ENGINEERING AGGREGATION OPERATORS FOR RELATIONAL IN-MEMORY DATABASE SYSTEMS

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

INGO MÜLLER

aus Breisach am Rhein.

Karlsruhe Institute of Technology

Tag der mündlichen Prüfung:	11. Februar 2016
Erster Gutachter:	Prof. Dr. rer. nat. Peter Sanders
Zweiter Gutachter:	Prof. Dr.-Ing. Wolfgang Lehner

Ingo Müller: *Engineering Aggregation Operators for Relational In-Memory Database Systems,* 11. Februar 2016

It is the supreme art of the teacher
to awaken joy in creative expression and knowledge.

— Albert Einstein

To my teachers.

ABSTRACT

Relational Aggregation is one of the major means to analyze large data sets since the creation of the first database systems. Available hardware performance continues to grow at an exponential rate, but increasingly so through specialization, which makes it non-trivial to leverage in software. At the same time, application demands grow at an even higher pace. This puts database systems in a continuous race for hardware-conscious system architectures, more efficient algorithms, and better implementations—with Aggregation being a fundamental building block.

In this thesis we study the design and implementation of Aggregation operators in the context of modern database systems. In particular, we identify and address the following challenges: cache-efficiency, CPU-friendliness, parallelism within and across processors, robust handling of skewed data, adaptive processing, processing with constrained memory for intermediate results, and integration with state-of-the-art database architectures. While many of these challenges have been studied in isolation, we are the first to address them at the same time.

To guide our algorithm design, we study cache-efficiency of Aggregation in several external memory models. The lower bounds we derive show that for many realistic machine parameters, Aggregation has the same cache-complexity as MultisetSorting, even if non-comparison-based techniques such as hashing are allowed. This proves a long-standing folklore conjecture of the database community. Furthermore, we show that linear speed-up is optimal and how it can be achieved in this model using any realistic number of processors. Our lower bounds also identify situations where Aggregation has a lower complexity than MultisetSorting and we provide algorithms that make our bounds tight for many parameter ranges.

We use these insights to design and implement a practical algorithm. It has the anatomy of a sort algorithm in order to achieve cache-efficiency, but sorts by hash value for better load balancing. Furthermore, the algorithm can adaptively decide to use hashing as a subroutine to benefit from high data locality. Low level tuning ensures that all routines make efficient use of modern hardware. The result is a novel relational Aggregation algorithm that is cache-efficient—independently and without prior knowledge of input skew and output cardinality—, highly parallelizable on modern multi-core systems, and operating at a speed close to the memory bandwidth, thus outperforming the state-of-the-art by up to 3.7×.

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

[61] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. “The SAP HANA Database – An Architecture Overview.” In: *IEEE Data Eng. Bull.* 35.1 (2012), pp. 28–33

[159] Peter Sanders, Sebastian Schlag, and Ingo Müller. “Communication efficient algorithms for fundamental big data problems.” In: *IEEE Big Data Conf.* 2013

[195] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. “Vectorizing Database Column Scans with Complex Predicates.” In: *ADMS*. 2013

[134] Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. “Retrieval and Perfect Hashing using Fingerprinting.” In: *SEA*. 2014

[132] Ingo Müller, Cornelius Ratsch, and Franz Faerber. “Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems.” In: *EDBT*. 2014

[83] Lorenz Hübschle-Schneider, Peter Sanders, and Ingo Müller. “Communication Efficient Algorithms for Top-k Selection Problems.” In: *CoRR* abs/1502.0 (2015)

[133] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. “Cache-Efficient Aggregation: Hashing Is Sorting.” In: *SIGMOD*. 2015

If I have seen further
it is by standing on the shoulders of giants.

— Jean-Pierre Maury [128]

ACKNOWLEDGMENTS

This thesis has been a long, hard, and instructive, but also beautiful journey, which would not have been possible without the many people who accompanied me. This is an attempt to express my gratitude towards them. It is doomed to fail—what I feel I owe to my fellow travelers is infinitely more than what words can say.

I would first like to thank my supervisors, who were most important for the scientific aspects of my work. It was Peter Sanders who introduced me to algorithm engineering and complexity theory, but also to our industry partner SAP many years ago when I was an undergraduate student. Since the first days, his brilliance, creativity, and integrity have amazed and inspired me and this thesis is but a humble attempt to follow his footsteps. Wolfgang Lehner complemented him perfectly, with his unparalleled ability to motivate me to work, his visionary ideas for the greater picture, and his devotion to make every story a great story. Without Wolfgang as my helmsman, none of my projects would have been of use for the database community.

I would also like to thank their counterparts at SAP. First, there is Franz Färber, my supervisor. He is the personification of the force of innovation and has been the visionary, restless initiator and godfather of countless projects in the company, this thesis being only one of many. Its impact on products would not have been the same, wouldn't it be for Franz' thrive for productization and support of all kind. And then there is Arne Schwarz, head of the database campus, like a mighty knight leading and protecting an army of students. He made me feel welcome and appreciated and gave me many opportunities to develop myself—thank you!

My most compassionate thoughts are with my fellow PhD students in the SAP HANA Campus. Like every ordeal in life that one goes through in a group, this time has created a bond among us that we will never forget. Together, we were a great team: Thomas Bach, Robert Brunel, Jan Finis, Philipp Große, Matthias Hauck, Martin Kaufmann, David Kernert, Ismail Oukid, Iraklis Psaroudakis, Marcus Paradies, Hannes Rauhe, Michael Rudolf, Abdallah Issa Salama, Francesc Trull, Elena Vasilyeva, Florian Wolf, and Mathias Wilhelm. In particular, I would like to thank Hannes for the consistently positive and inclusive spirit he brought to the group, which lasted until long after he had gone. I would also like to thank Michael, for helping everyone, unconditionally, always, with everything. Thank you for the many times you helped me! Thank you also, Jonathan, for the great technical and non-technical discussions we had, as controversial as they may have been at times, and the great time we had, at work and afterwards. Last but not least, I want to thank Ismail, for being like my small brother: looking up to me, but surprising me time and again by doing things better than me (although for a long time, there is no reason to be surprised anymore).

A special role for my thesis was played by Arnaud Lacurie. He joined SAP after doing his master thesis under my supervision and since then took over more and more of the code of this project. Without his moral support, his development effort, and the interesting conversations—related to work or not—I could not have completed my PhD. Thank you!

I would also like to thank my co-authors, without whom none of my publications could have been completed: Jonathan Dees, Franz Färber, Philipp Große, Lorenz Hübschle-Schneider, Arnaud Lacurie, Wolfgang Lehner, Norman May, Ismail Oukid, Cornelius Ratsch, Hannes Rauhe, Peter Sanders, Sebastian Schlag, Robert Schulze, Thomas Willhalm, and Wei Zhou. I owe particular to Nodari Sitchinava, for working on the external memory problem with me.

Maybe to their surprise, I also owe many things to the students I supervised. Cheng Chen, Arnaud Lacurie, Cornelius Ratsch, Sebastian Schlag, Andreas Schuster, and Wei Zhou. In this role I have learned so many things about myself! It was a pleasure to work with you.

Essential was also the support of the numerous people at our industry partner, including Frank Benkstein, Alexander Böhm, Nico Bohnsack, Nurten Celik, Roman Dementiev, Gordon Gaumnitz, Anil Goel, Arnaud Lacurie, Norman May, Kai Stammerjohann, Carsten Thiel, Frederik Transier, Robert Schulze, and Reza Sherkat.

Furthermore, I would like to thank my "second family" at the faculty: Yaroslav Akhremtsev, Julian Arz, Michael Axtmann, Tomáš Balyo, Timo Bingmann, Daniel Funke, Simon Gog, Lorenz Hübschle-Schneider, Moritz Kobitzsch, Tobias Maier, Sebastian Schlag, Christian Schulz, Jochen Speck, Darren Strash, Sascha Witt, G. Veit Batz, Johannes Fischer, Dennis Luxen, Vitaly Osipov, and Dennis Schieferdecker. It was always great to come to visit you in Karlsruhe and get a fresh point of view on my work and the rest of the world.

Finally, I would like to thank my family for giving me the best mental support one can hope for. To my parents for giving fostering my curiosity since I am on this world. And to Glòria, my future wife, for tolerating all of the countless times when finishing something at work took longer than expected. This thesis is really your merit.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF LISTS

LIST OF LISTS THAT DO NOT CONTAIN THEMSELVES [189]

```
LaTeX Warning: Label(s) may have changed. Rerun to get cross-
references right
```

1 INTRODUCTION

1.1 CONTEXT AND MOTIVATION

For decades our society has been experiencing an increasing digitalization of almost all of its aspects: business processes, financial transactions, governance, online shopping, research in all disciplines, medicine, personal communication, transport, and living, and many more. Mankind produces and collects unprecedented amounts of data that are difficult to apprehend; and at the same time, the ability to extract meaning out of it becomes more important and technically challenging. A large variety of software applications has emerged with that goal—sometimes grouped under the term *business intelligence*—, offering concepts, such as data analytics and reporting, online analytical processing (OLAP), data mining, et cetera, and are continuously being extended.

This trend is paralleled by technological advances in computing hardware. Processing power and storage space have sustained exponential growth since their early days half a century ago and promise to continue to do so in the future. For a long time, this development was completely transparent to software, but now advances often can only be achieved by specializing and explicitly exposing more and more hardware components. Software can thus benefit from the full computational power only if it is conscious about the inner workings of the hardware. This means that more and more responsibility is shifted to the software level, making software increasingly complex and difficult to develop.

To handle the confluence of these two trends, special software has been developed as abstraction layer: *relational database management systems*. By providing a logical, structured view of the data, they relieve applications from many technical aspects of data management so that they can focus on *what* should be done, while the systems take care about *how* to do it. In particular, it is the task of database management systems to process application requests such that the hardware is used at its fullest potential. Building database management systems is therefore a continuous race of adapting to every new hardware generation in order to process faster and faster the ever increasing amounts of data.

At the heart of this race lies a set of *operators*—algorithmic building blocks that the system combines to carry out computations upon application requests. One of the most important operators is AGGREGATION. It consists in summarizing a collection of records by dividing them into groups, each of which is aggregated into a single value. For example, the entries of all sold products in the book of a retail chain could be summarized by the total number of items sold by each store. Virtually all analytical applications require this functionality and a typical database system spends a large fraction of its processing time with this operator. The goal of the present thesis is to engineer a new generation of AGGREGATION operators that advances database management systems in their race to leverage modern hardware.

1.2 CHALLENGES

Building AGGREGATION operators is challenging for a number of reasons. Starting with the aforementioned hardware consciousness, concrete challenges are the following: First, it is important that the operators reduce costly data movement. In modern database systems that keep almost all data in main memory, this concerns vertical data movement between the main memory and processor caches, but also horizontal data movement between different processing units. Furthermore, modern processors can execute certain types of computing instructions faster than others, so algorithms should be implemented such that they use efficient instruction sequences as often as possible. Finally, today's computers consist of numerous computing units running in parallel—as multiple components inside a single chip, as several chips inside the same computer, or as several computers clustered together. Software can thus only use all compute resources if it can split its work into pieces that can be executed in parallel by independent processing units and communication between them is kept low. In short, to leverage modern hardware, AGGREGATION operators need to be built such that they are *cache-efficient*, *CPU-friendly*, highly *parallelizable*, and *communication-efficient*.

Another group of challenges is related to the desired generality of database operators. They should provide good performance independently of properties of the input data. For AGGREGATION, different strategies work best with modern hardware depending on the number of groups in the data. This is a property that is difficult to know in advance. Therefore, it is attractive for AGGREGATION operators to automatically *adapt* their strategy to the data during execution. Similarly, a situation where more records in the data belong to a certain group than to others is difficult to handle. In order to work robustly for all applications, we need mechanisms to deal with *skewed distributions* of groups. Both *adaptivity* and *skew resistance* are thus desirable properties, but challenging to achieve.

A final group of challenges stems from the fact that AGGREGATION operators are part of a larger system. First, this means that they need to share hardware resources with other software components. In the context of in-memory database systems, this is particularly difficult to achieve with the resource main memory. Operators need main memory to store intermediate results, but, depending on the system, the amount available for that purpose may be limited. Finally, different types of database management systems have different interfaces between operators, which may affect choices in the algorithm design. For the design of Aggregation algorithms, it is thus important to make sure that they are capable to run even under *constrained memory* and are *integrable* into state-of-the-art architectures of database management systems.

Each of these challenges is difficult enough to achieve in isolation. Matters are further complicated if all of them should be addressed *at the same time*. Each design decision to improve a certain property may negatively affect how well the resulting operator performs in a different dimension. The task of engineering complete Aggregation operators is therefore also challenging because it consists of balancing trade-offs that lead to an overall good solution.

1.3 CONTRIBUTIONS AND THESIS OUTLINE

In this thesis we take on the challenges of engineering Aggregation operators in the context of in-memory database systems.

In a first part, we study Aggregation from a theoretical point of view. We analyze the problem in several external memory machine models in order to understand its limits in cache efficiency. We are able to show that for many realistic parameters of these machine models, Aggregation has the same lower bound as MultisetSorting. This proves a long-standing folklore conjecture of the database community, which implies that the best strategy for Aggregation depends on how many groups there are in the input and that using a sort-based algorithm is optimal if the number of groups is large. The insights gained from this theoretical analysis form a timeless guideline for designing cache-efficient Aggregation algorithms.

In a second part, we iteratively engineer a practical algorithm that meets all challenges mentioned earlier. With the first iteration we challenge the commonly accepted view that the two traditional strategies, HashAggregation and SortAggregation, are completely opposite. Instead, we design a single algorithm that inherits features of both strategies. A simple, cheap mechanism lets our operator adapt its behavior to the data, during execution and without relying on external information given in advance. It is therefore at least as cache-efficient as any of the two static choices. Furthermore, the careful integration of a probabilistic streaming algorithm provides our operator with an accurate estimate of the number of groups in the input, which avoids potentially prohibitive resizing costs of the data structure that will hold the result. We also use a number of low-level tuning techniques to make our algorithm CPU-friendly and ensure that it can be integrated with state-of-the-art execution models of database systems. Finally, we show how to parallelize our operator both within and across processors and despite skewed input distribution such that it executes fully in parallel at all times.

With the second iteration we take away the common, but unrealistic assumption that memory available for intermediate results is unlimited. We develop techniques that allow executing different phases of a recursive algorithm in a pipeline and apply them to our AGGREGATION operator. They ensure that intermediate results after a phase can quickly be consumed by the next phase and their memory can be released. The more frequent switching of tasks inherent to pipelining is a potential source of overhead, but with clever intra-operator scheduling we are able to sustain performance close to the one of the original version while fully preserving its adaptivity and skew resistance.

We confirm the viability of our algorithm design with extensive experiments. In comparison with several state-of-the-art competitors our implementation is the fastest in almost all situations, with large speed-ups in many situations, peaking at a 3.7 times lower execution time for large numbers of groups. Our operator runs at the speed of the memory bandwidth, scales to a high number of processing units, and achieves the same performance even for extremely skewed inputs. Our cardinality estimator effectively eliminates resizing costs of the result data structure with very low overhead, thus saving up to half of the execution time. If memory for intermediate results is constrained to a mere 1.6 % of the original memory usage, our operator experiences an overhead of just 20 % to 47 % and can be tuned to higher performance by increasing the constraint.

In summary, by combining a thorough theoretical analysis, clever algorithmic design, and a series of engineering techniques, we build a single, versatile AGGREGATION operator that is state-of-the-art with respect to all challenges that arise in the context of in-memory database systems.

The rest of the thesis is organized as follows: In Chapter 2 and 3, we review the foundations of this thesis and prior work on AGGREGATION, respectively. In Chapter 4 we present the theoretical study of AGGREGATION in external memory models. The two design iterations of our practical operator are presented in Chapter 5 (combining HASHING and SORTING) and Chapter 6 (memory constraint) respectively. Finally, Chapter 7 concludes the thesis with a discussion of limits and possible extensions of our work.

Part I

STATE OF THE ART

The condition of scientific or technical knowledge, particularly the peak or highest level thereof, at a particular time.

— Wiktionary [193]

2 FOUNDATIONS

In this chapter, we lay the foundations of our work: First, we introduce the problem of Aggregation and present how it is used in real-world applications. Second, we review relational database management systems, the abstraction layer that most applications use for storing data and computing Aggregation and other operations. Then we summarize the most important features of modern hardware that database systems use to carry out application requests. Finally, we review the scientific method of *algorithm engineering*, which allows us to build an Aggregation operator that efficiently uses modern hardware and guides all other chapters of this thesis. Readers familiar with some or all of these topics may skip the respective sections.

2.1 GROUPING WITH AGGREGATION

2.1.1 *Example and Formal Definition*

Aggregation is a common means to summarize a large collection of data records such that a particular characteristic of the collection can be understood easily by a human. It consists in assigning each record to a group and to aggregate each group to a single record. Sometimes it is therefore called GroupingWithAggregation, which may be shortened to Grouping or Aggregation. Except where a distinction is needed, we use the latter abbreviation in this thesis.

Figure 1 shows a simple example of Aggregation in the context of a European office supply chain. It answers the question: "What is the sum of the prices of all sold items per store?". In this example, the entries of sold items are *grouped* into one group per store and the prices of the rows of each group are *aggregated* to their sum. This allows understanding which articles generated the highest revenue according to the books of the company.

This concept can be formalized [99, 37] as part of the *relational algebra*. The relational algebra is an algebra of *operators* on sets of *tuples* called relations. The operator Aggregation is defined as a function[1] γ on relations, parametrized with a set of *grouping attributes* $A_1, \ldots, A_g$ and a list of *aggregation functions* $f_2(A), \ldots, f_C(A)$, where A is the set of all attributes of R. It is often noted as follows [95]:

$$_{A_1,\ldots,A_g}\gamma_{f_2(A),\ldots,f_C(A)}(R) \qquad (1)$$

1 Sometimes Γ, g, $\mathcal{G}$, or $\mathcal{F}$ are used in the literature rather than γ.

Input

Store	Item	Price
Berlin	pen	1.00 €
Berlin	paper	3.00 €
Paris	ruler	2.00 €
Berlin	pen	1.00 €
Paris	pen	1.00 €
Vienna	paper	3.00 €

$\xrightarrow{\text{Aggregation}}$

Output

Store	Sum
Paris	3.00 €
Vienna	3.00 €
Berlin	5.00 €

Figure 1: Example aggregation query: “What is the sum of the prices of all sold items per store?”

For simplicity of exposition, we assume $g = 1$ in this thesis, i.e., we assume that a single attribute determines the group of each row, like in the example in Figure 1. This is not a real limitation since multiple attributes can be (logically) concatenated to a single one. For a similar reason, we also assume that every remaining attribute is the single input of exactly one function. This implies that both input and output relation have C attributes or *columns*. Our simplified definition of AGGREGATION has the following formal semantics:

$$
\begin{aligned}
{}_{A_1}\gamma_{f_2(A_2),\dots,f_C(A_C)}(R) = \Big\{ & (a_1, s_2, \dots, s_C) \,|\, a_1 \in \pi_{A_1}(R), \\
& s_2 = f_2\left(\pi_{A_2}\left(\sigma_{A_1=a_1}(R)\right)\right), \\
& \dots, \\
& s_C = f_C\left(\pi_{A_C}\left(\sigma_{A_1=a_1}(R)\right)\right)\Big\},
\end{aligned} \tag{2}
$$

where σ is the relational operator that *selects* the rows of a relation given a predicate, π is the relational operator that *projects* a relation to a subset of its attributes, and A_1 is the grouping attribute.

Applications typically express their requests to the database systems with the *Structured Query Language* (SQL) which is based on the relational algebra. However, while strict relational algebra is defined on sets, SQL's default is to work on multisets, i.e., duplicate rows are not eliminated. We interpret Equation 2 in this way and adopt this interpretation throughout the thesis. In SQL AGGREGATION is expressed with a GROUP BY clause. With accordingly defined tables, the above example query would be expressed in SQL like shown in Algorithm 1.

The most common aggregate functions in SQL are COUNT, SUM (as in the example in Figure 1), AVG (average), MIN, and MAX. The SQL standard [86, Section 4.16.4] also defines a range of other, less common aggregate functions and different vendors have added others as proprietary extensions. For a classification of aggregate functions, see

Algorithm 1 Example query expressed in SQL

```
1:     SELECT Store, SUM(Price) AS Sum
2:       FROM Sales
3: GROUP BY Store
```

Gray et al. [78]. Furthermore, each aggregate function has a DISTINCT variant, which only works on the distinct values of the attribute, but this is out of the scope of this thesis.

There are two problems that are considered to be similar to AGGREGATION: JOIN and DUPLICATEREMOVAL. In a JOIN, tuples of two (potentially different) relations have to be brought together, i.e., "joined", if they have the same (join) key. AGGREGATION can be seen as a JOIN with only one relation. In DUPLICATEREMOVAL we are asked to keep exactly one tuple of each group of tuples where all attributes are the same. This is equivalent to an AGGREGATION where all attributes are grouping attributes and no aggregates are computed. We discuss aspects of JOIN and DUPLICATEREMOVAL throughout the thesis where insights can be transferred to AGGREGATION.

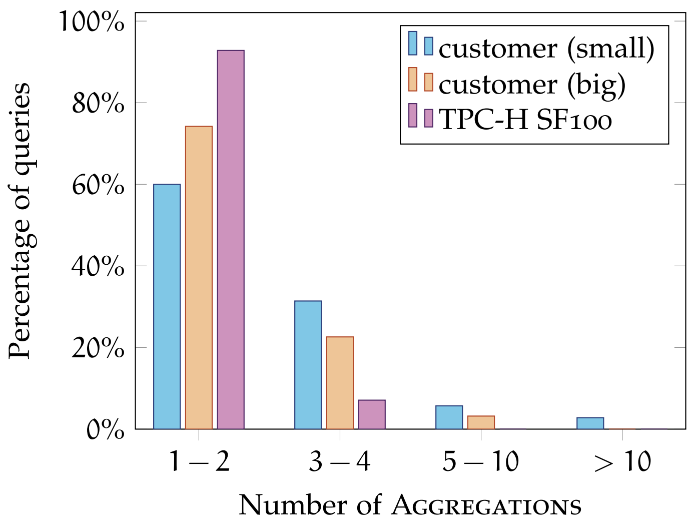
2.1.2 *Real-World Workload Study*

AGGREGATION plays an important role in analytical workloads. In this section, we present a workload study of a real-world business application, which initially motivated the project of this thesis. While it confirms many well known usage patterns of the AGGREGATION operator, our study also reveals some insights that contrast the textbook descriptions of typical workloads.

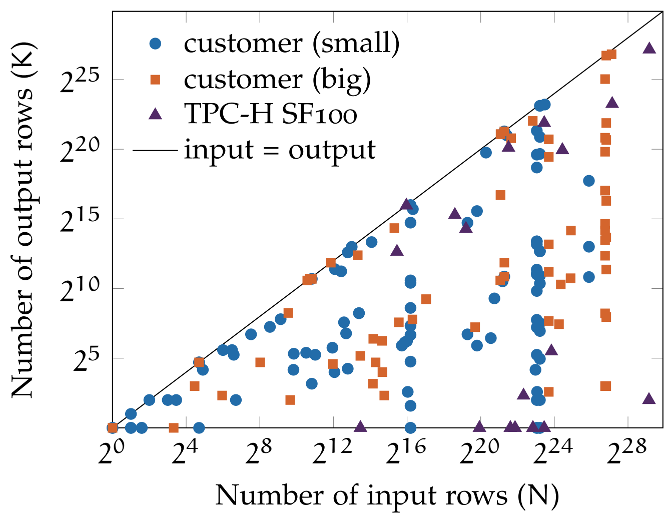
The workload of the study consists of two sizes of a pre-sales benchmark of business analytics on ERP data of a customer of SAP. The queries are particularly long running and difficult queries of an existing system of the customer for performance comparison with the SAP HANA Database. For reference we include the queries of the TPC-H benchmark [172] with scale factor 100. We run the queries on the SAP HANA Database [61] and trace the statistics of the AGGREGATION operator.

Our first observation is illustrated in Figure 2a, suggesting that the queries are complicated enough so that the execution plan contains more than one AGGREGATION operator. While the TPC-H query set consists mostly of rather simple queries with one or two AGGREGATIONS, our business use case is more complex and has queries with up to 16 instances of AGGREGATION operators.

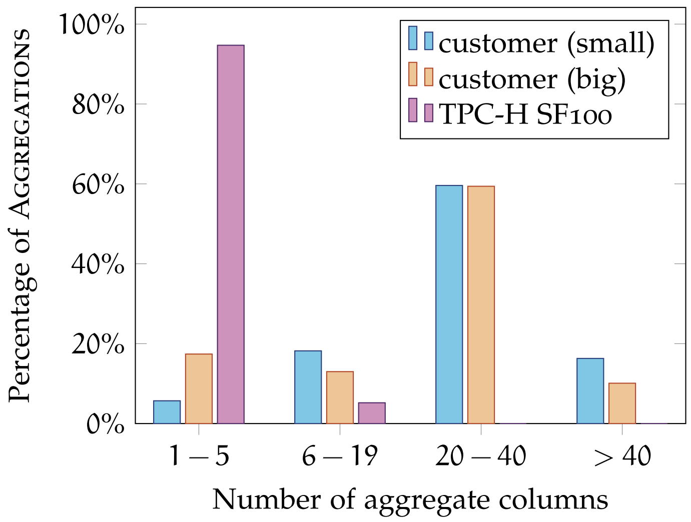
Our second observation concerns the distribution of input size and output size (denoted N and K respectively) and is illustrated in Figure 2b. While the TPC-H data sets consist rather of two distinct types of queries, one where the output relation is of small, fixed size and one where the output is almost as large as the input, all combinations



(a) Aggregations per query.



(b) Input and output size.



(c) Aggregate columns.

Figure 2: Characteristics of the aggregations in the TPC-H and customer queries.

exist in our real-life queries. This means that algorithms have to perform well on all combinations of N and K. Particularly interesting are queries with a large number of groups K, which has not been in the focus of optimization in most prior work. As Figure 2b shows, output sizes of several million groups are not uncommon. We observed that this often happens in legacy applications that "misuse" AGGREGATION to ensure uniqueness properties not ensured by constraints in the schema. This is something that a general purpose database system needs to support efficiently, but literature does not offer good solutions for.

Our final observation is illustrated in Figure 2c. It shows that the number of aggregation columns is high in analytical workload. The TPC-H queries consist mostly of less than five columns with a maximum of ten. Again, our business query set is more complex, with many queries having more than 20 aggregates and some up to more than 50. We conclude that benchmarks with a small number of columns are not realistic and that it is important to test AGGREGATION algorithms with a large number of columns.

2.2 RELATIONAL DATABASE MANAGEMENT SYSTEMS

We now review the relevant aspects of a typical *database management system* (DBMS) in order to understand what role the AGGREGATION operator has. Most if not all of this section is taught in every introductory course on database systems, so we refer to accompanying textbooks such as [95, 166] for more details.

A database management system provides a clear, abstract, and simple interface for storage, manipulation, and access of data, thus relieving application developers from these common tasks. By separating the *logical representation* of the data from the *physical representation*, applications only need to care about the former and the latter can be changed—manually or automatically—independently from the application. In this work we concentrate on the *relational data model* [45], where all data is stored in *relations* (or *tables*). A DBMS manages *databases* or—more accurately— *database instances*, which are collections of relations. Each relation is an instance of a precisely defined *schema*, a set of named *attributes*, possibly with a *constrained domain*. Applications store, manipulate, and query data through a *data manipulation language* (DML). For relational database systems, the declarative *Structured Query Language* (SQL) [86] is by far the most common one. It is theoretically backed up by the *relational algebra*, an algebra of *operators* on sets of *tuples* called relations, although it does not strictly adhere to it for efficiency reasons.

The physical representation of the relations is handled by a *storage manager*. The storage manager stores at least one copy of each relation on some *durable* storage medium, traditionally on disk. In these *disk-based systems*, often-used parts of the data is kept in faster main memory in a *buffer pool*. Query processing may write large intermediate results to disk, which has virtually unlimited capacity. In contrast, in *in-memory database systems*, the disk is only used for backups. All working data is stored instead in the faster, but smaller main memory, thus making RAM the "slow" storage medium *vis-a-vis* the CPU caches and limiting the capacity available for intermediate results. There are two main physical storage layouts: row-major order or *row-wise* storage and column-major order or *column-wise* storage, which are chosen depending on the dominant access pattern to the data. *Transactional workload* rather consists of queries that only touch a few rows, but most attributes of them, so the row-wise format is used for this workload. *Analytical workload* often consists of queries that only involve a few attributes, but many rows of a relation, so the column-wise format is used instead. By increasing locality for typical access patterns, choosing the right storage format increases efficiency of access to storage media. For details about the storage layer, we refer to [166, Chapter 10] or [95, Chapter 7].

Queries of the data manipulation language are first parsed into a *logical query plan* by a *DML compiler*. This plan consists of logical operators similar to the relational operators, such as SELECTION, PROJECTION, JOIN, or AGGREGATION. An *optimizer* then transforms the logical plan into a *physical query plan*. This allows choosing among several alternatives the execution plan that is best suited for a particular query on a particular relation. Typical transformations are changing the order of operators, using a series of physical operators for a relational operator, using a physical operator that computes the combination of several relational operators, adding operators that reduce the amount of work for later operators, and selecting one among several physical operators, some of them possibly using *indices*. For example SELECTION could be translated into INDEXSCAN or LINEARSEARCH or JOIN could be translated to HASHJOIN or SORTMERGEJOIN.

In order to select the best possible execution plan, many query optimization techniques have been developed. Most optimizers have a set of *equivalence rules* in order to generate equivalent plans for the same query. They use a combination of *heuristics*, *pruning*, and *dynamic programming* to efficiently select the best of them. A *cost models* of the operators, for example modelling the number of I/Os or cache misses caused by a plan, then allows selecting the cheaper of the two. A simple rule is *selection push-down*, where *Selection* operators are pushed down to the leaves, filtering out data early on in the query thus reducing the amount of work for later queries. Plans may also be transformed to contain *interesting orderings*: For example, a SORTMERGEJOIN followed by a special AGGREGATION optimized for sorted inputs could be faster than a HASHJOIN followed by HASHAGGREGATION. The costs of a plan depend to a large degree on the number of tuples, so it is important to have *size estimates* of intermediate results. Depending on the operators, sizes can be predicted based on *selectivity estimates* or *distinct value estimates*, which in turn can be produced using *sampling*, *histograms*, and other techniques. More details about query processing and query optimization can be found in [166, Chapters 12 and 13] and [95, Chapter 8].

Once the physical plan is created, it is executed by the *query execution engine*. The execution engine follows one of several *processing models*, i.e., physical ways to pass data between the operators of a query plan. For one, intermediate results may be represented row-wise or column-wise, which may or may not be the same representation as for the original relations. Furthermore, traditional systems often follow a *pull model*, where operators recursively ask their children to produce the next row (or batch of rows). Several operators are thus connected to *pipelines*, eliminated the need to materialize many intermediate results, separated by *pipeline breakers*, such as most JOIN or AGGREGATION operators. In contrast to that, particularly in systems with column-wise processing, sometimes a *push model* is used, which means that operators produce their complete result and give it to their parent(s) at once. Both models are implemented with an abstract interface, where operators call each other with virtual function calls. Because this can be prohibitively slow, a trend to overcome this problem has emerged recently: *just-in-time compilation* of the query plan of each query into machine code [140, 67, 138].

Many of these aspects have an influence on the design choices or at least the implementation of an AGGREGATION operator. We will come back to them in more detail when we discuss prior work in Chapter 3 and our operator in Chapter 5.

2.3 MODERN HARDWARE

One main task of database management systems is to ensure that computations carried out for applications fully utilize the potential of modern hardware in order to achieve the best possible performance. For that purpose, database systems in general and their implementation of operators in particular need to be designed with the hardware architecture in mind. In this section we review the most important aspects of modern hardware relevant for this process.

Today's computer hardware architecture is characterized by a complex hierarchy of specialized components, each designed to maximize performance of the software running on it, while keeping the interface to software as simple as possible. However, in the last decades, software has had to become more and more conscious of the inner workings of hardware in order to fully utilize it and database systems are no exception to this trend [41, 1, 20, 204, 124]. In this section we briefly review the most important features of modern hardware

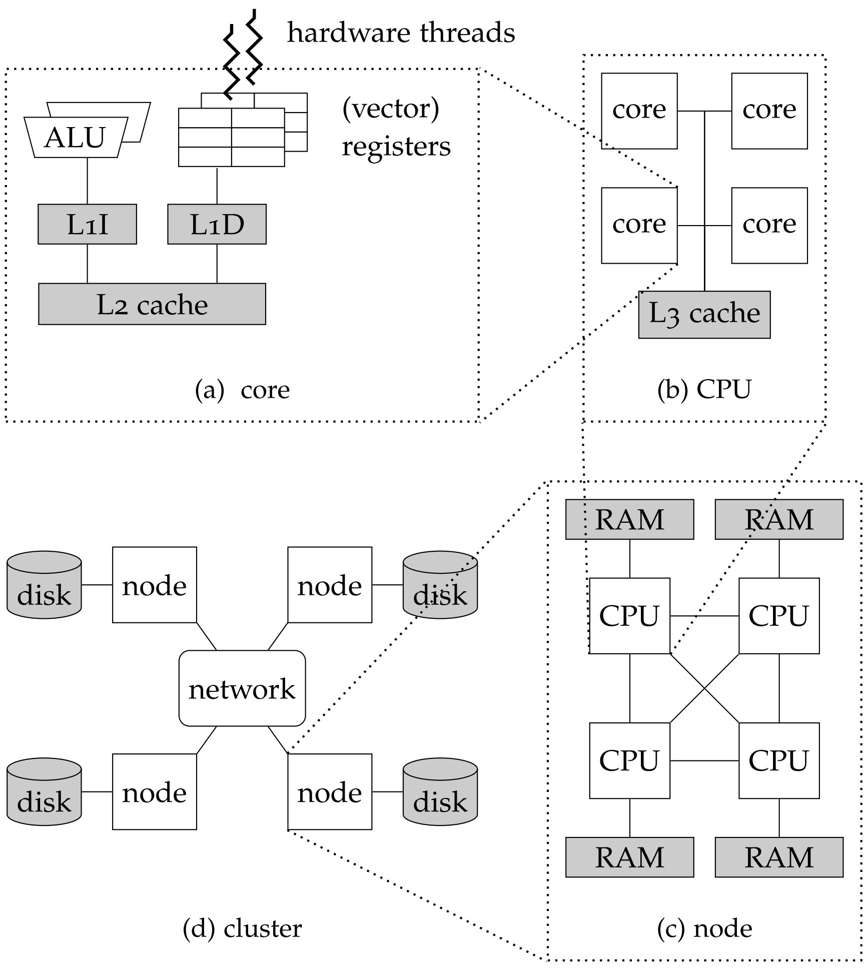


Figure 3: Compute and memory hierarchy of modern hardware

relevant for today's in-memory database systems. This review is necessarily simplified. Furthermore, we concentrate on large servers with Intel processors, which are dominant in our industry context. For a more in-depth coverage of the topic and other architectures, we recommend more specialized literature [145, 58, 49].

The shift in hardware architecture was mainly caused by the fact that three physical limitations were reached, which made performance improvements less automatic than previously [145, 147]. First, the so-called "power wall" prevents a further increase of processor frequency; second, the "ILP wall" (instruction level parallelism, see below) limits performance improvements of deeper pipelines; and third, the "memory wall", caused by the fact that processing speed increases at a faster pace than memory bandwidth, turns data access more and more into a bottleneck.

Modern hardware overcomes these limits through a hierarchy of compute resources and memory, illustrated by Figure 3. In short, a *node* has one or several *CPUs* (central processing units, usually one per *socket*), a certain amount of main memory or *RAM* (random access memory), and possibly one or several hard disks. Each CPU consists of several *cores* each of which has one or two levels of *cache*, which may make the distinction between instructions and data (like the first level in Figure 3 (a)) or be used for both (like the second level in the figure). Often the cores of one CPU share another level of cache. Furthermore, one core can often execute several threads at the same time, i.e., the hardware supports *simultaneous multithreading*. Making a single node larger is called "scale-up". It is also possible to connect several nodes to a *cluster* using high-speed networks, thus doing "scale-out".

The memory hierarchy within a node (usually excluding disks) is functionally transparent for software in the sense that programs use memory through a single virtual address space and the hardware moves the data through the different levels as needed. The closer the memory is to a compute resource, the faster it is, but also the smaller its capacity is. This allows hiding, at least in parts, the access costs of the next slower, but larger memory level. Different levels of the memory hierarchy are organized in blocks of different sizes and only entire blocks can be transferred between levels (*cache lines* in case of cache/RAM transfer and *pages* in case of RAM/disk transfer). The hardware decides which cache lines to place into which level of the cache based on the access pattern of the software using different *cache replacement policies*. Furthermore, a sequential access pattern is usually faster than random access thanks to a *prefetching* unit in the hardware. The hardware also keeps copies of the same cache line in different caches coherent through clever *cache coherency protocols* between the caches. In a multi-socket environment, each part of the main memory is only directly connected to one socket and accesses from one socket to memory of a different socket (remote accesses) are slower than memory access within a socket (local accesses). We speak of *Non-Uniform Memory Access* (NUMA). Disks and memory of different nodes in a cluster are usually addressed through file or message passing APIs respectively, although the operating system can also map them into the program's virtual address space.

The compute hierarchy of modern hardware is more explicit to the software than the memory hierarchy. Software can only run on different nodes, CPUs, or cores if it expresses coarse-grained parallelism (or *thread level parallelism*) through independent threads of control flow. Many modern CPUs also offer more fine-grained levels of parallelism: *data parallelism* (or *vectorization*) and *instruction level parallelism*. Vectorization consists of special instructions executing the same logical operation on multiple adjacent values stored in vector registers, similar to operations on vectors. Compilers can generate code using these instructions only in some situations; in general vector instructions need to be used explicitly for best performance. Instruction level parallelism consists in executing several instructions of the same thread at the same time, thus doing *super-scalar* execution. This is possible by breaking instructions up into many small stages, each executed by a dedicated unit inside the core arranged in a pipeline. Slow units, such as *arithmetic logic units* (ALUs), may be duplicated and the core may rearrange instructions without dependencies for better resource usage (we speak of *out-of-order execution*). Branches in the control flow can only be executed in the pipeline if a *branch prediction* unit correctly predicts them. If a CPU executes several threads simultaneously, instructions of different threads are executed in an interleaved fashion, thus allowing to use the resources inside the core to a fuller extent.

This hierarchy poses fundamental challenges to software architecture. A software system can only use all compute resources if it explicitly expresses coarse-grained parallelism as independent threads of control flow. This in turn makes synchronization [49] and work balancing necessary. Software also needs to be aware of the memory hierarchy, e.g., by favoring access patterns with locality. Finally, the performance critical paths of software have to be written in a "CPU-friendly" way, e.g., by avoiding control flow and data dependencies and by using vectorized instructions. One main theme of this thesis is to propose solutions for these challenges faced during the design and implementation of a relational AGGREGATION operator.

2.4 ALGORITHM ENGINEERING

In order to leverage the power of modern hardware to the fullest, we follow the philosophy of *algorithm engineering* in this thesis. Algorithm engineering is a methodology for designing and implementing usable algorithms with provable properties on real-world computer hardware. It has emerged from the community of algorithm theory in order to improve applicability of theoretical results and forms now a field of research of its own. Because many aspects of database systems research, in particular in query processing, concern the performance of algorithms and data structures on real systems, we think that it is interesting to see our work as an algorithm engineering show case.

In short, algorithm engineering can be explained as illustrated by Figure 4 (for details we refer to literature dedicated to the topic [158, 135]). The main idea is to see the engineering of algorithms as an inductive, iterative cycle of *design*, *analysis*, *implementation*, and *experiments* (the blue boxes in Figure 4): The design yields a concise, abstract description of the algorithm in question, possibly written in pseudo-code, and is usually based on realistic models of the machine and the problem. This algorithm can be formally analyzed to deduce performance guarantees in the given models. Traditional algorithm theory often ends at this point. In algorithm engineering however, falsifiable hypotheses about the algorithm behavior are formulated, and tested with a careful implementation and extensive experiments. Possible discrepancies between the expected and actual behavior are reduced in subsequent iterations of the whole cycle. Since the im-

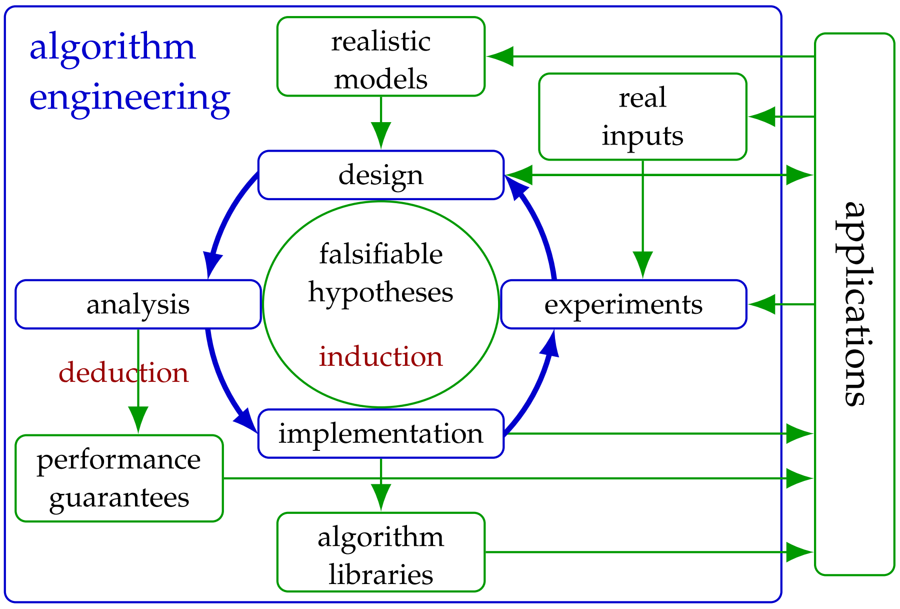


Figure 4: The cycle of algorithm engineering [158]

plementation of the algorithm is not seen as a detail, but as a first-class outcome of the engineering process, it is often materialized in a library, where it can be used by other algorithms, as well as applications. Applications also drive other aspects of the process: Experiments and their inputs are defined by the application, and the machine model and the algorithm design should be compatible with the requirements of the application as well, in order to ensure that the resulting algorithm can actually be used.

This methodology applies very naturally to query processing: As a very practical field, the database systems community has a strong tradition in implementing and conducting experiments with real inputs derived from real applications or realistic benchmarks—a tradition that we also apply in this work. In a way, a database system can even be seen as a library that makes storage and processing technology reusable for applications.[2] Furthermore, in query optimization, the performance of algorithms is often quantified by a cost model in order to let the system choose the most efficient way to execute a given query. In algorithm engineering, models play a slightly different role: they are used by a human—not a machine. They enable the algorithm engineer to drive the engineering process by explaining the interaction of the algorithm, the underlying hardware, and the application. In this work for example, we use the external memory model to understand the cache behavior of AGGREGATION algorithms (see next section and Chapter 4).

2 However, from the point of view of the AGGREGATION operator, the database system can also be seen as part of the *application*, since many aspects of the architecture of the system are immutable facts that have to be taken into account for its design.

3 RELATED WORK

In this chapter, we summarize the most important work on Aggregation operators, a line of work that spans almost four decades. We structure our review around eight challenges that we find reoccurring in the work of most authors: I/O efficiency or cache efficiency, CPU friendliness, parallelism within processing units and communication efficiency across them, robust handling of skewed data, adaptive strategies to handle locality, execution with constrained memory, and integration of algorithms into processing models. Most of this work was done in the context of disk-based architectures, the traditional storage medium of database systems, which we review briefly. We concentrate on work on in-memory database systems, which have got a lot of attention in the last one to two decades. As we argue throughout our review, the work on dask-based systems cannot be applied directly to the in-memory architecture, but many of the insights gained from disk-based algorithms are intrinsic to the problem of Aggregation and have necessarily influenced more recent work. Other insights have rather been forgotten, so reconsidering them in the context of in-memory systems may be worthwhile. For a more detailed summary, we refer to surveys on the topic, which exist in the context of disk-based [76, 77, 187, 186] and in-memory systems [203].

3.1 OVERVIEW: CHALLENGES OF AGGREGATION OPERATORS

Before examining previous work of other authors about Aggregation in detail, we give an overview of the challenges that they faced. This allows us to structure our study of prior work around these challenges. The challenges also serve as a check list that we use for a comparison of the most promising existing solutions and as requirements for the solutions we propose. The challenges themselves are well-known: many solutions were proposed for each of them. However, most authors concentrated on one or a few of them and, as we will see in our final comparison, no solution proposed so far solves all of them. We thus consider that naming all challenges explicitly is a contribution of us by its own right.

The challenges for the implementation of an Aggregation operator in in-memory database systems are the following:

CACHE EFFICIENCY. As a data-intensive operation, the performance of an Aggregation operator heavily depends on how it uses the memory subsystem. Our review of modern hardware shows that efficient usage of caches is required to overcome the bottleneck to slow main memory, i.e., *vertical* communication should be reduced. Furthermore, our workload study shows that different queries have a large variance in the number of groups, which is the determining factor of the size of the output and thus the cache footprint. AGGREGATION operators have to use the caches efficiently—independently of the number of groups.

CPU FRIENDLINESS. While data movement between main memory and caches is expensive, the amount of computations that can be completely overlapped with memory access (thus being *de facto* free) is very small on modern CPUs. This may make *I/O-efficient* solutions proposed for disk-based systems unsuited for the in-memory setup, if they are too intricate and complex. Our review of processors in the previous chapter rather shows that modern CPUs consist of sophisticated components such as deep instruction pipelines, vector instruction units, and branch predictors. To reduce the costs of computations, AGGREGATION algorithms thus need to be conscious about these inner workings of the hardware, i.e., be *CPU-friendly*.

PARALLEL EXECUTION. As discussed in Section 2.3, parallelism at several levels is fundamental for increasing performance on modern hardware. Because analytical queries such as those from our workload study in Section 2.1.2 have a large fraction of long-running AGGREGATIONS, parallelization of this operator is important. Within a CPU, the challenge consists in splitting up the work among multiple cores while keeping *synchronization costs* low.

COMMUNICATION EFFICIENCY. Another challenge arises as parallelization is scaled up to scenarios with several CPUs and non-uniform memory access or—to an even more extreme degree—as it is scaled out to scenarios with slow access to remote memory. In these cases *horizontal* communication, i.e., communication among the compute units, becomes the bottleneck. AGGREGATION operators that should scale to large nodes or even clusters thus have to be designed such that they minimize communication across compute units. In other words, they should be *NUMA-aware* and *communication efficient*.

SKEW RESISTANCE. Another challenge for AGGREGATION operators to handle of *skew*. Skew refers to a non-uniform distribution of grouping attributes and may have negative effects on the performance of some algorithms, for example due to contention during concurrent access to frequent groups or uneven distribution of work among threads. AGGREGATION operators should preserve robust performance independently of the distribution of the input.

ADAPTIVITY. As we discuss in Section 2.2, the traditional approach to achieve cache efficiency for all numbers of groups is to implement two algorithms and let an optimizer decide which one to choose. Similarly, some algorithms depend on a tuning parameter from the optimizer. However, we also discuss the consequent problem of bad optimizer decisions. An AGGREGATION operator can remove this decision by automatically *adapting* its behavior to the query and data in order to provide optimal performance in all situations and without intervention of an optimizer.

CONSTRAINED MEMORY USAGE. While the previous challenges aim at reducing execution time, it is also important that AGGREGATION operators do not use excessive amounts of memory for intermediate results. This aspect is often ignored when designing algorithms for performance benchmarks, but is fundamental for productive use in commercial systems. The challenge is of particular importance for in-memory database systems, where all intermediate results have to be kept in the limited main memory. It is therefore important that AGGREGATION operators continue to work at a high performance if their available memory is constrained in size.

SYSTEM INTEGRATABILITY. Finally, an AGGREGATION operator is always part of a system and has to integrate with specificities of the system's architecture. Relevant aspects include the storage format of relations and intermediate results or the functional interface between operators. Not all algorithms lent themselves equally well for all architectures. As the dominating processing models for business analytics are column-wise processing and Just-in-Time compiled query plans, we focus on integration with these two models.

After seeing a sketch of each challenge, we now present prior work for solving each of them. We concentrate on solutions that solve the challenges *inside* the AGGREGATION operator, but also review alternatives of this approach along with their advantages and disadvantages. Some proposed solutions appear several times, as they make major contributions to solving several of above challenges. In the end of this chapter, we summarize the most complete solutions and which of them solves which of the challenges.

3.2 CACHE EFFICIENCY

We start with the *efficient usage of caches*. We argue that this challenge for main memory systems is very similar to efficiently using main memory for disk-based systems. Consequently, we expect some aspects of I/O-efficient algorithms to be transferable to cache-efficient algorithms, so we review prior work in both areas.

3.2.1 *I/O Efficiency*

The main driver of system and algorithm design in the disk-based world is the reduction and hiding of I/O costs, which dominate almost all other costs. While storage space of disk is virtually unlimited, fast main-memory is rather small, so algorithms may be forced to write and read intermediate results to and from disk, possibly repeatedly, if they do not fit into main-memory. To what extend this is necessary primarily depends on the number of groups.

On the one hand side, several variants of SortAggregation were suggested. Epstein [60] was the first to use I/O-efficient sort algorithms to preprocess the input relation. This made it possible to benefit from the many known optimizations of Sorting. Maybe the most prominent example is ReplacementSelection [101, 76]. Depending of the data distribution, this technique improves the initial run production of MergeSort and reduces thus the number of required merge levels.

On the other hand side, Kitsuregawa et al. [98] proposed what is now known as GraceJoin (named after the system it was built in), which consists in recursively partitioning the input relation(s) until each partition can be processed in memory using a hash-based algorithm. DeWitt et al. [54] further improved this approach with their HybridJoin: By producing fewer partitions—just enough to enable in-memory processing of each of them—, they make space for a hash table where a certain fraction of the tuples can be processed immediately, thus smoothing the transition to additional levels of recursion. Both approaches also work for Aggregation and make HashAggregation usable for large output sizes.

Another important insight, specific to Aggregation, is the benefit of "early aggregation", first introduced by Bitton and DeWitt [27]. They modified the comparison operator of MergeSort such that it collapses (aggregates) the two compared tuples if they compare equal. Later Larson [111] extended this idea to other algorithms. Depending on the distribution, early aggregation reduces the amount of data written to disk for later merging considerably. If the output is small enough to fit into memory, only a single run is produced, so the input only needs to be read once. This makes SortAggregation usable for small output sizes.

Larson [112] also suggested a form of early aggregation on a query plan level (which he calls "partial pre-aggregation"): If Aggregation is preceded by a Join, some tuples may already be aggregated *before* or *during* the Join at almost no additional cost, thus reducing the input size of the subsequent Join and final Aggregation.

In parallel, theory was developed to analyze problems and their solutions in the disk-based setup: a variety of "external memory" models [3, 13, 14, 70]. In these models, the only cost of an algorithm is the number of block transfers between main memory and disk. Most upper bound analyses in the work cited above are carried out in this model, even if the original authors did not explicitly state so. For some problems lower bounds have been found—a minimal number of block transfers *any* algorithm necessarily has to do. For example, any algorithm for MultisetSorting requires a number of passes over the input that is proportional to a logarithm of the number of distinct keys [127]. Lower bounds are a useful tool to evaluate or design algorithms: When such a bound is known for a certain problem, algorithms with a matching upper bound are known to be optimal in a provable sense.

To the best of our knowledge, no lower bound exists for Aggregation that allows algorithms to use Hashing. In Chapter 4, we show that the long standing folklore conjecture is true: Aggregation requires a number of passes over the input that is proportional to a logarithm of the number of groups in the general worst case. This proves the asymptotic optimality (or "efficiency") of most practical algorithms presented above.

3.2.2 *Cache Efficiency*

We now turn our attention to prior work about the equivalent of I/O efficiency for Aggregation operators in in-memory database systems. As discussed before, in this type of systems, envisioned as early as in 1984 by DeWitt et al. [54], main memory is the primary storage and disks are only used for recovery. The first popular system was MonetDB [31], later extended to MonetDB/X100 [32], but many systems followed from both research [140] and industry [61, 57, 108, 154] and extend to many other approaches for in-memory data management and processing than traditional relational database systems.

From the point of view of early disk-based systems, doing pure in-memory processing looks like the "easy case". However, several authors have noted [163, 4, 39] that even in disk-based systems, memory access can account for a major part of the processing time besides I/O and have developed cache-conscious algorithms and data structures. In in-memory database systems, this effect is even more extreme: main memory is considered to be the new bottleneck and many architectural aspects are designed to work around this bottleneck [123, 32]. This means that, to a certain extent, the problem of efficient data access has not changed fundamentally, but only shifted up one level in the memory hierarchy.

It is therefore not surprising that algorithms optimized for cache efficiency have certain similarity with the I/O-efficient algorithms discussed before. For example, Shatdal et al. [163] propose an in-memory version of the GraceJoin. Manegold et al. [123] extended this approach: they observe that not only cache-misses are costly, but that TLB misses also play an important role. They propose RADIX-(CLUSTER)JOIN: multiple passes of partitioning on both sides of the join using radices of the join keys as partitioning criterion such that each partition can be processed in cache. The partitioning degree of every pass is kept low enough to avoid cache and TLB misses, but high enough to keep the number of passes low.

Subsequent work has brought variants and extensions of the RADIX-CLUSTERJOIN [96, 6, 20], all optimizing for various bottlenecks of the main memory. Other follow-up work include studies of cache-efficient partitioning [43, 149, 162], confirming it as an important building block for cache-efficient algorithms.

Cieslewicz and Ross [42] and Ye et al. [199] studied various hash-based AGGREGATION algorithms in the in-memory setup. With respect to cache efficiency, they find that hashing works well as long as there are few enough groups such that the hash table fits into cache. If the number of groups is huge, partitioning first can be better. They also show how partitioning can be combined with early aggregation: their algorithm PLAT (*Partition with a Local Aggregation Table*) [199] uses a small hash table for as many groups as the cache can hold while hash partitioning those tuples that belong to the other groups. In a second pass, the partitions are aggregated. DB2 BLU [154] and HyPer [113] use algorithms very similar to PLAT.

While the authors also study other aspects, which we discuss below, their findings with respect to cache efficiency shows the similarity to prior work in disk-based systems: the number of groups is a determining factor and early aggregation can reduce the amount of work in multi-pass algorithms. However, as pointed out before, disk-based algorithms used *recursive* partitioning in order to achieve I/O efficiency for large numbers of groups. What is therefore surprising is that none of the in-memory AGGREGATION algorithms above is recursive, which suggests that they handle large numbers of groups in a sub-optimal way.

Krikellas et al. [104] also study the effect of pre-processing on the cache efficiency of AGGREGATION (although they focus on the integration with Just-in-Time query compilation). They observe that the best way to pre-process depends on the number of groups: MAPPARTITIONING, where each group is mapped to a partition only for itself, is best if the number of groups is small enough such that the map fits into cache. For large number of groups, SORTING is better. They also propose HYBRIDPARTITIONING, where the input is first hash-partitioned, then each partition is sorted, which is slightly faster than plain SORTING. While the resulting HYBRIDHASHSORTAGGREGATION is recursive, the fact that PARTITIONING is used as pre-processing independently of the AGGREGATION logic means that the algorithm misses the potential of early aggregation.

As database practitioners shifted up one level in the hierarchy, theoreticians observed that the external memory models [3, 13, 14, 70] could also be used to study cache efficiency [70, 51, 12, 178, 11]. This means that I/O efficiency lower bounds found in these models are also cache efficiency lower bounds. This makes the lower bound for Aggregation that we present in Chapter 4 interesting for both the disk-based and the in-memory setting, and emphasizes the sub-optimality of the fixed-pass algorithms presented above.

3.3 CPU FRIENDLINESS

While the main memory / disk bottleneck and the cache / main memory bottleneck have certain things in common, there are also a few differences. One difference is the amount of computation that can be done per access to the slower level of the memory hierarchy, i.e., per I/O and per cache-line transfer respectively, which is much smaller in the in-memory setup. Many authors have thus studied ways to reduce computation for in-memory algorithms and data structures, and to make them more "CPU-friendly", which is the focus of this section. In Section 3.9, we also discuss how processing models evolved to become more CPU-friendly.

To the best of our knowledge, Zhou and Ross [205] were the first to suggest to use vector instructions for the implementation of database operators. This allowed them to increase data parallelism and eliminate branches in Scan, ScalarAggregation, NestedLoopJoin, and several types of indices. Follow-up work on the theme of vectorized operators is plentiful and includes the work of Willhalm et al. [196, 195], Lemire and Boytsov [114], and Li and Patel [117] for Scan, the work of Feng and Lo [64] on ScalarAggregation, the work of Kim et al. [96] and Balkesen et al. [20, 19, 21] on Join, as well as the work of Polychroniou et al. [148] on Scan, Hashing, Partitioning, Sorting, and Join. But also the importance of branch elimination for optimal use of super-scalar CPUs was confirmed by others: For example [206] study various compression schemes and their ability to keep the CPU pipeline busy.

The process of making algorithms CPU-friendly is difficult to the point that what could be seen as "implementation details" may invert the outcome of experiments: Blanas et al. [28] compared different Join algorithms and came to the conclusion that the simplest approach, a single hash table without partitioning, had a superior performance to sophisticated partitioning techniques. However, Balkesen et al. [20] were later able to improve the implementation of the latter algorithms by factor three "just" by making the inner loops more CPU-friendly. They thus arrived to the opposite conclusion, namely that partitioning with its increased cache efficiency is superior.

All these techniques aim at reducing CPU costs by implementing algorithms and data structures in a way that modern CPUs can handle them efficiently. The trend of ever increasing vector width and other specialized CPU features will probably make this statement even more true in the future.

Substantial lines of work have studied the implementation of database systems, algorithms, and data structures on even more specialized hardware, such as GPUs (*Graphic Processing Units*, for example by Govindaraju et al. [72]) and FPGAs (*Field-Programmable Gate Array*, for example by Mueller et al. [131]). While these research directions are somewhat related to our work, we consider them as out of scope, and concentrate on current mainstream server hardware instead.

3.4 PARALLELISM

As we discuss in Section 2.3, parallelism plays a fundamental role in efficiently using modern hardware on several levels. But parallelism was already studied in very early database systems and the main ideas developed there are still used today. We thus review work on early parallel database systems in Section 3.4.1, before reviewing more recent work on multi-core parallelism in Section 3.4.2. The subsequent Section 3.5 then continuous with work on even larger systems.

3.4.1 *Early Parallel Database Systems*

Database systems were one of the main users of parallel computers since the early days: In order to overcome the slow disks, more disks were used in parallel (along with compute resources to control them) in order to increase their aggregated bandwidth. Although this made systems more complex, parallel database systems became widespread in academia and industry, with early systems such as Gamma [56], Bubba [33], Grace [98], Volcano [73], and others. We briefly summarize the insights of that time and refer to surveys and books on the topic [55, 76, 143] for more details.

Parallelism can be achieved in different ways for query processing: different queries can be executed at the same time (*inter-query parallelism*), which is easy to achieve in a multi-user system and mainly helps to hide waiting time for I/O; different operators can be executed at the same time, either in a pipelined fashion (*vertical intra-query parallelism*) or among different branches of the query plan (*horizontal intra-query parallelism*); and several instances of the same operator may be run on partitions of the input (*intra-operator parallelism*). A popular means of achieving intra-operator parallelism in early systems was the use of a meta-operator that encapsulated the parallelism such as the EXCHANGE operator in Volcano [73]. This allows using unmodified relational operators, which are instantiated multiple times and connected via EXCHANGE operators, which take care of scheduling and partitioning. The various forms of parallelism are mostly orthogonal and most systems support several or even all of them at the same time, however it is non-trivial to balance the different forms (for example Mehta and DeWitt [129] discuss this challenge). In this work we concentrate on intra-operator parallelism because it is the only form that achieves scalable response time.

Different parallel computer architectures are conceivable, which come with different advantages and disadvantages. Stonebraker [170] compared the most prevalent ones and concluded that *shared memory* (or *shared everything*) had a limited scalability, *shared disk* had difficulty with interference of nodes, and *shared nothing*, though depending on good load balancing, was the overall most cost-effective. For a long time, shared nothing was the prevailing architecture, but as we discuss below, the distinction between them become blurry on modern hardware.

Several parallel AGGREGATION algorithms were proposed for the shared nothing architecture. DeWitt et al. [56] introduced TWOPHASEAGGREGATION, where every node computes AGGREGATION of the local data using a sequential algorithm. All intermediate results are then sent to a central node, which aggregates them to the final output. Graefe [76] later improved upon this algorithm by partitioning the intermediate results among the nodes based on the groups, such that the final aggregation could be computed in parallel by all nodes. Both algorithms work well if the number of groups is rather small. Graefe [76] also proposed to REPARTITION the input by groups among the nodes without local aggregation, which saves overhead in case the number of groups is very large. This complementary behavior is similar to the one of I/O-efficient algorithms discussed above, where different strategies are preferable depending on the number of groups.

3.4.2 *Multi-Core Parallelism*

As discussed in Section 2.3, there is abundant parallelism on several levels in modern hardware. We start with discussing multi-core parallelism, which is nowadays present in computers of all sizes. While we argue that there are strong similarities with the parallel systems built in earlier decades, there are also fundamental differences both in the hardware and the processing models of the database systems.

The first parallel processors to become main-stream could run several threads in a single core in an interleaved fashion (*symmetric multithreading* or *SMT*). Consequently, early work by Garcia and Korth [71] studied the use of SMT to hide the latency of cache misses. This was achieved by executing build and probe phase of a JOIN simultaneously. More recent work took the increasing number of cores into account: Kim et al. [96] showed how to scale the—initially sequential—RadixJoin [123] to multiple CPU cores by decomposing each phase of the algorithm into synchronization-free, thus easy to parallelize, sub-phases. This "bulk-synchronous" approach was subsequently used by other authors for Join [6, 20, 148], Sorting [93, 152, 92, 91, 160, 182, 149], and Aggregation [42, 199], which we discuss in more detail below. The NoPartitionJoin by Blanas et al. [28] constitutes an exception in that it uses a shared hash table for probing instead of bulk-synchronous, independent processing phases, but it was later found out not to be competitive [21]. Since Partitioning is an important building block for decomposing work for different cores, it was also studied in isolation [43, 149, 162].

The work of Cieslewicz and Ross [42] and Ye et al. [199] about Aggregation also studies multi-core parallelism. Their algorithm Independent is essentially the same as TwoPhaseAggregation [76], while PartitionAndAggregate is an in-memory equivalent of the Repartition algorithm by Graefe [76]. Another proposed algorithm is Atomic—an algorithm where all threads perform HashAggregation using a shared hash table made thread-safe with atomic instructions. While this works great on uniform data, it may create *contention* on skewed input. We discuss the remedies proposed by the authors below.

We find it interesting to observe that most algorithms reviewed above use the multiple cores of modern CPUs in a way earlier shared nothing database systems had used the nodes in a network: they split the problem into smaller, independent sub-problems in order to minimize synchronization and maximize the time efficient sequential algorithms can be used. Only a few algorithms (NoPartitionJoin and Atomic) use shared data structures and thus rather resemble solutions for the shared everything database architecture.

With modern hardware, we thus have the architectural choice: we can use the shared last level cache (or, to a degree, main memory) to build shared everything solutions, or private caches (or NUMA local memory) to build shared nothing solutions. The consequences seem to be similar as in distributed systems: Shared everything database systems may suffer from interference (except that it is called "contention" nowadays) while shared nothing systems have the challenge of work balancing. What makes multi-core systems different from distributed systems is the fact that we have the architectural choice *on the same hardware*, since modern CPUs have both private and shared levels of memory. This also makes it possible to mix both models into the same algorithm: Hybrid [42] for example has both a hash table in the private cache of each core and a global one in the shared cache level.

On the software side, the preferred processing model also seems to have changed: the algorithms reviewed above depart from the strict "open-next-close" pipelining model of Volcano [73], where all parallelism is encapsulated in the Exchange operator. Instead, the operator itself is responsible for parallelizing work and synchronizing different threads among each other. This gives the operator more control and thus enables more sophisticated scheduling techniques, such as the ones we employ as well. At the same time, other recent work depart from the Volcano model into a different direction: Leis et al. [113] propose *morsel-driven* parallelism with flexible degrees of parallelism and dynamic scheduling of work during runtime as opposed to the traditional plan-driven parallelism.

Finally, theoreticians have extended the external memory model to account for multi-core hardware: Arge et al. [12] proposed the *parallel external memory* model (PEM), where each of several processors has its own private internal memory and an algorithm is charged for rounds of (parallel) block transfers from any internal memory to external memory. Greiner [79] showed a lower bound for Sorting in this model. The proof we show in Chapter 4 is an extension of his proof, and shows that Aggregation is asymptotically as hard as Sorting in the parallel external memory model.

3.5 COMMUNICATION EFFICIENCY

As we discuss in Section 2.3, parallelizing systems and algorithms beyond the scale of a single processor has the additional challenge of slow communication across the involved compute units. In this section, we study prior work addressing this issue on two levels in the hierarchy of the hardware: Section 3.5.1 presents work on NUMA awareness, while Section 3.5.2 reviews work on parallel database systems in clusters and high-speed-networks.

3.5.1 *NUMA Awareness*

As main-memory becomes larger and the number of sockets it is connected to becomes higher, access latency and bandwidth become more and more non-uniform. This gave rise to work on *NUMA-aware* algorithms and systems in the recent years. The general goal of these algorithms is to reduce the amount of data that it transferred across NUMA boundaries: Wassenberg and Sanders [182] propose to implement RadixSort with a first pass on the highest radix that crosses NUMA boundaries, such that the subsequent passes can be done locally on every NUMA node. This was later extended by Polychroniou and Ross [149] to better support non-uniform distributions. Similarly, Albutiu et al. [6] built a NUMA-aware SortMergeJoin that carefully pre-processes both relations such that they can be shuffled with only sequential reads and such that the subsequent join phase is local and balanced among the nodes. Li et al. [116] showed that data shuffling across NUMA regions can be up to three times faster if it is carefully scheduled compared to a naive implementation, and use the improved shuffling to improve the JOIN of Albutiu et al. [6].

Lang et al. [109] follow a different approach: they make the NOPARTITIONJOIN of Blanas et al. [28] "NUMA-aware" by partitioning the input relations so that they are read only locally, and by interleaving the global shared hash table among all NUMA regions. They claim to achieve a two-fold speed-up compared to the hardware-conscious implementation of the parallel RADIXCLUSTERJOIN of Balkesen et al. [20]. This somewhat contradicts what the findings of other authors suggest, and may be due to imperfect tuning of the competitor: Balkesen et al. [19] later published numbers of their algorithm on a NUMA system that are up to almost four times higher than those reported by Lang et al.

Most of this work on JOIN only applies to AGGREGATION in case of large numbers of groups, i.e., when the REPARTITION algorithm is better than TWOPHASEAGGREGATION. This is also confirmed by the findings of Li et al. [116]: their approach of migrating *the output* to the socket of the respective partitions instead of migrating the partitions is very similar to TWOPHASEAGGREGATION, and clearly superior to partitioning for small number of groups.

Leis et al. [113] incorporate NUMA awareness into the processing framework of HyPer: relations as well as intermediate results are partitioned and every partition is processed NUMA-locally. AGGREGATION is done like in PLAT: tuples are pre-aggregated as long as the result fits into cache and partitioned (across NUMA regions) otherwise. This makes the system behave like TWOPHASEAGGREGATION for small outputs and like REPARTITION for large ones, which is similar to Shatdal's ADAPTIVE algorithm [164]. However, it misses potential for medium sized output just larger than the cache, where TWOPHASEAGGREGATION would probably be beneficial as well.

3.5.2 *Clusters & High-Speed-Networks*

At an even larger scale, distributed database systems were built: Probably, the most popular distributed, disk-based processing system is MAPREDUCE by Dean and Ghemawat [50], which is built for hundreds or even thousands of nodes. It inspired a series of other systems such as an open source implementation called Apache Hadoop [165] and systems with more expressive programming models such as Apache Spark [202]. MapReduce was also used for SQL-like workload: for example Blanas et al. [30] compared different JOIN algorithms on MapReduce. Other systems combining MapReduce and relational query processing include the Greenplum Database [179] and Cloudera's Impala [102].

More traditional distributed database systems are usually built for a slightly smaller scale, typically not more than ten or twenty nodes, which are connected with a high-speed network only slightly slower than NUMA interconnects. Distributed in-memory database systems include VectorWise Vortex [47] and HyPer [155].

Query processing algorithms thus aim at reducing network traffic while balancing it with other costs such as main memory access and CPU costs [24]. Polychroniou et al. [150] propose TrackJoin, which, in a first phase, tracks where the different joining tuples reside, in order to decide heuristically, for the second phase, which side of each match to transfer over the network in order to reduce the total amount of transferred data. NeoJoin by Rödiger et al. [156] has a similar approach, but with a larger granularity: the algorithm first partition both relations on each node and then formulate a mixed integer linear program that finds the partition-to-node assignment with minimal transfer cost. Frey et al. [69, 68] on the other hand built CycloJoin, which rotates partitions of the smaller relation over a ring of nodes in order to join each partition with stationary partitions of the other relation. They observe that, against intuition, the network is not the bottleneck of their algorithm, but the memory bandwidth.

As with NUMA-aware algorithms, the above work on Join only applies to Aggregation if the output is very large and misses the potential of pre-aggregation otherwise. Some systems such as HyPer [155] and Impala [102] solve this problem on a query plan level by adding PreAggregation operators. However, as discussed before, the way it is done either misses potential for output sizes just larger than the cache (in case of HyPer) or adds overhead without benefit if no pre-aggregation happens (in case of Impala).

Finally, as Rödiger et al. [155] point out, high-speed networks and multiple NUMA sockets constitute a network hierarchy, and algorithms have to take *both of them* into account for optimal performance.

3.6 SKEW HANDLING

Until here we have only seen different strategies to deal with different numbers of groups, i.e., with different output sizes. However, also the distribution of the groups may affect runtime and techniques have been developed to handle negative effects or even benefit from certain distributions.

We start again with work in early disk-baded systems. In these systems the so-called *placement skew*, the fact that the *order* of the tuples is in some way non-uniform, may be problematic. For example standard ReplacementSelection presented above has no benefit at all if the input is in reverse order, although it is able to produce runs twice the size of main memory if the groups are distributed uniformly at random. Follow-up work has fixed this problem in several ways: One possibility suggested by Graefe [74] is to use "poor man's normalized keys", i.e., to sort by hash values instead of by the grouping attributes. The hash values then behave like uniformly distributed random values, so runs are again double the size of main memory. Martinez-Palau et al. [126] recently proposed TwoWayReplacementSelection, which is able to benefit from both ascending and descending trends in key values and therefore handles placement skew in a more robust way as well.

Value skew, the fact that some groups occur more frequently than others, in particular if it is combined with placement skew, can be exploited by early aggregation techniques presented above. This is true for SortAggregation style algorithms such as the modified merge sort of Bitton and DeWitt [27], as well as for HashAggregation style algorithms such as HybridAggregation [54]. The more tuples of the same group occur close to each other in the input, the more tuples can be aggregated early on to save work for later processing. Recently Helmer et al. [82] went a step further and designed an algorithm similar to HybridAggregation specifically to benefit from skew: They maintain a hash table with an LRU replacement strategy in order to keep "hot" groups available for early aggregation and partition the other groups for recursive processing.

Skew was also discussed for parallel database systems starting from early work (see Walton et al. [181] for a taxonomy of types of skew for parallel join algorithms). DeWitt et al. [53] compared different Join algorithms for different skew types and degrees. One of the proposed techniques consist in sampling the inner relation to find "splitters" that split the relation into equal pieces. The winner for the high-skew case however was *virtual processor partitioning*. In this technique, much more tasks than processors are created and a heuristic called "Largest Processing Time First" is used for distributing partitions of virtual processors to physical ones. Similarly, Wolf et al. present a parallel SortMergeJoin [197] that subdivides too large tasks until they can be evenly distributed (using the same heuristic as Dewitt et al.), as well as a parallel HashJoin [198], approximates equal work distribution based on partition size estimates. Interestingly, similar techniques were revived in modern disk-based parallel systems such as MapReduce [153, 106, 107]. In short, the main challenge for shared-nothing disk-based algorithms posed by skew is work balancing and the proposed solutions are different ways of clever work distribution.

Since most algorithms proposed for the multi-core setup in the last years adopted a shared nothing design (see discussion above), they also include solutions to the work balancing challenge similar to the ones of parallel disk-based algorithms. First, there are solutions based on splitters: Polychroniou and Ross [149] determine splitters for the first pass of a parallel sorting pass and use sequential RadixSort for the subsequent passes on each resulting partition. Similarly, Albutiu et al. [6, 5] create histograms of both sides of a JOIN in order to find splitters that balances the subsequent partitioning and join costs.

In the main-memory setup, algorithms based on *task queuing* or *work stealing* become possible: The parallel RADIXJOIN of Kim et al. [96] creates more partitions than there are threads in the partitioning phase, such that in subsequent phases, threads can take tasks from a queue of partitions until the work is done, which balances the work dynamically. Polychroniou and Ross [149] use a similar mechanism for their RADIXSORT. The scheme was later improved by Balkesen at al. [20], who add the possibility of *task decomposition* in order to split tasks that would otherwise be too large and thus dominant. Note that task queuing and work stealing benefit from the fact that moving a task from one core to the other has virtually no costs since no data needs to be moved, and its application to NUMA or distributed systems may be limited.

Let us turn our attention to algorithms for AGGREGATION specifically. The ATOMIC algorithm of Cieslewicz and Ross [42] is also vulnerable to *contention* on frequently accessed groups. The authors propose two remedies: one is detecting contended groups and duplicating them [44]. The other one is a HYBRID algorithm, where each thread has a private, cache-sized hash table and only overflowing elements are spilled to a globally shared hash table. Since only infrequently occurring elements are spilled out of the private tables, accesses to the global table do not contend. Since the private hash tables enable early aggregation, the HYBRID algorithm actually benefits from skew as discussed above. The other algorithms of Cieslewicz and Ross [42] and Ye et al. [199] may experience unbalanced work depending on how well they aggregate in the first pass: INDEPENDENT produces perfectly aggregated intermediate results, which is trivial to split up equally; PLAT may aggregate frequent groups in the local hash tables, but this mechanism may not work in “unlucky” distributions, so in corner cases work may not be split up equally; and PARTITIONAND-AGGREGATE does no aggregation in the first pass and is therefore very exposed to skew.

3.7 ADAPTIVITY

The work reviewed above suggests that the question of which algorithm performs the best often depends on various factors, including characteristics of the input data. The traditional approach to build a static execution plan based on estimates of these characteristics has limitations that are known since a long time [119], and which are still not solved today [118]: for a variety of reasons, cost models and estimates may be wrong and thus lead to query plans orders of magnitude worse than the optimum. An important negative result is due to

Ioannidis and Christodoulakis [87], who established that estimation errors propagate exponentially with the number of joins. To alleviate this and other problems, several forms of *adaptive query processing* were proposed. We now present the major results related in this field and refer to surveys of Babu et al. [18] and Deshpande et al. [52] for a more detailed discussion of motivations for and forms of adaptive query processing.

One major line of work studied ways to adapt the query plan during execution. For example Kabra and DeWitt [94], Markl et al. [125], and Babu et al. [17] propose different variations of dynamic query reoptimization. The basic common idea is to collect statistics during execution and to compare them with previously made estimates, such that plans can be re-optimized if the difference becomes too large. Similarly, the learning optimizer *LEO* of Stillger et al. [169] uses statistics of previously executed queries to improve estimates. A different approach was proposed by Avnur et al. [16] who dynamically change the order in which tuples are routed through the operators depending on which order turns out to be faster.

Another line of work, to which we count our own, studied adaptivity within a single operator to make it more robust and thus to ease the decision of the optimizer. For example did the original GraceJoin and HybridJoin algorithms rely on information about the output size from the optimizer to know into which partitions to split the input and how much memory to allocate for the hash table. Nakayama et al. [139] and Kitsuregawa et al. [97] extended HybridJoin with a dynamic way to allocate memory to partitions, which make the information from the optimizer unnecessary.

A more principled way was taken recently by Graefe [75], who designed a sort-based Aggregation algorithm called G-Aggregation or GeneralizedAggregation. The core of the algorithm consists of a clever scheduling of disk pages that makes the algorithm behave like hash-based aggregation for small cardinalities while keeping its advantage of recursive processing for large cardinalities. The algorithm was later implemented by Albutiu et al. [6, 5]. As a single, robust operator for all situations, this may be the closest prior work to what we present in this thesis, but as we discuss below, its I/O-centered design makes it unclear how to transfer it to the in-memory setting.

Shatdal and Jeffrey [164] proposed adaptive algorithms for parallel processing in shared nothing systems. They exploit the complementary behavior of the two algorithms discussed above: TwoPhaseAggregation works well if the number of groups is small, while Repartition works better if the number of groups is large. They either sample to decide on the algorithm beforehand or switch from TwoPhaseAggregation to Repartition when the number of groups observed by the algorithm crosses some threshold. The latter mechanism is quite similar to ours, though less robust in some situations, and we use it for both cache efficiency and communication efficiency. Like us, they also observe that adaptivity can improve skew handling, since nodes with a small number of groups may pre-aggregate, while those with high number of groups do not have to.

While the above work was presented in the disk-based setting, we argue that the main ideas also apply for main-memory algorithms. However, there is also more recent work specifically for the main-memory setup. Chen et al. [40] for example propose InspectorJoin, which collects statistics about the input during its partitioning phase in order to select the most suited strategy for the join phase.

Cieslewicz and Ross [42] also propose an adaptive Aggregation operator: As discussed above, their Atomic algorithm with a single shared hash table and their Hybrid algorithm with an additional small hash table per core have complementary behavior vis-a-vis skew. If a short sampling phase detects skew, the skew-resistent Hybrid is chosen, otherwise Atomic. Furthermore, their algorithms activates optimizations for long runs of the same group as well as Aggregation with Min/Max if they are beneficial respectively.

Finally, it is interesting to note that the Reduce phase of MapReduce is very similar to Aggregation. It is therefore not surprising that Vernica et al. [175] study cache efficiency and adaptation mechanisms for the MapReduce framework [50] remotely similar to ours.

3.8 MEMORY CONSTRAINT

In the era of disk-based systems, it was a typical behavior of pipeline-breaking operators to spill out intermediate results to disk. A SortMergeJoin for example would first create sorted runs, which would then be merged recursively, with all runs being written to disk. As discussed above, this is (sometimes provably) unavoidable, but thanks to large sizes of disks, it was not a big problem. Note that no materialization is needed between pipelinable operators in an iterator-based processing model like Volcano [73].

In in-memory systems however, intermediate results are stored in main memory, which is much more limited. This is a problem in particular for the push-based column-wise processing model, where all intermediate results are materialized. Volcano-style pipelining on vectors of tuples like in MonetDB/X100 [32] reduces this problem, but only for pipelinable operators.

Blanas and Patel [29] study several Join algorithms, with respect to performance and memory footprint. They observe that HashJoin has the lowest memory usage of all algorithms, since it does not have intermediate results. With their implementations they also observe that it is faster than all other compared algorithms. This somewhat contradicts the finding of Balkesen et al. [20], which we suspect to be due to implementation differences.

Zukowski et al. [207] propose BestEffortPartitioning, a pipelinable Partitioning operator. By interleaving the Partitioning operator with its consumers, partial intermediate results can be processed before the entire input has been consumed. As a pre-processing step for Join, Sort, and Aggregation, this can help to reduce memory consumption of other operators. However, as a separate operator, interaction with its consumers is limited, and as we show, integrating pipelined Partitioning with Aggregation more closely can be better in some cases.

Begley et al. [25] propose McJoin (for MemoryConstrainedJoin), which is a BlockNestedLoopJoin: For every block of one relation, they iterate over blocks of the other relation and join the pair of blocks using RadixClusterJoin. This limits the memory usage to the two blocks. McJoin furthermore uses compression to increase the effective capacity of the blocks. Similar to the latter theme, Barber et al. [23] devise a ConciseHashTable in order to reduce memory consumption during Join.

3.9 PROCESSING MODELS

Changes in hardware and dominant workloads have caused major shifts in the processing models of prevailing database architectures. Some of these models pose restrictions on the implementation of operators, which have often been ignored in the discussion of operators in the past. Until the late 90s, the classical iterator model on single records, introduced first in Volcano [73], was used in almost every system. With a shift from mainly transactional workload accessing single entire records to mainly analytical workloads accessing few attributes of whole tables, column stores became popular rather than row stores.[1] Early research systems include MonetDB [31, 124], MonetDB/X100 [32], and C-Store [171] and nowadays all major vendors offer column-store products: Sybase IQ [121], SAP HANA database [61], IBM Blink [22] and IBM BLU [154], Microsoft Apollo/SQL Server [110], and Oracle TimesTen [108].

Another shortcoming of the Volcano model is the interpretation overhead on modern hardware, which was identified by Ailamaki et al. [4] among the first. This insight was another driver for the column-store architecture, where the interpretation overhead is only done once per column and thus completely amortized by the CPU-friendly column-wise processing in tight loops.

In recent years, several authors [104, 140, 67, 138] suggested going a step further by compiling each query into native machine code, i.e.,

1 Our real-world workload study of Section 2.1.2 shows that while indeed only a part of the attributes are typically accessed, the absolute number of columns that this represents is often much larger than textbook scenarios suggest. This does not question the motivation of column stores, but emphasizes the importance of supporting large numbers of columns.

to use "Just-in-Time compilation" (JiT). This eliminates the need for interpretation entirely. The resulting code only contains the CPU instructions that are strictly necessary for the computation of the query result and is thus very efficient.

Finally, as discussed before, processing models were also affected by the way parallelism is handled: the classical EXCHANGE operator is mainly abandoned in favor for intra-operator or morsel-driven [113, 155] parallelism.

When authors present JOIN algorithms, they most often concentrate on joining the two join attributes, producing a set of row identifiers of matching pairs of rows. However, as Manegold et al. [124] found out, the (late) materialization of the result can outweigh the costs of the JOIN. This is mainly due to random memory access in a naive materialization scheme, which they improve with a cache-conscious scheme. Abadi et al. [2] also compare different materialization schemes, finding that sometimes early materialization is better than late materialization and sometimes vice versa.

What is different with AGGREGATION is the fact that we can do early aggregation, and that this is necessary for optimal performance. However, early aggregation with early materialization necessarily implies interpretation overhead with Volcano iterators. Late materialization is not possible with early aggregation either. Furthermore, column stores are based on the principle to process the different columns separately, so different attributes of the same tuple have to be aggregated to the same group independently of each other. Therefore, the group of a tuple has to be tracked when the key column is processed and made available for the other columns. This is not possible with algorithms were tuples make many fine-granular movements, such as HYBRID of Cieslewicz and Ross [42] or sorting networks such as those used by Balkesen et al. [20]. We discuss the issue in more detail in Section 5.3.3.

With Just-in-Time compiled query plans, no restrictions seem to apply, but little work has been done on AGGREGATION specifically. Only the work of Krikellas et al. [104] introduced earlier was carried out in this processing model explicitly.

3.10 SUMMARY

As we discuss in this chapter, AGGREGATION is a very well understood problem with many authors studying it for several decades. It comes to no surprise that the main ideas reoccur: the groundwork was laid by early, disk-based systems, and more recent work concentrated on adapting solutions to new developments, such as new hardware or different workloads. At the same time, these trends are also studied in the context of other algorithms such as JOIN, as well as database system architecture in general. Hence, general techniques to cope with new hardware are well understood as well. To summarize, the challenges of implementing an AGGREGATION operator are well-known and solutions have been proposed for each of them, which we discuss in isolation in the previous sections. However, the challenges have interactions between them, which makes it hard to address them all at the same time. In the remainder of this section, we show that no solution proposed to date is complete in this sense.

To that aim, we summarize the discussion of this chapter in a back-of-the-envelope comparison of the most complete solutions vis-à-vis the challenges in Table 1. For the challenge of cache-efficient processing, we distinguish between the behavior at high and low locality, and assume that the algorithm gets perfect predictions from the optimizer. Furthermore, we assume that known parallelization techniques are applied to all algorithms, even if the original authors do not mention parallelization. Finally, we only compare how well algorithms can be integrated into column wise processing, since integrating them into a Volcano-style model is a non-goal and integrating them into a JiT-based model is easy for any algorithm.

We consider the following solutions for the comparison:

SORTMERGE: Textbook SORTMERGEAGGREGATION without early aggregation,

SORTMERGE+EA: SORTMERGEAGGREGATION with early aggregation as proposed by Bitton and DeWitt [27],

GRACE: Textbook HASHAGGREGATION with prepartitioning (named after the GRACEJOIN [98]),

ADAPTIVESN: The adaptive parallel AGGREGATION operator of Shatdal and Naughton [164], based on TWOPHASEAGGREGATION [56] and REPARTITION [76],

GENERALIZED: GENERALIZEDAGGREGATION as proposed by Graefe [75],

LRU: The skew-aware algorithm based on a hash table with LRU-eviction by Helmer et al. [82],

RADIXCLUSTER: A hypothetical AGGREGATION operator built on radix clustering like the original RADIXCLUSTERJOIN by Manegold et al. [123] and parallelized like the parallel version of the RADIX-CLUSTERJOIN of Kim et al. [96],

ATOMIC: The ATOMIC algorithm of Cieslewicz and Ross [42] (which is very similar to NOPARTITIONAGGREGATION built like NOPARTITIONJOIN [28, 109]),

ATOMIC+CONTDET.: The same ATOMIC algorithm extended by contention detection [44],

	Cache-Eff.		CPU-Friendly	Parallel	Skew	Adaptive	Mem. Const.	Col. Store
Algorithm	High Loc.	Low Loc.						
SortMerge	−−	++	−	++	−	−−	−−	−
SortMerge+EA [27]	++	++	−	++	++	++	−	−−
Grace [98]	++	++	−	++/−	+/−−	−−	+/−−	++
AdaptiveSN [164]	++	++	−	+	+	+	+/−−	++
Generalized [75]	++	++	−−	++	+	++	−	−
Lru [82]	++	+	−	−	++	−	−	−−
RadixCluster [123, 96]	++	++	++	++/−	−−	−−	−−	++
Atomic [42]	++/−−	−	++	++/−	+/−−	−−	++	+
Atomic+ContDet. [199]	++	−	++	++	+	−	++	−
Radix+Atomic	++	++	++	++/−	+/−	−	++/−−	++/−
Independent [42]	++	−−	++	++	+	−−	−−−	++
Part. & Agg. [199]	−−	+	+	++/−	−−	−	−−	++
Hybrid [42]	++	+/−	+	++	++	+/−	++	−−
AdaptiveCR [42]	++	+/−	++	++	++	+/−	++	−−
Plat [199]	++	+/−	+	++	+	+/−	−−	−
HybridHashSort [103]	++	+/−	+	++/−	+	−	−−	++
HwConscious [20, 19, 21]	++	++	++	++	+	+	−	−−
Mpsm [6]	++	++	++	++	++	++	−	−−
Shuffle [116]	++	−−	++	++	+	−−	−−	++

Table 1: Comparison of existing Aggregation algorithms

RADIX+ATOMIC: A hypothetical system with RADIXCLUSTER and ATOMIC depending on an optimizer decision and using BESTEFFORTPARTITIONING [207],

INDEPENDENT: The INDEPENDENT algorithm of Cieslewicz et al. [42] (which is really a modern TWOPHASEAGGREGATION [56]),

HYBRID: The HYBRID algorithm of Cieslewicz and Ross [42],

ADAPTIVECR: The ADAPTIVE algorithm of Cieslewicz and Ross [42],

PART. & AGG.: The PARTITIONANDAGGREGATE algorithm of Ye et al. [199] (which is really a modern version of Graefe's REPARTITION [76]),

PLAT: The PLAT algorithm (*Partition with a Local Aggregation Table*) of Ye et al. [199] (which is now used in DB2 BLU [154] and HyPer [113, 155] and really a modern version of HYBRIDJOIN [54]),

HYBRIDHASHSORT: The HYBRIDHASHSORTAGGREGATION algorithm of Krikellas et al. [103],

HWCONSCIOUS: A hypothetical SORTMERGEAGGREGATION with early aggregation built with hardware-conscious techniques like the SORTMERGEAGGREGATION of Balkesen et al. [20, 19, 21],

MPSM: A hypothetical AGGREGATION algorithm built like the MPSM-JOIN of Albutiu et al. [6] augmented with early aggregation,

SHUFFLE: The AGGREGATION operator based on NUMA-aware shuffling by Li et al. [116].

As Table 1 shows, some algorithms reviewed in this chapter are either designed for high or low locality. However, many also perform reasonably well with both. What is striking is that in particular new algorithms often do not scale to large output sizes. Concerning the CPU friendliness, it seems understandable that older algorithms designed for the disk-based setup are less suited for modern hardware than more recent proposals. Most algorithms can be reasonably well parallelized, however a few suffer from either contention or unbalanced work in case of skew. Furthermore, most algorithms seem to rely on external components such as the optimizer to employ them only in situations where they work well, for only a few of them are adaptive. Working under constrained memory does not seem to be a highly investigated challenge, as most algorithms performing well in this situation as a side effect of their simplicity rather than through sophistication. Finally, many algorithms that are otherwise promising do not fulfill the requirements of the—somewhat special—column-store architecture.

The following algorithms are among the most complete; however all of them leave at least one of the challenges unaddressed. GENERALIZEDAGGREGATION [75] is I/O-efficient for any input distribution and output cardinality, but the sophisticated mechanisms cannot be easily adapted for cache efficiency (see Section 3.7). A hypothetical combination of ATOMIC and RADIXCLUSTERAGGREGATION could cover small and large output cardinalities, but would rely on good optimizer predictions to distinguish the two cases. The algorithms HYBRID, ADAPTIVE, and PLAT of Cieslewicz and Ross [42] and Ye et al. [199] address most of the challenges we identified. However, they are not suited for the column-store architecture (see Section 3.9) and lack efficient handling of very large numbers of groups due to a fixed number of passes. We prove that this is not optimal in Chapter 4 and compare these algorithms with ours in Section 5.8. Finally, hypothetical sort-based algorithms with early aggregation, for example built like the JOIN algorithms MPSM and HWCONSCIOUS, address almost all our challenges, but are still to be built and many of their low-level optimizations do not work in column stores either (see Section 3.9).

To conclude, none of the AGGREGATION algorithms proposed to date solve all the challenges at the same time in a satisfying way. Bridging this gap is the main goal of this thesis.

Part II

THEORY

Theory is a contemplative and rational type of abstract or generalizing thinking, or the results of such thinking.

— Wiktionary [194]

4 EXTERNAL AGGREGATION AND RELATED PROBLEMS

AGGREGATION (or the REDUCE of MAPREDUCE) is a problem that is very well understood in practice. The database community has developed a series of upper bounds, which often coincide with the SORTING bound. It looks like there is a folklore conjecture that this is optimal for cache-efficient processing, but no formal lower bounds are known. At the same time, there are corner cases where these bounds are broken. In this chapter, we study several variants of AGGREGATION on several slightly different external memory models and prove lower bounds in all of them. In many cases, the SORTING bound is indeed a lower bound of AGGREGATION, so we prove the long standing conjecture of the database community. In some models and in some parameter ranges however, we get lower bounds (and often matching upper bounds) below the SORTING bound. Comparing these models shows us what features practical algorithms need in order to beat the SORTING bound, thus providing us in particular and the database community in general with timeless implementation guides.

This chapter is partially based on work that we published previously in [133].

4.1 INTRODUCTION

In the previous chapter, we have seen that a major challenge for implementing relational operators including AGGREGATION is *cache efficiency* (or I/O efficiency in disk-based systems). In this chapter we study this challenge from a theoretical point of view. We use tools that were developed to analyze the cache efficiency of problems and algorithms, namely a variety of *external memory models*.

4.1.1 *Motivation*

We motivate the study with simple and well-known analyses of the two textbook algorithms SORTAGGREGATION and HASHAGGREGATION. This will make us familiar with the external memory model and give some intuition for the complexity of the algorithms. More importantly it will help us to understand the known bounds of related problems, the subtle differences of the various models, and the implications of their choice on the results, as well as the contributions in the subsequent main part of this chapter.

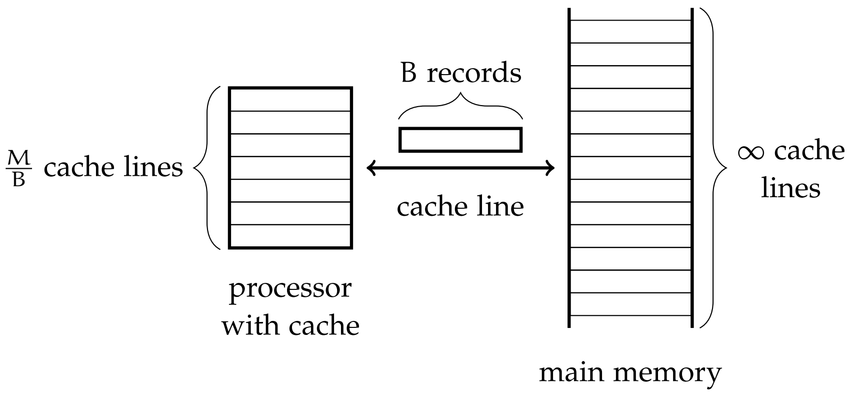


Figure 5: The external memory model [3]

4.1.1.1 *The External Memory Model*

We use the original external memory model of Aggarwal and Vitter [3] for this analysis, which is illustrated in Figure 5. In this model, a computer consists of a *cache* with limited capacity and *main memory* with unlimited capacity. An algorithm can only do computations with records that are in cache; and records can be transferred from cache to main memory and vice versa only in form of entire *cache lines*. For AGGREGATION the model has the following parameters for the input data and the cache:

$$\begin{aligned} N &= \text{number of input records} \\ K &= \text{number of groups in the input} \\ M &= \text{number of records fitting into cache} \\ B &= \text{number of records per single cache line} \end{aligned}$$

Note that the output will be of size K. The costs of an algorithm are the number of cache line transfers in the worst case—computations and access to the cache are free.

4.1.1.2 *Analysis of Sort-Based Aggregation*

We start with the analysis of SORTAGGREGATION. We use BUCKETSORT because the analysis is simple and the result is valid for any other cache-efficient sort algorithm. BUCKETSORT recursively partitions the input into buckets until the data is sorted. Then a final pass over the data aggregates the rows of the same group, which reside in consecutive memory locations in the sorted input.

We use the tree representing the recursive calls of the algorithm for the analysis of SORTAGGREGATION, which we develop in three iterations. The first, simple iteration of the analysis works as follows: Since we can sort each cache line for free before we write it, the recursion stops when all partitions have size B, so there are as many leaves in the call tree as there are cache lines in the input: $\frac{N}{B}$. Furthermore,

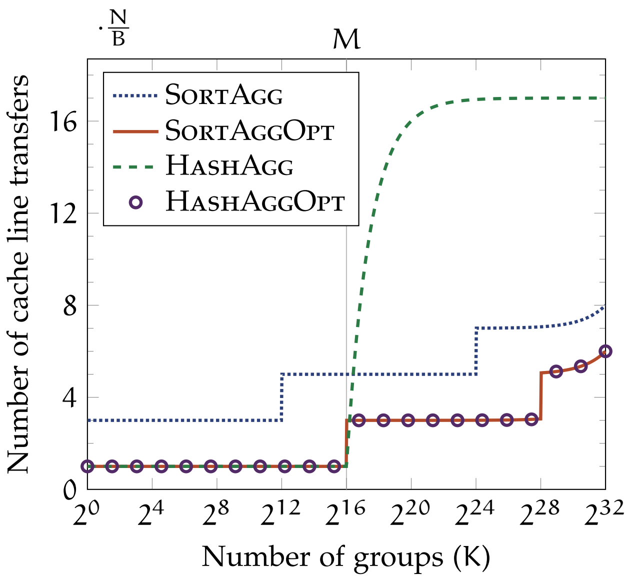


Figure 6: Comparison of aggregation algorithms in the external memory model for $N = 2^{32}$, $M = 2^{16}$, and $B = 16$.

the tree has degree $\frac{M}{B}$ since the number of partitions is limited by the number of buffers that fit into cache. This means that if we assume that the tree is somewhat balanced, it has a height of $\left\lceil \log_{\frac{M}{B}} \frac{N}{B} \right\rceil$. Since the input is read and written once per level of the tree and the subsequent aggregation pass reads the input ($\frac{N}{B}$) and writes the output once ($\frac{K}{B}$), the overall costs of SortAggregation are roughly:

$$\textsc{SortAggStat}(N, K) = 2 \cdot \frac{N}{B} \cdot \left\lceil \log_{\frac{M}{B}} \frac{N}{B} \right\rceil + \frac{N}{B} + \frac{K}{B}.$$

The analysis is slightly simplified because it assumes a static depth of the call tree independently of K. In a second iteration, we can make the analysis more precise by taking into account the fact that the keys form a multiset in the cases where $K < N$. In this case the recursion actually stops earlier than for the case where $K = N$. In fact, the call tree only has $\min(\frac{N}{B}, K)$ leaves, at most one for each partition, so SortAggregation needs the following number of cache line transfers:

$$\textsc{SortAgg}(N, K) = 2 \cdot \frac{N}{B} \cdot \left\lceil \log_{\frac{M}{B}} \left(\min \left(\frac{N}{B}, K \right) \right) \right\rceil + \frac{N}{B} + \frac{K}{B}.$$

The analysis for DistributionSort, SampleSort, QuickSort, and RadixSort all have (at least asymptotically) the same complexity with an analysis very similar to the above, first shown by Aggarwal and Vitter [3]. The same is true for variants of HeapSort [183] and Sorting with buffer trees [10].

Figure 6 plots the number of cache line transfers as function of K for $N = 2^{32}$, $M = 2^{16}$, and $B = 16$, which are typical values for modern CPU caches. For small K, SortAggregation needs one pass for sorting and one pass for aggregating. For larger K, the number of passes only increases logarithmically. Due to the large base and the rounding, the logarithm has only values $\in \{1, 2, 3\}$ in our plot (corresponding to the steps) and is never larger than four in most realistic settings. Only for very large K, where K gets close to N, the $\frac{K}{B}$ cache line transfers for writing the output become noticeable.

In the third iteration of the analysis, we make a small modification to SortAggregation: we merge the last BucketSort pass with the final Aggregation pass, i.e., instead of writing a cache-line to memory when the buffer of a partition runs full, we aggregate the elements of this cache-line to make space. Since there are few enough groups left in the last pass, this produces the final result of the current bucket in cache and thus completely eliminates one pass over the entire data. Furthermore, it allows us to hold a factor B more partitions (M instead of $\frac{M}{B}$), so there are now only $\frac{K}{B}$ leaves in the call tree of the algorithm. With this analysis and a small reorganization of the formula, the number of cache line transfers made by SortAggregationOptimized is the following:

$$\textsc{SortAggOpt}(N, K) = \frac{N}{B} + 2 \cdot \frac{N}{B} \left(\left\lceil \log_{\frac{M}{B}} \frac{K}{B} \right\rceil - 1 \right) + \frac{K}{B}.$$

The first and the last term correspond to reading the input and writing the output respectively. The second term corresponds to writing intermediate results and reading them again in the next level of recursion.

Figure 6 plots the costs of SortAggregationOptimized. It shows that the optimization eliminates an entire pass and slightly delays the necessity of an additional pass due to better cache usage in the last pass. In particular, for $K < M$, the algorithm just reads the data once and calculates the result in cache.

4.1.1.3 *Analysis of Hash-Based Aggregation*

We now analyze the number of cache line transfers that HashAggregation needs. Apart from the $\frac{K}{B}$ cache lines for writing the result, the algorithm just needs the $\frac{N}{B}$ cache line transfers for reading the input—as long as the resulting hash table fits into the cache, i.e., $K < M$, and assuming that intermediate aggregates can be saved in a state of size $\mathcal{O}(1)$ like above. In the other case, if $K > M$, even with a perfect cache and without hash collisions, only a fraction of $\frac{M}{K}$ rows can be in the cache at the same time, so every access to one of the other rows produces a cache miss ($= 1$ write $+ 1$ read). The overall number of cache line transfers is therefore:

$$\textsc{HashAgg}(N, K) = \frac{N}{B} + \begin{cases} \frac{K}{B} & \text{if } K < M, \\ 2 \cdot \left(1 - \frac{M}{K}\right) \cdot N & \text{otherwise.} \end{cases}$$

Figure 6 shows the costs of HashAggregation: As long as the output K is small enough to fit into the cache, HashAggregation is really fast. However, as soon as the cache cannot hold the output anymore, HashAggregation triggers a cache miss for almost every input row, so the number of cache line transfers explodes.

A common optimization to overcome this problem is to (recursively) partition the input by hash value and to apply HashAggregation on each partition separately. Since each partition contains only a part of the groups, i.e., since K is reduced, this makes the algorithm work in cache. However, the partitioning also entails costs, which are the same as the partitioning of bucket sort. Consequently, the analysis works the same way as the one for SortAggregationOptimized: $\left\lceil \log_{\frac{M}{B}} \frac{K}{B} \right\rceil - 1$ partitioning passes plus the reading ($\frac{N}{B}$) and writing ($\frac{K}{B}$) of the HashAggregation pass of the partitions. In total HashAggregationOptimized needs the following number of cache line transfers:

$$\textsc{HashAggOpt}(N, K) = 2 \cdot \frac{N}{B} \left(\left\lceil \log_{\frac{M}{B}} \frac{K}{B} \right\rceil - 1 \right) + \frac{N}{B} + \frac{K}{B}.$$

Figure 6 shows that HashAggregationOptimized has the same cost in terms of number of cache line transfers as the optimized sort-based aggregation above.

4.1.1.4 *From Upper Bounds to a Lower Bound*

Our analysis shows that SortAggregation and HashAggregation have indeed a complementary behavior if implemented naively: HashAggregation performs better if the number of groups is small, while SortAggregation is more efficient in the other case. However, the respective drawback of both algorithms can be removed if each of them includes a simple, well-known optimization: doing the aggregation pass together with the last sorting pass and recursive partitioning as preprocessing respectively. This suggests that—at least on this level of abstraction—there is no such thing as a duality between Hashing and Sorting. In terms of cache line transfers, the two approaches are actually the same.

Whether this is an intrinsic property of Aggregation itself rather than being a property of specific algorithms, i.e., whether there is a lower bound on the cache line transfers needed to compute an Aggregation query, has long been an open question. Since most algorithms proposed so far match the complexity of our analysis and no algorithm is known to date that requires fewer cache line transfers, there is an—at least implicit—folklore conjecture saying that there is indeed such a bound and that it matches the bound of MultisetSorting. In this chapter we prove that this long standing conjecture is actually true in many relevant cases. In the rest of this section, we describe the technical challenges in modelling and proving it.

4.1.2 *Known Results and Challenges*

The way we used the model in the motivating example was slightly informal: In order to be able to prove lower bounds, Aggarwal and Vitter [3] defined their model such that the only thing algorithms can do is to *move* (or *permute*) records and the records to them are *indivisible*. It is therefore also referred to as the *permutation model*. This restricted set of operations makes it possible to *count* the different number of permutations achievable with a certain number of cache line transfers and thus bound the progress any algorithm can make. This way Aggarwal and Vitter showed that PERMUTING needs at least the following number of cache line transfers in the worst case (using the same parameters as before):

$$\Theta\left(\min\left(\frac{N}{B}\cdot\left\lceil\log_{\frac{M}{B}}\frac{N}{B}\right\rceil, N\right)\right). \tag{3}$$

Since SORTING is a permutation, SORTING has the same lower bound. They also showed variants of DISTRIBUTIONSORT and MERGESORT that fit this bound, i.e., the bound in Equation 3 is asymptotically tight.

The simplicity of the model makes it very general: Since many other cache and I/O models also move indivisible records, the permutation model is a subset of them and lower bounds found here also hold in the other models. However, Aggarwal and Vitter only presented results for problems on sets, where all elements are assumed to be distinct. As we have seen in the motivating example, doing the same for AGGREGATION would ignore an important aspect, namely the number of groups in the input.

Matias et al. [127] extended the analysis of Aggarwal and Vitter to a variant of SORTING on multisets: They studied BUNDLESORTING, where an algorithm is asked to permute a multiset such that records with equal keys are placed in “bundles", i.e., adjacent to each other, but with an arbitrary order inside each bundle. For most parameter ranges, they use a counting argument similar to Aggarwal and Vitter, to show that BUNDLEPERMUTING and thus BUNDLESORTING need at least the following number of cache line transfers in the worst case:

$$\Theta\left(\frac{N}{B}\cdot\left\lceil\log_{\frac{M}{B}}\frac{K}{B}\right\rceil\right). \tag{4}$$

Furthermore, they present a modified DISTRIBUTIONSORT that matches this bound, i.e., the bound in Equation 4 is asymptotically tight.

While this is a step closer to our problem at hand, it still has an important shortcoming: Records cannot be combined (“aggregated”) or even dropped. A way to overcome this limitation was recently proposed by Jacob et al. [88]: They define an external memory model where records are indivisible atoms, along with a (black box) *semigroup operation* on these atoms. This makes it possible to model problems such as AGGREGATION, were records need to be combined, while keeping the model simple enough to use counting arguments such as those used in the permutation models reviewed above.

The external memory model of Aggarwal and Vitter was also extended to account for on-chip parallelism in the multi-core era: Arge et al. [14] proposed the *parallel external memory model* (*PEM*), which consists of P processors, each with a private cache of capacity M, and a shared, unlimited main memory. They present parallel versions of DistributionSort and MergeSort with an optimal speed-up (for reasonable values of P), i.e., with the following number of cache line transfers:

$$\Theta\left(\frac{N}{PB} \cdot \left\lceil \log_{\frac{M}{B}} \frac{N}{B} \right\rceil\right). \tag{5}$$

The original paper also derives the optimality of the complexity in Equation 5 directly from the complexity of Sorting in Aggarwal and Vitter's model. This turned out to be erroneous [167], however Greiner [79] provided a proof for the same bound using a counting argument similar to the ones used before [3, 127]. Analogous analyses in the PEM model for problems on multisets are not known to date.

Alternative external memory models were developed that are only slightly more restrictive, but yield lower bounds for a large set of practical problems. Arge et al. developed two techniques [13, 14] to derive a lower bound on cache line transfers from a lower bound of comparisons. This makes many lower bounds in the external memory model folklore. For example the well-known $n \log n$ comparison lower bound of Sorting translates into (the main term of) Equation 3 of Aggarwal and Vitter. Similarly, a bound for MultisetSorting can be derived, where a sorted set is produced followed by the duplicates in arbitrary order. Munro et al. [136] showed that it has a comparison lower bound of

$$N \log N - \sum_{i=1}^{K} N_i \log N_i + \mathcal{O}(N), \tag{6}$$

which translates into the following bound in the external memory Turing machine model:

$$\Theta\left(\max\left(\frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B} - \sum_{i=1}^{K} \frac{N_i}{B} \cdot \log_{\frac{M}{B}} N_i, \frac{N}{B}\right)\right). \tag{7}$$

Arge et al. [13] give an analysis of a MergeSort that filters out duplicate records during the merging and matches this bound for MultisetSorting and DuplicateRemoval. Farzan et al. [63] later match this bound for BundleSorting as well with an algorithm called FunnelSort, so the bound of Equation 7 is asymptotically tight for all these problems.

Although the models proposed by Arge et al. look very general, they are restricted in a very subtle way: The fact lower bounds are derived from comparison lower bounds *forces* algorithms in these models to sort the input—even if this is not actually required such as with AGGREGATION. This is even true in the more general of the two models, the *external memory Turing machine* model [14], where the algorithm can make arbitrary calculations (by executing a Turing machine) based on the content of the cache in order to decide which cache line to read or write next. However, the machine cannot store anything else than input records; in particular, it cannot create new records, which makes aggregating impossible—like in the permutation model. More importantly, the lack of storage space for non-input records also makes it impossible to use hash functions in this model.[1] In contrary to the permutation model, lower bounds in the external memory Turing machine model thus do not necessarily hold for algorithms using hashing, which could be more powerful (for example by taking the hash function as "address"). Whether this is the case for AGGREGATION was previously not known.

Finally, we note that some bounds given for problems on multisets express the complexities in terms of how many groups there are (K) while others express them in terms of how often each element occurs, i.e., in terms of their *multiplicities* N_i, $i = 1, \dots, K$. The bounds given in the latter form are more precise and can be simplified to the former form as explained in Appendix A.1. A lower bound in terms of K represents those multiplicities that incur the highest cost, i.e., it is the maximization of the bound in terms of multiplicities, which is the case where all $N_i = \frac{N}{K}$. Since analyses in terms of multiplicities are more complicated, they have not been found for all problems and models yet. This includes MULTISETSORTING and BUNDLESORTING in the permutation model, where the best known bound, due to Knudsen and Larsen [100], is only non-trivial for a restricted set of values for N, M, and B.

Table 2 summarizes the known results discussed in this section in various models. The seminal permutation lower bound by Aggarwal and Vitter for SORTING of sets were extended by lower bounds for problems on multisets, such as MULTISETSORTING, BUNDLESORTING, and DUPLICATEREMOVAL [63, 127]. However, in the latter bounds, comparison-based arguments were used in many cases, the bounds were often not expressed in terms of the multiplicities of the records, and no lower bound for AGGREGATION was given. In the multi-core setting, only SORTING was studied [12, 79]; no formal bounds for problems on multisets were given yet.

1 A hash function needs to be a random member of a family of functions, which needs to be stored during the runtime of the algorithm. We discuss this issue in more detail in Section 4.2.5.

Parallelism	Input	Bound	Problem	Multiplicities	Comparisons	Reference
sequential	set	upper & lower	SORTING	–	no	Aggarwal and Vitter [3]
sequential	multiset	lower	all except AGGREGATION	yes	yes	Arge et al. [13] + Munro et al. [137]
		upper	MULTISETSORTING DUPLICATEREMOVAL	yes	–	Arge et al. [13]
		upper & lower	BUNDLESORTING	no	no*	Matias et al. [127]
		lower	BUNDLESORTING	yes#	–	Knudsen and Larsen [100]
		upper	BUNDLESORTING	yes	–	Farzan et al. [63]
		upper	AGGREGATION	no	–	Section 4.1.1
		upper	AGGREGATION	yes	–	Section 4.2.4
		lower	AGGREGATION	yes	yes	Section 4.2
parallel	set	upper	SORTING	–	–	Arge et al. [12]
		lower			no	Greiner [79]
parallel	multiset	upper†	all	yes‡	–	Section 4.3.3
		lower		yes	no	Section 4.3.2

*Except for small K. #Only for restricted N, M, and B. †The bounds are only tight for a moderate number of processors P. ‡Except for BUNDLESORTING.

Table 2: Results of upper and lower bounds of AGGREGATION and related problems in different external memory models.

4.1.3 *Contributions*

In this chapter, we study AGGREGATION and related problems on multisets in two of the external memory models, namely the parallel external memory (PEM) model and the external memory Turing machine model. By comparing the bounds of different models and different problems, we can get an intuition about what makes the problems hard. Concretely, we make the following contributions.

- We extend the external memory Turing machine model such that payloads can be attached to records while preserving the property that bounds in the model can be derived from comparison bounds. This allows the machine to run AGGREGATION and enables us to derive a worst-case lower bound for AGGREGATION in this extended model.
- We show how to model AGGREGATION, DUPLICATEREMOVAL, and MULTISETSORTING where *dropping* records is allowed in the permutation model in a way that allows proving lower bounds.
- We give an asymptotic worst-case lower bound in terms of multiplicities for AGGREGATION, DUPLICATEREMOVAL, MULTISETSORTING, and BUNDLESORTING in the parallel external memory model. While these bounds are completely new in the PEM model, they are also more precise or—since they do not rely on comparisons—stronger than previously known sequential bounds if instantiated in the single processor case. Our bound for BUNDLESORTING also corrects a small erratum in the previously known bound [127].
- We present algorithms for AGGREGATION, DUPLICATEREMOVAL, MULTISETSORTING, and BUNDLESORTING in the parallel external memory model that match the lower bounds for the interesting parameter ranges of the model in terms of multiplicities (except BUNDLESORTING, where the bounds match only in terms of K).
- As some bounds are not tight for all parameter values or in terms of multiplicities, we give a list of open problems for further research.

As guideline for our practical implementation in the subsequent chapters of this work in particular and for the database community in general, we can gain the following concrete insights about AGGREGATION:

- Under realistic assumptions, the folklore conjecture is true: AGGREGATION is as difficult as MULTISETSORTING. This is true precisely in terms of multiplicities of the keys and either with one or multiple processors. This implies that instances with a small number of groups or with unevenly distributed keys are easier than those with a large number of groups or uniform key distribution. Furthermore, linear speed-up can be achieved by adding processors and this is optimal.

- Algorithms can beat the MultisetSorting bound for very large K—a hard case for a Sort algorithm—, but only if the order of the output records fundamentally depends on their order in the input. This is not the case for common hashing schemes, where the output order is given by the hash function. However, there are techniques that can achieve the better bound in practice.

- MultisetSorting is also not optimal on degenerated machines where B and M are extremely small compared to N. This is relevant in practice in situations where the external memory model is basically not applicable. The better bound only holds in the permutation model though, so techniques like integer sorting or hashing become essential in these cases. Hashing can thus be indeed more powerful than sorting.

- The bounds rely on the fact that records are indivisible, so they do not apply if we give algorithms access to the "inner information" of records. This can be exploited in practice for example for aggregation functions like `COUNT`, `MIN`, `MAX`, and `ANY`.

- Grouping and Aggregation as well as MultisetSorting and BundleSorting have the same cache complexity respectively. Whether or not we keep duplicate records, or drop or aggregate them does therefore not make an asymptotic difference. However, dropping and aggregating can not only be achieved with simpler algorithms, but also reduces the amount of work by constant factors in known algorithms.

- Our results are limited to aggregation functions that are commutative semigroup operations on single records. Algorithms for `MEDIAN` or for floating point numbers may thus be harder, even asymptotically.

The rest of this chapter is organized as follows. We first show lower and upper bounds of Aggregation in the external memory Turing machine model in Section 4.2. Then we contrast this analysis with upper and lower bounds in the parallel external memory model in Section 4.3. We discuss practical implications of the models for real-world implementations in Section 4.4 and related work in Section 4.5, before concluding the chapter with Section 4.6.

4.2 ANALYSIS IN THE EXTERNAL MEMORY TURING MACHINE MODEL

In this section we derive upper and lower bounds for the number of cache lines transfers required by AGGREGATION algorithms using comparison-based arguments and discuss the implications and limitations of these bounds.

4.2.1 *Overview*

Arge and Milterson [14] developed the *external memory Turing maching model* along with a method that allows deriving lower bounds in that model for a broad range of problems. The core of the method is a theorem that relates the number of cache line transfers required to solve a problem in their external memory model to the number of comparisons required to solve the same problem. Since comparison lower bounds are known for many problems, this yields many external memory lower bounds immediately, including SORTING MULTISETSORTING, BUNDLESORTING, and DUPLICATEREMOVAL. To derive a lower bound for AGGREGATION, we could argue for example that it solves DUPLICATEREMOVAL and thus needs at least the same number of comparisons (which is known), allowing us to derive a bound on cache line transfers using the theorem of Arge and Milterson.

However, there are a few issues that make it questionable whether we can even use the external memory Turing machine model for AGGREGATION. The main problem lies in the way the external memory Turing machine is defined: It mainly consists of an internal and external tape, corresponding to cache and main memory, and a Turing machine that can make arbitrary calculations based on the content of the internal tape in order to decide which records to move from internal to external tape and vice versa. It cannot, however, otherwise modify contents of either tape, i.e., it makes the so called *indivisibility assumption*, which is required to derive lower bounds in the model. This means that we cannot even run AGGREGATION out of the box: Unlike DUPLICATEREMOVAL, AGGREGATION not only moves around records without modifying them, but—by definition—creates new records by aggregating existing ones.

On first sight one might think that the *syntactic reductions* from the method of Arge and Milterson [14, Definition 13] might provide an alternative: a syntactic reduction allows specifying a method to convert records of one problem to those of another such that problems can be reduced to each other even if their records are not of the same type. This even allows associating additional information to the records, such as associating values to keys like in AGGREGATION. Since we can still not create new records, we would model AGGREGATION as a decision problem: provided a multiset of records and their aggregated set as input, decide whether the aggregated set contains the correct result. However, since we can only make computations *based on the content of the internal tape*, we could not compute aggregates of more records than fit there, which is not enough in the general case.

In short, the external memory Turing machine model is not expressive enough to allow Aggregation, due to the fact that it assumes indivisible records. However, as we show in the remainder of this section, only technical changes to the model are sufficient to reconcile aggregating with the indivisibility assumption in a formal way.

To that aim we propose an extension to the external memory Turing machine model. The key idea is to extend the machine by an internal and an external *payload* tape along with a second control unit to process the payload. The two new tapes cannot be accessed by the control unit responsible for determining what records to move next. Records on the two internal tapes and on the two external tapes have a one-to-one correspondence respectively that is maintained when records are moved. These extensions allow us to modify records using arbitrary functions as long we only modify the payload. At the same time, they do not alter any of the properties required for proving the central theorem of Arge and Milterson's method, so it continues to work in the extended version of the model and we can apply it to Aggregation.

Note that the general method for external memory lower bounds based on I/O decision trees by Arge et al. [13] can probably also extended in order to carry payload by doing only technical modifications. The resulting bounds would be the same as the ones obtained in this section.

4.2.2 *The External Memory Turing Machine Model with Payload Extension*

We now review the original external memory Turing machine model from Arge and Milterson [14] and present an extension to it that enables us to run Aggregation.

4.2.2.1 *Review of the External Memory Turing Machine Model*

The external memory Turing machine model consist of four tapes—an internal tape, an external tape, a work tape, and an instruction tape—as well as a finite state machine as control unit. We now go through the different components one by one.

The *external tape* represents main memory. The tape consists of L cells (where L is an arbitrary number independent of w, which is needed for proving the theorem). Each cell is either filled with records from the domain $\{0, 1\}^w$ or with the "blank" symbol $\perp$. The cells are organized in blocks of B records, which represent cache lines. At the beginning of the program, the tape is filled with N records constituting the input. We characterize the input by the number of distinct records K and the multiplicities N_i, $i = 1, \ldots, K$ of each record value. There is no head on this tape, i.e., the finite state machine cannot read from this tape.

The *internal tape* represents the cache. It consists of M cells organized in blocks of size B and containing either records or blanks as the external tape. The tape has a read/write head that the finite state machine can move and use to read and write records from and to the tape. If the machine never writes to this tape, we say that it satisfy the indivisibility assumption. Initially, the tape contains only blanks.

The *work tape* is an infinite tape that the finite control machine can use to make arbitrary computations.

The *instruction tape* is used by the finite state machine to initiate memory operations by writing instructions through a write-only head to the tape. It can be thought of as the memory controller of the machine. When the instruction "READ i" is written to the tape, the first block of the internal tape is replaced by the content of the ith block of the external tape. The instruction "WRITE i" has the opposite effect. "ASSIGN i TO j" overwrites record j of the internal tape with the record i of the internal tape. After every READ and WRITE instruction, the work tape is erased and the head set to its initial position in order to prevent the machine to use it as an "extended internal tape".

The *finite state machine* controls the heads as described above. Furthermore, it can end the execution by entering one of the special states ACCEPT, REJECT (in case of decision problems), and FINISH (in case of construction problems).

The costs of solving a certain problem on a certain input is the number of READ and WRITE instructions written to the instruction tape. This corresponds to the number of cache line transfers.

An external memory Turing machine has to solve a given problem for fixed parameters $B < M < N < L$, K, and N_i, as well as any w. Since N is finite, the machine can remember where it has moved the different records be encoding this information into its states. It is thus quite powerful even though the work tape is erased after every READ and WRITE instruction. However, since the same machine solves the problem for *any* w, it cannot remember the content of the records the same way, so the computations are indeed non-trivial. For a more detailed motivation of the design choices of the machine model, see the original paper [14].

4.2.2.2 *Payload Extension*

We now describe an extension of the external memory Turing machine model that is able to carry payload with the records and to perform arbitrary computations with them.

First, we add an *external payload* tape and an *internal payload* tape. The new tapes have properties similar to those of the internal and the external tapes respectively: They consist of cells containing either payload records or blanks and are organized in blocks of B cells. Payload records are of the domain $\{0, 1\}^u$; and the same machine has to work for any u. Furthermore, the payload tapes have the same sizes as their counterparts: the external payload tape has L cells, while the internal payload tape has M of them. When the computations starts, every of the N input records on the external tape has a payload input record associated with it on the external payload tape at the same position. Finally, the internal payload tape has a read/write head, while the external payload tape has none, like the original internal and external tapes.

Second, we slightly modify the semantics of the instruction tape: whenever a READ, a WRITE, or an ASSIGN instruction is written to the instruction tape and executed, not only the records are moved between the external and internal tapes, but also the associated records on the external and internal payload tapes are moved accordingly. This maintains the association of records and their payload.

Third, we add a *second finite state machine* with its own *work tape* that can work on the internal payload tape. We use the terms "cache control" and "payload processing control" to denote the two different finite state machines. With some exceptions, the two control units have no means to interact: only the cache control can use its work tape, the internal tape, and the instruction tape, while only the payload processing control can use the internal payload tape. None of them can use the two external tapes (since they do not have heads). However, both of them can control the head of the work tape of the payload processing control, though not at the same time: While the cache control is operating, the payload processing control is idle. When the cache control is in a special state "COMPUTE", the payload processing control operates until it enters a special state "DONE". When this happens, the content of the work tape of the payload processing control is erased and its head reset to the initial position and the cache controls continues to operate. Arbitrary information can thus flow from the cache control to the payload processing control, but none can flow into the other direction.

It is clear that we can use an external memory Turing machine with payload extension in order to solve a problem without payloads by initially filling the external payload tape with blanks and ignoring its content after the machine has finished. To formalize this observation, we first define a relationship between the two types of problems:

Definition 1. *Let P be a problem defined on (record, payload record) pairs from the domain $(\{0, 1\}^w, \{0, 1\}^u)$, P' a problem defined on records without payload from the domain $\{0, 1\}^w$, and r a mapping that maps every instance $x = ((k_1, v_1), \dots, (k_N, v_N))$ of problem P to the instance $r(x) = (k_1, \dots, k_N)$ of problem P', then we call P' the* payload reduction *of P if for all instances x of P, $x \in P \Rightarrow r(x) \in P'$.*

For example, DUPLICATEREMOVAL is the payload reduction of AGGREGATION, but also of any other problem where duplicate records from the external tape are removed, no matter what happens to the records on the payload tapes.

Now we establish a theorem about the impact of the payload extension on I/Os:

Theorem 2. *If an external memory Turing machine with payload extension satisfying the indivisibility assumption needs* t *I/Os to solve an instance of* P *in the worst case and* r *is the payload reduction from* P *to* P'*, then there is an external memory Turing machine without payload extension satisfying the indivisibility assumption that solves all instances of* P' *in* t *I/Os in the worst case.*

Proof. We construct an external memory Turing machine that solves P': We take the external memory Turing machine with payload extension that solves P and remove the two payload tapes, the payload processing control unit, and the COMPUTE states of the cache control unit. Note that this applies the payload reduction on the instance present on the external tapes. The resulting machine fits the original definition of an external memory Turing machine: By design, none of the remaining components could gain any information from those that were removed, i.e., the cache control unit was in no way influenced by the payload processing control unit or the content of the payload tapes. Therefore, the sequence of READ and WRITE instructions issued by the constructed machine has not changed, so it does solve $r(x) \in P'$ and needs t I/Os to do so, like the original machine. $\square$

4.2.3 *Lower Bound*

With our payload extension and Theorem 2, we can now apply the theorem of Arge and Milterson [14] to AGGREGATION in a straightforward manner. Concretely, the theorem says that for each external memory Turing machine satisfying the indivisibility assumption and solving an order invariant problem with t I/Os in the worst case, there is a comparison based algorithm solving the problem using less than $N \log B + t\left(B \log \frac{M-B}{B} + 3B\right)$ comparisons in the worst case. This means that a lower bound of comparisons in the comparison model implies a lower bound of I/Os in the external memory Turing machine model. In particular, it implies the following:

Lemma 3. *An order-invariant problem having a complexity of* $N \log N - \sum_{i=1}^{K} N_i \log N_i - \mathcal{O}(N)$ *in the comparison model, has the following complexity in the external memory Turing machine model:*

$$\Omega\left(\max\left(\frac{N}{B}, \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B} - \sum_{i=1}^{K} \frac{N_i}{B} \log_{\frac{M}{B}} N_i\right)\right).$$

Proof. The first term in the max comes from the fact that the external memory Turing machine has to read the input. The second term can be obtained by using said theorem of Arge and Milterson [14, Theorem 4] and solving the following inequality for t:

$$N \log B + t \cdot B \log \frac{M-B}{B} + 3B \geqslant N \log N - \sum_{i=1}^{K} N_i \log N_i - \mathcal{O}(N) \qquad \square$$

Lemma 4. DUPLICATEREMOVAL *requires at least the following number of comparisons:*

$$N \log N - \sum_{i=1}^{K} N_i \log N_i - \mathcal{O}(N).$$

Proof. We use an argument similarly used by Munro and Spira [136, Theorem 3.4] and by Farzan [62, Theorem 2.2.1] before. We argue that the total order of the elements must be known when the algorithm has finished, so the bound for multiset sorting applies. In order to determine whether there is a unique element by only using comparisons, we need to establish for every element in the input whether there is also another, equivalent element. By trying to find equivalent elements, we will also find out which elements are larger or smaller. Thus, the algorithm also determines the total order of the multiset and has consequently the lower bound of multiset sorting. This is known to be the desired bound [137]. □

DUPLICATEREMOVAL is order invariant and can be solved by an algorithm that uses only comparisons (and deletions). Hence, we can put Lemmas 3 and 4 together to obtain the following corollary:

Corollary 5. DUPLICATEREMOVAL *has the following complexity in the External-Memory Turing Machine model:*

$$\Omega\left(\max\left(\frac{N}{B}, \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B} - \sum_{i=1}^{K} \frac{N_i}{B} \log_{\frac{M}{B}} N_i\right)\right).$$

Finally, since DUPLICATEREMOVAL is the payload reduction of AGGREGATION, we can use Theorem 2 and Corollary 5 to establish the following corollary:

Corollary 6. AGGREGATION *has the following complexity in the External-Memory Turing Machine model with payload extension:*

$$\Omega\left(\max\left(\frac{N}{B}, \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B} - \sum_{i=1}^{K} \frac{N_i}{B} \log_{\frac{M}{B}} N_i\right)\right).$$

Our payload extension to the external memory Turing machine thus formalizes the intuition that doing work *in addition* to solving an order-invariant problem does not make the problem easier. In particular, it allows us to show that AGGREGATION has the same complexity as MULTISETSORTING in the external memory Turing machine model.

4.2.4 *Upper Bound*

We now show a matching upper bound for AGGREGATION in the external memory Turing machine model with payload extension. We use a standard MERGESORT, but modify the MERGE routine such that two records are aggregated when they compare equal. In our machine model, this works as follows: whenever the cache control unit finds out that two records in cells i and j on the internal tape are identical, it writes i and j to the work tape of the payload processing unit and enters the COMPUTE state. The payload processing control unit then aggregates the payload records in cells i and j, stores the result in cell i and enters the DONE state. Finally, the cache control unit continues running a standard MERGESORT, but henceforth ignores the record in cell j.

Arge et al. [13] show that a careful analysis of such an algorithm leads to the desired bound. They use a standard MERGESORT to solve DUPLICATEREMOVAL by modifying the MERGE routine similar to us. Their algorithm needs the following number of cache line transfers:

$$\mathcal{O}\left(\max\left(\frac{N}{B}, \frac{N}{B}\cdot\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=1}^{K}\frac{N_i}{B}\cdot\log_{\frac{M}{B}} N_i\right)\right). \tag{8}$$

The same analysis leads to the same bound for AGGREGATION, so the bound of Corollary 6 is tight.

4.2.5 *Discussion*

The analysis in this section gives a partial answer to the question about whether or not the folklore conjecture about AGGREGATION is true: in the external memory Turing machine model, using MULTISETSORTING is optimal to solve AGGREGATION. No algorithm in this machine model can do better.

However, there are reasons not to be completely satisfied with this result. First and foremost, the starting point of our analysis was the observation that HASHAGGREGATION with recursive pre-partitioning requires the same number of cache line transfers as MULTISETSORTING. However, the model does not even allow hashing! As mentioned in the introduction, for hash functions to be effective, we need to draw a member of a family of hash functions at random, and *remember* it during the runtime of the algorithm. However, the external memory Turing machine has nowhere to remember it: the internal tape only contains input records, the work tape is erased after every block transfer, and the finite state has only constant memory with respect to w.[2]

2 A hash function needs to be a random member of a family of functions and the size of the family depends on the size of the universe U (where $|U| = 2^w$). Identifying a member of the family thus needs a number of bits that depends on $\log|U| = w$, which is not constant with respect to w.

In a way this is not surprising: Arge and Milterson prove the relationship between comparison-based problems and problems in their external memory Turing machine model by emulating a Turing machine with comparisons [14, Lemma 1], so their model is not more powerful than what comparisons can do *by definition*. The question whether there is an algorithm faster than MultisetSorting is thus still open, *if it is allowed to use hashing*.

Furthermore, the external memory Turing machine model with its single cache and processing unit does not capture the on-chip parallelism omni-present in today's hardware. Since several processors have a larger combined cache than a single one, but may experience overhead due to synchronization and load-balancing, neither upper bounds nor lower bounds from sequential models may hold on this hardware. By analysing Aggregation in the parallel external memory model in the next section, we overcome both limitations.

4.3 ANALYSIS IN THE PARALLEL EXTERNAL MEMORY MODEL

In this section we analyse Aggregation and a set of related problems in the parallel external memory model. By contrasting the results to those from the previous section and comparing the results of different, similar problems, we can get insights which aspects of the problem and the machine model influence the bounds. Furthermore, we get results applicable for modern multi-core processors.

4.3.1 *Machine Model and Problem Definitions*

We start with formal definitions of the machine model and the problems we want to study.

4.3.1.1 *The Parallel External Memory Model*

We make the analyses of this section in the *atomic parallel external memory model* [88] (*atomic PEM*) of the CREW flavor (concurrent read exclusive write), which is illustrated by Figure 7. Like the original PEM model by Arge et al. [12], the atomic PEM consists of an infinite *main memory* and P processors, each with a private *cache* that can hold M records. We assume $1 \leqslant B < M < N$ and $N > PB$, which is realistic on real machines. Algorithms can only move, copy, or delete records, and records from or to the caches can only be moved in *cache lines*, i.e., up to B at the time. The cost of an algorithm is the number of *parallel* cache line transfers to or from up to P caches. Unlike the original PEM however, in the atomic PEM, we are strict about records being indivisible *atoms*, i.e., algorithms cannot gain any more information about them except identifying them. Algorithms have complete knowledge of the location of all records at all times, which implies that they can

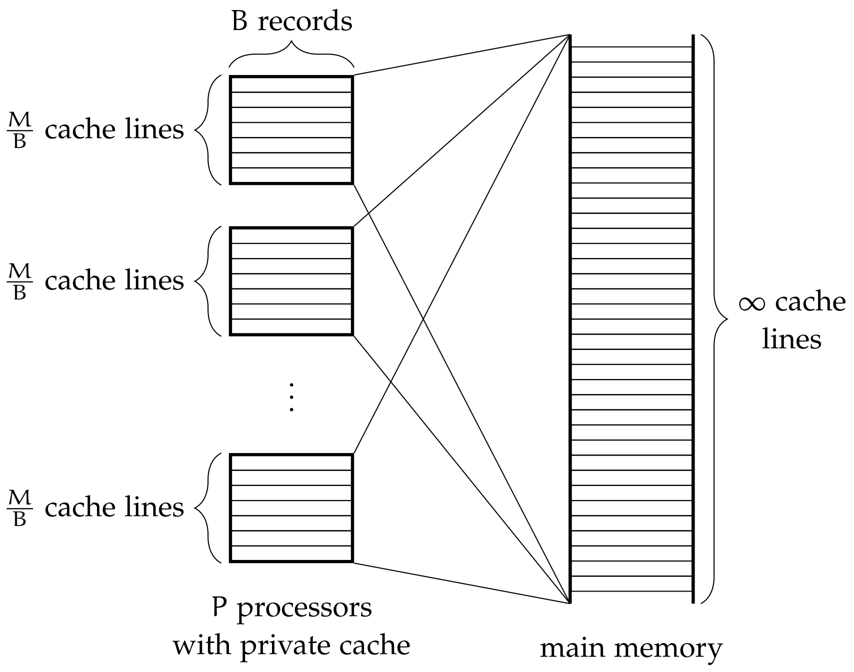

B records
$\frac{M}{B}$ cache lines
$\frac{M}{B}$ cache lines
$\frac{M}{B}$ cache lines
∞ cache lines
P processors with private cache
main memory

Figure 7: The parallel external memory model [12]

determine the rank (of any order) of each record. In problems on multisets, we distinguish *copies* of records from records *with the same key* (which is known to the algorithms).

When appropriate for the problem, algorithms can also do the commutative semigroup operation $\oplus$, which allows the creation of new records but does not reveal any other insight on the records (like Jacob et al. [88] do in their *semigroup PEM* model). Semigroup operations can only by applied to two records in the same cache. We assume that semigroup operations are "deterministic" in the sense that if two records and a copy of these records are added with the operation of the semigroup, the resulting records are copies of each other as well. In other words, copies are equivalent to each other and the semigroup operation is defined on equivalence classes of records. Or formally: if a' is a copy of a and b' a copy of b, then $a \oplus b$, $a \oplus b'$, $a' \oplus b$, and $a' \oplus b'$ are all considered copies of each other. In all our problems, we only allow semigroup operations on records with the same key and assume that the result has again that same key.

4.3.1.2 *Problems*

All problems in this section are defined on N records that initially reside in contiguous locations in main memory. The records have one of K different keys where key at rank i is used by N_i records, hence $\sum_{i=1}^{K} N_i = N$. The algorithms are asked to produce their result again in contiguous memory locations.

The problems are defined as follows:

BUNDLESORTING: The algorithm is asked to reorder the records by the rank of their keys. The relative order of elements with the same key can be arbitrary.

SORTING: The special case of BUNDLESORTING where $K = N$.

DUPLICATEREMOVAL: The algorithm is asked to remove all but one record for each of the K keys. To restrict the power of the algorithm, we only allow it to drop a record when another record of the same key is in the same cache.

MULTISETSORTING: The same as DUPLICATEREMOVAL, except that the remaining records are ordered by the rank of their keys.

GROUPING: The algorithm is asked to reorder the input such that all records with the same key reside in contiguous memory locations (i.e., in "groups").

AGGREGATION: The algorithm is asked to compute $\bigoplus\{e|e \text{ has rank } i\}$ for $i = 1, \dots, K$ from the input records and store them in main memory.

ORDEREDAGGREGATION: The same as AGGREGATION, but the algorithm is asked to order the result in an arbitrary, but a priori defined order.

Note that with our definitions, DUPLICATEREMOVAL and AGGREGATION, as well as MULTISETSORTING and ORDEREDAGGREGATION are respectively equivalent in the sense that the algorithm needs to bring records with the same key into the same cache in order to reduce the number of records with that key. Furthermore, we could also define ORDEREDGROUPING as a special case of GROUPING where we specify an order, which would be exactly equivalent to BUNDLESORTING.

Also note that MULTISETSORTING is sometimes defined differently. In our definition, duplicate records can be dropped, while other authors [63] have defined it like our definition of BUNDLESORTING. With our choice, we can reason about the impact of the ability to drop records more easily. Furthermore, MULTISETSORTING and DUPLICATEREMOVAL is sometimes defined such that duplicate records are to be put into a dedicated memory zone instead of just dropping them. This however does not make any asymptotic difference for neither upper bounds nor lower bounds.[3]

3 For the upper bound, a single cache line is enough to "collect" duplicates and flush them efficiently with only constant overhead. For the lower bound, the number of *equivalent* input configurations does not increase, since any order of the duplicates is a correct result (see discussion of Equation 11).

4.3.2 *Lower Bounds*

In order to prove lower bounds, we use a counting argument similar to what other authors have used before [3, 127, 79]: We bound the number of different input configurations that can lead to a certain output configuration with a given number of cache line transfers. This number must be at least as large as the number of all different input configurations that exist.

We first do the main part of the analysis common to all problems, and then derive concrete results for each of them specifically.

4.3.2.1 *Main Part*

The analysis is an adaptation of the analysis of Greiner [79] (in turn inspired by Bender et al. [26]) who calls the method "time backward analysis". It consists in counting the number of possible input configurations an algorithm can derive a particular output configuration from. This is done by looking at how an algorithm works backwards and examining carefully how many different configurations can *precede* a given configuration by each cache line transfer. In this approach semigroup operations are slightly easier to model than in a time forward analysis.

Before starting the analysis, we need to make some initial observations and assumptions. First, we argue that we can forbid copies of records without loosing generality: None of our problems requires copies of records in the output. Hence, we can transform the execution of any algorithm into an execution with the same result but without copies by working ourselves backwards through the execution of the algorithm and removing all records that do not participate in the output. This transformation does not increase the number of cache line transfers. In the same way, since every record either has a unique key or participates in exactly one semigroup operation in our problems, we can assume w.l.o.g. that the inputs of a semigroup operation are removed after the operation. Note that the way we define the dropping of elements behaves in all respects like semigroup operations, so we only speak of the latter in the remainder of the proof. With a similar argument, we assume that all possible semigroup operations are done immediately when two records with the same key are in the same cache together.

Second, only consider non-empty cache lines in main memory because two algorithm executions differing only by some empty cache lines can be transformed into each other by adding or removing empty cache lines. This does neither alter the number of cache line transfers nor the output configuration, which is required to reside contiguously in main memory.

Third, we also ignore the permutation of records within each cache line for the moment. The algorithm can permute records in the output in the desired way the last time each cache line is written, which is free. Permutations within cache lines of the input are taken into account when we consider the different input configurations below.

These observations allow us to bound the number of input configurations that any algorithm can transform to a given output configuration using ℓ (parallel) cache line transfers. With $\ell = 0$ there is exactly 1 configuration. By bounding the number of intermediate configurations preceding a given configuration, we bound the factor by which their total number increases with every cache line transfer.

First, we look at a single processor in isolation. We suppose that to go from one configuration to another, i.e., to go from ℓ to $\ell + 1$, the processor reads a cache line from main memory, does some semigroup operations and permutations in the cache, and writes a cache line back to main memory.[4] For the input, the processor might have read from at most $2N + 1$ different cache lines to produce the current configuration: since there are only N records in total and no copies exists, there are at most N non-empty cache lines and $N + 1$ cache lines around them, which may have been emptied by this input. This increases the number of preceding configurations by a factor $2N + 1$. Other processors might have read from the same cache line, but since we assume that there are no copies, every record only ended up in at most one processor cache. From the perspective of one processor, each of the B records of the cache line might or might not have ended up in its cache, so the input operation increases the number of preceding configurations by another factor 2^B. In total reading a cache line from main memory increases the number of configurations by factor $2^B(2N + 1)$.

After reading the cache line, its content is placed into the processor cache. Since all semigroup operations are done as soon as possible, i.e., directly after reading them from main memory, the only possible semigroup operations are those involving the B new records from this cache line. Since $\oplus$ is commutative, no other operations are possible afterwards. So if the cache of processor p contains $M_{p,l}$ records after l cache line transfers, only B of the $M_{p,l}$ records have changed with the current cache line transfer: each of them is either an input record that has just been read or the result of a semigroup operation. Hence, there are at most $\binom{M_{p,l}}{B}$ configurations preceding a given configuration through placing an input cache line into cache and doing all semigroup operations.

For the output, only one of the full N cache lines can have been changed by the last cache line transfer.

4 This is equivalent up to a constant factor to counting the reading and writing separately, but easier for the purpose of our analysis.

In total, the number of configurations preceding a given configuration by the cache line transfer of one processor is

$$2^B(2N+1)\cdot\binom{M_{p,l}}{B}\cdot N. \tag{9}$$

By taking into account the effect of all P processors, ℓ cache line transfers can turn at most the following number of different input configurations into a given output configuration:

$$\left(\prod_{p=1}^{P}\left((2N^2+N)\binom{M_{p,l}}{B}2^B\right)\right)^{\ell}. \tag{10}$$

We now quantify the number of possible input configurations. If there are N different records in the input, they can be permuted in N! different ways. Since we abstract from the permutations inside the $\frac{N}{B}$ output cache lines however, there are only $N!/B!^{\frac{N}{B}}$ different configurations of the input cache lines as discussed above. Furthermore, in problems on multisets and those where the order of the records in the output can be arbitrary, some input configurations are equivalent.[5] Let $\mathcal{P}$ be the number of equivalent configurations of a given problem, then there are $N!/B!^{\frac{N}{B}}/\mathcal{P}$ possible input configurations.

The number of cache line transfers ℓ needed to turn any possible input configuration into a valid output configuration is therefore bounded by the following inequality:

$$\left(\prod_{p=1}^{P}\left((2N^2+N)\binom{M_{p,l}}{B}2^B\right)\right)^{\ell} \geqslant \frac{N!}{B!^{\frac{N}{B}}\cdot\mathcal{P}}. \tag{11}$$

We can simplify the binomial coefficient by seeing that drawing all P cache lines of B records from the union of all caches at the same time does not reduce the number of preceding configurations. Furthermore, there cannot be more records in the caches than there are records in the input (N):

$$\prod_{p=1}^{P}\binom{M_{p,l}}{B} \leqslant \binom{\sum_{p=1}^{P}(M_{p,l})}{PB} \leqslant \binom{\min(MP,N)}{PB} \tag{12}$$

With this reflection and by making the first term slightly larger, Equation 11 becomes

$$\left(3N^2 2^B\right)^{P\ell}\binom{\min(MP,N)}{PB}^{\ell} \geqslant \frac{N!}{B!^{\frac{N}{B}}\cdot\mathcal{P}}. \tag{13}$$

5 The number of equivalent configurations would also be decreased if the semigroup operation had inverses: two records with the same key that are inverses of each other, do not need to contribute to the result and can thus be completely ignored. All permutations of the records that can be ignored would thus be equivalent.

By taking the logarithm on both sides and making the algebraic transformations detailed in Appendix A.2.1, we obtain

$$\ell \geqslant \frac{N \ln \frac{N}{eB} - \ln \mathcal{P}}{\Theta(P(\ln N + B \ln d))}, \tag{14}$$

with $d = \max\left(2, \min\left(\frac{M}{B}, \frac{N}{PB}\right)\right)$. We observe that $\ln N > B \ln d$, iff $N > d^B$, which is true, given that $N > 2^B$, iff $N > \left(\frac{M}{B}\right)^B$ or $P > \frac{N^{1-\frac{1}{B}}}{B}$. This means that the denominator of Equation 14 simplifies to $\Theta(PB \log d)$ except for ridiculously large N or P compared to M and B, and to $\Theta(\log N)$ otherwise.

4.3.2.2 Sorting

We now instantiate and simplify Equation 14 for different problems, starting with the known result for Sorting of sets. In this case all records have distinct keys, so all input configurations are different, i.e., $\mathcal{P} = 1$. We can distinguish two cases, depending on which term dominates the denominator. If $\ln N \geqslant B \ln d$, then $\ln N \geqslant B$, so $\log \frac{N}{B} = \Omega(\log N)$. This observation and the opposite case give the following, well known [12, 79] bound:

$$\ell = \Omega\left(\min\left(\frac{N}{P}, \frac{N}{PB} \log_d \frac{N}{B}\right)\right). \tag{15}$$

4.3.2.3 *Problems on Multisets with Specified Order*

MultisetSorting, BundleSorting (or OrderedGrouping), and OrderedAggregation are all equivalent in the number of input different configurations. In all three problems, only the key of the records matters for the input configuration: Two input configurations where two records with the same key are exchanged lead to the same output by *exactly the same sequence of cache line transfers*. For each key at rank i, we thus have $N_i!$ equivalent input configurations, so $\mathcal{P} = \prod_{i=1}^{K}(N_i!)$.

The problems only vary in how duplicates are treated. BundleSorting/OrderedGrouping keep all of them; OrderedAggregation combines them pairwise using the semigroup operation; and MultisetSorting drops one of two duplicates when they are in the same cache together. It is interesting to observe that whether or not the number of records can be reduced (by dropping or aggregating) does not change anything in the proof. However, it is important that an algorithm for MultisetSorting is forced to have the duplicates in cache when it drops one of them, as in our permissive model it could otherwise just ignore duplicate records from the beginning.

Hence, with $\mathcal{P}$ thus defined, we have

$$\begin{aligned}\ln \mathcal{P} &= \ln \prod_{i=1}^{K} (N_i!) = \sum_{i=1}^{K} \ln(N_i!) \\ &\leqslant \sum_{i=1}^{K} \left((N_i + \frac{1}{2}) \ln N_i - N_i + 1 \right) \leqslant \sum_{i=1}^{K} N_i \ln N_i,\end{aligned}$$

so Equation 14 becomes

$$\begin{aligned}\ell =& \Omega\left(\frac{N \ln \frac{N}{eB} - \sum_{i=1}^{K} N_i \ln N_i}{P(\ln N + B \ln d)} \right) \\ =& \Omega\left(\frac{1}{P} \cdot \min \left\{ \begin{array}{l} N \log_N \frac{N}{eB} - \sum_{i=1}^{K} N_i \log_N N_i \\ \frac{N}{B} \log_d \frac{N}{eB} - \sum_{i=1}^{K} \frac{N_i}{B} \log_d N_i \end{array} \right\} \right). \qquad (16)\end{aligned}$$

We can simplify Equation 16 by taking the multiplicities N_i that maximize it, which happens for $N_i = \frac{N}{K}$. Similar to the transformations in Appendix A.1, Equation 16 then becomes

$$\ell = \Omega\left(\min\left(\frac{N}{P} \log_N \frac{K}{B}, \frac{N}{PB} \log_d \frac{K}{B} \right) \right). \qquad (17)$$

Equation 17 is negative for $K < B$, which is not particularly useful as a lower bound. However, if we assume that the algorithm has to read the input,[6] we get an additional lower bound of $\frac{N}{PB}$. The lower bound on cache line transfers for MultisetSorting, BundleSorting, OrderedGrouping, and OrderedAggregation is thus

$$\Omega\left(\frac{N}{PB} \max\left(1, \min\left(B \log_N \frac{K}{B}, \log_d \frac{K}{B} \right) \right) \right). \qquad (18)$$

Note that Matias et al. [127] find the same bound, except that they handle the case of negative values of the bound differently (namely by falling back to an argument with comparisons). Furthermore, they erroneously omit the first term in the minimum; however this has no real consequences in practice: The first term of the bound only matters for ridiculously large N or P as discussed above.

4.3.2.4 *Problems on Multisets without Specified Order*

DuplicateRemoval, Grouping, and Aggregation are all oblivious to the order of records within each group like with the previous problems of the previous section. Furthermore, the algorithm is free to choose any of the K! orders of the keys in the output. This is the same as considering two input configurations equivalent if they only differ by a renaming of the keys. Hence, $\mathcal{P} = \prod_{i=1}^{K} (N_i!) \cdot K!$.

6 This technically means that we leave the atomic PEM model, where the algorithms knows where every element resides without even reading it. It thus might not have to read the entire input if part of it is part of a correct result. However, for a practical algorithm, this assumption is obviously realistic.

Since $\ln(K!) \leqslant K \ln K$, Equation 14 becomes

$$\ell = \Omega\left(\frac{N \ln \frac{N}{eB} - \sum_{i=1}^{K} N_i \ln N_i - K \ln K}{P(\ln N + B \ln d)}\right). \tag{19}$$

This is maximized by the multiplicities $N_i = \frac{N}{K}$ as above and thus leads to the simplified bound

$$\ell = \Omega\left(\frac{(N-K)\ln \frac{K}{B} + K \ln B}{P(\ln N + B \ln d)}\right). \tag{20}$$

For the case of small K, i.e., $N - K = \Omega(N)$, and under the assumption that $B = d^{\mathcal{O}(1)}$, we cannot gain anything by ignoring the order of the groups: With these assumptions, $\log_N B = \log_d B = \mathcal{O}(1)$, so the last term in the numerator is dominated by the first and we get the same bound as Equation 18:

$$\Omega\left(\frac{N}{PB} \max\left(1, \min\left(B \log_N K, \log_d \frac{K}{B}\right)\right)\right). \tag{21}$$

4.3.3 *Upper Bounds*

We now show that the bounds that we derive in the previous section are tight for many parameter ranges, in particular those relevant in practice.

4.3.3.1 *Sequential Algorithms for Problems with Specified Order*

In the single processor case, we can use the modified MergeSort from Section 4.2.4 again for MultisetSorting and OrderedAggregation. For MultisetSorting, one of two records that compare equal is simply dropped; for OrderedAggregation they are aggregated using the semigroup operation. This algorithm achieves the non-degenerated case of the lower bound in Equation 16, i.e., that bound is tight for $P = 1$ and either $N > (M/B)^B$ or $P > N^{1-\frac{1}{B}}/B$.

For BundleSorting, we cannot use the same algorithm since its analysis relies on the fact that the algorithm drops duplicate records. However, as mentioned earlier, FunnelSort from Farzan et al. [63] has the same asymptotic complexity, so in the non-degenerated case, our bound is tight for BundleSorting as well.

4.3.3.2 *Parallel* BundleSorting

We adapt again an existing Sort algorithm. The PemMergeSort presented by Arge et al. [12] uses a sequential external memory sort algorithm as the base case, which we replace by the BundleSort of Matias et al. [127]. This allows us to build a parallel BundleSort. In the worst case, a record of each of the K groups is present in the input of each processor, so each base case of size $\frac{N}{P}$ may need up to

$\frac{N}{PB} \max\left(1, \log_d \frac{K}{B}\right)$ I/Os. Hence, the algorithm requires at most the following number of cache line transfers in the worst case (see Corollary 3 of [12]):

$$\mathcal{O}\left(\frac{N}{PB} \max\left(1, \log_{\frac{M}{B}} \frac{K}{B} + \log_d P\right)\right). \tag{22}$$

If we assume that the number of processors is not too large, namely $P = (M/B)^{\mathcal{O}(1)}$ and $P < \frac{N}{M}$, then $\log_d P = \log_{\frac{M}{B}} P = \mathcal{O}(1)$, so a constant number of merge passes across processors is enough and the complexity of the algorithm is dominated by the base case. This assumption restricts us to an easy case in the PEM model; however it is not a restriction for current multi-core processors. Hence, in the non-degenerated cases (i.e., if $N < (M/B)^B$ and $P < N^{1-\frac{1}{B}}/B$), the algorithm matches the lower bound from Equation 18:

$$\mathcal{O}\left(\frac{N}{PB} \max\left(1, \log_{\frac{M}{B}} \frac{K}{B}\right)\right). \tag{23}$$

Note that building a parallel algorithm that achieves the precise bound in terms of multiplicities is not as trivial. If we just solved the base case with an optimal algorithm we would not achieve our goal: since the processors solve their base cases independently, it is sufficient that a single processor gets a hard distribution of multiplicities in order to slow the whole algorithm down. Instead, we present an algorithm for MultisetSorting and Aggregation with more sophisticated work balancing in the next section. Achieving the same for BundleSorting is an open problem.

4.3.3.3 *Parallel* MultisetSorting *and* OrderedAggregation

In this section we present a refined version of the algorithm above that balances the work of MergeSort in the base case of the algorithm. Since the work can become unbalanced after every merge level, we need to balance once per level.

The pseudo-code notation that we use in this section describes programs that are executed simultaneously on all processors; the current processor is called p (with $p \in \{1, \ldots, P\}$). Processor-local variables start with a lower case letter, while global variables start with an upper case letter. The shortcut A_p to a global array A gives access: **Array** $[1..n]$ gives access to p's fraction of A split into P equal parts, i.e., to $A[\frac{(p-1)n}{P} + 1..\frac{pn}{P}]$.

Algorithm 2 shows the refined algorithm. Like the original algorithm, we start with making runs of size M in a single pass, merge them iteratively until there is only $\mathcal{O}(1)$ run left per processor, and then use the cross-processor Merge algorithm of Arge et al. [12] to produce the final result. With the appropriate inner Merge routine, we can make the algorithm solve MultisetSorting or OrderedAggregation as desired.

Algorithm 2 PEM algorithm for MULTISETSORTING and ORDEREDAGGREGATION

```
1: func PEMMULTISETSORT(S: Array [1..N] of Record)
2:     runs ← MAKERUNS($S_p$)
3:     while PEMREDUCE($\|runs\|$, +) > P do
4:         runs ← PEMBALANCEDMERGE(runs)
5:     return PEMSAMPLEMERGE(runs)
```

The main mechanism for balancing the work is encapsulated in the routine PEMBALANCEDMERGE, which is detailed by the pseudo-code in Algorithm 3. In the pseudo-code, $\|S\|$ refers to a metric defined as the total number of records in a set of runs, i.e., $\|S\| = \sum_{s \in S} |s|$. The processors start by computing the total number of runs R still left in this merge level and the number of records N' in these runs. This means that every processors should get $\mathcal{O}(\frac{R}{P})$ runs with $\mathcal{O}(\frac{N'}{P})$ records. Each processor then filters out those runs that alone contain more records than $\frac{N'}{P}$ records. Then each processor filters out small runs, i.e., those runs that contain less than $\frac{N'}{2R}$ records, which are then redistributed equally among the processors. The remaining runs are put into bins of up to $\frac{2N'}{P}$ records using GREEDYBINPACKING, a 2-approximation of BINPACKING with linear runtime. Finally, each processor merges the runs of one bin and its small runs using a sequential MERGE algorithm.

Algorithm 3 Building blocks of PEMMULTISETSORT

```
1: func PEMCONCAT(S: Array of T)
2:     l ← Len[p] ← $|S_p|$                                  ▷ $\mathcal{O}(\log B)$
3:     Len ← PEMPREFIXSUM(Len)                              ▷ $\mathcal{O}(\log P)$
4:     return R with R[Len[p]..Len[p] + l] ← $S_p$           ▷ $\mathcal{O}(\frac{\max S_i}{B})$
5: func PEMBALANCEDMERGE(runs: Array [1..$r_p$] of Array of Record)
6:     $\langle N', R\rangle$ ← PEMREDUCE($\langle \|runs\|, r_p\rangle$, +)            ▷ $\mathcal{O}(\log P)$
7:     ignored_runs ← $\left\{run \in runs \middle| |run| > \frac{2N'}{P}\right\}$        ▷ $\mathcal{O}(\frac{\max r_i}{B})$
8:     small_runs ← $\left\{run \in runs \middle| |run| \leqslant \frac{N'}{2R}\right\}$      ▷ $\mathcal{O}(\frac{\max r_i}{B})$
9:     runs ← runs \ (small_runs ∪ ignored_runs)              ▷ $\mathcal{O}(\frac{\max r_i}{B})$
10:    Small_Runs ← PEMCONCAT(small_runs)           ▷ $\mathcal{O}(\frac{\max r_i}{B} + \log P)$
11:    bins ← GREEDYBINPACKING(runs, $\frac{2N'}{P}$)              ▷ $\mathcal{O}(\frac{\max r_i}{B})$
12:    Bins ← PEMCONCAT(bins)                       ▷ $\mathcal{O}(\frac{\max r_i}{B} + \log P)$
13:    return MERGE(Bins[p] ∪ $Small\_Runs_p$) ∪ ignored_runs
```

In order to analyse the complexity of PEMMULTISETSORT, we first analyse PEMBALANCEDMERGE for the case where there are still at least $R > P\frac{M}{B}$ runs. As we show, PEMBALANCEDMERGE merges them into $\mathcal{O}(R\frac{B}{M})$ runs with at most $\mathcal{O}(\frac{N'}{PB})$ parallel I/Os.

Initially, we assume that there are no large runs with more than $\frac{N'}{P}$ records. Later we show that ignoring large runs does not affect correctness of the algorithm or increase the number of I/Os.

We show that the number of runs and the number of records in these runs is asymptotically balanced among the threads. We start with the small runs, i.e., those that have less than $\frac{N'}{2R}$ records. Since there are only R runs in total, there are also at most R small runs, so each processor gets at most $\lceil \frac{R}{P} \rceil$ of them when they are redistributed. Each small run has at most $\left\lceil \frac{N'}{2R} \right\rceil$ records, so each processor gets at most $\lceil \frac{R}{P} \rceil \cdot \left\lceil \frac{N'}{2R} \right\rceil \leqslant \frac{R}{P} \cdot \frac{N'}{2R} + \frac{R}{P} + \frac{N'}{2R} + 1 \leqslant \frac{N'}{2P} + \frac{N'}{P} + \frac{N'}{2P} + 1 \leqslant \frac{2N'}{P} + 1$ records (since $R > P$ and $R \leqslant N'$). When the remaining runs of each processor are packed into bins, each bin has at most $\frac{2N'}{P}$ records by definition of the bins. Since we assume that no run has more than $\frac{2N'}{P}$ records, every run can be put into a bin. The records in the bins are in runs of at least $\frac{N'}{2R}$ runs, so there are at most $\frac{2N'}{P} \left(\frac{N'}{2R}\right)^{-1} = \frac{4R}{P}$ runs per bin. Finally, since GreedyBinPacking is a 2-approximation, there are at most P bins, so all bins can be processed in parallel by the final call to Merge. In total, there are at most $4\frac{N'}{P} + 1$ records in at most $\frac{5}{P} + 1$ runs per processor. Since Merge needs $\mathcal{O}(\frac{n}{B} + r)$ I/Os to merge r runs with a total of n records on a single processor, the last line needs at most $\mathcal{O}(\frac{N'}{PB} + \frac{R}{P})$ parallel I/Os. The values of $\frac{R}{P}$ of all merge levels form a geometric series, whose sum is smaller than the reading costs of $\frac{N}{PB}$, so the last line has the desired bound.

All other lines only manage runs previously produced on the current processor. All constant costs per run can thus be attributed to the previous level of recursion making only a constant difference and consequently ignored in the complexity analysis. The remaining lines hence only contribute $O(P)$ I/Os.

Now we show that PemBalancedMerge indeed reduces the number of runs by factor $\mathcal{O}(\frac{M}{B})$. Let r_i be the number of runs assigned to processor i for $1 \leqslant i \leqslant P$. Then $R = \sum_{i=1}^{P} r_i$ and each processor creates $\lceil r_i/(M/B) \rceil$ runs, so the total number of runs produced by the algorithm is:

$$\sum_{i=1}^{P} \left\lceil \frac{r_i}{\frac{M}{B}} \right\rceil \leqslant \sum_{i=1}^{P} \left(\frac{r_i}{\frac{M}{B}} + 1 \right) = \frac{R}{\frac{M}{B}} + P = \mathcal{O}(R\frac{B}{M}). \tag{24}$$

The last equality comes from the assumption that $R > P\frac{M}{B}$, which implies that $P < R\frac{B}{M}$.

With a similar argument, we can show that the algorithm produces a correct output even if we ignore large runs: Runs are ignored if they have more than $\frac{N'}{P}$ records, so there are at most P ignored runs in the input of N' records. Adding these P runs to the final set of runs does not increase their number asymptotically.

We have now everything at hand to determine the complexity of PemMultisetSort: The initial run production takes $\mathcal{O}(\frac{N}{PB})$ I/Os. Asymptotically, the last line of the algorithm excluding the last iteration, i.e., the last line of those iterations where $R > P\frac{M}{B}$, does the same number of I/Os as the sequential DUPLICATEREMOVAL algorithm of Arge [13] and has an optimal speed-up as shown above. It thus requires the following number of I/Os:

$$\mathcal{O}\left(\max\left(\frac{N}{PB}, \frac{N}{PB}\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=0}^{K}\frac{N_i}{PB}\log_{\frac{M}{B}} N_i\right)\right). \tag{25}$$

There is only a single merge level where $R < P\frac{M}{B}$ before the loop is ended because $\mathcal{O}(P)$ runs are left, which contributes at most $\mathcal{O}(\frac{N}{PB})$ I/Os for the last line. The other lines of the main loop contribute $\mathcal{O}(\log P \log_{\frac{M}{B}} K)$. Finally, there are at most $\log_d P$ cross-processor passes of PEMSAMPLEMERGE. This makes up the following total number of I/Os:

$$\begin{aligned}\mathcal{O}\Bigg(&\max\left(\frac{N}{PB}, \frac{N}{PB}\log_{\frac{M}{B}}\frac{N}{B} - \sum_{i=0}^{K}\frac{N_i}{PB}\log_{\frac{M}{B}} N_i\right)\\ &+\frac{N}{PB}\log_d P + \log P \log_{\frac{M}{B}} K\Bigg).\end{aligned} \tag{26}$$

If we assume again that the number of processors is small enough such that a constant number of PEMSAMPLEMERGE passes is sufficient, i.e., if we assume that $P = (M/B)^{\mathcal{O}(1)}$ and $P < \frac{N}{M}$, then we can drop the $\log_d P$ term. With this assumption, it also holds that $\log P \log_{\frac{M}{B}} K = \mathcal{O}\left(\log\frac{M}{B}\log_{\frac{M}{B}} K\right)$, which corresponds to $\mathcal{O}\left(\log\frac{M}{B}\right)$ I/Os per merge level. We can also drop this term with the following observation: in all but a constant number of merge levels, we still have $R \geqslant P\frac{M}{B}$ runs, so each processor has at least $\mathcal{O}\left(\frac{M}{B}\right)$ of them. Reading these runs dominates $\log\frac{M}{B}$, the term we want to drop.

We thus achieve the bound in Equation 16 for the non-degenerated cases (i.e., if $N < (M/B)^B$ and $P < N^{1-\frac{1}{B}}/B$), making the bounds for MULTISETSORTING and ORDEREDAGGREGATION tight for these cases.

4.3.3.4 *Parallel Problems on Multisets with Unspecified Order*

The lower bound for DUPLICATEREMOVAL, AGGREGATION, and GROUPING is the same as the one for the problems just discussed in the case where K is small enough such that $N - K = \Omega(N)$. In these cases we can use the same algorithm.

In the other case, i.e., if $N - K = o(N)$, then $K < \frac{N}{2}$, so there are $\Theta(K)$ unique and $\Theta(N - K)$ non-unique records. Assuming that we know which records are unique (for example using a pre-processing algorithm that breaks the indivisibility assumption), we can first scan the input in order to separate the former from the latter and then use the optimal MergeSort-based algorithm from above on the $N - K$ non-unique records. With the same assumptions as above, this requires the following number of cache line transfers:

$$\mathcal{O}\left(\frac{N}{PB} + \frac{N-K}{PB}\max\left(1, \log_{\frac{M}{B}}\frac{N-K}{B}\right)\right). \quad (27)$$

In the case where $N - K \leqslant \frac{N}{\log_{\frac{M}{B}} N}$, it holds that

$$\frac{N}{N-K} \geqslant \log_{\frac{M}{B}} N \geqslant \log_{\frac{M}{B}} N - \log_{\frac{M}{B}}\frac{BN}{N-K} = \log_{\frac{M}{B}}\frac{N-K}{B}, \quad (28)$$

so the sorting of the non-unique elements is dominated by the prior reading pass. The bound of Equation 27 is therefore equivalent to the following, thus making the lower bound tight:

$$\mathcal{O}\left(\frac{N}{PB}\right). \quad (29)$$

Narrowing the gap between upper and lower bound in the other cases, i.e., for K such that $\frac{N}{\log_{\frac{M}{B}} N} \leqslant N - K = o(N)$, is an open problem.

4.3.3.5 *Direct Shuffling*

For above upper bounds, we assume that $N < (M/B)^B$ and $P < N^{1-\frac{1}{B}}/B$, which are both realistic assumptions for real machines. For the contrary case, Aggarwal and Vitter [3] had proposed DirectShuffling in the sequential EM model: each record is moved directly to its rank. This is not directly applicable in the PEM model with CREW policy because possible write conflicts need to be resolved. The worst conflict arises if all accesses go to the same location, i.e., $K = 1$, which can be resolved by local pre-processing in $\mathcal{O}(\frac{N}{P})$ and a reduction in $\mathcal{O}(\log P)$ accesses. So under the realistic assumption that $P \log P = \mathcal{O}(N)$, the conflict resolution is dominated by the overall reading costs and can be ignored.[7] However, DirectShuffling is not sufficient to match our lower bounds, except for K large enough such that $K = N^{\Omega(1)}$, in which case the algorithm achieves the desired bound:

$$\mathcal{O}\left(\frac{N}{P}\max\left(1, \log_N K\right)\right) = \mathcal{O}\left(\frac{N}{P}\right). \quad (30)$$

Closing the gap between upper and lower bound for degenerated machines with either smaller K or unspecified output order remains an open, though practically probably irrelevant problem.

7 It is not possible to gather P records into the same cache line with less than $\log P$ cache line transfers [12], so what follows is actually optimal even without our assumption about P.

4.3.4 *Discussion*

The analysis of this section gives us a series of insights. Most importantly it confirms the folklore conjecture that AGGREGATION is as hard as MULTISETSORTING under many conditions. Unless N or P are unrealistically large, we can say the following: ORDEREDAGGREGATION and MULTISETSORTING have the same complexity, so if we build an AGGREGATION algorithm such that the order of the output does not fundamentally depend on the order of the input, SORTING is optimal. This is true for the precise bound in terms of multiplicities. In terms of K, BUNDLESORTING has also the same bound. If K is small enough such that $N - K = \mathcal{O}(N)$, then the order of the output does not matter and MULTISETSORTING is optimal again.

The analyses also show some cases where AGGREGATION is easier than MULTISETSORTING. If N is large compared to B and M, DIRECTSHUFFLING is faster than SORTING. This is below the lower bound of the external memory Turing machine model, which means that that model misses some fundamental features that the PEM model has. For AGGREGATION we identified one: in order to do DIRECTSHUFFLING, we need to take the record or a hash value of it as "address", which is not possible by doing only comparisons. Furthermore, if K is very large and we assume that we can identify duplicates somehow, a constant number of passes over the input is enough, which is better than SORTING (at least for reasonably small P). Finally, we can also observe that GROUPING and DUPLICATEREMOVAL as well as MULTISETSORTING and BUNDLESORTING have the same tight bounds in terms of K. This means that, somewhat counter-intuitively, dropping or aggregating records, does not make problems asymptotically easier in the general case.

We also identified a series of open problems. Most importantly our upper bounds in the PEM model can still be improved: In some cases we make quite strong assumptions, for example about P, and still do not get exactly matching bounds for all parameter ranges or with the desired precision. For the purpose of this work, in particular for practical algorithms on current, real-world hardware, our bounds are already very helpful, but from a theoretical perspective, we think that it is interesting to have more complete answers. Concretely, the following questions are still open:

- Is BUNDLESORTING harder than MULTISETSORTING in terms of multiplicities (in the EM or the PEM model)? We show that they are the same in terms of K, but do not know of an algorithm for BUNDLESORTING that matches the bound of MULTISETSORTING with the more precise analysis. In other words, are there input distributions where the fact that we can drop records make the problem asymptotically easier?

- Can we match the lower bounds for larger P? Here a better analysis of existing algorithms may already be enough, but maybe it is rather the lower bound that needs to be improved using a different proof technique.
- Can we close the gap (further) for order-independent problems—both for the sort-based algorithms and direct shuffling? Again, maybe a better analysis is already sufficient.

Furthermore, we only discuss worst-case complexities in our analyses, but the average case can also be interesting. Aggarwal and Vitter [3] and Greiner [79] obtain average case bounds for Sorting in the EM and PEM model respectively that match their worst-case counterparts.

Finally, it would be interesting to study Aggregation with non-commutative semigroup operations. Intuitively, this problem is still easier than Sorting a complete set since the relative order among the result records is arbitrary. However, within each group, the order of the input records determines which pairs of records need to be brought to cache together, so every group is an instance of the ProximateNeighbor problem, which is known to have Permutation bound [88]. This suggests that using non-commutative semigroup operations makes Aggregation harder. Assuming that we can save all K! permutations among the result records compared to an algorithm that can produce any Permutation of the input leads to the following conjecture:

Conjecture 7. Aggregation *with non-commutative semigroup operations need at least the following number of cache line transfers:*

$$\Omega\left(\frac{N \ln \frac{N}{eB} - K \ln K}{P(\ln N + B \ln d)}\right).$$

4.4 practical considerations

In this section we discuss practical aspects of the way we model Aggregation and the machine. This will help us to transfer the theoretical insights of this chapter to the real-world implementations in the subsequent chapters.

The external memory models that we use in this chapter were originally conceived for the analysis of algorithms on machines where internal and external memory correspond to main memory and disk respectively. The fact that CPU costs are ignored was undebatably realistic in this setup, considering the high costs of I/Os. Database query optimizers even used the number of I/Os as cost function for rather precise performance predictions.

When applied to CPU caches, the external memory model does not allow making as precise performance predictions. It is still widely adopted [70, 51, 12, 178, 11] as a formal method to understand and minimize memory access costs of algorithms and data structures. In this case the implicit assumption holds that CPU costs can be reduced or hidden in large parts thus making them close to "free", for example by making out-of-cache memory access sequential, eliminating branches, and using out-of-order execution or vectorization. As we show in the following chapters, this is not unrealistic for data intensive algorithms such as those used in database systems.

Furthermore, the models that we use keep many benefits, even if they differ from real-world hardware: For one the models are simple, so analyses in these models are intuitively understandable. Also, only the simplicity makes it possible to prove lower bounds. This is particularly interesting considering the fact that virtually all more sophisticated models are specializations of the permutation model (as they assume that data transfer from slow to fast memory is done in blocks), so the lower bounds proved in this model hold in the other models as well. These lower bounds show the hard kernel of problems, the amount of data that *any* algorithm *has* to transfer—even if the access pattern is optimal and CPU costs are negligible. It makes it possible to speak of an *optimal* amount of memory access and exclude non-optimal algorithms without implementing them.

In column-store database systems, cache efficiency is particularly relevant. Since data is stored and processed in narrow *columns* of typically 1 to 4 or 8 bytes,[8] we get relatively high values for parameter B even though cache lines are *much* smaller than disk pages (64 B instead of 4 KiB). The expectable speed-up through cache efficiency, B, is thus relatively high. At the same time, the same algorithm is executed repeatedly, once for the keys and then once for every aggregate column. In particular, the processing of the aggregate columns then only consists of moving data around. The benefit of cache efficiency thus pays off *once per column*.

The column-store layout also has another implication: it removes the necessity to load all data—the indivisibility assumption does not hold for *records*! If by merely looking at the key column, the algorithm can make some clever decisions, then this may make some problems simpler. We can thus easily build the algorithm for AGGREGATION on almost unique records, i.e., with $K - N = o(N)$, that we sketched in Section 4.3.3.4: Create a $\langle \text{key}, \text{group} \rangle$ vector with the key column using full sort complexity, annotating those records that are unique; then process the aggregate columns by removing the unique records first and then sorting the remaining ones. For very large K, where SORTING is the most expensive, we can thus get an effective one-pass algorithm.

8 Sometimes columns are even *less* than a byte wide, namely when they are compressed as in our work about bit packing [195].

Another question is whether we need a sorted output or not. In the permutation model that we use, integer sorting techniques are covered by the sorting bound, even though they do not use comparisons. Also, a standard hashing scheme, where the value of a hash function of the key is taken as address in a data structure, is often very similar to *sorting by hash value*, so our sorting bound also applies. We estimate that this is true for the vast majority of AGGREGATION algorithms in real-world database systems. In contrast, the algorithm we just sketched leaves the order on the unique elements as in the input and can thus beat the sorting lower bound. As mentioned before it is an interesting open question to design other more sophisticated algorithms whose output order fundamentally depend on the input order in order to make the bound of Equation 19 tight for all values of K.

Similarly, we might ask whether aggregate functions are actually semigroup operations on indivisible atoms. We argue that most real-world AGGREGATION algorithms treat the aggregate function as black box, where the function is given as parameter. In this case, at least the *attributes* are indivisible, as the function can only be invoked on two records that are in the cache, so it is indeed the responsibility of the algorithm to permute the records around in an efficient way. This is not without alternative though: Weidner et al. [185] and Pirk et al. [146] present approaches where AGGREGATION is only run on the most significant bits in order to reduce data transfer over network or to co-processing devices respectively. The result in full precision is then only computed for those records that are kept for the subsequent processing step.

Furthermore, modeling aggregate functions as semigroups restricts us to functions where intermediate results are of size $\mathcal{O}(1)$, which is true for *distributive* and *algebraic* aggregation functions as defined by Gray et al. [78]. This includes the most common ones like `COUNT`, `SUM`, `MIN`, `MAX`, and `AVG`, but not `MEDIAN` for example. These functions are also commutative and associative, but only on integer data types, not on types with floating point precision. In this case additional aspects of numerical stability may also have to be taken into account, which probably leads to more costly algorithms.

Finally, we want to highlight an observation that we already make in Section 4.1.1: on realistic machines, the term $\log_{\frac{M}{B}} \frac{K}{B}$ only takes values $\leqslant 4$. This means that asymptotic analyses are only of limited interest. While we can "hide" an additional pre- or postprocessing pass over the input "in the term of the reading costs", in practice this may mean intolerable extra costs. For example have we seen that GROUPING, where all N records from the input show up in the output, has the same complexity as AGGREGATION, where records with the same key are aggregated as soon as possible, at least in terms of K. However, in practice, *early aggregation* is a technique that every serious AGGREGATION algorithm uses to improve its constants.

4.5 RELATED WORK

In this section we briefly review prior work that is related to ours in various ways: algorithms and data structures in the external memory model, work on an extension to this model for hierarchical caches, and lower bounds for other relational operators.

4.5.1 *Work on External Memory Algorithms*

External memory models have been studied by many authors for a long time. We present the most relevant work in Section 4.1.2. In summary, Aggarwal and Vitter [3] introduced the most popular external memory machine model with upper and lower bounds for SORTING. Arge et al. [13, 14] developed two methods for deriving external memory lower bounds from comparison lower bounds and upper bounds for MULTISETSORTING and DUPLICATEREMOVAL. Matias et al. [127] proved upper and lower bounds for external BUNDLESORTING, which were improved by Farzan et al. FARZAN2005 to multiplicities. The permutation model was extended to multiple processors as the parallel external memory model by Arge et. al [12], which was again extended to incorporate semigroup operations by Jacob et al. [88]. For a more detailed discussion including sorting and related problems, linear algebra operations, data structures, problems on graphs, computational geometry, searching, and string processing, we refer to surveys on the topic [177, 176, 178].

Another type of problem bears some similarity to AGGREGATION and has been studied in depth in the context of external memory models: MATRIXMULTIPLICATION can be expressed as a JOIN followed by an AGGREGATION, and the sparsity of the matrices influences the runtime similar to how the number of groups influences the costs of AGGREGATION. First lower bounds date as far back as the work of Hong and Kung [90] about dense matrices. Later lower bounds were found for sparse matrix dense vector multiplication [26], sparse matrix dense matrix multiplication [81, 79], and sparse matrix sparse matrix multiplication [144]. Amossen and Pagh [7] even build an algorithm that can be used for both JOIN-PROJECT and MATRIXMULTIPLICATION.

Finally, the REDUCE step of MAPREDUCE [50] can also be seen as equivalent to AGGREGATION (though in practice with a more complex function than a semigroup operation). Greiner et al. [79, 80] study the complexity of MAPREDUCE in the context of external memory and find a tight lower bound with the technique we also use in our work. Since they make some assumptions about the data layout of the input (by modelling mappers such that each of them produces at most one record per key value), their result cannot be immediately transferred to AGGREGATION.

4.5.2 *Work on External Memory Data Structures*

SORTING and similar problems are related to some data structures. Probably, the most prominent example is Arge's buffer tree [10]. N batched insertions or queries to an initially empty buffer tree need $\mathcal{O}(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ I/Os and the inserted records can be read in sorted order with $\mathcal{O}(\frac{N}{B})$ I/Os. The buffer tree can hence be used to solve online SORTING optimally. This and the lower bound for SORTING also imply a lower bound of $\mathcal{O}(\log_{\frac{M}{B}}\frac{N}{B})$ amortized I/Os for each operation. Kumar and Schwabe [105] present an external memory heap with similar properties. To the best of our knowledge, it has not been studied yet whether a (possibly improved) buffer tree needs fewer I/Os if the records have $K < N$ distinct keys.

The buffer tree started a line of research about *dictionaries*, data structures that represent a set of $\langle \text{key}, \text{value} \rangle$ pairs and allow fast look-up with keys. Dictionaries could be used to solve AGGREGATION. Several authors studied the construction of dictionaries [157] and the trade-off between costs of insertions and online member queries—in external memory models making the indivisibility assumption [36, 35, 200], and in external memory models that do not [89, 184, 201, 174, 84]. Most of this work sees the dictionary rather as an index, so their results are only remotely related to our work.

A more detailed overview about data structures for external memory is provided by several surveys on that topic [9, 176, 178].

4.5.3 *Cache-Oblivious Model*

Another extension of the external memory model is the *cache-oblivious* model [70], where algorithms run on a machine with *unknown* cache size. Optimal algorithms in this model are also optimal in the cache-aware external memory model. What is more, they are particularly suited to run on a *hierarchy* of caches (such as multi-level caches of modern processors, see Section 2.3) because they exploit all of them optimally at the same time. At the same time, lower bounds of the cache-conscious model also hold in the cache-oblivious one.

The most relevant result for our work is the work on problems on multisets of Farzan et al. [62, 63]. They provide an upper bound for MULTISETSORTING, FUNNELSORT, which matches the cache-aware bounds of Arge et al. [13], and show an extension that solves DUPLICATEREMOVAL with the same bound. Modifying the DUPLICATEREMOVAL logic similarly to what we do in Section 4.3.3 would turn that algorithm into an optimal cache-oblivious AGGREGATION algorithm. For more work on cache-oblivious algorithms and data structures we refer again to surveys [11, 34].

4.5.4 *Work on (RAM) Lower Bounds of other Relational Operators*

There is a line of research about lower bounds of a particular kind of query in the RAM model: a series of JOIN operators in full conjunctive queries. Atserias et al. [15] showed that there is a tight bound on the result size of this type of queries. Subsequently, Ngo et al. [142], and later Veldhuizen [173] built algorithms with a runtime proportional to the result size. The lower bound was subsequently refined and matched with an upper bound by Ngo et al. [141]. However, we do not know of any external memory lower bound for other non-trivial relational operators than SORTING.

4.6 CONCLUSION

In this chapter we study AGGREGATION and related problems in two external memory models: We first extend the external memory Turing machine model to make aggregating records compatible with the indivisibility assumption. This allows us to derive a lower bound for AGGREGATION in terms of the multiplicities of the records, which is the same as MULTISETSORTING and can be achieved with an algorithm we provide. We then analyse a family of problems related to AGGREGATION in the more powerful parallel external memory (PEM) model. We are able to show that the same bound holds for many parameter ranges of this model as well and extends to machines with multiple processors. This proves the long-standing folklore conjecture of the database community that HASHAGGREGATION and SORTAGGREGATION have the same cache complexity in many interesting situations. At the same time, our bounds reveal configurations where the sorting bound does not apply. We still provide lower bounds and often matching upper bounds, at least for parameter ranges that are interesting in practice. As a side product of our study, we contribute upper and lower bounds of MULTISETSORTING, BUNDLESORTING, and DUPLICATEREMOVAL in the parallel external memory model and even improve some previous bounds in the sequential case. The insights we gain with this analysis shall serve as a design guideline for the subsequent practical chapters of this thesis and possibly for practical work of other authors in the database community.

Part III

PRACTICE

Actual operation or experiment, in contrast to theory.

— Wiktionary [192]

5 CACHE-EFFICIENT AGGREGATION: HASHING *IS* SORTING

For decades researchers have studied the duality of Hashing and Sorting for the implementation of relational operators, especially for efficient Aggregation. Depending on the underlying hardware and software architecture, the specifically implemented algorithms, and the data sets used in the experiments, different authors came to different conclusions about which is the better approach. In this chapter we argue that in terms of cache efficiency, the two paradigms are actually the same. Our claim is supported by the result of the previous chapter stating that the complexity of Hashing is the same as the complexity of Sorting in the external memory model. Furthermore, we make the similarity of the two approaches obvious by designing an algorithmic framework that allows switching seamlessly between Hashing and Sorting during execution. The fact that we mix Hashing and Sorting routines in the same algorithmic framework allows us to leverage the advantages of both approaches and makes their similarity obvious. On a more practical note, we also show how to achieve very low constant factors by tuning both the Hashing and the Sorting routines to modern hardware. Since we observe a complementary dependency of the constant factors of the two routines to the locality of the input, we exploit our framework to switch to the faster routine where appropriate. The result is a novel relational Aggregation algorithm that is cache-efficient—independently and without prior knowledge of input skew and output cardinality—, highly parallelizable on modern multi-core systems, and operating at a speed close to the memory bandwidth, thus outperforming the state of the art by up to 3.7×.

This chapter is based in large parts[1] on work that we published previously in [133]. The experimental results of that work were reproduced by a SIGMOD Review Committee and were found to support the central results reported in the paper.[2]

5.1 INTRODUCTION

As we have seen in Section 3.1, the dominant cost of Aggregation is, as with most relational operators, the movement of the data. Since the early days of disk-based database systems, relational operators have been designed to reduce the number of I/Os needed to access the

1 The main additions are Sections 5.6 and 5.7.
2 See http://db-reproducibility.seas.harvard.edu/, under "SIGMOD 2015".

disk whereas access to main memory was considered free. In today's in-memory database systems, the challenge stays more or less the same but moves one level up in the memory hierarchy [123]: How should an Aggregation operator be designed such that it uses the CPU caches efficiently to overcome the bottleneck of accessing the much slower main memory?

Traditionally, there are the two opposite approaches to implement this operator: Hashing and Sorting. HashAggregation inserts the input rows into a hash table, using the grouping attributes as key and aggregating the remaining attributes in-place. SortAggregation first sorts the rows by the grouping attributes and then aggregates the consecutive rows of each group. The question about which approach is better has been debated for a long time and different authors came to different conclusions about which is the better approach in which context [6, 20, 76, 96]. The consensus is that HashAggregation is better if the number of groups is small enough such that the output fits into the cache, and SortAggregation is better if the number of groups is very large. Many systems implement both operators and decide a priori which one to use. In this chapter we argue that in terms of data movement, the two paradigms are actually the same. By recognizing the fact that Hashing *is* Sorting, we can construct a single Aggregation operator with the advantages of both worlds.

As a first argument for our claim, we remind the reader of our result of the previous Chapter 4 about the complexity of Aggregation in terms of number of cache line transfers. The result is derived in a general external memory model [12], which holds in the cache setting as well as in the disk-based setting. As we show, there is a lower bound on the number of cache line transfers that *any* algorithm needs to incur under realistic assumptions, which matches the bounds of MultisetSorting in the common case. This means that the two textbook algorithms HashAggregation and SortAggregation can only exhibit a certain duality with respect to the number of groups if they are implemented naively. However, with two simple, commonly known optimizations, the respective drawbacks of both algorithms can be removed. The two approaches then match the lower bound and have exactly the same, inherent costs in terms of cache line transfers.

As a second argument for the similarity of the two approaches, we design an algorithmic framework that allows seamless switching between Hashing and Sorting during execution. It is based on the observation that Hashing is in fact equivalent to *sorting by hash value*. Since Hashing is a special form of Sorting, intermediate results of a Hashing routine can be further processed by a Sorting routine and vice versa. This allows us to apply state-of-the-art optimizations to both routines separately, but still combine them to benefit from their respective advantages. Furthermore, is Sorting by hash value an *easy* instance of Sorting, since Hashing makes the key domain dense and eliminates value skew. We also discuss how to apply this framework in the context of column-store database systems and just-in-time compiled query plans, which are the two most commonly used architectures for analytical workloads.

Furthermore, we address the "CPU friendliness" and "Parallelization" challenges (cf. Section 3.1) and show how to achieve very low constant factors for both the HASHING and the SORTING routine by tuning them to modern hardware. This also makes our analysis in the external memory model from the previous chapter meaningful, which assumes that computations are "free". The main techniques are wait-free parallelization, NUMA awareness, enabling super scalar instruction execution, and careful cache management. This is more important on modern in-memory database systems than on traditional disk-based systems, since much fewer CPU instructions can be executed in the time of a cache line transfer than in the time needed for loading a page from disk. With our careful tuning however, we are able to leave the memory access costs as the main remaining performance bottleneck.

Despite all tuning, there is an intrinsic reason why HASHING and SORTING have complementary performance vis-à-vis the locality of keys in the input: HASHING allows for *early aggregation* while SORTING does not. If several rows of the same group occur close to each other, HASHING aggregates them immediately to a single row. By reducing very early and possibly by large factors the amount of subsequent work, HASHING is much faster than SORTING in this case. In the other case however, i.e., in case of an input with few repeating keys, the additional effort of *trying* to aggregate is in vain, so regular SORTING without early aggregation is faster here. Our framework can exploit this complementarity by effectively detecting locality during execution and switching to the faster routine when appropriate, thus putting the insights of the theoretical analysis into practice. Finally, we carefully integrate an inexpensive cardinality estimator that allows us to effectively eliminate the potentially prohibitive resizing costs of the output data structure.

Putting everything together, we obtain a novel relational AGGREGATION algorithm that solves most of the challenges we identified in Section 3.1: It is optimal in terms of memory access complexity—independently and without prior knowledge of input skew and output cardinality—and has very low constant factors on modern hardware. It is cache-efficient, highly parallelizable on modern multicore systems, and operating at a speed close to the memory bandwidth. With our single, robust AGGREGATION operator, the possibly error-prone decision of the optimizer before the query execution is eliminated. We present extensive experiments on a variety of data sets comparing our implementation to the state of the art, which we are able to outperform by up to factor 3.7.

The rest of the chapter is organized as follows: In Section 5.2 we review state-of-the-art SORTING techniques, followed by the presentation of our algorithmic framework inspired by them in Section 5.3. We continue with tuning our routines to modern hardware in Section 5.4 and show how their advantages can be combined in Section 5.5. Section 5.6 presents our approach of integrating a cardinality estimator into our framework and Section 5.7 sketches how it can be extended to NUMA setups. The resulting AGGREGATION operator is evaluated in Section 5.8. Finally, we make some concluding remarks in Section 5.9.

5.2 REVIEW OF STATE-OF-THE-ART SORTING TECHNIQUES

In this section we review state-of-the-art SORTING techniques, from which we draw inspiration for the algorithm design of this chapter.

As we have shown in Chapter 4, in many realistic cases, the lower bound of AGGREGATION is the same as the one of MULTISETSORTING and that the bound is tight, at least for moderate numbers of processors. Consequently, using an optimal MULTISETSORTING algorithm is also optimal for AGGREGATION as far as the number of cache line transfers is concerned. Since SORTING is a very well-studied problem, we can reuse well-known techniques that make SORTING also run fast on real hardware. This is the underlying assumption of using external memory models for CPU caches, which completely ignores CPU costs: The model is only realistic if computations can be eliminated or hidden in large parts in the final implementation; only then, the fact is relevant that is provably impossible to reduce the memory accesses below a given lower bound.

A large body of prior work suggests that special techniques for sorting integers of dense domains outperform general purpose comparison-based sort algorithms and that the RADIXSORT family is the dominant such technique (see [188] for an overview). The main technique is to *avoid comparisons*—an intrinsic advantage of non-comparison based sort algorithms (although it is possible to eliminate the CPU costs of *branches* even for comparison based algorithms [160]). More recent literature studies tuning details such as fan-out, LSB vs MSB (least/most significant bits first), architecture specificities and parallelization [152, 92, 91, 182], rather than questioning their dominance. When the results are compared to QUICKSORT as a reference, RADIXSORT usually has a significant advantage (roughly 2x in [152], up to more than 4x in [91]). The work of Polychroniou and Ross [149] also comes to the conclusion that LSB RADIXSORT is the fastest known sort algorithm when it comes to dense domains.

In the next chapter, we use the insight that INTEGERSORTING techniques are often faster than comparison-based counterparts for the design of our AGGREGATION operator. We make them usable for arbitrary keys by projecting them into the dense domain of hash values.

Algorithm 4 Algorithmic Building Blocks

```
1:  func PARTITIONING(run: Seq. of Row, level)
2:      for each row in run do
3:          $R_h \leftarrow R_h \cup$ row with $h =$ HASH(row.key, level)
4:      return $(R_1, \ldots, R_F)$
5:  func HASHING(run: Seq. of Row, level)
6:      for each row in run do
7:          table.INSERTORAGGREGATE(row.key, row, level)
8:          if table.ISFULL() then
9:              tables $\leftarrow$ tables $\cup$ table ; table.RESET()
10:     return $(R_1, \ldots, R_F)$ with $R_i \leftarrow \bigcup_{t \in \text{tables}}$ GETRANGE(t,i)
```

5.3 ALGORITHMIC FRAMEWORK

In this section we present an algorithmic framework for an AGGREGATION operator putting the theoretical insights of Chapter 4 into practice. The main idea is to design the algorithm like an INTEGER-SORTING algorithm on the dense hash values with HASHING as a special case used for early aggregation. Furthermore, we show how to achieve wait-free parallelization of the algorithm. Finally, we also discuss how to fit the framework into popular processing models of modern database systems. In other words, in this section, we address the challenges for cache efficiency, parallelization, and system integration from Section 3.1.

5.3.1 *Mixing Hashing and Sorting*

As presented in the previous chapter, sort-based and hash-based AGGREGATION have the same complexity in the external memory model. This is very intuitive considering the fact that the two algorithms have the same high-level structure: they recursively partition the input—either by the keys of the groups or by their hash values—until there are few enough groups left to process each partition in cache. However, the two approaches are even more similar: the process of building up a hash table also partitions the input by hash value.

Consequently, we can define the following two partitioning routines, which will be the main building blocks of our framework and which are shown in Algorithm 4: plain partitioning by hash value called PARTITIONING (Line 1) and a partitioning routine based on the creation of hash tables called HASHING (Line 5). Both routines produce partitions in form of "runs":[3] PARTITIONING produces one run per partition by moving every row to its respective run. HASHING

[3] We consciously use the term "run" from the disk-based era, which was commonly used to denote temporary files for intermediate processing results on disk [76].

Algorithm 5 Aggregation Framework

```
1: AGGREGATE(SPLITINTORUNS(input), 0)                    ▷ initial call
2: func AGGREGATE(input: Seq. of Seq. of Row, level)
3:     if |input| == 1 and ISAGGREGATED(input[0]) then
4:         return input[0]
5:     for each run at index j in input do
6:         PRODUCERUNS ← HASHINGORPARTITIONING()
7:         $R_{j,1}, \dots, R_{j,F}$ ← PRODUCERUNS(run, level)
8:     return $\bigcup_{i=1}^{F}$ AGGREGATE($\bigcup_j R_{j,i}$, level + 1)
```

starts with a first hash table of the size of the cache and replaces its current hash table with a new one whenever it is full. Every full hash table is split into one run per partition—merely a logical operation since the hash values of one hash partition are stored in a consecutive range in the hash table.

Note that the working set of both HASHING and PARTITIONING is strictly limited to the CPU cache. The working set of PARTITIONING is limited through its partitioning fan-out, while HASHING has a limited working set because the size of the hash table is fixed to the size of the cache.

We can now combine these two building blocks into a recursive algorithm that is similar to both SORTAGGREGATION and HASHAGGREGATION at the same time. The algorithm is shown in Algorithm 5: The input is first split into runs. Then, each run of the input is processed by one of the two routines selected by HASHINGORPARTITIONING (line 6), which produces runs for the different buckets. Once the entire input has been processed, the algorithm treats all runs of the same partition as a single bucket and recurses into the buckets one after each other. With every step of the recursive partitioning, more and more hash digits are in common within a bucket, thus reducing more and more the number of groups per bucket. The recursion stops when there is a single run left for each bucket and in that run, all rows of the same group have been aggregated to a single output row.

Figure 8 illustrates how the run production of our framework works. The boxes represent runs and their background color the range of hash values of the contained rows. The relevant bits of the hash values are also given by numbers inside the boxes. The bits not playing a role for the placement within the box are marked as *x*; the underlined bits of the hash values are common for the entire box. For illustrative purposes, the runs in the figure are split in just two ranges by every recursive call.

Using the hash values as partition criterion has the advantage that it solves the problem of balancing the number of groups in the buckets. The hash function distributes the groups uniformly—it makes the key domain dense and hence the partitioning easier.

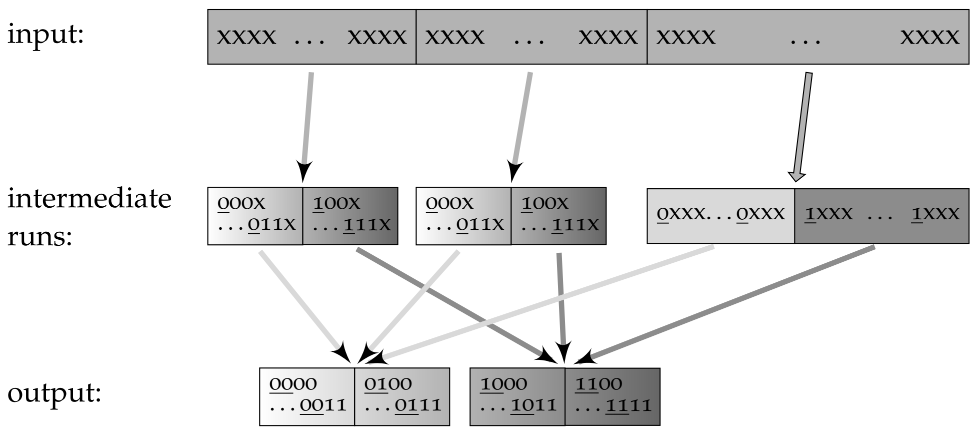


Figure 8: Recursive run-production of our algorithmic framework

The way we use HASHING has also the advantage that it enables *early aggregation* [76, 111]. In contrast to HASHAGGREGATION with pre-partitioning or traditional SORTAGGREGATION, we can now aggregate in *all* passes, not just the last. Since HASHING can aggregate rows from the same group, the resulting hash tables and hence the output runs are potentially much smaller than the input run, thus reducing the amount of data for subsequent passes by possibly large factors. It is important to understand that this does not change the number of cache lines needed in the worst case, but it is very beneficial in case of locality of the groups. So one might even wonder why we do not use HASHING all the time, similarly to recent work on disk-based systems [82]. As we show in the next section, on modern hardware, PARTITIONING can be tuned to a four times higher throughput than HASHING, which makes it the better routine in cases where early aggregation is not helpful.

The fact that our framework supports HASHING and PARTITIONING interchangeably not only makes the similarity between the two approaches obvious but also gives it the power to switch to the better routine where appropriate: In presence of locality or clusters, our framework can use HASHING, whose early aggregation reduces the amount of work for later passes. In the absence of locality, we can switch to the faster PARTITIONING instead. The switching can happen spontaneously during runtime, without loosing work accomplished so far, and without coordination or planning. Algorithm 5 still uses HASHINGORPARTITIONING as a black box, but in Section 5.5, we discuss several switching strategies.

We argue that the design of our framework resembles that of a sort algorithm. While the similarity to BUCKETSORT is obvious, our framework has similarities with other sort algorithms as well: Since the bucket of an element is determined by some bits of the hash value, our algorithm is in fact a RADIXSORT rather than a BUCKETSORT. This is how hashing turns AGGREGATION into an instance of (approximate)

IntegerSorting to make it easier than Sorting in general. Furthermore, it repeatedly merges intermediate runs from different parts of the input, so in a way the algorithm is also similar to MergeSort. Finally, hashing can be seen as a variant of InsertionSort, which is commonly used as base case in recursive sort algorithms [46].

An interesting interpretation is the following: The concatenation of the final runs of our algorithm is a hash table like HashAggregation would produce but it is built with a Sorting algorithm—which is actually much faster. This suggests that the optimal way to do Hashing *is* Sorting.

Finally, the fact that we mix Hashing and Sorting requires some technical attention: The buckets may contain rows that were just copied from the input, but also rows that are already aggregates of several rows from the input. In order to aggregate two aggregated values, one needs to use the so-called super-aggregate function [78], which is not always the same as the function that aggregates two values from the input. For example the super-aggregate function of `COUNT` is `SUM`. However, it is easy to keep some meta-information associated with the intermediate runs indicating which aggregation function should be used.

5.3.2 *Parallelization*

Apart from cache efficiency, the design of our framework addresses the challenge of multi-core parallelism inside operators, which we identified in Section 3.1. We only give an overview of our solution in this section and describe more details in Chapter 6.

Our framework allows for full parallelization of all phases of the algorithm: First, the main loop that partitions the input in Line 5 of Algorithm 5 can be executed in parallel without further synchronization since neither the input nor the output are shared among the threads. Second, the recursive calls on the different buckets in Line 8 can also be done in parallel. Only the management of the runs between the recursive calls (the $\bigcup$-operations in our pseudo-code) requires synchronization, but this happens infrequently enough to be negligible.

We use user-level scheduling to balance the two axes of parallelism as follows: we always create parallel tasks for the recursive calls, which are completely independent from one another, while we use work stealing to parallelize the loop over the input. It is important to see that the latter form of parallelization implies that several runs per bucket are produced (at least one per thread), which in turn implies another level of recursion. By using work stealing, our framework limits the creation of additional work to situations where no other form of parallelism is available. At the same time, parallelizing the main loop (Line 5) is required to achieve full parallelization. First, it is the only way to parallelize the initial call of the algorithm and second

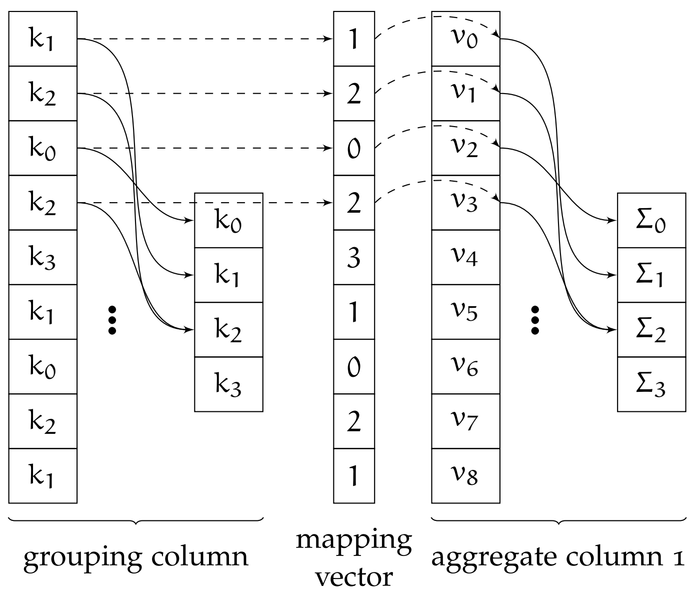


Figure 9: Column-wise processing.

it allows for full parallelization even in presence of heavy skew: the buckets after the first call can be of arbitrarily different sizes because even an ideal hash function only distributes the *groups* evenly over the buckets, but does not affect the distribution of *rows* into these groups (which is given by the input). With work stealing in the main loop however, our framework can schedule the threads to help with the large buckets once they have finished their own recursive call.

5.3.3 *System Integration*

In Section 3.1 we identify system integration as a challenge for Aggregation operators. Hence, we now discuss how to integrate our framework into the two prevailing processing models of analytical in-memory database systems, namely column-wise processing and just-in-time compiled (JiT) query plans.

Column-wise processing is the traditional processing model of column-store database systems. It imposes some specific requirements for the implementation of Aggregation operators. While our framework fulfills these requirements, this is not true for all Aggregation algorithms proposed in the past. In the column-store architecture, the questions arise *when* and *how* the aggregate columns should be processed with respect to the grouping columns. These questions have been discussed in the literature for the column-store architecture in general [31, 32, 124], but there are some additional aspects specific to Aggregation.

One possibility for column-wise processing is to process all columns at the same time, similarly to a row store. An Aggregation operator would read the values of a single row from one column after another, compute the aggregates, and then store them in their respective result columns one after each other. This approach is known to have the disadvantage that the data is not processed in tight loops [32], which results in considerable performance deterioration on modern hardware. Furthermore, it effectively decreases the size of the cache during aggregation: Since all relevant attributes of a row together are larger than just a single attribute, fewer elements fit into cache, so an AGGREGATION operator needs more passes for cache-efficient processing.

Another possibility is to do the processing one column at a time, like it is done for example in MonetDB [31, 124]. With this approach, AGGREGATION is split into two operators as illustrated by Figure 9: The first operator processes the grouping column and produces a vector with identifiers of the groups and a mapping vector, which maps every input row to the index of its group; the second operator applies this mapping vector by aggregating every input value with the current aggregate of the group as indicated by the mapping vector and is executed once for each aggregate column. This approach is known to have the disadvantage to require additional memory access to write and read the mapping vector. Furthermore, it ignores the insights of our analysis for cache-efficient AGGREGATION: if we aggregate the input values directly to their group in the output column, we get the same sub-optimal memory access pattern as HASHAGGREGATION, producing close to one cache miss for every input row for large outputs. Since there are often many more aggregate columns than grouping columns, this would have a worse impact on performance than inefficient processing of the keys.[4]

The state-of-the-art column-wise processing model was introduced in MonetDB/X100 [32] and combines the advantages of the above two possibilities by interleaving the processing of the different columns in blocks of the size of the cache. Applied to AGGREGATION, this allows processing the columns in tight loops without materialization of the mapping vector to memory. However, we also need to adopt this model *inside* the AGGREGATION operator to make it compatible with our recursive run production. Consequently, our framework operates as follows if used for column-wise processing: While producing a run of the grouping column, both HASHING and PARTITIONING produce a mapping vector as depicted in Figure 9, but *only for this run*. This mapping is then applied to the corresponding parts of the aggre-

4 Note that there are cases where this may be—contrary to our reasoning—the best strategy: In a complicated query with many large JOINS, it may be better to do all JOINS and the final AGGREGATION on the key columns alone, and then re-assemble the rows in a final step of “late materialization” (cf. work by Abadi et al. [2]). The re-assembly has a random access pattern with one cache miss per element, but this may be faster than materializing all columns repeatedly after every JOIN. However, deciding which strategy is better as well as handling AGGREGATION with late materialization are out of the scope of this thesis.

gate columns. When the corresponding runs of all columns have been produced, the framework continues with the processing of the rest of the input.

One important aspect of this discussion is that not all techniques for Aggregation proposed in prior work are compatible with column-wise processing. Our two routines, Hashing and Partitioning, move every input element directly to its final location, so it is easy to create the required mapping vector. However, it is unclear how to fit some other routines into this scheme, for example vectorized sorting networks recently used for Joins [20] or software-caching used for cache-efficient Aggregation in row stores [42].

Recent work [140, 67, 138] promises to replace column-wise processing by a processing model based on just-in-time compilation (JiT). In this model, each pipeline of the execution plan of a query is compiled into a fragment of machine code just before execution, thus enabling processing in tight loops without decoupling the processing of different columns. It is straightforward to fit our framework into this model: The main part of the operator, i.e., the initial call to the Aggregate function of Algorithm 5, is compiled into the pipeline fragment including (and ending with) the Aggregation operator. When this pipeline ends, all data resides in intermediate runs in buckets of the first level of the recursion. For the recursive function calls, a second code fragment is compiled, which only contains the code to process the buckets further. Both fragments contain the code path of both the Partitioning and the Hashing routines from Algorithm 4. Since the runtime decision between the two paths is the same for the entire run, it is easy to predict by the hardware and does hence not hurt performance.

5.4 minimizing computations

In this section we study the details of the two routines used in our algorithm: Hashing and Partitioning. This addresses the challenge of CPU friendliness from Section 3.1. The goal is to bring the behavior of our implementation on real hardware as close as possible to the idealized external memory machine model. Our analysis and the algorithmic framework built upon it are only relevant if we turn the movement of the data into the dominant part of the execution time by reducing the computational overhead to a minimum (i.e., ideally by making it "free"). We show that this is much more difficult in the cache setting of modern hardware than it used to be in the disk setting, since the gap between fast and slow memory is much narrower and small differences in the implementation can change the performance by factors.

The microbenchmarks in this section are run in the same experimental setup as the experiments in Section 5.8.

5.4.1 *Minimizing CPU costs of Hashing*

We conducted a performance comparison of several hash table implementations. It turned out that the simplest approach has the lowest CPU overhead: a single-level hash table with linear probing, similar to the state-of-the-art `dense_hash_map` of Google.[5] We fix the size of hash table to that of the L3 cache and consider it full at the very low fill rate of 25 %. With this configuration, collisions are very rare or even non-existing with high probability if the number of groups is more than two orders of magnitude smaller than the cache, so no CPU cycles are lost for collision resolution. The apparent waste of memory is in fact negligible because it is limited to one or very few hash tables per thread in our final algorithm presented in the next section. Interestingly, this is the opposite of what Barber et al. [23] recently proposed for JOINS, where denser storage increased performance. We tried out many different hash functions that are popular among practitioners and found that for small elements, MurmurHash2[6] is the fastest. On a technical note, we adapted the linear probing to work within blocks, such that we can cleanly split a table into ranges for the recursive calls. The final insertion costs of our implementation are below 6 ns per element. This is roughly four times more than an L1 cache access, but more than an order of magnitude faster than out-of-cache insertion, where CPU costs are dwarfed by the costs of cache misses and our reasoning in the external memory model is meaningful.

5.4.2 *Minimizing CPU costs of Partitioning*

To minimize the CPU costs of the PARTITIONING routine, we use a technique known from INTEGERSORTING for dense domains. As discussed in Section 5.2, this kind of sort algorithms is usually faster than comparison-based sorting, except special cases such as sorting almost sorted data [38]. With PARTITIONING, INTEGERSORTING is made branch-free and even comparison-free, which eliminates most CPU costs of comparison-based sort algorithms.

The key idea is to use a technique called "software write-combining", first described by Intel [85] and used by various other authors [20, 161, 182, 149, 162]. Software write-combining is designed to avoid the *read-before-write* overhead and to reduce the number of TLB misses inherent in partitioning [122], which writes to a high number of memory pages. It consists in buffering one cache line per partition, which is flushed when it runs full using a non-temporal store instruction that bypasses the cache. This scheme works best with 256 partitions, so we use this number to split runs into ranges in our framework. Since the final size of the partitions is unknown before

5 https://code.google.com/p/sparsehash/
6 https://code.google.com/p/smhasher/

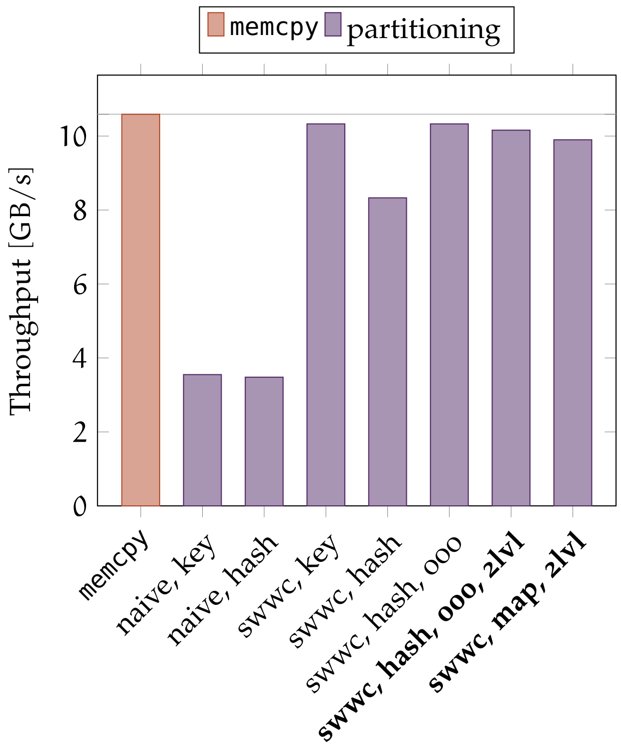


Figure 10: Microbenchmark of degenerated partitioning routines.

processing, most authors start with a counting pass to determine output positions. Wassenberg et al. [182] eliminate this pass using a trick with virtual memory by over-allocating every partition so that it can hold the entire input. This is not possible with the memory management of industry-grade database systems, but using a two-level data structure, a list of arrays, has the same benefit and only very low overhead, as we show below. We use this technique for processing not only the grouping column but also the aggregate columns, as discussed in Section 5.3.3, since their memory access pattern is equivalent.

We conducted a series of microbenchmarks to measure the impact of our optimizations. Figure 10 shows the (payload) bandwidth of different versions of the PARTITIONING routine on uniformly distributed random data. The first bar shows the bandwidth of a self-implemented `memcpy` using non-temporal store instructions as a reference. It represents the maximum memory throughput achievable in practice by using a sequential access pattern—PARTITIONING routines, with their inherently more difficult access pattern, can only be slower. The next two bars show the throughput of a naive partitioning scheme, one partitioning according to some bits of the keys themselves (called *key*) one partitioning according to bits of the hash function (called *hash*). The difference between the two is only small as the throughput is mainly limited by the inherent TLB misses mentioned before. The next two bars show how software write-combining considerably improves performance (called *swwc*): The *key*-variant is 2.9 times faster than the naive counterpart. The variant partitioning by hash value seems to suffer from the computational overhead though. However, we can benefit from out-of-order execution by manually unrolling the main loop into blocks of 16 elements, which are first all hashed and then all put into their partition buffers. The next bar (denoted *ooo*) shows that we gain 24 % throughput with this optimization, thus achieving a 3.0 times higher throughput than the naive partitioning routine. Finally, we replace over-allocated output partitions by the two-level data structure, which lowers performance by roughly 2 %. This final routine, *swwc, hash, ooo, 2lvl*, runs at 97 % of the bandwidth of our `memcpy`. The last bar shows the bandwidth of applying the mapping vector to an aggregate column using software write-combining and our two-level data structure (denoted *map*). Since reading the mapping vector adds memory traffic that we do not count as application bandwidth, the bandwidth here is slightly lower than that of the previous variant, namely 93 % of our measured memory bandwidth.

To sum things up, like in the case of HASHING, PARTITIONING of both grouping and aggregate columns can be tuned to modern hardware such that the inevitable movement of the data remains the dominant part of the processing time.

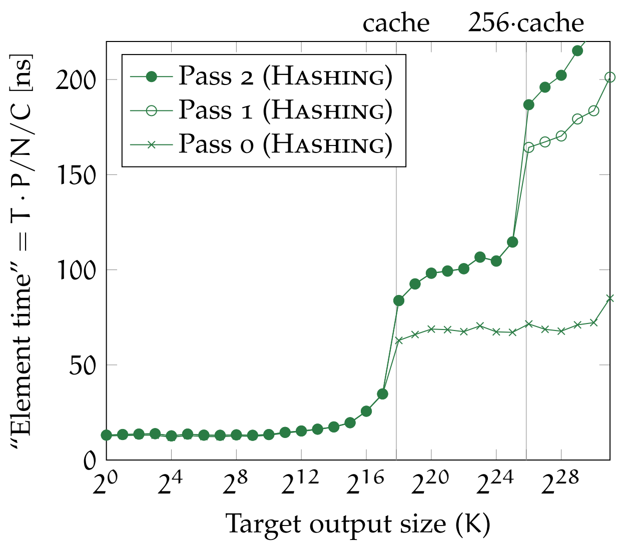
5.5 ADAPTATION TO LOCALITY

5.5.1 *Adaptation Mechanism*

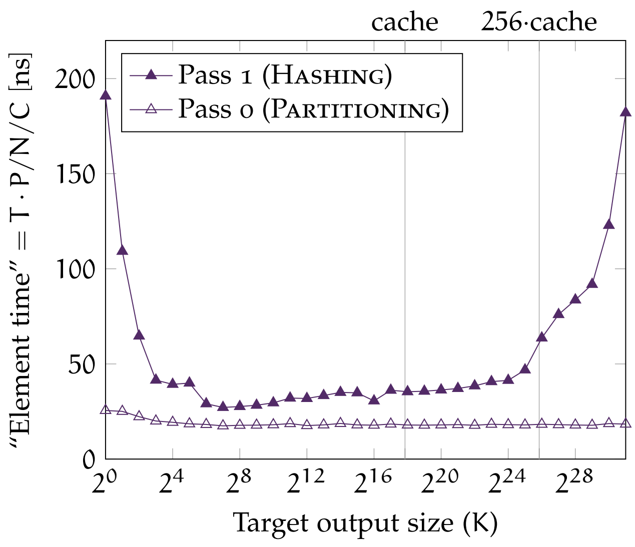
In the previous sections, we describe an algorithmic framework for designing an AGGREGATION operator similar to a SORTING algorithm and how to reduce the CPU costs of two possible subroutines. In this section, we answer the remaining question of when to select which of the two. This addresses the challenge of adaptive processing from Section 3.1.

To that aim we present a series of experiments with naive strategies for selection of one of the two routines that illustrate their respective performance characteristics. Figure 11 shows the results in terms of *element time*, a metric that expresses the time a single core spends with each element, which is formally defined in Section 5.8. We vary the parameter that has the strongest impact on performance: the number of groups K that the N input records belong to, which is the number of records in the result.

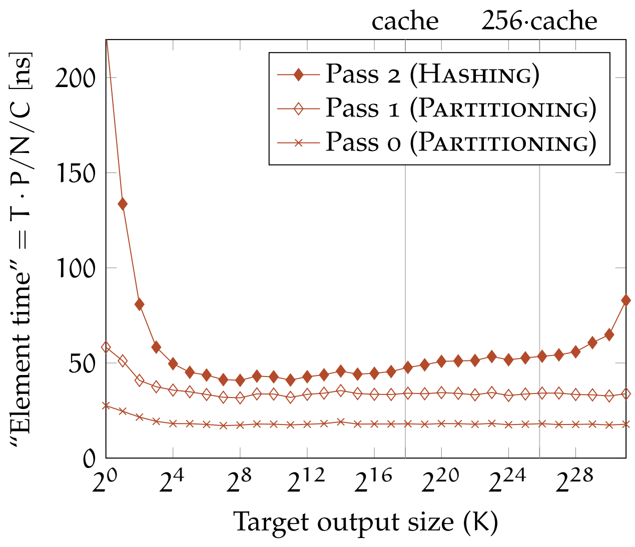
In HASHINGONLY the only subroutine used is HASHING (Figure 11a), whereas with PARTITIONALWAYS, the input is always preprocessed by one or two passes of PARTITIONING before a final HASHING pass (Figure 11b and 11c). To keep our implementation simple, we only allow a single HASHING pass by exceptionally letting its hash tables grow larger than the cache. This prevents full parallelization for very small



(a) HashingOnly



(b) PartitionAlways (2 passes)



(c) PartitionAlways (3 passes)

Figure 11: Breakdown of passes of illustrative Aggregation strategies using $P = 20$ threads.

K and cache misses for very large K, but these effects do not occur in our final algorithm. The experiments are run on uniformly distributed data.

The first observation in this experiment is the fact that HASHING-ONLY automatically does the right number of passes: If $K <$ cache, it computes the result in cache. The subsequent merging of the runs of the different threads is insignificant due to their small size and thus not visible in the plot. Once $K >$ cache, HASHING recursively partitions the input until the result is computed in cache and the recursion stops automatically. For PARTITIONALWAYS this is not the case. Since it does not aggregate during partitioning, it can only be used as preprocessing and external knowledge is necessary to find the right depth of recursion before the final HASHING pass.

The second observation is that PARTITIONING is *much* faster (by more than factor four in our experiments) than HASHING if $K >$ cache, i.e., if the latter produces more than one run. In this case HASHING suffers from its non-sequential memory access and wasted space and hence wasted memory transfers intrinsic to hash tables. Furthermore, as discussed in Chapter 4, as soon as there are only slightly more groups than fit into one hash table, chances are very low to find two elements with the same key, so the amount of data is not reduced significantly. In contrast, PARTITIONING achieves a high throughput independently of K thanks to the tuning from the previous section.

In the case of uniformly distributed data, the best strategy is obvious: use PARTITIONING until the number of groups per partition is small enough such that HASHING can do the rest of the work in cache. However, it is not clear how to find out when this is the case if K is not known. Furthermore, the best strategy is less obvious with other distributions: consider a clustered distribution with a high locality where each key mostly occurs in one narrow region of the input. HASHING is then able to reduce the amount of data significantly *although* the entire partition has more groups than fit into cache. Hence, HASHING can be the better choice even before the last pass if the ratio of input data size to output data size high enough.

This leads us to defining an ADAPTIVE strategy. Figure 12 illustrates the idea. The algorithm starts with HASHING. When a hash table gets full, the algorithm determines the factor $\alpha := \frac{n_{in}}{n_{out}}$ by which the input (run) has been reduced, where n_{in} is the number of processed rows and n_{out} the size of the hash table. If $\alpha > \alpha_0$ for some threshold α_0, HASHING was the better choice as the input was reduced significantly, so the algorithm continues with HASHING. Otherwise, it switches to PARTITIONING. The parameter α_0 balances the performance penalty of HASHING compared to PARTITIONING with its benefit of reducing the work of later passes. We show how we determine this machine constant later in this section. When enough data was processed with the faster PARTITIONING routine such that the overhead of the (inad-

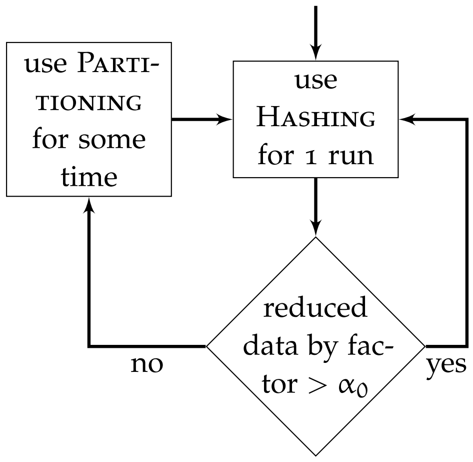

use PARTITIONING for some time
use HASHING for 1 run
reduced data by factor > α_0
no
yes

Figure 12: ADAPTIVE strategy

equate) HASHING is amortized, i.e., when $n_{in} = c \cdot |\text{cache}|$ for some constant c, the algorithm switches back to its initial mode in case the distribution has changed. In experiments below, we show how we can find a good value for c that balances amortization effect and reactivity to distribution changes.

Figure 13 shows the performance of ADAPTIVE compared to the illustrative strategies from Figure 11. It shows that ADAPTIVE automatically partitions the input using PARTITIONING until HASHING can process each partition in cache—without knowing K in advance. Consequently, its performance corresponds piecewise to the best of the other strategies. If $K < |\text{cache}|$ (or $K/256^i < |\text{cache}|$ for pass i), each thread only works with a single hash table, which never runs full. If however the input does not fit into one hash table, the input is partitioned with the faster PARTITIONING routine first. The fact that PARTITIONING is interleaved with occasional HASHING to check whether the distribution has changed has only low overhead and is barely noticeable for $K < 256 \cdot |\text{cache}|$. In the following section, we show that ADAPTIVE not only works well on uniform data, but has a robust performance on many distributions.

We see the main advantage of our approach in the fact that it combines the respective advantages of two complementary routines by switching between them based on a simple, local criterion. No completed work is ever thrown away; no extra work or preprocessing is necessary; no potentially imprecise information from the optimizer is needed; no synchronization is needed among the threads. In fact, the different threads do not even need to take the same decision: they can benefit from changing locality or clusteredness in the input by aggregating where the locality is high and partitioning first where it is low.

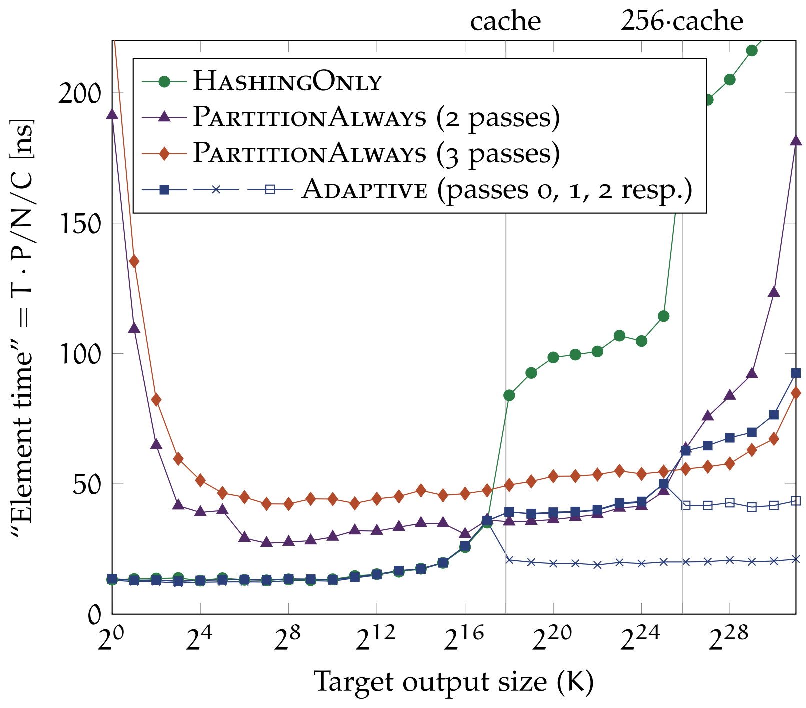


Figure 13: Adaptive strategy in comparison with HashingOnly and PartitionAlways (2 and 3 passes) using $P = 20$ threads.

5.5.2 *Tuning of Algorithm Constants*

In the remainder of this section, we show how we empirically determine the constants of the Adaptive algorithm.

5.5.2.1 *Switching Threshold α_0*

The parameter α_0 balances the performance penalty of Hashing compared to Partitioning with its benefit of reducing the work of later passes. In order to determine its value for our system, we run both HashingOnly and PartitionAlways on data sets with varying skew using our parameterized data generators, presented later in Section 5.8.5. For every data set, we observe the value of $\alpha = \frac{n_{in}}{n_{out}}$ in the traces of HashingOnly. For unclustered distributions like uniform, the transition from $\alpha = \infty$, where all elements fit into the single hash table, to the other extreme, $\alpha = 1$, where all keys are distinct, is very sharp and only small values of α occur at all. Almost any value of α_0 works in these cases.

However, the three distributions moving-cluster, self-similar, and heavy-hitter can be parameterized to a large range of degrees of spatial locality. In Figure 14, we plot the run times of HashingOnly and PartitionAlways on different data sets with these distributions as function of the observed values of α. As expected, for high values of α and any distribution, HashingOnly outperforms PartitionAlways.

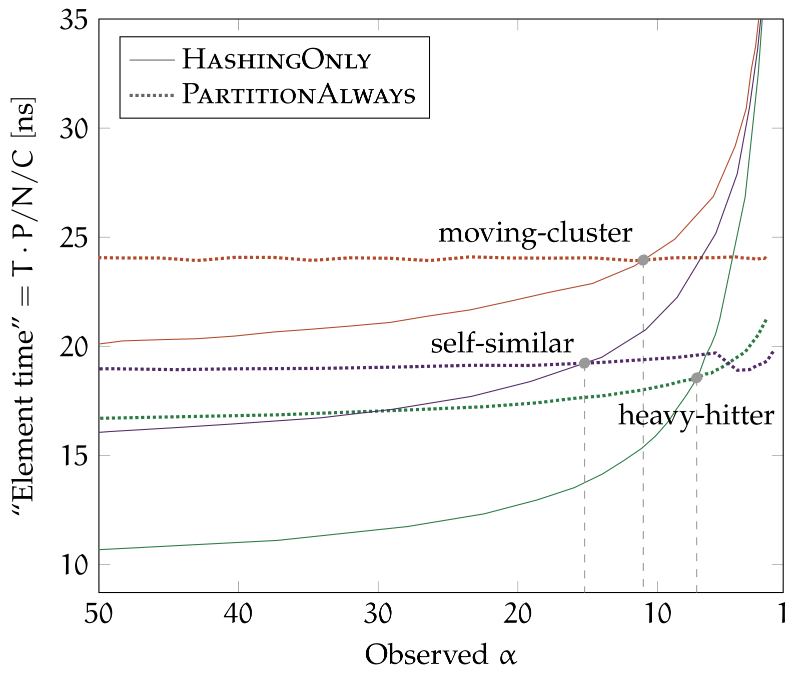


Figure 14: Determining the cross-over of HashingOnly and PartitionOnly.

In these cases the input of the first pass can be reduced by large factors, so it exhibits enough spatial locality for the Hashing routine to be beneficial. For values of α approaching 1, the order of the routines is inverted: it is better to ignore low locality and use Partitioning instead. The desired threshold α_0 should separate the first case from the latter. We observe that the respective lines of a particular distribution all intersect in the range of $\alpha \in [7, 16]$ and use the α with the smallest overall error as value for α_0 in our system, which is roughly 11.

5.5.2.2 Hashing *vs* Partitioning (c)

The constant c in the Adaptive algorithm controls how long the Partitioning routine is run before the algorithm switches back to Hashing, which happens after $c \cdot \text{cache}$ processed rows. Figure 15 shows the impact of c on the run time of Adaptive for different K and the uniform data set. For $K < \text{cache}$, such as $K = 2^{10}$ in the plot, the algorithm never switches to Partitioning in the first place, so c does not have any impact. In the extreme case of $c = 0$, the algorithm degenerates into HashingAlways, which is quite slow for $K > \text{cache}$ (cf. Figure 13). The larger c gets, the more data is processed with the faster Partitioning routine, so the more the performance of Adaptive approaches that of PartitionAlways. This suggests that $c = \infty$ is the best choice, i.e., to never switch back.

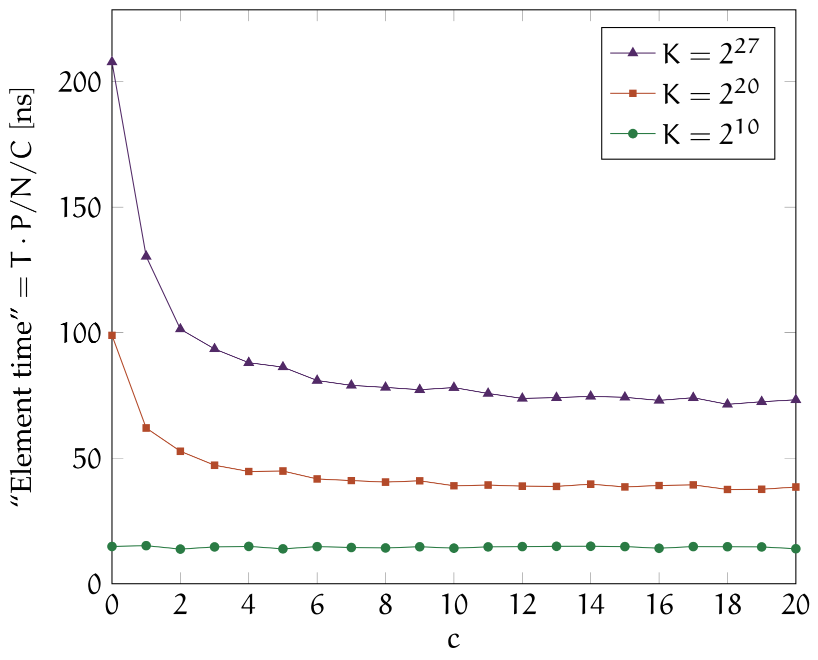


Figure 15: Impact of tuning constant c on the run time of ADAPTIVE.

However, smaller c have the benefit to be able to adapt to changing distributions, which are likely to occur after `UNION ALL` operators or as an artifact of reordering of rows for compression purposes [115]. This benefit is hard to quantify because it depends on the database system and typical workloads. Because the benefit diminishes with growing c, a smaller c seems affordable anyway: For $c = 5$, the difference to PARTITIONALWAYS is 17 % for $K = 2^{20}$ (19 % for $K = 2^{27}$), while it is 5 % (11 %) for $c = 10$ and still 4 % (5 %) for $c = 20$.

In summary, c allows us to choose a trade-off between robustness to changing distributions and maximum throughput. We choose a rather performance-oriented value of $c = 10$ for the experiments of this chapter, which are all done on data sets of a single distribution, but suggest a slightly lower value for productive systems.

5.6 ADAPTATION TO THE OUTPUT SIZE

5.6.1 *Motivation and Adaptation Mechanism*

While the ADAPTIVE strategy presented in the last section automatically chooses the most suited routine among HASHING and PARTITIONING, there is still a final, subtle challenge for our operator before it achieves complete independence from the optimizer. It is the challenge of knowing the size of the runs of HASHING when this routine is used to produce the result, i.e., when it is used in the last level of recursion. In the levels of recursion *before* the last the runs are fixed to the size of the cache to ensure cache efficiency. However, using this approach in *last* level can lead to a considerable waste of memory if the number of groups at that point of the recursion is much smaller than the cache. The folklore approach of filling a hash table of unknown size is to create a new data structure of twice the size of the current one every time a certain fill ratio is reached and to transfer all elements to the new copy. This process is called *rehashing*. However, for an AGGREGATION query with $K \approx N$, this effectively corresponds to an additional pass over the input, which increases the run time considerably. Hence, rehashing is not a viable solution either. Instead, we would want to know the size of the run before we create it, such that no resizing is necessary, but without require *a priori* information from the optimizer.

We propose to solve this problem by using a streaming algorithm that estimates the cardinality of the output during the first level of recursion our algorithm. When the final runs need to be constructed, which happens at the earliest just after the first level is completed, we thus have an estimate of their size. An approximate answer is perfectly suitable for this purpose: the sizes of our hash tables are always powers of two in order to avoid modulo computations, and occasional rehashing due to underestimations has only little impact on performance if they are not too frequent.

Cardinality estimation of streams has been studied extensively in the past. Appendix B gives a short overview. They all have in common that they visit each element exactly once, accumulate some approximate information, and finally derive some cardinality estimate of the data stream they have seen. For our purpose, PROBABILISTICCOUNTING with stochastic averaging [66] (PCSA) seems to provide the best trade-off: With 32-way accuracy, it achieves a standard error of 13 %. While having reasonable precision, this takes up to 128 B, i.e., two cache lines, and hence ensures that the estimator does not interfere with the careful cache management of the routines described in Section 5.4. Furthermore, it can be integrated into the routines with very little extra computations per record. PCSA is based on counting leading zeros of a single hash value, which we calculate for every record anyways. We thus directly present the hash value of each record that is processed in the first level of recursion to the estimator. Only four more instructions are then necessary for each record, including the counting the leading zeros in the hash value which is available as hardware instruction on many processors.[7] The details of PCSA are explained in Appendix B.

Two more properties of PCSA are worth noting: First, it is easily parallelizable. Every thread can maintain its own private state and at any given point, when a global estimate is required, one can combine the respective states to obtain the result a single thread would have

7 On the X86 architecture, the `lzcnt` instruction does this calculation. See `https://software.intel.com/sites/landingpage/IntrinsicsGuide/` or similar for details.

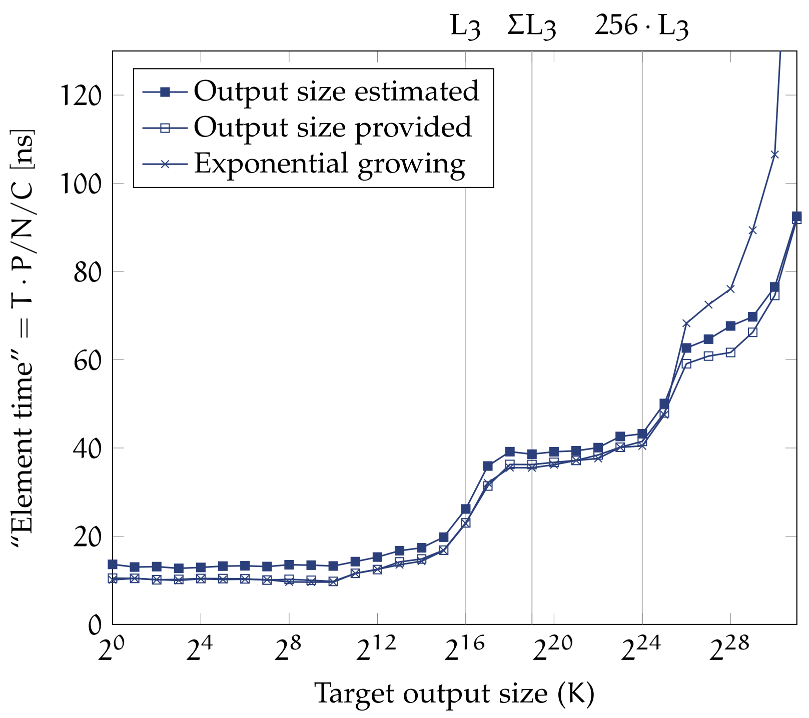


Figure 16: Impact of output size estimation on ADAPTIVE using $P = 20$ threads.

computed. Second, we can exploit the fact that at any point their states can be used to estimate the number of distinct elements *so far*. This is useful in the next chapter, where we overlap the processing of the different passes, which requires an estimate of the output size even before all input records have been processed for the first time.

Our approach has remote resemblance with INSPECTORJOINS proposed by Chen et al. [40], which collects statistics during the partitioning phase of the join in order to select the most appropriate join phase algorithm afterwards.

5.6.2 *Evaluation*

We now quantify the impact of PCSA experimentally. To that aim we compare the run time of three configurations of our ADAPTIVE operator on uniform data: First, as a reference, we run the operator without estimator, but provide the exact output size as a parameter so that it can immediately allocate the output data structure with the correct size. Second, we run it without estimator and external information, so that the output data structure has to be grown exponentially whenever it runs full. Third, we use the prediction of the estimator as described previously. Figure 16 shows the result.

By comparing the two configurations without estimator, we can see the costs due to growing the output data structure, which is necessary if we have no information about the output size. There is no visible difference between the two for $K < 2^{26} = \frac{N}{64}$, so the costs for growing until that small size are dominated by the much larger regular processing costs. For larger K, the growing costs continuously increase: while they amount to 15 % of additional costs for $K = 2^{26}$, they increase the element time from 92 ns to 198 ns for $K \approx N$ (the latter point is outside the plot), which corresponds to 115 % extra run time. This confirms the importance of knowing the output size and emphasizes that ignoring this aspect, as done by some work proposed previously, is not acceptable in practice.

We now compare the configuration with estimator with the one where the output size is known and the estimator deactivated. We can see that variant with the estimator is roughly 2.8 ns slower. This corresponds to the computational overhead of counting the leading zeros and storing the result. It represents around 30 % of additional run time for small K, but since it is a constant overhead, it diminishes relatively with the increasing run time of larger K. Remember that the experiments in Figure 16 are done with a query without grouping columns in order to make the impact of the estimator more visible. The additional processing costs of grouping columns will further decrease the relative costs of the estimator.

More importantly, other than the constant overhead of the estimator, there is no difference anymore, which means that we completely eliminated the costs of growing the output. Our approach thus achieves almost optimal performance even for very large K by only adding a small constant overhead.

The fact that the overhead is relatively high for small K can be mitigated by a small, pragmatic modification: We could activate the estimator only in the PARTITIONING routine of our algorithm. This way, it would never be used on inputs where K is small compared to N. In these cases, the costs of growing are negligible anyways. Only when K is large enough such that the PARTITIONING routine is used, we have to pay the overhead of the estimator. This approach does not take into account those keys that are only processed by the HASHING routine. However, that routine only processes a small fraction of the rows if K is large, so we expect the estimate to be only off by little. We leave the implementation of this modification and its experimental evaluation as a small open problem for future work.

5.7 EXTENSION TO NUMA AND REMOTE MEMORY

Until here, our framework assumes uniform access cost to memory. As we discuss in Section 3.5 however, this may not be the case for very large machines or even clusters, where accessing remote memory is much slower than accessing local memory. In order to scale to such large machines, the challenge becomes In this section, we thus discuss how to extend our framework with NUMA awareness. We believe that our solution is also a valid approach for clusters with fast interconnect networks, which behave similarly as NUMA sockets, but with somewhat different constants.

In order to reduce communication between the sockets, there are two well known strategies (cf. Section 3.4.1): To use the names of Graefe [76], in TwoPhaseAggregation on the one hand, first every node aggregates its own input, then these intermediate results are exchanged among the sockets such that each socket gets all results of a subset of the groups, which are finally aggregated by that socket. The Repartition algorithm on the other hand exchanges the input directly without prior, local aggregation, before each socket aggregates locally the groups it is responsible for. If the number of groups K is small, TwoPhaseAggregation is faster because it reduces the amount of data transferred over the network. With very large K, the effort of doing an initial, local aggregation may not be outweighed by the benefit of data reduction, so Repartition is faster in this case. These complementary choices were known in the early parallel database systems [76, 163], and recently confirmed to exist in modern NUMA systems as well [116]. As with the choice between HashAggregation and SortAggregation, the better strategy depends on K, which we only know at runtime. The question is thus how to select the right strategy *adaptively*.

We argue that the adaptive mechanism of the previous section can be applied to this question almost without modification. The general idea is to start like TwoPhaseAggregation until we have reason to believe that this was not the best strategy, at which point we operate like Repartitioning. The resulting algorithm basically works as follows. First, the threads of each socket start a local Aggregation on the input of their socket as described in the previous section: they process the input, using either Hashing or Partitioning depending on by how much the data was reduced, and recurse into the buckets thus produced. The Hashing routine is now modified in such a way that it stops as soon as the run it produces becomes larger than a certain threshold (which we define more formally below). If that happens, we switch to Partitioning for some time just as in the single-socket algorithm. The intermediate result of the local computation is thus a *partial* Aggregation, where the runs consist in a mix of hash tables and partitions and potentially contain several records that belong to the same group. When the local (partial) Aggregations are done, the buckets thus produced are repartitioned among the sockets such that each socket gets all records of a subset of the buckets. Finally, each socket aggregates all records of each of its buckets locally using the single-socket algorithm of the previous section.

Like the cache-efficient algorithm of the previous section, this mechanism makes us operate automatically in the mode of the better of two approaches, in this case REPARTITION and TWOPHASEAGGREGATION: If the number of groups K is small, the threshold of the HASHING routine is never reached, so each socket produces a full local AGGREGATION and the communication volume of the subsequent repartitioning is minimized. This is exactly how TWOPHASEAGGREGATION behaves. If the number of groups K is too large for the local AGGREGATIONS to be beneficial, then the HASHING routines soon interrupts because the runs get larger than the threshold. Hence, most runs are produced with the much faster PARTITIONING routine—at the expense of a slightly higher communication volume—which corresponds to how the REPARTITION algorithm behaves. Note that, unlike one might first think, the effort of producing many partitions locally does not represent additional work: The AGGREGATIONS after the repartitioning of the data need small enough partitions in order to aggregate them in cache anyways. Had the partitions not been produced before exchanging them, then this work would have to be done afterwards.

The threshold for the HASHING routine to stop is defined as follows. We assume that pre-aggregation is not helpful if $K \geqslant \beta_0 N$ for some machine constant $\beta_0 \in (0, 1]$ (N is the number of records in the input *of that socket* and K the number of groups among these N records). In terms of the current run, this is the case if the current run is larger than $\beta_0 \frac{N}{F^r}$, where F is the fan-out ($F = 256$ in our implementation) and r the depth of the recursion (starting with $r = 0$ for the input). Like α_0, β_0 is a machine constant that balances computation with communication, which has to be determined by experiments similar to those we describe in Section 5.5.2.1.

Finally, a few known techniques are required for full NUMA awareness: In order to cope with unequally distributed work or differences in available compute resources on the sockets, work stealing across sockets may be beneficial. In order to avoid unnecessary cross-socket communication, threads from a socket should only start stealing from other sockets when their local work is completely finished. However, it may be beneficial to limit work stealing to cases where the imbalance is so high that more than one pass is remaining on the overloaded socket. Otherwise, the additional memory costs induced by the stealing may outweigh the benefit of off-loading the work (see work of Psaroudakis et al. [151] for a more detailed description of this effect). Furthermore, as Li et al. [116] point out, it is important to cleverly schedule the partitioning in a topology-aware way in order to avoid that some sockets or connections are saturated while others are idle. Last but not least, it is also important to start communications before all computations are finished. Otherwise, the interconnects between the sockets are idle during the computation phases, which makes suboptimal use of this scarce resource (Balkesen et al. [19] do something similar for JOINS). In our algorithm, this can be achieved by processing the buckets in the same order on all sockets and starting the repartitioning and post-aggregation of a bucket as soon as it finishes. This way processing of the buckets is implicitly synchronized, so the communication of each bucket can be overlapped with the computation of other buckets. We only sketch the application of these techniques to our operator because we expect it to be similar to those described in the prior work we cited and difficulties mainly to arise in implementation details, which we leave as future work.

In summary, our adaptive mechanism for cache efficiency on memory with uniform access cost from the previous section extend to communication efficiency in a NUMA setup. As before, it provides a means to switch between two complementary strategies based on a very simple, local criterion and different sockets may take different decisions if the characteristics of their respective input vary. The mechanism does not require synchronization, sampling, or planning, but instead makes our algorithm adapt to the data at runtime. This way it behaves automatically like the best algorithm for the respective situation.[8]

5.8 EVALUATION

In this section we evaluate the effectiveness of our algorithm design, assess the quality of our implementation, and compare the performance of our operator with previous work.

We implemented our algorithm for column-wise processing and systems where the impact of NUMA is negligible. We argue that the experiments have a certain validity for the JiT processing model as well: Where not otherwise mentioned, the experiments are run just on the grouping column, so the inner loops of both processing models are exactly equivalent. The implementation of the NUMA extension is left for future work.

5.8.1 *Test Setup*

We run the experiments on two Intel Xeon E7-8870 CPUs[9] with 256 GiB of main memory. They run at 2.4 GHz and have 10 cores each. Each core has 64 KiB of private L1 cache, 256 KiB of private L2 cache, and access to a shared 30 MiB on-chip L3 cache (3 MiB per core). The TLB of the CPUs have two levels, the first of which have 64 entries for data and 128 for instructions and the second 512 entries for both com-

8 Note that in this work, we concentrate on the hard case, where no (potentially fuzzy) initial partitioning of the input is known. If such a partitioning is known, communication may not be necessary at all because then we know that all records of each group are in the input of the same socket. TRACKJOIN [150] and NEOJOIN [155] were designed to reduce communication for JOIN in these situations. Furthermore, for cases where $K \approx N$, it may be beneficial to detect distinct records in a pre-processing phase and only to exchange the remaining ones, such as we propose in prior work [83].

9 `http://ark.intel.com/products/53580`

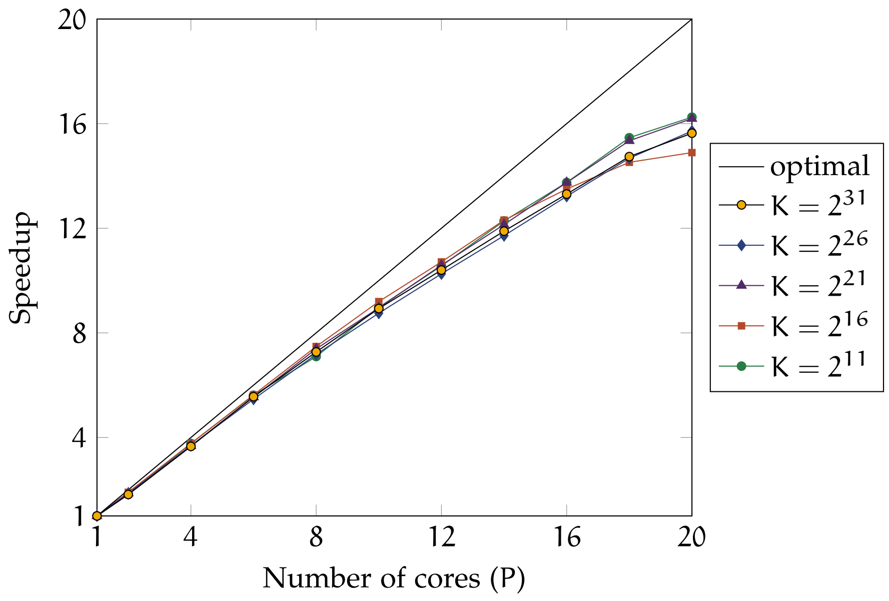


Figure 17: Speedup of ADAPTIVE compared to single core performance.

bined. The operating system is SLES 11.3 with Linux kernel 3.0.101 for x86_64. We use GCC 4.8.3 as compiler using `-O3 -march=native` optimizations.

Our data sets consist of $N = 2^{31}$ rows where all columns are 64-bit integers. If not mentioned otherwise we report run times as "Element Time" $= T \cdot P/N/C$, where T is the total run time, P the number of cores, and C the number of columns (grouping and aggregate columns combined). This metric represents the time each core spends to process one element and makes numbers of different configurations easily comparable—among themselves and to known machine constants such as the time of a cache miss.

All presented numbers are the median of 10 runs.

We previously presented the experiments of this chapter in [133] (with exception of Figure 16). They were reproduced by a SIGMOD Review Committee and were found to support the central results reported in the paper.[10]

5.8.2 *Scalability with the Number of Cores*

We first assess the parallelization mechanisms of Section 5.3.2 and the quality of our implementation in terms of scalability with the number of cores. Figure 17 shows the speedup of ADAPTIVE for different numbers of groups K compared to its respective performance on a single core. As the plot shows, the speedup is around 16 on our 20 CPU cores no matter K, which is as close to the optimal speedup as practical implementations usually get. Section 5.8.5 also shows the

10 See http://db-reproducibility.seas.harvard.edu/, under "SIGMOD 2015".

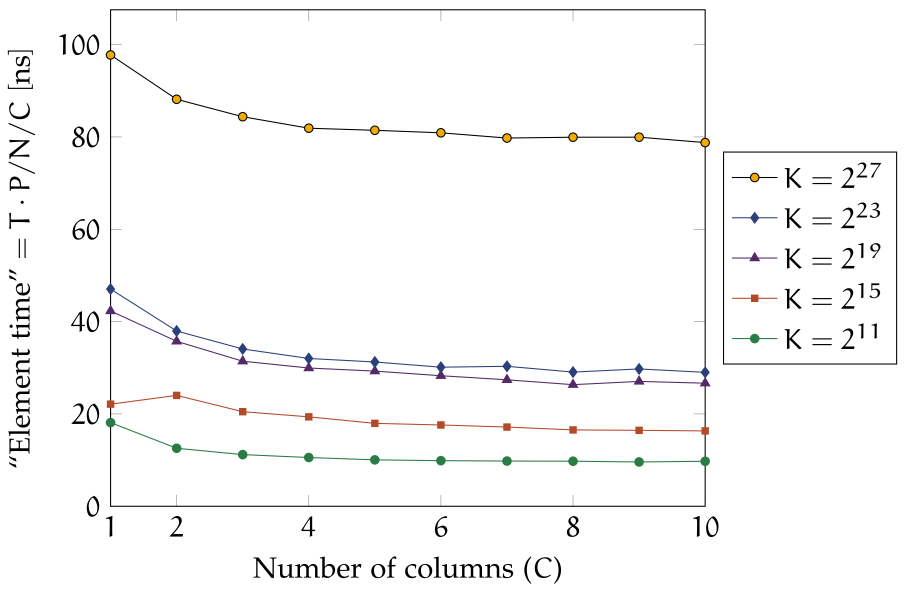


Figure 18: Scalability of ADAPTIVE with the number of columns using $P = 20$ threads.

experiments with other distributions, where we found our algorithm to perform just as well. Finally, we also ran this experiment with concurrent dummy threads on the idle cores in order to simulate a real system under load: If the dummy threads loop over their respective 3 MiB of cache to keep the cache warm, the performance of our algorithm is not influenced since its threads only rely on their respective part of the cache. However, if the dummy threads run an out-of-cache `memcpy`, the performance of our algorithm deteriorates by up to factor two, confirming that memory bandwidth is the main bottleneck our algorithm faces. The good scalability in all situations does not come to a surprise, since the threads of our algorithm do not share any resources and synchronize only at a very coarse granularity.

5.8.3 *Scalability with the Number of Columns*

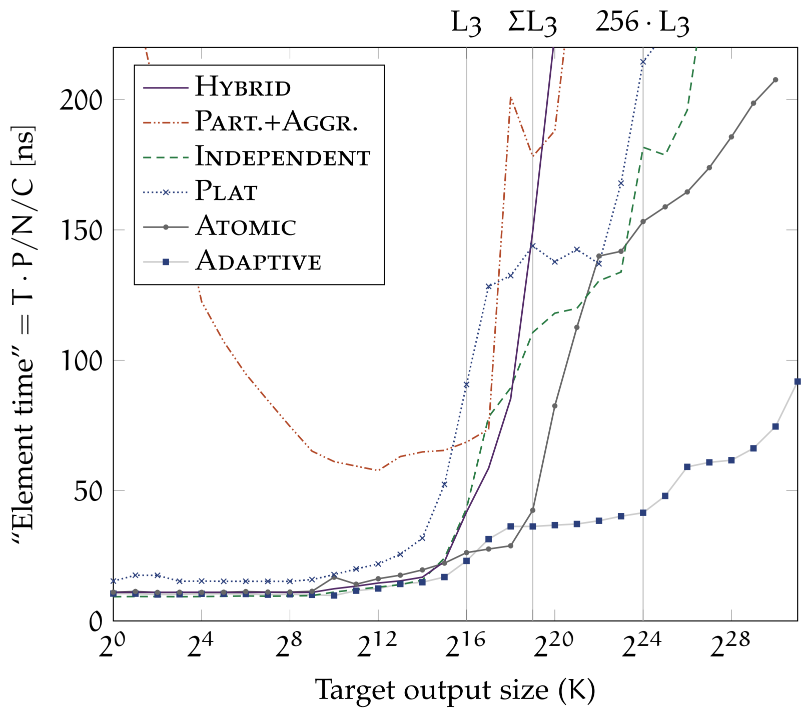
Figure 18 shows how the number of aggregate columns affects the performance of ADAPTIVE for different output cardinalities K. Just for this plot, we use $N = 2^{28}$ input elements to compensate the memory increase due to the additional columns. The experiment evaluates the effectiveness of the column-wise processing presented in Section 5.3.3, which is designed to process the different columns independently. Indeed, the plot indicates that the run time per element is almost the same for any number of columns. As discussed in Section 5.4, the processing of the grouping column is a bit more expensive than the processing of the other columns because of the hashing and collision resolution, which explains the slightly higher costs per element with lower number of aggregates.

We also confirmed the scalability of our operator with the number of columns in experiments with other data distributions, not presented here because they do not reveal further insights (we do show the impact of data distributions with a fixed number of columns below). Since the number of columns does not affect the processing cost per element, we run all other experiments only with a grouping column, i.e., without aggregate columns or $C = 1$.

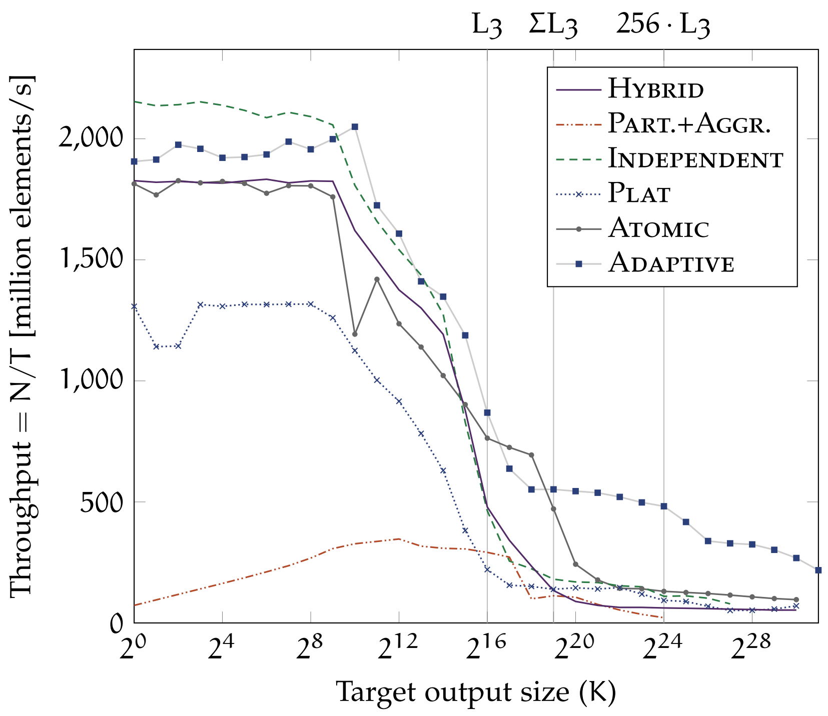
5.8.4 *Comparison with Prior Work*

We now show an experimental analysis of several state-of-the-art algorithms for in-memory AGGREGATION from the work of Cieslewicz and Ross [42] and Ye et al. [199] and compare them to ADAPTIVE. Since their work targets the row store architecture (implicitly assuming JiT query compilation) while our implementation targets the column-store architecture, we use a `DISTINCT` query with no aggregate columns ($C = 1$) for the comparison. In this type of query, the input and output data structures are equivalent in both architectures, and our algorithm does not need to produce a mapping vector for column-wise processing, so the experiment abstracts from all architectural differences.

We used the original implementations, but made the following modifications to tune them to this experiment: First, we changed the minimal output data structure size to the size of the L3 cache, which effectively eliminates collision resolution for small K and consequently reduces the run time in these cases by up to 25 %. Second, we removed padding and redundant fields in the intermediate and output data structures, in order to reduce tuple size and hence memory traffic. This reduces the run time by roughly 20 % for large K and even by up to 50 % where the reduction in size makes the output just fit into cache. The padding originally improved the collision resolution in the high-throughput cases of small K, but our first modification improves theses cases even more. Third, we replaced system mutexes by much smaller spin locks, again to reduce memory traffic, which also reduces the time by roughly 20 % for the variants using them. Finally, we replaced the multiplicative hashing by MurmurHash2, which we use in our algorithms as well. This has the same effect on the algorithms from prior work than on ours: a more predictable performance with up to 20 % run time reduction due to fewer collisions, but noticeable overhead for small K. Furthermore, we exceptionally deactivate the cardinality estimator of ADAPTIVE and instead provide it with the output size, which is an information that all algorithms from the shown competitors rely on. This makes our algorithms somewhat faster, but the benefit is only noticeable for large output cardinalities, i.e., if $K \approx N$, and always less than 10 %.



(a) Single core run time per element



(b) Total throughput

Figure 19: Comparison with prior work of Cieslewicz and Ross [42] and Ye et al. [199] using P = 20 threads.

Figure 19 shows a comparison of run times on data with uniform distribution. As we analyze in the following, all algorithms from prior work have an intrinsic limit in terms of K. They all consist of a fixed number of passes (either one or two) over the data, which means that they work well until a certain number of groups, but are penalized by a high number of cache misses beyond this limit. This is in line with our analysis of Chapter 4.

We describe the different algorithms and analyze their performance in more detail:

HYBRID [42] (1 pass): Each thread aggregates its part of the input into a private hash table with a size fixed to its part of the shared L3 cache. When this table is full, old entries are evicted similarly to an LRU cache and inserted into a global, shared hash table. This becomes inefficient as soon as most of the output does not fit into the private tables, which happens for $K > 2^{16}$ (marked with L3 in Figure 19).

ATOMIC [42] (1 pass): All threads work on a single, shared hash table protected by atomic instructions. This approach can suffer from contention due to concurrent updates as discussed by the original authors [42], but in the present `DISTINCT` query without aggregate columns, virtually no updates occur, so the problem is inexistent. It reaches its cache efficiency limit when the shared hash table exceeds the cache size, namely at around $K = 2^{19}$ (marked with ΣL3). This gives ATOMIC an advantage over all other algorithms for numbers of groups where it can fit its output into the *combined* cache while the shared-nothing approaches cannot.

INDEPENDENT [42] (2 passes): In a first pass, every thread produces a hash table of his part of the input. In a second pass, the hash tables are split and merged in parallel.[11] This makes the algorithm similar to HASHINGONLY with two passes, but since the hash tables of the first pass can be larger than the cache, both passes can trigger close to a cache miss per row. This limit is reached when the working set exceeds the L3 cache fraction *corresponding to each thread*, namely roughly at $K = 2^{16}$ for the first pass (marked with L3) and $K = 2^{24}$ for the second (marked with $256 \cdot$ L3).

PARTITION-AND-AGGREGATE [199] (2 passes): Similarly to PARTITION-ALWAYS with two passes, this algorithm first partitions the entire input by hash value and then merges each partition into its part of a hash table. Like our algorithm if limited to two passes as shown in Figure 11b, this algorithm cannot do the merging in a cache-efficient manner if $K > 256 \cdot \text{cache} = 2^{24}$ ($256 \cdot$ L3 in the plot). Furthermore, its partitioning uses the naive implementation as presented in Section 5.4 and is therefore slower than ours.

PLAT [199] (Partition with Local Aggregation Table, 2 passes): Similarly to HYBRID, in this algorithm each thread aggregates into a pri-

11 Note that the time of the second pass was not taken into account in the original paper [42].

vate, fixed-size hash table. When it is full, new entries are overflowed into hash partitions, which are merged in a subsequent pass like in the previous algorithm. This entails the same limit: the merging becomes inefficient if $K > 256 \cdot \text{cache} = 2^{24}$ (marked with $256 \cdot \text{L3}$ in the plot). The partitioning in itself is also less efficient than ours, but in contrary to the previous algorithm, our optimization could not be applied here, since the private hash tables would destroy the explicit L1 cache management of software write-combining.

ADAPTIVE (variable number of passes): Our algorithm is the only algorithm that gracefully degrades with larger K thanks to the efficient additional passes. Compared to the fastest of the other algorithms, it achieves a speedup of at least factor 2.7 for all $K \geqslant 2^{21}$.[12] The peak speedup factor 3.7 is achieved at $K = 2^{24}$ where ADAPTIVE needs only $41\,ns/element$ while ATOMIC needs $153\,ns/element$. Note that this speedup is higher than one usually hopes for for such a fundamental operator like AGGREGATION, where improvements of several tens of percent are already worth some effort.

It is also worth noting that the second best algorithm for large values of K is actually the simplest in terms of cache management: While the cache efficiency mechanisms of the other algorithms take extra time even though they do not work outside the range of K they were designed for, ATOMIC "just" pays a single cache miss per row.

As Figure 19b shows, ADAPTIVE is also at least as fast as almost all other algorithms for smaller values[13] of K. The similarity of all algorithms for small K does not come as a surprise, since all hash-based algorithms do effectively exactly the same in these scenarios. It is interesting to see however that the throughput starts dropping slightly later for ADAPTIVE than for the other algorithms. The reason is that the linear probing scheme our algorithms use can store more elements in the same amount of space than the chaining scheme used by the other algorithms, which need to store an additional pointer.[14] Only for $K = 2^{18}$, ATOMIC can fit the output just into its shared L3 cache, and is therefore slightly faster than ADAPTIVE, which already needs a partitioning pass. Since the partitioning is so fast though, the difference is only very small.

5.8.5 *Skew Resistance*

We now extend the experiments on uniform data to other input distributions in order to test the skew resistance of our ADAPTIVE operator. We use the synthetic data generators of Cieslewicz et al. [42],

12 As shown by our real-world workload study of Section 2.1.2 and unlike common belief, AGGREGATION with an output cardinality larger than two million rows is not at all uncommon.

13 We leave the narrow gap to INDEPENDENT as a small open problem.

14 This is also the reason why the cache sizes indicated in Figure 19 are off by factor two for ADAPTIVE.

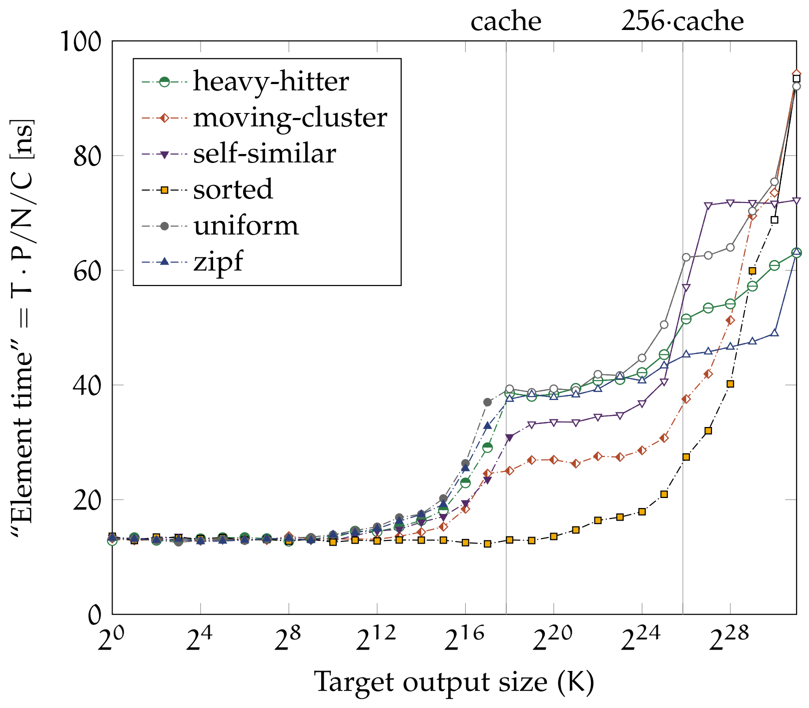


Figure 20: ADAPTIVE on different distributions using $P = 20$ threads.

which generate input data for any combination of N and K for a series of distributions with different characteristics. Since data cannot have $K = N$ groups *and* be skewed at the same time, K is only approximated. The distributions are namely heavy-hitter, moving-cluster, self-similar, sorted, uniform, and zipf.[15] In short, in heavy-hitter, 50 % of all records have the key 1, the others are distributed uniformly between 2 and K. In moving-cluster, the keys are chosen uniformly from a sliding window of size 1024. Self-similar is the Pareto distribution with an 80–20 proportion (also known as 80–20 rule) and zipf is the Zipfian distribution with exponent 0.5.

Figure 20 shows the performance of ADAPTIVE on all data sets. The first and most important observation is that ADAPTIVE is not slower on the other distribution than uniform. In this sense, uniform is the hardest distribution for our operator and skew only improves its performance. Since skew means that some keys occur more often than others, our operator can benefit from skew by using hashing for early aggregation of these values.

To show the mechanics of *how* our algorithm adapts to the skew, we plot those cases of each distribution with solid markers where our algorithm uses only hashing in the first phase, while we plot with hollow markers the cases where it switches to partitioning in the first phase at least once. This way we can see that on the sorted data set, ADAPTIVE switches to partitioning only where $K \geqslant N/2$. Before that

15 We omit the distribution sequence because of its similarity to uniform.

point every value is repeated several times, so every run of hashing reduces the data by factor $\alpha \geqslant \alpha_0$ or more, so our algorithm uses hashing for the next run as well. Since all hash tables are produced in cache and there are only very few of them except for the largest K, the second pass remains negligible and makes the run time only grow slowly with larger K. The behavior with moving-cluster is very similar, except that the hashing is more costly due to inferior locality. Since self-similar only exhibits relatively mild skew, the same effect is less pronounced on this data set: the reduction factor α is only high enough where there are just more groups than what would fit into cache. The run time is consequently very similar to that of uniform data, except that our algorithm switches to partitioning slightly later, namely for $K \geqslant 2^{19}$. Heavy-hitter even switches to partitioning at the same point as uniform, so for our algorithm this distribution does not have noticeable skew. In particular, it does not cause contention since there are no shared data structures in our algorithm. It rather seems like the non-hitter keys are the hard part of this distribution. The zipf data set is so little skewed that it is processed just like uniform data. The fact that it takes less time for the largest K is due to an artifact intrinsic to the generation of skewed data because there are actually less than K distinct values as discussed above.

As discussed and shown experimentally in the original papers, some of the algorithms of Cieslewicz and Ross [42] and Ye et al. [199] also adapt to skew, but to a lesser degree than ours. All of them are based on hashing and therefore profit from locality. However, only HYBRID can adapt to changes in locality as occurring in data sets like sorted or moving-cluster since it maintains a set of "hot" groups similarly to an LRU cache. Furthermore, HYBRID can be complemented with ATOMIC, which has the best performance of the algorithms known at the time for larger K, as shown by Cieslewicz and Ross [42]. By using sampling during execution, they can choose the best of the two algorithms, which is quite robust but has considerably higher constants than our ADAPTIVE. All other shown competitors have no mechanism to adapt to changing locality.

Even more importantly, the authors of the above algorithms do not give mechanisms to adapt to unknown K and rely on a prediction of the optimizer instead. This could be fixed by growing the data structures on demand, but would be highly non-trivial (if at all possible efficiently) for the shared data structure of ATOMIC and the resizing costs would considerably decrease performance for large K as shown in Section 5.6.2. In contrast, our recursive, run-based algorithm makes adapting to unknown K easier and the integration of a cardinality estimator lets us construct the hash table runs immediately in the correct size in order to transparently handle any K without overhead. Thanks to this feature our algorithm is the only one to fully meet our "adaptivity" challenge.

5.9 SUMMARY AND CONCLUSION

In this chapter we study the question of how adaptive query processing algorithms can estimate the size of the output data structure before allocating it. This is of great practical relevance for our own AGGREGATION operator and represents an aspect that has been ignored by prior work. As solution we propose to integrate an inexpensive cardinality estimator into an early phase of the processing such as partitioning. With experiments we show that our approach can cut processing costs by up to factor 2 by effectively eliminating the costs of resizing the data structure every time the current size is not sufficient anymore. Thanks to the simplicity of the estimator we use, the additional computational costs are limited. With this pragmatic building block, we ensure that our AGGREGATION operator is truly adaptive and achieves optimal performance without any type of external information.

In summary, our work starts with the assumption that even in the in-memory setting, the movement of data is the hard part of relational operators such as AGGREGATION. It builds on the insight of the previous chapter, where we establish a lower bound on the number of cache line transfers, which is matched by the traditional HASHAGGREGATION and SORTAGGREGATION with well-known optimizations. We design an algorithmic framework based on sorting by hash value that allows combining HASHING for early aggregation and state-of-the-art INTEGERSORTING routines depending on the locality of the data. We also show how to parallelize it within and across multi-core processors. Our framework thus addresses the cache efficiency and parallelization challenges we identified in Section 3.1. Furthermore, we tune both the HASHING and the SORTING routines to modern hardware and devise a simple, yet effective criterion of locality to switch between the two. As a consequence, our algorithm also meets the challenges of CPU friendliness and adaptivity. We show extensive experiments on different data sets and a comparison with several algorithms from prior work. Thanks to the combination of optimal high-level design guided by the theoretical analysis of the previous chapter and low-level tuning to modern hardware, we are able to outperform all our competitors by up to factor 3.7.

We expect work like ours to become of increasing importance in the near future, since memory bandwidth is developing at a lower rate than processing speed of multi-core CPUs. In particular, we invite more work on the duality of HASHING and SORTING. We believe that other cache-efficient sort algorithms can be augmented similarly to what we do with BUCKETSORT: Integrating early aggregation into an appropriate sort algorithm in an even tighter way could have the advantage of reducing the amount of data even if the reduction factor is lower than our α_0, but seems challenging to do without increasing computations too much.

6 MEMORY-CONSTRAINED AGGREGATION USING PIPELINING

Relational operators of the first generation of in-memory database systems were designed to increase execution speed through cache efficiency and CPU friendliness. Memory usage of intermediate results was often neglected, which does not affect performance of these operators in benchmarks but consumes one of the scarcest resources in systems of this architecture. This renders many approaches proposed in the past impractical. In this work we study the question of how to limit memory usage of the Aggregation operator without compromising performance. To that aim we propose a pipelined processing model *inside* the operator to overlap production and consumption of intermediate results. We show how clever scheduling of work between and within the different stages of the operator allows controlling the amount of auxiliary memory used by the operator and thus the trade-off between speed and memory consumption. Our resulting operator enables competitive processing performance until it is limited to a small fraction of the original memory usage. This design also makes our operator *malleable*, i.e., it can use or free resources during execution as provided or needed by the rest of the system.

6.1 MOTIVATION

In the previous chapter, we present an Aggregation operator that solves most of the challenges that we identify in Chapter 3: Through recursive algorithm design and low-level tuning, we make the operator cache-efficient and CPU-friendly, enable wait-free parallelization with balanced distribution of work, and ensure robustness against unexpected data characteristics. However, *in fine*, all these challenges aim at improving a single metric: speed of execution.

While execution speed is very important, it is not the only metric of interest. In industry-grade database systems, which should not only perform well in benchmarks but handle very diverse and potentially unusual situations, memory usage of operators is of almost equal interest. While affordable memory sizes increase at a rapid pace, it is still one of the scarcest resources of in-memory database systems. If it is handled without caution, queries may have to be aborted when the system runs out of memory, or new queries may not be admitted because not enough free memory is available for their execution.

As we discuss in Section 3.8, database systems traditionally limit memory consumption with a pipelined processing model. However, this approach does not apply to the auxiliary memory of operators, which may reach considerable orders of magnitude for pipeline breakers such as AGGREGATION. These operators often need several passes over the input, during which the amount of data is not reduced, even if the final output of the operator is much smaller than the input. Our AGGREGATION operator for example may produce runs whose total size is as large as the input: If the number of groups K is larger than the cache size M (i.e, if $K > M$) and two or more levels of recursion are needed, then the intermediate runs consists of roughly N rows for an input of size N. This is the case even if the result is quite small, say $K = 2M \ll N$.

To remedy this issue, we sketch a query processing system that tracks and possibly constrains the amount of memory used by different parts of each query and extend our AGGREGATION operator to this system. The query processing system that we envision organizes its main memory space in a hierarchical fashion: A certain amount of the total memory is dedicated to query processing and split among the currently running queries. Each query in turn divides its memory into the memory used by the different operators for their processing, the memory used by the intermediate results between operators and the final result, and the memory for metadata and bookkeeping of the query. The operators may organize their memory in another level of the hierarchy. All memory allocations need to be tracked and accounted for to one component in the hierarchy. We use this hierarchy to control memory usage in a tractable fashion: each level of the hierarchy breaks down its own constraint into constraints for its components.

In order to react to changes in resource utilization, memory constraints between levels may be renegotiated: As queries progress or terminate, they return unused memory budget to the system, which may allocate it to new queries or existing ones, which in turn may forward it to some of their operators. Operators may also request more memory if that permits increasing their efficiency, but must be able to deal with situations where their requests are denied. This allows us to impose a memory constraint on queries and operators in order to limit excessive use by single components, but is more flexible than static allocations of resources, which may lead to under-utilization in dynamic environments.

In order to use system resources efficiently, the optimizer may take estimates of memory usage into account and plan queries in such a way that expected memory pressure is never too high. Similarly, admission control may be extended to queue queries until they can comfortably be processed with the current memory budget. However, in this section, we focus on the mechanisms needed within operators, so studying the optimizer and admission control is out of the scope of our work.

We use the query processing system described above as context to study AGGREGATION in a memory-constrained environment. Concretely, we make the following contributions:

- We extend the AGGREGATION operator from the previous sections such that it strictly respects a given memory limit for its intermediate results. To that aim we pipeline the work of different levels of recursion, thus reducing the amount of intermediate results (and hence memory usage) present at any point inside the operator.
- We identify situations where dead-locks may occur and devise an intra-operator scheduling scheme that avoids them.
- Our intra-operator scheduler also ensures the best possible efficiency for the processing with a scheme similar to a buffer tree [10]. It exhibits a trade-off between the amount of memory used for intermediate results and processing speed, which can be chosen by the database system to react to changing workload during runtime.
- As a by-product, we also make our operator *malleable*, i.e., we make its degree of parallelism dynamic to enable changing the number of processing units depending on the system load.

The rest of this chapter is organized as follows: We first explain how work inside an operator can be pipelined in Section 6.2. Then we present how we schedule this work in order to maintain efficiency in Section 6.3. We discuss implementation details in Section 6.4 and evaluate the performance of our approach in Section 6.5. Finally, we make some concluding remarks in Section 6.6.

6.2 INTRA-OPERATOR PIPELINING

We start with describing the mechanisms of intra-operator pipelining. This introduces the basic concept of how to organize the work of our recursive algorithm, such that the recursion levels can be pipelined. We also show some pitfalls of this approach and the way we address them. This will lay the basis for the next section, where we discuss how to schedule the work in the pipeline such that best efficiency is achieved.

6.2.1 *Overview*

The main idea of intra-operator pipelining is based on the observation that each recursive call to the algorithm processes the data as if it were a stream: it reads each tuple from memory once, uses some constant amount memory to process it, and puts the results into one

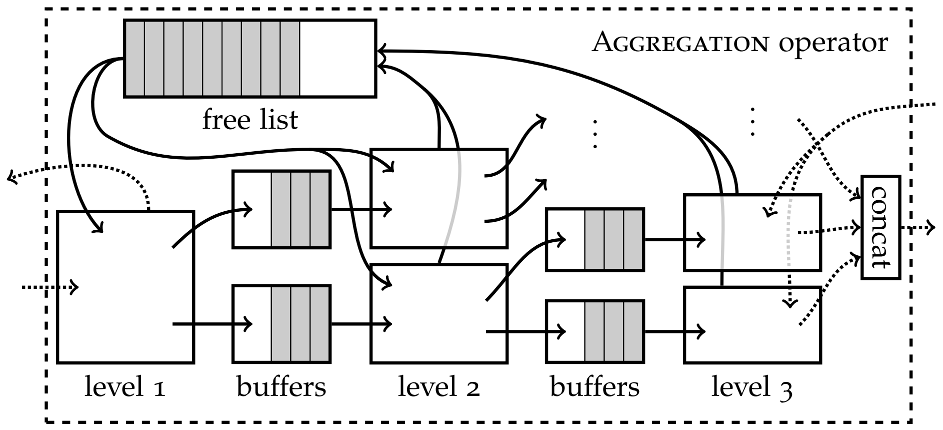


Figure 21: Intra-operator pipelining.

of the output partitions in memory, from where they are never read again. Consequently, the recursive call of a particular partition can start consuming the results of the current call as soon as they are produced. This is very similar to pipelined processing of relational operators, but happens *inside* an operator.

Figure 21 illustrates this idea. The recursive calls of the AGGREGATION operator form the stages of a branched pipeline organized in levels, which correspond to the levels of recursion of the original algorithm. Each stage consumes the blocks of tuples that the previous one produces. During processing, it holds one block per output partition, to which it writes the results. When the block of a partition becomes full, it is placed into a buffer of the corresponding next stage and replaced with a new one. When all tuples of a block are processed, the memory of the block is not returned to the operating system. Instead, the block is inserted into a free list, from where it is recycled.

With this model, the memory used by the operator is flowing in a closed cycle: it is used for the blocks of a running stage, the blocks in the buffers in between stages, and the blocks in the free list. By controlling the number of blocks in the cycle, we can control the memory consumed by the operator. Similarly, the memory used for input and output of the operator to communicate with the preceding and subsequent operators is flowing in respective cycles that can be controlled independently.

Intuitively, a stage needs two things in order to be *runnable*: there must be tuples in its buffer that it can process and there must be enough memory in the free list for its output. Runnable stages are subject to be run by a scheduler, to which a stage returns when the free list or the buffer become empty. We give a more formal definition of *runnable* stages in the following subsection and present the scheduling of the stages in the next section.

6.2.2 *Ensuring Efficient Progress*

The circular flow of blocks and the branched nature of the pipeline make it non-trivial to ensure efficient progress. In the following we establish the necessary mechanisms.

First, there have to be enough free blocks for the stage to start processing. Let m_i determine the minimum number of blocks for a stage of level i and l be the number of levels in the pipeline. Since the work of a stage of the last level consists of merging its input to the output of the operator, it does not need to take any more blocks, so $m_l = 0$. The work of a stage of the other levels consists of partitioning its input into a certain number of partitions, so these stages need a block per partition, hence $m_i = n_{\text{partitions}}$ for $i < l$, where $n_{\text{partitions}}$ is the number of partitions the PARTITIONING routine produces or its *fan-out*. If there are not at least m_i blocks in the free list, a stage of level i is not runnable.

However, this is not enough in a parallel setup: A stage may start running when there are m_i blocks in the free list and still not get all required blocks, since other stages may have taken some blocks in the meantime. In the worst case this may lead to a live lock. We solve this problem by taking all blocks required for a stage to start processing *atomically*. This ensures that a stage can either take all required blocks, m_i, or none at all. In Section 6.4 we give details about how to implement this scheme with regular allocators.

Second, it is important to be able to free memory at all times. Consider a point in time where all blocks in Figure 21 are in the buffers between stages of level 1 and 2. Consequently, the free list is empty. In particular, no stage in level 2 can get new blocks for its output, so no stage in level 2 can run. No other stage is runnable either because all other buffers are empty. Hence, no memory can be freed because there is no free memory—deadlock.[1]

To prevent this situation, we establish the following rule: A block can be taken from the free list only under the condition that the memory remaining in the free list is sufficient to process all tuples from the block such that it can be returned to the free list. In other words, a stage may only take a memory block if there are enough blocks for the stages directly following, which in turn may only run if their successors will also have enough memory, etc. So the number of blocks M_i required by subsequent stages of a stage of level i is defined recursively as follows: $M_l = 0$ since the stages of the last level do not have successors, and $M_i = m_{i+1} + M_{i+1}$. With this argument, a stage cannot start to run if there are not at least $m_i + M_i$ blocks in the free list, enough for the stage itself to start processing and for the subsequent stages to consume the results. If a running stage needs a new block, at

1 Note that this is similar to a deadlock problem observed with the EXCHANGE operator with a limited receive buffer by Graefe [76, Section 10.2].

least M_i blocks need to be in the free list—otherwise the stage cannot get the block and needs to stop running. This guarantees that there is always a runnable stage that eventually frees memory.

Third, it is important to ensure that the blocks are filled to a high degree. Otherwise, we not only waste memory, but more importantly also compromise the amortization effort of blockwise processing. With above definition of m_i and M_i, two effects can impede that. To understand the first effect, consider an input with an extremely high skew, where all but a tiny fraction of the tuples belong to the same "heavy-hitter" group. With this input, a situation may occur where a stage fills the block of the partition of the heavy-hitter group, while all the other blocks only contain one or very few tuples. If the stage cannot get new blocks at that point, it stops running. It puts its unfinished input block (if existing) back to its input buffer and the partially filled output blocks (if existing) into their respective output buffers. Subsequent stages consume the intermediate results and eventually, the stage continues its work where it left it. However, all but one of the blocks it produced are almost empty. In order to fill blocks to a high degree even in presence of skew, we refine the above definition of m_i to $m_i = 2n_{\text{partitions}}$ blocks. This is based on the observation that at most one block per partition can be partially empty, namely the last one. So if a stage produces $2n_{\text{partitions}}$ blocks—no matter how they are distributed over the partitions—on average the blocks must be at least half full, which bounds the worst case within a constant from the optimum.

The other effect that can cause blocks to be filled to a very low degree is due to the fact that the number of tuples per partition is divided on average by the fan-out of the PARTITIONING routine, $n_{\text{partitions}}$, in every level of the pipeline. Hence, if a certain amount of tuples should be processed on average by the stages of a certain level of the pipeline, the number of tuples processed by the stages of the preceding level has to be higher by this factor. We thus adapt the definition of m_i for a last time to $m_l = 0$, $m_{l-1} = 2n_{\text{partitions}}$, and $m_i = n_{\text{partitions}} \cdot m_{i+1}$ for $i < l - 1$, which is equivalent to $m_l = 0$ and $m_i = 2(n_{\text{partitions}})^{l-i}$ for $i < l$. We adapt the definitions of M_i accordingly, so as before, a stage is required to take m_i blocks (in an atomic fashion) when it starts running and needs to leave M_i blocks in the free list at all times.

Our scheme has a certain similarity with Arge's buffer tree [10]. In this tree conceived for the (single-processor) external memory model (cf. Chapter 4), inserted elements traverse the tree down to their final location through a sequence of buffers. The buffers are used such that flushing them to the next level is amortized by inserts from the previous level, which guarantees an optimal amortized insertion cost in terms of cache line transfers. As with our scheme, this means that a buffer of a certain level is flushed a factor less often than the previous level that corresponds to the fan-out of the tree. Sitchinava and Zeh [168] propose an extension of the buffer tree to the parallel external memory model. Their approach consists in parallelizing the routine that flushes the buffers, which only works efficiently with a fan-out of at least PB (on a machine with P processors and a block size of B tuples). This either represents a quite fine grained parallelization scheme (if we choose a small value for B, such as the cache line size) or forces us to have a huge fan-out (if we choose a larger value for B, such as the size of a memory page). A larger fan-out, in turn, would not work with the low-level tuning techniques of our Partitioning routine presented in Section 5.4, so we do not think that their approach can be applied to our problem directly. In contrast, we parallelize across nodes of the tree, such that most buffers are processed by a single thread, which requires significantly less synchronization.

6.2.3 *Columnwise Processing*

In above discussion about pipelining, we assume that every time a tuple is processed, it is processed in its entirety. However, as discussed in Section 5.3.3, this is not the case in column-store database systems, where rather *columns* are processed in their entirety: In the most extreme form of column-wise processing, the entire key column is processed first, producing a mapping vector, which is then used to process the aggregate columns one by one accordingly. In Section 5.3.3 we show how our framework adopts this scheme for runs by interleaving the processing of the key and aggregate columns. Here we extend the discussion about pipelining in a similar way in order to make it work for the column-wise processing model.

Let us look at what happens when the output block of a partition runs full during the processing of the key column. According to the pipelining scheme described above, the block should be put into the queue of the next stage corresponding to its partition. However, since the aggregate columns have not been processed yet, it is not a complete input of the next stage, so we cannot insert the block into the queue yet.

One solution would be to use the mapping vector in order to produce the aggregate columns just at the point when a block of keys is full. This way we complete the output block with the other columns and do not need to change the pipelining mechanism any further. However, this would mean that we constantly switch between the processing of different columns, which would lead to bad performance due to cache thrashing.

We thus extend all pipeline stages with internal buffers that allow amortizing the switching. Figure 22 illustrates this approach. When a stage starts running, it processes the key column and puts the result into its internal buffers. At some point in time, it decides to switch to

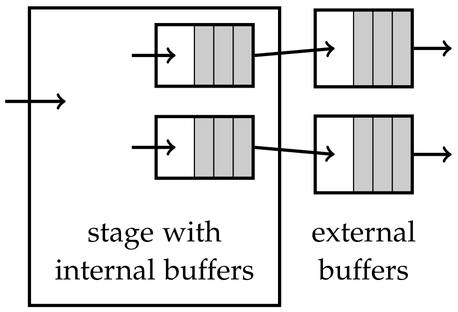


Figure 22: Extended pipelining stage with internal buffer.

processing the aggregate columns, thus completing the output blocks in its internal buffers with the remaining columns. Only when all columns have been processed, the complete blocks are finally removed from the internal buffers and inserted into the external buffers of the subsequent stages.

6.3 intra-operator scheduling

The previous section describes how we split the work of a recursive algorithm into blocks and how we route these blocks through a pipeline while guaranteeing efficient progress and handling multiple columns. This establishes the *mechanisms* of intra-operator pipelining. In this section we discuss two questions related to the *strategy* of intra-operator pipelining, namely how to schedule the work in the pipeline: which level of the pipeline to work on (thus potentially producing or consuming free memory), and which partition to work on inside a given level.

6.3.1 *Choosing a Partition Within a Pipeline Level*

The easier of the two questions is the one about which partition to work on in a given level of the pipeline. As we argued before, switching between partitions has warm-up costs, so we are interested to work on the partition we choose for a reasonably long time in order to amortize the switching costs. It is thus a good strategy to pick particularly full partitions. For the moment we ignore heavily skewed distributions and extend our solution later to support work stealing.

In order to find the partition with the most blocks in its queue quickly, we maintain a priority queue of the (non-empty) partitions of each level. Whenever a thread is looking for a job in a particular level, it pops the top-most partition from the priority queue. When blocks are inserted into the buffer of a partition by previous pipeline stages, the priority of the partition changes and the priority queue is updated accordingly. We calculate the priority of a partition as $\min(\lceil \log n_{\text{blocks}} \rceil, p_{\max})$, where n_{blocks} is the number of blocks in its queue and p_{max} the maximum priority value that we allow. There are several reflections behind this design: First, most of the times when a block is inserted into the buffer, the rounded logarithm of the length of the buffer does not change, so the priority queue does not need to be updated. This makes the common operation, the insertion of new blocks, cheap. Second, we do not need to know the exact length of the buffer—choosing "one of the fullest" buffers is good enough to amortize the costs for switching. In particular, the differences matter less the larger the buffers are (which is expressed by the logarithm) and eventually they do not make any difference at all anymore (expressed by the maximum value of p_{max}). We use $p_{\text{max}} = 10$, which worked well in our experiments. Finally, it allows us to use a bounded height priority queue, which just consists of a linked list per possible priority value. In this data structure, insertion is done in $\mathcal{O}(1)$ list accesses by adding the partition into the appropriate list. Popping the largest element is done in $\mathcal{O}(p_{\text{max}})$ list accesses in the worst case by iterating over all priority values in descending order until a priority with a non-empty list is found. Then, any element from this list is returned. To make this priority queue thread-safe, it is sufficient to use a thread-safe linked list. We use the lock-free linked list SkipListSet from `libcds`.[2]

With above design, only one thread can work on any given partition at each point in time: when it pops a partition from the priority queue, that partition cannot be found by any other thread. If the input is very skewed and the bulk of the work ends up in one partition, this can lead to a situation where only a single thread is working. We thus introduce work stealing by making the following changes: First, when a thread pops the largest partition from the priority queue, it inserts the same partition back into the queue immediately.[3] Furthermore, we calculate the priority of a partition as $\min\left(\left\lceil \log \frac{n_{\text{blocks}}}{P_{\text{partition}}+1} \right\rceil, p_{\text{max}}\right)$, where the new variable $P_{\text{partition}}$ is the number of threads currently working on the partition in question. Hence, the priority does not reflect just the amount of blocks in the buffer, but the amount of blocks in the buffer that each of the threads gets if the current thread chooses that buffer. Consequently, work stealing only happens on a partition that is so full that each of the threads gets a larger amount of work by sharing the partition than by taking any of the other partitions exclusively. In summary, our scheme schedules a number of threads to work on each partition that is proportional to its length and thus distributes the work equally among all threads even if the input is heavily skewed.

2 http://libcds.sourceforge.net/, v1.5.0.

3 It is often a good idea to require a minimum size of tasks that can be stolen. Our scheme can be extended with this feature by reinserting partitions into the priority queue only if they are larger than that minimum size.

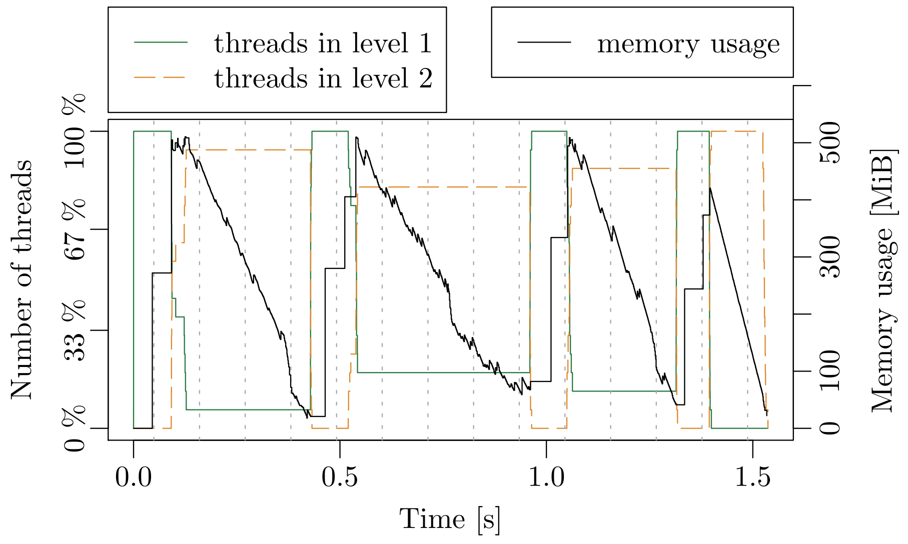
6.3.2 *Choosing a Pipeline Level*

Deciding in which level of the pipeline a given thread should work is a more complex question due to the cyclic dependencies of several dimensions discussed in Section 6.2.1: Each stage can consume blocks with intermediate results of the previous stage only if they are available; these intermediate results in turn can only be produced if free memory is available; and memory can only be freed by consuming blocks with intermediate results. At the same time it is important that when a thread starts working on a given partition, it continues to work there for as long as possible in order to amortize the overhead of switching partitions. We are thus interested in choosing a level that allows us to work on large partitions *taking the dependencies of the stages into account*.

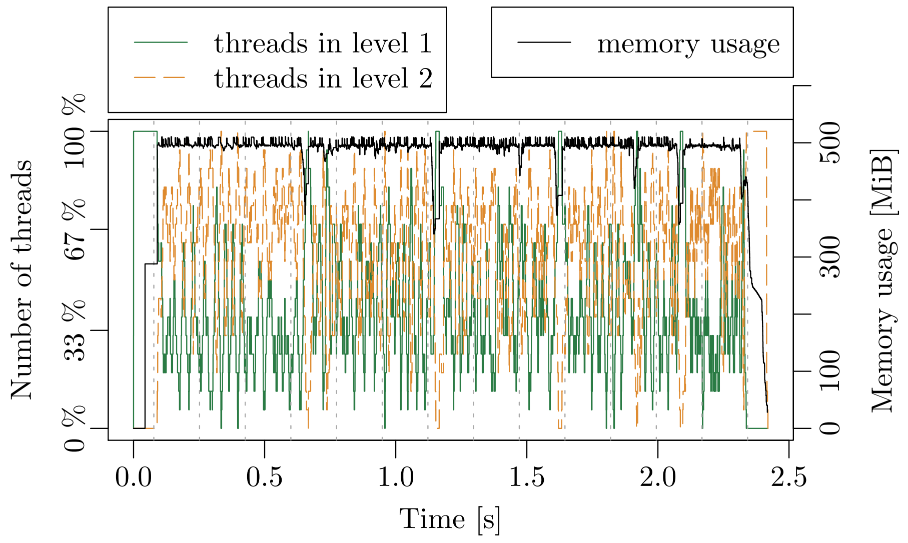
To give an intuition about the complexity of the question, we start with discussing a very simple strategy: whenever a thread finishes its current work, it chooses the lowest pipeline level where it can run. We thus call this strategy LowestFirst. To keep things simple, we assume that there is only one thread and two levels in the pipeline. The thread starts with processing the input (the stage of level 1), which it puts in blocks into the buffers of the next stages, as long as the number of blocks left in the free list is above the required minimum M_1. Since the thread cannot run in level 1 anymore, it will pick the next lowest level, level 2, pick one of the largest partitions it has just produced, and partition all of its blocks into blocks of the output. At this point it can go back to level 1, since a few blocks were freed by level 2, where it processes the input until the minimum M_1 of blocks in the free list is reached again. That will happen much faster this time since the only free blocks are those of a single partition freed just before. The thread then alternates between processing a single partition of level 2 and about the same amount of blocks of level 1. This means that only a small amount of memory is used in every step, namely as much as can be freed from a single partition—we unnecessarily restrict the memory that is actually used.

Figure 23a shows a sample run of the LowestFirst with multiple threads. The experiments are carried out with the test setup described in Section 6.5.1, but with $N = 2^{28}$ input records to increase readability of the plots.[4] As expected, all threads first work in the first level of the pipeline, which increases the memory consumption close to the limit. From then on, the threads oscillate between the two levels and can thus only make limited progress before they are rescheduled because memory usage is constantly close to the constraint.

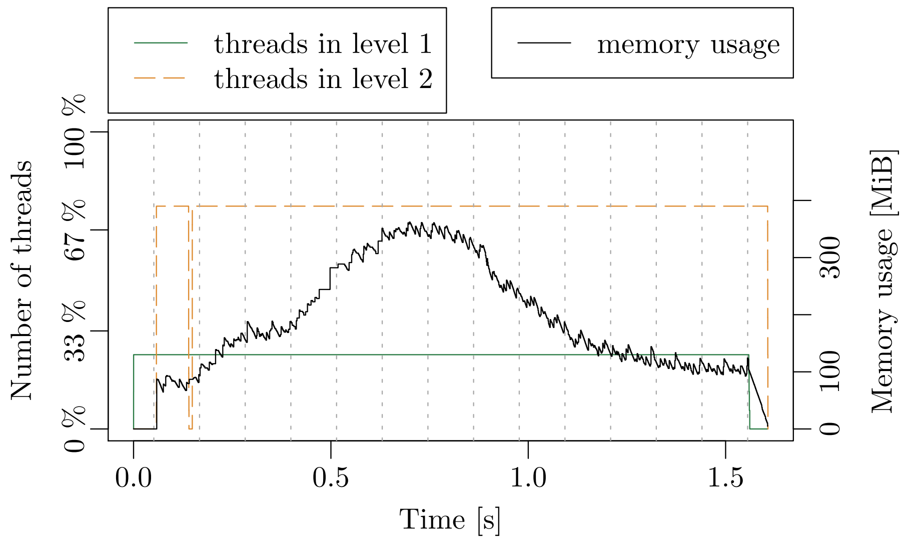
4 Note that the absolute run times in these plots are not particularly meaningful because the tracing facilities needed for the plots affect the performance of the algorithm negatively.



(a) LowestFirst



(b) LowestFirstInertia



(c) StaticRatio (25 %)

Figure 23: Memory consumption and thread activity over time using different naive scheduling strategies.

A better strategy is to alternate between production and consumption of intermediate blocks with a larger granularity by staying as long in each level as possible. With this strategy and again assuming a single-threaded case, the thread starts in level 1 and processes the input blocks until there are M_2 blocks in the free list. It then processes *all* available intermediate results in level 2, before finally starting over in level 1. At this point, all blocks have been freed, so all available memory can actually be used for the processing, so the thread can do the same work for a longer time before having to switch. We call this strategy LowestFirstInertia.

If we extend this strategy to multiple threads, we can see that this may still not be optimal: The threads first all work in level 1 until no more free blocks can be taken from the free list. All threads that need new memory from this point on are forced to switch to level 2 in order to free memory. Since it takes some time before memory can actually be released, in fact *most* threads change to level 2, so they all change levels more or less at the same time. They consume intermediate results, thus freeing memory blocks, until all buffers are empty and then all return to level 1, again more or less at the same time. Figure 23b shows a sample execution of the LowestFirstInertia strategy: The threads alternate between consumption and production of intermediate results as if they were synchronized such that the memory usage alternates between minimum and maximum. This means that, on average, only about half of the memory is used.

In order to improve the memory usage, we can attempt to produce and consume intermediate results at the same time: If each level gets assigned a fraction of the threads such that all levels process the data at the same rate, then the amount of memory used by each level remains more or less constant over time. If this equilibrium can be attained with a high memory usage, more blocks are in the partitions on average thus better amortizing the cost of partition switches. One heuristic with this goal is to statically assign a certain ratio of the threads to each of the levels. This strategy is called StaticRatio. This is helpful as reference, but has several practical shortcomings: First, the processing speed of the different levels depends at least on the implementation and the hardware, so we would have to find the desired fraction for every system anew. More importantly, it also depends on input distribution and output cardinality. If the output is smaller than one hash table, almost no work needs to be done in level 2, whereas for very large outputs, the number of tuples being processed level 2 is as high as the number of tuples in the input. Hence, whichever ratio with statically pick, it will not be optimal for all situations.

Figure 23c shows a sample execution of the StaticRatio strategy with a ratio of 25 % of the threads assigned to level 1 and the rest to level 2. It shows that after some start-up phase, production and consumption of intermediate results are more or less balanced, such that

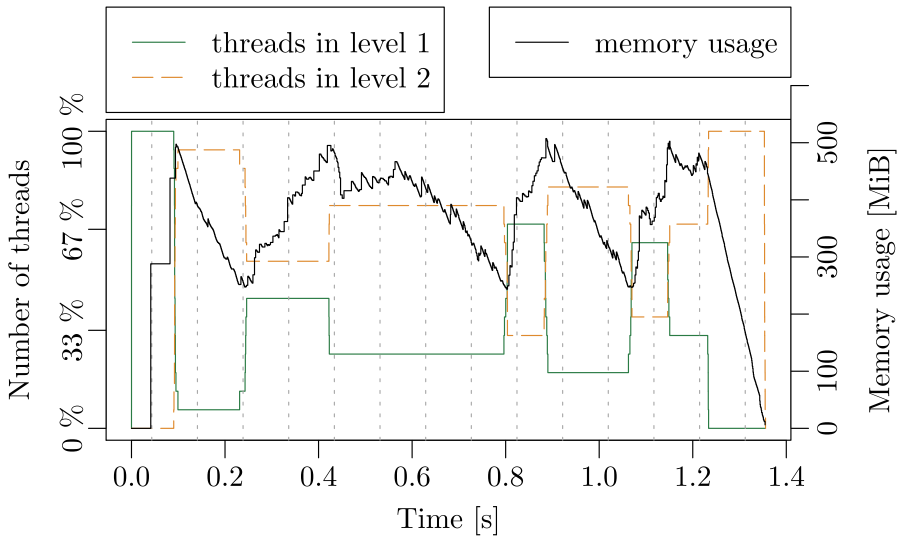


Figure 24: Memory consumption and thread activity over time using the MemoryTarget (50 %) scheduling strategy.

memory consumption remains stable. However, this equilibrium was found offline by trying out different ratios.[5] Furthermore, the ratio cannot be changed during execution. Consequently, before the first blocks of level 1 are completed, the threads of level 2 are completely idle. Similarly, the threads of level 1 stop working slightly earlier than those of level 2.

We thus propose a strategy that aims at dynamically maintaining a balance between production and consumption of intermediate results while using as much of the available memory as possible. The intention to increase the average amount of work the threads do before they are rescheduled. The strategy consists in choosing the pipeline level depending on how much of the available memory is currently used. If more than a ratio $m_{\text{target}} \in (0, 1)$ is used, we assume that the operator is about to run out memory, so threads are scheduled to level 2 in order to free memory. If less than a ratio of m_{target} is used, threads are scheduled to level 1 in order to produce more intermediate results. Whenever a thread does not find work in the level it was assigned to, it searches for work in the other level. We call this strategy MemoryTarget. Figure 24 shows a sample run of this strategy with $r_{\text{target}} = 50\,\%$. Similarly, as in the LowestFirstInertia strategy, some threads alternate between the two levels, but now more threads continue the work in their level and the available memory is never unnecessarily freed, so on average, the threads do more work in the same partition before they are rescheduled.

5 Assigning more threads to level 1 produces intermediate results too fast, such that eventually the memory constraint is reached, while assigning more threads to level 2 consumes the intermediate results even faster, such that the buffers cannot accumulate a lot of work before a thread empties them again.

6.3.3 *Malleable Execution*

The fact that we break down the work of the operator into small tasks has another benefit apart from making it possible to pipeline the processing of the work of different recursion levels: the degree of parallelism can now be varied during execution. This kind of parallelism is called *malleable* [120]. Our operator can trivially support it: Whenever a new thread starts working on a particular operator instance, it first selects a pipeline level as just described and pops a stage from the priority queue of that level—just like threads choose the next partition to work on when they have run out of memory or out of work. After a thread has done a sufficiently large amount of work, the operator yields it back to the inter-operator scheduler, which can decide that it should continue to work in the same operator or schedule it to do some other work. This way, the scheduler of the execution engine of the database system can react to dynamic workloads changes by increasing and decreasing the number of threads of the operator. As mentioned earlier, scheduling strategies between operators are out of the scope of our work and constitute a possible extension of this thesis.

6.4 IMPLEMENTATION DETAILS

In this section we briefly go over a few details that are necessary to make our implementation efficient and that are worth mentioning.

First, as mentioned earlier, we devise a scheme that allows us to guarantee that either all or none of a series of allocations succeed and that is independent of the underlying memory allocator. Doing several allocations "atomically" is necessary to ensure that the PARTITIONING routine can allocate at least a certain number of blocks for each of its output partitions once it starts processing. This mechanism should be orthogonal to the memory allocator, which is often a highly optimized software component [180]. Note that simply allocating a single chunk of memory is not sufficient because the different allocations are freed at different points in time.

Our solution consists in wrapping the allocator in a *memory pool* that decouples the *reservation* of memory from its *allocation*. The pool has a desired limit and maintains a counter for how much memory has been allocated or reserved. In order to make a series of allocations, other software components can make reservations of memory of a certain amount. The reservation is only granted if it does not exceed the limit of the pool, in which case the reserved amount of memory is deduced from the pool. A reservation is represented by a handle that has the interface of an allocator such that it can act as a proxy for the allocator of the pool, to which it has a reference. The reservation also tracks the total amount of memory that it has allocated. As long as there is reserved memory left, allocations are accounted for to the reservation handle and therefore guaranteed to succeed. Otherwise, they are accounted for to the pool, which may or may not be able to service the request. This mechanism guarantees that as much memory can be allocated as has been reserved and is transparent to the software components that use it. In the case of our PARTITIONING routine, we can thus first make a reservation that is large enough in order to ensure efficient progress according to the reasoning in Section 6.2.2 and then start the routine with the reservation handle as allocator. Similarly, our reservation scheme allows us to reserve the memory of several columns in an atomic fashion.

Second, we implement a lazy loading scheme for the data structures of the scheduler that belong to particular levels and stages of the pipeline. These data structures only do bookkeeping and the bulk of the work is done in the two main routines, so this is a rather small, though noticeable optimization. Since the length of the pipeline, which corresponds to the depth of the recursion, is not known at the beginning of the execution, we might construct the data structures of later levels and stages unnecessarily. Since there is one stage for every partition and the number of partitions is multiplied by the PARTITIONING fan-out with every additional pipeline level, this can represent noticeable effort (with our fan-out of 256, we have 65536 buckets in the third level). Furthermore, a third level of recursion may only be necessary for some of the partitions, namely due to high skew or sporadic work stealing as discussed in Chapter 5. Consequently, we construct each level and each stage in this level when they are first accessed, respectively. This reduces the overhead to a minimum as only those bookkeeping data structures are constructed that are actually used during execution.

Finally, we fit our HASHING routine into the pipelined execution model as follows. This routine produces the results for the subsequent partitions as a single piece of memory that does not consist of blocks. In order to place a part of a hash table into the buffer of the partition that corresponds to that part, we use a descriptor consisting of the first and the last index of that part of the hash table along with a shared pointer the hash table itself. This way, when a pipeline stage has produced a hash table, one such descriptor is put into each of the partitions of the next stages. The shared pointer ensures that the hash table is only released once all of its content has been consumed by the subsequent pipeline stages. Note that this means that consuming a hash table part does *not* release memory. However, this does not affect the guarantee of our scheme to be able to make progress at all times. Our mechanism is based on the invariant that all intermediate results that a certain stage produces can be consumed. In particular, this holds for all partitions after the current pipeline stage—whether they actually free memory or not. So eventually all descriptors will be processed and released, which finally also releases the memory of the hash table.

6.5 EXPERIMENTAL EVALUATION

In this section, we evaluate the impact of intra-operator pipelining and intra-operator scheduling under a constrained amount of memory. To that aim, we extended the implementation of the AGGREGATION operator of the previous chapter with the concepts presented in this chapter. Our implementation currently only works for two levels of recursion; the evaluation of our approach for three levels is left for future work.

6.5.1 *Experimental Setup*

We run the experiments on the same machine as in Chapter 5. It consists of two Intel Xeon E7-8870 CPUs[6] and 256 GiB of main memory. The processors run at 2.4 GHz and have 10 cores each. Each core has a private L1 cache of 64 KiB, a private L2 cache of 256 KiB, and can access a shared on-chip L3 cache of 30 MiB (equivalent to 3 MiB per core). The TLB of the CPUs have two levels, a first with 64 entries for data and 128 for instructions and a second with 512 entries for both combined. The operating system is SLES 11.3 with Linux kernel 3.0.101 for x86_64. We use GCC 4.8.3 as compiler using `-O3 -march=native` optimizations.

If not mentioned otherwise, our data sets consist of $N = 2^{31}$ tuples of 64-bit integers and we use all available $P = 20$ cores. As argued in Section 5.8, we run the experiments only on the grouping column and report running times in the metric "Element Time" $= T \cdot P/N/C$, where T is the total run time, P the number of cores, and C the number of columns (grouping and aggregate columns combined). All presented numbers are the median of 10 runs.

6.5.2 AGGREGATION *under Memory Constraint*

We first show the impact of a fixed memory constraint of 256 MiB on our algorithm using different intra-operator scheduling strategies for a varying number of groups K. Note that a constraint of this size represents a mere 1.6 % of the input size—which is the amount of memory used for intermediate results by a recursive algorithm without constraint. As a reference, we plot the performance of our ADAPTIVE algorithm without constraint, as well as the ATOMIC algorithm of Cieslewicz and Ross [42], which were the two fastest algorithms in the experiments in Section 6.5 of the previous chapter.

6 `http://ark.intel.com/products/53580`

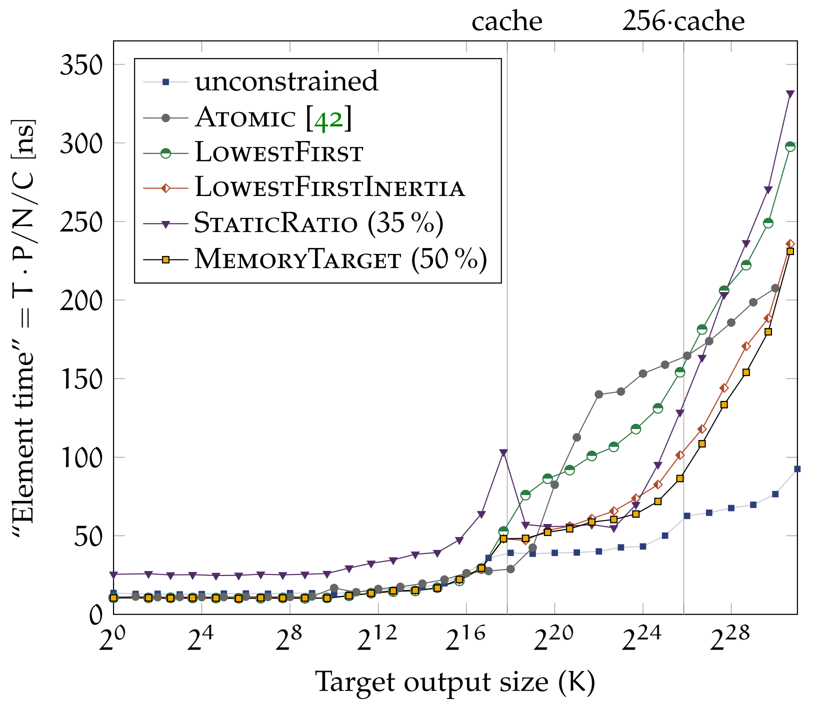


Figure 25: Comparison of intra-operator scheduling strategies under a memory constraint of 256 MiB.

Figure 25 shows the result. For $K \leqslant$ cache, almost all strategies have the same performance because they behave exactly the same way: every thread only uses the Hashing routine producing a single hash table that never runs full. After the initial allocation, the threads do not need to ask for new memory blocks, so they continue until the entire input has been consumed in order to do (negligible) work of the second level of the pipeline. Only StaticRatio behaves differently: in this strategy, the threads assigned to level 2 are idle during virtually all the execution. Hence, this strategy only uses a fraction of the available cores and is thus considerably slower.

The cases where $256 \cdot \text{cache} \geqslant K > \text{cache}$, where both pipeline levels have a roughly equal amount of work, are more interesting. As expected, the LowestFirst strategy performs worst, with run times between 1.9 and 2.7 times higher than the unconstrained algorithm. LowestFirstInertia and MemoryTarget are much faster, with LowestFirstInertia having an overhead of 20 % to 70 % compared to the unconstrained version and MemoryTarget having an overhead of 20 % to 47 %. This means that the fact that MemoryTarget takes the current level of memory usage into account gives it an advantage of about 15 % compared to the more simple LowestFirstInertia. Overall, both strategies make our approach of intra-operator pipelining viable.

The performance of StaticRatio varies relatively to the other strategies. This is expected since its thread ratio is hand-tuned to a particular relative processing speed of pipeline levels 1 and 2, which varies with K. However impractical this makes StaticRatio, it gives hope for finding even better scheduling strategies, but we leave this as an open problem for future work.

It is also interesting to compare our operator with memory constraint to the Atomic algorithm of Cieslewicz and Ross [42], which does not need intermediate memory at all. In the configuration shown in the plot, our algorithm with MemoryTarget scheduling strategy has still an advantage of factor 1.1 to 2.4 compared to Atomic for $256 \cdot \text{cache} \geqslant K > \text{cache}$. This is an important result, since it confirms that considerably higher performance than the simple Atomic algorithm can be achieved without excessive memory consumption.

For $K > 256 \cdot \text{cache}$, we reach the limitation of our current implementation, which only supports constrained memory with two pipeline levels. Like PartitionAlways with two passes, as shown in Figure 11b in the previous chapter, our algorithm suffers from the cache misses of the out-of-cache processing of the second pipeline level if the number of groups is that large. Despite this obvious inefficiency, our algorithm is still faster than the Atomic algorithm of Cieslewicz and Ross [42] for all but the largest K. We believe that our concepts can be extended to three pipeline levels and more, but leave the implementation and evaluation open for future work.

6.5.3 *Trade-Off between Performance and Memory Constraint*

We now show how the amount of memory available for intermediate results impacts the performance of our algorithm. Figure 26 shows the performance of the scheduling strategies evaluated before for a fixed number of groups of $K = 2^{23}$ and variable memory constraints between 32 MiB (equivalent to 0.2 % of the input size) and 4096 MiB (equivalent to 25 %). As a reference, we also plot the performance of our Adaptive operator without constraint and that of the Atomic algorithm of Cieslewicz and Ross [42].

As the figure shows, all strategies have a higher run time, the smaller the memory constraint is and approach the run time of the unconstrained version as the constraint tends towards the input size. This is expected, since a smaller constraint means that the threads need to switch between partitions more often, which induces more overhead. With constraints of 64 MiB and more, most strategies are faster than Adaptive, and with constraints of 256 MiB and more, the difference is larger than factor 2 (except the naive LowestFirst).

While the general trend is the same for all strategies, some are more affected by the constraints than others. As expected, LowestFirst is the slowest in almost all cases. The MemoryTarget strategy is again the fastest strategy in all situations except one, where the StaticRatio

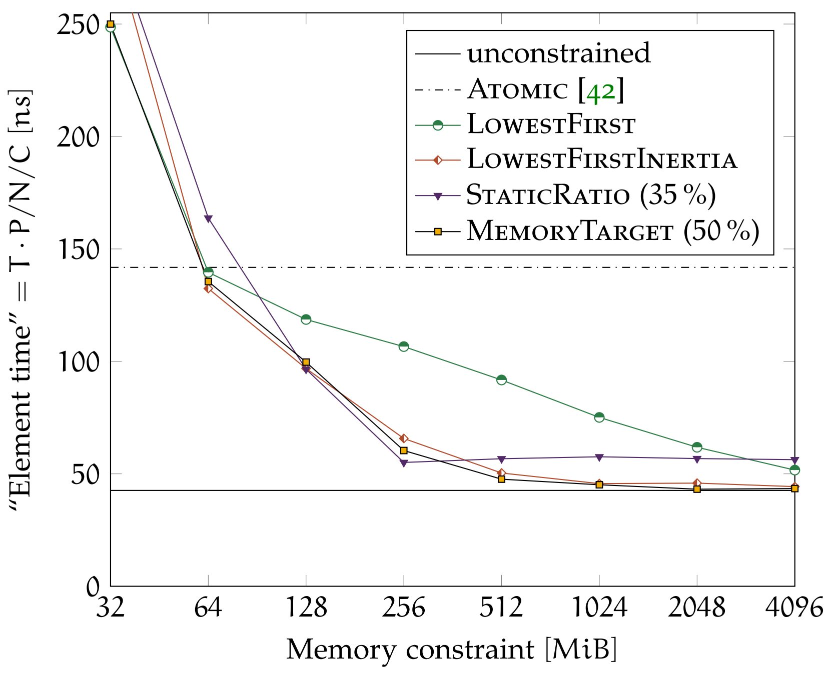


Figure 26: Impact of the amount of available memory on different intra-operator scheduling strategies for $K = 2^{23}$ groups.

strategy is slightly faster. In particular, MemoryTarget always has an edge over LowestFirstInertia, even if it is not always large.

6.6 SUMMARY AND CONCLUSION

In this chapter, we envision a query processing system that organizes and constrains its available memory in a dynamic hierarchy of memory budgets. We transform our recursive Aggregation algorithm of the previous chapter into a branched pipeline in order to work with a given constraint of such a system. To that aim, we devise an intra-operator pipelining scheme, compatible with columnwise processing, that avoids possible dead-locks and inefficiencies with a technique similar to buffer trees [10]. Furthermore, we define several intra-operator scheduling strategies that schedule the work of different stages of the pipeline. We show that a strategy that takes the current memory usage into account makes the best use of the available memory by balancing production and consumption of intermediate results. A series of experiments confirms the viability of our approach: For medium-sized outputs, setting a memory constraint of just 1.6 % of the original memory usage of our algorithm only incurs an overhead of 20 % to 47 %. This is still more than factor two faster than the best algorithm of prior work, which does not need any additional memory, and can be further improved by investing a higher memory budget. We thus solve the final "memory constraint" challenge of Section 3.1 without significantly compromising the performance of our operator with respect to the other challenges.

While we see significant advantages in our approach compared to previously proposed parallel buffer trees [168], in particular for practical implementations, we also think that more of the design space should be explored. Since in a buffer tree, the scheduling of the work is driven by the operations on the tree, its algorithm is not only simpler than ours, but also gives provable efficiency guarantees. In contrast, a formal efficiency proof seems difficult to achieve for our scheduling of pipeline stages. It is thus natural to ask for a simple, yet practical and highly tuned implementation of parallel buffer trees as a follow-up question of our work.

7 DISCUSSION

In this thesis we study the problem of engineering AGGREGATION operators for relational in-memory database systems. We set forth a list of eight challenges. Although most of them have been studied in the past in isolation, our work makes advances with each of the challenges and shows how to address all of them *at the same time*.

We start with a theoretical study of AGGREGATION in several external memory models, which reveals that AGGREGATION has the same cache complexity as MULTISETSORTING in many realistic situations. This serves as a guideline for our algorithm design. Consequently, we build an AGGREGATION algorithm with the recursive structure of a SORTING algorithm that inherits many good properties of HASHING. It thus combines two traditionally opposite approaches in order to achieve the cache efficiency of the better of the two. A simple, cheap mechanism lets our algorithm adapt to the data by switching between HASHING and SORTING routines during execution. We tune our implementation to low-level details of the hardware in order to utilize its performance to the fullest and show how to parallelize our operator such that it perfectly scales within and across processors. Furthermore, our operator is designed to work with column-wise processing and integrates well with Just-in-Time compiled query execution. We also extend the adaptive mechanism with an online cardinality estimator that makes our operator completely independent from potentially wrong statistics from the optimizer. Finally, we develop a technique to pipeline execution within our operator such that the amount of memory for intermediate results and thus the overall memory consumption is reduced.

We argue that—apart from our solution for each isolated challenge—our contribution also lies in their combination into one operator. A solution that fails to address just one of the challenges may not be of any value for productive use in a real database system. In our experience, the argument that the challenges are orthogonal often falls short. Obvious examples of a connection between challenges include the fact that skew is typically only a problem for parallel execution or that our sophisticated solution for processing under a memory constraint is only necessary for algorithms with large intermediate results. Similarly, as argued throughout the thesis, the reason why an adaptive solution is needed is the fact that different strategies are built to achieve cache efficiency in different situations. Of course, our intra-operator scheduling also reflects the high degree of parallelism. Maybe less easy to spot is the connection between CPU friendliness on the one hand and adaptivity and system integration respectively on the other. Too fine-grained or complex adaptive decisions may be too computationally expensive or may not work in a column-wise processing model. Our solution satisfies all three criteria by combining two tight loops on columns at a coarse granularity. The task of engineering AGGREGATION operators thus consists in combining several partial, well-chosen solutions such that together they form a greater one.

While, from an algorithmic perspective, our solution for AGGREGATION is the most complete to date, it leaves some questions related to software engineering unanswered. Our choice to implement all techniques inside the operator leads to a complex, monolithic implementation. Furthermore, its functionality partially overlaps with that of other components of a typical database system, such as memory management, user-level scheduling, and pipelining. However, the fact that we tailor each component to our particular operator allows us to show the performance that can be achieved if other aspects are subordinate. It thus has a definite value as reference point for future implementations. Furthermore, we believe that it is possible to find abstractions that allow modularizing our operator while preserving its performance. This may even make some building blocks from our memory management and user-level scheduling available for other operators. How and to what degree this is possible is an interesting question for future work.

Another direction is to reduce complexity by simplifying certain algorithmic aspects. One possibility is to incorporate very recent advances in hardware-conscious PARTITIONING routines. Schuhknecht et al. [162] presented a technique that allows partitioning with a much larger fan-out than what we use with only little more cost per element. If a large fan-out is required, namely if the number of groups is very large, then this requires fewer levels of recursion and thus reduces the overall execution time. Furthermore, it has the advantage that two levels of recursion are probably enough in all situations the system is designed for.[1] Under this assumption, hard-coding two passes might be simpler than a recursive algorithm, in particular in combination with intra-operator pipelining. Since the slightly higher partitioning costs of the larger fan-out are not needed for a small number of groups, one might think about finding a mechanism that adapts the fan-out of the PARTITIONING routine such that the smallest sufficient value is used.

In the context of Just-in-Time compiled query processing, one may also argue that cache efficiency can be achieved in a simpler way than by our recursive algorithm. With Just-in-Time compiled query plans

1 The technique of Schuhknecht et al. [162] works well with up to 2^{14} partitions. Since our HASHING routine works well with up to 2^{17} records of 32 bit, we could process up to 2^{31} groups efficiently. While larger scenarios are conceivable, one might rather want to process them with several sockets, so at least each socket would only have two levels.

it is possible to have tight loops *and* row-wise processing. With row-wise processing in turn, cache efficiency is easier to achieve: First, an algorithm that produces one cache miss per record will also produce exactly one cache miss *for the entire row*. With column-wise processing, however, the same algorithm is executed for every column, so it will produce a cache miss per row *and column*. In other words, row-wise processing can amortize cache misses better than column-wise processing. Furthermore, techniques like prefetching and simultaneous multi-threading may be used to hide latencies. Second, if rows are large enough to fill several cache lines, accessing data out of the cache does not load unnecessary data. In the external memory model, a cache line then corresponds to a block of size $B = 1$, which renders analyses in this model meaningless.[2] Making one out-of-cache access per input record may then be the best strategy. Combining all this techniques may lead to a "cache-efficient" algorithm that is much simpler than ours. Whether this is a viable solution, however, depends on many other things: Even with Just-in-Time compilation, row-wise processing makes vectorization hard to achieve; it only works in systems that support results in row format; and all the reasoning of this paragraph does not apply to rows that are so short that they are merely a "wide column". Just-in-Time compilation thus opens a huge design space that is interesting to explore in the context of cache-efficient AGGREGATION algorithms.

Finally, we want to mention another possible extension of this thesis. As we have argued before elsewhere [159], a communication volume that is sublinear in the input size is required to scale algorithms to large clusters and can be achieved for a surprisingly large number of problems. For AGGREGATION, TWOPHASEAGGREGATION achieves this property as long as the number of groups is small. In prior work [83] we have sketched an algorithm with sublinear communication volume for the other extreme: if most groups consist of a single record, it may be the best strategy to eliminate these unique records with a communication-efficient Bloom filter in a pre-processing phase and to run AGGREGATION only on the remaining ones. However, more work is needed with this algorithm so that it solves the other challenges as well.

2 However, even if cache misses are not a problem, TLB misses may be. One could then use the external memory model such that a block represents a *memory page* instead of a cache line and use the model to find TLB-efficient algorithms.

Part IV

APPENDIX

Something attached to something else; an attachment or accompaniment.

— Wiktionary [191]

A PROOFS

A.1 FROM MULTIPLICITIES TO THE NUMBER OF GROUPS

A multiset of size N can be characterized by the size of the corresponding set or by the multiplicities of the different elements. In this thesis, we denote the size of the set by K and the multiplicities by N_i, $i \in \{1, \dots, K\}$. Since doing analyses in terms of the multiplicities is more precise than doing them just in terms of K, we use the former characteristics whenever possible. However, if a simpler analysis is needed, most often it is possible to obtain a result in terms of K from the more precise result.

For example, the worst-case lower bound for the number of comparisons of MultisetSorting, i.e., sorting of a multiset, is known to be [136]:

$$N \log N - \sum_{i=1}^{K} N_i \log N_i + \mathcal{O}(N) \tag{31}$$

The worst-case complexity of MultisetSorting in terms of K is the worst case of a distribution of multiplicities that maximizes Equation 31. This can be done by using the *log sum inequality* [48]. The log sum inequality says that

$$\sum_{i=1}^{K} a_i \log \frac{a_i}{b_i} \geqslant A \log \frac{A}{B} \tag{32}$$

where $A = \sum_{i=1}^{K} a_i$ and $B = \sum_{i=1}^{K} b_i$ and that equality holds iff all $\frac{a_i}{b_i}$ are equal. Instantiated for our case (i.e., with $a_i = N_i$ and $b_i = K$), we get $A = N$, $B = K$, and

$$\sum_{i=1}^{K} N_i \log N_i \geqslant N \log \frac{N}{K}. \tag{33}$$

Hence, we can maximize the comparison lower bound of MultisetSorting from Equation 31 like this:

$$\begin{aligned} & N \log N - \sum_{i=1}^{K} N_i \log N_i + \mathcal{O}(N) \\ &\leqslant N \log N - N \log \frac{N}{K} + \mathcal{O}(N) \\ &= N \log N - N \cdot (\log N - \log K) + \mathcal{O}(N) \\ &= N \log K + \mathcal{O}(N) \end{aligned} \tag{34}$$

and equality holds iff all N_i are equal. Thus, the lower bound of MULTISETSORTING from Equation 31 has a simplified form using just N and K:

$$N \log K + \mathcal{O}(N). \tag{35}$$

A.2 ALGEBRAIC TRANSFORMATIONS OF SECTION 4.3.2

A.2.1 *Equation 14*

Here we show the algebraic transformations needed to transform Equation 13 into Equation 14 in the proof of the lower bounds in Section 4.3.2. In the transformations, we use Stirling's approximation [190] of the factorial:

$$\sqrt{2\pi}\, N^{N+1/2} e^{-N} \leqslant N! \leqslant e\, N^{N+1/2} e^{-N}, \tag{36}$$

as well as the following inequality to bound binomial expressions (cf. Greiner [79]):

$$\left(\frac{x}{y}\right)^{y} \leqslant \binom{x}{y} \leqslant \left(\frac{ex}{y}\right)^{y}. \tag{37}$$

The right side of Equation 13 under the logarithm is transformed as follows by using Stirling's approximation:

$$\begin{aligned}
&\ln(N!) - \frac{N}{B}\ln(B!) - \ln \mathcal{P} \\
\geqslant& \frac{1}{2}\ln 2\pi + \left(N + \frac{1}{2}\right)\ln N - N - \frac{N}{B}\left(1 + \left(B + \frac{1}{2}\right)\ln B - B\right) - \ln \mathcal{P} \\
=& N \ln \frac{N}{B} - \frac{N}{B}\left(1 + \frac{1}{2}\ln B\right) + \frac{1}{2}\ln B + \frac{1}{2}\ln 2\pi - \ln \mathcal{P} \\
\geqslant& N \ln \frac{N}{B} - \frac{N}{B}\left(1 + \frac{1}{2}\ln B\right) - \ln \mathcal{P} \\
\geqslant& N \ln \frac{N}{B} - N - \ln \mathcal{P} = N \ln \frac{N}{eB} - \ln \mathcal{P}
\end{aligned} \tag{38}$$

Similarly, the left side under the logarithm can be transformed as follows, using Equation 37 to bound the binomial expression:

$$\begin{aligned}
&P\ell\left(\ln 3 + 2\ln N + B\ln 2\right) + \ell \ln \binom{\min(MP, N)}{PB} \\
\leqslant& P\ell\left(\ln 3 + 2\ln N + B\ln 2\right) + \ell PB \ln \frac{e\min(MP, N)}{PB} \\
=& P\ell\left(\ln 3 + 2\ln N + B\ln\left(2 + e\min\left(\frac{M}{B}, \frac{N}{PB}\right)\right)\right)
\end{aligned} \tag{39}$$

Putting 38 and 39 together, we obtain Equation 14:

$$\begin{aligned}\ell \geqslant & \frac{N \ln \frac{N}{eB} - \ln \mathcal{P}}{P\left(\ln 3 + 2\ln N + B \ln\left(2 + e \min\left(\frac{M}{B}, \frac{N}{PB}\right)\right)\right)} \\ = & \frac{N \ln \frac{N}{eB} - \ln \mathcal{P}}{\Theta(P(\ln N + B \ln d))},\end{aligned}$$

with $d = \max\left(2, \min\left(\frac{M}{B}, \frac{N}{PB}\right)\right)$.

B PROBABILISTIC COUNTING ALGORITHMS

The problem of estimating the number of distinct elements in a stream has been studied by a variety of authors. The first considerable study was the seminal paper of Flajolet and Martin [66], which introduced ProbabilisticCounting (PC),[1] as well as an extension with *stochastic averaging* (PCSA). The basic approach tracks the numbers of leading zeros in the hash values of records and the extension increases accuracy by using memory space for several such estimators. The underlying intuition is that the more distinct elements there are, the more likely it is that one of them has many leading zeros. In subsequent work, ProbabilisticCounting was refined in order to improve its accuracy and to reduce its memory consumption [59, 65]. Alternative approaches were also proposed. We refer to comparisons by Aouiche and Lemire [8] and Metwally et al. [130] for an overview.

Algorithm 6 ProbabilisticCounting

```
1: func ProbabilisticCounting(S: Seq. of Row)
2:     bitmap ← {0, . . . , 0}
3:     for each row in S do
4:         $n_z$ ← Clz(Hash(row.key))        ▷ Clz: count leading zeros
5:         bitmap[$n_z$] ← 1
6:     R ← Clz(~bitmap)                     ▷ ~x: bitwise complement of x
7:     return $\frac{2^R}{\phi}$
```

The basic ProbabilisticCounting algorithm is shown as pseudo-code in Algorithm 6 and works as follows: For each record that we process, we count the number n_z of leading zeros in the binary representation ot the hash value of the record. We remember each value of n_z that we observe by setting to 1 the bit of a bitmap at position n_z. After all records have been processed, we count the number of leading 1-bits in the bitmap. Let the result be R. This means that at least one of the records has a hash value that starts with R zeros. Flajolet and Martin show that for K distinct values and perfectly uniform hash values, we expect to observe $\mathbb{E}(R) = \log_2 \phi K$ bits set, with $\phi = 0.77351\ldots$ [66], so K can be estimated by $\frac{2^R}{\phi}$.

Because the standard error of ProbabilisticCounting is quite high, Flajolet and Martin propose to use *stochastic averaging*. Algorithm 7 shows the pseudo-code of this approach. It consists in using m bitmaps to count observed numbers of leading zeros instead of just one, and to

1 Sometimes this technique is also called FMSynopsis, named after the two authors that proposed it.

Algorithm 7 ProbabilisticCounting with stochastic averaging

```
1: func PCSA(S: Seq. of Row, $m \in \mathbb{N}$)
2:     for $i \leftarrow 0$ to $m - 1$ do
3:         bitmap[i] $\leftarrow \{0, \dots, 0\}$
4:     for each row in S do
5:         $h \leftarrow$ Hash(row.key)
6:         $n_z \leftarrow$ Clz(h div m)
7:         bitmap[h mod m][$n_z$] $\leftarrow 1$
8:     $R \leftarrow \frac{1}{m} \sum_{m-1}^{i=0}$ Clz(~bitmap[i])
9:     return $\frac{2^R}{\phi}$
```

use some otherwise unused bits of the hash function to select which of the bitmaps to use for each record. This imitates having m simultaneous estimators, but still only needs a single hash value (instead of m of them). By using the average of the m values for R, we make the result more precise. For an analysis of the mathematical properties of these estimators, we refer to the original paper [66].

BIBLIOGRAPHY

[1] Daniel J. Abadi. "The Design and Implementation of Modern Column-Oriented Database Systems." In: *Foundations and Trends in Databases* 5.3 (2012), pp. 197–280.

[2] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel R. Madden. "Materialization Strategies in a Column-Oriented DBMS." In: *ICDE*. 2007.

[3] Alok Aggarwal and Jeffrey Scott Vitter. "The Input/Output Complexity of Sorting and Related Problems." In: *Commun. ACM* 31.9 (1988), pp. 1116–1127.

[4] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. "DBMSs on a Modern Processor: Where Does Time Go?" In: *PVLDB*. 1999.

[5] Martina-Cezara Albutiu. "Scalable Analytical Query Processing." PhD thesis. Technische Universität München, 2013.

[6] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. "Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems." In: *PVLDB*. 2012.

[7] Rasmus Resen Amossen and Rasmus Pagh. "Faster Join-Projects and Sparse Matrix Multiplications." In: *ICDT*. 2009.

[8] Kamel Aouiche and Daniel Lemire. "A Comparison of Five Probabilistic View-Size Estimation Techniques in OLAP." In: *DOLAP*. 2007.

[9] Lars Arge. "External Memory Data Structures." In: *Handbook of Massive Datasets*. Springer, 2002.

[10] Lars Arge. *The Buffer Tree: A New Technique for Optimal I/O-Algorithms*. Tech. rep. BRICS, RS-96-28. University of Aarhus, 1996.

[11] Lars Arge, Gerth Stølting Brodal, and Rolf Fagerberg. "Cache-Oblivious Data Structures." In: *Handbook of Data Structures and Applications*. Chapman & Hall/CRC, 2005, pp. 38-1–38-28.

[12] Lars Arge, Michael T. Goodrich, Michael Nelson, and Nodari Sitchinava. "Fundamental parallel algorithms for private-cache chip multiprocessors." In: *SPAA*. 2008.

[13] Lars Arge, Mikael Knudsen, Kirsten Larsen, Frank Dehne, Jörg-Rüdiger Sack, Nicola Santoro, and Sue Whitesides. "A General Lower Bound on the I/O-Complexity of Comparison-based Algorithms." In: *WADS*. 1993.

[14] Lars Arge and Peter Bro Miltersen. “On showing lower bounds for external-memory computational geometry problems.” In: *External Memory Algorithms*. 1999.

[15] Albert Atserias, Martin Grohe, and Dániel Marx. “Size Bounds and Query Plans for Relational Joins.” In: *FOCS*. 2008.

[16] Ron Avnur and Joseph M. Hellerstein. “Eddies: Continously Adaptive Query Processing.” In: *SIGMOD*. 2000.

[17] Shivnath Babu. “Adaptive query processing in the looking glass.” In: *CIDR* (2005), pp. 238–249.

[18] Shivnath Babu, Pedro Bizarro, and David Dewitt. “Proactive Re-Optimization.” In: *SIGMOD*. 2005.

[19] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. “Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited.” In: *PVLDB*. 2013.

[20] Cagri Balkesen, Jens Teubner, and Gustavo Alonso. “Main-Memory Hash Joins on Multi-Core CPUs : Tuning to the Underlying Hardware.” In: *ICDE*. 2013.

[21] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Ozsu. “Main-Memory Hash Joins on Modern Processor Architectures.” In: *TKDE* 27.7 (2015), pp. 1754–1766.

[22] Ronald Barber, Peter Bendel, Marco Czech, Oliver Draese, Frederick Ho, Namik Hrle, Stratos Idreos, Min-Soo Kim 0002, Oliver Koeth, Jae-Gil Lee, Tianchao Tim Li, Guy M. Lohman, Konstantinos Morfonios, René Müller, Keshava Murthy, Ippokratis Pandis, Lin Qiao, Vijayshankar Raman, Richard Sidle, Knut Stolze, and Sandor Szabo. “Business Analytics in (a) Blink.” In: *IEEE Data Eng. Bull.* 35.1 (2012), pp. 9–14.

[23] Ronald Barber, Guy Lohman, Ippokratis Pandis, Gopi Attaluri, Naresh Chainani, Sam Lightstone, Vijayshankar Raman, Richard Sidle, and David Sharpe. “Memory-Efficient Hash Joins.” In: *PVLDB*. 2015.

[24] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. “Rack-Scale In-Memory Join Processing using RDMA.” In: *SIGMOD*. 2015.

[25] Steven Keith Begley, Zhen He, and Yi-Ping Phoebe Chen. “MC-Join: A Memory-Constrained Join for Column-Store Main-Memory Databases.” In: *SIGMOD*. 2012.

[26] Michael A. Bender, Gerth Stølting Brodal, Rolf Fagerberg, Riko Jacob, and Elias Vicari. “Optimal Sparse Matrix Dense Vector Multiplication in the I/O-Model.” In: *SPAA*. 2007.

[27] Dina Bitton and David J. DeWitt. “Duplicate record elimination in large data files.” In: *TODS* 8.2 (1983), pp. 255–265.

[28] Spyros Blanas, Yinan Li, and Jignesh M. Patel. "Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs." In: *SIGMOD*. 2011.

[29] Spyros Blanas and Jignesh M. Patel. "Memory Footprint Matters: Efficient Equi-Join Algorithms for Main Memory Data Processing." In: *SoCC*. 2013.

[30] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. "A Comparison of Join Algorithms for Log Processing in MaPreduce." In: *SIGMOD*. 2010.

[31] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. "Database Architecture Optimized for the New Bottleneck: Memory Access." In: *PVLDB*. 1999.

[32] Peter A. Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution." In: *CIDR*. 2005.

[33] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. "Prototyping Bubba, A Highly Parallel Database System." In: *TKDE* 2.1 (1990), pp. 4–24.

[34] Gerth Stølting Brodal. "Cache-Oblivious Algorithms and Data Structures." In: *SWAT*. 2004.

[35] Gerth Stølting Brodal, Erik D. Demaine, Jeremy T. Fineman, John Iacono, Stefan Langerman, and J. Ian Munro. "Cache-Oblivious Dynamic Dictionaries with Update/Query Trade-offs." In: *SODA*. 2010.

[36] Gerth Stolting Brodal and Rolf Fagerberg. "Lower Bounds for External Memory Dictionaries." In: *SODA*. 2003.

[37] Donald D. Chamberlin and Raymond F. Boyce. "SEQUEL: A Structured English Query Language." In: *SIGFIDET Workshop on Data Description, Access and Control*. 1976.

[38] Badrish Chandramouli and Jonathan Goldstein. "Patience is a Virtue: Revisiting Merge and Sort on Modern Processors." In: *SIGMOD*. 2014, pp. 731–742.

[39] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. "Improving Hash Join Performance through Prefetching." In: *TODS* 32.3 (2007), pp. 1–32.

[40] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. "Inspector joins." In: *PVLDB*. 2005.

[41] John Cieslewicz, William Mee, and Kenneth A. Ross. "Cache-Conscious Buffering for Database Operators with State." In: *DaMoN*. 2009.

[42] John Cieslewicz and K.A. Ross. "Adaptive Aggregation on Chip Multiprocessors." In: *PVLDB*. 2007.

[43] John Cieslewicz and Kenneth A. Ross. "Data partitioning on chip multiprocessors." In: *DaMoN*. 2008.

[44] John Cieslewicz, Kenneth A. Ross, Kyoho Satsumi, and Yang Ye. "Automatic contention detection and amelioration for data-intensive operations." In: *SIGMOD*. 2010.

[45] Edgar F. Codd. "A relational model of data for large shared data banks." In: *Communications of the ACM* 13.6 (1970), pp. 377–387.

[46] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction To Algorithms*. MIT Press, 2001. ISBN: 0262032937.

[47] Andrei Costea and Adrian Ionescu. "Query Optimization and Execution in Vectorwise MPP." MA thesis. Vreije Universiteit Amsterdam, 2012.

[48] Imre Csiszár and Paul C. Shields. "Information Theory and Statistics: A Tutorial." In: *FTCIT* 1.4 (2004), pp. 417–528.

[49] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. "Everything you always wanted to know about synchronization but were afraid to ask." In: *SOSP*. 2013.

[50] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In: *CACM* 51.1 (2008).

[51] Erik D. Demaine. "Cache-Oblivious Algorithms and Data Structures." In: *BRICS* (2002).

[52] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. "Adaptive Query Processing." In: *Foundations and Trends in Databases* 1.1 (2006), pp. 1–140.

[53] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. "Practical Skew Handling in Parallel Joins." In: *PVLDB*. 1992.

[54] David J DeWitt, Randy H Katz, et al. "Implementation Techniques for Main Memory Database Systems." In: *SIGMOD*. 1984.

[55] David DeWitt and Jim Gray. "Parallel Database Systems: The Future of High Performance Database Systems." In: *CACM* 35.6 (1992), pp. 85–98.

[56] D.J. DeWitt, S. Ghandeharizadeh, D.A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. "The Gamma Database Machine Project." In: *TKDE* 2.1 (1990), pp. 44–62.

[57] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. "Hekaton: SQL Server's Memory-Optimized OLTP Engine." In: *SIGMOD*. 2013.

[58] Ulrich Drepper. "What every programmer should know about memory." In: *Red Hat, Inc* (2007). URL: http://people.freebsd.org/~lstewart/articles/cpumemory.pdf.

[59] Marianne Durand and Philippe Flajolet. "Loglog Counting of Large Cardinalities." In: *ESA*. 2003.

[60] Robert Epstein. *Techniques for processing of aggregates in relational database systems*. Tech. rep. Technical Report UCB/ERL, 1979.

[61] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. "The SAP HANA Database – An Architecture Overview." In: *IEEE Data Eng. Bull.* 35.1 (2012), pp. 28–33.

[62] Arash Farzan. "Cache-Oblivious Searching and Sorting in Multisets." PhD thesis. University of Waterloo, 2004.

[63] Arash Farzan, Paolo Ferragina, Gianni Franceschini, and J. Ian Munro. "Cache-oblivious comparison-based algorithms on multisets." In: *ESA*. 2005.

[64] Ziqiang Feng and Eric Lo. "Accelerating Aggregation using Intra-cycle Parallelism." In: *ICDE*. 2015.

[65] Philippe Flajolet, Éric Fusy, and Olivier Gandouet. "HyperLogLog : the analysis of a near-optimal cardinality estimation algorithm." In: *AofA*. 2007.

[66] Philippe Flajolet and G. Nigel Martin. "Probabilistic Counting Algorithms for Data Base Applications." In: *JCSS* 31.2 (1985).

[67] Craig Freedman, Erik Ismert, and Per-Åke Larson. "Compilation in the Microsoft SQL Server Hekaton Engine." In: *IEEE Data Eng. Bull.* 37.1 (2014), pp. 22–30.

[68] Philip W. Frey, Romulo Goncalves, Martin Kersten, and Jens Teubner. "A Spinning Join That Does Not Get Dizzy." In: *ICDCS*. 2010.

[69] Philip W. Frey, Romulo Goncalves, Martin Kersten, and Jens Teubner. "Spinning relations: High-Speed Networks for Distributed Join Processing." In: *DaMoN*. 2009.

[70] M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. "Cache-Oblivious Algorithms." In: *FOCS*. 1999.

[71] Philip Garcia and Henry F. Korth. "Database Hash-Join Algorithms on Multithreaded Computer Architectures." In: *CF*. 2006.

[72] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. "Fast computation of database operations using graphics processors." In: *SIGMOD*. 2004.

[73] Goetz Graefe. "Encapsulation of parallelism in the Volcano query processing system." In: *SIGMOD*. 1990.

[74] Goetz Graefe. "Implementing sorting in database systems." In: *ACM Computing Surveys* 38.3 (2006).

[75] Goetz Graefe. "New algorithms for join and grouping operations." In: *Computer Science – R&D* 27.1 (2011), pp. 3–27.

[76] Goetz Graefe. "Query evaluation techniques for large databases." In: *ACM Computing Surveys* 25.2 (1993), pp. 73–169.

[77] Goetz Graefe, Ross Bunker, and Shaun Cooper. "Hash Joins and Hash Teams in Microsoft SQL Server." In: *PVLDB*. 1998.

[78] J. Gray, A. Bosworth, A. Lyaman, and H. Pirahesh. "Data cube: a relational aggregation operator generalizing GROUP-BY, CROSS-TAB, and SUB-TOTALS." In: *ICDE*. 1996.

[79] Gero Greiner. "Sparse Matrix Computations and their I/O Complexity." PhD thesis. Technische Universität München, 2012.

[80] Gero Greiner and Riko Jacob. "The Efficiency of MapReduce in Parallel External Memory." In: *LATIN*. 2012.

[81] Gero Greiner and Riko Jacob. "The I/O Complexity of Sparse Matrix Dense Matrix Multiplication." In: *LATIN*. 2010.

[82] Sven Helmer, Thomas Neumann, and Guido Moerkotte. *Early Grouping Gets the Skew*. Tech. rep. University of Mannheim, 2011.

[83] Lorenz Hübschle-Schneider, Peter Sanders, and Ingo Müller. "Communication Efficient Algorithms for Top-k Selection Problems." In: *CoRR* abs/1502.0 (2015).

[84] John Iacono and Mihai Pătraşcu. "Using Hashing to Solve the Dictionary Problem (In External Memory)." In: *SODA*. 2012.

[85] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2009.

[86] *Information technology—Database languages—SQL—Part 2: Foundation*. ISO/IEC 9075-2:2011(E). 2011.

[87] Yannis E. Ioannidis and Stavros Christodoulakis. "On the propagation of errors in the size of join results." In: *ACM SIGMOD Record* 20.2 (1991), pp. 268–277.

[88] Riko Jacob, Tobias Lieber, and Nodari Sitchinava. "On the Complexity of List Ranking in the Parallel External Memory Model." In: *MFCS* 8635 (2014), pp. 384–395.

[89] Morten Skaarup Jensen and Rasmus Pagh. "Optimality in External Memory Hashing." In: *Algorithmica* 52.3 (2007), pp. 403–411.

[90] Hong Jia-Wei and H. T. Kung. "I/O complexity: The red-blue pebble game." In: *STOC*. 1981.

[91] D. Jimenez-Gonzalez, J.J. Navarro, and J.-L. Larriba-Pey. "CC-Radix: a cache conscious sorting based on Radix sort." In: *PDP*. 2003.

[92] Dani Jiménez-González, Josep Lluis Larriba-Pey, and Juan J. Navarro. "Case Study: Memory Conscious Parallel Sorting." In: *Algorithms for Memory Hierarchies*. 2003, pp. 355–377.

[93] Daniel Jiménez-González, Juan J. Navarro, and Josep-L. Larrba-Pey. "Fast Parallel In-Memory 64-bit Sorting." In: *ICS*. 2001.

[94] Navin Kabra and David J. DeWitt. "Efficient mid-query re-optimization of sub-optimal query execution plans." In: *ACM SIGMOD Record* 27.2 (1998), pp. 106–117.

[95] Rohit Khurana. *Introduction to Database Systems*. Pearson Education India, 2010. ISBN: 8131731928.

[96] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. "Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs." In: *PVLDB*. 2009.

[97] M. Kitsuregawa, M. Nakayama, and M. Takagi. "The effect of bucket size tuning in the dynamic hybrid GRACE hash join method." In: *PVLDB*. 1989.

[98] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. "Application of Hash to Data Base Machine and Its Architecture." In: *New Generation Computing* 1.1 (1983), pp. 63–74.

[99] Anthony Klug. "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions." In: *Journal of the ACM* 29.3 (1982), pp. 699–717.

[100] Mikael Knudsen and Kirsten Larsen. "I/O-complexity of comparison and permutation problems." MA thesis. Aarhus University, 1992.

[101] Donald E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley Professional, 1998, p. 800. ISBN: 0201896850.

[102] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. "Impala: A Modern, Open-Source SQL Engine for Hadoop." In: *CIDR*. 2015.

[103] Konstantinos Krikellas. "Case for holistic query evaluation." PhD thesis. University of Edinburgh, 2010.

[104] Konstantinos Krikellas, Stratis D. Viglas, and Marcelo Cintra. "Generating code for holistic query evaluation." In: *ICDE*. 2010.

[105] V. Kumar and E.J. Schwabe. "Improved algorithms and data structures for solving graph problems in external memory." In: *SPDP*. 1996.

[106] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. "SkewTune: Mitigating Skew in MapReduce Applications." In: *SIGMOD*. 2012.

[107] Yongchul Kwon, Kai Ren, Magdalena Balazinska, and Bill Howe. "Managing Skew in Hadoop." In: *IEEE Data Eng. Bull.* 36.1 (2013), pp. 24–33.

[108] Tirthankar Lahiri, Ma Neimat, and Steve Folkman. "Oracle TimesTen: An In-Memory Database for Enterprise Applications." In: *IEEE Data Eng. Bull* 36.2 (2013), pp. 6–13.

[109] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. "Massively Parallel NUMA-aware Hash Joins." In: *IMDM*. 2013.

[110] Eric N. Larson Påand Hanson and Susan L. Price. "Columnar storage in SQL Server 2012." In: *IEEE Data Eng. Bull.* 35.1 (2012), pp. 15–20.

[111] Per-Åke Larson. *Grouping and duplicate elimination: Benefits of early aggregation*. Tech. rep. Microsoft, 1997. URL: http://research.microsoft.com/pubs/69693/tr-97-36.pdf.

[112] Per-Åke Larson Larson. "Data reduction by partial preaggregation." In: *ICDE*. 2002.

[113] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. "Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age." In: *SIGMOD*. 2014.

[114] D. Lemire and L. Boytsov. "Decoding billions of integers per second through vectorization." In: *Softw. Pract. Exper.* 45.1 (2015), pp. 1–29.

[115] Christian Lemke, Kai-Uwe Sattler, Franz Faerber, and Alexander Zeier. "Speeding Up Queries in Column Stores – A Case for Compression." In: *DaWaK*. 2010, pp. 117–129.

[116] Yinan Li, Ippokratis Pandis, Rene Mueller, Vijayshankar Raman, and Guy Lohman. "NUMA-aware algorithms: the case of data shuffling." In: *CIDR*. 2013.

[117] Yinan Li and Jignesh M. Patel. "BitWeaving: Fast Scans for Main Memory Data Processing." In: *SIGMOD*. 2013.

[118] Guy M. Lohman. *Is Query Optimization a "Solved" Problem?* 2014. URL: http://wp.sigmod.org/?p=1075 (visited on 05/07/2015).

[119] Guy M. Lohman. “Is query optimization a “solved” problem.” In: *Proc. Workshop on Database Query Optimization*. 1989.

[120] Walter T. Ludwig. “Algorithms for Scheduling Malleable and Nonmalleable Parallel Tasks.” PhD thesis. University of Wisconsin-Madison, 1995.

[121] Roger MacNicol and Blaine French. “Sybase IQ Multiplex - Designed For Analytics.” In: *PVLDB*. 2004.

[122] S. Manegold, P. Boncz, and M. Kersten. “Optimizing main-memory join on modern hardware.” In: *TKDE* 14.4 (2002), pp. 709–730.

[123] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. “Optimizing database architecture for the new bottleneck: memory access.” In: *The VLDB Journal* 9.3 (2000), pp. 231–246.

[124] Stefan Manegold, Peter Boncz, Niels Nes, and Martin Kersten. “Cache-conscious radix-decluster projections.” In: *PVLDB*. 2004.

[125] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. “Robust query processing through progressive optimization.” In: *SIGMOD*. 2004.

[126] Xavier Martinez-Palau, David Dominguez-Sal, and Josep Lluis Larriba-Pey. “Two-way Replacement Selection.” In: *PVLDB*. 2010.

[127] Yossi Matias, Eran Segal, and Jeffrey Scott Vitter. “Efficient Bundle Sorting.” In: *SICOMP* 36.2 (2006), p. 394.

[128] Jean-Pierre Maury. *Newton: Understanding the Cosmos*. Thames and Hudson, 1992. ISBN: 0500300232.

[129] Manish Mehta and David J. DeWitt. “Managing Intra-operator Parallelism in Parallel Database Systems.” In: *PVLDB*. 1995.

[130] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. “Why Go Logarithmic if We Can Go Linear? Towards Effective Distinct Counting of Search Traffic.” In: *EDBT*. 2008.

[131] Rene Mueller, Jens Teubner, and Gustavo Alonso. “Data processing on FPGAs.” In: *PVLDB*. 2009.

[132] Ingo Müller, Cornelius Ratsch, and Franz Faerber. “Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems.” In: *EDBT*. 2014.

[133] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. “Cache-Efficient Aggregation: Hashing Is Sorting.” In: *SIGMOD*. 2015.

[134] Ingo Müller, Peter Sanders, Robert Schulze, and Wei Zhou. “Retrieval and Perfect Hashing using Fingerprinting.” In: *SEA*. 2014.

[135] Matthias Müller-Hannemann and Stefan Schirra. *Algorithm Engineering—Bridging the Gap Between Algorithm Theory and Practice*. Springer Berlin Heidelberg New York, 2010, p. 527. ISBN: 978-3-642-14865-1.

[136] Ian Munro and Philip M. Spira. "Sorting and Searching in Multisets." In: *SICOMP* 5.1 (1976), p. 1.

[137] J. Ian Munro and Venkatesh Raman. "Sorting Multisets and Vectors In-Place." In: *WADS*. 1991.

[138] Fabian Nagel, Gavin Bierman, and Stratis D. Viglas. "Code generation for efficient query processing in managed runtimes." In: *PVLDB*. 2014.

[139] Masaya Nakayama, Masaru Kitsuregawa, and Mikio Takagi. "Hash-Partitioned Join Method Using Dynamic Destaging Strategy." In: *PVLDB*. 1988.

[140] Thomas Neumann. "Efficiently Compiling Efficient Query Plans for Modern Hardware." In: *PVLDB*. 2011.

[141] Hung Q. Ngo, Dung T. Nguyen, Christopher Re, and Atri Rudra. "Beyond Worst-case Analysis for Joins with Minesweeper." In: *PODS*. 2014.

[142] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. "Worst-case Optimal Join Algorithms." In: *PODS*. 2012.

[143] M. Tamer Özsu and P Valduriez. *Principles of distributed database systems*. 3rd ed. Springer, 2011. ISBN: 9781441988331.

[144] Rasmus Pagh and Morten Stöckel. "The Input/Output Complexity of Sparse Matrix Multiplication." In: *ESA*. 2014.

[145] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Elsevier, 2012, p. 703. ISBN: 0123747503.

[146] Holger Pirk, Stefan Manegold, and Martin Kersten. "Waste not... Efficient co-processing of relational data." In: *ICDE*. 2014.

[147] Fred J. Pollack. "New microarchitecture challenges in the coming generations of CMOS process technologies." In: *MICRO* (1999), p. 2.

[148] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. "Rethinking SIMD Vectorization for In-Memory Databases." In: *SIGMOD*. 2015.

[149] Orestis Polychroniou and Kenneth A. Ross. "A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort." In: *SIGMOD*. 2014.

[150] Orestis Polychroniou, Rajkumar Sen, and Kenneth A. Ross. "Track Join: Distributed Joins with Minimal Network Traffic." In: *SIGMOD*. 2014.

[151] 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: *PVLDB*. 2015.

[152] Naila Rahman. "Algorithms for Hardware Caches and TLB." In: *Algorithms for Memory Hierarchies*. 2003, pp. 171–192.

[153] Smriti R. Ramakrishnan, Garret Swart, and Aleksey Urmanov. "Balancing Reducer Skew in MapReduce Workloads using Progressive Sampling." In: *SoCC*. 2012.

[154] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, et al. "DB2 with BLU Acceleration: So Much More than Just a Column Store." In: *PVLDB*. 2013.

[155] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. "High-Speed Query Processing over High-Speed Networks." In: *CoRR* abs/1502.0 (2015).

[156] Wolf Rodiger, Tobias Muhlbauer, Philipp Unterbrunner, Angelika Reiser, Alfons Kemper, and Thomas Neumann. "Locality-Sensitive Operators for Parallel Main-Memory Database Clusters." In: *ICDE*. 2014.

[157] Milan Ružić. "Constructing Efficient Dictionaries in Close to Sorting Time." In: *ICALP*. 2008.

[158] Peter Sanders. "Algorithm Engineering – An Attempt at a Definition." In: *Efficient Algorithms*. 2009.

[159] Peter Sanders, Sebastian Schlag, and Ingo Müller. "Communication efficient algorithms for fundamental big data problems." In: *IEEE Big Data Conf.* 2013.

[160] Peter Sanders and Sebastian Winkel. "Super Scalar Sample Sort." In: *ESA*. 2004.

[161] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. "Fast sort on CPUs and GPUs." In: *SIGMOD*. 2010, p. 351.

[162] Felix Martin Schuhknecht, Pankaj Khanchandani, and Jens Dittrich. "On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning." In: *PVLDB*. 2015.

[163] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. "Cache Conscious Algorithms for Relational Query Processing." In: *PVLDB*. 1994.

[164] Ambuj Shatdal and Jeffrey F. Naughton. "Adaptive Parallel Aggregation Algorithms." In: *SIGMOD*. 1995.

[165] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. "The Hadoop Distributed File System." In: *MSST*. 2010.

[166] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 2011. ISBN: 9780073523323.

[167] Nodari Sitchinava. Personal communication. 2014.

[168] Nodari Sitchinava and Norbert Zeh. "A parallel buffer tree." In: *SPAA*. 2012.

[169] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. "LEO - DB2's LEarning Optimizer." In: *PVLDB*. 2001.

[170] Michael Stonebraker. "The Case for Shared Nothing." In: *IEEE Data Eng. Bull.* 9.1 (1986), pp. 4–9.

[171] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. "C-Store: A Column-oriented DBMS." In: *PVLDB*. 2005.

[172] *TPC Benchmark™ H*. Revision 2.17.1. 2014.

[173] Todd L Veldhuizen. "Leapfrog Triejoin : A Simple , Worst-Case Optimal Join Algorithm." In: *ICDT*. 2014.

[174] Elad Verbin and Qin Zhang. "The Limits of Buffering: A Tight Lower Bound for Dynamic Membership in the External Memory Model." In: *STOC*. 2010.

[175] Rares Vernica, Andrey Balmin, Kevin S. Beyer, and Vuk Ercegovac. "Adaptive MapReduce using Situation-Aware Mappers." In: *EDBT*. 2012.

[176] Jeffrey Scott Vitter. *Algorithms and Data Structures for External Memory*. Now Publishers Inc, 2008. ISBN: 1601981066.

[177] Jeffrey Scott Vitter. "External Memory Algorithms." In: *ESA*. 1998.

[178] Jeffrey Scott Vitter. "External memory algorithms and data structures: dealing with massive data." In: *ACM Computing Surveys* 33.2 (2001), pp. 209–271.

[179] Florian M. Waas. "Beyond Conventional Data Warehousing — Massively Parallel Data Processing with Greenplum Database (Invited Talk)." In: *BIRTE*. 2009.

[180] Mehul Wagle, Daniel Booss, Ivan Schreter, and Daniel Egenolf. "NUMA-Aware Memory Management with In-Memory Databases." In: *TPCTC*. 2015.

[181] Christopher B. Walton, Alfred G. Dale, and Roy M. Jenevein. "A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins." In: *PVLDB*. 1991.

[182] Jan Wassenberg and Peter Sanders. "Engineering a Multi-core Radix Sort." In: *Euro-Par*. 2011.

[183] L.M. Wegner and J.I. Teuhola. “The External Heapsort.” In: *TSE* 15.7 (1989), pp. 917–925.

[184] Zhewei Wei, Ke Yi, and Qin Zhang. “Dynamic External Hashing: The Limit of Buffering.” In: *SPAA*. 2009.

[185] Martin Weidner, Jonathan Dees, and Peter Sanders. “Fast OLAP Query Execution in Main Memory on Large Data in a Cluster.” In: *IEEE Big Data Conf.* 2013.

[186] Jian Wen. “Revisiting Aggregation Techniques for Data Intensive Applications.” PhD thesis. University of California, Riverside, 2013.

[187] Jian Wen, Vinayak R. Borkar, Michael J. Carey, and Vassilis J. Tsotras. “Revisiting Aggregation Techniques for Data Intensive Applications: A Performance Study.” In: *CoRR* abs/1311.0 (2013).

[188] Wikipedia. *Integer Sorting — Wikipedia, The Free Encyclopedia*. 2015. URL: http://en.wikipedia.org/wiki/Integer_sorting (visited on 01/22/2015).

[189] Wikipedia. *Russell's paradox — Wikipedia, The Free Encyclopedia*. 2015. URL: https://en.wikipedia.org/wiki/Russell's_paradox (visited on 11/27/2015).

[190] Wikipedia. *Stirling's approximation — Wikipedia, The Free Encyclopedia*. 2015. URL: https://en.wikipedia.org/wiki/Stirling's_approximation (visited on 09/11/2015).

[191] Wiktionary. *appendix – Wiktionary*. 2015. URL: https://en.wiktionary.org/wiki/appendix (visited on 10/07/2015).

[192] Wiktionary. *practice – Wiktionary*. 2015. URL: https://en.wiktionary.org/wiki/practice (visited on 10/07/2015).

[193] Wiktionary. *state of the art – Wiktionary*. 2015. URL: https://en.wiktionary.org/wiki/state_of_the_art (visited on 10/07/2015).

[194] Wiktionary. *theory – Wiktionary*. 2015. URL: https://en.wiktionary.org/wiki/theory (visited on 10/07/2015).

[195] Thomas Willhalm, Ismail Oukid, Ingo Müller, and Franz Faerber. “Vectorizing Database Column Scans with Complex Predicates.” In: *ADMS*. 2013.

[196] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. “SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units.” In: *PVLDB*. 2009.

[197] J.L. Wolf, D.M. Dias, and P.S. Yu. “A Parallel Sort Merge Join Algorithm for Managing Data Skew.” In: *TPDS* 4.1 (1993), pp. 70–86.

[198] J.L. Wolf, P.S. Yu, J. Turek, and D.M. Dias. "A Parallel Hash Join Algorithm for Managing Data Skew." In: *TPDS* 4.12 (1993), pp. 1355–1371.

[199] Yang Ye, Kenneth A. Ross, and Norases Vesdapunt. "Scalable Aggregation on Multicore Processors." In: *DaMoN*. 2011.

[200] Ke Yi. "Dynamic Indexability and the Optimality of B-Trees." In: *JACM* 59 (2012), 21:1–21:19.

[201] Ke Yi and Qin Zhang. "On the Cell Probe Complexity of Dynamic Membership." In: *SODA*. 2010.

[202] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster Computing with Working Sets." In: *HotCloud*. 2010.

[203] Hao Zhang, Gang Chen, and Chin Ooi. "In-Memory Big Data Management and Processing: A Survey." In: *TKDE* 27.7 (2015), pp. 1920–1948.

[204] Jingren Zhou. "Architecture-sensitive database query processing." PhD thesis. Columbia University, 2004.

[205] Jingren Zhou and Kenneth A. Ross. "Implementing database operations using SIMD instructions." In: *SIGMOD*. 2002.

[206] Marcin Zukowski, S. Heman, N. Nes, and P. Boncz. "Super-Scalar RAM-CPU Cache Compression." In: *ICDE*. 2006.

[207] Marcin Zukowski, Sándor Héman, and Peter Boncz. "Architecture-Conscious Hashing." In: *DaMoN*. 2006.

ZUSAMMENFASSUNG

Seit Jahrzehnten erfährt unsere Gesellschaft eine fortschreitende Digitalisierung fast all ihrer Bereiche: Wirtschaftsprozesse, Finanztransaktionen, Regierungsführung, Online-Shopping, Forschung in allen Disziplinen, Medizin, persönliche Kommunikation, Transportwesen, Wohnen und viele mehr. Die Menschheit produziert und sammelt nie dagewesene Mengen an Daten, die schwierig zu fassen sind, und die Fähigkeit, aus ihnen Erkenntnisse abzuleiten, wird gleichzeitig immer wichtiger und technisch schwieriger. Mit diesem Ziel ist eine große Vielfalt an Software-Anwendungen entstanden – oft unter dem Begriff *Business Intelligence* zusammengefasst –, die Konzepte wie Datenanalyse, Berichtswesen, *Online Analytical Processing* (OLAP), Data-Mining, etc. anbieten und kontinuierlich ausgebaut werden.

Dieser Trend wird begleitet von technischen Fortschritten bei Computer-Hardware. Rechenleistung und Speicherkapazität unterliegen seit ihren ersten Tagen vor einem halben Jahrhundert einem exponentiellen Wachstum und versprechen dies auch in Zukunft weiter zu tun. Während langer Zeit war diese Entwicklung für Software transparent, aber heute kann Leistungssteigerung oft nur durch Spezialisieren und explizites Exponieren von mehr und mehr Hardware-Komponenten erreicht werden. Software kann daher nur dann von der vollen Rechenleistung profitieren, wenn sie sich der inneren Funktionsweise der Hardware bewusst ist. Dies bedeutet, dass mehr und mehr Verantwortung auf die Ebene der Software verlagert wird, wodurch Software immer komplexer und schwieriger zu entwickeln wird.

Um den Zusammenfluss dieser zwei Trends zu bewältigen, wurde eine spezielle Software als Abstraktionsschicht entwickelt: Relationale Datenbankmanagementsysteme. Sie bieten eine logische, strukturierte Sicht auf Daten und entlasten dadurch Anwendungen um viele technische Aspekte des Umgangs mit Daten, sodass diese sich darauf konzentrieren können, *was* zu tun ist, während die Systeme sich darum kümmern, *wie* es getan wird. Insbesondere ist es die Aufgabe von Datenbankmanagementsystemen, Anfragen von Anwendungen so zu bearbeiten, dass dabei das volle Potential der Hardware genutzt wird. Datenbankmanagementsysteme zu bauen ist daher ein ständiger Wettlauf, sich an jede neue Hardware-Generation anzupassen, um immer schneller die ständig wachsenden Datenmengen bearbeiten zu können.

Im Herzen diesen Wettlaufs liegt eine Reihe von *Operatoren* – algorithmische Bausteine, die Datenbanksysteme kombinieren, um auf Anfrage von Anwendungen Berechnungen auszuführen. Eine der wichtigsten Operatoren ist die Aggregation. Sie besteht darin, eine

Menge von Datenbankeinträgen zusammenzufassen, indem die Einträge in Gruppen eingeteilt werden und jede Gruppe zu einem einzelnen Wert aufaggregiert wird. So könnten zum Beispiel die Buchhaltungseinträge aller verkauften Waren einer Kaufhauskette durch die Anzahl verkaufter Artikel pro Filiale zusammengefasst werden. Nahezu alle Anwendungen zur Datenanalyse benötigen diese Funktionalität und ein typisches Datenbanksystem verbringt einen großen Teil seiner Rechenzeit mit diesem Operator. Das Ziel der vorliegenden Dissertation ist es, eine neue Generation von AGGREGATIONS-Operatoren zu bauen, die Datenbanksysteme in ihrem Wettlauf um das Ausnutzen neuer Hardware voranbringt.

AGGREGATIONS-Operatoren zu bauen, ist aus mehreren Gründen herausfordernd. Beginnend mit oben erwähntem Hardware-Bewusstsein sind folgendes konkrete Herausforderungen: Erstens ist es für diese Operatoren wichtig, dass sie kostenintensive Datenbewegungen reduzieren. In modernen Datenbanksystemen, die fast all ihre Daten im Hauptspeicher halten, betrifft dies vertikale Datenbewegungen zwischen Hauptspeicher und Prozessor-Caches, aber auch horizontale Datenbewegungen zwischen verschiedenen Recheneinheiten. Weiterhin können moderne Prozessoren bestimmte Arten von Rechenanweisungen schneller ausführen als andere, sodass Algorithmen so implementiert werden sollten, dass sie so oft wie möglich effiziente Anweisungsabfolgen verwenden. Zuletzt bestehen moderne Computer aus einer großen Zahl von parallel laufenden Recheneinheiten – in Form von mehreren Komponenten auf dem gleichen Chip, in Form von mehreren Chips innerhalb des gleichen Computers oder in Form von mehreren zusammengeschalteten Computern. Software kann daher nur dann alle verfügbare Rechenleistung nutzen, wenn sie ihre Arbeit in Arbeitspakete aufteilen kann, die gleichzeitig und unabhängig voneinander ausgeführt werden können, und dabei nur ein geringes Maß an Kommunikation zwischen den beteiligten Recheneinheiten geschieht. Zusammengefasst müssen AGGREGATIONS-Operatoren, um moderne Hardware voll auszunutzen, so gebaut werden, dass sie *cache-effizient*, *prozessor-freundlich*, hochgradig *parallelisierbar* und *kommunikationseffizient* sind.

Eine weitere Gruppe von Herausforderungen hängt mit der gewünschten allgemeinen Einsetzbarkeit von Datenbankoperatoren zusammen. Diese sollten unabhängig von den Eigenschaften der Eingabedaten eine gute Leistung erreichen. Bei AGGREGATION funktionieren mit moderner Hardware, abhängig von der Anzahl der Gruppen in den Daten, unterschiedliche Strategien am besten. Diese Eigenschaft der Daten ist schwierig im Voraus zu bestimmen. Daher ist es für AGGREGATIONS-Operatoren attraktiv, ihre Strategie während der Ausführung *adaptiv* an die Daten anzupassen. Ähnlich verhält es sich in den schwierigen Situationen, wo einer bestimmten Gruppe deutlich mehr Einträge angehören als anderen. Um robust für alle Anwendungen zu funktionieren, brauchen wir also Mechanismen, die mit *schiefen Verteilungen* von Gruppen umgehen können. Sowohl *Adaptivität* als auch *Schiefebeständigkeit* sind daher wünschenswerte Eigenschaften, welche zu erreichen eine Herausforderung darstellt.

Eine letzte Gruppe von Herausforderungen kommt von der Tatsache, dass AGGREGATIONS-Operatoren Teil eines größeren Systems sind. Erstens bedeutet das, dass sie Hardware-Ressourcen mit anderen Software-Komponenten teilen müssen. Im Kontext von hauptspeicherbasierten Datenbanksystemen ist das besonders schwierig mit der Ressource Hauptspeicher. Operatoren brauchen Hauptspeicher, um Zwischenergebnisse abzulegen, aber je nach System kann die dafür zur Verfügung stehende Menge eingeschränkt werden. Außerdem haben unterschiedliche Arten von Datenbankmanagementsystemen unterschiedliche Schnittstellen zwischen ihren Operatoren, was manche Entscheidungen im Algorithmen-Design beeinflussen kann. Beim Entwurf von AGGREGATIONS-Operatoren ist es daher wichtig, darauf zu achten, dass sie selbst mit *beschränktem Speicherplatz* laufen und in die Architektur von Datenbankmanagementsytemen des neusten Stands der Technik *integrierbar* sind.

Jede dieser Herausforderungen für sich genommen ist bereits schwer zu erreichen. Die Aufgabe wird jedoch weiter verkompliziert, wenn sie alle *gleichzeitig* angegangen werden sollen. Jede Design-Entscheidung, um eine bestimmte Eigenschaft zu verbessern, kann negativ beeinflussen, wie gut sich der entstehende Operator in einer anderen Dimension verhält. Die Aufgabe, einen kompletten AGGREGATIONS-Operator zu bauen, ist daher auch deshalb herausfordernd, weil sie darin besteht, verschiedene Kosten-Nutzen-Abwägungen in ein Gleichgewicht zu bringen, sodass eine insgesamt gute Lösung entsteht.

In dieser Arbeit gehen wir die Herausforderungen an, AGGREGATIONS-Operatoren im Kontext von hauptspeicherbasierten Datenbanken zu konstruieren.

In einem ersten Teil untersuchen wir AGGREGATION von einem theoretischen Blickwinkel aus. Wir analysieren das Problem in verschiedenen Externen-Speicher-Modellen, um seine Grenzen bei Cache-Effizienz zu verstehen. Wir können zeigen, dass AGGREGATION unter vielen realistischen Parameterwerten dieser Maschinenmodelle dieselbe untere Schranke hat wie MULTIMENGENSORTIEREN. Dies beweist eine langjährige folkloristische Vermutung der Datenbankgemeinschaft, die impliziert, dass die beste Strategie für AGGREGATION davon abhängt, wie viele Gruppen die Eingabe enthält, und dass es optimal ist, einen sortier-basierten Algorithmus zu verwenden, wenn die Anzahl der Gruppen groß ist. Die Erkenntnisse dieser theoretischen Analyse stellen eine zeitlose Richtlinie für den Entwurf von cache-effizienten AGGREGATIONS-Algorithmen dar.

In einem zweiten Teil konstruieren wir in mehreren Iterationen einen praxisbezogenen Algorithmus, der allen oben genannten Herausforderungen gerecht wird. In der ersten Iteration stellen wir die allgemein akzeptierte Ansicht in Frage, dass die zwei traditionellen Strategien, HashAggregation und SortierAggregation, komplett gegensätzlich sind. Stattdessen entwerfen wir einen einzigen Algorithmus, der Merkmale von beiden Strategien aufweist. Ein einfacher, kostengünstiger Mechanismus lässt unseren Operator sein Verhalten während der Ausführung an die Daten anpassen, ohne auf im Voraus bestimmte, externe Informationen angewiesen zu sein. Er ist daher mindestens so cache-effizient wie der jeweils beste der zwei statischen Auswahlmöglichkeiten. Außerdem versorgt ein behutsam integrierter, probabilistischer Streaming-Algorithmus unseren Operator mit einer genauen Schätzung der Anzahl der Gruppen in der Eingabe, was potentiell untragbare Kosten durch das Verändern der Größe der Datenstruktur verhindert, die das Ergebnis halten wird. Wir verwenden auch einige systemnahe Tuning-Techniken, um unseren Algorithmus prozessor-freundlich zu machen und sicherzustellen, dass er mit Ausführungsmodellen von Datenbanksystemen integriert werden kann, die dem neusten Stand der Technik entsprechen. Schließlich zeigen wir, wie unser Operator sowohl innerhalb eines Prozessors als auch über Prozessorgrenzen hinweg und bei schiefer Eingabeverteilung so parallelisiert werden kann, dass er zu jeder Zeit vollkommen parallel ausgeführt wird.

In einer zweiten Iteration verabschieden wir uns von der verbreiteten, aber unrealistischen Annahme, dass unbegrenzt Speicher für Zwischenergebnisse verfügbar ist. Wir entwickeln Techniken, die es erlauben, verschiedene Phasen eines rekursiven Algorithmus in einer Pipeline auszuführen, und wenden sie auf unseren Aggregations-Operator an. Diese Techniken stellen sicher, dass Zwischenergebnisse nach einer Phase rasch von der nächsten Phase konsumiert und ihr Speicher freigegeben werden kann. Das häufigere, dem Pipelining inhärente Wechseln zwischen Rechenaufgaben ist eine mögliche Quelle von Mehraufwand, aber durch cleveres Intra-Operatoren-Scheduling erhalten wir eine Leistung aufrecht, die nahe an der der ursprünglichen Version liegt, während die Adaptivität und Beständigkeit gegen Datenschiefe des Operators erhalten bleiben.

Wir bestätigen die Machbarkeit unseres Algorithmenentwurfs mit umfangreichen Experimenten. Im Vergleich mit mehreren Konkurrenzalgorithmen vom neusten Stand der Technik ist unsere Implementierung in fast allen Situationen die schnellste; in vielen Situationen mit einem großen Vorsprung, der in einer 3,7 mal niedrigeren Ausführungszeit für eine große Anzahl von Gruppen gipfelt. Unser Operator läuft mit der Geschwindigkeit der Speicherbandbreite, skaliert bis zu einer großen Anzahl von Recheneinheiten, und erreicht selbst bei extrem schiefen Eingaben die gleiche Leistung. Unser Kardinalitätsschätzer beseitigt effektiv und mit geringem Mehraufwand die Vergrößerungskosten der Ergebnisdatenstruktur, sodass bis zur

Hälfte der Ausführungszeit eingespart werden kann. Wenn der Speicher für Zwischenergebnisse auf gerade einmal 1.6 % des ursprünglichen Speicherverbrauchs beschränkt wird, erfährt unser Operator einen Mehraufwand von nur 20 % bis 47 % und kann durch Lockerung der Begränzung zu einer höheren Leistung gebracht werden.

Zusammengefasst konstruieren wir durch Kombination einer sorgfältigen theoretischen Analyse, cleverem Algorithmen-Design und einer Reihe von Ingenieurstechniken einen einzelnen, vielseitig einsetzbaren AGGREGATIONS-Operator, der mit Blick auf alle Herausforderungen auf dem neusten Stand der Technik ist, die im Kontext von hauptspeicherbasierten Datenbanksysteme aufkommen.

Ingo Müller

Curriculum Vitae

phd@ingomueller.net
born March 2, 1986 in Breisach, Germany

Research Interests

Core database system algorithms and data structures with a focus on adaptive mechanisms and efficient usage of modern hardware.

Academic Background

since 2011	**PhD in Computer Science**, *Karlsruhe Institute of Technology (KIT)*, Germany. Topic: Engineering Aggregation Operators for Relational In-Memory Database Systems. Supervisor: Prof. Peter Sanders. Industry Partner: SAP SE. Graduation: February 11, 2016. Grade: summa cum laude
2010	**Triple-Degree in Computer Science:**
2005–2010	**MSc in Computer Science**, *Karlsruhe Institute of Technology (KIT)*, Germany. Rank: 5%, Average: 1.1/5.0.
2008–2010	**Master's Degree in Engineering**, *Grenoble INP–Ensimag*, Grenoble, France. Rank: 2nd (of 252), Average: 17.0/20.
2009–2010	**MSc in Computer Science**, *Université Joseph-Fourier (UJF)*, Grenoble, France. Rank: 2nd (of 19), Average: 17.4/20.
2005	**Abitur** (sehr gut) and **Baccalauréat** (mention trés bien), Breisach, Germany. Prizes for my achievements in Physics and French.

Honors & Awards

2015	**ACM SIGMOD 2015 Programming Contest Finalist**. Achieved 2nd place of 38. Received SIGMOD Student Travel Award for finalists.
2010	**Graduation Award**, *Karlsruhe Institute of Technology (KIT)*. Award granted to the top 10% students of the class.
2009–2015	**Member of SAP FastTrack**. Alumni program for the top 10% interns.
2008–2010	**Mobility Scholarship**, *Deutsch-Französische Hochschule (French-German University)*.

Publications

Scientific

SIGMOD 2015 Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, Franz Färber. Cache-Efficient Aggregation: Hashing *Is* Sorting. In *Proceedings of the 36th ACM SIGMOD International Conference on Management of Data*, 2015.

CoRR 2015 Lorenz Hübschle-Schneider, Peter Sanders, Ingo Müller. Communication Efficient Algorithms for Top-k Selection Problems. In *Computing Research Repository*, 2015. arXiv:1502.03942 [cs.DS].

SEA 2014 Ingo Müller, Peter Sanders, Robert Schulze, Wei Zhou. Retrieval and Perfect Hashing using Fingerprinting. In *Proceedings of the 13th International Symposium on Experimental Algorithms*, 2014.

EDBT 2014 Ingo Müller, Cornelius Ratsch, Franz Faerber. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *Proceedings of the 17th International Conference on Extending Database Technology*, 2014

IEEE BigData 2013 Peter Sanders, Sebastian Schlag, Ingo Müller. Communication Efficient Algorithms for Fundamental Big Data Problems. In *Proceedings of the IEEE International Conference on Big Data*, 2013

ADMS 2013 Thomas Willhalm, Ismail Oukid, Ingo Müller. Vectorizing Database Column Scans with Complex Predicates. In *Proceedings of the Fourth International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, 2013.

Dat. Eng. Bull. 2012 Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, Jonathan Dees. The SAP HANA Database – An Architecture Overview. In *IEEE Computer Society Data Engineering Bulletin*, 35 (1): 28-33, 2012.

Master's Thesis Ingo Müller. Experimental Method Provides Adaptation of Parallel Algorithms in OpenCL to Fit a Variety of Platforms Preserving Near-Optimal Performance. Master's Thesis, *Université Joseph-Fourier, Grenoble* and *Karsrluhe Institute of Technology*, 2010.

Patents

2016 Arnaud Lacurie, Ingo Müller. Dynamic Hash Table Size Estimation During Database Aggregation Processing. US Patent App. 15/016,978.

2016 Arnaud Lacurie, Ingo Müller. Memory-Constrained Aggregation Using Intra-Operator Pipelining. US Patent App. 15/040,501.

2015 Arnaud Lacurie, Gordon Gaumnitz, Jonathan Dees, Ingo Müller. Aggregating Database Entries By Hashing. US Patent App. 14/726,251; EPO App. 16001194.6.

2014 Ingo Müller, Cornelius Ratsch, Franz Färber. Adaptive Dcitionary Compression/Decompression for Column-Store Databases. US Patent App. 14/139,669.

2012 Ingo Müller, Peter Sanders. Hash Table and Radix Sort Based Aggregation. US Patent App. 13/729,111

2011 Frederik Transier, Christian Mathis, Nico Bohnsack, Kai Stammerjohann, Peter Sanders, Ingo Müller. Aggregation in Parallel Computation Environments with Shared Memory. European Patent EP2469423 A1.

Conference & Industry Talks

Conferences

SIGMOD 2015 Cache-Efficient Aggregation: Hashing *Is* Sorting. Conference presentation, *SIGMOD*, 2015.

- Also presented at EPFL (2015), ETH Zürich (2015), TU Dortmund (2015), and SAP HANA Campus.

SIGMOD 2015	"SimpleMinds" in the SIGMOD Programming Contest. Contest finalist presentation, *SIGMOD*, 2015.
EDBT 2014	Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. Conference presentation, *EDBT*, 2014.

Industry

HANA Campus	Data compression: past, present, and a foresight (slides by Paolo Ferragina). Seminar presentation, *SAP HANA Database Campus Research Seminar*, 2014.
EPFL Forum	Current Research Project in the SAP HANA Database Campus. Industry talk, *EPFL Forum*, 2014.
HANA Campus	Introduction to H-Store, C-Store, MonetDB, and PAX. Seminar presentation, *SAP HANA Database Campus Research Seminar*, 2012.
SAP Research Workshop	Parallel Aggregation + "X". Workshop presentation, *SAP HANA Database Research Workshop*, 2012.
HANA Campus	Proving Lower Bounds in the External Memory Model. Seminar presentation, *SAP HANA Database Campus Research Seminar*, 2012.
HANA Campus	Adaptive Aggregation on Chip Multiprocessors (slides by John Cieslewicz and Kenneth A. Ross). Seminar presentation, *SAP HANA Database Campus Research Seminar*, 2011.
HANA Campus	Using Radix Sort for Aggregation. Seminar presentation, *SAP HANA Database Campus Research Seminar*, 2011.
IBM Research	Performance Portability for OpenCL thru Auto-Tuning – A Case Study on Financial Applications. Invited talk, *IBM Research Division*, Zurich, 2010.
IBM Montpellier	Introduction to OpenCL. Training course (1 day), *Deep Computing Team, PSSC, IBM*, Montpellier, France, 2010.

Thesis Supervision

2015	Cheng Chen. Vectorizing Recompression in Column-Based In-Memory Database Systems. Master's Thesis, *Technical University Dresden*, Germany.
2013	Arnaud Lacurie. Memory Constraint Aggregation Operator with Adaption to Output Size. Master's Thesis, *Grenoble INP–Ensimag*, Grenoble, France.
2013	Wei Zhou. A Compact Cache-Efficient Function Store with Constant Evaluation Time. Bachelor Thesis, *Karlsruhe Institute of Technology*, Germany.
2013	Cornelius Ratsch. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. Master's Thesis, *Universität Heidelberg*, Germany.
2013	Sebastian Schlag. Distributed Duplicate Removal. Master's Thesis, *Karslruhe Institute of Technology*, Germany.
2012	Andreas Schuster. Compressed Data Structures for Tuple Identifiers in Column-Oriented Databases. Master's Thesis, *Universität Heidelberg*, Germany.

Leadership and Service

Peer Reviews

Ext. Reviewer	SIGMOD 2015, SPAA 2014, IMDM 2013, DAWAK 2013, CSRD 2011.
Program Comm.	ALLDATA 2015.

SAP HANA Database Campus

Public Relations:

- Co-organized SAP booth and travel grant for EDBT 2013.
- Created and maintained SAP HANA Database Campus web site.
- Introduced display of student portraits and paper abstracts in the hallway of the SAP HANA Database Campus.

Recruiting Events:

- Co-organized Open House Day of the SAP HANA Database Campus, SAP Headquarters, 2014.
- Represented SAP at Grenoble INP–Ensimag Internship Fair, Grenoble, 2013.
- Co-organized Programming Contest, Karlsruhe Institute of Technology, 2012.
- Represented SAP at EPFL Forum, EPFL, 2012.
- Represented SAP at Tag der Informatik, University of Heidelberg, 2012.

Campus organization:

- Organized the SAP Database Research Workshop (2 days), November 2012.
- Organized the SAP HANA Database Campus Research Seminar.
- Created HANA Database Campus internal wiki and git server.
- Co-organized regular team building events for students.
- Synchronized with other SAP locations.

Work Experience & Internships

2011–2015 (5 years)	**Full time researcher**, *SAP HANA Database Campus*, Walldorf, Germany. - Cooperation of SAP SE and Karlsruhe Institute of Technology as PhD thesis project. - Research and product integration of core database system algorithms and data structures.
2010 (6 months)	**Internship**, *Deep Computing Team, PSSC, IBM*, Montpellier, France. - Cooperation of IBM Montpellier, KIT, and UJF as master's thesis project. - Detailed performance analysis and modeling on PPC, Cell, GPUs.
Summer 2009 (10 weeks)	**Internship**, *Technology Development, SAP AG*, Walldorf, Germany. Multicore Hash-Tables: Evaluation of different approaches for the computation of aggregation functions in an in-memory database.
2007–2008 (11 months)	**Student Assistant**, *Institut für Fördertechnik und Logistiksysteme*, Karlsruhe Institute of Technology, Germany.

Trainings

February 2015	**Winter School**, Database Implementation Techniques, 5 days, *Schloss Dagstuhl*, Germany.
September 2014	**Summer School**, Algorithm Engineering, 2 days, Bad Herrenalb, Germany.
February 2012	**Graduate Seminar**, Motivating and Efficient Supervision of Thesis Students, 2 days, *Hochschuldidaktikzentrum Baden-Württemberg*, Karlsruhe, Germany.

Language Proficiency

German	**native**
French	**bilingual** (DELF B2 in 2008, now estimated C2)
English	**fluent** (TOEFL in 2008: 111/120, now estimated C2)
Spanish	**advanced** (estimated C1)
Catalan	**beginner** (estimated A2)

Extracurricular Activities

since 2011 **AEGEE:** Various positions in pan-european student organization *AEGEE*:

- Elected Speaker of European-level Information Technology Committee (1 year),
- Delegate for European-level General Assembly (elected three times),
- Vice-President of local branch (1 year),
- IT responsible for local branch (4 years).

since 2007 **ThunderBirthDay**: Design, implementation, and maintenance of an add-on for Mozilla Thunderbird (500'000 downloads, 23 community translations).

2010–2011 Social project with street kids and travelling in Peru (5 months).

2008–2010 Class representative at Grenoble INP–Ensimag in pedagogic conferences.

References

Prof. Dr. rer. nat. Peter Sanders

Full Professor at *Karlsruhe Institute of Technology*, Karlsruhe, Germany
sanders@kit.edu

Prof. Dr.-Ing. Wolfgang Lehner

Full Professor at *Technical University Dresden*, Dresden, Germany
wolfgang.lehner@tu-dresden.de

Dr. Norman May

Senior Developer at *HANA Platform Core Department, SAP SE*, Walldorf, Germany
norman.may@sap.com

Dr. h.c. Franz Färber

Chief Development Architect at *HANA Platform & Database Department, SAP SE*, Walldorf, Germany
franz.faerber@sap.com

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LYX:

http://code.google.com/p/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Final Version as of July 7, 2016 (`classicthesis`).