Placement Strategies for Internet-Scale Data Stream Systems

Article *in* IEEE Internet Computing · November 2008

DOI: 10.1109/MIC.2008.129 · Source: IEEE Xplore

Geetika T. Lakshmanan
IBM
46 PUBLICATIONS 267 CITATIONS
SEE PROFILE

Ying Li
IBM
86 PUBLICATIONS 923 CITATIONS
SEE PROFILE

Robert Strom
IBM
87 PUBLICATIONS 5,413 CITATIONS
SEE PROFILE

Placement Strategies for Internet-Scale Data Stream Systems

Optimally assigning streaming tasks to network machines is a key factor that influences a large data-stream-processing system's performance. Although researchers have prototyped and investigated various algorithms for task placement in data stream management systems, taxonomies and surveys of such algorithms are currently unavailable. To tackle this knowledge gap, the authors identify a set of core placement design characteristics and use them to compare eight placement algorithms. They also present a heuristic decision tree that can help designers judge how suitable a given placement solution might be to specific problems.

Geetika T. Lakshmanan, Ying Li, and Rob Strom
IBM T.J. Watson Research Center

Over the past several years, researchers have made significant progress in stream-processing systems. Several groups have developed working prototypes,[1–5] and more than one stream-processing engine is now available or under development for commercial use (such as Exploratory Stream Processing Systems; see http://domino.research.ibm.com/comm/research_projects.nsf/pages/esps.index.html).[6] Stream-processing engines are emerging to help address the challenges in processing high-volume real-time data without using custom code. At these volumes, traditional techniques such as transactional databases don't scale well.[7] The US government, for instance, is using System-S, an ultra-powerful, large-scale (petabytes of data) stream-processing IBM system, currently running on 800 x86 computers with embedded Cell processors, that can analyze massive volumes of both structured or unstructured data in real time. (See "IBM Previews Ultra-Powerful Stream Processing System," www.wallstreetandtech.com/blog/archives/2007/06/ibm_previews_ul.html, for more details.) Fields that require such real-time stream processing range from e-trading to healthcare to intelligence.

Many factors can affect a stream-processing system's performance. Among them, one key issue is *operator placement*, how to optimally place stream-processing tasks onto a set of distributed machines.

In this article, we distill the expe-

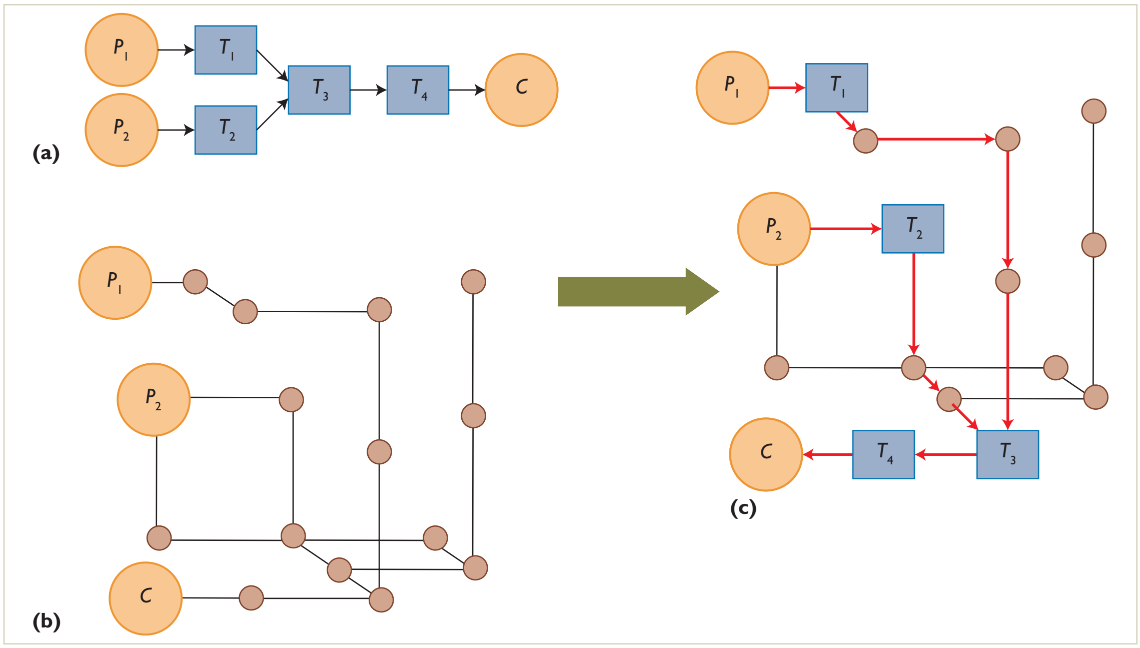

Figure 1. An illustration of the placement problem. (a) The logical flow graph consists of two producers, one consumer, and four tasks. (b) A network of machines where the producers and consumer are pinned and 14 other machines are available for hosting tasks. (c) A mapping of the unpinned operators in the flow graph in (a) to the machines in (b) that optimally satisfies some objective function.

rience of eight proposed and prototyped placement strategies by classifying them according to a set of core design characteristics. (For the sake of simplicity, we use the lead author's last name to identify each placement algorithm.) These eight placement algorithms represent a diverse sample of the different attempts researchers in both academia and industry have proposed to address operator placement. In addition, we provide an arrangement of stream-processing problem characteristics into a heuristic decision tree. We walk through the tree explaining its choices and why we believe one of the eight placement strategies is best suited to a specific problem. Although any particular set of design characteristics or *decision trees* might be inadequate for all purposes, we present the arrangement that we find most useful.

Operator Placement

Operator placement begins with the following information:

- a logical flow graph representing a set of message sources (producers) and a collection of stream-processing tasks (operators) that periodically consume messages, perform some processing, and deliver results either to message sinks (consumers) or to other stream-processing tasks;
- a physical topology representing a network of interconnected processing nodes available for deploying the tasks in the flow graph;
- a set of constraints; and
- measures or estimates of availability and demand, such as the message output rate of each operator, the expected CPU consumption of each task per single input message, the capacities of processing nodes, and latency and bandwidth measurements of links.

Some operators are bound to particular nodes, whereas certain operators have requests for specific resources. Operators that are restricted to a permanent physical network location are *pinned*; we also assume producers and consumers are pinned. Operators, such as a join or select that can be instantiated at an arbitrary node in the network, are *unpinned*.

A placement algorithm assigns operators to processing nodes while satisfying a set of constraints and attempts to optimize some objective function, such as the expected end-to-end latency for the assumed availability and demand estimates. Figure 1 illustrates the placement problem.

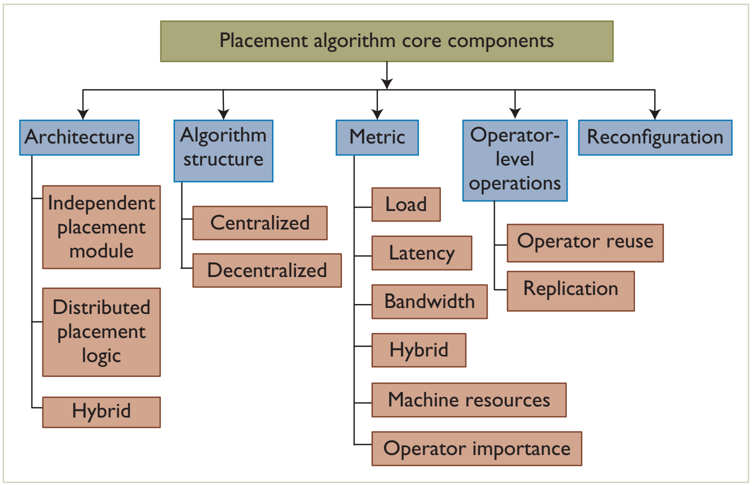

Figure 2. A placement algorithm's core components. These represent placement algorithm design choices that have significant impact on the performance of the stream-processing system.

In the flow graph, examples of processing operators include a join operator for coalescing financial data streams, an aggregation operator for monitoring seismic data, and a signal-processing operator for military surveillance. Figure 1a shows a simple flow graph, where P_1 and P_2 are two producers and C is a consumer. T_1 to T_4 are four processing operators. In some environments, the application designer creates the flow graph; in others, a compiler automatically generates the flow graph based on a set of input queries. In the latter case, each query requests some particular processing result that one or more consumers desires. Each operator in the graph does some processing, which possibly updates local state corresponding to the input message. An operator sends one or more output messages to either end consumers or one or more downstream operators.

Figure 1b shows a network of machines in a distributed stream-management system, where machines are represented by nodes. Producers P_1 and P_2 and consumer C are pinned to physical nodes in this example.

The placement problem refers to the optimal selection of the physical node that will host each unpinned operator in a flow graph so as to optimally satisfy a predefined global cost function. Examples of a cost function include the average end-to-end latency between all producers and consumers, or the average load on all machines in the network. Figure 1c shows one possible placement of the flow graph in Figure 1a to the network of machines in Figure 1b. Operator placement is an instance of a more general task-assignment problem that addresses how to optimally assign m tasks to n processors in a network. Finding the optimal assignment among the total possible assignments is computationally intractable; the problem is NP-complete.[8]

Core Components

Researchers have proposed innovative strategies for operator placement that have some features in common and in other respects differ widely. We propose a set of placement core components (as Figure 2 illustrates), against which we can compare and contrast eight strategies. These core components represent placement-algorithm design choices that have significant impact on the algorithm's performance.

Architecture

By *architecture*, we refer to the software implementation that executes placement decisions and how it interacts with the stream-processing system. Current implementations employ different ways of assembling a placement service into the stream-processing system's overall architecture:

- *Independent placement module*. In some placement implementations,[9,10] an independent service is responsible for computing the optimal placement. Having obtained a query plan, the placement service taps the system for updates on stream, network, and node conditions to guide its placement decision-making process. It then communicates the placement and any decisions to migrate operators from one node to another (also known as *operator migration decisions*) to the stream-processing layer. Thus, the placement service is separate from the stream-processing engine's remaining functionality. This type of architecture can affect the system performance because the placement module requires constant updates from the stream-processing system on network and node conditions. The length of time taken to provide these updates and the frequency with which they are provided will influence how quickly and accurately the system can respond to dynamic changes during runtime.
- *Distributed placement logic*. This approach implements a *local placement decision*

maker on each node that resides alongside a query processor element that collects node statistics such as input and output queue rates and resource information, including CPU utilization of operators that the node hosts. This decision maker uses information from the query processor element to make a placement decision. (The contract-based load management scheme that Magdalena Balazinska and her colleagues describe adopts this strategy.[11]) Such an architecture has good performance implications because resource information doesn't have to be routed to a central location in the network and is directly available where it's needed.
- *Hybrid*. In some implementations, a separate placement module is applied to execute an initial global placement for the entire network, while a local controller performs local adjustments on individual nodes to maintain an optimal placement in response to changing data, network, and resource conditions. (Representative works along this line are available elsewhere.[3,12,13])

Although a single independent placement module can execute placement decisions, the placement algorithm itself need not be centralized.[9,10]

Algorithm Structure

We characterize placement algorithms according to how they make placement decisions and whether these decisions require knowledge of the entire network.

Centralized algorithms have access to the entire network's current state, including workload information and resource availability. Although centralized placement algorithms have the advantage of potentially finding a globally optimal placement assignment,[3,6,8,12–15] they are vulnerable to scalability issues.

Decentralized algorithms can circumvent these scalability issues.[1,2,9–11,16–18] These algorithms make placement decisions based on local resource and workload information. Although most of these algorithms don't guarantee that their local optimization decisions are globally optimal, some can achieve convergence toward an optimal state, provided that certain assumptions hold.[11] Decentralized algorithms can efficiently respond to dynamic changes during runtime without resorting to network-wide computations.

Metric

The purpose of placement algorithms is to compute an assignment of operators to nodes that optimally satisfies an objective function. Thus, the objective function metric should be the one that best satisfies the stream-processing application's performance requirements.

Here, we've provided a list of some metrics popularly considered in defining objective functions:

- *Load*. System load is a measure of the amount of work a computer system is doing. CPU usage is one way of defining system load.[4,15] A placement algorithm might consider the aggregate of the CPU usage of all the operators placed on a given physical processing resource as a measure of the total load on that resource. An optimal placement in terms of load is one that ensures that the load on any machine doesn't exceed its capacity and that all machines are at or above their capacity thresholds if the total workload imposed on the system exceeds the sum of all the machines' capacity thresholds.[11] Rather than examining load itself, Ying Xing and her colleagues examined load variance.[13] They developed a correlation-based load-distribution algorithm that minimizes load variance and maximizes load correlation.
- *Latency*. A network link's latency is defined as the difference between the time the first byte of a packet is placed on a medium and the time the first byte is taken from the medium. This delay is caused by the rate at which signals are propagated in the link (such as electrons in the cable) and the medium's distance. The Pandit[14] and Lakshmanan[18] placement algorithms both use latency as a metric. The former uses a centralized algorithm, and the latter uses a decentralized algorithm to compute a mapping of operators in a query plan to available machines in the network such that it minimizes the average end-to-end latency between all producers and consumers.
- *Bandwidth*. Bandwidth refers to the amount of data that can be transmitted in a fixed amount of time. Link bandwidth is the rate at which bits can be inserted into the medium. Thus, the higher the bandwidth, the

more the bits that can be placed onto the medium in a given time frame. Yanif Ahmad and his colleagues computed an optimal placement by minimizing the network link bandwidth corresponding to the edges between an operator and its parent and child operators in the flow graph.[1,2]

- *Latency and load*. We can optimize placement for both latency and load. For instance, the Zhou[16] placement algorithm first computes an initial placement by minimizing the communication cost between producers and consumers. Then, it uses a load-balancing strategy to maintain a good placement and adapt to changes in data, network, and resource conditions. In addition to latency and load, the Ying[17] placement algorithm accounts for the cost of storing or caching operator-related data at a particular node.
- *Latency and bandwidth*. Other placement approaches[1,2,9,19] minimize the network usage, $u(q)$, defined as the amount of data that is in transit for a query q at a given instant:

  $$u(q) = \sum_{l \in L} DR(l).Lat(l),$$

  where L is the set of links the stream uses, $DR(l)$ is the data rate over link l, and $Lat(l)$ is the latency of l. This metric tolerates paths that require extra latency provided that they help reduce the overall stream bandwidth. Another metric that incorporates both latency and bandwidth maximizes a utility function that is defined in terms of network delay, available bandwidth, and required bandwidth on each edge of the flow graph.[10]
- *Machine resources*. A placement metric might incorporate operator-specific resource requirements, particularly if machines in the network are heterogeneous. For instance, an operator might require specific hardware or a certain allocation of CPU capacity.[8,12,20] Unless the placement algorithm takes such resource requirements into account, it might not find a feasible solution. This assumes that the stream-processing system users provide operator-specific resource requirements as input to the placement algorithm.
- *Operator importance*. We can express the relative importance of different operators, streams, or an entire flow graph in the form of weights on operators, where a higher weight denotes higher importance. For instance, the Amini[12] placement algorithm incorporates such constraints in its objective function in the form of weighted throughput: a positive weight is attached to each stream that is a system output. By summing the weighted throughput of each system output, it measures the system's total productive work, while taking into account the user-specified constraint that some results are more important than others. The Kumar[10] placement algorithm distinguishes between the importance of different flow graphs from the perspective of the stream-processing application through a user-provided priority value p.

In addition to the choice of an objective function metric, the stream-processing application also dictates the spectrum of optimizations that can be conducted at the operator level to improve performance.

Operator-Level Operations

We've identified *operator reuse* and *replication* as two performance-enhancing components of placement algorithms. Although these components aren't required to obtain an accurate placement, some works have demonstrated that they can considerably improve the efficiency of stream-processing systems.[9,20,21]

Consumers frequently execute identical or partially identical queries on data streams derived from sources such as financial markets and sensor networks. It would be a waste of system resources to instantiate new operators for each query, particularly if it's feasible to reuse result streams from currently deployed operators in the stream-processing system to completely or partially satisfy them. Therefore, several placement strategies first try to discover whether any of the requested streams have been generated by previously instantiated query plans.[9,14,16] To maximize the reusing benefit, the system reuses the result streams generated during the latest possible stages in the query plan. Thus, the system only needs to instantiate the remaining query plan for processing the reusable existing streams to generate user-requested streams.

Although they don't propose a placement algorithm, Thomas Repantis and his colleagues discussed an elaborate strategy for composing existing streams to satisfy new placement requests.[21] Their technique can be incorporated into an existing placement strategy. To efficiently provide operator reuse, placement algorithms must determine, among other things, whether the impact of the new application would cause quality-of-service violations to existing applications.

Stateless operators process messages independently of knowing the messages' history. A placement algorithm can replicate such operators to improve placement performance. Operator replication provides an efficient way to manage load when data rates are too high for a single machine to handle.[20] A replication strategy involves partitioning an operator's workload among multiple copies, which, if assigned to multiple machines, enables parallelizable message processing. An early, visionary research study identified such workload partitioning schemes as one of stream processing's eight real-time requirements.[7] Replication strategies can be easily incorporated into placement algorithms.[20] A good replication mechanism should automatically determine the operators to replicate, the number of replicas, and the proportion of workload to be assigned to each replica. Replicating stateful operators (or intraoperator parallelism), such as windowed joins and group-by-aggregate, has been well-studied in parallel database systems.[22]

Reconfiguration

How to adapt to changes is another important factor in the design of a placement algorithm for distributed stream-processing systems. Figure 3 illustrates reconfiguration components in a placement strategy. We can characterize such changes into three categories:

- *Network infrastructure*. This refers to changes in the availability of machines and physical links between machines. For instance, a network might have machines added, existing machines go down, and network links become congested and unreliable.
- *Data characteristics*. The rate at which data is streamed through the network might change. In particular, data rates could be bursty.
- *Flow-graph information*. Additional operators, producers, and consumers could be introduced at any time. It's also necessary to remove operators when the stream-processing system no longer requires their functionality.

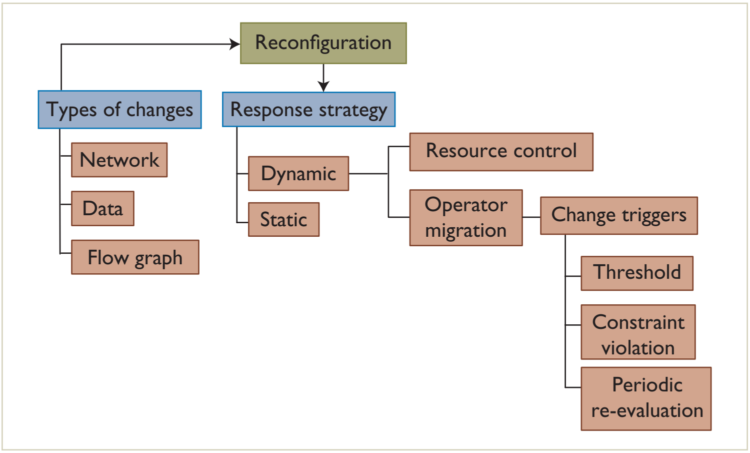

Figure 3. Components of reconfiguration in a placement strategy. This includes the types of changes that can occur and the flavor of response strategies. Current literature explores several different kinds of dynamic response strategies.

The frequency of these changes dictates the response strategy that the placement algorithm uses.

Response Strategies

The strategy for responding to changes, which is built into the placement algorithm, influences the placement results' performance and accuracy. By dynamically responding to changes during runtime, the algorithm can continue to maintain a good placement without interrupting the system's functionality.[1–3,9–13,16–18] Decentralized placement algorithms don't require access to the system's global resource and workload state. Therefore, they can respond dynamically by making local configuration changes. The placement's accuracy and stability depends on the extent to which the algorithm guarantees that locally optimal placement adjustments contribute toward achieving a globally optimal placement. On the other hand, static response strategies recompute the placement of all operators on the entire network offline.[14] Although the placement algorithm interrupts stream processing to deploy new placement results, the likelihood of obtaining a globally optimal placement is higher with static strategies.

One work circumvented the need for a response strategy by developing a placement scheme that's resilient to load changes.[15] These authors assumed that the cost of moving op-

Table 1. Comparing eight placement algorithms with respect to core components.*

Core components	Pietzuch[9]	Balazinska[11]	Abadi[3]	Zhou[16]	Kumar[10]
Architecture description	Independent module: placement is computed in an overlay network	Distributed placement logic: placement decision maker is within each node	Hybrid: global optimizer performs a one-time placement Local controller for each node	Hybrid: distributed initial placement Local controller for each node	Independent placement module: placement is orchestrated by autonomic middleware
Algorithm structure	Decentralized	Decentralized	Centralized initial placement Decentralized load shedding	Decentralized	Decentralized
Metric	Product of latency and bandwidth	Load and available resources (processor cycles, bandwidth, memory)	Latency and bandwidth	Latency for the initial placement, and load balancing for operator migration	Latency, required bandwidth, and available bandwidth
Operator-level operations	Reuse	Replication	Replication	Reuse	None
Response strategy	Dynamic: operator migration	Dynamic: operator migration	Dynamic: operator migration and load shedding	Dynamic: operator migration	Dynamic: operator migration
Change triggers	Periodic re-evaluation	Periodic re-evaluation	Threshold	Threshold	Threshold and constraint violation

*We use the lead author's last name to identify each placement algorithm.

erators is extremely high, so they developed a placement scheme that lets the system withstand the largest set of input rate combinations.

Static response. The static-response strategy Vinayaka Pandit and his colleagues described requires knowledge of each node's state in the network to compute a new placement.[14] This could serve as a significant bottleneck if nodes in the network are geographically distributed. On the other hand, this provides these algorithms greater leverage in terms of reaching an optimal solution. The Pandit technique computes a new placement for each unpinned operator in the flow graph each time it runs. This can be expensive if the majority of operators change locations between placements. In this case, the stream-processing system must stop the current execution and save all necessary state associated with an operator. Execution can only resume after the operator and its associated state are moved by the stream-processing system to the new location. Transporting state for state-intensive operators such as the symmetric multiple-way hash join operator on highly congested links could further delay processing.

Dynamic response. To maintain good performance in response to changing data, network, and resource conditions, existing works employ two main dynamic response strategies: *operator migration* and *resource control*.

Operator migration involves migrating operators between machines in the network to optimally satisfy a performance objective in response to changes in the data characteristics, flow graph, network conditions, and available resources. This is the most common strategy that placement algorithms use to respond to changes.

We've identified several different ways in which machines are selected for operator migration. Yongluan Zhou and his colleagues propose continually storing and updating a set of load-balancing partners in each node.[16] Balazinska and her colleagues allowed individual nodes to negotiate load-exchange contracts among themselves.[11] Some algorithms propose

Ahmad[1,2]	Amini[12]	Pandit[14]
Distributed placement logic: placement implemented by zone-based coordinator nodes	Hybrid: global optimizer performs a periodic placement. Local controller for each node	Independent module: global optimizer performs a periodic placement
Decentralized	Centralized initial placement Decentralized resource control	Centralized
Bandwidth, or product of latency and bandwidth	Latency, bandwidth, and machine resources	Latency
Replication	Replication	Reuse
Dynamic: operator migration	Dynamic: resource control on each node	Static: placement is recomputed each time
Threshold	Periodic re-evaluation	None: recomputing placement is initiated by user

dividing the machines in the network into either zones[1,2] or hierarchical clusters[10] and then entrusting the responsibility of operator migration between machines in the zone or cluster to a coordinator machine.

Migrating operators in response to every single change in network conditions regardless of how trivial they are could lead to unnecessary and expensive reconfigurations. Such reconfigurations will consequently cause delays in system response time. Existing operator migration strategies apply these three techniques to avoid these problems:

- *Performance threshold*. This approach triggers a reconfiguration when performance falls below a user-specified threshold.[10]
- *Constraint violation*. This approach specifies that reconfiguration should only occur when the current placement has violated some constraints, such as upper bounds on end-to-end latency.[10] The main rationale behind this approach is that it might be easier to check for a constraint's violation than to maintain a threshold against some performance metric's optimal value.
- *Periodic re-evaluation*. This approach periodically re-evaluates a placement decision and triggers change if the performance gain of migrating unpinned operators to different machines is higher than a preset minimum threshold.[9]

The resource control strategy controls input, output, and processing rates of the system's operators and then adjusts these rates gradually, as necessary, to keep input buffers at target levels.[12] This lets operators take advantage of batching, processing several data packets in quick succession, rather than having each operator process only one data packet at a time. Using this form of batching, the system can avoid context-switching overhead, decrease memory cache misses, and transfer data in larger chunks. Furthermore, by not letting buffers become too full, this strategy reduces the overall end-to-end processing latency. Another approach uses a local controller on each node to coordinate load shedding.[3]

A Comparison of Existing Work

There are numerous published papers on placement algorithms, from both industry[5,6,12,14,17,18] and academia.[1–4,8–11,16,19] Some have been implemented and tested on real systems,[1–4,9–11] while others have been implemented in simulation.[14,16,19] We've selected eight placement algorithms[1–3,9–12,14] and compared them in Table 1 according to the core components we discuss in this article. Our selection stems from the diversity these algorithms bring in terms of the core components. Furthermore, although all these placement algorithms are intended for stream-processing systems, they make different assumptions. Some assume that data sources are geographically distributed,[1,2,9,10,16] others consider nodes to be autonomous and spread across administrative domains,[11] and still others assume that nodes consist of a cluster of

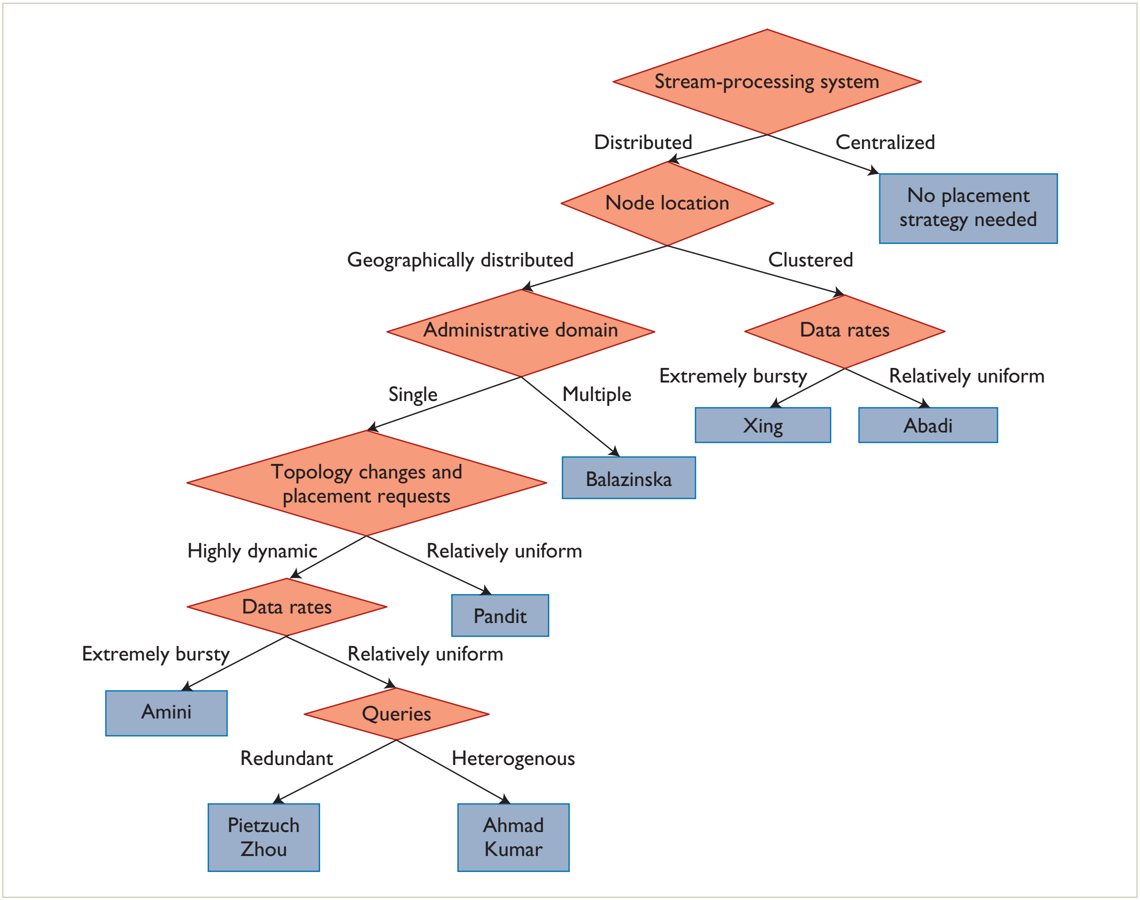

Figure 4. Decision tree. This tree analyzes the suitability of one of the placement solutions in Table 1 to specific problems.

homogenous, highly connected, extremely fast processors.[3,12] These different assumptions influence design choices in these algorithms.

The placement strategies in Table 1 differ widely in terms of how they're incorporated into the stream-processing system. Most of the algorithms are decentralized and able to respond to changes dynamically during runtime.[1,2,9,10,16] With the exception of Balazinska,[11] Amini,[12] and Ahmad,[1,2] none of the decentralized algorithms are accompanied with extensive theoretical guarantees on the optimality of the solutions they provide. Almost all the algorithms consider latency as an important metric in their objective functions. As most of these approaches point out,[1,2,10,16] some form of load balancing must accompany this metric because, without it, latency is adversely affected.

Six out of these eight placement strategies dynamically respond to changes during runtime by conducting operator migration, which has critical implications for a dynamic placement algorithm's stability and accuracy. In nonreplicated approaches, migrations can significantly interrupt the stream-processing system's functionality because the processing done by operators on the original node first needs to be paused, all necessary states saved, and finally the same processing is resumed on a new node. Furthermore, it's important to avoid migrations that have little or no impact on improving the system's overall performance. Accomplishing this is of course challenging, especially in a dynamic environment. Factors such as bursty data rates could make such estimation difficult and inaccurate. Algorithms that rely on thresholds to trigger operator migrations[1,2,10,16] need a procedure that updates these thresholds in response to changes in the data characteristics.

Determining the Right Placement Strategy

Although all the techniques in Table 1 share a common goal of achieving the highest possible quality of service, their individual design characteristics lead to different strengths. Figure 4 shows a decision tree organizing problem characteristics to determine whether a particular placement algorithm applies to a specific problem. Other metrics and decision trees could apply here, but we find this one particularly useful.

The top of the tree shows the stream-processing system axis. If the stream-processing system consists of one machine, then there's no need for a placement strategy because all necessary tasks will be placed on this one machine. Our tree thus only continues along the distributed stream-processing system axis.

Nodes in a distributed stream-processing system could either be in a cluster or geographically distributed. Clustered nodes are usually under the same administrative domain, and a centralized placement controller (such as Abadi[3] and Xing[13]) is effective in executing placement decisions. The cluster coordinator can efficiently access variable state in the network and use it to execute placement. It's also easy for the controller to notice topology changes in the network and update placement accordingly. If the workload is highly unpredictable, then the Xing load-balancing technique, which balances the average load among processing nodes and minimizes the load variance on each node, is appropriate.[13] On the other hand, if the workload is relatively uniform and placement can be maintained with load-shedding and local operator migration techniques, then the Abadi technique is more appropriate.[3] If nodes are geographically distributed, a central coordinator will be ineffective, and it's desirable to have a placement service that places operators according to a network-aware strategy that accounts for both latency and bandwidth.

Our next most important characteristic is the administrative domain, under which the geographically distributed nodes in a distributed stream-processing system are governed. Under multiple administrative authorities, a placement algorithm must specify different cooperation policies that determine the degree of trust and dependency between the systems.[5] The Balazinska algorithm achieves this through prenegotiated load-exchange contracts between nodes.[11]

The next axis in the tree is for topology changes and placement requests. Centralized placement algorithms (such as Pandit[14] and Srivastava[8]) are suitable for relatively static applications, where new placement requests and network topology changes are infrequent. These applications can benefit from the high quality of placement solutions that centralized placement algorithms provide without being adversely affected by the computational time these algorithms use. Decentralized algorithms are more suitable for efficiently responding locally to changes in highly dynamic environments.

Highly variable data rates can cause an existing placement to quickly become suboptimal. In such situations, algorithms that respond to changes by operator migration will lead to placement instability, so node-level resource control strategies (such as Amini[12]) are more suitable. When data rates are relatively uniform, several decentralized placement solutions that dynamically respond to changes during runtime are potentially appropriate.[1,2,9,10,16]

If queries in the stream-processing system tend to be highly redundant, then placement strategies that incorporate operator reuse (such as Pietzuch[9] and Zhou[16]) are much more efficient than the ones that do not.

Despite the extensive recent activity in research on placement algorithms for stream-processing systems, based on the algorithms we compared in this article, we believe much more work remains to be done in this field before we arrive at stable and practical placement strategies. Task placement is a fundamental problem that arises in other Internet-based applications, such as in service composition for service-oriented architectures, Web application placement for enterprise data centers, and sensor placements for sensor networks.

Advancement in one application domain can spark progress in another. Thus, we strongly suggest that placement algorithm designers consult these other domains for inspiration.

References

1. Y. Ahmad and U. Cetintemel, "Network-Aware Query Processing for Stream-Based Applications," *Proc. Conf. Very Large Databases*, Elsevier Science, 2004, pp. 456–467.
2. Y. Ahmad et al., "Network Awareness in Internet-Scale Stream Processing," *Proc. IEEE Data Eng. Bulletin*, vol. 28, no. 1, 2005, pp. 63–69.
3. D.J. Abadi et al., "The Design of the Borealis Stream Processing Engine," *Proc. 2nd Biennial Conf. Innova-*

tive Data Systems Research, Very Large Data Base Endowment, 2005, pp. 277–289.
4. M. Cherniack et al., "Scalable Distributed Stream Processing," *Proc. 1st Biennial Conf. Innovative Database Systems*, Very Large Data Base Endowment, 2003, pp. 23–35.
5. M. Branson et al., "CLASP: Collaborating, Autonomous Stream Processing Systems," *Proc. ACM Int'l Middleware Conf.*, ACM Press, 2007, pp. 348–367.
6. J. Wolf et al., "SODA: An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems," to be published in *Proc. ACM Int'l Middleware Conf.*, Springer-Verlag, 2008.
7. M. Stonebraker, U. Cetintemel, and S. Zdonik, "The Eight Requirements of Real-Time Stream Processing," *Proc. SIGMOD Record*, vol. 34, no. 4, 2005, pp. 42–47.
8. U. Srivastava, K. Mungala, and J. Widom, "Operator Placement for In-Network Stream Query Processing," *Proc. Principles of Distributed Systems*, ACM Press, 2005, pp. 250–258.
9. P. Pietzuch et al., "Network-Aware Operator Placement for Stream-Processing Systems," *Proc. Int'l Conf. Data Eng.*, IEEE CS Press, 2006, p. 49.
10. V. Kumar, B.F. Cooper, and K. Schwan, "Distributed Stream Management using Utility-Driven Self-Adaptive Middleware," *Proc. 2nd IEEE Int'l Conf. Autonomic Computing*, IEEE CS Press, 2005, pp. 3–14.
11. M. Balazinska, H. Balakrishnan, and M. Stonebraker, "Contract-Based Load Management in Federated Distributed Systems," *Proc. Symp. Network System Design and Implementation*, Usenix Assoc., 2004, pp. 197–210.
12. L. Amini et al., "Adaptive Control of Extreme-Scale Stream Processing Systems," *Proc. Int'l Conf. Distributed Computing Systems*, IEEE CS Press, 2006, pp. 71–71.
13. Y. Xing and S. Zdonik, and J-H. Hwang, "Dynamic Load Distribution in the Borealis Stream Processor," *Proc. Int'l Conf. Data Engineering.*, IEEE CS Press, 2005, pp. 791–802.
14. V. Pandit et al., *Performance Modeling and Placement of Transforms for Stateful Mediations*, tech. report RI08002, IBM, 2004; http://domino.watson.ibm.com/library/cyberdig.nsf/Home.
15. Y. Xing et al., "Providing Resiliency to Load Variations in Distributed Stream Processing," *Proc. Conf. Very Large Databases*, Very Large Data Base Endowment, 2006, pp. 775–786.
16. Y. Zhou et al., "Efficient Dynamic Operator Placement in a Locally Distributed Continuous Query System," *Proc. 14th Int'l Conf. Cooperative Information Systems*, Springer-Verlag, 2006, pp. 54–71.
17. L. Ying et al., "Distributed Operator Placement and Data Caching in Large-Scale Sensor Networks," *Proc. IEEE Conf. Computer Comm.* (INFOCOM), IEEE Press, 2008, pp. 977–985.
18. G. Lakshmanan and R. Strom, "Biologically-Inspired Distributed Middleware Management for Stream Processing Systems," to be published in *Proc. ACM Int'l Middleware Conf.*, Springer-Verlag, 2008.
19. B.J. Bonfils and P. Bonnet, "Adaptive and Decentralized Operator Placement for In-Network Query Processing," *Proc. Conf. Information Processing in Sensor Networks*, Springer-Verlag, 2003, pp. 47–62.
20. Y. Li, R. Strom, and C. Dorai, "Placement of Replicated Message Mediation Components," *Proc. ACM Int'l Middleware Conf.* (Demos and Posters), ACM Press, 2007, p. 2.
21. T. Repantis, X. Gu, and V. Kalogeraki, "Synergy: Sharing-Aware Component Composition for Distributed Stream Processing Systems," *Proc. ACM Int'l Middleware Conf.*, Springer-Verlag, 2006, pp. 322–341.
22. M. Shah et al., "Flux: An Adaptive Partitioning Operator for Continuous Query Systems," *Proc. 19th Int'l Conf. Data Eng.*, IEEE CS Press, 2003, pp. 25–36.

Geetika T. Lakshmanan is a research staff member at the IBM T.J. Watson Research Center. Her research interests include designing and prototyping solutions for computer systems that combine one or more of these disciplines: distributed systems, sensor networks, computer graphics, discrete and computational geometry, and biology. Lakshmanan has a PhD in computer science from Harvard University. She is a member of the ACM and the Sigma Xi Scientific Research Society. Contact her at gtlakshm@us.ibm.com.

Ying Li is a research staff member at the IBM T.J. Watson Research Center, where she leads projects on learning content analysis and management. Her research interests include digital image processing, multimedia content analysis, e-learning applications, and distributed computing. Li has a PhD in electrical engineering from the University of Southern California. She is a senior member of the IEEE. Contact her at yingli@us.ibm.com.

Rob Strom was a research staff member at the IBM T.J. Watson Research Center until he retired in 2007, but he continues to consult for IBM Research in areas related to cloud computing. His research interests include programming languages; distributed, fault-tolerant, and event-based systems; simplification of component-based programming; and fault-tolerance and transparent migration of applications in distributed computing environments such as cloud computing. Strom has a PhD in computer science from Washington University. Contact him at robstrom@us.ibm.com.