Peregreen – modular database for efficient storage of historical time series in cloud environments

Alexander Visheratin, Alexey Struckov, Semen Yufa, Alexey Muratov, Denis Nasonov, and Nikolay Butakov, *ITMO University;* Yury Kuznetsov and Michael May, *Siemens*

https://www.usenix.org/conference/atc20/presentation/visheratin

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Peregreen – modular database for efficient storage of historical time series in cloud environments

Alexander A. Visheratin
ITMO University

Alexey Struckov
ITMO University

Semen Yufa
ITMO University

Alexey Muratov
ITMO University

Denis Nasonov
ITMO University

Nikolay Butakov
ITMO University

Yury Kuznetsov
Siemens

Michael May
Siemens

Abstract

The rapid development of scientific and industrial areas, which rely on time series data processing, raises the demand for storage that would be able to process tens and hundreds of terabytes of data efficiently. And by efficiency, one should understand not only the speed of data processing operations execution but also the volume of the data stored and operational costs when deploying the storage in a production environment such as cloud.

In this paper, we propose a concept for storing and indexing numeric time series that allows creating compact data representations optimized for cloud storages and perform typical operations – uploading, extracting, sampling, statistical aggregations, and transformations – at high speed. Our modular database that implements the proposed approach – Peregreen – can achieve a throughput of 3 million entries per second for uploading and 48 million entries per second for extraction in Amazon EC2 while having only Amazon S3 as storage backend for all the data.

1 Introduction

Time series data plays a very important role in the modern world. Many fields of science and industry rely on storing and processing large amounts of time series – economics and finances [6], medicine [7], Internet of Things [10], environmental protection [11], hardware monitoring [9] and many others. Growing industry demand for functional capabilities and processing speed led to the sharp rise of specialized time series databases TSDB [16] and development of custom TSDB solutions by large companies, e.g. Gorilla [18] by Facebook and Atlas by Netflix.

One of the most challenging applications of TSDBs, which becomes especially demanded by industrial companies like Siemens, is processing of long periods of historical time series data. Historical data can contain tens and hundreds terabytes of data thus making usage of in-memory databases too expensive. Standard solutions like relational databases or key-value storages struggle to efficiently process such large amounts of time series data [8, 23]. Thus it is very important for a TSDB to achieve the best balance between execution speed and operational cost.

A platform for historical time series analysis groups the data by sensors – modules responsible for capturing one specific phenomenon, e.g. tickers in stock market, temperature sensor for industrial machines or pulse sensor in a hospital equipment. There can be distinguished several typical scenarios of users interaction with the platform for historical data analysis. In the first case, user browses the whole dataset containing millions of sensors and years of data. User can select any sensor to investigate its behavior at any period of time. The platform must visualize the data for the user and allow to quickly zoom in and out, e.g. from years interval into months into days and hours and backwards. To successfully perform in this scenario, the underlying time series storage must provide fast access to any part of the data and be able to extract aggregated values when visualizing long time intervals.

In the second case, user analyzes a specific sensor and wants to look into time intervals, in which the value satisfies some condition, for example amplitude search or outliers search. For this user first searches for all time intervals meeting the condition and after that he or she selects the interval of interest and the platform visualizes the data for this interval. For this scenario the underlying time series storage must be able to perform fast search for time intervals in the whole volume of data and perform fast extraction – loading time series for a specified period – along with aggregation (e.g. average and standard deviation) and transformation (e.g. moving average).

Another important feature for any data processing system is an ability to use it in cloud platforms, like Amazon Web Services, Google Cloud or Microsoft Azure, since they allow the system to be globally available and increase the number of potential users and customers. Considering this the target TSDB must provide a good compression of the raw data along with the small internal footprint to minimize the cost of storing the data in cloud platform. Additional advantage for the target TSDB would be an integration with cheaper data storage cloud services, e.g. Amazon S3 or Google Storage.

Summarizing the above, we faced following challenging requirements for the target time series storage in the beginning of the joint project between Siemens and ITMO University:

1. Store terabytes of time series data for tens of years.
2. 1 second for conditional search in whole database.
3. Execution speed of 450,000 entries per second per node.
4. All data must be stored in Amazon S3.

To address described above requirements we propose an approach for storing numeric time series that is based on twofold split of the data into segments and blocks. Statistical information about segments and blocks is incorporated into three-tier indices, which are independent for every stored sensor. For building an internal data representation we use read-optimized series encoding that is based on delta encoding and Zstandard algorithm and achieves up to 6.5x compression ratio.

We also present design and implementation of the modular distributed database based on the proposed concept. The database is called Peregreen in honor of the fastest bird on Earth. In the experimental studies we demonstrate that Peregreen outperforms leading solutions – InfluxDB and ClickHouse – in the described above use cases. We also show that our solution is able to operate fast in the Amazon EC2 environment while storing all indices and data in Amazon S3 object storage.

Currently Peregreen is being integrated into internal projects of Siemens related to real-time analysis of historical data and development of complex workflows for processing long time series.

2 Proposed approach

The main challenge of development a database that would operate in a cloud environment and store its data solely in a remote object storage is minimizing network overheads. The first aspect of this problem is that there is no reasonable way to perform scans on database files to find appropriate data for the request. Because of that the index of the database has to contain metadata that would allow to navigate to exact parts of database files where the data is located. The second aspect is that we need to minimize the amount of network connections required for data uploading and extraction. For achieving this goal there is a need to develop a format for database files that would allow writing and reading all required data within one request. And the third aspect is minimizing sizes of both database and index files. The smaller files are the cheaper their storage is. And additional advantage of small index files is that more indices can be loaded into the RAM of the processing node.

In this section we give a detailed description of our approach for indexing and storing time series data for fast operation in cloud environments. Target use cases of the initial project included processing of numeric time series that means sequences of pairs *(timestamp,value)* without any tags. To achieve high speed of parallel uploads and ease of operation, every sensor has its own index and processed independently from others. Input time series data is indexed via three-tier data indexing mechanism and converted into the internal representation using read-optimized series encoding. Details on how this approach helps to optimize data processing and fit to the project requirements are given in Section 3.1.

2.1 Three-tier data indexing

The first part of the approach is a three-tier data indexing. Schema of the indexing is presented in Figure 1. At the first step the input data is divided into data chunks – subsets containing *(timestamp,value)* pairs. Time interval for data chunks split is a parameter called block interval. After that for every data chunk we create an index block that contains meta information required for data extraction and a set of metrics – aggregated statistics for the data chunk, e.g. minimum, maximum or average. Metrics are used for conditional search for time intervals in the index, details on it are given in Section 3.1. Block meta includes following fields: number of elements in the data chunk, length of the encoded representation (more details on it in Section 2.2), and a flag whether encoded representation is compressed.

Index blocks are incorporated into an index segment. Blocks in the segment are located in the chronological order, which allows fast calculation of required parts of the segment file as will be described further. During segment creation we initialize all the blocks in it empty, and all changes modify the existing blocks.

The time interval that every segment covers is set by a parameter called segment interval. Every segment has its meta information and a set of metrics. Segment meta contains a unique identifier (ID), start time, which is calculated during segment creation, and a version of the segment. Segment ID sets the name of the segment binary file in the storage backend. The presence of the segment start time makes possible fast and compact encoding of timestamps in blocks as described in Section 2.2.

The start time for a new segment is calculated according to the first timestamp of the indexed data chunk and existing segments in the index. If there are no segments in the index, the start time of the segment is set to the first timestamp of the data chunk. Otherwise, the start time of the segment is calculated so that it is separated by a multiple of the segment intervals from other segments and is as close as possible from the bottom to the first timestamp in the data chunk. The version of the segment is an integer number that increments every time the segment is changed.

Segment metrics are calculated by combining metrics of its blocks. There are two types of metrics – cumulative and non-cumulative. The difference between them is that for combining cumulative metrics there is no need to have all values

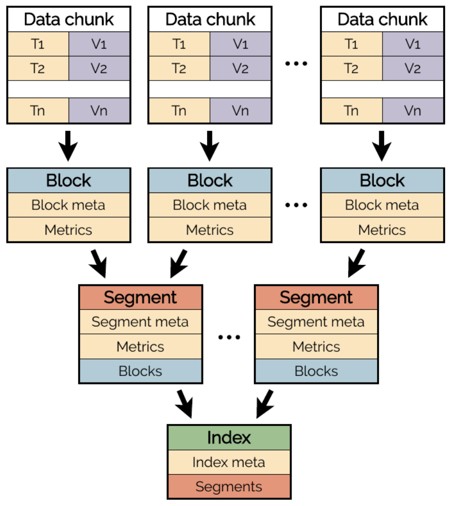

Figure 1: Schema of three-tier data indexing

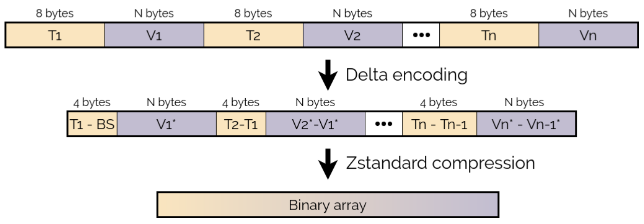

Figure 2: Schema of read-optimized series encoding

based on which these metrics were calculated. For example, minimum and maximum are cumulative metrics, and standard deviation and median are non-cumulative. For cumulative metrics combining function uses only their values while for non-cumulative metrics it uses all input values used to calculate these metrics. Metric instances of blocks and segments store input values during data uploading and delete them afterwards.

Segments are then incorporated into an index. Segments in the index are located in the chronological order. Index meta information includes unique identifier (name of the corresponding sensor), data type, segment interval, block interval and version. Data type determines the way how values are processed during data encoding. Supported data types are 8-bit integer, 16-bit integer, 32-bit integer, 64-bit integer, 32-bit float and 64-bit float. Block and segment intervals define time intervals for blocks and segments generation as it was discussed above. The version of the index is an integer number that increments every time the index is changed.

Indices are stored in the storage backend as individual files. File names for each individual index consist of index ID and index version.

2.2 Read-optimized series encoding

Since one of the most challenging requirements for our database was storing all data in Amazon S3, we needed an internal representation format that would allow performing data uploading and extraction in the best possible way. It is well known that one of the main factors that affect the speed of interaction with remote services is the network (connections instantiating, latency, etc.). That is why we aimed to create a format that would minimize the number of requests to the storage backend during extraction as much as possible.

The schema of the developed read-optimized encoding is presented in Figure 2. As the input we have a sequence of *(timestamp,value)* pairs from the data chunk. Timestamps are 64-bit integers and thus have the size of 8 bytes. The binary size of values **N** depends on the underlying data type and thus can be from 1 to 8.

The first part of the series processing is the delta encoding – instead of timestamps and values we store differences between current and previous items. It allows to reduce the amount of data required for storing timestamps from 8 to 4 bytes by assuming that difference between two consecutive timestamps would not be greater than ≈ 49 days ($(2^{32}-1)$ ms, maximum value of unsigned 32-bit integer). Delta for the first timestamp is calculated as a difference between the timestamp and block start (**BS** in the schema) that is calculated from the segment start time and the order of the block corresponding to the data chunk. The way how the algorithm generates stored differences for values depends on the processed data type: for integers diff is calculated as a subtraction of the next value from the previous one, and for floating point values a difference is generated as a difference between IEEE 754 binary representations of consecutive values, because such approach produces longer zero-filled bit sequences and can be compressed better.

At the second step calculated deltas are written in the same order as input values into the binary array (see Figure 2). At this point we already get some level of compression. And to get even more we use a high-performance algorithm Zstandard for compression of the obtained binary array. Our experiments shown that usage of the compression adds 10-15% overhead for data reading.

As a result we get a highly compressed representation of the input data chunk – data blocks. During uploading data blocks are written successively into a single array – data segment – that is written as a file into storage backend.

2.3 Discussion

It can be noted that the way of data organization in the presented approach resonates with columnar data formats, like Parquet and Apache ORC, or storage schema of other time series databases [12]. The most distinct and significant difference is that timestamps and values in our case are stored

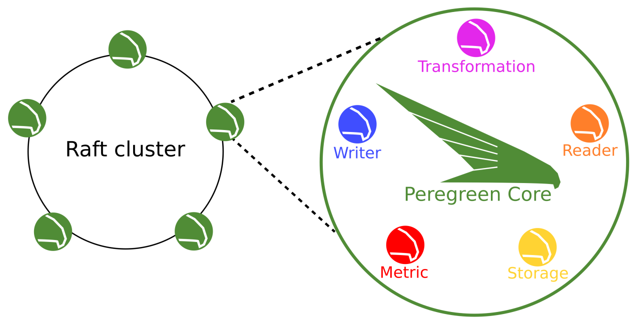

Figure 3: Overview schema of Peregreen. Every node in the Raft cluster includes Peregreen core and five modules.

not separately in different columns/series but together, one *(timestamp,value)* pair after another. This approach along with utilizing binary offsets allows the required parts of segment files to be extracted in a single I/O operation instead of many for other formats (timestamps reading and values reading). When using remote storage backends (Amazon S3 or HDFS) these additional I/O operations significantly decrease performance of data extraction. A possible logical drawback of our approach is a lower level of compression because of mixed sequences; however, experiments described in Section 4.1 show that read-optimized data encoding is more compact than in other solutions.

Another possibly questionable aspect of the proposed approach is the index organization when blocks in a segment and segments in an index are stored in the array rather than access-optimized data structures like trees. During development we experimented with different ways of organizing tiers in the index. We discovered that when having a reasonable number of blocks per segment and segment per index (up to 10,000) array-based index provides almost the same performance as B-tree-based index in terms of elements filtering. This is due to the simplistic logic of the array-based index and absence of recursion calls during the search.

3 Peregreen overview

Peregreen is a distributed modular database that was built to satisfy requirements described in Section 1. Peregreen was developed using Go programming language. Peregreen architecturally consists of three parts – core, modules, and cluster. This section describes these parts and gives information about limitations of the current implementation of the database. Overview schema of Peregreen is presented in Figure 3.

3.1 Peregreen core

Peregreen core is a backbone of the whole system. It implements concepts described in the previous section in a module responsible for the following CRUD operations.

Uploading. Data uploading procedure combines the three-tier indexing mechanism and read-optimized series encoding. At the first step input data is split into data chunks. If there is an existing data for the time interval covered by the new data, Peregreen core extracts it from the storage backend and combines new and existing data. After that the core goes through data chunks and simultaneously converts them into data blocks as described in Section 2.2 and creates index blocks as described in Section 2.1. The core then combines data blocks into segment files and index blocks into index segments. New version of the segment is embedded into the name of the segment file. After combining index segments into the index, segment files and the index are written to the storage backend.

Deletion. On deletion user specifies a time interval, in which the data must be deleted. Indexing engine extracts the data covering target interval from the storage backend, removes all entries that lie in between the start and finish timestamps, updates index and internal representation and loads them into the storage backend.

Search by value. This operation is designed to extract time intervals, values in which satisfy some conditions. Distinctive feature of search operation is that it does not access the data stored in the storage backend, but instead it works only with segments and blocks of the index. It allows to perform very fast search with complex queries over large amounts of loaded data. Time resolution of returned intervals equals to the block interval of the index.

During the search Peregreen core scans through segments of the index and checks whether they fall into the search interval. If a segment belongs to the search interval, indexer iterates over blocks of the segment and checks them for both matching to the search interval and to search query. Metrics available for search queries are all metrics that were computed during the uploading. If new metrics are added after the data was loaded, it is possible to recalculate new metrics for the existing data.

Peregreen provides an easy syntax for describing search queries. Queries consist of conditions combined by logical operators. Conditions have the following form:

```
metric operator value
```

where *metric* is a name of the metric in index blocks, *operator* is a conditional operator, and *value* is a value against which a value of the metric will be compared. Conditional operators used are very close to select operators in MongoDB – *lt* (lower than), *lte* (lower than or equal), *gt* (greater than), *gte* (greater than or equal) and *eq* (equal). Logical operators used for conditions combining include & (conjunction) and | (disjunction). For example, the following command: "min lte 200 & max gte 50 | avg eq 75" describes a query for searching time intervals where one of the following conditions is satisfied: (1) minimum metric is lower or equal to 200 and maximum metric is greater or equal to 50, (2) average metric is equal to 75.

Extraction. Peregreen provides a powerful extraction mechanism that allows performing a fast retrieval of large amounts of data along with data sampling, aggregations and transformations. For extraction, the user specifies sensor identifier, start and finish timestamps, partitioning time interval, and aggregation metrics or transformation functions. Four types of extraction are activated depending on these parameters:

1. Full extraction. This extraction is performed when the user does not specify partitioning time interval and aggregation metrics or transformation functions. In this case, the Peregreen core obtains all points from the storage backend.

2. Sampling. This extraction is performed when the user specifies a partitioning time interval and no aggregation metrics. Data sampling is the extraction of points, timestamps of which differ by user-specified interval, from the storage backend. This operation is useful in cases when the user wants to get a general picture of the data for some time interval. For sampling, the Peregreen core calculates time intervals according to start and finish timestamps, and extracts the first point for every time interval.

3. Aggregation. This extraction is performed when the user specifies a partitioning time interval and some aggregation metrics. This extraction provides the user with a deeper understanding of what is happening in the target time interval. For aggregation, the Peregreen core calculates time intervals according to start and finish timestamps, and generates a set of metrics for every time interval. Metrics available for aggregation include all metrics supported by Peregreen.

4. Transformation. This extraction is performed when the user specifies a partitioning time interval and some transformation functions. This extraction is useful when there is a need for modifying the initial form of the data, e.g. rolling average. For transformation, the Peregreen core applies specified transformation functions to values as they are retrieved.

For extracting values from the storage backend, Peregreen core utilizes meta information stored in all three tiers of the index. Based on start and finish timestamps the core calculates which segments have to be used. For this it uses segments' start times and segment interval from the index. For every segment the core then determines which blocks to extract using segment start time, block interval and blocks ordering. After that the core uses length of the encoded representation of blocks in the segment to calculate the range of bytes need to be extracted from the segment file. The core then extracts only the required part of the segment file, and for every data block of that part performs the actions of the encoding in the reverse order – decompress data with Zstandard, convert raw bytes to deltas and apply delta decoding for them. During decoding stage, Peregreen core applies required aggregations and transformations to the data if specified by user.

It can be seen that the way how internal representation in the Peregreen core is organized allows to achieve the optimal interaction with the storage backend by having only one read operation per segment file. This is the reason why the encoding is called read-optimized.

3.2 Peregreen modules

One of the most powerful features of Peregreen is its extensibility. This section describes five types of modules that allow to integrate various types of storage backends, aggregated statistics, transformations, input and output data types. In terms of implementation, modules are programming interfaces that provide a required set of methods.

Storage. *Storage* is a module, which is responsible for a proper integration of Peregreen with the storage backend. These methods include reading and writing indices, writing segment files and reading binary data for specified index blocks. Currently Peregreen contains three storage implementations out of the box – local file system, HDFS and Amazon S3. Local and HDFS storages allow to work with a data placed within a file system or HDFS respectively. They make use of file offsets for navigating in segment files and extracting only required data parts from them during data reading. S3 storage allows working with a data placed in the Amazon S3 object storage. It stores all data in one bucket specified in configuration file. S3 storage extracts only required data parts from segment files stored in S3 using HTTP bytes range headers.

Reader. *Reader* module is responsible for reading data elements from the input stream. It has only one method for reading a single data element. This is related to the internal mechanics of data reading in Peregreen – it creates data parts, which will be then converted into data blocks, on the fly to minimize memory consumption and total number of operations. That is why there is no need for a method to read all input data at once. The way how *Reader* will extract the data is defined by its implementation and two parameters – data type and format. Data type is a name of one of six supported data types – byte, short, integer, long, float and double. Depending on the data type the way of how *Reader* extracts element might slightly change. Format describes the rules of searching timestamps and values in the input stream. Lets examine how it works on the example of *Reader* implementation for CSV format, which is available in Peregreen by default. Its format definition looks as follows:

```
tsPos-valPos-sep
```

where *tsPos* is position of the timestamp in the string, *valPos* is position of the value in the string, and *sep* is a separator symbol. This simple notation allows to cover a wide range of CSV files layouts. Peregreen also supports JSON and MessagePack as input data formats.

Writer. There are four types of data that Peregreen can return on requests – search results, timestamp-value pairs, aggregations and transformations. *Writer* module describes methods for converting these types into desirable output format. At the moment Peregreen supports three types of output formats – CSV, MessagePack and JSON.

Metric. In order to provide a flexible search on the loaded data, Peregreen allows configuring and extending aggregated statistical characteristics calculated for the data with the help of *Metric* module. It describes three methods, by implementing which one can create new custom metric for the storage, – inserting a value to the metric, getting a value from the metric and combining two metrics into one. The latter method is used for creating metrics for long time intervals from metrics for short intervals. Currently Peregreen has implementations for count, minimum, maximum, average, standard deviation, median, percentile and mode metrics.

Transformation. Transformations allow to change the data as it is extracted. To do that the module provides a single method that converts input value in its new form. Specific implementations, like moving average, might have internal structures for storing intermediate values. Currently Peregreen has implementations for absolute value, difference, summation, subtraction, multiplication and moving average transformations.

3.3 Peregreen cluster

Peregreen cluster provides users horizontal scalability and fault tolerance. Our cluster operates over HashiCorp implementation[1] of the Raft consensus algorithm [17]. This algorithm provides consistency of a cluster. It is especially important for Peregreen because to eliminate the need to handle collisions only one sensor update at a time is allowed. That is why on every update we need to know where required indices are located in order to properly update them.

In Raft cluster write requests go through the leader node so Peregreen also has leader node in the cluster. All upload queries go through the leader node to get redirected to the node that will process uploading. Read requests are redirected directly to the responsible node by any of the cluster nodes. During the first start of the cluster, one node starts as a leader and all other nodes join to that node as followers.

Peregreen cluster state consists of three components:

1. List of nodes. It stores information required for nodes to connect with each other and with clients – node name, internal and external addresses and ports.

2. List of nodes per index. It stores information about mapping of indices onto cluster nodes. Every index is replicated to several nodes in runtime to achieve high availability. If the first node in the list is down, the second one will process the request, etc.

3. Number of indices per node. It stores information about how many indices are stored on every node of the cluster. This is required for the load balancing of the cluster.

[1] https://github.com/hashicorp/raft

Raft log in Peregreen stores information necessary for recreation of the cluster state when the node starts, i.e. information about the events that change the cluster state – addition of a new node, failure of a node, uploading of a new sensor and changing the replication factor.

3.4 Peregreen limitations

Since Peregreen and its underlying concepts are purposefully designed for storing large amounts of historical time series, it has a number of limitations when considering a general task of time series processing.

1. Preferably batch uploads. Although it is possible to load points one by one, it would be highly inefficient because on every such operation the whole data segment file will be downloaded from the storage backend and rebuilt. Even considering that every Peregreen node has a in-memory cache, loading single points brings too much overheads. But the main use case for Peregreen is loading large amounts of time series data at once and the longer input sequence, the better overall performance of Peregreen becomes.

2. One sensor update at a time. When leader receives an update request for a sensor, it locks index for that sensor until the request finishes. This approach was introduced into Peregreen architecture to eliminate the need for collision detection and resolution when more than one node modifies the data for some sensor. We will investigate the best practices for collisions resolution and address this limitation in the future.

3. No multi-sensor operations. All operations in Peregreen require the client to specify the target sensor in the request, and there are no options to specify several sensors. This comes from the fact that the project requirements specifically covered single-sensor operations. At the moment clients make several single-sensor requests for Peregreen and due to the high execution speed they still achieve good performance.

4. No SQL-like syntax. Peregreen has quite minimalistic REST API designed to execute CRUD operations as described in Section 3.1. We understand that supporting SQL would extend the number of use cases but integration of SQL into Peregreen would require significant changes to the architecture of the database.

4 Experimental evaluation

This section describes two series of experiments that were conducted to check different aspects of Peregreen performance in a range of conditions. The first experiment was performed in an on-premise environment on a rack of blade servers, whereas the second experiment was set in the Amazon AWS cloud. Data used in experiments was generated based on a sample of real data with a frequency about 1 Hz. We generated 1 year of data for 1000 sensors that resulted in 31.5 billion records and the total volume of 750 GB.

4.1 On-premise experiment

In order to check the performance of Peregreen compared to existing databases, we performed a detailed experimental evaluation of various data processing operations for Peregreen and two other solutions widely used for time series storage – InfluxDB and ClickHouse. InfluxDB [13] is an open-core time series data store that provides high throughput of ingest, compression and real-time querying. As of November 2018 it is the most popular time series storage [15]. ClickHouse [25] is an open source distributed column-oriented DBMS that provides a rich set of features to its users. Due to the MergeTree table engine and time oriented functions, ClickHouse has proven to be efficient for working with time series data [5,20].

Hardware setup for the first experiment consists of IBM blade servers, each of which has the following characteristics: 2x Intel Xeon 8C E7-2830 (32 vCPUs), 128 GB RAM, 100GB SSD local storage and 2000 GB attached storage from IBM Storwise V3700 storage system connected via fiber channel. All databases were deployed at the local storage whereas the data was stored in the attached storage. In our experiments we used 1, 4 and 7 instances of blade servers.

InfluxDB configuration. In our experiment we used default configuration of InfluxDB provided by its developers with two minor changes. The first is that limit on maximum size of the client request was disabled to upload historical data split by months. And the second, to perform search by values in more equal conditions we created additional aggregation table, which contained 1 hour time intervals and minimum and maximum values for these intervals. Schema of both main and aggregation tables can be found in the listing[2].

When performing several SELECT requests for different time intervals there are two options for InfluxDB – several individual requests or one request with subrequests. Our empirical study[3] shown that making several simultaneous requests is faster than one request with the same number of subrequests. That is why in our experiments we abandoned usage of subrequests.

It must also be mentioned that the trial enterprise version of InfluxDB cluster we used in experiments could not be deployed on 7 instances and because of that experiments for InfluxDB were conducted only for 1 and 4 instances.

ClickHouse configuration. ClickHouse was configured to use the following table schema: date Date, timestamp UInt64, sensorId UInt16, value Float64. Partitioning for the table was set by year and month using toYYYYMM function by date column. The primary key and thus physical order of the records was formed by sensorId and timestamp fields. Table for time series indexing was created as a materialized view of the previous table using 'group by' aggregation by date, sensorId, floor(divide(timestamp, <tdisr>)), where 'tdisr' is a block interval equal to 3600000 ms (1 hour).

[2]https://bit.ly/2P8d4B7
[3]https://bit.ly/2KIKTrG

Table 1: Data upload time for on-premise experiment, minutes

	1 instance	4 instances	7 instances
Peregreen	891	243	150
ClickHouse	530	193	83
InfluxDB	1847	583	n/a

Table 2: Total stored data volume, GB

Peregreen	ClickHouse	InfluxDB
172	301	272

The materialized view had the following schema: date, sensorId, min(timestamp) AS mnts, max(timestamp) as mxts, min(value) AS mnval, max(value) AS mxval. The primary key of the materialized view was formed by sensorId, mnval and mxval fields. For both tables we used MergeTree engine, which provides the best combination of scan performance, data insertion speed and storing reliability.

These two tables were created on each node due to master-master architecture of ClickHouse. Materialized view tables are updated automatically upon insertion into the main tables. Compression method was changed to Zstandard for all nodes. Index granularity of raw tables was left 8192 as it does not contribute significantly to the increase of data size stored on the disk but allows to position more precisely for 1 hour interval queries. Max thread setting was set to 16 for queries with a single client using SET instruction supported by SQL dialect of the ClickHouse. Other settings were left at their default values. To perform queries to ClickHouse, its native tool called clickhouse-client was used. In experiments with multiple clients, multiple processes were spawned each sending its own SQL query. Native format was used by clickhouse-client to retrieve data.

Peregreen configuration. For all experiments Peregreen was configured as follows: block interval was set to 3,600,000 ms (1 hour), segment interval was set to 31,622,400,000 ms (366 days), compression was enabled. With these settings an average block size was 15 KB, and the segment size was about 130 MB.

Data uploading. In this experiment data uploading requests were sent to storages from all instances of the cluster. It means that for setup with 1 instance it sent all 1000 sensors, with 4 instances each instance sent 250 sensors, and with 7 instances each sent 142-143 sensors. It allows to check scalability of the databases in terms of data uploading.

Results of data uploading time for different sets of nodes are presented in Table 1. It can be seen that ClickHouse provides the highest speed of uploading with the rate of 8 GB per minute on 7 instances. InfluxDB uploading speed was slightly less than two other solutions, but on 4 nodes it demonstrated the uploading rate of ~900,000 records per second, which is quite close to the InfluxData official benchmarks [8].

Table 2 shows the total volume of internal data representa-

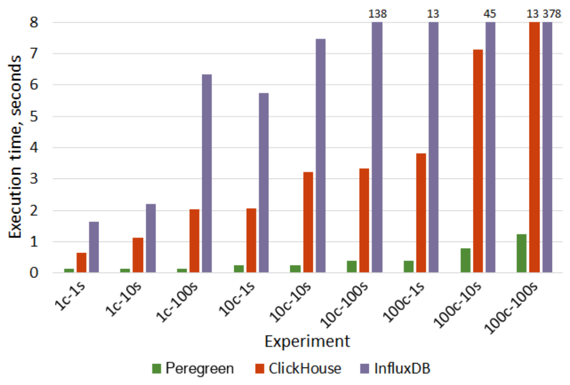

Figure 4: Values search execution time for 4 instances

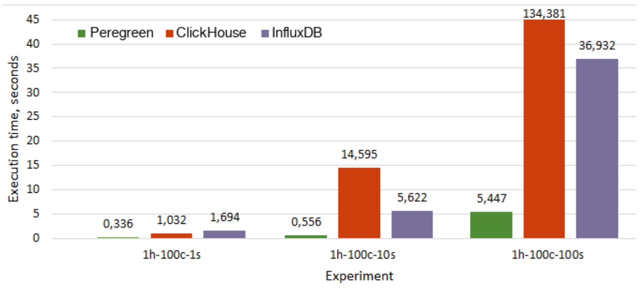

Figure 5: Data extraction time for small requests on 1 instance

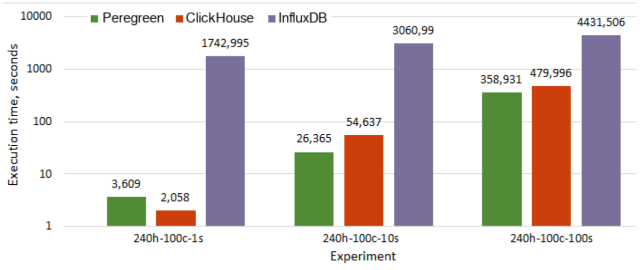

Figure 6: Data extraction time for large requests on 1 instance, logarithmic time scale

tion stored by three solutions. With the help of delta encoding and Zstandard compression Peregreen is able to compress the raw data 4 times to 172 GB from 750 GB. Other storages also provide quite high compression rates – 2.5x for ClickHouse and 2.7x for InfluxDB. Despite using Zstandard algorithm, compression rate for ClickHouse is lower than demonstrated in other benchmarks [1]. But compressibility heavily depends on the data characteristics (schema, frequency, etc.) and thus direct comparison of two benchmarks for different datasets would be far from the mark.

It also should be mentioned that indices for all 1000 sensors in Peregreen have the total size of 110 MB, 110 KB per sensor. Such small size ensures minimal footprint of the system and allows to store all indices in the memory in the runtime.

Values range search. In this series of experiments we simulated behavior of clients who try to perform various search by values requests. Number of clients varied in range $[1, 10, 100]$, number of sensors for which search requests were made by each client varied in range $[1, 10, 100]$. Every request was amplitude search, where minimum and maximum values cover all values in the dataset, so all solutions had to perform full scan of their storages. In Figure 4 results of this experiment for 4 instances are presented. Columns captions encode experiment types – c stands for the number of clients, s stands for the number of sensors per client. From the figure we can clearly see that Peregreen significantly outperforms two other solutions in all scenarios. This is related to the fact that search operations in Peregreen are performed using only indices, which are stored in the memory, whereas ClickHouse and InfluxDB select results from the tables that are stored on the disk.

Data extraction. These experiments were designed to investigate how well target storages can handle data extraction workloads of various sizes. Number of clients for this series varied in range $[1, 10, 100]$, number of sensors for which extraction requests were made by each client varied in range $[1, 10, 100]$, time interval for extraction varied in range $[1, 24, 240]$ hours. The largest case – 100 clients, 100 sensors per client, 240 hours – is a very extreme scenario that involves extraction of 8.6 billion records.

Sample results for small requests (1 hour, 100 clients, 1, 10, 100 sensors per client) execution on 1 instance are presented in Figure 5. Columns captions encode experiment types – h stands for the extracted time interval, c stands for the number of clients, s stands for the number of sensors per client. Execution time for Peregreen is much lower than for other systems, because it quickly navigates in segment files using the index and extract only required data blocks (no more than two for these experiments). InfluxDB in these experiments significantly outperformed ClickHouse, which is expected because InfluxDB is designed for working with short time series.

But the situation dramatically changes if we try to extract a lot of long time intervals. In Figure 6 results for large requests (1 hour, 100 clients, 1, 10, 100 sensors per client) are presented. Plots were generated using logarithmic time scale because results differ by orders of magnitude. InfluxDB demonstrates very low extraction speed, since extraction of this many long time intervals is far from its standard use case. ClickHouse, on the other hand, is very good at reading long sequences of data and because of that it demonstrates impressive results of 43,000,000 records per second for the scenario 240h-100c-1s. Although Peregreen performs almost twice slower in this scenario, with increase of parallel requests it starts to outperform ClickHouse and for the hardest case it demonstrates execution speed of 24,000,000 records per second against 18,000,000 for ClickHouse.

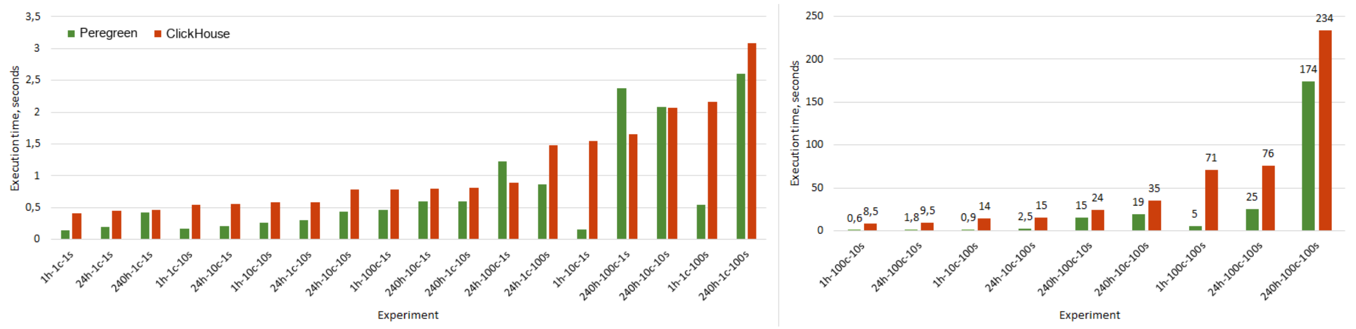

Figure 7: Data extraction time for 7 instances

In Figure 7 experimental results for all types of requests are presented. These experiments were conducted on 7 instances and because of that there are no results for InfluxDB. For relatively small requests, which are shown on the left plot, Peregreen and ClickHouse demonstrate comparable performance. But as the workload grows transcendence of Peregreen also grows. In the hardest scenario 240h-100c-100s Peregreen achieves the speed of 49,500,000 records per second. The speed of ClickHouse is slightly lower, but also very high – 37,000,000 records per second. Difference in the processing speed can be explained by following factors: (1) ClickHouse has lower compression rate (difference in blocks organization and no delta encoding) and because of that has to read more data from the disk during extraction; (2) data parts in ClickHouse are organized by size, not by time, that is why ClickHouse has to read some unnecessary data during extraction by time intervals; (3) ClickHouse creates large number of files with data parts and during extraction it has to read from a lot of files, which increases a number of random reads from disk, whereas Peregreen reads from very small number of segment files.

4.2 EC2 experiment

The second series of experiments was aimed to evaluate performance of Peregreen in a cloud environment for two types of storage backend – local file system and Amazon S3.

Hardware setup for this experiment consists of Amazon EC2 instances of c5.4xlarge type with 16 vCPUs, 32 GB RAM, 8 GB internal SSD storage and 500 GB attached EBS storage. All instances were localed in us-east-1 region. Peregreen was deployed on the local storage whereas the data was stored in the attached storage. Testbed contained 1, 4 and 7 instances.

Benchmarking utility was deployed on a dedicated instance to avoid its influence at performance of Peregreen nodes.

Data uploading. This experiment was designed the same way as in Section 4.1 – data is loaded from all instances of the cluster. Experimental results are presented in Table 3. We can see that in all cases uploading to the cluster with EBS storage backend is faster than to the one with S3 storage backend. It is expected because in the second case segment files are uploaded into remote web service. Nevertheless, both storages provide high uploading rates, 21-23 MB/s per node for EBS storage and 17-18 MB/s per node for S3 storage. Also, both storages demonstrate close to linear change in uploading time with increasing in number of instances, which is very important for the solution deployment on a large scale.

Table 3: Data uploading time for EC2 experiment, minutes

	1 instance	4 instances	7 instances
EBS backend	573	122	75
S3 backend	702	175	97

Search by values. In this series of experiments we performed various amount of parallel search by values requests. Number of parallel requests varied in range $[1, 10, 100, 1000, 10000]$. Since search by values request does not depend on storage backend, we did not compare setups with different backends, and only checked how well Peregreen handles increasing load. In order to make Peregreen process all segments and blocks, we used the following condition in search requests:

```
min lte 500 & max gte -200
```

Due to the fact that searching through indices located in RAM is a very lightweight operation, results for setups with 1, 4 and 7 did not differ significantly. In Figure 8 experimental results for 4 instances are presented. It is clear that search request scales very well and provides a sub-second execution time even for 10000 simultaneous requests.

Data extraction. In this series of experiments we investigated, how well can Peregreen with different storage backends perform under wide range of data extraction requests, and whether usage S3 as a persistent storage of large amounts of time series data can bring performance comparable to the EBS storage. For this we varied number of concurrent requests made to the manager node in range $[1, 10, 100, 1000, 10000]$ and extracted time interval in range $[1, 6, 24, 240]$ hours. Due to the extremely large memory requirements and limited

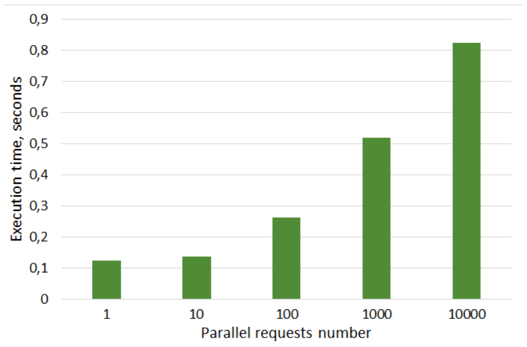

Figure 8: Data search execution time for 4 instances in EC2

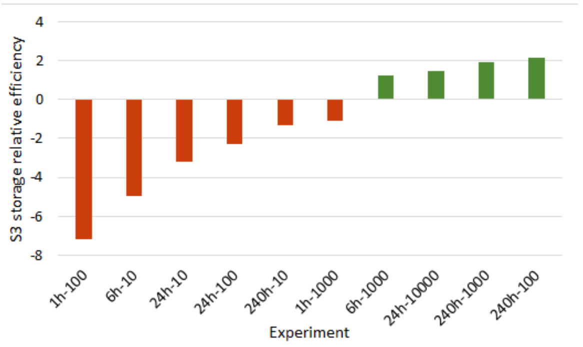

Figure 9: Efficiency of S3 storage backend with regard to EBS storage backend for 4 nodes in EC2

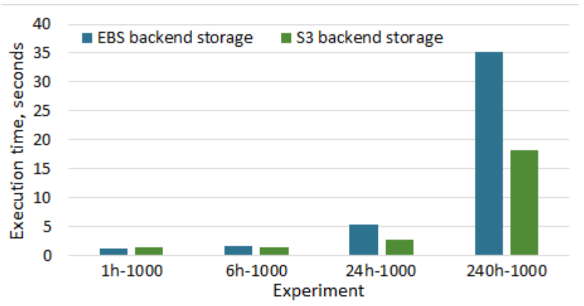

Figure 10: Data extraction results for 4 instances in EC2

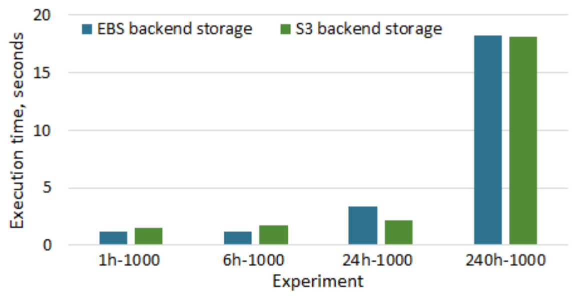

Figure 11: Data extraction results for 7 instances in EC2

amount of RAM on EC2 instances (320 GB in total) maximal request (10000 requests for 240 hours each) could not be executed, but all other requests completed successfully. All requests were performed 10 times to downplay possible network and hardware effects.

To compare performance of Peregreen with S3 storage backend and with EBSstorage backend we performed all described above experiments in both settings and calculated relative efficiency of S3 as a ratio of execution time with S3 backend to execution time with EBS backend. In Figure 9 S3 relative efficiency plot is presented. Experiment with 100 parallel extraction requests for 1 hour shows the worst S3 efficiency – 49 ms for EBS backend and 351 ms for S3 backend. But as the load (number of parallel requests and interval length) grows, S3 relative efficiency increases, and starting from 1000 requests for 6 hours S3 becomes more efficient than EBS (1320 ms for S3 and 1605 ms for EBS). Maximal loads (100 and 1000 requests for 240 hours) demonstrate two times lower execution time for S3 storage backend – 35096 ms for EBS backend and 18156 ms for S3 backend for 1000 requests. Such difference can be explained by the fact that EBS volumes have limits on a throughput and number of I/O operations per second [2]. S3, on the other hand, does not have such limitations [3], which allows it to scale better under the high workload.

In order to further investigate scaling of two storage backends, we analyzed performance of the same set of requests for different sets of nodes. In Figures 10 and 11 we present experimental results for 1000 parallel requests for extraction of 1, 6, 24 and 240 hours on 4 and 7 instances respectively. It is interesting to mention that for 6h-1000 scenario on 4 instances S3 storage backend is slightly more efficient than EBS (1320 ms and 1605 ms), whereas in case of 7 instances EBS backend provides smaller execution time (1233 ms and 1782 ms). But the most important observation is in the 240h-1000 scenario. When Peregreen deployed on 4 instances S3 provides twice faster execution, but for 7 nodes EBS storage demonstrates the same result as S3 storage, and execution time for S3 storage does not change. The reason why this scenario cannot be finished in less than 18 seconds is a network bandwidth limitations – this request generates 864 million data records with the volume of almost 18 GB. Even with a very high EC2 inter-instance throughput [4] it takes a fair amount of time to transfer such data volume to the testing instance.

5 Alternatives comparison

Today there are plenty of time series oriented databases and storages. In this section, we give a brief description of the most popular and mature solutions, which are used for storing time series data, and investigate how they compare against Peregreen. Comparison criteria are based on requirements

Table 4: Comparison of databases for storing time series

	InfluxDB	ClickHouse	Kudu	Cassandra	Gorilla	Gnocchi
Index type	sparse	controlled sparse	precise	precise	sparse	external
Internal data structures	LSM+ MVCC	LSM	LSM+ MVCC	LSM+ MVCC	PSP	n/a
Data layout	column	column	column	row-wise	ts-specific	n/a
TS-specific compression	yes	no	no	no	yes	n/a
Values-based index	mat. view	mat. view	no	special index	no	no
Aggregations	yes	yes	no	no	no	yes
SQL-like query syntax	yes	yes	no	yes	no	no
In-memory	no	yes	no	no	yes	no
Low-cost object storage	no	no	no	no	no	yes

	TimescaleDB	OpenTSDB	Snowflake	Prometheus	Peregreen
Index type	various	sparse	no	sparse	controlled sparse
Internal data structures	B-tree+ MVCC	n/a	tables	custom	custom
Data layout	row-wise	column	column	ts-specific	ts-specific
TS-specific compression	yes	yes	no	yes	yes
Values-based index	special index	no	no	special index	special index
Aggregations	yes	yes	yes	yes	yes
SQL-like query syntax	yes	no	yes	no	no
In-memory	no	no	no	yes	no
Low-cost object storage	no	no	partly	no	yes

from Section 1 and common use cases of working with time series. Results of the comparison are presented in Table 4.

InfluxDB is an open-core time series data store that provides high throughput of ingest, compression and real-time querying. As of November 2018 it is the most popular time series storage [15]. ClickHouse is an open source distributed column-oriented DBMS that provides a rich set of features to its users. Due to the MergeTree table engine and time oriented functions, ClickHouse has proven to be efficient for working with time series data [5, 20]. TimescaleDB is an open-source extension of PostgreSQL designed to overcome limitations of PostgreSQL when it comes to working with time series. TimescaleDB introduces hypertable – an entity, which has the same interface for the user as a simple table, but internally is an abstraction consisting of many tables called chunks. This mechanism allows avoiding slow disk operations by storing in memory only the chunk of the latest period. Apache Kudu [14] is a column-oriented data store that enables fast analytics in the Hadoop ecosystem. Apache Cassandra is a general-purpose distributed NoSQL storage, the main aims of which are to provide linear scalability and fault tolerance. Gorilla [18] is a fast in-memory time series database developed by Facebook. Gnocchi is an open-source TSDB that aims to handle large amounts of aggregated data. OpenTSDBis an open-source scalable, distributed time series database written in Java and built on top of HBase. Snowflake is an analytic data warehouse provided as Software-as-a-Service (SaaS). It uses custom SQL database engine with a unique architecture designed for the cloud. Prometheus is an open-source systems monitoring and alerting toolkit that has a powerful time series storage engine.

The first criterion for the comparison is an index type. Precise indices in Kudu and Cassandra make possible fast search of exact data entries, but they generally do not increase the speed of long continuous time series extraction compared to sparse indices in other solutions, and moreover, index sizes in Kudu and Cassandra are very large. A sparse index is much more suitable for the extraction of long time series, and control over the sparsity of the index is an additional advantage. Gnocchi relies on external indexing by PostgreSQL or

MySQL. The interesting fact about Snowflake is that it does not have indices and instead relies on data micro-partitioning and clustering [22] that have proven to be very effective on a large scale.

In terms of internal data structures, log-structured merge-tree (LSM) is the most widespread way to store the data. But InfluxDB, Kudu and Cassandra also use multiversion concurrency control (MVCC) to handle concurrent data modifications. And when using MVCC there can exist several versions of the data, and if these versions were not merged into one during the reading operation, the storage will have to merge versions upon reading, which decreases the overall performance. On the other hand, plain sorted partitions (PSP) used in Gorilla, make possible fast navigation and extraction of large amounts of data. OpenTSDB relies on the backend storage, usually HBase, for storing the actual data. Snowflake tables are the part of the proprietary platform thus no detailed information on them is available. The schema of storing the data in Prometheus includes custom data format for chunks and write ahead logs that are optimized for reading. Peregreen also has a custom format based on read-optimized series encoding.

The next criterion is an internal data layout. Row-wise store data layout of Cassandra and TimescaleDB can be efficient for sequential reads only when there are no other columns except timestamp and value. InfluxDB stores timestamps and values as two separate sequences in its blocks and in that sense its layout is also columnar. OpenTSDB stores the data in columnar storages, like HBase or Bigtable, and because of that provides good speed of data extraction. Prometheus [19] as well as Peregreen has time series specific data layout format. Regarding the time series specific compression, general purpose databases do not have special methods for efficient time series compression, like delta or delta-of-deltas compression. It is worth mentioning that TimescaleDB allows to compress the data by chunks but compressed chunks cannot be updated [24].

The next criteria are related to search by values and data aggregations. The best way to perform a fast search based on values conditions is to have some sort of values-based index. ClickHouse and InfuxDB provide such functionality through materialized views and continuous queries. Cassandra and TimescaleDB can easily create a secondary index over the values column and thus provide very fast search. Snowflake relies on clustering and intelligent query optimization for providing high speed of searching across values conditions. Prometheus allows querying the data based on values of the tags. Index of Peregreen stores aggregated metrics for blocks and segments that makes possible fast search in the large data volumes. All solutions except for Kudu, Cassandra and Gorilla, support data aggregation during extraction.

SQL-like query syntax is a great feature for any database since SQL is familiar to many developers and analysts. Prometheus provides a query language called PromQL that lets users select and aggregate time series data. It should also be noted that Peregreen has no support for SQL syntax and has a set of purposefully built operations available through API.

The last set of features is related to the support of various storage backends. Only ClickHouse, Gorilla and Prometheus can store the data in memory to provide the maximal speed of data access. The most interesting criterion is the ability of the storage to use low-cost object storages, like Amazon S3 or Google Cloud, because it is a very important advantage in terms of cost of usage. Apart from Peregreen only Gnocchi support such functionality. Snowflake supports integration with Amazon S3 but in order to efficiently perform operations over the data it have to be loaded into Snowflake tables [21].

As can be seen from the presented comparison, modern databases provide users with great functionality when speaking about working with time series. Nevertheless, Peregreen provides a unique set of features – high performance along with cloud object storage integration – for working with large volumes of historical time series.

6 Conclusion

In this paper we presented Peregreen – high performance storage for numeric time series. Peregreen is based on a three-tier indexing mechanism and read-optimized series encoding, which allows it to achieve high compression rate, small index size and high execution speed of data processing operations. Experimental results show that Peregreen performs better than InfluxDB and ClickHouse in operations of data search and extraction over large amounts of historical time series data. Peregreen also provides integration with Amazon S3 as a backend data storage out of the box, which means that it can greatly reduce the cost of data storing while increasing the overall efficiency compared to storing the data in EBS, as our experiments demonstrate.

7 Acknowledgements

We would like to thank our shepherd Gustavo Alonso and anonymous reviewers for their valuable feedback. This work is carried out in the Siemens-ITMO Industrial Cognitive Technologies Lab and is funded by Siemens.

References

[1] Altinity. Compression in ClickHouse, 2017.

[2] Amazon. Amazon EBS Volume Types, 2018.

[3] Amazon. Request Rate and Performance Guidelines, 2018.

[4] Amazon. The Floodgates Are Open – Increased Network Bandwidth for EC2 Instances, 2018.

[5] Dmitry Andreev. ClickHouse as Time-Series Storage for Graphite, 2017.

[6] W Brian Arthur. Asset pricing under endogenous expectations in an artificial stock market. In *The economy as an evolving complex system II*, pages 31–60. CRC Press, 2018.

[7] James Lopez Bernal, Steven Cummins, and Antonio Gasparrini. Interrupted time series regression for the evaluation of public health interventions: a tutorial. *International journal of epidemiology*, 46(1):348–355, 2017.

[8] Chris Churilo. InfluxDB Tops Cassandra in Time Series Data and Metrics Benchmark, 2018.

[9] Carlo Curino, Evan PC Jones, Samuel Madden, and Hari Balakrishnan. Workload-aware database monitoring and consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 313–324. ACM, 2011.

[10] Manuel Díaz, Cristian Martín, and Bartolomé Rubio. State-of-the-art, challenges, and open issues in the integration of internet of things and cloud computing. *Journal of Network and Computer applications*, 67:99–117, 2016.

[11] Chris C Funk, Pete J Peterson, Martin F Landsfeld, Diego H Pedreros, James P Verdin, James D Rowland, Bo E Romero, Gregory J Husak, Joel C Michaelsen, Andrew P Verdin, et al. A quasi-global precipitation time series for drought monitoring. *US Geological Survey Data Series*, 832(4), 2014.

[12] InfluxData. In-memory indexing and the Time-Structured Merge Tree, 2018.

[13] InfluxData. InfluxDB home page, 2018.

[14] Todd Lipcon, David Alves, Dan Burkert, Jean-Daniel Cryans, Adar Dembo, Mike Percy, Silvius Rus, Dave Wang, Matteo Bertozzi, Colin Patrick McCabe, et al. Kudu: storage for fast analytics on fast data. *Retrieved June from http://getkudu. io/kudu. pdf. Pages,, and*, 2015.

[15] Knowledge Base of Relational and NoSQL Database Management Systems. DB-Engines Ranking of Time Series DBMS, 2018.

[16] Knowledge Base of Relational and NoSQL Database Management Systems. DBMS popularity broken down by database model, 2018.

[17] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[18] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8(12):1816–1827, 2015.

[19] Prometheus. Prometheus storage schema, 2020.

[20] Alexander Rubin. A Look at ClickHouse: A New Open Source Columnar Database, 2017.

[21] Snowflake. Bulk Loading from Amazon S3, 2020.

[22] Snowflake. Micro-partitions Data Clustering, 2020.

[23] Timescale. TimescaleDB vs. PostgreSQL for time-series, 2017.

[24] Timescale. TimescaleDB compression, 2020.

[25] Yandex. ClickHouse home page, 2018.