Internet of Things and Cloud Computing
2017; 5(3): 59-63
http://www.sciencepublishinggroup.com/j/iotcc
doi: 10.11648/j.iotcc.20170503.14
ISSN: 2376-7715 (Print); ISSN: 2376-7731 (Online)

TideDB - A Distributed, Scalable Time Series Database

Xue Yingfei

Research and Development Department, Tide Cloud Company, Shanghai, China

Email address:
xueyingfei@tidecloud.org

To cite this article:
Xue Yingfei. TideDB - A Distributed, Scalable Time Series Database. *Internet of Things and Cloud Computing*. Vol. 5, No. 3, 2017, pp. 59-63.
doi: 10.11648/j.iotcc.20170503.14

Received: May 15, 2017; **Accepted**: July 15, 2017; **Published**: August 7, 2017

Abstract: Some of the largest datasets have strong time components, like machine monitoring, real-time alert and IoT devices, etc. Despite of so many applications of time series data, most storage options are either highly proprietary or worse, relational. Unlike other alternatives, TideDB does not have a data with multiple metrics broken down into multiple data with one metric that increases the pressure on system throughput dramatically, rather its data modeling based on the computed column and tag words index can provide high write throughput, low read latency, and petabytes storage. TideDB has been deployed in production settings on large clusters to manage multiple terabytes of storage at Taide Company. The paper describes the TideDB how to store and organize our time series data from about one hundred thousand devices and millions service modules.

Keywords: Distributed, Scalability, Time Series, Internet of Things, Metric, Performance

1. Introduction

There are plenty of storage engines out there, but none of them seem to offer fast and efficient time series storage and indexing. The existing options like MySQL aren't very fast, and the fast options like HBase aren't specifically made for time series and can lead to harsh operational issues. Before TideDB, we choose MySQL as our backend engine and do lots of maintenance work to improve the performance of MySQL. As the amount of data grows, the scalability of MySQL is very bad. Then the system used HBase as its backend engine, which looked better than MySQL, but still doesn't meet our requirements well. Instead of hacking on something like HBase to suit our needs, we decided to write our own time series storage engine. About nearly one year design and implementation TideDB can be used to manage the time series data from machine and application service for the TideCloud that monitors about one hundred thousand devices at Taide Company. These machines will produce a series data by the cycle of 10 seconds and inserted into TideDB cluster by 2 minutes, and these data will be accessed in various business applications that need low read latency. Every day there are 50 TB raw data to be inserted into TideDB, and lots of historical data query requests with 99 ile% latency < 200 ms. In Argus, TideDB shares many implementation strategies with Google's Bigtable, but TideDB provides a different data model and query optimizer than such systems and introduces the computed column into the data row. When a time series data is stored in a computed column, the data is stored contiguously and is retrieved with a minimum number of disk reads. It also provides some basic data operation for each component of Argus and some tools for the Argus administrator, such as query with where sub-clause, sharding/rebalance/check table, and support some basic aggregation computing.

Section 2 provides an overview of the TideDB. Section 3 describes the key features and the fundamentals of the TideDB implementation, and Section 4 presents some works on the performance and provides measurements of TideDB's performance. Based on that, Section 5 describes related work, and Section 6 ends up with our conclusions.

2. Architectural Overview

TideDB consists of four components: Metadata server, Ranger server, Broker server and Distributed File System (DFS), and stores data in a table which is sorted by a primary key that contains instance IP (or the name of service monitored by Argus) and timestamp. One instance's sequence of data points stored in a 4 M file, and several instances' files in logic form one data range. Metadata server provides the catalog information for all the tables in this system and builds the B-tree index structure for the range key of each table. Each Ranger server holds some ranges of the data, but all of the

range on one Ranger server is not necessarily continuous that make the range service more flexible because each data range is not dependent upon another data range. There are millions of instances or services time series data to be inserted by one cycle. In order to relieve the pressure on Ranger server this design introduce the Broker to do some of requests preprocessing received such as building one bulk of multiple inserts and splitting the query clause into multiple sub query plans. The Broker server deployed on a Ranger machine is a light API service, which supports the REST API to be called by applications and enables applications to access data stored in TideDB.

The incoming data to be inserted, which have a natural temporal ordering, is consistent with the TideDB data model design, so that the data will not be moved once written to memory file mapping disk one, and the query optimizer in Broker can make the query plan only hit one or two disk IO by mapping the query into multiple sub query plans. The data store only needs a normal file storage engine, which provides a uniform file namespace for Meta server and Ranger server, to do the persistence storage for TideDB. So this design chooses a distributed file system from the third party product that used to store the data with 3 replications that ensure the integrity of data.

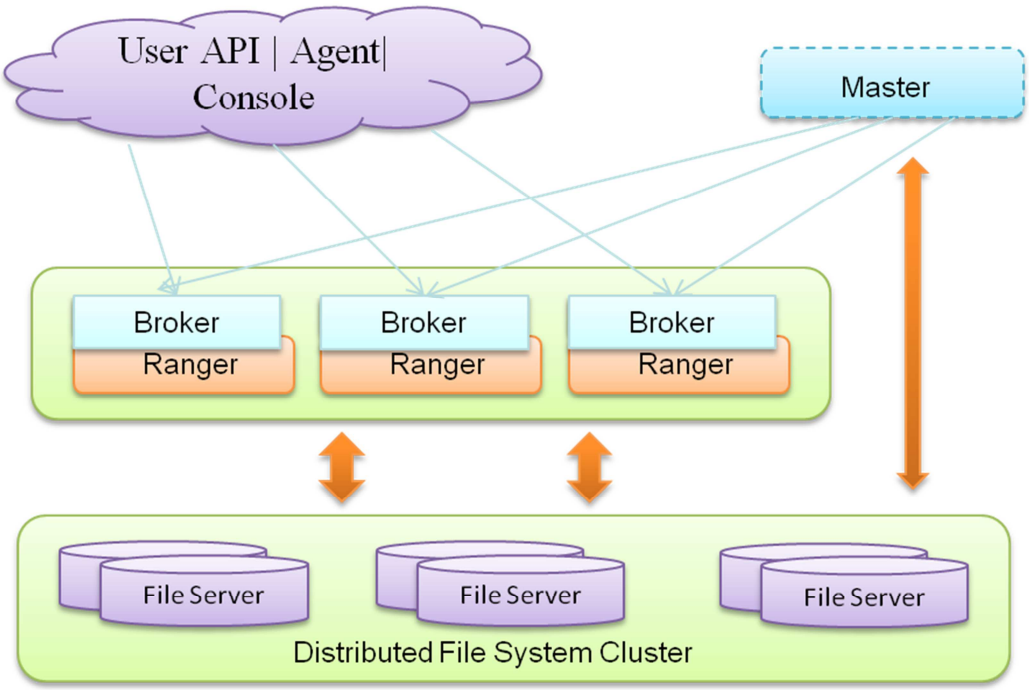

Figure 1. *What are the components in the system and how they relate to one another.*

Scaling is achieved by splitting tables into contiguous data ranges and assigning them up to different physical nodes. Meta Server which handles metadata management works and oversees the Ranger Servers is responsible for farming them out in an intelligent way and reassigns some ranges of the data locating at the overload Ranger Servers to other Range Servers that have enough space. New ranger nodes can be added as storage needs increase and the system automatically adapts to the new nodes. The Meta server receives the request of heartbeat from the Ranger server newly added, finds the Ranger server who own the most tablets and transfers some tablets to the new Ranger server to balance the service performance, and finally it adds the new Ranger server into the service list.

3. Features and Implementation

3.1. Data Row Format

There are about 10 TB raw data every day from millions of monitoring objects. If the system does not do any processing to store these raw data, it will store a large amount of redundant data and impact the write throughput and reading performance. In order to enhance storage and access efficiency TideDB's row format follows the design idea from the traditional relational database, that is one data row, which includes three parts, namely row header, field data and field offset. This design can apply query and field location. Header information includes the row state and row number that is the row order in the block, filed offset drives the field data. This data model will serves better than the JSON and BSON in certain way, such as space saving, field locating and compression, etc. When users want to find the value for the specified field, TideDB can directly depend on the field's offset to locate the value of this field without having to traverse all the field values.

3.2. Time Series Data Modeling

A time series is a sequence of data points, typically consists of successive measurements made over a time interval. There are about 100 thousand devices at Taide, and each node has

about 200 metrics including CPU used, memory used, QPS and UPS, etc. and it born the data by the cycle of 10 seconds. The data has the characteristics of time series and will be used to check if these machines have a good running state. For example, an operate engineer wants to know someone machine's disk writing status in the past 10 hours, he can query the metric of DISK_TOTAL_WRITE_KB by the specified time range, and the web will show the chart by the query data result from the DB.

The scenario mentioned above has two challenges: large writes throughput and low query latency. In order to solve this problem, our row format as follows:

| Machine_timestamp_metric | Cycle | Timestamp | Tag Words |Computed Column |

This row can store one hour data and the row key is built by the machine, the start point of one hour and metric. The column of Timestamp is the end point of one hour, the cycle is the interval of the new data born and the tag words flag this machine characters. Computed column contains two parts including computed header and data slot that records the actual data value from the machine. Computed header is comprised of 16 bytes that encode a slot map managing the data slot. The computed column has about 360 (1 hour dived by the cycle of 10 seconds) slots that store the arrange of data points. The first insert coming will build the new data row but the following insert only needs to do the in-place update of slot that will solve the large writes throughput. The query model use machine, time range and metric that the user specified to build the row key and query the data by the range index.

3.3. Query Optimizer

There are plenty of query cases at Taide based on the tag words condition including IDC, cluster and service name. It will hit lots of data rows if the system only use the time condition to locate the interesting data with specified tags. So this design introduce the tag words index into TideDB and avoid the redundant data scanning. TideDB tag words index have an inverted index design. Inverted indexes store a list of data rows that each tag word appears in. The index row only needs to be built and inserted as the data row builds. That is to say, one index row only inserts one time in one hour so that it will avoid write performance degradation raised by the index insert.

TideDB provides the basic data query functions such as select and update, support the range query and aggregation computing by any fields and the where condition sub-clause. While the range query is coming, it will split the query work over all the range nodes in the cluster, find the data row id by the index if there's one tag words index on the specified column, reduce their query result and send them to the client. In this machine monitor scenarios, TideDB's Broker maintains an optimizer that can normalize and optimize the user's query plan to make it suitable for TideDB's data storage model. For example the incoming query with the time range of 10 hours user specified and the metric of CPU_USED, but one TideDB's data row only contains the data of 1 hour and 1 metric, if the system directly use the clause user specified, the TideDB server will consume much time on the data scanning. So the Broker's optimizer splits the incoming clause into 10 sub clauses and maps 10 threads to do the sub-query separately, so as to avoid data scanning, reduce their query result and send them to the client.

3.4. Failure Recovery

TideDB must be devoted to maintaining integrity in the face of failure. Failure here can mean a user-level or fatal error in an individual write session, or a partial system error, or a total system error such as a power failure or software panic. Meta server maintains a heartbeat with each Ranger server. If the Meta server detects that the range server is unreachable, it will fail-over the data service locating on the crash Ranger server to another online Ranger server and continue the service for this range. Integrity here is limited to consistency. There are some types of log for data operation in TideDB. Data log can log the data insert and update, index log is to log the insert and update on table with index, split log is to log the block split, SStable split and tablet split. While one data insertion is coming, the data insertion log must be written to the disk and logging the data SStable split if it hit the SStable split case. In this machine monitor scenarios, TideDB's data row is sorted by the machine and timestamp and the data is produced by the growth of timestamp so that the new data to be inserted only need to append the SStable file and the SStable split is tail split except the rare case that need to do the middle split. Tail split doesn't need to log split case and avoid affecting the write performance by splitting.

3.5. Administrator Tool

TideDB provides some administrator tools to manage the metadata information in TideDB, such as TideDB Consistency Checking (TCC) that ensures the data consistency between on the Meta server and Ranger server, Rebalance Command that manually rebalance the data range amongst Ranger servers to help balancing the workload amongst nodes. For example, TideDB's administrator can transfer some data range from the 'hot' nodes to the 'code' nodes so that the responses to the client become more faster.

4. Performance Evaluation

Construct a performance evaluation 10 k+ bytes records by using 5 fields including 2 Varchar, 2 Int and 1 computed column. The data and do some data operation (insert/select) against the TideDB is driving the performance evaluation of the TideDB prototype. Our experiments show that TideDB's performance can meet the requirements of low read latency and high write throughput. Section 5.1 describes our experimental setup and section 5.2 present more detailed experimental results.

4.1. Experimental Setup

Each experiment uses two machines as TideDB server and

one client machine. Each machine has six quad-core Intel Xeon E 5-2620 processors, 64 GB RAM, a 1 Gbps network interface card, and runs RedHat Linux. On both two machines, this test start one Meta server to manage the meta information and one ranger server to process the user's data, and run the client driver cases on another machine.

4.2. Experimental Detail

This experiment uses Latency / Throughput as the metrics for our evaluation of insert operations and Average and 99 ile% Latency / Time Range to evaluate the performance of range query. There will be a table to insert using the data from 6,000 machines by the cycle of 10 seconds and query the series of data points with the specified time range. Finally, the experiment has the concurrent query evolution on the table to test the performance stability of the TideDB, in this experiment, we can see TideDB's query performance is very stable. It does not hit the bumps on the query latency due to the concurrent query increasing.

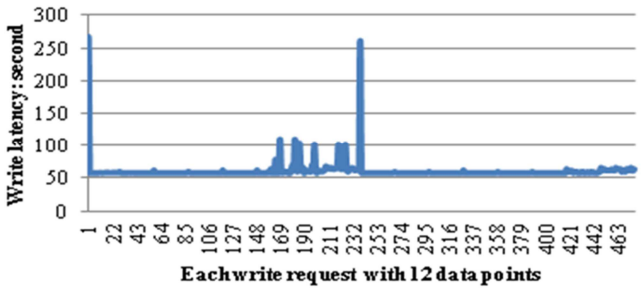

Figure 2. *Write as the data rows increase.*

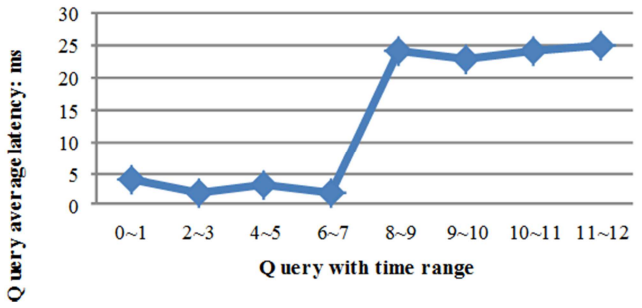

Figure 3. *Query with the 1 hours of time range.*

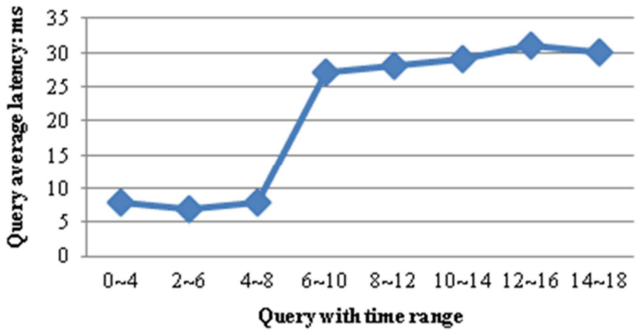

Figure 4. *Query with the 4 hours of time range.*

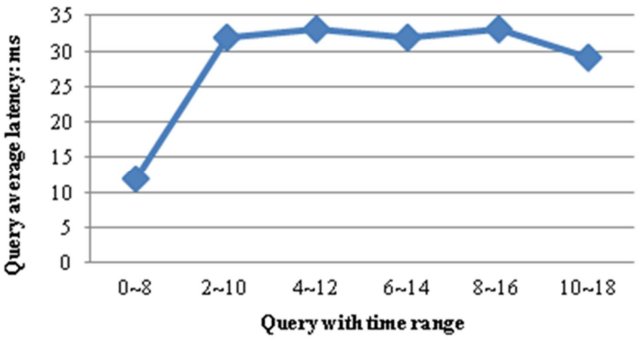

Figure 5. *Query by the 8 hours of time range.*

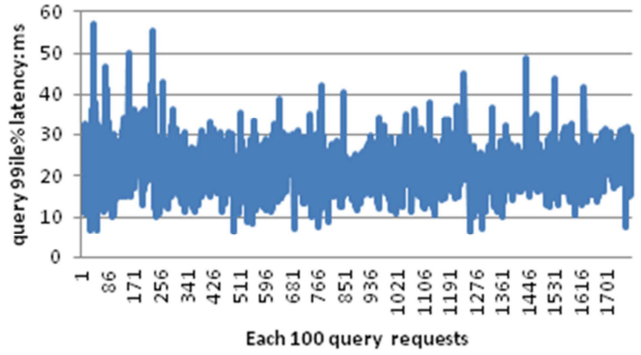

Figure 6. *Concurrent queries number: 4000.*

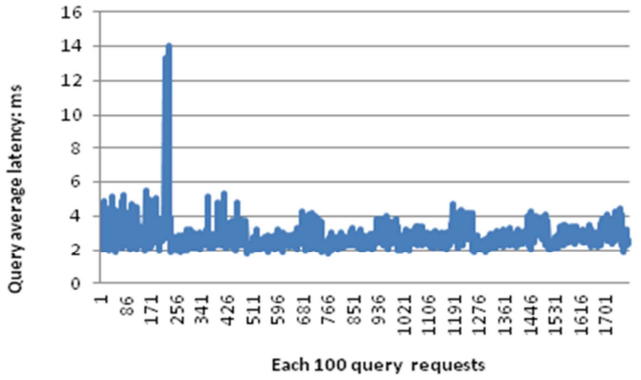

Figure 7. *Concurrent queries number: 4000.*

5. Related Work

There has been various study on the traditional database and the massive scale storage management system, including research on the dynamo at Amazon, Azure at Microsoft and bigtable/GFS at Google, and some popular Nosql products liking documented-based MongoDB, Cassandra and column-based HBase, etc. Our paper focuses on the time series storage, whose data model is not the format of JSON or BSON, but what adapts the innovative computed column and the row format from the traditional database. Its service is based on the data range, which demands more sophisticated data consistency and index techniques. Since data range may be split by the new insertion or manually rebalanced by administrator, the information (meta information, index, splitting work etc) have to be updated.

TideDB provides the data service more access-efficient

than other documented Nosql since the data service is split into different ranges that any two nodes don't have the override data range, while other documented Nosql products following the round-robin way that will leads to two nodes may hit same data and the redundant disk I/O.

6. Conclusions

In this paper, we have implemented one time series storage built on the distributed file system, named TideDB that supports large write throughput and low query latency using fewer machines than other systems like HBase. Our TideDB offers several novel features that make it especially attractive for large-scale storage systems for time series data. First, TideDB is designed to support effective data model which can enhance storage and access efficiency for time series scenario. Second, failure recovery mechanism can ensure the data consistency. Third, TideDB provides some administrator tools to manage the metadata information. Finally, by the tag words index mechanism and the range service, our TideDB provides better performance than other same type of products. Our experimental results over a wide range of data workloads have clearly demonstrated the benefits of TideDB, showing that our TideDB has rich functions, high reliability and high performance.

References

[1] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data.

[2] TideCloud, http://tidecloud.org/

[3] InfluxDB, https://www.influxdata.com/

[4] OpenTSDB, http://opentsdb.net/

[5] MongoDB, http://www.mongodb.org/

[6] Apache Cassandra, http://cassandra.apache.org/.

[7] Avinash Lakshman, Prashant Malik. Cassandra - A Decentralized Structured Storage System.

[8] Apache HBase, http://hbase.apache.org/.

[9] Rick Cattell. Scalable SQL and NoSQL Data Stores.

[10] James Cipar, Greg Ganger, Kimberly Keeton, Charles B. Morrey III, Craig A. N. Soules, Alistair Veitch. LazyBase: Trading Freshness for Performance in a Scalable Database.

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

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store.

[13] Sybase, http://www.sybase.com/products/databasemanagement/adaptiveserverenterprise.

[14] Rick Cattell. High Performance Scalable Data Stores.

[15] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System.

[16] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, Leonidas Rigas. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency.

[17] Daniel J. Abadi Samuel R. Madden Nabil Hachem. Column-Stores vs. Row-Stores: How Different Are They Really?

[18] Voldemort, http://project-voldemort.com/design.php

[19] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, Russell Sears. Benchmarking Cloud Serving Systems with YCSB.