Dynamic Migration of Processing Elements for Optimized Query Execution in Event-based Systems

Waldemar Hummer, Philipp Leitner, Benjamin Satzger, and Schahram Dustdar

Distributed Systems Group, Vienna University of Technology, Austria
{lastname}@infosys.tuwien.ac.at

Abstract. This paper proposes a method for optimized placement of query processing elements in a distributed stream processing platform consisting of several computing nodes. We focus on the case that multiple users run different continuous Complex Event Processing (CEP) queries over various event streams. In times of increasing event frequency it may be required to migrate parts of the query processing elements to a new node. Our approach achieves a tradeoff between three dimensions: balancing the load among nodes, avoiding duplicate buffering of events, and minimizing the data transfer between nodes. Thereby, we also take one-time costs for migration of event buffers into account. We provide a detailed problem description, present a solution based on metaheuristic optimization, and evaluate different aspects of the problem in a Cloud Computing environment.

Keywords: event-based systems, continuous queries, migrating query processing elements, placement of event subscriptions, WS-Aggregation

1 Introduction

In recent years, academia and industry have increasingly focused on event-based systems (EBS) and Complex Event Processing (CEP) [11] for Internet-scale data processing and publish-subscribe content delivery. The massive and continuous information flow of today requires techniques to efficiently handle large amounts of data, e.g., in areas such as financial computing, online analytical processing (OLAP), wireless and pervasive computing, or sensor networks [22]. In most of these application areas, filtering and combining related information from different event sources is crucial for potentially generating added value on top of the underlying (raw) data. Platforms that are specialized in continuously querying data from event streams face difficult challenges, particularly with respect to performance and robustness. Evidently, continuous queries that consider a sliding window of past events (e.g., moving average of historical stock prices in a financial computing application) require some sort of buffering to keep the relevant events in memory. State-of-the-art query engines are able to optimize this buffer size and to drop events from the buffer which are no more needed (e.g., [17]). However, a topic that is less covered in literature is how to optimize resource usage in a system with multiple continuous queries executing concurrently.

In our previous work we have presented *WS-Aggregation* [13, 14], a distributed platform for aggregation of event-based Web services and Web data. The

platform allows multiple users to perform continuous queries over event emitting data sources. WS-Aggregation employs a collaborative computing model in which incoming user requests are split into parts, which are then assigned to one or more aggregator nodes. For instance, when a query involves input data from two or more data sources, each of the inputs may be handled by a different aggregator. Throughout various experiments we observed that query distribution and placement of processing elements has a considerable effect on the performance of the framework. For the remainder of the paper, an *event subscription* determines the node that receives and processes the events of an event stream.

In this paper we study how the placement of processing elements affects the performance of single queries and the overall system. To that end, we take several aspects into account. Firstly, computing nodes have resource limits, and in times of peak loads we need to be able to adapt and reorganize the system. Secondly, complex queries over event streams require buffering of a certain amount of past events, and the required memory should be kept at a minimum. Finally, if the tasks assigned to collaborating nodes contain inter-dependencies, possibly a lot of network communication overhead takes place between the aggregator nodes. We propose an approach which considers all of the above mentioned points and seeks to optimize the system configuration. This work is highly important for the runtime performance of event-based systems that deal with load balancing and dynamic migration of event subscriptions, as exemplified using WS-Aggregation.

In the remainder of this paper, we first discuss related work in Section 2. In Section 3, we present the model for event-based continuous queries in WS-Aggregation. Section 4 discusses different strategies for optimal placement of processing elements in distributed event processing platforms and formulates the tradeoff between conflicting goals as a combinatorial optimization problem. Some implementation details are discussed in Section 6, and the overall approach is evaluated in Section 7. Section 8 concludes the paper with a future work outlook.

2 Related Work

Due to the large number of its application areas, event processing has attracted the interest of both industry and research [19, 27]. Important topics in CEP include pattern matching over event streams [1], aggregation of events [20] or event specification [10]. In this paper, we focus on optimizing the distributed execution of continuous queries over event streams. Hence, we concentrate on some related work in this area in the remainder of this section.

Optimized placement of query processing elements and operators has previously been studied in the area of distributed stream processing systems. Pietzuch et al. [23] present an approach for network-aware operator placement on geographically dispersed machines. Bonfils and Bonnet [7] discuss exploration and adaptation techniques for optimized placement of operator nodes in sensor networks. Our work is also related to query plan creation and multi query optimization, which are core fields in database research. In traditional centralized databases, permutations of join-orders in the query tree are considered in order

to compute an optimal execution plan for a single query [15]. Roy et al. [24] present an extension to the *AND-OR DAG* (Directed Acyclic Graph) representation, which models alternative execution plans for multi-query scenarios. Based on the AND-OR DAG, a thorough analysis of different algorithms for multi-query optimizing has been carried out. Zhu et al. [33] study exchangeable query plans and investigate ways to migrate between (sub-)plans.

Seshadri et al. [25, 26] have identified the problem that evaluating continuous queries at a single central node is often infeasible. Our approach builds on their solution which involves a cost-benefit utility model that expresses the total costs as a combination of communication and processing costs. Although the approaches target a similar goal, we see some key differences between their and our work. Firstly, their approach builds on hierarchical network partitions/clusters, whereas WS-Aggregation is loosely coupled and collaborations are initiated in an ad-hoc fashion. Secondly, their work does not tackle runtime migration of query plans and deployments, which is a core focus in this paper. In fact, WS-Aggregation implements the Monitor-Analyze-Plan-Execute (MAPE) loop known from Autonomic Computing [16]. In that sense, the purpose of our optimization algorithm is not to determine an optimal query deployment up front, but to apply reconfigurations as the system involves. Chen et al. [9] describe a way to offer continuous stream analytics as a cloud service using multiple engines for providing scalability. Each engine is responsible for parts of the input stream. The partitioning is based on the contents of the data, e.g., each engine could be responsible for data generated in a certain geographical region.

Several previous publications have discussed issues and solutions related to active queries for internet-scale content delivery. For instance, Li et al. [18] presented the OpenCQ framework for continuous querying of data sources. In OpenCQ a continuous query is a query enriched with a trigger condition and a stop condition. Similarly, the NiagaraCQ system [8] implements internet-scale continuous event processing. Wu et al. [32] present another approach to dealing with high loads in event streams, tailored to the domain of real-time processing of RFID data. Numerous contributions in the field of query processing over data streams have been produced as part of the Stanford Stream Data Manager (STREAM) project [21]. The most important ones range from a specialized query language, to resource allocation in limited environments, to scheduling algorithms for reducing inter-operator queuing. Their work largely focuses on how to approximate query answers when high system load prohibits exact query execution. Query approximation and load shedding under insufficient available resources is also discussed in [3]. Our approach does not support approximation, but exploits the advantages of Cloud Computing to allocate new resources for dynamic migration of query processing elements.

Furthermore, database research has uncovered that special types of queries deserve special treatment and can be further optimized, such as k-nearest neighbor queries [6] or queries over streams that adhere to certain patterns or constraints [4]. WS-Aggregation also considers a special form of 3-way distributed query optimization, which we have presented in earlier work [13].

3 Event-Based Continuous Queries in WS-Aggregation

In the following we establish the model for distributed processing of event-based continuous queries that is applied in WS-Aggregation. The model serves as the basis for the concepts discussed in the remainder of the paper. WS-Aggregation is a distributed platform for large-scale aggregation of heterogeneous internet-based data sources, which supports push-style updates using the notion of continuous event queries on data sources. More information can be found in [13].

Symbol	**Description**
$A = \{a_1, a_2, \ldots, a_n\}$	Set of deployed aggregator nodes.
$Q = \{q_1, q_2, \ldots, q_m\}$	Queries that are handled by the platform at some point in time.
$I = \{i_1, i_2, \ldots, i_k\}$	Set of all inputs over all queries.
$inputs : Q \rightarrow \mathcal{P}(I)$	Function that returns all inputs of a query.
$deps : Q \rightarrow \mathcal{P}(I \times I)$	Function that returns all data dependencies of a query.
$S = \{s_1, s_2, \ldots, s_l\}$	Data sources that emit events over which queries are executed.
$source : I \rightarrow S$	Function to determine the data source targeted by an input.
$query : I \rightarrow Q$	Function to determine the query an input belongs to.
$buf : A \rightarrow \mathcal{P}(S)$	Function to determine which data sources an aggregator buffers.

Table 1. Description of Symbols and Variables in Event-Based Query Model

Table 1 summarizes the symbols and variables that are used in the formalization. In our model, a number of aggregator nodes (A) are collectively responsible to execute multiple continuous user queries (Q). Each query processes one or more inputs (I) from external data sources (S). The function *inputs* maps queries to inputs ($\mathcal{P}(I)$ denotes the power set of I), and the function *source* returns the data source targeted by an input. The actual processing logic of the query is application specific and not directly relevant for our purpose. However, we consider that a query q may contain data dependencies among any two of its inputs $i_x, i_y \in inputs(q), i_x \neq i_y$. A dependency $(i_x, i_y) \in deps(q)$ means that i_y can only be processed after certain data from i_x have been received, because the data are required either 1) by the request to initiate the event stream from the data source underlying i_y, or 2) by a *preprocessing* query that prepares (e.g., groups, filters, aggregates) the incoming events for i_y. Such dependencies are often seen in continuous queries over multiple data streams [5], where subscriptions are dynamically created (or destroyed) when a specific pattern or result is produced by the currently active streams. An example could be a sensor emitting temperature data in a smart home environment, which only gets activated as soon as another sensor emits an event that a person has entered the room.

Although we use the terms service and data source interchangeably, strictly speaking the notion of data source is narrower, because every entry in S is identified by a pair $(epr, filter)$, where *epr* is the *Endpoint Reference* [28] (location)

of the service and the $filter$ expression determines which types of events should be returned. That is, different data sources may be accessed under the same service endpoint. The $filter$ may be empty, in which case events of all types are returned.

The reason for abstracting inputs from data sources is that different queries may require different data from one and the same source. As an example, assume a data source which every second emits an event with the market price of two stocks, and two queries which compute the *Pearson* correlation as well as the *Spearman* correlation of the historical prices. This means that each of the inputs needs to be processed (computed) separately, but the same underlying event buffer can be used for both inputs. We use the function buf to determine the data sources from which an aggregator "currently" (at some point in time) receives and buffers events.

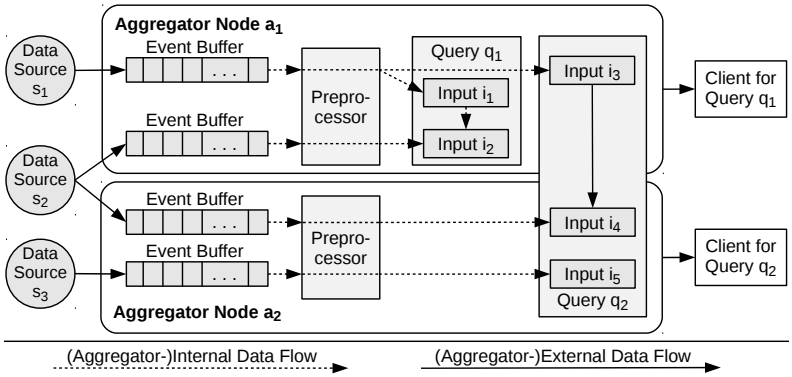


Fig. 1. Illustrative Instantiation of the Model for Distributed Event-Based Queries

The key aspects of the processing model are illustrated in Figure 1, which depicts two aggregator nodes (a_1,a_2) executing two queries (q_1,q_2) consisting of five inputs (i_1,...,i_5) in total. The query execution happens in two steps: firstly, the incoming events are buffered and preprocessed to become the actual inputs (e.g., average value of previous stock prices), and secondly the inputs are joined and combined as defined in the query specification. Dashed lines in the figure indicate aggregator-internal data flow, whereas solid lines stand for data exchanged with external machines. Aggregator a_1 orchestrates the execution of query q_1 and notifies the client of new results. We hence denote a_1 as the *master* aggregator for q_1 (analogously, a_2 is the master of q_2). The data source s_3 provides data for one single input (i_5), whereas inputs i_1/i_3 and i_2/i_4 are based on the events from s_1 and s_2, respectively. We observe that the events from s_2 are buffered both on a_1 and on a_2, which we denote buffer *duplication*. In Figure 1, an arrow pointing from an input i_x to i_y indicates a data dependency, i.e., that i_x provides some data which are required by i_y. In the case of i_1 and i_2, this passing

of data happens locally, whereas i_3 and i_4 are handled by different aggregators and hence data are transmitted over the network. We see that assigning i_3 to node a_1 has the advantage that the events from s_1 are buffered only once (for both i_1 and i_3), but is disadvantageous with respect to network traffic between the two aggregators a_1 and a_2. Conversely, s_2 is buffered on both aggregators, reducing the network traffic but requiring more memory. Section 4 deals with this tradeoff in more detail and further refines the optimization problem that we strive to solve.

4 Problem Formulation

In this section we provide a detailed definition for the problem of finding an optimal placement of processing elements in distributed event processing platforms. The basis for optimization is the current assignment of inputs to aggregators at some point in time, $cur : I \rightarrow \mathcal{P}(A)$, where $\mathcal{P}(A)$ denotes the powerset of A. We define that $cur(i) = \emptyset$ *iff* input i has not (yet) been assigned to any aggregator node. For now, we assume that each input (as soon as it has been assigned) is only handled by one aggregator, hence $|cur(i)| \leq 1, \forall i \in I$, but in Section 4.3 we will discuss the case that inputs are redundantly assigned to multiple aggregators for fail-safety. The desired result is a new assignment $new : I \rightarrow \mathcal{P}(A)$ in which all inputs are assigned to some aggregator, $|new(i)| = 1, \forall i \in I$. The difference between cur and new constitutes all inputs that need to be migrated from one aggregator to another, denoted as $M := \{i \in I \mid cur(i) \neq \emptyset \wedge cur(i) \neq new(i)\}$.

Migrating a query input may require to migrate/duplicate the event buffer of the underlying data source, if such a buffer does not yet exist on the target aggregator. The technical procedure of migrating event buffers and subscriptions is detailed in Section 6.1. The (computational) cost associated with this operation is proportional to the size of the buffer in bytes, expressed as $size : S \times (A \cup \{\emptyset\}) \rightarrow \mathbb{N}$. For instance, the buffer size for a source s on an aggregator a is referenced as $size(s, a)$. If the aggregator is undefined ($\emptyset$), then the buffer size function returns zero: $size(s, \emptyset) = 0, \forall s \in S$. The costs for migration of an input i from its current aggregator to a new node (function $migr$) only apply when the data source of i is not yet buffered on the new node, as expressed in Equation 1.

$$migr(i) := \begin{cases} size(source(i), cur(i)), & \text{if } source(i) \notin buf(new(i)) \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

In order to decide on actions for load balancing, we need to introduce some notion to express the current load of an aggregator node. In earlier work [14] we observed that the main influencing factor for the aggregators' workload in WS-Aggregation is the number of inputs and the data transfer rate of the underlying event streams. The transfer rate of data streams is therefore continuously measured and averaged over a given time interval (e.g., 1 minute). Every aggregator provides a metadata interface which can be used to retrieve this

monitoring information as a function $rate : (S \cup I) \rightarrow \mathbb{R}$, measured in kilobytes per second (kB/s). The $rate(s)$ of a data stream $s \in S$ is the transfer rate of external events arriving at the platform, and $rate(i)$ for an input $i \in I$ is the internal rate of events after the stream has passed the preprocessor. Based on the data transfer rate, we define the *load* function for an aggregator $a \in A$ as $load(a) := \sum_{s \in buf(a)} \sum_{i \in I_s} rate(s) \cdot c(i)$, where I_s denotes the set of all inputs targeting s, i.e., $I_s := \{i \in I \mid source(i) = s\}$, and $c : I \rightarrow \mathbb{R}$ is an indicator for the computational overhead of the preprocessing operation that transforms the data source s into the input i. The computational overhead depends on the processing logic and can be determined by monitoring. If no information on the running time of a processing step is available then $c(i)$ defaults to 1. For simplification, the assumption here is that n data streams with a *rate* of m kB/s generate the same load as a single data stream with a *rate* of $n * m$ kB/s. We denote the minimum load of all aggregators as $minload := min(\bigcup_{a \in A} load(a))$, and the difference between $minload$ and the load of an aggregator a as $ldiff(a) := load(a) - minload$.

To obtain a notion of the data flow, in particular the network traffic caused by external data flow between aggregator nodes (see Figure 1), Equation 2 defines the $flow$ between two inputs $i_1, i_2 \in I$. If the inputs are not dependent from each other or if both inputs are handled by the same aggregator, the $flow$ is 0. Otherwise, $flow$ amounts to the data transfer rate ($rate(i_1)$), measured in kB/s.

$$flow(i_1, i_2) := \begin{cases} rate(i_1), & \text{if } (i_1, i_2) \in deps(query(i_1)) \wedge new(i_1) \neq new(i_2) \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

Finally, Equation 3 introduces *dupl* to express buffer duplication. The idea is that each data source $s \in S$ needs to be buffered by at least one aggregator, but additional aggregators may also buffer events from the same source (see Figure 1). The function $dupl(s)$ hence subtracts 1 from the total number of aggregators buffering events from s.

$$dupl(s) := |\{a \in A \mid s \in buf(a)\}| - 1 \quad (3)$$

4.1 Optimization Target

We now combine the information given so far in a single target function to obtain a measure for the costs of the current system configuration and the potential benefits of moving to a new configuration. Overall, we strive to achieve a trade-off between three dimensions: balancing the load among aggregator nodes (L), avoiding duplicate buffering of events (D), while at the same time minimizing the data transfer between nodes (T). The goal of L is to keep each node responsive and to account for fluctuations in the frequency and size of incoming event data. The D dimension attempts to minimize the globally consumed memory, and T aims at a reduction of the time and resources used for marshalling/transmitting/unmarshalling of data.

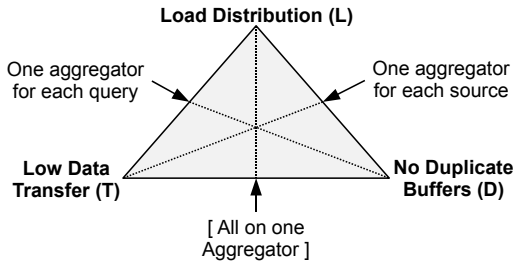


Fig. 2. Relationship between Optimization Targets

Figure 2 illustrates the tradeoff relationship as a "Magic Triangle": each pair of goals can be fulfilled separately, but the three goals cannot fully be satisfied in combination. For instance, a way to achieve a balanced load for all aggregators (L) in combination with no duplicate data source buffers (D) is to assign each source to a single aggregator. However, if a single query contains several interdependent data sources on different aggregators (which is likely to occur in this case), the aggregators possibly need to frequently transmit data. Conversely, to achieve load distribution (L) together with low data transfer (T), each query with all its inputs could be assigned to a single aggregator, but we observe that duplicate buffers come into existence if any two queries on different aggregators use the same underlying data source. Finally, to achieve both T and D at the same time, all requests could be assigned to one single aggregator. As indicated by the brackets in Figure 2, this possibility is generally excluded since we are striving for a distributed and scalable solution.

The function F' in Equation 4 contains the three components that are to be minimized. We observe that the three parts have different value ranges. Therefore, the target function includes user-defined weights (w_L,w_T,w_D) to offset the differences of the value ranges and to specify which of the parts should have more impact on the target function.

$$F' := w_L * \sum_{a \in A} ldiff(a) + w_T * \sum_{i_1, i_2 \in I} flow(i_1, i_2) + w_D * \sum_{s \in S} dupl(s) \rightarrow min! \quad (4)$$

We observe that the optimization target in Equation 4 only considers how good a new system configuration (i.e., assignment of inputs to aggregators) is, but not how (computationally) expensive it is to reach the new setup. To account for the costs of migrating query inputs, we make use of the $migr$ function defined earlier in Section 4 and use a weight parameter w_M to determine its influence. The final target function F is printed in Equation 5. Note that the additional one-time costs for migration in F are conceptually different from the cost components in F' which apply continuously during the lifetime of the queries.

$$F := F' + w_M * \sum_{i \in M} migr(i) \rightarrow min! \quad (5)$$

4.2 Elastic Scaling Using Cloud Computing

The premise for being able to change the current system configuration (moving from *cur* to *new*) as defined in the optimization target in Section 4.1 is that there are enough resources globally available to execute the migration tasks. To ensure that the resources are sufficient, we take advantage of *Cloud Computing* [2] techniques to elastically scale the platform up and down. To that end, each aggregator exposes metadata about the current stress level of the machine it is running on, and new machines are requested if all nodes are fully loaded. Conversely, if the nodes operate below a certain stress threshold, the queries can be rearranged to release machines back to the Cloud. For more details about elastic scaling in WS-Aggregation we refer to [13].

The notion of stress level is quite broad - it may include CPU and memory usage, list of open files and sockets, length of request queues, number of threads and other metrics. For simplification, we assume that the individual parts of the stress level function are added up and normalized, resulting in a function $stress : A \rightarrow [0, 1]$. Every aggregator provides a metadata interface which can be used to retrieve monitoring information and performance characteristics of the underlying physical machine. We use the upper bound of the stress level (value 1) to express that an aggregator is currently working at its limit and cannot be assigned additional tasks.

During the optimization procedure, the nodes' stress levels are continuously monitored. To determine whether a reconfiguration can be applied, it must be ensured that $\forall_{a \in A} : (stress(a) > \lambda) \implies \forall_{i \in I}(cur(i) = a \vee new(i) \neq a)$, for a configurable stress level λ (e.g., $\lambda = 0.9$). This criterion allows inputs to be removed from machines with high stress level, but prohibits the assignment of new query inputs. If the algorithm fails to find a valid solution under the given constraints, a new machine is requested from the Cloud environment and the optimization is restarted.

4.3 Extension: Robustness by Redundancy

So far, this paper has considered the case that each query input is handled by a single node. While this may be sufficient for most applications, in a safety-critical system it may be required to process inputs redundantly in order to mitigate the impact of machine outages. Our query optimization model therefore maps inputs to sets of aggregators ($cur : I \rightarrow \mathcal{P}(A)$), as defined in Section 4. As part of the specification of a query q, users define the required level of redundancy, $red(q) \in \{1, 2, \ldots, |A|\}$. The platform then duplicates the instantiation of the query, ensuring that each input is assigned to multiple aggregators, $\forall i \in inputs(q) : |cur(i)| \geq red(q)$. If, for any input $i \in I$, one of the aggregators $a \in cur(i)$ goes down, the event buffer migration technique allows to select a replacement aggregator for a and to copy the state from one of the replicated "backup" nodes $cur(i) \backslash \{a\}$.

5 Optimization Algorithm

The problem of finding an optimal input-to-aggregator assignment introduced in Section 4 is a hard computational problem, and the search space under the given constraints is prohibitively large (for a large number of inputs and many aggregators) and prohibits to compute exact solutions in feasible time. A formal proof of the problem's intractability is out of the scope of this paper, but we observe the combinatorial explosion as the algorithm needs to evaluate $\mathcal{O}(|A|^{|I|*red_{max}})$ solutions ($red_{max} := max(\bigcup_{q \in Q} red(q))$ denotes the maximum level of redundancy), each of which may potentially be optimal with respect to the target function F. In particular, pruning the solution space is hard to apply because during the search no solution can be easily identified as being suboptimal no matter what other solutions are derived from it. We therefore apply a metaheuristic and use Variable Neighborhood Search (VNS) [12] to approximate a near-optimal solution. The basic principle of VNS is to keep track of the best recorded solution x and to iterate over a predefined set of neighborhood structures which generate new solutions that are similar to x (for more details, see [12]). VNS has been successfully applied in a vast field of problem domains; one example is the multiplayer scheduling problem with communication delays [12], which has strong similarities to our problem. Figure 3 illustrates the encoding of a solution with 3 queries, 10 inputs and a maximum redundancy level of $red_{max} = 2$.

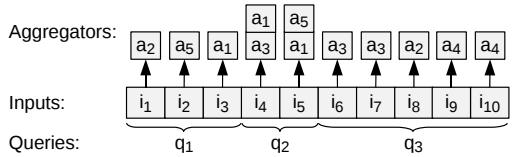


Fig. 3. Example of Solution Encoding in Optimization Algorithm with $red_{max} = 2$

5.1 Search Neighborhoods

The definition of neighborhoods in the VNS algorithm allows to guide the search through the space of possible solutions. In the following list of neighborhoods (NH), $temp : I \rightarrow A$ denotes the input-to-aggregator assignment of a temporary solution which is evaluated by the algorithm, and $temp' : I \rightarrow A$ denotes a solution which has been derived from $temp$ as specified by the NH.

- **avoid duplicate buffers NH**: This NH takes a random data source $s \in S$, determines all its inputs $I_s := \{i \in I \mid source(i) = s\}$ and the aggregators responsible for them, $A_s := \bigcup_{i \in I_s} temp(i)$. The NH then generates $|A_s|$ new solutions, in which all inputs in I_s are assigned to one of the responsible aggregators: $|\bigcup_{i \in I_s} temp'(i)| = 1 \wedge \forall i \in I_s : temp'(i) \in A_s$. When this neighborhood gets applied, the newly generated solutions will by tendency have less duplicate buffers.

- **bundle dependent inputs NH**: This NH selects a random query $q \in Q$ and generates new solutions in which all interdependent inputs of q are placed on the same aggregator node. More specifically, for each newly generated solution $temp'$ the following holds: $\forall (i_1, i_2) \in deps(q) : temp'(i_1) = temp'(i_2)$. Note that also transitive dependencies are affected by this criterion. The effect of this neighborhood is a reduced data traffic between aggregators.
- **equal data load per aggregator NH**: This NH selects the two aggregators $a_{max}, a_{min} \in A$ with $load(a_{max}) = max(\bigcup_{a \in A} load(a))$ and $load(a_{min}) = min(\bigcup_{a \in A} load(a))$, and generates a new solution by moving the input with the smallest data rate from a_{max} to a_{min}. More formally, let $I_{max} := \{i \in I \mid temp(i) = a_{max}\}$ denote the set of inputs that are assigned to aggregator a_{max} in the $temp$ solution, then the following holds in every solution derived from it: $temp'(\arg\min_{i \in I_{max}} rate(i)) = a_{min}$.
- **random aggregator swap NH**: This NH simply selects a random subset of inputs $I_x \subseteq I$ and assigns a new aggregator to each of these inputs, $\forall i \in I_x : temp'(i) \neq temp(i)$. The purpose of this NH is to perform jumps in the search space to escape from local optima.

VNS continuously considers neighborhood solutions to improve the current best solution until a termination criterion is reached. The criterion is either based on running time or solution quality. Furthermore, the algorithm only considers valid solutions with respect to the hard constraints (e.g., minimum level of redundancy as defined in Section 4.3). If a better solution than the current setting is found, the required reconfiguration steps are executed. The system thereby stays responsive and continues to execute the affected event-based queries.

6 Implementation

In the following we first briefly discuss how continuous queries are expressed and executed in WS-Aggregation, and then focus on implementation aspects concerning the migration of event buffers and subscriptions.

WS-Aggregation is implemented in Java using Web services [31] technology, and largely builds on WS-Eventing [29] as a standardized means to manage subscriptions for event notification messages. The platform is designed for loose coupling – aggregators may dynamically join and leave the system, and collaborative query execution across multiple aggregators is initiated in an ad-hoc manner. The endpoint references of currently available aggregator nodes are deployed in a service registry. WS-Aggregation employs a specialized query language named *WAQL* (Web services Aggregation Query Language), which is built on *XQuery* 3.0 [30] and adds some convenience extensions, e.g., to express data dependencies between query inputs. For the actual XQuery processing, we use the light-weight and high-performance *MXQuery* engine (http://mxquery.org/). More details can be found in [13, 14].

6.1 Migration of Event Buffers and Subscriptions

One of the technical challenges in our prototype is the implementation of event buffer migration, which becomes necessary when the result of the optimization (see Section 4) mandates that certain query inputs be moved between aggregators. The challenge is that transmitting the contents of a buffer over the network is a time-consuming operation, and new events for this buffer may arrive while the transmission is still active. At this point, it must be ensured that the arriving events are temporarily buffered and later forwarded to the target aggregator node. Therefore, transactionally migrating an event buffer while keeping the buffer state consistent at both the sending and the receiving node is a non-trivial task.

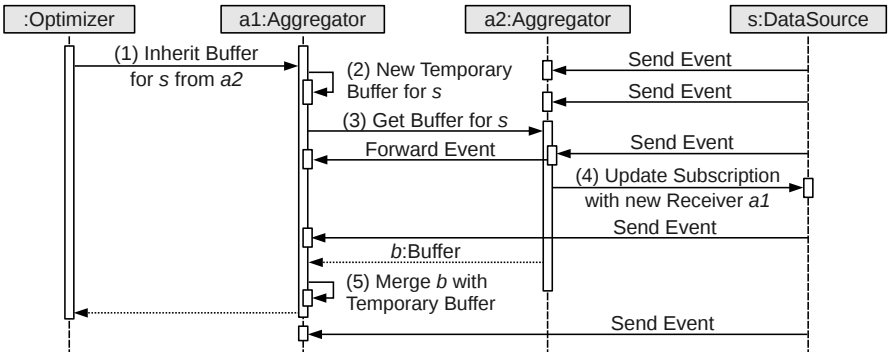


Fig. 4. Procedure for Migrating Buffer and Event Subscription between Aggregators

Figure 4 contains a UML sequence diagram which highlights the key aspects of our solution. It involves the optimizer component which instructs an aggregator $a1$ to *inherit* (become the new owner of) the event subscription for data source s together with the previously buffered events from aggregator $a2$ (point (1) in the figure). Data source d continuously sends events to the currently active subscriber. Before requesting transmission of the buffer contents from $a2$ (3), $a1$ creates a temporary buffer (2). Depending on the buffer size, the transmission may consume a considerable amount of time, and the events arriving at $a2$ are now forwarded to $a1$ and stored in the temporary buffer. The next step is to update the event subscription with the new receiver $a1$ (4). Depending on the capabilities of the data source (e.g., a WS-Eventing service), this can either be achieved by a single *renew* operation, or by a combination of an *unsubscribe* and a *subscribe* invocation. However, the prerequisite for keeping the event data consistent is that this operation executes atomically, i.e., at no point in time both $a1$ and $a2$ may receive the events. Finally, after the transmission has finished, the received buffer b is merged with the temporary buffer (5). If the execution fails at some point, e.g., due to connectivity problems, a rollback procedure is initiated and the process can be repeated.

7 Evaluation

To evaluate the performance effects that can be achieved with the proposed approach, we have set up an experimental evaluation in a private Cloud Computing environment with multiple virtual machines (VM), managed by an installation of *Eucalyptus*[1]. Each VM is equipped with 2GB memory and 1 virtual CPU core with 2.33 GHz (comparable to the *small* instance type in Amazon EC2[2]). Our experiments focus on three aspects: firstly, the time required to migrate event buffers and subscriptions between aggregators (Section 7.1); secondly, evolution of the network topology for different optimization parameters (Section 7.2); thirdly, performance characteristics of optimization weights (Section 7.3).

7.1 Migration of Event Buffers and Subscriptions

We first evaluate the effort required to switch from the current configuration to a new (optimized) configuration. For each input $i \in I$ there are three possibilities:

1. If $new(i) = cur(i)$ then there is nothing to do.
2. If $new(i) \neq cur(i) \wedge source(i) \in buf(new(i))$ then the new aggregator $a = new(i)$ needs to be instructed to handle input i, but no migration is required because the target buffer already exists on a.
3. If $new(i) \neq cur(i) \wedge source(i) \notin buf(new(i))$ then we need to perform full migration (or duplication) of the event buffer and subscription.

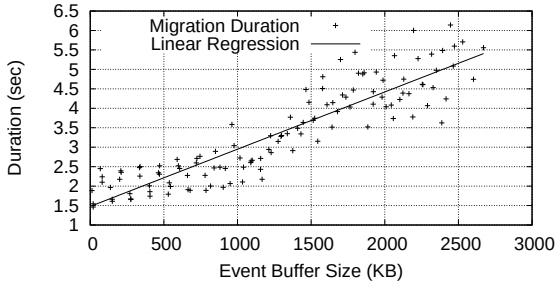


Fig. 5. Duration for Migrating Event Subscriptions for Different Buffer Sizes

Obviously, the most time-consuming and resource-intensive possibility in the list above is point 3. To measure the actually required time, we have executed various buffer migrations with different buffer sizes. Each data point in the scatter plot in Figure 5 represents a combination of buffer size and migration duration. The duration measures the gross time needed for the old aggregator a_1 to contact the new aggregator a_2, transmitting the buffer, feeding the buffer contents

[1] http://www.eucalyptus.com/
[2] http://aws.amazon.com/ec2

into the query engine on a_2, and freeing the resources on a_1. A linear regression curve is also plotted, which shows an approximate trendline (variance of residuals was 0.2735). Note that the numbers in the figure represent the net buffer size, that is, the actual accumulated size of the events as they were transported over the network (serialized as XML). The gross buffer size, which we evaluate in Section 7.3, is the amount of Java heap space that is consumed by the objects representing the buffer, plus any auxiliary objects (e.g., indexes for fast access).

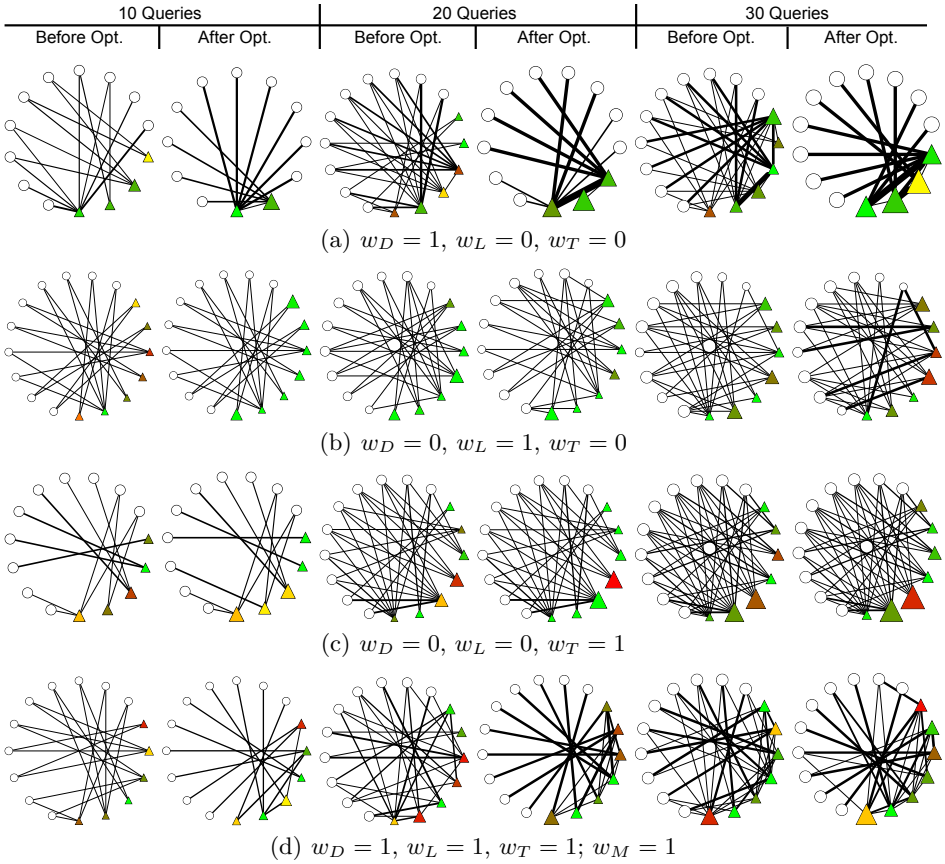
7.2 Evolution of Network Topology

The effect of applying the optimization is that the network topology (i.e., connections between aggregators and data sources) evolves according to the parameter weights. In our second experiment, we deployed 10 data sources (each emitting 1 event per second with an XML payload size of 250 bytes) and 7 aggregator nodes, and started 30 eventing queries in 3 consecutive steps (in each step, 10 queries are added). Each query instantiation has the following processing logic:

* Each query q consists of 3 inputs (i_q^1,i_q^2,i_q^3). The inputs' underlying data sources are selected in *round robin* order. That is, starting with the fourth query, some inputs target the same data source (because in total 10 data sources are available) and the buffer can therefore be shared.
* The events arriving from the data sources are preprocessed in a way that each group of 10 events is aggregated. The contents of these 10 events collectively form the input that becomes available to the query.
* Since we are interested in inter-aggregator traffic, each instance of the test query contains a data dependency between the inputs i_q^1 and i_q^2. This means that, if these two inputs are handled by different nodes, the results from i_q^1 are forwarded over the network to the node responsible for i_q^2.
* Finally, the query simply combines the preprocessed inputs into a single document, and the client continuously receives the new data.

Figure 6 graphically illustrates how the network topology evolves over time for different parameter settings. Each of the subfigures ((a),(b),(c)) contains six snapshots of the system configuration: for each of the 3 steps in the execution (10/20/30 queries), we record a snapshot of the system configuration *before* and *after* the optimization has been applied. In each step, the optimization algorithm runs for 30 seconds, and the best found solution is applied. Data sources are depicted as circles, aggregators are represented by triangles, and the nodes with data flow are connected by a line. The size of the nodes and lines determines the load: the bigger a circle, the more event subscriptions are executed on this data source; the bigger a triangle, the more data this aggregator is buffering; the thicker a line, the more data is transferred over the link. Furthermore, the aggregators' colors determine the stress level (green-yellow-red for low-medium-high).

We can see clearly that different optimization weights result in very distinct topological patterns. A characteristic outcome of emphasizing the w_D parameter (Figure 6(a)) is that few aggregators handle many event subscriptions and are



(a) $w_D = 1$, $w_L = 0$, $w_T = 0$

(b) $w_D = 0$, $w_L = 1$, $w_T = 0$

(c) $w_D = 0$, $w_L = 0$, $w_T = 1$

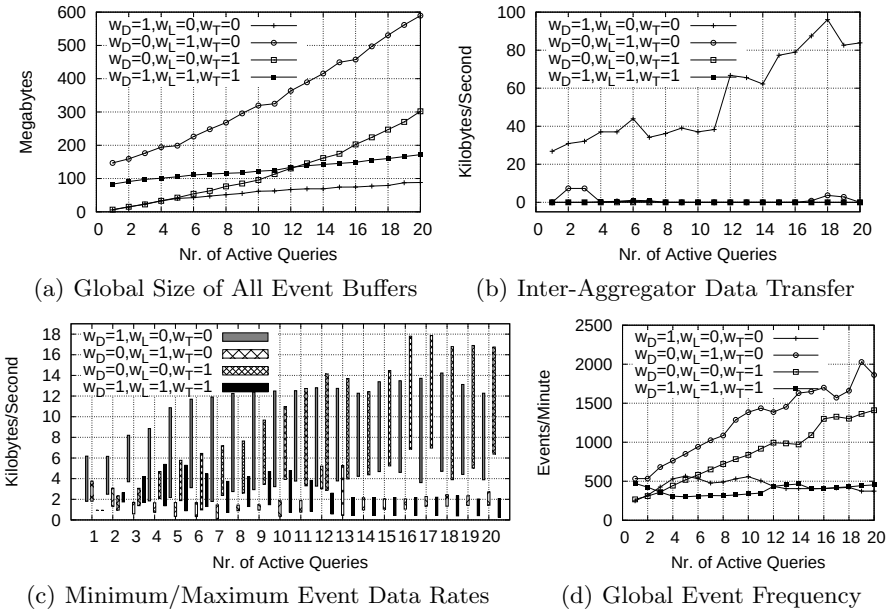
(d) $w_D = 1$, $w_L = 1$, $w_T = 1$; $w_M = 1$

Fig. 6. Effect of Optimization With Different Weights

hence loaded with a high data transfer rate. If the goal of preventing duplicate buffering is fully achieved, then there are at most $|S|$ active aggregators (and possibly less, as in Figure 6(a)), however, there is usually some inter-aggregator traffic required. In Figure 6(b) only the weight w_L is activated, which results in a more dense network graph. The inputs are spread over the aggregators, and in many cases multiple aggregators are subscribed with the same event source. Also in the case where w_T is set to 1, the resulting network graph becomes very dense. We observe that in Figure 6(c) there are no inter-aggregator connections, i.e., this setting tends to turn the network topology into a bipartite graph with the data sources in one set and the aggregator nodes in the second set. Finally, in Figure 6(c) all weights, including the penalty weight for migration (w_M) are set to 1. Note that the weights are subject to further customization, because setting equal weights favors parameters that have higher absolute values. In our future work, we plan to evaluate the effect of automatically setting the weights and normalizing the units of the optimization dimensions (D,L,T,M).

7.3 Performance Characteristics of Optimization Parameters

We now use the same experiment setup as in Section 7.2 and evaluate in more detail how the platform's performance characteristics evolve over time when optimization is applied. Again, 10 data sources and 7 aggregator nodes were deployed and queries were successively added to the platform. This time we took a snapshot 30 seconds after each query has been added for execution. The 4 subplots in Figure 7 illustrate the test results as a trendline over the number of active queries (x axis). To slightly flatten the curves, each experiment has been executed in 5 iterations and the numbers in the figures are mean values. The gross heap size of event buffer objects (Figure 7(a)) is determined using the Java instrumentation toolkit (*java.lang.instrument*) by recursively following all object references.



(a) Global Size of All Event Buffers

(b) Inter-Aggregator Data Transfer

(c) Minimum/Maximum Event Data Rates

(d) Global Event Frequency

Fig. 7. Performance Characteristics in Different Settings

Figure 7(a) shows that the global memory usage is particularly high (up to 600MB for 20 queries) for $w_L = 1$ and also for $w_T = 1$. Figure 7(b) depicts the inter-aggregator transfer, which in our experiments was quite high for $w_D = 1$, and near zero for the other configurations. The box plots in Figure 7(c) show the minimum and maximum event rates over all aggregators. We see that the absolute values and the range difference are high for $w_D = 1$ and $w_T = 1$, but, as expected, considerably lower for the load difference minimizing setting $w_L = 1$. Finally, the combined event frequency of all aggregators is plotted in Figure 7(d).

8 Conclusions

The placement of event processing elements plays a key role for the performance of query processing in distributed event-based systems. We have proposed an approach that performs dynamic migration of event buffers and subscriptions to optimize the global resource usage within such platforms. The core idea is that event buffers can be reused if multiple query inputs operate on the same data stream. We identified a non-trivial tradeoff that can be expressed as a "magic triangle" with three optimization dimensions: balanced load distribution among the processing nodes, minimal network traffic, and avoidance of event buffer duplication. Variable Neighborhood Search (VNS) has proven effective for exploring the search space and generating possible solutions.

We have exemplified our solution on the basis of several experiments carried out with the WS-Aggregation framework. The platform integrates well with the Cloud Computing paradigm and allows for elastic scaling based on the current system load and event data frequency. Our evaluation has illustrated how different optimization parameters can be applied to influence the evolution of the network topology over time. Furthermore, we have evaluated how different performance characteristics evolve in different settings. The experience we have gained in the various experiments conducted has shown that the (short-term) costs of migration or duplication are often outweighed by the (long-term) benefits gained in performance and robustness. As part of our ongoing work we are experimenting with very high-frequency event streams that cannot be handled by a single node. We envision an extension of the current query processing model to allow splitting up such streams to multiple aggregators. Furthermore, we are investigating Machine Learning techniques to automatically derive reasonable optimization parameters for the target function based on prior knowledge.

Acknowledgements The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 257483 (Indenica).

References

1. Agrawal, J., Diao, Y., Gyllstrom, D., Immerman, N.: Efficient Pattern Matching Over Event Streams. In: SIGMOD Int. Conference on Management of Data (2008)
2. Armbrust, M., et al.: Above the clouds: A berkeley view of cloud computing. Tech. Rep. UCB/EECS-2009-28, University of California at Berkeley (2009)
3. Ayad, A., Naughton, J.: Static optimization of conjunctive queries with sliding windows over infinite streams. In: SIGMOD Int. Conf. on Management of Data (2004)
4. Babu, S., Srivastava, U., Widom, J.: Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. ACM Transactions on Database Systems 29, 545–580 (2004)
5. Babu, S., Widom, J.: Continuous queries over data streams. ACM SIGMOD International Conference on Management of Data 30, 109–120 (2001)
6. Böhm, C., Ooi, B.C., Plant, C., Yan, Y.: Efficiently processing continuous k-nn queries on data streams. In: Int. Conf. on Data Engineering. pp. 156–165 (2007)

7. Bonfils, B.J., Bonnet, P.: Adaptive and decentralized operator placement for in-network query processing. Telecommunication Systems 26, 389–409 (2004)
8. Chen, J., DeWitt, D., Tian, F., Wang, Y.: NiagaraCQ: a scalable continuous query system for Internet databases. In: ACM SIGMOD International Conference on Management of Data. pp. 379–390 (2000)
9. Chen, Q., Hsu, M.: Data stream analytics as cloud service for mobile applications. In: Int. Symp. on Distributed Objects, Middleware, and Applications (DOA) (2010)
10. Cugola, G., Margara, A.: TESLA: a Formally Defined Event Specification Language. In: International Conference on Distributed Event-Based Systems (2010)
11. Etzion, O., Niblett, P.: Event Processing in Action. Manning Publications (2010)
12. Hansen, P., Mladenović, N.: Handbook of metaheuristics. Springer (2003)
13. Hummer, W., Leitner, P., Dustdar, S.: WS-Aggregation: Distributed Aggregation of Web Services Data. In: ACM Symposium On Applied Computing (2011)
14. Hummer, W., Satzger, B., Leitner, P., Inzinger, C., Dustdar, S.: Distributed Continuous Queries Over Web Service Event Streams. In: 7th IEEE International Conference on Next Generation Web Services Practices (2011)
15. Ioannidis, Y.E.: Query optimization. ACM Computing Surveys 28, 121–123 (1996)
16. Kephart, J., Chess, D.: The vision of autonomic computing. Computer 36(1) (2003)
17. Li, X., Agrawal, G.: Efficient evaluation of XQuery over streaming data. In: International Conference on Very Large Data Bases. pp. 265–276 (2005)
18. Liu, L., Pu, C., Tang, W.: Continual queries for Internet scale event-driven information delivery. IEEE Trans. on Knowledge and Data Engineering 11(4) (1999)
19. Luckham, D.C.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley Longman (2001)
20. Maybury, M.T.: Generating Summaries From Event Data. International Journal on Information Processing and Management 31, 735–751 (September 1995)
21. Motwani, R., et al.: Query processing, approximation, and resource management in a data stream management system. In: Conference on Innovative Data Systems Research (CIDR) (2003)
22. Mühl, G., Fiege, L., Pietzuch, P.: Distributed event-based systems. Springer (2006)
23. Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., Seltzer, M.: Network-aware operator placement for stream-processing systems. In: International Conference on Data Engineering (ICDE) (2006)
24. Roy, P., Seshadri, S., Sudarshan, S., Bhobe, S.: Efficient and extensible algorithms for multi query optimization. In: ACM SIGMOD International Conference on Management of Data. pp. 249–260 (2000)
25. Seshadri, S., Kumar, V., Cooper, B.: Optimizing multiple queries in distributed data stream systems. In: Int. Conference on Data Engineering, Workshops (2006)
26. Seshadri, S., Kumar, V., Cooper, B., Liu, L.: Optimizing multiple distributed stream queries using hierarchical network partitions. In: IEEE International Parallel and Distributed Processing Symposium. pp. 1–10 (2007)
27. Vitria: Complex Event Processing for Operational Intelligence. http://www.club-bpm.com/Documentos/DocProd00015.pdf (2010)
28. W3C: Web Services Addressing. http://www.w3.org/Submission/WS-Addressing/
29. W3C: Web Services Eventing. http://www.w3.org/Submission/WS-Eventing/
30. W3C: XQuery 3.0: An XML Query Language, http://www.w3.org/TR/xquery-30/
31. W3C: Web Services Activity (2002), http://www.w3.org/2002/ws/
32. Wu, E., Diao, Y., Rizvi, S.: High-performance complex event processing over streams. In: SIGMOD International Conference On Management of Data (2006)
33. Zhu, Y., Rundensteiner, E., Heineman, G.: Dynamic plan migration for continuous queries over data streams. In: SIGMOD Int. Conf. on Management of Data (2004)