Bachelor's Thesis

Optimizing Storage of Energy Event Data in In-Memory Databases

Optimierte Ablage von Energieereignisdaten in Hauptspeicherdatenbanken

Leonhard Schweizer
leonhard.schweizer@student.hpi.uni-potsdam.de

Hasso Plattner Institute for IT Systems Engineering
Enterprise Platform and Integration Concepts Chair

HPI Hasso Plattner Institut
IT Systems Engineering
Universität Potsdam

August-Bebel-Str. 88
14482 Potsdam, Germany
`http://epic.hpi.uni-potsdam.de/`

Supervisors:

Dr. Alexander Zeier
Matthieu-P. Schapranow
Christian Schwarz

Hasso Plattner Institute
Potsdam, Germany

June 30th, 2011

Zusammenfassung Im Zuge der Einführung von Smart Metering in Deutschland werden allein Privatkunden jährlich ca. 1.4 Billionen Datensätze durch ihre Stromzähler erzeugen. Energieanbieter müssen in anderen Worten entsprechend dem Messintervall moderner Zähler viertelstündlich 1.8GB an Rohdaten verarbeiten. Die Verarbeitung von kontinuierlichen Datenströmen dieser Größe stellt heutzutage eine große Herausforderung dar und herkömmliche OLAP Systeme sind nicht dazu in der Lage, derart große Datenmengen in Echtzeit zu analysieren. Aus diesem Grund werden Zählerstände bisher höchstens einmal täglich an die Energieanbieter gesendet und Analysepotentiale bleiben ungenutzt. In dieser Arbeit wird ein auf Hauptspeicherdatenbanken basierender Ansatz zur Echtzeitverarbeitung von Energieereignisdaten beschrieben. Darüber hinaus wird gezeigt, wie der Speicherplatzbedarf von Energieereignisdaten durch die Ausnutzung von Kompressionspotentialen in spaltenorientierten Tabellen drastisch gesenkt werden kann. Infolgedessen entstehen ungeahnte Möglichkeiten. Beispielsweise können Energiedaten in sekundenschnelle analysiert werden und dynamische Tarife, deren Preis sich an Angebot und Nachfrage orientiert, werden ermöglicht.

Abstract. In the course of the ongoing implementation of smart metering in Germany, residential customers alone will produce roughly 1.4 trillion records per year through their power meters. In other words, energy providers will have to deal with 1.8GB of raw data every 15 minutes, which is the default measurement interval of modern metering devices. The processing of continuous data streams of this dimension is a big challenge today and traditional OLAP systems aren't capable of analysing this huge amount of data in real-time. Thus, meter readings are sent to the providers at most once per day and analytical possibilities remain unused. This thesis describes an approach to the real-time processing of energy event data. By chosing an in-memory database as storage they can be processed and analyzed simultaneously while notably reducing the amount of required space at the same time through the utilization of compression potentials in column-based tables. As a result, new opportunities arise, like offering electricity rates with real-time pricing or managing supply and demand based on up-to-the-minute analytics.

Table of Contents

1 Introduction

Traditionally, database management systems are split into two categories [15]. On the one hand, there are write-optimized, row-oriented Online Transactional Processing (OLTP) systems, which in return lack in analytical performance. On the other hand, there are read-optimized Online Analytical Processing (OLAP) systems, which aren't suitable for transactional processing.

This is one of the reasons why utility companies today aren't capable to take full advantage of the data flood arising from a future nationwide smart metering infrastructure. For instance, the growing importance of renewable and distributed energy sources like wind and solar energy as part of the aimed-at energy turnaround leads to an increased need of information. Despite their unpredictable nature, energy providers will have to offer available-to-promise functions. While a smart grid will deliver the required data, there are no systems that enable the involved parties to gain the information which is neccesary to balance the short-term demand and the volatile energy input of renewable energy sources out of it in real-time.

At the same time, suppliers of electric energy are forced to offer new contract conditions and tariff models to their customers. Traditionally, consumers have to pay a fixed rate every month regardless of the actual consumption until rates can be adjusted accordingly when their power meters are read manually once per year. To increase cost transparency to consumers, billing intervals are supposed to be shortened to months at least [3]. But automated meter reading even offers the data to decrease this interval even further, e.g. to days or hours. Such short billing intervals in turn enable the consumers to change tariffs or energy providers with a much higher frequency as compared to the long minimum contract durations offered today. Again, technical insufficiencies are the primary reason why this isn't already happening.

With the introduction of in-memory databases, the promise has been made that the separation of OLTP and OLAP systems becomes superfluous, in particular with the help of column oriented tables [29] - a technique that was originally introduced as highly read-optimized approach by Stonebreaker et. al. [36]. In-memory technology has the potential to close the gap between the real-time capturing of the data amassing in a smart metering infrastructure and the real-time acquisition of information out of this data. In this thesis, it is shown how an in-memory database can be utilized to store the amount of meter readings which are assumed to arise after a Germany-wide implementation of smart metering in real-time with the help of SAPs In-Memory Computing Engine. It is further shown how the memory footprint of this data can be minimized through lightweight column compression techniques.

2 Simulation of an Advanced Metering Infrastructure

As an extensive rollout of smart metering devices and the corresponding metering infrastructure hasn't taken place yet in Germany, a simulation had to be used to generate a constant event stream of significant scale in order to allow for the evaluation of its processing and analysis. In this chapter, the characteristics of such an infrastructure are depicted as well as the mapping of this infrastructure to the simulation model. Finally, the experiences gained from the execution of the simulation system are summarized in Sect. 2.4.

2.1 AMI Characteristics

The smart metering infrastructure is still in the early stages of development in Europe. While the installation of smart electricity and gas meters recently became required by law in Germany for newly constructed buildings and in the case of total refurbishments [8], the automated reading of power meters is only realized in a few small model regions. There is no Germany-wide metering infrastructure. However, there seems to be a broad consent to the characteristics such an architecture has to feature [2,21,28]. Figure 1 depicts the most important components using the FMC notation [14].

In this architecture, which is commonly referred to as Advanced Metering Infrastructure (AMI), every household is equipped with a smart meter. Smart meters are digital meters which are equipped with a CPU, storage and communication interfaces. They collect information about the recent energy consumption in fixed time intervals, e.g. quarters of an hour, and send it to the energy provider for billing purposes. The smart meters are connected to intermediary data collectors which concentrate the meter readings before forwarding them to the utility companies. The connections between smart meters, collectors and the utility companies can be established via various channels, like ethernet, powerline communication or GSM.

Since standardization is not well advanced in the field of smart metering, it is safe to assume that smart meters from different manufacturers will use different data formats. Therefore, so-called Meter Data Unification and Synchronization

(MDUS) systems are introduced in order to harmonize the different protocols of different vendors.

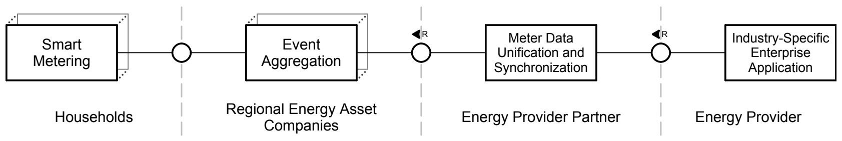
Smart Metering
Event Aggregation
R
Meter Data Unification and Synchronization
R
Industry-Specific Enterprise Application
Households
Regional Energy Asset Companies
Energy Provider Partner
Energy Provider

Fig. 1. The Advanced Metering Infrastrucutre as depicted in [33]

2.2 Simulation Requirements

Since the data generated by the simulated smart meters should not only be used for stress testing, but also for real-time analyses and visualization, various requirements have to be met by the simulation system.

The system has to be capable to simulate at least 100 million smart meters, each initiating one reading event per 15 minutes. Furthermore, the number of simulated metering devices should be freely configurable. Thus, the system can not only be used to simulate divergent numbers of customers for energy providers of different size, but also to simulate the projected total amount of events in the future smart grid of Germany.

In avoidance of unfavorable and advantageous effects of random data, for instance on compression rates, and with regard to data analysis and visualization tests, the generated readings should be based on real power consumption data. Furthermore, the simulation should be aware of single smart meters. That means that for a given meter id, reading events should occur in 15 minute intervals. Besides, the unity of all readings of any simulated smart meter should form a realistic consumption behaviour over extended periods of time, e.g. days, weeks, months and years.

2.3 Implementation

The simulation of an Advanced Metering Infrastructure has been split up into three components, which are depicted in Fig. 2. However, not all parts of the AMI have a counterpart. Namely the Meter Data Unification and Synchronization (MDUS) system is missing, since there is no gain in simulating different vendor specific protocols and data formats. In the first place, the MDUS systems don't have to store data, in fact they transform it. That means that in order to scale, only the throughput has to be enhanced, which can be achieved easily by additional hardware and parallelisation. Therefore, the MDUS system should not constitue the AMIs limiting factor.

To a greater degree, the database which has to process, to store and to analyse all readings is the bottleneck of the whole system. At this point, scaling can not be reached simply by adding more hardware, in the sense of multiplying database hosts and instances. For example, all instances would have to be utilized for analytical queries like the overall consumption of all customers, with the effect that the partitioning of the data doesn't necessarily result in a lower overall load for each single instance.

Thus, the primary target of the simulation is the generation of huge amounts of insert load rather than a realistic representation of the AMI. In this way, the limits of an in-memory based central system can be investigated. All other components of the metering infrastructure can be scaled easily, which is the reason why this paper focuses on the performance of the database system itself.

In the following, the data producer constituting the smart metering component of the AMI, the concentrator, which embodies the event aggregation component and the database client as representation of the interface of the industry-specific enterprise application are described in detail.

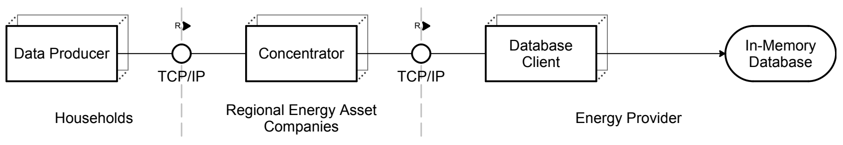

Fig. 2. Components of the AMI simulation

Data Producer An instance of the data producer, which is implemented as Java executable, represents any number of smart meters. One instance of the data producer can connect to exactly one concentrator. A dedicated, persistent TCP/IP connection is built between the data producer and a concentrator for each thread the data producer is spawning. One thread simulates up to 375,000 smart meters. The number of smart meters that can be simulated by a single instance is mostly limited by the number of threads the host system is capable to handle and the network capacity.

The data producer expects two main input parameters: An initial timestamp, which defaults to the current local time, and a standard load profile, which defaults to the H0 profile published by the BDEW[1]. Such profiles contain the average consumption of a specific customer base (which are residential customers in the case of the H0 profile) over a period of one or more years. They consist of counter readings or consumption deltas in 15 minute intervals, yielding 35040 values for non-leapyears. For the reasons depicted in Sect. 4.2, consumption deltas are preferred over counter readings.

Based on these two input parameters, readings are generated. The initial timestamp is used to calculate the current simulated day and time. The according consumption value is read from the standard load profile. A uniformly distributed variance is added to this value v for every generated reading r_{v} such that $0 \leq v - 0,20 \cdot v \leq r_{\mathrm{v}} \leq v + 0,20 \cdot v$. The timestamp of the generated reading is rounded down to 15 minute intervals. For instance, 13:14:55 would be replaced by 13:00:00 (cf. 4.2).

The third and last component of a reading is the unique integer device id by which every simulated smart meter can be identified. Each smart meter is associated with one customer, and each customer can have an arbitrary number of smart meters. This mapping information is not part of the table containing the meter readings.

The data producer generates discrete reading events for each of these identifiers every 15 minutes and sends them to the assigned concentrator instantly. The identifiers are spread randomly accross the 15 minute intervall, but keep their time slot across multiple intervals as long as the simulation is running. Table 1 shows one hour of readings for a given smart meter.

Since the simulation is executed over an ethernet network via TCP/IP, data loss is of no concern and the push architecture described here could be favored over the pull approach which dominates in real world metering infrastructures.

Concentrator Just like the data producer, the concentrator is implemented as Java executable. Its purpose is to aggregate events and to forward these aggregated batches to a database client, reducing the number of connections and insert events visible to the central database system.

[1] German Energy and Water Association, www.bdew.de

Table 1. Sample readings generated by the data producer

Smart Meter ID	Timestamp	Value [Wh]
32202775	1306612800	36
32202775	1306613700	37
32202775	1306614600	35
32202775	1306615500	35

A concentrator accepts any number of TCP/IP connections from any number of data producers. The concentrator can connect to one or more database clients, again via TCP/IP. The primary reason for supporting multiple database clients is the evaluation of distributed database systems, which is the subject of another thesis [6]. In this case, the concentrator is provided with the data partitioning instructions and forwards the readings to the proper destination database instance.

The concentrator collects incoming readings until the batch size reaches a configurable threshold (default: 100,000 readings) or the oldest reading in the batch exceeds a certain age (default: 5 minutes). Once this happens, the batch of readings gets converted into a column-wise fashion and is sent to the corresponding database client together with the number of readings contained in the batch. Figure 3 illustrates this conversion with the aid of the first two records of the row-based data shown in Table 1. The resulting column-based format is very beneficial for inserting data into column-based tables.

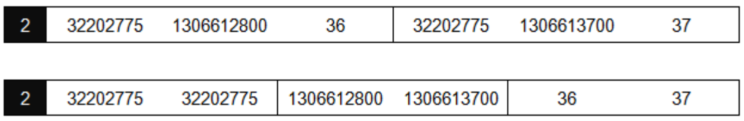

Fig. 3. The first two records from Table 1 in row format and formated as output of a concentrator

Database Client The main purpose of the database client is to expose a specialised interface for inserting batches of readings into the in-memory database, the back-end of the simulation system.

It accepts TCP/IP connections from an arbitrary number of concentrators and inserts the incoming batches into the database in a non-blocking fashion. Thus, it can be avoided that the database client becomes the bottleneck of the system rather then the database itself. Since the concentrators already convert the readings into the desired format, no additional transformations of the data have to be carried out by the database client. The measures that have been taken to speed up the process of inserting as much as possible in order to reduce the overall load of the database and to enable the simultaneous processing of analytical requests are presented in Sect. 3.

Since the database client is intended for running on the same host as the database system, it is important to reduce the footprint of the client. For that reason, the client is implemented as C++ native executable and connected to the database via ODBC. The C++ implementation reduces the main memory consumption from a maximum of 2.2GB to 110MB compared to a corresponding Java implementation.

2.4 Simulation Environment and Execution

According to the Federal Statistical Office there are 40.2 million households and 3.6 million businesses in Germany [34,35]. Since many companies have branches at multiple locations and huge branches will be equipped with multiple smart meters, the total number of smart meters is assumed to be around 60 million for non-residential customers. Thus, a total of 100 million smart meters can be expected after a nationwide rollout in Germany. For that reason, the primary goal of the simulation is to generate the load of 100 million smart meters which send meter readings in 15 minute intervals.

For the generation of the meter readings, four workstations with equipment equivalent to host HPC are used. On each of them, one single instance of the data producer is running, simulating 25 million smart meters. The data producer is connected to exactly one concentrator which is running on the same host. Since the simulated smart meters are distributed uniformly accross the 15 minute interval, each concentrator receives roughly 28,000 meter readings per second. Each concentrator is connected to a dedicated instance of the database client which is running on the host of the database HPB and inserting incoming readings into an instance of the table READINGS_RLE (cf. Listing 4).

The frequency of incoming data events is dependent on the batch size of the concentrator. Lower batch sizes lead to a smooth CPU workload on the database system but result in a higher overhead due to the increased number of executed INSERT-statements. When using 10,000 as batch size, every concentrator forwards data more than twice per second. The result is a CPU utilization of 40%-80% through the database. In contrast, a batch size of 1,000,000 leads to peaks of 170% when data packets are received from a concentrator every 36 seconds. During this interval, there is no CPU load caused by the insertion of new readings. In both cases, the CPU load increases up to 300% during executions of the merge process, since the test system is configured to utilize at most three cores during this task.

For the purpose of planning security, a smooth resource utilization is the preferable option. With regard to the optimization of insert performance, the highest possible batch size is the first choice. As real-time analysis only makes sense if the latest data is added to the database in real-time, the maximum turnaround time of meter readings is another constraint when chosing the best batch size. As trade-off of all this considerations, a batch size of 100,000 is chosen. That means that a new meter reading will be transported to the database system in under four seconds.

The CPU load of this configuration varies between 30% and 110%. This leaves enough ressources for the simultaneous execution of analytical queries which is investigated in another thesis [27]. The execution time after 1000 inserts averages 410ms in this scenario. That means that the target insert rate of 112,000 readings per second which is required to process the 100 million smart meters could be outperformed considerably. It is important to note that this time measurement is even falsified by a deficient implementation of the merge process (cf. 3.4) which blocks pending INSERT-statements during a merge although it should be carried out asynchronously [16].

During the execution of the simulation, three main problems became manifest. First of all, early implementations of the database client weren't performant enough to insert the incoming meter readings fast enough to avoid congestion. The measures that were taken to solve this problem are explained in Sect. 3. Secondly, first projections of the required amount of main memory for the depicted scenario were too high to allow for a real world implementaion, which lead to the investigation of compression potentials presented in Sect. 4. In the third place, the execution time of the merge process turned out to grow exponentially with the size of the dataset. But since Krüger et. al. [16] have already shown that the

```
CREATE COLUMN TABLE readings (meterid INTEGER, timestamp
    INTEGER, value INTEGER, PRIMARY KEY (meterid, timestamp))
```

Listing 1. Schema READINGS_PK

```
CREATE INSERT ONLY COLUMN TABLE readings (meterid INTEGER,
    timestamp INTEGER, value INTEGER, PRIMARY KEY (meterid,
    timestamp))
```

Listing 2. Schema READINGS_IO_PK

merge process can be carried out in linear dependency of the size of the dataset, this problem isn't considered in this thesis.

3 Acceleration of INSERT-Statements

In order to support a constant data stream originating from as many smart meters as possible, it is important to reduce the execution time of INSERT-statements to the greatest possible extent. The measures that have been taken into consideration for that reason are depicted and evaluated in this section.

Unless stated otherwise, all measurements in this section have been made under the following conditions: The database client is running on host HPC and connecting to a NewDB instance on host HPA via JDBC. The hosts are connected via Gigabit LAN. The execution of INSERT-statements takes place on empty tables, whereas each table is generated before and droped after each test run. Autocommit is disabled as well as automerge and merge times are not included in the measurements. Transactional logging isn't in effect during inserts. Time measurements are taken with the help of the Java getTimeInMillis()-API [25].

Three different schemas have been used for the measurements. The schema READINGS_PK (Listing 1) constitutes a regular column table with a natural primary key on the columns meterid and timestamp. The schema READINGS_IO_PK (Listing 2) is equivalent to READINGS_PK, except that it represents an insert-only column table. The third schema is READINGS_IO (Listing 3), which is an insert-only column table with omitted explicit primary key.

```
CREATE INSERT ONLY COLUMN TABLE readings (meterid INTEGER,
    timestamp INTEGER, value INTEGER)
```

Listing 3. Schema READINGS_IO

3.1 Prepared Statements

The naive approach to insert multiple rows into a table is to execute single INSERT-statements sequentially. The problem in doing so is that parsing and the generation of an execution plan is carried out again for each statement, although they are equivalent. The solution to this issue is the utilization of prepared statements. As its name implies, afore-mentioned tasks get executed only once and the statement can be reused for subsequent inserts.

Figure 4 compares the performance of the usage of standard and prepared statements. First, 100,000 readings have been inserted into an empty instance of READINGS_PK by generating a new statement for each reading. After 10 runs, the total insert time averages 3 minutes 38.530 seconds (standard deviation: 3.281 seconds). When repeating the measurement using a prepared statement, the average insert time gets reduced to 51.974 seconds (standard deviation: 365ms), which is equivalent to a speed-up of almost 80%. With relation to the clear difference, the relatively small number of measurement iterations and the high standard deviation are still sufficient to emphasize the statement.

3.2 Batch Inserts

As an extension of prepared statements, most database interfaces offer a possibility to execute batches of the same statement at once. The executeBatch-method implemented by JDBC [26] or the array binding capabilities offered by ODBC [22] are examples for such functionality.

In contrast to the execution of 100,000 prepared, but discrete inserts (average: 51.974 seconds, standard deviation: 365ms), the insert time can be further reduced to 646ms (standard deviation: 33ms) with the help of batch insert mechanisms, a speed-up of 99%. In the case of batch inserts, the measurement was determined through 1000 runs and the measured time includes the generation and execution of a single prepared statement containing 100,000 rows, as well as a single commit at the end of the transaction. As with the measurement in Sect. 3.1, an empty instance of READINGS_PK was used for every run.

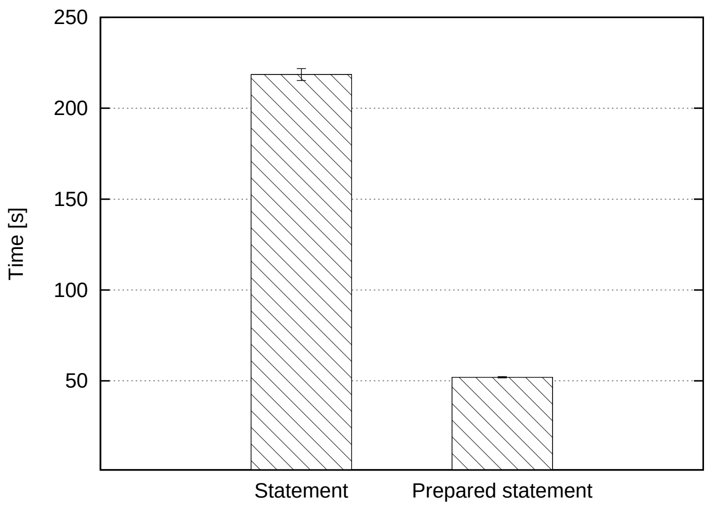

Fig. 4. Comparison of standard and prepared statements

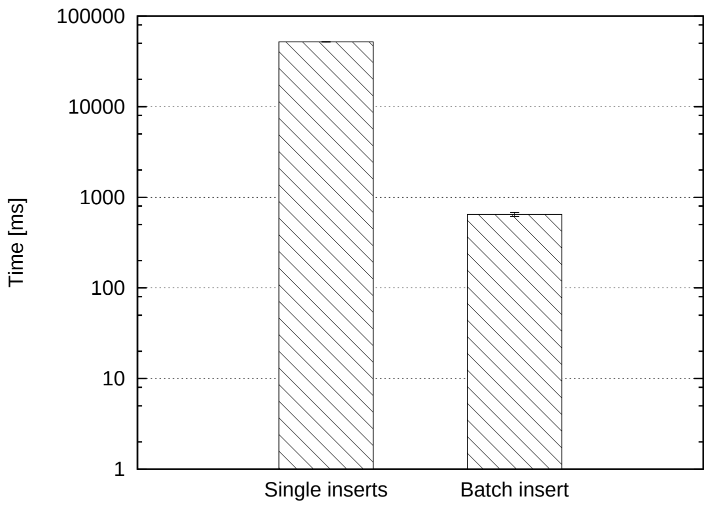

Fig. 5. Comparison of prepared statements and prepared batches

3.3 Omission of Natural Keys

When investigating the nature of a smart-meter reading, it becomes evident that the combination of its meter id and timestamp form a natural key that uniquely identifies each row. This key could very well serve as primary key for the table, as no two readings originating from the same meter may exist for a given point of time. However, this means that when inserting rows, time expensive tests on key violations would have to be performed. At the same time, there are no big drawbacks of allowing duplicate readings. Provided that meters normally don't record one point of time twice, saving such exceptional records might even be helpful for fraud and failure detection. Furthermore, keeping track of records might be a legal requirement in many countries [11].

The employed database system allows column tables without explicit primary key only in the form of so-called insert-only tables. As the name already suggests, records can only be added to a insert-only table, but neither be updated or deleted. These constraints can be accepted, since concurrent readings could be distinguished by timestamps or valid/invalid-flags and there are no reasons for frequent updates. Insert-only tables make use of an implicit row id as primary key which is comparable to auto-increment fields. While increasing the consumed amount of memory through this additional column, a considerable reduction of insert times can be achieved through this approach.

To evaluate the costs of explicit keys, batches of 100,000 readings have been inserted into the schemas READINGS_PK, READINGS_IO_PK and READINGS_IO. In each case, the measured time includes the generation of a prepared statement containing 100,000 rows, the execution of this statement on an empty table and a single commit at the end of this transaction. All three measurements have been repeated 1000 times.

The results are depicted in Fig. 6. Insertion into a regular column table with explicit primary key takes an average of 646ms (standard deviation: 33ms). Insert-only tables with an explicit primary key are only marginally faster, with an average of 612ms (standard deviation: 32ms). In contrast, insertions into a table without any explicit keys average 166ms, with a standard deviation of 16ms. That means that insert times can be reduced by approximately 76% through the omission of unnecessary keys.

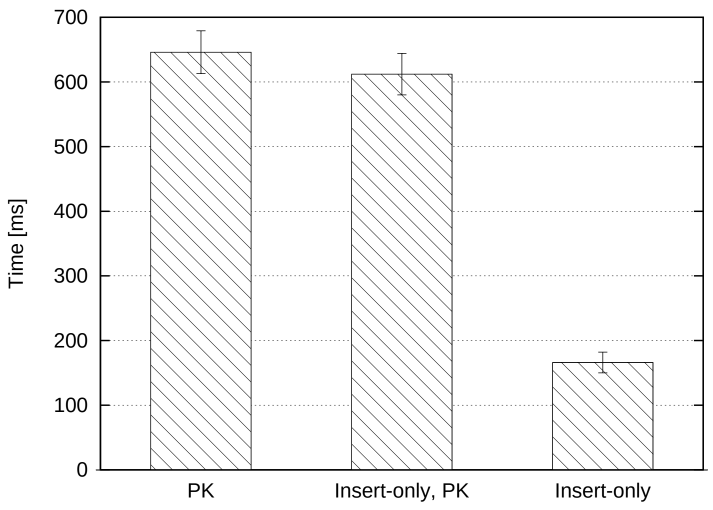

Fig. 6. Comparison of insert performance of tables with and without explicit primary key

3.4 Restraining Disaster Recovery

Transacation logs are an important measure to safeguard the compliance of the ACID properties, particularly with regard to atomicity and durability [12]. However, one could imagine to give up parts of the ACID properties in exchange for performance benefits due to the supposed architecture of an advanced metering infrastructure. In particular, the ability to answer requests of meter readings on demand is ranked as minimum requirement for smart meters [2], which can store the reading history of at least one year at the same time. This implies that it might be feasible to accept the loss of recent meter readings in the case of a database failure, since they could just be requested again.

The used database system splits up column stores into a read-optimized main store and a write-optimized differential buffer. New rows are inserted into the differential buffer and transfered to the main store during the so-called merge process [16,17]. If strict ACID compliance is desired, both of these structures have to be recovered in the case of a failure. However, all records are written to a non-volatile medium during the merge process, which means that the persistency of the main store is not depending on transaction logs once the merge is complete. From this follows that the impact of missing logs of the insert process itself is rather noncritical. In the worst case, all meter readings which haven't been merged yet would be lost temporarily.

The schema READINGS_IO was used to measure the impact of transaction logs on insert performance. Again, the measured time includes the generation of a prepared statement containing 100,000 rows, its execution on an empty table and a single commit at the end of this transaction. The measurement has been repeated 1000 times.

Figure 7 shows the results of this measurement. With activated logging, inserts average 177ms (standard deviation: 18ms). The same inserts take averagely 166ms (standard deviation: 16ms) when logging is disabled. In other words, transaction logs slow down the insert process by approximately 6%. Consequently, abandoning transaction logs for insertions can be considered doubtful for real world applications. The performance gain doesn't compensate the lost ability to recover unmerged readings in the case of a system failure.

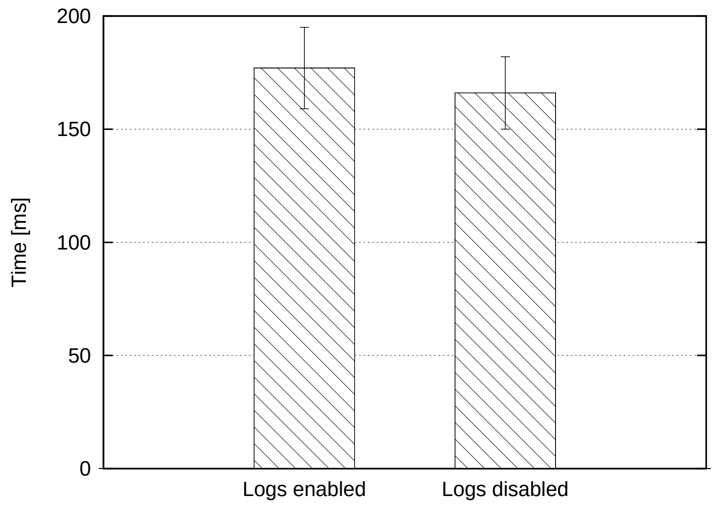

Fig. 7. Impact of transactional logging on insert performance

3.5 Parallel Execution of INSERT-Statements

In order to allow for a better evaluation of the impact of parallel inserts, a bigger dataset of 10,200,000 rows has been inserted into the Table READINGS_IO. The time measurement, which begins with the start of the first and ends with the return of the last spawned thread has been repeated 1000 times for n threads, with $n \in \{1, 3, 6, 12\}$. Thereby, the dataset is divided equally to all threads and each thread establishes a JDBC connection to the database and executes a single prepared INSERT-statement containing $\frac{10{,}200{,}000}{n}$ rows. Every thread initiates a commit before returning, so the measured time includes 1 commit for $n = 1$ and 12 commits for $n = 12$. The server HPA which hosts the database has a total of 12 cores.

The results of the measurement are depicted in Fig. 8. The insert time using one thread averages 29.935 seconds, with a standard deviation of 547ms. Based on this value, the theoretical optimum behaviour is plotted as $\frac{v_{t-1}}{2}$ for a number of t threads and the projected time measurement v_{t-1} of the preceding number of threads. In theory and without considering the overhead of multithreading, the insert process should take roughly 2.5 seconds when using 12 threads provided that it can be parallelized completely. The actual value is 4.248 seconds (standard deviation: 168ms). Considering the fact that database overhead like the increased commit number and the system overhead of multithreading is induced, it can be said that the insert time scales almost perfectly with the number of threads and cores.

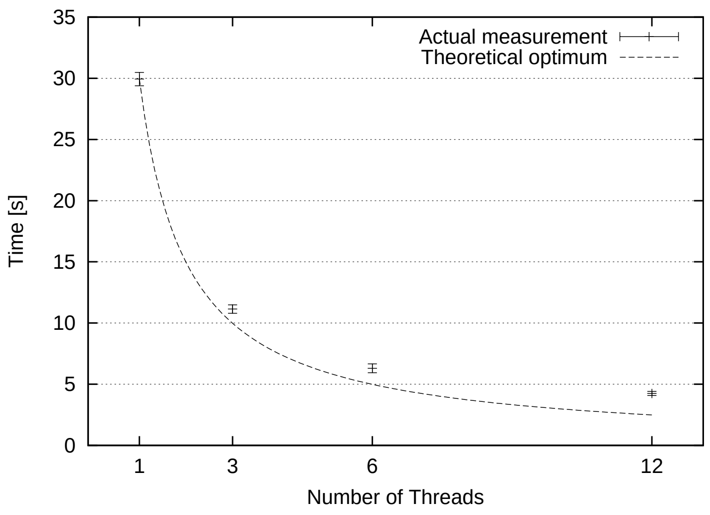

Fig. 8. Performance gain by parallel execution of INSERT-statements

4 Compression

Data volumes like the ones involved in a smart metering infrastructure entail the need for data compression. Since data of the same type is stored in continuous blocks, column stores are particularly suited for this task [5].

Heavyweight compression algorithms like Huffman coding [13], arithmetic coding [38] or the LZW algorithm [37] which give up compression and decompression speed to increase compression ratio typically result in performance losses. With insert performance beeing a crucial part of the depicted scenario, the focus of this paper is on lightweight column compression techniques. An overview over available techniques is given in the next section in order to make an evaluation in the context of energy event data possible.

4.1 Lightweight Column Compression Techniques

The lightweight column compression techniques which are presented in the following can be divided into three groups. Domain coding replaces values of any type by shorter ordinal references and is the underlying basis of all other methods. Common value suppression techniques like prefix coding and sparse coding and in a broader sense run-length encoding and cluster coding try to replace frequent values by shorter representations. Indirect coding introduces local dictionaries to reduce the width of references further.

Particularly domain and indirect coding would be pointless if full integer widths would have to be used to store values. For that reason, they rely on techniques to reduce value width. These include bit compression [32], variable byte coding [23] and patched frame-of-reference compression [39].

Domain Coding Domain coding or dictionary compression [19,4,30,18] is the fundamental compression algorithm which is utilized regardless of data types and structures and independently from other compression algorithms. All original values of a column are stored in a sorted dictionary and the column itself is represented as index vector consisting of ordinal references to this dictionary. According to [19], the memory footprint (in bits) of a domain coded column containing d distinct values of an arbitrary type and a total of t values can be calculated by formula 1.

$$t \cdot \lceil \log_2(d) \rceil \qquad (1)$$

The size of the dictionary itself has to be added to the total amount of required main memory, which carries weight especially when the record set contains a high percentage of distinct values.

Aside from the reduction of memory consumption, this method also entails an acceleration of processing speed due to the handling of smaller data volumes on a per request basis and the optimization of processing units for ordinal types on the hardware layer.

Prefix Coding Prefix coding [19,30,18] is powerful in such cases where a column contains one specific value very often (e.g. the NULL-value) and the table can be sorted by this column such that this value occurs at the beginning of this column. The prefix of equal values is then removed from the index vector completely. The prefix value and the number of occurrences are saved separately, each with a consumption of 32 bits. Lemke et. al. [19] have shown that the space requirements (in bits) of a prefix coded column with a total of t elements, d distinct values and a prefix of p elements thus can be calculated by formula 2.

$$(t - p) \cdot \lceil \log_2(d) \rceil + 64 \tag{2}$$

Sparse Coding If the occurences of the most frequent value are spread throughout a column, sparse coding [19,30,18] can be used to reduce the size of this column. In doing so, all occurences of the value are removed from the index vector and this so-called sparse value is saved once as reference into the dictionary, consuming 32 bits. A bit vector is generated for the column, indicating whether the value corresponding to the element at the given index equals the sparse value or not. In addition, a prefix coding is applied to this bit vector. According to [19], sparse compression reduces the size of a column with s occurences of the sparse value to the number of bits deduced from formula 3.

$$(t - s) \cdot \lceil \log_2(d) \rceil + (t - p) + 32 \tag{3}$$

To retrieve a value from a sparse encoded column, the bit vector has to be checked at the given index. If it indicates that the value differs from the sparse value, the index of the actual value within the index vector can be retrieved through the number of set bits up to the given index.

Cluster Coding When applying cluster coding [19,30,18], the index vector gets divided into equally sized blocks. All blocks which contain only one distinct value are then compressed by removing all but one occurence of the value within this block from the index vector. In addition, a bit vector is generated which indicates whether a block has been compressed or not.

Cluster coding is applicable in cases where a column contains only few distinct values which form blocks innately. For instance, a column containing the alternating values 1, 2 and 3 would be the worst case for cluster coding. To calculate the memory footprint (in bits) of a cluster coded column with a total of t and d distinct values, formula 4 has been deduced (where b is the block size and $b \mid t$).

$$\frac{t}{b} \cdot \lceil \log_2(d) \rceil + \frac{t}{b} \tag{4}$$

Indirect Coding Just like cluster coding, indirect coding [19,30,18] is based on a division of the index vector into blocks of equal size. However, a domain coding of these blocks takes place. Thus, every block can have its own dictionary as additional level of indirection between the actual value stored in the global dictionary and the bit vector pointing to this value from the index vector. Sharing dictionaries among subsequent blocks is possible as long as adding new values to the dictionary wouldn't increase the size of the bit vectors representing the keys. Furthermore, blocks with a high percentage of distinct values still can use the global dictionary directly.

Such beeing the case, indirect coding is particularly powerful in columns which contain blocks with few distinct values.

Run-Length Encoding Run-length encoding is a very simple lossless compression algorithm which unfolds its full potential in sorted columns.

According to the approach published by Golomb [9], sequences of a value (so-called "runs") are replaced by a single occurence of this value and the length of the original sequence. Due to read performance losses, this method is not applicable for column stores in its original form, as all preceding values of a given index would have to be touched in order to find the actual value.

For this reason, the technique used in column stores is slightly different [19,30,18]. To compress a column, all contiguous subsequent occurences of a value are removed from the index vector. Additionally, a second vector is generated. It contains the starting index of the index vectors corresponding entry.

Formula 5 has been deduced to calculate the size (in bits) of a *sorted* run-length compressed column containing a total of t and d distinct values.

$$d \cdot \lceil \log_2(d) \rceil + d \cdot \lceil \log_2(t) \rceil \tag{5}$$

Searching a given index becomes less complex compared to the original algorithm due to the modification mentioned above. For instance, it can be carried out in logarithmic time by binary search.

4.2 Compression of Energy Event Data

With due regard to the available compression techniques outlined in Sect. 4.1, the process of minimizing the space requirements of smart-meter readings comes down to three tasks. First, the number of distinct values has to be reduced as far as possible without losing relevant information in order to unfold the full potential of dictionary compression. Secondly, the spreading of values has to be analyzed in order to evaluate potential benefits of common value suppression techniques. In the third place, the readings have to be sorted in such a way that contiguous blocks of equal values which are as wide as possible occur. As mentioned in Sect. 2.3, the assumed model of meter readings consists of the three columns meter id, timestamp and value.

Reducing the Number of Distinct Values When investigating the necessary number of distinct values, the meter id column can be ruled out quickly. Obviously, there have to be exactly as many distinct ids as there are smart-meters known to the system. Furthermore, the specific characteristics aren't an issue due to dictionary compression, so the natural implementation as integer in the interval $[1, n]$ can be chosen, where n is the number of smart meters.

The situation is different with the timestamp column. In a first approach, timestamps exact to the second have been used, yielding $365 \cdot 24 \cdot 60 \cdot 60 = 31,536,000$ distinct values per year. But the relevant information for energy providers besides the consumption value is if there is a meter reading for a given time slot or not, and this information can be gained with less accurate timestamps. Even real-time analysis scenarios like the calculation of the current consumption of all customers can be carried out without the need of timestamps exact to the second. In this specific example, it would be sufficient to build the sum of all known timestamps of a given time slot.

Hence, it is acceptable to reduce the resolution of the saved timestamps to the one of the measurement interval. This means that the timestamps can be rounded down to 15 minute intervals. For example, 08:21:30 can be replaced by 08:15:00. One could as well save sequences instead of such timestamps, e.g. as ordinal reference to the quarter of an hour slot of a day, but since domain coding has this effect exactly, the additional logic doesn't have to be implemented manually. This strategy yields $365 \cdot 24 \cdot 4 = 35,040$ distinct values per year, which is a saving of 99.9% compared to the inital approach. In the abstract, the required amount of space could be reduced by truncating the timestamps. Table 1 shows that the last two digits of a timestamp are always 00, so stripping them would be a reversible operation. Through this procedure, 7 bits could be saved per timestamp. However, since every value is saved in the dictionary only once, this means savings of no more than roughly 30KB per year, which isn't profitable.

The consumption information in the value column can be saved in two ways. One possibility is to save counter readings. That means that every reading contains the total amount of consumed energy since the installation of the meter. The other possibility is to save consumption deltas. If the counter readings of two subsequent time slots are v_{t-1} and v_t, the value that would be saved for the time slot t would be $v_t - v_{t-1}$ in the case of consumption deltas. The two strategies are equivalent, as the sum of all deltas yields the counter reading, and the deltas can be calculated by subtracting subsequent counter readings. The question remains which approach requires less distinct values.

According to [1], power meters are measuring kilowatt hours and are calibrated to three decimal places (watt hours). The consulting company Capgemini estimates a lifespan of eight years for smart-metering devices [7]. That means that residential customers with an average consumption of 1000kWh per year would produce around $8 \cdot 1000 \cdot 1000 = 8,000,000$ distinct counter readings over the period of a meters lifetime.

In analysing the H0 profile, which contains average consumption deltas of residential customers in 15 minute intervals, it can be determined that all values are in the interval of $[0.001; 0.067]$ kWh. Since this profile is normalized to an annual consumption of 1000kWh, it can be compared directly to the calculation above. Even when taking huge deviations of the factor 100 or more into account, the number of distinct values is still only a fraction of those possible when saving counter readings. Therefore, it can be assumed that the number of distinct values can be reduced by 99.99% through the utilization of consumption deltas which thereby get the prefered option with regard to main memory consumption.

Common Values Since the meter id and timestamp column contain values wich are rather distributed uniformly, the only column that could potentially benefit of common value suppression techniques is the value column. When taking the H0 profile as a basis, it turns out that some values indeed occur ten times more often than others. However, there is no single value which would qualify as universal most common value. Furthermore, it can be doubted that this phenomenon also becomes manifest in real consumption data since the profile is normalized. Accordingly, the common value suppression techniques prefix coding and sparse coding aren't practicable for energy event data.

Finding the Optimal Ordering In order to achieve the highest possible compression, the table has to be sorted in a manner that continuous blocks of equal values are formed, preferebly in all columns. To evaluate the memory footprint of the meter readings table, the performance is measured as a function of the number of smart meters n, with every smart meter having a record set of one year in 15 minute intervals. In the following, the memory footprint S_{total} of a table is analyzed per column, so the total capacity requirements can be calculated by

$$S_{\text{total}}(n) = S_{\text{id}}(n) + S_{\text{datetime}}(n) + S_{\text{value}}(n) \tag{6}$$

which returns the minimum required number of bits for the chosen compression techniques. Thereby, the value column is assumed to contain at most 100 distinct values.

First of all, domain coding (cf. (1)) is applied to all columns. That means that there is a base compression on all columns which equals

$$S_{\text{id}}(n) = n \cdot 35040 \cdot \lceil \log_2(n) \rceil \tag{7}$$

$$S_{\text{datetime}}(n) = n \cdot 35040 \cdot \lceil \log_2(35040) \rceil \tag{8}$$

$$S_{\text{value}}(n) = n \cdot 35040 \cdot \lceil \log_2(100) \rceil \tag{9}$$

Nevertheless, the compression rate can be increased drastically by sorting and subsequent run-length encoding. In the abstract, all three columns qualify for a primary sorting. The goal is to form as few continuous blocks as possible. Sorting the table by the meter id yields n blocks with a length of 35040. If sorted by the timestamp column, the result are 35040 blocks with each having a lenght of n. In comparison, there are only 100 blocks with an average length of $\frac{n \cdot 35040}{100}$ when sorting the table by the value column. Hence, as run-length encoding gets more and more inefficient when the number of blocks increases, the value column is theoretically best suited for a run-length encoding.

However, one has to take into consideration that the domain coded timestamp column requires more space than the domain coded value column, since $\lceil \log_2(35040) \rceil > \lceil \log_2(100) \rceil$. In other words, the timestamp column consumes $9 \cdot n$ more bits. This also applies to the meter id column for $n > 100$. To solve the question if there are break-even points, the following auxiliary functions are used, where $\mathrm{D}(n)$ denotes the memory footprint of a domain coded column and $\mathrm{RLE}(n)$ the one of a run-length encoded column for n smart meters:

$$\mathrm{D}_{\mathrm{id}}(n) = n \cdot 35040 \cdot \lceil \log_2(n) \rceil \tag{10}$$

$$\mathrm{RLE}_{\mathrm{id}}(n) = n \cdot \lceil \log_2(n) \rceil + n \cdot \lceil \log_2(n \cdot 35040) \rceil \tag{11}$$

$$\mathrm{D}_{\mathrm{datetime}}(n) = n \cdot 35040 \cdot \lceil \log_2(35040) \rceil \tag{12}$$

$$\mathrm{RLE}_{\mathrm{datetime}}(\mathrm{n}) = 35040 \cdot \lceil \log_2(35040) \rceil + 35040 \cdot \lceil \log_2(n \cdot 35040) \rceil \tag{13}$$

$$\mathrm{D}_{\mathrm{value}}(\mathrm{n}) = n \cdot 35040 \cdot \lceil \log_2(100) \rceil \tag{14}$$

$$\mathrm{RLE}_{\mathrm{value}}(\mathrm{n}) = 100 \cdot \lceil \log_2(100) \rceil + 100 \cdot \lceil \log_2(n \cdot 35040) \rceil \tag{15}$$

The question is if there is a solution to the following inequation:

$$\mathrm{RLE}_{\mathrm{value}}(n) + \mathrm{D}_{\mathrm{datetime}}(n) \leq \mathrm{D}_{\mathrm{value}}(n) + \mathrm{RLE}_{\mathrm{datetime}}(n) \tag{16}$$

It turns out that the inequation is only true for $n \in [0, 128]$. For $n > 128$, sorting the table by the timestamp column has to be prefered. The same problem applies to the comparison of the meter id and timestamp column. Again, the question is which solution is true for

$$\mathrm{RLE}_{\mathrm{datetime}}(n) + \mathrm{D}_{\mathrm{id}}(n) \leq \mathrm{D}_{\mathrm{datetime}}(n) + \mathrm{RLE}_{\mathrm{id}}(n) \tag{17}$$

This time, there is a break-even point at $n = 35040$, or in other words the inequation is true for $n \in [35040, \infty[$. Hence, sorting by the timestamp column is preferable as soon as more than 35040 smart meters are in the database. Thus, function (8) can be replaced by

$$S_{\mathrm{datetime}}(n) = 35040 \cdot \lceil \log_2(35040) \rceil + 35040 \cdot \lceil \log_2(n \cdot 35040) \rceil \tag{18}$$

Now that the primary sorting of the table is by timestamp, no more compression techniques can be applied to the meter id column, since the number of distinct ids per timestamp block equals the length of this block. However, this doesn't apply to the value column, which could be sorted within the constraints of the primary sorting. Since there is a correlation between date and time and energy consumption, it can even be assumed that certain values accumulate at a specific date and time, which would qualify the column for a further run-length encoding. Since the column contains relatively few distinct values, applying an indirect coding and not sorting the column is conceivable as an alternative, too. As both scenarios can no longer be mapped reasonably as a function of the number of smart meters, the efficiency of both approaches has been evaluated experimentally.

In order to analyze the impact of both techniques, 10,000 profiles which conform to the conditions outlined in Sect. 4.3 have been inserted into the tables shown in Listings 4 and 5. The result is depicted in Table 2. The compression rate is based on the unmerged size of the column which is 321,482KB. With 1,912KB compared to 187,101KB, the run-length encoded column is almost 100 times smaller than the one compressed with indirect coding. Consequently, the former is the prefered option for further compressing the table.

Table 2. Compression rates and memory consumption of the value column with run-length encoding and indirect coding

Table	Compression Rate	Consumption [KB]
READINGS_RLE	99.6%	1,912
READINGS_INDIRECT	41.8%	187,101

```
CREATE INSERT ONLY COLUMN TABLE readings (meterid INTEGER,
    timestamp INTEGER, value INTEGER) WITH PARAMETERS ('
    COMPRESSION' = ('TIMESTAMP', 'RLE'), 'COMPRESSION' = ('
    VALUE', 'RLE'))
```

Listing 4. Schema READINGS_RLE

```
CREATE INSERT ONLY COLUMN TABLE readings (meterid INTEGER,
    timestamp INTEGER, value INTEGER) WITH PARAMETERS ('
    COMPRESSION' = ('TIMESTAMP', 'RLE'), 'COMPRESSION' = ('
    VALUE', 'INDIRECT'))
```

Listing 5. Schema READINGS_INDIRECT

In the worst case, the memory footprint of the value column hence can be calculated by

$$S_{\text{value}}(n) = 35040 \cdot 100 \cdot \lceil \log_2(100) \rceil + 35040 \cdot 100 \cdot \lceil \log_2(n \cdot 35040) \rceil \tag{19}$$

which is still better than domain coding alone for $n > 442$. The total amount of required memory thus can be estimated by

$$\begin{aligned} S_{\text{total}}(n) = n \cdot 35040 \cdot \lceil \log_2(n) \rceil + \\ + 35040 \cdot \lceil \log_2(35040) \rceil + 35040 \cdot \lceil \log_2(n \cdot 35040) \rceil + \\ + 35040 \cdot 100 \cdot \lceil \log_2(100) \rceil + 35040 \cdot 100 \cdot \lceil \log_2(n \cdot 35040) \rceil \end{aligned} \tag{20}$$

However, S_{total} doesn't cover system overhead like the row-id column (cf. Sect. 3.3). Therefore, the real consumption has been measured in Sect. 4.3 in order to give a more realistic estimation of the memory footprint of energy event data.

4.3 Estimation of Main Memory Consumption

To estimate the main memory consumption of large customer bases, profiles containing one year of readings have been generated with the assistance of the H0-profile. Each profile consists of 35040 readings, and the value r_v of each reading differs from the equivalent value v from the H0 profile with a uniformly distributed variance such that $0 \leq v - v \cdot 0,20 \leq r_{\mathrm{v}} \leq v + v \cdot 0,20$. The smart-meter ids have been chosen from $[1, n]$, where n is the number of generated profiles. However, the actual interval doesn't have a considerable effect on memory consumption due to the effects of dictionary compression. The year 2010 served as date range for the readings.

The profiles have been inserted and merged into the tables READINGS_IO and READINGS_RLE, so the measured memory footprint constitutes the size of the read-optimized main store. Since all rows have to be merged eventually to allow for real-time analyses, the consumption behaviour of the main store is the one that matters.

Estimation for the Table READINGS_IO Table 3 shows the memory footprint of the table READINGS_IO, which doesn't utilize any compression techniques besides domain coding. With the help of these measurements, the accuracy of the projection formula 1 for domain coded columns can be evaluated. The meter id column may server as an example. According to formula 1, it should require 584.80MB of main memory. The actual consumption differs from this value only by 30KB.

One can also see that the row id column, which is an implicit component of insert-only tables, is in no way optimized besides domain coding, since its footprint can be estimated very well by formula 1. The latter thus can be used to improve the projection of the total memory footprint of the table READINGS_IO for large n. Considering the already mentioned formulas 7, 8 and 9, the footprint projection for the table READINGS_IO can be calculated by formula 21.

$$\begin{aligned} S_{\mathrm{io}}(n) = n \cdot 35040 \cdot \lceil \log_2(n) \rceil + n \cdot 35040 \cdot \lceil \log_2(35040) \rceil + \\ + n \cdot 35040 \cdot \lceil \log_2(100) \rceil + n \cdot 35040 \cdot \lceil \log_2(n \cdot 35040) \rceil \end{aligned} \tag{21}$$

Estimation for the Table READINGS_RLE The memory footprint of the table READINGS_RLE, which applies a run-length encoding to the timestamp and value columns, is depicted in Table 4. When comparing the actual consumption values of the run-length compressed columns with the results from the formulas 13 and 19, it becomes evident that they have a much higher variance than the one for domain coding. For instance, the projected footprint of the timestamp column for $n = 10,000$ is around 190KB, a deviation of almost 60%. The projected consumption of the value column for the same number of profiles is around ten times higher than the actual one, but formula 19 maps the worst case anyway. It has to be assumed that these deviations are resulting from overheads of the run-length compression implementation. Considering the fact that these two columns only add up to a fraction of the total memory footprint of roughly 0.1% for $n = 10,000$, the approximation given by formulas 13 and 19 is still sufficient to project the footprint of the table READINGS_RLE for large n. Taking the already derivated formula 20 and the domain coding of the row id column into account, the footprint can be estimated with the help of formula 22.

$$S_{\text{rle}}(n) = S_{\text{total}} + n \cdot 35040 \cdot \lceil \log_2(n \cdot 35040) \rceil \tag{22}$$

Comparison to Storage on Disk In order to give a comparison of space requirements with disk-based storage, the same profiles which were inserted into the database were saved as comma separated values in plain text files (ASCII encoded, one byte per character). Since the timestamp has a fixed length and the values have an average of two digits, each line consists of a fixed part which adds up to 15 bytes, including separators and line break. The other part is the meter id, whose length varies. Assumed that the meter id lies in the interval mentioned above, the size (in bits) of a file containing one year profiles of n smart meters can be projected with formula 23.

$$S_{\text{csv}}(n) = n \cdot 35040 \cdot 15 \cdot 8 + \sum_{k=2}^{n+1} \lceil \log_{10}(k) \rceil \cdot 35040 \cdot 8 \tag{23}$$

A comparison of the three storage modes domain coded table, run-length compressed table and disk storage in CSV format is given in Table 5. While the values for 10,000 profiles are actual measurements, the others are projections with the help of the formulas 21, 22 and 23. They show that the footprint of 10,000,000 profiles can be reduced by roughly 65% compared to a storage on disk when making use of column compression potentials in in-memory databases.

Table 3. Memory footprint of table READINGS_IO for a number of smart meters n, broken down by column, in [MB]

	$n = 1$	10	100	1000	$10,000$
Meter ID	0.004	0.17	2.92	41.76	584.83
Timestamp	0.20	0.80	6.82	66.97	668.45
Value	0.03	0.29	2.92	29.24	292.40
Row ID	0.07	0.84	9.61	112.79	1290.84
Total	0.31	2.10	22.28	250.78	2836.53

Table 4. Memory footprint of table READINGS_RLE for a number of smart meters n, broken down by column, in [MB]

	$n = 1$	10	100	1000	$10,000$
Meter ID	0.004	0.17	2.92	41.76	584.83
Timestamp	0.27	0.28	0.29	0.30	0.32
Value	0.09	0.71	1.47	1.71	1.87
Row ID	0.07	0.84	9.61	112.78	1253.13
Total	0.43	2.00	14.30	156.58	1840.15

Table 5. Estimated space requirements of profiles containing one year of meter readings, on disk and in-memory

Profiles	Size on disk	READINGS_IO	READINGS_RLE
$10,000$	$\approx$ 6.2GB	$\approx$ 2.8GB	$\approx$ 1.8GB
$100,000$	$\approx$ 65GB	$\approx$ 30GB	$\approx$ 20GB
$1,000,000$	$\approx$ 682GB	$\approx$ 345GB	$\approx$ 230GB
$10,000,000$	$\approx$ 7.0TB	$\approx$ 3.8TB	$\approx$ 2.5TB

4.4 Impact on Insert Performance

Graefe and Shapiro have shown that data compression can speed up operations rahter related to analytical queries [10]. In contrast, it has to be assumed that compression does have a negative impact on inserts, or the merge process specifically. As compression isn't carried out on the write-optimized differential buffer, execution times of INSERT-statements aren't affected. However, a potentially expensive recompression has to be performed during each merge process.

In order to estimate the impact of the proposed compression strategy on the insert process, the execution time of the merge process has been measured for different base datasets. This base dataset, consisting of a number of profiles generated analogous to the ones in Sect. 4.3 is merged completely into the tables READINGS_IO and READINGS_RLE. Afterwards, another 100 profiles (3,504,000 rows) are inserted into the tables. Then, the time is measured it takes to merge the new records with the base dataset. The measurement was carried out on Host HPA and the database was configured to use at most four threads for the merge.

The results are depicted in Fig. 9. In contrast to the assumption, it shows clearly that a run-length encoding on the columns timestamp and value doesn't have a relevant impact on the merge process. One possible reason for this behaviour is that the compression overhead is compensated by the reduced data volume, which leads to a higher cache locality and reduced waiting time when reading data from main memory.

When merging with an empty table, the merge time averages 6033ms without run-length encoding (standard deviation: 54ms) and 6130ms (standard deviation: 72ms) when applying this additional compression. If the table already contains 1000 profiles (350,40,000 rows), the merge takes an average of 3 minutes 15.680 seconds (standard deviation: 5.089 seconds) for the table READINGS_IO and 3 minutes 10.890 seconds (standard deviation: 4.238 seconds) for the table READINGS_RLE.

5 Conclusion

By simulating the amount of meter readings that will arise when a smart metering infrastructure has been established in Germany it could be shown that this amount can be processed by an in-memory column store which utilizes a write-optimized differential buffer. The target insert rate of 112,000 meter readings per

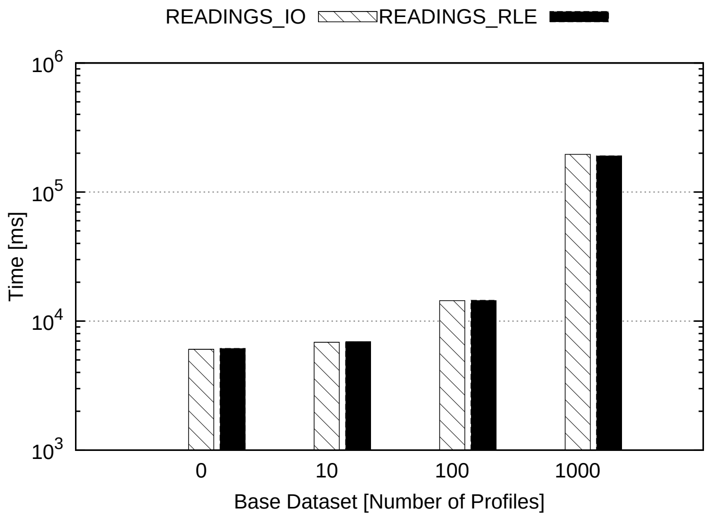

Fig. 9. Comparison of merge durations for different schemas and base datasets

second which corresponds to 100,000,000 smart meters could be outperformed significantly with the help of state-of-the-art hardware. The execution time of the insertion of 100,000 readings could be reduced to 170ms through the utilization of batch insert processing and insert-only tables, allowing for at least 500,000 inserts per second.

To achieve this insert rate, a trade-off between insert time and memory footprint had to be made. Insert-only tables can speed up the insertion by a factor of three compared to standard column tables since a natural primary key can be omitted. However, insert-only tables imply a row id column which can't be compressed further besides domain coding. This column makes up almost 70% of the total memory footprint of 1.8GB when storing 10,000 one year profiles. Thus, it might be preferable to give up the performance gain of insert-only tables if smaller customer bases have to be maintained.

Despite the increased footprint of insert-only tables, the amount of main memory which is required for the storage of meter readings could be reduced by 65% compared to a storage on disk by taking advantage of compression potentials in column stores. Lightweight column compression techniques were utilized solely to achieve this reduction. Since the compression overhead is compensated by the reduced data volume, the compression can be carried out without impairing the insert performance.

Even though not object of special attention, the merge process turned out to be the limiting factor. Although it can be carried out in linear dependency to the size of the dataset, the duration of the process would at some point outgrow the timespan until the size of the differential buffer induces the initiation of another merge. Thus, the implementation of data partitioning and aging strategies is inevitable.

In conclusion, it can be said that the foundation for the real-time analysis of energy event data could be laid by an in-memory database. New possibilities which aren't realizable up to date arise. Amongst others, real-time pricing based on supply and demand [31], short-term demand forecasting [24] and the real-time visualization of energy consumption for consumers [20] are feasible.

Appendix

Benchmark Environment

	Host HPA	**Host HPB**
CPU	2x Intel Xeon X5670	4x Intel Xeon X7560
Main Memory	144GB @ 800MHz	256GB @ 1333MHz
Operating System	openSUSE 11.2 2.6.31.14-0.8 (x64)	
Network Connectivity	82575EB Gigabit LAN	NX3031 10-Gigabit LAN
NewDB Version	1.50.00.327452 (dev)	

	Host HPC
CPU	Intel Core i5 750
Main Memory	8GB @ 1333MHz
Operating System	openSUSE 11.3 2.6.34.8-0.2 (x64)
Network Connectivity	82578DM Gigabit LAN

References

1. Directive 2004/22/EC of the European Parliament and of the Council of 31 March 2004 on measuring instruments. Official Journal of the European Union (March 2004)
2. Report on the Identification and Specification of Functional, Technical, Economical and General Requirements of Advanced Multi-Metering Infrastructure, Including Security Requirements. Tech. rep., The OPEN meter Consortium (July 2009)
3. Vorschlag für Richtlinie des Europäischen Parlaments und des Rates zur Energieeffizienz und zur Aufhebung der Richtlinien 2004/8/EG und 2006/32/EG. Europäische Kommission (2011)
4. Abadi, D., Madden, S., Ferreira, M.: Integrating Compression and Execution in Column-Oriented Database Systems. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data. pp. 671–682. SIGMOD '06, ACM, New York, NY, USA (2006)
5. Abadi, D.J.: Query Execution in Column-Oriented Database Systems. Ph.D. thesis, Cambridge, MA, USA (2008)
6. Aechtner, S.: Distributed Database Operations Related to Energy Data (2011)
7. Capgemini Consulting Österreich AG: Analyse der Kosten - Nutzen einer österreichweiten Smart Meter Einführung (January 2010), `http://oesterreichsenergie.at/Smart_Meter_Wunsch_und_Wirklichkeit.html?file=tl_files/DOWNLOADS/Pdf.%20Netze/Capgemini%20Kosten_Nutzenanalyse%20Smart%20Metering.pdf`, last checked 06/30/11
8. Energiewirtschaftsgesetz Deutschland: §21b Messeinrichtungen: Absatz 3a, `http://bundesrecht.juris.de/enwg_2005/__21b.html`, last checked 06/30/11
9. Golomb, S.W.: Run-Length Encodings. IEEE Transactions on Information Theory 12, 399–401 (September 1966)
10. Graefe, G., Shapiro, L.D.: Data Compression and Database Performance. In: In Proc. ACM/IEEE-CS Symp. On Applied Computing. pp. 22–27 (1991)
11. Grund, M., Krüger, J., Tinnefeld, C., Zeier, A.: Vertical Partitioning for Insert-Only Scenarios in Enterprise Applications. In: 16th International Conference on Industrial Engineering and Engineering Management (IE&EM), Beijing, China (2009)
12. Haerder, T., Reuter, A.: Principles of Transaction-Oriented Database Recovery. ACM Comput. Surv. 15, 287–317 (December 1983)
13. Huffman, D.A.: A Method for the Construction of Minimum-Redundancy Codes. Proceedings of the Institute of Radio Engineers 40(9), 1098–1101 (September 1952)
14. Knopfel, A., Grone, B., Tabeling, P.: Fundamental Modeling Concepts: Effective Communication of IT Systems. Wiley, 1 edn. (2006)
15. Krüger, J., Grund, M., Tinnefeld, C., Eckart, B., Zeier, A., Plattner, H.: Hauptspeicherdatenbanken für Unternehmensanwendungen - Datenmanagement für Unternehmensanwendungen im Kontext heutiger Anforderungen und Trends. Datenbank-Spektrum 10(3), 143–158 (2010)
16. Krüger, J., Grund, M., Tinnefeld, C., Plattner, H., Zeier, A., Faerber, F.: Optimizing Write Performance for Read Optimized Databases. In: Kitagawa, H., Ishikawa, Y., Li, Q., Watanabe, C. (eds.) DASFAA (2). Lecture Notes in Computer Science, vol. 5982, pp. 291–305. Springer (2010)
17. Krüger, J., Grund, M., Wust, J., Zeier, A., Plattner, H.: Merging Differential Updates in In-Memory Column Store. In: DBKDA (2011)

18. Lemke, C., Sattler, K.U., Faerber, F., Zeier, A.: Speeding Up Queries in Column Stores: a Case for Compression. In: Proceedings of the 12th International Conference on Data Warehousing and Knowledge Discovery. pp. 117–129. DaWaK'10, Springer-Verlag, Berlin, Heidelberg (2010)
19. Lemke, C., Sattler, K.U., Färber, F.: Kompressionstechniken für spaltenorientierte BI-Accelerator-Lösungen. In: Freytag, J.C., Ruf, T., Lehner, W., Vossen, G. (eds.) BTW. LNI, vol. 144, pp. 486–497. GI (2009)
20. Licker, R.: Visualization Methods of Real-Time Energy Data (2011)
21. McLaughlin, S., Podkuiko, D., McDaniel, P.: Energy Theft in the Advanced Metering Infrastructure. In: Proceedings of the 4th International Conference on Critical Information Infrastructures Security. pp. 176–187. CRITIS'09, Springer-Verlag, Berlin, Heidelberg (2010)
22. Microsoft: MSDN. Binding Arrays of Parameters, `http://msdn.microsoft.com/en-us/library/ms709287%28v=vs.85%29.aspx`, last checked 06/30/11
23. Silva de Moura, E., Navarro, G., Ziviani, N., Baeza-Yates, R.: Fast and Flexible Word Searching on Compressed Text. ACM Trans. Inf. Syst. 18, 113–139 (April 2000)
24. Neuhaus, J.: Vorhersagemodelle und Methoden in der Energiewirtschaft (2011)
25. Oracle: Java Platform Standard Ed. 6, Class Calendar, `http://download.oracle.com/javase/6/docs/api/java/util/Calendar.html#getTimeInMillis%28%29`, last checked 06/30/11
26. Oracle: Java Platform Standard Ed. 6, Interface java.sql.Statement, `http://download.oracle.com/javase/6/docs/api/index.html?java/sql/Statement.html`, last checked 06/30/11
27. Pade, S.: Optimizing Database Queries in Main Memory Databases for the Energy Industry (2011)
28. Petrlic, R.: A privacy-preserving Concept for Smart Grids. In: Sicherheit in vernetzten Systemen: 18. DFN Workshop. pp. B1–B14. Books on Demand GmbH (2010)
29. Plattner, H.: A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. In: Çetintemel, U., Zdonik, S.B., Kossmann, D., Tatbul, N. (eds.) SIGMOD Conference. pp. 1–2. ACM (2009)
30. Plattner, H., Zeier, A.: In-Memory Data Management - An Inflection Point for Enterprise Applications. Springer (2011)
31. Richter, O.: Modeling and Definition of Energy Rates for In-Memory Databases (2011)
32. Sanders, P., Transier, F.: Intersection in Integer Inverted Indices. In: ALENEX'07 (2007)
33. Schapranow, M.P., Kühne, R., Zeier, A., Plattner, H.: Enabling Real-Time Charging for Smart Grid Infrastructures Using In-Memory Databases. In: LCN. pp. 1040–1045. IEEE (2010)
34. Statistisches Bundesamt Deutschland: Haushalte, `http://www.destatis.de/jetspeed/portal/cms/Sites/destatis/Internet/DE/Content/Statistiken/Bevoelkerung/HaushalteFamilien/Aktuell,templateId=renderPrint.psml`, last checked 06/30/11
35. Statistisches Bundesamt Deutschland: Unternehmen und Betriebe im Unternehmensregister, `http://www.destatis.de/jetspeed/portal/cms/Sites/destatis/Internet/DE/Navigation/Statistiken/UnternehmenGewerbeInsolvenzen/Unternehmensregister/Unternehmensregister.psml`, last checked 06/30/11

36. Stonebraker, M., Abadi, D.J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., Zdonik, S.: C-Store: a Column-Oriented DBMS. In: Proceedings of the 31st International Conference on Very Large Data Bases. pp. 553–564. VLDB '05, VLDB Endowment (2005)
37. Welch, T.A.: A Technique for High-Performance Data Compression. Computer 17, 8–19 (June 1984)
38. Witten, I.H., Neal, R.M., Cleary, J.G.: Arithmetic Coding for Data Compression. Commun. ACM 30, 520–540 (June 1987)
39. Zukowski, M., Heman, S., Nes, N., Boncz, P.: Super-Scalar RAM-CPU Cache Compression. International Conference on Data Engineering p. 59 (2006)

Eigenständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die genannten Quellen und Hilfsmittel verwendet habe.

Potsdam, den 30. Juni 2011

Leonhard Schweizer