A 5-dimensional key in TStorage: a key to the future of data storage in Industry 4.0

Introduction

In 2024, the global smart grid market was valued at 55.54 billion USD, and by 2030, it is projected to grow to 145.42 billion USD [1]. The implementation of smart grids is facilitated by the smart metering systems, and the adoption of smart meters is driven by policy. In 2012 in the European Union, Directive 2012/27/EU required at least 80% of electricity consumers to be equipped with smart meters by 2020 (provided the roll-out of this technology was assessed positively); by the end of 2023, that goal was achieved in 15 of the member states [2]. The European requirements and standards for smart meters were prepared in 2011 by a consortium of CEN, CENELEC, and ETSI [3]. The key functionalities of smart meters identified in [3] involve the export of the measured data both for billing and analytical purposes. The system supporting such data gathering and analysis requires an efficient solution for data storage and retrieval.

It is worth pointing out that a database fit for a smart metering system will also be suitable for a number of IoT time-series applications, particularly any application that necessitates real time data acquisition from distributed devices such as sensors. TStorage, a NoSQL database developed by Atende Industries, is an example.

The main purpose of this paper is to present TStorage and its multidimensional key. After the brief section on the application of distributed networks of sensors, the design of TStorage's *key* is described. Then, an overview of NoSQL databases is given, and TStorage is compared with them. The next two sections delve into the issue of efficient range access and ability to track changes to data, after which two types of access pattern in time domain available in TStorage are described. The final section is the conclusion.

Problem description

Smart metering systems consist of a multitude of distributed meters. Each smart electricity meter can record information about the electric energy consumption, voltage, current, power factor, and more. Smart meters can also be used to measure the natural gas or water consumption. Other examples of use cases for distributed networks of sensors include air quality monitoring systems; control systems in power plants, factories, and warehouses; or vehicle fleet location and fuel consumption monitoring. In general, any system that collects data in real time requires a database to store it and retrieve it for purposes such as billing,

KRAJOWY PLAN ODBUDOWY
Rzeczpospolita Polska
Sfinansowane przez Unię Europejską NextGenerationEU
NCBR Narodowe Centrum Badań i Rozwoju

predictive maintenance, forecasting demand, assuring quality of a process, and more. The data has to be consistent and accessible with low latency. Moreover, the architecture of such a database has to support multiple request patterns, that is, allow for querying for different combinations of metadata characterizing measured values such as time when the measurement was taken and when was acquired by the system, and which sensors performed the measurement.

Solution

Five dimensions of the *key*

A database for the measurement data can be seen as a *key-value store*, where a *value* contains a *payload* with a measured value and associated metadata (e.g., a unit of the value, a flag whether the value is measured or estimated, etc.), and a *key* is its identifier. The structure of a *key* has to support multiple request patterns, as mentioned in the previous section.

A unique characteristic of TStorage is a 5-dimensional *key* that is well-suited to meet the needs of measurement systems. The *key*'s components are listed below:

- *cid*: the ID of the cluster/user,
- *mid*: the ID of the measurement device,
- *moid*: the measured quantity,
- *cap*: the time the measurement was captured,
- *acq*: the time the measurement was acquired by the system.

The cluster/user ID *cid* allows for the acquisition of data by different energy operators (each having a unique *cid*, or a set of *cid*s that they can assign to their individual customers) without interference; the measurement devices IDs *mid* have to be unique only within a cluster and not globally. Since a particular device *mid* can concurrently output values of quantities of various physical meaning (for example, in case of a meter measuring electric energy consumption, voltage, current, power factor, etc.), the identifier *moid* allows for differentiation between these types of quantities. The time the measurement was taken is registered in *cap*, and it allows for tracking the evolution of the measured quantity over time. The last component, *acq*, contains the time the measurement data was acquired by the database; it allows for tracking of incoming data and consistent data versioning.

KRAJOWY PLAN ODBUDOWY
Rzeczpospolita Polska
Sfinansowane przez Unię Europejską NextGenerationEU
NCBR Narodowe Centrum Badań i Rozwoju

The energy operator is free to choose the meaning and the values of the parameters *cid*, *mid*, *moid*, and *cap* according to its needs. Therefore, such an operator might want to set up a relational database to associate its individual customers with smart meters, and assign these customers to clusters (*cid*), *mid* to each smart meter, and *moid* to each measured quantity. It is worth pointing out that such a database— even if it contains millions or even billions of records—is bounded by the number

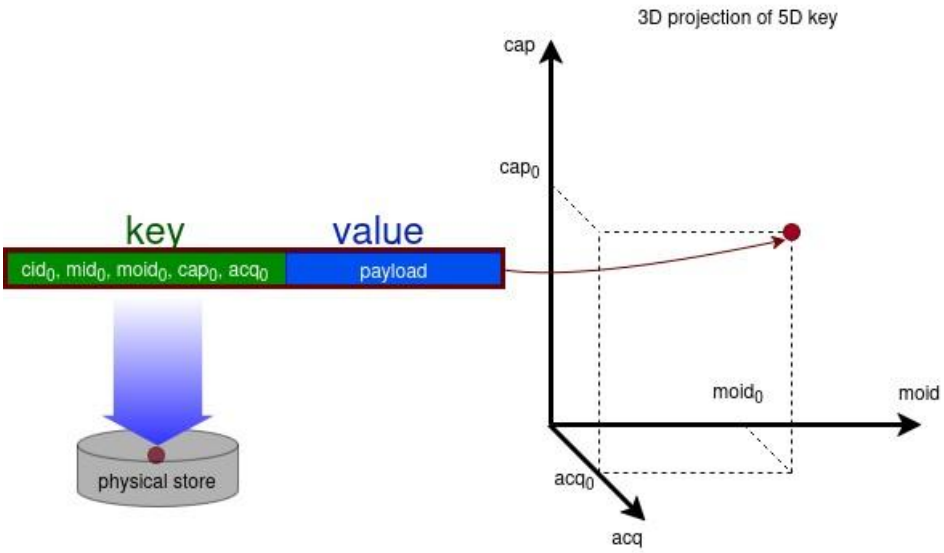

Figure 1: Relation between the *key* and data placement

of meters and customers. On the other hand, as our civilization continuously produces new information [4], the volume of measured quantities grows boundlessly. The goal of TStorage is to efficiently store and retrieve these vast amounts of data.

The tuple (*cid*, *mid*, *moid*, *cap*, *acq*) constitutes a *key* which together with a *value* represents *data record* (see Figure 1; for clarity, only 3 dimensions are shown instead of all 5). The *value* holds a *payload* that is a result of measurement and its metadata, and it does not have to be a scalar—it can be composed of multiple scalars (e.g., a matrix). What is more, even images and videos or compressed files can be stored in such a way. From the security point of view, the separation of the *key* and *value* allows for encryption of the *payload*. Each *data record* is independent from each other; different users of the database can have their own encryption strategies.

How does the design of the *key* in TStorage compare to others?

KRAJOWY PLAN ODBUDOWY
Rzeczpospolita Polska
Sfinansowane przez Unię Europejską NextGenerationEU
NCBR Narodowe Centrum Badań i Rozwoju

A *key-value* store belongs to a NoSQL, or non-relational, family of databases. The other types of NoSQL databases that can be used for storing sensor measurements (or other time-series data) are wide-column stores (such as [5, 6, 7]) and documentoriented databases (for example [8]), each representing a different approach to data access. The wide-column stores organize data into rows and columns to create a table; however, unlike the relational databases, the column schema is not fixed and can vary between the rows. In document-oriented databases, instead of tables there are collections, a row is replaced with a document (for example a JSON document), and a column is substituted by a field in a document. The *key-value* stores are often considered schema-less (however, TStorage has a schema in the form of a 5-element *key*), whereas wide-column stores and document-oriented databases have flexible schema. To make matters more interesting, some databases blur the line between thesedivisions(forexample, theyareorganizedintotablesbutcanholddocuments), and there is also a distinct category of time series databases (some of which can be classified as NoSQL and others as SQL). Therefore, the classification that follows is necessarily subjective and divided among the line whether the database organize data into a table or not; the former category will be called tabular, whereas the latter table-less. In such a classification, TStorage belongs to the table-less group.

The well-known tabular databases include: Amazon DynamoDB [9, 10] (a successor to widely-cited Dynamo [11]), Oracle NoSQL [12], InfluxDB [13], Amazon Timestream [14] (originally a stand-alone product, but it is no longer offered to new customers [15], and a version working with InfluxDB is recommended [16]), ScyllaDB [17], Apache Cassandra [18], Apache HBase [19], Amazon Keyspaces (works with Apache Cassandra) [20], Spanner [21], and Bigtable [22]. The *key* in a tabular database is usually made of a subset of columns; however, Bigtable [23] and Apache HBase [24] use only a single *row key* column.

The popular table-less databases are: Redis [25], Valkey [26], RocksDB [27], MemoryDB (works with Valkey and Redis OSS) [28], Memcached [29], ElastiCache (works with Valkey, Redis OSS, and Memcached) [30], Azure Cosmos DB [31], MongoDB [32], and Amazon DocumentDB [33]. The *key* in a table-less database is usually a textual variable, but in case of TStorage, as already mentioned, the *key* is a 5-element vector.

A common theme, especially among the tabular databases, is a query-driven design of the data model. In other words, the schema of the database has to be designed with a query workload in mind to ensure its efficiency. Such a design involves a proper choice of the *key*, which in tabular databases translates to a proper choice of the order of columns, since that order will be followed during a query. Moreover, the choice of the key and the order of columns is set for the lifetime of a table and cannot be changed.

KRAJOWY PLAN ODBUDOWY
Rzeczpospolita Polska
Sfinansowane przez Unię Europejską NextGenerationEU
NCBR Narodowe Centrum Badań i Rozwoju

The schema of Amazon DynamoDB includes a *partition key* and a *sort key*, each composed of one *attribute* (an *attribute* is analogous to a column in other databases) [34]. The *partition key* determines the partition on which the data is stored, whereas the *sort key* is used for sorting the data within the partition; during data retrieval the proper partition is located first, and then, inside the partition, the records of interest are found. In Oracle NoSQL a row in the table is identified by a *primary key*, whereas a *shard key* determines the shard on which the row is located; both the *primary* and *shard key* can be made of one or more columns. The order of columns of the *primary key* is important for query efficiency, and the column most often used in queries shall be specified as the most significant one in the *primary key* [35]. In InfluxDB, the row of data is identified by a *primary key* which is a combination of a timestamp (data is ordered by time) and *tags*. In turn, *tags* contain metadata about the data stored in *fields*, and both *tags* and *fields* are analogous to columns. *Tags* have to be ordered: the most commonly queried *tags* shall be put first to ensure query efficiency [36]. The *primary key* is also present in the schema of ScyllaDB, identifying a row in a table, and it consists of a *partition key* and *clustering columns*. The *partition key* can be made of one or more columns and assigns rows to partitions, whereas the *clustering columns* define which columns and in what order are used to sort the rows within each partition. The proper choice of the *partition key* ensures a uniform distribution of data along partitions, and the order of rows within partitions defined by *clustering columns* influences the efficiency of range queries [37]. The proper choice of a *primary key* is also crucial in Apache Cassandra. The first column or columns in the *primary key* constitute a *partition key*, whereas the rest of the columns are *clustering keys* used for sorting within a partition [38]; Amazon Keyspaces uses the same schema design [20]. In line with other discussed databases, in Spanner each row is identified by a *primary key* which is made of one or more columns, and rows are stored in order sorted by *primary keys* [39]. In Bigtable, the data is sorted by the *row key*, and the most efficient queries retrieve data using either a *row key*, or its prefix, or a range defined by starting and ending *row keys*; therefore, the proper choice of a *row key* is again of utmost importance [23]. A similar schema is used by Apache HBase [24]. Moreover, in both Bigtable and HBase, in a given row and column, multiple versions of *cells* (which, similarly to *values*, hold data) can be stored [23, 24]. These *cells* are sorted by timestamps which can mark the moment of writing into the database or be assigned by the user, but they have to be unique [23, 24].

In some table-less databases, there are also useful conventions for a *key* choice. Inparticular, inRedis, the*keys*canbegroupedintocategoriesiftheyconsistofcomponentsdividedbypunctuationmarks(aconventionsimilartothatofBigtable)[40]. For example, a key can have

KRAJOWY PLAN ODBUDOWY
Rzeczpospolita Polska
Sfinansowane przez Unię Europejską NextGenerationEU
NCBR Narodowe Centrum Badań i Rozwoju

the form of *product type:producer name:product id*. A query on a particular group of *keys* can be then performed (e.g., by using a prefix of the *key* [41]).

On the other hand, TStorage does not require any special consideration regarding its schema; it is designed to ensure the efficient access to data from within ranges on all dimensions of a *key*. The user is not required to order the *key* components by their importance, because they are all equal. Obviously, TStorage's designimposesaparticularstructureofa*key*, butsucha5-element*key* iswell-suited for the IoT purposes. There are obvious use cases for retrieving data within some range of time, such as forecasting future energy consumption based on a history between set dates. Other dimensions can support different interesting cases. For example, accessing the data from some subsets of devices (grouped by *mid*) might prove useful for analysis. If the operator plans to query data produced by a group of particular meters often, it can assign them *mid*s in proximity to each other. In such a way, the metering devices of interest would belong to one range in the *mid* dimension, resulting in an optimized data retrieval request.

Flexible access pattern

The flexible access pattern in TStorage also provides benefits in terms of space required for data storage. In other NoSQL databases, a secondary index has to be created to allow for more access scenarios. For tabular databases, a secondary index is a data structure that contains a copy of a subset of columns from the base table but has a different *key*. A covering index has all the columns required for the query. If a secondary index is not a covering index, then the unavailable columns have to be fetched from the the base table. Even though the index contains pointers to the appropriate rows of the base table, this operation still might add time to query processing. For table-less databases, in case of a document-oriented type, a secondary index is built using fields from documents instead of columns from a table.

A mechanism for creating secondary indexes is incorporated into many popular databases, such as: Amazon DynamoDB [42], Oracle NoSQL [43], InfluxDB [44], ScyllaDB [45], Apache Cassandra [46], Spanner [47], Bigtable [48], Azure Cosmos DB [49], MongoDB [50], and Amazon DocumentDB [51]. The implementation of a secondary index and the amount of data (columns or fields) it copies varies between each of these databases. Obviously, such an approach increases the amount of disk space needed, and renting space in a colocation center to accommodate more physical disks is rather quite expensive; moreover, the synchronization of multiple copies requires additional workload.

The issue with the usage of the secondary index is illustrated in Figure 2. For simplicity and clarity, only two dimensions are considered. The base table, shown in Figure 2 on the left, has a *mid* column chosen as a *partitionkey*, and a *cap*

KRAJOWY PLAN ODBUDOWY
Rzeczpospolita Polska
Sfinansowane przez Unię Europejską NextGenerationEU
NCBR Narodowe Centrum Badań i Rozwoju

column chosen as a *sort key*; the measurement data is held in a *payload* column. In order to provide the ability for efficient range queries in both dimensions, a secondary index is created, shown in Figure 2 on the right. Since it contains all the columns that will be requested in a query, it is a covering index, and its *primary key* is a *cap* column, whereas its *sort key* is a *mid* column. In an example, a base table can be used to efficiently handle a *get* request for data identified by *mid* equal to 1 and *cap* in the range from 2 to 3. First, a partition holding *mid* = 1 is located and then a *sort key* is used to narrow the results to a requested range of *cap*. On the other hand, a *get* request for *mid* in the range from 1 to 3 and *cap* equal to 3 cannot be served efficiently. First, partitions associated with *mid* equal to 1, 2, and then 3 have to be visited, and on each partition a search for *cap* = 3 has to be performed. Therefore, a secondary index is used to hasten the process (see the right side of Figure 2). It is a covering index holding a copy of the *payload* column, additionally saving time required for fetching the data from the base table. However, that time savings come with a cost of a greater memory usage.

If there are more than two dimensions, then even more copies of data are required to support *get* requests for various combinations of *primary* and *sort keys*. On the other hand, TStorage supports access to data in each of its 5 dimensions, without resorting to the creation of a secondary index and copying of the data.

Again, for simplicity, let the *key* space be 2-dimensional, as in the example from Figure 2. As shown in Figure 3, TStorage can easily retrieve data for both *get* requests (*mid* = 1 and *cap* in the range from 2 to 3 as well as *mid* in the range from 1 to 3 and *cap* = 3) using just one instance of the data store. A table-less design and a multidimensional *key* provides TStorage with many access patterns and allows it

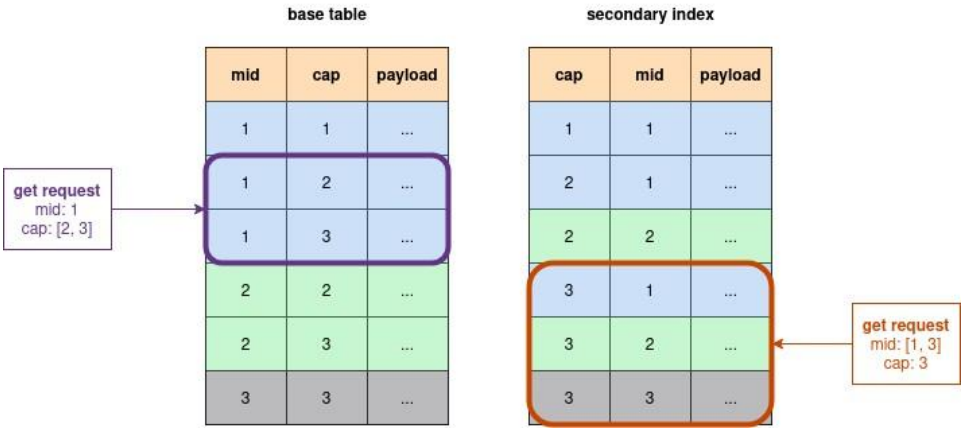

Figure 2: An example of a NoSQL database utilizing a secondary index to serve various *get* requests

KRAJOWY PLAN ODBUDOWY
Rzeczpospolita Polska
Sfinansowane przez Unię Europejską NextGenerationEU
NCBR Narodowe Centrum Badań i Rozwoju

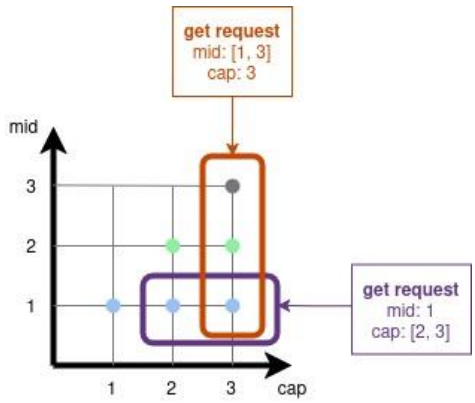

Figure 3: An example of TStorage natively serving various *get* requests

to save storage space in comparison with other databases. It also supports range queries in many dimensions at a time.

Change Data Capture

As shown in the previous section, the 5-dimensional *key* allows for an efficient access to data in TStorage; this also includes an access to different versions of records. Very often the data, such as measurements from an array of sensors, is constantly modified in real time. New records are inserted, and existing ones can be deleted or updated (for example, when an estimate is replaced with a previously unavailable measurement). These changes can be recorded and sent to third party systems to be audited, archived, or analyzed by analytics platforms. In such use cases, it is much more efficient to have the ability to capture only the records that are modified instead of constantly scanning the whole database for changes.

The mechanism for tracking and storing changes to data records is called *Change Data Capture* (CDC); the term *change stream* is also used by some vendors. Usually, this functionality is achieved with a supplemental component and requires additional memory for storage. For example, Amazon DynamoDB offers two solutions [52]: DynamoDB Streams, with data retention time of 24 hours [53], and Kinesis Data Streams, with data retention time up to 1 year, and priced separately from the database [54]. The plugin library of InfluxDB contains State Change Plugin for monitoring, among other things, changes of values in rows [55]. In ScyllaDB to query the history of all changes made to the table, a user has to enable CDC, which results in the creation of an additional entity called *log table* that records information about what field in the row was changed, what was its value before the change, and what is its current value [56]. Similarly, Apache

KRAJOWY PLAN ODBUDOWY
Rzeczpospolita Polska
Sfinansowane przez Unię Europejską NextGenerationEU
NCBR Narodowe Centrum Badań i Rozwoju

Cassandra offers CDC and stores the associated information in the CDC log [57]. Since Amazon Keyspaces is compatible with Apache Cassandra, it also supports CDC; row-level modifications are stored in a log for up to 24 hours [58]. Both Spanner and Bigtable provide *change streams* to record the modification type (insert, update, or delete) and a new value of the record; Spanner additionally can hold the old value. A retention period of *change streams* in Bigtable is between 1 to 7 days [59], whereas in Spanner between 1 to 30 days [60]. In both cases, enabling *change streams* requires additional storage space subject to storage costs [59, 60]. In MongoDB the CDC functionality is also known as *change stream*, and it allows external applications to access real-time data changes. Moreover, it has the option to output the document before and after changes; however, this option requires additional storage space and adds processing time [61]. In Azure Cosmos DB, *change feed* has two modes: one captures the latest change (insert or update) to a document whenever it happened during the document's lifetime, and the other captures all the changes (inserts, updates, deletes) but only during a specified period [62]; moreover, the second mode requires additional storage provided by the *continuous backup* which is free in case of a 7-day period but comes with a monthly fee in case of a 30-day period [63].

Meanwhile, TStorage does not need any supplemental components or additional memory to provide the functionality of *Change Data Capture*. Since the *acq* component of the *key* timestamps the moment the record was acquired by the database, the historical versions of data are available out-of-the-box. Moreover, the access to these historical versions is unlimited, no matter how far into the past they were acquired by the database. Due to the functionality provided by *acq*, the user is free to request the latest version of data or to continuously track the changes of data (such as inserts, updates, and deletes) and make them available to third party systems. The next section is devoted to these two different access patterns in time domain.

Two types of access patterns in time domain

In the time domain, there are two major use cases for data retrieval:

- requesting the latest version of data,
- tracking the incoming data.

Both of them are natively supported by TStorage thanks to the *acq* parameter of the *key* that acts as a timestamp for when the data was acquired by the database. The latest version of data is useful for:

- **The users who want to control their consumption of resources (e.g. electrical energy, gas, computing power, etc.).** To achieve that, it is

KRAJOWY PLAN ODBUDOWY
Rzeczpospolita Polska
Sfinansowane przez Unię Europejską NextGenerationEU
NCBR Narodowe Centrum Badań i Rozwoju

common to examine the newest version of data in a spreadsheet or a graph. The old version of data usually corresponds to an inaccurate value corrected by the new one.

- **Systemsperformingdataanalytics.** Thesesystems(includingtheAI-based) treat the newest version of data as the version that corrects previous inaccuracies.

There are also scenarios requiring the tracking of incoming data:

- **Billing of customers.** The customers are billed based on an actual consumption of resources or system utilization that is captured in a selected period. Examples of such resources include but are not limited to: electrical power or gas, computing power, disk space, number of requests served by a database, or tokens for interaction with AI agents. These areas are of particular interest to utility companies, AI services enterprises, as well as cloud service providers offering web hosting, data storage, and cluster computing services.
- **Invoice correction.** A system can capture a new value of a parameter that was used to bill the customer, such as a new value of energy consumption for a previous billing period. Then, the invoice issued based on the old data can be corrected.
- **Transfer of data to third party systems.** The data can be transferred to third party systems in a period that is independent from data acquisition. There is no need for additional data buffering like in a typical Change Data Capture solution. TStorage allows for flexible changes of the requested period of data without the need for any change of resource allocation, as it does not require any additional resources such as a separate table for recording modifications to data.
- **Validation of incoming data.** The incoming data can be validated in a flexible period.
- **Backup of data.** Since the backup of data does not require any special mechanism to preserve consistency, it can be scheduled based on regular *get* and *put* requests.

Conclusion

The main assets of TStorage, achieved by its utilization of a multidimensional *key*, can be summed as:

KRAJOWY PLAN ODBUDOWY
Rzeczpospolita Polska
Sfinansowane przez Unię Europejską NextGenerationEU
NCBR Narodowe Centrum Badań i Rozwoju

1. **Data model tuned to meet the IoT needs.** A 5-component *key* fits well to the variety of quantities measured.

2. **Security possible by *key-value* separation.** The measurement is represented by a *data record*. In turn, this data record is split into two parts: a *key* and a *value* that holds a *payload*. The *key* contains data that allows for localization of the measured quantity but itself is meaningless. Meanwhile, the *payload* contains the result of the measurement, but for the system it is a set of bits, so it can be stored in an encrypted form. System administrator does not need to know how the *payload* is encrypted, and different users can have their own encryption strategies.

3. **Flexible access pattern.** In the traditional NoSQL databases to allow multiple access scenarios, it is required to make multiple copies of data with different column selected as a *partition key*. Additional storage space and necessary synchronization of multiple copies comes up with an extra cost. Meanwhile, multi-access pattern to data stored in TStorage does not need redundancy. When requesting data, the users are free to choose any range on each of the 5 components of the *key*.

4. **Consistency appropriate for data analysis and external system oriented on increments of data (e.g., billing).** Two of the key components represent the time domain: the time the measurement was taken (*cap*) and when it was acquired by the system (*acq*). The latter allows for tracking of incoming data and consistent data versioning.

However, an optimal implementation of the database that utilizes a 5-dimensional *key* is not a trivial task. In the upcoming paper, the results of performance tests will prove that TStorage is indeed efficient in writing and reading operations.

Acknowledgments

This project is co-financed by the European Union as part of the IPCEI-CIS initiative under the contract number KPOD.05.10-IW.10-0001/24.

References

[1] GLOBE NEWSWIRE. Steady Growth Ahead: Smart Grid Market to Surge by 17% CAGR from 2025 to 2030. https://www.globenewswire.com/ news-release/2025/05/05/3073808/28124/en/Steady-Growth-

KRAJOWY PLAN ODBUDOWY
Rzeczpospolita Polska
Sfinansowane przez Unię Europejską NextGenerationEU
NCBR Narodowe Centrum Badań i Rozwoju

Ahead-Smart-Grid-Market-to-Surge-by-17-CAGR-from-2025-to2030.html, 2025. Accessed: 2025-07-16.

[2] J. Hawran, E. Taillanter, L. Wurker, J. Zahler, P. Vollmuth. Technical Frame-¨ work Conditions for Demand-Side Flexibility – A Comparison of Smart Meter Architectures Enabling Demand-Side Flexibility in France, the UK, and Germany. White Paper 10.34805/ffe-11-25, Forschungsstelle fur En-¨ ergiewirtschaft e.V., 2025.

[3] CEN-CENELEC-ETSI. Functional reference architecture for communications in smart metering systems. Technical Report CEN/CLC/ETSI/TR 50572, 2011.

[4] Statista. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2023, with forecasts from 2024 to 2028. https://www.statista.com/statistics/871513/worldwidedata-created, 2024. Accessed: 2025-07-16.

[5] M. Schulman, A. Joshi. Oracle NoSQL Database For Time Series Data. https://www.oracle.com/technetwork/database/databasetechnologies/nosqldb/overview/timeseries-2017-v54118485.html, 2017. Accessed: 2025-10-20.

[6] ScyllaDB, Inc. Cassandra Time Series Data Modeling. https://www.scylladb.com/glossary/cassandra-time-series-datamodeling/. Accessed: 2025-10-20.

[7] Google. Schema design for time series data. https://cloud.google.com/bigtable/docs/schema-design-time-series. Accessed: 2025-10-20.

[8] MongoDB, Inc. Time Series. https://www.mongodb.com/docs/manual/core/timeseries-collections/. Accessed: 2025-10-20.

[9] Amazon Web Services, Inc. What is Amazon DynamoDB? https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html. Accessed: 2025-07-29.

[10] M. Elhemali, N. Gallagher, N. Gordon, J. Idziorek, R. Krog, C. Lazier, E. Mo, A.Mritunjai, S.PerianayagamT.Rath, S.Sivasubramanian, J.C.SorensonIII, S. Sosothikul, D. Terry, A. Vig. Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service. *Proceedingsofthe 2022 USENIX Annual Technical Conference*, page 1037–1048, 2022.

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[12] Oracle. Oracle NoSQL Database Release 25.1. https://docs.oracle.com/en/database/other-databases/nosql-database/25.1/index.html. Accessed: 2025-09-15.

[13] InfluxData, Inc. InfluxDB 3 Enterprise documentation. https://docs.influxdata.com/influxdb3/enterprise/. Accessed: 2025-09-18.

[14] Amazon Web Services, Inc. Amazon Timestream. https://aws.amazon.com/timestream/. Accessed: 2025-09-15.

[15] Amazon Web Services, Inc. Amazon Timestream for LiveAnalytics availability change. https://docs.aws.amazon.com/timestream/latest/developerguide/AmazonTimestreamForLiveAnalyticsavailability-change.html. Accessed: 2025-09-15.

[16] Amazon Web Services, Inc. What is Timestream for InfluxDB? https://docs.aws.amazon.com/timestream/latest/developerguide/ timestream-for-influxdb.html. Accessed: 2025-09-15.

[17] ScyllaDB, Inc. ScyllaDB Documentation. https://docs.scylladb.com/manual/stable/index.html. Accessed: 2025-09-18.

[18] The Apache Software Foundation. Cassandra Documentation. https://cassandra.apache.org/doc/latest/index.html. Accessed: 2025-0912.

[19] Apache HBase Team. Apache HBase® Reference Guide. https://hbase.apache.org/book.html. Accessed: 2025-09-12.

[20] Amazon Web Services, Inc. Amazon Keyspaces: How it works. https://docs.aws.amazon.com/keyspaces/latest/devguide/howit-works.html. Accessed: 2025-09-15.

[21] Google. Spanner documentation. https://cloud.google.com/spanner/ docs. Accessed: 2025-09-08.

[22] Google. Bigtable overview. https://cloud.google.com/bigtable/docs/overview. Accessed: 2025-09-08.

[23] Google. Schema design best practices. https://cloud.google.com/bigtable/docs/schema-design. Accessed: 2025-09-08.

[24] Apache HBase Team. Apache HBase® Reference Guide. Data Model. https://hbase.apache.org/book.html#datamodel. Accessed: 202509-12.

[25] Redis. Develop with Redis. https://redis.io/docs/latest/develop/. Accessed: 2025-09-11.

[26] Valkey contributors. Valkey. https://valkey.io/. Accessed: 2025-09-04.

[27] Dhruba Borthakur et al. RocksDB Overview. https://github.com/facebook/rocksdb/wiki/RocksDB-Overview. Accessed: 2025-09-18.

[28] Amazon Web Services, Inc. What is MemoryDB. https://docs.aws.amazon.com/memorydb/latest/devguide/what-is-memorydb.html. Accessed: 2025-09-04.

[29] MemcachedOverview. https://docs.memcached.org/. Accessed: 202509-04.

[30] Amazon Web Services, Inc. What is Amazon ElastiCache? https://docs.aws.amazon.com/AmazonElastiCache/latest/dg/WhatIs.html. Accessed: 2025-09-04.

[31] Microsoft. Azure Cosmos DB documentation. https://learn.microsoft.com/en-us/azure/cosmos-db/. Accessed: 2025-09-18.

[32] MongoDB, Inc. What is MongoDB? https://www.mongodb.com/docs/manual/. Accessed: 2025-09-18.

[33] Amazon Web Services, Inc. What is Amazon DocumentDB (with MongoDB compatibility). https://docs.aws.amazon.com/documentdb/latest/developerguide/what-is.html. Accessed: 2025-09-04.

[34] Amazon Web Services, Inc. Core components of Amazon DynamoDB. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.CoreComponents.html. Accessed: 2025-09-12.

[35] Oracle. Choice of Keys in NoSQL Database. https://docs.oracle.com/en/database/other-databases/nosql-database/25.1/nsdev/ choice-keys-nosql-database.html. Accessed: 2025-09-15.

[36] InfluxData, Inc. InfluxDB schema design recommendations. https://docs.influxdata.com/influxdb3/enterprise/writedata/best-practices/schema-design/. Accessed: 2025-09-18.

[37] ScyllaDB, Inc. Data Definition. https://docs.scylladb.com/manual/stable/cql/ddl.html. Accessed: 2025-09-18.

[38] The Apache Software Foundation. Introduction. https://cassandra.apache.org/doc/latest/cassandra/developing/datamodeling/intro.html. Accessed: 2025-09-12.

[39] Google. Schemas overview. https://cloud.google.com/spanner/docs/schema-and-data-model. Accessed: 2025-09-08.

[40] Redis. Keys and values. https://redis.io/docs/latest/develop/ using-commands/keyspace/. Accessed: 2025-09-11.

[41] Redis. Redis as an in-memory data structure store quick start guide. https://redis.io/docs/latest/develop/get-started/data-store/. Accessed: 2025-09-11.

[42] Amazon Web Services, Inc. Improving data access with secondary indexes in DynamoDB. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/SecondaryIndexes.html. Accessed: 202509-12.

[43] Oracle. Using Indexes in NoSQL Database. https://docs.oracle.com/en/database/other-databases/nosql-database/25.1/nsdev/ using-indexes-nosql-database.html. Accessed: 2025-09-15.

[44] InfluxData, Inc. Manage file indexes. https://docs.influxdata.com/influxdb3/enterprise/admin/file-index/. Accessed: 2025-09-18.

[45] ScyllaDB, Inc. Global Secondary Indexes. https://docs.scylladb.com/manual/stable/features/secondary-indexes.html. Accessed: 2025-09-18.

[46] The Apache Software Foundation. Indexing concepts. https://cassandra.apache.org/doc/latest/cassandra/developing/cql/indexing/indexing-concepts.html. Accessed: 2025-09-12.

[47] Google. Secondary indexes. https://cloud.google.com/spanner/docs/secondary-indexes. Accessed: 2025-09-08.

[48] Google. Create an asynchronous secondary index. https://cloud.google.com/bigtable/docs/asynchronous-secondary-index. Accessed: 2025-09-08.

[49] Microsoft. Azure Cosmos DB for NoSQL global secondary indexes

(preview). https://learn.microsoft.com/en-us/azure/cosmos-db/nosql/global-secondary-indexes. Accessed: 2025-09-18.

[50] MongoDB, Inc. Indexes. https://www.mongodb.com/docs/manual/indexes/. Accessed: 2025-09-18.

[51] Amazon Web Services, Inc. Data modeling with Amazon DocumentDB. https://d1.awsstatic.com/product-marketing/Data%20modeling%20with%20Amazon%20DocumentDB.pdf. Accessed: 202509-04.

[52] Amazon Web Services, Inc. Change data capture with Amazon DynamoDB. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/streamsmain.html. Accessed: 2025-09-12.

[53] Amazon Web Services, Inc. Change data capture for DynamoDB Streams. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html. Accessed: 2025-09-12.

[54] Amazon Web Services, Inc. Using Kinesis Data Streams to capture changes to DynamoDB. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/kds.html. Accessed: 2025-09-12.

[55] InfluxData, Inc. State change plugin. https://docs.influxdata.com/influxdb3/enterprise/plugins/library/official/statechange/. Accessed: 2025-09-18.

[56] ScyllaDB, Inc. CDC Overview. https://docs.scylladb.com/manual/stable/features/cdc/cdc-intro.html. Accessed: 2025-09-18.

[57] The Apache Software Foundation. Change Data Capture. https://cassandra.apache.org/doc/latest/cassandra/managing/operating/cdc.html. Accessed: 2025-09-12.

[58] Amazon Web Services, Inc. How change data capture (CDC) streams work in Amazon Keyspaces. https://docs.aws.amazon.com/keyspaces/latest/devguide/cdc_how-it-works.html. Accessed: 2025-09-15.

[59] Google. Change streams overview. https://cloud.google.com/bigtable/docs/change-streams-overview. Accessed: 2025-09-08.

[60] Google. Change streams overview. https://cloud.google.com/spanner/docs/change-streams. Accessed: 2025-09-08.

[61] MongoDB, Inc. Change Streams. https://www.mongodb.com/docs/manual/changeStreams/. Accessed: 2025-09-18.

[62] Microsoft. Change feed in Azure Cosmos DB. https://learn.microsoft.com/en-us/azure/cosmos-db/change-feed. Accessed: 2025-09-18.

[63] Microsoft. Continuous backup with point-in-time restore in Azure Cosmos DB. https://learn.microsoft.com/en-us/azure/cosmos-db/ continuous-backup-restore-introduction. Accessed: 2025-10-01.