
**Open Access**

*Open Journal of Big Data (OJBD)*
*Volume 5, Issue 1, 2019*

http://www.ronpub.com/ojbd
**ISSN 2365-029X**

# Sparse and Dense Linear Algebra for Machine Learning on Parallel-RDBMS Using SQL

Dennis Marten[A], Holger Meyer[A], Daniel Dietrich[B], Andreas Heuer[A]

[A,B]Institute of Computer Science, Rostock University, Albert-Einstein-Straße 22, 18059 Rostock, Germany,
[A]{dm, hme, heuer}@informatik.uni-rostock.de, [B]Daniel.Dietrich@uni-rostock.de

## Abstract

*While computational modelling gets more complex and more accurate, its calculation costs have been increasing alike. However, working on big data environments usually involves several steps of massive unfiltered data transmission. In this paper, we continue our work on the PArADISE framework, which enables privacy aware distributed computation of big data scenarios, and present a study on how linear algebra operations can be calculated over parallel relational database systems using SQL. We investigate the ways to improve the computation performance of algebra operations over relational databases and show how using database techniques impacts the computation performance like the use of indexes, choice of schema, query formulation and others. We study the dense and sparse problems of linear algebra over relational databases and show that especially sparse problems can be efficiently computed using SQL. Furthermore, we present a simple but universal technique to improve intra-operator parallelism for linear algebra operations in order to support the parallel computation of big data.*

## Type of Paper and Keywords

Regular Research Paper: *linear algebra, sparse problem, dense problem, intra-operator parallelism, query performance, machine learning, scientific computation, relational databases, SQL, PArADISE*

## 1 Introduction

With increasing capabilities in gathering data, calculating scientific models have become more accurate, but also very hardware and time demanding. Since time is a crucial aspect, especially for real-time applications, modern research has been investigating parallel computation possibilities to speed up the calculation process over the past decades. Especially the areas of Big Data [6] and Machine Learning [16] have been thriving and are of interest for scientists and members of industry.

One of the most popular developments of scalable parallel computation is the MapReduce framework [4], which enables developers to transparently compute algorithms in a scalable manner. Many different research projects have been built upon this framework, mainly adding APIs (for instance Apache Spark [40]) or optimization techniques (like Apache Flink [7]) into MapReduce. This development has been a point of discussion since then, as parallel database systems have been developed long before the MapReduce framework, while having similar application areas. For example, Stonebraker et. al. analyzed in [37]

several scenarios in which parallel database systems outperformed MapReduce, even in typical MapReduce-applications. This circumstance is especially interesting, since using parallel database systems comes with many additional benefits, for instance concurrency control or data management functionality.

As a consequence of these findings, our research focuses on calculating machine learning and scientific computations on parallel database systems using SQL, while ensuring data privacy through query manipulation. As we have extensively described in [29–31] we are translating operators that are substantial elements of machine learning and scientific computation algorithms into SQL. In order to fully use the potential of database optimization, we create query plans for successive operations. One of the main benefits of this approach is the possibility to extend the architecture, as depicted in Figure 1, with back-end SQL interfaces to gain additional functionality. As already mentioned, we currently focus on manipulating the machine learning queries to achieve privacy protection [15] of sensitive sensor data.

This paper aims on deepening our previous analysis on how to efficiently calculate matrix operations on parallel relational database systems via SQL [29–31]. Therefore, we provide a brief description of our research project and give an overview on frequently used groups of methods which are essential for machine learning and computational modeling. Currently, we have picked a few of these methods per group to translate them efficiently into SQL. These groups are mostly split into two sub-categories, dealing with *dense* and *sparse* matrices. This sub-division is crucial to determine which kinds of problems can be calculated efficiently on parallel database systems in comparison to highly-tuned linear algebra libraries like LAPACK [2], ARPACK [25] or similar.

Our main focus in this article is to analyze important aspects of efficient linear algebra query calculation on parallel databases. The contributions of this article include:

- A detailed discussion on the choice of meaningful database schemas in Section 2;
- A categorization of suitable and unsuitable operations for database processing in Subsection 4.2;
- A discussion on how to treat highly iterative methods via nested queries in Subsection 4.3;
- An experimental study on the impact of indexes (GIN, B-Tree, Hash) for basic linear algebra operations in Subsection 4.4. The experimental study shows that indexes have only a positive effect on a small set of sparse basic operations, but can greatly improve highly selective Machine Learning algorithms;
- A discussion on why classical database partitioning strategies are inferior to the block-wise matrix partitioning on dense problems in Subsubsection 5.2.2, but are well suited on sparse problems in Subsection 5.3;
- The development of a simple, universal query decomposition technique that allows intra-operator parallelism for fundamental linear algebra operations in Subsection 5.4. The query decomposition technique is experimentally evaluated in Subsection 5.5. The evaluation provides extensive experimental evidences that the technique speedups the computation of queries in comparison to plain query evaluation plan.

Lastly, Section 6 concludes and summaries this work and provides some insights for future research projects.

## 2 State of the Art

Since this paper is mostly focused on the topic of database implementation aspects and algorithmic enhancements, we refer to [29] for a detailed state of the art analysis regarding existing connections of machine learning, database techniques and big data frameworks.

As we mentioned in the introduction, the starting point of our research can be found in [37], which provides an experimental evaluation of parallel database systems versus MapReduce implementations. In this paper, it has been shown that parallel database systems can outperform MapReduce frameworks by orders of magnitude in several scenarios. However, it has been mentioned that tuning the parallel database systems to its best possible performance can be hard and tedious.

Regarding the use of SQL, to our knowledge, there have only been few research projects that investigated performing scientific computation via SQL. One of them is RIOT-DB from Duke University [42], which maps R variables to the database via new data-types. This allows transparent database support for the user. The researchers stated that they tried to calculate matrix multiplications directly on the database via SQL, but did not follow up on this method, since their approach gained poor performance results. Since their presentation of this aspect was rather short, it is not possible to understand the reason of their performance problems. One possible answer might be the use of the row-store MySQL as the back end of their framework, which in general is not known for high performance.

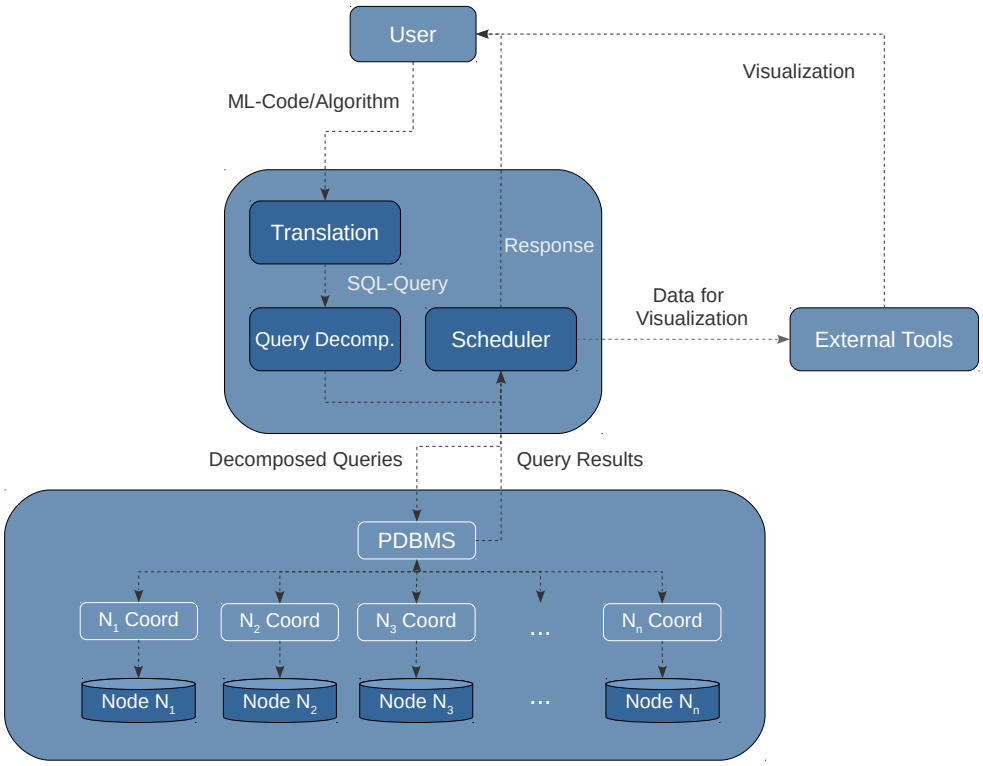



**Figure 1: Architecture for scientific computations and machine learning processing on parallel databases**

Another project has been investigating the calculation of a principal component analysis (pca) via SQL using SQL Server [32]. In their work, they tried to compute the pca using singular value decomposition in SQL via user defined functions. They compared this approach to a version implemented in Java, which used little SQL and database support. In this case the latter version was superior. The main reasons for poor performance might be the excessive use of non-optimized data manipulation statements and the low data size, which allowed in main memory calculations. Furthermore, there has been no information stated, how or if they tuned their database system. This is a crucial aspect, since most systems are working for multiple users with low-cost queries, in comparison to analytical work that requires as much memory and cache as possible for few users.

Other research projects have enabled scientific calculations by rewriting internals of existing database systems. For instance, researches have shown in [27,28], that internal rewriting of database types and optimization structures can have great performance enhancing effects on calculating dense matrix multiplications. Similarly to this, the MADlib project [19] does also provide internally written data types and also implemented commonly used methods for machine learning. Anyhow, both projects need data to be fully allocated in main memory and are not compatible with ANSI-SQL syntax as they rely on new implementations.

At this point we would like to give a very concise overview on some of the most noteworthy recent projects that are combining techniques from the area of database technologies and machine learning. One of the major research directions that has been taken is using machine learning to improve database technology. For instance, it has been shown in [24] that it can be very beneficial to use machine learning in real-world applications to learn new kinds of indexes that consider patterns of structured data for a given use-case, rather than traditional index structures which are designed to fit for more general applications. Furthermore, the OtterTune Service [41] uses Gaussian Process Regression models to find optimal configurations of database systems. This is a project with high potential as tuning databases with big workloads is very demanding and is usually done by paid experts, due to its complexity.

Other current research in this area focuses on lowering the costs of machine learning workflows by optimizing data handling and resources. For instance, Ease.ml [26] provides a declarative service platform where users only send a high-level schema of their desired application.

The system will then transparently choose models, type of shared storages and allocates computations resources in a meaningful way considering the overall workload. Other projects like SketchML [23] combine advanced compression techniques and statistics in order to decrease communication costs for distributed machine learning algorithms. Similarly, SystemML [3] provides fused plans for linear algebra operations in order to lower the necessity of scans and saving intermediates. Therefore the researchers have investigated several techniques for candidate exploration, selection of fusion plans and code generation of local and distributed operations over dense, sparse and compressed data.

Another promising project in this area is called MISTIQUE [39], which does lower the storage footprint of machine learning model diagnosis. Here, researches have introduced a number of methods like quantization, deduplication or summarization to lower the costs of saving and querying intermediates when learning or evaluating models. Finally, [17] introduces a speed up of analytical machine learning queries on typical data science frameworks by introducing a materialization of machine learning plans. Therefore, the computed models are stored with additional metadata, which allows to merge several combinable models later on and reusing these to get faster approximative results in comparison to learning a new model from the data.

## 3 Computational Environment

This section gives a brief overview of concepts and the historical development of our machine learning and scientific computation's environment on parallel databases. We will start with a short overview of our PArADISE project (more detailed descriptions of PArADISE are given in Section 3 of [29] and in [14]).

### 3.1 The PArADISE Project

As one of our current research projects, we are developing the PArADISE[1] framework. This framework aims at supporting developers of assistive systems in three development phases. In Figure 2, these phases are shown as Development (left-hand side), Data and Dimension Reduction (depicted by the arrow in the middle), and Usage (right-hand side):

- **Development:** Developers of Machine Learning (ML) and scientists of data are trying to detect and predict user activities, using data from a small amount of test persons, collecting sensor data for a short time period (maybe some weeks), annotating these sensor data with activity information, and then trying to learn the activity models by means of ML algorithms.

- **Data and dimension reduction:** In the development phase, there is a small amount of probands, but a large amount of sensors and a high frequency of the sensor data. After having derived the activity and intention models, one has to reduce the dimensions of the data (e.g., the number of sensors being evaluated) and the data itself (e.g., measuring and transmitting sensor data every minute instead of every milisecond).

- **Usage:** When using the assistive system afterwards for a huge number of clients (millions of clients having billions of sensors) with the reduced set of sensor data, one has to decompose the SQL queries detecting the activities and intentions of the users. This query decomposition aims at better performance because the query will be vertically pushed down to the sources of the data (the sensors) as close as possible. Even more importantly, the decomposition of the query results in better privacy protection for the user of the assistive systems, since most of the original sensor data has not to leave his personal equipment, his apartment, or his car. Only a *remainder query*, the part of the query that cannot be pushed down to the clients and sensors, has to be evaluated on the large cluster computers of the provider of the assistive system.

In this context, it is assumed that the provider of the globally distributed system is called *Poodle* [13]. Poodle uses ML development tools such as R or higher-level languages to derive the activities and intentions of the user. This ML code will then be transformed to a sequence of SQL statements. These SQL statements will then be evaluated in parallel on a large computer cluster, the parallelization will be introduced by the PArADISE system. This phase is called ML2PSQL in Figure 2.

To be able to automatically decide about the privacy-oriented decomposition of the queries, we have to use SQL queries as a basis for query containment and Anwering-Queries-using-Views techniques. Hence, it is crucial for this approach to be able to express ML code by a sequence of SQL queries in the development phase of the system. Only then, one can use the privacy-by-design principle when constructing the evaluation algorithms in the usage phase.

In the next section, the components of the ML2PSQL transformer will be discussed. We call this part of the PArADISE project PaMeLA[2].

[1] Privacy-aware assistive distributed information system environment

[2] PArallelization of MachinE Learning Algorithms and Linear Algebra

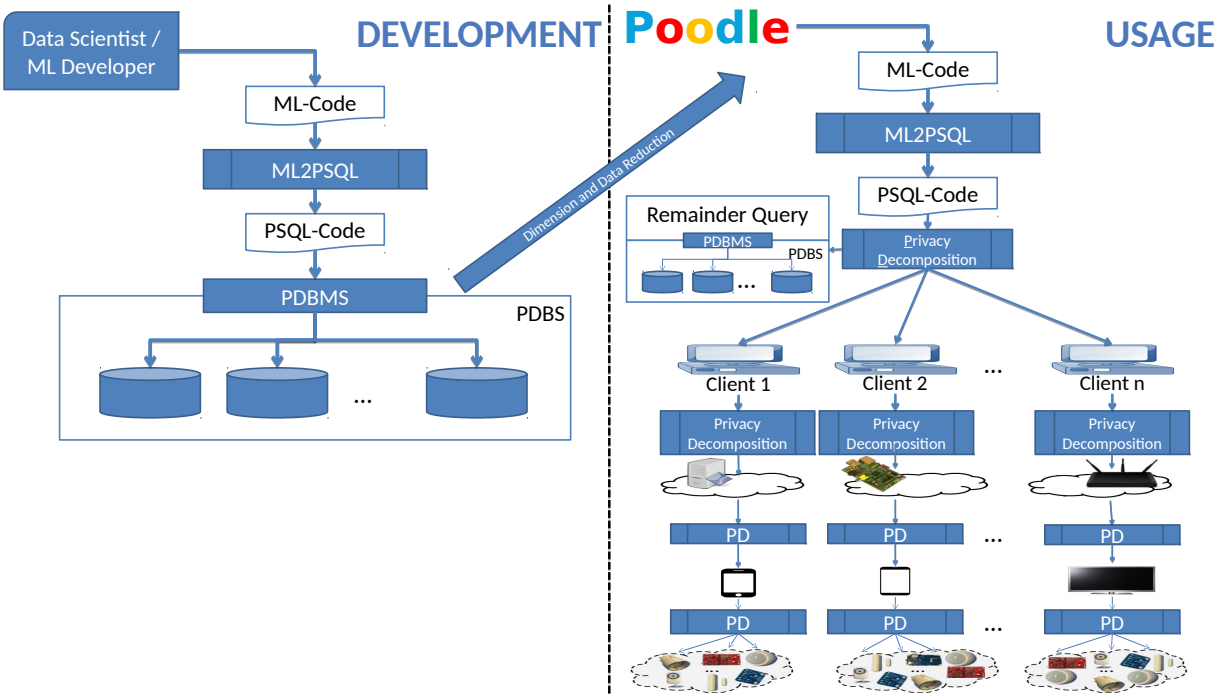



**Figure 2: The PArADISE environment [29]**

### 3.2 Historical Development in PaMeLA

The original goal of our PaMeLA project is to support developers of assistive systems in calculating models using huge sets of sensor data. In order to help them handling big data projects, we hypothesized that parallel database systems are a reasonable choice for most of their requirements. A possible architecture for our proposed framework is depicted in Figure 1. As can be seen, the main addition in this environment is the translation interface, which translates machine learning algorithms or general algorithms of scientific computations into SQL statements and decomposes these queries into sub-statements that can be calculated more efficiently by the database system. Furthermore the interface filters non-translatable operations (like visualizations), so that these operations can be processed afterwards by external software.

One of the main advantages of this approach is that the aspect of data privacy can be implemented reasonably well (see above, more details are given in [14, 15]). The manipulation is based on SQL queries, since SQL's core is and has been widely supported by nearly every common relational database system, making it a great choice for long term implementations, which are (nearly) independent from the used system. On the other hand, the core of SQL (an extended relational algebra) is deeply and theoretically understood so that we are able to prove desirable properties of the rewritings.

Due to the goal of full SQL support, it is important to note that we are not interested in internal reprogramming of common systems, rather than maximizing the use of established database techniques. Aside the aspect of universal long-term usage, conceptually, sensor data as well as vectors and matrices are usually strictly structured and can therefore be naturally modeled using relational databases.

Anyhow, the fundamental requirement for our proposed approach is to have meaningful queries that express computational modeling or machine learning algorithms. Since matrix and vector operations are not part of the SQL standard we needed to investigate several different aspects on this topic, mainly leading to questions like:

- Translation:
  - What are frequently used basic sub-methods in machine learning and scientific computations?
  - How can these sub-methods be translated?
  - How can these translated queries be split into sub-queries for efficient parallel computation?
- Performance:
  - What kind of problems or methods can be processed reasonably fast or slow ?

  - When can indexes be usefully applied?
  - How should data be partitioned or replicated?
  - What are other performance enhancing or degrading aspects?

Furthermore, we realized that common provenance management techniques are not sufficient to be used by developers of assistive systems, raising additional questions, like

- What sensors have the most impact on modeling results?

With these questions as a starting point, we started our investigation by comparing several different widely known non-parallel relational database systems on calculating linear regression [30, 31]. We have found that one can compute linear regression parameters for a single feature using column stores (MonetDB) as fast as using the statistical software R (using LAPACK). Furthermore, systems with advanced optimization like IBM's DB2 can come close to these computation times, although using a row-storage scheme. These results were somewhat surprising, but confirmed the power of internal optimization potential, since row-stores usually do perform significantly worse in comparison to column-stores on this kind of operations.

We continued our research in [29] showing that translating Hidden Markov models into SQL statements is possible without any loss of information. Furthermore, we have presented that methods like training model parameters (in this case transition matrices) can be calculated faster than in R even using in-memory data sizes. This can be explained since scanning, sorting and grouping are highly tuned methods in database systems. Furthermore, we showed that standard filter techniques (forward filtering) can be calculated with a reasonable performance gap in comparison to in-memory software. Database techniques scale according to needed floating point operations not only in-memory but also for big data on disks.

### 3.3 Frequently Used Sub-Methods and Operations

While analyzing internal computation of several different machine learning algorithms (such as linear/ logistic regression, Hidden Markov models, selection/ extraction operations and support vector machines), we have found sub-methods listed in Table 1 to be crucial for these operators.

Most of the representatives are chosen to work well on huge sparse problems. On the contrary, some of them perform poorly on dense problems as will be discussed in Section 4. All of these sub-methods have in common that they are basically composition of fundamental operators of the euclidean vector space. These include field operations, which are element-wise addition, subtraction, multiplication and division, and matrix-vector- and matrix-matrix-multiplication, which can be seen in Table 2.

Additionally, scalar functions like the square root $\sqrt{x}$ or trigonometrical functions like the exponential function $e^x$ or the logarithm $\log x$ are frequently used in statistics. Examples can be found when working with probability distributions or minimizing log-likelihoods of models and data.

With these sets of scalar and fundamental operators we have built the presented sub-methods. We are currently optimizing them according to the techniques presented in the following Section 4 and Section 5.

## 4 Query Formulation and Impact of Indexes

After the short résumé of our research project, this section presents and discusses the general performance influences. The main goal is to get as much performance as possible from the database system without internal rewriting. Therefore, general aspects like query formulation or choice of schema are investigated, although here the main focus is on the use of indexes. The following Section 5 will then continue to analyze the performance influencing aspects for parallel database systems.

The most natural aspect of performance influence is the amount of system resources the database system has access to. One has to distinguish two types of systems: the systems that can "take" resources as they need (for instance MonetDB) and the systems that allocate a predefined static amount of resources. The first type usually does have a practical advantage for a single user in comparison to the second type, as the systems get exactly the type and amount of resources as needed. Anyhow, the latter approach does ensure fair resource allocation for multiple users.

However, it is extremely difficult to tune such a static system for one user, since the resource demand of queries can differ substantially. In general, it has to be suspected that common settings for database systems do not fit with requirements for analysis, since it is unlikely that a high amount of users will work on the same data at the same time. For our setup we have chosen parameters according to community and developers' suggestions for general performance enhancement relative to the amount of main memory, type of secondary storage and the amount of expected simultaneously working users. These configurations will be addressed further in Section 5.5.

**Table 1: Sub-methods in machine learning algorithms and their representatives, which have been implemented in our research project.**

| Sub-Method | Representative |
|---|---|
| Matrix Factorization | $QR$ Factorization |
| Matrix Similarity Transformation | Bi- and Tridiagonalization |
| Eigenvalue / SVD analysis | $QR$ Method, Power Method |
| Feature Selection/ Extraction | Principal Component Analysis |
| Linear System Solver | Conjugated Gradient Method |
| Numerical Differentiation | Jacobi Matrix |
| Optimization Solver | Gradient Descent Method |
| Repartitioning Methods | Spectral Partitioning, Reverse Cuthill McKee |
| Fourier Transformation | to be done |

**Table 2: Fundamental operators used for the sub-methods in Table 1. Here, $A$ and $B$ represents matrices and $v, w$ represent vectors with corresponding dimension.**

| Operator | Representative |
|---|---|
| Field Operations | $+, -, \cdot, /$ |
| Transposition | $A^T$, $v^T$ |
| Matrix Multiplication | $AB$,$Av$, $v^TAw$, ... |

These sections provide several insights on performance analysis and improvement possibilities. All of the presented results have been calculated with PostgreSQL 10.1 on a computer with hardware specifications described in Table 3 and parameter reconfiguration according to Table 4. It is noteworthy that tuning PostgreSQL is crucial at this point, since the default configuration of the system is a "basic configuration tuned for wide compatibility rather than performance" [12]. Therefore, we have changed parameters locally and on the cluster-side according to the suggestions in [12]. Although substantially slowing down the processing time, all of the following calculations have been done without parallel processing to ensure continuous behavior when scaling problem sizes.

## 4.1 Choice of Schema

Another fundamental aspect of reasonable performance besides systems resources is the "correct" or reasonable use of relation schemas, as this choice will heavily influence query formulation& calculation, the usage and impact of indexes and others. As established in former publications [29–31], a reasonable choice for a matrix schema is

```
A( i int, j int , v double),
```

since it allows queries to address any element or kind of sub-matrices, including rows, columns or (secondary) diagonals. Furthermore, zero values do not have to be explicitly stored and different indexes can be created and used well on this schema as described in the upcoming Section 4.4. The question whether explicit declaration of the primary key `(i,j)` is useful is questionable and heavily dependent on the way the matrix is intended to be used. Usually, using primary keys comes along with building indexes (e.g. B-trees) in order to ensure efficient constraint checking. Anyhow, as will be seen in Section 4.4, using indexes is not always preferable, especially when using queries with a lot of update operations of intermediate relations and additional non-selective query operations like full matrix vector multiplications or similar operations.

Furthermore, there are many scenarios where temporary tables hold multiple values for any pair $(i, j)$ for later aggregation. This mainly happens when calculating sub-results on parallel or distributed databases. On the other hand, besides ensuring consistent data, there are some useful operations that explicitly require unique constraints such as primary keys. An example would be the MERGE operation (SQL:2003 [21] and SQL:2008 [22], also known as *upsert*), which can be beneficially used when dealing with field operations like $A = A + B$ or similar. While offered in most systems, it might be noted that MERGE is rarely found to be implemented in its standardized syntax, which unfortunately hurts the portability of this operation.

Apart from the primary key aspect, the schema performs poorly on dense problems (especially on row-stores), but better on sparse problems as will be described in Section 4.2. On the contrary, the good selectivity of elements and subgroups provides a wide range of SQL functionality.

In our research, we also tested some other schemas as

**Table 3: Hardware configuration for local experiments**

| Component | Value |
|---|---|
| Processor | Intel Core i7-4600U CPU @ 2.10 GHz × 4 |
| L1 Cache | 32 KB |
| L2 Cache | 256 KB |
| L3 Cache | 4 MB |
| RAM | 12 GB |
| Operating System | Ubuntu 17.10 (64-bit) |
| Secondary Storage | 250 GB |

**Table 4: Non-default parameters for PostgreSQL 10.1 used for local experiments**

| Parameter | Value |
|---|---|
| shared_buffers | 3GB |
| temp_buffers | 64MB |
| effective_cache_size | 9GB |
| work_mem | 1536MB |
| maintenance_work_mem | 1535MB |
| min_wal_size | 4GB |
| max_wal_size | 8GB |
| checkpoint_completion_target | 0.9 |
| wal_buffers | 16MB |
| default_statistics_target | 500 |
| random_page_cost | 4 |

for instance

```
A(i int,
  c1 double,
  c2 double,
  ⋮
  cn double)
```

which represents a matrix with $n$ columns. Additionally, one can interpret this schema as a materialized joined table of $n$ vectors using the prior schema. Materialized joins of vectors are rare to be found but are often reasonable when possible. We initially tried this approach in column stores on computing basic linear regression parameters which mainly consist of several dot products. While the performance results have been convincing, the lack of flexibility when dealing with matrices is a major drawback. Even fundamental operations like transposing do lead to enormously long SQL statements. Furthermore, most of the database systems do not even support huge amount of attributes in relations, so that big matrices could not even be stored in one relation.

Another schema we have tested used arrays for the row or column entries in order to increase the cache hit rate of elements for fundamental operations. The corresponding schema is

```
A(i int, v double array [n])
AT(i int, v double array [n])
```

where the transpose $A^T$ of a matrix should be stored in `AT` as well to grant fast column access. This approach was meant to increase cache hits and improve the way that multiplications are calculated. For example, if one considers $C = A \times B$, the multiplication should successively load row vectors $A_{i,:}$ and multiplicate them with $B$ in order to obtain the desired rows $C_{i,:}$ until $C$ is fully calculated. However, PostgreSQL or even SQL does not directly support element-wise field operations nor summation over arrays. Therefore, we had to implement these operations using Postgres' user-defined-functions, which is unfortunate, since our goal is to fully rely on the SQL core. In addition to these implementation downsides, we have encountered bad performance results as well. The main reason for this is that Postgres does unnest any array before it calculates summations or aggregations. Therefore, this approach not only needs to process the algebraic operations but also has to process a considerable overhead due to internal data management, and is therefore not a meaningful choice for this use case.

Ultimately, it can be stated that it is reasonable to choose the schemas

```
A( i int, j int, v double )
w( i int, v double ),
```
(1)

with row number `i`, column number `j` and the corresponding value `v` for scientific computation as they model matrices and vectors well and also provide a wide range of SQL functionality.

## 4.2 Dense and Sparse Problems

When examining the established schemas in (1), one can already suspect that they do not behave equally "good" for dense and sparse problems. As already

mentioned, this can be confirmed, especially on row-stores, where dense problems do perform significantly worse in comparison to standard cache-hit optimized libraries like LAPACK. In the analysis of the state of the art analysis in Section 2, this problem was already addressed in [19, 27, 28], where new in-memory matrix and vector data types have been internally implemented with great success.

Here, we would like to provide some simple analysis on matrix multiplication in order to understand what kind of multiplication should be avoided. Therefore, consider a query plan for the product $A \times B$ of random matrices $A$ and $B$ with fitting dimensions. The corresponding SQL statement can be written as

**SQL statement 1:**

```
select A.i,B.j,sum(A.v*B.v)
from A join B on A.j=B.i
group by A.i,B.j.
```

The calculation usually consists of 2 sequential scans, 1 hash join and 1 hash aggregate. Depending on the dimensions of $A$ and $B$ the relative amount of time for each operation is varying a few percent. Usually, scanning tables rarely exceed few percentage of the overall calculation time, while hash joining costs around 30 % to 40 % of the time, leaving 60 % to 70 % for the aggregation itself. Firstly, this reveals that there is limited potential for speeding up these dense multiplication queries using indexes, which will be discussed later in Section 4.4. Secondly, many database systems can not efficiently calculate the aggregation, as they fully process each aggregated tuple one by one, without acknowledging the potential of reusing columns for multiple calculations. This behaviour does reflect typical implementations of query evaluation engines, which usually follow the Volcano iterator model [11].

This can be confirmed, when investigating simple matrix multiplication

$$AB = \left( \sum_{l=1}^{k} a_{il} b_{lj} \right)_{ij} \qquad \begin{matrix} i \in \{1, \dots, n\} \\ j \in \{1, \dots, m\} \end{matrix}$$

of matrices $A \in \mathbb{R}^{n \times k}$, $B \in \mathbb{R}^{k \times m}$ resulting in

$$\tilde{f}(k, m, n) = nm(2k - 1)$$

floating point operations (flops). The resulting matrix has $nm$ entries, needing $k$ multiplications and $k - 1$ summations each. From the database point of view the value $nm$ does represent the number of hash buckets and $k$ represents the number of entries in each of the buckets. In order to test whether database systems are scaling well with the number of flops and the influence of the number of hash buckets and their respective sizes, two matrices $A, B$ with $A, B^T \in \mathbb{R}^{n \times k}$ have been considered.

Since $m = n$ one can find that $f(k, n) = \tilde{f}(k, n, n) = n^2(2k - 1)$ and further that

$$\begin{aligned} f(k, n + c) &= (n^2 + 2nc + c^2)(2k - 1) \\ &= n^2(2k - 1)(1 + 2c/n + c^2/n^2) \\ &= f\left(k + \frac{c}{n}(2k - 1) + \frac{c^2}{n^2}(k - 0.5), n\right) \end{aligned}$$

for $c \in \mathbb{N}$. The parameter $c \in \mathbb{N}$ is used to display how the number of flops change when varying the numbers of rows in $A$ and columns in $B$, and furthermore how the same number of flops can be achieved by only varying the numbers of columns in $A$ and the rows in $B$. This relationship can be used to check whether the time needed to process multiplications with many small buckets or few big buckets is roughly equal as long as the number of flops is the same. Therefore, we did a small experiment starting at $k = 20$, $n = 40$ following these two different paths:

- hold $k = 20$, varying $n = 40 + c$ with $c = 160\zeta$ and $\zeta = 1, 2, \dots$
- hold $n = 40$, varying $k = g(k, n, c)$ with $c = 160\zeta$ and $\zeta = 1, 2, \dots$

with

$$g(k, n, c) = k + \frac{c}{n}(2k - 1) + \frac{c^2}{n^2}(k - 0.5). \qquad (2)$$

Note that it is necessary to ensure that $c/n$ is an even number so that $g(k, n, c)$ becomes a natural number. The pure calculation results obtained in this experiment are depicted in Figure 3.

It can be seen that up to a certain point ($c \approx 1400$) the system does scale according to the flops. After this turning point, the calculation of many small buckets does perform worse. This is most likely because of the rapid increase of bucket handling and memory usage. As a side note, one can observe the same behavior when using LAPACK libraries with a much wider performance gap, which might be caused due to non-avoidable output costs. When adding any type of output (visual or inserts) to the query, the performance gap naturally widens as many more elements have to be processed in the varying $n$ case.

Anyhow, the relatively low gap is interesting when noting that the case $A, B^T \in \mathbb{R}^{n+c \times k}$ uses matrices with $(n + c)k$ entries each, while the corresponding

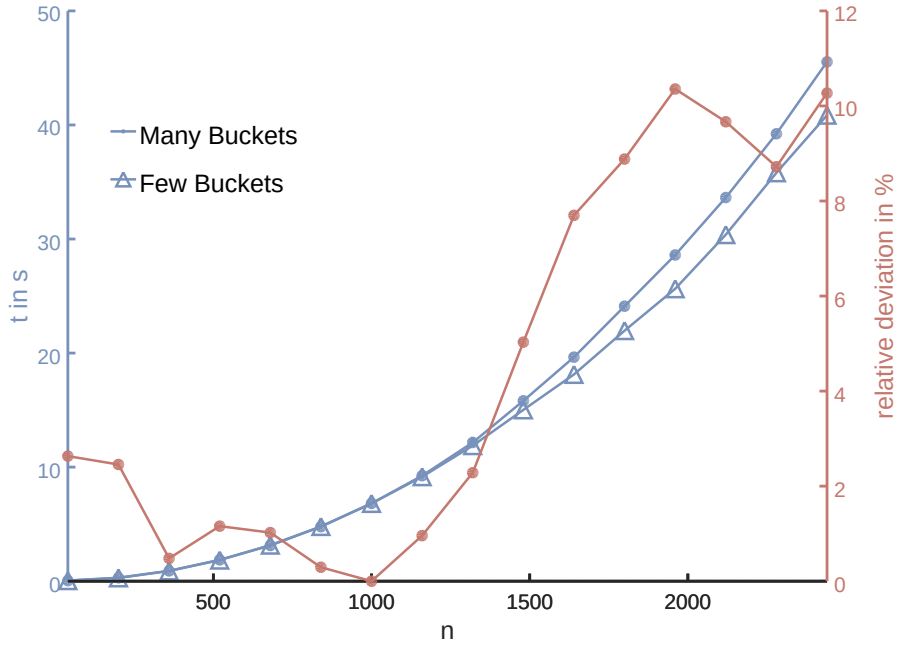



**Figure 3: Performance of matrix uultiplication $A \times B$ with matrices $A, B^T \in \mathbb{R}^{n \times k}$. The blue line with triangle markers (few big buckets) represents multiplications with matrix dimensions $n = 40$ and variable $k(c) = 20 + g(40, 20, c)$ with $g$ from (2), as the blue line with dot markers (many small buckets) represents the counterpart where $k = 20$ and $n = 40 + c$. The function $g$ and its parameter $c$ assures that both cases need the same amount of floating point operations. The red solid line showcases the relative deviation between both approaches.**

case $A, B^T \in \mathbb{R}^{n \times k + g(k,n,c)}$ uses matrices with $n\left(k + \frac{c}{n}(2k-1) + \frac{c^2}{n^2}(k - 0.5)\right)$ elements. These values differ immensely as one of them is increasing in a quadratic manner: For instance, the last points evaluated in the experiments are on the one hand $n = 2440$ and $k = 20$ leading to 48800 elements per matrix and on the other hand $n = 40$ and $k = 72560$ leading to 2902400 elements per matrix, which is already two magnitudes of order higher. This confirms the hypothesis that database systems do way better on dense problems that use "thin-shaped" matrices (i.e., matrices with few columns/rows) or even vectors (see Figure 4), when comparing them to in-memory scientific computation software. In fact, we have tested some explicit linear regression parameter computation using the statistic calculation software R [35] and MonetDB [20], finding that there is basically no performance gap for data in the area of main memory (when using pre-calculated joins). However, one of the big advantages is that database systems are not bound to allocation in main memory and could therefore calculate even a wider range of problems than the aforementioned software.

In contrast to dense problems, the presented schema is much more sufficient for sparse calculations, which are problems where the numbers of non-zero matrix entries are rare ($< 1\%$). These cases usually have much higher dimensions ($> 1e8$) and contain gigabytes or even terabytes of data. Therefore, corresponding calculations are usually more "selective" in comparison to dense problems, which means that the relative time needed for tuple-wise aggregation is lower, while the time needed for data handling (selecting, joining, sorting, etc.) is higher. This is, in regards of processing, somewhat similar to thin-shaped rectangular dense problems (except from joining, index using, etc.), which just have been shown to scale good. Therefore, sparse problems are an ideal use-case for database/SQL calculations, especially since techniques like indexes can be used to improve the effect, as will be seen in the upcoming sections.

### 4.3 Modeling and Implementation in SQL

While SQL in its declarative nature is accompanied with logical optimization, the performance of queries is mostly independent of the way problems are formulated.

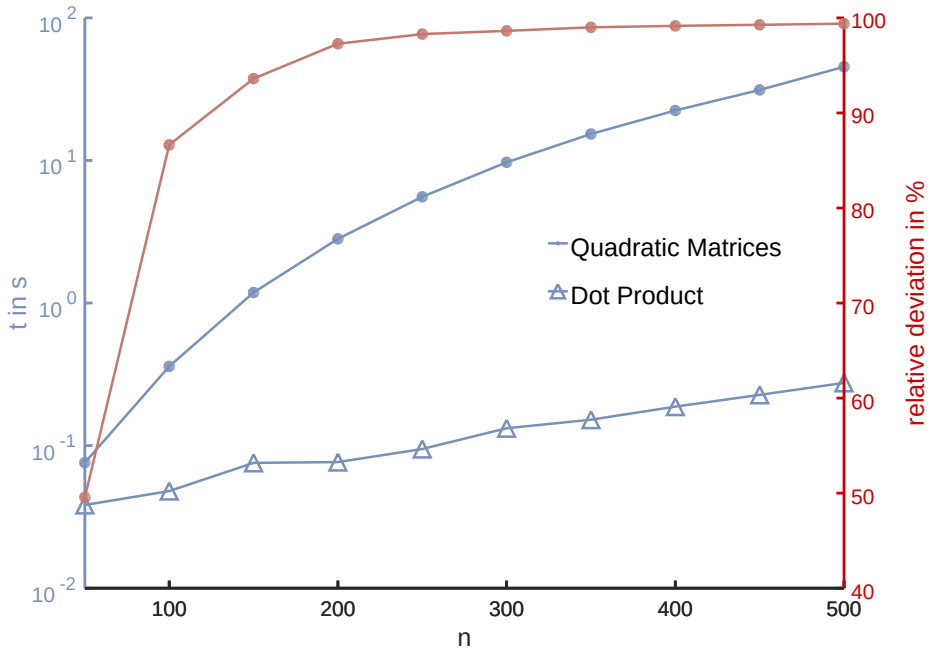



**Figure 4: Matrix multiplication for relations of the same size with $A, B \in \mathbb{R}^{n \times n}$ ( blue dots) and $A, B^T \in \mathbb{R}^{1 \times n^2}$ (blue triangles): thin-shaped rectangular matrices are calculated faster (also according to the number of flops needed) than quadratic ones. Therefore out-of-memory calculations can be motivated for dense vectors and thin-shaped rectangular problems.**

However, as will be described in this section, several influencing aspects, like a reasonable handling of iterations and nesting of SQL statements or providing extra information for the database optimizer using additional sub-queries for optimal index usage, need to be considered when aiming at high performance query processing. Here, we provide a discussion on the use of nested queries and concatenated queries for sparse problems.

### 4.3.1 Nested Queries

As established in former publications [29–31], nesting queries is generally very beneficial since intermediate results can be efficiently used and are not needed to be partially stored or logged. This has been motivated even further in Subsection 4.2 where we have shown that big intermediates scale reasonably well on database systems when output operations can be avoided.

We presented a comparison on a nested and successive realization of forward filtering in Hidden Markov models in [29], showing a performance benefit around factors from 2.1 to 4.7, increasing with problem size. Here we would like to provide a formal discussion on what scientific computation methods can be beneficially nested in SQL.

The main limitation on nesting is that one cannot iterate efficiently over multiple intermediates at the same time. This is due to our observation that common sub-expression detection does usually not work with complex queries in the **from** clause. Unfortunately this occurs in multiple methods, for example when calculating any kind of matrix factorization (orthogonalization, eigenvalue decomposition, singular value decomposition.). These methods require to manipulate (update/insert) multiple matrices in every iteration step and can therefore only be computed by successive queries.

So what kinds of methods can be nested? One of the major purposes of nesting is to deal with iterative algorithms, although a simple series of consecutive operations can be handled alike. Most of the iteration can be expressed as $A_{i+1} = f(A_i, B^{(1)}, \ldots, B^{(m)}, v^{(1)}, \ldots, v^{(n)})$, where $A_{i+1}$ depends on $A_i$ and constant matrices $B^{(1)}, B^{(1)}, \ldots, B^{(m)}$ as well as constant vectors $v^{(1)}, \ldots, v^{(n)}$. If the corresponding relations are only connected via the fundamental operators from Table 2, the iteration is nestable. These methods can be visualized as dependency graphs as depicted in Figure 5.

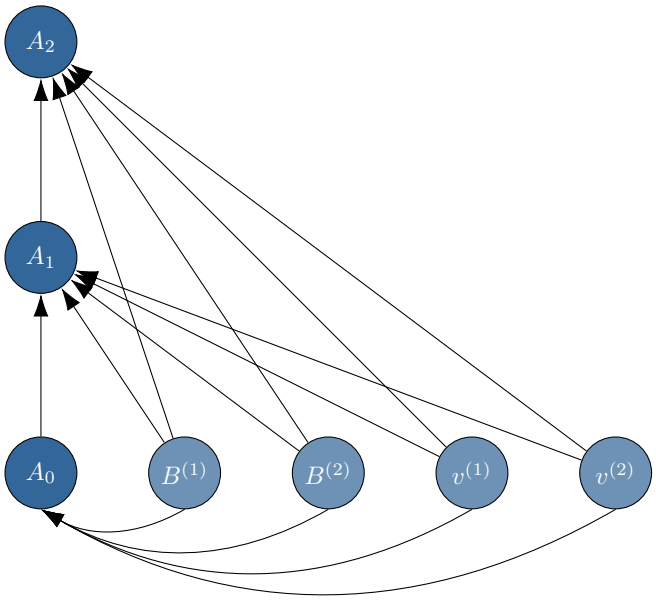



**Figure 5: General dependency flow graph of iterations $A_{i+1} = f(A_i, B^{(1)}, B^{(2)}, v^{(1)}, v^{(2)})$ that can be nested in a meaningful manner in SQL, where $f$ does represent a function that connects the input relations via fundamental operations stated in Table 2. Relations/matrices $B^{(1)}$ and vectors $v^{(1)}$ represent constant tables. The number of constant relations can differ as needed and $A_0$ can be composed of different finished iterations, which have not been visualized, due to clarity.**

A good example for a nestable query with common sub-expressions is the power iteration in Algorithm 1, which has also been depicted as a dependency graph in Figure 6. Another point to be noted is that the initial block $A_0$ can be composed of two (and therefore multiple) different finished iterations

$$A_0 = f(f_1(A_n^{(1)}, \ldots), f_2(A^{(k)}, \ldots)).$$

in the **from** clause. This can ultimately lead into a tree shaped dependency graph. This special nesting of multiple iterations might become relevant for certain algorithms but has not occurred at our research yet. Anyhow, this approach obviously does not cover every kind of nestable queries but includes the vast majority of iterative methods we have investigated.

### 4.3.2 Concatenated Queries

When investigating eigenvalue problems and principal component analysis we encountered sub-methods that consist of a multitude of non-nestable low-cost operations, usually implemented by loops. In this case the $QR$-factorization of a (symmetric) tridiagonal matrix $A \in \mathbb{R}^{n \times n}$ has been analyzed, which occurs frequently when calculating eigenvalues via the $QR$ method. This means, calculating an orthogonal matrix $Q \in \mathbb{R}^{n \times n}$ and

**Algorithm 1:** The power algorithm for the calculation of the absolute greatest eigenvalue of a matrix $A \in \mathbb{R}^{n \times n}$ with starting vector $v_0 \in \mathbb{R}^n$

```
for i = 1, ..., max_iterations do
    ṽ_i = A_{p_o,p_l} v_{i-1}
    ṽ_i = ṽ_i / ||ṽ_i||_2
    if break condition then
        break;
    end
end
```

a triangular matrix $R \in \mathbb{R}^{n \times n}$ so that

$$QR = A = \begin{pmatrix} a_{11} & a_{12} & & \\ a_{12} & a_{22} & \ddots & \\ & \ddots & \ddots & a_{n-1,n} \\ & & a_{n-1,n} & a_{nn} \end{pmatrix}.$$

This can be achieved by using Givens-Rotations, which are rotation matrices

$$G_{kl}(\theta) = \begin{cases} \cos(\theta) & \text{for } i = j = k \vee i = j = l \\ -\sin(\theta) & \text{for } (i = k \wedge j = l) \\ \sin(\theta) & \text{for } (i = l \wedge j = k) \\ 1 & \text{for } (i = j \wedge i \neq l \wedge i \neq k) \\ 0 & \text{other} \end{cases}$$

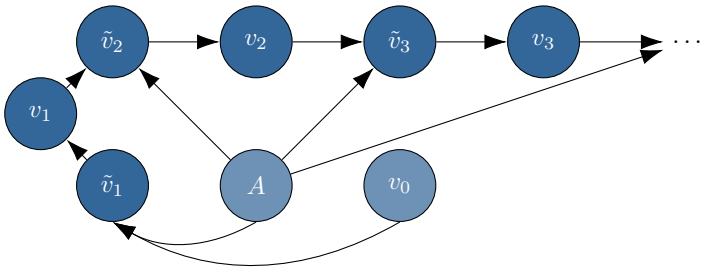



**Figure 6: A dependency flow graph of the power iteration in algorithm 1 as a representative of nestable iterations.**

that are specifically constructed, to eliminate the element $a_{k+1,k} = 0$ by multiplying $A := G_{k+1,k}(\theta)A$. Here, only elements in the upper triangular matrix in the $k$-th and $k+1$-th row are updated or inserted. Therefore,

$$\underbrace{G_{n,n-1}(\theta_{n-1})G_{n-1,n-2}(\theta_{n-2})\dots G_{2,1}(\theta_1)}_{=:Q^T} A = R$$

forms the desired $QR$-factorization. As one can see, calculating a $QR$ decomposition takes $n-1$ Givens-Rotations, where $n$ can possibly be a high number. Additionally, when calculating eigenvalues with the $QR$ method, several decompositions have to be computed, which increases the amount of low-cost queries even further.

As SQL is built on set operations, it naturally does not provide loop-syntax with the exception of system-dependent extra functionality. Therefore, working with large loops that cannot be expressed via joins or similar can be demanding in SQL. Two different points have to be addressed when one is trying to unnest large loops. First, even when any single query has a neglectable processing cost if examined on its own, looping over these queries, i.e. gradually sending queries to the system, performs very poorly. This is because network costs create a massive overhead in this case. Even when working on a desktop system, sending an "empty query" and receiving a database response takes several milliseconds, outperforming the calculation cost of the Givens-Rotation by a margin. The obvious solution is to send multiple queries via larger query plans. However, the query plans cannot be too large, since this can cause serious IO cost for the operating system and logging cost for the database system. There is usually a wide gap of reasonable query plan sizes that perform well, nonetheless, we are using system-dependent loops at the moment, since these perform even better.

### 4.3.3 Providing Sparsity Structure Information

Due to the data size, it is very important to avoid as many full table scans as possible when working on sparse problem. Hence, using indexes is a crucial component for efficient computation for these kinds of operations, as will be discussed in the upcoming section. While optimization usually ensures a good logical way of query computation, it seems that systems cannot "understand" the logical structure of sparse matrices. Therefore, it is sometimes advisable in sparse linear algebra to add extra sub-queries for more efficient use of indexes. This is especially the case when calculating operations that will end with a selection that is caused by the sparsity structure of matrices.

For instance, consider forward filtering in the Hidden Markov model, which can be described as

$$(\mathbf{w}^T A) \circ B_{:,k},$$

with sparse matrices $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$ and $w \in \mathbb{R}^n$, $k \in \{1,\dots,m\}$ and $\circ$ representing element-wise multiplication. Using the discussed Schemas (1) for matrices and vectors, the corresponding query in SQL can be written as

**SQL statement 2:**

```
select wTA.i,wTA.v*B.v
from (
    select A.j as i,
        sum(A.v*w.v) as v
    from A join w on A.i=w.i
    group by A.j ) wTA join B
    on wTA.i=B.i
where B.j=k.
```

While this query will return the correct answer, we have encountered that database systems will calculate the whole vector $w^T A$ before joining with $B$, even if $A$ and $B$ have the necessary indexes. The systems do not take advantage of the sparsity structure of $B$. In other words, the systems calculate two joins fully, although the first join results in tuples that will not be matched

in the second one. To compensate this, it is meaningful to add an additional sub-query that restricts the rows to be calculated in $w^T A$ as follows

**SQL statement 3:**

```
select wTA.i,wTA.v*B.v
from (
    select A.j as i,
        sum(A.v*w.v) as v
    from A join w on A.i=w.i
    where A.j in (
        select i
        from B
        where j=k )
    group by A.j ) wTA join B
    on wTA.i=B.i
where B.j=k.
```

This strategy filters many unnecessary calculated tuples and usually speeds up the calculation enormously (we have encountered factors of 17000 for medium-sized, i.e. $n = 1e7$, sparse problems), depending on the structure of $B$. In the following Subsection 4.4 the SQL statement 3 will also be evaluated as a good examples for efficient index usage.

## 4.4 Impact of Indexes

Indexes are one of the most powerful tools when working with databases to tune query processing time. This section discusses the use of indexes when calculating scientific computations or machine learning algorithms. As there exists little research on this topic, we investigated several different kinds of indexes (Hash, B-trees, GIN) in order to give a good overview of general index behavior. Since most of the sub-methods in Table 1 are compositions of the fundamental operations in Table 2, we have focused on the latter operations, while distinguishing dense and sparse matrices. Furthermore, sparse forward filtering (Hidden Markov models) will be tested as a representative of commonly used machine learning algorithm. Therefore, we choose the SQL statements 2 and 3 in Subsection 4.3 to also show the impact of meaningful query formulation. As GIN indexing is relatively unknown or rarely implemented in comparison to hash functions and B-trees, we would like to start this analysis on giving a small description on how GIN indexes work and how they can be applied to our use-case.

### 4.4.1 GIN Indexes

PostgreSQL supports some index methods for multi-column and multi-dimensional access, namely GIN and GiST. In contrast to the common B-tree, both methods allow for not only exact, range and prefix queries but support also partial match queries, where only a part (e.g. one column) of the composite search key is given in the query.

GiST stands for generalized search tree structure [18]. Its PostgreSQL implementation is lossy in nature and can be used on point or other geometry data, usually derived from double precision data types. Using it on integral matrix indices would require to provide a set of support functions for composite integer key values. For that reason, we used the index method supporting partial match queries, i.e. GIN.

GIN is a generalized inverted index, as known from information retrieval [33]. It was designed for handling cases where the items to be indexed are composite values, and the queries to be handled by the index need to search for values that appear within the composite items, i.e. partial match queries. Internally, a GIN index is a B-tree index constructed over keys where each key is an element of one or more indexed items. Entries in a leaf page contain either a pointer to a B-tree of heap pointers[3], or a simple list of heap pointers if the list is small enough to fit into a single index tuple along with the key value. If used for multi-column indexing, the GIN indexes are implemented by building a single B-tree over composite values (column number, key value) and the heap pointers. The key values for different columns can be of different types, and in our scenario of matrix indices they have the same integral type. This indexing scheme seems to accommodate the sparse characteristics of the matrices in use.

By using the GIN index, exact match queries as well as range and partial match queries can be supported by a single index structure. As a consequence, the index is more compact than using one or more B-tree indexes. Additionally, there is no *preferred* matrix dimension, i.e. column-wise access is about as fast as row-wise access.

### 4.4.2 Problem Oriented Index Creation

The B-tree index and hash functions have been specifically chosen to work on the respective join conditions. When working on matrix multiplication $AB$ and matrix addition $A+B$, the following SQL statements

[3] In PostgreSQL terminology, a heap pointer is an internal reference to the tuple itself, sometimes also known as tuple identifier.

**SQL statement 4:**

```
select A.i, B.j, sum(A.v*B.v)
from A join B on A.j=B.i
group by A.i,B.j
```

and

**SQL statement 5:**

```
select A.i, B.j, A.v+B.v
from A join B on
   A.i=B.i and A.j= B.j,
```

have been used for dense problems. For sparse problems, the element-wise operations have to be adjusted to

**SQL statement 6:**

```
select coalesce(A.i,B.i),
    coalesce(A.j,B.j),
    coalesce(A.v,0.0)+
        coalesce(B.v ,0.0)
from A full join B
    on A.i=B.i and A.j=B.j
```

in order to ensure that the tuples, which represent cases like $0 + b_{ij}$ or $a_{ij} + 0$, are not neglected. However, no matter whether problem structures are sparse or dense, multiplications have the join condition

**SQL statement 7:**

```
      A.j=B.i,
```

and element-wise operations have the condition

**SQL statement 8:**

```
      A.i=B.i and A.j=B.j.
```

Therefore, B-trees and hash functions have been built on the column attribute `j` for relation `A` and attribute `i` for relation `B` for multiplication. On the other hand, it is necessary to create multicolumn indexes on `(i,j)` or `(j,i)` on both relations in order to support element-wise operations. In conclusion, when working with matrices that need element-wise operations and multiplications, one would need to build trees or hash tables using attributes `(i,j),j` or `(j,i),i`, which can be quite costly when table updates are needed. For this reason, one needs indexes on `j` for relation `A` and `B` and on `i` on `w` when calculating forward filters. On the contrary, when using GIN indexes, only `(i,j)` is needed for all operations, which lead to smaller index structures and less updating overhead. Additionally, we tested `(i,j,v)` to check whether PostgreSQL would even work entirely on the index structure, instead of pointing from the index to the corresponding stored relation entries. However, this approach did not yield any effects, except for increasing the size of the index structure.

### 4.4.3 Experimental Results

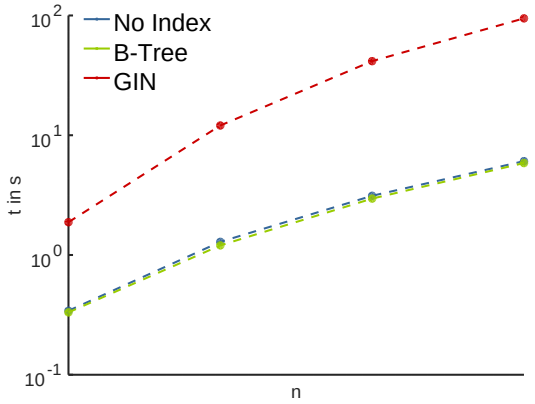
One of the first things we found was that using hash functions on any of our use cases in Table 2 or the forward filtering in SQL statement 2 and SQL statement 3 does perform significantly worse than B-trees or GIN. Therefore, we have decided to neglect them in the following experiments as they seem not to be a reasonable choice for linear algebra operations. The results of the other index types obtained from the experiments are depicted in Figures 7, 8 and 9.

All matrix and vector entries are created as random doubles using the uniform distribution. The corresponding C++ code for creating the data can be found in the Appendix. Experiments on sparse problems for fundamental operations have been evaluated on a grid with varying dimension sizes and branching factors, which here denotes the number of entries in every row of sparse matrices. Dense matrices do not have zero entries, which means that inner joins are sufficient for element-wise operations. Furthermore, we disabled sequential scans when working with indexes, forcing the database system to use index scans on the respective operations.

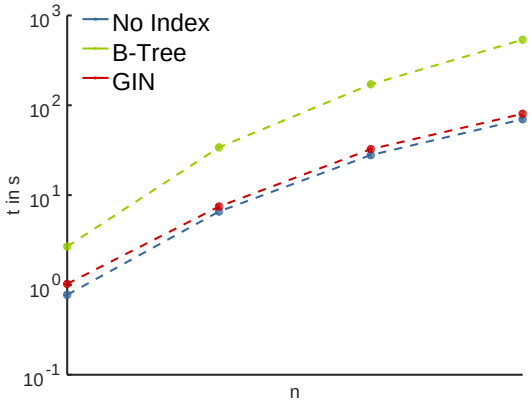
For clarity, we consistently used the following color scheme for the representation of the results (Figure 7, 8 , 9):

- dark blue: no index used
- green: b-tree index
- red: GIN index

It can be seen in Figure 7 that working with indexes on dense scenarios has, in the best case, no performance decrease, but certainly does not improve the speed of calculations. In fact, when using matrices with a relation size of 8.6 megabyte, forcing index usage leads to the materialization of a more than 2 gigabyte intermediate relation and therefore to a bad performance. The first real improvement on fundamental operations can be observed in Figure 8 (a) when calculating element-wise operations

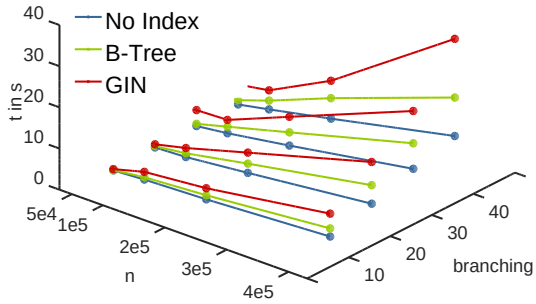



**(a) Dense addition $A + B$ with no indexes (dark blue), B-tree (green) on `(i,j)` for both matrices and a GIN index on `(i,j)` (red).**

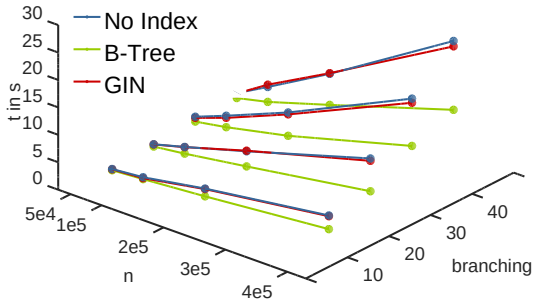



**(b) Dense matrix multiplication with no indexes (dark blue), a B-tree on `(i,j)` (green), a GIN index on `(i,j)` (red).**

**Figure 7: Index tests on dense fundamental operations, comparing B-tree indexes, GIN indexes and no indexes.**




**(a) Sparse multiplication $Aw$ with no indexes (dark blue), a B-tree (green) on `(j)` for $A$ and `i` for $w$, a GIN index (red) on `(i,j)` on both matrices.**




**(b) Sparse Addition $A + B$ with no indexes (dark blue), a B-tree (green) on `(j)` for $A$ and `i` for $B$, a GIN index (red) on `(i,j)` on both matrices.**

**Figure 8: Index tests on sparse fundamental operations, comparing B-tree indexes, GIN indexes and no indexes.**

on sparse matrices using B-trees, which do scale very good with increasing branching factor. However, sparse multiplication does again perform poorly on GIN and B-trees. The results on fundamental operations are somewhat disappointing, but not a surprise, considering that any of these operations works on entire relations. As an example of a frequently used high-selective operation we therefore considered forward filtering on sparse matrices $A, B$ for an operation that will most likely benefit enormously from index usage. The results we have obtained and depicted in Figure 9 do support this assumption. In this scenario, we focused on sparse matrices with a constant branching factor of 20. Furthermore, we tested the two different query formulations in Subsubsection 4.4.2 for the calculation of forward filters. Here, triangles represent the SQL statement 2 with no extra sparsity information and the filled circles represent the SQL statement 3 with extra information on the sparsity structure of `B`.

The first obvious conclusion that can be made is that providing extra information via extra sub-queries does improve performance for all types of queries (with

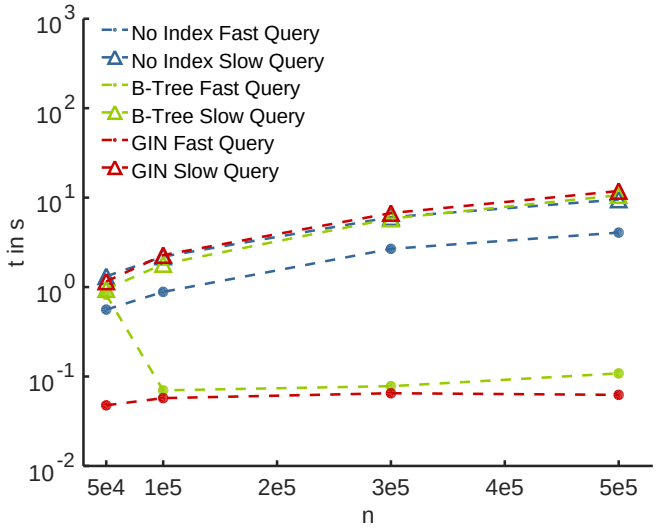



**Figure 9: Index tests on sparse forward filtering, comparing B-tree indexes (green), GIN indexes (red) and no indexes (dark blue). Testing the two different SQL queries 2 (triangles) and 3 (filled circles).**

or without indexes). Furthermore it can be observed that working with indexes only does slightly better than without, when using the statement without extra information. On the contrary, the extra sub-query allows indexes to perform really good. In fact, the processing time is nearly constant with increasing problem size. This can be explained with the constant branching factor and the slowly increasing number of extra floating point operations. However, working without indexes performs significantly worse (here, up to 2 orders of magnitude).

We stopped experimental runs at dimensions of 5e5 as query calculation became very costly, because every type of run had to be processed multiple times. However, we have tested some larger cases ($n > 1e6$) and saw that query costs with index usage remains below 1 second, while non-index-queries do increase continuously as the former trend indicated. We observed performance improvement factors of 20000 from non-index to index calculation at our last runs. We have not encountered any signs that the steady rise of the factor would stop at this point, showing the importance of indexes when processing filters on huge data sets.

In conclusion one can state that the only fundamental operations that benefit from index usage are sparse element-wise field operations when working with B-trees. High-selective operations do heavily benefit from index usage, no matter what kind of index is used. Since covering every possible **join** or **where** condition leads to very big index structures when using B-trees, it is often meaningful to use GIN indexes instead (except when frequently working on sparse element-wise operations).

Closing this section, it should be stated that high-selective operations are not exclusive to sparse problems. There exist numerous methods that will repeatedly work on a low amount of rows or columns of a matrix (so called matrix slices). These operations (see for example Householder Transformation or Givens Rotations) can also massively benefit from index usage. Further examples can be found when working on parallel databases systems with meaningful data partitioning and decompositions of queries into smaller range queries. The parallelism issues will be discussed in Section 5.

## 5 Parallel Linear Algebra in SQL

In this section we discuss intra-operator parallelism for linear algebra operations in SQL on parallel database system. As most of these systems already support inter- and intra-operator parallelism in a traditional database way, we will examine whether these approaches are sufficient for linear algebra. In order to answer this question, we split the parallelism aspect into two parts. The first part investigates established partitioning strategies of parallel linear algebra computations and compares these to established strategies supported in database systems. The second part analyses possibilities to achieve intra-operator parallelism for fundamental operators in SQL, which are evaluated in Section 4, using the partitioning strategies introduced before.

### 5.1 Data-Partitioning

Data partitioning is a well known problem in the area of distributed and parallel databases. Its main purpose is to

distribute data so that the load of all nodes is balanced as good as possible. Since the relation schemas in (1) discussed in Section 4.1 only consist of the primary key attributes and their corresponding value, vertical partitioning is not possible in this scenario. Therefore, horizontal partitioning, which is distributing partitions of tuple sets on different nodes, needs to be considered. Since the partitioning strategy is mainly influenced by local data and the operations to be computed, it is necessary

1. to understand which (linear algebra) operations are the basis for more complex and common algorithms,
2. to understand how these operations work and how it is beneficial to distribute the corresponding data, and
3. to know whether the matrices are dense or sparse.

In Section 3, we have established that the fundamental operations in Table 2 can be used to compute the methods in Table 1 and should therefore be a decisive factor for the way data is stored. While physically storing a transposed matrix is rarely needed and element-wise operations are always local when partitioning on column and row numbers, the main operations for partitioning matrices and vectors are matrix-vector- and matrix-matrix-multiplications.

As will be seen in this section, the question whether matrices are dense or sparse is crucial for choosing the right partitioning strategy. That is why both cases will be dealt in Section 5.2 and 5.3 respectively.

## 5.2 Partitioning of Dense Matrices

In order to show that the standard database partitioning techniques (hash, range) are not sufficient for dense matrices, we introduce the widely used block-wise partitioning [10], which can ideally be seen as a grid of equally sized quadratic matrices.

Here, we consider the operation

$$C = AB$$

as it is one of the fundamental operations and does give a good insight on why block-wise partitioning is beneficial for dense problems. In order to decide what strategy is sufficient we have compared the traditional database partitioning strategies

- row-wise/column-wise (hash)
- range
- round-robin

with the established

- block-wise partitioning,

which can be interpreted as a 2-dimensional range partitioning.

As a starting point, we constrained our analysis on the elements that have to be communicated and the locally needed floating point operations (FLOPs). Furthermore, it is important to note that elements of a result matrix have to be stored according to the chosen partitioning strategy. That means, if matrix elements with row index $i$ and column index $j$ are stored at node $o$, it is in most cases not sufficient to compute a corresponding tuple $(i, j, v)$ at a node $\tilde{o} \neq o$. In this case, the tuple has to be sent back to node $o$ in order to achieve a coherent partitioning scheme. Closing this introductory part, it is noteworthy that the following analysis can easily be expanded to the case of arbitrary rectangular matrices (and vectors). However, in regards to load balancing this would lead to numerous special cases that would unnecessarily inflate the given explanation.

### 5.2.1 Floating Point Analysis for Dense Partitioning Strategies

All the classical database partitioning strategies are related to this scenario in some way. For a formal approach, we will introduce the relevant parameters for the analysis of the partitioning strategies in the following paragraphs (see Table 5 for an overview).

Let $k \in \mathbb{N}$ be the number of nodes, $K = \{1, \ldots, k\}$ the set of all nodes, and let $I = \{1, \ldots, n\}$ denote the set of row or column indices of a given matrix $A \in \mathbb{R}^{n \times n}$. For convenience, we assume $n = k \cdot l$, since handling residual tuples only has a neglectable impact on the load balance. Finally, let $P = \{p_1, \ldots, p_k\}$ an equally sized partition of $I$, e.g.

$$\forall i \in K: \; p_i \in \mathcal{P}(I) \quad \wedge \quad \bigcup_{i=1}^{k} p_i = I \quad \wedge$$
$$\forall j \in K \backslash \{i\}: \quad p_i \cap p_j = \emptyset \quad \wedge \quad |p_i| = l$$

where $p_i$ is stored on node $i$. As the structure of any $p_i$ is not determined in this definition, one can built several different partitions from this.

For example, given a matrix $A = (a_{ij})_{ij} \in \mathbb{R}^{4 \times 4}$, when considering the row $i$ as the partitioning attribute, the matrix can be partitioned on 2 nodes in one of the following ways (the different shades of blue representing

**Table 5: Variables used for the analysis of partitioning strategies**

| Variable | Meaning |
|---|---|
| $n$ | Number of rows and columns of $A, B, C$ |
| $I$ | Set of all row/column numbers |
| $k$ | Number of nodes |
| $K$ | Set of all nodes |
| $l$ | Number of rows per node ($n = kl$) [row-wise partitioning] |
| $P$ | Partition of $I$ |
| $p_o$ | Set of rows in node $o$ |
| $\mathfrak{K}$ | Grid-length [block-wise partitioning] ($\mathfrak{K}^2 = k$) |
| $\mathfrak{l}$ | Number of rows/column of sub-matrices ($\mathfrak{l} = n/\mathfrak{K}$) |

the different partitions):

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

As can be seen, all of these partitions can be expressed using hash partitioning (or round-robin with ordered data). The partitioning $P = (\{1, 2\}, \{3, 4\})$ represents the only meaningful range partitioning.

If

$$A_{p_o,:}$$

denotes the sub-matrix of $A$, which holds all the rows $i \in p_o$, then one can rewrite $C = AB$ from a local perspective of node $o$ as

$$C_{p_o,:} = A_{p_o,:}B$$

and calculate the network and floating point costs of any of the meaningful classic partitioning schemas. It can be seen that in any scenario the whole matrix $B$ is needed to calculate $C_{p_o,:}$.

For the latter operations we have analyzed the two general approaches:

- make $B$ gradually available for every node
- make $A$ gradually available for every node.

The corresponding Algorithm 2 and 3 are written from a local perspective of a node $o \in K$. Starting with Algorithm 2, one can count the overall number of elements, which have to be sent as

**Algorithm 2:** Row-wise strategy 1 for calculating $C = AB$ (perspective: node o)

1: $C_{p_o,:} := \mathbf{0} \in \mathbb{R}^{l \times n}$
2: send $B_{p_o,:}$ to all nodes
3: **on receipt of** $B_{p_l,:}$ $(l \in K \backslash \{o\})$ **do**
4: $\quad C_{p_o,:} += A_{p_o,p_l} B_{p_l,:}$

**Algorithm 3:** Row-wise strategy 2 for calculating $C = AB$ (perspective: node $o$)

1: $C_{p_o,:} := \mathbf{0} \in \mathbb{R}^{l \times n}$
2: send $A_{p_o,p_l}$ to nodes $l$ $(l \in K \backslash \{o\})$
3: **on receipt of** $A_{p_l,p_o}$ $(l \in K \backslash \{o\})$ **do**
4: $\quad C^{(o)}_{p_l,:} = A_{p_l,p_o} B_{p_o,:}$
5: $\quad$ send $C^{(o)}_{p_l,:}$ to node $l$
6: **on receipt of** $C^{(l)}_{p_o,:}$ $(l \in K \backslash \{l\})$ **do**
7: $\quad C_{p_o,:} += C^{(l)}_{p_o,:}$

$$\begin{aligned} \zeta_1 &= \underbrace{k}_{\text{per node}} \overbrace{(k-1)}^{\text{other nodes}} \underbrace{l}_{\text{rows}} \overbrace{n}^{\text{columns}} \\ &= n^2(k-1) \end{aligned}$$

and the necessary FLOPs, which have to be computed locally as

$$\begin{aligned} \kappa_1 &= \underbrace{l}_{\text{rows}} \overbrace{n}^{\text{columns}} ( \underbrace{n}_{\text{mult. per el.}} + \overbrace{n-1}^{\text{sum. per el.}} ) \\ &= nl(2n-1). \end{aligned}$$

The communication costs of Algorithm 3 can be

derived as

$$\begin{aligned}\zeta_2 &= \underbrace{k}_{\text{per node}} \overbrace{(k-1)}^{\text{other nodes}} \underbrace{l}_{\text{rows}} \overbrace{l}^{\text{columns}} \\ &+ \underbrace{k}_{\text{per node}} \overbrace{(k-1)}^{\text{other nodes}} \underbrace{l}_{\text{rows}} \overbrace{n}^{\text{columns}} \\ &= n(k-1)(l+n) \\ &= n^2\left(k - \frac{1}{k}\right)\end{aligned}$$

elements and its local computation costs as

$$\begin{aligned}\kappa_2 &= \underbrace{n}_{\text{rows}} \overbrace{n}^{\text{columns}} (\underbrace{l}_{\text{mult.}} + \overbrace{(l-1)}^{\text{sum.}}) + \underbrace{nl(k-1)}_{\text{subsum.}} \\ &= nl(2n-1)\end{aligned}$$

FLOPs. Furthermore, the costs for the operations $A^TB, AB^T, A^TB^T$ are equal for the strategy that makes $B$ gradually available. The strategy that makes $A$ gradually available for any node is more costly and therefore worse.

### 5.2.2 Squared Block-wise Partitioning

In contrast to the partitioning strategies that are based on one attribute, we now consider squared block-wise partitioning, which can be interpreted as a range partitioning on two attributes. For convenience, the number of nodes $k$ is assumed to be a square number $k = \mathfrak{K}^2$ ($\mathfrak{K} \in \mathbb{N}$). Interpreting the matrix $A$ as a block matrix

$$A = \begin{bmatrix} A_{11} & \dots & A_{1\mathfrak{K}} \\ \vdots & \ddots & \vdots \\ A_{\mathfrak{K}1} & \dots & A_{\mathfrak{K}\mathfrak{K}} \end{bmatrix}$$

with square matrices

$$A_{ij} = \begin{pmatrix} a_{(i-1)\mathfrak{l}+1,(j-1)\mathfrak{l}+1} & \dots & a_{(i-1)\mathfrak{l}+1,j\mathfrak{l}} \\ \vdots & \ddots & \vdots \\ a_{i\mathfrak{l},(j-1)\mathfrak{l}+1} & \dots & a_{i\mathfrak{l},j\mathfrak{l}} \end{pmatrix} \in \mathbb{R}^{\mathfrak{l}\times\mathfrak{l}},$$

a squared block-wise partitioning can be achieved by storing the sub-matrix $A_{i,j}$ on the node $\xi(i,j)$, where $\xi(i,j)$ maps the block indices to the corresponding node number

$$\begin{aligned}&\xi : \{1,\dots,\mathfrak{K}\}^2 \mapsto \{1,\dots,k\} \\ &\xi(i,j) = (i-1)\cdot\mathfrak{K} + j.\end{aligned}$$

**Algorithm 4:** Block-wise strategy 1 for calculating $C = AB$ (perspective: node $\xi(i,j) = o$)

1: $C_{i,j} := \mathbf{0} \in \mathbb{R}^{\mathfrak{l}\times\mathfrak{l}}$
2: **on receipt of** $A_{i,h}, B_{h,j},\ h \in \{1,\dots,\mathfrak{K}\}$ **do**
3: $\quad C_{i,j} += A_{i,h}B_{h,j}$

**Algorithm 5:** Block-wise strategy 2 for calculating $C = AB$ (perspective: node $\xi(i,j) = o$)

1: $C_{i,j} := \mathbf{0} \in \mathbb{R}^{l\times n}$
2: **on receipt of** $B_{j,h}\ h \in \{1,\dots,\mathfrak{K}\}$ **do**
3: $\quad C^{(j)}_{i,h} = A_{i,j}B_{j,h}$
4: $\quad$ Send $C^{(j)}_{i,h}$ to node $\xi(i,h)$
5: **on receipt of** $C^{(h)}_{i,j}\ h \in \{1,\dots,\mathfrak{K}\}$ **do**
6: $\quad C_{i,j} += C^{(h)}_{i,j}$

Coming back to the $4 \times 4$ example, the partitioning on four nodes can be visualized as

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

with

$$\begin{aligned} n &= 4 & k &= 4 \\ \mathfrak{K} &= 2 & \mathfrak{l} &= 2. \end{aligned}$$

For the analysis of multiplication one considers an additional matrix $B \in \mathbb{R}^{n\times n}$ partitioned in the same way as $A$ and that the resulting matrix $C \in \mathbb{R}^{n\times n}$ follows the same grid schema with

$$C_{ij} = \sum_{h=1}^{\mathfrak{K}} A_{ih}B_{hj}. \tag{3}$$

In order to get some insight in the costs we again consider the two different strategies of Algorithm 4 and 5.

The first strategy needs to receive $\mathfrak{K} - 1$ sub-matrices of $A$ and $B$ each and therefore has global network costs of

$$\begin{aligned}\zeta_{b1} &= \underbrace{k}_{\text{per node}} \overbrace{2(\mathfrak{K}-1)}^{\text{sub-matrices}} \underbrace{\mathfrak{l}^2}_{\text{elements}} \\ &= n^2(2\mathfrak{K}-2)\end{aligned}$$

elements. Since Schema (3) consists of $\mathfrak{K}$ matrix multiplications and $\mathfrak{K} - 1$ matrix additions, the floating

**Table 6: Costs for matrix multiplication variations using row-/column-wise (Strategy 1 and 2) or block-wise partitioning (Strategy 1 and 2)**

| Strategy | Communication $\zeta$ in elements | Processing $\kappa$ in FLOPs |
|---|---|---|
| row/column-wise 1 | $n^2(k-1)$ | $nl(2n-1)$ |
| row/column-wise 2 | $n(k-1)(l+n)$ | $n^2(2l-1)$ |
| block-wise 1 | $n^2(2\mathfrak{K}-2)$ | $nl(2n-1)$ |
| block-wise 2 | $n^2(2\mathfrak{K}-1/\mathfrak{K}-1)$ | $nl(2n-1)$ |

point costs can be calculated as

$$\begin{aligned}\kappa_{b2} &= \underbrace{\mathfrak{K}}_{\#\text{mult.}} \overbrace{\mathfrak{l}^2(2\mathfrak{l}-1)}^{\text{matrix mult.}} + \underbrace{(\mathfrak{K}-1)}_{\#\text{add.}} \overbrace{\mathfrak{l}^2}^{\text{add.}} \\ &= n\mathfrak{l}(2\mathfrak{l}-1) + (n-\mathfrak{l})\mathfrak{l} \\ &= \mathfrak{l}^2(2n-1) \\ &= nl(2n-1).\end{aligned}$$

For the second strategy, one has to distinguish the cases $i \neq j$ and $i = j$, since the diagonal case can use the locally available $B_{ii}$ as well. Hence, the communication cost can be calculated as

$$\begin{aligned}\zeta_{b2} = &\underbrace{\mathfrak{K}}_{\#\text{diag. blocks}} \overbrace{(\mathfrak{K}-1)}^{B_{ih}} \underbrace{\mathfrak{l}^2}_{\text{elements}} \\ &+ \underbrace{(k-\mathfrak{K})}_{\#\text{non. diag.}} \overbrace{\mathfrak{K}}^{B_{jh}} \underbrace{\mathfrak{l}^2}_{\text{el.}} \\ &+ \overbrace{k}^{\text{nodes}} \underbrace{(\mathfrak{K}-1)}_{C_{ih}^{(j)}} \overbrace{\mathfrak{l}^2}^{\text{el.}} \\ &= n^2(2\mathfrak{K}-1) - n\mathfrak{l} \\ &= n^2\left(2\sqrt{k} - \frac{1}{\sqrt{k}} - 1\right)\end{aligned}$$

elements to be communicated and

$$\begin{aligned}\kappa_{b2} &= \underbrace{\mathfrak{K}}_{C_{ih}^{(j)}} \overbrace{\mathfrak{l}^2(2\mathfrak{l}-1))}^{\text{mat. mult.}} + \underbrace{(\mathfrak{K}-1)}_{\#\text{add.}} \overbrace{\mathfrak{l}^2}^{\text{add.}} \\ &= n(2\mathfrak{l}^2 - \mathfrak{l}) + n\mathfrak{l} - \mathfrak{l}^2 \\ &= \mathfrak{l}^2(2n-1) \\ &= nl(2n-1)\end{aligned}$$

floating point operations.

All of the results are summarized in Table 6. Since $\mathfrak{K} \geq 2$ and therefore $k \geq 4$, it is easy to see that the block-wise strategies have significantly less communication costs than any row-wise (hash/range) strategy, while having exactly the same local processing cost. It is also noteworthy that this gap expands with increasing $\mathfrak{K}$ and $k$. This theoretically proves the use of block-wise partitioning.

We have to mention that there exist even better strategies, like the Strassens algorithm [38], for matrix multiplications of very huge data sets. However, the presented strategies are sufficient for our motivations and also more suited for implementation in SQL.

### 5.3 Partitioning of Sparse Matrices

In contrast to dense problems the presented block-wise partitioning is not useful for sparse problems. This is mainly because of two reasons:

1. Matrix-matrix multiplication is rare (nearly non-existent) in common sparse algorithms. Instead, matrix-vector multiplications are substantial for these algorithms.

2. Often, it is desired to have matrices in band matrix form, which would heavily skew the load as will be described in the following.

In an ideal scenario, a sparse matrix $A \in \mathbb{R}^{n \times n}$ that is used for multiple multiplications with vectors is in band matrix form. Here, all non-zero entries $a_{ij}$ hold the restriction $|i-j| \leq p$ where $p$ is the bandwidth. There exist several similar definitions for the latter. Here we use

$$p := \min\{|i-j| \mid a_{ij} \neq 0\}. \tag{4}$$

For example, consider a matrix

$$A = \begin{pmatrix} a_{11} & & a_{13} & \\ a_{21} & a_{22} & a_{23} & \\ & a_{32} & a_{33} & \\ & & a_{43} & a_{44} \end{pmatrix}.$$

Due to $a_{13}$, this matrix has a bandwidth of $p = 2$ and has another characteristic which is frequently observable: all diagonal elements are non-zero values. This is often the case since sparse matrices usually represent some sort of correlation between states or objects. For instance, these can be state transition probabilities in Markov models

or any probabilistic state space model [36], hyper-links between websites [5], stiffness and mass properties of finite elements in structural analysis [43] and many more. The correlation also motivates why most of the problems usually use square matrices. Furthermore, depending on model sizes, the size of such matrices can (and do) easily exceed billions of rows and columns or giga- and terabytes of data. As we have established band matrices, the example further demonstrates why block partitioning is not sufficient as this would take the form

$$\begin{pmatrix} a_{11} & & a_{13} & \\ a_{21} & a_{22} & a_{23} & \\ & a_{32} & a_{33} & \\ & & a_{43} & a_{44} \end{pmatrix}$$

showing one partition consisting of only $a_{32}$.

The solution presented here is to range-partition the matrices and vectors row-wise in order to achieve as much locality as possible. For instance, consider range-partitioned matrix vector multiplication using the aforementioned example:

$$\begin{pmatrix} a_{11} & & a_{13} & \\ a_{21} & a_{22} & a_{23} & \\ & a_{32} & a_{33} & \\ & & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{pmatrix}$$
$$= \begin{pmatrix} a_{11}w_1 + a_{13} & \cdot w_3 \\ a_{21}w_1 + a_{22}w_2 + a_{23} & \cdot w_3 \\ w_2 & \cdot a_{32} + a_{33}w_3 \\ & a_{43}w_3 + a_{44}w_4 \end{pmatrix}$$
$$=: \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix}.$$

As can be seen, only few elements have to be sent from one node to another, as the band structure does mostly match locally stored vector elements with locally stored matrix elements, showcasing the usefulness for sparse band matrices.

However, not every sparse problem comes in band form. To apply the presented partitioning and speed up further calculations, it is useful to try to transform such matrices into band form. There exist many different heuristic algorithms which mainly focus either on minimizing the bandwidth or maximizing the elements in the block diagonals. We have implemented two different algorithms, namely: the reverse Cuthill McKee algorithm [9] and spectral partitioning [34]. The first algorithm is a heuristic method which is based on set operations and showed very good results and performance in first tests. The latter method mainly consists of an eigenanalysis and has been implemented with a slightly changed version of the nestable power method presented in Algorithm 1.

The established matrix partitioning strategies have now been presented. In the following subsection, we will present the possibilities for intra-operator parallelism based on these partitioning cases.

### 5.4 Intra-operator Parallelism via Query-Decomposition

After the discussion of established partitioning schemas for dense and sparse matrices, we will discuss the possibility and the need of intra-operator parallelism for linear algebra operations in SQL.

Parallel systems usually support some intra-operator parallelism by processing scans, hashs, aggregates or groupings in parallel, and this is unsurprising as they represent very basic components of query processing. However, as will be seen in the evaluation part (Section 5.5), relying on these parallel operations is not sufficient for linear algebra operations, even with meaningful partitioning. Loosely speaking, the main problem about this approach is that the systems do not seem to understand the structure behind matrices and methods. For example, consider again matrix multiplication $AB$ with $2 \times 2$ block-wise partitioned matrices $A, B \in \mathbb{R}^{n \times n}$. The whole computation can be described as

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}$$
$$= \begin{pmatrix} A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ A_{2,1}B_{1,1} + A_{2,2}B_{2,1} & A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{pmatrix}$$

Following Algorithm 4, it can be seen that for the computation of $AB$, essentially four smaller sub-matrices need to be calculated while every node needs to receive sub-matrices from different nodes. We have found when a database system receives the basic matrix multiplication query

**SQL statement 9:**

```
select a.i,b.j,sum(a.v*b.v)
from a join b on a.j=b.i
group by a.i,b.j,
```

it creates the full intermediate table $A \bowtie B$ usually by hash joining, since the system does not realize how the operation is influenced by the partitioning strategy. Besides the point that this approach works on a comparatively large intermediate relations $A \bowtie B$ ($n^3$ tuples), rather than smaller intermediate relations $A_{i,h} \bowtie B_{h,j}$ ($n^3/8$ tuples), the main troubling aspect is

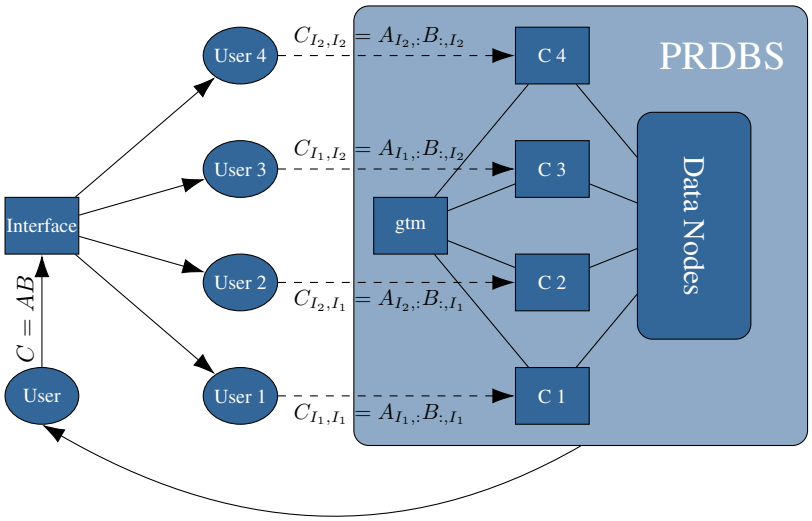



**Figure 10: Intra-operator Parallelism using 4 processes on a PRDBMS setup with at least 4 datanodes. The interface decomposes the operation $C = AB$ into the respective range queries for the operations $C_{I_j,I_k} = A_{I_j,:}B_{:,I_k}$ and starts 4 processes which sends the queries to the database. If consecutive queries are scheduled, the interface has to ensure that all processes have successfully finished.**

that one cannot influence how and where aggregations are finalized, and how many nodes work on the aggregate. As will be evaluated in the next section, it seems that the whole aggregation might even be calculated by one node only.

As transparent parallel calculation of queries is desired and usually implemented, the easiest way to achieve real parallel processing is to decompose the original query in multiple meaningful smaller queries, rather than rewriting the whole system, which is even often not possible.

We have implemented an interface which creates $K$ processes ($K$ = number of nodes) that simultaneously send sub-queries to the database system for some methods in Tables 1 and 2. Currently, there are two main types of methods that we distinguish. Firstly, methods, which mainly consist of few huge fundamental operations, cannot be processed efficiently with a single process. The most obvious choice are fundamental operations itself. These operations are also a convenient starting point as these operations only need one phase of simultaneous sub-querying and final synchronization. This approach has been visualized in Figure 10 for a matrix multiplication on four nodes. The second category of methods is to calculate a high amount of low cost fundamental operations. An example for this would be calculating a $QR-$decomposition using Givens rotation as presented in Subsection 4.3.

## 5.5 Evaluation

This subsection describes experiments that use the techniques for parallel query calculation explained in the previous Subsection 5.4. We evaluated the different partition strategies discussed in Subsection 5.1 for dense and sparse problems on fundamental operators, to prove our claim that the block-wise partitioning technique presented in Subsubection 5.2.2 outperforms traditional database strategies on the dense fundamental operations in Table 2.

### 5.5.1 Experimental Setup

#### I. Choice of Database Systems

We are currently using Postgres-XL 9.5 R1.6 [1] as the parallel relational database systems. The choice is mainly based on the fact that PostgresXL is a free system with high functionality, since it is based on the widely known and used PostgreSQL. However, it has been shown in previous tests [30] that PostgreSQL is comparatively slow in comparison to other database systems like MonetDB. One of the key aspects here is the row-wise storage scheme of PostgreSQL, which has been shown to be inferior to column-based schemes for scientific computational queries. In the future, we are aiming at column and vector-wise storage based systems to ensure better performance especially for dense problems.

## II. Database Architecture and Configuration

We now briefly explain the architecture of Postgres-XL 9.5 R1.6 in order to understand how the simultaneously processed queries are computed. Parallel database systems often consist of three main components:

- Global Transaction Managers (GTM),
- Coordinators,
- Data Nodes.

Data nodes store data and process local queries. Data nodes need the most computing power as most of the (sub-)queries are computed on the data nodes. The Coordinators connect the data nodes and fulfill multiple roles, coordinating queries, ensuring load balancing, and finalizing aggregates. Finally, the main task of the Global Transaction Manager is to ensure the Multiversion Concurrency Control (MVCC) and therefore does not need much computational power.

Depending on the average amount of simultaneously processed queries, it is useful to use multiple transaction managers or proxy versions. Since this does not apply to our analysis, we are currently using one GTM, five GTM-Proxys, five coordinators, and ten data nodes as can be seen in Figure 11. Besides the GTM, which is using its own node, every node contains two data nodes and one coordinator. We have chosen ten data nodes as at least nine processing nodes are needed for a reasonable test of the block-wise partitioning technique presented in Subsubsection 5.2.2 and the query decomposition strategy in Subsection 5.4 and the local machines we use have two cores each. The respective setup regarding the number of GTMs, GTM-proxys and coordinators has been motivated by suggestions in the official documentation of Postgres-XL [1]. The main reasoning here is to balance load and provide data locality as much as possible. The hardware specifications of the 5 processing nodes can be found in Table 7. As the default settings of the Postgres-XL components are not designed for few load intense queries, we changed them according to the Table 8 and 9, and these tables show only the parameters that we have changed.

### 5.5.2 Experiments

After setting up the hardware and implementing the different processing methodologies for multiplications in Subsection 5.2 and 5.3, we here present the results of the different experiments. Each experiment is executed three times, and the average value of the three runs is presented. The code for creating matrices and vectors can be found in Appendix A.

## I. Partitioning: Matrix Multiplication

The first evaluation on a parallel database system is a simple matrix multiplication $A \times B$ with $A, B \in \mathbb{R}^{n \times n}$ using different partitioning types, but the same basic SQL statement:

**SQL statement 10:**

```
select A.i,B.j,sum(A.v*B.v)
from A join B on A.j=B.i
group by A.i,B.j.
```

Postgres-XL supports two different ways to create different partitioning schemes. The first one is the **distribute by** statement, which is an system specific extension of the **create table** statement. This allows transparent partitioning of different standard schemes like hash, modulo, or round-robin. The second approach is to use a master table and several child nodes that inherit the schema from the master table. Each child table is stored on a different data node and a set of rules on insertion into the master table distributes the data accordingly to the child tables. This approach seems pretty handy for block partitioning as it gives users a relatively easy way to partition over multiple attributes.

In the experiment we compared block partitioning using the master child approach, hash partitioning on the row and the column index, as well as block partitioning via round-robin using the **distribute by** statement. The results of this test are depicted in Figure 12. It can be seen that using master and child tables (black solid line) is performing significantly worse then any approach using **distribute by**. Furthermore, block partitioning via **distribute by** (green dashed line) performs only slightly better (about 13 $\%$ at $n = 1800$) than the hashed row (red dashed line) or column (blue dashed line) versions.

The bad performance of the master child approach can be easily explained when looking at the execution plan of the database system. As Postgres-XL 9.5r1.6 is incapable of using any of the partitioning information given by internal statistics or the insertion rules, it materializes the master tables and executes the join and the aggregation accordingly. As this is the worst case scenario, one can only speculate that this behavior might change in future releases and perform significantly better on other systems.

It is difficult to reason why the performance gap between hash and block partitioning is not wider. Postgres-XL supports query execution plans, but does not offer many details on its intra-operator parallelism. What can be seen is that Postgres uses parallel

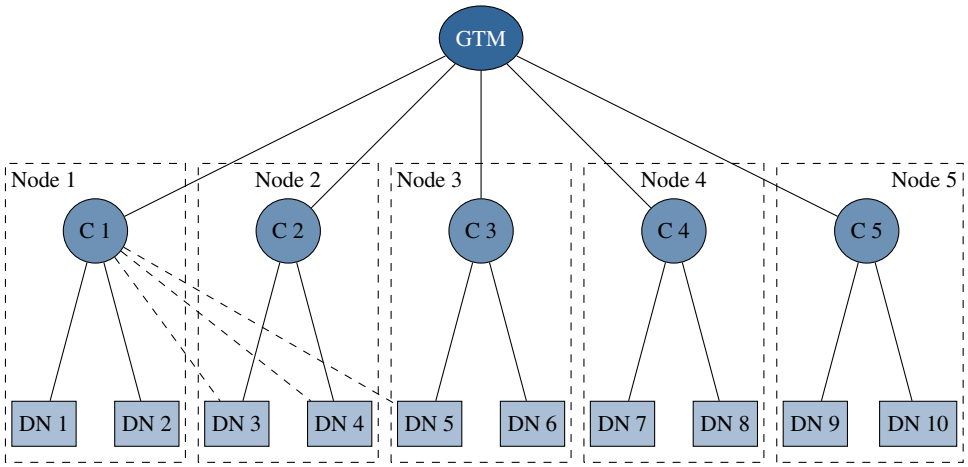



**Figure 11: The parallel database architecture setup: The Global Transaction Manager (GTM) is installed on an additional Node. Note that the Coordinators (C $i$) are connected to every Datanode (DN $i$). These internode connections are only shown for the first Coordinator for simplicity. Additionally, GTM proxy nodes have been established on every processing node in order to relieve the GTM.**

**Table 7: Hardware Specifications of the processing nodes containing 2 data nodes and 1 coordinator each.**

| Parameter of processing nodes | Parameter Value |
|---|---|
| Operating System | CentOS 7 |
| Processors | 2 × 1.9GHz |
| Cache Size | 2 × 4 MB |
| RAM | 16 GB DDR4 (2133 Mhz) |
| Secondary Storage (Data) | 1 TB |
| Secondary Storage (OS) | 50 GB |
| Secondary Storage (Temporary Table Processing) | 20 GB SSD |

**Table 8: Non-default parameters of PostgreSQL as used for Coordinators**

| Parameter | Value |
|---|---|
| shared_buffers | 1GB |
| effective_cache_size | 3GB |
| work_mem | 52428kB |
| maintenance_work_mem | 512MB |
| min_wal_size | 4GB |
| max_wal_size | 8GB |
| checkpoint_completion_target | 0.9 |
| wal_buffers | 16MB |
| default_statistics_target | 500 |
| random_page_cost | 4 |
| max_pool_size | 100 |
| max_connections | 400 |
| autovacuum_vacuum_scale_factor | 0.01 |
| autovacuum_vacuum_cost_limit | 1000 |

**Table 9: Non-default parameters of PostgreSQL as used for Data Nodes**

| Parameter | Value |
|---|---|
| shared_buffers | 1536MB |
| effective_cache_size | 4608MB |
| work_mem | 78643kB |
| maintenance_work_mem | 768MB |
| min_wal_size | 4GB |
| max_wal_size | 8GB |
| checkpoint_completion_target | 0.9 |
| wal_buffers | 16MB |
| default_statistics_target | 500 |
| random_page_cost | 4 |
| max_pool_size | 100 |
| max_connections | 400 |
| autovacuum_vacuum_scale_factor | 0.01 |
| autovacuum_vacuum_cost_limit | 1000 |

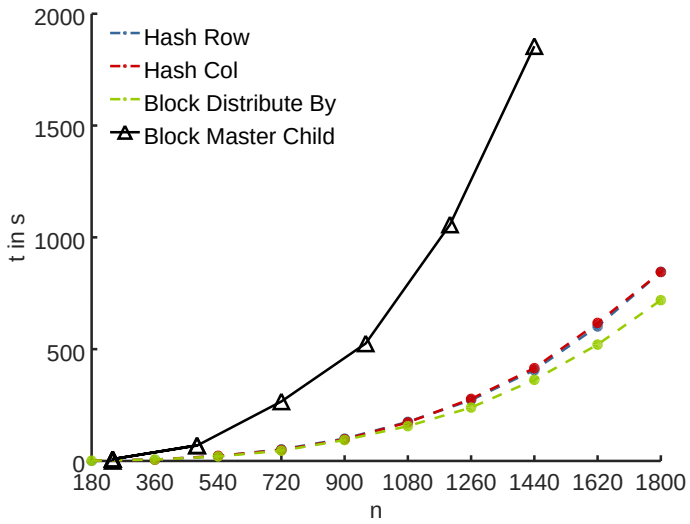



**Figure 12: Results of matrix multiplication on different partitioning schemes. Green represents block partitioning, red depicts hash partitioning on the column attribute, blue represents hash partitioning on the row attribute and the solid black line depicts block partitioning using master and child tables.**

hash aggregation and parallel hash joins on all three calculations. It seems that Postgres computes some sub-results of the whole matrix on each node and aggregates these results at the end, rather than computes whole sub-matrices on different nodes as described in Algorithm 4 and 5. However, it can surely be stated that the block-partitioning scheme is speeding up calculations in a proportional manner to its problem size.

## II. Multi-Query Matrix Multiplication

In the second experiment we investigated how good the multi-query approach is performing on block partitioning over a dense matrix. We tested the two different partitioning schemes described in the former experiment. When using the **distribute by** partitioning, the corresponding sub-queries for the multi-query method are simple range query variations, like

**SQL statement 11:**

```
select A.i,B.j,sum(A.v*B.v)
from A join B on A.j=B.i
where A.i between min_ai and max_ai
    and B.j between min_bi and max_bi
group by A.i,B.j.
```

When working with child tables, one can directly implement the block partitioning multiplication of Algorithm 4. On a $3 \times 3$ node-grid, one sub-matrix $C_{i,j}$ can be computed as follows

**SQL statement 12:**

```
select i,j, sum(v) from (
    select ai1.i as i, b1j.j as j,
        sum(ai1.v*b1j.v) as v
    from ai1
        join b1j on ai1.j=b1j.i
    group by ai1.i,b1j.j
    union all
    select ai2.i as i, b2j.j as j,
        sum(ai2.v*b2j.v) as v
    from ai2 join b2j
        on ai2.j=b2j.i
    group by ai2.i,b2j.j
    union all
    select ai3.i as i, b3j.j as j,
        sum(ai3.v*b3j.v) as v
    from ai3 join b3j
        on ai3.j=b3j.i
    group by ai3.i,b3j.j
) temp
group by i,j;
```

Here, the lower case relations `aik` and `bkj` denote the child tables, representing the partition corresponding to $A_{i,k}$ and $B_{k,j}$.

The results of the experiment are depicted in Figure 13. The black dashed line represents the

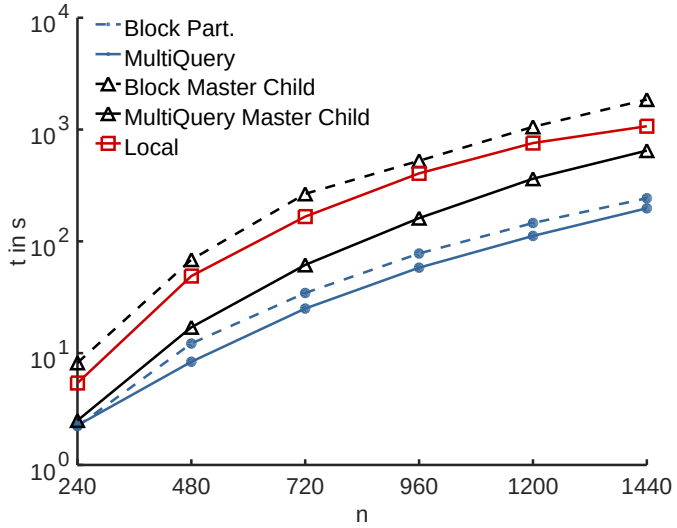



**Figure 13: Results of matrix multiplication with block partitioning and multi-query processing. Blue lines represent block partitioning using Postgres-XLs distribute by-statement. Black lines depict block partitioning using master and child tables. Dashed lines are to associate with standard one-query processing fully managed by the database system, as solid lines represent the proposed multi-query approach. Finally, the red line represents matrix multiplication on the local setup in Table 3 and 4.**

master child approach with one query (full database management), whereas the solid black line represents the aggregate multi-query approach. It can be seen that the multi-query approach does constantly perform better than the normal approach. The speedup factor is consistently between 3 and 4. As mentioned before, using **distribute by** at table creation seems to be better implemented in Postgres-XL as it transparently uses intra-operator parallelism in form of parallel hashs and aggregations. Therefore, the gap between the one-query version (blue dashed line) and the multi-query version (blue solid line) is not wide, but still significant Here, the speedup factor is between 1.25 and 1.45, with a suspected limit at 1.25. In other words, the multi-query approach takes a fifth of the time less than the one-query approach, making it a preferable choice for multiplication.

For a better grading of these results, we added the processing speed of local computation (red) with the setup from Table 3 and 4 into Figure 13. The local setup is slightly faster than one processing node of the parallel database setup. Additionally, there is no coordination overhead and network traffic, and therefore one can expect that the parallel system will not be 9 times faster than the local version. Indeed, the speedup factor for the multi-query approach is between 5.5 and 7, whereas the speedup factor for the one-query approach is between 4 and 5.2. This again emphasizes the benefit of the proposed decomposition-technique in Section 5.4.

## III. Sparse Matrix-Vector Multiplication

In this last experiment we tested the effect of different partitioning strategies and possible multi-query methods for sparse matrix vector multiplication $A \times w$ with $A \in \mathbb{R}^{n \times n}$ and $w \in \mathbb{R}^n$. For this, we created random matrices $A \in \mathbb{R}^{n \times n}$ with constant branching factor, i.e. in any row the diagonal value is non-zero as well as in 20 non-diagonal elements. We have tested two main partitioning strategies:

1. round-robin
2. range

where one can already expect that the range partitioning will surpass the round-robin one because of its data locality as presented in Section 5.3 The non-diagonal column indices are all randomly chosen (uniform distribution) within a (relatively wide) $\pm 3n/40$ band around the diagonal element to simulate the benefit of data locality when using range partitioning.

In contrast to the prior experiments, it is possible in Postgres-XL to achieve 100% local calculations in this setup using replicated vectors $w \in \mathbb{R}^n$ and the **execute direct on** statement, which enables complete local processing (but currently no inter-node computation).

The result of the experiment is depicted in Figure 14. It can be seen that the round-robin partitioning of $A$ (dashed blue line) performs way worse than the range

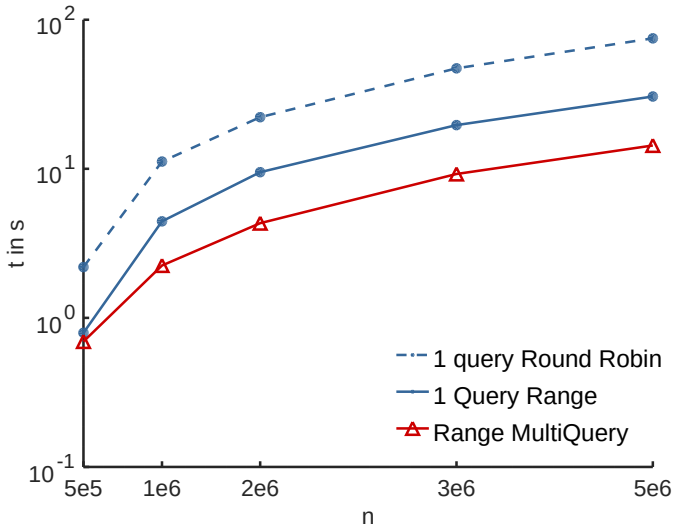



**Figure 14: Sparse Multiplication $Aw$ with 21 elements per row for matrix $A \in \mathbb{R}^{n \times n}$ and dense vector $w \in \mathbb{R}^n$. The vector $w$ has been replicated for this setup on every datanode. The dashed blue line represents runtimes of a one-query approach with round-robin matrix partitioning, whereas the blue solid line depicts the runtime of a range partitioned matrix $A$. Finally, the red line represents the multi-query approach.**

partitioning strategy (solid blue line). This is expected, since when processing with range partitioning, the data nodes already have all the tuples needed to fully calculate the desired results, whereas in the case of round-robin partitioning the nodes either calculate sub-aggregates, which have to be merged and finalized, or are demanding all needed tuples from other datanodes.

However, the range partitioned multi-query approach (red line) does heavily outperform the one query approach with a speedup factor of around 2. The comparatively wide gap in this setup can be explained as in this scenario the nodes operate completely local with no communication needed between coordinators and datanodes. This is especially meaningful as Postgres-XL often calculates each query on multiple nodes, even in the multi-query dense matrix multiplication scenario, where reasonable data partitioning has been offered. Similarly to this setup, one can benefit from this approach when calculating element-wise matrix operations.

In conclusion it can be said that the presented experiments have shown that providing meaningful partitioning and decomposing big linear algebra queries into respective sub-queries do improve significantly the performance of these operations and should therefore be considered when needed.

## 6 Conclusion and Future Work

In this article we have presented our research about the efficient computation of common scientific calculations needed for machine learning algorithms and several other areas of data science. A concise historical overview of former results has been given, which included motivational reasons for the use of parallel database systems and SQL for these non-traditional computations.

After that, several key aspects of performance for local and parallel computations have been discussed. Limitations for this approach, like dense matrix multiplication on row-stores or loops that consist of many low-cost queries, have been explained and possible solutions have been discussed. One of the main techniques that we have introduced and motivated is the parallel computation of common sub-methods using multiple synchronized database processes that simultaneously send sub-queries to the systems, in order to achieve intra-operator parallelism. This approach has been shown to perform consistently and significantly better than one-query processing fully managed by a database system (Postgres-XL 9.5 R1.6).

In the near future, we will focus on three main aspects. First, we will further tune query plans and develop meaningful intra-operator parallelism for the established sub-methods. Additionally, we will investigate the computation of fast fourier transformations in SQL. Secondly, since Postgres-XL has not been performing as good as we initially hoped (mainly on dense problems),

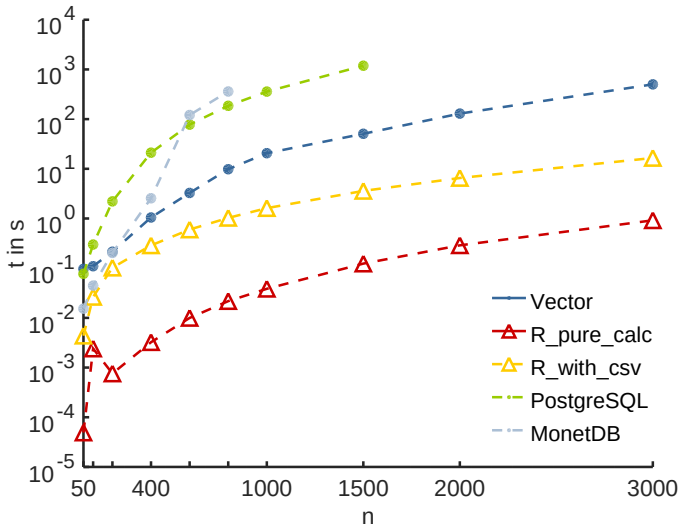



**Figure 15: Local matrix multiplication $AB$ with $A, B \in \mathbb{R}^{n \times n}$ on different database systems and BLAS/R (3.4.2): red (BLAS/R (3.4.2) no data saving), yellow (BLAS/R with storing result into a CSV file), blue (Actian Vector 5.0), green (PostgreSQL 10.1) and turquoise (MonetDB v11.27.5).**

we will putting efforts into testing different parallel database systems with the parallel methods we have implemented in SQL so far. Therefore, we have recently looked into Actian's column store Vector [44], which has been performing very well on TCP benchmarks, and its parallel version VectorH (also known as Vector in Hadoop) [8]. A first simple test has been depicted in Figure 15, where a simple matrix multiplication $C = A \times B$ (with **insert** for $C$) has been tested on different database systems on the hardware outlined in Table 3. As explained throughout the article, testing dense matrix multiplication is somewhat crucial as this operation does perform significantly worse in comparison to linear algebra libraries.

In Figure 15, we compared the database systems Actian Vector 5.0 (blue), PostgreSQL 10.1 (green) (tuned according to Table 4), MonetDB v11.27.5 (turquoise) and the linear algebra library BLAS called from R 3.4.2 (red and yellow). MonetDB and Actian Vector have been used on default settings. Since it is unfair to compare R's full in-memory calculation (yellow) to database systems writing on disk, we added an R calculation (yellow), which writes the resulting matrix $C$ into a CSV file. As can be seen, Postgres performs around factor 50 slower than Vector. The behavior of MonetDB is not fully understood as it performs really fast on small problems and then becomes significantly slower for increasing dimensions. However, the gap between the CSV writing BLAS/R and Actian Vector is only around one magnitude, which is impressive, considering the lack of SIMD/cache-hit possibilities on this kind of operations as explained in Subsection 4.2.

This result leaves us confident that it is a reasonable choice to use Vector and VectorH for our proposed framework. Therefore, as a last step, we will compare our implemented methods to implementations on several different MapReduce-based big data systems. Currently, we consider Apache Hadoop, Apache Flink, and Apache Spark as meaningful choices for this comparison.

## Acknowledgements

We would like to thank the anonymous reviewers of the first version of this paper for their many detailed and constructive comments that helped to improve this paper.

## References

[1] 2ndQuadrant, "Postgres-XL official website," 2018. [Online]. Available: https://www.postgres-xl.org

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, and Demmel, *LAPACK Users' Guide*, 3rd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999.

[3] M. Boehm, B. Reinwald, D. Hutchison, P. Sen, A. V. Evfimievski, and N. Pansare, "On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML," *PVLDB*, vol. 11, no. 12, pp. 1755–1768, 2018.

[4] E. A. Brewer and P. Chen, Eds., *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*. USENIX Association, 2004.

[5] S. Brin and L. Page, "The Anatomy of a Large-scale Hypertextual Web Search Engine," *Comput. Netw. ISDN Syst.*, vol. 30, no. 1-7, pp. 107–117, Apr. 1998.

[6] CACM Staff, "Big Data," *Commun. ACM*, vol. 60, no. 6, pp. 24–25, May 2017.

[7] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink™: Stream and Batch Processing in a Single Engine," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015.

[8] A. Costea, A. Ionescu, B. Răducanu, M. Switakowski, C. Bârca, J. Sompolski, A. Łuszczak, M. Szafrański, G. De Nijs, and P. Boncz, "VectorH: taking SQL-on-Hadoop to the next level," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 1105–1117.

[9] E. Cuthill and J. McKee, "Reducing the Bandwidth of Sparse Symmetric Matrices," in *Proceedings of the 1969 24th National Conference*, ser. ACM '69. New York, NY, USA: ACM, 1969, pp. 157–172.

[10] H. Eves, *Elementary Matrix Theory*, ser. Dover Books on Mathematics Series. Dover, 1966. [Online]. Available: https://books.google.de/books?id=ayVxeUNbZRAC

[11] G. Graefe, "Volcano — An Extensible and Parallel Query Evaluation System," *IEEE Trans. on Knowl. and Data Eng.*, vol. 6, no. 1, pp. 120–135, Feb. 1994.

[12] S. Greg, R. Treat, and C. Browne. Tuning Your PostgreSQL Server. Accessed September 09, 2018. [Online]. Available: https://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server

[13] H. Grunert and A. Heuer, "Datenschutz im PArADISE," *Datenbank-Spektrum*, vol. 16, no. 2, pp. 107–117, 2016.

[14] H. Grunert and A. Heuer, "Rewriting Complex Queries from Cloud to Fog under Capability Constraints to Protect the Users' Privacy," *Open Journal of Internet Of Things (OJIOT)*, vol. 3, no. 1, pp. 31–45, 2017, special Issue: Proceedings of the International Workshop on Very Large Internet of Things (VLIoT 2017) in conjunction with the VLDB 2017 Conference in Munich, Germany. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:101:1-2017080613421

[15] H. Grunert and A. Heuer, "Privacy Protection through Query Rewriting in Smart Environments," in *Proceedings of the 19th International Conference on Extending Database Technology EDBT*. Bordeaux, France: OpenProceedings.org, March 15-16, 2016, pp. 708–709.

[16] S. Günnemann, "Machine Learning Meets Databases," *Datenbank-Spektrum*, vol. 17, no. 1, pp. 77–83, 2017.

[17] S. Hasani, S. Thirumuruganathan, A. Asudeh, N. Koudas, and G. Das, "Efficient Construction of Approximate Ad-hoc ML Models Through Materialization and Reuse," *Proc. VLDB Endow.*, vol. 11, no. 11, pp. 1468–1481, Jul. 2018.

[18] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer, "Generalized Search Trees for Database Systems," in *Proceedings of 21th International Conference on Very Large Data Bases*, Zurich, Switzerland, September 11-15, 1995, pp. 562–573.

[19] J. M. Hellerstein, C. Ré, F. Schoppmann, Z. D. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar, "The MADlib Analytics Library or MAD Skills, the SQL," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-38, Apr 2012.

[20] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten, "MonetDB: Two Decades of Research in Column-oriented Database Architectures," *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 40–45, 2012.

[21] International Organization for Standardization, "Database Languages – SQL, ISO/IEC 9075-*:2003," ISO, Geneva, CH, Standard, 2003.

[22] International Organization for Standardization, "Database Languages – SQL, ISO/IEC 9075-*:2008," ISO, Geneva, CH, Standard, 2008.

[23] J. Jiang, F. Fu, T. Yang, and B. Cui, "SketchML: Accelerating Distributed Machine Learning with Data Sketches," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: ACM, 2018, pp. 1269–1284.

[24] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The Case for Learned Index Structures," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: ACM, 2018, pp. 489–504.

[25] R. B. Lehoucq, D. C. Sorensen, and C. Yang, *ARPACK users' guide - solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods*, ser. Software, environments, tools. SIAM, 1998.

[26] T. Li, J. Zhong, J. Liu, W. Wu, and C. Zhang, "Ease.Ml: Towards Multi-tenant Resource Sharing for Machine Learning Workloads," *Proc. VLDB Endow.*, vol. 11, no. 5, pp. 607–620, Jan. 2018.

[27] S. Luo, Z. J. Gao, M. Gubanov, L. L. Perez, and C. Jermaine, "Scalable linear algebra on a relational database system," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, vol. 33, no. 4, 2017, pp. 523–534.

[28] S. Luo, Z. J. Gao, M. Gubanov, L. L. Perez, and C. Jermaine, "Scalable Linear Algebra on a Relational Database System," *SIGMOD Record*, vol. 47, no. 1, pp. 24–31, March 2018.

[29] D. Marten and A. Heuer, "Machine Learning on Large Databases: Transforming Hidden Markov Models to SQL Statements," *Open Journal of Databases (OJDB)*, vol. 4, no. 1, pp. 22–42, 2017. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:101:1-2017100112181

[30] D. Marten and A. Heuer, "A Framework for Self-Managing Database Support and Parallel Computing for Assistive Systems," in *Proceedings of the 8th ACM International Conference on PErvasive Technologies Related to Assistive Environments*, Corfu, Greece, July 1-3, 2015, pp. 1–4.

[31] D. Marten and A. Heuer, "Transparente Datenbankunterstützung für Analysen auf Big Data," in *Proceedings of the 27th GI-Workshop Grundlagen von Datenbanken*, Gommern, Germany, May 26-29, 2015, pp. 36–41.

[32] M. Navas and C. Ordonez, "Efficient computation of PCA with SVD in SQL," in *Proceedings of the 2nd ACM SIGKDD Workshop on Data Mining Using Matrices and Tensors*, Paris, France, June 28, 2009.

[33] PostgreSQL Global Development Group, "PostgreSQL 9.6 Documentation, GIN index," 2017. [Online]. Available: https://www.postgresql.org/docs/9.6/static/gin-implementation.html

[34] A. Pothen, H. D. Simon, and K.-P. Liou, "Partitioning Sparse Matrices with Eigenvectors of Graphs," *SIAM J. Matrix Anal. Appl.*, vol. 11, no. 3, pp. 430–452, May 1990.

[35] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2016. [Online]. Available: https://www.R-project.org/

[36] L. R. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," in *Readings in Speech Recognition*, A. Waibel and K.-F. Lee, Eds. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990, pp. 267–296.

[37] M. Stonebraker, D. J. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin, "MapReduce and parallel DBMSs: friends or foes?" *Commun. ACM*, vol. 53, no. 1, pp. 64–71, 2010.

[38] V. Strassen, "Gaussian Elimination is Not Optimal," *Numer. Math.*, vol. 13, no. 4, pp. 354–356, Aug. 1969.

[39] M. Vartak, J. M. F. da Trindade, S. Madden, and M. Zaharia, "MISTIQUE: A System to Store and Query Model Intermediates for Model Diagnosis," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18. New York, NY, USA: ACM, 2018, pp. 1285–1300.

[40] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache Spark: A Unified Engine for Big Data Processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016.

[41] B. Zhang, D. V. Aken, J. Wang, T. Dai, S. Jiang, J. Lao, S. Sheng, A. Pavlo, and G. J. Gordon, "A Demonstration of the OtterTune Automatic Database Management System Tuning Service," *PVLDB*, vol. 11, no. 12, pp. 1910–1913, 2018.

[42] Y. Zhang, H. Herodotou, and J. Yang, "RIOT: I/O-Efficient Numerical Computing without SQL," *CoRR*, vol. abs/0909.1766, 2009.

[43] O. Zienkiewicz, R. Taylor, and D. Fox, "The Finite Element Method for Solid and Structural Mechanics," in *The Finite Element Method for Solid and Structural Mechanics*, 7th ed. Oxford: Butterworth-Heinemann, 2014.

[44] M. Zukowski and P. Boncz, "From x100 to Vectorwise: Opportunities, Challenges and Things Most Researchers Do Not Think About," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 861–862.

## APPENDIX A: CREATION OF MATRICES AND VECTORS

The following C++ code has been used for the creation of matrices and vectors for all of the presented experiments. Both methods `create_dense_matrix(int argc, char *argv[])` and `create_random_sparse_bandmatrix(int argc, char *argv[])` create csv-files using the established database schemas (1) presented in Section 4.1. In our experiments we have chosen a maximum value of $l = \max(n, 750000)$ for the bandwidth $l$ in order to loosely simulate the neighborhood of states/nodes, interpreting matrices as adjacency matrices of directed graphs or transition matrices of markov chains [29].

```
#include <iostream>
#include <fstream>
#include <ctime>
#include <sstream>
#include <algorithm>
#include <random>

using namespace std;

template<typename T>string to_string(const T& t)
{
        std::stringstream ss;
        ss << t;
        return ss.str();
}

int create_dense_matrix(int argc, char *argv[])
{
        if (argc < 3) {
                cout << "Not␣enough␣input␣arguments" << endl
                << "Input␣needed:" << endl
                << "1.␣matrix␣dimension" << endl
                << "2.␣file_name" << endl;

                return(-1);
        }

        double lv = -1.0, mv = 1.0;
        long   n = atol(argv[1]);

        srand((unsigned)time(0));
        ofstream fileDB((string(argv[2]) + ".csv").c_str());

        for (int ig = 0; ig < n; ig++) {
                for (int jg = 0; jg < n; jg++) {
                        double val = lv + (mv - lv) * ((double)rand() / RAND_MAX);
                        fileDB << 1 + ig << ",␣" << 1 + jg << ",␣" << val << endl;
                }
        }

        fileDB.close();
        return(0);
}

int create_random_sparse_bandmatrix(int argc, char *argv[])
{
        if (argc < 5) {
                cout << "Not␣enough␣input␣arguments" << endl
                << "Input␣needed:" << endl
                << "1.␣matrix␣dimension" << endl
                << "2.␣file_name" << endl
                << "3.␣rows␣per␣file" << endl
                << "4.␣elements␣per␣row" << endl;

                return(-1);
        }
```

```
        // -------------------------
        // ---- Variables
        // -------------------------

        double  lower_value_bound = -1.0,
        upper_value_bound = 1.0;
        string file_prefix = string(argv[2]) + "_";
        long n = atol(argv[1]),                 // matrix dimension
        rows_per_file = min(n, atol(argv[3])),  // elements per file
        l = (3 * rows_per_file) / 4,            // bandwwidth
        elements_per_row = atol(argv[4]),       // elements per row
        nofiles = n / rows_per_file;            // number of files

        if ((2 * l) <= elements_per_row) {
                cout << "bandwidth too low";
                return -2;
        }

        if (n % elements_per_row != 0) nofiles++;
        srand((unsigned)time(0));               // random seed

        // ------------------------
        // ---- Calculation / Writing
        // ------------------------

        for (long number_of_file = 0; number_of_file < nofiles; number_of_file++) {
                ofstream fileDB((file_prefix +
                to_string<int>(number_of_file) + ".csv").c_str());

                long start_row = number_of_file * rows_per_file;

                for (long i = 1 + start_row; i <= min(start_row + rows_per_file, n); i++) {
                        vector<int> columns;
                        columns.push_back(i);

                        fileDB << i << ", " << i << ", "
                        << lower_value_bound + (upper_value_bound -
                        lower_value_bound) * ((double)rand() / RAND_MAX) << endl;

                        for (long j = 0; j < elements_per_row; j++) {
                                long new_column;

                                while (true) {
                                        new_column = static_cast<long>
                                                (i + pow(-1, rand() % 2) * (rand() % l));
                                        new_column = min(n, max((long)1, new_column));

                                        if (find(columns.begin(), columns.end(),
                                                new_column) == columns.end()) {
                                                columns.push_back(new_column);
                                                break;
                                        }
                                }

                                fileDB << i << ", " << new_column << ", " << lower_value_bound
                                        + (upper_value_bound - lower_value_bound)
                                        * ((double)rand() / RAND_MAX) << endl;
                        }
                }
                fileDB.close();
        }
        cout << endl;
        return(0);
}
```

## AUTHOR BIOGRAPHIES

**Dennis Marten** received his M.Sc. in Mathematics with emphasis in numerical analysis in 2012 at Rostock University. After two years of work in the wind energy industry he started as a doctoral student in 2014 at Rostock University. After being funded by the German Research Foundation (DFG), Graduate School 1424 for the first 3 years of his doctoral studies, he currently is a member of the Database Research Group at Rostock University.

**Holger Meyer**, born 1962 in Neuhaus/Elbe (Germany), studied Technical Cybernetics at University of Rostock from 1982 to 1986. He got his PhD in Computer Science from University of Rostock in 1990. Since 1986 he is member of the Micro Systems and since 1992 the Database Research Group at University of Rostock. His research interests include transaction and query processing for non-standard applications on top of object-relational database systems. These applications encompasses graph processing within Digital Archives as well as long-term preservation in Digital Humanities projects. Holger Meyer is member IEEE, ACM, and DHd (Digital Humanities Germany).

**Daniel Dietrich**, born 1991 in Greifswald (Germany), studies Computer Science at the University of Rostock since 2014. He is also self-employed since 2013 and especially interested in Database and Information Systems, Privacy and IT-Security. Daniel Dietrich is an active Member, among others, of the Chaos Computer Club (CCC e.V.), the Club der Ehemaligen der Deutschen SchülerAkademien (CdE e.V.) and the Mathematik Olympiaden e.V.

**Andreas Heuer**, born 1958 in Uelzen (Germany), studied Mathematics and Computer Science at the Technical University of Clausthal from 1978 to 1984. He got his PhD and Habilitation at the TU Clausthal in 1988 and 1993, resp. Since 1994, he is full professor for Database and Information Systems at the University of Rostock. Andreas Heuer is interested in fundamentals of database models and languages, object-oriented databases and digital libraries, in database support for assistive systems, and in big data analytics, here especially in the four “P”: performance, privacy, preservation and provenance.