Adaptive Selective Replication for Complex Event Processing Systems

Vision Paper

Franz Josef Grüneberger
SAP AG
Chemnitzer Str. 48
01187 Dresden, Germany
franz.josef.grueneberger@sap.com

Thomas Heinze
SAP AG
Chemnitzer Str. 48
01187 Dresden, Germany
thomas.heinze@sap.com

Pascal Felber
University of Neuchâtel
Switzerland
pascal.felber@unine.ch

ABSTRACT

As of today, active replication is used in complex event processing systems to enable near zero latency take over in case of host failures. Moreover, elastic complex event processing systems adapt their resource consumption to the actual system load. However, active replication is a coarse-grained approach demanding the duplication of all used resources. Therefore, we envision a system adopting adaptive fine-grained replication strategies allowing to trade off availability and used resources.

1. INTRODUCTION

The dissemination of high frequency event sources raises the demand to extract information from high velocity data streams. Prevalent application domains comprise automatic stock trading, credit card fraud detection, automated homecare, as well as scientific experiments, logistics, and telecommunication. To process high velocity data (>10.000 events per second) with low latency ($<100\,\text{ms}$), a new class of applications enabling the efficient analysis of data in real-time has ermerged. Prominent examples of such complex event processing systems comprise Borealis [1], IBM System S [9], and Yahoo! S4 [17].

Distributed complex event processing systems spanning thousands of hosts cope with very high data rates and extensive computations. However, the error probability increases with the number of components in a system. Because in data centers the probability for a single host to fail at least once a year is between 4 and 8 percent [6, 20] and distributed complex event processing systems execute all computations in memory, fault tolerance techniques have to be leveraged to circumvent unrecoverable data loss.

Various fault tolerance techniques like active replication [2] and check pointing [11, 14] have been studied in the context of complex event processing systems. To ensure failover with almost zero latency, we focus on active replication. However, actively replicating a system requires at least twice the resources. Therefore, this paper envisions techniques and outlines the major challenges for an elastic fault-tolerant complex event processing system that achieves high availability via adaptive selective replication. Specifically:

1. We present an approach that leverages spare resources to increase the availability of queries by replicating selected parts, i. e. operators, of the running system. Specifically, we present different strategies to select operators for replication. Moreover, striving for maximal availability, we introduce different placement strategies to deploy operators on hosts.

2. We envision an optimization component supporting the replication of queries to hit a certain availability target while adhering to resource constraints. Moreover, the component should assist minimizing the resource usage for a certain availability goal.

3. We explore the challenges arising from replication in elastic distributed complex event processing systems, where both queries and hosts are dynamically added to and removed from the system.

The remainder of this paper is structured as follows: Section 2 introduces the assumed system model. Section 3 presents an approach leveraging spare resources to improve the availability of queries. In Section 4 we propose an optimization component to minimize the resource consumption for a certain availability target. The challenges arising from an application of the approaches in an elastic system are outlined in Section 5. Related research is discussed in Section 6. Finally, Section 7 concludes the paper.

2. SYSTEM MODEL

2.1 Query Model

Queries in the system are continuous queries that can be added and removed at any point in time. As opposed to one-shot queries that are sent to the system and then produce a result once, continuous queries remain in the system for a certain period of time and produce results continuously. Let $\mathcal{Q} = \{q_1, q_2, \ldots, q_l\}$ be the set of queries in the system.

Queries are specified by the user in an event processing language (EPL). In the system queries are represented as

directed acyclic graphs (DAGs). Nodes represent operators that are connected by unidirectional edges. A query compiler transforms queries specified in EPL into a query graph.

Each operator has one or two inputs and multiple outputs. Operators with two inputs are referred to as binary operators. Each operator has a type defining its basic behaviour: source, projection, filter, aggregation, sequence, join, and destination. Besides a type all operators, except sources and sinks, have a predicate refining its functionality. For example in case of a filter operator the predicate specifies the filter condition.

The following example query calculates the average price of the SAP stock over the last 10 minutes.

```
INSERT INTO outStream
   SELECT compName, avg(tick)
   FROM tickStream WITHIN 600 seconds
   GROUP BY compName
   WHERE compName = "SAP";
```

The corresponding query graph, which is depicted in Figure 2, contains a source, filter, aggregation, and destination operator.

To minimize the number of operators in the system, all queries are optimized via a query optimization component, which analyses the query graph of already running queries with respect to reusable operators leveraging incremental multi query optimization (MQO) techniques [12]. The optimizer maintains an internal global query graph subsuming all currently deployed queries. When adding a new query, reusable parts are discovered using a breadth-first search on the global query graph. Figure 1 shows an example for multi query optimization with two queries. Operators of the already running query are depicted in the upper lane, whereas operators for the newly added query are depicted in the lower lane. Assuming that the same operator name indicates the same operator type and predicate, the operators S_1 and F_1 are part of both queries. Instead of redeploying this operators, they are reused from the already running query, which is illustrated via the grey box.

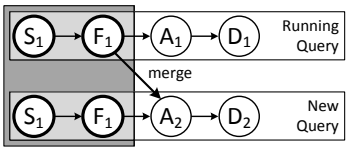

Figure 1: Incremental multi query optimization

2.2 Replication Model

To ensure near zero latency takeover, we assume active replication in the system. Each operator can have multiple replicas exhibiting exact the same behaviour as the primary operator. However, this approach results in the consumption of additional resources.

Query operators and replicas are deployed independently on available hosts. The current placement of operators on hosts is described via a placement function $plc : \mathcal{O} \rightarrow \mathcal{H}$, where $\mathcal{O} = \{o_1, o_2, \ldots, o_m\}$ is the set of operators for all queries in the system and $\mathcal{H} = \{h_1, h_2, \ldots, h_n\}$ denotes the set of used hosts.

2.3 Failure Model

According to the query model the system comprises several operators (timed processes) that are executed on a set of hosts. We assume that each host has a probability p to fail with a crash stop failure. Crash stop failures result in an immediate crash of all operators placed on the specific host. Moreover, we assume that Byzantine (value) errors caused by erroneous executions can be transformed into crash stop failures, e. g. by means of Software Encoded Processing [8]. In the following we consider the time period of one day. Due to the fact that in modern data centers the mean time to repair (MTTR) can be up to two days [6], we assume that crashed hosts will stay down for the whole day. Moreover, host crashes are assumed to be uncorrelated, i. e. if some hosts fail others will remain running.

A query is considered broken, iff for at least one of its operators neither a primary operator nor an operator replica is executed anymore.

2.4 Load Model

We assume a load model, which is based on work in the context of the data streaming system Borealis [23] and by Viglas et al. [19]. Each operator has one or two inputs and multiple outputs that are associated with a certain event rate. We assume that the input event rate for source operators is given and the output event rate for source operators equals the input event rate. Each operator op has a certain load $load(op)$. A load of 1.0 represents 100 % CPU usage in a fixed time interval - usually one second. This operator load is calculated as product of the input event rate of an operator multiplied by the cost c, which describes the time required by the operator to process a single tuple.

For the example query depicted in Figure 2 the load of the different operators can be specified as $load(S_1) = r_1 * c(S_1)$, $load(F_1) = r_1 * c(F_1)$, $load(A_1) = r_2 * c(A_1)$, $load(D_1) = r_3 * c(D_1)$. Since each operator is associated with a selectivity value, input event rates of operators can be calculated as linear combination of source stream rates and operator selectivity of predecessor operators. For example the input event rate r_3 of operator D_1 can be expressed as $r_3 = sel(A_1) * r_2 = sel(A_1) * (sel(F_1) * r_1)$.

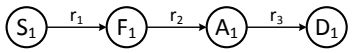

Figure 2: Example query graph

Moreover, each operator has incoming (net_{in}) and outgoing (net_{out}) network traffic. The incoming traffic $net_{in}(op)$ is calculated as product of input rate and input tuple size of the operator, whereas the outgoing traffic $net_{out}(op)$ is calculated by the output rate multiplied with the output tuple size.

For the sake of convenience in the remainder of this paper the term load is used interchangeably with CPU resources.

3. HEURISTIC REPLICATION

Complex event processing systems are exposed to a varying workload caused by different event rates as well as the addition and removal of queries. Figure 3 sketches a fictitious workload of a system processing queries for traffic monitoring. Peak loads indicate rush hour traffic. To deliver results for the queries in real-time, the system has to be equipped with

enough resources to handle those peak loads. However, since the peak load can only be estimated, phases of underprovisioning may occur, which are indicated by the dark gray areas. Moreover, this leads to phases of overprovisioning indicated by the dotted areas.

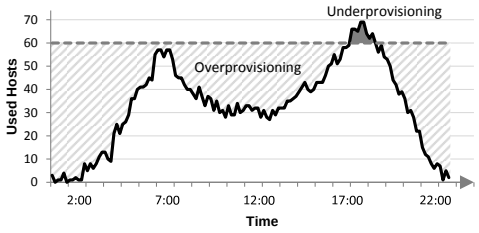

Figure 3: Workload of a data stream processing system

Heuristic replication targets the usage of such spare resources in distributed complex event processing systems to increase the availability of queries. Assume a system that comprises a set of n hosts $\mathcal{H} = \{h_1, h_2, \ldots, h_n\}$ with an overall CPU capacity of $load_{cap} = n * host_{cap}$. Query operators are placed according to an operator to host mapping *plc*. The unused CPU resources $load_{rem}$ can be calculated as

$$load_{rem} = load_{cap} - \sum_{op \in \mathcal{O}} load(op)$$

Since queries can be added to and removed from the system at runtime the amount of remaining resources usable to improve the availability of queries in the system changes over time. If the remaining load is smaller than the load of all queries in the system, not all operators can be replicated and, thus, a subset of operators has to be selected for replication. Thus, we propose a three step approach. First, all operators are rated according to their potential to improve query availability by means of an operator importance heuristic (see Section 3.1). Second, the set of operators as well as the replication level, i. e. the number of replica instances, is determined for each operator using an operator selection algorithm (see Section 3.2). Third, the replica instances are deployed on hosts according to an operator placement strategy (see Section 3.3). Finally, in Section 3.4 we present an evaluation.

3.1 Operator Importance Heuristics

The potential of operators to improve the availability of queries is rated using a normalized importance function $imp_{heur} : \mathcal{O} \rightarrow [0, 1]$ that associates each operator in the system with a normalized importance value. Depending on the source of information used for calculating this importance value, static and dynamic heuristics can be distinguished.

3.1.1 Static Heuristics

Static heuristics calculate the importance of operators based on properties observable in the global query graph. Due to the multi query optimization operators can be reused by multiple queries. Hence, if a reused operator fails, all depending queries crash. One possible heuristic, referred to as query out degree heuristic, is based on this observation and rates the importance of operators that are reused by multiple queries more important. The calculation of the query out degree heuristic can be expressed as

$$imp_{QOD}(op) = \frac{deg(op)}{|\mathcal{Q}|}$$

where $deg(op)$ is the number of queries leveraging the operator op, and $|\mathcal{Q}|$ is the number of queries running in the system.

3.1.2 Dynamic Heuristics

Dynamic heuristics assess runtime properties of operators and are, thus, dependent on models providing estimations if an operator is deployed for the first time. If for example the required resources are considered, the heuristics are dependent on the load model of the system (see Section 2.4). One example for a dynamic heuristic is the low utilization heuristic that tries to replicate as much operators as possible. Therefore, operators with a small load are preferred. The calculation of the heuristic can be expressed as follows:

$$imp_{LU}(op) = 1 - \frac{load(op)}{max(load(\mathcal{O}))}$$

where $max(load(\mathcal{O}))$ is the maximum operator load among all operators currently running in the system.

3.1.3 Query Level Heuristics

The heuristics introduced so far operate on an operator level, i. e. operators are treated independent of each other. Hence, operator-level heuristics might cause the selection of only most but not all operators for a query. Not selected operators are weak spots and decrease the achievable availability for the query tremendously. Therefore, we propose to combine operator importance values into query importance values, which are expressed using a normalized importance function for queries $impQ_{heur} : \mathcal{Q} \rightarrow [0, 1]$. Operator selection algorithms operating on query level heuristics will ensure that once a query was selected for replication all operators of that query are replicated. Only if not enough resources are available to replicate a full query, single operators would be selected.

3.2 Operator Selection Strategies

Operator selection strategies take care of the actual operator selection based on the operator importance and the remaining load $load_{rem}$ in the system. The calculated operator selection can be described as function $sel : \mathcal{O} \rightarrow \mathbb{Z}$, which associates each operator with a number of replicas referred to as replication level $rep(op)$.

3.2.1 Simple Operator Selection

A simple operator selection strategy sorts all operators descending based on their relative importance defined via imp_{heur}. Afterwards operators are selected for replication until all spare resources $load_{rem}$ are consumed. Selecting operators for replication includes the determination of the replication level, i. e. the number of replicas to create for a single operator. Different selection procedures can be established:

1. All operators can be selected for replication at least once if enough remaining resources are available. If afterwards spare resources are still available, the replication level for the already selected operators can be

increased stepwise. This procedure ensures that each operator is replicated at least once, if enough resources are available.

2. A replication level larger than one might be set directly for an operator. Hence, some operators might be excluded from replication, if already all resources are consumed.

3.2.2 Optimized Operator Selection

The simple operator selection strategy selects operators for replication based on either a static or dynamic heuristic. Thus, only one type of available information is incorporated at a time and the resulting selection of operators can lead to a non-optimal availability of queries. Therefore, we propose to augment the selection process based on a static heuristic with runtime information by modeling the operator selection problem as 0-1 knapsack problem [16].

Each operator $op_i \in \mathcal{O}$ has a value $v_i = imp_{heur}(op_i)$ and a weight $w_i = load(op_i)$. The maximum weight of the bag equals the remaining resources $load_{rem}$ in the system.

Because the 0-1 knapsack problem is NP-hard and load values of operators and, thus, the optimal solution changes continuously, we suggest an approximation by means of a combination of static and dynamic heuristics. The optimization is based on the intuition that operators with the greatest profit per weight unit have to be selected first. Thus, we propose to leverage a product of the query out degree and low utilization heuristic

$$imp_{QOD*LU}(op) = imp_{QOD}(op) * imp_{LU}(op)$$

as basis for the operator selection.

3.3 Fault-tolerant Operator Placement

Operators selected for replication via the operator selection algorithm have to be deployed on hosts. However, depending on the chosen operator placement for the same set of selected operators different availabilities can result.

3.3.1 Simple Bin Packing

A bin packing algorithm [4] for fault-tolerant operator placement minimizes the number of used hosts, so that idling hosts can be turned off to save energy. This property is in line with the notion of elasticity. We propose a bin packing algorithm, which is an extended version of a first-fit bin packing, which has a complexity of $O(mn)$, where m is the number of replicas that shall be placed and n the number of hosts.

Each host is modeled as bin, where the available CPU resources form the capacity. Replicas are first sorted in decreasing order according to their normalized importance and then assigned using their load as weight. Moreover, two constraints are ensured: (i) the network capacity of a host should not be exceeded, (ii) two replicas of the same operator are never placed on the same host. To reduce the consumed network bandwidth, possible hosts are ordered according to a neighboring factor representing the amount of predecessor and successor operators deployed on the same host.

3.3.2 Colored Bin Packing

Even though the simple bin packing algorithm is replica-aware due to the additional placement constraint, the system availability is not maximized because as many replicas of different operators as possible are placed on one host. Thus, multiple replicas crash in case of a host failure.

Therefore, we propose to use a bin packing algorithm with color constraints [5]. Besides load and network consumption, each replica will be associated with a unique color. The colored bin packing algorithm ensures that each host contains only replicas with at most c distinct colors, where c is a pre-defined positive integer. Because the number of replicas placed on the same host shall be minimized, an upper bound C with $c \leq C$ is specified for the color constraint. Then the placement problem can be formulated with an additional constraint that strives to minimize the parameter c. If no placement can be found without violating the upper bound C, the simple bin packing algorithm is used to calculate a placement.

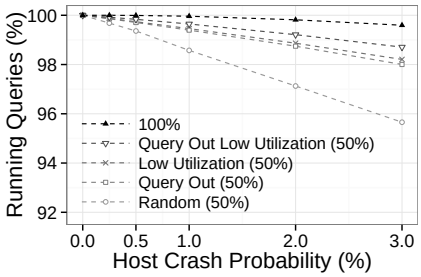

Figure 4: Comparison of query availability for different heuristics

3.4 Evaluation

We have performed a preliminary simulation-based evaluation of the proposed approaches. We used a query generator to generate queries based on six query templates that differ in structure and operator parameters. Operators were deployed on simulated hosts with a CPU capacity of $load_{cap} = 1$. We assume a system comprising 100 hosts. Moreover, operators were replicated at most once. To guarantee statistical correctness, 1000 runs were conducted for each experiment and values have been averaged.

Figure 4 depicts the percentage of remaining running queries after a time period of one day assuming different host failure probabilities $p = \{0.0025; 0.005; 0.01; 0.02; 0.03\}$ and different heuristics for rating the importance of operators. Because a constant system load $load(\mathcal{O}) = 100$ was generated, the stated percentages of resources available for replication are equal to the actual remaining load $load_{rem}$ in the system. Leveraging either the low utilization heuristic or the query out degree heuristic to select operators for replication improves the percentage of remaining running queries by approximately 1.7 percentage points compared to a random operator selection assuming a host failure probability of $p = 0.02$ and $load_{rem} = 50$. Using a combination of both heuristics improves the query availability further by 0.4 percentage points, resulting in 99.2 % remaining running queries.

Figure 5 shows the availability increase as a function of available resources for replication. A combination of query out degree and low utilization heuristic results in 99.2 % remaining running queries, if a host failure probability of $p = 0.02$ and $load_{rem} = 50$ is assumed. Moreover, the achievable percentage of running queries is only diminished

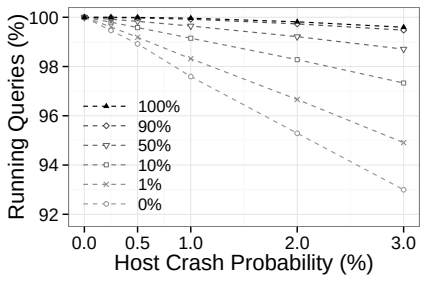

Figure 5: Query availability for combination of query out degree and low utilization heuristic

by approximately one percentage point compared to full replication, if 50 % less spare resources are used for replication and the host failure probability is $p = 0.03$. This result can be explained by the fact that with $load_{rem} = 50$ available for replication still 80 % of the query operators are replicated. Moreover, those operators are reused to a large extent or have small cost.

4. REPLICATION OPTIMIZER

Service level agreements are important in scenarios where a certain availability is required. However, achieving a higher availability by replicating more operators requires more resources. Therefore, the solicited availability and the resulting resource consumption should be traded off. Replication of operators in a system can be optimized with respect to two different optimization goals:

1. For a given resource limit $cost_{max}$, the availability of queries $A(\mathcal{Q})$ can be maximized.
2. For a given availability of queries $A(\mathcal{Q})$, the resource consumption $cost(\mathcal{Q})$ can be minimized.

The achieved availability of queries is influenced by the set of operators selected for replication, the number of replicas that is created for each of the selected operators, as well as the placement of replicas on hosts. All those influential parameters are reflected in the placement *plc* and, thus, the placement can serve as predictor for the achievable availability. To optimize the replication decisions according to the two specified optimization goals, additional models have to be incorporated. To estimate the costs for a placement, the load model for query operators can be leveraged. Moreover, an availability model is required that enables the estimation of the query availability resulting from a certain placement.

Assuming the availability of those models, the optimization problems can be formulated as follows:

$$max\{A(plc) \mid plc \in \mathcal{P}; cost(plc) \leq cost_{max}\}$$

and

$$min\{cost(plc) \mid plc \in \mathcal{P}; A(plc) \geq A(\mathcal{Q})\}$$

where $\mathcal{P}$ is the set of all possible placements resulting from different operator selections, different replication levels, as well as different placement strategies.

Those optimization problems can be solved using well-known optimization algorithms like genetic search [10]. However, to lower the effort for the optimization, heuristics to restrict the search space have to be developed.

5. DYNAMIC REPLICATION

Elastic data stream processing systems are able to cope with varying query as well as event load. Mechanisms to dynamically allocate and release hosts depending on the actual demand prevent costly overprovisioning and performance barriers due to underprovisioning. Ideally the system can scale out indefinitely to serve high event rates and scale in to lower the used resources in case of low utilization.

Processing queries in an elastic environment poses various challenges. Once new queries are submitted to the system, the encompassed operators have to be distributed to the running hosts. Because of operator reuse the load of already deployed operators changes. To not impair the performance of the system two reactive actions are taken: (i) overloaded hosts might be relieved by automatically migrating operators from one host to another, (ii) overload situations that cannot be resolved by moving operators from one host to another are resolved by splitting the operator into several operator instances that handle only a portion of the overall load and can be deployed independently.

However, handling replicas in a dynamic system is demanding:

- The complexity of the reconfiguration caused by the exoneration of overloaded hosts is increased since additional communication channels for replicas have to be maintained.
- If operators are split into several instances, all replicas have to be split too in order to maintain a consistent system state.
- Systems that are used in conjunction with heuristic replication (see Section 3), have to decide in case of spare resources whether to release hosts or to create additional replicas.
- Reconfiguration routines have to ensure that the new placement does not violate existing service level constraints for the queries (see Section 4).

6. RELATED WORK

Fault tolerance techniques for data stream processing systems like active replication [2], check pointing [14], and a combination of active and passive replication [24], are key enablers for our proposed approaches.

Moreover, operator placement algorithms are leveraged, which have been studies extensively by various authors. A survey is available in [15]. Repantis et al. [18] presented a procedure for replica placement considering performance constraints like end-to-end latency. The ZEN system [3] models different levels of availability in a systems and tries to assign most important operators to hosts with the highest availability. Another replication scheme based on graph coloring is presented in [22].

Optimization in the area of data stream processing systems is done for example to achieve an optimal overall utilization [21], or optimal utilization with limited resources [13].

An approach related to that in this paper has been studied by Fernandez et al. in [7]. The authors present an integrated approach to scale out streaming systems while achieving fault tolerance via check pointing. In this paper we focus, however, on active replication to ensure minimal latency and envision also scale in.

7. SUMMARY

As of today, many data stream processing systems use replication to ensure high availability in case of host failures. However, to replicate a system completely, at least twice the resources have to be allocated, which is costly.

We proposed a heuristic replication approach enabling the use of remaining system resources to increase the availability of queries. An operator selection algorithm is used to determine a subset of operators for replication that are then placed via an operator placement algorithm. Moreover, we suggested a replication optimizer which allows users to guide the replication while trading off cost and availability. Finally, the challenges arising from a combination of these techniques with elastic data stream processing systems were discussed. Using heuristic replication as well as the replication optimizer with a system reacting on changes, e. g. in event rate and selectivities, demands the adaptation of the placement routines. However, the normalized importance functions as well as the optimization routines might be reused unchanged.

To validate the approaches we simulated a heuristic replication approach comprising operator selection as well as operator placement strategies. Given a set of remaining resources, the fault tolerance of complex event processing systems is improved. Choosing a proper heuristic can improve the availability two percentage points compared to a random operator selection. Compared to full replication only one percentage point is lost spending 50 % less resources for replication.

In the future, we plan to implement the proposed approaches with a state-of-the-art streaming system.

8. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the Borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[2] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst.*, 33(1), 2008.

[3] N. Bansal, R. Bhagwan, N. Jain, Y. Park, D. S. Turaga, and C. Venkatramani. Towards optimal resource allocation in partial-fault tolerant applications. In *INFOCOM*, pages 1319–1327, 2008.

[4] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for np-hard problems. chapter Approximation algorithms for bin packing: a survey, pages 46–93. PWS Publishing Co., Boston, MA, USA, 1997.

[5] M. Dawande, J. Kalagnanam, and J. Sethuraman. Variable sized bin packing with color constraints. *Electronic Notes in Discrete Mathematics*, 7:154–157, 2001.

[6] J. Dean. Handling Large Datasets at Google: Current Systems and Future Directions. In *Data-Intensive Computing Symposium*, 2008.

[7] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *ACM International Conference on Management of Data (SIGMOD)*, New York, NY, 06/2013 2013. ACM, ACM.

[8] C. Fetzer, U. Schiffel, and M. Süßkraut. An-encoding compiler: Building safety-critical systems with commodity hardware. In *SAFECOMP*, pages 283–296, 2009.

[9] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: the system s declarative stream processing engine. In *SIGMOD Conference*, pages 1123–1134, 2008.

[10] D. E. Goldberg. *Genetic Algorithms*. Pearson Education, 2013.

[11] J.-H. Hwang, Y. Xing, U. Çetintemel, and S. B. Zdonik. A Cooperative, Self-Configuring High-Availability Solution for Stream Processing. In *ICDE*, pages 176–185, 2007.

[12] C. Jin and J. G. Carbonell. Predicate Indexing for Incremental Multi-Query Optimization. In *ISMIS*, pages 339–350, 2008.

[13] E. Kalyvianaki, W. Wiesemann, Q. H. Vu, D. Kuhn, and P. Pietzuch. Sqpr: Stream query planning with reuse. In *ICDE*, pages 840–851, 2011.

[14] Y. Kwon, M. Balazinska, and A. G. Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. *PVLDB*, 1(1):574–585, 2008.

[15] G. T. Lakshmanan, Y. Li, and R. E. Strom. Placement strategies for internet-scale data stream systems. *IEEE Internet Computing*, 12(6):50–60, 2008.

[16] S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations*. Wiley-Interscience series in discrete mathematics and optimization. J. Wiley & Sons, 1990.

[17] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *ICDM Workshops*, pages 170–177, 2010.

[18] T. Repantis and V. Kalogeraki. Replica placement for high availability in distributed stream processing systems. In *DEBS*, pages 181–192, 2008.

[19] S. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD Conference*, pages 37–48, 2002.

[20] K. V. Vishwanath and N. Nagappan. Characterizing cloud computing hardware reliability. In *SoCC*, pages 193–204, 2010.

[21] J. L. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, and L. Fleischer. Soda: An optimizing scheduler for large-scale stream-based distributed computer systems. In *Middleware*, pages 306–325, 2008.

[22] F. Xiao, T. Kitasuka, and M. Aritsugi. Economical and fault-tolerant load balancing in distributed stream processing systems. *IEICE Transactions*, 95-D(4):1062–1073, 2012.

[23] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. B. Zdonik. Providing resiliency to load variations in distributed stream processing. In *VLDB*, pages 775–786, 2006.

[24] Z. Zhang, Y. Gu, F. Ye, H. Yang, M. Kim, H. Lei, and Z. Liu. A hybrid approach to high availability in stream processing systems. In *ICDCS*, pages 138–148, 2010.