Efficient Dynamic Operator Placement in a Locally Distributed Continuous Query System

Yongluan Zhou, Beng Chin Ooi, Kian-Lee Tan, and Ji Wu

National University of Singapore

Abstract. In a distributed processing environment, the static placement of query operators may result in unsatisfactory system performance due to unpredictable factors such as changes of servers' load, data arrival rates, etc. The problem is exacerbated for continuous (and long running) monitoring queries over data streams as any suboptimal placement will affect the system for a very long time. In this paper, we formalize and analyze the operator placement problem in the context of a locally distributed continuous query system. We also propose a solution, that is asynchronous and local, to dynamically manage the load across the system nodes. Essentially, during runtime, we migrate query operators/fragments from overloaded nodes to lightly loaded ones to achieve better performance. Heuristics are also proposed to maintain good data flow locality. Results of a performance study shows the effectiveness of our technique.

1 Introduction

In many emerging monitoring applications (e.g. network management, sensor networks, financial monitoring etc.), data occurs naturally in the form of active continuous data streams. These applications typically require the processing of large volumes of data in a responsive manner. In order to scale up the volumes of streams and queries that can be processed, a distributed stream processing system is inevitable. However, as the properties of data streams (e.g., arrival rates) and the processing servers' load are hard to predict, the initial placement of query operators may result in unsatisfactory system performance. The problem is exacerbated by multiple continuous queries that run long enough to experience the changes in the environment parameters. As such, any suboptimal performance will persist for a long time.

Clearly, a distributed stream processing system must adapt to changes in environment parameters and servers' load. We believe a dynamic load management scheme is indispensable for the system to be scalable. In particular, we expect aggressive methods such as query operator migration during runtime to bring long term benefit (especially for long running continuous queries) even though they may incur some short term overhead. The necessity of dynamic load management for a scalable distributed stream processing system has also been identified in previous work [7,11]. However, to date few complete and practical solutions have been proposed for this problem. In this paper, we offer our solution to the problem. More specifically we make the following contributions:

– We formally define the metric *Performance Ratio (PR)* to measure the relative performance of each query and the objective for the whole system (informally, we want to minimize the worst relative performance among all queries).

– By building a new cost model, we identify the heuristics that can be used to approach the objective. More specifically, the heuristics (1) balance the load among all the processing nodes; (2) restrict the number of nodes that the operators of a query can be distributed to; (3) and minimize the total communication cost under conditions (1) and (2).

– The design objective of a platform independent (independent on the underlying stream processing engines) and non-intrusive load management scheme distinguishes our approach from existing ones (e.g. [14]). The proposed techniques are meant to allow the leveraging of existing well developed single-site stream processing engines without much modifications. This is reflected throughout the design of the whole system, especially the load selection strategy.

– To support heuristic (1), we focus on new architectural design that allows us to tap on existing well studied load balancing algorithms instead of proposing new ones. The architectural design includes constructing the load migration unit, load management partner selection, online collection of load statistics, selection of operators to be migrated, operator migration mechanisms.

– To reduce the overhead of employing heuristic (2), unlike existing proposals [7, 11, 14] where load (re)distribution is done at the operator level, we adopt the notion of *query fragments* (a subset of operators) as the finest migration unit. It also helps reduce the overhead of making load balancing decisions.

– To employ heuristic (3), we propose the data flow aware load selection strategy to select the query fragments to be migrated. It effectively maintains data flow locality so that the communication cost is minimized.

– We conducted an extensive simulation study to evaluate the proposed strategy. Results show that the proposed strategy can effectively adapt to the runtime changes of the system to approach our objective.

The rest of this paper is organized as follows. Section 2 formulates the problem and presents our analysis. We present the details of our system design in Section 3. Experiment results are presented in Section 4. Finally Section 5 concludes the paper.

2 Problem Formulation and Analysis

In this section, we present the system model and define the metric to measure the system performance, followed by a formal presentation of the problem statement. Finally, we analyze the problem by building a new cost model and present the proposed heuristics.

2.1 Problem Formulation

Our system consists of a set of geographically distributed data stream sources $S = \{s_1, s_2, \cdots, s_{|S|}\}$ and a set of distributed processing nodes $N = \{n_1, n_2, \cdots, n_{|N|}\}$

interconnected by a local network. As transfer cost from the sources to the processing nodes is much higher than the one among the processing nodes, each source stream is routed to multiple processing nodes through a delegation node. We denote the delegation scheme as Ω. Users impose a set of continuous queries $Q = \{q_1, q_2, \cdots, q_{|Q|}\}$ over the system. The set of operations $O_k = \{o_1, o_2, \cdots, o_{|O_k|}\}$ of query q_k might be distributed to a set of nodes $N_k \subseteq N$ for processing. The operators we consider include filters, window joins and window aggregations. In addition, we denote the set of streams that a query q_k operates on as S_k.

Like previous work on continuous processing of streams [5, 11], we are concerned about the delay of resulting data items, which is also one of the main concerns of end users in terms of system performance. More formally, if the evaluation of query q_k on a source tuple $tuple_l$ from stream s_l generates one or more result tuples, then the delay of $tuple_l$ for q_k is defined as $d_k^l = t_{out} - t_{in}$, where t_{in} is the time that $tuple_l$ arrived at the system and t_{out} is the time that the result tuple is generated. If there are more than one result tuples, then t_{out} is the time that the last one is generated. A similar metric was used in [11]. We focus on this metric because users in a continuous query system typically make decisions based on the results arrived so far. Shorter delay of result tuples would enable a user to make more timely decisions.

At a closer look, d_k^l includes the time used in evaluating the query (denoted as p_k^l), the time waiting for processing as well as the time it is transferred over the network connections. For a specific processing model and a particular query q_k, we regard the evaluation time p_k^l as the inherent complexity of q_k. Since different queries may have different inherent complexities, the value of d_k^l cannot reflect correctly the relative performance of different queries. For example, a query may experience a long delay because its evaluation time is long. We cannot conclude that the relative performance of this query is worse than another one which has a shorter evaluation time. However, in a multi-query and multi-user environment, we wish to tell the relative performance of different queries. Hence we propose a new metric *Performance Ratio (PR)* to incorporate the inherent complexity of a query. Formally, the PR_k^l of the processing of $tuple_l$ for q_k is defined as $PR_k^l = \frac{d_k^l}{p_k^l}$. And the performance ratio of q_k is defined as $PR_k = \max_{s_l \in S_k} PR_k^l$. PR_k reflects the relative performance of q_k. Our objective is to minimize the worst relative performance among all the queries.

The formal problem statement is as follows: *Given a set of queries Q, a set of processing nodes N, a set of data stream sources S and a delegation scheme Ω, according to the change of system state, dynamically distribute the operators of each query to the $|N|$ processing nodes so that the maximum performance ratio $PR_{max} = \max_{1 \le k \le |Q|} PR_k$ is minimized.*

2.2 Problem Analysis.

In this section we develop a cost model to estimate the values of d_k^l and p_k^l. Note that our cost model is meant to be simple for us to figure out the main

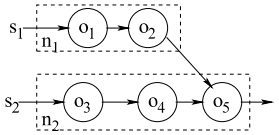


Fig. 1. An example query plan

factors that affect these values and to allow us to analyze the problem complexity. Finding that the problem is NP-hard, we design some heuristics to help solve the problem.

Cost Model. In our cost model we adopt the following simplifications and assumptions:

1. Operators of each query compose a separate processing tree. They are grouped into query fragments and distributed to the processing nodes. Figure 1 shows an example processing tree for a query whose operators are grouped into two query fragments and distributed to two nodes: n_1 and n_2. Tuples arrived at each node are processed in a FIFO manner. Only when an input tuple[1] is fully processed would a new input tuple be processed. The cost of delivering the final results to the users is not considered.

2. For an operator o_j, we assume its per-tuple evaluation time t'_j is independent of its location. And we define its average per-tuple selectivity sel_j as the average number of tuples that would be generated for a given input tuple.

3. Workload ρ_i of a node n_i is defined as the fraction of time that the node is busy.

Given these assumptions, we now look at how to estimate p_k^l and d_k^l. In a particular execution plan of a query, for source tuples from each querying source, there is a path composed by some operators and possibly some network connections. For example, in Figure 1, the path for source tuples from s_1 consists of o_1, o_2, o_5 and the connection between n_1 and n_2, while the path for those from s_2 comprises o_3, o_4 and o_5. Hence, roughly speaking, the p_k^l and d_k^l of a source tuple are respectively equal to the total processing time of the operators in its path and the total time that the tuple stays in its path. In the following paragraphs we will compute them one by one.

For query q_k, assume the path for source tuples from s_l comprises a set O_k^l of operators and some network connections. Furthermore, let O_k^l be distributed to a set N_k^l of nodes and $O_{k,i}^l \subseteq O_k^l$ be the subset of operators of O_k^l assigned to node n_i (where $n_i \in N_k^l$). Let the average per-tuple evaluation time of operator $o_{l_j} \in O_k^l$ be t'_j and its average per-tuple selectivity be sel_j. Without loss of generality, assume o_{l_j} is processed before $o_{l_{j+1}}$. Note that only those source tuples that would be output as result tuple(s) are counted in our metric (hence, each operator's selectivity on these particular tuples is at least 1). Assume $tuple_l$

[1] A tuple here could be a batch of individual tuples in a batch processing mode.

from s_l is such a tuple, then the average processing time of o_{l_j} incurred by $tuple_l$ is $t_j = t'_j \prod_{h=1}^{j-1} \max(sel_h, 1)$. Hence we have

$$p_k^l = \sum_{o_{l_j} \in O_k^l} t_j. \tag{1}$$

In our model every processing node is a queueing system. From queueing theories, in all solvable single task queueing systems, the time that a data item spends in a system can be calculated as $t = g(\rho) * t_s$, where t_s is the processing time of a data item and $g(\rho) \geq 1$ is a monotonically increasing concave function of the system's workload ρ. The exact form of $g(\rho)$ depends on the type of system, e.g. $g(\rho) = \frac{1}{1-\rho}$ in an M/M/1 system.

This inspires us to model the delay of $tuple_l$ as

$$d_k^l = (\sum_{n_i \in N_k^l} (f(\rho_i) \times \sum_{o_{l_j} \in O_{k,i}^l} t_j)) + t_c \times m, \tag{2}$$

where t_c is the communication delay of a tuple and m is the number of times that a tuple is transferred over the network. $f(\rho_i)$ is a monotonically increasing concave function. Note that $f(\rho_i)$ is different from $g(\rho)$ mentioned above and may have a much higher value than $g(\rho)$. That is because there are multiple tasks running on each node. We assume $f(\rho_i)$ is identical for all nodes. Hence the first term of the right-hand side of Equation (2) summarizes the delay in the processing nodes while the second term summarizes the delay caused by the communications.

Based on Equations (2.1), (1) and (2), we have

$$PR_k^l = PPR_k^l + CPR_k^l, \tag{3}$$

where

$$PPR_k^l = \frac{\sum_{n_i \in N_k^l} (f(\rho_i) \times \sum_{o_{l_j} \in O_{k,i}^l} t_j)}{\sum_{o_{l_j} \in O_k^l} t_j}, \tag{4}$$

and

$$CPR_k^l = \frac{t_c \times m}{\sum_{o_{l_j} \in O_k^l} t_j}. \tag{5}$$

We call PPR_k^l the processing performance ratio (PPR) and CPR_k^l the communication performance ratio (CPR). Analogously, $PPR_k = \max_{s_l \in S_k} PPR_k^l$ and $CPR_k = \max_{s_l \in S_k} CPR_k^l$.

Problem Complexity. Given the cost model, let us examine the complexity of the problem. We can observe that the total number of possible allocation schemes is $|N|^{|O|}$ where $O = \bigcup_{1 \leq k \leq |Q|} O_k$. Even worse, we can derive that the

problem is actually NP-hard. To see this let us first ignore the communication cost and only consider minimizing $PPR_{max} = \max_{1 \leq k \leq |Q|} PPR_k$. It is easy to see from Equation (4) that PPR_k^l is a weighted sum of the $f(\rho_i)$ values, where the weight for $f(\rho_i)$ is the fraction of evaluation time p_k^l allocated to node n_i. Assume we can migrate the load between nodes in the finest granularity. Then we have the following observation.

Observation 1 *To minimize PPR_{max}, PPR_k is equal for all queries and ρ_i is equal for all nodes.* □

The intuition behind it is when PPR_k of a query q_k is higher than the others, we can always allocate more resources to q_k (i.e. reducing the workload of some of the processing nodes for q_k by load migration to the other nodes) so that PPR_k is still the largest but is reduced. When the load is balanced then PPR_k equal to $f(\overline{\rho})$ for all queries, where $\overline{\rho}$ is the uniform workload of all nodes. However, we cannot migrate the load in the finest granularity in practice and hence the best plan is to minimize the difference of loads among all the nodes. By restricting our problem to ignore the communication cost, it is equivalent to a MULTIPROCESSOR SCHEDULING problem which is NP-hard. Hence our problem is NP-hard.

Heuristics. In view of the complexity of the problem, we opt to designing heuristics instead of finding an optimal algorithm. From the estimation equation d_k^l, we know that the extra delay is caused by the communication and the workload of the system. Hence, we adopt the following heuristics. (1) Dynamically balance the workload of the processing nodes. This heuristic is inspired by Observation 1. (2) Distribute operators of a query to a restricted number of nodes so that communication overhead of a query is limited. We call the maximum of this number as the distribution limit of that query. Note that always distributing all the operators of every query to a single node is impractical, because it would incur excessive data flow over the network.(3) Minimize the communication cost under conditions (1) and (2). In short, we have to design a dynamic load balancing scheme where the operations of each query should not be distributed to too many nodes and the total communication traffic is minimized.

Besides employing the heuristics stated above, the scheme should also satisfy the following objectives in the perspective of system design:

1. *It is fast and scalable.* Because dynamic re-balancing could happen frequently at runtime, the overhead of making re-balancing decisions should be kept low. Furthermore, a distributed scheme is preferred to enhance scalability and avoid bottleneck.

2. *It does not rely on any specific processing model.* There are different single-node processing models that are currently under development such as TelegraphCQ [6], Aurora [4] and STREAM [1]. Our system is not restricted to any processing model because it separates the stream processing engine in each node from the distributed processing details. Queries are compiled into logical query plans which consist of *logical operators*. The logical operators are distributed

to the processing nodes by our placement scheme. Then the logical operators would be mapped into *physical operators* by the stream processing engine for processing. Different engines under different processing models could map a logical operator into a different physical operator.

2.3 Related Work

Distributed continuous query systems attracted much research attention in the recent years. The necessity of dynamic load management in distributed stream processing has been identified in several published references e.g. [7,11]. However, the authors did not propose any complete and practical strategies. In Flux [9], a dynamic load balancing strategy for the *horizontal* (or intra-operator) parallelism was employed. While the mechanism was developed in the context of continuous queries, a centralized synchronous controller is used to collect workload information and to make load balancing decisions. Our work, on the other hand, takes a complimentary approach by focusing on allocation of operators in the context of *vertical* (or pipelined) parallelism. Furthermore, our approach is decentralized and asynchronous. More recently, Borealis system [14] also adopted a centralized load distribution technique in the context of vertical parallel processing. An innovative load balancing approach is presented which considers the time correlations between the operators. On the contrary, our work does not focus on proposing new load balancing algorithms.

Furthermore, the problems of the above two pieces of work bear a few important differences from ours. First, in their approach, stream delegation is not employed. Hence it is possible that some sources have to communicate with multiple processing nodes or some processing nodes may have to collect streams from a lot of sources. Second, the network resources are assumed to be abundant and hence communication cost is ignored in their techniques. How to take the communication cost in account is still unclear. Third, without considering the delegation scheme and the communication cost, the problem is only a partitioning problem which partitions the operators into balanced partitions. The processing nodes are identical in terms of partition allocation. However, our problem is essentially an assignment problem as assigning query operators to different nodes would have different communication cost for a given delegation scheme.

On the other hand, [3, 8] focused on minimizing communication cost but ignores load balancing. [10] studied the static operator placement in a hierarchical stream acquisition architecture, which is much different from our system architecture. Load balancing is also ignored in this piece of work. [15] proposed an adaptive scheme disseminate stream data to the distributed stream processors without considering operator placement.

3 System Design

In our dynamic operator placement scheme, we adopt a local load balancing strategy. Each node would select its load management partners and dynamically

balances the load between its partners. To implement this, there are several issues to be addressed: (1) initial placement of operators; (2) load management partner selection; (3) workload information collection; (4) load balance decision-making; (5) selection of operators for migration. We address these issues in the following subsections.

3.1 Initial Placement of Operators

In our initial placement scheme, we only consider minimizing the communication cost and leave the load balancing task to our dynamic scheme. The scheme generates one query fragment for each participating stream and then distributes the query fragments to the delegation nodes of their corresponding streams. More specifically, the scheme comprises the following steps:

1. When a query is submitted to the system, it is compiled and optimized into a logical query plan without considering the distribution of the data streams. The logical query plan, which is represented as a traditional query plan tree, determines the required logical operators such as filters, joins, aggregation operators and their processing orders. Existing optimization techniques [2, 12] can be applied at this step. Figure 2(a) is an example of the resulting query tree of this step.

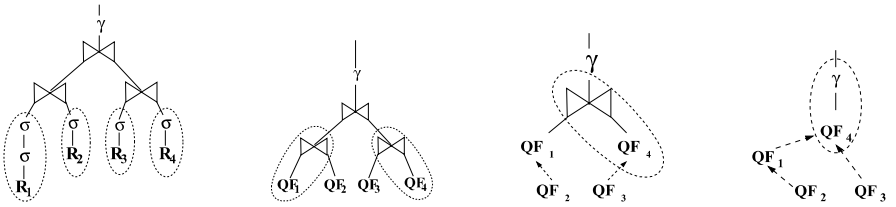
2. For each stream involved in the query, generate one query fragment which is initially set to empty. Add each leaf node (i.e. the stream access operators) to its corresponding query fragment QF_i and then replace it with QF_i.

3. For each query fragment, if the parent operator is a unary operator, the operator would be added to the query fragment and removed from the query tree. The step is repeated until all the operators are removed or the parent operator for every query fragment is a binary operator. Figure 2(b) is an example of the resulting query tree of this step. The intuition is to place each stream's filters at its delegation node to reduce the amount of data to be transferred.

4. Now we have a query tree in which all the next-to-leaf nodes are binary operators. Add each next-to-leaf binary operator to one of its two child query fragments, say QF_i, whose estimated resulting stream rate is higher than the other one. Then remove the other query fragment from the tree and push QF_i up a level to replace that binary operator. A binary operator is added to the query fragment of higher (estimated) resulting stream rate to reduce the volume of data that needs to be transmitted through the network if the two fragments of the two involved streams are to be evaluated at two different nodes. This process continues until all operators are removed or the parents of one or more of the remaining query fragments are unary operators. For the latter case, the algorithm goes back to step (3). Figures 2(c) and (d) illustrate the procedure of this step.

5. Distribute the query fragments to the delegation nodes of their corresponding streams.

Based on the operator ordering, there is a downstream and upstream relationship between some of the query fragments. For example, in Figure 2, results of QF_2 should be further processed by the binary operator of QF_1 and hence we



(a) Phase 1 (b) Phase 2 (c) Phase 3 (d) Phase 4

Fig. 2. Query Fragments Generation

Algorithm 1: PARTNERSELECT

```
sort neighbors in descending order of neighboring factor;
for (i ← 0; |g1| < max1 AND i <|neighbors|+ MaximumTry; i++) do
    if i < |neighbors| then n ← neighbors[i];
    else n ← a random node ∉ neighbors ∪ g1;
    if n ∈ g2 then
        move it from g2 to g1;
    else if n ∉ g1 then
        send a request to n;
        if the request is accepted then
            add n to g1;
        endif
    endif
endfor
```

call QF_2 the upstream query fragment of QF_1. Similarly, QF_1 is the upstream query fragment of QF_4. Symmetrically, we call QF_1 (or QF_4) the downstream query fragment of QF_2 (or QF_1). We call a query fragment's downstream or upstream query fragments its neighbors. For instance, QF_2 and QF_4 are neighbors of QF_1. Furthermore, if a query fragment QF_i's corresponding data stream is delegated to a node n_j then QF_i is called a *native query fragment* of n_j and n_j is a *native node* of QF_i. Otherwise, QF_i is called a *foreign query fragment* of n_j and n_j is a *foreign node* of QF_i.

Furthermore, the native nodes of two neighboring query fragments are called *neighbors* to each other. And the number of neighboring query fragments between two nodes is called the *neighboring factor*.

3.2 Partner Selection Strategy

As stated above, our dynamic load balancing scheme is a local strategy. Each node n_i has a number of load management partners (abbreviated as partners). The partner relationship is symmetric, i.e. if n_i is a partner of n_j, then n_j is also a partner of n_i. In this section, we discuss the partner selection strategy for each node.

In our scheme, each node sends out requests to some other nodes to initiate the partner relationships and receives such requests from its peers. We separate the partners of each node into two groups : (1) g_1, the relationship is created by the (explicit) request of this node; (2) g_2, the rest. There is a maximum bound for each group of partners denoted as max_1 and max_2 respectively. Each node would use Algorithm 1 to send out requests. Neighbors with higher neighboring factors with the current node have higher priority to be selected. That is to enhance the opportunity of reducing communication cost during load redistribution, which is can easily be seen in Section 3.5. Algorithm 1 is implemented in asynchronous mode in our system. It does not wait for a remote response but instead returns once all requests have been sent out. After a node receives a response message, the algorithm is called to resume the processing. Furthermore, a node n_i which receives a request will check whether the sender n_j is also being requested by n_i or is already in g_1. If so, n_i accepts the request and adds n_j into g_1 if necessary. Otherwise it adds n_j into g_2 if $|g_2| < max_2$ or sends back a reject message otherwise. A node will update its partners periodically.

3.3 Information Collection Strategy

The information collection strategy determines when and how workload information of nodes in the system is collected and also what information is to be collected.

We adopt a window based and asynchronous workload collection approach. Time is divided into windows which have static lengths τ. Each node accumulates the total processing time t of all its physical operators within each window and the workload with respect to a window is computed by dividing t by τ. Each node asynchronously collects its workload within each window and updates its workload once the current time window elapsed. It broadcasts the workload information to all its partners if its workload increases to κ or decreases to $1/\kappa$ times of the last broadcast value.

The above strategy performs well only if the input rate and the processing time are constants. But in practice they are random variables. The resulting workload may fluctuate over time, which renders the system unstable. As stated before, we only focus on adaptation to long term system changes which would bring long term benefits and alleviate the short term adaptation overhead. To prevent the system from reacting to short term fluctuations, we use a low pass filter to remove the high frequency noises (caused by the short term changes of stream rates, tuple processing time, etc.) in workload collection. In particular, workload is computed as $\rho_{i+1} = \alpha \times \rho_i + (1-\alpha) \times \rho_c$, where ρ_{i+1} and ρ_i are the workload information used for load balancing after $i+1$ and i time windows, and ρ_c is the collected workload within the $(i+1)$th time window. α is a parameter to determine the responsiveness of the estimated value to the workload changes. The purpose of using this formula in previous work is to give more weight to recent collected statistics. Here we analytically show that it can also smooth out short term fluctuations. A validation experiment can be found in [16]

We now consider how α would be set in a system. Without loss of generality, we assume the workload is increasing. Given the initial workload ρ_0 and that we want to filter out transient workload fluctuation where the workload is changed to $l\rho_0$ ($l > 1$) within $m_1\tau$ time and last for $m_2\tau$ time, we should choose α such that the estimated workload after $(m_1 + m_2)\tau$ time $\rho_{m_1+m_2}$ should satisfy $\rho_{m_1+m_2} \leq \kappa\rho_0$. In practice, the values of m_1, m_2 and l reflect the typical range and time span of short term fluctuations. They can be adaptively tuned by collecting the characteristics of the system. In our calculation, we assume the workload increases $(l-1)\rho_0/m_1$ within each τ time during the $m_1\tau$ period. After $(m_1 + m_2)\tau$ time, the estimated workload $\rho_{m_1+m_2}$ can be calculated as [16]:

$$\alpha^{m_2}\rho_0 + \frac{\alpha^{m_2}(l-1)\rho_0}{m_1}(m_1 + 1 + \frac{\alpha^{m_1+1} - 1}{1-\alpha}) + (1 - \alpha^{m_2})l\rho_0.$$

Substitute the above equation into the inequality $\rho_{m_1+m_2} \leq \kappa\rho_0$, we have

$$\alpha^{m_2} + \frac{\alpha^{m_2}(l-1)}{m_1}(m_1 + 1 + \frac{\alpha^{m_1+1} - 1}{1-\alpha}) + (1 - \alpha^{m_2})l \leq \kappa.$$

Hence we can calculate the lower bound of α by solving the above inequation given the values of m_1, m_2 and l. For example, given $m_1 = m_2 = 1$, $l = 2$ and $\kappa = 1.2$, we can get $\alpha \geq 0.9$. The case for short term workload decrease can be analyzed similarly. On the other hand, α also cannot be too close to 1, otherwise the current workload will not be reflected. A similar upper bound analysis can be performed [16].

3.4 Load Balance Decision Strategy

The load balance decision strategy determines whether it is beneficial to initiate a load balance attempt and how much workload should be transmitted between the nodes. Our strategy is adapted from the local diffusive load balancing strategy introduced in [13]. It is a receiver-initiated strategy, which is found to be more efficient in [13]. It works in rounds. The length of each round is denoted as Δ. Each node maintains its own value of Δ. At the start of each round, Algorithm 2 is run to generate one workload request if necessary. In this algorithm, the load request is generated by the potential load receiver (i.e., the node with smaller load initiates load balancing). Since we focus on continuous queries, load migration can bring long term benefits. As such our decision strategy does not consider the short term migration overhead. Once a node receives a workload request, it satisfies the request as much as possible, provided the workload to send out within each Δ time window is no more than half of its total workload at the beginning of the current window.

It is possible that the nodes in the system are separated into several non-overlapping groups and the workloads are not balanced between groups. Hence once a node in our system detects that itself and all of its partners are overloaded, it will randomly probe the other nodes until it finds an underloaded node to add it as a partner or the probe limit is reached.

Algorithm 2: GENERATEREQUEST

compute the average workload $\overline{\rho}$ within itself and its partners;
if *the local workload* $\kappa\rho_l < \overline{\rho}$ **then**
  find the partner n_i whose workload ρ_i is the largest;
  compute the load request $\rho_r = (\rho_i - \rho_l)/2$;
  request ρ_r amount of workload from n_i;
endif

3.5 Load Selection Strategy

As stated above, once a potential load sender receives a load request, it will select the *victim* query operators to satisfy the request as far as possible. When multiple such requests are received, the sender processes them in descending order of the workload amount requested. The sender will estimate its resulting workload after each migration, and if it detects that half of the workload has been exported within the current Δ interval, it will stop processing any request until the start of the next round. In this subsection, we explore how to select the victim operators for migration and discuss how to migrate them in the next subsection.

Migration Unit. The first question to be answered is what is the smallest task unit used for load migration. We consider the following choices:

1. Using *the whole query* as the migration unit is easy to implement. However, a good evaluation plan often distributes the operations across multiple nodes in order to minimize the communication overheads. So migrating in the unit of query is inappropriate.

2. *Operator* as another candidate is a fine-grained unit. Migrating at this level may result in better balance state. However, it is hard to implement our second heuristic which imposes a distribution limit on the query operators (see Section 2.2). When we are trying to move an operator, we have to know the location of the other operators belonging to the same query. Otherwise, we do not know if the distribution limit is violated. This results in high update overhead and is not compatible to our local strategy as a node cannot make decisions based on local information.

3. *Query fragments.* Based on the above analysis, a good candidate for migration unit should render the maintenance of good query plans easy and allow the separation of load balancing strategy from the underlying stream processing engine and hence introduce less complexity to the existing processing techniques. Furthermore, this unit should not be too coarse to restrict the adaptive ability of the load management module. For the above purposes, we would like to find a subset of operators that is of appropriate size and would be processed in the same site in most cases for a good query plan. Furthermore, we consider only candidates in the logical level. We adopt the notion of query fragment - a subset of logical operators of a query. We set the number of query fragments of a query

as its distribution limit. This exempts the task of keeping track of the distribution of all the operators of a query while we are implementing heuristic (2). The distribution limit would always be met no matter where we allocate the query fragments.

While a query can be fragmented in a lot of ways, we simply use the query fragments generated in our initial placement scheme as the migration units. Operators in each of such query fragments would be allocated to the same processing node in a good query plan generated by applying traditional optimization heuristics. Furthermore, by doing so, the distribution limit of a query is set to the number of streams involved by the query. Here, we assume that queries involving more streams are more complicated and hence can afford a higher distribution limit.

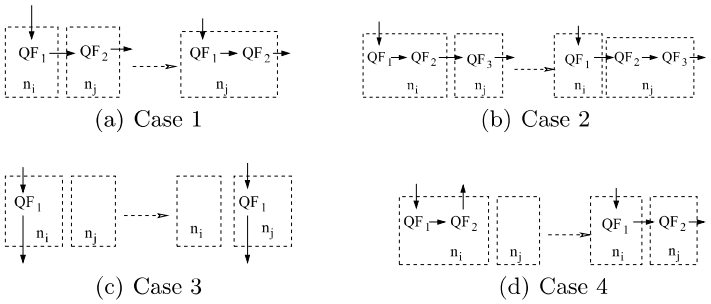
Data Flow Aware Load Selection. The choice of query fragments to be migrated is critical in maintaining data flow locality. A poor choice may cause streams to be scattered across too many nodes and result in network congestion. In this subsection, we propose a lightweight query fragment selection strategy which makes decisions only based on local information.

In our strategy, for each request, the sender chooses the query fragments in the following order until the request is satisfied or half of the workload of this node has been exported within the current Δ interval.

1. *Query fragments that are foreign to the sender but native to the receiver.* This kind of query fragments is considered to be of highest priority to migrate because migrating them has the potential to reduce the data flow.
2. *Other query fragments that are foreign to the sender.*
3. *Query fragments that are native to the sender.* This kind of query fragments is considered of lowest priority for migration because migrating them tends to scatter the streams delegated to this node.

The above heuristics are reasonable in maintaining data flow locality. However, its categorization is too coarse. The migrations of the query fragments within each category may still have different effects on the data flow locality and the delay of the queries. For example, migrating a query fragment QF_i to a node that is evaluating a neighbor of QF_i may bring less increase of data flow than migrating it to other nodes. This is because it avoids the transfer of the data flow between QF_i and its neighbor. Hence, within each of the above categories, we further classify the query fragments into one of the following categories and we list them in the order of descending migration priorities.

1. *Query fragments that have neighbors being evaluated at the receiver but none at the sender.* The migration of this class of query fragments eliminates the transmission of the data flow between the sender and the receiver caused by the migrated query fragment. Figure 3(a) shows a possible situation in this case. The situations before and after migration are plotted on the left and the right respectively. Solid arrows in the figure indicate the data flows between the query fragments. For brevity, the other query fragments being evaluated in the two nodes are not shown. In this example QF_1 is a neighbor of QF_2. n_i is the sender



(a) Case 1 (b) Case 2

(c) Case 3 (d) Case 4

Fig. 3. Query fragments migration cases

while n_j is the receiver. After migration, the data flow introduced by QF_1 and QF_2 between n_i and n_j is eliminated.

2. *Query fragments that have neighbors at both nodes.* This class of query fragments has lower migration priority than the above-mentioned one because the migration eliminates one data flow but also creates another one between the sender and the receiver. For example, in Figure 3(b), the transmission of the data flow introduced by QF_2 and QF_3 is eliminated while the one incurred by QF_1 and QF_2 is created by the migration.

3. *Query fragments have neighbors at neither node.* Figure 3(c) is an example situation.

4. *Query fragments that have neighbors at the sender but none at the receiver.* This class has lower priority than the third one because the migration may introduce extra data flow between the sender and the receiver. An example of this case can be found in Figure 3(d). The migration in this example creates the data flow between n_i and n_j caused by QF_1 and QF_2.

If there is more than one query fragment in the above subcategories, we will compute the migration priority for each of them and will migrate those with higher priorities first. The migration priority of a query fragment is computed as $\frac{\rho}{\max(size,1)}$, where ρ is the workload it incurs, and $size$ is its state size in bytes. We call this value the *load density* of the query fragment as it means the amount of workload will be migrated for each byte of state transmission. Furthermore, ρ is estimated by summing up the estimated workload incurred by each of its logical operator, which is estimated as $1/n$ of the workload caused by its corresponding physical operator. n is the number of logical operators sharing that physical operator.

4 A Performance Study

In our experiments, the stream processing engine in each node is an emulation of the TelegraphCQ system. We use a simulator to simulate the communication among the processing nodes. The simulator is implemented in JAVA using the JavaSim discrete event simulation package. We use 32 simulation nodes and an

additional sink node as our basic configuration. Each processing node is delegated 3 streams. Tuples from every stream are of 100 bytes and consist of 10 attributes. The bandwidth of the network connecting the nodes is modeled as 100Mbps.

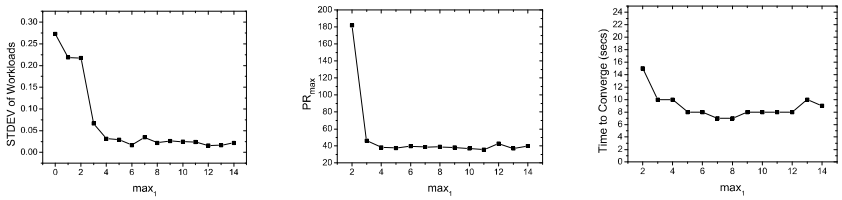
We use 500 randomly generated queries and a total of 5750 logical operators, to measure our system performance. The sliding window size for window joins is randomly selected from 5000 to 20000. The selectivities of the operators are from 0.5 to 0.8. We set the average data inter-arrival time to be $4ms$ and the mean processing time for each filter and join operation to be $20\mu s$ and $80\mu s$ respectively. Besides, we use the following algorithm parameters: the workload collection window $\tau = 100ms$, the length of load management round $\Delta = 1s$, and the threshold to broadcast workload $\kappa = 1.2$. The real values of p_k^l and d_k^l were collected online and the PR_k values were computed by the sink node when it received a result tuple.

4.1 Partner Selections

We have two parameters for our partner selection strategy: max_1 and max_2. In this experiment, we set $max_2 = \lceil \frac{1}{2} max_1 \rceil$ and vary the value of max_1. To generate an imbalanced workload, the streams that a query operates on are chosen according to a Zipfian distribution ($\theta = 0.95$). We use the standard deviation (STDEV) of the ρ_i for all processing nodes to measure the load imbalance, i.e. $\sqrt{\frac{\sum_i (\rho_i - \overline{\rho})^2}{|N|-1}}$. Figure 4(a) shows the final load distribution for different values of max_1. $max_1 = 0$ means that dynamic load balancing is disabled. We can see when $max_1 >= 4$ the load is well balanced. No significant improvement can be made by using a larger max_1 value. Figure 4(b) illustrates the PR_{max} after the system is stable. It is computed by averaging on the values within 10 seconds. It is clear that the PR_{max} values are also similar when $max_1 >= 4$. Figure 4(c) shows the time it takes to converge to the final load distribution. There is not much difference between small and large number of partners. The above comparisons show that our system works well with a small max_1 value. As a larger number of partners would increase the runtime cost (such as transferring workload update messages, making load balancing decisions), we could keep the number to a small value and hence keep the cost low. In the subsequent experiments, we set $max_1 = 5$ and $max_2 = \lceil \frac{1}{2} max_1 \rceil$.

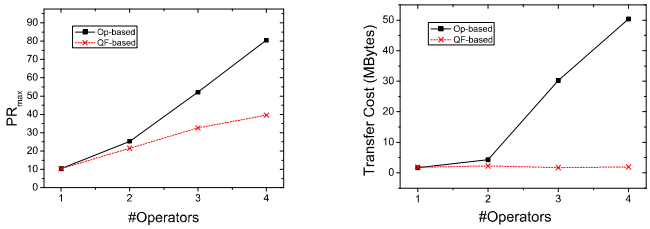
4.2 Load Selection Heuristics

The first experiment examines the necessity of imposing a distribution limit. This is done by comparing the QF-based (query fragment based) load balancing strategy with the OP-based (operator based) strategy proposed by reference [14]. The latter approach does not impose any distribution limit. We varied the number of operators per query fragment in our experiment. We ran the experiment under each case for 60 seconds simulation time and report the average values. We can see from Figure 5 that with more number of query operators, the PR_{max} value of the QF-based approach performs much better. Figure 5(b) may explain



(a) Load imbalance (b) PR_{max} (c) Time to Converge

Fig. 4. Effect of various partner selection parameter

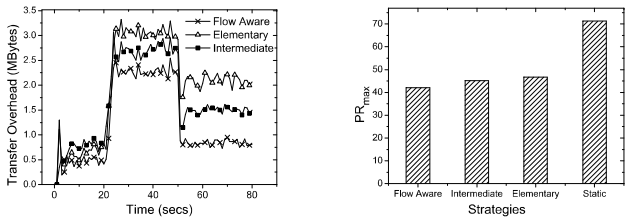


(a) PR_{max} (b) Transfer cost

Fig. 5. QF-based vs. OP-based

this phenomenon. The data transfer volume of the OP-based scheme increases quickly with more operators, because operators of a single query are migrated to too many site. Therefore the data streams are scattered over the network and leads to network congestion. On the other hand, the QF-based strategy still maintains small transfer overhead and hence it still performs well in data delay. Note that, by employing a distribution limit, an OP-based strategy can achieve better performance. However, as analyzed before, the cost to maintain such a limit would be higher than a QF strategy and such a scheme does not fit into a local load management strategy.

The second experiment examines the effectiveness of our flow-aware load selection strategy in maintaining good data flow locality. We impose an initially balanced load distribution over the processing nodes and use a uniform distribution to choose the querying streams S_k for every query q_k. At time $t = 20s$ we randomly select 4 nodes and then increase the input rates of the streams delegated to those nodes by 3 times. At $t = 50s$, the increased input rates drop back to their initial values. To show the effect of load selection strategy, we design another two approaches for comparison: (1) Elementary: the query fragments are selected in descending order of their load density. (2) Intermediate: the same as Elementary except foreign query fragments are given higher migration priori-



(a) Transfer Overhead (b) PR_{max}

Fig. 6. On load selection strategies

ties than the native query fragments. In previous work, such as [9,14], data flow relationship is not considered. Hence their effects on the communication cost can be well represented by the Elementary algorithm. We compare the transfer overhead introduced by the three strategies against the static query fragment allocation strategy, i.e. the initial placement scheme. The static strategy allocates the query fragments to their native nodes, hence its data flow transfer cost is minimum though it may incur very high data delay due to the unbalanced load allocation. We subtract the amount of transfer cost of the static strategy from those of the other three and then compare the extra transfer overheads of the three dynamic strategies over the static one.

From figure 6(a), we can see that the data flow aware strategy outperforms the other two at all stages of the experiment. Both Intermediate and Elementary , unlike the data flow aware strategy, fail to identify the neighborhood relationship of the query fragments. Intermediate is better than Elementary because it can differentiate between foreign query fragments and native query fragments and to some degree can help maintain data flow locality. At $t = 50s$ when the perturbed stream rates dropped back to the original value, all three strategies' transfer overheads are reduced. However, both Intermediate and Elementary cannot restore back to the state prior to the change. This is because both strategies are unable to identify their native nodes when migrating foreign query fragments. That means they would become worse and worse with the evolution of the system state while the data flow aware strategy is able to maintain a more stable state over time. Figure 6(b) shows the PR_{max} for all the four strategies. The values are calculated by averaging over the whole simulation time. The static strategy performed the worst simply because of the absence of load balancing strategy. Furthermore, the three dynamic strategies performed similarly. This is attributed to our heuristic to maintain a distribution limit for every query. Since processing load are similar for the three dynamic strategies due to the balanced load distribution, PR_{max} was similar for the three strategies. However, in the case when network traffic is so high that it approaches the bandwidth limit, the data flow aware strategy will do much better to avoid network congestion situation.

5 Conclusion

Distributed processing of continuous queries over data streams suffers from run time changes of system resource availability and data characteristics. Dynamic operator placement techniques are indispensable for a distributed stream processing system. In this paper, we formalized the problem and analyzed it by building a cost model. As shown in our experiments, load imbalance can cause severe performance degradation and our proposed techniques can alleviate such degradation by dynamic load balancing. Our data flow aware load selection strategy can help restrict the scattering of data flows and lead to lower communication cost.

References

1. A. Arasu, et al. Stream: The stanford stream data manager. *IEEE Data Eng. Bull*, 26(1):19–26, 2003.
2. A. Ayad and J. F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *SIGMOD*, pages 419–430, 2004.
3. Y. Ahmad and U. Çetintemel. Networked query processing for distributed stream-based applications. In *VLDB*, pages 456–467, 2004.
4. D. Carney, et al. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, 2002.
5. D. Carney, et al. Operator scheduling in a data stream manager. In *VLDB*, pages 838–849, 2003.
6. S. Chandrasekaran, et al. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.
7. M. Cherniack, et al. Scalable distributed stream processing. In *CIDR*, 2003.
8. P. Pietzuch et al. Network-aware operator placement for stream-processing systems. In *ICDE*, pages 49, 2006.
9. M. A. Shah, et al. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, pages 25–36, 2003.
10. U. Srivastava, et al. Operator Placement for In-Network Stream Query Processing In *PODS*, pages 250–258, 2005.
11. F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, pages 333–344, 2003.
12. S. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD*, pages 37–48, 2002.
13. M. Willebeek-LeMair and A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Trans. Parallel Distrib. Syst*, 4(9):979–993, 1993.
14. Y. Xing, et al. Dynamic load distribution in the Borealis stream processor. In *ICDE*, pages 791-802, 2005.
15. Y. Zhou, et al. Adaptive reorganization of coherency-preserving dissemination tree for streaming data. In *ICDE*, pages 55, 2006.
16. Y. Zhou, et al. Dynamic load management for distributed continuous query systems. Unpublished manuscript, 2005. `http://www.comp.nus.edu.sg/~zhouyong/papers/op.html`.