TRustDB — A Time Series Database for High-Cardinality, Multivariate Data

Desmond Cheong
Brown University
desmond_cheong@brown.edu

Nick Young
Brown University
nicholas_young@brown.edu

ABSTRACT

The adoption of microservices-based distributed systems has led to an increase in the collection and usage of telemetry data such as metrics. This data helps operators understand the state of complex systems, and is often stored in time series databases that support fast ingestion and querying of high-volume and high-sample-rate data. However, data produced by monitoring workloads for microservices additionally tend to have high-cardinality. For example, this happens when rolling updates of microservices are performed, causing a phenomenon known as series churn, where a set of time series become inactive and are replaced by a new set of time series. Or, when endpoint IP addresses change frequently, a different time series is created for each new address.

High-cardinality data poses its own set of challenges in terms of management and usage. In this paper, we address two of these: minimizing the overheads of storing high-cardinality metadata, and efficiently querying large numbers of time series for specific data points. We present TRustDB, a time series database that tackles these issues by (1) compressing metadata with finite-state transducers (FSTs), and (2) optimizing queries by rewriting and evaluating them in disjunctive normal form (DNF).

Our code is available on GitHub at:
https://github.com/n-young/trustdb

1. INTRODUCTION

Time series databases are widely used in storing telemetry data such as metrics and are a key component of monitoring systems. Work on time series databases typically revolves around achieving high ingest rates, low storage footprints, and fast query times. A primary issue that many time series workloads face is high-cardinality data — that is, data with a wide variety of values and tags, resulting in quick growth in the number of series being tracked, a phenomenon known as churn. Churn can lead to a growth in the size of metadata and an increase in query processing time; thus, solving the high-cardinality problem is key to meeting the primary objectives of a time series database.

A primary consequence of high-cardinality data is an increase in the number of time series relative to the number of data points being ingested. For each new time series, the system must track more metadata related to the time series, including series labels, metric names, timestamp ranges, *et cetera*. Higher cardinality leads to a lower ratio between metadata and the data itself; thus, as metadata grows to a non-negligible proportion of necessary data, so does the need to optimize how we process and store metadata. Moreover, as the number of time series increases, queries for time series data become much less straightforward to evaluate, as data is likely to be split across different series. More emphasis is placed on deciding which series to evaluate, rather than on the values of the data points themselves.

In the following sections, we explain our data model and system architecture, then our two primary contributions: compressing metadata storage by applying finite-state transducers (FSTs) and optimizing queries by converting to disjunctive normal form (DNF).

2. TRUSTDB

2.1 Data Model

We introduce TRustDB, a time series database for high-cardinality, multivariate data. TRustDB supports reads and writes of multivariate time series data in JSON format.

Data points consists of a name, a set of label-value pairs, a set of metric-value pairs, and a timestamp. Data can arrive out of order between series, but not within a given series. An example data point is shown below (shortened):

```
"name": "cpu",
"labels": {
    "hostname": "host_0",
    "region": "us-west-1"
},
"variables": {
    "usage_user": 58.0,
    "usage_system": 2.0
},
"timestamp": "2016-01-01 00:00:00+00:00"
```

Queries consist of equality or comparison statements, called

conditions, over label-key and label-value pairs, metric-name and metric-value pairs, or timestamps. Conditions are combined using 'And' or 'Or' nodes in a binary tree-like structure. An example query is shown below (shortened):

```
"condition": {
    "And": [
        {
            "Leaf": {
                "lhs": { "LabelKey": "Key" },
                "rhs": { "LabelValue": "Value" },
                "op": "Eq"
            }
        },
        {
            "Leaf": {
                "lhs": { "Variable": "Var" },
                "rhs": { "Metric": 6.0 },
                "op": "Gt"
            }
        }
    ]
}
```

2.2 System Architecture

TRustDB follows a rather conventional in-memory model with a simple scheme for flushing data to disk. Data is stored in Series, which consist of a set of label key-value pairs, a vector of metric names, and a vector of SeriesRecords, each containing just a vector of metric values and a timestamp. Series are stored in Blocks, of which at most one is maintained in memory at any given point in time. A Block consists of a vector of Series, start and end timestamps designating the range of the Block, a HashMap denoting the location of a Series in storage by its id, and an inverted index over the label key-value pairs and metric names. The inverted index is initially maintained as a HashMap but is later compressed to an FST, and each entry in the index points to a Bitmap indicating which Series are relevant.

When the current Block is considered full, it is frozen and flushed to disk while another Block is allocated to continue ingesting data. Additionally, a BTreeMap, used for performing range queries over data on disk, is maintained over all of the Blocks in-memory and on-disk, indexed on block start- and end-times. Blocks are stored in standalone files. To avoid reading excessive memory from disk, when a Block's relevance is being considered for a query, it is maintained as a PackedBlock, which contains all of the metadata relating to the Block's contents, but not the data itself.

All of the relevant structs are detailed in the appendix.

3. INVERTED INDEX COMPRESSION

Supporting data structures are needed to provide fast queries over time series. One common choice is an inverted index that maps from label key-value pairs and metric names to a set of relevant time series. However, high-cardinality data has two characteristics that make it challenging to apply the same techniques: (1) each series tends to have a greater amount of associated metadata which helps identify different series; (2) series tend to have fewer data points, meaning that a significant proportion of space is used by metadata.

These issues lead to the conclusion that special attention should be paid to metadata compression. Prior work done by ByteSeries [1] suggests that tries are a good option to compress inverted indices, because series often have overlapping label keys, and various label values (such as IP addresses) also have overlapping patterns. In comparison, M3DB[1] uses FSTs to store index information. We believe that FSTs should produce greater savings compared to tries: branches in a trie that have overlapping strings can be merged into the same path on a Directed Acyclic Graph, as in an FST.

As such, in TRustDB, when flushing an in-memory block to disk, we take the inverted index that was previously maintained as a hash map and store it as an FST.

4. QUERY OPTIMIZATION

TRustDB supports conditions over labels and over metrics; the former acts as a filter over entire series, while the latter acts as a filter over individual data points across series. Critically, we can apply label conditions using only series metadata, without needing to look at any data points. Moreover, notice that once we apply a metric condition and receive a vector of data points across series, we lose the ability to easily filter by label conditions. Thus, it is much more favorable to cull the search space using metadata before looking at the data itself. This leads to our first implication: that we should evaluate label conditions early, and metric conditions late.

To do so, we delay evaluation of metric conditions, a process we call unpacking, by saving the metric condition as a lambda function to be evaluated over all relevant time series when it is necessary. So now, the question becomes: when is it necessary to unpack? This is best illustrated through an example; consider the following condition:

$$x' \vee (y' \wedge y) \tag{1}$$

where x', y' are metric conditions and y is a label condition. We evaluate bottom-up, first evaluating $y' \wedge y$ — to do so, we save y' as a lambda function and apply y immediately. This returns us a set of relevant series and a filter to apply later. Next, we apply $x' \vee (y' \wedge y)$. We may be tempted to save x' as a lambda function, but x' is not bounded to apply to only series that fit y. Thus, there is no way to combine these two conditions into one result set under our scheme; it is in this case, encountering an 'Or' clause, that we are forced to unpack. So, to evaluate, we unpack the right subtree, unpack the left subtree, and then intersect our results. This leads to our second implication: that we should process 'And' clauses before 'Or' clauses.

However, because we evaluate clauses bottom-up, this is not always possible; thus, some query optimization is warranted to ensure that we always process 'And's before 'Or's. To achieve this, we convert our queries to disjunctive normal form (DNF), a structured form of boolean expression which consists of one large 'Or' clause over many 'And' clauses. To convert a query to DNF, we repeatedly distribute our con-

[1]https://m3db.io/

junctions over our disjunctions, pushing our 'And' clauses further down the tree. It's worth nothing that while this transformation does lead to potentially exponential growth in the size of the query, experiments show that it is a worthwhile trade-off in the face of unpacking. Moreover, other potential transformations, such as by applying De Morgan's Laws, may necessitate the use of 'Not' clauses, creating potentially massive sub-evaluations.

These implications lead to the following query processing scheme. First, convert our query to DNF. Queries are then evaluated bottom-up according to the tree structure defined by the data model, passing ResultSets up along the tree. On each level of evaluation, a ResultSet is either packed or unpacked; in its packed form, the ResultSet contains no Records - only a set of relevant Series and a set a lambda functions that filter out the relevant data points in each Series. In its unpacked form, the ResultSet contains the raw vector of Records. At the root of the condition, or when necessary, the ResultSet is unpacked and returned to the caller.

5. EXPERIMENTATION AND EVALUATION

To evaluate our optimizations, we used a slightly modified version of TSBS to generate a high-cardinality workload, with a log interval of 1s spanning an hour of data from multiple sources. The workload we evaluate on has 32,400 rows.

5.1 Evaluating Storage Optimizations

Passing the above workload as input into TRustDB, we track the amount of storage space needed to store our inverted index on disk in two different forms — (a) directly as a hash map, and (b) converted into an FST.

Pts/Blk	Churn Rate	Hash Map	FST	Size Reduction
10,000	0.001	1.35MB	1.33MB	1.4%
10,000	0.01	1.47MB	1.44MB	1.7%
10,000	0.1	2.10MB	1.81MB	13.8%
30,000	0.001	1.16MB	1.14MB	1.4%
30,000	0.01	1.27MB	1.22MB	3.8%
30,000	0.1	1.88MB	1.59MB	15.3%

Figure 1: Here, Pts/Blk refers to the number of data points whose meta we store together, Churn Rate refers to the probability that a time series stops emitting values. The sizes under "Hash Map" and "FST" refer to the storage space needed for the metadata with each storage method, and size reduction refers to the percentage space savings when going from hash maps to FSTs.

From Figure 1, we notice that as the data points stored per block increases, the space savings from switching from a hash map to an FST increases. A similar pattern occurs as churn rate increases. These trends make sense because the number of series metadata stored in the FST is proportional to the data points per block, and how often new series are created.

As an additional experiment, we generated another workload with 300,000 rows to investigate the savings with more data points per block. The results in Figure 2 show us the same trend from Figure 1. Comparing the results, we see that there is a sub-linear increase in the percentage of space saved as compared to the increase in the number of series' meta data being stored in the FST.

Although the space savings from using FSTs are promising (achieving up to an 18% savings on storage space in our largest workload of 300,000 input rows), before determining whether they are the right data structure to use, further experiments need to be carried out to evaluate the computational costs of converting the inverted index into an FST, and to understand how the use of an FST might impact the speed of query evaluation. We plan to take up these experiments in future work.

Pts/Blk	Churn Rate	Hash Map	FST	Size Reduction
100,000	0.001	8.88MB	8.71MB	2.0%
100,000	0.01	10.0MB	9.52MB	4.9%
100,000	0.1	15.9MB	13.0MB	18.3%

Figure 2: Space savings when we further increase the number of points per block.

5.2 Evaluating Query Optimizations

We test locally on a generated workload on three variants of TRustDB — (a) one with DNF conversion and deferred metric evaluation, (b) one with only deferred metric evaluation, (c) and one with neither optimization.

For our first run, we used two separate query generators to generate series-targeted queries in the same form as (1). We test each run on 20 queries of this form. Time is measured using the UNIX `time` command on a script that runs the given workload.

run 1	a	b	c
ingest	1m18.622s	1m30.694s	1m14.290s
queries	0m40.056s	4m52.767s	5m0.669s

Figure 3: Above are the evaluation times for 20 queries in minimal form (1) on 32,400 data rows.

For our second run, we use the same workload alongside 10 queries generated in conjunctive normal form (CNF), which is the opposite of DNF: a series of 'And's over units of 'Or's. Concretely, we evaluate on queries of the form:

$$(a' \vee a) \wedge (b' \vee b) \wedge (c' \vee c) \wedge (d' \vee d) \tag{2}$$

It's worth noting that queries of this form are the worst case scenario for converting to DNF, as the resulting query will have 2^n clauses. Our results are shown in Figure 4.

There is a clear increase in query evaluation speed when we apply deferred metric evaluation and DNF conversion both on the minimal case and on the worst case. This suggests promise in our evaluation scheme, and we hope to do further investigation and evaluation to see which workloads DNF might perform better or worse on.

run 2	a	b	c
ingest	1m23.338s	1m37.556s	1m33.024s
queries	0m1.202s	5m8.373s	4m56.741s

Figure 4: Above are the evaluation times for 10 queries in CNF (2) on 32,400 data rows. We get massive time speed up by converting to DNF (over 250x).

6. CONCLUSION

In this paper, we introduce TRustDB, a database which is optimized for incredibly high-cardinality data reads and writes. By introducing FSTs in the storage layer and using DNFs to optimize query evaluations, we produce significant space and time improvements when handling high-cardinality time series data, with our gains improving as cardinality and query complexity increase. We have good reason to suspect that similar optimizations could be applied to other time series databases to increase storage- and query-layer efficiency.

7. REFERENCES

[1] X. Shi, Z. Feng, K. Li, Y. Zhou, H. Jin, Y. Jiang, B. He, Z. Ling, and X. Li. Byteseries: an in-memory time series database for large-scale monitoring systems. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 60–73, 2020.

APPENDIX

Structs

```
pub struct BlockIndex {
    index: BTreeMap<i64, Vec<String>>,
}


pub struct Block {
    index: HashMap<String, Bitmap>,
    storage: Vec<Series>,
    id_map: Vec<String>,
    key_map: HashMap<String, usize>,
    start_timestamp: Option<DateTime<Utc>>,
    end_timestamp: Option<DateTime<Utc>>,
    frozen: bool,
    compressed_index: Option<Map<Vec<u8>>>,
    compressed_bitmaps: Vec<Bitmap>,
}


pub struct PackedBlock {
    start_timestamp: Option<DateTime<Utc>>,
    end_timestamp: Option<DateTime<Utc>>,
    index: HashMap<String, Bitmap>,
    filepath: String,
}


pub struct Series {
    id: usize,
    name: String,
    labels: HashMap<String, String>,
    variables: Vec<String>,
    records: RwLock<Vec<SeriesRecord>>,
}


pub struct SeriesRecord {
    metrics: Vec<f64>,
    timestamp: i64,
}


pub struct ResultSet {
    unpacked: bool,
    data: Vec<Record>,
    series: Bitmap,
    filters: Vec<(String, Box<dyn Fn(f64) -> bool>)>,
}
```