Job Admission and Resource Allocation in Distributed Streaming Systems

Joel Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, and Kun-Lung Wu

IBM T.J. Watson Research Center, Hawthorne, NY 10532, USA

Abstract. This paper describes a new and novel scheme for job admission and resource allocation employed by the *SODA* scheduler in *System S*. Capable of processing enormous quantities of streaming data, *System S* is a large-scale, distributed stream processing system designed to handle complex applications. The problem of scheduling in distributed, stream-based systems is quite unlike that in more traditional systems. And the requirements for *System S*, in particular, are more stringent than one might expect even in a "standard" stream-based design. For example, in *System S*, the offered load is expected to vastly exceed system capacity. So a careful job admission scheme is essential. The jobs in *System S* are essentially directed graphs, with software "processing elements" (*PE*s) as vertices and data streams as edges connecting the PEs. The jobs themselves are often heavily interconnected. Thus resource allocation of individual PEs must be done carefully in order to balance the flow. We describe the design of the *SODA* scheduler, with particular emphasis on the component, known as *macroQ*, which performs the job admission and resource allocation tasks. We demonstrate by experiments the natural trade-offs between job admission and resource allocation.

1 Introduction

We consider distributed computer systems designed to handle large-scale data stream processing jobs. This area of research is relatively new. Early examples include Borealis, TelegraphCQ, STREAM, StreamBase and Aurora [1,3,5,15,23]. These systems mostly take relational model as a basis. They process voluminous quantities of incoming stream data, performing relational operations on them.

We have been involved in an ambitious project, started in 2003 at the IBM T. J. Watson Research Center, known as *System S* [2, 8, 9, 11, 18, 19]. *System S* is a large-scale, distributed computer system designed to handle complex jobs involving enormous quantities of streaming data. The paradigm is significantly more general than a relational model-oriented streaming environment. A prototype of this system has been built and continues to evolve. The scheduler for *System S* is known as *SODA*.[1] In this paper, we will describe the *SODA* scheduler, focusing particularly on one key mathematical component known as

[1] *SODA* stands for Scheduling Optimizer for Distributed Applications.

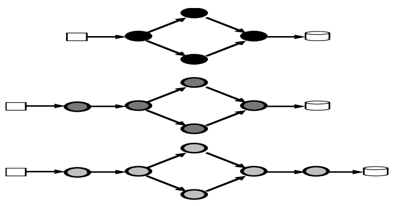
macroQ. Specifically, we will present the detailed mathematical formulation and algorithm of *macroQ*. This component deals with job admission and resource allocation, among other things, and is perhaps the most novel of the four mathematical components in *SODA* in terms of both functionality and design. Note that in a related but substantially different paper [18], we presented an overview of all the *SODA* components without mathematical details, and showed several SODA performance studies with System S applications.

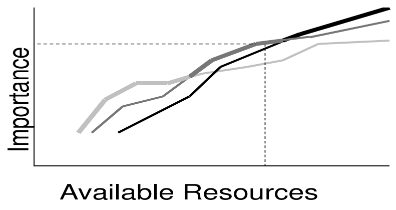
The basic unit of computational work in *System S* is called a *processing element (PE)*. PEs can be arbitrary stream processing software. They are the basic execution containers in *System S*. The PEs are connected via *streams*, which flow from an output port of one PE to an input port of another. The PEs and streams are grouped into *jobs* which represent the basic unit of admittable work in the system. Hence a job is represented by a directed graph. In the current *System S* implementation, each job consists of one or more alternative data flow graphs called *templates*. These templates could be provided to *SODA* by the application developers. Each template represents a different implementation of the same job, perhaps to achieve a different level of solution quality. The logical nodes in a given template correspond to PEs, and the directed arcs correspond to streams. PEs thus both consume and produce streams.

The *macroQ* module of *SODA* in *System S* provides three critical functions, namely job admission, template selection and resource allocation. We will define a new notion, known as *importance*, which can be thought of intuitively as a function which measures the "benefit" produced by a particular job template when allocated a specific amount of processing resources. The *macroQ* module will attempt to maximize the total importance across all the competing job templates by optimizing importance as a function of allocated resources. For those jobs that get admitted it will choose a template alternative and an amount of resources to be allocted to the PEs in that template. Even though job admission and resource allocation in themselves are not new, they are significantly more challenging in the context of a distributed streaming system. In particular, *System S* has even more stringent requirements than a standard streaming system, and the offered load is expected to vastly exceed system capacity. Our main contribution is a novel approach that combines these three critical decisions in a unified framework. We elaborate on these issues in the next few paragraphs.

A key design assumption of *Systems S* is that there will be too much work. Thus *System S* hardware must be made to run at nearly full capacity nearly all of the time. This includes the processing nodes, which must be utilized as completely as possible. Moreover, some work simply will not fit. It is the role of the *macroQ* module to make intelligent decisions about job admission. To the best of our knowledge, there are no other schedulers implemented in any stream processing system that consider job admission. (In [20], the authors describe a scheduler for a hypothetical system with a simplified model of stream processing.) Some schedulers [13, 16] perform load shedding to deal with dynamically overloaded processing nodes, but this is an inherently different concept.

A second role of the *System S* scheduler is to choose one of the alternative templates for each admitted job. Fig. 1(a) shows a job with three such templates. All the nodes in these data flow graphs are PEs, as noted above, except for the following left and right edge conventions: The left-hand node in each case is a dummy node, which may be used to "connect" this job to another. The right-hand node in each template represents disk storage, which can be regarded as a second type of dummy connector node, for persisted data. Both the left- and right-hand side streams are required by *SODA* to "match" in all templates, as are the dummy nodes, so that the identical inter-job connections may take place regardless of the template chosen.


(a) Templates



(b) Importance

Fig. 1. Template Alternatives

The example in Fig. 1(a) is one possible scenario which results in multiple templates for a job. In this case, the first template would provide the basic job functionality, consisting of 4 PEs. The second template adds a preprocessing PE to achieve a higher quality of solution. The third further adds a post-processing PE, to achieve an even higher quality of solution. But there is a correlation between the benefit of doing a job and the total resources allocated to it. Each alternative template will provide the greatest overall benefit within some range of total resources allocated to the job. In *SODA* the notion of importance, defined formally in the next section, is used to quantify benefit. Fig. 1(b) illustrates importance as a function of allocated resources for the three job templates. In this case, at high allocated resource levels the third template dominates. (At this level, sufficient resources are available for both the preprocessing and post-processing PEs.) At medium allocated resource levels the second template dominates, and at low allocated resource levels the first template dominates.

A third function of the scheduler in *System S* is resource allocation of the PEs in the various accepted jobs. It is the interconnected (producer/consumer) nature of these PEs, potentially even across jobs, that makes this problem difficult: Flow imbalances can lead on one hand to buffer overflows (and loss of data), and on the other to under-utilization of processing nodes. The resources allocated to a PE which produces a stream affect the resources required for the PE(s) that consume that stream. The *macroQ* module optimizes and flow balances the *amount* of processing resources allocated to each PE in the jobs that are admitted.

It is the role of other *SODA* modules to take these processing goals and fractionally assign each PE to one or more acceptable processing nodes [18]. Thus the problem of determining *quantity* of PE resource allocations is effectively decoupled from the problem of determining *where* the PEs should be executed. In assigning the PEs in the chosen templates of the accepted jobs to processing nodes, there is a trade-off between the load on the processing nodes and stream traffic on the network. Assigning two PEs connected by a stream to the same processing node eliminates the contribution of that stream to network traffic, but may contribute instead to overloading the processing node. So *SODA* attempts to achieve a balanced placement that does not overload either network links or node capacities. In fact, it attempts to minimize a weighted average of six separate metrics associated with processing loads on the nodes and traffic on the network links. The assignment problem is made more complex by the addition of many special constraints imposed by *System S*. These include, among many others, hardware constraints for certain PEs and nodes (*resource matching*), security and license constraints, constraints that pairs of PEs be placed together (*colocation*), or that pairs of PEs be placed on distinct nodes (*ex-location*). Of course, many PEs may share a node. *SODA* attempts to provide each PE with a fraction of the processing power of any node to which it assigned, matching as closely as possible the overall PE flow balancing goals already computed.

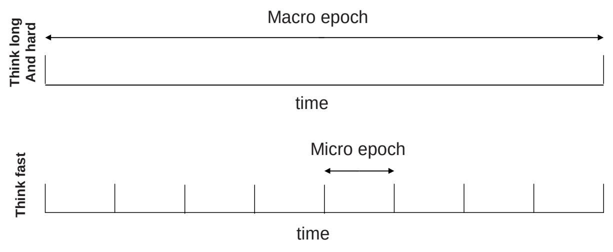


Fig. 2. Macro and micro epochs.

Finally, in order to react quickly in a highly dynamic environment, *SODA* is an *epoch*-based scheduler. There are two kinds of epochs: macro epochs and micro epochs. Each macro epoch contains several micro epochs. Fig. 2 shows the temporal hierarchy between macro and micro epochs.[2] The *macroQ* component operates at the macro epoch level, and hence we focus mainly on macro epochs in this paper. At the beginning of each epoch, *SODA* obtains as input a snapshot of the current system state, including the jobs running on the system and the jobs waiting to be admitted. It then computes for most of an epoch, finally outputting its scheduling decisions at the end of the epoch. That is, it produces a list of accepted and rejected jobs. For the accepted jobs it produces a choice

[2] For reasonably sized *System S* installations the macro and micro epochs can also be solved sequentially.

of templates and a set of fractional allocations of the PEs to processing nodes. Those decisions are enforced by *System S* during the following epoch, and the entire process repeats indefinitely. Epoch lengths are a *SODA* settable parameter, but macro epochs on the order of a minute are typical. This is a reasonable compromise between the staleness of the input data and the time required for the mathematical components of *SODA* to make high quality decisions. Each macro epoch usually corresponds to five micro epochs, allowing *SODA* to respond quickly to changes in system.

Our contributions can be summarized as follows:

1. We provide the first stream processing scheduler in a working system that performs *job admission.* The choice to admit a job will depend on whether or not the optimal total importance occurs when that job is allocated a positive amount of resources, given certain natural constraints.
2. We provide a systematic way of optimally choosing one of several job *templates* for newly admitted jobs. The choice will depend on the relative importance of work which can be produced by these templates as well as that of the other potential work in the system.
3. We provide flow balanced resource allocations for each of the PEs in the chosen templates of accepted jobs, while simultaneously optimizing the overall allocation of resources in the system. In other words, each admitted job will get an appropriate total amount of resources based on its contribution to overall importance, and the PEs within that job will be allocated those resources in a balanced manner. Given the highly interconnected nature of the data flow graphs this is a difficult optimization problem.
4. We provide appropriate constructs to allow the scheduler to react quickly and intelligently to dynamic changes in the system, including the arrival and departure of jobs, nodes going up and down, and also changes in the relative importance of the work in the system.
5. We have designed a real-time scheduler which makes complicated decisions in each epoch, using algorithms that are deadline-aware.

The remainder of this paper is organized as follows. Section 2 contains preliminaries, including a glossary of new terms used by *SODA*, and by *macroQ* in particular. Section 3 contains an overview of the *SODA* scheduler itself, describing each of the four major mathematical components. In Section 4 we give the description, formulation and the solution approach to *macroQ*. (We are focusing here on *macroQ* because of its novelty, but also because of space limitations: A very complete description of all of the *SODA* components, associated infrastructure components and many other *SODA* capabilities is available [17].) Section 5 describes experiments showing the natural trade-offs associated with job admission and resource allocation. (We should point out that while alternative template selection is a current *macroQ* feature, the *System S* infrastructure does not yet support multiple templates. So we do not provide experiments illustrating this feature in the current paper.) Section 6 describes related work. Finally, Section 7 gives conclusions.

2 Preliminaries

In *macroQ*, we use a number of terms that have very specific meanings to the scheduler. We list these below, with explicit definitions. These concepts are critical to the discussions that follow. The first two items, the *value function* and *weight*, are the key components of the third item, *importance*. Roughly speaking, value functions measure benefit. Weights are used in their traditional sense as multiplicative "knobs", in this case accentuating or decentuating value. The product of the two is importance. Importance, in turn, is the metric that *macroQ* tries to maximize. The fourth item, the *resource function (RF)*, is essentially the means by which we iteratively compute this notion of importance. Finally, *rank*, the fifth item, is an orthogonal notion to importance. It is a priority metric assigned to each job. Jobs which produce little importance but have a better rank may get admitted *instead* of jobs which have more importance but have a worse rank. Some of these terms are not new in themselves, but the combination of them is novel.

1. *Value function:* Each derived stream produced by a potential *System S* job has a *value function* associated with it. The domain of this function might typically be the projected rate of the stream. Or it might instead be a stream quality measure, such as projected goodput. In theory it could be a cross product of a variety of quantity, quality and even other measures of benefit. The definition is intentionally general, though early *SODA* instances have employed simple rate-based value functions. Also note that value functions which are 0 everywhere will typically predominate: Although the notion is also intentionally general we expect to see non-trivial value functions mostly on terminal streams of various jobs. These are, of course, the "end products" of *System S* work, and one would thus naturally want to measure benefit there.
2. *Weight:* Each derived stream produced by a potential *System S* job also has a *weight* associated with it. Non-trivial weights will also typically be quite sparse, since we will see that the weight may as well be 0 unless the stream also has a non-zero value function.
3. *Importance:* Each derived stream produced by a potential *System S* job has an *importance* which is the product of the weight and the value function. Importance is therefore a function of the rate or quality of the stream, which in turn depends on the resources allocated to all the upstream PEs – in other words, those PEs which help to produce the stream. The summation of this importance over all derived streams is the *overall importance* being produced by *System S*, and this is what *macroQ* attempts to maximize. (Again, a large majority of streams will typically not contribute to this importance metric.) Consider Fig. 3, representing the flow graph of the same job in scenarios involving two different sets of weights. In Fig. 3(a), positive weights are at all the terminal, "starred" streams. But in Fig. 3(b) the second weight has been eliminated (changed to 0). It follows that the 2 PEs immediately upstream of that weight cannot do work which contributes to overall importance. *SODA*

will therefore not allocate resources to them. (Other PEs, further upstream, do useful work in support of streams with positive weights. They may get fewer resources than they would in the upper half, of course.) Weights are thus a multiplier knob to turn on and off portions of a job and, more generally, a simple way to adjust relative importance.

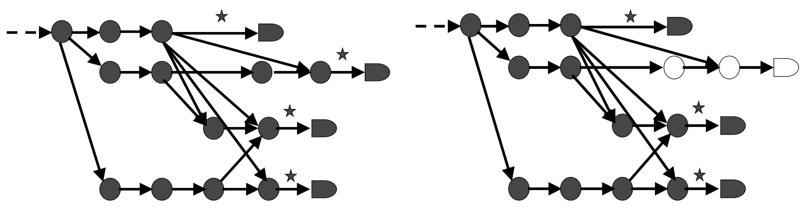
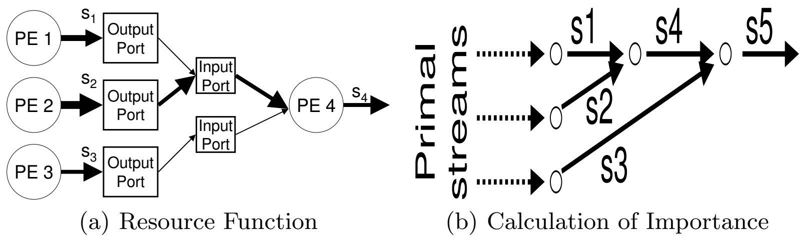


Fig. 3. Varying the Weights

4. *Resource function:*[3] If importance is the metric to be maximized, the natural question is how to compute it. The first part of the answer is as follows: Each derived stream s in *System S* (and by approximate terminology the PE that produces that stream) has an *RF* associated with it. The *RF* is multidimensional. If there are N input streams to the producer PE, then the *RF* has $N + 1$ input parameters. There is one parameter for each of the input streams, each with the same domain as the value function. The final input dimension is the (computational) resources which may be allocated to the PE, in *millions of instructions per second (MIPS).* The output of this function for stream s is again in terms of the same domain. See, for example, Fig. 4(a). Assuming the domain to be rate-based, the *RF* for stream s_4 takes 4 parameters as input. The first three are the rates of streams s_1 through s_3, and the fourth is the MIPS allocated to PE 4. The output is the rate of stream s_4. (Some details are hinted at in the figure. Output ports filter the streams, and the output from PEs 1 and 2 are aggregated into the first input port, effectively decreasing the dimensionality of this *RF* by one.) The *RF* needs to be "learned" over time by a *SODA* infrastructure component known as the *Resource Function Learner (RFL).* The second part of computing importance involves iteratively traversing the data flow graphs from "left" to "right", ending in a final value function calculation. Consider Fig. 4(b). By topologically sorting [7] a directed acyclic graph, we can apply ready list scheduling [4, 6] to compute the importance for stream s_5. In the figure three *RF*s are initially ready because they are fed by primal (external) streams. So we obtain the rates at streams s_1 through s_3. One additional *RF* becomes ready in each of the next two steps (because their inputs have been computed), and we obtain the rates at streams s_4 and s_5

[3] A paper about these *RF*s is forthcoming.

in succession. Finally we apply the weighted value function at s_5 to obtain importance. (*SODA* can also handle data flow graphs with cycles, but we omit details for that case.)



(a) Resource Function (b) Calculation of Importance

Fig. 4. Using Resource Functions to Calculate Importance

5. *Rank:* Each job in *System S* has a *rank*, a positive integer which is used to determine whether the job should be run at all. A lower job rank is better than a higher one. (There are two seemingly irreconcilable camps on the issue of whether rank should improve with value or the reverse. Our motivation in using the convention we chose is twofold: First, it is common to say that something is "priority one", meaning it is most important. Second, one is inarguably the smallest positive integer, and thus we definitively will know that a job with rank one is most essential. On the other hand, it is certainly true that adopting this definition causes rank to be inversely related to the assigned rank number.)

   The rank of a job is set by a separate, independent component in *System S* based on a set of criteria, beyond the scope of this paper. The importance, on the other hand, determines the amount of resources to be allocated to each job that will be run. A lower job rank is better than a higher one. We will shortly provide a notion of *rank-legality* which will describe the possible subsets of jobs that can be admitted into *System S*. There is a specific job rank for which the following holds: All jobs with lower ranks are admitted, and all jobs with higher ranks are not admitted. Jobs with that rank may or may not be admitted, depending on the available resources and the importance associated with the (streams of the) jobs themselves. We call this property *rank-legality.* (This statement is a slight simplification, since one needs to account for inter-job dependencies. We will describe this notion of *revised rank* shortly.) Fig. 5 illustrates job admission based on revised rank. The *waterline* (that is, the cardinality of the highest admitted job rank) goes up in the case of lighter load conditions or with more processing power in the system. It goes down in heavier load conditions or with less processing power.

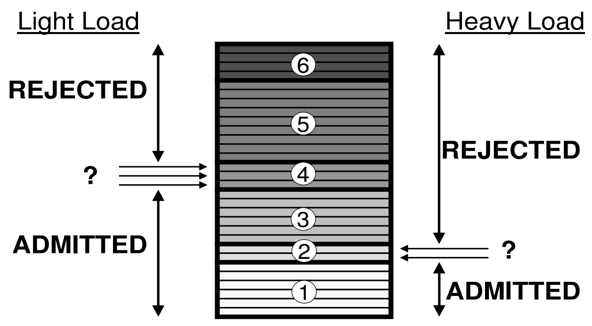


Fig. 5. Job Admission as a Function of Rank

3 Overview of *SODA*

In this section we describe the four major mathematical components of *SODA*. We describe the solution approaches and motivate them from a practical standpoint, emphasizing how the solutions are dictated and/or guided by the *SODA* design philosophy.

To make the scheduling problem tractable, each *SODA* epoch is divided into four mathematical phases. Each of the four phases corresponds to a mathematical optimization module. The first two phases are known collectively as the *macro* model, while the second two are known as the *micro* model. The two temporally hierarchical levels and their goals are:

– *The macro model*, which chooses the jobs that will be admitted, the templates for those jobs, and the candidate nodes to which the PEs in those jobs and templates can be assigned. The choices made in the macro model are respected by the micro model during the micro epochs of the *next* macro epoch, making the decisions of the micro model easier and more effective.
– *The micro model*, which chooses the fractional allocations of the PEs in the jobs and templates that have been chosen by the macro model. Fractional allocations of PEs are 0 for a particular node *unless* that node has been chosen as a candidate node by the macro model. The micro model handles dynamic variability in the relative importance of work (via revised weights), and changes in the state of the system (via nodes and PEs that go up or down), without having to consider the difficult constraints handled in the macro model.

This decomposition is not perfect. Periodically there could be solutions from the macro model which are inconsistent with the constraints of the micro model. A "micro to macro" feedback loop would seem to be useful, but we have not seen examples where it is needed in practice.

Now we describe the individual decoupled quantity and where components for both the macro and micro models:

- *macroQ*, the *macro quantity* model, maximizes projected importance by deciding which jobs to admit, which templates to choose, and by computing flow balanced PE processing allocation goals (in MIPS), subject to job rank-legality, required jobs, minimum and maximum MIPS constraints. We describe *macroQ* in the next section.
- *macroW*, the *macro where* model, minimizes projected network traffic and load balances the nodes, allocating uniformly more candidate nodes than PE goals dictate, subject to resource matching, specialized hardware, security, licensing, memory, PE exclusivity, maximum PEs per node, maximum degrees of parallelism for each PE, fixed PEs, mutual PE exclusion and colocation, legal fractional allocations and various incremental movement limits. It optimizes a weighted average of six separate metrics, three of which are averages of the utilizations of the nodes, the traffic in the network, and the bandwidth in and out of the nodes. The other three are maximum values on these same components. The overallocation allows more flexibility in handling micro epoch dynamics.
- *microQ*, the *micro quantity* model, maximizes projected importance, computing more accurate MIPS allocation goals for the PEs than those of *macroQ* by taking the candidate nodes into account. (Recall that *macroQ* does not know this information.) It also deals with revisions due to changes in node states, PE states and the like.
- *microW*, the *micro where* model, minimizes the differences between the goals output by *microQ* and achieved fractional allocations subject to various constraints on incremental movement and node changes, fixed PEs and so on. The output is thus a set of fractional assignments of the PEs to the nodes whose sum across all nodes is as close to the allocation goals as possible.

4 *MacroQ* Algorithm

The *macro quantity* model, *macroQ*, finds a set of jobs to admit during the next macro epoch. For each job it chooses a template from among the options given to it. Each template represents an alternative plan for performing the job. The jobs have ranks, and the jobs that are chosen by *macroQ* must respect a rank-legality constraint. Required jobs must be admitted. (Without loss of generality we can assume required jobs have rank 1.) Minimum and maximum PE MIPS constraints must also be respected. The goal of the *macroQ* model is to maximize the projected importance of the streams produced by the winning jobs and templates. In the process of solving the problem *macroQ* computes the optimal importance, the list of job and template choices, and finally the set of processing power goals (measured in MIPS) for each of the PEs within the chosen list. We formalize this below.

The problem formulation and the algorithm in *macroQ* are fairly elaborate. For the reader's convenience, Table 4 provides a summary of notation used, in order of appearance. And Fig. 4 provides a summary of the *macroQ* pseudo-code. Note that there are basically three nested loops.

- The outer loop, from line 3 to line 26, considers different levels of resolution granularity for the resource allocation problems that will be solved. A coarse level of granularity provides a quick solution, while a fine level provides an accurate solution. Because *SODA* is a real-time scheduler, *macroQ* must have a solution by the time the macro epoch completes. A quick, coarse solution will serve this purpose.
- The middle loop, from line 5 to line 24, decrements the possible revised rank waterlines, considering less and less jobs as it goes.
- The inner loop, from line 7 to line 23, is a divide and conquer approach based on the number of so-called *weak components* of the relevant data flow graphs. The overall resource allocation problem to be solved can be handled by solving an elaborate problem on each weak component, and then combining the solutions via a simpler problem across *all* components. We will describe these in more detail later in this section. problems are solved as we go along.

Ultimately we output the best solution discovered, on line 27.

```
 1: Set $OPT = \infty$
 2: Set $OK$ = false
 3: while $OK$=false do
 4:    Pick resolution granularity $\bar{G}$
 5:    for $r = R$ to 1 by -1 do
 6:       Create list $(\mathcal{J},T)_1, ..., (\mathcal{J},T)_{L_r}$ of rank-legal job/templates with waterline $r$
 7:       for $l = 1$ to $l = L_r$ do
 8:          Compute $C_{(\mathcal{J},T)}$ weak components
 9:          for $c = 0$ to $c = C_{(\mathcal{J},T)} - 1$ do
10:             NSDP scheme to solve component $c$ RAP with granularity $\bar{G}$
11:          end for
12:          Compute number of components with concave importance functions
13:          if all are concave then
14:             Galil-Megiddo scheme to solve inter-component RAP with granularity $\bar{G}$
15:          else if none are concave then
16:             DP scheme to solve inter-component RAP with granularity $\bar{G}$
17:          else
18:             Fox/DP scheme to solve inter-component RAP with granularity $\bar{G}$
19:          end if
20:          if $\mathcal{I} > \mathcal{OPT}$ then
21:             OPT=$\mathcal{I}$
22:          end if
23:       end for
24:    end for
25:    Evaluate $OK$
26: end while
27: Output $OPT$
```

Fig. 6. *macroQ* Pseudocode

Variable	Definition
J	Number of jobs being considered
$\pi(j)$	Original rank of job j
N_j	Number of templates for job j
$\mathcal{J}$	Job list
T	Template list
$\mathcal{T}_\mathcal{J}$	Set of all template lists for job list $\mathcal{J}$
$\mathcal{L}$	Job/template list
$D_{(\mathcal{J},T)}$	Directed acyclic graph associated with job/template pair $(\mathcal{J}, T)$
$\mathcal{P}_{(\mathcal{J},T)}$	Nodes (PEs) in $D_{(\mathcal{J},T)}$
$d_{(\mathcal{J},T)}$	Asymmetric distance function
$\mathcal{D}_{(\mathcal{J},T)}(p)$	Set of PEs which depend on PE p
$\pi_{(\mathcal{J},T)}(j)$	Revised rank for job j
$\hat{\mathcal{L}}$	Rank-legal job/template list
V_s	Composite value function for stream s
w_s	Weight for stream s
I_s	Importance function for stream s
G	Total MIPS in system
m_p	Minimum MIPS for PE p if admitted
M_p	Maximum MIPS for PE p if admitted
$hat\mathcal{J}$	Jobs that must be admitted
$\bar{G}$	Number of resource units in discrete RAP
$\bar{m}_p$	Minimum resource units for PE p if admitted
$\bar{M}_p$	Maximum resource units for PE p if admitted
$C_(\mathcal{J}, T)$	Number of weak components for job/template list $(\mathcal{J}, T)$
$\mathcal{I}_c$	Importance function for weak component c
$\bar{m}_c$	Minimum resource units for weak component c
$\bar{M}_c$	Maximum resource units for weak component c
L_r	Number of job/template alternatives examined of revised rank r

Table 1. Key *macroQ* Notation

4.1 Notation

Let J denote the number of jobs being considered, indexed by j. Each job j has a *rank* $\pi(j) \in \mathbb{N}$. (Here, $\mathbb{N}$ represents the natural numbers.) We adopt the (slightly unnatural) convention that lower numbers indicate better ranks. Thus the best possible rank is 1. Each job j comes with a small number of possible job *templates*. This number may be 1. It *will* be 1 if the job has already been instantiated, because we assume that the choice of a template is fixed throughout the "lifetime" of a job. It is, however, the role of the *macroQ* model to make this choice for jobs that are newly admitted. Let N_j denote the number of templates for job j, indexed by t.

Any subset $\mathcal{J} \in 2^J$ will be called a *job list*. For each job list $\mathcal{J}$ a function $T : \mathcal{J} \to \mathbb{N}$ satisfying $T(j) \leq N_j$ for all j will be called a *template list*. Denote the set of all template lists for $\mathcal{J}$ by $\mathcal{T}_\mathcal{J}$. Finally, define the *job/template list* to be the set $\mathcal{L} = \{(\mathcal{J}, T) | \mathcal{J} \in 2^J, T \in \mathcal{T}_\mathcal{J}\}$. A major function of *macroQ* is to make a "legal and optimal" choice of a job/template list.

We will make the assumption, for ease of exposition, that no cycles exist in the directed flow graphs for a job and template choice. *SODA* can actually handle intra- and inter-job cycles, but the details are somewhat complex.

Each job/template list $(\mathcal{J}, T)$ gives rise to a directed acyclic graph $D_{(\mathcal{J},T)}$ whose nodes $\mathcal{P}_{(\mathcal{J},T)}$ are the PEs in the template and whose directed arcs are the streams. (This digraph is "glued" together from the templates of the various jobs in the list, and we omit the exact details. These PE nodes may come from multiple jobs.) Assigning length one to each of the directed arcs, there is an obvious notion of an asymmetric distance function $d_{(\mathcal{J},T)}$ between pairs of relevant PEs. Note that $d_{(\mathcal{J},T)}(p,q) < \infty$ means that PE p precedes PE q, or, equivalently, that q depends on p. Let $\mathcal{D}_{(\mathcal{J},T)}(p)$ denote the set of PEs $q \in D_{(\mathcal{J},T)}$ for which q depends on p. This notion of dependence gives rise, in turn, to the notion of dependence between the relevant jobs: Given jobs $j, j' \in \mathcal{J}$, we will say that j' depends on j provided there exist PEs q and p, belonging to j' and j, respectively, for which $d_{(\mathcal{J},T)}(p,q) < \infty$. Let $\mathcal{D}_{(\mathcal{J},T)}(j)$ denote the set of jobs $j' \in \mathcal{J}$ for which j' depends on j.

We now define a revised job rank notion based on a particular job/template list $(\mathcal{J}, T)$ by setting

$$\pi_{(\mathcal{J},T)}(j) = \begin{cases} \min_{j' \in \mathcal{D}_{(\mathcal{J},T)}(j)} \pi(j') & \text{if } j \in \mathcal{J} \\ \pi(j) & \text{otherwise.} \end{cases}$$

This is well-defined. We can define the notion of a *rank-legal* job/template list $(\mathcal{J}, T)$ as follows: We insist that $j \in \mathcal{J}$ and $j' \notin \mathcal{J}$ implies that $\pi_{(\mathcal{J},T)}(j) \leq \pi_{(\mathcal{J},T)}(j')$. (This is equivalent to the statement that there is a value for which all jobs with lower revised ranks will be admitted and all jobs with higher revised ranks will not be admitted.) Let $\hat{\mathcal{L}}$ denote the set of rank-legal job/template lists.

Define the decision variable g_p to be the resource allocation, in MIPS, given to PE p. As noted, any derived stream s associated with job/template list $(\mathcal{J}, T)$ has a *value function*. The stream, in turn, is created by a unique PE p associated with $(\mathcal{J}, T)$. The PE p gives rise to a set $\{q_1, ..., q_{k_p}\}$ of k_p PEs q_i for which $p \in \mathcal{D}_{(\mathcal{J},T)}(q_i)$. This set includes p itself. We have also introduced the notions of learned *RFs* which can be iteratively composed to create a function from the processing power tuple $(g_{q_1}, ..., g_{q_{k_p}})$ to the domain of the value function. And so the composition of these recursively unfolded functions with the value function yields a mapping V_s from the tuple $(g_{q_1}, ..., g_{q_{k_p}})$ to the non-negative real numbers for stream s. This function is called the *composite value function* for s. Multiplied by a weight w_s for stream s it becomes a *stream importance* function I_s mapping $(g_{q_1}, ..., g_{q_{k_p}})$ to the non-negative real numbers $[0, \infty)$. Finally, aggregating all the stream importance functions together for all streams which are created by a given PE p yields a *PE importance function* $\mathcal{I}_p$.

Let G denote the total amount of *System S* processing power, in MIPS. Let m_p denote the minimum amount of processing power which can be given to PE p if it is admitted, and M_p denote the maximum amount of processing power which can be given to PE p if it is admitted. Suppose that the set $\hat{\mathcal{J}}$ represents the jobs that must be admitted.

4.2 Mathematical Formulation

We seek to maximize the overall importance, which is the sum of the PE importance functions across all possible rank-legal job/template lists. The objective is therefore to find

$$\max_{(\mathcal{J},T)\in\hat{\mathcal{L}}} \sum_{p\in\mathcal{P}_{(\mathcal{J},T)}} \mathcal{I}_p(g_{q_1},...,g_{q_{k_p}})$$

subject to the following constraints:

$$\sum_{p\in\mathcal{P}_{(\mathcal{J},T)}} g_p \leq G, \tag{1}$$

$$m_p \leq g_p \leq M_p \qquad \forall p \in \mathcal{P}_{(\mathcal{J},T)}, \tag{2}$$

$$\hat{\mathcal{J}} \subseteq \mathcal{J} \tag{3}$$

Constraint 1 is the resource allocation constraint. It ensures that all of the resource is used if it is useful and possible to do so. Constraint 2 requires a PE p to be within some minimum and maximum range if it is admitted. Constraint 3 insists that required jobs are admitted.

4.3 Solution Approach

We discretize the above continuous resource allocation problem by dividing the total amount of resource G into $\bar{G}$ equal size atomic units of "resolution" $G/\bar{G}$ MIPS each. Assume that this value $\bar{G}$ is given. For each PE p let $\bar{m}_p = \lfloor m_p\bar{G}/G\rfloor$ and $\bar{M}_p = \lceil M_p\bar{G}/G\rceil$ represent the discrete analogues of the minimum and maximum MIPS constraint terms. Also assume a fixed rank-legal job/template list $(\mathcal{J},T) \in \hat{\mathcal{L}}$ containing all the required jobs $\mathcal{J}$. Partition the PEs and streams into $C(\mathcal{J},T)$ weak components and fix one such component c.

We consider, using the natural change in notation, the corresponding discrete resource allocation problem of maximizing $\sum_{p\in c}\mathcal{I}_p(\bar{g}_{q_1},...,\bar{g}_{q_{k_p}})$ subject to the constraints $\sum_{p\in c}\bar{g}_p \leq \bar{G}$ and $\bar{m}_p \leq \bar{g}_p \leq \bar{M}_p$ for all $p \in c$. This problem can be solved by a scheme known as *Non-Serial Dynamic Programming (NSDP)* [10]. NSDP is a complex dynamic programming scheme designed specifically to handle difficult (non-separable) resource allocation problems. (See line 10 of Fig. 4.) As part of the solution methodology we obtain the optimal values $\mathcal{I}_c(\bar{g}_c)$ for every $\bar{g}_c$ between 1 and $\bar{G}$, *as well as* the PE MIPS allocations that constitute this optimal solution. We can thus regard $\mathcal{I}_c$ as a *component importance function* of the resources $\bar{g}_c$ allotted to component c. Set $\bar{m}_c = \sum_{p\in c}\bar{m}_p$ and $\bar{M}_c = \sum_{p\in c}\bar{M}_p$.

Note that the objective function can be regarded as a "black box", calculated by iterative *RF* compositions followed by a weighted value function calculation. To make this as efficient as possible the *macroQ* code has itself been carefully optimized. Careful analyses are performed to determine which sub-graph calculations are strictly necessary and which are redundant. A cache of previous results is also employed. Also, *macroQ* code is aware of time and is given a deadline by *SODA*. So it occasionally takes "shortcuts", using a partially greedy scheme instead of a full NSDP algorithm. This fits the design philosophy: *SODA* is a *real-time* scheduler.

Having performed this NSDP on *each* component we now consider the problem of optimizing over *all* components. The good news here is that the problem is a *separable* resource allocation problem: We wish to maximize $\sum_c \mathcal{I}_c(\bar{g}_c)$ such that $\sum_c \bar{g}_c \le \bar{G}$ and $\bar{m}_c \le \bar{g}_c \le \bar{M}_c$ for all c. Separability here means that each summand is a function of a single decision variable, and such resource allocation problems are inherently easier to solve.

In fact, if the component importance functions happen to be *concave* the problem can be solved by one of three algorithms: These are the schemes by Fox, Galil and Megiddo, and Frederickson and Johnson, which can be regarded as fast, faster and (theoretically) fastest, respectively. If the component importance functions, on the other hand, are not concave, the problem may still be solved by dynamic programming. See [10] for details on all of these algorithms. Also see lines 14, 16 and 18 of Fig. 4.

It turns out that concavity is not an uncommon condition for our component importance functions. So we test each component for concavity and adopt one of three approaches, depending on the results.

- If all component importance functions are concave we solve the resource allocation problem by the Galil and Megiddo algorithm. This algorithm is quite fast and easier to code than the Fredrickson and Johnson scheme.
- If all the component importance functions are not concave we solve the resource allocation problem by dynamic programming.
- In other cases we solve the concave portion of the problem by the Fox algorithm (because it provides the needed intermediate values) and then solve the remainder of the problem by dynamic programming.

At the end of this step we have computed the optimal MIPS allocations for each PE. But this can be regarded as just the inner loop of a three step nested process. In the central loop we evaluate all rank-legal templates. In the outer loop we evaluate successively finer resolution granularities. Again, see Fig. 4.

The evaluation of all rank-legal templates is obviously exponential [7] in nature, though the problem is generally not large: *SODA* only evaluates alternative for *new* jobs. Once a template decision has been reached it lasts for the remaining epochs of the job. And most jobs, in fact, only have a single template. The rank-legality constraints adds another exponential term, but this process can also be streamlined if time is an issue. The code loops through each rank value, working from the highest rank (R) to lowest rank (1): For any given rank value it assumes all higher revised rank jobs will not be admitted, all lower revised rank jobs will be admitted, and has to decide which jobs of that revised rank will be admitted. For all but the highest revised rank these jobs were admitted in the previous calculation. The code computes their importance divided by

%	Rank 1							Rank 2						Rank 3					
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
100	786	882	462	540	558	318	336	786	870	462	702	264	366	786	870	516	558	294	336
95	780	840	462	540	558	318	336	780	780	462	696	264	366	780	780	516	558	294	336
90	786	792	462	540	558	318	336	780	780	462	612	264	366	696	780	516	558	294	
85	696	792	372	540	468	318	318	696	780	372	612	264	366	696	780	516	468	294	
80	690	786	366	534	462	312	312	690	780	366	606	258	366	696	774	510		288	
75	684	780	366	534	354	270	312	626	768	366	606	258	360	684	774	510			
70	690	786	366	534	444	312	312	690	774	366	606	258	360	690		510			
65	696	792	372	540	468	318	336	696	780	390	612	264	366			516			
60	696	792	372	540	468	318	324	696	780	372	612	264	366						
55	696	792	462	540	552	318	336	696	780		612	264							
50	690	786	366	534	426	312	312	692	774		606								
45	690	792	366	534	462	318	312	696	780										
40	696	792	462	540	558	318	336	696											
35	626	710	354	494	426	300	314	626											
30	510	558	344	426	354	272	292	544											
25	510	558	344	426	354	272	286												

Table 2. MIPS x 100

their resource allocations and orders the jobs accordingly. If a full exponential evaluation will not complete in time the code admits jobs of that revised rank based on this ordering. The case where there are a large number of jobs of the highest revised rank is obviously less satisfactory. And this case includes the case where all jobs have the same revised rank. The code performs a greedy scheme if pressed for time, but the results may be less than optimal. The philosophy is that an imperfect *macroQ* solution is better than no solution at all. In the pseudo-code we let L_r the be the number of job/template alternatives examined of revised rank r, whether linear or exponential.

The resolution granularity loop is simple in nature: *macroQ* starts with a coarse resolution to obtain a quick solution. Then it uses the time already spent to estimate the finest resolution it believes it can safely solve in the remaining time, subject to a reasonable minimum MIPS value. It reports the best importance found, and this is typically based on the finer resolution.

5 Experiments

In this section we experimentally evaluate *SODA* performance, focusing on the functions of job admission and resource allocation. Note that the trade-offs between the two can be quite subtle. In order to better reveal these subtle trade-offs, a carefully chosen set of well-controlled experiments were conducted. In these experiments, a variety of job submission scenarios were simulated. For the complete *SODA* performance with real applications running on *System S*, we refer readers to [17,18].

Now we will describe the experimental setup. The largest system installation we consider has 100 processing nodes with a rating of 11,000 MIPS each. In the experiments we examine the effect of removing 5 processing nodes (and thus 5% of the processing power) from the system at a time. The jobs presented to *macroQ* always remain the same: There are 19 jobs (labeled A through S), consisting of 7 required jobs of rank 1, 6 optional jobs of rank 2, and 6 optional

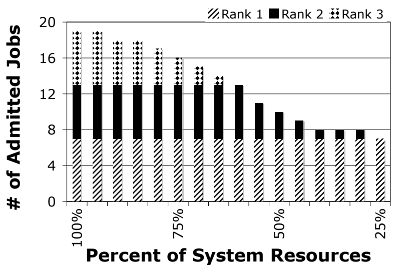
%	Rank 1							Rank 2						Rank 3					
	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
100	402	389	136	262	222	279	13	402	389	136	261	250	291	402	390	304	222	279	13
95	402	388	136	262	222	279	13	402	388	136	261	250	291	402	388	304	222	279	13
90	402	388	136	262	222	279	13	402	388	136	259	250	291	400	388	304	222	279	
85	400	388	129	262	216	279	10	400	388	129	259	250	291	400	388	304	216	279	
80	395	383	126	260	212	275	9	395	388	126	257	245	291	400	383	301		275	
75	389	377	126	260	215	274	9	389	377	126	257	245	286	389	383	301			
70	395	383	126	260	205	275	9	395	383	126	257	245	286	395		301			
65	400	388	129	262	216	279	13	400	388	129	259	250	291			304			
60	400	388	129	262	216	279	11	400	388	129	259	250	291						
55	400	388	136	262	221	279	13	400	388		259	250							
50	395	383	126	260	178	275	9	395	383		257								
45	395	388	126	260	212	279	9	400	388										
40	400	388	136	262	222	279	13	400											
35	359	355	125	259	178	273	10	389											
30	301	333	116	253	170	262	6	380											
25	301	333	116	253	170	262	5												

Table 3. Importance

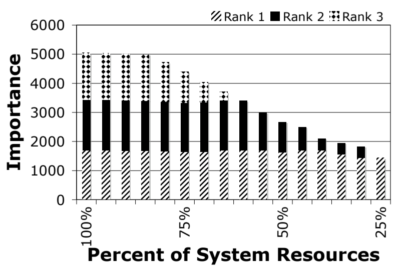
jobs of rank 3. The jobs are not interconnected, so rank and revised rank are identical for each job. The experiments are designed so that at 100 processing nodes the jobs will nearly (but not quite) use all the resources in the system when each is allocated their maximum useful resources. This occurs, as per the previous section, when each of the component importance functions becomes flat as a function of allocated resources. In fact, when *macroQ* is run on the full 100 processing nodes all 19 jobs are admitted and the average utilization of the processors is 97%.

Fig. 7(a) shows the number of jobs admitted by rank as the number of processing nodes decreases in 5% increments from 100 nodes to 25 nodes. At 95% all jobs are still admitted, though the processor utilization now is 100%. From there on the utilization remains at 100%, as one would expect based on *macroQ*'s design: The system is overloaded. At 90% 1 job of rank 3 is rejected, and all of the rank 3 jobs are gone by the 60 processing node level. But rank 1 and 2 jobs remain during the 65% to 100% range. Thus the rank waterline is 3. In the 30% to 60% range the rank waterline is 2. All rank 1 jobs are admitted, but more and more rank 2 jobs are rejected as the processing power decreases. At 25 processing nodes only the required rank 1 jobs are admitted. The system is fully stressed at this point, and a *macroQ* run at 20% of the processing nodes would not find a feasible solution: There would not be sufficient processing power to admit all of the required jobs even at their minimum acceptable resource allocations.

Fig. 7(b) shows the contribution to overall importance by rank as the number of processing nodes decreases from 100 to 25. Importance is a decreasing function of system resources, as should be the case. But between 100 and 85 nodes the importance curve is actually quite flat: The component importance curves turn out to be concave or close to concave, and there are sufficient resources available so that the solution lies near the flat part of each curve. Job S is rejected at 85% and 90%. But its importance is low and its resource requirements is high. One can see this in Tables 2 and 3. The first table shows the allocated resources (in hundreds of MIPS) by individual jobs as the number of processing nodes



(a) Admitted Jobs by Rank



(b) Importance by Rank

Fig. 7. Effect of Rank on Admission

decreases. The second table shows the corresponding importance values. Job S contributes an importance of only 13 at 33600 MIPS, so it is clearly highly expendable, and being of rank 3 it is jettisoned as soon as the offered load exceeds available resources. The effect on overall importance is minimal. The only other job with a poor ratio of importance to resources is job G. Job G is a twin of Job S, but it is required and *macroQ* cannot reject it. Observe in Fig. 7(b) that importance does start to decrease linearly from 80 down through 25 processing nodes. The available MIPS dictate that the solution to the component resource allocation problems occur on the steeper portion of each importance curve.

Examine Table 2 in the 3 ranges of resource allocations for which the admitted jobs remains identical. (The 100% and 95% rows have this property. So do the 90% and 85% rows. And finally, so do the 40%, 35% and 30% rows.) If the component importance curves are concave for each admitted job in these ranges, the separable resource allocation problems in *macroQ* would solve where the first differences (effectively the derivatives) at each job would be as close to equal as possible. And that would, in turn, imply that the resource allocations for each job would be monotone non-increasing as the number of processors decrease. In fact, this is the case in each of the three ranges, as an examination of the relevant columns shows.

Overall, however, the allocated resources for each job will not be monotone as the number of processors decrease. Consider, for example, the column for Job C in Table 2. The MIPS allocated are high in the 85% to 100% range, because the system is not heavily overloaded. As the number of processors decrease the MIPS allocated to Job C exhibits somewhat oscillatory behavior, decreasing through 70%, then increasing back to its maximum useful allocation at 55%, and so on. This behavior is primarily due to the changes in admission of the *other* jobs. As jobs get rejected the allocations they would have received become available, and *macroQ* will distribute these to the jobs that remain. (A secondary reason for the lack of monotonicity is the slight deviations from concavity.) At the 25 and 30 processor levels the system is truly stressed and there the job is given minimum acceptable MIPS allocations.

We have focused on the *macroQ* problems of job admission and resource allocation in these experiments. Extensive experimental analyses of the overall performance of *SODA* can be found in [17, 18].

6 Related work

Stream processing systems have been an active area of research in recent years. Example systems include Borealis [1], TelegraphCQ [5], STREAM [3], Aurora, StreamBase [15] and Medusa [23] and so on. These systems are mostly based on relational model and process voluminous quantities of incoming stream data. In contrast, *System S* is much more general in terms of programming model. It does not assume a relational model and it allows arbitrarily complex operators.

Most of these stream processing systems are designed to be run on more than one node, and thus there has also been work on scheduling and load-balancing the stream operations. While these scheduling approaches have some of the flavor of the work we have presented here, none targets our problem exactly. We describe some of these related scheduling approaches here.

The FIT algorithm [16] is a load-shedding algorithm which intelligently drops load. Determining where best to drop load can be quite a complex problem, since dropping at a particular operator has an effect on the downstream operators, sometimes an unintended one. In some cases, shedding load on a particular operator increases the resources for other operators on that node, and so could *increase* load at nodes downstream. FIT cleverly addresses this problem in a distributed way, but without a global notion of importance. The *SODA* scheduler provides this same functionality as part of its resource allocation and scheduling, and does so in a way that takes into account the processing graph for a job and the total system objectives.

In [21, 22], the authors address the problem of variance in stream rates. Both papers describe a way to distribute the load so that changes in input rate have a smaller chance of overloading the system. However, they do not address the case when the system is overloaded, and make no decisions about job admission. In [14], the authors provide a scheduling algorithm for a wide-area network that places operators so as to minimize network latency. In the local area network that we address, bandwidth, not network latency, is the main concern. In addition, their work does not address the problem of job admission. Others [12] also address scheduling to minimize latency.

The STREAM project [13] has goals somewhat similar to those presented in this paper. Their system handles queries in an SQL-like language. When resources are tight, they revise queries by dropping packets and/or changing internal parameters. Finally, in [20], the authors address admission control problem in a hypothetical stream processing systems. Their model assumes a linear processing graph. In other words, input stream is processed, successively, by a series of operators. Thus, no operator takes input from more than one source stream.

7 Conclusions

In this paper we have described the *SODA* scheduler for large-scale distributed stream processing applications. We have focused on one component, *macroQ*, in particular. This component is responsible for the two key functions of job admission and resource allocation. We have provided an introduction to *System S*, an introduction to the scheduler in general, and to the *macroQ* component of *SODA* in particular. We have evaluated the subtle trade-offs between job admission and resource allocation via simulation experiments, showing the capabilities of the scheduler.

References

1. Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the Borealis stream processing engine. In *CIDR*, 2005.
2. Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. SPC: A distributed, scalable platform for data mining. In *DMSSP*, 2006.
3. A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. STREAM: The Stanford stream data manager. *IEEE Data Engineering Bulletin*, 26, 2003.
4. J. Blazewicz, K. Ecker, G. Schmidt, and J. Weglarz. *Scheduling in Computer and Manufacturing Systems*. Springer-Verlag, 1993.
5. Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Vijayshankar Raman, Fred Reiss, and Mehul A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.
6. E. Coffman. *Computer and Job-Shop Scheduling Theory*. John Wiley and Sons, 1976.
7. T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw Hill, 1985.
8. B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: The System S declarative stream processing engine. In *ACM SIGMOD*, 2008.
9. K. Hildrum, F. Douglis, J. Wolf, P. S. Yu, L. Fleischer, and A. Katta. Storage optimization for large-scale stream processing systems. *ACM Transactions on Storage*, 3(4), 2008.
10. T. Ibaraki and N. Katoh. *Resource Allocation Problems*. MIT Press, 1988.
11. N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, implementation and evaluation of the linear road benchmark on the stream processing core. In *ACM SIGMOD*, 2006.
12. Geetika Lakshmanan and Robert Strom. Biologically-inspired distributed middleware management for stream processing systems. In *Middleware*, 2008.
13. Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcokc, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

14. Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *IEEE ICDE*, Washington, DC, USA, 2006. IEEE Computer Society.
15. StreamBase Systems. http://www.streambase.com.
16. Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *VLDB*, pages 159–170, 2007.
17. J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K-L Wu, and L. Fleischer. A scheduling optimizer for distributed applications: A reference paper. Technical Report 24453, IBM Research Report, 2007.
18. J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, and L. Fleischer. SODA: An optimizing scheduler for large-scale stream-based distributed computer systems. In *Middleware*, 2008.
19. K.-L. Wu, P. S. Yu, B. Gedik, K. W. Hildrum, C. C. Aggarwal, E. Bouillet, W. Fan, D. A. George, X. Gu, G. Luo, and H. Wang. Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on System S. In *VLDB*, 2007.
20. Cathy H. Xia, Don Towsley, and Chun Zhang. Distributed resource management and admission control of stream processing systems with max utility. In *ICDCS*, 2007.
21. Ying Xing, Jeong-Hyon Hwang, Uğur Çetintemel, and Stan Zdonik. Providing resiliency to load variations in distributed stream processing. In *VLDB*, pages 775–786. VLDB Endowment, 2006.
22. Ying Xing, Stan Zdonik, and Jeong-Hyon Hwang. Dynamic load distribution in the Borealis stream processor. In *IEEE ICDE*, pages 791–802, Washington, DC, USA, 2005. IEEE Computer Society.
23. S. Zdonik, M. Stonebraker, M. Cherniack, U. Cetintemel, M. Balazinska, and H. Balakrishnan. The Aurora and Medusa projects. *IEEE Data Engineering Bulletin*, 26(1), 2003.