Design and Implementation of RDMA-based Database Operators

Master Thesis

Author(s):
Willi, Roman A.

Publication date:
2017-12-21

Permanent link:
https://doi.org/10.3929/ethz-b-000223924

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 181

Systems Group, Department of Computer Science, ETH Zurich

Design and Implementation of RDMA-based Database Operators

by

Roman Willi

Supervised by

Claude Barthels,
Dr. Ingo Müller,
Prof. Dr. Gustavo Alonso

June 2017–December 2017

inf | Informatik
Computer Science

The journey is the destination.

— Confucius

To my faithful companions.

Abstract

In this thesis, we present a novel, extensible RDMA database operator interface — where RDMA is not simply the acronym for Remote Direct Memory Accesses – in addition it also stands for **R**eusable, **D**istributed **MA**in-memory database operator interface.

The interface is designed and implemented for distributed query pipelines scaling up algorithms to many thousand cores. It provides the usability of SQL, combined with the expressiveness and extensibility of Spark and will eventually achieve the performance of hand-tuned algorithms written in C++. We implement a distributed radix hash join algorithm and query 1 of the TPC-H Benchmark with the building blocks provided by the operator abstraction. While the former is intended to compare the performance to a hand-tuned, highly optimized implementation, the latter shows the expressiveness of the interface.

Although the radix hash join implemented on top of the operator interface exhibits less performance, it has a similar sub-linear scaling behaviour. In the discussion section, we show the differences in the implementations and that the operator interface simply needs more fine-tuning to match similar performance or even surpass it.

Keywords: operator interface, extensible, reusable, RDMA, main-memory database, micro-operator

Acknowledgments

First of all, I would like to thank my supervisors Claude Barthels and Dr. Ingo Müller for their great support and guidance. Their constructive feedback, critical discussions, and intellectual brilliance were a valuable contribution to my work. I also want to thank Professor Dr. Gustavo Alonso for offering me the opportunity to write my masters thesis at the Systems Group at ETH.

Furthermore, I want to express my gratitude towards my family. To my parents, who made my studies at ETH feasible and provided me the best possible environment. And to Claudia, my girlfriend of many years, who always supported me to the fullest and not seldom put her own interests last.

Last but not least, my thanks go to my friends from the university, the VIS, and beyond for their ongoing moral support during my entire studies at ETH. Lukas Bischofberger deserves a special mention and thanks for proofreading this thesis.

Nomenclature

Acronyms and Abbrevations

ASIC	Application-specific integrated circuit
DBMS	Database Management System
DMAPP	Distributed Memory Application API
GCC	GNU Compiler Collection
HPC	High Performance Computing
HTAP	Hybrid transactional/analytical processing
LLVM	This is not and acronym, look it up in the Glossary
MPI	Message Passing Interface
NIC	Network Interconnect
OLAP	Online Analitical Processing
OLTP	Online Transaction Processing
RDMA	Remote Direct Memory Access
RMA	Remot Memory Access
SIMD	Single Instruction, Multiple Data
SLA	Service Level Agreement
SQL	Structured Query Language

Glossary

C++ template mechanism	Meta-programming, achieved for type-safe containers and generic programming. The C++ template mechanism is itself Turing-complete
ColumnRelation	column-oriented storage model
Driver of a pipeline	Operator at the end of a pipeline that controls the execution
Exabyte	10^6 Gigabyte
InfiniBand	Computer-networking communication standard for HPC
LLVM Compiler	Collection of modular and reusable compiler and toolchain technologies and LLVM is not an acronym itself
Micro-Operator	Smallest possible amount of functionality encapsulated inside a building block of the interface
Operator	Whenever used we mean micro-operator
RowRelation	row-oriented storage model
TPC-CH Benchmark	OLTP Benchmark for multiple transaction types
TPC-H Benchmark	Decision support benchmark for ad-hoc queries

Contents

List of Tables

List of Figures

List of Lists

1 Introduction

1.1 Motivation

In recent years, the amount of data generated and stored all over the world across any economic sector has grown exponentially by virtue of the omnipresent Internet. According to some sources [1, 2], 90% of the worlds data has been created in the last 2 years and on every day in 2015, 2.5 Exabytes of new data were created. This number has increased ever since, and will explode in the near future once the Internet of Things (IoT) really takes off.

Storing such enormous amounts of data is one problem, querying another. By querying we refer to efficiently extract meaningful information – which is summarized under the term *business intelligence* – and includes real-time analytics, machine learning applications, and decision support systems together with traditional data warehouse workloads. All those applications have a crucial demand to efficiently process complex queries over huge sets of data.

Traditional database management systems running on single cores are not able to meet those requirements anymore. Therefore, complex systems, composed of rack-scale clusters or HPC scale computing systems connected by high-throughput, low-latency networks, have been engineered. Those systems reduce vertical data movement by keeping almost all data in the processor caches and main memory. Simultaneously the adoption and optimization of existing algorithms towards the new hardware architecture started. Writing

hand-tuned code for distributed algorithms, designed for massive scale, is tedious and requires deep engineering expertise beyond system and algorithmic topics. Thus, we belief that a layer of abstraction should be added that provides expressiveness, ease of use and high performance simultaneously. Proposed solutions were announced as experimental research like Volcano [3] or translated the improvements in expressiveness and ease of use to a major set-back in performance like Apache Hadoop [4]. Even the 100 times faster Apache Spark [5] does not reach the bare-metal speed, which we show, that we can achieve it for some of the operators.

Our goal is to design and engineer a new operator interface that is easily extensible, reusable, and provides functionality for high-bandwidth, low-latency networks. We plan to encapsulate those functionality inside *micro-operators* to scale queries with the number of machines and to facilitate the development of distributed query pipelines in the future.

1.2 Challenges

Designing an expressive, expandable operator interface to provide *micro-operators* as building blocks is challenging for multiple reason. Starting with the preceding expressiveness: First the interface must be applicable in a high number of specific cases, which before were covered by specialized hand-tuned code. Second there should be few if any restrictions on the data model that can be queried.

Additionally the operator interface should not only be reusable, but extensible in a way that the development of new *micro-operators* requires low effort and does not need system-wide prowess.

The aforementioned hand-tuned algorithms, written by highly-skilled experts, lead us to the third challenge: performance. Efficiency and performance have always been among the main objectives. Balancing trade-offs between expressiveness, extensibility while remaining highly performant is the true art.

Other challenges are posed by the hardware complexity, the distributed environment, and the orchestration of the single operators and query pipelines. It requires expert knowledge to achieve the peak performance. In addition, it is not always clear how zero-copy mechanisms or the interleaving of communication and computation can be optimally exploited.

1.3 Contributions

This thesis provides a design and implementation of an extensible, reusable operator interface, which solves the former mentioned expressiveness, ease of use, and extensibility. In addition we present several performance benchmarks for the designed *micro-operators* in isolation and stacked into pipelines. The comparison to a distributed, hand-tuned radix hash join executed on a cluster of several thousand cores yields a 12 fold increase in execution time. This comparison reveals the major drawbacks of the operator interface that obviously still lacks the proper fine-tuning. A thorough analysis provides a detailed outline to alleviate the discovered overheads. Solving them will position the proposed operator abstraction as a front-runner among solutions to a core mismatch in current high-performance computing, distributed database management systems, and many other research and non-research areas.

1.4 Structure of the Report

The current chapter of the report will state the motivation, challenges and contributions of this thesis. In Chapter 2, we address the necessary background knowledge and state of research that covers different operator models introduced by the research.

The main work is presented in Chapter 3. We outline our design philosophy on the operator model, which were the foundations for the development of the operator interface and its internal auxiliary resources.

Chapter 4 provides various performance benchmarks for the operators in isolation and as combined, complex pipelines. To emphasise the re-usability, composability, and expressiveness we implemented a distributed radix hash join algorithm and query 1 from the TPC-H Benchmark.

In Chapter 5, we critically review the design decisions and implementation of the provided operator interface to reveal its current shortcomings and limitations as well as its huge potential. The discussed limitations directly state the next imminent work items.

The related work to this thesis is presented in Chapter 6 and puts our operator abstraction into perspective.

The final Chapter 7 shortly summarizes the contributions of this thesis and outlines possible directions for the broader future work once the task of the former chapter are addressed.

2

Background

In this section, we provide the necessary information and understanding on central topics of the following thesis. We start by revisiting several operator models proposed in research and their limitations. We will cover technologies used in high-performance computing systems to scale up algorithms to several thousand cores. Finally, we discuss distributed join algorithms, a state-of-the-art distributed radix hash join and a sort-merge join algorithm implemented for running at huge scales.

2.1 Different Operator Models

There are three different models that are widely used in query processing systems to control dataflow and parallelize query execution. The three main models to introduce parallelism in database query execution are (1) the bracket model [6, 7], (2) the iterator model [8], and (3) a push-based model [9], which are described in the following sections. The iterator model is often referred to as Volcano Model and incorporates a demand-driven dataflow scheme also known as pull-based.

The bracket model has mainly been used in systems as Bubby [6] and Gamma [7]. A simple operator model can be found in R* [10]. The most prominently and sophisticated example is the Volcano [3, 11, 12] system itself.

We look at push-based models as well, since we can think of several algorithms or procedures like partition algorithms or operations across machine boundaries that are favoured by a data-driven dataflow or can be easily scheduled by both dataflow paradigms.

2.1.1 Bracket Model

In the **bracket model**, a generic template process is used. This template process is wrapped around the actual query processing algorithm and controls its execution. The template has predefined in- and outputs and invokes the underlying operator, which does the actual work. By this means, a variety of algorithms can be executed. The downside is that only exactly one operator at a point of time can be scheduled. The bracket model, which has been successfully integrated in distributed-memory relational prototypes, has two main drawbacks: extensibility and performance.

First, when a system is extended with a new algorithm or parallel execution strategy, both software and modules must be modified because the semantics and execution strategies of an algorithm are located in two separate places. This is extremely unsuitable for extensible DBMSs.

Second, the operators are implemented in such a way that they obtain input and deliver output by expensive network I/O, therefore passing a data item between operators requires inter-process communication (IPC) system calls. Both, network I/O and IPC calls, result in undesired performance overheads, which theoretically could be removed in situations where a query is evaluated on a single machine or no data repartitioning is needed (Section 4.2 Join Queries [13]). Both limitations have been solved in the operator model proposed by Volcano.

2.1.2 Volcano

Volcano [3, 11, 12] was designed as a new dataflow query processing system for database systems research. It provides a uniform interface between algebra operators, which enables good extensibility and in combination with a specialized *exchange operator*, efficient parallelism.

The system was designed using the *operator model* described in this section with a special *exchange operator*, which encapsulates the logic for parallelism. It permits intra-operator parallelism on partitioned datasets and both vertical and horizontal inter-operator parallelism. This approach makes the implementation of parallel database algorithms significantly easier and more robust.

The **operator model** relies on anonymous inputs or streams. An operator has no knowledge about the operator that produced its input. This is achieved by a uniform interface: *Open-Next-Close*, where *Open* initializes and sets-up all necessary inputs and local data structures, *Next* does the actual work by producing output, and *Close* "shuts

down" everything again. Queries are executed by a demand-driven dataflow also known as pull-based approach. All issues of control regarding parallelism are localized in one operator that uses and provides the former standard iterator interface of the operators above and below in a query tree.

The exchange module is an operator relying on the aforementioned interface. Its design goal was to parallelize all existing query processing algorithms without modifying their implementation and to provide extensibility for future algorithms. An *exchange operator* can be inserted into a complex query tree anywhere suitable to achieve i) a pipeline between different processes, ii) parallel execution of different subtrees, or iii) parallel execution of the same operation on distinct subsets of a dataset. The query execution engine should only provide mechanisms for different parallelization policies and the query optimizer should incorporate and decide on those policies.

We discover several problems regarding the state-of-the-art system architectures of today's in-memory DBMSs. First, in Volcano essentially all operators have the ability to partition their output by means of support functions. We believe that a specialized operator is more suitable for the in-memory setup we design for since it requires heavy tuning. Second, the *exchange operator* only handles exactly one format of *blocks*. We envision to support multiple formats like row or column stores or compressed tuples. Third, regarding control to decide how much work is done, in a specific scenario where the *filter operator* does find a matching tuple, it reads its entire input.

2.1.3 Push-based Model

The previously mentioned iterator model introduces several drawbacks like poor code locality and complex bookkeeping. Those limitations are a product of the tuple-stream oriented processing, which results in millions of rather expensive calls – virtual or via function pointer – to the *Next* method.

An obvious strategy suggests producing more than one tuple during a *Next* call or even producing all tuples. This block processing proposed by Padmanabhan et al. [14] is used to amortize the calling costs and is used in SharedDB [15] and BatchDB [16] as well. However by using this approach the main advantage of the "pipeline" is lost. Pipelining allows an operator to pass data to his downstream operator without copying or otherwise materializing (writing to memory) it. This drawback results in higher consumption in memory bandwidth. The sole advantage generated by the materialization is the possibility to use vectorized operations [17].

Neumann [9] proposes a new, different query compilation strategy to leverage aforementioned problems that are part of HyPer [18]:

1. Push-based operators to achieve better code and data locality.
2. Data centric processing, tuples kept in CPU registers as long as possible (including blurring of operation boundaries).
3. Compilation into native machine code using the optimizing LLVM compiler framework.

The query compiler is designed to maximize locality in both data and code by introducing *pipeline breakers*: An algebraic operator is a *pipeline breaker* for a given input side if it takes an incoming tuple out of the CPU registers. It is a *full pipeline breaker* if it materializes all incoming tuples from this side before continuing processing. Spilling data to memory is considered as a pipeline breaking operation. The data control flow is *reversed* to a push-based approach. Therefore data is always pushed until it reaches a pipeline breaker and hence it is pushed from one pipeline breaker into another.

The data-centric compilation of algebraic expressions allows to generate near-optimal assembly code and to keep all relevant values in CPU registers. Processing multiple tuples at a time allows to use SIMD instructions and helps delay branching. SIMD instructions are a kind of inter-tuple parallelism, i.e., processing multiple tuples with one instruction.

The performance for several OLTP and OLAP workloads has been evaluated on the HyPer [18] system once compiled by C++ and once generated through LLVM. The former has only shown small differences among TPC-C transaction run times, but resulted in up to ten times faster compilation for LLVM. The latter revealed 2-4 times faster performance for TPC-CH queries while using LLVM.

Inspecting the code quality, the LLVM code contained far less branches than MonetDB [19, 20] and the number of branch mispredictions was significantly lower. The LLVM code showed in addition better data locality, more compactness, and less cache misses – up to a factor of ten.

2.1.4 Overview

In this section we provide a short overview in Table 2.1 of the features supported by each approach and points taken for our own design.

	Bracket Model	Volcano	Push-based	Points taken
demand-driven	✗	✓	✓	✓
block-wise processing	✗	✗	✓	✓
pipelining	✗	✓	✓	✓
uniform interface	✓	✓	✓	✓
efficient parallelism	✓	✓	✓	✓
extensible	✗	✓	✓	✓
reusable	✓	✓	✓	✓
scheduling	✓	✓	✓	✓
oblivious to underlying storage	✗	✗	(✗)	✓

Table 2.1: Overview of features grouped by different approaches.

2.2 RDMA and RMA

High-performance computing systems composed of several multi-core machines often employ high-throughput, low-latency networks such as InfiniBand [21] or Cray Aries [22] in order to efficiently transmit data. These kind of networks offer Remote Direct Memory Access (RDMA) to provide efficient inter-machine data movement. These light-weight communication mechanisms bypass the kernel and avoid intermediate copies and therefore save up CPU cycles for other processing tasks. Large data transfers, therefore benefit from RDMA, which can hide network latency and reduce some of the bottlenecks that arise when scaling out.

However, access to remote memory is slower than to the local memory, therefore, communication and computation must be interleaved to hide the network latency. This fact demands thoughtfully designed algorithms that are aware of machine boundaries and expose communication patterns that allow interleaving of communication. In addition, an efficient buffer management is a must for high performance as well. This can be achieved by registering memory for RDMA-enabled buffers prior to execution and reuse them subsequently as shown by Frey et al. [23].

RDMA offers two programming abstractions to choose from: (1) send & receive (two-sided) or (2) read & write (one-sided) operations which both offer similar performance on most of today's networks. To complete a data transfer in two-sided operations or channel-semantics, both endpoints need to be active. The receiver provides several RDMA-enabled buffers where the network card writes the packages from the sender into. The sender does

not know exactly the locations of these buffers. For one-sided operations the initiator controls where the data of the request will be placed. Those read and write operations are executed without interaction of the other node and represent the *remote memory access* (RMA) semantics. Due to the involvement of the receiver in those two abstractions, in two-sided operations it is called *active target* and in one-sided operations it is referred to as a *passive target*.

Memory accessed through RDMA operations usually has to be registered that the network card and pinned thereafter such that it cannot be swapped out. For one-sided operations, it is necessary that the reader or writer holds the required access information to the data.

We design, implement, and evaluate our operator abstraction on a hybrid Cray XC40/ XC50 supercomputer [24] that features many thousand cores and RMA support over a high-bandwidth, low-diameter network topology.

2.3 Message Passing Interface

The Message Passing Interface (MPI) [25] provides a standardized high-level interface that enables writing portable distributed applications involving RMA operations. It is widely used in high-performance computing systems and has spread to other areas of computer science. MPI has been chosen as the communication layer for multiple data processing systems and distributed database systems [26, 27] because of its rich hardware-independent interface and its responsibility to select the most appropriate communication method for each pair of processes. MPI is only the interface abstraction and the actual implementation can change. Examples are OpenMPI [28], MPICH [29], MVAPICH [30], and foMPI [31].

The needs in HPC and DBMS applications are however different. In HPC the workload and communication patterns are usually a priori known, whereas the workload in distributed database systems is more dynamic and not known in advance. This results in different bottlenecks for those applications. In distributed database systems the network is often the bottleneck, because vast amounts of data have to be shuffled among nodes. To efficiently handle the dynamic workloads in DBMSs a thread per machine can be pinned entirely to the network operations. In HPC this approach is not necessary, because scheduling and resource allocation decisions can be made statically.

OpenMPI provides a full implementation for MPI, for both, the two-sided and one-sided abstractions. There exist several other, specialized partial implementations for MPI. An

example of a more efficient, open-source implementation for only RMA protocols is Fast One-sided MPI (foMPI) [31]. It is only an implementation of the one-sided operations of MPI-3 and is highly optimized to operator on Distributed Memory Application API (DMAPP) and XP-MEM. Within the same computing node it interacts with the kernel module XP-MEM. This module allows for intra-node communication, to map the memory of one process into the virtual address space of another. Communication to a remote node is accomplished by interfacing with the DMAPP.

We use the most basic point-to-point communication mechanism offered by MPI, which are blocking/synchronous operations:

- `MPI_Send`: Function to send an amount of data from a specified buffer to a dedicated destination process. The amount of data to send is defined by a count and the datatype of each buffer element. The communication happens over a predefined communicator and a message can be tagged.
- `MPI_Receive`: Used to receive a message sent with a specific tag and communicator. The data is written into a previously allocated buffer. Each *MPI_Send* function must be matched by a call to *MPI_Receive*.
- `MPI_Probe`: Allows to check for incoming messages for a given tag and source node without actually receiving it. It updates an *MPI_Status* struct.
- `MPI_Get_count`: Determine the amount of received elements out of the *MPI_Status* struct. This method is used to allocate a dynamic buffer to receive a message into.

We implement operators to provide two-sided communication mechanism as building blocks in a first phase, because it is fairly straight forward. In a second phase additional operators with one-sided communication mechanism based on foMPI [31] should be added as well.

2.4 Distributed Join Algorithms

Different join algorithms like no partitioning join, the shared partitioning join, the independent partitioning join, the sort-merge join, and the radix join have been studied by research in different hardware and network set-ups. We will look at the radix hash join and the sort-merge join.

2.4.1 Radix Hash Join

We dedicate this section to a distributed, hardware-conscious main-memory hash join, implemented and benchmarked by Barthels et al. [32, 33] proposing modified partitioning, build, and probe phases. In previous work conducted by Balkesen et al. [34, 35], the authors have shown that a carefully tuned radix hash join is able to outperform other join algorithms on multi-core machines.

The algorithm features two stages that can be executed in parallel and offers a multi-pass partitioning scheme to prevent excessive cache misses and misses in the translation lookaside buffer (TLB). Thus, it offers good characteristics to be used in a distributed environment. The partitioning mechanism, in addition, enables to generate a large number of partitions to prevent multi-core machines of becoming idle.

A preprocessing stage determines the size of RDMA-enabled network buffers that need to be allocated and the machine-to-partition assignment for each node, done in a round-robin scheme or dynamically for skewed input data. The required information is gathered from a global histogram that is computed in three steps: (1) building a local histogram for each thread, (2) merge these into a machine-level histogram, and (3) combining them into the global histogram.

In the first actual stage, the radix hash join algorithm partitions the input relations into disjoint partitions based on their join attribute. The resulting smaller partitions fit into the cache of each CPU core. This is important for the actual performance of the join task, which benefits from hash tables that fit into the cache and therefore provides a low cache miss rate [36]. The partitioning is performed in p passes with different hash functions that take different bits into account and limit the number of simultaneous partitions such that the number of cache lines or TLB entries is not exceeded. In the second stage a hash table over each partition of the inner relation is built and in the following probed by data of the corresponding partitions of the outer relation. Additionally, the non-overlapping partitions of the input relations allow a high degree of parallelism, because they can be assigned to different worker threads.

It is worth to have a closer look at the multi-pass partitioning stage consisting of a network partitioning phase and a local partitioning phase. This phases ensure that enough partitions are generated to assign at least one partition to each core ($\#partitions >= \#total\ CPU\ cores$). In the network partitioning at least two RDMA-enabled buffers are assigned per thread to allow continuation of processing during network operations. Thus, they achieve the aforementioned parallel execution through overlapping of the phases.

Buffer registration overhead is hidden by reusing buffers that are preallocated and drawn from a buffer pool. No synchronization is required since all the buffers are private to each thread. Regarding the amount of available memory, it is meaningful to choose between one-sided or two-sided RDMA operations. If the amount of main memory is large, one can allocate large RDMA-enabled buffers for each partition and each remote machine while using one-sided operations. Otherwise, small buffers for the RDMA operations together with two-sided operations are used. In addition, another larger non-RDMA buffer is allocated to copy the received data. The local partitioning pass creates cache-sized partitions and does not involve network operations anymore. These are the final partitions used in the build and probe phase.

The radix hash join attracts a great deal of attention in research, is highly performant, can be parallelized, and exhibits good scaling behaviour. Therefore, it suits well as an example to be implemented on top of the operator interface.

2.4.2 Sort-Merge Join

Barthels et al. [33, Sort-Merge Join] implemented a sort-merge join in order to compare the performance metrics of the radix hash join algorithm. Their results have confirmed recent studies [34] that the radix hash join outperforms the sort merge join algorithm – in distributed environments, too.

On a high-level abstraction, the distributed sort-merge join can be described as follows: i) A first phase partitions, sorts, and transmits the partitions over the network, which allows parallel execution of sorting a partition and network operations, ii) after receiving, several passes, which perform *m-way* merging of multiple sorted runs (producing itself a single sorted output), are scheduled until the inner and outer relation are fully sorted, and iii) in a final pass the two sorted relations are joined on a partition granularity.

To demonstrate the expressiveness of our interface, this algorithm is another good example that should be implemented.

3

Interface Design and Implementation

In the following we provide detailed insights to the approach taken and decisions that led to the current system design. We discuss several design decisions and the implications they have on the overall system. There are multiple options or solutions that each have their own advantages and disadvantages in comparison to our vision. Each of the taken solutions itself proposes new challenges which have to be solved.

The chapter is divided into the presentation of our operator model, the logical data model our system operates on, the internal auxiliary structures, and the resulting interface itself. The final section describes the new operator interface in depth and provides an overview of explicitly implemented *micro-operators* using the provided building blocks.

3.1 Design Philosophy

Our impression, from the discussed operator models of the previous Section 2.1, is that the pull-based approach fits nicely the concept of a pipeline and pull- or push-based approaches are fairly similar or even interchangeable. We therefore go for a volcano-like, demand-driven operator interface.

Similar to Volcano, we design and implement a uniform interface between operators for dataflow query processing, designed as anonymous inputs or streams. Therefore operators

can easily be reused and connected with a high degree of freedom to form complex query pipelines. We also strive for extensibility, meaning that our operator interface should be general enough that in the future a variety of operators can be implemented on top of it. Our philosophy is that an operator or micro-operator should provide a minimal single functionality, in contrast to commonly known database operators, which i.e. implement a complete hash join. Therefore we use the terms operator and micro-operator interchangeably and express each time our notion of an operator. Scheduling decisions shall not be coded inside the operator itself but rather be made by an independent component. Ideally we mitigate or even eliminate the discovered weaknesses of the different operator models discussed in Section 2.1 by an ingenious design or sophisticated implementation.

3.2 Data Model: Tuple & Relations

The data model of a system is the crucial component that sets several limitations and defines its expressiveness. Therefore the fewer restrictions the better. With this in mind we define, in the database context, the logical Tuple and Relations. The Tuple is defined as following:

Definition 3.2.1.
$tuple := < attribute_0,\ ...,\ attribute_n >$
$attribute := atomic\ value \mid tuple$

To keep full flexibility and generality there are no restrictions on the actual key or attribute types nor on their nesting/recursion. This allows a key, e.g., to be a tuple itself. Tuples are defined to be one of the intermediate representations of passing values in our system. The utility library of C++ provides since C++11 an exact implementation of the formerly described properties called *std::tuple* [37]. Therefore, a Tuple in our logical model is in fact implemented and represented in the form of a *std::tuple*. Those tuples are isomorphic to flat tuples and therefore we can pass them through registers.

Our operator interface is completely oblivious to the storage model achieved through different scan operators. We currently implemented two distinct but very common relation models: row-oriented and column-oriented relations. The former is in the following referred to as *RowRelation* and the latter as *ColumnRelation*. By providing both storage formats one can increase query performance for very large data sets by organizing data in the appropriate format, which allows precise access of the desired data but only the desired data. We currently designed operators to handle former relation models. However, we see

no limitations in introducing additional data models such as document stores along with appropriate (scan) operators.

The aforementioned flexibility is achieved by the C++ template mechanism, which is Turing-complete. C++ Templates allow polymorphism that occurs during compilation through instantiation of the templates and function overloading resolution. Each instantiation of a function template with a different template parameter results in a different function to be called. This technique is heavily used throughout our entire system because it shifts work done at runtime to compile time and we achieve exact type information on our tuples, relations, and operators at any point during execution in our program.

3.3 Query Pipeline

We strive for a uniform operator interface such that different operators can be freely chained together into a complex query pipeline. As presented in Figure 3.5, the start of a pipeline is marked by reading its input, whereas the output is an intermediate or the final query result. Operators of a query pipeline exchange one tuple at a time over the exposed interface methods. The last operator in a pipeline has the control over the dataflow and is responsible for signalling its upstream operator(s) when to produce a next tuple. An operator can successively demand its upstream operator to produce a next tuple until the input stream is completely processed or exhausted.

Invoking a new function call, virtual or via a function pointer, for every tuple in the input relation might seem like an overhead. This overhead becomes even more critical since those calls are performed by every operator in a pipeline. The shortcomings manifest in reduced code and data locality, stack allocations/deallocations, and cache-inefficiency.

Neumann [9] extensively investigated those issues in combination with branch prediction on modern CPUs. He proposes data-centric query processing, which organizes the control and data flow in such a way that data is kept in registers as long as possible to achieve excellent data locality. In addition tuples should directly be pushed from one pipeline breaker into another. We believe that our design delivers those solutions out of the box in combination with today's powerful compilers. All operators inside a query pipeline are heavily inlined such that the compile can optimize the code up to a very high degree. Figure 3.1 specifies what we understand under the term inlining. First, each operator gets a copy of its upstream operator inside its *body*. By this approach every pipeline breaker – the last operator in a pipeline – has knowledge of all operators of his

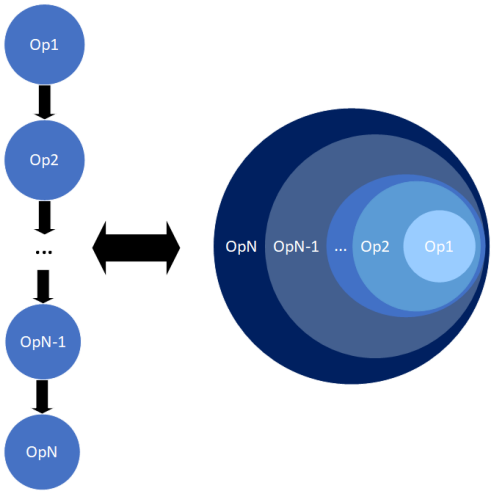


Figure 3.1: Operators of a query pipeline are heavily inlined to gain performance.

pipeline and therefore the compiler has all the required information for targeted optimization. Second, on an implementation layer, all exposed member functions are annotated by the *__attribute__((always_inline))* of GCC [38] which tells the compiler to inline functions. The *TransformOperator* 3.8.1 shows this optimization nicely.

However, while implementing several operators by means of the operator interface, we discover that those former techniques are not sufficient and applicable in every single case. As a solution, we adopt another topic discussed by Neumann: block-wise processing investigated by Padmanabhan et al. [14], which we discuss in the next section.

3.4 Pipeline Instantiation

In Section 3.3, we describe on a high level, how operators are stacked inside a pipeline and their actual types are propagated. Listing 3.1 presents a possible implementation example that adds a constant value to the second element of each tuple. First, a scan operator and a transformation function are instantiated in Lines 1 and 3, which both are passed to a transform operator. However, one immediately detects the *verboseness* of the code and gets a feeling that this approach becomes cumbersome for complex pipelines. The construction of the operators in Line 2 and 5 require template arguments for their constructors, because every template argument must be known, to instantiate function/class templates (the mechanism used to achieve generic programs). Trailing template-arguments can be left unspecified, however not in constructors (will first appear in the *c++17* standard). In lower versions therefore a special delegation through *make_** functions do the trick as

shown in Listing 3.2. In a stage called template argument deduction the compiler tries to deduce such missing template arguments that are not specified by the programmer. The corresponding template arguments are in the following replaced by the template arguments which have been specified, deduced or obtained from default template arguments.

```
1 ColumnRelationScanOperator< uint64_t , uint64_t>  s =
2         ColumnRelationScanOperator< uint64_t , uint64_t> (relation) ;
3 auto func = AddConstantToValueTransformation< 1 , 9001> () ;
4 TransformOperator< decltype(s) , decltype(func)>  t =
5         TransformOperator< decltype(s) , decltype(func)> (s , func) ;
```

Listing 3.1: Cumbersome instantiation of operators in the pipeline

In Listings 3.2 and 3.3 we show the helper function and its usage. The aforementioned mechanisms assure that the constructor for the *TransformOperator* in Line 8 of Listing 3.2 knows its exact types, which are deduced in the *make_transform_operator* function from the parameters passed in Lines 2 and 3 of Listing 3.3. This solution, in combination with the *auto* specifier, enables to seamlessly pipeline multiple operators and propagate their actual type to every intermediate operator. For variables declared with the *auto* specifier, their type is automatically deduced from its initializer.

```
1  template < typename UpstreamOperator , typename TransformFunc>
2  TransformOperator< UpstreamOperator , TransformFunc>
3          make_transform_operator(
4                  UpstreamOperator upstream_operator ,
5                  TransformFunc transform_func)
6  {
7      // constructor will be called with exact types
8      return TransformOperator< UpstreamOperator , TransformFunc> (
9                  upstream_operator , transform_func) ;
10 }
```

Listing 3.2: *make_** helper function

```
1 auto s = make_column_relation_scan_operator(relation);
2 auto t = make_transform_operator(s,
3         AddConstantToValueTransformation< 1, 9001> ());
```

Listing 3.3: Nice instantiation of operators

3.5 Memory Management

Our operator interface needs memory management since the native operating system's implementation is a general-purpose allocation mechanism. The OS is not aware of the workload and of how much memory is required per query. Therefore it is usually not optimal. Though, our own memory management is aware of those characteristics and can ensure fair resource utilization and meeting of Service Level Agreements. The major argumentation for our own memory management is the overall performance of queries and thus the system itself.

By requesting and allocating the entire memory needed for the query execution, the database system is able to provide an appropriate, efficient, and extremely individual memory handling. A DBMS usually requires memory in blocks or chunks to store intermediate results or, in the case of in-memory database systems to hold relations and final query results. Efficiently issuing those blocks upon request with minimal overhead is a complex problem, which has been studied exhaustively [39, 40, 41].

We design a simple, yet efficient memory management that relies on a single block size. A large, consecutive block of memory is allocated prior to the query execution and divided into equally-sized smaller chunks. During query execution, the memory manager keeps track of free memory blocks, to efficiently provide new, free memory. Letting the operation system handle this task becomes more expensive once the heap has not enough space left and the kernel has to dynamically enlarge it. Over time the heap, becomes fragmented, which makes the heap management more complex.

Figure 3.2 gives the reader an abstract view of the memory pool and its implementation. The free blocks are internally organized in a linked list to update the **next* pointer without additional computational overhead. We want to point out that this structure is initialized prior to query execution as well. Once a block is requested from the memory pool, the

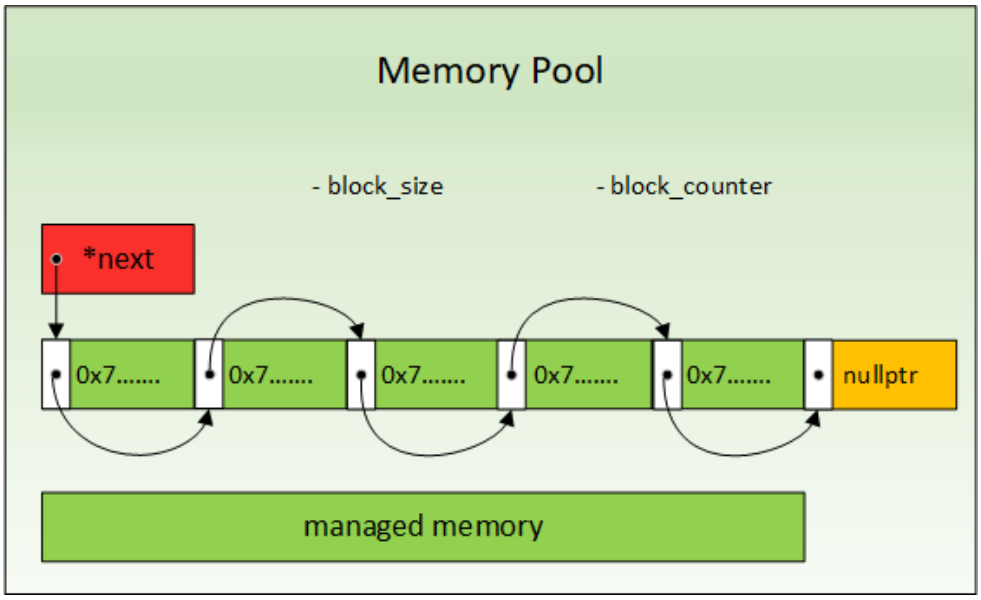
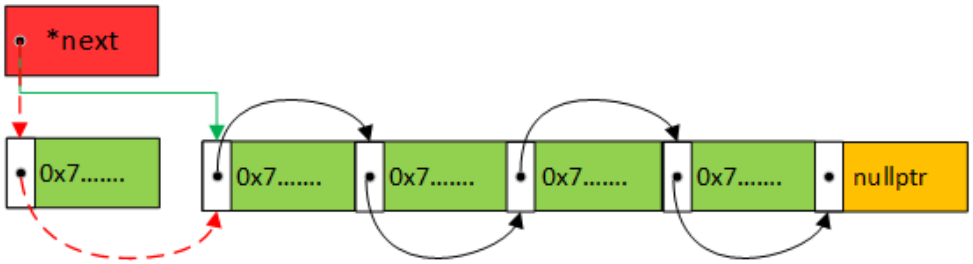


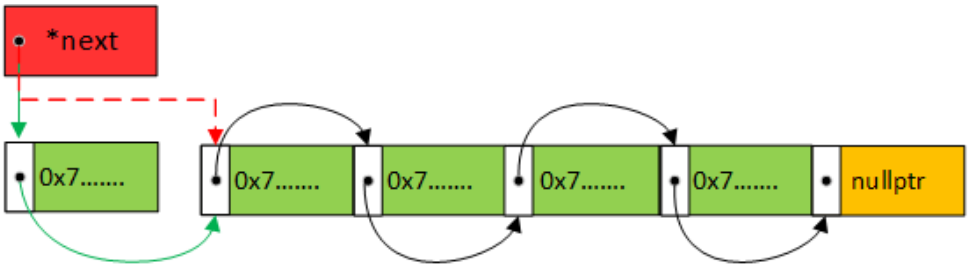
Figure 3.2: Memory management of the operator interface.

**next* pointer is simply updated to point to the next block in line, as shown in Figure 3.3a. This single action is required, since the content of the delivered block is considered to be invalid and the receiver writes to it anyway. In Figure 3.3b we present the reverse action: Memory blocks that are not used any more by the query are returned to the memory manager and inserted into the pool of available memory blocks. Inserting a block into the list of free blocks needs two updates: First, the inserted block has to point to the block at which currently **next* is pointing to. Second, the **next* pointer must be updated to point to the newly inserted block. This allows us to reduce the management overhead of a query by shifting the work of memory allocation, reuse, and de-allocation away from the actual query execution.

To deal with raw memory blocks on the operator level is quite tedious. To alleviate this undesirable necessity, we introduce a wrapper around a memory chunk. The *Block* abstraction shown in Figure 3.4 does not only provide a clean interface for the operators to use but enriches it as well. Through the *Block* design, an operator gains access to specific dynamic run-time information like the current element count and can write elements to the memory location that the *Block* embodies. By this means the operator interface can deal with the memory manager on a high abstraction level, which would even allow to redesign the current memory management.



(a) Removing a block from the list of free blocks.



(b) Inserting a free block.

Figure 3.3: Inserting and removing memory blocks from the pool of free blocks.

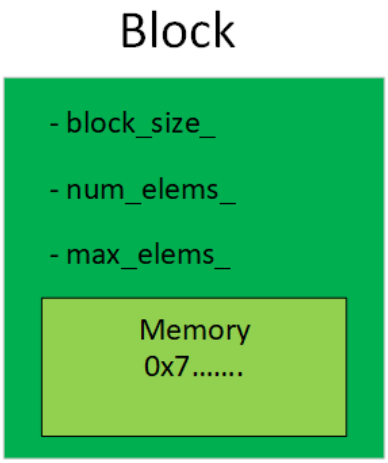


Figure 3.4: *Block abstraction* to provide a clean interface.

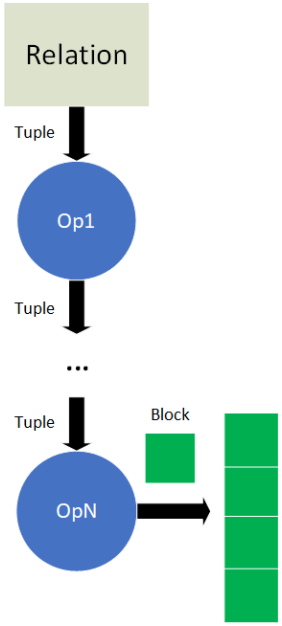


Figure 3.5: Operators chained into a query pipeline, that produce blocks.

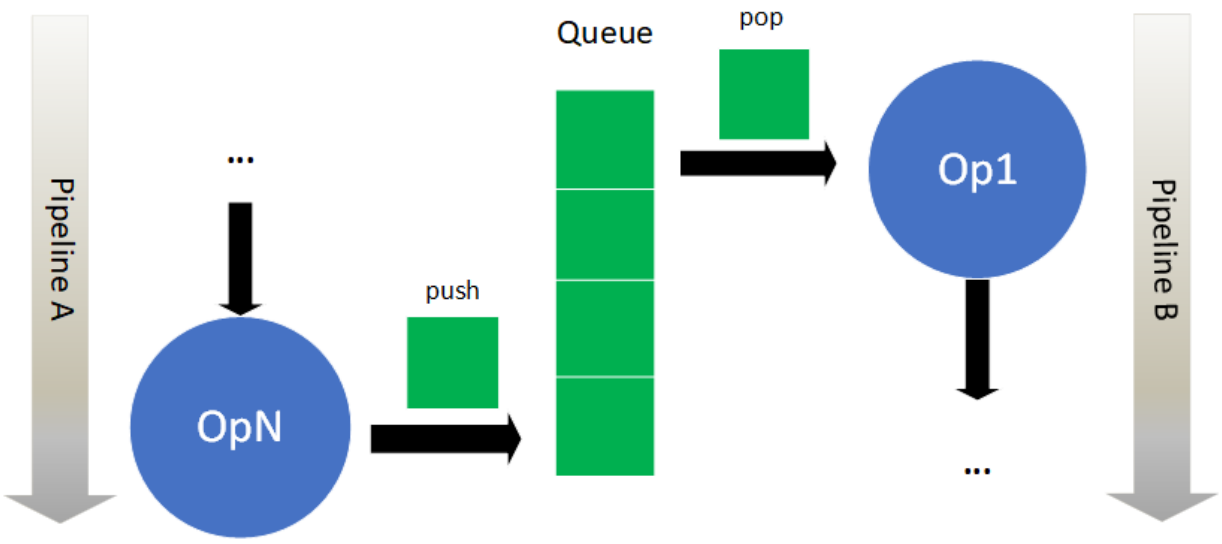


Figure 3.6: Passing *Blocks* among pipelines via the *internal queue*.

3.6 Internal Queue

As powerful as today's compilers and their optimizing mechanisms are, we detect performance issues for passing single tuples in certain operators such as the *MergeOperator* (Section 3.8.2). A possible solution, as already mentioned, is block-wise processing although it is creating additional overhead by memory accesses. Nevertheless, for big enough blocks, this additional memory access pays off and we can alleviate the overhead. The concept of block-wise processing in addition fits in quite nicely with our overall system design and we can reuse our *block abstraction* (Section 3.4) to store tuples.

Producing multiple tuples packed inside a block however is only one concern. The second issue is to efficiently store the blocks and provide access for the next operator. For this purpose, we designed an *internal queue* that holds blocks produced by aforementioned operators. We decide to implement a *dumb* queue in the sense that no additional logic or functionality is embodied other than pushing and popping blocks along with the ability to query for the size and whether the queue is empty or not. We have several ideas to integrate some network functionalities to send/receive blocks over the network directly inside the queue by means of MPI and RDMA. This idea however breaks our initial design goals of implementing all data processing logic inside operators and interchangeability of operators.

Figure 3.6 presents the queue used as means of synchronization between pipelines. Incidentally, this choice adds the possibility for scheduling decisions about which pipeline should be run first or in which order, in case there are no data dependencies among them. On today's multi-core machines the ability to parallelize the computation is mandatory. For this purpose, the input relation has to be partitioned and each partition must be assigned to a separate pipeline. To meet this requirement, we introduce an additional dimension to our queue, which allows to create a separate queue per partition. Currently the fanout is set statically during compile time and each subsequent pipeline must be assigned a partition of the queue. A scheduler could with minimal additional effort dynamically control the actual fanout up to a statically set maximum.

3.7 Operator Interface

Relying on the formerly presented structures, the decisions made in Chapter 3.1, and several re-designing iterations, we present the final operator interface in Figure 3.7. The following sections and Figure 3.8 explain the methods provided by the interface in more depth. We first explain how a tuple is produced by a **TupleProducer** 3.7.1 and consumed by the succeeding **TupleConsumer** 3.7.2. Next we present the production of many tuples packed into a **Block** with the concept of the **BlockProducer** 3.7.3. Finally the **BlockConsumer** 3.7.4 show how a *Block* of tuples is consumed.

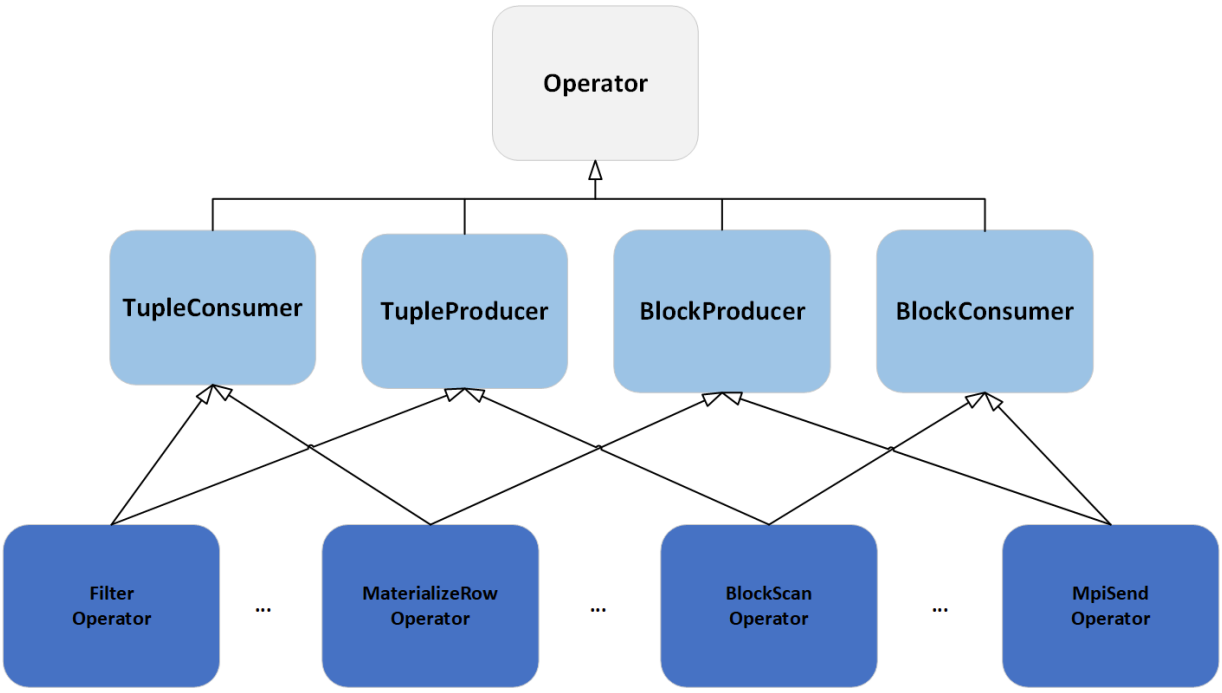


Figure 3.7: Final operator interface along with its core concepts.

Our operator interface represents the conceptual abstractions to solve the aforementioned shortcomings in expressiveness, reuseability, extensibility, and performance. The interface provides flexibility by its four core concepts of which an operator can inherit from. In general, operators realised atop of the provided building blocks make use of one of the consumer and one of the producer interfaces each, as shown in Figure 3.7. Resulting operators can therefore freely be chained together and describe a demand-driven dataflow within a pipeline implemented by means of iterators or streams. The prior discussed chaining of operators results in an additional benefit: Every operator knows the exact types for their upstream operators and tuples provided by the C++ template mechanism. This fact provides type-safety and additional performance gains, since we rely on static binding, which occurs at compile time and no virtual function calls have to be resolved during runtime.

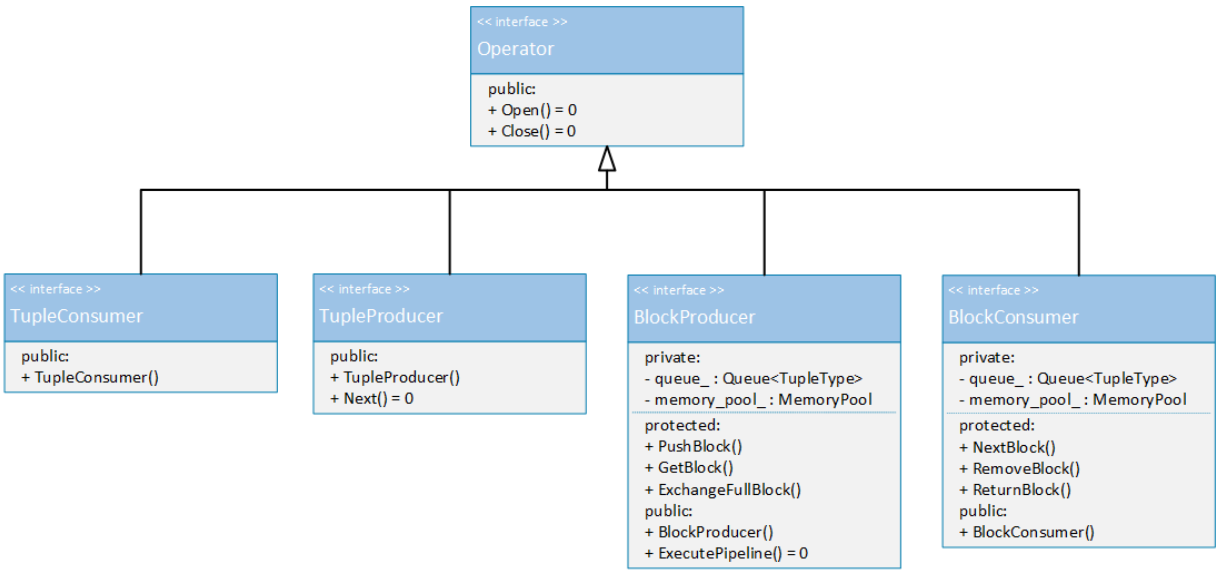


Figure 3.8: Detailed interface overview presenting member functions and fields.

3.7.1 TupleProducer

A first version of the TupleProducer interface exposes the boolean *hasNext()* functions, to indicate whether a new tuple can be produced from the upstream. However we detect a major performance overhead by the lack of inlining and the resulting additional method call. For this reason we remove it and designed a slightly different solution which has the same functionality. The TupleProducer interface therefore provides only a single method *Next()*, which is exposed to his downstream operator and is successively called by it. Operators implementing this interface must provide the logic how a next tuple is produced and return a pair, consisting of the actual tuple and a status flag to indicate whether the returned tuple is valid.

3.7.2 TupleConsumer

This interface exists only as conceptual design as it provides no additional methods or fields. It is basically a no-op and shown for completeness and understanding.

3.7.3 BlockProducer

The class of operators that are unable to pass a tuple at a time, can be categorized by some common properties and are subsumed under the *BlockProducer* concept. While implementing and benchmarking the *MergeOperator* 3.8.2, we detected that operators that

have to keep program state over function boundaries suffer heavily from the additional bookkeeping. The control flow becomes the limitation alongside the branch prediction that is reduced if the tuples are not processed in a tight loop inside those operators.

The solution is to put the control flow into this type of operator. We call them *driver of a pipeline* or simply *driver* in the remainder of the thesis. Block-wise processing addresses former problem and solves it to a high degree. In Section 3.8.2 we closely discuss how the *MergeOperator* influences the design in combination with block-wise processing.

The *BlockProducer* interface provides two sets of functions: First, the method that produces an output *Block* and therefore is the driver of the pipeline. The *ExecutePipeline()* function is responsible for the progress of its upstream operator(s). The second set of functions are designed to interact with its queue to push full *Blocks* and get a new *Block* with free memory from the previously allocated memory pool.

3.7.4 BlockConsumer

As the name suggests, the *BlockConsumer* abstraction operates not on a single tuple but on many of them packed into a *Block*. Therefore, it provides the logic to interact with the queue and the memory pool through protected methods. The *NextBlock()* and *RemoveBlock()* functions fetch a next *Block* from the queue and remove the *Block* from it respectively. *ReturnBlock()* signals the memory pool that a certain *Block* and associated memory is available again and thus can be inserted into the pool of free memory blocks.

3.8 Operator Implementation

This section presents a few sample implementations of building blocks out of many that we believe demonstrate nicely the overall concept, have influenced our design decisions, or are particularly interesting. At the end of this section, we present a table featuring all implemented operators in combination with their classification, used interface building blocks and a short description.

3.8.1 Transform Operator

In Section 3.3 we mention that the ability to inline as many function calls as possible is crucial for the achieved performance. The *TransformOperator* perfectly demonstrates this ability. In Listings 3.4 and 3.5, we compare *C++* code to the intermediate representation of LLVM. The LLVM IR can be used to produce machine code and is easier to read for

humans than assembly code. The *C++* code shows multiple transform operators inside the same pipeline that add each a different constant value to the first attribute of a tuple. *AddConstantToValueTransformation<1,1000>* adds the constant value of 1000 to the second element of the tuple.

```
1  auto s = make_column_relation_scan_operator(relation);
2  // pipeline multiple transform operators that stack up to 1337
3  auto t1000 = make_transform_operator(s,
4                  AddConstantToValueTransformation< 1, 1000> ());
5  auto t1300 = make_transform_operator(t1000,
6                  AddConstantToValueTransformation< 1, 300> ());
7  auto t1330 = make_transform_operator(t1300,
8                  AddConstantToValueTransformation< 1, 30> ());
9  auto t1337 = make_transform_operator(t1330,
10                 AddConstantToValueTransformation< 1, 7> ());
```

Listing 3.4: C++ code for multiple, pipelined *TransformOperators*

All highlighted constants sum up to 1337 and we can show, that in the intermediate representation of LLVM, only the final value of 1337 (Line 16) is used and not all intermediate values. The function attribute in Line 1 and the function name, spanning the Lines 3 to 6 in Listing 3.5, strongly confirm the ability to inline as well. Therefore the compiler is able to perfectly fold the constants and produce optimized and fast machine code.

```
1  ;Function Attrs: alwaysinline uwtable
2  define linkonce_odr void
3  @_long_unreadable_fct_name_
4  AddConstantToValueTransformation
5          ILm1ELm1000EEEEENS6_ILm1ELm300EEEEENS6_
6          ILm1ELm30EEEEENS6_ILm1ELm7EEEE4
7  {
8  ;< some code omitted>
9
10 ;< label> :12:  ;preds = %2
11   %13 = add i64 %4 , 1
12   store i64 %13 , i64* %3 , align 8 , !tbaa !140 , !noalias !309
13   %14 = getelementptr inbounds i64 , i64* %9 , i64 %4
14   %15 = getelementptr inbounds i64 , i64* %11 , i64 %4
15   %16 = load i64 , i64* %15 , align 8 , !tbaa !12 , !noalias !309
16   %17 = add i64 %16 , 1337
17   br label %20
18
19 ;< some code omitted>
20 }
```

Listing 3.5: Corresponding IR in LLVM of code presented in Listing 3.4

3.8.2 Merge Operator

In Listing 1 we present pseudo-code of an optimal solution outside of our interface abstraction. The merge routine is essentially composed out of three loops, which are executed based on the state of the two input relations. The loop in Line 5 is executed until only one relation has tuples left. In the following either the loop in Line 16 or Line 20 are executed, but never both.

For integrating the optimal solution into the interface two functions are required. First, a function to get the input states of the relations. Second, a function that fetches the tuple,

compares them, and returns the matching one. Therefore, in a prior version the interface exposes the *hasNext()* function, to indicate whether a new tuple can be produced from the upstream. The second functionality is provided by *Next()* that returns the proper tuple. Listing 3.6 presents C++ code of aforementioned functions. In comparison to the presented pseudo-code, both functions need to be called to produce a single tuple and the states must be encoded in $left$ and $right$ and kept over the function boundary. This overhead in book-keeping and the additional function calls – which the compiler failed to inline – slowed down the performance.

Algorithm 1 Merge algorithm

```
1: function MERGE(left, right)                    ▷ Merge left and right relation
2:     merged ← new relation
3:     i ← 0
4:     j ← 0
5:     while i < left.size && j < right.size do    ▷ Both relations have tuples
6:         left_tup ← left[i]
7:         right_tup ← right[j]
8:         if left ≤ right then                    ▷ Decide which tuple to insert
9:             merged.insert(left_tup)
10:            i++
11:        else
12:            merged.insert(right_tup)
13:            j++
14:        end if
15:    end while
16:    while i < left.size do                      ▷ Process remaining tuples from left
17:        merged.insert(left[i])
18:        i++
19:    end while
20:    while j < right.size do                     ▷ Process remaining tuples from right
21:        merged.insert(right[j])
22:        j++
23:    end while
24:    return merged                               ▷ Merged result
25: end function
```

```
1  Tuple next_left;
2  Tuple next_right;
3  bool left = false;
4  bool right = false;
5
6  bool INLINE hasNext() override final
7  {
8          // fetch one tuple from each upstream
9          if (!left && left_upstream_operator_-> hasNext()){
10             next_left = left_upstream_operator_-> Next();
11             left = true;
12         }
13         if (!right && right_upstream_operator_-> hasNext()){
14             next_right = right_upstream_operator_-> Next();
15             right = true;
16         }
17         // return upstream states
18         return left || right;
19 }
20
21 Tuple INLINE Next() override final
22 {
23         // both upstreams have a valid tuple
24         if (left && right){
25             // decide which tuple has to be delivered first, and set the flag
26             if (comp_func_(next_left, next_right)){
27                 left = false;
28                 return next_left;
29             }else{
30                 right = false;
31                 return next_right;
32             }
33         }// Once one upstream is exhausted, only pull from the remaining
34         else if (left){
35             left = false;
36             return next_left;
37         }else{
38             assert(right);
39             right = false;
40             return next_right;
41         }
42 }
```

Listing 3.6: C++ code for *hasNext()* and *Next()* methods of the *MergeOperator* prior to the redesign that eliminated *hasNext()*

To improve the performance, we eliminate the call to *hasNext()* by moving the "driver" inside the operator to achieve a tight loop, simpler control flow, and improved branch prediction. In addition we introduce block-wise processing. Listing 3.7 presents the current version of the *MergeOperator* that has more resemblance again to the solution outside of the operator abstraction. Lines 3 and 4 fetch a tuple from each upstream like the corresponding Lines 6 and 7 from Listing 1. Lines 8, 22 and 29 show the three different loops that are executed based on the state of the two upstream operators. From the latter two loops still only one is executed per call to *ExecutePipeline*. The *MergeOperator* inherits from the *BlockProducer* abstraction. Therefore multiple tuples are accumulated in a *Block* and enqueued to the internal queue, once the *Block* is full.

```
1  void INLINE ExecutePipeline() override final
2  {
3          auto left_next_tuple = left_upstream_operator_.Next();
4          auto right_next_tuple = right_upstream_operator_.Next();
5          Block< OutputType>  block = BlockProducer< TupleType> ::GetBlock();
6
7          // is executed as long as both streams have tuples
8          while (left_next_tuple.second == TupleFlag::kValid &&
9                  right_next_tuple.second == TupleFlag::kValid){
10             BlockProducer< TupleType> ::ExchangeFullBlock(partition_id_, &block);
11             // decide which tuple has to be delivered first, and set the flag
12             if (comp_func_(left_next_tuple.first, right_next_tuple.first)){
13                 block.insert(left_next_tuple.first);
14                 left_next_tuple = left_upstream_operator_.Next();
15             }else{
16                 block.insert(right_next_tuple.first);
17                 right_next_tuple = right_upstream_operator_.Next();
18             }
19         }
20         // is executed at the end as long as the left stream has tuples
21         // but not the right stream
22         while (left_next_tuple.second == TupleFlag::kValid){
23             BlockProducer< TupleType> ::ExchangeFullBlock(partition_id_, &block);
24             block.insert(left_next_tuple.first);
25             left_next_tuple = left_upstream_operator_.Next();
26         }
27         // is executed at the end as long as the right stream has
28         // tuples but not the left stream
29         while (right_next_tuple.second == TupleFlag::kValid){
30             BlockProducer< TupleType> ::ExchangeFullBlock(partition_id_, &block);
31             block.insert(right_next_tuple.first);
32             right_next_tuple = right_upstream_operator_.Next();
33         }
34         // push remaining
35         BlockProducer< TupleType> ::PushBlock(partition_id_, std::move(block));
36 }
```

Listing 3.7: Code of the *MergeOperator*'s final version

3.8.3 Partition Operator

The partitioning operator implements a slightly modified, highly optimized, and cache-efficient partitioning routine. It uses software-managed buffers, which were previously implemented in a distributed radix hash join by Barthels et al. [32, 33] and many others. The work is based on findings by Balkesen et al. [34, 35, 42] and Manegold et al. [43]. The general idea is that if tuples are partitioned and written to their corresponding destination partition one-by-one, the *translation lookaside buffer (TLB)* poses an upper limit on the partitioning fan-out. By first buffering the tuples in cache and only writing full cache lines to the destination, the TLB misses become infrequent and eventually out-of-order execution by the CPU can hide the resulting latency.

Our modification to the partitioning routine addresses solely the prior limitations on the tuple size of 64-bit. By using the C++ template mechanisms, we can process tuples with much more freedom. Still three restrictions apply: (1) The size of a single tuple can be at maximum the size of a cacheline, but (2) not smaller than twice the size of an *uint32_t* and (3) the tuple size must be a power of two.

3.8.4 MpiSend & MpiReceive Operators

The pair of the *MpiSendOperator* and *MpiReceiveOperator* implement synchronous send/receive operations of blocks over the network. Therefore, they logically implement the concepts of **BlockProducer** and **BlockConsumer**. However they are special in the sense that the *MpiSendOperator* implements no producer and the *MpiReceiveOperator* no consumer interface since blocks in between those operators are passed over the network rather than into the internal queue. The current versions of the two operators utilizes the synchronous *MPI_Send* and *MPI_Receive* methods of the MPI library [25] for their purpose that simplifies the buffer management of the send buffer. The *MpiSendOperator* can directly send a *Block* taken out of the internal queue and immediately release it, after returning from the MPI call. Releasing a *Block* in this context means returning the memory to the memory pool.

The operator interface handles two different block types – memory blocks and network blocks. The former is used to interact with the internal queue whereas the latter is a buffer dedicated for sending over the network. Network blocks are operator-local to the *MpiSendOperator* and not exposed elsewhere. We decide for simplicity to restrict the block size for network blocks to at most the size of a regular memory block. On the one hand, the decision is based on the fact that the performance overhead between sending always

fully filled blocks to sending eventually partially filled blocks is negligible. In addition, we believe that accumulating several memory blocks to fully fill a network block results in a higher overhead caused by *memcpy* operations. On the other hand, disassembling a larger memory block to fill multiple network blocks requires additional bookkeeping and is most likely computational more expensive as well.

3.8.5 Micro-Operator Overview

In Table 3.1 we give a simplified summary of the basic building blocks of the operator interface. We feature two classes of operators, one class implements relational algebra operators, and the other group is for control mechanisms like sorting, partitioning, and network operations. We present the logical operator, the implemented concepts of the interface along with the name of the building block and a short, high-level description. The *Filter-*, *Aggregation-* and *TransformOperator* are generic in the sense that they require a method that implements the desired functionality and is able to operator on the provided tuple types. By the same mechanism the *SortOperator* can sort tuples according to the key or an attribute in ascending or descending order.

	Logical Operator	Operator Classification	Interface Building Blocks	Description
Implementation	Retrival	TupleProducer	ColumenRelationScan RowRelationScan	Scan a relation and produce tuples
		BlockConsumer TupleProducer	BlockScan	Scan a a block from the internal queue and produce tuples
	Selection. Projection	TupleConsumer TupleProducer	Filter	Filter relation according to some predicate
	Aggregation	TupleConsumer TupleProducer	Aggregation (via hash)	Hash aggregation according to an aggregation function e.g. count or sum
	Unary Operations (Map)	TupleConsumer TupleProducer	Transform	Transform or map tuple attributes according to a transform function
	Binary Operations	TupleConsumer TupleProducer	Join	Hash join operator
			TwoWayMerge	Fast merge routine for 2 input relations relying on work conducted by Balkesen et al. [34, 35, 42]
			MultiWayMerge	Fast merge routine for N input relations relying on work conducted by Balkesen et al. [34, 35, 42]
		TupleConsumer BlockProducer	Merge	Merge routine for 2 input relations
			SortedJoin	Join on two sorted input relations relying on work conducted by Barthels et al. [33]
Special Purpose	Network Operation	BlockConsumer	MpiSend	Send block over network with synchronous MPI_Send
		BlockProducer	MpiReceive	Receive block over network with synchronous MPI_Receive
	Sort Order	TupleConsumer TupleProducer	Sort	Sort input relation according to a sort function
			AvxSort	Fast sorting routine for narrow 8-byte tuples relying on work conducted by Balkesen et al. [34, 35, 42]
		BlockConsumer BlockProducer	AvxBlockSort	Fast sorting routine for narrow 8-byte tuples applied on block granularity. Relying on work conducted by Balkesen et al. [34, 35, 42]
	Data Partitioning	TupleConsumer BlockProducer	Partition	Partition the input relation into a specified number of partitions according to a partition function
	Printing	TupleConsumer TupleProducer	PrintTuple	Print tuple to *std::out*
	Materialization	TupleConsumer BlockProducer	MaterializeRow	Produce blocks from tuples and push to internal queue

Table 3.1: Classification of operators provided as basic building blocks.

4

Experimental Evaluation

In this section, we evaluate the expressiveness and performance of the operator interface. First, we present micro-benchmarks of each operator in isolation. Second, we compare the performance to a distributed radix hash join implemented by Barthels et al. [33]. Third, we demonstrate the wide range of applicability of the interface by running query 1 of the TPC-H Benchmark.

4.1 Experimental Setup

The Cray XC40/XC50 system [24] used in the experimental evaluation provides a combined total number of 5320 Hybrid, and 1431 Multicore compute nodes, which are fitted into 28 compute cabinets. A cabinet can be filled with up to three chassis, of which each can stack at most 16 compute blades. A single compute blade consists of 4 compute nodes that has two Intel Xeon Scalable processors integrated.

The Cray XC40 and Cray XC50 offer the same architecture and differ only in their node design. On each compute node the Cray XC40 offers two 18-core processors (Intel Xeon E5-2695 v4 @ 2.1GHz) and either customizable 64 or 128 GB of main memory per node. Cray XC50 in contrast provides 12-core (Intel Xeon E5-2690 v3 @ 2.60GHz) with each 64 GB of main memory per node [44].

The system implements a high-bandwidth, low-diameter network topology called Dragonfly [22, 45] in combination with the Aries ASIC network interconnect. The Aries interconnect consists of four NIC's and an Aries router, which is a system-on-a-chip device.

The connection between each NIC of the blade and the compute nodes is implemented by a standard 16x PCI Express 3 interface.

4.2 Operator Baseline Experiments

To benchmark operators in isolation, we compare the runtime and throughput of each micro-operator to an equivalent implementation – regarding memory accesses – implemented in plain C++.

4.2.1 Workloads

The experiments are conducted with narrow 8-byte tuples consisting of a 4-byte key and attribute respectively. We vary the relation size from 1M to 10M in steps of 1M each, which results in a peak of 80MB in tuples per relation.

4.2.2 Results

In Figure 4.1 we compare each micro-operator to a fast implementation in plain C++ to show the performance overhead of the interface. We present the relative difference in runtime between those two measurements.

We immediately see that more than half of the operators are actually slightly faster or slower in the range of a few percentage points. Thus the indirection introduced by the interface results in general not in a performance decrease. The reason that an operator is faster than its baseline is most likely due to noise.

The *Add* and *Double* benchmarks both show the same (absolute) performance as the *MaterializeOperator*. The reason is the *MaterializeOperator* at the end of either pipeline, which is needed to properly consume the input. Otherwise the compiler would optimize away the complete computation and those two benchmarks would be amazingly fast. Therefore, we decide to accumulate the results for those two benchmarks in a register and compared them to a similar implementation in C++. These results are shown with the additional label *[Scalar]*. As expected and previously shown in Section 3.8.1, the add operation is already inlined at compile time and thus the compiler can optimize away several instructions. The 20% runtime reduction, in comparison to the baseline, however can not only be explained by the noise. We need to further investigate this issue to track down the cause of the difference and to ensure that the baseline and the operator implementation perform exactly the same computational steps. The double operation is as

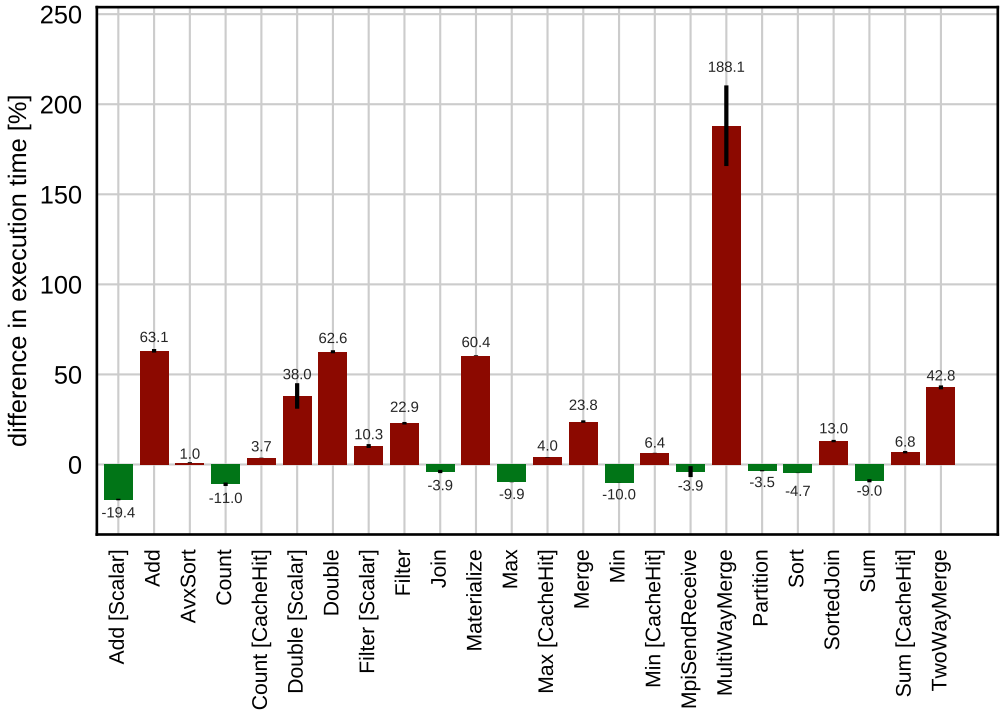


Figure 4.1: Relative runtime difference of conducted micro-benchmarks on each operator in isolation for relation size 10M.

well faster than the version limited by the *MaterializeOperator*. The *FilterOperator* suffers from the same implementation mismatch as well. By accumulating the results from the filter operation in a register, we gain an additional 12% and bring it down to only a 10% slowdown.

The aggregation operations *Count*, *Max*, *Min*, and *Sum* all outperform their baseline implementations. However, the benchmark is designed such that the hash table will not fit into the cache, which can be seen by their corresponding throughputs in Figure 4.2. Nevertheless, the operator interface shows a slightly better performance, which indicates that there is again some noise or the rare chance that the compiler indeed is able to optimize to a higher degree. This could actually possible in cases where the compiler gains a better global view through inlining a complete pipeline. For a second set of benchmarks indicated by *CacheHit*, where the hash table fits into the cache, this advantage is lost and the operators are 3.7-6.8% slower.

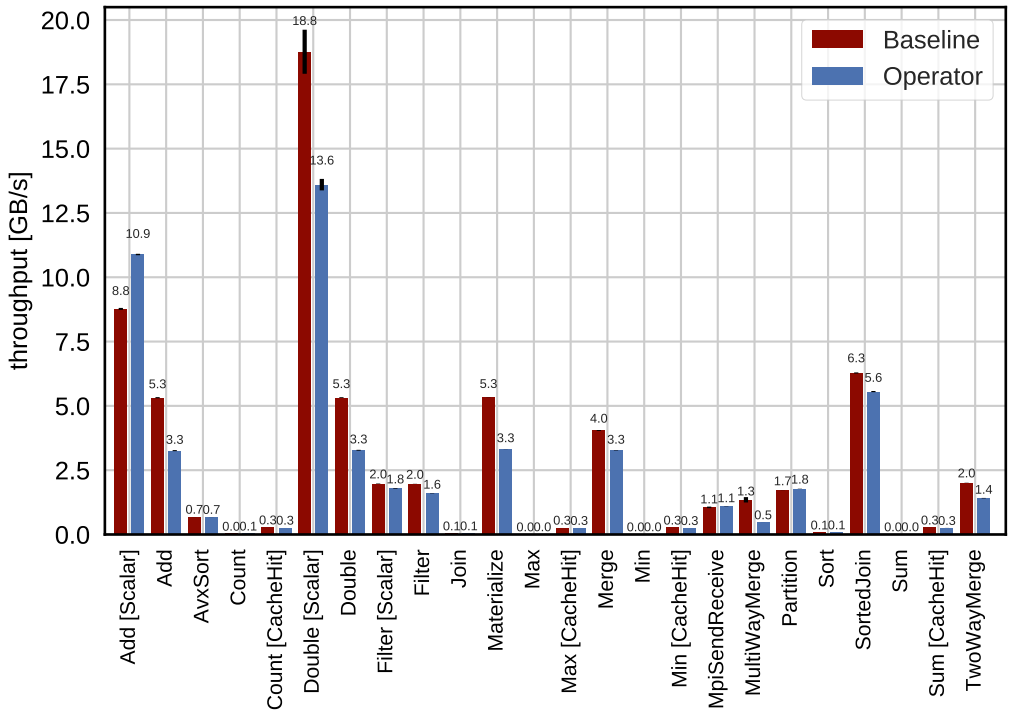


Figure 4.2: Throughput comparison of conducted micro-benchmarks on each operator in isolation for relation size 10M.

The outlier is the *MultiWayMerge* benchmark that is nearly 3 times slower. This operator together with the *TwoWayMerge*, *SortedJoin*, and *AvxSort* are re-implemented from previous work conducted by Barthels et al. [32, 33] and Balkesen et al. [34, 35, 42]. Therefore, they do not perfectly fit into the operator interface: First, they usually act as full pipeline break meaning they materialize the complete input inside their body. Second, they allocate memory on their own and therefore are not benefiting from the memory management implemented by the interface.

The runtime of an operator compared to a baseline implementation is not the sole performance metric. We are additionally concerned about achieving the upper limit of the memory bandwidth offered by the system. Therefore, Figure 4.2 shows the throughput in GB/s for the baselines and the micro-operators.

As expected, the micro-operators still lack some additional fine-tuning to push the performance further towards the baseline. All in all, the operator interface does not add a

huge performance penalty on the micro-operator level. The overhead is in general in the range of 5-20%.

4.3 Radix Hash Join

We re-implemented a distributed radix hash join priorly designed and fine-tuned by Barthels et al. [32, 33]. The algorithm was implemented as closely as possible out of the building blocks provided by the operator interface. The implementation roughly spans 700 lines of code and differs in several key components:

1. **Network operations:** The operator interface has only synchronous *MPI_Send* and *MPI_Receive* operations to send data over the network in contrast to the asynchronous one-sided communication mechanism foMPI [31] used by Barthels et al. [32, 33].

2. **Scheduling:** The original algorithm is optimized for two overlapping phases that can run in parallel. Our implementation does not (yet) feature those phases. In Chapter 5 we address this topic in more detail.

3. **Compression:** Prior to transmission, the tuples are compressed into 64-bit values using prefix compression [33, p. 5]. The operator interface does not offer this functionality. This could be solved by introducing a compression/decompression operator since our design is extensible.

4. **Build-Probe:** The operator interface relies on a *std::unordered_map* for hashing, which is considerably slower and cache-unfriendly in comparison to the solution presented by Balkesen et al. [35].

5. **Computation nodes**: Due to the synchronous network operations we require dedicated nodes for sending and distinct nodes for receiving. This difference, however, does not ultimately effect the performance like the former factors.

4.3.1 Workloads

We design and carry out the experiment as closely as the scale-out experiment described in Section Experimental Evaluation [33]. The tuples are composed of 8-byte key and 8-byte attribute value. The attribute can have any random value. For the final join with highly

distinct key values, it is essential that each key appears only once in the inner and the outer relation since we focus only on 1-to-1 workloads. Each "send" core gets assigned the same amount of input data – per core we assign 40 million tuples per relation. This results in 2.4 TB of data on 2048 "send" cores (and 2048 "receive" cores respectively). We assign a unique range of keys to each of the nodes. Keys are exchanged globally to ensure that each node possesses keys of the entire value range. Finally, we shuffle the keys in a random order.

4.3.2 Results

Our experimental results of Figure 4.3 show a peak performance of 2.96 billion tuples per second – or 47.35 GB/s – for 2048 sending and receiving cores. While adding more and more cores to the system the throughput is able to increase. The standard deviation however increases as well for bigger deployments, which is somewhat normal because there is certainly interference and noise on the shared system.

Figure 4.4 presents the execution time per send and receive core respectively, as a breakdown per operation. The different phases of the join include following operations: i) Scanning the input relations and network partitioning of the data, ii) sending the partitions to their dedicated cores, iii) receiving the transferred packages, iv) the local partitioning pass to ensure that partitions fit into the processor cache, and v) the join that includes the build and probe phase. The compute imbalance shown is the difference between the average and total execution time. As expected and previously shown by Barthels et al. [33, p. 7], the time spent for the two network operations increases significantly. This behaviour is aggravated even more by the synchronous MPI operations used. The experiment for 64 send and receive cores is clearly an outlier and suffers from a bad configuration, which can be seen best in the detailed breakdown in Figure 4.4. As we add more cores, the compute imbalance increases clearly, which is owed to the shared resources on the supercomputer that does not guarantee complete performance isolation. In addition, not all nodes can be physically co-located for large deployments, which results in higher remote memory access latencies for several nodes.

In Figure 4.5, we show the aforementioned comparison to the presented, highly-tuned radix hash join algorithm implemented by Barthels et al. [33]. For apparent reasons, we do not achieve similar performance and the comparison is not entirely fair. The achieved performance is in the range of one-tenth to one-fifteenth of the original performance.

Table 4.1 presents a comparison on a per operation level. The operator interface does

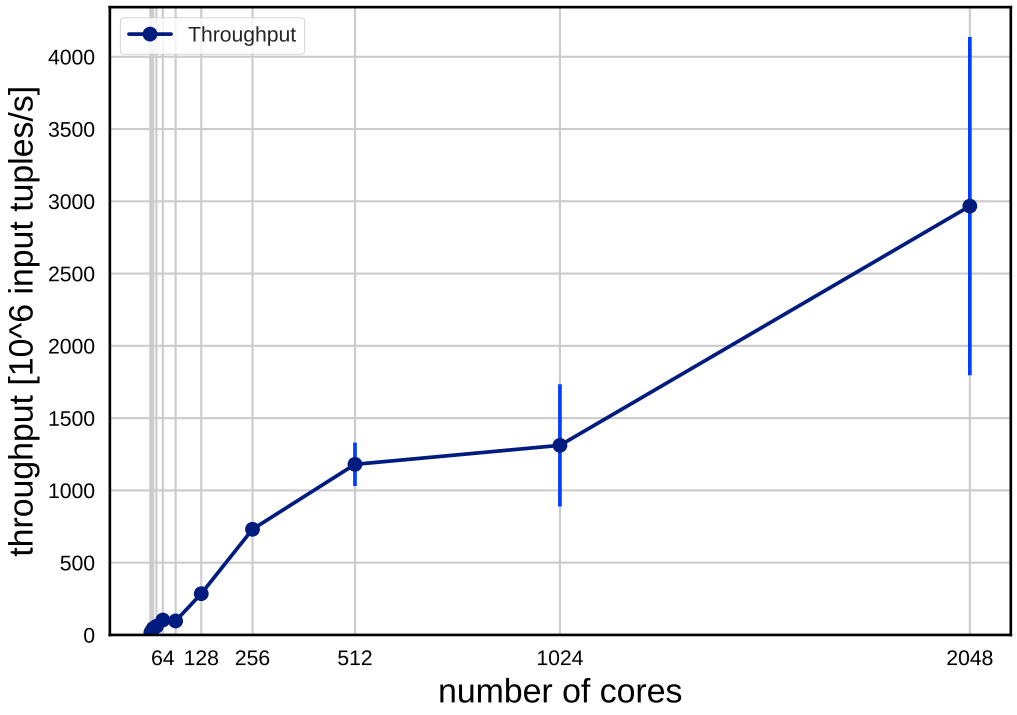


Figure 4.3: Scale-out experiment of the radix hash join for the operator interface. The error bars present the standard deviation.

not compute a histogram nor does it allocate windows for the network communication yet. The relevant difference in execution time is added in the send/receive and build-probe phases. Those phases are previously stated to differ most among the two implementations, favouring the solution proposed by Barthels et al. [33].

Nevertheless, the design goals of re-usability are met and nicely shown. In Chapter 5, we have a closer look at the reasons for the large performance gap.

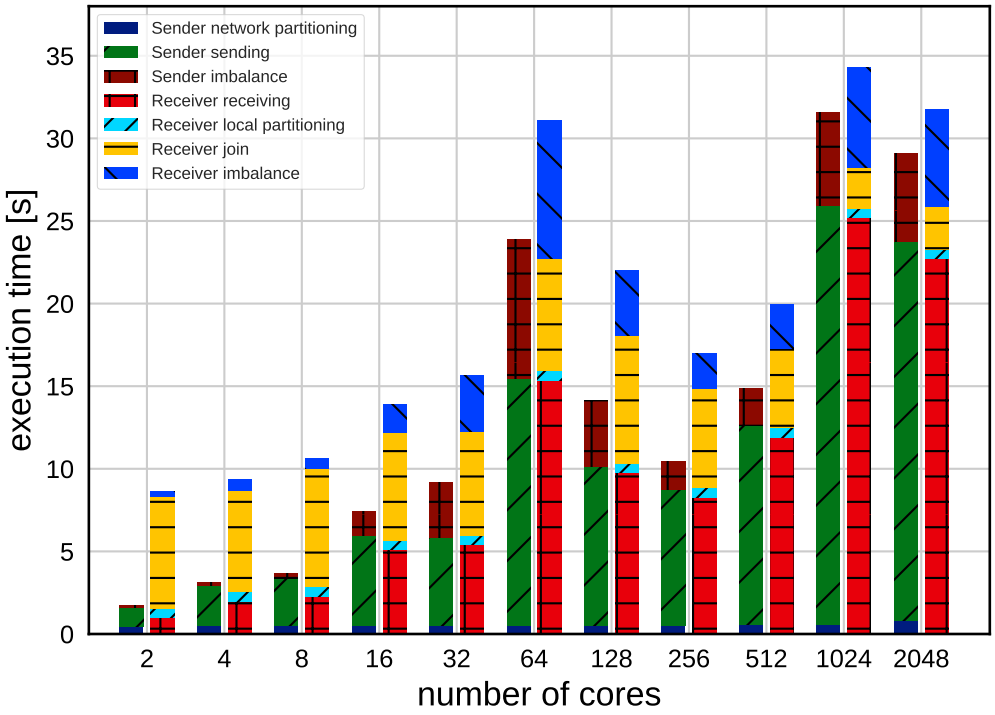


Figure 4.4: Detailed breakdown of the execution time of the radix hash join for 40 million tuples/relation/core.

Phase	Execution Time [s]		Diff. %
	Baseline	Interface	
Histogram Comp.	0.34	-	-100.00
Window Allocation	0.21	-	-100.00
Network Partitioning	0.60	0.58	-4.14
Send	1.48	25.39	3317.46
Receive		25.19	
Local Partitioning	0.58	0.57	-1.38
Build-Probe	0.51	2.51	391.36
Imbalance	0.62	11.96	1829.83
Total	4.34	66.20	1425.26

Table 4.1: Comparison of different phases for 1024 cores and 40M tuples/relation/core.

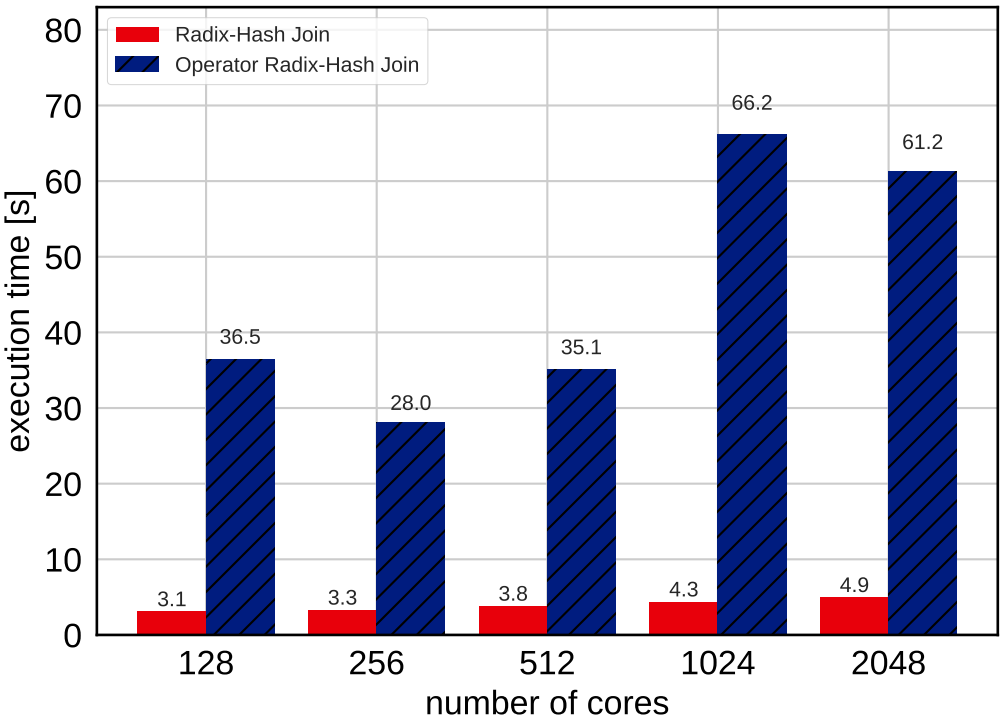


Figure 4.5: Comparison of the execution time of the two different radix hash join implementations for 40 million tuples/relation/core.

4.4 TPC-H Query 1

The TPC-H Benchmark [46] consists of ad-hoc queries and data modifications to support business decisions. Our intention on this evaluation metric is not the performance but rather to show the expressiveness and re-usability of the interface and its micro-operators. To accomplish query 1, we implemented only two additional functions: (1) a predicate function to filter the relation and (2) an aggregation function that computes the sums, averages, and counts, which are selected by the output.

4.4.1 Workloads

We built the *LINEITEM* table shown in Table 4.2 in memory prior to execution. The *LINEITEM* table has a base cardinality of 6 million rows, which can be scaled by an appropriate scale factor *SF* to obtain the desired database size. We ran our experiments

Column Name	Datatype	C++ type	Comment
L_ORDERKEY	identifier	uint64_t	
L_PARTKEY	identifier	uint64_t	
L_SUPPKEY	identifier	uint64_t	
L_LINENUMBER	integer	int	
L_QUANTITY	decimal	double	
L_EXTENDEDPRICE	decimal	double	
L_DISCOUNT	decimal	double	
L_TAX	decimal	double	
L_RETURNFLAG	fixed text, size 1	int	combined,
L_LINESTATUS	fixed text, size 1	int	encoded as single integer [0,3]
L_SHIPDATE	date	uint64_t	encoded as UNIX timestamp
L_COMMENTDATE	date	uint64_t	encoded as UNIX timestamp
L_RECEIPTDATE	date	uint64_t	encoded as UNIX timestamp
L_SHIPINSTRUCT	fixed text, size 25	std::string	
L_SHIPMODE	fixed text, size 10	std::string	
L_COMMENT	variable text, size 44	std::string	

Table 4.2: LINEITEM table layout

for $SF = \{1, 2, 4, 8\}$. A single tuple of the *LINEITEM* table expressed in C++ has the size of 192 bytes, which results in overall database sizes in the range of 1.2 GB up to 9.2 GB stored in a columnar layout. For simplicity we encode RETURNFLAG and LINESTATUS as a single integer in the interval [0,3].

The *Pricing Summary Report Query*, also known as query 1, reports a summery about all lineitems that are shipped up to a given date. Listing 4.1 shows the corresponding SQL query, which provides the totals for several attributes. Those totals are aggregated and grouped by RETURNFLAG and LINESTATUS. The result is in addition sorted in ascending order and features a count of the number of lineitems in each group. The substitutional parameter *DELTA* in line 14 controls how many rows are scanned and should result in a table scan of 95% to 97% of all rows.

```
1  SELECT
2    L_RETURNFLAG, L_LINESTATUS,
3    SUM(L_QUANTITY) AS SUM_QTY,
4    SUM(L_EXTENDEDPRICE) AS SUM_BASE_PRICE,
5    SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS SUM_DISC_PRICE,
6    SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)*(1+L_TAX)) AS SUM_CHARGE,
7    AVG(L_QUANTITY) AS AVG_QTY,
8    AVG(L_EXTENDEDPRICE) AS AVG_PRICE,
9    AVG(L_DISCOUNT) AS AVG_DISC,
10   COUNT(*) AS COUNT_ORDER
11 FROM
12   LINEITEM
13 WHERE
14   L_SHIPDATE ⩽ date '1998-12-01'-interval'[DELTA]'day
15 GROUP BY
16   L_RETURNFLAG,
17   L_LINESTATUS
18 ORDER BY
19   L_RETURNFLAG,
20   L_LINESTATUS
```

Listing 4.1: TPC-H Query 1 - Pricing Summary Report

4.4.2 Results

Figure 4.6 and Figure 4.7 show the execution time and throughput, respectively, of the query executed at different relation sizes. In Figure 4.6, we include the time required to build the *LINEITEM* table as well. We are aware of the fact that our quantitative results, performed on a single node without parallel execution, are not comparable and the scale factor for the *LINEITEM* table usually ranges up to 100'000. However, the query provides a stable throughput while scaling out. The main objectives in expressiveness and re-usability are accomplished by the fact that a minimal effort was required to build this query. To execute the query a total of 5 operations are necessary: (1) scan, (2) filter, (3) aggregation, (4) sort, and (5) materialize. Our query pipeline features exactly those 5 operators and only those. We are confident that additional TPC-H queries

can be implemented with similarly low effort. The basic building blocks of the interface provide in addition all necessary functionalities to execute the query in parallel. The *PartitionOperator* mentioned in Section 3.8.3 is able to partition the input relation whereas the *MpiOperators* of Section 3.8.4 can distribute the data among multiple cores to run it simultaneously.

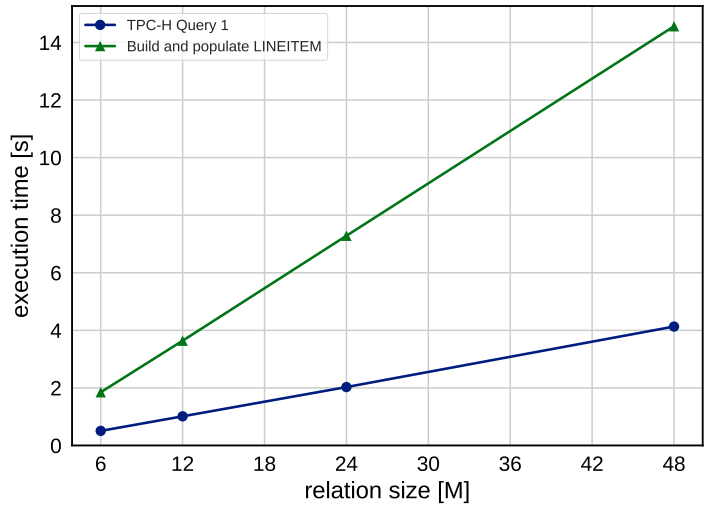


Figure 4.6: Scale-out behaviour for query 1 of TPC-H Benchmark.

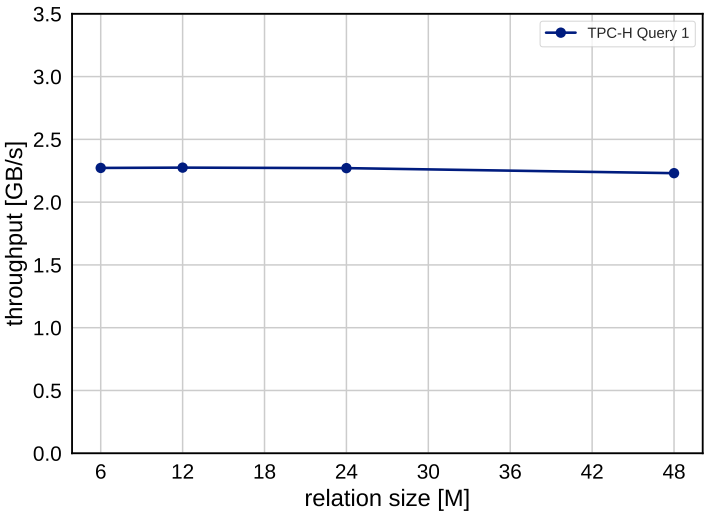


Figure 4.7: Throughput for query 1 of TPC-H Benchmark.

5 Discussion

A thorough review and analysis of a system is necessary out of two reasons. First, we want to detect current shortcomings and limitations. Second, we need to justify the possible potential in order to put the future work into a perspective. In this chapter we shortly talk about the future potential but mainly focus on the limitations that were discovered in Chapter 4. In Section 7.2 on future work we present the work items that can be accomplished after the following disadvantages have been resolved.

5.1 Future Potential

Providing a common code base and interface as fundamentals to integrate into novel algorithms would not only speed up the development process, but facilitate debugging to a large extent. Many functionalities like the memory management can be provided "out-of-the-box" instead of being duplicated for every new algorithm. In addition, less specific knowledge is needed to implement an algorithm and therefore the saved time can be invested in other areas.

Optimizations to common building blocks would contribute to reduce execution times in many algorithms that use those operators. It is worth mentioning that identifying phases of a complex algorithm that can run in parallel is usually easier than designing and implementing the algorithm in such a way. Separating parallel phases and scheduling them sometimes requires so much effort and knowledge, it renders impossible. We believe

that the operator interface in combination with compiler optimizations could solve that problem and even achieve better performance.

5.2 Limitations

Chapter 4 has clearly shown that the most striking drawback is the performance, manifesting as a central theme throughout the entire evaluation chapter.

5.2.1 General

The micro-benchmarks on the operators in isolation exhibit just a moderate overhead for the operator interface in comparison to the baseline implementations. However, in terms of throughput and in comparison to the theoretical upper memory bandwidth that is feasible on the underlying system, the operators (and benchmarks) still lack additional fine-tuning. Tuning operators is a tedious and time-consuming activity that requires more than just a profound knowledge about algorithmic design, systems programming, and the hardware and is, unfortunately, out of scope for this thesis. To increase the throughput, additional cache-efficient, independent, and highly parallelizable algorithms must be integrated, e.g., for aggregation as engineered by Müller [47].

5.2.2 Radix Hash Join Analysis

On the one hand, the 10-15 fold decrease in performance of the interface abstraction, shown in Figure 4.5, is quite severe. On the other hand, it is "just" a 10-15 fold slowdown compared to hand-tuned C++ code. Similar comparisons between Spark and hand-written C code however detected a much bigger performance difference in the range of 100x, favouring plain C [48]. In contrast to these findings we are on the right track, even if we are not yet on target. We are confident that the overhead can be decreased to a factor of 2 or lower with additional effort in fine-tuning.

Fine-tuning of the *micro-operators* is one means to achieve better performance. Another more influencing aspect is to allow a more fair comparison of the two different radix hash joins. While one was designed and implemented with efficiency in mind, was optimized and refactored to squeeze out every last drop of performance, the other algorithm was realised out of basic building blocks provided by the operator interface. The comparison 4.3 of the two algorithms has shown multiple differences. Most of them were expected

to directly interfere with the resulting performance of the algorithm. The obtained results proofed our intentions.

To address these mismatches we propose following solutions:

1. **Network operations:** The implementation of one-sided, asynchronous communication mechanism is a must in order to i) allow parallel execution, ii) increase usage of network bandwidth, and iii) to decrease the total amount of required cores. To be able to compare the network operations exactly, the foMPI library [31] would be the appropriate candidate to choose.

2. **Scheduling:** An external scheduling mechanism needs to be designed that schedules pipelines to achieve some degree of parallelism.

3. **Compression:** Compression and de-compression operators should be added to the set of building blocks, which are needed to reduce the amount of transferred bytes over the network. This lossless and reversible transformation can be accomplished after partitioning, when all tuples in the same bucket exhibit a common property, e.g., a subset of identical bits in the hash key.

4. **Build-Probe:** The implementation for hashing has to be upgraded to a faster and cache-friendly approach, e.g., to the algorithm proposed by Balkesen et al. [35].

5. **Computation nodes:** Once one-sided communication mechanism are implemented, this issue can be solved easily. Reducing the size of a deployment lowers the probability of interference on shared resources and unfavourable (physical) node allocation.

As one can see in Table 4.1, we can blame the network operations for a performance decrease up to a factor of 10. If we factor in the computation imbalance, essentially the complete overhead is resulted by the different communication mechanisms. By implementing one-sided network operations and reducing the deployment size, the compute imbalance should decrease as well.

5.2.3 Sort-Merge Join Analysis

To show the expressiveness of the operator interface, we attempted to re-implement this join algorithm as well.

Several discussions about the sort-merge join design have shown the current limitation of existing building blocks. To provide the necessary functionality, we add operators

like *TwoWayMerge*, *MultiWayMerge*, *AvxSort*, *AvxBlockSort*, and *SortedJoin* shown in Table 3.1 and discuss their performance in Section 4.2.2. Comparing the aforementioned operators and the different phases of the high-level overview in Section 2.4.2, one could get the impression that it is quite a good fit. However they only offer a solution to some of the missing pieces. Even though we consider a few simplifications, the interface offers too little support yet. The main issues detected are the following:

- **M-way merge:** The integrated *m-way* merge operators from Balkesen et al. [34, 35, 42] operate solely on the whole input relation, not on block granularity as it would be necessary.
- **Block size:** For the *m-way* merge operators, we need to lift the restriction of a single block size in order to serve the memory needs. Each merge pass combines m blocks of size s, which results in a single large block of size $m \cdot s$. Thus, the memory pool must be scaled to feature lists of free memory blocks for every required block size or different efficient approaches.
- **Additional new operators:** Other auxiliary operators might be needed to manage or distribute blocks in an appropriate fashion. We identified the need for a *RoundRobinOperator* that simply redistributes blocks among several queues. A function passed to this operator should control how exactly blocks are distributed. By this means, one can group blocks into runs that can be merged with the discussed *m-way* merge operators.

Since the sort-merge join is inferior in performance, it is not absolutely mandatory to solve the mismatch. However, if one would like to do so, a redesign of the mentioned operators to work on block granularity would be necessary. In addition, the memory management must support different block sizes up to the size of the finally sorted relations that are used in the join operation. We did not mention the asynchronous network operations anew because we assume that this issue is solved by that time.

6
Related Work

History is repeating itself and questions to identical, re-occurring topics are asked and answered by different people at different times. Therefore the proposed solutions often differ due to the current needs and advances in technology. In the following, we look at different approaches taken in research and industry to propose an interface and model for a query engine to process OLAP workloads.

The earliest solutions to a uniform interface were the Bracket Model, implemented in Bubby [6] and Gamma [7], the simple operator model used in R* [10], and the most prominent and sophisticated example called Volcano [3, 11, 12]. The performance of Volcano's iterator model suffers from query interpretation overhead and was designed more than 20 years ago for substantially different hardware systems. Therefore many newer systems switched to compiling query plans by Just-in-Time (JiT) compilation or compiling the entire query plans to native machine code.

Many commercial and research systems have evolved based on existing ideas, such as the iterator model, or present new ones. One of the first system was MonetDB [20] (later called MonetDB/X100 [17]). MonetDB provides execution primitives that process data in a column-at-a-time fashion, which minimizes the interpretation overhead. X100 became the new execution engine for the MonetDB system [49]. It features a resembling, classical Volcano-style engine. The engine is highly CPU-efficient, though because all executions are based on the concept of vector processing. Therefore the joined system combines the benefits from vectorized execution with no intermediate result materialization and column-wise processing.

Later on MonetDB/X100 evolved into the VectorWise technology. VectorWise was acquired by Actian Corporation and integrated into the Ingres database. Costea et. al [50] propose a massively parallel processing solution to the VectorWise DBMS [51], which is a state-of-the-art relational database management system based on a vectorized query execution engine designed for OLAP workloads. They present a distributed exchange operator, similar to Volcano's exchange operator [3], to provide distributed parallelism. In small deployments they are able to achieve a linear scale-up. For larger deployments, the network bandwidth becomes the limiting factor.

SharedDB [15] has the ability to processes OLTP, OLAP, and mixed workloads while sustaining high throughput with response time guarantees such as Service Level Agreements (SLA). It was developed in the context of multi-query optimization and data stream processing. It has a batch-oriented query processing model enhanced with a unique computation sharing approach that allows hundreds of concurrent queries and updates. Giannikis et al. [52] further explore the problem of shared workload optimization (SWO) through a branch and bound optimization technique. Our approach proposes some kind of bulk processing too, in the form of the block producing and consuming interface, though we process OLAP workloads only.

Oracle and SAP proposed a hybrid, distributed query and transaction processing system depending on shared buffer caches [53] and shared logs [54], implemented in Oracle Database In-memory Option (DBIM) and in SAP HANA, respectively.

Rödiger et al. [55] address the problem of switch contention due to uncoordinated communication and load imbalance caused by the inflexibility of the classic exchange operator in OLAP workloads. They propose a mechanism for better scalability by distinguishing between local and distributed parallelism, called hybrid parallelism. To improve communication, they present a communication multiplexer on top of RDMA. The work was integrated and evaluated on HyPer [18], which is a hybrid in-memory database system supporting OLAP and OLTP workloads. Similar to our approach, they address the problem of network communication and propose an approach using RDMA.

Liu et al. [56] investigate the performance of a data shuffling operator using RDMA. They rely on a pull-based, vectorized operator model to return tuples in batches. According to their results, the two-sided Send/Receive operations over the Unreliable Datagram transport service offer the most robust performance across all tested configurations. Like our execution model they rely on a pull-based model with the ability to return tuples in blocks and use two-sided RDMA operations.

I-Store [57] proposes a shuffle-based execution model to speed up query execution for distributed database systems built on fast networks. Their system is a distributed query engine implemented on Network-Attached-Memory (NAM) [58], which is a dedicated architecture for fast networks. It decouples compute and memory nodes using RDMA for fast communication between those nodes. Their stream-based execution model partially replicates data to execute fully pipelined plans without intermediate shuffle operators. The compute server can be scaled independently from the memory servers. Their approach is based on fast networks and RDMA like ours, however their streaming model with data replication and dedicated compute and memory nodes differs from our solution.

The Wildfire system [59] is designed as a layer sitting between Spark executors and the shared file system, providing support for Hybrid Transactional and Analytical Processing (HTAP). Wildfire extends Spark to provide improved OLAP performance by the ability to push-down queries into the Wildfire engine and by support of broader user-defined functions (UDF) and user-defined aggregation functions (UDAF). Queries are issued over the Spark SQL API, and the connection between a Spark Executor and Wildfire's columnar engine enables analytical capabilities to the entire Spark ecosystem. The Wildfire engine supports extensible UDF and UDAF and is therefore extensible like our approach.

The Myria Big Data Management Service [60, 61] is a cloud service that is easy to use through its web page based interface. MyriaX is the shared-nothing relational query engine that uses dataflow operators with support for parallelism, pipelining, and cyclic graphs. MyriaX features a Volcano-style exchange operator and other operators that consume and produce tuples in batches (multiple tuples). The operators rely on either a pull-based or push-based model. They encapsulate queries without data-shuffling inside a fragment. Such queries inside a fragment are executed with a pull-based approach. Across such fragments the push-based approach is used. They claim to combine ease-of-use with performance. Our approach features a similar pull-based approach to produce tuples in batches. The web interface is easier to use compared to our solution – we still need to write C++ code to build our query pipelines.

Wang et al. [62] focus on a system to query solely tree-structured data. They, too, propose a Volcano-style query execution engine in combination with multi-threading support for common relational operators. Their approach is limited to tree-structured data, whereas our solution is oblivious to the underlying storage.

Menon et al. [63] present the idea of a relaxed operator fusion (ROF) query processing model for in-memory OLAP DBMS. By introducing staging points in query plans the

DBMS is able to temporarily materialize results. ROF is a hybrid scheme between single tuple and vectorized processing. This approach allows faster query execution through improved inter-tuple parallelism by using a combination of prefetching and SIMD vectorization. They evaluated their approach in the Peloton in-memory DBMS. Their runtime comparison against HyPer and Actian Vector showed lower execution times by a factor of 1.8. The introduced staging points resemble our internal queue. In both approaches temporary results are materialized.

Flare [48] proposes a new query engine backend for Spark, which compiles queries to native code without loosing Spark's expressiveness. Flare replaces several parts of the underlying runtime of Spark, adds better optimization for large classes of user-defined functions, and increases the overall performance. It is based on a compiler framework for high-performance domain-specific languages called Delite [64, 65, 66]. Similar approaches were taken in TupleWare [67], Weld [68], and HPAT [69]. They all compile the logic of parallelizable dataflow operators gathered from DAGs into native machine code. The compilation of dataflow operators to native code is similar to our approach, in addition to the provided expressiveness by our interface.

7 Conclusions

7.1 Summary

In this thesis we discuss the problem of how to combine expressiveness, simple usage, and high performance to overcome the shortcomings in current systems. As solution we propose the design of composable *sub-operators*, reuse them in complex pipelines, and provide an expressive interface that can easily be extended.

Our work is summarized by the finding that current compilers can remove the overhead, introduced by an operator abstraction, for most operators. Our standardized interface provides composable building blocks that can be used to build complex pipelines to express high level algorithms. Dedicated operators provide efficient communication mechanisms for high-bandwidth, low-latency networks. Missing functionality can be added to the system with ease, encapsulated in additional operators and benefit from the introduced concepts of the interface.

While we see significant potential in our approach compared to previously proposed operator models, we see the necessity to tune for efficiency first. The advantages in expressiveness, re-usability, and ease-of-use are diminished by the current performance but are expected to shine in the not-so-distant future.

7.2 Directions for Future Work

Once performance restrictions have been removed and the potential of the operator abstraction is confirmed, the long time visions can be addressed. We believe that the additional topics of future work are orthogonal. They are orthogonal to each other in the sense that they can be accomplished in parallel with few dependencies among each other.

The main topics are:

1. Design and integration of a scheduling mechanism, to interleave computation and communication.
2. Integration of partial execution of an operator, e.g., co-routines that store the execution state and allow to return to it later on.
3. Integration of additional *micro-operators* to provide richer functionality, since our interface is extensible.
4. Implementation of complex algorithms or pipelines expressed by the operator interface.
5. Addition of multi-threading to the interface to execute a pipeline by multiple threads.
6. Re-design to a hybrid model that allows push and pull based operators in unison in combination with supporting multiple outputs per operator.
7. Evolution of the internal queue to be able to store as well hash tables, columns, or other appropriate formats.

Bibliography

[1] VCloudNews. Every day big data statistics, April 2015.

[2] Jeff Schultz. How much data is created on the internet each day?, October 2017.

[3] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, February 1994.

[4] The Apache Software Foundation. Hadoop, November 2017.

[5] The Apache Software Foundation. Apache spark: Lightning-fast cluster computing, November 2017.

[6] 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. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):4–24, Mar 1990.

[7] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):44–62, March 1990.

[8] R A Lorie. XRM - an extended (n-ary) relational memory. Technical Report G320-2096, IBM Research Report, 1974.

[9] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[10] Bruce G. Lindsay, Laura M. Haas, C. Mohan, Paul F. Wilms, and Robert A. Yost. Computation and communication in r*: A distributed database manager. *ACM Trans. Comput. Syst.*, 2(1):24–38, February 1984.

[11] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 102–111, New York, NY, USA, 1990. ACM.

[12] G. Graefe and D. L. Davison. Encapsulation of parallelism and architecture-independence in extensible database query execution. *IEEE Trans. Softw. Eng.*, 19(8):749–764, August 1993.

[13] Dina Bitton, David J. DeWitt, and Carolyn Turbyfill. Benchmarking database systems a systematic approach. In *Proceedings of the 9th International Conference on Very Large Data Bases*, VLDB '83, pages 8–19, San Francisco, CA, USA, 1983. Morgan Kaufmann Publishers Inc.

[14] Sriram Padmanabhan, Timothy Malkemus, Ramesh C. Agarwal, and Anant Jhingran. Block oriented processing of relational database operations in modern computer architectures. In *ICDE*, pages 567–574. IEEE Computer Society, 2001.

[15] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. Shareddb: Killing one thousand queries with one stone. *Proc. VLDB Endow.*, 5(6):526–537, February 2012.

[16] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. Batchdb: Efficient isolated execution of hybrid oltp+olap workloads for interactive applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 37–50, New York, NY, USA, 2017. ACM.

[17] Peter Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *In CIDR*, 2005.

[18] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 195–206, Washington, DC, USA, 2011. IEEE Computer Society.

[19] Stefan Manegold, Martin L. Kersten, and Peter Boncz. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *Proc. VLDB Endow.*, 2(2):1648–1653, August 2009.

[20] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the 25th International*

Conference on Very Large Data Bases, VLDB '99, pages 54–65, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[21] InfiniBand Trade Association. Infiniband architecture specification, November 2017.

[22] Cray® xc™ series network.

[23] P. W. Frey and G. Alonso. Minimizing the hidden cost of rdma. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 553–560, June 2009.

[24] Cscs piz daint supercomputer.

[25] Mpi 3.1, 2015.

[26] K. Anikiej. Multi-core parallelization of vectorized query execution. *Master's thesis, VU University*, 2010.

[27] Andrei Costea, Adrian Ionescu, Bogdan Raducanu, Michal Switakowski, Cristian Barca, Juliusz Sompolski, Alicja Luszczak, Michal Szafranski, Giel de Nijs, and Peter Boncz. Vectorh: Taking sql-on-hadoop to the next level. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1105–1117, New York, NY, USA, 2016. ACM.

[28] Open mpi, 2017.

[29] Mpich, 2017.

[30] Mvapich, 2017.

[31] Robert Gerstenberger, Maciej Besta, and Torsten Hoefler. Enabling highly-scalable remote memory access programming with mpi-3 one sided. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 53:1–53:12, New York, NY, USA, 2013. ACM.

[32] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. Rack-scale in-memory join processing using RDMA. In *SIGMOD*, pages 1463–1475, 2015.

[33] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. Distributed join algorithms on thousands of cores. *PVLDB*, 10(5):517–528, 2017.

[34] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proc. VLDB Endow.*, 7(1):85–96, September 2013.

[35] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *ICDE*, pages 362–373. IEEE Computer Society, 2013.

[36] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 510–521, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[37] ISO/IEC JTC 1/SC 22. C++11 standard (iso/iec 14882:2011), 2011.

[38] Gnu compiler collection, 2017.

[39] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. Memory management techniques for large-scale persistent-main-memory systems. In *VLDB 2017, August 28 to September 1, 2017, Munich, Germany.2017.*, 8 2017.

[40] Benoît Dageville and Mohamed Zait. Sql memory management in oracle9i. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 962–973. VLDB Endowment, 2002.

[41] H. Zhang, G. Chen, B. C. Ooi, K. L. Tan, and M. Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, July 2015.

[42] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-memory hash joins on modern processor architectures. *IEEE Trans. Knowl. Data Eng.*, 27(7):1754–1766, 2015.

[43] Stefan Manegold, Peter Boncz, and Martin Kersten. Optimizing main-memory join on modern hardware. *IEEE Trans. on Knowl. and Data Eng.*, 14(4):709–730, July 2002.

[44] Cray xc series.

[45] J. Kim, W. J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. In *2008 International Symposium on Computer Architecture*, pages 77–88, June 2008.

[46] Tpc benchmark^{TM}h (tpc-h), 2017.

[47] Ingo Müller. *Engineering Aggregation Operators for Relational In-Memory Database Systems*. PhD thesis, Karlsruhe Institute of Technology, Germany, 2016.

[48] Grégory M. Essertel, Ruby Y. Tahboub, James M. Decker, Kevin J. Brown, Kunle Olukotun, and Tiark Rompf. Flare: Native compilation for heterogeneous workloads in apache spark. *CoRR*, abs/1703.08219, 2017.

[49] Marcin Zukowski, Peter A. Boncz, Niels Nes, and Sándor Héman. Monetdb/x100 - a dbms in the cpu cache. *IEEE Data Eng. Bull.*, 28:17–22, 2005.

[50] Andrei Costea and Adrian Ionescu. Query optimization and execution in vectorwise MPP. Master's thesis, Vrije Universiteit Amsterdam, August 2012.

[51] M. Zukowski, M. van de Wiel, and P. Boncz. Vectorwise: A vectorized analytical dbms. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1349–1350, April 2012.

[52] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. Shared workload optimization. *Proc. VLDB Endow.*, 7(6):429–440, February 2014.

[53] Niloy Mukherjee, Shasank Chavan, Maria Colgan, Dinesh Das, Mike Gleeson, Sanket Hase, Allison Holloway, Hui Jin, Jesse Kamp, Kartik Kulkarni, Tirthankar Lahiri, Juan Loaiza, Neil Macnaughton, Vineet Marwah, Atrayee Mullick, Andy Witkowski, Jiaqi Yan, and Mohamed Zait. Distributed architecture of oracle database in-memory. *Proc. VLDB Endow.*, 8(12):1630–1641, August 2015.

[54] Anil K. Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean, Franz Färber, Francis Gropengiesser, Christian Mathis, Thomas Bodner, and Wolfgang Lehner. Towards scalable real-time analytics: An architecture for scale-out of olxp workloads. *Proc. VLDB Endow.*, 8(12):1716–1727, August 2015.

[55] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. High-speed query processing over high-speed networks. *Proc. VLDB Endow.*, 9(4):228–239, December 2015.

[56] Feilong Liu, Lingyan Yin, and Spyros Blanas. Design and evaluation of an rdma-aware data shuffling operator for parallel database systems. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 48–63, 2017.

[57] Abdallah Salama, Carsten Binnig, Tim Kraska, Ansgar Scherp, and Tobias Ziegler. Rethinking distributed query execution on high-speed networks. *IEEE Data Eng. Bull.*, 40(1):27–37, 2017.

[58] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It's time for a redesign. *PVLDB*, 9(7):528–539, 2016.

[59] Ronald Barber, Christian Garcia-Arellano, Ronen Grosman, René Müller, Vijayshankar Raman, Richard Sidle, Matt Spilchen, Adam J. Storm, Yuanyuan Tian, Pinar Tözün, Daniel C. Zilio, Matt Huras, Guy M. Lohman, Chandrasekaran Mohan, Fatma Özcan, and Hamid Pirahesh. Evolving databases for new-gen big data applications. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[60] Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. Demonstration of the myria big data management service. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 881–884, New York, NY, USA, 2014. ACM.

[61] Jingjing Wang, Tobin Baker, Magdalena Balazinska, Daniel Halperin, Brandon Haynes, Bill Howe, Dylan Hutchison, Shrainik Jain, Ryan Maas, Parmita Mehta, Dominik Moritz, Brandon Myers, Jennifer Ortiz, Dan Suciu, Andrew Whitaker, and Shengliang Xu. The myria big data management and analytics system and cloud services. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[62] Zhiyi Wang, Dongyan Zhou, and Shimin Chen. Steed: An analytical database system for tree-structured data. *Proc. VLDB Endow.*, 10(12):1897–1900, August 2017.

[63] Prashanth Menon, Andrew Pavlo, and Todd C. Mowry. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *PVLDB*, 11(1):1–13, 2017.

[64] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s):134:1–134:25, April 2014.

[65] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. *SIGPLAN Not.*, 46(8):35–46, February 2011.

[66] Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Arvind K. Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, pages 194–205, New York, NY, USA, 2016. ACM.

[67] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B. Zdonik. Tupleware: "big" data, big analytics, small clusters. In *CIDR*. www.cidrdb.org, 2015.

[68] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, and Matei Zaharia. Weld: A common runtime for high performance data analytics. January 2017.

[69] Ehsan Totoni, Todd A. Anderson, and Tatiana Shpeisman. HPAT: high performance analytics with scripting ease-of-use. *CoRR*, abs/1611.04934, 2016.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Design and implementation of RDMA-based database operators

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Willi	Roman

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Wettingen, 5.12.2017	R. Willi

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.