Master Thesis

Producing building blocks for data analytics

Author(s):
Koutsoukos, Dimitrios

Publication Date:
2019-09

Permanent Link:
https://doi.org/10.3929/ethz-b-000409593

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
Producing building blocks for data analytics

by

Dimitrios Koutsoukos

Supervised by

Prof. Gustavo Alonso
Dr. Ingo Müller
Renato Marroquin

March 2019–September 2019
Abstract

The ever increasing diversity of data analytics and AI applications has had a tremendous impact on the number of tools that were developed during the past few years. The developers of these tools usually do not spend a lot of time thinking which are the building blocks that lie in their core. As a result, they sometimes have to produce many slightly different versions of the same code fragments. Instead, they could reduce their implementation effort by designing reusable and recomposable building blocks. Then, they could simply orchestrate them in a different order across execution plans. In this thesis, we study the level of granularity of these building blocks. We start with a state-of-the-art high-performance distributed hash join, which we split into smaller operators that have a single functionality. We explore different levels of granularity and study their impact on reusability and performance. Our proposed granularity level yields operators that are reusable and have almost no performance overhead. We present a variety of use cases where we can apply them in modern ML and data analytics scenarios. By using the same operators, the original join algorithm has similar performance and it is even faster in some cases.
Acknowledgements

This thesis signals the end of my MSc studies at ETH and also the end of an important part of my life and I would like to thank everybody that helped me get until here. First of all, I would like to thank Prof. Alonso that gave me the opportunity to work in the Systems Group and for providing insightful comments throughout the course of the thesis. I would also like to thank Ingo and Renato for supervising this thesis, for our detailed discussions even in the smallest detail and for keeping their office door open for my countless questions. This thesis would not have been possible without my friends that although are spread throughout the globe now, are always there for me. So thank you Eleanna, Vassilis, Katerina, Tania, Eleni, Antonis, Giannis, Panagiotis, Orestis, Chloe, Nikos, Marcel and Stratos for all the moments that we have shared together. Finally, I would like to thank my family and especially my parents Anastasia and Giannis for their support all these years.
# Contents

List of Figures 3  
List of Tables 4  

1 Introduction 5  

2 Related work 7  
 2.1 Introduction .................................................. 7  
 2.2 Joins ............................................................. 7  
 2.3 Distributed joins and RDMA .................................. 8  
 2.4 Data analytics systems ....................................... 10  
 2.5 Systems based on abstractions .............................. 11  

3 Background 13  
 3.1 Introduction .................................................. 13  
 3.2 High-performance distributed hash join .......................... 13  
 3.2.1 Remote Direct Memory Access (RDMA) ......................... 13  
 3.2.2 Message Passing Interface (MPI) ............................ 14  
 3.2.3 Algorithmic description .................................... 15  
 3.2.4 Implementation description .................................. 16  
 3.3 Collection Virtual Machine ................................. 19  
 3.3.1 Overview ................................................... 19  
 3.3.2 Collection-oriented IR ...................................... 22  
 3.3.3 Architecture specifics ....................................... 24  
 3.3.4 Parallel Map ................................................ 25  

4 First design 28  
 4.1 Introduction .................................................. 28  
 4.2 CVM on a cluster ............................................... 28  
 4.3 The distributed hash join as a single operator ............... 29  
 4.4 Experiments .................................................... 33  

5 Splitting the join into smaller operators 37  
 5.1 Introduction .................................................. 37  
 5.2 Smaller non-composable operators ........................... 37  
 5.3 Smaller composable operators ................................ 42  
 5.4 Experiments .................................................... 48
6 Reusing join building blocks

6.1 Introduction ......................................................... 53
6.2 Zip operator .......................................................... 53
6.3 Hash-Probe operator ............................................... 55
6.4 Partitioning operators ............................................. 57
6.5 Grid Join ............................................................. 59

7 Conclusion ............................................................. 63

Bibliography ............................................................ 64
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Schematic representation of the distributed hash join with two processes</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>CVM architecture</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>Initial (left) and optimized (right) DAG representation of CVM IR</td>
<td>23</td>
</tr>
<tr>
<td>3.4</td>
<td>Example architecture</td>
<td>25</td>
</tr>
<tr>
<td>3.5</td>
<td>Inside the ParallelMap operator</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>Updated ParallelMap operator</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>The ConcurrentExecutor operator running a distributed MPI application</td>
<td>30</td>
</tr>
<tr>
<td>4.3</td>
<td>Inner DAG containing the MPIJoin operator</td>
<td>31</td>
</tr>
<tr>
<td>4.4</td>
<td>MPIJoin operator specifics</td>
<td>32</td>
</tr>
<tr>
<td>4.5</td>
<td>High level overview of the initial integration</td>
<td>33</td>
</tr>
<tr>
<td>4.6</td>
<td>Execution time of the experiments with uncompressed tuples</td>
<td>34</td>
</tr>
<tr>
<td>4.7</td>
<td>Execution time of the experiments with compressed tuples</td>
<td>36</td>
</tr>
<tr>
<td>5.1</td>
<td>Inner plan depicting the split of the MPIJoin to smaller operators</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>NetworkPartitioning operator specifics</td>
<td>39</td>
</tr>
<tr>
<td>5.3</td>
<td>LocalPartitioning operator specifics</td>
<td>40</td>
</tr>
<tr>
<td>5.4</td>
<td>HashProbe operator specifics</td>
<td>41</td>
</tr>
<tr>
<td>5.5</td>
<td>Inner plan depicting the final split of the MPIJoin to smaller operators</td>
<td>43</td>
</tr>
<tr>
<td>5.6</td>
<td>Zip operator specifics</td>
<td>44</td>
</tr>
<tr>
<td>5.7</td>
<td>HashProbe operator specifics</td>
<td>46</td>
</tr>
<tr>
<td>5.8</td>
<td>ParametrizedMap operator specifics</td>
<td>47</td>
</tr>
<tr>
<td>5.9</td>
<td>Execution time of the final experiments (with local fanout $2^{10}$)</td>
<td>49</td>
</tr>
<tr>
<td>5.10</td>
<td>Execution time of the final experiments (with local fanout $2^{16}$)</td>
<td>51</td>
</tr>
<tr>
<td>6.1</td>
<td>Quadratic loss function as a CVM plan</td>
<td>54</td>
</tr>
<tr>
<td>6.2</td>
<td>Unifying features and labels as a CVM plan</td>
<td>55</td>
</tr>
<tr>
<td>6.3</td>
<td>Calculating the intersection of two sets as a CVM plan</td>
<td>56</td>
</tr>
<tr>
<td>6.4</td>
<td>Matrix vector multiplication as a CVM plan</td>
<td>57</td>
</tr>
<tr>
<td>6.5</td>
<td>Mini-batch creation as a CVM plan</td>
<td>58</td>
</tr>
<tr>
<td>6.6</td>
<td>Sparse matrix vector multiplication as a CVM plan</td>
<td>59</td>
</tr>
<tr>
<td>6.7</td>
<td>Grid join for three relations</td>
<td>60</td>
</tr>
<tr>
<td>6.8</td>
<td>Grid constructed by the join for $k = 16$ processes</td>
<td>61</td>
</tr>
<tr>
<td>6.9</td>
<td>Grid join as a CVM plan</td>
<td>62</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th></th>
<th>Table Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Machine specs for the Euler cluster</td>
<td>33</td>
</tr>
<tr>
<td>4.2</td>
<td>Execution time of the experiments with uncompressed tuples</td>
<td>34</td>
</tr>
<tr>
<td>4.3</td>
<td>Execution time of the experiments with compressed tuples</td>
<td>35</td>
</tr>
<tr>
<td>5.1</td>
<td>Execution time of the experiments with local fanout $2^{10}$</td>
<td>49</td>
</tr>
<tr>
<td>5.2</td>
<td>Breakdown analysis per phase for local fanout $2^{10}$</td>
<td>50</td>
</tr>
<tr>
<td>5.3</td>
<td>Breakdown analysis of local partitioning per phase for local fanout $2^{10}$</td>
<td>50</td>
</tr>
<tr>
<td>5.4</td>
<td>Execution time of the experiments with local fanout $2^7$</td>
<td>51</td>
</tr>
<tr>
<td>5.5</td>
<td>Breakdown analysis per phase for local fanout $2^7$</td>
<td>52</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The ever increasing diversity of data analytics and AI applications has had a tremendous impact on the number of tools that has been developed during the past few years. Even in the simplest cases, data scientists use a variety of toolboxes in order to accomplish their goals. This explosive growth, however, has come at a cost. Most of the tools available have to make a compromise between easy of use, performance, and expressiveness. Attempts as Spark [59], Weld [49], and Voodoo [52] try to create a solid end-to-end pipeline that produces high performance and scalable code but they are not always fully efficient across a variety of hardware architectures.

A research question that lies in the core of these systems is what are the building blocks that they consist of. Usually, developers do not devote a lot of time on this question and, as a result, they implement many different versions of the same code fragments. This, apart from the additional effort that it requires, makes even harder the application of optimizations for a large category of hardware platforms. Instead, if the developers of these systems build recomposable and reusable building blocks, then they can simply orchestrate them in different order across execution plans.

In this thesis, we study the level of granularity of these building blocks. We start with a state-of-the-art high-performance distributed hash join [18] algorithm, which we iteratively split into smaller operators such that we can reuse them in other programs. To do so, we base our integration into the Collection Virtual Machine (CVM) platform. CVM takes as input a program in a Intermediate Representation (IR) and creates an optimized version of it that is highly efficient for the target hardware that the program runs. In its core lies its IR, which maps the input program to simple, low-level, Volcano-style operators, forming a graph of execution that makes possible subsequent optimizations before it produces the final executable.

We choose to explore the building blocks using as basis a distributed hash-join algorithm. We do this for a variety of reasons: First, a join is present even in the most simple queries used in production systems today. Second, the data growth mentioned before has made traditional algorithms running on a single machine obsolete, substituting them with their parallel and distributed counterparts. Third, although networks were considered a bottleneck in data processing previously, recent developments such as Remote Direct Memory Access (RDMA) and very fast networks like InfiniBand [2] has made their bandwidth comparable to the bandwidth of the local memory bus [22]. This poses the need to reimplement our traditional algorithms and systems in a way that they can take advantage of the latest network features. Finally, it is very common for developers to deal with joins as black boxes.

To this end, our first approach involves only a single black-box operator, which we subsequently split into smaller ones. We first introduce operators that are not reusable, as an intermediate baseline to the runtime of the original algorithm and to understand the limita-
tions that the CVM platform puts into the original implementation. We then refine these operators to make them reusable in arbitrary execution plans. Although our proposed granularity level is not the smallest possible one, we can still showcase a variety of applications where we can use our operators. We particularly give emphasis on data analytics and machine learning algorithms.

Using the same operators, the original join algorithm has similar performance and it is even faster in some cases. As a result, we can say that although there is not a correct granularity level, granularity certainly does not hurt performance. With our proposed approach, we can combine high-performance computing with reusability and recomposability.

The rest of the thesis is structured as follows: In Chapter 2 we present the related work. In Chapter 3 we give the necessary background, going into detail about how CVM works and an algorithmic and implementation description of the high-performance distributed join. In Chapter 4 we provide details on the architecture and the integration procedure that we use to run the distributed join as a single black-box operator. In Chapter 5 we split the “fat” operator into smaller ones and we show that we can preserve the original performance of the algorithm. In Chapter 6 we reuse the building blocks that we introduce in the previous chapter in other programs, with a focus on data analytics and machine learning algorithms. Finally in Chapter 7 we conclude the thesis and we give some directions for future work.
Chapter 2

Related work

2.1 Introduction

In this chapter we give related work to our thesis. We start by presenting categories of high-performance joins, where the authors always treat them as black-boxes and they do not pay attention to the granularity of the different phases of the algorithm. We subsequently see that the abundance of data makes clear the need for distributed joins and databases that use the Remote Direct Memory Access (RDMA) technology. Although there are some discrete phases in these type of algorithms, the authors still treat them as black-boxes and they do not study the granularity level of their phases. We then turn our attention to data analytics systems that data scientists heavily use today. The authors of these systems devote some effort to apply optimization techniques from databases like just-in-time query compilation, but they do not study the level of granularity of their building blocks. Finally, we examine systems that use abstractions in their core. Their developers, having understood that they can reduce their implementation effort using recomposable building blocks, are careful in the design of what lies in the basis of these systems.

2.2 Joins

A join operation is one of the most basic and frequently used in databases today. Its semantics are simple. It takes as input two relations $R, S$ and produces the set of all combinations of tuples that have equal values on the common attribute names of both of them. Researchers devote a lot of effort to find faster join algorithms or make optimizations to older ones, mainly for two reasons: First, even the simplest queries in production systems today require a join of two tables. Second, a join is one of the most expensive actions in a database system. The latter gets more important, due to the explosion of data present in modern computing clusters. In fact, sometimes joins are so expensive that researchers introduce metrics to determine if it is safe to avoid them in their data preprocessing and feature selection.

There are a lot of algorithms for performing a join but two of them are prevalent: hash-probe joins and sort-merge joins. Hash-probe joins build a hash table from the one of the two relations and probe the elements of the other relation to the hash table to find matching tuples. They may split the input relations into smaller chunks before building the hash table, to speed up the algorithm’s performance by avoiding cache and TLB misses. On the other hand, sort-merge joins sort both relations on the join attribute. Then, they scan the resulting sorted relations and compare the common attributes for matching tuples.

There is a large dispute in the scientific community on whether hash-probe joins or sort-
merge joins are faster. Kim et al. [36] implement both algorithms in an optimized parallel version that takes advantage of the latest processor features. They report that their hash-probe join implementation achieves a bandwidth of more than 100M tuples per second, whereas their sort-merge one, a bandwidth of more than 50M tuples per second. Therefore the hash-probe join outperforms the sort-merge one. However, they claim that the current architectural trends such as multiple hardware threads on each core (SMT), vector instructions (SIMD) operating on 128-bit vectors, compute density of chip multiprocessors (CMP) and memory bandwidth per core will favor sort-merge joins in the future, especially with the usage of larger SIMD.

Based on this claim, Balkesen et al. [13] revisit the hash-join vs. sort-merge join dilemma. They develop carefully tuned implementations of all state-of-the-art join strategies and they compare their performance under a large grid of experimental factors and parameters. They establish conclusions on how input sizes, degree of parallelism, cache contention, and SIMD influence performance. Based on these, they conclude that radix hash-join algorithms (a type of hash-join that involves partitioning the input relations into small chunks) still outperform sort-merge ones despite the latest architecture trends, and sort-merge joins are preferable only for very large input sizes. They also improve state-of-the-art results 2–3 times for both types of joins.

Since hash-joins are superior in their bandwidth utilization and execution time in the current hardware, researchers study how different optimizations improve their performance. Blanas et al. [23] investigate variants of hash-probe join algorithms on two different hardware configurations. They study each phase of the hash-probe algorithm separately and they discover that a very simple hash join algorithm is comparable to ones with sophisticated optimizations, especially in the presence of data skew. This simple algorithm has a shared hash table and does not partition the input relations. They support the competitiveness of this simple algorithm by saying that modern hardware is very good at hiding cache and TLB misses.

On the other hand, Balkesen et al. [14, 15] overthrow the previous claim by conducting a thorough experimental analysis. They prove that the majority of hardware-conscious algorithms have a better performance than hardware-oblivious ones, despite the latest compute architecture developments. Their work also shows that hardware-oblivious algorithms are competitive under specific workloads and architectures with aggressive simultaneous multi-threading. Furthermore, they investigate through their experiments, parameters that make hardware-conscious algorithms more efficient such as Virtual Memory page sizes, number of threads, number of cores, SMT, SIMD, and prefetching. Their best implementation reaches a throughput close to 200 million tuples per second.

As we can see from all the above works, although there is an increasing need for high-performance joins and there are many works for them, their authors always treat them as black-boxes and they do not pay significant attention to what are their building blocks, but only care about optimizing their runtime.

### 2.3 Distributed joins and RDMA

All of the previous papers conduct the analysis of the join algorithms on a single host that utilizes only its main memory. However, the big data era requires the development of join algorithms that span over thousands of cores and over more than one machines, taking advantage of the computing power available today. To this end, researchers have developed in the previous years distributed join algorithms that use Remore Direct Memory Access (RDMA). RDMA is a communication technology that transfers data directly into the main memory of a remote host. That makes the data transfer much faster than using the Transmission Control Protocol (TCP) because the CPU does not copy data into intermediate buffers.
Using RDMA, Barthels et al. [17] introduce a novel distributed and parallel radix hash join. To overcome the bottleneck that network transfer usually presents, they use Infiniband [2], which is a low-latency, high-throughput network. They evaluate their implementation on two different types of networks. They showcase how the algorithm can use RDMA buffers to partition and distribute the data without sacrificing performance. They also observe that the scalability of the algorithm is dependent on the available processing power and the bandwidth of the network. Lastly, they come up with an analytical model of the algorithm and they show that its predictions are very similar to the true performance observed.

On a subsequent paper [18], they develop a distributed radix hash and sort-merge join on top of the Message Passing Interface (MPI). They give details on how to use MPI for the joins implementation and also present the RDMA effect on the algorithms runtime. Their implementations achieves a throughput of 48.7 billion input tuples per second on 4096 cores. To accomplish there results, they use vector instructions to speed up the processing and access remote data through fast one-sided memory operations. They observe that maximum performance needs the right balance between computation and communication as the network can impose a significant burden on performance. Finally, they develop performance models to predict the algorithms’ performance which proves that the sort-merge join achieves its theoretical maximum, while the radix hash join is far from it, due to the network overhead. Despite that, the hash join still outperforms the sort-merge one. For the scope of this thesis we use the hash join algorithm and its implementation.

Taking into account that the network is still a bottleneck in distributed hash joins, Polychroniou et al. [53] develop the Track Join algorithm. The motivation behind it comes from a case study that the authors did, noticing that the five most expensive queries on a real analytical workload from a large commercial vendor spend approximately 70% of their time on network transfer. Track Join minimizes network traffic by generating an optimal schedule for transferring each distinct join key, after it is aware of the initial locations of the join tuples. Using Track Join, the authors outperform distributed hash join implementations on these real scenarios.

As we see, RDMA plays an important role in the development of distributed algorithms since it is much faster than other alternatives and because of that it has been studied extensively during the past years. Frey et al. [30] analyze the applications where RDMA should be used and they outline hidden costs in its setup. They prove with their experiments that these hidden costs, if not handled with care, can remove any advantage that RDMA has in its performance. Additionally, the authors propose a number of optimizations that make a considerable difference in the overall performance of applications that use RDMA.

Orthogonal to the previous work is the one of Liu et al. [42]. The authors develop a data shuffling operator for parallel database systems that uses RDMA for data exchange. To do so, they take into account a number of parameters in the design space such as the number of open connections and the RDMA transport function used. Their final implementation, which uses the RDMA Send/Receive transport function over the Unreliable Datagram transport outperforms an RDMA-capable MPI implementation approximately 4 times.

Although the previous works present the network as a bottleneck, its speed becomes increasingly better. Database servers today have 56 Gbps FDR InfiniBand and also 100 Gbps EDR InfiniBand devices have appeared in the market [42]. Recently Amazon announced the availability of C5n instances that can utilize up to 100 Gbps of network bandwidth. This forces researchers to rethink how distributed databases work and redesign them in a way that uses high-speed networks. Binnig et. al [22] present with benchmarks the performance characteristics of the latest InfiniBand standards and show that the network bandwidth is getting close to

local memory bus bandwidth. Based on this, they claim that traditional distributed Database
Management Systems (DBMS) architectures cannot take advantage of them. To this end, they
propose a Network-Attached Memory (NAM) architecture and introduce new algorithms for it.
The main idea behind NAM is that it disconnects compute and memory nodes and uses RDMA
for the communication of all nodes. Continuing on this architecture [55], they investigate query
execution for distributed database systems on fast networks. By avoiding the shuffling of inter-
mediate data and by streaming data from memory servers to compute servers at runtime, they
develop a system called I-Store, which outperforms three times a shuffle-based execution model
optimized for RDMA.

Parallel to the previous work is the one of Barthels et. al [16]. The authors present a
detailed background on RDMA and some related concepts such as Remote Memory Access
(RMA) and Partitioned Global Address Space (PGAS). They explore how these technologies
influence distributed join algorithms and transaction processing and they give future research
directions. In addition to that, they introduce new communication techniques that close the
gap between what high-speed networks provide and what data processing systems need.

Following the same vision, Alonso et. al [12] propose DPI: The Data Processing Interface for
modern networks. DPI is an interface that provides abstractions that are adaptable such that
they can take advantage of features of modern networks. Some of these features are RDMA
and in-network processing. The main idea behind DPI is to present data movements as flows.
DPI flows enclose both data movement and data processing and developers can use them to
indicate optimization hints. For example, a developer can decide if he wants to minimize transfer
network latency or maximize bandwidth-utilization. Through initial experiments, the authors
show the potential of DPI for data-intensive applications.

Besides the abundance of data and the need of distributed algorithms that heavily use the
network through RDMA operations, the distributed algorithms are still treated as black-boxes.
We can distinguish more clearly now their phases, since some of the authors try to minimize
network communication, but the granularity level of their building blocks is still not the driving
force of the research conducted.

2.4 Data analytics systems

As we can observe from the previous works, data analytics and big data systems are in the
center of research today. This started with MapReduce [28] and Spark [59], which were among
the first popular systems that could run data analysis programs at scale. However, especially
for Spark, its performance is sometimes very far from specialized handwritten code [1, 29, 47].
A number of works try to fix this by combining Spark internals with MPI. Malitsky et. al [43]
try to bring big data applications closer to HPC ones with the Spark-MPI platform. The core
component of Spark-MPI is the Process Management Interface which is the MPI-equivalent to
the Spark driver. In the same spirit, Gittens et. al [32, 33] propose Alchemist, a system that calls
MPI-based libraries from Apache Spark. Alchemist spawns MPI processes on start and transfers
data from Spark processes to MPI ones. MPI handles the computation and returns the results to
Spark. The authors showcase their platform on truncated Singular Value Decomposition (SVD)
and the conjugate gradient method and they outperform Spark by an order of magnitude or
more.

Orthogonal to the above work is Flare [29]. Flare is a new Spark backend that achieves
high speedups on scale-up architectures for a variety of applications. It is based on query
compilation techniques from main-memory database systems, compiling whole queries instead
of individual query stages to bring Spark code to comparable speed with the best SQL engines.
It also connects with external Machine Learning (ML) systems to facilitate highly optimized
heterogeneous workloads. Building on the idea of query compilation, Crotty et. al \[27\] propose Tupleware. Tupleware is a high-performance distributed analytics system, that takes advantage of the LLVM \[40\] compiler framework. The motivation behind it is that ML and data analytics programs are expressed as sequences of User-Defined Functions (UDFs), where each UDF is a step of the algorithm. Tupleware takes into account data properties and hardware configurations and automatically compiles UDF workflows. Its target audience is people working on compute-intensive memory analytics with relative small datasets (less than a few terrabytes) on small clusters (fewer than 10 nodes).

In this section we observe that besides the need for fast data analytics systems, the authors do not study the building blocks that comprise them. Although they apply optimizations that are known in database systems like just-in-time query compilation, they could heavily improve their performance if they build recomposable and reusable building blocks that are optimized across a variety of hardware architectures.

2.5 Systems based on abstractions

Pirk et. al \[52\] state that systems as Tupleware are designed to generate code for a specific hardware platform, because different architectures apply very different optimization techniques to gain performance. If these systems were transferred to a new hardware architecture, they would have to be rewritten completely to achieve the same performance. To address this issue they build Voodoo, a new intermediate algebra, that is portable to a variety of hardware architectures and expressive enough to incorporate the majority of optimizations that main-memory query processors employ in the literature. Voodoo compiles to OpenCL and the authors use it as a MonetDB backend. Through their experiments they demonstrate that they achieve similar performance to database engines that consist of hand-optimized code and are designed for specific hardware architectures.

Except for the big-data systems mentioned before, data scientists use a vast number of packages and implementations. The majority of them is written in Python and R. For example, BigR \[58\] is a platform that combines an R interface with a Hadoop backend. It thus allows data analysis, using R as a query language, and partitioned execution by running a function on dataset chunks across cluster nodes. It also enables large scale ML, allowing algorithms to train and evaluate on arbitrarily big data.

On the other hand, Scikit-learn \[51\], NumPy \[5\], and Pandas \[45\] are amongst the most popular Python packages utilized for data analytics and ML tasks. However, despite their popularity, they are designed to run on a single machine, usually without any parallelization. To this end, Müller et. al introduce Pydron \[46\], a system to parallelize and execute sequential Python code on a variety of hardware infrastructures. Its API comprises of two Python decorators that either parallelize functions or mark them free of side-effects. By using this semi-automatic parallelization they demonstrate that application developers can take advantage of modern infrastructures without significant overhead. Similar to Pydron is Dask \[54\]. Dask is a NumPy clone that facilitates parallel and out-of-core computation. It encodes parallel algorithms using Python callables and primitive data structures such as dictionaries and tuples and parallelizes the produced code using blocked algorithms and task scheduling.

Instead of trying to optimize only NumPy like Dask, Palkar et. al build Weld \[49\]. Weld is a common runtime for existing data analytics libraries. Its motivation is that data scientists write applications using multiple libraries and functions and this imposes a significant barrier even to simple optimizations. Weld allows libraries to express their computations in a functional Intermediate Representation (IR) and then applies end-to-end optimizations, generates native code and runs the produced program. The authors achieve 23× acceleration on single thread.
workloads and 80× acceleration on eight threads.

Although the previous works apply multiple optimizations and achieve significant speedups for programs related to ML and data analytics, the sudden popularity around Deep Learning (DL) the previous years introduced the need for specialized software for automatic differentiation in computation graphs that takes advantage of the large computing power that GPUs and Tensor Processing Units (TPUs) provide. The most popular packages that offer the aforementioned functionalities are TensorFlow [9] and PyTorch [50].

Although both of these frameworks support distributed training on GPUs across multiple servers, they require users to modify their code heavily and also achieve suboptimal speedup compared to their ideal scaling. Recognizing this problem, Sergeev et. al propose Horovod [56]. Horovod uses a ring reduction mechanism for inter-GPU communication and requires only a few lines of modification to user code for allowing distributed training. It is build on top of MPI and experiments demonstrate that it achieves close to ideal scaling. With the similar aim of trying to bring Deep Learning (DL) closer to supercomputing scale, Ben-Nun et. al introduce Deep500 [21]. Deep500 is a customizable benchmarking infrastructure. Although its main goal is to provide a fair comparison of DL frameworks by comparing customizability, metrics, performance, validation, and reproducibility among them, it also implements a software platform that enables them to use the most powerful supercomputers for arbitrarily large scale workloads.

As we observe the previous works try to bring DL frameworks closer to big-data scale, a similar objective that systems like Dask, Pydron, Voodoo, Weld and Tupleware have for ML frameworks. Taking inspiration from the query compilation that some of these systems propose, Chen et. al build TVM [25]. TVM is a compiler that takes as input the computational graph IR that DL frameworks use and from it generates low-level optimized code for a plethora of hardware backends. To do so, it proposes novel schedule primitives that exploit hardware intrinsics and it also introduces a machine learning based optimization system that explores the optimization search space. Through their experiments the authors show competitive performance with state-of-the-art handtuned libraries for CPUs and GPUs.

All of the above have the objective of providing highly-optimized code for specific frontend languages across specialized hardware infrastructures. On the other hand, there are two projects that try to yield high-performance code on a variety of frontends and backends, namely MLIR [41] and CVM. MLIR is a compiler infrastructure that attempts to unify a variety of frontends used in ML today and to produce code with similar performance to hand-written assembly. Its motivation comes from the TensorFlow ecosystem that has devoted a substantial effort to rewrite part of its codebase to optimize performance for different backends. Its basis is an IR that is subsequently optimized according to the target hardware.

With a similar motive, CVM recognizes that most of the programs that data scientists develop today are “nested transformations of nested collections”. CVM takes an input program written in an IR, then it does a series of rewritings and optimizations and finally expresses it in a executable form that consists of a handful of low-level operators. Since our work is heavily based on CVM, we give a thorough analysis in Chapter 3.

We can observe in this last section that the authors start to realize the importance of recomposable and reusable building blocks and they try to encapsulate abstractions in their work, such that they can reuse algorithm parts that they have previously implemented. This helps them to apply a variety of optimizations and makes their code highly efficient across different hardware platforms.
Chapter 3

Background

3.1 Introduction

In this chapter, we describe the necessary background that forms the basis for the integration of the high-performance distributed hash join into the CVM platform. We split the chapter into two parts. In the first part, we describe the distributed hash join algorithm in detail. Except for the algorithmic part, we also give some information about the Message Passing Interface (MPI) and Remote Direct Memory Access (RDMA) technology, which is essential for the understanding of the join implementation. We closely follow the VLDB paper [18], where the authors originally introduce the algorithm. We give emphasis to potential caveats and limitations that exist in the original implementation and we should resolve in our integration, since the join is part of a system and not run as a stand-alone algorithm. In the second part, we describe the Collection Virtual Machine (CVM) platform. We dive into technical details, as opposed to the previous chapter where we gave a brief motivation and overview, presenting in more depth parts of the platform that we modify or extend later in order to run the distributed join.

3.2 High-performance distributed hash join

3.2.1 Remote Direct Memory Access (RDMA)

Remote Direct Memory Access (RDMA) is a technology that allows directly accessing the memory of one computer into that of another, without any involvement of either one’s Operating System (OS). Since the OS does not know the data transfer that is taking place and therefore the Central Processing Unit (CPU) does not take part in the transfer, it remains available to perform processing. This allows the system to interleave computation and communication and can be very beneficial in massively parallel computer clusters. Another advantage of RDMA is that the host does not need to copy data into intermediate buffers in the network stack but instead it stores the data directly at specific locations in main memory. RDMA communication mechanisms are distinguished into two categories: two-sided and one-sided. The core operations of two-sided RDMA are send and receive, whereas the ones of the one-sided RDMA are read and write.

Two-sided operations have as a prerequisite that both the sender and the receiver must be active for the transfer to complete. The receiver, also known as active target, makes available several RDMA-enabled buffers, where the network card will write the data. These buffers also have associated descriptor elements that the receiver inserts into a receive queue. It is the receiver’s responsibility to allocate buffers of sufficient size and quantity and to make sure that the queue always has available buffers. The sender of a message does not have exact knowledge
of the locations of the receive buffers. It is the target’s network card duty to take the head of
the respective receive queue, verify necessary conditions for the data transfer such as sufficient
buffer size, and then complete the data transfer to that location.

One-sided operations, on the other hand, have the Remote Memory Access (RMA) as their
basis. RMA is a concept that allows access to remote memory regions through one-sided read
and write operations. The process that initiates a request can decide, without any interme-
diaries, where it will save data in the memory of the remote host. In contrast to two-sided
operations, read and write operations do not put any responsibility on the target machine,
making it a passive target. What is happening under the scenes is that these operations simply
move data from one buffer to another. For example, a read operation will make a data transfer
from a remote machine to a local buffer, while a write operation will perform the opposite task.
A consequence of how RMA operates is that, on a remote machine the host has to move data
first into a local main memory buffer before it loads it into a register.

Based on the primitives described above RDMA enables modern low-latency networks to
achieve high bandwidth utilization while keeping the latency low. An example of such a network
is Infiniband [2]. A hidden detail in the whole RDMA procedure is that the program utilizing
an RMA operation has to register the memory it requires with the network card before any
instruction can use it for RDMA transfers. The registration involves pinning a memory part
such that other processes cannot use it or the OS cannot move it to the swap partition. It also
contains memory translations that the network card can access. Introducing a bit of terminology,
this pinned memory that is accessible by RDMA operations is called a memory region.

3.2.2 Message Passing Interface (MPI)

The Message Passing Interface (MPI) is one of the most widely used standards for writing
parallel High Performance Computing (HPC) applications. Although the original target group
of the interface was parallel large-scale systems, programmers use it for code designed to run
from small clusters to supercomputers. One reason behind the popularity of MPI is that it
provides a high-level description and gives the freedom to developers to create different imple-
mentations, optimized for different platforms or supercomputers. For example, Fast One-sided
MPI (foMPI) [31] is an MPI implementation optimized for the Cray XC30 and XC40 systems.1
The two most popular MPI implementations are OpenMPI [6] and MVAPICH [3]. For the scope
of this thesis, due to performance reasons described in the next chapter, we use OpenMPI.

When writing an MPI application, the developer does not take into full account the internal
topology of the distributed environment. It is the specific implementation’s responsibility to
select the most applicable communication approach for each pair of processes. That makes the
application code portable, while simultaneously takes advantage of the hardware the program
uses. The latter is another reason that accounts for the wide usage of MPI in HPC applications.

Diving into some specifics of MPI, the core entity of parallelism of every MPI application is
the process. The programmer determines the degree of parallelism, which is equal to the number
of processes, when dispatching a job. The dispatch system, which is usually a shell script called
mpirun, attempts to hide the differences in starting jobs for various devices from the user. Its
main task is to initialize all the processes that will run the same code on all the machines that
the jobs have been assigned to. What distinguishes processes between them is their rank, an
integer number that is unique to every process. Communication between processes can happen
only when the programmer indicates the rank of the target process.

In the rest of this section, we describe all the one-sided RMA operations that the MPI
standard specifies and the authors in the original paper [18] use in the development of the HPC

---

1https://www.cray.com/products/computing/xc-series
distributed hash join. In order to perform any RMA operation each process has to allocate a memory window, meaning a contiguous section of main memory accessible by other processes through RMA operations. The MPI standard implements this with the MPI_Win_create operation and is a collective call, meaning that all processes execute this instruction, even if one of them does not need to register memory itself.

After the creation of the window and before the program executes any other operation on it, the developer must synchronize the processes with the MPI_Win_fence function. The synchronization ensures that all incoming and outgoing RMA operations will complete at that process before the fence call returns. The time span mediating between two fence function calls is called an RMA epoch. Fence synchronization is also a collective call.

MPI provides two types of target synchronization: active and passive. MPI_Win_fence belongs to the former. Active target synchronization is more appropriate for applications where each process has to undergo the same number of RMA epochs. On the other hand, for applications that have more complicated communication patterns, passive target synchronization is more suitable. MPI_Win_lock and MPI_Win_unlock are two functions that implement this synchronization primitive. The application has to acquire a lock before every RMA operation on a specific window. There are two types of locks: one that provides exclusive access (MPI_LOCK_EXCLUSIVE) and one that provides concurrent access (MPI_LOCK_SHARED). Before the program releases a lock, all pending RMA operations have to return both at the source and the target. To compensate for the synchronization cost, multiple data transfers should be set up per every RMA epoch.

Finally, to read and write from a remote window the developer should use the MPI_Get and MPI_Put functions. To be sure that all the pending RMA operations that the calling process starts on the target window are complete without any need of releasing the lock, MPI has the MPI_Win_flush operation. This function is available only in passive target synchronization. After this call, the program can reuse or read all the buffers used by previous MPI_Get and MPI_Put function calls.

### 3.2.3 Algorithmic description

A join workload consists of two tables where each table entry contains a 16-byte tuple, where the key is 8 bytes and the row ID another 8 bytes. This 16-byte tuple represents every table row. A high-level algorithmic overview of the distributed hash-join algorithm consists of three phases:

1. Histogram and assignment computation
2. Multi-pass partitioning including network transfer
3. Hash table build and probe

In the first phase, the algorithm generates a global histogram of the keys of the data for both relations. This is necessary for determining the size of the communication buffers and the RMA memory windows for communication amongst machines. The join also needs the histogram for computing offsets within the RMA window, where each process has exclusive write access. The latter is crucial to avoid synchronization costs during the execution of the join algorithm. At that phase it also needs to calculate the assignment map. The assignment map is a vector, where each entry contains the process responsible for a data partition.

In the second phase, the algorithm has to partition the data such that each tuple from the inner relation has its matching tuple from the outer relation residing in the same machine. It should also create small partitions of both relations, such that they fit inside the cache. This
step, except for essential for the correctness of the algorithm, is as well advantageous for the build-probe phase, because by fitting the hash table into the cache, it avoids the cost of cache misses. The partitioning fanout determines the number of partitions and their size. The larger the fanout is, the more and smaller partitions we have. However, if there are too many partitions, the program has to access too many random locations in the memory while partitioning the data and this will lead to a significant amount of Translation Lookaside Buffer (TLB) misses. The latter is true only if the number of partitions is bigger than the number of TLB entries [14]. Therefore, we have to tune our partitioning fanout so as to avoid both TLB and cache misses. We can achieve this by following a multi-pass partitioning approach. At each step, the current partitioning phase takes as input the partitions of the previous phase and re-partitions them, by using a fanout that takes into account the TLB and cache entries as constraints. At every partitioning phase, the algorithm uses a different hash function. This whole approach is known in literature as a radix hash join [23].

Considering all the above, the partitioning phase of the distributed join works as follows: In the beginning the program partitions the relation either into a local or an RDMA buffer, based on if later phases of the algorithm will process the data locally or on a remote node. In order to gain performance by interleaving computation and communication, it executes RDMA operations when a partition is ready, while it creates the next partition. If other partitioning phases are necessary, the program executes them locally without the need of involvement of the network. After the execution flow finishes with all the partitioning phases, data is ready for the build and probe phase, and the partitioning phases have ensured the assignment of matching tuples to matching partitions for both relations.

In the third and final phase, the algorithm builds a hash table for each partition of one of the relations. Usually, it uses the smaller relation for this task. Without loss of generality we assume that the inner relation $R$ is the smaller one. For every hash table created from the partition $R_p$, it uses data from the corresponding partition of the outer relation $S_p$ to probe it. That concludes the execution of the join algorithm, which does not include any materialization.

### 3.2.4 Implementation description

This section gives details about the implementation of the algorithm described above. A schematic overview of the algorithm and the implementation is presented in Figure 3.1 for two processes. Note that we do not depict the network and local partitioning of the outer relation for brevity reasons.

For the histogram and assignment computation, each process scans the data currently resilient in its main memory for both the inner and outer relation and computes the histogram for each of them. It combines the local histograms into a global histogram with the `MPI_Allreduce` function using a summation for the reduction, in a manner similar to a map/reduce paradigm. This function, apart from computing the result, is also responsible for distributing it back to all the processes. The assignment of partitions to processes uses a round-robin arrangement. That means that every process with MPI rank $p$ out of total $n$ ranks is responsible for the partitions that satisfy $x \mod n = p$ where $x \in [0, F - 1]$ and $F$ is the partitioning fanout.

Afterwards, each process sums the entries of the global histogram assigned to it to compute its window size. Furthermore, it also computes the private offsets. To do that, it first calculates the starting offsets of each partition in regard to the starting address of the window. Secondly, it computes the relative offsets inside a partition. To accomplish this, it uses the local histogram to compute a prefix sum. Precisely, the `MPI_Scan` function estimates this prefix sum, by calculating the partial reductions of data on a collection of processes according to a user defined operation (handle). In our case each process uses a sum operation and thus the `MPI_Scan` returns the sum
of the entries of the local histograms of the processes 0 to \( i \). Finally, the program computes the private offsets by summing the corresponding base and relative offsets.

After this step, every process knows the partitions for which it is responsible, how much data it is going to receive in the RMA window, and where to write the partitions that it will compute. We depict the different partitions with different colours in Figure 3.1.

For the network partitioning, each process first allocates one window for each relation using the \texttt{MPI\_Win\_create} function. As we mention in Section 3.2.2 this function is a collective call, requiring global synchronization. After the process allocates the windows, it acquires a shared lock (\texttt{MPI\_LOCK\_SHARED}) for all of them. Since the algorithm has computed private offsets inside the window for each process and the accesses are non-overlapping, it can access the respective memory concurrently. The setup phase finishes by allocating a set of communication buffers, with size of 64KB each, into where each process will partition the data. Having done these preliminary steps, the program can do the actual partitioning of the input relations. The partitioning code transfers data with AVX instructions into the local buffers. Once a buffer is full, the process issues an \texttt{MPI\_Put} to transmit the data to the target window. In order to interleave computation and communication, each process has to allocate at least two buffers for each partition, because otherwise it has to wait for the transmission to end to start creating the next partition. When all the buffers have been used and before reusing one of them, it has to make sure that it is safe to utilize them again. The \texttt{MPI\_Win\_Flush} function accomplishes this.
We illustrate the network partitioning phase with the arrows that end up in the window box on Figure 3.1. As mentioned, the partitions can end up either in the same or in a different process.

During this phase, the program compresses the workload of the algorithm that comprises of 16-bytes \langle key, row ID \rangle tuples into 8 bytes. This is beneficial for the network partitioning since it reduces the total amount of the data transmitted by a factor of 2. The compression takes advantage of the fact that radix partitioning groups together keys with \log F identical bits when the identity hash function is used, where F is the partitioning fanout. For the implementation of the algorithm, the authors chose to hash each tuple on the least significant bits on every partitioning phase, therefore these are the ones that are identical. Therefore, at each partitioning phase these bits are redundant for the phases afterwards (either subsequent partitioning or hash-probing) and each process can drop them. Since the original goal of the paper is to scale to thousand of cores and more specifically to a supercomputer with 4096 cores, the minimum network fanout, given one process per core is $2^{12}$. If we make the assumption, that we do not have more than 4 TB per relation then 38 bits are enough to represent the key and the row ID without any loss of information. Summarizing the above calculations, we can compress a tuple into $2 \cdot 38 - 12 = 64$ bits = 8 bytes. Naturally, with smaller payloads for the key and row ID, we can use a smaller partitioning fanout and still preserve the original information.

Thus, if P stands for the payload, meaning the amount of useful data for the key and the row ID, and F is the fanout, to compress the 16 byte tuple the program does the following:

1. First it drops the identical \log F bits by shifting to the right the key.
2. After that, it shifts left the remaining part of the key by \log(P + F) bits, to move it to the leftmost bits.
3. Finally, it adds the row ID, to the previous quantity, ending up with a tuple that has the key in the “left” part and the row ID in the “right” part.

When all the data partitioning is complete, the process can safely release the shared window lock. As mentioned in Section 3.2.2, the release is successful only if all the pending outgoing RMA operations have been completed. After it releases the lock, the program still has to ensure that there are not any incoming RMA operations. It accomplishes this by using an MPI.Barrier. When all the processes move past the barrier, all the partitioning buffers are no longer necessary and the program can release them, signaling the end of the network partitioning phase.

After the network partitioning phase, each process partitions the data again to ensure that the hash tables fit into the cache. The second partitioning as well as the hash-probing phase follow the task queue model. To facilitate that, each phase of the algorithm (histogram computation, network partitioning, local partitioning, hash-probe) is inherited from a Task interface and there exists a queue variable that takes elements of type Task with global visibility.

To start the local partitioning, each process gets the assignment vector that it computes as described previously. Then it executes a loop with length equal to the previous partitioning fanout. For every partition assigned by the network partitioning phase to the process currently running the above loop, the process gets the corresponding partitions of the inner and the outer relation and adds a local partitioning task to the queue. Once each process finishes the loop, the algorithm has created all the local partitioning tasks and therefore it can release all the memory related to the histogram and network calculations except for the window itself.

At this point, it is clear, that since the compression removed some bits of the key, the hash probe phase cannot materialize the output of the join without any loss of information. However, we can take advantage of the following observation: At every point in the previous step that the algorithm adds a local partitioning task to the task queue, the current value of the loop iterator is equal to the bits that the compression discarded previously. Therefore, we just have
to pass along this information to the local partitioning task such that the hash-probe phase is able to perform the materialization. We portray the local partitioning phase with the boxes that are inside the windows and are later split into smaller boxes in [Figure 3.1].

When the loop that creates the local partitioning tasks finishes, each process starts executing them. The local partitioning tasks do not contain any MPI instructions or network transfer but the general structure resembles the one of the network partitioning. The partitioning begins by computing the local histogram of the partitions received for both the inner and the outer relation. Afterwards, by taking into account the new partitioning fanout $F$, it computes the offsets of every newly created partition by calculating the prefix sum of the histogram of the previous partitions. Namely, for a partition $p$ the prefix sum $p_{si}$ is equal to $\sum_{k=0}^{i-1} \text{hist}[k]$, where $i \in [0, 2^F - 1]$. At this point the local partitioning also adds padding to ensure that every partition is cache-line aligned. When the partitioning phase calculates the offsets, it can start the partitioning by using an output buffer, similar to the AVX partitioning done in the previous phase. However, the output buffer is not of fixed size, but of size $2^F - 1$ such that it can hold all the data. After each process finishes executing a partitioning task, it adds a hash-join task to the queue, that has as information the newly created partitions and their size. To facilitate the materialization, each local partitioning task also has to pass along to the hash probe tasks the network bits that it inherited when it was created.

After the program removes and executes all the local partitioning tasks from the queue, it no longer needs the windows and it can safely free the memory it allocated for them. Then it can start running the hash-probe tasks one by one. The hash-probe implementation used in running algorithm was introduced by Balkesen et al. [14]. Its main goal is to provide cache-friendly access by reducing the cache misses both in the build and the probe phase. To do that, the authors implement the main hash table as a contiguous array of buckets, closely resembling a linked list implementation. Precisely, they have a table that has the next position of a hashed element and a table that holds the next available position that the hash-probe phase will use to hash an element or the end of the list of a hash. Once they construct these tables in the probe phase, they traverse the “linked list” or “chain” that an element hash creates, until they get to the beginning of the list. The previous description makes clear that this implementation is specific and works on partitions, and the program needs to know the partition size in advance. We demonstrate the hash-probe phase in the middle of [Figure 3.1] where the algorithm creates hash-tables (depicted in black boxes) from partitions of the inner relation and probes them with partitions from the outer relation. After the program completes the last hash-probe task amongst all processes and the task queue is empty the join algorithm finishes its execution.

### 3.3 Collection Virtual Machine

#### 3.3.1 Overview

The Collection Virtual Machine (CVM) is a platform trying to fill the gap in the landscape of tools that data scientists use today. Since most of these tools do not provide high-level interfaces and data independence but instead are often designed only for a specific target hardware, data scientists constantly switch tools depending on their needs, leading to significant time spent on reconfiguring and reimplementing their preprocessing pipelines and models. In fact, despite the growth of the available data, it is surprising that some of the most popular packages in the Python ecosystem that data scientists work with today (e.g. scikit-learn, pandas) are built to run on a single machine, restricting the amount of data that they can handle to tens of gigabytes at the most.

Recognizing the problems mentioned above, CVM tries to offer a more unified programming
Figure 3.2: CVM architecture
experience. To do so, it builds upon the following realization: All the types of data analysis today, on almost all the hardware platforms can be described as (nested) transformations of (nested) collections. The reader might be surprised by this claim, but the explanation is twofold. On one hand, all types of data analysis, whether graph analysis that works on a set of edges and vertices or machine learning that works on vectors and matrices of numbers, operate on “collections” of “tuples” of “atoms”. On the other hand, implementations on a very abstract level are transformations of individual atoms in the case of inner loops or nested compositions of transformations in the case of the orchestration code around loops.

Based on this abstraction, CVM supports data analysis on a variety of frontends and backends. On a very high level we could compare it with a compiler, as it takes as input a program, rewrites it in an Intermediate Representation (IR), which it then optimizes though a sequence of rewritings and it finally expresses it with a batch of low-level operators that it lowers into an efficient executable form. What is clear from the above is that the IR lies in the core of the CVM platform. It consists of “instructions”, which are transformations of collections. CVM calls these “operators”. The IR is generic enough such that it can express both high or low-level operators but at the same time a CVM user can easily extend it with new operators, making automatic optimizations and low-level implementation techniques possible. These optimizations, combined together with manual tuning whenever necessary, make the produced code highly efficient.

We present the CVM architecture in Figure 3.2. The arrows in this figure represent the program’s flow of execution. We will use similar arrows for representing execution flow throughout the course of this thesis. On the top level we observe numerous examples of frontend languages and interfaces that work with collections. Initially, CVM translates these frontends into IR without intervening substantially with the program. The platform allows frontends to define new high-level operators or new collection types. In the case of ML and graph analytics, the frontends may consist of libraries expressed in other frontend languages and packages in CVM IR, such that other frontends can reuse them.

After CVM converts the program into IR, it performs a series of rewritings to bring it into an optimized and subsequently an executable form. The user selects the rewritings and optimizations that he wants to apply. These optimizations are highly reconfigurable. For example, the platform may convert a high-level operator to a sequence of low-level operators. Some of the optimizations already present is a rule to inline LLVM functions whenever these are present, a rule to gather filter operators together and then push them as down as possible in the set of the flow of the program to reduce the input size and a rule to split the input of the program and parallelize it such that it can run the program through concurrent routes of execution, e.g., using threads. A Directed Acyclic Graph (DAG) that shows the data flow between operators represents the IR.

To demonstrate all these in action, we present an example of a JSON serialization of an IR in Listing 3.1 and a DAG representation of the previous serialization in Figure 3.3. We use vector arrows throughout the thesis, as the ones in Figure 3.3, to represent CVM operators that connect with each other in the presented DAG. That also means that a downstream operator respects the semantics and types that the upstream operator returns. In Listing 3.1 we see the operators that comprise the plan, their ID which represents their order of execution, their predecessors in the program flow and finally their output type. On the left part of Figure 3.3, we observe how the input program of the previous serialization looks like in a DAG representation. Indeed, as the input code just creates a range of numbers, the IR contains the minimal operators to run this kind of program. On the right part, after multiple optimizations, such as the parallelize rule we mention before, the output program seems very different than its left counterpart. The DAG representation is inspired by the AST representation made by compilers before they produce
their intermediate and final code. It is clear from this figure that all rewritings use the same IR as their input and output.

After CVM performs all the rewritings that the user selects, the target backend runs the program. This includes a final lowering pass, that the backend executes. Examples of this final pass are just-in-time compilation to native machine code or translation into precompiled executable operators.

```json
{
  "operators": [
    {
      "id": 0,
      "predecessors": [],
      "op": "parameter_lookup",
      "output_type": [{"type": "long"},
                      {"type": "long"},
                      {"type": "long"}]
    },
    {
      "id": 1,
      "predecessors": [0],
      "op": "range_source",
      "output_type": [{"type": "long"}]
    },
    {
      "id": 2,
      "predecessors": [1],
      "op": "materialize_row_vector",
      "output_type": [
        {"type": "array",
         "dim": 1,
         "layout": "C",
         "output_type": [{"type": "long"}]
        }
      ]
    }
  ],
  "outputs": [{"op": 2, "port": 0}],
  "inputs": [{"op": 0, "op_port": 0, "dag_port": 0}]
}
```

Listing 3.1: IR example

### 3.3.2 Collection-oriented IR

In this subsection we give a formal definition of CVM and its IR. We also describe categories and examples of collection types and instructions and finally we dive into some detail about the architecture style and techniques that the “lowerings” part of the platform uses.

CVM has registers that store “collections” and executes linear sequences of “instructions” called “programs”. The platform describes collections with the following recursive definition:

$$\text{item} := \text{atom} | \text{tuple of items} | \text{collection of items}$$

An atom is an undividable value of a particular domain, a tuple is a mapping from a domain of names to items and a collection is the generalization of any data type holding a finite, homogeneous multiset.

CVM uses collections as its core data abstraction and it separates their types into three categories: abstract collection types, physical collection types, and custom collection types. It uses abstract collection types to represent abstract data types. Some examples of abstract
Figure 3.3: Initial (left) and optimized (right) DAG representation of CVM IR
collection types CVM defines are: Set, Bag, Seq (for sequences), kDSeq (for k-dimensional sequences). The platform uses physical collection types to express physical data layouts. The basic building blocks of physical data layouts is the generic type Vec (standing for vector), that CVM uses to represent an array of items in a single contiguous block of memory and tuple, that it uses to represent fixed-width records with ordered fields (like C structs). With these basic types, the user can implement any row-store and row-column-store layout of relations as array of structs and struct of arrays, respectively. The backend determines the actual physical representation of these types. Finally, the platform introduces custom collections times to support any physical format and data structure.

Instructions perform transformations of collections into other collections. They can be distinguished to high-level, control flow, and low-level instructions. Their level of abstraction corresponds to the level of abstraction of the collections that they work on. High-level instructions are generally present in the initial translation of a user-defined program into CVM IR. Some examples of them are MMMult (matrix-to-matrix multiplication) and the Map function. Control flow instructions try to overcome the lack of traditional control flow operators (e.g., jumps), forbidden by the definition of CVM IR. Examples of this type of instructions are LOOP and ParallelMap. Lastly, low-level instructions serve as specific building blocks of different backends. The philosophy behind them is to present a sophisticated combination of simple operators rather than a simple combination of sophisticated operators. Examples of them are a RowScan operator and a MaterializeRowVector operator.

As a last part of this subsection, we explain the philosophy of the “lowerings” part of CVM. The main motivation is to use techniques that avoid unnecessary materializations. To this goal, the platform extracts tree-shaped parts of a program and converts them to pipelines of operators that pass individual tuples between them. This is inspired by the database community and matches with Volcano-style iterators. Volcano-style iterators have a “pull” philosophy behind them. The execution of the program starts by the bottom of the tree which forces the upper parts to run their code and pass individual tuples through a next interface every operator implements. The next interface is one of the three that are obligatory for every operator where the other two are open and close. The above execution strategy forces the platform to connect the operators in a way that their respective types match. To facilitate this, the backend has a type inference module, that first infers the types of the individual operators and then also checks the output program for type consistency.

As many modern database systems do to permit low-level optimizations across operators, each pipeline is just-in-time compiled to native machine code. The inputs and outputs of every pipeline establish points in the program where materialization is necessary. Nonetheless, pipelines are built such that they can operate on partial results, whenever this is possible, which allows for streaming blocks of tuples between them.

### 3.3.3 Architecture specifics

In this subsection we elaborate on the general overview of the architecture given in Section 3.3.1 by providing a specific example with a Python frontend and a C++ backend. We follow the MSc thesis [11], that originally introduces the architecture we describe.

On the frontend side, the user writes his data flow program in Python, using a functional programming approach, with functions like map, reduce, and filter. CVM translates the frontend program to its IR, which it represents as a DAG of operators. Throughout the course of this thesis, we refer to the terms DAG or plan interchangeably. Then, CVM splits the DAG into pipelines, whose ends are necessary points of materialization, and serializes it into a JSON format. In case of User-Defined Functions (UDFs), it generates LLVM IR using the Numba
Python package and it embeds directly the code into the DAG. At this point the control flow is passed on to the backend, which is responsible to transform and optimize this DAG and to produce the final C++ code from it. It then compiles and links this code into a shared library. The control flow returns to the frontend, which calls a known entry point function that the backend creates with name mangling. With this function call, it executes the program that the backend generated, receives its results and returns them to the user. A caching mechanism reuses previously compiled programs, in case these exist. A schematic overview of the specific architecture presented is depicted in Figure 3.4.

3.3.4 Parallel Map

A necessary prerequisite to run the HPC join inside CVM even on a single machine with multiple threads is to have some form of parallelization in the backend. The control flow operator ParallelMap is responsible for that. Following the definitions introduced in Section 3.3.2 ParallelMap follows the Map semantics, but unlike Map, it takes a function comprised of functions instead of just a plain function when it computes its output. This operator also makes clear to the backend that it has to execute the inner plan in parallel on different inputs, produced by splitting the original input in parts. In the case of the specific architecture described in the previous subsection, the backend uses OpenMP for the parallelization.

To explain exactly how the parallelization works, we have to delve into details about the specifics of ParallelMap. The operator internals are shown in Figure 3.5.
As every Volcano-style operator, `ParallelMap` has to implement three methods, namely `open`, `next`, and `close`. In the `open` method it opens and initializes the upstream operator in the IR DAG. In the `close` method it closes the upstream operator. The `next` method is the one that produces the actual results. More specifically, it executions an inner plan. To run this inner plan, it has a function that accepts a function. The code where this function is called is inside a `#pragma omp` frame, which itself is inside a loop that iterates for all the input. At the end of the function `ParallelMap` sets an iterator to the beginning of the vector that contains the results. Then it returns to the `next` function, which passes the results one-by-one to the downstream operator.

```cpp
// ** ParallelMapOperator **/
auto op_1 = makeParallelMapOperator<tuple_3>(&op_2, execute_tuples);

Optional<tuple_3> execute_tuples(Optional<tuple_0> input_0)

/** ConstantTupleOperator **/
 auto op_2_1 = makeConstantTupleOperator<tuple_0>(input_0);
/** RangeSourceOperator **/
 auto op_1_1 = makeRangeSourceOperator<tuple_1>(&op_2_1);
/** MaterializeRowVectorOperator **/
 auto op_0 = makeMaterializeRowVectorOperator<tuple_3, tuple_1>(&op_1_1);
/** collecting the result **/
 op_0.open();
 auto result = op_0.next();
 op_0.close();
 return result;
}
```

Listing 3.2: Generated code of ParallelMap running an inner plan
We can inspect a simplified example of generated code that showcases how \texttt{ParallelMap} runs an inner plan in \texttt{Listing 3.2}. As we see, \texttt{ParallelMap} is an operator that takes a function (the \texttt{execute_tuples} one) as a parameter. Besides that, we can also look at the implementation of most of the things that we discuss in the previous subsections. For instance, we can examine the operator interface and more specifically that the program runs as a sequence of operators. We can also observe the type that an operator gets as input or returns as output in the form of a struct called \texttt{tuple}. For example, the \texttt{MaterializeRowVector} operator gets as input a struct called \texttt{tuple_1} and returns a \texttt{tuple_3}, which is also the return type of the function \texttt{execute_tuples}. In addition to that, we can inspect the Volcano-style operators in action, as in the end of the function the last operator calls its \texttt{open}, \texttt{next} and \texttt{close} functions, which force the previous operators to do their respective computations. Finally, we observe that the operators get their upstream operator as a reference parameter. This creates the need for a glue operator, that is at the beginning of every plan. The task of this operator is to get the input of the program or function and return it to the next operator. This allows subsequent operators to take operators as parameters, as reading from a parameter has a different interface than reading from an operator. If CVM does not encapsulate the input reading in an operator, all operators would need to be able to deal with it as well. That would be more cumbersome than introducing this glue operator, which CVM calls \texttt{ConstantTupleOperator} or \texttt{ParameterLookup}.
Chapter 4

First design

4.1 Introduction

In this chapter we present the design choices and the architecture that we use for the initial integration of the HPC distributed hash join into the CVM platform. The goal of this design is to preserve the original performance of the join algorithm. We start by sketching how we can run an MPI application inside the ParallelMap operator. Afterwards, we describe the necessary requirements for running a distributed MPI application and how we can run a CVM program on a cluster. Having acquired all the necessary background to run a distributed CVM application, we continue by portraying the rewriting pass performed by the optimizer as well as the form of the simplest final DAG that we use to run the join. We conclude the chapter by presenting the results of our experiments showcasing that the performance of the original join code is preserved.

4.2 CVM on a cluster

From Section 3.3.4 it is clear that a first step towards running CVM code on a cluster would be to replace the OpenMP parallel code with corresponding MPI code. However, this presents a significant difficulty due to the difference in semantics between OpenMP and MPI. Concisely, the difference is the following: On one hand, to run parallel OpenMP code, the developer must write the appropriate instructions and then (at least for C/C++) he must compile his program with the -fopenmp flag, which we can easily do for the shared library object that is just-in-time compiled for the architecture we describe in the previous chapter. On the other hand, to run parallel MPI code, as described in Section 3.2.2, the mpirun script must take an executable program (and not a shared library object) and run this with the desired parameters. This reveals the need for some glue code that serves as a bridge between the ParallelMap and the inner plan that ParallelMap runs.

We encapsulate this glue code into a program that we call MPIWorker. MPIWorker is an executable that we precompile and mpirun starts on each rank. During runtime, we spawn mpirun using POSIX and we communicate data back and forth from the MPIWorker by serializing them to files, in a location that the program knows. In our case, MPIWorker runs in parallel the inner plan that ParallelMap run before inside the OpenMP instructions. To do that, the backend compiles the inner plan as a separate shared library object at runtime, in a location that it is known to the ParallelMap operator. After MPIWorker finishes with the execution of the inner plan, the control flow passes to the ParallelMap which reads the results from the files, encapsulates them in the right CVM abstraction and concludes its execution.
Now that we have the basic structure to run a parallel application using OpenMPI, we need to apply only minor changes to run the same application in a distributed mode. Specifically, to run a general distributed OpenMPI application, one needs passwordless SSH access inside a cluster connected in the same network and a shared file system such as a Networked File System (NFS) where the MPI program resides. Based on these, we should place the MPIWorker, the compiled inner shared library object and the input and output data into the NFS store, so that they are visible from the ParallelMap operator for all the nodes. After doing all that, we can successfully run CVM code on a cluster and we are almost equipped with all the necessary tools to run the HPC join inside CVM. An overview of our current architecture can be viewed on Figure 4.1.

4.3 The distributed hash join as a single operator

Despite the architecture sketched in the previous subsection, there is still a small piece missing to be able to run the distributed hash join inside CVM. This is the ability to run parallel code in an operator that accepts two upstreams, as in relational algebra a join is a two input function. To this end, we introduce the ConcurrentExecutor operator. Its semantics are similar to the ParallelMap operator, with two differences. The first one is that it guarantees concurrent execution, which is necessary for the side-effects of the MPI functions (e.g. the ones that transfer data through the RMA window). The second one is that it has two upstream operators instead of one. That means that now we have to serialize two upstreams that the
MPIWorker gives as input to the inner plan into a file. We can see a schematic depiction of the ConcurrentExecutor operator running a distributed MPI application in Figure 4.2.

Figure 4.2: The ConcurrentExecutor operator running a distributed MPI application

With the implementation of this operator, we have no open ends towards the integration of the join algorithm. For the first version, we run the whole join code inside just a single operator called MPIJoin, since our goal is to keep the original performance of the join algorithm. Of course, this requires to have a set of operators suitable to preceed and succeed this operator in the inner plan, forming a DAG. We can see a DAG example in Figure 4.3. We explain the need for the ParameterLookup operators in Section 3.3.4. The DataGenerator is an operator that we introduce specifically for our integration. Its goal is to produce similar workloads to the RDMA verbs paper [17]. To this end, it consumes all of its upstream and it then produces the required data. It returns this data one tuple at a time, through its next interface. Finally, the MaterializeRowVector operator is necessary for the materialization of the join. It is also a mandatory operator at the end of every CVM plan. The operator pushes into a struct as many tuples as the upstream operator returns to it, introducing a level of nesting to its output.

We should note here a limitation that is present throughout the course of this thesis. More specifically, the types supported for the join integration will be the ones that the authors use.
in the original algorithm, meaning a pair of longs for each input tuple. This also facilitates the comparison with the state-of-the-art implementation.

After describing what the inner plan contains, we give details on exactly how the MPIJoin operator works. We can see a schematic representation of the internals of the operator in Figure 4.4. As shown, the execution starts in the open function, where we materialize the upstream data from both upstreams into two STL vectors. This is necessary because the initial join implementation that we compile into a shared library object at runtime, requires all the algorithm input before it starts its execution. Then, through an entry function to the HPC join we “feed” the algorithm with the data, perform the join, and return the results to the open function.

At this point, we elaborate a bit on how we materialize the join output. In order to do that, we should decide on how we save the materialization result, but first we have to undo the compression introduced in the network partitioning phase. To store the materialization result, since our implementation is in C++, we choose a Standard Templated Library (STL) vector. To undo the compression, once we find a matching key we have to first shift it right by \( \log(F + P) \) bits, where \( F \) is the network partitioning fanout and \( P \) the payload bits. Then we must shift it left by \( F \) bits and finally we must add the partition bits that the compression removed in the network partitioning phase and are now a member of every hash-probe task as we describe in Section 3.2.4. Due to the workloads that we introduce the algorithm with and some limitations in the serialization of the data in the ConcurrentExecutor, instead of pushing tuples that contain the key and the row ID of both relations to the result vector, we only push the key.

After the join algorithm finishes, we initialize an iterator to the beginning of the results vector and the open function finishes its execution, passing the control flow to the next function, which
iterates though the vector and returns the results to the downstream operator one tuple at a
time by wrapping them around the right level of abstraction.

As we can observe from the previous description, although this integration is simple enough,
it has some obvious flaws. First of all, it violates the philosophy of CVM to have simple operators
instead of sophisticated ones. This is essential for the operator composability and reusability in
other plans. With our current integration, we cannot run any other plan except for the HPC
join. Secondly, it introduces a lot of unnecessary materialization and dematerialization inside
the MPIJoin operator, because the operator itself cannot run on partial results, but requires
the full upstream data. We resolve both these issues when we split the MPIJoin operator to
smaller ones in the subsequent chapter of the thesis.

As a high-level overview, we show in Figure 4.5 how our initial architecture looks like. We
can see with the thick arrows from left to right how the control flow and the data pass among the components and with the dashed arrows from right to left how they return through the same components until the get to the ConcurrentExecutor which triggers the execution in the first place.

![Diagram](image)

Figure 4.5: High level overview of the initial integration

### 4.4 Experiments

In this section, we present the experimental results of the initial integration that we outline in the previous part of the chapter. We run all the experiments for the scope of the thesis on the Euler cluster, which has 8 machines with the specs described in Table 4.1.

<table>
<thead>
<tr>
<th>Euler cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUs</td>
</tr>
<tr>
<td>Cores/Threads</td>
</tr>
<tr>
<td>RAM</td>
</tr>
<tr>
<td>L1 Cache</td>
</tr>
<tr>
<td>L2 Cache</td>
</tr>
<tr>
<td>L3 Cache</td>
</tr>
<tr>
<td>InfiniBand</td>
</tr>
</tbody>
</table>

Table 4.1: Machine specs for the Euler cluster

We utilize every machine fully, when we use it for an experiment. Based on the description of the previous section, throughout our experiments we use all 8 machines as MPI workers and before we spawn the workers, we use one as a driver. Because the driver does not actively wait for the results, this approach does not cause any performance degradation.

Due to the network partitioning, which exhibits a small variation in its time execution, the numbers that we show are averages that we obtain by running the respective experiment 5 times. We do not report standard deviations as they account for less than 1% of the overall execution time. Our workload for the rest of this and the next chapter consists of relations with 2048 million tuples each, similar to the workloads that the authors follow in Figure 7(a) of [17]. As we mention before, we construct exactly the same workloads with the original code in the DataGenerator operator.

Our experimental procedure follows an incremental approach as we try to get a comparable baseline of the original codebase with our integration. We start by taking the code provided from the authors for the original RDMA verbs distributed join implementation [17], as well as the one of the MPI implementation [18], which is considered state-of-the-art. The RDMA implementation follows the general pattern of the algorithm described in Section 3.2.3 but it

---

1https://www.systems.ethz.ch/node/334
does not introduce compression in the network partitioning phase. It also uses specific RDMA verbs code for the network transfer instead of MPI functions. We present our results for the uncompressed tuples in Figure 4.6 and in more detail in Table 4.2.

<table>
<thead>
<tr>
<th>machines</th>
<th>SIGMOD paper [s]</th>
<th>SIGMOD code [s]</th>
<th>VLDB code [s]</th>
<th>VLDB code w/window alloc [s]</th>
<th>VLDB code w/window alloc w/materialization [s]</th>
<th>CVM integration [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>11.16</td>
<td>11.38</td>
<td>18.18</td>
<td>19.77</td>
<td>23.41</td>
<td>Out of memory</td>
</tr>
<tr>
<td>3</td>
<td>8.68</td>
<td>9.00</td>
<td>12.87</td>
<td>14.088</td>
<td>17.22</td>
<td>17.15</td>
</tr>
<tr>
<td>4</td>
<td>7.19</td>
<td>7.41</td>
<td>10.21</td>
<td>10.98</td>
<td>12.79</td>
<td>12.84</td>
</tr>
<tr>
<td>5</td>
<td>6.09</td>
<td>6.31</td>
<td>8.36</td>
<td>9.03</td>
<td>10.71</td>
<td>10.77</td>
</tr>
<tr>
<td>6</td>
<td>5.36</td>
<td>5.53</td>
<td>6.92</td>
<td>7.54</td>
<td>9.10</td>
<td>9.09</td>
</tr>
<tr>
<td>7</td>
<td>5.02</td>
<td>5.09</td>
<td>6.32</td>
<td>6.89</td>
<td>8.39</td>
<td>8.34</td>
</tr>
<tr>
<td>8</td>
<td>4.46</td>
<td>4.64</td>
<td>5.64</td>
<td>6.02</td>
<td>6.94</td>
<td>6.96</td>
</tr>
</tbody>
</table>

Table 4.2: Execution time of the experiments with uncompressed tuples

In the first column of Table 4.2 we provide the number of machines that we use for a specific experiment. Then we present the original numbers reported in the SIGMOD paper [17] for a join with two relations with 2048 million tuples each. Afterwards, we run the code on the exact same hardware as the one used for the original numbers and we report the execution times. Our recorded times are a bit slower than the ones in the paper, but we can explain this with some patches done in the machines for the Spectre and Meltdown vulnerabilities [8] and more specifically the CVE–2018–3646 patch which introduces a bit of a slowdown to the executed
Subsequently, we modify the MPI implementation and remove the compression that the algorithm carries out in the network partitioning phase so that we can compare it with the RDMA verbs version. The execution times are much larger than the SIGMOD code, because of the network partitioning phase. More specifically, the authors in their original code use foMPI [31], an MPI implementation that introduces fast-one sided RDMA operations, which the join implementation heavily uses. This MPI implementation can only be used on Cray machines. On the other hand, we use OpenMPI 3.1.4 which does not interleave computation and computation.

In the next two columns of Table 4.2, we add some necessary features that the join requires, for which the authors do not report the execution times in the SIGMOD paper [17]. This happens either because the corresponding phase is not a part of the algorithm (in the case of materialization) or because the corresponding phase uses some other technology that the paper did not report its execution time (in the case of the allocation of RDMA buffers). It is normal that when we add more complexity to the original algorithm the execution time is increased. We can note that the biggest overhead comes from the materialization, which pushes into vectors 2048 million tuples, as the workload of the algorithm for the two relations has a 1-on-1 match.

After we include the window allocation and the materialization times in the original implementation we can compare the original codebase with our integration by looking at the two final columns of Table 4.2. As we see, our integration preserves the original performance. We cannot report the numbers for the 2 machine case, because the machines do not have the required memory to run the corresponding experiments. That happens because we keep the output of the join twice in our integration, once at the STL vector in the shared library object and once in the MaterializeRowVector operator.

<table>
<thead>
<tr>
<th>machines</th>
<th>VLDB code [s]</th>
<th>VLDB code w/window allocation [s]</th>
<th>VLDB code w/window alloc [s]</th>
<th>CVM integration [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>11.04</td>
<td>11.79</td>
<td>13.12</td>
<td>13.05</td>
</tr>
<tr>
<td>3</td>
<td>7.75</td>
<td>8.33</td>
<td>9.40</td>
<td>9.40</td>
</tr>
<tr>
<td>4</td>
<td>6.22</td>
<td>6.60</td>
<td>7.18</td>
<td>7.22</td>
</tr>
<tr>
<td>5</td>
<td>5.09</td>
<td>5.42</td>
<td>5.97</td>
<td>6.04</td>
</tr>
<tr>
<td>6</td>
<td>4.41</td>
<td>4.72</td>
<td>5.23</td>
<td>5.23</td>
</tr>
<tr>
<td>7</td>
<td>3.88</td>
<td>4.16</td>
<td>4.70</td>
<td>4.65</td>
</tr>
<tr>
<td>8</td>
<td>3.36</td>
<td>3.55</td>
<td>4.09</td>
<td>3.88</td>
</tr>
</tbody>
</table>

Table 4.3: Execution time of the experiments with compressed tuples

Finally, in Table 4.3 and Figure 4.7 we observe the results for the compressed tuples case. Our observations stay the same, namely that materialization puts a significant burden in the execution time and that the CVM integration preserves the original performance. As we can observe, by compressing the tuples the algorithm has a much smaller execution time since the network traffic is cut down to half. This makes the running time comparable to the original SIGMOD code that does not introduce any compression.

---

2This is the MPI implementation among the many we attempted that produces the fastest execution times.
Figure 4.7: Execution time of the experiments with compressed tuples
Chapter 5

Splitting the join into smaller operators

5.1 Introduction

As observed in the previous chapter, by running the join as a single operator we preserve the original performance, but we are restricted to running this exact join algorithm. There are even cases where the extra materialization imposes such a memory burden that we cannot run the algorithm for a specific workload. We also violate basic CVM principles, i.e., building small composable operators instead of large sophisticated ones and avoiding unnecessary points of materialization. To this end, in this chapter we split the large MPIJoin operator into smaller ones. Initially, we introduce a split that does not use the CVM abstraction for the data that is “flowing” between operators and we do not produce operators that are as minimal and reusable as possible. We do this to get an intermediate baseline to the runtime of the original algorithm and to understand the limitations that the CVM platform puts into the original implementation. Based on the observations that we make, we create a second split that “thins” the operators even more and as we show in the next chapter, makes them reusable in other programs, mainly for data analytics algorithms that data scientists heavily use today. As we see at the end of the chapter, we do not only keep the original performance, but in some cases we also improve it.

5.2 Smaller non-composable operators

There are numerous possibilities on how we can split the MPIJoin operator into smaller ones. A “natural” split is to produce an operator for every phase of the original algorithm. We could even split even more some phases into their individual components. For example we could separate the histogram and prefix computation from the partitioning operators. However, due to time constraints, we do not produce the smallest possible granularity level and we make the decision that the histogram calculation as well as any prefix or assignment computation, which the subsequent partitioning operator needs, are part of the partitioning operator itself. Based on this, we partition the MPIJoin into the following operators: NetworkPartitioning, LocalPartitioning, and HashProbe operator. The corresponding inner plan, which the ConcurrentExecutor operator runs, is depicted in Figure 5.1.

A first advantage of splitting the MPIJoin into small operators is that now we do not have to compile the join separately into a shared library object. The corresponding code lies instead in the compiled inner plan because the logic that was before in the shared library object is now part of the operators.
In the rest of this section we present the specifics of each operator and we outline potential advantages and disadvantages of our approach.

We start by presenting the rationale that the NetworkPartitioning contains, as we explain the DataGeneration and ParameterLookup operators in Section 4.3 and Section 3.3.4 respectively. As we see in Figure 5.2, the operator starts its execution by consuming all the upstream data to compute the local histograms and then combines them into a global histogram by exchanging them over the network. It then computes the offsets and the assignment of partitions to nodes and does the network partitioning using the RMA window. We give more details about what is happening in these phases in Section 3.2.4.

In the next function we introduce a small algorithmic change in the original algorithm. Instead of running a loop that spans the partitioning fanout and checks the assignment vector for the assignment of partitions to nodes, we calculate a closed formula that does the same.
More specifically, the partition ID that the algorithm assigns to a node follows the sequence:

\[ p_{id} = r + i \cdot n, \quad p_{id} < 2^F, \quad i \in \mathbb{N}_0, \quad i \in [0, \left\lfloor \frac{2^F}{n} \right\rfloor] \]

where \( p_{id} \) is the partition ID, \( r \) is the MPI rank of the process, \( n \) is the total number of processes, and \( F \) is the partitioning fanout. Therefore, the \texttt{next} function checks if the partition ID is smaller than the partitioning fanout and if this is the case, it then returns to the next operator an STL tuple containing the memory address of the partition, the number of elements that the partition contains, and the partition bits that are equal to the partition ID and are necessary for the materialization as described in Section 3.2.4. We should emphasize at this point that this tuple is not a CVM one, and that deprives us of the composability of this specific
operator, since it returns a type that the downstream operator exactly expects and is designed for.

We also observe that we do not need the task queue. One of the reasons that the authors introduce the queue is because they do not know the number of local partitioning and hash-probe tasks in advance. In our case the Volcano-style operators solve this problem, as an operator returns as many result tuples as it has and an empty result to signal the end of its output. Thus, we do not need to know the number of tasks beforehand. Finally, in the close function we release any allocated memory and we also close the upstream operator.

Figure 5.3: LocalPartitioning operator specifics
Do you have any tuples to return?

Are there more partitions?

Build a hash table on a partition

Probe the outer relation on the hash table

Push the matching tuples in a vector and return the first one

Return a tuple until the results vector is empty

Finish the execution

Close both upstream operators

Figure 5.4: HashProbe operator specifics
After the NetworkPartitioning operator, the LocalPartitioning operator consumes the corresponding STL tuples. When the control flow passes to the open function, every node gets each STL tuple assigned to it. Since the local partitioning is also histogram-based, we first compute the histogram of the input data and its prefix sum to determine where data resides inside an output partition. Afterwards, we partition the incoming tuples again based on the new partitioning fanout, such that the hash tables built for the corresponding data segments fit inside the cache. The next function returns to the downstream operator an STL tuple containing the memory address of a data partition, its size, and the network bits that the compression removed, that are necessary for the materialization. Again, this strips away the composability of this operator. Lastly, the close function, closes the upstream operator. We can see the above description as a diagram in Figure 5.3.

Finally, the execution plan finishes with the HashProbe operator, which produces a vector of results from one left and one right partition. These partitions correspond to the ones that the LocalPartitioning operator returns. In the open function the HashProbe operator opens both upstreams and then it executes the next function. There, we build a hash table from the left upstream and we probe the elements of the right upstream to this hash table. We push the matching tuples into a result vector and we return them one-by-one until the result vector is empty. The execution finishes when we have consumed all the partitions either from the left or the right upstream. The close function simply closes both upstreams. The structure that we describe for the HashProbe operator is shown in Figure 5.4.

Although the above hash probing logic does not have to keep the whole algorithm output twice, as we do in the previous integration, it still introduces some redundant memory allocation. More specifically, when we do not have any tuples in the result vector, we keep all the matching tuples from a partition and we do not return a tuple at a time when we find it. This is a weakness of our current integration, which we resolve on our subsequent split of the current operators.

Summarizing all the above, we notice that by splitting the “fat” MPIJoin operator into smaller ones, we do not have to just-in-time compile the join code as a separate library object. On the other hand, our operators are not flexible at all. More specifically the partitioning ones return STL tuples specifically designed to plug with the next operator in the plan of Figure 5.1 and not with an arbitrary operator. In addition to that, they also contain the scan of the data, for which we can utilize the RowScan operator. However, this integration gives us an intermediate baseline of the runtime of the original algorithm. Furthermore, it helps us identify potential difficulties in the integration of the original implementation into Volcano-style operators. We will use these observations for optimizing our final split.

5.3 Smaller composable operators

As discussed in the previous section, although the first division of the MPIJoin operator has some advantages, it does still not produce operators that are flexible enough such that we can reuse them in arbitrary plans. Also, the deduced operators could be more minimal (e.g., we could remove the scanning of the data from the partitioning operators). To this end, we do the following changes: First, we discard the data scanning from any partitioning operator. The operator responsible for this already exists in the CVM platform and is called RowScan. Its semantics are very simple. It gets an array (encapsulated in a CVM tuple), and in every next function call it returns an element of this array.
Figure 5.5: Inner plan depicting the final split of the MPIJoin to smaller operators
Second, the partitioning operators now return CVM tuples in their next calls and not STL tuples. That means that we cannot construct tuples of arbitrary length as we do in the previous split to pass along the network bits until the HashProbe operator. Instead of that, and following CVM’s principles, the partitioning operators should return tuples that consist of \( \langle \text{partition ID}, \text{partition data} \rangle \) pairs. Third, every operator should work in a way such that it consumes one tuple at a time whenever this is possible and it should avoid unnecessary points of materialization such as pushing results into a vector, like the HashProbe operator does in our previous approach. Following these principles, we present our final plan, which is also the plan that the ConcurrentExecutor runs in Figure 5.5. In the rest of the section, we explain how this inner plan works, as well as any operators that we introduce, in detail. We do not give details of the ParameterLookup and DataGenerator operators wherever we use them in this plan, as we explain their purpose in the previous chapters.

The execution of the inner plan starts with the NetworkPartitioning operator. The internals of this operator are the same as the ones in Figure 5.2. The only difference lies in what we return at every call of the next function. Instead of the STL tuple that we returned before, we now return a CVM tuple that has as members the partition ID (which is the same as the
network bits that we need for the materialization later) and the partition data, which is an array of longs, since the network partitioning also introduces the compression from pairs of longs to a long. Summarizing, the return types of this operator are tuples of \( \langle \text{long}, \text{Array} \langle \text{long} \rangle \rangle \).

The inner plan continues with the Zip operator. Its task is to take two tuples from its two upstreams and zip them into one tuple. We introduce this operator specifically for the integration of the join into the CVM platform. We present its semantics in Figure 5.6. As in every Volcano-style operator, the execution begins in the open function, where we open both upstreams. In the next function, we get one tuple from both upstreams, and we concatenate these two tuples into one. Following the return tuple types of the NetworkPartitioning, the Zip operator returns tuples of type \( \langle \text{long}, \text{Array} \langle \text{long} \rangle, \text{long}, \text{Array} \langle \text{long} \rangle \rangle \). In case the length of the data the two upstream operators return does not match, Zip throws a runtime error. This does not happen in our case, because the partitioning operators return matching partitions based on the same partitioning fanout. Finally, in the close function, the operator closes both upstreams. The reader should note the Zip operator works with arbitrary types and not just with longs. Since its task is simply to concatenate two tuples into one, it is type-agnostic.

At this point, every tuple that Zip returns is the input of a NestedMap operator. This operator is almost exactly the same as ParallelMap, which we present in Figure 3.5, with the exception that the execution of then nested plan does not lie inside #pragma omp instructions. We introduce this operator because in Volcano-style operator programming, when we want to run a series of functions for an input partition, it is necessary to put these functions in a nested plan. Therefore, the duty of this operator is to execute a sequence of functions without any parallelization.

In the beginning of the nested plan, we have to split the \( \langle \text{long}, \text{Array} \langle \text{long} \rangle, \text{long}, \text{Array} \langle \text{long} \rangle \rangle \) tuples into their individual components that the downstream operators need. We can do this using the Projection operator, which is already part of CVM. This operator takes a tuple that contains more than one field and keeps one or more of them, similar to the projection in relational algebra. As we observe in Figure 5.5, the output of the first Zip operator is an input to 4 different ParameterLookup operators. Depending on if we want the partition data for the subsequent local partitioning phase or the partition ID so that we can later materialize in the hash probe phase we should project on different positions of the Zip output tuple. Precisely, for the partition data we should project on the second or fourth position and for the partition ID we should project on the first or third position, depending on the upstream. After doing that, every downstream operator has its expected input.

The projected partition data of each of the two sides ends up in the LocalPartitioning operator. The operator specifics are almost the same as the ones in Figure 5.3. There are two main differences: First of all, the upstream RowScan operator returns its result one tuple at a time instead of one materialized array with known size as the previous NetworkPartitioning does. Therefore in the open function, we have to open the upstream operator, consume its upstream data, and close it twice, once for computing the histogram and the offsets and once for doing the actual partitioning. The second difference is that instead of returning an STL tuple, we return a CVM tuple with contents \( \langle \text{long}, \text{Array} \langle \text{long} \rangle \rangle \).

When we create the local partitions, we have to append to each of them the network bits that are the previous partition IDs and are necessary for the materialization. To do that, we use the CartesianProduct operator. This operator is the CVM equivalent of a mathematical cartesian product, namely if we have inputs \( A, B \) the operator will return pairs of \( (a, b) \) where \( a \in A \) and \( b \in B \). Note that in our case \( B \) only contains one tuple. Therefore the CartesianProduct operator returns CVM tuples consisting of \( \langle \text{long}, \text{Array} \langle \text{long} \rangle, \text{long} \rangle \) triples, where the first two fields are what the LocalPartitioning returns and the third field is the partition ID.
that the NetworkPartitioning gives to the NestedMap before.

At this point of the program execution, when we finish with the local partitioning, we are ready to execute the hash probe phase. The hash probe phase is inside another NestedMap operator, to execute it once for each pair of local partitions. Therefore as before we have to zip the output of the CartesianProduct operators and then project on the fields that the downstream operators need. To get back the network bits that we need for the decompression, we can take advantage of the following observation: Since we find matches on the elements that the hash table contains, we only need the partition IDs of the left upstream. After the projection, the RowScan takes the partition data and produces one \( \langle \text{long} \rangle \) tuple that is the input of the HashProbe operator for both the left and the right upstream.

The HashProbe operator internally works very differently from the previous section, as we put some effort to follow the principles of CVM and convert this operator to a generic join one. We present its specifics in Figure 5.7.

![Figure 5.7: HashProbe operator specifics](image)
The execution starts in the `open` function, where we open both upstreams and we consume all the left one, to built a hash table. In the `next` function, we consume all the right upstream and we probe the hash table, with the tuples that the upstream operator returns. Every time we find a matching tuple, we return it to the next operator. In case the right upstream does not have any more data to return or we cannot build any more hash tables from the left upstream, the `next` function returns an empty tuple, concluding the execution. In the `close` function, we close both upstreams. According to the pattern that we describe above, the `HashProbe` operator now avoids any materialization and instead forwards to the next operator one tuple at a time, following the CVM principles.

As we mention in Section 3.2.4 the hash table implementation that the join algorithm uses is specialized to work on partitions and assumes that the actual data does not lie in the hash table, but exist already in a different table. However, as `RowScan` returns a tuple at a time instead of a materialized array we have to allocate memory again, in order to store the actual workload.
of the algorithm, in addition to the memory devoted to the hash tables. This also implies that we should tune our local partitioning fanout in a way that the three allocated arrays (one for the data and two for the hash tables) fit inside the cache.

After the hash probe phase finds and returns the matching tuples, in order to undo the compression that the network partitioning does, we should follow the procedure described in Section 4.3. This decompression is simple enough to be done with a generic map function that takes some compile time parameters. To this end, we introduce a ParametrizedMap operator. We present its internals in Figure 5.8. Its execution starts in the open function where we open both upstreams and we get the necessary parameters that the map function needs. In our case to undo the compression, we need the network partitioning fanout, the number of payload bits, and the network bits. The first two are compile time parameters, whereas the latter is what the right upstream operator returns. In the next function, we apply the function that uncompresses the workload of the algorithm and we return the uncompressed tuple to the next operator. Finally, in the close function, we close both upstreams.

The operators following the ParametrizedMap operator are necessary for producing valid CVM plans. More specifically, every nested and inner plan must finish with a MaterializeRowVector operator. Therefore, although the first MaterializeRowVector is necessary for the materialization of the join, the second one is essential for the compilation of the outer nested plan. As we explain in Section 4.3 MaterializeRowVector introduces a level of nesting (because it returns an array), which requires a RowScan for the unnesting. Since we have two MaterializeRowVector operators in the inner plan that the ConcurrentExecutor runs, we need two RowScans. The final MaterializeRowVector is essential for the compilation of the inner plan of the ConcurrentExecutor.

This concludes the final split of the MPIJoin to as small and reusable operators as possible. Following the arguments of the previous section, now our operators follow the CVM encapsulation and we can reuse them in other scenarios, given that we pay attention to their semantics. In addition, we thin them even more and we use other CVM operators whenever this is feasible, instead of putting more than one functions inside one operator. Finally, we do not have any unnecessary materialization, but we work with partial results as CVM principles indicate.

5.4 Experiments

In this section we present the experimental results of our integration with the non-composable and the composable operators. The experimental setup is the same as the one of Section 4.4. As explained in the previous section, we have to allocate one extra table for the HashProbe operator. Because of this, we run our experiments with two local partitioning fanouts. The default one that the authors of the original paper use, which is equal to \(2^{10}\) and a fanout of \(2^7\), which is the one that produces the three largest possible tables that still fit in the cache. This is the configuration that produces the best performance for the version with the composable operators.

In Table 5.1 and Figure 5.9 we present the results with local partitioning fanout equal to \(2^{10}\). We can observe that both our integrations are faster than the original code for almost all the experiments and that the one with the non-composable operators is faster than the one with the composable operators. To investigate even more why this is happening we do a breakdown analysis for every phase for the case of 8 machines and we showcase the results in Table 5.2.
<table>
<thead>
<tr>
<th>machines</th>
<th>VLDB code w/window alloc</th>
<th>Non-composable operators [s]</th>
<th>Composable operators [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>13.11</td>
<td>12.58</td>
<td>13.16</td>
</tr>
<tr>
<td>3</td>
<td>9.41</td>
<td>8.99</td>
<td>9.25</td>
</tr>
<tr>
<td>4</td>
<td>7.18</td>
<td>6.73</td>
<td>7.09</td>
</tr>
<tr>
<td>5</td>
<td>5.98</td>
<td>5.56</td>
<td>5.81</td>
</tr>
<tr>
<td>6</td>
<td>5.23</td>
<td>4.77</td>
<td>4.91</td>
</tr>
<tr>
<td>7</td>
<td>4.70</td>
<td>4.26</td>
<td>4.34</td>
</tr>
<tr>
<td>8</td>
<td>4.07</td>
<td>3.65</td>
<td>3.81</td>
</tr>
</tbody>
</table>

Table 5.1: Execution time of the experiments with local fanout 2\textsuperscript{10}

![Figure 5.9: Execution time of the final experiments (with local fanout 2\textsuperscript{10})](image)
As we can see, the histogram/assignment/offset computation is faster in our integrations compared to the initial code, although the logic behind the code is the same. The same happens with the window allocation and network partitioning. We attribute this speedup to a heavy refactoring that we did to the original code, which contained a lot of class and/or function encapsulation, which is known [14] to make code execution slower. The local partitioning phase is the only phase that is slower in our integrations and we do an even more detailed breakdown only for this phase below. The hash-probe phase is faster in our version with the non-composable operators because we also have some code refactoring and slower in our version with the composable operators because, as explained before, we allocate memory to store the partitioned data while the original implementation gets the original partitions without reallocating memory.

We present the detailed breakdown of the local partitioning for 8 machines in Table 5.3. For the prefix computation and memory allocation, the recorded times are so small that we cannot make any conclusions, since there is no reliable method to record so small time intervals in a code block. Between the version with the non-composable operators and the original code, the difference in the histogram computation and the partitioning is very small. Since the code in these two programs is very similar, the only assumption that we can make is that the compiler in the first case can apply some additional optimizations that are not possible in the second case, as the code around the local partitioning is very different. Between the version with the non-composable operators and the version with the non-composable operators, we can observe a big difference in the histogram computation and partitioning sub-phases. The difference here is that in the final split the operators do not directly read the data from a memory address, but get a tuple at a time from the RowScan operator.
To investigate this further, we run the following benchmark: We generate 1 billion integers and we record the time that RowScan needs to read them and compute their sum, compared to a simple C++ program which does the same. RowScan needs about 1 second, whereas the C++ program needs around 0.8 seconds. Thus, we can attribute the difference in these sub-phases in the RowScan operator. This also leads us to the conclusion that sometimes a smaller operator granularity and the usage of an abstraction like RowScan can cause a slowdown in performance.

To demonstrate the difference that the local partitioning fanout makes in the running time, we rerun the above experiments but with a local partitioning fanout of $2^7$. We show our results in Table 5.4 and Figure 5.10. We can observe that our integrations are faster than the original code and the version with the non-composable operators is also closer in time to the version with the composable operators, compared to when we run the code with local partitioning fanout equal to $2^{10}$.

<table>
<thead>
<tr>
<th>machines</th>
<th>VLDB code</th>
<th>Non-composable operators [s]</th>
<th>Composable operators [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>w/window alloc w/materialization</td>
<td>14.29</td>
<td>12.66</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>10.18</td>
<td>9.05</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>7.78</td>
<td>6.71</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>6.44</td>
<td>5.59</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>5.61</td>
<td>4.77</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>5.04</td>
<td>4.28</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>4.40</td>
<td>3.65</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4: Execution time of the experiments with local fanout $2^7$

Figure 5.10: Execution time of the final experiments (with local fanout $2^7$)
To investigate the differences more, we do again a breakdown analysis by phase, which we present in Table 5.5. Our conclusions for the first three phases of the algorithm stay the same as before, with the difference that the local partitioning phase now takes less time for all the code versions. This is normal because with a smaller fanout we produce bigger and fewer partitions.

What is surprising is that now the hash-probe phase for the final split is faster than all the other times. However we can explain this by digging into the internals of an STL vector used in the original code and the version with the non-composable operators. When the memory allocated for the vector does not suffice to store the elements which we append to it, C++ uses the allocator interface to increase its memory. This is much slower than \texttt{realloc}, which the \texttt{MaterializeRowVector} operator uses.

<table>
<thead>
<tr>
<th>phase</th>
<th>VLDB code w/window alloc w/materialization</th>
<th>Non-composable operators [s]</th>
<th>Composable operators [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Histogram/assignment/offset computation</td>
<td>0.17</td>
<td>0.16</td>
<td>0.16</td>
</tr>
<tr>
<td>Window allocation and network partitioning</td>
<td>2.94</td>
<td>2.58</td>
<td>2.53</td>
</tr>
<tr>
<td>Local partitioning</td>
<td>0.43</td>
<td>0.44</td>
<td>0.66</td>
</tr>
<tr>
<td>Hash-probe</td>
<td>0.82</td>
<td>0.52</td>
<td>0.45</td>
</tr>
</tbody>
</table>

Table 5.5: Breakdown analysis per phase for local fanout $2^7$

Summarizing our experiments, we observe that by splitting the \texttt{MPIJoin} operator into smaller ones, we improve the original performance of the algorithm. We can attribute this either to low-level tuning (in the case for \texttt{realloc} for the materialization) or to code refactoring that removes the majority of function calls. The only phase that remains faster in the original implementation is the local partitioning. One interesting observation that we make is that thinning the operators sometimes costs in performance, as it is the case for the scanning of the data that we assign to the \texttt{RowScan} operator in our final split. Having produced the split that we think is optimal for reusing the operators and that also preserves the original performance, we can introduce the additional plans which use our operators in the next chapter. In that way, we can show the composability of our proposed operators.
Chapter 6

Reusing join building blocks

6.1 Introduction

In the previous chapter, we split the “fat” join operator into smaller ones such that they are recomposable and reusable. To facilitate that, except for the operators that already exist in CVM, we introduce some additional ones such as the Zip operator, the ParametrizedMap operator, the NetworkPartitioning operator, the LocalPartitioning operator, and the HashProbe operator. In this chapter we turn our attention to the reusability and recomposability of these operators and we showcase a number of programs where we can utilize them. We focus on data analytics algorithms as this is the target audience that CVM is addressed to. For each program we show the corresponding inner plan that CVM builds when this is not too tedious, and in other cases we provide some pseudocode or a high level description. We give examples that can represent a large class of algorithms to demonstrate the generality of the operators. At the end of the chapter we demonstrate a type of join called Grid Join where we can reuse almost all of our operators. We should note that we do not give any explicit examples of the ParametrizedMap operator, since this is a special case of a map function. A map function is one of the most basic and most frequently used ones in programming and we can rewrite every transformation of a data collection such that it incorporates it.

6.2 Zip operator

A loss or cost function in ML is a function that calculates the difference of the predictions of a model with the real values that the output of the model should have. One of the most widely used loss functions is the quadratic difference loss function. We can express this loss function as \( \lambda(x) = C(t - x)^2 \) where \( C \) is a constant, \( t \) is the target, and \( x \) is the value that the model predicts. If we write this mathematical function in pseudocode we get Algorithm 1. We can also see the corresponding CVM plan in Figure 6.1.

**Algorithm 1: Quadratic loss function**

<table>
<thead>
<tr>
<th>Data:</th>
<th>predictions, true values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result:</td>
<td>quadratic loss function value</td>
</tr>
</tbody>
</table>

1. Combine vectors elementwise
2. Compute square of difference for each pair of elements
3. Sum up the squared differences

---

53
As we observe, in order to calculate the quadratic loss, we have to zip the predictions with the real values. In the CVM plan, we read these collections with the RowScan operators. We then calculate the quadratic difference between each pair of values using a map function and the sum of these values using a reduce function. We note here that we can compute almost any loss function used in ML with the plan that we show before. The only adjustment that we should make is change the function that map takes as a parameter to something different than the quadratic difference loss function.

Another very common operation used in ML algorithms is to get a table of features and a table of labels and unify them into one table. This table subsequently is split into chunks to train the model using cross-validation.\footnote{https://en.wikipedia.org/wiki/Cross-validation_(statistics)} We show a CVM plan that implements this functionality in Figure 6.2.
6.3 Hash-Probe operator

There is a large class of models in ML that are called generative. Their goal is to approach the probability distribution of their input data. Some examples of these models are Generative Adversarial Networks [34] and Variational Autoencoders [37]. To validate their output, we calculate the difference between two probability distributions (whenever this is feasible). One way to do so is the Kullback–Leibler (KL) divergence. The KL divergence between two probability distributions $P, Q$ is defined as

$$D_{KL}(P \parallel Q) = - \sum_{x \in \mathcal{X}} P(x) \log \left( \frac{Q(x)}{P(x)} \right)$$

To compute this distance in the common domain $\mathcal{X}$ of the two probability distributions we first have to find the intersection of the two domains of each distribution. Otherwise, we may have zero quantities in the dominator of the log function.

In the case of discrete domains, we can calculate their intersection using our HashProbe operator. We show how to do so in Algorithm 2 and the corresponding CVM plan in Figure 6.3.

**Algorithm 2**: Calculating the intersection of two sets

<table>
<thead>
<tr>
<th>Data:</th>
<th>set1, set2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result:</td>
<td>intersection (set1, set2)</td>
</tr>
<tr>
<td>1</td>
<td>Build a hash table from set1;</td>
</tr>
<tr>
<td>2</td>
<td>Probe set2 in the previous hash table;</td>
</tr>
</tbody>
</table>

[https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence](https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence)
The execution starts by reading the two collections that represent the two domains and building a hash table from one of them. Then we probe the other one to find the matching elements, which constitute the intersection of the two sets.

Another example where we can use the HashProbe operator is matrix multiplication. Matrix multiplication is central to the majority of linear algebra algorithms. One way to express matrix multiplication is the concatenation of matrix-vector operations and to calculate those we can use our HashProbe operator if we get our data in a specific format. We show how to do so in Algorithm 3 and the corresponding CVM plan in Figure 6.4. Our program gets as a first input a matrix in the format of triples $(i,j,v_A)$, where $i$ is the row, $j$ is the column, and $v_A$ is the matrix value at the position $(i,j)$. It also receives a vector as a second input in the format of tuples $(j,v_B)$ where $v_B$ is the value of the vector at position $j$. It then maps the matrix row together with the value to produce tuples of the form $(j,(i,v_A))$. Then the HashProbe operator matches the keys that contain the values of the corresponding $j$ columns. Then a Map operator multiplies these values and a ReduceByKey operator sums up the values per row $i$ to calculate the final result. To calculate a matrix-to-matrix multiplication, we have to create a loop for the whole the procedure for all the rows $i$ of the first matrix and all the columns of the second matrix.

**Algorithm 3:** Matrix Vector Multiplication

Data: matrix as triples $(i,j,v_A)$, vector as tuples $(j,v_B)$

Result: matrix vector multiplication product

1. Project away the matrix values to remove the row $i$
2. Build a hash table from the column values $j$
3. Probe the vector $(j,v_B)$ to find matching tuples
4. Multiply the matching tuples using a map function $(j,v_A \cdot v_B)$
5. Reduce by key the resulting vector using a sum
6.4 Partitioning operators

In modern neural network training, it is usual to train models using mini-batches of data points, because researchers have proved that this improves generalization [44]. We can use our LocalPartitioning operator for this application. We show the corresponding plan in Figure 6.5, where the RowScan operator gives the original data to the LocalPartitioning operator. The partitioning operator creates the mini-batches (or partitions) using the row number as the key. Then a Projection operator removes these keys and returns partitions of the input data.
Another category of programs where we can use the \texttt{LocalPartitioning} operator is everywhere that there is a \texttt{HashProbe} operator. Similar to the join algorithm, partitioning the input data to small chunks can significantly benefit performance as the number of cache and TLB misses is heavily reduced.

A broad class of algorithms related to data analytics today is involved with large-scale graphs. These graph algorithms are solved using iterative methods that involve sparse matrix-vector and sparse matrix-matrix multiplications. Famous examples that fall into this category are PageRank \cite{48} and HITS \cite{38}. There are numerous works \cite{24,35} that accelerate these methods by using parallel and distributed sparse matrix-vector multiplications, which include data transfer between nodes for the respective computations. With minor modifications such as the inclusion of a different hash function, we can use our \texttt{NetworkPartitioning} algorithm for these kind of transfers. Building on the previous example of the matrix-vector multiplication, we just have to add on top a network partitioning phase, such that each node has the right data. Precisely, for input data triples \((i, j, v_A)\) for the matrix and \((j, v_B)\) for the vector, if we decide to partition the matrix by rows then we have to broadcast them to every node. For completeness reasons, we present the respective plan in Figure 6.6. For brevity, we do not repeat the matrix-vector multiplication, that we first present in Algorithm 3 and in Figure 6.4.
6.5 Grid Join

In this section we present a special type of join called Grid Join \cite{10} that uses almost all of our introduced operators. Its main motivation is to reduce the network communication between nodes in a sequence of joins in a distributed environment. It is especially beneficial for cycle joins e.g., of type $R_{a,b} \bowtie S_{b,c} \bowtie T_{c,a}$

We demonstrate the network communication reduction with a small example. Let us assume that we have three relations $R, S, T$ that we want to join that have sizes $r, s, t$ respectively. We can do the three-way join as a cascade of two-way joins, meaning that first we join $R, S$ and then the output of this join with $T$. For the first join, we shuffle amongs nodes and process $r + s$ amount of data and the output of the join is equal to $r \cdot s \cdot p$, where $p$ is the probability of a matching tuple in relations $R, S$. For the second join, we shuffle and process $r \cdot s \cdot p + t$ amount of data. On the other hand, the Grid Join does the three-way join with a shuffling phase that transmits $r + s + t$ records and a computing phase that processes $(r + s + t) \cdot \sqrt{k}$ data, where $k$ is the number of processes that are involved in the join computation. Putting numbers into the above equation, for relations that consist of 16 byte $\langle$ key, row ID $\rangle$ tuples have a size of 2 GB each, probability of a matching tuple equal to $2^{-10}$ and 64 processes, the Grid Join shuffles and processes 54 GB of data, whereas a cascade of two-way joins shuffles and processes 32 TB of data.

To showcase exactly how the join works, we also assume that the number of processes is the square of an integer, meaning that $k = m^2$. Then for a three-way join of a relation $R$ with attributes $(a, b)$, a relation $S$ with attributes $(b, c)$ and a relation $T$ with attributes $(c, d)$ the algorithm works as follows: First, it uses two different hash functions $h_1, h_2$ with range $[1 \ldots m]$ and it sends every tuple of a relation to the process indicated by these hash functions. To be precise, for the relation $R$ it sends its tuples to the nodes responsible to process tuples $(h_1(b), a)$,
for the relation $T$ to the nodes responsible to process tuples $(h_2(c), d)$ and for the relation $S$ to the nodes responsible to process tuples $(h_1(b), h_2(c))$. We show the above shuffling method for the three relations in Figure 6.7.

![Diagram of three relations and a network processing network](image)

**Figure 6.7: Grid join for three relations**

This network shuffling method splits the processes like a grid and that is where the name of the algorithm comes from. We demonstrate this grid for $k = 16$ processes for the above relations in Figure 6.8. The algorithm splits the 16 processes into a $4 \times 4$ grid, where the $x$-axis covers the hash function $h_2$ and the $y$-axis the function $h_1$. Then it transfers each tuple of a relation only to the process that the corresponding hash function shows. For example, only tuples that satisfy $h_1(S.b) = 3$ and $h_2(S.c) = 2$ go to the process that occupies the cell (2, 3) in the grid.

This example is a simplification of the original algorithm, shown just to demonstrate the motivation behind it. For more details, we advise the reader to look at the original paper [10]. Although the authors introduce this algorithm for the case of a two-dimensional grid for a three-way join relation, Beame et. al [19, 20] refine this to work into arbitrary dimensions and they call it the HyperCube join. Chu et. al [26] and Wu et. al [57] use this algorithm in real world systems and they demonstrate its advantage. This type of join as a cascade of joins reuses all of our introduced operators, if we extend them to work on arbitrary number of inputs (e.g. the Zip operators should take an arbitrary number of inputs and the network partitioning should incorporate the grid logic). We present the deduced plan for a three-way join in Figure 6.9. The execution plan is similar to the one that we presented in the previous chapter. We start by reading the three relations and then partition them according to the hash functions through the network. Afterwards we get our corresponding partitions and we partition them again. For the hash probing we build a hash table from the left-most upstream and we probe the other two upstreams to it. This plan, apart from the Grid Join algorithm, can also run a sequence of three joins that shuffles all the relations in the beginning instead of shuffling the output of the join of the first two relations through the network again.
Figure 6.8: Grid constructed by the join for $k = 16$ processes
Figure 6.9: Grid join as a CVM plan
In this thesis, we study the level of granularity that building blocks of data analytics systems have. We take as a case study a distributed hash join and we incorporate it into the CVM platform. We start our integration using a black-box approach and enclose all the algorithm logic inside a single operator. Subsequently we split the “fat” operator into smaller ones. We first construct operators that do not respect the CVM principles in their returned data types to have an intermediate baseline for the running time of our algorithm. We then refine the above operators to fully accommodate them in CVM. We showcase a number of programs where we can use our proposed operators, especially in the case of ML and data analysis systems. We also preserve the performance of the original join algorithm and even improve it in some cases.

A major conclusion that we gain from this thesis is that there is not a correct level of granularity. Although we propose an approach in [Chapter 5], one could also split subsequently the partitioning operators and calculate the histograms and the prefixes into other operators. In fact, calculating a local histogram is very similar to a reduce-by-key operator. Another insight that we observe is that although sometimes an abstraction like RowScan causes a bit of a slowdown, in general, granularity does not hurt performance. We therefore show that, by producing smaller building blocks, we can take the best of the world of high-performance computing with the advantage of reusability and recomposability.

As future work, we can implement some of the plans that we mention in [Chapter 6] to validate the assumption of highly reconfigurable high-performance computing.
Bibliography


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):

PRODUCING BUILDING BLOCKS FOR DATA ANALYTICS

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):

KOUTSOUKOS

First name(s):

DIMITRIO

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

Zurich, 16/08/2018

Signature(s)

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