JSQ: Distributed querying of JSON stream data

© Konstantin Abakumov

ISP RAS
abakumov@ispras.ru

Abstract

Nowadays, the necessity for online processing of data is becoming more evident. The most convenient way to perform analytical online processing is declaring continuous queries using special query languages. The goal of this work is to propose the system for distributed continuous query processing on clusters of commodity computers. We studied existing solutions and requirements for such systems and proposed JSQ system desing.

1 Introduction

The common property of most popular, fast-growing and advanced applications today — is a deep-personalized interaction with every user, that requires gathering and generating of big amounts of heterogeneous and often very customized for every user information. Every year thousands of new web- and mobile- services emerging and many of them contain bright and original ideas, that able to lift up our digital experience to the new levels. The examples of such applications is social networks, search engines, collaborative work tools, etc.

At the current moment, important step in evolution of such applications — is to bring their reaction time closer to the realtime, to minimize the delay before provided results will take into account all new gathered information. The simplest example — the delay between the arbitrary web-page update and showing it's new content in search engine's results. It can vary from few weeks to several seconds depending on search engine's indexing resourses and web-page popularity. Another example is detection of trending events in social networks. It can be done once every day by analyzing stored activity, or list of such events can be maintained in near-realtime. Such near-realtime computations are often referred to as online computations.

In the last decade, necessary data amounts and their growth rates forced industry to develop new soultions for efficient data management. As a result, the most popular and perspective approach today — is the distributed scalable computing on clusters of commodity computers. The pioneers of the approach is such technologies as MapReduce, GFS, BigTable and their open-source analogs. But in the majority, these solutions targeted for batch-oriented data processing and not well suited for online computations. Only several years ago there began to emerge solutions, oriented for distributed online computations. The pioneers and most popular of those systems today — frameworks S4 [4] and Storm [5]. They provide simple programming models, allowing easy developement of a scalable online computation programs. Only after the year of public release, Storm got a big popularity and many companies started using it to make their applications more interactive.

Proceedings of the Ninth Spring Researcher's Colloquium on Database and Information Systems, Kazan, Russia, 2013

In this work we will focus more on analytical tasks, and application of online computations for their solution. Such tasks often have one of the following forms:

- Filtering out interesting events and count aggregate values over them in realtime, getting their growth dynamics
- Determining current trends, appearing in system workflow
- Detection of interesting event sequences

Majority of these tasks can be formalized in the terms of following three components:

1. **Event** — some action or state change in the observing system, that seems interesting to us.
2. **Event stream** — sequence of events (possibly infinite), each coming in some time moment.
3. **Continuous query** — persistent query over event streams, that allows to receive new results when they become available.

There exists a big area of stream data processing, in which systems for defining and executing continuous queries are developed. Such systems usually called DSMS (Data-Stream Management System). Continuous queries in them usually defined using special query languages, which are differ in expressivity. In this paper, we will only consider tasks, whose solution can be represented as a continuous query in some query language.

Modern applications, that we discussed in the beginning, also require the solution of tasks that can be represented as continuous queries, mostly for internal usage — counting different sorts of statistics in realtime. More specific examples will appear in the next chapter. Due to the specificity and amounts of data, the execution of continuous queries should be distributed, scalable and, possibly, fault-tolerant. But so far there is a lack of systems for continuous querying of stream data, satisfying these requirements. So the goal of the work is to propose such a system, which will use already existing frameworks for online computations and perform distributed, scalable and fault-tolerant continuous query execution.

There are some choices should be done while developing such a system. The first, is the input event format. It is selected to be JSON — a standard for communication between web-applications. Most of web services today provide their API's responses in JSON format, so they can be routed directly to the system for querying. Moreover, JSON provides simple minimal-overhead structure for complex events. Hence the name of the system is follows — JSQ (JSON Stream Querying). Another important choice — is a selection of continuous query language. It is more compex problem, so we will discuss it in Section 2.

The remainder of the paper organized as follows: In Section 2 requirements for JSQ system and it's query language are described. In Section 3 there are proposed JSQ implementation details are presented. In Section 5 future directions of work are described.

2 JSQ requirements

2.1 Continuous query languages overview

Every considered problem can be presented as continuous query. Therefore set of acceptable tasks depends on query language expressivity. Using the exprience of real stream processing problems and existing systems, we can formalize the set of query language features, that require special syntax constructions. There are several recent works on this topic. This set can be the following:

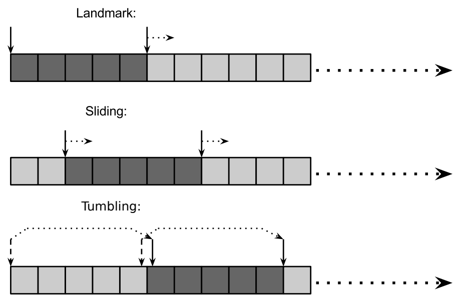
1. **Accessing the event data**. In many systems events are presented and processed just as a fact of some action happening. To distinguish events from each other such systems have support for different event types. But more complex approach is to allow the use of inner event data within queries.

2. **Event filtering**. Events from input stream can be filtered using various predicates on it's inner type or inner data.

3. **New event construction**. The output of continuous query can be a valid event itself — so it can be sent to the input of other continuous query. That allows to express more comlex queries.

4. **Grouping and data aggregation**. Means that we can partition input events using some predicate and count statistics over resulting groups.

5. **Window semantics**. Counting actual statistics over all events in input streams is often very complex problem and not even required in majority of tasks. But what is actually important — is the ability to define "windows" of events to consider. There are two ways how such windows can be parametrized. First — is the way how their length is measured: using last N events, or events during last N time frames (seconds, minutes, etc). The second — is the way how often windows are extracted from stream and processed by the system. There are three ways:



6. **Detection of event patterns**. The other aspect of event analyzing besides statistics on event data — is the detection of interesting event sequences, given as an accepted pattern. This requires entirely different semantics and execution model, but not interfer with another functionality — both aspects can coexist in one system.

7. **Integration with non-event data**. While evaluating predicate or staticstics on event data, it can be useful to employ some external data, for example, from some database.

8. **Joining multiple streams**. Another important feature in some cases — is the ability to handle multiple streams and query some of them simultaneously, using predicates on events, similar to join table operations in relational database management systems.

There were presented the main features that can be expressed with continuous query languages. According to [1], [2], there is no system, that supports all the features described, while every provides only some subset of them and JSQ is not the exception.

2.2 JSQ query language description

Based on described features, we propose query langugage for JSQ system. The general view of continuous query in this language is the following:

```
publish
  <output event description>
  <filter clause>
  <group clause>
  <window clause>
```

Here the query language feature set for the JSQ prototype:

1. **New events construction** The mandatory clause for every continuous query in JSQ — is the definition of resulting events, which are correct JSON events itself. For each field in resulting event the corresponding value expression should be defined. These expressions can contain operations with incoming event data and aggregate expressions results. Possible arithmetic operations — +, -, *, /, div, mod. Now every expression should be defined explicitly, but later it seems reasonably to add let clause for common expression binding. Example:

```
publish
  {
    "field1": <expr1>,
    "field2":
      {
        "innerfield" : <expr2>,
        "arrayfield":
          [<expr3>, <expr4>]
      }
    }
  }
```

2. **Accessing the event data**

   The other important part — is how data in incoming events can be accessed. There is a standard solution — we can nest into JSON structure using field access using dot notation (”field1”.”innerfield”) and access array elements with the use of parenthesis (”arrayfield”[3].”innerfield”). Access to the top-level field is performed without any prefix, it is assumed that top element of current event is in context (to access field ”x” in top-level of event you should simply write ”x”). Such nesting can be used in any expressions in JSQ queries.

3. **Event filtering** Event filtering can performed using the following optional clause:

   ```
   where [not] <filter_expr>
     [and/or [not] <filterexpr> [...]]
   ```

   Filter expressions can contain any nesting into event structure and compare fields with constant values or other fields. It is important how values are treated: JSQ distinguishes only three types of values: boolean, number and strings. Other possible JSON types (object, array and null) are treated as their string representation. On all types main comparison operations are defined: $>, >=, <, <=, ==, != $.

4. **Grouping and data aggregation**

   Events can be partitioned into the groups of interest using the group by clause and an arbitrary partition expression:

   ```
   group by <partition_expr>
   ```

   The partiton expression value can have valid JSON type (including object and array) and the new group is created for every existing value in event stream. For each created group special variable *groupkey* is created, containing the counted partition expression value for the group.

   Moreover, while constructing result events in *publish* clause we can use predefined aggregate functions: count, count-distinct, sum, avg, min, max. Their value will be calculated for each group independently. If there is no *group by* clause provided, it is considered that all events share single group and aggregates are counted over all stream (of course, with respect to the *where* clause).

   There is the special case for min and max functions — it is clear that we often need not only the maximum value in group, but the events where this value is reached. So the values of these function is JSON object themselves — and contain two fields: ”value” is the actual maximum value of partition expression, and ”objects” — contains the array of events, on which this value is reached.

5. **Window semantics** All described in previous subsection windows parameters can be set using the *window* clause:

   ```
   {sliding|tumbling} window
   N (seconds|minutes|hours|events)
   ```

   The first part defines the window application policy and the second — the window building policy (*events* keyword sets the window to combine last N events, all others — events from N last corresponding time frames). By default, when window clause defined, but window application policy is not set — the type of window assumed to be sliding. If there is no *window* clause presented at all, the landmark window is used. For convenience, both only and plural forms of keywords accepted (second/seconds, event/events, etc)

Because JSON is a semi-structured format, every event can have it’s own information set inside. And that is normal case that some operations cannot be performed over particular event (for example, we need access to the ’data’ field, which is not presented) — these events just skipped and not participate in following computations. The output of continuous query is a event stream itself, and events are produced only when the result of the query is changed.

2.3 Scalability requirements

There are some requirements for how system should behave in distributed environment. It should be able to execute continuous queries on clusters of arbitrary computers. This involves requirements for scalability, so adding of additional resources into cluster should cause performance increase. Another important requirement — is fault-tolerance, so the system should be able to continue correct execution of continuous queries under situation of software or hardware fault. So the computations from broken cluster node should be moved to others and missed results should be re-calculated.

2.4 Targeted tasks examples

There presented two example tasks and queries for them, for which JSQ is targeted.

2.4.1 Sale statistics

During the business processes of a company there is a stream of sale events is generated.

And the task is to retrieve some statistics using continuous queries, such as:

Count the average current deal price of in each region (current average price can be the average of all deals in last hour):

```
{
  "product name" : "ford focus",
  "price" : "20000",
  "quantity" : 2,
  "region" : "Asia"
}
```

Listing 1: Sale event example

```
publish
  [groupkey, avg("price" * "quantity")]
  group by "region"
  sliding window 1 hour
```

2.4.2 Content popularity in Twitter

Consider message stream in Twitter, where message is metadata of every tweet.

```
{
  "user" : "user",
  "message" : "Some politic stuff",
  "country": "Great Britain"
}
```

Listing 2: Tweet event example

Count the number of active users in each country in each hour:

```
publish
  {
    "country": groupkey,
    "count": distinct-count("user")
  }
  group by "country"
  tumbling window 1 hour
```

3 JSQ prototype design

As it was said earlier, JSQ is proposed to be built on top of existing framework for distributed online computations. Now there are only two such frameworks — Storm and S4. For this work Storm was selected. Actually, JSQ should be a code generator for execution with Storm framework. So now let's review the principles of programming online computation programs using Storm framework. To create an online computation program, you have to define objects of three types:

1. **Spouts** — subprograms, that create events for the computation. Usually they fetch information from external sources and convert it into internal event format.

   Bolts — subprograms, that perform some part of computations. Each of bolts consumes it's own event stream, and using it, produces new stream, which is routed to the other bolts.

   Topology — description of spouts, bolts, and how events streams are routed between them. Resulting graph called topology. It fully defines the whole computation task.

The whole computation is automatically distributed using Storm, because it automatically distributes copies of bolts to machines in cluster, and each bolt can have many instances on different cluster nodes. Besides simple execution, Storm guarantees fault-tolerant execution, tracking correctness of processing for every particular event. When any node survives a failure, it's subprograms are reassigned to another nodes in cluster.

Given a query, JSQ should build a topology for Storm. Bolts and spouts needed for query execution are implemented as a library. Topology for one query can be chained with other query topologies to form more complex computations. Also, JSQ query topologies can be embedded in usual (manually) Storm topologies, simplifying online processing tasks.

Such an approach was used in Gigascope[3] system and showed it's efficiency in telecommunication tasks.

- Query parsing and execution plan building module
- Topology building module
- Storm executor module

4 Future work

In this work we proposed the desing of JSQ system. The next part of the work is actual development. Also, some new features should be reviewed:

- **TopK operations** — Now only the search of max and min elements is proposed, but more general task is to find the top list of elements. This will require some query language modifications.
- **Web GUI** — JSQ proposed to be used as a standalone system and it is necessary to have a GUI for defining queries and their input streams.
- **Array and string operations** — Many queries depend on deeper string and array data usage than comparing them in lexicographical order — so array iterating operations, string operations can be added.

References

[1] *Survey and Comparison of Event Query Languages Using Practical Examples*. Hai-Lam Bui, Master's thesis, 2009.

[2] *David Luckham. A Brief Overview of the Concepts of CEP*. http://complexevents.com/wp-content/uploads/2008/07/overview-of-concepts-of-cep.pdf, 2008.

[3] *Gigascope: a stream database for network applications*. Chuck Cranor , Theodore Johnson , Oliver Spataschek, ACM SIGMOD, 2003.

[4] *S4: Distributed Stream Computing Platform*. http://s4.io

[5] *Storm. Distributed and fault-tolerant realtime computation*. http://storm-project.net