Pollux: Towards Scalable Distributed Real-time Search on Microblogs

Liwei Lin[†]
linliwei@mail.sdu.edu.cn

Xiaohui Yu[†,♮,*]
xhyu@yorku.ca

Nick Koudas[§]
koudas@cs.toronto.edu

[†]School of Computer Science & Technology, Shandong University, Jinan, China
[♮]School of Information Technology, York University, Toronto, ON, Canada
[§]Department of Computer Science, University of Toronto, Toronto, ON, Canada

ABSTRACT

The last few years have witnessed a meteoric rise of microblogging platforms, such as Twitter and Tumblr. The sheer volume of the microblog data and its highly dynamic nature present unique technical challenges for the platforms that provide search services. In particular, the search service must provide real-time response to queries, and continuously update the results as new microblogs are posted. Conventional approaches either cannot keep up with the high update rate, or cannot scale well to handle the large volume of data.

We propose Pollux, a system that provides distributed real-time indexing and search service on microblogs. It adopts the distributed stream processing paradigm advocated by the recently developed platforms that are designed for real-time processing of large volume of data, such as Apache S4 and Twitter Storm. Although those open-source platforms have found successful applications in production environments, they lack some critical features required for real-time search. In particular: (1) they only implement partial fault tolerance, and do not provide lossless recovery in the event of a node failure, and (2) they do not have a facility for storing global data, which is necessary in efficiently ranking search results.

Addressing those problems, Pollux extends current platforms in two important ways. First, we propose a failover strategy that can ensure high system availability and no data/state loss in the event of a node failure. Second, Pollux adds a global storage facility that supports convenient, efficient, and reliable data storage for shared data. We describe how to apply Pollux to the task of real-time search. We implement Pollux based on Apache S4, and show through extensive experiments on a Twitter dataset that the proposed solutions are effective, and Pollux can achieve excellent scalability.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing*; D.4.5 [**Operating Systems**]: Reliability—*Fault Tolerance*

*Corresponding author.

EDBT'13, March 18 - 22 2013, Genoa, Italy

General Terms

Algorithms, Performance, Experimentation

Keywords

Distributed processing, data stream, search, microblog, fault tolerance

1. INTRODUCTION

Microblogging services have gained tremendous momentum during the past few years. Twitter, the world's largest microblogging service provider, is estimated to have 500 million accounts as of January 2012, and this number is increasing at a rate of 10 accounts per second [9]. On average, there are over 200 million microblogs (hereinafter also referred to as tweets following Twitter's terminology) posted per day on Twitter alone, consitituting a huge volume of data. Much of the microblogging service's daily content is of a short temporal span (i.e., real-time content) but can contain significant societal, cultural, and commercial implications because of its diverse sources [23]. It is therefore highly desirable to provide real-time search capabilities over microblogs.

However, providing real-time search service over microblogs is a challenging task. First, the service must be able to deal with a vast amount of highly dynamic data. A peak rate of 25,088 tweets per second has been reported [5]. Massive indexing work therefore needs to be done continuously in order to make every tweet searchable. Conventional indexing methods adopted by DBMSs cannot handle the high frequency of data arrival and index update, because of the noticeable latency caused by frequent locking operations and disk I/Os. Second, the query results must be continuously updated when new tweets arrive, rendering it infeasible to simply adopt the methods developed for conventional Web search. Designed mainly for batch processing, Hadoop and other MapReduce variants cannot handle real-time processing well. Although there have been recent efforts on adapting Hadoop to real-time tasks [13], many technical challenges still remain [30].

The leading microblogging platforms and search engines, such as Twitter and Google, have started to offer real-time search services. Unfortunately, there is still very little work in the public domain. TI [15] is the first work that addresses the indexing and ranking issues for microblogs. However, TI only indexes a subset of the tweets (called the distinguished tweets) due to resource constraints, which may lead to inaccurate search results. More importantly, the inherently centralized architecture of TI makes it very difficult to scale out for managing the ever-increasing volume of data. Earlybird [14] is the most up-to-date solution deployed at Twitter; however, as a proprietary solution, some essential archi-

tectural issues of Earlybird are not made public (refer to Section 2.1 for details).

We propose Pollux, a scalable distributed search engine as a solution to the problem of real-time search over microblogs. Pollux performs two tasks: indexing and searching. It maintains a full index of all incoming tweets in real-time, while at the same time searches, ranks, and continuously updates the matching tweets for given queries. Pollux's architecture allows it to support various ranking strategies.

Pollux adopts the actor-model based distributed stream processing paradigm common to many recently developed platforms, such as Twitter Storm [8] and Apache S4 [30], which have found successful applications in production environments. Although these platforms excel at real-time processing of large volume of streaming data, they lack some critical features that are required to support real-time search. Those features are what Pollux is designed to provide. In particular:

(1) *Fault tolerance*: The current platforms only implement partial fault tolerance, and lacks effective failover management. When one node goes down, a standby node simply takes its place without restoring the data stored in it, resulting in incomplete data and search results. Since we aim to provide full indexing of tweets and accurate search results, this simplistic failover strategy does not suffice. We therefore design and implement a novel lossless failover management strategy that takes into consideration the specific requirements and non-requirements of the real-time search task.

(2) *Global storage*: Data in the current platforms is stored internally in each processing unit; they lack a global storage facility for storing and sharing information commonly used by many processing units, which some applications including real-time search require. We thus introduce in Pollux global storage units, which offer a simple yet powerful set of APIs that processing units can use to efficiently store and retrieve information. This new addition simplifies the task of the processing units, greatly reducing the excessive message passing when ranking the results. We show that it can be used to support a wide range of ranking strategies.

We implement Pollux based on Apache S4, but the design can be adopted to other similar platforms (such as Twitter Storm) as well. We describe in detail how Pollux can be used to support real-time search over microblogs.

Our main contributions can be summarized as follows.

- We design and implement Pollux, a real-time index and search system over microblogs. Pollux is decentralized, scalable, and elastic; it can be easily expanded to meet higher volume of data or query loads.
- We propose a failover management strategy to ensure lossless recovery, as required by the task of real-time search.
- We propose a global storage facility to supplement the local storage available within each processing unit, allowing search and ranking to be done efficiently.
- We perform extensive experiments with Pollux using real and synthetic Twitter data. Results demonstrate the effectivenss of our proposal and the high performance, high availability, and high scalability of Pollux.

The rest of the paper is organized as follows. In Section 2, we introduce the necessary background and provide an overview of the related work. In Section 3, we outline the requirements and non-requirements of a real-time search system, and present the design of Pollux. We present the failover strategy in Section 4. In Section 5, we describe the global storage model. We discuss the implementation of Pollux to support real-time search in Section 6. Experimental results are presented in Section 7. Section 8 concludes this paper.

2. BACKGROUND AND RELATED WORK

2.1 Real-time Microblog Search

TI [15] is the first work that publicly tackles the problem of real-time indexing and search for microblogs. TI essentially classifies the incoming tweets into two categories, distinguished tweets and noisy tweets, based on past queries. Tweets matching those past queries that are expected to appear again in the near future (based on statistics of past queries) are classified as distinguished. All other tweets are considered noisy. The distinguished tweets are indexed in real-time, while the noisy tweets are indexed only periodically in a batch fashion. Making this distinction between distinguished and noisy tweets allows the system to focus on the indexing of more popular tweets to meet the demands of high-frequency real-time queries.

However, TI has several drawbacks: (1) The classification is based on past statistics and may not accurately reflect what is going to happen in the future, especially in a microblog setting where things evolve very fast. It is very possible for tweets that will match future queries to be mis-classified as noisy, thus resulting in cases where relevant tweets are not returned (because they are not indexed as distinguished tweets); (2) Queries that do not overlap in keywords with previous queries will not be answered properly, because many of the matching tweets are likely to be classified as noisy, as they do not contain keywords appearing in previous queries. (3) The design of TI is inherently centralized; it is unclear how this design can scale up in a distributed setting to handle the extremely high volume of data and large number of users. Our proposal, in contrast, provides full indexing of all tweets in a tunable time window, ensuring that all relevant tweets can be returned in the search result. Moreover, our design of Pollux as a distributed stream processing system allows it to scale up really well.

Twitter has developed a distributed system called Earlybird [14, 3] enabling real-time search, but they reveal only the index organization mechanism (optimizations of *inverted index*) and the control of concurrent reads and writes in a single Earlybird server. Issues related to distributed search are not discussed in detail. In particular, these include (1) what to do when node failures occur; (2) how different ranking strategies are supported; and (3) how well the system performs as a whole. Pollux tackles all these issues.

2.2 Stream Data Processing Platforms

Multiple stream data processing platforms have been proposed in the literature [29, 12, 11, 10, 6]. Amini et al. [12] provides a review of the various systems, projects and commercial engines. Unfortunately, the use of those platforms is still restricted to highly specialized applications [30].

The recently developed Apache S4 [30] is a general-purpose, distributed, scalable stream-processing platform. In S4, the basic logical computing unit is called a PE (processing element). PEs are very light-weight; thus thousands of PEs may reside on a single node (usually a commodity PC). They interact with each other through event emission and consumption. A PE does one or both of the following after consuming an event: 1) emitting one or more events to other PEs, 2) publishing the results. Concisely speaking, an event in S4 is a 3-ary tuple: $\langle$*event type, a key-value pair, message entity*$\rangle$. As illustrated in Figure 1(a), after PE A emits an event to the data bus, two identical copies are routed to PE B and PE D

respectively, where the event copies are consumed. At startup, PEs register to listen to particular types of events. One can also specify PEs to listen to particular sub-sets of the same event type via more detailed key-value pairs. A *message entity* is an object that represents the message data. Event passing usually forms a pipe-line (see Figure 1(b)), through which events flow from top to bottom. Loops are not technically forbidden in S4, but are seldom needed.

Storm [8], open-sourced by Twitter, is another distributed stream processing engine serving similar purposes as S4. Both Storm and S4 lack support of lossless failover management and global storage, which are required to support real-time search. In our work, although we focus on extending S4 with these new capabilities, we expect that similar ideas could also be applied to extending Storm.

2.3 Distributed Key-Value Store

Various distributed key-value stores have been proposed in recent years. Dynamo [17], Cassandra [27] and PNUTS [16] are all distributed, high-available stores. However they are disk-based, and thus have considerable latencies. RAMCloud [31] is online, but utilizes disks to store replicas, causing noticeable downtime in the event of a failure. In [32], Paxos is used to build a strongly consistent distributed data store, but the resource utilization is too low to be acceptable for our task: up to k node failure is tolerable when $2k+1$ replicas are deployed, e.g., 3 replicas to tolerate 1 node failure and 5 replicas to tolerate 2 node failures. MemCached [4] is a distributed memory object caching system, which lacks effective fail-over management.

Compared with existing key-value stores, Pollux's global storage are online and highly available, tailor-made for the task of real-time search to support a wide range of ranking strategies.

2.4 Failover Management for Stream Processing

A number of proposals for failover management in stream processing exist. *Fast-recovery-based strategy* [31] causes noticeable downtime. The work in [34, 20] focuses on *precise recovery*, but it incurs high run-time overhead as it aims at strict synchronization of nodes and their backups (in the sense that they always have the same states). *Passive stand-by*, *upstream recovery* and *active stand-by* are failover strategies that provide lossless recovery, but they rely on order preservation to ensure their correctness [20]. Multi-upstream to multi-downstream flow [21] is introduced to overcome network problems but again relies on order-preservation. In contrast, Pollux utilizes a failover strategy that works without requiring order preservation.

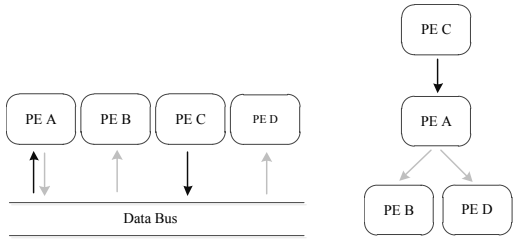
3. REQUIREMENTS AND SYSTEM DESIGN

3.1 Requirements for Real-Time Search

Before deciding on a particular software stack to provide a solution, we first look at the requirements for real-time search over microblogs[1], which are summarized as follows.

- **Real-time processing:** The microblogs should be indexed in real-time, and queries should be answered promptly. In addition, query results must be updated when new microblogs arrive.
- **High availability:** The service must be highly available, covering both scheduled and unscheduled downtime.

[1]In what follows, microblogs are also referrred to as tweets using Twitter's terminology.



(a) Data-bus view (b) Pipeline view

Figure 1: Data-bus view and pipeline view of the S4 event flow

- **Lossless recovery:** While having imprecise results (such as the appearance of irrelevant microblogs in the results) is a fact of life in information retrieval systems as it is unlikely to have a "perfect" ranking function, losing a whole chunk of data (and therefore leaving them unsearchable) due to hardware failure is unacceptable.
- **Scalability and elasticity:** We need to be able to add incremental capacity to our system to handle growing amount of requests for the applications. System performance should improve proportionally to the hardware capacity added and the expansion should bring minimal overhead and no downtime.
- **Ranking support:** We should provide a framework for real-time indexing and search that supports the implementation of various ranking functions, as the search results should be effectively sorted to present information relevant to the queries.

It is also worth pointing out the non-requirements:

- **Transient inconsistencies:** There is a trade-off between availability, performance and consistency. We argue that for microblog search, it is acceptable to have some transient inconsistencies in the search results. For example, if two tweets t_a, t_b and a query searching for t_a and t_b arrive almost simultaneously, then it is acceptable that t_b is instantly searchable and returned while t_a is not for the moment and will be returned after a very short delay. As will be shown in the sequel, by relaxing the requirement on consistency, it is possible to achieve higher performance and availability.
- **Order preservation:** This is derived from the non-requirement on consistency. Messages do not need to be processed in exactly the same order as they arrive; nor do they have to arrive at a processing element in the same order as they are emitted.

3.2 Design of Pollux: Architectural Overview

Pollux consists of two components: a faster in-memory real-time processing component, which takes care of indexing and search of recent tweets (within a configurable time threshold), and a slower on-disk batch processing component, which is responsible for tweets from the more distant past. For a given query, only recent tweets returned by the real-time component are first presented to the user, who then has the option of viewing older tweets (to be returned by the batch component) should he/she so wish. This two-tier design is based on the observation that in most cases, users of microblogging services focus more on recent tweets [25, 18]. The batch component can be implemented in a traditional fashion using full-text search engine such as Apache Lucene. The cutoff point

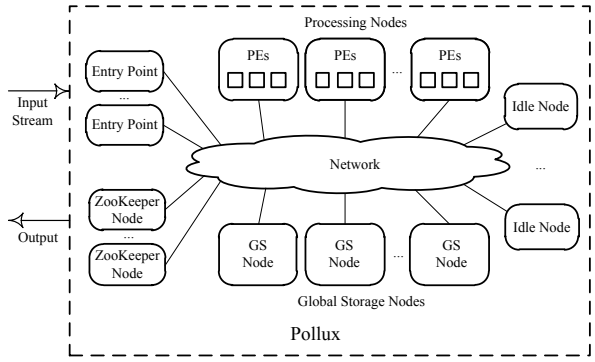


Figure 2: Pollux's architectural overview: processing nodes, global storage (GS) nodes, entry points, ZooKeeper nodes, and idle nodes

(in time) between the two components can be configured based on the amount of available system resources, user requirements, etc. In this paper, we only focus on the discussion of the real-time component, as it presents more technical challenges. Unless otherwise pointed out, the term Pollux refers to specifically the real-time component in the rest of the paper.

Pollux is built based on the S4 platform (an overview of S4 is provided in Section 2.2), but the ideas could also be applied to extend other stream processing systems such as Storm. Pollux employs a decentralized architecture, i.e., there is no master node; all jobs are done through the coordination between processing nodes, sometimes with the help of ZooKeeper [2]. There are four types of functioning nodes in Pollux: the *processing node*, the *global storage node* (or *GS node* for short), the *entry point node*, and *the ZooKeeper node*. The processing nodes hold the PEs, while the GS nodes hold the global data. The entry point nodes receive input streams, and distribute them to the processing nodes. ZooKeeper monitors the status of the system, reports node failures, and records system running statistics. There are some idle nodes waiting in a pool; if a failure occurs to a node of type X, an idle node immediately joins and serves as a node of type X. All nodes are connected through high-bandwidth network to ensure performance.

The failover strategy, the design and usage of the global storage, and the implementation of real-time search over microblogs are the three key issues we shall explore in the following sections.

4. FAULT TOLERANCE

The current implementations of the stream processing platforms (S4, Storm, etc.) simplify the task of failover management by assuming that lossy failover is acceptable. Take S4 for an example. Upon a node failure, the PEs running on that node are automatically re-allocated on a standby node. The states of the PEs, which are stored in memory, are lost during the hand-off. This leads to incorrect and unacceptable output. We therefore have to come up with a failover strategy to ensure timely output of correct results in the event of a node failure.

As analyzed in Section 3, the task of real-time search requires high availability and lossless recovery, while transient inconsistencies can be tolerated and order-preservation is not required. These characteristics must be taken into consideration when designing the failover strategy. Moreover, the structure of Pollux is symmetric, meaning that every processing node hosts a mixture of different types of PEs. Compared with an asymmetric design where each processing node is dedicated to a particular type of PE, this symmetric design provides better load balancing in that different types of PEs exhibit different resource usage profiles (e.g., some PEs are CPU-intensive whereas some PEs are memory-intensive). This structure of Pollux must also be taken into account for failover management.

4.1 Existing Failover Strategies

The existing failover management methods (e.g., [34, 20, 22, 21, 26, 33]) roughly fall into three categories:

- *Disk-based recovery*. Work such as [26, 33] saves checkpoints of PEs' states on a distributed, replicated file system such as Hadoop HDFS, and restores them upon a node failure.
- *Online-standby-pairs recovery*. Hwang et al. [20] propose three types of online recovery: *active standby*, *passive standby* and *upstream recovery*. Active standby ([34, 20, 21]) utilizes a secondary processing node for each processing node (the primary). A secondary node receives the same events as the primary to update its state, but never emits events. Upon failure of the primary, the secondary node becomes the primary. Passive standby ([20, 22]) utilizes a same secondary node, but differs from active standby in that the secondary is updated through checkpoints periodically by the primary. Upstream recovery keeps those events in the upstream nodes which are not thoroughly processed by the whole system and removes them otherwise.
- *Reliable WAL-based recovery*. Some systems employ BookKeeper [1], a reliable WAL service, for recovery. A secondary node reads and performs the update from the WAL and upon the primary's failure becomes the primary.

Among the above methods, active standby fits into our setting. First, for disk-based recovery, data cannot be seamlessly recovered without noticeable delay. Kwon et al. [26] notes "*it could thus recover a 128MB HA unit in under two seconds*", and that is basically one minute for every 4GB of data. Second, the problem with upstream recovery is that the state updated by the events that are already removed can never be restored. Third, reliable WALs themselves requires extra nodes as inner replication and performs no better than active standbys. Lastly, secondary nodes as active standby can be altered to produce output events as well (in order to achieve better over-all performance) while secondary nodes as passive standbys cannot.

We need to come up with a new active standby failover strategy because: (1) some active standby strategies focus on precise recovery ([34, 20]), which is too strict for our task and may lead to performance degradation; (2) other active standby strategies ([20, 21]) rely on order-preservation to ensure their correctness. Our approach aims at lossless recovery over orderless events, working quite differently from these strategies.

4.2 Overview of the Failover Strategy

We propose a novel failover strategy matching our requirements for the stream platforms (S4, Storm, etc.), using S4 as the basis for our implementation. For the failure model, we assume that a failure can take one of the following two forms: a node failure, or a failure of the connection between a node and the network switch. We do not consider Byzantine failures where faulty components can behave in arbitrarily erroneous ways.

The intuition behind Pollux's failover strategy is that, if we run two (or more) isolated sets of identical online nodes (both processing nodes and GS nodes), then if a node in one set fails, its

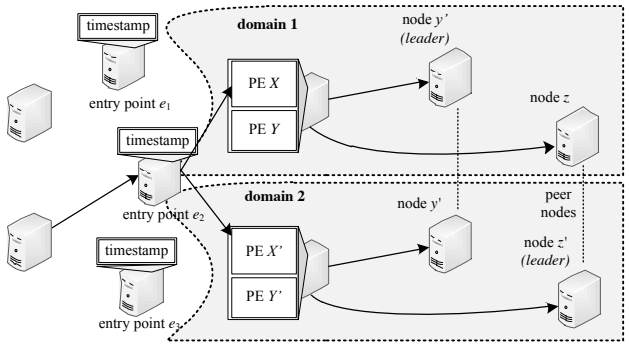


Figure 3: Illustration of entry points, domains, peer nodes, and the leader for each set of peer nodes

peer node in the other set can take over the work of the failed one and help recover the data. Here we refer to such a set of processing nodes (or GS nodes) as a *domain*. For example, nodes shown in Figure 7 can constitute a domain. A domain does not contain ZooKeeper nodes as ZooKeeper nodes are self-replicated. Domains are self-contained; events do not flow across domains except during recovery.

A Pollux instance can be configured to have k domains and thus can withstand $k-1$ node failures. Note that the domains are all online; thus they are redundant. This is a trade-off we make between resource usage and high-availability. Without loss of generality, in the following discussion we focus on the case of $k = 2$ and denote a PE (or node) x's peer PE (or node) as x'.

We introduce entry points, which serve as the starting points for reliable stream processing. An entry point receives input streams and replicates them to k copies. It then stamps the copies with the same time and emits the copies into k different domains so that each domain gets one. Entry points are stateless; if one fails, an idle node simply becomes an entry point and starts to serve.

In the next subsection, we shall explain the failover strategy in detail.

4.3 Failover Strategy

Pollux utilizes the always-online replica strategy for PEs: peer PEs serve as online replica for each other. At start-up, every set of peer nodes, with the help of ZooKeeper, elect a *leader*, which is responsible for collecting state information among peer nodes. If such a leader fails, the remaining peer nodes quickly (typically within a few milliseconds) elect a new leader. One leader is elected among each set of peer nodes; thus multiple leaders exist for different sets, as shown in Figure 3.

The failover process is non-trivial because the parallel domains may not have exactly the same state. We use the following typical sample scenario with two domains to illustrate this. The system state at the moment of node failure is shown in Figure 4(1). PE U is about to send event a to PE Y which has already received event b; PE X has received event h. In the other domain, PE V' is about to send event b to PE Y' which has already received event a; PE X' has received event h. When a failure occurs, both node 1 and event a are lost. Replicate only node $1'$ back to be the new node 1 could not get event a back for PE Y. Under the assumption of order-preservation, a comparison between the order number of the last event from PE Y and PE Y' can immediately reveal how many and exactly what events are lost. When it is orderless, however, lots of work need to be done.

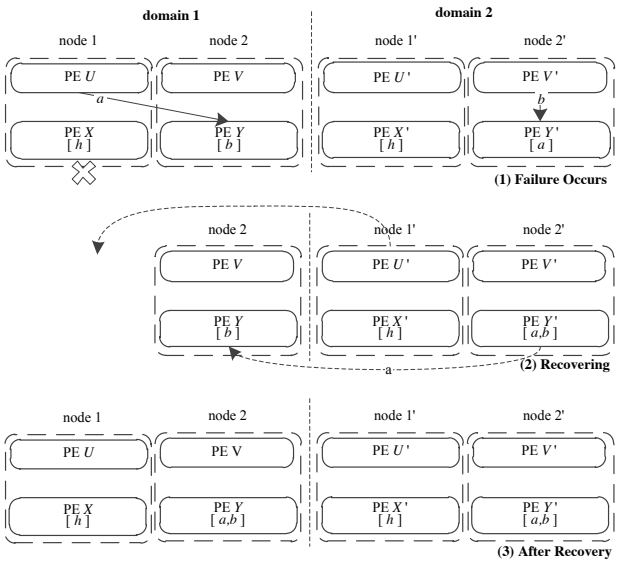


Figure 4: A typical failure-recovery process (illustrated with two domains): (1) PE U is about to send event a to PE Y when failure occurs to node 1; PE U, PE X and event a are lost; (2) node $1'$ replicates its internal state to the new node 1, and PE Y' remedies event a to PE Y via slot-reconciliation; (3) after recovery, no data is lost.

The basic idea of our strategy is to use an auxiliary data structure maintained at each node to facilitate periodic synchronization between peer nodes. That is, a processing node not only applies events that can change the state of a PE, but also captures them in *slot*s. A *slot* is a 5-ary tuple:$\langle$ time period, event count, hash value, identifiers, events$\rangle$, where *time period* specifies a range of timestamps, and any event with a timestamp falling in that range should be directed to the same slot; *event count* is the number of events received during that time period; *hash value* is 512-bit long and is the XOR of the current hash value and the hash value of the new incoming event; the field *identifiers* keeps track of additional flags; *events* is a set of the events associated with this slot. The *slot digest* of a slot tuple consists of all fields except *events*.

For synchronization between peers, periodically (the interval is configurable) *slot-reconciliation* is initiated by a leader. It requests and compares the slot digests from all peer nodes. If all digests are identical, the corresponding slots are cancelled (erased), as shown by Figure 5. If there is at least one digest that is different from the others, then the slots remain. Normally this is due to the latency of event passing through the PEs: an event may have been received at some peers but not the others. However, if the problem persists for a considerable amount of time, then it is clearly an indication of possible problems with the system. We use a configurable *latency threshold* for this purpose. If some slot lives longer than this threshold, then during slot-reconciliation, Pollux performs a round of forced synchronization, where the leader will compute a union of all events associated with this slot from all peer nodes, and then send them to each peer (modulo the events already in there: note event a in Figure 4(2)). This ensures that all slots are now back to the same page; see Figure 4(3).

Two notes about the strategy: (1) It is possible that a slot for the same period re-appears after its being cancelled (because another event from the same period arrives), but its digest and event data should differ from the previous one's; (2) If two identical events

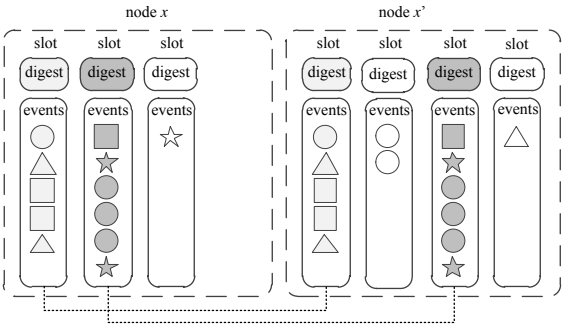


Figure 5: Slots-reconciliation: two of the slots can be cancelled; the others cannot

arrive, only the first one should be buffered in the slots and the other discarded.

In our example above, when node 1 fails, node $1'$ replicates its internal state to the new node 1. The next rounds of slot-reconciliation between node 2 and node $2'$ would discover that, for node $2'$, the slot containing event a lives longer than the latency threshold and is not cancelled. Thus, events including a in the slot are emitted to node 2, and the two slots can then be cancelled. In the end, no data are lost.

The following theorem formally justifies the correctness of the proposed failover strategy. Interested readers are referred to [28] for the proof.

THEOREM 1. *For a given time window $w \gg t_{last} - t_{first}$, where t_{first} and t_{last} are the arrival timestamps of the first event and the last event in any slot respectively, a given filter function $filter_w(e)$, discarding an event e if it has appeared in w, a given number of domains k, and a number of failed peer nodes $f < k$, the failover strategy described above ensures that all peer nodes will process the same set of events.*

In summary, Pollux utilizes two or more domains in which the replicas are distributed. PEs are backed by the always-online-replica strategy, and no event or data are lost due to slots-reconciliation. At the expense of more resource usage, Pollux manages to react promptly to node failures, leading to no downtime and preventing data loss.

5. GLOBAL STORAGE

5.1 The Problem

Existing stream processing platforms (such as S4, Storm, etc.) store data locally in each processing unit; they lack a global storage facility for storing and sharing information commonly used by many processing units, which the task of real-time search requires. For example, to effectively rank a list of tweets matching a query, the author's information for each of the tweets may be needed. Such information is shared by many processing units, and it is unwise to store them locally in each unit, because doing so would cause huge update overhead and potentially many consistency problems. As such, there is a need for a global storage facility to store the shared data. Without this facility, ranking strategies except for the most straightforward one, which orders the tweets by their timestamps, are either impossible or too complicated to implement.

A global storage meets our requirements: PEs read the globally shared data from or if necessary write the shared data to the global storage. Key principles that must be considered are:

- **Efficient data access:** Since data is accessed at high rates during the computing process of PEs in real-time, storing data in main memory is preferrable to disk-based solutions to reduce latencies.
- **High availability:** It would conflict the whole system's vision if the storage fails and cannot resume serving within a very short period of time.
- **Scaling well**: Data that need to be kept in the storage might be of large volume, making a distributed model desirable.
- **Simple data models:** Real-time search on microblogs does not rely on complex data model such as relational data models. Simple data models would suffice.

One possible solution is to implement such a global storage as a set of regular S4 PEs. For example, we can add several StoragePEs, serving data-access requests. PEs emit events to StoragePEs requesting data, and StoragePEs emit data back to the requesting PEs. However, this solution is not the best choice due to the following problems: (1) Many rounds of event passing are required in order to accomplish a single process of multiple point data access. This leads to more network traffic overhead and increased latency for data fetching; (2) Since a requesting PE may receive multiple events from different StoragePEs, each event containing a piece of the data, the requesting PE has to constantly check whether all data needed have arrived before the computation continues; (3) StoragePEs are exposed to the application layer, and the application has to explicitly manage the failover and load balancing issues of the StoragePEs.

5.2 The Solution

We propose a data store solution with potentially unbounded storage space (only limited by the amount of available memory in the cluster), and is accessible to every PE in S4. The entire store is partitioned and reside on multiple GS (global storage) nodes with no data overlap; but externally the global storage is exposed to the application PEs as a whole. We adopt the key-value data model as it is general enough to handle different types of data.

The API of the proposed global storage is fairly simple, consisting of the following methods:

- test-and-put(key, value, version)
- test-and-put-all(keys, values, versions)
- get(key)
- get-all(keys)
- increase(key, value, identifier)
- increase-all(keys, values, identifiers)

Some notes about the API. The *test-and-put()* method requires a *version*, and simply returns if a newer version already exists. The *increase()* method requires an *identifier*, which identifies the source of the increment. Increments caused by the same source to the same key will be executed only once. The *test-and-put-all()* method is a short-cut that combines many *test-and-put()* calls. Any call to the global storage is a standalone call, and does not rely on any prior calls.

Those six methods are all synchronized, meaning that when a PE makes a call to any of the six methods, it would block until the data is written to or read from the storage. This is reasonable because data is essential for a PE to continue its execution. In addition, this simplifies the programming model, freeing programmers from writing callbacks.

5.3 Implementation Issues

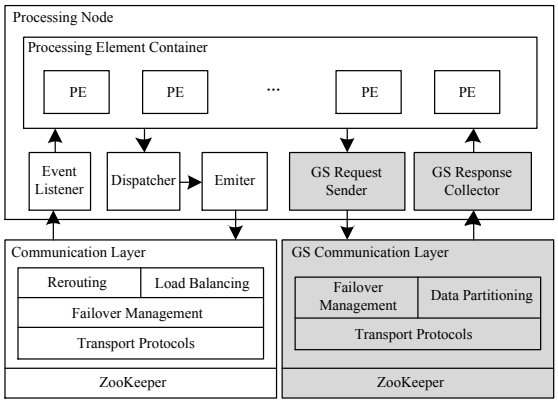


Figure 6: The modified processing node in Pollux: the global storage components are marked with shaded background

Figure 6 shows the modification made to the original processing node. The original communication layer sends packages containing the event messages, and a similar structure is employed in implementing the global storage communication layer. The Pollux global storage is implemented partially in a client and partially in a server. Each client has the over-all information on the GS nodes forming the global storage, and thus can forward a PE's request to the appropriate nodes. Sometimes, a *get-all()* request involves data from more than one GS node. In such cases, the *GS response collector* combines the data retrieved from different nodes, and hands them as a whole to the PE that has originated the request. The global storage communication layer delivers messages back and forth to the corresponding GS nodes. On each GS node, all data are kept in memory and all operations are atomic.

A design choice that has to be made is the relationship between the GS node and the processing node. One possibility is to integrate the GS node into the processing node. However, it is important to note that processing nodes and GS nodes usually have very different needs for scaling, and the integration of the two types of nodes would make it inflexible for them to scale independently. Therefore, Pollux keeps processing nodes and GS nodes separate (see Figure 2). We use the same failover strategy proposed in Section 4 for both PE and GS nodes.

6. SUPPORTING REAL-TIME SEARCH

In this section, we discuss how real-time search is supported by Pollux. In particular, we discuss how tweets are indexed in real-time, how queries are answered in real-time, and how the tweets are ranked with the support of global storage.

6.1 Real-Time Indexing

*TweetEntrancePE*s and *WordPE*s are built to index the tweets for later search. TweetEntrancePEs serve as the entrances where tweets flow in, and WordPEs maintain the *inverted indices*.

Exactly one TweetEntrancePE is deployed on each processing node. A tweet is wrapped in a *TweetEntityEvent*, whose event fields (*event type, key-value pair* and *message*, introduced in Section 2.2) are shown in Table 1. When a *TweetEntityEvent* arrives (indicated by ① in Figure 7; similar notations are used hereinafter), the TweetEntrancePE consumes it by splitting the content of that tweet into unique words, then wraps each of the words into a *WordEvent*, and finally emits all of the *WordEvent*s (②). Then the TweetEntracePE saves the full content of a tweet into the global storage (③).

Many WordPEs may exist on a processing node. When a *WordEvent* arrives, the corresponding WordPE fetches the tweet id (*tid*) from the event and adds it into the *inverted index* maintained in local memory. Thanks to S4's fast and smart PE-creation, no predefined vocabulary has to be setup in advance. WordPEs can be created on demand when the words are seen for the first time. As time elapses, the list in each WordPE keeps growing. Cleaning-up can be done periodically to keep the lists in the WordPEs slim, as Pollux focuses only on recent tweets. Efficient index organization and concurrent read/write control are not the focus of Pollux; interested readers can refer to [14] for details.

6.2 Query Processing

We now explore the event flow for query processing. Each processing node hosts one *QueryEntrancePE* to serve queries wrapped in *QueryEntityEvent*s. A QueryEntrancePE emits *QueryPieceEvent*s describing the query piece after splitting the queries into words; each word forms a query piece (⑤). Unlike words in a tweet, query pieces need to be aware of each other so that later during query processing their search results can be combined. Therefore we mark each query piece by the query id (*qid*) along with with an n/m notation, where m is the number of keywords in the query, and n is the position of the keyword in the original query.

Upon receiving a QueryPieceEvent, the WordPE wraps the $query\ id$, $user\ id$(*uid*), n/m notation from QueryPieceEvent together with the $list\ of\ tids$ into a *MergeEvent* and then emits it. A MergeEvent is defined in Table 1.

The m lists associated with the m pieces of the query are collected and merged by a *MergePE*. The MergePEs are evenly spread to each node in order to better distribute the load across server nodes and/or processors in the cluster. We ensure that all m pieces (*MergeEvent*s) of the same query flow into the same MergePE(⑥) through hashing. Currently we use the "AND" semantics for calculating the intersection of the m lists in the MergePE, but other semantics such as "OR" can be adopted as well. After the MergePE merges the m lists into a final list, it emits a *RankEvent* if the list contains more than one tweet. If the final list contains no tweet or it has been reported that at least one of the m lists is empty (and thus the final list must also be empty), then an *OutputEvent* is emitted. In the latter case, a MergePE does not even have to wait for all of the m lists before emitting the OutputEvent.

A RankPE receives a *RankEvent* (⑦) and ranks the tweets according to some pluggable strategy. The RankPE resorts to global storage to retrieve the information needed (⑧). This may happen several times if the ranking procedure is iterative.

Finally the RankPE emits its result in an *OutputEvent* for some OutputPE to report the list (⑨). *RankEvent* and *OutputEvent* are also defined in Table 1.

To summarize, for a given query, QueryPieceEvents are generated and flow into WordPEs, leading them to emitting MergeEvents containing the ids of the tweets matching the keywords. Then at MergePEs the tweet ids are merged into a single list and at a RankPE the list is ordered. Finally the results are reported by OutputPEs.

6.3 Supporting Ranking Strategies

Various ranking functions can be supported by Pollux with global storage. Example strategies for ranking tweets or users include those presented in *TI* [15], *Ranking Twitter Users* [25] and *Learning to Rank of Tweets* [19]. Without global storage, all strategies except for the naive one that orders tweets by time, are too compli-

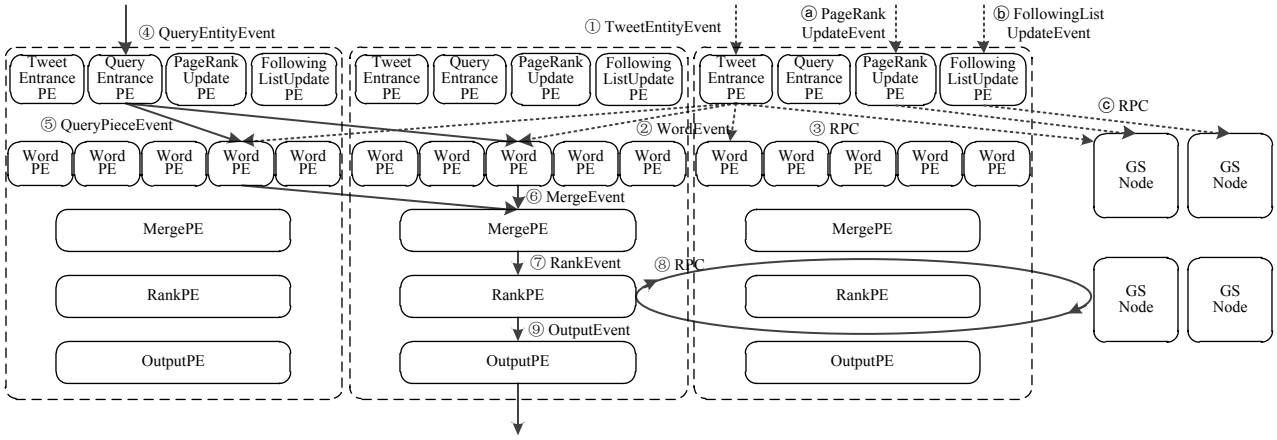


Figure 7: Event flow in Pollux

Table 1: Events in Pollux

Section	Event Type	Keyword-value Pair	Message Entity
6.1	TweetEntityEvent	null	TweetEntity⟨tid, uid, tree_root_node_tid, time_stamp, content⟩
6.1	WordEvent	⟨"word",some word⟩	⟨tid, time_stamp⟩
6.2	QueryEntityEvent	null	QueryEntity⟨qid, uid, time_stamp, content⟩
6.2	QueryPieceEvent	⟨"keyword",some keyword⟩	⟨uid, qid, n/m notation, time_stamp⟩
6.2	MergeEvent	⟨"queryID",qid⟩	⟨qid, uid, n/m notation, list of tids⟩
6.2	RankEvent	⟨"queryUID",uid⟩	⟨qid, uid, list of tids⟩
6.2	OutputEvent	⟨"queryID",qid⟩	⟨qid, uid, list of tids⟩
6.3	PageRankUpdateEvent	null	⟨uid, PageRank, version⟩
6.3	FollowingListUpdateEvent	null	⟨uid, following_list, version⟩
6.4	QueryEntityEvent	null	QueryEntity⟨qid, uid, time_stamp, expiry, content⟩

cated to implement, as discussed in Section 5.1. In what follows, we shall briefly mention some of the strategies, and list (but not explain in detail for every strategy due to space limitation) in Table 2 how information can be retrieved or updated to support those strategies.

TI: the Twitter Index. TI [15] proposes a ranking function to sort the result list of tweets. It considers four factors: 1) the tweet author's PageRank: tweets from authoritative authors should rank higher; 2) the popularity of the topic: hotly-discussed tweets should rank higher; 3) the timestamp: more recent tweets should rank higher; 4) the textual similarity between the query and the tweet. If we choose to support TI's ranking strategy, then two types of pluggable ranking-assistant PEs, i.e., the PageRankPE(ⓐ) and the FollowingListUpdatePE(ⓑ) in Figure 7 can be introduced in Pollux. PageRankPEs update the users' PageRank information in the global storage (ⓒ); FollowingListUpdatePEs update the users' following list information in the global storage(ⓒ). A topic tree's popularity is updated by TweetEntrancePEs. Scores are calculated in RankPEs following the score function proposed in TI; any global information needed are retrieved from the global storage.

Ranking Twitter Users. In [25], Twitter users are ranked by the number of followers (R_F), their PageRank values (R_{PR}), and the total number of retweets (R_{RT}). Interestingly, ranking results by R_F and by R_{PR} are similar, but R_{RT} is different, indicating a gap between the number of followers and the popularity of one's tweets, which brings forward a new perspective on influence in Twitter [25]. R_F and R_{RT} are essentially counters, which can be perfectly supported by the GS's atomic increase() operation.

Learning to Rank of Tweets. Duan et al. [19] explores several features that might affect the rank of a tweet, and determines using *learning to rank* algorithms the best set of features. It is demonstrated that the best combination is whether a tweet contains URL or not, the length of the tweet, and the account's authority. The account authority feature in [19] contains many aspects which all can be well supported by our global storage model, but only PageRank is demonstrated in Table 2. It is pretty straightforward to obtain the length of a tweet or whether URLs exist if such information is stored as $\langle length, I_{URL} \rangle$ pairs (where I_{URL} is a 0/1 indicator for the presence of any URL) in the GS when the tweet is indexed.

6.4 Resumed Query and Continuous Query

If node failures occur when a query is in progress, some queries are interrupted, and thus must be resumed. Upon a node failure, one of the remaining domains resumes the query by restarting it from the very beginning. This can be done without further data exchange between domains as long as the node failure is detected by Pollux, as all queries are essentially replicated into all k domains (although initially only in one domain the QueryEntrancePE conducts the event flow to answer that query). Query states inside other PEs, including MergePEs, RankPEs, and OutputPEs, do not need to be recovered because the query is replayed at another place and query states are "recovered" along the way at those PEs.

Pollux also offers support for continuous queries, i.e., the query results can be updated as new tweets arrive. To achieve this, we redefine the QueryEntityEvent, as shown in Table 1, by adding an $expiry$ attribute to indicate the time of expiry for a query. Query re-

Table 2: Example ranking strategies supported by Pollux

Work	Feature	Retrieval	Update	Notes
[15]	PageRank	*get("pr"+uid)*	*test-and-put("pr"+uid, PageRank, version)*	
	TreePopularity	*get("pr"+uid)*	no operation	if *tree_rootnode_tid* is *null*
		get("tp"+tree_rootnode_tid)	*increase("tp"+tree_rootnode_tid, PageRank, tid)*	if *tree_rootnode_tid* is not *null*
	Timestamp	*tweet.getTimestamp()*	no operation	
	Similarity	tweet = *get("tc"+tid)*	*test-and-put("tc"+tid, tweet, 1)*	cos(tweet, query)
[25]	R_F	*get("rf"+uid)*	*increase("rf"+uid, follower uid)*	
	R_{PR}	same as PageRank in [15]	same as PageRank in [15]	
	R_{RT}	*get("rrt"+uid)*	*increase("rrt"+uid, tid)*	
[19]	AccountAuthority	same as PageRank in [15]	same as PageRank in [15]	
	TweetLength	tweet = *get("lurl"+tid)*	*test-and-put("lurl"+tid, len+0/1, 1)*	length and 0/1 indicator
	URL	tweet = *get("lurl"+tid)*	*test-and-put("lurl"+tid, len+0/1, 1)*	length and 0/1 indicator

sults will be continuously updated until its expiry. We also modify WordPE's inner data structure by adding a query set keeping track of the queries that are still active. When new matching tweets arrive, Pollux appends them to the original result of the active queries.

6.5 Load Balancing and Elasticity

We adopt Constant Hashing[24, 17] to make Pollux very elastic, i.e., Pollux is able to automatically add or remove nodes from the cluster depending on the system load. Intuitively, Consistent Hashing is based on mapping each object to a point on the edge of a circle, and mapping each node to many arcs separated by the some pseudo-randomly-chosen points. A node stores an object if the object "falls" on the node's arc. New nodes joined "steal" arcs from other nodes, benefiting load balancing and elasticity in the following way: a overloaded node can disperse its load to other nodes, and if most nodes are overloaded, new nodes can be added. Under the coordination of ZooKeeper, processing or GS nodes join and participate in the circle; all information about the circle are stored in ZooKeeper. In our implementation, we set up some threshold exceeding which a certain number of nodes would join. It is our on-going work to develop some sophisticated load balancing strategies, such as one that is aware of hot topics.

7. EXPERIMENTAL EVALUATION

7.1 Experiment Setting

We carry out experiments to evaluate the performance of Pollux as well as the effectiveness of the failover strategy and the GS facility. Since a full Twitter feed is unavailable to us, we use a dataset provided by Twitter for academic research [7]. This dataset contains approximately 16 million tweets sampled between January 23rd and February 8th, 2011, which are reduced to 7,847,176 tweets by filtering out non-English contents. We further synthetically generate the user graph and the tweet graph following the distributions reported in [25]. The user graph contains 3,040,392 users. To boost the volume of data, tweets are repeatedly fed to the system (but each time with a different tweet id and user id). We form the queries by randomly drawing keywords from the tweets and combining them, with the number of keywords in a query following Zipf's Law: the percentage of 1-keyword queries, 2-keyword queries and multi-keyword (3 to 10 keywords) queries are 60%, 30% and 10%, respectively.

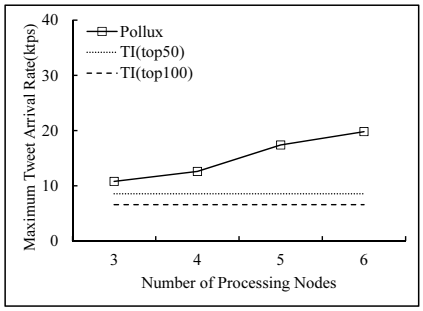
We deploy Pollux on a cluster of 13 Dell R210 servers with Gigabit Ethernet interconnect. Each node has one 2.4GHz Intel Xeon X3430 processor and 8GB of RAM. Out of the 13 nodes, 10 nodes serve as processing nodes or GS nodes according to different experiment settings, and the remaining 3 nodes serve as the entry points and host the ZooKeeper. Unless otherwise specified, we adopt TI's ranking function in the experiments. Each experiment is repeated ten times and the average result is reported.

In the experiments below, we shall evaluate how well Pollux scales, the effectiveness of the failover strategy, the performance of the global storage, and the over-all performance of query processing with Pollux. We do not make a quantitative comparison to Earlybird [14] for the following reasons: (1) Earlybird is a proprietary solution, and not enough details are made public for us to replicate its implementation or to make a meaningful comparison; (2) The paper [14] itself only presents the performance results on a single server; it does not present any evaluation results in a cluster setting except for briefly mentioning that "... typically observe a 10 second indexing latency (from tweet creation time to when the tweet is searchable) and around 50 ms query latency" without any details on the number of servers, etc.; in contrast, we focus on the scalability, failover managment, support of various ranking strategies and query performance.

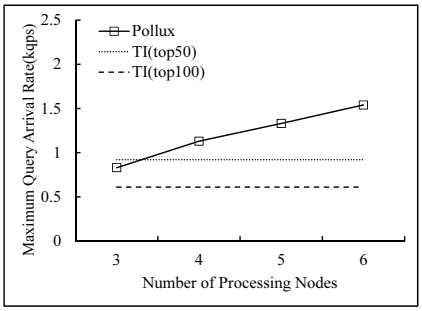
7.2 Scalability

To evaluate the scalability for one domain, we start with 3 processing nodes, and add more processing nodes to the cluster one by one until all 6 processing nodes available are used. The number of GS nodes is fixed at 4. We record both the maximum sustainable tweet arrival rate (with the query arrival rate fixed at 0.4kqps) and the maximum query arrival rate (with the tweet arrival rate fixed at 2.5ktps) for varying number of nodes, and the results are shown in Figure 8(a) and Figure 8(b). The maximum sustainable rates are determined by observing whether an overflow has occurred in a processing node's buffer queue which is used to buffer events yet to be processed by the PEs on that node. It can be observed that both the maximum tweet and query arrival rates increase at a linear rate as more nodes are added, demonstrating the superior scalability of Pollux.

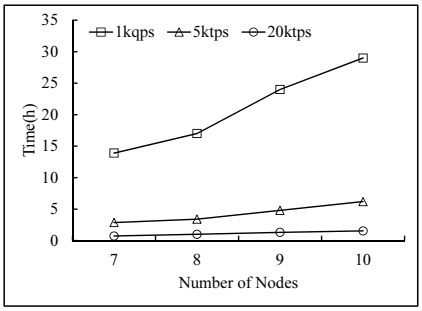
For comparison, we have implemented TI, an inherently centralized solution. We test TI to obtain the maximum query arrival rate and the maximum tweet arrival rate under the same conditions outlined above. The two optimizations[15] for TI are turned on, and the configurable number of tweets in the query result are set to 50 and 100 respectively. The experiments are performed on one of the servers. Since the architecture of TI is inherently centralized, the dotted lines of TI in Figure 8(a) and Figure 8(b) are flat. While TI is able to sustain a higher level of tweet and query arrival rate then Pollux when the number of nodes is low, it is outperformed by Pollux when the number of nodes increases, as Pollux is able to scale out to handle larger volume of tweets/queries.



(a) Maximum sustainable tweet arrival rates with query arrival rate fixed at 0.4kqps

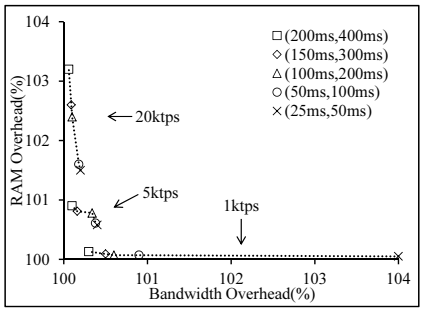


(b) Maximum sustainable query arrival rates with tweet arrival rate fixed at 2.5ktps

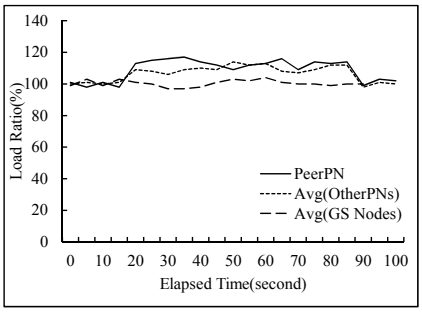


(c) Maximum time window for real-time indexing

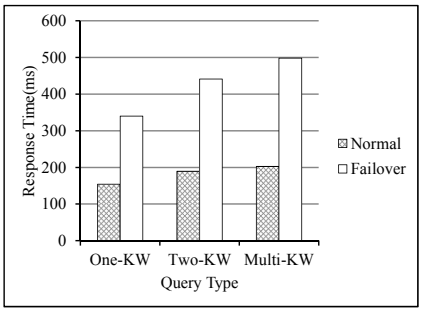
Figure 8: Scalability of Pollux



(a) Runtime overhead for varying slot-reconciliation interval and latency threshold values



(b) Load dispersion during recovery



(c) Response time of queries: running normally vs. resumed after failover

Figure 9: Failover performance

Another experiment is conducted to test for how long tweets can be retained in Pollux's real-time component. We start with 7 nodes; nodes are then added one by one. The ratio of the number of processing nodes to the number of GS nodes is optimized individually, i.e. it is 5/2 for 7 nodes available, 5/3 for 8, 6/3 for 9 and 6/4 for 10. We set the tweet arrival rate at 1ktps, 5ktps, and 20ktps, respectively. When the average memory usage of all PE nodes' and all GS nodes' reaches a steady state we track the time a tweet can stay in memory before having to be removed. As can be observed from Figure 8(c), the capacity grows roughly linearly with respect to the number of nodes.

7.3 Failover Performance

Several experiments are conducted to evaluate the performance of the failover strategy at both runtime and recovery. In this set of experiments, the 10 nodes are divided into 2 domains, each domain containing 3 processing nodes and 2 GS nodes.

Figure 9(a) shows the relationship between memory overhead and bandwidth overhead at runtime as the slot-reconciliation interval and latency threshold pair takes the values of (25ms, 50ms), (50ms, 100ms), (100ms, 200ms), (150ms, 300ms), and (200ms, 400ms). Apart from having to maintain two domains, slot reconciliation causes very little overhead ($0.06\% \sim 4\%$).

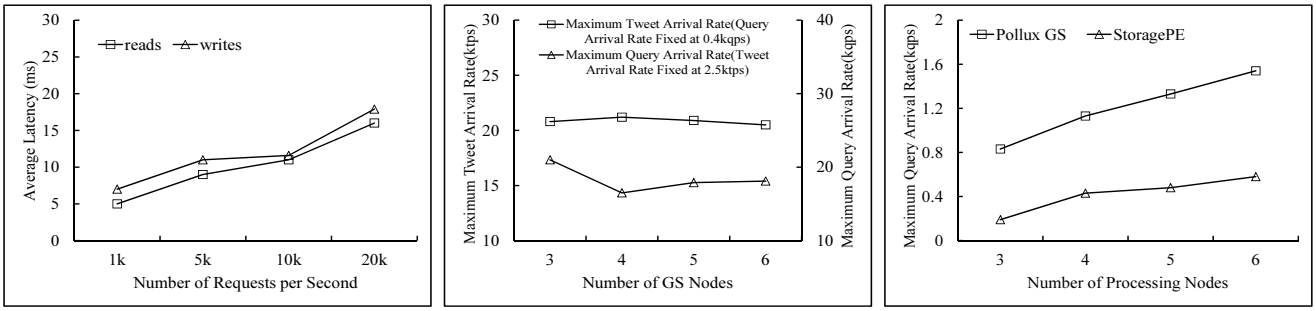
Figure 9(b) presents the load dispersion at failure and recovery. We conduct the experiments twice and identical input data are used. For the first experiment, we shut down the S4 process on a randomly-selected processing node, and monitor the load (mainly measured by bandwidth) of all remaining nodes over time until all data are recovered (on a newly joined replacement node). For the second experiment, no process is shut down and everything runs normally. We compute the ratio of the load observed from the first experiment with that from the second and present the results in Figure 9(b). The loads of the peer node for the failed node, the other processing nodes, and the GS nodes, are plotted separately. As shown in Figure 9(b), the peer node's load increases by about 14.2% on average (not including the load increase due to replicating its data to the new replacement node), and the other processing nodes' load increases by about 11.1%. However, the GS nodes' load does not increase because the failure is on a processing node.

To evaluate the query performance upon failover, we set the tweet and query arrival rates at 2.5ktps and 0.4kqps respectively, and then shut down the S4 process on one of the processing nodes. Figure 9(c) shows that the interrupted queries typically take one or two times more time to process, but are still answered in sub-seconds. These results verify the effectiveness of the proposed fault-tolerance strategy.

7.4 Performance of Global Storage

Several experiments are conducted to evaluate the effectiveness and the efficiency of the global storage.

To test the read and write performance, we first deploy 4 GS nodes and 6 processing node in one domain, and vary the number of read requests from 1K/sec to 20K/sec. We then vary the number of write requests again from 1K/sec to 20K/sec. Figure 10(a) shows that events at a high rate of 20K requests per second, the global storage still offers very low-latency. This verifies the global storage's efficiency.

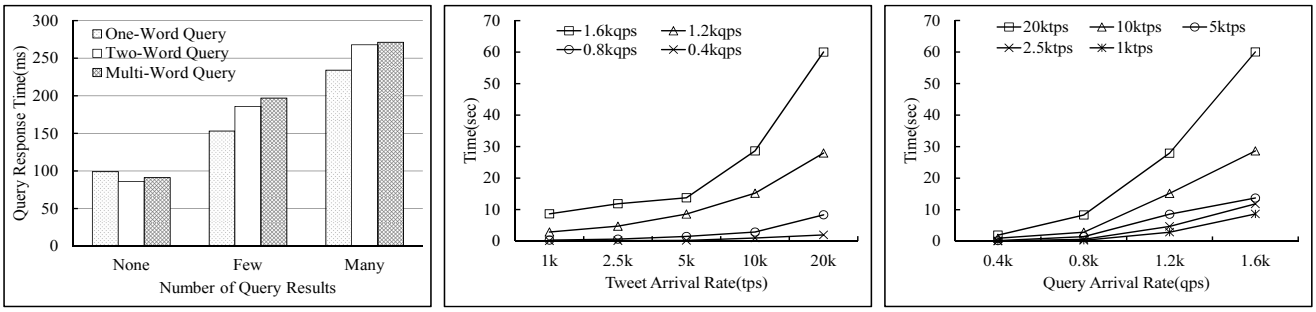


(a) Average latency for read requests only and write requests only

(b) Maximum sustainable tweet and query arrival rate for varying number of GS nodes

(c) A comparison of Pollux GS and StoragePE implementation

Figure 10: Performance of global storage



(a) Response time for queries at 0.4kqps and 2.5ktps

(b) Average response time for varying tweet arrival rates

(c) Average response time for varying query arrival rates

Figure 11: Query response time for varying tweet and query arrival rates

To evaluate how global storage affects the maximum sustainable tweet and query arrival rates, we start with 4 processing nodes and 3 GS nodes in a domain, and add more GS nodes to the domain one by one until all 6 GS nodes available are used. We record both the maximum sustainable tweet arrival rate (with the query arrival rate fixed at 0.4kqps) and the maximum query arrival rate (with the tweet arrival rate fixed at 2.5ktps) for varying number of GS nodes, and the results are shown in Figure 10(b). Neither the tweet arrival rate nor the query arrival rate increases as GS nodes are added. This is because the bottleneck lies not in the global storage (the global storage has provided sufficient bandwidth for data access) but in the processing nodes. Nonetheless, adding more GS nodes will increase the maximum time windows for real-time indexing, as shown in Figure8(c).

Figure 10(c) presents the maximum sustainable query arrival rate with or without the Pollux's global storage. Without the global storage, we have to use a solution based on S4 PEs (called the StoragePEs in Section 5.1), which can be deployed onto multiple processing nodes. As shown in Figure 10(c), the GS solution has a performance advantage of $129\% \sim 161\%$ over the StoragePE solultion.

7.5 Performance of Query Processing

In this set of experiments, we focus on the performance of query processing. Experiments are conducted in one domain with 6 processing nodes and 4 GS nodes.

The response times for queries with varying tweet and query arrival rates are shown in Figure 11. We first set the query arrival rate at 0.4kqps and tweet arrival rate at 2.5ktps, and compare the response time for queries with varying lengths (one, two, many) and result sizes (no result, few results, and many results). If the number of the results are among the top 20% of the result sizes of all queries, it is considered "many", otherwise "few". The comparison is shown in Figure 11(a). It can be observed from the figure that the result size has a notable effect on the response time, because more processing has to be done at MergePEs and RankPEs. The number of keywords, on the other hand, does not have a significant impact on the response time, although in the case of one-word queries, response is generally faster as less work has to be done. Figure 11(b) and Figure 11(c) show that the average response time increases as with increased tweet or query arrival rate. As can be observed from Figure 11(c), when the query arrival rate goes beyond a certain level, the response time will become unacceptable. This is because in those cases, the system becomes over-stressed, coinciding with the results discussed in the preceding subsection.

8. CONCLUSIONS

The quest for real-time indexing of microblogs has recently become pressing due to the inability of traditional indexing and search methods in providing real-time search. The sheer volume of fast arriving microblog data, along with the large number of concurrent queries, pose significant technical challenges. We have proposed Pollux, a distributed, scalable and elastic system enabling real-time indexing and search. We propose a failover strategy that assists Pollux with fast and lossless recovery from node failures, and develop a global storage facility that stores shared data and makes the ranking operation more efficient. The experiments on a Twitter dataset demonstrates the effectiveness of the proposed solutions and the scalability of Pollux.

For future work, we would like to study how to apply Pollux to other tasks involving real-time processing of social media contents, such as online clustering and classfication of microblogs, as well as automated event detection.

9. ACKNOWLEDGMENT

This work was supported in part by National Natural Science Foundation of China Grants (No. 61272092, No. 60903108), the Program for New Century Excellent Talents in University (NCET-10-0532), the Natural Science Foundation of Shandong Province of China Grant (No. ZR2012FZ004), the Independent Innovation Foundation of Shandong University (2012ZD012), the SAICT Experts Program, and NSERC Discovery Grants. The authors would like to thank the anonymous reviewers, whose valuable comments helped improve this paper.

10. REFERENCES

[1] Apache bookkeeper. http://zookeeper.apache.org/bookkeeper/.

[2] Apache zookeeper. http://zookeeper.apache.org/.

[3] The engineering behind twitter's new search experience. http://engineering.twitter.com/2011/05/engineering-behind-twitters-new-search.html.

[4] Memcached. http://memcached.org/.

[5] New tweets per second record – 25,088 tps – set by screening of japanese movie "castle in the sky". http://techcrunch.com.

[6] Streambase. http://streambase.com.

[7] Tweets2011. http://trec.nist.gov/data/tweets/.

[8] Twitter storm. https://github.com/nathanmarz/storm.

[9] Twopcharts.com. http://www.twopcharts.com.

[10] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[11] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *CIDR*, Asilomar, CA, January 2005.

[12] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani. Spc: A distributed, scalable platform for data mining. In *Proceedings of the 4th international workshop on Data mining standards, services and platforms*, pages 27–37. ACM, 2006.

[13] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, R. Schmidt, and A. Aiyer. Apache Hadoop goes realtime at Facebook. In *SIGMOD*, pages 1071–1080, 2011.

[14] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-time search at twitter. In *ICDE*. IEEE, 2012.

[15] C. Chen, F. Li, B. Ooi, and S. Wu. Ti: an efficient indexing mechanism for real-time search on tweets. In *SIGMOD*, pages 649–660, 2011.

[16] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[17] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[18] A. Dong, R. Zhang, P. Kolari, J. Bai, F. Diaz, Y. Chang, Z. Zheng, and H. Zha. Time is of the essence: improving recency ranking using twitter data. In *WWW*, pages 331–340, 2010.

[19] Y. Duan, L. Jiang, T. Qin, M. Zhou, and H. Shum. An empirical study on learning to rank of tweets. In *COLING*, pages 295–303, 2010.

[20] J. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *ICDE*, pages 779–790, 2005.

[21] J. Hwang, U. Cetintemel, and S. Zdonik. Fast and highly-available stream processing over wide area networks. In *ICDE*, pages 804–813, 2008.

[22] J. Hwang, Y. Xing, U. Cetintemel, and S. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *ICDE*, pages 176–185. IEEE, 2007.

[23] B. J. Jansen, G. Campbell, and M. Gregg. Real time search user behavior. In *ACM CHI*, pages 3961–3966, 2010.

[24] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, pages 654–663. ACM, 1997.

[25] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600. ACM, 2010.

[26] Y. Kwon, M. Balazinska, and A. Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. *PVLDB*, 1(1):574–585, 2008.

[27] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[28] L. Lin, X. Yu, and N. Koudas. "Pollux: Towards scalable distributed real-time search on microblogs". Technical Report. http://www.yorku.ca/xhyu/ITEC-TR-201203.pdf.

[29] D. Luckham. *The power of events: an introduction to complex event processing in distributed enterprise systems*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[30] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: distributed stream computing platform. In *ICDM Workshops*, pages 170–177. IEEE, 2010.

[31] D. Ongaro, S. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *SOSP*, pages 29–41. ACM, 2011.

[32] J. Rao, E. Shekita, and S. Tata. Using paxos to build a scalable, consistent, and highly available datastore. *PVLDB*, 4(4):243–254, 2011.

[33] Z. Sebepou and K. Magoutis. Scalable storage support for data stream processing. In *Proceedings of IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6. IEEE, 2010.

[34] M. Shah, J. Hellerstein, and E. Brewer. Highly available, fault-tolerant, parallel dataflows. In *SIGMOD*, pages 827–838. ACM, 2004.