A peer-reviewed version of this preprint was published in PeerJ on 30 March 2016.

View the peer-reviewed version (peerj.com/articles/cs-52), which is the preferred citable publication unless you specifically need to cite this preprint.

MacDonald A. 2016. PhilDB: the time series database with built-in change logging. PeerJ Computer Science 2:e52
https://doi.org/10.7717/peerj-cs.52

PhilDB: The time series database with built-in change logging

Andrew MacDonald[1]

[1]**Melbourne, Australia**

ABSTRACT

PhilDB is an open-source time series database that supports storage of time series datasets that are dynamic, that is it records updates to existing values in a log as they occur. PhilDB eases loading of data for the user by utilising an intelligent data write method. It preserves existing values during updates and abstracts the update complexity required to achieve logging of data value changes. It implements fast reads to make it practical to select data for analysis.

Recent open-source systems have been developed to indefinitely store long-period high-resolution time series data without change logging. Unfortunately such systems generally require a large initial installation investment before use because they are designed to operate over a cluster of servers to achieve high-performance writing of static data in real time. In essence, they have a 'big data' approach to storage and access. Other open-source projects for handling time series data that avoid the 'big data' approach are also relatively new and are complex or incomplete. None of these systems gracefully handle revision of existing data while tracking values that changed. Unlike 'big data' solutions, PhilDB has been designed for single machine deployment on commodity hardware, reducing the barrier to deployment.

PhilDB takes a unique approach to meta-data tracking; optional attribute attachment. This facilitates scaling the complexities of storing a wide variety of data. That is, it allows time series data to be loaded as time series instances with minimal initial meta-data, yet additional attributes can be created and attached to differentiate the time series instances when a wider variety of data is needed. PhilDB was written in Python, leveraging existing libraries. While some existing systems come close to meeting the needs PhilDB addresses, none cover all the needs at once. PhilDB was written to fill this gap in existing solutions.

This paper explores existing time series database solutions, discusses the motivation for PhilDB, describes the architecture and philosophy of the PhilDB software, and performs a simple evaluation between InfluxDB, PhilDB, and SciDB.

Keywords: time series, database, logging, python, data science

1 INTRODUCTION

This paper will explore existing time series database solutions. It will examine the need for a liberally licensed, open-source, easily deployed time series database, that is capable of tracking data changes, and look at why the existing systems that were surveyed failed to meet these requirements. This paper will then describe the architecture and features of the new system, PhilDB, that was designed to meet these outlined needs. Finally, a simple evaluation will be performed to compare PhilDB to the most promising alternatives of the existing open-source systems.

2 BACKGROUND: EXISTING SYSTEMS

2.1 Proprietary systems

There are a number of proprietary solutions for storage of time series data that have been around since the mid-nineties to the early 2000s. Castillejos (2006) identified three proprietary systems of note, FAME, TimeIQ, and DBank, that have references that range from 1995 to 2000. There are other proprietary systems, such as kdb+[1], that are commercially available today. This shows that time series data storage is an existing problem. Compared to proprietary systems, open-source systems can generally be used with the scientific Python ecosystem as described by Perez et al. (2011). Ready access to open-source systems also make them easier to evaluate and integrate with. Therefore existing proprietary systems were not evaluated any further. Discussion on the need for an open-source system is further covered in section 3.

2.2 Open-source systems

In recent years the development of open-source time series databases has taken off, with most development beginning within the last five years. This can be seen by the number of projects discussed here along with noting the initial commit dates.

2.2.1 'Big data' time series databases

Some of the most successful projects in the open-source time series database space are OpenTSDB[2], Druid[3], Kairosdb[4], and InfluxDB[5]. The earliest start to development on these systems was for OpenTSDB with an initial commit in April 2010. These systems are designed to operate over a cluster of servers to achieve high-performance writing of static data in real time. In essence, they have a 'big data' approach to storage and access. The architectural approach to address big data requirements means a large initial installation investment before use.

2.2.2 Alternate time series databases

In contrast to the 'big data' time series systems some small dedicated open-source code bases are attempting to address the need for local or single server time series data storage. These systems, however, have stalled in development, are poorly documented,

[1] http://kx.com/software.php
[2] OpenTSDB initial commit: 2010-04-11; https://github.com/OpenTSDB/opentsdb
[3] Druid initial commit: 2012-10-24; https://github.com/druid-io/druid/
[4] Kairosdb initial commit: 2013-02-06; https://github.com/kairosdb/kairosdb
[5] InfluxDB initial commit: 2013-04-12; https://github.com/influxdb/influxdb

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1488v2 | CC-BY 4.0 Open Access | rec: 12 Feb 2016, publ: 12 Feb 2016

or require a moderate investment of time to operate. For example Timestore[6] was, at the time of writing, last modified August 2013 with a total development history of 36 commits. Some of the better progressed projects still only had minimal development before progress had ceased, for example tsdb[7] with a development start in January 2013 and the most recent commit at time of writing in February 2013 for a total of 58 commits. Cube[8] has a reasonable feature set and has had more development effort invested than the other systems discussed here, with a total of 169 commits, but it is no longer under active development according the Readme file. Searching GitHub for 'tsdb' reveals a large number of projects named 'tsdb' or similar. The most popular of these projects (when ranked by stars or number of forks) relate to the 'big data' systems described earlier (in particular, OpenTSDB, InfluxDB, and KairosDB). There are numerous small attempts at solving time series storage in simpler systems that fall short of a complete solutions. Of the systems discussed here only Cube had reasonable documentation, Timestore had usable documentation, and tsdb had no clear documentation.

2.2.3 Scientific time series databases

At present, the only open-source solution that addresses the scientific need to track changes to stored time series data as a central principle is SciDB (Stonebraker et al. 2009 and Stonebraker et al. 2011). SciDB comes with comprehensive documentation[9] that is required for such a feature rich system. The documentation is however lacking in clarity around loading data with most examples being based around the assumption that the data already exists within SciDB or is being generated by SciDB. While installation on a single server is relatively straight forward (for older versions with binaries supplied for supported platforms) the process is hard to identify as the community edition installation documentation is mixed in with the documentation on installation of the enterprise edition of SciDB. Access to source code is via tarballs; there is no source control system with general access to investigate the history of the project in detail.

3 MOTIVATION

The author's interest is derived from a need to handle data for exploratory purposes with the intention to later integrate with other systems, with minimal initial deployment overhead. It is assumed that the smaller time series database systems discussed previously derive from similar needs. The author has found "[m]ost scientists are adamant about not discarding any data" (Cudré-Mauroux et al. 2009). In particular, the author's experience in hydrology has found hydrological data requires the ability to track changes to it, since streamflow discharge can be regularly updated through quality control processes or updates to the rating curves used to convert from water level to discharge. Open-source 'big data' time series database offerings don't support the ability to track any changed values out of the box (such support would have to be developed external to the system). Their design targets maximum efficiency of write-once and read-many operations. When

[6]Timestore http://www.mike-stirling.com/redmine/projects/timestore; https://github.com/mikestir/timestore initial commit 2012-12-27

[7]tsdb initial commit: 2013-01-11; most recent commit at time of writing: 2013-02-17; https://github.com/gar1t/tsdb

[8]Cube initial commit: 2011-09-13; https://github.com/square/cube

[9]http://www.paradigm4.com/HTMLmanual/15.7/scidb_ug/

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1488v2 | CC-BY 4.0 Open Access | rec: 12 Feb 2016, publ: 12 Feb 2016

streamflow data is used within forecasting systems, changes to the data can alter the forecast results. Being able to easily identify if a change in forecast results is due to data or code changes greatly simplifies resolving issues during development and testing. Therefore, both requirements of minimal deployment overhead and logging of any changed values rule out the current 'big data' systems.

While SciDB does address the data tracking need, recent versions of the community edition are complex to install since they require building from source, a process more involved than the usual './configure; make; make install'. Older versions are more readily installed on supported platforms, however the system is still complex to use, requires root access to install, a working installation of PostgreSQL and a dedicated user account for running. Installation difficulty isn't enough to rule out the system being a suitable solution, but it does diminish its value as an exploratory tool. SciDB is also licensed under the GNU Affero General Public License (AGPL) that can be perceived as a problem in corporate or government development environments. In these environments integration with more liberally licensed (e.g. Apache License 2.0 or 3-clause BSD) libraries is generally preferred with many online discussions around the choice of liberal licences for software in the scientific computing space. For example, it can be argued that a simple liberal license like the BSD license encourages the most participation and reuse of code (Brown 2015, VanderPlas 2014, Hunter 2004).

Finally, SciDB has a broader scope than just storage and retrieval of time series data, since "SciDB supports both a functional and a SQL-like query language" (Stonebraker et al. 2011). Having SQL-like query languanges does allow for SciDB to readily support many high performance operations directly when handling large already loaded data. These query languages do, however, add additional cognitive load (Sweller et al. 2011) for any developer interfacing with the system as the query languages are specific to SciDB. If using SciDB for performing complex operations on very large multi-dimensional array datasets entirely within SciDB, learning these query languages would be well worth the time. The Python API does enable a certain level of abstraction between getting data out of SciDB and into the scientific Python ecosystem.

Of the other existing systems discussed here, none support logging of changed values. Limited documentation makes them difficult to evaluate, but from what can be seen and inferred from available information, the designs are targeted at the 'write once, read many' style of the 'big data' time series systems at a smaller deployment scale. These systems were extremely early in development or yet to be started at time the author began work on PhilDB in October 2013.

The need of the author is purely to store simple time series of floating point values and extract them again for processing with other systems.

3.1 Use case

To summarise, PhilDB has been created to provide a time series database system that is easily deployed, used, and has logging features to track any new or changed values. It has a simple API for writing both new and updated data with minimal user intervention. This is to allow for revising time series from external sources where the data can change over time, such as streamflow discharge data from water agencies. Furthermore, the simple API extends to reading, to enable easy retrieval of time series, including the ability to read time series as they appeared at a point in time from the logs.

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1488v2 | CC-BY 4.0 Open Access | rec: 12 Feb 2016, publ: 12 Feb 2016

4 ARCHITECTURE

PhilDB uses a central 'meta-data store' to track the meta information about time series instances. Relational databases are a robust and reliable way to hold related facts. Since the meta data is simply a collection of related facts about a time series, a relational database is used for the meta-data store. Time series instances are associated with a user chosen identifier and attributes and each time series instance is assigned a UUID (Leach et al. 2005) upon creation, all of which is stored in the meta-data store. The actual time series data (and corresponding log) is stored on disk with filenames based on the UUID (details of the format are discussed in section 5.2). Information kept in the meta-data store can then be used to look up the UUID assigned to a given time series instance based on the requested identifier and attributes. Once the UUID has been retrieved, accessing the time series data is a simple matter of reading the file from disk based on the expected UUID derived filename.

4.1 Architecture Philosophy

The reasoning behind this architectural design is so that:

* A simple to use write method can handle both new and updated data (at the same time if needed).
* Read access is fast and easy for stored time series.
* Time series are easily read as they appeared at a point in time.
* Each time series instance can be stored with minimal initial effort.

Ease of writing data can come at the expense of efficiency to ensure that create, update or append operations can be performed with confidence that any changes are logged without having to make decisions on which portions of the data are current or new. The expectation is that read performance has a greater impact on use as they are more frequent. Attaching a time series identifier as the initial minimal information allows for data from a basic dataset to be loaded and explored immediately. Additional attributes can be attached to a time series instance to further differentiate datasets that share conceptual time series identifiers. By default, these identifier and attribute combinations are then stored in a tightly linked relational database. Conceptually this meta data store could optionally be replaced by alternative technology, such as flat files. As the data is stored in individual structured files, the meta-data store acts as a minimal index with most of the work being delegated to the operating system and in turn the file system.

5 IMPLEMENTATION

PhilDB is written in Python because it fits well with the scientific computing ecosystem (Perez et al. 2011). The core of the PhilDB package is the PhilDB database class[10], that exposes high level methods for data operations. These high level functions are designed to be easily used interactively in the IPython interpreter (Perez and Granger 2007) yet still work well in scripts and applications. The goal of interactivity and scriptability are

[10] http://phildb.readthedocs.org/en/latest/api/phildb.html#module-phildb.database

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1488v2 | CC-BY 4.0 Open Access | rec: 12 Feb 2016, publ: 12 Feb 2016

to enable exploratory work and the ability to automate repeated tasks (Shin et al. 2011). Utilising Pandas (McKinney 2012) to handle complex time series operations simplifies the internal code that determines if values require creation or updating. Returning Pandas objects from the read methods allows for data analysis to be performed readily without further data munging. Lower level functions are broken up into separate modules for major components such as reading, writing, and logging, that can be easily tested as individual components. The PhilDB class pulls together the low level methods, allowing for the presentation of a stable interface that abstracts away the hard work of ensuring that new or changed values, and only those values, are logged.

Installation of PhilDB is performed easily within the Python ecosystem using the standard Python setup.py process, including installation from PyPI using 'pip'.

5.1 Features

Key features of PhilDB are:

* A single write method accepting a pandas.Series object, data frequency and attributes for writing or updating a time series.
* A read method for reading a single time series based on requested time series identifier, frequency and attributes.
* Advanced read methods for reading collections of time series.
* Support for storing regular and irregular time series.
* Logging of any new or changed values.
* Log read method to extract a time series as it appeared on a given date.

5.2 Database Format

The technical implementation of the database format, as implemented in version 0.6.1 of PhilDB (MacDonald 2015), is described in this section. Due to the fact that PhilDB is still in the alpha stage of development the specifics here may change significantly in the future.

The meta-data store tracks attributes using a relational database, with the current implementation using SQLite (Hipp et al. 2015). Actual time series data are stored as flat files on disk, indexed by the meta-data store to determine the path to a given series. The flat files are implemented as plain binary files that store a 'long', 'double', and 'int' for each record. The 'long' is the datetime stored as a 'proleptic Gregorian ordinal' as determined by the Python datetime.datetime.toordinal method[11] (van Rossum 2015). The 'double' stores the actual value corresponding to the datetime stored in the preceding 'long'. Finally, the 'int' is a meta value for marking additional information about the record. In this version of PhilDB the meta value is only used to flag missing data values. Individual changes to time series values are logged to HDF5 files (The HDF Group 1997) that are kept alongside the main time series data file with every new value written as a row in a table, each row having a column to store the date, value, and meta value as per the file format. In addition, a final column is included to record the date and time the record was written.

[11] https://docs.python.org/2/library/datetime.html#datetime.date.toordinal

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1488v2 | CC-BY 4.0 Open Access | rec: 12 Feb 2016, publ: 12 Feb 2016

6 EVALUATION

Of the open-source systems evaluated (as identified in section 2.2), InfluxDB came the closest in terms of minimal initial installation requirements and feature completeness, however, it doesn't support the key feature of update logging. Contrasting with InfluxDB, SciDB met the requirement of time series storage with update logging but didn't meet the requirement for simplicity to deploy and use. Both these systems were evaluated in comparison to PhilDB.

To simplify the evaluation process and make it easily repeatable, the SciDB 14.3 virtual appliance image[12] was used to enable easy use of the SciDB database. This virtual appliance was based on a CentOS Linux 6.5 install. The PhilDB and InfluxDB databases were installed into the same virtual machine to enable comparison between systems. The virtual machine host was a Mid-2013 Apple Macbook Air, with a 1.7 GHz Intel Core i7 CPU, 8GB of DDR3 RAM and a 500GB SSD hard drive. VirtualBox 4.3.6 r91406 was used on the host machine for running the virtual appliance image with the guest virtual machine being allocated 2 processors and 4GB of RAM.

Write performance was evaluated by writing all time series from the evaluation dataset (described in section 6.1) into the time series databases being evaluated. This first write will be referred to as the initial write for each database. To track the performance of subsequent updates and reading the corresponding logged time series a further four writes were performed. These writes will be referred to as 'first update' through to 'fourth update'. The update data was created by multiplying some or all of the original time series by 1.1 as follows:

* First update: multiplied the last 10 values in the time series by 1.1 leaving the rest of the record the same.

* Second update: multiplied the first 10 values by 1.1, resulting in reverting the previously modified 10 values.

* Third update: multiplied the entire original series by 1.1 resulting in an update to all values aside from the first 10.

* Fourth update: the original series multiplied by 1.1 again, which should result in zero updates.

The SciDB load method used in this experiment did not support updating individual values. The entire time series needed to be passed or the resulting array would consist of only the supplied values. Due to this only full updates were tested and not individual record updates or appends.

Performance reading the data back out of each database system was measured by recording the time taken to read each individual time series, after each update, and analysing those results.

As can be seen by figure 1, InfluxDB performance was a long way behind SciDB and PhilDB. Given the performance difference and that InfluxDB doesn't support change logging only the initial load and first read were performed for InfluxDB.

[12] https://downloads.paradigm4.com/QuickStart/14.3/SciDB14.3-CentOS6-VirtualBox-4.2.10.ova

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1488v2 | CC-BY 4.0 Open Access | rec: 12 Feb 2016, publ: 12 Feb 2016

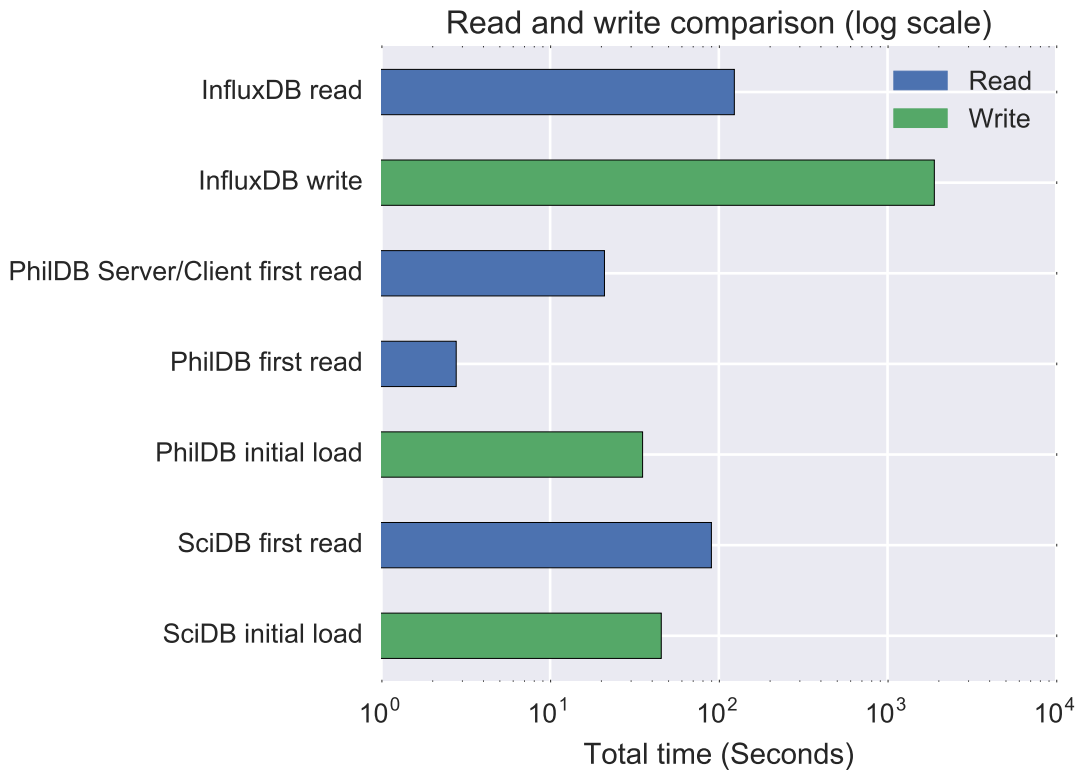


Figure 1. Mean write/read time for 221 daily time series

Disk usage was measured by recording the size of the data directories as reported by the 'du' Unix command. The size of the data directory was measured before loading any data and subtracted from subsequent sizes. Between each data write (initial load and four updates) the disk size was measured to note the incremental changes.

For both PhilDB and SciDB the evaluation process described in this section was performed four times and the mean of the results analysed. Results between the four runs were quite similar so taking the mean gave results similar to the individual runs. Analysing and visualising an individual run rather than the mean would result in the same conclusions.

6.1 Evaluation dataset

The Hydrological Reference Stations (Zhang et al. 2014) dataset from the Australian Bureau of Meteorology[13] was used for the evaluation. This dataset consists of daily streamflow data for 221 time series with a mean length of 16,310 days, the breakdown of the series lengths are in table 1 and visualised in figure 2.

6.2 InfluxDB

Paul Dix (CEO of InfluxDB) found that performance and ease of installation were the main concerns of users of existing open-source time series database systems (Dix 2014). InfluxDB was built to alleviate both those concerns.

[13] http://www.bom.gov.au/water/hrs/

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1488v2 | CC-BY 4.0 Open Access | rec: 12 Feb 2016, publ: 12 Feb 2016

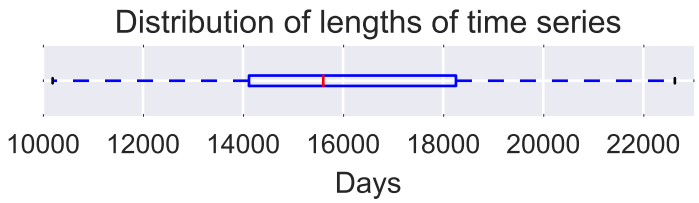


Figure 2. Distribution of time series length for the 221 time series in the evaluation dataset

mean	16310 days
std	2945 days
min	10196 days
25%	14120 days
50%	15604 days
75%	18256 days
max	22631 days

Table 1. Breakdown of length of time series in sample dataset (all values rounded to nearest day)

While InfluxDB is designed for high performance data collection, it is not designed for bulk loading of data. Searching the InfluxDB issue tracker on github[14], it can be seen that bulk loading has been a recurring problem with improvement over time. Bulk loading performance is, however, still poor compared to SciDB and PhilDB, as seen later in the performance results (section 6.5). A key feature of interest with InfluxDB was the ability to identify time series with tags. This feature is in line with the attributes concept used by PhilDB, thereby allowing multiple time series to be grouped by a single key identifier but separated by additional attributes or tags.

6.2.1 Installation

InfluxDB is easily installed compared to the other open-source systems reviewed, as demonstrated by the short install process shown below. Installation of pre-built packages on Linux requires root access[15]. Installation of InfluxDB was performed in the CentOS Linux 6.5 based virtual machine containing the pre-installed SciDB instance.

```
wget http://influxdb.s3.amazonaws.com/influxdb-0.9.6.1-1.x86_64.rpm
sudo yum localinstall influxdb-0.9.6.1-1.x86_64.rpm
```

Starting the InfluxDB service with:

```
sudo /etc/init.d/influxdb start
```

6.2.2 Usage

Loading of data into the InfluxDB instance was performed using the InfluxDB Python API that was straight forward to use. However, poor performance of bulk loads lead to a

[14]https://github.com/influxdata/influxdb/issues
[15]https://influxdb.com/docs/v0.9/introduction/installation.html

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1488v2 | CC-BY 4.0 Open Access | rec: 12 Feb 2016, publ: 12 Feb 2016

lot of experimentation on how to most effectively load large amounts of data quickly, including trying curl and the Influx line protocol format directly. The final solution used was to chunk the data into batches of 10 points using the Pandas groupby functionality before writing into InfluxDB using the InfluxDB Python API DataFrameClient write_points method, for example:

```
streamflow = pandas.read_csv(filename, parse_dates=True, index_col=0, header = None)
for k, g in streamflow.groupby(np.arange(len(streamflow))//100):
    influx_client.write_points(g, station_id)
```

In addition to experimenting with various API calls, configuration changes were attempted resulting in performance gains by lowering values related to the WAL options (the idea was based on an older GitHub issue discussing batch loading[16] and WAL tuning to improve performance). Despite all this effort, bulk data loading with InfluxDB was impractically slow with a run time generally in excess of one hour to load the 221 time series (compared to the less than 2 minutes for SciDB and PhilDB). Reading was performed using the Python API InfluxDBClient query method:

```
streamflow = influx_client.query('SELECT * FROM Q{0}'.format('410730'))
```

6.3 PhilDB

PhilDB has been designed with a particular use case in mind as described in section 3.1. Installation of PhilDB is quite easy where a compatible Python environment exists. Using a Python virtualenv removes the need to have root privileges to install PhilDB and no dedicated user accounts are required to run or use PhilDB. A PhilDB database can be written to any location the user has write access, allowing for experimentation without having to request a database be created or needing to share a centralised install.

6.3.1 Installation

Installation of PhilDB is readily performed using pip:

```
pip install phildb
```

6.3.2 Usage

The experimental dataset was loaded into a PhilDB instance using a Python script. Using PhilDB to load data can be broken into three key steps.

First, initialise basic meta information:

```
db.add_measurand('Q', 'STREAMFLOW', 'Streamflow')
db.add_source('BOM_HRS', 'Bureau of Meteorology; Hydrological Reference Stations
    dataset.')
```

This step only need to be performed once, when configuring attributes for the PhilDB instance for the first time, noting additional attributes can be added later.

Second, add an identifier for a time series and a time series instance record based on the identifier and meta information:

```
db.add_timeseries(station_id)
db.add_timeseries_instance(station_id, 'D', '', measurand = 'Q', source = 'BOM_HRS')
```

Multiple time series instances, based on different combinations of attributes, can be associated with an existing time series identifier. Once a time series instance has been created it can be written to and read from.

Third, load the data from a Pandas time series:

[16] https://github.com/influxdata/influxdb/issues/3282

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1488v2 | CC-BY 4.0 Open Access | rec: 12 Feb 2016, publ: 12 Feb 2016

```
streamflow = pandas.read_csv(filename, parse_dates=True, index_col=0, header = None)
db.write(station_id, 'D', streamflow, measurand = 'Q', source = 'BOM_HRS')
```

In this example the Pandas time series is acquired by reading a CSV file using the Pandas read_csv method, but any data acquisition method that forms a Pandas.Series object could be used. Reading a time series instance back out is easily performed with the read method:

```
streamflow = db.read(station_id, 'D', measurand = 'Q', source = 'BOM_HRS')
```

The keyword arguments are optional provided the time series instance can be uniquely identified.

6.4 SciDB

SciDB, as implied by the name, was designed with scientific data in mind. As a result SciDB has the feature of change logging, allowing past versions of series to be retrieved. Unfortunately SciDB only identifies time series by a single string identifier, therefore storing multiple related time series would require externally managed details about what time series are stored and with what identifier. Due to the sophistication of the SciDB system it is relatively complex to use with two built in languages, AFL and AQL, that allow for two different approaches to performing database operations. This, in turn, increases the amount of documentation that needs to be read to identify which method to use for a given task (such as writing a time series into the database). While the documentation is comprehensive in detailing the available operations, it is largely based on the assumption that the data is already within SciDB and will only be operated on within SciDB, with limited examples on how to load or extract data via external systems.

6.4.1 Installation

SciDB does not come with binary installers for newer versions and the build process is quite involved. Instructions for the build proccess are only available from the SciDB forums using a registered account[17]. Installation of older versions is comparable to InfluxDB with the following steps listed in the user guide:

```
yum install -y https://downloads.paradigm4.com/scidb-14.12-repository.rpm
yum install -y scidb-14.12-installer
```

Same as InfluxDB, SciDB requires root access to install and a dedicated user account for running the database. A PostgreSQL installation is also required by SciDB for storing information about the time series data that SciDB stores. Unlike InfluxDB, SciDB has authentication systems turned on by default that requires using dedicated accounts even for basic testing and evaluation.

Only Ubuntu and CentOS/RHEL Linux variants are listed as supported platforms in the install guide.

6.4.2 Usage

It took a considerable amount of time to identify the best way to load data into a SciDB instance, however once that was worked out, the actual load was quick and effective consisting of two main steps.

First, a time series needs to be created:

[17] http://paradigm4.com/forum/viewtopic.php?f=14&t=1672&sid=6e15284d9785558d5590d335fed0b059

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1488v2 | CC-BY 4.0 Open Access | rec: 12 Feb 2016, publ: 12 Feb 2016

```
iquery -q "CREATE ARRAY Q${station} <date:datetime, streamflow:double> [i
    =0:*,10000,0];"
```

It is worth noting that datetime and double need to be specified for time series storage, since SciDB can hold many different array types aside from a simple time series. Additionally, SciDB identifiers can not start with a numeric character so all time series identifiers were prefixed with a 'Q' (where 'Q' was chosen in this case because it is conventionally used in the hydrological context to represent streamflow discharge).

Second, the data is written using the iquery LOAD method as follows:

```
iquery -n -q "LOAD Q${station} FROM '/home/scidb/${station}.scidb';"
```

This method required creating data files in a specific SciDB text format before hand using the csv2scidb command that ships with SciDB.

Identifying the correct code to read data back out required extensive review of the documentation, but was quick and effective once the correct code to execute was identified. The SciDB Python code to read a time series back as a Pandas.DataFrame object is as follows:

```
streamflow = sdb.wrap_array('Q' + station_id).todataframe()
```

A contributing factor to the difficulty of identifying the correct code is that syntax errors with the AQL based queries (using the SciDB iquery command or via the Python API) are at times uninformative about the exact portion of the query that is in error.

6.5 Performance

It should be noted that PhilDB currently only supports local write, which is advantageous for performance, compared to InfluxDB that only supports network access. InfluxDB was hosted locally, which prevents network lag, but the protocol design still reduced performance compared to the direct write as done by PhilDB. Although SciDB has network access, only local write performance (using the SciDB iquery command) and network based read access (using the Python API) were evaluated. SciDB was also accessed locally to avoid network lag when testing the network based API. For a comparable network read access comparison the experimental PhilDB Client/Server software was also used.

6.5.1 Write performance

Write performance was measured by writing each of the 221 time series into the database under test and recording the time spent per time series.

As can be seen in figure 1, SciDB and PhilDB have a significant performance advantage over InfluxDB for bulk loading of time series data. SciDB write performance is comparable to PhilDB, so a closer comparison between just SciDB and PhilDB write performance is shown in figure 3.

It can be seen that while PhilDB has at times slightly better write performance, SciDB has more reliable write performance with a tighter distribution of write times. It can also be seen from figure 3 that write performance for SciDB does marginally decrease as more updates are written. PhilDB write performance while more variable across the dataset is also variable in performance based on how much of the series required updating. Where the fourth update writes the same data as the third update it can be seen that the performance distribution is closer to that of the initial load than the third load, since the data has actually remained unchanged.

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1488v2 | CC-BY 4.0 Open Access | rec: 12 Feb 2016, publ: 12 Feb 2016

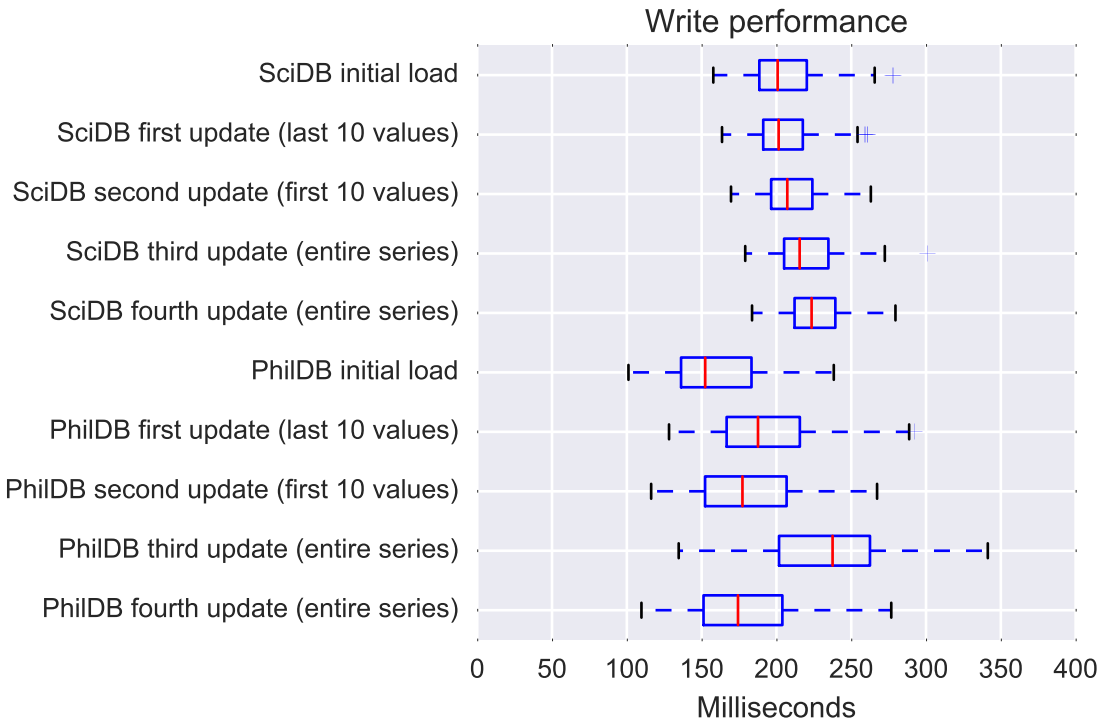


Figure 3. Distribution of write times for 221 time series

Both SciDB and PhilDB perform well at loading datasets of this size with good write performance.

6.5.2 Read performance

InfluxDB read performance is adequate and SciDB read speed is quite good, however PhilDB significantly out-performs both InfluxDB and SciDB in read speed, as can be seen in figure 1. Even the PhilDB server/client model, which has yet to be optimised for performance, out-performed both InfluxDB and SciDB. Read performance with PhilDB is consistent as the time series are updated, as shown in figure 4, due to the architecture keeping the latest version of time series in a single file. Reading from the log with PhilDB does show a decrease in performance as the size of the log grows, but not as quickly as SciDB. While PhilDB maintains consistent read performance and decreasing log read performance, SciDB consistently decreases in performance with each update for reading both current and logged time series.

6.5.3 Disk usage

After the initial load InfluxDB was using 357.21 megabytes of space. This may be due to the indexing across multiple attributes to allow for querying and aggregating multiple time series based on specified attributes. This is quite a lot of disk space being used compared to SciDB (93.64 megabytes) and PhilDB (160.77 megabytes) after the initial load. As can be seen in figure 5, SciDB disk usage increases linearly with each update when writing the entire series each time. In contrast, updates with PhilDB only result in moderate increases and depends on how many values are changed. If the time

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1488v2 | CC-BY 4.0 Open Access | rec: 12 Feb 2016, publ: 12 Feb 2016

Figure 4. Distribution of read durations for the 221 time series from the evaluation dataset

series passed to PhilDB for writing is the same as the already stored time series then no changes are made and the database size remains the same, as can be seen between update 3 and 4 in figure 5.

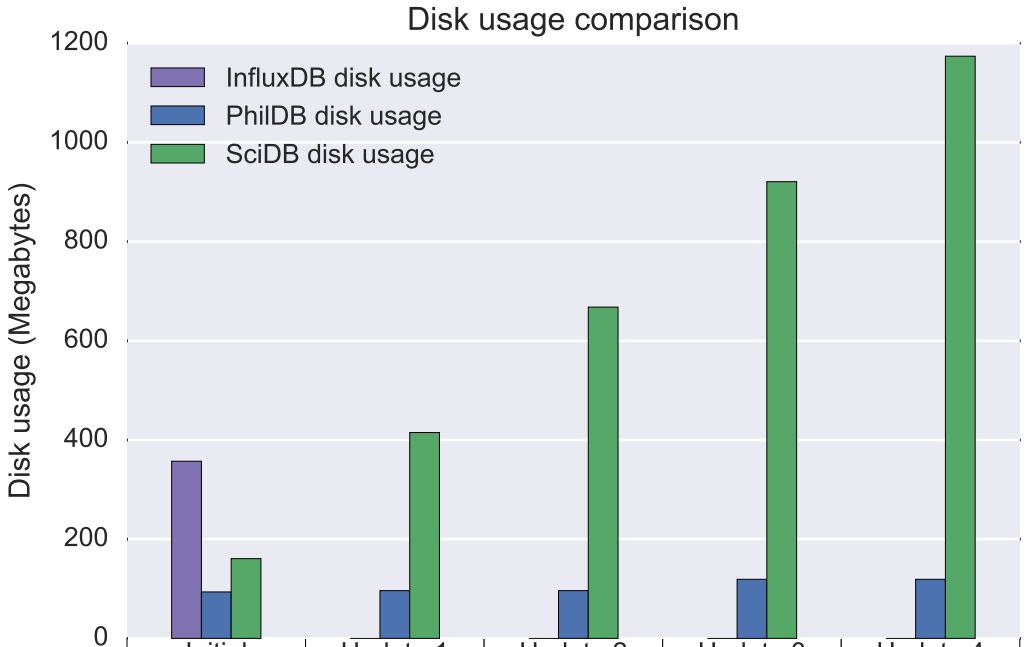
6.5.4 Performance summary

Each database has different design goals that results in different performance profiles. InfluxDB is not well suited to this use case with a design focusing on high performance writing of few values across many time series for metric collection, leading to poor performance for bulk loading of individual time series.

SciDB fares much better with consistent read and write performance, with slight performance decreases as time series are updated, likely due to design decisions that focus on handling large multi-dimensional array data for high performance operations. Design decisions for SciDB that lead to consistent read and write performance appear to also give the same read performance when accessing historical versions of time series. Achieving consistent read and write performance (including reading historical time series) seems to have come at the expense of disk space with SciDB consuming more space than PhilDB and increasing linearly as time series are updated.

PhilDB performs quite well for this particular use case, with consistently fast reads of the latest time series. This consistent read performance does come at the expense of reading historical time series from the logs, which does degrade as the logs grow. Write performance for PhilDB, while variable, varies due to the volume of data changing.

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1488v2 | CC-BY 4.0 Open Access | rec: 12 Feb 2016, publ: 12 Feb 2016



	Initial	Update 1	Update 2	Update 3	Update 4
InfluxDB disk usage	357.21	nan	nan	nan	nan
PhilDB disk usage	93.64	96.26	96.28	119.12	119.12
SciDB disk usage	160.77	415.05	667.91	920.79	1173.74

Figure 5. Disk usage after initial data load and each subsequent data update

The performance of PhilDB (particularly the excellent read performance) compared to SciDB for this use case was unexpected since the design aimed for a simple API at the expense of efficiency.

7 FUTURE WORK

PhilDB is still in its alpha stage. Before reaching the beta stage, the author shall investigate:

* Complete attribute management to support true arbitrary attribute creation and attachment.

* Possible alternative back ends, using alternative data formats, disk paths, and relational databases.

* More sophisticated handling of time zone meta-data.

* Storage of quality codes or other row level attributes.

* Formalisation of UUID usage for sharing of data.

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1488v2 | CC-BY 4.0 Open Access | rec: 12 Feb 2016, publ: 12 Feb 2016

8 CONCLUSION

In conclusion, there is a need for an accessible time series database that can be deployed quickly so that curious minds, such as those in our scientific community, can easily analyse time series data and elucidate world-changing information. For scientific computing, it is important that any solution is capable of tracking subsequent data changes.

Although InfluxDB comes close with features like tagging of attributes and a clear API, it lacks the needed change logging feature and presently suffers poor performance for bulk loading of historical data. InfluxDB has clearly been designed with real-time metrics based time series in mind and as such doesn't quite fit the requirements outlined in this paper.

While SciDB has the important feature of change logging and performs quite well, it doesn't have a simple mechanism for tracking time series by attributes. SciDB is well suited for handing very large multi-dimensional arrays, which can justify the steep learning curve for such work, but for simple input/output of plain time series such complexity is a little unnecessary.

PhilDB addresses this gap in existing solutions, as well as surpassing them for efficiency and usability. Finally, PhilDB's source code has been released on GitHub[18] under the permissive 3-clause BSD open-source license to help others easily extract wisdom from their data.

9 ACKNOWLEDGEMENTS

I would like to thank Di MacDonald for her editorial advice on early drafts, my fiancée Katrina Cornelly for her support and editorial advice, and my colleague Richard Laugesen for his valuable review comments on an earlier draft. PhilDB was named in memory of my father Phillip MacDonald.

[18] https://github.com/amacd31/phildb

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1488v2 | CC-BY 4.0 Open Access | rec: 12 Feb 2016, publ: 12 Feb 2016

REFERENCES

Brown, C. T. (2015). On licensing bioinformatics software: use the BSD, Luke. http://ivory.idyll.org/blog/2015-on-licensing-in-bioinformatics.html.

Castillejos, A. M. (2006). *Management of Time Series Data*. PhD thesis, University of Canberra.

Cudré-Mauroux, P., Kimura, H., Lim, K.-T., Rogers, J., Simakov, R., Soroush, E., Velikhov, P., Wang, D. L., Balazinska, M., Becla, J., and others (2009). A demonstration of SciDB: a science-oriented DBMS. *Proceedings of the VLDB Endowment*, 2(2):1534–1537.

Dix, P. (2014). InfluxDB: One year of InfluxDB and the road to 1.0. https://influxdb.com/blog/2014/09/26/one-year-of-influxdb-and-the-road-to-1_0.html.

Hipp, D. R., Kennedy, D., and Mistachkin, J. (2015). SQLite. https://www.sqlite.org/download.html.

Hunter, J. D. (2004). Why we should be using BSD. http://nipy.sourceforge.net/nipy/stable/faq/johns_bsd_pitch.html.

Leach, P. J., Mealling, M., and Salz, R. (2005). A Universally Unique IDentifier (UUID) URN Namespace.

MacDonald, A. (2015). phildb: Version 0.6.1. http://dx.doi.org/10.5281/zenodo.32437.

McKinney, W. (2012). *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. "O'Reilly Media, Inc.".

Perez, F. and Granger, B. E. (2007). IPython: a system for interactive scientific computing. *Computing in Science & Engineering*, 9(3):21–29.

Perez, F., Granger, B. E., and Hunter, J. D. (2011). Python: an ecosystem for scientific computing. *Computing in Science & Engineering*, 13(2):13–21.

Shin, D., Schepen, A., Peatey, T., Zhou, S., MacDonald, A., Chia, T., Perkins, J., and Plummer, N. (2011). WAFARi: A new modelling system for Seasonal Streamflow Forecasting service of the Bureau of Meteorology, Australia. In *MODSIM 2011, Modelling and Simulation Society of Australian and New Zealand*.

Stonebraker, M., Becla, J., DeWitt, D. J., Lim, K.-T., Maier, D., Ratzesberger, O., and Zdonik, S. B. (2009). Requirements for Science Data Bases and SciDB. In *CIDR*, volume 7, pages 173–184.

Stonebraker, M., Brown, P., Poliakov, A., and Raman, S. (2011). The Architecture of SciDB. In Cushing, J. B., French, J., and Bowers, S., editors, *Scientific and Statistical Database Management*, number 6809 in Lecture Notes in Computer Science, pages 1–16. Springer Berlin Heidelberg.

Sweller, J., Ayres, P. L., and Kalyuga, S. (2011). *Cognitive load theory*. Explorations in the learning sciences, instructional systems and performance technologies. Springer, New York.

The HDF Group (1997). Hierarchical Data Format, version 5. http://www.hdfgroup.org/HDF5/.

van Rossum, G. (2015). Python Programming Language. www.python.org.

VanderPlas, J. (2014). The Whys and Hows of Licensing Scientific Code. http://www.astrobetter.com/blog/2014/03/10/the-whys-and-hows-of-licensing-scientific-code/.

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1488v2 | CC-BY 4.0 Open Access | rec: 12 Feb 2016, publ: 12 Feb 2016

Zhang, S. X., Bari, M., Amirthanathan, G., Kent, D., MacDonald, A., Shin, D., and others (2014). Hydrologic reference stations to monitor climate-driven streamflow variability and trends.

PeerJ PrePrints | https://doi.org/10.7287/peerj.preprints.1488v2 | CC-BY 4.0 Open Access | rec: 12 Feb 2016, publ: 12 Feb 2016