Resource Allocation Strategies for Constructive In-Network Stream Processing

Anne Benoit[1] Henri Casanova[2] Veronika Rehn-Sonigo[1] Yves Robert[1]

[1] École Normale Supérieure de Lyon, France
[2] University of Hawai‘i at Manoa, Honolulu, USA
{Anne.Benoit|Veronika.Rehn|Yves.Robert}@ens-lyon.fr,henric@hawaii.edu

Abstract

We consider the operator mapping problem for in-network stream processing, i.e., the application of a tree of operators in steady-state to multiple data objects that are continuously updated at various locations in a network. Examples of in-network stream processing include the processing of data in a sensor network, or of continuous queries on distributed relational databases. Our aim is to provide the user a set of processors that should be bought or rented in order to ensure that the application achieves a minimum steady-state throughput, and with the objective of minimizing platform cost. We prove that even the simplest variant of the problem is NP-hard, and we design several polynomial time heuristics, which are evaluated via extensive simulations and compared to theoretical bounds.

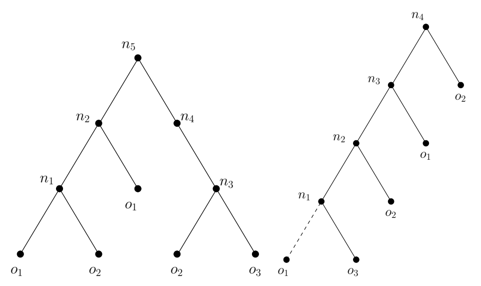
Keywords: *in-network stream processing, cloud computing, operator mapping, complexity, polynomial heuristics.*

1. Introduction

We consider the execution of applications structured as trees of operators. The leaves of the tree correspond to basic data objects that are spread over different servers in a distributed network. Each internal node in the tree denotes the aggregation and combination of the data from its children using some operator, which in turn generates new data that is used by the node's parent. Basic data objects are continuously being updated, so that the tree of operators must be applied continuously. The goal is to produce final results (at the root node) at some desired rate.

The above problem, which is called *stream processing* [3], arises in several domains. An important domain of application is the acquisition and refinement of data from a set of sensors [16]. For instance, [16] outlines a video surveillance application in which the sensors are cameras located at different locations over a geographical area. The goal of the application could be to identify monitored areas in which there is significant motion between frames, particular lighting conditions, and correlations between the monitored areas. This can be achieved by applying several operators (e.g., filtering, pattern recognition) to the raw images, which are produced/updated periodically. Another example arises in the area of network monitoring [17, 7]. In this case routers produce streams of data pertaining to forwarded packets. One can often view stream processing as the execution of one or more "continuous queries" in the relational database sense of the term (e.g., a tree of join and select operators). A continuous query is applied continuously, i.e., at a reasonably fast rate, and returns results based on recent data generated by the data streams. Many authors have studied the execution of continuous queries on data streams [11].

In practice, the execution of the operators on the data streams must be distributed over the network. In some cases, for instance in the aforementioned video surveillance application, the servers that produce the basic objects do not have the computational capabilities to apply all operators. Besides, objects must be combined across devices, thus requiring network communication. Although a simple solution is to send all basic objects to a central compute server, it proves unscalable for many applications due to network bottlenecks. Also, this central server may not be able to meet the desired target rate for producing results due to the sheer amount of computation involved. The alternative is then to distribute the execution by mapping each node in the operator tree to one or more compute servers in the



(a) Standard tree. (b) Left-deep tree.

Figure 1. Examples of applications structured as a binary tree of operators.

network. One then talks of *in-network* stream processing. In-network stream processing systems have been developed [6, 17, 12]. and face the following question: where should operators be mapped in the network?

The operator-mapping problem for in-network stream processing was studied in [16, 15]. Most relevant to our work is the recent work in [15], in which the problem is studied for an ad-hoc objective function that trades off application delay and network bandwidth consumption. In this paper we study a more general objective function. We enforce the constraint that the rate at which final results are produced, or *throughput*, is above a given threshold. This corresponds to a Quality of Service (QoS) requirement, which is almost always desirable in practice (e.g., up-to-date results of continuous queries must be available at a given frequency). Basic objects may be replicated at multiple locations, i.e., available and updated at these locations. In terms of the computing platform we consider a "constructive" scenario: either the user can build the platform from scratch using off-the-shelf components, or computing and network units are rented by a cloud provider (e.g. [1]). Our goal is to construct a distributed network dedicated to the given application, which minimizes the monetary cost while ensuring that the desired throughput is achieved.

Our contributions are as follows: (i) we formalize the operator-placement problem; (ii) we establish complexity results (all problems turn out to be NP-complete); (iii) we propose several polynomial heuristics; (iv) we compare heuristics through extended simulations, and assess their absolute performance.

2. Models

Application Model – We consider an application that can be represented as a set of operators $\mathcal{N} = \{n_1, n_2, \dots\}$ arranged as a binary tree, as shown in Figure 1. Operations are initially performed on basic objects, which are made available and continuously updated at given locations in a distributed network. We denote the set of basic objects, which are leaves of the tree, by $\mathcal{O} = \{o_1, o_2, \dots\}$. Several leaves may correspond to the same object, as illustrated in the figure. Internal nodes represent operator computations. For an operator n_i we define $\mathit{Leaf}(i)$ as the index set of the basic objects needed for the computation of n_i, if any, $\mathit{Ch}(i)$ as the index set of the node's children in $\mathcal{N}$, if any, and $\mathit{Par}(i)$ as the index of the node's parent in $\mathcal{N}$, if it exists. We have the constraint that $|\mathit{Leaf}(i)| + |\mathit{Ch}(i)| \leq 2$ because the tree is binary. All functions above are extended to sets of nodes: $f(I) = \cup_{i \in I} f(i)$, where I is an index set and f is Leaf, Ch or Par. If $|\mathit{Leaf}(i)| \geq 1$, then operator n_i needs at least one basic object for its computation. We call such an operator an *al-operator* (for "almost leaf").

The application must be executed so that it produces final results, where each result is generated by executing the whole operator tree once, at a target rate. We call this rate the application *throughput*. Each operator $n_i \in \mathcal{N}$ must compute (intermediate) results at a rate at least as high as the target application throughput. Conceptually, a server executing an operator consists of two concurrent threads that run in steady-state. One thread periodically downloads the most recent copies of the basic objects corresponding to the operator's leaf children, if any. For our example tree in Figure 1(a), n_1 needs to download o_1 and o_2 while n_2 downloads only o_1 and n_5 does not download any basic object. Note that these downloads may simply amount to constant streaming of data from sources that generate data streams. Each download has a prescribed cost in terms of bandwidth based on application QoS requirements (e.g., so that computations are performed using sufficiently up-to-date data). A basic object o_k has a size δ_k (in bytes) and needs to be downloaded by the processors that use it with frequency f_k. Therefore, these basic object downloads consume an amount of bandwidth equal to $rate_k = \delta_k \times f_k$ on each network link and network card through which the object is communicated. Another thread receives data from the operator's non-leaf children, if any, and performs some computation using downloaded basic objects and/or data received from other operators. The operator produces some output that needs to be passed to its parent operator. The computation of operator n_i (to evaluate the operator once) requires w_i operations, and produces an output of size δ_i.

Platform Model – The target distributed network is a fully connected graph interconnecting a set of resources $\mathcal{R} = \mathcal{P} \cup \mathcal{S}$, where $\mathcal{P}$ denotes compute servers, or *processors* for short, and $\mathcal{S}$ denotes data servers, or *servers* for short. Servers hold and update basic objects, while processors apply operators of the application tree. Each server $S_l \in \mathcal{S}$ (resp. processor $P_u \in \mathcal{P}$) is interconnected to the network via a network card with maximum bandwidth Bs_l (resp. Bp_u). We assume that the same interconnect technology is used to connect all processors, and thus the link between two distinct processors P_u and P_v is bidirectional and has bandwidth bp, while the network link from a server S_l to a processor P_u has bandwidth bs_l; on such links the server sends data and the processor receives it. In addition, each processor $P_u \in \mathcal{P}$ is characterized by a compute speed s_u. We denote the case in which all processors are homogeneous because only one type of CPUs and network cards can be acquired ($Bp_u = Bp$ and $s_u = s$) Constr-Hom. Correspondingly, we term the case in which the processors are heterogeneous with various compute speeds and network card bandwidth Constr-LAN.

Resources operate under the full-overlap, bounded multi-port model [9], where a resource can be involved in computing, sending data, and receiving data simultaneously. The “multi-port” assumption states that resource R can send/receive data simultaneously on multiple network links. The “bounded” assumption states that the total transfer rate of data sent/received by resource R is bounded by its network card bandwidth.

Mapping Model and Constraints – Our objective is to purchase/rent a set of processors, and then to map operators, i.e., internal nodes of the application tree, onto these processors. Additionally, if a tree node has at least one leaf child, then it must continuously download up-to-date basic objects from the fixed set of servers, which consumes bandwidth on its processor's network card. Each processor is in charge of one or several operators. For each operator on processor P_u, while P_u computes for the t-th final result, it sends to its parent (if any) the data corresponding to intermediate results for the $(t-1)$-th final result. It also receives data from its non-leaf children (if any) for computing the $(t+1)$-th final result. Recall that all three activities are concurrent. We assume that a basic object can be replicated, in some out-of-band manner specific to the target application (e.g., via a distributed database infrastructure). In this case, a processor can choose among multiple data sources when downloading a basic object. Conversely, if two operators require the same basic object and are mapped to different processors, they must both continuously download that object (and incur the corresponding network overheads).

We denote the mapping of the operators in $\mathcal{N}$ onto the processors in $\mathcal{P}$ using an allocation function a: $a(i) = u$ if operator n_i is assigned to processor P_u. Conversely, $\bar{a}(u)$ is the index set of operators mapped on P_u: $\bar{a}(u) = \{i \mid a(i) = u\}$. We also introduce new notations to describe the location of basic objects. Processor P_u may need to download some basic objects from some servers. We use $DL(u)$ to denote the set of (k,l) couples where processor P_u downloads object o_k from server S_l. Each processor has to communicate and compute fast enough to achieve the application throughput ρ. A communication occurs only when a child or the parent of a given tree node and this node are mapped on different processors. We have the following constraints:

• Each processor P_u cannot exceed its computation capability:

$$\forall P_u \in \mathcal{P}, \quad \sum_{i \in \bar{a}(u)} \rho \cdot \frac{w_i}{s_u} \leq 1 \tag{1}$$

• P_u must have enough bandwidth capacity to perform all its basic object downloads and all communication with other processors. The first term corresponds to basic object downloads, the second term corresponds to inter-node communications when a tree node is assigned to P_u and some of its children nodes are assigned to another processor, and the third term corresponds to inter-node communications when a tree node is assigned to P_u and its parent node is assigned to another processor:

$$\forall P_u \in \mathcal{P}, \quad \sum_{(k,l) \in DL(u)} rate_k + \sum_{j \in Ch(\bar{a}(u)) \setminus \bar{a}(u)} \rho.\delta_j + \sum_{j \in Par(\bar{a}(u)) \setminus \bar{a}(u)} \; \sum_{i \in Ch(j) \cap \bar{a}(u)} \rho.\delta_i \leq Bp_u \tag{2}$$

• Server S_l must have enough bandwidth capacity to support all basic object downloads:

$$\forall S_l \in \mathcal{S}, \quad \sum_{P_u \in \mathcal{P}} \sum_{(k,l) \in DL(u)} rate_k \leq Bs_l \qquad (3)$$

• The link between server S_l and processor P_u must have enough bandwidth capacity to support all possible object downloads from S_l to P_u:

$$\forall P_u \in \mathcal{P}, \forall S_l \in \mathcal{S}, \quad \sum_{(k,l) \in DL(u)} rate_k \leq bs_{l,u} \qquad (4)$$

• The link between P_u and P_v must have enough bandwidth capacity to support all possible communications between the nodes mapped on both processors. This constraint can be written similarly to constraint (2) above, but without the cost of basic object downloads, and specifying that P_u communicates with P_v:

$$\forall P_u, P_v \in \mathcal{P}$$

$$\sum_{\substack{j \in Ch(\bar{a}(u)) \\ \cap \bar{a}(v)}} \rho.\delta_j + \sum_{\substack{j \in Par(\bar{a}(u)) \\ \cap \bar{a}(v)}} \sum_{\substack{i \in Ch(j) \\ \cap \bar{a}(u)}} \rho.\delta_i \leq bp_{u,v} \qquad (5)$$

3. Complexity

Unsurprisingly, most operator mapping problems are NP-hard, because downloading objects with different rates on two identical servers is the same problem as 2-Partition [8]. Let us consider the simplest problem class, i.e., mapping a fully homogeneous left-deep tree application [10] (see Fig. 1(b)) without communication costs ($\delta_i = 0$), with objects placed on a fully homogeneous set of servers, onto a fully homogeneous set of processors. The objective function consists now in minimizing the number of used processors. It turns out that even this problem is NP-hard, due to the combinatorial space induced by the mapping of basic objects that are shared by several operators. Due to lack of space we refer the interested reader to [4] for the proof. It uses a reduction from 3-Partition, which is NP-complete in the strong sense [8]. Note that this problem becomes polynomial if one adds the additional restriction that no basic object is used by more than one operator in the tree. In this case, one can simply assign operators to $\lceil |\mathcal{N}| \times w/s \rceil$ arbitrary processors in a round-robin fashion.

Linear Programming Formulation: We also provide a formulation of the optimization problem as an integer linear program (ILP), but due to lack of space we refer the interested reader to [4].

4. Heuristics

In this section we propose several polynomial heuristics to solve the operator-placement problem. The code for all of them is available on the web [2]. Each heuristic works in two steps: (i) an operator placement heuristic determines the number of processors that should be acquired, and decides which operators are assigned to which processors; (ii) a server selection heuristic decides from which server each processor downloads all needed basic objects.

Operator Placement Heuristics – Note that in most of these heuristics, only the most powerful processors and network cards are acquired. However, these are later replaced by the cheapest ones that still fulfill throughput requirements. This is done just after the server selection step, as a third "downgrade" step, in a view to minimizing cost.

Random – While there are some unassigned operators, the Random heuristic picks one of these unassigned operators randomly, say *op*. It then acquires the cheapest possible processor that is able to handle *op* while achieving the required application throughput. If there is no such processor, then the heuristic considers *op* along with one of its children operators or with its parent operator. This second operator is chosen so that it has the most demanding communication requirements with *op* (in an attempt to reduce communication overhead). If no processor can be acquired that can handle both operators together, then the heuristic fails. If the additional operator had already been assigned to another processor, this last processor is sold back.

Comp-Greedy – The Comp-Greedy heuristic first sorts operators in non-increasing order of w_i, i.e., most computationally demanding operators first. While there are unassigned operators, the heuristic acquires the most expensive processor available and assigns the most computationally demanding unassigned operator to it. If this operator cannot be processed on this processor so that the required throughput is achieved, then the heuristic uses a grouping technique similar to that used by the Random heuristic (i.e., grouping the operator with its child or parent operator with which it has the most demanding communication requirement). If after this step some capacity is left on the processor, then the heuristic tries to assign other operators to it. These operators are picked in non-increasing order of w_i, i.e., trying to first assign to this processor the most computationally demanding operator.

Comm-Greedy – The Comm-Greedy heuristic attempts to group operators to reduce communication costs. It picks the two operators that have the largest communication requirements. These two operators are grouped and assigned to the same processor, thus saving costly communication between both processors. There are three cases to consider: (i) both operators were unassigned, in which case the heuristic simply acquires the cheapest processor that can handle both operators; if no such processor is available then the heuristic acquires the most expensive processor for each operator; (ii) one of the operators was already assigned to a processor, in which case the heuristic attempts to accommodate the other operator as well; if this is not possible then the heuristic acquires the most expensive processor for the other operator; (iii) both operators were already assigned on two different processors, in which case the heuristic attempts to accommodate both operators on one processor and sell the other processor; if this is not possible then the current operator assignment is not changed.

Subtree-Bottom-Up – This heuristic first acquires as many most expensive processors as there are al-operators and assigns each al-operator to a distinct processor. The heuristic then tries to merge the operators with their father on a single machine, in a bottom-up fashion (possibly returning some processors). Consider a processor on which one or more operators have been assigned. The heuristic first tries to allocate as many parent operators of the currently assigned operators to this processor. If some parent operators cannot be assigned to this processor, then one or more new processors are acquired. This mechanism is used until all operators have been assigned to processors.

Object-Grouping – For each basic object, this heuristic counts how many operators need this basic object. This count is called the "popularity" of the basic object. The al-operators are then sorted by non-increasing sum of the popularities of the basic objects they need. The heuristic starts by acquiring the most expensive processor and assigns to it the first al-operator. The heuristic then attempts to assign to it as many other al-operators that require the same basic objects as the first al-operator, taken in order of non-increasing popularity, and then as many non al-operators as possible. This process is repeated until all operators have been assigned.

Object-Availability – This heuristic takes into account the distribution of basic objects on the servers. For each object k the number av_k of servers handling object o_k is calculated. Al-operators in turn are treated in increasing order of av_k of the basic objects they need to download. The heuristic tries to assign as many al-operators downloading object k as possible on a most expensive processor. The remaining internal operators are assigned similarly to Comp-Greedy, i.e., in decreasing order of w_i of the operators.

Server Selection Heuristics – Once an operator placement heuristic has been applied, each al-operator is mapped on a processor, which needs to download basic objects required by the operator. Thus, we need to specify from which server this download should occur. For the Random heuristic, once the mapping of operators onto processors is fixed, we associate randomly a server to each basic object a processor has to download.

For all other heuristics, we use a more sophisticated heuristic, using three loops. The first loop assigns objects that are held exclusively by a single server. If not all downloads can be guaranteed, the heuristic fails. The second loop associates as many downloads as possible to servers that provide only one basic object type. The last loop finally tries to assign the remaining basic objects that must be downloaded. For this purpose, objects are treated in decreasing order of nbP/nbS, where nbP is the remaining number of processors that need to download the object, and nbS is the number of servers where the object still can be downloaded. In the decision process, servers are considered in decreasing order of the minimum between the remaining bandwidth capacity of the servers network card, and the bandwidth of the communication link.

Once servers have been selected, processors are downgraded if possible: each processor is replaced by a less expensive model that fulfills the CPU and network card requirements of the allocation.

5. Simulation Results

Simulation Methodology – All our simulations use randomly generated binary operator trees with at most N operators, which we vary. All leaves correspond to basic objects, and each basic object is chosen randomly among 15 different types. For each of these 15 basic object types, we randomly choose a fixed size. In simulations with *small object sizes*, in the $\delta_k \in [5, 30]$ MB range, whereas *large object sizes* are in the $\delta_k \in [450, 530]$ MB range. The download frequency for basic objects is either *low* ($f_k = 1/50s$) or *high* ($f_k = 1/2s$). Recall that the download rate for object o_k is then computed as $rate_k = \delta_k \times f_k$.

The computation amount w_i for an operator n_i (a non-leaf node in the tree) depends on its children l and r (basic object or operator): $w_i = (\delta_l + \delta_r)^\alpha$, where α is a constant fixed for each simulation run, and δ is either the size of the basic object, or the amount of data sent by the child operator. The same principle is used for the output size of each operator, setting for all simulations $\delta_i = \delta_l + \delta_r$. The application throughput ρ is fixed to 1 for all simulations. Throughout the whole set of simulations we use the same server architecture: we dispose of 6 servers, each of them equipped with a 10 GB network card. The 15 different types of objects are randomly distributed over the 6 servers. We assume that servers and processors are all interconnected by a 1 GB link. The rest of the platform can be purchased at the costs from Table 1 (configurations of Intel's high-end, rack-mountable server, PowerEdge R900).

Processor		
Performance (GHz)	Cost ($)	Ratio (GHz/$)
11.72	7,548 + 0	1.55 $\times 10^{-3}$
19.20	7,548 + 1,550	1.93 $\times 10^{-3}$
25.60	7,548 + 2,399	2.38 $\times 10^{-3}$
38.40	7,548 + 3,949	3.12 $\times 10^{-3}$
46.88	7,548 + 5,299	3.43 $\times 10^{-3}$

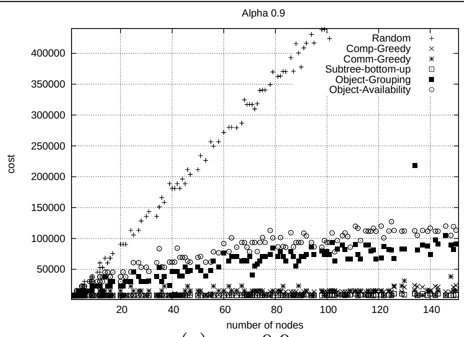
Network Card		
Bandwidth (Gbps)	Cost ($)	Ratio (Gbps/$)
1	7,548 + 0	1.32 $\times 10^{-4}$
2	7,548 + 399	2.51 $\times 10^{-4}$
4	7,548 + 1,197	4.57 $\times 10^{-4}$
10	7,548 + 2,800	9.66 $\times 10^{-4}$
20	7,548 + 5,999	14.76 $\times 10^{-4}$

Table 1. Platform costs (based on data from the Dell Inc. web site, as of March 2008).

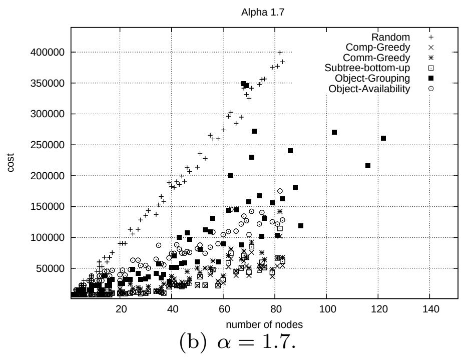
Results – Due to lack of space, we only present results for selected sets of significant experiments (see [4] for more results). In the first set of simulations, we study the behavior of the heuristics when the download frequency is high (1/2s) and object sizes small (5-30MB). Figures 2(a) and 2(b) show the cost as the number of nodes N in the tree varies, with a fixed computation factor α. As expected, Random performs poorly. Subtree-bottom-up achieves the best costs. All Greedy heuristics exhibit similar performance, poorer than Subtree-bottom-up. Perhaps surprisingly, the heuristics that pay special attention to basic objects, Object-Grouping and Object-Availability, perform poorly. With a larger value of α (cf. Figure 2(b)) the operator tree size becomes a more limiting factor. For trees with more than 80 operators, almost no feasible mapping can be found. However, the relative performance of our heuristics remains almost the same, with two notable features: a) Object-Grouping still finds some mappings for operator trees with up to 120 operators; b) Comp-Greedy performs as well as and sometimes better than Subtree-bottom-up when the number of operators increases.

Figure 3 shows the behavior of the heuristics when N is fixed and the computation factor α increases. Up to a threshold, the α parameter has no influence on the heuristics' performance. When α reaches the threshold, the solution cost of each heuristic increases until α exceeds a second threshold after which solutions can no longer be found. Depending on the number of operators both thresholds have lower or higher values. In the case of small operator trees with only 20 nodes, the first threshold is for α=1.7 and the second at α=2.2 (vs. α=1.6 and α=1.8 for operator trees of size 60, as seen in Figure 3). Subtree-bottom-up behaves in both cases the best, whereas Random performs the poorest. Object-Grouping and Object-Availability change their position in the ranking: for small trees Object-Grouping behaves better, while for larger trees it is outperformed by Object-Availability. The Greedy heuristics are between Subtree-bottom-up and the object sensitive heuristics.

With the same experimental setting but large object sizes (450-530MB), the results are similar except that no feasible solution can be found as soon as the trees exceed 45 nodes. In general, Subtree-bottom-up still achieves the best costs, but at times it is outperformed by Comm-Greedy. Subtree-bottom-up even fails in two cases (the server selection does not succeed be-



(a) $\alpha = 0.9$.



(b) $\alpha = 1.7$.

Figure 2. Simulation with high frequency and small object sizes, increasing N.

cause of bandwidth limitation), while other heuristics find a solution. Please refer to [4] for the detailed results. The behaviors of the heuristics with low download frequencies ($f_k = 1/50s$) are almost the same as for high frequency. In general the heuristics lead to the same operator mapping, but in some cases the purchased processors have less powerful network cards.

In another set of experiments, we study the influence of download rates on the solution. Recall that the download rate of a basic object k is computed by $rate_k = f_k \times \delta_k$. A first result is that frequencies smaller than $1/10s$ have no further influence on the solution. All heuristics find the same solutions for a fixed operator tree (see figures in [4]). For frequencies between $1/2s$ and $1/10s$, the solution cost changes. In general the cost decreases, but for $N = 160$ the cost for the Object-Grouping heuristic increases. Furthermore, the heuristic ranking remains: Subtree-bottom-up, followed by the Greedy family, followed by the object sensitive ones, and Random. Interestingly, the costs of Object-Availability de-

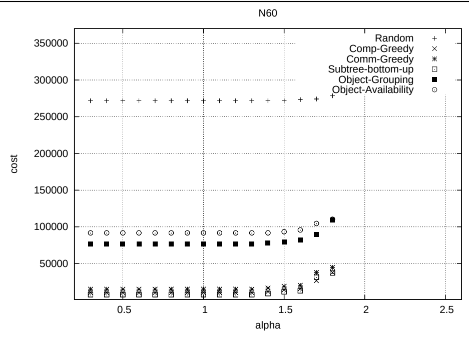


Figure 3. Simulation with high frequency and small object sizes, increasing α, $N = 60$.

crease with the number of operators. In this case the number of operators that need to download a basic object increases, and hence the privileged treatment of basic objects in order of availability on servers becomes more important. We conclude that the level of replication of basic objects on servers may matter for application trees with specific structures and download frequencies, but that in general we can consider that this parameter has little or no effect on the heuristics' performance.

The last set of experiments is dedicated to the evaluation of our heuristics versus a lower bound given by the solution of our ILP. We use the commercial Cplex 11 solver to solve our linear program. Unfortunately, the ILP is so enormous that, even when using only 5 possible groups of processors and using trees with 30 operators, the ILP description file could not be opened in Cplex. For trees with 20 operators, Cplex returns the optimal solution, which consists in all cases in buying a single processor. Therefore, we decided to compare the heuristic solution with the optimal solution only in a homogeneous setting, in which there is only a single processor type. In this case we can skip the downgrading step after the server allocation step. Both for α values lower and higher than 1, Subtree-bottom-up finds the optimal solution in most of the cases. The same ranking of the heuristics holds in the homogeneous setting: Subtree-bottom up, the Greedy family, followed by Object-Grouping, Object-Availability and finally Random. Focusing on the Greedy family, we observe that in most cases Comm-Greedy achieves the best cost.

Summary of results – Results show that all our more sophisticated heuristics perform better than the simple random approach. Unfortunately, the object sensitive heuristics, Object-Grouping and Object-Availability, do not show the desired performance. We believe that in some situations these heuristics could lead to good performance, but this is not observed on our set of random application configurations. We have found that Subtree-bottom-up outperforms other heuristics in most situations and also produces results very close to the optimal (for the cases in which we were able to determine the optimal). There are some cases for which Subtree-bottom-up fails. In such cases our results suggest that one should use one of our Greedy heuristics.

6. Conclusion

In this paper we have studied the problem of resource allocation for in-network stream processing, with the objective of minimizing the platform cost. We have formalized the operator-placement problem. The complexity analysis showed that all problems are NP-complete, even for the simpler cases. We have derived an integer linear programming formulation, and we have proposed several polynomial time heuristics. We have compared these heuristics through simulation and we have assessed the absolute performance of our heuristics with respect to the optimal solution of the linear program for homogeneous platforms and small problem instances. The Subtree-bottom-up heuristic almost always produces optimal results and almost always outperforms the other heuristics.

An interesting direction for future work is the study of the case when multiple applications must be executed simultaneously so that a given throughput must be achieved for each application. In this case a clear opportunity for higher performance with a reduced cost is the reuse of common subexpressions between trees [14, 13]. Another direction is the study of applications that are mutable, i.e., whose operators can be rearranged based on operator associativity and commutativity rules [5].

References

[1] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

[2] Source Code for the Heuristics. http://graal.ens-lyon.fr/~vsonigo/code/query-streaming/.

[3] B. Badcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the Intl. Conf. on Very Large Data Bases*, pages 456–467, 2004.

[4] A. Benoit, H. Casanova, V. Rehn-Sonigo, and Y. Robert. Allocation Strategies for Constructive In-Network Stream Processing. Research Report 2008-20, LIP, ENS Lyon, France, June 2008.

[5] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries. In *Proceedings of ICDE*, 2002.

[6] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. of the CIDR Conf.*, January 2003.

[7] E. Cooke, R. Mortier, A. Donnelly, P. Barham, and R. Isaacs. Reclaiming Network-wide Visibility Using Ubiquitous End System Monitors. In *Proceedings of the USENIX Annual Technical Conf.*, 2006.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[9] B. Hong and V. K. Prasanna. Adaptive allocation of independent tasks to maximize throughput. *IEEE Trans. Parallel Distributed Systems*, 18(10):1420–1435, 2007.

[10] Y. E. Ioannidis. Query optimization. *ACM Computing Surveys*, 28(1):121–123, 1996.

[11] J. Kräme and B. Seeger. A Temporal Foundation for Continuous Queries over Data streams. In *Proceedings of the Intl. Conf. on Management of Data*, pages 70–82, 2005.

[12] D. Logothetis and K. Yocum. Wide-Scale Data Stream Management. In *Proceedings of the USENIX Annual Technical Conference*, 2008.

[13] K. Munagala, U. Srivastava, and J. Widom. Optimization of continuous queries with shared expensive filters. In *PODS '07: Proc. of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2007.

[14] V. Pandit and H. Ji. Efficient in-network evaluation of multiple queries. In *HiPC*, 2006.

[15] P. Pietzuch, J. Leflie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, 2006.

[16] U. Srivastava, K. Munagala, and J. Widom. Operator Placement for In-Network Stream Query Processing. In *Proceedings of the 24th Intl. Conf. on Principles of Database Systems*, 2005.

[17] R. van Rennesse, K. Birman, D. Dumitriu, and W. Vogels. Scalable Management and Data Mining Using Astrolabe. In *Proceedings from the First Intl. Workshop on Peer-to-Peer Systems*, 2002.

Acknowledgment We thank the reviewers for their comments and suggestions. This work was supported in part by the ANR StochaGrid project.