative approaches on five different query networks. These networks differ in the way they apportion the query load across different query paths. In Figure 8(a), we are showing that centralized implementation of FIT outperforms Solver and it is also less sensitive to query load imbalance. The performance difference is mainly due to the time Solver spends in searching the space of infeasible points, while FIT only deals with the feasible points. In Figure 8(b), we compare Solver with its variation Solver-W. Solver-W essentially takes workload knowledge into account and tries to meet a given expected error threshold for the average case. In other words, some infeasible points are known to be less likely than others. Errors of such points contribute less to the average error. Therefore, the algorithm spends less time exploring the corresponding subspaces. The result is some improvement in plan generation performance. Further experimental results can be found in our previous work [17].

5 High-Availability in Borealis

In a distributed stream processing system, servers may fail and this can significantly disrupt or even halt overall stream processing. In case of failures, large amount of transient information may be lost and the servers downstream from a failed one may stop making any progress. Therefore, a distributed stream processing system must provide high-availability (HA) mechanisms that allows processing to continue in spite of server failures. These mechanisms must take correctness (e.g., data loss, duplicates) and performance (e.g., latency introduced during regular processing and during recovery time) requirements of the applications into account.

5.1 Basic HA Models

In Borealis, we define three types of recovery guarantees to address different HA requirements:

- **Precise Recovery.** Post-failure output is exactly the same as the output without failure. Many financial services applications have such strict correctness requirements.
- **Rollback Recovery.** The output produced after a failure is "equivalent" to that of *an execution* without failure, but not necessarily to the output of *the failed execution.* The output may also contain duplicate tuples. Thus, information loss is avoided, but the output can still be imprecise. Event detection applications such as fire alarms, theft prevention are examples.
- **Gap Recovery.** This is the weakest form of recovery where data loss is acceptable for better performance. Sensor-based environment monitoring where recent data is more important is an example.

Each primary server has an associated backup server. A backup server runs its own stream processing engine and has the same query network fragment as its primary, but its state is not necessarily the same as that of the primary. If a primary server fails, its backup server immediately detects the failure and takes over the operation of the failed server.

Borealis provides four recovery approaches that can provide one or more of the above recovery guarantees. These approaches mainly differ in how primary and backup servers prepare for failures. Each approach uses a different combination of redundant processing, checkpointing, and remote logging. As a result, they offer different tradeoffs between run-time overhead and recovery performance.

- **Amnesia.** This approach does not involve any preparation for failures. As soon as the backup server detects that the primary has failed, it restarts the failed query network from an empty state.
- **Passive Standby.** Each primary server periodically checkpoints (i.e., reflects its state updates) to its backup server. The backup server takes over from the latest checkpoint when the primary fails.
- **Active Standby.** The backup server processes all tuples in parallel with its primary. The output tuples of the backup server are not sent downstream; instead they are logged at the output queues. If the primary fails, the backup

takes over by sending the logged tuples to all downstream neighbors and then continuing its processing.

- **Upstream Backup.** Upstream servers preserve tuples in their output queues while their downstream neighbors are still processing them. If a server fails, an empty backup server rebuilds the latest state of the failed primary from the logs kept at the upstream server.

The amnesia approach can only provide gap recovery guarantee, while the other approaches provide rollback recovery in their simplest forms and can be extended to provide precise recovery. In principle, the guarantee of precise recovery requires a higher run-time cost than other weaker recovery guarantees. Furthermore, the query operators may also affect recovery semantics and associated cost requirements. Some Borealis operators are deterministic (i.e., they produce the same output stream every time they start from the same initial state and receive the same input tuples), while others are arbitrary due to dependence on time or arrival order. Thus, deterministic ones are less costly to provide better guarantees [19].

An in-depth algorithmic analysis of all of the above basic HA alternatives together with results from our experimental study showed that each HA approach poses a clear tradeoff between recovery time and processing overhead [19]. In fact, each approach covers a complementary portion of the solution space. To summarize:

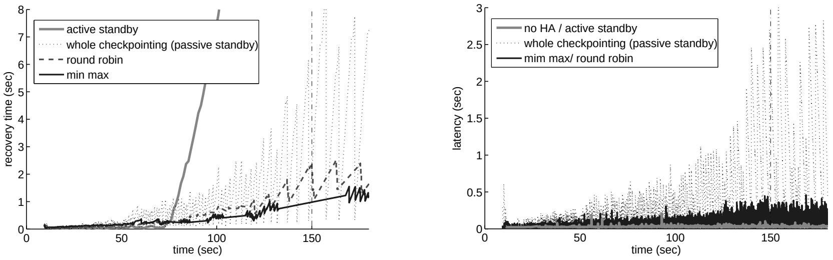
- Active standby has high run-time overhead, but provides very fast recovery.
- Passive standby performs worse than active standby both in terms of recovery time and run-time overhead. However, it is the only approach that easily provides precise recovery for arbitrary query networks. Additionally, it can flexibly trade off between run-time overhead and recovery speed by adjusting the checkpoint interval.
- Upstream backup provides precise recovery for most query networks with the lowest run-time overhead, but at the cost of a longer recovery, depending on the amount of logged data to process during recovery.

5.2 Cooperative and Self-Configuring HA for Server Clusters

A server cluster is a popular form of shared-nothing computing architecture where commodity servers are connected by fast local area networks. Borealis may distribute its processing load onto such a cluster for better scalability. For such environments, we designed and implemented a self-configuring HA approach that enables fast recovery as well as minimal slow-down for regular processing. Unlike our basic HA mechanism described in Section 5.1 where each server is assigned one other backup server, in this case, each server is backed up by multiple servers in a cooperative fashion. Each of these backup servers are in charge of a disjoint query network fragment (called an “HA unit”) of the primary server. Thus, they can take over the failed execution in parallel, which speeds up the recovery time of rebuilding the latest state of the failed server. Furthermore, HA tasks are performed when servers are idle, which reduces the interference with regular stream processing.

In this work, we focused on checkpoint-based passive standby as the recovery approach. This choice is mainly due to the fact that checkpointing works for a larger set of workload and usage scenarios than the other alternatives. Below we briefly summarize the important features of this approach; more technical details can be found in our previous work [20].

- **The HA Mechanism.** Query network on each server is partitioned into HA units. Each such unit is assigned to a different backup server. The preparation for failures involves two HA tasks, namely *capture* and *paste*, to be performed during idle periods. Capture is performed by the primary server, while paste is performed by the backup server. In capture, the primary selects one of its HA units, prepares a checkpoint message for it that includes all the state changes since the last checkpoint, and sends this message to the associated backup server. In paste, the backup selects one of the checkpoint messages that it has received from a primary, copies the message to the corresponding backup image, and notifies the sender primary that the checkpointing request has been completed. Each server is periodically pinged by another designated server. If a failure is detected, then this is broadcast to other servers in the cluster. Each of these notified servers immediately pastes any checkpoint messages from the failed server to the corresponding backup images. Then the execution of these backup images start while the necessary input and output streams are redirected so that stream processing can continue at the backup servers. This HA mechanism provides precise recovery because each backup image can obtain the tuples that the primary has processed since the last checkpoint. This is achieved by keeping output queues at the output of each HA unit to retain those tuples that the downstream backups are currently missing. These output queues are pruned when the downstream server processes them and checkpoints the effect onto the backup server.
- **Checkpoint Scheduling.** A server can be a primary for some HA units and can be a backup for others. Therefore, when it is idle, it can perform either a capture task, or a paste task. The recovery time can be significantly reduced by a careful scheduling of these HA tasks. We developed an algorithm called the "Min-Max Checkpoint Scheduling Algorithm". The idea is to schedule the HA task that would minimize the maximum recovery load among the ones that are in the task queue. This algorithm first finds the best capture task, i.e., the capture of the HA unit with high processing load and low checkpointing cost. Similarly, it finds the best paste task that would help the HA unit with the largest recovery load. Finally, it performs the best task found.
- **Dynamic Backup Assignment.** Assignment of HA units to backup servers can also affect the recovery time. For example, a server which is assigned too many HA units for backup may become a bottleneck. Furthermore, an existing backup assignment may need to be changed with varying system



(a) Recovery time

(b) End-to-end latency

Fig. 9. Performance of min-max checkpoint scheduling

conditions (e.g., changing input rates). Therefore, our approach periodically runs a "Backup Reassignment Algorithm". This algorithm detects *the worst point of failure* (i.e., the server whose failure would cause the longest recovery), and balances its backup load with another server whose failure would cause the shortest recovery.

- **Delta Checkpointing.** HA tasks can be performed more efficiently using operator-specific delta-checkpointing techniques. This is important for stateful operators such as aggregate and join. We use dirty bits for aggregate groups and windows to mark whether they were created after the last checkpoint or not. Dirty windows are fully captured/pasted while others are partially captured/pasted. For join, only the tuples that entered the window after the last checkpoint are captured.

We performed various experiments on the Borealis prototype in order to evaluate the performance of the above techniques [20]. Figure 9 is a basic result that shows how our min-max checkpoint scheduling algorithm effectively reduces the recovery time while being minimally intrusive to regular query processing. In this experiment, 16 aggregates were deployed on each of 5 identical servers, and input stream rates were increased at time point 150 seconds, when each server became around 90% utilized for query processing. Figure 9(a) shows that min-max algorithm provides the fastest recovery even after the system load is increased. Figure 9(b) shows how HA tasks affect query processing performance. Our finer-grained checkpoint technique disrupts processing much less than the standard whole checkpointing approach.

6 Conclusions and Future Work

This paper provides an overview of the Borealis system and three of its features that are key for its scalable and reliable distributed operation. With our resilient operator distribution algorithm, Borealis can withstand high degrees of load without the need for any operator migration. Beyond that, our correlation-based operator distribution algorithm can dynamically balance server loads by taking the relationship between load variations of operators and nodes into account. Our work on distributed load shedding has focused on the load dependency between different servers, and has proposed two alternative solution architectures for removing CPU overload, where scalable coordination between neighboring servers can be achieved in a centralized or a distributed way. Finally, our work on high availability has explored various recovery guarantees and models that may be demanded by different applications, and has shown the existing tradeoffs between performance and correctness. This work has further explored efficient checkpoint-based recovery techniques for server clusters based on cooperation among multiple servers and automatic self-configuration with changing load. All of these algorithms have been implemented and experimentally evaluated on our system prototype. The latest Borealis prototype code can be downloaded from http://www.cs.brown.edu/research/borealis/.

We are currently working on a replication-based stream processing scheme that will provide Borealis with faster and more reliable operation over wide-area networks [21]. Other future work items include support for richer data types (such as video streams) in the form of multi-dimensional arrays, and seamless integration of stream processing with large-scale data collection and dissemination.

Acknowledgements. We thank all members of the Borealis project for their support. This research has been sponsored by the NSF under the grants IIS-0086057 and IIS-0325838.

References

1. Whitney, A.T., Shasha, D.: Lots o' Ticks: Real-Time High Performance Time Series Queries on Billions of Trades and Quotes (Demo). In: ACM SIGMOD Conference, Santa Barbara, CA (2001)
2. Babu, S., Subramanian, L., Widom, J.: A Data Stream Management System for Network Traffic Management. In: ACM Workshop on Network-Related Data Management (NRDM), Santa Barbara, CA (2001)
3. Stefanidis, A., Nittel, S., eds.: Geosensor Networks. CRC Press (2004)
4. Leonhardt, U., Magee, J.: Multi-sensor Location Tracking. In: International Conference on Mobile Computing and Networking (MobiCom), Dallas, TX (1998)
5. Franklin, M.J., Jeffery, S.R., Krishnamurthy, S., Reiss, F., Rizvi, S., Wu, E., Cooper, O., Edakkunni, A., Hong, W.: Design Considerations for High Fan-In Systems: The HiFi Approach. In: CIDR Conference, Asilomar, CA (2005)
6. Shah, M.A., Hellerstein, J.M., Brewer, E.: Highly-Available, Fault-Tolerant, Parallel Dataflows. In: ACM SIGMOD Conference, Paris, France (2004)
7. Abadi, D., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.: The Design of the Borealis Stream Processing Engine. In: CIDR Conference, Asilomar, CA (2005)
8. Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., Seltzer, M.: Network-Aware Operator Placement for Stream-Processing Systems. In: IEEE ICDE Conference, Atlanta, GA (2006)

9. Amini, L., Jain, N., Sehgal, A., Silber, J., Verscheure, O.: Adaptive Control of Extreme-scale Stream Processing Systems. In: IEEE ICDCS Conference, Lisboa, Portugal (2006)
10. Zdonik, S., Stonebraker, M., Cherniack, M., Çetintemel, U., Balazinska, M., Balakrishnan, H.: The Aurora and Medusa Projects. IEEE Data Engineering Bulletin (Special Issue on Data Stream Processing) **26**(1) (2003)
11. Abadi, D., Lindner, W., Madden, S., Schuler, J.: An Integration Framework for Sensor Networks and Data Stream Management Systems (Demo). In: VLDB Conference, Toronto, Canada (2004)
12. Ahmad, Y., Berg, B., Çetintemel, U., Humphrey, M., Hwang, J., Jhingran, A., Maskey, A., Papaemmanouil, O., Rasin, A., Tatbul, N., Xing, W., Xing, Y., Zdonik, S.: Distributed Operation in the Borealis Stream Processing Engine (Demo). In: ACM SIGMOD Conference, Baltimore, MD (2005)
13. Abadi, D., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: A New Model and Architecture for Data Stream Management. VLDB Journal **12**(2) (2003)
14. Xing, Y., Zdonik, S., Hwang, J.H.: Dynamic Load Distribution in the Borealis Stream Processor. In: IEEE ICDE Conference, Tokyo, Japan (2005)
15. Xing, Y., Hwang, J.H., Çetintemel, U., Zdonik, S.: Providing Resiliency to Load Variations in Distributed Stream Processing. In: VLDB Conference, Seoul, Korea (2006)
16. Tatbul, N., Çetintemel, U., Zdonik, S., Cherniack, M., Stonebraker, M.: Load Shedding in a Data Stream Manager. In: VLDB Conference, Berlin, Germany (2003)
17. Tatbul, N., Çetintemel, U., Zdonik, S.: Staying FIT: Scalable Load Shedding Techniques for Distributed Stream Processing. Technical Report CS-06-13, Brown University, Computer Science (2006)
18. Tatbul, N., Zdonik, S.: Dealing with Overload in Distributed Stream Processing Systems. In: IEEE International Workshop on Networking Meets Databases (NetDB), Atlanta, GA (2006)
19. Hwang, J.H., Balazinska, M., Rasin, A., Çetintemel, U., Stonebraker, M., Zdonik, S.: High-Availability Algorithms for Distributed Stream Processing. In: IEEE ICDE Conference, Tokyo, Japan (2005)
20. Hwang, J.H., Xing, Y., Çetintemel, U., Zdonik, S.: A Cooperative, Self-Configuring High-Availability Solution for Stream Processing. In: IEEE ICDE Conference, Istanbul, Turkey (2007)
21. Hwang, J.H., Çetintemel, U., Zdonik, S.: Fast and Reliable Stream Processing over Wide Area Networks. In: IEEE International Workshop on Scalable Stream Processing Systems (SSPS), Istanbul, Turkey (2007)