LogKV: Exploiting Key-Value Stores for Event Log Processing

Zhao Cao[1] Shimin Chen[1] Feifei Li[2] Min Wang[1] X. Sean Wang[3]

[1]HP Labs, China
{zhao.cao, shimin.chen, min.wang6}@hp.com

[2]School of Computing, University of Utah
lifeifei@cs.utah.edu

[3]School of Computer Science, Fudan University
xywangCS@fudan.edu.cn

ABSTRACT

Event log processing and analysis play a key role in applications ranging from security management, IT trouble shooting, to user behavior analysis. Recent years have seen a rapid growth in system scales and the corresponding rapid increase in the amount of log event data. At the same time, as logs are found to be a valuable information source, log analysis tasks have become more sophisticated demanding both interactive exploratory query processing and batch computation. Desirable query types include selection with time ranges and value filtering criteria, join within time windows, join between log data and reference tables, and various aggregation types. In such a situation, parallel solutions are necessary, but existing parallel and distributed solutions either support limited query types or perform only batch computations on logs. With a system called LogKV, this paper reports a first study of using Key-Value stores to support log processing and analysis, exploiting the scalability, reliability, and efficiency commonly found in Key-Value store systems. LogKV contains a number of unique techniques that are needed to handle log data in terms of event ingestion, load balancing, storage optimization, and query processing. Preliminary experimental results show that LogKV is a promising solution.

1. INTRODUCTION

Event log processing and analysis play a key role in applications ranging from security management [3,8], IT trouble shooting [1,3,12], to user behavior analysis [14]. Event records are generated by a wide range of hardware devices (e.g., networking devices) and software systems (e.g., operating systems, web servers, database servers), reporting information about system status, configurations, operations, warnings, error messages, and so on. Security analysts process event logs to detect potential security breaches. System admins or software developers analyze event logs to find root causes of errors or system crashes. Since logs (e.g., web server logs) also provide valuable information about application workloads and user behaviors, application designers can analyze logs to improve the efficiency and quality of applications.

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2013.

6th Biennial Conference on Innovative Data Systems Research (CIDR '13) January 6-9, 2013, Asilomar, California, USA.

1.1 Requirements for Event Log Processing

Recent years have seen rapidly growing system scales, and accordingly a rapidly increasing amount of log event data. For the purposes of auditing and data analysis, it is often required to store important event log data for several years. The growing system scales put significant pressures on the storage and processing of event logs.

Moreover, event logs from different hardware devices and software systems usually have different formats. There may even be multiple types of event records in a single log (as seen in, e.g., unix syslog). Records of different formats may contain different kinds of information, making it difficult to design a uniform schema for all logs.

Furthermore, log analysis tasks can be sophisticated. From a processing perspective, both interactive exploratory queries and batch computations are crucial. The former is useful when users look into a specific problem or try to gain new insights, and the latter can be adopted when the processing task is well defined. Desirable query types include selections on specified time ranges and with categorical filtering criteria, joins among event log data in time windows (e.g., detecting user sessions, looking for correlated events), joins between event log data and reference tables (e.g., user profiles), and aggregations.

Finally, many analysis tasks require the flexibility of incorporating user implemented algorithms [8]. It is quite a challenge to design a high-throughput, scalable, and reliable log management system to support the wide range of log formats and log analysis tasks.

1.2 Related Work

Existing distributed solutions for log processing either support limited query types or perform only batch computations on logs. Commercial event log management products [3] provide distributed search capability, but only supporting selection queries. Map/Reduce has been used to process large web logs [4] and network logs [8]. However, it works poorly with interactive exploration of data.

Moreover, event log processing considered in this paper is related to but different from data streams processing [6]. While the latter focuses on real-time processing of pre-defined operations on data streams, the focus of this paper is on storing and processing a large amount of log event data in both interactive and batch modes. The two types of systems are complementary and may be combined in practice.

Furthermore, previous work [10] proposes a centralized data streams warehouse system for archiving and analyzing data streams, with an emphasis on the maintenance of complex materialized views. Because of the centralized setting, the system can support an ingestion throughput of about 10MB per second. In comparison, we aim to support orders of magnitudes higher event ingestion throughput with a scalable solution based on Key-Value stores.

1.3 Our Solution: LogKV

Key-value stores (e.g., Dynamo [9], BigTable [5], SimpleDB [2], Cassandra [11], PNUTS [7]) are widely used to provide large-scale highly-available data store in the cloud. We find that the characteristics of key-value stores are a good fit for the requirements of an event log management system. First, the reliability and scalability requirements match those for key-value stores well. Second, event fields extracted from raw event records can be naturally represented as key-value pairs. In a key-value store that supports row keys and column keys [5, 11], we can represent every event record as a row, and use column key-value pairs to store the extracted event fields. Since we can store different number of column key-value pairs for event records with different formats, this method is capable of accommodating the wide range of log formats. Finally, by using a key-value store, it is straightforward to apply filtering criteria on key-value pairs, which forms the basis for implementing efficient log processing operations. However, directly using existing Key-Value stores will suffer from low ingestion throughput, large storage overhead, and low query performance for log processing (c.f. Section 5).

Thus we propose *LogKV*, a log management system that exploits the scalability, reliability, and efficiency of key-value stores for log processing. Based on our experience, we set up our goal to store and process 10 PB of log data, and handle up to a peak of 100 TB of incoming log data per day. To achieve this goal, we face a number of significant challenges:

- *Storage overhead*: We would like to use as fewer machines as possible for achieving the goal. To reliably store 10 PB of data, LogKV will maintain three data copies, i.e. storing 30 PB of data in total. Suppose every machine node has 10TB of local disk space[1]. Then, 3000 machines are required to store the 30 PB of data. To reduce storage overhead, we will investigate log compression techniques. 5:1, 10:1, and 20:1 compression ratios will lead to 600, 300, and 150 machines, respectively.
- *Efficient query processing*: It is critical to minimize inter-machine network communications in order to achieve good query performance on the stored log data. The main challenge lies in the join queries, which usually require shuffling data across machines. Among the two types of join queries that we consider, we decide to optimize data layout for joins among log data in time windows because both join data sources can be very large. (In the other join type, i.e., joins between log data and reference tables, reference table size is relatively small.) LogKV will co-locate log data in a time range on the same machine so that most computation can be performed locally.
- *High-throughput log ingestion*: Suppose data are to be kept for 3 years, which is slightly over 1000 days. Then 10 PB log data in total means about 10 TB log data per day. However, the daily log traffic may be well beyond the computed average storage size per day because of two factors. First, there could be sudden bursty log traffic. For example, a faulty component may generate significantly more messages than usual [12]. An external popular event (e.g., Olympics) can significantly increase system usage, such as web site visits. Second, log data may not be equally important. System admins may make decisions on the time length for keeping each type of log data. Some data may be deleted earlier than the other. Given these considerations, we set our goal to support 10x of the daily average, i.e. 100 TB of incoming log data per day, or 1.2 GB per second.

[1]Presently, a blade server can have two disks of 4TB each. As hard disk capacity doubles roughly every two years, 8TB hard disks are expected to appear in a couple of years. Moreover, customized data center machine designs or data-intensive servers can have four or even eight disk slots. Therefore, it is reasonable to assume that a machine node will have 10–20TB of local disk space.

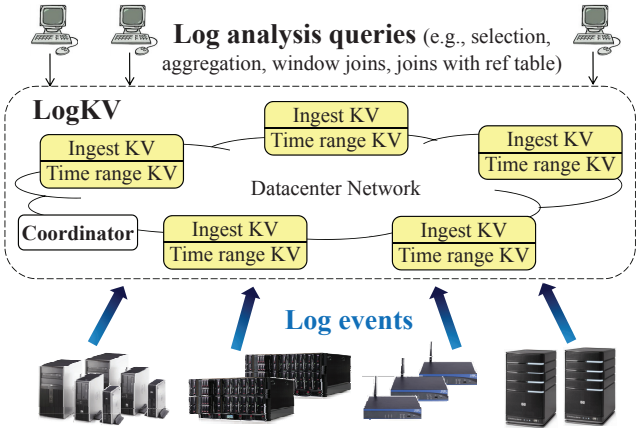

Figure 1: LogKV overview.

Figure 1 depicts the overall architecture of LogKV. There are a coordinator node and a set of worker nodes connected through the data center network in the same data center. Worker nodes receive log event records from a wide range of log sources (i.e., hardware devices or software systems generating log event records), store log data, and support query processing. The coordinator node maintains meta-information about the mapping of log sources to worker nodes, and the layout of stored log data on worker nodes. Log sources and query clients may reside in the same data center as LogKV does or may be located outside of the LogKV's data center.

There are two components on every worker node. *IngestKV* stores incoming log event records for high-throughput log ingestion, while *TimeRangeKV* provides the main storage for log data. At initialization time, system admins set a Time Range Unit (TRU) configuration parameter. Time is conceptually divided into TRU-sized buckets. TimeRangeKV will co-locate log data in each TRU bucket on the same machine. Without delving into details (cf. Section 4), the choice of TRU should consider the common window size of join queries. However, sending all incoming log events to a single TimeRangeKV (associated with the current TRU) will create a performance bottleneck. IngestKV addresses this problem by appending incoming events at all worker

Table 1: Terms used in this paper.

Term	Description
TRU	Time Range Unit
N	Number of LogKV nodes
S	Number of log sources
l_i	Ingestion bandwidth of log source i
B_{ingest}	Total log ingestion bandwidth $B_{ingest} = \sum_{i=1}^{S} l_i$
B_{node}	Network bandwidth of a single LogKV node
M	Number of concurrent receivers
f	Log compression ratios, e.g., 10

Algorithm 1 Assign individable log sources

Require: Throughput of individable log sources are $l_1, l_2, ..., l_d$

Ensure: Aggregate throughput assigned to worker k is $w[k]$, where $k = 1, 2, ..., N$

1: Set all $w[k]$ to 0, where $1 \le k \le N$
2: Sort $l_1, l_2, ..., l_d$ in descending throughput order
3: **for** $i = 1 \to d$ **do**
4: $k = \arg\min\{w[k]\}$, $1 \le k \le N$
5: Assign log source i to worker k
6: $w[k] \leftarrow w[k] + l_i$

nodes and intelligently shuffling log data to achieve the co-location property at TimeRangeKV.

The rest of the paper is organized as follows. Section 2 presents how IngestKV supports log ingestion and shuffles log data into TimeRangeKV. Section 3 describes data compression and storage in TimeRangeKV. Section 4 illustrates LogKV's support for the desired query types. Section 5 reports preliminary evaluation of the LogKV idea. Finally, Section 6 concludes the paper.

2. HIGH-THROUGHPUT LOG INGESTION

LogKV obtains log event data using either pull or push based methods from log sources. Incoming log data will first reside in IngestKV, then be shuffled to TimeRangeKV. The system maintains reliability through data replication. In the following, we describe the different aspects of this process. Table 1 summarizes the terms used in this paper.

2.1 Log Source Mapping to Worker Nodes

LogKV obtains log event data from a log source in one of the following three ways. First, the ideal case is that a LogKV agent can run on the log source machine, extracting and sending log events to LogKV. This method is preferred to the remaining two when this is a possible option in a log source. Second, log sources (e.g., syslog) may be configured to forward log events to a remote host, i.e. a LogKV worker node. Finally, a log source stores log events to local files, and LogKV uses ftp/scp/sftp to obtain the log data. The coordinator of LogKV records the log extraction method for every log source.

Suppose the average throughput of the log sources are $l_1, l_2, ..., l_S$, and the total incoming throughput is B_{ingest}. That is, $B_{ingest} = \sum_{i=1}^{S} l_i$. LogKV aims to balance the log ingestion throughput across the N worker nodes, ideally

Algorithm 2 Assign dividable log sources

Require: Throughput of dividable log sources are $l_{d+1}, ..., l_s$

Ensure: Aggregate throughput assigned to worker k is $w[k]$, where $k = 1, 2, ..., N$

1: Compute $divTotal = \sum_{i=d+1}^{s} l_i$
2: Sort worker nodes in descending throughput order so that $w[ord[k]] \le w[ord[k+1]]$, where $ord[.]$ is an index array, and $1 \le k < N$
3: $t \leftarrow divTotal$
4: **for** $k = 1 \to N$ **do** {Compute $targetMin$ }
5: $t \leftarrow t + w[ord[k]]$ $\{t = divTotal + \sum_{j=1}^{k} w[ord[j]]\}$
6: **if** $(k == N)$ OR $(t/k \le w[ord[k+1]])$ **then**
7: $targetMin \leftarrow t/k$
8: break
9: $i \leftarrow d + 1$
10: **for** $j = 1 \to k$ **do** {Assign log sources}
11: $diff \leftarrow targetMin - w[ord[j]]$
12: **while** $diff > 0$ **do**
13: **if** $diff \ge l_i$ **then**
14: Assign the rest of l_i to worker $ord[j]$
15: $diff \leftarrow diff - l_i$
16: $l_i \leftarrow 0$
17: $i \leftarrow i + 1$
18: **else**
19: Assign $diff$ from l_i to worker $ord[j]$
20: $l_i \leftarrow l_i - diff$
21: $diff \leftarrow 0$
22: $w[ord[j]] \leftarrow targetMin$

achieving an average node throughput of B_{ingest}/N. Note that log sources supporting the first extraction method are *dividable* in that they can divide their log traffic among multiple worker nodes. The other log sources are *individable*, and must each be assigned to a single worker node.

We assign log sources to worker nodes in two steps. In the first step, we assign individable log sources using a greedy algorithm as shown in Algorithm 1. The algorithm sorts all individable log sources in descending order of their throughput (Line 2). Then it goes into a loop to assign every individable log source (Line 3). An iteration in the loop computes the worker with the least log ingestion throughput that has been assigned in previous iterations (Line 4), and assigns the next log source in the sort order to the worker (Line 5–6).

In the second step, we assign dividable log sources in order to balance the assigned throughput across worker nodes as much as possible, as shown in Algorithm 2. $divTotal$ is the aggregate throughput of all dividable log sources. The algorithm sorts all the worker nodes in the ascending order of the current assigned throughput (Line 2).

We would like to reduce the difference between the maximum and minimum assigned throughput for individual workers. Conceptually, we can assign log ingestion throughput to worker $ord[1]$, which has the current minimum throughput. When its throughput is increased to the same as $w[ord[2]]$, both of worker $ord[1]$ and $ord[2]$ have the minimum throughput. Then we will need to assign throughput to both worker $ord[1]$ and $ord[2]$ in order to increase the minimum throughput among all workers. Similarly, when the assigned through-

put of worker $ord[1]$ and $ord[2]$ is equal to $w[ord[3]]$, we will need to increase the throughput of three workers together. In general, suppose the final minimum throughput among all workers is $targetMin$. Then k workers $(ord[1], \cdots, ord[k])$ will have $targetMin$ as their throughput, while the throughputs of the other workers remain unchanged. Since the newly assigned throughput is equal to $divTotal$, we have the following:

$$\begin{aligned} &\sum_{j=1}^{k} (targetMin - w[ord[j]]) = divTotal \\ &k == N \wedge targetMin \leq w[ord[k+1]] \end{aligned} \quad (1)$$

Using Equation 1, Algorithm 2 computes $targetMin$ as $(divTotal + \sum_{j=1}^{k} w[ord[j]])/k$ (Line 3–8). Finally, it assigns dividable log sources to worker $ord[1], \cdots, ord[k]$ to increase their throughput to $targetMin$ (Line 9–22).

After the computation, the coordinator configures the mapping from log sources to worker nodes accordingly.

Analysis. In Algorithm 1, the sort operation in Line 2 takes $O(dlogd)$ time. A heap can be used to efficiently obtain the worker with the current minimum throughput in Line 4 with $O(logN)$ cost per iteration. Therefore, the time complexity of Algorithm 1 is $O(d(logN + logd))$.

Moreover, we can prove that after Algorithm 1,

$$w[k] < \frac{1}{N} \sum_{i=1}^{d} l_i + \max_{1 \leq i \leq d} \{l_i\}, 1 \leq k \leq N \quad (2)$$

Suppose $w[k]$ has the maximum assigned throughput after Algorithm 1, and l_j is the last log source assigned to worker k. It must satisfy that $w[k] - l_j < \frac{1}{N} \sum_{i=1}^{d} l_i$. Otherwise, there must have been some other worker with lower throughput at the time of assignment. Then l_j would have been assigned to this other worker. Since $l_j \leq \max_{1 \leq i \leq d} \{l_i\}$, we can derive Equation 2.

In Algorithm 2, sorting takes $O(NlogN)$ time, while the two loops have linear costs. Therefore, the time complexity of Algorithm 2 is $O(NlogN)$. The algorithm will reduce the gap between the maximum and minimum assigned throughput as much as possible. It is easy to see that if none of the worker nodes exceeds B_{ingest}/N after Algorithm 1, then the algorithm will achieve the ideal assignment.

2.2 Log Parsing

Worker nodes perform event log parsing in a fully distributed fashion. On a worker node, IngestKV runs a log source adapter module. It maintains a log source table, registering the type and parsing rules of every assigned log source. When an incoming log event arrives at a worker node, the adapter module parses the log event to extract important information as key-value pairs. There are also special key-value pairs representing the log source, the arrival timestamp, and if required, the raw event. IngestKV maintains log event records in time order on a worker node.

The above description assumes that log parsing rules exist. However, in practice, it is challenging to manually generate the log parsing rules for all different types of log sources. The number of log sources increases as the number of hardware devices and software systems increase. A manual solution is difficult to keep up. An automatic solution will be desirable, but is beyond the scope of the current paper.

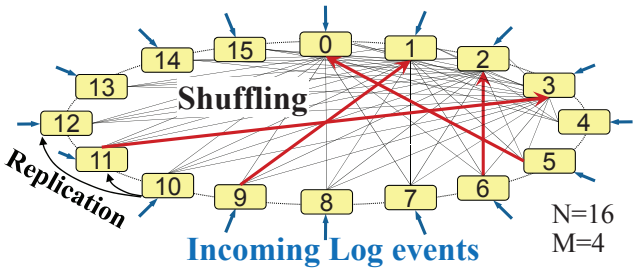

Figure 2: Log ingestion in LogKV.

Algorithm 3 Log shuffling

Require: Receiver i must obtain event log data from worker 1, ..., N

1: $list \leftarrow \{1, ..., N\}$
2: **for** $i = 1 \rightarrow N$ **do**
3: $k \leftarrow random(list)$
4: **if** Worker k is busy **then**
5: $k \leftarrow random(list)$
6: Transfer event log data from worker k
7: Remove k from $list$

2.3 Log Shuffling

IngestKV shuffles log events to TimeRangeKV. Figure 2 illustrates the log shuffling idea. There are N LogKV worker nodes, each with an ID from 0 to $N-1$. $N = 16$ in the figure. In TimeRangeKV, a TRU bucket is mapped to a worker node if the start timestamp of the time range satisfies the following:

$$nodeID = \lfloor \text{StartTimeStamp/TRU} \rfloor \ \ modulo \ \ N \quad (3)$$

Note that $i = \lfloor \text{StartTimeStamp/TRU} \rfloor$ uniquely identifies the TRU bucket. We say that the node is the owner of TRU_i. During log shuffling, we use *receivers* to denote worker nodes receiving shuffled events. Suppose TRU of the current time is mapped to node j. If the log events were immediately forwarded to node j, node j would be overwhelmed by a shuffling bandwidth of B_{ingest}.

In order to reduce the shuffling bandwidth, we exploit the property of data center network that non-conflicting network connections can often achieve the maximum bandwidth at the same time. IngestKV performs shuffling in rounds. A round takes $M \cdot TRU$ time. $M = 4$ in Figure 2. Up to two rounds of incoming event log data are kept in IngestKV on a worker node. In round r, IngestKV buffers the incoming event log data for the current round, and shuffles the event log data arrived in round $r-1$ to M receivers. At the end of round r, the event log data of round $r-1$ is removed, and the event log data of round r will be shuffled in the next round. In this design, the average shuffling bandwidth of a receiver is reduced to B_{ingest}/M. This is because the same amount of event log data are shuffled to the receiver, while the shuffle time is increased by a factor of M.

Now we face the problem of shuffling event log data from N nodes to M receivers. To avoid hot spots, we design a randomized shuffle algorithm as shown in Algorithm 3. A receiver maintains a to-do list, which is initialized with all worker nodes (Line 1). Then the receiver goes into a loop. In every iteration, it randomly selects a node from the to-do list (Line 3), transfers event log data from the node (Line 6), then updates the to-do list by deleting the node (Line 7). As

shown in Figure 2, there are four receivers, node 0, 1, 2, and 3. Event log data will be transferred from all 16 nodes to the 4 receivers as shown by the thin lines. In a certain iteration, the receivers choose different nodes to transfer event log data from using the randomized algorithm as illustrated by the bold red arrows.

This algorithm works well when $N >> M$. The to-do list shrinks at every iteration. After $N-K$ iterations, the to-do list contains K nodes. A node appears in $P = \frac{KM}{N}$ lists on average. If a node is selected by a receiver, the probability that it is chosen by another receiver is $(1-(1-\frac{1}{K})^{P-1})$. This is roughly equal to $1-e^{\frac{M}{N}}$ when K is large. Clearly, when $N >> M$, the probability is low.

To deal with cases where M is close to N, the algorithm checks whether the chosen node is already serving another receiver (Line 4). If yes, then it can randomly choose another node in the to-do list (Line 5).

2.4 Replication for Reliability

Both IngestKV and TimeRangeKV replicate log data for high availability. IngestKV buffers incoming log data for a short amount of time, while TimeRangeKV is supposed to store log data for up to several years. Consequently, data in TimeRangeKV are expected to experience more failures compared to IngestKV. Therefore, we maintain two copies of log data in IngestKV, while producing three data copies in TimeRangeKV.

Figure 2 shows the replication strategy for TimeRangeKV. A node j will replicate its log data to node $j+1$ and node $j+2$ modulo N. (We can easily adapt existing Key-Value stores, e.g., Cassandra, to support this replication strategy.) Note that three adjacent TRU buckets will be present on a single node due to the replication strategy. This facilitate window join queries (cf. Section 4).

For IngestKV, the coordinator will balance the network traffic by choosing an appropriate replication strategy. In the simplest case, every two nodes are organized into a replication pair, forwarding incoming log events to each other. However, if the log source assignment does not achieve ideal balance, the coordinator can decide to divide the replicating traffic intelligently in order to balance the overall log ingestion traffic including replication. Moreover, the replication efficiency in IngestKV may be improved if log sources are capable of re-sending recent log events (e.g., with the support of LogKV agent). Suppose the coordinator knows that log source A can re-send last t seconds of log events. Then, rather than forwarding every individual event, IngestKV will forward a batch of A's log events every t seconds to reduce network cost.

A node failure is detected when a network connection (for replication, shuffling, or query processing) times out. When a node i fails, the coordinator will remap the TRUs owned by node i temporarily to node $i+1$ modulo N. In the meantime, it will obtain and initiate another machine in the data center to replace the failed node. We use Apache ZooKeeper to ensure the reliability of the coordinator.

2.5 Parameter Selection

We discuss the choice of M. The overall network traffic of a receiver must be lower than its available bandwidth. Log ingestion and replication consumes $2B_{ingest}/N$ bandwidth. Log shuffling takes B_{ingest}/M bandwidth. Suppose the log compression ratio is f. Replicating two data copies in TimeRangeKV takes $2B_{ingest}/(Mf)$ bandwidth. Therefore, we have the following constraint:

$$(\frac{2}{N}+\frac{1}{M}+\frac{2}{Mf})B_{ingest} \leq B_{node} \quad (4)$$

We would like to minimize the amount of log data buffered in IngestKV for query efficiency. Window joins on data in IngestKV require expensive data shuffling operations (cf. Section 4). Therefore, we can compute M as follows:

$$M = \lceil (1+\frac{2}{f})\frac{1}{\frac{B_{node}}{B_{ingest}}-\frac{2}{N}} \rceil \quad (5)$$

Suppose B_{ingest}=1.2GB/s, B_{node}=100MB/s, N=150 nodes, f=20. Then M=16 concurrent receivers.

2.6 Coping with Bursty Log Data

We already consider 10x higher log ingestion throughput than average in our design goal, as discussed in Section 1. The coordinator can also try balancing event log ingestion traffic when a subset of log sources suddenly experience high traffic. If a dividable log source sees a dramatic throughput increase, the coordinator can divide this log source and reassign it easily. If an individable log source sees a sudden increase in throughput, the coordinator can re-assign the log source in the same spirit as Algorithm 1. After removing the old throughput of this log source, it can compute the worker node with the minimum assigned throughput, and then assign the individable log source to it. The coordinator can periodically check the gap between the maximum and minimum assigned log ingestion throughput among worker nodes. If the gap is above a threshold, the coordinator can re-compute the log source mapping using Algorithm 1 and Algorithm 2.

3. LOG STORAGE IN TimeRangeKV

The TimeRangeKV provides persistent storage for LogKV through a key-value store. As explained in Section 2, it uses the time range (in multiple of the TRU) as the row key to partition incoming log records. This design is motivated by the observations that most if not all query processing tasks in log analysis is time-dependent; furthermore, most query tasks access log records within a user-specified, contiguous time range, i.e., there is strong temporal locality in terms of the query accessing patterns in log analysis.

That said, all records within a time range in $[\ell \cdot \text{TRU}, (\ell+1)\text{TRU})$ of length TRU $(\exists \ell \in [0, \lceil \text{Now}/\text{TRU} \rceil - 2M))$ are stored in one worker node, running a TimeRangeKV instance (except the most recent $2M \cdot$ TRU records that will be buffered in IngestKV, or being shuffled from IngestKV to TimeRangeKV). During log shuffling, the log events transferred from each IngestKV instance are in time order. A receiver node j will store the log data shuffled from an IngestKV instance temporarily in a file. When node j receives the log data from all N IngestKV instances, it will perform a multi-way merge operation on the N temporary files, and store the sorted records into the underlying key-value store.

There can be multiple ways to store the data. In LogKV, we exploit the following design, as shown in Figure 3, to maximize the compression ratio and the query performance. Suppose the log records in a time range of length TRU coming into node j are an ordered set $R = \{r_1, r_2, \ldots, r_u\}$ for some integer value u, ordered by their timestamps. Note

$t_i \in [\ell \cdot \text{TRU}, (\ell+1) \cdot \text{TRU})$ row key: $k = t_i/\text{TRU}$

record	timestamp	a_1	$\cdots$	a_d
r_1	t_1	$v_{1,1}$	$\cdots$	$v_{1,d}$
r_2	t_2	$v_{2,1}$	$\cdots$	$v_{2,d}$
$\cdots$				
r_u	t_u	$v_{u,1}$	$\cdots$	$v_{u,d}$

column a_1, $V_1 = \{v_{1,1}, \ldots, v_{1,d}\}$
$V_1' = \{v_{1,2}, v_{1,5}, \ldots\}$, $b_1 = [0, 1, 0, 0, 1, \ldots]$
$\bullet \quad \bullet \quad \bullet$
column a_d, $V_u = \{v_{u,1}, \ldots, v_{u,d}\}$
$V_u' = \{v_{u,1}, v_{u,4}, \ldots\}$, $b_u = [1, 0, 0, 1, \ldots]$

Figure 3: Local storage in LogKV.

that each log record is converted from a log event in the raw data by IngestKV, consisting of a timestamp and a set of key-value pairs (c.f. Section 2). For example, a log record from a network traffic log may look like {t, (source IP, value), (destination IP, value)}. Without loss of generality, we assume that each record has a format of $\{t, (a_1, v_1), \ldots, (a_d, v_d)\}$; and the jth value of the ith record in R is denoted as $v_{i,j}$. Since different records may not have the same set of key-value pairs, value $v_{i,j}$ may simply be a NULL value, e.g., if there is no parsing rule on extracting the value of key a_j for a record r_i. Hence, d denotes the total distinct number of keys for which IngestKV has parsed and extracted any values from raw log events in a time range.

All records in R is stored in a single row with the row key $k = \lfloor t/\text{TRU} \rfloor$. We call the ith key-value pair (a_i, v_i) for all records in R as the ith column of these records. We organize the ith column from all records of R into an array A_i. Array A_i is stored as a column key-value pair under row key k, where the column key is a_i, and the value V_i contains all the values from the entire column i. That is, $V_i = \{v_{1,i}, \ldots, v_{u,i}\}$. By doing so, we end up with d columns for row key k, which will be stored into the underlying key-value store.

Before inserting row key k and the columns into the underlying key-value store, TimeRangeKV compresses the data in order to reduce storage overhead, and hence improve query efficiency as well. A straightforward solution is to simply compress the value V_i of a column a_i at its entirety. However, a critical observation that we have made is that V_i often has a lot of NULL values, since log events are of different types, and some log events with certain attributes occur only infrequently (e.g., when an exception did occur). As a result, compressing V_i directly might not lead to the best compression ratio.

An improvement is to compress V_i after removing its NULL values. To do so, we introduce a bitmap index b_i of size u for V_i, where $b_i(j) = 0$ if $v_{j,i}$ =NULL and 1 otherwise. After which, we produce a vector V_i' that consists of only non-NULL values from V_i (the order is still preserved). Clearly, given V_i' and b_i one can easily reconstruct V_i. We then compress V_i' and b_i separately for column a_i and store the two compressed values into the underlying key-value store (under the column a_i for the row key k). This process is repeated for each column, from a_1 to a_d in turn.

When a time range contains a large number of records, we can refine the above process by introducing smaller "rows" locally. The idea is to further partition a time range into a number of smaller, but contiguous, time ranges in a local node j. Then, within each smaller time range, the above process is applied. This will achieve better query efficiency for point queries because only the smaller time range of data are fetched from disk and uncompressed.

Lastly, we highlight that since we are using a key-value store as the underlying storage layer, the replication of records in a time range from a TimeRangeKV instance to other TimeRangeKV instances on neighboring nodes can be automatically taken care of by leveraging the replication mechanism of the underlying key-value store, which is a basic service in most key-value stores, such as Cassandra.

4. QUERY PROCESSING ON EVENT LOGS

TimeRangeKV stores event log data that arrive $2M \cdot$TRU time ago, while more recent log events reside in IngestKV. For simplicity, we do not use the log data being shuffled in query processing. According to a query's time range predicate (or any time if not specified), LogKV executes the query either on TimeRangeKV or on IngestKV or on both if the time range spans the timestamp (now $-2M \cdot$ TRU).

In the following, we discuss the important design considerations for query processing.

4.1 Supporting Important Query Types

We discuss query processing for TimeRangeKV and IngestKV, respectively.

- *Selection/Projection Queries in TimeRangeKV and IngestKV*: Selections and projections can be efficiently processed in a distributed fashion on worker nodes. In IngestKV, a row key corresponds to an event record, and extracted event fields are stored as column key-value pairs. Therefore, selections and projections are performed on the row records. In TimeRangeKV, selections use bitmap indexes to extract qualified records, and projections directly take advantage of the column representation. As all columns are stored in the same order, it is straightforward to reconstruct event records.
- *Window Joins between Event Log Data in TimeRangeKV*: The replication strategy in TimeRangeKV ensures that $\forall i$, TRU_i's owner also stores a replica of TRU_{i-1} and TRU_{i-2} (cf. Section 2), thus it contains any time window ending in TRU_i and with a size up to 2TRU. What this means is that window joins with window size $\leq$ 2TRU can be handled efficiently by performing local joins on worker nodes.

  Larger window sizes require communication among worker nodes. For window size $\in$ (2TRU, 5TRU], TRU_i's owner needs to communicate with TRU_{i-3}'s owner so that event log data in TRU_{i-5}, ..., TRU_i will be available for the join. In general, a worker node must communicate with k other worker nodes for a join query if its window size $\in$ ((3k-1)TRU, (3k+2)TRU], $k = 1, ..., \lceil \frac{N}{3} \rceil$.
- *Window Joins between Event Log Data in IngestKV*: In contrast to TimeRangeKV, IngestKV must perform log data shuffling for window joins. This can be implemented with a Map/Reduce framework (e.g., Hadoop).
- *Joins between Event Log Data and Reference Tables in TimeRangeKV and IngestKV*: Previous study [4] compares join algorithms for joining event log data and reference tables in Hadoop when input data are stored on HDFS. We reference this work in our design. However,

we note that compared to HDFS, LogKV has much better selection support. It will be interesting future work to re-evaluate the join algorithms in the context of LogKV.

- *Group-by/Aggregation Queries in TimeRangeKV and IngestKV*: Group-by/aggregation queries can be processed using the Map-Reduce framework (e.g., Hadoop).

4.2 Choice of TRU

The choice of TRU impacts the query performance of both TimeRangeKV and IngestKV. Smaller TRU leads to smaller amount of data stored in IngestKV. Consequently, window join queries spanning both IngestKV and TimeRangeKV would shuffle less data in IngestKV, thereby obtaining better performance.

In TimeRangeKV, larger TRU means that window joins with larger time windows can be efficiently handled. On the other hand, a time range predicate may be mapped to fewer number of worker nodes. This will affect neither large analysis queries spanning long time ranges nor point queries focusing on very short time periods. However, selection queries with medium sized time range predicates may see fewer computation nodes when TRU gets larger.

In summary, we find that the choice of TRU is workload dependent. A rule of thumb is to choose TRU to be half of the most common time window size in window joins.

4.3 Discussions

Approximate Query Processing. Providing fast approximate estimations is a promising strategy to support interactive data explorations. LogKV is amenable to sampling based approximate query processing. For analysis queries on large time ranges, we can sample a subset of TRUs for estimations. If necessary, a finer-grain sample can be generated by sampling events in every related TRU. The latter mainly reduces network communications, while the former reduces, local I/O, computation, and network cost.

Secondary Indices. If a particular event attribute often appears in the filtering criteria of queries, one may want to create a secondary index on this event attribute to speed up query processing. We consider how to support secondary indices on individual TimeRangeKV. There can be two design choices. First, we can create a secondary index on all the event log data in a TimeRangeKV. Second, we can create a secondary index for every TRU in the TimeRangeKV. We prefer the latter because an index in a TRU can be created once then read only, which significantly simplifies the index implementation and reduces index maintenance overhead. Various types of indices can be employed. For example, if the number of distinct values of the event attribute is low, we can employ a bitmap index [13], and store a bitmap for every value. Each bitmap can be stored efficiently using the scheme as in Section 3.

Other Joins. LogKV is optimized for joins with time windows. From our experience, we expect that it is rare to process other types of joins between event log data that do not use time windows. However, if it is desirable to optimize event log data layout for other types of joins (e.g., location window joins), the same optimization strategy can be applied. One can also take advantage of data replication so that one replica is optimized for one type of joins (e.g., time window joins), while the other replica is optimized for another type of joins (e.g., location window joins). In such situation, the log shuffling step will need to perform both types of shuffling.

5. EXPERIMENTAL EVALUATION

In this section, we present preliminary evaluation results for understanding the performance of *LogKV*. We study (1) log ingestion throughput, (2) storage cost, (3) selection query performance, and (4) window join query performance.

5.1 Experimental setup

Implementation. We implemented an initial prototype in Java, supporting the basic functionality of *LogKV*. It consists of the following five modules:

- IngestKV, which receives log events and sends buffered TRUs to TimeRangeKV. While receiving an event record, IngestKV first serializes the event and then stores it to a log structured file, which is created per TRU on a node. IngestKV maintains the TRU meta-data (such as TRU id, and the location of the log structured file) in a memory structure. Once the log ingestion of the current TRU completes, the TRU's meta-data structure is added to a waiting list to be shuffled to the destination TimeRangeKV. After the shuffling of a TRU is done, IngestKV removes the TRU meta-data structure and the log structured file.
- Shuffle Receiver in TimeRangeKV, which receives shuffled TRUs, merges them, compresses merged log events, and inserts the compressed events into the underlying key-value store (i.e. Apache Cassandra 1.0.6 in our implementation). Shuffle receiver first de-serializes each received log structured TRU and stores the TRUs in memory. When it has received all the local TRUs from all the worker nodes, it merges the local TRUs into a global TRU using the sort merge algorithm.
- Query Proxy, which runs on every worker node, accepts queries from clients, divides queries into sub-queries, and sends sub-queries to their corresponding query processors on worker nodes.
- Query Processor in TimeRangeKV, which processes subqueries in the local worker node and returns results to the Query Proxy.
- Coordinator, which maps the log source to worker nodes, and manages the replication operation. We use Apache ZooKeeper (ZK) to protect the coordinator against node failures. The coordinator data is stored in an ZK instance, which is distributed, consistent and highly available. There is one active coordinator and multiple standby coordinators in LogKV. The active coordinator creates an ephemeral node in ZK. If the active coordinator fails, the ephemeral node will be automatically deleted after a configurable timeout. When the other standby coordinators detect the failure of the active coordinator, they will attempt to acquire a lock. If a standby coordinator successfully creates the lock, it will become the new active coordinator. ZK guarantees that at most one coordinator is active at any time.

To improve the efficiency of TRU serialization for transmission, the class TimeRangeUnit implements a Writable interface. Shuffle Receiver implements compression using java.util.zip package.

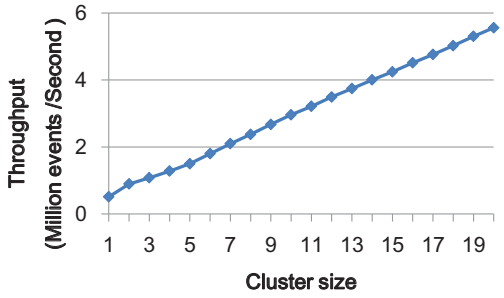

Figure 4: Log ingestion throughout.

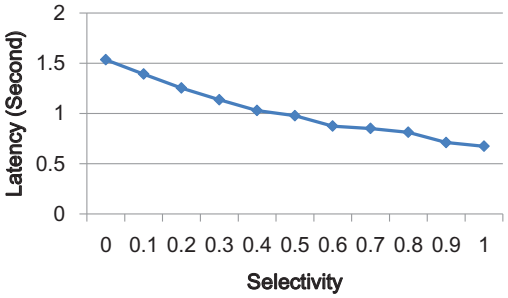

Figure 6: Selection query performance.

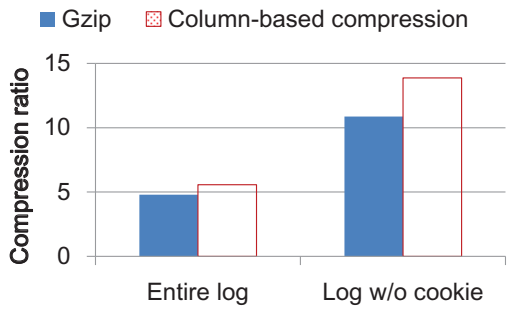

Figure 5: Compression ratio in TimeRangeKV.

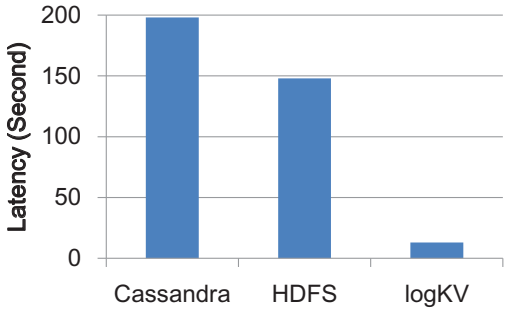

Figure 7: Time window join query performance.

In addition to the above modules, we are still working on the Query Processor in IngestKV. The reported experimental results do not contain this module.

Evaluation Environment. We perform the experiments on a cluster of 10 nodes by default. To evaluate the scalability of LogKV, we use a cluster of 20 nodes. Every node is an HP ProLiant BL460c blade server equipped with two Intel Xeon X5675 3.06GHz CPUs (6 cores/12 threads, 12MB cache), 96GB memory, and a 7200rpm HP SAS hard drive. The blade servers are connected through a 1Gb ethernet switch. The blade servers are running 64bit Red Hat Linux 2.6.32-220 kernel. We use Oracle Java 1.7 in our experiments.

Workload Generation. We use a real-world log event trace from a popular web site in our experiments. We emulate high-throughput log sources by running a log event generator on every worker node that sends log events to IngestKV on the same node. In the storage cost evaluation, we use the original real-world log data. In the other experiments, for generating a large number of log events, we generate synthesized data based on the real-world log (by repeatedly using the data). An event includes the following fields: event ID, timestamp, source IP, and i other fields ($i = 0, 1, ..., 8$). The average event size is 100 bytes.

5.2 Experimental results

Log Ingestion Throughput. Figure 4 measures the maximum sustained log ingestion throughput varying the number of worker nodes (N) from 1 to 20 in LogKV. We set $M = N$ to obtain the maximum ingestion throughput. To reach stable performance fast, we set TRU to be 10 seconds in this experiment. As shown in Figure 4, the Y-axis is the number of ingested events per second. We see that the log ingestion throughput increases linearly as the number of worker nodes grows. The randomized shuffling algorithm works well. When N=1, the per-node throughput is higher than the other points. This is because there is actually no shuffle in this case.

From the figure, we estimate that LogKV achieves an ingestion throughput of about $0.28N$ million events per second, where N is the number of cluster nodes. As an event record size is about 100 bytes, LogKV can sustain about 28 MB/s maximum log ingestion bandwidth per worker node. Therefore, the design goal of 1.2 GB/s aggregate log ingestion throughput can be achieved with about 43 machines that are fully devoted to log ingestion. This gives a lower bound of the actual number of nodes in a design, whose choice must also consider query performance.

Storage Cost. We load 409 MB of log events into a single *LogKV* node. Figure 5 reports the compression ratio measured as input log size divided by the storage size used by the log. We compare plain Gzip implementation and our column-based compression scheme. We use the GZIPInputStream and GZIPOutputStream of java.util.zip in both implementations. Figure 5 shows two sets of bars. The left set of bars show the compression ratios for the entire log. We see that the compression ratios of Gzip and column-based compression are quite similar. This is because the cookie fields in the log events contain almost distinct long strings that results in very low compression ratios. The right set of bars compare the two schemes when the cookie fields are removed. We see that our column-based compression schemes achieve significant improvement over plain Gzip, and it achieves about 15:1 compression ratio.

Selection Query Processing Latency. In the next experiment, we use the selection query which selects events whose given attribute is greater than a specified value (θ) in 1 or more randomly selected TRUs. We vary θ to obtain different selectivity, which is *one minus the ratio of returned records to the number of total records*. In this and all the following experiments, N is fixed to 10. Figure 6 compares the query response time varying the selectivity. The response

time decreases as the selectivity increases. This is because with higher selectivity, fewer records are returned to the client, which results in lower network cost and computation overhead. Overall, the selection query processing latency is quite low because of the following two features in our design : (1) the column based store scheme in TimeRangeKV, which supports retrieving and processing only the columns used in a query; and (2) distribution of TRUs across worker nodes, which support parallel processing.

Window Join Query Processing Latency. In the next experiment, we compare the efficiency of a window join operation across three schemes: (1) MapReduce when event log data are stored in Apache Cassandra; (2) MapReduce on Hadoop distributed file system (HDFS); and (3) our solution LogKV. The query conducts a self-join using the source IP field as the join key within 10-second time windows. For (1), we directly store events in Cassandra. The event ID is used as the key and the other fields are stored as separate columns. For (2), there will be one HDFS file per worker node, and events are naturally stored in an HDFS file in time order. Note that it is non-trivial to merge all the events into a single HDFS file upon receiving the events, which will require the same log shuffling mechanism in LogKV. We implemented the map-reduce repartition join [4] in Apache Hadoop 1.0.0 for (1) and (2). We use the CassandraColumnFamilyInputFormat as the input format of the self-join for (3), and the TextInputFormat as the input format for (2). In (3), LogKV performs the self join per TRU using a hash join based algorithm. As shown in Figure 7, LogKV reduces the latency of Cassandra and HDFS by a factor of 15 and 11, respectively. This is because LogKV reduces event data shuffling in a time window based join by storing all event records within a time range in a TRU.

6. CONCLUSION

This paper presents the LogKV system that leverages a key-value store for supporting the storage and query processing of massive log data. Our design is motivated by the observations that numerous analytics tasks and query processing jobs in log data are executed via extracting key-value pairs from the raw log events as a first step (e.g., the popular log management system in ArcSight [3]). Thus, it is only a natural choice to use existing key-value stores as the underlying storage engine, and design a log-data management system on top of it based on a cluster of commodity machines. Our design also considers important problems such as fault tolerance, durability, reliability, robustness, bursty data-arrival rates, and load-balancing among different nodes. We have also explored effective buffering and data compression techniques to improve the overall performance in LogKV. Experimental results on large log data have verified the effectiveness of our approach in supporting flexible log data storage and efficient query processing for different types of query workloads.

7. ACKNOWLEDGMENT

We would like to thank Haiyan Song, Wenting Tang, and Marylou Orayani from HP ArcSight for many interesting discussions on event log processing and management. We thank the anonymous reviewers for their insightful feedbacks. The third author was supported in part by an HP Labs Innovation Research Program award.

8. REFERENCES

[1] *First USENIX Workshop on the Analysis of System Logs*. USENIX Association, 2008.

[2] Amazon SimpleDB. http://aws.amazon.com/simpledb/.

[3] ArcSight. www.arcsight.com/Logger.

[4] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD*, 2010.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[6] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003.

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

[8] T. Dawson. Performing network & security analytics with hadoop. In *Hadoop Summit*, 2012.

[9] 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. In *SOSP*, 2007.

[10] L. Golab, T. Johnson, J. S. Seidel, and V. Shkapenyuk. Stream warehousing with datadepot. In *SIGMOD Conference*, pages 847–854, 2009.

[11] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *Operating Systems Review*, 44(2), 2010.

[12] A. J. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *DSN*, 2007.

[13] P. E. O'Neil. Model 204 architecture and performance. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, pages 40–59, 1989.

[14] J. Srivastava, R. Cooley, M. Deshpande, and P.-N. Tan. Web usage mining: discovery and applications of usage patterns from web data. *SIGKDD Explor. Newsl.*, 1(2), 2000.

9. APPENDIX: DEMONSTRATION

We demonstrate the LogKV system using real-world log data of different formats and query workloads. In particular, our demonstration focuses on (1) user interface; (2) load-balancing in LogKV; (3) the throughput rate in IngestKV; (4) the storage cost in TimeRangeKV; (5) the efficiency of query processing for different query workloads; and (6) the reliability and durability of LogKV.

Our current implementation of the LogKV system is based on Cassandra, which we use in our demonstration as well. Note that the design of the LogKV system can be easily migrated to any other key-value stores. We use web server logs, Unix system logs, and other log data (available from HP ArcSight Logger) during our demonstration. To simulate the bursty ingestion, we use an event generator as log source in each worker node.

User Interface. We demonstrate a graphical user interface (GUI) of LogKV. The main part of GUI consists of: (1) Data source section, which shows the status and statistics of each log data source; (2) Worker node section, which shows the status and statistics of each worker node; (3) Shuffling process animation, which animates the real-time data shuffling process; (4) System statistics dashboard; and (5) Query interface, where a user could input query, view the query execution process and the results.

Load Balancing in LogKV. To show the effectiveness of LogKV in processing incoming log events, we will display the network and CPU usage of each cluster node to demonstrate that our design indeed avoids "hot spot". We demonstrate this in the following three ways:

- Add an event log source. We add a new event log source using the menu in the data source section. It will display a log source configuration dialog, where we can specify the log parser, ingest rate, whether it is dividable, etc. After configuring the new log source, LogKV will map the log source to worker node(s). The newly added log source and its associated worker node(s) will be highlighted in the data shuffling animation panel. In this panel, each rectangle node represents a cluster node; and each circle node represents a log source. We will add a batch of log sources to the system one by one, including both dividable and individable log sources. We could see the difference of the mapping schemes between dividable log sources and individable log sources.
- Real-time dashboard. It shows the real-time network bandwidth and CPU usage of each node using a bar chart. Each bar group consists of two bars, which represent the network bandwidth usage and the CPU utilization of a worker node, respectively.
- Statistics. After adding several log sources, the statistics panel shows the historical network bandwidth usage and the CPU usage of each node in every time interval.

We will also show the effect of the parameter M by varying it from 2 to N. While a new M is configured, the LogKV needs to be restarted, we then repeat the aforementioned step to add the data sources one by one. We could see the load balancing status both in the real-time dashboard and in the statistics panel. We could also export the dashboard bar chart for each M for comparison purpose.

Throughput Rate in LogKV. We will vary the incoming event log rates and the number of nodes in the cluster, to demonstrate the high ingestion throughput rate that LogKV can achieve. In this process, we use the log event generator in Section 5 to simulate different ingestion rate.

- Vary incoming rates. We change the incoming rates using the menu in the log source section. We will show the throughput of the system and each worker node using the dashboard and the statistics panel.
- Vary number of nodes in the cluster. We change the number of nodes in the cluster and then restart LogKV. We will show the throughput of the system and each worker node using the dashboard and the statistics panel.

In this process, we will also examine the effect of the parameter M (i.e., how much data we buffer locally at a node). While a new M is configured, LogKV is restarted, we then repeat the aforementioned approach varying the number of nodes in the cluster and varying the incoming rates.

Storage Cost in TimeRangeKV. The next component in our demo is to illustrate how TimeRangeKV stores incoming records (transferred from IngestKV instances) into the underlying key-value store, and how well our compression scheme performs.

- The storage part of the dashboard panel shows the total size of used storage. By selecting a node in the worker nodes section, we could see the used storage size of this worker node.
- For comparison purpose, we will setup another key-value store instance, where log events are directly inserted into the store. We calculate the storage cost of this key-value store instance in each worker node.

Query Processing in LogKV. We will use both synthetic and real query workloads to demonstrate the query efficiency in LogKV. We compare LogKV with a naïve approach that stores event log data in a key-value store directly (see our discussion in Section 5). We will demonstrate selections and window-joins. Audience will have the opportunity to issue queries of their choices, and interact with the system during this part of the demonstration.

After a query is input for processing, the animation panel will show the query execution process, such as which worker nodes are involved and the processing progress of each node. The statistic panel will also show the detailed execution cost. We will issue the same query to another key-value store instance, which directly stores the log events, for comparing its execution time with that of LogKV.

Reliability and Durability in LogKV. Lastly, we will demonstrate the reliability and durability of LogKV by purposely disabling some node while LogKV is running. Our design leverages data replication, and the underlying key-value store for handling system failures. We will demonstrate the following two cases:

- Failure of a coordinator node. We will disable the active coordinator node. One of the standby coordinators will become active soon. The service keeps alive in this failover process.
- Failure of a worker node. The audience will have the opportunity to select a worker node to disable. We will see that: (1) LogKV keeps working; and (2) the log source mapping is automatically changed to meet the replication and load balance requirements.

  We will also demonstrate that the query result is correct even if a node fails. We first issue a query and remember the result. Then, we disable a worker node, and issue the same query again. We will see the same result.