Unary Relational Operators over Phoenix 2

Petros Lambropoulos

Master of Science

Computer Science

School of Informatics

University of Edinburgh

2012

Abstract

The efficient use of parallelism in multicore shared memory systems, is a hot topic in the setting of multicore chips with increasing core numbers. Referring to this issue, this thesis explores the applicability of a large-scale distributed computing model and programming paradigm in this new setting.

A multicore version of the popular MapReduce distributed paradigm is utilized for the development of a new library. This library, called Relational Phoenix, enables the MapReduce parallelism pattern to be used for relational processing, implementing the basic unary relational operators. By allowing them to combine and form more complicated types of processing, this library aims to make a first step towards other declarative-style, relational processing solutions utilizing MapReduce.

Relational Phoenix is designed and implemented to serve the application area, taking into consideration the context of shared-memory multicores. It is evaluated accordingly, in terms of usability, scalability and efficiency in the application area.

The experiments demonstrate that the operators behave well on different record cardinality and show good scalability for up to 8 threads. Usability is measured in lines of code and programs written with Relational Phoenix, are shorter by an order of magnitude. Finally, possible extensions of the library are examined.

Acknowledgements

I would like to thank my friends in Edinburgh for their support during this year.

My supervisor, Stratis Viglas, for his guidance, his persistent effort when helping me and his poised and calm attitude towards this thesis.

My friends, Themistoklis Diamantopoulos and Michalis Pitidis for their valuable inputs, but most importantly for the interesting conversations we had throughout the year. Without them, the outcome would not have been the same.

Last but not least, I would like to thank my family for their invaluable support.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Petros Lambropoulos*)

Table of Contents

List of Figures

Chapter 1

Introduction [1]

In the past decade, processor clock speeds and caches seized to increase with the rate of the preceding years. The much slower rate, along with the exhausting exploitation of instruction-level-parallelism (IPL), led to a dramatic decrease of improvement in computing power for single core computers. In 2005, Intel followed IBMs lead to support performance increase on multiple cores. Nowadays, every major hardware manufacturer is turning towards multicore and the number of cores per system will keep increasing [3, 4].

Although multicore systems are helpful for multiple programs running simultaneously -otherwise referred to as multi-programmed workloads- it is a different challenge to run individual tasks faster. Either by using CPU core programming and scheduling or by using multiple threads to achieve parallelism, development of even modestly parallel programs is a much more challenging task from their sequential equivalent [4].

1.1 Parallel Programming

Parallel software development is generally considered a time consuming and effort intensive task [4]. There are several reasons for this. Parallel programming by itself is much more complex to specify, because of concurrency. It also requires coordination of tasks, which can be very perplexing [5]. Parallel programs face race conditions and deadlocks and even if correct, they can be wildly non-deterministic and hard to test [6].

There are several side issues as well. Parallel software development tools and environments are not as standardized and mature as their sequential counterparts are, while there are serious problems with scalability and algorithm portability. All of these rea-

[1] Part of this work appears in [1, 2]

sons lead to considering whether parallelism of an application is worth the effort. In many cases the long process needed to accelerate an application with such means may render it obsolete, after new changes in hardware and software platforms take place [5].

1.2 Cloud on a Chip

As the number of cores per chip is expected to increase geometrically in the following years, using machines with hundreds of processor cores will become a reality. Handling this new type of computer power effectively is one of the current challenges in the scientific community. An interesting way to approach this problem effectively is by applying already existing methods from large-scale distributed computing. The purpose of this project is to explore such use of large-scale and likely cloud-based data processing paradigms on multicore chips.

Distributed computing has been exploring the use of multiple computing units for years and although it is a different setting from multicores, some ideas can be inherited from previous endeavours. Parallel programming frameworks like the streaming dataflow based model FastFlow [7] and Thain's parallel abstractions [8] derive from the distributed field, while in the database field traditional parallel databases are challenged by distributed implementations even on multicore [9].

Perhaps the most characteristic example of such a transition is MapReduce [10]. Phoenix [11] was the first attempt to make use of the popular distributed dataflow processing framework on shared memory parallelism with different cores acting as worker MapReduce nodes.

1.3 Motivation

In the above described context, this project extends one of the existing systems that have ported the distributed MapReduce programming paradigm to the field of multicore. It aims to explore possibilities for data processing and data querying applications.

Particularly, *Phoenix 2* , a MapReduce system implemented for shared-memory multicore hardware, will be modified for the popular use of relational data processing and querying. The system will be extended in a way, so that is realizes unary relational algebra operators efficiently. The proposed changes will also mostly derive from work researched in the distributed field.

This approach has not been thoroughly researched in the shared-memory setting, thus it provides an interesting field of exploration. In particular, unary relational algebra operators are missing from the multicore implementations of MapReduce. These missing operators, like *aggregation*, *sorting*, *projection*, *selection* and *partition*, are very often used with MapReduce. The resulting model is designed, implemented and evaluated according to appropriate criteria, which apply for shared memory multicore systems.

1.4 Project Outline

This document is organized as follows:

- *Chapter 2* outlines the background of the project. It discusses MapReduce, the Phoenix projects and relational operators in general.
- *Chapter 3* is the related work. It includes high-level parallel programming paradigms, systems similar to Phoenix, and relational processing alternatives on MapReduce.
- *Chapter 4* explains the design decisions of the developed relational operator library, Relational Phoenix. Algorithms for every operator and the idea of combining them.
- *Chapter 5* describes the implementation of the operators. It includes the use of the library and the API.
- *Chapter 6* discusses the evaluation of the operators for various benchmarks, on two different architectures.
- *Chapter 7* is the future work and conclusion of the overall project.

Chapter 2

Background

2.1 MapReduce model

In the distributed setting, MapReduce was introduced by Google in 2006 [10] as a distributed data-processing framework targeted at large-scale, highly fault-tolerant, scalable clusters of commodity hardware. The ease of use of the framework made it easy to exploit parallelism for processing big-data. Apache Hadoop is a popular open-source implementation of the Google MapReduce model and the Google File System (GFS) [12].

Much like in functional programming, the model inputs a set of key/value pairs (k_1, v_1) and outputs a list of intermediate key/values (k_2, v_2) . In the second step it takes all values associated with the same key and produces a list of key/value pairs.

Map: $(k_1, v_1) \rightarrow (k_2, v_2)*$

Reduce: $(k_2, v_2*) \rightarrow (k_2, v_3)*$

The parallel execution of the two primitives is managed by the MapReduce runtime. The framework can also be characterized as a distributed sort, while a MapReduce applications dataflow is defined by the following entities:

- An *input reader* or *splitter*, which divides the input to appropriately sized *'chunks'*, fed to the *Map* function.
- A *Map function*, which takes a series of input keys and values, processes them and emits zero or more key/value pairs.
- A *partition function*, used to distribute pairs given the keys, to the number of *reducers* or *reduce tasks*.

- A *compare function*, used to sort the input of each reducer according to key.
- A *Reduce function*, which iterates over the values of each unique key to produce zero or more pairs to the output.
- The *output writer* writes the emitted pairs of the reducer to permanent storage, usally distributed file system.

As the model passed to multicores, new variations of the model sprung into existence targeting multicore CPUs, GPUs and heterogeneous platforms.

2.2 The Phoenix projects

Phoenix [11] is an implementation of the MapReduce model for shared-memory systems, which is used as a library providing a new programming API. The runtime manages thread creation, dynamic task scheduling, data partitioning and fault tolerance across processor cores. The model has proven promising with good scalability on performance and simple, functional-style, parallel code.

2.2.1 The API

The Phoenix API provides an application programming interface for C and C++. Two sets of functions are provided. The first set is provided by Phoenix runtime and is used to initiate the system and emit output pairs. The second set is consisted by the programmer defined functions. Except Map and Reduce the API provides the customizability of data-partitioning and key comparison.

Additionally, the runtime scheduler requires a series of arguments to be defined, which are used to specify and initialize runtime behaviour. There are eight basic fields and six used for performance tuning. Several of these fields integrate the API functions in the scheduler.

2.2.2 The Runtime

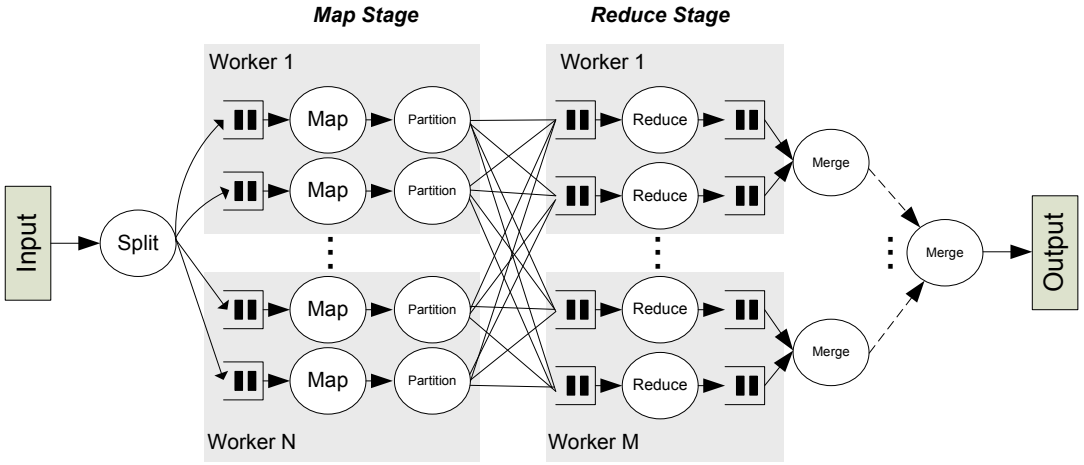


Figure 2.1: The Phoenix adaptation of the MapReduce workflow. Figure reproduced from [11].

Central to the Phoenix architecture is the *Intermediate Buffer*, a data structure formed as a matrix of buckets that stores intermediate results. Each *row* in this buffer is used by a specific Mapper, while each *column* is consumed by a specific Reducer.

At the beginning of every MapReduce job, the dispatcher spawns multiple worker threads that bind to the CPU cores. Worker threads are the equivalent of Virtual Machines in Hadoop's MR. Therefore, every worker is finally represented by a core, due to CPU bindings and all the data is shared.

In the Map phase every worker continuously splits the data and processes them through the user defined *map* function. The scheduler uses the *Splitter* to divide input pairs into equally sized units for the *map* tasks to process every worker bound to a *map* task. The *map* function emits key/value pairs of intermediate data, writing them to the corresponding *row* of the *Intermediate Buffer*. The possibility to optionally use a *combine* function exists. Like in the original model it performs local reduction after the Map phase.

At the Reduce phase, the *Partition* function splits the intermediate pairs into units for the *reduce* tasks. Workers feed the columns of intermediate data to the *reduce* functions generating the final result for the key. Again each worker is assigned a reduce function. The Phoenix implementation finishes computation with the Merge stage at the end of the Reduce phase. All generated results by the Reducers are merged to a single sorted-by-key output. Figure 2.2 illustrates the intermediate data structure of the system and its use in the workflow.

The runtime provides support for fault tolerance. Detection of faults occurs by worker time-outs. Fault recovery is attempted by re-execution of the failed task, with separate buffers allocated to avoid conflicts. There is no fault recovery for the scheduler. Also, the system can handle automatic scheduling decisions during execution.

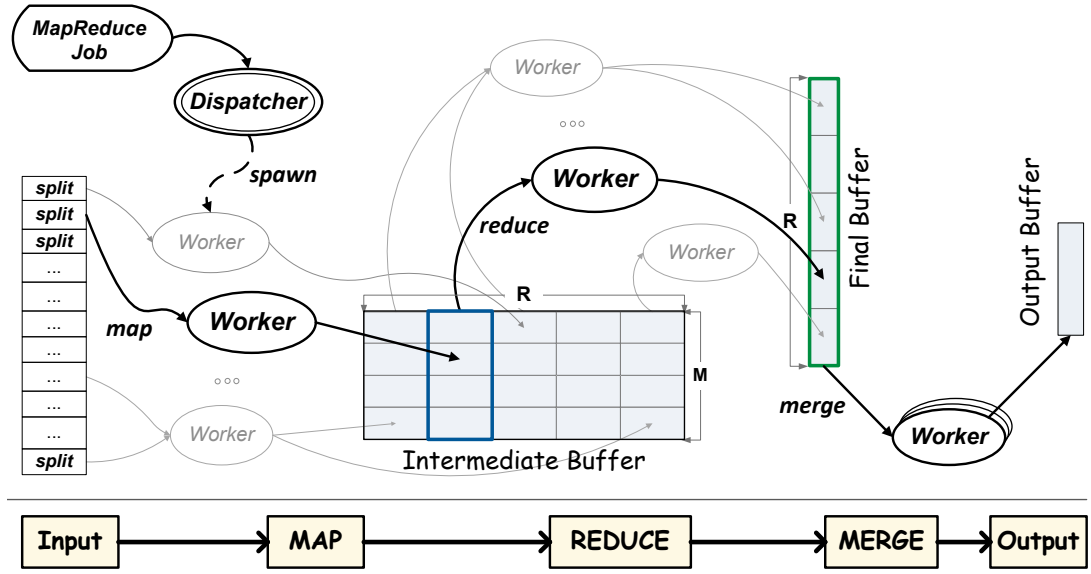


Figure 2.2: The Phoenix *Intermediate Buffer* and operation in the MapReduce workflow. Figure reproduced from [13].

2.2.3 Evaluation

Phoenix has been tested on two shared memory systems, using 8 benchmarks widely applicable with MapReduce. Applications from the enterprise computing domain included *Word Count, Reverse Index and String Match*. From scientific computing, *Matrix Multiply*. From artificial intelligence *Kmeans, PCA and Linear Regression* and from image processing *Histogram*.

Three datasets tested locality and scalability issues. Benchmarks were also run on their sequential versions for baseline and on statically scheduled P-threads for a direct comparison. Fault tolerance was also tested. The findings from the evaluation results showed Phoenix leads to scalable performance for multicore chips and symmetric multiprocessors. Despite runtime overheads the system performed similarly to applications written directly in P-threads for most cases.

2.3 Recent Phoenix versions

The Phoenix implementation was followed by a series of optimized systems. Several optimizations [14], shortcomings and limitations have since been explored and the popular distributed model has been implemented in multicore in two optimized versions since then.

2.3.1 Phoenix 2

Phoenix 2 [15] is an evolved dynamic runtime MapReduce implementation, that optimizes the Phoenix system. It proves that to efficiently execute a large-scale parallel system of this type, multi-layered optimizations should be applied by the developer. The runtime algorithms are possible to be selected taking into account NUMA challenges and also the interactions with the OS can be managed by Phoenix++.

The evaluation revealed a significant speedup (2.5x-19x times) over Phoenix for 256-thread runtimes. In cases, the system revealed bottlenecks that limit further scaling.

2.3.2 Phoenix++

With the limitations of Phoenix apparent, Phoenix++ [16] is an alternative implementation by the same group of researchers. A modular and flexible pipeline is provided to the user, that can easily be adapted to the characteristics of a particular workload.

Phoenix++ is compared against Phoenix 2, achieving a 4.7x speedup on average over the second. It proves faster and more scalable, often in logarithmic scale.

2.4 Relational Model

The relational model for database management is a database model proposed by Codd [17], based on first order predicate logic. The model provides a declarative method to specify data and queries. In the relational database system, users directly state the contents of a database and request the information, than let the system take care of describing data structures for storing the data and retrieval procedures for answering requests.

2.4.1 Relational languages for data management

In database systems, often information on a relational, or generally a database table needs to be extracted through some filtering process. This process is defined as *querying* and the high-level abstract language that enables an interface for the user to be able to specify data to be retrieved from the database's table(s), according to specific criteria, is called a *relational language.* The two most frequent relational languages are *relational algebra* and *relational calculus*.

SQL is a *declarative* programming language designed to manage data in relational database management systems (RDBMS). SQL is based on relational algebra and tuple relational calculus.

2.4.2 Relational algebra operators

Relational algebra is a *procedural* programming language, consisting of a set of *relational operators*. These *operators* manipulate *relations* based on other existing *relations*. A *relation* is a data structure with an unodered set of tuples with the same heading, or otherwise, with the same set of attributes. The most frequent relational algebra operations grouped in the following rough categories:

- *Primitive operations,* like selection, projection and rename.
- *Join and join-like operators,* such as natural join, θ-join and equijoin, semi-join, division and outer joins.
- *Operators for domain computations,* like aggregation.

2.4.3 Unary operators over MapReduce

This project concerns *unary* relational operators. A unary operation is an operation with only one operand, i.e. a single input. The unary relational operations discussed in this text are:

- *Select:* used to select a subset of tuples from a relation that satisfy a selection condition (or predicate).
- *Project:* used to select certain columns from a table and discard the other columns.
- *Sort:* returns the tuples ordered according to a field or a value.

- *(Hash) Partition:* is the division of the database elements to distinct parts. In hash partition, a value of a hash function decides membership in a partition.
- *Aggregate:* where the values of multiple rows are grouped as input on a function that applies certain criteria to form a single value, or a more significant meaning in a set.

Based on these operators, other relational operations can be expressed. For example, the implementation of a hash join requires the partitioning operation. Therefore, the unary operators can act as building blocks for more elaborate ones.

In the case of distributed MapReduce, these algorithms are already existent and used in practice. Figures 2.3, 2.4 show examples of such algorithms that run on Hadoop. They also show that the the algorithms emit records as keys.

This is typical for most algorithms in distributed MapReduce, however, it is not necessarily optimal for shared memory systems. These algorithms will be contrasted with the different algorithms designed for Phoenix, the choices of which were made taking into consideration of shared memory and pointer manipulation capabilities.

```
1 class Mapper
2         method Map(rowkey key, tuple t)
3                 if t satisfies the predicate
4                         Emit(tuple t, null)
```

Figure 2.3: An example SELECT algorithm for the MapReduce model on distributed systems, in Java-like pseudocode

```
1 class Mapper
2    method Map(rowkey key, tuple t)
3       tuple g = project(t)  // extract required fields to tuple g
4       Emit(tuple g, null)
5
6 class Reducer
7    method Reduce(tuple t, array n)   // n is an array of nulls
8       Emit(tuple t, null)
```

Figure 2.4: An example algorithm for PROJECT for the MapReduce model on Hadoop, in Java-like pseudocode. The reducer is used to eliminate possible duplicates.

Chapter 3

Related Work

3.1 High-level parallel programming models

Several programming models have been designed to exploit coordination of parallelism separately from the computations. The MapReduce programming model belongs to one of these. The categories include algorithmic skeletons, patterns, templates and components, as summarized in [18].

3.1.1 Algorithmic skeltons

Parallel algorithmic skeletons introduced by Cole [19], are higher level parallel pattern abstractions that can be reused on different problems referring to the same parallelism method.

Algorithmic skeletons allow through careful selection of skeleton nesting, complex parallel applications to be developed. Process scheduling, decomposition and mapping, as well as, memory management and communication are handled by the skeleton system, sometimes referred to as an Algorithmic Skeleton Framework (ASkF).

The developer works in a higher abstraction level, avoiding non-determinism and many of the typical parallel programming difficulties [18, 20, 21]. However, everything comes at a cost, since the only tools the programmer has are the parallel patterns supported by the skeleton itself. A successful skeleton system is difficult to implement, because it has to deal with complicated issues the developer is no longer responsible for, at the same time, it must support a wide range of patterns for the developer to use [20].

3.1.2 Patterns

Parallel programming patterns are developed under parallel pattern languages, patterns suggest code design that forms standardized parallel structure, communication and flow of data. Originally designed as abstractions in object-oriented programming, they have been utilized to create parallelism [22]. The programmer can parametrize the abstractions and generate parallel code, along with sequential parts.

As such implementations, one could consider *Intel Threading Building Blocks* [23] and *Microsoft Task Parallel Library*; The *Google MapReduce* model arguably also falls into this category [18]. All created by large technology companies. These solutions, developed far from academia are composed of heterogeneous code generation techniques, often losing track of structural abstraction. Nevertheless, they are highly promoted and are accepted as mainstream tools by developers.

3.1.3 Templates

Commonly known as *templates* or *genrics*, in object-oriented programming, templates provide support for data-parallel programming through objects and collective operations. Such examples include *join*, *map*, and *reduce* [24].Templates are integrated into C++ through the Standard Template Adaptive Parallel Library, which can use abstract data structures for parallel programming [25].

3.1.4 Components

Components are objects to associate operations with events. The model consists of a set of objects with published interfaces, that follow the rules of a specific concurrent model. Parallel programs are assembled in a component architecture, which is defined by a component model incorporating system components. Components are effective in the development of infrastructure services, which led to the Common Component Architecture [26], a standard for developing such applications.

3.2 MapReduce systems for multicores

Phoenix gave way to a variety of new systems, which came into existence aiming to utilize the MapReduce model on multicores. Many of these systems are orthogonal to the work in this project, as the developed operator library could be ported to any

of them. A careful analysis of the systems was conducted, to help on the choice of selecting an appropriate candidate underlying platform for the library.

3.2.1 Phoenix variations: Tiled-Map-Reduce

Tiled-Map-Reduce [13] is a general MapReduce programming model on multicore with use of ”tiling”. It attempts to optimize resource usage of data-parallel applications, with the argument that it is more efficient for MapReduce to iterate over smaller chunks of data instead of large ones on shared memory multicore systems. TMR partitions a large MapReduce job to several smaller sub-jobs and continues to process them with greater efficiency.

TMR extends the general MapReduce programming model and the resulting implementation is *Ostrich*. Fault tolerance is extended in Ostrich, with the support of backing up in persistent storage, results of sub-jobs in case of global failure or computational reuse. The changes in the model target shared-memory systems, with the smaller intermediate-data cycles aiming to reduce cache misses.

3.2.1.1 Evaluation

The evaluation of Ostrich on a shared memory machine for different applications and dataloads, revealed speed-up compared to Phoenix from 1.2x to 3.3x in all cases. The examined improvements were memory reuse, cache locality from the memory-aware scheduler and the relevance of iteration size, the benefit of the pipeline and fault tolerance. The system, however, is outdated in comparison with newer versions of Phoenix.

3.2.1.2 Other Similar Systems

There have been several more efforts to alter the MapReduce programming model in order to accomplish better shared-memory performance while retaining the simplicity and scalability benefits.

MATE [27] is an alternate framework, were the Map and Reduce phases are fused into a single MapReduce-type phase, changing the programming model to succeed in better performance. Ex-MATE [28] is an improved version of the system that supports arbitrary sizes of reduction objects. SciMATE [29] is a customisable system, which is developed to be adaptable on processing any of the scientific data formats. As the programming model has been changed, however, such systems are out of the scope of this project.

3.2.2 Multicore GPUs: Mars

Mars [30] is a MapReduce framework on graphic processors (GPUs). GPUs have an order of magnitude higher computation power and memory bandwidth compared to CPUs. Their architectures are designed as special-purpose co-processors, with a programming interface targeted at graphics applications. This renders GPUs even harder to program than typical many-core machines. Mars uses the familiar MapReduce interface to cover the programming complexity of GPUs.

3.2.2.1 Evaluation

The Mars framework was evaluated on GPUs against Phoenix. Six different applications and different data-sizes were evaluated in the study, and Mars performed from 1.5x to 16x better than Phoenix when the data set is large. The target hardware of GPUs, however, is out of the scope of this project.

The MapReduce framework has also been adapted in other non-CPU architectures as well. Successful efforts include FPGA [31] and the Cell B.A [32] architecture.

3.2.3 MapReduce on Heterogeneous Platforms

There are heterogeneous systems that make use of the MapReduce model. These systems pose significant interest for several reasons. First, they bridge different underlying hardware platforms over a common familiar framework, with good scalability, fault tolerance and performance.

3.2.3.1 MapCG

Both CPUs and GPUs are playing an important role in the parallel programming setting. MapCG [33] is designed to support a programming model which provides source code level compatibility between CPUs, and GPUs and different GPUs. The system allows programmers to execute the same code in either CPUs or GPUs, without modifications.

Evaluation was on a multicore machine with a GPU. Good scalability and speedup are achieved, while comparison with Phoenix-2 on the multicore machine reveals 2-3x speedup of MapCG. More experiments on the GPU, showed MapCG performing better than Mars. Running MapCG applications on both CPUs and GPUs at the same time, revealed same, if not degrading results.

3.2.3.2 Merge

The Merge [34] framework is a general purpose MapReduce-based programming model for heterogeneous multicore systems. It employs libraries to automatically distribute computation to heterogeneous cores, aiming for increased performance and energy efficiency. It provides a high-level library-oriented parallel language based on Mapreduce. It also encompasses a compiler and runtime, dynamically selecting and executing the appropriate function call per input and machine configuration.

The system is capable to to automatically map and load balance computations across all heterogeneous cores available in the computer system. The model has been successfully been evaluated on heterogeneous systems, but such use is not relevant for this project.

3.2.3.3 MITHRA

Finally, MITHRA [35] is a heterogenous framework of MapReduce, with the ability to use GPGPU accelerators in cluster nodes by deploying Apache Hadoop over NVIDIA CUDA technologies. It has been evaluated primary for scientific simulation problems.

3.3 Relational processing alternatives over distributed MapReduce

Improvements in the context of declarative query languages are interesting additions for consideration in the multicore version of the MapReduce model. Such languages have already been researched for distributed clusters. Examples of systems that provide equivalents to such languages are Microsoft SCOPE [36], Apache Pig [37, 38] and Apache Hive [39, 40].

SCOPE provides a type of SQL views, C# works on top of the Cosmos system, with the operators of Map-Reduce-Merge. Hive and the SQL-style language HiveQL are designed to provide data warehousing on top of Hadoop. Pig has been designed in the footsteps of Google's Sawzall [41] to support ad-hoc analysis of very large data.

Pig incorporates *Pig Latin*, a high-level dataflow language and its framework. Build on top of Hadoop, Pig breaks down a Pig Latin program to a query plan and then to consecutive MapReduce jobs [38].

Chapter 4

Design

Although Phoenix does follow the MapReduce programming paradigm from the distributed setting, there are fundamental differences when using such a system on a shared memory multicore environment. The differences of these two worlds - distributed and shared memory - pose different challenges over the requirements that need to be met and the bottlenecks that need to be confronted.

This is the reason that the relational operator algorithms written for distributed MapReduce, were redesigned specifically for the chosen system. Furthermore, a scheme was developed to allow easy and efficient piping between the operators. This scheme was attempted to be both comparable in speed and usability, against using the operators independently.

4.1 Design choices

Choosing between the different implementations of MapReduce systems on multicore architectures, and deciding upon the method to propagate records through the system, were the two decisions faced at the beginning of the project. The choices made, distinctly characterize the work.

4.1.1 Choice of underlying system

As discussed in section 1.3, the aim of this project is to use one of the existing systems enabling the distributed programming paradigm of MapReduce for shared memory multi-cores and extend relational processing over MapReduce, to work for one of these multicore implementations. From the aforementioned systems, Phoenix 2 was chosen

as the most appropriate for the development of the relational operators.

The Phoenix projects are the most extensive in term of use and best documented compared to all alternatives. Additionally, they are targeted for generic use, while presenting a good, stable behaviour. Their API and design are very similar, usually used as reference point from all other systems. This automatically renders their applications the easiest to port, not only between these three versions, but also between alternate platforms.

Phoenix++ is one of the newest systems, with the advantage of presenting some of the fastest results. On the other hand, Phoenix 2 is the most frequent reference in the evaluation between systems, as it was reported in section 3.2. Along with the preference to avoid the bloated and convoluted syntax of C++, Phoenix 2 was chosen over Phoenix++, as the underlying development platform for the relational operator library.

4.1.2 Record propagation through pointer manipulation

Implementing relational processing of records in MapReduce for the distributed setting, imposes certain unavoidable requirements. As the workflow of the model dictates in section 2.1, data processed in different nodes need to be sent between nodes for the shuffling phase. Therefore, records processed by relational operators, need to be copied through the cluster nodes, when using the algorithms of operators of section 2.4.3.

The Phoenix library expects that the user will allocate any memory his values may require, inside the user defined functions. The system is designed to manipulate only keys, keeping the values associated with those keys as mere pointer references. In the case of relational processing, following the distributed algorithms exactly allows two choices; either passing records as keys for the library to automatically handle according to its inner workings, or allocate new memory in the mapper to pass them as values.

In shared memory multicore machines, there is obviously no need to send data around, as the memory is uniform. Hence, the processed data through the system need not be deep copies of the initial records, but only pointer representations to them. Actually, Phoenix already makes use of this as the emitted data after the merging phase are pointer representations of key-value pairs with the actual data located elsewhere in memory.

The MapReduce model requires immutability of the input data and this restriction is kept in Phoenix. This along with the differences of shared-memory architectures,

have led to the decision to use an alternate method for propagation of records in the output. Records are passed to the model as values, but without a new memory allocation and a deep copy for each record.

The value pointers point to the immutable input array and the final key-value pairs consist of pointers pointing back to the input. The model is illustrated in Figure 4.1. There is, however, one exception. If a field is smaller than the size of a pointer, for example it is a single `char` or an `int`, than it may become efficient to create a deep copy.

This pattern was successfully followed in the test programs. New record structures, were a mixture of field pointers to the original data -when those fields exceeded 8 bytes in size for the 64-bit architecture- and actual deep copies of fields where created when they were smaller. Although, it is not clear whether allocating a smaller memory chunk and copying data is faster than allocating a larger sized pointer and pointing to the data, it is certainly more memory efficient.

Most operators are very well modelled with this method, even when used in cascading pipe fashion, as the data inside the records need not be altered. The advantage of not requiring deep copies of records is significant in terms of speed. Another positive side effect, is that the length of a record has a smaller impact in execution time. The only data-size factors that heavily influence speed, are the number of processed records and the size of the keys. Finally, this approach is more memory efficient, an important factor which is separately discussed in section 4.3.3.

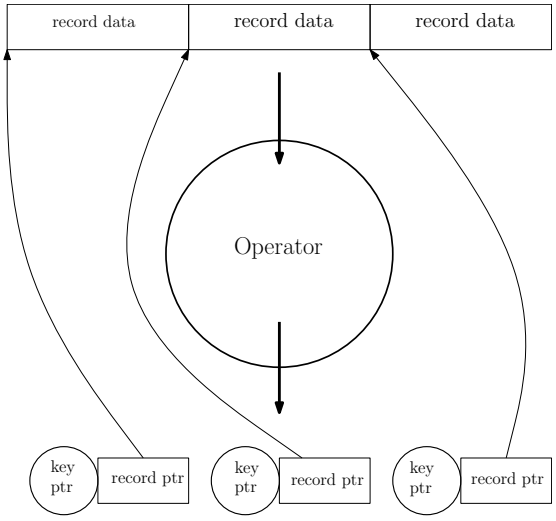


Figure 4.1: The model for propagating records with use of pointers on Phoenix 2

4.2 Unary operators over Phoenix

Redesigning the algorithms for Phoenix 2, often deviated from their distributed equivalents, examples of which were shown in Figures 2.3 and 2.4. The algorithms for 6 unary relational operators represent the elements of selection, projection, sorting, hash-partitioning, aggregation and finally secondary sorting. The workflow of a generic operator is described in Figure 4.2.

4.2.1 Select

The algorithm for selection is relatively straightforward. For every extracted record from the *chunk* appointed to the mapper, the predicate condition is checked. If the predicate is true, the record is emitted as the value, while the adjacent key can be anything. An interesting way to utilize the key, is to return an incremental record id in it, to guide the sorting of records in the output. The process continues until all records in the chunk have been examined.

The reducer needs only to propagate the key-value pairs to the merging phase. The record pointer in the final key-value pair, is pointing to the place the record was allocated in the initial input array. Pseudo-code for the algorithm is provided in Figure 4.3.

```
1  function Mapper
2          while (more records)
3                  extract a record
4                  if the predicate is true
5                          emit:
6                          key -> something (i.e. incremental number)
7                          value -> a pointer to the record
8
9  function Reducer                  //identity reducer
10         for keys in reduce task
11                 for value in key
12                         emit (key, value)
```

Figure 4.3: The designed selection algorithm for the MapReduce model on Phoenix 2

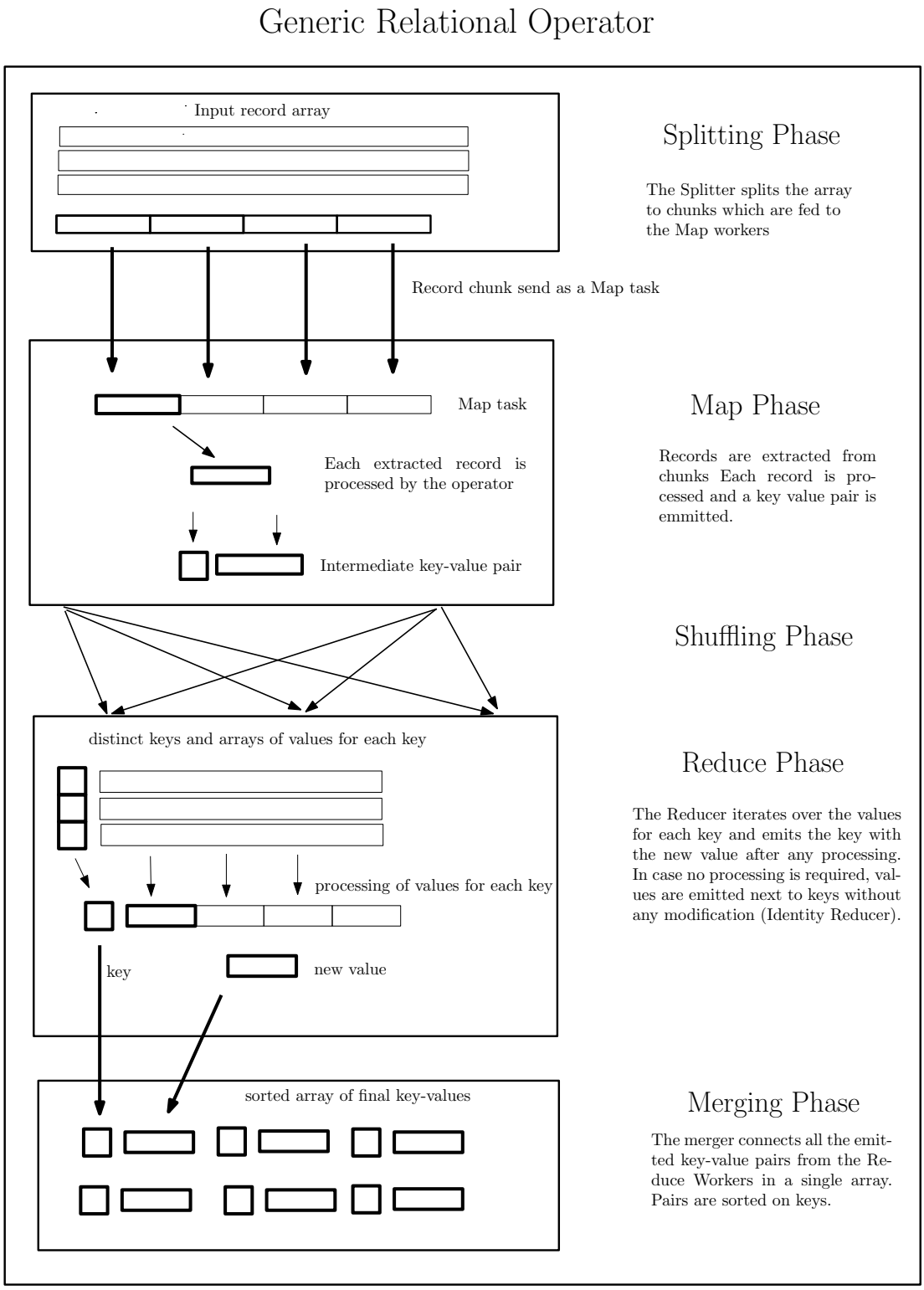


Figure 4.2: The Phoenix MapReduce workflow of a generic relational operator manipulating records. Every phase of the MapReduce model is maintained, along with the extra *merging* phase to unify the outputs of the reducers.

4.2.2 Project

Projection is somewhat more complicated than selection. Again a record is extracted from the current chunk processed by the mapper and for each record, a pointer structure is allocated to hold the projected fields. The pointers are meant for each of the projected fields and are initialized to point at the beginning of the field they represent in the record.

A pointer to the new record structure is emitted as a value, with a key that, is inconsequential for projection. The reducer is an identity reducer, feeding key-value pairs to the merging phase, without imposing any alterations. Figure 4.4 describes the algorithm in detail.

```
1  function Mapper
2          while (more records)
3                  extract a record
4                  allocate a pointer structure with projected fields
5                  point to the required fields in the record
6                  emit as:
7                          key -> something (i.e. incremental number)
8                          value -> a pointer to the new structure
9
10 function Reducer                  //identity reducer
11         for keys in reduce task
12                 for values in key
13                         emit (key, value)
```

Figure 4.4: The designed projection algorithm for the MapReduce model on Phoenix 2

4.2.3 Sort

Sorting of records is almost entirely handled by MapReduce. Each extracted record in the map task is emitted as a value, while the adjacent key is used as the sorted element that will determine the records place in the output array. The process is repeated until all records are emitted with the appropriate key from the mapper. An identity reducer is used as the algorithm for the sorting operator depicts in Figure 4.5.

```
function Mapper
        while (more records)
                extract a record
                        emit:
                        key -> key to sort records
                        value -> a pointer to the record

function Reducer                //identity reducer
        for keys in reduce task
                for values in key
                        emit (key, value)
```

Figure 4.5: The designed sorting algorithm for the MapReduce model on Phoenix 2

4.2.4 Hash-Partition

To achieve hash-partitioning, extracted records in the mapper need to be hashed by a hash function and then partitioned, according to their hash value. Each record is emitted as value to the intermediate MapReduce shuffling phase, while the key is the number of the partition the record belongs to.

The reducer is an identity reducer, while the values in the emitted output pairs are record pointers to the initial input. Figure 4.6 describes the algorithm.

```
function Mapper
        while (more records)
                extract a record
                apply the hash function
                decide partition over hash value
                        emit:
                        key -> number of partition
                        value -> a pointer to the record

function Reducer                //identity reducer
        for keys in reduce task
                for values in key
                        emit (key, value)
```

Figure 4.6: The designed hash-partitioning algorithm for the MapReduce model on Phoenix 2

4.2.5 Aggregate

The aggregation operator uses the mapper to emit a key to groupby key for each record and the corresponding value. The combiner function applies the aggregation routine to all values with the same key at the output of each mapper. Finally, for every key assigned to the reducer, the same aggregation is applied to the partially aggregated results.

The output pairs have the groupby values as keys and the result of the aggregations as values. Figure 4.7 outlines the aggregation algorithm in detail.

```
1  function Mapper
2          while (more records)
3                  extract a record
4                          emit:
5                          key -> groupby value (i.e. a record field)
6                          value -> value to be aggregated
7  function Combiner
8          for values in key
9                  apply aggregation to value
10                 return partially aggregated value
11 function Reducer
12         for keys in reduce task
13                 for values in key
14                         read partially aggregated value
15                         apply aggregation function to value
16                 emit:
17                 key-> groupby value
18                 value-> aggregated result
```

Figure 4.7: The designed aggregation algorithm for the MapReduce model on Phoenix 2

4.2.6 Secondary sort

The secondary sorting operator was designed having in mind the use of preparing the data for a map-side join. The algorithm in Figure 4.8 explains the process. After extracting a record from the map task, the primary sorting key is emitted along with a pointer to the corresponding record as value and the process is repeated for all records in the task.

After the shuffling phase, the reducer allocates a separate container for every key he has been assigned to and inserts the values of this key to the container. When all records of the key are placed in the container, the records are sorted on the secondary key. The output pair consists of the primary key and the record, while every group with the same primary key are also sorted with respect to the secondary key.

```
1  function Mapper
2          while (more records)
3                  extract a record
4                          emit:
5                          key -> primary key to sort records
6                          value -> a pointer to the record
7  function Reducer
8          for keys in reduce task
9                  allocate a container for the key
10                 for values in key
11                         extract record
12                         insert record to container
13                 sort container with secondary key
14                 for records in container
15                         emit:
16                         key -> primary key
17                         value -> a pointer to the record
```

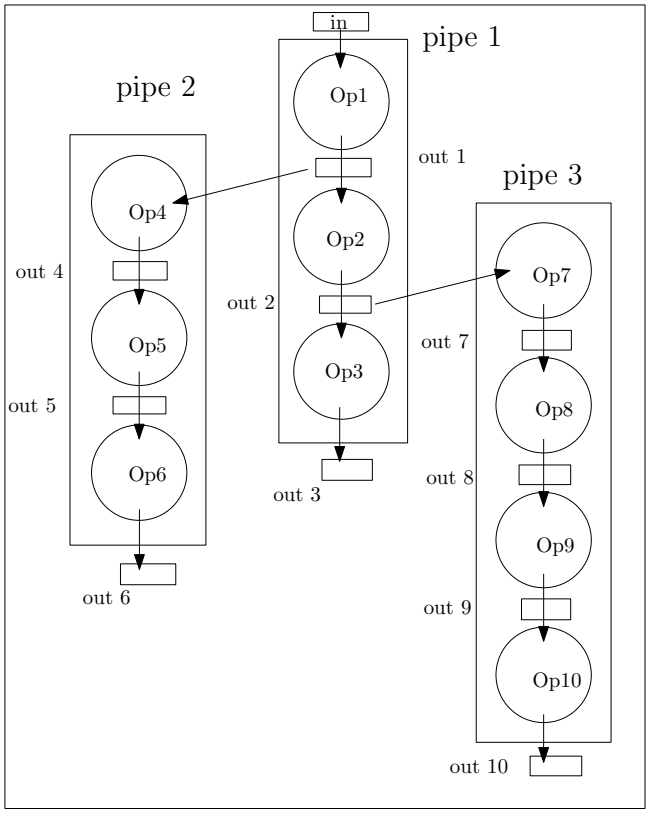
Figure 4.8: The designed secondary sorting algorithm for the MapReduce model on Phoenix 2

4.3 Combining the operators

As explained in section 2.4, relational algebra operators can be used to implement higher level languages and functionalities. A first step to that direction, is the use of operator combinations on the same data set, to accomplish more elaborate record processing. A part this projects work, is to enable the developed unary operators for multiple reuse in a serial pipe, allowing more complex processing of the input.

4.3.1 Multiple MapReduce jobs for piped execution

The designed library plans on using the operators in a cascading piped fashion, with the output of one operator used as the input of the next in the pipe. Operators can be combined in any possible order, any number of times in the same pipe. The library allows the user to branch out new pipes, to any point of an existing one. Figure 4.9 illustrates.



pseudo-code form

```
/* pipe 1 */
Op1 (in, out 1)
Op2 (out 1, out 2)
Op3 (out 2, out 3)

/* pipe 2 */
Op4 (out 1, out 4)
Op5 (out 4, out 5)
Op6 (out 5, out 6 )

/* pipe 3 */
Op7 (out 2, out 7)
Op8 (out 7, out 8)
Op9 (out 8, out 9)
Op10 (out 9, out 10)
```

Figure 4.9: The use of operator pipe branching. On the left, three branches starting at different times on different input data, create different processed results. On the right, the form of the pseudo-code, using the operator library.

Each relational operator, executes a MapReduce workflow by utilizing the Phoenix library. The result of a pipe is a cascading series of multiple MapReduce jobs, that execute different relational operations, on different forms of the initial record data. Inside a pipe, an operator begins to process data right after the previous operator has finished its MapReduce execution.

Synchronization is achieved, by not allowing a Phoenix 2 scheduler instance start when another Phoenix 2 scheduler instance is not finalized for a given main thread of execution. In other words, two Phoenix 2 scheduler instances executing differ-

ent MapReduce jobs are serialized when scheduled to run on the same main thread. The thread pool, which is used only for scheduling workers, is instantiated once and reused by the different scheduler instances. Therefore, the overhead consists only of re-instantiating the Phoenix 2 scheduler.

In the case of a pipe, processed data by the first operator are merged into a single contiguous *intermediate array* in memory and until all results are allocated, they are not to be touched by the following operator in the pipe. As the initial input record array and all intermediate ones are immutable, a new pipe can begin from any point of a previous one, at any time, to form a new series of cascading executions. Phoenix 2 instances are scheduled to run serialized on the same main thread, so all pipes running on the same thread are also serialized, which ensures synchronization.

In theory, the design allows creating parallelly executing operators on different threads and therefore parallelly executing pipes, by running Phoenix 2 instances on different threads in parallel. Care must be taken, however, not to run an operator or a pipe on a thread, which accepts input created by another operator or a pipe on a different thread, without the second having finished allocating results.

This can be resolved, either by only forking threads to branch parallel operators, or by using barriers. In practice, testing Phoenix 2 scheduler instances running on different threads in parallel, has not been conducted in this project and consequentially, only the conceptual design of using parallel operators and pipes is presented.

4.3.2 Data manipulation

For the endeavour of piped executions to succeed, two requirements need to be met. First, the initial input array, the final results array and all intermediate arrays of a pipe, should be stored in a consistent form for the operators to read, regardless of their place in execution. The other solutions is for the operators to read multiple types of input.

This, however, would complicate the design of operators unnecessarily. More importantly, in the case of a future expansion of the library to include binary operators, this could potentially lead to incompatibilities to binary inputs. Therefore, using a consistent representation form was selected as most suitable and data are handled by the *preparation operator* first, before they are processed by the actual relational operators. The preparation operator is described in detail later, in section 5.3.1.

Second, piped execution must agree with the design decision of using pointer manipulation for record propagation, as described in section 4.1.2. As it turns out, pointer

manipulation is a very good fit for the model. As the results from each operator contain pointer references to the original data, record pointers resulting from multiple transformations, also point back to them.

In more detail, two types of operators were developed. The first does not change the form of records, so it outputs references to the original records. An example of such an operator is *selection*. The second type, changes the record form and to accomplish that, creates new record structures with altered fields, or with pointers pointing back to the input. An example of such an operator is projection.

In the example of a projection-selection pipe, the projection operator creates a new record structure, with pointers pointing to fields in the original records. The selection operator that follows, will consider as records the new pointer structure and access the original data fields through the new record. The results of selection will point to this new records. Figure 4.10 illustrates.

By taking these pointer meta-data into account, the user can manipulate records to access the raw data he requires. If he so wishes to create new fields for records, or change the contents of a field, he must allocate new memory and point to the new field with a new record structure.

Altering the raw input data is also possible, but it must be conducted with care, as it can potentially cancel the validity of all intermediate results in the pipe. Additionally, it slightly breaks the MapReduce model, which describes that input data is immutable.

4.3.3 Memory efficiency

As already discussed in section 4.1.2, the design decision to use pointer manipulation, has positive impact on performance in general and on performance with regard to record size. When extending the operators to be used in pipes, the issue of memory efficiency also gains important weight.

The reason is that, use of the MapReduce paradigm was intended to process Big Data of several gigabytes and the Phoenix 2 project was initially developed with such a scope in mind. The speed of Phoenix relies heavily on the allocation of data in memory. Memory, however, is not a cheap commodity as disk space is and should be used with more care.

Processing of several gigabytes of records that require equal amount of memory to store results is memory inefficient. Even more, creating a chain of MapReduce jobs with intermediate results of several gigabytes is almost without merit. And arranging

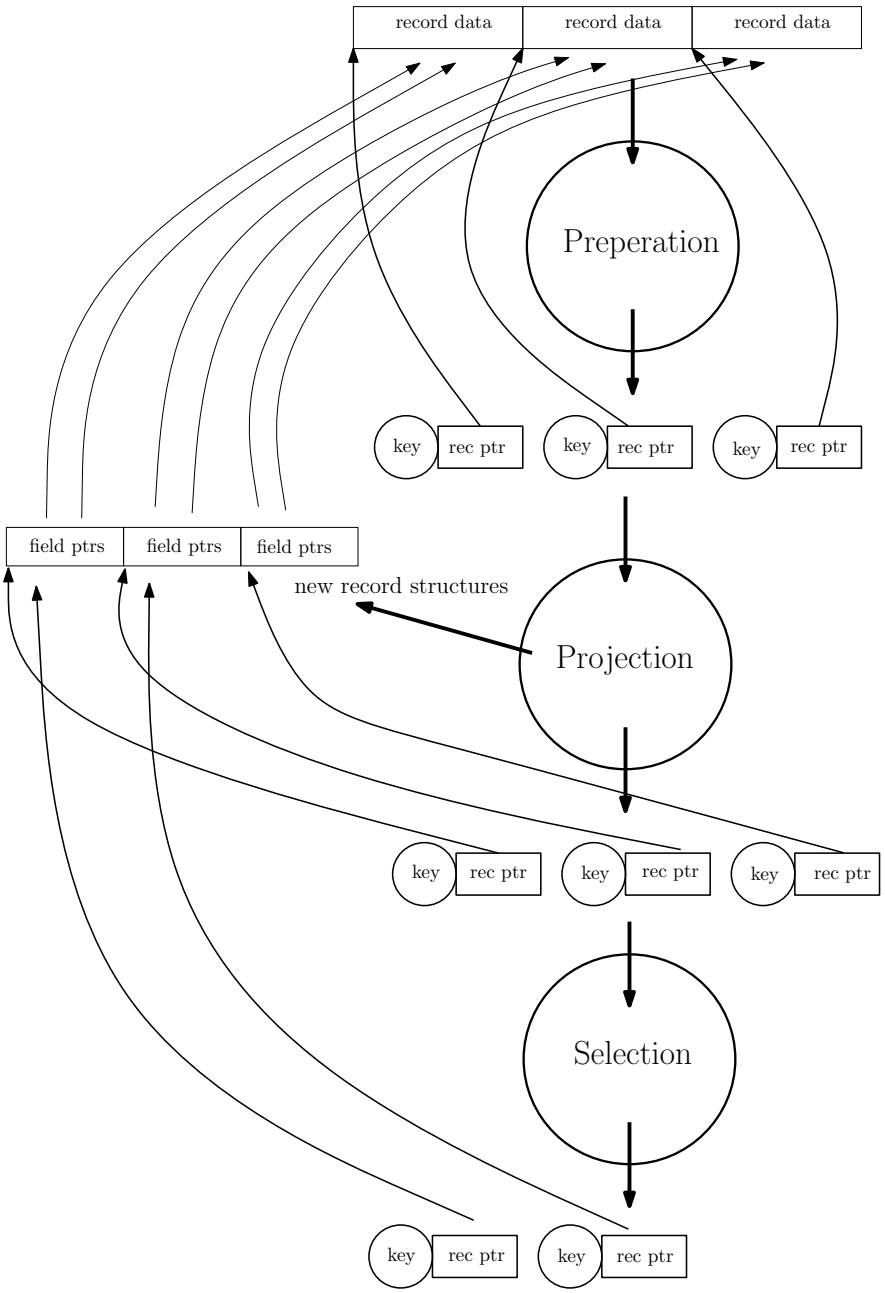


Figure 4.10: The model for propagating records with use of pointers, on cascading operators. The example is a selection after a projection, with data preparation as the first operator in the chain. Pointers always lead to the original record data.

frequent memory deallocations to be scheduled is a performance limiting factor.

Chapter 5

Implementation

The design decisions from the previous chapter were taken into account and the resulting operators form a new library on top of Phoenix 2, called Relational Phoenix. The relational operators were implemented in `C`. Relational Phoenix, provides a new API over Phoenix 2, with the ability to pipe all different operators and maintain the input order of the records when the user requires.

5.1 Data I/O

The user must provide an input file or a single array in memory, with records in a particular form. Loading and unloading records in memory is handled by the library. Special data handling allows piping the operators as required.

5.1.1 Record Requirements

The library can handle records in specific form that the user must provide. Records must be stored next to each other, in a single contiguous byte array, for the library to load them into Phoenix 2. Each record is required to start with a four byte integer determining the *length* of the record and the record itself following again in byte form. Relational Phoenix is agnostic of the fields included in each record.

Extra fields can be added to each array by the *preparation* operator, which is described in section 5.3.1. However, the input records used for evaluation have already been prepared with two extra fields before the required *length* field.

Before the *length* field, a single byte field is used to describe the *table name* the record belongs to. *Table name* is described with a single character, while another four

byte integer follows containing the incremental number of the *record id*. Figure 5.1 illustrates.

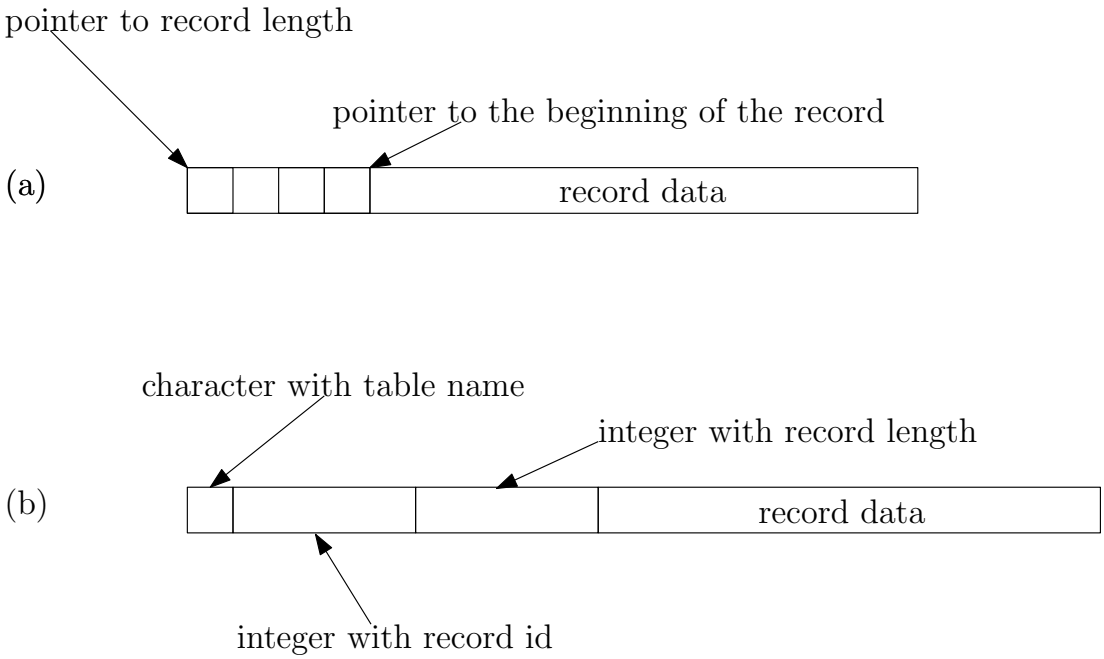


Figure 5.1: The form of the data that the library processes must be in the form of a continuous byte array of records. Their form can be of type (a), for evaluation purposes type (b) is used in test benches.

5.1.2 Loading/Unloading Data

The `load_op`, `unload_op` operators are used for disk I/O and initialization of the Phoenix 2 library and should be called first and last respectively, when using Relational Phoenix. These operators are serial and do not utilize Phoenix for data processing. The `load_op` initializes the thread pool of Phoenix 2, reads from the input file and stores it in a single contiguous array in memory.

To load data into memory, `mmap()` is used. For the large input files and relatively low thread numbers Relational Phoenix is tested on, it is the best alternative. For large numbers of threads, however, when many threads try to fault in data pages, `mmap()` performance can deteriorate. If such behaviour is noticed, in case of a small size input file, `load_op` can be tuned to use `read()` to load files into memory.

The `load_op` operator returns a pointer to a structure of type `filedata_t`, which contains information about the input data, like their start and their length. A handle to the opened file is also returned to allow further manipulation of the file. A counter on

the input data length and the length of each record -approximately if necessary in the case of records- are included to help the splitting of map tasks later.

The `unload_op` operator finalizes Phoenix 2 and closes the opened input file, but does not free the memory used by the input record array. The initial array is not deallocated, as it holds the base data that the results point to, in regard to the design decisions discussed in section 4.3.2. Figure 5.2 depicts the two operators signatures.

```
1 filedata_t *load_op (char *fname,
2                       size_t record_length)
3 void unload_op (filedata_t *fd)
```

Figure 5.2: Signatures of the load and unload operators.

5.1.3 Piping Data

A useful functionality of relational operators is using them in combinations, as discussed in section 4.3. The presented difficulty, is that every operator must be aware of the type of input that it processes. As initially loaded data are stored in a contaguous array and indexed by a `filedata_t` structure, results of the Phoenix 2 library are stored in a different type of structure `final_data_t`, in key value form.

Therefore, depending on which operator is scheduled first, different data handling must occur. To avoid this, the `prepare_op` operator is run first and transforms data to the form of `final_data_t` and make it consistent with the intermediate forms emitted by a cascading chain of piped operators as discussed in the design. The `final_data_t` structure type, contains the number of key-value pairs contained in the results and a pointer to the first key-value.

Key-values are stored next to each other, again in a single array, ready to be processed by the next operator. Key-values are of type `keyval_t`, which consist of pointers to the key and value of the particular pair. As a result the key-value pairs have fixed length, which simplifies the process of splitting them for the next operator.

5.1.4 Maintaining record order

As the Phoenix 2 library processes records in parallel, the order of records of the output is shuffled. However, the added field of *record id* is added in front of every record to

provide the feature of maintaining input order in the output.

Therefore, preparation of data, selection and projection, can all maintain record order according to the *record id*. The activation of the feature is possible by setting the *input_order* flag in the API of these three operators.

Disabling key sorting is only supported in Phoenix++ and not in Phoenix and Phoenix 2. So, maintaining the order of records is kept as default. Disabling was implemented for Relational Phoenix by oveerriding the system and with the consequence of overwriting some of the keys in the key-value pairs. However, this has no impact on the records emitted from the operators and speeds up execution.

5.2 Operator Skeleton and API

The operators of Relational Phoenix were implemented in respect to the requirements posed by the Phoenix 2 library. A new API was developed and the result is that Phoenix 2 is completely hidden from the user of the operators, who must now write his programs with the new API of the the operator library.

5.2.1 General skeleton of the operators

The basic skeleton of every operator, consists of several functions for the Phoenix API, that are to be provided to the runtime along with the initialization of several arguments. The scheduler is then ready to be called into action.

The skeleton consists of several elements. For one, it carries the arguments provided by the user. Function pointers are initialized, so that the user provided arguments are passed to the Phoenix API routines. A number of arguments are also set, to provide initializations and fine tuning of the Phoenix 2 library. Such arguments are the number of processors to use, the number of mapper and reducer threads plus L1 cache sizes. Several other, tune the number of reduce tasks, estimate the input to output size ratio and the use of queues per map and reduce task.

There are two types of splitters used in the basic skeleton, one that splits the record array initially provided by the user and loaded into memory by the `load_op` and another that handles data already processed by operators and stored in key-value form.

5.2.2 General design and use of the API

The Relational Phoenix library API is designed so as to completely hide the underlying Phoenix system, providing MapReduce parallelism under the hood and a higher usability for the programmer of relational data processing applications. A typical signature of an operator used to process data, is depicted in Figure 5.3.

A program written in the Relational Phoenix library starts with the `load_op` followed by the `prepare_op` and then the operators the programmer wishes to use. Simply by redirecting one operator output to the input of another, produces a piping effect. As already discussed in section 4.3. Finally, calling the `unload_op` will terminate the library. Figure 5.4 lays out the form of a program written with the library.

```
1 void general_op (final_data_t *input,
2                   type1_t (*function1_t)(type1_t *ptr),
3                   type2_t (*function2_t)(type2_t *ptr),
4                   ..........................
5                   final_data_t *output,
6                   int maintain_input_order_flag)
```

Figure 5.3: General signature form of an operator. The signature of every operator also acts as its respective API, for the Relational Phoenix library.

5.2.3 Selection without data preparation

The *preparation* operator (as presented in 5.3.1) handles initial data presented by the `load_op`, but unavoidably produces unnecessary overhead by conducting an extra MapReduce job. To evaluate this overhead, a selection operator that handles raw data was developed. This *select raw* operator works on the initial data, without the need of any preparation. Benchmarks are included in the evaluation, to compare with the typical selection operator.

The difference is that this selection incorporates a more complex splitter, similar to the one in the *preperation* operator. It was abandoned against the choice of having data prepared first, a design decision that favours simplicity. The signature of the operator is outlined in Figure 5.5.

The `select_op_raw` accepts an array of records of type `filedata_t`. The `sel()` function pointer argument should be a user provided function, that states the selection

```
1  function1()
2  function2()
3  .....
4  functionY()
5  functionZ()
6
7   main(){
8          filedata_t *fd;
9          final_data_t results;
10         final_data_t results2;
11         ...........
12         final_data_t resultsN;
13
14
15         fd=load_op (fname, record_size);
16         prepare_op(fd, &results, 1);
17
18         operator1_op(&results, function1, &results2, 1);
19         operator2_op(&results2, function2, &results3, 1);
20         ........................................
21         operatorN_op(&resultsM, functionY, functionZ, &resultsN);
22
23         unload_op(fd);
24 }
```

Figure 5.4: Example C-like algorithm written with the Relational Phoenix library. Function definitions are on top, with result structures allocated at the beginning of `main()`. The library is initialized and the data is prepared to be used by the operator pipe depicted afterwards. Finalization of the library is achieved when calling `unload_op`.

predicate. It accepts a pointer to a record and should return *true* when that predicate is met. The output is an array of key-value pairs in the form of a `final_data_t` type. When the `input_order` flag is enabled, the selected records keep the same order as the input.

```
1 void select_op_raw (filedata_t *in,
2                     int (*sel)(void *rec),
3                     final_data_t *op_results,
4                     int input_order)
```

Figure 5.5: Signature of the selection operator without data preparatiion

5.3 Piped operators

There are *five* relational operators implemented, that can be piped in any order. The function signature for every operator, is also the API of the library for the particular operation. As the Relational Phoenix library is agnostic of the internal fields of records, user provided functions usually accept a record pointer as an argument and manipulate the record fields internally.

5.3.1 Preparation operator

The *preparation* operator is used to prepare the input data in the key-value form that the Phoenix system emits. This is helpful, as it eliminates the need of reading into records to determine their size, when splitting them to map tasks. But most importantly, the relational operators can be piped in any order, as after preparation stage all input/output arrays are of the same `final_data_t` type.

Additionally, this operator can be useful to add fields to each record, before records are processed by the actual relational operators. These extra fields can for example be *table name* and the incremental *record id*, which is used to keep the ordering of the records. In general, this operator could potentially serve to pre-process the records in several ways, but such uses are out of the scope of this project.

The signature of `prepare_op` is shown in Figure 5.6. The input is data loaded from file by `load_op` of type `filedata_t`, and the output is of the uniform type of result data

final_data_t. If the input_order flag is set, records maintain their original places in the array.

```
1 void prepare_op (filedata_t *in,
2                  final_data_t *out,
3                  int input_order)
```

Figure 5.6: Signature of the prepare operator.

5.3.2 Selection operator

Just like the selection operating on *raw* data in section 5.2.3, the selection operator processes records based on the sel() user provided function. The difference is that the splitter in this version accepts prepared data of final_data_t type.

The output is naturally of final_data_t, while the input_order flag allows maintaining the record order of the input. The signature of the operator function, also acting as a part of the API, is on Figure 5.7.

```
1 void select_op (final_data_t *in,
2                 int (*sel)(void *rec),
3                 final_data_t *out,
4                 int input_order)
```

Figure 5.7: Signature of the selection operator.

5.3.3 Projection operator

The projection operator accepts final_data_t type input and emits of the same type. The input_order flag has the same functionality as in selection. The prj() user defined function accepts a pointer to a record as an argument and returns a pointer to the new record with projected fields. It is possible for this function to allocate far less memory for each new record, by implementing it having pointer manipulation in mind. This saves the effort from copying the fields to a new memory location. Figure 5.8 describes the signature of the operator.

```
void project_op (final_data_t *in,
                 record_t *(*prj)(void *rec),
                 final_data_t *out,
                 int input_order)
```

Figure 5.8: Signature of the projection operator.

5.3.4 Sorting operator

Sorting uses the same input/output types as the rest of the piped relational operators. The `srt()` user defined function accepts pointers to compare between two keys. It acts as a comparator, returning zero when the keys are equal, a positive integer when the first is larger than the second, or a negative number when it is the other way around.

The `key_prt()` and `key_size()` functions should also be provided by the user, and must return a pointer to the key records will be sorted by and the size of this key, for each record. Figure 5.9 has the detailed syntax of the operators callable function signature, which also acts as the API of this operator.

```
void sort_op (final_data_t *in,
              int (*srt)(const void *k1, const void *k2),
              void *(*key_ptr)(void *rec),
              int (*keysize)(void *rec),
              final_data_t *out)
```

Figure 5.9: Signature of the sorting operator.

5.3.5 Partitioning operator

Partitioning of records is achieved by returning the number of the partitions each record belongs to in the key field of output pairs. Outputted records are sorted by the number of the partition. The user provided `hsh_prt()` hashes and returns the partition of each record it accepts in integer form. The signature of the operator is depicted in Figure 5.10.

```
1 void partition_op (final_data_t *in,
2                    int (*hsh_prt)(void *record),
3                    final_data_t *out)
```

Figure 5.10: Signature of the partition operator.

5.3.6 Aggregation operator

Aggregation of records is possible with the `aggregate_op`. User functions should provide the pointer and the size of the key on which to group-by, by implementing `key_prt()` and `key_size()` similarly to `sort_op`. A comparator to compare keys should also be provided again as in the case of `srt()`, in section 5.3.4. The `aggr_val()` function, should return a pointer to the aggregated value for each record it receives.

The extra feature in this operator the `aggr()` function pointer passing an aggregation function on the values grouped by key after the shuffling phase. An iterator interface is exposed to the user with which he can iterate over the values of each key. The aggregation function will be applied to each distinct key separately, so the user needs not to worry about iterating over the keys. The operator signature is depicted in Figure 5.11.

It is the users responsibility to allocate any space in memory for the aggregated values and results. There is however an alternative, if the aggregated value or result is smaller or equal to the memory allocated for a `C` pointer (i.e. 8 bytes when referring to a 64-bit system). In that case it can be stored in the memory of the pointer itself. This has been implemented in the evaluated example and has the speed advantage of not requiring any extra memory allocations, which is a common slow-down factor. Values written in the memory of a pointer, can be read for example by using the `intptr_t` data type, with an aggregated value that can be served by an integer.

```
1 void aggregate_op (final_data_t *in,
2                    void *(*aggr)(iterator_t *itr),
3                    int (*srt)(const void *k1, const void *k2),
4                    void *(*key_ptr)(void *rec),
5                    int (*keysize)(void *rec),
6                    void *(*aggr_val)(void *rec),
7                    final_data_t *out)
```

Figure 5.11: Signature of the aggregation operator.

Chapter 6

Evaluation

The implemented relational operators were tested on three different benchmarks for two datasets on two different machines of alternate architectures (*Xeon* and *i5*). Experiments varied the number of threads and size of the input for all the operators individually as well ass aligned in a pipe. Several other settings were also tested, like disabling sorting of the input and other internal settings of Phoenix 2.

A clear and organized approach of scientific experimentation was attempted during the evaluation. A methodology was developed and the design of the experiments with the purpose of evaluating the Relational Phoenix library.

6.1 Methodology

The key factors around which the evaluation of Relational Phoenix was based, are speed over size of input, scalability over cores and usability. As exhaustive evaluation was not a feasible goal, experimentation was chosen selectively, to evaluate particular hypothesis questions, posed on the above points.

Experiments were conducted with the use of scripts, to assure that the procedure would be exact for every experiment. Every experiment was repeated a number of times to acquire a better sample for statistical analysis. It was judged as sufficient to estimate the average and standard deviation for the conducted experiments. Measures where also taken to ensure validity -and to an extent accuracy- of the timing results.

6.2 Experiment setup

The two data sets used for evaluation, differ in record cardinality by an order of magnitude. The experiments try to measure the efficiency of the operators for inputs that vary significantly in record count. Record size has a small impact, because of the design decisions discussed in section 4.1.2. Record size was set to approximately 100 bytes, even after using projections.

The smaller data set consists of 10 files, varying in record cardinality from one hundred thousand to one million records, incrementing with a step of a hundred thousand records between files. The larger data set, which was chosen as the primary data set, is structured in a similar way, starting from one million records for the smallest file, to ten million for the largest one. The largest file, exceeds 1 GB in size.

To evaluate parallelism, experiments were run with different thread numbers. The small data set was run on a personal computer of 2 cores with hyperthreading and 4 GB of memory, for up to 4 threads, while the large one was tested on a server with 2 slots of quad processors with hyperthreading, for a maximum of 8 threads. Separate experiments for up to 16 threads were also conducted. The specifications and load of the target machines are included in section 6.2.1.

The operators were evaluated separately, as well as in a pipe. Separate testing was conducted for the five different operators, the preparation operator, and a sample selection operator that accepts raw input data that need not pass from the preparation operator. The operator pipe was tested with all five different operators in the same linear pipe, with data prepared first by the preparation operator. Finally, tests were conducted to evaluate the operators without the *maintain-input-order flag* and different settings of the Phoenix 2 library, like the *one-queue-per-task flag*. These tests were conducted with the small data set on the personal computer and their results also presented. The fields of the generated records and the functionality of the operators over them, are described in section 6.2.2.

6.2.1 Target machines

The experiments of this project were run on the `student.compute` server and a personal computer. Their specifications are presented next in more detail.

6.2.1.1 The `student.compute` server

The `compute` server is a *Dell Poweredge R610s* with `48GB` of memory and `2.66GHz` processors. In particular it consists of two sockets of quad-cores with *Intel Xeon X5550* CPUs, with *Hyperthreading* enabled. Cache and memory configuration includes:

- `64 KB L1 cache/core (32 KB L1 Data + 32 KB L1 Instruction)`
- `4 x 256 KB L2 Cache`
- `8 MB L3 Cache`

On the period experiments were run, the server presented the *load* of Table 6.1.

Table 6.1: Server load

load information	on average
tasks	*150*
threads	*330*
running	*25*
load	*30*

6.2.1.2 The PC machine

The second machine is a laptop running `64-bit Linux` with `4GB` of main memory, an *Intel Dual Core i5 M450* at `2.40GHz` and *Hyperthreading* enabled. Specifications:

- `64 KB L1 cache/core (32 KB L1 Data + 32 KB L1 Instruction)`
- `2 x 256 KB L2 Cache`
- `3 MB L3 Cache`

6.2.2 Records and Test Program

Records generated in test files are of fixed size and containing fixed fields. The tested operators used and their basic functionality were chosen so as to evaluate different types of fields. In cascading operator execution, the cardinality and size of records were only slightly changed during processing, so as to evaluate all operators fairly.

6.2.2.1 Records

The records in the data consist of five fields, which vary in size from 1 to 80 bytes and include different types. In particular, the *key* field contains random integers from 0 to 100, while the *record id* contains unique integers incrementing to the size of the record cardinality. The two fields were chosen, so as to test system behaviour in selecting and sorting.

Similarly, while the *data* field is an 80-byte long string consisting of a random sequence from all the letters in the alphabet, the *value* field is a string of 10 characters created only by random permutations of two. In the second case, that reduces the possible combinations of different strings to only 1024. This field seen as a key and used for sorting, partitioning or comparing, will have a different impact on the library. All the different fields are outlined separately below.

- *table name* - field of size `char`.
- *record id* - field, incremental number of size `int`.
- *length* - field, length of the record of size `int`.
- *value* - field, a 10-byte `char` array consisting of a random sequence of two letters.
- *key* - field containing a random integer from `0` to `100`.
- *data* - field containing a random sequence of the alphabet letters, 80 bytes long.

6.2.2.2 Test program

The operators used in the test program were implemented so as to test the robustness of the library, by manipulating different types and sizes of data. Their functionality is presented below.

- `select_op` selects records if their *key* field is larger than zero (99% of input).
- `project_op` projects all fields except *key*.
- `sort_op` sorts records according to the contents of their *value* field.
- `partition_op` hashes the *value* field and then partitions each record in 4 partitions.
- `aggregate_op` counts the number of occurrences for each of the *value* fields.

6.3 Impact on cardinality

The first benchmark attempts to evaluate the operators separately, over an increasing number of input records. The large data set run on the `student.compute` server is used, with similar results reported for the small dataset, which is included in the Appendix.

The graphs on Figure 6.1, depict the average time each operator required to process the input file in nanoseconds, with different bars indicating a different file. As described in section 6.2, the 10 input files contain from 1 million to 10 million records. All tests were repeated 10 times for each input file on every operator. Standard deviations are included as error bars in the graphs, using 4 threads.

All operators, except the `select_raw_op`, require the `prepare_op` to prepare input data before processing it. Measurements represent the execution time of the relational operators after the `prepare_op`, which is separately shown on Figure 6.1a. Therefore, when comparing the `select_raw_op` on Figure 6.1g with `select_op` on Figure 6.1b, it is required to add the overhead of the `prepare_op`.

6.3.1 Evaluation

According to the test results, all operators have good behaviour with varying number of records in the input. This has also been confirmed for the small data set, which was utilized for testing of all three benchmarks on the second machine. Standard deviation is acceptable in regard to the mean average. The fact that it appears to increase with the size of the input, can be attributed to the increased number of context switches.

As expected, the `select_op` with the overhead the `prepare_op`, is almost twice as slow as `select_raw_op`. On the other hand if examined on its own, `select_op` is faster than `select_raw_op`. This is an interesting observation, in case of a long series of cascading operators in a pipe.

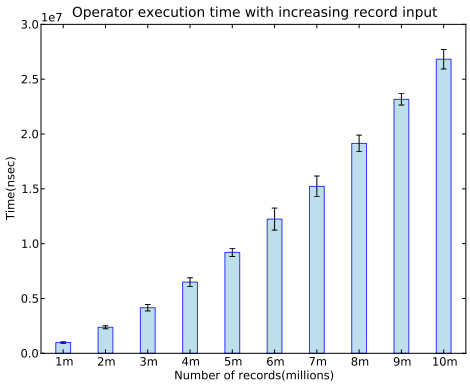
Another observation is that the graphs of Figures 6.1a, 6.1b, 6.1c and 6.1g tend to increase slightly exponentialy, while the operators on Figures 6.1d, 6.1e, 6.1f seem linear. The reason behind this trend, is the difference of unique keys that the first group needs to sort in comparison to the second (record fields and the functionality of tested operators are described in section 6.2.2).

Selection, projection and preparation operators, maintain the input order of records on the output as a default option during testing. This requires sorting of unique keys up to the number of records. On the other group, sorting, partitioning and aggregation

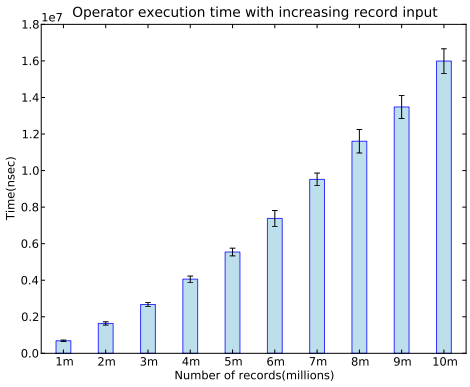
only use a small number of keys, up to 1024, for any input size.

Having come across the difficulty of Phoenix 2 in sorting to sort unique keys in the past, the results are justifiable and can be explained, when the operations in the shuffling phase are examined more closely. The intermediate data structure of the Phoenix system is described in section 2.2.2. To shed sorting of keys in the intermediate phases, the *one-queue-per-task flag* is set for the experimental procedure. This also fixes the number of *reduce tasks* to 256.

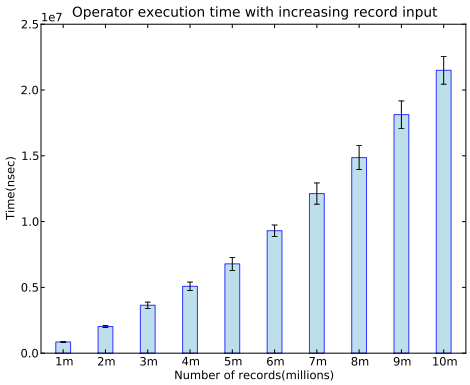
Entering a large number of unique keys distributes many keys to the same reduce task. Even with intermediate sorting disabled, the time to look up a key in each cell of a row that represents the reduce task, will increase for more keys hashed in the available 256 rows. Additionally, for larger input size the splitter assigns more columns to the intermediate array. This pattern explains the observed increase.



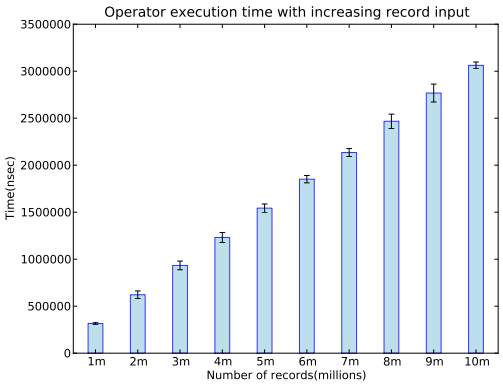
(a) prepare_op



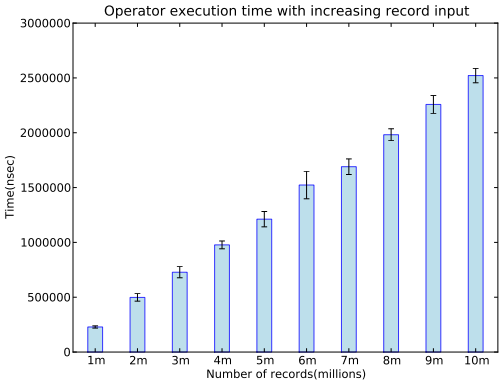
(b) select_op



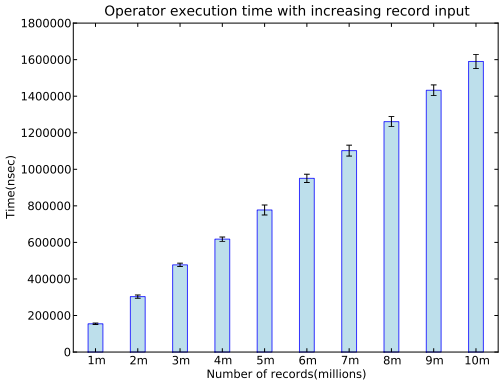
(c) project_op



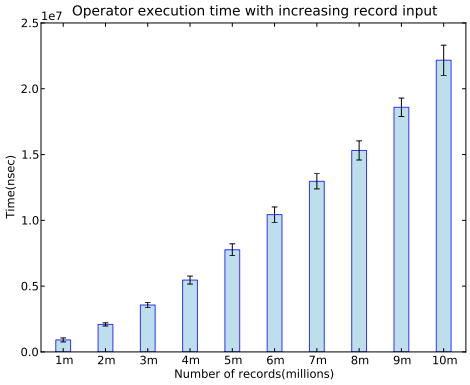
(d) sort_op



(e) partition_op



(f) aggregate_op



(g) select-raw_op

Figure 6.1: Impact on cardinality for individual operators on the large data set, with varying record input. 10 runs with standard deviation.

6.4 Impact of multithreading

The second benchmark attempts to evaluate individual operator performance separately, over an increasing number of threads. The large data set run on the `student.compute` server is depicted, with similar results reported for the small dataset, included in the Appendix.

The graphs on Figure 6.2 depict the average time each operator required to process the input file of 5 million records in nanoseconds, with different bars indicating different thread numbers. All tests were repeated 10 times for each separate thread number on every operator. Standard deviations are included in the graphs.

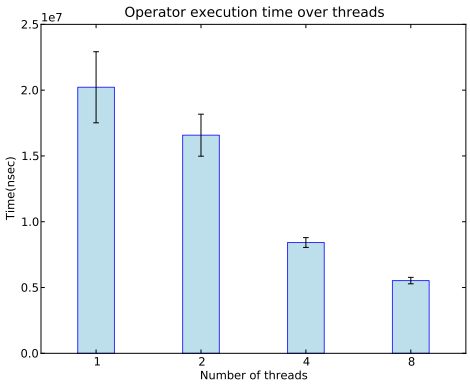
Evaluation

The results of the second benchmark clearly demonstrate that all the operators successfully utilize threads to achieve the desired parallelism. Again the increase of deviation can be attributed to context switches. There is, however, a variance between speedups, especially over 4 threads.

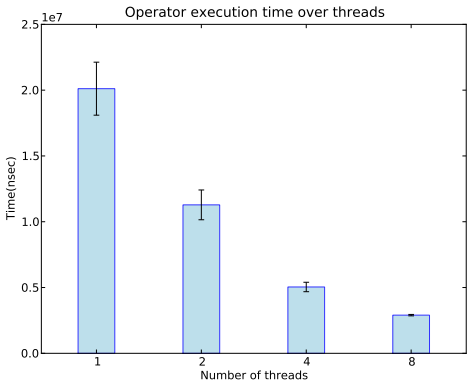
Often this variance is within the error margins of standard deviation, like in the case of two threads for the `prepare_op` in Figure 6.2a. Moreover, the increased load on the processors of the server (Table 6.1) can contribute to such deviations. Another explanation applies in the cases above 4 threads, where the observed mean diverges significantly from the expected.

As described in the specifications of the `compute` server in section 6.2.1, the targeted machine does not host an 8-core processor, but two sockets of quad-cores with hyperthreading. Phoenix 2 was calibrated to work with the L1 cache of the machine for the experimental procedure, but Phoenix does not provide for selecting cores in the same socket.

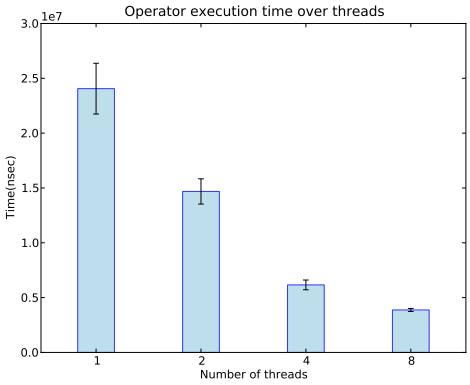
Setting aside the fact that hyperthreading in not equivalent to physical cores in terms of performance, it is possible that common L3 cache might not always have existed during testing. This can largely increase the I/O cost, as accesses to main memory are more expensive than access to cached data. Further testing suggested, that the library cannot utilize parallelism effectively for more than 8 threads on the target machine.



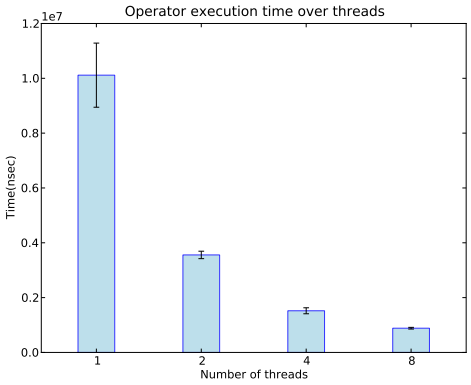
(a) prepare_op



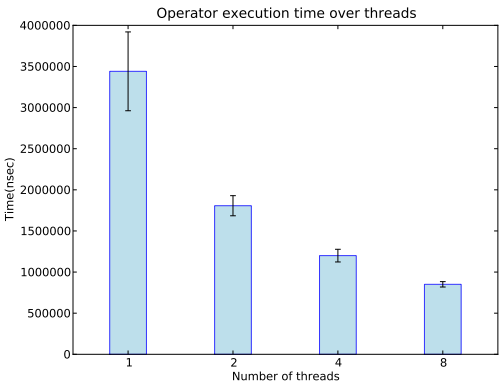
(b) select_op



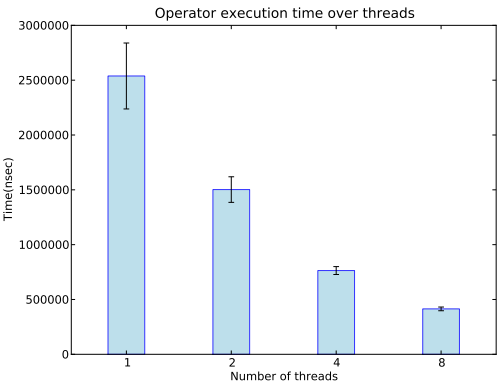
(c) project_op



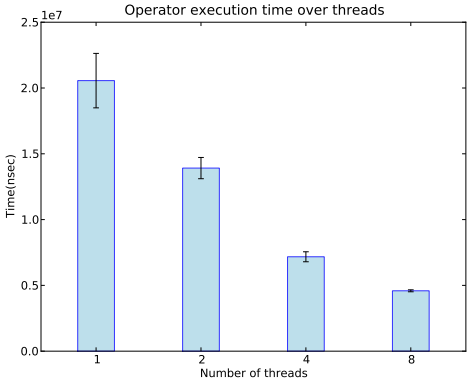
(d) sort_op



(e) partition_op



(f) aggregate_op



(g) select-raw_op

Figure 6.2: Impact of multithreading, individual operators for different threads.

6.5 Behaviour of pipes

The third benchmark attempts to evaluate the operators in a linear pipe, over an increasing number of input records and an increasing number of threads. The large data set run on the `student.compute` server is depicted, with similar results reported for the small dataset. The pipe includes the following series of operators:

1. `prepare_op`
2. `select_op`
3. `project_op`
4. `partition_op`
5. `aggregate_op`

In the case of increasing record input, the graphs of Figure 6.3 depict the average time the pipe required to process the input file in nanoseconds, with different bars indicating a different file. As described in section 6.2, the 10 input files contain from 1 million to 10 million records. All tests were repeated 10 times for each input file. Standard deviations are included in the graphs.

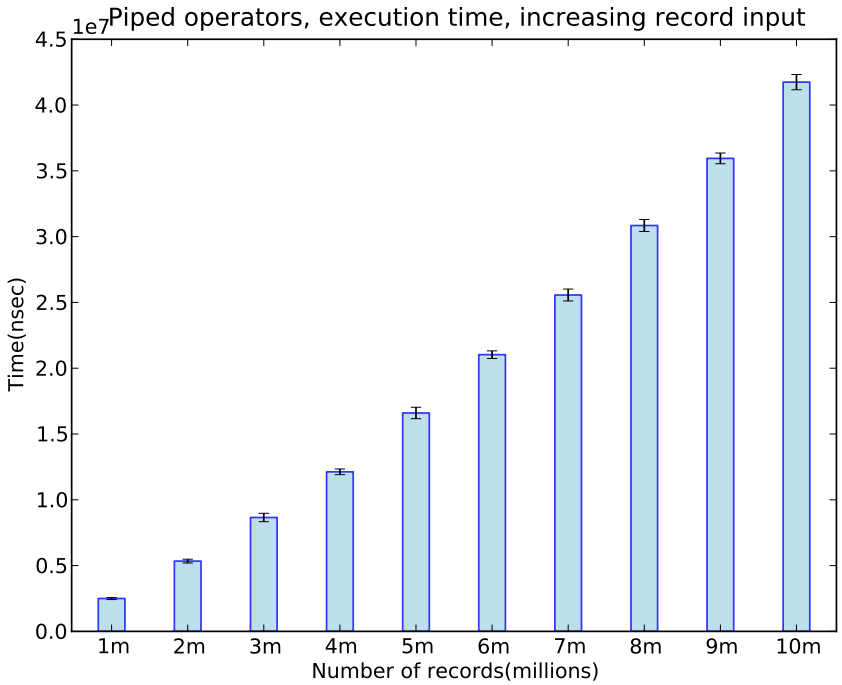
In the case of increasing thread count, the graphs depict the average time the operator pipe required to process the input file of 5 million records in nanoseconds, with different bars indicating different thread numbers. All tests were repeated 10 times for each separate thread number. Standard deviations are included in the graphs.

6.5.1 Evaluation

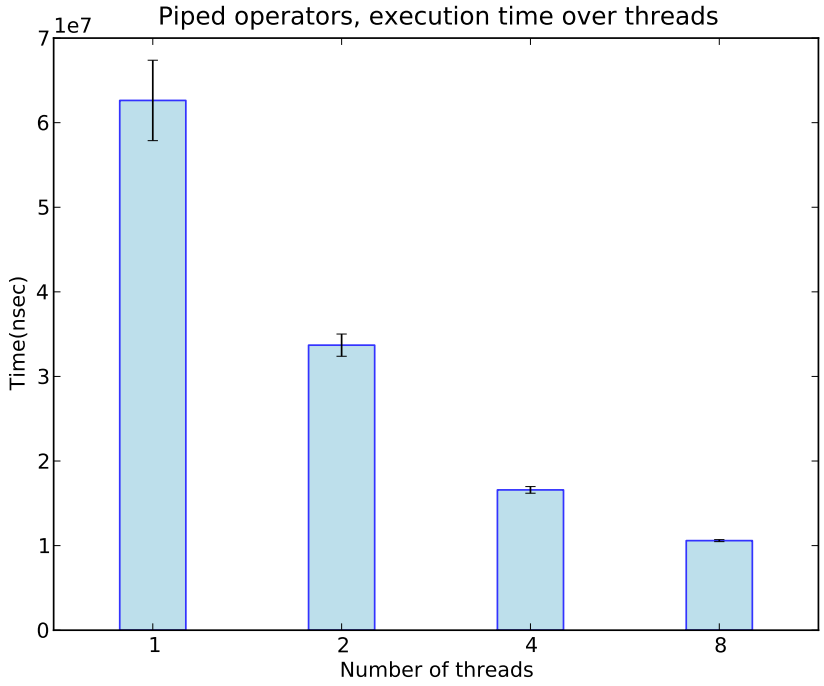
The piped operators behaved well in the conducted experiments and the results are presented in Figure 6.3. Once again results for 8 threads present the same issues as in benches 1 and 2.

6.6 Other experiments

For all three benchmarks the operators were also tested on the smaller data set on the personal computer for up to 4 threads and returned similar results. In addition, several other experiments were conducted, to evaluate different aspects of the ones posed on the three main benchmarks.



(a) cascading operators with increasing record input on 4 threads



(b) cascading operators on increasing number of threads. Record input fixed on 5 million records

Figure 6.3: Behaviour of pipes, six cascading operators on the large data set, with varying record input size and threads. 10 runs, with standard deviation, time in nanoseconds. Scalability is slightly limited in (b) for 8 cores and this is attributed to the nature of the hardware.

The feature of shuffled versus ordered output was tested with the results shown in Figures 6.5 and 6.6. It was observed that shuffled output demonstrates a performance gain. Disabling sorting, however, is not supported in Phoenix 2 and the library had to be overridden by a heuristic method. The feature can be used with Phoenix++, which supports this functionality. Maintaining input order for large numbers of records, requires sorting of equally many unique keys, thus explaining the great difference between the two types of execution.

6.6.1 Profiling operators

The operators were profiled internally, by measuring execution times for the three MapReduce phases. Figure 6.4 illustrates. User time was measured separately and in the Map phase, user time includes the execution time of the splitter. Combiner time is depicted separately. The three phases of Phoenix 2 are *Map*, *Reduce* and *Merge*.

For the `prepare_op`, it is evident that the splitting phase is very small. As in the case of selection and projection, it maintains the input order of records. This creates many individual keys, causing constant reallocations to the size of the key arrays maintained as individual key containers, for each cell of the Intermediate Buffer. This explains the time of the Map phase. The reducer is an identity reducer and in the merging phase final sorting of pairs by key takes place.

Both `select_op` and `project_op` are similar to the `prepare_op`. User time for the `select_op` is more than the `prepare_op` as it executes the user provided predicate for selection. The `project_op` user time is even larger, as it requires new record structures to be allocated. As the chosen design minimizes these allocations, the difference is small, compared to with the `select_op`.

In the cases of `sort_op` and `partition_op` the Reduce phase time of the system changes. This is expected, as the type o keys have changed. The `sort_op` sorts 10-byte long strings, that only have 1024 unique permutations. Therefore, the Reduce phase is burdened, having to iterate over many values of each key to produce the final pairs. For `partition_op`, hash and partition of records create a large overhead, as seen on the user map time for the operator. By creating only four distinct partitions, the Reduce phase is burdened, in a similar fashion to the `sort_op` case.

The performance of the `aggregate_op` is easy to explain. Initially, the Map phase is delayed to process all records, but the majority of the aggregations are executed by the combiners. After the mappers emit the partially aggregated values of merely

1024 distinct keys, the Reduce and Merge phases are next to trivial. Finally, the `select_raw_op` follows the execution pattern of `select_op`.

6.6.2 Library usability

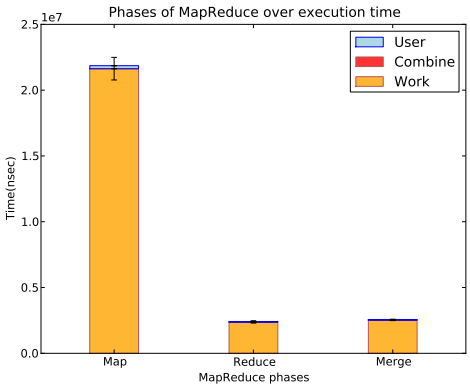
The usability of the library was empirically evaluated. Testing other settings of Phoenix 2 like *L1 cache* size,the number of *reduce tasks* and the use of *one queue* per task, helped calibrate the library. Finally, testing for up to 16 threads was conducted and is depicted in Figure 6.7.

An application written for Phoenix usually consists of several hundreds of lines of code and the difficulty factor multiplies along with devising the algorithms, manipulating the API, tuning the library and debugging the code. The seven example test applications written by the creators of Phoenix 2 for evaluation purposes, vary from 200 to 500 lines of code each, with an average of about 350 lines of `C` code.

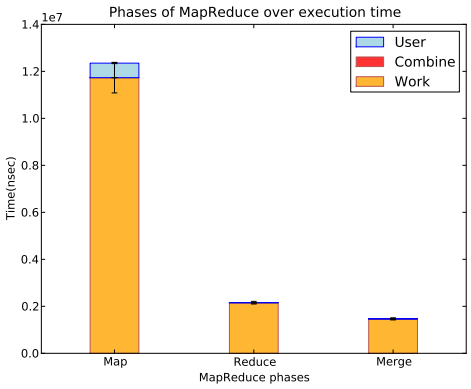
A program written with Relational Phoenix consisting of seven different operators in piped fashion and therefore using seven different operators written under Phoenix 2, is 200 lines long with a main function of 30-40 lines. An example written in pseudo-code with Relational Phoenix was depicted in Figure 5.4. The overall simplicity of a programm contributes to the usability of the library. Table 6.2 presents in lines of code, the difference between operators written with Phoenix 2 and operators using Relational Phoenix.

Table 6.2: Comparing operators between libraries in lines of code

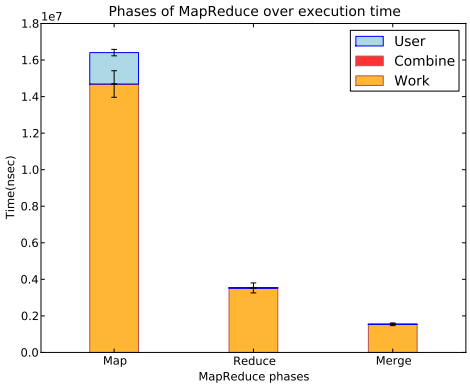
operator	***Phoenix 2***	***Relational Phoenix***
select	164	15
project	173	17
sort	169	26
partition	185	21
aggregate	198	34



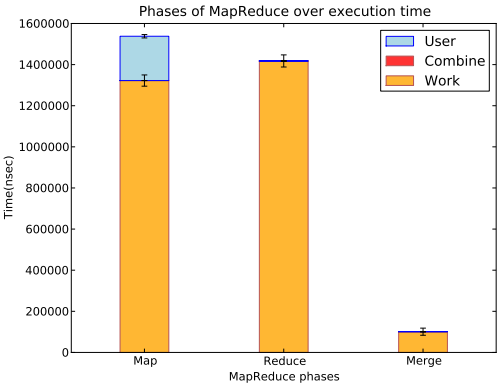
(a) prepare_op



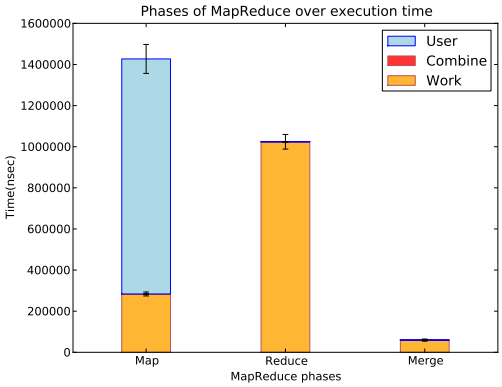
(b) select_op



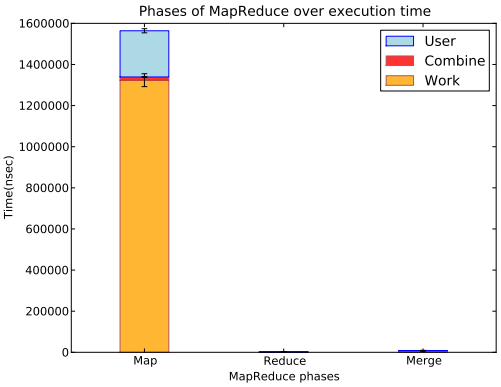
(c) project_op



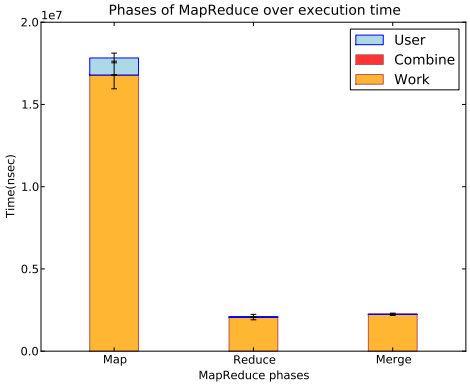
(d) sort_op



(e) partition_op



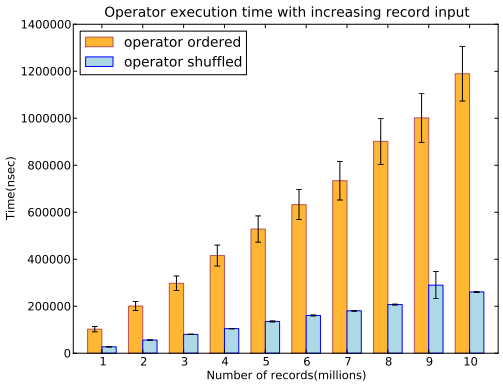
(f) aggregate_op



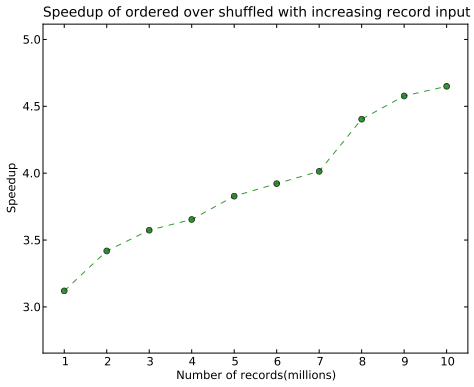
(g) select-raw_op

Figure 6.4: Profiling operators for execution times for different MapReduce phases.

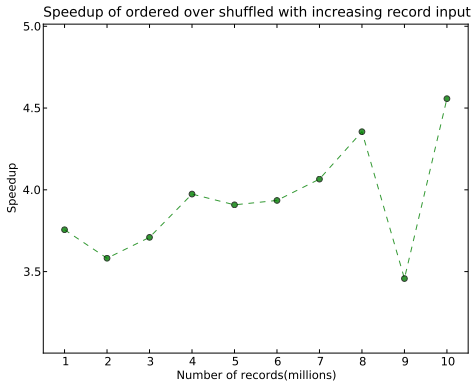
(a) `select_op` ordered vs shuffled



(b) `project_op` ordered vs shuffled

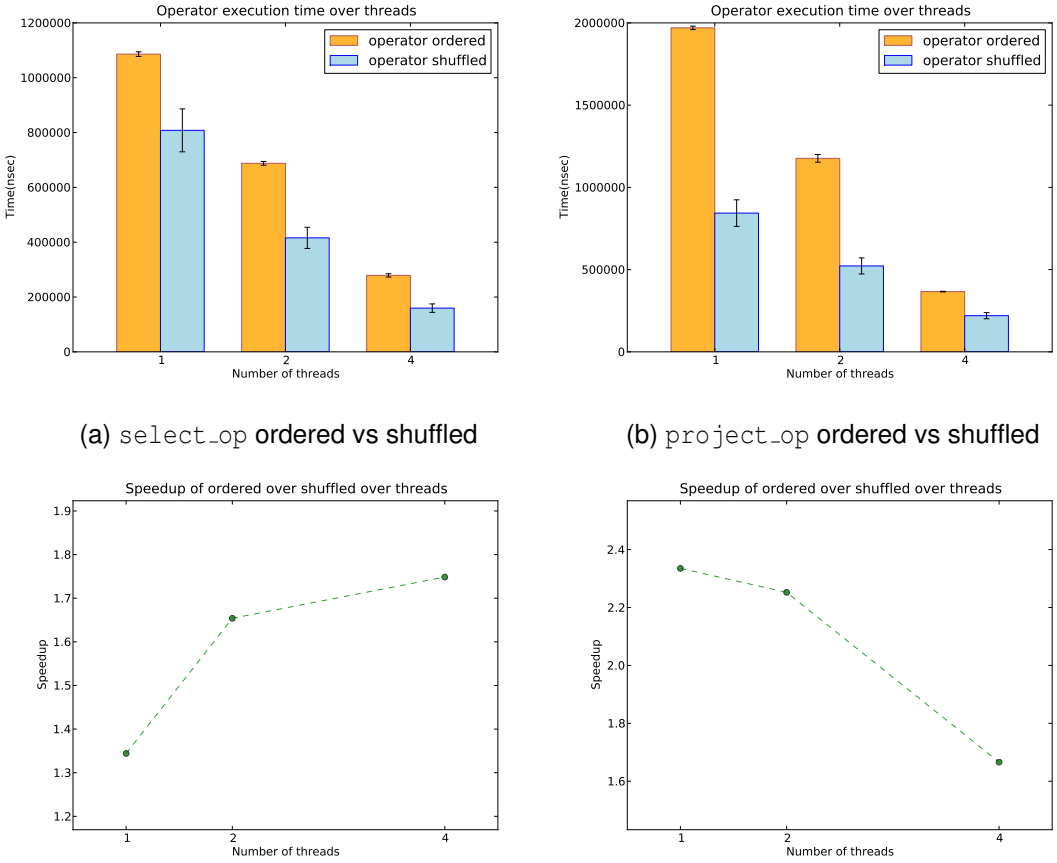


(c) `select_op` ordered vs shuffled speedup



(d) `project_op` ordered vs shuffled speedup

Figure 6.5: Impact on cardinality for individual operators on small data set for increasing record input on personal computer. Comparison between maintaining input order of records versus shuffled record output. On the top, shuffled output exhibits improved performance. On the bottom, speedup is presented. Default setting is to maintain input order, as Phoenix 2 does not support disabling of sorting. Ten files with 100K to 1 million records. Tests run 10 times with standard deviation and time in nanoseconds for 4 threads.



(a) `select_op` ordered vs shuffled

(b) `project_op` ordered vs shuffled

(c) `select_op` ordered vs shuffled speedup

(d) `project_op` ordered vs shuffled speedup

Figure 6.6: Impact of multithreading for individual operators on small data set for different thread number on personal computer. Comparison between maintaining input order of records versus shuffled record output. On the top, the left columns depict shuffled output and exhibits a performance gain. On the bottom, the speedup. Default setting is to maintain input order, as Phoenix 2 does not support disabling of sorting. Fixed input of 500K records. Tests run 10 times with standard deviation and time in nanoseconds.

Figure 6.7: Behaviour of pipes for cascading operator pipe consisting of all 6 operators for up to 16 threads on the `compute` server. It is apparent that after 8 threads scalability is limited. This is possibly a hardware limitation, as the machine provides two quad-cores with hyperthreading, limiting performance. Input is fixed on 5 million records, tests run 10 times with standard deviation and time in nanoseconds.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This work explored the use of the MapReduce model and programming paradigm on shared memory multicore architectures, in the application area of relational processing and query processing. The Phoenix 2 system was used to realize the MapReduce workflow on multicore CPUs and implement a library of composable unary relational operators. The aim was to enable parallelism, while creating a basis on top of which more complicated relational operations can be build.

The library was designed according to the architecture of a shared-memory multicore system, deviating from the distributed equivalents, in order to adapt to the requirements of the new hardware platform. The functionality of cascading operators in a piped fashion was implemented with the added ability of branching those pipes was allowed. An API was developed to allow clear and concise use of relational processing, while at the same time hiding completely the details of parallelism and the underlying MapReduce workflow.

The relational operator library was evaluated on two different CPU architectures, for different datasets and for record cardinalities varying over a large spectrum. The scalability of the library was also tested, by using different thread configurations. All the operators were tested in isolation as well as combined in pipe. Results shows that Relational Phoenix scales well, both when increasing record cardinality and when increasing the number of threads.

7.2 Future Work

Further work on developing the library can extend to several different directions, serving a variety of purposes. Other are easier to develop, while other look further into the future.

Porting Relational Phoenix to other systems

The Relational Phoenix library is orthogonal to most systems that implement the MapReduce workflow on multicore platforms, as described in the Related Work section. Therefore, it could be ported to any of them, utilizing MapReduce parallelism on very diverse CPU architectures, FPGAs, GPUs and heterogeneous hardware. Additionally, several steps can be taken to bring it closer to the form of a complete relational processing system, with declarative query features.

Implementing other operators

The secondary sort algorithm was designed in Chapter 4 and is a very useful operator to develop, especially because it prepares relational data for a map-side join. Another extension of the library, is to include binary operators like Union, Set or Difference. The case of Join operators presents particular interest, as it has been researched extensively in distributed MapReduce. Different algorithms of join have been developed like *Map-side* and *reduce-side* joins [42, 43], MapReduce model variants for Joins [44, 45] and other join types [46, 47, 48].

Further parallelism

As proposed in section 4.3.1, it is possible to run operators and even operator pipes to process the same data in parallel, by running Phoenix 2 on different threads. Taking care of the described potential pitfalls, it is a viable alternative, as an immutable input array does not require the use of locks for different Phoenix 2 instances to access it. The use of pipelining can also be explored, with cascading operators processing data before previous operators have finished execution.

A different solution to improve performance, is to change the synchronization condition, allowing the MapReduce runtime scheduler to be reused for a series of operators in a pipe, thus avoiding the re-instantiation overhead. Finally, the library can be

tested with different memory allocators, to overcome the bottleneck presented by the implementation of `malloc()` provided in the `C` standard library.

Towards more declarative languages

A first step to optimize the Relational Phoenix library is to allow multiple operators to be executed in a single MapReduce job. The correct placement of the operators in a workflow can be left for the user to organize, in a high-level fashion. Further work can follow the example of *Pig Latin* [38], breaking down a program to a query plan and than to consecutive MapReduce jobs. Another development can also be the creation of a language, adding a parser and extending the library to support a vocabulary, that describes a set of commonly used functions (i.e. AVG, MAX, SUM).

Bibliography

[1] Petros Lambropoulos. Algorithmic skeletons as a modern model of parallel programming, informatics research review, 2012.

[2] Petros Lambropoulos. Cloud on a chip, informatics research proposal, 2012.

[3] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[4] Krste Asanovic, John Wawrzynek, David Wessel, Katherine Yelick, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, and Koushik Sen. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56, October 2009.

[5] Vipin Kumar. *Introduction to Parallel Computing*. January 2002.

[6] M.D. McCool. Structured parallel programming with deterministic patterns. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, pages 5–5. USENIX Association, 2010.

[7] M. Aldinucci, M. Torquati, and M. Meneghin. Fastflow: Efficient parallel streaming applications on multi-core. *Arxiv preprint arXiv:0909.1187*, 2009.

[8] L. Yu, C. Moretti, A. Thrasher, S. Emrich, K. Judd, and D. Thain. Harnessing parallelism in multicore clusters with the all-pairs, wavefront, and makeflow abstractions. *Cluster Computing*, 13(3):243–256, 2010.

[9] Tudor-Ioan Salomie, Ionut Emanuel Subasu, Jana Giceva, and Gustavo Alonso. Database engines on multicores, why parallelize when you can distribute? *Proceedings of the sixth conference on Computer systems - EuroSys '11*, page 17, 2011.

[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[11] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. IEEE, 2007.

[12] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[13] R. Chen, H. Chen, and B. Zang. Tiled-mapreduce: optimizing resource usages of data-parallel applications on multicore with tiling. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 523–534. ACM, 2010.

[14] Yandong Mao and Robert Morris. Optimizing MapReduce for multicore architectures. *Computer Science and Artificial*, 2010.

[15] R.M. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 198–207. IEEE, 2009.

[16] Justin Talbot, Richard M Yoo, and Christos Kozyrakis. Phoenix++. In *Proceedings of the second international workshop on MapReduce and its applications - MapReduce '11*, page 9, New York, New York, USA, 2011. ACM Press.

[17] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[18] Horacio Gonz, Horacio González-Vélez, and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, November 2010.

[19] Murray I Cole. *Algorithmic Skeletons : Structured Management of Parallel Computation Table of Contents*. 1989.

[20] M Aldinucci, M Coppola, and M Danelutto. Rewriting skeleton programs: How to evaluate the data-parallel stream-parallel tradeoff. In *Proc. of International*

Workshop on Constructive Methods for Parallel Programming. Technical Report MIP-9805. University of Passau. Passau. Citeseer, 1998.

[21] Mario Leyton and J.M. Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 289–296. IEEE, February 2010.

[22] Berna L. Massingill, Timothy G. Mattson, Beverly A. Sanders, and Timothy G. Mattson Intel Corporation. Patterns for parallel application programs, 1999.

[23] Reinders J. OReilly Media Sebastopol. Intel threading building blocks: outfitting c++ for multi-core processor parallelism, 02007.

[24] Jrg Nolte, Mitsuhisa Sato, and Yutaka Ishikawa. Exploiting cluster networks for distributed object groups and collective operations. *Future Generation Computer Systems*, 18(4):461 – 476, 2002. ¡ce:title¿Best papers from Symp. on Cluster Computing and the Grid (CCGrid2001)¡/ce:title¿.

[25] Matthew H. Austern. *Generic programming and the STL: using and extending the C++ Standard Template Library*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[26] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. In *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*, pages 115 –124, 1999.

[27] W. Jiang, V.T. Ravi, and G. Agrawal. A map-reduce system with an alternate api for multi-core environments. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 84–93. IEEE Computer Society, 2010.

[28] Wei Jiang and Gagan Agrawal. Ex-MATE: Data Intensive Computing with Large Reduction Objects and Its Application to Graph Mining. *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 475–484, May 2011.

[29] Yi Wang and W Jiang. SciMATE: A Novel MapReduce-Like Framework for Multiple Scientific Data Formats. *cse.ohio-state.edu*.

[30] Bingsheng He, W Fang, Qiong Luo, and NK Govindaraju. Mars: a MapReduce framework on graphics processors. *Proceedings of the 17th*, pages 260–269, 2008.

[31] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. FPMR. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays - FPGA '10*, page 93, New York, New York, USA, 2010. ACM Press.

[32] M. de Kruijf and K. Sankaralingam. Mapreduce for the cell be architecture. 2007.

[33] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin. Mapcg: writing parallel program portable between cpu and gpu. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 217–226. ACM, 2010.

[34] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge: a programming model for heterogeneous multi-core systems. *SIGPLAN Not.*, 43(3):287–296, March 2008.

[35] R. Farivar, A. Verma, E.M. Chan, and R.H. Campbell. Mithra: Multiple data independent tasks on a heterogeneous resource architecture. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[36] Ronnie Chaiken, Bob Jenkins, Per-AAke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, August 2008.

[37] Alan F Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a High-Level Dataflow System on top of Map-Reduce : The Pig Experience. *Proceedings of the VLDB Endowment*, pages 1414–1425, 2009.

[38] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*, page 1099, New York, New York, USA, 2008. ACM Press.

[39] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using Hadoop. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 996–1005. IEEE, 2010.

[40] Ashish Thusoo, JS Sarma, Namit Jain, and Zheng Shao. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, pages 1626–1629, 2009.

[41] Rob Pike, Sean Dorward, and Robert Griesemer. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, pages 1–33, 2005.

[42] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in MaPreduce. *Proceedings of the 2010 international conference on Management of data - SIGMOD '10*, page 975, 2010.

[43] Jimmy Lin and Chris Dyer. Data-intensive text processing with mapreduce. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Tutorial Abstracts*, NAACL-Tutorials '09, pages 1–2, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

[44] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. Mapreduce-merge. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data - SIGMOD '07*, page 1029, New York, New York, USA, 2007. ACM Press.

[45] David Jiang, Anthony K. H. Tung, and Gang Chen. MAP-JOIN-REDUCE: Toward Scalable and Efficient Data Analysis on Large Clusters. *IEEE Transactions on Knowledge and Data Engineering*, 23(9):1299–1311, September 2011.

[46] Foto N. Afrati and Jeffrey D. Ullman. Optimizing joins in a map-reduce environment. *Proceedings of the 13th International Conference on Extending Database Technology - EDBT '10*, page 99, 2010.

[47] Alper Okcan and Mirek Riedewald. Processing theta-joins using mapreduce. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 949–960, New York, NY, USA, 2011. ACM.

[48] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using MapReduce. In *Proceedings of the 2010 international conference on Management of data - SIGMOD '10*, page 495, New York, New York, USA, 2010. ACM Press.