Open Streaming Operation Patterns

Qiming Chen and Meichun Hsu

HP Labs Palo Alto, California, USA
Hewlett Packard Co.
{qiming.chen,meichun.hsu}@hp.com

Abstract. We describe our *canonical dataflow operator framework* for distributed stream analytics. This framework is characterized by the notion of *open-executors*. A dataflow process is composed by chained operators which form a graph-structured topology, with each logical operator executed by multiple physical instances running in parallel over distributed server nodes. An open executor supports the streaming operations with specific characteristics and running pattern, but is *open* for the application logic to be plugged-in. This framework allows us to provide automated and systematic support for executing, parallelizing and granulizing the continuous operations.

We illustrate the power of this approach by solving the following problems: first, how to categorize the meta-properties of stream operators such as the I/O, blocking, data grouping characteristics, for providing unified and automated system support; next, how to elastically and correctly parallelize a stateful operator that is history-sensitive, relying on the prior state and data processing results; how to analyze unbounded stream granularly to ensure sound semantics (e.g. aggregation); and further, how to deal with parallel sliding window based stream processing systematically. These capabilities are not systematically supported in the current generation of stream processing systems, but left to user programs which can result in fragile code, disappointing performance and incorrect results. Instead, solving these problems using open-executors benefits many applications with system guaranteed semantics and reliability.

In general, with the proposed canonical dataflow operator framework we can *standardize* the operator execution patterns, and to support these patterns systematically and automatically. The value of our approach in real-time, continuous, elastic data-parallel and topological stream analytics has been revealed by the experiment results.

1 Introduction

Real-time stream analytics has increasingly gained popularity since enterprises need to capture and update business information just-in-time, analyze continuously generated "moving data" from sensors, mobile devices, social media of all types, and gain live business intelligence.

We have built a stream analytics platform with code name ***Fontainebleau*** for dealing with *continuous, real-time* data-flow with *graph-structured* topology. This platform is *parallel* and *distributed* with each logical operator executed by multiple

A. Hameurlain et al. (Eds.): TLDKS XII, LNCS 8320, pp. 83–104, 2013.

physical instances running in parallel over distributed server nodes. The stream analysis operators are defined by users flexibly. From stream abstraction point of view, our stream analytics cluster is positioned in the same space of System S(IBM), Dryad(MS), Storm(Tweeter), etc. However, this work aims to advance the state of art by providing canonical execution support for stream analysis operators.

1.1 The Challenges

A stream analytics process is composed by multiple operators and pipes connecting these operators. The operators for stream analysis have certain meta-properties representing their I/O characteristics, blocking characteristics, data grouping characteristics, etc, as well as the functionalities common to various types of applications, which can be categorized for introducing unified system support. Categorizing stream operators and their running patterns to provide automatic support accordingly, can ensure the operators to be executed optimally and consistently, as well as ease user's effort for dealing with these properties manually which is often tedious and risky. Unfortunately, this issue has been missed by the existing stream processing systems.

There exist several key requirements in stream processing which demand automated and systematic support. First, to scale out, the data-parallel execution of operators must be taken into account, where how to ensure the correctness of data-parallelism is the key issue which requires the appropriate system protocol to guarantee; particularly in parallelizing stateful stream operators where the stream data partitioning and data buffering must be consistent. Next, stream processing is often made in granule. For example, to provide sound aggregation semantics (e.g. sum), the infinite input data stream must be processed chunk by chunk where each operator may punctuate data based on different chunking criteria such as in 1-minute or 1-hour time windows (certain constraints apply, e.g. the frame of a downstream operator must be the same as, or some integral number of, the frame of its upstream operator). Granulizing dataflow analytics represents another kind of common behavior of stream operators which also need to be supported systematically.

Current large-scale data processing tools, such as Map-Reduce, Dryad, Storm, etc, do not address these issues in a canonical way. As a result, the programmers have to deal with them on their own, which can lead to fragile code, disappointing performance and incorrect results.

1.2 The Proposed Solution

The operators on a parallel and distributed dataflow infrastructure are performed by both the infrastructure and the user programs, which we refer to as their **template behavior** and **dynamic behavior**. The template behavior of a stream operator depends on its meta-properties and its running pattern. For example, a map-reduce application is performed by the Hadoop infrastructure as well as the user-coded map

function and reduce function. Our streaming platform is more flexible and elastic than Hadoop in handling dynamically parallelized operations in a general graph structured dataflow topology, and our focus is placed on supporting the template behavior, or operation patterns, **automatically** and **systematically**.

Unlike applying an operator to data, stream processing is characterized by the flowing of data through a *stationed* operator. We introduce the notion of **open-station** as the container of a stream operator. The stream operators with certain common meta-properties can be executed by the class of open-stations specific to these operators. Open-stations are classified into a station hierarchy. Each class provides an **open-executor** as well as related system utilities. In the OO programming context, the open-executor is coded by invoking certain abstract functions (methods) to be implemented by users based on their application logic. In this way the station provides designated system support, while *open* for the application logic to be plugged-in. In this work we use the proposed architecture to solve several typical stream processing problems.

The key to ensure safe parallelization is to handle data flow group-wise - for each vertex representing a logical operator in the dataflow graph; the operation parallelization with multiple instances comes with input data partition (grouping) which is consistent with the data buffering at each operation instance. This ensures that in the presence of multiple execution instances of an operator, *O*, every stream tuple is processed *once and only once* by one of the execution instances of *O*; the historical data processing states of every group of the partitioned data are buffered with *one and only one execution instance* of *O*. Our solution to this problem is based on the open station architecture.

The key to ensure the granule semantics is to handle dataflow chunk wise by punctuating and buffering data consistently. Our solution to this problem is also based on the open station architecture.

As a generalization of these solutions, we show how to use the open station architecture to provide system support for handling parallel sliding window based stream processing.

In general, the proposed canonical operation framework allows us to *standardize* various operational patterns of stream operators, and have these patterns supported systematically and automatically. Our experience shows its power in real-time, continuous, elastic data-parallel and topological stream analytics.

The rest of this paper is organized as follows: section 2 describes the notions of open-station and open-executor; then based on these notions section 3 discusses how to guarantee the correctness of data-parallel execution of stateful operations, and how to deal with the granular execution of stream operations; in section 4 we further show how to use the open station architecture to provide system support for handling parallel sliding window based stream processing; some experimental results are illustrated in section 5; finally section 6 compares with related work and concludes the paper.

2 Open Station and Open Executor of Stream Operator

2.1 Continuous, Parallel and Elastic Stream Analytics Platform

Fontainebleau is a real-time, continuous, parallel and elastic stream analytics platform. There are two kinds of nodes on the cluster: the *coordinator node* and the *agent nodes* with each running a corresponding daemon. A dataflow process is handled by the coordinator and the agents spread across multiple machine nodes. The coordinator is responsible for distributing code around the cluster, assigning tasks to machines, and monitoring for failures, in the way similar to Hadoop's job-tracker. Each agent interacts with the coordinator and executes some operator instances (as threads) of the dataflow process. The *Fontainebleau* platform is built using several open-source tools, including ZooKeeper[12], ØMQ[11], Kryo[13], Storm[14], etc. ZooKeeper coordinates distributed applications on multiple nodes elastically. ØMQ supports efficient and reliable messaging. Kryo deals with object serialization. Storm provides the basic dataflow topology support.

A stream is an unbounded sequence of tuples. A stream operator transforms a stream into a new stream based on its application-specific logic. The stream transformations are packaged into a graph-structured "topology" which is the top-level dataflow process. When an operator emits a tuple to a stream, it sends the tuple to every successor operators subscribing to that stream. A stream grouping specifies how to group and partition the tuples input to an operator. There exist a few different kinds of stream groupings such as hash-partition, replication, random-partition, etc.

To support elastic parallelism, we allow a logical operator to be executed by multiple physical instances, as threads, in parallel across the cluster; they pass messages to each other in a distributed way. Using the ØMQ library [11], message delivery is reliable; messages never pass through any sort of central router, and there are no intermediate queues.

To provide an overview, we use a simplified as well as extended Linear-Road (LR) benchmark to illustrate the notion of stream process. The LR benchmark models the traffic on 10 express ways; each express way has two directions and 100 segments. Cars may enter and exit any segment. The position of each car is read every 30 seconds and each reading constitutes an event, or stream element, for the system. A car position report has attributes *vehicle_id*, *time* (in seconds), *speed* (mph), *xway* (express way), *dir* (direction), *seg* (segment), etc. With the simplified benchmark, the traffic statistics for each highway segment, i.e. the number of active cars, their average speed per minute, and the past 5-minute moving average of vehicle speed, are computed. Based on these per-minute per-segment statistics, the application computes the tolls to be charged to a vehicle entering a segment any time during the next minute. As an extension to the LR application, the traffic statuses analyzed and reported every hour. The logical stream process for this example is given in Fig. 1.

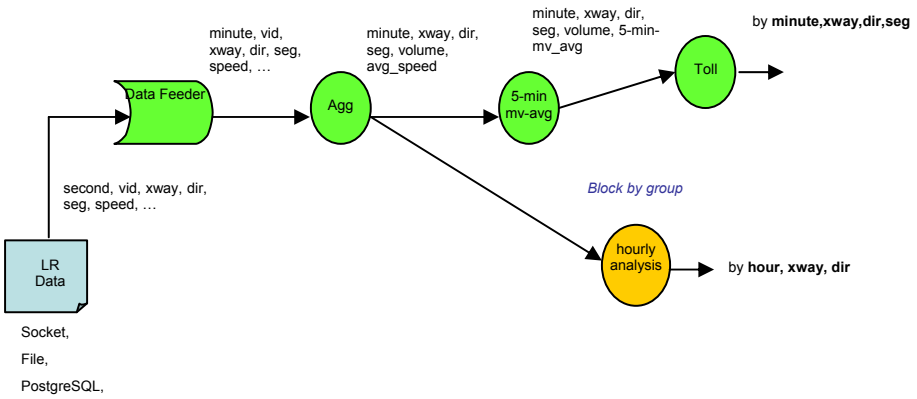

Fig. 1. The extended LR logical dataflow process with operators linked in a topology

This stream analytics process is specified by the Java program illustrated below.

```
public class LR_Process {
…
public static void main(String[] args) throws Exception {
        ProcessBuilder builder = new ProcessBuilder();
        builder.setFeederStation(“feeder”, new LR_Feeder(args[0]), 1);
        builder.setStation("agg", new LR_AggStation(0, 1), 6) .hashPartition(“feeder”,
          new Fields("xway", "dir", "seg"));
        builder.setStation("mv", new LR_MvWindowStation(5), 4).hashPartition("agg",
          new Fields("xway", "dir", "seg"));
        builder.setStation("toll", new LR_TollStation(), 4).hashPartition("mv",
          new Fields("xway", "dir", "seg"));
        builder.setStation("hourly", new LR_BlockStation(0, 7), 2).hashPartition("agg",
          new Fields("xway", "dir"));
        Process process = builder.createProcess();
        Config conf = new Config();   conf.setXXX(…);  …
        Cluster cluster = new Cluster();
        cluster.launchProcess("linear-road", conf, process);
        …
}
```

In the above topology specification, the hints for parallelization are given to the operators “agg” (6 instances), “mv” (5 instances), “toll” (4 instances) and “hourly” (2 instances), the platform may make adjustment based on the resource availability. Then the physical instances of these operators for data-parallel execution are illustrated in Fig 2.

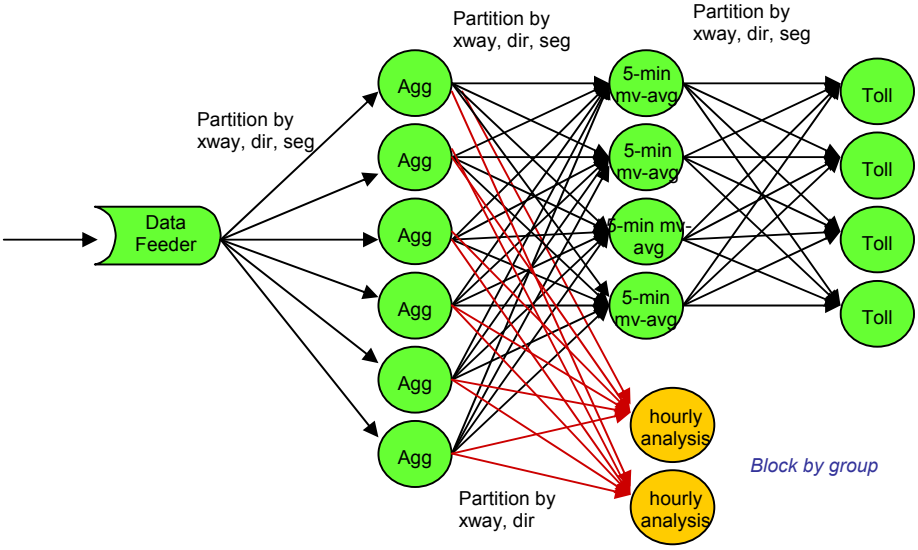

Fig. 2. The LR dataflow process instance with elastically parallelized operator instances

2.2 Meta Characteristics of Operators

Stream operators have certain characteristics in several dimensions, such as the provisioning of initial data, the granularity of event processing, memory context, invocation patterns, results grouping and shuffling, etc, which may be considered as the meta-data, or the design pattern of operators. Further, the operators for supporting a kind of applications also have certain common characteristics. Below we briefly list some characteristics.

I/O Characteristics specifies the number of input tuples and the output tuples the stream operator is designed to handle the stream data *chunk-wise*. Examples are 1:1 (one input/one output), 1:N (one input/multiple outputs), M:1(multiple inputs/ one output) and M:N (multiple inputs/ multiple outputs). Accordingly we can classify the operators into Scalar (1:1); Table Valued (TV) (1:N); Aggregate (N:1), etc, for each chunk of the input. Currently we support the following chunking criteria for punctuating the input tuples: (a) by cardinality, i.e. number of tuples; and (b) by granule as a function applied to an attribute value, e.g. *get_minute* (timestamp in second).

Blocking Characteristics tells that in the multiple input case, the operator applies to the input tuple one by one incrementally (e.g. per-chunk aggregation), or first pools the input tuples and then apply the function to all the pooled tuples. Accordingly the block mode can be *per-tuple* or *blocking*. Specifying the blocking characteristics tells the system to invoke the operator in the designated way, and save the user's effort to handle them in the application program.

Caching Characteristics is related to the 4 levels potential cache states:

- per-process state that covers the whole dataflow process with certain initial data objects;
- Per-chunk state that covers the processing of a chunk of input tuples with certain initial data objects;
- Per-input state that covers the processing of an input tuple possibly with certain initial data objects for multiple returns;
- Per-return state that covers the processing of a returned tuple.

Grouping Characteristics tells a topology how to send tuples between two operators. There's a few different kinds of stream groupings. The simplest kind of grouping is called a "random grouping" which sends the tuple to a random task. It has the effect of evenly distributing the work of processing the tuples across all of the consecutive downstream tasks. The hash grouping is to ensure the tuples with the same value of a given field go to the same task. Hash groupings are implemented using consistent hashing. There are a few other kinds of groupings.

Function Characteristics underlies the common feature of a kind of stream processing applications. Support those features systematically can ease the effort and improve the quality of application development.

2.3 Stationed Streaming Operators

Ensuring the characteristics of stream operators by user programs is often tedious and not system guaranteed. Instead, *categorizing the common classes of operation characteristics and supporting them automatically and systematically* can simplify user's effort and enhance the quality of streaming application development. This has motivated us to introduce **open-stations** for holding stream operators and encapsulating their characteristics – towards the **open station class hierarchy** (Fig 3).

Each open-station class is provided with an "open executor" as well as related system utilities for executing the corresponding kind of operators; that "open" executor invokes the abstract methods, which are defined in the station class but implemented by users with the application logic. In this way a station provides the designated system support, while open for the application logic to be plugged-in. A user defined operator captures the designated characteristics by subclass the appropriate station, and captures the application logic by implementing the abstract methods accordingly.

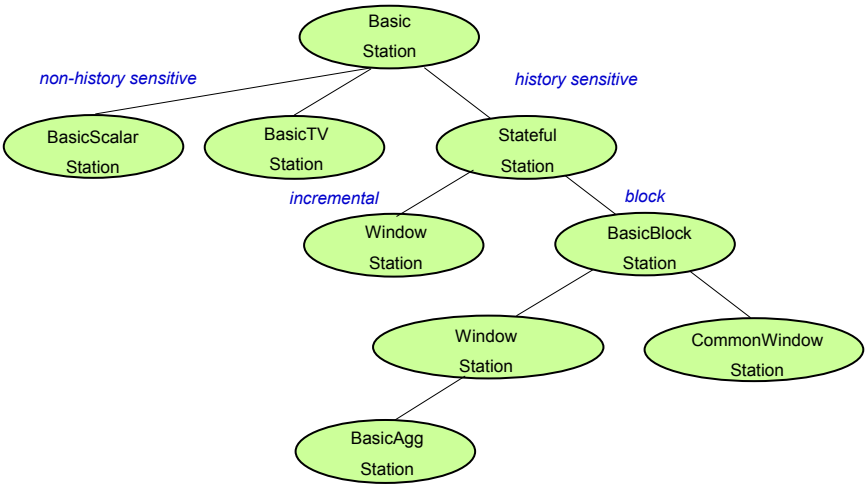

Fig. 3. Station Hierarchy Example

2.4 Open Executor

Specified in a station class, there are two kinds of pre-prepared methods: the system defined ones and the user defined ones.

- The system defined methods include the **open-executor** and other utilities which is open to plugging-in application logic, in the sense that they invoke the abstract methods to be implemented by users according to the application logic.
- The abstract methods to be implemented by the user based on the application logic.

For example, the WindowStation that extends BasicBlockStation, is used to support chunk-wise stream processing, where the framework provided functions, hidden from user programs, include

```
public boolean nextChunk(Tuple, tuple) {// group specific …}
public void execute(Tuple tuple, BasicOutputCollector collector) {
    boolean new_chunk = nextChunk(tuple);
    String grp = getGroupKey(tuple);
    GroupMeasures gm = null;
    if (new_chunk) {
       gm = getGKV().dump(grp);
    }
    updateState(getGKV(), tuple, grp);
    if (new_chunk) { //emit last chunk
       processChunkByGroup(gm, collector);
    }
}
```

The three functions marked Station are to be implemented based on the application logic; the others are system defined for encapsulating the chunk-wise stream processing semantics.

In addition to offering the dataflow operation "executor" abstraction, introducing open station also aims to provide canonical mechanisms to **parallelize stateful and granule dataflow process**. The core is to handle data flow chunk-wise and group-wise - for each vertex representing a logical operator in the dataflow graph; the operation parallelization (launching multiple instances) comes with input data partition (grouping) which is consistent with the data buffering at each operation instance. These are discussed in the following sections.

3 Support Parallelized and Granulized Stream Processing Patterns

3.1 Data-Parallel Execution of Operators

Under our approach, logically, the dataflow elements, i.e. tuples, either originated from a data-source or derived by a logical operator, say *A*, are sent to one or more receiving logical operators, say *B*. Since each logical operator may have multiple execution instances, the dataflows from *A* to *B* actually form multi-to-multi messaging channels.

To handle data-parallel operations, an operator property: parallelism hint, can be specified, that is the number (default to 1) of station threads for running the operator. The number of actual threads will be judged by the infrastructure and load-balanced over multiple available machines.

For the sake of correct parallelism, the stream from *A*'s instances to *B*'s instances are sent in a *partitioned* way (e.g. hash-partition) such that the data sent from any instance of *A* to the instances of *B* are partitioned in the same way. This is similar to the data shuffling from a Map node to a Reduce node, but in more general dataflow topology.

Although our platform offers the flexibility of dataflow grouping with options hash-partition, random-partition, range-partition, replicate, etc, the platform enforces the use of hash partition for the parallelized operators. In case an operator is specified to have parallel instances in the user's dataflow process specification, the input stream to that operator must be defined as hash-partitioned; otherwise the process specification would be invalidated.

Further, there can be multiple logical operators, B_1, B_2, …, B_n, for receiving the output stream of *A*, but each with different data partition criterion, called *inflow-grouping-attributes* (a la SQL group by). The tuples falling in the same partition, i.e. grouped together, have the same "***inflow-group-keys***". For example, the tuples representing the traffic status of an express way (*xway*), direction (*dir*) and segment (*seg*), are partitioned, thus grouped by attributes <xway, dir, seg>; tuples of each group has the same inflow-group-key derived from the values of xway, dir and seg. An operation instance may receive multiple groups of data. The abstract method, ***getGroupKey***(tuple), must be implemented, which is invoked by the corresponding open-executor.

3.2 Parallelize Stateful Streaming Operators Group-Wise

A stateful operator caches its state for future computation, and therefore is history sensitive. When a logical stateful operator has multiple instances, their input data must be partitioned, and the data partition must be consistent with the data buffering.

For example, given the logical operation, *O*, for calculating moving-average and with the input stream data partitioned by <xway, dir, seg>, the data buffers of its execution instances are also partitioned by <xway, dir, seg>, which is prepared and enforced by the system.

For history-sensitive data-parallel computation, an operation instance keeps a state computed from its input tuples (other static states may be incorporated but not the focus of this discussion). We generally provide this state as a KV store where keys, referred to as *cachhing-group-keys*, are Objects (e.g. String) extracted from the input tuples, and values are Objects derived from the past and present tuples such as numerical objects (e.g. sum, count), list objects (certain values derived from each tuple), etc. the multiple instances of a logical operation can run in data-parallel provided that the inflow-group-keys are used as the caching group-keys. In this sense we refer to the KV store as Group-wise KV store (GKV). APIs for accessing the GKV are provided as well. As illustrated in the last section, an important abstract method, `updateState()`, is defined and to be implemented by users.

With the above mechanisms, in the presence of multiple execution instances of an operator, every stream tuple is processed **once and only once** by one of the execution instances; the data processing states of every group of the partitioned input data (e.g. the tuples belonging to the same segment of the an express-way in a direction) are buffered in the function closure of **one and only one execution instance** of that operator. These properties are common to a class of tasks thus we support them in the corresponding station class, that, substantially, is subclassifiable.

3.3 Window Based Stream Analytics

Although a data stream is unbounded, very often applications require those infinite data to be analyzed granularly. Particularly, when the stream operation involves the aggregation of multiple events, for semantic reason the input data must be punctuated into bounded chunks. This has motivated us to execute such operation *window by window* to process the stream data *chunk by chunk*.

For example, in the previous car traffic example, the operation "agg" aims to deliver the average speed in each express-way's segment per minute. Then the execution of this operation on an infinite stream is made in a sequence of *windows*, one on each stream chunks. To allow this operation to apply to the stream data one chunk at a time, and to return a sequence of chunk-wise aggregation results, the input stream, is cut into 1 minute (60 seconds) based chunks, say S_0, S_1, …S_i, … such that the execution semantics of "agg" is defined as a sequence of one-time aggregate operation on the data stream input minute by minute.

In general, given an operator, *O*, over an infinite stream of relation tuples *S* with a criterion ϑ for cutting *S* into an unbounded sequence of chunks, e.g. by every

1-minute time window, $<S_0, S_1, \ldots, S_i, \ldots>$ where S_i denotes the *i-th* "chunk" of the stream according to the chunking-criterion ϑ. The semantics of applying O to the unbounded stream S lies in

$$Q\,(S) \rightarrow < Q\,(S_0), \ldots\ Q\,(S_i), \ldots >$$

which continuously generates an unbounded sequence of results, one on each *chunk* of the stream data.

Punctuating input stream into chunks and applying operation *window by window* to process the stream data *chunk by chunk*, is a template behavior common to many stream operations, thus we consider it as a kind of meta-property of a class of stream operations and support it automatically and systematically by our operation framework. In general, we host such operations on the **window station** (or the ones subclassing it) and provide system support in the following aspects (please refer to the window station example given previously).

- A window station hosts a stateful operation that is data-parallelizable, and therefore the input stream must be hash-partitioned which is consistent with the buffering of data chunks as described in the last section.
- Several types of stream punctuation criteria are specifiable, including punctuation by cardinality, by time-stamps and by system-time period, which are covered by the system function

  public boolean nextChunk(Tuple, tuple)

  to determine whether the current tuple belongs to the next chunk or not.
- If the current tuple belongs to the new chunk, the present data chunk is dumped from the chunk buffer for aggregation/group-by in terms of the user-implemented abstract method *processChunkByGroup*().
- Every input tuple (or derivation) is buffered, either into the present or the new chunk.

By specifying additional meta properties and by subclassing the window station, more concrete system support can be introduced. For example, an aggregate of a chunk of stream data can be made once by end of the chunk, or tupe-wise incrementally. In the latter case an abstract method for per-tuple updating the partial aggregate is provided and implemented by the user.

The *paces of dataflow* wrt timestamps can be different at different operators; for instance, the "agg" operator is applied to the input data minute by minute, so are some downstream operators of it; however the "hourly analysis" operator is applied to the input stream minute by minute, but generates output stream elements hour by hour.

The combination of group-wise and chunk-wise stream analytics provides a generalized abstraction for parallelizing and granulizing the continuous and incremental dataflow analytics.

4 Support Parallel Sliding Window Stream Processing Patterns

In this section we extend our discussion to Parallel Sliding Window (PSW) based stream analysis and illustrate the benefits of open stations in dealing with PSW. PSW based stream processing has certain meta-properties in punctuating and grouping input data, in retaining and shifting intermediate results, and in synchronizing parallel chunking. Generalizing and categorizing these operators and their running patterns allows us to provide automatic support accordingly, to ensure the operators to be executed optimally and consistently, as well as ease user's effort for dealing with them.

We build abstract stations to support the common PSW related features such as handling punctuation and parallelism where having the application specific semantics left as abstract methods for users to implement. The abstract stations form a hierarchy; they provide the mechanisms for managing the data granules, the slide and window boundaries, for punctuating input data stream for switching slides and windows, as well as for retaining and intermediate results. In general, they provide system support for synchronizing the slide and window switching wrt multiple, parallel input streams.

4.1 Sliding Window Based Stream Analytics

In stream processing, the tuples transferred between tasks can be granule based on timestamps or so; and we introduce three levels of boundaries for grouping data in the context of PSW: ***granule***, ***slide*** and ***window***; a granule is the basic unit for grouping data, it could be, for example, a chunk of N tuples or the tuples with timestamps falling in one minute; a slide is defined as a given number or range of granules, for example a slide of 10 minute is composed by 10 one-minute granules; a window is also defined as a given number or range of granules, but the size of a window is at least the size of a slide.

A sliding window based operation keeps the following variables for dealing with sliding window semantics.

- ***window_size*** – the number of *granules* per window;
- ***slide_size,*** *or* ***delta*** - the number of *granules* per slide;
- ***current*** – the current granule number;
- ***ceiling*** – the ceiling of the current slide by granule number, after the window_size is reached, it is the ceiling of current window;
- ***window*** – the number (ID) of the current window.

As usual, for each operation we provide two major system abstract methods***:*** ***initialize()*** and ***execute().*** The *initialize()* method is invoked before the per-tuple processing for instantiating the settings and gathering the topology information, e.g. the input channels. The *execute()* method is invoked upon receipt each input tuple which provides the functions of processing a tuple, or, if a slide or window boundary is reached, calculating the slide or window based summaries and outputing the data mining results.

For example, a sliding window based stream analytics operation with window based summarization but without slide based stepwise summarization, has the following operation logic.

```
1.      current = resolveGranule(tuple);
2.      if (current >= ceiling) {
3.         if (current >= window_size) {
4.            summarize_window();
5.            window++;
6.         }
7.         ceiling = (current/delta + 1)*delta;
8.         process_held_tuples(ceiling);
9.      }
10.   if (getGranule(tuple) >= ceiling) {
11.      held_tuples.add(tuple);
12.   } else {
13.      process_tuple(tuple);
14.   }
```

The above logic can be described as below.

1. resolve the least granule# from all input channels;
2. if the next slide has been reached (this tuple belong to the next slide according to its granule#);
3. if the window boundary also reached (the usual case after completing the first window, if the sliding is defined as the shift of one slide);
4. make window based data mining and output the results;
5. advance the window#;
6. .
7. update the ceiling to the upper bound of the next slide;
8. process the held tuples falling in the next slide;
9. .
10. if the overall slide operation does not advance even if this tuple is beyond the boundary of the current slide
11. hold this tuple
12. .
13. otherwise
14. process this tuple

In general, upon receipt of a tuple, the system first resolve the current granule by taking into account all input channels; if the input tuple belongs to the current slide or window it gets processed, otherwise it is held to be processed in the next or even further windows where it fits in.

The window based data mining takes place at the boundary of two consecutive windows. Since we deal with sliding window, the partial results must be retained and shift – i.e. sliding.

4.2 Parallelize Sliding Window Based Stream Analytic Operation

When a sliding window based stream analytics task has multiple parallel input channels, their punctuation must be synchronized. For example, assume a task *T* has 4 input channels and currently working in window #3, after *T* receives a tuple belonging to window #4 it may or may not be able to "conclude" window #3 depending on whether all the input channels have started to supply tuples belonging to window #4 or beyond; if not, concluding window #3 would yield inaccurate result.

We assume for each input channel the tuples are delivered in the order of granules. The granule boundary of data processing by the current task is determined by taking into account all the input channels based on the following mechanism.

- The current granule number of each input channel if maintained in the ***granuleTable***.
- Upon receipt a new input, the ***granuleTable*** is updated, and the current granule number is resolved as the minimal granule number of all input channels.
- If the granule number of the current input is larger than the resolved one, this tuple is to be held without processing; it will be processed later in the next or a future window instead.
- Once a window boundary is reached by referring to all the input channels, the data analytics results for the current window is generated and finalized, and the data analytics process enters the next window boundary, starting with processing those held tuples that fall into the new window boundary; the tuples falling in future windows will continue being held.

4.3 The Generalized Framework

We provide generalized algorithms to support PSW based incremental stream analysis performed on the per-granule, per-slide and per-window basis. We coded these algorithms as open-executors held by open-stations.

The Top Level Abstract Station. This station provides the generalized algorithm for sliding window based incremental stream analysis which covers he per-granule, per-slide and per-window based incremental stream processing. The flowchart is shown in Fig 4, where several abstract methods are provided which are to be implemented by user based on their application logic.

```
public void execute(Tuple tuple) {
   //resolve the granule the task is working on
long resolved = resolveGranule(tuple);
// If granule is advanced,
//   summarize the current granule
//   sliding the list of partial results and
//   process the held tuples in the next granule boundary
   if (this.scope == SumScope.GRANULE) {
      if (resolved > current) {
         partialResult = partial_summarize();
```

```
            this.partialResultList.add(partialResult);
            if (partialResultList.size() > window_size) {
               this.partialResultList.removeFirst();
            }
            process_held_tuples(resolved+1);
         }
      }
      current = resolved;

      // if slide is advanced
//    summarize the current slide
//    if window is advanced, get window summarization results
//    sliding the list of partial results and
//    process the held tuples in the next granule boundary
      if (current >= ceiling) {
         if (this.scope == SumScope.SLIDE) {
            partialResult = partial_summarize();
            this.partialResultList.add(partialResult);
            if (partialResultList.size() > window_size / delta) {
               this.partialResultList.removeFirst();
            }
         }
         if (current >= window_size) {
            summarize_window();
            window++;
         }
         //handle next slide
         ceiling = (current/delta + 1)*delta;
         if (this.scope == SumScope.WINDOW || this.scope == SumScope.SLIDE) {
            process_held_tuples(ceiling);
         }
}
// if tuple falls in the current scope, process it, if beyond the current scope, hold it
      long upper = (this.scope == SumScope.GRANULE)? current:ceiling;
         if (getGranule(tuple) >= upper) {
            held_tuples.add(tuple);
         else {
            //normal call
            process_tuple(tuple);
         }
}
public abstract long getGranule(Tuple tuple);
public abstract void summarize_window();
public abstract Object partial_summarize();
public abstract void process_tuple(Tuple tuple);
```

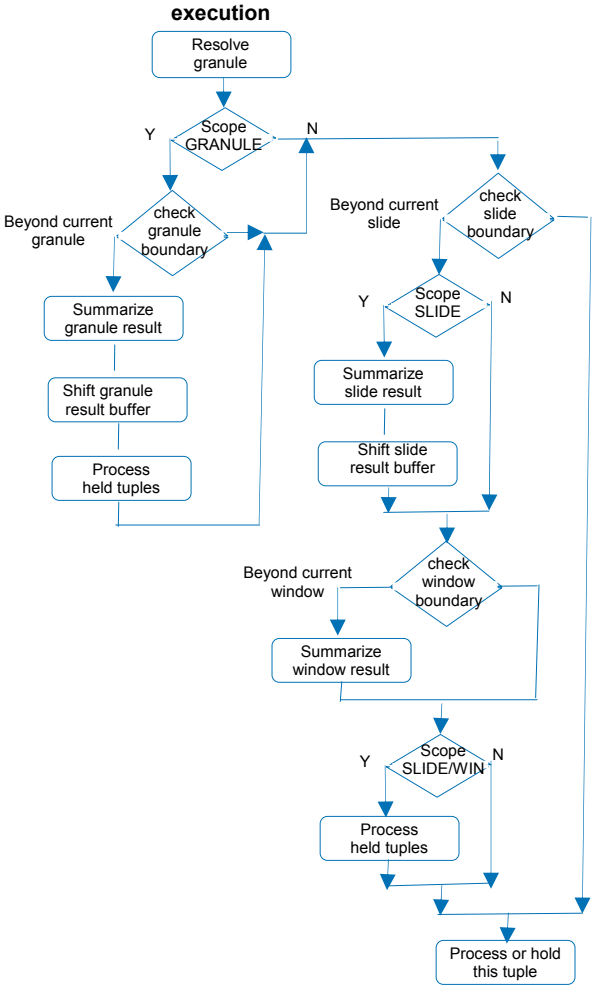

Fig. 4. Generalized PSW Framework

Abstract Station Supporting Window Oriented Summarization. This abstract station subclass the above generalized abstract PSW station; it provides the abstract algorithm with window oriented summarization as illustrated in Fig 5. It subclasses the above generalized PSW station simply by making the unrelated functions (those not highlighted) as dummy functions. In fact, this abstract station class is implemented by only one method:

```
public Object partial_summarize() {
        return null;
}
```

The rest methods are inherited.

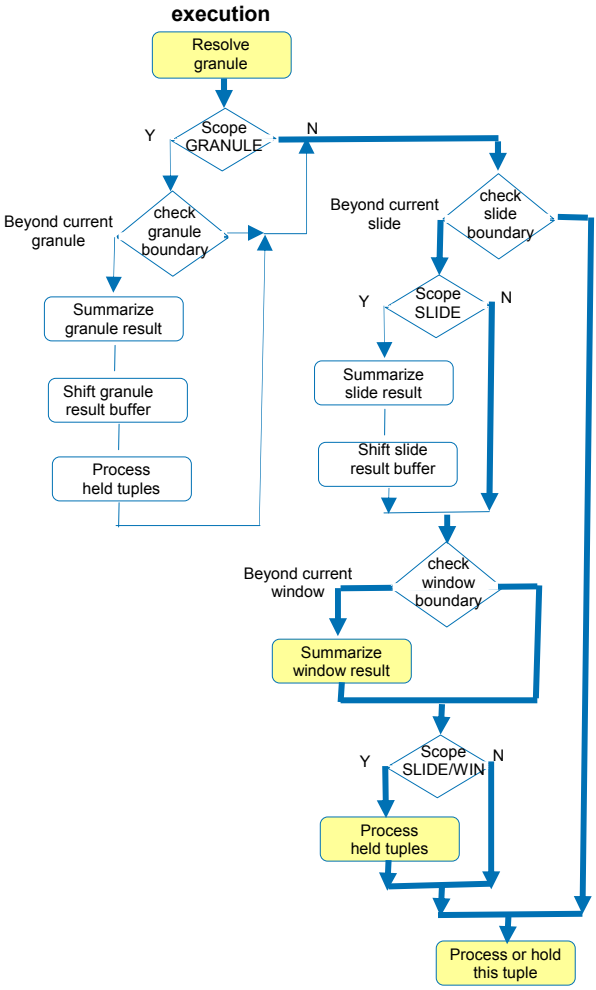

Fig. 5. Abstract Execution Framework for Window Oriented Summarization

Abstract Station Supporting Slide Oriented Summarization. The flow-chat for the abstract algorithm with slide oriented summarization is illustrated in Fig 6. The station supporting the corresponding execution pattern subclasses the above generalized PSW station simply by making the unrelated functions (those not highlighted) as dummy functions.

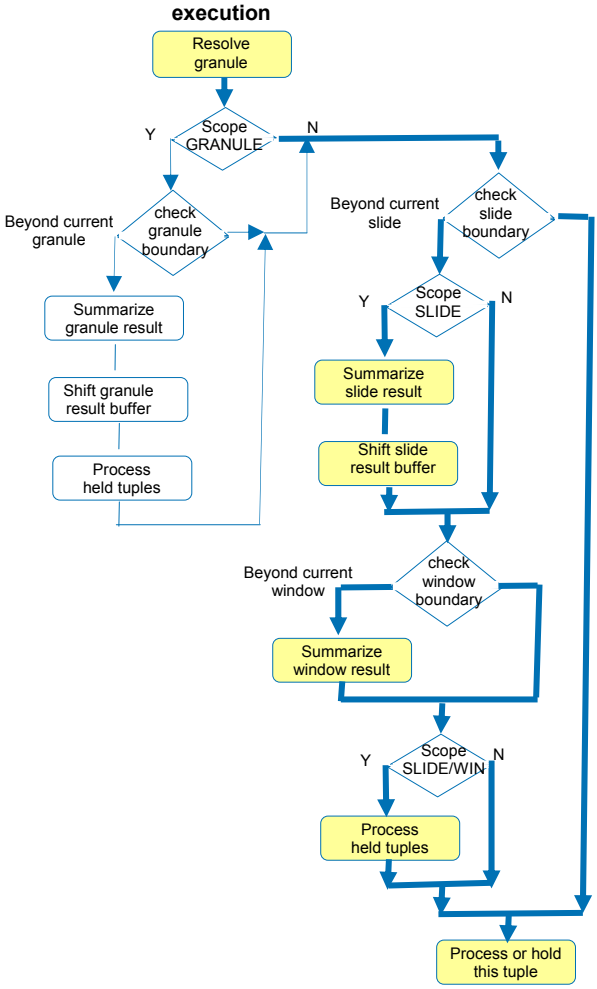

Fig. 6. Abstract Execution Framework for Slide Oriented Summarization

Abstract Station Supporting Granule Oriented Summarization. The flow-chat for the abstract algorithm with granule oriented summarization is illustrated in Fig. 7. The station supporting the corresponding operation pattern subclasses the above generalized PSW station simply by making the unrelated functions (those not highlighted) as dummy functions.

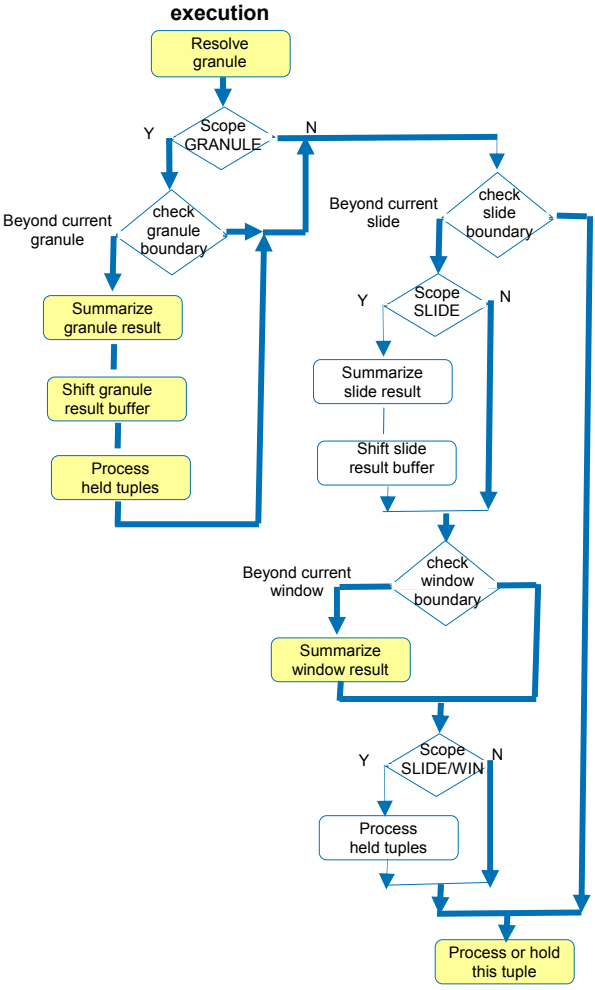

Fig. 7. Abstract Execution Framework for Granule Oriented Summarization

4.4 A PSW Stream Process Topology Example

Below we show a stream processing topology example for frequent pattern mining, we do not discuss the application here, only illustrate the role of parallel sliding window operation oriented stations in a stream analytics dataflow process. The topology specification is illustrated below

```
TopologyBuilder builder=new TopologyBuilder();
builder.setSpout("spout",
        new FileChunkItemsetSpout(filename, chunk_size));
builder.setStation("pre",
```

```
        new FPItemsetSortStation(ItemSequence), N).shuffleGrouping("spout");
    builder.setStation("mining",
        new FPSlidingWindowStation(window_size, slide_size, chunk_size,
        ItemSequence), N).fieldsGrouping("pre", new Fields("leader"));
    builder.setStation("combine",
        new FPWindowCombineStation(1, threshold), 1)
        .fieldsGrouping("mining", new Fields("itemset"));
    builder.setStation("output",
        new FPPrintStation(), 1)
        .fieldsGrouping("combine", new Fields("itemset"));
```

The process contains the following sequential building blocks (operations) but each of them can have multiple instances, and the data partition between them is defined to make the parallel processing correct. These operations are

- ***spout*:** generates stream tuples with fields "granule", plus other fields.
- ***pre*:** pre-processing the input data on the per-tuple basis, such as filtering or sorting, these tasks are not necessarily sliding window based.
- ***mining*:** the major operator for playing data mining, that is coded using the parallel sliding window framework.
- ***combine*:** combining the output of multiple ***mining*** tasks, that also follows the parallel sliding window framework.
- ***output*:** send out the combined data mining results, these tasks are not necessarily sliding window based as far as their upstream tasks are.

5 Experiments

We have built the Fontainebleau prototype based on architecture and policies explained in the previous sections. In this section we briefly overview our experimental results. Our testing environment include 16 Linux servers with gcc version 4.1.2 20080704 (Red Hat 4.1.2-50), 32G RAM, 400G disk and 8 Quad-Core AMD Opteron Processor 2354 (2200.082 MHz, 512 KB cache). One server holds the coordinator daemon, 15 other servers hold the agent daemons, each agent supervises several worker processes, and each worker process handles one or more task instances. Based on the topology and the parallelism hint of each logical task, one or more instances of that task will be instantiated by the framework to process the data streams.

Below we present the experiment results of running the example topology that is similar to the Linear Road scenario; our topology modifies that scenario but we use the same test data under the stress test mode - the data are read from a file continuously without following the real-time intervals, leading to a fairly high throughput.

The performance show in Fig. 8 is based on the event rate of 1.33 million per minute with approximate 12 million (11,928,635) input events. Most of the tasks have 28 parallel instances except one having 14 parallel instances. There is no load-shedding (dropping events) observed.

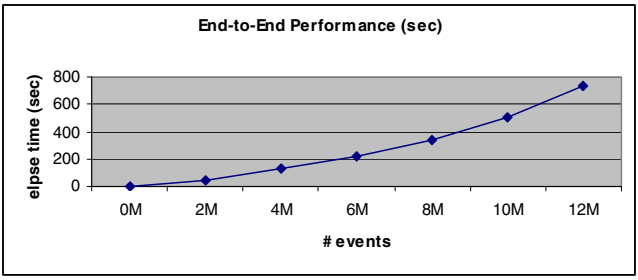

Fig. 8. The performance of data-parallel stream analytics with the LR topology

6 Related Work and Conclusions

In this paper we described our parallel and distributed stream analysis system capable of executing the real-time, continuous streaming process with general graph-structured topology. We focused on the canonical operation framework for standardizing the operational patterns of stream operators, and providing a set of open execution engines for supporting these operational patterns. We examined the power of the proposed framework by supporting the combination of group-wise and chunk-wise stream analytics which provides a generalized abstraction for parallelizing and granulizing continuous dataflow analytics, and further, the generalized support for handling parallel sliding window based stream processing..

Compared with the notable data-intensive computation systems such as DISC [3], Dryad [8], etc, our platform supports more scalable and elastic parallel computation. We share the spirit with Pig Latin [10], etc, in using multiple operations to express complex dataflows. However, unlike Pig Lain, we model the graph structured dataflow by composing multiple operations rather than decomposing a query into multiple operations; our data sources are dynamic data streams rather than static files; we partitioning stream data on the fly dynamically, rather than prepare partitioned files statically to Map-Reduce them. This work also extends the underlying tools such as Storm by elaborating it from a computation infrastructure to a state conscious computation/caching infrastructure, and from the user task oriented system to the execution engine oriented system.

Supporting truly continuous operations distinguish our platform from the current generation of stream processing systems, such as System S (IBM), STREAM (Stanford) [1], Aurora, Borealis[2], TruSQL[9], etc.

Envisaging the importance of standardizing the operational patterns of dataflow operators, we are providing a rich set of open execution engines and linking stations with existing data processors such as DBMS and Hadoop, towards the integrated dataflow cloud service.

References

[1] Arasu, A., Babu, S., Widom, J., The, C.Q.L.: Continuous Query Language: Semantic Foundations and Query Execution. VLDB Journal 15(2) (June 2006)
[2] Abadi, D.J., et al.: The Design of the Borealis Stream Processing Engine. In: CIDR (2005)
[3] Bryant, R.E.: Data-Intensive Supercomputing: The case for DISC. CMU-CS-07-128 (2007)
[4] Chen, Q., Hsu, M., Zeller, H.: Experience in Continuous analytics as a Service (CaaaS). In: EDBT 2011 (2011)
[5] Chen, Q., Hsu, M.: Experience in Extending Query Engine for Continuous Analytics. In: Bach Pedersen, T., Mohania, M.K., Tjoa, A.M. (eds.) DAWAK 2010. LNCS, vol. 6263, pp. 190–202. Springer, Heidelberg (2010)
[6] Chen, Q., Hsu, M.: Continuous mapReduce for in-DB stream analytics. In: Meersman, R., Dillon, T., Herrero, P. (eds.) OTM 2010. LNCS, vol. 6428, pp. 16–34. Springer, Heidelberg (2010)
[7] Dean, J.: Experiences with MapReduce, an abstraction for large-scale computation. In: Int. Conf. on Parallel Architecture and Compilation Techniques. ACM (2006)
[8] Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: Distributed data-parallel programs from sequential building blocks. In: EuroSys 2007 (March 2007)
[9] Franklin, M.J., et al.: Continuous Analytics: Rethinking Query Processing in a Network-Effect World. In: CIDR 2009 (2009)
[10] Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig Latin: A Not-So-Foreign Language for Data Processing. In: ACM SIGMOD 2008 (2008)
[11] ØMQ Lightweight Messaging Kernel, http://www.zeromq.org/
[12] Apache ZooKeeper, http://zookeeper.apache.org/
[13] Kryo - Fast, efficient Java serialization, http://code.google.com/p/kryo/
[14] Twitter's Open Source Storm Finally Hits, http://siliconangle.com/blog/2011/09/20/twitter-storm-finally-hits/