Epimorphics
CONNECTING DATA

UK Centre for
Ecology & Hydrology

Survey of Time Series Database Technology

version: 1.0.0
date: 30 March 2020

Author(s) Brian McBride (Epimorphics Ltd.)
Dave Reynolds (Epimorphics Ltd.)

Reviewer(s) Matt Fry (UKCEH)
Oliver Swain (UKCEH)
Simon Stanley (UKCEH)
Mike Brown (UKCEH)

Contents

Introduction

This report has been prepared by Epimorphics Ltd. as part of the ENTRAIN project[1] (NERC grant number NE/S016244/1) which is a feasibility project within the "NERC Constructing a Digital Environment Strategic Priorities Fund Programme".

The Centre for Ecology and Hydrology(CEH)[2] is a research organisation focusing on land and freshwater ecosystems and their interaction with the atmosphere. The organization manages a number of sensor networks to monitor the environment, and also handles large databases of 3rd party data (e.g. river flows measured by the Environment Agency and equivalents in Scotland and Wales). Data from these networks is stored and made available to users, both internally (through direct query of databases, and externally via web-services). The ENTRAIN project aims to address a number of issues in relation to sensor data storage and integration, using a number of hydrological datasets to help define use cases:

- COSMOS-UK, a network of ~50 sites measuring soil moisture and meteorological variables at 1-30 minute resolutions
- The CEH Greenhouse Gas (GHG) network of ~15 sites measuring sub-second fluxes of gases and moisture, subsequently processed up to 30-minute aggregations
- The Thames Initiative, a database of weekly and hourly water quality samples from sites around the Thames basin

In addition this report considers the UK National River Flow Archive, a database of daily river flows and catchment rainfall derived by regional environmental agencies from 15-minute measurements of river levels and flows.

The current storage system for the COSMOS-UK and NRFA is a relational database, using an Oracle system supported by NERC and CEH IT resources. As it becomes possible to record and analyse data at higher frequencies, data rates and volumes increase, and it may become increasingly difficult, complex and expensive to scale a relational database to cope.

CEH commissioned this report to survey alternative technologies for storing sensor data that scale better, could manage larger data volumes more easily and less expensively, and that might be readily deployed on different infrastructures.

The CEH sensor data conforms to a pattern known as *time series data*. Roughly speaking, time series data can be thought of as a sequence of data records ordered by time.

The approach taken to producing this survey has been:

- carry out a review of the general landscape of time series database solutions
- identify the features that are of interest to CEH in selecting a new solution
- describe the scale of data that CEH might wish to store

[1] http://www.ceh.ac.uk/our-science/projects/entrain
[2] https://www.ceh.ac.uk/

- review a representative list of time series data base products and open source solutions
- interview several operators of similar storage solutions to learn from their experiences

In selecting the list of products and solutions for review more popular solutions were favoured and also at least one example from each of the different kinds of approach to the problem was included. Solutions that didn't appear to match the problem, e.g. in memory only solutions and solutions that were focussed on solving the problems of a specific domain, e.g. large scale monitoring in the internet of things were eliminated.

Time Series Data Overview

Roughly speaking, time series data is a sequence of data records ordered by time. Data produced by an environmental sensor which measures some value, e.g. surface water temperature at regular intervals, is a typical example of time series data.

The time intervals between records may be regular or irregular. The term *metrics* is used to refer to data at regular intervals. The term *events* is used to refer to data about events that occur at irregular times. Lightning strikes are events. 30 minute water temperature measurements are metrics.

An event series can be converted to a metric series by aggregating the event data over regular time intervals.

The time associated with a data record may be an instantaneous time or a time interval. Times may be distinct as in the example of regular surface water temperature measurements, or not, e.g. where the data represents observations of events which may be overlapping in time.

The data associated with a time may be a simple numerical value or a complex value such as a map or an image or a structured object such as a JSON formatted data structure.

Metadata may be associated with each time series, with individuals records and with sequences of records within a time series. Such metadata might, for example, provide information about the sensor that made the measurement or the method used to make the measurement.

Time series data can be stored in the normal way using traditional tools such as relational databases. Such databases are sufficient for many applications and have the advantage of familiarity. Existing data users are likely familiar with these tools and can use their existing skills to query and manage the data.

There have however, in recent years, been new applications of time series data which have overwhelmed the capabilities of relational databases.

The amount of data that needs to be collected and processed has been dramatically increasing. This is a challenge shared with managing many other kinds of data and has led to the development of technologies such as Big Data and so called NoSQL databases that are not built on the traditional relational model.

Also, the rate at which data is collected has increased beyond the point at which traditional tools can write it to persistent storage.

The metadata associated with time series data has been growing in complexity.

New technical approaches to managing time series data are being developed to cope with these challenges. These approaches optimise for the characteristics of typical time series data:

- the data tends to arrive in roughly time order
- the data tends to be accessed in sequential order
- the data is rarely updated
- For many applications the data can be aged, that is aggregated or discarded after a suitable interval of time

Different kinds of applications have different requirements for the mix of reading and writing the data. In some applications, the data is stored and rarely read, for example, where events are logged and the logs need only be checked in exceptional cases. In some applications the mix of reading and writing of data is about equal and in others the data is written once and read many times

Features of Interest

This report focuses on a number of features of interest that are likely to be useful in selecting an appropriate solution for CEH.

Scale

The solution should be able to support the anticipated volume of data, the ingest rates and query load. The following simple table describes the relationship between measurement rates and the number of metrics generated per second and the volume of data to be stored (assuming data is never deleted) for current and anticipated CEH sensor networks.

Network	Sites	Metrics per Site	Series	Rate	Metrics Per Second	Metrics Per Year	History	Target New Data Years	Total Metrics
Current COSMOS	50	60	3000	1 per 30 minutes	1.66	52M		10	520M
COSMOS @ 1 minute	50	60	3000	1 per minute	50	1.6B		10	16B
Larger COSMOS @ 1 minute	500	60	30000	1 per minute	500	16B		10	160B

NFRA[3]	1.5k	2	300	1 per day	0.03	1M	50M	10	60M
NFRA @ 15 minutes	1.5k	2	300	1 per 15 minutes	3.33	105M	50M	10	1.1B
GHG @ 20Hz	50	20	1000	20 per second	20 000	31.5B		10	315B

Are SQL and its Drivers (ODBC, JDBC) Supported

SQL is the query language supported by the currently used storage solution. Whilst SQL does not guarantee interoperability of queries on different implementations, solutions that support SQL may require less disruption to the existing system when a new solution is deployed.

Conceptual Model of a Time Series

Whilst there is a general notion of how time series data is structured, different solutions offer different models for representing time series data. There can be differences in how time is represented, data types supported and how metadata is represented.

CEH currently undertake a significant amount of time series analysis, often using bespoke software tools, and also have existing tools for time series data management. There are therefore existing implementations of time series models. The ease with which current ways of working, including code implementing these models, can be used with different database solutions is therefore important. Whilst this can't be directly assessed here it is important to understand the time series model used by the systems under review.

In the *wide table* model, a time series is a table keyed by time, with many columns of values for different metrics. This model may be appropriate where a single sensor measures multiple parameters.

In the *narrow table* model, a time series is a table, keyed by time, with a single column of values. In this model the data from a sensor measuring multiple metrics would be represented as multiple time series of single metrics.

Many time series databases have the notion of *tags*. A tag consists of a pair - a name and a value. A time series may have an associated set of tags describing it.

Some time series databases have the notion of *labels*. Labels are also name value pairs, but are associated within individual rows in a time series. Labels can be used to add metadata to individual measurements, e.g. a sensor id, where the sensor is allowed to vary within a time series.

[3] UK National River Flow Archive

Approach to Data Storage

Many time series databases are constructed using a general purpose storage solution such as a relational database or a NoSQL store such as Casandra. Some have their own implementation of a NoSQL store.

These different approaches to data storage have some fundamental strengths and weaknesses so it is important to understand the approach to data storage and the strengths and weaknesses of each approach.

At heart, solutions built on relational databases don't scale as well as solutions built on an appropriate NoSQL store. Relational databases are more limited in the total volume of data they can handle and the rate at which data can be added, deleted and modified. NoSQL stores can achieve greater scaling at the cost of limiting functionality such as the ability to perform join queries or transactional (as opposed to eventual) consistency.

These scaling limitations are largely inherent in the use of the relational approach. Normally a relational database engine will assume that the data is all available within a single logical machine. So when data grows large enough that index sizes exceed available memory then data must be swapped to disk resulting in a performance impact. Scaling a relational database typically means *scaling up* - configuring a larger host machine with more memory and more processors, whereas the various NoSQL approaches provide some form of sharding - splitting the data across machines so that it is possible to *scale out* by 'just' adding more servers.

However, there is no hard and fast line here. High end relational databases do provide some ability to partition data across multiple instances, or partition tables into sub-tables, and very large Oracle instances are certainly possible. It is more that the cost and complexity of scaling up a commercial relational database can be much higher than that of scaling out a NoSQL option (provided the NoSQL option can deliver the desired functionality).

Solutions built on a relational database, either directly or using time series specific extensions such as TimescaleDB have their own advantages, such as the ability to represent complex metadata and wider availability of knowledge and experience of using them and operating them.

Some NoSQL databases are wide column stores such as Casandra and HBase, that store data in tables with a very large number (millions) of columns. Columns are organised into groups called families where all the columns within a family are stored within a single server, but columns in different families can be spread across a cluster of computers.

Some NoSQL databases are column stores, where the data for each column is stored contiguously on disk. These are particularly suitable for analytic workloads in which queries access only a few of the columns in a row and only the data for the columns accessed need be read from disk. The data values in a single column are often very similar and column stores can take advantage of this by compressing the data in each column independently to achieve better compression ratios and higher query performance.

OpenTSDB, TimelyDB and Heroic are examples of time series databases built on NoSQL stores.

Some time series databases, such as the popular InfluxDB, build their own custom storage solutions based on the same basic algorithm (LSM Trees) as other NoSQL stores.

And finally, some time series solutions, such as Chronix, are built on top of a document database such as MongoDB or Lucene.

Deployment Infrastructure

The ease of deployment, and IT requirements, of any new system are essential considerations for CEH. An important use case is for streaming sensor data via web-services, and in this case CEH's Environmental Information Platform (EIP) infrastructure would be used. This infrastructure is based on virtual machines deployed using Docker and a Kubernetes cluster. The fit to this infrastructure of the database solutions considered, in a single or multiple server configuration, and ease of ongoing maintenance, is an important consideration.

In addition, since scalability is one of the reasons for looking at possible alternatives to the current relational database based solution, it is likely that the solution will be deployed on a cluster of servers rather than on a single server.

Some solutions come with their own capability to manage a cluster of servers. They can, in essence, be deployed stand alone on a collection of commonly available servers running a commonly available operating system.

Other solutions are deployed into a cluster controlled by separate cluster management software such as Hadoop or Kubernetes.   Managing such a cluster to support the time series database solution requires additional learning and administration and this is part of the cost of the solution.

CEH has expertise in running a docker container based cluster using Kubernetes, the platform used to support its Environmental Information Platform (EIP). It is therefore of interest whether a solution is available packaged for deployment on such platforms.

Containers simplify software installation and maintenance. Docker containers are essentially prepackaged collections of software that are easier to deploy because they can be run and deployed as a complete unit.   Prior to containers software usually had a list of prerequisites that had also to be pre-installed for the software to work. Ensuring the prerequisite software was installed and resolving conflicts when different software components had conflicting prerequisites could be difficult and complex.

Kubernetes is a modern tool for managing a cluster of servers which is rapidly growing in popularity.   There are multiple tools for describing the collection of containers making up a software application and how they should be deployed and configured. A popular way of doing this is to use Helm charts and we will use the availability of Helm charts for a solution as an indicator of whether there are prebuilt configurations for deployment on a Kubernetes cluster.

It should be noted in passing however, that whilst it is possible to deploy stateful applications such as databases into a Kubernetes cluster, this is currently a more complex and difficult undertaking than deploying stateless or near stateless applications such as web services.

Some solutions are available as managed services which take care of hardware provisioning, software installation, configuration and maintenance, backups etc where a provider offers to run the service for a fee. This significantly reduces the burden of managing and maintaining the software and servers running the solution.   A number of storage solutions are available from a number of vendors in this form, including relational databases, various NoSQL databases and specialist time series databases.

Software Licensing and Support

Different timescale database solutions are available under different licensing terms and support agreements.

Some are available as commercial solutions where a license to use the software must be paid for and this usually comes with commercial support should problems be encountered.

Some are available for free under an open source license and with support dependent on the good will of the development team. In such cases there is often a separate commercial support option.

Some are available under hybrid terms, with free availability subject to some restrictions such as limited functionality or limits to scale and a paid version without those limitations.

Maturity and Popularity

In selecting a solution, it is important to have confidence that the solution will continue to exist, improve and flourish into the future. A mature solution which has been available for some time has proved its staying power.

More popular solutions are likely to be of acceptable quality, for it to be easier to find or develop personnel with experience of the solution and to be better funded and supported.

General Considerations

Handling complex metadata

Most stores can store simple metadata, such as a sensor id easily. Relational stores can also store more complex metadata, e.g. further information about a sensor such as its manufacturer, model number, serial number, maintenance history, etc. However, NoSQL stores are designed for large volumes of relatively simple data tables and since they do not support joins are not good at querying more complex structures.

Where a time series solution doesn't itself handle complex metadata (for sensors, instruments, provenance etc) the metadata can be stored elsewhere such as a relational database, graph or triple store, property graph store or document store. There are two fundamental approaches to how this might be done.

In the first approach links to the metadata can be stored along with the time series data records. For example, a data record might contain a column (or several columns) with an identifier for the sensor (or sensors) that made the measurements. This identifier can then be used as a key to query the metadata store to retrieve metadata about the sensor. It is also possible to scan the time series data to identify measurements made by a particular sensor, given its identifier. With this approach, the links to the metadata should be part of the data ingested into the store and the data in the time series store and the data in the metadata store have to be kept consistent.

In the second approach, there is no explicit link from the sensor data to the rich metadata. Rather the metadata describes the sensor data that it applies to. For example, the metadata store may store information for each sensor and also which time series it generated data for over what time intervals. For a given data record, knowing what time series it is in and its timestamp, the sensor metadata can be retrieved. The other way round, given a sensor, it is possible to query the metadata store to determine what data records it produced. With this approach, the ingested data need not include links to the metadata and maintaining consistency may be easier.

Techniques for Improving Scalability and Performance

There are techniques for improving the scalability and performance of a time series store that may be used with several different kinds of store in some circumstances.

Batching Incoming Data

If ingest performance is an issue, Incoming data can be collected into batches prior to being loaded into the store. Inserting a single large batch of data is often more efficient than loading many individual observations. Some stores have specialised API entry points for loading batches of data. To avoid the potential loss of Incoming data before it is loaded into the database, it can be collected in a reliable queue, such as Apache Kafka, prior to being formed into a batch and loaded into the store.

Many Measurements per Row

Many measurements can be stored in a single row in the database. There is some performance cost associated with storing each row in a store which can be reduced by storing multiple observations in a single row. Some NoSQL stores can store millions of columns in a row which allows all the measurements for quite a large time period to be stored in one row. This clearly works better when observations are inserted into the store in batches.

Data Compression

The stored data can be compressed. This places a higher processing burden on the store but can significantly reduce the amount of data to be transferred to and from non-volatile storage. It is reported that this can result in a 1000 fold improvement in insert and access performance in some circumstances.[4]

Data need not be compressed when it is ingested. A common technique is to load the data into the store uncompressed which makes it easier to handle cases when data arrives out of order, but compress it later in the background once it is stable.

Many solutions have compression built in, but compression outside the store could be considered for those that do not.

Example Time Series Databases

Relational Database

Time series data can be stored in a traditional relational database, such as the open source PostgreSQL or commercial Oracle. CEH currently stores its time series data in an Oracle relational database.

Relational databases permit the storage and querying of data organised as tables, a model which fits time series data well. It enables the storage and querying of metadata, also organised as tables.

Relational databases can support both wide and narrow table models as appropriate for individual time series.

Data values can be any of the data types supported by the database, including integers, floats and strings. Some relational stores, such as PostgreSQL, support storing values as complex data types such as json records.

Data updates are transactional, that is, when an update occurs a user or a program querying the database will either see the data before the update is applied or after the update is applied but will not see the database with the update partially applied.

Queries can be expressed in the SQL query language, which is mature and well understood by many developers. There are mature tools for querying a database from computer programs written in a variety of programming languages.

Queries can join data from multiple tables enabling filtering of retrieved data based on metadata values and also returning composite data from multiple time series. Relational databases have built in functions for aggregating data at query time, enabling queries to compute new time series from existing ones, such as a lower resolution summary time series from raw data.

An issue with using a relational database is its ability to scale. Relational databases are more limited than other solutions in the rate at which data can be ingested into the store and the total volume of data that can be effectively stored.

There are reports that PostgreSQL database ingest performance will start to reduce from 15K rows per second when the number of rows exceed 50M. The CEH ingest rates are much lower than this figure, except for full GHG data at 50ms resolution. However,

[4] Time Series Databases: New Ways to Store and Access Data, Dunning, T & Friedman E, Chapter 3.

full GHG data at 50ms resolution would also exceed the volume of data a relational store could handle.

A relational store will store 1B rows of data in a table, but may begin to struggle above 10s of billions. This is a factor for CEH. The current COSMOS and NFRA data for the next 10 years will fit within this limit, but potential increases in resolution to 1 minute for COSMOS data may stress the capabilities of a relational store.

A relational database may be replicated on multiple servers. This is usually to provide greater resilience should a server fail and also increases the query load that can be supported by enabling multiple queries to run in parallel on different servers. The number of servers in such a cluster tends to be low, e.g. 2 or 3. As noted above larger clusters are possible but are expensive and complex to manage.

Relational databases are a mature well understood technology with well resourced commercial and open source implementations. There is a wide availability of expertise both in operating databases, tuning them and querying them. Additional functionality has been developed to augment relational databases, such as Postgis which adds support for geospatial queries to PostgreSQL.

Relational databases are available as prebuilt packages for multiple operating systems, as docker containers and with Helm charts for deploying to Kubernetes.

TimescaleDB

TimescaleDB is an extension to PostgreSQL designed to mitigate some of the limitations of a relational store. It can maintain a high ingest rate as the volume of data in the store increases and has recently (November 2019) added a new low level storage mechanism that supports high levels of data compression, reduces storage costs and increases query performance.[5] This active development of significant new functionality is encouraging.

TimescaleDB shares many of the characteristics of using a traditional database store directly. It can support both narrow and wide table models, and can be queried with the SELECT SQL command including with arbitrary WHERE clauses, GROUP BY and ORDER BY commands, joins, subqueries, window functions, user-defined functions (UDFs), HAVING clauses, and so on.

Time Series Conceptual Model

TimescaleDB does not impose its own model of a time series. It supports tables and the user is free to design how they represent their time series data in tables.

Higher Ingest Rates

The reason that the ingest rate degrades in a normal relational database is that once a database table exceeds a certain size, its index no longer fits in main memory and the database has to access secondary storage to update the index. Access to a secondary store such as a SSD or magnetic disk is much slower than accessing main memory.

[5] https://blog.timescale.com/blog/building-columnar-compression-in-a-row-oriented-database/

Hypertables

TimescaleDB works by splitting the data in a table up into multiple database tables known as *chunks* each of whose indexes fit in main memory. It does this by adding the feature of a *hypertable* to PostgreSQL. A hypertable is a large table of data that can be queried like a normal database table, but is implemented by splitting it up into a number of chunks. The size of each chunk is managed by TimescaleDB to ensure that its index fits in main memory. Because time series data arrives in roughly time order and is keyed by time, new data is inserted into the chunk containing the most recent data. This operation is significantly quicker than inserting the data into a full sized table because that chunks index fits in main memory.

TimescaleDB claim they can support 10's of billions of row hypertables without slowing down ingest rates on modest hardware.

Data Compression

In a recent innovation[6], TimescaleDB has added support for column oriented storage. PostgreSQL normally stores each row of data in a table contiguously. This is efficient when queries access many of the columns in a row and for data insertions when the wide table data model is used. But is less efficient when queries access only a small proportion of the columns in a table and it makes compression more difficult and less effective.

TimescaleDB now support a hybrid storage model, where, when data is inserted, each row each row is stored contiguously and uncompressed, but some configurable time later, in the background, this data is reformatted to restructure the data into a compressed column format. This enables faster querying because less data has to be read from storage, partly because the data is compressed and partly because only the data for columns that are accessed need be accessed. Once compressed the data cannot be modified without decompressing it.

Deployment

TimescaleDB is available as open source deployable software and as a commercial cloud service. There are software packages available for a number of platforms including Ubuntu, Debian, Redhat/Centos, MacOS and Microsoft Windows. It is also available as a docker container and as an Amazon virtual machine image (AMI). A Kubernetes Helm chart is also available.

InfluxDB

InfluxDB is a specialist, purpose built time series database with its own NoSQL store implementation and an SQL like query language without joins. It is significantly the most popular time series database at the time of writing (Nov 2019) as measured by DB-Engines.

6 https://blog.timescale.com/blog/building-columnar-compression-in-a-row-oriented-database/

Time Series Conceptual Model

In InfluxDB a *measurement* corresponds to the concept of a table containing multiple columns containing data and tags. Data columns are called *fields* and may contain string, float, integer or boolean values. Other columns contain tags whose values must be strings. Each row in the measurement, known as a *point*, has a timestamp, a *field set* (set of fields and values) and a *tag set* (set of tags and values)

A *series* is a collection of points from the same measurement with the same tag set. Tags are used to identify time series and a unique combination of tags and their values in a measurement defines a time series.

For example, a network of sensors measuring water temperature, depth and flow rate might be a single measurement. Each point would contain 3 fields, temperature and depth and flow rate and a tag would define the location of the sensor. Each location would then define a different time series. Alternatively, each of water temperature, depth and flow rate might be defined as a separate measurement each with only one field.

Tags are metadata that are used to identify series rather than additional information about the data, such as the sensor used to collect the data. Such additional metadata can be stored either in tags, in which case changing the value will create a new series, or in fields, in which case the data is not indexed and can only used as a filter when scanning.

Querying Data

Queries are expressed in InfluxQL query language which has an SQL like syntax and returns results in JSON format.

InfluxQL supports SELECT for projecting data from measurements, WHERE for filtering result streams, GROUP BY for aggregating data, ORDER BY either ascending or descending time and LIMIT and OFFSET for selecting subsets of the points in a measurement.

InfluxQL supports some basic operations such as add, subtract etc and a range of functions to select, transform and aggregate data.

So called *continuous queries* are queries that are run automatically at specified intervals and can produce new data that is stored in the database.

InfluxQL does not support joins which are left to downstream processing. Joins are supported by Flux, from the same company, a functional language for querying an InfluxDB database. Flux is based on JavaScript and has a very different conceptual model to SQL.

Scale

InfluxDB claim[7] that a single 8 core server with 32GB of ram can support 10M unique time series, ingest 250k field values per second and execute 25 'moderate' queries per second.

[7] https://docs.influxdata.com/influxdb/v1.7/guides/hardware_sizing/

Given the sizing of larger COSMOS at 30k series and GHG at 1000 series and an ingestion rate of about 20k metrics per second, a single server implementation of InfluxDB should be sufficient for CEH's scale requirements. Consideration might be given to running a cluster of servers to provide for resilience against server failure and enabling one server to be taken out of service at a time for maintenance.

Licensing

A single server implementation is available under the open source MIT license.

A single server *community edition* with extra features is available, subject to the condition that it is not resold. The extra features include:

- advanced analytical functions
- automatic continuous aggregation

The enterprise edition is available for a fee with commercial support and further features including the ability to run on a cluster of computers..

Deployment

Software packages are available for MacOS and Linux. It is available as a Docker container. and a Helm chart for deploying to Kubernetes is available.

The enterprise edition is also available as a cloud service.

OpenTSDB

At the time of writing TSDB is the 6th most popular time series database as ranked by DB Engines[8], though its popularity has declined from the year before.

OpenTSDB stores time series data in Apache HBase or MapR-DB. These are databases for storing very large sparse tables of data (based on Google BigTable design) that are in turn built on the Apache Hadoop big data platform. Running OpenTSDB therefore requires running a Hadoop cluster.

The implementation basically consists of a daemon process that accepts data and stores it into the underlying storage database, implements a query API and a number of other simple services.

Time Series Conceptual Model

In TSDB a time series data point consists of:

- a metric name
- a timestamp
- a value
    - 64 bit integer
    - single precision floating point
    - JSON formatted string

[8] https://db-engines.com/en/ranking/time+series+dbms

- a set of tags (key/value pairs) that identify the time series of the point

Querying Data

TSDB supports a RESTful web API for querying and accessing data. The default format for data returned from the API is JSON, but other data formatters can be plugged in to the implementation.

The fundamental query operation of the API is to retrieve and aggregate data from one or more time series, specifying:

- the metric to be retrieved
- the tags that identify the series
- filters on the data
- and aggregator, e.g. sum or average

The API is designed to support selecting and retrieving data for processing by elsewhere in the cluster. Joins are not supported.

Scale

OpenTSDB can be scaled to support very large amounts of data and very high ingest rates by adding more servers to the cluster. The ingest rate is achieved through a range of techniques such as batching updates and exploiting the wide-columns available in the underlying storage. The HBase store can support "billions of rows with millions of columns" so collections of time series totalling 10^{12} measurements or more are feasible on a large enough cluster.

Deployment

OpenTSDB deploys into a Hadoop cluster.   At the time of writing neither official docker containers or Helm charts were available although a web search will find some unofficial docker containers.

Cassandra

Cassandra is not specifically a time series database but it is used sufficiently often in such a role, either directly or as the storage layer of other tools (see Heroic below), that it is worth reviewing here.

Cassandra is a horizontally scalable NoSQL store. In the underlying implementation it is a wide column family store which orders, groups and partitions data across servers based on a partition key. It provides an SQL-like query language, CQL, which hides the underlying flexibility of the wide-column store by requiring an explicit schema to be declared (including a definition of the partition key) and then used by queries. However, the datatypes available do include maps which provide some flexibility.

Time Series Conceptual Model

Since Cassandra is a general purpose column store it is up to the system developer to choose a suitable schema within the limitations imposed by the NoSQL model. A typical approach is as follows:

Represent each data point as:

- a metric or measure reference[9]
- a timestamp
- a date[10]
- a value (typically a double)
- a metadata annotation (using the map data type to allow arbitrary extensible annotations of each data point, e.g. to record quality flags)

To support different query patterns the same data is stored multiple times with different index keys.[11] For example, to allow retrieval by either date range or by a measurement (or set of measurements) we would create two "tables", one using a key of `date/metric/timestamp` and one using `metric/date/timestamp`.

While, in principle, the sensor metadata could also be stored in Cassandra the limitations of the Cassandra query language (particularly the lack of joins) means that it is more common to store the metadata separately in a relational, graph or document database.

Querying Data

The data is queried using CQL with familiar SQL like syntax using SELECT, WHERE, GROUP BY and ORDER BY. However, query patterns that would be inefficient given the table's key structure will be rejected (whereas in a relational model they would just be slow). No joins are supported.

Basic aggregation functions are supported (count, min/max, sum, average).

Scale

Cassandra is horizontally scalable and, so long as the table keys are chosen well, has excellent scaling. Initial writes are very cheap, though deleting and rewriting is more expensive.[12] There are cassandra installations that scale to hundreds or thousands of nodes and support up to 1 million queries a second. Instagram use it to support 1 billion monthly active users.

Deployment

Cassandra contains its own cluster management software and can be deployed on a cluster of computers without additional infrastructure. It is available packaged as a docker container and there is also a Helm chart to support deployment on a Kubernetes cluster. In addition there are several Kubernetes *operators* available for Cassandra which help with operational tasks such as backups.

[9] Some identifier that represents the time series, this would typically combine both a site identifier and an identifier for the particular parameter measured.
[10] This is redundant since it can be computed from the timestamp but it allows data to grouped conveniently.
[11] In Cassandra writes are cheap and data is compact so the redundancy is not that much worse than declaring different indexes in a relational model.
[12] A trade off which is a reasonable match to time series data

Heroic

Heroic is an open source time series database that stores its metric data in Cassandra or Google Cloud Big Table service and metadata in Elasticsearch. It is designed to support cases where there are many millions of separate time series defined by a complex set of tags where a search engine is required to find the tags and values that define the time series of interest. Whilst its initial release was in 2014, it describes itself as "alpha: use at your own risk" which may rule it out.

Data Model

The data model is familiar. A time series is identified by a key and a unique set of tags and resource identifiers. Tags are indexed and can be used efficiently in searches and filters. Resource identifiers are columns that are not indexed, but can be used for aggregating data. All data values are 64 bit floating point numbers.

Query Language

Heroic queries are fundamentally expressed as a JSON formatted data structure, and can also be represented in a textual query language HQL.

HQL supports:

- specifying a time range
- selecting time series based on tags
- downsampling time series, reducing the granularity of a time series by computing some aggregation function such as counting, summing, averaging etc values over intervals
- GROUP BY enables aggregation over time series with tag values in common.

HQL does not support joins.

Scale

Heroic is highly scalable as it builds on two horizontally scalable components, Cassandra or Big Table for metric data storage and Elastic Search for metadata storage.

Deployment options are more limited than some other solutions. Heroic itself is available as a docker container, or can be built from scratch. Pre-built packages for common operating systems do appear to be readily available. To construct a complete solution several other components will have to be installed and configured separately including a storage engine such as Cassandra, Elasticsearch and Apache Kafka.

Amazon Timestream

Amazon Timestream is a new cloud service product from Amazon which is currently in preview. Limited information about it is available.

It is a specialist time series store offered as a serverless service. SQL like query is available.

It is marketed as being capable of handling very high ingest rates (trillions of events per day) and a very large volume of data, suitable for large IoT applications for example. It is primarily being marketed as a significantly faster and lower cost solution than a relational store when ingest rates are high.

That said its marketing suggests that industrial sensor monitoring is also a target market, where its selling point is to be able to automatically flex the number of servers to keep performance predictable and costs low.

Direct query to file

A group of relatively new technologies provide the option of scalable query and analytics directly over data held in plain files, generally providing a fully featured SQL interface without necessarily an actual relational or other database. Several of these include objc/jdbc connectors to allow the query engine to be treated as if it *were* a database. These tools blur the lines between a true database and a simple file store.

They provide the option to treat a bulk file store as just another part of an extended database and so give the option of query access to raw high frequency data which might normally be regarded as too bulky and/or infrequently accessed to be worth managing in a true database. For example, it might be reasonable to store and access raw 20Hz GHG data in such a way while storing the derived 30 min (or 1 min) averages in the main database.

There are two elements to this approach - high performance SQL query engines and columnar/indexed file formats.

A number of open source and commercial SQL-over-file engines have been developed in recent years including Apache Spark, Apache Drill and Dremio. Each provides a fully featured SQL implementation, including joins, but are able to query a range of data sources including simple CSV and JSON files, "big data" file formats and other databases (both relational and NoSQL). Each allows the processing to be distributed across a number of nodes in a cluster so as to gain high performance through parallelism. Whereas Apache Drill and Dremio focus on scalable query, Apache Spark is a fully featured analytic engine that can run analyses expressed in Python, R or Scala/java, at scale. Apache Spark, for example, can be deployed as a standalone cluster or within a Hadoop or kubernetes cluster. It supports explicit declaration of data models in the form of *DataFrames* but can also extract models from some source formats such as Apache Parquet (see below).

Some amount of indexing can be achieved by grouping the data files hierarchically by e.g. site and date. The SQL engines can then use this hierarchy in order to limit the files that need to be scanned for a given query.

For higher performance, instead of storing data in plain text files such as CSV format a binary and indexed format can be used. In particular, a core file format used in the Apache Hadoop "big data" ecosystem is Apache Parquet. This format stores data within files as columns with column compression and range bounds metadata in each file. This allows query engines to efficiently read only the columns relevant to a query and to rapidly skip files and blocks which are out of range of the query.

Typically, such files are then arranged in a hierarchy which carries partition information. For example, a directory per site, each containing a directory per month, each containing a file per day of measurements. Tools such as Apache Drill, Apache Spark and Apache Hive can use this partition information as part of query processing so as to avoid even querying the index of files that are in irrelevant partitions. Information on how files are laid out and partitioned may be defined when configuring a specific tool or can be held in a separate metadata store. For example, multiple tools can reuse an Apache Hive *metastore* as their source of information of where datasets are and how they are laid out.

A brief experimental investigation[13] evaluated some simple test queries over simulated 20Hz measurement data comprising around 2B measurements. In that test, use of Apache Parquet format gave a 40x performance boost compared to processing raw CSV files and a single desktop machine was able to deliver sub-second queries over the files at that scale of data.

A separate group of options in this same space are the binary file formats in use in scientific computing, particularly the NetCDF format used in climate and forecasting applications. The NetCDF format stores data as arrays of fixed size values. This allows (API) queries to retrieve elements from the mult-dimensional arrays by direct file seeks, with no scanning required. This allows large sets of data to be published as collections of NetCDF files and users can extract a desired slice of the data very efficiently. Tools such as NOAA's ERDDAP exploit this format to provide fast access to data set archives including mapping the extract data slices to alternative delivery formats.

Note that there are also experimental platforms which combine use of formats like NetCDF and HDF with the Big Data tools for parallel distributed query. ClimateSpark[14] is an example of this which provides connectors to such formats from Apache Spark.

Summary Comparison Table

Database	**Core Tech**	**Data Model**	**Rich Metadata**	**Query**	**Joins**	**Scale**	**Infrastructure**	**License**	**Maturity**	**DB Engines TSDB Rank**
PostgreSQL	relational	flexible	yes	SQL	yes	1-10B rows ingest performance degrades as size exceeds 50M rows	stand alone hosted service Docker Container Helm Chart	Open Source	>10 years	n/a
TimescaleDB	relational (extended postgreSQL)	flexible	yes	SQL	yes	10B rows	stand alone hosted service Docker Container Helm Chart	open source + commercial	1.0 release in 2018	8

[13] See *Note: File-based time series access*, Dave Reynolds, 16 August 2019

[14]https://www.researchgate.net/publication/323908538_ClimateSpark_An_in-memory_distributed_computing_framework_for_big_climate_data_analytics

Cassandra	NoSQL column family store	flexible	no	CQL	no	horizontally scalable	stand alone hosted service Docker Container Helm Chart	open source	2008	n/a
OpenTSDB	HBase or Map-R	time series with tags	no	simple API		horizontally scalable	Hadoop	open source	initial release 2011	5
InfluxDB	Custom NoSQL store using LSM trees	time series with tags	no	SQL like syntax	no	horizontally scalable	stand alone hosted service Docker container Helm Chart	open source + commerci al	First release in 2013 1.0 release in 2016	1
Heroic	Cassandra + Elastic Search	time series with tags	yes	HQL + JSON API	no	horizontally scalable	stand alone +serverless	open source	initial release in 2014	14
Apache Drill (example Direct query to file)	distributed analytics engine over parquet files	flexible	No	SQL	yes	horizontally scalable	Hadoop	open source	first release in 2013	n/a

Other Systems

This section includes brief summaries of approaches taken by other organisations to address similar requirements.

Irish Marine Institute

The Irish Marine Institute[15] is a state agency responsible for marine research, technology development and innovation in Ireland. Its National Marine Data Centre provides online marine data services and stores about 6TB of data of which about 4.5TB is video and mapping data.

Their time series data is being moved to a TimescaleDB database. Previously they had used Cassandra to store their time series data. Technically they were happy with Cassandra, but there were issues with availability of expertise so they moved to TimescaleDB.

They ingest about 10k measurements an hour into this store from their sensor networks. Their metadata is stored in a separate relational database.

[15] https://www.marine.ie/Home/home

For dissemination of the data it is exported in NetCDF file format and made available through an ERDDAP data browser.

Met Office

The UK Met Office are building a new system to ingest and store monitoring data from approximately 300 land based monitoring sites. Most sites are generating data at a rate of 1 set of measurements per minute. In addition there are approximately 100 marine based sensors on buoys, light ships and other vessels. Most data is transferred from the data loggers in the field hourly but this might be varied depending on conditions. The number of parameters measured at each site varies between 8 and 16; on average there are approximately 12 per sensor. Approximately 5000 metrics per minute are being ingested.

That is about 2.5B metrics per year. If we assume a straight forward database schema with each measurement from each location stored in a single row, each year's data will occupy approximately 210M rows which is about 2B for a 10 year period. This is similar in scale to the current COSMOS network recording data at 1 minute intervals.

The Met Office have made a strong commitment to a serverless architecture for this new system, that is an architecture that uses storage and compute services offered by a vendor who manages the servers on which those services run and there is no need for the Met Office to manage any servers or software themselves.

They have elected to use the MySQL flavour of the serverless version of the Relational Database Service (RDS) provided by Amazon Web Services (AWS).

British Geological Survey

BGS operate a centralised service for storing data for a number of sensor networks providing data in near real time. Networks have varying scales from one with 50 bore holes and dozens of sensors in each to others with just a few sensors.

The different networks generate data at different rates and there is a desire to be able to vary the rate on demand, e.g. when some site is to be affected by weather conditions that may generate some interesting data. Some sites and networks are used for specific experiments, some of which can generate a relatively huge amount of data (20TB/week) when running. Separate arrangements are made for data delivered at this scale.

The main sensor network data is currently stored in an Oracle relational database. As well as storing the data, this database also stores metadata and data model definitions such as a glossary of controlled terms.

We do not, at the moment, have figures for the scale of the rate at which data is ingested or the amount stored other than that the database has tables containing millions of rows, but not billions. They are confident that they could store double or triple the volume of data they currently have, but a ten fold increase would require some thought.

BGS are currently less concerned about being able to scale data storage than they are about users being able to make sense of the volume of data being generated.

Primarily raw sensor data is stored. This data is subject to processing and QA to create published datasets.

Their goal is to provide consumers with uniform access to data. Whilst access may appear seamless to data consumers, they envisage in the future a multi-architecture approach to storing data rather seeking a one size fits all solution to their range of needs.

Environment Agency Hydrology Service

The Environment Agency maintains a long term, quality controlled, archive of data covering river levels and flows, groundwater levels and rainfall. The data covers around 5000 sites and in some cases the series extend back as far as the 1960s. Recent data is 15 min resolution with daily mean derived time series also being available. The data is managed using Kister's Wiski time series database using Oracle as the underlying store, which meets operational needs for data management, distributed access and processing of time series.

In order to make the data available as open data EA commissioned a data platform for open publishing of data at this scale. The platform[16] uses Apache Cassandra for storing the time series data (using the modelling approach sketched earlier) and stores the metadata on the sites, locations, and particular measurements in a separate triples store (Apache Jena). The data is queried through an API which allows users to retrieve metadata on stations (aka sites) and time series and to retrieve the data for individual time series (all, latest or by date range) and to retrieve the latest measurements for all time series.

The service was scaled to, in principle, publish the whole of the data archive O(10^{10})annotated measurements). However, to date only river flow data for around 1000 sites at daily mean resolution has been published.

Implications for CEH

CEH are right to be concerned that increasing the resolution at which data from COSMOS from 30 minutes to 1 minute and significantly increasing the size of the COSMOS network would stretch the capabilities of the current relational database solution for storing and querying the data.

That said, CEH's scale requirements are relatively modest when compared to some of the larger scale financial or Internet of Things (IOT) applications.

Bearing in mind the Irish Marine Institute's experience, if SQL access continues to be important, then TimescaleDB offers a route to increasing ingestion rates and increasing the volume of data stored sufficiently to meet likely CEH's requirements.

This solution however, does not offer horizontal scalability (the ability to truly distribute the data storage and workload over a cluster of servers) and so, especially if high rate GHG data is to be stored, this solution may be approaching the limits of what this technology can offer.

If horizontal scalability is a requirement, then it is best to avoid setting up a Hadoop cluster just to support a time series database when there are good alternatives that do

[16] Developed by Epimorphics.

not require this. Hadoop is now quite a mature software platform and the more modern, container based Kubernetes is the current cluster management platform de jour.

Both InfluxDB and Cassandra are horizontally scalable, are designed to run on a cluster of servers and do their own cluster management. According to DB-Engines, InfluxDB is significantly the most popular time series database solution. Whilst it has open source and community offerings its cluster solution is part of the paid-for enterprise version. It is not clear how much of its stated popularity relates to the high end cluster version and how much to the free versions.

Cassandra is a popular and mature NoSQL store with an open source license that is widely used for storing time series data. Standard approaches to storing time series data in Cassandra are publicly documented and easily available.

InfluxDB and Cassandra offer SQL like query languages without some relational operators such as join. The principal technical disadvantages of NoSQL stores is that they don't do joins and they are not designed for representing rich metadata.

The issue with joins can be mitigated by:

- populating the store with precomputed joined data where the joins are known in advance
- providing APIs that allow computing joins outside the store (see External Analytics below)

The issue of storing rich metadata can be solved by storing the metadata in a suitable store such as a document store or a triple/quad store.

Heroic is an off the shelf solution that offers an integration of Cassandra for data storage and Elastic Search for metadata storage and should be considered if a NoSQL option is to be considered further. However it does describe itself as still in 'alpha' release and that may be sufficient reason to rule it out.

In Epimorphics, we have had good results integrating Cassandra for data storage with a triple store for storing rich metadata with access mediated by an API.   The principal advantage of using a triple store is that it supports a flexible and expressive graph structured data model and an expressive query language.

Also note the option to store raw data in plain or binary/indexed files but still provide interactive query over that data, through the use of modern high performance SQL engines. This may be relevant to cases like GHG where the raw 20Hz measurements could be stored and processed this way - avoiding the expense and complexity of scaling up the main time series database to hold this raw data while still offering good query performance.

Whatever solution is adopted, consideration should be given to migrating data consumers from direct query access to the database to mediated access through an API. A change of database technology would be an opportunity to introduce this. There are a number of advantages of mediating access through an API:

- it gives CEH flexibility to make changes to how the data is stored without impacting users
- it enables users to access the data without having to understand the storage design

There are hosted service options available for many of the solutions discussed in this report.

Appendix 1 - External Analytics

In addition to simple retrieval of (time windowed) sets of time series there is a potential role for the time series database in supporting analysis on or across time series. However, such use cases are seen as secondary and are not the primary focus of this investigation and report.

Consideration could be given to augmenting a data storage solution with an additional capability to perform more complex analytical operations on data retrieved from the store. Depending on need this might be used to enable a more powerful API to access the data or direct access to this capability might be available to some data users.

Analysis on a single time series might include aggregation queries (min, max, mean) to summarize a whole time series or generate a derived series (e.g. daily mean values).

There are potential use cases for query across time series. For example looking for time periods where there are correlated changes in measurements across different sites or from different sensors.

Solutions based on a relational database, such as TimescaleDB over Postresql, offer normal SQL capabilities for aggregate queries across a time series, windowed aggregation and join queries which span time series (e.g. TimescaleDB over Postgresql). Whereas noSQL solutions, such as Cassandra, typically support aggregation, but often do not support general joins. This is part of the trade off of such designs, achieving horizontal scaling through partitioning of data means that queries which require joins across partitions on different servers will be more expensive and complex to support.

One potential solution to this is to conduct any non-trivial analysis using an external analysis engine. Indeed, even where the database directly supports complex analytic queries it may be more appropriate to offload this processing so that large ad hoc queries don't impose extra load on the operational store and risk interfering with updates.

Some time series databases offer an integrated solution where an analysis engine is included alongside the database. For example, InfluxDB offers a built in scripting language (Flux) which offers both stream transforms and cross-stream joins via a functional language. Separately, a number of very flexible open source stream analysis engines have become available in recent years that could be used to complement a time series database installation with a scalable analytics capability. While not a focus of this report we note some of the offerings here for completeness:

Apache Spark

A very general purpose data analysis engine which scales transformation/analysis by parallelizing over the data, spreading the processing across a directed acyclic graph (DAG) of processing nodes. Supports a variety of programming models including a stream processing model (based on splitting a stream of data into "micro batches") as well as a data frame model with distributed SQL support. Supports joins across heterogeneous data sources.

The transforms and analysis can be written in Python and R, as well as Java and Scala, while still making use of the parallel engine. The Python support includes efficient connection to Pandas via an in-memory columnar data format (Apache Arrow).

Whilst Spark is part of the Apache Hadoop project it can be deployed standalone or on Kubernetes and does not necessarily require a full Hadoop cluster to run on.

Spark is a mature, well known and widely used framework. As a result there is a good range of connectors available. In particular there are connectors for Cassandra and relational databases (over jdbc) as well as support for Hadoop data stores (HBase, Hive and the full range of Hadoop file formats).

Apache Storm

Apache Storm is aimed at stream processing where one or more of the data sets are unbounded streams of events or measurements. Also distributes processing across a DAG of processing elements to achieve performance scaling but here it is the processing steps that are distributed, rather than distributing across the data. Data is processed in real time - as each row ("tuple") is injected into the system it flows through the processing graph immediately.

Of the pure distributed stream processing engines this is the oldest and most mature and reliable. It is primarily good for real time processing with relatively simple processing steps. It lacks any inbuilt state management or more sophisticated features such as windowing and aggregations. As such it is less relevant to sensor data analysis, though could in principle be relevant for processing high rate raw telemetry before arrival at the database.

Apache Flink

This is another distributed stream processing system aimed at real time event processing. Scalable with low latency and high throughput.

Much more recent than Storm and has a much richer feature set of sophisticated stream operations. Being newer has corresponding less adoption as yet but is reportedly used for a number of very challenging cases (at Uber and Alibaba).

Others

There are other frameworks for stream processing such as Kafka Streams and Apache Samza but these are more tied to specific message pipelines (particularly Apache Kafka) and are less relevant to the sensor network data.

Comment

Most of the innovation in this space appears to focus on stream processing for real time monitoring and analysis. While potentially relevant to handling of high volume raw sensor data these systems are less relevant for more sophisticated analysis of more modest rate time series and offline processing.

In some cases a non-parallel solution using standard Python packages or R may be sufficient and all time series databases should be accessible from such environments.

If the volume of data, the complexity of the analysis, or the number of analysis queries is likely to be problematic then, of the options briefly surveyed, Apache Spark looks the best match for this use case. The batch processing of Spark matches how the sensor data is likely to be processed while still supporting a stream programming model if preferred (through micro-batches). While not aimed at very low latency real time use it does offer good response times and is suitable for interactive queries as well as large scale batch analysis. It scales well, is widely used and supported and would be a safe choice.

Appendix 2 - Time Series Databases Encountered

Listed here are software packages and services that were encountered whilst preparing this report. They are not exhaustive lists, for example in memory only databases are excluded. More can be found in Survey and Comparison of Open Source Time Series Databases

Time Series Data Specific Databases

1. Actian X
2. Amazon Timestream
3. Akumuli
4. Apache Apex - not specifically time series
5. Apache Chukwa - focused on log file processing
6. Artic - sits on MongoDB - tickstore for financial series data - not updated in a few years
7. Atlas - Netflix - multi-dimensional time series data for near real time operational insight
8. Axibase - built on HBase - commerical - less popular than openTSDB
9. Beringei - in memory only time series database- open source version of Facebook Gorilla - developed because HBase backed TSDB would not scale to meet future requirements - stores the last 24 hours of Facebook IT monitoring data
10. Blueflood - not a lot of information or documentation
11. BtrDB - Berkley Tree Database for time series data - claims very high write rates ; integrates with Apache SPARK; requires Ubuntu! so not serious for production
12. Chronix Server - built on Apache SOLR
13. Cyanite - drop in replacement for Graphite
14. DalmatinerDB - built on Riak core; needs ZFS ;
15. eXtremeDB - embedded database in memory database extended with sharding and columnar layout
16. DolphinDB - SQL access; commercial ; not much info on the technology
17. EventQL - not updated in a while
18. FaunaDB - serverless - a multimodel database - but not really a TSDB
19. Gnocchi - web site not responding
20. Gorilla - in memory only time series database used by Facebook for system monitoring - see also Beringei
21. Graphite
22. GridDB - scale out like NoSQL - but with strong transaction guarantees - and built for speed
23. Hawkular Metrics - multi-tenanted; built on Cassandra; bit low level
24. Heroic - based (bigtable | cassandra) + elastic search
25. IBM Db2 Event Store - focus on ingest streamed data for event driven applications - not applicable
26. InfluxDB - popular time series database solution
27. IoTDB - internet of things - so aimed at lots of devices; built on hadoop; supports storage, query, visualization and analysis

28. IRONdb - focused on simple operations, resiliency, embedded analytics/computation
29. KairosDB - spin out of openTSBD to build on Cassandra rather than hbase; stores 3 weeks (hard coded) of metric data in one row
30. Kdb+ in memory
31. Kister Wiski time series database
32. M3DB from web page "M3DB not suitable for use as a general purpose time series database."
33. Machbase - focussed on IoT and log files; integrated search for e.g. searching for error messages in logs; sql query ;
34. MetricTank - focused on supporting the Grafana API; multi-tenanted; Casandra for storage; ElasticSearch for metadata
35. Newts - based on CasandraOpenTSDB based on HBase
36. QuasarDB high speed; combined in-memory and persistent; sql-like access; commercial
37. Rhombus - hasn't been updated in years
38. Riak TS - clustered based on Riak property value store
39. Roshi - hasn't been updated in years
40. RRDtool - round-robin database
41. SiriDB - custom solution written in C focussing on high performance;
42. SiteWhere - k8s deployment; multitenant; IoT platform - but little mention of time series - uses InfluxDB
43. Tgres - not actively maintained
44. TimelyDB - NSA - based on Accumulo - focus on security
45. TimescaleDB
46. Trendalyze - focused on spotting micro trends in financial data
47. TempoIQ - focus on creating no code dashboards, mobile apps and websites for IoT
48. VictoriaMetrics - a time series store for prometheus; prometheusQL ; compares itself to influxdb and timescaledb
49. Vulcan - no longer maintained
50. Warp 10 - time series database with geospatial - support for moving sensors; built on HBase or LevelDB
51. Yanza - web site returns blank page

Time Series Database Hosted Solutions

1. Amazon Timeseries

More General Databases Used for Time Series Data

1. Accumulo
2. Cassandra
3. ClickHouse
4. Druid
5. Elastic Search
6. EventQL
7. GridDB
8. HBase
9. MariaDB
10. MonetDB

11. MongoDB
12. Pinot
13. Scylla (faster C++ implementation of Cassandra)

DB-Engines List of Time Series Databases Ordered by Popularity

DB-Engines is a website that has a list of time series databases ordered by a popularity metric. The metric is based on:

- *number of results in search engine queries*
- *frequency of searches in Google Trends*
- *number of questions and interested in users on Stack Exchange and DBA Stack Exchange*
- *number of job offers on Indeed and Simply Hired*
- *number of profiles in LinkedIn and Upwork*
- *number of Twitter tweets mentioning the database*

This is what it showed in March 2020:

1. InfluxDB
2. Kdb+
3. Prometheus
4. Graphite
5. RRDtool
6. OpenTSDB
7. TimescaleDB
8. Druid
9. FaunaDB
10. KairosDB
11. GridDB
12. Alibaba Cloud TSDB
13. ExtremeDB
14. Amazon Timestream
15. DolphinDB
16. IBM DB2 Event Store
17. Riak TS
18. Heroic
19. Axibase
20. QuestDB
21. Warp 10
22. M3DB
23. QuasarDB
24. Blueflood
25. Victoria Metrics
26. Hawkular Metrics
27. Machbase
28. IRONdb
29. SiriDB
30. Newts

31. SiteWhere
32. Hyprcubd
33. Yanza