Large-Scale Network Traffic Monitoring with DBStream, a System for Rolling Big Data Analysis

Arian Bär*, Alessandro Finamore†, Pedro Casas*, Lukasz Golab‡, Marco Mellia†

*FTW Vienna, Austria - email: {baer, casas}@ftw.at
†Politecnico di Torino, Italy - email: {finamore, mellia}@tlc.polito.it
‡University of Waterloo, Canada - email: lgolab@uwaterloo.ca

***Abstract*—The complexity of the Internet has rapidly increased, making it more important and challenging to design scalable network monitoring tools. Network monitoring typically requires *rolling* data analysis, i.e., continuously and incrementally updating (rolling-over) various reports and statistics over high-volume data streams. In this paper, we describe DBStream, which is an SQL-based system that explicitly supports incremental queries for rolling data analysis. We also present a performance comparison of DBStream with a parallel data processing engine (Spark), showing that, in some scenarios, a single DBStream node can outperform a cluster of ten Spark nodes on rolling network monitoring workloads. Although our performance evaluation is based on network monitoring data, our results can be generalized to other big data problems with high volume and velocity.**

***Keywords*-Big Data Analysis; Data Stream Processing; Network Data Analysis; System Performance.**

I. Introduction

The complexity of large-scale, Internet-like networks is constantly increasing. With more services being offered, the massive adoption of Content Delivery Networks (CDNs) and Cloud services for traffic hosting and delivery, and the continuous growth of bandwidth-hungry video-streaming services, network and server infrastructures are becoming extremely difficult to monitor. In particular, the challenge faced by Network Traffic Monitoring and Analysis (NTMA) is to process *big*, *heterogeneous* and *high-speed* data. Network monitoring data are heterogeneous by nature, containing multiple types of measurements coming from different kinds of logging systems. In addition, network monitoring data come in the form of high-speed *streams*, which need to be continuously analyzed. The notion of a data stream used in this paper is that of a continuous flow of measurements coming in the form of short time slices or batches, e.g., all the TCP flows captured in a backbone link in the last minute. These batches can contain a very large number of samples, given the high capacity of network links and the dynamics of Internet traffic.

NTMA and other monitoring applications typically perform what we refer to as *rolling* data analysis: results are periodically and incrementally updated (rolled-over) as new data arrive. In this paper, we describe DBStream, which is a system built upon the PostgreSQL database that explicitly supports incremental queries for rolling data analysis. DBStream, recently introduced in [2], ingests data streams coming in the form of short time-scale aggregated batches (i.e., 1 minute) from a wide variety of sources (e.g., passive network traffic data, active measurements, router logs and alerts, etc.) and performs complex continuous analysis, aggregation and filtering jobs. DBStream can store tens of terabytes of heterogeneous data, and allows both real-time queries on recent data as well as deep analysis of historical data.

The technical contributions of this paper are as follows. First, we present the Continuous Execution Language (CEL), which is a *declarative* SQL-based interface for specifying rolling data analysis in DBStream. CEL allows DBStream users to rapidly implement advanced data analytics which run in parallel and continuously over time using just a few lines of code, accelerating the development of new applications. Second, we compare the performance of DBStream with the popular Spark parallel processing engine using real network traffic data from an operational network. We show that rolling queries can be easily implemented in CEL, and a single DBStream node can, in some scenarios, execute them faster than a cluster of ten Spark nodes.

The remainder of the paper is organized as follows. Sec. II discusses the related work; Sec. III presents the rolling data analysis capabilities of DBStream; Sec. IV compares DBStream with Spark; and Sec. V concludes the paper.

II. Related Work

There has been a great deal of effort to improve the performance and scalability of traditional database management systems by re-implementing the data processing engine, relaxing data consistency constraints and/or applying novel data processing paradigms. Still, a major limitation is the inability to cope with continuous/rolling analytics. Some relational database systems support materialized views, but incremental view maintenance over time is restricted to simple types of queries such as filters and joins, which is not sufficient for monitoring applications. Furthermore, NoSQL systems such as Hadoop [16] have been considered in the context of network monitoring [11], but they are suitable for off-line rather than rolling analytics. However, there has been some recent work

The research leading to these results has received funding from the European Union under the FP7 Grant Agreement n. 318627 (Integrated Project "mPlane".)

on enabling real-time and/or incremental analytics in NoSQL systems, such as Incoop [4], Muppet [10], SCALLA [12] and Spark Streaming [18]. Additionally, Data Stream Management Systems (DSMSs) such as Borealis [1], Gigascope [6] and Streambase [15] support continuous processing, but they usually cannot support analytics over historical data.

Recently, Data Stream Warehouses (DSWs) have been introduced, which extend traditional database systems with (nearly) continuous data ingest and processing. DataCell [13] and DataDepot [8] are two examples, as well as the DBStream system presented in this paper. The novelty of DBStream is that it enables users and applications to declaratively specify, using arbitrary SQL, exactly how to update a view when a new batch of data arrives at the system. These specifications may even refer to previously generated results that are stored in the same view, which, to the best of our knowledge, is not *declaratively* supported by any other system.

Finally we note that there has been recent work in the networking community on extending SQL with additional functionalities required for network monitoring; examples include complex window expressions [5] and sequential patterns [9]. However, none of these proposals include the declarative rolling analytics that DBStream supports.

III. Rolling Analytics in DBStream

DBStream is a rolling analysis system implemented as a Data Stream Warehouse (DSW). Its main purpose is to process and combine data from multiple sources as they are produced, create aggregations, and store query results for further processing by external analysis modules or visualization. The system targets, but is not limited to, continuous network monitoring. For instance, smart grid, intelligent transportation systems, or any other use case that requires continuous processing of large amounts of data over time can take advantage of DBStream.

In this paper, we focus on the following two important features of DBStream:

• It supports incremental queries defined using a declarative interface based on SQL. Incremental queries are those which update their results by combining newly arrived data with previously generated results rather than re-computing them from scratch (see Sec. III-A for more details). This enables efficient processing of two interesting groups of queries. First, aggregated variables can be kept for the elements of the monitored set, e.g., the number of bytes uploaded and downloaded by each client over a sliding window of time. Second, a set of items can be monitored over time by looking at the last state plus the new data, e.g., monitoring the set of all server IP addresses that are accessed within a sliding window of time such as in the last two weeks.

• In contrast to many other systems, DBStream does not change the query processing engine. Instead, queries over data streams are evaluated as repeated invocations of a process that consumes a batch of newly arrived data and combines them with the previous result to come up with the new result. Therefore, DBStream is able to reuse the full functionality of the underlying DBMS, including its query processing engine and query optimizer.

A. Continuous Execution Language (CEL)

In this section we describe the user and application interface to DBStream, based on SQL, to define rolling analytics. We give a high-level overview of CEL using examples from the networking domain. Let us assume we have a stream of data coming from a router. It sends one row per minute and per TCP flow[1] with information about the uploaded and downloaded bytes typical for NetFlow [14] compliant routers. The schema of input data is thus known. We are interested in how many bytes are uploaded and downloaded per hour on that link. In CEL, this can be expressed as the following job:

```
<job inputs="A (window 60min)"
     output="B"
 schema="serial_time int4, total_download int8,
  total_upload int8">
<query>
select _STARTTS, sum(download),
  sum(upload) from A group by _STARTTS
</query></job>
```

The `inputs` attribute defines the input window and the `output` attribute defines the destination for the result. Here, a 60-minute window over A is specified, meaning that for each new hour of data in A, the query specified in the `query` element will run and its results will be appended to table B. DBStream supports all SQL queries that are supported by the underlying DBMS, which is PostgreSQL at the moment. In this example, the query sums up the uploaded and downloaded bytes for each hour. The query includes a "from A" statement, which does not actually read all of A, only the window of A that was specified in the inputs statement (i.e., the most recent 60 minutes). The schema of the output stream B is defined using the `schema` statement, for which the first field must be a timestamp called serial_time. In the above example, the timestamp field is the start time of a window, denoted by `_STARTTS`.

Fig. 1 illustrates the supported window definitions. For each job, one window is defined to be the primary window and is marked with the **primary** keyword. After a job instance is done, the state of the job is shifted by the size of the primary window. As soon as there is a full new primary input window, the next instance of the job is executed. The other important keyword is **delay**, which shifts a window into the past by a given amount of time.

Part A) of Fig. 1 shows the simplest window definition, similar to the previous example. Only a single window exists, which is also the primary window. Therefore, the defined query is executed for every minute of the input stream. In Part B), we have two windows. Every three minutes (the length

[1]Although more complex definitions can be used, here a flow can be identified by the 5-tuple: source IP, destination IP, source port destination port and IP protocol.

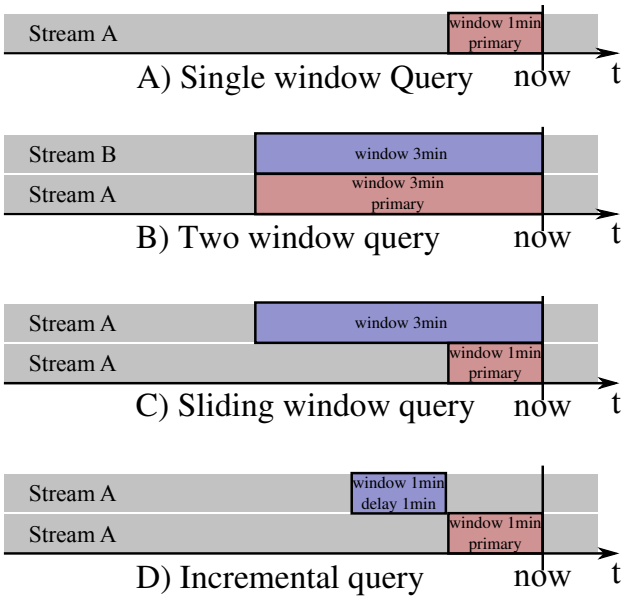


Fig. 1. Multiple input window definitions possible in DBStream's Continuous Execution Language (CEL).

of the primary window), the query for this job reads data from each of the two input windows. Part C) shows how to independently define the window length and the frequency of query execution. The primary window is one minute long, meaning that the query is executed every minute. However, the query can access the last three minutes of the same stream A through the other window, enabling many interesting kinds of queries, such as a rolling average, sum or any other aggregation. Part D) explains the delay keyword. Here, the same input stream is referenced twice, but for the second window, a delay of one minute is specified. As a result, the query can read data from both the current minute (window 1min) and the previous minute (window 1min delay 1min) of stream A. This makes complex incremental queries possible, such as a rolling/moving set or median, by being able to reference the previous state of the data and compare it with the current state.

The main difference between DBStream's CEL and stream processing languages is the handling and definition of windows and sliding windows in particular. For example, in StreamBase [15], windows are specified as `[SIZE x ADVANCE y TIME]`, where `x` defines the length of the window and `y` the query execution frequency. In CEL, the *primary* keyword corresponds to `ADVANCE`, but is specified only once regardless of the number of inputs to make it clear how often to re-compute the query.

Although Fig.1 shows several possible window types, it still covers only a small fraction of possible window definitions. Since data in DBStream are always stored on non-volatile storage, windows can reference past history. It is possible to reference data from one week or even one month ago, e.g., to compare the current state of the network with the past.

B. Examples of Rolling Analytics

We now give two more complex incremental job examples, detailing how rolling analytics can be implemented in CEL.

We start with a rolling window average shown below, in which every minute, we calculate the average uploaded and downloaded bytes over the last three minutes.

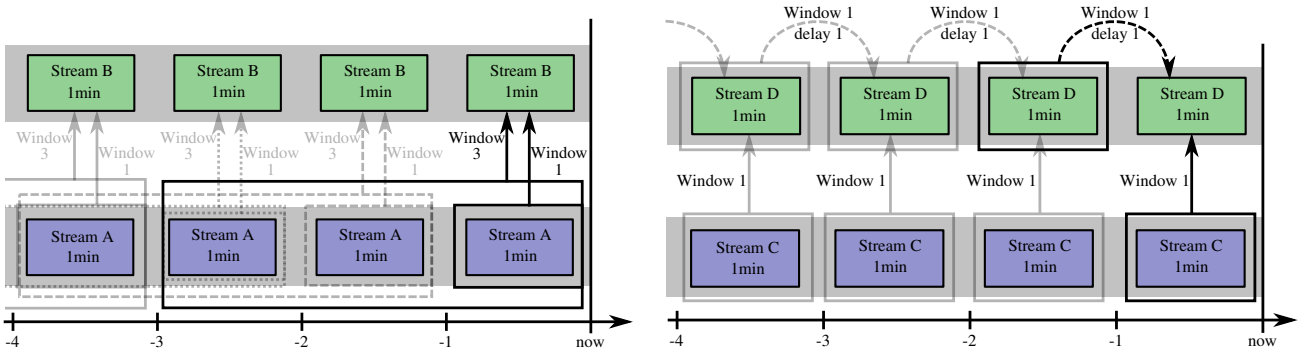
```
<job inputs="A (window 1min primary) as A,
             A (window 3min) as A2"
     output="B"
 schema="serial_time int4, avg_download float8,
  avg_upload float8">
<query>
  select _STARTTS, avg(download),
  avg(upload) from A2
</query></job>
```

The first window `A` is the primary window that denotes the query execution frequency. The second window, `A2`, is used to run the actual average calculation. Fig. 2(a) illustrates how the windows over stream A correspond to results appended to B; the output of the above job is a sequence of new results generated every minute, all of which are stored in B and identified by their `_STARTTS` (window start time) timestamps.

There is a simple performance optimization that can easily be expressed in CEL: we can pre-aggregate each minute of the data in A using one query, and then write a second query to add up the three most recently pre-aggregated windows and compute the three-minute aggregates.

In the next example, we compute the distinct set of IP addresses active in the last hour, updated every minute. A naive approach is to always scan the last hour of data from scratch whenever the result is to be updated. A more efficient approach is to keep an intermediate state of distinct IP addresses of the last hour in memory. Then, we can compute the distinct set of IP addresses for the current minute as the union of the set of IP addresses from the current minute and those from the last 59 minutes. However, since state is kept in memory, it must be re-built in case of a system crash. In CEL, we can implement the latter via a job that uses its own past output as input. This approach is not only more efficient, but also, as we show in Sec. IV-C, it is more fault-tolerant since the state of the computation is actually stored in the output table.

The corresponding CEL job definition is shown below. The input is a stream `C`, which contains, among other things, the IP addresses of active terminals. We now want to transform stream `C` into a new stream `D` containing, for each minute, the distinct set of active IP addresses in the last hour. To achieve this, we first add a new timestamp `last` to `D` recording the time of the last activity of a IP address. Now, from the current minute of `C`, we produce a new tuple for each distinct IP address and we set the last activity to the start of the current window using the `_STARTTS` keyword. From the previous minute of `D` we select those IP addresses which where active less than one hour ago. We then combine those two results using the SQL `UNION ALL` operator and select for each



(a) Rolling average over the last 3 minutes, updated every minute.

(b) Complex data processing flow for an incremental query.

Fig. 2. Data flow of two example incremental jobs; the windows of the current task are marked in black.

distinct IP address, the current time, the maximum value of the last activity timestamp, and the IP address itself. By using this feedback loop, we can efficiently compute the set of IP addresses active in the last hour per minute, without keeping any additional state information. The windows used in this computation are visualized in Fig. 2(b).

```
<job inputs="C (window 1min primary),
             D (window 1min delay 1min)"
     output="D"
 schema="serial_time int4, last int4, ip inet">
<query>
select _STARTTS, max(last), ip
from (
 select _STARTTS as last, ip
 from C
 group by 1,2
   union all
 select last, ip
 from D where last <= _STARTTS-60
 group by 1,2)
group by 1,3
</query></job>
```

IV. Performance Analysis

We now compare DBStream with respect to the state-of-the-art Big Data framework Spark. Spark is an open-source MapReduce solution proposed by the UC Berkley Amplab. It exploits Resilient Distributed Datasets (RDDs), i.e., a distributed memory data abstraction which allows in-memory operations on large clusters in a fault-tolerant manner [17]. This approach has been demonstrated to be particurlarly efficient [3] enabling both iterative and interactive applications in Scala, Java or Python. Spark does not strictly require the presence of Hadoop cluster to run. In fact, despite the system is commonly used in combination with Hadoop and HDFS, it also offers a simple, standalone resource manager to coordinate the activities of different hosts and supports direct access to the Linux file system.

A recent evolution of Spark is Spark Streaming [18]. Differently from Spark, which is a pure batch processing solution, Spark Streaming enables real time analysis through processing of short batches. Of particular interest are the system primitives for defining sliding windows and developing incremental queries similarly to what was discussed in Sec. III-A. However, Spark Streaming targets mainly real time analysis scenarios and offers limited support for processing historical data, which is also required by NTMA. Recent discussions on the Spark Streaming mailing list suggest that some workarounds may be possible[2]. However, we were unable to implement these and therefore we leave the evaluation of Spark Streaming for rolling analytics as future work.

A. System Setup and Datasets

We installed DBStream and Spark on a set of 11 machines having the same hardware (6 core XEON E5 2640, 32 GB of RAM and a 5 HD of 3TB each). One machine has been dedicated to DBStream, recombining 4 of the available HDs in a RAID10 and installing PostgreSQL v9.2.4 as a underlying Database Management System (DBMS). The remaining 10 machines compose a production Hadoop that runs CDH 4.6 with Map Reduce v1 Job Tracker enabled. On the cluster we also installed Spark v1.1.0 where we could only enable the standalone resource manager[3].

All machines are located within the same rack connected through a 1Gb/s switch. The rack also contains a 40TB NAS used to collect historical data. In particular, we use four 5 day-long datasets, each collected at a different network Vantage Point (VP) in a real ISP network between February 3 and February 7, 2014. Each VP is instrumented with Tstat [7] to produce per-flow text log files from monitoring the traffic of more than 20,000 households. For the purpose of this work we focus only on TCP traffic for which Tstat reports more than 100 network indexes and generates a new log file each hour. Overall, each VP generated a dataset of about 160 GB of raw

[2]http://apache-spark-user-list.1001560.n3.nabble.com/window-analysis-with-Spark-and-Spark-streaming-td8806.html#a9185

[3]Apparently, the implementation of Yarn provided in CDH 4.6 has some incompatibilities with Spark. These seem be solved in CDH 5 providing Yarn by default and a parcel for Spark v1.1.0. Unfortunately, testing such a configuration requires an upgrade of the node operating systems, which was not possible to do in our production environment.

data (i.e., about 5 times the memory available on each node) for a total of about 640 GB (i.e., twice the memory available on the whole cluster).

B. Benchmark Definition

We use a set of 7 jobs, representing daily operations performed on a production Hadoop cluster we are considering.

J1: for every 10 minutes, i) map each destination IP address to its organization name (orgname for short) through the Maxmind Orgname database (www.maxmind.com/en/geoip2-isp), and ii) for each Orgname found, compute aggregated traffic statistics (min/max/average Round-Trip Time (RTT), number of distinct server IP addresses, total number of uploaded/downloaded bytes).

J2: for every hour, i) compute the orgname-IP mapping as in J1, ii) filter all orgname's related to the Akamai CDN, and iii) compute some aggregated statistics (min/max/average RTT).

J3: for every hour, i) compute the orgname-IP mapping as in J1, and ii) select the top 10 orgname having the highest number of distinct IP addresses.

J4: for every hour, i) transform the destination IP address into a /24 subnet, and ii) select the top 10 /24 subnets having the highest number of flows.

J5: for every minute, for each source IP address, compute the total number of uploaded/downloaded bytes and flows.

J6: for every minute, i) find the set of distinct destination IP addresses, and ii) use it to update the set of IP addresses that were active over the past 60 minutes.

J7: for every minute, i) compute the total uploaded/downloaded bytes for each source IP address, and ii) compute the average over the past 60 minutes.

Overall, these jobs define performance indexes related to CDN (J1 to J4), statistics related to the monitored households (J5), and two incremental queries (J6 and J7).

C. Benchmark implementation

Each analysis engine has different peculiarities, properties and tuning options. Different implementations are therefore possible for the defined benchmark. We define a possible implementation that we consider reasonable, discussing possible modifications that can affect performance.

DBStream benchmark: All queries are expressed in the Continuous Execution Language (CEL). The fact that the output of a job is stored on disk and can be used as input to another job is exploited to achieve better performance. Fig. 3 shows the resulting job dependencies, where the nodes represent the jobs and an arrow from e.g. job `J1` to `J2` means that the output of `J1` is used as input to `J2`. The number next to an arrow indicates the size of the input window in minutes. For instance, `J4` and `J5` are implemented in a single step using a input window of 60 minutes of imported data. Conversely, `J6` is implemented using an intermediate step `J6 prepare` which pre-aggregates the set of active IP addresses per minute in windows of 10 minutes of imported data. Now, `J6` can utilize the output of `J6 prepare` and combine it with its own past as output, as indicated by the reflexive arrow starting from and going back into `J6`, to compute the final result. Please note that each minute of `J6` contains the active IP addresses of the last 60 minutes along with a timestamp indicating when those IPs was last active. In each one minute step of `J6` this timestamp is checked and IPs which were last active longer than 60 minutes ago are removed.

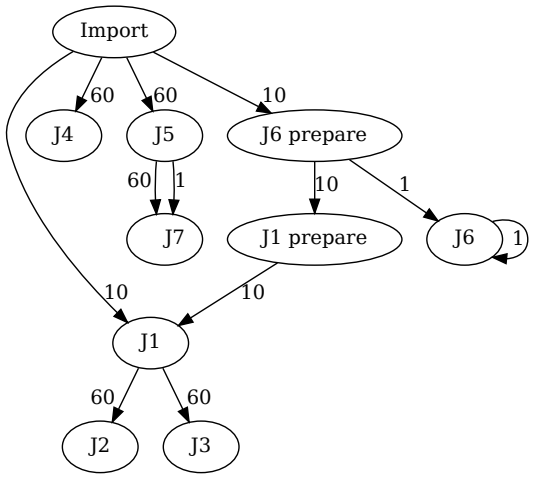


Fig. 3. Job inter-dependencies for the DBStream implementation. Nodes represent jobs and arrows precedence constraints.

Spark benchmark: Each job is implemented as a separate Spark application using Scala. Each application receives a list of files located on HDFS as input and processes them sequentially. The first 5 jobs have a straightforward implementation, since the do not present strong data dependencies and data are already split per hour. The two incremental queries, `J6` and `J7`, instead are more complex to implement. In fact, we need to implement the logic to store and update the data in windows. We consider a simple approach, creating an RDD collecting per-minute data bins on which we then loop to compose 60 minute windows. Our implementation processes data in a stream of hourly batches, where the results are available after each the processing for each batch has finished.

D. Results

Fig. 4 shows the results of running Spark on our cluster of 10 machines. The labels "1VP" and "4VP" correspond to the number of vantage points collecting data, i.e., 4VP corresponds to four times as much data as 1VP. For the jobs `J1` to `J5`, Spark offers excellent performance and the whole cluster is perfectly able to parallelize processing, leading to very good results. However, jobs `J6` and `J7` do not scale well. `J6` in particular cannot be parallelized very well, since data have to be synchronized and merged in one single location after each minute. We also tried different implementations of `J6` using more complex strategies and higher number of map/reduce tasks aiming to utilize further cluster resources, which turned out to be even less performing. Also for `J7`, the computation has to be synchronized for every minute, but here the amount of data is smaller since the output for every minute is only a single number. This might be the reason why `J7`

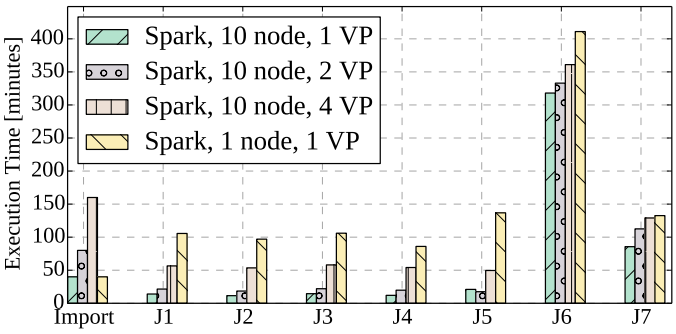


Fig. 4. Performance numbers for different setups using Spark.

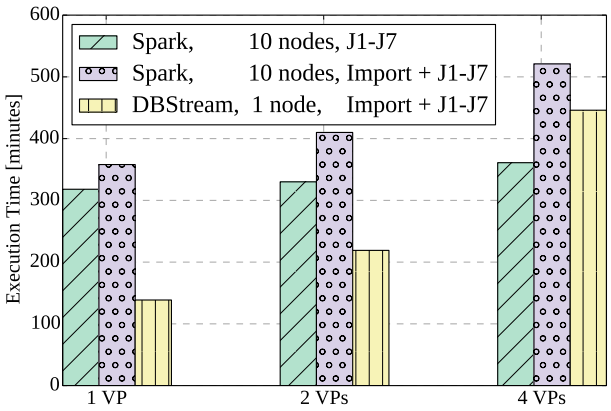


Fig. 5. Scalability comparison of DBStream and Spark.

does show a better performance than `J6`. Whereas we can not exclude the possibility of more performance implementation in Spark for `J6` and `J7`, these results show that obtaining good performance with Spark in such scenarios is not at all straightforward. Typical optimization used for such a problem such as skip lists or complex tree structures are hard to parallelize and would not be a fair comparison to a declarative language like CEL.

In Fig. 5, we compare the performance of Spark and DBStream. In DBStream, the total execution time is measured from the start of the import of the first hour of data until all jobs finished processing the last hour of data. For Spark, all jobs were started at the same time in parallel. We report the total execution time of the job finishing last, which was `J6` in this experiment. Since for Spark, data import and data processing is separated, we also report the solve job processing time without data import.

For DBStream, the execution time increases nearly linearly with the number of VPs and indicating a linear scalability, at least up to the used amount of VPs.In contrast, for Spark the main bottleneck is the execution time of `J6`. The total execution time does not increase much with more VPs, since multiple instances of `J6` run in parallel. Therefore, Spark is able to utilize its parallel nature better, the more jobs are running, whereas DBStream shows better performance for incremental jobs. Notably, for the 1 VP case, Spark takes 2.6 times longer to finish importing and processing the data.

V. Conclusion

In this paper, we presented the DBStream system for rolling big data analysis. We focused on the way in which DBStream allows a declarative specification of incremental queries, including those which access their previous results in order to compute new results. When tested with real network monitoring datasets and workloads, a single DBStream node performed as well as a cluster of ten Spark nodes due to the performance advantages of incremental processing.

There are several interesting directions for future work. One is to develop DBStream on top of a parallel database engine such as Greenplum so that it can scale-out as well as or better than Spark on cluster implementations. Another option is to use Spark (in particular, its latest version that can directly execute SQL queries) as DBStream's processing engine, and compare the two architectures. Finally, since network monitoring (and other monitoring applications) often involves complex machine learning that cannot be easily expressed in SQL, we will investigate how to implement rolling machine learning operators in DBStream.

References

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, M. Tatbul, S. Zdonik, "Aurora: a new model and architecture for data stream management", *THe VLDB Journal* 12(2):120-139 (2003).

[2] A. Bär, P. Casas, L. Golab, A. Finamore, "DBStream: an Online Aggregation, Filtering and Processing System for Network Traffic Monitoring", in *IWCMC 2014 - 5th TRAC Workshop*, 2014.

[3] Berkeley AMPLab, "Big Data Benchmark", https://amplab.cs.berkeley.edu/benchmark/, 2014.

[4] P. Bhatotia, A. Wieder, R. Rodrigues, U. Acar, R. Pasquin, "Incoop: MapReduce for Incremental Computations", in *SOCC* 2011, 1-14.

[5] K. Borders, J. Springer, M. Burnside, "Chimera: A Declarative Language for Streaming Network Traffic Analysis", in *USENIX Security Symp.*, 2012.

[6] C. Cranor, T. Johnson, O. Spatscheck, V. Shkapenyuk, "Gigascope: a stream database for network applications", in *SIGMOD* 2003, 647-651.

[7] A. Finamore, M. Mellia, M. Meo, M. Munafo, P. D. Torino, D. Rossi, "Experiences of internet traffic monitoring with tstat". *IEEE Network* 25(3): 8-14 (2011)

[8] L. Golab, T. Johnson, J. S. Seidel, V. Shkapenyuk, "Stream Warehousing with DataDepot", in *SIGMOD* 2009, 847-854.

[9] L. Golab, T. Johnson, S. Sen, J. Yates, "A sequence-oriented stream warehouse paradigm for network monitoring applications", in *PAM* 2012, 53-63.

[10] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, A. Doan, "Muppet: MapReduce-style processing of fast data", *PVLDB* 5(12):1814-1825, 2012.

[11] Y. Lee, Y. Lee, "Toward Scalable Internet Traffic Measurement and Analysis with Hadoop", in *SIGCOMM Comput. Commun. Rev. (CCR)* 43(1):5-13, 2012.

[12] B. Li, E. Mazur, Y. Diao, A. McGregor, P. Shenoy, "SCALLA: A platform for scalable one-pass analytics using MapReduce", *ACM Transactions on Database Systems* 37(4):1-43, 2012.

[13] E. Liarou, S. Idreos, S. Manegold, M. Kersten, "MonetDB/DataCell: online analytics in a streaming column-store", *PVLDB* 5(12):1910-1913, 2012.

[14] RFC 3954 - Cisco Systems NetFlow Services Export Version 9", 2004.

[15] StreamBase. "Streambase: Real-time, low latency data processing with a stream processing engine." http://www.streambase.com, 2014.

[16] T. White, "Hadoop: the definitive guide", *O'Reilly*, 2012.

[17] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, I. Stoica, "Spark: Cluster Computing with Working Sets", in *HotCloud* workshop, 2010.

[18] M. Zaharia, T. Das, H. Li, S. Shenker, I. Stoica, "Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters", in *HotCloud* workshop, 2012.