StreamCloud: An Elastic and Scalable Data Streaming System

Vincenzo Gulisano, *Student Member*, *IEEE Computer Society*,
Ricardo Jiménez-Peris, *Member*, *IEEE Computer Society*,
Marta Patiño-Martínez, *Member*, *IEEE Computer Society*,
Claudio Soriente, *Member*, *IEEE Computer Society*, and
Patrick Valduriez, *Member*, *IEEE Computer Society*

Abstract—Many applications in several domains such as telecommunications, network security, large-scale sensor networks, require online processing of continuous data flows. They produce very high loads that requires aggregating the processing capacity of many nodes. Current Stream Processing Engines do not scale with the input load due to single-node bottlenecks. Additionally, they are based on static configurations that lead to either under or overprovisioning. In this paper, we present *StreamCloud*, a scalable and elastic stream processing engine for processing large data stream volumes. *StreamCloud* uses a novel parallelization technique that splits queries into subqueries that are allocated to independent sets of nodes in a way that minimizes the distribution overhead. Its elastic protocols exhibit low intrusiveness, enabling effective adjustment of resources to the incoming load. Elasticity is combined with dynamic load balancing to minimize the computational resources used. The paper presents the system design, implementation, and a thorough evaluation of the scalability and elasticity of the fully implemented system.

Index Terms—Data streaming, scalability, elasticity

✦

1 INTRODUCTION

A number of application scenarios where massive amounts of data must be processed in quasi real-time are showing the limits of the traditional "store-then-process" paradigm [1]. In this context, researchers have proposed a new computing paradigm based on Stream Processing Engines (SPEs). SPEs are computing systems designed to process continuous streams of data with minimal delay. Data streams are not stored but are rather processed on-the-fly using *continuous queries*. The latter differs from queries in traditional database systems because a continuous query is constantly "standing" over the streaming tuples and results are continuously output.

In the last few years, there have been substantial advancements in the field of data stream processing. From centralized SPEs [2], the state of the art has advanced to SPEs able to distribute different queries among a cluster of nodes (interquery parallelism) or even distributing different operators of a query across different nodes (interoperator parallelism) [3]. However, some applications have reached the limits of current distributed data streaming infrastructures. For instance, in cellular telephony, the number of Call Description Records (CDRs) that must be processed to detect fraud in real time is in the range of 10,000-50,000 CDR/second. In such applications, most queries for fraud detection include one or more self-joins of the CDR stream using complex predicates, requiring comparison of millions of CDR pairs per second. Such applications call for more scalable SPEs that should be able to aggregate the computing power of hundreds of cores to process millions of tuples per second. The solution to attain higher scalability and avoid single-node bottlenecks of current SPEs lies in architecting a parallel-distributed SPE with intraoperator parallelism [4]. However, this requires addressing a number of additional challenges. Parallelization should be syntactically and semantically transparent. Syntactic transparency means that query parallelization should be oblivious to the user. That is, users write a regular (i.e., nonparallel) query that is automatically parallelized by the system. Semantic transparency means that, for a given input, parallel queries should produce exactly the same output as their nonparallel counterparts. On the other hand, resource usage should be cost effective. Many applications exhibit sudden, dramatic changes in the workload that can result in a variation of 1-2 orders of magnitude between peak and valley loads. For instance, in Asian cellular phone networks, streams of CDRs reach peaks of hundreds of thousands of records, while valley loads are in the range of thousands of records per second. A parallel but static SPE, that is, deployed on a fixed number of processing nodes, leads to either under-provisioning (i.e., the number of nodes cannot handle the workload) or overprovisioning (i.e., the allocated nodes are running below their full capacity). Underprovisioning results in the violation of service level agreements that can incur high economic costs. Even with best effort agreements, underprovisioning is responsible for losing unhappy users and raising bad reputation from disgruntled users. Overprovisioning is not cost effective as resources are not fully utilized. A parallel SPE should be *elastic* and adjust the amount of its resources (i.e.,

the number of allocated nodes) to the current workload. Moreover, elasticity should be combined with dynamic load balancing. Without dynamic load balancing, the system would provision new nodes as a result of uneven load distribution. Therefore, the saturation of a single node would lead to unnecessary provisioning of new instances. With dynamic load balancing, new nodes are provisioned only when the system as a whole does not have enough capacity to cope with the incoming load.

In this paper, we present *StreamCloud* (*SC*) [5], a scalable and elastic SPE. *SC* builds on top of Borealis [3] and provides transparent query parallelization. That is, users express regular queries that are automatically parallelized. A compiler takes the abstract query and generates its parallel version that is deployed on a cluster of nodes. The system features high scalability, elasticity, and load balancing. *SC* provides high scalability by exploiting intraoperator parallelism. That is, it can distribute any subset of the query operators to a large set of shared-nothing nodes. Logical data streams are split into multiple physical data substreams that flow in parallel, thus avoiding single-node bottlenecks. Communication across different nodes is minimized and only performed to guarantee semantic transparency. *SC* performs content-aware stream splitting and encapsulates the parallelization logic within smart operators that make sure that the outcome of the parallel execution matches the output of the centralized execution. *SC* monitors its activity and dynamically reacts to workload variations by reorganizing the load among its nodes as well as provisioning or decommissioning nodes. Elastic resource management is performed on-the-fly and shows very low intrusiveness, thus making provisioning and decommissioning cost effective. The contributions of this paper can be summarized as follows:

- a highly scalable and elastic SPE for shared-nothing clusters. *SC* is a full-fledged system, with a complete implementation currently being used for industrial applications;
- a novel parallelization approach that minimizes the distribution overhead;
- transparent parallelization of queries via a query compiler;
- effective algorithms for elastic resource management and load balancing that exhibit low overhead;
- a thorough evaluation of scalability and elasticity of the proposed system in a large cluster with 320 cores.

The rest of the paper is organized as follows. Section 2 provides a brief introduction to stream processing. Section 3 discusses various parallelization strategies while Section 4 shows how *SC* parallelizes queries to attain high scalability while preserving both syntactic and semantic transparency. Section 5 presents elastic resource management and load balancing algorithms used in *SC*. Section 6 provides a thorough evaluation of the scalability and elasticity of *SC* while Section 7 surveys relevant work in the area. The paper concludes in Section 8. Additional sections have been added in a supplementary document, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2012.24, which is organized as follows: Section 9 provides basic background on Data Stream Processing while Section 10 details the setup used for the evaluation as well as the queries and the data set. Finally, Section 11

provides a cost model for the query parallelization strategies considered in this manuscript.

2 DATA STREAM PROCESSING

Data Stream Processing is a novel computing paradigm particularly suited for application scenarios where massive amount of data must be processed with small delay. Rather than processing stored data like in traditional database systems, SPEs process tuples on-the-fly. This is due to the amount of input that discourages persistent storage and the requirement of providing prompt results. SPEs handle streams of tuples just as traditional database systems handle relations. A stream is a potentially infinite sequence of tuples sharing a given schema, denoted as $(A_1, A_2, \ldots, A_n)$; we refer to a generic attribute A_i of tuple t as $t.A_i$. We assume that, regardless of their schema, all tuples have a time stamp attribute set at the data sources. The data sources have clocks that are well synchronized with other system nodes as already done in [6]. When clock synchronization is not feasible, tuples can be time stamped at the entry point of the data streaming system. Queries used in SPEs are defined as "continuous" since they are continuously standing over the streaming data, i.e., results are pushed to the user each time the streaming data satisfies the query predicate. A query is defined as a direct acyclic graph where each node is an operator and edges define data flows. Typical query operators of SPEs are analogous to relational algebra operators and can be classified as stateless or stateful [7]. Stateless operators (e.g., Map, Union, and Filter) do not keep state across tuples and perform their computation solely based on each input tuple. Stateful operators (e.g., Aggregate, Join, and Cartesian Product (CP)) perform operations on sequences of tuples. Because of the infinite nature of the data stream, stateful operators perform their computation on sliding windows of tuples defined over a period of time (e.g., tuples received in the last hour) or as a fixed number of tuples (e.g., last 100 tuples). A detailed description of streams, operators, and queries can be found in Section 9 of the supplementary document, available online.

3 PARALLEL DATA STREAMING

In this section, we discuss the different alternatives we considered to parallelize queries in *SC* and the overhead associated with each of them. We also show how *SC* parallelizes queries and discuss the operators that encapsulate the parallelization logic and guarantee parallelization transparency.

3.1 Query Parallelization Strategies

When moving from a centralized to a parallel execution, the challenge lies in guaranteeing semantic transparency, i.e., that the output of the parallel execution matches the output of the centralized one, while minimizing overhead. Particular attention must be given to stateful operators, as we must ensure that all tuples that must be aggregated/joined together are processed by the same node. For instance, an Aggregate running on a particular node and computing the daily average expense of mobile phone users must process
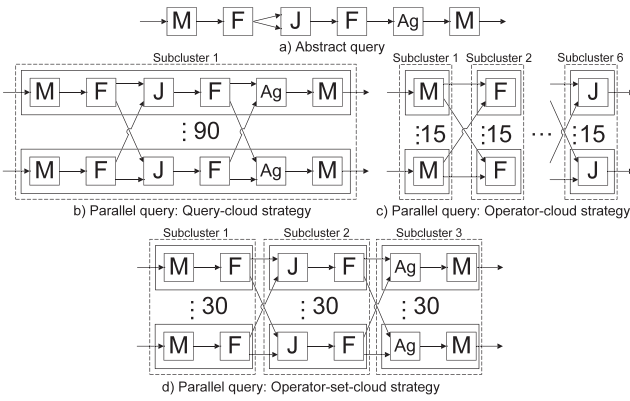
Fig. 1. Query parallelization strategies.

all tuples belonging to the same user in order to produce the correct result. We characterize the parallelization strategies in a spectrum. The position of each strategy in the spectrum depends on the granularity of the parallelization unit. At one extreme, we have a parallelization strategy that keeps the whole query as its parallelization unit. At the other extreme, we have a parallelization strategy that uses individual operators as parallelization units. Intermediate approaches define an arbitrary set of operators as their parallelization unit. The main two factors for the distribution cost are the number of hops performed by each tuple and the communication fan out of each node with the others. We illustrate three alternative parallelization strategies by means of the abstract query in Fig. 1a and its deployment on a cluster of 90 nodes. The query is composed by two stateful operators, i.e., a Join (J) and an Aggregate (Ag) and four stateless ones, i.e., two Maps (M) and two Filters (F).

- **Query-cloud strategy—QC** (Fig. 1b). If the parallelization unit is the whole query, the latter is deployed in each of the 90 nodes. Semantic transparency is attained by redistributing tuples just before each stateful operator. Therefore, tuples that should be aggregated/joined together are received by the same node. Each node receives one ninetieth of the incoming stream and communication takes place, for every stateful operator, from each node to all other nodes (peers). The number of hops of each tuple is equal to the number of stateful operators (i.e., 2) and the fan out for each node is equal to the number of nodes minus one (i.e., 89).
- **Operator-cloud strategy—OC** (Fig. 1c). In this strategy, the parallelization unit is a single operator. Each of them is deployed over a different subset of nodes (called subcluster). In this example, each subcluster has 15 nodes and communication happens from each node of one subcluster to all its peers in the next subcluster. The total number of hops is equal to the number of operators minus one (i.e., 5), while the fan out for each node is given by the size of the following subcluster (i.e., 15).
- **Operator-set-cloud strategy—SC** (Fig. 1d). The above distribution strategies exhibit a tradeoff between the distribution costs (i.e., fan out and

number of hops). The Operator-set-cloud strategy aims at minimizing both at the same time. The rationale is that, to guarantee semantic transparency, communication is required only before stateful operators. Therefore, we map parallelization units (called *subqueries*) to stateful operators. Each query is split into as many subqueries as stateful operators, plus an additional one, if the query starts with stateless operators. A subquery consists of a stateful operator followed by all the stateless operators connected to its output, until the next stateful operator or the end of query. If the query starts with stateless operators, the first subquery consists of all stateless operators before the first stateful one. As the query of Fig. 1a has two stateful operators plus a stateless prefix, there are three subqueries. Each subquery is deployed on a subcluster of 30 nodes. The first one contains the prefix of stateless operators, and the next two, one stateful operator with the following stateless operators. The total number of hops is equal to the number of stateful operators (minus one, if the query does not have a stateless prefix), while communication is required from all nodes of a subcluster to the nodes of the next subcluster (i.e., 2 and 30, respectively).

Clearly, the Query-cloud and Operator-set-cloud strategies minimize the number of tuple hops, while the lowest fan out overheads are achieved by the Operator-cloud and Operator-set-cloud strategies. A detailed cost model of the three query parallelization strategies can be found in Section 11 of the supplementary document, available online.

SC employs the Operator-set-cloud strategy as it strikes the best balance between communication and fan out overhead. According to the Operator-set-cloud strategy, queries are split into subqueries and each subquery is allocated to a set of *SC instances* grouped in a subcluster. In the rest of the paper, we use *instance* to denote a *SC* processing unit. This is because *SC* leverages multiple CPUs, cores, and hardware threads of a node, so any node can run as many *SC* instances as available processors. Data flow from one subcluster to the next one, until the output of the system. All instances of a subcluster run the same subquery, called *local subquery*, for a fraction of the input data stream, and produce a fraction of the output data stream. Communication between subclusters guarantees semantic transparency.

4 QUERY PARALLELIZATION IN STREAMCLOUD

In this section, we show how *SC* parallelizes a query and we introduce the operators that encapsulate the parallelization logic, Load Balancers (LBs), and Input Mergers (IMs).

Query parallelization is presented by means of the sample query in Fig. 2 that is used in mobile telephony applications to spot heavy customers. It receives a stream of CDRs with origin phone number, start, and end call time. The Map (M) operator computes the call duration from the start and end time. The Aggregate (Ag) operator groups CDRs by phone number and computes a per-hour average call duration. Finally, tuples with an average call duration
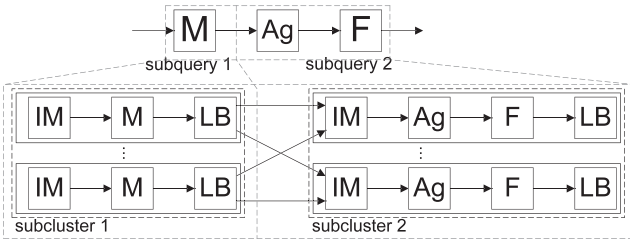
Fig. 2. Query parallelization in *SC*.

greater than a given threshold are forwarded by the Filter (F) operator. As the query has one stateful operator and a stateless prefix, it is split into two subqueries. Subquery1 consists of the Map operator while Subquery2 has the Aggregate and the Filter operators. They are allocated to subcluster 1 and 2, respectively. Given a subcluster, we term as *upstream* and *downstream* its previous and next peers, respectively.

To guarantee effective tuple distribution from one subcluster to the downstream one, output tuples of each subcluster are assigned to *buckets*. Tuple-buckets assignment is based on the attributes of the tuple. Given B distinct buckets and tuple $t = (A_1, A_2, \ldots, A_n)$, its corresponding bucket b is computed by hashing one or more of its attributes[1] modulus B, e.g., $b = H(A_i, A_j) \bmod B$. All tuples belonging to a given bucket are forwarded to and processed by the same downstream instance.

In order to distribute the buckets across N downstream instances, each subcluster employs a *bucket-instance map* (BIM). The BIM associates each bucket with an instance of the downstream subcluster, so that $BIM[b].dest$ provides the downstream instance that must receive tuples belonging to bucket b. In the following, we say that instance A "owns" bucket b (that is, A is responsible for processing all tuples of bucket b) if, according to the BIM of the upstream subcluster, $BIM[b].dest = A$. Tuple-buckets assignment and BIMs are endorsed by special operators, called *Load Balancers*. They are placed on the outgoing edge of each instance of a subcluster and are used to distribute the output tuples of the local subquery to the instances of the downstream subcluster.

Similarly to LBs on the outgoing edge of a subcluster, *SC* places another special operator, called *Input Merger*, on the ingoing edge. IMs take multiple input streams from upstream LBs and feed the local subquery with a single merged stream. The pseudocode of the algorithms executed by LBs and IMs is shown in Fig. 3.

4.1 Load Balancers

Load Balancers are in charge of distributing tuples from one local subquery to all its downstream peers. To guarantee that tuples that must be aggregated/joined together are indeed received by the same instance, upstream LBs of a stateful subquery must be enriched with *semantic awareness*. In particular, they must be aware of the semantics of the downstream stateful operator. In what follows, we discuss the parallelization of stateful subqueries for each of the stateful operators we have considered:

---

1. As explained later, the attributes used to compute the hash depend on the semantics of the downstream operator.

Load Balancer for Join & Aggregate
Upon: arrival of t:
1:   forward(t, nextSubcluster[BIM[hash($t.A_1$, $t.A_2$, ...) % B].dest])

Load Balancer for Cartesian Product
Upon: arrival of t from stream S (left or right):
2:   **for** each destination in BIM[hash($t.A_1$, $t.A_2$, ...) % B].dest **do**
3:      forward(t, nextSubcluster[destination])

Timeout Management at All Load Balancers
Upon: forwarding t to nextSubcluster[destination]:
4:   lastTS:= $t.ts$
5:   lastTime[destination]:= currentTime()
Upon: $\exists$ dest | $currentTime() \geq lastTime[dest] + d$:
     {d time units passed from last sent tuple}
6:   dummy.ts:= lastTS
7:   forward(dummy, nextSubcluster[destination])
8:   lastTime[destination]:= lastTime[destination]+d

Input Merger
Upon: arrival of t from stream i:
9:   buffer[i].enqueue(t)
10: **if** $\forall i$, buffer[i].nonEmpty() **then**
11:     t_o = earliestTuple(buffer)
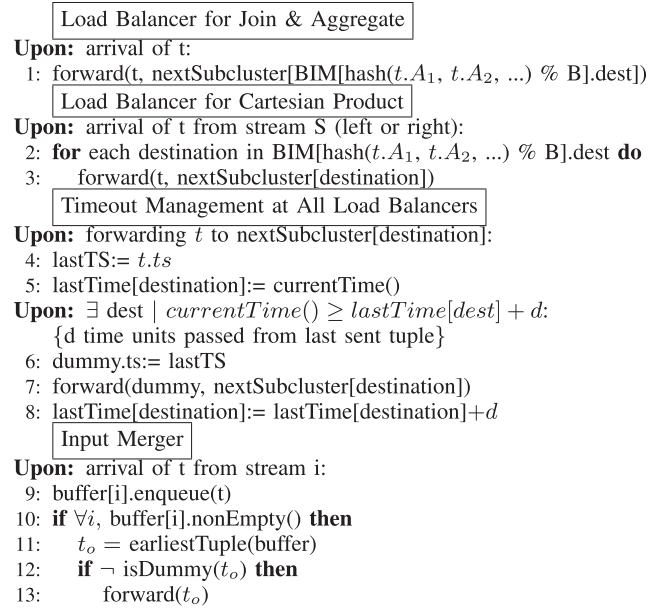12:   **if** $\neg$ isDummy(t_o) **then**
13:        forward(t_o)

Fig. 3. Pseudocode for load balancers and input mergers.

Join (in its variants of equijoin and general join, or Cartesian Product), and Aggregate.

4.1.1 Join Operator (Equijoin)

The Join we consider is an equijoin, i.e., it has at least one equality clause in its predicate.

SC uses a symmetric hash join approach [4]. Upstream LBs partition each input stream into B buckets and use the BIM to route tuples to the N instances where the Join is deployed (Fig. 3 line 1). The attribute specified in the equality clause is used at upstream LBs to determine the bucket and the recipient instance of a tuple. That is, let A_i be the attribute of the equality clause, then for each tuple t, $BIM[H(t.A_i) \bmod B].dest$ determines the recipient instance to which t should be sent. If the operator predicate contains multiple equality clauses, the hash is computed over all the attributes of those clauses. Therefore, tuples with the same values of the attributes defined in the equality clauses will be sent to the same instance and matched together.

4.1.2 General Join Operator (Cartesian Product)

The Cartesian Product operator differs from the Join operator in its predicate that can be arbitrarily complex (i.e., no equality clauses). Upstream left and right LBs partition their data into B_l and B_r buckets, respectively. The BIM of the LB associates one recipient instance for each $(b_l, b_r) \in B_l \times B_r$. Given a predicate over attributes A_i, A_j and input tuple t entering the upstream left LB, the tuple is forwarded to all instances $BIM[b_l, b_r].dest$ having $b_l = H(A_i, A_j) \bmod B_l$. Similarly, a tuple t' entering the upstream right LB, is forwarded to all instances $BIM[b_l, b_r].dest$ having $b_r = H(A_i, A_j) \bmod B_r$. From an implementation point of view, the BIM used to feed CP operators (i.e., the ones used at upstreams LBs) associates multiple recipient instances to each entry (Fig. 3 lines 2-3).

Fig. 4a depicts a sample query with two Maps (M_l and M_r) and a CP operator. M_l divides by two the incoming integers. M_r capitalizes incoming letters. The CP operator
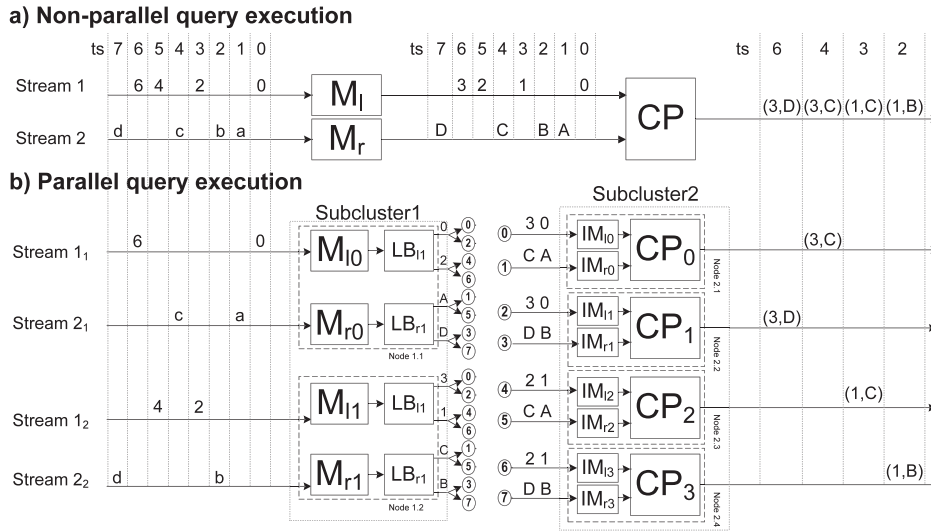
Fig. 4. Cartesian product sample execution.

has predicate $left.number \bmod 2 = 1 \wedge right.letter \neq A$ and a temporal window of 2 seconds. Fig. 4a also shows a sample input and the resulting output. Tuple time stamps are indicated on the top of each stream (the values right of the "ts" tag). Fig. 4b shows the parallel version of the query. According to the Operator-set-cloud strategy, the query is split into a stateless prefix (the two Map operators) and the CP operator. The stateless prefix is deployed on a subcluster of two nodes and the CP operator in a subcluster of four nodes ($N = 4$). Left and right incoming streams of the CP operator are 0-3 and A-D, respectively.

Both streams are split into two buckets (hence, $|B_l \times B_r| = 4$). The BIM for the left LB is built up so that tuples with integers in $\{0, 3\}$ are sent to CP instances 0 and 1, while tuples with integers $\{1, 2\}$ are sent to CP instances 2 and 3. The BIM for the right LB is built up so that tuples with letters in $\{A, C\}$ are sent to CP instances 0 and 2, while tuples with letters in $\{B, D\}$ are sent to CP instances 1 and 3. Each of the four CP instances performs one fourth of the whole Cartesian Product on the incoming streams.

4.1.3 Aggregate Operator

Parallelization of the Aggregate operator requires that all tuples sharing the same values of the attributes specified in the *Group-by* parameter, should be processed by the same instance. In the example of Fig. 2, the Aggregate groups calls by their originating phone number. Hence, data are

partitioned in a similar way to the Join operator (Fig. 3 line 1). That is, the set of attributes used in the *Group-by* parameter are hashed and a BIM is used to partition the stream across the N instances of the downstream subcluster.

4.2 Input Mergers

Just like LBs, Input Mergers are key to guarantee semantic transparency. A naïve IM that simply forwards tuples coming from its upstream LBs might lead to incorrect results. For instance, in Fig. 4a, the nonparallel CP operator receives each input stream in timestamp order, (0, 1, 2, 3) and (A, B, C, D). The evolution of the time windows for the nonparallel query is depicted in Fig. 5a (oldest tuples appear in the rightmost positions of each window). When a new tuple t is received on the left (resp. right) input stream, the right (resp. left) window is updated, removing those tuples t' such that $t.ts - t'.ts$ is greater than the window size.[2] In the example, tuple "0" is purged when tuple "C" arrives (step 4 in Fig. 5a); tuple "A" is purged on the arrival of tuple "2," and so on.

Using naïve IMs, input tuples at the downstream subcluster might be arbitrarily interleaved. This is because the tuple order of a logical stream might not be preserved when it is split in multiple physical streams, processed by different instances and finally merged. For example, CP_3 might receive $1, 2, D, B$ as input sequence. Tuple "D" causes tuple "1" to be purged. Hence, when tuple "B" arrives, it is not matched with tuple "1" and the output tuple (1,B) is be missed.

The transparent parallelization of *SC* (Fig. 3 lines 9-13) preserves the tuple order arrival of logical streams providing *transparent IMs*. The IM performs a merge of multiple time stamp ordered input streams coming from upstream LBs and feeds the local subquery with a single time stamp ordered input stream. As a result, the local subquery will produce a time stamp ordered output stream. To guarantee a correct sorting, it is sufficient that the IM forwards the tuple with the smallest time stamp, any time it has received at least one tuple from each input stream (lines 10-13). To

a) Window sequence and emitted tuples at the non-parallel CP operator

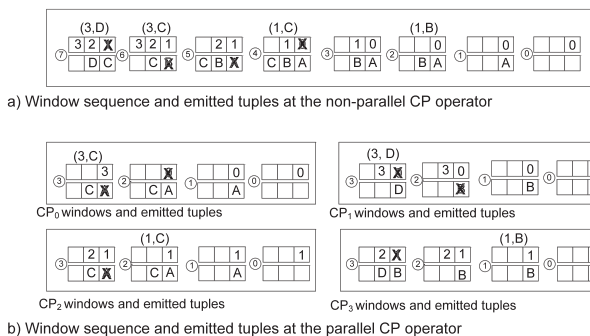b) Window sequence and emitted tuples at the parallel CP operator

Fig. 5. Cartesian product sample execution—operator window evolution.

2. Window update is independent of the interleaving on the input streams and only depends on the time stamps of the incoming tuples.
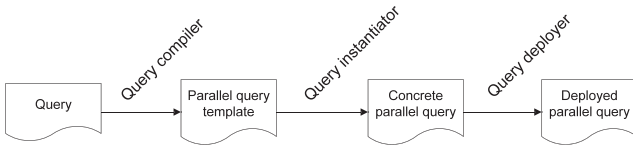
Fig. 6. Query compiler.

Fig. 7. Elastic management architecture.

avoid blocking of the IM, upstream LBs send dummy tuples for each output stream that has been idle for the last d time units. Dummy tuples are discarded by IMs and only used to unblock the processing of other streams.

In the example of Fig. 4, the transparent IM would endorse the window evolution shown in Fig. 5b, independently of the interleaving of the input streams. That is, despite output streams of each LB are not coordinated, IMs guarantee that tuples enter CP windows in timestamp order, thus assuring an execution equivalent to the nonparallel operator.

4.3 Query Compiler

SC builds on top of Borealis [3], a nonparallel SPE. Input queries are written in the Borealis query language and automatically parallelized by *SC* through its query compiler. The latter allows easy deployment of arbitrary complex queries over large clusters with just a few steps. The user is only required to input a query written in the Borealis query language, a list of available nodes (i.e., their IP addresses) and the elasticity rules (see Section 5). The process is shown in Fig. 6. The query compiler takes a regular query (e.g., the one in Fig. 1a) and splits it into subqueries according to the Operator-set-cloud parallelization strategy (e.g., Fig. 1d). At this stage, each subquery is augmented with LBs and IMs on their outgoing and ingoing edges, respectively. The query instantiator takes as parameters the number of available instances and the set of subqueries. It assigns a number of instances to each subquery, according to the per-tuple processing cost. The latter is measured as the time required to process a tuple by all the operators of the subquery, measured on a sample execution. The output of the query instantiator is a concrete parallel query that has placeholders for IPs and ports to be used by each *SC* instance. The final step is carried out by the query deployer that, given a deployment descriptor with IPs and ports of the available instances, deploys the concrete query assigning IPs/ports to each *SC* instance.

5 ELASTICITY

In this section, we show elastic resource management and dynamic load balancing of *SC*.

Fig. 7 illustrates a sample configuration with elastic management components. *SCs* architecture includes an *Elastic Manager (EM)*, *Resource Manager (RM)*, and *Local Managers (LMs)*. Each *SC* instance runs an LM that monitors resource utilization and incoming load, and is able to reconfigure the local LB (i.e., update its BIM). Each LM periodically reports monitoring information to the EM that aggregates it on a per-subcluster basis. Based on the collected data, the EM may decide to reconfigure the system either to balance the load, to provision or decommission instances. Reconfiguration decisions are taken and executed independently for each subcluster. If instances must be provisioned or decommissioned, the EM interacts
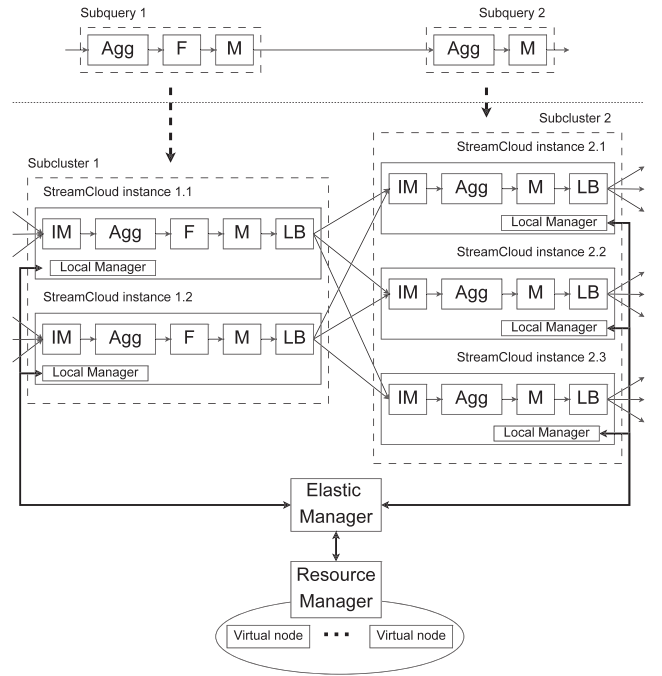
with the RM. The latter keeps a pool of instances where *SC* software is running but no query is deployed. We target critical application scenarios with high performance requirements that typically rely on private cloud infrastructures[3] (e.g., telecommunication companies). We assume *SC* to be already running as it does not consume much resources: the CPU usage of an idle *SC* process is in the order of 0.002 percent while its memory footprint is around 20 MB (hence the node can be used, e.g., for offline bill processing). Once the EM receives a new instance, the subquery is deployed and the instance is added to the subcluster that was about to saturate; we account for query deployment time (e.g., less than 10 ms seconds for the stateful subquery of Fig. 2) in all our experiments.

SC complements elastic resource management with dynamic load balancing to guarantee that new instances are only provisioned when a subcluster is not able to cope with the incoming load. Both techniques boil down to the ability to reconfigure the system in an online and nonintrusive manner.

5.1 Elastic Reconfiguration Protocols

Reconfiguring a subcluster requires transferring the ownership of one or more buckets from one instance (old owner) to another (new owner) in the same subcluster. For instance, bucket ownership of an overloaded instance may be transferred to a less loaded instance or to a new one. The basic idea is to define a point in time p so that tuples of bucket b earlier than p are processed by the old owner and tuples later than p are processed by the new one. This is straightforward for stateless operators. However, it is more challenging when reconfiguring stateful operators. Due to the sliding window semantics used in stateful operators, a

---

3. Public cloud systems, e.g., Amazon EC2, are not an option in our settings, as provisioning a node in the public cloud can take from several minutes to a few hours while starting *SC* only takes 1.2 seconds.
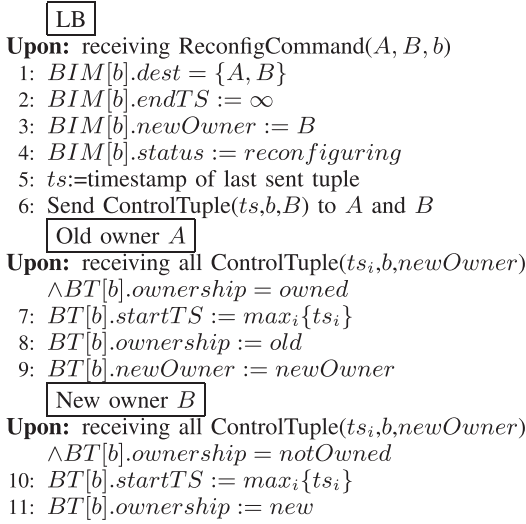
LB

Upon: receiving ReconfigCommand(A, B, b)
1: $BIM[b].dest = \{A, B\}$
2: $BIM[b].endTS := \infty$
3: $BIM[b].newOwner := B$
4: $BIM[b].status := reconfiguring$
5: $ts :=$ timestamp of last sent tuple
6: Send ControlTuple(ts, b, B) to A and B

Old owner A

Upon: receiving all ControlTuple($ts_i, b, newOwner$)
$\wedge BT[b].ownership = owned$
7: $BT[b].startTS := max_i\{ts_i\}$
8: $BT[b].ownership := old$
9: $BT[b].newOwner := newOwner$

New owner B

Upon: receiving all ControlTuple($ts_i, b, newOwner$)
$\wedge BT[b].ownership = notOwned$
10: $BT[b].startTS := max_i\{ts_i\}$
11: $BT[b].ownership := new$

Fig. 8. Reconfiguration start protocol.

Fig. 9. Sample reconfigurations.

tuple might contribute to several windows. For instance, in an Aggregate operator with a window size of 1 minute and an advance of 20 seconds, a tuple contributes to three consecutive overlapping windows. Thus, there will be tuples that need to be processed by both the old and new owners.

SC reconfigures a subcluster by triggering one or more reconfiguration actions. Each action changes the ownership of a bucket from the old owner to the new one within the same subcluster. Reconfiguration actions only affect the instances of the subcluster being reconfigured and the upstream LBs. We propose two alternative elastic reconfiguration protocols for stateful subqueries that trade completion time for communication between instances being reconfigured. Both protocols are explained considering ownership transfer of a bucket b from old owner A to new owner B. Proposed protocols have a common prefix; we first present this initial part and later detail the two protocols individually.

5.1.1 Reconfiguration Start

Fig. 8 shows the pseudocode common to both reconfiguration protocols. The process is initiated by the EM that decides to perform a reconfiguration either for provisioning, decommissioning, or load balancing purposes. The EM triggers the reconfiguration in a subcluster by sending to upstream LBs a reconfiguration command ($ReconfigCommand$) that specifies the bucket of which ownership is being transferred, the old and the new owner. At the end of the protocol, the instances being reconfigured should agree on a common time stamp to start the reconfiguration ($startTS$). Each LB proposes a time stamp based on the last tuple of bucket b that has been forwarded. The highest time stamp becomes the logical start of the reconfiguration. LBs maintain several variables for each entry $BIM[b]$; $BIM[b].endTS$ specifies the reconfiguration end timestamp, $BIM[b].newOwner$ specifies the new owner and $BIM[b].status$ is used to distinguish between *normal* processing and *reconfiguring* actions.

Fig. 9 shows a sample execution with the information exchanged between LBs and the instances being reconfigured. The figure only considers tuples and control messages related to bucket b. However, we stress that LBs, A and B might simultaneously process tuples belonging to other buckets. The bottom part of Fig. 9 shows the
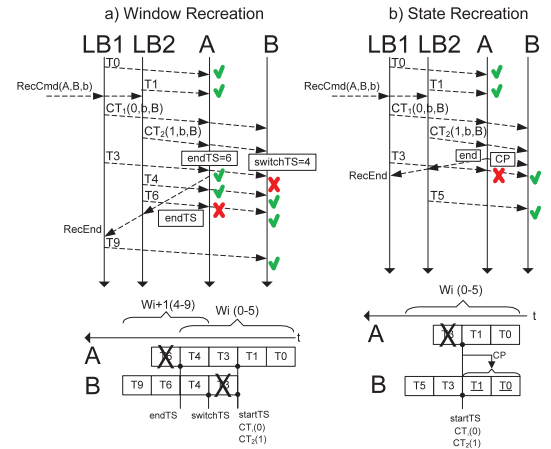
windows managed by A and B, respectively. Windows are time-based and have a size and advance of 6 and 2 time units, respectively. Initially, LB1 and LB2 route tuples corresponding to bucket b (T0, T1 in Fig. 9) to A. Upon reception of the reconfiguration command, each upstream LB updates its BIM setting $BIM[b].dest$ to both old and new owner, $BIM[b].endTS$ to ∞, $BIM[b].newOwner$ to the new owner of the bucket being reconfigured and $BIM[b].status$ to *reconfiguring*. Finally, LBs send a control tuple to both instances involved in the reconfiguration, indicating the time stamp of the last tuple processed (Fig. 8 lines 1-6). In the example of Fig. 9, LB1 sends $CT_1(0, b, B)$ and LB2 sends $CT_2(1, b, B)$. When both A and B receive all control tuples from upstream LBs, they update their *Bucket Table* (BT), setting $BT[b].startTS$ equal to the maximum time stamp received (Fig. 8 lines 7 and 10) that becomes the logical start of the reconfiguration ($BT[b].startTS = 1$ in the example of Fig. 9). A also sets $BT[b].newOwner$ to B (Fig. 8 line 9). Finally, A and B register their roles as old and new owner, respectively, setting the variable $BT[b].ownership$ (Fig. 8 lines 8 and 11).

5.1.2 Window Recreation (WR) Protocol

The Window Recreation protocol aims at avoiding communication between the instances being reconfigured. During a time interval proportional to the window size, tuples are processed, as part of separate windows, by both instances involved in the reconfiguration. When the old owner has processed all its windows, tuples are only sent to the new owner and regular processing is resumed.

The pseudocode is shown in Fig. 10. A is in charge of processing all windows with an initial time stamp earlier than $BT[b].startTS$ (Fig. 10 lines 6-12), while B will process all later windows (Fig. 10 lines 13-17). Due to the overlapping semantics of sliding windows, tuples later than $BT[b].startTS$ will be forwarded to both instances and processed as part of different windows. Given $BT[b].startTS$, windows size and advance, A computes the end time stamp of the last windows it has to process, i.e., $BT[b].endTS$.[4] Similarly, B computes the time stamp of the first window it has to process, i.e., $BT[b].switchTS$.

---

4. All windows of buckets being reconfigured share the same $startTS$. This is because *SC* enforces that all windows are aligned to the same time stamp as in [3].

LB

Upon: receiving t for bucket b
1: **if** $BIM[b].status = reconfiguring$
   $\wedge t.ts > BIM[b].endTS$ **then**
2:    $BIM[b].dest = BIM[b].newOwner$
3:    $BIM[b].status := normal$
4: Send t to $BIM[b].dest$
Upon: receiving EndOfReconfiguration(b, ts)
5: $BIM[b].endTS := ts$

Old owner A

Upon: $BT[b].ownership = old$
6: $BT[b].endTS := ComputeEndTS()$
7: Send EndOfReconfiguration$(b, BT[b].endTS)$ to upstream LBs
Upon: receiving tuple t for bucket $b \wedge BT[b].ownership = old$
8: **if** $t.ts < BT[b].endTS$ **then**
9:    process t
10: **else**
11:    discard t
12:    $BT[b].ownership = notOwned$

New owner B

Upon: $BT[b].ownership = new$
13: $BT[b].switchTS := computeSwitchTS()$
Upon: receiving tuple t for bucket $b \wedge BT[b].ownership = new$
14: **if** $t.ts < BT[b].switchTS$ **then**
15:    discard t
16: **else**
17:    $BT[b].ownership = owned$
      {Start regular processing of bucket b}

Fig. 10. Window recreation protocol.

LB

Upon: receiving t for bucket b
1: Send t to $BIM[b].dest$
Upon: receiving EnfOfReconfiguration(b)
2: $BIM[b].dest = BIM[b].newOwner$

Old owner A

Upon: $BT[b].ownership = old$
3: Send EndOfReconfiguration(b) to upstream LBs
4: cp:= Checkpoint(b)
5: Send(cp) to $BT[b].newOwner$
Upon: receiving t for bucket $b \wedge BT[b].ownership = old$
6: Discard t
7: $BT[b].ownership = notOwned$

New owner B

Upon: receiving t for bucket $b \wedge BT[b].ownership = new$
8: Buffer t
Upon: receiving Checkpoint(cp)
9: Install(cp)
10: Process all buffered tuples of bucket b
11: $BT[b].ownership = owned$
   {Start regular processing of bucket b}

Fig. 11. State recreation protocol.

$BT[b].endTS$ is sent by A to upstream LBs via an $EndOfReconfiguration$ message. LBs end the reconfiguration and resume regular tuple processing, i.e., tuples are only forwarded to B, when they receive the first tuple with a time stamp later than $BT[b].endTS$ (Fig. 10 lines 1-5).

Fig. 9a shows a sample execution where window size and advance are set to 6 and 2, respectively. Before receiving the $ReconfigCommand$ from the Elastic Manager, upstream LBs perform regular tuple processing and send each tuple to the current owner of the corresponding bucket. Upon receiving the $ReconfigurationCommand$, each LBs provides the time stamp of the latest tuple forwarded, via a control tuple: CT_1 carries time stamp 0 and CT_2 carries time stamp 1. Using the control tuples, the window size and advance, A computes $BT[b].startTS = 1$ and $BT[b].endTS = 6$; B computes $BT[b].switchTS = 4$. A becomes responsible for all windows up to W_i since its starting time stamp is lower than $BT[b].startTS$. B becomes responsible for all windows starting from W_{i+1} since its starting time stamp is greater than $BT[b].startTS$. Tuples $T3$ to $T4$ are sent from LBs to both instances because their time stamp is earlier than $BIM[b].endTS$. Tuple $T6$ should be sent only to B (its time stamp being 6) but it is sent to both instances because LB2 processes it before receiving the $EndOfReconfiguration$ message. Tuples $T3$ is discarded by B because its time stamp is earlier than $BT[b].switchTS$. Tuples $T6$ is discarded by A because its time stamp is equal to $BT[b].endTS$. Starting from tuple $T9$, LBs only forward tuples to B.

5.1.3 State Recreation (SR) Protocol

The Window Recreation protocol avoids communication between instances being reconfigured, but its completion time is proportional to the window size. Hence, if window size is large (e.g., 24 hours), completion time can be too long for the protocol to be effective. The State Recreation protocol aims at completing the reconfiguration independently of the window size. This is achieved by transferring the state of the bucket being reconfigured, from the old owner to the new one.

The pseudocode is shown in Fig. 11. Once $BT[b].startTS$ has been set, A sends the $EndOfReconfiguration$ message with $BT[b].endTS = BT[b].startTS$ to upstream LBs. All tuples with time stamp later than or equal to $BT[b].startTS$ are discarded by A. At this time, A also serializes the state associated with bucket b and sends it to B (Fig. 11 lines 3-7). Upstream LBs forward to both A and B all tuples processed after reception of the $ReconfigurationCommand$ that have a time stamp earlier than $BIM[b].endTS$. Tuples with a time stamp later than $BIM[b].endTS$ are only forwarded to B (Fig. 11 lines 1-2). B buffers all tuples waiting for the state of bucket b. Once the state has been received and installed, B processes all buffered tuples and ends the reconfiguration (Fig. 11 lines 8-11).

Fig. 9b shows a sample execution where window size and advance are set to 6 and 2, respectively. The execution resembles the one in the example of the Window Recreation protocol, up to the time when $BT[b].startTS$ is computed. Tuple $T3$ has time stamp later than $BIM[b].endTS$ but it is forwarded to both A and B because LB1 processes it before receiving the $EndOfReconfiguration$ message. The tuple is discarded by A. B processes $T3$ because it has already received the state associated to bucket b (denoted as CP in Fig. 9b). Tuple $T5$ is only sent to B since it is processed by LB2 after the reconfiguration has ended.

5.2 Elasticity Protocol

Elasticity rules are specified as thresholds that set the conditions that trigger provisioning, decommissioning or load balancing. Provisioning and decommissioning are triggered if the average CPU utilization is above the Upper Utilization Threshold (UUT) or below the Lower Utilization Threshold (LUT). Reconfiguration actions aim at achieving

ElasticManager

Upon: new monitoring period has elapsed
1: $U_{av} = \frac{\sum_{i=1}^{n} U_i}{n}$
2: **if** $U_{av} \notin [LUT, UUT]$ **then**
3:     old:= n
4:     $n := ComputeNewConfiguration(TUT, old, U_{av})$
5:     **if** $n > old$ **then**
6:         Provision(n-old)
7:     **if** $n < old$ **then**
8:         freeNodes:= Offload(old-n)
9:         Decommission(freeNodes)
10: $U_{sd} = \sqrt{\sum_{i=1}^{n}(U_i - U_{av})^2}$
11: **if** $U_{sd} > UIT$ **then**
12:     BalanceLoad(U_{sd})

Fig. 12. Elastic management protocol.

an average CPU utilization as close as possible to the Target Utilization Threshold (*TUT*).

Load balancing is triggered when the standard deviation of the CPU utilization is above the Upper Imbalance Threshold (*UIT*). A Minimum Improvement Threshold (*MIT*) specifies the minimal performance improvement to endorse a new configuration. That is, the new configuration is applied only if the imbalance reduction is above the *MIT*. The goal is to keep the average CPU utilization within upper and lower utilization thresholds and the standard deviation below the upper imbalance threshold in each subcluster.

SC features a *load-aware provisioning* strategy. When provisioning instances, a naïve strategy would be to provision one instance at a time (*individual provisioning*). However, individual provisioning might lead to cascade provisioning, i.e., continuous allocation of new instances. This might happen with steadily increasing loads when the additional computing power provided by the new instance does not decrease the average CPU utilization below *UUT*. To overcome this problem, *SC* load-aware provisioning takes into account the current subcluster size and load to decide how many new instances to provide in order to reach for *TUT*.

The protocol for elastic management is illustrated in Fig. 12. In order to enforce the elasticity rules, the EM periodically collects monitoring information from all instances on each subcluster via the LMs. The information includes the average CPU usage (U_i) and number of tuples processed per second per bucket (T_b). The EM computes the average CPU usage per subcluster, U_{av} (Fig. 12 line 1). If U_{av} is outside the allowed range, the number of instances required to cope with the current load is computed (Fig. 12 lines 2-4). If the subcluster is underprovisioned, new instances are allocated (Fig. 12 lines 5-6). If the subcluster is overprovisioned, the load of unneeded instances is transferred to the rest of the instances by the *Offload* function and the unneeded instances are decommissioned (Fig. 12 lines 7-9).

Load Balancing is triggered if $U_{sd} > UIT$ (Fig. 12 lines 10-12) and is based on a greedy algorithm[5] similar to Bubble-sort. Initially, instances are sorted by U_i and, for each

5. Optimal load balancing is equivalent to the bin packing problem that is known to be NP-hard. In fact, each instance can be seen as a bin with given capacity and the set of tuples belonging to a bucket b is equivalent to an object "to be packed." Its "volume" is given by the sum of all T_b at each instance of the subcluster.
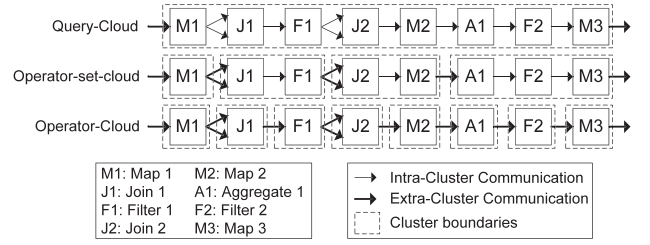

Fig. 13. Query used for the evaluation.

instance, buckets are sorted by T_b. At each iteration, the algorithm identifies the most and least loaded instances; the bucket with the highest T_b owned by the most loaded instance is transferred to the least loaded one. The CPU usage standard deviation (U_{sd}) is updated and iteration halts when the relative improvement (i.e., difference of standard deviation between two consecutive iterations) is lower than *MIT*. The provisioning strategy is encapsulated in the *ComputeNewConfiguration* function (Fig. 12 line 4). The interaction with the pool of free instances (e.g., a cloud resource manager) is encapsulated in functions *Provision* and *Decommission*. The load balancing algorithm is abstracted in the *BalanceLoad* function (Fig. 12 line 12).

6 EVALUATION

6.1 Evaluation Setup

The evaluation was performed in a shared-nothing cluster of 100 nodes (blades) with 320 cores. Further details of the evaluation setup can be found in Section 10 of the supplementary document, available online. All the experiments have three phases: warm up, steady state, and cool down. The evaluation was conducted during the steady-state phase that lasted for at least 10 minutes. Each experiment was run three times and we report averaged measurements.

6.2 Scalability Evaluation

In this section, we evaluate the scalability of *SC* with respect to the input load. We first evaluate the scalability and overhead of the parallelization strategies of Section 3.1. Two additional sets of experiments focus on individual operators and evaluate their scalability for increasing input loads. Finally, we highlight how *SC* takes full advantage of multicore architectures.

6.2.1 Scalability of Queries

The query of Fig. 13 was deployed in a cluster of 30 instances, according to the three parallelization strategies of Section 3.1. For each of the three approaches, Fig. 14 shows how the CPU usage is split among tuple processing, distribution overhead and idle cycles.

The query-cloud approach requires communication from each of the 30 instances to all other peers, for each of the three stateful operators (roughly $3 \cdot 30^2$ communication channels). Fig. 14 shows that the overall distribution overhead is around 40 percent. The remaining 60 percent is used for tuple processing.

The operator-cloud strategy shows a distribution overhead of more than 30 percent, while CPU ratio used to process tuples is roughly 35 percent. The unused CPU cycles (tagged as "idle" in Fig. 14) are related to the difference between the nominal subcluster sizes and the
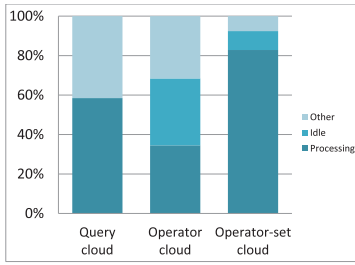
Fig. 14. Parallelization strategies evaluation—CPU usage breakdown.

actual ones. For instance, when computing the optimal distribution plan, an operator might require 4.3 instances that translates to an actual assignment of five instances, thus leading to one CPU that is not fully utilized.

The operator-set-cloud approach exhibits the lowest communication overhead (roughly 10 percent). As with the previous approach, the difference between nominal and actual subcluster sizes lead to unused resources. However, since fewer subclusters are generally used, the "idle" percentage is lower than that of the operator-cloud approach.

The upper part Fig. 15 shows the scalability of the three approaches, using up to 60 instances. For the query-cloud approach, we could only use half of the instances because the fan-out overhead with more than 30 instances was already exceeding the available resources at deployment time. For each strategy, different subcluster sizes have been evaluated and we only report the configurations that achieved the highest throughput. The *SC* approach (operator-set-cloud) attains a performance that is 2.5 to 5 times better than operator cloud and query cloud, respectively.

The bottom part of Fig. 15 also shows the evolution of the CPU usage for increasing loads. The query-cloud and operator-set-cloud approaches reach 100 percent CPU utilization. However, the former hits the maximal CPU usage with low input loads ($\leq 10{,}000$ tuples per seconds or t/s) while the operator-set-cloud approach sustains up to 70,000 t/s. The operator-cloud approach shows a maximal CPU utilization of 60 percent; roughly 40 percent of the CPU is not used.

6.2.2 Scalability of Individual Operators

This set of experiments focuses on the scalability of subclusters with one deployed operator (i.e., Aggregate, Map, Join, and Cartesian Product) and shows the associated overhead (CPU utilization and queue sizes). All the experiments share the same input schema: a CDR consisting of calling and called number, call start and end time, district, latitude and longitude coordinates, and emission time stamp.

The experiments show the throughput behavior as the injected load increases. We experienced a common pattern that can be summarized in three stages: 1) an initial stage with increasing throughput, CPU utilization below 100 percent and empty queues; 2) a second stage where throughput increases with a milder slope: instances are close to saturation and queues start growing; 3) a final stage showing 100 percent CPU utilization where queues reach their limits and throughput becomes stable. Each stage can be clearly seen in the bottom parts of Figs. 16a, 16b, 16c, and 16d, where solid lines show the CPU usage (left Y axis) and dotted lines show queue lengths (right Y axis).
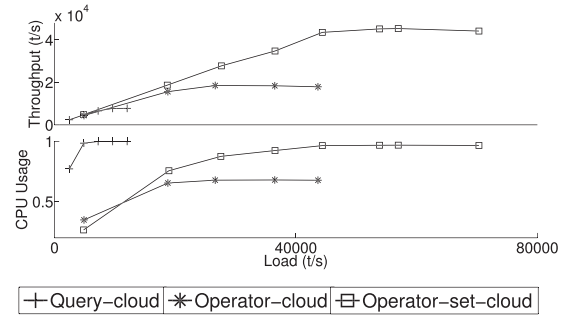

Fig. 15. Parallelization strategies evaluation—Scalability.

The Aggregate operator computes the number of calls and average duration by grouping results by district; windows size and advance are set to 60 and 10 seconds, respectively. It exhibits a linear evolution of the throughput for different input rates and number of instances (upper part of Fig. 16a). Twenty instances manage an input rate of roughly 200,000 t/s while 40 instances double the handled input rate, reaching roughly 400,000 t/s.

For each input CDR, the Map operator computes the call duration, given the start and end time. The upper part Fig. 16b shows a linear throughput where 20 instances process 230,000 t/s; doubling the number of available instances the throughput reaches 450,000 t/s.

The Join operator matches phone calls made by the same user every minute and computes their distance in time and space. The upper part of Fig. 16c shows the evolution of the throughput in comparisons per second (c/s) for all configurations. From 20 to 40 instances, the throughput almost doubles, which means that scalability is almost linear.

The evaluation of the Cartesian Product operator is shown in the upper part of Fig. 16d. Also in this case, the scale out is almost linear. Twenty instances achieve close to two billions c/s and 40 instances reach four billions c/s.

6.2.3 Scaling with Fixed-Size Windows

In this set of experiments, we focus on CP operators defined over fixed-size windows. The operator is the same as the one used in Section 6.2.2. The window size has been set to 20,000 tuples. Each tuple is 12 bytes long yielding to a memory footprint of 240 Kbytes.

Distributing CP operators defined over fixed-size windows has the advantage of reducing the number of comparisons per instance. The former translates to a smaller per-tuple processing time. For example, in a nonparallel CP operator with windows of size 20,000 and an average input load of 1,000 t/s, the operator must perform 20 million c/s. Distributing the operator over N instances, each instance would perform $\frac{1}{\sqrt{N}}$th of 20 millions comparisons. For instance, if $N = 4$, the number of c/s per instance drops to 10 million. In other words, the per-tuple processing time is halved.

Fig. 17 shows superlinear scalability of the throughput. In fact, as the per-tuple processing cost at each instance decreases, the parallel CP operator outperforms its non-parallel counterpart by a factor of N^2.
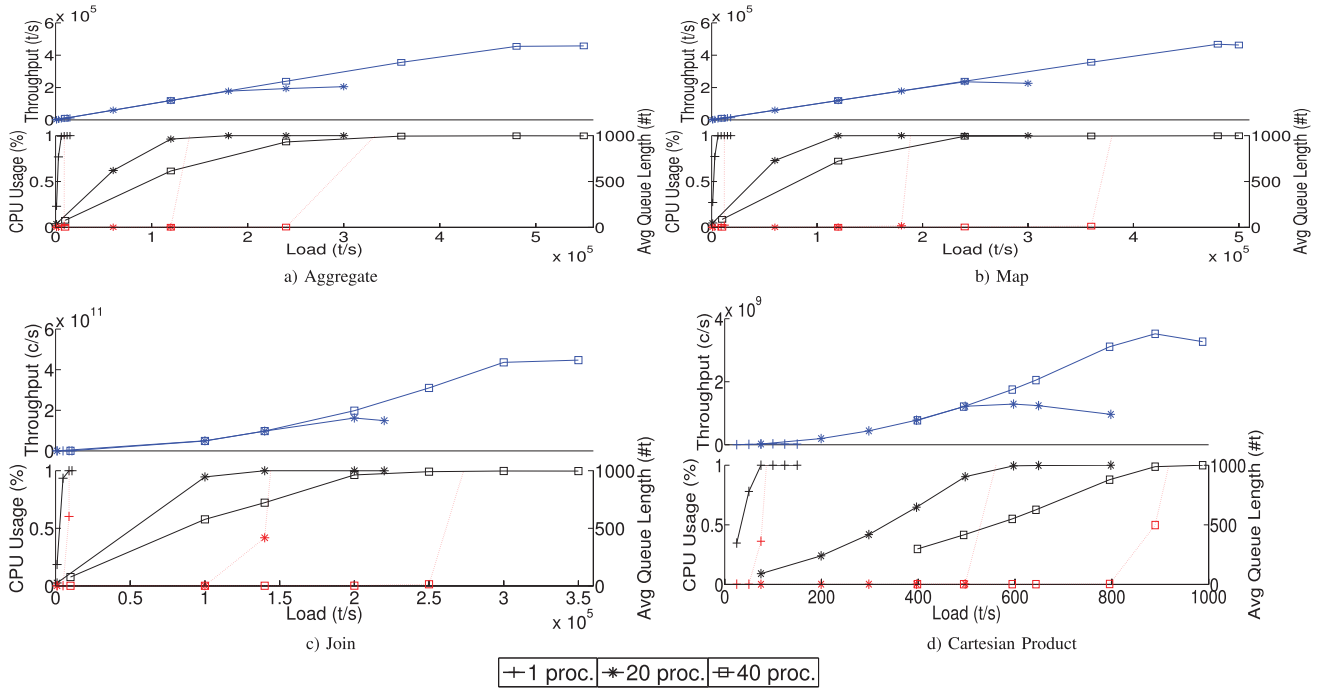
Fig. 16. Individual operator and query scalability evaluation results. Solid lines in the upper part of each figure refer to throughput (t/s). In the bottom part of each figure, solid lines refer to CPU usage (percent) while dotted lines refer to queue lengths (#t).

6.2.4 Multicores

In this experiment, we aim at quantifying the scalability of *SC* with respect to the number of available processors in each node, that is, to evaluate whether *SC* is able to effectively use the available CPUs/cores/hardware threads of each node.

We focus on the Join operator of Section 6.2.2 deployed over 1, 10, and 20 quad-core nodes, respectively. On each node, up to four *SC* instances were deployed.

Fig. 18 shows linear scalability with respect to the number of *SC* instances per node. For example, one node running four *SC* instances handles 0.4×10^{11} c/s, 10 nodes running four *SC* instances each can reach 4×10^{11} c/s and 20 nodes with four *SC* instances each go up to 8×10^{11} c/s. This is because *SC* defines three threads for receiving, processing, and sending tuples. As the scheduling policy enforces only one active thread at a given point in time, we can deploy as many *SC* instances as available cores and scale with the number of cores per node.

6.3 Elasticity Evaluation

This section presents the experiments performed to evaluate the elasticity of *SC*.

6.3.1 Elastic Reconfiguration Protocols

This set of experiments aims at evaluating the tradeoff between the elastic reconfiguration protocols of Section 5. We run the stateful subquery of Fig. 2, with window size of 1, 5, and 10 seconds (WS labels in Fig. 19), respectively. UUT is set to 80 percent, that is, when the average CPU utilization reaches 80 percent, the Elastic Manager provisions a new instance and running instances transfer ownership of some of their buckets to the new one. For each reconfiguration protocol, Fig. 19 shows the completion times and the amount of data transferred between instances. Completion time is measured from the sending of the reconfiguration command to the end of the reconfiguration at the new owner. Figs. 19a, 19b, and 19c show the time required to complete a reconfiguration from one to two instances, from 15 to 16 instances and from 30 to 31 instances, respectively, with an increasing number of transferred windows. Completion
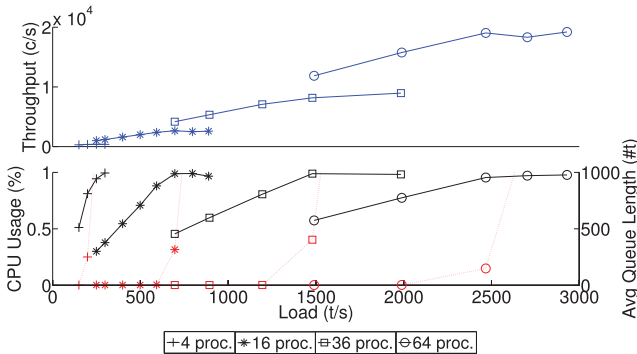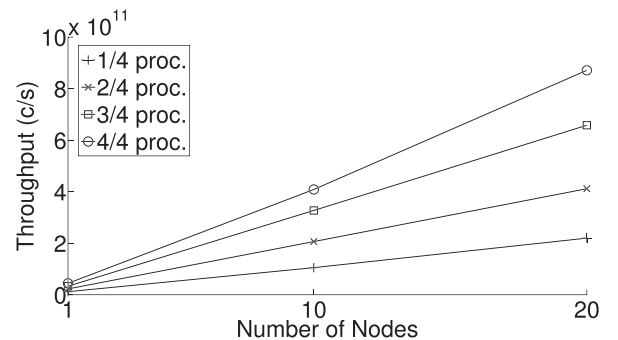

Fig. 17. CP over fixed-size windows.

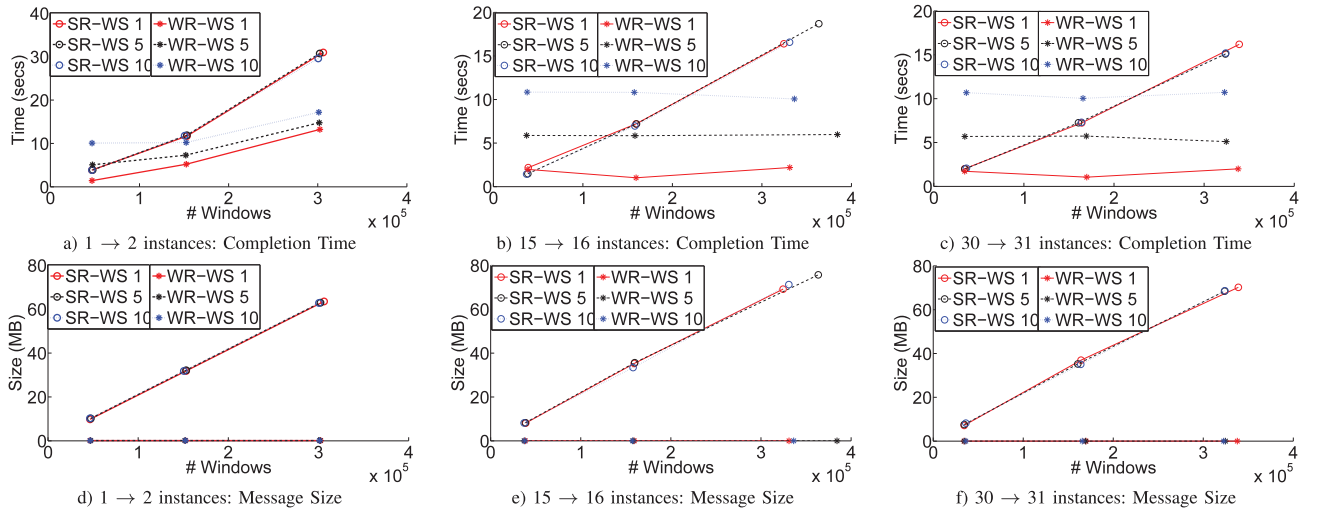Fig. 18. Join max throughput versus number of instances per node.

Fig. 19. Evaluation of the elastic reconfiguration protocols.

time for the State Recreation protocol grows linearly with the number of windows being transferred. This is because all the windows of the buckets being transferred must be sent to the new owner.

The Window Recreation protocol takes a time proportional to the window size, regardless of the number of windows to be transferred. The time to complete shown in Fig. 19a increases with a steeper slope with respect to Figs. 19b and 19c. This is because there is only one instance transferring a large number of buckets; for configurations with a higher number of instances, this effect disappears and the completion time only depends on the window size. Figs. 19d, 19e, and 19f show the amount of data transferred to the new owner in each configuration. With the SR protocol, data received by the new instance grow linearly with the number of transferred windows; using the WR protocol, no data are exchanged between instances being reconfigured.

Comparing the results of this set of experiments, we conclude that SR provides better performance as long as the completion time (dependent on the windows being transferred) does not exceed the time to fill up a window.

6.3.2 Load Balancing

The goal of this set of experiments is to evaluate the effectiveness of load balancing for input loads with a distribution that changes over time. We use the query of Fig. 2 and monitor the evolution of the stateful subquery deployed in a subcluster of 15 instances that process a constant input load of 150,000 t/s.

Input tuples have 10,000 different phone numbers, i.e., the *Group-by* parameter of the Aggregate operator has 10,000 keys. Input tuples were generated according to a normal distribution that varies over time, either its average μ or its standard deviation σ. The goal is to show that load balancing allows keeping a balanced CPU utilization rate of the allocated instances, despite the variability of the input load.

Fig. 20a compares system performance with and without load balancing. The experiment is divided in periods. During the first period, input data follow a uniform distribution. In all other periods, we use a normal distribution with μ that changes periodically from 2,000 to 8,000 in steps of 2,000. In Fig. 20a, periods are separated with vertical lines and for each of them, μ and σ are

specified. Dashed lines show the CPU average utilization rate (primary Y-axis) and its standard deviation (secondary Y-axis) when load balancing is not enabled. In this case, the standard deviation grows each time μ changes. Solid lines show the performance when load balancing is enabled. The standard deviation exhibits a peak after the beginning of each new period, which is reduced after SC balances the load. Load balancing keeps the standard deviation constant, despite the changes in the input distribution. Both configurations show a constant average utilization rate because we fixed the injected load. Fig. 20b provides results of the experiment where μ is kept constant and σ changes periodically from 100 to 25 in steps of 25. Without load balancing, the imbalance among instances increases as σ decreases. With load balancing, the load is redistributed
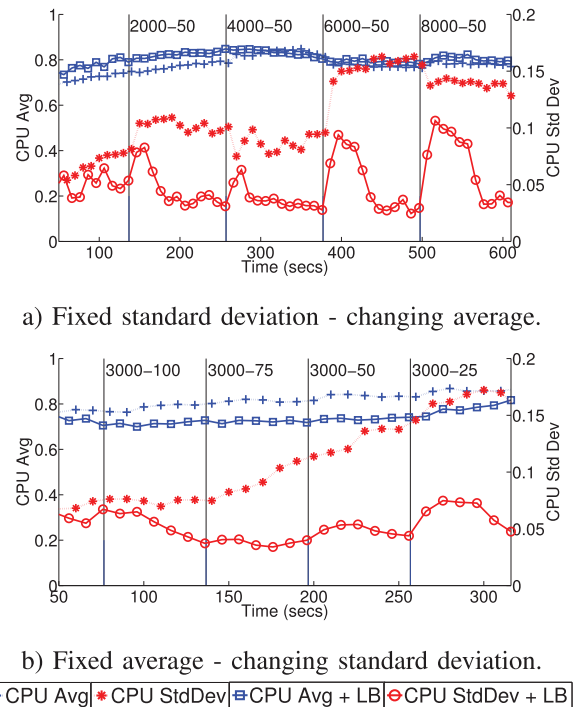

a) Fixed standard deviation - changing average.

b) Fixed average - changing standard deviation.

Fig. 20. Elastic management—load balancing.

a) Individual Provisioning

b) Load-Aware Provisioning

c) Load-Aware Decommissioning

Fig. 21. Elastic management—provisioning strategies.

TABLE 1
Static versus Elastic Configurations Overhead

Configuration	CPU usage (%)	Throughput (t/s)
Static (11 instances)	0.81	64, 506
Elastic (11 instances)	0.80	64, 645
Static (17 instances)	0.85	100, 869
Elastic (17 instances)	0.84	102, 695

the system suffers from cascade provisioning. Fig. 21b shows the effectiveness of the *SC* load-aware provisioning strategy. As the number of provisioned nodes is computed on the current subcluster size and load, each provisioning step achieves the target utilization threshold. Moreover, load-aware provisioning affords less frequent reconfiguration steps than individual provisioning. Hence, the system can reach higher throughput with fewer reconfiguration steps. In other words, load-aware provisioning is less intrusive than individual provisioning. Once reached 27 instances, we start decreasing the load and show the system behavior in Fig. 21c. Decommissioning works as effectively as provisioning. The decommissioning intrusiveness is even lower than provisioning due to the decreasing load. That is, once instances are transferring ownership of their buckets, the decreasing input rate results in a CPU average utilization slightly below the TUT.

Elastic resource management overhead does not affect the performance of the parallelization technique used in *SC*. To support our claim, we ran the same query used for the experiment of Fig. 21 for two static configurations of 11 and 17 *SC* instances, respectively, and we compared throughput and CPU consumption with the results of Fig. 21b. Size and injected data rate of each configuration were chosen upon the experiment of Fig. 21b. During this experiment, the cluster reached size 11 around time 500 (seconds) and ramped up to size 17 around time 900 (seconds). Results are provided in Table 1: for both cluster sizes, the static and the elastic configuration reach about the same throughput with similar CPU usage.

7 RELATED WORK

Distributed stream processing. Distributed SPEs (e.g., Borealis [3]) allow deployment of each operator to a different node. Their scalability is bounded by the capacity of a single node that must process the whole input data stream. *SC* overcomes this issue by never concentrating the data stream in any single node.

Parallel stream processing. Aurora* [8] and Flux [9] are two early proposals for parallel stream processing in shared nothing environments. Aurora* distributes the load across several nodes running the same operator and is mostly related to our parallelization approach. The main differences stem from the fact that Aurora* box splitting uses a single filter upstream of the parallel operator and a single union downstream. This results in the whole data stream going through a single node (the filter or the union), which bounds its scalability. In contrast, in *SC*, a data stream never needs to go through a single node. Instead, it travels from a cluster of nodes to another cluster in parallel. To support our claims, we have run the Join operator of Section 6.2.2 over two clusters of 20 and 40 nodes running Aurora*. The smaller cluster reached a throughput of less than 13,000 t/s while the

and the standard deviation is constantly kept below the upper imbalance threshold.

6.3.3 Self-Provisioning

In this set of experiments, we evaluate the effectiveness of provisioning and decommissioning instances on the fly. We use the query of Fig. 2 and monitor the evolution of the stateful subquery as the size of the subcluster changes. We set $LUT = 0.5$, $UUT = 0.9$, and $TUT = 0.6$. The load is increased (resp. decreased) linearly to observe the effectiveness of provisioning (resp. decommissioning). Fig. 21a shows the behavior of the individual provisioning strategy, i.e., when provisioning one instance at a time. We study the system behavior when growing from 1 to 15 instances. The throughput increases linearly with the input load, despite negligible variations at each provisioning step. However, the target utilization is achieved only when moving from one to two instances. Starting from size 3, each provisioning step does not provide enough additional computing power to decrease the average CPU utilization to the target one (TUT). For larger configurations (e.g., 15 nodes), provisioning of one instance results in a negligible increase of the overall computing power, leading to an average CPU utilization close to the upper threshold. As soon as the average CPU utilization stays above the upper threshold,

largest configuration reached roughly 11,000 t/s.[6] With the same configurations, *SC* has reached around 170,000 and 300,000 t/s, respectively (as shown in Fig. 16). Flux extends the exchange operator [10] to a shared nothing environment for data streaming. The exchange operator is a parallelization operator for parallel databases in multiprocessors environments. It has a role similar to our load balancer, that is, parallelizing without having to customize query operators. Both provide semantic awareness or, in Flux terminology, "content sensitive routing." One important difference between Flux and *SC* is that the exchange operator in Flux needs to be implemented for each SPE, while our load balancer is implemented via standard filters. Further, Flux's evaluation was performed using simulations and focused on a single operator. In contrast, *SC* has been evaluated in a real deployment with both individual operators and full queries. Neither Aurora* nor Flux perform any evaluation of the scalability of their approaches, that is one of the main contributions of our paper. A novel approach to data stream processing based on NFA was initially proposed in [11] and later extended to distributed processing by [12]. Cayuga [11] does not support Cartesian Product and windowed Join operators. Its successor, Johka [12], provides two combined techniques to afford distributed processing. The first technique, row/column scaling replicates a query over several machines; hence, a query-aware data partitioning technique forwards only to a subset of the machines, all events related to a given query. The second technique, called pipelining, splits a query over several nodes so that sequentially process the input data. Both techniques are similar to the semantic-aware data partitioning and the operator-set-cloud query partitioning of *SC*. However, [12] lacks details about automatic partitioning of the data and the query, suffer from the same semantic limitations of [11] (i.e., no Cartesian Product nor windowed Join operators) and does not provide elasticity. S4 [13] is a scalable distributed SPE that does not use windowing. State information in S4 are continuously maintained but stale data are periodically purged to free memory. As in *SC*, S4 uses the hash of the keys in an event to distribute them across multiple processing nodes. The system is at an early development stage and currently does not provide elasticity. One work that is mostly related to *SC* is [14]. The parallelization approach of Andrade et al. [14] is similar to the one of *SC* but is tailored for one specific type of query (refereed to "Sense-and-Respond"). Differently, we propose a generic approach to parallelize any query. Finally, Andrade et al. [14] propose a static system while *SC* provides elastic resource management. This paper is an extension of [5] where the static version of *SC* is presented. In this manuscript, we extend *SC* with elasticity and load balancing.

Load balancing. The authors of [15] study static load balancing and use correlation information of input loads to improve the operator distribution plan. They achieve fine resource utilization by allocating operators with highly correlated input streams on different nodes. Dynamic load balancing is addressed in [9] with a focus on intraoperator load distribution, but the authors do not provide a detailed evaluation of their system. In [16], overloaded operators trigger a reconfiguration of the load distribution policy with a "backpressure" message to upstream peers. However, the authors of [16] only consider stateless operators. *SC* provides load balancing for stateless and stateful operators and redistributes the load on the fly.

Elasticity. Elasticity in streaming environments has not been considered before. In the context of traditional database management systems, the solutions in [17] and [18] provide autonomic provisioning of database servers in order to keep the average query latency under an arbitrary service level agreement. However, database applications typically do not require near real-time processing and do not face a number of issues that we address in the provisioning for SPEs. Gounaris et al. [19] propose an adaptive SPE where resources are dynamically allocated. The system uses exchange operators similar to those in [9] but does not provide details on how states are managed among a variable set of instances. The evaluation lacks stateful operators and only shows query response times as the number of available resources increases.

In some settings, elasticity is closely related to fault tolerance [20] where the former adds resources as the system saturates while fault tolerance requires new computing resources as some of them fail. However, we assume fault tolerance to be beyond the scope of this paper and we plan to add fault tolerance to *SC* as future work. Both fault tolerance and elasticity require state transfer between processing units. However, we argue that the approaches are different: fault tolerance requires proactive state transfer because the state of a processing unit is lost once the unit fails; elasticity requires reactive state transfer as the saturated instances is still working and can transfer its state information to a peer.

8 CONCLUSIONS

We have presented *SC*, a highly scalable and elastic data streaming system. *SC* provides transparent parallelization that preserves the syntax and semantics of centralized queries. Scalability is attained by means of a novel parallelization strategy that minimizes the distribution overhead. Elasticity and dynamic load balancing minimize the number of resources used for coping with varying workloads. The evaluation demonstrates the large scalability and effectiveness of elasticity of *SC*.

REFERENCES

[1] M. Stonebraker, U. Çetintemel, and S.B. Zdonik, "The 8 Requirements of Real-Time Stream Processing," *SIGMOD Record,* vol. 34, no. 4, pp. 42-47, 2005.
[2] S. Chandrasekaran, O. Cooper, A. Deshpande, M.J. Franklin, J.M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M.A. Shah, "Telegraphcq: Continuous Dataflow Processing for an Uncertain World," *Proc. First Biennial Conf. Innovative Data Systems Research (CIDR),* 2003.

---

6. The smaller configuration has better performance as the single upstream Filter incurs in less fan-out overhead.

[3] 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," *Proc. Second Biennial Conf. Innovative Data Systems Research (CIDR)*, pp. 277-289, 2005.

[4] M.T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*, third ed. Springer, 2011.

[5] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, and P. Valduriez, "Streamcloud: A Large Scale Data Streaming System," *Proc. Int'l Conf. Distributed Computing Systems (ICDCS '10)*, pp. 126-137, 2010.

[6] N. Tatbul, U. Çetintemel, and S.B. Zdonik, "Staying Fit: Efficient Load Shedding Techniques for Distributed Stream Processing," *Proc. Int'l Conf. Very Large Data Bases (VLDB)*, pp. 159-170, 2007.

[7] D.J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S.B. Zdonik, "Aurora: A New Model and Architecture for Data Stream Management," *VLDB J.*, vol. 12, no. 2, pp. 120-139, 2003.

[8] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S.B. Zdonik, "Scalable Distributed Stream Processing," *Proc. Biennial Conf. Innovative Data Systems Research (CIDR)*, 2003.

[9] M.A. Shah, J.M. Hellerstein, S. Chandrasekaran, and M.J. Franklin, "Flux: An Adaptive Partitioning Operator for Continuous Query Systems," *Proc. IEEE Int'l Conf. Data Eng. (ICDE)*, pp. 25-36, 2003.

[10] G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System," *Proc. SIGMOD Conf.*, pp. 102-111, 1990.

[11] A.J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W.M. White, "Cayuga: A General Purpose Event Monitoring System," *Proc. Biennial Conf. Innovative Data Systems Research (CIDR)*, pp. 412-422, 2007.

[12] L. Brenna, J. Gehrke, M. Hong, and D. Johansen, "Distributed Event Stream Processing with Non-Deterministic Finite Automata," *Proc. ACM Int'l Conf. Distributed Event-Based Systems (DEBS)*, pp. 1-12, 2009.

[13] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed Stream Computing Platform," *Proc. IEEE Int'l Conf. Data Mining Workshops (ICDM)*, pp. 170-177, 2010.

[14] H. Andrade, B. Gedik, K.-L. Wu, and P.S. Yu, "Processing High Data Rate Streams in System S," *J. Parallel Distributed Computing*, vol. 71, no. 2, pp. 145-156, 2011.

[15] Y. Xing, S.B. Zdonik, and J.-H. Hwang, "Dynamic Load Distribution in the Borealis Stream Processor," *Proc. Int'l Conf. Data Eng. (ICDE)*, pp. 791-802, 2005.

[16] R.L. Collins and L.P. Carloni, "Flexible Filters: Load Balancing through Backpressure for Stream Programs," *Proc. Seventh ACM Int'l Conf. Embedded Software (EMSOFT)*, pp. 205-214, 2009.

[17] G. Soundararajan, C. Amza, and A. Goel, "Database Replication Policies for Dynamic Content Applications," *Proc. ACM SIGOPS/ EuroSys European Conf. Computer Systems (EuroSys)*, pp. 89-102, 2006.

[18] J. Chen, G. Soundararajan, and C. Amza, "Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers," *Proc. IEEE Int'l Conf. Autonomic Computing (ICAC)*, pp. 231-242, 2006.

[19] A. Gounaris, J. Smith, N.W. Paton, R. Sakellariou, A.A.A. Fernandes, and P. Watson, "Adaptive Workload Allocation in Query Processing in Autonomous Heterogeneous Environments," *Distributed and Parallel Databases*, vol. 25, no. 3, pp. 125-164, 2009.

[20] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker, "Fault-Tolerance in the Borealis Distributed Stream Processing System," *Proc. SIGMOD Conf.*, pp. 13-24, 2005.

Vincenzo Gulisano received the BS and MS degrees in computer science from the University of Trieste, Italy. He is currently working toward the PhD degree at the Universidad Politécnica de Madrid, Spain. His research interests include highly scalable and fault-tolerant solutions for parallel-distributed data streaming applications. He is a student member of the IEEE Computer Society.

Ricardo Jiménez-Peris is an associate professor at Universidad Politécnica de Madrid and codirector of the Distributed Systems Laboratory. He has published more than 100 papers. He has served in the Program Committee of VLDB, ICDCS, ICDE, DSN, DISC, SRDS, Middleware, etc., and he has served as general chair of SRDS, PC chair of EDCC, workshop track chair at ICDCS, and tutorial chair of LADC. He is currently involved in the MASSIF project where complex event processing is being applied to security and more concretely to Security Information and Event Management systems. He is a member of the IEEE Computer Society.

Marta Patiño-Martínez is a codirector of the Distributed Systems Laboratory and an associate professor at Universidad Politécnica de Madrid. She has published several papers on international conferences and journals. She has served in the program committee of several conferences, including VLDB, ICDCS, ICDE, Middleware, SRDS, and EDCC. She is currently coordinating the CumuloNimbo European project on ultrascalable transactional processing. He is a member of the IEEE Computer Society.

Claudio Soriente received the PhD degree in networked systems from the University of California, Irvine. He is a Juan de la Cierva fellow at the Universidad Politécnica de Madrid, Spain. His research interest include distributed systems, security and privacy. He is a member of the IEEE Computer Society.

Patrick Valduriez received the PhD degree and Doctorat d'Etat in CS from University Paris 6 in 1981 and 1985, respectively. He is a senior researcher at INRIA and LIRMM, University Montpellier 2, France. He has also been a professor of CS at University Paris 6 (1999-2002) and a researcher at Microelectronics and Computer Technology Corp. in Austin, Texas. He is the head of the Zenith team that focuses on data management in large-scale distributed and parallel systems (P2P, cluster, grid, cloud), in particular, scientific data management. He has authored and coauthored more than 200 technical papers and several textbooks, among which "*Principles of Distributed Database Systems*." He currently serves as associate editor of several journals, including the *VLDB Journal*, *Distributed and Parallel Databases*, and *Internet and Databases*. He has served as PC chair of major conferences such as SIGMOD and VLDB. He was the general chair of SIGMOD '04, EDBT '08, and VLDB '09. He was the recipient of the 1993 IBM scientific prize in CS in France. He received the best paper award at VLDB '00. http://www-sop.inria.fr/members/valduriez. He is a member of the IEEE Computer Society.