DITIR: Distributed Index for High Throughput Trajectory Insertion and Real-time Temporal Range Query

Ruichu Cai[◇] Zijie Lu[◇] Li Wang[◇¶] Zhenjie Zhang[¶] Tom Z. J. Fu[◇¶] Marianne Winslett[†]

[◇]School of Computer Science
Guangdong University of Technology
{cairuichu, wslzj40}@gmail.com

[¶]Advanced Digital Sciences Center
Illinois at Singapore Pte. Ltd.
{wang.li, zhenjie, tom.fu}@adsc.com.sg

[†]Department of Computer Science
University of Illinois at Urbana Champaign
winslett@illinois.edu

ABSTRACT

The prosperity of mobile social network and location-based services, e.g., Uber, is backing the explosive growth of spatial temporal streams on the Internet. It raises new challenges to the underlying data store system, which is supposed to support extremely high-throughput trajectory insertion and low-latency querying with spatial and temporal constraints. State-of-the-art solutions, e.g., HBase, do not render satisfactory performance, due to the high overhead on index update. In this demonstration, we present DITIR, our new system prototype tailored to efficiently processing temporal and spacial queries over historical data as well as latest updates. Our system provides better performance guarantee, by physically partitioning the incoming data tuples on their arrivals and exploiting a template-based insertion schema, to reach the desired ingestion throughput. Load balancing mechanism is also introduced to DITIR, by using which the system is capable of achieving reliable performance against workload dynamics. Our demonstration shows that DITIR supports over 1 million tuple insertions in a second, when running on a 10-node cluster. It also significantly outperforms HBase by 7 times on ingestion throughput and 5 times faster on query latency.

1. INTRODUCTION

With the prosperity of mobile social network and location-based services, a large amount of trajectory data is generated every second. To achieve analytical and administrative purposes, it becomes increasingly desirable for data store systems to support low-latency temporal and range queries over the fast trajectory data stream. For instance, one of our business clients requires to record real-time GPS data at around 800 thousand tuples per second and to run interactive queries, such as getting all GPS data in a certain geographical region and within the last 5 minutes. However, state-of-the-art data stores, such as HBase, cannot achieve high data insertion rate and efficient temporal range queries simultaneously. In particular, to enable low-latency temporal range queries over a massive amount of data, index has to be created on the temporal and spatial columns. As a result, incoming tuple insertion is coupled with index update, which is known to be expensive and therefore prevents high-throughput insertion. The index update cost cannot be simply reduced by employing batch-based insertion in our scenario, because data tuples are required to be immediately visible on their arrivals.

To solve this problem, we propose DITIR, a distributed append-only store capable of inserting trajectory data at high rate and answering temporal range queries on both newly arriving data and historical data. The key idea behind our prototype system is to design a storage schema that physically partitions incoming tuples on their arrivals based on their locations and timestamps, so that only a subset of data partitions are involved for a temporal range query. We employ Z-order [11] to convert two-dimensional spacial values to one-dimensional z-codes. Within each partition, data tuples are maintained in a B+ tree based on the z-codes, for efficient spacial search. To enable efficient insertion without batching, we propose a template-based insertion schema for B+ tree, which avoids the index node splits and improves both insertion and search performance. Based on the partitioning, for each query, DITIR generates a set of independent subquries on the involved data partitions and executes them across the cluster in parallel. In addition, load balancing mechanism is proposed for both data insertion and query evaluation, to better utilize the computational resources.

The major contributions of this paper are as follows:

1. We propose a distributed append-only store that supports efficient trajectory data insertion and real-time temporal range queries.
2. We introduce a template-based B+ tree insertion schema to enable efficient indexing over fast data stream.
3. Load balance mechanism is proposed to guarantee reliable performance of data insertion and queries against workload dynamics.
4. We implement DITIR and show the superiority of our system by comparing with HBase.

Proceedings of the VLDB Endowment, Vol. 10, No. 12

2. RELATED WORK

Distributed data stores, such as Dynamo [9], BigTable [8], CLAIMS [16], HBase [1] and MongoDB [5], store and manage massive amounts of data by distributed data storage and processing. BigTable [8] and its open-source implementation HBase [1], for example, organize data as distributed multi-dimensional sorted maps and provide efficient and scalable queries. Our work differs from those systems in the sense that our framework is optimized for high-throughput trajectory data insertion and efficient evaluations of temporal range queries. Our framework can be extended to support more complex queries, such as join and aggregation [15], based on the results of the temporal range queries.

Indexing is a commonly used technique to improve query performance when some columns are frequently used in the query criteria. However, the index maintenance overhead is too high to afford when inserting tuples at high rate. Bulk loading/insertion amortizes the overhead by inserting a batch of tuples at a time rather than inserting each tuple individually. For instance, [7] propose a distributed bulk loading method with optimizations on load balancing across the partitions. Unfortunately, those techniques are not applicable in the scenarios where data tuples should be immediately visible on their arrivals. To improve the insertion performance without batching, LSM-tree [12] and its variant [13] have been widely used in many state-of-the-art database management systems, such as HBase [1], MongoDB [5], Cassandra [2] and InfluxDB [4]. The key idea is to maintain B+ trees in two or more layers, where the tree in a higher layer is kept in a faster storage medium with smaller capacity. However, the insertion performance of LSM-trees is still limited, due to the inefficient insertion to its first in-memory layer. In this paper, we propose a new template-based insertion schema in DITIR to achieve high insertion performance.

3. FRAMEWORK OVERVIEW

3.1 Data Model and Assumptions

In this paper, we focus on the applications where a data tuple is in the form of $< x, y, t, e >$, where x, y, t and e are the longitude, latitude, timestamp and payload, respectively. Before insertion, an input data tuple $< x, y, t, e >$ is converted into $< z, t, e >$ by applying Z-order [11] on x and y, where z is called a *location key* or *key* for short. We assume tuples arrive in the increasing order of their timestamps. Time and key form a two-dimensional space $\mathcal{R}$. A user query is based on a range of keys and a range of timestamp. We call a time duration as a *time interval* and a range of keys as a *key interval*. Given any time interval and any key interval, a rectangle, called *region*, is uniquely determined in $\mathcal{R}$.

3.2 Framework Architecture

DITIR runs on a cluster of shared-nothing commodity servers, interconnected by high-speed local network. The architecture of our system is shown in Figure 1. *Dispatcher servers* receive the continuous input data tuples and dispatch them to the insertion servers. The insertion servers organize data tuples and store them as data chunks on the distributed file system. The metadata server maintains the state of the system, including the partitioning schema of dispatcher servers and the metadata of the data chunks. Based on the query criteria and the information on the metadata server, the query coordinator converts a user query into independent subqueries and executes them across the insertion servers and the query servers in parallel.

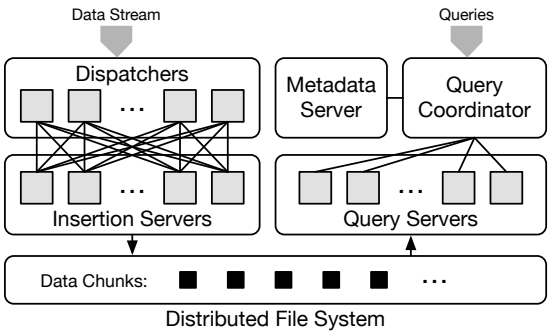

Figure 1: Architecture Overview.

4. DATA INGESTION

4.1 Data Partitioning

The objective of data partitioning in our framework is to enable efficient key-range and temporal queries while making data insertion as efficient as possible. This is achieved by physically partitioning data tuples based on the keys and the timestamps into regions, called *data regions*, as they arrive. By doing this, query evaluation can be effectively accelerated by skipping the data regions that do not overlap with the query criteria. To achieve range partition on the key domain, the incoming data tuples are range-partitioned to the ingestion servers by the dispatchers, such that each ingestion server is responsible for inserting data tuples in a unique key range. Each ingestion server stores its input data tuples in its in-memory B+ tree. To achieve range partition on the time domain, an ingestion server flushes its B+ tree into the distributed file system as a data chunk when the B+ tree reaches a predefined capacity. Since the timestamps of the incoming tuples are in increasing order, data chunks generated by the same ingestion server are naturally in a range partitioning of the time domain.

Such a data partitioning schema is insertion-friendly, because insertion servers work independently and data chunk will never be modified once generated. To further improve the ingestion performance, we make optimizations to improve the performance of a single insertion server and balance the workloads among insertion servers, as will be discussed in the following subsections.

4.2 Template-Based B+ tree

Inserting a large number of tuples to a B+ tree is known to be very inefficient, due to the overhead of node splits. Unfortunately, bulk load and bulk insertion techniques, such as [6] and [14], are not applicable in our scenario, because tuples should be visible to the queries once they arrive. To achieve both high-throughput and rapid insertion, a template-based B+ tree insertion schema is proposed in our prior work [10]. The intuition is that the key distribution of the input data tuples typically does not change much in a period of time. This provides us opportunities to utilize the B+ tree structure from previous data chunks to build the B+ tree for the new one, avoiding building the index from scratch. The implementation of template-based B+ tree is simple and

straightforward. After a B+ tree is flushed to the persistent storage, the structure containing the root node and the inner nodes, called *template*, is kept and reused to construct a new B+ tree with all the leaf nodes empty for the next data chunk. Then the new incoming data tuples are inserted into the new B+ tree as normal.

Given a relatively stable key distribution, this method is efficient and scalable with multiple read and insertion threads. However, key distribution may not be always stable in real applications. Key distribution changes introduce skewed tuple distribution across the leaf nodes, making both insertion and read inefficient in the over-sized leaf nodes. To handle workload dynamics, we propose a new template update method, as will be discussed blow.

Key Skewness Detection: Let $K(k^-, k^+) \in \mathcal{K}$ be the key interval that the current B+ tree is responsible for. Without losing generality, we assume the template-based B+ tree has 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 inserted to the tree, and thus the desirable number of tuples on each leaf node is $n = |d|/l$. Let $d(k_i)$ denote the set of tuples on any leaf node i. We propose distribution skewness, $S(P, d)$, to quantify the skewness of the tuple distribution in the leaf nodes:

$$S(P, d) = \max_{1 \le i \le l} \frac{|d(k_i)| - n}{n} \tag{1}$$

When the data distribution skewness factor s exceeds the predefined threshold, e.g., 0.2, the template of the B+ tree is considered as improper. As a consequence, we pause inserting tuples to the B+ tree until we update the template based on the new key distribution.

Template Update: Given the number of leaf nodes and the fanout, the structure of the template is purely decided by P, the range partition of keys across the leaf nodes. Consequently, the objective of template update is to get a new range partition P' that minimizes the skewness factor s, as formally defined as:

$$P' = \arg\min_{\tilde{P}} S(\tilde{P}, d) \tag{2}$$

We logically view the tuples in all the leaf nodes as an ordered array with no-decreasing keys, and denote the key of the j-th tuple as $key[j]$. Then $P' = \{k'_1, k'_2, \dots k'_l\}$ can be easily computed by evenly dividing the tuples into l ranges:

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

Given the new range partition of keys P', we employ the traditional bulk load technique to quickly build the template upwards, from the bottom inner nodes to the root node. After that, we complete the template update by reorganizing the data tuples in the leaf nodes according to the range partition schema of the new template. Some optimization tricks can be made in template update, to guarantee the process is no longer than several milliseconds. We omit the discussions, due to space limitation.

4.3 Adaptive Key Space Partitioning

Based on the fact that the overhead of inserting each input tuple is nearly the same, we measure the key distribution and evenly range-partition the keys among the insertion servers based on their frequencies, to balance the workload across the insertion servers. To do it in an efficient manner, we evenly divide the key space $\mathcal{K}$ into many fine-grained intervals and range-partition those intervals to the insertion servers. In particular, we let each dispatcher server maintain the key frequency of each interval in recent several seconds. A centralized system process periodically calculates the global key frequencies of the intervals by combining the values from all the dispatchers. Based on the key frequencies of the intervals and the interval to insertion server assignment, the process is able to estimate the workload on a particular ingestion server, simply as the sum of key frequencies of all the intervals assigned to it. If the workload distribution is skewed, the process adjusts the interval to dispatcher assignment to guarantee a balanced workload.

5. QUERY EVALUATION

5.1 Query Decomposition

To efficiently obtain the set of data regions covered by a query, the metadata server uses R-tree to store the data regions. When the system receives a query $q =< k_q, t_q >$, where k_q and t_q is the key interval and the time interval of the query respectively, the query coordinator refers to the R-tree in the metadata server and gets a set of regions R_q that are covered by the query q. For each data region $r_i =< k_i, t_i > \in R_q$ where k_i and t_i is the key interval and the time interval of the data region respectively, a subquery $q_i =< k_i \cap k_q, t_k \cap t_q >$ is generated and sent to the corresponding insertion server or one of the query servers for processing.

5.2 Caching

Reading data chunks from distributed file system is the dominant overhead of the subquery evaluation in the query servers. Consequently, we leverage caching techniques to reduce the subquery cost by keeping the frequently accessed data in the main memory of the query servers. In our implementation, a template or a leaf node is regarded as a basic caching unit. Compared with caching a data chunk in entirety, such fine-grained caching has better space-efficiency. Since the memory space is limited, we employ LRU policy to evict old caching units. In the rest of this section, we discuss our subquery dispatching optimization to maximize the effectiveness of caching by improving data locality.

5.3 Subquery Dispatching

We propose a simple but efficient method to guarantee the load balance and data locality at the same time. In our method, we create one queue for each insertion server in the query coordinator and hash-partition the subqueries to the queues. We take one subquery for each queue and send it to the corresponding query server for processing. When a query server finishes a subquery, we dispatch a new subquery from its queue if any. Otherwise, a subquery from the longest queues is taken instead. Such a policy achieves load balance by letting the query server finished earlier take over some work from the lagging ones. Also, data locality is retained, as the policy prefers to dispatch subqueries on the same data chunk to the same query server.

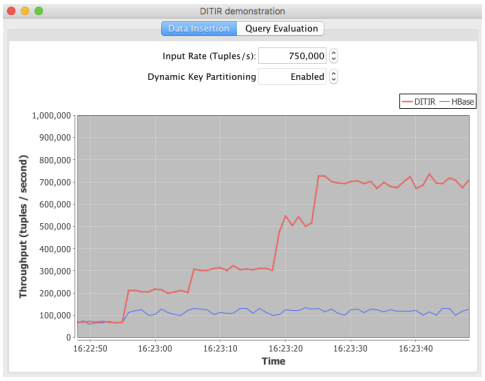

Figure 2: Insertion throughput display GUI.

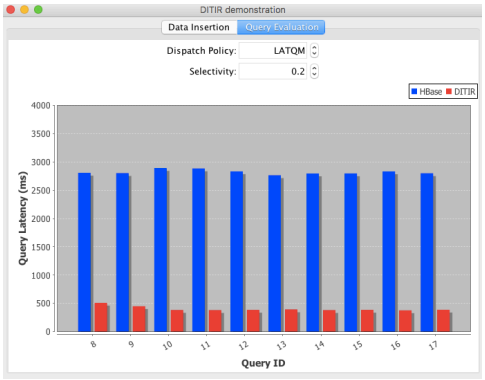

Figure 3: Query latency display GUI.

6. DEMONSTRATION OVERVIEW

In this demonstration, we run a mixed workload with trajectory data insertion and temporal spatial querying on DITIR and HBase simultaneously, and display the instantaneous performance metrics of both systems on a Java GUI. In particular, we implement our DITIR based on Apache Storm [3]. DITIR and HBase are executed on two 10-node clusters, respectively. For each system, we launch a background process to continuously insert trajectory data to the system and issue queries. The Java GUI program connects to the background processes, controls key parameters, and displays the performance metrics.

Insertion Throughput: On the insertion throughput window, as shown in Figure 2, the throughputs in the recent 60 seconds are displayed in the plot. Users are allowed to change the input data arrival rate and enable/disable the dynamic key partitioning feature for DITIR. Users will observe that the data insertion capacity of DITIR is 1180K tuples/s, outperforming HBase by 7 times. Also, a 20% throughput increase will see in DITIR when the dynamic key partitioning feature is enabled.

Query Latency: On another GUI window, as shown in Figure 3, the recent 10 query response times are displayed for both systems. Users are allowed to change two parameters, i.e., query selectivity and dispatching policy, to see how those parameters affect the query latencies. Generally speaking, the queries in DITIR is 5 times faster than HBase.

7. CONCLUSION

In this work, we present DITIR, our new distributed indexing framework to support highly parallelized trajectory insertion and efficient temporal range querying. DITIR outperforms state-of-the-art solutions by a substantial margin, in terms of both insertion throughput and query responsiveness. To the best of our knowledge, DITIR is the only applicable solution to real-time trajectory indexing over one million insertions per second.

8. ACKNOWLEDGMENTS

This study is partially supported by the research grant for the Human-Centered Cyber-physical Systems Programme at the Advanced Digital Sciences Center from Singapore's A*STAR and Infosys, NSFC-Guangdong Joint Found (U15-01254), Natural Science Foundation of Guangdong (2014A0-30306004, 2014A030308008), Science and Technology Planning Project of Guangdong (2015B010108006, 2015B01013-1015).

9. REFERENCES

[1] http://hbase.apache.org.
[2] https://cassandra.apache.org.
[3] http://storm.apache.org.
[4] https://www.influxdata.com/.
[5] https://www.mongodb.com/.
[6] D. Achakeev and B. Seeger. Efficient bulk updates on multiversion b-trees. *Proceedings of the VLDB Endowment*, 6(14):1834–1845, 2013.
[7] M. K. Aguilera, W. Golab, and M. A. Shah. A practical scalable distributed b-tree. *Proceedings of the VLDB Endowment*, 1(1):598–609, 2008.
[8] 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.
[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. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.
[10] P. Mazumdar, L. Wang, M. Winslet, Z. Zhang, and D. Jung. An index scheme for fast data stream to distributed append-only store. In *Proceedings of the 19th International Workshop on WebDB*. ACM, 2016.
[11] G. M. Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company New York, 1966.
[12] 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.
[13] R. Sears and R. Ramakrishnan. blsm: a general purpose log structured merge tree. In *SIGMOD*, pages 217–228. ACM, 2012.
[14] 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.
[15] 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.
[16] L. Wang, M. Zhou, Z. Zhang, Y. Yang, A. Zhou, and D. Bitton. Elastic pipelining in an in-memory database cluster. In *SIGMOD*, pages 1279–1294. ACM, 2016.