MONASH University

A Relational Model for Parallel Computation

by

Harald Bögeholz, Dipl.-Math.

A thesis submitted for the degree of Doctor of Philosophy at

Monash University in 2022

Faculty of Information Technology

© Copyright

by

Harald Bögeholz

2022

A Relational Model for Parallel Computation

Harald Bögeholz, Dipl.-Math.
harald.boegeholz@monash.edu
harald@boegeholz.org
https://orcid.org/0000-0001-5091-1644
Monash University, 2022

Supervisor: Adj. A/Prof. Michael Brand
michael.brand@monash.edu
Associate Supervisor: Prof. Graham Farr
graham.farr@monash.edu

Abstract

In the era of Big Data, data scientists are faced with the challenge of scaling up their analytics algorithms to an ever-growing amount of data. This is commonly achieved by distributed computing, and over the last 15 years the ecosystem of distributed platforms for Big Data analytics has grown at a staggering pace. There is, however, no adequate theoretical foundation for reasoning about the efficiency of algorithms at a level of abstraction matching the needs of these platforms.

We propose the Relational Machine as a new model of parallel computation on Big Data. It offers a minimal set of operations for manipulating relations, making it conducive to mathematical reasoning. Based on this core model we also define a higher-level model called the Database Machine that operates on multirelations and has a slightly larger set of basic operations. This model represents a common subset of the functionality of relational databases and several other platforms for Big Data analytics.

Formulating algorithms in terms of the Database Machine allows them to be translated to any platform that supports a small set of basic relational operations. In this way, the Database Machine acts as a universal bridge between algorithms for Big Data analytics and the platforms to run them on. It allows a meaningful theoretical analysis of Big Data algorithms at a high level of abstraction.

A significant theoretical contribution of this thesis is establishing the connection between the Database Machine and the Parallel Random Access Machine (PRAM), a model of parallel computation widely studied in the 1980s. We prove that the two models can simulate each other and also show how PRAM algorithms from the literature can be translated for the Database Machine. From there, we demonstrate how to derive Big Data-practical implementations.

In memory of my father Heinz Bögeholz, who laid the foundation for all this
by giving me a TTL data book at age 11
and a computer at age 13

Contents

List of Algorithms

List of Tables

List of Figures

A Relational Model for Parallel Computation

Declaration

I declare that this thesis is an original work of my research and contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made.

Harald Bögeholz
May 29, 2022

Acknowledgements

Solving mathematical puzzles can disrupt your career – that's what happened to me after I discovered the website "Using your Head is Permitted" and tackled the difficult riddles it presented every month. Through this activity I got acquainted with the person behind it, my supervisor Michael Brand. It was Michael who encouraged me, after a career as an IT journalist in Germany, to embrace change and return to academia after leaving pure mathematics behind long ago. I am deeply grateful to him for believing in me and taking me on as a student – and most of all for his patience, which I believe I stretched to its utmost limit. Michael has been very caring and supportive throughout my candidature, always available to answer emails and gently nudging me back towards our vision whenever I lost sight of it.

My associate supervisor, Graham Farr, was an invaluable complement to Michael, offering different perspectives and inspiration whenever I got stuck. I would like to thank him not only for his academic advice, but especially for providing a sense of connection and emotional support way beyond our regular work meetings throughout the Covid-19 pandemic and the many lockdowns Victoria went through.

Mens sana in corpore sano – the mental effort of a PhD requires a healthy body, and this was heavily impacted by a severe traffic accident in the first year of my candidature. It was the orthopaedic surgeon Russell Miller who put the pieces of my smashed elbow back together. I am no medical expert, but from all I know about my injury and how well I recovered in the end, I can only conclude that Mr. Miller has worked a miracle. I heartily recommend him to anybody who needs a knee or shoulder repaired or replaced (and in an emergency he apparently also fixes elbows).

The PhD journey is an opportunity to make new friends, of which I can only name a few here. I appreciate the lunchtime walks and mathematical discussions with Srinibas Swain and Mahmoud Hossam, the founding of a little Go club with Thomas Hendrey and Nathan Companez, and the company of my soon-to-be academic sibling Ben Jones, back in the good old pre-pandemic times when we shared an office. I am glad to have EJ Watkins as a housemate and friend, making me truly feel at home in Australia and enduring my grumpiness during the final months of writing this thesis. I'd also like to mention an old friend and colleague I have known for more than 30 years: Andreas Stiller. For many years, Andreas has been my friendly competition in solving the monthly "Ponder This" riddles by IBM Research, and I value the advice and support he gave me when I was pondering whether to start this new journey and leave my job and him as a colleague behind.

It was an invaluable reinforcement of the foundations of my work to tutor Monash University's unit FIT2014 – Theory of Computation – and I'd like to acknowledge the three different teaching teams I worked with and became friends with.

I believe my application for a PhD scholarship was highly unusual, given I had spent 22 years in journalism and away from academia. All the more I thank Monash University and the Faculty of Information Technology for supporting me financially with a living allowance and tuition fee sponsorship, respectively.

Among the many difficulties I faced when writing this thesis, the English language was not the greatest one, and I owe this largely to my high school English teacher Franz Pfitzner who instilled in me a life-long love of learning languages.

Finally, I'd like to thank Donald E. Knuth for creating TEX and introducing me, through his series of books on computers and typesetting, to the art of creating beautiful documents. Whatever typographical beauty this thesis has is due to him and the many people who came after him and developed LATEX, Ti*k*Z etc.

Harald Bögeholz

Monash University
February 2022

Chapter 1

Introduction

The emergence of Big Data has had a profound impact on our daily lives and almost every scientific domain. According to market research by Valuates [78], the global Big Data and business analytics market size was valued at US-$ 198.08 billion in 2020, and is projected to reach US-$ 684.12 billion by 2030.

There are many attempts to characterise what Big Data is. Gartner's Information Technology Glossary [37] defines it using the "three V's": "high-volume, high-velocity and/or high-variety information assets [...]". Other authors have added two more V's: Veracity and Value [100]. Pospiech and Felden have modelled the concept from a System Sciences perspective [74].

The focus of this thesis is the aspect of volume. Traditional tools for data analysis such as R and Matlab can only handle datasets that fit into the main memory of a single machine. However, today's massive amounts of data have outgrown what can be processed by standard *up-scaling*, i.e. increasing the amount of memory, processing power, and storage space of a single machine. The solution to this is *out-scaling*: harnessing the power of multiple computers to perform computations in parallel. This usually goes hand in hand with distributed storage.

Processing data in a distributed computing environment generally requires refactoring or redesigning algorithms to fit the programming paradigm of the target platform. An early example of such an environment is the MapReduce framework [26]. It was developed in the early 2000s and is now part of the platform Apache Hadoop [1], together with the distributed file system HDFS. Hadoop is designed to be run on a cluster of commodity hardware and provides resilience to hardware failure by replication. A central idea is to bring the computation to the data instead of vice versa. Code to be executed is distributed within the cluster such that every machine can run it on its shard of data, followed by a shuffle and reduce phase to combine and redistribute the results. This was encapsulated in the MapReduce framework. In order to be run on MapReduce, a computing job has to be formulated entirely in terms of these map and reduce operations, creating new challenges in algorithm design for Big Data.

Over time, many new technologies for distributed processing evolved (see Section 1.3). Each new technology introduced a new programming interface and possibly a different language and a different data model. As a consequence, solutions cannot be easily ported

between different platforms and sometimes have to be redesigned from scratch to fit a new programming paradigm or data model. Furthermore, there is an imbalance between the effort put into Big Data infrastructure and application software development that has been estimated as high as 80:20 [59].

An additional challenge in developing algorithms for Big Data is the lack of adequate theoretical models to reason about the efficiency of such algorithms. As explained in more detail in Section 1.4 below, current Big Data platforms require algorithms to operate at a higher level of abstraction than that used by recent models of parallel and distributed computation. A detailed model of communication complexity, for example, requires an algorithm to be formulated in terms of communicating processes for which the communication patterns are exactly known. Such a model can simply not be applied in a scenario when an algorithm is based on an SQL-accessible cloud service such as Google BigQuery [32], which is opaque to the user and hides all details of the cluster and its communication patterns.

For addressing the practical challenges mentioned above, we propose a novel model of computation we call the Database Machine, introduced in Chapter 2. It operates on multirelations, i.e. tables that allow duplicate rows, and offers a small set of operations that can be executed efficiently in a distributed setting. These operations are the common ground already present in existing platforms for Big Data processing.

The Database Machine satisfies the requirements for a model of parallel computation as formulated by Skillicorn [88]: it offers

- a *methodology of software development* that is independent of a particular system architecture;
- an architecture-independent *way of measuring software's cost*;
- an *intellectually manageable model* of what happens when the software executes, relieving the developer from the cognitive burden of partitioning into threads or processes and explicitly describing communication and synchronisation.

Software written in high-level languages such as our multiset comprehension notation (Section 2.4) can be mechanically translated to the Database Machine's instruction set. Implementors of platforms for Big Data analytics can focus on implementing the core operations and optimising their performance to create infrastructure for the execution of such programs. In that way, the Database Machine acts as a bridge between algorithms and Big Data Platforms. An algorithm written for the Database Machine can be run on many existing Big Data platforms and reused on emerging platforms, provided they support the Database Machine's small set of core operations. This helps to protect the investment in designing new algorithms for Big Data as the landscape of Big Data platforms continues to change.

To further analyse the *theoretical* possibilities and limitations of computation on Big Data, we introduce in Chapter 3 a second new computational model, the Relational Machine. It has a more primitive instruction set that is clearly divided into instructions requiring communication and instructions performing computation. We show that the

Relational Machine can simulate the Database Machine with polylogarithmic slowdown, which we deem feasible for Big Data.

The most significant theoretical contribution of this thesis is the characterisation of the relationship between our relational models of computation and the Parallel Random Access Machine (PRAM). The PRAM is a powerful and universal model of parallel computation that was extensively studied (see Section 1.2.2). We show in Chapter 4 that a Database Machine can simulate a word PRAM with essentially the same asymptotic time and space complexity. A PRAM can in turn simulate a Relational Machine with a slowdown logarithmic in the space usage of the Relational Machine. (For an overview of computational models in this thesis see Figure 1.1 at the end of this chapter.)

The connection to the PRAM model is important not only for theoretical reasons. There is a large body of literature on parallel algorithms for the PRAM and this continues to be the standard way in which parallel algorithms are communicated in the literature, even if references to the PRAM model are in modern papers mostly implicit. Parallel algorithms are usually written in pseudocode in a SIMD style that uses only a subset of the full PRAM functionality. In Chapter 5 we demonstrate how such algorithms can be translated in a systematic and straightforward manner to the Database Machine. From there they can be readily implemented on a Big Data processing platform. We give a fully worked-out example of the Shiloach–Vishkin [85] algorithm for computing the connected components of a graph, turning an algorithm from the PRAM literature to a Big Data-practical implementation in Python and SQL that can be run in today's distributed relational databases.

1.1 Contributions

The contributions of this thesis can be summarised as follows.

- A framework for the development and analysis of algorithms on Big Data that is independent of specific platforms. Algorithms formulated in the high-level language developed in this thesis for the Database Machine can be systematically translated to current and future platforms for Big Data processing, thereby protecting the investment in their development. Furthermore, the model is simple enough to allow for rigorous mathematical analysis of algorithms.

- A model for relational computation with a minimal instruction set, the Relational Machine. This can be viewed as a RISC architecture for computations on Big Data. We show that it is as expressive as the Database Machine and hence spans the relational algebra. The Relational Machine may be the foundation for new approaches to parallel query optimisation in distributed relational databases, although this is not explored in this thesis.

- We show the connection between our models and the Parallel Random Access Machine (PRAM), a widely studied model of parallel computation. Our contribution allows the many years of algorithmic research that have been invested into the study

of PRAMs to be directly applied on every Big Data platform by a process of translation that is completely mechanical and requires no refactoring. This is rigorously proved in Chapter 4 of this thesis and demonstrated on a practical algorithm in Chapter 5.

The remainder of this Chapter presents related work. In Section 1.2 we first give an overview about the field of parallel computation, which is the theoretical foundation of our work. Section 1.3 presents the relevant platforms for Big Data analytics. The mismatch between the frameworks for formulating Big Data analytics algorithms and existing theoretical models of parallel computation motivates the need for a new computational model (Section 1.4). Core ideas are drawn from the relational model of database management (Section 1.5).

1.2 Models of parallel computation

Broadly speaking, the study of parallel computation started before parallel computers became widely available, much as Turing's work on computability [95] preceded the availability of computers. It is therefore the theoretical possibilities and limitations of parallel computation that were the subject of early research in this field.

One way to describe a parallel computation is to consider the problem as a Boolean function, which is then in turn computed by a combinational circuit constructed out of logic gates (see Section 1.2.1). A closer approximation of a universal parallel computer is the Parallel Random Access Machine (PRAM). This model was extensively studied in the early days of parallel computers (see Section 1.2.2), but then abandoned because it was realised that such machines would not be realisable in practice. However, it turns out that our novel relational models of computation are closely related in power to word PRAMs, building a bridge between the large body of PRAM research and today's platforms for Big Data analytics. The remainder of this section gives a brief account of these and other models of parallel and distributed computation.

1.2.1 Combinational circuits

Early research in parallel computation considered combinational circuits. A combinational circuit is a circuit made of logic gates without any feedback loops. Its outputs depend solely on the current state of its inputs and not on the history of previous inputs. Circuit complexity is measured in terms of *size* – the number of gates – and *depth* – the length of the longest path from an input to an output. If we assume that a gate takes one unit of time to perform its function, then depth corresponds to parallel time.

Unlike a Turing Machine which can take inputs of varying length and produce outputs of varying length, a combinational circuit has a fixed number of inputs and a fixed number of outputs. To compare the two models, one therefore has to consider families of circuits, one for every input length. Another difference between Turing Machines and circuit families is that a Turing Machine has a finite description and a circuit family is per se an infinite object. Borodin [14] has therefore introduced the notion of uniformity. A uniform circuit

family is described by a function that generates for any given input length the corresponding circuit.

A motivation for studying circuits is that they are mathematically very simple. This makes them a good candidate for establishing lower bounds on the inherent computational complexity of a problem. It was hoped in the 1980s that circuit complexity might help resolve the P vs. NP question [8, p. 305], but those hopes did not come true. For a survey on lower bound results on circuit complexity see [13].

The correspondence between circuit depth and parallel time puts the study of combinational circuits at the foundation of parallel complexity theory. It has been shown that sequential time is related to circuit size [84] and that sequential space is related to circuit depth [14].

The complexity class NC was first identified by Nicholas Pippenger [73] in the study of combinational circuits and is now commonly called NC for "Nick's Class" [23]. It can be defined in terms of a PRAM (see below) as the class of problems solvable in polylogarithmic time using a polynomial number of processors. Since a PRAM can be simulated by a Database Machine, this implies that problems in NC can be solved by a Database Machine in polylogarithmic time using polynomial space.

1.2.2 Parallel Random Access Machines

The Parallel Random Access Machine (PRAM) was introduced in the late 1970s as a natural generalisation of the Random Access Machine [5]. It is in fact not a single computational model but rather an umbrella term for a whole range of models where multiple RAMs operate in parallel. Usually they communicate via some form of shared memory, with a notable exception being the machines considered by Savitch and Stimson [82] which support parallel recursive calls and pass parameters only when a parallel subprocess is started or ended. The other PRAM models differ in memory layout, the capabilities of the individual RAMs, independent versus synchronous control flow, and more.

Fortune and Wyllie's PRAM model [35] has an unbounded number of processors, each with its own local memory and communicating via a shared global memory. All processors execute the same program, but each processor has its own instruction pointer and can follow an individual execution path.

Goldschlager's SIMDAG [42] has a single CPU controlling the flow of execution and an unbounded number of parallel processing units that are directed by the CPU and access a common shared memory. At each step, they either pause or perform the same operation, but on different data. SIMDAG stands for "single instruction stream, multiple data stream, global memory"; it is a SIMD computer according to Flynn's classification [34].

Instead of a single global shared memory, Parberry's version of a PRAM [71] has an unbounded number of processors, each of which has an unbounded number of registers. Any processor can access any other processor's registers. In our definition of a PRAM in Chapter 4, we adopt this two-dimensional memory layout because it aligns well with our constructions for simulating the Relational Machine.

PRAM models are classified according to whether they allow multiple processors concurrently accessing the same memory location. In the weakest type of PRAM, the EREW (Exclusive Read Exclusive Write) PRAM, such concurrent access is prohibited and the resulting behaviour undefined. The CREW PRAM allows concurrent reads, but still requires exclusive writes; Fortune and Wyllie's PRAM model [35] is an example of this. The strongest PRAM models allow concurrent reads and concurrent writes (CRCW) and are further differentiated by the resolution strategy in case of write conflicts. Several possibilities have been explored:

Weak: Concurrent writes are only allowed if all processors are writing the value zero.

Common-mode: All processors writing to the same location must write the same value.

Arbitrary-winner: If multiple processors write to a location, an arbitrary one succeeds, and it may be a different one if the same step is repeated.

Priority: In a write conflict, the processor with the lowest identifier wins.

Strong: Of multiple values concurrently being written to a location, the smallest wins.

The PRAM models above are listed in the order from weakest to strongest, i.e. an algorithm for one model will have the same complexity in any of the models subsequently described. Eppstein and Galil [29] give various simulations between PRAM models, the most important of which is simulating the strongest model by the weakest one. They prove that a parallel computation that can be performed in time t using p strong CRCW processors can also be performed in time $t \log p$ using p EREW processors. This implies that an algorithm for any type of PRAM model can also be executed on any other type of PRAM with at most a slowdown logarithmic in the number of processors.

1.2.3 The Bulk Synchronous Parallel model

The PRAM model plays an important role in exploring the theoretical limitations of parallel computation and was the subject of substantial research in the 1980s. It was, however, realised, that its assumption of unit-cost memory access cannot be attained by a physical machine. A consequence of this is that a provably optimal PRAM algorithm may not be optimal on any physical machine. Snyder [89] has demonstrated this using Valiant's algorithm [96] for finding the maximum of n elements with n processors.

The focus of research has therefore shifted to more realistic models of computation that take the cost of synchronisation and communication into account.

Valiant [97] has proposed the Bulk Synchronous Parallel (BSP) model as a bridging model between parallel algorithms and parallel computers. It is not a rigidly defined mathematical model but rather a framework to describe parallel computation that allows some variation. A formalisation of perhaps the simplest instance of the BSP model is the XPRAM [98].

A BSP computer consists of a number of processing components working in parallel, a router that delivers messages point-to-point between those components, and facilities

for synchronisation. Computation proceeds in supersteps during which each processor independently performs operations on data available in memory local to it. Before each superstep, each component sends and receives at most h messages.

A periodicity parameter L describes a regular check for synchronisation. After each period of L time units, the machine checks if all components have finished the current superstep. Only if they have, they move on to the next superstep. As opposed to a PRAM, where the processors execute each instruction in lockstep, a BSP computer uses this kind of barrier synchronisation where the individual processors execute asynchronously within each superstep.

1.2.4 Vertex-centric graph processing

An application of the BSP paradigm to large-scale distributed graph processing is Google's Pregel system [60]. It introduced a vertex-centric programming model for formulating graph algorithms based on the BSP pattern of supersteps. It operates on directed graphs. Within a superstep, each vertex executes some user-defined function receiving as input messages from incoming edges of the graph. The function can modify the vertex's state and then send messages across its outgoing edges, to be received in the next superstep. Vertex programs can also mutate the topology of the graph. Pregel was used by Google to implement their famous PageRank algorithm [16].

Since then, several vertex-centric frameworks for large-scale distributed graph processing have appeared; for a survey see [62].

Fan et al. [31] make the case that instead of introducing a specialised graph processing engine, an enterprise can leverage an existing relational database management system for performing graph analytics. They present a syntactic layer named Grail for formulating vertex-centric graph algorithms to be run in a database. Grail can be compiled to translate graph queries to SQL. Our simulation of a PRAM by a Database Machine and the associated methods for translating high-level PRAM algorithms (Chapters 4 and 5) can be viewed as an extension and generalisation of this idea.

1.2.5 Message passing systems

Most of today's systems for high-performance computing are clusters of computers communicating via high-speed interconnects. The Message Passing Interface MPI [3] is the most common programming model in writing parallel scientific applications [92, 28]. It allows more flexible communications patterns than the aggregate communication of the BSP model.

A large body of literature exists on modelling these message passing systems. The survey by Rico-Gallego et al. [79] lists 25 communication performance models appearing between 1992 and 2014. We summarise here the foundational steps in this development.

The first important improvement on previous models of communication was taking latency into account, i.e. the time it takes for a message from one processor to another to be delivered, switching from a "telephone model" where communication is assumed to be instantaneous to a "postal model" [11].

Two years later, Hockney [46] benchmarked the communication performance of parallel computers, adding the bandwidth to the picture. The time for transmitting a message is thus modelled as being determined by two parameters: the startup time and a transmission time proportional to the length of the message. This simple linear model has played an important role in the evaluation and optimisation of message-passing algorithms [79].

A more detailed model of communication in a parallel computer, taking into account the processing required for message transfer as well as communication delays, is the LogP model [25]. Its name is just the concatenation of its main four parameters. L is the latency, an upper bound on the delay incurred in sending a short message. The overhead, o, is defined as the length of time that a processor is busy sending or receiving a message. The gap, g, is the reciprocal of a processors's communication bandwidth. It signifies the minimum interval between consecutive message transfers. P, finally, is the number of available processors. To model network congestion, it is further assumed that at most $\lceil L/g \rceil$ messages can be in transit at any time; a processor attempting to send a message that would exceed this limit is stalled.

In comparison with the BSP model, the LogP model allows more flexible communication patterns. With the overhead parameter, the LogP model takes into account the time a processor is busy with sending and receiving messages, during which time it is not available for computation. A limitation of LogP is that it only considers short messages. This was remedied by Alexandrov et al. [6] with their LogGP model, adding another parameter G to capture the network bandwidth and thereby more accurately modelling the difference between shorter and longer messages. Over the following years, many refinements were made adding more and more details like network contention, heterogeneous networks, and synchronisation overhead; for surveys see [79, 30].

A notable milestone in accurately modelling complex systems for high-performance computation is the $\log_n$P model [17]. It is a software-parameterised model, taking into account the impact of middleware and the hierarchical nature of distributed communication. There is a growing gap between CPU performance, memory performance and network performance. A transfer of a message can be as simple as copying a memory buffer in an SMP system or as complex as going through multiple implicit transfers between source and target memories across a network. A "strided" message, i.e. a message stored in noncontiguous memory, will incur a higher cost than a contiguous message because MPI middleware will perform a series of implicit communications to complete the transfer: it will have to pack strided data at the source and unpack it at the target. The $\log_n$P model is parameterised by the number n of implicit transfers between the endpoints of communication.

The $\log_n$P model can accurately predict communication costs when the exact communication pattern of an algorithm is known and the parameters of a model have been measured. The authors of [17] demonstrate this experimentally by considering the special case $n = 3$, i.e. three levels of communication: first middleware, then network, then middleware again. They apply a $\log_3$P model to different algorithms for 3D Fast Fourier Transform, showing superiority over the LogP model which does not take the overhead of the middleware into account, in this case the open-source implementation of MPI, MPICH [65]. The authors

conclude, however, that "prediction is more cumbersome for the irregular patterns present in some codes that use sparse matrices".

This exemplifies a common theme in the development of models of communication complexity. As these models are taking more and more details of the underlying architecture into account, they become more accurate, but also more complex to apply. They require their parameters either to be experimentally determined or estimated, but, more importantly, they require detailed knowledge about communications patterns, narrowing their applicability.

1.3 Big Data processing platforms

The platforms designed to handle Big Data are different from systems for scientific high-performance computing in that their main focus is data management. Since Codd's seminal work on the relational model in the 1970s [20], data management became more or less synonymous with using relational database management systems. But in the last decade, another trend called NoSQL has emerged, which is most commonly interpreted as "Not only SQL". The change is that relational databases are just one option for data storage, others being key-value stores, document databases, column-family stores, graph databases, and the data lake, to name a few. Sadalage and Fowler [80] call this trend "polyglot persistence".

Our focus is on those technologies that deal with performing computations on large datasets of well-structured data, as opposed to managing storage and dealing with other aspects of Big Data like its variety. In this section we will go over the major technologies presently addressing this problem.

1.3.1 Relational databases

Relational databases have been around since the 1970s and are ubiquitous in today's business world. The Structured Query Language SQL is the universal language of these databases, adopted as an ANSI/ISO standard, but also extended by each database vendor in some proprietary way. The list of relational databases is too long to be given here. Broadly speaking, relational databases are used in two scenarios. One is online transaction processing where many users simultaneously update records. Another is data warehousing where existing data is rarely or never updated, but large amounts of data are continuously added and the challenge is to analyse these datasets.

A popular kind of database for this type of data warehousing is a Massively Parallel Processing (MPP) database with a shared-nothing architecture [91]. Teradata Corporation shipped the first production version of this kind of database in 1984 [101]. Other examples include Greenplum, Netezza, Vertica, and Apache HAWQ (see Section 1.3.3 below). Google BigQuery [32] is an SQL-accessible cloud service with similar capabilities.

One library of analytics algorithms that runs in this kind of environment is called MADlib and has been under development for a decade [44]. It is actively being developed as a top-level Apache project and was one inspiration for creating a relational model of computation that allows us to reason about algorithms formulated in this way.

1.3.2 Hadoop/MapReduce

One of the earliest frameworks for Big Data analytics on a distributed system was MapReduce, developed at Google [26]. It expresses a computation as a series of rounds where each round comprises a *map* phase, followed by a *reduce* phase. The map phase processes each row of a dataset in parallel and produces a set of key-value pairs. These are then shuffled to bring pairs with the same key together on the same machine. A reducer then reduces all values with the same key. MapReduce was originally implemented on Google's proprietary distributed file system GFS [39].

MapReduce later became part of the Apache Hadoop open source framework for distributed computing [1] which also includes an open source implementation of GFS named HDFS (Hadoop Distributed File System) [86]. Over time, Hadoop has grown into a large ecosystem with many related projects and is still actively being developed today.

1.3.3 HAWQ

Apache HAWQ [19] (which stands for Hadoop With Queries) is an open source MPP relational database that runs on top of Hadoop's distributed file system (see below) and utilises a cluster of machines for distributed query processing. From the underlying file system, it inherits the restriction that tables can only be newly created or appended to, but existing records cannot be modified. It turns out that despite this restriction, a database like HAWQ is a good execution environment for Big Data analytics algorithms, as we have shown in our own research on Big Data graph analytics (see Appendix B).

1.3.4 Pig

Another project built on top of Hadoop is Pig, a high-level data-flow language and execution framework for parallel computation [2]. The authors "have designed [it] to fit in a sweet spot between the declarative style of SQL, and the low-level, procedural style of map-reduce" [70]. Pig has a rich data model allowing nested structures comprised of atoms, tuples, bags and maps. Bags are collections of tuples with possible duplicates; in that sense they resemble the multisets we introduce in Chapter 2. They are, however, much more flexible in allowing tuples of mixed datatypes, nested tuples and even different types of tuples within the same bag. The language Pig Latin is a procedural language that offers a range of relational operators (FOREACH, FILTER, GROUP, COGROUP, JOIN, UNION, SPLIT) to manipulate such bags. Pig Latin compiles into MapReduce jobs that can then be executed on a Hadoop cluster.

1.3.5 Hive

The Facebook Data Infrastructure Team has built an open-source data warehousing solution on top of Hadoop named Hive [94]. It is another example of a system that is at its core relational. It uses an SQL-like declarative query language called HiveQL. HiveQL queries are compiled into MapReduce jobs that are then run on Hadoop. Users are able

to include custom MapReduce scripts into queries, thereby allowing them to mix the two programming models.

1.3.6 Spark

An important development is Spark, started at the University of California, Berkeley in 2009 [105]. It addresses the issue that MapReduce is not very well suited to applications that reuse a working set of data across multiple parallel operations, such as iterative machine learning algorithms. Instead of writing results to mass storage after each round, Spark introduces the concept of a Resilient Distributed Dataset [104], an in-memory data structure distributed across a cluster of machines. Resilience against node failure is achieved by recording all operations performed on such datasets in a lineage graph. If a node fails, the missing data partition can be recomputed on a different node. Spark was refined over time to allow materialising intermediate results to simplify fault recovery. Spark became a top-level Apache project in February 2014 [51].

A Spark RDD is essentially a multiset of tuples. The basic operations on RDDs are, again, the typical relational operations map, filter, join, groupByKey, union etc. The next step in Spark development was the module Spark SQL [7], which quickly became the most actively developed component of Spark; it was the top active component in the major 3.x release in 2020 [4]. Spark SQL provides a DataFrame API that unifies access to Spark's internal distributed collections and external data sources. Furthermore, queries can be formulated in SQL, allowing more complex transformations to be expressed in a single statement. SQL queries can then be optimised using an extensible query optimiser called Catalyst.

1.4 The need for a new computational model

As outlined in Section 1.2, recent research on models of parallel computation has focussed on capturing the increasing complexity of systems for high-performance computation, and in particular on the cost of communication within such systems. Applying these models requires knowing precisely how a computation is distributed in a system and the exact communications patterns to be used. This makes these models a good fit for high-performance scientific computations where software is often hand-crafted or at least fine-tuned for a specific supercomputer.

The programming interfaces of today's Big Data platforms, however, as outlined in Section 1.3, operate at a higher level of abstraction and hide from the user the concrete number of machines involved in the computation and their communications patterns. They allow the user to focus on specifying *what* is being computed and leave the details of *how* it is computed to the underlying layers of software and hardware.

For this reason existing models of parallel computation are ill-suited for analysing such Big Data algorithms. What is needed is a computational model that operates at the same level of abstraction as current Big Data platforms. By selecting a small number of basic operations commonly found in Big Data platforms, such a model will make it possible to

analyse and compare different algorithms for Big Data on this higher level of abstraction, gaining insights that will be valid regardless of the underlying platform.

If algorithms for Big Data analytics are written to a high-level interface such as we are proposing with the Database Machine or the Relational Machine, this decouples their optimisation from that of the underlying systems. Developers of systems for Big Data processing implementing this kind of interface can focus on optimising the performance of the basic building blocks using computational models such as those presented in Section 1.2.

There is also potential for optimisation at an intermediate level. If an algorithm for the Database Machine is implemented on a distributed relational database management system in SQL, it will profit from any query optimisation the database has to offer and therefore from decades of research into SQL query optimisation.

1.5 Relational data management vs. computation

The relational model for database management is hardly new, with Codd's seminal paper [20] appearing in 1970 and Version 2 of the relational model appearing as a book in 1990 [21]. The contribution of this thesis is a new perspective on the well-established idea of manipulating relations, namely as way of modelling computation as opposed to a way of modelling information.

Although the title of this thesis is "A Relational Model for Parallel Computation", it has very little in common with Codd's relational model for database management, other than the fact that the basic unit of information is a relation or, in the case of the Database Machine (Chapter 2), a multirelation.

Codd's theory of database management focuses on the semantics of the data being stored. A database is a representation of facts about the world and the relational model offers a systematic way of encoding such knowledge – and of manipulating it in a consistent way as the world changes. It is an extensive body of ideas; Version 2 of the relational model, published in 1990, already has a total of 333 features.

Beyond a large arsenal of operators for manipulating relations, it covers, among other aspects,

- integrity concepts within the relational model like entity integrity and referential integrity and how to maintain them;
- domains as a concept for representing the allowed range of values of a column;
- commands for the database administrator to manipulate properties of the database as a whole;
- a methodology to represent missing information, including a four-valued logic;
- the enforcement of semantic integrity constraints by the database, constraints on the stored relations following from the information they represent (i.e. rules pertaining to the business);

- dealing with multiple users and granting different access rights to different users or groups.

The sheer number of relational operators alone (37 "basic" and 44 "advanced") makes Codd's model a feature-rich tool for data management and manipulation, but unsuitable for mathematical reasoning. By contrast, our models are mathematical models of computation with the intention of providing the smallest possible number of operations to allow a rigorous study of computational complexity in terms of these operations.

The Relational Machine (Chapter 3) draws only core ideas from Codd's model. It works on relations that are split into a key and a value component, requiring the key component to be unique within the relation. This corresponds to the concept of a primary key in Codd's model. There are three relational operations (Section 3.2.1). The `JOIN` operation is what Codd calls an **equi-join**, the most simple version of joining that joins only on equality. Our `RANGE` operation is a restricted form of Codd's **project** operator. The third major operation of the Relational Machine, `SINGLES`, is novel as a primitive operation. It eliminates altogether all rows whose value components are not unique, as opposed to making rows unique by eliminating duplicates. The rest of the operations are data-parallel primitive building blocks for computation such as increment, decrement and bit shift, operating on all rows of a relation in parallel. They also have no direct counterpart in Codd's model. One could describe the Relational Machine as a kind of assembly language for database operations.

The Database Machine (Chapter 2) is a higher-level computational model closer to the actual Big Data processing systems and with a richer set of basic operations. It operates on multirelations instead of relations; these can be thought of as unordered tables allowing multiple copies of the same row. There are two reasons for allowing duplicate rows. For one, it is simply because SQL and most existing relational databases allow them. The other reason is that in a distributed Big Data scenario eliminating duplicates is a costly operation that should be assigned some cost in a model of computation.

In his Relational Model/Version 2, Codd dedicates a whole chapter to "serious flaws in SQL" with one of them being that SQL allows duplicate rows. He writes [21, p. 374]:

> "A fact is a fact, and in a computer its truth is adequately claimed by one assertion: the claim of its truth is not enhanced by repeated assertions. In database management, repetition of a fact merely adds complexity, and, in the case of duplicate rows within a relation, uncontrolled redundancy."

This illustrates once again the difference in perspective on relations. Codd's focus is the semantics of data as a model of representing aspects of the real world. By contrast, we look at (multi-)relations as a way to model parallel computation and put the computational complexity of operations at centre stage. Codd's model does offer a way to cleanly deal with duplicates in a relation, namely adding what he calls a DOD column (for degree of duplication). We use the same technique for simulating multirelations on the Relational Machine, where we call the extra column multiplicity.

Codd's model and its numerous manifestations in relational database systems based on SQL is declarative. An SQL query, possibly with nested subqueries, can specify a complex

expression on (multi-)relations yielding a (multi-)relation as a result. It does, however, not prescribe the exact order in which operations are performed in order to obtain the desired result. A relational database will convert an SQL query to an expression in terms of the relational algebra [87]. Such an expression can be transformed in various ways without changing the result. The query optimiser of a relational database management system will use such transformations in order to minimise time and resource usage.

By contrast, the computational models developed in this thesis are procedural. They allow the specification of operations on (multi-)relations without ambiguity about the exact order of operations. The operations of the Database Machine are small subset of the extended relational algebra, chosen to be equally expressive and with execution on a distributed platform for Big Data processing in mind.

1.6 Thesis structure

The remainder of this thesis is structured as follows. In Chapter 2 we define the Database Machine, a computational model closely resembling a Big Data-practical subset of the operations a distributed relational database provides. We introduce a high-level notation for writing algorithms for the Database Machine and present basic in-database techniques like generating sequences, sorting, and prefix computation.

Chapter 3 introduces the Relational Machine, a "database assembly language" with a more primitive instruction set operating on relations. The main theorem of this chapter, Theorem 3.5.11, states that a Relational Machine can simulate a Database Machine with polylogarithmic slowdown.

Chapter 4 establishes the connection between our novel computational models and the well-known Parallel Random Access Machine (PRAM). It turns out that a word PRAM, a PRAM with a logarithmic bound on the word size of its registers, can be simulated by a Database Machine (Theorem 4.1.1). In turn, a PRAM can simulate a Relational Machine (Theorem 4.4.6), completing the cycle as illustrated in Figure 1.1.

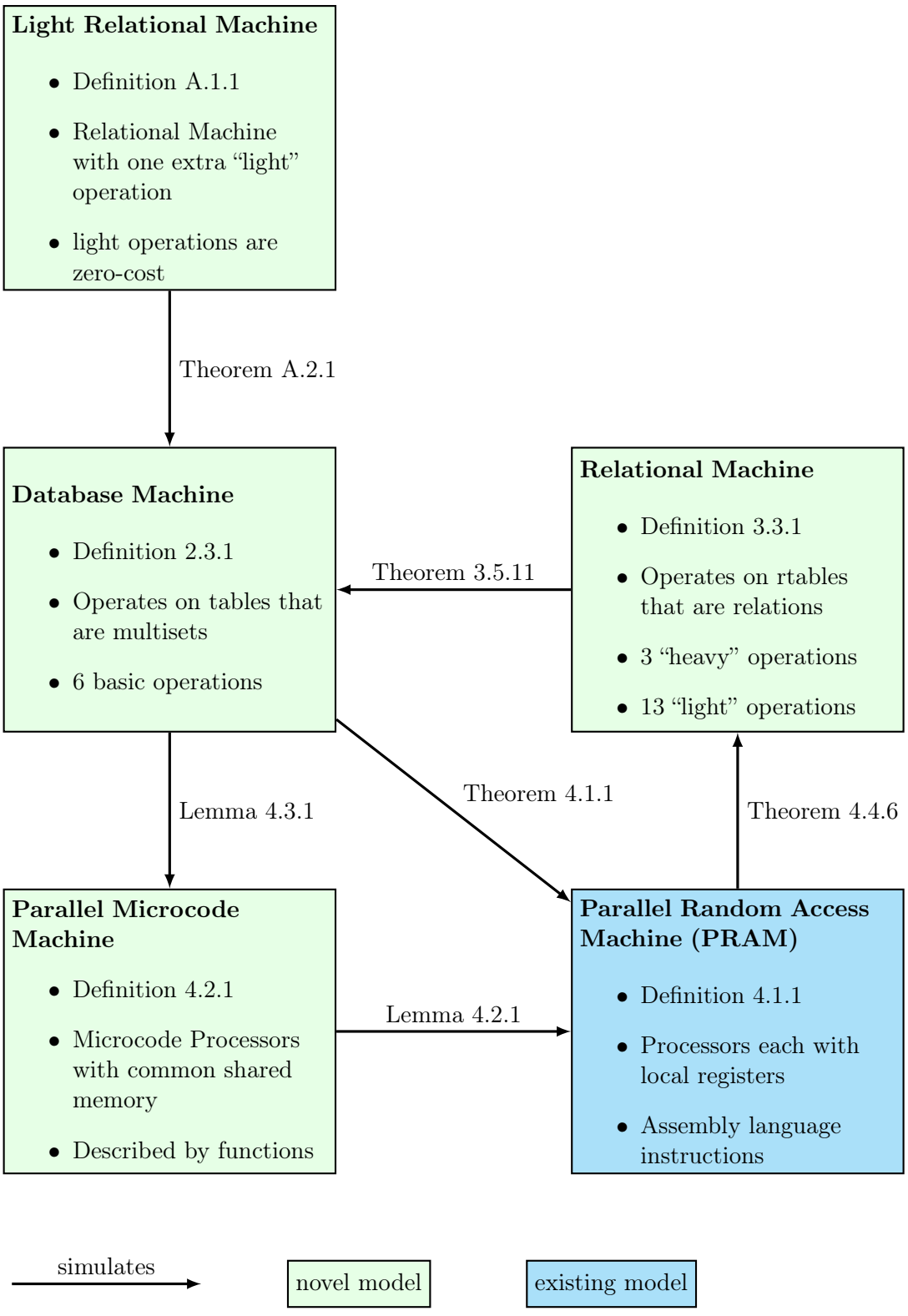
To pull the concepts from the entire thesis together, we demonstrate in Chapter 5 how PRAM algorithms written in some high-level language can be directly translated to our high-level Database Machine notation and from there to a practical implementation in SQL and Python. This leads to a Big Data-practical adaptation of a PRAM algorithm from the literature that computes the connected components of a graph.

Chapter 6 concludes the main part of this thesis and presents avenues for future work.

One of those avenues is explored in Appendix A: the Light Relational Machine. It is a variation of the Relational Machine that assigns zero cost to the "light" operations. This allows the simulation of the **map** operation of the Database Machine in constant time (Theorem A.2.1).

Appendix B is our published paper on a new in-database algorithm for computing the connected components of a graph.

Figure 1.1: Overview of computational models in this thesis and the relationships between them



Chapter 2

A model of in-database computation

To build a relational model of parallel computation rather than for data management, we have identified among the capabilities of SQL a set of operations that can effectively be applied to Big Data. The guiding principle was that it must be reasonable to assume unit cost for these operations when executed in a suitably-scaled distributed system. We have left out SQL features with large hidden complexity such as window functions. These are defined in terms of an ordering of a dataset and thus implicitly cause the whole dataset to be sorted before their execution. When needed, the functionality of these features can be built using our basic operations, yielding a more realistic computational cost.

We make the fundamental assumption that for an input table of n rows, the width of the individual rows is limited to $\mathrm{O}(\log n)$ bits throughout the computation. Without a limit on the row width, i.e. allowing arbitrary-size integers, our model would become unreasonably powerful [75, 15]. A fixed limit, on the other hand, would not scale with the input size. A model whose parameters depend in this way on the input size was named a trans-dichotomous model by Fredman and Willard [36] and affords realistic computational complexity bounds for integer computations, less conservative than Turing Machine bounds but more conservative than integer RAM bounds.

2.1 Tables are multisets

To introduce our new computational model, we need a few preliminary definitions.

Definition 2.1.1. Let D be a set. A function $M\colon D \to \mathbb{N}_0$ is called a *multiset* with ground set D. The set of all multisets with ground set D is denoted by

$$[\![D]\!] := \mathbb{N}_0^D.$$

For $d \in D$ the number $M(d)$ is called the *multiplicity* of d in M. Sets are special cases of multisets where the multiplicity of each element is at most 1.

Multiset membership is written analogously to set membership: For $M \in [\![D]\!]$ and $d \in D$ let

$$d \in M :\Leftrightarrow M(d) > 0.$$

The *cardinality* of a multiset $M \in [\![D]\!]$ is defined as

$$|M| := \sum_{d \in D} M(d).$$

We use the notation $M = \emptyset$ to denote that the multiset M has cardinality 0.

Definition 2.1.2. Let $\oplus\colon D \times D \to D$ be an associative and commutative binary operation on D and $M \in [\![D]\!]$. The result of *reducing* M using $\oplus$ is

$$\bigoplus_{d \in\!\in M} d := \bigoplus_{d \in D} (\underbrace{d \oplus d \oplus \cdots \oplus d}_{M(d) \text{ copies of } d}).$$

Similarly, we allow a predicate $p\colon D \to \{true, false\}$ to be used to denote reducing only those elements of the multiset for which the predicate is *true*:

$$\bigoplus_{\substack{d \in\!\in M \\ p(d)=true}} d := \bigoplus_{\substack{d \in D \\ p(d)=true}} (\underbrace{d \oplus d \oplus \cdots \oplus d}_{M(d) \text{ copies of } d}).$$

Note that this definition introduces the symbol $\in\!\!\in$. The notation $d \in\!\in M$ indicates that d runs through all elements of a multiset M, taking multiplicities into account, i.e. using each value of d as many times as specified by its multiplicity.

We are now ready to define the fundamental unit of computation of the Database Machine, the table. Informally, a table is a rectangular grid of nonnegative integers where no particular ordering of the rows is assumed. By defining it as a multiset, we allow it to contain multiple copies of the same row.

Definition 2.1.3. For a nonnegative integer $k \in \mathbb{N}_0$, a multiset of k-tuples, $T \in [\![\mathbb{N}_0^k]\!]$, is called a *table* with k columns, or a k-table for short. We assume throughout this thesis that tables have finite cardinality. An element $t \in T$ is called a *row*. For a fixed i with $1 \leq i \leq k$ the multiset of x_i for $(x_1, \ldots, x_k) \in\!\in T$ is called a *column*. The number

$$\sum_{i=1}^{k} \max_{(x_1,\ldots,x_k) \in T} \lceil \log(x_i + 1) \rceil$$

is called the *row width* of table T. It is the number of bits needed per row to store the whole table in a fixed-width format with the same number of bits for all elements of each column.

This definition of a table differs from the notion of a table in SQL:

- Columns in our model are unnamed which implies that the *order of elements within a tuple is significant*; implicitly, the columns are numbered. In SQL relations, columns are named and their order is trivially changeable. Semantically, a system with column numbers and a system with column names are equally expressive.

- All tuple elements in our model are nonnegative integers, whereas SQL allows different domains for different columns.

- The rows of a table in our model are *unordered* whereas SQL has the notion of ordering a table, e.g. by using an `ORDER BY` clause, and the notion of a row number within a partition given by the window function `ROW_NUMBER()`.

The rationale behind defining the rows within a table to be unordered is the following. In a relational database stored on a single machine, the rows of a table naturally have an order, namely the order in which they are stored on disk. In the scenario of a distributed relational database where a table is stored across multiple machines, the order of rows is already less obvious. Since the Database Machine is a theoretical model of parallel computation where each row of a table can conceptionally be handled by a separate processor, we choose not to impose an order on the rows.

In order to reason about operations on tables with bounded row width the following definition will be useful.

Definition 2.1.4. Let $x = (x_1, \ldots, x_k) \in \mathbb{N}_0^k$ be a k-tuple. We define the *width* of x as

$$\text{width}(x) := \sum_{i=1}^{k} \lceil \log(x_i + 1) \rceil.$$

Let $\oplus \colon \mathbb{N}_0^k \times \mathbb{N}_0^k \to \mathbb{N}_0^k$ be a binary operation. We say that $\oplus$ is *width-bounded* if for all $x, y \in \mathbb{N}_0^k$

$$\text{width}(x \oplus y) \leq \max\{\text{width}(x), \text{width}(y)\} + c$$

for a constant $c \in \mathbb{N}_0$.

An example for a width-bounded operation is addition: the result is at most one bit wider than the larger operand. Multiplication on the other hand is not width-bounded.

2.2 Basic operations on tables

In this section we define a set of operations for manipulating tables. Together they comprise the instruction set of the Database Machine, which we will formally define in Section 2.3. The Database Machine has a collection of registers, each storing a table, and manipulates these registers with the instructions defined here. Each operation takes one or more table registers as input, denoted by variables A and B, and stores its output in a table register.

Some of these operations use the notion of a *database function*. The idea of this term is to characterise a function that can reasonably be computed inside a database or database-like execution environment and be assigned unit cost. More precisely, it is a function that can be computed by a Turing Machine in polynomial time and linear space. We will postpone the rigorous definition (Definition 3.4.5) to Section 3.4.7, where we define it together with the exact flavour of Turing Machine to be used.

The following operations are supported by the Database Machine.

union. Form the disjoint union of two tables with the same number of columns. Let $A, B \in [\![\mathbb{N}_0^k]\!]$. We define the union as $A \uplus B := U \in [\![\mathbb{N}_0^k]\!]$ where for all $t \in \mathbb{N}_0^k$

$$U(t) := A(t) + B(t).$$

select. Form a subset of the rows of a table satisfying a given predicate. Let $A \in [\![\mathbb{N}_0^k]\!]$ and $p\colon \mathbb{N}_0^k \to \{0,1\}$ be a database function. The operation $\mathbf{select}_p\colon [\![\mathbb{N}_0^k]\!] \to [\![\mathbb{N}_0^k]\!]$ is defined as follows: let

$$\mathbf{select}_p(A) := A'$$

where for all $a \in \mathbb{N}_0^k$

$$A'(a) := \begin{cases} A(a) & p(a) = 1 \\ 0 & \text{otherwise.} \end{cases}$$

Note that under the **select** operation the cardinality of a table can only decrease, i.e. we have $|A'| \leq |A|$.

map. Apply a function to all rows of a table. Let $f\colon \mathbb{N}_0^k \to \mathbb{N}_0^l$ be a database function. The operation $\mathbf{map}_f\colon [\![\mathbb{N}_0^k]\!] \to [\![\mathbb{N}_0^l]\!]$ is defined as follows: for $A \in [\![\mathbb{N}_0^k]\!]$ let

$$\mathbf{map}_f(A) := A'$$

where for all $a' \in \mathbb{N}_0^l$

$$A'(a') := \sum_{\substack{a \in \mathbb{N}_0^k\colon \\ f(a) = a'}} A(a).$$

Note that f is not required to be injective. If multiple tuples map to the same result tuple, their multiplicities are added. This implies that A' has the same cardinality as A. It further implies that the **select** operation cannot be defined in terms of **map**, because **select** can decrease the cardinality of a table.

join. Join two tables on a subset of their columns. For nonnegative integers k, l, m with $m \leq k, l$ let $A \in [\![\mathbb{N}_0^k]\!]$ and $B \in [\![\mathbb{N}_0^l]\!]$ be two tables. The operation $\mathbf{join}_m\colon [\![\mathbb{N}_0^k]\!] \times [\![\mathbb{N}_0^l]\!] \to [\![\mathbb{N}_0^{k+l-m}]\!]$ is defined as follows: let

$$\mathbf{join}_m(A, B) := J$$

where for all $(a_1, \ldots, a_k, b_{m+1}, \ldots, b_l) \in \mathbb{N}_0^{k+l-m}$

$$J(a_1, \ldots, a_k, b_{m+1}, \ldots, b_l) := A(a_1, \ldots, a_k) \cdot B(a_1, \ldots, a_m, b_{m+1}, \ldots, b_l).$$

The result of joining A and B on the first m columns consists of all combinations of tuples from A and B that have identical values in the first m columns. The multiplicities of the result tuples are the product ("$\cdot$") of the respective multiplicities in A and B.

ljoin. Left outer join of two tables on a subset of their columns. For nonnegative integers k, l, m with $m \leq k, l$ let $A \in [\![\mathbb{N}_0^k]\!]$ and $B \in [\![\mathbb{N}_0^l]\!]$ be two tables. The operation $\mathbf{ljoin}_m\colon [\![\mathbb{N}_0^k]\!] \times [\![\mathbb{N}_0^l]\!] \to [\![\mathbb{N}_0^{k+l-m+1}]\!]$ is defined as follows: let $p\colon \mathbb{N}_0^l \to \mathbb{N}_0^m$ be the function defined by $p(b_1, \ldots, b_l) := (b_1, \ldots, b_m)$ for all $(b_1, \ldots, b_l) \in \mathbb{N}_0^l$, i.e. the

projection to the first m columns. Let $D = \mathbf{map}_p(B)$. We now define

$$\mathbf{ljoin}_m(A, B) := J$$

where for all $(a_1, \ldots, a_k, b_{m+1}, \ldots, b_l, \sigma) \in \mathbb{N}_0^{k+l-m+1}$

$$J(a_1, \ldots, a_k, b_{m+1}, \ldots, b_l, \sigma) := \begin{cases} A(a_1, \ldots, a_k) \cdot B(a_1, \ldots, a_m, b_{m+1}, \ldots, b_l) & \\ \qquad\qquad \text{if } \sigma = 1 \text{ and } (a_1, \ldots, a_m) \in D & \\ A(a_1, \ldots, a_k) \quad \text{if } \sigma = 0 \text{ and } (a_1, \ldots, a_m) \notin D & \\ \qquad\qquad \text{and } b_{m+1}, \ldots, b_l = 0 & \\ 0 \qquad\qquad \text{otherwise.} & \end{cases}$$

The result of **ljoin** contains all tuples in the result of **join**, augmented with an extra tuple element $\sigma = 1$. In addition, it contains all tuples from A not matched in B, padded with zeros and augmented with an extra tuple element $\sigma = 0$.

group. Group the rows of a table by a subset of columns and reduce to one row per group using a binary operation. Let $A \in [\![\mathbb{N}_0^k]\!]$ be a table and $m \leq k$. Let $\oplus\colon \mathbb{N}_0^{k-m} \times \mathbb{N}_0^{k-m} \to \mathbb{N}_0^{k-m}$ be a commutative and associative binary operation that is a width-bounded database function. The operation $\mathbf{group}_{m,\oplus}\colon [\![\mathbb{N}_0^k]\!] \to [\![\mathbb{N}_0^k]\!]$ is defined as follows: let

$$\mathbf{group}_{m,\oplus}(A) := G$$

where for all $(a_1, \ldots, a_k) \in \mathbb{N}_0^k$

$$G(a_1, \ldots, a_k) := \begin{cases} 1 \quad \text{if } (a_{m+1}, \ldots, a_k) = \bigoplus_{\substack{(x_1,\ldots,x_k)\in A:\\ (x_1,\ldots,x_m)=(a_1,\ldots,a_m)}} (x_{m+1}, \ldots, x_k) \\ 0 \quad \text{otherwise.} \end{cases}$$

Note that the result of the **group** operation is a set.

Control flow of the Database Machine can be changed in the following ways:

jump. Transfer control to a different location in the program.

conditional jump. Transfer control to a different location if a table register contains the empty table or if a table register contains a nonempty table.

halt. End the computation.

2.3 The Database Machine

We are now ready to define the Database Machine.

Definition 2.3.1. A *Database Machine* consists of a program to be executed and four integer bounds r, l, c_1 and c_2. The machine has registers $R_1, \ldots, R_r$, each storing a

table, and a read-only register **1** containing the single-row 0-table. The program is a finite sequence of instructions, indexed by positive integers. Each instruction is one of the operations defined in Section 2.2, taking as input one or more table registers and storing the output in a register. A computation on the machine is defined as follows. The input is placed into register R_1 and all other registers contain the empty table. The *input size*, n, is defined as the cardinality of the input table. Execution starts at instruction 1 and continues to the following instruction unless control is transferred to a different location by one of the jump instructions or the machine halts. When the machine halts, register R_1 is considered to be the output. Execution is constrained by the following rules:

1. The maximum tuple length in any register is l.
2. The row width of the input is not larger than $c_1 \cdot \log(n+2)$.
3. The row width of any register is not larger than $c_2 \cdot \log(n+2)$.

If any operation – including placing the input in register R_1 – would violate them, the machine *crashes* and the output is undefined. An input that does not immediately crash the machine when it is placed in register R_1 is called *valid*.

When describing algorithms for the Database Machine, we will use capital letters to stand for table registers. We assume that different letters stand for different registers and that the total number of registers used is within the register bound r of the machine.

Definition 2.3.2. Let $T, S \colon \mathbb{N}_0 \to \mathbb{N}_0$. A Database Machine is said to compute in *time* $T(n)$ if it halts after executing at most $T(n)$ instructions for any input of size n. It is said to compute in *space* $S(n)$ if for any input of size n the total number of rows in all registers at any point in the computation is at most $S(n)$.

Definition 2.3.3. We call a Database Machine *Big Data-practical* if it runs, for any input of size n, in polylogarithmic time and $\mathrm{O}(n)$ space. We call it *Small Data-practical* if it runs in polylogarithmic time and polynomial space.

Unless stated otherwise, whenever we speak of time in this thesis we are referring to the theoretical notion of time according to Definition 2.3.2 – or similar definitions for the other computational models defined in later chapters – as opposed to execution time on a real system. This definition assigns unit cost to the basic operations on tables defined in Section 2.2.

The rationale behind assigning unit cost to these operations is the following. We are envisioning a distributed system for Big Data processing where local computation is significantly faster than communication between nodes. We further assume that the system scales with the data, i.e. tables of size n will be handled by $\Theta(n)$ nodes, each of which holds a portion of the data of substantial size.

In this scenario, **select** and **map** are data-parallel operations that can be run independently on all nodes, involving no inter-node communication. The same holds for the **union** operation where each node conceptually makes copies of its respective portions of

the two input tables that stay within the same node. Since tables are immutable in our model, this can in practice be a pure pointer operation.

The **join** operation is defined to be able to join on equality only to allow for an efficient implementation using hashing. Though in practice more costly than **select** and **map**, it still is a one-step operation to bring together each pair of rows to be joined on a single machine. When we consider Big Data-practical algorithms according to Definition 2.3.3, we also know that the result size of a **join** operation is $\mathrm{O}(n)$ instead of the worst-case $\mathrm{O}(n^2)$ for inputs of size n.

Since **group** is defined in terms of a binary operation, it inherently takes $\log n$ steps of computation to reduce n elements. However, we are assuming local computation to be fast so that all rows on the same machine can be reduced very quickly. Little communication and computation is then required to gather and reduce intermediate results from different machines.

In the world of Big Data, it is common to analyse algorithms in terms of powerful high-level operations. For example, in the MapReduce framework, time is usually measured in terms of *rounds* comprising a map phase and a reduce phase [76]. In the same vein, *time* on the Database Machine is measured in terms of the number of the basic operations defined above.

The basic operations of the Database Machine are chosen to be a Big Data-practical subset of the operations of the relational algebra. In modern treatments [87], $\mathbf{select}_p$ is usually written as σ_p and $\mathbf{map}_f$ is an extended version of the project operator Π, allowing not only the projection to a subset of existing columns but the mapping of an arbitrary function f. Our $\mathbf{join}_m$ is a restricted version of the theta-join in the relational algebra, which is usually written as $r \bowtie_\theta s$ for two relations r and s. Here, θ signifies a predicate on $r \times s$ which is the most general form of join. The Database Machine allows joining on equality of the first m tuple elements only and also avoids duplicating these in the result. In the same way, we restrict our left outer join $\mathbf{ljoin}_m$ to joining on equality of the first m columns instead of allowing an arbitrary predicate θ.

The relational algebra uses the standard set operator $\cup$ to denote the union of relations. Since the Database Machine operates on multirelations, we use the symbol $\uplus$ to emphasise the fact that duplicates are not removed and multiplicities are added. Finally, our operation **group** corresponds to the aggregate operation, written as γ in the extended relational algebra. We have specialised it to always group by the first m elements of a tuple and generalised it to reduce the remaining tuple elements using a commutative and associative binary operation.

Since the basic operations chosen for the Database Machine are available in the major frameworks for Big Data processing, practical implementations of algorithms written for the Database Machine can be readily derived. This could make the high-level language introduced in the next section the *lingua franca* for Big Data processing. Because of the close relationship with the relational algebra, new implementations of this set of operations can leverage decades of research into database optimisation. The restrictions in comparison

with the full relational algebra, such as allowing joins only on equality, facilitate efficient execution in parallel architectures.

2.4 A high-level database language

In this section we introduce a high-level notation to express algorithms for the Database Machine. In a real-world implementation of an algorithm running in a database, there is a procedural language involved that sends queries to the database and can perform local computation using scalar variables. Instead of adding scalar variables as a feature to the Database Machine, we show how to emulate them as single-row, single-column tables, making them a mere notational convention. We also introduce a powerful high-level notation for specifying a sequence of operations to manipulate tables that can be directly translated to an SQL implementation.

2.4.1 Copying and literal tables

The Database Machine does not have a built-in operation to copy the contents of a table register to another table register, but this can readily be achieved by applying a $\mathbf{map}_{\mathrm{id}}$ operation where id is the identity function. If A and B are table registers, we denote copying the contents of B to A as

$$A \leftarrow B.$$

A literal table containing a single row with any desired tuple $(x_1, \ldots, x_l)$ can be created by applying $\mathbf{map}_f$ to the built-in constant single-row 0-table $\mathbf{1}$ where f is the empty function that maps the 0-tuple to the desired l-tuple. We write creation of such a literal table in a table register A as an assignment

$$A \leftarrow [(x_1, \ldots, x_l)].$$

Using $\uplus$ we can create any constant table in a constant number of operations. We write $[t_1, t_2, \ldots, t_n]$ for a multiset containing the tuples $t_1, \ldots, t_n$, each with multiplicity 1.

2.4.2 Scalar variables

A *scalar variable* is a single-row 1-table, i.e. a table containing a single integer. We will use lower case names to stand for scalar variables. A computation f involving scalar variables $s_1, \ldots, s_k$ and yielding a result r will be denoted as $r \leftarrow f(s_1, \ldots, s_k)$ and carried out by executing the following sequence of database operations: After copying s_1 to r, use a $\mathbf{join}_0$ operation for each of the remaining scalars $s_2, \ldots, s_k$ to add them as additional tuple elements to r. The result is a single-row table since joining two single-row tables on the empty tuple always yields a single-row table. We then apply $\mathbf{map}_f$ to compute the expression; the result is a scalar in r.

For example, the expression

$$c \leftarrow a + b$$

stands for the sequence $c \leftarrow a$; $c \leftarrow \mathbf{join}_0(c, b)$; $c \leftarrow \mathbf{map}_+(c)$.

2.4.3 Control flow

The only condition a Database Machine can test for to change control flow is the emptiness of a table. This is sufficient to test for any predicate that can be computed by a database function. Let $p(s_1, \ldots, s_k)$ be an expression in scalar variables yielding a result in $\{0, 1\}$. Then computing this expression as described above and applying $\mathbf{join}_1$ with the single-row table containing the integer 1 yields the empty table if the predicate returned zero and a single-row table otherwise. A conditional jump instruction can then be used to change control flow accordingly.

We will use the usual pseudo-code constructs **if**, **while**, **repeat**, and **for** to specify conditional execution and loops based on Boolean expressions involving scalar variables to indicate executing the above sequence of operations.

2.4.4 Manipulating tables

We now introduce a high-level notation to specify operations on tables. It is inspired by the list comprehension construct in the Haskell programming language [61] and translates directly to a sequence of the basic operations defined for tables in Section 2.2. It can also be translated to SQL in a straightforward manner (see Chapter 5).

In order to create a convenient notation for applying the **group** operation, we first introduce the notion of an aggregate function. An *aggregate function* reduces a nonempty table to a single tuple by using an associative and commutative binary operation. It is defined in terms of three other functions:

1. an *encoding function* $e\colon \mathbb{N}_0^k \to \mathbb{N}_0^l$
2. a binary operation $\oplus\colon \mathbb{N}_0^l \times \mathbb{N}_0^l \to \mathbb{N}_0^l$
3. a *final function* $f\colon \mathbb{N}_0^l \to \mathbb{N}_0^m$

The aggregate function $\mathbf{agg}_{e,\oplus,f}\colon [\![\mathbb{N}_0^k]\!] \setminus \emptyset \to \mathbb{N}_0^m$ is defined as

$$\mathbf{agg}_{e,\oplus,f}(T) = f\Big(\bigoplus_{t \in T} e(t)\Big).$$

Note that aggregate functions are not defined for the empty table since they cannot be applied to the empty table using our notation. Hence, the binary operation $\oplus$ does not need to have an identity element for the above expression to be well-defined.

In simple cases, the encoding and final functions are the identity. For example, using integer addition as the binary operation, we have $\mathrm{sum} := \mathbf{agg}_{\mathrm{id},+,\mathrm{id}}$ and write $\mathrm{sum}(x)$ to denote the sum of a multiset x of integers. A slightly more involved example is computing

the average. Let

$$e(x) := (x, 1)$$
$$(x_1, c_1) \oplus (x_2, c_2) := (x_1 + x_2, c_1 + c_2)$$
$$f(x, c) := \lfloor x/c \rfloor$$

Then with avg := $\mathbf{agg}_{e,\oplus,f}$ we can write avg(x) to compute the (rounded-down) average of a multiset of integers x.

A *multiset comprehension expression* describes how to compute a table from a number of existing tables and scalar variables. It has the form $[E \mid M]$ where E is an expression and M is a comma-separated list of terms. It builds a table by starting with a single-row table and successively applying the terms in M from left to right, building intermediate tables along the way. Finally, the expression E specifies a map from the last intermediate table to the final result.

Names are used throughout to refer to columns of the current table. A name is *bound* if it has been assigned a scalar variable in the context surrounding the multiset comprehension expression or if has been bound by a previous term. Otherwise it is called *free*. Note that the expression E, although written at the beginning of the multiset comprehension expression (resembling established mathematical notation for sets), conceptually comes last and can use all names bound by the terms in M. A column is *named* if a name is bound to it and *anonymous* otherwise.

The initial intermediate table contains a single row with the 0-tuple (). Each term in M takes the current table and transforms it into a new table, carrying along column names for future reference. If any term uses a name that is bound to a scalar variable in a surrounding context, that variable is first added as a named column to the current table by executing a $\mathbf{join}_0$ operation (recall that a scalar variable is a single-row 1-table). This effectively duplicates the value in each row of the table. The terms can be of the following types:

- $(x_1, \ldots, x_k) \Subset T$. Join the current table with the k-table T on the columns specified by the names that are already bound, binding all other names. This comprises the following steps:

  1. Apply a **map** to T to rearrange its columns such that the columns referenced by bound names among $x_1, \ldots, x_k$ are at the beginning of the tuple, followed by the free variables, in the order they occur in the list.
  2. Apply a **map** to the current table to rearrange its columns such that the columns referenced by bound names are at the beginning of the tuple in the same order they appear in T.
  3. Replace the current table C by $\mathbf{join}_m(C, T)$ where m is the number of bound names, binding the previously free names to the newly added columns.

  Any of the x_i can be the symbol $*$. Every occurrence of $*$ is considered a new free name, and instead of binding it to a column in the process described above, it will

cause the corresponding column to become anonymous. Anonymous columns cannot be referenced in subsequent terms.

- $(x_1, \ldots, x_k) \Subset_\sigma T$. Perform a left outer join of the current table with the k-table T on the columns specified by the names that are already bound, binding all other names and σ. This behaves like the term described above, but uses **ljoin** instead of **join**, binding the name σ to the additional column returned by **ljoin**.

- a Boolean expression p using only bound names. This executes $\mathbf{select}_p$ on the current table, eliminating all rows for which p is *false*.

- $(v_1, \ldots, v_l) \leftarrow e$ where $v_1, \ldots, v_l$ are free variables and e is an expression involving only bound names that evaluates to an l-tuple. For expressions evaluating to a 1-tuple the parentheses are omitted and the term is written as $v \leftarrow e$. This executes $\mathbf{map}_f$, where f is a function mapping a k-tuple of the current table to a $(k+l)$-tuple. The new tuple elements are computed according to the expression e and the names $v_1, \ldots, v_l$ are bound to it.

- **groupby**(*vars*, *exprs*). Group the current table by the named columns given in *vars*, producing one row for each distinct tuple and computing additional columns from aggregate expressions, binding names to them. The argument *vars* is a comma-separated list of bound names and *exprs* is a comma-separated list of terms of the form $(v_1, \ldots, v_t) \leftarrow a(r)$ where $v_1, \ldots, v_t$ are free names, a is an aggregate function returning a t-tuple and r is an expression involving only bound names. For aggregate functions returning a 1-tuple the parentheses are omitted and the term is written as $v \leftarrow a(r)$.

  groupby is executed as follows.

  1. Apply a **map** to the current table. It outputs the columns in *vars* for a total of m columns, followed by additional columns corresponding to aggregate functions used in the expressions in *exprs*. For each invocation $a(r)$ of an aggregate function $a = \mathbf{agg}_{e,\oplus,f}$ where r is an expression involving the current row, a sub-tuple is appended containing the tuple $e(r)$ obtained by evaluating the expression r and applying the encoding function e.
  2. Execute $\mathbf{group}_{m,\oplus}$ where $\oplus$ is a binary operation that applies the binary operations of all involved aggregate functions, each operating on their corresponding sub-tuple.
  3. Apply $\mathbf{map}_h$. The function h is the identity on the first m columns. For each of the terms $(v_1, \ldots, v_t) \leftarrow a(r)$ in *exprs* where $a = \mathbf{agg}_{e,\oplus,f}$, it produces a tuple $(v_1, \ldots, v_t)$ by applying the final function f to the corresponding sub-tuple.

  The result of **groupby** is a table with only the following columns:

  - the columns specified in *vars*;
  - all columns v_i in the expressions $(v_1, \ldots, v_t) \leftarrow a(r)$ in *exprs*.

All other column names are unbound so that they are free in the following terms.

Note that this notation not only specifies the resulting table but the exact sequence of operations to be performed. For the analysis of relational algorithms it is important to take into account the size of all intermediate tables produced while evaluating a multiset comprehension expression. For example, let A be a table of pairs $(i, i+1)$ for $i = 1, \ldots, n$. It has n rows. Consider the following two expressions:

$$B \leftarrow [(i,k) \mid (i,j) \in\!\!\in A, (j,k) \in\!\!\in A]$$

and

$$C \leftarrow [(i,l) \mid (i,j) \in\!\!\in A, (k,l) \in\!\!\in A, j = k]$$

The results B and C are identical: both tables have $n-1$ rows containing the pairs $(i, i+2)$ for $i = 1, \ldots, n-1$. The expression for B computes an intermediate table with $n-1$ rows containing triples $(i, i+1, i+2)$ and in the final step eliminates the unnecessary middle column. The expression for C first computes the Cartesian product of A with itself, a table with n^2 quadruples, and then filters the result down to $n-1$ rows using the predicate $j = k$. This is an important difference between our notation and SQL, which does not specify exactly how computations occur, but leaves this to the query optimiser of a database management system.

The example in Figure 2.1 shows step by step how a more involved multiset comprehension expression is evaluated. It uses the aggregate functions avg() and min() where avg() is the average function defined on page 26 and min = $\mathbf{agg}_{\mathrm{id,min,id}}$, i.e. min($x$) computes the minimum of a multiset x of integers.

2.4.5 Simulating arrays of tables

The Database Machine has a fixed number of table registers. However, some algorithms can be most elegantly expressed using unbounded arrays of tables. These can readily be simulated by storing the whole array in a single table, using an extra column as a table index. Let Y be a table with tuples $(i, \tilde{x})$ and X be a table with tuples $\tilde{x}$ where $\tilde{x}$ stands for any fixed number of integers $x_1, \ldots, x_k$. Then the use of $Y_0, Y_1, \ldots$ in multiset comprehension expressions like an unbounded array of k-tables is defined according to Figure 2.2.

2.5 Basic techniques for the Database Machine

In this section we present basic techniques for computations on the Database Machine that are either of independent interest or will be useful in later chapters.

Figure 2.1: This example demonstrates how a multiset comprehension expression is interpreted in terms of the basic relational operations.

Given $s = 2$, $\quad A = \begin{bmatrix} 4 & 1 & 5 \\ 1 & 2 & 4 \\ 6 & 2 & 3 \\ 3 & 3 & 2 \end{bmatrix}$, $\quad B = \begin{bmatrix} 2 & 1 & 6 \\ 3 & 2 & 3 \\ 3 & 3 & 4 \\ 3 & 4 & 9 \\ 2 & 5 & 1 \end{bmatrix}$, evaluating

$$
\begin{aligned}
&[(g, s \cdot x + m) \mid (i, j, k) \in A, (j, *, l) \in B, g \leftarrow i + l \\
&\qquad , \mathbf{groupby}(g, m \leftarrow \min(k), x \leftarrow \mathrm{avg}(l \cdot i)) \\
&\qquad , m < 3]
\end{aligned}
$$

$[()] \xRightarrow[\textbf{join}]{(i,j,k)\in A}$

i	j	k
4	1	5
1	2	4
6	2	3
3	3	2

$\xRightarrow[\textbf{join}]{(j,*,l)\in B}$

i	j	k	$*$	l
1	2	4	1	6
1	2	4	5	1
6	2	3	1	6
6	2	3	5	1
3	3	2	2	3
3	3	2	3	4
3	3	2	4	9

$\xRightarrow[\textbf{map}]{g\leftarrow i+l}$

i	j	k	$*$	l	g
1	2	4	1	6	7
1	2	4	5	1	2
6	2	3	1	6	12
6	2	3	5	1	7
3	3	2	2	3	6
3	3	2	3	4	7
3	3	2	4	9	12

$\xRightarrow[\text{1. } \textbf{map}]{\textbf{groupby}(\cdots)}$

g	$e_{\min}(k)$	$e_{\mathrm{avg}}(l \cdot i)$	
7	4	6	1
2	4	1	1
12	3	36	1
7	3	6	1
6	2	9	1
7	2	12	1
12	2	27	1

$\xRightarrow[\text{2. } \textbf{group}]{\textbf{groupby}(\cdots)}$

g	$\bigoplus_{\min}$	$\bigoplus_{\mathrm{avg}}$	
2	4	1	1
6	2	9	1
7	2	24	3
12	2	63	2

$\xRightarrow[\text{3. } \textbf{map}]{\textbf{groupby}(\cdots)}$

g	m	x
2	4	1
6	2	9
7	2	8
12	2	31

$\xRightarrow[\textbf{select}]{m<3}$

g	m	x
6	2	9
7	2	8
12	2	31

$\xRightarrow[\textbf{join}_0]{\text{using } s}$

g	m	x	s
6	2	9	2
7	2	8	2
12	2	31	2

$\xRightarrow[\textbf{map}]{(g,x)}$

g	$s \cdot x + m$
6	20
7	18
12	64

Figure 2.2: Notation for simulating arrays of tables on a Database Machine

Notation	Defined as
$X \leftarrow Y_i$	$X \leftarrow [\tilde{x} \mid (i, \tilde{x}) \in Y]$
$\tilde{x} \in Y_i$	$(i, \tilde{x}) \in Y$
$Y_i \leftarrow X$	$Y \leftarrow [(i', \tilde{x}) \mid (i', \tilde{x}) \in Y, i' \neq i] \uplus [(i, \tilde{x}) \mid \tilde{x} \in X]$
delete Y_i	$Y \leftarrow [(i', \tilde{x}) \mid (i', \tilde{x}) \in Y, i' \neq i]$

2.5.1 Multiset difference

Definition 2.5.1. Let $A, B \in [\![\mathbb{N}_0^k]\!]$ be two tables. The *multiset difference* $D = A \setminus B$ is defined as $D \in [\![\mathbb{N}_0^k]\!]$ where for all $d \in \mathbb{N}_0^k$

$$D(d) := \begin{cases} A(d) & B(d) = 0 \\ 0 & \text{otherwise} \end{cases}$$

Theorem 2.5.1. *Algorithm 1 computes the multiset difference of two tables A, B using a constant number of operations using linear space.*

Proof. The algorithm first computes the multiplicity for each unique element in A and $A \uplus B$ using the aggregate function count $:= \mathbf{agg}_{1,+,\mathrm{id}}$ where 1 is the function mapping the zero-tuple () to the number 1. It then computes a table U of unique elements of A not present in B: they are characterised by having the same multiplicity in A and $A \uplus B$. By joining this with A, the elements of A are returned with their original multiplicities.

Algorithm 1 Computing the multiset difference of two tables

1: **function** MULTISETDIFFERENCE(A, B)
2: $C_1 \leftarrow [(a_1, \ldots, a_k, c) \mid (a_1, \ldots, a_k) \Subset A, \mathbf{groupby}(a_1, \ldots, a_k, c \leftarrow \text{count}())]$
3: $C_2 \leftarrow [(a_1, \ldots, a_k, c) \mid (a_1, \ldots, a_k) \Subset A \uplus B, \mathbf{groupby}(a_1, \ldots, a_k, c \leftarrow \text{count}())]$
4: $U \leftarrow [(a_1, \ldots, a_k) \mid (a_1, \ldots, a_k, c) \Subset C_1, (a_1, \ldots, a_k, c) \Subset C_2]$
5: **return** $[(a_1, \ldots, a_k) \mid (a_1, \ldots, a_k) \Subset U, (a_1, \ldots, a_k) \Subset A]$
6: **end function**

The sizes of C_1 and C_2 are not larger than the input. Since the elements of both C_1 and C_2 are unique, the join to compute U cannot increase the number of rows. The final join in the **return** statement returns a subset of A, so overall space usage stays linear in the input size, as claimed. □

2.5.2 Simulating left outer join

Theorem 2.5.2. *The operation* **ljoin** *(left outer join) can be expressed using a constant number of the other operations, utilising temporary space linear in the input.*

Proof. For nonnegative integers k, l, m with $m \leq k, l$ let $A \in [\![\mathbb{N}_0^k]\!]$ and $B \in [\![\mathbb{N}_0^l]\!]$ be two tables. The operation $\mathbf{ljoin}_m(A, B)$ is implemented as shown in Algorithm 2.

The result of a left outer join is computed as the union of two tables, J and O. The result of the normal join is formed in J, tagged with a one in the additional column. Table O contains all rows of A not matched in B, tagged with a zero in the additional column. To compute it, Algorithm 2 first computes the multiset difference between the projections of A and B on the first m-tuple. This works the same way as in Algorithm 1. Line 7 uses this to compute O. Note that each row in C_1 and C_2 is unique, so that the join between them creates one row for each matching tuple $(a_1, \ldots, a_m)$. The final join with A returns the full unmatched rows in A with their original multiplicities. The sizes of the other temporary tables are bounded as claimed: $[C_1| \leq |P_1| = |A|$ and $|C_2| \leq |P_2| = |A| + |B|$. □

Algorithm 2 Implementing outer join using other operations

function $\textsc{LeftOuterJoin}_m(A, B)$
  $J \leftarrow [(a_1, \ldots, a_k, b_{m+1}, \ldots, b_l, 1) \mid (a_1, \ldots, a_k) \in A, (a_1, \ldots, a_m, b_{m+1}, \ldots, b_l) \in B]$
  $P_1 \leftarrow [(a_1, \ldots, a_m) \mid (a_1, \ldots, a_k) \in A]$
  $P_2 \leftarrow P_1 \uplus [(b_1, \ldots, b_m) \mid (b_1, \ldots, b_l) \in B]$
  $C_1 \leftarrow [(a_1, \ldots, a_m, c) \mid (a_1, \ldots, a_m) \in P_1, \mathbf{groupby}(a_1, \ldots, a_m, c \leftarrow \text{count}())]$
  $C_2 \leftarrow [(a_1, \ldots, a_m, c) \mid (a_1, \ldots, a_m) \in P_2, \mathbf{groupby}(a_1, \ldots, a_m, c \leftarrow \text{count}())]$
  $O \leftarrow [(a_1, \ldots, a_k, \underbrace{0, \ldots, 0}_{l-m \text{ times}}, 0) \mid (a_1, \ldots, a_m, c) \in C_1, (a_1, \ldots, a_m, c) \in C_2, (a_1, \ldots, a_k) \in A]$
  return $J \uplus O$
end function

Like the **join** operation, **ljoin** can produce an output whose size is quadratic in the input size. The claim on temporary space usage only refers to the auxiliary tables used in the simulation. It will be important when this theorem is used as part of a larger construction in the proof of Theorem 3.5.11.

2.5.3 Generating sequences

Generating a sequence of consecutive integers is a basic building block for constructions that appear later in this thesis. The technique presented here is also generalised and used in the next section.

Theorem 2.5.3. *For a positive integer n, a table containing the integers $0, \ldots, n-1$ can be generated using $\mathrm{O}(\log \log n)$ operations and space $\mathrm{O}(n)$.*

To prove this, we present an algorithm that repeatedly joins a table of all i-bit integers with itself to generate a table of all $2i$-bit integers (Algorithm 3). We first prove three lemmas about this algorithm.

Lemma 2.5.4. *At the beginning of each iteration of the loop of Algorithm 3, table S contains all i-bit integers and $i \leq b$. Before the* **return** *statement, S contains all b-bit integers where $b = \lceil \log n \rceil$.*

Proof. The claimed loop invariant holds after initialisation: $S = [0, 1]$ and $i = 1$. Joining S with itself creates all combinations of an i-bit integer s with another i-bit integer s', which are combined to create all $2i$-bit integers. Observe that $i \leq b$ at the beginning of the loop and that this invariant is preserved: i is only replaced by $2i$ if $2i < b$. The loop finishes with $i \leq b \leq 2i$.

For the last step, each integer is split into a pair (l, h) where l contains the lower $b-i$ bits and h the rest of the bits. For each value of h there are 2^{b-i} rows with $l = 0, \ldots, 2^{b-i} - 1$. Joining this table with itself on h multiplies the number of rows by 2^{b-i} so that it reaches 2^b. Combining the bits from the individual components creates the desired table of all b-bit integers. □

Lemma 2.5.5. *Algorithm 3 executes $\mathrm{O}(\log \log n)$ operations.*

Algorithm 3 Generating a sequence of consecutive integers

```
1: function GenerateSequence(n)
2:   if n = 0 then
3:     return []
4:   end if
5:   S ← [0]
6:   if n = 1 then
7:     return S
8:   end if
9:   S ← S ⊎ [1]
10:  i ← 1
11:  b ← ⌈log n⌉
12:  while 2i < b do
13:    S ← [s + 2^i s' | s ∈ S, s' ∈ S]
14:    i ← 2i
15:  end while
16:  S ← [(s mod 2^{b−i}, ⌊s/2^{b−i}⌋) | s ∈ S]
17:  S ← [l + 2^{b−i} h + 2^i l' | (l, h) ∈ S, (l', h) ∈ S]
18:  return [s | s ∈ S, s < n]
19: end function
```

Proof. The loop starts with $i = 1$ and doubles i at each step until it reaches $b/2$, which takes $\mathrm{O}(\log b)$ iterations. Since $b = \lceil \log n \rceil$, this amounts to $\mathrm{O}(\log \log n)$ operations. □

Lemma 2.5.6. *Algorithm 3 uses space* $\mathrm{O}(n)$.

Proof. The algorithm uses a single table S. By Lemma 2.5.4, its size is $2^i \leq 2^b$ throughout the loop. The next assignment leaves the number of rows unchanged and the last assignment brings its size to 2^b rows where $b = \lceil \log n \rceil$. Since $\lceil \log n \rceil < \log n + 1$, we have $|S| < 2n$, or $|S| = \mathrm{O}(n)$, as claimed. The expression in the **return** instruction creates a table with n rows. □

Proof of Theorem 2.5.3. Lemma 2.5.4 establishes that Algorithm 3 creates the sequence $0, \ldots, 2^b - 1$. The expression in the **return** statement filters the list to the exact required size. Lemmas 2.5.5 and 2.5.6 establish the claimed bounds on time and space. □

2.5.4 Converting a multiset to a set

We now show how to make all elements of a multiset unique, turning it into a set with the same cardinality. To do this, Algorithm 4 generates a sequence of numbers for each distinct element in parallel.

Theorem 2.5.7. *Algorithm 4 takes a table T of integers and returns a table with the same cardinality and an additional column such that each row is unique. The result contains for each element $t \in T$ with multiplicity m the distinct pairs $(t, 0), (t, 1), \ldots, (t, m-1)$. For an input table of n rows, the algorithm executes $\mathrm{O}(\log \log n)$ operations and uses space $\mathrm{O}(n)$.*

We split the proof into three lemmas, establishing a loop invariant and bounds on time and space.

Algorithm 4 Converting a multiset to a set

1: **function** ConvertToSet(T)
2: $C \leftarrow [(t, c) \mid t \Mapsto T, \textbf{groupby}(t, c \leftarrow \text{count}())]$
3: $B \leftarrow [(t, \max\{1, \lceil \log c \rceil\}) \mid (t, c) \Mapsto C]$
4: $S \leftarrow [(t, 1, (b = 1\ ?\ 0 : 1), 0, 0) \mid (t, b) \Mapsto B]$ ▷ C-style (*condition* ? *iftrue* : *iffalse*)
5: $S \leftarrow S \uplus [(t, i, j, j, 1 - j) \mid (t, i, j, *, *) \Mapsto S]$
6: **repeat**
7: $p \leftarrow |S|$
8: $S \leftarrow \big[(t, i', j', l', h') \mid (t, i, j, l_1, h) \Mapsto S, (t, i, j, l_2, h) \Mapsto S, (t, b) \Mapsto B$
$\quad , i' \leftarrow i + j, j' \leftarrow \min\{i', b - i'\}$
$\quad , s \leftarrow l_1 + 2^j h + 2^i l_2$
$\quad , l' \leftarrow s \bmod 2^{j'}, h' \leftarrow \lfloor s/2^{j'} \rfloor\big]$
9: **until** $|S| = p$
10: **return** $[(t, h) \mid (t, *, *, *, h) \Mapsto S, (t, c) \Mapsto C, h < c]$
11: **end function**

Lemma 2.5.8. *Table S obeys the following loop invariant: For each pair $(t, b) \in B$ it contains 2^i quintuples (t, i, j, l, h) where i and j are constant with $j \le i \le b$ and*

$$j = \begin{cases} i & \text{if } 2i \le b \\ b - i & \text{otherwise.} \end{cases}$$

The pairs (l, h) run through all combinations of integers $0 \le l < 2^j$ and $0 \le h < 2^{i-j}$. Another way to describe this is that the tuples run through all i-bit integers, splitting them into their j lower bits and $i - j$ higher bits.

Proof. S is initialised with a table containing two rows for each distinct t. They have $i = 1$, and for the value of j the two cases $b = 1$ and $b > 1$ are distinguished to obey the invariant. Inside the loop, table S is joined with itself on t, i, j, and h. By the invariant, for each distinct t each of the 2^i rows in the first copy of S is matched by 2^j rows in the second copy of S, producing 2^{i+j} rows. This intermediate table is then joined with B on t, keeping the same number of rows because B has exactly one row for each distinct t.

For each distinct t, $i' = i + j$ reflects the new number of rows as claimed. The invariant condition on j guarantees that $i' \le b$. We have $j' = \min\{i', b - i'\} \le i'$ as claimed and

$$j' = \begin{cases} i' & \text{if } 2i' \le b \\ b - i' & \text{otherwise} \end{cases}$$

because $2i' \le b \Leftrightarrow i' \le b - i'$. The value s is computed to run through $0, \ldots, 2^{i+j} - 1$ for each t and then split into the lower j' bits in l' and the rest of the bits in h' as claimed. This shows that the invariant is maintained in each iteration. □

Lemma 2.5.9. *For an input table of n rows, Algorithm 4 executes $O(\log \log n)$ operations. After exiting the loop, the tuples $(t, i, j, l, h) \in S$ have $i = b$ and $j = 0$ for all $(t, b) \in B$.*

Proof. In each iteration of the loop, for every t the value of i is doubled until it reaches b. The loop terminates when the size of S does not change. This happens when $i = b$

and $j = 0$ for all t. The number of iterations is therefore determined by the highest multiplicity occurring in the input multiset T. The worst case is T containing one element with multiplicity n, in which case $b \leq \log n+1$ and the loop takes $\mathrm{O}(\log \log n)$ iterations. □

Lemma 2.5.10. *For an input table of n rows, Algorithm 4 uses space $\mathrm{O}(n)$.*

Proof. By Lemma 2.5.8, table S contains for each $(t, b) \in B$ at most 2^b rows and we have $b \leq \log T(t) + 1$. Therefore, $|S| \leq \sum_{(t,b)\in B} 2^b \leq \sum_{t\in T} 2 \cdot T(t) = 2|T|$. For an input table of n rows the space usage of the algorithm stays within $\mathrm{O}(n)$, as claimed. □

Proof of Theorem 2.5.7. Algorithm 4 first reduces the input multiset T to a set C of pairs (t, c) containing the multiplicities of all distinct elements of T. Table B receives for each distinct element $t \in T$ the pair (t, b) where b is the number of bits required to store the sequence of integers generated for t. The algorithm then essentially executes Algorithm 3 in parallel for each distinct $t \in T$.

At the end of the loop, by Lemma 2.5.9 and Lemma 2.5.8, the value h in each subset of tuples (t, i, j, l, h) for fixed t runs through $0, \ldots, 2^b - 1$ and the expression in the **return** instruction transforms and filters them to create the correct number of pairs for each distinct t. Lemma 2.5.9 and Lemma 2.5.10 establish the claimed bounds on the number of operations and on space usage. □

2.5.5 Computing the rank

Another basic relational technique is determining the rank of a number within a multiset, i.e. the number of other elements that are smaller than it. This is implemented as Algorithm 5 and is a prerequisite for the sorting algorithm presented in the next section, which is an application of the well-known counting sort [54, pp. 75 ff.]. Instead of simply counting, Algorithm 5 borrows the idea of comparing elements bit by bit from radix sort [54, section 5.2.5].

Both computing the rank and its application to sorting are not used in the remainder of this thesis. They are presented here because they are of independent interest. The algorithm for computing the rank shows how to trade the quadratic space usage of a naive implementation for a logarithmic number of steps, using the fact that the Database Machine has a logarithmic bound on its row size. This makes the algorithm Big Data-practical in the sense of Definition 2.3.3.

Theorem 2.5.11. *For each element $x \Subset T$, Algorithm 5 counts the number of elements $y \Subset T$ with $y < x$ and returns a pair (x, c) where c is this count. For an input table of n rows, it executes $\mathrm{O}(\log n)$ operations and uses space $\mathrm{O}(n)$.*

This transformation could almost be trivially expressed as

$$[(x, c) \mid x \Subset T, y \Subset T, y < x, \mathbf{groupby}(x, c \leftarrow \mathrm{count}())],$$

but this forms the Cartesian product of T with itself, which produces n^2 rows and hence is not Big Data-practical. Also, it eliminates the minimum element from T.

Algorithm 5 Computing the rank of a number in a multiset

```
1: function ComputeRank(T)
2:   if |T| = 0 then
3:     return T
4:   end if
5:   R ← [(x, x, 0) | x ∈ T]
6:   repeat
7:     C ← [(h′, c) | (∗, h, ∗) ∈ R, h′ ← ⌊h/2⌋, groupby(h′, c ← sum(1 − (h mod 2)))]
8:     R ← [(x, h′, s + (h mod 2 = 1 ? c : 0)) | (x, h, s) ∈ R, h′ ← ⌊h/2⌋, (h′, c) ∈ C]
9:   until |C| = 1
10:  return [(x, s) | (x, ∗, s) ∈ R]
11: end function
```

In order to save space, we count the smaller elements in multiple steps, comparing them bit by bit. For fixed $x \in T$, let

$$S_i(x) := \left[y \in T \mid \lfloor y/2^i \rfloor \bmod 2 < \lfloor x/2^i \rfloor \bmod 2 \wedge \lfloor y/2^{i+1} \rfloor = \lfloor x/2^{i+1} \rfloor\right]$$

Algorithm 5 uses the following:

Lemma 2.5.12. *For fixed $x \in T$,*

$$\left|[y \in T \mid y < x]\right| = \sum_i |S_i(x)|.$$

Proof. If x and y are nonnegative integers with $y < x$, there is a unique most significant bit position i at which the binary representations of x and y differ. $S_i(x)$ contains those $y \in T$ with $y < x$ for which this most significant bit position equals i. This implies that the multisets $S_i(x)$ are disjoint for fixed x and the claim follows. □

Lemma 2.5.13. *After the i-th iteration of the loop in Algorithm 5, R contains for each $x \in T$ a triple (x, h, s) where $h = \lfloor x/2^i \rfloor$ and $s = \sum_{j<i} |S_j(x)|$. In particular, R has the same number of rows as the input T throughout the algorithm.*

Proof. The claim is true for the initial table R before entering the loop, i.e. after $i = 0$ iterations. Assume it is true after i iterations. In iteration $i+1$, C is computed to contain exactly one pair (h', c) for each unique value $h' = \lfloor x/2^{i+1} \rfloor$ over all $x \in T$. c is the count of those $x \in T$ with $\lfloor x/2^i \rfloor \bmod 2 = 0$. When computing R, first note that in the join between R and C on h' each row of R is matched by exactly one row of C so that the number of rows in R is preserved. For each tuple $(x, h, s) \in R$, the new tuple (x, h', s') satisfies the claim after iteration $i+1$: we have $h' = \lfloor h/2 \rfloor = \lfloor x/2^{i+1} \rfloor$ by assumption. For the new value of s' there are two cases: if $\lfloor x/2^i \rfloor \bmod 2 = 1$ then $|S_i(x)| = c$, otherwise $|S_i(x)| = 0$. In both cases, $s' = s + |S_i(x)| = \sum_{j<i+1} |S_j(x)|$ by assumption. The claim follows by induction on i. □

Lemma 2.5.14. *For an input table of n rows, Algorithm 5 executes $\mathrm{O}(\log n)$ operations.*

Proof. The basic assumption of our computational model is that for an input table of n rows the individual entries are $\mathrm{O}(\log n)$ bits wide. Thus, by Lemma 2.5.13, after at most

$\mathrm{O}(\log n)$ iterations the tuples $(x, h, s) \Subset R$ will all have $h = 0$, at which time C will contain the single pair $(0, |T|)$ and the loop will terminate. □

Lemma 2.5.15. *For an input table of n rows, Algorithm 5 uses space $\mathrm{O}(n)$.*

Proof. Table R starts with n rows and stays the same size throughout the algorithm by Lemma 2.5.13. Table C is the result of reducing R, so $|C| \leq |R|$ at all times. □

Proof of Theorem 2.5.11. Algorithm 5 computes the sum in Lemma 2.5.12 for all x in parallel, essentially making it a parallel version of radix sort. Lemma 2.5.13 establishes that it outputs the correct result while Lemma 2.5.14 and Lemma 2.5.15 establish the claimed bounds on time and space. □

2.5.6 Sorting

Since we have defined a table to be an unordered multiset, it is inherently not possible to sort a table. We therefore define sorting a table as adding a column of indices that are consecutive integers starting at zero such that the rows are numbered in ascending order. More formally:

Definition 2.5.2. Let $T \in [\![\mathbb{N}_0]\!]$ be a table of integers. We define *sorting* as the process of creating from T a table $S \in [\![\mathbb{N}_0^2]\!]$ with the following properties:

1. $[x \mid (x, *) \Subset S] = T$, i.e. the first column of S contains the values from the input table T with the same multiplicities.
2. $[i \mid (*, i) \Subset S] = [0, 1, \ldots, |T| - 1]$, i.e. the second column of S contains consecutive integers starting at zero.
3. For all $(x_1, i_1), (x_2, i_2) \in S$: $i_1 < i_2 \Rightarrow x_1 \leq x_2$.

We call the elements of the second column the *indices*.

With this definition of sorting, the natural choice of an algorithm is to compute the rank of each element; the only refinement required is the treatment of multiple identical elements. If we apply Algorithm 5 to a multiset that is a set, the result is a sorted table according to this definition. If, however, any element x occurs with multiplicity $T(x) > 1$, then there will be $T(x)$ identical rows in the result, violating property 2. Algorithm 6 combines Algorithm 5 and Algorithm 4 to produce the desired consecutive sequence of integer indices.

When talking about sorting in the context of Big Data, usually external sorting algorithms such as polyphase merge sort come to mind [54, Section 5.4.2]. These are expressed in terms of the sequential processing of files larger than memory. These algorithms are unsuitable for our computational model because the Database Machine operates at a higher level of abstraction: it considers operations on multirelations as a whole without requiring – or even allowing – us to know how they are distributed among files or machines. Also, these algorithms are inherently mostly sequential, whereas the Database Machine models parallel computation.

Algorithm 6 Counting Sort for multisets of integers

```
1: function CountingSort(T)
2:     U ← ConvertToSet(T)
3:     S ← ComputeRank(T)
4:     S ← [(x, s) ∈ S, groupby(x, s)]                ▷ make distinct
5:     return [(x, s + i) | (x, i) ∈ U, (x, s) ∈ S]
6: end function
```

Theorem 2.5.16. *For an input table of n rows, Algorithm 6 sorts this table, executing $\mathrm{O}(\log n)$ operations and using a total of $\mathrm{O}(n)$ rows.*

Proof. Algorithm 6 first uses Algorithm 4 to create a table U in which each row is unique and for each $x \in T$, multiple occurrences are indexed with consecutive integers starting at 0. Note that U satisfies property 1 of a sorted table. Next, the output of Algorithm 5 is reduced to its distinct pairs (x, s) where for each x, the index s is the beginning of the desired sequence of integer indices for all copies of x. When joining U with S, each row of U matches exactly one row of S, keeping the same number of rows and preserving property 1. The indices are computed to satisfy property 2. Adding the number of operations and rows used by the two algorithms, the claimed time and space bounds follow from Theorem 2.5.11 and Theorem 2.5.7. □

2.5.7 Prefix computation

Definition 2.5.3. Let $\oplus\colon \mathbb{N}_0 \to \mathbb{N}_0$ be an associative binary operation. A *prefix computation* takes as input a sequence of integers $x_1, \ldots, x_n$ and produces a sequence of integers $s_1, \ldots, s_n$ where $s_k = \bigoplus_{i=1}^{k} x_i$.

Theorem 2.5.17. *A prefix computation with n inputs can be performed in $\mathrm{O}(\log n)$ operations using space $\mathrm{O}(n)$.*

Proof. Algorithm 7 takes as input a table of pairs (i, x_i) where $i = 0, \ldots, n-1$. We will show that it performs a prefix computation on the sequence $x_0, \ldots, x_{n-1}$ and outputs the result as a table of pairs (k, s_k) with $s_k = \bigoplus_{i=0}^{k} x_i$. We call the first element of each pair the index.

Algorithm 7 Performing prefix computation on a sequence of integers

```
1: function ComputePrefixes⊕(T)
2:     S ← [(0, x) | (0, x) ∈ T]
3:     C ← T
4:     j ← 1
5:     repeat
6:         S′ ← [(k + j, s ⊕ c) | (k, s) ∈ S, (k + 1, c) ∈ C]
7:         S ← S ⊎ S′
8:         C ← [(k, c ⊕ c′) | (k, c) ∈ C, (k + j, c′) ∈ C]
9:         j ← 2 · j
10:    until |C| = 0
11:    return S
12: end function
```

The following invariants hold at the beginning of each iteration of the loop in Algorithm 7:

1. S contains exactly one pair (k, s_k) for $0 \leq k < j$ and $k < n$, satisfying $s_k = \bigoplus_{i=0}^{k} x_i$
2. C contains exactly one pair (k, c_k) for $0 \leq k \leq n - j$, satisfying $c_k = \bigoplus_{i=k}^{k+j-1} x_i$

Both invariants hold after initialisation. Assume they hold at the beginning of the loop for some j. The rows to be added to S are computed in S'. If $j < n$, by assumption S contains rows (k, s) for $k = 0, \ldots, j-1$, so $k + j = j, \ldots, 2j - 1$ and the indices of $S \uplus S'$ run through a contiguous sequence of integers. A row with index $k + j$ is produced in S' only if $(k + 1, c) \in C$ which, by assumption, is the case for $k + 1 \leq n - j$, implying $k + j \leq n - 1$. Thus, the indices of the added rows will be less than n, as claimed. If $j \geq n$, no rows are produced because of the join with table C: We have $k + 1 > n - j \leq 0$ for any k. For $(k, s) \in S$ and $(k + 1, c) \in C$, the newly created row with index $k + j$ has $s \oplus c = \bigoplus_{i=0}^{k} x_i \oplus \bigoplus_{i=k+1}^{k+j} x_i = \bigoplus_{i=0}^{k+j} x_i$, showing that invariant 1 holds. For $(k, c) \in C$ and $(k+j, c') \in C$, each newly computed row $(k, c \oplus c')$ for table C satisfies, by assumption, $c \oplus c' = \bigoplus_{i=k}^{k+j-1} x_i \oplus \bigoplus_{i=k+j}^{k+2j-1} x_i = \bigoplus_{i=k}^{k+2j-1} x_i$ and $k + j \leq n - j$, from which we conclude $k \leq n - 2j$ as claimed in invariant 2.

Algorithm 7 starts with $j = 1$ and doubles j on each iteration of the loop. After $\mathrm{O}(\log n)$ iterations we have $j > n$ at which time C is empty (invariant 2) and the loop terminates. By invariant 1, S contains the result of the prefix computation at this time. We have also established that both tables have at most n rows throughout the computation, implying that total space usage is $\mathrm{O}(n)$ rows. □

Chapter 3

A database assembly language

The Database Machine as defined in Chapter 2 has a small set of basic operations, but allows the use of powerful "database" functions in them. In this chapter we define an even simpler machine that can be regarded as a kind of assembly language for relational databases, the Relational Machine. It uses only three "heavy" operations that require communication between different table rows and otherwise offers data-parallel mapping primitives with very basic functionality like increment, decrement and bit shift. The main result of this chapter is that the Database Machine can be simulated by the Relational Machine (Theorem 3.5.11).

3.1 Relational tables

The basic unit of computation in the Relational Machine is the relational table.

Definition 3.1.1. A *relational table*, for short *rtable*, is a set of *rows*. Each row is a pair (k, v) where k is an s-tuple of nonnegative integers called the key tuple and v is a t-tuple of nonnegative integers called the value tuple for some $s, t \in \mathbb{N}_0$. We refer to a 0-tuple as an *empty tuple* and denote it by $()$. The tuple lengths s and t are the same for all rows of an rtable; to specify them, we speak of an (s, t)-rtable. Tuple lengths can be different for different rtables. Within an rtable, the key tuples k are unique whereas there is no such restriction on the value tuples v. We call the collection of all elements at a fixed position within either the key tuple or the value tuple of an rtable a *column*. An rtable is *empty* if it has no rows, i.e. it is the empty set. The *cardinality* of an rtable is the number of rows, i.e. its cardinality as a set.

Note that this definition implies that a relational table can be regarded as a function mapping key tuples to value tuples.

Definition 3.1.2. Let T be an (s, t)-rtable. The *row width* of T is defined as

$$\sum_{i=1}^{s} \max_{((k_1,\ldots,k_s),v)\in T} \lceil \log(k_i + 1) \rceil + \sum_{j=1}^{t} \max_{(k,(v_1,\ldots,v_t))\in T} \lceil \log(v_j + 1) \rceil.$$

The row width is defined in terms of the maximum value in each column so that it equals the number of bits needed per row to store the whole rtable in a fixed-width format.

3.2 Instruction set

In this section we define a set of operations for manipulating rtables. Together they comprise the instruction set of the Relational Machine, which we will formally define in Section 3.3. The Relational Machine has a set of registers, each storing an rtable, and manipulates these registers with the instructions defined here. The variables A, B, and C in the following definitions refer to (not necessarily distinct) registers.

3.2.1 Relational operations

The operations in this section take one or two rtables as an input and produce an rtable with empty value tuples as an output. They have in common that they have to consider multiple rows, whether in the same rtable or in different rtables, to compute their result.

- `JOIN` A, B, C: join rtables A and B on their value, storing the result in C. For tuples $a = (a_1, a_2, \ldots, a_s)$ and $b = (b_1, b_2, \ldots, b_t)$ let $a +\!+ b$ denote the concatenation $(a_1, a_2, \ldots, a_s, b_1, b_2, \ldots, b_t)$. Then this instruction performs

$$C \leftarrow \{(a +\!+ b, ()) \mid (a, v_a) \in A, (b, v_b) \in B, v_a = v_b\}$$

- `RANGE` A: turn the set of values into keys and clear the values:

$$A \leftarrow \{(v, ()) \mid (k, v) \in A\}$$

  Note that this effectively eliminates duplicates since the result is a set, forcing the key tuples to be unique.

- `SINGLES` A: erase all rows (k, v) of A for which v is not unique and set the value of the remaining rows to the 0-tuple $()$:

$$A \leftarrow \{(k, ()) \mid (k, v) \in A \text{ such that } v \text{ is unique among the values of } A\}$$

`JOIN` and `RANGE` are well-known relational operations. `SINGLES` could be expressed as an SQL query; introducing it as a primitive out of which other operations can be constructed is, to the best of our knowledge, a new idea. We will see in the remainder of this chapter that these three relational primitives, together with the mapping primitives introduced below, are enough to simulate all operations of the Database Machine. Note that `SINGLES` cannot be expressed in terms of the other operations.

3.2.2 Mapping primitives

The operations in this section form the building blocks for computing arbitrary functions on each row of an rtable in parallel. They modify an rtable A in place by applying the same operation to each row.

We describe each instruction as a function applied to every row of A. All instructions halt the machine if the number of key tuple elements or value tuple elements is not sufficient

to carry out the operation. Note that all operations preserve the uniqueness of the key tuples (only the first one modifies it at all).

Manipulating the key

- `ROTK` A: rotate key tuple to the left.

$$((k_1, k_2, \ldots, k_s), v) \mapsto ((k_2, \ldots, k_s, k_1), v)$$

Key to value

- `COPY` A: copy first key element to value tuple.

$$((k_1, k_2, \ldots, k_s), (v_1, v_2, \ldots, v_t)) \mapsto ((k_1, k_2, \ldots, k_s), (k_1, v_1, v_2, \ldots, v_t))$$

Manipulating the value

- `ROTV` A: rotate value tuple left.

$$(k, (v_1, v_2, \ldots, v_t)) \mapsto (k, (v_2, \ldots, v_t, v_1))$$

- `SWAP` A: swap first two elements of value tuple.

$$(k, (v_1, v_2, v_3, \ldots, v_t)) \mapsto (k, (v_2, v_1, v_3, \ldots, v_t))$$

- `INC` A: increment first element of value tuple.

$$(k, (v_1, v_2, \ldots, v_t)) \mapsto (k, (v_1 + 1, v_2, \ldots, v_t))$$

- `DEC` A: decrement first element of value tuple.

$$(k, (v_1, v_2, \ldots, v_t)) \mapsto \begin{cases} (k, (v_1 - 1, v_2, \ldots, v_t)) & v_1 > 0 \\ (k, (v_1, v_2, \ldots, v_t)) & v_1 = 0 \end{cases}$$

- `SHIFT` A: shift the least significant bit from the first value element into the second value element.

$$(k, (v_1, v_2, v_3, \ldots, v_t)) \mapsto (k, (\lfloor v_1/2 \rfloor, 2v_2 + (v_1 \bmod 2), v_3, \ldots, v_t))$$

- `PUSH` A: extend the value tuple by inserting a zero as the first element.

$$(k, (v_1, v_2, \ldots, v_t)) \mapsto (k, (0, v_1, v_2, \ldots, v_t))$$

- `POP` A: remove the first element of the value tuple.

$$(k, (v_1, v_2, \ldots, v_t)) \mapsto (k, (v_2, \ldots, v_t))$$

- `CSWAP` A: conditional swap. If the first value element is nonzero, swap the second and third elements.

$$(k, (v_1, v_2, v_3, v_4, \ldots, v_t)) \mapsto \begin{cases} (k, (v_1, v_3, v_2, v_4, \ldots, v_t)) & v_1 \neq 0 \\ (k, (v_1, v_2, v_3, v_4, \ldots, v_t)) & v_1 = 0 \end{cases}$$

- `CINC` A: conditional increment. If the first value element is nonzero, increment the second element.

$$(k, (v_1, v_2, v_3, \ldots, v_t)) \mapsto \begin{cases} (k, (v_1, v_2 + 1, v_3, \ldots, v_t)) & v_1 \neq 0 \\ (k, (v_1, v_2, v_3, \ldots, v_t)) & v_1 = 0 \end{cases}$$

- `CDEC` A: conditional decrement. If the first value element is nonzero, decrement the second element.

$$(k, (v_1, v_2, v_3, \ldots, v_t)) \mapsto \begin{cases} (k, (v_1, v_2 - 1, v_3, \ldots, v_t)) & v_1 \neq 0 \wedge v_2 \neq 0 \\ (k, (v_1, v_2, v_3, \ldots, v_t)) & \text{otherwise} \end{cases}$$

- `CSHIFT` A: conditional shift. If the first value element is nonzero, shift the least significant bit from the second into the third element.

$$(k, (v_1, v_2, v_3, v_4, \ldots, v_t)) \mapsto \begin{cases} (k, (v_1, \lfloor v_2/2 \rfloor, 2v_3 + (v_2 \bmod 2), v_4, \ldots, v_t)) & v_1 \neq 0 \\ (k, (v_1, v_2, v_3, v_4, \ldots, v_t)) & v_1 = 0 \end{cases}$$

The set of mapping primitives is small, but we are not aiming for minimality at all cost. For example, the non-conditional operations are redundant, but it would clutter the presentation to replace them with their conditional counterpart and force the condition to be true.

3.2.3 Copying and constants

- `MOV` A, B: copy rtable A to rtable B.

$$B \leftarrow A$$

- `TWO` is a read-only $(1, 0)$-rtable containing the two rows $(0, ())$ and $(1, ())$.

- `ONE` is a read-only $(0, 0)$-rtable containing the row $((), ())$. It can be computed by simply applying `RANGE` to a copy of `TWO`, but is predefined for convenience.

3.2.4 Control flow

The following instructions change the control flow of the Relational Machine, to be defined in Section 3.3.

- `JMP` *location*: unconditional jump. Continue execution at *location*.
- `JE` A, *location*: conditional jump. Continue execution at *location* if A contains an empty rtable. Otherwise, continue sequentially.
- `JNE` A, *location*: conditional jump. Continue execution at *location* if A contains a nonempty rtable. Otherwise, continue sequentially.
- `HALT`: halt the machine.

3.3 The Relational Machine

We are now ready to define the Relational Machine.

Definition 3.3.1. A *Relational Machine* consists of a program to be executed and four integer bounds r, l, c_1, and c_2. The program is a finite sequence of instructions from the instruction set defined in Section 3.2, indexed by positive integers. The machine has registers $R_1, \ldots, R_r$, each storing an rtable, and two built-in read-only rtables as described in Section 3.2.3. A computation on the machine takes as input a single rtable and produces as output a single rtable where information is encoded in the key tuples only, i.e. both input and output rtables have the empty tuple () as value tuples. The *input size* is defined as the cardinality of the input rtable. All registers start empty and the input is placed into register R_1. Execution starts at instruction 1 and continues at the following instruction unless control is transferred to a different location by one of the jump instructions or the machine halts. When the machine halts, register R_1 must have empty value tuples and is considered the output. Execution is constrained by the following rules, depending on the input size n:

1. In any register, the maximum length of the key tuple and the maximum length of the value tuple is l.
2. The row width of the input is not larger than $c_1 \cdot \log(n+2)$.
3. The row width of any register is not larger than $c_2 \cdot \log(n+2)$.

If any operation – including placing the input in register R_1 – violates these rules, the machine *crashes* and the output is undefined. An input that does not immediately crash the machine is called *valid*.

When describing algorithms for the Relational Machine, we will use capital letters to signify rtable registers. We assume that different letters stand for different registers and that the total number of registers used is within the register bound r of the machine.

When we claim that a certain computation can be performed by a Relational Machine, we claim that there exist bounds r, l, c_1, and c_2 such that the computation can be performed within these bounds for any input size.

It is important to note that when speaking of solving problems on the Relational Machine, the encoding has to be part of the problem specification. Take, for example, any

problem where the input is a graph, given as an edge list. An edge is specified as a pair of integers where each integer is a vertex identifier, so the problem can be encoded as a two-column rtable. However, the constraint on the row width implies that the vertex IDs cannot be chosen arbitrarily but have to be somewhat densely packed. If the vertex IDs of an n-vertex graph were chosen as 2^i for $i = 1, \ldots, n$, the row width would be $\Theta(n)$ which is not $\mathrm{O}(\log n)$. In this case there is no Relational Machine that could even accept the input for all problem sizes.

Definition 3.3.2. Let $T, S \colon \mathbb{N}_0 \to \mathbb{N}_0$. A Relational Machine is said to compute in *time* $T(n)$ if it halts after executing at most $T(n)$ instructions for any input of size n. It is said to compute in *space* $S(n)$ if for any input of size n the total number of rows in all registers at any point in the computation is at most $S(n)$.

Definition 3.3.3. We call a Relational Machine *Big Data-practical* if it runs, for any input of size n, in polylogarithmic time and $\mathrm{O}(n)$ space. We call it *Small Data-practical* if it runs in polylogarithmic time and polynomial space.

To facilitate the presentation of algorithms for the Relational Machine, we introduce a notation using named columns. It will always be clear from the context whether a name refers to a column of the key tuple or a column of the value tuple.

Definition 3.3.4. A *labelled rtable* is an rtable in which each column has a name. Column names are unique within the key tuple and within the value tuple, but a value column can have the same name as a key column. The constant `TWO` can be used as a labelled rtable with the single key column named c. Table 3.1 defines macros for applying the mapping primitives from Section 3.2.2 to labelled rtables. For labelled rtables A and B with the same set of value column names, $\mathrm{Join}(A, B, C[\mathit{replacements}])$ denotes executing `JOIN` A, B, C on equality of values in columns with corresponding names. The key columns in C inherit the names from A and B unless mentioned in *replacements*. *replacements* is a comma-separated list of rules "$\mathit{newname} \leftarrow A.\mathit{oldname}$" or "$\mathit{newname} \leftarrow B.\mathit{oldname}$" specifying how names from A or names from B are to be changed in the process, respectively, to make all resulting key column names in C unique. $\mathrm{Range}(A)$ denotes executing `RANGE` A on a named rtable, turning value column names into key column names. $\mathrm{Singles}(A)$ denotes executing `SINGLES` A, keeping key column names.

Lemma 3.3.1. *Labelled rtables can be implemented on a Relational Machine such that each of the operations listed in Table 3.1 executes a bounded number of Relational Machine instructions on an rtable A, independent of its size.*

Proof. The Copy() operations are implemented as follows: For each key tuple element to be copied, we use the `ROTK` operation to move that column to the first place, rotating the list of names accordingly, and use `COPY` to copy it to the value tuple, assigning the same or a new name to it as specified. Repeating this allows us to copy an arbitrary combination of elements from the key tuple to the value tuple.

Likewise, we can rearrange the elements of the value tuple by using the `ROTV` and `SWAP` instructions, again keeping track of the names along the way. It is well known that these

Table 3.1: Notation for specifying Relational Machine operations on a labelled rtable A. Each macro stands for applying a suitable combination of `ROTK`, `ROTV`, and `SWAP` to move the specified columns to the beginning of the tuple and then applying the elementary operation from Section 3.2.2 with the same name as the macro.

Macro	Meaning
A.Copy(*list*)	copy key columns to value tuple. *list* is a comma-separated list where each item is either a key column name *name* or an expression *newname* $\leftarrow$ *name*, giving the copied column a new name.
A.Copy($*$)	copy all key columns to value tuple, keeping names
A.Swap(a, b)	swap the values in value columns a and b
A.Inc(a)	increment value column a
A.Dec(a)	decrement value column a
A.Shift(a, b)	shift LSB from value column a to column b
A.Push($a_1, \ldots, a_k$)	create new value columns named $a_1, \ldots, a_k$, initialised with 0
A.Pop($a_1, \ldots, a_k$)	remove value columns $a_1, \ldots, a_k$
A.CSwap(c, a, b)	swap values in value columns a and b if $c > 0$
A.CInc(c, a)	increment value column a if $c > 0$
A.CDec(c, a)	decrement value column a if $c > 0$
A.CShift(c, a, b)	shift LSB from value column a to column b if $c > 0$
A.Rename($b \leftarrow a$)	rename column a to b; no operations executed

two operations are sufficient to create any desired permutation. All operations in Table 3.1 except Copy() and Rename() are implemented by permuting the value tuple such that the specified columns appear at the beginning of the value tuple in the order specified. Then the elementary operation of the same name is executed. Rename() executes no operations; it merely renames a column for notational convenience.

Recall that in any Relational Machine, the lengths of the key and value tuple are bounded by a constant. Hence, the number of `ROTK` instructions required for Copy() and the number of `ROTV` and `SWAP` instructions required for the other macros is also bounded by a constant. □

Lemma 3.3.2. *All operations listed in Table 3.1 increase a non-zero row width by at most a constant factor.*

Proof. The only operations that can increase the row width are `COPY`, `INC`, `SHIFT`, `CINC`, and `CSHIFT`. `COPY` increases the row width by at most a factor of 2; this happens when it is applied to a $(1, 0)$-rtable. The increment and shift instructions each increase the row width by at most one bit which amounts, for a non-zero row width, to at most a factor of 2. The macros in Table 3.1 execute a bounded number of these instructions by Lemma 3.3.1. □

Labelled rtables offer a way to express operations on rtables without regard to the order of tuple elements. For each of the macros specified in Table 3.1, the exact sequence of elementary Relational Machine operations depends on the length of the tuple and the order the columns are currently in. We usually do not care about the specifics, only that the number of instructions executed is bounded by a constant as guaranteed by Lemma 3.3.1.

Figure 3.1: Example of a sequence of operations denoted by A.CShift$(l, \mathit{left}, \mathit{right})$ under the assumption that the value tuple starts with columns in the order $(k, q, \mathit{left}, \mathit{right}, l)$.

instruction	new value tuple
	$(k, q, \mathit{left}, \mathit{right}, l)$
`ROTV` A	$(q, \mathit{left}, \mathit{right}, l, k)$
`ROTV` A	$(\mathit{left}, \mathit{right}, l, k, q)$
`ROTV` A	$(\mathit{right}, l, k, q, \mathit{left})$
`SWAP` A	$(l, \mathit{right}, k, q, \mathit{left})$
`ROTV` A	$(\mathit{right}, k, q, \mathit{left}, l)$
`ROTV` A	$(k, q, \mathit{left}, l, \mathit{right})$
`ROTV` A	$(q, \mathit{left}, l, \mathit{right}, k)$
`ROTV` A	$(\mathit{left}, l, \mathit{right}, k, q)$
`SWAP` A	$(l, \mathit{left}, \mathit{right}, k, q)$
`CSHIFT` A	$(l, \mathit{left}, \mathit{right}, k, q)$

See Figure 3.1 for an example of executing CShift$(l, \mathit{left}, \mathit{right})$ on a labelled rtable A. If we were to execute the same macro again immediately, it would not require any `ROTV` or `SWAP` instructions since the columns are already in the correct order. In this case, it would just expand to `CSHIFT` A.

The notation introduced in Table 3.1 is chosen to resemble method calls in an object-oriented programming language. We use it only for macros that modify an rtable in place by executing a constant number of operations that manipulate the value tuple only (except for Copy(), which also rearranges key tuple columns, but does not change their content).

Three more macros of this kind will be useful. The sequence of operations given as Algorithm 8 and denoted as A.Not(b) negates a Boolean value in value column b with the usual convention that a nonzero integer represents *true* and zero represents *false*. The sequence given as Algorithm 9 and denoted as A.CZ(c, v) conditionally sets a column v to zero if column c is nonzero.

Algorithm 8 A.Not(b): Negate a Boolean value in value column b

1: A.Push(temp)
2: A.Inc(temp)
3: A.CDec(b, temp)
4: A.Pop(b)
5: A.Rename$(b \leftarrow \mathit{temp})$

Algorithm 9 A.CZ(c, v): Set $v = 0$ if $c > 0$

1: A.Push(temp)
2: A.CSwap(c, v, temp)
3: A.Pop(temp)

By using the shift and increment instructions, an arbitrary constant can be created in a new value column using a constant number of instructions as shown in Algorithm 10. We

write this as A.Constant($name \leftarrow value$). In order to present such a sequence of instructions depending on one or more constant parameters, we introduce the following meta-programming constructs to describe how to generate it: a block surrounded by **expand if** is to be generated only under the given condition and otherwise ignored. An **unroll for** block indicates a sequence of copies of this block where a pseudo-variable takes the values specified. For added clarity, we will highlight meta-instructions, i.e. instructions that describe how to generate a fixed sequence of operations as opposed to instructions to be carried out at runtime, in blue.

Algorithm 10 A.Constant($name \leftarrow value$)

```
1:  A.Push(name)
2:  A.Push(temp)
3:  expand if value > 0
4:    unroll for i = ⌊log value⌋ downto 0
5:      A.Shift(temp, name)
6:      expand if bit i of value is set
7:        A.Inc(name)
8:      end expand if
9:    end unroll for
10: end expand if
11: A.Pop(temp)
```

When presenting algorithms for the Relational Machine, we will use the notation

procedure ProcedureName($A[structure_A], B[structure_B], \ldots$)
  ... operations on labelled rtables ...
  return $R[structure_R]$
end procedure

to describe the table structure of the input and output of the algorithm. $structure_A$ is of the form $((kname_1, \ldots, kname_s), (vname_1, \ldots, vname_t))$ and assigns the specified names to the columns of A; likewise for other rtables mentioned. This turns an rtable into a labelled rtable so that the notation in Table 3.1 can be used to describe the algorithm without explicitly keeping track of the order of the columns. The **return** statement describes the order in which to arrange the tuple elements before dropping the names, turning the result back into an (unlabelled) rtable. All rtable names not mentioned as input or output rtables are assumed to be in unique rtable registers not conflicting with anything else. For aesthetic reasons, we write $B \leftarrow A$ instead of `MOV` A, B.

3.4 Basic relational techniques

In this section we show how to perform some basic manipulations of rtables on a Relational Machine and present them as a series of short algorithms. These algorithms will serve as building blocks in the next section to demonstrate how to simulate a Database Machine on a Relational machine. Each algorithm is considered to be part of a larger Relational Machine program.

3.4.1 Time

For the remainder of this section, we assume that n is the input size of the whole machine. The term *time* refers to the number of Relational Machine instructions as defined in Definition 3.3.2. We say an algorithm uses *constant time* if the number of instructions it executes is bounded by a constant independent of n. The number of instructions may depend on other factors such as the tuple length of an rtable, but this is always bounded by a constant by definition of the Relational Machine. Likewise, when we say an algorithm takes polylogarithmic time or time $O(\log n)$, we are always referring to the overall input size n of the whole machine and not just the input size of the algorithm in question.

3.4.2 Frugality

The following definition will be useful to keep track of temporary resource usage.

Definition 3.4.1. An algorithm for the Relational Machine, taking one or more rtables as input and producing one or more rtables as output, is said to be *frugal* if it meets the following criteria:

- The number of registers used for temporary data in addition to input and output is bounded by a constant.
- The tuple length of all rtables involved in the computation is linear in the maximum tuple length among the input and output rtables.
- The row width of all rtables involved in the computation is linear in the maximum row width among the input and output rtables.
- The sum of the cardinalities of all rtables involved in the computation is linear in the sum of the cardinalities of all input and output rtables.

Note that the composition of two frugal algorithms is not necessarily frugal. For example, consider a (1,0)-rtable A and the one-line "algorithm" `JOIN` A, A, A. It creates a (2,0)-rtable containing the Cartesian product of A with itself. The result has $|A|^2$ rows, but the algorithm is still frugal since the space used is equal to the size of the output.

As a second algorithm, consider the following instruction sequence that takes a $(2, 0)$-rtable as input and produces a $(1, 0)$-rtable as output: `COPY` A; `RANGE` A. This is the projection on the first column. The result is at most as large as the input and no temporary rtables are used, so the algorithm is frugal. The composition of these two algorithms is the identity on the input rtable A. It is not frugal since both the input and output size are $|A|$ but the temporary space usage is $|A|^2$.

Since frugality is not compositional, it does not directly replace space complexity, but it will be an important tool to structure our proofs of space bounds on Relational Machine programs composed of smaller algorithms. Recall that when we claim that something can be computed by a Relational Machine using space $S(n)$, we also implicitly claim the existence of the constants r, l, c_1, and c_2 from Definition 3.3.1 that globally bound the number of registers, tuple lengths, and row width.

In the remainder of this chapter we will present constructions that combine several frugal algorithms. To show that the overall program uses space $O(S(n))$ we will establish that it adheres to this space bound after each step. Frugality then guarantees that this is also true for the temporary space used by each step. By the same argument, we will observe that intermediate results will stay within the bounds r, l, and c_2 at each step. Frugality then guarantees that there exist constants r', l', and c_2' such that the overall program runs within these bounds: r' needs to be larger than r by at most a constant and the bounds l' and c_2' need to be larger than their counterparts by at most a constant factor to accommodate the temporary requirements of the constituent algorithms.

3.4.3 Disjoint union

The union of two rtables can be formed if both have the same tuple lengths. It will be useful as a building block for future operations to combine the rows of two rtables into one without removing duplicates, using additional key columns to allow this. We call this the *disjoint union*. Algorithm 11 forms the disjoint union of two nonempty $(s, 0)$-rtables A and B and leaves the result in the value tuple of A. We denote this operation as

$$\text{DisjointUnion}(A, B).$$

Following this by $\text{Range}(A)$ yields the (set) union in the key tuples of A.

Algorithm 11 $\text{DisjointUnion}(A, B)$

procedure DISJOINTUNION($A[((a_1, \ldots, a_s), ())], B[((b_1, \ldots, b_s), ())]$)
  $\text{Join}(\texttt{TWO}, A, A)$ ▷ add a column c (`TWO` has structure $((c), ())$)
  $A.\text{Copy}(*)$
  unroll for $i = 1, \ldots, s$
    $A.\text{CZ}(c, a_i)$
  end unroll for
  $\text{Range}(A)$
  $\text{Join}(\texttt{TWO}, B, B)$
  $B.\text{Copy}(*)$
  unroll for $i = 1, \ldots, s$
    $B.\text{CZ}(c, b_i)$
  end unroll for
  $B.\text{Not}(c)$
  $\text{Range}(B)$
  $A.\text{Copy}(c)$
  $B.\text{Copy}(c)$
  $\text{Join}(A, B, A[c \leftarrow A.c, c' \leftarrow B.c])$
  $A.\text{Copy}(c, a_1, \ldots, a_s, b_1, \ldots, b_s)$ ▷ omitting c'
  unroll for $i = 1, \ldots, s$
    $A.\text{CSwap}(c, a_i, b_i)$
    $A.\text{Pop}(b_i)$
  end unroll for
  $A.\text{Pop}(c)$
  return $A[((c, a_1, \ldots, a_s, c', b_1, \ldots, b_s), (a_1, \ldots, a_s))]$
end procedure

Lemma 3.4.1. *Let s be an integer and A and B two distinct registers, each containing a nonempty $(s, 0)$-rtable. Then Algorithm 11 returns in the value tuples of A the disjoint union of the key tuples of the inputs A and B with duplicates differentiated by distinct key tuples. The key tuples have the form $(c, a_1, \ldots, a_n, c', b_1, \ldots, b_s)$ where $c = c' \in \{0, 1\}$. For $c = 0$ the a_i correspond to an original tuple from A and for $c = 1$ the b_i correspond to an original tuple from B. The algorithm is frugal and takes constant time.*

Proof. The idea for forming the disjoint union of two rtables is to use a join to bring the two together into a single rtable. In order to do that, we add an extra row to both rtables and join them in such a way that the extra row of A matches the original rows of B and vice versa.

For nonempty A, joining with `TWO` doubles all rows of A, adding an extra key tuple element $c \in \{0, 1\}$. We then copy the key tuple to the value tuple, yielding a value tuple $(c, a_1, \ldots, a_s)$. After conditionally setting all a_i to zero and applying Range(A), register A contains rows $(0, a_1, \ldots, a_s)$ for all original key tuples $(a_1, \ldots, a_s)$ and one extra row $(1, 0, \ldots, 0)$. The procedure is repeated for B with a negated c column, yielding a row $(1, b_1, \ldots, b_s)$ for each original key tuple $(b_1, \ldots, b_s)$ and one extra row $(0, 0, \ldots, 0)$.

After joining A and B on c, register A contains key tuples $(c, a_1, \ldots, a_n, c', b_1, \ldots, b_s)$ where the a_i correspond to an original tuple from A for $c = c' = 0$ and the b_i correspond to an original tuple from B for $c = c' = 1$, as claimed. By using conditional swap instructions, the algorithm moves all original tuples to the a_i and then deletes all remaining columns. The value tuples of A now contain the disjoint union of the original key tuples of A and B as claimed.

All macros used in the algorithm take constant time. Some are repeated s times, but s is bounded by the maximum tuple length of the machine and independent of the overall input size n. Therefore the algorithm takes constant time, as claimed.

All steps of the algorithm are frugal. No temporary registers are used and tuple lengths are at most $2s+2$. The largest row width occurs on line 18. If w is the maximum row width among the input rtables A and B, the row width after line 18 is at most $4w + 3$ ($2w + 2$ in the key tuple and $2w+1$ in the value tuple). The number of rows in A and B is temporarily doubled before it is reduced again and the result is produced. The cardinalities of all rtables involved are therefore linear in the cardinality of the input. Overall, this establishes that the algorithm is frugal, as claimed. □

3.4.4 Creating literals

We now show how to create literal rtables.

Lemma 3.4.2. *An rtable with arbitrary content in the key tuples can be created on a Relational Machine by a frugal sequence of operations taking constant time.*

Proof. By starting with the single-row rtable `ONE` and repeatedly adding constant columns, a single-row rtable with an arbitrary value tuple can be created. The `RANGE` operation moves the value tuple to the key tuple. To construct an rtable with multiple rows, we start by creating a single-row rtable in A. For each additional row, we create that row in B

and execute DisjointUnion(A, B); Range(A) to add it to A. Since this whole procedure involves a constant number of constant-time operations, the overall time is constant. All steps are frugal and all intermediate results are at most as large as the end result (in all four required dimensions: register count, tuple length, row width, and space), establishing that the overall sequence is frugal. □

3.4.5 Scalar variables and loops

In order to execute a loop for a non-constant number of iterations, we need a way to count iterations and stop at the desired count.

Definition 3.4.2. A *scalar variable* is a $(1, 0)$-rtable with a single row, storing a single integer. We denote scalar variables in algorithms with lower case letters. When used in the context of a labelled rtable, we assume that the single key column has the same name as the scalar variable. For a scalar variable s and a constant c we denote by $s \leftarrow s + c$ and $s \leftarrow s - c$ the sequence of Relational Machine instructions to increment or decrement s for c times.

Lemma 3.4.3. *A scalar variable can be tested for equality with another scalar variable or a constant on a Relational Machine by a frugal sequence of operations taking constant time.*

Proof. The idea is that joining two single-row rtables yields a non-empty result if and only if the values are equal. Let s and t two scalar variables to be tested for equality. The sequence $U \leftarrow s$; U.Copy(s); $V \leftarrow t$; Copy$(V, s \leftarrow t)$; Join(U, V, U) yields the empty rtable in U if the integer in s is not equal to t and a single-row rtable in case of equality. To test for equality with a constant c instead of another scalar, we construct the temporary rtable V by executing $V \leftarrow$ `ONE`; Constant$(V, s \leftarrow c)$. In both cases, a conditional jump instruction can be used to change control flow based on the result of the test.

The algorithm uses two temporary registers, does not increase tuple length, at most doubles the row width and uses two additional rows of space, making it frugal. By construction, it takes constant time. □

We will use common notation like **repeat** ... **until** *scalar* = *value* in pseudocode to indicate the use of this technique for changing control flow based on the value of a scalar variable.

3.4.6 Testing for all-zero columns

For the next tasks we need to be able to test if a subset of columns $c_1, \ldots, c_k$ of an rtable A contains zeros in all rows. We will denote this test as

$$\text{AllZero}(A, c_1, \ldots, c_k)$$

and allow ourselves to use it in pseudocode in control structures like **repeat**/**until** with the understanding that it can be implemented using the conditional jump instructions of

the Relational Machine. The statements **return** *true* and **return** *false* in Algorithm 12 indicate the appropriate use of conditional jump instructions, not the return of Boolean values.

Lemma 3.4.4. *Let A be a labelled rtable with zero-length value tuples and let $c_1, \ldots, c_k$ be a subset of the key columns. Then Algorithm 12 determines if A contains zeros in all rows of all specified columns. It is frugal and takes constant time.*

Proof. The condition is trivially true for an empty input rtable, which is tested first. For nonempty input, the algorithm first creates in U a copy of the input rtable A and eliminates all columns except the ones to be tested. After a `RANGE` operation, U contains a single row with all zeros if our condition is true; otherwise, it contains one or more rows of not all zeros. Note that U is a key-only rtable and thus does not contain duplicate rows. Using DisjointUnion(), the algorithm adds a single row of zeros to U and executes Singles(U). If U was the single-row zero rtable, the result will be the empty rtable, otherwise at least one nonzero row remains. The condition can then be tested by testing if U is empty.

Each line takes constant time and is frugal by Lemma 3.3.1 and Lemma 3.4.1, so the algorithm takes constant time. It uses two temporary registers, does not increase tuple length, at most doubles row width and uses $|A| + 1$ additional rows, making it frugal overall. □

Algorithm 12 AllZero($A, c_1, \ldots, c_k$): Test if A has only zeros in all columns specified

```
 1: procedure AllZero(A, c_1, ..., c_k)
 2:   if A is empty then
 3:     return true
 4:   else
 5:     U ← A
 6:     U.Copy(c_1, ..., c_k)
 7:     Range(U)
 8:     V ← ONE
 9:     V.Push(c_1, ..., c_k)
10:     Range(V)
11:     DisjointUnion(U, V)
12:     Singles(U)
13:     if U is empty then
14:       return true
15:     else
16:       return false
17:     end if
18:   end if
19: end procedure
```

3.4.7 Turing Machines

In the next section we will show how the Relational Machine can simulate the **map** operation of a Database Machine. Recall that this operation requires the notion of a database

function, which we only introduced informally in Section 2.2. In order to define it formally, we need to settle on a specific flavour of Turing Machine. Turing Machines hardly need an introduction. In his seminal work [95], Turing has shown that Turing Machines are universal, i.e. they can simulate the execution of any Turing Machine on any input string. The Church-Turing thesis informally states that any function that can be effectively computed, i.e. described as an algorithm, can be computed by a Turing Machine.

There are many possible definitions of a Turing Machine differing in details like the number of tapes, kinds of tapes, tape alphabet, and input alphabet. Most of them can simulate each other with polynomial overhead [8]. For the purposes of this thesis, we define a Turing Machine as follows.

Definition 3.4.3. A Turing Machine is a finite automaton equipped with a tape as memory. The tape consists of an infinite sequence of cells extending to the right of the starting cell. Each cell stores a symbol from a tape alphabet. At each step the machine examines the symbol under the head. Depending on the symbol read and the current state, it writes a new symbol to the current head position and moves the head left or right or leaves it in place. The tape is one-sided infinite. If the machine tries to move the head left from the leftmost cell, it will stay in place.

This is formally described by a tuple (Γ, Q, δ) consisting of

- A *tape alphabet* Γ that includes a designated blank symbol "$\sqcup$". We assume that $\Gamma = \{\mathtt{0}, \mathtt{1}, \sqcup\}$.
- A finite set Q of *states* the machine can be in. We assume that Q is a set of consecutive integers starting at 0.
- A *transition function* $\delta\colon Q \setminus \{0\} \times \Gamma \to Q \times \Gamma \times \{\mathrm{L}, \mathrm{S}, \mathrm{R}\}$ describing the rules to perform each step. Its input is the current state, along with the symbol under the head. The output comprises the new state, the symbol to be written to tape, and the movement of the head where the symbols L, S, and R stand for left, stay in place, and right, respectively.

Input and output are strings over the alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$. At the beginning of the computation the tape contains the input string followed by an infinite sequence of "$\sqcup$". The machine starts in state 1 with the head positioned on the first input symbol. The machine halts when it reaches state 0. The string at the beginning of the tape up to and excluding the first blank symbol is considered the output.

In order to define functions $f\colon \mathbb{N}_0 \to \mathbb{N}_0$, we will use the following encoding.

Definition 3.4.4. Let $x \in \mathbb{N}_0$ be a nonnegative integer. The *string encoding* of x is the binary representation of $(x + 1)$ with the most significant 1 removed. When used as an input for a Turing Machine, it is written to the tape in big-endian order, i.e. with the most significant bit in the leftmost tape cell.

For example, the integer zero is encoded as the empty string "" and one is encoded as the string "0". The integer 18 is encoded as "0011" (the binary representation of 19

is 10011_2). This slightly unusual encoding was chosen to establish a bijection between the nonnegative integers and the set of all strings over Σ. We could have worked with just writing the standard binary representation to the tape, but then leading zeros would allow for multiple string representations of the same integer.

Lemma 3.4.5. *The string encoding of nonnegative integers is a bijective map from* $\mathbb{N}_0$ *to* Σ^*.

Proof. The mapping is clearly injective since different integers have different binary representations. It is also surjective, since by taking any string $s \in \Sigma^*$ (including the empty string), prepending a one and interpreting it as the binary representation of an integer, we get a positive integer y. Then $y - 1$ is a nonnegative integer with string encoding s. □

Definition 3.4.5. A function $f\colon \mathbb{N}_0 \to \mathbb{N}_0$ is called a *database function* if there exists a a Turing Machine $\mathcal{M}$ such that

1. $\mathcal{M}$ uses time polynomial and space linear in the length of its input string;

2. when $\mathcal{M}$ halts, the head is on the leftmost tape cell and there are no nonblank symbols on the tape after the first blank;

3. f can be computed by running $\mathcal{M}$ on the string encoding of the input value and applying the inverse of string encoding to the resulting string.

To extend this definition to higher-arity functions, we pack multiple integers into a single integer by interleaving their bits.

Definition 3.4.6. Let $x_1, \ldots, x_s \in \mathbb{N}_0$ be integers and let $x_i = \sum_j x_{i,j} 2^j$ be their binary representation with $x_{i,j} \in \{0, 1\}$. Then the result of *zipping* them is the integer $y = \sum_k y_k 2^k$ where $y_k = x_{i(k),j(k)}$ with $i(k) = (k \bmod s) + 1$ and $j(k) = \lfloor k/s \rfloor$. We also call the integers $x_1, \ldots, x_s$ the result of *unzipping* y into s integers.

Definition 3.4.7. Let $k, l \in \mathbb{N}$. A function $\tilde{f}\colon \mathbb{N}_0^k \to \mathbb{N}_0^l$ is called a *database function* if there exists a database function $f\colon \mathbb{N}_0 \to \mathbb{N}_0$ and $\tilde{f}$ can be computed by performing the following steps:

1. Zip the input k-tuple into a single integer x.

2. Compute $y = f(x)$.

3. Unzip y into an l-tuple.

We call f the *underlying* database function.

Note that this definition agrees with Definition 3.4.5 for $k = l = 1$.

3.4.8 Universal mapping

In this section we show how to compute an arbitrary database function on a Relational Machine.

Theorem 3.4.6. *Let $f\colon \mathbb{N}_0 \to \mathbb{N}_0$ be a database function. Let A be a labelled $(s, 0)$-rtable with key tuples $(k_1, \ldots, k_{s-1}, x)$ such that the sub-tuples $(k_1, \ldots, k_{s-1})$ are unique within the rtable. Then the result of mapping f over column x, i.e. replacing x by $f(x)$ in each row of rtable A, can be computed by a frugal algorithm on a Relational Machine using time polylogarithmic in the input size of the machine.*

Proof. We demonstrate how to simulate running a copy of the Turing Machine computing the function in each row of the input rtable. The following is an outline of the construction:

- Encode the input column x as a tape (Algorithm 13).
- Create a literal rtable describing the transition function.
- Repeat executing a Turing Machine step until all machines have halted (Algorithm 14).
- Decode the output on the tape as an integer (Algorithm 15).

Let $\mathcal{M} = (\Gamma, Q, \delta)$ be the Turing Machine that computes the function f. In order to simulate a one-sided infinite tape, we augment the tape alphabet by a start symbol "$\triangleright$" and simulate a Turing Machine $\mathcal{M}' = (\Gamma', Q, \delta')$ with $\Gamma' = \{\sqcup, \mathtt{0}, \mathtt{1}, \triangleright\}$ and

$$\delta'(q, \gamma) = \begin{cases} (q, \triangleright, \mathrm{R}) & \gamma = \triangleright \\ \delta(q, \gamma) & \text{otherwise.} \end{cases}$$

We run the simulation on a two-sided infinite tape that has a start symbol in the cell to the left of the starting position of the head. The augmented transition function δ' simply moves the head one cell to the right whenever it encounters the start symbol, thereby preserving it and simulating a machine that cannot move its head off the left end of the tape.

The tape is encoded as a pair of integers (*left*, *right*) where *right* contains the symbols under and to the right of the head and *left* contains the symbols to the left of the head. Each symbol is encoded as a 2-bit integer according to Table 3.2 with the least significant bits containing the symbols closest to the head. The encoding has the order of the two bits reversed between *left* and *right* so that the head can simply be moved one cell by executing two shift instructions of the relational machine. Note that the blank symbol "$\sqcup$" is encoded as zero so that *right* can be interpreted as an infinite tape with all blank symbols to the right of any nonblank symbols.

Algorithm 13 turns x into a Turing Machine configuration with the string encoding of x (see Definition 3.4.4) on the tape and the machine in its initial state 1. Inside the loop, it extracts the least significant bit from x and computes its complement (recall from Table 3.2 that the 2-bit encoding of the symbols "`0`" and "`1`" consists of the corresponding binary digit and its complement). The conditional shift instructions on line 10 then shift

Table 3.2: Encoding of Turing Machine symbols as 2-bit integers.

symbol	encoding *left*	encoding *right*
␣	0 (00_2)	0 (00_2)
0	1 (01_2)	2 (10_2)
1	2 (10_2)	1 (01_2)
▷	3 (11_2)	3 (11_2)

these two bits into *right*, but only if $x \neq 0$. This stops the shifting once all but the most significant 1 of the input value has been shifted onto the tape and allows a single loop manipulating all rows in parallel to shift an individual number of bits onto the tape in each row. The loop ends when all values have been fully encoded. The algorithm finishes by putting the encoded start symbol into *left* and the initial Turing Machine state 1 into q.

Algorithm 13 TMEncode(A): Prepare simulation of Turing Machine

```
1:  procedure TMEncode(A[((k_1, ..., k_{s-1}, x), ())])
2:      A.Copy(∗)
3:      A.Push(right)
4:      A.Inc(x)
5:      Range(A)
6:      repeat
7:          A.Copy(∗)
8:          A.Push(bit); A.Shift(x, bit)                          ▷ extract next bit
9:          A.Constant(¬bit ← 1); A.CDec(bit, ¬bit)               ▷ compute inverse
10:         A.CShift(x, ¬bit, right); A.CShift(x, bit, right)     ▷ write encoded bit if x ≠ 0
11:         A.Pop(bit, ¬bit)
12:         Range(A)
13:     until AllZero(A, x)
14:     A.Copy(k_1, ..., k_{s-1}, right)
15:     A.Constant(q ← 1)                                         ▷ initial state for TM is 1
16:     A.Constant(left ← 3)                  ▷ write encoded start symbol "▷" left of head
17:     Range(A)
18:     return A[((k_1, ..., k_{s-1}, q, left, right), ())]
19: end procedure
```

Recall that if a Relational Machine is run with an input of size n, the machine's row width can be at most $O(\log n)$. This bounds the length of the string encoding of x. Therefore the number of iterations of the loop in Algorithm 13 and hence the overall time is $O(\log n)$. The algorithm executes a sequence of frugal operations, uses no additional registers, does not increase tuple length beyond the output tuple length and the size of A stays constant throughout. Since each bit of x is encoded as two bits, the row width stays linear in the input row width. Overall, this establishes that Algorithm 13 is frugal.

The transition function is encoded as an rtable T as follows: For each $q \in Q \setminus \{0\}$ and $\gamma \in \Gamma'$ we generate a row with key tuple (in, cmd) where in encodes the input of the transition function and cmd encodes the output. Let c be the integer encoding of γ according to the *left* column of Table 3.2, let $(q', \gamma', d) = \delta'(q, \gamma)$ and let c' be the integer

encoding of γ', again according to the *left* column of Table 3.2. Then

$$in = 4q + c$$
$$cmd = 16q' + c' + \begin{cases} 0 & d = \mathrm{S} \\ 4 & d = \mathrm{L} \\ 8 & d = \mathrm{R} \end{cases}$$

For state 0 and each input symbol, we add a no-op row that writes the same symbol back and leaves the head in the same position. This ensures that executing additional steps from the halt state will not change the configuration of the machine. It is important because we are going to simulate an rtable of Turing Machines running in parallel, and they may each run for a different number of steps. By Lemma 3.4.2, the literal rtable T can be generated in constant time using a frugal algorithm. We then execute T.Copy(in) to prepare T for use in Algorithm 14.

Algorithm 14 now simulates running the Turing Machines. This involves a loop that terminates once all machines have reached their halting state $q = 0$. Each iteration simulates executing one step. First, the encoded input to the transition function is computed by setting in to the state q and then shifting two bits from *right* into in. A is then joined with rtable T to look up the value of the transition function. Since T encodes a function, the number of rows in A stays the same. The encoded result of the transition function, cmd, contains the new state augmented by four bits and is renamed to q. After removing the symbol under the head from *right*, the two low order bits are shifted into *right* to write the new symbol. The next bit encodes whether to move the head left, and conditional shift instructions (line 13) perform that movement. Line 16 conditionally moves the head right depending on the next command bit. The new state remains in q.

Algorithm 14 consists of frugal operations, each taking constant time. Thus, each iteration of the loop takes constant time. The number of iterations is the maximum number of steps any of the simulated Turing Machines executes. Since f is assumed to be a database function, the Turing Machines take polynomial time in their input size, which in turn is $\mathrm{O}(\log n)$ in relation to n, the input size of the Relational Machine. Hence, the time is polylogarithmic in n, as claimed.

The number of rows in A stays constant throughout the algorithm. No temporary registers are used, and the tuple length is only increased by a constant. Since f is assumed to be a database function, we know that the Turing Machine use spaces linear in its input. The representation of the Turing Machine tape in the integers *left* and *right* uses twice that number of bits plus two bits for the start symbol. Hence, the row width stays linear in the input row width, establishing that Algorithm 14 is frugal overall.

After all Turing Machines have halted, the output is in *right* and it remains to decode the output as shown in Algorithm 15. This is a straightforward reversal of the encoding process. Initialise $x = 1$, the leading one dropped during encoding. Successively shift bits from *right* to x if *right* $\neq 0$ until all *right* $= 0$. Finally, decrement x and return the output

Algorithm 14 TMRun(A, T): Run a Turing Machine simulation

procedure TMRun($A[((k_1, \ldots, k_{s-1}, q, \mathit{left}, \mathit{right}), ())], T[((\mathit{in}, \mathit{cmd}), (\mathit{in}))]$)
  repeat
    A.Copy($\mathit{in} \leftarrow q, \mathit{right}$)
    A.Shift($\mathit{right}, \mathit{in}$); A.Shift($\mathit{right}, \mathit{in}$) ▷ read symbol under head
    A.Pop(right)
    Join(A, T, A) ▷ look up transition function
    A.Copy($k_1, \ldots, k_{s-1}, q \leftarrow \mathit{cmd}, \mathit{left}, \mathit{right}$) ▷ omitting in
    A.Push(temp)
    A.Shift($\mathit{right}, \mathit{temp}$); A.Shift($\mathit{right}, \mathit{temp}$) ▷ erase symbol under head
    A.Pop(temp)
    A.Shift(q, right); A.Shift(q, right) ▷ write new symbol
    A.Push(l); A.Shift(q, l);
    A.CShift($l, \mathit{left}, \mathit{right}$); A.CShift($l, \mathit{left}, \mathit{right}$) ▷ conditionally move head left
    A.Pop(l)
    A.Push(r); A.Shift(q, r)
    A.CShift($r, \mathit{right}, \mathit{left}$); A.CShift($r, \mathit{right}, \mathit{left}$) ▷ conditionally move head right
    A.Pop(r)
    Range(A)
  until AllZero(A, q) ▷ until all machines have halted
  return $A[((k_1, \ldots, k_{s-1}, q, \mathit{left}, \mathit{right}), ())]$
end procedure

Algorithm 15 TMDecode(A): Decode Turing Machine Tape as an integer

procedure TMDecode($A[((k_1, \ldots, k_{s-1}, q, \mathit{left}, \mathit{right}), ())]$)
  A.Copy($k_1, \ldots, k_{s-1}, \mathit{right}$)
  A.Constant($x \leftarrow 1$) ▷ leading 1 was dropped from x in encoding
  Range(A)
  repeat
    A.Copy($*$)
    A.CShift($\mathit{right}, \mathit{right}, x$) ▷ extract bit to x
    A.Push(temp)
    A.CShift($\mathit{right}, \mathit{right}, \mathit{temp}$) ▷ dispose of $\overline{\mathit{bit}}$
    A.Pop(temp)
    Range(A)
  until AllZero(A, right)
  A.Copy($k_1, \ldots, k_s, x$)
  A.Dec(x) ▷ undo increment from encoding
  return $A[((k_1, \ldots, k_{s-1}, x), ())]$
end procedure

in the desired format. By the same reasoning as for the encoding process, this is frugal and takes time $O(\log n)$.

Overall, this proves the theorem's claims on time and space. □

This shows how to compute a unary function $f\colon \mathbb{N}_0 \to \mathbb{N}_0$ by simulating a Turing Machine. To extend this to higher-arity functions, we pack multiple integers into a single integer by interleaving their bits.

Lemma 3.4.7. *Let A be a labelled rtable with empty value tuple and let $x_1, \ldots, x_s$ be a subset of the key columns. Then Algorithm 16 replaces columns $x_1, \ldots, x_s$ by a column x containing in each row the result of zipping the integers in these columns (see Definition 3.4.6). Algorithm 17 performs the inverse unzipping operation. Both operations are bijections, leaving the number of rows in A unchanged. Both algorithms are frugal and take time $O(\log n)$.*

Algorithm 16 ZipColumns$(A, x_1, \ldots, x_s, x)$: Replace key columns $x_1, \ldots, x_s$ by a new key column x that contains the result of zipping the integers in $x_1, \ldots, x_s$

```
1:  procedure ZipColumns(A, x_1, ..., x_s, x)
2:      A.Copy(*)
3:      A.Constant(temp ← 1)                    ▷ marker for reversing bit order
4:      A.Push(x, b)
5:      Range(A)
6:      while not AllZero(A, x_1, ..., x_s) do
7:          A.Copy(*)
8:          unroll for i = 1, ..., s
9:              A.Shift(x_i, temp)
10:         end unroll for
11:         Range(A)
12:     end while
13:     while not AllZero(A, temp) do
14:         A.Copy(*)
15:         A.Shift(temp, b)                    ▷ now temp = 0 iff b is the marker bit
16:         A.CShift(temp, b, x)
17:         unroll for i = 2, ..., s
18:             A.CShift(temp, temp, x)
19:         end unroll for
20:         Range(A)
21:     end while
22:     A.Copy(*)
23:     A.Pop(x_1, ..., x_s, temp, b)
24:     Range(A)
25: end procedure
```

Proof. Each iteration of the loop at line 6 of Algorithm 16 shifts one bit from each x_i into *temp*, interleaving them as desired. Note that the **while** loop terminates only after all nonzero bits from all rows have been shifted, ensuring that the same number of shifts are executed for all rows and columns. After the loop, *temp* contains the interleaved bits in

Algorithm 17 UnzipColumns$(A, x, x_1, \ldots, x_s)$: Replace the key column x by key columns $x_1, \ldots, x_s$ containing the result of unzipping the integer in x. This operation is the inverse of ZipColumns$(A, x_1, \ldots, x_s, x)$.

```
1:  procedure UnzipColumns(A, x, x_1, ..., x_s)
2:      A.Copy(*)
3:      A.Constant(temp ← 1)                    ▷ marker for reversing bit order
4:      A.Push(x_1, ..., x_s, b)
5:      Range(A)
6:      while not AllZero(A, x) do
7:          A.Copy(*)
8:          unroll for i = 1, ..., s
9:              A.Shift(x, temp)
10:         end unroll for
11:         Range(A)
12:     end while
13:     while not AllZero(A, temp) do
14:         A.Copy(*)
15:         A.Shift(temp, b)                    ▷ now temp = 0 iff b is the marker bit
16:         A.CShift(temp, b, x_s)
17:         unroll for i = s − 1, s − 2, ..., 1
18:             A.CShift(temp, temp, x_i)
19:         end unroll for
20:         Range(A)
21:     end while
22:     A.Copy(*)
23:     A.Pop(x, temp, b)
24:     Range(A)
25: end procedure
```

reverse order with the most significant bit of x_s in its least significant bit. The next **while** loop reverses the bit order by shifting all bits from *temp* into the result column x.

In order to know the correct number of iterations for this loop, *temp* was initialised with 1 as a marker. Line 15 first extracts one bit from *temp* into b. At this point we have $temp = 0$ if and only if b was the marker bit. The following conditional shift instructions shift exactly s bits into x, but only if this was not the marker, thereby removing the marker from the result. The result has all the bits from the x_i in the correct order and the input and temporary columns are removed to obtain the result.

The algorithm executes only frugal constant-time operations. Since the row width is $\mathrm{O}(\log n)$, the number of iterations and hence the number of operations in each loop is $\mathrm{O}(\log n)$. The number of rows in A stays constant throughout the algorithm. No temporary registers are used and tuple lengths are temporarily only increased by a constant. The row width increases by at most a constant factor with the worst case being a factor of s when zipping a nonzero column x_s with the remaining columns of width zero. Since the row width of the result is linear in the input row width and bits are shifted to the result one by one, the row width stays linear in the input row width throughout, making the algorithm frugal overall.

Algorithm 17 undoes the operation of Algorithm 16 by first pushing a marker bit and reversing the bits in the input column. In the second loop, note that after executing line 15, we have $temp = 0$ if and only if the marker bit has been shifted into b. If this is not the case, all of the following s conditional shift instructions will be executed since the marker is s or more bits away – each iteration of the second loop either shifts one bit into each of the x_i or none of them. On the last iteration of the loop, the conditional shift instructions do not shift. By the same arguments as for Algorithm 16, Algorithm 17 is frugal and takes time $\mathrm{O}(\log n)$ as claimed.

Since the two operations are inverses of each other, they are both bijective and preserve the number of rows in A, regardless of the presence of any other columns. □

Theorem 3.4.8. *Let $f\colon \mathbb{N}_0^l \to \mathbb{N}_0^m$ be a database function. Let A be a labelled $(s,0)$-rtable with key tuples $(k_1,\ldots,k_{s-l},x_1,\ldots,x_l)$ such that the sub-tuples $(k_1,\ldots,k_{s-l})$ are unique within the rtable. Then the result of mapping f over the columns x_i, i.e. replacing the sub-tuple $(x_1,\ldots,x_l)$ by the m-tuple $f(x_1,\ldots,x_l)$, can be computed by a frugal algorithm on a Relational Machine using time polylogarithmic in the input size of the machine.*

Proof. Let n be the input size of the Relational Machine. We first combine the input columns into one column using $\mathrm{ZipColumns}(A,x_1,\ldots,x_l,x)$. By Lemma 3.4.7, this is frugal and takes $\mathrm{O}(\log n)$ operations. The function is then computed as a function of a single integer. By Theorem 3.4.6, this is frugal and takes time polylogarithmic in n. Next, we split the result into the desired columns using $\mathrm{UnzipColumns}(A,x,x_1,\ldots,x_l)$. By Lemma 3.4.7, this is frugal and takes $\mathrm{O}(\log n)$ operations.

The three steps do not use additional registers and keep the number of rows in A constant. Tuple length does not increase. Row width increases by at most a factor of l by the zipping operation. Since f is a database function, it stays linear during the function

application. The final unzipping operation cannot further increase the row width. This establishes that the algorithm is frugal. □

3.4.9 Filter for zeros

For the following, we require another helper operation: selecting all those rows of an rtable that have a zero in a specified column.

Lemma 3.4.9. *Given a labelled $(s, 0)$-rtable A with a column c, Algorithm 18 removes from A all rows that have a nonzero value in column c. It is frugal and takes constant time.*

Proof. In a helper register U, the algorithm creates a single-row $(0, 1)$ rtable with a zero in the value tuple. Joining this with the input A on column c keeps only the desired rows in A, not adding any columns to the key tuple since the key tuple of U is empty. The number of rows in A cannot increase since each row in A matches at most the single row in U. So the algorithm is frugal and takes constant time as claimed. □

Algorithm 18 FilterZero(A, c): Keep only rows with a zero in column c

```
1: procedure FilterZero(A, c)
2:     U ← ONE
3:     U.Push(c)
4:     A.Copy(c)
5:     Join(A, U, A)
6: end procedure
```

3.4.10 Computing an aggregate

One feature of the Database Machine is the computation of an aggregate. This involves grouping the rows of a table by some criterion and reducing all rows in the same group using a binary operation. The operation needs to be commutative and associative in order for the result to be well defined. We now show how to compute an aggregate on the Relational Machine. Since an aggregate combines values from multiple rows – potentially all rows of an rtable – care must be taken to ensure that the row width of the result stays bounded as required for a Relational Machine. The following theorem offers two ways to ensure this.

Theorem 3.4.10. *Let A be a labelled $(3, 0)$-rtable with key tuples (k, g, v) such that the values of the pair (k, g) are unique within the rtable. Let $\oplus\colon \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ be a commutative and associative binary operation that is a database function. Algorithm 19 computes a $(2, 0)$-rtable of pairs $(g, \tilde{v})$ containing a row for each distinct value of g in A such that*

$$\tilde{v} = \bigoplus_{\substack{(k,g',v)\in A:\\ g'=g}} v$$

on a Relational Machine. It is frugal and uses time polylogarithmic in the machine's input size, provided that at least one of the following conditions holds:

1. *For each value in column g, the number of rows with that value is bounded by a constant d, independent of n.*

2. *The binary operation is width-bounded (see Definition 2.1.4 on page 19).*

Proof. The idea of Algorithm 19 is to bring pairs of rows with the same value in column g together into a single row using a join and applying the binary operation to reduce. This is repeated until there is only one row left for each distinct value in g.

Algorithm 19 RunAggregate$_\oplus(A)$: Group rows and reduce values using a binary operation

```
 1: procedure RunAggregate⊕(A[((k, g, v), ())])
 2:     repeat
 3:         A.Copy(∗)
 4:         A.Push(b)
 5:         A.Shift(k, b)
 6:         Range(A)
 7:         C ← A
 8:         C.Copy(k, g)
 9:         Singles(C)
10:         C.Copy(k, g, v)
11:         Range(C)
12:         B ← A
13:         FilterZero(A, b)
14:         B.Copy(b)
15:         B.Not(b)
16:         Range(B)
17:         FilterZero(B, b)
18:         A.Copy(k, g)
19:         B.Copy(k, g)
20:         Join(A, B, A[k′ ← B.k, g′ ← B.g, v′ ← B.v, b′ ← B.b])
21:         A.Copy(k, g, v, v′)
22:         Range(A)
23:         replace v by v ⊕ v′ by simulating a Turing Machine
24:         DisjointUnion(A, C)
25:         Range(A)
26:     until AllZero(A, k)
27:     A.Copy(g, v)                                   ▷ remove k
28:     Range(A)
29:     return A[((g, v), ())]
30: end procedure
```

To see the correctness of the algorithm, observe that it maintains the invariant that the pair (k, g) is unique within the rtable. On each iteration, k is divided by 2. When all $k = 0$, the loop terminates with unique g as claimed.

If we let $k' = \lfloor k/2 \rfloor$, then each pair (k', g) is either unique or occurs exactly twice, namely for $k = 2k'$ and $k = 2k' + 1$. After dividing k by 2 and shifting the least significant

bit into the newly created column b, the algorithm makes a copy of A in C and isolates those rows for which (k, g) is unique. Starting with another copy of A in B, it filters A to contain only rows with $b = 0$ and B to contain only rows with $b = 1$. The join on line 20 now brings together pairs of rows with the same (k, g). Unmatched rows in either A or B are dropped by this operation, but appear in C so that all values of the original rtable A are used exactly once. The binary operation on the two values v and v' is then computed by simulating a Turing Machine as discussed in Section 3.4.8. The result is reunited with the set-aside single values in C using DisjointUnion(A, C). This concludes a single iteration in which some pairs of values are combined using the binary operation and some values are preserved.

If the input size of the Relational Machine is n, the row width and hence the number of bits in the k column is $\mathrm{O}(\log n)$ so that the number of iterations until all k values are reduced to zero is also $\mathrm{O}(\log n)$. This implies that for each value in column g, $\tilde{v}$ is the result of applying the operation $\oplus$ in a (usually not complete) binary tree of depth $\mathrm{O}(\log n)$. We need to show that the row width of the rtable remains $\mathrm{O}(\log n)$ throughout the computation.

If condition 1 of the theorem holds, there is a bound d on the number of rows in each group. This bounds the depth of the calculation. Since the binary operation $\oplus$ is a database function, the row width can increase by at most a constant factor f on each application so that the overall increase is at most by a constant factor of f^{d-1}, ensuring that the row width remains $\mathrm{O}(\log n)$ throughout the computation.

If condition 2 of the theorem holds, a tree of applications of the operation $\oplus$ of depth $\mathrm{O}(\log n)$ adds at most $c \cdot \mathrm{O}(\log n)$ to the row width of the rtable so that the resulting row width is still $\mathrm{O}(\log n)$.

In any case, by Theorem 3.4.8 the Turing Machine simulation inside the loop is frugal and takes time polylogarithmic in n. Since the number of iterations is $\mathrm{O}(\log n)$, the total time is polylogarithmic in n.

All operations in the algorithm are frugal and it uses only two temporary registers, B and C. Tuple lengths and row widths are at most doubled. During each iteration, the total number of rows it at most tripled by copying A to C and B. At the end of the iteration, A cannot have more rows than before. Overall, this establishes that the algorithm is frugal as claimed. □

3.5 Simulating the Database Machine

We are now ready to show how to simulate a Database Machine on a Relational Machine. Recall that the Database Machine works with tables which are multisets whereas the Relational Machine uses rtables which are sets. Since the input of a Database Machine is a multiset, it cannot be directly used as the input to a Relational Machine. We therefore need to define a suitable input format.

Definition 3.5.1. A *keyed multiset* is an $(s + 1, 0)$-rtable with key tuples $(k, a_1, \ldots, a_s)$. A keyed multiset A is said to represent a multiset $\tilde{A}$ if $\tilde{A}$ is the result of removing the

column k. Slightly abusing the notation from Section 2.4.4:

$$\tilde{A} = [(a_1, \ldots, a_s) \mid ((k, a_1, \ldots, a_s), ()) \Subset A]$$

Let w be the row width of A, $\tilde{w}$ the row width of $\tilde{A}$ and $n = |A| = |\tilde{A}|$. The keyed multiset A is called *slim* if $n = 0$ or $w \leq \tilde{w} + \lceil \log n \rceil$.

Note that although the column k is named to suggest it functions as a key, the definition does not require it to be unique. Its role is merely to make the overall tuple unique and thus allow multiple instances of the sub-tuple $(a_1, \ldots, a_s)$ in the key tuples of an rtable.

Lemma 3.5.1. *Let $\tilde{A} \in [\![\mathbb{N}_0^s]\!]$ be a multiset with cardinality $n > 0$ and row width w. Then there is a slim keyed multiset A with cardinality n representing $\tilde{A}$.*

Proof. Adding a column containing the numbers $0, \ldots, n-1$ in any order turns $\tilde{A}$ into a keyed multiset A representing it. The additional column needs $\lceil \log n \rceil$ bits, keeping A slim, as claimed. □

In order to simulate a Database Machine on a Relational Machine, we will also use a more convenient representation of multisets.

Definition 3.5.2. A *set with multiplicities* is an $(s+1, 0)$-rtable that has key tuples $(m, a_1, \ldots, a_s)$ such that the sub-tuples $(a_1, \ldots, a_s)$ are unique and all m are positive. A set with multiplicities A is said to represent a multiset $\tilde{A} \in [\![\mathbb{N}_0^s]\!]$ if for all $((m, a_1, \ldots, a_s), ()) \in A$ we have $\tilde{A}(a_1, \ldots, a_s) = m$ and $\tilde{A}(a_1, \ldots, a_s) = 0$ for all other tuples $(a_1, \ldots, a_s)$, recalling from Section 2.1 that $\tilde{A}(a_1, \ldots, a_s)$ denotes the multiplicity of the element $(a_1, \ldots, a_s)$ in the multiset $\tilde{A}$.

Lemma 3.5.2. *Let $\tilde{A} \in [\![\mathbb{N}_0^s]\!]$ be a multiset with cardinality $n > 0$ and row width w and let A be the set with multiplicities representing it. Then $|A| \leq n$ and the row width of A is at most $w + \lceil \log(n+1) \rceil$.*

Proof. A has an extra column m containing the multiplicity of each element of $\tilde{A}$. Its width is determined by the largest multiplicity, which is at most n in the case that the multiset $\tilde{A}$ contains a single element with multiplicity n. Storing the number n requires $\lceil \log(n+1) \rceil$ bits. $|A|$ is the number of distinct elements of $\tilde{A}$, so it can be at most n. □

Lemma 3.5.3. *Algorithm 20 turns a keyed multiset A into a set with multiplicities representing the same multiset on a Relational Machine. It is frugal and uses time polylogarithmic in the machine's input size.*

Proof. The algorithm first duplicates the payload columns $a_1, \ldots, a_s$ and adds a column m for the multiplicity, initialising it to 1. Next, it zips columns $k, a_1, \ldots, a_s$ into a unique key column k' which takes logarithmic time by Lemma 3.4.7. It zips the copies of the payload columns $a'_1, \ldots, a'_s$ into a single column a, again taking logarithmic time.

In the next step, equal payload values are combined and their multiplicities added by RunAggregate$_+$(). By Theorem 3.4.10, this is frugal and takes polylogarithmic time

Algorithm 20 CountMultiplicities(A): Turn keyed multiset into set with multiplicities

```
 1: procedure CountMultiplicities(A[((k, a_1, ..., a_s), ())])
 2:     A.Copy(*)
 3:     A.Copy(a'_1 ← a_1, ..., a'_s ← a_s)
 4:     A.Constant(m ← 1)
 5:     Range(A)
 6:     ZipColumns(A, k, a_1, ..., a_s, k')
 7:     ZipColumns(A, a'_1, ..., a'_s, a)
 8:     RunAggregate_+(A[k', a, m])
 9:     UnzipColumns(A, a, a_1, ..., a_s)
10:     return A[((m, a_1, ..., a_s), ())]
11: end procedure
```

(noting that addition is a width-bounded database function). The payload is then unzipped into the original tuples which, by Lemma 3.4.7, is frugal and takes logarithmic time. The tuple length is at most doubled during the process. The row width stays linear at each step and the number of rows in A stays constant, so the algorithm is frugal as claimed. □

We will now show how each of the basic operations of a Database Machine defined in Section 2.2 can be simulated on a Relational Machine, representing multisets as sets with multiplicities. We will omit the simulation of the **ljoin** operation since it has already been shown to be redundant in Theorem 2.5.2.

Lemma 3.5.4 (**union**). *Let A and B be sets with multiplicities representing multisets $\tilde{A}, \tilde{B} \in [\![\mathbb{N}_0^s]\!]$. Algorithm 21 computes a set with multiplicities representing the multiset union $\tilde{A} \uplus \tilde{B}$ on a Relational Machine. It is frugal and takes time polylogarithmic in the machine's input size.*

Proof. After copying A to A', the algorithm handles the edge cases of one of the sets being empty. It then forms the disjoint union with B which is a frugal operation taking constant time by Lemma 3.4.1. Next, equal tuples $(a_1, \ldots, a_s)$ from the two sets need to be combined and their multiplicities added. To do this, the algorithm zips copies of columns from the key tuple into a unique key column k and zips the payload tuple $(a_1, \ldots, a_s)$ into a grouping column a.

RunAggregate$_+$() is then used to add the multiplicities. Since integer addition is a width-bounded database function, this is frugal and takes polylogarithmic time by Theorem 3.4.10. The payload tuple is then unzipped to obtain the result. By Lemma 3.4.7, each zipping and unzipping operation is frugal and takes logarithmic time, so the overall time is polylogarithmic as claimed.

The algorithm consists only of frugal operations and uses one additional register. The initial tuple length of $s + 1$ is increased to at most $3s + 2$ (line 11, value tuple) and the row width also stays linear in the input row width after each step. The number of rows in A' is at most the sum of the input sizes. Overall, this establishes that the algorithm is frugal. □

Algorithm 21 MultisetUnion(A, B): Form union of two sets with multiplicities

procedure MULTISETUNION($A[((m_a, a_1, \ldots, a_s), ())], B[((m_b, b_1, \ldots, b_s), ())]$)
  $A' \leftarrow A$
  if emtpy(A) **then**
    $A' \leftarrow B$
    return A'
  else if empty(B) **then**
    return A'
  else
    DisjointUnion(A', B)
      ▷ A' has structure $((c, m_a, a_1, \ldots, a_s, c', m_b, b_1, \ldots, b_k), (m_a, a_1, \ldots, a_s))$
    A'.Copy$(c, a'_1 \leftarrow a_1, \ldots, a'_s \leftarrow a_s, b'_1 \leftarrow b_1, \ldots, b'_s \leftarrow b_s)$
    Range(A')
    ZipColumns$(A', c, a'_1, \ldots, a'_s, b'_1, \ldots, b'_s, k)$
    ZipColumns$(A', a_1, \ldots, a_s, a)$
    RunAggregate$_+(A'[k, a, m_a])$
    UnzipColumns$(A', a, a_1, \ldots, a_s)$
    return $A'[((m_a, a_1, \ldots, a_s), ())]$
  end if
end procedure

Lemma 3.5.5 (**select**). *Let A be a set with multiplicities representing a multiset $\tilde{A} \in [\![\mathbb{N}_0^k]\!]$ and $p\colon \mathbb{N}_0^k \to \{0, 1\}$ a database function. Algorithm 22 computes a set with multiplicities A' representing the multiset* $\mathbf{select}_p(\tilde{A})$ *on a Relational Machine. It is frugal and takes time polylogarithmic in the machine's input size.*

Proof. After duplicating the payload tuple $(a_1, \ldots, a_s)$, the algorithm computes the predicate by simulating a Turing Machine, which is frugal and takes polylogarithmic time by Theorem 3.4.8.

Algorithm 22 MultisetSelect$_p(A)$: Compute $\mathbf{select}_p$ on a set with multiplicities

procedure MULTISETSELECT$_p(A[((m, a_1, \ldots, a_s), ())])$
  $A' \leftarrow A$
  A'.Copy$(*)$
  A'.Copy$(a'_1 \leftarrow a_1, \ldots, a'_s \leftarrow a_s)$
  Range(A')
  simulate Turing Machine on A' to replace $a'_1, \ldots, a'_s$ by $c = p(a'_1, \ldots, a'_s)$
  A'.Copy(c)
  $T \leftarrow$ `ONE`
  T.Constant$(c \leftarrow 1)$
  Join(A', T, A')
  A'.Copy$(m, a_1, \ldots, a_s)$
  Range(A')
  return $A'[((m, a_1, \ldots, a_s), ())]$
end procedure

A join with a single-row rtable containing the value 1 is then used to keep only those rows for which the predicate has the value 1. Removing column c yields the desired result.

All steps of the algorithm are frugal. It uses two additional registers and at most doubles tuple length and row width. The number of rows in A' starts out as the input size and can only get smaller by the filtering operation. Overall, the algorithm is frugal, as claimed. □

Lemma 3.5.6 (**map**). *Let A be a set with multiplicities representing a multiset $\tilde{A} \in [\![\mathbb{N}_0^s]\!]$ and $f\colon \mathbb{N}_0^s \to \mathbb{N}_0^l$ a database function. Algorithm 23 computes a set with multiplicities A' representing the multiset $\mathbf{map}_f(\tilde{A})$ on a Relational Machine. It is frugal and takes time polylogarithmic in the machine's input size.*

Proof. The function $\tilde{f}$ does not have to be injective. According to the definition of **map** on page 20, the multiplicity of a value $y = f(x)$ is the sum of the multiplicities of all pre-images of y. Algorithm 23 computes this by first zipping the payload tuple and creating a copy of it to keep as a unique key column k. It then simulates a Turing Machine to compute the underlying database function of f according to Definition 3.4.7.

By Theorem 3.4.6, this is frugal and takes polylogarithmic time; note that the unique keys in column k guarantee that all rows are preserved.

Algorithm 23 MultisetMap$_f(A)$: Compute $\mathbf{map}_f$ on a set with multiplicities

```
1: procedure MultisetMap_f(A[((m, a_1, ..., a_s), ())])
2:     A' ← A
3:     ZipColumns(A', a_1, ..., a_s, a)
4:     A'.Copy(∗)
5:     A'.Copy(k ← a)
6:     Range(A')
7:     simulate Turing Machine in A' compute the underlying database function of f
8:     RunAggregate_+(A'[k, a, m])
9:     UnzipColumns(A', a, a_1, ..., a_l)
10:    return A'[((m, a_1, ..., a_l), ())]
11: end procedure
```

RunAggregate_+ is then used to group the results of the function application and add the multiplicities. Since integer addition is a width-bounded database function, this is frugal and takes polylogarithmic time by Theorem 3.4.10. It remains to unzip the result into an l-tuple. By Lemma 3.4.7, all zipping and unzipping operations are frugal and take logarithmic time, so the overall time is polylogarithmic as claimed.

To see that the algorithm is frugal, note that one additional register is used and tuple lengths are bounded as required. The row width increases by zipping and duplicating the zipped value, but stays linear in the input row width as required. After running the aggregate, it is linear in the output row width. The number of rows can only decrease by computing the aggregate. □

Lemma 3.5.7 (**join**). *For nonnegative integers k, l, m with $m \leq k, l$, let A and B be sets with multiplicities representing multisets $\tilde{A} \in [\![\mathbb{N}_0^k]\!]$ and $\tilde{B} \in [\![\mathbb{N}_0^l]\!]$, respectively. Algorithm 24 computes a set with multiplicities J representing the multiset $\mathbf{join}_m(\tilde{A}, \tilde{B})$ on a Relational Machine. It is frugal and takes time polylogarithmic in the machines's input size.*

Proof. The algorithm joins the two input rtables on the first m columns of their payload tuples. It then simulates a Turing Machine to multiply the multiplicities. Since multiplication is a database function, this is frugal and takes polylogarithmic time by Theorem 3.4.8. It remains to reformat the rtable to remove unnecessary columns.

Algorithm 24 MultisetJoin$_m(A, B)$: Join two sets with multiplicities

```
1: procedure MultisetJoin_m(A[((m_a, a_1, ..., a_k), ())], B[((m_b, b_1, ..., b_l), ())])
2:     A' ← A
3:     B' ← B
4:     A.Copy(a_1, ..., a_m)
5:     B.Copy(b_1, ..., b_m)
6:     Join(A, B, J)
7:     simulate Turing Machine on J to replace m_a and m_b by m_j = m_a · m_b
8:     J.Copy(m_j, a_1, ..., a_k, b_{m+1}, ..., b_l)
9:     Range(J)
10:    return J[((m_j, a_1, ..., a_k, b_{m+1}, ..., b_l), ())]
11: end procedure
```

To see that the algorithm is frugal, note that it uses three additional registers. Tuple length is doubled by the join operation. Row width is linear in the input row width until line 7 and linear in the output row width after the multiplication is performed. □

In order to simulate the operation **group** of a Database Machine on a set with multiplicities, we need a helper algorithm that reduces each element according to its multiplicity using a binary operation.

Definition 3.5.3. Let $\oplus\colon D \to D$ be an associative binary operation, $m \in \mathbb{N}$ and $v \in D$. The result of reducing m copies of v using $\oplus$ is written as

$$m \odot v := \underbrace{v \oplus v \oplus \cdots \oplus v}_{m \text{ copies of } v}$$

Lemma 3.5.8. *Let $\oplus\colon D \to D$ be an associative binary operation, $m \in \mathbb{N}$ and $v \in D$. Let $m = \sum_k m_k 2^k$ with $m_k \in \{0, 1\}$ for all $k \in \mathbb{N}_0$ be the binary representation of m and let $\{k_1, \ldots, k_w\} = \{k \mid m_k = 1\}$ be the set of indices with binary ones. Then*

$$m \odot v = \bigoplus_{i=1}^{w} \left(2^{k_i} \odot v\right)$$

.

Proof. By associativity, we have $(a + b) \odot v = (a \odot v) \oplus (b \odot v)$ for any $a, b \in \mathbb{N}$. Applying this to the sum $m = \sum_{i=1}^{w} 2^{k_i}$, we get the claimed identity. Note that we do not require $\oplus$ to have an identity element since the sum is never empty ($m > 0$). The order of the k_i does not matter since $(a \odot v) \oplus (b \odot v) = (b \odot v) \oplus (a \odot v)$ for all $a, b \in N$. □

Lemma 3.5.9. *Let A be a set with multiplicities where m is the multiplicity column and v is another column. Let $\oplus\colon \mathbb{N}_0 \to \mathbb{N}_0$ be an associative binary operation that is a width-bounded*

database function. Algorithm 25 reduces the values v according to their multiplicity, producing an rtable where in each row columns m and v are replaced by a single column v containing the value $m \odot v$. All other columns are preserved. The algorithm is frugal and takes time polylogarithmic in the Relational Machine's input size.

Proof. The algorithm uses the decomposition of m into powers of 2 from Lemma 3.5.9 to successively build up the result in the accumulator *acc*. Denote by v_0 the initial value in column v. At the beginning of the i-th iteration of the loop, we have $v = 2^{i-1} \odot v_0$. This can be seen by noting that v is unchanged within the loop, except on the last line where it is replaced by $v \oplus v$. In iteration i of the loop, bit $(i-1)$ is extracted from m and *acc* is replaced by $acc \oplus v$ if the bit was 1 and unchanged otherwise.

Algorithm 25 ReduceMultiples$_\oplus(A, m, v)$: Reduce according to multiplicity using binary operation $\oplus$

```
 1: procedure ReduceMultiples⊕(A, m, v)
 2:     A.Copy(∗)
 3:     A.Copy(acc ← v)
 4:     A.Constant(empty ← 1)
 5:     Range(A)
 6:     while not AllZero(A, m) do
 7:         A.Copy(∗)
 8:         A.Push(b)
 9:         A.Shift(m, b)
10:         A.Copy(v′ ← v)
11:         Range(A)
12:         simulate a Turing Machine in A to compute additional column acc′ ← acc ⊕ v
13:         A.Copy(∗)
14:         A.CSwap(b, acc, acc′)
15:         A.CSwap(empty, acc, v′)
16:         A.CDec(b, empty)
17:         A.Pop(b, acc′, v′)
18:         Range(A)
19:         simulate a Turing Machine in A to compute v ← v ⊕ v
20:     end while
21:     A.Copy(∗)
22:     A.Pop(m)
23:     A.Rename(v ← acc)
24:     Range(A)
25:     return A
26: end procedure
```

Since the operation $\oplus$ does not necessarily have an identity element, we cannot simply initialise *acc* with that identity element. Instead, we use a flag *empty*, initialised with 1, to indicate that the accumulator is empty.

After provisionally computing $acc' \leftarrow acc \oplus v$, a conditional swap instruction swaps *acc* with acc' if $b = 1$. However, if the accumulator is still supposed to be empty, as indicated by $empty = 1$, line 15 exchanges it with a copy of v that was set aside in the variable v' earlier. This is the correct value to assign to *acc* the first time it becomes nonempty. The accumulator becomes nonempty once something is combined with it, which happens

if $b = 1$. Line 16 reflects this fact; note that decrementing zero yields zero on the Relational Machine so that subsequent iterations do not change *empty*.

The overall effect is that the first time we have $b = 1$ on iteration i of the loop, *acc* gets initialised with $v = 2^{i-1} \odot v_0$ and on subsequent iterations with $b = 1$ it gets replaced by $acc \oplus (2^{i-1} \odot v_0)$ which, by Lemma 3.5.8, computes the desired result. Note that since the input is a set with multiplicities, we always have $m > 0$ so that the binary representation has at least one nonzero bit and the accumulator is always initialised.

Since the row width of an rtable is logarithmic in the Relational Machine's input size n, the number of iterations is $\mathrm{O}(\log n)$. The binary operation $\oplus$ is a database function, so by Theorem 3.4.8 the two Turing Machine simulations in the loop take polylogarithmic time, making the overall time polylogarithmic in n.

All steps of the algorithm are frugal. Tuple lengths are increased by at most 4. Since $\oplus$ is a width-bounded database function, the width of column v increases by at most a constant on each iteration and we have $\mathrm{width}(acc) \leq \mathrm{width}(v)$ at the end of each iteration so that the overall row width stays at $\mathrm{O}(\log n)$. The number of rows in A stays the same if columns with unique values are present and can otherwise only decrease. Overall, the algorithm is frugal as claimed. □

Lemma 3.5.10 (group). *Let A be a set with multiplicities representing a multiset $\tilde{A} \in [\![\mathbb{N}_0^s]\!]$ and $l \leq s$. Let $\oplus \colon \mathbb{N}_0^{s-l} \times \mathbb{N}_0^{s-l} \to \mathbb{N}_0^{s-l}$ be a commutative and associative binary operation that is a width-bounded database function. Algorithm 26 computes a set with multiplicities G representing the multiset $\mathbf{group}_{l,\oplus}(\tilde{A})$ on a Relational Machine. It is frugal and takes time polylogarithmic in the machine's input size.*

Proof. The algorithm first duplicates the payload tuple and zips it into a unique key column. It then zips the first l ("grouping") columns into a column g and the remaining ("value") columns into a column v. The set with multiplicities is then reduced to a set according to the multiplicities, preserving the unique key column k. This is frugal and takes polylogarithmic time by Lemma 3.5.9. $\mathrm{RunAggregate}_{\oplus}(G)$ is then used to compute the aggregate, a frugal algorithm taking polylogarithmic time by Theorem 3.4.10.

The result of this computation is a set which is then unzipped into the original column structure and augmented with a multiplicity column of 1 to turn it back into a set with multiplicities. All zipping and unzipping operations are frugal and take logarithmic time by Lemma 3.4.7 so that the overall time is polylogarithmic, as claimed.

The algorithm uses one additional register and the tuple length is at most doubled. Zipping columns only increases row width linearly. The number of rows can only decrease when running the aggregate. Overall, this establishes that the algorithm is frugal. □

We are now ready to prove the main theorem of this section, namely that a Relational Machine can simulate a Database Machine with a slowdown polylogarithmic in the input size.

Recall that we have introduced two ways to represent a Database Machine table as an rtable in a Relational Machine: the keyed multiset (Definition 3.5.1) and the set with multiplicities (Definition 3.5.2). All constructions in this section use sets with multiplicities

Algorithm 26 MultisetGroup$_{l,\oplus}(A)$: Compute **group**$_{l,\oplus}$ on a set with multiplicities

procedure MULTISETGROUP$_{l,\oplus}(A[((m, a_1, \ldots, a_s), ())])$
  $G \leftarrow A$
  G.Copy$(*, a'_1 \leftarrow a_1, \ldots, a'_s \leftarrow a_s)$
  Range(G)
  ZipColumns$(G, a'_1, \ldots, a'_s, k)$
  ZipColumns$(G, a_1, \ldots, a_l, g)$
  ZipColumns$(G, a_{l+1}, \ldots, a_s, v)$
  ReduceMultiples$_{\oplus}(G, m, v)$
  RunAggregate$_{\oplus}(G[k, g, v])$
  UnzipColumns$(G, g, a_1, \ldots, a_l)$
  UnzipColumns$(G, v, a_{l+1}, \ldots, a_s)$
  G.Copy$(*)$
  G.Constant$(m \leftarrow 1)$
  Range(G)
  return $G[((m, a_1, \ldots, a_s), ())]$
end procedure

to simulate the basic operations of a Database Machine on a Relational Machine. For the input and output format, however, we choose keyed multisets.

The reason for this is that both kinds of machines by definition have a bound on their row widths that is logarithmic in their input size. The input size of the Database Machine is the cardinality of the input multiset, defined as the sum of all multiplicities. If we used sets with multiplicities as the input representation for the simulation on a Relational Machine, the input size of the Relational Machine could be much smaller than the input size of the simulated Database Machine, making the simulation impossible because of the logarithmic row width bound. The extreme case is an input multiset with a single input element x of multiplicity m, which would be represented as a set with multiplicities containing the single row $((m, x), ())$. This is an input size of 1 in the Relational Machine model, independent of the multiplicity m.

The representation of a multiset as a keyed multiset does not have this problem. It assumes that the rows of the input table are made unique by adding an additional column, ensuring that the input size on the simulating Relational Machine is the same as the input size of the simulated Database Machine. Lemma 3.5.3 shows how to convert a keyed multiset into a set with multiplicities on a Relational Machine and Theorem 2.5.7 provides the opposite direction on a Database Machine, which, as we will prove, can be simulated on a Relational Machine to convert the output back to a keyed multiset.

Theorem 3.5.11. *Let $\mathcal{D}$ be a Database Machine that does not crash on any valid input and, for an input of size n, runs in time $T(n)$ and uses space $S(n)$ where $S(n)$ is polynomial in n. Then there exists a Relational Machine $\mathcal{R}$ that when given an input $\tilde{I} \in [\![\mathbb{N}_0^s]\!]$ of $\mathcal{D}$, represented as a slim keyed multiset I, produces the same output $\tilde{O}$ as $\mathcal{D}$, represented as a slim keyed multiset O. $\mathcal{R}$ takes time $\mathrm{O}(T(n) \cdot \log^t n)$ for some $t \in \mathbb{N}$ and uses space $\mathrm{O}(S(n))$.*

Proof. Let $\tilde{r}, \tilde{l}, \tilde{c_1}, \tilde{c_2}$ be the bounds on the number of registers, tuple length, input row width, and overall row width of the Database Machine $\mathcal{D}$, respectively (see Definition 2.3.1). We will show that there exists a Relational Machine $\mathcal{R}$ with corresponding bounds r, l, c_1, c_2 that simulates $\mathcal{D}$ within the claimed time and space.

The Relational Machine $\mathcal{R}$ will convert the input keyed multiset to a set with multiplicities, simulate each operation of the Database Machine $\mathcal{D}$ on this representation and finally convert the result back to a slim keyed multiset. During the simulation, $\mathcal{R}$ will use a bounded number of extra rtables and also temporarily increase tuple length, space usage and row width. These increases will be suitably bounded to ensure the existence of the bounds r, l, c_1, c_2 governing the whole execution of the Relational Machine.

Let $\tilde{I} \in [\![\mathbb{N}_0^s]\!]$ be a valid input of $\mathcal{D}$ of size n and I a slim keyed multiset representing $\tilde{I}$. $\mathcal{R}$ receives I as an input of size n and executes Algorithm 20, converting it to a set of multiplicities representing $\tilde{I}$. By Lemma 3.5.3, this is frugal and takes time polylogarithmic in n.

Now the simulation of $\mathcal{D}$ commences on the representation of the input as a set with multiplicities. Each operation of the Database Machine is translated to a sequence of operations on the Relational Machine and takes time polylogarithmic in n by Lemma 3.5.4 (**union**), Lemma 3.5.5 (**select**), Lemma 3.5.6 (**map**), Lemma 3.5.7 (**join**), and finally Lemma 3.5.10 (**group**). The operation **ljoin** can be expressed in terms of a constant number of other operations of the Database Machine by Theorem 2.5.2, hence taking a total time polylogarithmic in n. The jump and conditional jump instructions of the Database Machine directly correspond to operations of the Relational Machine.

When the simulated Database Machine halts, the output of the simulating Relational Machine needs to be converted back to a slim keyed multiset. Let $\tilde{O}$ be the output of the simulated Database Machine. By the assumption of the Theorem, we have $|\tilde{O}| = \mathrm{O}(n^k)$ for some $k \in \mathbb{N}$. We now simulate running Algorithm 4 to convert $\tilde{O}$ to a set on the Database Machine. By Theorem 2.5.7, this takes $\mathrm{O}(\log\log|O|) = \mathrm{O}(\log\log n)$ Database Machine operations, each of which takes time polylogarithmic in n on the simulating Relational Machine. So overall, this step takes polylogarithmic time. Let O be the set with multiplicities representing the converted output. Since the converted output is a set, every element has multiplicity 1. Removing the multiplicity column from O turns O into a keyed multiset representing the output of the Database Machine. It is slim because Algorithm 4 adds, for an element with multiplicity m, the numbers $0, \ldots, m-1$ in the key column, which increases the row width by at most $\lceil \log m \rceil \leq \lceil \log n \rceil$.

Algorithm 4 uses four temporary tables and the tuple length is increased by at most 4. By Lemma 2.5.8 the row width is bounded to be linear in the output row width, making the algorithm frugal overall.

The above establishes the claim that $\mathcal{R}$ takes time $T(n) \cdot \mathrm{O}(\log^t n)$ for some $t \in \mathbb{N}$. It remains to show the claim on space usage and the existence of the bounds r, l, c_1, and c_2.

If an input of $\mathcal{D}$ conforms to its input row width bound $\tilde{c_1}$, there is a corresponding bound c_1 for $\mathcal{R}$ because the input is represented as a slim keyed multiset. Since we assumed $\mathcal{D}$ not to crash on any valid input, it is known that the state of the simulation conforms

to the bounds $\tilde{r}, \tilde{l}, \tilde{c_2}$ at each step. At each step, the representation in the simulating machine $\mathcal{R}$ will use the same number of registers and the tuples of the simulating rtables will be one longer than the simulated tuples, namely by the multiplicity column.

All constructions involved in the simulation are frugal by the Lemmas cited above. Thus, the bound r is obtained from $\tilde{r}$ by increasing it by the maximum amount of temporary registers used. Likewise, the bound l is obtained from $\tilde{l}$ by taking into account the extra column for the multiplicity plus the maximum number of temporary columns used.

For the existence of the bound c_2, it suffices to show that the row width of the simulating rtables is $\mathrm{O}(\log n)$ throughout. Every simulated table $\tilde{A}$ has size at most $S(n) = \mathrm{O}(n^k)$ and row width $w = \mathrm{O}(\log n)$. By Lemma 3.5.2, the size of the simulating rtable A is at most $S(n)$ and the row width is at most $w + \lceil \log(S(n) + 1) \rceil$ which is $\mathrm{O}(\log n)$. Both size and row width are temporarily increased by at most a constant factor during the simulation since all constructions are frugal, establishing the existence of the bound c_2 and the claim on the space usage of the simulation. □

Corollary 3.5.12. *A Big Data-practical Database Machine can be simulated by a Big Data-practical Relational Machine. A Small Data-practical Database Machine can be simulated by a Small Data-practical Relational Machine.*

Proof. Both notions of practicality assume a polylogarithmic runtime. Since the simulation has a polylogarithmic slowdown by Theorem 3.5.11, the runtime stays polylogarithmic. Space usage increases linearly by the simulation, preserving the $\mathrm{O}(n)$ space usage of a Big Data-practical machine and the polynomial space usage of a Small Data-practical machine. □

Chapter 4

Parallel Random Access Machines

A model of parallel computation that has been widely studied since the late 1970s, starting with, e.g., [35, 42], is the Parallel Random Access Machine. It describes parallel computation in terms of Random Access Machines running synchronously and having access to each other's or a central shared memory. There are many flavours of PRAM models, differing in the capabilities of the individual RAMs, the structure of shared vs. local memory, possible limitations on connectivity, how processors are started, and more. Many of these are polynomially related [71].

4.1 The Parallel Random Access Machine

Our definition of a PRAM closely resembles the definition of a "network" in [71] in that each processor has its own local registers and the ability to access another processor's registers. It differs in how the input and output are formatted.

Definition 4.1.1. Let $P\colon \mathbb{N}_0 \to \mathbb{N}_0$ be a database function (see Definition 3.4.5). A *Parallel Random Access Machine (PRAM)* with processor bound P has $P(n)$ processors, depending on the input size n. Each processor has an unbounded number of registers $r_0, r_1, \ldots$, each capable of holding a nonnegative integer, and a read-only register "*PID*" containing a processor identifier. The machine has a program that is a finite sequence of instructions from Table 4.1, indexed by nonnegative integers. The binary operation $\oplus$ in Table 4.1 can be any database function according to Definition 3.4.7.

A computation on a PRAM maps an input tuple of n nonnegative integers $x_0, \ldots, x_{n-1}$ to an output tuple of m nonnegative integers $y_0, \ldots, y_{m-1}$. It proceeds as follows. The input size n is placed in r_0 of processor 0. x_k is placed in r_i of processor j where $i = 2 \cdot \lfloor k/P(n) \rfloor + 1$ and $j = k \bmod P(n)$. All other registers are initialised with zero. $P(n)$ processors with IDs $0, \ldots, P(n)-1$ are activated and start executing at instruction 0 of the program. All processors share the same program, but have individual instruction pointers so that they can take different execution paths. Instructions are executed synchronously in parallel by all active processors, proceeding to the next instruction unless a conditional jump changes control flow or the **halt** instruction is executed. At each step, all read operations occur in parallel, followed, if applicable, by local computation, followed by the

Table 4.1: PRAM instruction set

Instruction	Description
$r_i \leftarrow c$	load register with constant c
$r_i \leftarrow r_j \oplus r_k$	locally compute a binary operation $\oplus$
$r_i \leftarrow r_{r_j}$	indirect load from local register
$r_{r_i} \leftarrow r_j$	indirect store into local register
$r_i \leftarrow PID$	load register with ID of this processor
$r_i \leftarrow (r_{r_j}$ **of** $r_k)$	indirect load from processor r_k's register r_{r_j}
$(r_{r_i}$ **of** $r_j) \leftarrow r_k$	indirect store into processor r_j's register r_{r_i}
goto l **if** $r_i > 0$	conditional jump to location l
halt	halt execution of this processor

write operation. If multiple processors write to the same register, the processor with the lowest ID succeeds and the other writes are ignored. Reading a register of a nonexisting processor returns zero; writes to nonexisting processors are ignored. Execution ends when all processors have halted. At this time, r_0 of processor 0 contains the size of the output, m, and output y_k is found in r_i of processor j where $i = 2 \cdot \lfloor k/P(n) \rfloor + 1$ and $j = k \bmod P(n)$ for $k = 0, \ldots, m-1$.

This definition places the input and output in odd-numbered registers only and ensures that the even-numbered registers, except for r_0 of processor 0, are free to be used without regards to the input size n, which may be larger than the processor bound $P(n)$.

Definition 4.1.2. Let $T, S, W : \mathbb{N}_0 \to \mathbb{N}_0$. A PRAM is said to execute within *time* $T(n)$ if for all inputs of size n all processors halt within $T(n)$ steps. It is said to use *space* $S(n)$ if the number of distinct registers accessed during any computation on an input of size n is at most $S(n)$. All input values and all output values are always counted as being accessed, as well as the *PID* register of each processor. The machine has *word size* $W(n)$ if all register contents are less than $2^{W(n)}$ throughout any computation with an input of n integers that are each less than $2^{c \log(n+2)}$ for some constant c, where T, S, and W may depend on c. The processor ID registers are included in the word size bound.

This definition implies that the word size bound also bounds the number of processors. We will consider PRAMs with $W(n) = \mathrm{O}(\log n)$. Since the *PID* register of any processor must be less than $2^{W(n)}$, the number of processors on such a machine is polynomial in n. Since the *PID* registers are included in the space bound, we always have $S(n) \geq P(n)$. The input and output values are included in the space bound in order to avoid degenerate cases like a machine mapping zero-tuples to zero-tuples without using any space.

Our definition differs from Parberry's in that it counts any register ever accessed during a computation wheres Parberry considers the maximum number of registers with nonzero contents at any given time during the computation. Parberry's definition would allow the use of large sparse hash tables in which only the entries actually used would count towards the space usage. This assumption is unrealistic on an actual machine where memory would have to be allocated for such a hash table, no matter which portion of it is actually accessed.

The main result of this chapter is that under certain conditions a Database Machine can simulate a PRAM and vice versa. To be able to formally state these as a theorems, we need to specify what it means for the Database Machine to perform the same computation as a PRAM. A PRAM takes as an input a tuple of nonnegative integers and produces a tuple of nonnegative integers as an output. A Database Machine takes as input a table and produces a table as an output.

Definition 4.1.3. Let $\mathcal{V}$ be a machine that takes as an input a tuple of positive integers and produces a tuple of positive integers as an output, such as a PRAM. Let $x_0, \ldots, x_{n-1}$ be an input of $\mathcal{V}$ of size n and $y_0, \ldots, y_{m-1}$ the corresponding output. A Database Machine $\mathcal{D}$ is said to *compute the same result* as $\mathcal{V}$ if it takes as input a table containing a pair (i, x_i) for $i = 0, \ldots, n-1$ and produces as output a table containing pairs (j, y_j) for $j = 0, \ldots, m-1$.

Theorem 4.1.1. *Let $\mathcal{P}$ be a Parallel Random Access Machine with processor bound $P(n)$, space bound $S(n)$ and word size $W(n) = \mathrm{O}(\log n)$ that takes time $T(n)$. Then there is a Database Machine $\mathcal{D}$ that computes the same result. $\mathcal{D}$ takes time $\mathrm{O}(T(n) + \log\log n)$ and space $\mathrm{O}(S(n))$.*

To prove this theorem, we first define an auxiliary model.

4.2 The Parallel Microcode Machine

In this section we define a model that is similar to the PRAM and more suitable to be simulated by a Database Machine. It is called a Parallel Microcode Machine and consists of Microcode Processors accessing a shared memory.

Definition 4.2.1. Let fetch: $\mathbb{N}_0^4 \to \mathbb{N}_0$, compute: $\mathbb{N}_0^5 \to \mathbb{N}_0^4$ and P: $\mathbb{N}_0 \to \mathbb{N}_0$ be database functions. A *Parallel Microcode Machine* with microcode (fetch, compute) and processor bound P has $P(n)$ Microcode Processors, depending on the input size n. Each *Microcode Processor* has four registers pc, acc, i, m, each holding a nonnegative integer, and a read-only register *pid* holding a nonnegative integer. All Microcode Processors have access to an unbounded number of shared memory cells $M_0, M_1, \ldots$, each holding a nonnegative integer. A processor is said to be *halted* if $pc = 0$ and *active* otherwise.

Executing a step on an active Microcode Processor involves optionally reading a memory cell, performing a computation and optionally writing a memory cell. The location to be read is computed as $i = \mathrm{fetch}(pid, P(n), pc, acc)$ where $i \neq 0$ indicates reading from memory cell M_{i-1} and $i = 0$ indicates using the value 0 instead; the value read is placed in the register m. The function call $\mathrm{compute}(pid, P(n), pc, acc, m)$ returns a 4-tuple (pc, acc, i, m) with new register values. Finally, if $i \neq 0$ the value m is written to memory cell M_{i-1}. This sequence is shown as Algorithm 27.

A computation on the Parallel Microcode Machine maps an input tuple of n nonnegative integers $x_0, \ldots, x_{n-1}$ to an output tuple of m nonnegative integers $y_0, \ldots, y_{m-1}$. It proceeds as follows. The input size n is placed in memory cell M_0. Input value x_i is placed in memory cell M_{2i+1}. All other memory cells initially contain zero. A number of $P(n)$ Microcode Processors are activated with processor identifiers $pid = 0, \ldots, P(n) - 1$, $pc = 1$ and zero

Algorithm 27 Executing one step on a Microcode Processor

```
1: if pc > 0 then
2:    i ← fetch(pid, P(n), pc, acc)
3:    if i > 0 then
4:       m ← M_{i−1}
5:    else
6:       m ← 0
7:    end if
8:    (pc, acc, i, m) ← compute(pid, P(n), pc, acc, m)
9:    if i > 0 then
10:      M_{i−1} ← m
11:   end if
12: end if
```

in the other registers. All processors execute steps synchronously. If multiple processors simultaneously write to the same memory cell, the processor with the lowest *pid* succeeds and the other writes are ignored. The machine stops when all processors have halted. At this time, M_0 contains the output size m and output value y_i is found in M_{2i+1} for $i = 0, \ldots, m-1$.

The idea of this definition is to encapsulate the whole program of a PRAM in two functions instead of viewing it as a list of instructions. At each step, a Microcode Processor reads a memory cell into a register m, performs some computation and optionally writes a memory cell. The function fetch() decides which memory cell to read and the function compute() encapsulates computation, possible change of control flow and the decision what to write to memory. Both functions take as input the read-only processor identifier *pid*, the processor bound $P(n)$ and the registers *pc* and *acc*; compute() in addition takes m, the value read from memory, as a fifth argument. The register *pc* represents the state of the computation and acts as a program counter telling the functions which computation to perform at the current step. The register *acc* acts as an accumulator to carry intermediate results from one step of the computation to the next. Register i holds the index of a memory location to read or write, offset by 1 and using the value zero to indicate that no read or write is to occur.

This definition places the input and output in odd-numbered memory cells and uses M_0 to explicitly pass the input and output size. This simplifies programming the machine since the even-numbered memory cells except M_0 are free for use regardless of the input size.

Definition 4.2.2. Let $T, S, W : \mathbb{N}_0 \to \mathbb{N}_0$. A Parallel Microcode Machine with processor bound $P(n)$ is said to execute within *time* $T(n)$ if for all inputs of size n all processors halt within $T(n)$ steps. It is said to use *space* $S(n)$ if the number of distinct shared memory cells accessed during any computation on an input of size n is at most $S(n) - P(n)$. All input values and all output values are always counted as being accessed. The machine has *word size* $W(n)$ if the contents of all registers and shared memory cells are less than $2^{W(n)}$ throughout any computation with an input of n integers that are each less than $2^{c \log(n+2)}$

Figure 4.1: Memory map for simulating a PRAM on a Parallel Microcode Machine. The top table illustrates how the input values x_i are placed in PRAM registers; n is the input size, $p = P(n)$ the processor bound and the s_i are scratch registers that are not part of the input. The bottom table shows how these values are arranged in the shared memory of a Parallel Microcode Machine.

PID	r_0	r_1	r_2	r_3	r_4	r_5	$\cdots$
0	n	x_0	s_p	x_p	s_{2p}	x_{2p}	$\cdots$
1	s_1	x_1	s_{p+1}	x_{p+1}	s_{2p+1}	x_{2p+1}	$\cdots$
2	s_2	x_2	s_{p+2}	x_{p+2}	s_{2p+2}	x_{2p+2}	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
$p-1$	s_{p-1}	x_{p-1}	s_{2p-1}	x_{2p-1}	s_{3p-1}	x_{3p-1}	$\cdots$

k	0	1	2	3	4	5	6	7	$\cdots$
M_k	n	x_0	s_1	x_1	s_2	x_2	s_3	x_3	$\cdots$

for some constant c, where T, S, and W may depend on c.. The processor ID registers are included in the word size bound.

Note that the processor bound $P(n)$ is included in the space bound $S(n)$ to make this definition compatible with Definition 4.1.2 by taking into account the local registers of the Microcode Processors. Since we are interested in asymptotic bounds only, we omit the factor of 5 for the number of processor registers. We can now simulate a PRAM using a Parallel Microcode Machine.

Lemma 4.2.1. *Let $\mathcal{P}$ be a Parallel Random Access Machine with processor bound $P(n)$ using space $S(n)$, time $T(n)$ and word size $W(n)$. Then there is a Parallel Microcode Machine $\mathcal{M}$ with processor bound $P(n)$ that simulates $\mathcal{P}$ within space $S(n)$, time $\mathrm{O}(T(n))$ and word size $\mathrm{O}(W(n))$.*

Proof. The two machines have the same processor bounds. We will construct machine $\mathcal{M}$ such that it simulates each instruction of $\mathcal{P}$ by executing three steps. A processor of $\mathcal{P}$ executing an instruction at location l is simulated by $\mathcal{M}$ executing three steps with register $pc = 3l + s$ where $s = 1, 2, 3$.

Throughout the simulation, a one-to-one correspondence between the registers of the simulated PRAM and the memory cells of the simulating Parallel Microcode Machine is maintained as follows: r_i of processor j of $\mathcal{P}$ is stored in memory cell M_k of $\mathcal{M}$ where

$$k = 2 \cdot (\lfloor i/2 \rfloor P(n) + j) + (i \bmod 2)$$

This correspondence is illustrated in Figure 4.1. Note that it holds for the input and output of the computation by definition and is maintained throughout the simulation.

Recall from Definition 4.1.1 that reading a register from a nonexisting processor on a PRAM returns zero and writes to registers of nonexisting processors are ignored. To

simulate this behaviour, we define the function

$$\text{mmap}_p(j,i) = \begin{cases} 2 \cdot (\lfloor i/2 \rfloor p + j) + (i \bmod 2) + 1 & \text{if } j < p \\ 0 & \text{otherwise.} \end{cases}$$

It represents the memory map of the simulation, offset by one and using the value zero to handle reads and writes to nonexisting processors; the processor bound $P(n)$ is passed as a parameter p for this purpose.

We now show how to construct the functions fetch() and compute() of $\mathcal{M}$ from the program of $\mathcal{P}$ and that they are database functions. Let l_{max} be the location of the last instruction of $\mathcal{P}$. Both functions have the structure shown as Algorithm 28. They test for all possible values of pc and return the values specified in Tables 4.2 and 4.3. To see that both functions are database functions, i.e. they can be computed by a Turing Machine in linear space and polynomial time relative to their input size, observe that this is true for all individual entries in Tables 4.2 and 4.3. The only operations used are

- constants,
- copy an argument,
- increment,
- the function mmap(),
- test if an argument equals zero,
- multiplication by 3,
- the binary operation $\oplus$.

All of these are database functions (recall that $\oplus$ is assumed to be a database function in the definition of a PRAM). It follows that fetch() and compute() are database functions.

Algorithm 28 Structure of functions fetch() and compute()

```
unroll for l = 0, ..., l_max
  unroll for s = 1, 2, 3
    if pc = 3l + s then
      compute according to Tables 4.2 and 4.3
    end if
  end unroll for
end unroll for
```

To see that $\mathcal{M}$ computes the same result as $\mathcal{P}$, we will show how a Microcode Processor of $\mathcal{M}$ simulates each instruction of the corresponding processor of $\mathcal{P}$, producing the same result under the above mentioned correspondence between PRAM registers and Microcode Machine memory.

Consider a processor of $\mathcal{P}$ with processor identifier pid executing, at location l, the instruction $r_i \leftarrow (r_{r_j}$ **of** $r_k)$. The corresponding Microcode Processor of $\mathcal{M}$ with the same pid will have $pc = 3l + 1$. Table 4.4 shows a trace of the register contents as the processor

Table 4.2: Microcode for PRAM simulation, part 1. For the PRAM instruction at location l, the table entries under Step s give the function values for $pc = 3l + s$ as follows: $\text{fetch}(pid, p, pc, acc) := i_{read}$ and $\text{compute}(pid, p, pc, acc, r) := (pc', acc', i_{write}, w)$.

Instruction		Step 1	Step 2	Step 3
$r_i \leftarrow c$		no-op	no-op	write
	i_{read}	0	0	0
	pc'	$pc+1$	$pc+1$	$pc+1$
	acc'	acc	acc	0
	i_{write}	0	0	$\text{mmap}_p(pid, i)$
	w	0	0	c
$r_i \leftarrow r_j \oplus r_k$		read r_j	no-op	write
	i_{read}	$\text{mmap}_p(pid, j)$	0	$\text{mmap}_p(pid, k)$
	pc'	$pc+1$	$pc+1$	$pc+1$
	acc'	r	acc	0
	i_{write}	0	0	$\text{mmap}_p(pid, i)$
	w	0	0	$acc \oplus r$
$r_i \leftarrow r_{r_j}$		get addr	no-op	write
	i_{read}	$\text{mmap}_p(pid, j)$	0	$\text{mmap}_p(pid, acc)$
	pc'	$pc+1$	$pc+1$	$pc+1$
	acc'	r	acc	0
	i_{write}	0	0	$\text{mmap}_p(pid, i)$
	w	0	0	r
$r_{r_i} \leftarrow r_j$		get addr	no-op	write
	i_{read}	$\text{mmap}_p(pid, i)$	0	$\text{mmap}_p(pid, j)$
	pc'	$pc+1$	$pc+1$	$pc+1$
	acc'	r	acc	0
	i_{write}	0	0	$\text{mmap}_p(pid, acc)$
	w	0	0	r
$r_i \leftarrow PID$		no-op	no-op	write
	i_{read}	0	0	0
	pc'	$pc+1$	$pc+1$	$pc+1$
	acc'	acc	acc	0
	i_{write}	0	0	$\text{mmap}_p(pid, i)$
	w	0	0	pid

Table 4.3: Microcode for PRAM simulation, part 2. For the PRAM instruction at location l, the table entries under Step s give the function values for $pc = 3l + s$ as follows: $\text{fetch}(pid, p, pc, acc) := i_{read}$ and $\text{compute}(pid, p, pc, acc, r) := (pc', acc', i_{write}, w)$.

Instruction		Step 1	Step 2	Step 3
$r_i \leftarrow (r_{r_j}$ **of** $r_k)$		get addr	get pid	write
	i_{read}	$\text{mmap}_p(pid, j)$	$\text{mmap}_p(pid, k)$	acc
	pc'	$pc + 1$	$pc + 1$	$pc + 1$
	acc'	r	$\text{mmap}_p(r, acc)$	0
	i_{write}	0	0	$\text{mmap}_p(pid, i)$
	w	0	0	r
$(r_{r_i}$ **of** $r_j) \leftarrow r_k$		get addr	get pid	write
	i_{read}	$\text{mmap}_p(pid, i)$	$\text{mmap}_p(pid, j)$	$\text{mmap}_p(pid, k)$
	pc'	$pc + 1$	$pc + 1$	$pc + 1$
	acc'	r	$\text{mmap}_p(r, acc)$	0
	i_{write}	0	0	acc
	w	0	0	r
goto l **if** $r_i > 0$		no-op	no-op	jump
	i_{read}	0	0	$\text{mmap}_p(pid, i)$
	pc'	$pc + 1$	$pc + 1$	$(r > 0\ ?\ 3l + 1 : pc + 1)$
	acc'	acc	acc	0
	i_{write}	0	0	0
	w	0	0	0
halt		no-op	no-op	halt
	i_{read}	0	0	0
	pc'	$pc + 1$	$pc + 1$	0
	acc'	acc	acc	0
	i_{write}	0	0	0
	w	0	0	0

Table 4.4: Trace of a Microcode Processor simulating a PRAM processor with processor identifier pid executing the instruction $r_i \leftarrow (r_{r_j}$ **of** $r_k)$ at location l. The line numbers refer to Algorithm 27. Modified registers at each step are ▸highlighted.

Step	Line	pc	acc	i	m	Memory
1	2	$3l+1$	$*$	▸$\text{mmap}_p(pid, j)$	$*$	
1	4	$3l+1$	$*$	$\text{mmap}_p(pid, j)$	▸r_j	
1	8	▸$3l+2$	▸r_j	▸0	▸0	
2	2	$3l+2$	r_j	▸$\text{mmap}_p(pid, k)$	0	
2	4	$3l+2$	r_j	$\text{mmap}_p(pid, k)$	▸r_k	
2	8	▸$3l+3$	▸$\text{mmap}_p(r_k, r_j)$	▸0	▸0	
3	2	$3l+3$	$\text{mmap}_p(r_k, r_j)$	▸$\text{mmap}_p(r_k, r_j)$	0	
3	4	$3l+3$	$\text{mmap}_p(r_k, r_j)$	$\text{mmap}_p(r_k, r_j)$	▸$(r_{r_j}$ **of** $r_k)$	
3	8	▸$3l+4$	▸0	▸$\text{mmap}_p(pid, i)$	▸$(r_{r_j}$ **of** $r_k)$	
3	10	$3l+4$	0	$\text{mmap}_p(pid, i)$	$(r_{r_j}$ **of** $r_k)$	▸$r_i \leftarrow (r_{r_j}$ **of** $r_k)$

executes three steps to simulate the PRAM instruction. In step 1, r_j is read and placed in the accumulator. In step 2, r_k is read and combined with the accumulator to form $\text{mmap}_p(r_k, r_j)$, the address of register r_j of processor r_k in shared memory. In the third step, this address is used to read from shared memory and write the result to register i of the current processor. After this we have $pc = 3l + 4 = 3(l + 1) + 1$, ready to simulate the next PRAM instruction at location $l + 1$. The other PRAM instructions are simulated in a similar way with no-op steps inserted to ensure synchronisation. Note that all writes are executed in the third step of the simulation.

Since each instruction of $\mathcal{P}$ is simulated by executing three steps of $\mathcal{M}$, the time on $\mathcal{M}$ is $3 \cdot T(n) = \text{O}(T(n))$ as claimed.

To see that $\mathcal{M}$ runs within space $S(n)$, note that by construction of the functions fetch() and compute() the simulation maintains the above-mentioned one-to-one correspondence between the processor registers of $\mathcal{P}$ and the shared memory cells of $\mathcal{M}$. This implies that the number of distinct shared memory cells accessed by $\mathcal{M}$ equals the number of distinct registers accessed by $\mathcal{P}$. The number of processors, $P(n)$ is included in the space bounds of both machines by definition and is the same for $\mathcal{P}$ and $\mathcal{M}$.

To establish the word size bound for $\mathcal{M}$, we need to consider the possible values in the local registers in addition to the shared memory contents. Apart from values that also appear in shared memory, the local registers contain the program counter and the results of address calculations. The maximum value $pc = 3l + s$ is a constant determined by the length of the program and independent of the input size n. Since $p - 1 < 2^{W(n)}$, the value $\text{mmap}_p(j, i)$ has word size $\text{O}(W(n))$, making the overall word size $\text{O}(W(n))$, as claimed. □

4.3 Simulation on a Database Machine

The Parallel Microcode Machine lends itself to be simulated by a Database Machine. Since the Database Machine has a bound on the row size that is logarithmic in the input size n, this requires the Parallel Microcode Machine to have a word size bound of $W(n) = \mathrm{O}(\log n)$.

Lemma 4.3.1. *Let $\mathcal{M}$ be a Parallel Microcode Machine with processor bound $P(n)$, space bound $S(n)$ and word size $W(n) = \mathrm{O}(\log n)$ that executes within time $T(n)$. Then there is a Database Machine $\mathcal{D}$ that computes the same result within time $\mathrm{O}(T(n) + \log\log n)$ and space $\mathrm{O}(S(n))$.*

Proof. Algorithm 29 simulates a Parallel Microcode Machine $\mathcal{M}$ on a Database Machine $\mathcal{D}$. The algorithm begins by determining the input size n and computing p, the number of processors to be activated. Recall that the processor bound $P(n)$ is a database function. Next, the algorithm converts the input to a table M representing the shared memory of $\mathcal{M}$. A pair $(i+1, x) \in M$ indicates that the shared memory cell M_i of $\mathcal{M}$ contains the value x. The index 0 is reserved for simulating reading a zero from an undefined memory location. Throughout the computation, only nonzero values will be stored in M.

Algorithm 29 Simulating a Parallel Microcode Machine on a Database Machine

```
1:  procedure SimulatePMM(I)
2:      n ← |I|
3:      p ← P(n)
4:      M ← [(1, n)] ⊎ [(2i + 2, x) | (i, x) ∈ I, x ≠ 0]
5:      C ← GenerateSequence(p)                          ▷ see Algorithm 3
6:      C ← [(pid, 1, 0, 0, 0) | pid ∈ C]                ▷ C has tuples (pid, pc, acc, i, m)
7:      while |C| > 0 do
8:          C ← [(pid, pc, acc, i_read, r)
                    | (pid, pc, acc, ∗, ∗) ∈ C
                    , i_read ← fetch(pid, p, pc, acc), (i_read, r) ∈_σ M]
9:          C ← [(pid, pc′, acc′, i_write, w)
                    | (pid, pc, acc, ∗, m) ∈ C
                    , (pc′, acc′, i_write, w) ← compute(pid, p, pc, acc, m)]
10:         U ← [(i_write, 0, pid, w) | (pid, ∗, ∗, i_write, w) ∈ C, i_write > 0]
11:         U ← U ⊎ [(i, 1, 0, x) | (i, x) ∈ M]
12:         M ← [(i, x′) | (i, c, pid, x) ∈ U, groupby(i, (c′, pid′, x′) ← min((c, pid, x)))
                    , x′ ≠ 0]
13:         C ← [(pid, pc, acc, i, m) | (pid, pc, acc, i, m) ∈ C, pc ≠ 0]
14:     end while
15:     m ← [m | (1, m) ∈_σ M]
16:     O ← GenerateSequence(m)
17:     O ← [(i, y) | i ∈ O, (2i + 2, y) ∈_σ M]
18:     return O
19: end procedure
```

Lines 5 and 6 initialise a processor table C. Each row is a tuple (pid, pc, acc, i, m) representing an active processor of $\mathcal{M}$, initialised with $pid \in \{0, \ldots, P(n)-1\}$, $pc = 1$ and zero in the remaining registers as per definition.

When a processor halts during the simulation, it will be removed from the processor table. While not all processors have halted, the simulation proceeds as follows.

Line 8 implements the memory read (lines 2 and 4 of Algorithm 27). Each processor computes $i_{read} = \text{fetch}(pid, p, pc, acc)$, the index to read from. A left outer join (denoted by $\Subset_\sigma$; see page 27) is then used to retrieve the value r corresponding to this index from table M. If i_{read} is not matched in M, the outer join returns $r = 0$. This is the case if $i_{read} = 0$, which is used for simulating reads from nonexisting processors, and also for memory cells containing zero which are not stored in M.

Line 9 implements the main computation (line 8 of Algorithm 27). Each processor computes $(pc', acc', i_{write}, w) = \text{compute}(pid, p, pc, acc, m)$ where pc' is the new value of the program counter and acc' is the new value of the accumulator. i_{write} is the index to write to and w is the value to be written.

To compute a new table M representing the state of the shared memory after writing, the algorithm needs to resolve any conflicting concurrent writes by multiple processors and to preserve the contents of all memory cells not written to. This is achieved by prioritising the writes by processor ID and treating the existing contents of memory the same as a write, but with a priority below that of any processor. Line 10 creates a table U where each update is represented as an index to write to (recall that the special index 0 indicates no write) and a triple (c, pid, x) with $c = 0$. The existing memory contents are then added to this table with $c = 1$ and $pid = 0$. Picking for each address i the minimum triple (c, pid, x) in lexicographic order has the desired effect of prioritising writes ($c = 0$) over existing memory contents ($c = 1$) and prioritising among different writes the one with the lowest processor ID. Line 12 computes the new memory contents by grouping by address and picking said minimum, eliminating all rows with a value of zero. Note that this preserves the invariant that M contains at most one row (i, x) for every index i. To prepare for the next iteration, all processors that have halted ($pc = 0$) are eliminated from the processor table C. After all processors have halted, the output O is extracted from the memory table M, using outer joins to fill in zeros not stored in M.

To analyse time and space complexity of the simulation, we first recall from Definition 4.2.2 that the processor IDs are bounded by the word size bound, which is assumed to be $W(n) = \mathrm{O}(\log n)$. This implies $P(n) = n^{\mathrm{O}(1)}$. The **while** loop is executed once for each step of the simulated machine $\mathcal{M}$, thus taking time $\mathrm{O}(T(n))$. Generating the processor table (line 5) takes $\mathrm{O}(\log \log P(n))$ operations by Theorem 2.5.3, which is $\mathrm{O}(\log \log n)$ since $P(n)$ is polynomial in n. Generating the index sequence for the output table (line 16) takes $\mathrm{O}(\log \log m)$ for an output size of m, which is also $\mathrm{O}(\log \log n)$ because m is bounded by the word size bound. Thus, the total time of the Database Machine is $\mathrm{O}(T(n) + \log \log n)$ as claimed.

Recall from Definition 4.2.2 that the space bound of a Parallel Microcode Machine bounds the number of accessed memory cells plus the number of processors. During the simulation on $\mathcal{D}$, this bounds the sizes of tables M and C, respectively. The initial creation of C takes $\mathrm{O}(P(n))$ space by Theorem 2.5.3. The size of C stays the same after the join on line 8 since each memory address is unique in M. It can only decrease when the rows

representing halted processors are removed on line 13. The size of table U is at most $|U| = |M| + |C|$. Overall, this proves that $\mathcal{D}$ uses space $O(S(n))$ as claimed.

Since the word size of $\mathcal{M}$ is assumed to be $O(\log n)$ and all tables used in $\mathcal{D}$ have fixed-length tuples containing values of this word size, there is a suitable row width bound for $\mathcal{D}$ as required by the definition of a Database Machine (Definition 2.3.1). □

Combining this result with the simulation from the previous section now allows us to prove the main theorem of this chapter.

Proof of Theorem 4.1.1. By Lemma 4.2.1, the PRAM $\mathcal{P}$ can be simulated by a Parallel Microcode Machine $\mathcal{M}$ with the same processor bound $P(n)$, space $S(n)$, time $O(T(n))$ and word size $O(W(n)) = O(\log n)$. $\mathcal{M}$ satisfies the requirements of Lemma 4.3.1, so it can be simulated by a Database Machine $\mathcal{D}$ as claimed in the theorem. □

4.4 Simulating a Relational Machine

In the previous sections we have shown that a Database Machine can simulate a PRAM with logarithmic word size bound. In this section we will show that a PRAM can simulate a Relational Machine with a logarithmic slowdown.

4.4.1 Notational conventions

To formulate PRAM algorithms, we will use a high-level language that can readily be translated to a sequence of "assembly language" instructions for a PRAM. Instead of register numbers, we will use variable names and assume that different variable names refer to different registers. The exact allocation of variables to PRAM registers does not matter, but it is assumed that the lowest-numbered registers are used so that the word size required for address calculations is minimal. A bounded number of scratch registers starting at index zero is reserved for translating arithmetic expressions and indirect addressing.

Nested data structures and arrays are written similar to the ANSI C programming language, with the exception that we use subscripts for array indexing. For example, let l and r be integer constants and consider the data structure that is given in C syntax in Figure 4.2. This is an array of $r + 1$ structures, each of which contain three variables and two arrays of size l. An expression like $R_3.k_5$ refers to a single integer variable. The notation R_1 refers to the $2l + 3$ variables $R_1.e$, $R_1.s$, $R_1.t$, $R_1.k_0$, ..., $R_1.k_{l-1}$, $R_1.v_0$, ..., $R_1.v_{l-1}$ and $R_1.k$ refers to the l variables $R_1.k_0$, ..., $R_1.k_{l-1}$. An assignment $R_i \leftarrow R_j$ copies all $2l + 3$ variables from one structure to another; the assignment $R_i \leftarrow 0$ denotes setting all $2l + 3$ variables to zero.

All variable names refer to local registers of the current processor. A processor ID in square brackets is used to denote accessing a variable in a different processor. For example, if PRAM register r_5 is allocated to variable x, r_6 is allocated to variable y and the processor with $PID = 7$ executes the assignment $x \leftarrow y[PID + 1]$, this translates to the following sequence of operations: $r_0 \leftarrow 6$; compute $r_1 = 8$ by loading PID and incrementing it; $r_5 \leftarrow (r_{r_0}$ **of** $r_1)$.

Figure 4.2: Data structure for simulating a Relational Machine in C notation

```
struct _rtable
{
    int e;
    int s;
    int t;
    int k[l];
    int v[l];
} R[r+1];
```

Each line of code is executed by all processors synchronously. To maintain this synchronisation, conditional instructions, denoted by **if** *condition* **then** *statements*$_1$ **else** *statements*$_2$ **end if**, are assumed to be translated such that the block of code representing *statements*$_1$ is executed first by those processors for which *condition* is *true* while the other processors execute the same number of instructions $r_2 \leftarrow 0$, reserving PRAM register r_2 as a dummy for this purpose. Next, the processors for which *condition* was *false* execute *statements*$_2$ while the other processors execute the same number of padding instructions.

We use C-style notation for the logical operators and assume short-circuit evaluation: the term p && q denotes p **and** q and q is only evaluated if p is *true*. The term $p \mid\mid q$ denotes p **or** q and q is only evaluated if p is *false*.

We speak of a *column* x to refer to the contents of the variable x across all processors.

4.4.2 Prerequisite algorithms and techniques

For simulating a Relational Machine on a PRAM, we need some way to represent rtables and other two-dimensional arrays as helper tables.

Definition 4.4.1. A *packed array* is a data structure in the register set of a PRAM representing an $n \times m$ array of nonnegative integers as follows. The array is stored in data columns $x_1, \ldots, x_m$. There is an *indicator column* e such that $e[i] = 1$ for $0 \leq i < n$ and $e[i] = 0$ otherwise. The i-th row of the array is stored in registers $x_1, \ldots, x_m$ of processor $i - 1$.

When writing code that works on all rows of a packed array in parallel, we can simply wrap it in **if** $e \neq 0$ **then** ... **end if** to execute it only on those processors in which rows are actually present. Each processor has access to one row in its local registers $x_1, \ldots, x_m$ and can access any element $a_{i,j}$ by referencing $x_j[i - 1]$.

Lemma 4.4.1 (row count)**.** *A PRAM can determine the number of rows in a packed array in* O(1) *time and* O(1) *space.*

Proof. Algorithm 30 determines the number of rows in a packed array with indicator e. A variable n in processor 0 is initialised to zero. Every processor then checks if it has the last row of the table and in this case writes the number of rows to the variable $n[0]$.

Algorithm 30 Determining the size of a packed array with indicator e

```
1: function GetRowCount(e)
2:     n[0] ← 0
3:     if e ≠ 0 && e[PID + 1] = 0 then
4:         n[0] ← PID + 1
5:     end if
6:     return n[0]
7: end function
```

Since the table is assumed to be stored contiguously, starting at processor 0, only the processor holding the last row will execute this write, unless the packed array is empty in which case the result is also correct. The single register $n[0]$ is the only space used. □

Lemma 4.4.2 (binary search)**.** *Let a packed array with n rows and indicator e_v be stored in a PRAM in ascending order by a column v and let x be a column of a packed array with indicator e_x. Algorithm 31 computes on each processor with $e_x = 1$ the largest index l such that $v[l] \leq x$ or 0 if such an index does not exist. It takes time $\mathrm{O}(\log n)$ and space $\mathrm{O}(1)$.*

Proof. Algorithm 31 is a simple binary search running on all rows of x in parallel. As a first step, the size of the lookup array v is computed as $n[0]$, taking $\mathrm{O}(1)$ operations by Lemma 4.4.1.

Algorithm 31 Parallel binary search

```
1: function ParallelBinarySearch(e_v, v, e_x, x)
2:     n[0] ← GetRowCount(e_v)
3:     if e_x ≠ 0 then
4:         l ← 0
5:         r ← n[0]
6:         while n[0] > 1 do
7:             m ← ⌊(l + r)/2⌋
8:             if v[m] > x then
9:                 r ← m
10:            else
11:                l ← m
12:            end if
13:            if PID = 0 then
14:                n ← ⌈n/2⌉
15:            end if
16:        end while
17:    end if
18:    return l
19: end function
```

The condition of the **while** loop depends on the same variable $n[0]$ for all processors, ensuring that all processors execute the same number of iterations. The following invariants are maintained throughout the loop on each processor and are easily verified:

1. $r \geq l$;
2. $r - l \leq n[0]$;

3. $v[l] \leq x$, unless $v[0] > x$ in which case we have $l = 0$ throughout.

In the loop, $n[0]$ is maintained as an upper bound on the length of the search interval on any processor. The number of iterations is $\mathrm{O}(\log n)$ since the search interval is cut in half at each step. □

Lemma 4.4.3 (prefix computation)**.** *Let $v_1, \ldots, v_n$ be a sequence of nonnegative integers and $\oplus\colon \mathbb{N}_0 \to \mathbb{N}_0$ an associative binary operation that is a database function. Then the sequence $p_k = \bigoplus_{i=1}^{k} v_i$, $k = 1, \ldots, n$, can be computed on a PRAM by n processors in $\mathrm{O}(\log n)$ time using $\mathrm{O}(n)$ space.*

Proof. The algorithm was expressed in terms of circuits by Ladner and Fischer [55] and is given as a recursive algorithm for the PRAM in [29]. It works as follows: for even i, compute $v'_{i/2} = v_{i-1} \oplus v_i$. For the resulting sequence $v'_1, \ldots, v'_{\lfloor n/2 \rfloor}$, recursively compute prefixes $p'_k = \bigoplus_{i=1}^{k} v'_i$, $k = 1, \ldots, \lfloor n/2 \rfloor$. The resulting prefix sequence is then obtained as follows: $p_1 = v_1$. For even i, set $p_i = v'_{i/2}$. For odd $i > 1$, compute $p_i = v'_{(i-1)/2} \oplus v_i$.

Algorithm 32 implements this idea using loops instead of recursion. The input sequence is stored in register x across processors $0, \ldots, n[0] - 1$; the length of the sequence is passed in $n[0]$.

Algorithm 32 Prefix computation

```
 1: function ComputePrefixes⊕(x, n[0])
 2:     y_0 ← x
 3:     c_0[0] ← n[0]
 4:     i[0] ← 0
 5:     while c_{i[0]}[0] > 1 do
 6:         i[0] ← i[0] + 1
 7:         c_{i[0]}[0] ← ⌊c_{i[0]−1}[0]/2⌋
 8:         if PID < c_{i[0]}[0] then
 9:             y_{i[0]} ← y_{i[0]−1}[2 · PID] ⊕ y_{i[0]−1}[2 · PID + 1]
10:         end if
11:     end while
12:     while i[0] > 0 do
13:         i[0] ← i[0] − 1
14:         if PID > 0 && PID < c_{i[0]}[0] then
15:             if PID mod 2 = 0 then
16:                 y_{i[0]} ← y_{i[0]+1}[PID/2 − 1] ⊕ y_{i[0]}
17:             else
18:                 y_{i[0]} ← y_{i[0]+1}[(PID − 1)/2]
19:             end if
20:         end if
21:     end while
22:     return y_0
23: end function
```

The algorithm first copies the input sequence x to y_0, which will contain the result at the end, and sets the row count $c_0[0]$. Register $i[0]$, a single register in PRAM processor 0, is used to keep track of the recursion level. At level $i[0]$, $y_{i[0]}$ plays the role of v' in the above

description and $c_{i[0]}[0]$ is the count of elements in that sequence. The first loop computes the binary operation on adjacent pairs for all levels. The second loop then combines them in the reverse order according to the recursive algorithm described above. Note that the terminating conditions for both **while** loops depend on variables in processor 0 only, ensuring that all processors execute the same number of iterations and synchronisation is maintained.

Since the length of the new sequence $y_{i[0]}$ is halved at each iteration, the number of iterations of each loop is $\log n$, making the time $O(\log n)$, as claimed. Temporary space usage for the sequences y_i is the geometric sum $\lfloor n/2 \rfloor + \cdots + \lfloor n/2^i \rfloor + \cdots + 1 < n$ and there are $\log n$ variables for the length counts. This makes the total temporary space usage $O(n)$, as claimed. □

Lemma 4.4.4 (compacting an array)**.** *Let r be a column, $e \in \{0, 1\}$ on all processors and $n[0]$ such that $e[i] = 0$ for all $i \geq n[0]$. Algorithm 33 converts column r to a packed array with indicator e, i.e. it rearranges all rows with $e = 1$ such that they are stored contiguously in processors $0, 1, \ldots$, preserving their order. It takes time $O(\log n[0])$ and space $O(n[0])$.*

Proof. Algorithm 33 performs a prefix computation on column e using integer addition, taking time $O(\log n[0])$ and space $O(n[0])$ by Lemma 4.4.3. Since $e \in \{0, 1\}$, the result is a contiguous sequence of integers starting at 1 in all rows with $e = 1$. Subtracting 1 yields the ID of the processor in which to store each row.

Algorithm 33 Compacting an array into a packed array

```
1:  procedure CompactArray(e, r, n[0])
2:      i ← ComputePrefixes+(e, n[0])
3:      if e ≠ 0 then
4:          r[i − 1] ← r                          ▷ simultaneous write after read
5:      end if
6:      if PID ≤ i[n[0] − 1] then
7:          e ← 1
8:      else
9:          e ← 0
10:     end if
11: end procedure
```

It remains to copy each row to its new location and to turn e into an indicator column, using $O(1)$ operations. Note that the PRAM executes instructions synchronously in such a way that reading occurs before writing. This ensures that no rows get overwritten before being copied to their new location. □

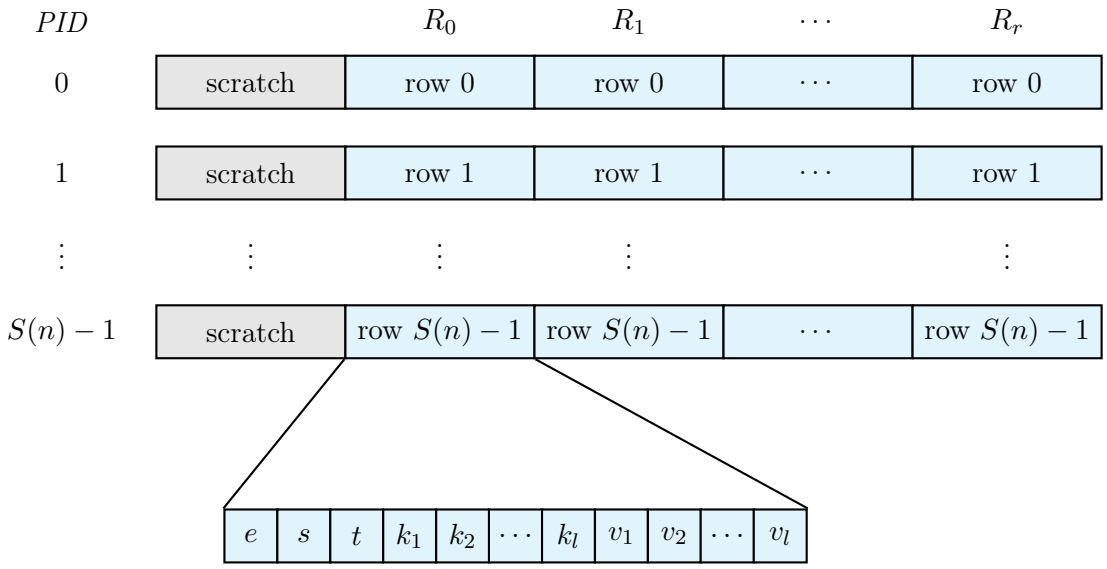
Lemma 4.4.5 (sorting)**.** *A packed array with n rows (stored in n different processors by definition) can be sorted by a column v using $O(\log n)$ time and $O(n)$ space.*

Proof. This can be done using the parallel merge sort algorithm presented in [22]. □

We denote sorting a packed array A with indicator column $A.e$ by value V as

$$\mathrm{Sort}(A, A.e, A.v).$$

Figure 4.3: Memory map for simulating a Relational Machine on a PRAM



4.4.3 The simulation

In order to show that a PRAM can simulate a Relational Machine, we will use the following memory map throughout this section. Let $\mathcal{R}$ be a Relational Machine with r registers and a maximum key and value tuple length of l. The simulating PRAM will use only even-numbered registers to store the contents of $\mathcal{R}$'s rtable registers according to the memory map in Figure 4.3, reserving the odd-numbered registers for the input and output. Each processor stores one row of each rtable register in its local registers. The processor bound of the simulating PRAM therefore needs to accomodate the maximum number of rows in any rtable register of $\mathcal{R}$ which is bounded by $\mathcal{R}$'s space bound.

For each row, l registers each are allocated for the key and value tuples. The length of the key tuple is stored in s and the length of the value tuple is stored in t. Each rtable R_i is stored as a packed array with its own indicator column $R_i.e$. This memory layout corresponds to the structure presented as an example in Figure 4.2 above. A fixed number of registers starting at r_0 and an extra rtable register R_0 are allocated as scratch space.

Theorem 4.4.6. *Let $\mathcal{R}$ be a Relational Machine that runs in time $T(n)$ and space $S(n)$ where $S(n)$ is a database function. Then there exists a PRAM $\mathcal{P}$ with processor bound $S(n)$ that simulates $\mathcal{R}$ in time $\mathrm{O}(T(n) \cdot \log S(n))$, space $\mathrm{O}(S(n))$ and with a word size bound of $\mathrm{O}(\log n)$.*

The proof is structured as a series of lemmas showing how to simulate each of the basic operations of a Relational Machine on a PRAM. The most involved of these constructions is the simulation of `JOIN` A, B, C, which we will discuss first. Recall that this operation joins two tables on equality of their value tuples. For brevity, we write v for the entire value tuple $v_1, \ldots, v_l$ and assume lexicographic ordering. We will accompany the following explanation with a running example of two tables being joined. Figure 4.4 shows the first

Figure 4.4: Example for `JOIN` A, B, C

PID	*A.v*	*B.v*		*A.v*	*B.v*
0	5	7		1	3
1	7	3		1	3
2	5	5		3	5
3	1	8	sort $\Longrightarrow$	5	5
4	7	3		5	7
5	1	8		5	8
6	3	5		6	8
7	5			7	
8	6			7	

step, sorting both tables by value tuple. In the figures, we use a single integer to represent the value tuple.

Definition 4.4.2. Let A be an rtable stored in a PRAM such that it is sorted by its value tuple. A *range table* for A is a packed array with columns $(v, start, end)$ and indicator e that contains for each value v occurring in the value tuple of A a row with the range of row indices (*start* to *end* inclusive) of this value in A.

Lemma 4.4.7. *Algorithm 34 takes an rtable A, assumed to be sorted by its value tuple $A.v$, and computes a range table with the specified indicator e and columns $(v, start, end)$ that is also sorted by v. It takes $\mathrm{O}(\log |A|)$ time on a PRAM.*

Algorithm 34 Computing a range table

```
1:  procedure ComputeRangeTable(A, e, v, start, end)        ▷ A is sorted by A.v
2:      if A.e ≠ 0 then
3:          first ← (PID = 0 || A.v ≠ A.v[PID − 1] ? 1 : 0)
4:          last ← (A.e[PID + 1] = 0 || A.v ≠ A.v[PID + 1] ? 1 : 0)
5:      end if
6:      n[0] ← GetRowCount(A.e)
7:      i_first ← ComputePrefixes_+(first, n[0])
8:      i_last ← ComputePrefixes_+(last, n[0])
9:      if first ≠ 0 then
10:         e[i_first − 1] ← 1
11:         v[i_first − 1] ← A.v
12:         start[i_first − 1] ← PID
13:     end if
14:     if last ≠ 0 then
15:         end[i_last − 1] ← PID
16:     end if
17: end procedure
```

Proof. The algorithm first computes two helper columns *first* and *last*, containing 1 if a row is the first or the last in a range of rows with equal values of v, respectively, and 0 otherwise. A prefix sum computation is then performed on each of these columns, yielding

Figure 4.5: Example for `JOIN` A, B, C, continued

PID	*A.v*	*first*	*last*	i_{first}	i_{last}
0	1	1	0	1	0
1	1	0	1	1	1
2	3	1	1	2	2
3	5	1	0	3	2
4	5	0	0	3	2
5	5	0	1	3	3
6	6	1	1	4	4
7	7	1	0	5	4
8	7	0	1	5	5

(entries not used in the next step are shown in grey)

Figure 4.6: Example for `JOIN` A, B, C, continued (2)

PID	e_A	v_A	$start_A$	end_A	e_B	v_B	$start_B$	end_B
0	1	1	0	1	1	3	0	1
1	1	3	2	2	1	5	2	3
2	1	5	3	5	1	7	4	4
3	1	6	6	6	1	8	5	6
4	1	7	7	8				

columns i_{first} and i_{last}, respectively (see Figure 4.5). Finally, lines 9–16 compute the resulting packed array: each processor that has a first row of a range writes its *PID* to *start* at the correct index and each processor that has a last row of a range does the same for *end*. Note that the number of ones in *first* and *last* is identical since each range has exactly one first row and one last row. The result is illustrated in Figure 4.6 for both input tables.

By Lemma 4.4.3, the prefix computations take time $O(\log |A|)$. The remaining instruction take time $O(1)$, making the overall time for ComputeRangeTable$(A, \ldots)$ $O(\log |A|)$ as claimed. □

Lemma 4.4.8. *Algorithm 35 simulates the Relational Machine operation* `JOIN` A, B, C *on a PRAM, taking time* $O(\log |A| + \log |B|)$.

Proof. Here is a brief overview of the construction. The algorithm first sorts each of the input tables by their value tuple and computes a range table. From these two tables, it derives a table combining for each value of v occurring in both A and B the corresponding row ranges in A and B. For each value of v, the product of the size of the range in A and the size of the range in B is the size of the range in the result table C. A prefix sum calculation yields the total number of rows in C and the starting row of each range. As a last step, $|C|$ processors assemble their rows in parallel, computing the required indices in A and B using these helper tables.

Figure 4.6 shows the range tables computed for our example join in the first step. Next, each processor holding a row of the range table of A with value v performs a binary search

Algorithm 35 Simulating `JOIN` A, B, C on a PRAM

1: $n[0] \leftarrow \text{GetRowCount}(e_A)$
2: $\text{Sort}(A, A.e, A.v)$
3: $\text{ComputeRangeTable}(A, e_A, v_A, start_A, end_A)$
4: $\text{Sort}(B, B.e, B.v)$
5: $\text{ComputeRangeTable}(B, e_B, v_B, start_B, end_B)$
6: $b \leftarrow \text{ParallelBinarySearch}(e_B, v_B, e_A, v_A)$ ▷ find v_A in v_B
7: **if** $e_A \neq 0$ **then**
8: **if** $v_B[p] = v_A$ **then**
9: $start'_B \leftarrow start_B[b]$
10: $end'_B \leftarrow end_B[b]$
11: **else**
12: $e_A \leftarrow 0$
13: **end if**
14: **end if**
15: $\text{CompactArray}(e_a, (start_A, end_A, start'_B, end'_B), n[0])$
16: **if** $e_A \neq 0$ **then**
17: $r \leftarrow (end_A + 1 - start_A) \cdot (end'_B + 1 - start'_B)$ ▷ length of range in result table
18: **end if**
19: $n[0] \leftarrow \text{GetRowCount}(e_A)$
20: $r' \leftarrow \text{ComputePrefixes}_+(r, n[0])$
21: **if** $e_A \neq 0$ && $e_A[PID + 1] = 0$ **then**
22: $r'_{max}[0] \leftarrow r'$
23: **end if**
24: **if** $PID < r'_{max}[0]$ **then**
25: $R_0.e \leftarrow 1$
26: $p \leftarrow PID$
27: $i \leftarrow \text{ParallelBinarySearch}(e_A, r', R_0.e, p)$ ▷ find largest i with $r'[i] \leq p$
28: **if** $p \geq r'[i]$ **then**
29: $p \leftarrow p - r'[i]$
30: $r_A \leftarrow start_A + \lfloor p/(end'_B[i + 1] + 1 - start'_B[i + 1]) \rfloor$
31: $r_B \leftarrow start'_B[i + 1] + p \bmod (end'_B[i + 1] + 1 - start'_B[i + 1])$
32: **else**
33: $r_A \leftarrow start_A + \lfloor p/(end'_B[i] + 1 - start'_B[i]) \rfloor$
34: $r_B \leftarrow start'_B[i] + p \bmod (end'_B[i] + 1 - start'_B[i])$
35: **end if**
36: $R_0.s \leftarrow A.s[r_A] + B.s[r_B]$
37: $R_0.t \leftarrow 0$
38: $R_0.k \leftarrow A.k[r_A]$
39: **for** $j = 1, \ldots, B.s[r_B]$ **do**
40: $R_0.k_{j+A.s[r_A]} \leftarrow B.k_j[r_B]$
41: **end for**
42: **end if**
43: $C \leftarrow R_0$
44: $R_0 \leftarrow 0$

Figure 4.7: Example for `JOIN` A, B, C, continued (3)

PID	e_A	v_A	$start_A$	end_A	b	e_B	v_B	$start_B$	end_B
0	1	1	0	1	0	1	3	0	1
1	1	3	2	2	0	1	5	2	3
2	1	5	3	5	1	1	7	4	4
3	1	6	6	6	1	1	8	5	6
4	1	7	7	8	2				

Figure 4.8: Example for `JOIN` A, B, C, continued (4)

PID	e_A	v_A	$start_A$	end_A	$start'_B$	end'_B
0	~~1~~ 0	1	0	1	0	0
1	1	3	2	2	0	1
2	1	5	3	5	2	3
3	~~1~~ 0	6	6	6	0	0
4	1	7	7	8	4	4

⇓ compact

PID	e_A	v_A	$start_A$	end_A	$start'_B$	end'_B
0	1	3	2	2	0	1
1	1	5	3	5	2	3
2	1	7	7	8	4	4

for v in the range table of B, taking time $O(\log |B|)$ by Lemma 4.4.2. The result is shown as column b in Figure 4.7. If v is found in B, the algorithm copies the corresponding starting and ending indices to $start'_B$ and end'_B. Otherwise it marks the row to be deleted by setting $e_A \leftarrow 0$.

Compacting the array with indicator e_A and columns $(v_A, start_A, end_A, start'_B, end'_B)$ using Algorithm 33 yields a packed array containing only values v_A occurring in both input tables together with their corresponding row ranges within the sorted tables (Figure 4.8). This takes time $O(\log |A|)$ by Lemma 4.4.4.

The next step (line 17) is to compute for each value in v_A the number of rows with this value in the result table, r. A prefix sum computation yields r' where the starting row for the range of values $v_A[i]$ is $r'[i-1]$, or zero for $i = 0$. Since the number of distinct values in the result is at most $|A|$, this prefix sum computation takes time $O(\log |A|)$ by Lemma 4.4.3. The maximum value in column r' is the number of rows in the result table and stored in $r'_{max}[0]$ (line 22). See Figure 4.9 for a continued example.

Now each processor with $PID < r'_{max}[0]$ assembles a row of the result in the scratch space R_0. To compute which input rows to join, it looks up its PID in column r' using binary search. By Lemma 4.4.2, this returns the largest index i such that $r'[i] \leq p$ or zero if $r'[0] > p$. Subtracting $r'[i]$ from p if possible yields the relative position p of the current output row within the range. From this, the row numbers r_A and r_B within the input tables are computed (see Figure 4.10).

Figure 4.9: Example for `JOIN` A, B, C, continued (5)

PID	e_A	v_A	$start_A$	end_A	$start'_B$	end'_B	r	r'	r'_{max}
0	1	3	2	2	0	1	2	2	10
1	1	5	3	5	2	3	6	8	
2	1	7	7	8	4	4	2	10	

Figure 4.10: Example for `JOIN` A, B, C, continued (6)

PID	$start_A$	end_A	$start'_B$	end'_B	r'	r'_{max}	i	p	r_A	r_B
0	2	2	0	1	2	10	0	0	2	0
1	3	5	2	3	8		0	1	2	1
2	7	8	4	4	10		0	0	3	2
3							0	1	3	3
4							0	2	4	2
5							0	3	4	3
6							0	4	5	2
7							0	5	5	3
8							1	0	7	4
9							1	1	8	4

It remains to assemble the output key tuple by concatenating the input key tuples. Finally, the output is copied to the destination C and the scratch space R_0 is cleared. Summarising the time taken by this algorithm, we get

- $O(\log |A| + \log |B|)$ for sorting the input rtables A and B (Lemma 4.4.5),
- $O(\log |B|)$ for binary search in the range table for B (line 6, Lemma 4.4.2)
- $O(\log |A|)$ for compacting the range table (line 15, Lemma 4.4.4)
- $O(\log |A|)$ for prefix computation (line 20, Lemma 4.4.3)
- $O(\log |A|)$ for binary search (line 27, Lemma 4.4.2)
- $O(1)$ for everything else.

Note that the range tables computed from A and B can have at most as many rows as A and B, respectively. This makes the overall runtime of the algorithm $O(\log |A| + \log |B|)$, as claimed. □

Lemma 4.4.9. *Algorithm 36 simulates the Relational Machine operation* `RANGE` A *on a PRAM taking time* $O(\log |A|)$.

Proof. The algorithm first sorts the table lexicographically by the value tuple, taking $O(\log |A|)$ time by Lemma 4.4.5.

Each row whose value tuple is equal to that of the previous row is marked to be deleted by setting $A.e$ to 0. This preserves for every distinct value of $A.v$ only the first row with that value. The value tuples are moved to the key tuples and the table is compacted, again taking $O(\log |A|)$ time by Lemma 4.4.4. □

Algorithm 36 Simulating `RANGE` A on a PRAM

```
1: n[0] ← GetRowCount(A.e)
2: Sort(A, A.e, A.v)
3: if A.e ≠ 0 then
4:    if PID ≠ 0 && A.v = A.v[PID − 1] then
5:       A.e ← 0
6:    end if
7: end if
8: A.k ← A.v
9: A.s ← A.t
10: A.v ← 0
11: A.t ← 0
12: CompactArray(A.e, A, n[0])
```

Lemma 4.4.10. *Algorithm 37 simulates the Relational Machine operation* `SINGLES` A *on a PRAM taking time* $O(\log |A|)$.

Proof. The algorithm first sorts the table lexicographically by the value tuple, taking $O(\log |A|)$ time by Lemma 4.4.5.

Algorithm 37 Simulating `SINGLES` A on a PRAM

```
1: n[0] ← GetRowCount(A.e)
2: Sort(A, A.e, A.v)
3: if A.e ≠ 0 then
4:    if (PID ≠ 0 && A.v = A.v[PID − 1]) || (A.e[PID + 1] ≠ 0 && A.v = A.v[PID + 1])
   then
5:       A.e ← 0
6:    end if
7: end if
8: A.v ← 0
9: A.t ← 0
10: CompactArray(A.e, A, n[0])
```

Each processor tests its row for uniqueness of the value tuple. It is not unique if it is equal to the previous row or to the next row. In this case, the row is marked to be deleted by setting $A.e$ to 0. It remains to clear the value tuple and to compact the table, taking time $O(\log |A|)$ by Lemma 4.4.4. □

Lemma 4.4.11. *Each of the Relational Machine mapping primitives defined in Section 3.2.2 can be simulated by a PRAM taking time* $O(1)$ *and* $O(1)$ *temporary space.*

Proof. This is straightforward. All of the mapping primitives perform the same operations on all rows of an rtable in parallel. To implement this on a PRAM that has a processor assigned to each row, we just need to make sure that only those processors that have a valid row execute the operation.

Algorithm 38 shows how the `ROTK` operation is implemented. Note that the variables $A.s$ and $A.t$ that store the key and value tuple lengths of the rtable contain the same values for each row. This is preserved as an invariant by all operations. It ensures that the **for**

Algorithm 38 Simulating `ROTK` A on a PRAM

```
1: if A.e ≠ 0 && A.s > 1 then
2:    temp ← A.k_1
3:    for i = 1, ..., A.s − 1 do          ▷ A.s is the same across all processors
4:       A.k_i ← A.k_{i+1}
5:    end for
6:    A.k_{A.s} ← temp
7: end if
```

loop executes the same number of iterations on all processors as required by our use of pseudo-code for formulating PRAM algorithms.

Algorithm 39 Simulating `CDEC` A on a PRAM

```
1: if A.e ≠ 0 && A.t ≥ 2 && A.v_1 ≠ 0 && A.v_2 ≠ 0 then
2:    A.v_2 ← A.v_2 − 1
3: end if
```

Algorithm 40 Simulating `CSHIFT` A on a PRAM

```
1: if A.e ≠ 0 && A.t ≥ 3 && A.v_1 ≠ 0 then
2:    A.v_3 ← 2 · A.v_3 + (A.v_2 mod 2)
3:    A.v_2 ← ⌊A.v_2/2⌋
4: end if
```

Note that incrementing, decrementing, and bit shifting are all expressible as binary operations that are database functions and thus implementable with our PRAM instruction set. Algorithms 39 and 40 show example simulations of two more mapping primitives, `CDEC` and `CSHIFT`. The rest of the mapping primitives are implemented in the same way and omitted here for brevity. □

Lemma 4.4.12. *The Relational Machine instructions for copying data, loading constants and changing control flow described in sections 3.2.3 and 3.2.4 can be simulated on a PRAM using time* O(1) *and no temporary space.*

Proof. Copying an rtable register to another is simply done by all processors in parallel, one PRAM register at a time. Since the simulation uses a fixed number of registers per rtable row, this takes O(1) operations. Loading one of the two constant tables into any rtable register is also straightforward: first, all processors clear the register and then one or two processors write the one or two rows of the constant tables `ONE` or `TWO`.

The Relational Machine can change control flow only based on the emptiness of an rtable register. To test whether a simulated rtable register A contains the empty table, all processors simply check if $A.e[0] = 0$ and simultaneously change control flow based on the result. □

The previous Lemmas have shown how to simulate each of the instructions of a Relational Machine on a PRAM assuming its rtable registers are stored in the even-numbered registers of the PRAM as described at the beginning of this section. We can now put these together to prove the main theorem.

Proof of Theorem 4.4.6. Let $\mathcal{R}$ be a Relational Machine that runs in time $T(n)$ and space $S(n)$. Recall that a computation on a Relational Machine maps a key-only input rtable to a key-only output rtable. A PRAM maps an input n-tuple to an output m-tuple of nonnegative integers. To simulate $\mathcal{R}$ on a PRAM, we serialise the input and output rtables into a tuple of integers as follows: let s be the key tuple size of an rtable with empty value tuples and n the number of rows. The rtable is represented as a tuple of $n \cdot s + 1$ integers $x_0, \ldots, x_{n \cdot s}$ where $x_0 = s$ and the j-th tuple element of row i is stored in $x_{i \cdot s + j}$; tuple elements are numbered $j = 1, \ldots, s$ and rows are numbered $i = 0, \ldots, n - 1$.

The simulating PRAM $\mathcal{P}$ uses $S(n)$ processors so that each rtable row can always be stored in a different processor. The first step of the simulation is to deserialise the input rtable into even-numbered PRAM registers according to the memory map described at the beginning of this section. Algorithm 41 performs this transformation. The notation $r_i[j]$ stands for PRAM register r_i on processor j; otherwise we use variable names referring to the memory map shown in Figure 4.3 as described in Section 4.4.1.

The algorithm first determines the number of processors for index calculations on $\mathcal{P}$'s input sequence and computes the number of rows of the input rtable from the total size of the input. Next, one processor per input row copies its row from the input sequence to the memory allocated for rtable register R_1. This takes time O(1) since the input tuple size s is bounded by $\mathcal{R}$'s maximum tuple size which is a constant.

Algorithm 41 Deserialising an rtable from an input sequence

```
1:  p ← PID
2:  if p[PID + 1] = 0 then          ▷ reading from a nonexisting processor returns zero
3:      p[0] ← PID + 1              ▷ p[0] is now the number of activated processors
4:  end if
5:  if PID = 0 then
6:      rows ← (r_0 − 1)/r_1
7:  end if
8:  if PID < rows[0] then
9:      R_1.e ← 1
10:     R_1.s ← r_1[0]
11:     for i[0] = 1, ..., r_1[0] do
12:         j ← PID · r_1[0] + i[0]
13:         R_1.k_{i[0]} ← r_{2⌊j/p[0]⌋+1}[j mod p[0]]
14:     end for
15: end if
```

The simulation now commences using the algorithms presented on the previous pages to simulate each Relational Machine instruction. The three "heavy" relational Machine instructions take time $O(\log S(n))$ by Lemma 4.4.8 (`JOIN`), Lemma 4.4.9 (`RANGE`), and Lemma 4.4.10 (`SINGLES`). The remaining instructions take time O(1) by Lemma 4.4.11 (mapping primitives) and Lemma 4.4.12 (miscellaneous operations). This establishes the overall claim that $\mathcal{P}$ takes time $O(T(n) \cdot \log S(n))$.

The Relational Machine's `HALT` instruction is simulated by executing Algorithm 42 to serialise the output rtable in R_1. This is a straightforward reversal of the deserialisation at the beginning, taking time O(1).

Algorithm 42 Serialising an rtable to an output sequence

```
1: p ← PID
2: if p[PID + 1] = 0 then                ▷ reading from a nonexisting processor returns zero
3:    p[0] ← PID + 1                     ▷ p[0] is now the number of activated processors
4: end if
5: rows[0] ← GetRowCount(R_1.e)
6: if PID = 0 then
7:    r_1 ← R_1.s                                                          ▷ key tuple width
8:    r_0 ← rows · R_1.s + 1                                               ▷ compute length of output sequence
9: end if
10: if R_1.e ≠ 0 then
11:    for i[0] = 1, ..., r_1[0] do
12:       j ← PID · r_1[0] + i[0]
13:       r_{2⌊j/p[0]⌋+1}[j mod p[0]] ← R_1.k_{i[0]}
14:    end for
15: end if
```

Finally, all processors of $\mathcal{P}$ halt simultaneously. □

Theorem 4.4.6 makes a claim about the asymptotic complexity of simulating a Relational Machine on a PRAM, and the constructions used for its proof were chosen for that purpose. It seems likely that an actual implementation of the Relational Machine primitives on a parallel computer can be made more efficient. This is left as future work.

Chapter 5

Running PRAM algorithms in-database

In Chapter 4 we have shown how an algorithm formally specified in terms of our PRAM instruction set can be translated to an algorithm for the Database Machine. This is a general-purpose method that could be used to implement a compiler for this kind of translation.

However, PRAM algorithms in the literature are rarely presented in assembly language but given in some kind of high-level pseudocode with an assumed understanding of how they can be translated to the instruction set of the underlying machine. Unlike the previous chapters, this chapter is informal. We will demonstrate by example how PRAM algorithms written in pseudocode for human consumption can conveniently be directly formalised in our high-level language for the Database Machine (Section 2.4), using the ideas developed for the formal proofs. From there, a practical implementation for an SQL-accessible database or a platform like Spark can be obtained.

On a PRAM as defined in Definition 4.1.1 all processors share the same program, but each processor can take a different execution path depending on the input data. But that freedom is not really necessary. In fact, a SIMD model (Single Instruction stream – Multiple Data stream) according to Flynn's classification [34] is equivalent in power to our PRAM model [71]. The high-level pseudocode notation we chose in Section 4.4.1 to present PRAM algorithms is very close to SIMD in that we assume that all processors execute the same line of code at all times with the exception that in conditional blocks of code some processors may be executing no-ops. PRAM algorithms in the literature are usually presented in this manner.

Our definition of a PRAM has a two-dimensional memory layout: each of the unbounded number of processors has an unbounded number of registers and can access each other processor's registers. This definition was chosen because it nicely aligns with our goal of simulating a Relational Machine. By contrast, most publications on PRAM algorithms use a one-dimensional memory model where processors with a bounded number of local registers have access to a common shared unbounded memory. The Parallel Microcode Machine uses this memory model. The two ways of defining a PRAM, one-dimensional memory and two-dimensional memory, are equivalent in power [71].

In the following sections we will demonstrate by example how to translate algorithms for both kinds of memory models to a Database Machine. Section 5.1 introduces basic translation techniques for the two-dimensional memory model using PRAM algorithms from this thesis as examples. In Section 5.2 we take an algorithm from the PRAM literature that uses a one-dimensional memory model and give a fully worked translation to the Database Machine. Section 5.3 shows how to translate this to a practical implementation in SQL and Python.

5.1 Basic translation techniques

For the two-dimensional memory model of our PRAM, the fundamental idea is to use a table with one row per active processor that contains a unique processor ID and the local variables of that processor. Here is a brief overview of the translation techniques we will present in the following subsections.

Scalar variables. Some of our algorithms have all processors access a single variable in processor 0 only, most notably in control structures, ensuring that processors always execute the same line in lockstep. This gets translated to a scalar variable in the Database Machine (which is technically a single-row, single-column table, see Section 2.4.2).

Computation on local registers. A parallel computation that only uses local registers of each processor and assigns the result to another local register gets translated to a **map** operation.

Reading remote registers. Reading a register from a remote processor corresponds to accessing a row in the processor table different from the current one. This is translated to a **join** operation, joining on the index of the row to be accessed. Care must be taken not to join with nonexisting rows, otherwise simulated processors would vanish.

Writing to remote registers. This is the most complicated translation since it involves resolving multiple concurrent write operations and preserving values that are not overwritten. It is achieved by first creating a table of old values plus all desired updates and then executing **groupby** to reduce to one value per processor, similar to the technique used in Algorithm 29.

5.1.1 Scalar variables and local computation

As a first example, we will translate PRAM Algorithm 31 (parallel binary search), going through it line by line. The translated algorithm is presented in full as Algorithm 43 below.

The PRAM algorithm takes as input a packed array v with indicator e_v that is assumed to be sorted and a packed array x with indicator e_x and performs a parallel binary search, locating for each active x a row in the packed array v. In the corresponding Database

Machine algorithm, the input is represented as two tables. V contains pairs (i, v) and X contains pairs (i, x) such that the indices i are consecutive integers starting at 0.

Algorithm 31 first determines the row count of the input packed array v:

PRAM

$n[0] \leftarrow \text{GetRowCount}(e_v)$

This becomes a scalar variable in the Database Machine translation:

Database Machine

$n \leftarrow |V|$

A common pattern in PRAM algorithms is limiting execution to a subset of the available processors. Here, we execute the algorithm only on those processors that have a row of the packed array x.

PRAM

if $e_x \neq 0$ **then**
  ...
end if

In this case our Database Machine algorithm already has a table X with exactly one row for each processor to execute the conditional block. By simply working with table X, no additional code is needed for the conditional. We will see in Section 5.2 below an example of an algorithm that alternates between running code on two subsets of processors. This is implemented equally naturally by using two tables, one for each subset of processors. In more complicated cases, conditional expressions inside the multiset comprehension can be used to manipulate only part of the table and leave the rest unchanged.

Assignments to local variables are a simple case of local computation.

PRAM

$l \leftarrow 0$
$r \leftarrow n[0]$

They translate to a **map** operation, creating from the table X with tuples (i, x) a result table R with tuples (i, x, l, r) where l and r are the new local variables.

Database Machine

$R \leftarrow [(i, x, 0, n) \mid (i, x) \in X]$

A loop using a variable in processor 0 ...

PRAM

```
while n[0] > 1 do
  ...
  if PID = 0 then
    n ← ⌈n/2⌉
  end if
end while
```

... becomes a loop using a scalar variable.

Database Machine

```
while n > 1 do
  ...
  n ← ⌈n/2⌉
end while
```

The core of the binary search is a parallel local computation, followed by a conditional assignment based on a variable from another processor:

PRAM

```
m ← ⌊(l + r)/2⌋
if v[m] > x then
  r ← m
else
  l ← m
end if
```

Recall that this piece of PRAM code lives inside a conditional (**if** $e_x \neq 0$) so that it is executed by those processors that have a row of the packed array x. These correspond to the rows of table R in the translation. Table R is updated as follows to simulate this block of code: For each $(i, x, l, r) \in R$ we first compute $m \leftarrow \lfloor (l + r)/2 \rfloor$. Next, the PRAM algorithm accesses the remote variable $v[m]$. The PRAM array v is simulated as the database table V; reading the remote variable becomes a join $(m, v) \in V$. Now the new values of l and r can be computed by an expression using the C-style ternary conditional operator "(? :)": We have $l \leftarrow (v > x \,?\, l : m)$ and $r \leftarrow (v > x \,?\, m : r)$. All this is compactly expressed in our high-level database language as a single line.

Database Machine

$$R \leftarrow [(i, x, (v > x \,?\, l : m), (v > x \,?\, m : r)) \mid (i, x, l, r) \in R \\ , \; m \leftarrow \lfloor (l + r)/2 \rfloor, (m, v) \in V]$$

The full translation of Algorithm 31 is presented as Algorithm 43.

Algorithm 43 Translation of PRAM Algorithm 31 (parallel binary search) to a Database Machine

```
1: function PRAMBinarySearch(V, X)                    ▷ search X in sorted table V
2:   n ← |V|
3:   R ← [(i, x, 0, n) | (i, x) ∈ X]
4:   while n > 1 do
5:     R ← [(i, x, (v > x ? l : m), (v > x ? m : r)) | (i, x, l, r) ∈ R
                                                    , m ← ⌊(l + r)/2⌋, (m, v) ∈ V]
6:     n ← ⌈n/2⌉
7:   end while
8:   return [(i, l) | (i, ∗, l, ∗) ∈ R]
9: end function
```

5.1.2 Remote registers and conditionals

The next example, PRAM Algorithm 32 for prefix computation, illustrates a more complex interplay of conditional instructions and remote variable access. It also uses arrays of local variables.

The PRAM algorithm takes as input a sequence x of length n where the elements of x are stored across processors $0, \ldots, n-1$ and the variable n is stored in processor 0 (denoted as $n[0]$). The Database Machine receives the input as a single table X of pairs (j, x_j) for $j = 0, \ldots, n-1$. It does not require n as a separate input because the size of the input can simply be determined as $|X|$. The PRAM algorithm uses an array of sequences y_i where each sequence is stored in processors $0, \ldots, c_i - 1$. These are represented in the Database Machine as tables Y_i containing pairs $(j, y_{i,j})$ for $j = 0, \ldots, c_i - 1$, i.e. the first element of each pair, j, corresponds to the processor ID in the PRAM algorithm.

The control structure of the PRAM algorithm involves a variable i and an array of variables c_i in processor 0.

PRAM

```
y_0 ← x
c_0[0] ← n[0]
i[0] ← 0

while c_{i[0]}[0] > 1 do
  i[0] ← i[0] + 1
  c_{i[0]}[0] ← ⌊c_{i[0]−1}[0]/2⌋
  ...
end while

while i[0] > 0 do
  i[0] ← i[0] − 1
  ...
end while
```

The loops get translated as in the previous example, using a a scalar variable i and an array of scalar variables c_i in the Database Machine. (Recall from Section 2.4.2 that a

scalar variable in the Database Machine is a single-row 1-table and that arrays of tables are a notational convention for manipulating a larger table as defined in Section 2.4.5.)

Database Machine

$Y_0 \leftarrow X$
$c_0 \leftarrow |X|$
$i \leftarrow 0$
while $c_i > 1$ **do**
  $i \leftarrow i + 1$
  $c_i \leftarrow \lfloor c_{i-1}/2 \rfloor$
  ...
end while
while $i > 0$ **do**
  $i \leftarrow i - 1$
  ...
end while

Inside the first **while** loop of the PRAM algorithm, there is a block that uses a subset of the processors to create a new column $y_{i[0]}$:

PRAM

if $PID < c_{i[0]}[0]$ **then**
  $y_{i[0]} \leftarrow y_{i[0]-1}[2 \cdot PID] \oplus y_{i[0]-1}[2 \cdot PID + 1]$
end if

On the Database Machine, we translate this as creating a new table Y_i with c_i rows. To do this, we need the indices $j = 0, \ldots, c_i - 1$ to start with. In general, these could be generated on a Database Machine from scratch using Algorithm 3. In this case, however, we happen to have a table Y_{i-1} available containing a sufficient number of consecutive integers in the first column. By starting with $(j, *) \in Y_{i-1}, j < c_i$, we obtain the required indices j. Accessing the nonlocal PRAM variable $y_{i[0]-1}[2 \cdot PID]$ translates to the join $(2j, y_0) \in Y_{i-1}$, placing the result in y_0. Similarly, accessing $y_{i[0]-1}[2 \cdot PID + 1]$ translates to the join $(2j + 1, y_1) \in Y_{i-1}$, placing the result in y_1. Overall, the creation of table Y_i amounts to one line on the Database Machine, combining the two values y_0 and y_1 using the binary operation $\oplus$.

Database Machine

$Y_i \leftarrow [(j, y_0 \oplus y_1) \mid (j, *) \in Y_{i-1}, j < c_i, (2j, y_0) \in Y_{i-1}, (2j + 1, y_1) \in Y_{i-1}]$

The code block inside the second **while** loop is a little more involved:

PRAM

if $PID > 0$ && $PID < c_{i[0]}[0]$ **then**
  if $PID \bmod 2 = 0$ **then**
    $y_{i[0]} \leftarrow y_{i[0]+1}[PID/2 - 1] \oplus y_{i[0]}$
  else
    $y_{i[0]} \leftarrow y_{i[0]+1}[(PID - 1)/2]$
  end if
end if

This code assigns a new value to each row of the packed array y_i except for row 0. On the Database Machine, this translates to changing all but one row of the table Y_i. Since tables are immutable, we cannot just update some rows. Instead, we have to create a new version of the full table. We do this by forming the union of a table with the unchanged rows – in this case, only row 0 – with a table of updated rows. This technique can be applied in general when a conditional in a PRAM algorithm translates to changes in a subset of a table's rows. The unchanged part of Y_1 is $[(0, y) \mid (0, y) \notin Y_i]$, a single-row table with the tuple for row 0. The other expression starts with $(j, y) \notin Y_i, j > 0$ to access all the other rows.

The inner **if** of the PRAM algorithm selects between two different remote processors to be accessed, corresponding to two different rows in table Y_{i+1}. We therefore calculate, using the ternary conditional operator, as an auxiliary value j' the row number to join on and bring in the corresponding value y' of Y_{i+1} using the join $(j', y') \notin Y_{i+1}$. The result can then be expressed using a ternary conditional expression.

Database Machine

$$Y_i \leftarrow [(0, y) \mid (0, y) \notin Y_i] \uplus [(j, (j \bmod 2 = 0 \,?\, y' \oplus y : y')) \mid (j, y) \notin Y_i, j > 0 \;,\; j' \leftarrow (j \bmod 2 = 0 \,?\, j/2 - 1 : (j - 1)/2) \;,\; (j', y') \notin Y_{i+1}]$$

The complete translation of PRAM Algorithm 32 to a Database Machine is given as Algorithm 44. Note that this algorithm is different from and more complicated than the arguably more natural implementation of prefix computation on a Database Machine presented as Algorithm 7 in Section 2.5.7, but that it nevertheless has the same asymptotic complexity.

5.1.3 Exclusive writes to remote registers

The next example is the translation of the PRAM algorithm for computing a range table, Algorithm 34. It illustrates a special case of writing to remote registers where the algorithm guarantees exclusive writes to any register.

The PRAM algorithm takes as input a packed array A with indicator $A.e$ that is assumed to be sorted by a column $A.v$. It produces a packed array with indicator e and columns v, *start*, and *end*. Each processor that has a row of A first computes whether

Algorithm 44 Translation of PRAM Algorithm 32 (prefix computation) to a Database Machine

function PRAMPrefixes($\oplus, X$) ▷ tuples (i, x)
  $Y_0 \leftarrow X$
  $c_0 \leftarrow |X|$
  $i \leftarrow 0$
  while $c_i > 1$ **do**
    $i \leftarrow i + 1$
    $c_i \leftarrow \lfloor c_{i-1}/2 \rfloor$
    $Y_i \leftarrow [(j, y_0 \oplus y_1) \mid (j, *) \in Y_{i-1}, j < c_i, (2j, y_0) \in Y_{i-1}, (2j+1, y_1) \in Y_{i-1}]$
  end while
  while $i > 0$ **do**
    $i \leftarrow i - 1$
    $Y_i \leftarrow [(0, y) \mid (0, y) \in Y_i] \uplus [(j, (j \bmod 2 = 0 \,?\, y' \oplus y : y')) \mid (j, y) \in Y_i, j > 0, j' \leftarrow (j \bmod 2 = 0 \,?\, j/2 - 1 : (j-1)/2), (j', y') \in Y_{i+1}]$
  end while
  return Y_0 ▷ tuples (j, y) where $y = \bigoplus_{i=0}^{j} x_i$
end function

it is first and/or last in a range of identical values and stores the result in *first* and *last*, respectively.

PRAM

if $A.e \neq 0$ **then**
  $first \leftarrow (PID = 0 \;||\; A.v \neq A.v[PID - 1] \,?\, 1 : 0)$
  $last \leftarrow (A.e[PID + 1] = 0 \;||\; A.v \neq A.v[PID + 1] \,?\, 1 : 0)$
end if

The Database Machine algorithm takes as input a table A with tuples (i, v) where i is an index and column v is assumed to be sorted. Its output will be a table with tuples $(i, v, start, end)$ containing the range table. The computation of *first* and *last* is a straightforward translation using a join to access a remote row. For clarity, we present it in two steps, building in table B a copy of A augmented by the two columns *first* and *last*. In the first line, the edge case of the first row is handled by a conditional expression for the index to join on, joining on the dummy index 0 for the first row. In the second line, the edge case of the last row of the table is handled by using an outer join (recall from page 27 that the notation $\in_\sigma$ indicates a left outer join (**ljoin**), setting $\sigma = 1$ if a row was matched and $\sigma = 0$ for unmatched rows).

Database Machine

$B \leftarrow [(i, v, (i = 0 \;||\; v \neq v' \,?\, 1 : 0)) \mid (i, v) \in A, ((i > 0 \,?\, i - 1 : 0), v') \in A]$
$B \leftarrow [(i, v, first, (\sigma = 0 \;||\; v \neq v' \,?\, 1 : 0)) \mid (i, v, first) \in B, (i + 1, v') \in_\sigma A]$

Next, prefix sums on *first* and *last* are computed.

PRAM

$n[0] \leftarrow \text{GetRowCount}(A.e)$
$i_{first} \leftarrow \text{ComputePrefixes}(first, n[0], +)$
$i_{last} \leftarrow \text{ComputePrefixes}(last, n[0], +)$

On the Database Machine this can be done using Algorithm 44 from the previous example or, alternatively, using the "native" Database Machine algorithm from Section 2.5.7.

Database Machine

$F \leftarrow \text{PRAMPrefixes}(+, [(i, first) \mid (i, *, first, *) \in B])$
$L \leftarrow \text{PRAMPrefixes}(+, [(i, last) \mid (i, *, *, last) \in B])$

The last step is the creation of the final range table. On the PRAM, this is written to columns v, *start* and *end* with indicator e. Note that only the processors with $first \neq 0$ write to index $i_{first} - 1$ and only the processors with $last \neq 0$ write to index $i_{last} - 1$, ensuring that each row of the final table is written by exactly one processor.

PRAM

if $first \neq 0$ **then**
  $e[i_{first} - 1] \leftarrow 1$
  $v[i_{first} - 1] \leftarrow A.v$
  $start[i_{first} - 1] \leftarrow PID$
end if
if $last \neq 0$ **then**
  $end[i_{last} - 1] \leftarrow PID$
end if

On the Database Machine, the result table R has tuples $(j, v, start, end)$, corresponding to the variables v, *start*, and *end* stored in PRAM processor i. It is generated in two steps, corresponding to the two **if** blocks. The first step generates a table with tuples $(j, v, start)$ where j is the new index in the output table. The second step adds column *end*. In the first step, each row of B with $first \neq 0$ generates a row of R. The indices are looked up in the prefix sum table F. Since the PRAM algorithm guarantees exclusive writes, we know that all indices $j = f - 1$ are unique within the result and no resolution of write conflicts is necessary.

In the next step, column *end* is added to table R. Each row of B with $last \neq 0$ contributes a value *end* and the matching previously computed row of R is brought in via a join to form the final result.

Database Machine

$R \leftarrow [(f - 1, v, i) \mid (i, v, first, *) \in B, first \neq 0, (i, f) \in F]$
$R \leftarrow [(l - 1, v, start, i) \mid (i, *, *, last) \in B, last \neq 0, (i, l) \in L, (l - 1, v, start) \in R]$

The complete translation of PRAM Algorithm 34 is shown as Algorithm 45.

Algorithm 45 Translation of PRAM Algorithm 34 (ComputeRangeTable) to a Database Machine

1: **function** PRAMRangeTable(A) ▷ tuples (i, v), sorted by v
2: $B \leftarrow [(i, v, (i = 0 \mid\mid v \neq v' \,?\, 1 : 0)) \mid (i, v) \in A, ((i > 0 \,?\, i - 1 : 0), v') \in A]$
3: $B \leftarrow [(i, v, \mathit{first}, (\sigma = 0 \mid\mid v \neq v' \,?\, 1 : 0)) \mid (i, v, \mathit{first}) \in B, (i + 1, v') \in_\sigma A]$
4: $F \leftarrow$ PRAMPrefixes$(+, [(i, \mathit{first}) \mid (i, *, \mathit{first}, *) \in B])$
5: $L \leftarrow$ PRAMPrefixes$(+, [(i, \mathit{last}) \mid (i, *, *, \mathit{last}) \in B])$
6: $R \leftarrow [(f - 1, v, i) \mid (i, v, \mathit{first}, *) \in B, \mathit{first} \neq 0, (i, f) \in F]$
7: $R \leftarrow [(l - 1, v, \mathit{start}, i) \mid (i, *, *, \mathit{last}) \in B, \mathit{last} \neq 0, (i, l) \in L, (l - 1, v, \mathit{start}) \in R]$
8: **return** R ▷ tuples $(j, v, \mathit{start}, \mathit{end})$
9: **end function**

5.1.4 Concurrent writes

To demonstrate the translation of concurrent writes to remote registers, we will use a contrived example of a tiny part of a PRAM algorithm. A real-world example will follow in Section 5.2. Assume the memory of a PRAM contains a packed array with indicator column e and columns v and w and also another column named r.

Consider the following PRAM code:

PRAM

```
if e ≠ 0 then
  r[v] ← w
end if
```

If multiple rows of the packed array contain the same value of v, this is a concurrent write operation. According to Definition 4.1.1, in this case the processor with the lowest ID succeeds and the other writes are ignored.

In the Database Machine translation, assume that we chose to store the packed array in a table E with tuples (pid, v, w) and the column r in a separate table V with tuples (pid, r) where in both cases pid corresponds to the processor ID of the PRAM. As seen in the previous examples, it is often convenient to represent packed arrays as tables containing only those rows for which the value of the indicator is 1.

The PRAM write operation translates to the following steps to update table R on the Database Machine.

1. Compute a table U of updates to be made to R. U contains triples (i, c, pid, w) where i is the processor ID to be written, $c = 0$, pid is the process ID of the writing processor and w is the value to be written.

2. Augment table U by the current contents of R, setting the second tuple element to $c = 1$ and the third element to $pid = 0$.

3. Compute the new table R by reducing U using **groupby** to a single row per processor i. The aggregate function $\min((c, pid, w))$ runs over triples (c, pid, w) and computes the minimum according to lexicographic order. This has the effect that updates $(c = 0)$ take precedence over the original values $(c = 1)$ and that multiple updates to the same location are resolved to the minimum processor ID.

Database Machine

$U \leftarrow [(v, 0, pid, w) \mid (pid, v, w) \Mapsto E]$
$U \leftarrow U \uplus [(pid, 1, 0, r) \mid (pid, r) \Mapsto R]$
$R \leftarrow [(i, w') \mid (i, c, pid, w) \Mapsto U, \mathbf{groupby}(i, (c', pid', w') \leftarrow \min((c, pid, w))]$

Note that step 2 is only necessary if we care about the previous content of column r in the PRAM (correspondingly, the previous contents of R). If it were omitted, the result R would contain only rows (pid, r) for which pid occurs among the values of v in the packed array.

5.2 A fully worked example from the literature

The previous examples have illustrated various translation techniques using PRAM algorithms from this thesis and a two-dimensional memory model. We will now turn to the one-dimensional memory model more commonly found in the PRAM literature and conclude this chapter with translating a more elaborate algorithm to a Big Data-practical in-database implementation.

Let $G = (V, E)$ be an undirected graph with $n = |V|$ vertices and $m = |E|$ edges where each edge is an unordered pair of vertices. The problem of computing the connected components of G is to assign to each vertex $v \in V$ a representative $r(v) \in V$ such that for any two vertices $v, w \in V$ we have $r(v) = r(w)$ if and only if there is a path from v to w. The best known parallel algorithm for this problem was given by Shiloach and Vishkin [85]. It uses $n + 2m$ processors and takes time $\mathrm{O}(\log n)$.

We are going to present this algorithm in the exact original notation used in [85] and translate it line by line to an algorithm for a Database Machine. To simplify the description, the authors assume that $V = \{1, \ldots, n\}$. During the whole algorithm each vertex i has a pointer $D(i)$ through which it points to another vertex or to itself. When the algorithm terminates, this array of pointers contains the result of the computation. The notation $D_s(i)$ indicates the value of $D(i)$ after iteration s of the algorithm. In addition there is an auxiliary vector Q of length n; the elements are written as $Q(i)$.

The algorithm uses one processor per vertex and two processors per edge. The variable i denotes the processor identifier and processors $i = 1, \ldots, n$ are designated the vertex processors. The remaining processors are allocated to the edges such that each edge $\{i_1, i_2\}$ for $i_1, i_2 \in V$ is assigned two processors identified by the ordered pairs (i_1, i_2) and (i_2, i_1). The order of the edges is immaterial. The algorithm is described in multiple steps. Each step starts with either the condition **if** $i \leq n$, indicating that it is executed on the vertex processors, or the condition **if** $i > n$, indicating that it is to be executed on the edge processors.

Note that the algorithm assumes a one-dimensional memory model. The only processor-local variables are the processor IDs i, i_1, and i_2. All other variables are global and shared.

In our database implementation, we assume that the input is presented as a table V of vertex IDs and a table E of edges where each edge appears in both directions, i.e. for

an edge $\{v, w\}$, table E contains the pairs (v, w) and (w, v). Executing a block of code only on the vertex processors corresponds to manipulating a database table with one row per vertex and executing something on the edge processors corresponds to using the edge table as a starting point.

The PRAM algorithm comprises an initialisation and a loop that repeatedly executes five steps. We translate each step in turn.

Initialisation is carried out by the vertex processors as indicated by the condition $i \leq n$:

PRAM

if $i \leq n$ **then**
  $D_0(i) \leftarrow i$
  $Q(i) \leftarrow 0$
  $s \leftarrow 1$
  $s' \leftarrow 1$
end if

In the database implementation, we store the arrays D_0 and Q in tables of the same name where each row is a pair of a vertex ID and the corresponding value.

Database Machine

$D_0 \leftarrow [(i, i) \mid i \in V]$
$Q \leftarrow [(i, 0) \mid i \in V]$
$s \leftarrow 1$
$s' \leftarrow 1$

The rest of the steps are executed in a loop.

PRAM

while $s = s'$ **do**

Database Machine

while $s = s'$ **do**

Step 1 is executed on the vertex processors as indicated by the condition $i \leq n$:

PRAM

if $i \leq n$ **then**
  $D_s(i) \leftarrow D_{s-1}(D_{s-1}(i))$
  if $D_s(i) \neq D_{s-1}(i)$ **then**
    $Q(D_s(i)) \leftarrow s$
  end if
end if

To simulate this, we use a table with one row per vertex. In the first assignment, table D_{s-1} is used a starting point to get the PRAM variable $D_{s-1}(i)$ and then joined with another copy to access the PRAM variable $D_{s-1}(D_{s-1}(i))$:

Database Machine

$D_s \leftarrow [(i, d') \mid (i, d) \Uparrow D_{s-1}, (d, d') \Uparrow D_{s-1}]$

The conditional assignment is the first example where concurrent writes to the same memory cell can occur. This is simulated in three steps as shown in Section 5.1.4.

1. Compute a table U of updates to be made to Q. This table contains triples (i, c, q) where i is the index to be written, i.e. the vertex ID, $c = 0$ and q is the value to be written. Since this is computed by vertex processors on the PRAM, a table with one row per vertex is used as a starting point, in this case D_s.

2. Augment table U by the current contents of Q, setting the middle tuple element to $c = 1$.

3. Compute the new table Q by reducing U using **groupby** to a single row per vertex i. The aggregate function $\min((c, q))$ runs over pairs (c, q) and computes the minimum according to lexicographic order. This has the effect that updates ($c = 0$) take precedence over the original values ($c = 1$) and that multiple updates to the same location are resolved to the minimum value. Note that in this algorithm all processors write the same value so that it does not matter which processor succeeds in writing it. We have therefore omitted the processor ID from the update table.

Database Machine

$U \leftarrow [(d, 0, s) \mid (i, d) \Uparrow D_s, (i, d') \Uparrow D_{s-1}, d \neq d']$
$U \leftarrow U \uplus [(i, 1, q) \mid (i, q) \Uparrow Q]$
$Q \leftarrow [(i, q')) \mid (i, c, q) \Uparrow U, \mathbf{groupby}(i, (c', q') \leftarrow \min((c, q)))]$

Step 2 is executed on the edge processors. In the original algorithm, the following is executed in parallel for each edge (i_1, i_2):

PRAM

if $i > n$ **then**
  if $D_s(i_1) = D_{s-1}(i_1)$ **then**
    if $D_s(i_2) < D_s(i_1)$ **then**
      $D_s(D_s(i_1)) \leftarrow D_s(i_2)$
      $Q(D_s(i_2)) \leftarrow s$
    end if
  end if
end if

We split this into an update of table D_s and an update of table Q, using the same 3-step technique for preserving original values of these tables as shown above. Running something in parallel on the edge processors corresponds to computing the updates U starting with the edge table E. The following computes the updates for D_s. Note that the check for equality $D_s(i_1) = D_{s-1}(i_1)$ is realised by joining D_s and D_{s-1} on d_1.

Database Machine

$$U \leftarrow [(d_1, 0, d_2) \mid (i_1, i_2) \in E, (i_1, d_1) \in D_s, (i_1, d_1) \in D_{s-1} \\ , (i_2, d_2) \in D_s, d_2 < d_1]$$
$$U \leftarrow U \uplus [(i, 1, d) \mid (i, d) \in D_s]$$

Then we compute the updates for Q in the same manner. Note that the updates for Q have to be computed before updating D_s since the conditional expressions depend on D_s.

Database Machine

$$U' \leftarrow [(d_2, 0, s) \mid (i_1, i_2) \in E, (i_1, d_1) \in D_s, (i_1, d_1) \in D_{s-1} \\ , (i_2, d_2) \in D_s, d_2 < d_1]$$
$$U' \leftarrow U' \uplus [(i, 1, q) \mid (i, q) \in Q]$$

Finally, we apply the updates to D_s and Q.

Database Machine

$$D_s \leftarrow [(i, d') \mid (i, c, d) \in U, \mathbf{groupby}(i, (c', d') \leftarrow \min((c, d)))]$$
$$Q \leftarrow [(i, q') \mid (i, c, q) \in U', \mathbf{groupby}(i, (c', q') \leftarrow \min((c, q)))]$$

Step 3 is executed on the edge processors. In the original algorithm, the following is executed in parallel for each edge (i_1, i_2):

PRAM

```
if i > n then
  if D_s(i_1) = D_s(D_s(i_1)) and Q(D_s(i_1)) < s then
    if D_s(i_1) ≠ D_s(i_2) then
      D_s(D_s(i_1)) ← D_s(i_2)
    end if
  end if
end if
```

This translates as

Database Machine

$$U \leftarrow [(d_1, 0, d_2) \mid (i_1, i_2) \in E, (i_1, d_1) \in D_s, (d_1, d_1) \in D_s \\ , (d_1, q) \in Q, q < s \\ , (i_2, d_2) \in D_s, d_1 \neq d_2]$$
$$U \leftarrow U \in [(i, 1, d) \in (i, d) \in D_s]$$
$$D_s \leftarrow [(i, d') \mid (i, c, d) \in U, \mathbf{groupby}(i, (c', d') \leftarrow \min((c, d)))]$$

Step 4 is another step of pointer jumping, executed by the vertex processors:

PRAM

```
if i ≤ n then
  D_s(i) ← D_s(D_s(i))
end if
```

Database Machine

$D_s \leftarrow [(i, d') \mid (i, d) \in D_s, (d, d') \in D_s]$

Step 5 is the bookkeeping for the **while** loop. Each vertex processor i checks its value $Q(i)$ and increments s' based on the result. Note that if multiple processors increment s' during this step, they do so simultaneously and they read s' before writing the new value to s'. That implies that s' is incremented exactly once if any number of vertex processors decide to increment it and remains unchanged otherwise.

PRAM

if $i \leq n$ **and** $Q(i) = s$ **then**
  $s' \leftarrow s' + 1$
end if
$s \leftarrow s + 1$

Since s' is a scalar value that is either incremented or not, we do not need the full write conflict resolution using **groupby**. Instead, we compute a table U that has a row for each vertex processor that decides to increment s' and then increment s' if U is nonempty.

Database Machine

$U \leftarrow [() \mid (i, s) \in Q]$
if $|U| \neq 0$ **then**
  $s' \leftarrow s' + 1$
end if
$s \leftarrow s + 1$

This concludes our translation of the Shiloach–Vishkin algorithm for graph connectivity to a Database Machine. The complete translated algorithm is given as Algorithm 46.

With this translation to a Database Machine we have stayed as close as possible to the original notation in order to illustrate the general principle. In particular, we have used separate tables D_s for each step s as introduced in Section 2.4.5. This is, however, not necessary. At each step, the new table D_s is computed using only the previous table D_{s-1} and all tables from earlier iterations can be discarded. Accordingly, the algorithm can easily be modified to just use two tables for D_s and D_{s-1}.

Algorithm 46 Translation of the Shiloach–Vishkin PRAM algorithm for graph connectivity to a Database Machine

1: **function** ConnectedComponents(V, E)
2: $D_0 \leftarrow [(i, i) \mid i \in V]$
3: $Q \leftarrow [(i, 0) \mid i \in V]$
4: $s \leftarrow 1$
5: $s' \leftarrow 1$

6: **while** $s = s'$ **do**
7: $D_s \leftarrow [(i, d') \mid (i, d) \in D_{s-1}, (d, d') \in D_{s-1}]$ ▷ Step 1
8: $U \leftarrow [(d, 0, s) \mid (i, d) \in D_s, (i, d') \in D_{s-1}, d \neq d']$
9: $U \leftarrow U \uplus [(i, 1, q) \mid (i, q) \in Q]$
10: $Q \leftarrow [(i, q')) \mid (i, c, q) \in U, \mathbf{groupby}(i, (c', q') \leftarrow \min((c, q)))]$

11: $U \leftarrow [(d_1, 0, d_2) \mid (i_1, i_2) \in E, (i_1, d_1) \in D_s, (i_1, d_1) \in D_{s-1}$ ▷ Step 2
$, (i_2, d_2) \in D_s, d_2 < d_1]$
12: $U \leftarrow U \uplus [(i, 1, d) \mid (i, d) \in D_s]$
13: $U' \leftarrow [(d_2, 0, s) \mid (i_1, i_2) \in E, (i_1, d_1) \in D_s, (i_1, d_1) \in D_{s-1}$
$, (i_2, d_2) \in D_s, d_2 < d_1]$
14: $U' \leftarrow U' \uplus [(i, 1, q) \mid (i, q) \in Q]$
15: $D_s \leftarrow [(i, d') \mid (i, c, d) \in U, \mathbf{groupby}(i, (c', d') \leftarrow \min((c, d)))]$
16: $Q \leftarrow [(i, q') \mid (i, c, q) \in U', \mathbf{groupby}(i, (c', q') \leftarrow \min((c, q)))]$

17: $U \leftarrow [(d_1, 0, d_2) \mid (i_1, i_2) \in E, (i_1, d_1) \in D_s, (d_1, d_1) \in D_s$ ▷ Step 3
$, (d_1, q) \in Q, q < s$
$, (i_2, d_2) \in D_s, d_1 \neq d_2]$
18: $U \leftarrow U \notin [(i, 1, d) \notin (i, d) \in D_s]$
19: $D_s \leftarrow [(i, d') \mid (i, c, d) \in U, \mathbf{groupby}(i, (c', d') \leftarrow \min((c, d)))]$

20: $D_s \leftarrow [(i, d') \mid (i, d) \in D_s, (d, d') \in D_s]$ ▷ Step 4

21: $U \leftarrow [() \mid (i, s) \in Q]$ ▷ Step 5
22: **if** $|U| \neq 0$ **then**
23: $s' \leftarrow s' + 1$
24: **end if**
25: $s \leftarrow s + 1$
26: **end while**
27: **return** $D_{s'}$
28: **end function**

5.3 Converting to SQL/Python

In this section we continue with the example from the previous section and demonstrate the last step of the conversion process, namely translating Algorithm 46 to SQL and Python. The code is given at the end of this section. It has been stripped of the surrounding infrastructure for setting up a connection to the database. `db.query()` executes an SQL query and returns the number of rows generated.

Algorithm 46 takes as input a vertex table V and an edge table E. Our database implementation assumes that the input is just an edge list in a table whose name is passed in the variable `dataset`. The input table is assumed to have two columns `i1` and `i2`, each containing a vertex ID. A preprocessing step creates an edge table "`edges`" where each edge is stored in both orientations. The `distributed by` clause is a hint for our research database Apache HAWQ and can be ignored.

Python/SQL

```
db.query("""\
create table edges as
    select i1, i2 from {0}
    union all
    select i2, i1 from {0}
    distributed by (i1);
""".format(dataset))
```

The vertex table V is used only in the initialisation.

Database Machine

$D_0 \leftarrow [(i,i) \mid i \in V]$
$Q \leftarrow [(i,0) \mid i \in V]$

Our SQL implementation deduces it from the edge list by computing the distinct values of `i1`. The second query uses the result of the first one as a shortcut.

Python/SQL

```
db.query("""\
create table tbl_d as
    select distinct i1 as i, i1 as d from edges
    distributed by (i);
""")

db.query("""\
create table tbl_q as
    select i, 0 as q from tbl_d
    distributed by (i);
""")
```

Recall that each iteration of the loop in Algorithm 46 only uses D_{s-1} and D_s; no previous D_i are needed. Our SQL implementation therefore uses only two tables: `tbl_d` corresponds to D_{s-1} and `tbl_d2` corresponds to D_s. Both have columns `i` and `d`. Table `tbl_q` plays the role of Q with columns `i` and `q`. We implement the **while** loop by checking the exit condition at the end and use a Python variable `s` for the scalar variable s.

Step 1 starts with a "shortcutting" operation.

Database Machine

$D_s \leftarrow [(i, d') \mid (i, d) \in D_{s-1}, (d, d') \in D_{s-1}]$ ▷ Step 1

SQL table `tbl_d2` is computed to represent table D_s of the Database Machine.

Python/SQL

```
db.query("""\
create table tbl_d2 as
      select a.i as i, b.d as d
      from tbl_d as a, tbl_d as b
      where a.d = b.i
      distributed by (i);
""")
```

Next, the update table U is computed.

Database Machine

$U \leftarrow [(d, 0, s) \mid (i, d) \in D_s, (i, d') \in D_{s-1}, d \neq d']$
$U \leftarrow U \uplus [(i, 1, q) \mid (i, q) \in Q]$

We compute the SQL table `tbl_u` representing U by a single query, using an SQL array to group c and q into a single column `p`. Note how the Python variable `s` is inserted into the query using Python string formatting. After computing the updates, `tbl_q` is no longer needed and therefore dropped.

Python/SQL

```
db.query("""\
create table tbl_u as
    select a.d as i, array[0, {}] as p
    from tbl_d2 as a
         join tbl_d as b on (a.i=b.i)
    where a.d != b.d
  union all
    select i, array[1, q] as p from tbl_q
distributed by (i);
""".format(s))

db.query("drop table tbl_q;")
```

Recall that in the Database Machine algorithm, the aggregate function min(), when applied to a pair (c, q), is assumed to return the minimum according to lexicographic order. This is used to resolve the updates to table Q.

Database Machine

$$Q \leftarrow [(i, q')) \mid (i, c, q) \in U, \mathbf{groupby}(i, (c', q') \leftarrow \min((c, q)))]$$

In SQL, the new table `tbl_q` is computed from `tbl_u`, using the fact that the SQL aggregate function `min()`, when applied to an array, assumes lexicographic ordering. Hence, the expression `(min(p))[2]` corresponds to q' in the Database Machine code.

Python/SQL

```
db.query("""\
create table tbl_q as
    select i, (min(p))[2] as q from tbl_u group by i
    distributed by (i);
""")

db.query("drop table tbl_u;")
```

The other steps are translated in the same way.

At the end of the loop of Algorithm 46, s' gets incremented if another iteration is required.

Database Machine

$U \leftarrow [() \mid (i, s) \in Q]$ ▷ Step 5
if $|U| \neq 0$ **then**
  $s' \leftarrow s' + 1$
end if

Since this is the only function of the variable s', we have eliminated it and just exit the loop if s' would not have been incremented, i.e. a table of updates to s' is empty. Recall that `db.query()` returns the number of rows returned by an SQL query.

Python/SQL

```
rowcount = db.query("""\
select i from tbl_q where q={};
""".format(s))

if rowcount==0:
    break
```

This concludes the translation. The resulting algorithm can be run on a distributed relational database. For an input of n rows, corresponding to a graph with n edges, it uses space $O(n)$ and executes $O(\log n)$ SQL queries.

This is currently the best known in-database algorithm in terms of formal complexity. However, its practical efficiency can be improved upon. We have developed a novel algorithm for computing connected components in a relational database, Randomised Contraction. It is a Las Vegas algorithm, always returning a correct result, but with a stochastic runtime. In expectation, Randomised Contraction achieves the same formal complexity as the Shiloach–Vishkin algorithm, but improves the practical performance. Our publication is included in this thesis as Appendix B.

The remainder of this section is the complete translation of Algorithm 46 to Python and SQL.

```
db.query("""\
create table edges as
    select i1, i2 from {0}
    union all
    select i2, i1 from {0}
    distributed by (i1);
""".format(dataset))

db.query("""\
create table tbl_d as
    select distinct i1 as i, i1 as d from edges
    distributed by (i);
""")

db.query("""\
create table tbl_q as
    select i, 0 as q from tbl_d
    distributed by (i);
""")

s = 1
while True:

    db.query("""\
    create table tbl_d2 as
          select a.i as i, b.d as d
          from tbl_d as a, tbl_d as b
          where a.d = b.i
          distributed by (i);
    """)

    db.query("""\
    create table tbl_u as
        select a.d as i, array[0, {}] as p
        from tbl_d2 as a
             join tbl_d as b on (a.i=b.i)
        where a.d != b.d
```

```
    union all
      select i, array[1, q] as p from tbl_q
distributed by (i);
""".format(s))

db.query("drop table tbl_q;")

db.query("""\
create table tbl_q as
    select i, (min(p))[2] as q from tbl_u group by i
    distributed by (i);
""")

db.query("drop table tbl_u;")

db.query("""\
create table tbl_u as
    select a.d as i, array[0, c.d] as p
    from edges as e
         join tbl_d2 as a on (e.i1=a.i)
         join tbl_d as b on (e.i1=b.i)
         join tbl_d2 as c on (e.i2=c.i)
    where a.d=b.d and c.d<a.d
  union all
    select i, array[1, d] as p from tbl_d2
distributed by (i);
""")

db.query("""\
create table tbl_u2 as
    select c.d as i, array[0, {}] as p
    from edges as e
         join tbl_d2 as a on (e.i1=a.i)
         join tbl_d as b on (e.i1=b.i)
         join tbl_d2 as c on (e.i2=c.i)
    where a.d=b.d and c.d<a.d
  union all
    select i, array[1, q] as p from tbl_q
distributed by (i);
""".format(s))

db.query("drop table tbl_d;")
db.query("drop table tbl_d2;")

db.query("""\
create table tbl_d2 as
    select i, (min(p))[2] as d from tbl_u group by i
    distributed by (i);
```

```
    """)

    db.query("drop table tbl_u;")
    db.query("drop table tbl_q;")

    db.query("""\
    create table tbl_q as
        select i, (min(p))[2] as q from tbl_u2 group by i
        distributed by (i);
    """)

    db.query("drop table tbl_u2;")

    db.query("""\
    create table tbl_u as
        select a.d as i, array[0, c.d] as p
        from edges as e
              join tbl_d2 as a on (e.i1=a.i)
              join tbl_d2 as b on (a.d=b.i)
              join tbl_d2 as c on (e.i2=c.i)
              join tbl_q as q on (a.d=q.i)
        where a.d=b.d and q.q<{} and a.d != c.d
      union all
        select i, array[1, d] as p from tbl_d2
    distributed by (i);
    """.format(s))

    db.query("drop table tbl_d2;")

    db.query("""\
    create table tbl_d2 as
        select i, (min(p))[2] as d from tbl_u group by i
        distributed by (i);
    """)

    db.query("drop table tbl_u;")

    db.query("""\
    create table tbl_d as
          select a.i as i, b.d as d
          from tbl_d2 as a, tbl_d2 as b
          where a.d = b.i
          distributed by (i);
    """)

    db.query("drop table tbl_d2;")

    s +=1
```

```
rowcount = db.query("""\
select i from tbl_q where q={};
""".format(s))

if rowcount==0:
    break
```

Chapter 6

Conclusions and future work

6.1 Conclusions

The world of Big Data moves fast, with new technology appearing almost every year. With the advent of each new platform, algorithms have to be refactored or reinvented to run on that particular platform. What has been missing is a high-level, platform-independent way to formulate and analyse algorithms for computations on Big Data.

The computational models developed in this thesis, the Database Machine and the Relational Machine, fill this gap, the former offering a more practical, richer set of operations and the latter trying to minimise functionality so as to be more conducive to mathematical reasoning. We have proved that the two models are equal in their computational power and have shown how to mechanically translate between them.

Algorithms written for the Database Machine can be run on many of the existing platforms for Big Data processing. In this way, the Database Machine acts as a bridge between algorithms for Big Data analytics and their concrete realisation in a distributed system. This work shows that different platforms for Big Data have more in common than is obvious at first sight, and it seems that, over time, platforms tend to gravitate towards a relational model of computation. This is exemplified by HAWQ, which added an SQL engine on top of Hadoop, or SparkSQL, which added SQL as a relational query language on top of Spark.

In a certain sense, the Database Machine is a model of communication. It does not make assumptions about the platform it is running on and is ignorant of the number of machines involved in the computation. Instead, by modelling only the data flow involved in a computation, it models the communication that is required to bring the required data elements together to perform operations on them.

In the Relational Machine, the line between communication and computation is drawn most clearly. The three "heavy" operations `JOIN`, `RANGE`, and `SINGLES` are the communications instructions in the sense that they require multiple rows, from one or two rtables to interact with one another. The "light" data-parallel mapping primitives are the instructions from which computation is built.

This focus on communication inherent in the algorithms makes both models relevant for the scenario of distributed or cloud computing on Big Data where local computation is much faster than inter-machine communication.

The most significant theoretical contribution of this thesis is establishing the connection between the Database Machine and the PRAM. When we published our paper on connected components [12] (Appendix B), this connection was unknown and it was thought that the assumptions of the PRAM, "which are idealisations of the parallelised computation set-up, do not accurately reflect the realities of parallel computing architectures, making its algorithms unrealistic to implement or not truly attaining the reported performance complexity bounds" (quoting ourselves).

It is now clear that the high abstraction level of the Database Machine, which makes it very powerful as a computational model, aligns nicely with the equally over-powered PRAM. The consequence is that, as demonstrated in Chapter 5, the research on the PRAM model from the 1980s can be viewed in a new light as a source for practical algorithms for today's platforms for Big Data analytics.

6.2 Future work

The Database Machine and the Relational Machine offer a new perspective on parallel computation. They form a bridge between algorithms for computations on Big Data and the systems on which these algorithms are executed, opening up further avenues for research, both practical and theoretical in nature.

6.2.1 Languages and programming models

It seems promising to look into compiling algorithms developed using different programming paradigms to the Database Machine. One such paradigm, often used implicitly in the description of parallel algorithm, is the PRAM. Chapter 5 gives an informal account of how to translate PRAM algorithms written in a high-level language and in a SIMD style to the Database Machine. This could be formalised and possibly implemented as a compiler.

We have also shown in Chapter 5 how a Database Machine can be systematically and mechanically translated to a combination of SQL and Python. It is equally straightforward to translate Database Machine algorithms to the other Big Data processing platforms mentioned in Section 1.3 and these translations could also be implemented as a compiler. In this way the Database Machine lends itself to be the *lingua franca* for formulating distributed algorithms on Big Data.

A different area for further research would be to investigate other popular theoretical parallel computation models and their power relative to the Database Machine. A good first candidate for such an exploration may be, as an example, message passing systems [77]: if entities in a message-passing system are modelled in the Database Machine as table rows, message-passing communication between them may be as straightforward to model as via a simple join operation.

6.2.2 More fine-grained modelling of operation costs

The Database Machine, as it is currently defined, makes a unit-cost assumption for all its operations. It may be worthwhile to consider other or more detailed cost models.

One possible direction is the Light Relational Machine presented in Appendix A. The Light Relational Machine focuses on modelling communication complexity by only charging for those instructions that are communication-intensive and assigning zero cost to the data-parallel mapping primitives. Using these assumptions, Theorem A.2.1 justifies assigning unit cost to the **map** operation of the Database Machine and Corollary A.2.2 suggests assigning cost $O(\log n)$ to the **group** operation.

6.2.3 Parallel database query optimisation

In current relational database management systems SQL queries are processed by first translating them to an internal representation based on the extended relational algebra. A query optimiser then tries to determine the most efficient way to compute the result [87, Chapters 15 and 16].

An expression can be transformed using equivalence rules within the extended relational algebra, leaving the result unchanged. For each basic operation of the relational algebra there are various choices of implementation. Which of these it best depends on the data at hand. Statistics on the data can be used to estimate join sizes, select sizes, and data distribution in order to inform the choice of algorithms.

Overall, there is a vast space of possible execution plans that can be explored for optimising a complex SQL query. An interesting research direction would be to look into SQL query optimisation in terms of the Relational Machine. The extended relational algebra could be compiled to the Database Machine, and we have shown in Chapter 3 that this in turn can be simulated by the Relational Machine.

Switching from the extended relational algebra to the Relational Machine could be like the shift from a CISC architecture in processor design to a RISC architecture where it has proven useful to have a simpler instruction set to allow for better code optimisation by compilers as well as a simpler hardware design.

6.2.4 Randomisation

A very promising area for future theoretical research is the introduction of randomness into the Database Machine and the Relational Machine. Randomness can increase the power of a computational model [15]. Probabilistic Turing Machines gave rise to the complexity class RP [41]: languages that can be recognised by such a probabilistic Turing Machine in polynomial time. A probabilistic Turing Machine has two transition functions and makes a random choice between them at each step. In other words, it generates one random bit per step.

The straightforward idea to add randomisation to the Relational Machine is an instruction that augments each row of an rtable by a random bit. We call this capability

strong randomisation. When simulating Turing Machines in parallel as described in Section 3.4.8, this affords the simulation of probabilistic Turing Machines, allowing mapping and aggregating using functions that are in RP instead of P. It is an open question whether P = RP.

There is, however, another natural way to introduce randomness, which is also analogous to the probabilistic Turing Machine, but on a different level. The Relational Machine can be viewed as a finite automaton with a transition function that for each state has a single next state, except for the conditional jump instructions, which have two possible follow-up states. Instead of introducing a second transition function and making a random choice between two transition functions, we can introduce a random jump instruction that takes the jump with 50% probability. This instruction effectively generates one random bit. We call this *thrifty randomisation*.

Strong randomisation provides much more entropy than thrifty randomisation. For an input rtable of n rows, a Big Data-practical algorithm using thrifty randomisation can only execute a polylogarithmic number of instructions and hence generate only a polylogarithmic number of random bits in total, whereas a single instruction using strong randomisation generates n random bits.

Our published Randomised Contraction algorithm (Appendix B) illustrates the two different forms of randomisation. The random reals method requires the generation of a random number per vertex of the input graph, i.e. strong randomisation. By coming up with the finite fields method, we were able to reduce this requirement to thrifty randomisation. This led to a measurable performance improvement and is the method we used for the published performance measurements.

It remains an open question whether randomisation adds to the power of the Relational Machine, and if so, whether the two levels of randomisation differ in power.

Appendix A

A communication-focussed model

The motivation for creating the computational models in this thesis was to model computation on Big Data in a distributed computing scenario. Our high-level model, the Database Machine, assigns unit cost to powerful basic operations like **map** and **group**. When simulated by our low-level Relational Machine, these two operations take polylogarithmic time. In the following section we introduce a variation of the Relational Machine that can simulate **map** in constant time and **group** in logarithmic time.

A.1 The Light Relational Machine

The instruction set of the Relational Machine allows us to clearly identify the operations that potentially require communication between different nodes.

`JOIN`, `RANGE`, and `SINGLES` are the only operations that require rows from different rtables or different rows from the same rtable to be brought together on the same node. All the mapping primitives can be executed in parallel and independently of each other on all nodes, assuming, of course, that a single row is not split among multiple nodes.

A distributed computing setting in which communication is much more expensive than computation motivates the following definition which effectively assigns zero cost to the data-parallel mapping primitives.

Definition A.1.1. A *Light Relational Machine* is a Relational Machine with the addition of a conditional jump instruction `JVZ` A that jumps to a different location if rtable register A contains the single-row $(0,1)$-rtable $[((),(0))]$; execution continues sequentially otherwise. The operations `JOIN`, `RANGE`, and `SINGLES` are called *heavy* operations. All other operations, including the new conditional jump, are *light*. Let $T\colon \mathbb{N}_0 \to \mathbb{N}_0$. A Light Relational Machine is said to compute in *time* $T(n)$ if it halts after executing at most $T(n)$ heavy instructions for any input of size n.

When presenting algorithms for the Light Relational Machine, we use the notation A.ValueZero() in control structures like **repeat** ... **until** to indicate the use of the `JVZ` instruction to test the condition that rtable A is the single-row rtable containing 0 as a value. This follows the convention introduced on page 46 of using method notation

"A.Method(...)" for macros that execute a constant number of light operations, aiding the reasoning about algorithms for the Light Relational Machine.

A consequence of having zero-cost mapping primitives is that database functions (see Definition 3.4.5) can be computed in constant time, provided an upper bound on the initial input size of the Relational Machine is available. This can either be supplied as part of the input or computed as a one-time setup step in logarithmic time.

Lemma A.1.1. *Let the input of a Light Relational Machine be an $(s, 0)$-rtable I with n rows. Algorithm 47 computes a $(0, 1)$-rtable $\hat{N}$ containing a single integer i such that $i \geq n$. The algorithm is frugal and takes time $\mathrm{O}(\log n)$.*

Algorithm 47 BoundInputSize(I): Compute rtable $\hat{N}$ with upper bound on input size

```
 1: procedure BoundInputSize(I[((x_1, ..., x_s), ())])
 2:     X ← I
 3:     N̂ ← ONE
 4:     N̂.Constant(i ← 1, zero ← 0)
 5:     unroll for j = 1, ..., s
 6:         while not AllZero(X, x_j) do
 7:             N̂.Shift(zero, i)                          ▷ i ← 2i
 8:             X.Copy(∗)
 9:             X.Push(temp)
10:             X.Shift(x_j, temp)
11:             X.Pop(temp)
12:             Range(X)
13:         end while
14:     end unroll for
15:     N̂.Pop(zero)
16:     return N̂[((), (i))]
17: end procedure
```

Proof. Algorithm 47 starts by copying the input to X and initialising the result rtable $\hat{N}$ with $i = 1$, which corresponds to a row width of zero. At each iteration of the **while** loop, it shifts a non-zero column x_j right by one bit and doubles i, reducing the row width of X by one. When the last loop terminates, we have $i = 2^r$ where r is the row width of the rtable. Since all key tuples of an rtable have to be different, the input I can contain at most $2^r = i$ rows, as claimed. The row width of $\hat{N}$ is $r + 1$.

All operations in the algorithm are frugal and take constant time (for AllZero(), see Lemma 3.4.4). Since the row width of the input is logarithmic in the input size by the definition of the Light Relational Machine, the number of iterations and hence the total time is $\mathrm{O}(\log n)$. □

A.2 Constant-time mapping

Theorem A.2.1. *Let $f : \mathbb{N}_0 \to \mathbb{N}_0$ be a database function. Let A be a labelled $(s, 0)$-rtable with key tuples $(k_1, \ldots, k_{s-1}, x)$ such that the sub-tuples $(k_1, \ldots, k_{s-1})$ are unique within*

the rtable. Then the result of mapping f over column x, i.e. replacing x by $f(x)$ in each row of rtable A, can be computed by a Light Relational Machine using linear space and constant time, provided that an upper bound on the initial input size of the machine is available.

Proof. The proof closely follows the proof of Theorem 3.4.6, modifying the construction to reduce the number of heavy operations so that it is bounded by a constant.

Let $\mathcal{M} = (\Gamma, Q, \delta)$ be the Turing Machine to compute the function. In order to simulate a one-sided infinite tape, we augment the tape alphabet by a start symbol "$\triangleright$" and simulate a Turing Machine $\mathcal{M}' = (\Gamma', Q, \delta')$ with $\Gamma' = \{\sqcup, \mathtt{0}, \mathtt{1}, \triangleright\}$ and

$$\delta'(q, \gamma) = \begin{cases} (q, \triangleright, \mathrm{R}) & \gamma = \triangleright \\ \delta(q, \gamma) & \text{otherwise.} \end{cases}$$

We start the simulation with a start symbol in a cell to the left of the head. The augmented transition function δ' simply moves the head one cell to the right whenever it encounters the start symbol, thereby preserving it and simulating a machine that cannot move its head off the left end of the tape. We further augment δ' and define that it is a no-op in state 0, i.e. $\delta'(0, \gamma) = (0, \gamma, \mathrm{S})$ for all $\gamma \in \Gamma'$.

The tape is encoded as a pair of integers (*left*, *right*) where *right* contains the symbols under and to the right of the head and *left* contains the symbols to the left of the head. Each symbol is encoded as a 2-bit integer according to Table 3.2 with the least significant bits containing the symbols closest to the head. The encoding has the order of the two bits reversed between *left* and *right* so that the head can simply be moved one cell by executing two shift instructions of the relational machine. Note that the blank symbol "$\sqcup$" is encoded as zero so that *right* can be interpreted as an infinite tape with all blank symbols to the right of any nonblank symbols.

So far, this is exactly the same construction we used in the proof of Theorem 3.4.6. We continue to follow the structure of that construction, but replace some of its steps by new algorithms that execute only light instructions. Algorithm 48 computes the database function mapping, but contrary to the previous construction it requires an extra table $\hat{N}$ which it assumes to store an upper bound of the initial input size of the Relational Machine. Such a table $\hat{N}$ can, for example, be computed by running Algorithm 47 beforehand.

Algorithm 48 computes the function in three stages:

- Encode the input column x as a tape (using Algorithm 49).
- Simulate Turing Machines (using Algorithm 50 to execute each step).
- Decode the output on the tape as an integer (using Algorithm 51).

Each stage comprises two nested loops. The outer loop uses heavy operations to check for termination. The inner loop uses only light operations and performs a fixed number of iterations determined by the pre-computed upper bound on the input size. Time on the Light Relational Machine is therefore determined by the number of iterations of the outer loops only.

Algorithm 48 ConstantTimeMap(A): Simulate a Turing Machine using O(1) heavy operations

```
1: procedure ConstantTimeMap(A[((k_1, ..., k_{s-1}, x), ())], N̂[((), i)])
2:     A.Copy(∗)
3:     A.Push(right)
4:     A.Inc(x)
5:     Range(A)
6:     repeat                                   ▷ encode x as a Turing Machine tape
7:         A.Copy(∗)
8:         I ← N̂
9:         repeat
10:            A.TMLightEncode(x, right)        ▷ see Algorithm 49
11:            I.Dec(i)
12:        until I.ValueZero()
13:        Range(A)
14:    until AllZero(A, x)
15:    A.Copy(k_1, ..., k_{s-1}, right)         ▷ omitting x
16:    A.Constant(q ← 1)                        ▷ initial state for TM is 1
17:    A.Constant(left ← 3)                     ▷ write encoded start symbol "▷" left of head
18:    Range(A)
19:    repeat                                   ▷ run Turing Machine
20:        A.Copy(∗)
21:        I ← N̂
22:        repeat
23:            A.TMLightStep(q, left, right)    ▷ see Algorithm 50
24:            I.Dec(i)
25:        until I.ValueZero()
26:        Range(A)
27:    until AllZero(A, q)
28:    A.Copy(k_1, ..., k_{s-1}, right)         ▷ omitting q, left
29:    A.Constant(x ← 1)                        ▷ leading 1 was dropped from x in encoding
30:    Range(A)
31:    repeat                                   ▷ decode Turing Machine tape as value x
32:        A.Copy(∗)
33:        I ← N̂
34:        repeat
35:            A.TMLightDecode(x, right)        ▷ see Algorithm 51
36:            I.Dec(i)
37:        until I.ValueZero()
38:        Range(A)
39:    until AllZero(A, right)
40:    A.Copy(k_1, ..., k_{s-1}, x)             ▷ omitting right
41:    A.Dec(x)                                 ▷ undo increment from encoding
42:    Range(A)
43:    return A[((k_1, ..., k_{s-1}, x), ())]
44: end procedure
```

The light step for the string encoding of the input is the same as in the proof of Theorem 3.4.6: A.TMLightEncode(x, *right*) (Algorithm 49) extracts one bit from x and writes the corresponding encoded symbol to the tape by shifting two bits into *right*, using conditional instructions to stop the shifting once all bits but the most significant one have been written.

Algorithm 49 A.TMLightEncode(x, *right*): Encode a bit on a simulated Turing Machine tape

1: A.Push(*bit*); A.Shift(x, *bit*)	▷ extract next bit
2: A.Constant($\overline{bit} \leftarrow 1$); A.CDec(*bit*, $\overline{bit}$)	▷ compute inverse
3: A.CShift(x, $\overline{bit}$, *right*); A.CShift(x, *bit*, *right*)	▷ write encoded bit
4: A.Pop(*bit*, $\overline{bit}$)	

The key difference to the construction used for Theorem 3.4.6 is the method to simulate executing one step of a Turing Machine without using a join to look up the value of the transition function in a table. This is implemented as Algorithm 50 and denoted as A.TMLightStep(q, *left*, *right*).

The algorithm first encodes the input to the transition function as a single integer by shifting two bits (i.e. one encoded symbol) from *right* into q and renaming it to *in*. What follows is an unrolled loop over all possible values of *in*, starting with 1.

Each unrolled block tests if $in = 1$, executes a sequence of instructions that manipulate the Turing Machine configuration only if $in = 1$ and decrements *in*. This ensures that out of all the possible manipulations of the Turing Machine configuration, exactly one is carried out, depending on the initial value of *in*.

Lines 5 through 9 create a variable $c = 0$, increment it if $in > 0$ and decrement it if $\lfloor in/2 \rfloor > 0$. This results in $c = 1$ if $in = 1$ and $c = 0$ otherwise; the value of *in* is preserved.

All instructions between line 11 and line 32 are conditioned on c, i.e. they change the configuration only if $in = 1$. Lines 11 through 18 write a new symbol to the tape by shifting two bits into *right*. Lines 19 through 24 move the head right or left. Lines 25 through 32 build the new state in q.

Overall, Algorithm 50 is a potentially very long but constant sequence of light instructions hard-coding the transition function of the Turing Machine to be simulated.

After Algorithm 48 has simulated enough steps for all Turing Machines to have reached the halting state 0, it remains to decode the result as an integer. This is a straightforward reversal of the encoding process. A.TMLightDecode(x, *right*) (Algorithm 51) extracts one symbol from the tape and shifts the corresponding bit into the result x, again using conditional instructions to stop the shifting once the tape is empty.

All three inner loop algorithms – A.TMLightEncode() as well as A.TMLightStep() and A.TMLightDecode() – have the property that superfluous invocations do not do any harm. The encoding and decoding steps use conditional instructions to stop when done and the Turing Machine's transition function is a no-op when the machine is in the halting state 0. The outer loops around these constructs check that encoding, Turing Machine execution, and decoding have finished in all rows, thereby guaranteeing the correct result.

Algorithm 50 A.TMLightStep(q, *left*, *right*): Simulate one step of a Turing Machine

```
1:  A.Shift(right, q); A.Shift(right, q)                ▷ read symbol under head
2:  A.Rename(in ← q)          ▷ this is now the input to the transition function
3:  A.Push(q)                                           ▷ prepare new state q = 0
4:  unroll for i = 1, ..., 4 · max Q + 3
5:    A.Push(c, temp)
6:    A.CInc(in, c)
7:    A.Shift(in, temp)                                 ▷ in ← ⌊in/2⌋
8:    A.CDec(in, c)
9:    A.Shift(temp, in)                                 ▷ restore in; temp = 0
10:   let (q', γ, m) = δ'(⌊i/4⌋, decode(i mod 4))       ▷ decode via Table 3.2 (left)
11:   A.CShift(c, temp, right)
12:   expand if γ = 0
13:     A.CInc(c, right)
14:   end expand if
15:   A.CShift(c, temp, right)
16:   expand if γ = 1
17:     A.CInc(c, right)
18:   end expand if
19:   expand if m = R
20:     A.CShift(c, right, left); A.CShift(c, right, left)   ▷ move head right
21:   end expand if
22:   expand if m = L
23:     A.CShift(c, left, right); A.CShift(c, left, right)   ▷ move head left
24:   end expand if
25:   expand if q' > 0
26:     unroll for j = ⌊log q'⌋ downto 0
27:       A.CShift(c, temp, q)
28:       expand if bit j of q' is set
29:         A.CInc(c, q)
30:       end expand if
31:     end unroll for
32:   end expand if
33:   A.Pop(c, temp)
34:   A.Dec(in)
35: end unroll for
36: A.Pop(in)
```

Algorithm 51 A.TMLightDecode(x, *right*): Decode a bit from a simulated Turing Machine tape

```
1:  A.CShift(right, right, x)                           ▷ extract bit to x
2:  A.Push(temp)
3:  A.CShift(right, right, temp)                        ▷ dispose of ¬bit
4:  A.Pop(temp)
```

It remains to prove that Algorithm 48 runs in constant time, i.e. the number of heavy operations is bounded by a constant. Recall that there is a constant number of heavy operations per iteration of the outer loops, so it suffices to show that the number of iterations of the outer loops is bounded by a constant.

The Turing Machine computing a database function is assumed to take time polynomial in its input size. This input size is bounded by the row width, which in turn is logarithmic in the initial input size n of the Light Relational Machine. Hence, there are constants c and k such that the number of steps required for the Turing Machine is at most $c \cdot \log^k n$ for large enough n. There exists an n_0 such that $n \geq c \cdot \log^k n$ for all $n \geq n_0$. Let $b \geq n$ be the bound pre-computed and stored in $\hat{N}$. The inner loop executes b steps. It follows that for all $n \geq n_0$ the execution of all Turing Machines finishes in one iteration of the outer loop. For $n < n_0$, more iterations may be required, but there are only finitely many cases and their maximum is a constant independent of n. Hence, the number of iterations of the outer loop is bounded by a constant, as claimed.

Since the encoding and decoding stages each require $\mathrm{O}(\log n)$ steps, the same argument proves that both of their outer loops execute $\mathrm{O}(1)$ iterations, proving the overall claim on time. □

Corollary A.2.2. *The Light Relational Machine can simulate the* **group** *operation of the Database Machine taking time logarithmic in the machine's input size.*

Proof. The proof is the same as for Lemma 3.5.10. It relies on Algorithm 19 and Algorithm 25, which are proved to take polylogarithmic time in Theorem 3.4.10 and Lemma 3.5.9, respectively. If we replace the Turing Machine simulations in these algorithms by the constant-time map of Theorem A.2.1, these algorithms take logarithmic time overall. □

Appendix B

The Randomised Contraction algorithm

In Section 5.2 we showed how to translate a PRAM algorithm for labelling the connected components of a graph to the Database Machine and from there to an implementation in SQL and Python that can be run in a distributed relational database.

This Appendix presents a novel algorithm for computing connected components in a relational database, Randomised Contraction. It is a Las Vegas algorithm, always returning a correct result, but with a stochastic runtime. It was published as a paper at the 2020 IEEE 36th International Conference on Data Engineering (ICDE) [12].

The original Randomised Contraction algorithm with the random reals method was developed by Michael Brand while he was employed at Pivotal Software, Inc., and is covered by U. S. Patents. Radu-Alexandru-Todor contributed the idea for the proof in Appendix B.B. My contributions are the encryption method and the finite fields method for randomisation, all experimental work, and 95% of the writing.

The finite fields method improved the practical performance of the algorithm, as explained in the paper. It also led to the discovery of a promising avenue for future research on relational computation as outlined in Section 6.2.4: thrifty randomisation. The two variants of the Randomised Contraction algorithm are examples for the two methods of adding randomisation to our computational model. The random reals method uses $O(n \log n)$ random bits per iteration by generating a random number for each vertex. The finite fields method requires only $O(\log n)$ random bits per iteration since it only uses two word-size random numbers.

Chronologically, this paper comes before Chapter 4 of this thesis. At the time of writing, we were aware of Shiloach and Vishkin's PRAM algorithm for connected components [85] but had not yet discovered how closely the word PRAM and the Database Machine are related. It turns out that translating the PRAM algorithm as described in Chapter 5 leads to an in-database implementation that deterministically achieves the same asymptotic complexity that the Randomised Contraction algorithm achieves stochastically.

In practice, however, the implementation presented in Section 5.3 appears to be slower than Randomised Contraction. This is to be expected because the total number of SQL

queries of the deterministic algorithm is much larger than that of the Randomised Contraction algorithm. We have therefore not pursued the deterministic algorithm any further after preliminary testing on some of the smaller data sets.

The remainder of this Appendix is the unmodified content of the published paper, reformatted to fit the page layout of this thesis. The two appendices of the paper appear as sub-appendices B.A and B.B.

B.1 Introduction

Connected component analysis [48], the assignment of a label to each vertex in a graph such that two vertices receive the same label if and only if they belong to the same connected component, is one of the tent-pole algorithms of graph analysis. Its wide use is in applications ranging from image processing (e.g., [52, 90, 69, 102], to name a few recent examples) to cyber-security (e.g., [40, 72, 47, 103]). The most well-known theoretical result regarding connectivity analysis is perhaps the Union/Find algorithm [49, 93, 24], ensuring that labels can be maintained per vertex in an amortised complexity on the order of the inverse Ackermann function per edge, which is the theoretical optimum.

In real-world settings, however, large graphs such as those analysed in Big Data data science are stored on distributed file systems and processed in distributed computing environments. These are ill-suited for the Union/Find algorithm. For example, Union/Find involves following long linked lists, which is inefficient if the items in these lists reside on different machines.

A widely used platform for Big Data processing is Hadoop with its distributed and redundant file system HDFS and the MapReduce framework for implementing distributed computation [64]. Another, more recent distributed computing framework is Apache Spark [50], building on Hadoop HDFS for data storage. These two have in common that algorithms have to be specifically designed for the respective framework.

However, most of the world's transactional business data is stored natively in large, relational, SQL-accessible databases, and is only treated as graph data in certain contexts. It is therefore beneficial to have an efficient solution for graph algorithms, and particularly for the connected components algorithm, within the framework of relational databases. Such a solution obviates the need for data duplication in a separate storage system and for supporting multiple data storage architectures. It also avoids the potential for data conflicts and other problems arising from performing data analysis in two disparate systems.

The present paper presents a new algorithm for connected components analysis, Randomised Contraction. It is practical for Big Data data analytics in the following respects:

In-database execution. Our algorithm uses SQL queries as its basic building blocks. It can therefore be natively executed in a relational database, and specifically within the framework of Massively Parallel Processing (MPP) databases [27] where the architecture is designed for efficient parallel processing.

Scalability. Randomised Contraction uses (for any input graph) an expected logarithmic number of queries, running over exponentially decreasing amounts of data. Our

empirical results obtained with an MPP database show it to smoothly scale out to Big Data, running, in total, in an amount of time quasi-linear in the input size.

Space efficiency. Typical database maintenance uses some bounded fraction of available space. Therefore, practical in-database algorithms for use on mass data should not create intermediate data that is more than linear in the size of the input. Our algorithm satisfies this criterion.

Our empirical results show that Randomised Contraction outperforms other leading connected components algorithms when implemented in an MPP database. Furthermore, our in-database implementation of one of the algorithms runs faster than the original Spark implementation and uses fewer resources, allowing it to scale up to larger datasets.

The paper is structured as follows: Section B.2 presents related work. In Section B.3, we describe the problem formally. In Section B.4, we discuss naive approaches to a solution and show where they fail. In Section B.5, we describe our new algorithm, Randomised Contraction, with several refinements, and in Section B.6 we analyse its theoretical performance. Section B.7 gives empirical results. A short conclusions section follows. Appendix B.A presents excerpts of the code used for experiments. Appendix B.B gives improved theoretical bounds on graph contraction that may be of independent interest.

B.2 Related work

Many researchers have long tried to optimise connected component finding for parallel computing environments (e.g., [45]). Most suited for this pursuit from a theoretical perspective is the theoretical framework of the Parallel Random Access Machine (PRAM) [33, 81]. PRAM algorithms for connected components finding were presented, e.g., in [85, 99, 9]. In [38], it was noted that randomised algorithms may have an advantage in this problem. The best result obtained by the randomised approach is [43], where a randomised EREW PRAM algorithm is presented that finds connected components of a graph $G = \langle V, E \rangle$ in $\mathrm{O}(\log |V|)$ time using an optimal number of $\mathrm{O}((|V| + |E|)/\log |V|)$ processors. Its result is always correct and the probability that it does not complete in $\mathrm{O}(\log |V|)$ time is at most n^{-c} for any $c > 0$.

However, as observed by Eppstein and Galil [29], the PRAM model is "commonly used by theoretical computer scientists but less often by builders of actual parallel machines". Its assumptions, which are idealisations of the parallelised computation set-up, do not accurately reflect the realities of parallel computing architectures, making its algorithms unrealistic to implement or not truly attaining the reported performance complexity bounds.

Indeed, the papers that explore connected components algorithms for large-scale practical architectures do so using decidedly different algorithms. The first MapReduce algorithms that run in a logarithmic number of rounds were proposed by Rastogi et al. [76]. Among several variations of new algorithms presented, they report the overall best practical performance for the Hash-to-Min algorithm. This algorithm, however, has a worst case

space usage of $O(|V|^2)$. The best known space usage of a MapReduce algorithm is linear in the input size and achieved by the Tho-Phase algorithm by Kiveris et al. [53]. This algorithm, however, takes $\Theta(\log^2 |V|)$ rounds. The Cracker algorithm proposed by Lulli et al. [58] is implemented in Spark and once again improves the number of rounds to $O(|V|)$, but it does so at the expense of increasing the communication cost to $O(\frac{|V|\cdot|E|}{\log |V|})$.

As outlined in the introduction, if the data to be analysed is already stored in a distributed relational database, it is beneficial to be able to run algorithms in-database instead of exporting data to a different platform for analysis. This led to the development of the open source machine learning library Apache MADlib [44]. This library implements, among a small set of other graph algorithms, a connected components algorithm using Breadth First Search. We show in section B.4 that its worst case behaviour makes it unsuitable for Big Data data science.

Our novel Randomised Contraction algorithm has an efficient implementation in an MPP database and achieves both the best time complexity and space complexity among the above mentioned algorithms. Like the PRAM algorithms of [38, 43], it is randomised. It is guaranteed to terminate and to do so with a correct answer, and for any given $\epsilon > 0$ guarantees to terminate after $O(\log |V|)$ SQL queries with probability at least $1 - \epsilon$, where $|V|$ is the number of vertices in the input graph. The algorithm's space requirements can be made linear deterministically, not merely in expectation, and it can be implemented to use temporary storage not exceeding four times the size of the input plus $O(|V|)$. This is at worst a five-fold blow-up, which is within the typical range for standard database operations.

B.3 Problem description

A graph $G = \langle V, E \rangle$ is typically stored in a relational database in the form of two tables. One stores the set of vertices V, represented by a column of unique vertex IDs and optionally more columns with additional vertex information. Another table stores the edge set E in two columns containing vertex IDs and optionally more columns with additional edge information. In the context of connected component analysis, graphs are taken to be undirected, so an (x, y) edge is considered identical to a (y, x) edge. For simplicity we present our algorithm such that its only input is an edge table containing two columns with vertex IDs from which the set of vertices is deduced. Isolated vertices can be represented in this table as "loop edges", (v, v), if necessary.

The output of the algorithm is a single table with two columns, v and r, containing one row per vertex. In each row v is a vertex ID and r is a label uniquely identifying the connected component the vertex belongs to. A correct output of the algorithm is one where any two vertices share the same r value if and only if they belong to the same connected component. Connected component analysis does not make any specific requirement regarding the values used to represent components other than that they are comparable.

B.4 Simple solution attempts

Perhaps the simplest approach to performing in-database connected components analysis is to begin by choosing for each vertex a representative by picking the vertex with the minimum ID among the vertex itself and all its neighbours, then to improve on that representative by taking the minimum ID among the *representatives* of the vertex itself and all its neighbours, and to continue in this fashion until no vertex changes its choice of representative. We refer to this naive approach as the "Breadth First Search" strategy: after n steps each vertex's representative is the vertex with the minimum ID among all vertices in the connected component that are at most at distance n from the original vertex.

Though the algorithm ultimately terminates and delivers the correct result, its worst-case runtime makes it unsuitable for Big Data. Consider, for example, the sequentially numbered path graph with IDs 1, 2, ..., n. For this graph, Breadth First Search will take $n - 1$ steps.

To remedy this, consider an algorithm that calculates G^2, i.e. the graph over the same vertex set as G whose set of edges includes, in addition to the original edges, also (x, z) for every x and z for which there exists a y such that both (x, y) and (y, z) are edges in G.

Calculating G^2 can be done easily in SQL by means of a self-join. A tempting possibility is therefore to repeat the self-join and calculate G^4, G^8, etc. Such a procedure would allow us to reach neighbourhoods of radius 2^n in only n steps.

Unfortunately, this second approach does not yield a workable algorithm, either. The reason for this is that in G^k each vertex is directly connected to its entire neighbourhood of radius k in G. For a single-component G, the result is ultimately the complete graph with $|V|^2$ edges. This is a quadratic blow-up in data size, which for Big Data analytics is unfeasible.

Our aim, in presenting a new algorithm, is therefore to enjoy the best of both worlds: we would like to be able to work in a number of operations logarithmic in the size of the graph, but to require only linear-size storage.

B.5 The new algorithm

We present our new algorithm for calculating connected components, **Randomised Contraction**, by starting with its basic idea and refining it in several steps.

B.5.1 The basic idea

Let $G = \langle V, E \rangle$ be a graph. The algorithm contracts the graph to a set of representative vertices, preserving connectivity, and repeats that process until only isolated vertices remain. These then represent the connected components of the original graph.

Denote by $N_G[v]$ the *closed neighbourhood* of a vertex v, i.e. the set of all vertices connected to v by an edge in E plus v itself. Let $G_0 = \langle V_0, E_0 \rangle$ be the original graph.

At step i, map every vertex v to a *representative* $r_i(v) \in N_{G_{i-1}}[v]$. The contracted graph $G_i = \langle V_i, E_i \rangle$ is then constructed as $V_i = \{r_i(v) \mid v \in V_{i-1}\}$ and $E_i = \{(r_i(v), r_i(w)) \mid$

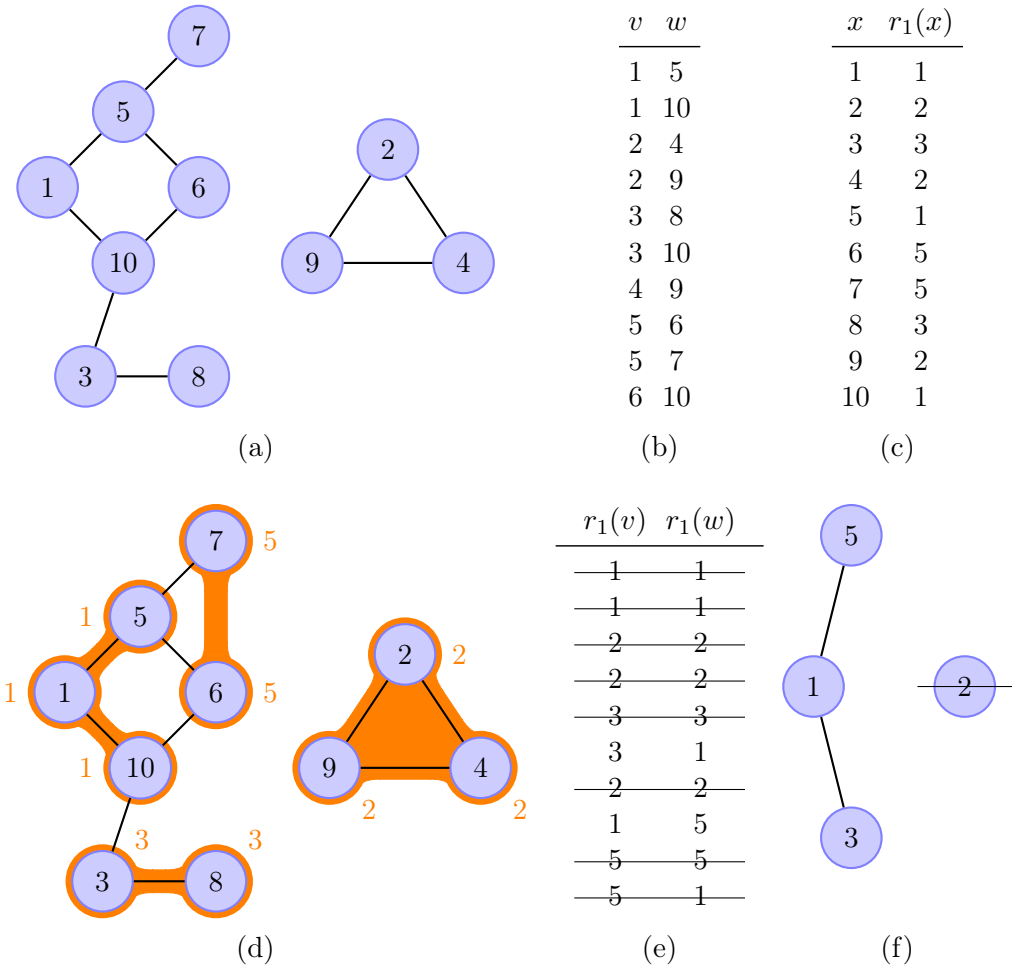


Figure B.1: (a) An undirected graph G_0 with vertex IDs shown inside the nodes. (b) The representation of G_0 as a list of edges. (c) The choice of representative $r_1(x)$ for each vertex x. (d) The graph with representative choices shown at the side of each node. Bubbles around the nodes indicate sets of vertices with the same choice of representative. These will be contracted to single vertices. (e) The edge list of the graph G_1 is computed by mapping the function r_1 over the edge list of G_0. Duplicates and loop edges, shown struck out, are eliminated. (f) The resulting graph G_1 after one contraction step. The isolated vertex 2, shown struck out, is excluded from further computation.

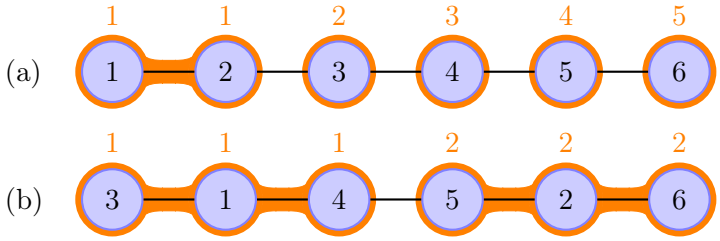


Figure B.2: (a) In a sequentially numbered path graph, every vertex but the first one will choose its left neighbour as a representative. This is the worst case: the contracted graph is only one vertex smaller. (b) If the same path graph is numbered optimally, it contracts to 1/3 the number of vertices.

$(v, w) \in E_{i-1}$ and $r_i(v) \neq r_i(w)\}$. Note that two vertices are connected in G_{i-1} if and only if their representatives are connected in G_i. In other words, for each connected component of G_{i-1} there is a corresponding connected component in G_i.

Repeat this contraction process until reaching a graph G_k that contains only isolated vertices. At that stage each of these represents one of the connected components of the original graph. Applying all the maps r_i in sequence maps each vertex to an identifier unique to its connected component: the composition of the representative functions $r_k \circ r_{k-1} \circ \cdots \circ r_1$ is the output of the algorithm.

Assuming the vertices are ordered, the basic idea for the choice of representatives is to set $r_i(v) = \min N_{G_{i-1}}[v]$. After each contraction step, isolated vertices can be excluded from further computation since each of them is known to form a connected component by itself. If the graph is only represented by its edge set, the removal of loop edges effectively eliminates isolated vertices. This leads to a natural termination condition: the algorithm terminates when the edge set becomes empty. Figure B.1 illustrates one contraction step using this idea. The graph (a) is represented as a list of edges (b). The edge list of the contracted graph (e) is obtained by mapping the representative function over all vertex IDs in this list, eliminating duplicates and eliminating loop edges.

B.5.2 Randomisation

The algorithm in the previous section still suffers from the same worst case as the Breadth First Search strategy described in Section B.4. Consider a sequentially numbered path graph on n vertices as shown in Fig. B.2(a). Each vertex except the first one will choose as its representative the neighbour preceding it. The result of contraction is a sequentially numbered path graph on $n - 1$ vertices. This implies that the algorithm takes $n - 1$ steps until the path is contracted to a single vertex. If, on the other hand, the path is labelled differently, it can contract to 1/3 of its vertices in the optimal case as shown in Fig. B.2(b).

A solution for avoiding worst case contraction is to randomise the order of the vertices. We show in Section B.6 that the graph will then, in expectation, shrink to at most a constant fraction γ of its vertices, with $\gamma < 1$. We further show that if the randomisation is performed independently at each step, this leads to an expected logarithmic number of steps. As a result, the algorithm behaves well for any input. By contrast, other algorithms that rely on a worst case being "unlikely" are vulnerable in an adversarial scenario where such a worst case can be exploited to an attacker's advantage.

We remark that vertex label randomisation, critical to our algorithm, would not have aided the simple solution attempts described in Section B.4. The complexity of Breadth First Search, for example, is bounded by the diameter of the analysed graph, regardless of how vertices are labelled.

B.5.3 Randomisation methods

In a practical implementation, choosing a random permutation of the vertices is itself a nontrivial task, especially in a distributed computing scenario such as an MPP database. One way to achieve this is the **random reals method**. At step i, generate for each vertex

v a random real $h_i(v)$ uniformly distributed in $[0, 1]$. The choice of the representative then becomes $r_i(v) = \arg\min_{w \in N_{G_{i-1}}[v]} h_i(w)$.

This method in theory achieves *full randomisation*, a uniform choice among all $|V|!$ possible orderings of the vertices, for which the best performance bounds can be proved (see Appendix B.B). The advantage of the random reals method over brute-force random permutation generation is that the table of random numbers can be created in parallel in a distributed database. A disadvantage is that this table has to be distributed to all machines in the cluster for picking representatives.

A more efficient idea is to pick a pseudo-random permutation by means of an encryption function on the domain of the vertex IDs. If the vertex IDs are 64-bit integers, a suitable choice is the Blowfish algorithm [83] which can be implemented in a database as a user-defined function. Let e_k denote an encryption function on the domain of the vertex IDs with key k. The **encryption method** then works as follows: at step i, choose a random key k_i. Let $r_i(v) = \arg\min_{w \in N_{G_{i-1}}[v]} e_{k_i}(w)$. Note that an encryption function is by definition a bijection which guarantees a unique choice of representatives.

The encryption method is more efficient than the random reals method in a distributed setting since it obviates the need to communicate one random number per vertex across the network to every node that needs it. Instead, only the encryption key needs to be distributed and each processor can compute the pseudo-random vertex IDs independently as necessary. This exploits the fact that in a realistic setting, communication across computation nodes is much slower than local computing.

While encryption functions are designed to be "as random as possible" and work well in practice, it is hard to rigorously prove for them the required graph contraction properties. Also, they are computationally expensive. We therefore present as the final refinement of the Randomised Contraction algorithm the **finite fields method**. Assume the domain of the vertex IDs is a finite field $\mathbb{F}$ with any ordering. To determine the representatives at step i, choose $0 \neq A_i \in \mathbb{F}$ and $B_i \in \mathbb{F}$ uniformly at random and let $r_i(v) = \arg\min_{w \in N_{G_{i-1}}[v]} h_i(w)$ where $h_i(w) = A_i \cdot w + B_i$ with multiplication and addition carried out using finite field arithmetic. Note that h_i is a bijection: in a field, every $A \neq 0$ has a unique multiplicative inverse A^{-1}. If $y = A \cdot x + B$, we have $x = A^{-1} \cdot (y - B)$.

If the vertex IDs are fixed-size integers with b bits, this data type can be treated as a finite field with 2^b elements by performing polynomial arithmetic modulo an irreducible polynomial [57, Thm. 3.2.6]. Note that while the calculation of $h_i(w)$ is performed in the finite field $\mathbb{F}$, the result is stored as an integer and the calculation of arg min is done with reference to integer ordering. Since finite field arithmetic over this field is awkward to implement in SQL, we wrote a fast implementation in C and loaded it as a user-defined function into the database. An SQL-only implementation could alternatively choose a prime number p known to be larger than any vertex ID and use normal integer arithmetic modulo p, giving the data type of the vertex IDs the structure of $\mathbb{F} = \mathrm{GF}(p)$.

```
procedure RandomisedContraction(G)
  create table E as
    select v, w from G union all select w, v from G;
  firstround ← true
  repeat
    choose 0 ≠ A ∈ 𝔽 and B ∈ 𝔽 uniformly at random
    create table R as                                  ▷ compute representatives
      select v, least(axb(A, v, B), min(axb(A, w, B))) as r
      from E group by v;
    create table T as                                  ▷ contract by transforming edge table
      select distinct V.r as v, W.r as w
      from E, R as V, R as W
      where E.v = V.v and E.w = W.v and V.r != W.r;
    rowcount ← number of rows generated by the previous query
    drop table E; alter table T rename to E;
    if firstround then
      firstround ← false
      alter table R rename to L;
    else
      create table T as                                ▷ compose representative functions
        select L.v as v, coalesce(R.r, axb(A, L.r, B)) as r
        from L left outer join R on (L.r = R.v);
      drop table L, R; alter table T rename to L;
    end if
  until rowcount = 0
  alter table L rename to Result;
end procedure
```

Figure B.3: SQL-like pseudocode for the Randomised Contraction algorithm with deterministic space usage using the finite fields method. axb is assumed to be a user-defined function that computes the term $A \cdot x + B$ using arithmetic over the finite field $\mathbb{F}$.

B.5.4 SQL implementation

Our implementation of the Randomised Contraction algorithm in SQL takes as input a table G with two columns, v and w, containing vertex IDs, where each row represents an undirected edge of the input graph. Isolated vertices may be represented in this table as loop edges. The output is a table named *Result* with columns v and r, containing for each vertex v a row assigning a label r to the connected component of v.

Figure B.3 shows an SQL-like pseudocode implementation of Randomised Contraction using the finite fields method. It assumes the existence of a user-defined function $\mathrm{axb}(A, x, B)$ that treats a vertex ID x as an element of a finite field and computes the expression $A \cdot x + B$ using finite field arithmetic. Its implementation along with the actual Python/SQL code used for our experiments is given in Appendix B.A.

At each step, the choice of representatives is computed as a table R. For performance optimisation, we compute the representative as $r_i(v) = \min_{w \in N_{G_{i-1}}[v]} h_i(w)$ instead of using $\arg\min$. This runs faster because min is a built-in aggregate function in SQL. Since the values of r_i are no longer vertex IDs of the original graph, the vertices effectively get relabelled at each contraction step. Relabelling does not affect the correctness of the algorithm since the ultimate connected component labels are not required to be vertex IDs, but merely to satisfy uniqueness. Uniqueness is guaranteed by the fact that the functions h_i are bijections on the finite field used as the domain of the vertex IDs.

The contraction step replaces the vertex IDs in each row of the edge table E by their respective representatives, writing the result to a temporary table T. This is implemented by joining the edge table E with one copy of R for each of the two vertices involved. Loop edges are removed from the result to exclude isolated vertices from further computation.

Recall from section B.5.1 that the output of the algorithm is the composition of the representative functions $r_k \circ r_{k-1} \circ \cdots \circ r_1$. At step i, the algorithm uses the partial composition $r_{i-1} \circ \cdots \circ r_1$ stored in a table L to compute the next partial composition $r_i \circ \cdots \circ r_1$ by joining table L with table R. Since isolated vertices get deleted during the course of the algorithm, R represents only a partial function and a left outer join of L and R has to be used to preserve a row for each of the original vertices. Note that the relabelling introduced by the performance optimisation mentioned above has to be applied to all rows of L that do not have a counterpart in R. This is accomplished using the SQL function coalesce() which returns its first non-NULL argument.

The algorithm in Figure B.3 has deterministic space usage. Table E gets smaller at each step since duplicate edges and loop edges are removed. Table R, containing one row per vertex in E, shrinks accordingly. Table L, however, maintains its size throughout, storing one row per vertex of the input graph.

Figure B.4 shows a faster version of Randomised Contraction using slightly more intermediate storage. Instead of joining with the full table L at each step, we first compute and store all representative tables R_i. Each one is smaller than the previous one since it contains only one row for each vertex remaining in the computation. In a second loop, these tables are then joined "back to front" in a left outer join, again taking the necessary relabelling into account.

```
procedure RandomisedContractionFast(G)
  create table E as
    select v, w from G union all select w, v from G;
  initialise S with an empty stack
  i ← 0
  repeat
    i ← i + 1
    choose 0 ≠ A ∈ 𝔽 and B ∈ 𝔽 uniformly at random
    push (A, B) onto stack S
    create table R_i as                          ▷ compute representatives
      select v, least(axb(A, v, B), min(axb(A, w, B))) as r
      from E group by v;
    create table T as                            ▷ contract by transforming edge table
      select distinct V.r as v, W.r as w
      from E, R_i as V, R_i as W
      where E.v = V.v and E.w = W.v and V.r != W.r;
    rowcount ← number of rows generated by the previous query
    drop table E; alter table T rename to E;
  until rowcount = 0
  (A, B) ← (1, 0)
  while i > 1 do
    i ← i − 1
    pop (α, β) from stack S
    (A, B) ← (axb(A, α, 0), axb(A, β, B))
    create table T as                            ▷ compose representative functions
      select L.v as v, coalesce(R.r, axb(A, L.r, B)) as r
      from R_i as L left outer join R_{i+1} as R on (R_i.r = R_{i+1}.v);
    drop table R_i, R_{i+1}; alter table T rename to R_i;
  end while
  alter table R_1 rename to Result;
end procedure
```

Figure B.4: A faster version of Randomised Contraction with stochastic space usage. axb is assumed to be a user-defined function that computes the term $A \cdot x + B$ using arithmetic over the finite field $\mathbb{F}$.

The result of both algorithms is $r = r_k \circ r_{k-1} \circ \cdots \circ r_1$. The algorithm in Figure B.3 computes $(r_k \circ (r_{k-1} \circ \cdots \circ (r_2 \circ r_1)))$ whereas the algorithm in Figure B.4 computes the expression $(((r_k \circ r_{k-1}) \circ \cdots \circ r_2) \circ r_1)$. Note, however, that while the algorithm in Figure B.3 guarantees linear space requirements deterministically, the algorithm in Figure B.4 only guarantees this in expectation, as shown in Section B.6.2. The latter algorithm runs faster because it joins the representative tables in small-to-large order whereas the former one joins with the full-size representative table L at each step.

B.6 Performance analysis

B.6.1 Time complexity

The critical observation regarding the Randomised Contraction algorithm is that at each iteration the graph shrinks to at most a constant fraction γ of its vertices, in expectation, with $\gamma < 1$. Here we will prove $\gamma \leq 3/4$ for the random reals method and the finite fields method. A better bound of $2/3$ is proved in Appendix B.B for the case of full randomisation, such as with the random reals method. Note that we only need to consider graphs without isolated vertices since all isolated vertices get removed at the end of each step of the algorithm.

Theorem B.6.1. *Let $G = \langle V, E \rangle$ be a graph without isolated vertices. For each vertex v, let $h(v)$ denote either the random real allotted to v by the random reals method or the integer assigned by the finite fields method. Choose representatives $r(v) = \arg\min_{w \in N[v]} h(w)$. Then the expected total number of vertices chosen as representatives is at most $3/4|V|$.*

Proof. Divide the vertices into high and low vertices according to the median m of the distribution of a random $h(v)$: the high vertices v are those with $h(v) \geq m$.

For a vertex v to choose a high vertex as its representative, it must (1) itself be a high vertex, and (2) have only high vertices as neighbours. Given that v is not isolated, let us pick an arbitrary neighbour of it, w, and consider a weaker condition than (2): w must be a high vertex. For the random reals method, both conditions occur independently with probability $1/2$. For the finite fields method, let $q = |\mathbb{F}|$. The first condition occurs with probability $\lceil q/2 \rceil / q$ and the second condition, given the first, with probability $(\lceil q/2 \rceil - 1)/q$.

Thus, in expectation, no more than $1/4$ of the vertices choose a high vertex as a representative, proving that in total no more than $1/4|V|$ high vertices will be chosen as representatives. Even if all low vertices are representatives, this still amounts to an expected number of no more than $3/4|V|$ representatives in total. □

Let γ_i be the actual shrinkage factor at step i of the Randomised Contraction algorithm. This is a random variable with $\mathrm{E}(\gamma_i) \leq \gamma$. By re-randomising the vertex order at each step, all γ_i become independent and therefore uncorrelated. This guarantees that the total shrinkage over the first k steps is in expectation

$$\mathrm{E}(\prod_{i=1}^{k} \gamma_i) = \prod_{i=1}^{k} \mathrm{E}(\gamma_i) \leq \gamma^k.$$

Table B.1: Connected component algorithms

Algorithm	Number of steps	Space
Randomised Contraction[1]	exp. $O(\log \|V\|)$	exp. $O(\|E\|)$
Hash-to-Min [76]	$O(\log \|V\|)$	$O(\|V\|^2)$
Two-Phase [53]	$O(\log^2 \|V\|)$	$O(\|E\|)$
Cracker [58]	$O(\log \|V\|)$	$O(\frac{\|V\|\cdot\|E\|}{\log \|V\|})$

We now show that for any given $\epsilon > 0$ the algorithm terminates with probability $1 - \epsilon$ after $O(\log |V|)$ steps. Let R_k be the random variable describing the number of remaining vertices after k steps. The probability of the algorithm not terminating after k steps is $\Pr(R_k \geq 1)$. By Markov's inequality we have $\Pr(R_k \geq 1) \leq \mathrm{E}(R_k) \leq \gamma^k |V|$. Now $\gamma^k |V| \leq \epsilon \Leftrightarrow k \geq \log_\gamma \epsilon - \log_\gamma |V| = O(\log |V|)$, which is the desired conclusion.

B.6.2 Space requirements

The Randomised Contraction algorithm can be implemented in two variants shown in Figures B.3 and B.4, both using the finite fields method. Both require $\Theta(|E|)$ space for storing the edge table E. Note that the size of this edge table decreases at each step of the algorithm.

The first algorithm uses one table L of size $\Theta(|V|)$ and another table R starting at the same size and strictly shrinking throughout the algorithm, so that space usage for these tables is bounded deterministically by $\Theta(|V|)$. The algorithm shown in Figure B.4 stores intermediate tables of expected sizes $|V|, \gamma|V|, \gamma^2|V|, \ldots, \gamma^k|V|$, which sums up to a space usage of $\Theta(|V|)$ in expectation.

If the random reals method is used instead, both algorithms require an additional $\Theta(|V|)$ for storing a random number for each vertex, which does not change the overall space complexity.

In summary, since $|V| \leq |E|$, the space complexity of the first algorithm is $\Theta(|E|)$ deterministically while it is $\Theta(|E|) + \text{expected } \Theta(|V|)$ for the second algorithm.

In practice, if the algorithms are implemented as shown, the edge table is blown up two-fold in the setup stage. Also, at every iteration, a new edge table has to be generated before the old one is deleted, so, in total, the space requirements for storing edge information during the execution of the algorithm are up to four times the size of its original input.

B.7 Empirical evaluation

To evaluate the practical performance of our Randomised Contraction algorithm we used the open source MPP database Apache HAWQ which runs on an Apache Hadoop cluster. Since SQL does not natively support any control structures, we implemented the algorithm shown in Figure B.4 as a Python script that connects to the database and does all the "heavy

[1] Space usage can be made deterministic using the implementation in Fig. B.3.

lifting" using SQL queries. Finite field arithmetic over 64-bit integers was implemented in C as a user-defined SQL function.

We compare Randomised Contraction to three other leading algorithms for calculating connected components in a distributed computation setting. Their proven time and space complexities are summarised in Table B.1. Hash-to-Min and Two-Phase were implemented by their authors in MapReduce [64] whereas Cracker uses Spark [50].

The use of different execution environments and programming paradigms makes a direct comparison of the algorithms difficult. The authors of Hash-to-Min [76] and Two-Phase [53] did not publish original code, and comparison difficulties are further exacerbated by the fact that they did not document their cluster configuration and that [53] provides only relative timing results. We therefore had to port these algorithms to a unified execution environment.

We converted the two MapReduce algorithms and the Spark algorithm to SQL using direct, one-to-one translations. For example, in MapReduce, a "map" using key-value messages was converted to the creation of a temporary database table distributed by the key, and the subsequent "reduce" was implemented as an aggregate function applied on that table. Spark was converted using an equally direct, straightforward command-to-command mapping. This allows a comparison of different algorithms executing in the same relational database.

For Cracker, we were in addition able to run the original Spark code published in [58] on our cluster. We also implemented our Randomised Contraction algorithm in Spark SQL. This allows a limited comparison between the two execution environments Spark vs. MPP database.

B.7.1 Datasets

The datasets used are summarised in Table B.2. An application to a real-world dataset with nontrivial size is the analysis of the transaction graph of the crypto-currency Bitcoin [66]. At its core, Bitcoin is a data structure called blockchain that records all transactions within the system and is continuously growing.

On April 9, 2019 it consisted of 570,870 blocks with a total size of 250 GB, which we imported into our relational database. Transactions can be viewed as a bipartite graph consisting of transactions and outputs which in turn are used as inputs to other transactions. Each output is associated with an address, and it is a basic step for analysing the cash flows in Bitcoin to de-anonymise these addresses if possible. We used a well-known address clustering heuristic for this [63]: if a transaction uses inputs with multiple addresses then these addresses are assumed to be controlled by the same entity, namely the one that issued the transaction. To perform this analysis, we created the graph "Bitcoin addresses", linking addresses to the transactions using them as inputs. The connected components of this graph contain addresses assumed to be controlled by the same entities.

We also calculated the connected components of the full Bitcoin transaction graph. This reveals different markets that have not interacted with each other at all within the crypto-currency.

Table B.2: Datasets

Dataset	$\|V\|$	$\|E\|$	components
Andromeda	1,459 M	2,287 M	62,166 k
Bitcoin addresses	878 M	830 M	216,917 k
Bitcoin full	1,476 M	2,079 M	37 k
Candels10	83 M	238 M	39 k
Candels20	166 M	483 M	48 k
Candels40	332 M	975 M	91 k
Candels80	663 M	1,958 M	224 k
Candels160	1,326 M	3,923 M	617 k
Friendster	66 M	1,806 M	1
RMAT	39 M	2,079 M	5 k
Path100M	100 M	100 M	1
PathUnion10	154 M	154 M	10

Another important application of our algorithm is the analysis of social networks. We used the "com-Friendster" dataset from the Stanford Large Network Dataset Collection [56], the largest graph from that archive.

Connected component analysis can be used as an image segmentation technique. We converted a Gigapixel image ($69{,}536 \times 22{,}230$ px) of the Andromeda galaxy [67] to a graph by generating an edge for every pair of horizontally or vertically adjacent pixels with an 8-bit RGB colour vector distance up to 50. The vertex IDs were chosen at random so that they would not reflect the geometry of the original image.

The same technique can be applied to three-dimensional images such as medical images from MRI scans, or to video. We used a 4K-UHD video of a flight through the CANDELS Ultra Deep Survey field [68] and converted some frames of it to a graph using pixel 6-connectivity (x, y, and time) and a colour difference threshold of 20, again randomising the vertex IDs. By using an increasing number of frames we generated a series of datasets (Candels10 ... Candels160) with similar properties and of increasing size for evaluating scalability of the algorithms.

For comparison with [53], we generated a large random graph using the R-MAT method [18] with parameters (0.57, 0.19, 0.19, 0.05), which are the parameters used in [53]. Vertex IDs were randomised to decouple the graph structure from artefacts of the generation technique.

Two worst-case graphs complete our test bench. As shown in the theoretical analysis, Randomised Contraction maintains its logarithmic and quasi-linear performance bounds on any input graph. By contrast, all other algorithms examined have known worst-case inputs that exploit their weaknesses. Path100M is a path graph with 100 million sequentially numbered vertices causing prohibitively large space usage in Hash-to-Min and Cracker. PathUnion10 is the worst case for the Two-Phase algorithm, a union of path graphs of different lengths with vertices numbered in a specific way.

Our 2D and 3D image connectivity datasets are low-degree graphs: each vertex connects only to a handful of other vertices (at most 4 in 2D, at most 6 in 3D). This is a property

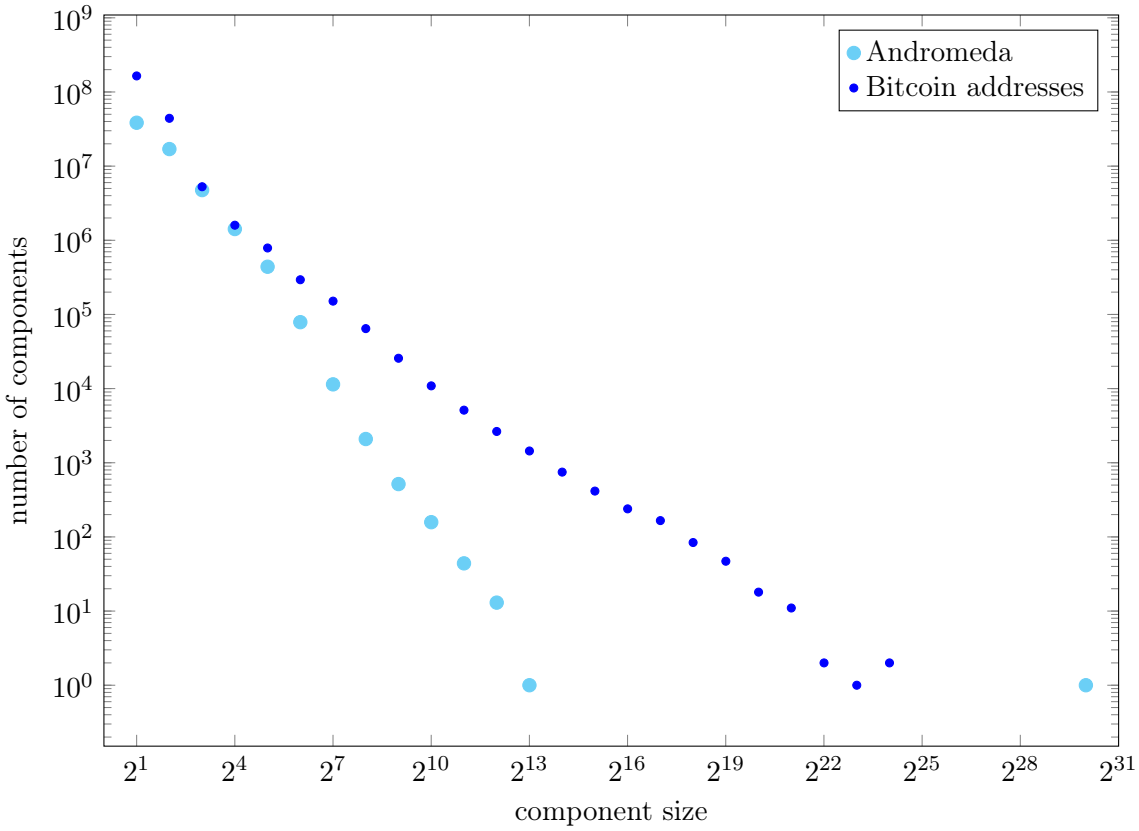


Figure B.5: Connected component sizes exhibit a roughly scale-free distribution for both the Andromeda and the Bitcoin address datasets.

that holds in a larger class of graphs of real-world interest, such as, for example, street networks.

With the exception of this degree restriction (for the Andromeda and Candels graphs), however, all graphs in our benchmark exhibit traits that are emblematic of the general class of real-world large graphs, for which reason we are confident that our results are general.

As an example, consider the distribution of our graphs' component sizes. Large real-world graphs typically exhibit a property known as scale-freedom. Scale-freedom in component sizes indicates that on a log-log scale a graph exhibits a (roughly) linear relationship between the size of a component and the number of components of this same size. In Figure B.5, we demonstrate that the Bitcoin address graph, predictably, shows this log-log linear behaviour.

As can also be seen in Figure B.5, however, the corresponding plot for the Andromeda benchmark graph shows the same behaviour, so is, in the relevant metrics, also representative of large real-world graphs, despite its construction from an image. (Notably, the single outlier for Andromeda is the image's black background.)

B.7.2 In-database benchmark results

For performance measurements we used a database cluster consisting of five virtual machines, each with 48 GiB of RAM and 12 CPU cores (Intel Skylake @2.2 GHz), running HAWQ version 2.3.0.0 on the Hortonworks Data Platform 2.6.1. The tests were run on an otherwise idle database.

We have run each of the algorithms three times on each of the target data sets and measured the mean and the standard deviation of the computation time. Like any other parallel processing, in-database execution entails its own inherent variabilities, for which reason we did not expect even the deterministic algorithms to complete in precisely consistent run-times. We did, however, expect the randomised algorithm to have somewhat higher variability in its completion time. Observing the relative standard deviation (i.e. the ratio between the standard deviation and the mean), the average value for Randomised Contraction was 4.0% as compared to 2.2%, 2.1%, and 1.6% for Hash-to-Min, Two-Phase, and Cracker, respectively. We conclude that the variability added by randomisation is not, comparatively, very high.

Table B.3 and Figure B.6 show the average runtimes in seconds. Hash-to-Min did not finish on the larger datasets with the available resources. Both Hash-to-Min and Cracker cannot handle the Path100M dataset due to their quadratic space usage (on a shorter path of 100,000 vertices they already use more than 100 GB). On all datasets Randomised Contraction performed best, generally leading by a factor of 2 to 12 compared to the other algorithms. On the graph RMAT the advantage was least pronounced.

The sequence of Candels datasets, roughly doubling in size from one to the next, demonstrates the scalability of the Randomised Contraction algorithm. Its runtime is essentially linear in the size of the graph.

Real world space usage of the algorithms has two aspects. One is the maximum amount of storage used by the algorithms at any given time, taking into account the amount of

Table B.3: Runtimes in seconds

Dataset	RC	HM	TP	CR
Andromeda	**5431**	–	37987	14506
Bitcoin addresses	**1530**	11696	9811	3457
Bitcoin full	**6398**	–	77359	26015
Candels10	**424**	3178	1425	867
Candels20	**749**	5868	2836	1766
Candels40	**1482**	13892	6363	3726
Candels80	**3463**	–	15560	8619
Candels160	**9260**	–	32615	23409
Friendster	**2462**	9554	4409	5092
RMAT	**2151**	4384	2816	3187
Path100M	**366**	–	1406	–
PathUnion10	**386**	–	4022	1202

RC = Randomised Contraction, HM = Hash-to-Min
TP = Two-Phase, CR = Cracker

Table B.4: Maximum space used in GB

Dataset	input	RC	HM	TP	CR
Andromeda	59	276	–	**115**	263
Bitcoin addresses	21	109	88	**43**	110
Bitcoin full	72	255	–	**108**	272
Candels10	6	27	21	**12**	24
Candels20	12	55	42	**24**	50
Candels40	25	110	86	**48**	100
Candels80	50	221	–	**96**	201
Candels160	102	443	–	**193**	403
Friendster	47	190	183	**91**	181
RMAT	54	217	120	**86**	169
Path100M	3	13	–	**5**	–
PathUnion10	4	20	–	**8**	20

Table B.5: Total gigabytes written

Dataset	input	RC	HM	TP	CR
Andromeda	59	**552**	–	1768	905
Bitcoin addresses	21	**215**	804	557	306
Bitcoin full	72	**690**	–	1858	1151
Candels10	6	**48**	148	93	61
Candels20	12	**97**	295	179	125
Candels40	25	**196**	618	369	251
Candels80	50	**394**	–	774	504
Candels160	102	**790**	–	1481	1009
Friendster	47	309	481	**258**	294
RMAT	54	259	248	**169**	177
Path100M	3	**31**	–	75	–
PathUnion10	4	**48**	–	264	116

space freed by deleting temporary tables. The other, arguably more important metric for database implementations is the total amount of data written to the database while executing the algorithms.

The latter is significant if the whole algorithm is implemented as a *transaction* in a database. A transaction combines a number of operations into one atomic operation that either succeeds as a whole or gets undone completely (*rollback*). In order to achieve this behaviour, most databases delete temporary tables only at the successful completion of the whole algorithm, and therefore storage is needed for all data written during its execution.

Table B.4 shows the algorithms' maximum space usage in comparison with the input size. Here the Two-Phase algorithm uses the least space on all datasets, taking no more than 2 times the storage of the input dataset. Our time-optimised implementation of the Randomised Contraction algorithm stays within the expected bounds and is never more than 2.6 times the space requirements of the Two-Phase algorithm. Table B.5 shows the total amount of data written which would need to be stored in a transaction. Here Randomised Contraction is best in most cases and performs worse only on Friendster and RMAT.

B.7.3 Database performance vs. Spark

In [58], Lulli et al. implement Cracker, an optimised version called Salty-Cracker, Hash-to-Min, and several other algorithms in the distributed computing framework Spark [50]. Their published source code is memory intensive and works within our resources only on smaller graphs. Its execution failed on graphs in our test-bench.

For their most highly optimised version of the Cracker algorithm the dataset with the highest runtime was "Streets of Italy" (19 M vertices, 20 M edges). The reported time was 1338 seconds, which was the best among all algorithms compared. We ran our Randomised Contraction algorithm on this same dataset in-database and it finished in 143 seconds. Our database implementation of the Cracker algorithm took 261 seconds.

Note the considerable difference between resources used: the results reported in [58] were obtained on five nodes with 128 GB of RAM and 32 CPU cores each. Our database cluster had less than half the RAM and half the CPU cores. Also the database was configured as it might be in a real-world production environment, never allocating more than 20% of the resources to a single query.

Formulating one's algorithm in the form of SQL queries also has advantages beyond in-database execution, as it allows utilising it in other SQL and SQL-like execution environments. As an example, we implemented the Randomised Contraction algorithm in Spark SQL using Spark 2.1.1 and ran it on the Candels10 dataset, exported from the database as a distributed set of text files. This allowed the algorithm to scale up properly, but we note that it was still slower in Spark SQL than when executing in the database. The runtime on our cluster was roughly 2.3 times as long for the Spark SQL implementation as for the in-database one, despite both executing the same SQL code on the same hardware. We conjecture that the main reason for this is the higher level of maturity of the query optimisation that databases such as HAWQ provide.

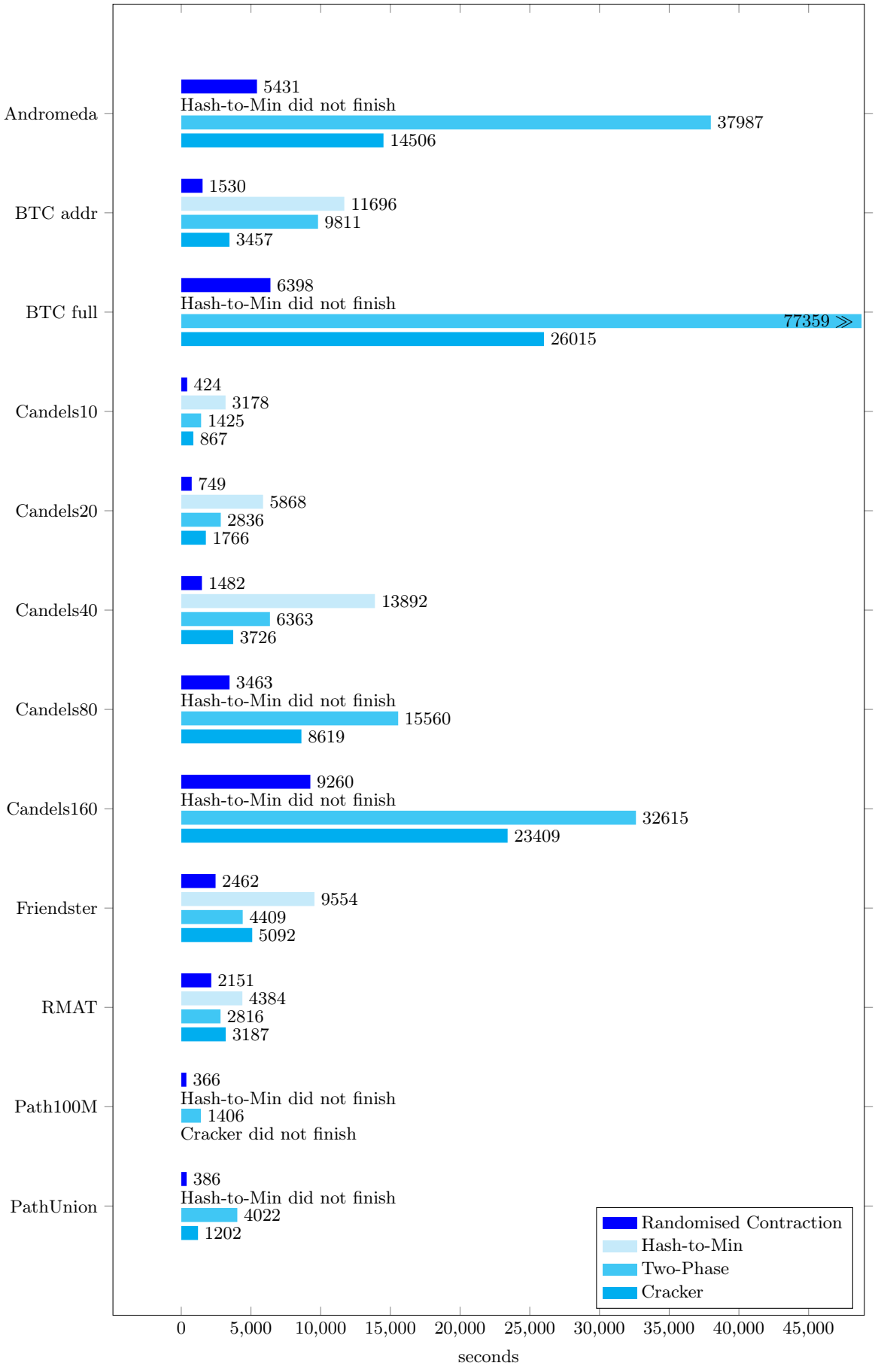


Figure B.6: In-database execution times for real world and synthetic datasets.

```
/* axplusb(a, x, b) calculates a*x+b over GF(2^64).
   Irreducible polynomial: x^64 + x^4 + x^3 + x + 1
*/
#define IRRPOLY 0x1b

PG_FUNCTION_INFO_V1(axplusb);
Datum
axplusb(PG_FUNCTION_ARGS)
{
  int64 a = PG_GETARG_INT64(0);
  int64 x = PG_GETARG_INT64(1);
  int64 b = PG_GETARG_INT64(2);

  int64 r = 0;
  while (x)
    {
      if (x & 1)
        r ^= a;
      x = (x>>1) & 0x7fffffffffffffff;
      if (a & (1LL << 63))
        a = (a<<1) ^ IRRPOLY;
      else
        a <<= 1;
    }
  PG_RETURN_INT64(r ^ b);
}
```

Figure B.7: The user-defined function axplusb.

We note that even this factor of 2.3 does not take into account the amount of time required to export the data from the database for analysis or to re-import the results back into the database, operations that would likely be required in a real world implementation.

B.8 Conclusions

We describe a novel algorithm for calculating the connected components of a graph that can be implemented in SQL and efficiently executed in a massively parallel relational database. Its robustness against worst case inputs and its scalability make it practical for Big Data analytics. The performance measured is not only due to our algorithm's ability to use a minimum number of SQL queries and to minimise the amount of data handled by each query, but also due to the work of the database's native, generic query execution optimiser.

With relational databases poised to remain the standard for storing transactional business data and with query execution engines improving year to year, the Randomised Contraction algorithm demonstrates that in-database processing can be a viable and competitive addition to the more widely used Big Data processing technologies.

B.A Implementation in Python/SQL

In this Appendix we show the implementation of the Randomised Contraction algorithm we used to run the experiments. The user-defined function implementing finite field arithmetic on 64-bit integers in C is shown in Figure B.7. It is called from SQL as axplusb(A,x,B) and computes the expression $A \cdot x + B$.

Our Python code is given below. It has been stripped of the surrounding infrastructure code. In the excerpt shown, dataset contains the name of the input table which is assumed to contain the edge list of the graph in two columns v1 and v2, each containing a 64-bit vertex ID.

r.log_exec() executes the SQL query passed as the third parameter and returns the number of rows generated. r.log_drop() drops the indicated table. r.execute() executes miscellaneous SQL queries. r.axplusb(A,x,B) calls the corresponding function in the database for finite field arithmetic.

```
r.log_exec("setup", 0, """\
create table ccgraph as
    select v1, v2 from {0}
    union all
    select v2, v1 from {0}
    distributed by (v1);
""".format(dataset))

roundno = 0
stackA = []
stackB = []
while True:
    roundno += 1
    ccreps = "ccreps{}".format(roundno)
    r_A = 0
    while r_A == 0:
        r_A = random.randint(-2**63,2**63-1)
    r_B = random.randint(-2**63,2**63-1)
    stackA.append(r_A)
    stackB.append(r_B)

    r.log_exec("ccreps", roundno, """\
    create table {ccreps} as
        select v1 v,
               least(axplusb({A},v1,{B}),
                     min(axplusb({A},v2,{B}))) rep
        from ccgraph
        group by v1
        distributed by (v);
    """.format(ccreps=ccreps, A=r_A, B=r_B))

    r.log_exec("ccgraph2", roundno, """\
```

```
    create table ccgraph2 as
        select r1.rep as v1, v2
        from ccgraph, {} as r1
        where ccgraph.v1 = r1.v
        distributed by (v2);
    """.format(ccreps))
    r.log_drop("ccgraph")

    graphsize = r.log_exec("ccgraph3", roundno, """\
    create table ccgraph3 as
        select distinct v1, r2.rep as v2
        from ccgraph2, {} as r2
        where ccgraph2.v2 = r2.v
              and v1 != r2.rep
        distributed by (v1);
    """.format(ccreps))
    r.log_drop("ccgraph2")
    r.execute("alter table ccgraph3 rename to ccgraph")

    if graphsize == 0:
        break

accA = 1
accB = 0

while True:
    roundno -= 1
    (accA, accB) = (r.axplusb(accA, stackA.pop(),0),
                    r.axplusb(accA, stackB.pop(), accB))
    if roundno == 0:
        break
    ccrepsr = "ccreps{}".format(roundno)
    ccrepsr1 = "ccreps{}".format(roundno+1)
    r.log_exec("result", roundno, """\
    create table tmp as
        select r1.v as v,
            coalesce(r2.rep,axplusb({A},r1.rep,{B})) as rep
        from {r1} as r1 left outer join
             {r2} as r2
             on (r1.rep=r2.v)
        distributed by (v);
    """.format(A=accA, B=accB, r1=ccrepsr, r2=ccrepsr1))
    r.log_drop(ccrepsr)
    r.log_drop(ccrepsr1)
    r.execute("alter table tmp rename to {}".format(ccrepsr))

r.execute("alter table ccreps1 rename to ccresult")
r.log_drop("ccgraph")
```

B.B Bounds on graph contraction

The Randomised Contraction algorithm requires that at each iteration the number of remaining vertices in the graph drops, in expectation, to at most a constant factor γ of the initial number, with $\gamma < 1$. In the body of the paper we prove $\gamma \leq 3/4$, requiring only the weaker form of randomisation that is achieved by the finite fields method. In this appendix we take a closer look at graph contraction under full randomisation and prove a better bound of $\gamma \leq 2/3$ for this case.

To do this, we generalise the problem to directed graphs. We use the following notation [10]: let $G = \langle V, A \rangle$ be a directed graph with n vertices. For a vertex $v \in V$, the set $N^+(v) = \{u \mid vu \in A\}$ is called the *out-neighbourhood* of v and the set $N^-(v) = \{w \mid wv \in A\}$ is called its *in-neighbourhood*. The sets $N^+[v] = N^+(v) \cup \{v\}$ and $N^-[v] = N^-(v) \cup \{v\}$ are called the *closed* out- and in-neighbourhoods, respectively.

We represent an ordering of the vertices by assigning to each vertex v a unique label $L(v) \in \{1, \ldots, n\}$. The representative of a vertex v under the order induced by the labelling L is defined as $r_L(v) = \arg\min_{w \in N^+[v]} L(v)$.

An undirected graph can be considered as a special case of a directed graph where each undirected edge corresponds to a pair of arcs in both directions. In this case we have $N(v) = N^+(v) = N^-(v)$ for all vertices v and our Randomised Contraction algorithm at each iteration chooses representatives as defined above. The total number of distinct representatives then determines the size of the next iteration's graph and therefore the amount of contraction at each iteration. We note that we do not know of any natural interpretation for the result of running the Randomised Contraction algorithm on a directed graph. Certainly, the output is not a division into connected components.

Given an ordering of the vertices, a vertex can have one of three types: it can be not the representative of any vertex (**type 0**), the representative of exactly one vertex (**type 1**), or the representative of two or more vertices (**type 2+**).

Lemma B.B.1. *Let $G = \langle V, A \rangle$ be a directed graph with n vertices. Fix a vertex $v \in V$ with $N^+(v) \neq \emptyset$. Then the number of orderings under which v is of type 1 is less than or equal to the number of orderings under which it is of type 0.*

Proof. We prove this by constructing an injective mapping from the labellings that make our fixed vertex v a type 1 vertex to those that make it a type 0 vertex. Consider a labelling L that makes v a type 1 vertex. Then there are two cases: (a) v represents itself and (b) v is the representative of exactly one other vertex.

In case (a) we have $L(v) = \min_{w \in N^+[v]} L(w)$. Let $u_1 = \arg\max_{w \in N^+(v)} L(w)$ and let L' be a new labelling obtained from L by exchanging the labels of v and u_1. Under this new labelling, v is of type 0: it no longer represents itself and it also does not represent any other vertex because its label is larger than before. Note that we can uniquely identify u_1 in this new labelling as $u_1 = \arg\min_{w \in N^+(v)} L'(w)$.

In case (b) we have $v = r_L(u_2)$ for some vertex $u_2 \in N^-(v)$ and $r_L(v) \neq v$. Let $u_1 = r_L(v)$. Then $L(u_2) > L(v) > L(u_1)$. Let L' be a new labelling obtained from L by exchanging the labels of v and u_2. Under this new labelling, u_2 represents itself and v is

of type 0: it is no longer a representative of u_2 and it also has not become a representative for any other vertex because its label is larger than before. Furthermore, $L'(u_2) = L(v) > L(u_1) = L'(u_1)$. Note that we can uniquely identify u_2 in this new labelling as the largest-labelled vertex in the in-neighbourhood of v that represents itself. To see this, assume by contradiction that there is a $w \in N^-(v)$ with $L'(w) > L'(u_2)$ and $r_{L'}(w) = w$. Then $u_2 \notin N^+(w)$, $v \in N^+(w)$, and $L(w) = L'(w) > L'(u_2) = L(v)$. From this and the fact that $L(w) = L'(w) = \min_{x \in N^+[w]} L'(x) \leq \min_{x \in N^+[w] \setminus \{v\}} L'(x) = \min_{x \in N^+[w] \setminus \{v\}} L(x)$ we conclude that $r_L(w) = v$. So under the labelling L, v is the representative of two distinct vertices u_2 and w, contradicting the assumption that it is of type 1.

To see that the mapping from L to L' is injective, it remains to be shown that from L' we can uniquely determine whether it was obtained from case (a) or case (b). Let $u_1 = \arg\min_{w \in N^+(v)} L'(w)$ and $u_2 = \arg\max_{w \in N^-(v):\, w = r_{L'}(w)} L'(w)$. If the latter does not exist, L' must have resulted from case (a). We show that otherwise L' satisfies $L'(u_2) > L'(u_1)$ if and only if it is the result of case (b). We have seen in case (b) that $L'(u_2) > L'(u_1)$. In case (a) we have $L(u_2) = L'(u_2) = \min_{x \in N^+[u_2]} L'(x) \leq \min_{x \in N^+[u_2] \setminus \{v, u_1\}} L'(x) = \min_{x \in N^+[u_2] \setminus \{v, u_1\}} L(x)$ and $L(v) < L(u_1)$. If $L(v) < L(u_2)$, this would imply that $r_L(u_2) = v$, contradicting the assumption that v is of type 1. So $L'(u_1) = L(v) > L(u_2) = L'(u_2)$. We conclude that the two cases cannot produce the same labelling and thus our mapping is injective. □

We can now prove the central theorem of this Appendix.

Theorem B.B.2. *Let $G = \langle V, A \rangle$ be a directed graph with n vertices and for all $v \in V$, $N^+(v) \neq \emptyset$. Let L be a labelling of G chosen uniformly at random. Then the expected number of vertices chosen as representatives by any vertex satisfies $\mathrm{E}(|\{r_L(v) \mid v \in V\}|) \leq (2/3)n$. This is a tight bound.*

Proof. Let R_0, R_1, and R_{2+} be the expected number of vertices of type 0, 1, and 2+, respectively. From Lemma B.B.1 we know that for any fixed vertex v its probability of being of type 1 is less than or equal to its probability of being of type 0, since these probabilities are proportional to the corresponding numbers of orderings. This shows $R_1 \leq R_0$. Using $R_0 + R_1 + R_{2+} = n$, we get

$$2R_1 + R_{2+} \leq n.$$

By counting the number of vertices being represented by each vertex we have

$$R_1 + 2R_{2+} \leq n.$$

Summing the last two equations and dividing by 3 we get

$$R_1 + R_{2+} \leq \frac{2}{3}n,$$

which is the desired conclusion because $R_1 + R_{2+}$ is the expected number of representatives.

To prove that the bound is tight, consider that $\gamma = 2/3$ is attained when G is the directed 3-cycle. □

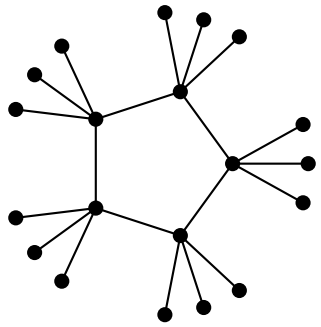


Figure B.8: Graph with highest known contraction factor γ

Note that the proven bound is only tight for directed graphs. The worst-case (highest) value of γ for undirected graphs is an open question. The graph with the highest γ value known is the one depicted in Figure B.8. It has $\gamma = 81215/144144 \approx 56.343\%$.

References

[1] Apache Hadoop – An open-source framework for reliable, scalable, distributed computing. `https://hadoop.apache.org/`.

[2] Apache Pig – A high-level data-flow language and execution framework for parallel computation. `https://pig.apache.org/`.

[3] MPI forum. `https://www.mpi-forum.org/`. Accessed: December 16, 2021.

[4] Spark release 3.0.0. `https://spark.apache.org/releases/spark-release-3-0-0.html`, June 2020. Accessed: July 27, 2020.

[5] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *The design and analysis of computer algorithms*. Addison-Wesley series in computer science and information processing. Addison-Wesley Pub. Co., Reading, Mass., 1974.

[6] Alexandrov, A., Ionescu, M. F., Schauser, K. E., and Scheiman, C. LogGP: Incorporating long messages into the LogP model – One step closer towards a realistic model for parallel computation. In *Proceedings of the Seventh Annual ACM symposium on Parallel Algorithms and Architectures* (1995), pp. 95–105.

[7] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al. Spark SQL: Relational data processing in Spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of Data* (2015), pp. 1383–1394.

[8] Arora, S., and Barak, B. *Computational complexity: A modern approach*. Cambridge University Press, 2009.

[9] Awerbuch, B., and Shiloach, Y. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *Computers, IEEE Transactions on 100*, 10 (1987), 1258–1263.

[10] Bang-Jensen, J., and Gutin, G. Z. *Digraphs: Theory, Algorithms and Applications*, second ed. Springer Science + Business Media, 2009.

[11] Bar-Noy, A., and Kipnis, S. Designing broadcasting algorithms in the postal model for message-passing systems. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures* (New York, NY, USA, 1992), SPAA '92, Association for Computing Machinery, pp. 13–22.

[12] Bögeholz, H., Brand, M., and Todor, R.-A. In-database connected component analysis. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)* (2020), IEEE, pp. 1525–1536.

[13] Boppana, R. B., and Sipser, M. The complexity of finite functions. In *Algorithms and complexity*, J. van Leeuwen, Ed., Handbook of Theoretical Computer Science. Elsevier, 1990, pp. 757–804.

[14] Borodin, A. On relating time and space to size and depth. *SIAM journal on computing 6*, 4 (1977), 733–744.

[15] Brand, M. *Computing with Arbitrary and Random Numbers*. PhD thesis, Monash University, 2013.

[16] Brin, S., and Page, L. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems 30*, 1-7 (1998), 107–117.

[17] Cameron, K. W., Ge, R., and Sun, X.-H. $\log_n$P and $\log_3$P: Accurate analytical models of point-to-point communication in distributed systems. *IEEE Transactions on Computers 56*, 3 (2007), 314–327.

[18] Chakrabarti, D., Zhan, Y., and Faloutsos, C. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining* (2004), SIAM, pp. 442–446.

[19] Chang, L., Wang, Z., Ma, T., Jian, L., Ma, L., Goldshuv, A., Lonergan, L., Cohen, J., Welton, C., Sherry, G., et al. HAWQ: A massively parallel processing SQL engine in Hadoop. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of Data* (2014), pp. 1223–1234.

[20] Codd, E. F. A relational model of data for large shared data banks. *Communications of the ACM 13*, 6 (1970), 377–387.

[21] Codd, E. F. *The relational model for database management: Version 2*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[22] Cole, R. Parallel merge sort. *SIAM Journal on Computing 17*, 4 (1988), 770–785.

[23] Cook, S. A. A taxonomy of problems with fast parallel algorithms. *Information and control 64*, 1-3 (1985), 2–22.

[24] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms*, vol. 3. MIT Press Cambridge, MA, 2001, ch. 21.

[25] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and Von Eicken, T. LogP: Towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming* (1993), pp. 1–12.

[26] Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation – Volume 6* (2004), pp. 137–150.

[27] DeWitt, D. J., and Gray, J. Parallel database systems: The future of high performance database processing. *Communications of the ACM 36* (1992).

[28] Dongarra, J., Beckman, P., Moore, T., Aerts, P., Aloisio, G., Andre, J.-C., Barkai, D., Berthou, J.-Y., Boku, T., Braunschweig, B., et al. The international exascale software project roadmap. *The International Journal of High Performance Computing Applications 25*, 1 (2011), 3–60.

[29] Eppstein, D., and Galil, Z. Parallel algorithmic techniques for combinatorial computation. *Annual Review of Computer Science 3* (1988), 233–283.

[30] Ezhova, N. A., and Sokolinskii, L. B. Survey of parallel computation models. *Vestnik Yuzhno-Ural'skogo Gosudarstvennogo Universiteta. Seriya Vychislitelnaya Matematika i Informatika 8*, 3 (2019), 58–91.

[31] Fan, J., Raj, A. G. S., and Patel, J. M. The case against specialized graph analytics engines. In *7th Biennial Conference on Innovative Data Systems Research (CIDR)* (2015).

[32] Fernandes, S., and Bernardino, J. What is BigQuery? In *Proceedings of the 19th International Database Engineering & Applications Symposium* (2015), pp. 202–203.

[33] Fich, F. E. The complexity of computation on the Parallel Random Access Machine. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, J. van Leeuwen, Ed. Elsevier, Amsterdam, 1990, pp. 757–804.

[34] Flynn, M. J. Very high-speed computing systems. *Proceedings of the IEEE 54*, 12 (1966), 1901–1909.

[35] Fortune, S., and Wyllie, J. Parallelism in Random Access Machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing* (1978), pp. 114–118.

[36] Fredman, M. L., and Willard, D. E. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. In *Proceedings 31st Annual Symposium on Foundations of Computer Science* (1990), IEEE, pp. 719–725.

[37] Gartner. Definition of Big Data – IT glossary. `https://www.gartner.com/en/information-technology/glossary/big-data`. Accessed: October 8, 2021.

[38] Gazit, H. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM Journal on Computing 20*, 6 (1991), 1046–1067.

[39] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating Systems Principles* (2003), pp. 29–43.

[40] Giani, A., Bitar, E., Garcia, M., McQueen, M., Khargonekar, P., Poolla, K., et al. Smart grid data integrity attacks. *Smart Grid, IEEE Transactions on 4*, 3 (2013), 1244–1253.

[41] Gill, J. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing 6*, 4 (1977), 675–695.

[42] Goldschlager, L. M. A universal interconnection pattern for parallel computers. *Journal of the ACM (JACM) 29*, 4 (1982), 1073–1086.

[43] Halperin, S., and Zwick, U. An optimal randomized logarithmic time connectivity algorithm for the EREW PRAM. In *Proceedings of the Sixth Annual ACM symposium on Parallel Algorithms and Architectures* (1994), ACM, pp. 1–10.

[44] Hellerstein, J. M., Ré, C., Schoppmann, F., Wang, D. Z., Fratkin, E., Gorajek, A., Ng, K. S., Welton, C., Feng, X., Li, K., et al. The MADlib analytics library: Or MAD skills, the SQL. *Proceedings of the VLDB Endowment 5*, 12 (2012), 1700–1711.

[45] Hirschberg, D. S., Chandra, A. K., and Sarwate, D. V. Computing connected components on parallel computers. *Communications of the ACM 22*, 8 (1979), 461–464.

[46] Hockney, R. W. The communication challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel computing 20*, 3 (1994), 389–398.

[47] Hogan, E., Hui, P., Choudhury, S., Halappanavar, M., Oler, K., and Joslyn, C. Towards a multiscale approach to cybersecurity modeling. In *Technologies for Homeland Security (HST), 2013 IEEE International Conference on* (2013), IEEE, pp. 80–85.

[48] Hopcroft, J., and Tarjan, R. Algorithm 447: Efficient algorithms for graph manipulation. *Communications of the ACM 16*, 6 (1973), 372–378.

[49] Hopcroft, J. E., and Ullman, J. D. Set merging algorithms. *SIAM Journal on Computing 2*, 4 (1973), 294–303.

[50] Karau, H., and Warren, R. *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*. O'Reilly, 2017.

[51] Khudairi, S. The Apache Software Foundation announces Apache Spark as a top-level project. `https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces50`, Feb. 2014. Accessed: October 7, 2021.

[52] Kikuchi, K., Masuda, Y., Yamashita, T., Sato, K., Katagiri, C., Hirao, T., Mizokami, Y., and Yaguchi, H. A new quantitative evaluation method for age-related changes of individual pigmented spots in facial skin. *Skin Research and Technology 22*, 3 (2016), 318–324.

[53] Kiveris, R., Lattanzi, S., Mirrokni, V., Rastogi, V., and Vassilvitskii, S. Connected components in MapReduce and beyond. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), ACM, pp. 1–13.

[54] Knuth, D. E. *The Art of Computer Programming, Volume 3 (2nd Ed.): Sorting and Searching*. Addison-Wesley, Reading, MA, USA, 1998.

[55] Ladner, R. E., and Fischer, M. J. Parallel prefix computation. *Journal of the ACM (JACM) 27*, 4 (1980), 831–838.

[56] Leskovec, J., and Krevl, A. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[57] Ling, S., and Xing, C. *Coding theory: A first course*. Cambridge University Press, 2004.

[58] Lulli, A., Carlini, E., Dazzi, P., Lucchese, C., and Ricci, L. Fast connected components computation in large graphs by vertex pruning. *IEEE Transactions on Parallel and Distributed systems 28*, 3 (2017), 760–773.

[59] Madhavji, N. H., Miranskyy, A., and Kontogiannis, K. Big picture of Big Data software engineering: With example research challenges. In *2015 IEEE/ACM 1st International Workshop on Big Data Software Engineering* (2015), IEEE, pp. 11–14.

[60] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (2010), ACM, pp. 135–146.

[61] Marlow, S., et al. Haskell 2010 language report. `https://www.haskell.org/onlinereport/haskell2010`, 2010.

[62] McCune, R. R., Weninger, T., and Madey, G. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Surveys (CSUR) 48*, 2 (2015), 25.

[63] Meiklejohn, S., Pomarole, M., Jordan, G., Levchenko, K., McCoy, D., Voelker, G. M., and Savage, S. A fistful of Bitcoins: Characterizing payments among men with no names. In *Proceedings of the 2013 conference on Internet measurement conference* (2013), ACM, pp. 127–140.

[64] Miner, D., and Shook, A. *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems*. O'Reilly, 2012.

[65] MPICH.ORG. High-performance portable MPI. https://www.mpich.org/. Accessed: January 6, 2021.

[66] NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008. Accessed: May 27, 2019.

[67] NASA, ESA, DALCANTON, J., WILLIAMS, B. F., JOHNSON, L. C., THE PHAT TEAM, AND GENDLER, R. Sharpest ever view of the andromeda galaxy. http://www.spacetelescope.org/images/heic1502a/, Jan. 2015. Accessed: January 23, 2018.

[68] NASA, ESA, SUMMERS, F., DEPASQUALE, J., BACON, G., AND (STSCI), Z. L. A flight through the CANDELS ultra deep survey field. http://hubblesite.org/video/984/science, Sept. 2017. Accessed: January 23, 2018.

[69] NOWOSIELSKI, A., FREJLICHOWSKI, D., FORCZMAŃSKI, P., GOŚCIEWSKA, K., AND HOFMAN, R. Automatic analysis of vehicle trajectory applied to visual surveillance. In *Image Processing and Communications Challenges 7*. Springer, 2016, pp. 89–96.

[70] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of Data* (2008), SIGMOD '08, pp. 1099–1110.

[71] PARBERRY, I. *Parallel complexity theory*. John Wiley and Sons Inc., New York, NY, 1987.

[72] PATIL, G. P., ACHARYA, R., AND PHOHA, S. Digital governance, hotspot detection, and homeland security. *Encyclopedia of Quantitative Risk Analysis and Assessment* (2007).

[73] PIPPENGER, N. On simultaneous resource bounds. In *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)* (1979), IEEE, pp. 307–311.

[74] POSPIECH, M., AND FELDEN, C. Big Data – A theory model. In *2016 49th Hawaii International Conference on System Sciences (HICSS)* (Los Alamitos, CA, USA, Jan 2016), IEEE Computer Society, pp. 5012–5021.

[75] PRATT, V. R., AND STOCKMEYER, L. J. A characterization of the power of vector machines. *Journal of Computer and System Sciences 12*, 2 (1976), 198–221.

[76] RASTOGI, V., MACHANAVAJJHALA, A., CHITNIS, L., AND SARMA, A. D. Finding connected components in Map-Reduce in logarithmic rounds. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on* (2013), IEEE, pp. 50–61.

[77] RAYNAL, M. *Distributed Algorithms for Message-Passing Systems*, vol. 500. Springer, 2013.

[78] Reports, V. Big Data & business analytics market statistics 2030. `https://reports.valuates.com/market-reports/ALLI-Manu-3K13/gblobal-big-data-and-business-analytics`. Accessed: November 24, 2021.

[79] Rico-Gallego, J. A., Díaz-Martín, J. C., Manumachu, R. R., and Lastovetsky, A. L. A survey of communication performance models for high-performance computing. *ACM Computing Surveys (CSUR) 51*, 6 (2019), 1–36.

[80] Sadalage, P. J., and Fowler, M. *NoSQL distilled: A brief guide to the emerging world of polyglot persistence*. Pearson Education, 2013.

[81] Savage, J. E. *Models of Computation: Exploring the Power of Computing*, 1st ed. Addison-Wesley, Boston, MA, USA, 1997.

[82] Savitch, W. J., and Stimson, M. J. Time bounded Random Access Machines with parallel processing. *Journal of the ACM (JACM) 26*, 1 (1979), 103–118.

[83] Schneier, B. Description of a new variable-length key, 64-bit block cipher (Blowfish). In *International Workshop on Fast Software Encryption* (1993), Springer, pp. 191–204.

[84] Schnorr, C.-P. The network complexity and the Turing machine complexity of finite functions. *Acta Informatica 7*, 1 (1976), 95–107.

[85] Shiloach, Y., and Vishkin, U. An O(log n) parallel connectivity algorithm. *Journal of Algorithms 3*, 1 (1982), 57–67.

[86] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The Hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on* (2010), IEEE, pp. 1–10.

[87] Silberschatz, A., Korth, H. F., and Sudarshan, S. *Database System Concepts*, 7th ed. McGraw-Hill Education, New York, NY, USA, 2020.

[88] Skillicorn, D. B. The Bird-Meertens formalism as a parallel model. In *Software for Parallel Computation*. Springer, 1993, pp. 120–133.

[89] Snyder, L. Type architectures, shared memory, and the corollary of modest potential. *Annual Review of Computer Science 1*, 1 (1986), 289–317.

[90] Song, W., Wu, D., Xi, Y., Park, Y. W., and Cho, K. Motion-based skin region of interest detection with a real-time connected component labeling algorithm. *Multimedia Tools and Applications* (2016), 1–16.

[91] Stonebraker, M. The case for shared nothing. *IEEE Database Eng. Bull. 9*, 1 (1986), 4–9.

[92] Sur, S., Koop, M. J., and Panda, D. K. High-performance and scalable MPI over InfiniBand with reduced memory usage: An in-depth performance analysis. In

Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (New York, NY, USA, 2006), SC '06, Association for Computing Machinery.

[93] Tarjan, R. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM) 22*, 2 (1975), 215–225.

[94] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Antony, S., Liu, H., and Murthy, R. Hive – A petabyte scale data warehouse using Hadoop. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)* (2010), IEEE, pp. 996–1005.

[95] Turing, A. M. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society 42*, 2 (1936), 230–265.

[96] Valiant, L. G. Parallelism in comparison problems. *SIAM Journal on Computing 4*, 3 (1975), 348–355.

[97] Valiant, L. G. A bridging model for parallel computation. *Commun. ACM 33*, 8 (Aug. 1990), 103–111.

[98] Valiant, L. G. Chapter 18 – General purpose parallel architectures. In *Algorithms and Complexity*, J. van Leeuwen, Ed., Handbook of Theoretical Computer Science. Elsevier, Amsterdam, 1990, pp. 943–971.

[99] Vishkin, U. An optimal parallel connectivity algorithm. *Discrete Applied Mathematics 9*, 2 (1984), 197–207.

[100] Wamba, S. F., Akter, S., Edwards, A., Chopin, G., and Gnanzou, D. How "Big Data" can make big impact: Findings from a systematic review and a longitudinal case study. *International Journal of Production Economics 165* (2015), 234–246.

[101] Willson, I. A. The evolution of the massively parallel processing database in support of visual analytics. *Information Resources Management Journal (IRMJ) 24*, 4 (2011), 1–26.

[102] Wu, X., Yuan, P., Peng, Q., Ngo, C.-W., and He, J.-Y. Detection of bird nests in overhead catenary system images for high-speed rail. *Pattern Recognition 51* (2016), 242–254.

[103] Yip, M., Shadbolt, N., and Webber, C. Structural analysis of online criminal social networks. In *Intelligence and Security Informatics (ISI), 2012 IEEE International Conference on* (2012), IEEE, pp. 60–65.

[104] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient Distributed Datasets: A

fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (2012), pp. 15–28.

[105] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., Stoica, I., et al. Spark: Cluster computing with working sets. *HotCloud 10*, 10-10 (2010), 95.