TECHNISCHE UNIVERSITÄT DRESDEN

Density-Aware Linear Algebra in a Column-Oriented In-Memory Database System

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Dipl.-Phys. David Kernert
geboren am 02. April 1987 in Karlsruhe

Gutachter:

Prof. Dr. Wolfgang Lehner
Technische Universität Dresden
Fakultät Informatik, Institut für Systemarchitektur
01062 Dresden

Prof. Dr. Peter Fischer
Albert-Ludwigs-Universität Freiburg
Technische Fakultät, Institut für Informatik
79110 Freiburg im Breisgau

Tag der Verteidigung: 07. Juli 2016

ABSTRACT

Linear algebra operations appear in nearly every application in advanced analytics, machine learning, and of various science domains. Until today, many data analysts and scientists tend to use statistics software packages or hand-crafted solutions for their analysis. In the era of data deluge, however, the external statistics packages and custom analysis programs that often run on single-workstations are incapable to keep up with the vast increase in data volume and size. Meanwhile, there are scalable platforms for machine learning that provide linear algebra, but in a read-only mode with static data structures. However, they are often poorly suited for the increasing demand of scientists for large scale data manipulation, orchestration, and in particular, advanced data management capabilities. These are among the key features of a mature relational database management system (DBMS), which have proved their capability of large scale data management over the last decades, but lack a deep integration of linear algebra functionality. With the rise of main-memory database systems there is a trend towards online analytical platforms, which target at serving complex analytical queries of any type. It now has become feasible to also consider applications that built up on linear algebra.

This thesis presents a deep integration of linear algebra operations into an in-memory column-oriented database system. In particular, this work shows that it has become feasible to execute linear algebra queries on large data sets directly in a DBMS-integrated engine (Lapeg), without the need of transferring data and being restricted by hard disc latencies. From various application examples that are cited in this work, we deduce a number of requirements that are relevant for a database system that includes linear algebra functionality. Beside the deep integration of matrices and numerical algorithms, these include optimization of expressions, transparent matrix handling, scalability and data-parallelism, and data manipulation capabilities.

These requirements are addressed by our linear algebra engine, which has a multi-layered architecture. In particular, the core contributions of this thesis are: firstly, we show that the columnar storage layer of an in-memory DBMS yields an easy adoption of efficient sparse matrix data types. Furthermore, we present how linear algebra operations, in particular multiplications, are physically implemented based on the presented matrix formats. In the logical layer of the engine, we show that the execution of linear algebra expressions significantly benefits from different techniques that are inspired from database technology. In a novel way, we implemented several of these optimization strategies in LapegâĂŹs optimizer (SpMachO), which generates an optimal execution plan for sparse matrix chain multiplications, and outperforms linear algebra runtimes like R or Matlab by orders of magnitude. An important component of the optimizer is the advanced density estimation method (SpProdest) for an efficient prediction of the matrix density of intermediate results. Moreover, we present the adaptive matrix data type AT Matrix that obviates the requirement of scientists to select appropriate matrix representations, and provides transparent matrix operations. AT Matrix is a topology-aware data structure internally consisting of sparse and dense matrix tiles of variable size. We show that the performance of matrix multiplications benefits from cache locality and runtime optimization by dynamic tile conversions (ATmult). The tiled substructure improves the saturation of the different sockets of a multi-core main-memory platform, reaching up to a speed-up of 10x compared to alternative approaches. Finally, a major part of this thesis is devoted to the topic of data manipulation: we propose a matrix manipulation API that includes insertions and

deletions of single elements, as well as matrix rows, columns, and regions, which is not covered by most linear algebra frameworks. Therefore, we present different mutable matrix types that enable fast updates via a read-optimized main, and a write-optimized delta part. Since our structures are embedded into the DBMS-managed column-oriented storage layer, several transactional features are inherited from the DBMS. In addition, the indirection layer of the novel AT Matrix data type yields efficient deletion of arbitrary matrix sub-ranges, and enables a speed-up of more than 2x for a dynamic matrix workload. We evaluated the Lapeg regarding all of the presented aspects, by using different static and dynamic matrix workloads, where the matrices are either synthetic or real-world matrices from various domains.

With the contributions presented in this thesis, we finally conclude that our DBMS-integrated linear algebra engine is well-suited to serve as a basis for an efficient, scalable in-memory platform to process dynamic, large matrix workloads. By the combining data management capabilities with efficient linear algebra support, this work tackles some of the most pressing needs of data-intensive science in the era of ever increasing data amounts in science and industry.

ACKNOWLEDGEMENTS

This thesis would have not been possible without many people who accompanied me along my way as a PhD student.

First and foremost, I would like to thank my supervisor Wolfgang Lehner for giving me the unique opportunity to realize this research project. Wolfgang always acted as a great advisor and contributed to many inspirations by sharing his technological experience. He supported and motivated me steadily in pursuing my PhD over the last three years, even in the hard times of doubt and struggle I had gone through in the beginning.

Furthermore, I express my gratitude to many colleagues at SAP SE in Walldorf, without whom this work would not have been possible. First of all, I thank Franz FÃd'rber for realizing this project from the SAP side, and for letting me pursue my doctorate within the database department. I also thank Frank KÃűhler, who always shared productive ideas from the engineering perspective in valuable coffee corner meeting sessions. Then there is Arne Schwarz, head of the SAP HANA campus, who made a real team out of our PhD student group, provided us with the opportunity to supervise students, and contributed a lot to make the HANA campus to become a convenient research environment – thank you! There are many more people I want to acknowledge for making my thesis project possible – for example Michael Hladik and Yorck Leschber, who initiated interesting cooperation projects with physics groups of universities in Nantes and Darmstadt. In particular, I am very grateful to the group of Prof. Roth from the Institute of Theoretical Nuclear Physics at the TU Darmstadt for giving us insights into their domain-specific problem setting.

Then I would like to express a deep thanks to Prof. Peter Fischer for taking over the role as co-referee of this thesis, and the fruitful discussions we had during my welcoming visit in Freiburg.

My deepest gratitude goes to my friends and colleagues who proof-read my work and gave me valuable feedback. In particular, I thank Thomas Bach, Ingo MÃijller, Ismail Oukid, Iraklis Psaroudakis, Michael Rudolf, Hannes Voigt, and Florian Wolf for reviewing parts of this thesis. A special appreciation is dedicated to Paul Turner who endured proof-reading and spell-checking the full thesis document. I am also very thankful to Norman May for giving me useful and rich feedback for my research work over the last years.

My time as a doctoral student would not have been as productive without the companionship of my fellow PhD colleagues and campus members in Walldorf, whom I had several rewarding conversations with, and after all, also a lot of fun: Thomas, Robert, Jonathan, Philipp G., Matthias, Martin, Ingo, Ismail, Marcus, Iraklis, Hannes, Michael, Francesc, Elena, Florian and Max – thank you for the great time! I particularly thank my office mate Michael for being a great support for all kinds of technical and non-technical problems that occurred in everyday life. Special thanks goes also to my student Philipp Schulz for valuable contributions to the Lapeg prototype. Moreover, I thank the colleagues in Dresden, who always gave me a warm welcome at the database chair, and supported me with helpful feedback: Ahmad, Katrin, Patrick, Lars D., Julian, Gunnar, Ulrike, Dirk, Martin, Claudio, Kai, Juliana, Tobias, Tomas, Lars K., Tim, Thomas, Elvis, Till, Alexander, Johannes, Kasun, Vasileios, Maik, Annett, Robert and Benjamin. A special thanks goes to Hannes for helping me out with several administrative topics related to my PhD.

Last but not least, I am deeply grateful to my family and my friends who have always been supporting me in good and in hard times of my PhD, and never lost confidence in me. Thank you!

ABBREVIATIONS

ACID Atomicity, Consistency, Isolation and Durability

API Application Programming Interface

ATMATRIX Adaptive Tile Matrix

ATMULT Adaptive Tile Matrix Multiplication

BLAS Basic Linear Algebra Subprograms

COO Coordinate Matrix Format

CPU Central Processing Unit

CSC Compressed Sparse Column

CSR Compressed Sparse Row

DAG Direct Acyclic Graph

DBMS Database Management System

FLOP Floating Point Operation

GEMM General Matrix Multiply

GPU Graphics Processing Unit

HDFS Hadoop Distributed File System

LAPACK Linear Algebra Package (Anderson et al. (1999))

LAPEG Linear Algebra Processing Engine

LLC Last Level Cache

MVCC Multiversion Concurrency Control

NUMA Non-Uniform Memory Access

RAM Random Access Memory

RDD Resilient Distributed Dataset

RDBMS Relational DBMS

ScaLAPACK Scalable LAPACK (Blackford et al. (1997))

SIMD Single Instruction Multiple Data

SMT Simultaneous Multithreading

SpGEMM Sparse GEMM

SpMACHO Sparse Matrix Chain Optimizer

SpPRODEST Sparse Product Estimator

SQL Structured Query Language

SVD Singular Value Decomposition

UDF User Defined Function

SYMBOLS

Symbol	Meaning	Derivation
$\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$	Matrices	
$\boldsymbol{x}, \boldsymbol{y}$	Vectors	
m, k, n	Matrix dimensions	
N_{nz}	Number of non-zero elements	
α, β, γ	Constant parameters	
ρ	Scalar matrix population density	$\rho = \frac{N_{\text{nz}}}{m \times n}$
$\hat{\rho}_A$	Estimated scalar population density of matrix $\boldsymbol{A}$	
$\boldsymbol{\rho}$	Matrix population density map	$(\boldsymbol{\rho})_{ij} = \frac{N_{\text{nz}}^{(ij)}}{m_i \times n_i}$
$\hat{\boldsymbol{\rho}}_C$	Estimated density map of matrix $\boldsymbol{C}$	
$\rho_0^{R/W}$	Read/Write density threshold	
$\mathcal{S}_{d/sp}$	Byte size of a single matrix element in the dense/sparse representation	
b_{atomic}	Fixed logical block size (atomic)	
$\tau_{\text{max}}^{d/sp}$	Maximum physical size of dense/sparse tiles	
Δ	Fraction of matrix elements accommodated by delta structure	
Δ_T	Relative delta size threshold	

CONTENTS

1 Introduction 13

1.1 Challenges and Requirements . . . 16

1.2 Contributions . . . 18

1.3 Outline . . . 20

2 Applications and Systems 21

2.1 Preliminaries . . . 21

2.2 Energy States of a Nucleus . . . 22

2.3 Other Application Scenarios . . . 26

2.3.1 Analyzing Text Documents . . . 26

2.3.2 Graph Analysis . . . 29

2.4 Computational Linear Algebra . . . 31

2.4.1 Basic Linear Algebra Subprograms . . . 31

2.4.2 LAPACK and ScaLAPACK . . . 31

2.4.3 Sparse BLAS . . . 32

2.4.4 HPC Algorithms . . . 33

2.5 Languages and Software Toolkits . . . 34

2.6 Linear Algebra in Databases . . . 36

2.6.1 Linear Algebra with SQL . . . 36

2.6.2 Array DBMS . . . 37

2.6.3 MapReduce and Hadoop-based Systems . . . 38

2.6.4 Commercial Systems and R-Integrations . . . 40

2.7 Summary . . . 42

3 Matrix Storage and Operations 45

3.1 Main-Memory Column-Store . . . 45

3.2 Arrays in a Column-Store DBMS . . . 47
3.2.1 A Straw Man's Method. . . . 47
3.2.2 Storage Engine Prerequisites . . . 48
3.3 Vector Data Types . . . 49
3.3.1 Dense Vectors . . . 49
3.3.2 Sparse Vectors . . . 50
3.4 Matrix Data Types . . . 50
3.4.1 Dense Matrices . . . 50
3.4.2 Sparse Matrices . . . 52
3.4.3 Leveraging CSR to Index Relational Tables . . . 54
3.4.4 Impact of the Linearization Order . . . 56
3.5 Matrix-Vector Multiplication . . . 57
3.5.1 Inner Product Formulation . . . 58
3.5.2 Outer Product Formulation . . . 59
3.6 Matrix-Matrix Multiplication . . . 60
3.6.1 Dense Matrix-Matrix Multiplication . . . 61
3.6.2 Sparse Matrix-Matrix Multiplication . . . 62
3.6.3 Mixed Matrix Multiplication . . . 65
3.7 Parallelization . . . 66
3.8 Summary . . . 68
4 Expression Optimization 69
4.1 Motivation . . . 69
4.2 Expression Optimization . . . 71
4.2.1 Density-Agnostic Matrix Chain Optimization . . . 72
4.2.2 Density-Aware Matrix Chain Optimization . . . 73
4.2.3 Sparse Matrix Chain Optimizer . . . 76
4.3 Execution Time Cost Model . . . 77
4.4 Density Estimation . . . 82
4.4.1 Scalar Density . . . 82
4.4.2 Estimation Errors . . . 83
4.4.3 Density Map . . . 84
4.4.4 Matrix Disorder Measures . . . 85
4.4.5 Sparse Product Density Estimator . . . 86

4.5 Evaluation . . . 87
4.5.1 Accuracy of the Density Estimate . . . 87
4.5.2 Plan Ranking . . . 89
4.5.3 Performance Comparison . . . 90
4.6 Related Work . . . 97
4.6.1 Optimizing Linear Algebra Expressions . . . 97
4.6.2 Sparse Matrix Chain Multiplication and Density Estimation . . . 99
4.6.3 Join Optimization . . . 100
4.7 Summary . . . 102
5 Adaptive Matrices 103
5.1 Motivation . . . 103
5.2 Adaptive Tile Matrix . . . 104
5.2.1 Choosing the Right Representation . . . 104
5.2.2 Adaptive Tiles . . . 105
5.2.3 Recursive Matrix Partitioning . . . 108
5.2.4 Internal Structure . . . 112
5.3 Matrix Multiplication . . . 114
5.3.1 Multiplication Kernels . . . 116
5.3.2 Referenced Submatrix Multiplications . . . 116
5.3.3 Dynamic Multiplication Optimizer . . . 117
5.3.4 Density Estimation . . . 117
5.3.5 Memory Resource Flexibility . . . 118
5.4 Parallelization . . . 119
5.4.1 Scheduling Strategies . . . 120
5.4.2 NUMA-aware Partitioning . . . 122
5.5 Evaluation . . . 122
5.5.1 Experiment Setup . . . 123
5.5.2 Partitioning Costs . . . 123
5.5.3 Performance Comparison . . . 124
5.5.4 Runtime Optimizations . . . 127
5.5.5 Impact of Single Optimization Steps . . . 128
5.6 Related Work . . . 129
5.6.1 Matrix Representations in Numerical Libraries . . . 129
5.6.2 Parallel Sparse Matrix Multiplication . . . 130

5.6.3 Tiled Arrays and Matrices . . . 131
5.7 Summary . . . 133
6 Updates and Mutability 135
6.1 Motivation . . . 135
6.2 Application Programming Interface . . . 137
6.2.1 Matrix Processing Language . . . 137
6.2.2 Matrix Access Patterns . . . 137
6.2.3 Data Manipulation Primitives . . . 138
6.2.4 Implementation of the Nuclear Physics Analysis . . . 140
6.2.5 Language and UDFs . . . 141
6.3 Updatable Matrix Architecture . . . 142
6.3.1 The Cost of Updates . . . 142
6.3.2 Mutable Sparse Matrix . . . 145
6.3.3 Delta-Aware Operators . . . 149
6.4 Mutable Adaptive Tile Matrix . . . 151
6.5 Evaluation . . . 155
6.5.1 Experiment Setup . . . 155
6.5.2 Read-Query Response Times . . . 157
6.5.3 Delta Size Experiment . . . 158
6.5.4 Throughput Measurement . . . 159
6.5.5 Deletions of Rows and Columns . . . 162
6.6 Related Work . . . 164
6.6.1 Update Strategies in Databases . . . 164
6.6.2 Mutable Sparse Matrices . . . 165
6.7 Summary . . . 166
7 Conclusion and Outlook 169
A Tables 173

1 INTRODUCTION

DATA ANALYTICS SPEAKS LINEAR ALGEBRA

The data deluge in science and industry ushered a new era in the ways scientific research is conducted, and how business decisions are made. As Gray (2007) put it, data-intensive science has become the "fourth paradigm" of scientific exploration. A lot of data is produced by scientific simulations and collected by detector experiments. In fact, for many of today's large scientific projects the cost spend on developing and maintaining software for data processing dominates the capital expenditure (Hey et al., 2009). Similarly, a wide variety of enterprises spent considerable efforts on accumulating "big data" in order to get valuable insights and improve their businesses (McGuire et al., 2012). Regardless of the domain, the vast amount of captured data has to be curated (Miller, 2013), and most importantly, it needs to be *analyzed*.

Whereas developing complex analysis models for their data has been part of scientists' everyday life for decades, the increasing demand of advanced statistical data analysis in business environments has brought forth a new job description, called the *data scientist* (Davenport and Patil, 2012). Indeed, the requirements of a data scientist overlap largely with the skill set scientists have acquired in research institutions, which is why most data scientists in industry have a scientific background in mathematics, computer science, or natural sciences such as physics. In the following, we use the terms *scientist* and *data scientist* interchangeably for denoting users that *analyze data by using sophisticated mathematical or statistical algorithms*.

Apart from the high performance computing clusters that are built for giant scientific projects or by the top-level data companies such as Google and Facebook, the majority of final data analysis actually takes place at the work desk of users of small research institutions and companies (Hey et al., 2009). In fact, many scientists of various domains are facing restrictions of computational nature when they try to extract knowledge from their data. While data set sizes are steadily growing, the algorithms and scripts used to analyze them fail to scale to the same pace. This problem applies to a wide range of users, who often use highly customized handwritten code, or poorly scalable software packages to perform their analysis.

The big data movement of the recent decade has created many database systems that are capable of processing enormous data sets in a highly parallelized way, and often within the main memory of large machines. However, the absence of an appropriate language interface and the mathematical

functionality that is required in most analytical applications makes common databases unsuitable for data scientists (Stonebraker et al., 2007).

While speaking about the algorithms of scientists, one of the first questions arising is which semantic language they use to express their analysis queries. Following the brief survey below, one can quickly conclude that the majority of algorithms are conceptually written in terms of linear algebra. Most scientists have a strong mathematical background and predominantly think of matrices and vectors when they model their data (Cohen et al., 2009). Hence, linear algebra operations form the fundamental building blocks of many algorithms. Some examples will be described in greater detail later in this thesis.

In general, linear algebra applications on large data sets are very diverse. In the environment of scientific research institutions, there are for example nuclear physicists (e.g. Roth, 2009) who perform quantum mechanical computations to determine the energy states of a nucleus. Quantum mechanics is described by linear algebra, and thus, the calculation of energy states is essentially computing eigenvalues of a large, sparse matrix. Another emerging, and very data-intensive domain is the field of bioinformatics. In genomics (e.g. Moret et al., 2001; Li et al., 2001), matrix operations are used to study evolutionary relationships among groups of organisms. Users from the domain of mechanical engineering (e.g. Teranishi et al., 2008) determine the thermal behavior and mechanical deformations of a three-dimensional car model. They often use finite element methods, which involve iterative solving of a large, sparse matrix equation as core computation. In the medical sector, researchers cluster gene expressions into groups (e.g. Liu et al., 2013; Kluger et al., 2003) and search for correlations between cancer occurrences and the respective genotypes. Clustering algorithms can in general be expressed by means of vector space operations, and consequently, by operations on matrices and vectors. A recent work by Hahmann et al. (2014) describes how multiple clustering algorithms are broken down into several matrix processing steps, e.g. determining the similarity matrix.

In the business world, data scientists mostly implement statistical algorithms to acquire knowledge from various data sources. Typical examples originate from the large field of data mining, and include regression analysis or principal component analysis of economic data. Some users come from the financial sector (e.g. Rebonato and JÃd'ckel, 1999), market research (e.g. Constantin, 2014), or customer relations. In the age of the World Wide Web, social networks and a massively growing online retail market, data science has significantly increased in importance, and accordingly the demand for linear algebra operations on large and dynamic data sets. For example, recommendation machines (Sarwar et al., 2002) and text mining algorithms (Xu et al., 2003) make intensive use of singular value decomposition (SVD), usually of large, sparse matrices. Moreover, many analytics operations on graphs, such as on social networks (Skillicorn, 2005), can be expressed in terms of linear algebra (Kepner and Gilbert, 2011). To name an example, the famous Page Rank (Page et al., 1998) algorithm is merely a modified matrix decomposition, implemented via iterative matrix-vector multiplications of the adjacency matrix.

Having outlined the importance of linear algebra in various fields, it is evident that supporting matrix operations is a substantial requirement for any analytical system. This passes over to the discussion on what these applications have in common with conventional database systems. It is needless to say that efficient, purely number-crunching linear algebra algorithms have been developed for decades. Nevertheless, there is an increasing number of limitations adhering to traditional

approaches: most of the above mentioned applications are processed in work flows that are often highly customized – that is, many scientists read data from comma separated value (CSV)- or binary files, and process it using static hand-written scripts or numeric programs. For scientists, this comes with an enormous maintenance overhead, since storing data in raw files is suboptimal regarding both persistence and data provenance. Furthermore, the even greater challenge is to develop and maintain algorithms that scale to arbitrarily large data sets, and are regularly adapted and tuned to utilize recent hardware features. Most importantly, this approach is way to static for exploratory, human-driven data science. In a world of high data volumes and fast data changes, scientists need the capability to ad-hoc analyze their data that is constantly up-to-date, without copying and reloading the complete data set on every change.

These were among the reasons that have driven scientists of several domains to glimpse into a universe, where everything is about storing, updating, and processing data: the universe of database management systems.

In the 1990s, applications from astronomical image processing initiated the development of the array database systems, such as RasDaMan (Baumann et al., 1998), and the more recent SciDB (Brown, 2010). Array DBMSs were built because there was indeed the requirement of large scale data management, but also a mismatch between the demands of the scientists and conventional relational database management systems. To be more precise, the relational set-based data layout of traditional DBMS was not well-suited for the strictly structured, spacial data that requires multidimensional locality for processing. Relational database systems were at that time not known as "data-crunching" machines for large scale processing – the disk-based data management and the granting of ACID[1] features often led to a rather complex design that incurred a negative impact on processing performance. However, advances in hardware as well as several architectural and technical innovations have also significantly improved the performance of relational database systems. For instance, the decreased price and increased capacity of random access memory (RAM) over the last few years laid the foundation for the shift from disk-centric to main-memory-centric data management and processing, which resulted in a considerable performance boost of analytical queries on large data sets (Garcia-Molina and Salem, 1992). Moreover, the introduction of a column-oriented database design (Boncz et al., 2005; Abadi et al., 2005) has shown performance advantages on analytical workloads in contrast to conventional row-oriented approaches. These advances made state-of-the art DBMS interesting for science domains again: for example, the ProteomicsDB (ProteomicsDB, n.d.; Wilhelm et al., 2014) stores terabytes of proteomes, and utilizes fast column searches to provide sub-second search results. With efficient aggregation and join operators, modern relational DBMS (RDBMS) can also be used for particle physics analysis (Kernert et al., 2015b).

Despite the advances in relational database systems, most conventional DBMS do not support linear algebra natively. This is not only an insurmountable barrier to the market of many science applications for DBMS vendors, but also an obstacle for data scientists in business environments, where data already resides in a relational DBMS. In such a setup, we have observed two different workarounds to implement linear algebra functionality that are equally unfavorable: either the data is processed in statistical or mathematical frameworks, such as R or MATLAB, or some mathematical operations, such as matrix multiplications, are mutilated into complex structured query language

[1] atomicity, consistency, isolation and durability (ACID) are desired properties of a DBMS

(SQL) expressions. As mentioned in the beginning, the former approach incurs the disadvantage of a prior extract-transform-load (ETL) process, in addition to the fact that common statistics software environments do not scale to very large data sets. The latter approach of using relational expressions for linear algebra is not only unnatural for users, but also set-based relational operators are poorly suited for matrix processing. The gap between data-parallel systems and the provisioning of linear algebra functionality has also been addressed by Stonebraker et al. (2013b), who sees "an emerging need for complex analytics on massive data sets. In fact, most of the underlying algorithms are expressed as sequences of linear algebra operations on matrices [...] the relational model is a poor fit to this class of problems." The claims are backed by benchmark studies (Stonebraker et al., 2007) that reveal poor performance of RDBMS for matrix processing, in particular matrix multiplications.

To summarize, we identify database-integrated support of linear algebra as the key to overcome the divergence between what relational database systems and data warehouses offer, and the demands of many science and analytics applications. Hence, the initial motivation of this work is the deep integration of a linear algebra runtime in a relational DBMS, in particular into an in-memory column-store. However, in the course of this work, we elaborated another aspect: we show that a lot of techniques from database management systems can in a novel way be used to improve the execution performance of linear algebra queries. These techniques include join optimization, cardinality estimation, dynamic rewrites, to name just a few. The overlaps of optimizing relational queries and linear algebra expressions are manifold. Consequently, we identified and implemented optimization approaches inspired from database technologies to bridge the gap between the relational world and linear algebra processing.

1.1 CHALLENGES AND REQUIREMENTS

By taking a glimpse into the notebooks of data scientists, one recognizes that linear algebra expressions mostly contain the following data objects: scalars, vectors and matrices. As a matter of fact, a system that provides linear algebra functionality should support matrices as *first class citizens*, in the same way as tables are the primary data objects in a relational DBMS. Moreover, the corresponding matrix operations, such as matrix additions, matrix-matrix and matrix-vector multiplications are the respective top level operations – similar to what joins and aggregations are to an RDBMS, to complete the analogy.

In order to achieve our goal of a linear algebra-supporting system, we define several requirements that are crucial for a system that accommodates matrices and vectors as primary data objects, and additions and multiplications as primary data operations.

Expression optimization. The exposure of a linear algebra interface requires more than the implementation of a few primitives, such as matrix multiplications or additions. Just like other descriptive languages, such as SQL, linear algebra expressions are often composed of multiple operations and operands. A prominent example for such a composite operation is matrix chain multiplication. It occurs for example in Markov chains (Yegnanarayanan, 2013), the transfer matrix method (Suzuki, 2003), linear discrete dynamical systems (Feng, 2002), or in randomized SVD methods (Tulloch, 2014). In practice, the user is often let alone with the execution order of the expression when more than two matrices are involved. Although the mathematical characteristics

of this operation reveal significant optimization opportunities, standard numerical linear algebra systems only provide a naive, often non-optimal execution of matrix chain multiplications. Many of the large matrices that appear in scientific data sets are sparse, which makes an optimal execution of a matrix chain multiplication even more challenging. When matrices are fully populated, the cost of any intermediate matrix multiplication can be precomputed since it solely depends on matrix dimensions. In contrast, the cost of sparse multiplications depend on the number of non-zero elements which is not priorly known for intermediate results. Moreover, in mathematical libraries based on the popular basic linear algebra subprograms (BLAS, Dongarra et al., 1990) interface, users have to pick one out of several dozens of different algorithm implementations from an application programming interface (API) that is difficult to comprehend. However, the average data scientist does not have a profound knowledge about the performance differences among the algorithms, which often have a complex runtime behavior. For example, when a matrix is populated to only about 20%, dense multiplication algorithms have a better performance than the corresponding sparse algorithms. How is the user supposed to know the density threshold value for picking the dense kernel? Therefore, expression optimization inevitably requires a density estimation for intermediate results.

Transparency. A favorable characteristic of database systems is the abstraction of the logical standardized language interface in the front-end, and the physical implementation of data structures and algorithms in the back-end. For instance, a table in a relational database system could be stored following a row-oriented or a column-oriented layout, and could further be compressed using various compression methods. However, the logical interface and the functionality exposed to the user does not depend on the concrete implementation, hence, it is *transparent* to the user. In a very similar way, the decoupling of the physical representation from the logical interface should hold for matrices. Matrix elements can be linearized in a row-major or column-major order, and there is a multitude of different storage representations for sparse matrices. Unfortunately, such an abstraction does mostly not exist for linear algebra. In most numerical frameworks and statistical toolkits, such as R or MATLAB, the user explicitly defines the data representation by explicitly creating an instance of either a full or a sparse matrix type. On the contrary, an ideal interface to the user would be to provide a single logical matrix data type – once the data is physically loaded, the system decides which data representation is chosen internally, optimized to the hardware configuration.

Scalability and performance. The requirement of scalability and efficiency is substantial for a linear algebra runtime, since the main bottleneck of many applications in scientific environments and data science is mostly computations on large matrices. Many algorithms, such as the various implementations of BLAS, have already been developed, efficiently implemented and tuned for decades. The high performance community and several hardware vendors frequently publish new implementations of common linear algebra primitives on the latest hardware architectures. It is common sense that scientists, who either use hand-written programs for their analysis, or common numerical frameworks like R or MATLAB, are unlikely to switch to any other system, if they expect a significant degradation in performance. Consequently, the aim derived from this requirement is to provide a system that has at least equal performance compared with specialized libraries on a given hardware.

Data manipulation. Unlike common intuition, matrices are no static objects. They often contain *dynamic* data in a way where entries are changed, inserted or removed. Modifications are applied to single elements or to complete rows, columns, or rectangular submatrices. As an example, we refer to an application from nuclear physics, which will be explained in more detail in Section 2.2. Therein, a technique is used that iteratively interleaves linear algebra calculations with matrix modifications to refine the obtained result. These modifications include deletions and insertions of matrix rows and columns of a large, sparse matrix. Another example is a network graph, which is under constant change: new friends connection are established in a social network, and roads are added to a street network. These operations correspond to insertions or deletions of elements, rows, or columns in the corresponding adjacency matrix. When data becomes extremely big, ETL processes are unfeasible. Thus, dynamic data is a dilemma for every user who relies on linear algebra libraries and frameworks, as they are mostly based on static files and structures.

1.2 CONTRIBUTIONS

To address the aforementioned requirements, this thesis presents an architectural design for a linear algebra processing engine (Lapeg). The multi-layered concept of the engine is depicted in Figure 1.1. The architecture is conceptually divided into the following components: a language interface for linear algebra, a logical layer that contains optimization mechanisms to rewrite linear algebra expressions, and a physical layer which contains different data structures and algorithms.

In particular, the core contributions of this thesis are:

Survey of applications and systems. We sketch three example applications from the science and business world, and without loss of generality, focus on a reduced number of linear algebra operations that are of great importance. From these applications we also derive several demands of the scientists, and survey the field of linear algebra solutions with respect to the identified requirements. In particular, we discuss state-of-the art libraries and established frameworks that have been used by data scientists for many years. Finally, we review recent literature on the integration of linear algebra functionality in database systems and scalable data processing frameworks.

Columnar data structures and algorithms. We describe the integration of fundamental sparse and dense matrix types into the column-oriented storage layer of an in-memory system, and how they are exposed as relational database tables. Furthermore, we show how the internal representation of relational tables in a column-store can benefit from indexed sparse matrix structures. Based on the columnar matrix representations, we introduce basic linear algebra algorithms such as matrix-vector and matrix-matrix multiplication, which are essential for many applications. In this context, we also discuss strategies for parallelizing matrix multiplication in shared memory. Some parts of this contribution have been developed jointly with Wolfgang Lehner and Frank Koehler, and were published at the SSDBM'14 conference (Kernert et al., 2014).

Expression optimization. We consider linear algebra expressions that contain multiplications of more than two matrices, and how their execution can be optimized. In particular, we present a sparse optimizer that builds an optimal execution plan for matrix chain multiplications (SpMachO),

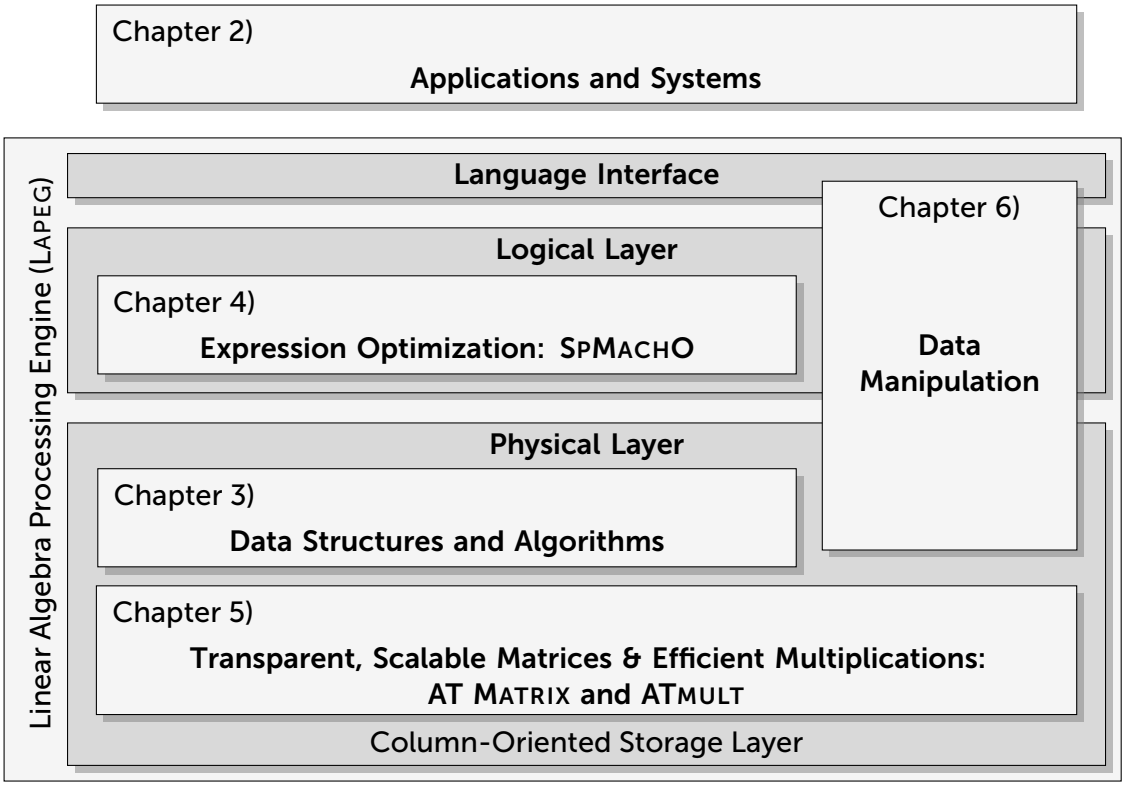


Figure 1.1: Organization of the thesis and contributions. The core part is formed by the linear algebra processing engine (LAPEG), which contains multiple architectural components.

which uses a sophisticated cost model derived from algorithmic access patterns and automatically fitted to the underlying hardware configuration. As an important pillar of the optimizer, we further present SPPRODEST, an efficient and novel technique to estimate the densities of intermediate result matrices. The techniques we are using are inspired from database technologies such as cardinality estimation, join enumeration, and cost models. We evaluate the quality of the optimizer, and compare the execution runtime of the generated expressions against alternative approaches, including R and MATLAB. Parts of the optimizer have been developed jointly with Wolfgang Lehner and Frank Koehler, and were published at the EDBT'15 conference (Kernert et al., 2015a).

Transparent, adaptive matrices. We combine parts of the previous contributions by presenting a novel topology-aware, heterogeneous data type (AT MATRIX) for big data matrices that contains an adaptive layout with dense and sparse tiles. Moreover, we introduce a multiplication operator (ATMULT) that uses dynamic optimizations based on density estimates and cost functions of kernel algorithms. Based on the multiplication operator, we discuss different parallelization approaches to scale up on multi-core main-memory machines, including scheduling and partitioning strategies. We evaluate and compare ATMULT against different alternative approaches, and show that ATMULT is able to outperform state-of-the-art methods by several factors. Some of the presented ideas have

been developed jointly with Wolfgang Lehner and Frank Koehler, and were published at the ICDE'16 conference (Kernert et al., 2016).

Mutable matrices. We describe a manipulation application interface for matrices that includes typical manipulation commands on arbitrary shaped, multidimensional subregions of matrices. Furthermore, we inspect different matrix data structures with respect to their updatability, and show how we implement updates and data manipulations on a sparse matrix. In particular, we employ a main-delta approach by leveraging the native updatable table column container in the column-oriented storage layer of the DBMS. Moreover, we present two different ways to integrate mutability into the heterogeneous AT Matrix data type, and provide a comprehensive evaluation of our proposed data structures using different data manipulation workflows. Some parts of this contribution have been developed jointly with Wolfgang Lehner and Frank Koehler, and were published at the SSDBM'14 conference (Kernert et al., 2014).

In this thesis, the components of the logical and physical layer are paramount, since the topic of language interface for linear algebra is to our notion already covered by the prevailing numeric frameworks, such as R, and Matlab. Consequently, apart from the manipulation commands on API level, the language interface will not be the main focus. Furthermore, due to the embedding of the Lapeg in a column-oriented main-memory database system, the physical design of data structures for matrices and vectors builds up on the column-oriented storage layer. Nonetheless, most of the components described in this work are of general nature and work independently of the underlying storage layer.

1.3 OUTLINE

The organization of the thesis is embedded in Figure 1.1 and can be summarized as follows.

Chapter 2 contains the introduction of example applications, and a survey of existing work on processing and managing linear algebra workflows. Following Chapter 2 we step through the components of the architecture that are highlighted in Figure 1.1. In particular, the columnar matrix data structures and fundamental algorithms of the Lapeg are introduced in Chapter 3. Many parts of this chapter can be considered as building blocks for the following components. Chapter 4 is about the optimization of linear algebra expression, with the focus on matrix chain multiplications. In Chapter 5 we describe the adaptive matrix type AT Matrix and the tiled multiplication operator ATmult. Chapter 6 is devoted to the topic of data manipulation.

The thesis is finalized by the conclusion and outlook in Chapter 7.

2 APPLICATIONS AND SYSTEMS

As outlined in the introduction, linear algebra is used as the mathematical model to approach various problems in science and data analytics. As a matter of fact, many efforts have been made to perform linear algebra calculations efficiently on computer systems. In this chapter, we give a more detailed description of three linear algebra applications from different domains. Thereafter, we survey state-of-the-art libraries for linear algebra and statistical software frameworks. Moreover, we present related literature about recent approaches to integrate linear algebra functionality in database systems and big data frameworks. Finally, we discuss their applicability to our example workflows, especially with respect to the requirements stated in Chapter 1.

2.1 PRELIMINARIES

First, we briefly introduce the mathematical foundation and the matrix notation that is used consistently throughout this thesis.

Matrices and vectors are printed in bold letters, whereas scalars and single elements are printed in regular letters. The fundamental linear algebra operations are written as follows:

- The product of an $m \times n$ matrix $\boldsymbol{A}$ with an n-dimensional vector $\boldsymbol{v}$ is denoted as

$$\boldsymbol{v}' = \boldsymbol{A}\boldsymbol{v}, \tag{2.1}$$

  where the elements v'_i of the m-dimensional result vector $\boldsymbol{v}'$ are calculated as:

$$v'_i = \sum_{j=1}^{n} a_{ij} v_j, \quad i \in \{1..m\}. \tag{2.2}$$

- The *inner* (scalar) product of two n-dimensional vectors $\boldsymbol{v}, \boldsymbol{w}$ is denoted as:

$$\boldsymbol{v} \cdot \boldsymbol{w} = \boldsymbol{v}^T \boldsymbol{w} = \sum_{j=1}^{n} v_j w_j, \tag{2.3}$$

  where $\boldsymbol{v}^T$ is a row vector, i.e. the *transposed* of the column vector $\boldsymbol{v}$.

- The matrix multiplication of an $m \times k$ matrix $\boldsymbol{A}$ with a $k \times n$ matrix $\boldsymbol{B}$ is written as

$$\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B}, \tag{2.4}$$

  where each element c_{ij} is defined as

$$c_{ij} = \sum_{l=1}^{k} a_{il} b_{lj} = \boldsymbol{a}_{i*} \boldsymbol{b}_{*j}, \;\; i \in \{1..m\}, j \in \{1..n\}, \tag{2.5}$$

  using the inner product formulation. $\boldsymbol{a}_{i*}$ is the i-th *row* of $\boldsymbol{A}$, and $\boldsymbol{b}_{*j}$ the j-th *column* of $\boldsymbol{B}$.
- A matrix transposition of a $m \times n$ matrix $\boldsymbol{A}$

$$(\cdot)^T : \boldsymbol{A} \rightarrow \boldsymbol{A}^T \tag{2.6}$$

  turns matrix rows $\boldsymbol{a}_{i*}$ into columns $\boldsymbol{a}_{*i}$ (and vice versa), such that $a_{ij} \rightarrow a_{ji}$. Thus, the dimensionality of $\boldsymbol{A}^T$ is $n \times m$.
- The Euclidean norm of a vector $\boldsymbol{v}$ is determined via its dot product by

$$\|\boldsymbol{v}\| = \sqrt{\boldsymbol{v} \cdot \boldsymbol{v}} = \sqrt{\sum_{i=1}^{n} v_i^2}. \tag{2.7}$$

- The determinant of a square $n \times n$ matrix $\boldsymbol{A}$ can be recursively defined:

$$det(\boldsymbol{A}) = \sum_{j=1}^{n} (-1)^{j+1} a_{1j} det(\boldsymbol{A}_{1j}), \tag{2.8}$$

  where $\boldsymbol{A}_{1j}$ is the $(n-1) \times (n-1)$ submatrix of $\boldsymbol{A}$, which is obtained by deleting the first row and the j-th column of $\boldsymbol{A}$.

2.2 ENERGY STATES OF A NUCLEUS

To provide a better insight into typical application scenarios that use linear algebra, we sketch multiple example workflows from different domains. The first one is taken from nuclear science, and is a part of a cooperation project with SAP SE and the nuclear physics group of TU Darmstadt. The aim of the application is to determine the energy states of a nucleus in the so-called *no-core shell model* (NCSM) (Roth, 2009). In the theory of quantum mechanics, the energetic states of a nucleus are encoded in the *Hamiltonian operator*, which is represented as a symmetric matrix $\boldsymbol{H}$ (Griffiths, 2005). Here, the matrix $\boldsymbol{H}$ stems from a preprocessed nuclear physics simulation, and is usually large and sparse. A small example instance of $\boldsymbol{H}$ is displayed Fig. 2.1. The simulation can be scaled up arbitrarily, and the matrices of higher dimensions tend to be sparser than that of Fig. 2.1 (other example instances are listed in Table A.1). The physicists are usually interested in the equilibrium energy states of the nucleus, which are derived by calculating the *eigenvalues* of $\boldsymbol{H}$. For the sake of clarity, we skip further domain-specific details about the analysis, and focus on the mathematical and computational aspects of this application. In the context of this cooperation project, we received matrix data from Prof. Roth and his students. In return, we investigated how their analytical workflow can be integrated in the scope of a main-memory database system.

Eigenvalues. Formally, an eigenvalue λ of a square matrix $\boldsymbol{A}$ is a value that fulfills the equation

$$\boldsymbol{A}\boldsymbol{u} = \lambda\boldsymbol{u} \tag{2.9}$$

$$\Leftrightarrow (\boldsymbol{A} - \lambda\boldsymbol{I})\boldsymbol{u} = 0, \tag{2.10}$$

where $\boldsymbol{u}$ are the corresponding eigenvectors. The eigenvalues can be determined exactly by solving the equation $det(\boldsymbol{A} - \lambda\boldsymbol{I}) = 0$.

In practice, the exact determination of the eigenvalues is numerically not feasible due to the high dimension of $\boldsymbol{H}$. Furthermore, the scientists are mostly interested in either only the highest or the lowest eigenvalues of the matrix. Therefore, it is common to use methods that iteratively approximate the most dominant eigenvalues. In our example, the physicists use the Lanczos algorithm, which is a method to determine the k lowest eigenvalues of an arbitrary symmetric square matrix.

Algorithm 1 Lanczos algorithm

1: **function** LANCZOS(Hamiltonian Matrix $\boldsymbol{H}$)
2: $\quad\boldsymbol{v}_1 \leftarrow$ random vector with norm 1
3: $\quad\boldsymbol{v}_0 \leftarrow \boldsymbol{0}$
4: $\quad\beta_1 \leftarrow 0$
5: $\quad$**for** $1 \leq j < k$ **do**
6: $\quad\quad\boldsymbol{w}_j^* \leftarrow \boldsymbol{H}\boldsymbol{v}_j$
7: $\quad\quad\alpha_j \leftarrow \boldsymbol{w}_j^* \cdot \boldsymbol{v}_j$
8: $\quad\quad\boldsymbol{w}_j \leftarrow \boldsymbol{w}_j^* - \alpha_j\boldsymbol{v}_j - \beta_j\boldsymbol{v}_{j-1}$
9: $\quad\quad\beta_{j+1} \leftarrow \|\boldsymbol{w}_j\|$
10: $\quad\quad\boldsymbol{v}_{j+1} \leftarrow \boldsymbol{w}_j/\beta_j$
11: $\quad\boldsymbol{w}_k^* \leftarrow \boldsymbol{H}\boldsymbol{v}_k$
12: $\quad\alpha_k \leftarrow \boldsymbol{w}_k^* \cdot \boldsymbol{v}_k$
13: $\quad$Construct tridiagonal matrix $\boldsymbol{T}$
14: $\quad\boldsymbol{V} \leftarrow \{\boldsymbol{v}_1 \cdots \boldsymbol{v}_k\}$
15: $\quad$**return** $\boldsymbol{T}, \boldsymbol{V}$

Algorithm 1 sketches the Lanczos algorithm as described in the book of Saad (2011). It contains several linear algebra expressions using the mathematical notation as described above. The return value of the algorithm is the (small) tridiagonal $k \times k$ matrix that is created in line 13:

$$\boldsymbol{T} = \begin{pmatrix} \alpha_1 & \beta_2 & & & & 0 \\ \beta_2 & \alpha_2 & \beta_3 & & & \\ & \beta_3 & \alpha_3 & \ddots & & \\ & & \ddots & \ddots & \beta_{k-1} & \\ & & & \beta_{k-1} & \alpha_{k-1} & \beta_k \\ 0 & & & & \beta_k & \alpha_k \end{pmatrix}.$$

The k lowest eigenvalues of the large matrix $\boldsymbol{H}$ are equivalent to the set of eigenvalues of the much smaller matrix $\boldsymbol{T}$. All eigenvectors and eigenvalues of $\boldsymbol{T}$ can be determined in $\mathcal{O}(k^3)$ time using

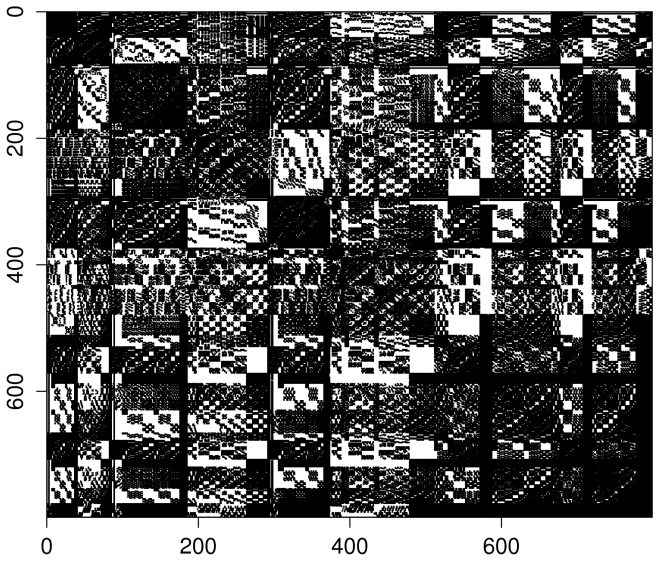


Figure 2.1: A small exemplary matrix of the NCSM simulation. Regions populated with non-zero elements are drawn black, white areas correspond to non-populated (zero) regions.

a conventional solver, such as the QR-method (Francis, 1961, 1962). We refer readers that are interested in more details about solving linear systems to common literature of numerical linear algebra, e.g. the book of Demmel (1997).

As outlined before, the reason for using the Lanczos algorithm instead of a conventional, direct solver is the infeasibility of applying an exact solver on an extremely large sparse matrix. Besides the cubical scaling CPU complexity, applying the QR-method on the $n \times n$ matrix $\boldsymbol{H}$ would require to save the full $\boldsymbol{Q}$-matrix, which has the same dimensions $(n \times n)$ as the initial matrix $\boldsymbol{H}$. As a result, the memory consumption of $\mathcal{O}(n^2)$ would exceed that of the original sparse matrix $\boldsymbol{H}$ by orders of magnitude, and potentially hit the system memory limit.

In contrast, the Lanczos method belongs to the *Krylov* subspace methods, whose general objective is to project the large vector space onto a smaller space: the Krylov subspace (Saad, 2011). In other words, the large matrix problem is reduced to a smaller one that can then be tackled using conventional methods. This is a fairly common technique, and one of the reasons why sparse matrix-vector multiplication is of such a great importance. Besides the Lanczos algorithm, there are further, similar methods, such as the Arnoldi iteration, and several variations (e.g. implicitly restarted methods, Calvetti et al., 1994; Benner and Fassbender, 1997; Wu and Simon, 2000), some of which have been implemented in freely available software libraries like ARPACK (R. B. Lehoucq, 1997). Nevertheless, many scientists – including the physicists we cooperated with – prefer to hand-code the iterations of Algorithm 1, in order to be able to manually adapt and tune the algorithm.

The pseudo-code in Algorithm 1 is written in an abstract linear algebra language that preserves the mathematical semantics, while abstracting the implementation details of fundamental operations such as the matrix-vector multiplication. The set of linear algebra expressions is the natural

way data scientists with a mathematical background query their data: vectors and matrices are first-class citizens, and multiplications, additions, and subtractions are top-level operations. In our referred use case, and for the Lanczos algorithm in general, the iterative multiplication of the large, sparse Hamiltonian matrix $\boldsymbol{H}$ with vector $\boldsymbol{v}_j$ (line 6) exhibits a computational bottleneck. As a consequence, scientists have put a considerable effort in developing an efficient implementation for sparse-matrix vector multiplication, which led to numerous publications in the high performance community (e.g., Schubert et al., 2011). Thus, the demand for time efficiency underlines our requirement of **scalable and efficient algorithms**.

Importance Truncation

An improved version of the energy state analysis proposed by Roth (2009) foresees the *truncation* of certain quantum states. A quantum basis state i refers to the i-th row of $\boldsymbol{H}$, and due to the symmetry, also to the i-th column. Hence, the truncation effectively translates into a deletion of matrix rows and columns. The method proposed by Roth (2009) is an iterative procedure: starting from the initial, full Hamiltonian matrix, a reference subset $\mathcal{M}_{\text{ref}}$ is formed via truncation. The reference subset corresponds to a submatrix of $\boldsymbol{H}$, and it is assumed that eigenvalues and eigenvectors $\boldsymbol{v}_{\text{ref}}$ of $\mathcal{M}_{\text{ref}}$ are already known from previous calculations. Then, the eigenvectors $\boldsymbol{v}_{\text{ref}}$ of the reference problem can be used as a starting point for the solution of the large problem. By utilizing methods from perturbation theory, corrections to $\boldsymbol{v}_{\text{ref}}$ can be derived in the form of additional basis states:

$$\boldsymbol{v}'_{\text{ref}} \leftarrow \boldsymbol{v}_{\text{ref}} + \sum_{i \notin \mathcal{M}_{\text{ref}}} c_i \boldsymbol{u}_i \tag{2.11}$$

Each basis state $\boldsymbol{u}_i$ of the weighted sum in Equation (2.11) refers to a row-column pair of the full matrix $\boldsymbol{H}$ that is not yet contained in the submatrix of $\mathcal{M}_{\text{ref}}$. However, only the states that have a weight c_i that exceeds a certain threshold value will be considered in the next iteration. In particular, these rows and columns are added to the matrix, transferring $\mathcal{M}_{\text{ref}}$ into the modified matrix $\mathcal{M}'_{\text{ref}}$. Then, the Lanczos method is applied to obtain the eigenvectors to the lowest eigenvalues of $\mathcal{M}'_{\text{ref}}$, which will serve as input vectors for the next iteration. The procedure is repeated until a convergence criteria is met. The pseudocode of this method is sketched in Algorithm 2.

Algorithm 2 Importance Truncation Method

```
1:  function importanceTruncation(Matrix H)
2:      Load n × n Hamiltonian matrix H
3:      Form M_ref by deleting rows/columns of H
4:      v_ref ← v_0
5:      while not converged do
6:          Derive relevant basis states U = {u_l..u_p} from the perturbation of v_ref
7:          Add U as rows and columns to the matrix: M_ref ← M_ref + U
8:          Solve eigenvalue problem: lanczos(M_ref)
9:          Set v_ref to the first eigenvector of M_ref
10:     return Solution for eigenvalue problem of full matrix H
```

An important conclusion that can be drawn from this workflow is that matrices are no static objects. The matrix $\mathcal{M}_{\text{ref}}$ results from a manipulation of the large matrix $\boldsymbol{H}$, and is iteratively growing over the iterations. In contrast to a common misconception, this example shows that numeric matrices are indeed manipulated before and after taking part in a mathematical operation. In this iterative procedure, the matrix mutations are interleaved with multiplication operations. Furthermore, we observe that not single element updates are performed, but rather complete rows, columns, row blocks, column blocks, or windows. As a result, we deduce the requirement that we mentioned in the introduction: there is a need for **data manipulation capabilities** for a system that processes analytical queries on matrices, such as the importance truncation method.

We will refer to the energy analysis again as an example for the integration of mutable matrix workflows in Chapter 6. Nevertheless, the Lanczos method, and iterative eigenvalue routines in general are not restricted to the field of physics. As a matter of fact, eigenvalue calculations play an important role in other domains, e.g. in spectral graph partitioning (Pothen et al., 1990), or the page rank algorithm (Page et al., 1998).

2.3 OTHER APPLICATION SCENARIOS

The nuclear physics application is our main example use case. Nevertheless, as indicated in the introduction, there is a variety of other application scenarios for linear algebra. We cite two further applications that are addressed by the LAPEG, and are based on example use cases in the environment of the SAP HANA analytics platform (Färber et al., 2015).

2.3.1 Analyzing Text Documents

Due to the ever growing number of digitized articles and reports, there is an increasing demand for powerful tools to analyze the sheer amount of textual documents. In the era of web mining and knowledge discovery in databases, text analysis has significantly gained in importance. Consequently, these applications are considered to be part of an own discipline called *text mining*. The analysis we sketch here lean on the text mining features of the SAP HANA database (Färber et al., 2015): similarity analysis and document clustering.

Consider the relation $\mathcal{R}$ that connects terms with the documents they appear in: $\langle$*Document*, *Term*$\rangle$. The latter is a m-to-n relation, as each document usually contains multiple terms, and most terms appear in multiple documents. As a matter of fact, one term can be contained multiple times in a single document. Consequently, the occurrence of each term per document can be counted to create the relation $\mathcal{R}'$: $\langle$*Document*, *Term*, *Frequency*$\rangle$. The transformation $\mathcal{R} \rightarrow \mathcal{R}'$ can be easily performed by relational means with a count aggregation. In fact, $\mathcal{R}'$ can be represented as a matrix $\boldsymbol{A}$, where *Document* refers to a matrix row, *Term* to a matrix column, and *Frequency* to the value of the corresponding matrix element. The matrix $\boldsymbol{A}$ is commonly known as the *document-term matrix*, or *term-document matrix*, depending on the orientation of $\boldsymbol{A}$. In practice, the value of the matrix element is the *tf-idf* (term frequency · inverse document frequency) measure rather than the plain term frequency (Jones, 1972; Salton and Buckley, 1988; Manning et al., 2008). The measure is

defined as

$$tf_{ij} \cdot idf_i = \frac{f_{ij}}{\max_j\{f_{ij}\}} \cdot \log \frac{N}{n_j} \equiv (\boldsymbol{A})_{ij}, \tag{2.12}$$

where f_{ij} is the frequency of term j in document i, N is the number of documents in the corpus, and n_j denotes in how many documents of the corpus a term j occurs, hence $1 \leq n_j \leq N$. The measure is known to be more meaningful to describe the weight of terms in documents. In particular, the *idf* (inverse document frequency) suppresses terms that appear in nearly every document, and thus are less useful for the semantic separation of documents.

Given the matrix $\boldsymbol{A}$, the challenge is to extract meaningful information. We start with a simple example, the cosine similarity: each row $\boldsymbol{a}_{p*}$ can be interpreted as a feature vector of a document p in the multidimensional term space. A measure for the similarity between two documents p and q is the angle θ of their feature vectors, which is determined by the dot product:

$$\cos\theta_{pq} = \frac{\boldsymbol{a}_{p*} \cdot \boldsymbol{a}_{q*}}{\|\boldsymbol{a}_{p*}\|\|\boldsymbol{a}_{q*}\|}. \tag{2.13}$$

Moreover, let $\tilde{\boldsymbol{A}}$ be the matrix that results from $\boldsymbol{A}$ by normalizing each row using the Euclidean norm as of equation (2.7), such that $\tilde{\boldsymbol{a}}_{i*} = \boldsymbol{a}_{i*}/\|\boldsymbol{a}_{i*}\|$. Then, the matrix multiplication

$$\boldsymbol{D} = \tilde{\boldsymbol{A}}\tilde{\boldsymbol{A}}^T \tag{2.14}$$

yields the symmetric cosine similarity matrix $\boldsymbol{D}$. The matrix elements $(\boldsymbol{D})_{pq}$ identify with the pair-wise distances of the corresponding document pairs (p, q) as defined in Equation (2.13). In order to get the k most similar documents to any given document p, one can just fetch the corresponding vector $\boldsymbol{d}_{p*}$ of $\boldsymbol{D}$ and rank its values. It is worth mentioning that the matrix multiplication in equation (2.14) is a very compute-intensive operation, since the matrix can be very large. To name an example, the term-document matrix of Wikipedia has dimensions of 1.5 million $\times$ 14 million, with almost a billion non-zero elements. The multiplication of such a matrix produces a very large result that might exceed the system's memory resources. We address the problem of result size estimation and resource management in Chapters 4 and 5.

Besides the cosine similarity, a more comprehensive analysis of the document is often desired, e.g., clustering documents and classifying them into domains. A typical approach for the latter analysis is a hierarchical clustering on the distance matrix of Equation 2.14, which returns groups of documents that have a high pair-wise similarity. Another way is described by Borko and Bernick (1963), who proposed a factor analysis of the term correlation matrix $\boldsymbol{D}_T = \tilde{\boldsymbol{A}}^T\tilde{\boldsymbol{A}}$, which is based on the principal component analysis (PCA) of $\boldsymbol{D}_T$.

More recently, promising methods to retrieve the associations between documents and terms include latent semantic indexing (LSI), as proposed in the work of Deerwester et al. (1990), or non-negative matrix factorizations (Xu et al., 2003). These approaches are inherently similar, since they are both based on a factorization of the document-term matrix $\tilde{\boldsymbol{A}}$. In brief, the LSI method effectively relies on the singular value decomposition (SVD) of matrix $\tilde{\boldsymbol{A}}$, which maps the documents into a "semantic" space:

$$\tilde{\boldsymbol{A}} \rightarrow \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\boldsymbol{T}} \tag{2.15}$$

The vectors $\boldsymbol{v}_{*j}$ of matrix $\boldsymbol{V}$ can be interpreted as linear independent basis vectors of the semantic space, each composed of a weighted combination of terms. In the literature, the *latent semantic vectors* are commonly interpreted as base topics (Xu et al., 2003). Moreover, they identify with the factors resulting from the one-mode factor analysis (Deerwester et al., 1990). Matrix $\boldsymbol{U}$ contains the coefficients u_{ij} that connect each document i with a linear combination of semantic vectors. The diagonal matrix $\boldsymbol{\Sigma}$ contains the singular values, which imposes an additional weight on the most dominant semantic vectors.

Once the SVD of matrix $\tilde{\boldsymbol{A}}$ is determined, the eigenvector decomposition of $\boldsymbol{D}$ and $\boldsymbol{D}_T$ is obtained by

$$\boldsymbol{D} = \tilde{\boldsymbol{A}}\tilde{\boldsymbol{A}}^T = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\boldsymbol{T}}\boldsymbol{V}\boldsymbol{\Sigma}\boldsymbol{U}^{\boldsymbol{T}} = \boldsymbol{U}\boldsymbol{\Sigma}^{\boldsymbol{2}}\boldsymbol{U}^{\boldsymbol{T}}, \tag{2.16}$$

$$\boldsymbol{D}_T = \tilde{\boldsymbol{A}}^T\tilde{\boldsymbol{A}} = \boldsymbol{V}\boldsymbol{\Sigma}\boldsymbol{U}^{\boldsymbol{T}}\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\boldsymbol{T}} = \boldsymbol{V}\boldsymbol{\Sigma}^{\boldsymbol{2}}\boldsymbol{V}^{\boldsymbol{T}}. \tag{2.17}$$

Instead of the exact SVD, often only a low-rank approximation of the SVD is computed by utilizing dimension-reducing methods. Consequently, the similarity matrix $\boldsymbol{D}$ and its approximate eigenvalue decomposition can be determined via the approximate SVD, while avoiding the costly full multiplication of Equation (2.14).

Randomized Methods.

The exact SVD (or QR) factorization of a $m \times n$ matrix can be computed with double-precision accuracy with $\mathcal{O}(mn \min\{m, n\})$ flops (Halko et al., 2011). However, the term-document matrix that is analyzed can be extremely large, and an exact factorization is often not feasible with reasonable effort. In most cases such large matrices have a numerical rank k that is substantially smaller than m and n. Therefore, it is possible to produce a low-rank decomposition that approximates the matrix sufficiently well, e.g. by using *randomized* methods. The latter are described in recent numerical algebra literature (Rokhlin et al., 2010; Halko et al., 2011; Martinsson et al., 2011). As a matter of fact, methods based on singular value decomposition are not limited to text analysis. SVD is used in a variety of applications, e.g., it is the core component of the Netflix movie recommendation algorithm (Salakhutdinov et al., 2007). Randomized SVD methods have also been used by Facebook to analyze hashtag-word co-occurrences (Tulloch, 2014).

Algorithm 3 A randomized method for low-rank approximate matrix factorization (from Halko et al. (2011)).

1: **function** RANDOMIZEDSVD(Matrix $\boldsymbol{A}$)
2: Generate an $n \times k$ random matrix $\boldsymbol{\Omega}$
3: Form $\boldsymbol{Y} \leftarrow (\boldsymbol{A}\boldsymbol{A}^T)^q\boldsymbol{A}\boldsymbol{\Omega}$
4: Construct matrix $\boldsymbol{Q}$ as orthonomal basis for the range of $\boldsymbol{Y}$
5: Form $\boldsymbol{B} \leftarrow \boldsymbol{Q}^T\boldsymbol{A}$
6: Compute SVD of the small matrix: $\boldsymbol{B} \leftarrow \tilde{\boldsymbol{U}}\boldsymbol{\Sigma}\boldsymbol{V}^{\boldsymbol{T}}$
7: Set $\boldsymbol{U} \leftarrow \boldsymbol{Q}\tilde{\boldsymbol{U}}$
8: **return** $\boldsymbol{U}, \boldsymbol{\Sigma}, \boldsymbol{V}$

Algorithm 3 shows the common randomized SVD as presented by Halko et al. (2011). The exponent q in line 2 affects the accuracy of the approximation, and is usually chosen between one and two. Nonetheless, the particular implementation of randomized SVD as of Algorithm 3 might deviate for each individual analysis. Similar to our first example, the randomized SVD method comprises several linear algebra expressions. Data scientists want to hand-write code similar to Algorithm 3, either to try completely new methods or to apply variations to their scripts. For instance, additional operations that manipulate the matrices could be added, e.g. to improve the stability of the algorithm.

In practice, most of the computation time is spent in the matrix chain multiplication in line 2 (Huang et al., 2013), which involves the very large, sparse matrix $\boldsymbol{A}$ and the dense, and rather rectangular matrix $\boldsymbol{\Omega}$. Hence, we detect another key requirement that was mentioned in the introduction: the system that executes Algorithm 3 should evaluate the **expression** in line 2 in an **optimized**, time-efficient way.

2.3.2 Graph Analysis

Another emerging field in data science and knowledge discovery is the analysis of large data graphs. There is a plethora of different real world examples for large graphs. In times of web mining and social media, the graphs that are probably most in demand are social or purchasing networks. Examples are the social circles of Facebook, Twitter, or the Amazon product co-purchasing graph. The University of Stanford assembled these graphs and other examples in the publicly available SNAP data set (Leskovec and Krevl, 2014).

Breadth-first search.

Consider a social network graph $\mathcal{G} = (V, E)$ where the vertices V correspond to members and the edges E correspond to friendship connections between members. A common query on a social networks is: *'who are the friends, and friends of the friends of person v_1?'*, which is an inclusive breadth-first search (BFS) with depth two, starting from node v_1.

Due to the dualism between a graph and its adjacency matrix, many graph algorithms can be expressed in terms of linear algebra (Kepner and Gilbert, 2011). Usually the adjacency matrix $\boldsymbol{G}$ of the graph $\mathcal{G}$ is large and very sparse. Each row of the adjacency matrix $(\boldsymbol{G})_{i*}$ contains the adjacencies of the vertex i as non-zero elements. Consequently, the breadth-first search on a graph corresponds to a (Boolean) matrix-vector multiplication on the adjacency matrix $\boldsymbol{G}$. In this thesis we restrict our description to this graph algorithm only. For further details about the relation between graphs and matrices, we refer to the work of Kepner and Gilbert (2011), which gives a more comprehensive insight into how to utilize linear algebra operations for different graph algorithms.

We call a breadth-first search *inclusive* when it returns all vertices that are discovered on paths that have a length up to a certain traversal depth. In order to obtain an *exclusive* BFS, the discussed algorithm can be rewritten in a way that only vertices that are reachable via paths of a certain length are returned.

Algorithm 4 shows the breadth-first algorithm. The algorithm starts with a conversion of the start vertex into a vector $\boldsymbol{x}$, which is then multiplied with the matrix iteratively in a loop (line 6). Finally, the resulting vector $\boldsymbol{x}$ is converted back into a result set $\mathcal{P}$ of found vertices. Mathematically

Algorithm 4 Inclusive Breadth-First Search

1: **procedure** BFSEARCH(Adjacency Matrix $\boldsymbol{G}$, VertexID $start$, unsigned $depth$)
2: $\mathcal{P} \leftarrow 0$ ▷ $\mathcal{P}$: result set
3: $\boldsymbol{x} \leftarrow 0$ ▷ Initialize $n \times 1$ vector
4: $(\boldsymbol{x})_{\text{start}} \leftarrow 1$ ▷ Set vector element at start ID position
5: **while** $depth > 0$ **do** ▷ Level-synchronous BFS iteration
6: $\boldsymbol{x}^T \leftarrow \boldsymbol{x}^{\boldsymbol{T}}\boldsymbol{G}$
7: $depth \leftarrow depth - 1$
8: **for** $0 \leq i < n$ **do**
9: **if** $(\boldsymbol{x})_{\text{i}} \neq 0$ **then** ▷ Assemble all non-zero elements in $\boldsymbol{x}$
10: $\mathcal{P} \leftarrow i$
11: **return** $\mathcal{P}$ ▷ Return found vertices

speaking the x_i values are multiplied with the sparse matrix values $(\boldsymbol{G})_{ij}$, which usually refer to the edge weights. In fact, a social network graph does not necessarily have weighted edges, hence, in this case the BFS is fully expressed by a Boolean matrix-vector multiplication. Nevertheless, the wide class of linear algebra algorithms is written and optimized for floating point values. Thus, in the case of unweighted graph edges the floating point matrix values of $\boldsymbol{G}$ might just be filled with non-zero dummy values, e.g. 1.0.

Besides breadth-first search, there are further, more complex graph analysis queries that make use of sparse linear algebra. We already mentioned spectral graph partitioning (Pothen et al., 1990), and the page rank algorithm (Page et al., 1998), which both can be expressed as eigenvalue calculation (and consequently, they can be implemented using the Lanczos method we described in Section 2.2.) In contrast to the Boolean matrix multiplication for BFS, these workflows actually work with weighted edges, and hence, make use of the floating point values in $\boldsymbol{G}$. Nevertheless, we again refer to related work for more details about these algorithms. In this work, we focus on the conceptual requirements for the linear algebra processing platform to process such workflows.

Particularly, an online social network graph is growing over time as more and more connections are established. In matrix terms the dynamic growth corresponds to inserting non-zero elements into the sparse matrix, or from a more abstract point of view, setting certain matrix elements from zero to a non-zero value. Consequently, deletions of connections are essentially removals of the corresponding non-zero matrix elements. When new users are added, additional vertices are included in the graph. Since the social network graph is undirected, adding a vertex means inserting a symmetric row-column pair to the adjacency matrix $\boldsymbol{G}$. This can be done by appending the new row $(\boldsymbol{G})_{(n+1)*}$ to the bottom, and the column $(\boldsymbol{G})_{*(n+1)}$ to the right side of matrix $\boldsymbol{G}$, respectively.

The key requirements that we infer from the matrix-based analysis of a dynamic graph are similar to those of the nuclear physics application scenario: the **efficiency** of the sparse matrix-vector **multiplication** on one hand, and the incremental updatability and **manipulation** of matrix parts on the other hand.

2.4 COMPUTATIONAL LINEAR ALGEBRA

In this section and the following, we present popular libraries and frameworks that are commonly used to perform static matrix calculations.

2.4.1 Basic Linear Algebra Subprograms

As early as in the late 1970s, scientists began to assemble basic linear algebra routines together in a library in order to remedy redundant implementations on every new computer. As a result, the Basic Linear Algebra Subprograms (BLAS) library was created and written in Fortran by Lawson et al. (1979). Over the years, the BLAS Technical Forum was founded and established a standard for the routines (Dongarra et al., 2001). An official reference implementation for standardized BLAS can be downloaded from the Netlib website (BLAS, n.d.).

The BLAS routines are arranged in the following three levels:

1. **Level 1** contains the vector routines described in the initial paper Lawson et al. (1979), for instance, dot products, vector norms, and vector additions.

2. **Level 2** was first presented by Dongarra et al. (1988), and includes matrix-vector operations, e.g. the generalized matrix-vector multiplication (GEMV).

3. **Level 3** was introduced in Dongarra et al. (1990) and adds matrix-matrix operations, such as the general matrix multiply (GEMM).

Besides fundamental addition and multiplication operations, BLAS further contains a solver method for triangular or band diagonal matrix problems.

Since the 1990s, BLAS has become the de-facto interface standard for linear algebra calculations. Nearly every hardware vendor provides an implementation that, in contrast to the reference implementation, is optimized for speed on the corresponding hardware. Examples are the Intel Math Kernel Library (MKL, n.d.), AMD Core Math Library (ACML, n.d.), IBM Engineering and Scientific Subroutine Library (ESSL, n.d.), HP Mathematical Software Library (MLib, n.d.), Sun Performance Library (SPL, n.d.), et cetera. Moreover, there are open source implementations among which the most popular is the Automatically Tuned Linear Algebra Software library (ATLAS, n.d.).

2.4.2 LAPACK and ScaLAPACK

Initially BLAS Level 1 routines were utilized by LINPACK (Dongarra et al., 1979), a software package that offers algorithms for solving dense and banded linear equations as well as linear least squares problems. Later however, LINPACK was superseded by LAPACK (Anderson et al., 1999), which also included solving methods for eigen- and singular value problems. LAPACK makes extensive use of Level 2 and 3 routines, and works with block-partitioned algorithms in order to efficiently exploit the memory hierarchy of modern computers.

Following the development of supercomputers and high performance computing clusters, LAPACK was extended by SCALAPACK (Blackford et al., 1997). SCALAPACK is a distributed memory

implementation of LAPACK, and internally uses LAPACK and BLAS for local computation. Local results of the distributed computation are combined using the basic linear algebra communication subprograms (BLACS) as communication framework. The latter is based on the message passing interface (MPI, n.d.). Both LAPACK and SCALAPACK are also implemented by some of the hardware-vendor-provided math libraries, for instance by Intel MKL (MKL, n.d.).

Recently, there have been further developments to adopt the functionality of SCALAPACK for many-core and heterogeneous architectures (Agullo et al., 2009). The MAGMA project (Tomov et al., 2010; Dongarra et al., 2014) aims at writing a dense linear algebra library similar to LAPACK but for heterogeneous/hybrid architectures, starting with "multi-core CPU+GPU" systems. The PLASMA project (Kurzak et al., 2010) refurbishes the LAPACK and SCALAPACK algorithms in order to fully exploit thread-level parallelism. In particular, mathematical functions are decomposed into a set of *tasks* that operate on small matrix portions (*tiles*), and are arranged in a directed acyclic graph (DAG) plan. This *dataflow model* allows for more flexibility and improves performance on multi-core platforms due to out-of-order execution. Although on a coarser level, the tiled matrix multiplication operator (ATMULT) presented in Chapter 5 also uses a dataflow DAG to schedule asynchronous tile-multiplications.

2.4.3 Sparse BLAS

The standard BLAS routines, as well as the LAPACK and SCALAPACK libraries, are designed for dense and band[1] matrices. In fact, the initial BLAS had no special implementation for sparse matrices. Since this limitation alone would discourage the use of their library for a wide range of applications, the BLAS technical forum began to establish routines for sparse matrices in its BLAS standard specification (Dongarra et al., 2001). The suggested extensions are based on the ideas of Duff et al. (1997). A fundamental difference to the dense BLAS is the abstraction of the sparse storage type in the generic interface. In particular, there is no definition on how the sparse matrix should be laid out in storage. Instead, the interface works with a sparse matrix handle that accommodates a user-defined, custom sparse matrix storage implementation. Another difference to the dense BLAS is the reduced functionality set: there are no matrix operations in which both operands are sparse, e.g. a sparse matrix-sparse matrix multiplication.

The National Institute of Standards and Technology (NIST) provides two versions of the sparse BLAS: a reference implementation that is compliant to the BLAST standard (NISTa, n.d.), and the preceding *original* implementation (NISTb, n.d.). The latter deviates from the standard as of Duff et al. (1997) in the following way: instead of a general function that takes an abstract matrix handle, NISTb (n.d.) provides distinct functions for some of the most common sparse matrix storage types. Indeed, this version seems to be more practical than the official sparse BLAS (NISTa, n.d.) since it is partially implemented by many available BLAS libraries. In contrast, barely any of the vendor-provided libraries implement the more recent standard (Duff et al., 1997; Dongarra et al., 2001) based on sparse matrix handles. For instance, the AMD CML only provides some of the sparse Level 1 routines. IBM ESSL has implementations for several Level 1 and Level 2 methods, albeit with slightly deviating naming conventions. HP MLIB, SPL, and Intel MKL offer sparse implementations

[1] Band matrices have non-zero elements only on the matrix diagonal and secondary diagonals.

for Level 2 and Level 3 routines that are based on the original NIST version (NISTb, n.d.) with predefined sparse matrix storage types.

To our notion the OSKI library by Vuduc et al. (2005) from the University of Berkeley offers the implementation that is closest to the current sparse BLAS standard as of (NISTa, n.d.). Furthermore, the library provides automated tuning, such that the matrix the user provides is analyzed and converted accordingly into the most efficient data structure. There are only few BLAS-based libraries that provide a sparse matrix-sparse matrix multiplication routine (SPGEMM, also called SPMM), in addition to the sparse BLAS specification. An example is Nvidia's CuSparse (n.d.) for GPUs. Nevertheless, the routine is implemented by complementary libraries such as CHOLMOD/SuiteSparse (Chen et al., 2008).

To summarize, the BLAS interface has proven its importance as the state-of-the-art interface for numerical linear algebra. It is implemented by various hardware vendors, and many methods, particularly operations on dense matrices, have been thoroughly tuned and optimized. Consequently, they are hard to outperform from a performance perspective. Nevertheless, most BLAS libraries are written in Fortran or C/C++, thus requiring decent programming skills from their users. Furthermore, the introduction of sparse BLAS makes library implementations even more complex due to the plurality of different sparse matrix storage formats. The intention of the sparse BLAS standard is to target a "a sophisticated user community but not necessarily one that is or needs to be familiar with details of sparse storage schemes" (Dongarra et al., 2001). However, we argue that this aim has not been achieved, since most libraries do not implement the abstraction of the physical sparse matrix structure. In fact, our adaptive matrix type that is described in Chapter 5 implements the decoupling between the logical matrix interface and its physical implementation.

2.4.4 HPC Algorithms

Clearly, SCALAPACK became the state-of-the-art for highly efficient and scalable library routines for dense linear algebra, complemented by the developments of the PLASMA and MAGMA projects. However, the obvious disadvantage of these libraries is that they do not include any methods for sparse matrix computations. Only lately, a few sparse methods has been added to the MAGMA project (Yamazaki et al., 2014). In fact, the runtime of many applications such as the sparse eigenvalue computation described in Section 2.2 heavily depends on the efficiency of the sparse matrix-vector multiplication implementation. As a matter of fact, there are numerous groups in the high performance community that have been tuning these algorithms for scalability on specific hardware setups.

Algorithms for sparse matrix-vector multiplication (SPGEMV, also called SPMV) have been optimized for multi-core processors (e.g., Vuduc and Moon, 2005) as well as for distributed hybrid CPU+GPU environments (e.g., Schubert et al., 2011). Moreover, SPGEMM is another key routine for which there are efficient implementations on multi-core (Patwary et al., 2015), distributed memory (Buluç and Gilbert, 2012), GPU (Dalton et al., 2015), and hybrid CPU+GPU platforms (Matam et al., 2012). Meanwhile, many algorithms of different HPC research groups are assembled and published as libraries, e.g., PETSc (Balay et al., 2015), LAMA (Kraus et al., 2013), GHOST (Kreutzer et al., 2015), PARALUTION (Paralution Labs, 2015).

We mention these algorithms for completeness, while emphasizing that any HPC algorithm or a library call alone is actually not serving the requirements stated in the introduction. Many algo-

rithms might perform well on their targeted platforms, but require specially designed data structures. As mentioned before, these matrix data types are usually transparent and designed for either homogeneous sparse matrices, or specific topologies such as band-diagonal matrices. Moreover, sparse matrices in HPC contexts are mostly static and not designed to be updated in an ad-hoc fashion, which is one of our main requirements. In HPC environments, data is commonly loaded statically from a file, and the algorithms run completely agnostic of any resource-management in the system.

Nonetheless, these algorithms complement our work, and many can be used as multiplication *kernels* in our system to benefit from their performance. For instance, we use a BLAS library method for dense matrix multiplications in the DBMS-integrated LAPEG. In the same way, other algorithms could be used for sparse operations, given that the input format is compatible with that of the LAPEG. Acting in the layer above, our LAPEG optimizer, which is described in Chapter 4, selects a suitable algorithm according to its expected runtime performance. Moreover, we discuss in Chapter 5 how our system supervises the memory resources consumption in accordance with the system's resource management.

Further Library Abstractions

As mentioned before, a major shortcoming for users of BLAS and HPC libraries is the bloated interface of different routine calls, which is dependent on the underlying matrix storage type, matrix shape and orientation, element data type, et cetera. Therefore, some of the mentioned libraries abstract the matrix implementation from the matrix interface. Hence, algorithms can be written independently from the underlying physical matrix format, including dense and sparse matrix types. Examples for such libraries are PETsc, OSKI, LAMA, and the MTL4 (Siek and Lumsdaine, 1998). In particular, LAMA (Förster and Kraus, 2011) uses expression templates to offer a C++-embedded, natural language interface for linear algebra operations such as matrix-vector multiplication.

2.5 LANGUAGES AND SOFTWARE TOOLKITS

Users from math and science environments often do not have a profound C++/Fortran knowledge that is required to use BLAS or HPC libraries natively. Hence, many mathematicians prefer to tackle their linear algebra computations in a more natural and compact way, e.g., in a form comparable to our pseudocode in Algorithm 1. For this reason, most data scientists make use of numerical software toolkits to write and execute their analysis. As a result, various math- and linear algebra-aware languages and runtimes have been developed in the last decades, some of which are still continuously extended.

Matlab

In the late 1970s, Cleve Moler started to develop MATLAB in order to enable his students to use the LINPACK and EISPACK (Garbow, 1974) libraries without needing to know Fortran. A few years later, MATLAB (Mathworks Inc., n.d.) has become a widely used proprietary software for matrix computations. The language of MATLAB is matrix-based: matrices are first-class citizens, and arithmetic

operations, including multiplications or additions of matrices and vectors, are top level-operations. A public adoption of the MATLAB framework is the open source project GNU Octave, which is in large parts compatible with the MATLAB language.

The R Project for Statistical Computing

Besides MATLAB and its related numerical algebra software packages, there were also efforts from the statistics community to create a domain-specific language for statistical computations. As a matter of fact, statistical calculations inherently include linear algebra operations. At the Bell Laboratories, John M. Chambers et. al. developed the proprietary, statistical language S "to turn ideas into software, quickly and faithfully" (Chambers, 1969). For many years the market was dominated by proprietary statistical software packages, including IBM SPSS, SAS, or StatSoft Statistica, to name a few. It was almost two decades later that S was adopted and implemented by the open-source project R (R Foundation, n.d.), which has become one of the most popular programming languages in statistics and science nowadays. Similar to MATLAB, R supports, besides its rich and comprehensive statistical functionality, basic linear algebra operations including first-level data types for matrices and vectors.

Although there are more interpreter-based software packages that are alike, such as Python-based NumPy (Walt et al., 2011) and SciPy (Jones et al., 2001–), we mostly refer to R and MATLAB. They probably offer the most comprehensive functionality, and might have the largest user base, since they are both listed among the top 20 of the most popular programming languages overall [2] according to the generic TIOBE Index (TIOBE, 2016).

Limitations

Both MATLAB and R are mainly interpreted languages, although there is the possibility to deploy precompiled byte- (R) or binary- (MATLAB) code that can be run without the front-end. The runtime and all data objects are held in main memory, which sets a physical limit on scalability. However, with modern server machines offering more than a terabyte of RAM, it has become feasible to load considerable data sets into a MATLAB or R runtime, including large matrices. The data objects, such as matrices, can be saved to and reloaded from binary files on disc, thus offering a minimal persistence functionality. In order to perform dense linear algebra computations, MATLAB and R can be configured to use optimized BLAS and LAPACK libraries internally. In contrast, custom implementations are used for the processing of sparse matrices in MATLAB (Gilbert et al., 1992; Shah and Gilbert, 2005). For R there are several packages that implement the virtual sparse matrix interface, such as the *RMatrix* (Bates, n.d.) or *Slam* (Hornik, n.d.) packages. To our surprise, we observed in our experiments in Chapter 4 that sparse matrix multiplications in both frameworks are sequential, in contrast to the dense counterparts. To the best of our knowledge, both frameworks do not perform any optimization on linear algebra expressions. As our experiments in Chapter 4 further reveal, a matrix chain multiplication is compiled by R or MATLAB into a naive left-deep operator tree. Consequently, the execution of long expressions results in a poor performance since considerable optimization opportunities are missed out.

[2] as of November 2015

2.6 LINEAR ALGEBRA IN DATABASES

In contrast to the data of science and research institutions that is primarily stored in files, the majority of actual data in commercial environments is persisted in relational database systems. For many years, data analytics in the business world was limited to either SQL queries in a relational DBMS or multidimensional data processing using data warehousing systems. However, since sophisticated data mining algorithms require linear algebra functionality, data scientists often had no other option than extracting the data from the DBMS, and loading it into R or MATLAB for analysis. In fact, the increasing demand of advanced analysis on databases has shifted the perspective of the database community towards a stronger support and a tighter coupling of database systems with domain-specific languages, in particular with linear algebra functionality (Stonebraker et al., 2013b; Färber et al., 2015; Cohen et al., 2009). In this context, we briefly describe some systems that have addressed this topic recently.

2.6.1 Linear Algebra with SQL

There are a few approaches that integrate linear algebra into a DBMS by using the *relational* processing and storage engine for numerical computation. Popular works to mention within this field are:

MAD Skills

Cohen et al. (2009) demonstrate how plain SQL can be utilized to calculate linear algebra expressions, referred to as magnetic, agile, deep (MAD) skills. Two ways of storing matrices in a relational DBMS are presented: the first is a $\langle row, column, value \rangle$-table of matrix element triples; the second is to create a table of $\langle row, values[] \rangle$-pairs that store complete matrix rows as numeric arrays. Moreover, they conduct basic operations including matrix transposition and matrix multiplication inside the relational DBMS via SQL and user-defined operators. As an example application, they mention the document cosine similarity analysis, which we also discussed in Section 2.3.1. Their implementation builds on PostgreSQL (Stonebraker, 1990), which is extendable by new data types (Stonebraker, 1986). To support matrix multiplication on their $\langle row, values[] \rangle$ structure, they added a custom infix operator for the element-wise multiplication and addition of numeric array data types.

Listing 2.1: A matrix multiplication expressed in SQL. A and B are represented as $\langle row, col, val \rangle$ triple tables.

```
  select sum(A.val*B.val)
    from A, B
   where A.col = B.row
group by A.row, B.col
```

Listing 2.1 shows the implementation of a matrix multiplication in SQL using the $\langle row, column, value \rangle$-table layout for matrices. The authors admit that "vanilla SQL" terms are "pairing up scalar values, rather than treating vectors as whole objects" (Cohen et al., 2009), and further conclude that

SQL does not match the natural way of thinking of a data scientist with a mathematical background. Moreover, to express linear algebra in SQL, it is required to know the underlying storage representation. In their setup, this could be the triple representation or the dense row representation for matrices, which are both not optimal for every situation. In other words, there is no transparency of the physical representation. Regarding time efficiency, we emphasize that relational operators are not designed for "number crunching" operations like matrix multiplications. In particular, the relational set-based join and aggregation operators are unable to exploit the ordered structure of matrices, which usually leads to a poor performance for linear algebra queries (Stonebraker et al., 2007).

RIOT-DB and RIOT

Another work to mention in this context is RIOT-DB (Zhang et al., 2009), which stands for "R with I/O Transparency." They provide an R package which maps matrix and vector operations to *views* on a relational DBMS back-end. Similar to MAD Skills, they use n-tuple tables as storage representation for multidimensional arrays with $1..n-1$ coordinate columns as primary key, and a value column. The focus of Zhang et al. (2009) is disk I/O-optimization of R expressions by avoiding materialization of intermediate results and deferring computation. Both of these characteristics are basically given out-of-the-box by leveraging database views, since most query processors work iterator-based in a pipelined fashion, and use late materialization strategies. Despite using MySQL as RDBMS back-end, they are "aware of the inadequacy of a relational database system [to run numeric computations] [...] storing array indexes in tables incurs significant storage and processing overhead" (Zhang et al., 2009). Therefore, they suggest a "next generation" RIOT system (Zhang et al., 2009, 2010), which leverages an array store similar to ChunkyStore (Stonebraker et al., 2007) rather than using relational tables, and carries out database-style optimizations. In contrast to MAD Skills, they also propose higher-level inter-operator optimizations of linear algebra expressions. Nonetheless, they discuss a very limited example of dense-only matrix operations, e.g., dense matrix chain multiplications, and optimize on disk I/O rather than in-memory computation time.

2.6.2 Array DBMS

As mentioned in the introduction, the demand of astronomers for a system that is capable of tackling the exploration of their large, array-structured image data sets has driven the development of array-based DBMS in the late 90s. As a consequence, the initial usage of array-DBMS was dominated by these domain-specific applications, in particular multidimensional image processing. Hence, the resulting array-processing languages, such as RasQL for the pioneering array-DBMS RasDaMan (Baumann et al., 1998), ArrayQL (Lim et al., 2012), or SciQL as MonetDB extension (Kersten et al., 2011), were geared towards locality-aware and multidimensional operators (e.g. spatial aggregation) rather than to general linear algebra operations on arrays and matrices. The mismatch between spatial operations on multidimensional arrays and linear algebra functionality is addressed by Stonebraker et al. (2013a). In fact, the recent system SciDB (Brown, 2010) targets to overcome this discrepancy.

SciDB

The distributed, disk-based array-DBMS provides three different programming interfaces, each with a different flavor. Besides an Array Query Language (AQL), and an Array Functional Language (AFL), they offer an R-interface (SciDB-R), and state that "R is by far the most popular statistics package" (Stonebraker et al., 2013b). With SciDB-R, users write scripts in R, while computing-intensive tasks are pushed down to the database, e.g., matrix multiplications. Following the argumentation of Stonebraker, SciDB uses "carefully optimized C++ libraries" for linear algebra calculations, since "it is common pragma that analysis packages like ScaLAPACK [...] are considerably faster and more extensive than what can be freshly coded with ordinary effort" (Stonebraker et al., 2013b). However, the disadvantages of using an external library are the required data conversion into the appropriate representation, and the problem of resource management in this hybrid model. Moreover, the distributed, disk-based DBMS's exhibit poor performance for ad-hoc queries, as well as high latencies and long response times for linear algebra operations in general. Since ScaLAPACK, which is used by SciDB for dense linear algebra, does not include sparse matrix operations, they provide their own implementations for sparse matrix-vector and matrix-matrix multiplication in the enterprise version of SciDB.

2.6.3 MapReduce and Hadoop-based Systems

In line with the big data and massive data parallelism movement, a lot of approaches emerged to implement machine learning applications based on the MapReduce paradigm and Hadoop. However, since MapReduce does not offer any linear algebra functionality natively, there had been a market gap for analytical systems providing linear algebra functionality, built on top of Hadoop.

SystemML

SystemML (Ghoting et al., 2011) is a system that aims to serve as a scalable platform for data mining applications. It is a Hadoop-based, scalable machine-learning framework with a custom, declarative machine learning language (DML) that is similar to R. DML statements are compiled in two stages into a high level operator (HOP), and a low level operator (LOP) plan. Finally, the operators are mapped onto map-reduce tasks and executed in Hadoop. Their matrix data structure consists of individual, fixed-size blocks that can either have a dense or a sparse representation. Some high-level operations such as matrix multiplication can have multiple low-level implementations, two of which are described in their paper. We briefly summarize them in the related work section of Chapter 4. Unlike the previously presented systems that mostly offer isolated linear algebra operations, the abstraction of the generated execution plan from the logical linear algebra expression reveals significant optimization opportunities. Indeed, expression optimization is one of the core objectives of SystemML. Their optimizer is presented in greater detail in a follow-up paper by Boehm et al. (2014b): it includes techniques like static and dynamic algebraic rewrites, many of which are complementing our ideas. In fact, they also discuss matrix chain multiplication optimizations; in particular, they identify a matrix chain multiplication as the core computation in the exemplary logistic regression script sketched in their paper. We revisit the optimizer of SystemML in Section 4.6.1, and reveal some of its shortcomings that have been addressed by our SpMachO optimizer.

Nevertheless, it should be noted that SystemML is a read-only system that exclusively focuses on the optimization of linear algebra scripts. Matrices are read from and written into files, whereas data manipulation operations as such are not foreseen. Furthermore, the disk-based distributed Hadoop file system offers a near-to-arbitrary scalability, while at the same time sacrificing quick query response times for smaller computations. Although in a more recent publication Boehm et al. (2014b) suggest to process small-sized problems in local memory, the engine implementation of SystemML is based on Java, and does not reach the performance of high performance C++ Blas-libraries.

As from November 2015, SystemML has become an open-source project of the Apache Software Foundation (Apache Software Foundation, 2016).

Cumulon

A closely related system that should be mentioned in this context is Cumulon by Huang et al. (2013). Similar to SystemML, it is implemented on Hadoop and targets data analysis applications, despite being designed and optimized for running in a cloud environment with a flexible hardware configuration. They also create a logical plan from an R-like expression, and conduct a series of rule-based logical plan rewrites, including matrix multiplication optimizations. In line with SystemML, they employ a simple cost model to reduce the plan runtime. However, apart from time cost models for different hardware settings, they also optimize for monetary costs that are caused by the cloud usage. As a result, they can generate a deployment plan that generates the least cost for a cloud user. Interestingly, they explicitly state that the MapReduce paradigm is not a good fit for matrix-based computations. They further address SystemML directly in the discussion about the inefficiency of a MapReduce-parallel matrix multiplication. Their key argument is that map jobs are only allowed to have *disjoint* input sets, whereas the tasks in a blocked matrix multiplication require *overlapping* input sets. Thus, the map-tasks perform no useful computation other than replicating matrix data for the reduce tasks. As a consequence, their approach departs significantly from the traditional MapReduce-paradigm, and builds up on "map only" jobs: a "map task does not receive input as key-value pairs; instead, it simply gets specification of the input splits it is responsible for, and reads directly from Hadoop distributed file system (HDFS) as needed".

Unlike the recent version of SystemML, Cumulon is a purely disk-based, read-only system, similar to SciDB. Hence, it is neither suited for efficient ad-hoc queries, nor does it serve our manipulation requirements.

Apache Spark & MLib

The open-source project Spark (Zaharia et al., 2010, 2012) was launched in 2010 and is a framework for cluster computing. Besides HDFS, Spark can also interface other distributed storage frameworks, e.g. the MapR File System (Scott, 2014) or Cassandra (Lakshman and Malik, 2010). In Spark, a "driver" program launches a script that is written in either of Scala, Java, or Python, and invokes parallel computations that are either scheduled locally or on the cluster (Zaharia et al., 2010). Rather than working on key-value pairs, Spark uses an resilient distributed dataset (RDD) (Zaharia et al., 2012). The major advantage over conventional MapReduce is that data is cached in distributed memory, and thus many operations can be executed in-memory while avoiding disk I/O. Hence, Spark

yields a significant speed-up over disk-based MapReduce jobs for ad-hoc analysis. Linear algebra functionality is offered by the LINALG library (Zadeh et al., 2015), which is packaged together with several other algorithms in the Spark machine learning library MLlib (Meng et al., 2014). At the time of writing this thesis, LINALG included the distributed matrix storage types CoordinateMatrix, BlockMatrix and (Indexed-)RowMatrix, where the latter two can contain dense or sparse blocks and rows, respectively. However, not only has the user to select the most appropriate matrix among these types, but also is the matrix processing functionality type-dependent and often very limited. For instance, it is not possible to multiply matrices of type CoordinateMatrix at all, and a (Indexed-)RowMatrix can only be multiplied with a local matrix of different type (Spark Documentation, n.d.). Some high-level algorithms are hard-coded for certain matrix types, e.g., a column similarities method for RowMatrix that is based on a approximate matrix square computation (Zadeh and Carlsson, 2013). Moreover, LINALG comprises a non-distributed SparseMatrix type based on the compressed sparse column (CSC) format[3], which includes a *single-core* sparse matrix-dense matrix multiplication method (Zadeh et al., 2015), but no sparse matrix-sparse matrix counterpart. To summarize, the sparse functionality is very limited, and scalability is mostly restricted to dense linear algebra and sparse matrix-vector multiplication. As for our remaining requirements, neither are the spark matrices able to be manipulated in an ad-hoc fashion, nor does the runtime foresee any logical optimizations of linear algebra expressions.

2.6.4 Commercial Systems and R-Integrations

Despite the need of their customer base to have tools for advanced statistical data analysis, many commercial database system vendors deferred providing statistical functionality for a long time. Only recently, in response to the emerging market of data science and advanced analytics, major DBMS started to extend their statistical feature set. Due to the dominance of statistical software packages, some vendors integrated R with their systems instead of offering custom implementations. Examples are Oracle R Enterprise (Oracle, 2015), Microsoft SQL Server R Services (Microsoft, 2016), the SAP HANA R Integration (Große et al., 2011), and IBM Netezza Analytics (IBM, 2014b). As a matter of fact, published details about proprietary systems are rare. We give a short review about different approaches and extensions of proprietary systems providing linear algebra functionality.

SAP HANA R Integration

The R runtime offers the ability to access the data of any relational database system that implements the standardized JDBC or ODBC interface via the RODBC and RJDBC packages (Ripley, 2015; Urbanek, 2014). However, these network-based interfaces are commonly known for poor performance, in particular for batch transfers of larger data sets. This problem is tackled by the integration of R in the in-memory column-store SAP HANA DB . As presented by Große et al. (2011), the data transfer implementation is based on a shared memory (SHM) method, which is superior to the standard SQL interface. From the user perspective, R-scripts are embedded in R-*nodes*, as part of a program written in SQL-script, which is a proprietary SQL meta-programming language. The DBMS exe-

[3] Efficient sparse matrix formats will be elaborated in greater detail in Chapter 3.

cution engine then calls the R runtime for each R-node, and converts all database tables that are referenced within the R script into the corresponding R-data types, while all data completely remains in the shared memory. The actual computations take place in a single-threaded R runtime, whereas the DBMS acts as a data orchestrator, and starts multiple R runtimes in parallel. However, all linear algebra operations such as matrix multiplications rely on the R algorithm implementations. Consequently, they can only be run within the sequential R runtime on the local data chunk, rather than on a distributed matrix data set as a whole. This opposes the idea of a comprehensive, deep integration of linear algebra in the DBMS engine.

Oracle ORE

Oracle offers the Oracle R Enterprise (ORE) runtime as an optional part of their Enterprise DBMS Edition. The ORE includes a transparency layer, i.e. a collection of R packages that support "mapping of R data types and generate SQL transparently in response to R expressions on mapped data types" (Oracle, 2015). According to Oracle (2013), the mapping of R functions into database-resident operations happens wherever possible: "when in-database equivalents are not available for contributed R packages, Oracle Advanced Analytics can run them via embedded R mode [...]", meaning that additional R runtimes are spawned by the DBMS, which execute the respective R code. The reference manual (Oracle, 2012) indicates a list of functions and operations that are integrated directly in the database engine. The integration includes a matrix type and basic arithmetic operations such as element-wise and full multiplication of matrices. However, it is unclear if matrices are represented as relational tables (e.g., triple tables) in the database system, or if an implicit data conversion to an internal matrix type is performed. The former approach would yield a poor performance for dense matrices, whereas the latter would incur additional conversion overhead and undesired data replication. Moreover, the interpretation of algebraic expressions is left to the R runtime, which results in missed opportunities to optimize on expression level.

As another way to employ linear algebra, Oracle offers the *UTL_NLA* package (Oracle Doc., n.d.) that exposes a subset of the Blas and Lapack operations on vectors and matrices when they are represented as *VARRAY*s. The latter is a proprietary data type for an array table attribute, and thus, not a top level data object. The downsides of *VARRAY*s are manifold: they may not be partially manipulated, they can not be split, partitioned and orchestrated via SQL, and finally, they are only suitable for small and dense matrices.

IBM Netezza

A different way of integrating linear algebra functionality is employed by the data warehouse system Netezza of IBM (2014a), which includes a *MatrixEngine*, and exposes matrices as first class citizens to the user. Although an additional conversion from a $\langle row, column, value \rangle$ database table to a Netezza matrix is required, their approach comes closer to a deep integration than the R-integrations presented above. Matrix operations are top-level operations, and part of their SQL dialect NZPLSQL. Computational intensive tasks, such as matrix multiplications, are handled by calling the Scalapack implementation of the Intel MKL. To our surprise, only dense matrices are supported, which narrows down the applicability of Netezza significantly. Indeed, a wide range of

applications require sparse linear algebra, including all applications presented in the beginning of this chapter.

2.7 SUMMARY

To summarize, we conclude that there have been efforts by the open source community as well as by commercial DBMS vendors to offer linear algebra functionality. Table 2.1 shows a comparison of all approaches that have been mentioned in this chapter with regard to the properties that we have identified as crucial for linear algebra applications in database environments. These are:

1. The ability of processing both dense and sparse matrices.
2. The presence of efficient, parallelized, and scalable operator implementations (e.g., by utilizing shared-memory parallelization.)
3. An expressive, natural language interface for linear algebra.
4. A deep integration of linear algebra and built-in matrix types. This is opposed to the alternative approach of DBMSs that transfer data to an external statistics runtime/library.
5. A transparent interface that abstracts the logical matrix from its physical implementation.
6. The capability to manipulate matrix data in the primary data structures.
7. A high-level optimization of linear algebra expressions.
8. ACID features for matrix data, in particular consistency and persistence.

From Table 2.1 we observe that the different libraries, software toolkits, or systems all cover different aspects of our requirements, but none is addressing all of them equally well. In particular, we have seen that there are efficient library implementations that have been tuned for decades, but are based on non-transparent, static data types, and do not offer a natural language interface. For example, the efficient hardware-vendor-provided BLAS and SCALAPACK libraries, and several hand-tuned high performance algorithms are very strong in parallel resource utilization, but all of them are tightly bound to specific, static data structures. In contrast, native DBMS solutions with a SQL-based implementation (MAD Skills) show a poor performance, but inherently include the advantage of updates and ACID features for relational data. Commercial DBMS vendors offer linear algebra applications to be run in an external R runtime rather than integrating matrices and vectors as first class citizens into their system, which incurs ETL processes and data inconsistencies.

Fully transparent data types, topology-aware matrix structures and a sophisticated expression optimization have barely been covered by any system. This is where our approach can generate most value compared to the other approaches.

Table 2.1: Tabular comparison of frameworks and systems that support linear algebra operations.

Name	Dense LA	Sparse LA	Scalability Multi-core	Language Interface	Deep Integration	Abstract Matrix Type	Data Manipulation	Expression Optimization	ACID Features
Libraries and Software Toolkits									
BLAS	Yes	Partly[a]	+[b]	No	-	Partly[c]	No	No	No
SuiteSparse	No	Yes	- (SpMM)	No	-	No	No	No	No
(SCA-)LAPACK	Yes	No	++[bd]	No	-	No	No	No	No
HPC algorithms	Yes	Yes	++[bd]	No	-	No	No	No	No
LAMA	Yes	Yes	++[bd]	Yes	-	Partly	No	No	No
MATLAB	Yes	Yes	-/Partly[e]	Yes	-	Partly	Partly	No	No[f]
R	Yes	Yes	-/Partly[e]	Yes	-	Partly	Partly	No	No[f]
DBMS, Warehouses, and Analytical Systems									
MAD Skills	Via SQL	Via SQL	-/Via DBMS	Partly (SQL)	No/Via SQL	No	Via SQL	No	No
RIOT-DB	Via SQL	Via SQL	-/Via DBMS	Yes	No/Via SQL	No	Via SQL	No	No
SciDB	Yes	Yes	+[d]	Via R	Yes	Partly	Yes	No	No
SystemML	Yes	Yes	+[d]	Yes	Yes	Yes	No	Yes	Yes
Cumulon	Yes	Partly	+[d]	Yes	Yes	Yes	No	Yes	Partly
Spark & MLib	Yes	Partly	+[d]	API-Level	Yes	No	No	No	No
Oracle R Ent.	Yes	Via R	(unpublished)	Via R	Partly[g]	No	inherent[h]	No	inherent
SAP HANA R	Via R	Via R	Partly[b]	Via R	No	No	inherent[h]	No	inherent
IBM Netezza	Yes	No	+[d]	Partly (SQL)	Yes	No	Yes	No	Partly
Our LAPEG	Yes	Yes	+[b]	Yes[i]	Yes	Yes	Yes	Yes	inherent

[a] Only if the sparse BLAS extensions are included, which depends on the library vendor
[b] Parallelism by simultaneous multithreading (SMT) on multi-core processors.
[c] Clear distinction between dense and sparse matrices, but the sparse extension allow abstract handles
[d] Distributed environment, might involve communication overhead and higher latencies.
[e] Only for dense linear algebra, and with installed vendor-provided BLAS library.
[f] Binary file-based persistence, minor manipulation capabilities.
[g] It has not been published which functions are pushed down to the database kernel.
[h] Might involve internal ETL processes to transfer updates in the relational table to the R runtime.
[i] We presume that our system has a native language interface that is based on R or MATLAB.

3

MATRIX STORAGE AND OPERATIONS

In this chapter, we start with a brief introduction of the in-memory column-store, which is the platform our linear algebra processing engine (Lapeg) is based on. Then, we present fundamental data structures for vectors and matrices. In particular, we consider different matrix types with regard to their integrability into the column-oriented storage layer. We propose multiple ways to map sparse and dense matrices onto table columns, and show how they are exposed as relational tables. Thereafter, we describe algorithms for dense-, sparse- and mixed dense/sparse-matrix multiplications, which will be referred to as multiplication kernels in the remainder of this thesis.

3.1 MAIN-MEMORY COLUMN-STORE

The decrease in RAM prices over the last decades made it feasible and economical to equip single servers with main memory capacities of 1 terabyte and beyond. As a result, the abundance of fast memory has led to a shift from disk-centric to main-memory-centric data processing. In-memory database systems exhibit substantial performance improvements over disk-based systems for analytical queries on large data sets, as reported by Garcia-Molina and Salem (1992). Moreover, the data shift from disk into main memory has further advantages: the data is byte-wise addressable and immediately accessible via the virtual address space, which reduces the complexity over conventional disk-based data processing. For instance, the particularities of disk-based data management are eliminated, including paging, disk buffers, prefetching, et cetera.

Besides the change due to the emerging hardware trends, another architectural redesign that had a significant influence on modern database systems was the introduction of a column-oriented data layout (Copeland and Khoshafian, 1985; Boncz et al., 2005). Abadi et al. (2005) showed that column-stores have a performance advantage over conventional row-oriented approaches on analytical workloads. In a column-store, each table column is stored separately. This enables an efficient predicate evaluation on relational tables, since only the affected attributes are scanned instead of entire rows. The efficiency is particularly increased for querying wide tables, where the memory region that has to be scanned by the system is significantly reduced. Such read-intensive queries predominantly appear in workloads of online analytical processing (OLAP), for which column store exhibit the best performance (Abadi et al., 2008).

As of today, a multitude of column store DBMS have appeared on the market, including commercial and non-commercial systems. Nearly every DBMS vendor offers a column-oriented data

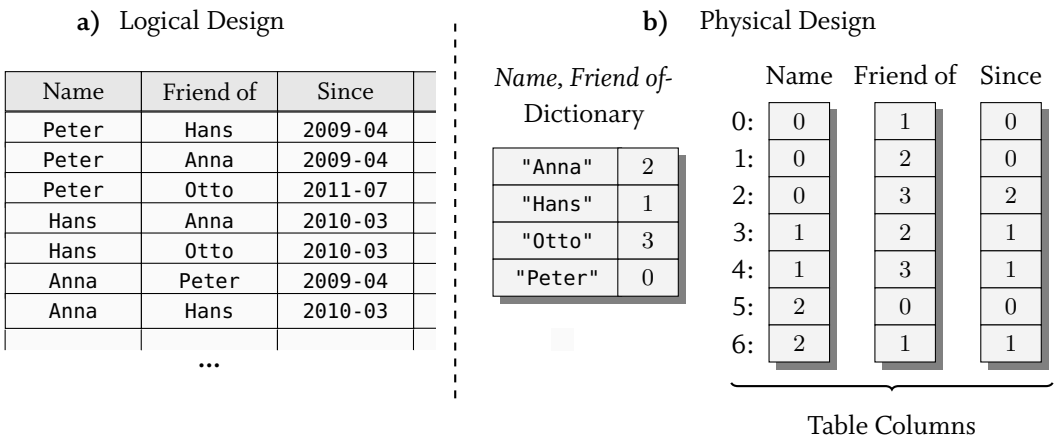


Figure 3.1: Physical organization of a relational social network table in a columnar database system. a) The logical view of the table in the database, which has the attributes *Name, Friend of* and *Since*. b) The internal representation consisting of integer columns and a shared dictionary for *Name, Friend of* (the dictionary for *Since* is omitted).

layout, either as primary architecture or as extension. Examples for column-oriented DBMS include Sybase IQ (MacNicol and French, 2004), MonetDB (Boncz et al., 2008), HP Vertica (Lamb et al., 2012), Vectorwise (Zukowski et al., 2012), IBM DB2 BLU (Raman et al., 2013), Greenplum Database (Pivotal Software, n.d.), Oracle Exadata (Fitzjarrell and Spence, 2013), and the SAP HANA DB (Sikka et al., 2012), which is an in-memory column-store DBMS.

Figure 3.1 shows the logical and physical representation of a relational table in a column-store. The table (Fig. 3.1a) depicts a social network relation that contains connection between friends. It consists of the attributes *Name*, *Friend of*, and possibly a number of additional columns, such as the date when the connection was established (*Since*). It is customary that DBMS encode values of wide data types (e.g., strings and timestamps) as small integers to compress data (Graefe and Shapiro, 1991; Roth and Horn, 1993), and to improve the query performance that is often bound by the data volume that has to be transferred from disk or main memory. Column-stores usually employ a dictionary for each column (Abadi et al., 2006). In particular, the dictionary maps attribute values to consecutive integer values. Hence, the dictionary is a transformation $\mathcal{D}$ that assigns an integer id to each distinct attribute value:

$$\mathcal{D} : \textit{Attribute value} \rightarrow \text{value id} \in \{0, 1, ..., n_{\text{values}} - 1\} \tag{3.1}$$

Figure 3.1b sketches the dictionary for the *Name* attribute (note that the value set of the *Name* attribute is in this case identical to that of *Friend of*, which is why they can *share* a single dictionary.) Only the integer value ids are contained in the physical column representation. By using dictionaries, the memory consumption of columns can be significantly reduced, in particular if the column contains recurring values of wide attribute data types. In commercial environments, business data that is stored in relational tables predominantly contains multiple string attributes. In these situations, the dictionary-encoded representation consumes only a small fraction of the memory that would be occupied by a non-encoded, materialized table. The memory consumption of the integer

id columns can be further reduced by the application of additional compression techniques, e.g. by bit-packing and run-length encoding (RLE) (Abadi et al., 2006).

Apart from a good data compressibility, particularly the table scan performance benefits from a columnar architecture. By utilizing vectorized instructions and simultaneous multithreading (SMT), the throughput of a full column scan in a main-memory DBMS is ultimately limited by the system's memory bandwidth. To make an illustrative example, consider the table of Figure 3.1 that has 10 million rows, and 50,000 distinct names. Consequently, the bit-packed size of the *Name* column is $10^7 \cdot 2$ Byte = 20MB. Assuming an effective memory bandwidth of 20GB/s[1], the scan takes only $(20\text{MB}/20\text{GB})\text{s} \approx 1$ millisecond to complete.

The transition from disk-based, row-oriented DBMS to columnar, in-memory DBMS marked a paradigm shift in database system design, and has created a deep impact on recent DBMS architectures. With the primary data residing in RAM and the growing demand of advanced analytics applications inside the DBMS, it has become worthwhile to investigate how structures and algorithms of linear algebra can be integrated into the column-oriented storage layer of an in-memory DBMS. In the following sections, we show that a columnar storage layout yields an easy adoption of known data representations for matrices and vectors, which enables a multitude of linear algebra applications to be run directly in a column-oriented main-memory DBMS.

3.2 ARRAYS IN A COLUMN-STORE DBMS

Unlike relational tables in business environments, the majority of data in scientific fields is contained in arrays, mainly vectors and matrices. In the following, we first discuss a standard way of representing arrays in a relational DBMS, and outline its shortcomings. We then impose a few requirements necessary to store vectors and matrices more efficiently in a column-oriented DBMS. Thereafter, concrete data types for vectors and matrices are described in Sections 3.3 and 3.4.

3.2.1 A Straw Man's Method.

A straightforward way of storing general n-dimensional arrays in a relational DBMS is to use a table with n key attributes for the cell coordinates, and at least one attribute for the cell value. Hence, each row in the database table represents one element of the array, and conversely, each element (including null and not-null elements) of the array is represented by a table row. For instance, this is the storage representation of matrices in RIOT-DB (Zhang et al., 2009), who are using a row-store DBMS (MySQL). By interpreting the n-tuple as a relation $\mathcal{R}$, some linear algorithms can indeed be expressed by relational means using this format, for instance a matrix multiplication, as we sketched in Listing 2.1 (Section 2.6.1).

Nevertheless this approach has some drawbacks:

- The unordered n-tuple table exhibits rather poor performance for complex operations such as matrix multiplication. This is because unlike relational algebra, which is based on unordered sets of tuples, linear algebra operations on such tables can not benefit from the inherent structure of an array.

[1] Modern multi-core platforms easily reach total bandwidths of 50GB/s and more.

- The n-tuple array table usually results in abundant data storage for fully populated arrays, especially if the dimensionality is high.

- If arrays are sparsely populated, it is further inefficient to store null elements (i.e., zero elements for sparse matrices).

Regarding row-store DBMSs, it was shown that this data layout is by an order of magnitude less inefficient for array processing compared to a system with native array support (Stonebraker et al., 2007). In contrast, we show that a column-store exhibits more efficient ways to store and process arrays. To tackle the disadvantages mentioned above we impose a few basic prerequisites for the column-oriented storage layer, such as the ability to retain a stable table order. The Lapeg can employ much more efficient algorithms on *ordered* and *indexed* data structures, which are common in numerical high performance libraries.

As the majority of analysis applications are based on matrices, we focus on the representation of matrix data in the remainder of this thesis. Although a matrix is a two-dimensional array, there is not a single representation that suits all matrices perfectly, in particular when the matrix sparsity is taken into account. Therefore, we examine separate structures for dense and sparse vectors and matrices.

3.2.2 Storage Engine Prerequisites

To overcome the aforementioned shortcomings of general relational DBMS, we first define some prerequisites to efficiently store and process arrays in the columnar storage layer of a main-memory DBMS. Note that these properties are required to let the Lapeg make direct use of efficient kernel algorithms on the core data structures, e.g. routines from the Blas library. All of the following characteristics are featured by the SAP HANA DB (Färber et al., 2012). These are:

Dictionary-less columns. Usually vectors and matrices accommodate numeric values per definition, which are either of type integer, float or double. Moreover, most matrices contain many different values. Since neither extensive data types are used, nor many duplicate values are present, the aforementioned dictionary encoding becomes superfluous for the majority of scientific data sets. An exception are matrices derived from contextual data, which contain auxiliary, semantic information. For example, consider the term-document relations described in Section 2.3.1: in this case, the dictionaries of the dimension attributes implicitly map arbitrary attribute values (e.g., strings) to matrix coordinates. We will readdress this characteristic in Section 3.4.3. Until then, the following columnar data representations are based on the assumption of dictionary-less, integer and float columns.

Row-positional access. Although relational tables are not foreseen to be referenced by their row position, the Lapeg *internally* requires positional access to table columns. That is, each column in the column-oriented storage layer should be addressable by its index, i.e. the absolute position of a value in the column. This corresponds to the table row position if the column is part of a relational table. Position-accesses are commonly used in column-store operators, e.g., for materializing result rows of a filter query.

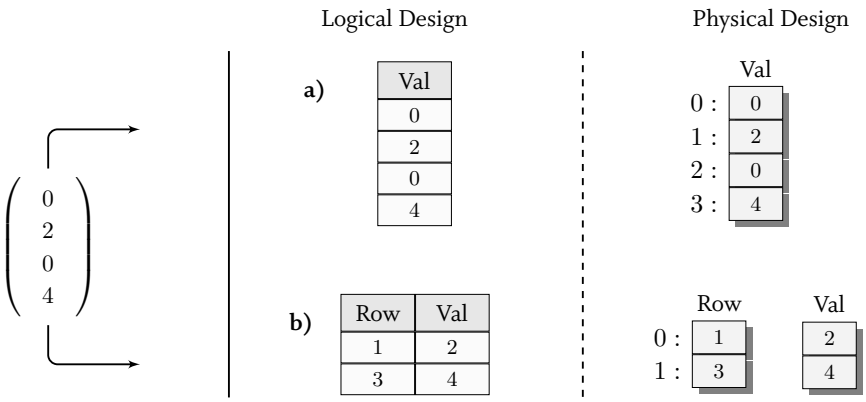


Figure 3.2: Dense (a) and sparse (b) representation of a vector (left) in a column-oriented database system. Middle: Logical exposition as table. Right: physical representation in the column-oriented storage layer.

Configurable table sort. The storage layer should provide a table-wide, flexible sorting mechanism for columns. That is, each column of a table should be ordered at certain reorganization checkpoints (e.g., at the initial creation of a matrix) according to a given mapping that is internally provided by the Lapeg. This mapping result from the sort order of one or multiple attributes of the table (e.g., the matrix row coordinate). Internal column reordering during table reorganization is a common technique in column-stores to improve the compressibility of tables (Abadi et al., 2006).

Row order preservation. Finally, we require that a table containing a matrix or vector representation preserves the current order at any time, except if the reorganization is triggered by the Lapeg. In fact, general column-store tables are usually only reordered during a reorganization process, e.g., to improve the table compression (Lemke et al., 2010). For general relational tables, this reorganization may decide to reorder a table according to some heuristics, which is undesired for our array structures that require a stable order. Hence, the column tables containing matrix data are flagged by the Lapeg to be ignored by any heuristic reorganization routine of the DBMS.

3.3 VECTOR DATA TYPES

A vector is a linear data object, and is commonly represented as a structured, sequential collection of values. We distinguish between *dense* and *sparse* vectors.

3.3.1 Dense Vectors

As shown in Figure 3.2a, the dense vector just corresponds to a single *Val* column in the column-oriented storage layer, which stores each value including zeros. The logical position of a value in the vector simply corresponds to the physical position of the value in the *Val* column. This column could be exposed as a database table with preserved row order.

3.3.2 Sparse Vectors

In contrast to dense vectors, a sparse vector contains one or more zero-values. Due to their special mathematical characteristics, zero-values can graciously be ignored in arithmetic calculations: addition with zero does not change the result, and multiplication with zero always produces a zero-value. Hence, similar to the null value suppression in relational data (Abadi et al., 2006), zero-values are not stored.

Since exclusively non-zero elements are stored, the logical vector position does no longer correspond to the physical position in the value column. Therefore, we add the vector's row indices as additional *Row* attribute to the table. The sparse vector representation deviates from the "straw man's method" for one-dimensional arrays, since the table columns only store non-zero elements and are ordered by ascending row coordinates. The two columns illustrated in Figure 3.2b both have length N_{nz} (=number of non-zero values).

3.4 MATRIX DATA TYPES

In contrast to vectors, the two-dimensionality of matrices exhibits many more possibilities for the data structure to be laid out in memory. In general, we again differentiate primarily between dense and sparse matrices.

3.4.1 Dense Matrices

There are different ways to represent a dense matrix in the column-oriented storage layer, two of which we briefly describe.

Matrix table. Due to its two-dimensionality, a naive way to store a dense matrix in an RDBMS is a direct mapping of matrix rows to table rows and matrix columns to table columns. This approach results in a $m \times n$-sized table for an $m \times n$ matrix, as shown in Figure 3.3a. The apparent advantage of the direct matrix table representation is the preservation of the logical two-dimensionality of a matrix, hence, it is intuitive to query and visualize. Obviously, another advantage is fast matrix column reads. However, this representation is impracticable when the matrix is extremely wide, since the number of table columns in common DBMS is usually restricted. Moreover, another drawback is the prohibitive inefficiency of matrix row reads: not only are adjacent elements of a matrix row contained in different memory segments, but also their distance is irregular, since separate columns in the column-oriented storage layer are generally not stored in contiguous memory sections. Hence, a matrix row read results in an unpredictable access pattern for the memory prefetcher. In the worst case, each of the n row elements would ultimately incur a random memory access.

Linearized Value Column. Another way to represent a dense matrix is to linearize the full matrix into a contiguous sequence of values. In the column-oriented storage layer this approach yields a single, large value column as sketched in Figure 3.3b. The value column is equivalent to a dense representation of a large vector that contains all values of the linearized matrix. The remaining degree of freedom is the linearization order of the two dimensions. In the context of two-dimensional

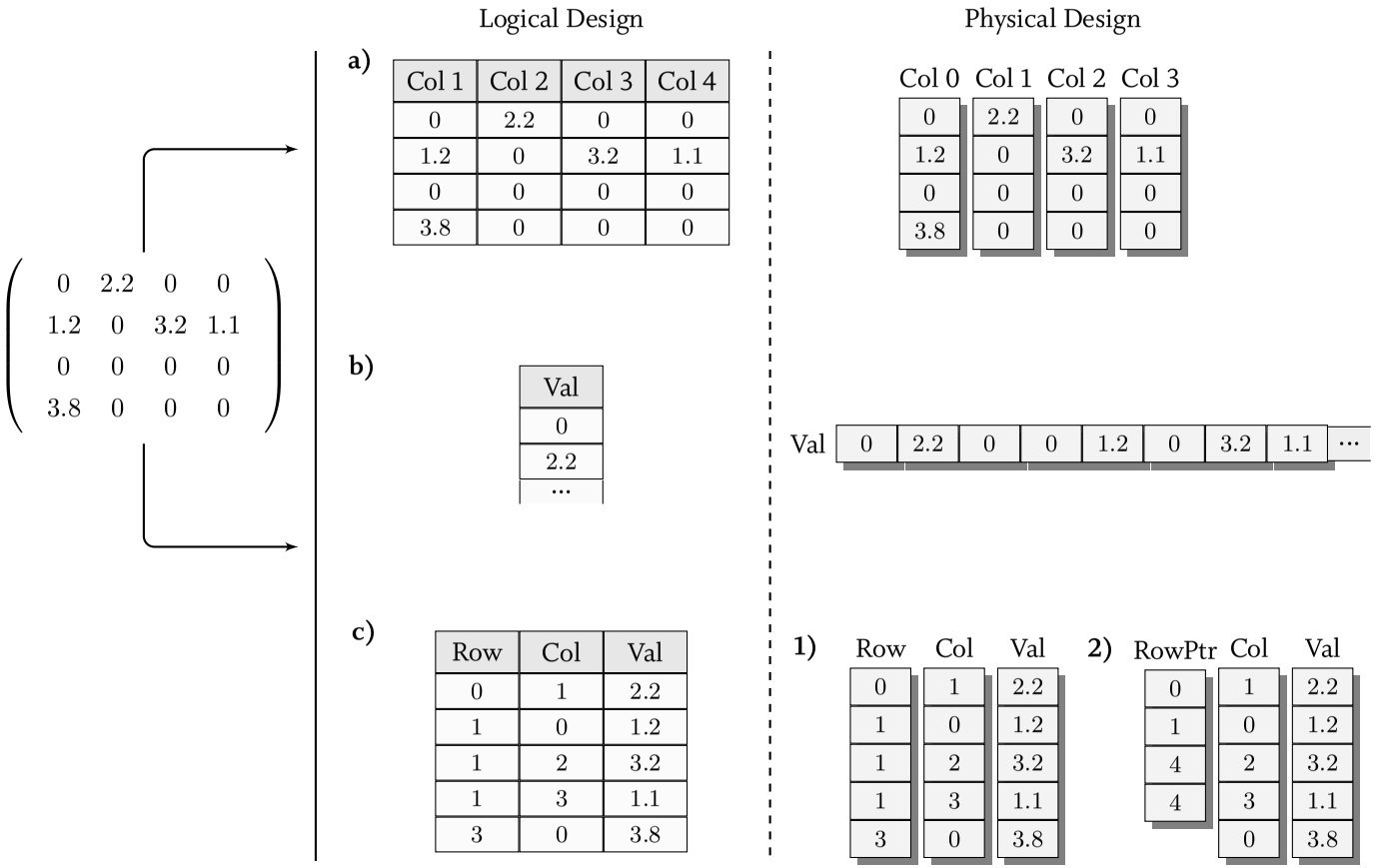


Figure 3.3: Overview of the different approaches to represent a matrix logically in a DBMS (middle) and their internal representations in a columnar storage (right): a) Matrix table, b) single value column, c1) triple table, c2) CSR representation

arrays and matrices, it is common to choose either a *row major*, or a *column major* order, depending upon whether the matrix row or column is selected as the leading dimension. In addition, more complex matrix linearizations can be used to create hierarchical storage formats, e.g., to leverage recursive matrix algorithms (Valsalam and Skjellum, 2002; Gottschling et al., 2007).

Since the linearization order is a property that affects nearly every matrix representation, the topic is addressed at several places of this thesis. Although it is a good practice to chose an order that is coinciding with the algorithm access pattern, most tuned implementations of linear algebra algorithms (e.g., the LAPACK library) process dense matrices in a block-wise manner. The blocking lessens the impact of linearization, and yields a good cache re-usage. For now, we therefore neglect the impact of the array linearization on the performance, and without loss of generality, arrange dense matrices in a row major order. Note that the content of the dense matrices in a row-major linearized *Val* column is internally equivalent to a flat C-style array. Hence, the position of each matrix element in the value sequence as of Figure 3.3b is calculated via its (i, j) coordinates, and the matrix dimensions $m \times n$, such that $pos(i, j) = i \cdot n + j$.

3.4.2 Sparse Matrices

Nearly every big data matrix in real-world applications is sparse. Hence, the majority of matrix elements is zero and does not need to be considered in any matrix operation. Examples are the sparse Hamiltonian matrix $\boldsymbol{H}$, the sparse term-document matrix $\boldsymbol{A}$, or the adjacency matrix $\boldsymbol{G}$ of a network graph, described in our application sections 2.2- 2.3.2. For readers interested in further examples of sparse matrix data, we refer to the Florida Sparse Matrix Collection (Davis and Hu, 2011).

In the context of a column-oriented database engine, one could argue that any sparse matrix contained in a dense representation as of Figure 3.3a or Figure 3.3b can just be compressed to obtain a space-efficient storage representation. For instance, by applying an order-preserving compression algorithm on the individual columns. However, the approach of storing a sparse matrix by using a "dense representation plus compression" is unfavorable. In the first instance, sparse matrix input data does usually not contain any zero values. Hence, it is inconvenient to artificially insert zero values just to compress them in a later step. Furthermore, a general purpose compression algorithm such as RLE generates an encrypted compressed structure that must be decompressed before the data can be further processed, e.g., in a matrix operation, adding further overhead to the total execution time. In contrast, there are compressed data representations that are specially designed for sparse matrices, and most matrix algorithms directly work on these compressed data structures.

Note that we will not provide an extensive overview of every available specialized matrix structure within this thesis. Instead, we first present two approaches that seamlessly integrate with the column-oriented storage layer, and revisit the topic of sparse matrix structures when we introduce our advanced matrix representation in Chapter 5. For a more complete overview and detailed descriptions about other sparse matrix representations we refer to the work of Saad (1994).

Triple Representation

One way to represent a sparse matrix is by a collection of triples (Fig. 3.3c), where each triple contains the row and column coordinate, and the value of the corresponding non-zero matrix element:

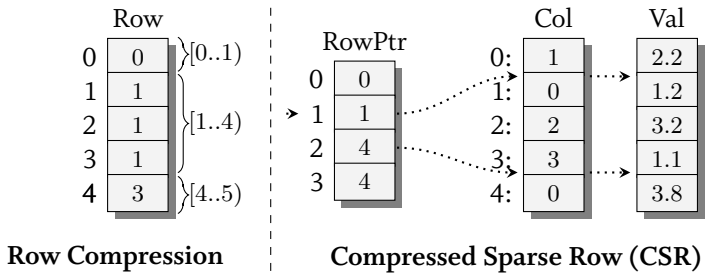


Figure 3.4: The CSR representation. Left: Compression of the row container. Right: The row pointer vector (RowPtr). The access path to the first matrix row is sketched.

$\langle row, col, val \rangle$. The unordered "triple table" corresponds to the n-tuple table for $n = 2$-dimensional arrays, except that only the non-zero matrix elements are stored. In the column-oriented storage layer the triple table is represented by three separate columns (Fig. 3.3c1). Additionally, a composite key constraint on the *row* and *col* attributes can be leveraged to avoid duplicate matrix elements.

The resulting internal column structure is also known as the coordinate format (COO), which is a supported sparse matrix type in some numerical libraries (e.g. NISTb, n.d., and its derivatives). Each of the columns has the length N_{nz}, which is equal to the number of non-zero matrix elements. By using long integers for the *row* and *col* attributes, and double type for *val*, the total memory consumption is $N_{\text{nz}} \cdot 24$ bytes. To find an element in an unsorted, non-indexed triple table a full column scan is required in $\mathcal{O}(N_{\text{nz}})$ time.

As mentioned before, the concept of having a unordered set of triples is obstructive for the development of efficient matrix algorithms. In particular, the triple table representation reveals additional compression and indexing opportunities, when it is sorted by one of the two coordinates. The sort generates adjacent chunks of identical numbers in the corresponding *row*, or *col* column, which can further be compressed, and replaced by an index. However, these transformations are significantly mutating the triple representation, and influence both the update and algorithmic behavior. Hence, the compressed format is considered as a separate representation.

CSR Representation

Consider a matrix in the triple representation as of Figure 3.3c1, which is sorted according to the row major order. Due to the ordering, the row column contains adjacent chunks of identical matrix row indices. This allows us to replace the column by a smaller row pointer (RowPtr) structure that only contains the start positions of each matrix row, as shown in Figure 3.4. To put it differently, the row pointer is an effective compression of the sorted row column. However, the major role of the row pointer is to act as an index for the matrix: the physical position of a matrix row in the column and value attribute of the triple table are obtained by only two look-ups in the RowPtr. For instance, to get all elements of matrix row 1, each triple starting from position RowPtr[1] up to position RowPtr[2]-1 is returned. The row pointer vector itself has size $m + 1$ for a $m \times n$ matrix, where the $(m + 1)^{\text{th}}$ element denotes the end position of the last row.

The resulting physical structure as of Figure 3.4 is identical to a widely used format for storing sparse matrices, called compressed sparse row format (CSR, aka. CRS – compressed row storage).

It first appeared in the literature as the data structure in the description of a sparse matrix multiplication algorithm by Gustavson (1978). Due to the simple and efficient layout, CSR remained one of the most popular sparse matrix representations supported by many numeric libraries, including the original sparse BLAS NISTb (n.d.). Moreover, Vuduc (2004) found that CSR-based algorithms tend to have the best performance for sparse matrix vector multiplications. The total memory consumption of the CSR representation is $(16N_{nz} + 8(m + 1))$ bytes. Hence, it is lower than that of the triple format given that $(m + 1) \leq N_{nz}$, which is always the case except for hypersparse matrices that contains rows of zeros ("empty" matrix rows).

A complementary format to CSR is the compressed sparse column (CSC) format (Barrett et al., 1994; Saad, 1994). Simply put, CSC is the column-major equivalent to CSR. Interestingly, the CSC representation of a matrix $\boldsymbol{A}$ is equal to the CSR representation of the transposed matrix $\boldsymbol{A^T}$, and vice versa. As a consequence, some algorithms written for CSR matrices can be used as well for CSC matrices in a slightly modified way, e.g. by a prior transformation of the operation into the transposed problem. We will address this row-major/column-major dualism in greater detail in Section 3.4.4.

Besides CSR/CSC, there are many more storage formats for sparse matrices. As mentioned before, a comprehensive list of different types can be found in related work (Saad, 1994; Vuduc, 2004). However, the efficiency of a storage representation strongly depends on the specific topology of the matrix, since there are typical, recurring shapes, such as diagonal, block diagonal or blocked matrices. In particular, many representations are well designed for distinct non-zero patterns (the distribution of non-zero elements in the matrix), such as band or diagonal-dominated matrices, but are poorly suited for general matrices, which are rather uniformly populated. Therefore, we mostly adhere to CSR to represent homogeneous, general-purpose sparse matrices in a read-optimized way. Among the reasons for this selection are the universality, simplicity (Saad, 1994), and good performance of CSR for multiplication operations (Vuduc, 2004), but primarily that it can be seamlessly integrated into the column-oriented storage layer. Based on that, however, we will present a superior, heterogeneous data structure that internally consists of multiple CSR and dense arrays as substructures in Chapter 5.

3.4.3 Leveraging CSR to Index Relational Tables

The CSR representation and the concept of the row pointer index are not limited to numerical matrix data. Indeed, any database table accommodating a relation $R = \{a_1, a_2, ...\}$ of one or more attributes of arbitrary value type is internally transformable into a CSR-like structure. The only prerequisite for creating a CSR representation is the presence of a *sorted integer column*, which corresponds to the leading *row* coordinate attributes of a sparse matrix in the representation as of Figure 3.3c1. Fortunately, the integer column is implicitly given for attributes of any data type by the default dictionary encoding. The remaining requirement is the ability to sort the table by the leading column, which is supported by most columnar storage engines. Optionally, the table may be ordered by secondary columns such as the *col* attribute. In relational DBMS, a similar technique is known as dimension clustering (Baumann et al., 2016), which is used to speed up analytical workloads.

As an example, we again refer to the social network table of Figure 3.1. At first the dictionary maps the string entries of the *Name* attributes to consecutive integer ids, starting with zero.

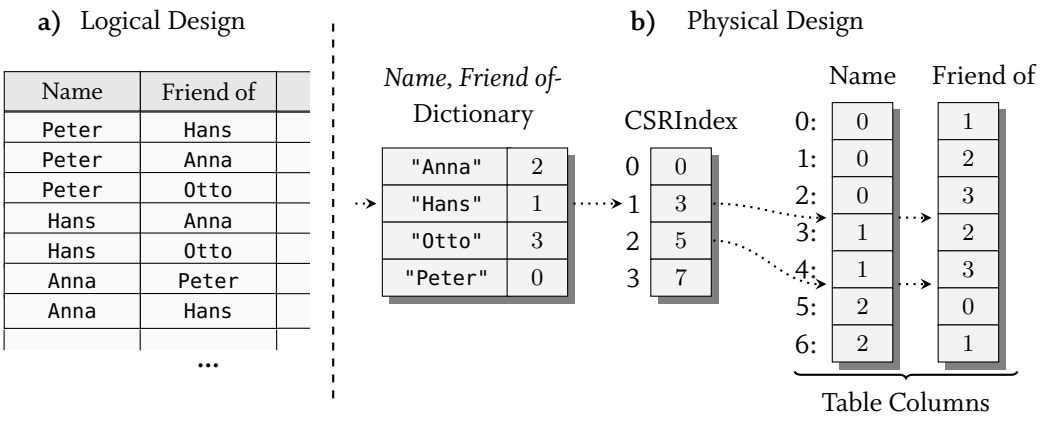


Figure 3.5: The dictionary-encoded social network graph table of Figure 3.1 indexed with the row pointer as CSR index. The *Name* and *Friend of* attributes denote the graph topology, auxiliary attributes are omitted for illustration.

Then, all physical table columns are synchronously sorted according to the value ids of its leading attribute. The latter is defined by the desired topology, which is the *Name* attribute in Figure 3.5, corresponding to the *row* coordinate in the triple matrix representation (*Friend of* corresponds to the *col* coordinate). Following the sort, the row pointer is created, which is depicted in Figure 3.5b. Here, we call it *CSRIndex*, which is formally equivalent to the RowPtr vector (Fig. 3.4). However, in contrast to CSR, it does not replace the *Name* column, but coexists as auxiliary index structure. Figure 3.5b illustrates the search path for querying all friend connections of "Hans": first, a binary search on the sorted dictionary reveals the value ID=1. The latter corresponds to the position in the CSRIndex, which returns the start position ($= 3$) and end position ($= 5 - 1 = 4$) of the region of matching value IDs in the column. Finally, the query engine assembles the value IDs of all remaining attribute columns in the affected region, and returns their materialized values via inverse look-ups on the dictionary.

It should be mentioned that the table may contain an arbitrary number of additional attributes. In addition, the sorting would naturally imply that the data structure is static, since any updates would disrupt the order. In fact, our data structures are *not static* due to an innovative update mechanism. However, we leave this discussion to Chapter 6, which will be devoted to the mutable matrix data structures.

Graph Tables

In a column-oriented DBMS, graph data is present in the form of a relational table as shown in Figures 3.1 and 3.5. The social network table reflects a directed graph topology: the *Name* attribute denotes the start vertices, and *Friend_of* the target vertices of the network graph, respectively. Each entry of the table refers to an edge between these two vertices, hence, the table is consequently called *edge table* in related work (Rudolf et al., 2013; Paradies et al., 2015). In the property graph model, an arbitrary number of additional property attributes can be added to the edge table (Fig. 3.1). As mentioned before, additional columns can simply coexist in the data structure, since the adja-

cency matrix is merely defined via the topology-defining dimension attributes (*row* and *col*). As a result, the social network graph can be efficiently stored by using our CSR-based sparse matrix layout in the column-oriented storage layer.

To summarize, we conclude that general relational tables with a two-dimensional topology (e.g., edge tables) can be interpreted and efficiently stored as sparse matrices. Without loss of generality, we will proceed with numeric matrices and integer matrix coordinates in the remainder of this paper. We assume that in the case of non-integer data types, either an explicit- (by the user) or implicit dictionary encoding (by the DBMS) is applied to raw data.

3.4.4 Impact of the Linearization Order

The linearization order plays an important role in most of the above-mentioned representations. Almost every algorithm has an access pattern that favors a certain order.

Row-major vs. Column-major. The common row-major and column-major orderings are biased regarding read access in one of the two dimensions. For instance, a row-major ordered dense matrix should not be chosen as internal layout when it is most likely that matrix columns are read. In such a case, the iteration over a matrix column would cause a strided memory access: adjacent elements in a matrix column are separated by a matrix row width. Consequently, the elements of the matrix column are spread over the complete memory section that accommodates the matrix. In contrast, if the matrix is column-major ordered, the same read operation yields a cache-efficient, locally restricted and sequential memory scan.

CSR and CSC. In a similar way, the linearization order is encoded in the selection of either the CSR (row-major) or the CSC (column-major) sparse matrix format. In other words, the dualism between row major and column major-ordered dense matrices corresponds to the dualism between CSR and CSC. With the row pointer vector as index, algorithms with a row-centric pattern obviously benefit from using a CSR structure, whereas column-centric algorithms would favor a CSC matrix layout. The clear distinction between these layouts obviously leads to an asymmetric matrix row/column read performance, once a particular format is selected.

Space Filling Curves. There are also non-biased linearizations based on space filling curves, such as the recursive Z-order by Morton (1966), or the Hilbert curve. These linearizations aim at preserving locality in both dimensions equally. However, they exhibit rather poor cache locality for accessing either full matrix columns or rows compared to CSR/CSC. A majority of kernel algorithms have a rather one-dimensional access pattern. More precisely, they are written such that primarily either matrix rows, or matrix columns are read. One reason is that one-dimensional access pattern eases the exploitation of vectorization commands. Examples are the matrix-vector or matrix-matrix multiplication algorithms that are described in Sections 3.5ff. Nonetheless, it should be mentioned that HPC dense linear algebra algorithms benefit from blocking, which is why meanwhile some HPC libraries went over to use tiled data layouts (Buttari et al., 2009). Similar layouts are also used in cache-oblivious, recursive algorithms (Valsalam and Skjellum, 2002; Gottschling et al., 2007). In particular, they use a combination of the Z-order for the coarse arrangement of blocks, and row-

major/column-major for the fine-granular, internal layout within these blocks. Nevertheless, we omit further details here, since we devoted Chapter 5 to the topic of matrix tiling and tiled multiplication.

One Linearization Fits it all? The selection of a linearization order is usually made when the matrix is loaded, either by the user, or by the linear algebra runtime/system. We now consider a case where an algorithm expects a CSC matrix structure, but the matrix is only available in the CSR representation. Then, there are two options:

- **Option A)** The first option is to convert the matrix from one representation into the other prior to the operation. Obviously, frequent conversions of large matrices are undesirable, in particular if the conversion costs do not amortize over time. Therefore, the Lapeg can accommodate query statistics and act as advisor, which reorders the matrix representation according to the most frequent access pattern (Kernert et al., 2013).

- **Option B)** In some cases a conversion from a row major into a column major representation can be omitted by exploiting the mathematical characteristics of matrix transposition. Indeed, it is possible to use a column major multiplication algorithm for row major ordered matrices. This is explained by the mathematical characteristics of a transposed matrix multiplication, and due to the fact that the row major (rm) representation of a matrix $(\boldsymbol{A})_{\text{rm}}$ is identical with the column major (cm) representation of its transpose $(\boldsymbol{A}^{\text{T}})_{\text{cm}} = (\boldsymbol{A})_{\text{rm}}$. Thus, it is

$$\boldsymbol{AB} = ((\boldsymbol{B}^{\text{T}})(\boldsymbol{A}^{\text{T}}))^{\text{T}} \tag{3.2}$$

$$\begin{aligned} \textsc{rm_mult}((\boldsymbol{A})_{\text{rm}}, (\boldsymbol{B})_{\text{rm}})_{\text{rm}} &= \textsc{cm_mult}((\boldsymbol{B}^{\text{T}})_{\text{cm}}, (\boldsymbol{A}^{\text{T}})_{\text{cm}})^{\text{T}}_{\text{cm}} \\ &= \textsc{cm_mult}((\boldsymbol{B})_{\text{rm}}, (\boldsymbol{A})_{\text{rm}})_{\text{rm}}. \end{aligned} \tag{3.3}$$

  Analogously, it is $\boldsymbol{A}_{\text{CSR}} = (\boldsymbol{A}^{T})_{\text{CSC}}$, so equation 3.3 is also valid for the multiplication of two CSC (CSR) matrices with an CSC (CSR) algorithm.

The prototype implementation of Lapeg uses row-major order for dense matrices, and CSR for sparse matrices as default. These orders are supported by all of our described algorithms, as well as by algorithms of common high-performance Blas libraries, such as Intel MKL.

3.5 MATRIX-VECTOR MULTIPLICATION

The following sections are devoted to the major linear algebra operations that are used in our example applications described in Chapter 2. Namely, we focus on multiplication algorithms as they form the building blocks of the iterative linear algebra computations. Of particular importance is the multiplication of a large, dense or sparse matrix with a vector. Most of the time in the nuclear physics application (Section 2.2) is spent in the computation of matrix-vector multiplications (line 6 in Algorithm 1).

Generally, the multiplication of a matrix with a vector $\boldsymbol{y} = \boldsymbol{Ax}$ as of Equation (2.2) can be implemented in multiple ways. We present three multiplication approaches that we implemented in our Lapeg, where each one can be more or less suited depending on the situation.

3.5.1 Inner Product Formulation

If the vector $\boldsymbol{x}$ is dense, it is reasonable to use an inner product formulation, known from undergraduate math courses. Conceptually, each element y_i of the target vector $\boldsymbol{y}$ is calculated by the inner product of the corresponding matrix row $\boldsymbol{a}_{i*}$ with the vector $\boldsymbol{x}$:

$$\boldsymbol{y} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \\ a_{21} & a_{22} & a_{23} & \vdots \\ a_{31} & a_{32} & a_{33} & \\ & \dots & & \ddots \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \end{pmatrix} = \begin{pmatrix} \boldsymbol{a}_{1*}\boldsymbol{x} \\ \boldsymbol{a}_{2*}\boldsymbol{x} \\ \boldsymbol{a}_{3*}\boldsymbol{x} \\ \vdots \end{pmatrix}$$

This method is conveniently implemented by using a row-major ordered data structure for the matrix $\boldsymbol{A}$. For instance, we use a row major ordered value sequence column (Fig. 3.3b) if the matrix is dense, or a CSR representation (Fig. 3.3c2) in case the matrix is sparse.

Algorithm 5 Ordered Inner Product Matrix Vector Multiplication (Conceptual)

```
1: function GEMV(Matrix A, Dense Vector x)
2:     for each row i ∈ A do                                  ▷ do in parallel
3:         for each element a_ij in row i do
4:             y.Val.at(i) += a_ij × x.Val.at(j)              ▷ global y
5:     return y
```

The operation is sketched by the pseudocode in Algorithm 5. The columns of the corresponding representation of matrix $\boldsymbol{A}$ are processed in a single pass. For each matrix element, the algorithm involves a read-access to the vector column $\boldsymbol{x}$, as well as a write-access to the target vector column $\boldsymbol{y}$. In case of a sparse matrix $\boldsymbol{A}$, the j-coordinate of the non-zero elements a_{ij} is not contiguous and potentially yields in random memory accesses. However, if the dense vector columns are small enough to fit in one of the caches, the vector memory accesses play a minor role in the computation time. This should be the case in instances where the dimension of the dense vector is below about $m \leq 10^5$.

Up to now we assumed that the matrix $\boldsymbol{A}$ is present in a row-ordered format. If the matrix is instead present in an unordered format, such as the unordered triple table, the sequential row-after-row processing order has to be relaxed. As a result, the elements of the target vector $\boldsymbol{y}$ are written in an arbitrary order.

Algorithm 6 Unordered Inner Product Matrix Vector Multiplication (Conceptual)

```
1: function GEMV(Matrix A, Dense Vector x)
2:     for any next element a_ij ∈ A do                       ▷ do in parallel
3:         y_L.Val.at(i) += a_ij × x.Val.at(j)                ▷ thread-local y
4:     y = MERGE(y_L1, y_L2, ...)                             ▷ merge local results into global y
5:     return y
```

The pseudocode of the unordered method is sketched in Algorithm 6. Similar to the ordered method, only a single pass over the matrix $\boldsymbol{A}$ is required. However, due to the disorder, the elements

of $\boldsymbol{y}$ are written in an arbitrary order. This complicates a parallelization of the unordered algorithm compared to the ordered inner product formulation (Alg. 5). In contrast to the former, the latter guarantees that different outer loop iterations write to disjunct positions of the target vector $\boldsymbol{y}$. Hence, the outer loop (line 2) of Algorithm 5 can be safely parallelized, whereas parallel execution units of the loop (line 2) in Algorithm 6 might indeed write to the same i locations. Therefore, each thread first writes into a local buffer $\boldsymbol{y}_L$, which is merged in a second step.

3.5.2 Outer Product Formulation

The pitfall of the inner product formulation is that each element of the vector $\boldsymbol{x}$ is considered in the computation, even if many of the x_i's are zero. If $\boldsymbol{x}$ is sparse, a lot of instructions are wasted by multiplication with zero. In this case, it is more promising to use the outer product formalization, and skip multiplications for every element x_i that is zero:

$$\boldsymbol{y} = x_1 \cdot \begin{pmatrix} a_{11} \\ a_{21} \\ a_{31} \\ \vdots \end{pmatrix} + x_2 \cdot \begin{pmatrix} a_{12} \\ a_{22} \\ a_{32} \\ \vdots \end{pmatrix} + x_3 \cdot \begin{pmatrix} a_{13} \\ a_{23} \\ a_{33} \\ \vdots \end{pmatrix} + \ldots$$

Here we assume that $\boldsymbol{x}$ is present in the sparse vector format (Fig. 3.2b), so that only non-zero x_i are evaluated. Due to the columnar access to $\boldsymbol{A}$, it is intuitive to use a column major representation for $\boldsymbol{A}$, such as the CSC representation. However, we can transform the multiplication into the transposed problem:

$$\boldsymbol{y}^T = \begin{pmatrix} x_1 & x_2 & x_3 & \ldots \end{pmatrix} \cdot \begin{pmatrix} a_{11} & a_{21} & a_{31} & \\ a_{12} & a_{22} & a_{32} & \vdots \\ a_{13} & a_{23} & a_{33} & \\ & \ldots & & \ddots \end{pmatrix} = x_1 \cdot \boldsymbol{a}_{*1}^T + x_2 \cdot \boldsymbol{a}_{*2}^T + \ldots$$

$$\boldsymbol{y}^T = \quad x_1 \cdot \begin{pmatrix} a_{11} & a_{21} & a_{31} & \ldots \end{pmatrix} + x_2 \cdot \begin{pmatrix} a_{12} & a_{22} & a_{32} & \ldots \end{pmatrix} + \ldots \tag{3.4}$$

The product in Equation (3.4) is efficiently implemented by using the CSR representation of the transpose $\boldsymbol{A}^{\mathrm{T}}$. However, if matrix $\boldsymbol{A}$ is already present in the CSR format, the Lapeg may decide to perform a conversion $\boldsymbol{A}_{\mathrm{CSR}} \rightarrow \boldsymbol{A}_{\mathrm{CSR}}^{\mathrm{T}}$ prior to the computation. The transformation costs are usually amortized by a few iterative matrix vector multiplications. In fact, a conversion can be completely obviated if the matrix $\boldsymbol{A}$ is symmetric, since then $\boldsymbol{A}_{\mathrm{CSR}} = \boldsymbol{A}_{\mathrm{CSR}}^{\mathrm{T}}$. Symmetric matrices appear in many applications, including the Hamiltonian matrices of the nuclear physics use case described in Section 2.2, or the breadth-first search on undirected graphs introduced in Section 2.3.2.

Algorithm 7 sketches the conceptual outer product algorithm, whereas Algorithm 8 shows its concrete implementation using a CSR matrix, and a sparse vector representation. The structure of the resulting vector $\boldsymbol{y}$ can either be dense or sparse, and should be chosen in accordance with the estimated density of $\boldsymbol{y}$. In order to run Algorithms 7 and 8 in parallel, each thread has a local buffer for the target vector, which is merged in a second step, similar to the unordered inner product algorithm (Alg. 6).

Algorithm 7 Outer Product Matrix Vector Multiplication (Conceptual)

1: **function** GEVM(Matrix $\boldsymbol{A}$, Vector $\boldsymbol{x}$)
2: **for each** (non-zero) element $x_i \in \boldsymbol{x}$ **do** ▷ do in parallel
3: **for each** (non-zero) element a_{ij} in *row* i **do**
4: $(\boldsymbol{y}_L)_i \mathrel{+}= x_i \times a_{ij}$ ▷ thread-local y
5: $\boldsymbol{y} = \text{MERGE}(\boldsymbol{y}_{L1}, \boldsymbol{y}_{L2}, ...)$ ▷ merge local results into global y
6: **return** $\boldsymbol{y}$

Algorithm 8 Outer Product Matrix Vector Multiplication (Implementation)

1: **function** GEVM(CSR Matrix $\boldsymbol{A}$, Sparse Vector $\boldsymbol{x}$)
2: **for** $0 \leq x_iter < \boldsymbol{x}.size()$ **do** ▷ do in parallel
3: $x_col = \boldsymbol{x}.\text{Col}.at(x_iter)$
4: $a_row_start = \boldsymbol{A}.\text{RowPtr}.at(x_col)$
5: $a_row_end = \boldsymbol{A}.\text{RowPtr}.at(x_col + 1)$
6: **for** $a_row_start \leq a_iter < a_row_end$ **do**
7: $a_col = \boldsymbol{A}.\text{Col}.at(a_iter)$
8: $\boldsymbol{y}_L.\text{Val}.at(a_col) \mathrel{+}= \boldsymbol{x}.\text{Val}.at(x_iter) \times \boldsymbol{A}.\text{Val}.at(a_iter)$
9: $\boldsymbol{y} = \text{MERGE}(\boldsymbol{y}_{L1}, \boldsymbol{y}_{L2}, ...)$ ▷ merge local results into global y
10: **return** $\boldsymbol{y}$

Regardless of which of the above algorithms is used, and whether the matrix $\boldsymbol{A}$ is stored in a sparse or dense representation, the performance of the matrix vector multiplication (sparse or dense) is generally limited by the system memory bandwidth (Goumas et al., 2009; Schubert et al., 2011). In practice, the bandwidth puts a physical limit on the tweaking potential of the GEMV operation. Consequently, the primary goal of any implementation is to fully saturate the memory bandwidth, which is easily achieved by utilizing multiple processors. We address matrix-vector multiplications again in Chapter 6, where we present an implementation of GEMV based on our *updatable* matrix structure.

3.6 MATRIX-MATRIX MULTIPLICATION

In this section and the following, we will present different approaches to multiply a matrix $\boldsymbol{A}$ with another matrix $\boldsymbol{B}$. According to the dense BLAS classification, matrix multiplication is a level 3 operation, since it has with $\mathcal{O}(n^3)$ a higher complexity than the level 2 operations that are in $\mathcal{O}(n^2)$, e.g., a matrix-vector multiplication. Both dense and sparse matrix multiplications are the most time- and space-consuming operations in many linear algebra applications. To name an example, we recall the calculation of the large similarity matrix in the document clustering application (Section 2.3.1). Therefore, efficient multiplication routines are paramount in the LAPEG. Consequently, we dedicate a major fraction of this thesis to the intra-operator optimization of adaptively tiled matrix multiplication (e.g., in Chapter 5), and the inter-operator optimization of chain multiplication expressions (Chapter 4).

3.6.1 Dense Matrix-Matrix Multiplication

The dense matrix multiplication $\boldsymbol{C} = \boldsymbol{AB}$ (Eq. 2.5) is typically implemented using the popular inner product formulation, where each row of matrix $\boldsymbol{A}$ is multiplied with each column of matrix $\boldsymbol{B}$:

$$c_{ij} = \boldsymbol{a}_{i*}\boldsymbol{b}_{*j} \quad \{\forall i, j | i \in [1, m], j \in [1, n]\} \tag{3.5}$$

To remain consistent, we will use the dimension naming convention as introduced in Equation (2.5) throughout this thesis. That is, matrix $\boldsymbol{A}$ has the dimensions $m \times k$, $\boldsymbol{B}$ is a $k \times n$ matrix, and consequently, $\boldsymbol{C}$ is of dimensionality $m \times n$. The product in Equation (3.5) is conveniently implemented by an algorithm with three for-loops: the first loop iterates over the rows of $\boldsymbol{A}$, the second over the columns of $\boldsymbol{B}$, and the innermost over the summands of the inner product. However, in high-performance numerical algebra, the naive three-loops algorithm is commonly discouraged due to the absence of data reusage, which leads to a poor performance in hardware with a hierarchical cache architecture. Instead, efficient BLAS implementations block the multiplication to improve cache locality (e.g., in Anderson et al., 1999).

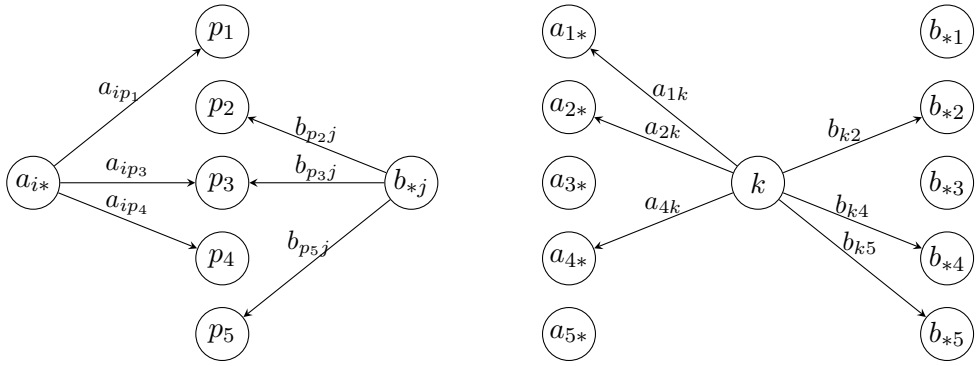
Algorithm 9 Blocked Inner Product Matrix Multiplication (Sequential Implementation)

```
1: procedure DDD_GEMM(DenseMatrix A, DenseMatrix B, DenseMatrix C)
2:     for 0 ≤ I < m step B do                          ▷ for each block row in A
3:         for 0 ≤ J < n step B do                      ▷ for each block column in B
4:             for 0 ≤ P < k step B do                  ▷ for each block step
5:                 for I ≤ i < min{I + B − 1, m} do
6:                     for J ≤ j < min{J + B − 1, n} do
7:                         for P ≤ p < min{P + B − 1, k} do
8:                             C.Val.at(i · n + j) += A.Val.at(i · k + p) × B.Val.at(p · n + j)
```

The blocked, dense matrix multiplication algorithm DDD_GEMM (DDD=*dense matrix* × *dense matrix* → *dense matrix*) is sketched in Algorithm 9, which uses a nested approach with six for-loops. The algorithm is folklore and can be found in various research papers on cache-efficient linear algebra (e.g. Lam et al., 1991).

In contrast to matrix-vector multiplication, dense matrix multiplication is a typical example for a CPU-bound operation. Hence, several tweaks can be applied to the algorithm to further improve its performance, including loop unrolling, pipelining, and vector instructions. Since performance tuning is strongly coupled with the underlying hardware, efficient and parallelized dense matrix multiplication implementations are often provided by the hardware vendors themselves, as part of the BLAS standard library. Consequently, we were not bothered to challenge these highly tuned library methods by an own implementation, in contrast to the sparse multiplication that are rather bound by memory bandwidth. Due to the fact that our dense matrix representation in the column-oriented storage layer as sketched in Figure 3.3b is internally equivalent to a row major-ordered array, our LAPEG just invokes the efficient Intel MKL DGEMM method directly on the primary data structure.

It is worthwhile mentioning that there are recursive algorithms in theory that have a lower complexity than the above $\mathcal{O}(n^3)$ algorithm. Back in the late 1960s, Strassen (1969) presented his



(a) Inner product (b) Outer product

Figure 3.6: Illustration of the inner product vs. outer product formulation for sparse matrix multiplication in the *layered graph model*. The vertices in the leftmost layer depict the row coordinate of matrix $\boldsymbol{A}$. The vertices of the middle layer denote the matching dimension coordinate, which refers to both the columns of $\boldsymbol{A}$ and the rows of $\boldsymbol{B}$. Lastly, the right layer correspond to the column coordinate of $\boldsymbol{B}$. The edges denote non-zero elements of matrix $\boldsymbol{A}$ (left-to-middle layer) and $\boldsymbol{B}$ (middle-to-right layer).

famous algorithm that runs in $\mathcal{O}(n^{2.8074})$. Since then, there has been a lot of related work that is aiming to further reduce the complexity of the dense matrix multiplication. The current lowest complexity for full matrix multiplication is $\mathcal{O}(n^{2.37286})$, based on the algorithm of Coppersmith and Winograd (1990). Indeed, the research to find an even lower bound is yet ongoing (e.g. Le Gall, 2014). In practice, the constants of these algorithms are so high that there are rarely used, particularly not in vendor-tuned BLAS libraries. Nonetheless, there are works (e.g. Valsalam and Skjellum, 2002) that claim a performance improvement over the ATLAS by utilizing a NUMA-aware hierarchical storage format and Strassen's algorithm for multiplying large, dense matrices.

3.6.2 Sparse Matrix-Matrix Multiplication

In contrast to dense matrix multiplication, the inner product formulation is not a good match for sparse matrix multiplication. This can be illustrated with the layered graph model for sparse matrices (e.g., see Buluç and Gilbert, 2010) depicted in Figure 3.6a: imagine the innermost loop of Algorithm 9 would iterate only over all *non-zero* elements of $\boldsymbol{A}$ in row i. Given any non-zero element a_{ip}, it would have first to be checked if its matching "join-counterpart" b_{pj} exists as non-zero element in matrix $\boldsymbol{B}$. Depending on the matrix non-zero pattern it could happen that none of the elements in matrix row $\boldsymbol{a}_{i*}$ match any non-zero element in matrix col $\boldsymbol{b}_{*j}$, but a lot of work has been spent on the search for potential matches.

Therefore, most sparse matrix multiplication methods are based on the outer product formulation instead:

$$\boldsymbol{C} = \boldsymbol{a}_{*1}\boldsymbol{b}_{1*} + \boldsymbol{a}_{*2}\boldsymbol{b}_{2*} + \boldsymbol{a}_{*3}\boldsymbol{b}_{3*} + ..., \tag{3.6}$$

Algorithm 10 Sparse Matrix Multiplication (Sequential Implementation)

```
1:  function SPSPSP_GEMM(CSR Matrix A, CSR Matrix B, CSR Matrix C)
2:      sp_acc = 0                                  ▷ Initialization of the sparse accumulator.
3:      c_iter = 0
4:      for row i ∈ A do
5:          C.RowPtr.at(i) = c_iter
6:          a_row_start = A.RowPtr.at(i)
7:          a_row_end = A.RowPtr.at(i + 1)
8:          for a_row_start ≤ a_iter < a_row_end do
9:              a_col = A.Col.at(a_iter)
10:             b_row_start = B.RowPtr.at(a_col)
11:             b_row_end = B.RowPtr.at(a_col + 1)
12:             for b_row_start ≤ b_iter < b_row_end do      ▷ Accumulate row b_{a_col,*}
13:                 b_col = B.Col.at(b_iter)
14:                 if sp_acc.contains(b_col) then
15:                     sp_acc.at(b_col) += A.Val.at(a_iter) × B.Val.at(b_iter)
16:                 else
17:                     sp_acc.at(b_col) = A.Val.at(a_iter) × B.Val.at(b_iter)
18:                     C.Col.append(b_col)
19:                     c_iter = c_iter + 1
20:         for C.RowPtr.at(i) ≤ i < c_iter do    ▷ Write results back into the result structure.
21:             c_col = C.Col.at(i)
22:             C.Val.append(sp_acc.at(c_col))
```

where each of the summands is a sparse matrix of dimension $m \times n$. The reason can be illustrated with Figure 3.6b: given any non-zero element a_{ik} in column $\boldsymbol{a}_{*k}$ of $\boldsymbol{A}$, it matches with *each* of the non-zero elements in row $\boldsymbol{b}_{k*}$ of matrix $\boldsymbol{B}$. Hence, every non-zero element that is read by the algorithm also contributes to the multiplication. However, the pitfall of a direct implementation of the outer product as of Equation (3.6) is that each of the summands are matrices of the same size as the result. Consequently, they have to be merged in n sparse matrix additions. Such matrix additions can be costly, since the sparse matrix representations have to be "joined" to find all matching non-zero elements that are added. The solution to this problem is to avoid the materialization of the intermediate matrices in Equation (3.6). Instead, the matrix multiplication is processed *row-by-row*, or *column-by-column*.

A method that elegantly solves this problem was presented about thirty years ago by Gustavson (1978). He proposed an algorithm to multiply two matrices that are represented in a CSR format. Algorithm 10 shows the pseudocode of the sequential version of the SPSPSP_GEMM kernel (SPSPSP = *sparse matrix* × *sparse matrix* → *sparse matrix*), which leans on the algorithm by Gustavson. Algorithm 10 is based on the *sparse accumulator* method, which is widely used in sparse matrix algorithms (e.g., Gilbert et al., 1999).

The algorithm is illustrated in Figure 3.7 and works as follows: matrix $\boldsymbol{A}$ is processed row-after-row (outermost loop in line 4), and correspondingly, generates the result matrix $\boldsymbol{C}$ row-after-row.

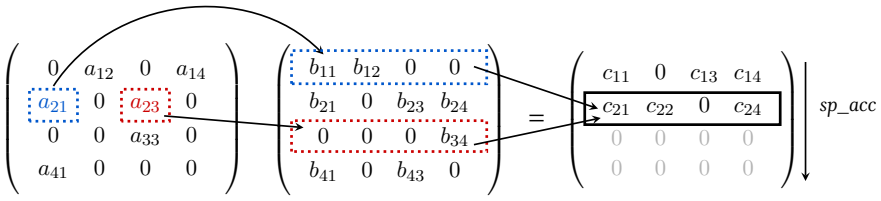


Figure 3.7: Illustration of the Gustavson algorithm on exemplary 4×4 sparse matrices.

This is because the computation of the i-th row of $\boldsymbol{C}$ is entirely determined by multiplying the i-th row of $\boldsymbol{A}$ with matrix $\boldsymbol{B}$. Since $\boldsymbol{A}$ is sparse, the algorithm loops only over the non-zero elements a_{ik} in the i-th row (line 8). For each a_{ik} there are multiple matching elements b_{kj}, which are all contained in the k-th row of matrix $\boldsymbol{B}$ (this corresponds to the outer product of Figure 3.6b). All non-zero elements b_{kj} of the k-th row of $\boldsymbol{B}$ are multiplied with a_{ik} and accumulated into the j-th position of the sparse accumulator (called *sp_acc* in Algorithm 10). The latter acts as a buffer for the i-th matrix row of $\boldsymbol{C}$ that is currently processed or "active". In other words, multiple rows of matrix $\boldsymbol{B}$ are *accumulated* in *sp_acc*, while each contributing row $\boldsymbol{b}_{k*}$ is weighted with the value of a_{ik}. In the original implementation *sp_acc* consists of two dense arrays, one for holding the element values, and the other for keeping track of the row an element belongs to.

When all non-zero elements of the i-th row of $\boldsymbol{A}$ have been processed, the sparse accumulator contains the final row i of matrix $\boldsymbol{C}$. In the next step, all (non-zero) elements of *sp_acc* are written into the i-th row of the $\boldsymbol{C}$ result representation (line 20), which is also CSR. The algorithm proceeds then with the next (i + 1-th) row and reuses the sparse accumulator. In line with the original algorithm, each matrix in our method is represented in the CSR format.

It should be noted that the sparse accumulator multiplication approach is not limited to CSR-represented matrices. A similar methodology is used in column-based CSC multiplication algorithm implementations, e.g. in the sparse multiplication of MATLAB (Gilbert et al., 1992), CombBLAS (Buluç and Gilbert, 2011, 2012), or CHOLMOD (Chen et al., 2008).

Cache efficiency

Consider the innermost loop of Algorithm 10, where *sp_acc* is written in arbitrary order (lines 15 and 17). Clearly, the internal representation of *sp_acc*, and in particular its random write performance, has a significant influence on the algorithm execution time. For reasons of efficiency and minimal overhead, sparse accumulators are often implemented as simple arrays (e.g. in Patwary et al., 2015). This is a reasonable solution for small- and medium-size matrix dimensions. However, if the matrices have very wide dimensions (e.g. $m \gg 10^5$), then the array size might exceed the size of the system caches. In the worst case, every write to *sp_acc* is a cache miss, resulting in an execution time of at least $T = N_{\text{nz}}^{C} \times \tau$, where τ = the cache miss latency. To improve the cache locality of the sparse accumulator in case of large and hypersparse matrices, we replace the array by a hash table. The latter is designed such that it minimizes the hashing overhead and fits in L2 memory. We developed, implemented, and successfully evaluated this approach together with Luben Alexandrov as part of his master thesis (Alexandrov, 2014).

3.6.3 Mixed Matrix Multiplication

Besides dense-only (DDD_GEMM) and sparse-only (SPSPSP_GEMM) multiplication, it frequently occurs that a sparse matrix is multiplied with a dense matrix. To name an example we recall the approximate SVD method (Algorithm 3), where the sparse matrix $\boldsymbol{A}$ is multiplied with a dense, random matrix $\boldsymbol{\Omega}$. Obviously, neither of the algorithms described above is suited for this situation. Moreover, the representation of the result matrix can as well be dense or sparse. Thus, there is a "combinatorial space" of different multiplication methods, which each differ in their composition of matrix storage types. We use the notation XYZ_GEMM to denote a multiplication kernel, where X is the left-hand input-, Y is the right-hand input type and Z the output matrix storage type, which can be either sparse (SP) or dense (D). Remember that GEMM is the common notation for the general matrix multiply routine, as used in the BLAS standard (Dongarra et al., 2001).

Considering all combinations of dense and sparse matrices, there are $2^3 =$ eight different multiplication kernel routines:

SPSPSP_GEMM	*sparse* × *sparse* → *sparse*	DSPSP_GEMM	*dense* × *sparse* → *sparse*
SPSPD_GEMM	*sparse* × *sparse* → *dense*	DSPD_GEMM	*dense* × *sparse* → *dense*
SPDSP_GEMM	*sparse* × *dense* → *sparse*	DDSP_GEMM	*dense* × *dense* → *sparse*
SPDD_GEMM	*sparse* × *dense* → *dense*	DDD_GEMM	*dense* × *dense* → *dense*

In fact, the BLAS standard only offers a DDD_GEMM and a SPDD_GEMM routine in the sparse extension. As mentioned in Section 2.4.3, the limited functionality of sparse BLAS merely consists of a sparse triangular solver and the SPDD_GEMM kernel. Dongarra et al. (2001) justify their selection by stating that these "are the primary computational kernels in many sparse linear equation and eigensystem solvers". However, based on our experiences and application studies, we oppose this opinion. Instead, we underline the importance of providing a complete set of multiplication primitives. Our argumentation is backed by the following reasons: first, Dongarra et al. (2001) are ignoring sparse × sparse matrix multiplications, which in fact is a widely used computational kernel, for instance in the document clustering as of Section 2.3.1, or in graph algorithms such as all-pairs shortest path (Kepner and Gilbert, 2011; Buluç and Gilbert, 2012). Second, data scientists generally favor a linear algebra environment where they can design and develop new analysis algorithms. These algorithms might involve multiplications of matrices of arbitrary type and in an arbitrary order. Moreover, some applications contain chain multiplications of both dense and sparse matrices, for example the randomized SVD method (Alg. 3) described in Section 2.3.1. Finally and ultimately, the existence of a complete set of multiplication kernels will be required by the expression optimizer, which we discuss in Chapter 4, as well as the adaptive tile multiplication (ATMULT) that is described in Chapter 5. In particular, ATMULT decomposes a large sparse matrix into sparse and dense subparts. As a consequence, the multiplication contains a series of mixed sparse and dense sub-multiplications.

In the LAPEG we implemented the following multiplication kernels: SPSPSP_GEMM, SPSPD_GEMM, SPDD_GEMM, DSPSP_GEMM, and DSPD_GEMM. Our mixed sparse/dense multiplication kernels are designed in a straightforward way. Simply put, they mix fractions of the dense algorithm (Alg. 9) with the sparse algorithm (Alg. 10), depending on which of three matrices are sparse or dense, respectively.

Algorithm 11 Mixed Dense-Sparse into Dense Matrix Multiplication (Seq. Implementation)

```
1: function DSPD_GEMM(DenseMatrix A, CSR Matrix B, DenseMatrix C)
2:     for 0 ≤ i < m do
3:         for 0 ≤ p < n do
4:             if A.Val.at(i · k + p) ≠ 0 then
5:                 b_row_start = B.RowPtr.at(p)
6:                 b_row_end = B.RowPtr.at(p + 1)
7:                 for b_row_start ≤ b_iter < b_row_end do
8:                     b_col = B.Col.at(b_iter)
9:                     C.Val.at(i · n + b_col) += A.Val.at(i · k + p) × B.Val.at(b_iter)
```

For instance, Algorithm 11 sketches the DSPD_GEMM kernel, where matrices $\boldsymbol{A}$ and $\boldsymbol{C}$ are present in a dense format, and matrix $\boldsymbol{B}$ in a sparse CSR representation. As mentioned before, it can be beneficial to store matrices as dense representation even if they have a considerable fraction of zero values (e.g., 50%). To skip the loop body execution in Algorithm 11 for zero-valued matrix elements of $\boldsymbol{A}$, we included an additional check for zero values (line 4). However, if the matrix is still 100% populated, the branch prediction of the CPU should eliminate the overhead of the additional check.

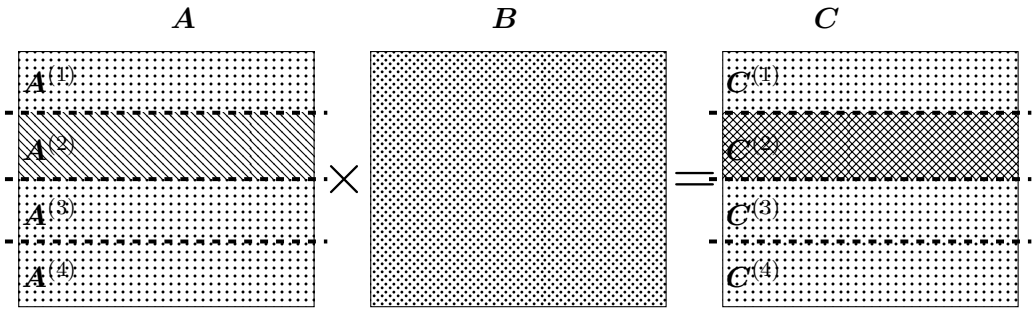
All remaining kernels are composed accordingly; kernels that produce a sparse result matrix $\boldsymbol{C}$ use the sparse accumulator approach, and kernels producing a dense result write directly in the target dense matrix representation (of course, concurrent worker threads write in disjoint memory sections, as illustrated in Section 3.7). However, we will not discuss each multiplication kernel in detail and omit further pseudocode for reasons of clarity.

3.7 PARALLELIZATION

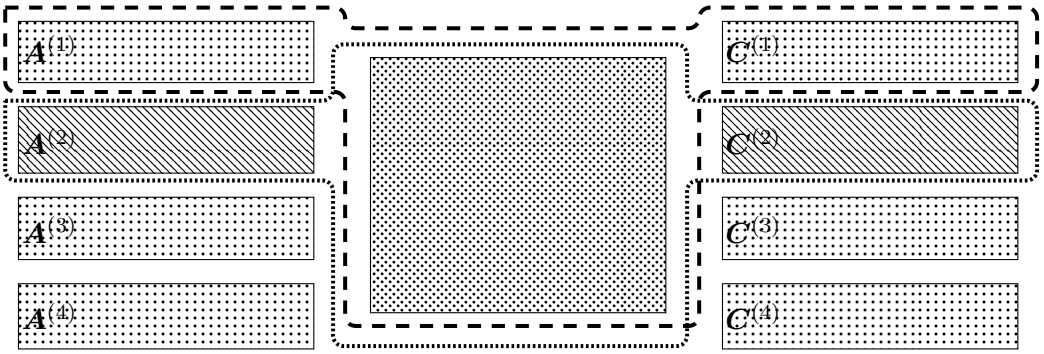
We already discussed the parallelization approach for the matrix-vector multiplication kernels in Algorithms 9 to 11, which is mainly based on parallelizing the for loop (as indicated in the code comments), and depending on the algorithm, by introducing a thread-local intermediate result vector $\boldsymbol{y}$.

The parallelization of the matrix-matrix multiplication is more complex. Hence, the pseudocodes for Algorithms 9 to 11 only contain the sequential version for the sake of simplicity. Therefore, we now discuss the general parallelization scheme, which we use in all of our kernel implementations. Note that in the context of a main-memory machine, we only consider shared-memory parallelism. Algorithms based on distributed systems are not discussed in the scope of this thesis.

Similar to the parallelization of the matrix-vector multiplication, the parallelization of each of our kernels is primarily realized by splitting the outer for-loop over the rows of matrix $\boldsymbol{A}$ (e.g., line 4 in Algorithm 10). From a more conceptual perspective, we effectively parallelize the algorithm by separating $\boldsymbol{A}$ into multiple, disjunct row chunks $\boldsymbol{A}^{(i)}$, which form independent work units that are processed in parallel. Figure 3.8a sketches the access pattern of a partitioned matrix multiplication, and Figure 3.8b shows the separation into the work units. The advantage of the row-centric partitioning is that each matrix row i of $\boldsymbol{A}$, multiplied with (potentially the complete) matrix $\boldsymbol{B}$, creates



(a) Access pattern of a row-partitioned matrix multiplication. The pattern indicates which parts are involved in the calculation of $\boldsymbol{C}^{(2)}$: $\boldsymbol{A}^{(2)}$ and the full matrix $\boldsymbol{B}$



(b) Separation in independent work units. The surrounding paths denote the input and output matrices of the two exemplary work units 1 and 2.

Figure 3.8: Parallelization approaches for row-based matrix multiplication algorithms.

the corresponding row i in the result $\boldsymbol{C}$. Hence, each work unit will produce a disjunct chunk $\boldsymbol{C}^{(i)}$ of the result matrix.

In case of a dense result matrix, the threads just write into the disjunct memory locations of the previously created dense matrix representation of $\boldsymbol{C}$. In contrast, if the target matrix is in a sparse layout, each of the disjunct part results $\boldsymbol{C}^{(i)}$ have a CSR layout, which can not be trivially appended. Instead, an additional step is required to assemble the row chunks together into a final CSR result representation. This step is done in a separate parallelized procedure.

It should be mentioned that each thread, which is processing any of the working units (i), potentially reads the complete matrix $\boldsymbol{B}$. At least, it has to read all rows that are required in the multiplication with $\boldsymbol{A}^{(i)}$. In a naive shared-memory parallel model, where each thread is assumed to have equal access times to any memory location, this should not have any impact on the performance. In practice, however, blocking of the dense or sparse matrix $\boldsymbol{B}$ yields a far better scaling behavior, due to an increased cache locality. In particular we observed substantial performance improvements by using our tiled data structure in the evaluation of Chapter 5. This observation was also made by Patwary et al. (2015), who used a columnar blocking of $\boldsymbol{B}$ (and simultaneously $\boldsymbol{C}$) to improve the cache locality of both reads and writes. Furthermore, on modern multi-core machines, memory read latencies for different physical cores diverge due to effects of non-uniform memory access (NUMA). A main-memory system with many memory sockets behaves somewhat

similar to a distributed multi-host landscape. In a distributed environment with communication overhead, different aspects come into place: Buluç and Gilbert (2010) found that parallel sparse matrix multiplication using an 1D partitioning of matrix $\boldsymbol{A}$ (as of Figure 3.8b) fails to scale, whereas a 2D partitioning exhibits a superior, scalable asymptotic behavior.

We revisit the topic of matrix multiplication in connection with our adaptive tile matrix data structure in Chapter 5. Therein, we present a tile-based multiplication algorithm, which will be discussed with respect to parallel scheduling, cache-efficiency, and NUMA-partitioning.

3.8 SUMMARY

In this chapter, we have set the foundations for the following chapters of this thesis. In particular, we introduced the column-oriented storage engine of the main memory system, on which the Lapeg is based on. Moreover, we presented ways to represent dense and sparse in a columnar DBMS that are significantly more efficient than the alternative approaches of related work (Zhang et al., 2009). Furthermore, we outlined the analogy between efficient sparse matrix formats and clustering as well as indexing relational tables in column-stores.

We surveyed different strategies of multiplying dense and sparse matrices with vectors and other matrices based on the column-oriented storage layer matrix data structures, and introduced basic parallelization constructs. In particular, we emphasized the importance of having a full set of mixed sparse- and dense matrix multiplication kernels. In fact, these are not provided by most numerical libraries, e.g. the NIST sparse Blas(NISTb, n.d.), limiting their applicability for flexible, matrix-based ad-hoc data analysis.

The different multiplication kernels and especially their runtime behavior are again a topic of Chapter 4, where we discuss the optimization of arbitrary multiplication expressions. Furthermore, based on these kernels we present a sophisticated matrix multiplication operator in Chapter 5, which exploits heterogeneous tile multiplications, dynamic task scheduling, and cost-based runtime optimizations. We also discuss related work about advanced matrix storage formats in greater detail in Section 5.6.

4 EXPRESSION OPTIMIZATION

The optimizer forms the fundamental building block of the logical layer in the Lapeg (Figs. 1.1 and 4.2). For each linear algebra expression there might be multiple execution plans, and only few execute the requested operations in an efficient, optimal way. Similar to that of a relational execution engine, the Lapeg generates a plan that is composed of physical plan operators, including multiplications and storage type transformations of matrices. This chapter is devoted to the optimizer of the Lapeg, with focus on the optimization of matrix multiplication expressions by considering physical properties of matrices and matrix algorithms.

4.1 MOTIVATION

Many workflows contain one or more linear algebra expressions that are computationally expensive. In this chapter, we focus on optimizing those expressions that amount for the major part of the analysis execution time. Therefore, we recall the randomized SVD method described in Section 2.3.1: the core computation in Algorithm 3 is a matrix chain multiplication of multiple instances of the sparse term-document matrix with a dense, Gaussian random matrix.

Matrix chain multiplications occur in a variety of other applications. Examples are transitive closure computations, Markov chains (Yegnanarayanan, 2013), linear transformations (Edelman et al., 1994), linear discrete dynamical systems (Feng, 2002), or multi-source, multi-level breadth first search (Kepner and Gilbert, 2011). In fact, an efficient execution of sparse matrix chain multiplications is nontrivial, especially if intermediate result matrices become dense. In many situations the runtime performance can be significantly improved by changing the execution order of the expression, or by switching from a sparse to a dense multiplication algorithm during execution.

The Density of Intermediates. To illustrate the problem of optimizing the execution of sparse matrix expressions, we present a brief experimental study of the graph analysis scenario introduced in Section 2.3.2. The breadth-first search algorithm can be rewritten by unrolling the main loop (line 5 in Algorithm 4), and assembling the matrix-vector multiplications into a single multiplication expression. The resulting expression yields:

$$\boldsymbol{y}^{\mathrm{T}} = (\ldots(((\boldsymbol{x}^{\mathrm{T}} \underbrace{\boldsymbol{G})\boldsymbol{G})\boldsymbol{G}}_{\text{search depth}})\ldots\boldsymbol{G}). \tag{4.1}$$

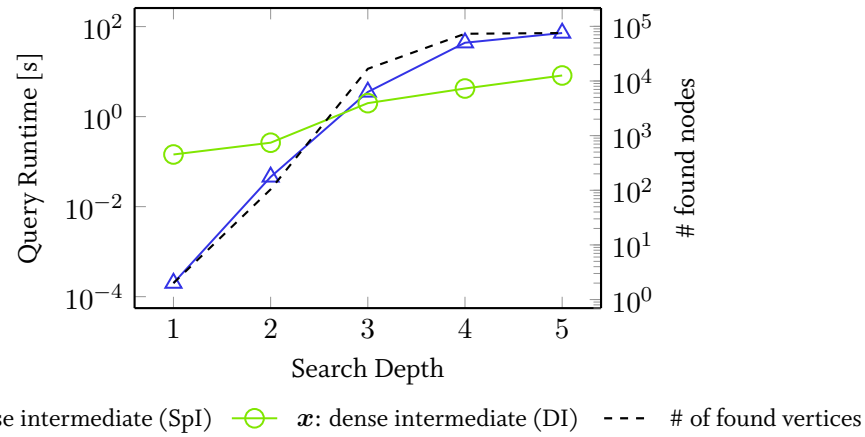


Figure 4.1: Comparison of the execution duration of the breadth-first search (Equation 4.1) on a social network graph by using a sparse (SpI) and a dense (DI) intermediate structure. The x-axis denotes the depth parameter of Algorithm 4. The adjacency matrix is stored in a CSR format.

Indeed, Equation 4.1 is a matrix chain multiplication of the sparse $1 \times n$ matrix $\boldsymbol{x}^{\mathrm{T}}$, which is effectively a row vector, with d (search depth) instances of the sparse adjacency $n \times n$ matrix $\boldsymbol{G}$.

We now illustrate the impact of densities and data structures on the execution time when expression (4.1) is executed *naively* from left to right by using the outer product sparse matrix-vector multiplication algorithm (Algorithm 7 in Section 3.5.2.) In this small experiment depicted in Figure 4.1, we used two different versions of the intermediate result vector; a dense vector representation (DI) as shown in Figure 3.2a, and a sparse one (SpI) as of Figure 3.2b. Furthermore, we took the adjacency matrix ($\boldsymbol{G}$) of a social network graph[1] and varied the search depth parameter d, which corresponds to the multiplication chain length and is denoted along the x-axis of the plot.

In the initial phase of Algorithm 4 the vector $\boldsymbol{x}$ is only filled with a single element, which represents the start vertex ID. With every multiplication, new vertices are found, i.e., more non-zero elements are added to the intermediate result vector. Hence, the density of the intermediate vector increases, which has a considerable impact on the multiplication runtime. Furthermore, it reveals the different complexity between the DI and the SpI version. The DI implementation of the multiplication iterates over every matrix entry (zero- and non-zero-valued), whereas the SpI implementation only iterates over non-zero elements. As a consequence, the DI approach performs worse for small depths than the SpI one. However, with increasing search depth the $\boldsymbol{x}$ vector density reaches a turning point at which the DI-based implementation becomes more efficient. The exact value of this point depends on the details of the respective implementation. In our experiment, it is reached between depth two and three for the social graph Gra1. In the analogous measurement on Gra2, the turning point is at a considerably larger depth, which exceeds the x-range of the plot.

From this experimental study it can be concluded that switching internally from a sparse to a dense representation of vector $\boldsymbol{x}$ at a certain density clearly offers a tuning opportunity, which should be exploited by an optimizer. In fact, this example only shows one aspect of optimization, namely the internal representation of intermediate results. Other aspects that can affect the run-

[1] More characteristics of the graph data sets are displayed in Table A.2.

time include parenthesization and execution order, which we discuss in more detail below and which are both addressed by our optimizer.

The problem is closely related to join enumeration in relational algebra, where the optimal join order and the selection of join algorithms is largely determined by the cardinality of intermediate results. Since it is usually impossible for database users to manually overview the cardinalities of intermediate relations, the RDBMS takes over this task by generating an optimized execution plan based on cardinality estimates and physical properties of the system. Likewise, data scientists are usually not familiar with algorithmic details of multiplication kernels and system parameters, and do not have a profound knowledge of the characteristics of their matrices. Hence, it makes sense to leave these decisions as well to the system.

Despite the number of systems that offer a matrix-based language interface (some of which have been presented in Section 2.5ff,) only little has been done in the direction of optimizing matrix expressions. Our idea is the following: similar to the RDBMS optimizer that creates optimized execution plans from SQL expressions, we propose an optimizer component that generates an optimal execution plan for linear algebra. As illustrated in Figure 4.2, this component forms the foundation of the logical layer of the Lapeg, and works on top of the physical storage engine. The latter accommodates matrices in the native column-oriented storage layer, using the data structures and algorithms that were presented in Chapter 2.

The optimizing component comprises the following integral parts:

- **SpMachO**, a general matrix chain multiplication optimizer based on a dynamic programming approach, which leverages density estimations of intermediate results and different multiplication kernels to minimize the total execution runtime. The optimization problem and the SpMachO algorithm are presented in Section 4.2.

- **A comprehensive cost model**, which is used by SpMachO to determine the costs of subchain multiplications. It is derived from the number of memory accesses and floating point operations of the different matrix multiplication kernels that were introduced in Section 3.6. The cost model is described in Section 4.3.

- **SpProdest**, a sparse matrix density estimator, which predicts the density structure of intermediate and final result matrices, by using a novel skew-aware stochastic density propagation method. It is described in detail in Section 4.4.

Moreover, we present an extensive evaluation and comparison of the execution runtime of SpMachO-generated plans against alternative execution strategies and other numerical algebra systems in Section 4.5. Finally, we will discuss related work in Section 4.6, followed by the summary in Section 4.7.

4.2 EXPRESSION OPTIMIZATION

We start by discussing what aspect of the execution influences the execution time the most. Consider the set of linear algebra expressions that consist of multiplications and additions of general $\mathcal{R}^{m \times n}$ matrices. Further operations, like subtraction or division by a matrix $\boldsymbol{A}$ can be represented

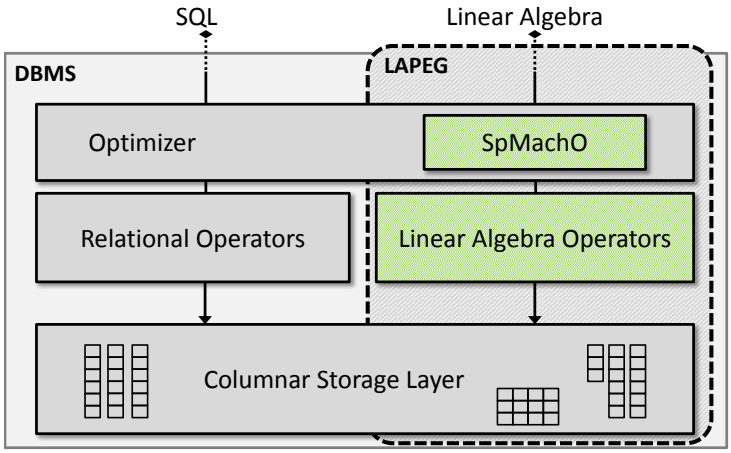


Figure 4.2: Example architecture of the DBMS-integrated linear algebra processing engine (LAPEG) with the optimizer component (SPMACHO.)

by using the corresponding inverse of addition $(-\boldsymbol{A})$, or inverse of multiplication $\boldsymbol{A}^{-1}$, respectively. Thus, let the expression be of a form:

$$\boldsymbol{C} = \boldsymbol{A}_1 \cdot \boldsymbol{A}_2 \cdot ... \cdot \boldsymbol{A}_p + \boldsymbol{A}_{p+1} \cdot ... \cdot \boldsymbol{A}_l + ... \quad \boldsymbol{A}_i \in \mathcal{R}^{m_i \times m_{i+1}}, \tag{4.2}$$

In general, the mathematical order of operations follows the rule *multiplication precedes addition*, given that there is no prioritizing parenthesization. Consequently, expression (4.2) is separable into the multiplication chains

$$\tilde{\boldsymbol{A}}_1 = \boldsymbol{A}_1 \cdot ... \cdot \boldsymbol{A}_p, \quad \tilde{\boldsymbol{A}}_2 = \boldsymbol{A}_{p+1} \cdot ... \cdot \boldsymbol{A}_l \quad \boldsymbol{A}_i \in \mathcal{R}^{m_i \times m_{i+1}}, \tag{4.3}$$

which are executed first. In a second step, the additions of the intermediate results $\tilde{\boldsymbol{A}}_1 + \tilde{\boldsymbol{A}}_2 + ...$ follow.

In fact, the additions in the computation of $\boldsymbol{C}$ are cheap compared to the multiplications. Since most of the execution time is spent in multiplication, we proceed in the following sections with the optimization of matrix chain multiplications. This problem is well-solved for dense-only matrices, but not for expressions that contain sparse, or sparse and dense matrices. We first introduce the known, textbook-approach for dense matrices, and then present our novel, density-aware optimization approach.

4.2.1 Density-Agnostic Matrix Chain Optimization

The algebraic degree of freedom to execute a matrix chain multiplication consists in the setting of parenthesis, since matrix multiplications are associative, i.e.

$$(\boldsymbol{A}_1 \cdot \boldsymbol{A}_2) \cdot \boldsymbol{A}_3 = \boldsymbol{A}_1 \cdot (\boldsymbol{A}_2 \cdot \boldsymbol{A}_3). \tag{4.4}$$

Altering the parenthesization does not change the result, but the number of operations required in the computation of the complete chain can vary significantly. The problem of finding the optimal parenthesization for a dense, non-square matrix chain multiplication is well understood and serves

as a text book example for the use of dynamic programming (e.g. in Godbole, 1973; Cormen et al., 2001). The idea of the dynamic programming approach is to iteratively construct the optimal parenthesization as a combination of optimal sub-parenthesizations. Therefore, one minimizes a cost recurrence expression with respect to the split point k. The recurrence can be written as

$$C_{\pi^{\text{B}}(ij)} \quad = \quad \min_{i \leq k < j} \left\{ C_{\pi^{\text{B}}(ik)} \quad + \quad C_{\pi^{\text{B}}((k+1)j)} \quad + \quad \texttt{TM}\left(\boldsymbol{A}_{[i\ldots k]}, \boldsymbol{A}_{[k+1\ldots j]}\right) \right\}, \quad (4.5)$$

where we use the following notation:

- C_π: the cost for executing the matrix chain multiplication, given a certain parenthesization π.
- $\pi(ij)$: a parenthesization for the matrix (sub)chain $\boldsymbol{A}_{[i\ldots j]}$
  $\pi^{\text{B}}(ij)$ denotes the optimal ("best") one out of all possible parenthesizations.
- $\texttt{TM}\left(\boldsymbol{A}_{[i\ldots k]}, \boldsymbol{A}_{[k+1\ldots j]}\right)$: the cost function for multiplying the two matrices that result from the subchains $\boldsymbol{A}_{[i\ldots k]}$ and $\boldsymbol{A}_{[k+1\ldots j]}$

In the textbook case, the costs are set equal with the number of floating point operations for multiplying two dense intermediate matrices. For the classical dense inner product algorithm (Alg. 9), the number of floating point multiplications can be exactly determined a priori as

$$\texttt{TM}\left(\boldsymbol{A}_{[i\ldots k]}, \boldsymbol{A}_{[k+1\ldots j]}\right) = m_i m_{k+1} m_{j+1}, \quad (4.6)$$

where $m_i \times m_{k+1}$ is the dimension of the chain result $\boldsymbol{A}_{[i\ldots k]}$, and $m_{k+1} \times m_j$ the dimension of $\boldsymbol{A}_{[k+1\ldots j]}$.

4.2.2 Density-Aware Matrix Chain Optimization

By restricting the cost model in Equation (4.5) to dense-only matrices, the applicability of the textbook-approach is limited in practice. In fact, many of the real-world matrices in big data environments are sparse. A sparse $m \times n$ matrix is not only defined by its row and column dimensions, but also by the number N_{nz} of non-zero elements as well as the non-zero pattern (the distribution of non-zero elements in the matrix.)

In order to quantify the sparsity or non-zero population of a matrix, we use a measure that relates N_{nz} to the absolute matrix dimensions: the *matrix density*:

$$\rho = \frac{N_{nz}}{mn} \quad (4.7)$$

Most of the related work on matrix chain multiplications considers dense-only multiplications (Godbole, 1973; Cormen et al., 2001). However, we identify two major arguments why the sparsity of matrices has a significant influence on the optimization:

Efficient Sparse Multiplication kernels. As a matter of fact, the complexity of multiplying two sparse matrices differs significantly from the naive inner product algorithm for dense matrices. By a swift look at our sparse matrix algorithm (Alg. 10), one quickly concludes that multiplication costs rather depend on the number of non-zero elements in the CSR representation than on the matrix dimensions. Consequently, the cost model TM in the optimization recurrence (Eq. 4.5) has to be changed in a way that it takes the matrix density into account.

Variation of Intermediate Densities. The dense-only approach is agnostic to the fact that densities of intermediate result matrices can *vary* significantly from the initial matrix densities. For example, the density of the result matrix $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ can be much higher, or even less than that of both $\boldsymbol{A}$ and $\boldsymbol{B}$ (see Figure 4.8). Despite the asymptotic complexity, it is in many cases more efficient to use a dense algorithm, even if the matrix contains a significant fraction of zero elements. This can be reasoned with the efficient and well-tuned implementations of dense matrix multiplication kernels in the BLAS library, which have low constants in their cost function.

Our idea is to take the individual characteristics of the different matrix representations and multiplication kernels into account. In particular, we exploit the potential performance benefits from changing the physical implementation of the initial matrices or intermediate results. Therefore, we construct an expression execution plan that may contain dense, sparse, and mixed dense/sparse matrix multiplications. Furthermore, the execution plan could include conversions from a sparse into a dense representation. To generate the plan, we adopt the idea of dynamic programming and modify it in such a way that it incorporates the physical properties of the matrices: the recurrence (4.5) is extended by adding input and output storage types as independent dimensions, and including additional cost functions for the storage type conversions:

$$C_{\Pi^{\mathrm{B}}(ij)S^o} = \min_{\substack{i \le k < j \\ S^l, S^r, S^1, S^2 \in \mathcal{S}}} \Big\{ C_{\Pi^{\mathrm{B}}(ik)S^l} + C_{\Pi^{\mathrm{B}}((k+1)j)S^r} + \mathsf{TT}_{S^1}\left(\boldsymbol{A}_{[i..k],S^l,\rho^l}\right) + \mathsf{TT}_{S^2}\left(\boldsymbol{A}_{[k+1..j],S^r,\rho^r}\right) + \mathsf{TM}_{S^o}\left(\boldsymbol{A}_{[i..k],S^1,\rho^1}, \boldsymbol{A}_{[k+1..j],S^2,\rho^2}\right) \Big\} \quad (4.8)$$

The terms in Equation (4.8) are defined as follows:

- $\Pi(ij)$: execution plan for a matrix (sub)chain multiplication. It contains the multiplication execution order, as well as all storage transformations. Π^{B} denotes the optimal plan.
- S^X: storage type, which is either dense or sparse. The superscript X labels each of the *five* matrix "checkpoints" that are considered per execution node, as illustrated in Figure 4.3. An execution node comprises one multiplication operation, and potentially two preceding

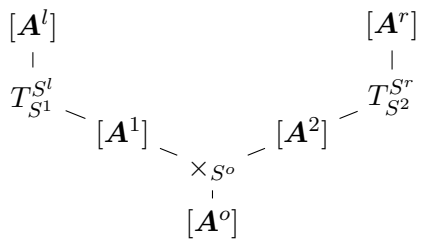


Figure 4.3: The matrix checkpoints in a multiplication execution node.

transformations of the left and/or the right input matrices. Hence, the checkpoints labeled with X are the following: l: left subplan output-, r: right subplan output-, 1: left input-, 2: right input-, o: execution node output matrix.

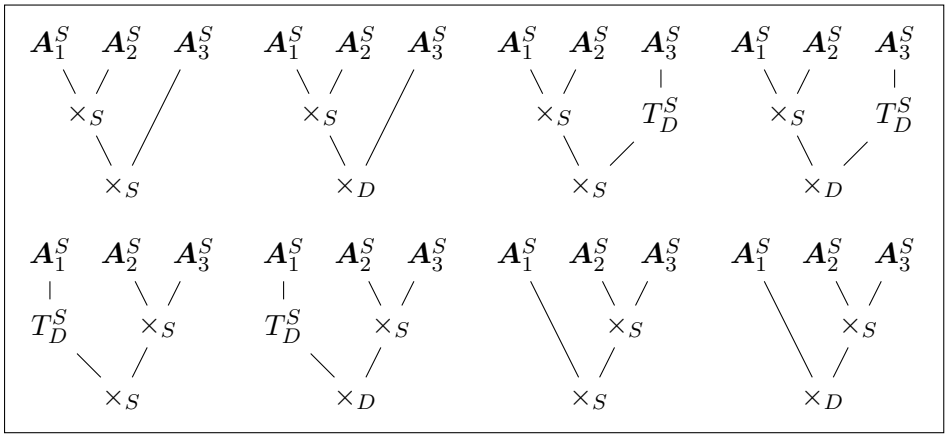


Figure 4.4: Eight of the possible 128 execution plans for the multiplication of three sparse matrices $\boldsymbol{A}_1 \cdot \boldsymbol{A}_2 \cdot \boldsymbol{A}_3$. $\times$ are multiplication operators, and T are storage type transformations of matrices. The super-/subscripts denote the input/output matrix representation of the respective operator: S/D denotes the internal *sparse*/*dense* representation.

- ρ^X: Density of intermediate matrix at checkpoint X.
- $\mathsf{TT}_{S^Y}\left(\boldsymbol{A}_{[i..k],S^X,\rho^X}\right)$: cost function for the conversion of a matrix from type S^X into type S^Y. The costs are zero if the storage type S^X is equal to S^Y.

The cost functions TM in Equation (4.8) not only depend on the densities, but also on the storage types of the input and output matrices. Since single-pass storage type conversions are usually less costly than multiplications, it could be beneficial to convert a matrix from one into the other representation prior to the multiplication. Hence, besides the parenthesization split point k, we vary the input and output storage types for each step in the recurrence. For example, if the initial matrices are in a sparse representation, and the dense multiplication kernel plus the conversion has a far lower cost than the sparse multiplication, then they are first converted into the dense representation. The value of the conversion cost $\mathsf{TT}(\cdot)$ equals zero for identity transformations. Thus, if a matrix is already in the optimal representation, no transformation is performed.

Some parameters that contribute to the cost functions $\mathsf{TM/TT}(\cdot)$ are not known prior to the execution and have to be estimated, for instance the density ρ of intermediate results. Therefore, we developed the sparse matrix product density estimator (SpProdest), which is described in Section 4.4. The final costs derived from the recurrence in Equation (4.8) are minimal, given that the estimated costs encoded in TM and TT are determined precisely. In particular, the optimality or *goodness* of our sparse matrix chain optimizer (SpMachO) depends on two aspects that potentially contain uncertainties: first, the accuracy of the quantitative cost model of the multiplication kernels, and second, the precision of the density estimates provided by SpProdest.

The total number of the possible execution plans using our model (Eq. 4.8) for a matrix chain multiplication of length p is

$$\mathcal{C}_{p-1} \cdot 2^{3(p-1)}, \tag{4.9}$$

where $\mathcal{C}_{p-1}$ denotes the Catalan number $\mathcal{C}_n = \frac{(2n)!}{(n+1)!n!}$.

$\mathcal{C}_{p-1}$ reflects the number of possible parenthesizations, which is the same as for the textbook case (Cormen et al., 2001). The second factor is related to the 2^3 storage type combinations {left input type, right input type, output type} that are connected with each of the $p-1$ type multiplication nodes. The number in (4.9) equals the size of the plan search space, which is growing exponentially with the length p. To give an example, it yields 2560 for a matrix chain of length $p = 4$, and already 1,376,256 for $p = 6$. Some of the possible plans for $p = 3$ are shown in Figure 4.4. Like in the dense-only case, we solve the dynamic programming recurrence (4.8) in $\mathcal{O}(p^3)$ time using a bottom-up approach.

4.2.3 Sparse Matrix Chain Optimizer

We implemented the dynamic programming solver for the optimization of matrix chain multiplications in the LAPEG, using the column-oriented storage layer-based data structures and algorithms that we introduced in Chapter 3.

Algorithm 12 SPMACHO

```
1:  function spMacho(MatrixChain A_[1..p], TM, TT)
2:      ρ̂[][] ← spProdest(A_[1..p])
3:      for 1 ≤ j < p, j > i ≥ 0 do
4:          for i ≤ k < j do
5:              for types ∈ {sparse, dense} do
6:                  if !chkMemLimit(A, k, ρ, types) then
7:                      continue
8:                  q ← TT(A_[i..k..j], ρ̂[][], types)
9:                  q ← q + TM(A_[i..k..j], ρ̂[][], types)
10:                 if q < cost[i][j][S^o] then
11:                     cost[i][j][S^o] ← q
12:                     plan[i][j][S^o] ← k, types
13:     if cost[1][p][·] ≥ MAXVAL then
14:         /* memory exceed exception */
15:     else
16:         return min(plan[1][p][sparse], plan[1][p][dense])
```

The pseudocode of SPMACHO is sketched in Algorithm 12. The cost of the optimal sub-chain multiplications and the relevant plan information (all split points and storage types per sub-chain) are cached in three-dimensional array structures. For each combination in the inner loop, the method CHKMEMLIMIT checks if the total memory consumption of the matrices and intermediate results in the current plan configuration would exceed the system limit. The memory required for dense $m \times n$ matrices is $\mathcal{O}(mn)$, and $\mathcal{O}(N_{nz} = mn\rho)$ for sparse matrices. Since SPMACHO optimizes the runtime performance, the plan may contain conversions from sparse into dense matrix representations whenever the dense kernel leads to a lower overall runtime. However, the conversions into dense representations potentially increase the memory consumption compared to a

sparse-only plan. Our strategy is that every conversion is allowed, as long as the total memory consumption at every point in time does not exceed a hard memory limit. Execution paths that would exceed the memory limit are automatically skipped (line 6). We assume that there is at least one plan that does not exceed the memory limit. If not, SpMachO returns with an exception (line 14). Finally, the resulting plan, which can be converted into a directed acyclic graph representation as of Figure 4.4, is returned to the system for execution.

As mentioned before, the complexity of Algorithm 12 is $\mathcal{O}(p^3)$, which can be derived from the dynamic programming loop and is the same as in the dense-only problem (Cormen et al., 2001). The additional complexity by the introduction of storage type transformations yields a constant factor, since the inner loops over the storage types do not depend on the chain length p. A few pruning methods could be applied to reduce the execution plan space, for example, by excluding that the product of two dense can be sparse. However, this does not lower the asymptotic complexity of the algorithm.

The Lapeg executes the plan using the corresponding transformation and matrix multiplication operators. While the unary transformation operator either performs a dense-to-sparse or a sparse-to-dense storage transformation, the multiplication is delegated to one of the eight different multiplication kernels (xyz_gemm), which we introduced in Chapter 3. As of the current status, we provide our own implementation of all kernels except for ddd_gemm, for which our Lapeg calls the efficient Intel MKL implementation via its Blas interface. However, we remark that the dependency on the physical representation and particular algorithm implementation is solely encoded in the cost functions. Since the cost model is orthogonal to the general optimization approach, we discuss it separately.

4.3 EXECUTION TIME COST MODEL

In order to obtain the optimal execution plan by solving the recurrence in Equation (4.8), the cost model and its corresponding parameters have to be determined accurately for each multiplication kernel. Since the actual runtime depends on many parameters and external influences, an exact determination cannot be guaranteed. However, even for small variations, the plan generated by SpMachO is still near-optimal, which is confirmed in the evaluation in Section 4.5.

The cost for multiplying two dense or sparse matrices generally depends on the matrix dimensions m, k, n, the number and distribution pattern of non-zero elements of input and result matrices, and the implementation details of the corresponding kernel algorithm. The idea is to reduce these parameters into a set of only a few significant dimensions, and create the cost model based on this reduced parameter space.

Our first approximation to simplify the model is to let the runtime of a single multiplication only depend on the total number of the matrices' non-zero elements rather than their individual non-zero pattern. Indeed, the pattern might influence the memory access times of the algorithm; for example, skewed patterns with local clusters of a high non-zero element concentration yield a better cache utilization for *sp_acc*-writes in Algorithm 10. However, related work (Alexandrov, 2014) has shown that cache effects influence the performance of sparse matrix multiplication by less than 20% even for very skewed patterns. Thus, we incorporate the *average* memory read/write-time into our model coefficients that are learned from the training data, which contain multiple

matrix instances with varying non-zero distributions. The evaluation of the cost model (Fig. 6) confirms that this approximation has a negligible impact on the model validity. Consequently, our simplified parameter space has the dimensions m, k, n, ρ_A, ρ_B, and $\hat{\rho}_C$, which corresponds to the product density estimation, which is determined by SpProdest.

As an example, we examine the spspsp_gemm kernel as of Algorithm 10. To not repeat the algorithm description in detail, we only sketch the derivation of the cost function for spspsp_gemm. Compared to other kernels that are based on dense representations, the cost of spspsp_gemm is rather tightly coupled with the density estimates, due to its clear dependency on the number of non-zero elements.

For reasonably large sparse matrices, the runtime of sparse kernels is often dominated by main memory bandwidth (Dalton et al., 2015). Hence, mainly the memory accesses contribute to the runtime of an algorithm, which is in conformance to the external memory (I/O) model. For spspsp_gemm, the access pattern can be formulated as

$$\begin{aligned}\texttt{TM}(m,k,n,\rho_A,\rho_B) \propto m \cdot (&\mathit{read}_{\text{A.RowPtr}} + k\rho_A \cdot \\ &(\mathit{read}_{\text{A.Col,B.RowPtr}} + n\rho_B \cdot (\mathit{read}_{\text{B.Col,A.Val,B.Val}} + \mathit{write}_{\mathit{sp_acc}})) + n\hat{\rho}_C \cdot \text{write}_{\text{C.Col,C.Val}}),\end{aligned}$$

where the read/write denote the accesses to the respective CSR data structures in memory (e.g., $\boldsymbol{A}$.RowPtr, $\boldsymbol{A}$.Col, $\boldsymbol{A}$.Val. See Figure 3.3 for more details.) Note that the read accesses of the different matrix representations do not have uniform costs. For example, a *read* on $\boldsymbol{A}$.Col has a higher on-average cost than a *read* on $\boldsymbol{B}$.Col: since a complete row of matrix $\boldsymbol{B}$ is touched in-between two consecutive $\boldsymbol{A}$.Col-reads, the system has possibly evicted the cache line of $\boldsymbol{A}$.Col and has to fetch it again. The exact number of cycles, and hence, the required time per read or write access depends on whether the addressed cache line resides in the system cache. However, since we consider large matrices with sizes that are by factors larger than the last level system cache, we approximate each *read* and *write* access as a fixed time constant in our model. Moreover, instead of determining the individual time constants for the $\mathit{read}_X/\mathit{write}_X$ access, we abstract them into a few parameters that are then determined empirically.

Therefore, the expression of TM is expanded, and all time constants of the *read/write* accesses are accumulated into the constant parameters α, β, γ. Thus, we obtain a simple time approximation for the wall clock time:

$$T \approx \alpha(\underbrace{m \cdot k \cdot \rho_A}_{N^A_{\text{nz}}}) + \beta(\underbrace{m \cdot k \cdot \rho_A \cdot n \cdot \rho_B}_{\hat{N}_\times}) + \gamma(\underbrace{m \cdot n \cdot \hat{\rho}_C}_{\hat{N}^C_{\text{nz}}}),$$

which only depends on the constant parameters

$$\begin{aligned}\alpha &= T(\mathit{read}_{\text{A.Col,B.RowPtr}}),\\ \beta &= T(\mathit{read}_{\text{B.Col,A.Val,B.Val}} + \mathit{write}_{\mathit{sp_acc}}),\\ \gamma &= T(\mathit{write}_{\text{C.Col,C.Val}}),\end{aligned}$$

and the *derived* dimensions of the independent variable space:

- N^A_{nz}: the number of non-zeros in matrix $\boldsymbol{A}$,
- $\hat{N}_\times$: the estimated number of multiplications,

Table 4.1: The matrix multiplication kernels for the product $\boldsymbol{C}^{m\times n} = \boldsymbol{A}^{m\times k}\boldsymbol{B}^{k\times n}$ and their cost functions used in SpMachO. $N_\times$ denotes the number of actual multiplications, the hat indicates the corresponding estimated value. α, β, γ are constant parameters.

Kernel	Cost Function
DDD_GEMM	$\alpha(mkn)$
SPDD_GEMM	$\alpha(N^A_{\text{nz}}n)$
DSPD_GEMM	$\alpha(mk) + \beta\hat{N}_\times$
SPSPD_GEMM	$\alpha\hat{N}_\times + \beta(mn)$
DDSP_GEMM	$\alpha(mkn) + \beta\hat{N}^C_{\text{nz}} + \gamma(mn)$
SPDSP_GEMM	$\alpha N^A_{\text{nz}} + \beta(N^A_{\text{nz}}n) + \gamma\hat{N}^C_{\text{nz}}$
DSPSP_GEMM	$\alpha\hat{N}_\times + \beta\hat{N}^C_{\text{nz}} + \gamma(mk)$
SPSPSP_GEMM	$\alpha N^A_{\text{nz}} + \beta\hat{N}_\times + \gamma\hat{N}^C_{\text{nz}}$

- $\hat{N}^C_{\text{nz}}$: the estimated number of non-zero elements in the result matrix.

The cost function of the other multiplication kernels are deduced similarly. We omit their derivation here and list the final cost function for each kernel in Table 4.1. Each cost function is a linear combination of different derived dimensions, weighted by constant parameters α, β, γ. The constant parameters are estimated for each kernel by a multilinear least-squares fit. We used the fitting routines from the GNU scientific library (GSL, n.d.). Since all parameters are dependent on the system hardware, the fit has to be done once for each system configuration.

Figure 4.5 shows the scaling behavior of the matrix multiplication kernels SPSPSP_GEMM, SPSPD_GEMM, DSPD_GEMM, SPDD_GEMM, and DDD_GEMM. We measured the multiplication runtime of $\boldsymbol{AB}$ with respect to the matrix dimensions m, k, and densities ρ_A, ρ_B of the factor matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, respectively. While varying one of these variables, all other dimensions are fixed in the measurement. We observe that our cost models (lines) as of Table 4.1 conform very well with the actual algorithm runtimes (markers). The SPSPD_GEMM shows the best performance for the majority of the measurements. However, note that all kernels that write to a *dense* target matrix require a prior allocation and initialization of the target array, whereas the memory of sparse matrix representations is dynamically allocated during the multiplication. Hence, the allocation costs, which depend on m and n, have to be considered in addition to the XXD_GEMM kernel runtime. For illustrational purposes, we added the pre-allocation costs only for the SPSPD_GEMM kernel in Figure 4.5 (black dashed line.)

The plots in Figures 4.5a-4.5b clearly show the linear dependency of the kernel runtimes on the matrix dimensions m and k, except for the SPSPSP_GEMM kernel. The latter mostly depends on the estimated output size $\hat{N}^C_{\text{nz}}$, which is non-linear in k. From Figures 4.5c-4.5d we can infer the following scenario: if the density ρ_A (ρ_B) falls below a certain threshold (here $\approx 10^{-3}$ while the other is 0.03), then SPSPSP_GEMM is more efficient than SPSPD_GEMM (incl. allocation). In contrast, the latter outperforms all other approaches for intermediate densities, except for SPDD_GEMM and $\rho_B > 0.4$ in Figure 4.5d. A different situation is found in Figures 4.5e-4.5f: if ρ_A or ρ_B reach a higher value than about 0.3 (while the other is 0.15), it can be worthwhile to convert *both* $\boldsymbol{A}$ and $\boldsymbol{B}$ into dense representations, and continue with the dense DDD_GEMM kernel. Nevertheless, in order

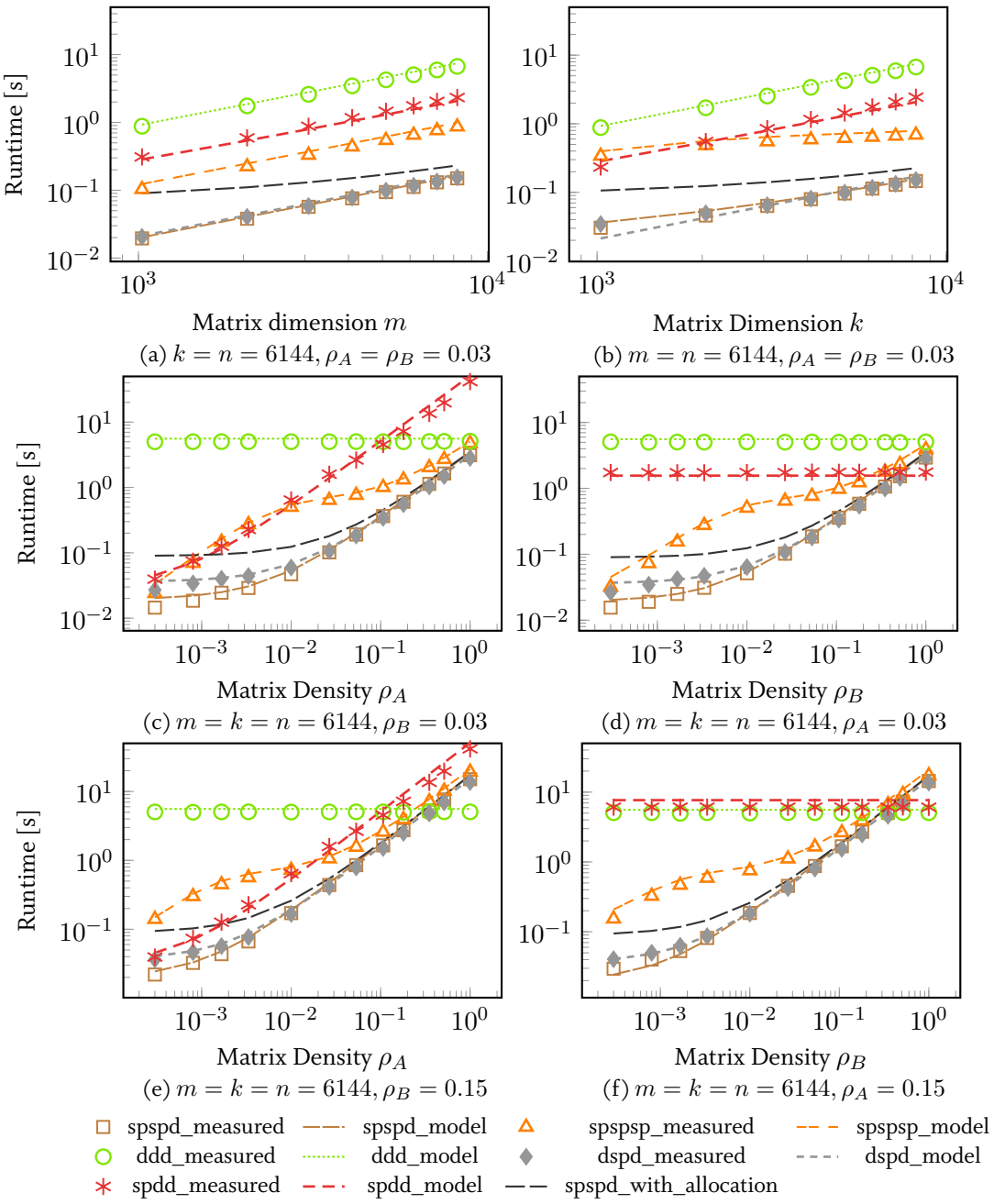


Figure 4.5: Measured runtimes (markers) and time estimates (lines) for the scaling behavior of different matrix multiplication kernels: each plot varies one of m, k, n, ρ_A, ρ_B while the other dimensions are fixed.

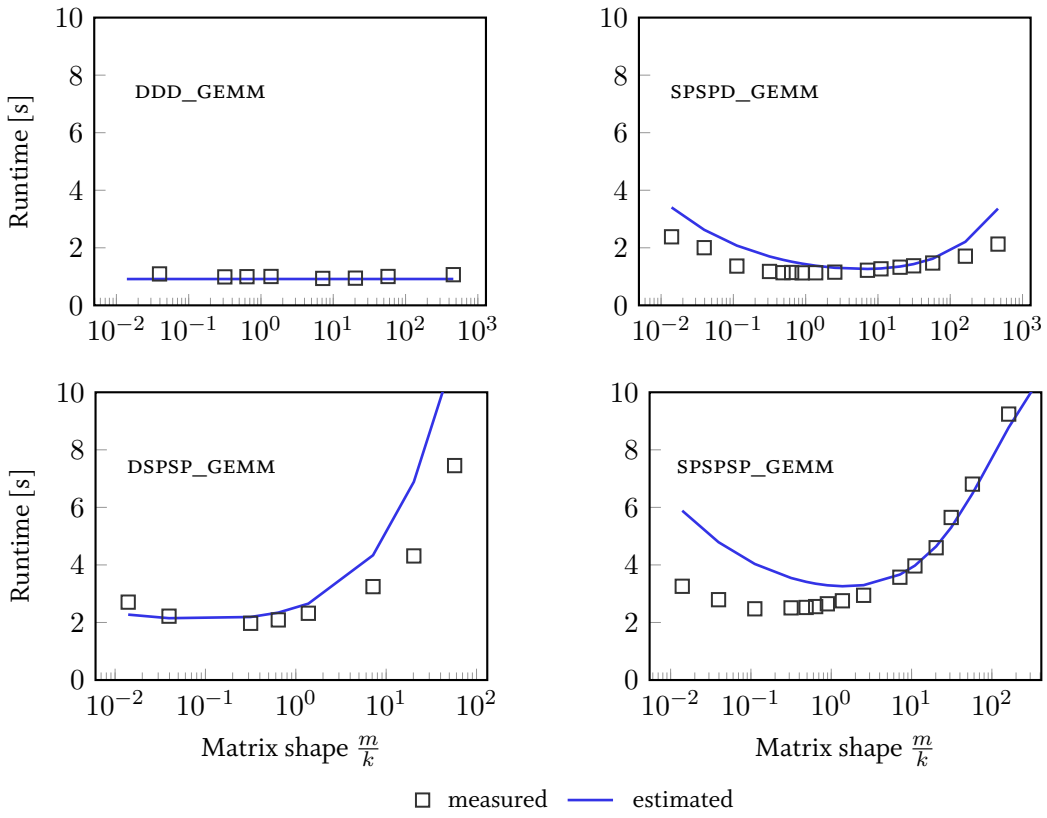


Figure 4.6: Measured runtimes (markers) and time estimates (lines) for $\boldsymbol{AB}$, where $\boldsymbol{A} \in \mathbb{R}^{m \times k}$ and $\boldsymbol{B} \in \mathbb{R}^{k \times n}$, by using different matrix multiplication kernels. The matrix shape m/k is varied while the number of element multiplications $N_\times$ is fixed in each plot.

to convert from sparse into dense representations, there must be enough available memory in the system. This is often not the case for matrices that have high dimensions ($m, k, n \gg 10^5$).

By the measurements shown in Figure 4.6 we cross-validate our cost model for non-well-shaped input parameters. In particular, the densities ρ_A, ρ_B and the product $(m \cdot k \cdot n)$ are fixed in this experiment, and only the relative matrix shapes $\frac{m}{k} \equiv \frac{k}{n}$ vary. Hence, at both edges of each plot the matrices are extremely rectangular, in contrast to the majority of the matrices that we used to train the model. The deviation of the actual times from our estimates shows that our cost model has limited accuracy in these extreme cases. However, the deviation is still acceptable, since in the corner cases ($\frac{m}{k} \ll 1$ or $\frac{m}{k} \gg 1$) the costs are always overestimated. Consequently, this leads to a more robust behavior: if a multiplication time is overestimated, then SpMachO likely selects another multiplication kernel that potentially has a higher accuracy. This is more favorable behavior than if costs were underestimated, because then the kernel with the lowest estimation would be picked, and the faulty deviation would be propagated into the overall estimation of the execution plan.

One conclusion we deduce from Figure 4.6 is that simplistic cost models, which solely depend on the number of non-zero multiplications ($N_\times$) are not able to describe the shape dependency, since $N_\times$ is fixed for each plot. Nonetheless, these simplistic models are commonly used in related work (e.g. in Boehm et al., 2014b).

4.4 DENSITY ESTIMATION

There are several reasons to have a prior estimate of the density structure of intermediate and result matrices, and they are barely different from the motivation of cardinality estimation for relational join processing: first of all, the optimizer's decision naturally depends on the matrix density, which is the most relevant parameter in the cost model of sparse multiplication kernels. Second, knowing the result density is crucial for memory resource management. Memory restrictions could also affect the choice of data representations in the optimizer, e.g., to avoid out-of-memory situations. In this section, we present a method to estimate densities of intermediate result matrices, which is a crucial part of the SpMachO optimizing process.

In fact, the *exact* non-zero structure can only be found through the actual execution of the multiplication. Some spspsp_gemm library implementations include a prior Boolean matrix multiplication in order to pre-allocate the target representation, e.g. Cholmod (Chen et al., 2008). In contrast, we require an estimate at negligible runtime cost compared to the actual execution, and the costs for a Boolean matrix chain multiplication are prohibitive. Therefore, we use a density estimation procedure based on the methodology of *probability propagation*. The general idea is to encode the non-zero structure into a small set of values ("density sketches") without losing important information. We now discuss two versions of density sketches, the *scalar density* value, and the multidimensional *density map*.

4.4.1 Scalar Density

A matrix can be considered as a two-dimensional object that has a certain population density ρ of non-zero elements. As an example, Figure 4.7a shows a 4×4 matrix that has five non-zero elements and thus, a total population density of $\rho = 5/16 \approx 0.31$. The scalar density value does not reflect any patterns in the matrix, but contains sufficient information if and only if the matrix is *uniformly* populated with non-zero elements. In this case the probability of a randomly picked matrix element for being non-zero is $p((A)_{ij} \neq 0) = \rho$, for instance $\rho = 0.31$ in Figure 4.7a.

Lemma 4.4.1. *Given that the non-zero elements are uniformly distributed, the density estimate $\hat{\rho}$ of a product of two matrices $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ can be calculated using probabilistic propagation as*

$$\hat{\rho}_C = \hat{\rho}_{A \cdot B} = 1 - (1 - \rho_A \rho_B)^k. \tag{4.10}$$

The estimate of Equation (4.10) is unbiased, i.e. $E[\hat{\rho}_C] \equiv \rho_C$.

Proof) Lemma 4.4.1 can be derived as follows: using the inner product formulation, the elements c_{ij} of the result matrix are calculated as $\sum_k a_{ik} b_{kj}$. The probability for a_{ik} being non-zero is $p(a_{ik} \neq 0) = \rho_A$, for b_{kj} accordingly: $p(b_{kj} \neq 0) = \rho_B$. Thus, every single summand $a_{ik} b_{kj}$ is nonzero with probability $p_{nz}(a_{ik}) \wedge p_{nz}(b_{kj}) = \rho_A \rho_B$. Furthermore, c_{ij} is non-zero if any of the summands $a_{ik} b_{kj}$ is non-zero. We leverage the inverse probability and obtain

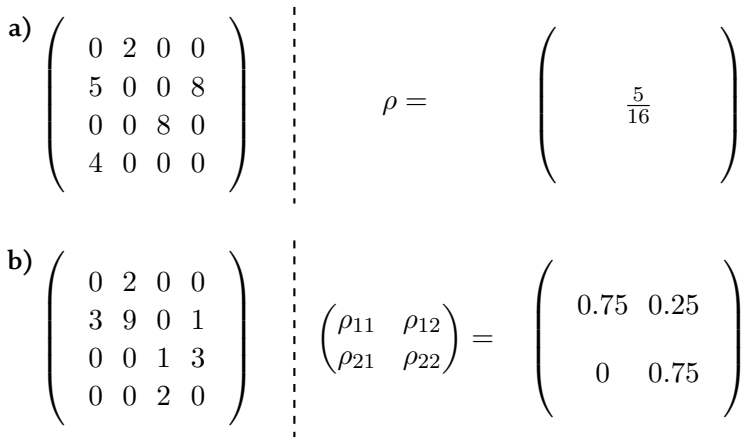


Figure 4.7: Assignment of non-zero population densities for two different matrices using a) a scalar density, or b) a density map.

$p(c_{ij} = 0) = \Pi_k(1 - \rho_A\rho_B)$. Finally, with $p(c_{ij} \neq 0) = 1 - p(c_{ij} = 0)$, and $p(c_{ij} \neq 0) = \rho_C$, equation (4.10) results.

We remark that we are assuming *no cancellation*, i.e., a sum of products of overlapping non-zero elements never cancel to zero, which is a very common assumption in mathematical programming literature (Cohen, 1998). For a better readability, we will denote the operation in Equation (4.10) with the single symbol $\odot$, thus, $\hat{\rho}_C = \rho_A \odot \rho_B$.

Equation (4.10) can be used as an $\mathcal{O}(1)$ estimator for the result density prediction of a multiplication of two matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, which have uniform non-zero patterns. Hence, the density prediction for a chain multiplication of p matrices using the scalar density has linear time complexity $\mathcal{O}(p)$. The density prediction is independent of the parenthesization.

4.4.2 Estimation Errors

The obvious disadvantage of maintaining a scalar density is that the density estimate as of Equation (4.10) is only valid for matrices with uniformly distributed non-zero elements. Although the uniform approximation holds for many matrices to a certain degree, there are matrices that have distinct non-zero patterns, i.e., a topology with some regions significantly more dense than others. For these matrices, a density prediction according to Equation (4.10) does not provide an accurate, unbiased result.

In extreme non-uniform cases, the scalar $\hat{\rho}$ estimate according to Equation (4.10) could produce an asymptotic maximum error of 100%. Figure 4.8 shows two example cases where it fails significantly: In the first case, the product of two $n \times n$ matrices with each $\rho = 0.5$ cancel out into an empty matrix with $\rho = 0$, whereas the naive scalar density estimate is $\hat{\rho} = 1 - (0.75)^n \xrightarrow{n\to\infty} 1$. Hence, the density value is maximally overestimated. The second example is a multiplication of two sparse $\rho = \frac{1}{n}$ matrices, which are zero except one full column in $\boldsymbol{A}$ and the matching row in $\boldsymbol{B}$. The resulting full matrix has $\rho = 1$, whereas the naive prediction gives $\hat{\rho} = 1 - (1 - \frac{1}{n^2})^n \xrightarrow{n\to\infty} 0$.

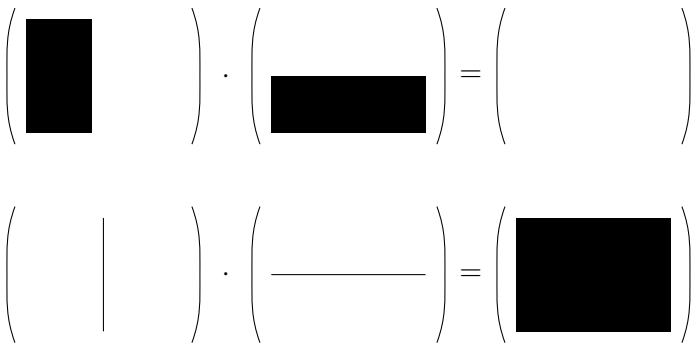


Figure 4.8: Product density in extreme non-uniform cases. The upper row shows how two half-populated matrices cancel to zero. The lower row shows how two, almost empty sparse matrices produce a full matrix (outer vector product).

In order to lower the average estimation error, we estimate matrix densities on a finer granularity. In particular, we use a *density map* for non-uniform sparse matrices, which reflects a 2D-matrix pattern on a configurable, granular level.

4.4.3 Density Map

The density map $\boldsymbol{\rho}_A$ of a $m \times n$ sparse matrix $\boldsymbol{A}$ is effectively a $\frac{m}{b} \times \frac{n}{b}$ density histogram. It consists of $\left(\frac{mn}{b^2}\right)$ density values $(\rho)_{ij}$, each referring to the density of the corresponding block $\boldsymbol{A}_{ij}$ of size $b \times b$ in matrix $\boldsymbol{A}$. As an example, Figure 4.7b shows the density map of a 4×4 matrix with blocks of size 2×2.

In the following we sketch how the density map $\hat{\boldsymbol{\rho}}_C$ of the result matrix $\boldsymbol{C}$ is estimated from the density maps $\boldsymbol{\rho}_A$ and $\boldsymbol{\rho}_B$ of its factor matrices $\boldsymbol{A}$ and $\boldsymbol{B}$. Therefore, it is worthwhile to take a glance at blocked matrix multiplication. Assuming square blocks, the product can be represented as

$$\boldsymbol{C} = \begin{pmatrix} \boldsymbol{A}_{11} \cdot \boldsymbol{B}_{11} + \boldsymbol{A}_{12} \cdot \boldsymbol{B}_{21} & \boldsymbol{A}_{11} \cdot \boldsymbol{B}_{12} + \boldsymbol{A}_{12} \cdot \boldsymbol{B}_{22} \\ \boldsymbol{A}_{21} \cdot \boldsymbol{B}_{11} + \boldsymbol{A}_{22} \cdot \boldsymbol{B}_{21} & \boldsymbol{A}_{21} \cdot \boldsymbol{B}_{12} + \boldsymbol{A}_{12} \cdot \boldsymbol{B}_{22} \end{pmatrix}. \tag{4.11}$$

For the derivation of the density map estimation, we first define an estimator for the *addition* of two matrices:

Lemma 4.4.2. *Given that the non-zero elements are uniformly distributed, the density estimate $\hat{\rho}$ of the addition of two matrices $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ can be calculated using probabilistic propagation as*

$$\hat{\rho}_{A+B} = \rho_A + \rho_B - (\rho_A \rho_B) \equiv \rho_A \oplus \rho_B. \tag{4.12}$$

Equation (4.12) simply reflects the addition law of probability, also known as the sum rule. The derivation is straightforward and can be obtained by using the Kolmogorov axioms (Shafer and Vovk, 2006). We refer interested readers to the related literature for more details.

By combining Equation (4.11) with (4.10) and (4.12), one obtains

$$\hat{\boldsymbol{\rho}}_C = \begin{pmatrix} \rho_{A_{11}} \odot \rho_{B_{11}} \oplus \rho_{A_{12}} \odot \rho_{B_{21}} & \rho_{A_{11}} \odot \rho_{B_{12}} \oplus \rho_{A_{12}} \odot \rho_{B_{22}} \\ \rho_{A_{21}} \odot \rho_{B_{11}} \oplus \rho_{A_{22}} \odot \rho_{B_{21}} & \rho_{A_{21}} \odot \rho_{B_{12}} \oplus \rho_{A_{22}} \odot \rho_{B_{22}} \end{pmatrix}$$

for the density propagation of a 2×2 map. Density maps of a finer granularity, i.e., with more than four blocks, are calculated accordingly.

To summarize, the average density estimation error is significantly lowered by using density maps instead of the scalar density estimation, which we verify empirically in the evaluation in Section 4.5. As a matter of fact, the smaller the block size and the higher the granularity, the more information is stored in the density map and finer structures can be resolved. However, the runtime of the density map estimation also grows with the granularity, since its complexity is in $\mathcal{O}(\left(\frac{n}{b}\right)^3)$, and hence, $\mathcal{O}(p\left(\frac{n}{b}\right)^3)$ for a chain estimation of length p. For infinitesimal block sizes $b \to 1 \times 1$ the estimation error vanishes completely, but then the determination of $\hat{\boldsymbol{\rho}}_C$ becomes equivalent with the Boolean matrix multiplication of $\boldsymbol{A} \cdot \boldsymbol{B}$, which has the same problem complexity as the actual multiplication. Thus, the block size configuration is generally a trade-off between accuracy and runtime of the prediction.

Nevertheless, to reduce the prediction time we employ a greedy strategy by using density maps only for matrices with a skewed non-zero distribution, and scalar density for matrices with an approximately uniform non-zero distribution. In order to decide whether a given matrix has a uniformly distributed or a skewed non-zero pattern, we define a quantitative disorder measure for sparse matrices.

4.4.4 Matrix Disorder Measures

In order to determine the skewness of the matrix density, we introduce two measures: the first is based on an *F-test statistic*, the other on the *entropy*. In particular, they quantify to what extent the non-zero pattern of a sparse matrix deviates from a uniform distribution. We approach the disorder problem by using the same block-granularity as for the density map.

Variance Analysis

A possible way to decide whether a matrix is approximately uniformly distributed is by using a statistical hypothesis test. The scalar density propagation formulas shown in Lemmas 4.4.1 and 4.4.2 are based on the assumption that every matrix element has the same probability to be populated, and the probability is equal to the overall density ρ. This assumption forms the null hypothesis H_0. Consequently, the number of non-zero elements in each $b \times b$-block would follow a binomial distribution $\mathcal{B}(N, p)$ with $N = b^2$ and $p = \rho$. This can be deduced in the same way as a coin toss experiment, where the number of experiments N is equal to the total number of matrix elements in a block, and the *success* probability $p = \rho$ equals the global population density. From elementary statistics (e.g. Box et al., 2005) we obtain

$$E[N_{\text{nz}}^{b\times b}] = b^2\rho, \qquad E[V(N_{\text{nz}}^{b\times b})] = b^2\rho(1-\rho) \tag{4.13}$$

as expectation values for the number of non-zero elements $N_{\text{nz}}^{b\times b}$ in one block, and the variance of $V(N_{\text{nz}}^{b\times b})$, based on a binomial distribution $\mathcal{B}(b^2, \rho)$.

The conformance of the null hypothesis H_0 with the reality can be determined using a simple one-factorial variance analysis. Therefore, we use the F-test (Box et al., 2005), which is a likelihood

quotient test that checks the accordance of the observed variance of two normally distributed[2] random variables X and Y. If the test statistic $f = V_X/V_Y$ (ratio of variances in X and Y) exceeds a critical value f_{crit} according to an α-quantile of the F-distribution, H_0 is rejected, and X and Y are assumed to be of different distributions by the probability of 1-α.

For a matrix with N_B blocks, we define

$$V_{\text{observed}} = \frac{1}{N_B - 1} \sum_{ij} (N_{\text{nz}}(ij) - E[N_{nz}^{b\times b}])^2 \tag{4.14}$$

$$f = \left(\frac{V_{\text{observed}}}{V_{\text{expected}}}\right) = \left(\frac{V_{\text{observed}}}{E[V(N_{nz}^{b\times b})]}\right) \tag{4.15}$$

For uniformly distributed matrices, f has the expectation value $E[f] = 1$. The exact choice of the threshold, however, depends on the sample size, i.e. the number of blocks N_B and the desired accuracy.

Entropy

A known measure for the disorder of elements is the entropy

$$\sum_{i}^{N} -p_i \ln p_i, \tag{4.16}$$

which is defined over a space with N entities (or states) i that have a relative probability p_i. The entropy is used in a variety of contexts. To name an example, the entropy definition by Shannon (2001) is used in information theory to quantify the information content of a message with N characters as entities. Similarly, we can define and quantify the information content of the non-zero pattern of a sparse matrix, by identifying the p_i with the local block density ρ_{ij}.

The entropy (4.16) is maximal if each entity has the same probability. In the terms of sparse matrices, the theoretical disorder is maximized if every block has the same local density $\rho_{ij} \equiv \rho$. The scaled entropy is defined as

$$\tilde{H} = \frac{H}{H_{\text{max}}} = \frac{\sum_{ij}^{N_B} \rho_{ij} \ln \rho_{ij}}{N_B \rho \ln \rho} \in [0, 1]. \tag{4.17}$$

In fact, both $\tilde{H}$ and f are sensitive to the matrix density skew, which is backed by our evaluation in Section 4.5. However, in contrast to f, the entropy is merely suited for measuring *relative* changes in the non-zero disorder, since it is hard to interpret the absolute value of H.

4.4.5 Sparse Product Density Estimator

The pseudocode of our sparse product density estimator (SpProdest) is sketched in algorithm 13. As a first step, the disorder measure $\delta = \sqrt{1/f}$ is retrieved (getDisorder) for each matrix, which is a modified version of f according to Equation (4.15). Based on δ it is decided whether to

[2] for sufficiently large N and $Np \to$ const., the binomial distribution can be approximated by a normal distribution

Algorithm 13 The sparse product density estimator (SpProdest)

```
1:  function spProdest(MatrixChain A_[1..p])
2:      ρ̂[][] ← 0
3:      for Matrix A_i ∈ A_[1..p] do
4:          δ ← getDisorder(A_i)
5:          if δ < δ_T then
6:              ρ̂[i][i] ← scalarDensity(A_i)
7:          else
8:              ρ̂[i][i] ← densityMap(A_i)
9:      for 1 < j < p do
10:         for j > i > 0 do
11:             ρ̂[i][j] ← estProdDensity(ρ̂[i][j − 1], ρ̂[j][j])
        return ρ̂[][]
```

create a scalar density value only, or the full density map (line 4). If it is lower than a certain threshold δ_T, then the density map is created. In contrast, if the disorder is higher (i.e. sufficiently high), then a scalar density is chosen. Finally, the density estimates are computed for every matrix subchain by using the probabilistic density propagation method (EstProdDensity) according to Equations (4.10) and (4.12). Note that instead of calculating δ and $\hat{\boldsymbol{\rho}}$ for each expression (line 4-8), they can be cached as matrix statistics in the system, and reused for further multiplications.

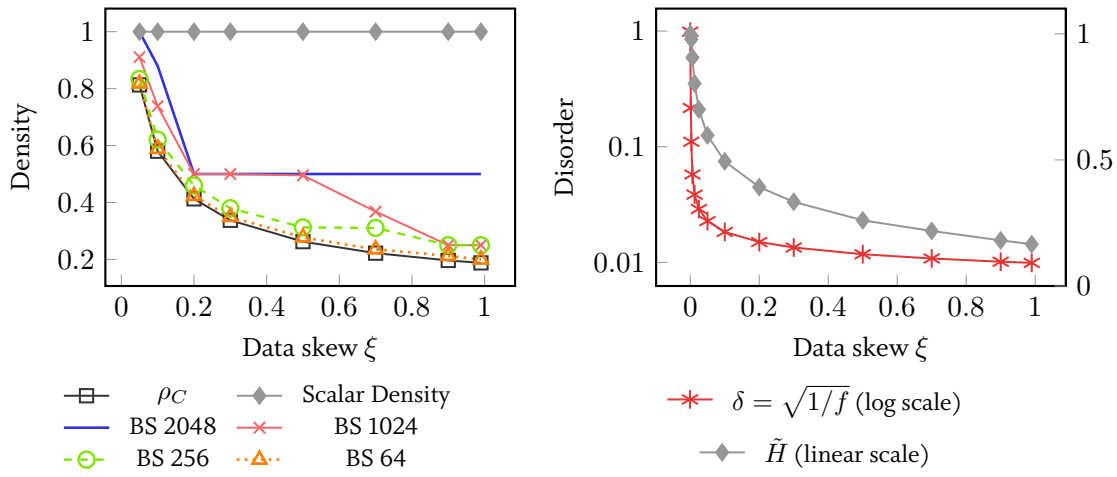
The complexity of SpProdest depends on the chain length and the actual granularity of the density map. Assuming a chain multiplication of p square matrices, the time complexity would be in the best case $\mathcal{O}(p^2)$, when only scalar densities are used. The term is quadratic in p due to the nested loop in line 10. In the worst case, it is $\mathcal{O}(p^2(\frac{n}{b})^3)$, where $\frac{n}{b}$ is the dimension of the density map. Analogously, the space complexity of SpProdest is in the best case $\mathcal{O}(p)$ and in the worst case $\mathcal{O}(p(\frac{n}{b})^2)$. In practice, the overhead of the SpProdest component is negligible against the potential speedup gained by the SpMachO, which is manifested in our evaluation.

4.5 EVALUATION

In this section, we first evaluate the accuracy of spProdest. Secondly, we apply SpMachO on different matrix chains and compare the execution runtime against R and a popular commercial numerical algebra system for matrix computations (called system A here), and two further execution approaches. The platform for our prototype implementation is a two-socket Intel Xeon X5650 CPU with 2×6 cores with 2.66 GHz and a total of 48 GB RAM.

4.5.1 Accuracy of the Density Estimate

As mentioned before, the density ρ_C of a matrix $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ not only depends on the densities of the factor matrices ρ_A and ρ_B, but also on their non-zero patterns, especially on the pattern skew. To show the effect of the non-zero pattern skew on the density estimation, we generated a set of matrices with increasing skew. The skew parameter ξ in our example defines the slope



(a) Estimated density vs. actual density (ρ_C) of matrix $\boldsymbol{C} = \boldsymbol{AB}$ using a scalar density and density maps with different grid block sizes (BS).

(b) Influence of the matrix $\boldsymbol{A}, \boldsymbol{B}$ nonzero skew on the disorder measures $\delta, \tilde{H}$.

Figure 4.9: Accuracy of the density estimate and skew sensitivity of disorder measures. Matrices: $\boldsymbol{A} \in \mathcal{R}^{4096\times2048}$, $\boldsymbol{B} \in \mathcal{R}^{2048\times4096}$ and average density $\langle\rho\rangle = 0.1$.

of a linear ascend in the density distribution with increasing row number r: $\rho(r) = \xi \cdot r$. We fixed the total density of $\boldsymbol{A}$, (thus, N_{nz} is constant,) and set $\boldsymbol{A} = \boldsymbol{B}$. Figure 4.9a shows the actual density ρ_C and the densities estimate of the scalar density approach and density maps with different block sizes. In our example, the density of the product matrix ρ_C decreases with increasing pattern skew. This conforms with our notion, that in most cases a higher skew at constant density leads to a lower density of the result matrix. Nevertheless, there are cases which show the opposite behavior, for example as shown in Figure 4.8. In fact, the density only decreases below $\min \rho_A, \rho_B$ if both matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ contain empty columns and/or rows.

From Figure 4.9a we observe that the simple scalar density estimate only provides a sufficient accuracy for approximately uniform nonzero patterns ($\xi \to 0$.) Furthermore, it is clearly observable that a finer granularity of the density map, and thus, a smaller block size, results in a better density estimation. Since a finer granular density map negatively influences the runtime performance of spProdest, our strategy is to choose the block size as small as necessary, but as large as possible. Here, we chose a block size of 256×256 as a good compromise between accuracy and estimation runtime. In general, however, the block size should be chosen adaptively depending on the matrix dimensions, its density, and the desired estimation accuracy.

Figure 4.9b confirms that the disorder measures are both sensitive to the nonzero skew. Nevertheless, the test statistic-based disorder measure δ is much more sensitive to the skew than the entropy-based $\tilde{H}$.

4.5.2 Plan Ranking

In this experiment we evaluate the overall accuracy of the cost estimation for a complete execution plan by comparing estimated with actual runtimes. As a matter of fact, SpMachO is optimal if the plan with the lowest estimated runtime has also the lowest actual runtime. However, the exact experimental verification of optimality requires running all possible execution plans. In practice, this is not feasible for longer matrix chains due to the exponentially growing number of plans according to Equation (4.9). Thus, we ran a "brute-force evaluation" only for matrix chains of length $p = 3$:

$$\boldsymbol{A}_1^S \cdot \boldsymbol{A}_2^S \cdot \boldsymbol{A}_3^S. \tag{4.18}$$

Although there are only two ways of setting the parenthesis in this expression, with all the possible storage type transformations per multiplication node, one obtains 128 different execution plans. Figure 4.4 shows some of the possible plans, to illustrate the problem complexity. The execution plan of a matrix chain multiplication is composed of

- **multiplication operators** ${}^{D/S}\times^{D/S}_{D/S}$, which can either produce a *sparse* result matrix or a *dense* one. For a chain of length p there are exactly $p - 1$ multiplication operators.
- **transformation operator** $T^{S/D}_{S/D}$ that transforms the intermediate result from one storage representation into another (which is either dense or sparse.) There can be none or up to 2^{3p-1} transformation operators.

SpMachO estimates the cost for each operator via the cost functions `TM` and `TT` introduced in Section 4.2. As a consequence, each operator estimation has an uncertainty that contributes to the absolute error of the execution runtime.

In this experiment, we first executed all 128 plans and measured the *actual* execution runtime. Thereafter, we computed the runtime estimations using SpMachO's cost model for each plan. Figure 4.10 shows a two-sided vertical histogram for the actually measured (left-hand side) and the estimated (right-hand side) execution runtimes. Note that the vertical axis has a logarithmic scale, hence, the width of the upper bins refers to a larger time interval than the width of the lower bins. The connecting edges in-between the bins of the two sides of the histograms show where the plans of an actual runtime bin are placed in the estimated runtime bin. If there is an ascending edge, for example from the lowest bin in the left side to the second lowest bin of the right side, then there is at least one plan whose runtime was overestimated. If instead the edge is horizontal, then the estimated runtimes of all plans corresponding to this edge are within the same time interval as their actual runtimes. The line width of the edges indicates how many plans are affected. It is worth mentioning that for the selection of the best execution plan, the absolute value of the plan estimate can deviate arbitrarily from the actual time, as long as the actual runtime plan *order* is correctly preserved in the estimated plan ranking. This condition is only violated for the edges that are crossing another edge. If the total runtime for a plan is significantly under- or overestimated, its corresponding edge crosses multiple other edges. Indeed, the goodness of the cost model can be defined by the number of edge crossings, weighted by the line width of the crossing edges.

The majority of estimations that are shown in Figure 4.10 are in the corresponding/same time bin in both sides of the histogram. Although there are in fact some crossings, especially in the

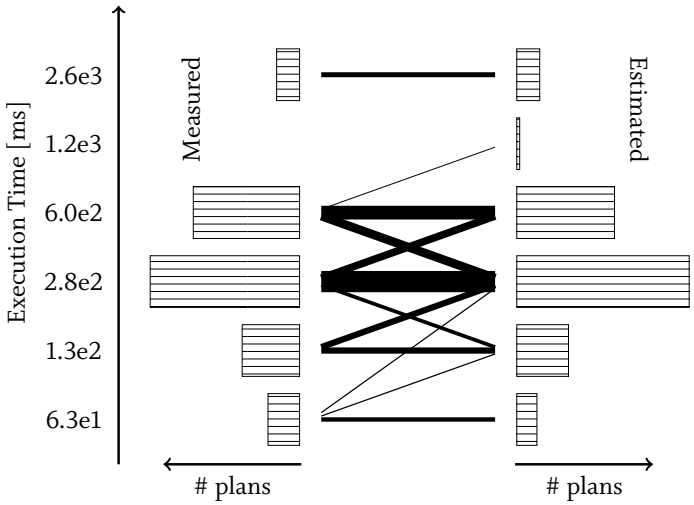


Figure 4.10: Vertical histogram of the actual (bars on left-hand side) and estimated (bars on right-hand side) runtimes of all 128 possible execution plans. The edges denote the plan (dis-)placement, the line width correlates with the corresponding number of plans. The vertical time axis has a logarithmic scale.

middle part, most of the edges only span over to the neighboring bin. Moreover, for the selection of the most efficient plan, only the lower part of Figure 4.10 is relevant. In particular, the plan with the lowest estimated runtime should be contained in the lowest bin of the actual runtime histogram. Since all edges of the lowest estimated time bin originate from the lowest actual time bin, we observe that the SpMachO selected plan is at least among the top k plans.

4.5.3 Performance Comparison

We compared the absolute execution runtime of a matrix chain multiplication expression using the optimized plan by SpMachO against R and commercial system A. Since this experiment aims at evaluating the execution of complete matrix *expressions* rather than the single multiplication performance, we explicitly chose these frameworks that provide a expressive language interface. Both systems contain classes and algorithms for dense and sparse matrices. In R (V3.0.0) we used the Cran R matrix package V1.0.12 (Bates, n.d.), in commercial system A we used the native sparse matrix representation. We are aware that there are also HPC libraries offering single sparse matrix multiplication, however, these are rather orthogonal to our work since SpMachO is concerned with the optimization of composite expressions on a higher level, and may use interchangeable sparse multiplication algorithms as kernels.

In addition, we included alternative fixed execution approaches to the measurement:

1. **Left-deep, sparse only:** All matrices are multiplied using the sparse-sparse into sparse spspsp_gemm kernel ($^S\times^S_S$), starting with the first left pair and proceeding into the right direction.

2. **Right-deep, sparse-dense into dense:** The outermost right pair is multiplied using sparse-sparse into dense multiplication ($^{S}\times^{S}_{D}$). Then, the matrices on the left are consecutively multiplied with the right-hand dense intermediate result matrix using the SPDD_GEMM multiplication kernel ($^{S}\times^{D}_{D}$).

Both alternative approaches are implemented in the LAPEG and thus, use the same infrastructure as SPMACHO. The first, sparse-only execution strategy is straightforward and simple. It reflects a naive way, since it does not take the evolving density of intermediates into account. We chose left-deep over right-deep, since the standard way of compilers is to transform a composite expression left-to-right into an abstract syntax tree (AST). This is the way how most linear algebra language interpreters like R or MATLAB handle sparse matrix chain expressions internally. The second strategy is intuitive for two reasons: firstly, we observe that the density of matrix products is often significantly higher than that of its factor matrices, hence, it might make sense to store all intermediate results as dense. Secondly, we chose right-deep over left-deep, since the SPDD_GEMM kernel tends to have a better performance than DSPD_GEMM according to our cost model in Section 4.3. Either of these two execution strategies turned out to be good (or even optimal) for some of the test instances.

Further strategies. We also tried other alternatives to the dynamic programming approach of SPMACHO, for example, a method that picks an execution plan based on simulated annealing (Kirkpatrick et al., 1983), a meta-heuristic to approximate global optimization in a large search space. However, due to the high dimensionality of the search space and the large discrepancy in the runtimes, it turned out to be far worse in most cases, hence, we did not include it in the measurements.

Experimental Setup

Since there are currently no standardized benchmarks for large scale linear algebra expressions, it is generally difficult to provide a comprehensive performance comparison. Therefore, we created two performance experiments: first, we took real world-matrices of different domains and compared the execution runtime of self-multiplication chains (matrix powers). Table A.3 in appendix A lists the matrix data sets which we used in the evaluation. Thereafter, we generated random matrices of different dimension and density skews in order to study the systematic behavior of SPMACHO.

Our LAPEG prototype uses the sparse/dense matrix multiplication kernels that were described in Chapter 3. We emphasize that the conceptual execution plan optimization of SPMACHO works orthogonal to the individual performance of the multiplication kernels. Nonetheless, the absolute execution runtime does obviously also depend on the low level implementation of each algorithm. Hence, there is still potential to reduce the overall runtime further by further tweaking the multiplication kernels.

Self Multiplications

In this part we discuss the performance of matrix self multiplications (matrix powers), which is for example used for the calculation of Markov chains models.

The left column of Figure 4.11 shows the absolute runtimes using SPMACHO versus R, commercial system A, and the right-deep SPSPSP- and left-deep SPDD_GEMM-only approaches. In addition

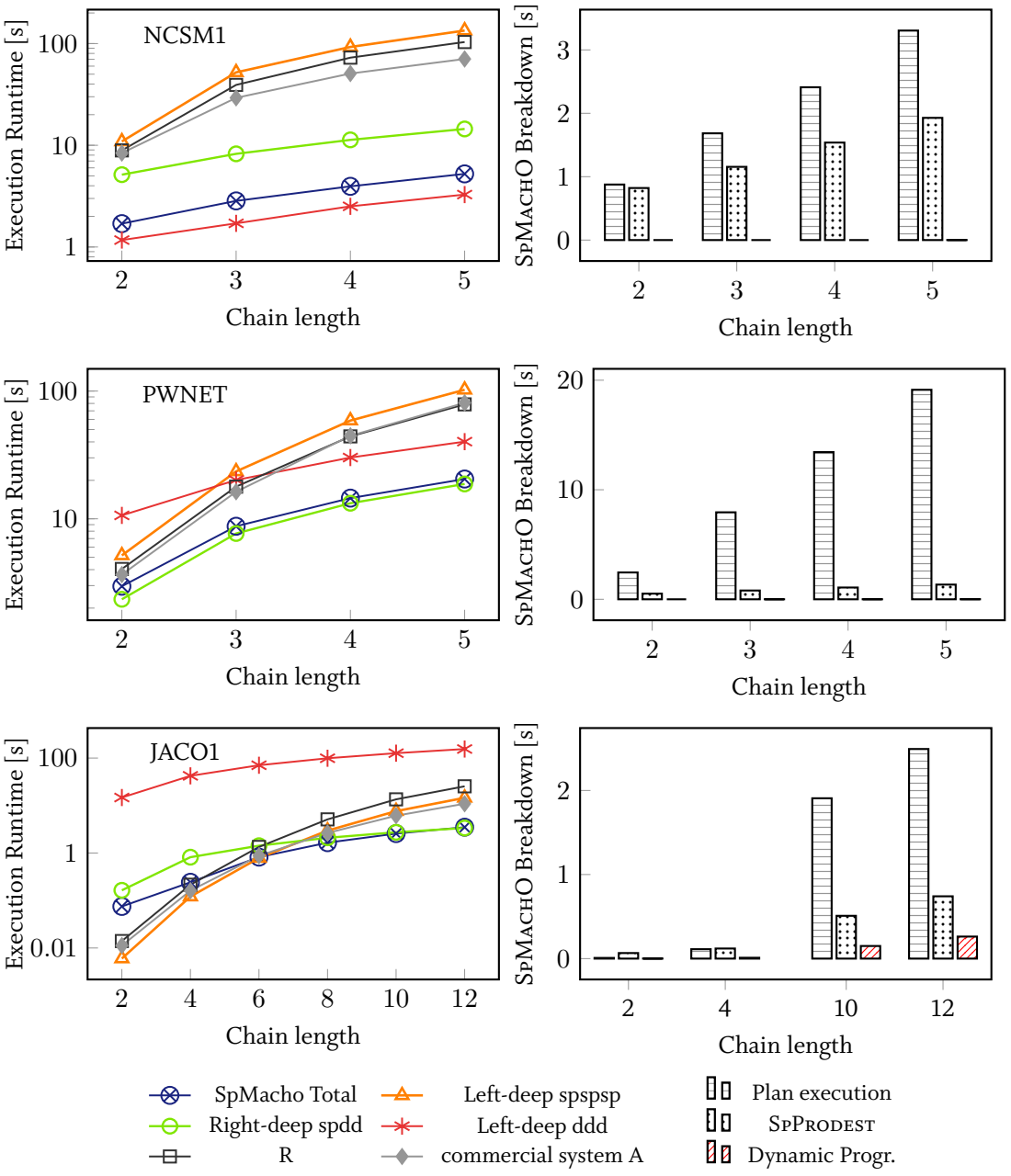


Figure 4.11: *Left Column*: Measurement of the execution runtime of sparse matrix chains (matrix powers). *Right Column*: The total runtime of SpMachO is visually separated into its components: the plan execution, the SpProdest runtime and the dynamic programming part.

to the original measurements of our publication (Kernert et al., 2015a), we add a left-deep ddd-approach, which uses dense multiplications only. The first notion is that the relative performance speedup of SpMachO becomes more significant with increasing chain length. For matrices with

a relatively high density, e.g. NCSM1, SpMachO recognizes that it is worth to convert the matrix into dense representations prior to the multiplication, and uses the dense ddd_gemm kernel (Intel MKL) internally for each multiplication. The remaining deviation of SpMachO from the dense-only execution is caused by the optimization and conversion overhead. For the "medium-sparse" matrix PWNET chain, SpMachO selects immediatly a strategy that is similar to the right-deep spdd. The optimization overhead is amortized for a chain length greater than two. In the measurement of the hypersparse matrix JACO1, the overhead of SpMachO amortizes not below a chain length of four. Note that in the case of sparse matrix self-multiplications (except for the relatively dense NCSM1), the performance is mainly related to the density evolution of the intermediate results. As soon as the intermediate matrix reaches a relatively high density in an early stage of the execution plan, SpMachO likely chooses a dense intermediate format and proceeds with dense multiplication kernels. Similarly, right-deep spdd_gemm-strategy exhibits a good performance when the intermediate results are dense, which is immediatly the case for PWNET, and for longer chains (≥ 8) of JACO1. Conversely, the left-deep spspsp_gemm-only strategy tends to have a worse performance for longer chains, since at a certain length the intermediate is rather dense such that a mixed algorithm would have been the better choice. Nevertheless, it outperforms the other strategy for short, sparse chains (e.g., for the matrix JACO1).

The right column of Figure 4.11 shows the separate runtimes of each component of SpMachO, which are: the plan execution, the SpProdest runtime, and the dynamic programming loop. The runtime of the dynamic programming part is negligible (<1ms) and only visible for longer chains (10-12). Despite the actual plan execution, the major part of SpMachO is taken by the density estimator SpProdest, whose complexity is quadratic in the chain length p (whereas the dynamic programming part is cubic in p). Note that the SpProdest cost can be reduced by caching the density maps. As of now, they are created once prior to each expression execution, adding a fixed overhead to SpProdest's runtime. Moreover, the density map estimation algorithm also depends on the block size, which should be adapted to the matrix characteristics, as stated before. Hence, it could be further accelerated by choosing a coarser density map, while, however, sacrificing some estimation accuracy.

The R and commercial system A runtimes show astonishing similarity with our left-deep spspsp_gemm approach. It stands to reason that they use a similar way of execution, e.g., by compiling the multiplication chain naively from left to right into a left-deep, sparse-only multiplication operator tree. We remark that in Figure 4.11 showing the original results of (Kernert et al., 2015a) a *sequential* version of the sparse-only kernel (spspsp_gemm) was used. To our surprise, this is in line with the implementation of common sparse matrix libraries including R (Bates, n.d.), Matlab (Gilbert et al., 1992) and cholmod (Chen et al., 2008), although R and Matlab use parallelized (Blas) algorithms for dense matrix multiplications. This also explains why the left-deep ddd-strategy (using the parallelized ddd_gemm) outperforms the left-deep spspsp-approach using matrix PWNET already for chain lengths greater than three. We repeated the measurement in Figure 4.12 using our recent, parallelized spspsp_gemm kernel, and an improved spdd_gemm algorithm. As expected, the performance of the updated left-deep spspsp approaches that of the right-deep spdd-strategy (which already uses a parallelized spdd_gemm kernel). Nevertheless, we emphasize that tuning the internal multiplication kernels do not change the qualitative results of the previous measurements. On the contrary, SpMachO's performance naturally benefits from the im-

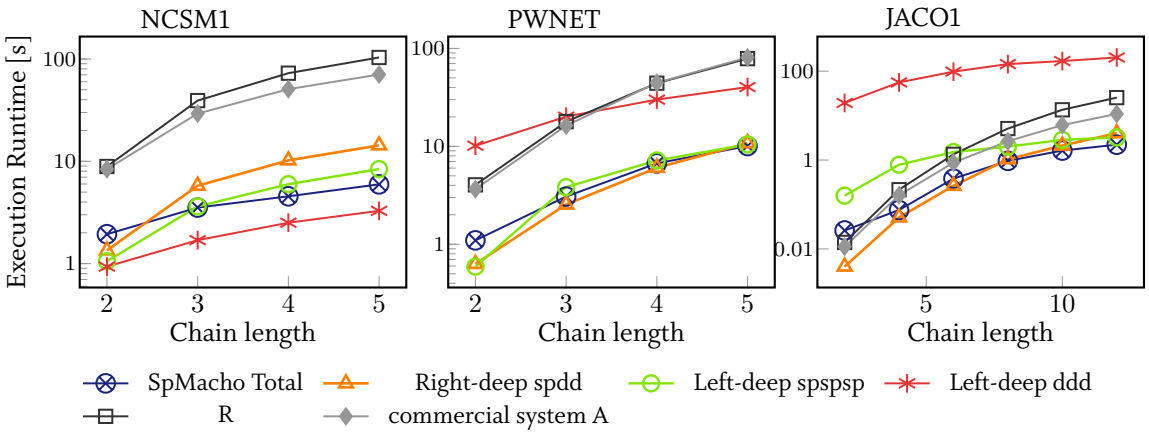


Figure 4.12: Multiplication chain experiment of Figure 4.11 with a parallel version of SPSPSP_GEMM and an updated SPDD_GEMM multiplication kernel.

proved kernels, and picks a near-optimal strategy based on their cost model. In all measurements of the Figures 4.11 and 4.12 one of the included alternative strategies turned out to be (near-)optimal. However, so far we only considered the rather special case of matrix self-multiplications. In the general case, i.e., multiplication chains of *different* matrices, the naive approaches might deviate significantly from the optimal strategy, particularly if there is a high data skew.

Random Matrix Chains

In the next experiment, we measured the performance for the multiplication of three randomly created matrices, which are pair-wise different. Products of three matrices are very common in many applications, in particular in iterative matrix factorization algorithms. Nevertheless, we remark that we observed even more significant speedups of SPMACHO for longer multiplication chains, since these exhibit a higher optimization potential.

Also note that like in the previous experiment, our original measurement (Kernert et al., 2015a) shows the performance using the sequential SPSPSP_GEMM. Again, we included the strategies using more recent kernel algorithms (the parallelized SPSPSP_GEMM and an updated version of SPDD_GEMM). Although we used the same skew distributions, it is not possible to recreate the exactly same test conditions due to the random generation of matrix instances. Hence, we show the updated measurements in separate plots (the updated versions are shown on the left-hand side in Figure 4.13).

In order to observe the systematic influence of the data skew on the execution runtime, we varied three skew dimensions: the matrix shape skew, the inter-matrix density skew and the matrix intra-density skew. The different skew types are illustrated on the right-hand side of Figure 4.13. For each skew dimension, we varied a parameter $\xi \in [0, 1]$ that quantifies the skew in a range from zero (no skew) to one (maximum skew). More precisely, the parameter dimensions $(m/n)_i$, ρ_i and the intra-density skews ξ_i, are randomly picked from a <*min*, *max*, *average*> distribution, where ξ corresponds to the deviation from the *average* value.

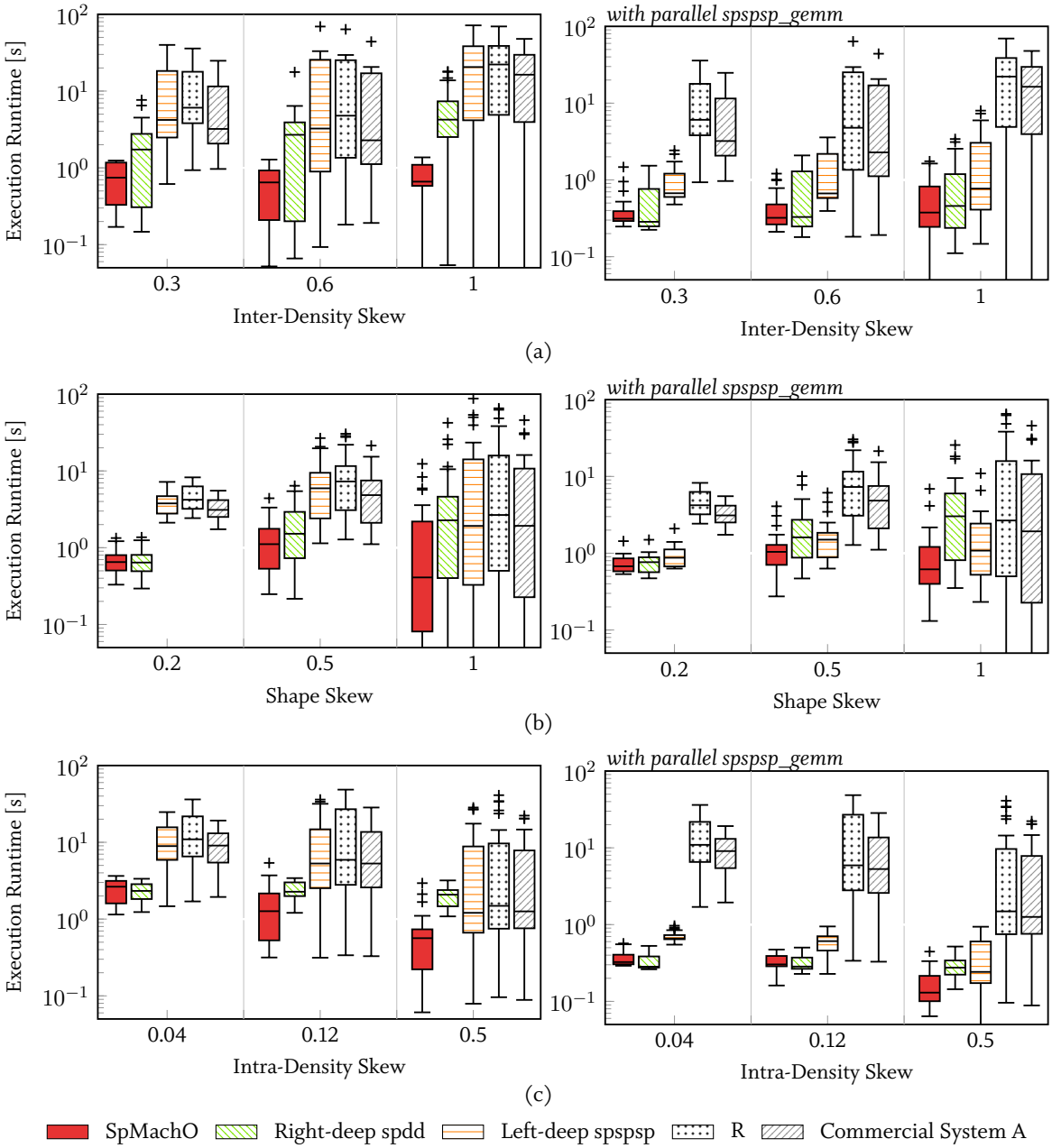
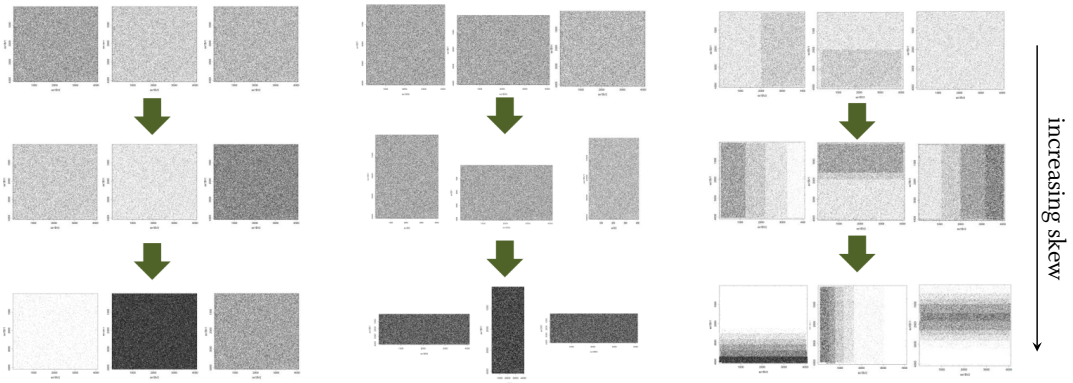


Figure 4.13: Average runtime and variance for expression $\boldsymbol{A}_1 \cdot \boldsymbol{A}_2 \cdot \boldsymbol{A}_3$. Compared are SpMachO vs. right-deep spdd and left-deep spspsp approaches, R, and commercial system A. Varied in x-direction: inter-density skew (a), shape skew (b) and intra-density skew (c). The skew types are illustrated in Figure 4.14. The bounding $\langle min,max,avg \rangle$ distributions for the data skew are $\langle 0.001, 0.5, 0.025 \rangle$ for matrix densities, and $\langle 32, 16384, 3072 \rangle$ for matrix dimensions. The intra-density skew ξ was chosen according to Figure 4.9 with values ranging from 0 to 0.5.



(a) Inter-density skew (b) Shape skew (c) Intra-density skew

Figure 4.14: Different skew types for the matrix chain multiplication experiment of Figure 4.13.

Since we created the matrices randomly for each skew parameter configuration, one single configuration can have various random instances, which results in a potentially large variety of different runtimes. This is reasoned by the fact that a skew in the matrices can affect the execution runtime in both directions – increasingly or decreasingly. Generally speaking, a large skew in the data can slow down naive execution approaches considerably, but also reveals a large optimization potential for SpMachO by exploiting the skew. In contrast to the previous self-multiplication experiment, a skew in the matrices leads to a higher influence of the parenthesization, and the selection of different storage representations and algorithms.

To increase the robustness of this measurement against outliers of particular random matrix instances, the measurements were repeated multiple times, i.e., 25 different randomly created sparse matrix chains of length three were instantiated per skew configuration. Figure 4.13 shows for each skew configuration a box plot with the corresponding median, lower quartile, upper quartile and whiskers of the execution runtimes. Note that our measured runtime of SpMachO includes the time for the density estimation that is caused by SpProdest.

We observe the following characteristics: first, for low skew parameters, both SpMachO and the right-deep spdd outperform the other approaches for most matrix instances. This underlines the result that we already obtained in the previous experiment for matrix chains with similar densities, i.e., that the right-deep spdd multiplication execution is optimal if intermediate results are rather dense. Second, and more interesting, there is an observable trend of the runtime distributions with higher data skews. In the inter-density skew experiment (left plot), the median execution time in R, the commercial system A and the left-deep spspsp approach increases, whereas the SpMachO median time stays low and drifts even lower for $\xi = 1$. Moreover, the variance in time of SpMachO grows notably slower than those of the other systems. In the other two plots, we see a similar picture, although the median execution times decrease slightly in the shape skew experiment (middle plot), and more significantly in the intra-density skew experiment (right plot). Here, the increased intra-density skew leads in the majority of cases to a reduced execution runtime. This conforms to our notion that products of matrices tend to be sparser if they have a skewed non-zero distribution, rather than a uniform one. Still, in a few cases the execution times of the other systems break out,

leading to the observed high variance and scattering of the measured times. The right-deep spdd approach is more robust, but not optimal for high skews.

As expected, the absolute runtime of SpMachO and left-deep spspsp decreases notably by using the parallelized spspsp_gemm kernel. Nonetheless, their qualitative behavior with increasing skew does not deviate significantly. To summarize, we conclude that SpMachO is able to reveal the skew and exploit it for optimization, which leads to an performance improvement by up to several factors over established systems and common alternative approaches.

4.6 RELATED WORK

In this section, we discuss the related work of multiple research areas that overlap with our optimization approach. Therefore, we subdivide the discussion into the major subtopics: optimization of linear algebra expressions, matrix chain multiplication, density estimation, and join optimization.

4.6.1 Optimizing Linear Algebra Expressions

Despite the optimization potential, we did not find that common numerical algebra systems, such as R or Matlab, optimize the execution of linear algebra expressions on large data matrices based on sparsity and dimension characteristics. In the case of matrix chain multiplication the linear algebra expression is commonly interpreted or compiled into a left-deep operator tree, if there is no parenthesization provided by the user (Tierney, 2016). However, we observed in the evaluation that a naive left-deep sparse multiplication approach can be up to orders of magnitude slower than an optimized execution.

SystemML

In contrast, the idea of optimizing linear algebra operations on system level has been picked up by the frameworks SystemML (Ghoting et al., 2011), which was already introduced in Section 2.6.3.

Matrix Multiplications. In the first paper Ghoting et al. (2011) describe two different strategies to execute a matrix multiplication, called "Cross Product based Matrix Multiplication (CPMM)" and "Replication based Matrix Multiplication (RMM)". Both are block based, and treat sparsity only on block-level: each block of the matrix can either be sparse or dense. Hence, operations on blocks, including multiplication algorithms, depend on the dynamic block type. In contrast to our adaptive tiling that is described in Chapter 5, the blocks in SystemML are of fixed size. Translated to our notation of Section 3.6, RMM is a blocked inner product formulation, and CPMM denotes a blocked outer product formulation. In their execution engine RMM is mapped into a single matrix multiplication low-level operator (LOP), whereas CPMM is split into one multiplication, and a following aggregation operation. They introduce a simple cost model to decide which of RMM vs CPMM to choose. Since SystemML is implemented on top of a distributed file system and Hadoop, their main focus is on the communication costs, in contrast to our in-memory Lapeg. Hence, they calculate costs solely by counting the memory traffic of the MapReduce shuffle-jobs and I/Os for

each matrix. Thus, unlike our in-memory cost model they are agnostic to fine variations of different multiplication kernels.

One-dimensional Cost Functions. The description of SystemML's optimizer is continued by Boehm et al. (2014b), where they introduce a simple runtime cost model based on *offline performance profiling*. As an example they use a multiplication of a (potentially sparse) matrix with its transpose, for which they created two independent, one-dimensional scaling functions: the first describes the time dependency on the data set size while the sparsity is fixed; the second describes the dependency on the matrix sparsity while the matrix dimensions are fixed. Since their data set size d_s is defined as the matrix area $d_s = m \cdot k$, their cost scaling function is unable to resolve the matrix shape (m/k) dependency. Therefore, they add a factor to correct the matrix shape influence, which is not described in more detail.

According to our observations, such simple cost models are unable to reconstruct the complicated runtime behavior of matrix multiplication kernels. In particular, our cost analysis revealed that matrix dimensions and the matrix sparsity can not be regarded as independent parameters, but have to be considered in a multidimensional cost model space. This is why we also put our focus on accurate cost models that were created from the inspection of the underlying algorithms. In addition, our model applies for any two matrices via the five generic parameters m, k, n, ρ_A, ρ_B, whereas the described scenario in their paper (Boehm et al., 2014b) is limited to the multiplication of a matrix with itself.

Density of Intermediates. Another aspect under consideration is the density estimation of intermediate result matrices. In the context of memory estimation, Boehm et al. (2014b) assume full density ($\rho = 1$) for intermediate results, which is a crude approach especially for hypersparse matrices. In contrast, SpProdest estimates intermediate densities with a tendency to overestimate density, which is a desired behavior to avoid out-of-memory situations. Nevertheless, we can set a flexible memory limit that is below the actual system limit, and leaves a buffer for matrices with a higher size than expected. In SystemML as well as in our Lapeg, the memory estimation is used by the system's resource management.

Chain Multiplications The impact of the execution order on a matrix chain multiplication was already mentioned in the first paper of Ghoting et al. (2011), where the full intermediate matrix size $(m \cdot k)$ enters the I/O cost model combined with the selection of RMM vs CPMM. However, the actual optimization approach is mentioned later in (Boehm et al., 2014a), where they state they use "a classic dynamic programming algorithm". In particular, SystemML optimizes costs by assuming "independence with regard to the sparsity of intermediates" according to the authors. In contrast, we observed that particularly in situations with large density differences (inter-density skew) the density of intermediate results influences the optimization significantly, and thus, the runtime of a matrix chain multiplication. In situations where the sparsity of intermediates is unknown, Boehm et al. (2014a) propose to dynamically recompile the execution plan. However, it is unclear how often they recompile the plan. In particular, recompiling after every sparsity-changing operation comes at a cost - firstly, it breaks the plan pipeline, and secondly, the greedy, iterative reconstruction of the execution plan dismantles the global optimality. In contrast, by leveraging density estimations for intermediate results, our initial plan is globally optimal.

Cumulon

Another system we shortly introduced in Chapter 2 is Cumulon (Huang et al., 2013), which also contains a matrix multiplication and processing engine based on execution plans. In their paper, Huang et al. (2013) mention to "conduct a series of logical uses logical rewrites" to obtain "generally better logical plans." However, it is not conveyed which rules these rewrites are based on. As an example, they mention the dense matrix chain parenthesization problem that is easily solved using the classic dynamic programming approach. It is not indicated that they use density estimates for intermediate matrices at all.

RIOT-DB

The system RIOT-DB (see Section 2.6.1) executes R code with the data being stored in a relational DBMS. In (Zhang et al., 2009) they also propose to integrate high-level linear algebra optimizations into their system, and use matrix chain multiplication as an example. In particular, their aim is to optimize the expression in an I/O model, assuming the matrices on disk are larger than the available memory. However, their cost model assumes dense matrices and only takes I/O transfers into account, which also exhibits a $\mathcal{O}(mkn)$ complexity like in the conventional dense matrix chain problem. Sparsity is not addressed in their optimization, although it might impact the I/O costs significantly.

4.6.2 Sparse Matrix Chain Multiplication and Density Estimation

As mentioned before, the optimization of dense matrix chain multiplications has been discussed thoroughly in the past decades (e.g., in Cormen et al., 2001; Godbole, 1973; Lee et al., 2003). Throughout the related work, the problem of finding the best parenthesization is consistently solved by dynamic programming. However, there is only little work about sparse matrix chain multiplications.

An interesting paper that should be mentioned in this context was written by Cohen (1996). In her work, Cohen extends the dynamic programming approach idea to sparse matrices. Therefore, she minimizes the overall number of floating point operations that are needed to compute the matrix chain product. To obtain the non-zero structure for intermediate result matrices, Cohen leverages a prediction technique on matrix row/column-level.

Density Estimation

According to Cohen and Gilbert and Ng (1993), who she is referring to, "many sparse matrix algorithms have a phase that predicts the nonzero structure of the solution from the nonzero structure of the problem." A closer look at the related work by Gilbert (1994) reveals the prediction algorithm, which is based on an inspection of the corresponding bipartite graph representation of the matrix. Moreover, most algorithms are concerned with the nonzero prediction of direct sparse matrix factorizations, which might have a more complex access pattern than matrix multiplications. Nevertheless, the nonzero structure of a sparse matrix-matrix product according to Gilbert (1994) is obtained by generalizing the prediction of a sparse matrix-vector product. However, it seems

that his method, which is not explicitly described in (Gilbert, 1994), visits the full graph or matrix topology in the same way as the multiplication itself, just without considering the numerical values. This could be inefficient, as we sketch in the following case study: assume a sparse multiplication is bound by the memory bandwidth[3], and the matrices are stored in a CSR representation with integer type for the coordinates, and double type for the values. Then, the prediction method loads at least one third of the data that is loaded for the actual multiplication, resulting in an overhead that is at least one third of the actual multiplication runtime. In contrast, our approximate method based on density maps is far more lightweight by requiring only a vanishing fraction of the actual multiplication time, and the approximation is sufficient in most cases. Unfortunately, Gilbert (1994) did also not add any performance evaluation of their prediction techniques.

Sparse Matrix Chains

The density prediction algorithm of sparse matrix chain multiplication in (Cohen, 1996) is based on random number propagation in a layered graph. Therein, each layer i contains m_i vertices that belong both to the row dimension of $\boldsymbol{A}_p$, and to the column dimension of $\boldsymbol{A}_{p-1}$. The edges between layers $(p-1)$ and p correspond to the non-zero elements of matrix $\boldsymbol{A}_p$. To determine the density of a particular row i in the intermediate matrix $\boldsymbol{A}_{[p..l]}$, ν random numbers are drawn for each vertex in layer l. Then, the minima for each number in the ν random number series over all vertices are propagated through the edges from layer l to p. Based on the distribution of ν absolute values that have reached the row vertex i in layer p, an estimate for the row density is derived. Cohen's algorithm has the asymptotic complexity of $\mathcal{O}(\nu p \cdot \sum^{p} N_{nz,i} + p^3 N)$ for a matrix chain of length p and maximum matrix dimension N, which might be very costly. Moreover, due to the randomness the accuracy of the method significantly depends on the number ν of random instances. We can expect that, for a sufficiently high ν, Cohen's method would provide relatively accurate estimates on a row- and column-based level. Nonetheless, the granularity of estimates on matrix row/column level could not be fully exploited by our cost model as of the current state. Cohen's approach might still be used as an extension to our estimator in situations where a high accuracy is promising, e.g. for very skewed non-zero patterns. However, we also observed that the actual runtime costs of sparse matrix multiplication kernels are not just proportional to the number of floating point operations (see Section 4.2.) Moreover, from a real system perspective, we not only consider plain sparse-sparse matrix multiplications, but leverage sparse-dense transformations, and exploit the coexistence of sparse and dense matrices to optimize on a more complete level.

4.6.3 Join Optimization

The problem of sparse matrix chain multiplication is related to join enumeration and cardinality estimation in a relational database management system. In fact, the connection becomes more obvious when sparse matrices are represented as $\langle row, col, val \rangle$ triple tables, as sketched in Figure 3.3c1 of Section 3.4. As we have already shown by the SQL expression in Listing 2.1, Section 2.6.1, a matrix multiplication can be expressed as a join on $\boldsymbol{A}.col = \boldsymbol{B}.row$, followed by a sum aggregation, which groups the matrix values by their target coordinates $\boldsymbol{A}.row, \boldsymbol{B}.col$. Vice versa, a certain type of

[3]This is the case for sparse matrix-vector multiplications, see Section 3.5

join could also be re-interpreted as a sparse matrix multiplication. For example, Amossen and Pagh (2009) describe how they accelerate a join-project by using a fast Boolean matrix multiplication.

Multi-way Join Order

A join of multiple tables (multi-way join) can be executed in many ways. Similar to the case of matrix chain multiplication, the size (or cardinalities) of intermediate join products influence the execution time, as well as the selection of different join algorithms. Indeed, the use of dynamic programming in join optimization is common sense. As early as four decades ago Selinger et al. (1979) proposed a cost-based optimization of join plans and join enumeration using a bottom-up dynamic programming approach in "System R". In a recent paper, Leis et al. (2015) survey and evaluate query optimizers of state-of-the-art DBMS. They found that the query optimization of multi-join queries primarily depends on the quality of the cardinality estimation, and bad plans mostly yield from estimation errors that quickly grow as the number of joins increase. Furthermore, the authors conclude that the query optimization depends only insignificantly on *accurate* cost models. This result, however, might be true for most join algorithms that all depend on the number of tuples, but is definitely not applicable to our setup, which consists of sparse and dense multiplication kernels each having a completely different dependency on the matrix density.

In contrast to matrix multiplications, joins are generally commutative. For instance, consider a three-way join on an attribute that is common to all tables: $T_1 \bowtie_k T_2 \bowtie_k T_3$. This join could be executed as $(T_1 \bowtie_k T_2) \bowtie_k T_3$, but alternatively in the order $(T_1 \bowtie_k T_3) \bowtie_k T_2$, which commutes table T_2 with T_3. An explicit connection from joins to matrix chain multiplication is drawn by Moerkotte (2003), who considers *order-preserving* joins. These are non-commutative and thus, more closely related to our problem. However, the cost model shown by Moerkotte (2003) merely consists of summing cardinalities. Despite being adequate for relational joins, it can not be used in the same way for sparse matrix multiplication, since the intermediate cardinality of the join $\boldsymbol{A}.col = \boldsymbol{B}.row$ is much higher than the result size of a matrix product. This is due to the aggregation step (Amossen and Pagh, 2009), which reduces the size of the matrix. Naturally, this aggregation has also to be considered in the matrix density estimation, and is the reason why simple join cardinality estimations can not be applied to our case.

Cardinality Estimation

Some parts of our approach to estimate the density of sparse matrix intermediates are related to methods used for cardinality estimation used for relational query processing. In particular, attribute histograms are used for estimating the selectivity of a query (e.g. in Piatetsky-Shapiro and Connell, 1984), approximate query answering and load balancing in join execution. A good overview of using histograms in databases is given by (Ioannidis, 2003). In particular, multidimensional histograms can be used for the optimization of queries on multidimensional data (e.g. in Muralikrishna and DeWitt, 1988). Moreover, the concept of integrating physical properties into the optimization process, such as the data representation, has also been treated in the context of relational plan generation. For example, Graefe and McKenna (1993) outline the dependency of the plan selection on physical properties of the corresponding intermediate results. It should be mentioned here that we prefer *equi-width* multidimensional histograms (divided into buckets of equal areas) for the density map

rather than the *equi-depth* (divided into buckets of equal number of entries and varying areas). This is because equi-depth histograms would complicate the density estimation and thus, negatively impact the performance of SpProdest.

To summarize, many aspects of the works in the relational universe were inspiring for this work. Although the mathematical characteristics of matrices and multiplications require a slightly different perspective, we find it an interesting result that some ideas of relational join optimization can be used in a similar way for linear algebra.

4.7 SUMMARY

We argued in the introduction that integrating linear algebra operations, such as matrix multiplications, is not just adding data structures and algorithms to the database engine. This chapter revealed that the existence of an expression-based language interface incurs additional complexity, especially if linear algebra expressions should be executed in an optimized way. In fact, due to different matrix representations, algorithms, and the presence of data skew, we observed that a naive execution of sparse matrix chain multiplications can be up to orders of magnitude slower than an optimized one.

In this chapter we presented SpMachO, the optimizer component in the logical layer of our Lapeg. SpMachO optimizes sparse, dense and mixed matrix multiplications of arbitrary length by creating an execution plan, which consists of transformation and multiplication operators. By using a detailed cost model of different sparse, dense and mixed matrix multiplication kernels, SpMachO leads to a faster and more robust execution compared to widely used algebra systems. The required estimation of intermediate matrix densities is handled by our prediction approach SpProdest, which is more efficient compared to conventional methods by working on density sketches rather than visiting the complete matrix representation. Moreover, with an entropy-based skew awareness, SpProdest is able to dynamically adjust the granularity of the density map, and enables accurate memory consumption and runtime estimates at each stage in the execution plan. The density estimate of SpProdest is further used for a resource-aware, dynamic selection of the output matrix representation in the context of our adaptively tiled multiplication operator (ATmult), which is presented in Chapter 5.

Finally and most impressively, we were able to show how optimization methods inspired from database technology can improve the execution of linear algebra expression. In particular, by leaving the optimized execution of expressions to the system we reduce the computational complexity for data scientists – who should not be required to have profound knowledge about the connections between mathematical optimization, matrix characteristics, algorithmic complexities and the hardware parameters of their system.

5 ADAPTIVE MATRICES

5.1 MOTIVATION

In Chapter 3, we introduced data structures for sparse and dense matrices in the column-oriented storage layer. In particular, we showed that efficient matrix representations seamlessly integrate with the columnar layout. Nevertheless, so far we assumed that the user is required to *predefine* the final data structure of a matrix, for example whether it is stored in a sparse or in a dense format. The same selection must be made by users of math software, such as R, MATLAB or BLAS libraries, where they have to choose among an even wider variety of sparse matrix types. However, the pre-configuration of matrix storage types is disadvantageous, since it can lead to a negative impact on the performance, as we observed for sparse matrix chain multiplications in Chapter 4. However, not only does the distinction between plain dense and sparse matrix operands influence the execution performance, but also the physical organization of each matrix on its own. A matrix is commonly stored as a whole in a static, homogeneous sparse or dense format (e.g., in SCALAPACK), resulting in poor memory utilization and processing performance if the data representation was not chosen wisely. In fact, there is additional tuning potential that can be leveraged when matrices are considered as *heterogeneous* objects. Conventional multiplication algorithms are agnostic of density variations within matrices, whereas at the same time there are many efficient routines for either plain sparse or plain dense matrices. Hence, the idea is that the LAPEG splits each matrix into several sparse, dense, and potentially further constituents, in order to decompose a single multiplication operation into multiple optimized sub-multiplications. This motivated us to rethink and redesign data structures, and processing practices for large, sparse data matrices.

In this chapter, we push the envelope further towards a fully dynamic, and adaptive physical organization of matrices, completely transparent to the user. Therefore, we developed an adaptive tile matrix (AT MATRIX) data structure, which uses an optimized data layout based on the matrix non-zero topology. Moreover, we present a matrix multiplication operator ATMULT that is able to exploit the heterogeneity of the AT MATRIX, by using optimization techniques that are inspired from relational query processing.

The core contributions comprised in this chapter are:

1. A hybrid matrix representation AT MATRIX consisting of adaptive tiles to store large sparse and dense matrices of any individual non-zero pattern efficiently.

2. A time- and space-efficient matrix multiplication operator ATMULT that performs dynamic tile-granular optimizations based on density estimates and a sophisticated cost model.
3. The adoption of several methods based on database technologies for ATMULT, such as 2D indexing, cardinality estimation or just-in-time partial data conversions.
4. An evaluation of AT MATRIX and ATMULT using a wide range of synthetic and real world matrices from various domains.

Section 5.2 starts with the description of our adaptive tiled matrix data type, followed by a description of the partitioning algorithm that converts a raw matrix into an AT MATRIX. In Section 5.3 we present our matrix multiplication operator ATMULT, including our tile-granular optimization approach that uses just-in-time data conversions. Finally, the extensive evaluation of ATMULT is presented in Section 5.5.

5.2 ADAPTIVE TILE MATRIX

In the following, we discuss the problem of selecting an appropriate data representation for large, sparse matrices with arbitrary non-zero topologies, which has barely been addressed by the related work. Thereafter, we propose a self-organizing matrix representation that automates the creation of an optimized data layout for a matrix of arbitrary shape, density, and non-zero pattern.

5.2.1 Choosing the Right Representation

Scientists with a strong algorithmic background typically chose the matrix representation in accordance with the used algorithms, which often have distinct data access patterns. Most of the efficient and high-performance algorithms are even particularly written for a certain data representation. As a consequence, the selection of the data structure has a significant impact on the processing performance, and accordingly on the execution time of the analysis. Naturally, the selection of data representation and algorithm should also depend on matrix characteristics, such as the matrix density ρ, if only to reduce the memory consumption. However, we observed that the widely applied, crude separation between plain dense and plain sparse matrices does not match the particularities of large matrices from real world scenarios. Their non-zero patterns are usually non-uniform and often contain distinct regions of higher and lower densities. Therefore, our adaptive, heterogeneous data representation (AT MATRIX) is able to accommodate individual dense or sparse substructures for matrices of any nature.

As mentioned in Section 3.4.2, numeric libraries such as the sparse BLAS (Duff et al., 1997) or SuiteSparse (Chen et al., 2008) offer a variety of sparse matrix types. Besides the triple representation (aka coordinate format), and the indexed CSR/CSC, there are many more representations, such as diagonal, band-diagonal, block-CSR, and others (for more details, we again refer to Saad (1994)). However, these representations are mostly designed for distinct non-zero patterns, such as band, nearly diagonal, bock-diagonal matrices, but are poorly suited for general matrices with a deviating pattern. We emphasize that our approach significantly differs from any static sparse data format in the way that it *dynamically adapts* to any non-zero pattern by using complementary local storage types for arbitrary sized matrix subregions. Indeed, heterogeneous blocked matrix types are offered by some systems (Ghoting et al., 2011; Huang et al., 2013; Zadeh et al., 2015), but in all

cases the fixed block-size is predefined, and often the user has to pre-determine the storage type for each block (Zadeh et al., 2015).

Nonetheless, those static data structures are yet widely used even for applications that work with very large matrices. Indeed, we identified several reasons for why one should refrain from using them naively: first of all, without a detailed knowledge about the data and the corresponding non-zero pattern, it is unlikely that a user will pick the *best* data structure for a sparse matrix with respect to either algorithm performance or memory consumption. Moreover, we observed in our experiments that plain CSR-based matrix multiplication does not scale well on multi-core machines. This can be explained with the decreased cache locality due to large matrix dimensions. Similar observations were also made in related work (e.g., Patwary et al., 2015). Finally, large sparse matrices often have a topology with distinct areas of a significantly higher density. Using dense subarrays reveals a significant optimization opportunity by exploiting efficient dense multiplication kernels.

5.2.2 Adaptive Tiles

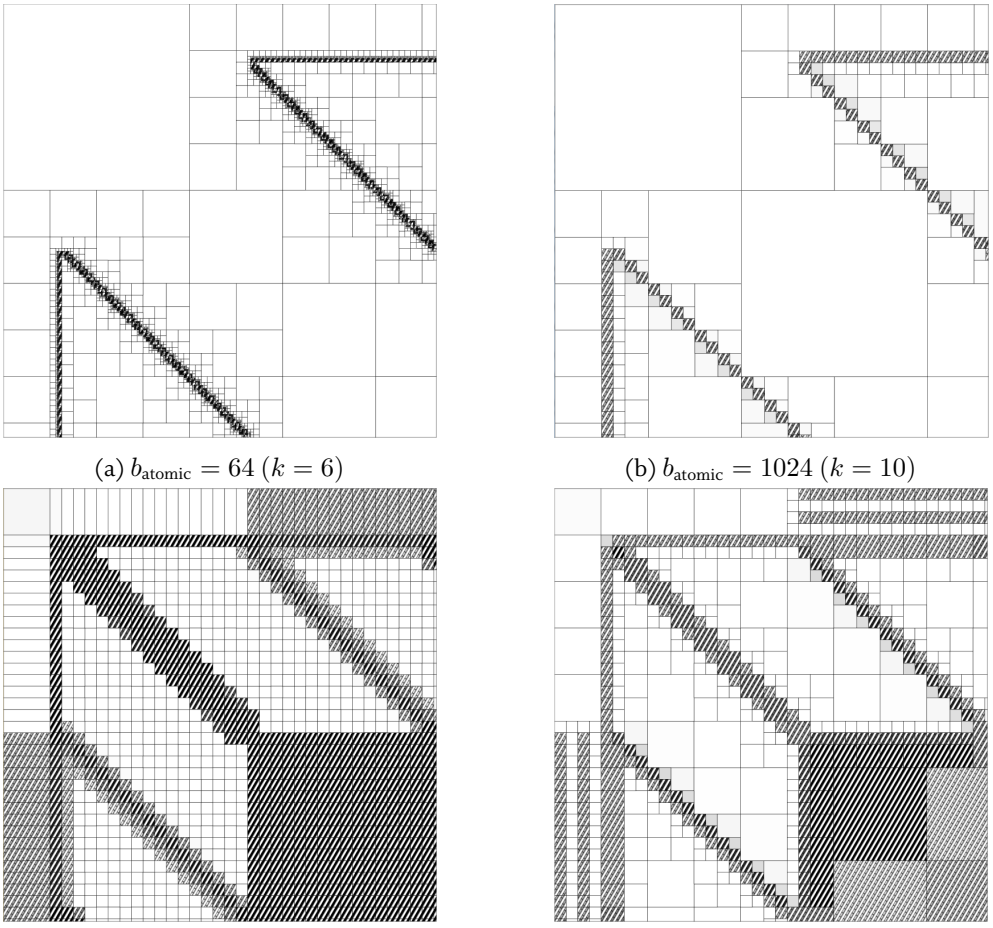
Due to the bad scaling behavior of naive sparse data representations, and the performance improvements that are incurred by using dense algorithms for dense subparts of matrices, we made two major design choices for our AT Matrix structure: firstly, for a better cache locality of dense and sparse algorithms, each matrix is *tiled* with a variable tile size. Secondly, tiles with a high population density are stored as a dense array, tiles of a lower density have a CSR representation, resulting in a *heterogeneous* matrix structure. In the following, we will discuss the criteria for the maximum and minimum tile sizes.

The selection criteria of the individual tile representation is described later together with our partitioning algorithm. For now, it is sufficient to know the representation is chosen according to a density threshold value that minimizes the runtime costs of the algorithm under consideration, e.g., matrix multiplication.

Note that we distinguish in our notation between physical matrix *tiles* and logical, atomic *blocks*: tiles define the bounding box of the physical representation that accommodates the corresponding part of the matrix, and may have variable sizes. In contrast, a logical block is *atomic*, i.e., it refers to a square area of the fixed size $b_{\text{atomic}} \times b_{\text{atomic}}$ that is not split any further. Hence, a logical block is our unit of granularity. Its internal density is approximated as uniform, and no heterogeneity is resolved below its block size. Except for corner situations (e.g., at matrix boundaries that are not aligned to the block grid,) a physical matrix tile is greater than or equal to a logical block, and could potentially span a region of multiple logical blocks. The tiles are adaptive, thus, the size is chosen such that an AT Matrix tile covers the greatest possible matrix part of either a low or a high population density, without exceeding the maximum tile size.

Maximum Tile Size

For dense tiles, the maximum tile size $\tau^d_{\text{max}} \times \tau^d_{\text{max}}$ is fixed and chosen such that α tiles fit in the last level cache (LLC) of the underlying system. For example, a parameter $\alpha = 3$ preserves the cache locality for a dense matrix tile multiplication, since all three tiles (left input-, right input-, and target tile) would fit in the cache. Regarding sparse tiles, there are two upper bounds for the size: the first is variable and such that a sparse tile with density ρ does not occupy more memory than $\frac{1}{\alpha}$ of the



(a) $b_{\text{atomic}} = 64$ $(k = 6)$ (b) $b_{\text{atomic}} = 1024$ $(k = 10)$

(c) Estimated $\hat{\rho}_C$ (d) Result $C = \text{R3} \cdot \text{R3}$

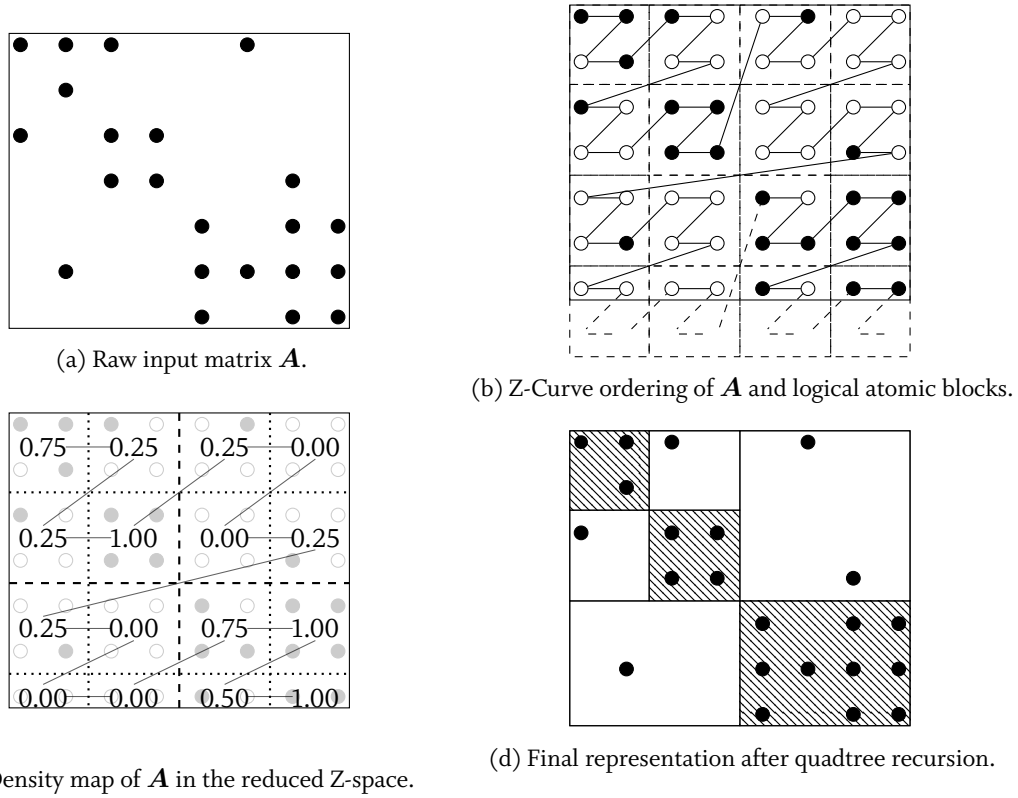
Figure 5.1: The TSOPF_RS_b283 matrix (R3) as AT Matrix using different granularities (a,b), the density estimation of the self-multiplication (c), and the actual result (d). The grayscale indicates the population density of sparse tiles, dense tiles are marked with a diagonal pattern.

LLC size. The other bound is a bit more subtle: it is dimension-based, i.e., chosen such that at least β 1D-arrays with the length of one tile-width should fit in the LLC, which is motivated by the width of accumulator arrays used in sparse matrix multiplication kernels (see *sp_acc* in Algorithm 10.) Hence, the system-dependent maximum tile sizes are calculated as:

$$\tau_{\text{max}}^{d} = \sqrt{\frac{\text{LLC}}{\alpha \mathcal{S}_d}}, \tag{5.1}$$

and

$$\tau_{\text{max}}^{\text{sp}} = \min\left\{ \sqrt{\frac{\text{LLC}}{\alpha \rho \mathcal{S}_{sp}}}, \frac{\text{LLC}}{\beta \mathcal{S}_d} \right\}, \tag{5.2}$$



(a) Raw input matrix $\boldsymbol{A}$.

(b) Z-Curve ordering of $\boldsymbol{A}$ and logical atomic blocks.

(c) Density map of $\boldsymbol{A}$ in the reduced Z-space.

(d) Final representation after quadtree recursion.

Figure 5.2: Schematic illustration of the quadtree partitioning using a sparse 7×8 matrix and a 2×2 block granularity. The diagonal stripe pattern depicts dense tiles, the remaining tiles are sparse.

with $\mathcal{S}_{d/sp}$ referring to the size of a matrix element in bytes: $\mathcal{S}_d = 8$ bytes in the dense array representation and $\mathcal{S}_{sp} \approx 16$ bytes in the sparse CSR representation due to the additional storage that is required for the element coordinates. For our experiments, we chose $\alpha = \beta = 3$, which assures that at least three matrix tiles fit in the LLC at the same time. Since the LLC is shared between all threads of a CPU socket, the cache locality also depends on concurrency, as well as on the data and thread distribution. Hence, the ideal tile size and values of α, β might deviate from our heuristic selection, leaving room for further tuning. However, the actual selection of α, β also depends on the alignment constraints imposed on the tile size, which we describe in Section 5.2.3.

Minimum Tile Size

The minimum size of matrix tiles defines the granularity of the AT Matrix, and is equal to the logical block size. The selection of the granularity is generally a trade-off between overhead and optimization opportunity: the finer grained the tiles are, the higher is the administrative cost of the AT Matrix. Too small atomic blocks not only result in a higher overhead of any algorithm dealing with the partitioned AT Matrix structure, but also increase the recursion depth of the partitioning

algorithm described in Section 5.2.3. In contrast, very coarse grained tiles might not be able to resolve the heterogeneous substructure of a matrix.

For now, we restrict the dimensions of logical blocks to be a power of two, i.e., $b_{\text{atomic}} = 2^k$, which is in line with our quad-tree-based partitioning approach[1]. Naturally, k has to be adapted to the system configuration, since the optimal minimum tile size depends on hardware parameters such as the cache size. For a system with a last level cache of 24MB[2], our multiplication experiments have shown the best results for $k = 10$, which yields $b_{\text{atomic}} = 1024$ and is equal to the maximum dense tile size τ^d_{max} of Equation (5.1).

Figures 5.1a-5.1b show the AT Matrix layout for the matrix R3 (see Table A.1 for details) and the two different granularities $k = 6$ and $k = 10$. The heterogeneous tiling yield a significant performance gain by using different multiplication kernels, which is exploited by our ATMult operator that we present in Section 5.3. In fact, the multiplication result matrix in Figure 5.1d shows a substructure with distinct sparse and dense regions, underlining the motivation for a topology-aware matrix data type.

Unlike our AT Matrix approach, a naive matrix tiling with *fixed* block size, as it is done in some implementations (Blackford et al., 1997; Zadeh et al., 2015; Boehm et al., 2014b; Huang et al., 2013), often results in adding overhead without any positive impact on the runtime performance. This holds in particular for hypersparse matrices, thus when blocks are badly dimensioned and barely contain any element. Therefore, our sparse tiles are adaptive such that the tile size depends on the number of non-zero elements: they logically "grow" until the maximum tile size as of Equation (5.2) is reached. As a consequence, sparse, and in particular large, hypersparse matrices without notable dense subregions are not split, as long as they do not exceed a certain dimension threshold. Assuming an LLC of size 24 MB and following Equation (5.2), a sparse matrix with dimensions of $300,000 \times 300,000$ and a homogeneous density of $\rho = 10^{-5}$ would be stored in a single, sparse tile. Thus, no overhead is added by our AT Matrix structure that creates a tile substructure only if it is beneficial for later processing.

5.2.3 Recursive Matrix Partitioning

We developed a recursive partitioning method that identifies regions of different density and sizes in the matrix topology, and creates the corresponding matrix tiles in the AT Matrix representation.

It is worth mentioning here that despite one being able to construct the matrix representation in a dynamically growing fashion, e.g., by iteratively inserting elements into balanced two-dimensional tree structures such as the UB-tree (Markl, 1999), or LAB-tree (Zhang et al., 2011), we pursue a different approach. Many of the matrices in analytical applications are *initially* loaded from a file, or are contained relational tables that can be re-interpreted as matrices (as we showed in Chapter 3). Since the costs for incrementally constructing a tree-structured matrix from scratch are prohibitive, our approach is to pursue an initial bulk reorganization process that efficiently creates an AT Matrix. Note that our AT Matrix representation is nevertheless updatable after being created. We address data manipulation operations on AT Matrix and other representations in Chapter 6.

[1] Other atomic block sizes are conceptually possible, but might complicate calculations to transform coordinates back and forth from the two-dimensional matrix space into the one-dimensional linearized space. See Section 5.2.3 for more details.

[2] Comparable with our test environment

Algorithm 14 Recursive Quadtree Partitioning

1: **function** RECQTPART(ZMatrix Src, $ZBlockCnts[]$, AT MATRIX Tgt, $zStart$, $zEnd$)
2: $range \leftarrow zEnd - zStart$
3: **if** $range == 0$ **then** ▷ Recursion anchor.
4: **if** $ZBlockCnts[zStart] == -1$ **then** ▷ Check if block is outside matrix bounds.
5: **return** (*OUT_OF_BOUNDS*, 0)
6: **else** ▷ Else calculate density and return to higher recursion level.
7: $\rho \leftarrow$ CALCDENSITY($ZBlockCnts[zStart]$)
8: **return** (*FORWARD*, ρ)
9: **else** ▷ Enter quadtree-recursion.
10: $stride \leftarrow (zEnd - zStart + 1)/4$
11: *UL* $\leftarrow$ RECQTPART($Src, ZBlockCnts[], Tgt, zStart, zStart + stride$)
12: *UR* $\leftarrow$ RECQTPART($Src, ZBlockCnts[], Tgt, zStart + stride, zStart + 2 \cdot stride$)
13: *LL* $\leftarrow$ RECQTPART($Src, ZBlockCnts[], Tgt, zStart+2 \cdot stride, zStart+3 \cdot stride$)
14: *LR* $\leftarrow$ RECQTPART($Src, ZBlockCnts[], Tgt, zStart+3 \cdot stride, zStart+4 \cdot stride$)
15: **if** *UL*,*UR*,*LL*,*LR* **homogeneous and** $range <$*MAXSIZE* **then**
16: **return** (*FORWARD*, AVG(*UL*, *UR*, *LL*, *LR*)) ▷ Pass average density to higher recursion level.
17: **else**
18: **return** (*MATERIALIZED*,
Tgt.INSERT(Src.MATERIALIZETILES(*UL*, *UR*, *LL*, *LR*, ρ_0^R))) ▷ Create physical tiles and insert into the AT MATRIX target.

The matrix reorganization, or partitioning process, can be divided into the components: loading, reordering, identifying, and materialization. It is illustrated in Figure 5.2 for an exemplary, small sparse matrix.

Locality-Aware Element Reordering

Firstly, the initial matrix data is loaded into a temporary, unordered staging representation, which is simply the matrix triple table as of Figure 3.3c1, containing the coordinates and values of each matrix element. In order to identify areas of different densities in the two-dimensional matrix space, it is crucial to transform the staged raw matrix into a format that preserves the 2D-locality of matrix elements in memory. Obviously, locality is not well preserved for the widely used row-major and column-major layouts: two elements that are placed in the same column and in neighboring rows of an $m \times n$ row-major matrix have a distance of up to n elements ($n\mathcal{S}_{sp}$ bytes in memory), and equivalently, two adjacent elements in one row in a column-major matrix have a potential distance of up to m elements. In order to iterate in either column or row direction, a large memory segment of the matrix needs to be scanned, which is unfavorable for locality-aware algorithms that process data in two-dimensional space.

Instead, our algorithm recurses on two-dimensional quadtrees by using a quadtree-aligned storage layout that preserves locality in both dimensions. In particular, all elements of the matrix are reordered according to a *space-filling curve*. This technique is inspired from indices for spatial or multidimensional data, such as in Hilbert R-trees (Kamel and Faloutsos, 1994) or UB-trees (Markl, 1999). In fact, there are several space-filling curves that provide a quadtree ordering, and many are equally suited for our recursive partitioning method. We decided in favor of the Z-curve, also called Morton order (Morton, 1966), since the Z-value can be efficiently computed with bit interleaving (Markl, 1999). The locality of two matrix-elements is effectively preserved within one quadrant, which is an internal node of the quadtree. This property is recursive, such that the four child quadrants of any node are always stored consecutively in memory, as sketched in Figures 5.2b-5.2c. As a matter of fact, the quadrant dimensions are aligned to a power of two, since with each recursion the quadrant covers exactly a fourth of its parent area. In order to compute the minimal square Z-space that is required to cover the complete matrix with a Z-curve, both matrix dimensions are logically padded to the next largest common power of two. This results in a Z-space size of $K = 4^{\max\{\lceil \log_2 m \rceil, \lceil \log_2 n \rceil\}}$.

Identifying Sparse and Dense Submatrices

Algorithm 14 sketches our recursive partitioning routine, which takes the Z-ordered source matrix, the AT Matrix target, two Z-coordinates, and the buffer array *ZBlockCnts[]* as arguments. The latter stores the number of non-zero elements in each logical block $b_{\text{atomic}} \times b_{\text{atomic}}$, and is precalculated by a single scan over the keys of the Z-ordered source matrix. As mentioned before, our atomic block dimensions are aligned to a power of two. This is because the logical block size must be aligned to the recursive quadtree pattern that is implied by the logically padding of K. The resulting *ZBlockCnts* array is also Z-ordered, but due to the blocking, the full matrix Z-space of size K is reduced by the factor of b^2_{atomic} (Fig. 5.2c), and consequently, the recursion depth by a factor of $\log_2 b_{\text{atomic}}$.

The algorithm recurses on the *ZBlockCnts*-array on the range that is defined by the Z-coordinates *zStart* and *zEnd*. In particular, the four subranges in-between *zStart* and *zEnd* correspond to the upper-left (*UL*), upper-right (*UR*), lower-left (*LL*) and lower-right (*LR*) subsquare of the corresponding recursion step in the Z-space (line 11-14). The anchor is reached when *range* is zero (line 3), which means that the subsquares have the minimal block size. On the bottom-up recursion path, it is checked if all four neighboring blocks are homogeneous – i.e., if all of them are of the same type, and neither of the two maximum block criteria of Equations (5.1&5.2) is met. If so, the homogeneous blocks are logically melted into a four times larger block, which is then returned (*forwarded*) to the recursion level above. Instead, if there are neighboring blocks of different density types at any recursion level, then each of the four corresponding blocks is *materialized* into either a sparse or a dense matrix tile (line 18). Finally, the references of the resulting tiles are added to the target AT Matrix.

Depending on the matrix shape and dimension, the padding of the Z-space can result in a few logical blocks in the *ZBlockCnts* array that are outside of the actual matrix bounds, like the lower parts in Figure 5.2c. However, these are ignored in the recursion and do not contribute to the submatrix materialization. Figure 5.2d sketches the resulting AT Matrix from Algorithm 14 on the sample matrix of Figure 5.2a.

In total, the computational complexity of the restructuring algorithm for a matrix $\boldsymbol{A}$ with N_{nz}^A elements is in $\mathcal{O}\left(N_{nz}^A + K/b_{\text{atomic}}^2 + N_{nz}^A \log N_{nz}^A\right)$. The sort term stems from both the preceding Z-ordering and the sparse tile materialization. In contrast, the complexity of a sparse matrix multiplication $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{A}$ is given as $\mathcal{O}\left(N_{nz}^A \sqrt{N_{nz}^C}\right)$ in related work (Amossen and Pagh, 2009; Pagh and StÃűckel, 2014). We show in our experiments that the partitioning costs are usually compensated by the performance gain of a single multiplication, except for situations where N_{nz}^C is small or K/b_{atomic}^2 is extremely large (hypersparse matrices).

Quality Considerations

Our prior aim of the heterogeneous tiling is to optimize the processing performance of large sparse matrices, with the focus on matrix multiplication. Unfortunately it is difficult to judge the quality of the obtained partitioning for a single matrix under this aspect, since the performance of a multiplication heavily depends on the second matrix, which is not known before runtime.

A key aspect of the partitioning algorithm is to decide if a given logical matrix block with density $\rho_{(ij)}$ is treated as sparse or dense, which is part of the homogeneity check of Algorithm 14. In particular, it is checked whether $\rho_{(ij)}$ exceeds the density read threshold value ρ_0^R. The latter is chosen in accordance with the *density turnaround point*: it is effectively the critical density value at the intersection of the multiplication cost functions, at which the dense algorithm starts to be more time-efficient than the sparse algorithm. We already identified this point in the discussion about the multiplication cost model in Section 4.3. In Figure 4.5 we observed that using a dense matrix representation is mostly more efficient if the matrix density exceeds a value of about $\rho_{ij} \geq 0.3$. However, since the critical density value also depends on the other dimensions m, n, k, and might deviate among the eight-fold cost functions, it is not clearly defined, and is only approximated by ρ_0^R. Depending on the actual characteristics of the second matrix tile at runtime, the threshold ρ_0^R might deviate significantly from the *actual* critical density value of the corresponding cost function. In this case, the matrix tile might not be present in the optimal representation. Nevertheless, such a situation is detected by our runtime multiplication optimizer of ATMULT, which triggers a tile conversion from a sparse to a dense representation (or vice versa) at runtime. In the worst case, every matrix tile is later converted into another representation. We simulated this situation in our evaluation (Section 5.5, Figure 5.11b- 5.11a) by multiplying a heterogeneous sparse matrix with a full matrix, which, however, resulted in a conversion overhead of not more than 10% of the total runtime. Since the cost model is not only dependent on the second matrix density, but also on the system configuration, ρ_0^R is besides b_{atomic} another adaptable tuning parameter.

The memory consumption of the AT MATRIX can either be lower or higher than that of a pure sparse matrix representation, but is always lower than or equal to that of a plain dense array. The worst case is present when all tiles have densities slightly above ρ_0^R. Then, the matrix would be stored as dense, consuming $\mathcal{S}_d/(\rho_0^R \mathcal{S}_{sp})$ as much memory as the sparse representation (maximum 2x in our configuration).

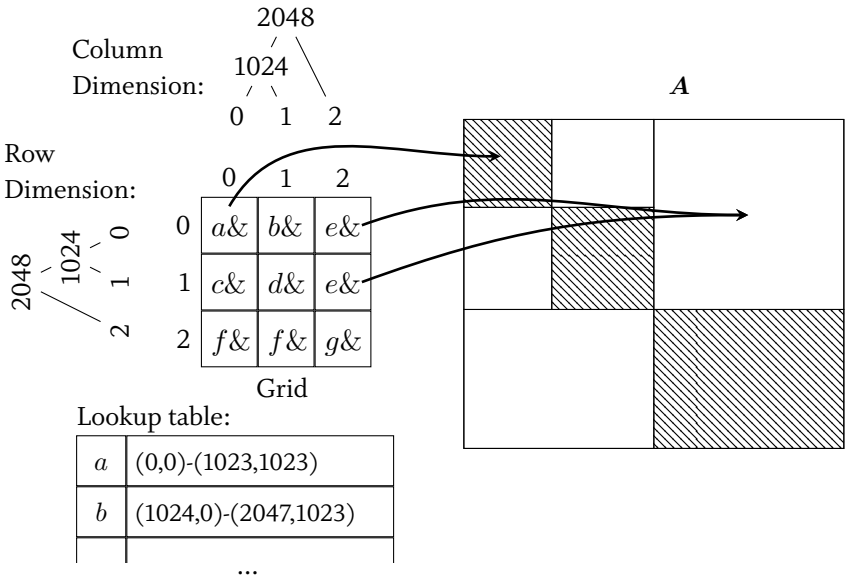


Figure 5.3: The internal organization of the AT Matrix. Matrix $\boldsymbol{A}$ is assumed to have dimensions 3500×3500.

5.2.4 Internal Structure

The organization of the adaptive tiles in the AT Matrix is internally realized via a non-periodic grid index. It is similar to the organization of the Grid File (Nievergelt et al., 1984), but instead of buckets, the two-dimensional grid indexes adaptive matrix tiles. Figure 5.3 sketches the different physical components of the AT Matrix. We utilize a dynamic 2D-array that contains references of tile instances. The tile instances are regular instances of either a sparse (CSR) or a dense matrix in the column-oriented storage layer, as described in Section 3.4. In order to query matrix $\boldsymbol{A}$, the AT Matrix contains a row-, and a column dimension-index, in the form of two separate search trees. Note that having two separate indexes is advantageous over a single, linearized tree as the LAB-tree (Zhang et al., 2011), since most operations process matrix tiles in either row- or column direction, e.g., in the blocked matrix multiplication. The leaf nodes map from a range in absolute matrix coordinates to the corresponding index in the grid. The leaf-ranges are non-periodic, and depend on the actual topology of the matrix. Let the matrix in Figure 5.3 have the dimensions 3500×3500. Hence, with $b_{\text{atomic}} = 1024$, there can at most be 4 intervals in each direction, thus a maximum of $4 \times 4 = 16$ tiles in total. Due to the particular topology of $\boldsymbol{A}$, the row-dimension is split into only three intervals {[0,1023], [1024,2047], [2048,3499]}. Furthermore, the example matrix is symmetric, hence, the matrix row- and column-dimension indexes are equivalent. In general this is not the case, and the grid can index arbitrary non-symmetric matrices. It is worthwhile mentioning that each grid entry refers to an area that has at least the size of a logical block, but at most covers one physical tile. In particular, one matrix tile is potentially referenced by multiple grid entries. For instance, *Grid[0][2]* and *Grid[1][2]* both point to tile e in Figure 5.3.

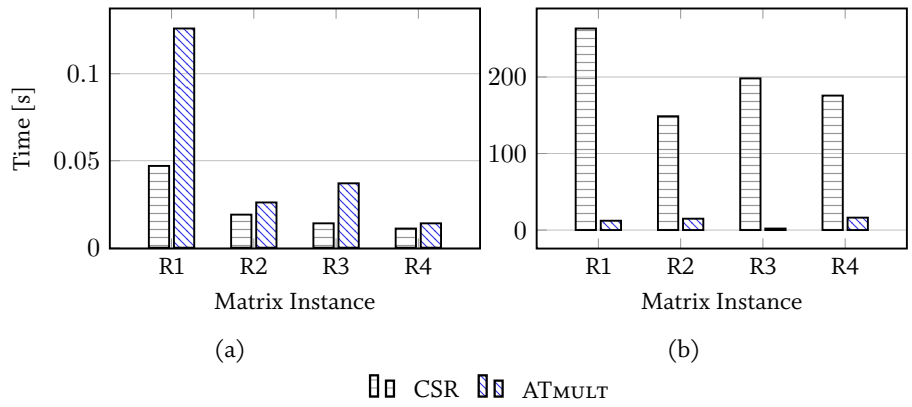


Figure 5.4: The performance for reading 2000 random single rows (a) and columns (b) of an AT Matrix vs. a naive CSR representation for several real world matrices of Table A.1.

The translation of absolute matrix coordinates (i, j) into the grid coordinates (ti, tj) and the additional indirection naturally incurs some overhead. However, regarding set- or block-based algorithmic patterns, whole blocks are queried rather than single matrix elements, and many index lookups can be avoided. An example of such a pattern is the tiled matrix multiplication operator, which will be explained in greater detail in Section 5.3. Since the tile grid is row-major ordered, the operation control flow can be shifted to directly work on the grid. In particular, the matrix multiplication is driven by iterating over grid coordinates (ti, tj), and works on a complete tile-level rather than on the single element-level. In addition, the AT Matrix contains a hash table that maps each tile reference to its absolute position in matrix $\boldsymbol{A}$. This is necessary, since tile instances themselves do not carry any metadata information, i.e., they are "unaware" of their membership in the AT Matrix.

Accessing Rows and Columns

Many algorithms work either on complete matrix rows and/or matrix columns. As a consequence, data structures are usually chosen such that they offer a good access performance of either one or the other, by indexing either the rows (CSR) or the columns (CSC). However, there are also use cases where access in both directions is desired. For example, imagine an adjacency matrix of a directed graph, where one is interested in both outgoing (row access) and incoming edges (column access) of a certain vertex. In order to fetch a single column from the CSR structure, a full scan of all elements is required, and vice versa. A naive workaround for this problem is to keep two copies of the matrix, one in a CSR-, and the other in a CSC representation. Obviously, this is an undesired solution for very large matrices that occupy a significant fraction of the system's memory. Moreover, the manipulation of a matrix in such a setup creates a considerable synchronization overhead, since both matrices have to be maintained in parallel.

In contrast, our approach foresees the installation of a single AT Matrix instance for a matrix. Unlike a plain CSR data representation, the tiled substructure of our AT Matrix already lowers the penalty of column reads significantly. This is because only the affected tiles that are in the queried

column range have to be scanned. Figure 5.4 shows a small study on the single-core read performance by using the AT Matrix data structure vs. a plain CSR data structure for row- (Fig. 5.4a) and column reads (Fig. 5.4b). In this experiment, 2000 randomly selected matrix rows and columns were sequentially read from different matrices of Table A.1, using an Intel Xeon X5650 CPU with 48 GB RAM as platform. The matrix dimensions range from 17,040 (R1) to 45,101 (R4). In the left plot of Figure A.1, we observe that the overhead of the additional indirection, which is involved in the reads of AT Matrix-rows, results in a runtime that is at most doubled compared to CSR. However, for column accesses, the read performance on AT Matrix is up to 25x faster. As a consequence, the AT Matrix can be considered as much more robust against irregular access patterns.

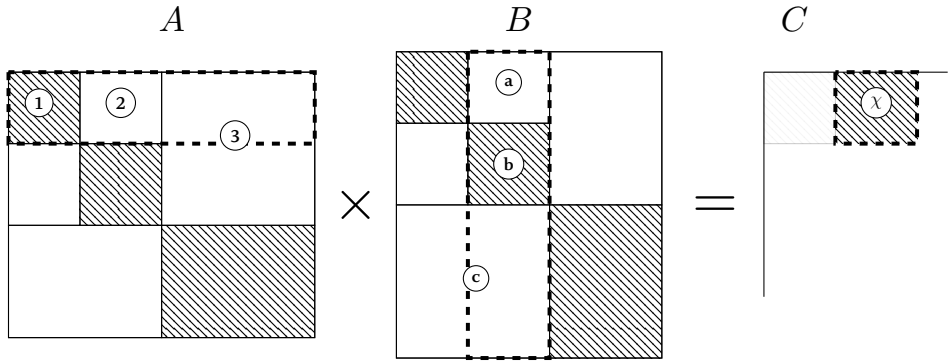
Note that unlike the random reads of single rows/columns, many operations such as blocked matrix multiplication are tile-based. Hence, tiles are read as a whole, and the incurred overhead by the additional indirection is significantly lower. Furthermore, the overhead is easily overcompensated by the advantages of the tiled, heterogeneous AT Matrix substructure, as can be seen in the evaluation in Section 5.5. Nonetheless, although we used CSR as sparse tile representation, the concept of the AT Matrix is orthogonal to the physical tile representations. As mentioned in Chapter 3, many algorithms can be written in a way that they are solely based on either of both representations. For example, the sparse accumulator algorithm sketched in Algorithm 10 is CSR-based, but there is an equivalent CSC-version (Gilbert et al., 1992). If for some reason, the most frequent access pattern are random reads on columns, then CSC can be used instead of CSR to increase the efficiency. For instance, an internal tile optimizer might select the layout based on the user's priority, and the usage statistics of a matrix.

5.3 MATRIX MULTIPLICATION

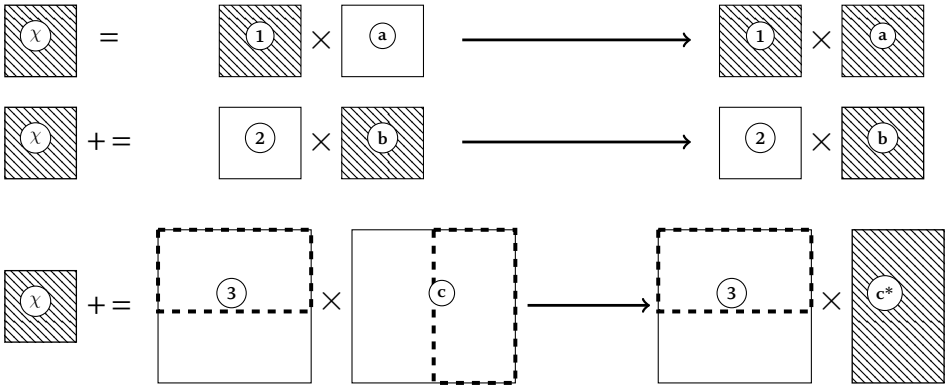
In this section we describe ATMult, our matrix multiplication operator for general matrices. ATMult calculates the result of the multiplication $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ of the two matrices $\boldsymbol{A}$ and $\boldsymbol{B}$. The matrix data type of each of $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$ can be one of the following: a plain matrix structure such as dense arrays or sparse CSR matrices, or a heterogeneous AT Matrix. The advantage of our tiled matrix multiplication is the performance improvement by exploiting cost-based optimization of tile-multiplications, and just-in-time transformations.

Algorithm 15 Sequential version of ATMult

```
1:  function atMult(AT Matrices A, B, C)
2:      ρ̂_C ← estimateDensity(ρ̂_A, ρ̂_B)
3:      ρ_D^W ← min{ρ_0^W, waterLvlMethod(ρ̂_C, MEMLIMIT)}
4:      for all tile-rows ti in A do
5:          for all tile-cols tj in B do
6:              C_{ti,tj} ← ((ρ̂_C)_{ti,tj} ≥ ρ_D^W ? DenseTile : SparseTile)
7:              for all matching tiles k in A_{ti} and B_{tj} do
8:                  w ← calculateRefWindow(A_{ti,k}, B_{k,tj})
9:                  A'_{ti,k}, B'_{k,tj} ← optimize(A_{ti,k}, B_{k,tj}, C_{ti,tj})
10:                 TileMultiply(A'_{ti,k}, B'_{k,tj}, C_{ti,tj}, w)
```



(a) Referenced submatrix multiplication.



(b) Cost-based optimization of tile multiplications using dynamic tile conversions.

Figure 5.5: Adaptive tile multiplication

The pseudocode for ATMULT is shown in Algorithm 15. For illustrational purposes, we first sketch the *sequential* version of ATMULT, and address the parallelization later separately. In general, multiplications that involve one or more AT MATRICES are internally processed as multiple, mixed multiplications of either *dense* or *sparse* matrix tiles in a block-wise manner, as depicted in Figure 5.5a.

Each matrix tile multiplication is then processed by the multiplication kernel algorithm that matches the storage types of the left- and right-hand side matrix tiles. However, unlike naive block-wise matrix multiplication, one particularity of the ATMULT operator is that matrix tiles may have different block sizes. This entails that some multiplications involve only the matching subpart of a matrix tile, since a matrix multiplication is only defined for matrices with a matching contraction (inner) dimension k. Figure 5.5a sketches a situation, where the matrix tile $\boldsymbol{C}_{(\chi)}$ results from the accumulation of three tile-multiplications $\boldsymbol{A}_1\boldsymbol{B}_a$, $\boldsymbol{A}_2\boldsymbol{B}_b$ and $\boldsymbol{A}_3\boldsymbol{B}_c$. However, only the upper half of tile $\boldsymbol{A}_{(3)}$ is multiplied with the right half of tile $\boldsymbol{B}_{(c)}$. We denote this operation as *referenced submatrix multiplication*.

5.3.1 Multiplication Kernels

The focus of this chapter is the optimal processing of tiled matrix multiplication rather than the particularities of low-level algorithms of the multiplication kernels. We already introduced some of the multiplication kernels in Section 3.6. Instead, our ATMULT approach acts in the layer above the kernels, which are merely invoked in the form of multiplication tasks on pairs of matrix tiles (line 10 of Algorithm 15). In other words, the tile optimization is decoupled from the implementations, given that the cost functions of all kernels are known to the optimizer. Note that our AT MATRIX architecture uses the column-oriented storage layer-integrated matrix representations that were shown in Section 3.4. Consequently, sparse matrix tiles are represented in the popular and widely used CSR layout. Hence, our approach benefits from a broad variety of libraries, which provide efficient implementations for the corresponding multiplication kernels. Since sparse and dense matrix multiplication kernels are continuously improved, and tuned for modern hardware (Matam et al., 2012; Patwary et al., 2015; Buluç and Gilbert, 2012; Vuduc and Moon, 2005; Schubert et al., 2011), it is beneficial to have the option to use existing high performance libraries as multiplication kernels wherever possible. As a consequence, new algorithms from the high performance community that are based on the same basic data structures could just be plugged into our system.

Nevertheless, for some of the eight different multiplication kernels of mixed dense/sparse matrices, many high performance libraries such as Intel MKL offer no reference implementation. As described in Section 3.6, we therefore implemented our own shared-memory parallel implementations based on the sparse accumulator approach. In accordance to the BLAS standard, all multiplication kernels are accumulative, i.e. $\boldsymbol{C}' = \boldsymbol{C} + \boldsymbol{AB}$. Hence, if the result matrix $\boldsymbol{C}$ is non-zero prior to the multiplication, all elements resulting from the multiplication are accumulated into the target matrix tile.

5.3.2 Referenced Submatrix Multiplications

The ability of multiplying only certain subparts of matrix tiles is a basic building block of ATMULT. Arbitrary rectangular subparts can be referenced via four coordinates, namely the coordinates of the *upper left* edge (x_{ul},y_{ul}), and the coordinates of the *lower right* edge (x_{lr},y_{lr}), which are both relative to the upper left end of the matrix tile $(0, 0)$. In Algorithm 15 the referenced coordinates are encoded in the reference window w.

Fortunately, the implementation of referencing in the multiplication kernels is less intrusive than one might expect, in particular for dense matrix multiplications: the BLAS GEMM interface already distinguishes between the embracing array size that could differ from the actual matrix dimensions, by providing the additional parameters *lda*, *ldb*, and *ldc* (leading array dimension in $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$). So the only adaption we made is transforming the submatrix coordinates into an offset that points to the beginning of the matrix in the array, and passing the respective *lda* and *ldb* to the DDD_GEMM kernel.

Regarding sparse kernels, referencing subranges in the *row* dimension is equally trivial, since matrix rows in the CSR representation are indexed. In contrast, matrix elements are not index-addressable in the column direction, which complicates the adaption of our sparse multiplication kernels to process column ranges. We altered the row-based algorithm to process only the column indices that are contained in the referenced row range. To avoid undesired column range scans, the

elements in each matrix row are sorted by their column coordinate at creation time to enable binary search for column entries. Moreover, the number of non-zero elements in a sparse tile of size $\tau_{\text{max}}^{\text{sp}}$ is restricted (Eq. 5.2), which limits the number of elements in a column range. In fact, we observed in our experiments that the overall performance improvement due to dynamic multiplication optimization overcompensates the remaining overhead due to referenced submatrix multiplications.

5.3.3 Dynamic Multiplication Optimizer

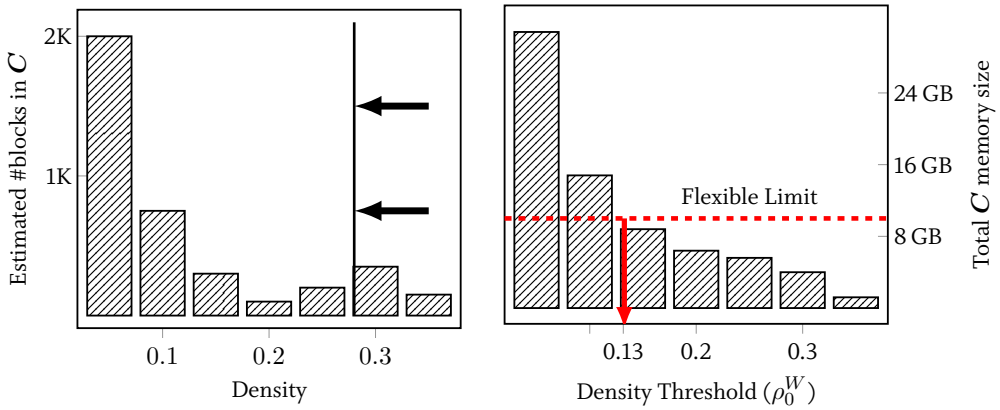
As described in Section 4.3, each of the tile multiplication kernel functions has a different runtime behavior that is reflected by a comprehensive cost model. The runtime of each kernel depends on the dimensions $m \times k$ of matrix $\boldsymbol{A}$, $k \times n$ of matrix $\boldsymbol{B}$, the densities ρ_A, ρ_B, ρ_C of both input matrices, as well as the *estimated* result density $\hat{\rho}_C$. A common situation is that a matrix tile, or even a subpart of a matrix tile, exceeds a characteristic density threshold of the cost model. Then it might be beneficial to convert the respective part into a dense representation prior to the multiplication operation. Hence, the optimizer dynamically converts the input matrix representations whenever it leads to a reduction of the multiplication runtime for the respective part multiplication (line 9 in Algorithm 15).

Although the tile multiplication optimization seems similar to that of the matrix chain optimizer (SpMachO) of Chapter 4, the difference is substantial. In particular, the target tile density is not solely defined by the estimation of the two input tiles. In contrast to a single multiplication operation, the target tile $\boldsymbol{C}_{(\chi)}$ of Figure 5.5a is written accumulatively in multiple tile-multiplications (i.e., in each multiplication of the corresponding block-row $\boldsymbol{A}_1\boldsymbol{B}_a$, $\boldsymbol{A}_2\boldsymbol{B}_b$ and $\boldsymbol{A}_3\boldsymbol{B}_c$). Hence, the physical representation of a $\boldsymbol{C}_{(\chi)}$ tile is selected according to its *final* density, which is taken from the density estimate $\hat{\rho}_{C(\chi)}$.

There is a significant discrepancy in the cost model regarding the density turnaround point between the matrix $\boldsymbol{A}$, $\boldsymbol{B}$-tiles that are *read*, and the $\boldsymbol{C}$-tiles that are *written* as part of the multiplication. In general, writing to a sparse tile is much more expensive than reading from it, compared to a dense tile, which has a smaller read/write cost asymmetry. This is why we introduce two separate density thresholds: one for tiles that are read ρ_0^R, and one for tiles that are written ρ_0^W. The latter has usually a lower value than the former, which also explains the good results of the SPSPD_GEMM kernel in our evaluation compared to SPSPSP_GEMM. However, storing a rather sparse result matrix in a dense array is an excessive waste of memory that should be avoided, even when the cost-based optimization might decide so. In a resource-managed system, such as a DBMS, there are usually some operational service level agreements to meet, for example a restriction of the total memory consumption. Therefore, our ATMult approach employs a flexible *write* density threshold ρ_D^W depending on the desired memory space limit of the target matrix (line 3). It should be noted that the adaption to runtime-available resources might sacrifice performance in favor of a lower memory consumption.

5.3.4 Density Estimation

There are several reasons for having a prior estimate of the block-density structure of the result matrix, and they are barely different from the motivation of cardinality estimation for relational join processing: first, the optimizer's decision naturally depends on the output density, which is



(a) One-dimensional histogram of logical block densities.

(b) The memory consumption depending on the density threshold.

Figure 5.6: Water-level method: the water-level (blue line, Figure 5.6a) scans the logical blocks and approaches the intersection. The scan is stopped when the intersection of the total memory consumption with the *flexible limit* (dashed line, Figure 5.6b) is reached.

an influential parameter in the cost model of sparse multiplication kernels. Second, knowing the result density is crucial for memory resource management, which includes memory restrictions that also affect the choice of data representations in the optimizer, e.g. in order to avoid out-of-memory situations. However, the exact non-zero structure can only be found through the actual execution of the multiplication. Therefore, we make use of the density estimation procedure based on probability propagation SpProdest, which was introduced in Section 4.4. In particular, ATMult uses the "density map" estimator to obtain an estimate at negligible cost compared to the actual multiplication time. The estimator takes the density maps of the input matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ and returns a density map estimate of the resulting matrix $\boldsymbol{C}$ (e.g. in Figure 5.1c).

5.3.5 Memory Resource Flexibility

In order to determine the dynamic, matrix-wide write density threshold ρ_D^W for *target* tiles with respect to the flexible memory consumption limit, we employ a "water level method" that uses the density map estimate $\hat{\boldsymbol{\rho}}_C$ (line 3 in Algorithm 15): consider $\hat{\boldsymbol{\rho}}_C$ as a two-dimensional histogram with a density bar ρ_{ij} for each logical matrix $(b_{\text{atomic}} \times b_{\text{atomic}})$ block (ij) – the higher the density ρ_{ij} of block (ij), the higher its bar in the histogram. The idea is to start with a high water level that covers all bars in the two-dimensional histogram. If the level is lowered, the blocks with the highest density bars will be visible first – hence, storing these blocks as dense is most promising with regard to the potential performance improvement. The level will be continuously lowered until the accumulated memory consumption of all dense and sparse blocks hits the memory limit. The method can easily be transformed from two-dimensional space into a one-dimensional space, as shown in Figure 5.6: instead of a 2D-histogram, we create a 1D histogram that displays the absolute number of logical blocks per density bin ρ_j. Lowering the water level can be imagined as a sliding vertical line (Fig. 5.6 *left*) from the right to the left histogram side. All blocks right of the line contribute with dense size

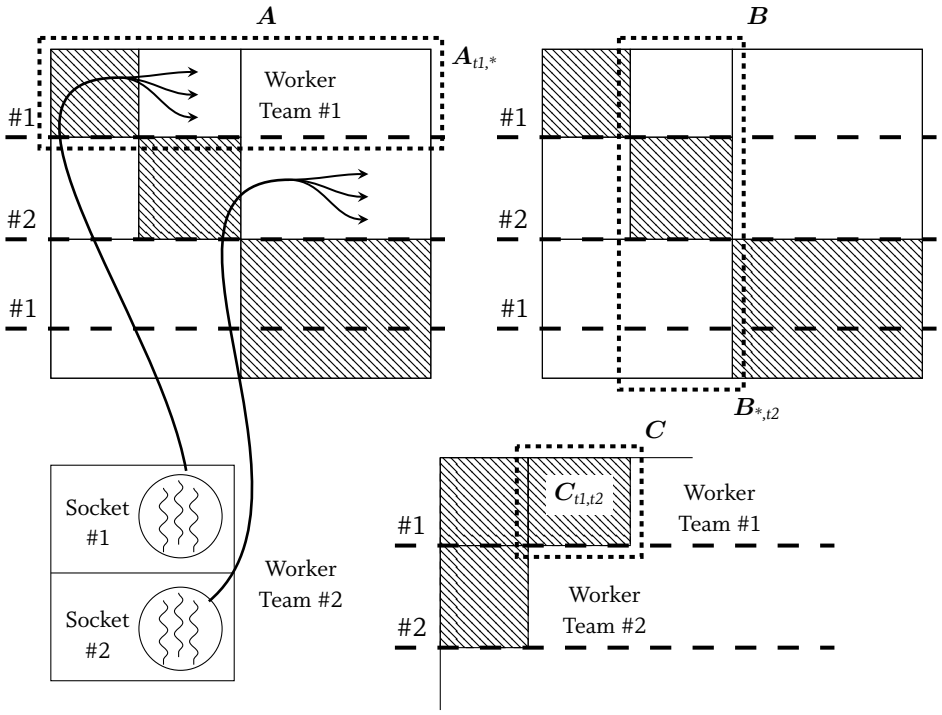


Figure 5.7: Parallel resource distribution and NUMA partitioning for a ATMULT operation on a two-socket system.

$B\mathcal{S}_d$ to the memory consumption, whereas all blocks on the left-hand side only with the size of the sparse representation $\rho_i \cdot B\mathcal{S}_{sp}$. The resulting density threshold equals the intersection of the accumulated histogram (Fig. 5.6 *right*) with the flexible memory limit.

5.4 PARALLELIZATION

Due to the tiled structure in the AT MATRIX representation, a characteristic of the ATMULT multiplication is that there are *two* levels of parallelization: the *inter-tile* parallelization (number of worker teams) and the *intra-tile* parallelization (number of threads in a team), as illustrated in Figure 5.7.

Inter-tile parallelization. Firstly, pairs (*ti*,*tj*) of tile-rows *ti* of matrix $\boldsymbol{A}$ and tile-columns *tj* of matrix $\boldsymbol{B}$ are formed. For instance, in Figure 5.7 the pair ($\boldsymbol{A}_{t1,*}$,$\boldsymbol{B}_{*,t2}$) is marked. Each pair represents a set of tile-multiplication tasks that write a target tile $\boldsymbol{C}_{ti,tj}$ in the result matrix. Multiple worker teams are running in parallel on different pairs, e.g. team #1 and #2 in Figure 5.7. All tile-multiplications referring to a particular *tile-row-column pair* (*ti*,*tj*), i.e., all multiplications that write into the same target tile $\boldsymbol{C}_{ti,tj}$, are executed one after another, and by the same worker team.

Intra-tile parallelization. All multiplication kernels used for tile-multiplications are internally parallelized on shared memory. As described in Chapter 3, the internal parallelization of the sparse,

and mixed sparse-dense multiplication kernel implementations is rather straightforward. Most of them contain outer loops that can be batched in row-blocks, and separated into multiple disjoint work units. Write conflicts could only occur if the target tile is in a CSR representation, and are completely avoided by utilizing a local CSR buffer for each worker thread, which will be merged before return.

The parallel resources should be distributed among these two levels, which is generally nontrivial: assigning all threads to the intra-tile level might lead to over-parallelization, when blocks are small and sparse. In contrast, leaving all threads to the coarse-grained inter-tile level might have a negative influence on the last level cache re-usage. In fact, the cache is "polluted" if many disjoint tiles are touched concurrently. The LLC is separate for each memory socket in the system. Hence, to avoid cache pollution we spawn only as many worker teams as there are sockets on the system, so that each worker team writes to its local memory node. The number of threads per team is related to the number of cores on the socket.

5.4.1 Scheduling Strategies

There are different ways to implement the two-leveled parallelism described above. We present two competing strategies, and explain why we prefer one over the other. It is worthwhile mentioning that it is common to differentiate between *data-parallelism*, which mostly refers to partitioned data chunks that are processed in parallel, and *task-parallelism*, which refers to different tasks that are run concurrently.

Nested Parallel For

A very common parallelization strategy in high performance environments is using *parallel for* constructs, also called ParFOR (Boehm et al., 2014b). ParFOR constructs are included in many parallelization frameworks, e.g., in OpenMP (Dagum and Menon, 1998).

In particular, parallel for-loops can be nested to achieve multi-level parallelism. For the parallelization of ATMULT as of Algorithm 15, we parallelized the for-loop over the tile-rows (line 4). The kernel algorithms for the tile-multiplications internally may also be parallelized using ParFOR loops (Section 3.6), thus their parallelism is *nested* in the outer tile-row loop. Each ParFOR loop is assigned a configurable number of worker threads, which can be used to balance the resources between the inter-tile and the nested intra-tile loop. For the implementation we used OpenMP which supports nested parallelism. Note that we use the Intel MKL routine for DDD_GEMM, which however, is also based on OpenMP internally[3] and provides an API to configure the number of threads.

The drawback of this approach is that it requires a careful distribution of parallel resources balancing the number of threads between the intra-tile and inter-tile level. The resource assignment is rather static, such that if n_{intra} threads are assigned to the nested level and n_{inter} to the outer level, the number $n_{\text{intra}} \times n_{\text{inter}}$ should match the total number of available cores. Moreover, the n_{intra} OpenMP threads of an inner nested group are not able to steal work of another nested group. In addition, each of the inner-level thread groups is synchronized using barriers, which can result in an underutilization of CPU resources.

[3] In fact, Intel has its own implementation of the OpenMP runtime (`https://www.openmprtl.org/`).

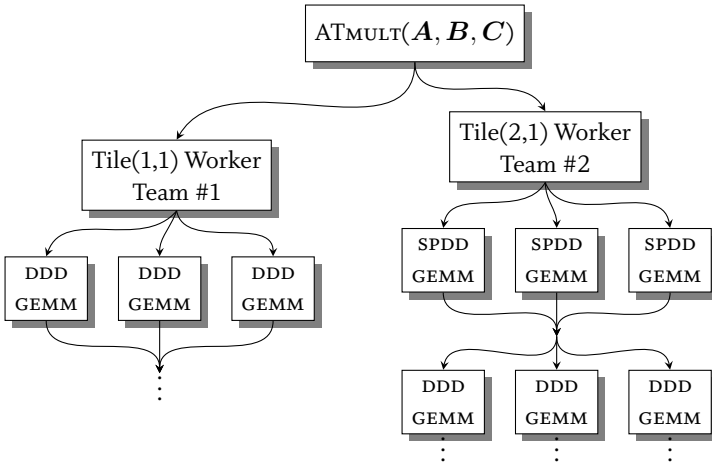


Figure 5.8: Upper part of the task DAG for the ATMULT multiplication using the example of Figure 5.7.

Task-Based Scheduling

To overcome the shortcomings of the nested ParFOR strategy, we have chosen a task-based approach. That is, each thread of the thread-pool[4] pops a task from a task priority queue, processes it until it is finished, and pops the next task. Moreover, there can be dependencies between tasks that define the order of execution.

In particular, the ATMULT routine constructs a task direct acyclic graph (DAG) as shown in Figure 5.8. The DAG contains dependencies between the tasks, e.g. tile multiplications that write into the same target tile as a prior tile multiplication wait until the prior tasks finish. Note that each tile multiplication, e.g. SPDD_GEMM in Figure 5.8, is further split into multiple tasks according to the row-block partitioning scheme of Figure 3.8b. Finally, the DAG is executed by a threads-pool while conforming to the given constraints. For instance, these include that some tasks might be pinned to be executed on a specific CPU socket.

The advantage of the task scheduling strategy is that it avoids idle cores by balancing the independent, fine-granular tasks across all available threads. Hence, it is more flexible and exhibits a better CPU utilization in contrast to the static, nested ParFOR construction, whose thread groups are fixed, and cannot take over the work of the other nested thread groups.

Meanwhile, there are several frameworks that offer basic task scheduling, including the recent version of OpenMP. We decided in favor of the job scheduling framework that is used in the SAP HANA database (Psaroudakis et al., 2015), which contains socket-local task queues and task-stealing across multiple task queues. Moreover, it offers the option to pin tasks to CPU cores and NUMA nodes. Compared with our initial nested parallel for-implementation using OpenMP, we observed a significantly better resource utilization by using a task-scheduling strategy.

[4]The thread-pool comprises all available threads in the system. The number of available threads is commonly in the order of the number of the physical cores plus hyper-threading.

Resource-Sharing with OpenMP

Despite using a custom task scheduling framework, the OpenMP runtime is still invoked by methods that we call from external libraries, but are not under our control, e.g. the DDD_GEMM routine of Intel MKL. Obviously, the creation of two separate thread pools where each is agnostic of the other is undesirable and should be avoided. Therefore, we plan to integrate recent work that provides an OpenMP runtime that internally uses a task scheduling framework instead of system threads directly (Wolf et al., 2015). Thus, all parallel resources are managed by a central scheduler using a single thread pool, and any overutilization of system resources in terms of excessive thread creation is avoided – which is essential for operators that run on resource managed systems, like a commercial DBMS.

5.4.2 NUMA-aware Partitioning

On modern multi-core machines, main memory is physically distributed across several sockets that each contain multiple cores. The consequence of this hardware design is that remote accesses from one CPU to another socket are slower than accesses to the socket's local memory. This effect is known as non-uniform memory access (NUMA)[5]. To reduce remote accesses, our matrices are distributed across the memory nodes as part of the partitioning process.

It is worthwhile mentioning that for the matrix multiplication $\boldsymbol{A} \cdot \boldsymbol{B}$ each row slice of $\boldsymbol{A}$ must be multiplied with all of matrix $\boldsymbol{B}$. For perfect data-local processing it would be beneficial to replicate matrix $\boldsymbol{B}$ to each memory socket. However, replication to each socket in the system is not a scalable approach regarding very large matrices having sizes of hundreds of gigabytes, and the perspective of increasing numbers of sockets in a system. Hence, we pursue a straightforward range-partitioning strategy: since it is generally unknown whether a matrix will take part as the left or the right operand in a matrix multiplication, all matrices are in the same way horizontally partitioned. Consequently, $\boldsymbol{A}_{ti,*}$ and $\boldsymbol{B}_{ti,*}$ tile-rows are distributed round-robin-wise across the sockets.

We implemented the partitioning on Linux using the *libnuma* library (Kleen, 2005). As mentioned before, the task scheduling framework (Psaroudakis et al., 2015) enables to bind the resources that execute a task to a particular socket. The worker thread teams of ATMULT are pinned to the socket where the respective $\boldsymbol{A}_{ti}$ tile-row is located, and thus, all $\boldsymbol{A}$-tiles are local in the computation node. Moreover, the target tiles in $\boldsymbol{C}_{ti,tj}$ are dynamically allocated by a thread of the respective worker team, and also initialized if their representation is dense. Due to the first touch policy (Lameter, 2013) of Linux, all newly allocated and initialized memory pages are socket-local. As a result, the target tiles are placed on the memory node where the multiplication takes place, and at most the tiles of $\boldsymbol{B}$ are remotely accessed. Eventually, $\boldsymbol{C}$ effectively inherits the tile-row memory distribution scheme from matrix $\boldsymbol{A}$.

5.5 EVALUATION

In this section we present measurements of the partitioning routine and the ATMULT matrix multiplication runtimes using a variety of real-world matrices. Moreover, we add synthetic matrices

[5]Note that with NUMA we actually mean *cache coherent* non-uniform memory access, also known as ccNUMA

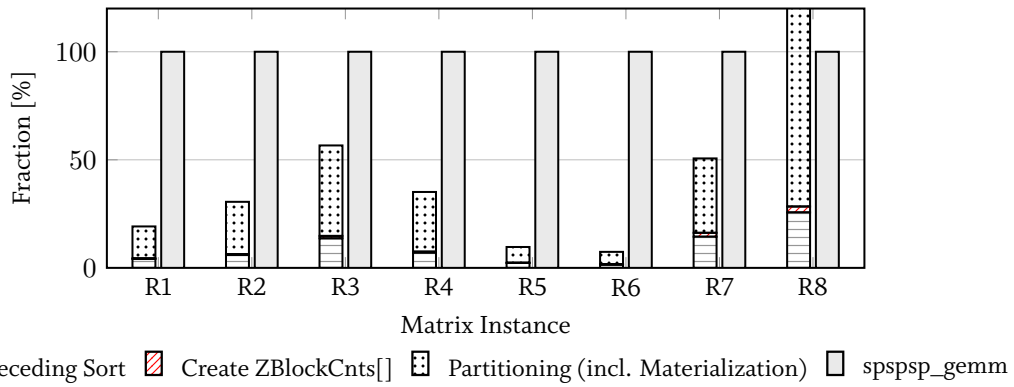


Figure 5.9: Duration of the components in the partitioning process of AT MATRIX, relative to a sparse matrix multiplication (executed once).

using the RMAT graph generator (Chakrabarti et al., 2004) to systematically evaluate the influence of data skew.

5.5.1 Experiment Setup

We used a four-socket Intel E7-4870 system with 4×10 cores (80 physical threads via hyper-threading) @2.40 GHz and a total of 1 TB RAM. Regarding the tunable parameters, our system configuration yields $b_{\text{atomic}} = 1024$ ($k = 10$) for the minimum tile size and $\rho_0^R = 0.25$ as tile read density threshold.

Table A.1 lists the sparse matrices that we used in the experiments. All real-world matrices (Ri) except the Hamiltonian matrices are taken from the Florida Sparse Matrix Collection(Davis and Hu, 2011), which contains matrices of various domains, such as power networks, genomics or structural problems. The Hamiltonian matrices were provided by a nuclear physics group we cooperated with. In general, we tried to select matrices of different shapes and sizes in order to compare our approach on a wide scale of data. However, to understand the implications of matrix size, density, and data skew on our algorithm better, we added synthetic matrices (Gi). These were generated using a recursive RMAT graph generator (Chakrabarti et al., 2004), and are listed in the lower half of Table A.1. The RMAT-generated matrices are configurable via the parameters: dimension, number of non-zero elements, and the four values a, b, c, d. The latter control the relative fractions of non-zero elements that are contained in the upper left, upper right, lower left and lower right part of the submatrix at each recursion step.

5.5.2 Partitioning Costs

Figure 5.9 shows the relative duration of the following components of the partitioning process: the preceding sort to create the Z-order, the creation of the *ZBlockCnts* array, and the recursive partitioning routine itself, including the tile materializations. The partitioning time is dominated by the materialization, which includes copying and reordering into CSR, or row-major array representation. Except for matrix R8, the duration of the partitioning process is smaller than a single

execution of the traditional *sparse* × *sparse* → *sparse* multiplication algorithm (SPSPSP_GEMM). Matrix R8 matches a case we mentioned in section 5.2.3, where the non-zero size of the output matrix is relatively small, but the dimensions are large.

5.5.3 Performance Comparison

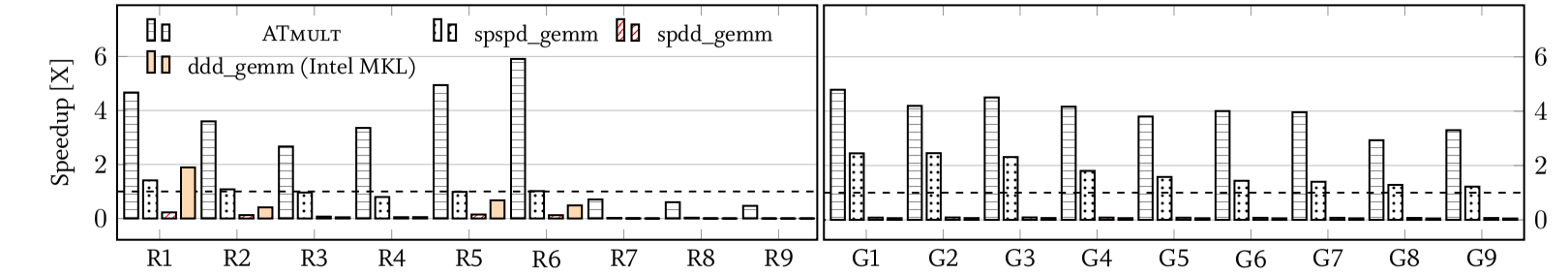
Figure 5.10 shows the matrix multiplication performance of our ATMULT approach relative to the SPSPSP_GEMM multiplication kernel. We chose it as baseline ($\equiv$ 1) because it is similar to the algorithm used in R or MATLAB, which however, only have a sequential sparse matrix multiplication implementation. In addition, we included the SPSPD-, SPDD-, and DDD_GEMM (Intel MKL) kernels in the measurement.

We observe that ATMULT outperforms the alternative approaches in most cases. Only for matrices R7-R9 the ATMULT performance is slightly behind the SPSPSP_GEMM performance. This can be explained by the matrix characteristics: matrices R7-R9 do not contain any region of a higher density, hence, offering little optimization potential. As a consequence, the partitioning overhead just adds to the ATMULT execution runtime. Obviously, the other approaches that involve plain dense matrix representations (SPSPD-, SPDD- & DDD_GEMM) have an even worse performance for R7-R9. Nevertheless, for most of the other instances, the SPSPD_GEMM kernel (dense target array) seems to be the better alternative to SPSPSP_GEMM, since the result matrix is often significantly denser than the input matrices. The advantage of ATMULT over the naive algorithms is most clear when there are distinct regions of a significantly higher local density in the matrix non-zero pattern, as for example for matrix R3, which is illustrated in Figure 5.1. But even for matrices that have a dense result and a relatively uniform non-zero pattern (R4, G1, G2), we observe that ATMULT can outperform the others, particularly the SPSPD_GEMM kernel. This is an indication that part of the performance gain results from an increased cache locality due to the tiling caused by the block size limit of Equation 5.2.

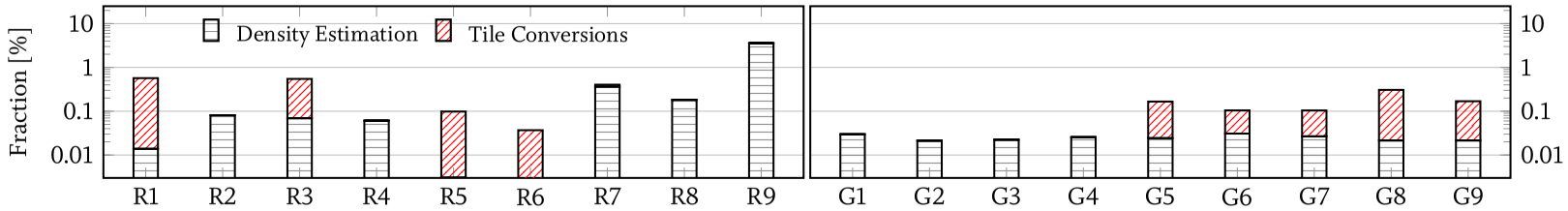
Regarding the memory consumption depicted in Figure 5.10c, the AT MATRIX result matrix ($\boldsymbol{C}$) created by ATMULT tends to have an equal or lower size than the output size of alternative approaches, sometimes even lower than the sparse CSR format (matrices R1, R3, R5, and R6). This case occurs when there are dense regions $\rho > (\mathcal{S}_d/\mathcal{S}_{sp})$ that are stored more efficiently in a dense tile than in a sparse representation.

To confirm our implications in a more systematic manner, we generated large RMAT matrices G1-G9 with increasing skew in the matrix non-zero pattern. For equal parameters $a \equiv b \equiv c \equiv d$ the non-zero element distribution is nearly uniform. In contrast, the higher the parameter a, the more non-zero elements are placed in the upper left quarter of each recursion step, and hence, the higher is the skew of the matrix. All generated matrices with their parameters are also listed in Table A.1.

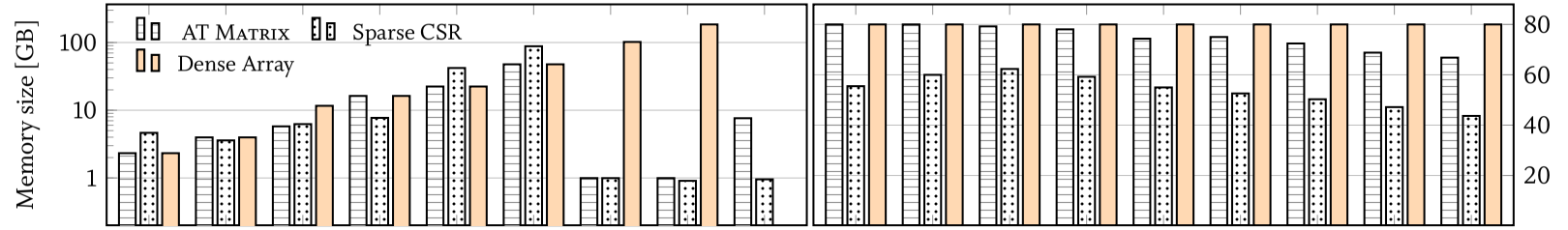
The right-hand side of Figure 5.10a illustrates how the skew affects the runtimes of the sparse matrix-sparse matrix multiplication. We observe that ATMULT outperforms SPSPSP_GEMM by a factor of 3-5x, and SPSPD_GEMM by a factor of about 2-2.5x. With increasing skew, the internal execution of ATMULT changes, which can be inferred from the increased optimization efforts in Figure 5.10b. Interestingly, the speedup over SPSPSP_GEMM is slightly decreasing due to the reduced runtime of the latter, which is correlated to the sparse output size (see Figure 5.10c). The skew reduces the number of non-zero elements in the multiplication result matrix, as more of the element



(a) Sparse-sparse matrix multiplication performance. The speedup is given relative to the SPSPSP_GEMM runtime.



(b) Density estimation and optimization (*including* tile conversions) as part of the absolute ATMULT runtime.



(c) Memory consumption of $\boldsymbol{C}$ in the AT MATRIX, *sparse* (CSR) and *dense* representation.

Figure 5.10: Sparse $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B}$ multiplication experiments where $\boldsymbol{A} = \boldsymbol{B}$, using real-world and synthetic matrices (see Table A.1 for details about matrices).

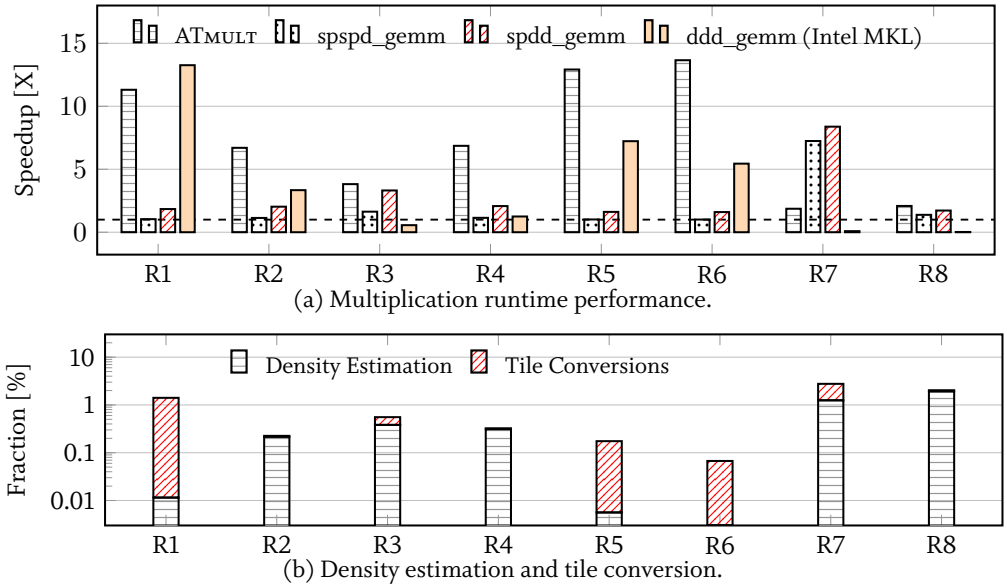


Figure 5.11: Multiplication of a *sparse* matrix $\boldsymbol{A}$ (Rn) with a *full* matrix $\boldsymbol{B}$: speedup of ATMULT and optimization time breakdown.

multiplications $(A)_{ik}(B)_{kj}$ fall into the same target coordinates (ij). In contrast to the naive SPSPD_GEMM approach, the memory size of the ATMULT result reflects this trend, as internally more result tiles will be stored sparse instead of dense.

Besides sparse-sparse matrix multiplication, which is for example used in graph algorithms, many applications involve the multiplication of a sparse with a full, dense ($\rho = 1.0$) matrix. Therefore, we included mixed sparse-dense matrix multiplication in the experiments. In particular, we distinguish between the case where $\boldsymbol{A}$ is sparse and $\boldsymbol{B}$ is dense (Fig. 5.11), and the case where $\boldsymbol{A}$ is dense and $\boldsymbol{B}$ is sparse (Fig. 5.12). The dense matrices are formed such that the number of elements $m \cdot k$ is in the same order of magnitude as the number of sparse matrix elements N_{nz}. Hence, the dense matrix is rectangular, and the independent dimension is calculated as $n = \gamma N^A_{\text{nz}}/k$ for b) and $m = \gamma N^B_{\text{nz}}/k$ for c), respectively (we chose $\gamma = 3$).

Similar to the results of the sparse-sparse matrix multiplication, we observe that ATMULT outperforms the alternatives for all mixed sparse-dense test instances except for the following: The relatively dense and small matrix R1 is multiplied most efficiently with DDD_GEMM (MKL). Although ATMULT also uses this kernel internally, the dynamic optimizer has to convert some initial sparse tiles into dense tiles, adding overhead to the total execution time. Second, ATMULT is outperformed by SPDD- and SPSPD_GEMM for the light, hypersparse matrix R7. Here, the overhead results from the implicit slicing of $\boldsymbol{A}$ in the multiplication, due to referenced submatrix multiplications caused by the actual partitioning of $\boldsymbol{B}$. Such situations could be avoided by a dynamic re-tiling of the left-hand matrix as a part of a pre-multiplication optimization, which, however, is left for future work.

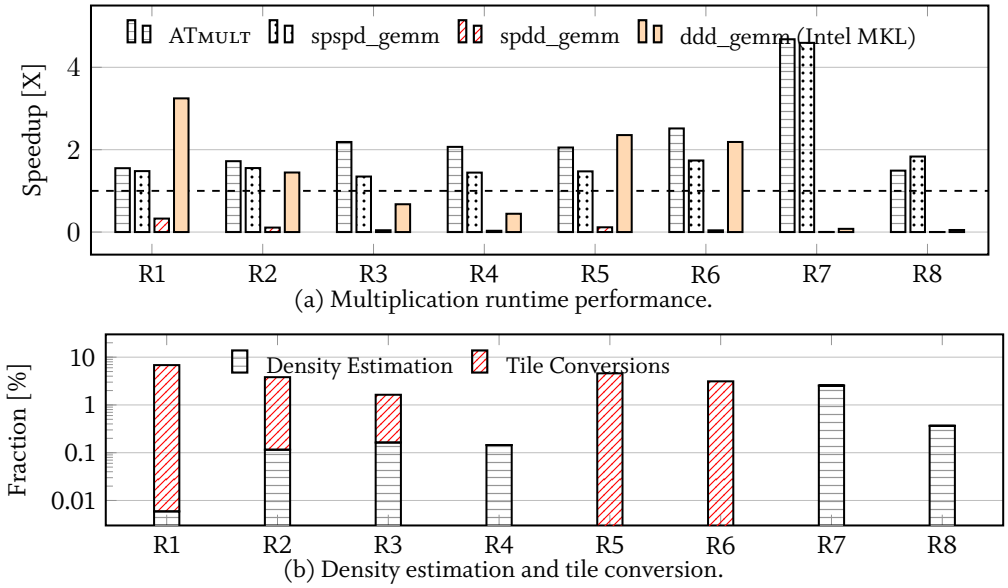


Figure 5.12: Multiplication of a *full* matrix $\boldsymbol{A}$ with a *sparse* matrix $\boldsymbol{B}$ (Rn): speedup of ATMULT and optimization time breakdown.

5.5.4 Runtime Optimizations

Finally, we evaluate how the total runtime of a ATMULT multiplication operation breaks down into its processing steps: the density estimation, the dynamic optimization including tile conversions, and the multiplication runtime itself. In fact, for all instances the execution time of ATMULT is dominated by the runtime of the actual tile multiplications, which is why we only show the relative fraction of ATMULT time that is spent on density estimation and optimization (tile conversion) in Figures 5.10b, 5.11b, and 5.12b. First of all, we observe from the right-hand side of Figure 5.10b that the dynamic optimization time almost vanishes for nearly uniform matrices, but grows with increasing skew. It reaches a peak of about 7.5% of the total runtime for the dense-sparse multiplication with matrix R1 in Figure 5.12b. This peak can be explained by the topology of the AT MATRIX R1: many tiles have a density slightly below the read density threshold, thus, they are stored in the sparse representation. In fact, the optimizer then may decide to convert these tiles into a dense representation prior to the tile multiplication, adding conversion time to the optimizer runtime. Nevertheless, in the vast majority of cases the overall ATMULT time including the conversion is still lower compared to any alternative multiplication approach that uses plain matrix representations.

The part of the density estimation is for most instances with less than 0.1% of ATMULT runtime negligible. As a matter of fact, the runtime of the density estimation procedure is independent from the number of non-zero elements, but depends on both matrix dimensions and the logical block size, which together define the size of density grid. As a consequence, it becomes more significant (5% for matrix R9) for hypersparse matrices with very high dimensions.

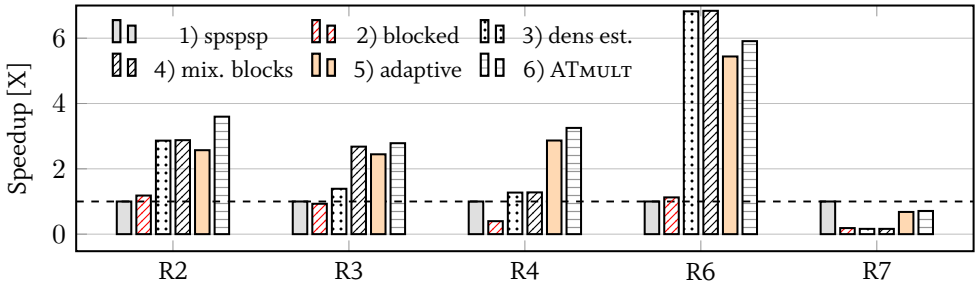


Figure 5.13: Relative multiplication performance, where $\boldsymbol{A}$ and $\boldsymbol{B}$ are sparse. Impact of the different optimization steps of ATMULT on the performance.

5.5.5 Impact of Single Optimization Steps

In this final experiment, we evaluated step-by-step the influence of adding different optimization components, and how each of them impacts the multiplication runtime performance. Starting from the regular SPSPSP_GEMM multiplication, we incrementally add the optimization steps that are contained in our ATMULT approach. They can be separated as follows:

1. Baseline: SPSPSP_GEMM on unpartitioned sparse matrices.
2. Fixed-size, sparse-only tiles: The matrix is tiled into a fixed grid of sparse tiles. The tiles of the target matrix $\boldsymbol{C}$ are also sparse.
3. Fixed-size, sparse-only tiles, and density estimation: Same as 2), but with target density estimation. Target tiles are dense if the estimated density exceeds ρ_0^W.
4. Fixed-size, mixed tiles, and density estimation: Same as 3), but the matrix blocks with a density exceeding ρ_0^R are stored as dense.
5. Adaptive, mixed tiles, and density estimation: In contrast to 4), tiles are adaptive, but without dynamic tile conversion
6. Adaptive, mixed tiles, density estimation, and dynamic tile conversion optimization (ATMULT.)

The way how the independent steps affect the performance significantly depends on the matrix topology. Therefore, we chose five different instances of the real-world matrices, and assembled the relative multiplication performance results in Figure 5.13. The first observation is that if the target tiles are sparse-only (2), then a naive fixed-size $b_{\text{atomic}} \times b_{\text{atomic}}$ blocking of the matrices results in almost no improvement. However, a significant performance boost for some matrices (R2, R6) is achievable by enabling density estimation and dense target tiles (3). In contrast to (2), dense target tiles are written more efficiently, in particular, regarding the accumulative tile writes that result from the tiled matrix multiplication. Hence, only with dense target tiles the fixed-size tiling optimization reveals its actual improvement potential. If we further add the storage type optimization (4), all fixed-size tiles of matrix $\boldsymbol{A}$ that exceed the density threshold are stored in a dense tile representation. As a consequence, we observe a considerable performance gain for matrices that have distinct dense substructures, such as R3. For other matrices, such as R6, the optimization potential is already fully exploited by the steps up to (4). In fact, for these cases the adaptive tiling (5) incurs some overhead that lowers the performance, but the drop amounts to less than 20%. In contrast, for the larger and slightly sparser matrix R4 the fixed-block approaches (2-4) bear a way higher overhead. In this case, the adaptive tile multiplications (5-6), outperform (2-4)

by 3x. Finally, using ATMULT(6) further increases the performance of (5) by allowing dynamic tile conversions. It is noteworthy that for the sparsest matrix in this comparison (R7) the runtime of (5) and ATMULT is close to the naive SPSPSP_GEMM, since, if at all, a very coarse grained tiling is chosen. In contrast, all fixed size tiling approaches (2-4) fail completely, and are orders of magnitude slower. This is why we emphasize that the fixed-size tiling approaches in related work (e.g., in Ghoting et al., 2011; Huang et al., 2013; Zadeh et al., 2015) are often suboptimal. The tiling must be chosen in accordance with the matrix characteristics, however, the details of which the user is often unaware of.

5.6 RELATED WORK

Related work about sparse matrix storage formats and tuned multiplication algorithms is extensive. Our contribution differs in the way that firstly, we target the storage and processing of large matrices in a column-oriented, main-memory database system. But more importantly, we approach the problem from a higher level, and apply optimizations inspired from database technology on top of the fundamental data structures and algorithms. As this work has overlaps with multiple research areas, we subdivide the discussion into the following major subtopics, and selected the most relevant related work papers.

5.6.1 Matrix Representations in Numerical Libraries

As outlined in the beginning, a lot of effort has been invested in implementing efficient multiplication algorithms and data structures for sparse matrices in recent decades, starting from early Fortran implementations in the 70s. We already showed in Section 3.4.2 how the triple (COO) and CSR (Gustavson, 1978) representations are seamlessly integrated on top of our column-oriented storage layer.

In fact, CSR and CSC presentations are still used in many recent works of the high performance community. For instance, Yzelman and Bisseling (2009) uses a "Zig-Zag CSR" for sparse matrix vector multiplication (SPGEMV), where the non-zero elements in adjacent rows are ordered alternatingly in ascending and descending order, which improves the cache locality in the right hand side vector. Nevertheless, besides COO and CSR, there are other data representation that – depending on the matrix topology and hardware setup – can show better performance. E.g., such data representations are the ELLPACK storage (ELL), the jagged diagonal storage (JDS or JAD), or blocked representations, such as block-CSR (BCSR). The latter stores small, fixed-size dense blocks instead of single matrix elements. Moreover, there are even further specialized matrix representations for certain matrix topologies, such as triangular, band matrices, diagonal (DIA) or almost diagonal matrices (Saad, 1994; Vuduc, 2004). Some representations, such as the JDS, require a prior permutation of the matrix that reorders the rows in decreasing order of number of non-zeros per row.

There is a lot of research activity in the field of implementing sparse matrix algorithms on co-processor platforms. E.g., Schubert et al. (2011) provide a CSR-based SPGEMV algorithm for hybrid CPU+GPU platforms. Bell and Garland (2009) implemented SPGEMV on GPUs, and showed that DIA, and hybrid ELL+COO show better performance than CSR, as they are better suited for vector

processors and GPUs. Choi et al. (2010) revisits the algorithm, and presents block-ELL, which further improves the non-blocked ELL-based version (Bell and Garland, 2009) by a factor of 1.5-1.8x, however, by the cost of an additional reordering/permutation, and repacking of rows. Kreutzer et al. (2015) propose a generalized, flexible, sparse matrix format called "SELL-C-σ", which uses a combination of partial reordering and padding to fully exploit single instruction multiple data (SIMD) vectorization. In particular, CSR ($C = 1$, $\sigma = 1$) and ELL ($C = N$, $\sigma = 1$) are special cases of their layout.

To summarize, there are a lot of sparse matrix representations that each have their justification, but are rather orthogonal to our work. In particular, most are designed for sparse matrix-vector multiplications instead of SPSPSP_GEMM, and often they are tuned for a specific system architecture. Some structures are suited for vector processors and GPUs (e.g. ELL) whereas others are for scalar processors. Vuduc (2004) observed that CSR tends to have the best performance for SPGEMV on superscalar multi-core processors for a wide class of *general purpose* matrices, which supports our decision to select CSR as the internal sparse tile representation. Note that AT MATRIX could as well use other matrix representations, since we do not make any assumption about the internal topology of a single tile. Our actual optimization approach builds on top of the low-level tile structures, and our contribution consists in the high-level separate treatment of distinct sparse and dense regions of matrices.

5.6.2 Parallel Sparse Matrix Multiplication

We use the multi-threaded sparse multiplication algorithm SPSPSP_GEMM that we described in Section 3.6. It is based on a row-block parallel version of the sequential algorithm of Gustavson (1978) which uses CSR. However, the sparse accumulator algorithm is easily transformable into a CSC-matrix based version, which is folklore and used (sequentially) in numerical libraries like CHOLMOD (Chen et al., 2008) as part of SuiteSparse (Davis, 2011), and frameworks like R or MATLAB (Gilbert et al., 1992).

There have been further, recent works on a scalable matrix multiplication for parallel processors (distributed- and shared-memory): Buluç and Gilbert (2012) present a distributed multiplication algorithm, where they store the matrix by using a grid of hypersparse, doubly compressed sparse column (DCSC) blocks. The algorithm is contained in the Combinatorial BLAS library (Buluç and Gilbert, 2011). However, it turned out to be comparatively slow on multi-core systems, as observed in a recent paper (Patwary et al., 2015). The authors of the latter present their own approach based on a partitioned Gustavson algorithm, where matrix $\boldsymbol{A}$ is divided in horizontal blocks and $\boldsymbol{B}$ in vertical blocks. Vertical partitioning of $\boldsymbol{B}$ is also used in the algorithm of Matam et al. (2012) for a hybrid CPU+GPU system.

From a theoretical perspective, Yuster and Zwick (2005) showed that an optimal algorithm can be achieved by separating the matrix into a dense part and a sparse part, and apply the respective optimal algorithm for each part. In fact, they assume the complexity of the Coppersmith-Winograd algorithm ($\mathcal{O}(n^{2.38})$) for the dense multiplication part, which is in practice not used due to the high constant costs. Nonetheless, the general idea of separating the multiplication into dense and sparse fractions remains valid, since the constants of the dense multiplication ($\mathcal{O}(n^3)$) complexity are significantly lower than those of the sparse multiplication for a full matrix.

In practice, most vendor-provided BLAS libraries like Intel MKL (MKL, n.d.), and CuSparse (CuSparse, n.d.), adhere to the original NISTb (n.d.) interface. Hence, they include only an implementation for sparse-dense matrix multiplication. Since they offer no general purpose sparse-sparse matrix multiplication, they are not directly applicable in a wide number of use cases, e.g., in the calculation of the distance matrix described in Section 2.3.1. Similar to specific matrix representations, the low-level tuning of sparse (and dense) multiplication algorithms are rather orthogonal to ATMULT. By training their cost behavior to our model they can be used internally as kernels for tile multiplications.

5.6.3 Tiled Arrays and Matrices

Lastly, we discuss the related work about tiling strategies for arrays and matrices in particular, and tiling in regard to linear algebra operations.

Array DBMS. Chunking or tiling of arrays is a common storage model used by array DBMS, since conventional FORTRAN or C++ array linearization orders will "optimize for one access pattern while making all others very inefficient" (Sarawagi and Stonebraker, 1994). Array DBMSs usually target to support *arbitrary* access patterns, leaving users the options to configure the tiling according to his access pattern. The system SciDB decomposes the array into a number of equal sized, and potentially overlapping chunks (Brown, 2010). Although the chunk partitioning might be customized by providing a configuration of the dimension-partitioning, once selected the chunk size is fixed and not adaptive. Another system is RasDaMan (Baumann et al., 1999), which employs an arbitrary tiled storage model with "R+-tree-like indexes" (Furtado and Baumann, 1999) for general multidimensional arrays. In particular, their storage model also adapts the tile size to the sparsity of the array, but it does not include heterogeneous tile data structures. An upper limit is imposed on the tile size depending on the system characteristics, e.g., the page size. Different tiling strategies are offered, which for example optimize the tiling along a dimension direction, or based on query statistics. The resulting tile layout is more flexible than ours, as it is could also be "totally nonaligned". However, a nonaligned alignment strategy is usually disadvantageous for join-similar tile operations such as tiled matrix multiplication. The main difference is that the interface to RasDaMan (AQL) is a database-centric, low-level array language that does not offer any linear algebra primitives as high-level operations. Instead, Furtado and Baumann (1999) define accesses exclusively as multidimensional range queries. A user might implement a matrix multiplication algorithm via AQL, but then he is responsible to write the query in an efficient way. This could be tedious without proper the language constructs, and the performance is likely to fall back behind that of carefully tuned C++ multiplication kernels that ATMULT uses internally. In particular, none of the systems really ties the storage model in with a cost-based, optimized matrix multiplication operator like ATMULT.

Tiled matrices. Many of the related papers proposed a rather fine-grained matrix-partitioning into small, fixed-size blocks. For instance, Vuduc and Moon (2005) used a variable block row structure (VBR) for SPGEMV. The latter is an *unaligned* BCSR, and serves the purpose to achieve higher performance by register-blocking. However, their maximum block size is 3×3 – hence, their focus is rather on microscopic tuning than on high-level tile optimizations. SystemML (Ghoting et al., 2011) as well uses a fixed blocking (2×2 in their example), where blocks are either a 1D array

(dense) or a hashmap (sparse). In other work, Huang et al. (2013) motivate a macroscopic, fixed block size (2048×2048) by arguing that two matrix tiles should fit entirely into main memory, in the context of a disk-based system. Hierarchical data structures in the context of matrix multiplication have also been proposed by Valsalam and Skjellum (2002). Like in our partitioning process, they also apply the Morton Order (Morton, 1966) on matrix elements to achieve a locality preserving partitioning of a large matrix, and a higher cache efficiency. In contrast to our AT MATRIX, they restrict their algorithm to dense matrices only, and do not consider sparse matrices at all.

Adaptive tensors. Besides fixed-size blocking approaches, we have found little research on *adaptively* tiled sparse matrix structures. One work that touches this approach was presented by Smith et al. (2015), which deals with tensor-matrix multiplications for three-mode (three-dimensional) tensors. They "propose a method of growing tiles to adapt to the sparsity pattern of a given tensor." In particular, they first statically partition the mode-1 dimension. Then, for each layer, they independently tile the second mode, and then the third mode, such that each tile nearly contains a number of floating point values that fits in cache. This is similar to our sparse tile grow condition, which is bound by the absolute number of non-zero elements in a sparse matrix tile. However, our data structure exploits the heterogeneity by using different data structures for different tiles, whereas Smith et al. (2015) use a uniform data representation for all tiles.

LAB-tree. The work probably most related to our adaptively tiled data structure is the Linearized Array B-tree (LAB-tree) by Zhang et al. (2011). Although their approach is focused on storing matrices on disk, and minimize I/O for batch-wise matrix writes, they employ a hybrid tree structure that stores sparse and dense parts differently. In particular, the LAB-tree contains key-value pairs as records, where the key is the linearization of the matrix input dimensions (i, j) given by a user-defined space-filling curve $f : (i, j) \rightarrow k$. Naturally, each leaf spanning a certain key range ℓ is constrained to accommodate a limited number of records κ, which is connected to the disk block size. If a leaf overflows, the leaf is either split into two sparse leafs, or a dense sub-array is created if the density in the *effective range* $\ell*$ (key range of actual non-zeros in the leaf) is above 0.5 and $\ell*$ is smaller than κ. As a result, the dense array covers the leaf range ℓ only *partially*. However, this might lead to the obscure effect that an existing dense array might be re-converted into two sparse leafs, if additional elements are inserted outside the effective range $\ell*$. Their aligned-splitting mode ensures that sparse leaf splits occur at multiples of κ, which asserts that – at least the sparse leafs – follow the higher level topology implicated by the space-filling curve.

As we did, Zhang et al. (2011) benchmark their LAB-tree data structure with matrix multiplications. Therefore, they load matrix blocks into memory and apply either a dense BLAS multiplication kernel, or the CHOLMOD (Chen et al., 2008) implementation for sparse parts. Since CHOLMOD requires a CSC structure while the LAB-tree contains key-value pairs, an additional conversion is required. In fact, they also use matrices of the Florida Sparse matrix collection in the measurements. For example, for a self-multiplication of the upper-triangular[6] matrix TSOPF_RS_b2383 (R3) their system takes 388 seconds. We re-ran our multiplication measurement by using only the upper triangular part of R3, and obtained a runtime of 2.4 seconds for SPSPSP_GEMM and 836 milliseconds for

[6]Note that many examples of the Florida Sparse Matrix Collection are symmetric matrices, for which only the upper-triangular is stored. Regarding our measurements, however, we explicitly mirrored the upper half to obtain the complete matrix.

ATMULT, which is a speed-up of almost three orders of magnitude. Obviously, contrasting the disk-based LAB-tree with our in-memory approach is like comparing apples and oranges. Nevertheless, apart from the tremendous performance difference, the AT MATRIX deviates from the LAB-tree for multiple reasons: firstly, AT MATRIX arranges sparse and dense tiles in a quad-tree structure and always aligns them to logical blocks, which is beneficial for join processing, e.g. in the tiled matrix multiplication. Instead, the LAB-tree could contain unaligned dense arrays that partially cover a logical block, and presumably need a copy before being further processed. Secondly, we use eight different matrix multiplication kernels. In contrast, CHOLMOD only contains SPSPSP_GEMM and SPDD_GEMM, so it is unclear how the remaining 5 situations are handled. Thirdly, ATMULT uses an enhanced cost model and tile conversion at runtime to accelerate the multiplication performance, whereas the selection of the multiplication kernel by Huang et al. (2013) is hand-coded and independent of hardware characteristics. Most importantly, the LAB-tree is indeed adaptive regarding *leaf size* and *storage type*, but unlike AT MATRIX, its matrix topology is predominantly influenced by the user-defined (or default) linearization. In fact, they therefore hand-crafted individual linearization functions, which create block sizes matching the pattern of each matrix under consideration.

5.7 SUMMARY

With the increase in data volume and computation effort in many analytical applications, efficient processing of large sparse matrices often becomes performance-critical. To fill the gap that is left between generic array DBMS and highly specialized numerical libraries, we redesign sparse and dense matrix processing for the LAPEG in a novel way, going beyond simple array data structures, BLAS libraries, and low-level tuning of sparse algorithms.

Therefore, we presented the AT MATRIX that has an adaptive, heterogeneous storage layout for large matrices of any topology, by internally using the native sparse and dense matrix data structures that are integrated in the column-oriented storage layer. Moreover, we showed how the matrix multiplication operator ATMULT accelerates matrix multiplication by elegantly applying several optimizations introduced in Chapter 4. These include the density estimation technique (SPPRODEST), a cost-based selection of multiplication kernels, and dynamic runtime conversions on tile-level. Our approach outperformed common multiplication algorithms, similar to those that are still used for example in MATLAB or R, by a factor of up to 6x, while maintaining configurable memory restrictions. Nevertheless, our optimization approach is general and orthogonal to the multiplication kernels. At present, we have not yet made use of several performance tweaks in our custom kernels, and expect even further improvement potential by implementing them.

6 UPDATES AND MUTABILITY

In this chapter, we address the final requirement that was identified in the introduction: data manipulation. To be more precise, we will discuss the mutability of matrices with regard to updates, inserts, and deletions of matrix parts. In this context, we consider applications that have mixed read-and-write workloads, and applications that are based on a dynamically evolving data set, which could be run in the Lapeg.

6.1 MOTIVATION

The ability of consistently updating and persisting data is a feature that is natural to users of ACID-compliant database systems, and is the reason why business data has been stored in relational DBMS for decades. In fact, the powerful data management capabilities of a DBMS makes it also appealing as a storage and processing platform for new application scenarios, e.g., scientific computations (Hey et al., 2009). Formerly, scientists across all domains have widely been depending on static data files, which were processed by hand-written programs. File-based data is tedious to maintain and update. However, a common (mis-)conception of poor performance is among the reasons why database management systems have barely been considered by science users as an alternative processing platform (Gray et al., 2005; Buneman, 2002). Moreover, relational DBMSs lack an interface for scalable math operations, in particular linear algebra (Stonebraker et al., 2013b). Nonetheless, the advantages of a DBMS and efficient implementation of analytical queries are not mutually exclusive. The DBMS-integrated Lapeg proves that a columnar in-memory DBMS not only acts as storage back-end, but can also be used as an efficient in-memory engine for matrix processing on a large scale.

We have argued in the introduction that if data is persisted and kept consistently in a single database system with integrated linear algebra functionality, expensive copying into external systems such as R or Matlab becomes dispensable. Furthermore, the existence of a *single source of truth* avoids data inconsistencies, since there are no redundant copies of the data in external systems. However, as a consequence of relying on a single data storage for matrices, all data changes have to take place on the primary data representations. Regarding the Lapeg these changes take place in the columnar storage layer that we introduced in Chapter 3.

Since matrices are non-static objects in several analytics workflows, this chapter focuses on the implementation and exposition of data manipulation commands. The nuclear energy state analysis

described in Section 2.2 serves as an example. As part of the importance truncation, several rows and columns, which refer to quantum states, are simultaneously cut out of a Hamiltonian matrix $\boldsymbol{H}$. Then, row and column pairs are iteratively added to the submatrix $\mathcal{M}_{\text{ref}}$ again.

Moreover, matrices are directly or indirectly manipulated in a multitude of other applications. In the environment of a transactional, online DBMS, matrix data sets evolve over time. For instance, consider the term-document matrix $\boldsymbol{A}$ of Section 2.3.1, where documents are continuously added to the database. The latter translates into appending additional rows or columns to $\boldsymbol{A}$. Another example is the social network graph (Section 2.3.2), where each non-zero element of the adjacency matrix corresponds to a connection between two persons (vertices). Every newly found or deleted connection requires an update of the corresponding matrix element. Hence, the physical and logical organization of matrices should enable reads, writes, and deletions of single elements, rows, columns, and individual matrix subregions. In order to offer these manipulations as functionality in the Lapeg, we propose a standardized user API that comprises all of the aforementioned manipulation primitives. Furthermore, we discuss the physical implementation of updates and deletions on matrices, and show that the required modifications seamlessly integrate with the matrix data structures in the column-oriented storage layer.

In particular, the main contributions of this chapter are:

- **Matrix application programming interface**. Similar to the data manipulation language of transactional, relational systems, we sketch the different access patterns and an application interface to read and manipulate matrices.

- **Mutable sparse matrix architecture**. We show how a two-layered main-delta storage can be leveraged to provide updatable sparse matrices. By using the native DBMS columns we automatically benefit from multiversion control and transactionality features of the DBMS.

- **Mutable adaptive tile matrix**. We present two different ways of integrating mutability into the AT Matrix representation, and how wide matrix manipulations spanning over multiple tiles are efficiently implemented using the AT Matrix's indirection layer.

- **Evaluation**. We thoroughly evaluate the performance of the mutable matrix architectures against alternative approaches. Therefore, the example workload was separated in order to consider mixed insert-read-queries and deletions separately.

This chapter is organized as follows: an introduction about the different matrix access patterns, manipulation types and linear algebra primitives is provided in Section 6.2. Then, we show how the presented manipulation API is applied using the nuclear physics analysis application as an example. In Section 6.3 we discuss the complexity of update operations on different matrix representations. Thereafter, we present the architecture of the mutable sparse matrix, and the AT Matrix$^{\text{GD}}$ and AT Matrix$^{\text{LD}}$ representations. The evaluation of these representations and multiple other approaches under different manipulation aspects is presented in Section 6.5. Finally, Section 6.6 provides an overview of related research on updatability in databases and mutable matrix structures.

6.2 APPLICATION PROGRAMMING INTERFACE

In typical database workloads, tables are dynamically manipulated by inserting, updating or deleting data elements. For this purpose SQL comprises several data manipulation language (DML) commands. In fact, some of the dynamic characteristics of relational database workflows also hold for many matrix-processing-based applications. For instance, in the nuclear science use case we presented in Section 2.2, matrices are updated in batches of single or multiple columns and rows. Hence, it is a valid assumption that a matrix is not just queried in a single pass, but rather modified in-between subsequent query executions as part of an analytical workflow. Therefore, the Lapeg offers an interface that allows matrix data to be manipulated in a similar manner as relational tables. This interface can then be used by an application programmer, for instance to construct a user defined function (UDF) for complex linear algebra scripts. In this section, we discuss basic manipulation primitives for matrix data from a logical perspective and describe how a matrix API could look like.

6.2.1 Matrix Processing Language

In our system, matrices are treated as *first-class citizens* throughout the complete software stack – from the physical storage layer up to the language interface. As of the current status, our DBMS-integrated Lapeg has two different interfaces. The first is a linear algebra extension of SQL; matrices are defined in a data definition language (DDL) by specifying its dimensions, which are stored as metadata in the system. Linear algebra operations on the matrices, such as multiplications, are exposed as built-in procedures. The second interface to Lapeg is the L language (Sikka et al., 2012) of the SAP HANA database, which is close to C. The following interface commands can be exposed to either of the two interfaces.

Note that we omit a full grammar description as well as a detailed discussion about syntactical language characteristics, since this would exceed the scope of this thesis. There are already popular and expressive language interfaces for matrix processing like those of R and Matlab, which are widely used, and could be integrated as another interface to Lapeg in future work. Instead, we rather focus on the *functional* aspects of the API. The concrete language implementation is orthogonal thereof, and might change.

6.2.2 Matrix Access Patterns

As a prerequisite for the following discussion, we briefly introduce the different access patterns of typical linear algebra algorithms. Rather than entering single matrix elements, the commands of the API take a generic sub-matrix *reference* as argument. That is, the argument contains the 2D-region that defines the bounding box of the submatrix. Each of the two matrix dimensions is queried by either a point, range or no restriction. Depending on the dimension restrictions, the resulting *2DRef* includes single elements, matrix rows and columns, row- and column ranges, rectangular submatrices, or the complete matrix (no restriction). Some possible cases are illustrated in Figure 6.1.

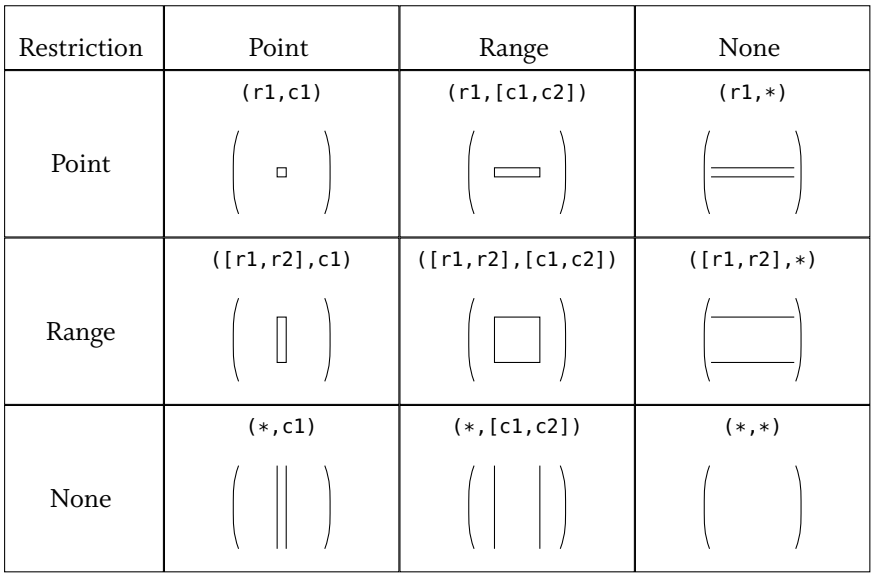


Figure 6.1: Matrix sub-array access patterns.

We now define the first command of our API, which is to retrieve elements from a matrix in an arbitrarily shaped area:

- **get**(*Matrix*, *2DRef*): is the logical counterpart of the relational *select ... where* statement, where the filter condition is replaced by a topological reference according to patterns shown in Figure 6.1. For example, $\textsc{get}(\boldsymbol{A}, (5, 3))$ returns the single matrix element $a_{5,3}$, whereas $\textsc{get}(\boldsymbol{A}, (*, 3))$ fetches the third column and $\textsc{get}(\boldsymbol{A}, (1, *))$ the first row of matrix $\boldsymbol{A}$. Two-dimensional submatrices are referenced by defining their row and column range, for instance $\textsc{get}(\boldsymbol{A}, ([2, 5], [3, 5]))$ to retrieve the rectangular region spanned by the included edge elements $a_{2,3}$ and $a_{5,5}$. The complete matrix is referenced by providing no restriction in both dimensions, thus by *2DRef*=(*,*).

6.2.3 Data Manipulation Primitives

The DML usually comprises commands to insert, delete, and update elements. In a sparse matrix context every element of a matrix space $m \times n$ logically already exists, regardless of its value. Thus, the logical interpretation of inserting a single matrix element is rather updating the existing, potentially zero-valued matrix element at the corresponding position (assuming its coordinates are within the matrix bounds). In the same way, a deletion of a single element is logically described by setting the nonzero value at the corresponding matrix position to zero. However, if a complete row or column, or a submatrix is inserted that spans over the full row- or column range, then an insert can also be interpreted as an *expansion* of the matrix by a number of rows or columns, respectively. Furthermore, a deletion of a matrix region spanning the whole row- or column range can be regarded as narrowing the matrix, respectively. Both operations alter the matrix dimensions. To remove this ambiguity, we define the following command semantics:

- **set/unset**(*Matrix*, *2DRef*, *ValHandle*): sets any single element or region in the matrix space $m \times n$ and overrides the previous value of the corresponding matrix region. As an example, SET($\boldsymbol{A}$, $(9, 3)$, 5.0) sets the value $a_{9,3}$ to 5.0, whereas SET($\boldsymbol{A}$, $([2, 2], [3, 3])$, *val*$[1.0, 1.0, 1.0, 1.0]$) sets all the values of the referenced square submatrix to 1.0. Conversely, the command **unset** sets the value of the referenced matrix coordinates to zero.

- **delete**(*Matrix*, *2DRef*): applies to either a $m \times p$ ($p \in [1, m]$ rows) or a $p \times n$ ($p \in [1, n]$ columns) subregion of the corresponding $m \times n$ matrix. It affects the matrix dimension in the way that adjacent parts are shifted to the first free place that was formerly populated by a deleted row/column. Thus, the resulting matrix has either the dimension of $(m - p) \times n$ or $m \times (n - p)$, respectively. For instance, DELETE($\boldsymbol{A}$, $(*, 3)$) executed on a 4×4 matrix $\boldsymbol{A}$ deletes the third column, and changes the dimensions of $\boldsymbol{A}$ to 4×3.

- **insert**(*Matrix*, *2DRef*, *ValHandle*): is the logical counterpart of the delete operation. The insertion of either p rows at row position i_0, or p columns at column position j_0, results in altered matrix dimensions of either $(m + p) \times n$, or $m \times (n + p)$, respectively. Rows that were previously below i_0 and columns right of j_0 are thereby shifted by offset p, i.e. $i \rightarrow i' = i + p,\ \forall\ i \geq i_0$, and $j \rightarrow j' = j + p,\ \forall\ j \geq j_0$.
  Note that rows or columns can also just be *appended* to the matrix, which refers to an insert at the lower ($i_0 = m$) or right ($j_0 = n$) end. There is no need to introduce an additional command for the append operation, but the matrix implementation could internally treat this special case more efficiently.

- **copy**(*Matrix*, *2DRef*, *2DRef*): copies single elements, complete rows, columns or submatrices from any source position to a target position. If the target bounding box exceeds the matrix bounds, the copy operation will only apply to full column- or row-range-spanning submatrices (i.e., $m \times p$ or $p \times n$). Then, the exceeding rows or columns alter the matrix dimension in the same way as the INSERT. The copy operation is derived by consecutive GET and SET operations if the target position stays within the matrix bounds, and by a get and combined SET/INSERT operation if the target position exceeds the matrix bounds.

- **flip**(*Matrix*, *2DRef*, *2DRef*): exchanges a $p \times l$ submatrix from a source position with an same-sized submatrix at a target position, which must not exceed the matrix bounds. The flip can not be implemented solely by consecutive get and set commands, since either the target or source region has to be buffered temporarily.

Apart from these basic commands, one could define a variety of further, more complex operations, which however, might be a composition of several basic manipulations. Note that we categorize operations such as the transposition or inversion of a matrix as mathematical methods rather than a DML manipulation. Hence, we restrict our discussion to the set of manipulation operations as of above, which are fairly common. As an example, setting single elements in a matrix is a fundamental operation in a variety of applications, e.g. LU-decomposition methods, including Gaussian Elimination or the Doolittle algorithm (Mittal and Al-Kurdi, 2002). Other linear algebra algorithms tend to remove complete matrix rows or columns. In the example from nuclear science described in Section 2.2, chunks of k rows and columns are deleted from and inserted to the sparse matrix.

More obvious examples are graph algorithms, where the elimination of a graph vertex corresponds to the removal of the respective adjacency matrix row and column.

6.2.4 Implementation of the Nuclear Physics Analysis

In Section 2.2 we introduced the importance truncation method as part of the nuclear physics analysis, which is an algorithm that iteratively manipulates a large, sparse matrix. Here we focus on the implementation part, in particular the mapping of matrix manipulations onto our API commands. To avoid a redundant description of the analysis, we refer to Section 2.2 for further details.

Algorithm 16 Implementation of Algorithm 2 using the manipulation API.

```
1:  function IMPORTANCETRUNCATION(Hamiltonian Matrix H)   ▷ H: Symmetric n × n Matrix
2:      M_ref ← H                                          ▷ Initialize working copy
3:      DELETE(M_ref, ([r_1, r_2], *))                     ▷ Delete rows from range r_1 to r_2
4:      DELETE(M_ref, (*, [c_1, c_2]))                     ▷ Delete columns from range c_1 to c_2
5:      v_ref ← v_0
6:      while not converged do
7:          {u_i} ← PERTURBATE(H, v_ref) ▷ Get additional basis vectors from perturbation theory
8:          for each vector u_i do                         ▷ The p vectors u_i are of dimension n + p.
9:              INSERT(M_ref, (n, *), (u_i^T)_(1..n))
10:             INSERT(M_ref, (*, n), (u_i)_(1..n+1))
11:             n ← n + 1                                  ▷ The matrix dimensions increase by 1
12:         T, V ← LANCZOS(M_ref)          ▷ Solve and return eigenvectors and tridiagonal matrix
13:         v_ref ← GET(V, (*, 1))                         ▷ Set v_ref to first eigenvector (first column of V)
14:     {λ_k} ← DIAGONALIZE(T)
15:     return {λ_k}
```

Algorithm 16 shows the implementation of the importance truncation that was presented as pseudocode in Algorithm 2. The code contains several calls to the manipulation API of the Lapeg. Lines 3-4 correspond to the creation of the submatrix ($\mathcal{M}_{\text{ref}}$ in Algorithm 2) by removing several quantum states that were determined by a prior calculation. Since the matrix is symmetric, the operation comprises the removal of both the row and the respective column range. Hence, the DELETE API command is called twice (lines 3-4). In the actual loop, several vectors derived from perturbation theory are appended to the matrix by the INSERT command (line 9-10). Again, the p vectors of dimension $n + p$ are added both as columns and in transposed form as rows, due to the matrix symmetry. Note that only the first n elements of each $\boldsymbol{u}_i^T$ are appended as row in line 9, whereas the first $n + 1$ elements of each $\boldsymbol{u}_i$ are added as column in line 10.

Furthermore, the loop contains calls to the LANCZOS (line 12), and to the PERTURBATE function. As we have described in Section 2.2, the LANCZOS method (Alg. 1) is an iterative algorithm to find the dominant eigenvalues of a symmetric matrix, and most of its computational effort is spent on multiple matrix-vector multiplications. Since the LANCZOS function call does not manipulate $\boldsymbol{M}_{\text{ref}}$, it can be considered as the part of the algorithm that *reads* the matrix. The definition of the PER-

TURBATE function (line 7) contains a lot of domain-specific calculations. As it does not have any qualitative influence on the discussion about our manipulation interface, we omit further details for the sake of clarity.

The body of the while loop is repeated until a convergence criteria is reached, that is when the deviation of $\boldsymbol{v}_{\text{ref}}$ between adjacent iterations falls below a predefined threshold. Finally, the resulting eigenvalues are calculated by diagonalizing the tridiagonal matrix $\boldsymbol{T}$. The set of eigenvalues $\{\lambda_k\}$ corresponds to the lowest energy states of the nucleus that are encoded in the Hamiltonian matrix $\boldsymbol{H}$.

The energy state analysis workflow combines large scale linear algebra processing with dynamic data manipulation on a large sparse matrix. Our evaluation in Section 6.5 shows that such an analysis scenario significantly benefits from our mutable data structure that is presented in this chapter. Nonetheless, the manipulation API is also applicable to the other use cases introduced in Chapter 3. In the text mining scenario (Section 2.3.1), it is usual to add new documents to the data set, which corresponds to insertions of rows to the document-term matrix (or accordingly, columns to the term-document matrix). Similarly, the data set in graph applications (Section 2.3.2) is often subject to permanent changes, e.g., a social network graph that is continuously growing over time. In this case the creation of new connections correspond to setting non-zero elements in the matrix $\boldsymbol{G}$. Deletions of matrix ranges may occur as a preprocessing step in an interactive analysis workflow. For instance, given a large matrix $\boldsymbol{G}$, a user might want to create a working copy $\tilde{G}$, and manually "strip it down" to refine the analysis to run on the most interesting parts of the data set.

6.2.5 Language and UDFs

Algorithm 16 is written in an abstract higher level language. The general idea is to expose our matrix manipulation API together with basic linear algebra operations to a scripting language that can be used to build UDFs. The latter are constructs provided by many DBMS to offer the user an opportunity to implement custom functionality for complex queries. Depending on the particular DBMS, UDFs can either be written in SQL, or in a programming language like C++, R, or others. Algorithm 16 by itself represents the definition of the UDF IMPORTANCETRUNCATION. Moreover, it contains calls to external UDFs PERTURBATE, LANCZOS, and DIAGONALIZE. The idea of leveraging UDFs to encapsulate linear algebra functionality and enable iterative algorithms has also been presented by Ordonez and García-García (2006), and by Hellerstein et al. (2012) with MADLib, which is a collection of statistics algorithms implemented with UDFs combined with SQL.

In particular, the UDF language interface should include commands to perform basic linear algebra operations, such as matrix-vector multiplications that are needed in the LANCZOS algorithm. For instance, Hellerstein et al. (2012) use SQL commands together with additional, user-defined infix operators for matrix multiplications and additions (see Section 2.6.1). For our implementation we used the L language, which we extended with linear algebra functionality. L is used in the SAP HANA database (Sikka et al., 2012), and is a language close to C. However, we emphasize that our work is generally independent from the language interface. The only requirement is that our previously defined matrix API can be exposed to the application programmer, in order to benefit from our low-level operators implemented in the LAPEG.

6.3 UPDATABLE MATRIX ARCHITECTURE

The challenge of many analytical database systems, which strive for fast query execution and immediate updates, is the dualism between read- and write-optimized data structures. The same situation holds for many matrix representations that are used in linear algebra implementations, since they are often optimized for reads only.

6.3.1 The Cost of Updates

In this context, we examine the representations that we presented in Sections 3.4.1 to 3.4.2 according to the following two criteria: the efficiency of different read access patterns, and further, the complexity of manipulations.

Dense Matrices

The dense matrix representation introduced in Section 3.4.1 (Figure 3.3b) exhibits the following behavior:

Get/set elements. Single elements are read (GET) in $\mathcal{O}(1)$ time and can be written (SET) in-place, yielding a low constant complexity for single element updates. Moreover, disjoint parts of a dense matrix can be written concurrently, which is exploited by our multiplication kernels where each thread writes into its assigned row block.

Insertions. The *dimension-altering* insertions or deletions of rows, columns, or submatrices are more tedious to implement. In particular, the linearization plays an important role in this case: when inserting a matrix *row* into a row-major-ordered matrix at position i_0, only the rows below $\forall\ i \geq i_0$ have to be shifted by one row size in memory. In contrast, the insert of a matrix *column* changes the leading dimension of the dense array, which incurs a rearrangement of the whole matrix. Moreover, the new matrix size could exceed the allocated memory. In this worst case situation, the matrix is reallocated according to its altered dimensions $m \times n \rightarrow m' \times n'$, and every element of the old matrix is copied to the newly allocated one.

Deletions. The linearization also plays a major role regarding the deletion performance of rows and columns. If a deletion occurs in a "linearization-friendly" way (in the leading dimension), e.g., when a row chunk $[r_1, r_2]$ is deleted in a row-major representation, only elements below r_2 must be rearranged in order to avoid a gap in the dense representation. In this case, the rearrangement can be implemented by using a cache-efficient mem-copy instruction. If instead the deletion is "linearization-unfriendly", i.e., when a column chunk $[c_1, c_2]$is deleted in a row-major representation, then all remaining matrix elements have to be rearranged.

Unordered Triple Table

The triple table introduced in Section 3.4.2 is considered as an unordered sparse matrix representation, since the matrix triples are not necessarily sorted. It has the following update characteristics:

Get/set elements. For single matrix element as well as row and column lookups (GET), a full column scan on $\boldsymbol{A}$.Row and $\boldsymbol{A}$.Col is required, leading to a complexity of $\mathcal{O}(N_{\text{nz}})$. Hence, all algorithms that benefit from indexed accesses, e.g., the sparse outer product algorithm for matrix-vector multiplication, are relatively inefficient using the triple representation, whereas the inner product formulation can be implemented quite efficiently by a single scan (see Section 3.5).

Regarding a matrix element (SET) it might happen that the non-zero element already exists in the data structure. In this case, it can be updated in-place but must first be located, which also requires a full column scan. Therefore, we add an *unchecked set* mechanism to the API specification[1], in order to set single matrix elements without a check for a previous element value at the specified coordinates. The check can be omitted when it is clear from the application side that this matrix element had not been set previously.

Direct sets have the advantage that additional non-zero matrix elements (INSERT) can just be added to the end of the respective $\boldsymbol{A}$.Row-, $\boldsymbol{A}$.Col-, and $\boldsymbol{A}$.Val- columns of the triple table (see Figure 3.3c1). Each of the table columns has a pre-allocated buffer where elements are inserted into. When the buffer capacity is reached, a larger memory segment is allocated and the old data is copied to the new memory location. This is a similar behavior to that of the `vector` container of the C++ standard template library, which doubles its capacity with every re-allocation, while exhibiting an *amortized constant* insert time (Tarjan, 1985; Mortoray, 2014).

Insertions. As mentioned before, one can distinguish between *displacing inserts*, e.g. when a row or column is inserted in the middle of the matrix and requires a shift of the adjacent rows/columns, and *appending inserts*, i.e. when the row or column is inserted at the lower or right end of the matrix, respectively. The former requires a relabeling of triple coordinates which is in $\mathcal{O}(N_{\text{nz}})$ (scan complexity.) In contrast, the latter can be efficiently implemented using unchecked sets, where all elements are just appended to the end of the data structure in amortized $\mathcal{O}(1)$ per element.

Deletions. The naive triple table is an append-only structure. Thus, in-place deletions of matrix elements in the native triple table require a complete rearrangement, similar to the deletion of rows (or columns) in a dense matrix.

To summarize, the major drawback of the unordered triple representation is the poor performance for querying elements of a particular row and or column, but its advantage are efficient unchecked sets and appending inserts of matrix rows and columns.

CSR Representation

Lastly, we evaluate the read and write characteristics of the CSR representation (Section 3.4.2, Figure 3.3c2). The CSR is an ordered representation, for construction it requires a preceding sort of the matrix elements by their row coordinates. It has the following behavior:

Get elements and matrix rows. The obvious advantage of CSR is that the rows are indexed via the row pointer, resulting in an $\mathcal{O}(1)$ access to the beginning of any matrix row. Since matrix columns are not indexed in CSR, a column read incurs a full scan of the data structure (we already

[1] Direct sets are expressed by an additional flag in the SET command.

discussed the dualism of the CSR and the column-indexed CSC representation in Chapter 3.) To address a *single* element in a CSR matrix via its coordinates, a partial scan is required, starting from the beginning of the matching row. Assuming an average population of N_{nz}/m for a matrix row of an $m \times n$ sparse matrix with N_{nz} non-zero elements, the resulting average GET/SET complexity of a single matrix element is $\mathcal{O}(N_{\text{nz}}/m)$. In fact, by sorting the elements in each matrix row by their column coordinates, a binary search could be used to accelerate searches. Hence, in the sorted case the complexity is reduced to $\mathcal{O}(\log N_{\text{nz}}/m)$. The sorted property is not fundamental to the CSR representation, but could further improve the performance of CSR-based algorithms. For instance, an ordered row results in a cache-friendlier access pattern on the sparse accumulator structure in the sparse matrix multiplication algorithm (line 13 in Algorithm 10.) This can explain the following effects: during a single row multiplication, the column coordinates refer to positions in the sparse accumulator array. If they were randomly arranged, many cache-lines would have to be evicted and reloaded again, whereas an ascending order leads to cache-friendly writes.

Row coordinate lookup. Unlike the uncompressed triple representation the row coordinate is not stored explicitly for each matrix element in the CSR representation. Hence, the complexity for the row coordinate lookup is not constant, i.e., for finding the matrix row coordinate i of an element at some physical position p in the $\boldsymbol{A}$.Val and $\boldsymbol{A}$.Col columns (see Figure 3.4). This operation requires to determine the interval of the row start positions $I : [I_i, I_{i+1}]$ in the row pointer, where $p \in I$ and i is the matrix row that starts at position I_i. Again, a binary search can be used for this task, incurring an asymptotic complexity of $\mathcal{O}(\log m)$ that depends on the size of m of the ordered row pointer structure. Note that a partial or complete decompression of the row pointer is not necessary. However, such row-lookups are rare, since most mathematical algorithms can be written in a way that they make use of the row index.

Insertions. The native CSR is an ordered thus static representation. Consequently, randomly inserted non-zero elements can not just be appended at the bottom of its structure. Since there is no buffer space for additional elements within the row clusters, a rebuild of the complete CSR structure is required to insert any new non-zero matrix element. Eventually, for complete matrix row insertions (and only if there is buffer space at the bottom of the CSR structure,) one could shift all elements that are succeeding the inserted row(s) by the size of the newly inserted elements with a single memory move operation.

Deletions. Just like the triple table, there are no gaps in the native CSR structure. Hence, every element that is removed would require a rearrangement of the remaining elements in the $\boldsymbol{A}$.Col and $\boldsymbol{A}$.Val columns as well as an update of the row pointer entries. However, even for row deletions the movement of large memory sections is inefficient and should be avoided for *ad-hoc* manipulations[2].

In general, row-based operations are more efficiently implemented on the CSR representation than column-based operations due to the nature of the representation: manipulations on matrix columns always affect every matrix row, and consequently, the complete CSR structure has to be rearranged. The duality in the performance between row- and column manipulations in the CSR structure is observable in our evaluation in Section 6.5.

[2]We denote the deletion or insertion of a small number of elements, rows, or columns with ad-hoc manipulations.

6.3.2 Mutable Sparse Matrix

The result of the previous discussion is that many native matrix representations are not mutable with reasonable effort. However, many applications indeed require single or batch-wise ad-hoc matrix updates, or iterative data changes over time. In particular when matrices are very large and sparse, a reconstruction of optimized matrix representations such as CSR is costly regarding time and memory efficiency. Depending on the workload, a full reconstruction of CSR might not amortize over time. Therefore, we present an approach to integrate mutability for sparse matrices without losing the advantage of read-optimized storage structures. Later in this section, we extend the mutability features to our novel AT Matrix data type.

The Main-Delta Approach

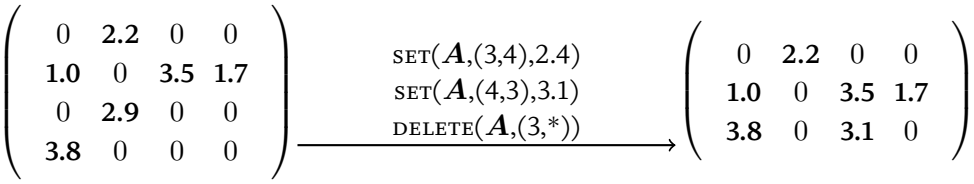
Common read-optimized, indexed sparse matrix representations (e.g., in Saad, 1994) are static, whereas the update-friendly structures such as the triple table often exhibit poor algorithmic performance. Since these opposed characteristics are not satisfactorily achievable by a single structure, we employ a *main-delta* mechanism. In particular, the physical sparse matrix structure is *conceptually* separated into two complementary representations: a static main part and an updatable delta part. The main part is a read-optimized, compressed structure, whereas the delta is an uncompressed, write-optimized structure. Figure 6.2b illustrates the conceptual separation of the mutable sparse matrix into a main, and a delta part. As a convenient side effect, the mutable matrix is essentially a composition of the data structures of Section 3.4.

The general idea of separating data structures into a read-optimized and an incremental, write-optimized part has been presented before in the context of databases (Jagadish et al., 1997) or graph processing systems (Macko et al., 2015). Furthermore, the concept is also applied in column-stores (e.g. Färber et al., 2012) to enable updates for compressed column tables. In particular, the columns in the column-oriented storage layer *already implement a main-delta mechanism*, since the Lapeg is embedded in a transactional DBMS (Fig. 4.2). We show that the updatable main-delta matrix is seamlessly implemented using the native, column-integrated main-delta mechanism, while only minimal adaption is required (Fig. 6.2c).

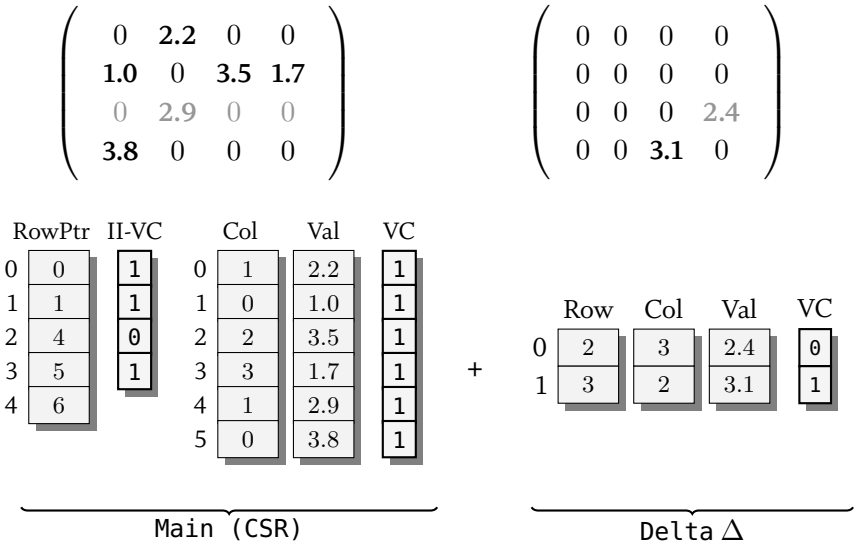
In a main-delta architecture, insertions of new data are exclusively executed on the delta part, which naturally grows over time in a dynamic workload. Moreover, the incoming queries are processing the data that is contained in both the main and the delta structure. Since the read-performance of the delta structure is commonly worse than that of the main, the elements in the delta component are periodically merged into the main storage. This operating mode is also known as *log-merge*: each merge step includes a reorganization of the data structures, implemented completely transparently to the user. The reorganization usually includes the reconstruction of the read-optimized data structures in the main part.

Static Main Component

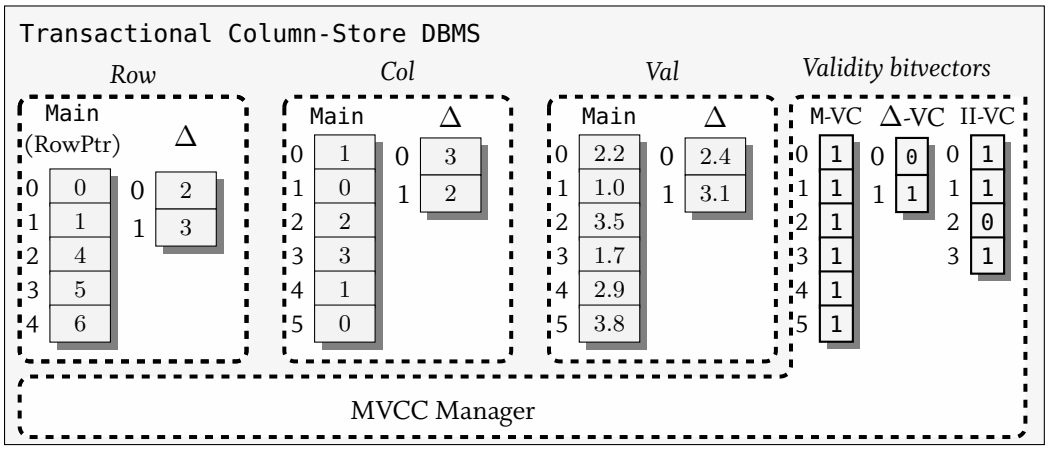
The static main component of our architecture consists of a data representation that is optimized for read-operations, with focus on matrix-vector and matrix-matrix multiplications. In Chapter 3 we inferred that using the CSR representation tends to result in the best performance especially



(a) Matrix manipulation sequence that causes a main-delta structure as of Figure 6.2b.



(b) Mutable sparse matrix architecture that *conceptually* separates matrix $\boldsymbol{A}$ into a static main and an incremental delta structure. Additionally, the validity bitvectors (VC) are shown, which keep track of deleted matrix elements (e.g., a_{34} in the delta component), and deleted rows (e.g., $\boldsymbol{a}_{3*}$ in the main component).



(c) The implementation of a mutable sparse matrix via version-controlled table columns in a transactional, column-store DBMS. The validity bitvectors are provided by the MVCC manager.

Figure 6.2: Manipulations of a matrix from a conceptual point of view, and how it is actually implemented in the column-oriented storage layer.

with regard to matrix vector multiplications, which is backed by experiments of related work (e.g., Vuduc, 2004). Moreover, the CSR representation integrates well with the column-oriented storage layer of a column-store DBMS, and is widely used in many high performance libraries for linear algebra. Hence, it is reasonable to select the CSR layout as the main component of the mutable sparse matrix.

The CSR structure is constructed in the reorganization phase of the system, which merges the delta into the main component. In this phase, all non-zero elements are initially merged together into a large, unordered triple table (see Figure 3.3c1 and c2). Then, the table is ordered by a nested sort: firstly, the triples are ordered according to the matrix row coordinate, and secondly, each row chunk is internally sorted by the values of their column coordinate. Finally, $\boldsymbol{A}$.Row is replaced by the row pointer as illustrated in Figure 3.4. All elements are now confined in the read-optimized CSR representation, and the delta structure is empty.

During the construction of the CSR representation, most of the time is spent in the sort, which incurs a $\mathcal{O}(N_{\text{nz}} \log N_{\text{nz}})$ complexity. Therefore, the reorganization phase is scheduled in batches, i.e., it is triggered only when the size of the delta structure has reached the relative delta size threshold Δ_T. The latter is a trade-off between the reduced read performance of the non-empty delta, and the reorganization overhead.

Incremental Delta Component

The aim of the main-delta concept is to maintain little overhead for slight changes, like setting or inserting a small number of non-zero matrix elements, while providing a fairly efficient read access at the same time. In contrast to the read-optimized main component, the incremental delta structure is designed for fast writes. Based on the example of the nuclear physics application, the delta structure should support efficient append-insertions of rows and columns. According to its manipulation characteristics, the unordered triple table suffices this requirement ($\mathcal{O}(1)$ time per element).

However, instead of an additional, high-level main-delta separation of the matrix into a CSR and triple table as indicated in Figure 6.2b, we can simply use the columns' own delta structures in the column-oriented storage layer as shown in Figure 6.2c. As a result, the delta parts of the *Row-*, *Col-*, and *Val-* columns together form an unordered triple table that coexists with the static CSR structure, which in return is formed by the columns' compressed main parts. Hence, for a given snapshot the internal representations of Figure 6.2b and 6.2c are *physically equivalent*. All inserts and direct sets of non-zero elements are now efficiently incorporated by the incremental column deltas, before they are merged into the main CSR structure in the reorganization phase.

Validity Control Structures and Transactionality

We already mentioned that deletes on the native CSR and triple representations can not be trivially implemented without moving large memory sections. To avoid such costly operations, both main and the delta component are *extended* by an additional control structure that flags meanwhile deleted elements. In particular, there are *validity control* (VC) bitvectors for the CSR representation (including the row pointer index) as well as for the unordered triple table (see Figure 6.2b). For each deleted element, the bit at the corresponding position in the respective main or delta VC-bitvector

is unset. Moreover, if a complete matrix row is removed, for instance row i, then the corresponding bit at position i of the II-VC index bitvector is unset. A similar concept to our validity control structures is also used in related work (Macko et al., 2015), where they are called deletion vectors.

The advantage of our approach using the column-oriented storage layer of a DBMS supporting transactions is that we can just use the integrated multiversion concurrency control (MVCC) functionality (Sikka et al., 2012). Since our matrix representations builds on top of version-controlled columns, our mutable sparse matrix already benefits from the native transactional features of the DBMS "out of the box", avoiding the need to hand-craft updatable data structures like in (Macko et al., 2015). Consequently, our mutable sparse matrix tackles transactionality by leveraging the MVCC manager of the DBMS. The latter administrates the validity bitvectors, and maintains multiple consistent snapshots of the data. Hence, whenever the MVCC manager receives a read request, it returns the bitvectors in a version that is valid in the current transaction session. The main- and delta column parts on their own do not contain any visibility information. The main parts are in the state as of the last reorganization phase, and the delta parts contain all appended (and possibly already deleted) matrix elements. Eventually, a consistent snapshot is obtained for each column by the main combined with the delta part and the corresponding validity control bitvectors received from the MVCC manager. The only adaption we employ is to equip the MVCC with an additional validity bitvector (II-VC) for the *compressed row column* (RowPtr), which optimizes the performance for matrix row reads.

The functionality of the main-delta mechanism and the validity bitvectors is illustrated by Figures 6.2a and 6.2b: starting from an initial matrix that is fully contained in the main part, Figure 6.2a shows a manipulation sequence that causes exactly the state of the main-delta structure that is sketched in Figure 6.2b. The manipulations include two SETS of non-zero elements, followed by the deletion of a matrix row. Consequently, the row pointer bitvector II-VC at position 2 is unset, as well as the delta VC-bitvector at position 0[3].

Determining the Delta Size

The selection of the delta threshold Δ_T impacts both the read-query time and the amortized time spent in the delta merges. For a fast insertion of elements, the delta should be large enough to avoid frequent delta merges. Conversely, a large delta has a negative impact on the read response times, since the triple representation has an inferior multiplication performance compared to CSR and AT MATRIX. As a result, finding a reasonable delta size is an optimization problem.

In fact, the delta size optimum also depends on the workload, in particular the ratio of read queries vs. write queries. The reorganization strategies of data structures in database has been thoroughly discussed in the literature (e.g., an overview is given by Sockut and Iyer, 2009). We adopt a theoretical model of Shneiderman (1973), which aims at optimizing database reorganization points depending on the search deterioration rate, and reorganization costs. It comprises an expression for the total search excess costs $C(t)$ incurred by the additional buffer structure while assuming regular reorganization intervals. In particular, $C(t)$ depends on the reorganization interval t (Shneiderman,

[3]Note that we internally use zero-based indexing.

1973):

$$C(t) = \frac{1}{2}\left(T(\Theta_1 - \Theta_2)t\right) + \frac{TR_0}{t} + \frac{1}{2}\left(\frac{T^2}{t} + T\right)\mu, \tag{6.1}$$

where Θ_1 is the rate of deterioration of the search cost without reorganization (here: cost for reads on both CSR-main and triple-delta), Θ_2 is the rate of deterioration of the search cost if the data structure (here: only CSR main) was constantly being reorganized, R_0 is the initial reorganization/merge cost, and μ is the rate of increase of reorganization cost. T is the total time frame under consideration, and, in case of a *constant insert rate*, T can be set equivalent with the total number of inserts.

Consequently, by setting the derivation $\partial C/\partial t$ to zero and solving for t, the optimal time interval for reorganization yields:

$$t = \left(\frac{2R_0 + \mu T}{\Theta_1 - \Theta_2}\right)^{\frac{1}{2}}. \tag{6.2}$$

Note that for a constant insert rate, t is equivalent to Δ_T. Hence, if the underlying assumptions are fulfilled and the hardware-specific parameters Θ_1, Θ_2, μ are known, we can apply Equation 6.2 to find an approximately optimum delta size. Note that we determine the optimal delta size empirically in the evaluation section of this chapter, since the parameters in Equation 6.2 are not known in advance.

6.3.3 Delta-Aware Operators

We now show how the main-delta approach is integrated in linear algebra operators by using the example of matrix-vector multiplication. As mentioned before, the kernel algorithms of the Lapeg depend on the matrix representation. For example, the sparse matrix multiplication algorithm (Alg. 10, Section 3.6.2) is tweaked for CSR-represented matrices. Hence, to run a multiplication operation on the main-delta storage, different algorithms apply to the main and the delta part of the matrix. In this section, we describe how the sparse matrix-vector multiplication is implemented on our mutable sparse matrix architecture. This kernel is of particular importance, since it serves as a building block for many applications.

We can rewrite the mutable matrix $\boldsymbol{A}$ by logically separating it into its physical components:

$$\boldsymbol{A} = \boldsymbol{A}^{\mathrm{M}} + \boldsymbol{A}^{\Delta} \quad \boldsymbol{A} \in \mathcal{R}^{m \times n}, \tag{6.3}$$

consisting of a main part $\boldsymbol{A}^{\mathrm{M}}$, and a delta part $\boldsymbol{A}^{\Delta}$. Along with expression (6.3), a matrix vector multiplication decomposes into

$$\boldsymbol{y} = (A^{\mathrm{M}} + A^{\Delta})\boldsymbol{x} = \underbrace{\boldsymbol{A}^{\mathrm{M}}\boldsymbol{x}}_{①} + \underbrace{\boldsymbol{A}^{\Delta}\boldsymbol{x}}_{②}, \tag{6.4}$$

and consequently:

$$\boldsymbol{y}^T = \underbrace{\boldsymbol{x}^T(\boldsymbol{A}^{\mathrm{M}})^T}_{①} + \underbrace{\boldsymbol{x}^T(\boldsymbol{A}^{\Delta})^T}_{②}, \tag{6.5}$$

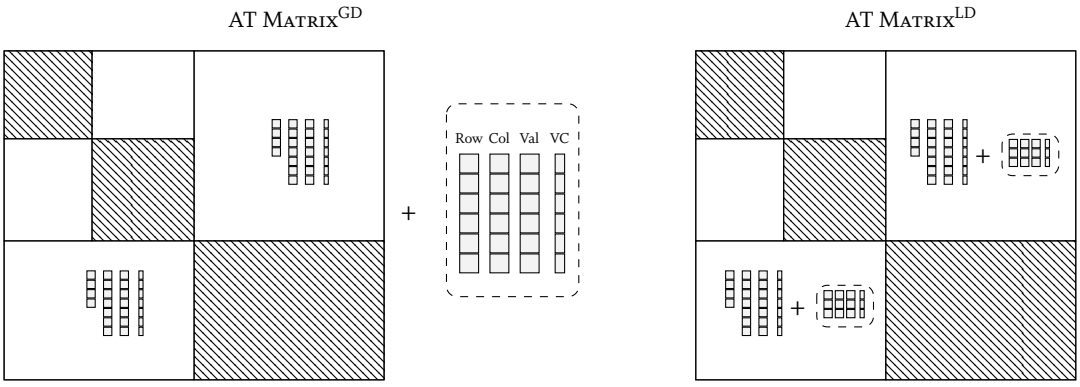
Algorithm 17 GEVM for mutable sparse matrices

```
1:  function GEVM(Mutable SpMatrix A, Vector x)
2:      y ← 0                                                    ▷ y: Target Vector
3:      MvccManager.INITIALIZEVALIDITYBITVECTORS(A, GETTRANSACTIONID())
    ▷ Main part ①
4:      for x_i ≠ 0 ∈ x do
5:          if A.II-VC.at(i) = 0 then continue
6:          a_row_start ← A.RowPtr.at(i)
7:          a_row_end ← A.RowPtr.at(i + 1)
8:          for a_row_start ≤ a_iter < a_row_end do
9:              if A.VC.at(k) = 0 then continue
10:             a_col ← A.Col.Main.at(a_iter)
11:             y.Val.at(a_col) += x_i × A.Val.Main.at(a_iter)
    ▷ Delta part ②
12:     for 0 ≤ a_iter < A.Delta.size() do
13:         if A.Δ-VC.at(a_iter) = 0 then continue
14:         a_col ← A.Col.Delta.at(a_iter)
15:         y.at(a_col) += x.at(A.Row.Delta.at(a_iter)) × A.Val.Delta.at(a_iter)
16:     return y
```

due to the linearity of the operation. As a result, the two multiplications can be processed separately and independently on both the main and the delta part of $\boldsymbol{A}$.

The implementation of the sparse matrix-vector multiplication based on our main-delta architecture is sketched in algorithm 17. We use $\boldsymbol{A} = \boldsymbol{A}^T$, since the applications discussed here are based on symmetric matrices (e.g., the nuclear physics analysis described in Section 6.2.4). The first part of the code shows the multiplication with the CSR main component (line 4-11). It is an adapted version of the outer product algorithm (Alg. 8), which additionally takes the validity control structures into account. For each non-zero vector element x_i an index lookup for the corresponding matrix row i provides the start and end position of the containers in the main structure. In addition, the II-VC bitvector is checked for each x_i, in order to skip meanwhile deleted matrix rows. The same check is performed in the inner loop for each element using the VC bitvector, right before the target vector $\boldsymbol{y}$ is written. The second part of the algorithm (line 12–15) iterates over valid elements of the incremental delta structure and adds the element multiplication result to the respective position in the target vector. As a matter of fact, the additional checks of the VC bitvectors incur some overhead in the inner loops of both parts. However, since the zero-entries in the bitvector are reset with the delta merges, the if conditions in lines 9 and 13 yield true in most cases, making it easily predictable for the CPU's branch prediction. The performance comparison of the delta-aware GEVM compared to static CSR GEVM is also part of our evaluation in Section 6.5.

For illustrational purposes, Algorithm 17 shows the sequential version of the multiplication. Nonetheless, we already indicated in Section 3.5.2 that the outer-product matrix-vector multiplication can be run in parallel by producing local results per worker thread, followed by a merging step



(a) Global delta structure. (b) Local delta structures.

Figure 6.3: Two variants of a mutable AT Matrix. For illustrational purposes, the tile substructures are sketched for the two largest tiles in the matrix.

that combines the local results into $\boldsymbol{y}$. The delta-aware version in Algorithm 17 can be additionally parallelized by executing the main ① and the delta ② part concurrently.

6.4 MUTABLE ADAPTIVE TILE MATRIX

So far we have discussed updates only for the basic sparse and dense matrix representations of Chapter 3. In this section, we address the integration of mutability into our heterogeneous AT Matrix representation (Fig. 5.3), which was introduced in Chapter 5. In particular, we add a mutability mechanism to AT Matrix, and compare it to several alternative approaches for dynamic workloads.

In Section 5.5 we already observed that the AT Matrix is superior to the CSR representation for static matrix multiplication workloads. Hence, the next step is to examine if the adaptive tile layout is also advantageous for update operations. The heterogeneous substructure naturally influences the read and write accessing behavior compared to the homogeneous data representations. The major advantage of the AT Matrix is the inherently contained indirection layer and data partitioning in the form of the grid that maintains the submatrices. This characteristic contributes to more efficient read- and write operations. We already elaborated this behavior regarding row- and column reads in Section 5.2.4. Similarly, write operations only affect the locally confined matrix tiles, and merely adjust the metadata. As a consequence, we expect that these operations are also significantly less costly than manipulating a matrix-wide CSR representation, particularly in column direction.

Following the introduction of the main-delta concept for the mutable sparse matrix, the remaining open question is how to combine these results with the adaptively tiled AT Matrix. We are considering two ways to integrate mutability with the AT Matrix, which are illustrated in Figure 6.3. In particular, these are:

1. A static AT Matrix as main structure, and a single triple table acting as *global delta* for all tiles. We denote this variant as AT Matrix^{GD} (Fig. 6.3a).

2. A dynamic AT Matrix with all tiles replaced by the corresponding mutable versions of sparse- and dense tiles. Hence, every sparse matrix tile is updatable via their *local delta* structures. We call this variant AT Matrix^{LD} (Fig. 6.3b).

The first option (AT Matrix^{GD}) is conceptually similar to the mutable sparse matrix of Section 6.3.2. Nevertheless, this time we employ a *separate*, matrix-wide delta triple table that coexists with the AT Matrix. The global delta has the advantage that single inserts are inexpensive, since they are just appended to the delta triple table. In contrast, the second option has the advantage that it enables fine-granular merges on a local level. Each of the local tile-delta merges requires only a fraction of the time compared to a global delta merge. We evaluated and compared the two different approaches in the evaluation section.

Get/Set Elements

In order to read (get) a single element from the AT Matrix$^{LD/GD}$ representations, the affected matrix tile is first located via the internal index structures. The tile location is determined by tree searches on the row- and column dimension index tree structures shown in Figure 5.3. The only difference is that for AT Matrix^{GD} the global delta structure is scanned, whereas only the tile-local delta components for AT Matrix^{LD}.

sets of matrix elements into the AT Matrix^{GD} are handled similarly as in the mutable sparse matrix representation presented before. Added non-zero elements are simply appended to the global delta triple representation, until its size has reached the delta threshold Δ_T. Then, it is merged with the main AT Matrix structure, which involves a complete rebuild and re-run of the partitioning process.

In order to set a single element into the AT Matrix^{LD}, the affected tile is first located and then inserted into the local delta components. As a consequence of the search overhead, the set costs of AT Matrix^{LD} are slightly increased over that of AT Matrix^{GD}. Conversely, the advantage of AT Matrix^{LD} over maintaining a global delta is that the tile deltas are merged locally and asynchronous, thus distributing the merge costs over time. In particular, if the local tile density exceeds the read density threshold ρ_0^R (see Section 5.2.3) before the merge, the tile will be automatically transformed into a dense representation as part of the merge process. Hence, the heterogeneous structure of the AT Matrix^{LD} changes in a more dynamic, incremental manner, and no complete rebuild and re-run of the partitioning process is required.

Insertions

There are different possible scenarios for the insertion of new rows or columns, or even row or column blocks. Appending insertions into the AT Matrix^{GD} can be simply incorporated into the global delta component. For AT Matrix^{LD}, any row or column insertion is delegated into multiple small row and column insertions on the affected matrix tiles. Regarding displacing insertions, both representations must update the affected tiles as well as the altered dimensions in their auxiliary index structures. Indeed, the general advantage of the AT Matrix is that dimension-altering changes only affect a limited number of tiles and the auxiliary data structures (the dimension trees and the

lookup tables), which is less expensive than the matrix-wide relabeling that is required in the triple table representation, or the massive rearrangements required in the CSR representation.

It would also be thinkable to split tiles affected by displacing insertions, similar to the behavior of a multidimensional tree structure such as the LAB tree (Zhang et al., 2011). For example, it could be that so many rows or columns are inserted to a particular tile that one of the maximum tile criteria is met (see Section 5.2.2). However, split strategies are currently not implemented in our prototype since we assume *few* insertions, but it might be part of future work.

Deletions

Single row deletions are carried out in a similar way as described for the mutable sparse matrix. As the tiles of both AT MATRIX^{LD} and AT MATRIX^{GD} are based on regular column tables, thus, they contain validity bitvectors maintained by the MVCC manager that mark the deleted elements. In contrast, deletions of whole columns, or blocks of columns are handled differently. The advantage of the AT MATRIX over CSR is that the deleted region is constrained to only the affected tiles. This reduces the memory region that has to be visited by the manipulation operation. We observed a similar behavior in Section 5.2.4: the response time for a matrix column read (GET) on a AT MATRIX is significantly lower compared to a column read on a plain CSR representation. Here, AT MATRIX^{LD} has the additional advantage over AT MATRIX^{GD} that only the local deltas have to be scanned instead of the global delta.

A special situation is present if the deletion range extends the range of one or more physical tiles. In this case, tiles that are fully covered by the deletion range can just be removed from the indirection layer, hence no expensive memory movements are required[4]. The range is then split into a part that is fully overlapping tiles, and parts that cover the remaining range fraction. In the example of Figure 6.4, the range is covering the tiles from row 1024 to 2047 completely, hence the corresponding tile-rows in the grid can be removed (marked in yellow). The remaining range fractions, i.e. from 1000 to 1023 and 2048 to 2100, are converted to *submatrix references* that are relative to the position of the affected matrix tiles (a, b, e, f, g). The procedure is similar to the tiled multiplication described in Section 5.3.2: the UNSET or DELETE method is called recursively on the affected local tiles using the submatrix references. Finally, the secondary structures including the position map and the row/column dimension index tree are updated (not required for UNSET).

Submatrix Flip

Flips of matrix regions can be handled very efficiently, especially if the affected regions are aligned or almost aligned to the tile bounds. Figure 6.5 sketches an example, where the submatrix $\boldsymbol{A}([0, 0],[1023, 1023])$ of the 3500×3500 matrix of Figure 5.3 is flipped with $\boldsymbol{A}([0, 1024], [1023, 2047])$. In this case, the submatrices match the matrix tiles, and the flip is ultimately performed by swapping the corresponding matrix tile pointers in the grid (here $a*$ is swapped with $b*$, as highlighted in Figure 6.5).

If the submatrix is not aligned to tile boundaries, the range is split into fully overlapping parts and the remaining fractions, similar to the aforementioned row-range-delete case. The fully covered

[4] We assume the tile memory de-allocation is internally handled by a memory pool allocator at negligible cost.

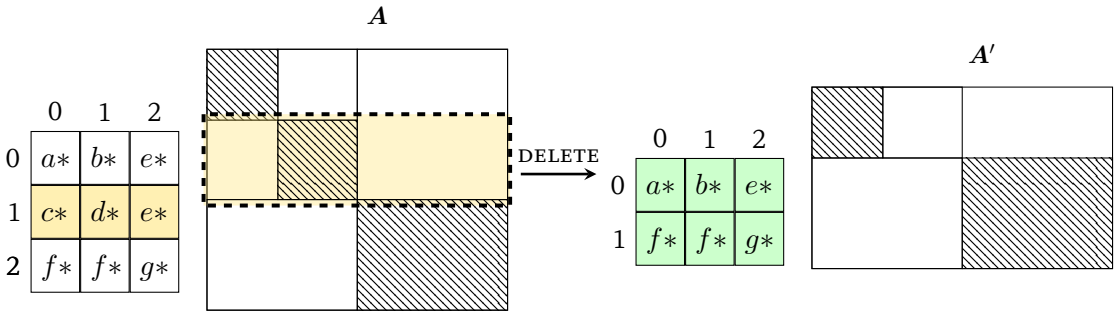


Figure 6.4: Implementation of a row block-delete operation in the 3500×3500 AT MATRIX of Figure 5.3: the row-block range $\boldsymbol{A}([1000, 0], [2100, 3499])$ is deleted.

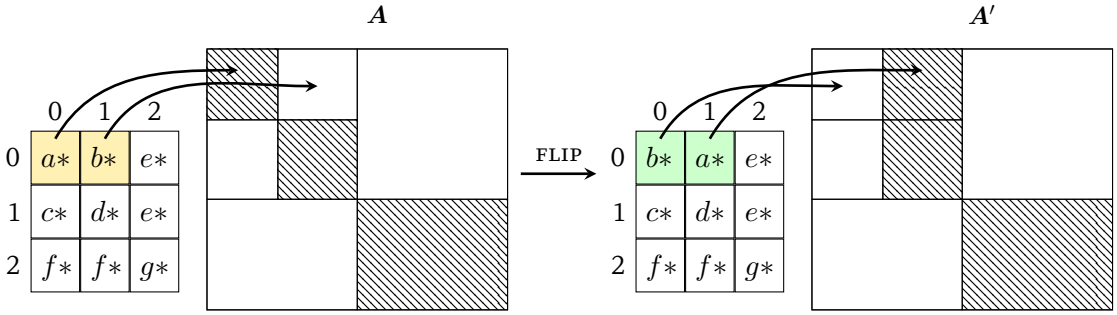


Figure 6.5: Implementation of a flip operation in the AT MATRIX of Figure 5.3: the submatrix $\boldsymbol{A}([0, 0], [1023, 1023])$ is flipped with $\boldsymbol{A}([0, 1024], [1023, 2047])$.

tiles are handled like in the aligned case by adjusting the grid structure. The remaining partial tile flips are implemented by consecutive delete and insert operations on the tile subregions, while maintaining a temporary matrix tile as intermediate swap buffer.

Maintaining the Density Map

Note that all manipulations, such as inserts or deletions, also affect the density map $\boldsymbol{\rho}$ (Fig. 5.2c). As explained in Section 5.2.3, the creation of the density map is piggybacked in the partitioning process of the AT MATRIX and cached for further reuse, e.g., in the ATMULT optimizer. It is obviously inefficient to rebuild the density map with every manipulation operation. Consider a single insert or deletion of a matrix element $(\boldsymbol{A})_{ij}$, which would increase/decrease the density of $(\boldsymbol{\rho})_{I,J}$ by a value of $\epsilon = \frac{1}{b_{\text{atomic}} \times b_{\text{atomic}}}$, where $(I, J) = (i,j) \mod b_{\text{atomic}}$. Thus,

$$(\boldsymbol{\rho})^{\text{new}}_{I,J} = (\boldsymbol{\rho})^{\text{old}}_{I,J} \pm \epsilon, \tag{6.6}$$

From the perspective of numerical stability, the summation in Equation 6.6 becomes inaccurate when ϵ is close to the floating point accuracy. Such a situation is likely when the density map is updated for every single insert, given that the block size is in the region of 10^4 for float values, and 10^7 for double values, respectively. Consequently, it is more efficient and numerically more robust

to batch the updates, such that on average $|\epsilon_{\text{batch}}| \gg |\epsilon|$. In order to batch the updates, the density-map is only updated with every delta merge. Hence, ρ is rebuilt globally after a delta merge of the AT Matrix^{GD}, whereas the affected regions of ρ are updated with every local delta merge of AT Matrix^{LD}. As a result, the entries of the density map of AT Matrix^{LD} are more recent.

In contrast to single element updates or deletions, changes that affect matrix regions spanning over multiple tiles are handled differently. Examples are row range deletions, column block deletions, or flips. In these cases, the density map is instantaneously updated, by deleting, swapping, or adjusting the corresponding entries in ρ.

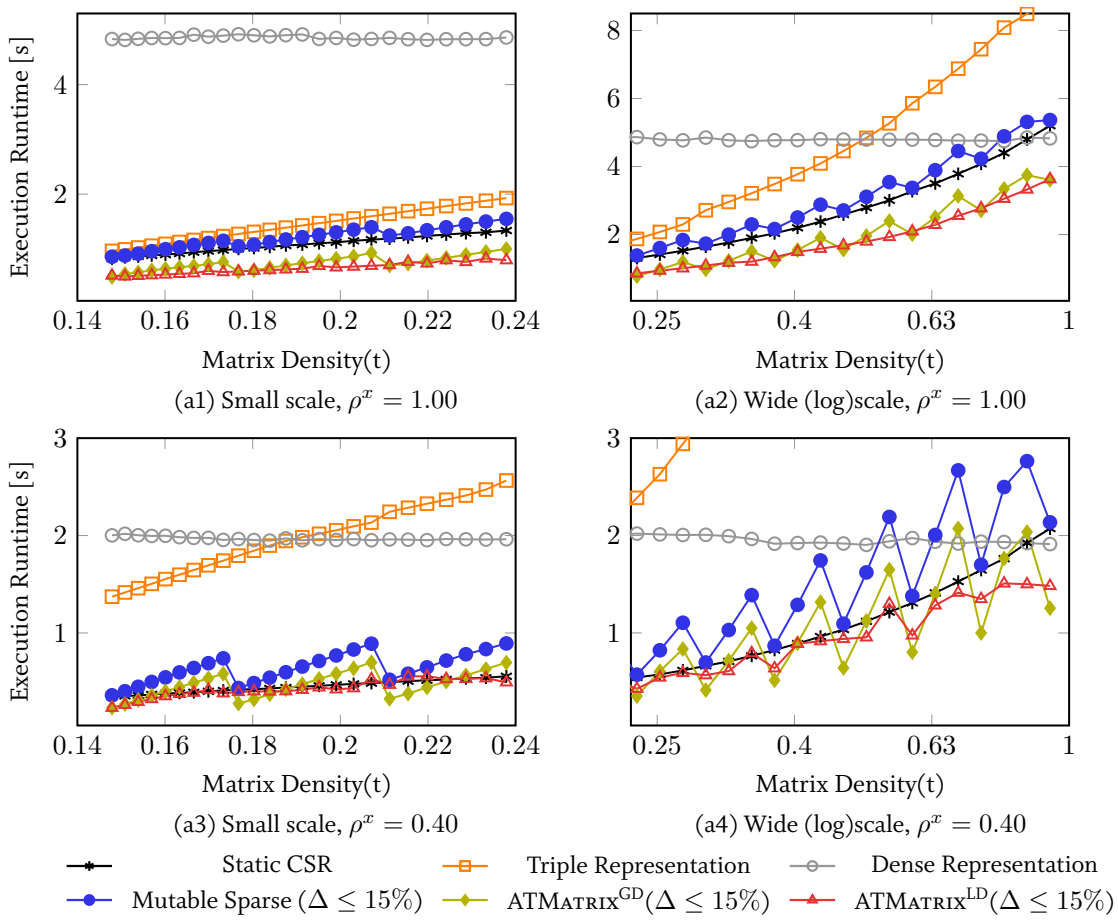
6.5 EVALUATION

In this section, we evaluate our mutable sparse matrix architecture as well as AT Matrix^{GD} and AT Matrix^{LD}, and compare it against alternative approaches based on the immutable CSR, triple, and dense matrix representations. Therefore, we first compare the static query execution performance for different fill states of the delta structure. Secondly, we evaluate how the query throughput performance on dynamic workloads improves against naive approaches. Finally, we compare the performance of row- and column block delete operations.

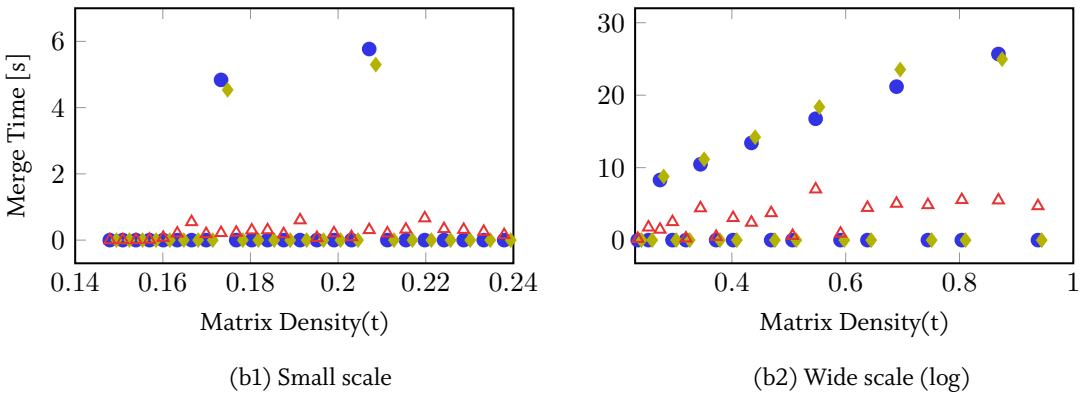
6.5.1 Experiment Setup

For the experiments we used an Intel E7-4870 system with a total of 1 TB RAM. As benchmark for a dynamic matrix processing workload, we took the implementation of our example application from nuclear physics described in 6.2.4. To evaluate our approaches in a more systematic manner, we simplified the implementation of Algorithm 16 and extracted two orthogonal components: firstly, an insert/read workload that refers to the loop iterations in line 6, and interleaves batch inserts with read queries. Second, we implemented a delete/read workload that corresponds to the truncation of states (lines 3-4), i.e., executes row/column deletes followed by consecutive read queries. The *read* workload is actually more complex than simple matrix row- or column reads. Instead, it is defined by a series of matrix-vector multiplications forming the core part of the Lanczos method (line 12). In fact, the outer product formulation of the matrix-vector multiplication (Alg. 17) reads either the full matrix or large parts of the matrix, depending on the density of the vector $\boldsymbol{x}$. Based on these workloads, we carried out the experiments presented in the following sections.

As for the data sets, we used several matrices that are listed in Table A.1. In particular, these include the Hamiltonian matrices from the nuclear physics simulation. Nonetheless, we found it useful to include further matrices from other domains, since the presented matrix manipulation primitives are of general nature, and applicable in various applications. It should be noted that in the context of this chapter, we study the update behavior of our data structures rather than the multi-core performance of matrix multiplications, which is covered by the previous chapter. For all write and read operations, the process and memory were pinned to one socket in order to avoid NUMA effects.



(a) Read-query response time performance for multiplying matrix Case1B (R1) with a vector, by using our mutable matrices vs. different alternatives. The x-axis denotes the matrix population density, which is increasing over time. The subplots differ in the density range, and in the $\boldsymbol{x}$-vector density ρ^x.



(b) Merge behavior of mutable sparse, AT Matrix^{GD} and AT Matrix^{LD} for the workload above.

Figure 6.6: Temporal behavior of read-query response times and merge times of an insert/read workload.

6.5.2 Read-Query Response Times

In this experiment, we measured the performance of the isolated read part of the workload, while constantly increasing the matrix non-zero population. Since this experiment is intended for evaluating the impact of the delta structure on the query performance, we subsequently inserted non-zero elements into the initially sparse matrix. Furthermore, to get an unbiased perception of the write performance, *randomly* selected matrix elements were written (unchecked set) rather than just appending full matrix columns or rows. In order to avoid double-setting of matrix elements, we assert per construction that each element is set at most once.

We executed the workload using our mutable sparse matrix representation (Fig. 6.2b), as well as the AT MATRIX^{GD} (Fig. 6.3a) and AT MATRIX^{LD} (Fig. 6.3b) structures. Moreover, we included the following alternatives:

- **Static CSR representation (baseline)**: We take the immutable CSR representation as the baseline for the static execution of the read queries. Note that the read-query time does not include the time required for constructing the CSR[5] .

- **Triple Representation**: The triple table is updatable in append-only mode, and serves as a straightforward approach for database-integrated linear algebra on mutable sparse matrices.

- **Dense Representation**: For illustrational purposes, the dense representation is included in the measurement of the relatively dense matrix Case1B (R1). However, it is omitted in most other measurements, since the dense representation does not scale for large, sparse matrices, and its memory consumption is order of magnitudes higher than that of the sparse representations.

Figure 6.6a shows the read-query performance comparison for the matrix Case1B (R1). Since we are constantly inserting matrix element in-between consecutive query executions, the matrix population density ρ of the matrix $\boldsymbol{A}$ increases over time (t). The density(t) is denoted along the x-axis of the plots, and reaches up to full occupation with non-zero elements for matrix R1 ($\rho \rightarrow 1.0$). The delta merge threshold was set for all mutable delta structures to $\Delta_T = 15\%$. Additionally, we included two different $\boldsymbol{x}$ vector densities (40% and full), since ρ^x has an influence on the sparse version of the outer product implementation of GEVM (Alg. 7/8).

The query response time increases with the matrix density for all representations except the dense matrix. The break-event point, at which the dense representation becomes more efficient than CSR, is reached at a density of about 0.85. In Figures 6.6a1 and 6.6a2, we see that our adaptive matrix structures AT MATRIX^{GD} and AT MATRIX^{LD} outperform all other approaches, followed by the static CSR performance (Baseline), mutable sparse matrix, triple representation, and the dense representation (for $\rho < 0.85$). Moreover, we observe that the vector density ρ^x has a considerable influence on the multiplication time. A lower $\boldsymbol{x}$-density decreases the runtime of all multiplication approaches except for the triple, which is insensitive to ρ^x. When the vector is full ($\rho^x = 1.00$, Figures 6.6a1 and 6.6a2), the triple performance approaches that of the indexed CSR structure, since every matrix row is visited in the multiplication. In contrast, when the vector is sparse (Figs.6.6a3

[5]Instead, the construction time is taken into account in the second experiment (throughput measurements).

and 6.6a4), the other representations (except the triple) exploit the vector sparsity due to their row-indexed structure, which leads to a reduced response time. Consequently, the separation between the triple curve and the remaining lines becomes wider.

Both curves of AT MATRIX^{GD} and of the mutable sparse matrix exhibit a saw-tooth shape, which is clearly visible in Figures 6.6a3 and 6.6a4. As expected, the shape reflects the behavior of a global delta structure under a stream of dynamically inserted non-zero elements. This behavior is equivalent to the deterioration of search costs on an organized file data structure that is constantly updated and periodically reorganized, as shown by Shneiderman (1973). All elements inserted into the mutable sparse matrix and AT MATRIX^{GD} are at first accommodated by their delta triple representation. Thus, the delta size continuously increases until the delta occupation threshold Δ_T is reached, which we set to 15% in this experiment (we address the determination of the delta size in the following section). As soon as Δ_T is reached the delta part is merged into the main structure, and the delta occupation shifts back to zero. Hence, the execution time of the mutable sparse matrix and AT MATRIX^{GD} is effectively a composition of the delta- and main execution time, i.e.:

$$T_{\text{Main/Delta}} = T_{\text{Main}}((1-\Delta)\rho) + T_{\text{Delta}}(\Delta\rho), \tag{6.7}$$

where Δ is the fraction of elements that is currently accommodated by the delta.

As mentioned before, the construction overhead for the mutable CSR and AT MATRIX representations after each merge is not included in the measurement of Figure 6.6a. Instead, we added two additional plots in Figures 6.6b, which show the separate merge runtimes of the mutable sparse, AT MATRIX^{GD}, and AT MATRIX^{LD} representations. As expected, the merge costs of the mutable sparse matrix and AT MATRIX^{GD} are concentrated at the points where the delta reaches the threshold Δ_T and is merged. In contrast, the merge costs of the AT MATRIX^{LD} are rather uniformly distributed over time. Consequently, we see equidistant peaks in Figures 6.7a and 6.7b for the mutable sparse matrix and AT MATRIX^{GD}, which both have a similar merge time. Note that the peak height increases with increasing density (Fig. 6.7b) due to the larger number of non-zero elements that have to be ordered and rearranged. Interestingly, the merge time of AT MATRIX^{LD} seems to have a more robust merge behavior, as the merge times remain rather stable.

6.5.3 Delta Size Experiment

In the measurements above we used the delta threshold $\Delta_T = 0.15$, which means that the delta is merged into the main structure when its size reaches a fraction of 15% of all non-zero elements in the matrix. In this measurement, we evaluate the dependency of the query throughput on the delta size in order to empirically find an appropriate threshold Δ_T, and to confirm or reject the model of Shneiderman (1973). Therefore, we carried out an experiment using the same workload as in the first experiment, consisting of batch-wise non-zero element insertions interleaved by multiple subsequent matrix-vector multiplications. In contrast to the first experiment, we now measured the total read *and* write time, i.e., including both the execution time of multiplications plus the averaged time spent in delta merges. Figure 6.7 shows the average time per read-write query for two different matrices. As expected, all curves exhibit a minimum, which we locate in the range $0.10 \leq \Delta_T \leq 0.15$.

The curves in Figures 6.7a and 6.7b resemble the $\frac{\alpha}{t} + \beta t$ behavior of Equation 6.1, which is indeed a hint that the model of Shneiderman (1973) is quite a good fit for our measurements. We

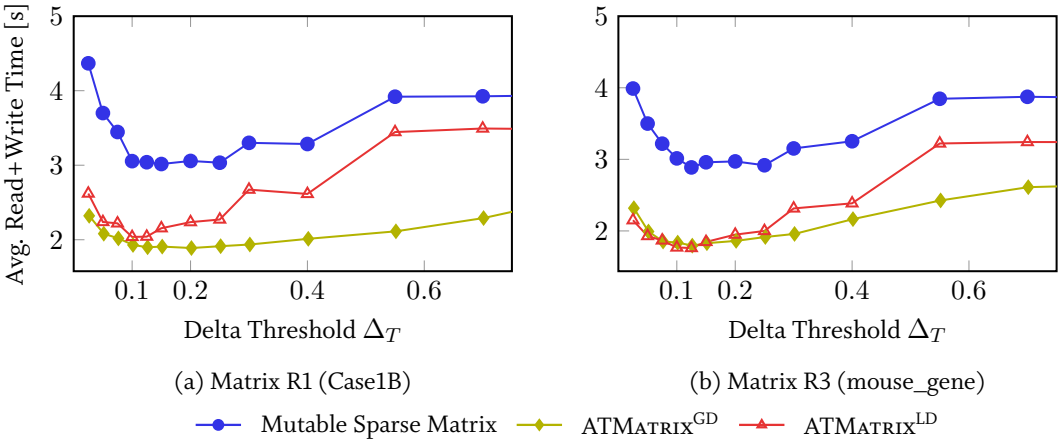


Figure 6.7: Average response time of read+insert workload behavior of mutable sparse, AT MATRIX^{GD} and AT MATRIX^{LD} for the workload above. $\rho_x = 0.60$

chose a balanced but rather write-intensive workload, by inserting elements in batches of 1.5% of the current matrix size after each read query. From the curve minimum in Figure 6.7 we infer that a slightly larger delta has little impact on the performance, whereas a too small Δ_T leads quickly to an increase in the response time. Reversely, a too large delta has a negative influence on read-intensive workloads. For the following experiments we chose $\Delta_T = 0.13$.

6.5.4 Throughput Measurement

In this experiment, we measured the total query throughput of the dynamic read/write workload. Similar to the previous measurements we interleaved batch insertions (=write queries) with multiple matrix-vector multiplications (=read queries), but in contrast to the previous workload, we have varied the ratio of read queries N_{read} over write queries N_{write}. To vary the read/write ratio, we step-wise increased the fraction of write queries in the workload according to the formula:

$$N_{\text{queries}} = N_{\text{read}} + N_{\text{write}} = (1 + \alpha)N_{\text{read}}. \tag{6.8}$$

The parameter α is the average fraction of write queries per read query $\alpha = N_{\text{write}}/N_{\text{read}}$. It takes values from $\alpha = 0.02$, which corresponds to one write every 50 read queries, up to $\alpha = 1$, meaning that each read query is followed by a write query.

We run this experiment using six different dynamic matrix strategies. Naturally, we included all mutable matrix data structures including the triple representation, the mutable sparse matrix, and the AT MATRIX^{GD} and AT MATRIX^{LD}. However, we added two alternative ways to run dynamic matrix workloads based on the immutable CSR. These are:

- **Copy-sort CSR**: It is fair to compare against a naive approach, which includes copying and the construction of a static CSR matrix prior to each read query.

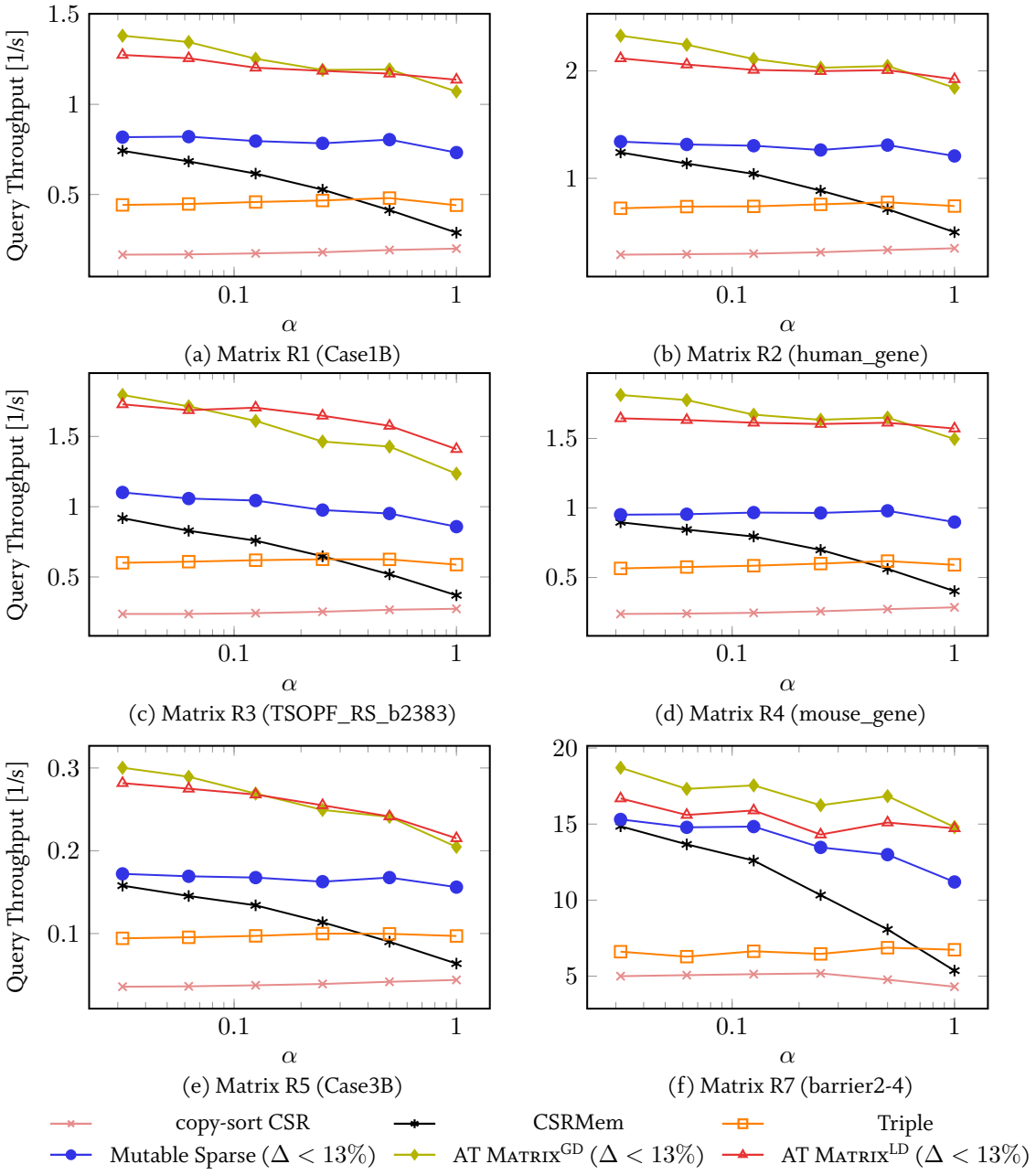
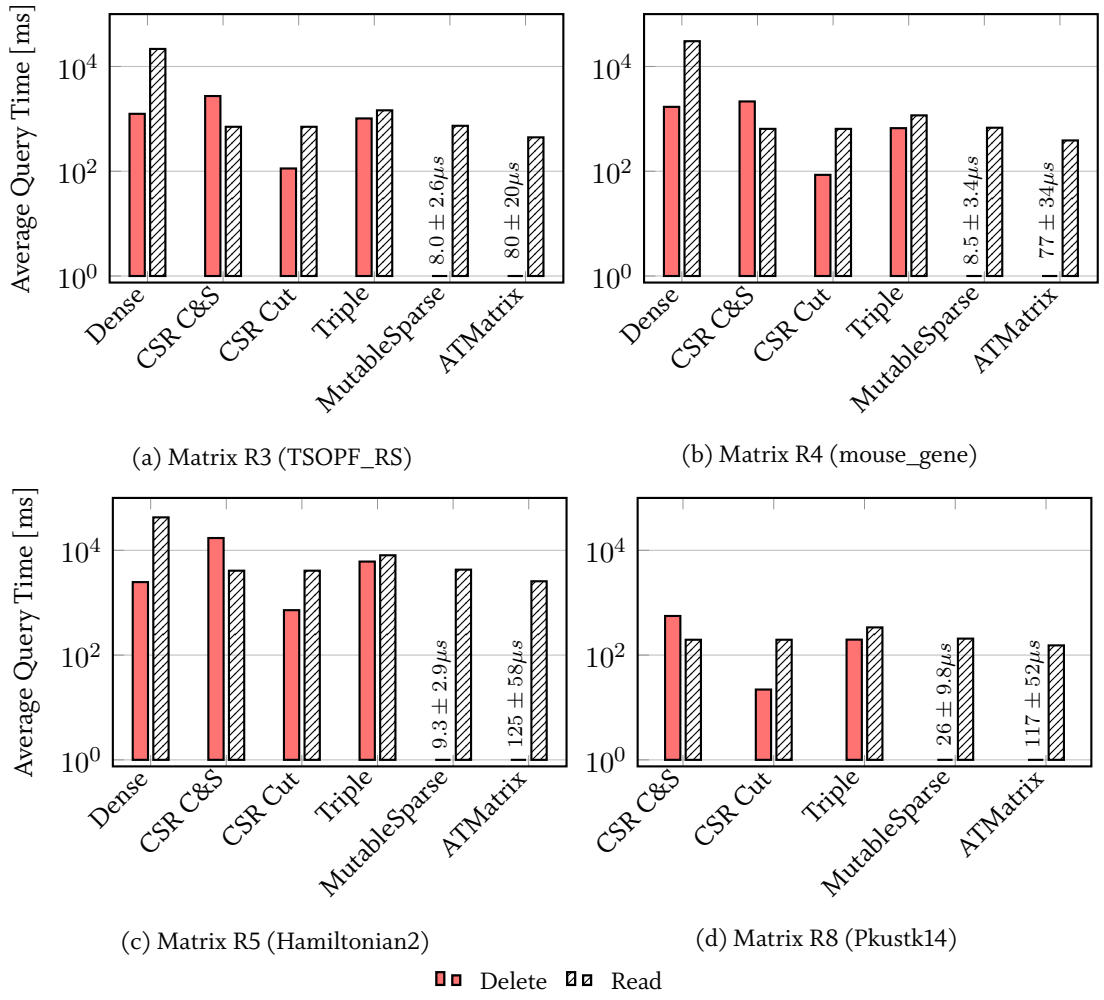


Figure 6.8: Total read/write query throughput of multiple approaches on different matrices listed in Table A.1. We used the mixed workload that interleaves $N = 50$ read queries (each executes GEVM 10 times) with αN write queries (each inserts $0.015 N_{nz}$ elements.)



(a) Matrix R3 (TSOPF_RS)

(b) Matrix R4 (mouse_gene)

(c) Matrix R5 (Hamiltonian2)

(d) Matrix R8 (Pkustk14)

Figure 6.9: Average duration of deletions of k rows, and the following read query, using different matrices of Table A.1 with $k = 0.2m$.

- **Cached CSR**: In this approach, a cached CSR version of the sparse matrix is kept in memory. The cached version is invalidated when the matrix is manipulated, and will not be rebuilt before a read query is requested (lazy reconstruction.)

The copy-sort CSR represents a common way in scientific environments, where data is often transferred and brought in shape for a third party system or library. As we have observed in Chapter 2.7 (Tab. 2.1), mutability is not supported by the majority of libraries and systems. Note that our copy-sort CSR approach does not include data transfers to an external process, which would further decrease its performance. Figure 6.8 presents the results of the throughput measurements using several matrices of Table A.1. As expected, the triple representation and the copy-sort CSR approach are more or less insensitive to the composition of read- and write queries. In contrast, the cached CSR approach shows a clear dependency on the ratio parameter α. The behavior can be

explained as follows: for low α, only a few rebuilds of the CSR structure are required, whereas for $\alpha \to 1$, a rebuild is required after each query (since it includes a write).

Similar to our first experiment (Fig. 6.6a), we observe that our tiled matrix types show a superior read- and write performance over the mutable sparse matrix. It is notable that the relative performance between AT Matrix^{GD} and AT Matrix^{LD} changes over the range of α: for a low and medium number of write queries (α < 1), AT Matrix^{GD} seems to be the better strategy. However, for a high number of updates $\alpha \to 1$ AT Matrix^{LD} shows a better behavior.

6.5.5 Deletions of Rows and Columns

In this part, we implemented a delete/read workload that is close to the truncation of states in the nuclear physics analysis scenario sketched in Algorithm 16 (lines 3-4). The original scenario comprises row- and column deletions. In order to systematically evaluate the difference between row- and column manipulations due to the effects of linearization (see section 3.4.4), we measured the row- and column deletion separately. Therefore, we run a number of write queries that each delete a variable number of rows (columns), and each of these write queries is followed by a read query. Next to the alternatives from the previous experiment, we added a "CSR Cut" approach, which reflects a partial move-on-delete strategy by utilizing the row pointer index: whenever a certain row range $[i_{from}, i_{to}]$ is deleted, the affected contiguous memory region between position RowPtr$[i_{from}]$ to position RowPtr$[i_{to}+1]-1$ of the $\boldsymbol{A}$.Val and $\boldsymbol{A}$.Col columns (see CSR, Figure 3.4) are removed, and the adjacent contiguous memory region starting from RowPtr$[i_{to}+1]$ is shifted to fill up the gap.

Figure 6.9 shows the average duration of a k-rows delete operation ($k = 0.2m$) followed by a read query, using different matrices of Table A.1. The fraction of $k = 0.2m$ means that each delete query removes roughly 20% of the matrix rows. For all matrices the dense matrix representation is inferior regarding both deletion and query time compared to the remaining representations (for illustrational purposes we omit the bar for the dense representation for matrix R8 (Pkustk14), since it is more than three orders of magnitude slower than the others.) Besides the dense representation, the manipulation costs of the copy&sort approach are high as expected, since it involves a complete matrix copy and reconstruction of the CSR representation. A similar behavior is exhibited by the triple representation, which requires a full scan and rebuild due to its unordered structure. The CSR Cut strategy outperforms the triple representation by a factor of 2-3×. In fact, we observe that the deletion cost for the mutable sparse and AT Matrix is close to zero. This can be explained by the row pointer validity vector (II-VC in Figure 6.2b), which is used by the mutable sparse matrix and AT Matrix to efficiently tag the deleted rows. The penalty due to using the validity-aware algorithm is low against the plain CSR read performance. Furthermore, the AT Matrix-based approach outperforms the CSR implementation due to the tiling. Note that we did not distinguish in the measurement between AT Matrix^{GD} and AT Matrix^{LD}. Since no elements are inserted in the delete/read-workload, the deltas are empty and both behave equivalently.

A slightly different behavior is revealed by the column delete/read-measurements shown in Figure 6.10. Firstly, the deletion time for CSR Cut and the triple representation is identical. This is because unlike matrix rows the non-zero elements of a matrix *column* are not contained in a contiguous region in the row-centric CSR representation, but rather spread across the complete memory section that accommodates the matrix. This is also the reason why the deletions on the mutable

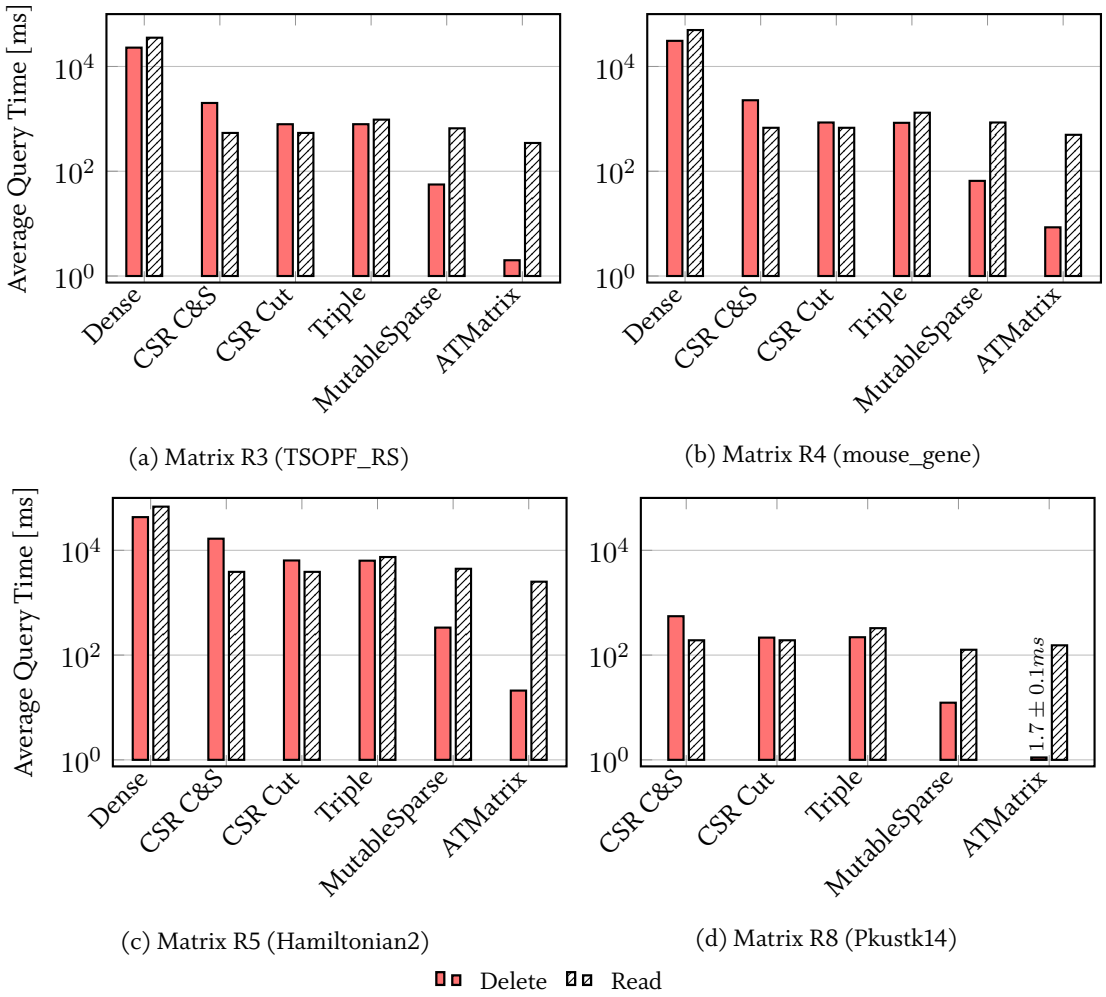


Figure 6.10: Average duration of deletions of k columns, and the following read query, using different matrices of Table A.1 with $k = 0.2n$.

sparse and AT Matrix representation have non-vanishing costs, which are, however, still orders of magnitude lower compared to the alternative approaches. In contrast to the row deletions (Fig. 6.9), Figure 6.10 reveals that column deletions on the AT Matrix are about an order of magnitude faster than on the mutable sparse representation. This is explainable by the following two effects: firstly, the 2D partitioning obtained by the adaptive tiling reduces the affected memory sections to the local matrix tiles. Secondly, by exploiting the abstraction layer of AT Matrix, multiple tile-spanning deletions are carried out particularly efficiently by removing the affected tiles from the grid and index structures.

6.6 RELATED WORK

The work of this chapter partially overlaps with multiple research areas, which are batch updates of indexed data structures in general, and particularly, mutability and multidimensional manipulations of matrix data structures.

6.6.1 Update Strategies in Databases

Batching updates in order to avoid frequent page writes is common in DBMS, and of prime importance for any system based on a hierarchical storage. Besides the motivation adhering from the I/O-optimization of disk-based systems, another objective of secondary buffers is to exploit the different read- and write-characteristics of data structures.

In the early decades of database system research, Shneiderman (1973) and Lohman and Muckstadt (1977) investigated optimal strategies for batching updates from a theoretical perspective, independent from the underlying data structure. In particular the work of Shneiderman (1973) is a good match to our observations, and by applying his model to our scenario we can create a connection between the insert-workload, and the delta size Δ_T. This result is tempting to conduct further research about workload-driven delta structures, which adapt their size dynamically based on workload statistics. Lohman and Muckstadt (1977) presented a more comprehensive model for optimal batch operations that includes non-constant update rates and Poisson-distributed failures, and derived the model from Shneiderman (1973) as a special case of their model.

O'Neil et al. (1996) presented the Log-Structured Merge-Tree (LSM) to tackle the problem of indexing fast-growing, transactional data, by batching and deferring index changes. Moreover, Jagadish et al. (1997) proposed a stepped-merge-insertion, which lazily stores arriving records in sorted runs, which are later merged into a final B^+-tree. Both ideas are similar to our global delta structure, although applied in a different context and with different main data structures.

The concept of organizing the physical data representation into a separated read-, and a write-optimized part is nowadays fundamental to many systems that are geared to serve both online analytical processing (OLAP), as well as online transactional processing (OLTP) workloads. For instance, the in-memory column-store DBMS SAP HANA (Färber et al., 2012; Sikka et al., 2012) splits the physical representation of each table column into a static main, and a short cascade of delta structures, which are periodically merged into the main storage. Each of the merge steps includes a reorganization phase, the purpose of which is to improve the compression ratio by reordering table rows. Moreover, they employ a sophisticated multiversion concurrency control mechanism (Sikka et al., 2012).

The latter work is orthogonal to ours, since our DBMS-embedded Lapeg uses the MVCC columns of an in-memory DBMS as storage basis for the matrix structures. Hence, we automatically benefit from the transactional features provided by the DBMS, and are able to leverage the updatability features for matrices on a higher level. The conceptual focus of this thesis, however, is the ability to dynamically, bulk-wise manipulate read-optimized matrix representations rather than revisiting the particularities of a multiversioned, transactional system.

6.6.2 Mutable Sparse Matrices

Manipulations of a sparse matrix, such as insertion of new elements, is not foreseen in common algebra systems. For instance, MATLAB stores sparse matrices in a CSC layout (Gilbert et al., 1992). In line with the discussion presented in this chapter, Gilbert et al. (1992) admit that "inserting or removing a non-zero may require extensive data movement", since the CSC structure is like the CSR per se immutable. However, we showed that manipulations on CSR or CSC matrices can be integrated in a non-dissipative way by leveraging a main-delta architecture.

As outlined before, CSR is a popular storage representation for graphs, since it can be considered an efficient adjacency list implementation. For instance, it serves as physical back-end in the boost graph library (Siek et al., 2002). Consequently, CSR is often used for highly efficient graph algorithm implementations (e.g., in Pearce et al., 2010; Hong et al., 2012). In the field of evolving and dynamic data, there are many uprising, specialized graph systems like the disk-based GraphChi (Kyrola et al., 2012) or Llama (Macko et al., 2015) that strive to combine efficient analysis on large data graphs with dynamic data structures.

In particular, (Kyrola et al., 2012) have a *shard*-partitioned data structure, where each shard contains all in-edges of vertices in a certain interval *p*, sorted by their source vertices. In some sense, each of their shards corresponds to a *partial CSR* for the subgraph that only contains the in-edges of the respective vertices in interval *p*, which corresponds to a column-partitioning of the adjacency matrix. By the target vertex partitioning they establish some locality in both edge directions (in- and out-going), since a plain CSR only stores either the outgoing or the incident edges. Moreover, they allow adding edges "to the graph efficiently by implementing a simplified version of I/O efficient buffer trees." In particular, adding edges corresponds to the insertion of non-zero elements into the adjacency matrix. Therefore, Kyrola et al. (2012) attach *n* buffers to each partial CSR, which are indexed by the source vertex interval within their partition ($p, i \in n$). If the buffer exceeds a predefined limit, they will merge them with the shards/partial CSR. Similar to our approach, removed edges are flagged, and ignored in the computation.

A more recent dynamic graph system based on CSR is LLAMA (Macko et al., 2015), which was published a year after our main-delta-based mutable sparse matrix representation was presented (Kernert et al., 2014). The dissertation of Macko (2014) provides a quite comprehensive oversight of different ways to make read-optimized CSR structures mutable. According to his thesis there are two common ways: one is to equip CSR with a delta map as done in GraphChi. The other variant is called log-based CSR: it copies complete adjacency lists (corresponding to matrix rows) whenever a single element of it is updated, and uses a pre-allocated buffer at the bottom of the data structure. LLAMA implements a multi-snapshot approach by holding one vertex indirection table (corresponding to the row pointer) per snapshot. When an edge (or non-zero element) is inserted, it is entered in the new snapshots' own adjacency list (corresponding to the *Col* column), and the respective vertex entry is updated in the vertex table (row pointer). Moreover, the incremental adjacency lists contain a next pointer per snapshot, which redirects to remaining edges of a particular vertex in an older snapshot. Similar to our architecture based on MVCC validity bitvectors, deletions are implemented in LLAMA by using a deletion vector. However, LLAMA "does not currently provide strong transactional guarantees other than ensuring that new data is durable as soon as snapshot creation completes" (Macko et al., 2015).

The implementation of batch updates on a tree-partitioned matrix layout is discussed in the LAB-tree (Zhang et al., 2011). The heterogeneous LAB-tree data structure stores sparse and dense submatrix leafs on disk, and maintains an update buffer per leaf. Inserts to the LAB-tree are performed in batches, and as mentioned in Section 5.6.3, may lead to a conversion from sparse key-value pairs to a partial dense array chunk if a leaf's density exceeds a threshold. However, unlike our delta structure their update buffer has a limited size K, and needs to be flushed (=merged) to disk pages when it is full. This problem is similar but not equivalent to cache replacement strategies, since the buffer space devoted to a page changes as the number of requests increases. In particular, (Zhang et al., 2011) discuss different buffer flushing strategies, and evaluate them using different matrix insert patterns (e.g., sequential row-major element inserts vs. random element inserts.) To summarize, their work is rather orthogonal to ours, since flushing pages to disk does not play a role in our in-memory setup. Moreover, they only consider the insert-stream performance rather than mixed workloads on a dynamic matrix, and deletions as well as matrix range-wide operations are not covered.

In contrast to the systems above, the charm of our mutable sparse matrix is the minimization of implementation overhead by utilizing two independent and self-contained data structures (CSR and Triple Table). Hence, the implementation complexity is significantly reduced. Indeed, several adoptions like the paging in Llama or the shard-partitioning in GraphChi were particularly tweaked for disk-based, external-memory machines, and are either inappropriate or expendable regarding the in-memory machines this work is targeting. Moreover, we emphasize that the focus of this thesis is not only to support single element updates, but also enable bulk-wise matrix manipulations, which are efficiently implemented by the abstraction layer of the AT Matrix.

6.7 SUMMARY

This chapter presented the integration of mutability with the data structures of Lapeg. From the logical point of view, we presented a representative selection of abstract API commands to manipulate matrices in different two-dimensional shapes, and showed how this API is leveraged to implement a dynamic matrix application from nuclear physics.

Furthermore, we surveyed the update behavior of the different physical matrix data structures of Chapter 3, and presented a mutable variant of the CSR sparse matrix based on a main-delta approach. By using the version-controlled columns of the in-memory DBMS, we can leverage the column-integrated main-delta mechanism "for free" to implement our updatable matrix structure on top. Thus, our matrix representation automatically benefits from the transactional features provided by the MVCC manager as a side effect.

We adopted the main-delta approach to our AT Matrix, and proposed two different integration strategies: AT Matrix^{GD} (one global delta), and AT Matrix^{LD} (multiple local deltas). We evaluated different approaches using two separate dynamic workloads that are extracted from the nuclear physics implementation, comprising an insert/read-, and a delete/read-workload, where the read queries are a number of consecutive matrix-vector multiplications. Compared to alternative approaches, our mutable matrix representations are up to 2x more efficient in terms of the total query throughput of a mixed read/write workload. In particular, the AT Matrix^{GD} and AT Matrix^{LD} structures exhibit the highest throughput, with a slight advantage of AT Matrix^{GD} for

read-intensive workloads. Conversely, AT Matrix^{LD} outperforms AT Matrix^{GD} for write-intensive workloads, and moreover, shows a more robust merge behavior by the distribution into smaller, local delta merges. The fact that row and column range-deletions are large handled by the altering the auxiliary structures and only a few tile-local manipulations reveals a fundamental improvement of using a AT Matrix representation over the alternative approaches.

To put it in a nutshell, we showed that we are able to introduce mutability of matrices almost without sacrificing performance of read queries. By separating indexed sparse matrix structures into a read-optimized main, and a write-optimized delta part, we again used a technique inspired from database technology in order to improve the performance of dynamic matrix workloads of compelling scientific applications. Finally, we showed that many manipulation operations can be seamlessly implemented using our novel AT Matrix data structure via cost-efficient modifications of its secondary data structures in the grid abstraction layer.

7 CONCLUSION AND OUTLOOK

In times of ever increasing data amounts being collected from various data sources such as scientific experiments, business transactions, or social media, there is a vital need for powerful tools for managing and analyzing data (Gray, 2007). Linear algebra operations are part of nearly every application in advanced analytics, machine learning, and of various science domains. Nevertheless, until today many data analysts and scientists have been working with single-workstation programs, or hand-crafted solutions that are incapable to keep up with the vast increase in data volume and size. Although there are scalable platforms for machine learning, they often provide read-only access, and depend on specialized data structures. In contrast, based on the application examples presented in this thesis, we pointed out that there is an increasing market for scalable linear algebra platforms that include data management capabilities.

For data analysis, one possibility is to move the data to the scientists, but the other possibility is to move the scientists' query to the data (Hey et al., 2009). The rise of main memory database systems has driven the trend towards online analytical platforms, which target at serving complex analytical queries of any type. However, applications that are built upon linear algebra is a domain that has not been served well by existing DBMS (Stonebraker et al., 2013b). Therefore, this thesis presented a deep integration of linear algebra operations into an in-memory column-oriented DBMS, in the form of a DBMS-integrated linear algebra processing engine (Lapeg).

As the first pillar of this work, it is shown that the column-oriented storage layer of an in-memory DBMS yields an easy adoption for efficient sparse matrix data types. But in the course of this thesis, we observed that processing linear algebra significantly benefits from different techniques inspired from database technology. In a novel way, we applied several optimizations on matrix processing workflows, including expression rewrites, separation of logical data type from physical implementation, data partitioning, multidimensional indexing, et cetera. In particular, we summarize the core achievements of this thesis:

Linear algebra matters: easy-to-use, optimized and dynamic. We picked three of the application scenarios mentioned in the introduction, and derived the following requirements they imply for a linear algebra platform: deep integration of matrix data types, expression optimization, transparency, scalability & performance, and data manipulation. The survey of several existing commercial, open-source and research libraries and systems showed that none of the related approaches is serving all requirements equally well. In fact, these requirements are addressed by our contributions.

Deep integration with data management systems. We described how state-of-the-art matrix structures are seamlessly mapped onto the columnar storage layer, and how matrices can be exposed as relational tables to the system. From a system perspective, the ability to shape the primary data structures into dense arrays and the CSR representation in the database engine opens the door to employ efficient numerical C++ libraries (e.g., the BLAS library,) as computational kernel. The deep integration of matrices as first class citizens and linear algebra primitives as top-level operations dispenses the need of costly data transfers to third party software, while at the same time users are offered the powerful native data management tools of the DBMS.

Database-inspired methods to optimize expression execution. We presented a novel approach to optimize linear algebra expressions with sparse matrices. In particular, we identified the multiplication of two or more matrices as the most costly and computationally intensive operation in linear algebra expressions. Our optimizer (SPMACHO) draws the analogy from sparse matrix multiplications to cardinality estimation and join order enumeration of the relational universe. Together with our efficient intermediate matrix density estimator (SPPRODEST) that predicts the approximate nonzero pattern of intermediate results, our optimized execution reduces the costs for time-intensive multiplication expressions significantly.

Matrix processing benefits from density-aware data types. We presented the fully adaptive, optimized matrix data-type AT MATRIX. It is created by a topology-aware clustering process, and autonomously partitions the matrix into adaptive, heterogeneous tiles. The distinction between dense and sparse matrices on tile-level is based on algorithm characteristics and improves both space-efficiency and matrix processing performance. Most importantly, AT MATRIX abstracts the concrete physical implementation into the data type, and thus, relieves users from the burden of choosing the appropriate data representation for their matrix data.

Scalability and resource-awareness by adaptive tiling. We presented a multiplication operator ATMULT that multiplies AT MATRICES and outperforms state-of-the-art sparse and dense matrix multiplication approaches by up to 10×. In particular, ATMULT benefits from cache locality and runtime optimizations by exploiting different multiplication kernels and dynamic tile conversions. Moreover, the shared-memory parallel ATMULT uses a two-leveled data- and task parallel approach in order to saturate the sockets of a multi-core main-memory platform. We prevent the overutilization of parallel resources by using a DBMS-integrated task scheduler that manages all resources system-wide. In addition, we showed a resource-aware concept for managing ATMULT's memory consumption to avoid out-of-memory situations.

Matrix manipulations: fast and efficient. We showed how our universal matrix manipulation API is used to implement dynamic matrix workflows. In particular, the API is centered around insertions and deletions of 2D matrix regions rather than single elements. We presented a mutable sparse matrix type that enables fast inserts and efficient reads by a composition of a read-optimized main, and a write-optimized delta part. Since our LAPEG is embedded into an in-memory DBMS, this composition can be obtained "out of the box" by leveraging the column-integrated main-delta mechanism in the column-oriented storage layer. In addition, users of LAPEG matrices automatically benefit from the transactionality features of the DBMS, which is a considerable advantage over other

numerical systems. We further elaborated the characteristics of insertions and deletions for the novel AT Matrix data type. Finally, we were able to speed up dynamic matrix workloads by a factor of 2-3 using our mutable AT Matrix^{GD} and AT Matrix^{GD} types.

We motivated the requirement for linear algebra and matrix processing functionality by citing three examples that are cooperative projects or applications in the environment of the SAP HANA database platform. With the introduction of Lapeg we have set the way to support these advanced analytical applications *natively* inside a main-memory DBMS. As a result, we believe that our work contributes to the ambition of modern analytical data management platforms to bring complex analysis queries closer to the data.

Nonetheless, the full establishment of a linear algebra runtime environment requires more functionality than could have been covered within this work. In fact, the presented matrix data structures and algorithms form the foundation of the in-DBMS linear algebra processing engine. We picked several aspects of the logical and physical layer that we examined in greater detail, but many of which could still be extended by including additional functionality. In the course of this work, we have found many aspects that are worth investigating further in the future. In particular, these include:

Additional operations. There are some important linear algebra operations that were not discussed in the scope of this thesis. For example solvers for linear equations, or matrix inversions. We stated that multiplications are of high significance, since they are often part of *iterative solvers*, and are therein the most expensive operations. However, it could be worthwhile to elaborate how *direct* solvers (e.g. Cholesky- or LU-decompositions) on large, sparse matrices could benefit from our heterogeneous tiled matrix storage design.

Hypersparse structures and kernels. So far our matrix structures mostly rely on dense and CSR or triple sparse matrix representations, which form the physical tiles of the AT Matrix. However, unlike some examples in this work (e.g. the Hamiltonian matrices of the nuclear physics use case,) many matrices of the real world are hypersparse. Consequently, even a generous tiling would generate submatrices with nearly or completely empty rows and columns. Therefore, it would be worthwhile to add a hypersparse matrix (tile) representation, e.g. by using a hash map, or doubly compressed sparse row (DCSR) or column formats (DCSC, as proposed in Buluç and Gilbert, 2012). As a matter of fact, the inclusion of additional structures necessitates an implementation through the full stack of the Lapeg, i.e. multiplication kernels, cost functions, et cetera.

Comprehensive expression optimization. The expression optimization chapter of this thesis is focused on the optimization of matrix chain multiplications. However, one could think of further situations that can be optimized. For example, we observed performance improvements by using pipelining for the summations of multiple sparse matrices rather than materializing intermediate results. Moreover, as proposed by Boehm et al. (2014a), some algebraic expressions can be rewritten such that expensive operations can be avoided, e.g. the diagonalization of a matrix.

Preprocessing sparse matrices. Our heterogeneous adaptive tile matrix data representation (AT Matrix) benefits from a clear distinction between dense- and sparse matrix regions. In related

work about graph partitioning (e.g., in Karypis, 2011; Sanders and Schulz, 2013) sparse adjacency matrices are permuted and reordered such that non-zero elements are clustered together, usually to reduce the communication overhead in distributed partitioning. These methods could also be applied as a preprocessing step in our AT Matrix partitioning to improve the tile substructure.

A
TABLES

Table A.1: Sparse matrices of different dimensions and population densities. The $\rho = N_{nz}/(n \times n)$ value denotes the population density of each matrix. All matrices are square ($n \times n$.) The binary size is given for the sparse triple/coordinate format (COO) consisting of $\langle int, int, double \rangle$.

Number	Name	Matrix Domain	Dimensions	N_{nz}	ρ [%]	Bin. Size	Result Size
	Real-World Matrices						
R1	Hamiltonian1	Nuclear Physics	17040×17040	42.95 M	14.8	687 MB	4.64 GB
R2	human_gene	Gene Expr. (BioInf.)	22283×22283	24.67 M	5.0	395 MB	3.58 GB
R3	TSOPF_RS_b2383	Power Network (Eng.)	38120×38120	32.31 M	2.2	517 MB	6.24 GB
R4	mouse_gene	Gene Expr. (BioInf.)	45101×45101	28.97 M	1.4	463 MB	7.72 GB
R5	Hamiltonian2	Nuclear Physics	52928×52928	188.93 M	6.7	3.02 GB	42.1 GB
R6	Hamiltonian3	Nuclear Physics	77205×77205	319.30 M	5.4	5.11 GB	88.1 GB
R7	barrier2-4	Semicond. Device (Eng.)	$113\,\text{K} \times 113\,\text{K}$	2.13 M	0.016	34 MB	993 MB
R8	pkustk14	Structural Problem (Eng.)	$152\,\text{K} \times 152\,\text{K}$	11.20 M	0.048	179 MB	907 MB
R9	msdoor	Structural Problem (Eng.)	$416\,\text{K} \times 416\,\text{K}$	19.17 M	0.011	230 MB	955 MB
	Generated Matrices	Parameters $\{a, b, c, d\}$					
G1	RMAT1	$\{0.25, 0.25, 0.25, 0.25\}$	$100\,\text{K} \times 100\,\text{K}$	20 M	0.2	320 MB	55.5 GB
G2	RMAT2	$\{0.35, 0.22, 0.22, 0.21\}$	$100\,\text{K} \times 100\,\text{K}$	20 M	0.2	320 MB	60.0 GB
G3	RMAT3	$\{0.45, 0.18, 0.18, 0.19\}$	$100\,\text{K} \times 100\,\text{K}$	20 M	0.2	320 MB	62.3 GB
G4	RMAT4	$\{0.55, 0.15, 0.15, 0.15\}$	$100\,\text{K} \times 100\,\text{K}$	20 M	0.2	320 MB	59.2 GB
G5	RMAT5	$\{0.61, 0.13, 0.13, 0.13\}$	$100\,\text{K} \times 100\,\text{K}$	20 M	0.2	320 MB	55.0 GB
G6	RMAT6	$\{0.64, 0.12, 0.12, 0.12\}$	$100\,\text{K} \times 100\,\text{K}$	20 M	0.2	320 MB	52.6 GB
G7	RMAT7	$\{0.67, 0.11, 0.11, 0.11\}$	$100\,\text{K} \times 100\,\text{K}$	20 M	0.2	320 MB	50.3 GB
G8	RMAT8	$\{0.70, 0.10, 0.10, 0.10\}$	$100\,\text{K} \times 100\,\text{K}$	20 M	0.2	320 MB	47.2 GB
G9	RMAT9	$\{0.73, 0.09, 0.09, 0.09\}$	$100\,\text{K} \times 100\,\text{K}$	20 M	0.2	320 MB	43.7 GB

increasing skew ↓

Table A.2: Adjacency matrices of two example graphs of different dimension and population density. The $\rho = N_{nz}/(n \times n)$ value denotes the population density (rounded) of the adjacency matrix.

Abbreviation	Name	Domain	Dimensions	N_{nz}	$\rho[\%]$
Gra1	Slashdot Netw.	Social Network	77360×77360	905 K	0.01
Gra2	Roadnet CA	Street Network	$1971\text{K} \times 1971\text{K}$	5.533 M	10^{-6}

Table A.2 lists the properties of the two graph data sets which were use in the evaluation: a social (Gra1) and a street network graph (Gra2). The graphs are taken from the SNAP graph library (Leskovec and Krevl, 2014).

Table A.3: Sparse matrices of different dimensions and population densities. The $\rho = N_{nz}/(n \times n)$ value denotes the population density (rounded) of each matrix. All matrices are square ($n \times n$.)

Abbreviation	Name	Domain	Dimensions	N_{nz}	$\rho[\%]$
NCSM1	Hamiltonian4	Nuclear Physics	3440	2.930 M	24.7
PWNET	pwnet	Power Engineering	8140	2.017 M	3.0
JACO1	jaco1	Econometric	9129	56 K	0.07

Table A.3 lists the matrix data sets which we used in the evaluation in Chapter 4. The first matrix NCSM1 is taken from a nuclear physics group, the other two (PWNET, JACO1) are from the Florida Sparse Matrix Collection (Davis and Hu, 2011).

BIBLIOGRAPHY

Daniel J. Abadi, Mike Stonebraker, Edmond Lau, Amerson Lin, et al. C-Store: A Column-oriented DBMS. In *Proc. of 31st VLDB Conf.*, pages 553–564, 2005.

Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. In *Proc. of ACM SIGMOD*, pages 671–682, 2006.

Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. Row-stores: How Different Are They Really? In *Proc. of SIGMOD*, pages 967–980. ACM, 2008.

ACML. AMD Core Math Library, n.d. http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml/ [Accessed: 2016-04-05].

Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, et al. Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects. *J. Phys.: Conf. Ser.*, 180 (1):012037, 2009.

Luben Alexandrov. Parallel Sparse Matrix-Matrix Multiplication. Master's thesis, Karlsruhe Institute of Technology, 2014.

Rasmus Resen Amossen and Rasmus Pagh. Faster Join-Projects and Sparse Matrix Multiplications. In *Proc. of 12th ICDT*, pages 121–126. ACM, 2009.

Ed. Anderson, Zhaojun Bai, Chris Bischof, Jim Demmel, et al. *LAPACK Users' Guide*. SIAM, third edition, 1999.

Apache Software Foundation. Apache SystemML, 2016. http://systemml.apache.org [Accessed: 2016-05-04].

ATLAS. Automatically Tuned Linear Algebra Software ATLAS, n.d. http://math-atlas.sourceforge.net/ [Accessed: 2016-04-05].

Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, et al. PETSc Users Manual. Technical Report ANL-95/11 - Revision 3.6, Argonne National Laboratory, 2015.

R. Barrett, M. Berry, T. F. Chan, J. Demmel, , et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, 1994.

Douglas Bates. Matrix: Sparse and Dense Matrix Classes and Methods, n.d. http://cran.r-project.org/web/packages/Matrix/index.html [Accessed: 2016-04-05].

Peter Baumann, Andreas Dehmel, Paula Furtado, Roland Ritsch, and Norbert Widmann. The Multidimensional Database System RasDaMan. *SIGMOD Rec.*, 27(2):575–577, June 1998.

Peter Baumann, Andreas Dehmel, Paula Furtado, Roland Ritsch, and Norbert Widmann. Spatio-Temporal Retrieval with RasDaMan. In *Proc. of 25th VLDB*, pages 746–749, 1999.

Stephan Baumann, Peter Boncz, and Kai-Uwe Sattler. Bitwise dimensional co-clustering for analytical workloads. *The VLDB Journal*, pages 1–26, 2016.

Nathan Bell and Michael Garland. Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors. In *Proc. of Int. Conf. High Perf. Comp., Netw., Stor. An., SC*, pages 18:1–18:11. ACM, 2009.

Peter Benner and Heike Fassbender. An Implicitly Restarted Symplectic Lanczos Method for the Hamiltonian Eigenvalue Problem. *Linear Algebra and its Applications*, 263:75 – 111, 1997.

L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, et al. *ScaLAPACK Users' Guide*. SIAM, 1997.

Netlib BLAS. Basic Linear Algebra Subprograms, n.d. http://www.netlib.org/blas/ [Accessed: 2016-04-05].

Matthias Boehm, R. Burdick Douglas, Alexandre V. Evfimievski, et al. SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. *IEEE Data Eng. Bull.*, 37(3):52–62, 2014.

Matthias Boehm, Shirish Tatikonda, Berthold Reinwald, et al. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *PVLDB*, 7(7):553–564, 2014.

Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. of CIDR*, pages 225–237, 2005.

Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, 2008.

Harold Borko and Myrna Bernick. Automatic Document Classification. *J. ACM*, 10(2):151–162, April 1963.

George E. P. Box, J. Stuart Hunter, and William G. Hunter. *Statistics for Experimenters: Design, Innovation, and Discovery , 2nd Edition*. Wiley-Interscience, 2 edition, May 2005.

Paul G. Brown. Overview of SciDB: Large Scale Array Storage, Processing and Analysis. In *Proc. of ACM SIGMOD*, pages 963–968. ACM, 2010.

Aydin Buluç and John R. Gilbert. Highly Parallel Sparse Matrix-Matrix Multiplication. *CoRR*, abs/1006.2183, 2010.

Aydin Buluç and John R. Gilbert. The Combinatorial BLAS: Design, Implementation, and Applications. *Int. J. High Perform. Comput. Appl.*, 25(4):496–509, November 2011.

Aydin Buluç and John R. Gilbert. Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments. *SIAM J. Scientific Computing*, 34(4), 2012.

Peter Buneman. Why Don't Scientists Use Databases. Talk at UK National e-Science Centre, 2002.

Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. *Parallel Computing*, 35(1):38–53, 2009.

D. Calvetti, L. Reichel, and Danny C. Sorensen. An implicit restarted Lanczos method for large symmetric eigenvalue problems. *ETNA. Electronic Transactions on Numerical Analysis [electronic only]*, 2:1–21, 1994.

Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *Proc. of 4th SIAM Int. Conf. Data Mining*, pages 442–446, 2004.

John Chambers. A Statistical Data Language. In , *Statistical Computation*. Academic Press, 1969.

Yanqing Chen, Timothy A. Davis, William W. Hager, and Sivasankaran Rajamanickam. Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate. *ACM Trans. Math. Softw.*, 35(3), 2008.

Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven Autotuning of Sparse Matrix-vector Multiply on GPUs. In *Proc. of 15th ACM SIGPLAN PPoPP*, pages 115–126. ACM, 2010.

Edith Cohen. On Optimizing Multiplications of Sparse Matrices. In *Proc. of 5th Int. Conf. Integer Programming and Combinatorial Optimization, IPCO*, pages 219–233. Springer, 1996.

Edith Cohen. Structure Prediction and Computation of Sparse Matrix Products. *Journal of Combinatorial Optimization*, 2(4):307–332, 1998.

Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, et al. MAD Skills: New Analysis Practices for Big Data. *PVLDB*, 2(2):1481–1492, August 2009.

Cristinel Constantin. Principal Component Analysis - A Powerful Tool in Computing Marketing Information. *Bull. Trans. University Brasov*, 7(2), 2014.

George P. Copeland and Setrag Khoshafian. A Decomposition Storage Model. In *Proc. of ACM SIGMOD*, pages 268–279, 1985.

Don Coppersmith and Shmuel Winograd. Matrix Multiplication via Arithmetic Progressions. *J. Symb. Comput.*, 9(3):251–280, March 1990.

Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

CuSparse. Nvidia CuSparse Library, n.d. http://docs.nvidia.com/cuda/cusparse [Accessed: 2016-04-05].

Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

Steven Dalton, Luke Olson, and Nathan Bell. Optimizing Sparse Matrix-Matrix Multiplication for the GPU. *ACM Trans. Math. Softw.*, 41(4):25:1–25:20, October 2015.

Thomas H. Davenport and D.J. Patil. Data Scientist: The Sexiest Job of the 21st Century. *Harvard Business Review*, 90(10):70–76, October 2012.

Timothy A. Davis. Algorithm 915, SuiteSparseQR: Multifrontal multithreaded rank-revealing sparse QR factorization. *ACM Trans. Math. Softw.*, 38(1):8, 2011.

Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by Latent Semantic Analysis. *JASIS*, 41(6):391–407, 1990.

James W. Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

Jack Dongarra, C. Moler, J. Bunch, and G. Stewart. *LINPACK Users' Guide*. SIAM, 1979.

Jack Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, March 1988.

Jack Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, March 1990.

Jack Dongarra, Susan Blackford, Sven Hammarling, Iain Duff, and Jim Demmel. Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard. *IJHPCA*, 16(1):1–111, 2001.

Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki. Accelerating Numerical Dense Linear Algebra Calculations with GPUs. *Numerical Computations with GPUs*, pages 1–26, 2014.

Iain S. Duff, Michele Marrone, Giuseppe Radicati, and Carlo Vittoli. Level 3 Basic Linear Algebra Subprograms for Sparse Matrices: A User-level Interface. *ACM Trans. Math. Softw.*, 23(3):379–401, September 1997.

A. Edelman, S. Heller, and S. Lennart Johnsson. Index Transformation Algorithms in a Linear Algebra Framework. *IEEE Trans. Parallel Distrib. Syst.*, 5(12):1302–1309, Dec 1994.

ESSL. IBM Engineering and Scientific Subroutine Library, n.d. http://www-03.ibm.com/systems/power/software/essl/ [Accessed: 2016-04-05].

Franz Färber, Norman May, Wolfgang Lehner, Philipp Grosse, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.*, 35 (1):28–33, 2012.

Franz Färber, Jonathan Dees, Martin Weidner, Stefan Baeuerle, and Wolfgang Lehner. Towards a web-scale data management ecosystem demonstrated by SAP HANA. In *Proc. of 31st ICDE*, pages 1259–1267, April 2015.

Gang Feng. Stability analysis of piecewise discrete-time linear systems. *IEEE Transactions on Automatic Control*, 47(7):1108–1112, Jul 2002.

David Fitzjarrell and Mary Mikell Spence. *Oracle Exadata Survival Guide*. Apress, 1st edition, 2013.

Malte Förster and Jiri Kraus. Scalable parallel AMG on ccNUMA machines with OpenMP. *Computer Science - R&D*, 26(3-4):221–228, 2011.

J. G. F. Francis. The QR Transformation A Unitary Analogue to the LR TransformationâĂŤPart 1. *The Computer Journal*, 4(3):265–271, 1961.

J. G. F. Francis. The QR TransformationâĂŤPart 2. *The Computer Journal*, 4(4):332–345, 1962.

Paula Furtado and Peter Baumann. Storage of Multidimensional Arrays Based on Arbitrary Tiling. In *Proc. of 15th ICDE*, pages 480–489, 1999.

Burton S. Garbow. {EISPACK} âĂŤ A package of matrix eigensystem routines. *Computer Physics Communications*, 7(4):179 – 184, 1974.

H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Trans. Knowl. Data Eng.*, 4(6):509–516, December 1992.

Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, et al. SystemML: Declarative Machine Learning on MapReduce. In *Proc. of 27th ICDE*. IEEE, 2011.

John R. Gilbert. Predicting Structure in Sparse Matrix Computations. *SIAM J. Matrix Anal. Appl.*, 15 (1):62–79, 1994.

John R. Gilbert and Esmond G. Ng. Predicting Structure in Nonsymmetric Sparse Matrix Factorizations. In , *Graph Theory and Sparse Matrix Computation*, volume 56 of *The IMA Volumes in Mathematics and its Applications*, pages 107–139. Springer, 1993.

John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse Matrices in Matlab: Design and Implementation. *SIAM J. Matrix Anal. Appl.*, 13(1):333–356, January 1992.

J.R. Gilbert, W.W. Pugh, and T. Shpeisman. Ordered sparse accumulator and its use in efficient sparse matrix computation, November 9 1999. US Patent 5,983,230.

Sadashiva S. Godbole. On Efficient Computation of Matrix Chain Products. *IEEE Trans. Comput.*, 22(9):864–866, September 1973.

Peter Gottschling, David S. Wise, and Michael D. Adams. Representation-transparent Matrix Algorithms with Scalable Performance. In *Proc. of 21th International Conference on Supercomputing, ICS*, pages 116–125, 2007.

Georgios Goumas, Kornilios Kourtis, Nikos Anastopoulos, Vasileios Karakasis, and Nectarios Koziris. Performance Evaluation of the Sparse Matrix-vector Multiplication on Modern Architectures. *The Journal of Supercomputing*, 50(1):36–77, 2009.

Goetz Graefe and William J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proc. Int. Conf. Data Eng.*, pages 209–218. IEEE, 1993.

Goetz Graefe and Leonard D. Shapiro. Data Compression and Database Performance. In *Proc. of ACM/IEEE-CS Symp. On Applied Computing*, pages 22–27, 1991.

Jim Gray. eScience – A Transformed Scientific Method. NRC-CSTB meeting, 2007.

Jim Gray, David T. Liu, Maria A. Nieto-Santisteban, Alexander S. Szalay, Gerd Heber, and David DeWitt. Scientific Data Management in the Coming Decade. *ACM SIGMOD Record*, 34(4):35–41, January 2005. also as MSR-TR-2005-10.

D.J. Griffiths. *Introduction to Quantum Mechanics*. Pearson international edition. Pearson Prentice Hall, 2005.

Philipp Große, Wolfgang Lehner, Thomas Weichert, Franz Färber, and Wen-Syan Li. Bridging Two Worlds with RICE Integrating R into the SAP In-Memory Computing Engine. *PVLDB*, 4(12): 1307–1317, 2011.

GSL. GNU Scientific Library, n.d. http://www.gnu.org/software/gsl/ [Accessed: 2016-04-05].

Fred G. Gustavson. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, September 1978.

Martin Hahmann, Dirk Habich, and Wolfgang Lehner. Modular Data Clustering - Algorithm Design beyond MapReduce. In *Proc. of 17th EDBT Workshops*, pages 50–59, 2014.

N. Halko, P. G. Martinsson, and J. A. Tropp. Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions. *SIAM Review*, 53(2):217–288, 2011.

Joseph M. Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, et al. The MADlib Analytics Library: Or MAD Skills, the SQL. *PVLDB*, 5(12):1700–1711, August 2012.

Tony Hey, Stewart Tansley, and Kristin M. Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. *SIGPLAN Not.*, 47(4):349–362, March 2012.

Kurt Hornik. Slam: Sparse Lightweight Arrays and Matrices, n.d. https://cran.r-project.org/web/packages/slam/index.html [Accessed: 2016-04-05].

Botong Huang, Shivnath Babu, and Jun Yang. Cumulon: Optimizing Statistical Data Analysis in the Cloud. In *SIGMOD*, pages 1–12. ACM, 2013.

IBM. *Matrix Engine Developer's Guide*. IBM Netezza Analytics Release 3.0.2.0. IBM, 2014.

IBM. *Netezza Package for R Developer's Guide*. IBM Netezza Analytics Release 3.0.1.0. IBM, 2014.

Yannis E. Ioannidis. The History of Histograms (abridged). In *Proc. of 29th VLDB*, pages 19–30, 2003.

H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and Rama Kanneganti. Incremental Organization for Data Recording and Warehousing. In *Proc. of 23rd VLDB*, pages 16–25. Morgan Kaufmann Publishers, 1997.

Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Accessed: 2016-04-05].

Karen Sparck Jones. A Statistical Interpretation of Term Specificity and its Application in Retrieval. *Journal of Documentation*, 28(1):11–21, 1972.

Ibrahim Kamel and Christos Faloutsos. Hilbert R-tree: An Improved R-tree using Fractals. In *Proc. of 20th VLDB*, pages 500–509. Morgan Kaufmann Publishers, 1994.

George Karypis. METIS and ParMETIS. In *Encyclopedia of Parallel Computing*, pages 1117–1124. 2011.

J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*. Software, Environments, Tools. SIAM, 2011.

David Kernert, Frank Köhler, and Wolfgang Lehner. Bringing Linear Algebra Objects to Life in a Column-Oriented In-Memory Database. In *Rev. Sel. Papers of 1st and 2nd IMDM Int. Workshops*, pages 44–55, 2013.

David Kernert, Frank Köhler, and Wolfgang Lehner. SLACID - Sparse Linear Algebra in a Column-oriented In-memory Database System. In *Proc. of 26th SSDBM*, pages 11:1–11:12. ACM, 2014.

David Kernert, Frank Köhler, and Wolfgang Lehner. SpMacho - Optimizing Sparse Linear Algebra Expressions with Probabilistic Density Estimation. In *Proc. of 18th EDBT*, pages 289–300, 2015.

David Kernert, Norman May, Michael Hladik, Klaus Werner, and Wolfgang Lehner. From Static to Agile - Particle Physics on the SAP HANA DB. In *Proc. of 4th DATA*, 2015.

David Kernert, Wolfgang Lehner, and Frank Köhler. Topology-Aware Optimization of Big Sparse Matrices and Matrix Multiplications on Main-Memory Systems. In *Proc. of 32nd ICDE*, 2016.

M. Kersten, Y. Zhang, M. Ivanova, and N. Nes. SciQL, a Query Language for Science Applications. In *Proc. of EDBT/ICDT Workshop on Array Databases*, pages 1–12. ACM, 2011.

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220 (4598):671–680, 1983.

Andi Kleen. A NUMA API for LINUX. Technical report, SUSE Labs, 2005. http://halobates.de/numaapi3.pdf [Accessed: 2016-04-05].

Yuval Kluger, Ronen Basri, Joseph T. Chang, and Mark Gerstein. Spectral Biclustering of Microarray Cancer Data: Co-clustering Genes and Conditions. *Genome Research*, 13:703–716, 2003.

Jiri Kraus, Malte Förster, Thomas Brandes, and Thomas Soddemann. Using LAMA for efficient AMG on hybrid clusters. *Computer Science - R&D*, 28(2-3):211–220, 2013.

Moritz Kreutzer, Jonas Thies, Melven Röhrig-Zöllner, Andreas Pieper, et al. GHOST: Building Blocks for High Performance Sparse Linear Algebra on Heterogeneous Systems. *CoRR*, abs/1507.08101, 2015.

Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa M. Badia. Scheduling Dense Linear Algebra Operations on Multicore Processors. *Concurrency and Computation: Practice and Experience*, 22 (1):15–44, 2010.

Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-scale Graph Computation on Just a PC. In *Proc. of 10th USENIX Conf. Operating Systems Design and Implementation, OSDI*, pages 31–46. USENIX Association, 2012.

Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. *SIGPLAN Not.*, 26(4):63–74, April 1991.

Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. The Vertica Analytic Database: C-Store 7 Years Later. *PVLDB*, 5(12):1790–1801, 2012.

Christoph Lameter. NUMA (Non-Uniform Memory Access): An Overview. *ACM Queue*, 11(7):40, 2013.

C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

François Le Gall. Powers of Tensors and Fast Matrix Multiplication. In *Proc. of 39th Int. Symp. Symbolic and Algebraic Computation, ISSAC*, pages 296–303. ACM, 2014.

Heejo Lee, Jong Kim, Sung Je Hong, and Sunggu Lee. Processor Allocation and Task Scheduling of Matrix Chain Products on Parallel Systems. *IEEE Trans. Parallel Distrib. Syst.*, 14(4):394–407, April 2003.

Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How Good Are Query Optimizers, Really? *PVLDB*, 9(3):204–215, 2015.

Christian Lemke, Kai-Uwe Sattler, Franz Faerber, and Alexander Zeier. Speeding Up Queries in Column Stores - A Case for Compression. In *Proc. of. 12th DAWAK*, pages 117–129, 2010.

Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data, June 2014.

Ming Li, Jonathan H. Badger, Xin Chen, et al. An Information-Based Sequence Distance and its Application to Whole Mitochondrial Genome Phylogeny. *Bioinformatics*, 17(2):149–154, 2001.

K.-T. Lim, D. Maier, M. Kersten, Y. Zhang, and M. Stonebraker. Array QL Syntax. Technical report, XLDB, 2012. http://www.xldb.org/wp-content/uploads/2012/09/ArrayQL-Draft-4.pdf [Accessed: 2016-04-05].

Weixiang Liu, Tianfu Wang, and Siping Chen. Regularized Nonnegative Matrix Factorization for Clustering Gene Expression Data. In *BIBM, IEEE*, pages 1–4, Dec 2013.

Guy M. Lohman and John A. Muckstadt. Optimal Policy for Batch Operations: Backup, Checkpointing, Reorganization, and Updating. *ACM Trans. Database Syst.*, 2(3):209–222, 1977.

Peter Macko. *LLAMA: A Persistent, Mutable Representation for Graphs*. PhD thesis, Havard University, December 2014.

Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *Proc. of 31st ICDE*, pages 363–374, 2015.

Roger MacNicol and Blaine French. Sybase IQ Multiplex - Designed For Analytics. In *Proc. of 30th VLDB*, pages 1227–1230, 2004.

Christopher D. Manning, Prabhakar Raghavan, and Hinrich SchÃijtze. Scoring, term weighting, and the vector space model. In *Introduction to Information Retrieval*, pages 100–123. Cambridge University Press, 2008. Cambridge Books Online.

Volker Markl. *MISTRAL: Processing Relational Queries using a Multidimensional Access Technique*, volume 59 of *DISDBIS*. Infix, 1999.

Per-Gunnar Martinsson, Vladimir Rokhlin, and Mark Tygert. A randomized algorithm for the decomposition of matrices. *Applied and Computational Harmonic Analysis*, 30(1):47 – 68, 2011.

K. Matam, S.R.K.B. Indarapu, and K. Kothapalli. Sparse Matrix-matrix Multiplication on Modern Architectures. In *Proc. of 19th Int. Conf. High Perf. Comp., HiPC*, pages 1–10, Dec 2012.

Mathworks Inc. MATLAB - The Language of Technical Computing, n.d. www.mathworks.com/matlabcentral/ [Accessed: 2016-04-05].

Tim McGuire, James Manyika, and Michael Chui. Why Big Data is the New Competitive Advantage. *Ivey Business Journal*, July 2012. http://iveybusinessjournal.com/publication/why-big-data-is-the-new-competitive-advantage/ [Accessed: 2016-04-05].

Xiangrui Meng, Ameet Talwalkar, Evan Sparks, Virginia Smith, and Xinghao Pan. MLlib: Scalable Machine Learning on Spark, 2014.

Microsoft. What's New in SQL Server R Services, 2016. https://msdn.microsoft.com/en-us/library/mt604847.aspx [Accessed: 2016-04-05].

Renee J. Miller. Big Data Curation. In *Proc. of DIMACS/CCICADA Big Data Integration Workshop*, 2013.

R.C. Mittal and A. Al-Kurdi. LU-Decomposition and Numerical Structure for Solving Large Sparse Nonsymmetric Linear Systems. *J. Comput. Appl. Math.*, 43(1âĂŞ2):131 – 155, 2002.

MKL. Intel Math Kernel Library (MKL), n.d. http://software.intel.com/en-us/intel-mkl [Accessed: 2016-04-05].

MLib. HP Mathematical Software Library (MLIB), n.d. www.hp.com/go/mlib [Accessed: 2016-04-05].

Guido Moerkotte. Constructing Optimal Bushy Trees Possibly Containing Cross Products for Order Preserving Joins is in P. Technical report, University of Mannheim, 2003.

Bernard M. E. Moret, David A. Bader, and Tandy Warnow. High-Performance Algorithm Engineering for Computational Phylogenetics. In , *Proc of. Int. Conf. Computational Science, ICCS. Part II*, volume 2074 of *LNCS*, pages 1012–1021. Springer, 2001.

G.M. Morton. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. IBM, 1966.

Musing Mortoray. The Amortized Cost of Vector Insert is 3. *Programming and Language Design. Blog Post*, 2014. https://mortoray.com/2014/08/22/the-amortized-cost-of-vector-insert-is-3/ [Accessed: 2016-04-05].

Netlib MPI. MPI - The Complete Reference, n.d. http://netlib.org/utk/papers/mpi-book/mpi-book.html [Accessed: 2016-04-05].

M. Muralikrishna and David J. DeWitt. Equi-depth Multidimensional Histograms. *SIGMOD Rec.*, 17(3):28–36, June 1988.

J. Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. Database Syst.*, 9(1):38–71, March 1984.

NISTa. Sparse Basic Linear Algebra Subprograms (BLAS) Library, n.d. http://math.nist.gov/spblas/ [Accessed: 2016-04-05].

NISTb. Basic Linear Algebra Subprograms, n.d. http://math.nist.gov/spblas/original.html [Accessed: 2016-04-05].

Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The Log-structured Merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, June 1996.

Oracle. Oracle R Enterprise 1.3 Reference Manual, 2012. http://www.oracle.com/technetwork/database/options/advanced-analytics/r-enterprise/ore-reference-manual-1882822.pdf [Accessed: 2016-04-05].

Oracle. Oracle Advanced Analytics. Enterprise-Wide Predictive Analytics, 2013. http://www.oracle.com/technetwork/database/options/advanced-analytics/oaa12cds-1964642.pdf [Accessed: 2016-04-05].

Oracle. Oracle R Enterprise User's Guide, 2015. https://docs.oracle.com/cd/E11882_01/doc.112/e36761/intro.htm#OREUG109 [Accessed: 2016-04-05].

Oracle Doc. Oracle Database Online Documentation, 10g Release2, n.d. https://docs.oracle.com/cd/B19306_01/appdev.102/b14258/u_nla.htm#CIABEFIJ [Accessed: 2016-04-05].

Carlos Ordonez and Javier García-García. Vector and matrix operations programmed with UDFs in a relational DBMS. In *Proc. of 15th CIKM*, pages 503–512. ACM, 2006.

L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford University, 1998.

Rasmus Pagh and Morten StÃűckel. The Input/Output Complexity of Sparse Matrix Multiplication. In *Algorithms - ESA 2014*, volume 8737 of *LNCS*, pages 750–761. Springer, 2014.

Marcus Paradies, Wolfgang Lehner, and Christof Bornhövd. GRAPHITE: An Extensible Graph Traversal Framework for Relational Database Management Systems. In *Proc. of 27th SSDBM*, pages 29:1–29:12. ACM, 2015.

Paralution Labs. PARALUTION v1.0.0, 2015. http://www.paralution.com [Accessed: 2016-04-05].

Md.Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, et al. Parallel Efficient Sparse Matrix-Matrix Multiplication on Multicore Platforms. In *Proc. of 30th Int. Conf. ISC High Perf.*, volume 9137 of *LNCS*, pages 48–57. Springer, 2015.

Roger Pearce, Maya Gokhale, and Nancy M. Amato. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *Proc. of Int. Conf. High Perf. Comp., Netw., Stor. An., SC*, pages 1–11. IEEE, 2010.

Gregory Piatetsky-Shapiro and Charles Connell. Accurate Estimation of the Number of Tuples Satisfying a Condition. *SIGMOD Rec.*, 14(2):256–276, June 1984.

Pivotal Software. The Greenplum Database, n.d. http://greenplum.org/ [Accessed: 2016-04-05].

Alex Pothen, Horst D. Simon, and Kan-Pu Liou. Partitioning Sparse Matrices with Eigenvectors of Graphs. *SIAM J. Matrix Anal. Appl.*, 11(3):430–452, May 1990.

ProteomicsDB, n.d. https://www.proteomicsdb.org/ [Accessed: 2016-04-05].

Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-aware Data and Task Placement. In *Proc. of 41st VLDB Conf.*, 2015.

C. Yang R. B. Lehoucq, D. C. Sorensen. ARPACK, 1997. http://www.caam.rice.edu/software/ARPACK/ [Accessed: 2016-04-05].

Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, et al. DB2 with BLU Acceleration: So Much More than Just a Column Store. *PVLDB*, 6(11):1080–1091, 2013.

Riccardo Rebonato and Peter JÃď ckel. The Most General Methodology to Create a Valid Correlation Matrix for Risk Management and Option Pricing Purposes, 1999.

R Foundation. The R Project for Statistical Computing, n.d. https://www.r-project.org [Accessed: 2016-04-05].

Brian Ripley. An ODBC database interface, 2015. http://cran.r-project.org/web/packages/Matrix/index.html [Accessed: 2016-04-05].

Vladimir Rokhlin, Arthur Szlam, and Mark Tygert. A Randomized Algorithm for Principal Component Analysis. *SIAM Journal on Matrix Analysis and Applications*, 31(3):1100–1124, 2010.

Mark A. Roth and Scott J. Van Horn. Database Compression. *SIGMOD Record*, 22(3):31–39, 1993.

Robert Roth. Importance Truncation for Large-Scale Configuration Interaction Approaches. *Phys.Rev.*, C79, 2009.

Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. The Graph Story of the SAP HANA Database. In *BTW*, pages 403–420, 2013.

Yousef Saad. *SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations*. University of Minnesota Department of Computer Science and Engineering, version 2 edition, June 1994.

Yousef Saad. *Numerical Methods for Large Eigenvalue Problems*. Algorithms and Architectures for Advanced Scientific Computing. SIAM, 2011.

Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. Restricted Boltzmann Machines for Collaborative Filtering. In *Proc. of 24th Int. Conf. Machine Learning, ICML*, pages 791–798. ACM, 2007.

Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5):513 – 523, 1988.

Peter Sanders and Christian Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Proc. of 12th Int. Symp. Experimental Algorithms , SEA*, volume 7933 of *LNCS*, pages 164–175. Springer, 2013.

Sunita Sarawagi and Michael Stonebraker. Efficient Organization of Large Multidimensional Arrays. In *Proc. of 10th ICDE*, pages 328–336, 1994.

Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Incremental Singular Value Decomposition Algorithms for Highly Scalable Recommender Systems. In *Proc of. 5th Int. Conf. Computer and Information Science*, pages 27–28, 2002.

Gerald Schubert, Georg Hager, Holger Fehske, and Gerhard Wellein. Parallel Sparse Matrix-Vector Multiplication As a Test Case for Hybrid MPI+OpenMP Programming. In *Proc. of 25th IEEE IPDPSW*, pages 1751–1758. IEEE, 2011.

Jim Scott. Get Real with Hadoop: Read-Write File System. Blog Post., 2014. https://www.mapr.com/blog/get-real-hadoop-read-write-file-system [Accessed: 2016-04-05].

P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, pages 23–34. ACM, 1979.

Glenn Shafer and Vladimir Vovk. The Sources of KolmogorovâĂŹs Grundbegriffe. *Statist. Sci.*, 21 (1):70–98, 02 2006.

Viral Shah and John R. Gilbert. Sparse Matrices in Matlab*P: Design and Implementation. In , *Proc. of 12th Int. Conf. High Perf. Comp., HiPC*, volume 3296 of *LNCS*, pages 144–155. Springer, 2005.

C. E. Shannon. A Mathematical Theory of Communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(1):3–55, January 2001.

Ben Shneiderman. Optimum Data Base Reorganization Points. *Commun. ACM*, 16(6):362–365, 1973.

Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., 2002.

Jeremy G. Siek and Andrew Lumsdaine. The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra. In *Proc. of 2nd Int. Symp. Computing in Object-Oriented Parallel Environments, ISCOPE*, pages 59–70, 1998.

Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *Proc. of ACM SIGMOD*, pages 731–742. ACM, 2012.

D. B. Skillicorn. *Social Network Analysis via Matrix Decompositions*, pages 367–391. John Wiley and Sons, 2005.

S. Smith, N. Ravindran, N.D. Sidiropoulos, and G. Karypis. SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication. In *Proc. of 29th IEEE IPDPS*, pages 61–70, May 2015.

Gary H. Sockut and Balakrishna R. Iyer. Online Reorganization of Databases. *ACM Comput. Surv.*, 41(3):14:1–14:136, July 2009.

Spark Documentation. Spark Data Types - MLlib, n.d. http://spark.apache.org/docs/latest/mllib-data-types.html [Accessed: 2016-05-04].

SPL. Oracle Sun Performance Library, n.d. https://docs.oracle.com/cd/E24457_01/html/E21987/gkezy.html [Accessed: 2016-04-05].

Michael Stonebraker. Inclusion of New Types in Relational Data Base Systems. In *Proc. of 2nd ICDE*, pages 262–269. IEEE, 1986.

Michael Stonebraker. The Postgres DBMS. In *Proc. of ACM SIGMOD*, page 394, 1990.

Michael Stonebraker, Chuck Bear, Ugur Çetintemel, Mitch Cherniack, Tingjian Ge, et al. One Size Fits All? Part 2: Benchmarking Studies. In *Proc. of 3rd CIDR*, pages 173–184, 2007.

Michael Stonebraker, Paul Brown, Donghui Zhang, and Jacek Becla. SciDB: A Database Management System for Applications with Complex Analytics. *Computing in Science and Engineering*, 15 (3):54–62, 2013.

Michael Stonebraker, Sam Madden, and Pradeep Dubey. Intel "Big Data" Science and Technology Center Vision and Execution Plan. *SIGMOD Rec.*, 42(1):44–49, May 2013.

Volker Strassen. Gaussian Elimination is not Optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

Masuo Suzuki. Quantum transfer-matrix method and thermo-quantum dynamics. *Physica A: Stat. Mech. Appl.*, 321(1-2):334–339, April 2003.

Robert Endre Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

Keita Teranishi, Padma Raghavan, Jun Sun, and Panagiotis Michaleris. An Evaluation of Limited-Memory Sparse Linear Solvers for Thermo-Mechanical Applications. *Int. J. Num. Meth. Eng.*, 74 (11):1690–1715, 2008.

Luke Tierney. A Byte Code Compiler for R. Technical report, University of Iowa, 2016.

TIOBE. Programming Community Index, 2016. http://tiobe.com/index.php/content/paperinfo/tpci/index.html [Accessed: 2016-04-05].

Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, June 2010.

Andrew Tulloch. Fast Randomized SVD. *Blog Post*, 2014. https://research.facebook.com/blog/fast-randomized-svd/ [Accessed: 2016-04-05].

Simon Urbanek, 2014. RJDBC: Provides access to databases through the JDBC interface, https://cran.r-project.org/web/packages/RJDBC/index.html.

Vinod Valsalam and Anthony Skjellum. A framework for High-Performance Matrix Multiplication based on Hierarchical Abstractions, Algorithms and Optimized Low-Level Kernels. *CCPE*, 14(10): 805–839, 2002.

Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A Library of Automatically Tuned Sparse Matrix Kernels. In *Proc. of SciDAC, J. Physics: Conf. Ser.*, volume 16, pages 521–530, 2005.

Richard W. Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels*. PhD thesis, University of California, January 2004. http://bebop.cs.berkeley.edu/pubs/vuduc2003-dissertation.pdf [Accessed: 2016-04-05].

Richard W. Vuduc and Hyun-Jin Moon. Fast Sparse Matrix-vector Multiplication by Exploiting Variable Block Structure. In *Proc. of 1st Int. Conf. High Perf. Comp. Com., HPCC*, pages 807–816. Springer, 2005.

StÃľfan van der Walt, S. Chris Colbert, and GaÃńl Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

Mathias Wilhelm, Judith Schlegl, Hannes Hahne, Amin M. Gholami, Marcus Lieberenz, et al. Mass-spectrometry-based draft of the human proteome. *Nature*, 509(7502):582–587, May 2014.

Florian Wolf, Iraklis Psaroudakis, Norman May, Anastasia Ailamaki, and Kai-Uwe Sattler. Extending Database Task Schedulers for Multi-threaded Application Code. In *Proc. of 27th SSDBM*, pages 25:1–25:12. ACM, 2015.

Kesheng Wu and Horst Simon. Thick-Restart Lanczos Method for Large Symmetric Eigenvalue Problems. *SIAM J. Matrix Anal. Appl.*, 22(2):602–616, May 2000.

Wei Xu, Xin Liu, and Yihong Gong. Document Clustering Based on Non-negative Matrix Factorization. In *Proc. of 26th ACM SIGIR Conf. Research and Development in Informaion Retrieval*, SIGIR '03, pages 267–273. ACM, 2003.

Ichitaro Yamazaki, Hartwig Anzt, Stanimire Tomov, Mark Hoemmen, and Jack Dongarra. Improving the Performance of CA-GMRES on Multicores with Multiple GPUs. In *Proc. of 28th IEEE IPDPS*, pages 382–391, 2014.

V. Yegnanarayanan. An application of matrix multiplication. *Resonance*, 18(4):368–377, 2013.

Raphael Yuster and Uri Zwick. Fast Sparse Matrix Multiplication. *ACM Trans. Algorithms*, 1(1):2–13, July 2005.

A. N. Yzelman and Rob H. Bisseling. Cache-Oblivious Sparse Matrix-Vector Multiplication by Using Sparse Matrix Partitioning Methods. *SIAM J. Scientific Computing*, 31(4):3128–3154, July 2009.

Reza Bosagh Zadeh and Gunnar Carlsson. Dimension Independent Matrix Square using MapReduce. *CoRR*, abs/1304.1467, 2013.

Reza Bosagh Zadeh, Xiangrui Meng, Burak Yavuz, Aaron Staple, Li Pu, Shivaram Venkataraman, et al. linalg: Matrix Computations in Apache Spark. *CoRR*, abs/1509.02256, 2015.

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proc. of 2nd USENIX Conf. Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10. USENIX Association, 2010.

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, et al. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proc. of 9th USENIX Conf. Networked Systems Design and Implementation, NSDI*, pages 2–2. USENIX Association, 2012.

Yi Zhang, Herodotos Herodotou, and Jun Yang. RIOT: I/O-Efficient Numerical Computing without SQL. In *Proc. of 4th CIDR*, 2009.

Yi Zhang, Weiping Zhang, and Jun Yang. I/O-efficient statistical computing with RIOT. In *Proc. of 26th ICDE*, pages 1157–1160, 2010.

Yi Zhang, Kamesh Munagala, and Jun Yang. Storing Matrices on Disk: Theory and Practice Revisited. *PVLDB*, 4(11):1075–1086, 2011.

Marcin Zukowski, Mark van de Wiel, and Peter A. Boncz. Vectorwise: A Vectorized Analytical DBMS. In *Proc. of 28th ICDE*, pages 1349–1350, 2012.

List of Figures

1.1 The LAPEG architecture and organization of the thesis. 19

2.1 A small exemplary matrix from the NCSM simulation. 24

3.1 Physical organization of a relational table in a column-store. 46
3.2 Representation of vectors in the column-oriented storage layer. 49
3.3 Representations of matrices in the column-oriented storage layer. 51
3.4 The CSR representation. 53
3.5 The dictionary-encoded social network table. 55
3.6 Inner vs. outer product as layered graph model. 62
3.7 Illustration of the Gustavson algorithm. 64
3.8 Parallelization approaches for row-based matrix multiplication. 67

4.1 Performance dependency of Algorithm 4 on search depth. 70
4.2 Architecture of the DBMS-integrated linear algebra processing engine. 72
4.3 A matrix multiplication execution node. 74
4.4 Possible execution plans for the multiplication of three matrices. 75
4.5 Runtimes and time estimates for the scaling behavior of multiplication kernels. . . 80
4.6 Runtimes and time estimates for multiplication kernels by varying the matrix shape. 81
4.7 Example for scalar density and the density map. 83
4.8 Product density in extreme non-uniform cases. 84
4.9 Accuracy of the density estimation and disorder measure. 88
4.10 Quality of the optimizer decision. 90
4.11 Execution runtime of sparse matrix chain multiplications. 92
4.12 Chain multiplication experiment with parallel spspsp_gemm. 94
4.13 Runtime comparison of SPMACHO vs. alternative approaches. 95
4.14 Different skew types. 96

5.1 The TSOPF_RS_b283 matrix (R3) as AT MATRIX. 106
5.2 Schematic illustration of the quadtree partitioning. 107
5.3 The internal organization of the AT MATRIX. 112
5.4 The read performance of AT MATRIX vs. CSR 113
5.5 Adaptive tile multiplication 115
5.6 The 1D water-level method on the accumulated density distribution. 118
5.7 Parallel resource distribution for ATMULT. 119
5.8 Task DAG of the ATMULT operator. 121
5.9 Component durations of the AT MATRIX partitioning. 123
5.10 Multiplication experiments where $\boldsymbol{A} = \boldsymbol{B}$ (sparse-sparse). 125
5.11 Multiplication experiments where $\boldsymbol{B}$ is dense (sparse-dense). 126

5.12 Multiplication experiments where $\boldsymbol{A}$ is dense (dense-sparse). 127
5.13 Impact of optimization steps on the performance. 128

6.1 Matrix sub-array access patterns. 138
6.2 Logical and physical view of matrix manipulations. 146
6.3 Two variants of a mutable AT Matrix. 151
6.4 Implementation of a row block-delete operation in AT Matrix. 154
6.5 Implementation of a flip operation in AT Matrix. 154
6.6 Temporal behavior of an insert/read workload. 156
6.7 Delta Size Experiment. 159
6.8 Total read/write query throughput of different approaches. 160
6.9 Average duration of matrix row delete operations. 161
6.10 Average duration of matrix column delete operations. 163