2010 International Conference on Distributed Computing Systems

StreamCloud: A Large Scale Data Streaming System*

Vincenzo Gulisano[1], Ricardo Jimenez-Peris[1], Marta Patino-Martinez[1], Patrick Valduriez[2]

[1] *Facultad de Informatica, Universidad Politecnica de Madrid, Madrid, Spain*
{vgulisano,rjimenez,mpatino}@fi.upm.es

[2]*INRIA and LIRMM, Montpellier, France,* Patrick.Valduriez@inria.fr

***Abstract*— Data streaming has become an important paradigm for the real-time processing of continuous data flows in domains such as finance, telecommunications, networking, ... Some applications in these domains require to process massive data flows that current technology is unable to manage, that is, streams that, even for a single query operator, require the capacity of potentially many machines. Research efforts on data streaming have mainly focused on scaling in the number of queries or query operators, but overlooked the scalability issue with respect to the stream volume. In this paper, we present *StreamCloud* a large scale data streaming system for processing large data stream volumes. We focus on how to parallelize continuous queries to obtain a highly scalable data streaming infrastructure. *StreamCloud* goes beyond the state of the art by using 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. *StreamCloud* is implemented as a middleware and is highly independent of the underlying data streaming engine. We explore and evaluate different strategies to parallelize data streaming and tackle with the main bottlenecks and overheads to achieve scalability. The paper presents the system design, implementation and a thorough evaluation of the scalability of the fully implemented system.**

I. Introduction

There is a wide spectrum of applications that require real-time processing of data streams, e.g. financial data analysis, processing the output of large sensor networks, etc. The requirements of these applications have been identified in [1] where it has been recognized the inability of former paradigms such as store and process provided by transactional databases to deal with the real time processing of data streams. Stream processing engines (SPEs) have been proposed to address the specific challenges of these new applications [2]–[6].

Some emerging applications such as network monitoring and fraud detection, are pushing the limits of current data streaming infrastructures. These applications are characterized by their real-time requirements and very high-volume data streams. For instance, in cellular telephony and credit card payment systems, because of these current limitations, fraud can only be detected after it happens, what is very expensive in economical terms. 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. Most queries for fraud detection require one or more joins of the CDR stream with itself using complex join predicates, requiring the comparison from 100 million to 2,500 million pairs of CDRs per second.

In the last few years, there have been substantial advancements in the field of data stream processing. From the initial centralized SPEs, the state of the art has progressed to SPEs able to distribute queries among a cluster of nodes or even distributing different operators of a query across different nodes [7]. However, there is an increasing number of emerging applications with requiring real-time processing of very large-volume data streams. The volume of the data streams is such that a single node is not able even to process a single operator of a query for the entire data stream. Distributed SPEs, such as Borealis [7], limit their scalability by allocating an individual operator to a single node. That is, a node running a particular operator has a maximum throughput T. Therefore, it can at most process data streams at a rate of T, thus limiting the stream volume which the whole system can handle to the capacity, T, of a single node. Parallelization of SPEs in cluster environments has been mostly overlooked with a few exceptions such as Aurora* [8] and Flux [9]. Aurora* provides box splitting as a mechanism to scale. However, it requires the entire stream to go through a single node at the beginning of the split box and a single node at the end of the split box. No performance evaluation of the Aurora* approach is available. But it introduces a single-node bottleneck and hampers scaling up to stream volumes larger than the ones a single node can process. Flux [9] resorts to the exchange parallel operator that requires to be implemented for each specific data stream system. The performance evaluation results provided in [9] are based on simulations and are limited to a single operator.

This paper proposes a highly scalable data stream system able to process massive data flows with emphasis on maximizing scale-out. The problem under consideration is the processing of large data flows at input rates that largely exceed the processing capacity of individual nodes even though a node only runs a single query operator. In this paper, we present *StreamCloud*, a highly scalable stream processing system. *StreamCloud* provides high scalability by exploiting intra-operator parallelism [10]. That is *StreamCloud* is able of distributing any individual subquery (as small as a single operator) to *a large set of shared-nothing nodes (subcluster)*. *StreamCloud* does not concentrate a data stream

*Patent pending.

1063-6927/10 $26.00 © 2010 IEEE
DOI 10.1109/ICDCS.2010.72

over a single node. Instead, a logical data stream is split into multiple physical data substreams that flow in parallel from one subcluster to the next one, thus avoiding single-node bottlenecks. This approach enables both to scale with respect to the stream volume by aggregating the computing power of subcluster nodes and with respect to the continuous query window size by aggregating their memory. That is, on one hand, the computing power of multiple nodes is used to share the load of processing a data stream. On the other hand, the windows of stateful operators are distributed across nodes enabling to have larger (distributed) windows. The parallelization performs communication only when it is absolutely required, i.e. for repartitioning data before each stateful operator thus minimizing the overhead introduced by distribution. The minimization of the distribution overhead is a crucial feature of *StreamCloud* since the targeted large scale (100s of nodes) requires the minimization of the footprint of the system in terms of required number of nodes for processing a given load and therefore the resulting cost.

StreamCloud provides transparent query parallelization. That is, the user expresses regular queries that are automatically parallelized by *StreamCloud*. A compiler transforms the abstract query into a parallel query that is automatically allocated to a given cluster of nodes.

StreamCloud is implemented as a middleware system that runs on top of a distributed SPE (currently Borealis [7], yet highly independent of it). This independence is achieved by parallelizing queries using regular data stream operators.

StreamCloud is also able to exploit the multiple CPUs, cores, and hardware threads of each node. It can scale with both by increasing the number of nodes and the number of processors (CPUs, cores and/or hardware threads) per node. The scalability of *StreamCloud* has been evaluated extensively, evaluating both, the scalability of individual operators and full queries. To the best of our knowledge this is the first real implementation and evaluation of a scalable data stream processing system. The results demonstrate its superior scalability with respect existing approaches.

The contributions of this paper are:

- *A highly scalable data stream processing system* for shared-nothing clusters. *StreamCloud* is a full-fledged system, with a complete implementation currently being used for industrial applications;
- a novel parallelization approach that minimizes the distribution overhead;
- a *parallelization technique independent of the underlying SPE* based on standard data stream operators;
- *transparent parallelization of queries with a query compiler*;
- and a thorough *scalability evaluation of a real and fully implemented system in a large cluster (60 dual and quad-core nodes amounting 160 processors)*.

The rest of the paper is organized as follows. In Section II, we make precise the system model and give the problem definition. In Section III, we present the overall system architecture of *StreamCloud* and the detailed description of its parallelization strategy. Section IV gives a thorough evaluation of *StreamCloud*. Section V discusses related work. Section VI concludes.

II. System Model and Problem Definition

A. System Model

A data stream S_i is an infinite append-only sequence of tuples with the same schema $(A_1^i, A_2^i, \ldots, A_{n_i}^i)$. Tuples of a stream S_i have a timestamp that specifies their time of origin. We assume that query sources and system nodes are equipped with well-synchronized clocks using a clock synchronization protocol such as Network Time Protocol as in [11]. Continuous queries are defined over one or more input data streams, $\{S_1, S_2, \ldots, S_m\}$. Each query is modeled as a network of connected operators. A connection represents a data flow. Typical query operators of SPEs are *filter, map, union, join, and aggregates* [4]. These operators are analogous to relational algebra operators. Operators can be classified as stateless (filter, map and union) or stateful operators (join and aggregates). Stateless operators do not hold any state across tuples and perform their computation on a single tuple. Because of the infinite nature of the data stream, stateful operators perform their computation over 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). The amount of data in the window and the amount by which the window slides are parameters.

Map is a generalized projection operator. It applies a set of functions to each input tuple. The output stream can have a schema different from that of the input stream, but the timestamps of input tuples are preserved.

Filter is like a case statement. For every input tuple, it checks if it satisfies predicates $P_1, \ldots, P_m$. Its output consists of $m + 1$ streams. Tuples that satisfy the predicate P_1 are output on the first output stream, the ones that satisfy P_2 but not P_1 are output in the second stream and so on.

Union is used to merge two or more streams with the same schema into a single output stream. Union can output tuples in any order. Output tuples have the same schema and values as input tuples.

Aggregate computes an aggregate function (e.g. average) on sliding windows over its input stream. When the difference between the timestamps of an input tuple and the oldest tuple in the window is larger than the size of the window (time based window) or the number of tuples exceeds the size of the window, an output tuple is produced. Output tuples are tagged with the smallest timestamp in the window.

Join is a binary operator that takes two input streams and a predicate over pairs of tuples (each one from one input). Given two input streams, S and U, both with timestamp t, and a window size, w, join matches tuples that satisfy the predicate and $|s.t - u.t| <= w$. It outputs a tuple that is the concatenation of the input tuples with timestamp $min(s.t, u.t)$. The join operator is deterministic. It takes always the tuple with the smaller timestamp from the two incoming streams.

B. Problem Definition

The focus of this paper is on a parallel-distributed SPE deployed in a shared-nothing cluster connected with a LAN. We aim at supporting intra-operator parallelism in addition to inter-query and inter-operator parallelism. That is, enabling a query operator (in general, any subquery) to be allocated to an arbitrary set of nodes or *subcluster*.

StreamCloud addresses some specific goals: **Scalability**. It is the main design goal with emphasis on the ability to scale with respect to the data stream volume (e.g., tuples per second processed) by exploiting intra-operator parallelism in a shared-nothing architecture [10].

Transparency. Queries should be parallelized in a transparent way, without any user intervention.

Portability. The parallelization should be highly independent from the underlying distributed SPE to be used with different SPEs and thus be more general.

III. *StreamCloud*: Parallel Data Streaming

A. Overview

The *StreamCloud* system is deployed on a cluster of nodes. Each node might run as many instances of the *StreamCloud* system as processors (CPUs, cores and/or threads) available. In order to attain intra-operator parallelism queries are split into subqueries. A subquery can be as small as a single operator. Each subquery is allocated to a subset of nodes or *subcluster*. That is, each *StreamCloud* instance running at each of the nodes of the subcluster executes the same subquery (called local subquery) for a fraction of the input data stream, and produces a fraction of the output data stream. In order to attain high scalability, bottlenecks should be avoided. This means that a logical data stream is never routed through an individual node. Instead, a data stream always flows in parallel from one subcluster to the following one as multiple physical substreams. Therefore, the output from a node of a subcluster might be potentially distributed to all the nodes of the following subcluster(s).

StreamCloud requires some extra components to achieve the parallelization of a query (see Figure 1). It requires a *load balancer (LB)* at the end of each subquery of each node of a subcluster to redistribute the output to the nodes of the next subcluster(s). If the next subcluster contains only stateless operators, it distributes the load evenly across all nodes. Otherwise, if it contains a stateful operator, it distributes the data with *semantic awareness*. In this case, it applies a *data partitioning scheme* aware of the operator semantics that guarantees that data that needs to be aggregated or joined together are received by the same node. For instance, in the case of an equijoin, data of the two incoming streams is partitioned so that all tuples of both streams with the same value of the joining attribute are received by the same node.

Each node of a subcluster therefore receives input streams sent from the load balancers of the previous subcluster(s). An *input merger (IM)* is used at the beginning of the allocated subquery to merge all the input streams received from the

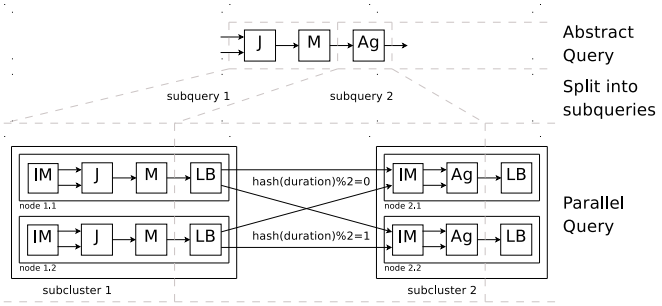

Fig. 1. Query Parallelization

load balancers of the previous subcluster(s) and forward the merged stream to the local subquery. For the join and Cartesian product operators, there is one IM for each input stream. *StreamCloud* provides two different IMs: 1) A best effort IM that simply merges tuples from the incoming streams as they are received; and 2) an IM that guarantees that the parallel operator executes queries equivalently to the non-parallel operator, i.e., it enforces fully transparent parallel processing.

In order to attain the portability goal, our parallelization strategy, materialized as load balancers and the input mergers, is implemented using standard data stream operators instead of custom parallelization operators (unlike Flux). The load balancer is implemented as a filter operator that routes each tuple to the node in charge of the data partition the tuple belongs to. The input merger is implemented as a union operator that merges all incoming substreams into a single one. Since the parallelization is implemented using standard data streaming operators, filters and unions, *StreamCloud* becomes independent of the underlying SPE.

B. Parallel Data Streaming Strategies

The main goal of *StreamCloud* is to scale up to high volumes of streaming data and it attains this goal by parallelizing queries. Scaling with respect to the stream volume requires *intra-query parallelism*, and more precisely *intra-operator parallelism* [10]. There are several alternative strategies to parallelize queries in a shared-nothing environment. A full query can be allocated to a subcluster of nodes in which each node executes the full query for a fraction of the input data stream. Communication across nodes needs to happen at least before each stateful operator to partition data in a consistent and balanced way. We call this parallelization strategy *query-cluster*. This means that, in general, there will be communication from all to all nodes (N to N, N being the total number of nodes) for every stateful operator in the query. In Figure 2, the abstract query on the top of the figure consists of a map, filter, join, filter, aggregate, and a map. In the example, the query is parallelized using this strategy across a cluster of 90 nodes. Each node receives a 90th of the input data and executes the query for that data. Since each stateful operator requires to partition data in such a way that data to be aggregated/joined together is sent to the same instance,

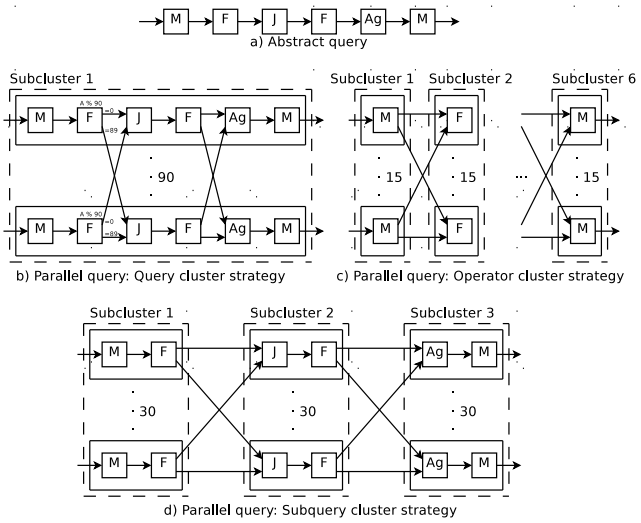

Fig. 2. Query Parallelization Strategies

which requires redistributing all data being sent to the stateful operator. In the query-cluster approach, the communication is from all nodes to all nodes for every stateful operator.

Another parallelization strategy consists in allocating each individual operator of a query to a set of nodes (operator subcluster). Operator subclusters are connected according the query. In this case, the communication pattern is n_i to n_j (being n_i the number of nodes of the operator subcluster i). We call this strategy *operator-cluster*. In Figure 2, each operator is executed in a subcluster of 15 nodes. In general, the output of each subcluster needs to be redistributed to the next subcluster since subclusters may vary in size. In this case, there would be 4 communication steps (one between each pair of consecutive operator subclusters). It would involve communication from all nodes of one subcluster to the next one.

It should be observed that in order to parallelize a query, the only indispensable communication happens for stateful operators to guarantee that data that should be aggregated/joined together are received by the same node. Stateless operators can be easily parallelized since they compute their result based on the current tuple. On the other hand, stateful operators compute their results aggregating/joining data. If the data is distributed blindly, some results may be missed. For instance, two tuples that need to be aggregated together if they reach two different nodes, they will never be aggregated together. Therefore, data needs to be partitioned consistently before each stateful operator. By mapping subclusters to stateful operators one can minimize the communication overhead. That is, data do not travel across the network unless necessary. Our parallelization strategy is based on this fact and its goal is to parallelize queries while minimizing the distribution overhead. For this purpose, each query is divided into *subqueries*. There are as many subqueries as stateful operators the query has, plus potentially 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(s) or the end of the query. If the query starts with stateless operators, the first subquery consists of all stateless operators before the first stateful operator(s). In Figure 2, this parallelization strategy is depicted. Three subclusters are used. The first one contains the prefix of stateless operators, and the next two, a stateful operator followed by the following stateless operators. Each subcluster contains a third of the nodes, i.e. 30 nodes. The communication is like in the query-cluster strategy, but not from all nodes to all nodes, but from one subcluster to the next one.

Comparing the cost of the three strategies, we have to take into account two overheads: 1) The number of communication steps that involve serializing, deserializing and communicating data for each tuple; 2) The cost associated to the fan-out, that is, the number of communication channels maintained, each consuming a set of resources (e.g. memory). The query-cluster and subquery-cluster strategies perform the minimal number of communication steps, that is, as many as stateful operators in the query. The operator-cluster, on the other hand, involves the maximal number of communication steps, as many as operators in the query. In terms of fan-out overhead, query-cluster has the highest one since it involves communication from all to all nodes as many times as stateful operators appear in the query. While the operator-cluster and subquery-cluster strategies have the minimal fan-out overhead, involving only communication across subclusters.

We illustrate the query parallelization process by means of the sample query depicted in Figure 1. The query contains three operators: join (J), map (M) and aggregate (Ag). This query receives two streams, one with call description records (CDRs) with origin phone number, start and end time, and another one with suspicious phone numbers. It performs a join that matches CDRs with suspicious phone numbers. Then, it transforms the start and end times of the call into duration in minutes via a Map. Finally, it aggregates calls by grouping them by duration. The query is split into two subqueries. Subquery1 consists of a Join followed by a Map, and subquery2 consists of the aggregate operator. They are allocated to subclusters 1 and 2, respectively. The data partitioning for the join is done according to the hash of the phone number modulo 2. The map operator, since it is stateless and does not require any specific data partition, is collocated with the join operator. The data partitioning for the aggregate is performed on the hash of the duration, the attribute on which the aggregation is performed, modulo 2. Node 2.1 is responsible for %2=0, and node 2.2 for %2=1. At each node of subcluster2, the input merger receives the results of subquery1 coming from the two nodes of subcluster1 and merges and forwards them to the local aggregate operator.

The aforementioned parallelization is best effort. That is, it does not guarantee that the parallel execution is equivalent to the non-parallel one. *StreamCloud* also provides a parallelization that guarantees that the parallel execution is equivalent to a non-parallel one. This fully transparent parallelization is useful for applications with strict requirements in the ordering

of tuples within data streams.

In the rest of this section, we describe the parallelization of the stateless and stateful subqueries in more detail, in terms of the load balancers and input mergers which encapsulate the parallelization logic. We first describe the best effort parallelization (Figure 3.a) and then, the fully transparent parallelization (Figure 3.b).

C. Parallelization of Stateless Subqueries

Stateless subqueries are those that only have stateless operators, that is, either the full query does not have any stateful operator or the query starts with one or more stateless operators. In the later case, all the stateless operators until the first stateful operator form a stateless subquery. Since the subquery only has stateless operators, each input tuple can be processed by any node in the subcluster assigned to the subquery. The load balancer (see Figure 3 lines 1-5) applies a round-robin strategy to distribute the tuples across nodes of the subcluster.

D. Parallelization of Stateful Subqueries

Stateful subqueries in *StreamCloud* contain one stateful operator followed by zero or more stateless operators. Stateless subqueries allow sending each input tuple to an arbitrary node of the operator subcluster. However, the load balancer for stateful subqueries needs to be enriched with semantic awareness. Basically, in order to split the load across the nodes of a stateful subquery subcluster, it should take into account the semantics of the stateful operator and its data partitioning scheme. In what follows, we discuss the parallelization of stateful subqueries containing the three main operators we have considered: join (equijoin), Cartesian product (general join), and aggregate.

1) Join Operator: The join operator we consider is an equijoin with a set of predicates between pairs of attributes of the input streams, for instance, *right.phoneNumber = left.phoneNumber*, on the left and right input streams being joined. *StreamCloud* uses partitioned parallelism [10] where different nodes execute the same subquery on different input data partitions and a symmetric hash join approach [10]. The join matches tuples from the windows (e.g. tuples received in the last hour or the last 1,000 received tuples) associated to the incoming streams that fulfill the joining predicate. Each input stream is split by load balancers (see Figure 3 lines 6-10) into N substreams (being N the number of nodes in the join subcluster) according to the hash of the attribute used in the equality predicate. Assuming that A is the attribute of one of the input streams used in the equality predicate, $hash(A)\%N$ is used to determine to which target node of the subcluster the tuple should be sent. In this way, tuples that have to be matched together are received by the same node. For instance, in the example in Figure 1,the join is an equijoin that matches the phone number in CDRs in one of the input streams with suspicious phone number in the other input stream. So, hashing is applied to the phone number attribute. Every tuple with the same phone number will be redirected to the same node in the subcluster (their hash value is the same) and therefore, all the required matchings are found. If the equijoin contains multiple predicates, the hash is computed over all the attributes referred in the predicates. Thus, each particular combination of attribute values will go to the same node and could be matched locally.

2) Cartesian Product Operator: The Cartesian product (CP) operator performs general joins with arbitrarily complex predicates. As in the join operator, it takes two input streams and considers the tuples stored in the windows associated to each stream and applies a predicate to each pair formed by a tuple of each window producing tuples that match the predicate. This operator is parallelized as follows (see Figure 3 lines 11-23). Given a subcluster of N nodes to execute the CP operator, each tuple is sent to $M = \sqrt{N}$ nodes of the destination subcluster (lines 15-22). Therefore, each load balancer splits its output into M substreams (line 13), according to a hash of the tuples $\%M$ (line 14). The streams sent to the CP operator are dealt with differently depending on whether they correspond to the right or left input streams in order to distribute data coherently. Each CP instance receives two set of substreams, each set consisting of the substreams sent by each node in the previous subcluster.

Figure 4 depicts an example with a query with two map operators (M^l and M^r) and a CP operator with time windows. On the top part of the figure, the abstract query is shown with a sample input and the resulting output. In the bottom part, we show the parallel version of the query in which map operators are deployed on a subcluster of 2 nodes and the CP operator in a subcluster of 4 nodes ($N = 4$). The timestamps of the tuples are indicated on the top of each stream (the values at the right of the "ts" tag). One of the incoming streams is 0-3 and the other A-D. Each of these 2 streams are split into two substreams. The stream 0-3 is split into 2 substreams with the values (0,3) and (1,2). The stream A-D is split into 2 substreams with the values (A,C) and (B,D). The substream (0,3) is sent to CP instances 0 and 1, and the substream (1,2) is sent to the instances 2 and 3. The substream (A,C) is sent to instances 0 and 2 and the substream (B, D) to instances 1 and 3. This splitting results in each of the 4 CP instances performing 1/4th of the whole Cartesian product on the incoming streams. That is, each CP instance would perform the Cartesian products (0,3)×(A,C), (0,3)×(B,D), (1,2)×(A,C), and (1,2)×(B,D), respectively.

3) Aggregate Operator: In order to produce the aggregation in a consistent way and minimize communication, the parallelization of the aggregate operator requires that all the tuples to be grouped together according to the group-by clause of the operator should to be processed by the same node of the subcluster. This is the criterion used to partition data across nodes in the subcluster. In the example in Figure 1, the operator Ag aggregate calls by their duration. In order to parallelize this operator, the data is partitioned in a similar way to the join (see Figure 3 lines 6-10). That is, the set of attributes used in the group-by clause are hashed and the resulting value modulo N is used to split the stream across

```
1  Load Balancer for Stateless Subquery
2  Upon arrival of t:
3      destination:=(destination+1) % N;
4      forward(t, nextSubcluster[destination]);
5  end
6  Load Balancer for Join & Aggregate
7  Upon arrival of t:
8      destination:= hash(t.A, t.B, ...) % N;
9      forward(t, nextSubcluster[destination]);
10 end
11 Load Balancer for Cartesian Product
12 Upon arrival of t from stream S (left or right):
13     M:= √N;
14     j:= hash(t) % M;
15     for i:= 0 to M-1 do
16         if left(S) then
17             destination:= j*M+i;
18         else
19             destination:= j+i*M;
20         end
21         forward(t, nextSubcluster[destination]);
22     end
23 end
24 Best Effort Input Merger
25 Upon arrival of t:
26     forward(t, localSubquery);
27 end
```

a) Best effort parallelization

```
28 Timeout Management at All Load Balancers
29 Upon forwarding a message to subcluster[destination]:
30     lastTS[destination]:= t.TS;
31     lastTime[destination]:= currentTime();
32 end
33 Upon ∃dest ∈ {0..N − 1}|currentTime() >= lastTime[dest]+d
34         -- d time units elapsed since last sending to the
           destination;
35     dummy.TS:= lastTS[dest];
36     forward(dummy, nextSubcluster[dest]);
37     lastTime[dest]:= lastTime[dest]+d;
38 end
39 Transparent Input Merger
40 Upon arrival of t:
41     if ¬ dummy(t) then
42         buffer[t.sender].enqueue(t) ;
43     end
44     latestTS[t.sender]:= t.TS;
45     -- get oldest TS across senders
46     lastTS:= min_{i∈{0..N−1}}(latestTS);
47     next:= j|∀i ∈ {1..M} Buffer.nonEmpty[i] ∧ latestTS[j] <= latestTS[i];
48     -- forward oldest tuple, if available;
49     if next ≠ ∅ then
50         forward(buffer[next].dequeue(), localSubquery);
51     end
52 end
```

b) Additions for transparent parallelization

Fig. 3. Formal Description of Query Parallelization

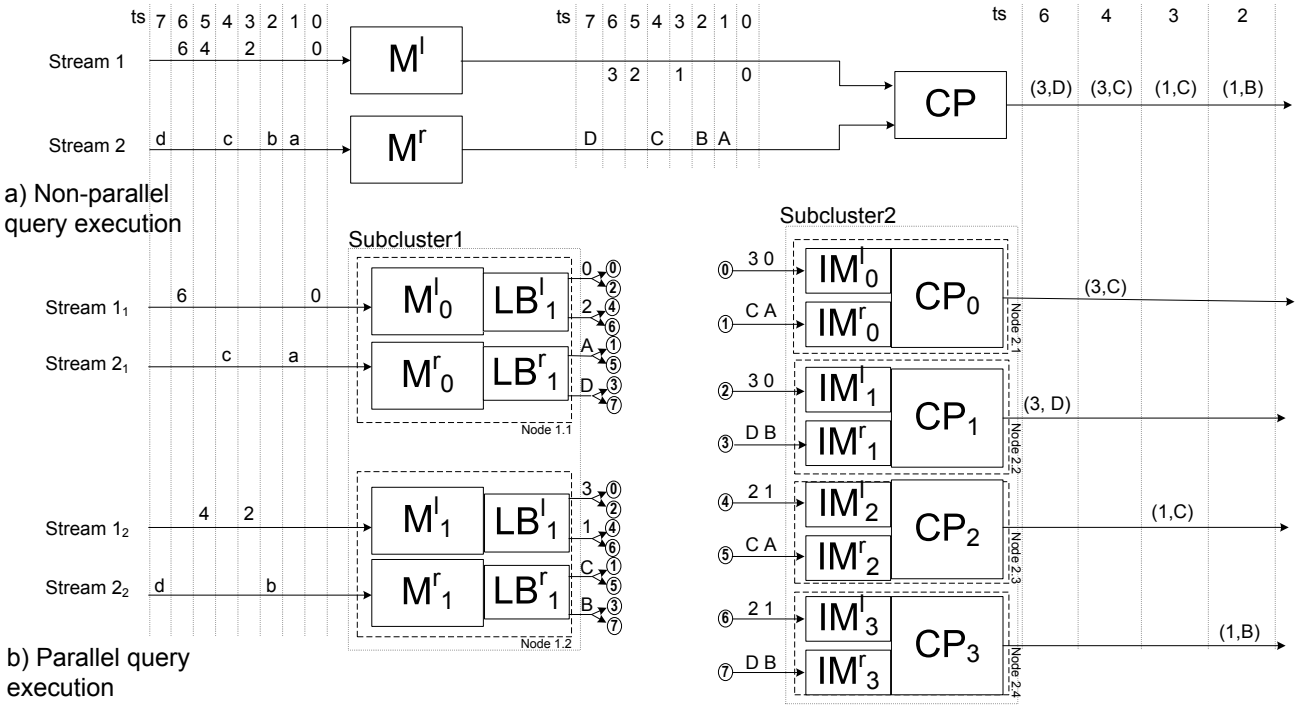

Fig. 4. Cartesian Product Sample Execution

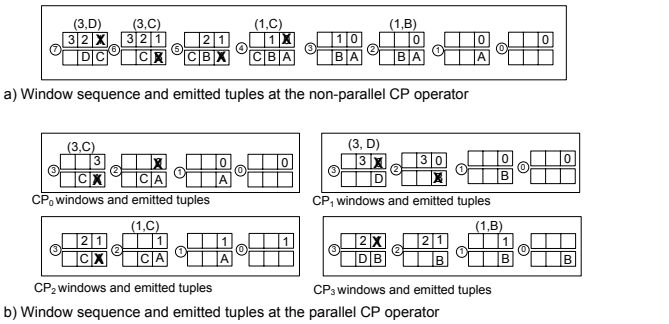

Fig. 5. Cartesian Product Sample Execution Operator Window Evolution

the N subcluster nodes.

E. Transparent Parallelization

The best-effort query parallelization described so far may produce results that are not equal to the ones produced by the non parallel version. For instance, in the previous example in Figure 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 Figure 5.a. When a new tuple t is received the window corresponding to the other input stream is trimmed, removing those tuples with a timestamp difference with t timestamps bigger than the window size. In the example, the reception of the tuple with value C (step 4 in Figure 5.a) causes the removal of the tuple with value 0, the arrival of the tuple with value 2 causes the removal of tuple with value A (step 5), and so on. This update is independent of the interleaving on the incoming streams and only depends on the timestamps of the incoming tuples.

With the best-effort parallel version, the following scenario could happen. At CP_3 the tuples with values 1 and 2 are received. Then, the tuple with value D is received. The arrival of D causes the removal of 1 from the window. Then, the tuple with value B is received, but the tuple with value 1 is not in the window anymore, which results in not yielding the output tuple (1, B). The reason is that, because the map operator is parallelized and D and B are produced by two different nodes, they might be interleaved in an arbitrary order at CP_3, such as D is first received and then B. This results in a different tuple order in this input stream than the one that would happen with a non-parallel map, in which case it would be guaranteed that tuples in the same stream are received in timestamp order. Therefore, the best effort parallelization strategy can produce results that would never be produced with the original non-parallel query, thus failing to provide transparent parallelism.

The transparent parallelization of *StreamCloud* (see Figure 3 lines 28-52) addresses this issue by enhancing the input merger (IM). The IM (lines 39-52) performs a merge sort that takes timestamp ordered substreams coming from the load balancers and generates a single merged substream that is timestamp ordered. It guarantees that each local operator instance gets a timestamp ordered input stream and therefore, it produces a timestamp ordered output stream. The IM outputs a tuple whenever it has received at least one tuple from each input substream in order to take the one with the smallest timestamp (lines 46-51). In order to avoid blocking of the IM, in case that one or more of the input substreams become empty for a period of time, load balancers generate dummy tuples for those destinations for which they have not generated any tuple in the last d units of time (lines 28-38). The dummy tuple is timestamped with the timestamp of the last tuple sent to that destination plus the $units$ of time elapsed since then (line 37). The dummy tuple is not forwarded to the operator (lines 41-43) but enables to take tuples from other substreams at the IM (lines 44-47) and output new tuples, which avoids blocking. This IM can be implemented by means of standard operators such as union, order, and synchronization operators.

Let us look again at the example in Figure 4 where the non-parallel and parallel execution of a query are shown (IMs and LBs not shown to simplify the picture). The transparent IM would produce the window evolution shown in Figure 5.b independently of the relative speeds across the nodes of a subcluster, thus resulting in different interleavings of the different substreams. That is, despite the output streams of each map instance are not coordinated, the IM guarantees that tuples enter the CP window in timestamp order, thus, guaranteeing an equivalent execution to the non-parallel operator. The scenario aforementioned for the parallel execution that resulted in missing the production of the tuple (1, B) would be treated as follows. The arrival order of tuples was (1, 2, D, B). At CP_3 the IM receiving the numerical stream would receive first 1, store it in buffer[1] (Figure 3 line 42). Since buffer[0] is empty (line 44-49), it would not forward any tuple (line 50). When tuple with value 2 arrives, it would be stored in buffer[0]. Now tuple 1 (the one with smallest timestamp) would be forwarded to CP_3 (line 50). Then, the tuple with value D arrives at the IM of the alphabetical stream and is stored in buffer[1]. Since buffer[0] is empty, no tuple is forwarded to CP_3. When a tuple with value B is received, it is stored in buffer[0]. Now, tuple B that has the smallest timestamp is forwarded to CP_3. CP_3 then produces the tuple (1, B). As can be seen, the transparent IM input merger solves the issue of the missed tuple (1, B) in the scenario that the best effort IM failed to produce.

F. Optimizations

The parallelization described so far splits a query into a set of subqueries, each containing a stateful operator followed by zero or more stateless operators (with the possible exception of the first subquery that might be fully stateless). If two consecutive stateful subqueries, that is, with join and/or aggregate operators, partitioned data in the same way (e.g. a join by phone number and an aggregate with a group by also by phone number), then the two subqueries may be merged in a single one to avoid communication (and the associated processing like splitting, balancing and merging the stream) between two subclusters.

Since all subqueries contain an input merger at the beginning and a load balancer at the end, unions at the beginning

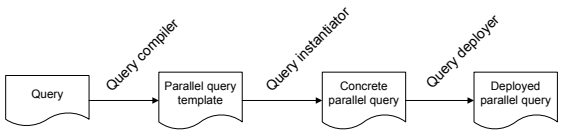

Fig. 6. Query parallelization

of a subquery are subsumed by the input merger, and filter operators appearing just at the end of a subquery are removed and its logic combined with the logic of the filter used in the load balancer located at the end of the subquery.

G. Query Compiler

StreamCloud parallelizes queries automatically through its query compiler. The process is shown in Figure 6. The compiler takes a regular query and splits the query into subqueries according the aforementioned parallelization strategy. It then applies the optimizations discussed in the previous section. The output of the query compiler is the set of subqueries. To instantiate the parallel query, the query instantiator takes the number of available processors and the set of subqueries. It evaluates the cost of each subquery and assigns proportionally to this cost processors to each subcluster. The cost of each subquery is measured as the time needed to process one tuple. The output of the query instantiator is a concrete parallel query that has placeholders for IPs and ports to be used by each *StreamCloud* instance. The final component is the query deployer that, given a deployment descriptor with the IPs and ports of the processors to be used, deploys the concrete query assigning IPs/ports to each *StreamCloud* instance and yielding the deployed parallel query.

IV. Experimental Evaluation

The evaluation targets to measure the scalability of the *StreamCloud* approach. It evaluates the scalability with respect to the computing power (number of processors) and window size. The evaluation methodology is the following. Experiments inject increasing input loads for different configurations. *StreamCloud* instances process tuples at the input rates until they get overloaded. No load shedding was used during the evaluation. When the system gets overloaded, the input rate becomes higher than the operator processing rate and tuples start to be queued at all instances until they are processed. Each experiment shows the evolution of the *throughput* (measured as processed tuples or comparisons performed by an operator or complete query per second) for increasing loads. The experiments also report the *average CPU usage* (average use across all processors in the system) and the *average queue length* (average queue length across all *StreamCloud* instances in the evaluated configuration) to determine when the system saturates. The first set of experiments (Sections IV-B, IV-C) evaluates how increasing both the computing power and window size by aggregating the computer power and memory of multiple nodes enables to deal with higher input loads. This set of experiments has been performed in two stages. In the first stage, the scalability of each individual operator has been evaluated. This evaluation shows how operator subclusters scale and the associated overhead. For this purpose, the throughput of each operator is evaluated individually for increasing loads (input tuples per second) and different numbers of nodes. In the second stage, the scalability of full queries has been evaluated. We compare our approach with two other parallelization techniques: 1) Every node in the cluster executes the full query for a subset of the data (query-cluster approach), and 2) each operator is executed in a subcluster of nodes (operator-cluster approach). This enables to measure the scalability of the proposed parallelization technique system in a real deployment.

A second set of experiments evaluates how, for computing intensive operators, it is possible to scale the input load with respect a fixed a global window size by increasing the system size.

In the third set of experiments, we evaluate how *StreamCloud* scales not only with respect to the overall number of processors, but how it scales with respect to the number of processors available per node, by deploying multiple instances of *StreamCloud* per node. This demonstrates the ability of *StreamCloud* to exploit the available CPUs, cores and hardware threads of each node independently of the ability of the underlying SPE to exploit them (e.g. Borealis does not fully exploit the available processors on each node).

A. Evaluation Setup

The evaluation was performed in a cluster of 60 nodes (blades) with 160 cores (processors). All blades are Supermicro SYS-5015M-MF+ equipped with 2GB of RAM and 1Gbit Ethernet and a directly attached 0.5TB hard disk. Blades are distributed in 3 racks: Rack 1 has 20 blades with a dual-core Intel PentiumD@2.8GHz. Rack 2 has 20 blades with dual-core Intel Xeon 3040@1.86GHz. Rack 3 has 20 blades with quad-core Intel Xeon X3220@2.40GHz. The cluster is shared-nothing. Around half of the nodes were required to inject a load able to saturate the larger configurations.

Since the blades are multi-core, we deployed multiple instances of *StreamCloud* per node (one instance per core available). We report in each experiment how many instances were used. Collocated instances do not communicate among them and thus, the experiments provide the scalability in terms of number of processors.

Two baselines are used to compare the scalability of *StreamCloud*: a) a centralized system consisting of one Borealis instance deployed on a single dual-core node, and b) a single dual-core node running two *StreamCloud* instances. The former enables to compare the scalability of *StreamCloud* with respect a centralized SPE. The latter enables to evaluate the scalability of *StreamCloud* in a uniform way (different configurations with the same number of instances deployed per node, i.e. 2).

All the experiments have 3 phases: warm-up, steady-state, cold-down. The evaluation was conducted during the steady-state phase that lasted for 10 minutes. Each experiment was

run three times, we report the average.

B. Scalability of Individual Operators

We have evaluated both stateless (map) and stateful operators (join, aggregate, and Cartesian product). All the experiments share the same input schema: a call description record consisting of calling and called number, call starting time and duration, district, x and y coordinates and emission timestamp.

1) Map Operator: Figure 7.a shows the evolution of the throughput for different input rates and number of processors. The evaluation shows a linear evolution of the throughput. 20 processors manage an input rate of 30,000 tuples/second (t/s) that is roughly 10 times the input rate managed by two instances. 40 processors double the managed input rate, reaching 60,000 t/s. The throughput curves have two clear different regions separated by the point at which saturation is reached (100% CPU usage). Before saturation, the queue length is null. When saturation is reached, the queue length grows very fast due to the high input rates.

2) Join Operator: The join operator matches phone calls made by the same user and computes their distance in time and space. The window size is one minute. Since Borealis does not implement an equijoin operator, the Cartesian product operator is used (called join in Borealis) to perform the equijoin. This means that the number of comparisons is higher than it would be in a real equijoin. As the throughput is given in number of comparisons per second (c/s), this can give an idea of how many comparisons a real equijoin would make. Figure 7.b shows the evolution of the throughput in c/s for all configurations. From 20 to 40 processors, the throughput almost duplicates, which means that the scalability is almost linear. An interesting observation is that when the load is between 300 and 500 t/s, 40 processors do less comparisons for the same input rate than 20 processors. *StreamCloud* by distributing is removing the unnecessary comparisons performed by the underlying Borealis join. For a given input rate, the number of tuples that are on the time window of each processor will be smaller for larger configurations, since the tuples are split among a higher number of nodes. This means that the unnecessary work performed by the Borealis join - a Cartesian product - will be smaller, which results in larger configurations making less unneeded comparisons than smaller ones, therefore performing better.

Looking at the queue length in Figure 7.b, we can identify three different regions. First, the flat region (value 0), then a region with a mild slope and finally, another region with a steeper slope. For 20 processors the second region goes from 300 to 400 t/s and for 40 processors goes from 500 to 700 t/s. If we compare this graph with the one of the CPU usage, the end of the second region matches the saturation point (i.e. 100% CPU usage). This means that to reach full CPU utilization, some queuing is needed. This effect is related to the imbalance across processors that is higher for larger configurations. An interesting fact is that the length of the queues is smaller for larger configurations, which means that they have a more grateful degradation when getting close to saturation. This is also true for input rates beyond the saturation point.

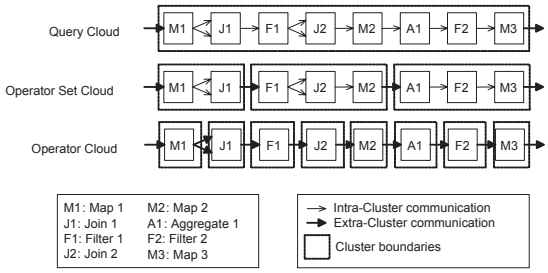

Fig. 8. Query used for the evaluation

3) Aggregate Operator: The aggregate used in the evaluation calculates the number of calls and average duration, grouping results by district over a one minute window which advances every ten seconds. Figure 7.c shows the evolution of the throughput of the aggregate operator with increasing input rates for all configurations. The scalability is linear. 20 processors are able to handle 40,000 t/s, while 40 processors handle, twice as much, 80,000 t/s. The growth of the queue length upon saturation is very explosive due to the high input rates handled.

4) Cartesian Product Operator: In the Cartesian product (CP) experiments, the output and input schemas are the same, every matching tuple is just forwarded as is. The predicate checks whether the tuples of two input streams intersect in time. The window size is one minute.

In Figure 7.d, we can see the evolution of the throughput for increasing input rates. The scaleout is almost linear. 20 processors achieve close to 2,000 million c/s and 40 processors 4,000 million c/s. One difference between the CP operator and all other operators is that, when saturation is reached, the degradation is explosive as can be seen in the queue length evolution in Figure 7.d. This more explosive degradation has to do with the quadratic cost with respect the input load. The explosion in the queue length happens exactly at the points at which CPU usage gets close to 100% (Figure 7.d). With the largest configuration the degradation is milder, which indicates that large configurations are more tolerant to peaks.

C. Scalability of Queries

In this section, we compare the scalability for full queries of the *StreamCloud* parallelization strategy, subquery-cluster with the other two alternatives, query-cluster and operator-cluster. The query used for the evaluation is depicted in Figure 8. In this figure, it is also shown how the query is split into subclusters for the evaluated approaches. All operators (except aggregate) share the same input and output schema, which consists of an input stream with an integer. In this way, we can stress the system with higher input rates.

We evaluated the three parallelization strategies for increasing input rates with 60 processors. For the query-cluster approach, we could only use up to 30 processors, since the system crashed at deployment time with larger configurations.

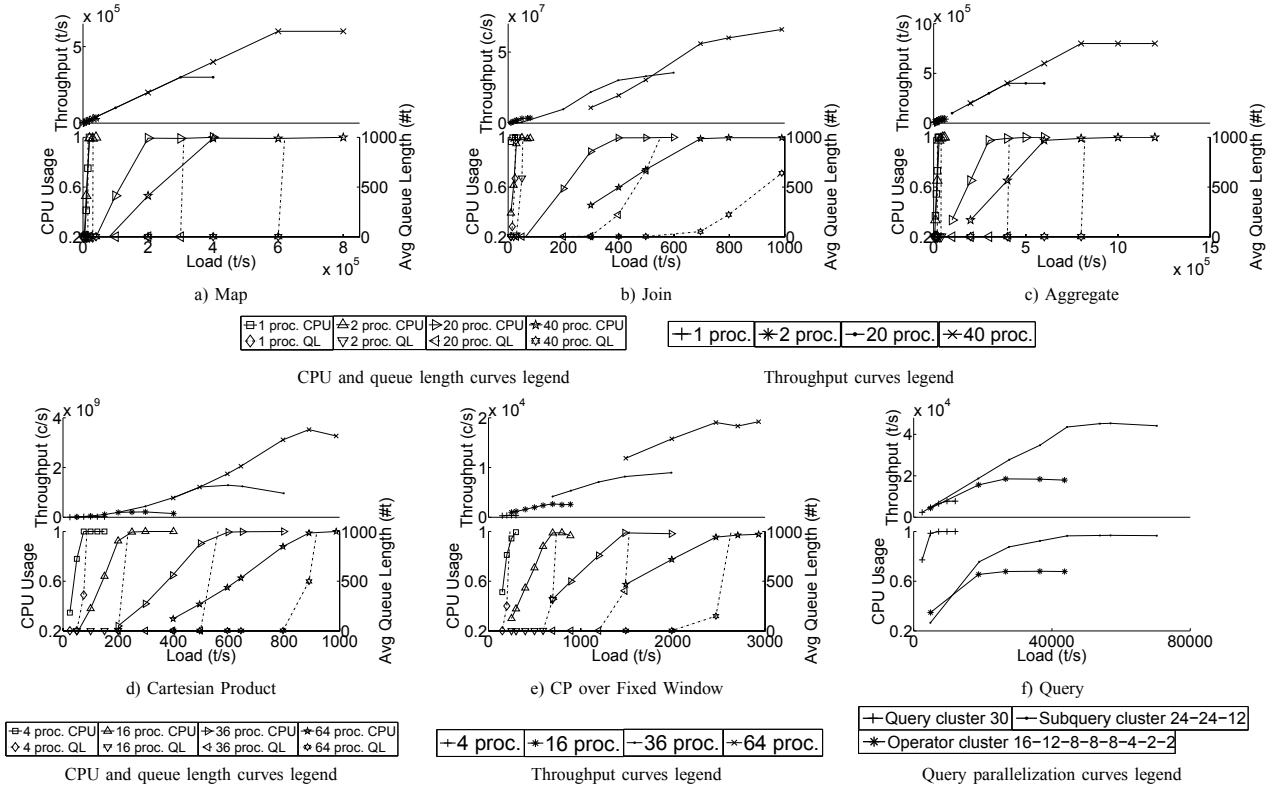

Fig. 7. Individual Operator and Query Scalability Evaluation Results

The use of resources of the query on each processor was proportional to the number of processors in the cluster what made impossible even to deploy the query for 60 processors. For each approach, we used the optimal size for each cluster[1]. Figure 7.f shows the performance of the three parallelization strategies. The throughput of *StreamCloud* is much higher than the other alternatives. Far behind is the operator-cluster approach and at the bottom is located the query-cluster approach.

The main shortcoming of the query-cluster approach is that it requires communication from all to all nodes three times (one for each of the three stateful operators). As can be seen in Figure 9 (to enable the comparison this profiling has been made with 30 nodes for all approaches), the aggregated communication cost and context switches reported as ”other” is above 40%.

The performance difference between operator-cluster and subquery-cluster (*StreamCloud*) is due to two reasons. The first and most important reason is that, when setting the cluster sizes, there is always some unused capacity per cluster due to the inability of using a fractional number of processors. This unused capacity (tagged as ”idle” in the figure) is proportional to the number of subclusters that is higher in the operator-cluster case. The subquery-cluster shows less than 10% of unused capacity, while the operator-cluster shows close to 35% of unused capacity. The second reason is related to the distribution overhead (i.e. communication related costs). Figure 9 shows that the cost of distribution due to filters and unions and underlying inter-node communication is much higher in the operator-cluster than in the subquery-cluster (reported as ”other” in the graph). This is because there are more subclusters in the former case than in the latter one.

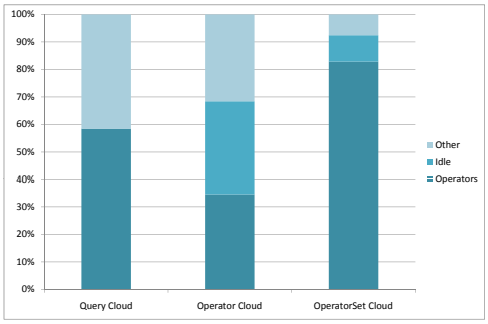

Fig. 9. Query Processing Costs

From the evaluation, we can conclude that the mapping of operators to clusters is crucial for attaining high scalability. *StreamCloud* approach (subquery-cluster) attains a performance that is 2.5 to 5 times better than operator-cluster and

[1]The legends indicate the number of processors per subcluster of each approach. For instance, in Figure 7.f the operator-set approach uses a subcluster per operator with 16, 12, 8, 8, 8, 4, 2, and 2 processors per subcluster.

query-cluster, respectively.

D. Scaling with Fixed-Size Windows

In some cases, the processing associated to each tuple is so heavy that the only concern is to reduce the tuple processing time and enable to process higher input rates for a given fixed window size. For instance, in a Cartesian product (CP), the number of comparisons to be made for each incoming tuple in a large window (say 10s of thousands tuples and above) is very high. In this experiment, we evaluate how much *StreamCloud* can scale with a fixed window size (bounded in number of tuples). The operator used was the CP with a window size fixed to 20,000 tuples being each tuple 12 bytes long yielding to a window of 240 KB. In a single site, this is translated into a single window of the full size. Then, for a configuration with N processors, each instance processes $1/\sqrt{N}$ of the incoming streams. Thus, if there are 4 processors, each processor has a window size of $1/\sqrt{4} = 10,000$ tuples, for 16 processors $1/\sqrt{16} = 5000$ tuples, ...

Figure 7.e depicts the evolution of the throughput. Since windows are bounded by the number of tuples and the selectivity is fixed, the throughput evolution shows how the performance evolves with the number of processors. We can make two observations in Fig. 7.e. First, the scalability is super-linear. The reason is that the higher the number of processors, the smaller the local windows, and thus the smaller the amount of comparisons to be performed per tuple at each processor. This results in processing more tuples per second with configurations with larger numbers of processors. Second, for a given system input rate the throughput is different for each configuration. This is due to the aforementioned effect of decreasing the amount of work per tuple for larger window configurations.

In Figure 7.e, we can also see the evolution of CPU usage and queue length. The CPU usage shows the saturation points (i.e. when CPU usage reaches 100%). Then, if we compare this against the queue length, we observe that the saturation point is reached with very long queues. This means that, from a pragmatic point of view, allowing a maximum queue length of 200 tuples, 4 processors would deal with a system input rate of around 200 t/s, 600 t/s with 16 processors, 1400 t/s with 36 processors, and 2500 t/s with 64 processors.

E. Multi-Cores

We have evaluated so far how *StreamCloud* scales with respect to the overall number of processors used. In this experiment, we want to quantify the scalability with respect to the number of available processors in each node. That is, to evaluate whether *StreamCloud* is able to use effectively the available CPUs/cores/hardware threads of each node by increasing the number of instances allocated to each node. For this experiment, we used the rack with quad-cores and performed the scale-out evaluation with an equi-join from 1 to 4 instances of *StreamCloud* on each node and from 1 to 20 quad-core nodes.

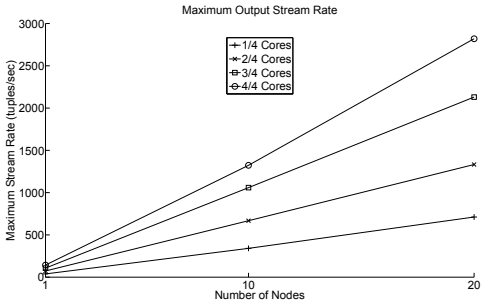

Fig. 10. Join max. throughput vs. number of instances per node

Figure 10 shows the results of the evaluation. We see that the more instances of *StreamCloud* per node (until the 4 cores available per node), the higher throughput is obtained. The reason is the following. Borealis, the underlying SPE, is internally multi-threaded. However, the multi-threading is used for different tasks of a pipeline, like receiving, processing and sending tuples. It does not use multiple threads for processing tuples. This multi-threading results in having active most of the time a single thread, alternating among the receiving, processing and sending ones. In practical terms, this multi-threading uses one core most of the time. This means that *StreamCloud* is able to overcome the lack of parallelism of the underlying SPE by allocating multiple instances to multi-processor nodes.

V. Related Work

Distributed Stream Processing. Distributed SPEs (e.g., Borealis [7]) enable to allocate each operator to a different node. The scalability with respect to the stream volume of current distributed SPEs is limited by the capacity of a single node, since all the data stream that goes through an operator is processed necessarily by the single node that is hosting it. *StreamCloud* overcomes this issue by never concentrating the data stream in any single node.

Clusterized Stream Processing. To the best of our knowledge Aurora* [8] and Flux [9] are the only efforts for parallelizing data streaming in shared-nothing environments. Aurora* provides two load sharing mechanisms, called box sliding and box splitting. Box sliding enables to re-allocate consecutive operators to the upstream or downstream node. This enables to balance the load across nodes for changes in the workload. Box splitting is a more powerful load sharing mechanism that consists in splitting the load across several nodes running the same operator. This approach can be considered the closest to our approach. The main differences stem from the fact that box splitting uses a single filter operator node at the front and a single union operator node at the end. This results in the whole data stream going through a single node (the filter and the union), which bounds its scalability. In contrast, in *StreamCloud*, a data stream never needs to go through a single node. Instead, it goes through from a cluster of nodes to another cluster of nodes in parallel.

Flux extends the exchange operator [12] to a shared-nothing environment for data streaming. This operator is a parallelization operator for parallel databases for multi-processors. The exchange operator has similar goals to our load balancer, namely, parallelizing without having to customize query operators. Both provide semantic awareness (called content sensitive routing in Flux terminology). One important difference between both approaches is that the exchange operator needs to be implemented for each SPE, while our load balancer is implemented via standard filters (and unions for the input merger). Finally, Flux evaluation was performed using a simulation and for a single operator, while we evaluated *StreamCloud* in a real deployment with both individual operators and full queries. These are important issues to quantify the scalability of a parallelization approach. Neither Aurora* nor Flux perform any evaluation of the scalability of the proposed approaches unlike the present paper.

Load Balancing and load shedding. The correlation of workloads to allocate sets of operators to different nodes to avoid correlated load bursts hitting the same node is studied in [13]. [9] repartitions data to balance the node across nodes of an operator set in a shared-nothing architecture.

Load balancing enables to maximize the utilization of available capacity but does not enable increasing the scalability as parallelization does. In fact, it is an orthogonal technique that is also used by *StreamCloud*.

Load shedding complements load balancing when the system reaches saturation by either discarding tuples [11], [14]–[16] or summarizing data tuples [15], [17]. Load shedding enables to increase the scalability by trading off accuracy. In contrast, the parallelization of *StreamCloud* enables to scale without losing accuracy. [16] focuses on how shed load whilst preserving the integrity of windows through the query plan by being aware of the windows of stateful operators, the window size and slide. [11] studies how to coordinate load shedding to attain end-to-end control on the output data quality.

[17] performs semantic load shedding for joins. The output of the operator is approximated by maximizing a user-defined similarity metrics between the exact and approximate answer. [15] presents a load shedding architecture for TelegraphCQ [6] that summarizes tuples using synopses.

Continuous joins over DHTs. There has been recent work on exploiting peer-to-peer networks, in particular, distributed hash tables (DHTs) for processing Continuous multi-way joins over data streams [18] [19]. Although these works exploit hash-based join algorithms, the objective (increasing the size of the sliding window with addition of peers) is different than ours (scaling out) and the assumptions regarding the network (clusters vs. WANs) are very different.

Hardware Stream Processing. In [20], it is proposed to implement stream queries in hardware using Field Programmable Gate Arrays. The hardware implementation of stream operators can dramatically increase the performance by removing many of the software overheads. However, the capacity of the system would still be bounded by the capacity of a single node.

VI. Conclusions

We have presented *StreamCloud* a large scale data streaming system. *StreamCloud* uses a novel parallelization strategy for shared-nothing clusters. *StreamCloud* runs on top of a distributed SPE, but is highly independent from it by implementing the parallelization with standard stream operators. The query compiler automatically parallelizes queries. It also provides a parallelization strategy that is fully transparent, producing executions equivalent to non-parallel SPEs. The evaluation shows that close to linear scalability can be achieved for a large number of processors (60) for both individual operators and full queries. The evaluation demonstrates that the parallelization strategy of *StreamCloud* maximizes the scalability of the system, outperforming by a factor of 2.5-5 the other two alternatives.

VII. Acknowledgments

This research has been partially funded by the European Commission under the Stream project (FP7-216181), the Spanish National Science Foundation (MICINN) under grant TIN2007-67353-C02, the Madrid Regional Research Council (CAM) under the AUTONOMIC project (S-0505/TIC/285) and the CLOUDS project (S2009/TIC-1692), and Microsoft Research Cambridge under the PhD Award programme.

References

[1] M. Stonebraker *et al.*, "The 8 requirements of real-time stream processing," *SIGMOD Record*, vol. 34, no. 4, 2005.
[2] J. Chen and all, "Niagaracq: A scalable continuous query system for internet databases," in *SIGMOD*, 2000.
[3] D. Carney *et al.*, "Monitoring streams - a new class of data management applications," in *VLDB*, 2002.
[4] D. J. Abadi *et al.*, "Aurora: a new model and architecture for data stream management," *VLDB J.*, vol. 12, no. 2, pp. 120–139, 2003.
[5] A. Arasu and all, "Stream: The stanford stream data manager," in *SIGMOD*, 2003, p. 665.
[6] S. Chandrasekaran *et al.*, "Telegraphcq: Continuous dataflow processing for an uncertain world," in *CIDR*, 2003.
[7] D. J. Abadi *et al.*, "The design of the borealis stream processing engine," in *CIDR*, 2005, pp. 277–289.
[8] M. Cherniack *et al.*, "Scalable distributed stream processing," in *CIDR*, 2003.
[9] M. A. Shah *et al.*, "Flux: An adaptive partitioning operator for continuous query systems," in *ICDE*, 2003, pp. 25–36.
[10] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems, Second Edition*. Prentice-Hall, 1999.
[11] N. Tatbul *et al.*, "Staying fit: Efficient load shedding techniques for distributed stream processing," in *VLDB*, 2007.
[12] G. Graefe, "Encapsulation of parallelism in the volcano query processing system," in *SIGMOD Conf.*, 1990, pp. 102–111.
[13] Y. Xing *et al.*, "Dynamic load distribution in the borealis stream processor," in *ICDE*, 2005.
[14] B. Babcock, M. Datar, and R. Motwani, "Load shedding for aggregation queries over data streams," in *ICDE*, 2004, pp. 350–361.
[15] F. Reiss and J. M. Hellerstein, "Data triage: An adaptive architecture for load shedding in telegraphcq," in *ICDE*, 2005, pp. 155–156.
[16] N. Tatbul and S. B. Zdonik, "Window-aware load shedding for aggregation queries over data streams," in *VLDB*, 2006, pp. 799–810.
[17] A. Das, J. Gehrke, and M. Riedewald, "Approximate join processing over data streams," in *ACM SIGMOD Conf.*, 2003, pp. 40–51.
[18] S. Idreos, E. Liarou, and K. M., "Continuous multi-way joins over distributed hash tables," in *EDBT*, 2008.
[19] E. P. W. Palma, R. Akbarinia and P. Valduriez, "Dhtjoin: Processing continuous join queries using dht networks," *DPD*, 2009.
[20] R. Mueller, J. Teubner, and G. Alonso, "Streams on Wires - A Query Compiler for FPGAs," in *VLDB*, 2009.