



Chronix: Long Term Storage and Retrieval Technology for Anomaly Detection in Operational Data

*Florian Lautenschlager, QAware GmbH; Michael Philippsen and Andreas Kumlehn,
Friedrich-Alexander-Universität Erlangen-Nürnberg; Josef Adersberger, QAware GmbH*

<https://www.usenix.org/conference/fast17/technical-sessions/presentation/lautenschlager>

**This paper is included in the Proceedings of
the 15th USENIX Conference on
File and Storage Technologies (FAST '17).**

February 27–March 2, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-36-2

**Open access to the Proceedings of
the 15th USENIX Conference on
File and Storage Technologies
is sponsored by USENIX.**

Chronix: Long Term Storage and Retrieval Technology for Anomaly Detection in Operational Data

Florian Lautenschlager,¹ Michael Philippsen,² Andreas Kumlehn,² and Josef Adersberger¹

¹*QAware GmbH, Munich, Germany*
florian.lautenschlager@qaware.de
josef.adersberger@qaware.de

²*University Erlangen-Nürnberg (FAU)
Programming Systems Group, Germany*
michael.philippsen@fau.de

Abstract

Anomalies in the runtime behavior of software systems, especially in distributed systems, are inevitable, expensive, and hard to locate. To detect and correct such anomalies (like instability due to a growing memory consumption, failure due to load spikes, etc.) one has to automatically collect, store, and analyze the operational data of the runtime behavior, often represented as time series. There are efficient means both to collect and analyze the runtime behavior. But traditional time series databases do not yet focus on the specific needs of anomaly detection (generic data model, specific built-in functions, storage efficiency, and fast query execution).

The paper presents Chronix, a domain specific time series database targeted at anomaly detection in operational data. Chronix uses an ideal compression and chunking of the time series data, a methodology for commissioning Chronix' parameters to a sweet spot, a way of enhancing the data with attributes, an expandable set of analysis functions, and other techniques to achieve both faster query times and a significantly smaller memory footprint. On benchmarks Chronix saves 20%–68% of the space that other time series databases need to store the data and saves 80%–92% of the data retrieval time and 73%–97% of the runtime of analyzing functions.

1 Introduction

Runtime anomalies are hard to locate and their occurrence is inevitable, especially in distributed software systems due to their multiple components, different technologies, various transport protocols, etc. These anomalies influence a system's behavior in a bad way. Examples are an anomalous resource consumption (e.g., high memory consumption, growing numbers of open files, low CPU usage), sporadic failures due to synchronization problems (e.g., deadlock), or security issues (e.g., port scanning activity). Whatever their root causes are,

the resulting behavior is critical and may in general lead to economic or reputation loss (e.g., loss of sales, productivity, or data). Almost every software system has hidden anomalies that occur sooner or later. Hence one needs to detect them in an automated manner soon after their occurrence in order to initiate measures.

There are efficient means both to collect and analyze the runtime behavior. Tools [10, 18, 44] can collect all kinds of operational data like metrics (e.g., CPU usage), traces (e.g., method calls), and logs. They represent such operational data as time series. Analysis tools and research papers [27, 33, 43, 44, 47] focus on the detection of anomalies in that data. But there is a gap between collection and analysis of operational time series, because typical time series databases are general-purpose and not optimized for the domain of this paper. They typically have a data model that focuses on series of primitive type values, e.g., numbers, booleans, etc. Their built-in aggregations only support the analysis of these types. Furthermore, they do not support an explorative and correlating analysis of all the raw operational data in spontaneous and unanticipated ways. With a domain specific data model and with domain specific built-in analysis functions we achieve better query analysis times. General-purpose time series databases already have a good storage efficiency. But we show that by exploiting domain specific characteristics there is room for improvement.

Chronix, a novel domain specific time series database, addresses the collection and analysis needs of anomaly detection in operational data. Its contributions are a multi-dimensional generic time series data model, built-in domain specific high-level functions, and a reduced storage demand with better query times. Section 2 covers the requirements of such a time series database. Section 3 presents Chronix. Section 4 discusses the commissioning of Chronix and describes a methodology that finds a sweet performance spot. The quantitative evaluation in Section 5 demonstrates how much better Chronix works than general-purpose time series databases.

2 Requirements

Three main requirements span the design space of a time series database that better suits the needs of an anomaly detection in operational data: a generic data model for an explorative analysis of all types of operational data, analysis support for detecting runtime anomalies, and a time- and space-efficient lossless storage.

As shown in Table 1, the established general-purpose time series databases Graphite [12], InfluxDB [24], KairosDB [25], OpenTSDB [32], and Prometheus [36] do not or only partially fulfill these requirements.

Generic data model. Software systems are typically distributed and a multi-dimensional data model allows to link operational data to its origin (host, process, or sensor). We argue that a generic multi-dimensional data model is necessary for an explorative and correlating analysis to target non-trivial anomalies. *Explorative* means that a user can query and analyze the data without any restrictions (as by a set of pre-defined queries) to verify hypotheses. *Correlating* means that the user can combine queries on all different types of operational data (e.g., traces, metrics, etc.) without any restrictions (as by pre-defined joins and indexes). Imagine an unanticipated need to correlate the CPU usage in a distributed system with the executed methods. Such an analysis first queries the CPU usage on the hosts (metrics) for a time range. Second, it queries which methods were executed (traces). Finally, it correlates the results in a histogram.

The traditional time series databases have a specific multi-dimensional data model for storing mainly scalar/numeric values. But in the operational data of a software system there are also traces, logs, etc. that often come in (structured) strings. As explicit string encodings require implementation work, often only such raw data is encoded and collected that appears useful at the time of collection. Other raw data is lost, even though an explorative analyses may later need it. Moreover, any string encoding loses the semantics that come with the data type. Therefore, while the traditional time series databases support explorative and correlating anal-

Table 1: Design space requirements.

Time Series Database	Generic data model	Analysis support	Lossless long term storage
Graphite	○	◐	○
InfluxDB	◐	◐	●
OpenTSDB	○	○	●
KairosDB	◐	◐	●
Prometheus	○	◐	◐
Chronix	●	●	●

○ = No, ◐ = Partly, ● = Yes

yses on scalar values, they often fail to do so efficiently for generic time series data (e.g., logs, traces, etc.). InfluxDB also supports strings and booleans. KairosDB is extensible with custom types. Both lack operators and functions and hence only partly fulfill the requirements.

Analysis support. Table 2 is an incomplete list of basic and high-level analysis functions that a storage and retrieval technology for anomaly detection in operational data must support in its plain query language. Graphite, InfluxDB, KairosDB and Prometheus only have a rich set of basic functions for transforming and aggregating operational data with scalar values. OpenTSDB even supports only a few of them. But domain specific high-level functions that other authors [27, 33, 43, 44, 47] successfully use for anomaly detection in operational data (lower part of Table 2) should be built in natively to execute them as fast as possible, without network transfer costs, etc. The evaluation in Section 5 shows the runtime benefits of having them built-in instead of emulating them.

As such a set of functions can never be complete its extensibility is also a requirement.

Efficient lossless long term storage. Complex analy-

Table 2: Common query functions.

	Graphite	InfluxDB	OpenTSDB	KairosDB	Prometheus	Chronix
Basic						
distinct	×	✓	×	×	×	✓
integral	✓	×	×	×	×	✓
min/max/sum	✓	✓	✓	✓	✓	✓
count	×	✓	✓	✓	✓	✓
avg/median	✓	✓	✓	✓	✓	✓
bottom/top	×	✓	×	×	✓	✓
first	✓	✓	×	✓	×	✓
last	✓	✓	×	✓	×	✓
percentile (p)	✓	✓	✓	✓	✓	✓
stddev	✓	✓	✓	✓	✓	✓
derivative	✓	✓	×	✓	✓	✓
nnderivative	✓	✓	×	×	×	✓
diff	×	✓	×	✓	✓	✓
movavg	✓	✓	×	×	×	✓
divide/scale	✓	✓	×	✓	✓	✓
High-level						
sax [33]	×	×	×	×	×	✓
fastdtw [38]	×	×	×	×	×	✓
outlier	×	×	×	×	×	✓
trend	×	×	×	×	×	✓
frequency	×	×	×	×	×	✓
grpsize	×	×	×	×	×	✓
split	×	×	×	×	×	✓

ses to identify and understand runtime anomalies, trends, and patterns use machine learning, data mining, etc. They need quick access to the full set of all the raw operational data, including its history. This allows interactive response times that lead to qualitatively better explorative analysis results [30]. Thus a lossless long term storage is required that achieves both a small storage demand and good access times. A domain specific storage should also allow to store additional pre-aggregated or in other ways pre-processed versions of the data that enhance domain specific queries.

The operational characteristics of anomaly detection tasks are also specific as there are comparatively few batch writes and frequent reads/analyses, hence the storage should be optimized for this.

As Table 1 shows, the traditional time series databases (except for Graphite due to its round-robin storage, see Section 6) can be used as long term storage for raw data. Their efficiency in terms of domain specific storage demands and query runtimes is insufficient as the evaluation in Section 5 will point out. Prometheus is designed as short-term storage with a default retention of two weeks. Furthermore it does not scale, there is no API for storing data, and it uses hashes for time series identification (the resulting collisions may lead to a seldom loss of a time series value). It is therefore not included in the quantitative evaluation.

3 Design and Implementation

Generic data model. Chronix uses a generic data model that can store all kinds of operational data. The key element of the data model is called a *record*. It stores an ordered chunk of time series data of an arbitrary type (n pairs of timestamp and value) in a binary large object. There is also type information (e.g., metric, trace, or log) that defines the available type-specific functions and the ways to store or access the data (e.g., serialization, compression, etc.). A record stores technical fields (version and id of the record, as needed by the underlying Apache Solr [5]), two timestamps for start and end of the time range in the chunk, and a set of user-defined attributes, for example to describe the origin of the operational data, e.g., host and process. Thus the data model is multi-dimensional. It is explorative and correlating as queries can use any combination of the attributes stored in a chunk as well as the available fields.

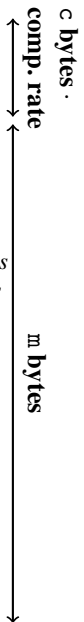
Listing 1 shows a record for a metric (type) time series, with dimensions for host name, process, group (a logical group of metrics), and metric. Description and version are optional attributes.

The syntax of Chronix queries is:

```
q=<solr-query> [ & cf=<chronix-functions> ]
```

```

1 record{
2   //payload
3   data:compressed{<chunk of time series data>}
4
5   //technical fields (storage dependent)
6   id: 3dce1de0-...-93fb2e806d19 //16 bytes
7   _version_: 1501692859622883300 //8 bytes
8
9   //logical fields
10  start: 1427457011238 //27.3.2015 11:51:00 8 bytes
11  end: 1427471159292 //27.3.2015 15:45:59 8 bytes
12  type: metric //Data types: metric, log, trace etc.
13
14  //optional dimensions
15  host: prodI5
16  process: scheduler
17  group: jmx
18  metric: heapMemory.Usage.Used
19
20  //optional attributes
21  description: Benchmark
22  version: v1.0-RC
23 }
```



Listing 1: Record with fields, attributes, and dimensions.

The underlying Solr finds the data according to *q*. Then we apply the optional Chronix functions *cf* before we ship the result back to the client. *Cf* is a ;-separated list of filters, each of the form

```
<type>'{'<functions>}'
```

The ;-separated list of <type>-specific <functions> is processed as a pipeline. A <function> has the form

```
<analysis-name>[:<params>]
```

for analysis names from Table 2. Here is an example from the evaluation project 4 in Section 5:

```
q=host:prod* AND type: [lsof OR strace]
& cf=lsof{grpsize:name,pipe};strace{split:2030}
```

First, *q* takes all time series from all hosts whose name starts with *prod* and that also hold data of the UNIX commands *lsof* or *strace*. Then *cf* applies two functions. To the *lsof* data *cf* applies *grpsize* to select the group named *name* and counts the occurrences of the value/word *pipe* in that group. On the *strace* data *cf* performs a *split*-operation on the command-column. For each of the arguments (here just for the file handle ID 2030) it produces a split of the data that contains "2030".

Analysis support. Chronix offers all the basic and high-level functions listed in Table 2, e.g., there are detectors for outliers and functions that check for trends, similarities (*fastdtw*), and patterns (*sax*). The plug-in mechanism of Chronix allows to add functions that run server-side. They are fast as they operate close to the data

```

1 class GroupSize implements ChronixAnalysis<LsofTS>{
2   public GroupSize(String[] args) {
3     field = args[0];filters = args[1];
4   }
5   public void execute(LsofTS ts, Result result) {
6     result.put(this,new Map())
7     for (Group group : ts.groupBy(field)) {
8       if (filters.contains(group.key()))
9         result.get(this).put(group.key(), group.value().size());
10    }
11  }
12  public String getQueryName() {return "grpsize";}
13  public String getTimeSeriesType() {return "lsof";}
14 }

```

Listing 2: Plug-in for a function *GroupSize*.

without shipping (e.g., through HTTP). For security reasons only an administrator is capable (and thus responsible) for installing plug-ins while regular users can only use them afterwards. The plug-in developer has to implement an interface called *ChronixAnalysis*. It defines three methods: *execute* holds the code that analyzes a time series, *getQueryName* is the function’s name in Chronix’ Query Language, and *getTimeSeriesType* binds the function to a time series type. An additional Google Guice [19] module is needed before (after a reboot of Chronix with the operator code in the classpath) the newly added function is available. Chronix parses a given query and delegates the execution to the plugged-in function. Listing 2 illustrates how to add the *grpsize* analysis for the time series type *lsof*. The function runs on the server, groups the time series data with respect to a field (given as its first argument), and returns the sizes of the groups. The raw time series data is never shipped to the client. With this function added, regular users just need to call one function instead of several queries to emulate the semantics themselves.

Functionally-lossless long term storage. Chronix optimizes storage demands and query times and is designed for few batch writes and frequent reads. The storage is functionally-lossless and preserves all the data that can potentially be useful for anomaly detection. We clarify the two situations below when *functionally-lossless* is different from *lossless*.

Chronix’ pipeline architecture has four building blocks that run multi-threaded: *Optional Transformation*, *Attributes and Chunks*, *Compression*, and *Multi-Dimensional Storage*, see Fig. 1. Not shown is the input buffering (called Chronix Ingester in the distribution) that batches incoming raw data points.

The *Optional Transformation* can enhance or extend what is actually stored in addition to the raw data, or instead of it. The goal of this optional phase is an opti-

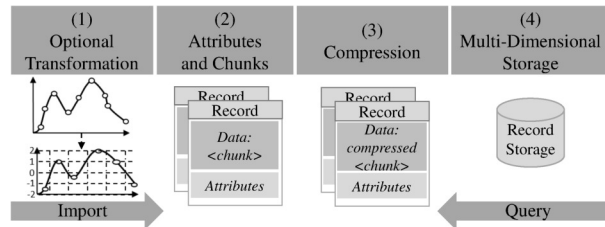


Figure 1: Building blocks of Chronix: (1) optional transformation, (2) grouping of time series data into chunks plus attributes, (3) compression of chunks, (4) storage of chunks with their attributes. The data flows from left to right for imports. For queries it is the other way round.

mized format that better supports use-case specific analyses. For example, it is often easier to identify patterns in time series when a symbolic representation is used to store the numerical values of a time series [33]. Other examples are averaging, a Fourier transformation [9], Wavelets [35], or a Piecewise Aggregate Approximation [26]. The more is known about future queries and about the shape or structure of a time series or its pattern of redundancy, the more can a transformation speed up analyses. In general, this phase can add an additional representation of the raw data to the record. Storing the raw data can even be omitted if it can be reconstructed. In rare situations, an expert user knows for sure that certain data will never be of interest for any of her/his anomaly detection tasks. Think of a regular log file that in addition to the application’s log messages also has entries that come from the framework hosting the application. The latter often do not hold anomaly-related information. It is such data that the expert user can decide to drop, keeping everything else and leaving the Chronix store *functionally-lossless*. The decision is irreversible – but this is the same with the other time series databases.

Attributes and Chunks breaks the raw time series (or the result of the optional transformation) into chunks of *n* data points that are serialized into *c* bytes. Instead of storing single points, chunking speeds up the access times. It is known to be faster to read one record that contains *n* points instead of reading *n* single points. A small *c* leads to many small records with potentially redundant user-defined attributes between records. The value of *c* is therefore a configuration parameter of the architecture.

This stage of the pipeline also calculates both the required fields and the user-defined attributes of the records. The required fields (of the data model) are the binary data field that holds a chunk of the time series, and the fields *start* and *end* with timestamps of the first and the last point of the chunk. In addition, a record can have user-defined attributes to store domain specific information more efficiently. For instance, information that

```

1 public interface RecordConverter<T> {
2     //Convert a record to a specific type.
3     //Use queryStart and queryEnd to filter the record.
4     T from (Record r, long queryStart, long queryEnd);
5     //Convert a specific type to a record.
6     Record to (T tsChunk);
7 }

```

Listing 3: The time series converter interface.

is known to be repetitive for each point, e.g., the host name, can be stored in an attribute of the record instead of encoding it into the chunk of data multiple times.

As it is specific for a time series type which fields are repetitive Chronix leaves it to an expert user to design the records and the fields. Redundancy-free data chunks are a domain specific optimization that general-purpose time series databases do not offer. To design the records and fields the expert has two tasks: (1) define the conversion of a time series chunk to a record (see the example `RecordConverter` interface in Listing 3) and (2) define the static schema in Apache Solr that lists the field names, the types of the values, which fields are indexed, etc. Chronix stores every record that matches the schema.

Compression processes the chunked data. Chronix exploits domain specific characteristics in three ways.

First, Chronix compresses the operational data significantly as there are only small changes between subsequent data points. The cost of compression is acceptable as there are only few batch writes. When querying the data, compression even improves query times as compressed data is transferred to the analysis client faster due to the reduced amount.

Second, time series for operational data often have periodic time intervals, as measurements are taken on a regular basis. The problem in practice is that there is often a jitter in the timestamps, i.e., the time series only have *almost*-periodic time intervals because of network latency, I/O latency, etc. Traditional time series databases use optimized ways to store periodic time series (run-length encoding, delta-encoding, etc.). But in case of jitter they fall back to storing the full timestamps/deltas. In contrast, Chronix' Date-Delta-Compaction (DDC) exploits the fact that in its domain the exact timestamps do not matter that much, at least if the difference between the expected timestamp and the actual timestamp is not too large. Here Chronix' storage is *functionally-lossless* because by default it drops timestamps if it can almost exactly reproduce them. If in certain situations an expert user knows that the exact timestamps do matter, they can be kept. In contrast to timestamp jitter, Chronix never drops the exact values of the data points. They always matter in the domain of anomaly detection. Nevertheless,

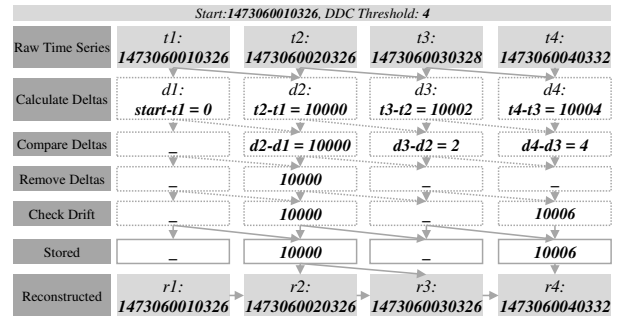


Figure 2: DDC calculates timestamp deltas (0, 10000, 10002, 10004) and compares them (0, 10000, 2, 4). It removes deltas below a threshold of 4 (–, 10000, –, –). Without additional deltas, the reconstruction would yield timestamps with increments of 10000. Since the fourth of them would be too far off from t_4 (drift of 6) DDC stores the correcting delta 10006. With the resulting two stored deltas (–, 10000, –, 10006) the reconstructed timestamps are error-free, even for r_4 .

some parts of the related work are lossy, see Table 1.

The central idea of the DDC is: When storing an almost-periodic time series, the DDC keeps track of the expected next timestamp and the actual timestamp. If the difference is below a threshold, the actual timestamp is dropped as its reconstructed value is close enough. The DDC also keeps track of the accumulated drift as the difference between the expected timestamps and actual timestamps adds up with the number of data points stored. As soon as the drift is above the threshold, DDC stores a correcting delta that brings the reconstruction back to the actual timestamp. DDC is an important domain specific optimization. See Section 4 for the quantitative effects. The DDC threshold is another commissioning parameter. Fig. 2 holds an example.

There are related approaches [34] that apply a similar idea to both the numeric values and the timestamps. Chronix' DDC avoids the lossy compression of values as the details matter for anomaly detection. Chronix also exploits the fact that deltas are much smaller than full timestamps and that they can be stored in fewer bits. Chronix' serialization uses Protocol Buffers [20].

Third, to further lower the storage demand, Chronix compresses the records' binary data fields. The attributes remain uncompressed for faster access. Chronix uses one of the established lossless compression techniques $t = \text{bzip2}$ [39], gzip [14], LZ4 [11], Snappy [21], and XZ [42] (others can be plugged in). Since they have a varying effectiveness depending on the size of the data blocks that are to be compressed, the best choice t is another commissioning parameter.

The *Multi-Dimensional Storage* handles large records

of pure data in a compressed binary format. Only the logical fields, attributes, and dimensions that are necessary to locate the records are explicitly visible to the data storage which uses a configurable set of them for constructing indexes. (Dimensions can later be changed to match future analysis needs.) Queries can use any combination of the dimensions to efficiently locate records, i.e., to find a matching (compressed) chunk of time series data. Information about the data itself is not visible to the storage and hence it is open to future changes as needed for explorative and correlating analyses. Chronix is based on Apache Solr as both the document-oriented storage format of the underlying reverse-index Apache Lucene [4] and the query opportunities match Chronix' requirements. Furthermore Lucene applies a lossless compression (LZ4) to all stored fields in order to reduce the index size and Chronix inherits the scalability of Solr as it runs in a distributed mode called SolrCloud that implements load balancing, data distribution, fault tolerance, etc.

4 Commissioning

Many of the available general-purpose time series databases come with a set of parameters and default values. It is often unclear how to adjust these values to tune the database so that it performs well on the use-case at hand.

We now discuss the commissioning that selects values for Chronix' three adjustable parameters: d (DDC threshold), c (chunk size), and t (compression technique). Commissioning has two purposes: (1) to find default parameters for the given domain (and to show the effects of an unfortunate choice) and (2) to describe a tailoring of Chronix for use-case specific characteristics.

Let us start with a typical domain specific dataset and query mix (composed from real-world projects). We then sketch the measurement infrastructure that the evaluation in Section 5 also uses and discuss the commissioning.

Commissioning Data = Dataset plus Query Mix. To determine its default parameter configuration, Chronix relies on three real-world projects that we consider typical for the domain of anomaly detection in operational data. From these projects, Chronix uses the time series data and the queries that analyze the data.

Project 1 is a web application for searching car maintenance and repair instructions. In production, 8 servers run the web application to visualize the results and 20 servers perform the users' searches. The operational time series data is analyzed to understand the resource consumption for growing numbers of users and new functions, i.e., to answer questions like: 'How do multiple users and new functions affect the CPU load, memory consumption, or method runtimes?', or 'Is the time needed for a user search still within predefined limits?'

Table 3: Project Statistics.

Project	1	2	3	total
pairs (mio)	2.4	331.4	162.6	496.4
time series	1,080	8,567	4,538	14,185

(a) Pairs and time series per project.

Project	1	2	3	average
Attributes (bytes)	43	47	48	46

(b) Average size of attributes per record.

r	0.5	1	7	14	21	28	56	91
q	15	30	30	10	5	3	1	2

(c) Time ranges (in days) and their occurrence.

Project 2 is a retail application for orders, billing, and customer relations. The production system has a central database, plus two servers. From their local machines, users run the application on the servers via a remote desktop service. The analysis goals are to investigate the impact of a new JavaFX-based UI Framework that replaces a former Swing-based version, and to locate the causes of various reported problems, e.g., memory leaks, high CPU load, and long runtimes of use-cases.

Project 3 is a sales application of a car manufacturer. There are two application servers and a central database server in the production environment. The analysis goals are to understand and optimize the batch-import, to identify the causes of long-running use-cases reported by users, to improve the database layer, and understand the impact that several code changes have.

In total, the projects' operational data have about 500 million pairs of timestamp and (scalar) value in 14,185 time series of diverse time ranges and metrics, see Table 3(a). Two projects have an almost-periodic time interval of 30 or 60 seconds. All projects also have event-driven data, e.g., the duration of method calls. There are recurring patterns (e.g., heap usage), sequences of constant values (e.g., size of a connection pool), or even erratic values (e.g., CPU load).

All time series of the three projects have the same user-defined attributes (host, process, group, and metric) that takes 46 bytes on average, see Table 3(b). The required fields of the records take 40 bytes, leading to a total of $m = 40 + 46 = 86$ uncompressed bytes per record.

The three projects have 96 queries in total. Table 3(c) shows what time ranges (r) they are asking for and how often such a time range is needed (q). For example, there are two queries that request the log data that was accumulated over 3 months (91 days).

Measurement Infrastructure. Measurements were conducted on a 12-core Intel Xeon CPU E5-2650L v3@1.80GHz, equipped with 32 GB of RAM and a 380 GB SSD and operating under Ubuntu 16.04.1 x64.

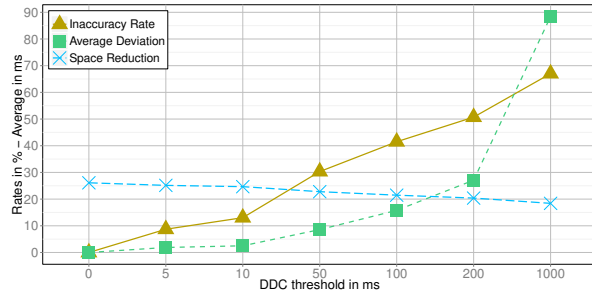


Figure 3: Impact of different DDC thresholds.

Commissioning of the DDC threshold. The higher the DDC threshold is, the higher is the deviation between the actual and the reconstructed timestamp. On the other hand, higher DDC thresholds result in fewer deltas that hence need less storage. Since for anomaly detection, accuracy is more important than storage consumption, and since the acceptable degree of inaccuracy is use-case specific, the commissioning of the DDC threshold focuses on accuracy and only takes the threshold’s impact on the storage efficiency into account if there is a choice.

The commissioning works as follows: For a broad range of potential thresholds, apply the DDC to the time series in the dataset. For each timestamp, record whether the reconstructed value is different from the actual timestamp, and if so, how far the reconstructed value is off. The fraction of the number of inaccurately reconstructed timestamps to error-free ones is the *inaccuracy rate*. For all inaccurately reconstructed timestamps compute their *average deviation*. With those two values plotted, the commissioner selects a threshold that yields the desired level of accuracy.

Default Values. The commissioning of the DDC threshold can be done for individual time series or for all time series in the dataset. For the default value we use all time series that are not event-driven. For thresholds from 0 milliseconds up 1 second. Fig. 3. shows that the *inaccuracy rate* grows up to 67%; the *average deviation* of the reconstructed timestamps grows up to 90 ms.

From the experience gained from anomaly detection projects (the three above projects are among them) an average deviation of 30 ms seems to be reasonably small – recall that the almost-periodic time series in the dataset have intervals of 30 or 60 seconds. The acceptable jitter is thus below 0.1% or 0.05%, resp. Therefore we choose a default DDC threshold of $d = 200$ ms which implies an *inaccuracy rate* of about 50% that we deem acceptable because of the absolute size of the deviation.

Note that the DDC is effective as the resulting data only take 27% to 19% of the original space. But for the dataset the curve of the *space reduction* is too flat to affect the selection of the DDC threshold.

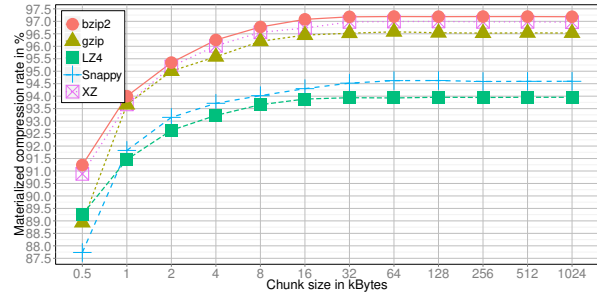


Figure 4: Materialized compression rate in the data store.

Commissioning of the compression parameters. Many of the standard compression techniques achieve better compression rates on larger buffers of bytes [7]. On the other hand, it takes longer to decompress larger buffers to get to the individual data point. Because of the conflicting goals of storage efficiency and query runtimes, Chronix uses a domain specific way to find a sweet spot. There are three steps.

1. Minimal Chunk Size. First, Chronix finds a minimal chunk size for its records. Chronix is not interested in the compression rate in a memory buffer. For anomaly detection, what matters instead is the total storage demand, including the size of the index files (smaller for larger and fewer records). On the other hand, smaller chunks have more redundancy in the records’ (potentially) repetitive uncompressed attributes which makes the compression techniques work better. We call the quotient of this total storage demand and the size of the raw time series data the *materialized compression rate*.

Finding a minimal chunks size works as follows: For a range of potential chunk sizes, construct the records (with their chunks and attributes) from the DDC-transformed time series data and compress them with the standard compression techniques. With the materialized compression rate plotted, the commissioner finds the minimal chunk size where saturation sets in.

Default Values. Fig. 4 shows the materialized compression rates that $\tau =$ bzip2, gzip, LZ4, Snappy, and XZ achieve on records with various chunk sizes that are constructed from the (DDC-transformed) dataset. Saturation sets in around a minimal chunk size of $c_{min} = 32$ KB. Larger chunks do not improve the compression rate significantly, regardless of the compression technique.

2. Candidate Compression Techniques. Then Chronix drops some of the standard compression techniques from the set of candidates. Papers on compression techniques usually include benchmarks on decompression times. But for the domain of anomaly detection, the decompression of a whole time series that is in a memory buffer is irrelevant. What matters instead is the total time needed to find and access an individual record, to then ship the

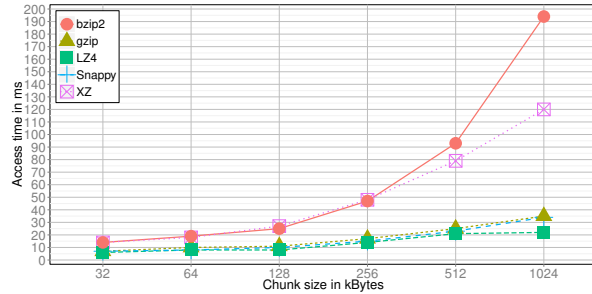


Figure 5: Access time for a single chunk (in ms).

compressed record to the client, and to decompress it there. Note that the per-byte cost of shipping goes down with growing chunks due to latency effects.

Finding the set of candidate compression techniques works as follows: For potential chunk sizes above c_{min} , find a record, ship it, and reconstruct the raw data. For meaningful results, process all records, and compute the average runtime. The commissioner drops those compression techniques that have a slow average access to single records. The reason is that in anomaly detection data is read much more often than written.

Default Values. On the compressed dataset, all of the standard compression techniques take longer to find, access, and decompress larger chunks, see Fig. 5. It is obvious that bzip2 and XZ can be dropped from the set of compression techniques because the remaining ones clearly outperform them.

3. Best Combination. Now that the range of potential values for c is reduced and the set of candidates for the compression technique t is limited, the commissioning considers all combinations. Since query performance is more important than storage efficiency for the domain of anomaly detection, commissioning works with a typical (or a use-case specific) query mix. The access time to a single record as considered above can only be an indicator, because real queries request time ranges that are either part of a single record (a waste of time in shipping and decompression if the record is large) or that span multiple records (a waste of time if records are small).

This commissioning step works as follows: Randomly retrieve q time ranges of size r from the data. The values of r and q reflect the characteristics of the query mix, see Table 3(c). Repeat this 20 times to stabilize the results. For a query of size r the commissioning does not pick a time series that is shorter than r . From the resulting plot, the commissioner then picks the combination of chunk size c and compression technique t that achieves the shortest total runtime for the query mix.

Default Values. For chunk sizes c from 32 to 1024 KB and for the remaining three compression techniques t , Fig. 6 shows the *total access time* of all the $20 \cdot 96$

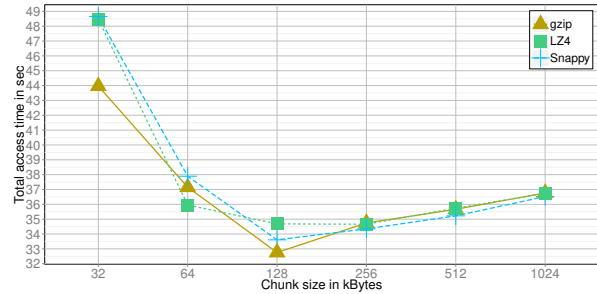


Figure 6: Total access times for $20 \cdot 96$ queries (in sec.).

= 1920 randomly selected data retrievals that represent the query mix. There is a bath tub curve for each of the three compression techniques t , i.e., there is a chunk size c that results in a minimal total access time. As t =gzip achieves the absolute minimum with a chunk size of c =128 KB, Chronix selects this sweet spot, especially as the other options do not show better materialized compression rates for that chunk size (see Fig. 4). The default parameters are good-natured, i.e., small variations do not affect the results much. Fig. 6 also shows the effect of suboptimal choices for c and t .

Commissioning and re-commissioning for a use-case specific set of parameters are possible but costly as the latter affects the whole data. At the end of the next section, we discuss the effect of a use-case specific commissioning compared to the default parameter set.

5 Evaluation

We quantitatively compare the memory footprint, the storage efficiency, and the query performance of Chronix (without any optional transformation and without any pre-computed analysis results) to InfluxDB, OpenTSDB, and KairosDB. After a sketch of the setup, we describe two case-studies whose operational data serve as benchmark. Then we discuss the results and demonstrate that Chronix' default parameters are sound.

Setup. For the minimal realistic installation all databases run on one computer and the Java process issuing the queries via HTTP runs on a second computer (for hardware details see Section 4). InfluxDB, OpenTSDB, and KairosDB store time series with different underlying storage technologies: InfluxDB (v.1.0.0) uses a custom storage called time structured merge tree, OpenTSDB (v.2.2.0) uses the wide-column store HBase [23] that stores data in the Hadoop Distributed File System (HDFS) [41], and KairosDB (v.1.1.2) stores the data in the wide-column store Cassandra [3]. They also have different strategies for storing the attributes: InfluxDB stores them once with references to the data points they belong to. OpenTSDB and KairosDB store

Table 4: Project Statistics.

Project		4	5	total
time series		500	24,055	24,555
pairs (mio)	metric	3.9	3,762.3	3,766.2
	lsof	0.4	0.0	0.4
	strace	12.1	0.0	12.1

(a) Pairs and time series per project.

r	0.5	1	7	14	21	28	56	91	180	
q	2	11	15	8	12	5	1	2	2	58
b	1	6	5	7	2	4	4	1	2	32
h	2	6	10	8	6	6	3	2	0	43

(b) Time ranges r (days); # of raw data queries (q), of queries with basic (b) and high-level (h) functions.

them in key-value pairs that are part of the row-key. We use default configurations for each database. To measure the query runtimes and the resource consumptions each database is called separately and runs it in its own Docker container [16] to ensure non-interference. We batch-import the data once and run the queries afterwards. We do not discuss the import times as they are less important for the analysis.

Case-studies/benchmark. We collected 108 GByte of operational data from two real-world applications. In contrast to the dataset used in the commissioning there are also data of `lsof` and `strace`, see Table 4(a).

Project 4 detects anomalies in a service application for modern cars (such as music streaming). The goal is to locate the root cause of a file handle leak that forces the application to nightly rebootings of its two application servers. The collected operational data covers 3 hours with an almost-periodic interval of 1 second.

Since we have mentioned an example query of this project in Section 3 and since we have shown how the `grpsize` function (Listing 2) can be plugged-in, let us give some more details on this project.

The initial situation was that the application kept opening new file handles without closing old ones. After reproducing the anomaly in a test environment, for the explorative analysis we employed `lsof` to show the open files, and stored this operational data in Chronix. The results of queries like

```
q=type:lsof & cf=lsof{grpsize:name,*}
```

were the key to explain the rise in the number of open file handles as about 2,000 new file handles were pipes or `anon_inodes` that are part of the `java.nio` package. Hence it was necessary to dig deeper and to link file handles to system calls. To do so, we used `strace` and also stored the data in Chronix. By narrowing down the time series data to individual file handle IDs, with correlating queries like the one shown in Section 3 we found

```
1 //End of strace for file handle ID = 2030
2 epoll_ctl(2129, EPOLL_CTL_ADD, 2030,
3     {EPOLLIN, {u32=2030, u64=2030}}) = 0
4 //End of strace for file handle ID = 2032
5 epoll_ctl(2032, EPOLL_CTL_ADD, 1889,
6     {EPOLLIN, {u32=1889, u64=1889}}) = 0
```

Listing 4: Last `strace` calls for two file handle IDs.

Table 5: Memory footprint in MBytes.

	InfluxDB	OpenTSDB	KairosDB	Chronix
Initially	33	2,726	8,763	446
Import (max)	10,336	10,111	18,905	7,002
Query (max)	8,269	9,712	11,230	4,792

that `epoll_ctl` was often the last function call before the anomaly (see Listing 4). By then analyzing which third party libraries the application uses and by gathering information on `epoll_ctl` we deduced that the application used an old version of Grizzly that leaks selectors [1] when it tries to write to an already closed channel. The solution was to upgrade the affected library.

Project 5 detects anomalies in an application that manages the compatibility of software components in a vehicle. The production system has a central database and six application servers. The operational data is analyzed to find and understand production anomalies, such as long running method calls, positive trends of resource consumption, etc. The dataset has an almost-periodic interval of 1 min. and holds seven months of operational data.

Table 4(b) shows the mix of the 133 queries that the projects needed to achieve their analysis goals. There are different ranges (r) for the 58 raw data retrievals (q) that do not have a `cf`-part and also for the 32 queries that use basic analysis functions (b-queries) and for the 43 h-queries that use high-level analyses. Table 8 lists which of the built-in analysis functions from Table 2 the projects actually use (and how often).

Memory Footprint. Table 5 shows the memory footprint of all database *processes* at different times. This is relevant as analyses on large amounts of time series data are often memory intensive. The first line shows the memory consumption just after a container's start, when all components of the time series database are up and running. The next two lines show the maximal memory footprints that we encountered while the data of the two benchmark projects was imported and while the query mix was executed. All databases stay below the maximal available memory of 31.42 GB. The import (buffering, index construction, etc.) needs more memory than the query mix (reading, decompression, serialization, ship-

Table 6: Storage demands of the data in GBytes.

Project	Raw data	InfluxDB	OpenTSDB	KairosDB	Chronix
4	1.2	0.2	0.2	0.3	0.1
5	107.0	10.7	16.9	26.5	8.6
total	108.2	10.9	17.1	26.8	8.7

ping, etc.). OpenTSDB and KairosDB clearly take the most memory due to their various underlying components. InfluxDB is better but still takes 1.5 times more memory than Chronix. The reasons for Chronix' lower memory usage are: (a) it does not hold large amounts of data in memory at once, (b) it runs as a single process with lightweight thread parallelism, and (c) its implementation avoids expensive object allocations.

Storage Efficiency. Table 6 shows the storage demands of the data, including the index files, commit logs, etc. There are three aspects to note. First, out-of-the-box none of the general-purpose databases can handle the `lsnf` and `strace` data. Extra programming was needed to make these databases utilizable for the case-studies. (For both OpenTSDB and InfluxDB we had to encode the non-numeric data in tags or in timestamps plus strings, including some escape mechanisms for special characters. For KairosDB we had to explicitly implement and add custom types.) Second, both OpenTSDB and KairosDB cannot handle the nanosecond precision of the `strace` data. We chose to let them store imprecise data instead, because (explicitly) converted timestamps would have taken even more space. Third, all measurements are done after the optimizations and compactions. During the import the databases temporarily take more disk space (e.g., for commit logs etc.).

Chronix only needs 8.7 GBytes to store the 108.2 GBytes of raw time series data. Compared to the general-purpose time series databases Chronix saves 20%–68% of the storage demand. This is caused by Chronix' domain specific optimizations and by differences in the underlying storage technologies. By default, OpenTSDB does not compress data, but for a fair comparison we used it with `gzip`. InfluxDB stores rows of single data points using various serialization techniques, such as Pelkonen et al. [34] for numeric data. KairosDB uses LZ4 that has a lower compression rate.

Data Retrieval Performance. The case-studies have 58 raw data queries (q) in their mix, with various time ranges (r). Table 7 gives the retrieval times. They include the time to find, load, and ship the data and the time to deserialize it on the side of client. For the measurements, the data retrieval mix is again repeated 20 times to stabilize the results, with q randomly picked time ranges r.

Table 7: Data retrieval times for 20 · 58 queries (in s).

r	q	InfluxDB	OpenTSDB	KairosDB	Chronix
0.5	2	4.3	2.8	4.4	0.9
1	11	5.2	5.6	6.6	5.3
7	15	34.1	17.4	26.8	7.0
14	8	36.2	14.2	25.5	4.0
21	12	76.5	29.8	55.0	6.0
28	5	7.9	3.9	5.6	0.5
56	1	35.4	12.4	24.1	1.2
91	2	47.5	15.5	33.8	1.1
180	2	96.7	36.7	66.6	1.1
total		343.8	138.3	248.4	27.1

Table 8: Times for 20 · 75 b- and h-queries (in s).

Basic (b)		InfluxDB	OpenTSDB	KairosDB	Chronix
4	avg	0.9	6.1	9.8	4.4
5	max	1.3	8.4	9.1	6.0
3	min	0.7	2.7	5.3	2.8
3	stddev.	6.7	16.7	21.1	2.3
5	sum	0.7	6.0	12.0	2.0
4	count	0.8	5.5	10.5	1.0
8	perc.	10.2	25.8	34.5	8.6
High-level (h)					
12	outlier	30.7	29.1	117.6	18.9
14	trend	162.7	50.4	100.6	30.2
11	frequency	47.3	23.9	45.7	16.3
3	grpsize	218.9	2927.8	206.3	29.6
3	split	123.1	2893.9	47.9	37.2
75	total	604.0	5996.3	620.4	159.3

InfluxDB is the slowest, followed by KairosDB and OpenTSDB. Chronix is the fastest and saves 80%–92% of the time needed for the raw data retrieval. For all databases the retrieval times grow with larger ranges. But for Chronix, they grow more slowly. There are several reasons for this: (a) Chronix uses an index to access the chunks and hence avoids full scans, (b) its pipeline architecture ships the raw chunks to the client that can process (decompress, deserialize) them in parallel, and (c) Chronix selects its chunk size and compression to suit these queries.

Built-in Function Advantages. In addition to raw data retrieval, anomaly detection in operational data also needs analyzing functions, several of which the general-purpose time series databases do not natively support (see Table 2) and whose functionality has to be implemented by hand and typically with more than one query.

Table 8 shows the runtimes (20 repetitions for stabilization) that use basic (b) and high-level (h) functions and how often the projects use them (first column). In total, Chronix saves 73%–97% of the time needed by the general-purpose time series databases. We discuss the results for queries with basic functions (b-queries) and with high-level functions (h-queries) in turn.

In total, the 32 b-queries that other time series databases also natively support account for not more than about 17% of the total runtime. Thus, speed variations for b-queries do not matter that much for anomaly detection tasks. Nevertheless, let us delve into the upper part of Table 8. OpenTSDB and KairosDB are often slower than InfluxDB or Chronix. Whenever InfluxDB can use its pre-computed values (for instance for average, maximum, etc.) it outperforms Chronix. When on-the-fly computations are needed (deviation and percentile), Chronix is faster.

The lower part of Table 8 illustrates the runtimes of the 43 h-queries. They are important for the anomaly detection projects as they are used much more often than the other functions. Here Chronix has a much more pronounced lead over the general-purpose databases. The reason is that Chronix offers built-in means to evaluate these functions server-side, whereas they have to be manually implemented on the side of the client in the other systems, with additional raw data retrievals.

Let us look closer at the penalties for the lack of such built-in functions. To implement an *outlier* detector in the other systems, one has to calculate the threshold value as $(Q3 - Q1) \cdot 1.5 + Q3$ where Q1 is the first and Q3 is the third quartile of the time series values. With InfluxDB this needs one extra query. KairosDB needs two extra queries, one for getting Q1 and one for Q3, plus a client-side filter. OpenTSDB does neither provide a function for getting Q1 nor for filtering values. In the other systems a *trend* detector (that checks if the slope of a linear regression through the data is above or below a threshold) has to be built completely on the side of the client. A *frequency* detector (that splits a time series into windows, counts the data points, and checks if the delta between two windows is above a predefined threshold) is more costly to express and to run in the other systems as well. InfluxDB needs one extra query and a client-side validation. OpenTSDB and KairosDB need a query plus code for an extra function on the side of the client. The *grp-size* and the *split* functions that run through this paper are crucial for project 4 both have to be implemented on the side of the client with an extra query for raw values.

Although it was possible to emulate the high-level functions, we ran into problems that are either caused by the missing support of nanosecond timestamps (KairosDB and OpenTSDB) or the string encoding (*lsof/strace*) in tags (OpenTSDB). Missing precision

causes the *split* function to construct wrong results – we ignored this and measured the times nevertheless. String decoding and serialization simply took too long, so we measured the time of the raw data retrieval only.

The online distribution of Chronix holds the code and also the re-implementation of the queries with other time series databases.

Extra queries and client-side evaluations cause a significant slowdown. This can be seen in the lower part of Table 8 where Chronix is faster. But this effect is also visible in the b-queries. For instance, InfluxDB needs $343.8 \text{ s} / (58 \cdot 20) = 0.3 \text{ s}$ on average for a raw data query without evaluating any function at all. Its average for the $43 \cdot 20$ b-queries instead is only 0.03 s because the function is evaluated server-side and only the result is shipped. This is similar for the other databases. Built-in functions are therefore a clear advantage.

Default values of Chronix. All the results show that even with its default parameters Chronix outperforms the general-purpose time series databases on anomaly detection projects. A use-case specific commissioning with both projects’ datasets as input did not change the values for *c* and *t* and did not buy any extra performance.

For the default DDC threshold of $d=200 \text{ ms}$ we see an inaccuracy rate of 20% for both projects. The average deviations are around 42 ms and 80 ms, resp. From our experience, this is acceptable. With the DDC threshold set to the period of the almost-periodic time series, inaccuracy reaches the worst case as only the first timestamp is stored. But even then the resulting materialized compression rate would only be 1.1% lower but for the costs of a high inaccuracy rate.

6 Related Work

We discuss related work along the main requirements of Sec. 2 and the domain specific design decisions of Sec. 3.

Generic data model. We are not aware of any time series database that has such a generic data model as Chronix. Often only scalar values are supported [2, 12, 13, 28, 31, 32, 34, 36]. InfluxDB [24] has also strings and boolean types. KairosDB [25] is extensible but the types lack support for custom operators and functions. As discussed in Section 2, this is too restrictive for the operational data of software systems.

Analysis support. There are indexing approaches for an efficient analysis and retrieval of time series data, e.g., approximation techniques and tree-structured indexes [6, 8, 15, 26]. They optimize an approximate representation of the time series with pointers to the files that contain the raw data for example for a similarity search. In contrast, Chronix is not tailored to a specific analysis but it is optimized for explorative and correlating analyses of operational time series data. Note that the Optional

Transformation stage *can* add indexing values.

Several researchers presented methods that detect anomalies in time series [43, 45, 46]. Chronix implements in its API most of them and also offers plug-ins.

Efficient long term storage. While many time series databases focus on cloud computing and PBytes of general time series data, Chronix is domain specific for the anomaly detection in operational data. Chronix builds upon and extends the architectural principles proposed by Shafer et al. [40] and Dunning et al. [17]. Its strengths and the reasons for the better performance are its pipeline architecture with domain specific optimizations and the commissioning methodology.

Many time series databases are distributed systems that run in separate processes [24, 25, 28, 32] on multiple nodes. Some are affected by synchronization costs and inter-process communication even when configured to run on a single node [25, 28, 32]. In contrast, on a single node Chronix uses lightweight thread parallelism within a process. Moreover, since OpenTSDB and KairosDB use an external storage technology (HBase [23] and Cassandra [31]) with a built-in compression, they cannot save time by shipping compressed data to/from the analysis client [22], whereas Chronix uses the chunk size and the compression technique that not only achieve the best results on the data but also cut down on query latency. OpenTSDB and KairosDB have a large memory footprint due to their building blocks.

In-memory databases or databases that keep large parts of the data in memory, like Gorilla [34] or BTrDB [2], can quickly answer queries on recent data, but they need a long term storage like Chronix for the older data that some anomaly detectors need.

Lossless storage. A few of the related time series databases are not lossless. RRDtool [31], Ganglia [29] (that uses RRDtool), and Graphite [12] store data points in a fixed-size cyclic buffer, called Round Robin Archive (RRA). If there is more data than fits into the buffer, they purge old data. This may cause wrong analysis results. For identification purposes Prometheus [36] uses a 64-bit fingerprint of the attribute values to find data. Potential hash collisions of fingerprints may cause missed data.

Focus on Queries. While most of the databases are optimized for write throughput [2, 12, 13, 25, 31, 32, 34] and suffer from scan and filter operations when data is requested, Chronix optimizes the query performance (mainly by means of a reverse index). This is an advantage for the needs of anomaly detection. Chronix delays costly index reconstructions when a batch import of many small chunks of data is needed.

Chronix processes raw data for aggregations. To optimize such aggregates Chronix can be enriched with techniques of related time series databases to store pre-aggregated values [2, 24].

Domain specific compression. Most time series databases use some form of compression. There are lossy approaches that do not fit the requirements of anomaly detection in operational data. (For instance, one idea is to down-sample and override old data [12, 31].) Many time series databases [2, 25, 28, 32, 34, 36] use a loss-less compression. (TsdB [13] applies QuickLZ [37] and only achieves a mediocre rate of about 20% [13].) Most of them also use a value encoding that is similar in spirit to the DDC. The difference is that Chronix only removes jitter from the timestamps in almost-periodic intervals as exact values matter for anomaly detection.

Commissioning. For none of the other time series databases there is a commissioning methodology to tune it to the domain specific or even use-case specific needs of anomaly detection in operational data.

7 Conclusion

The paper illustrates that general-purpose time series databases impede anomaly detection projects that analyze operational data. Chronix is a domain specific alternative that exploits the specific characteristics of the domain in many ways, e.g., with a generic data model, extensible data types and operators, built-in high-level functions, a novel encoding of almost-periodic time series, records with attributes and binary-encoded chunks of data, domain specific chunk sizes, etc. Since the configuration parameters need to be chosen carefully to achieve an ideal performance, there is also a commissioning methodology to find values for them.

On real-world operational data from industry and on queries that analyze these data, Chronix outperforms general-purpose time series databases by far. With a smaller memory footprint, it saves 20%–68% of the storage space, and it saves 80%–92% on data retrieval time and 73%–97% of the runtime of analyzing functions.

Chronix is open source, see www.chronix.io.

Acknowledgements

This research was in part supported by the *Bavarian Ministry of Economic Affairs and Media, Energy and Technology* as an IuK-grant for the project *DfD – Design for Diagnosability*.

We are grateful to the reviewers for their time, effort, and constructive input.

References

All online resources accessed on Sep. 22, 2016.

- [1] <https://java.net/jira/browse/GRIZZLY-1690>.
- [2] ANDERSEN, M. P., AND CULLER, D. E. BTrDB: Optimizing Storage System Design for Timeseries Processing. In *USENIX Conf. File and Storage Techn. (FAST)* (Santa Clara, CA, 2016), pp. 39–52.
- [3] APACHE CASSANDRA. Manage massive amounts of data. <http://cassandra.apache.org>.
- [4] APACHE LUCENCE. A full-featured text search engine. <http://lucene.apache.org>.
- [5] APACHE SOLR. Open source enterprise search platform. <http://lucene.apache.org/solr>.
- [6] ASSENT, I., KRIEGER, R., AFSCHARI, F., AND SEIDL, T. The TS-tree: Efficient Time Series Search and Retrieval. In *Intl. Conf. Extending database technology: Advances in database technology* (Nantes, France, 2008), pp. 252–263.
- [7] BURROWS, M., AND WHEELER, D. A Block-sorting Lossless Data Compression Algorithm. Tech. Rep. 124, Systems Research Center, Palo Alto, CA, 1994.
- [8] CAMERRA, A., PALPANAS, T., SHIEH, J., AND KEOGH, E. iSAX 2.0: Indexing and Mining One Billion Time Series. In *Intl. Conf. Data Mining (ICDM)* (Sydney, Australia, 2010), pp. 58–67.
- [9] CHAN, K.-P., AND FU, A. W.-C. Efficient Time Series Matching by Wavelets. In *Intl. Conf. Data Eng.* (Sydney, Australia, 1999), pp. 126–133.
- [10] COLLECTD. The system statistics collection daemon. <https://collectd.org>.
- [11] COLLET, Y. LZ4 is lossless compression algorithm. <http://www.lz4.org>.
- [12] DAVIS, C. Graphite project – a highly scalable real-time graphing system. <https://github.com/graphite-project>.
- [13] DERI, L., MAINARDI, S., AND FUSCO, F. tsdb: A Compressed Database for Time Series. In *Intl. Conf. Traffic Monitoring and Analysis* (Vienna, Austria, 2012), pp. 143–156.
- [14] DEUTSCH, P. GZIP file format specification version 4.3. <https://www.ietf.org/rfc/rfc1952.txt>, 1996.
- [15] DING, H., TRAJCEVSKI, G., SCHEUERMANN, P., WANG, X., AND KEOGH, E. Querying and Mining of Time Series Data: Experimental Comparison of Representations and Distance Measures. In *Very Large Databases Endowment* (Auckland, New Zealand, 2008), pp. 1542–1552.
- [16] DOCKER. Docker is the leading software containerization platform. <https://www.docker.com>.
- [17] DUNNING, T., AND FRIEDMAN, E. *Time Series Databases: New Ways to Store and Access Data*. O’Reilly Media, 2014.
- [18] ELASTIC. Logstash: Collect, Parse, Transform Logs. <https://www.elastic.co/products/logstash>.
- [19] GOOGLE. Guice. <https://github.com/google/guice>.
- [20] GOOGLE. Protocol buffers – google’s data interchange format. <https://github.com/google/protobuf>.
- [21] GOOGLE. Snappy. <http://google.github.io/snappy>.
- [22] GRAEFE, G., AND SHAPIRO, L. Data Compression and Database Performance. In *Symp. Applied Computing* (Kansas City, MO, 1991), pp. 22–27.
- [23] HBASE, A. The hadoop database, a distributed, scalable, big data store. <http://hbase.apache.org>.
- [24] INFLUXDATA. InfluxDB: Time-Series Storage. <https://influxdata.com/time-series-platform/influxdb>.
- [25] KAIROSDB. Fast Time Series Database. <https://kairosdb.github.io/>.
- [26] KEOGH, E., CHAKRABARTI, K., PAZZANI, M., AND MEHROTRA, S. Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases. *Knowledge and information Systems* 3, 3 (2001), 263–286.
- [27] LAPTEV, N., AMIZADEH, S., AND FLINT, I. Generic and scalable framework for automated time-series anomaly detection. In *Intl. Conf. Knowledge Discovery and Data Mining (KDD)* (Sydney, Australia, 2015), pp. 1939–1947.
- [28] LOBOZ, C., SMYL, S., AND NATH, S. DataGarage: Warehousing Massive Amounts of Performance Data on Commodity Servers. In *Very Large Databases Endowment* (Singapore, 2010), pp. 1447–1458.
- [29] MASSIE, M. L., CHUN, B. N., AND CULLER, D. E. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing* 30, 7 (2004), 817–840.

- [30] MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., AND VASSILAKIS, T. Dremel: Interactive Analysis of Web-Scale Datasets. In *Very Large Databases Endowment* (Singapore, 2010), pp. 330–339.
- [31] OETIKER, T. RRDtool: Data logging and graphing system for time series data. <http://oss.oetiker.ch/rrdtool>.
- [32] OPENTSDDB. The Scalable Time Series Database. <http://opentsdb.net>.
- [33] PATEL, P., KEOGH, E., LIN, J., AND LONARDI, S. Mining Motifs in Massive Time Series Databases. In *Intl. Conf. Data Mining* (Maebashi City, Japan, 2002), pp. 370–377.
- [34] PELKONEN, T., FRANKLIN, S., TELLER, J., CAVALLARO, P., HUANG, Q., MEZA, J., AND VEERARAGHAVAN, K. Gorilla: A Fast, Scalable, In-Memory Time Series Database. In *Conf. Very Large Databases* (Kohala Coast, HI, 2015), pp. 1816–1827.
- [35] POPIVANOV, I., AND MILLER, R. Similarity Search Over Time-Series Data Using Wavelets. In *Intl. Conf. Data Eng.* (San Jose, CA, 2002), pp. 212–221.
- [36] PROMETHEUS. Monitoring system and time series database. <http://prometheus.io>.
- [37] QUICKLZ. Fast compression library. <http://www.quicklz.com>.
- [38] SALVADOR, S., AND CHAN, P. FastDTW: Toward Accurate Dynamic Time Warping in Linear Time and Space. *Intelligent Data Analysis* 11, 5 (2007), 561–580.
- [39] SEWARD, J. bzip2. <http://www.bzip.org>.
- [40] SHAFER, I., SAMBASIVAN, R., ROWE, A., AND GANGER, G. Specialized Storage for Big Numeric Time Series. In *USENIX Conf. Hot Topics Storage and File Systems* (San Jose, CA, 2013).
- [41] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop Distributed File System. In *Symp. Mass Storage Systems and Technologies* (Lake Tahoe, NV, 2010), pp. 1–10.
- [42] TUKAANI. Xz. <http://tukaani.org/xz>.
- [43] VALLIS, O., HOCHENBAUM, J., AND KEJARIWAL, A. A Novel Technique for Long-Term Anomaly Detection in the Cloud. In *USENIX Conf. Hot Topics Cloud Computing* (Philadelphia, PA, 2014).
- [44] VAN HOORN, A., WALLER, J., AND HASSELBRING, W. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Intl. Conf. Perf. Eng. (ICPE)* (Boston, MA, 2012), pp. 247–248.
- [45] WANG, C., VISWANATHAN, K., CHOUDUR, L., TALWAR, V., SATTERFIELD, W., AND SCHWAN, K. Statistical Techniques for Online Anomaly Detection in Data Centers. In *Intl. Symp. Integrated Network Management (IM)* (Dublin, Ireland, 2011), pp. 385–392.
- [46] WERT, A., HAPPE, J., AND HAPPE, L. Supporting Swift Reaction: Automatically Uncovering Performance Problems by Systematic Experiments. In *Intl. Conf. Soft. Eng. (ICSE)* (San Francisco, CA, 2013), pp. 552–561.
- [47] XU, W., HUANG, L., FOX, A., PATTERSON, D. A., AND JORDAN, M. I. Mining console logs for large-scale system problem detection. In *Conf. Tackling Computer Systems Problems with Machine Learning Techniques* (San Diego, CA, 2008).