2018 IEEE 34th International Conference on Data Engineering

Waterwheel: Realtime Indexing and Temporal Range Query Processing over Massive Data Streams

Li Wang*†, Ruichu Cai†, Tom Z. J. Fu*†, Jiong He*, Zijie Lu†, Marianne Winslett‡, Zhenjie Zhang*
* *Advanced Digital Sciences Center, Illinois at Singpore Pte. Ltd., Singapore*
Email: {wang.li, tom.fu, jiong.he, zhenjie}@adsc.com.sg
† *Guangdong University of Technology, China*
Email: {cairuichu, wslzj40}@gmail.com
‡ *University of Illinois Urbana-Champaign, USA*
Email: winslett@illinois.edu

***Abstract*—Massive data streams from sensors in Internet of Things (IoT) and smart devices with Global Positioning System (GPS) are now flooding to database systems for further processing and analysis. The capability of real-time retrieval from both fresh and historical data turns out to be the key enabler to the real world applications in smart manufacturing and smart city utilizing these data streams. In this paper, we present a simple and effective distributed solution to achieve millions of tuple insertions per second and *ad-hoc* temporal range query processing in milliseconds. To this end, we propose a new data partitioning scheme that takes advantage of the workload characteristics and avoids expensive global data merging. Furthermore, to resolve the throughput bottleneck, we adopt a template-based index method to skip unnecessary index structure adjustments over the relatively stable distribution of incoming tuples. To parallelize data insertion and query processing, we propose an efficient dispatching mechanism and effective load balancing strategies to fully utilize computational resources in a workload-aware manner. On both synthetic and real workloads, our solution consistently outperforms state-of-the-art open-source systems by at least an order of magnitude.**

***Index Terms*—index; query processing; distributed system;**

I. Introduction

The demand for high-throughput data ingestion and real-time data retrieval are arising quickly with the explosive growth of high-speed data generated by sensors from Internet of Things (IoT) [21] and smart devices with location information [25]. It is also one of the most crucial data processing capabilities to support applications in smart manufacturing and smart city so that system users are able to quickly retrieve historical as well as fresh data on demand. In Figure 1, we present an example application of real-time indexing and query processing, in which the analytical system attempts to analyze the network traffic in real time, with samples collected at a backbone network of a telecommunication company at a very high rate. Typical queries over the streaming data retrieve all sample packets from specified IP ranges and within given time durations in order to identify potential risks of network attacks and pinpoint network failures. Thus, it is crucial for the system to keep the flooding samples immediately visible upon arrival while supporting temporal and range queries at low response latency. Similarly, the data store may also be used to maintain fresh updates from sensors, e.g., vibration sensors on essential equipments [21], while database users (a human or an analytical engine) could identify sensors with readings in particular ranges.

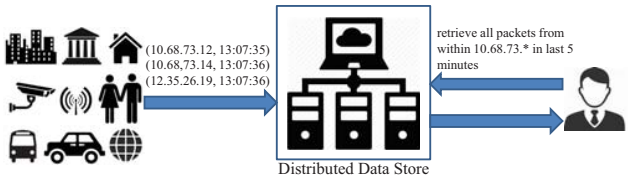


Fig. 1. Backbone network of telecommunication company collects samples of packets from all IP addresses, and analyzes the tuples to detect potential network attacks and equipment failures.

All those applications above require the system to support extremely high throughput of tuple insertion, as well as real-time response to temporal range queries over specified domains. Table I reviews the state-of-the-art systems on their performance guarantees to our target applications. Key-value stores, such as HBase [17] and levelDB [24], organize data tuples as sorted maps and support efficient key range queries. However, range queries over non-key attributes like temporal queries are not first-class citizens and thus cannot be executed efficiently. Besides, although those systems manage to reduce the overhead of updates by using LSM-tree [33] instead of traditional B+ tree, updates still need to be merged with historical data, resulting in significant data merging overhead and limiting the insertion throughput. Time series databases, such as Druid [47], Gorilla [36] and BTrDb [2], are designed for real-time time series data ingestion and low-latency queries with time constraints, but they do not render efficient range queries over non-temporal attributes due to the lack of secondary range indexes.

In this paper, we present Waterwheel, a simple yet effective distributed solution that supports *realtime* indexing over a million tuples per second and *low-latency* queries with key and temporal range constraints. To the best of our knowledge, this is the first framework supporting efficient temporal and range queries and extremely high throughput data insertion simultaneously. The major challenge to Waterwheel is to efficiently index data tuples on both time and key domains while keeping fresh incoming tuples immediately visible to

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00033

TABLE I
COMPARISON OF EXISTING SYSTEMS IN TERMS OF QUERY EFFICIENCY AND INSERTION THROUGHPUT.

Existing systems	Query efficiency		Insertion rate
	Key range	Time range	
HBase, levelDB	✓	✗	low
Druid, Gorilla, BTrDb	✗	✓	low
Waterwheel	✓	✓	high

queries upon arrival. We tackle this challenge by exploiting the characteristics of the workloads in real world, namely *almost ordered arrival* and *slow distribution evolution*, respectively. Firstly, the streaming tuples arrive at the system almost in the same order as they are generated. In our motivation example in Figure 1, for example, the collection of the sample packets almost follows the order on the timestamps of sample packets. Secondly, the distribution of the incoming tuples typically does not change dramatically over time. Given a variety of data sources, e.g., the network packets generated by smart devices, surveillance cameras and smart buildings in Figure 1, the distribution of the sample packets over IP address domain typically evolves in a slow and gentle manner.

Based on the observations and intuitions above, we completely redesign the architecture for realtime indexing and query processing over massive data streams. The key idea behind Waterwheel is to exploit the characteristics of real workload by physically partitioning and maintaining the incoming data into separate data chunks of different temporal and key ranges. Within each data chunk, a template B+ tree index structure is built to enable highly efficient tuple insertion and data retrieval. By using the template, it is unnecessary for the B+ tree to adjust the structure of inner nodes, thus saving index maintenance overhead and achieving high performance. This simple bi-layer data index scheme enables the system to easily re-scale based on the varying workload. To improve the performance and robustness of our system, we devise a template update method and a dynamic key partitioning mechanism so that the system can adapt to workload dynamics. To fully utilize the computational resources in query processing, we propose a new query dispatch algorithm which retains load balance and data locality simultaneously. Overall, Waterwheel achieves significant performance improvement over state-of-the-art open-source solutions in the literature. To summarize, we list the major technical contributions as follows:

1) We propose a generic bi-layer index architecture for million tuple insertions per second and millisecond temporal and key range query processing.
2) We design an enhanced template-based B+ tree which significantly reduces indexing maintenance overhead and achieves high concurrency.
3) We design a distributed query dispatch algorithm and load balancing mechanism to fully utilize the computational resources in the cluster.
4) We evaluate our prototype system with both synthetic and real-world workloads, and demonstrate significant performance improvement against state-of-the-art solutions in the literature.

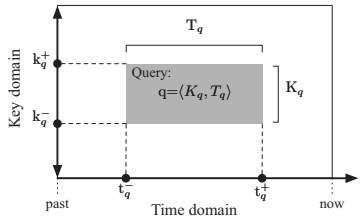


Fig. 2. A typical ad-hoc temporal and key range query.

The rest of the paper is organized as follows. Section II presents the data model and overviews our system. Section III describes the data partitioning schema and explores optimizations to achieve high-throughput data insertion. Section IV discusses parallelized query processing and optimization opportunities. Section V presents the fault tolerance mechanism in Waterwheel. Section VI evaluates our solution with experiments. Section VII reviews related work. Section VIII concludes this paper and highlights future research directions.

II. PRELIMINARY

In this section, we introduce the data model and assumptions made in this paper, and give an overview of our framework.

A. Data Model, Query and Assumptions

We focus on the scenario where data tuples are continuously streamed into the system at extremely high rate and realtime queries are issued to retrieve both recent and historical data under specific key and temporal ranges on the fly. To be precise, a data tuple d is a triplet $d = \langle d_k, d_t, d_e \rangle$, where d_k, d_t and d_e are (unnecessarily unique) index key, timestamp and payload of the tuple, respectively. In the rest of this paper, we refer to index key as key for short. The payload of tuple comprises a group of objects, each of which is either a primitive or a user-defined object. We use $\mathcal{K}$ and $\mathcal{T}$ to denote the key domain and the time domain of the tuples, respectively. We assume the timestamps of incoming data tuples are generally in an increasing order. Based on the definition, $\mathcal{K}$ is a fixed domain, while $\mathcal{T}$ grows infinitely. As shown in Figure 2, $\mathcal{K}$ and $\mathcal{T}$ together define a two-dimensional space $\mathcal{R} = \langle \mathcal{K}, \mathcal{T} \rangle = \{\langle k, t \rangle | k \in \mathcal{K}, t \in \mathcal{T}\}$. A *key interval* is defined as $K(k^-, k^+) = \{k \in \mathcal{K} | k^- \leq k \leq k^+\}$. Similarly, a *time interval* is defined as $T(t^-, t^+) = \{t \in \mathcal{T} | t^- \leq t \leq t^+\}$. Given a pair of key interval $K(k^-, k^+)$ and time interval $T(t^-, t^+)$, a rectangle $r = \langle K, T \rangle = \{\langle k, t \rangle \in \mathcal{R} | k \in K, t \in T\}$ is uniquely determined in $\mathcal{R}$. We refer to a rectangle in $\mathcal{R}$ as a *region* in the rest of this paper. Given any two regions $r_1 = \langle K_1, T_1 \rangle$ and $r_2 = \langle K_2, T_2 \rangle$, r_1 *overlaps* r_2 if $K_1 \cap K_2 \neq \emptyset$ and $T_1 \cap T_2 \neq \emptyset$. A user query is defined as a triplet $q = \langle K_q, T_q, f_q \rangle$, in which $K_q \in \mathcal{K}$ and $T_q \in \mathcal{T}$ are the selection criteria on key and time domains, respectively, and $f_q : t \rightarrow \{true, false\}$ is a user-defined predicate function which determines whether a tuple t qualifies. As shown in Figure 2, given a user query q, the region determined by the key interval K_q and the time interval T_q is called a *query region*, and the query results is defined as a set of all tuples satisfying both temporal and key range conditions as well as the user-defined predicate f_q.

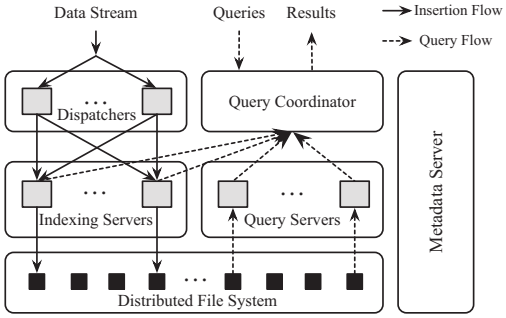


Fig. 3. Framework overview.

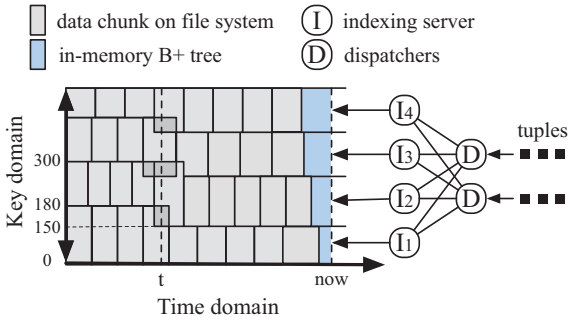


Fig. 4. Global data partitioning in our system.

B. Framework Overview

Waterwheel runs on a cluster of commodity PCs interconnected by local network. In Figure 3, we present the general architecture of the system which consists of a number of independently executed components. *Dispatcher servers* receive continuous incoming data tuples and dispatch them to indexing servers. *Indexing servers* reorganize the incoming data tuples in index structures and periodically flush the received tuples to data chunks in an external distributed file system. *Metadata server* maintains the states of the system, including the partitioning schema of dispatcher servers and the property information of the data chunks for query processing. Based on selection conditions of the queries and metadata information, the *query coordinator* converts a user query into a group of independent subqueries and executes them across the indexing servers (for retrieving fresh incoming data) and/or the query servers (for retrieving historical data) in parallel. The query coordinator aggregates the results from all subqueries and returns them to the user as query results.

In the following two sections, we introduce our new techniques used in high-throughput tuple insertion (in which data flows along the solid lines in Figure 3) and realtime query processing (in which data flows along the dashed lines in Figure 3), respectively.

III. Data Insertion Workflow

In this section, we introduce the global data partitioning, review the template-based B+ tree for efficient data insertion, and then introduce template update and dynamic key partitioning techniques to adapt to dynamics in key distribution.

A. Global Data Partitioning

The objective of global data partitioning is to enable efficient data insertion, while keeping temporal range queries on both historical and recent data as efficient as possible. To this end, our data partitioning schema exploits the characteristics of workloads, namely *almost ordered arrival* and *slow distribution evolution*. Specifically, we partition the key-time space into rectangles, called *data regions*, and store incoming data tuples in their corresponding data regions upon arrival, as shown in Figure 4. Given any query with a specific query region, such partitioning schema can effectively accelerate the query execution by skipping the data regions without any overlap with the query region. More importantly, as data tuples arrive at the system roughly in the order of their timestamps, new arrival tuples are always inserted into the data regions with the latest timestamps, i.e., the rightmost data regions in Figure 4, rather than the historical data regions. Therefore, in contrast to other data structures such as LSM tree that mix the historical and new data, the physical isolation between the recent and historical data in our data partitioning schema avoids costly global data merging, thus enabling high-throughput data insertion.

As shown in Figure 4, we run an indexing server for each unique key interval in the partitioning schema so that new arrival tuples in different key intervals can be injected into the system in parallel. We also run multiple dispatchers to dispatch incoming tuples to the appropriate indexing servers based on the global partitioning schema. To achieve the temporal partitioning, each indexing server accumulates the received data tuples in memory and flushes them as an immutable *data chunk* in the file system once the size of the in-memory tuples reaches a predefined threshold, e.g., 16 MB. Note that the flush operations on different indexing servers are asynchronous and thus the temporal boundaries among different key intervals are not aligned, as shown in Figure 4. To further improve the efficiency of temporal range queries, it is desirable to index data tuples based on the key and time domain within each data region. However, we opt to build the B+ tree on key domain based on the following considerations. First, multi-dimensional index structures such as R-tree [5] incur higher cost in data insertions operations than B+ tree because of their complex structures. Second, real world queries usually involve low selectivity on key domain and high selectivity on temporal domain. Therefore, it is more effective to index tuples on their keys instead of their timestamps. Third, in cases where queries are selective in both domains, we can apply techniques such as Z-ordering [31] to convert two domains into one-dimensional integers and adopt B+ tree for efficient queries.

B. Template-Based B+ Tree

Inserting massive tuples to the B+ tree is known to be inefficient, mainly due to the excessive overhead of node splits in the B+ tree. A large body of bulk loading and bulk insertion techniques [1], [39] has been proposed to amortize the node split overhead by inserting a batch of tuples rather than one tuple at a time. Bulk loading methods, however, are not applicable in our scenario, since they accumulate incoming

tuples into large batches before they are actually inserted into the B+ tree, introducing undesirable delay to the visibility of new data tuples.

To achieve both efficient and realtime insertion, we adapt an enhanced template-based B+ tree index based on our prior work [7], [26]. The intuition behind using index template is that when key distribution of the input data tuples does not vary dramatically over time, the structure of B+ tree index is therefore reusable for most of the time. This motivates us to recycle existing B+ tree structure of previous data chunk to avoid high cost of rebuilding the index from scratch. The implementation of the template-based B+ tree is simple and straightforward. After a running B+ tree is flushed to the persistent storage, we only eliminate the leaf nodes of the tree. The rest of the structure, which we call *template*, is retained and reused for new incoming data tuples. By using template, data tuples are routed to the target leaf node by traversing the tree from root without any modifications to the non-leaf nodes. Besides, the adoption of B+ tree template can also improve the concurrency of insertion and read operations, as the template is read-only during insertion and querying. Although latching or locking is still needed to protect the concurrent reads and insertions on the leaf nodes, such contention is negligible in practice as leaf nodes are usually much more than concurrent insertion or read threads.

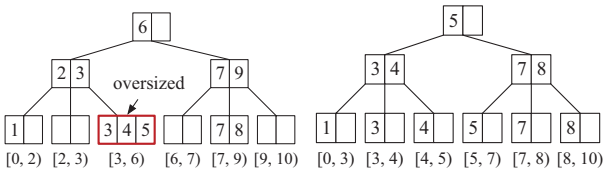
C. Adaptive Template Update

In real applications, the assumption of stable key distribution may not always hold. When key distribution changes, tuples may not be evenly distributed among the leaf nodes, resulting in higher cost of insertion and read in the overflowed leaf nodes. When the key distribution change is minor, it does not introduce noticeable performance penalty. However, as the change accumulates, some leaf nodes may grow too large, introducing undesirable performance degradation. To make our template-based B+ tree robust to unstable key distribution, we propose a template update mechanism, including key skewness detection and boundary updates, as discussed in details below.

1) Key Skewness Detection: Let $K(k^-, k^+)$ be the key interval of the B+ tree. Without losing generality, we assume the tree has exactly l leaf nodes. The structure of the template implies a range partition of $K(k^-, k^+)$ across l leaf nodes: $P = \{K_1, K_2, \dots K_l\}$, in which $K(k^-, k^+) = \cup_{1 \le i \le l} K_i$ and $K_i \wedge K_j = \emptyset$ for any $i \neq j$. We use D to denote the set of tuples in the tree, and thus the desirable number of tuples on each leaf node is $n = |D|/l$. We denote $K_i(D)$ as the set of tuples on leaf nodes i, and propose a distribution skewness factor function, $S(P, D)$, to quantify the skewness of the key distribution in the leaf nodes:

$$S(P, D) = \max_{1 \le i \le l} \frac{|K_i(D)| - n}{n}. \quad (1)$$

When the skewness factor exceeds a predefined threshold (e.g., 0.2), the template of the B+ tree is marked as obsolete. To reflect the distribution change, we pause all tuple insertion threads on this B+ tree and rebuild the template based on the new key distribution using the method described below.



(a) Leaf page overflow (b) Updated template

Fig. 5. A running example of template B+ tree and template update.

2) Key Boundary Updates: Given the number of leaf nodes and the fanout, the structure of the template is purely decided by P, i.e., the range partitioning of the keys across the leaf nodes. The objective of key boundary updates is to reconstruct a new range partition P' in order to minimize the skewness factor function $S(P', D)$, which can be formulated as

$$P' = \arg\min_{\tilde{P}} S(\tilde{P}, D). \quad (2)$$

For a given set of tuples D in the B+ tree, the skewness factor $S(\tilde{P}, D)$ is decided by the largest leaf node under partition $\tilde{P}$. Consequently, the skewness factor can be minimized by using a new range partitioning that evenly divides the tuples across leaf nodes. According to the property of the B+ tree, all keys in the leaf nodes from the leftmost to the rightmost are in an increasing order. We use $\mathbf{k}$ to denote the array of keys and use $k[j]$ to denote the j-th key in this array. Then the new range partition $P' = \{K'_1, K'_2, \dots K'_l\}$ is simply obtained by evenly dividing the keys into l partitions:

$$K'_i = \begin{cases} [k^-, k[in+1]) & i = 1 \\ [k[(i-1)n+1], k[in+1]) & 2 \le i \le l-1 \\ [k[(i-1)n+1], k^+] & i = l \end{cases} \quad (3)$$

Once the new range partition of keys P' is computed, we reorganize the tuples according to the new partitioning and employ a mechanism similar to the traditional bulk loading techniques to build the template upwards, from the bottom inner nodes to the root node. To be more specific, let N be the list of the uppermost nodes that have been updated and M be the list of nodes that are direct parents to the nodes in N and are to be updated. For each parent node, we update its keys according to the new key intervals of its child nodes. We continue this procedure to update the structure of the next upper-layer nodes until the root node is updated. Figure 5(a) shows an example of template-based B+ tree with 6 leaf nodes. The B+ tree is responsible for key interval $[0, 10)$ with an old range partition $P = \{[0, 2), [2, 3), [3, 6), [6, 7), [7, 9), [9, 10)\}$ over the leaf nodes. In the running example, one leaf node (marked in red) overflows with three tuples, while another two leaf nodes are completely empty. To correct the range partitions and balance the insertion loads, our template update algorithm revises the key boundaries in the inner nodes as shown in Figure 5(b). The algorithm first retrieves the list of keys from the leaf nodes, followed by calculation of a new partition $P' = \{[0, 3), [3, 4), [4, 5), [5, 7), [7, 8), [8, 10)\}$ according to Equation 3. Based on the new partition P', we reorganize the data tuples among the leaf nodes and update

the key boundaries of the two inner nodes to {3, 4} and {7, 8}, respectively. Finally, we complete the template update by resetting the key boundaries in the root node to {5}.

D. Adaptive Key Partitioning

While our template-based B+ tree enables efficient data insertion on a single indexing server, it is equally important to guarantee load balancing among indexing servers in order to maximize the overall insertion throughput and to keep a balanced global data partition for better query performance. In this part of section, we discuss the adaptive key partitioning strategy to dynamically control the workload distribution among the indexing servers against key distribution changes.

Based on the observation that the insertion cost per tuple on different indexing servers is statistically the same, the workload of the indexing servers is balanced if the frequency of the keys assigned to each indexing server is equal. In our implementation, each dispatcher samples the key frequencies of its input stream in a sliding widow of a few seconds. A centralized system process periodically calculates the global key frequencies by accumulating values from all dispatchers. If the workload is skewed, e.g., the workload of any indexing server deviates 20% from the average workload, the process adjusts the global key partitioning to balance the workload.

Although the dynamic key partitioning policy is simple and effective, it may result in a partitioning schema with overlapping data regions, which must be handled carefully to avoid introducing incomplete query results. Consider the example shown in Figure 4. The system performs a key repartitioning at time t, which updates the key intervals of indexing server I_1 and I_2 from (0, 180] and (180, 300) to (0, 150] and (150, 300], respectively. After the key repartitioning, data tuples under (150, 180] will be sent to I_2 instead. Consequently, before the two indexing servers flush their in-memory data tuples, they both have data tuples within (150, 180] and their actual key intervals will be (0, 180] and [150, 300), respectively, with overlapping keys. To guarantee query correctness, the metadata server maintains the actual key interval for each indexing server so that the subqueries covering those overlapping keys will be sent to both indexing servers for processing. After their in-memory B+ trees are flushed to the distributed file system, their actual key intervals will be consistent with new key partitioning schema. The data region of the first data chunk generated by an indexing server after a key repartitioning might overlap with the regions of others as shown in the figure. As a result, if a query involves the overlapping region of two data chunks, subqueries on both data chunks will be executed to avoid missing query results.

IV. Query Execution Workflow

In this section, we discuss how to execute user queries across the cluster in parallel, present the execution logic of the subqueries on indexing servers and chunk servers, and introduce our subquery dispatch policy which guarantees load balance and data access locality simultaneously.

A. Query Decomposition

To execute a given query in Waterwheel, the first process is to decompose the query into independent subqueries that can be distributed across the cluster and executed in parallel. We refer to this process as *query decomposition*. As discussed in Section III-A, a user query specifies a region in $\mathcal{R}$ and covers a set of data regions, called *query candidates*, which may contain target data tuples. We generate a subquery for each data region candidates. To efficiently reason about the data regions covered by a given user query, the coordinator maintains a copy of the metadata of the data regions and employs a R-tree [5] to manage the data. Given any query, the R-tree can efficiently retrieve the set of query region candidates, i.e., data regions overlapping with the query region.

Based on the query decomposition mechanism, the overall workflow of query execution is described as follows. For a given new query $q =< K_q, T_q, f_q >$, the query coordinator refers to the R-tree and gets a set of query region candidates, denoted as R_q. For each query region $r_i =< K_i, T_i > \in R_q$, a subquery $q_i =< K_i \cap K_q, T_i \cap T_q, f_q >$ is generated and sent to the appropriate indexing server if the data region has not been flushed to the file system yet, or sent to one of the query servers if otherwise. The policy to select query servers for the subqueries will be discussed in Section IV-C. After all subqueries have been processed, the query coordinator merges the results of all subqueries and returns them to the users.

B. Subquery Execution

To execute a subquery, the system first locates the tuples matching the query criteria on the key domain by traversing the corresponding B+ tree, either in the main memory of an indexing server or on a data chunk, then scans those tuples and filters out those failing to satisfy the selection criteria on the time domain and the predicate function f. Since the data tuples are only indexed on the keys, all the tuples satisfying the query criteria on the key domain needs to be accessed to test against the temporal query criteria, even though only a small number of tuples may pass when the temporal criteria is relatively selective. To avoid unnecessary data access, we partition the time domain into mini-ranges and use a bloom filter for each leaf node as a sketch representing the mini-ranges covered by the tuples in the leaf node. The bloom filters are associated with the references to the leaf nodes on the last-level inner nodes. By using the bloom filters, the system has a high probability to skip the leaf nodes without any tuple satisfying the temporal query criteria.

When the subquery is on a data chunk, the system needs to read data from the distributed file system, which introduces dominant cost in subquery evaluation due to the expensive network and disk I/Os. Consequently, we reduce the cost by keeping the frequently accessed data in memory. We regard a template or a leaf node as the basic caching unit and employ LRU policy [32] to evict the old caching units. To maximize the effectiveness of caching, our system prefers to dispatch subqueries on the same data chunk to the same query server, as will be discussed in next subsection.

C. Subquery Dispatch

For any given query decomposed into a set of subqueries, our subquery dispatch algorithm aims to achieve (a) *load balance*, (b) *cache locality* and (c) *chunk locality* simultaneously, in order to minimize the query response time. Specifically, load balance requires the system to appropriately assign subqueries to the query servers such that the idle time on all the servers is minimized [43]. Cache locality can be achieved by consistently dispatching subqueries on a particular data chunk to the same query server, such that the data loaded into the cache by previous subqueries can be reused by subsequent subqueries. For a given subquery, the corresponding data chunk may reside in one or more nodes in the cluster. For instance, for each data chunk, HDFS by default maintains replicas across three random nodes. Chunk locality can be achieved by dispatching a subquery to the query servers where the data chunk is located, so that the data chunk can be accessed locally.

With the goals above in mind, we design a *locality-aware dispatch algorithm* (LADA), as demonstrated in Figure 6. Given a query with a set of subqueries, the algorithm maintains all the unprocessed subqueries in a hash set, called *pending set*. For each query server, it also maintains a preference array containing all the subquery IDs, indicating the order in which the query server bids for the unprocessed subqueries. The construction of the preference arrays is the key process in our algorithm and will be discussed shortly. When the algorithm starts, each query server repeatedly tries to bid for an unprocessed subquery from the pending set and executes the subquery before the next bid, in the order specified in its preference array. The algorithm terminates when the pending set is empty, at which point all the subqueries has been dispatched. Since an idle query server can always get an unprocessed subquery when the pending set is non-empty, this dispatching algorithm achieves load balance. In the following, we discuss how to achieve chunk locality and cache locality by carefully constructing the preference arrays.

The high-level idea of constructing the preference arrays is twofold. First, to achieve chunk locality, for each query server, we make sure that the subqueries whose data chunks are co-located with the query server rank higher in the preference array so that they are guaranteed to be executed before the others. Second, to achieve the cache locality, we rank the subqueries in the preference arrays such that (a) among different queries, each query server has consistent preference to the subqueries on certain data chunks; and (b) the preference varies from one query server to another. By doing this, among multiple queries, the subqueries on the same data chunk are more likely to be processed by the same query server, thus achieving cache locality.

In particular, for any subquery $q_i \in q$, we denote $S(q_i)$ as the array of query servers that are co-located with the data chunk of q_i and $\hat{S}(q_i)$ as the array of the rest of the query servers. The algorithm shuffles the entries in $S(q_i)$ and $\hat{S}(q_i)$, respectively, using two permutations generated with the chunk ID of q_i as the random seed. Then the algorithm concatenates

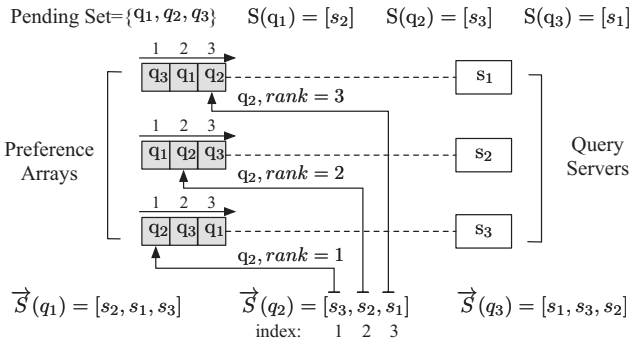


Fig. 6. An example of subquery dispatch mechanism.

the shuffled $S(q_i)$ and $\hat{S}(q_i)$ to get a new array $\overrightarrow{S}(q_i)$. In $\overrightarrow{S}(q_i)$, the smaller offset of a query server indicates that the query server has a higher preference to the subquery q_i. For each query server in $\overrightarrow{S}(q_i)$, we add q_i to its preference array with its offset in $\overrightarrow{S}(q_i)$ as the rank of q_i in the preference array. After all the subqueries of q are processed in the same way, we finally sort the entries in each preference array based on the ranks. Figure 6 shows an example of this process with three query servers and three subqueries. Take subquery q_2 as an example. Its corresponding data chunk is co-located with s_3, so $S(q_2) = [s_3]$ and $\hat{S}(q_2) = [s_1, s_2]$. We assume that after applying the shuffles, $S(q_2) = [s_3]$ and $\hat{S}(q_2) = [s_2, s_1]$, and consequently we get $\overrightarrow{S}(q_2) = [s_3, s_2, s_1]$. Since the offsets of s_1, s_2 and s_3 in $\overrightarrow{S}(q_2)$ is 3, 2 and 1, q_2 will be inserted to the preference arrays of s_1, s_2 and s_3 with the ranks as 3, 2, and 1, respectively. Apparently, chunk locality is retained, as for any query server, the subqueries whose data chunks are co-located with it rank higher in its preference array than the others. Also, cache locality is achieved, as the query servers' preferences to subqueries on the same data chunk are different from each other but are consistent across different queries due to the usage of chunk IDs as the shuffle seed.

D. Dealing with Out of Order Arrival

In real applications, the arrival of data tuples may be sightly delayed due to various factors such as device failures and network congestion. The consequence is that two consecutive data regions generated by the same indexing server may have minor overlap in the time domain. It will not affect the correctness of queries over historical data regions as R-tree is capable of handling data regions with overlap. However, when a late tuple arrives at an indexing server, the left time boundary of its data region may move leftward. If the query coordinator decomposes a query without awareness of the update to the temporal boundaries of the indexing servers, the late tuples may be invisible to the query. To solve this problem without forcing indexing server to notify the query coordinator upon every update to its left temporal boundary, we define a late visibility parameter Δ_t and guarantee that tuples arriving no later than Δ_t seconds are visible to the query. In particular, for the data region in each indexing server, we update its temporal

interval from $T(t^-, t^+)$ to $T(t^- - \Delta_t, t^+)$ in the R-tree. In this way, the query coordinator presumes the presence of late tuples no later than δ seconds in the indexing servers and thus does not miss them if they actually exist.

Note that when some tuples arrive significantly late, data regions may have large overlap in the temporal domain, which increases query latency due to the reduced pruning effect of our partitioning schema. We adopt a simple solution to maintain those delayed tuples separately from normally arriving data tuples to tighten the temporal boundaries of the data chunks containing ordinary data tuples.

V. Fault Tolerance

Waterwheel guarantees consistent query results and no data loss in case of node failures. To reduce the implementation effort, Waterwheel leverages HDFS [18] for the storage of data chunks, Zookeeper [51] for metadata management, and Kafka [22] for messaging passing and tuple replay. To minimize the overhead of achieving fault tolerance, we carefully design the system, such that partial states are maintained in external storage for persistence and other system states are stored internally and reconstructed upon failure. In the following, we present the design choices we make to achieve fault tolerance for the insertion workflow and query workflow, respectively.

Insertion workflow: Since the HDFS and Zookeeper are persistent, once the in-memory B+ tree has been flushed into the file system as a data chunk and the metadata of the data chunk is stored on the metadata server, the flushed data tuples are considered safe. Therefore, to make insertion workflow fault tolerant, we only need to find a way to recover the in-memory B+ tree on a failed indexing server. To this end, we enforce the input queue for each indexing server to be reliable, e.g., by residing on a partition of a topic in Kafka. In Kafka, tuples on each partition are given increasing offsets and tuples starting from a given offset can be replayed on request. When an indexing server flushes the in-memory B+ tree to the distributed file system, it stores the current read offset on the metadata server. As such, when the indexing server fails, it can be re-launched and its in-memory B+ tree can be restored by replaying tuples from the previously stored offset.

Querying workflow: Since the input of any subquery is a data chunk or an in-memory B+ tree, which are persistent or can be restored, subquery can be easily restarted upon failure. Consequently, Waterwheel does not persist any intermediate query results for performance concern. Instead, the coordinator maintains the subquery that is being executed on each query server. In case of a query server failure, the coordinator discards any received query results of the subquery being processed by the query server and re-dispatches it to another query server. To deal with the failure of the query coordinator with minimized state persistence overhead, the system maintains the running queries in the metadata server. When the coordinator fails, the system simply cancels all the ongoing subqueries and re-initializes the queries on a newly created query coordinator.

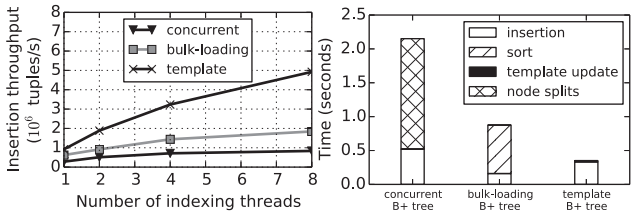
VI. Experiments

Waterwheel is implemented on top of Apache Storm [40] as an application-level topology with about 15,000 lines of Java code. The source codes are available at [45]. We implement the servers in Waterwheel as different operators and define appropriate data routing rules among them. Apache Storm is only responsible for resource allocation of operators and data communication among them. We use HDFS as the underlying distributed file system for data chunk storage. We set 16 MB as the default data chunk size. The cache size for each query server is set to 1 GB. Unless otherwise stated, our experiments are conducted on a 12-node local cluster, with 8 CPU cores (3.4GHz) and 16 GB RAM in each node. The nodes are interconnected by 1Gbps Ethernet network. We also use Amazon EC2 cluster with up to 128 t2.2xlarge instances (nodes) for system scalability evaluation. On each node, we run 2 indexing servers, 4 query servers and 2 dispatchers. HDFS is co-located with our system, with a DataNode daemon running on each node. Since HDFS NameNode daemon, metadata server and query coordinator in our system are computationally light, we randomly assign them to three different nodes.

Our experiments are evaluated on two real datasets, *T-Drive* [48] and *Network*. The T-Drive dataset contains trajectories of 10,357 taxis in Beijing for a week. Each record consists of four attributes, i.e., taxi ID, latitude, longitude and timestamp. The dataset has 15 million records and the total driving distance of the trajectories is 9 million kilometers. Before sending records to indexing servers, the dispatchers preprocess the data by applying z-ordering [31] to transform the latitudes and longitudes into one-dimensional z-codes. After preprocessing, each tuple is 36 bytes in length. We define z-codes as the index key. A query on this dataset is to find taxis appearing in a given geographical rectangle during a specified time interval. Given a query with time range and geographical rectangle, the geographical rectangle is converted to one or more intervals in z-code domain. For each of the z-code intervals, the system issues a query with the time range and the z-code range. The Network dataset, collected by a major telecommunication company, contains over 6 million anonymous website access records. Each record consists of a user ID, source IP, destination IP, a web URL and a timestamp. We define source IP as the indexing key. The data tuples are 50 bytes on average. A query on Network dataset is to find all the access records from a given range of IP addresses within a given time duration. Throughout the experiments, we generate queries with different key and time ranges to control the selectivity of key and temporal domains for various evaluation purposes. We emulate the stream by feeding the tuples to our system based on their timestamps.

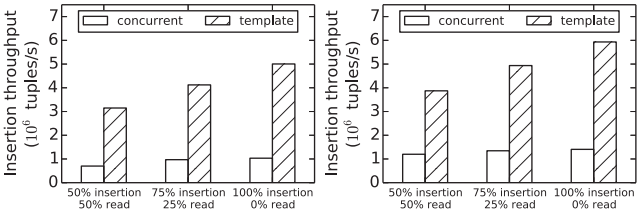
A. Indexing Performance Evaluations

We evaluate the performance of our template-based B+ tree by comparing it with two baseline methods, namely traditional concurrent B+ tree [4] and bulk-loading B+ tree [15]. The concurrent B+ tree is implemented with exactly the same data structures as our template-based B+ tree. The only difference



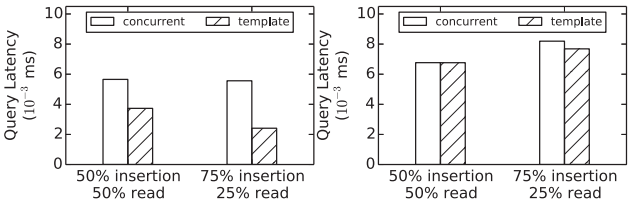
(a) Insertion throughput (b) Insertion time breakdown

Fig. 7. Insertion performance comparison of the three B+ trees.



(a) T-Drive dataset (b) Network dataset

Fig. 8. Insertion throughput under mixed workloads.



(a) T-Drive dataset (b) Network dataset

Fig. 9. Query latency under mixed workloads.

is that it may split nodes during insertions and follows a widely adopted concurrency protocol [4]. The bulk-loading tree is also implemented with the same data structures, but it sorts all the tuples first and then builds the index structure in a bottom-up manner. Since all data tuples in the bulk-loading B+ tree are invisible before the completion of the index build, the query performance of the bulk-loading B+ tree is not evaluated.

1) Insertion throughput: Figure 7(a) shows the insertion throughput of the three indexes with varying number of insertion threads. Since the performance on both real datasets is in a similar pattern, we only show the results on T-Drive dataset to save space. We observe that the insertion throughput of our template-based B+ tree is consistently higher than the baseline methods. As the number of threads increases, the insertion throughput of our method increases significantly while that of the baseline methods does not. This is mainly because in our method, multiple insertion threads do not compete for the write locks on the non-leaf nodes due to the read-only property of the template and thus higher insertion throughput can be achieved with more concurrent indexing threads.

To better understand the underlying reasons behind the huge performance difference among the three methods, we show the breakdown of the insertion time in Figure 7(b). We can see that the concurrent B+ tree spends excessive time in node splits, leading to poor insertion throughput. By sorting tuples before insertion, node splits are avoided in the bulk-loading index and thus the total insertion time is reduced, although additional overhead of data sorting is introduced. Our template-based B+ tree avoids node split overhead by reusing template without introducing noticeable template update overhead, which justifies the performance improvement over the baseline methods.

2) Performance under mixed workloads: To make comprehensive performance evaluation, we compare the insertion throughput and the query latency of template-based B+ tree and the concurrent B+ tree with three representative workloads: a) 100% insertion workload; b) 25% read and 75% insertion workload; and c) 50% insertion and 50% read workload. Each operation is based on a key randomly chosen from the key domain. Figure 8 shows the insertion throughput with three workloads on two datasets. We can see that the insertion throughput in our method is 2 - 3 times higher than that of the concurrent B+ tree. Figure 9 shows the average query latency on two datasets. We find that the query latency in our method is even shorter than that in the concurrent method, although a read operation in our method needs to access more inner nodes on average than the concurrent method due to the use of template. This is mainly because the non-leaf nodes, i.e., template, in our method is read-only during the insertion and thus read operations do not need to acquire locks or latches on non-leaf nodes, consequently achieving higher concurrency and better performance.

3) Template update efficiency: Keep a low template update latency is key to the adaptivity of our method to key distribution fluctuations. To evaluate the template update latency, we run insertion operations on both real datasets and show the average update latency in Figure 10, where x-axis is percentage of the number of existing tuples in the B+ tree to the B+ tree capacity. The results show that the template update latency in both datasets are below 10 ms. As the template update happens infrequently, the latency only consumes a negligible proportion in the total insertion time, as shown in Figure 7(b). The results also demonstrate that the template update latency increases with the percentage of tuples in the B+ tree. That is because with more tuples in the B+ tree, more tuples will be moved among the leaf nodes during the template update, thus incurring higher overhead.

B. Impacts of Data Chunk Size

This group of experiments evaluate the impacts of data chunk size on the performance of Waterwheel. Figure 11(a) shows the system insertion throughput as the chunk size varies. We observe that as chunk size increases, the insertion throughput slightly increases at first but gradually decreases after the chunk size exceeds 32 MB. With larger chunk size, the B+ trees in the indexing servers will be flushed less frequently. Therefore, the overhead of file system I/Os and meta-data update associated with the flush of B+ tree can be reduced, resulting in higher insertion throughput. However, when the chunk size becomes too large, the insertion

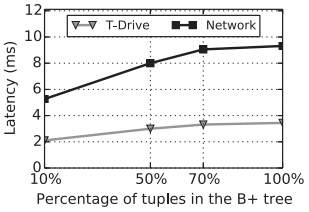
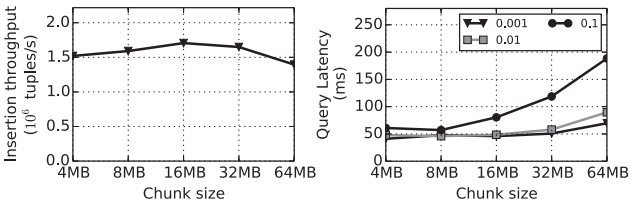


Fig. 10. Template update latency.



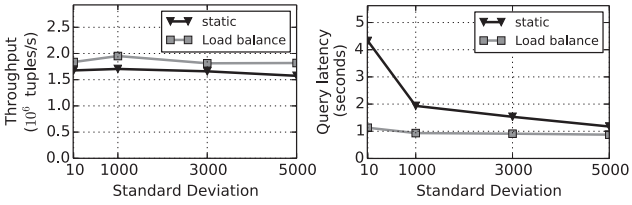
(a) Insertion throughput (b) Subquery latency

Fig. 11. The effects of chunk size to insertion and query performance.

throughput drops because an indexing server has to wait for a relatively long time before its B+ tree is fully filled to be flushed to HDFS, during which the network bandwidth is totally wasted. Figure 11(b) shows the subquery latency with different selectivity of key domain as the chunk size varies along the x-axis. We observe that the query latency increases with the chunk size. That is because, for a subquery with a given selectivity on the key domain, as the data layout in our data chunks allows the system to read only the needed leaf nodes for the given key range, the amount of data needed to read for answering the subquery is proportional to the chunk size. However, the marginal benefit brought by using small chunks size diminishes when the chunk size is smaller than 16 MB. HDFS has an additional delay of 2 - 50 milliseconds associated with each file access, regardless of how much data is read. When the chunk size is small, this additional delay becomes dominant and prevents the query latency from further decreasing. Based on the results shown in Figure 11, we use 16 MB as our default chunk size in the rest of experiments to balance the insertion and query performance.

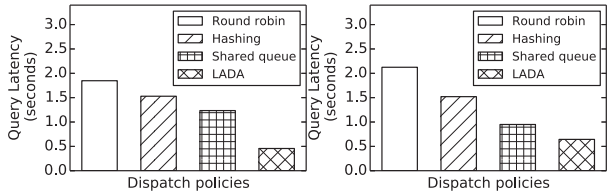
C. Adaptivity Evaluation

1) Adaptive key partitioning: To better evaluate the effectiveness of adaptive key partitioning, we compare the insertion throughput and query latency under workloads with different key skewness. To easily control the skewness, this group of experiments are conducted on a synthetic dataset. The keys of tuples are generated in normal distributions, with $\mu = 0$ and σ ranging from 10 to 5000 to control the key skewness. The data tuple is 30 bytes in size. We generated 1000 random queries with 0.1 selectivity on the key domain and 60 recent seconds as the temporal constraints. The average insertion throughput and query latency with various skewness of key distribution are shown in Figure 12(a) and Figure 12(b), respectively. The results show that both the insertion and query processing performance with adaptive key partitioning feature enabled is consistently higher than that without it. With better load



(a) Insertion throughput (b) Query latency

Fig. 12. Evaluation of the effectiveness of adaptive key partitioning.

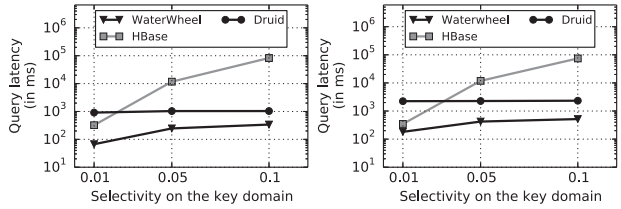


(a) T-Drive dataset (b) Network dataset

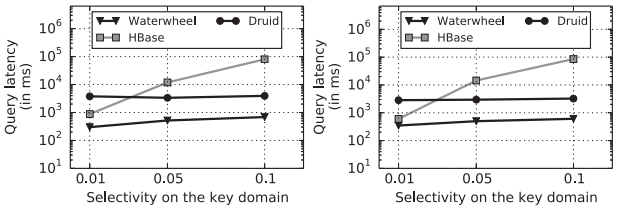
Fig. 13. Query latency under different subquery dispatch policies.

balancing on the indexing servers, the insertion workload can be evenly distributed, resulting in higher insertion throughput. Such balanced workload also contributes to a more balanced global data partitioning, which increases the pruning effect on the subqueries and improves query processing performance. The improvement of insertion throughput is lower than our expectation. That is because, with the adoption of efficient template-based B+ tree, insertion operations become network-bound instead of CPU-bound, which prevents the insertion throughput from further increasing. We expect that the adaptive key partitioning will bring more performance improvement when the system is equipped with network devices with higher bandwidth such as 10Gbps Ethernet or Infiniband.

2) Subquery dispatch: The superiority of our LADA dispatch policy is verified by comparing it with three baseline dispatch policies, namely *round robin*, *hashing dispatch* and *shared queue*. The round robin dispatch policy assigns subqueries to query servers in a round robin manner. In the hashing dispatch, subqueries are hash-partitioned to the query servers based on the corresponding chunk IDs. In the shared queue policy, all subqueries are placed in a global shared queue and each query server picks and executes one subquery from the queue at a time until all subqueries are processed. We conduct 1000 random queries with 0.1 selectivity on both temporal and key domains under different dispatch policies, and show the average query latency for each policy in Figure 13. As expected, the round robin policy performs the worst due to imbalanced workload and poor data locality. The shared queue policy achieves better performance than the round robin policy as it balances the workload among query servers by allowing those fast query servers to help the slow ones. The hashing dispatch policy always assigns subqueries on the same data chunk to the same query server. As such, it retains cache locality and thus shows better query performance than the round robin. Our LADA policy guarantees load balance, cache locality and chunk locality at the same time and consequently outperforms the baseline policies by a substantial margin.



(a) Recent 5 seconds (b) Recent 60 seconds



(c) Recent 5 minutes (d) Historic 5 minutes

Fig. 14. Query latency comparison on Network dataset.

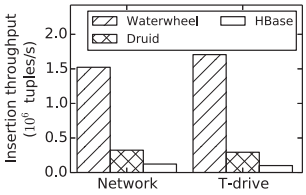
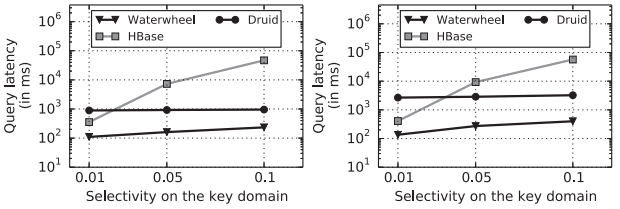
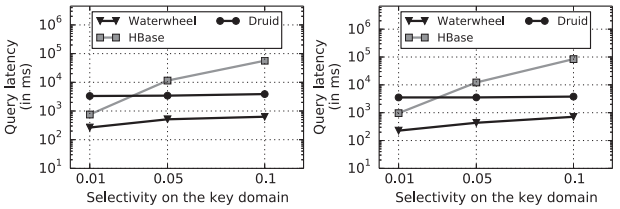


Fig. 15. Insertion throughput comparison between our system and HBase.



(a) Recent 5 seconds (b) Recent 60 seconds



(c) Recent 5 minutes (d) Historic 5 minutes

Fig. 16. Query latency comparison on T-Drive dataset.

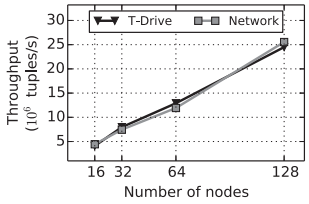


Fig. 17. Insertion throughput as the number of nodes increases.

D. Overall performance evaluation

1) Comparison with state-of-the-art systems: We compare our system with HBase [17] and Druid [47]. HBase stores data tuples as a distributed sorted map and thus supports efficient key range queries. It uses LSM-tree as the underlying data structure for efficient data insertion and updates. Druid is an open-source distributed timeseries data store tailored to realtime queries over streaming and historical data.

We first evaluate the insertion throughput and show the results in Figure 15. The results show that on both datasets, our system is able to insert over 1.5 million tuples per second, an order of magnitude higher than that of HBase and Druid. The results verify the huge benefit of global data partitioning schema in our system, which avoids global data merging by isolating the fresh incoming data from the historical data. We also compare the query latency by varying the key range and temporal range constraints in the queries. In particular, we use four representative temporal ranges, i.e., recent 5 seconds, recent 60 seconds, recent 5 minutes and historical 5 minutes. The time range for the historical 5 minutes is randomly chosen between the system start time and the time when the query is issued. The key ranges are randomly generated so that the selectivity of key domain is 0.01, 0.05 and 0.1, respectively. To guarantee that those queries with the same time and key range are comparable among three systems, we fix the insertion rate as 50K tuples per second, i.e., 50% of the maximum insertion rate in HBase, for three systems during the query evaluation. Figure 14 and Figure 16 show the average query latency with different key range and temporal constraints on Network and T-drive datasets, respectively. In general, the query latency of our system is consistently lower than that of HBase and Druid under various key and temporal constraints. As the selectivity of key domain increases, the performance gap between our system and HBase widens. Since HBase does not support range index on those non-key attributes, all tuples satisfying the key range constraint must be read and tested against the temporal constraint, resulting in higher query latency. In contrast, by using the global data partitioning, our system is capable of bypassing the data chunks out of the temporal constraint and consequently achieves lower latency for all the queries. Similarly, due to the lack of support of range indexes in Druid, all tuples satisfying the temporal constrain should be read and verified against the key range constraint. Therefore, query latency in Druid is high but stable as the selectivity of key domain varies. In conclusion, the results in Figure 14 and Figure 16 verify the importance of supporting both temporal and key indexing on those workloads.

2) Scalability: This group of experiments evaluates the scalability of our system. We employ a 128-node EC2 cluster, where each node (EC2 instance) has 8 CPU cores and 32 GB RAM. Figure 17 shows the maximum insertion throughput of our system as the number of nodes increases. We observe that as the number of nodes grows from 16 to 128, the insertion throughput of our system on both datasets grows approximately linearly. This is mainly because a) the data partitioning schema allows each indexing server work independently and therefore avoids the synchronization overhead; and b) adaptive key partitioning keeps workload balanced among indexing servers so that the system can fully utilize the computation resource and network bandwidth of the indexing servers.

VII. Related Work

Key-value stores, such as Dynamo [12], BigTable [10], Memcached [27] and Amazon S3 [34], are designed to manage massive key-value pairs based on distributed data storage and processing. BigTable [10] and its open-source implementation HBase [17], for example, organize data tuples as multi-dimensional sorted map and support efficient key range query. Our work differs from those systems in that a) our system is an append-only store optimized for high-throughput data inserting and b) we support both key and temporal range search. Claims [44] and Druid [47] are distributed systems for real-time data analytics. Druid shares the similar ideas with Waterwheel in that it physically isolates fresh data from historical data. However, Druid only supports inverted indexes and thus cannot execute key range query efficiently. Vertica [23], Spanner [11], Mesa [16] and Megastore [3] are distributed databases with focus on transactional processing. Different from these systems, our system is an append-only store with strong requirement of real-time data visibility and does not account for supports to transaction processing.

Indexing is a commonly used technique to improve query performance when the target columns are frequently involved in the selection criteria in queries. Index maintenance usually incurs high overhead when inserting tuples at high rates. Bulk loading/insertion amortizes the overhead by inserting a batch of tuples instead of a single tuple into index each time. For example, Achakeev et al. [1] adopt bulk updates to improve the update performance of multi-version B-trees. Unfortunately, those techniques are not applicable in our scenarios where data tuples must be immediately visible upon arrival. Montgomery et al. [30] propose a distributed, RDMA-friendly B-tree that balances network and CPU loading by enabling client-side search based on RDMA and appropriately switches between client-side search and server-side search based on measurements over the workload. In contrast, our system is a generic system without any hardware-specific optimization. Wu et al. [46] adopt bitmap as the basic indexing tool and build a tree structure over the bitmaps across nodes for efficient distributed data retrieval. However, it is difficult to determine the optimal bitmap vector size, which is crucial to the performance of bitmap index. Pedreira et al. [35] design a fast distributed index system that mainly targets data warehouse accessed by hierarchical selection queries. Although the system also supports ad-hoc queries, the performance heavily depends on the granularity of the cells chosen at the initialization of the system. Recently, Braun and Kipf et al. propose analytics in motion, which focus on real-time materialization over pre-specified views on fast data stream at scale of 100,000 records per second [6]. Our system, on the contrary, supports ad-hoc range queries at runtime over data stream with a million tuples per second. LSM-tree [33] and its variants [38], [41] have been widely used in state-of-the-art database management systems, such as HBase [17], MongoDB [29] and Cassandra [9]. The key idea is to maintain the index structure in multiple layers, where a higher layer is kept in a faster storage medium with smaller capacity. However, the insertion performance of LSM-trees is still limited, mainly due to the unavoidable data merging overhead. In this paper, we propose a new bi-layer index architecture with template-based insertion schema. To the best of our knowledge, this is the first work that supports both data ingestion over a million tuples per second and real-time temporal range queries.

A substantial body of work focuses on scheduling tasks with load balancing and locality-sensitive optimizations. Delay Scheduling [49] improves data locality by delaying jobs shortly for a better chance to be scheduled on the preferred nodes. However, it is designed for the scenario where many short and concurrent queries (jobs) are competing resources on the nodes, and therefore it is not applicable in our system which runs a single query at a time to minimize the query response time. Load balancing problem can be viewed as bin packing problem by regarding servers as bins and tasks as balls, and thus can be effectively solved by the heuristic FFD algorithm [20]. However, when the cost of tasks is unknown before scheduling, any offline solutions are not applicable. Extensive work attempts to solve this problem with online scheduling algorithm. For example, Mitzenchacher et al. [28] shows that load balancing can be achieved without prior knowledge of task cost, by dispatching each task to the least-loaded one between two randomly selected servers. [37] is similar to our load balancing strategy in that it maintains a local queue for each process and balances the length of the local queues when necessary. This method is based on the assumption that tasks have the same cost, and thus is not applicable when the cost of tasks (subqueries) varies greatly.

Distributed stream processing systems, such as Storm [40], DRS [13], [14], Elasticutor [42], Spark Streaming [50], Flink [8] and Heron [19], are designed to process massive data streams. Due to their low-latency data processing capability and flexible programming model, our system can be naturally implemented as a high-level application running on one of those systems and relies on the underlying platform for resource allocation, data transmission and fault-tolerance.

VIII. Concluding Remarks

In this paper, we present a simple and effective distributed solution to achieve extremely fast data stream ingestion and low-latency querying. By embracing a suite of new techniques including domain partitioning, template B+ tree, load balancing and query dispatching, our system shows excellent performance on insertion throughput and query response latency and outperforms state-of-the-art system by a substantial margin.

As future work, we will continue our research investigation on two directions. First, as our system is scaled to more computing nodes, the network bandwidth may become a new performance bottleneck. It is necessary to consider bandwidth optimization within domain partitioning and load balance strategies. Second, we will add secondary index structure by bitmap and bloom filters, to enable index retrieval on non-key and non-temporal attributes.

ACKNOWLEDGMENT

This research is partially supported by NSFC-Guangdong Joint Found (U1501254), Natural Science Foundation of China (61472089, 61702109, 61702113), China Postdoctoral Science Foundation (2017M612615, 2017M612613), Natural Science Foundation of Guangdong (2014A030306004, 2014A030308008), Science and Technology Planning Project of Guangdong (2015B010108006, 2015B010131015).

REFERENCES

[1] D. Achakeev and B. Seeger. Efficient bulk updates on multiversion b-trees. *Proceedings of the VLDB Endowment*, 6(14):1834–1845, 2013.

[2] M. P. Andersen and D. E. Culler. Btrdb: Optimizing storage system design for timeseries processing. In *FAST*, pages 39–52, 2016.

[3] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. 2011.

[4] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta informatica*, 9(1):1–21, 1977.

[5] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *ACM Sigmod Record*, volume 19, pages 322–331. Acm, 1990.

[6] L. Braun, T. Etter, G. Gasparis, M. Kaufmann, D. Kossmann, D. Widmer, A. Avitzur, A. Iliopoulos, E. Levy, and N. Liang. Analytics in motion: High performance event-processing AND real-time analytics in the same database. In *ACM SIGMOD*, pages 251–264, 2015.

[7] R. Cai, Z. Lu, L. Wang, Z. Zhang, T. Z. Fu, and M. Winslett. Ditir: distributed index for high throughput trajectory insertion and real-time temporal range query. *Proceedings of the VLDB Endowment*, 10(12):1865–1868, 2017.

[8] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in apache flink®: Consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, Aug. 2017.

[9] Cassandra. https://cassandra.apache.org, 2017.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[11] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[12] 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.

[13] T. Z. Fu, J. Ding, R. T. Ma, M. Winslett, Y. Yang, and Z. Zhang. Drs: dynamic resource scheduling for real-time analytics over fast streams. In *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*, pages 411–420. IEEE, 2015.

[14] T. Z. Fu, J. Ding, R. T. Ma, M. Winslett, Y. Yang, and Z. Zhang. Drs: Auto-scaling for real-time stream analytics. *IEEE/ACM Transactions on Networking*, 25(6):3338–3352, 2017.

[15] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database system implementation*, volume 654. Prentice Hall Upper Saddle River, NJ:, 2000.

[16] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. Dhoot, A. Kumar, and et. al. Mesa: Geo-replicated, near real-time, scalable data warehousing. In *VLDB*, 2014.

[17] HBase. http://hbase.apache.org, 2017.

[18] HDFS. https://hadoop.apache.org, 2017.

[19] Heron. https://twitter.github.io/heron, 2017.

[20] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of computer and system sciences*, 9(3):256–278, 1974.

[21] D. Jung, Z. Zhang, and M. Winslett. Vibration analysis for iot enabled predictive maitenance. In *ICDE*, 2017.

[22] Kafka. https://kafka.apache.org, 2017.

[23] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012.

[24] LevelDB. https://github.com/google/leveldb.

[25] V. C. Liang, R. T. B. Ma, W. S. Ng, L. Wang, M. Winslett, H. Wu, S. Ying, and Z. Zhang. Mercury: Metro density prediction with recurrent neural network on streaming CDR data. In *ICDE*, pages 1374–1377, 2016.

[26] P. Mazumdar, L. Wang, M. Winslet, Z. Zhang, and D. Jung. An index scheme for fast data stream to distributed append-only store. In *WebDB*. ACM, 2016.

[27] Memcached. https://memcached.org, 2017.

[28] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[29] Mongodb. https://www.mongodb.com, 2017.

[30] C. M. K. Montgomery, L. Nelson, S. Sen, and J. Li. Balancing cpu and network in the cell distributed b-tree store. In *Proceedings of USENIX ATC16 2016 USENIX Annual Technical Conference*, page 451, 2016.

[31] G. M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company New York, 1966.

[32] E. J. O'neil, P. E. O'neil, and G. Weikum. The lru-k page replacement algorithm for database disk buffering. *ACM SIGMOD Record*, 22(2):297–306, 1993.

[33] P. O'Neil, E. Cheng, D. Gawlick, and E. O?Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[34] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel. Amazon s3 for science grids: a viable solution? In *Proceedings of the 2008 international workshop on Data-aware distributed computing*, pages 55–64. ACM, 2008.

[35] P. Pedreira, C. Croswhite, and L. Bona. Cubrick: indexing millions of records per second for interactive analytics. *Proceedings of the VLDB Endowment*, 9(13):1305–1316, 2016.

[36] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8(12):1816–1827, 2015.

[37] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In *ACM SPAA*, pages 237–245. ACM, 1991.

[38] R. Sears and R. Ramakrishnan. blsm: a general purpose log structured merge tree. In *ACM SIGMOD*, pages 217–228. ACM, 2012.

[39] A. Silberstein, B. F. Cooper, U. Srivastava, E. Vee, R. Yerneni, and R. Ramakrishnan. Efficient bulk insertion into a distributed ordered table. In *ACM SIGMOD*, pages 765–778. ACM, 2008.

[40] A. Storm. http://storm.apache.org, 2017.

[41] W. Tan, S. Tata, Y. Tang, and L. L. Fong. Diff-index: Differentiated index in distributed log-structured data stores. In *EDBT*, pages 700–711, 2014.

[42] L. Wang, T. Z. Fu, R. T. Ma, M. Winslett, and Z. Zhang. Elasticutor: Rapid elasticity for realtime stateful stream processing. *arXiv preprint arXiv:1711.01046*, 2017.

[43] L. Wang, M. Zhou, Z. Zhang, M. C. Shan, and A. Zhou. Numa-aware scalable and efficient in-memory aggregation on large domains. *IEEE Transactions on Knowledge and Data Engineering*, 27(4):1071–1084, 2015.

[44] L. Wang, M. Zhou, Z. Zhang, Y. Yang, A. Zhou, and D. Bitton. Elastic pipelining in an in-memory database cluster. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1279–1294. ACM, 2016.

[45] Waterwheel. https://github.com/adsc-cloud/waterwheel.

[46] S. Wu, G. Chen, X. Zhou, Z. Zhang, A. K. H. Tung, and M. Winslett. PABIRS: A data access middleware for distributed file systems. In *ICDE*, pages 113–124, 2015.

[47] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli. Druid: A real-time analytical data store. In *ACM SIGMOD*, pages 157–168. ACM, 2014.

[48] J. Yuan, Y. Zheng, X. Xie, and G. Sun. Driving with knowledge from the physical world. In *ACM SIGMOD*, pages 316–324. ACM, 2011.

[49] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys*, pages 265–278. ACM, 2010.

[50] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *ACM SOSP*, pages 423–438. ACM, 2013.

[51] Zookeeper. https://zookeeper.apache.org, 2017.