Specialized Storage for Big Numeric Time Series

Ilari Shafer, Raja R. Sambasivan, Anthony Rowe, Gregory R. Ganger
Carnegie Mellon University

Abstract

Numeric time series data has unique storage requirements and access patterns that can benefit from specialized support, given its importance in Big Data analyses. Popular frameworks and databases focus on addressing other needs, making them a suboptimal fit. This paper describes the support needed for numeric time series, suggests an architecture for efficient time series storage, and illustrates its potential for satisfying key requirements.

1 Introduction

"Big Data" analysis is being used today to yield extraordinary insights in a variety of fields, including cancer research, traffic congestion, and datacenter health. Since no single model is sufficient for all data [28], recent research has led to the creation of many frameworks for storing and supporting analyses on different data types. Examples include frameworks for querying multidimensional maps [6], analyzing hierarchically structured records [19], and analyzing graphs [14]. But, apart from a few recent efforts (e.g., [26]), one important type of Big Data has received remarkably little attention—numeric time series.

We are swimming in numeric time series data. Though much work has demonstrated the utility of analyzing such data via stream processing [2, 7] and data mining [10, 12, 15], support for efficiently storing and querying time series data has languished. For example, traditional frameworks (e.g., relational databases) do not easily cater to the usage models of numeric time series. As such, many important Big Data analyses on time series data (see Figure 1 for examples) are instead performed in ad hoc fashion—e.g., via text files that are read into MATLAB and R. Not only is this ad-hoc process inefficient, it also severely limits ease of use and scalability.

To address the growing needs of time series Big Data, this paper enumerates properties needed from a storage framework to efficiently support numeric time series storage and common access patterns. It shows that traditional databases do not fit the bill, because they are good at supporting unneeded features (e.g., arbitrary updates), but inefficient at supporting the needed ones (e.g., multi-stream analyses and strong compression). Based on the properties identified, we propose an architecture for efficiently supporting numeric time series' unique needs. Initial experiments illustrate the potential of this architecture to provide efficient storage through compression.

Computer system performance metrics: Capacity planning, modeling failures and anomalies
Stock prices: Backtesting financial models, market characterization, mathematical finance
Audio signals: Language translation, music matching
EEG, ECG signals: Illness identification, brain research
Sensor networks: Retrospective surveillance, weather pattern identification, infrastructure planning

Figure 1: Common numeric time series and example Big Data analyses for which they are useful.

2 Unneeded, currently efficient features

Traditional databases support a number of operations and guarantees that are unnecessary for numeric time series, adding inefficiency and unneeded complexity. This section describes five examples.

[U1] **Support for non-numeric queries:** traditional expensive operations such as string-based `LIKE` and equality `JOIN`, which are also present in many stream-processing languages [2, 4, 20], are inapplicable to numeric observational data. This reduces the demand for auxiliary structures such as bitmap indices. Point queries that ask for only a single fact (e.g., the natural query for a key-value store) are also much less common.

[U2] **Support for arbitrary updates to stored data:** whereas more general data can have multiple versions, observational numeric time series are only sensed once. Updates to one observation stream do not read the writes of another stream or write to the same logical location. This property greatly simplifies any approach to providing the guarantees of concurrency control—in particular, consistency maintenance and isolation.

[U3] **Arbitrary deletion:** "random-access" deletion need not be supported: observations of numeric time series are removed, cleaned, or coalesced in large, temporally contiguous swaths. Removal techniques used in practice for time series (e.g., round-robin databases [21] and log rotation) delete or downsample the oldest data.

[U4] **Variable ingest rates:** many (though not all) observational data sources, including sensors, numeric monitoring systems, and investment markets, emit data at predictable, near-constant rates. Some of the avoidable complexity of general stream processing systems arises from the requirement to deal with data with a highly variable ingest rate [2, 20, 27].

[U5] **Strong durability guarantees:** as operational data

that might otherwise be discarded, the need for immediate durability for time series is naturally lower than for transactional data. Furthermore, the storage systems that would run a time series archival system are often more robust than the data sources (e.g., sensors) feeding them.

3 Time series data mining needs

The statistics and data mining approaches commonly applied to archived time series do not resemble traditional database operations. Although many of the inputs of such analyses can be extracted from a general framework, they stress operations that are not typically a primary focus of existing platforms. Furthermore, it is not possible to stream all these approaches over the data (i.e., with a stream processing engine). Three key requirements for these analyses are:

[range] **Support for raw access patterns:** A central primitive shared by most time series mining methods that access raw data is a *range query* between two points in time. This access pattern is visible in the first column of Table 1, which summarizes a few techniques for time series analysis and their common patterns. For example, an analogy to `LIKE` for time series is matching a shorter time series (subsequence) within a longer one; methods for doing so on raw data require scanning ranges of the longer series. Some methods (primarily, techniques that correlate or cluster time series) operate on ranges of multiple streams, as seen in the third column ("multivar").

[summ] **Support for preprocessing and summarization:** A first step taken in most methods is resampling a stream at a fixed rate (here, "preprocessing"), which can be lower than the original signal. Such multiresolution data, even when not explicitly required (the second column in Table 1), can be useful for approximate queries and visualization. Furthermore, many time series data mining approaches do not operate on raw time series data; to keep analyses computationally tractable, they rely on higher-level representations of the data [10, 24]. For the subsequence matching example, a line of work has built specialized indices to accelerate queries [11]. These abstractions use signal processing (e.g., FFT), to summarize and index streams.

As one example of a summarization access pattern, consider running an FFT over each of multiple streams for preprocessing, indexing, or analysis. This requires all data within the desired summarization range of each stream (some algorithms can parallelize a first step over even and odd elements), but each operation only accesses a single stream.

[comp] **Accommodating compression:** Raw numeric time series are often machine-generated and large. As a system for analysis of data that often might otherwise be thrown away, the burden an analysis framework places on infrastructure should be as low as possible without compromising the ability to mine the data. Storing data so that it may be losslessly compressed is, therefore, a key desire.

	Access Patterns				
Technique	Range	Multires	Multivar	All Data	Streaming
Basic operations					
Plotting	✓	-			✓
Zooming visualization	✓	✓			
Preprocessing					
Decimation & interpolation	-			-	-
Similarity, correlation (on raw data)					
Many clustering techniques use summaries					
Auto- & cross-correlation	✓		2	✓	
Convolution	-		2	-	-
SVD (+SVD-based [23])	✓	-	*N*	-	-
Raw dynamic time warping	✓	-	2	✓	
Raw elementwise distance			2		✓
Prediction					
AR(I)MA(X) (forecast)			-		-
AR(I)MA(X) (ID/model)	✓		-	✓	
Query by content (search for similar subsequences)					
Most techniques use summaries such as DFT or PAA					
Sequential Scan	✓				✓
Segmentation / changepoint detection					
Sliding-window			-		✓
Top-down & Bottom-up	✓		-	✓	
Summarization					
Wavelet (DWT, FWT)	✓	✓		✓	
Spectral (DCT, FFT)	✓			✓	
Symbolic (PAA, iSAX [25])	✓	-			✓
Amnesic [22] (PLR,...)					✓

Table 1: Access patterns of common time series queries: Columns correspond to access patterns for some (-) or nearly all (✓) surveyed algorithms for a method (rows), and mean:

- Range: reads contiguous data in the time dimension
- Multires: uses or produces multiresolution data
- Multivar: accesses multiple streams: 2 or multiple (*N*)
- All data: requires the entire time range of operation
- Streaming: can operate in purely streaming fashion

4 The state of the art

Of the existing systems that could satisfy the demands of numeric time series laid out above, column stores come the closest. By storing sorted timestamps and observations in separate, internally contiguous regions, these databases can perform range queries and compress data much more effectively than traditional databases. However, they fall short by having an ingest layer that does not fully exploit the desire to query time series streams

System	Numeric	Multires	All data	API	Compression	Selected architectural differences from ideal
Ideal	✓	✓	✓	¬SQL	specialized	
DataGarage [18]	✓	✓	✓	SQL	domain spec.	specialized for sys. monitoring, SQL DB-backed
RRDtool [21]	✓	✓		range	none	no timestamps, fixed-length raw & aggregate data
FinanceDB [1]	✓		✓	custom	proprietary	static partitioning, offline merge to historical store
tsdb [8]	✓			chunk	generic	thin layer on Berkeley DB, all small chunks
TSDS [29]	✓	✓		filters	generic	mostly a caching layer on top of other stores
OpenTSDB [26]	✓	✓		range	generic	layer on multidimensional map, no subsecond res.
Dremel [19]			✓	SQL	generic	optimized for aggregation queries on nested data
Vertica [17]		✓	✓	SQL	multiple [13]	general-purpose column store
DataSeries [3]			✓	modules	multiple	record-oriented trace format

Table 2: Archival frameworks for time series data. A ✓ in a column signifies: **Numeric:** focused on numeric time series, **Multires:** supports storing or caching multiresolution data, **All data:** supports operations that map over entire streams.

in the same order they arrive (see the second design principle in Section 5.1).

We survey existing systems that are described in the literature in Table 2. Most are effectively (if not nominally) column stores. Among the more widely used tools in practice are RRDtool [21] and (recently) OpenTSDB [26], a column-family store. Both are used primarily in monitoring interfaces. Of all the systems, DataGarage [18] comes close to ideal, but is specialized for datacenter monitoring. We believe time series access demands do not have much overlap with a SQL-based interface, as detailed in Section 6, so we list the ideal API as not SQL. The last column highlights additional differences from an "ideal" solution.

5 Toward improved time series datastores

Based on the requirements above, this section identifies three design principles for addressing the needs of time series data and queries more efficiently by dropping support for generality. An architecture that integrates these principles is shown in Figure 2; its guiding notion is to partition input streams into time-ordered blocks that are specialized for range queries and compression.

5.1 Design principles

Separate incoming timestamps and values: As data from a given numeric time series source enters the framework, it should be separated into two components: *timestamp streams* and *value streams*. Just as in a table, multiple value streams can be associated with a single timestamp stream to reduce the storage overhead. However, this association need not be limited to a predefined schema, particularly when such a schema is not known in advance (e.g., due to adaptive monitoring [16]).

When separated, timestamp streams can frequently be represented compactly. A start value and time interval captures the common case of a fixed sample period [U4] [21], with data indices obtained through computation. The combination of encoding the differences between observations (deltas) and runs in the result generalizes this notion to mostly-periodic timestamp streams, can be done online, and still presents opportunities for efficient overlying index structures [comp]. The next two principles make use of this clean separation.

Partition buffered values: As value streams arrive, they should be partitioned in memory (and/or fast non-volatile storage, if immediate persistence is desired) by the unique identity of the stream (for example, a sensor name plus metric). By doing so, new observations can simply be appended to the tail of time-ordered lists, which support range queries without sorting [range]. Also thanks to this ordering, amnesic [22] or simple multiresolution summaries can be updated online [summ].

Archive in time-ordered blocks: Raw range queries and summary construction demand access to large, temporally contiguous regions of data [range][summ]. To meet these demands and take advantage of the write-once nature of data [U2][U3], the archival primitive of

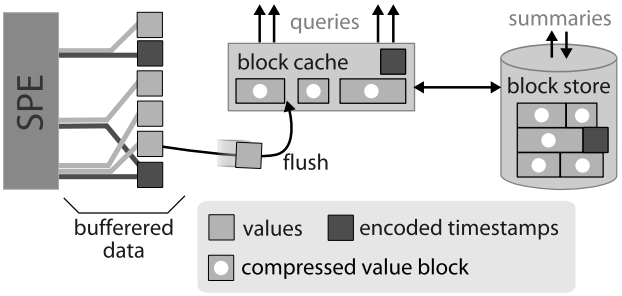

Figure 2: Possible design for numeric time series archival. Streams are separated into timestamps and values, buffered, and written back to a store of compressed blocks. Not shown are metadata storage, higher-level query architecture, or possible distribution across nodes. A stream processing engine (SPE) emphasizes that streaming queries on incoming data are not within scope and can be handled separately.

numeric time series should be blocks of values and timestamps that are ordered by time. Value blocks can contain either raw data or time-oriented summaries [summ].

Incoming numeric time series data naturally arrives time-ordered. When combined with the other two properties, this property allows incoming data to be appended to the current block (or to start a new one) rather than having to be sorted into previously stored blocks. Also, since data is indexed only on time [range][U1], timestamp blocks are the only necessary index, and they can be used for direct lookup without structures such as bloom filters in a table store [6].

Furthermore, unlike SSTables in a table store, these blocks do not need to be recompacted: they are already sorted in their final order and compressed, and they are never updated after they are complete [U2]. Intermittent table store compaction is, therefore, unnecessary [comp].

Unlike the primitives of many of the systems in Table 2, these blocks (once uncompressed) are in a form that is suitable for direct analysis: summarization and resampling techniques can map over them, and aging out older data [U3] amounts to removing the appropriate large value blocks.

5.2 Initial promise: compact storage

To show the potential of our architecture, this section evaluates its ability to support effective compression. Our initial experiments examine how compactly these large persistent blocks can store numeric time series. Specifically, we compare existing systems and our prototype implementation of large time-ordered blocks with regards to how their internal storage representation can compress four datasets. These datasets, shown in Table 3, are selected to be diverse and reasonably large.

The implementation of our proposed layout uses one file per timestamp stream. A given timestamp stream may be associated with multiple value streams, which are also single files (with a maximum size of 4006K), according to the schema of the input data. Timestamps are delta+RLE (DRLE) encoded, and value streams use one of two compressors (LZO or gzip).

For comparison, we use three representative data stores: OpenTSDB, FinanceDB, and SQLite. To our knowledge, OpenTSDB represents the state-of-the-art in large-scale general time series databases, and FinanceDB is a high-performance financial database (its license precludes us from publishing its name). Row-oriented storage such as the default SQLite backend is also widely used. For example, historical stream query work in the database community has built on systems such as PostgreSQL (e.g., [5]) and MySQL (e.g., [9]).

Figure 3 shows the results of comparing our layout with existing storage systems. All measurements are obtained with `du` on the same NTFS filesystem (4K cluster size). Since the `stocks` data consists of many small files in the proposed layout, we conservatively show the larger (actual) size on disk. As a row store, SQLite cannot compress columns, and we suspect that its smaller-than-binary size is due to its variable-width integer format. We believe that the overhead for OpenTSDB is at least partly due to the overhead of row key duplication in HBase, an area of active development for the database.

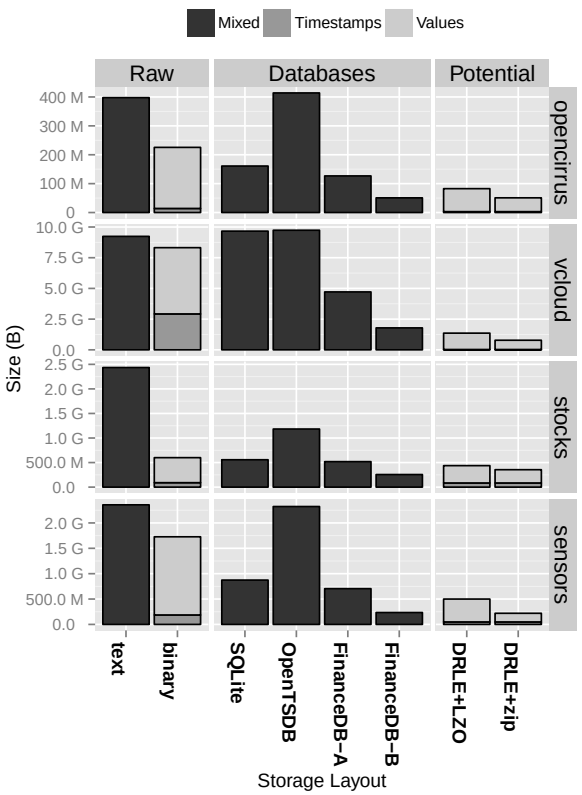

Figure 3: Space used by storage layouts. Different variations on existing and potential approaches are grouped horizontally.

- SQLite tables are built for the schema of incoming data
- OpenTSDB 1.0.0 with default configuration is backed by a single-node HBase server (CDH4 distribution) with LZO HFile compression. Sizes are taken after a completed major compaction, and measure the data table only.
- We compare two compressors in FinanceDB, -A and -B

In all, these early experiments show promise. Even a separation of numeric time series with lightweight timestamp encoding and generic value compression can deliver storage sizes that are similar to the optimized FinanceDB, smaller than more broadly-used systems, and significantly more compact than the raw formats used for analysis. With data-specific compressors, we would expect additional reduction in size.

6 Discussion

While this initial experiment examines potential space savings, performance is clearly an important consideration, as compression comes at the cost of query latency

Dataset	# Streams (t/v)	Granularity
opencirrus	78/1170	60 sec
One month of long-term systems monitoring data		
vcloud	2209/4315	20 sec
Systems monitoring data from a virtualized cluster		
stocks	8238/34513	1 day
Historical investment prices from Yahoo! Finance		
sensors	58/477	1 sec[1]
Sensor network data (temperature, humidity, power, ...)		

Table 3: Datasets used for experiment. # Streams lists the number of timestamp/value (t/v) streams in each, which are sampled at the period in **Granularity**, modulo jitter. Timestamps are seconds since the epoch, and like values are 4-byte integers. The vcloud dataset does not have a natural tabular schema; we build schemas that deduplicate timestamps.

and (often, though not always) throughput. A natural next step is to explore the tradeoff between space and speed for numeric time series archival. Opportunities for achieving better points in this tradeoff space include indexing built on top of encoded timestamps and fast specialized time series compressors, which can help reduce latencies for interactive analyses.

Important questions surround how this architecture interacts with existing systems. For queries, the presence of existing systems that offer a range-based interface (see Table 2) suggests that an API that offers range queries and simple aggregates might meet many current needs, and that there are use cases that view numeric data in isolation. Nonetheless, there are certainly broader options worth considering—for example, offering common summaries as built-in operations. Alternately, the system need not be a complete database, and could simply exist as another storage engine within a DBMS. For ingest, it may be desirable to not demand a schema upfront (which was necessary for SQLite and FinanceDB), to reduce the burden of adding or modifying data sources. For the system itself, many components ought to take advantage of existing underlying infrastructure: for example, a file-like interface that supports seek and append seems a natural fit for the block store.

7 Conclusion

Despite the variety of existing frameworks for warehousing large data, the needs of numeric time series mining could benefit from a more focused solution that has the potential to be more efficient. We contend that there is a significant opportunity to rethink the fundamentals of numeric time series storage and demonstrate through preliminary experiments that following a few guiding principles with even simple methods can often archive data more compactly than existing frameworks.

[1]Recorded timestamps are lower resolution than the inherent 640ms sample rate of the data. As was done with a previous import of the data, we have removed duplicate observations within a single interval.

Acknowledgements: We thank Max Buevich, Susan Cross, Christos Faloutsos, Mitch Franzos, Rajeev Gandhi, Georg M. Goerg, Mark Lucovsky, Michelle Mazurek, Vijay Pandurangan, Florentina Popovici, Niranjini Rajagopal, Randy Sargent, Alexey Tumanov, and Xiaolin Zang for their thoughts and assistance. We thank the members and companies of the PDL Consortium (including Actifio, APC, EMC, Emulex, Facebook, Fusion-io, Google, Hewlett-Packard Labs, Hitachi, Intel, Microsoft Research, NEC Laboratories, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, VMware, and Western Digital) for their interest, insights, feedback, and support. This research was sponsored in part by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC), and by an NSF Graduate Research Fellowship.

References

[1] Database name anonymized due to licensing restrictions.
[2] ABADI, D.J. ET AL. Aurora: a new model and architecture for data stream management. *The VLDB Journal 12*, 2 (Aug. 2003).
[3] ANDERSON, E. ET AL. DataSeries: An efficient, flexible data format for structured serial data. Tech. Rep. HPL-2009-323, HP Labs, 2009.
[4] ARASU, A., BABU, S., AND WIDOM, J. An abstract semantics and concrete language for continuous queries over streams and relations. In *DBPL'03*.
[5] BALAZINSKA, M. ET AL. Moirae: History-Enhanced Monitoring. In *CIDR'07*.
[6] CHANG, F. ET AL. Bigtable: a distributed storage system for structured data. In *OSDI '06*.
[7] CHEN, J., DEWITT, D. J., TIAN, F., AND WANG, Y. Niagaracq: A scalable continuous query system for internet databases. In *SIGMOD'00*.
[8] DERI, L., MAINARDI, S., AND FUSCO, F. tsdb: a compressed database for time series. In *TMA'12*.
[9] DINDAR, N. ET AL. Efficiently correlating complex events over live and archived data streams. In *DEBS'11*.
[10] ESLING, P., AND AGON, C. Time-series data mining. *ACM Computing Surveys 45*, 1 (Dec. 2012).
[11] FALOUTSOS, C., RANGANATHAN, M., AND MANOLOPOULOS, Y. Fast subsequence matching in time-series databases. In *SIGMOD'94*.
[12] HALEVY, A., NORVIG, P., AND PEREIRA, F. The unreasonable effectiveness of data. *IEEE Intelligent Systems 24*, 2 (Mar. 2009).
[13] HP. Vertica enterprise edition 6.0: Data encoding and compression. https://my.vertica.com/docs/6.0.1/HTML/index.htm#2415.htm.
[14] KANG, U ET AL. Gbase: an efficient analysis platform for large graphs. *The VLDB Journal 21*, 5 (2012).
[15] KEOGH, E. A decade of progress in indexing and mining large time series databases. In *VLDB'06*.
[16] KUTARE, M., EISENHAUER, G., AND WANG, C. Monalytics: Online monitoring and analytics for managing large scale data centers. *ICAC'10*.
[17] LAMB, A. ET AL. The Vertica analytic database: C-store 7 years later. *VLDB Endowment 5*, 12 (2012).
[18] LOBOZ, C., SMYL, S., AND NATH, S. DataGarage: Warehousing Massive Performance Data on Commodity Servers. In *VLDB'10*.
[19] MELNIK, S. ET AL. Dremel: interactive analysis of web-scale datasets. In *VLDB'10*.
[20] MOTWANI, R. ET AL. Query processing, resource management, and approximation in a data stream management system. In *CIDR'03*.
[21] OETIKER, T. RRDtool, 2011. http://www.mrtg.org/rrdtool/.
[22] PALPANAS, T. ET AL. Online amnesic approximation of streaming time series. In *ICDE'04*.
[23] PAPADIMITRIOU, S., SUN, J., AND FALOUTSOS, C. Streaming pattern discovery in multiple time-series. In *VLDB'05*.
[24] RATANAMAHATANA, C.A. ET AL. Mining time series data. In *Data Mining and Knowledge Discovery Handbook*. 2010.
[25] SHIEH, J., AND KEOGH, E. iSAX: indexing and mining terabyte sized time series. In *KDD'08*.
[26] SIGOURE, B. OpenTSDB. http://opentsdb.net/.
[27] SIRISH, C. ET AL. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR'03*.
[28] STONEBRAKER, M., AND CETINTEMEL, U. "one size fits all": An idea whose time has come and gone. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pp. 2–11.
[29] WEIGEL, R.S. ET AL. TSDS: high-performance merge, subset, and filter software for time series-like data. *Earth Science Informatics 3*, 1-2 (June 2010).