Automated Operator Placement in Distributed Data Stream Management Systems Subject to User Constraints

Cory Thoma, Alexandros Labrinidis, Adam J. Lee

Department of Computer Science, University of Pittsburgh
{corythoma, labrinid, adamlee}@cs.pitt.edu

***Abstract*—Traditional distributed Data Stream Management Systems assign query operators to sites by optimizing for some criterion such as query throughput, or network delay. The work presented in this paper begins to augment this traditional operator placement technique by allowing the user issuing a continuous query to specify a variety of constraints—including collocation, upstream/downstream, and tag- or attribute-based constraints—controlling operator placement within the query network. Given a set of constraints, operators, and sites; four strategies are presented for optimizing the operator placement. An optimal brute force algorithm is presented first for smaller cases, followed by linear programming, constraint satisfaction, and local search strategies. The four methods are compared for speed, accuracy, and efficiency, with constraint satisfaction performing the best, and allowing assignments to be adapted on the fly by the DDSMS.**

I. Introduction

Distributed data stream management systems (DDSMS), much like distributed database management systems, require certain operators to be placed on certain sites in order to satisfy the query. Traditional operator placement in DDSMS have focused on solving the problem of finding the optimal assignment for each node based on optimizing some fitness function [1] [2] [3]. The optimization algorithms honor system constraints where an operator must be collocated with a source stream, but otherwise focus on placing operators based on the fitness function. The work presented in this paper explores the addition of *querier* constraints into the placement process. The user provides a set of constraints to limit the placement of specific operators onto specific sites. These constraints give the user the ability to explicitly place an operator onto or off of a site, or to implicitly place an operator by constraining it against other operators in the network. Other than giving a set of constraints, the user has no other interaction with the placement, leaving actual assignments to the automated solver explored in this paper.

Consider the case where three servers are running at three different cities, Atlanta, Boston, and Calgary, and a user issues a distributed query to them. One server is placed in Atlanta and contains private data, and the other two are placed in Boston and Calgary respectively, and do not contain any private data. In order to make sure private data is not leaked, the user would like to make sure no JOINS happen at Boston or Calgary with data from Atlanta, and that there is no operator from Atlanta upstream from any other city. An ideal query plan would resemble Figure 1. Note Atlanta does not send data to the other cities, and therefore the user's constraints are met. Clearly, in such a small example it is trivial to find a good solution. However, one can clearly see that this would not be straight-forward for the general case. This is exactly the problem we address in this paper. In particular, we make the following contributions:

1) We identify several different, yet important, types of constraints a user will use in a DDSMS, and formalize their inputs.
2) We explore different approaches for satisfying the constraints with a fitness function.
3) We identify a Constraint Satisfaction Solver which outperforms the other approaches tested and evaluate its performance on larger datasets.

Road map The rest of this paper is structured as follows. In Section II, the process for creating and enforcing constraints is introduced. Section III presents all of the approaches considered (brute force, constraint satisfaction, linear programming, and satisfiability solvers). Section IV evaluates the four solvers, and details the best one, Section V presents related work, and Section VI concludes.

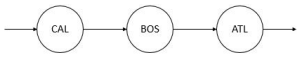


Fig. 1. Ideal Plan where Atlanta is far downstream.

II. Problem Statement

A. Assumptions

The main focus of our work is operator placement in a query plan, and as such, constraints deal with operator assignment to sites. Constraints deal directly with *where* operators should be placed within the query network. For the remainder of this paper, we assume that the query plan is provided with operators requiring system input streams having already been placed, and the inter-operator streams and dependencies have already been decided. Therefore, the only task that remains is the assignment of the remaining operators to sites.

978-1-4799-3481-2/14/$31.00 © 2014 IEEE

We also assume that each site has a descriptor which contains essential information such as the computational capacity of the system, the transmission rate for the network, the operators already placed onto the site, tags users have added to the descriptor, as well as location, pricing, and organization information. Operators have similar descriptors which provide details regarding the cost to execute the operator, the selectivity, the streams in and out, the operator type, source or sink, and a set of user defined tags. These descriptors are used to ensure physical limitations are not broken and user constraints can be satisfied.

B. Operator Constraints

A user may identify many ways they wish to constrain their query. For example, a user may wish to stack as many operators on their own machine as possible to avoid paying for server time. Users may wish to ensure no sensitive data is transmitted from one source to another. Users may also wish to ensure that a select and a join happen on the same node. This section provides an overview of the constraints considered and implemented in this work, as well as an input specification for each constraint.

System Constraints These constraints are derived from the characteristics of the system. Each site can only supply a certain amount of computational resources, and therefore operators can only be assigned up to the maximum available resources for each site. Network connections and transmission rates are also defined by the system and therefore must be constrained as operators are assigned. Finally, since most raw data streams inputs are associated with a source site, those operators must remain on their given site. These constraints are given the highest priority, and must always be satisfied. System Constraints are derived from system information in the input file. Constraints of this type are also in Stanoi et al [4]. A system administrator may also impose constraints on the system, which should be considered as part of the system set up. System constraints are applied to all queries in the system, unless specified otherwise by the administrator. Note that we expect to have a mix of system constraints (specified by an administrator) together with other types of constraints (specified by users submitting queries); operator placement should then consider all the constraints specified.

Collocation Collocation (referred to as location) constraints are used to either put two operators on the same site, or ensure they are on different sites. Collocation constraints are represented as an equality or a non-equality, meaning that the site variable in the descriptor of one operator must equal or not equal the site variable in the descriptor of the other. The same syntax is used to ensure collocation (or separation) between a site and an operator. If an operator must be on a site, then it should be collocated with it. For two operators o_1 and o_2 a typical constraint would be $o_1.site = o_2.site$ or $o_1.site$!= $o_2.site$. For the case when an operator needs to be assigned to a specific site, instead of $o_1.site$!= $o_2.site$, it is simply $o_1.site = s_i.site$ or $o_1.site$!= $s_i.site$.

Upstream and Downstream Since we know the interconnections of operators, we may wish to restrict the flows which come into or out of a given site. For example, if the user does not want any information from Atlanta going to Boston, they can constrain that Boston never be downstream from Atlanta. To ensure the constraint is held, we must guarantee no stream from an operator assigned to Atlanta ever feeds a stream assigned to Boston, which requires traversing each path leaving Atlanta. Upstream constraints are encoded with $s_i \backslash\backslash s_j$, and downstream constraints with $s_i // s_j$. An upstream constraint is simply the inverse of a downstream constraint, but both options are provided for simplicity to the user.

Tag-Based Included in every descriptor is a set of tags. These tags are used to provide information not already captured in the descriptor such as typical load, owner, uptime, maintenance calls, and many other types of tags. If a user wishes to make sure an operator is never placed on a site with a certain tag, they simply check to see if the tag is listed in the descriptor with an equality constraint. Tag constraints simply take the tag as a string and are encoded by "tag in s_i" or "tag !in s_i".

Attribute-Based The final supported constraint relies on the attributes of the site as provided in the descriptor. If a user wishes to ensure their operator will not end up on a slow machine, for example, they can constrain the operator to only be assigned to a site if the site's speed is greater than some threshold. We do not support aggregate constraints at this point, where the user may want to ensure a machine's total utilization is above some threshold, but it is part of the future work. Typical attribute constraints are encoded with the mathematical operators $>, <, =, !=$. An example would be "Network_Transmission $>$ .5".

Global Aggregate Constraints Global aggregate constraints apply to a complete plan. A typical constraint of this type would involve aggregate information collected once each operator has been assigned. For example, if the user wants to ensure the total computational load of a site is only 75% of its capacity, the full plan would be required as all operators must be placed to ensure no other system constraints are violated (if all of the sites were nearly full, this constraint may not be satisfiable, but we wont know until all operators are placed). The current implementation does not support these types of constraints, but should be included in future work.

All of the constraint types are summarized in Table III. In order to express a constraint, one simply concatenates two applicable strings with a constraint in the middle. If the strings exist in the system, the constraint is accepted, else it is rejected. These five types of constraints cover operator assignment related to the site and to other operators, as well as the flow of the sites with respect to other sites. Returning to the opening example, a user could state that Boston or Calgary never be downstream of Atlanta, or they could constrain Boston and Calgary to be upstream of Atlanta. To show exactly how these constraints can be used, consider the query network in Figure 2. The "AGGR" functions are aggregates, each with an associated window and slide. There are also two selects

Constraint	Syntax	Applicable Strings	Example
Stream	\\, //	Sites	Calgary // Boston
Attribute-Based	<, >, <=, >= ,=,!=	Sites.attribute	Select1.selectivity >= .9
Tag-Based	in, !in	Sites.tag, Operator.tag	Calgary.tag != closed
Global Aggregation	<, >, <=, >= ,=,!=	global attribute (not supported)	GlobalSelectivity >= .9
Colocation	=, !=	Operator, Site	Select1=Atlanta

TABLE I
SYNTAX AND EXAMPLES FOR EACH TYPE OF CONSTRAINT

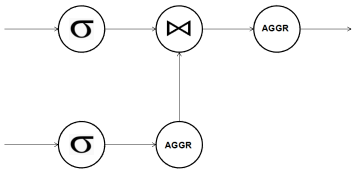


Fig. 2. Simple Query Network

and a join. The join can be expensive, and one may wish to ensure its computation would remain on a powerful server. An equality constraint can be used to satisfy this. Shipping data can also be expensive, and sending the entire results of the select to the join could be costly. One may wish to keep them on the same node to reduce this cost, and a location constraint (or equality constraint over the site) can be used to ensure this. Finally, data being shipped from the aggregate to the join may contain sensitive data which one may not want to reveal to a certain server. In order to ensure that the server never gets the data, a stream constraint can be used to ensure that that site is never downstream from the site containing the join. Again, these constraints assume we have a query plan provided to the solver, and in future work, we wish to extend these constraints to a higher level policy which maps to the work we have presented here.

III. Approaches for Finding Operator Placement

The authors in [4] state that finding an optimal placement of operators in a DDSMS is NP-hard. If the size of the query plan is large, the time it takes to produce a plan will become prohibitive. In this section we explore different options for finding optimal operator placement that would still run in a reasonable amount of time.

A. Brute Force Algorithm

The first algorithm used is a simple brute force approach, and is used as a baseline for the other approaches. Every possible configuration is considered, so the result is guaranteed to be optimal. The first step is to identify all of the operators which have no system constraints (provided) over them. The next step is to create all of the combinations of assignments of operators to sites. Once the list is created, each combination is tested to make sure that it satisfies all of the system constraints (e.g. a machine can not be assigned more than it can handle), and all of the user constraints. If the plan successfully satisfies them, it is scored by a fitness function described below. After iterating through all possible combinations, the optimal solution(s) are given, and the one with the lowest cost is selected.

The fitness function is a simple one. Each machine has associated with it a speed, and a network transmission cost, relative to a standard reference machine. Each operator's cost is multiplied with its site's speed, and the sum of the products is calculated to get the computational time on each machine. Then, every time an operator sends data to another site, the minimum of the sending site's and the receiving site's transmission rate is multiplied with the link speed to get the network delay time. This delay is added to the overall cost and the cost is associated with the plan and stored.

As one would expect, the brute force approach yields a runtime of $O(o^s)$ where o is the number of operators and s is the number of sites. For o and s sufficiently large ($o = 15$, $s = 10$), the solver would take more than a half hour to return the result, making this approach unrealistic. However, since it yields the optimal solution, it is still a very good baseline to compare the other approaches to. The brute force approach would yield the best result if the query plan was small, and the user did not mind a delay. however, we can not take queries running on the system down to optimize them for a long period of time, as this will seriously impact the performance of continuous queries, which are typically used by monitoring applications. However, if the user is willing to wait a given length of time, they could use the brute force approach.

B. Constraint Satisfaction Approach

The first non-brute force approach tested is a natural choice, namely, a constraint satisfaction problem (CSP) solver. Given a set of constraints, a set of operators, a set of sites, and a fitness function, the solver attempts to place each operator without violating the constraints, while optimizing for the fitness function. The output produced by the solver is a complete and near optimal solution. The solver we chose to use is the open source JaCoP program [5]. JaCoP provides many types of constraints and many types of search strategies. JaCoP also allows for optimization functions to be taken over the entire problem. One key feature of JaCoP used in this work is the conditional constraints, which allow our solver to check up/down stream constraints at each assignment.

In order to check the optimality of the CSP approach, the same small cases used in the brute force approach were run on the CSP approach, and for all small cases, the same result was produced. This leads one to believe that the CSP will find

a near optimal solution for larger problems as well. The CSP solver is comprised of three parts. In order to guarantee global system constraints (such as making sure no site is overloaded beyond its capacity), JaCoP provides a bin-packing constraint. For the purposes of operator placement, the bins represent the sites, and their sizes represent their computational capacity. The items represent the operators, and each size represents its cost in terms of computation.

Equality and location constraints are easy. Operators which can not be collocated with one another are simply constrained with a primitive "not equals" constraint, so that they are not put into the same bin. Operators which need to be on a certain site are simply put into the correct bin. Streaming constraints are the most robust, and therefore the most difficult to implement. If a stream constraint is given to the system, the system uses the query network to determine which operators could potentially violate the constraint. For example, if there is an upstream constraint saying Atlanta can not be upstream from Boston, every operator is paired with all operators that are upstream from it, then a constraint is added which simply states (assuming operator$_x$ is upstream from operator$_y$ in the query plan) "if operator$_x$=Atlanta then operator$_y$!= Boston". In the worst case, there will need to be $o!$ constraints added to the system, where o is the number of operators.

The CSP approach satisfies all of the constraints, as well as producing a near optimal result (since optimality is not guaranteed). In Section IV, we will revisit the CSP solver.

C. Satisfiability Approach

Another approach is to view the automated operator placement problem as a satisfiability problem. However, there is one major drawback to the satisfiability approach in that it is just a satisfaction of the constraints, and not an optimization. Given this drawback, it would be unjust to directly compare the CSP approach (which can give a near optimal result) to the satisfiability approach (which has no optimization at all), so we simply focus on satisfaction-only constraints, without needing optimization. This approach utilizes the open source satisfiability modulo theory (SMT) solver yices [6]. Yices takes as input a list of variable definitions, a list of assertions (x=y), and finally a command to satisfy the constraints. A typical input may resemble:

```
(define x::int (subrange 0 2))
(define y::int (subrange 0 2))
(define z::int (subrange 0 2))
(assert (= z 1))
(assert (/=x 0))
(assert (= x y))
```

This simple example provides insight into how yices can be used to place operators subject to user constraints. Here, each variable definition is an operator. The subrange represents the sites on which each operator can be placed. So in this example, we have operators x, y, and z and sites 0, 1, and 2. Then we assert that operator z must be assigned to site 2 (like assigning a select operator to the Atlanta location), and we assert operator x can not be placed on site 0 (like placing a join at site Boston). Finally, a constraint where the operator x must be the same as the operator y, or that they must be collocated (or easily /= for disjunction). In simple terms, we have shown the collocation and equality constraints are supported. This leaves upstream, downstream, and tag-based constraints. Tag-based constraints are handled via preprocessing. If a site has a tag, it is converted to an (assert ($\bar{z}$ site)) constraint. This preprocessing is done in one simple loop, taking no more steps than the number of tag constraints.

The final constraint type is the upstream and downstream constraints. These constraints require verification each time an assignment is made. For example, if we want to ensure Atlanta is never upstream from Boston, then each assignment would check to make sure this constraint is not violated. The first assignment would skip this check, and then from the second assignment on, the graph would be checked to ensure that the new assignment did not set Atlanta upstream from Boston. Yices does not handle this type of dynamic constraint, but we can take advantage of knowing the query network. Much like in the CSP approach, we know all potential upstream conflicts, since we know which operators are upstream from one another in the query network.

Using this knowledge, we can enumerate all of the possible conflicts. Consider the "Atlanta not upstream from Boston" example. We know when a given operator is upstream from another since we are given the query network as input. Consider all pairs of nodes where node x is upstream from node y. We can enumerate all possible pairs of upstream nodes and make a constraint which would say "if operator $o_1 == x$ and operator $o_2 == y$ then o_1!= x" where x and y are an upstream pair, and o_1 and o_2 are the constraint pairs, which can handle the upstream problem. Yices allows us to do just this with an if-then-else command. We can write "(if (= x 1 and =y 2) /=x 1)". Each possible combination is listed in a preprocessing state, and then it is satisfied in the satisfaction stage. Preprocessing is done by a Perl script which prints a yices script and calls the yices program on that input.

The setup for these experiments is straightforward. Constraints are added to the system, yices is used, and the runtime is collected. The graph type (i.e. query network) is held constant. Of course, the final output of the Yices is not optimized. It also does not allow a global constraint on the limit of operators onto one site, or how much a site can handle (a count constraint). The solver does honor every constraint if they are all satisfiable, and can be used as a first step to an optimization solver. Most results are produced in under a second, so it is quick as well.

D. Linear Programming Approach

At first glance, a linear programming approach seemed as if it would be the best for the operator placement problem. It allows for an optimization formula and almost all types of constraints. The only types of constraints that the linear programmer could not efficiently handle are the system constraints. Using a system of equations, one can not constrain the number of times a value in the range was selected, let alone assign

	Plan Execution Cost (× Brute Force)	Std. Dev.	Plan Generation Time(ms)	Std. Dev.
Brute Force	1.00	.15	1,560,235	208421.20
CSP	1.07	.19	412	89.12
LPS	2.29	.11	652	42.50
SMT	3.07	.09	212	34.83

TABLE II
RUNTIMES FOR EACH APPROACH ON THE SAME QUERY, WITH THE FINAL ASSIGNMENTS' COST.

a mapping which would allow operator costs to be summed under a sites total computational power. The main problem resides in how a problem is represented. Each solution is simply a set of integers satisfying the system of equations, and the entirety of the solution is not known until the end. Since we can not alter a solution until all assignments are made, we can not eliminate solutions which violate stream constraints, since all assignments must be known. This also hindered the ability to determine network costs, since a whole set of assignments must be known before the cost can be calculated. Our initial work in linear programming used Matlab. The Matlab script takes as input an optimization function, and then a system of linear equations which act as the constraints. Much like in the yices, the operators are the variables, and their domain represents the sites. The next input involves instantiating each operator: $x_1 = 0, x_2 = 0 \ldots$ until each operator is instantiated. The last input involves constraining the inputs. To add sites, we simply limit the domain of each operator to the number of sites. Then, a simple $x_i = (!=)i \in \mathbb{Z}$ or $x_i = (!=)y$ where y is an operator will give us all of the equality and collocation constraints. Tag constraints are again preprocessed by a Perl script to make them an equality (or inequality) constraint, leaving only the upstream and downstream approaches to be added. In Matlab, these are more difficult.

Matlab allows for an "if" statement to be used in a linear problem, so we will take advantage of it. In the optimization step, it appears the expressions are checked at each assignment, and not after the entire script is run. Using our technique from the CSP and SMT, we enumerate each possible conflict and add it as input to the system. The variable assignment will be checked for each upstream pair and if the assignment violates the upstream pair, we do not allow the upstream node to remain upstream, or the downstream node to remain downstream (with a 50% chance of one happening). Since Matlab can not score an assignment and test the result before all assignments are made, we must randomly chose which to move, meaning that the result may not be the actual optimal result. A typical if statement is simply:

```
if( x=constrainedSite)&(y = constrainedSite2){
x~=constrainedSite; }
```

Here two sites are constrained such that one is not upstream of the other (constraiedSite // constraintedSite2). Since x is assigned to a constrained site and y is assigned to the other site, one needs to be moved, and in this case, x is moved from its assigned site. Adding all of the possible statements to the Matlab script will provide the optimal result given the random assignments to avoid upstream and downstream conflicts.

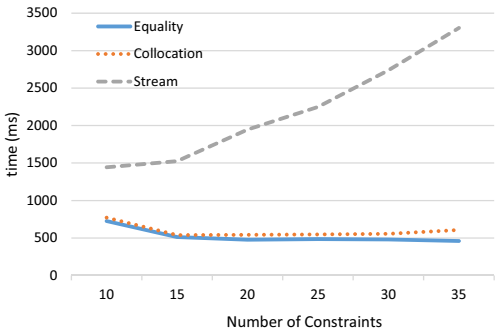
There are some obvious drawbacks with the Linear Programming approach. The optimization has no way of recognizing a graph structure separate from the linear system at the current moment, so it can not include network costs in its optimization function. It also requires preprocessing like Yices, differing only in the optimization function, and the optimization is not guaranteed to be optimal since upstream/downstream conflicts are resolved by randomly moving one. Finally, it may not produce an optimal result, given a modest cost function of speed of the operator multiplied by the cost of the site to which it was assigned.

IV. EVALUATION

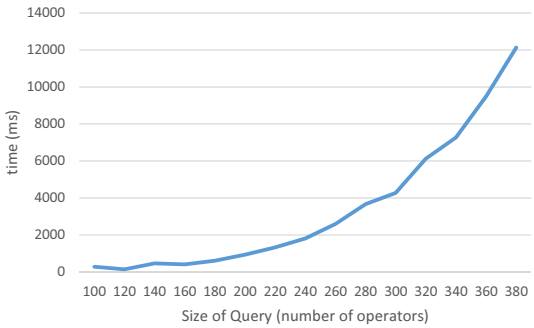
The four approaches listed in Section III each have their advantages and disadvantages. In this section, we compare and contrast the approaches. Each approach (brute force, CSP solver, LPS, and the SMT solver) can be generally described by the three categories of speed, number of constraints satisfied, and the optimality of the final satisfaction. Recall that finding optimal placement of the operators is exponential if all cases are considered, hence the brute force approach is the slowest by a large margin. The CSP is quick and satisfies all of the constraints, but is not guaranteed to be optimal. LPS fails to satisfy all of the constraints, and SMT fails to produce the optimal answer.

Table III gives more detail on which constraints each approach can satisfy. Obviously, the brute force approach can satisfy all of the constraints, since it considers all possibilities. The CSP can also satisfy all of the approaches, but may not be optimal. The SMT solver can satisfy all of the constraints so long as stream and tag constraints are preprocessed. The SMT also does not allow for network cost optimality since it does not yield an optimal solution, but rather a complete assignment of all operators. LSP can satisfy all of the primitive constraints, but can not guarantee a site will not be overloaded, nor can it identify network costs. In order to test the approaches, the following experiments were run on a workstation with a 2.53GHz dual core processor with 4GB of RAM.

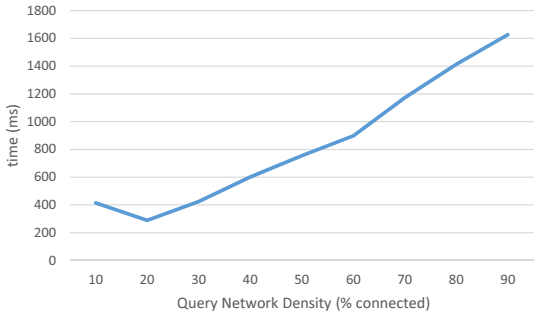
Experiment 1 This experiment focused on comparing the runtimes of each approach with the fitness function score of the final output. The experiment was run over 5 randomly connected query network with 20 sites and 40 operators, and the averages are reported. Each trial was run once since the brute force, LSP, and SMT solver will always return the same result (per query). Table II displays the results. Although optimal, the brute force approach took nearly half an hour



(a) Time to generate operator assignment vs. Size of Query in CSP approach



(b) Time to generate operator assignment vs. Size of Query in CSP approach



(c) Time to generate operator assignment vs. Query Network Density in CSP approach

Fig. 3. Experimental Results

	Brute Force	CSP	LPS	SMT
Operator (in)Equality	Y	Y	Y	Y
Operator to Site (in)equality	Y	Y	Y	Y
Stream Constraints	Y	Y	Y*	Y*
Tag	Y	Y	Y*	Y*
Global Site Capacity	Y	Y	N	Y*
Global Network Optimization	Y	Y	N	N

TABLE III
* REQUIRES PREPROCESSING ON EXPONENTIAL INPUT SIZE.

to return the assignment. The SMT and LPS were both quick, but their results were further from the optimal assignment. The CSP approach does not return the optimal assignment, but returns one which is close, and does so in *far* less time than the brute force approach. Combining this experiments results with Table III lead us to believe a constraint satisfaction solver is the best approach. The CSP generates operator assignments which may not be optimal, but are at least going to provide a good result. CSP also satisfies all of the constraints, both user and system. Since it does not consider all of the possible configurations, CSPs can find a solution quickly. In order to test exactly how quick the CSP can find an answer, three more experiments were attempted. Each experiment was the average of 5 iterations.

Experiment 2 The second experiment alters the number of constraints over the same query plan, and measures its runtime. Figure 3(a) displays the results. As the number of stream constraints increases, so does the amount of time it takes to process and satisfy them. This is simply due to the number of constraints that must be considered. Recall that the number of added constraints in the worst case $o! \times c$ where c is the number of constraints, so as c and o get bigger, we can expect the time to increase. The equality and location constraints have an opposite effect. The more constrained the assignments, the lower the number of states to be considered, and therefore a quicker execution. In the brute force approach, there was a dramatic difference in runtimes when increasing the number of location and equality constraints, but since the CSP does not consider them all, it is not as dramatically affected.

Experiment 3 The third experiment compared the size of the query network to the runtime, over the same percentage of constraints. For example, the query network ranges in the number of sizes and nodes, and 20% of the operators are constrained (7% equality, 7% location, and 6% stream constraints). In order to ensure the query plan remained similar as the size grew, operators were interconnected with a 50% probability, and their loads were scaled randomly between 1 and 5. Figure 3(b) displays the results. As expected, as the problem grows larger, so does the runtime. With 380 operators (with half as many sites) the runtime was just over 12s.

Experiment 4 The final experiment focused on the streaming constraints relative to the density of the graph. For this experiment, the number of sites and operators was fixed, but they were connected to other operators with some probability.

	Quick	Constraints	Optimal
Brute Force	**NO**	YES	YES
CSP	YES	YES	NO
LPS	YES	NO	MAYBE
SMT	YES	YES	NO

TABLE IV
PROPERTIES OF THE TESTED APPROACHES.

There were 50 sites and 100 operators, and of the 50 sites, 15 were constrained with an upstream or downstream constraint. Figure 3(c) displays the results. The y-axis represents the percentage of operators a given operator is connected to. Again, as expected, the more dense the query network, the longer the runtime. The runtime is not affected by increasing density for equality and location constraints.

Combining the results of the three experiments shows that a dense, highly constrained (where most constraints are stream constraints), large network is the hardest to satisfy. Easier loads would include large networks with a large number of equality and location constraints, or just smaller networks.

V. RELATED WORK

(I,A)-Privacy. The authors in [7] and [8] provide extensive work into ensuring user privacy in distributed database management systems. They investigate the location at which an operator is executed with regards to the data needed to preform that operator at that location. A user's intension can be realized if data collected and sent from one site to another (in order to preform an operation at that site) reveals what the true meaning of the user is. In order to keep the intension private, the authors propose (I,A)-privacy, where a user reserves some part of their query as an intensional region, and the adversary is to never learn anything in that region. In order to keep the intensional region private, the authors propose extensions to SQL which allow the user to constrain part of their queries so that operators execute on sites which are approved by the user. This keeps their intensions private by only moving data from site to site if intensions are not revealed. Our work differs in that we include more constraint types, and operate on continuous queries. We also focus on a complete query plan, whereas their work focuses on building a plan around a set of constraints.

WhiteWater. The authors in [4] provide an optimization technique for placing operators in a DDSMS. The authors differentiate their work from related work by being the first to use throughput as a means for measuring the effectiveness of a query network. The authors solve the problem of operator placement using a constraint satisfaction solver utilizing simulated annealing. The authors therefore have the ability to hard code some constraints into the system and allow the rest to be chosen by the CSS. Our work differs from theirs in that we allow the user to define the constraints to be satisfied and then allow them to move to better optimize.

Borealis Borealis [2] uses a two step protocol [1] in distributing operators to nodes. An initial phase is used to gather statistics on load and usage. A pair-wise algorithm is then used to set the initial configuration to distribute the load to each node. In the initial phase, operators are ordered by load, and the largest is paired with the smallest in recursive fashion until all are assigned. At runtime, each paring is compared for to see if the difference in load is above some threshold, and if so, the load is migrated between two nodes (sites). Our work differs from [1] in that constraints are considered as first class citizens, followed by an optimization.

VI. CONCLUSION

In this paper we introduced user preferences to operator placement in DDSMS. User preferences are encoded through the use of constraints over which sites an operator can be assigned to. Constraints at the query plan level allow the user to ensure that certain operators remain under their control though the optimization process. We introduced four techniques for finding an assignment; brute force, a constraint satisfaction solver, a SMT solver, and a linear programming solver. Table IV summarizes our results. Through implementation and evaluation, we have shown the CSP solver to be the most apt for the operator assignment problem. Given its lower runtime, it is feasible to use it in real time to adapt and changes in user preferences or the environment to re-optimize under the new settings.

As of now, queries are handled on an equal basis and are serviced at random. Future work would include assigning priorities to queries so system constraints are not an issue until lower priority queries, where they would simply be unsatisfied, and the best solution would be used. In general, the idea of using user constraints to place operators is not a problem unique to streaming applications, and can be generalized to placing processing engines into a distributed setting. The work proposed in this paper is unique to streaming applications as selectivities and the query plan help guide the placement. Streams also introduce their own class of constraints, which we begin to explore with the upstream and downstream constraints. In the future, we hope to incorporated this constraint-aware optimizer into a DDSMS to evaluate its performance.

ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation under awards CNS-1253204 and IIS-0746696.

REFERENCES

[1] Y. Xing, S. Zdonik, and J. Hwang, "Dynamic load distribution in the borealis stream processor," in *ICDE 2005*. IEEE, 2005, pp. 791–802.
[2] D. Abadi *et al.*, "The design of the borealis stream processing engine," in *CIDR*, 2005.
[3] T. Pham and A. Labrinidis, "A practical load manager for data stream management systems," *SMDB 2012*, 2012.
[4] I. Stanoi, G. Mihaila, C. Lang, and T. Palpanas, "Whitewater: distributed processing of fast streams," *TKDE*, vol. 19, no. 9, pp. 1214–1226, 2007.
[5] JaCoP.com, "Jacop," http://jacop.osolpro.com/.
[6] Yices.com, "Yices," http://yices.csl.sri.com/.
[7] N. Farnan, A. Lee, P. Chrysanthis, and T. Yu, "Don't reveal my intension: Protecting user privacy using declarative preferences during distributed query processing," *ESORICS*, pp. 628–647, 2011.
[8] ——, "Paqo: Preference-aware query optimization for decentralized database systems," in *ICDE*. IEEE, 2014.