POS: A High-Level System to Simplify Real-Time Stream Application Development on Storm

Bin Cui¹ · Jie Jiang² · Quanlong Huang¹ · Ying Xu¹ · Yanjun Gui² · Wenyu Zhang²

Received: 25 August 2015 / Revised: 25 August 2015 / Accepted: 27 August 2015 / Published online: 8 September 2015
© The Author(s) 2015. This article is published with open access at Springerlink.com

Abstract Real-time stream computing becomes increasingly important due to the sheer amount of content continually generated in various kinds of social networks and e-commerce websites. Many distributed real-time computing systems have been built for different applications, and Storm is one of the most prominent systems with high-performance, fault-tolerance and low-latency features. However, the Storm programming paradigm is low level and leaves programmers' codes hard to maintain and reuse. In this paper, we present a high-level abstraction system on Storm, called POS. The POS system provides a Pig Latin-like language on top of the Storm execution engine. Programmers can write POS program, and the system compiles the program into physical plans which are executed over Storm. We discuss the challenges in developing POS system and elaborate on its implementation details. Our experiments show that POS yields satisfactory performance compared with raw Storm.

Keywords Real-time · Stream processing · Pig · Storm

1 Introduction

The fast-generated content from various kinds of websites poses great challenges on data processing systems [1]. The widely used Hadoop system [2] is born to do batch processing, which is not suitable in stream computing circumstances. Many real-time computing systems gradually draw users' attention, such as the open-source systems Storm [3] and Yahoo S4 [4], business softwares Esper [5] and Stream-Base [6].

As a distributed system, Storm has some promising features, e.g., fault-tolerance, low latency and high performance. Many companies, including Twitter, Alibaba and Baidu, use Storm to deal with their data analysis jobs. In Tencent Inc., a Storm cluster with more than 2000 physical nodes has been built to provide service for a wide variety of applications, such as QQ, Tencent video and Wechat. The service includes basic data processing, item recommendation, advertisements billing and real-time monitoring. A Storm program for a complex application may contain tens of Storm units (spout or bolt), and the total code length may exceed several thousands.

Nevertheless, as the online applications become complex, Storm gradually shows its inefficiency in code developing and program maintaining. Storm provides two programming primitives, spout and bolt. Users must be skilled in one programming language and master the two primitives programming specifications to process data analysis tasks, which makes it hard for non-expert or new users to use. In addition, Storm does not support common operations, e.g., projection and filtering. Furthermore, the rigid and low-level programming paradigm makes the codes hard to maintain and reuse. However, unlike batch processing frameworks which have many high-level programming languages such as Pig [7],

✉ Bin Cui
bin.cui@pku.edu.cn

Jie Jiang
zeus@tencent.com

Quanlong Huang
Quanlong@pku.edu.cn

Ying Xu
yingxu@pku.edu.cn

Yanjun Gui
ivangui@tencent.com

Wenyu Zhang
gabyzhang@tencent.com

¹ Key Lab of High Confidence Software Technologies (MOE), School of EECS, Peking University, Beijing, China

² Department of Big Data Platform, Tencent Inc., Shenzhen, China

DryadLINQ [8], SawZall [9], Storm has few high-level programming languages by far.

To address the aforementioned problems, we develop a high-level system for Storm, named *Pig On Storm* (POS). Pig [10, 11] is a big data analysis system, which provides a high-level language Pig Latin for Hadoop. It transforms Pig Latin scripts into Map-Reduce jobs which are executed on Hadoop. Pig can provide high-level operations, e.g., filtering, projection, aggregation and user-defined functions (UDF), and highly increase the productivity of coding Map-Reduce jobs. However, transplanting Pig onto Storm is not a trivial task. Pig is designed to execute jobs on batch processing system Hadoop, which is totally different from Storm. Many Pig Latin syntax do not make sense on Storm execution engine. Moreover, as a real-time processing platform, Storm requires fast response time which is not the case for Hadoop.

In this paper, we discuss the challenges in developing POS and present the implementation details of POS. In the POS system, we adopt Pig Latin-like language as our programming language. In order to enable the real-time features, we extend the Pig Latin syntax by adding four new operators: TAP, PARTITION, BIND and WINDOW. To achieve low latency of Storm, we develop an auxiliary in-memory key-value cache to store the intermediate status and results. A POS compiler is constructed to compile POS scripts into Storm topologies. When a script is submitted to the POS compiler, it will go through syntactical analysis, physical plan construction, topology plan generation stages, and finally, it will be transformed into a Storm topology. We conduct extensive experiments on POS. The results demonstrate that POS has a comparable processing ability with raw Storm. POS has been deployed in Tencent to support various applications [12], including item recommendation, advertisement targeting. The daily processed tuples are more than 5 billion per application. To sum up, the contribution of our work can be listed as follows:

1. To the best of our knowledge, POS is the first practical work transplanting Pig to Storm. It has been already deployed online and yielded satisfactory performance and good stability in real business applications.
2. By extending Pig Latin syntax, our POS system can satisfy the real-time demands with the easiness of programming and less code maintenance efforts.
3. The efficiency of POS system has been comprehensively measured, and the experimental results demonstrate that our system has a comparable performance with raw Storm and meanwhile improves the developing productivity largely.

The rest of the paper is organized as follows. Section 2 introduces the related big data processing systems and the challenges in developing POS. In Sect. 3, we present the system overview of POS. We elaborate on the POS syntax

in Sect. 4 and introduce the implementation details of POS compiler in Sect. 5. Section 6 shows the experimental evaluation. Finally, we conclude our work in Sect. 7.

2 Background and Challenges

2.1 Storm

Storm [3] is a distributed fault-tolerant real-time computation system which was proposed by Twitter. It is designed to do real-time data analysis tasks, such as processing the continuous Twitter tweet streams. In Hadoop, data analysis task is included in one or more Map-Reduce jobs. Accordingly, in Storm, we construct the topologies to process data analysis tasks. A *topology* consists of spouts and bolts and the links between them. These units together construct a data processing Directed Acyclic Graph (DAG) which represents the whole stream processing procedure. A *spout* is the source of a data stream. It reads in data tuples and emits them to the following bolts. A *bolt* is a data processing unit, which consumes any number of input streams, conduct some specific processing, and finally may emit new tuples out to other bolts. A complex data processing task always needs multiple bolts.

2.2 Pig

Apache Pig [7] is a big data analysis platform. It is composed of a high-level dataflow language Pig Latin and a lower-layer data processing and transforming system. The infrastructure layer is generally based on Hadoop. Pig takes a Pig Latin script as input and transforms it into a Hadoop Map-Reduce job, then monitors the job running on a Hadoop cluster. Taking advantage of the powerful processing ability of Hadoop, Pig provides very easy programming primitives and resolves big data problem in a large-scale distributed way.

Pig Latin contains many commonly used high-level data transformation operators like LOAD, FILTER, FOREACH, JOIN. These operators make it convenient for programmers to resolve data analysis jobs, and the readers can refer to [7, 11] for more details. Another important issue about Pig is its data model. Pig Latin has a rich data model, which contains simple atom data types, including int, long, float, double, boolean, chararray, as well as the complex data types including tuple, map and bag. These complex data types are consistent with programmers' programming habits. With the rich data model and step-by-step procedural ability, Pig provides users with excellent programming experience.

2.3 High-Level Systems on Storm

Trident¹ and Summingbird [13] are two high-level systems on top of Storm. Trident is an abstraction for Storm, pro-

¹ <https://storm.apache.org/documentation/Trident-tutorial.html>.

viding convenient APIs like joins, aggregations, grouping, filters. When using Trident to develop applications, programmers are still required to understand the concepts of Storm and define topologies by themselves. Trident just encapsulates some common operations for programmers to reduce some programming workload. It lacks the flexibility of high-level languages like Pig or Hive. Summingbird is a domain-specific language designed to integrate batch and online processing. One Summingbird program can run on two different platforms, Hadoop for batch processing and Storm for stream processing.

However, these two systems cannot well support the real-time processing jobs in real applications. First, many of Tencent applications need to count exact statistics at any time point, like the Unique Visitor (UV) and Page View (PV) of a website or some monitoring data. Trident processes the stream as small batches of tuples, which makes the results not precise enough at any specific time point. We need a system which can process stream in a finer-grained tuple-by-tuple way. Second, in many online business, time window operation is most commonly used. Though Trident and Summingbird implicitly support sliding window operation, they do not provide a window operator with which user can operate on tuples collected during arbitrary time intervals conveniently. In addition, these two systems can only be used by programmers and are more complex than Pig.

2.4 Challenges of Transplanting Pig

In this paper, we aim to develop POS system. However, transplanting Pig onto Storm faces many challenges.

First, many Pig operators are not suitable for stream processing scenario. Storm manipulates data streams which are unbounded and endless, whereas Hadoop processes the fixed data relations stored in HDFS [14]. The original Pig operators such as GROUP and LIMIT are designed to work on the fixed data relations. In POS, these operators must change their semantics and acquire different processing logic.

Second, a Pig script is compiled into sequences of Map-Reduce jobs, and the intermediate results between two Map-Reduce jobs are stored in HDFS. However, as a real-time processing platform, Storm requires fast response time. If we store the intermediate results on disk, the I/O expense will result in very high latency. POS must find out ways to ensure the low latency of Storm system.

Third, in stream processing applications, time window operation can occupy about 80 percent of online applications. We need to provide a high-performance time window operation in POS system, which is not supported by Pig.

We also face many implementation obstacles in developing POS, like the incompatibility between Pig data model and Storm data model, the bolts partition problem in topol-

ogy generation procedure. All the aforementioned challenges will be further discussed and solved in the following sections.

3 System Overview

Figure 1 shows the framework of POS system, which consists of four parts, i.e., user interface, POS compiler, Storm executor and in-memory key-value engine (KVE).

The user interface provides the Pig Latin-like language. In POS, we extend Pig Latin syntax to satisfy real-time processing features. Users can write POS scripts according to the syntax rules and submit to POS compiler.

The POS compiler works in four stages. First, the submitted POS script is parsed syntactically and semantically into a logical plan which is made up of logical operators and their relations. Second, a physical plan is generated by traversing the logical plan to convert each logical operator into an executable physical operator. Third, a topology plan is constructed by partitioning the physical operators into different parts. Finally, an executable Storm topology is built from the partitioned topology plan.

In real-time stream computing system, online applications will produce many intermediate results and data status. In original Pig system, all generated data are stored on disk. But the frequent disk operations will definitely increase latency, resulting in a huge efficiency drop of the system. Therefore, we develop an auxiliary system, i.e., an in-memory KVE to solve this problem. KVE is a networked, distributed in-memory key-value cache like Redis [15]. Like many other key-value data stores, KVE offers users two common data

Fig. 1 System design overview

access functions, “get” and “put” for reading and writing data pairs. The maximum size of key is 4 KB, and the value’s maximum size is 1 MB. The value field can be a simple string containing multiple data fields connected by separators like a tab ‘\t’ or a comma, as well as a complex data object defined in the form of Prototype [16].

The KVE adopts client–server framework. It has two types of servers, configuration server and data server. The configuration servers keep the matching information between data items and data locations in route table. The data servers store the real data items and their backups. When a read or write operation is submitted to KVE, the client connects a configuration server to get the route table and replies with the data location. Then, the application can exchange data with data server directly. This structure makes KVE a reliable data storage with high scalability. Due to its high-performance and scalability, it now supports an average of 500 billion data accesses daily. In POS, we encapsulate KVE operations into several UDFs, which greatly simplify the interaction with KVE for POS programmers.

4 POS Syntax

In this section, we present the syntax of POS language. The basic syntax rules of POS, such as case sensitive and naming rules, are remained the same as original Pig Latin. POS supports nearly all the Pig operators and built-in functions, except that we make some alterations on operators and functions to adapt to real-time features. Our system provides extensive UDF for several data processing situations, e.g., input and output normalizing, string parsing, time-related processing and status monitoring. We also collect the universally used algorithms like the counting of click through rate (CTR) and various kinds of hashing functions into one algorithm package. All these UDFs can be easily used in POS scripts.

4.1 An Overall Syntax Comparison of Apache Pig and POS

The following is a comparison between Apache Pig script and POS script. Both these two programs solve the WordCount problem.

Apache Pig

```
a = load 'word_count_text.txt';
b = foreach a generate flatten,
    (TOKENIZE((chararray)$0)) as word;
c = group b by word;
d = foreach c generate COUNT(b),
    group; store d into 'pig_wordcount.txt';
```

POS

```
a = tap using LocalFileTap('word_count_text.txt'); (1)
b = foreach a generate flatten,
    (TOKENIZE((chararray)$0)) as word; (2)
c = partition a by word; (3)
d = foreach c generate word,
    LoadFromKVE(word) as count; (4)
e = foreach d generate WriteToKVE(word,
    (count is not null ? (int)count + 1 : 1)); (5)
```

As shown in these two scripts, we can see the difference in each step of POS:

- (1): We replace LOAD operator with TAP to deal with stream inputs. LocalFileTap() is an UDF to read data tuples from external disk storage;
- (2): The FOREACH operator’s usage is as same as the original one;
- (3): We add a new operator PARTITION to create a new bolt. This statement will partition the same words into same bolts in Storm topology, ensuring the same words are counted in one processing unit;
- (4) and (5): Original Pig relies on Hadoop to execute the script. Therefore, the data being processed by each Pig statement is a whole data relation stored on HDFS. For example, in Pig, ‘c’ is a whole relation grouped by each word. Counting the word amount is very easy by performing COUNT built-in function on ‘c’. However, in POS, we use Storm as our execution engine and each POS statement processes one input at a time. We cannot use COUNT function to do calculation on one input tuple. Therefore, we are forced to record the intermediate results. In this example, we need to record a <word, count> map. Here, we utilize the in-memory store KVE to save data status. The UDF LoadFromKVE() plays the role of reading word-count status from KVE. And we use WriteToKVE() UDF to write the result back into the KVE instead of using STORE operator.

4.2 The Altered Pig Operators

In POS, we retain the FILTER, FOREACH and SPLIT operators since they have clear semantics in real-time circumstances. Relational operators are selectively supported in POS system, including UNION, LIMIT, DISTINCT, GROUP and ORDER BY. We drop the LOAD, STORE and Map-Reduce operators which are meaningless in real-time computing situation.

In original Pig, relational operators can cover most of the data processing tasks, whereas in POS, they are not fre-

quently used in practical applications. This phenomenon can be explained from the aspect of the execution model. Original Pig depends on batch processing system Hadoop. Pig operators process the whole data set that can be treated as a relation on which relational operators are applied naturally. However, POS relies on Storm. The operators only process one input tuple at a time, making it senseless to do relational operations.

The original relational operations such as GROUP and LIMIT must have different semantics in POS. We find out that in stream processing, there are two cases where the relational operators can still work. One case is using a special kind of input UDF. This kind of UDF reads in one tuple and expands it into a bag of tuples, forming a small relation. The other one is accessing data from KVE. The value in KVE can be of a complex type. If the value contains a bag of multiple tuples, it will correspond to a small relation, too. In these two cases, we need relational operations to process the data item. In Pig, relational operators usually need several physical operators to realize its function, but in POS, since the relation size is relatively small, we use one physical operator to play its role, and this will be detailed in following sections.

In our real-time data processing tasks, the retained operators and the new created operators (introduced in the following section) together can meet all our real-time computation requirements. The operators of POS are listed in Table 1.

Table 1 POS operators and usage

POS operator	Usage scenario
TAP	Read tuples from an input stream
DUMP	Write results to stdout
FILTER	Eliminate the unwanted tuples by a specific condition
FOREACH	Generate one or more new tuples for each input tuple according to a certain mapping function
GROUP	Collect tuples with the same key together from one or more relations
ORDER	Rank the bag tuples by a certain field
DISTINCT	Remove the duplicate tuples
UNION	Merge two bags of tuples
LIMIT	Limit the number of tuples of a bag
SPLIT	Split the input tuples into different relations. This operator can be treated as a multi-condition FILTER
WINDOW	Group the tuples accumulated in a certain time interval together to do some aggregation jobs
PARTITION	Split the operators into different bolts
BIND	Combine two or more new bolts generated by PARTITION to be one bolt

4.3 New POS Operators

Original Pig operators cannot satisfy real-time demands. To meet the special requirements of real-time computation, we design four new operators elaborately, i.e., TAP, PARTITION, BIND and WINDOW. Their syntax is listed as follows.

- *TAP*: *alias* = *TAP* [*USING function*] [*AS schema*] [*PARALLEL n*];
- *PARTITION*: *alias* = *PARTITION alias BY expression* [*PARALLEL n*];
- *BIND*: *BIND alias, alias [, alias ...]*;
- *WINDOW*: *alias* = *WINDOW alias INTERVAL n GROUP {ALL | BY expression} GENERATE expression* [*AS schema*] [*expression* [*AS schema*] ...];

TAP: This operator is used to read data into the system. Unlike Hadoop in which data are read from disks, a spout may read tuples from a Kestrel [17] queue or connect to the Twitter API in Storm system. We use TAP operator to fulfill this task. This operator will map to the spout module in the topology.

e.g., `data_items = TAP USING SysTap('a.txt')`
`AS (f1 : int, f2 : chararray);`

PARTITION: PARTITION is used to divide streams into many small streams according to the partitioned field. And the PARTITION operator also works as a symbol to split POS operators into different bolts.

e.g., `data_partitioned = PARTITION data_items BY f1;`

BIND: Contrary to the PARTITION operator, BIND operator combines two or more streams into one. In a Storm topology, sometimes two or more streams from different bolts need to flow into one bolt. In POS, this is realized by BIND operator.

e.g., `BIND data_partitioned, data_partitioned_2;`

WINDOW: Doing some statistical analysis of the data during a certain time interval is a very common computation scenario in real-time stream systems. However, this function is not supported in original Pig Latin syntax. So we devise WINDOW operator to deal with time window operation. The WINDOW operator first accumulates a certain amount of tuples in a specified time interval. These tuples together make up a relation. Then, a GROUP operation is done on this relation. Finally, we implement COUNT, AVG, SUM or other statistical counting and relational operations on the grouped relation.

e.g., `data_wined = WINDOW data_partitioned`
`INTERVAL 60`
`GROUP BY f1 GENERATE *,`
`COUNT(data_partitioned.f2);`

The new designed operators solve stream data accessing problem, fulfill the stream partitioning and binding function, and accomplish time window operation. Consequently, POS can tackle stream computation tasks easily.

5 POS Compiler

This section describes how to translate a POS script into a Storm topology. We describe logical plan, physical plan and topology plan in details and explain the translation process between them.

5.1 Logical Plan Generation

A user-submitted POS script will go through several transformation steps. The first step is parsing. This procedure includes syntactic errors checking, schema inference, UDF initialization. We adopt ANTLR [18] to parse the script and construct an Abstract Syntax Tree (AST). A logical plan is generated based on AST afterward. In this conversion, each statement in script matches with one logical operator in the plan. Each logical operator contains rich information, including the alias, the type schemas and sometimes a sublogical plan which contains the specific processing logic. Figure 2 illustrates a logical filter operator. In Fig. 3, the transformation from (a) to (b) illustrates the one-to-one mapping mechanism of POS script to a logical plan.

5.2 Physical Plan Generation

Even though a logical plan consists of all the useful information of a script, it is just a static structural presentation of the parsing results. In order to execute each operator's processing logic, a logical plan needs to be translated into a physical plan.

A physical plan is made up of physical operators and their relations. Unlike logical operators, physical operators

Fig. 2 Operator FILTER in a logical plan

are embedded with the concrete execution procedure. For example, a FILTER operator accomplishes its operation of discarding the unwanted tuples in the functions of its physical operator. The processing functions of physical operators can be directly called later by Storm bolts or spouts.

However, the execution of physical operators faces the data format conversion challenge. Pig supports nested complex data structure as introduced in Sect. 2.2, whereas Storm uses tuples with one or more fields as its data model. How to represent different kinds of nested Pig data structures in form of tuples is a vital problem. We accomplish this issue by devising a wrapper of the Pig data model. When transmitting data from one operator to others, a complex Pig data structure is wrapped into a unified tuple with one field of “Wrapper” type. With the wrapping process, the complex Pig data model is invisible to Storm spouts and bolts. To be specific, when a physical operator gets the input wrapped tuple, it will first unpack the wrapper and parse the data structure with the user-defined or automatically inferred Pig data schema. Then, the physical operator does its own processing logic. Finally, the physical operator will again wrap the output data into a unified tuple to transmit to other operators.

In Pig system, some logical operators need more than one physical operator to realize their functionality, such as JOIN and COGROUP [10]. This is because the original COGROUPed or JOINed relations deal with batch data which can be too large to be executed on one machine. As a result, COGROUP or JOIN operator is translated to several physical operators which will be executed on different physical machines. Contrary to Pig, POS processes the data tuples one by one. In POS, the relation processed by operators is small compared to that in Pig (if the tuple is read from KVE, then the relation size cannot exceed 1 MB); thus, we do not need to split one logical operator into several physical ones to execute on different machines. In POS, one logical operator becomes one corresponding physical operator, obeying the one-to-one mapping rule, leading to a similar plan graph as shown in Fig. 3b.

5.3 Building a Topology Plan and an Executable Topology

A topology plan is constructed based on a physical plan. Figure 3, the transformation from (b) to (c), illustrates the converting procedure from a physical plan to a topology plan. All the physical operators are embedded into their corresponding topology operators, except for PARTITION and BIND, which are used as partitioning markers. These two operators do not have specific processing logic but are used to split the processing procedure into different bolts.

A Storm topology is built on the basis of the topology plan. In constructing an executable topology, POS calls the Storm topology builder APIs. Each topology operator will

Fig. 3 The conversion from a POS script to a topology plan. **a** POS script. **b** Logical and physical plan. **c** Topology plan

match to one Storm topology module, spout or bolt. When a topology is executed on a cluster, each topology operator's subplan that consists of all the partitioned physical operators will be executed. In Fig. 3c, "Topology Operator1" will be translated into a spout, and the other three topology operators will be translated into corresponding bolts.

When a topology plan is constructed, the operators PARTITION and BIND play a key role. PARTITION and BIND operators realize stream grouping mechanisms. In topology plan building procedure, they serve as functional markers. They do not have real processing logic and will not impair a topology's execution efficiency. PARTITION finishes the current spout or bolt and starts a new bolt for the following operators, i.e., it is used to create new bolts. Additionally, it tells that the tuples transferred between the separated two components follow field grouping rule.

BIND must be used immediately after two or more PARTITION operators, that is, BIND operator must have PARTITION operators as its parents in a logical and physical plan. BIND marks that the newly generated bolts by PARTITION are combined into one bolt. From the perspective of stream grouping, BIND realizes the all grouping mechanism. It sends tuples with same fields into one processing task.

To be specific, take WordCount problem as an example. We have two data sources, i.e., spout1 and spout2. They both have an 'id' field. In one Tencent application, we need to calculate the occurrences of items with different ids from different streams. Apparently, the counting process needs to make contact with a temporary storage (in our system, we use KVE) to store the counting value in form of a map item $\langle id, count \rangle$. If the counting is processed in different bolts, it will result in the inconsistency of data. To be specific, when bolt1 and bolt2 simultaneously read $\langle id : "id1", count : "5" \rangle$ from the KVE, they update the KVE by increasing the

count field by 1; then, they both write to KVE with the new value $\langle id : "id1", count : "6" \rangle$ successively. After the updating, the KVE will have $\langle id : "id1", count : "6" \rangle$ instead of the right value $\langle id : "id1", count : "7" \rangle$. Therefore, the counting job must be done in one bolt; then, the readings and writings of KVE will be queued to guarantee the correctness of the KVE status. In this simple example, we first use two PARTITION operators separately for the two streams to send tuples according to their fields; then, we use BIND operator to combine the same fields tasks from two streams into one.

Semantically, BIND is used to combine multiple bolts into one. However, Storm does not provide interface for bolts combination. Here we devise a "backward seen" strategy to realize the function of BIND. In the process of building topology plan, when a PARTITION operator is visited, it will first look backwards to see whether there is a BIND operator following it. If not, it will create a new bolt and continue the visiting process. Otherwise, it will first mark itself as a "no-new" PARTITION and delay the bolt creation procedure to the following BIND operator. The operators coming after PARTITIONs will be cut off and moved to the BIND operator. Phase 1 in Fig. 4 illustrates the rebuilding process. The FOREACH1 operator following PARTITION1 and the FOREACH2 operator following PARTITION2 are both cut-off and reconnected to BIND operator. After re-building the physical plan, it is very easy to split the physical plan into different topology units, as shown in phase 2 of Fig. 4.

PARTITION and BIND not only play the partitioning and binding role, but also provide users with the ability to assign system resources to each partitioned part, because spouts and bolts are the basic resource allocation units in Storm. Using PARTITION and BIND is a convenient way to control the program's parallelism as well as system resources allocation.

Fig. 4 The functionality of PARTITION and BIND

6 Experiments

In this section, we evaluate the performance of POS by conducting three groups of experiments. For each group, we write POS programs and the corresponding Java-written Storm programs and run them on the same hardware separately to compare their processing efficiency. Since the POS programs are executed in an interpretive way, they are doomed to be slower than the raw Storm ones. Please note, the experimental results in [10] also showed that the Pig transformation introduces performance degeneration with a performance ratio of 1.5 (the running time of Pig program/the running time of a Java Map-Reduce program) on average.

The first two groups of experiments measure the translation efficiency of POS. To avoid the impact of network latency and the heterogeneity of the Storm cluster, we conduct the first two groups of experiments on a single physical node. In both experiments, the spouts are set to emit tuples at the highest speed. The total number of the tuples emitted at a time is set to 1000.

Additionally, we evaluate the POS performance by an online application. This application runs on a Storm cluster of 50 machines. Each machine is equipped with a four-core processor and 32 GB of RAM. Since the production generates a great amount of partial results, it needs 8 machines of KVE cluster. Each one is equipped with 64 GB of RAM to store the key-value items.

6.1 Single Operator Performance Evaluation

In the first experiment, we measure the transforming efficiency of POS operators one by one. We put each operator into a separate Storm bolt and compare it with the corresponding Java-written Storm bolt. For both POS and raw Storm, we

Table 2 POS operators versus raw Storm implementation

POS operators	Storm (ms)	POS (ms)	POS/Storm
FILTER	26.4	29.4	1.11
FOREACH	28.7	32.6	1.14
GROUP	34.8	53.4	1.53
ORDER	34.4	60.3	1.75
DISTINCT	27.8	48.4	1.73
LIMIT	24.7	37.0	1.50
UNION	28.2	34.7	1.23
SPLIT	23.6	32.4	1.37
WINDOW	23.9	28.6	1.20

construct simple topologies consisting of one spout and one bolt to avoid the influence of other parts. We add timer in the bolts to record the time of executing a pre-defined number of tuples. Here the number is set to 1000, as mentioned above. For each operator, we run the program ten times and report the average time cost. Table 2 shows the comparison of the average performance of each operator. For example, in Table 2, the average time for operator FILTER to process 1000 tuples is 29.4 ms, whereas the average time of raw Storm bolt fulfilling the logic of FILTER is 26.4 ms. The ratio of the running time of POS FILTER to Storm bolt is 1.11. From the table, we can see that the relational operators perform worse than the others. This is because the relational operators, i.e., GROUP, LIMIT, DISTINCT and ORDER, work with an extra FOREACH operator. A FOREACH operator with a “flatten” clause can expand one input tuple into a tuple list representing a relation on which relational operators can operate. The tuple expanding process is generally costly; thus, the relational operators perform worse than the others.

6.2 POS Script Performance Evaluation

We next evaluate the whole POS scripts which may contain some built-in functions and combined operators. We use two simple topologies that are provided in Storm-Starter,² i.e., the WordCount and the Exclamation topology. Both topologies consist of one spout and two bolts. We also specialized two simulated complicated examples which cover all aforementioned operators. The first simulated topology has one spout and three bolts with a procedure of stream forking, and the second topology consists of two spouts and three bolts with a stream converging procedure.

We execute 20 times for each task and calculate the average execution time. Table 3 shows the performances of these four topologies. From this table, we can see that the WordCount topology takes much longer time than the other three

² <http://storm.incubator.apache.org/>.

Table 3 POS scripts versus raw Storm implementation

Topology name	Storm (ms)	POS (ms)	POS/Storm
WordCount	824.75	1001.45	1.21
Exclamation	90.1	117.35	1.30
Simulation1	106.4	220.7	2.07
Simulation2	330.9	794.9	2.40

topologies. The reason is that the WordCount topology first reads in a sentence and splits it into words which expand the stream. The expansion of tuples is also happened in topology Simulation2.

This group of experiments actually shows the worst case of our POS system, as the processing bolts are over loaded in this setting. On the contrary, the input tuples are not emitted at the highest speed in real application scenarios, but generally flow into the systems one by one with short latency which will leave time for bolts to process, thus making POS and raw Storm topologies differ little in the performance. This finding will be verified by the next evaluation on the real applications.

6.3 Real Application of POS

Here we describe an online Tencent business application running on POS, which counts the UV and PV of advertisements on all Tencent websites, including QQ space, Tencent News, Videos and Shopping websites. Since Tencent has a great amount of users (QQ users and Wechat Users), the total dataflow of this application is up to 5 billion daily. In this application, we need to count the real-time PV and UV in different dimensions (more than 40 dimensions in total), such as user properties: age phase, geographical domain, gender; ads property: type, advertiser, creativity id; action property: visiting time, behavior type and so on. The counting task can be very complicated due to the various dimensions combinations. The code length of Java-written raw Storm program exceeds 2000 lines; nevertheless, the POS script only contains less than 20 statements. In this real application scenario, POS processes about 95 thousand tuples per minute per worker, whereas the raw Storm topology can process 108 thousand tuples. POS has only a 13 % efficiency drop compared with raw Storm. This ratio represents a reasonable trade-off between execution efficiency and code development and maintenance effort.

7 Conclusion

In this paper, we have presented the POS system, which combines the Pig and Storm for real-time data analysis jobs. We designed four new operators to support specific operations in

real-time computing systems. By transforming a parsed logical plan into a physical plan and then into a topology plan, we can finally convert a POS script into an executable Storm topology. The experiments show that POS can yield comparable efficiency with the raw Storm implementation, but with much less code development and maintenance efforts.

The POS system is the first practical work transplanting Pig to Storm, which can be further improved by the following ways. First, when we build the Storm topology using POS, we need more elaborated optimization strategy-based cost analysis. Second, during the execution of topology, the input stream may change overtime; thus, the system will be more efficient if it can monitor the data update and adjust the topology dynamically.

Acknowledgments The research is supported by the National Natural Science Foundation of China under Grant No. 61272155 and Tencent Research Grant (PKU).

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Cui B, Mei H, Ooi BC (2014) Big data: the driver for innovation in databases. *Natl Sci Rev* 1(1):27–30
2. Hadoop: open-source implementation of mapreduce. <http://hadoop.apache.org/>
3. Storm: distributed and fault-tolerant realtime computation. <http://storm.incubator.apache.org/>
4. S4: distributed stream computing platform. <http://incubator.apache.org/s4/>
5. Esper. <http://esper.codehaus.org/>
6. Streambase. <http://www.streambase.com/>
7. Pig: a high-level big data processing language. <http://pig.apache.org/>
8. Yu Y, Isard M, Fetterly D, Budiu M, Erlingsson U, Gunda PK, Currey J (2008) Dryadlinq: a system for general-purpose distributed data-parallel computing using a high-level language. In: *Proc. of OSDI*, pp 1–14
9. Pike R, Dorward S, Griesemer R, Quinlan S (2005) Interpreting the data: parallel analysis with Sawzall. *Sci Progr* 13(4):277–298
10. Gates AF, Natkovich O, Chopra S, Kamath P, Narayanamurthy SM, Olston C, Reed B, Srinivasan S, Srivastava U (2009) Building a high-level dataflow system on top of map-reduce: the pig experience. *PVLDB* 2009:1414–1425
11. Olston C, Reed B, Srivastava U, Kumar R, Tomkins A (2008) Pig latin: a not-so-foreign language for data processing. In: *Proc. of SIGMOD*, pp 1099–1110
12. Huang Y, Cui B, Zhang W, Jiang J, Xu Y (2015) TencentRec: real-time stream recommendation in practice. In: *Proc. of SIGMOD*, pp 227–238
13. Boykin PO, Ritchie S, O’Connell I, Lin J (2014) Summingbird: a framework for integrating batch and online mapreduce computations. In: *PVLDB*, pp 1441–1451

14. Shvachko K, Kuang H, Radia S, Chansler R (2010) The hadoop distributed file system. In: Proc. of MSST, pp 1–10
15. Redis: open-source advanced key-value store. <http://redis.io/>
16. Prototype. <http://prototypejs.org/>
17. Kestrel: a simple, distributed message queue system. <http://github.com/twitter/kestrel>
18. Antlr. <http://wwwantlr.org/>