StreamCloud: An Elastic and Scalable Data Streaming System

Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martínez, Claudio Soriente, Patrick Valduriez

StreamCloud: An Elastic and Scalable Data Streaming System

Vincenzo Gulisano*, Ricardo Jiménez-Peris*, Marta Patiño-Martínez*, Claudio Soriente*, Patrick Valduriez†
*Universidad Politécnica de Madrid, Spain, {*vgulisano,rjimenez,mpatino,csoriente*}*@fi.upm.es*
†INRIA and LIRMM, Montpellier, France, *patrick.valduriez@inria.fr*

***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 over-provisioning.**

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.**

I. 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 (inter-query parallelism) or even distributing different operators of a query across different nodes (inter-operator 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 intra-operator 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., non-parallel) 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 non-parallel 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 over-provisioning (i.e., the allocated nodes are running below their full capacity). Under-provisioning results in the violation of service level agreements that can incur high economic costs. Even with best effort agreements, under-provisioning is responsible for losing unhappy users and raising bad reputation from disgruntled users. Over-provisioning 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

This research has been partially funded by the Madrid Regional Council (CAM), FSE and FEDER under project CLOUDS (S2009TIC-1692), the Spanish Research Agency MICINN under project CloudStorm (TIN2010-19077) and the Juan de la Cierva Fellowship of Claudio Soriente, and the European Commission under project MASSIF (FP7-257475). Ricardo Jiménez-Peris and Marta Patiño-Martínez filed a patent at USPTO with number US13/112,628 related to this paper.

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 intra-operator 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 re-organizing 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 II provides a brief introduction to stream processing. Section III discusses various parallelization strategies while Section IV shows how *SC* parallelizes queries to attain high scalability while preserving both syntactic and semantic transparency. Section V presents elastic resource management and load balancing algorithms used in *SC*. Section VI provides a thorough evaluation of the scalability and elasticity of *SC* while Section VII surveys relevant work in the area. The paper concludes in Section VIII. Additional sections have been added in a supplementary document which is organized as follows. Section IX provides basic background on Data Stream Processing while Section X details the setup used for the evaluation as well as the queries and the data set. Finally, Section XI provides a cost model for the query parallelization strategies considered in this manuscript.

II. 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 timestamp 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 timestamped 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) 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 IX of the supplementary document.

III. Parallel Data Streaming

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

A. 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 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

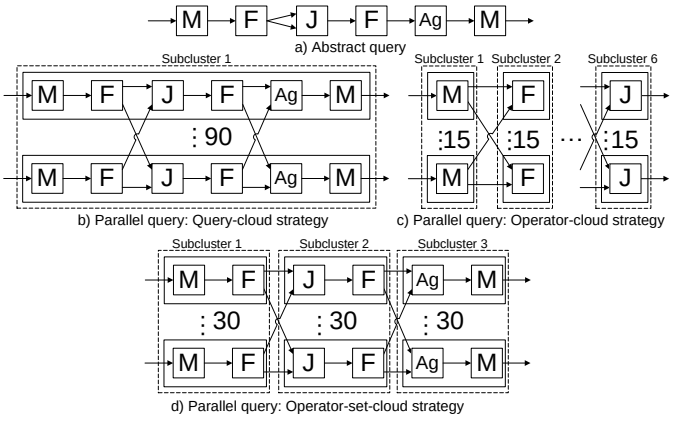


Fig. 1. Query Parallelization Strategies.

of each node with the others. We illustrate three alternative parallelization strategies by means of the abstract query in Fig. 1.a 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. 1.b). 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. 1.c). 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. 1.d). The above distribution strategies exhibit a trade-off 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. 1.a 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).

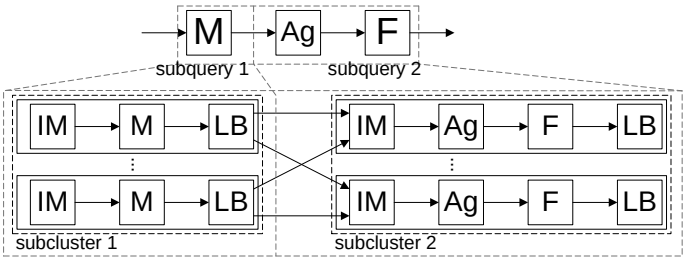


Fig. 2. Query Parallelization in *SC*.

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 XI of the supplementary document.

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 flows 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.

IV. 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 and Input Mergers.

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 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

```
Load Balancer for Join & Aggregate
Upon: arrival of t:
 1: forward(t, nextSubcluster[BIM[hash(t.A1, t.A2, ...) % 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.A1, t.A2, ...) % 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: ∃ dest | currentTime() ≥ 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 ∀i, buffer[i].nonEmpty() then
11:     t_o = earliestTuple(buffer)
12:     if ¬ isDummy(t_o) then
13:         forward(t_o)
```

Fig. 3. Pseudocode for Load Balancers and Input Mergers.

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 $t = (A_1, A_2, ..., A_n)$, its corresponding bucket b is computed by hashing one or more of its attributes[1] modulus B, e.g., $b = H(A_i, A_j) \mod B$. 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 (LB)*. 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 (IM)*, 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.

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

a) 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, $BIM[H(t.A_i) \mod B].dest$ 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.

b) General Join operator (Cartesian Product): The Cartesian Product (CP) 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 $b_l = H(A_i, A_j) \mod B_l$. Similarly, a tuple t' entering the upstream right LB, is forwarded to all instances $BIM[b_l, b_r].dest$ having $b_r = H(A_i, A_j) \mod B_r$. 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).

Figure 4.a 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 has predicate $left.number \mod 2 = 1 \wedge right.letter \neq A$ and a temporal window of 2 seconds. Figure 4.a also shows a sample input and the resulting output. Tuple timestamps are indicated on the top of each stream (the values right of the "ts" tag). Figure 4.b 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 2 nodes and the CP operator in a subcluster of 4 nodes ($N = 4$). Left and right incoming streams of the CP operator are 0-3 and A-D, respectively.

Both streams are split into 2 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 4 CP instances performs one fourth of the whole Cartesian Product

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

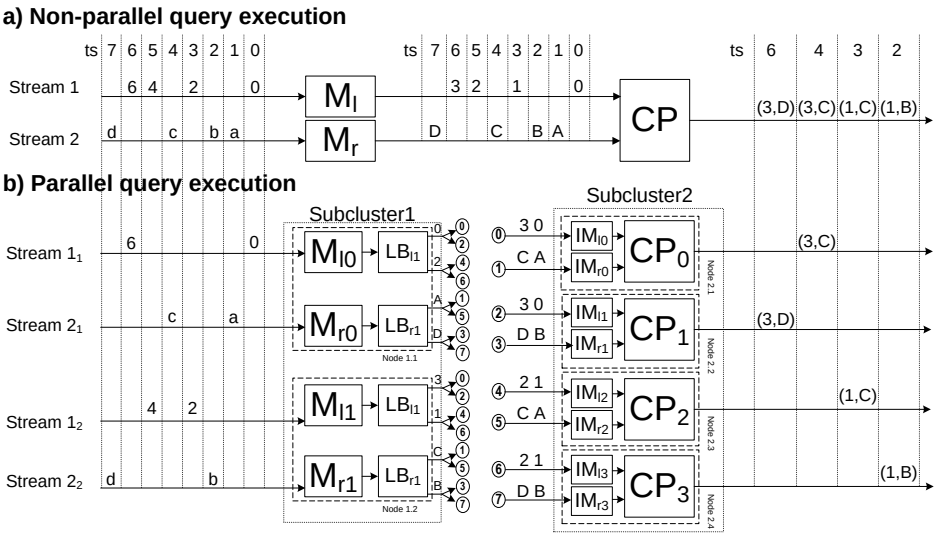


Fig. 4. Cartesian Product Sample Execution.

on the incoming streams.

c) 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 is 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.

B. Input Mergers

Just like LBs, Input Mergers (IMs) 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. 4.a, the non-parallel 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 non-parallel query is depicted in Fig. 5.a (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. 5.a); tuple "A" is purged on the arrival of tuple "2", and so on.

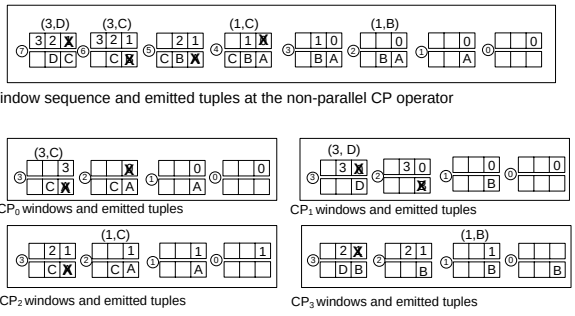


Fig. 5. Cartesian Product Sample Execution - Operator Window Evolution.

[2]Window update is independent of the interleaving on the input streams and only depends on the timestamps of the incoming tuples.

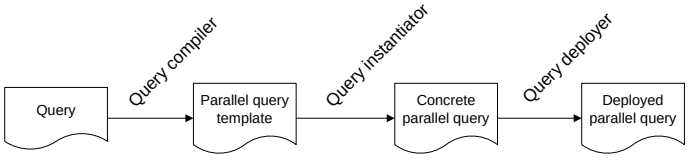


Fig. 6. Query Compiler.

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 timestamp ordered input streams coming from upstream LBs and feeds the local subquery with a single timestamp ordered input stream. As a result, the local subquery will produce a timestamp ordered output stream. To guarantee a correct sorting, it is sufficient that the IM forwards the tuple with the smallest timestamp, any time it has received at least one tuple from each input stream (lines 10-13). To 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. 5.b, 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 non-parallel operator.

C. Query Compiler

SC builds on top of Borealis [3], a non-parallel SPE. Input queries are written in the Borealis query language and automatically parallelized by *SC* through its query compiler.

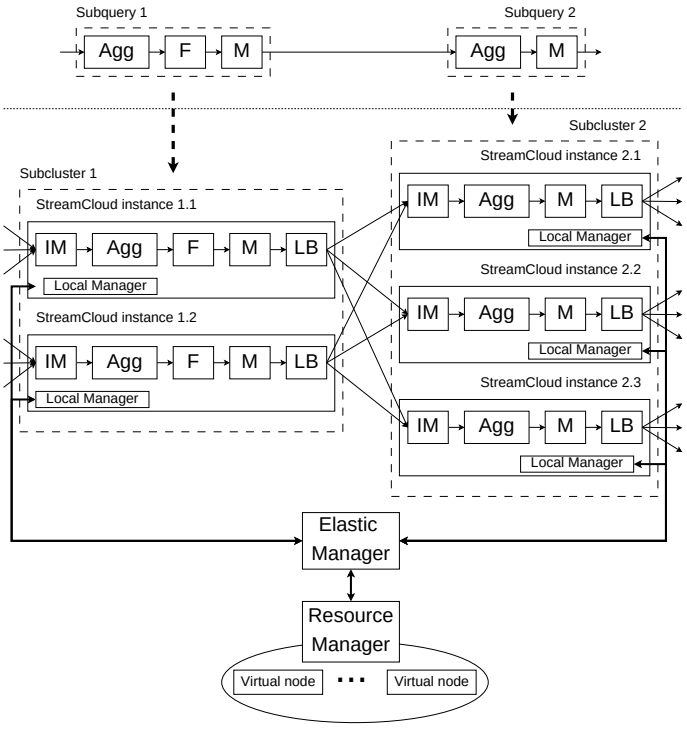


Fig. 7. Elastic management architecture.

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 V). The process is shown in Fig. 6. The query compiler takes a regular query (e.g., the one in Fig. 1.a) and splits it into subqueries according to the Operator-set-cloud parallelization strategy (e.g., Fig. 1.d). 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.

V. Elasticity

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

Figure 7 illustrates a sample configuration with elastic management components. *SC*'s 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% while its memory footprint is around 20MB (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 10ms 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 non-intrusive manner.

A. 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 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 3 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.

1) Reconfiguration Start: Figure 8 shows the pseudocode common to both reconfiguration protocols. The process is

[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.

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 timestamp to start the reconfiguration ($startTS$). Each LB proposes a timestamp based on the last tuple of bucket b that has been forwarded. The highest timestamp 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.

Figure 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 timestamp 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 timestamp received (Fig. 8, line 7 and line 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

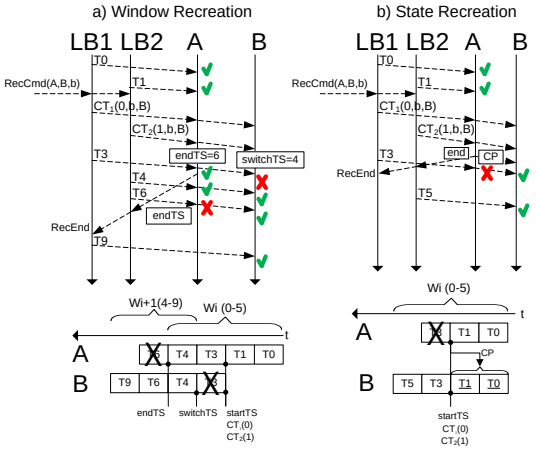


Fig. 9. Sample reconfigurations.

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.

register their roles as old and new owner, respectively, setting the variable $BT[b].ownership$ (Fig. 8, line 8 and line 11).

2) Window Recreation 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 timestamp 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 timestamp of

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.

the last windows it has to process, i.e., $BT[b].endTS$[4]. Similarly, B computes the timestamp of the first window it has to process, i.e., $BT[b].switchTS$. $BT[b].endTS$ 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 timestamp later than $BT[b].endTS$ (Fig. 10, lines 1-5).

Figure 9.a 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 timestamp of the latest tuple forwarded, via a control tuple: CT_1 carries timestamp 0 and CT_2 carries timestamp 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 timestamp is lower than $BT[b].startTS$. B becomes responsible for all windows starting from W_{i+1} since its starting timestamp is greater than $BT[b].startTS$. Tuples $T3$ to $T4$ are sent from LBs to both instances because their timestamp is earlier than $BIM[b].endTS$. Tuple $T6$ should be sent only to B (its timestamp 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 timestamp is earlier than $BT[b].switchTS$. Tuples $T6$ is discarded by A because its timestamp is equal to $BT[b].endTS$. Starting from tuple $T9$, LBs only forward tuples to B.

3) State Recreation 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 timestamp later than or equal to $BT[b].startTS$ are discarded by A. At this time, A also serializes the state associated to 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 timestamp earlier than $BIM[b].endTS$. Tuples with a timestamp 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).

Figure 9.b 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 timestamp 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. 9.b). Tuple $T5$ is only sent to B since it is processed by LB2 after the reconfiguration has ended.

B. 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 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

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

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.

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 under-provisioned, new instances are allocated (Fig. 12, lines 5-6). If the subcluster is over-provisioned, 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).

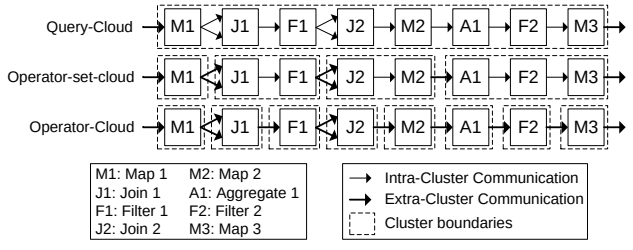


Fig. 13. Query used for the evaluation.

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

[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.

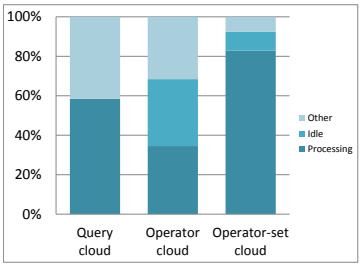


Fig. 14. Parallelization strategies evaluation - CPU usage breakdown.

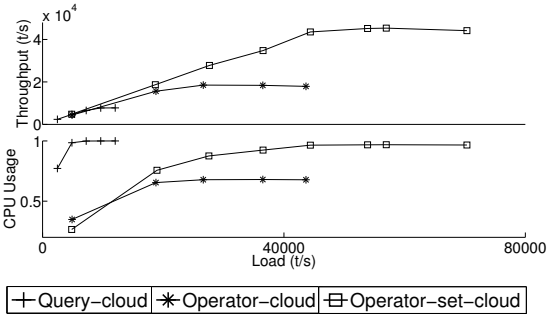


Fig. 15. Parallelization strategies evaluation - Scalability.

VI. Evaluation

A. 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 X of the supplementary document. All the experiments have 3 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.

B. 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 III-A. 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 multi core architectures.

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 III-A. 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). Figure 14 shows that the overall distribution overhead is around 40%. The remaining 60% is used for tuple processing.

The operator-cloud strategy shows a distribution overhead of more than 30%, while CPU ratio used to process tuples is roughly 35%. The unused CPU cycles (tagged as "idle" in Fig. 14) are related to the difference between the nominal subcluster sizes and the actual ones. For instance, when computing the optimal distribution plan, an operator might

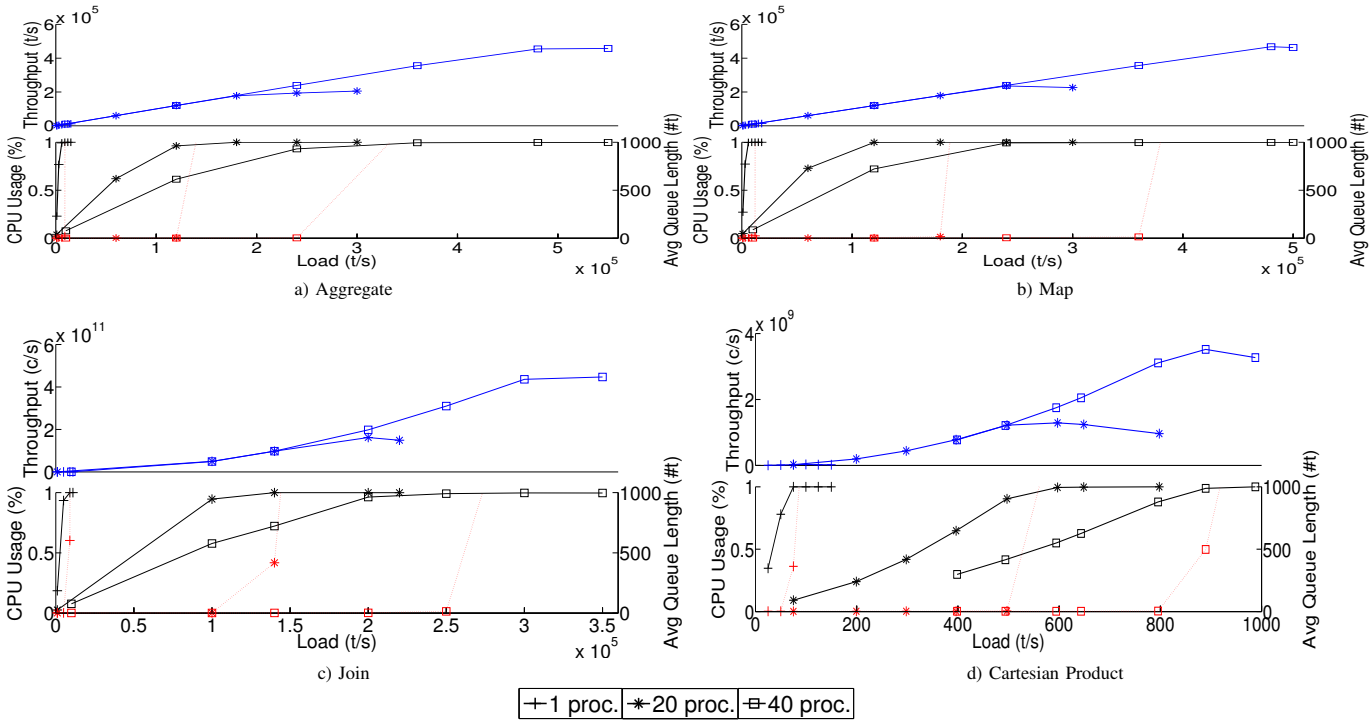


Fig. 16. Individual Operator and Query Scalability Evaluation Results. Solid lines in the upper part of each figure refer to throughput (t/s). In the bottom part of each figure, solid lines refer to CPU usage (%) while dotted lines refer to queue lengths (#t).

require 4.3 instances that translates to an actual assignment of 5 instances, thus leading to one CPU that is not fully utilized.

The operator-set-cloud approach exhibits the lowest communication overhead (roughly 10%). 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% 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%; roughly 40% of the CPU is not used.

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 timestamp.

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% 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% CPU utilization where queues reach their limits and throughput becomes stable. Each stage can be clearly seen in the bottom parts of Fig. 16.a through Fig. 16.d, where solid lines show the CPU usage (left Y axis) and dotted lines show queue lengths (right Y axis).

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. 16.a). Twenty instances manage an input rate of roughly 200,000 t/s while forty 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. 16.b shows a linear throughput where twenty 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. 16.c 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

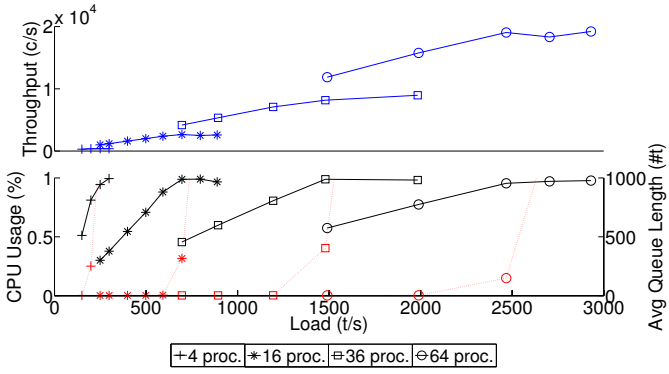


Fig. 17. CP over fixed-size windows

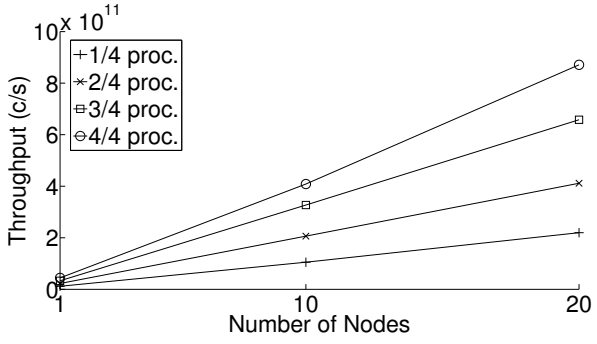


Fig. 18. Join max throughput vs. number of instances per node.

in the upper part of Fig. 16.d. 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.

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 VI-B.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 non-parallel 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.

Figure 17 shows super-linear 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.

4) Multi-Cores: 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 VI-B.2 deployed over 1, 10 and 20 quad-core nodes, respectively. On each node, up to 4 *SC* instances were deployed.

Figure 18 shows linear scalability with respect to the number of *SC* instances per node. For example, 1 node running 4 *SC* instances handles 0.4×10^{11} c/s, 10 nodes running 4 *SC* instances each can reach 4×10^{11} c/s and 20 nodes with 4 *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.

C. Elasticity Evaluation

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

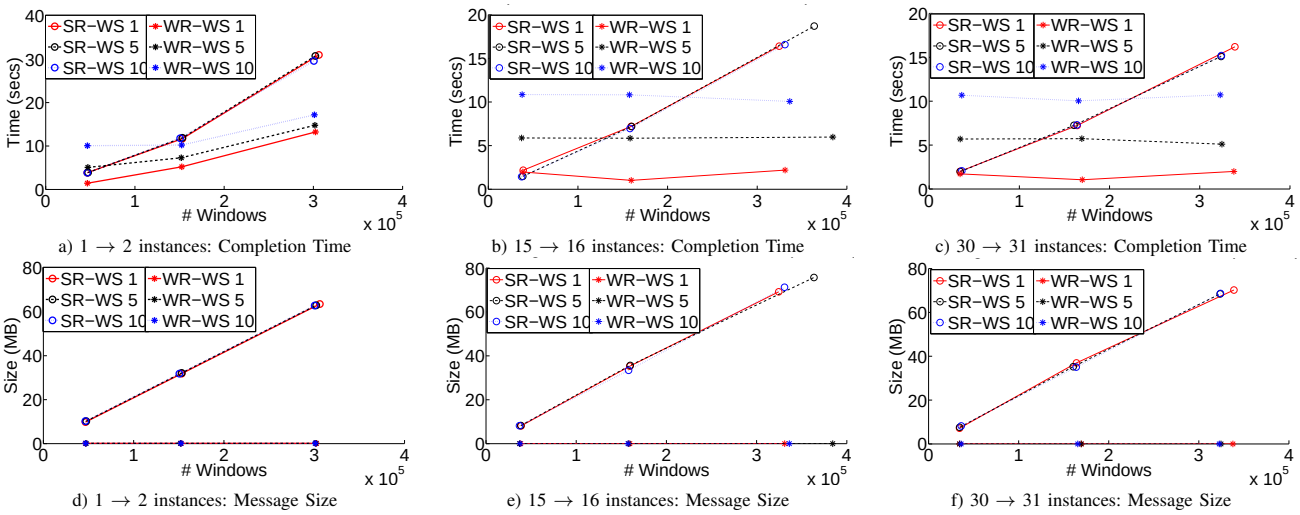
1) Elastic Reconfiguration Protocols: This set of experiments aims at evaluating the trade-off between the elastic reconfiguration protocols of Section V. We run the stateful subquery of Fig. 2, with window size of 1, 5 and 10 seconds (WS labels in Fig. 19), respectively. UUT is set to 80%, that is, when the average CPU utilization reaches 80%, 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. Figures 19.a, 19.b, and 19.c show the time required to complete a reconfiguration from 1 to 2 instances, from 15 to 16 instances and from 30 to 31 instances, respectively, with an increasing number of transferred windows. Completion time for the State Recreation (SR) 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 (WR) 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. 19.a increases with a steeper slope with respect to Fig. 19.b and Fig. 19.c. 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. Figures 19.d, 19.e, and 19.f show the amount of data transferred to the new owner in each configuration. With the SR protocol, data received by the new instance grows linearly with the number of transferred windows; using the WR protocol no data is 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.

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



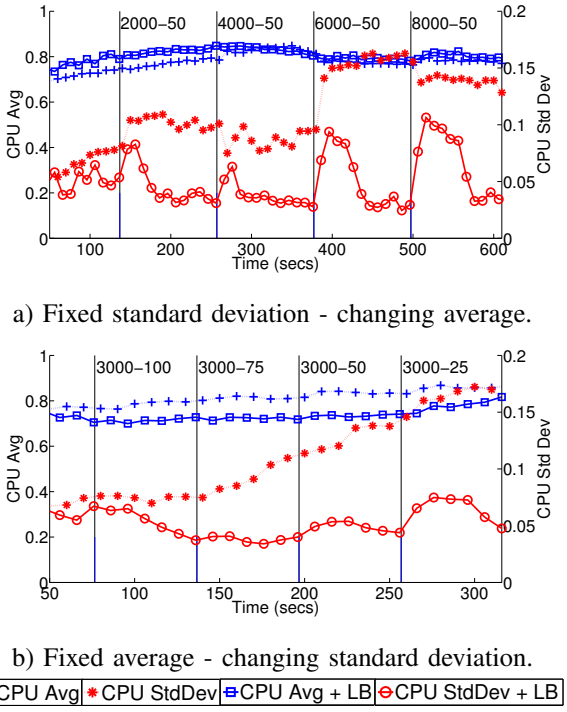
a) 1 → 2 instances: Completion Time
b) 15 → 16 instances: Completion Time
c) 30 → 31 instances: Completion Time
d) 1 → 2 instances: Message Size
e) 15 → 16 instances: Message Size
f) 30 → 31 instances: Message Size

Fig. 19. Evaluation of the elastic reconfiguration protocols.

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.

Figure 20.a compares system performance with and without load balancing. The experiment is divided in periods. During the first period, input data follows 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. 20.a, 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. Figure 20.b 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 and the standard deviation is constantly kept below the upper imbalance threshold.

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). Figure 21.a 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 1 to 2



a) Fixed standard deviation - changing average.

b) Fixed average - changing standard deviation.

Fig. 20. Elastic Management - Load Balancing.

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, the system suffers from cascade provisioning. Figure 21.b, 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

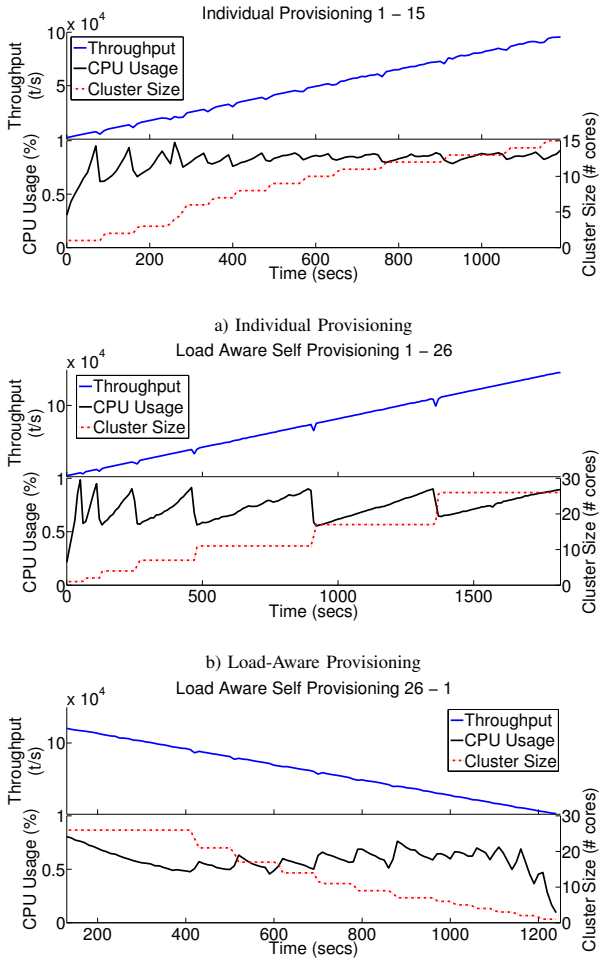


Fig. 21. Elastic Management - Provisioning Strategies.

TABLE I

STATIC VS. 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

is less intrusive than individual provisioning. Once reached 27 instances, we start decreasing the load and show the system behavior in Fig. 21.c. 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. 21.b. Size and injected data rate of each configuration were chosen upon the experiment of Fig. 21.b. 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 I: for both cluster sizes, the static and the elastic configuration reach about the same throughput with similar CPU usage.

VII. 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 VI-B.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 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 multi-processors 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

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

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 is 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 [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, [14] proposes 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 intra-operator 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. [19] proposes 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.

VIII. 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," in *CIDR*, 2003.

[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," in *CIDR*, 2005, pp. 277–289.

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

[5] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, and P. Valduriez, "Streamcloud: A large scale data streaming system," in *International Conference on Distributed Computing Systems (ICDCS'10)*, 2010, pp. 126–137.

[6] N. Tatbul, U. Çetintemel, and S. B. Zdonik, "Staying fit: Efficient load shedding techniques for distributed stream processing," in *International Conference on Very Large Data Bases (VLDB)*, 2007, pp. 159–170.

[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," in *CIDR*, 2003.

[9] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin, "Flux: An adaptive partitioning operator for continuous query systems," in *ICDE*, 2003, pp. 25–36.

[10] G. Graefe, "Encapsulation of parallelism in the volcano query processing system," in *SIGMOD Conference*, 1990, pp. 102–111.

[11] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White, "Cayuga: A general purpose event monitoring system," in *CIDR*, 2007, pp. 412–422.

[12] L. Brenna, J. Gehrke, M. Hong, and D. Johansen, "Distributed event stream processing with non-deterministic finite automata," in *ACM International Conference on Distributed Event-Based Systems (DEBS)*, 2009, pp. 1–12.

[13] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *IEEE International Conference on Data Mining Workshops (ICDM Workshops)*, 2010, pp. 170–177.

[14] H. Andrade, B. Gedik, K.-L. Wu, and P. S. Yu, "Processing high data rate streams in system s," *J. Parallel Distrib. Comput.*, 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," in *ICDE*, 2005, pp. 791–802.

[16] R. L. Collins and L. P. Carloni, "Flexible filters: load balancing through backpressure for stream programs," in *EMSOFT*, 2009, pp. 205–214.

[17] G. Soundararajan, C. Amza, and A. Goel, "Database replication policies for dynamic content applications," in *EuroSys*, 2006, pp. 89–102.

[18] J. Chen, G. Soundararajan, and C. Amza, "Autonomic provisioning of backend databases in dynamic content web servers," in *ICAC*, 2006, pp. 231–242.

[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," in *SIGMOD Conference*, 2005, pp. 13–24.

Supplementary Document *StreamCloud*: An Elastic and Scalable Data Streaming System

Vincenzo Gulisano*, Ricardo Jiménez-Peris*, Marta Patiño-Martínez*, Claudio Soriente*, Patrick Valduriez†
*Universidad Politécnica de Madrid, Spain, {*vgulisano,rjimenez,mpatino,csoriente*}*@fi.upm.es*
†INRIA and LIRMM, Montpellier, France, *patrick.valduriez@inria.fr*

***Abstract*— This manuscript includes the supplementary sections for the paper titled "*StreamCloud*: An Elastic and Scalable Data Streaming System". It provides more details on data stream processing and the testbed used for performance evaluation of *StreamCloud*.**

IX. A primer on Data Stream Processing

In contrast with the traditional store-then-process paradigm, Stream Processing Engine (SPE) defines a new computing paradigm where data is processed on the fly. This is particularly suited in many application scenarios where both the input rates and the tight processing time requirements call for fast data analysis and prompt result reporting.

SPEs handle *continuous queries* over streaming data. A continuous query differs from its traditional database counterpart as it is always "standing" over the streaming data. That is, once the query has been deployed, results are produced each time the input data satisfies the query predicate.

A data stream S is a potentially infinite sequence of tuples. All tuples of a stream share the same schema. The latter defines name and type of each attribute of a tuple; common data types are $string$, $integer$, $double$, $time$, $etc.$. We denote the schema of a generic stream as $(A_1, A_2, \ldots, A_n)$ and refer to attribute A_i of tuple t as $t.A_i$. We also assume that data sources and system nodes are equipped with well-synchronized clocks, using a clock synchronization protocol (e.g., NTP [1]) as already done in [2]. When clock synchronization is not feasible at data sources, tuples can be timestamped at the entry point of the data streaming system. We refer to the timestamp of tuple t as $t.ts$.

Continuous queries are defined over one or more input data streams and can have multiple output streams. A continuous query is modeled as a directed acyclic graph, with additional input and output edges representing input and output streams, respectively. Each node u in the graph is an operator, which consumes tuples of at least one input stream and produces tuples for at least one output stream. The presence of an edge (u, v) means that the output stream of node u is the input stream of node v, i.e., node v consumes tuples produced by node u.

Typical query operators of SPEs are similar to relational algebra operators. They are classified depending on whether they keep state information across input tuples. Stateless operators (e.g., Map, Union and Filter) do not keep any state across tuples and perform one-by-one computation; that is, each incoming tuple is processed and an output tuple is produced, if any. Stateful operators (e.g., Aggregate, Join and Cartesian Product) perform computation over multiple input tuples; that is, each incoming tuple updates the state information of the operator and contributes to the output tuple, if any. Because of the infinite nature of data streams, stateful operators keep state information only for the most recent incoming tuples. This technique is referred to as *windowing*. Windows can be defined over a period of time (e.g., tuples received in the last hour) or over the number of received tuples (e.g., last 100 tuples).

The rest of this Section provides details on the main SPE operators.

A. Map

The Map operator is a generalized projection operator defined as

$$M\{A'_1 \leftarrow f_1(t_{in}), \ldots, A'_n \leftarrow f_n(t_{in})\}(I, O)$$

where I and O denote the input and output stream, respectively. t_{in} is a generic input tuple and $\{A'_1, \ldots, A'_n\}$ is the schema of the output stream. The operator transforms each input tuple via the set of user-defined functions $\{f_1, \ldots, f_n\}$. The output stream schema might differ from the input one, but the output tuple preserves the timestamp of the input one.

B. Filter

The Filter operator is used either to discard tuples or to route them over different output streams. It is defined as

$$F\{P_1, \ldots, P_m\}(I, O_1, \ldots, O_m[, O_{m+1}])$$

where I is the input stream, $O_1 \ldots, O_m, O_{m+1}$ is an ordered set of output streams and $P_1, \ldots, P_m$ is an ordered set of predicates.

The number of predicates equals the number of output streams and each input tuple is forwarded over the output stream associated to the first predicate that the tuple satisfies. That is, t_{in} is forwarded over O_j where $j = \min_{1 \leq i \leq m}\{i \mid P_i(t_{in}) = TRUE\}$.

Tuples that satisfy none of the predicates are output on stream O_{m+1}, or discarded if output $m + 1$ has not been defined. Output tuples are identical to input tuples.

C. Union

The Union operator merges multiple input streams sharing the same schema into a single output stream. Input tuples are propagated over the output stream in FIFO order. The Union operator is defined as

$$U\{\}(I_1, \ldots, I_n, O)$$

where $I_1, \ldots, I_n$ is a set of input streams and O is the only output stream; all streams share the same schema.

D. Aggregate

The Aggregate operator computes aggregate functions (e.g., average, count, etc.) over windows of tuples. It is defined as

$$\begin{aligned} Ag\{&Wtype, Size, Advance, A'_1 \leftarrow f_1(W), \ldots \\ &\ldots, A'_n \leftarrow f_n(W), \\ &[Group-by = (A_{i_1}, \ldots, A_{i_m})]\}(I, O) \end{aligned}$$

Tuples over input stream I are stored in the current window W until it becomes full. $Wtype$ specifies the window type that can be either time-based ($Wtype = time$) or tuple-based ($Wtype = numTuples$). If the window is time-based, it is considered full if the time distance between the incoming tuple and the earliest tuple in the window exceeds the window $Size$. In case of Tuple-based windows, a window is full if it contains $Size$ tuples.

Once a window is full, an output tuple is produced. Output tuples are propagated over stream O and have timestamp equal to the timestamp of the earliest tuple in the current window. The output tuple schema is $\{A'_1, \ldots, A'_n\}$ and $\{f_1, \ldots, f_n\}$ is the set of user-defined functions (e.g., sum, count, average, etc.) computed over all tuples in the window.

After an output tuple has been propagated, the window is updated (or "slid" forward) and stale tuples are discarded according to parameter $Advance$. If $Wtype = time$ and t_{in} is the current input tuple, a tuple t in the current window is discarded if $t_{in}.ts - t.ts > Size$. If $Wtype = numTuples$, the earliest $Advance$ tuples are discarded from the current window.

Finally, the parameter $Group-by$ is optional and is used to define equivalence classes over the input stream. In particular, assume $Group - by = A_i$, where A_i is an attribute of the input schema. Then, the Aggregate operator handles separate windows for each possible value of A_i.

E. Join and Cartesian Product

Join and Cartesian Product operators are used to correlate tuples from multiple streams. They only differ in the complexity of their predicate: Join operator requires a predicate with at least one equality statement (e.g., $t.A_i = t'.A_j$), while the predicate of the Cartesian Product can be arbitrarily complex. The Join operator is defined as

$$J\{P, Wtype, Size\}(S_l, S_r, O)$$

while the definition of the Cartesian Product is

$$CP\{P, Wtype, Size\}(S_l, S_r, O)$$

S_l, S_r are two input streams referred to as *left* and *right*, respectively, while O denotes the output stream. P is a predicate over pairs of tuples (one from each input stream) while $Wtype$ and $Size$ are windows parameters similar to the ones in the Aggregate operator.

Both operators keep two separate windows, W_l, W_r, for each input stream. Tuples arriving on the left (resp. right) stream are stored in the left (resp. right) window and used to update (i.e., slide forward) the right (resp. left) window. If $Wtype = time$, upon arrival of tuple $t_{in} \in S_l$, window W_r is updated by removing all tuples t such that $t_{in}.ts - t.ts \geq Size$. If $Wtype = NumTuples$, upon arrival of tuple $t_{in} \in S_l$, window W_r, if full, is updated by removing the earliest tuple.

After window update, for each $t \in W_r$, the concatenation of tuples t_{in} and t is produced as a single output tuple if $P(t_{in}, t) = TRUE$.

Window update, predicate evaluation and output propagation for input tuples over the right stream are performed in a similar fashion.

X. Evaluation testbed and data set

This section provides details on the evaluation testbed, the data set and the semantics of the queries using for the evaluation of *StreamCloud*.

All experiments were run in a shared-nothing cluster of 100 nodes (blades) with 320 cores. All blades are Supermicro SYS-5015M-MF+ equipped with 8GB of RAM and 1Gbit Ethernet and a directly attached 0.5TB hard disk. Blades are distributed into 4 racks: Rack 1 has 20 blades, with a dual-core Intel PentiumD@2.8GHz. Rack 2 has 20 blades, with a dual-core Intel Xeon 3040@1.86GHz. Rack 3 and rack 4 have 30 blades, each with a quad-core Intel Xeon X3220@2.40GHz. During the experiments, roughly half of the available nodes was devoted to load injection in order to reach input rates that would saturate the remaining nodes.

Our evaluation focused on mobile telephony application scenarios where activities like customer billing and fraud detection, require processing massive amount of calls per seconds. Each call generates a tuple referred to as Call Description Record (CDR), that contains information about the parties involved in the call. The schema of the CDRs used for the evaluation is shown in Table II. Each CDR carries the number of the caller and the callee, the start and end time of the call, localization information of the caller and a timestamp related to the moment when the CDR was emitted.

A. Scalability

THe evaluation of *StreamCloud* scalability focused on the scalability of individual operators. Particular emphasis was given to the scalability of stateful operators that are more challenging to distribute.

In particular, the experiments evaluated the scalability of the following operators: Aggregate, Map, Join and Cartesian Product. The semantics of each of them is as follows:

TABLE II
CALL DESCRIPTION RECORD SCHEMA USED FOR EVALUATION.

Name	Type	Description
Src	string	Caller's number
Dst	string	Callee's number
$Start$	time	Start time of the call
End	time	End time of the call
$District$	integer	Area-Id where caller is located
Lat	double	Latitude coordinate of the caller
Lon	double	Longitude coordinate of the caller
Ts	time	Emission timestamp of the CDR

- **Aggregate.** Computes the number of calls and average duration grouped by $District$. Aggregates are computed over windows of 60 seconds with advance of 10 seconds.
- **Map.** Computes the call duration time, given the call start and end time, i.e., $Duration = End - Start$.
- **Join.** Computes the distance in time and space of two calls originated at the same number within an interval of 60 seconds. That is, for each two CDRs, say CDR_i and CDR_j, if $CDR_i.Src = CDR_j.Src \wedge |CDR_i.Start - CDR_j.Start| \leq 60$, the operator computes the orthodromic distance on the earth surface between $CDR_i.Lat, CDR_i.Lon$ and $CDR_j.Lat, CDR_j.Lon$.
- **Cartesian Product.** Checks whether two calls that have the same number (either as source or destination) overlap in time, i.e., whether $CDR_i.Start < CDR_j.Start < CDR_i.End \vee CDR_j.Start < CDR_i.Start < CDR_j.End$.

Experiments titled **Scaling with Fixed-Size Windows** and **Multi-Cores** were carried out using the above Cartesian Product and Join operator, respectively.

B. Elasticity

This set of experiments evaluates the elastic capabilities of *StreamCloud*. Evaluation focused on the query of Fig. 22 that is used to detect heavy customers in a mobile telephony application. Input tuples are CDRs with the schema given in Table II. CDRs are fed to the Map operator (M) that computes the call duration time given the start and end time of the CDR. The following Aggregate operator (Ag) groups CDR on the Src attribute (i.e., $Group-by = Src$) and computes the average call duration time on an hourly-basis, with granularity of one minute (i.e., $Size = 3,600$ and $Advance = 60$). Finally, the Filter operator (F) only propagates tuples that have an average call duration above an arbitrary threshold.

According to the parallelization strategy of *StreamCloud* the query is split into two subqueries. The first one only contains the Map operator (i.e., the stateless prefix of the query), while the second (stateful) subquery contains the Aggregate and the Filter operators.

Unless stated otherwise, in all experiments regarding elasticity we were monitoring the behavior of the subcluster where the stateful subquery was deployed (i.e., the subcluster with the Aggregate and the Filter operators).

XI. COST MODEL OF PARALLELIZATION STRATEGIES

In this section we provide a cost model for the three parallelization strategies taken into account during the design of *StreamCloud*. The three strategies taken into account are *Query-cloud* (QC), *Operator-cloud strategy* (OC) and *Operator-set-cloud* (OS).

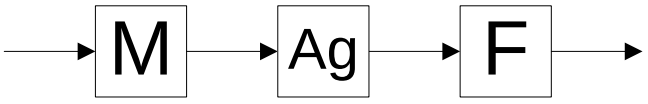


Fig. 22. Query used for the Elastic Reconfiguration Protocols evaluation.

Cost analysis of a parallelization strategy must take into account (1) the fan-out, i.e., the system-wide cost related to establishing and keeping communication channels between instances and (2) the per-tuple overhead (for serialization/deserialization) that is related to the number of hops the tuple must traverse.

Our analysis focuses on a cluster of N nodes where a query with l operators (of which $s \geq 1$ are stateful) is deployed; the query is receiving input tuples at rate ρ. Without loss of generality, we assume that the query has a stateless prefix and that all subclusters (if any) have the same size[1].

For any parallelization strategy X we define the cost function $c(X) = \alpha \cdot f(X) + \beta \cdot \rho \cdot h(X)$ where $f(X)$ is the fan-out overhead, $h(X)$ is the number of hops overhead, and $\alpha, \beta \in [0, 1]$ are two arbitrary weights.

The fan-out overhead for the QC strategy is quadratic in the number of instances, since each node must keep a communication channel towards all other instances. The overhead related to the number of hops is given by the number of stateful operators; this is because, to guarantee semantic transparency, a tuple might be redirected to a different instance right before each stateful operator.

The OC strategy deploys one operator per subcluster and each instance of a subcluster must keep a communication channel with all the instances of the downstream subcluster: hence, $f(OC) = \frac{N}{l} \cdot (l-1)$. The number of hops per tuple is $l-1$ as each node runs only one operator.

The OS strategy uses a number of subclusters that is proportional to the number of stateful operators and each instance must keep communication channels with all instances in the downstream subcluster. Since the OS also assigns a subcluster for the stateless prefix of the query, we have $f(OS) = \frac{N}{s+1} \cdot s$. In this case, there will be $s+1$ subclusters so the number of hops of a tuple is s.

From the above discussion, we can summarize the cost of each parallelization strategy as:

$$
\begin{aligned}
c(QC) &= \alpha \cdot N \cdot (N-1) + \beta \cdot \rho \cdot s \simeq \alpha \cdot N^2 + \beta \cdot \rho \cdot s \\
c(OC) &= \alpha \cdot \frac{N}{l} \cdot (l-1) + \beta \cdot \rho \cdot (l-1) \simeq \alpha \cdot N + \beta \cdot \rho \cdot l \\
c(OS) &= \alpha \cdot \frac{N}{s+1} \cdot s + \beta \cdot \rho \cdot s \simeq \alpha \cdot N + \beta \cdot \rho \cdot s
\end{aligned}
$$

We claim that, for realistic parameters of α, β, ρ, we have $c(QC) > c(OP) > c(OS)$, that is, OS is the least expensive parallelization strategy. Clearly, QC is the worst strategy as the quadratic-term increases the value of its cost function.

[1]The analysis can be easily extended to the case where the query has no stateless prefix and/or subclusters have different sizes.

Further, $c(OC) > c(OS)$ since $l > s$ and, for the considered application scenarios, ρ should at least be in the order of 10^5.

Finally note that, there is a natural trade-off between $f(X)$ and $h(X)$. On one side, $h(X)$ lies within $\rho \cdot s$ and $\rho \cdot (l-1)$. In fact, semantic transparency requires to re-distribute tuples before each stateful operator ($h(X) \geq \rho \cdot s$); at the other extreme, we can choose to re-distribute tuples before each (either stateful or stateless) operator as in the OC strategy ($h(X) \leq \rho \cdot (l-1)$).

On the other side, minimizing $f(X)$ requires picking small subcluster sizes that, given a fixed number of available instances, leads to a larger number of subclusters. The latter translates in a larger fan-out overhead.

We anticipate that in realistic setting, the number of hops overhead should have a greater impact compared to the fan-out overhead (i.e., $\beta >> \alpha$). This is especially true in application scenarios with massive data rates. Further, subcluster sizes are not fixed but benefit from elasticity and load balancing to vary in accordance with the current input load. Hence, we conclude that the best parallelization strategy should set $h(X)$ to its lower bound as a constraint and, at the same time, try to minimize $f(X)$, given the current input load. This is exactly what *StreamCloud* does, partitioning a query according to the number of stateful operators and assigning to each subcluster the minimum number of instances required to cope with the input load.

REFERENCES

[1] D. L. Mills, "A brief history of ntp time: memoirs of an internet timekeeper," *Computer Communication Review*, vol. 33, no. 2, pp. 9–21, 2003.

[2] N. Tatbul, U. Çetintemel, and S. B. Zdonik, "Staying fit: Efficient load shedding techniques for distributed stream processing," in *International Conference on Very Large Data Bases (VLDB)*, 2007, pp. 159–170.