tsdb: A Compressed Database for Time Series

Luca Deri[1,2], Simone Mainardi[1,3], and Francesco Fusco[4]

[1] Institute of Informatics and Telematics, CNR, Pisa, Italy
[2] ntop, Pisa, Italy
[3] Department of Information Engineering, University of Pisa, Pisa, Italy
[4] IBM Zürich Research Laboratory, Rüschlikon, Switzerland
{luca.deri,simone.mainardi}@iit.cnr.it
ffu@zurich.ibm.com

Abstract. Large-scale network monitoring systems require efficient storage and consolidation of measurement data. Relational databases and popular tools such as the Round-Robin Database show their limitations when handling a large number of time series. This is because data access time greatly increases with the cardinality of data and number of measurements. The result is that monitoring systems are forced to store very few metrics at low frequency in order to grant data access within acceptable time boundaries.

This paper describes a novel compressed time series database named tsdb whose goal is to allow large time series to be stored and consolidated in real-time with limited disk space usage. The validation has demonstrated the advantage of tsdb over traditional approaches, and has shown that tsdb is suitable for handling a large number of time series.

Keywords: Large-scale datasets, time series, web-based data visualization.

1 Introduction and Motivation

The demand of (near) real-time monitoring as well as the analysis of high-speed networks has put several constraints on monitoring systems. Users are demanding solutions able to interactively drill-down data while simultaneously collecting (hundred of) thousand metrics from various network sensors. In order to increase measurement accuracy, network administrators often reduce the sampling frequency of counters and gauges. If some years ago, a sampling period of 5 minutes was acceptable, nowadays network administrators require higher frequency samples for detecting anomalies that would not be detected by monitoring the same data at lower frequencies. For instance, detection of traffic spikes and microbursts require tenth (if not hundred) samples per second. The consequence of this trend is that monitoring systems produce an ever-increasing amount of data that needs to be stored and analyzed in a limited amount of time.

With the advent of multi-Gbit networks, the traffic being analyzed and the corresponding number of measured metrics increased significantly. Periodically accounting host traffic for a /24 subnet is very different from performing the same activity on a network backbone. In the latter case, monitoring systems cannot tolerate delays while saving/reading data from/to the disk, as data access slow-down would prevent the system from carrying on the tasks within the expected timeframe.

As discussed in the following section, both relational databases and specialized tools such as the rrdtool [3], are used in the industry for handling *time series*. A time series is a sequence of data points measured at uniform time intervals (e.g. every 5 minutes) [16]. Unfortunately both solutions are only capable of satisfying requirements coming from small to medium environments, where the number of monitored metrics does not exceed a few tenth of thousand. However, collecting a much higher number of time series is not uncommon these days. Even a simple ntop installation [4] deployed for monitoring a medium network, has to keep track of several tens of thousand metrics just for keeping a few counters (e.g. bytes/packets sent/received). If additional counters are measured (e.g. traffic per protocol and network), the dataset size can quickly increase. According to the tests we carried on, existing open-source solutions and relational databases are affected by serious scalability issues when collecting hundreds of thousands (millions) metrics, making them *practically unusable* for large monitoring systems. In fact, without scalable and efficient time series handling tools, monitoring systems cannot store data at fine-grained granularity and are forced to decrease the monitoring accuracy. The lack of open-source tools for an efficient handling of time series has been the main motivation for creating a new open-source time series database for time series called *tsdb*.

The rest of the paper is organized as follows. Section 2 analyzes the various alternatives for handling time series. Section 3 presents the design and implementation of tsdb. Section 4 covers the tsdb validation and compares it with similar tools. Section 5 highlights some open issues and future work items.

2 Background and Related Work

2.1 Database Systems

For years network developers have used relational databases for storing network data persistently. Data with uniform characteristics are organized in tables linked by relationships. Each table is logically divided in columns and rows, where a row is uniquely identified by means of a primary key. Data stored on the database can be modified and deleted. Unfortunately network data is not characterized by many relationships, it is usually unchangeable (i.e. changing measured data might indicate a counterfeit), and the same data is repeated over time at every measurement interval, making relational databases not convenient for handling this type of data.

The reasons are manifold:

- At every measurement interval, tables are populated with fresh data that increases table cardinality. The consequence is that table cardinality as well the space taken on disk increases with the number of measurements.
- As soon as table indexes become large enough to prevent themselves to be cached, data retrieval becomes significantly slow [21, 14] thus jeopardizing the performance of applications sitting on top of the database.

A partial solution for avoiding these slowdowns is the use of binary large objects (BLOB) for storing time series. In this case the drawback is that the database is unable to search directly on blobs. As it can just read/store raw data, it must delegate to third party applications the implementation of such retrieval, and data management facilities. In order to address these issues with relational databases, time series database servers (TSDS) [1] have been created. They have been designed for enabling efficient data retrieval within some defined date/time ranges, as well for handling date and timezone conversions. Unfortunately TSDS are mostly used in the industry, and the only open-source alternative OpenTSDB [2], has a pretty complex architecture with several components interacting over a network, making it suitable only for distributed systems.

2.2 Round Robin Database

The Round-Robin Database (RRD) is a great alternative to relational databases for storing time series. It implements a file-based persistent circular buffer where data is stored according to its timestamp. When the database is created it is necessary to specify the data lifetime as well the frequency (named step in the rrd parlance) at which data is stored. For instance, it is possible to store a value every 5 minutes for at most 30 days. As all the information is specified at database creation, rrd files do not grow over time: their size is static and as large as the circular buffer. Each rrd database can store multiple time series, not necessarily all sharing the same lifetime and frequency parameters.

Typically rrd databases are small in size (64 KB or less) and stored as files on disk. Database files can be manipulated using a command-line tool named *rrdtool*, with no network access (e.g. via SQL-like query languages) typical of relational databases. Both rrdtool and its companion *librrdtool* library have been designed as tools to be accessed from the command line. Therefore, everything is file centric. Each database manipulation requires the library to open, manipulate, and save the file. If multiple operations have be performed on the same rrd file, the library needs to open/save/close the file multiple times. Another limitation is that most of the library functions require parameters to be specified in the argc/argv format, as required by the main() C function. The consequence is that due to these design shortcomings, it is not practically possible to define rrd databases with hundreds or thousands time series, thus limiting each rrd file to a few series.

This means that:

- The order of magnitude of rrd database files on disk is the same as the number of time series we need to handle.
- When we need to manipulate time series, we have to open/save/close as many files as the number of time series we plan to manipulate.

Although the rrdtool uses *mmap()* system call to reduce the number of read/write operations when updating an rrd or extracting a time series, the fact that rrd relies on the filesystem underneath is a matter of fact. Databases implement their internal data indexing and housekeeping, whereas in the rrd world this is delegated to the filesystem. Table 1 shows how rrdtool behaves while creating and updating a simple rrd archive (Ubuntu Linux 11.04 64 bit, Intel i7 860, SATA 3 Gb/s, ext4 filesystem). Note that in the RRD ecosystem, as the database is created as a circular buffer able to contain all defined time series, we use the term update (i.e. replace, if any, the previous value in the circular buffer) rather than the term append that is more appropriate for relational databases.

Table 1. rrdtool performance while creating and update rrd's

Number of RRDs	RRD Creation (Total)	RRD Update (Total)
10	0.53 sec	0.24 sec
100	5.34 sec	2.76 sec
1,000	58.13 sec	53.97 sec
10,000	600.74 sec	467.89 sec

The time spent per RRD is almost constant across all runs, with a limited increase with the number of files, probably due to disk management. We also performed additional tests using RAID, RAM disks, and various file systems (i.e. ext3, XFS, Oracle brtfs). Some setups reported better results with respect to Table 1, however the overall performance did not change significantly. This is because each rrd operation takes a few milliseconds; even if minimized, it needs to be multiplied for all rrds we plan to manipulate. This boils down to the conclusion that rrd is not able to manipulate a large number of time series within limited time boundaries. For example on our test system we have not been able to update more that 64k rrd's within 5 minutes time, although in our system no other application was accessing the disk and using CPU cycles.

The reasons for this behavior are manifold:

- RRD is file oriented, thus its performance cannot exceed the performance of the filesystem and disk it relies on.
- Each rrd file contains no more than a few time series, so at each time step all rrd databases need to be updated.

The rrd community has realized this limitation, and therefore on large installations a caching RRD daemon named *rrdcached* is used to cache updates in memory and perform them periodically. Even if this solution can reduce the number of rrd updates, it does not decrease the cost of per-rrd manipulation. Therefore, manipulating hundred of thousands (millions) rrds requires hours, making RRD unsuitable for effectively monitoring a large number of time series.

2.3 Additional Time Series Database and Tools

In addition to relational databases and rrdtool, there are other tools designed for handling time series such as TelegraphCQ [17], STATStream [18], and iSAX [19].

These tools had a large impact on research, but some of them are not maintained anymore, and others are just software prototypes used to validate the research work.

2.4 Numeric Databases and Compression

Compressing time series data can be fundamental not only for the obvious storage size reduction but also for improving performance. This is because less data needs to be read/written on disk. Suppose your data is stored in a block device (e.g. hard disk), reducing the data size by a factor of two, only half of the blocks will be read/written with respect to the same system when not using compression. The drawback is that compression has a cost in terms of time that is negligible if compared to the time required for moving mechanical parts (e.g. heads) of a hard disk.

Time series compression has applications in many network monitoring contexts where thousands if not million of monitoring metrics have to be collected at high frequency. The strategy to be chosen for compressing time series highly depends on the context. First of all, there are domains where a lossless compression is mandatory and others where an approximated representation of the time series is sufficient (lossy compression). The most widely used lossy time series compressors are based on concepts coming from signal theory and exploit specific properties, such as seasonality, to approximate the discrete signal represented by the time series of values. The rationale of these methods is to represent the signal in the frequency domain instead of the time domain to capture important signal properties such as periodicity. By capturing predominant patterns, lossy time series compression techniques enable operations such as nearest neighbor searches, and pattern searches [5] directly in the compressed domain, that are not possible with lossless compression techniques unless additional indexes are used. The main drawback of lossy compression techniques is that they rely on specific patterns for providing a good approximation of the given time series. This is the main reason why lossy compression has been rarely applied to network monitoring contexts, where the patterns of time series can drastically change due to anomalous events or to transient networking issues.

Lossless time series compression can be achieved by applying general-purpose lossless compression techniques, such as the popular Lempel-Ziv based compressors [6], or other compression techniques designed for better capturing and exploiting specific data patterns, such as the presence of long sequences of repeated symbols, or geometry distributions [7]. High-speed lossless compression of numerical values has been an active research topic during the last decade and has been driven by high-speed compression requirements coming from columnar databases [8] and information retrieval [9]. The main research focus has been on designing compressors optimized for achieving high compression and decompression speed rather than for achieving compression ratios as close as possible to the optimum. High-compression speeds, and more importantly decompression speeds are desired in all the contexts where the data has to be stored in a compressed form for reducing the volumes, but still frequently accessed for answering queries. If the compressors are able to provide a decompression speed that is higher than the read I/O bandwidth, then the compression is not only beneficial for reducing the volumes, but also, and more importantly for reducing the query response time. Among the family of speed-optimized compressors, Simple9 [9] and PFor [10] stand out for their performance. Both compressors achieve high decompression performance by optimizing the decompression routines for avoiding conditional branches, which are the bottleneck of current and future super-scalar processors. Compared to general-purpose compressors, the performance of high-speed integer compressors, both in terms of compression ratio and decompression speed, is more dependent on the data to be compressed. It has been shown that high-speed variant of LZ based compressors, such as LZO, have to be preferred over high-speed integer compressors, in case of data composed of many distinct numerical values [11].

3 tsdb: Design Goals and Implementation

As stated in the previous sections, both relational databases and tools like rrdtool are suitable for handling a limited number of time series as their performance is not satisfactory when the time series number increases (e.g. 100k or more). This has been the motivation for designing a new type of database named time series data base *(tsdb)* which is able to:

1. Handle millions of time series with minimal append and data extraction time.
2. Perform updates/appends on all the time series, as well as on a subset of them.
3. Add, remove and re-add time series as needed, without having to reconfigure or rearrange its structure.
4. Avoid data consolidation during append as in the RRD database. Thus it is possible to update/modify past data just by appending fresh data to the database, contrary to RRD where past data cannot be modified at all. If necessary, modification of existing data points can be prevented in tsdb.
5. Provide compressed data storage on a single file for all the time series. Backup and synchronization across network storage servers are easier without having to handle millions of files as with RRD database.
6. Support for time-limited series, so that the database will automatically purge data if older than the specified limit, similar to what rrdtool does.
7. Store time series in its native value without any aggregation (e.g. min/max/average) that might lead to loss of accuracy.
8. Support data extraction mechanisms for creating content suitable for web 2.0 applications that might plot data on a dynamic web page.
9. Provide simultaneous accesses to multiple applications, without writers blocking data readers.

Tsdb stores the database on disk and accesses it through a C API. It has been designed as a better RRD, able to handle millions of time series with limited disk space requirements and to feature fast data insert/retrieval for interactive use. For this reason both time series insertion and extraction have strict time requirements. Fast insertions are required to limit the latency between measurements and their consolidation into the database. High-speed retrievals are essential for minimizing the response time when, for example, web-based applications interactively read from the database Table 1 shows how the rrdtool speed limits the update time (e.g. it takes about 90 minutes to update 1 million series). This automatically implies the determination of a lower bound on the time interval between two consecutive measurements. The more time series are handled, the less often measurements can happen. For example, 90 minutes is the minimum time interval between two consecutive measurements in the previous case. Smaller time intervals would end up in determining buffer overflows.

As limiting data append time is critical, it is necessary to arrange time series so that their manipulation does not take too long. Databases and rrdtool arrange data on a per-time-series rather than on a per-measurement basis. This means that each individual rrd database contains all the values of a given series, so that at each measurement interval *all* the rrd database files have to be updated. Conversely, if data were arranged on a per-measurement basis, each update would have required the manipulation of a single rrd database. In a certain sense, rrdtool arranges data on disk in a way that is orthogonal to the more natural per-measurement way. It is worth noting that similar comments can be made for relational databases, where data is arranged per row (i.e. per time series) instead of per column (i.e. per measurement interval).
Changing the way data is arranged, significantly increases the database append speed but it has a negative impact on data retrieval. In fact whenever all the data points of a time series have to be retrieved, it is necessary to crawl across all the measurement intervals that fall within the time range of interest.
In tsdb all the time series stored in the same database share the same time intervals, so that all the series have the same number of data points and begin at the same time. For instance, all the series start October 1st 2011, and their samples are collected every 5 minutes. One one hand, this looks like a limitation with respect to other tools such as rrdtool where each rrd is independent. On the other hand, this is a nice feature for several reasons:

1. As all the series share the same time, comparing and extracting data points is simple and immune to time manipulation issues.
2. In real life, samples are collected at fixed-time intervals (e.g. once an hour) so it does not make sense to have different time points for series stored into the same database.

Having a uniform time across all the series, enables us to simplify the database design. tsdb has been developed incrementally. We started from a simple design, satisfying only a subset of the requirements discussed above and moved towards the current version, which satisfies them all.

In order to minimize the tsdb append time, we decided to arrange data per time interval. In early prototypes tsdb was implemented as a large extensible array, mapped on disk using the mmap() system call, with the time series in the rows and the measured values in the columns. The adoption of memory mapping minimized memory usage while creating a data structure large enough for storing all the data. Adding new measured values is straightforward as we simply need to add rows as in databases. This design guarantees very fast append time thanks to memory mapping and operating system caching. In fact data append basically happens in memory and a limited

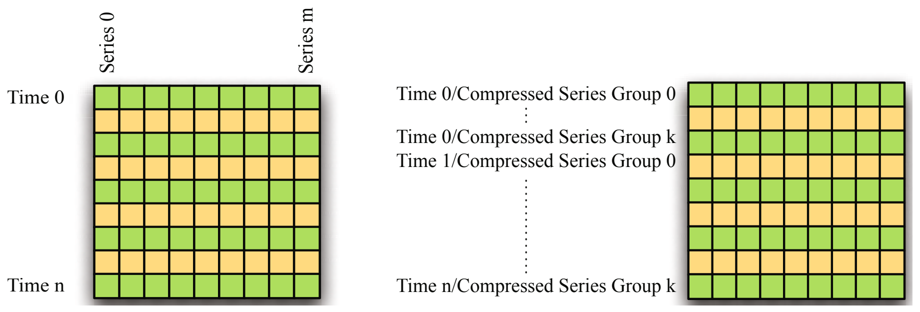
Series 0
Series m
Time 0
Time n
Time 0/Compressed Series Group 0
Time 0/Compressed Series Group k
Time 1/Compressed Series Group 0
Time n/Compressed Series Group k

Fig. 1. (a) Time series arranged in an early tsdb prototype, (b) Compressed time series in the current tsdb implementation

slow-down is experienced only when the cache is flushed to disk. Concurrent operations are straightforward to implement, at least within the same process, as multiple threads see the database as a plain memory area. Inter-thread communication and synchronization can be avoided, unless the same memory locations need to be written by multiple threads. Unfortunately this design has also some drawbacks such as:

1. Data on disk is not compressed and thus its size increases linearly with the number of time series and data points.
2. Once the database is created, it is not possible to add/delete time series without rearranging the whole array.
3. Time series cannot be identified with a symbolic name but only by its index in the array, thus requiring an additional name to the index mapping facility.

In order to overcome these limitations, the current tsdb design is no longer a flat array but it is based on a key/value database used for:

1. Storing database metadata such as inter-measurement time interval duration (e.g. 1 hour), number of stored series and time of the last data update.
2. Associating time series name with a numerical index that is used to uniquely reference it inside the database. This mapping is more than a name-to-index association as it also contains the time ranges where such association is valid. For instance, suppose tsdb assigns index Y to a time series named X, added on January 1st 2011. If on March 1st 2011 X is dropped, the values until that day will be preserved into the database, but the index Y will be made available for new time series that might be later added to the database. If on June 23rd 2011, X is added again, a new index K will be assigned to it. Using the name-to-index mapping, tsdb is able to properly return all the data points values for X, according to the selected time interval, or the value 'undefined' for the intervals where X is undefined.
3. Storing time series according to the time interval. Suppose tsdb has to store values obtained from a measurement made on July 11th 2011 at 15:00. First, it converts the time into Unix epoch (1310389200 in the previous case). Then it creates a database entry identified by such epoch value. Tsdb is also responsible for normalizing the time according to the database configuration as well as for handling timezone conversion (i.e. all epochs are stored in GTM, Greenwich Mean Time).

The current tsdb database implementation is written in C and it is based on BerkeleyDB [13]. During the implementation of tsdb we have also explored other embedded databases such as GNU gdbm, but we have preferred BerkeleyDB as it is much faster when the database has several entries. Furthermore this database is resilient to crashes as it implements advanced techniques for preventing data loss, and it is commercially supported by a well-established database company such as Oracle. Contrary to many file-based databases such as SQLlite and gdbm, multiple clients can concurrently extract data. The tsdb library is implemented in about 1000 lines of code with data points represented as 32-bit integers. When data points for a new time interval are made available, tsdb creates an in memory array with an entry for each time series to be stored. For each time series it writes the corresponding measured value into the array. When the array is populated, tsdb stores it into the database using as key the epoch corresponding to the data. As motivated later in this section, data points for a given epoch are split in chunks of 10k elements and not saved all with the same epoch key. The key named convention we used is X-Y, where X is the epoch, and Y is the chunk id. For instance if we need to save 15k data points for epoch X, we will save the first 10k data points with key X-0, and the remaining 5k points with key X-1. In order to save space, prior to store data points, tsdb compresses and stores them in memory using QuickLZ [11] that we chose for its small memory footprint and high compression speed. As previously explained we did not evaluate the use of lossy compression as we are interested in storing exact values and completely avoid any approximation issues.

In addition to efficient storage of data points, it is essential to provide efficient data retrieval. As tsdb database records contain data points for a given epoch, when accessing data spanning across multiple time intervals, it would be necessary to decompress all the records falling within such interval before extracting the values. In order to minimize the amount of data to be decompressed and thus to reduce the data retrieval time, data points at a given epoch are grouped together and then saved into the database. From our tests, a good compromise between decompression speed and compression ratio is to group data points in chunks not larger than 10k elements. Even with a million of time series spanning one year and dumped at 1-hour intervals, the number of records is 876 k that is not a large number for the tsdb database.

In case most time series need to be analyzed, it is more efficient to convert a tsdb database in a format where all the data points of a time series are contiguous and uncompressed. In order to extract a single time series, we must decompress the chunk containing it for each time interval of interest (see Figure 1b). Assume each chunk contain data points for w time series (w=10k in the previous discussion). If the cost of decompressing a chunk is O(w), we have that extracting all the w times series yields an amortized cost per series which is O(1). We can apply this reasoning also to the whole tsdb database. If the total number of time series in the database is N, you have ceiling(N/w) chunks for each time interval. Since each chunk requires O(w) time to be decompressed, the cost for decompressing a time interval of the whole database is O(N) and consequently the amortized cost per time series is O(1). However, the total cost for decompressing a whole database grows linearly with the number T of time intervals of interest since the previous operations have to be performed T times.

The tsdb database comes with a simple C API for creating applications based on it, as well as companion applications for storing and extracting data from the database, including a utility for extracting time series form tsdb and dumping them into an rrd database. This allows developers to use tsdb as a "faster rrd" while preserving backwards compatibility with rrd that is widely used in the network monitoring community. The following section explains how tsdb has been validated in real life, and compares its performance with similar tools.

4 tsdb Validation

tsdb has been created for effectively monitoring the Italian ".it" DNS registry, where two authors of this paper are currently working. In particular, we want to keep track of DNS queries for each individual ".it" Internet domain. At the time of writing such domains are more than 2.2 million. Before creating tsdb, we have used other databases such as MySQL and also advanced no_SQL [20] key-value databases such as Redis that natively supports the *set* data type, which we used for implementing time series. Both solutions perform reasonably well when handling a limited number of time series, but unfortunately their performance does not scale with the number of time series. For this reasons we have coded a few C test applications for comparing the various solutions. Figure 2 shows typical performance of the tools we have evaluated, when adding a variable number of data points on an empty database. It is worth mentioning that:

- With rrdtool and Redis, the number of database keys corresponds to the number of domains. Whenever new data points are added, the number of keys does not change as data points are added to the list of values associated with each key.
- Due to the nature of relational databases and thus of MySQL, whenever a new data point is added a new record with key <timestamp, key> is created, thus increasing the cardinality of table. This also happens with tsdb but at a much lower pace as records are packed in chunks of 10k records.

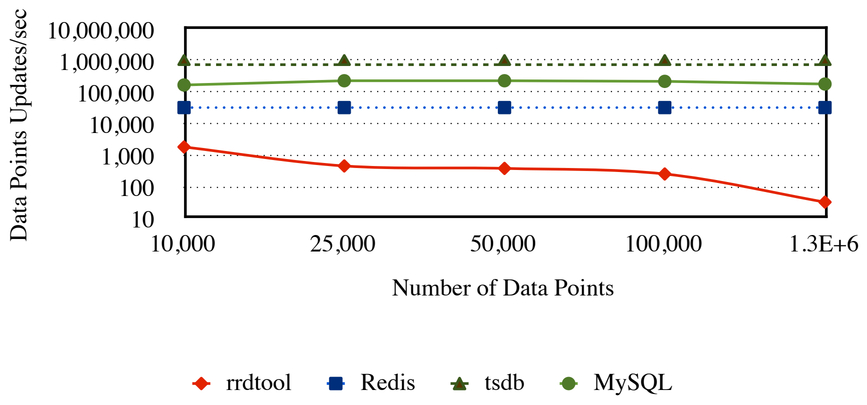

Fig. 2. Time series update processing speed (logarithmic scale)

The rrd performance started at 1600 updates/sec with small databases, and it decreases to 32 updates/sec after processing a few thousand records. We believe that the bottleneck is the disk subsystem, as rrdtool requires open/update/save for each rrd file. Redis performance is acceptable but still not comparable with the tsdb performance. In fact, this database is optimized for in-memory operations, so as long as there is free memory its performance is good. As soon as the available (8 GB) memory is exhausted (by Redis) the performance becomes poor (~ 30 updates/sec) and the system becomes due to disk swap. MySQL append performance of about 200k records/sec is only second to tsdb (that is almost 1M records/sec). Thanks to the use of QuickLZ, data compression ratio of data chunks is 1:5 in average with no noticeable performance degradation with respect to uncompressed data when appending data to tsdb. Storing fewer data has not just space but also performance advantages as previously discussed. For understanding disk space requirements of tsdb, we have created a monitoring application that saves in a tsdb database the number of queries for all ".it" DNS domains (approximately 2.2 million). Data retrieval is efficient, especially if compared with the performance offered by relational databases such as MySQL. To better position tsdb against MySQL, we have created two databases in both formats, containing the time series containing the time series for all .it Internet domains over a period of several months with daily measurements (i.e. one data point per domain per day). As we are monitoring the .it DNS since about 6 months, in order to simulate a larger dataset, we have replicated the data across several months. For tsdb we have used a single-file database for all tests instead of creating a database per year in order to create smarter queries that would only target a specific set of yearly databases. The MySQL table is defined as follows: `CREATE TABLE `domains_summary` ( `day` int(11) NOT NULL, `domain_id` int(10) unsigned NOT NULL, `num_queries` int(11) NOT NULL, UNIQUE KEY `day` (`day`,`domain_id`)) ENGINE=MyISAM DEFAULT CHARSET=latin1`. We have performed the same tests for both tsdb and MySQL with the exception for the largest dataset where only tsdb has been tested. In fact we believe that in this case the MySQL database would be partitioned based on the day, thus queries across several years would basically be as fast as the same query for a single year.

Table 2. tsdb vs. MySQL: (a) Database Size and (b) Search Time

Months of DNS Data	MySQL Database Size/Rows		tsdb Database Size/Keys	
6	8.2 GB	~ 300 M	0.8 GB	~ 41.6 K
18	25.0 GB	~ 900 M	2.6 GB	~ 125.6 K
69	-	-	6.3 GB	~ 497 K

Months of DNS Data	MySQL Search Time	tsdb Search Time
6	82 sec (40 sec)	2.21 sec (1.88 sec)
18	135 sec (124 sec)	2.47 sec (2.29 sec)
69	-	2.78 sec (2.31 sec)

In order to prevent data caching from affecting tests results, all search tests have been performed after a reboot. Table 2a compares the databases size. Thanks to data compression, tsdb is about 1:10 of the equivalent MySQL database. The use of chunks data aggregation allows the cardinality of tsdb keys to be greatly reduced. Please note that whenever Internet domains have no queries for a given day, in MySQL we do not insert a record (in tsdb the space is taken anyway although compressed at a higher ratio) explaining why the number of MySQL records is less than the number of tsdb keys multiplied by 10k (i.e. the chunk size). Table 2b shows the results when extracting 6 months of data for a single domain. Even on search time, tsdb performs dramatically better than MySQL, being also characterized by a lower pace when searching on larger files. In order to understand how cache affects results, between brackets we have put the time taken for immediately repeating the same query using a different domain name. We believe that in tsdb (a) the limited benefit of data caching, combined with (b) slightly flickering search time (i.e. searching two different Internet domains is not performed in the same amount of time) is due to data compression. This is because decompression times as well compression ratio are not completely uniform across data chunks.

Figure 3 shows a tool we created for exporting data from tsdb via JSON (JavaScript Object Notation) to a 2.0 web application that dynamically shows queries originated by a specific Autonomous System (AS) to the .it DNS servers. This has been possible only because the tsdb data extraction is so quick that we can perform it interactively on a dynamic web page.

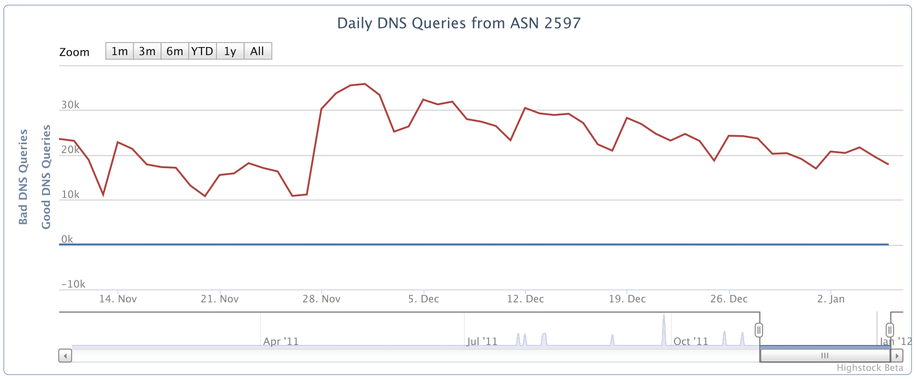

Fig. 3. tsdb time series export to a Web 2.0 interface

As previously discussed, tsdb does not have to be considered just as better tool for handling time series but rather as an enabling technology. One of the projects where we are working on at IIT/CNR focuses on monitoring the Internet AS-level structure. Within the scope of this project we have created a BGP (Border Gateway Protocol) client that connects to a border gateway and gets BGP updates from the router. Route changes are logged to file, and consolidated using custom tools we developed. So far we have used a relational database for tracking route changes. However we were not satisfied as its speed decreased over time due to the number of stored records. For this reason we migrated the system to tsdb where we maintain AS and route time series. This allowed us to both store data points at a much higher granularity (15 minutes in tsdb vs. one day in the relational database) and save disk space with respect to the previous solution. Having a higher granularity is a prerequisite for analyzing Internet topology changes and understanding its dynamics.

5 Open Issues and Future Work

So far tsdb is a library and a set of command line tools. We are planning to develop a tsdb server that can be remotely queried and instrumented through a network connection. We believe that the Redis query language is a good example of how to implement this in a simple and elegant way.

We are also trying to explore how tsdb could be enhanced for finding similarities between stored time series [15]. This is important for identifying Internet domains with overlapping behavior in DNS traffic, and autonomous systems with by similar route changes with the same time interval.

6 Final Remarks

Efficient time series handling is are very common requirement in many network monitoring contexts. Even if there is a significant amount of works described in literature, we have not been able to identify a tool for efficiently handling millions of time series in a simple yet effective way, and thus we have developed tsdb.

Its excellent performance when compared to similar tools and its simple design, makes it easy to integrate into existing applications. The validation phase has confirmed that tsdb can be effectively used for storing and analyzing a large number of time series in real life scenarios using limited disk space and characterized by append/search performances that are orders of magnitude better than the current state of the art. Finally, its backward compatibility with the popular rrdtool simplifies the migration of large existing infrastructures built upon rrdtool to tsdb.

Availability. This work is distributed under the GNU GPL license and it can be downloaded from https://svn.ntop.org/svn/ntop/trunk/tsdb/.

Acknowledgments. We thank Alfredo Cardigliano <cardigliano@ntop.org> and Alessandro Improta <alessandro.improta@iet.unipi.it> for their comments and valuable discussions during the design of tsdb.

References

1. Box, G., Jenkins, G.: Time Series Analysis: Forecasting and Control. Prentice Hall PTR (1994) ISBN: 0130607746
2. StumbleUpon, OpenTSDB: Open Time Series Database, `http://opentsdb.net`
3. Oetiker, T.: RRDtool: Round Robin Database Tool, `http://oss.oetiker.ch/rrdtool/`

4. Deri, L., et al.: Monitoring Networks Using ntop. In: Proc. of IM 2001 (May 2001)
5. Vlachos, M., Kozat, S., Yu, P.S.: Optimal Distance Bounds for Fast Search on Compressed Time-series Query Logs. TWEB 4(2) (2010)
6. Salomon, D.: Data Compression: The Complete Reference. Springer, Heidelberg (2000)
7. Rice, R.F., Plaunt, R.: Adaptive Variable-Length Coding for Efficient Compression of Spacecraft Television Data. IEEE Transactions on Communications 16(9), 889–897 (1971)
8. Harizopoulos, S., Liang, V., Abadi, D.J., Madden, S.: Performance Tradeoffs in Read-Optimized Databases. In: Proc. of VLDB (2006)
9. Anh, V.N., Moffat, A.: Inverted Index Compression Using Word-Aligned Binary Codes. Journal of Information Retrieval 8(1), 151–166 (2005)
10. Zukowski, M., et al.: Super-Scalar RAMCPU Cache Compression. In: Proc. of International Conference on Data Engineering, ICDE (2006)
11. Abadi, D., Madden, S., Ferreira, M.: Integrating Compression and Execution in Column-Oriented Database Systems. In: Proc. of 32nd ACM SIGMOD International Conference on Management of Data (2006)
12. Reinhold, L.M.: QuickLZ (2011), `http://www.quicklz.com`
13. Olson, M., Bostic, K., Seltzer, M.: Berkeley DB. In: Proc. of Usenix Annual Technical Conference (1999)
14. Mullins, C.S.: Database Administration: The Complete Guide to Practices and Procedures (2002) ISBN: 0-201-741296
15. Rafiei, D., Mendelzon, A.: Similarity-based queries for time series data. In: Proc. of the ACM SIGMOD (1997)
16. Brillinger, D.R.: Time Series: Data Analysis and Theory. Society for Industrial and Applied Mathematics (2001) ISBN-10: 0898715016
17. Krishnamurthy, S., et al.: TelegraphCQ: An Architectural Status Report. IEEE Data Engineering Bulletin 26(1) (March 2003)
18. Zhao, X.: High Performance Algorithms for Multiple Streaming Time Series, Ph.D. Dissertation, New York University (2006)
19. Shieh, J., Keogh, E.: iSAX: Indexing and Mining Terabyte Sized Time Series. In: Proc. of ACM SIGKDD (2008)
20. Tiwari: Professional NoSQL. John Wiley and Sons (2011) ISBN: 047094224X
21. Zaitsev, P.: Why MySQL could be slow with large tables? (June 2006), `http://www.mysqlperformanceblog.com`