Data model for sequential database

Sequential data is a series of data generated over time, which is simply time-stamped data. A sequential database is a database optimized for ingesting, processing, and storing timestamp data. Such data may include metrics from servers and applications, readings from sensors in the Internet of Things, user interactions on websites or applications, or transaction activity in financial markets.

In short, the sequential database is a database specially designed to store and process time series data, supporting efficient read and write of sequential data, highly compressed storage, interpolation and aggregation.The data types I know are as follows:

1、Druid

Druid is a different gameplay from KV databases like HBase and Kudu. Druid is a fully columnar storage system with no HBase primary key.Druid is a column database, so each column is stored independently, for example Timestamp columns are stored together as a file, publish columns are stored together as a file, and so on. Careful children will say that such storage, there will still be a large number of data source redundancy problem. To solve the redundancy problem, Druid and HBase use the same encoding dictionary to encode label values. Label values of the string type are encoded as int values. However, unlike HBase, Druid encoding is local encoding. Both Druid and HBase use LSM structure. Data is written to memory first and then flushed to data files.

In addition, Druid's columnar storage has the following benefits:

1. The data storage compression ratio is high. Each column is stored independently. You can compress each column and set a compression strategy for each column. For example, time column, INT, FLOAD, double, and String can be compressed separately to achieve better compression effect.

2. Support multi-dimensional search. Druid creates a Bitmap index for each of the datasource columns. The Bitmap index can be used to query the views of all the ads published in the USA at 20110101T00:00:00. You can find the row number in the Bitmap index based on country=USA, and then locate the metrics to be checked based on the row number.

However, there are some problems with such a storage model:

Problem 1: Data is still redundant. Like OpenTSDB, tags have a lot of redundancy.

Problem two: Scoping a specified data source is not as efficient as OpenTSDB. This is because Druid breaks the data source into multiple tags, each indexed by Bitmap, and then uses and operations to find the line number that meets the criteria. This process requires some overhead. On the other hand, OpenTSDB can create rowkeys based on data sources to search for B+ tree indexes, which is more efficient.

2、InfluxDB

Compared to OpenTSDB and Druid, many children may not be particularly familiar with InfluxDB, yet InfluxDB is far ahead on the leaderboard of the sequential database. InfluxDB is a professional temporal database that only stores temporal data, so a lot of optimization can be done for temporal data in the storage of the data model.

To ensure efficient writing, InfluxDB also uses the LSM structure. Data is written to memory first and flushed to files when the memory capacity reaches a certain threshold. InfluxDB proposes a very important concept in the design of sequential data models: seriesKey. SeriesKey is actually a datasource +metric. Sequential data is organized according to seriesKey after being written to memory.The memory is actually a Map, in which a SeriesKey corresponds to a List, which stores timeline data. After data in according to the datasource (tags) + metric Mosaic SeriesKey, then Timestamp | Value portfolio Value write time line data List. Flush indicates that the timeline data in the same SeriesKey is written to the same Block, that is, all the data in a Block belong to the same metric of the same data source.

This design we think is to pick out the time series data according to the time line. Let's look at the benefits of this design:

Benefit 1: Tags from the same data source are no longer stored redundantly. All data within a Block shares a SeriesKey, which can be written to the Trailer part of the Block. The storage of time series data is greatly reduced.

Benefit 2: Time series and value can be stored separately in the same Block. In this way, time column and value column can be compressed separately. InfluxDB stores time columns in accordance with Beringei's compression mode. The delta-delta compression mode greatly improves compression efficiency. The compression of Value can adopt the same compression efficiency for different data types.

Benefit 3: For a given data source and time range of data search, can be very efficient. This is the same with OpenTSDB.

1. Beringei

Beringei is a timing database system that Facebook opened source this year. The design of time series data model can select time series according to data source and metric well, which solves the problem of redundant storage of dimension column values and ineffective compression of time column. However, the problem of write cache compression is not well addressed by InfluxDB: InfluxDB does not compress when writing to memory, but compresses data when writing to files. We know that one of the biggest characteristics of sequential data is that the most recently written data is the hottest. Storing all the recently written data in memory can greatly improve the reading efficiency. Beringei is a good solution to this problem. Streaming compression means that data is compressed after being written to memory. This allows more sequential data to be cached in memory, which is helpful for querying recent data.

Beringei's design of timing data model is basically consistent with that of InfluxDB, and also puts forward a concept similar to SeriesKey to separate time lines. However, there are two major differences between InfluxDB and InfluxDB:

1. Different file organization forms. Beringei's file storage form is organized according to time window. For example, the data of the last 5 minutes are all written into the same file, which is divided into many blocks, and all the sequential data in each block share a SeriesKey. The Beringei file has no index, InfluxDB does.

2. Beringei currently has no inverted index mechanism, so it is not efficient for multidimensional queries.

Beringei data writing, streaming compression and file formats will also be introduced. In the author's opinion, if Beringei and InfluxDB are effectively combined, timing data can be efficiently stored in memory. In addition, data is organized according to dimensions, which can effectively improve the storage efficiency and query efficiency of data in files. Finally, multi-dimensional query ability can be effectively improved by combining the InfluxDB's reverse index function.

This paper is the first paper on the technical system of sequential database. The author mainly introduces the storage model of sequential data in the form of data by combining OpenTSDB, Druid, InfluxDB and Beringei respectively. Each database has its own storage mode, and each storage mode has its own advantages and disadvantages. It is these advantages and disadvantages that directly determine the compression performance and read and write performance of corresponding sequential database.

1. OpenTSDB(HBase)

OpenTSDB stores time series data based on HBase, and designs RowKey rules at the HBase level as follows: Metric +timestamp+datasource(Tags). HBase is a KV database. If a time series data (point) is represented in the form of KV, V must be the specific value of point, and K is naturally the datasource+metric+timestamp that uniquely determines the value of point. This rule applies not only to HBase but also to other KV databases such as Kudu.

Since K in HBase consists of datasource, metric, and TIMESTAMP, rowkey can be simply considered as the combination of the three. Then, the question arises: What is the combination order of the three?

Let's start with which one should come first. Data in a table in HBase is organized in alphabetical order based on rowkeys. To collect all data of the same indicator, HBase puts the metric before the rowkey. If timestamp is put in the first place, the data at the same time will be written into the same data fragment, unable to play the effect of hash; However, if the datasource (tags) is put first, there is a bigger problem. The datasource itself consists of multiple tags. If you specify some of the tags to be searched without prefix tags, a large range of scanning and filtering queries will be performed in HBase, resulting in low query efficiency. For example, if we put datasource first, Rowkey = publisher=ultrarimfast.com&advertiser:google.com&gender:Male&country:USA\_impressions\_20110101000000, At this point, the user wants to search the page views of all advertisements published in USA at 20110101000000, that is, to search an index at a specified time point only according to the information of one dimension, country=USA, which is not a prefix dimension, a large number of records will be scanned for filtering.

Now that the metric is first, should we put the datasource in the middle or timestamp in the middle? Putting metric first already addresses the requirement for uniform distribution (hashing) of requests, so HBase puts timestamp in the middle and datasource in the end. Imagine putting the datasource in the middle and running into the same postfix dimension lookup problem described above.

Therefore, the design of the rowkey in OpenTSDB is: metric + timestamp + datasource, so HBase can set only one columnfamily and one column. So what's wrong with OpenTSDB's design? Before understanding the design issues, take a quick look at the way HBase stores KV in files. That is, the way a series of sequential data is stored in files and memory.