Programming Abstractions, Compilation, and Execution Techniques for Massively Parallel Data Analysis

vorgelegt von
Dipl.-Inf. Stephan Ewen
aus Mainz

der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr. Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Peter Pepper
Gutachter:	Prof. Dr. Volker Markl
	Prof. Dr. Odej Kao
	Prof. Mike Carey, Ph.D.

Tag der wissenschaftlichen Aussprache: 22. Oktober 2014

Berlin 2015
D 83

Acknowledgments

During the process of creating this thesis, I have been accompanied by a great many people over the time. Many of them offered valuable discussions and helped to shape the outcome of this thesis in one way or another.

First and foremost, I want to thank my adviser Volker Markl. He has not only provided the great work environment and the collaborations under which this thesis was created, but helped with countless comments and valuable insights. Throughout my entire academic life, Volker has mentored me actively and given me the opportunity to work on amazing projects.

I would like to thank my advisers Odej Kao and Mike Carey for their helpful comments and the discussions that we had at at various points, like project meetings, conferences, or academic visits.

I worked with many great people on a day to day basis, among who I want to give special thanks to Alexander Alexandrov, Fabian Hüske, Sebastian Schelter, Kostas Tzoumas, and Daniel Warneke. Thank you guys for the great work atmosphere, your ideas, your sense of humor, and for making the frequent long working hours not feel that bad. In addition, I had the pleasure of supervising several Master students who did amazing work in the context of our research project. At this point, I want to highlight the contributions made by Joe Harjung and Moritz Kaufmann. The collaboration with them yielded insights that evolved into aspects of this thesis. I also want to give thanks to the current generation of Master students, especially Ufuk Celebi, Aljoscha Krettek, and Robert Metzger. They are doing great work in both stabilizing the code base and infrastructure of the project, improving usability, and adding new ideas to the research system. With their help, it was made possible that many of the results of this thesis will live on under the umbrella of an open source project under teh Apache Software Foundation.

Last, but not least, I would like to thank my girlfriend Anja for having my back while I wrote this thesis. Thank you for your kindness and understanding during this time.

This thesis would not have been possible without the wondrous drink called Club Mate.

Abstract

We are witnessing an explosion in the amount of available data. Today, businesses and scientific institutions have the opportunity to analyze empirical data at unpreceded scale. For many companies, the analysis of their accumulated data is nowadays a key strategic aspect. Today's analysis programs consist not only of traditional relational-style queries, but they use increasingly more complex data mining and machine learning algorithms to discover hidden patterns or build predictive models. However, with the increasing data volume and increasingly complex questions that people aim to answer, there is a need for new systems that scale to the data size and to the complexity of the queries.

Relational Database Management Systems have been the work horses of large-scale data analytics for decades. Their key enabling feature was arguably the declarative query language that brought physical schema independence and automatic optimization of queries. However, their fixed data model and closed set of possible operations have rendered them unsuitable for many advanced analytical tasks. This observation made way for a new breed of systems with generic abstractions for data parallel programming, among which the arguably most famous one is *MapReduce*. While bringing large-scale analytics to new applications, these systems still lack the ability to express complex data mining and machine learning algorithms efficiently, or they specialize on very specific domains and give up applicability to a wide range of other problems. Compared to relational databases, MapReduce and the other parallel programming systems sacrifice the declarative query abstraction and require programmers to implement low-level imperative programs and to manually optimize them.

This thesis discusses techniques that realize several of the key aspects enabling the success of relational databases in the new context of data-parallel programming systems. The techniques are instrumental in building a system for generic and expressive, yet concise, fluent, and declarative analytical programs. Specifically, we present three new methods: First, we provide a programming model that is generic and can deal with complex data models, but retains many declarative aspects of the relational algebra. Programs written against this abstraction can be automatically optimized with similar techniques as relational queries. Second, we present an abstraction for iterative data-parallel algorithms. It supports incremental (delta-based) computations and transparently handles state. We give techniques to make the optimizer iteration-aware and deal with aspects such as loop invariant data. The optimizer can produce execution plans that correspond to well-known hand-optimized versions of such programs. That way, the abstraction subsumes dedicated systems (such as Pregel) and offers competitive performance. Third, we present and discuss techniques to embed the programming abstraction into a functional language. The integration allows for the concise definition of programs and supports the creation of reusable components for libraries or domain-specific languages. We describe how to

integrate the compilation and optimization of the data-parallel programs into a functional language compiler to maximize optimization potential.

Zusammenfassung (German Abstract)

Aufgrund fallender Preise zur Speicherung von Daten kann man derzeit eine explosionsartige Zunahme in der Menge der verfgbaren Daten beobachten. Diese Entwicklung gibt Unternehmen und wissenschaftliche Institutionen die Mglichkeit empirische Daten in ungekannter Grenordnung zu analysieren. Fr viele Firmen ist die Analyse der gesammelten Daten aus ihrem operationalen Geschft lngst zu einem zentralen strategischen Aspekt geworden. Im Gegensatz zu der seit lngerem schon betriebenen *Business Intelligence*, bestehen diese Analysen nicht mehr nur aus traditionellen relationalen Anfragen. In zunehmendem Anteil kommen komplexe Algorithmen aus den Bereichen *Data Mining* und *Maschinelles Lernen* hinzu, um versteckte Muster in den Daten zu erkennen, oder Vorhersagemodelle zu trainieren. Mit zunehmender Datenmenge und Komplexitt der Analysen wird jedoch eine neue Generation von Systemen bentigt, die diese Kombination aus Anfragekomplexitt und Datenvolumen gewachsen sind.

Relationale Datenbanken waren lange Zeit das Zugpferd der Datenanalyse im groen Stil. Grund dafr war zum groen Teil ihre deklarativen Anfragesprache, welche es ermglichte die logischen und physischen Aspekte der Datenspeicherung und Verarbeitung zu trennen, und Anfragen automatisch zu optimieren. Das starres Datenmodell und ihre beschrnkte Menge von mglichen Operationen schrnken jedoch die Anwendbarkeit von relationalen Datenbanken fr viele der neueren analytischen Probleme stark ein. Diese Erkenntnis hat die Entwicklung einer neuen Generation von Systemen und Architekturen eingelutet, die sich durch sehr generische Abstraktionen fr parallelisierbare analytische Programme auszeichnen; *MapReduce* kann hier beispielhaft genannt werden, als der zweifelsohne prominenteste Vertreter dieser Systeme. Zwar vereinfachte und erschloss diese neue Generation von Systemen die Datenanalyse in diversen neuen Anwendungsfeldern, sie ist jedoch nicht in der Lage komplexe Anwendungen aus den Bereichen *Data Mining* und *Maschinelles Lernen* effizient abzubilden, ohne sich dabei extrem auf spezifische Anwendungen zu spezialisieren. Verglichen mit den relationalen Datenbanken haben MapReduce und vergleichbare Systeme auerdem die deklarative Abstraktion aufgegeben und zwingen den Anwender dazu systemnahe Programme zu schreiben und diese manuell zu optimieren.

In dieser Dissertation werden verschiedene Techniken vorgestellt, die es ermglichen etliche der zentralen Eigenschaften von relationalen Datenbanken im Kontext dieser neuen Generation von daten-parallelen Analysesystemen zu realisieren. Mithilfe dieser Techniken ist es mglich ein Analysesystem zu beschreiben, dessen Programme gleichzeitig sowohl generische und ausdrucksstark, als auch prgnant und deklarativ sind. Im einzelnen stellen wir folgende Techniken vor: Erstens, eine Programmierabstraktion die generisch ist und mit komplexen Datenmodellen umgehen kann, aber gleichzeitig viele der deklarativen Eigenschaften der relationalen Algebra erhlt. Programme, die gegen dies Abstraktion

entwickelt werden knnen hnlich optimiert werden wie relationale Anfragen. Zweitens stellen wir eine Abstraktion fr iterative daten-parallele Algorithmen vor. Die Abstraktion untersttzt inkrementelle (delta-basierte) Berechnungen und geht mit zustandsbehafteteten Berechnungen transparent um. Wir beschreiben wie man einen relationalen Anfrageoptimierer erweitern kann so dass dieser iterative Anfragen effektiv optimiert. Wir zeigen dabei dass der Optimierer dadurch in die Lage versetzt wird automatisch Ausfhrungsplne zu erzeugen, die wohlbekannten, manuell erstellten Programmen entsprechen. Die Abstraktion subsumiert dadurch spezialisierte Systeme (wie Pregel) und bietet vergleichbare Performanz. Drittens stellen wir Methoden vor, um die Programmierabstraktion in eine funktionale Sprachen einzubetten. Diese Integration ermgliche es prgnante Programme zu schreiben und einfach wiederzuverwendenden Komponenten und Bibliotheken, sowie *Domnenspezifische Sprachen*, zu erstellen. Wir legen dar wie man die bersetzung und Optimierung des daten-parallelen Programms mit dem Sprachbersetzer der funktionalen Sprache so integriert, dass maximales Optimierungspotenzial besteht.

Contents

1. Introduction

The last decade has seen an explosion in the amount of available data. With ever falling costs for persistent storage, increasing importance of Internet-based services, and technology trends like smartphones and pervasive sensors, many companies and scientific institutions are sitting on huge piles of raw data. For example, the Large Hadron Collider at CERN produces roughly 15 petabytes per year [38]. Petabyte volumes are also expected to be created by next-generation DNA sequencing technologies [19]. The infrastructure behind Microsoft's search web-search engine and other web-based services (SCOPE) stores and processes many petabytes and is being enhanced for data in exabyte order [114]. It is a wide-held believe that a number of future scientific breakthroughs will be the result of an efficient and effective analysis of huge amounts of data. In astronomy, the LSST telescope is being set up to repeatedly photograph the entire available sky. The tracking and classification of objects as well as the search for extraordinary occurrences will happen through data analytics [23]. The potential of large scale analytics becomes even bigger when applied across domain boundaries, such as combining scientific data with census-, mobile-, or social-media data. The latter has, for example, been successfully demonstrated in case of detecting unknown pharmaceutical side effects with the help of web data [105].

But also aside from large scientific institutions and the "big players" in industry, data analytics plays and exceedingly important role for many enterprises as a means to optimize logistics, improve the quality of their services, or to better understand their target market [27]. While certain sciences have employed specialized systems for very specific analytical tasks on very specific data (e. g., nucleotide alignments [11] or particle physics [20, 38]), most applications are spread over a wide and heterogeneous spectrum. The analytical queries are very diverse, are posed in an ad-hoc fashion, or evolve quickly over time. Business intelligence/decision support [82], fraud detection [28], or behavior modeling and recommendations [72] are prominent examples of applications for which it is crucial to be able to quickly adapt queries and models to changing settings.

For the longest time, this type of flexible analytics has been carried out by relational database management systems like Oracle [43], IBM DB2 [29], or Teradata. The key to the success of these systems was the algebraic data model and query language the are at the heart of their design. The relational model and algebra [42] create an abstraction between the logical query and all physical execution aspects. Queries are defined in a concise

and declarative fashion, while a compiler and an optimizer derive an execution plan that describes an efficient sequence of physical steps to compute the query result [2, 53]. The compilation takes into account the specific system capabilities (available data structures and algorithms), the system parameters (such as network-/disk-/CPU speed ratios), physical storage structures, or characteristics of the data's statistical distributions.

Despite these facts, the recent trend of large data sets and complex analysis tasks, has pushed the architecture of the relational database systems, which essentially goes back to the architecture of the System R prototype [18], to its limits. The types of challenges imposed on the systems are often classified through the *"Three Vs"* [27]:

1. Volume: The size of the data is too large to apply the conventional techniques. Traditional setups scale only to one or few machines, due to a tight coupling between the machines. They aggressively exploit the the available resources, but implement no elasticity as required for very large setups. The small setups in turn offer not enough storage capacity, bandwidth, memory, or CPU power to complete the analysis tasks within a reasonable time. The large volume requires new distributed architectures that combine resources of many machines to execute analytics in a massively parallel fashion, or novel algorithms that allow to answer the analytical questions with fewer passes over the data, or by looking only at specific subsamples.

2. Variety: The majority of the data to be analyzed came traditionally from transactional database systems and, as such, was structured and comparatively homogeneous. Recently, many new data sources provide input to the analytical processes with data in various formats like relational/tabular, hierarchical, text, graph-structured, and others. These diverse types of data make it obvious that the relational model by itself is too limited to be able to represent the data structures and useful operations on top of them. Closely related to "Variety" is also the trend to apply increasingly complex analytical tasks for classification or predictive modeling with the help of machine learning methods. Such applications include often graph- and linear algebra operations and naturally require a larger variety of data models, such as matrices or graphs. These tasks, frequently referred to as "Deep Analytics" or "Big Analytics"[1] [95] need support for much more complex operations, including iterative/recursive computations.

3. Velocity: High production rates of data and requirements for near-time feedback require to break with the traditional paradigm of analytical applications, where data changes slowly and queries are manifold and may change quickly. Many applications like monitoring and intrusion detection have long running queries that

[1] As opposed to "shallow" or "small" analytics which are based on the classical operations such as filtering, joining, and aggregating.

need to examine data elements as they are produced. While research and industry came up with a series of systems dedicated to *data stream analysis* [1, 17], many research questions remain, including how to best design the interaction between "batch" and "stream" analytics.

Several companies extended this set of challenges with additional aspects that they deem critical. The most prominent ones are *Veracity* and *Visualization*. The aspect of *Veracity* emphasizes the fact that a large portion of the data comes no longer from trusted and accurate data sources (like for examples centralized transactional database systems), but rather from diverse and noisy sources. Examples include web data (access logs, or social media data, possibly extracted using algorithms with limited accuracy) and data gathered from sensors and sensor networks, where the results typically vary within a certain measurement error margin. The challenge is to correctly handle a combination of accurate and noisy data. The aspect of *Visualization* stresses the point that any findings from the analytics are only valuable if there are means to illustrate them and use them to adjust processes or implement new processes. Such illustration typically requires appropriate means of visualization, for example for high dimensional distributions or clustering.

This observation has led to plethora of new developments, trying to address one or more of the problems. While all challenges are under active research, many people consider "Volume" mostly solved for example by the new generation of distributed relational database management systems, either focusing on transactions [67] or analytics [96]. The other challenges are much less precisely defined and further away from solution [95].

One of the arguably most influential architectures addressing simultaneously the "Volume" and "Variety" challenge was *MapReduce* [44], as published by Google in 2004. MapReduce builds on runtime building blocks known from parallel databases [45, 52], but defines a generic parallel programming abstraction based on the second-order functions *Map* and *Reduce*. The result is that any program that decomposes into a pair or a *record-at-a-time* Map function[2] and a *group-at-a-time* Reduce function can be executed by the MapReduce framework in a highly parallel fashion. MapReduce builds on a distributed file system for storage and a key/value data model where the value types can be arbitrary data structures and are only interpreted by Map and Reduce functions. That way, the framework can work with almost arbitrary data models and can express many operations beyond what relational databases can express. In its underpinnings, the MapReduce framework implements a scalable parallel sort, where the Map function preprocesses data before the sort and the Reduce function processes the grouped data after the sort. The open source MapReduce framework Hadoop [106] has risen to become a very prominent

[2] In correct functional programming terms one would labeled it a *flatMap* function.

technology in the field of data analysis and has been the target of countless scientific papers.

The MapReduce abstraction has proven useful for a variety of applications, from search index building and log file analysis [44] to clustering and fuzzy grouping [71,102]. Through exploiting the flexibility of a file-based key/value data model and the ability to execute quasi arbitrary program code inside the Map and Reduce functions, the applicability of the MapReduce framework can be pushed far beyond what fits the functional model of the map and reduce functions. Hadoop has, for example, served as a basis for SQL- and XQuery-like languages where queries are translated into a sequence of MapReduce programs [26, 85, 99], or as a target runtime for a machine learning library [14]. The ability to act as a runtime for relational queries, for queries over hierarchical data, or algorithms over matrices illustrates that the MapReduce approach does in fact solve some of the problems subsumed under the aspect of data "Variety".

A series of other systems have been inspired by MapReduce and aim at mitigating its shortcomings, while retaining or even broadening the wide applicability. *MapReduceMerge* [110] extended the MapReduce pipeline with a configurable binary operator. *Cascading* [37] offers additional second-order functions and supports connecting them to directed acyclic graphs (DAGs). *Dryad* [64], *Nephele* [103], and *Storm* [79] offer an even more generic stream programming model. *PACT* [9, 22] processes DAGs of operators with more semantics than MapReduce and Cascading, while *Hyracks* [31] supports a complex assembly of programs from operator primitives such as sorting, hashing, or data exchange.

All of these systems have in common that they adopt some form of the principle of *data-parallel programming*. Analysis programs consist largely or purely of user-defined functions (UDFs), which are applied in parallel on partitioned data. The specific sequence of computation is represented by chaining UDFs together, usually as directed acyclic graphs (DAGs), with MapReduce being a very simple special case of a DAG. In the scope of this thesis, we refer to this type of abstraction as *"generic data-parallel programming"*, to stress that it acts as a flexible general-purpose abstraction, rather than one tailored towards a special class of problems, algorithms, or data structures.

1.1. Problem Definition

The wide range of applications on top of these generic data-parallel programming models and -systems is a testament to the fact that this abstraction is well suited to solve many of the problems associated with data- and operator "Variety". However, the flexibility comes at a rather high price. Compared to relational databases, these data parallel programming systems offer reduced performance, more verbose and complicated

programs, as well as harder maintenance and evolution of data formats and queries [76,88]. For complex multi-pass algorithms, for example from the areas of machine learning and graph analysis, these systems exhibit up to an order of magnitude lower performance than specialized systems [77, 78]. As a result, general-purpose, relational, and several other specialized systems are commonly used together in today's analytical pipelines, causing overly complicated implementations, more complex infrastructure maintenance, additional data movement, and resource management deficiencies.

In the scope of this thesis, we will discuss two big aspects which we believe to be the among the most important shortcomings of the current breed of generic data-parallel programming systems. Solving those shortcomings enables systems that offer a query abstraction that is similar in conciseness to that of relational databases and offer performance that is competitive with specialized code and dedicated systems even for many complex graph- and machine learning algorithms. At the same time, the systems retain the original flexibility and wide applicability. The two shortcomings of the state-of-the-art can be summarized as follows:

1. The current systems lack a means to express iterative or recursive computation, including situations where the solutions are incrementally computed. The parallel programs are typically used to express the step function of an iteration (the "contents" of the loop), while the loop itself is constructed around the programs in the form of drivers (external or with tool support by the framework). This solution prohibits the system from performing any form of optimization (such as efficient handling of loop-invariant computations) and causes for many algorithms unnecessary re-computation or data movement.

2. There is no separation between the model against which a queries is written and the physical execution of the queries. In the context of the UDF-centric data-parallel programming systems, one can observe that programmers sacrifice either performance, when they program against higher abstractions and libraries, or they sacrifice generality, when optimizing the data types and operators for very specific contexts and data sets. This follows the general principle in software engineering and programming languages that generic high-level abstractions are typically paid for by reduced runtime performance, because the underlying implementation cannot exploit the various specialized solutions for the different individual cases. In contrast to that, relational databases managed to offer an *"abstraction without regret"* solution. Their means to achieve this is the strict separation between an algebraically specified declarative query and a runtime execution that is customized for each query and data set by means of an optimizing query compiler.

Current solutions to these problems fall short on those crucial requirements[3]. To support iterative algorithms, the state-of-the-art systems either specialize on very specific domains and loose their generality, or they only provide tooling that more easily support the external loops around the program. The first approach covers only a fragment of the application spectrum, the later approaches do not achieve competitive performance. The recognition of the "abstraction vs. performance" problem let to the creation of a series of higher-level languages on top of the Hadoop MapReduce system. *Hive* [99], *Pig* [85], and *JAQL* [26] implement languages that are similar to SQL or XQuery. While some of them support more complex data models than relations (such as nested objects, or *list* and *map* data types), they loose the flexibility and generality of MapReduce and other generic data-parallel programming systems. As soon as an operation does not fit the set of provided operators, the programmer needs to resort back to writing UDFs against the MapReduce abstraction. There is a severe impedance mismatch between the data model from the higher-level languages (*relational* or NF^2) and that of the UDFs (*object oriented* or *functional*), requiring intransparent, costly, and error-prone type mapping. In addition to introducing error sources and fragmenting the program, the state-of-the-art way of embedding UDFs breaks the language optimization and impedes runtime efficiency. We also argue that a principled approach must use information from the algebraic part of the query to contextualize the UDF execution, for example through customized data types or pruning of unnecessary fields.

There is an inherent conflict between the goal to make an abstraction declarative or algebraic, and the goal to make it flexible and open to complex operations. However, a higher expressiveness comes not necessarily with a proportional loss of declarativity. An example for this is arguably the MapReduce abstraction, which demonstrated that a highly expressive programming model can support automatic parallelization of the programs. In Figure 1.1, we list more examples of such programming abstractions. The figure shows a qualitative sketch that classifies the abstractions with respect to how declarative and how expressive they are. We label one abstraction as more declarative than another, if programs can express more operations through explicit constructs or primitives offered by the abstraction. Furthermore, the constructs must be transparent to the system and leave the exact runtime realization of their operations to be decided by the system. As an example, an abstraction based on list comprehensions [49] is closely as expressive as MapReduce with arbitrary UDFs (without external drivers and side effects[4]). At the same time, a list comprehension abstraction is much more transparent to

[3] Upon closer examination, these two stated points are related: The first one is in many ways a prerequisite to the second one. A declarative or algebraic abstraction requires that the program describes the operation directly, or the "intent" of an operator, rather than hard-code its exact realization such as in the form of side effects as or an external driver program. An expressive set of primitives is necessary for a program to explicitly state the intent of its operations, at least to the extent as it is relevant to the interplay of operations and thus to the executing system.

[4] With side effects we refer here to the access of state outside the Map or Reduce function.

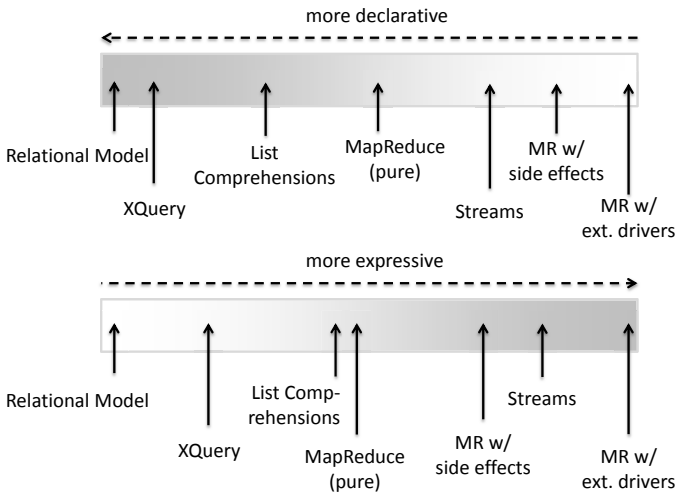

Figure 1.1.: A sample spectrum of programming models with respect to algebraic properties and expressiveness.

the executing framework. Similarly, the streams abstraction [64, 103] is more transparent to a system than MapReduce with side effects, but in many ways even more flexible.

We argue that a right combination of techniques supports the creation of an abstraction that is extremely flexible and at the same time highly declarative, leading to very efficient query execution. We summarize the research problems addressed in this thesis under the following question:

> *"How can we create a programming abstraction that has MapReduce-style UDFs as first-class citizens, operates on quasi arbitrary data models, and supports complex iterative/recursive algorithms, but retains at the same time many of the declarative properties of the relational model such that concise query languages and relational-style query optimization are possible?"*

Note: In the course of this thesis, we frequently build the chain of arguments and ideas such that we start from a MapReduce-like abstraction level and progress towards something rich and algebraic, like the relational model, while trying to keep the flexibility of MapReduce. We could also have taken the opposite approach and started with the relational model, and relax the operators while trying to retain as many of the algebraic properties as possible. Both choices would arrive at a similar point in the spectrum between the relational model and MapReduce-style abstractions.

1.2. Contributions of this Thesis

The contributions of this thesis follow the outlined question and problems.

In the first part of this thesis, we discuss the an *"abstraction for data parallel programs"* that is flexible and retains a comparatively high level of declarativity, leaving choices about execution strategies open to the system. A central aspect is to show how to *"integrate iterative computations with data-parallel programming"*. We deal with the questions of creating a generic and expressive, yet efficient, abstraction for iterative algorithms. Among these algorithms, we also consider algorithms that exploit incremental and dynamic computation, which is an inherently stateful computation and thus a challenge for parallel systems. The abstraction we present can capture the semantics to support automatic parallelization. We elaborate on methods and techniques to optimize iterative programs, such as detecting loop-invariant computation and the creation of execution strategies that maximize these loop-invariant parts. An implementation into the Stratosphere system [97] shows that our approach can compete with the performance of specialized systems, while not restricting the applicability of the system.

In the second part of this thesis, we discuss the *"design and the compile chain of a data-parallel analysis language"*. We review the differences in the handling of queries between the UDF-centric data-parallel programming systems and relational databases. Based on these findings, we propose the design of a data-parallel language for analytics that we embed it into the popular Scala language. Our language shares many properties with the relational algebra, supports UDFs as first class citizens and blurs the distinction between referencing a core language operator or specifying a concise UDF. It features a highly flexible functional data model that is algebraic, but allows to represent virtually arbitrary data structures. We describe techniques for a compile chain that analyzes the data model and relevant parts of UDF code to create the input to a relational-style query optimizer and to generate efficient runtime data structures specialized to the specific query. As a result, the language supports very concise analysis programs that transparently mix and match operators with user-defined functions and shows little runtime overhead when compared to very specialized and manually optimized programs.

Parts of this thesis have been published in the following papers and articles:

Conference- and Journal Papers

1. Dominic Battré, **Stephan Ewen**, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke
Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing
In: Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC 2010), pp. 119-130, 2010.

2. Alexander Alexandrov, Dominic Battré, **Stephan Ewen**, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, Daniel Warneke
Massively Parallel Data Analysis with PACTs on Nephele
In: Proceedings of the VLDB Endowment, 3(1-2), pp. 1625-1628, 2010.

3. Alexander Alexandrov, **Stephan Ewen**, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, Daniel Warneke
MapReduce and PACT - Comparing Data Parallel Programming Models
In: Proceedings of the 14th Conference on Database Systems for Business, Technology, and Web (BTW 2011), pp. 25-44, 2011.

4. **Stephan Ewen**, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl
Spinning Fast Iterative Data Flows
In: Proceedings of the VLDB Endowment, 5(11), pp. 1268-1279, 2012.

5. **Stephan Ewen**, Sebastian Schelter, Kostas Tzoumas, Daniel Warneke, Volker Markl
Iterative Parallel Data Processing with Stratosphere: An Inside Look
In: Proceedings of the ACM SIGMOD International Conference on Management of Data,
pp. 1053-1056, 2013

6. Alexander Alexandrov, Rico Bergmann, **Stephan Ewen**, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, Daniel Warneke
The Stratosphere Platform for Big Data Analytics
In: VLDB Journal (to appear)

Books

1. Kai-Uwe Sattler, Wolfgang Lehner (Eds.)
   Web-Scale Data Management for the Cloud
   Stringer, 2013, ISBN 978-1-4614-6855-4
   ⇒ Chapter 4 has been mainly authored by this thesis' principal author.

1.3. Outline of the Thesis

The remainder of this thesis is structured as follows:

Chapter 2 - Background: Data-Parallel Programming

In Chapter 2, we review Data-Parallel Programming - a concept that is used by all prominent programming abstraction and systems for large-scale analytical programs. We discuss the MapReduce system both from a programming abstraction and execution model view and contrast it with the stream programming model.

Chapter 3 - The Parallelization Contract Model

In Chapter 3, we review the concept of Parallelization Contracts (PACTs), a generalization of the MapReduce abstraction. We describe PACTs as an abstraction between semantically poor MapReduce-style UDF programming and semantically rich relational queries. PACTs are amendable to many optimizations known from relational databases without giving up the generality of MapReduce and thus form the basis for a seamless combination of UDFs with relational-style operations.

Chapter 4 - An Abstraction for Iterative Algorithms

Chapter 4 introduces concepts and abstractions to specify iterative data flow programs in a transparent way. The abstraction enables programs to express iterative algorithms on the same abstraction as regular relational- or MapReduce style queries, obviating the need to create specialized systems for those tasks. A special focus is put on algorithms that *incrementally* compute their results and consequently require the maintenance of a state across iterations. Furthermore, we discuss extensions to relational-style optimizers to handle these iterative programs efficiently, by as caching loop-invariant data and pushing computation out of the loop.

Chapter 5 - A Functional Language Frontend

In Chapter 5, we describe the design of query language embedded in a functional programming language. We discuss how the combination of an algebraic data model and relatively simple code analysis techniques allows our system to compile these queries much like a database system compiles relational queries. At the same time, our query language

retains the expressiveness of MapReduce and seamlessly mixes built-in operations and UDFs without impedance mismatch.

Chapter 6 - Related Systems

While Chapters 3, 4, and 5 discuss related work specific to their individual contributions, Chapter 6 briefly reviews systems that are related in their overall goals and architecture. We contrast these systems with the *Stratosphere* system, where we implemented the techniques presented in this thesis.

Chapter 7 - Conclusions and Outlook

Chapter 7 concludes the thesis with a summary of our contributions. We finally give an outlook into interesting open questions and follow-up research.

2. Background: Data-Parallel Programming

In the introduction, we motivated data-parallel programming as an abstraction to implement parallelizable programs. We have briefly hinted at the strengths of the concept and shortcomings of current systems. In this chapter, we will review the details of data-parallel programming models. We revisit the basic concepts behind data-parallelism and review the programming abstraction and execution model of two well known systems for parallel data analysis.

2.1. Basics of Data-Parallel Programming

Parallel processing is a requirement for the analysis of large data sets. For a few years now, the speed of individual CPU cores has not increased significantly - instead, the number of cores per CPU, the number of hardware thread contexts, and the number of CPUs per machine have been increasing fast. Furthermore, the vast amount of data produced in many scenarios can only by analyzed using the aggregate compute power, memory, and I/O bandwidth of many machines. In order to scale an analysis program to many machines and processor cores, the program must be specified in a way that is inherently parallelizable.

The principle idea behind data parallelism is to parallelize by data elements[1], i. e., to process different data elements (or groups of elements) independently on different machines, or by different CPU cores. To apply an operation to the data set in a data-parallel fashion, multiple instances of the operator execute concurrently and process different sets of data elements. For efficient parallelization across multiple machines, it is important that the different operator instances do not modify any shared data structures. It is consequently important to form the sets of input elements for each operator instance such that they can work independently of each other. The composition of the independent sets is specific to each operation.

[1] In the course of this thesis, we will use the terms *element* (of a data set) and *record* (as a physical structure that holds an element) synonymously.

Example Consider two operations from different data analysis tasks (Figure 2.1). The first operation takes a collection of web pages and extracts links from these web pages, for example as a preparation to build a graph representation of the web pages. The second operation takes a collection of instantiations from different random variables $(A, B, C, \dots)$ and computes for each random variable the mean and the variance. The first operation can work independently on each data element (here: web page). While processing a single page, the operation needs no information about any other page. The result of the operation is simply the union of the links created for each individual web page. Such operations are often called "embarrassingly parallel". In contrast, the second operation cannot work independently on each element (here: instantiation of a random variable). To compute the mean and variance of each random variable, all instantiations of that variable must be processed together. The operation can be executed on a data-parallel fashion under the constraint that instantiations of the same variable are processed by the same instance of the operator.

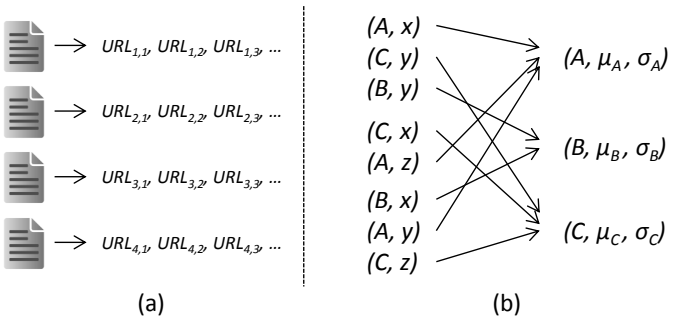

Figure 2.1.: Examples of Data-Parallelism: (a) Extracting links from individual web pages, (b) Computing descriptors for the distribution of different random variables.

Data parallelism stands in contrast to other models of parallelism that are often used in database systems, such as pipeline parallelism. In the case of pipeline parallelism, multiple operations on the same data elements happen in parallel. Typical implementations of that concept use separate operators for different steps of the processing. While one operator is working on data items, it pipes its results directly to the succeeding operator, who starts processing the data elements immediately. Pipeline parallelism is limited by the number of independent operations that can be performed on a data element. because this number is typically rather small, it can not form the basis of massively parallel data processing.

2.2. MapReduce: A Simple Data-Parallel Programming Model

One of the simplest and best-known representatives of data-parallel programming models is *MapReduce* [44]. It has been suggested by Dean et al. in 2004 as a framework for distributed data processing programs that run on very large clusters of machines. It is designed for programs that process very large amounts of data, such as a web crawl of the entire Internet, or a history of all user interactions with the search engine over years. The paper explains how Google uses MapReduce for various tasks on different types of data, such as the analysis of click log files (with filters, transformations, and aggregations), inverting web graphs (from pages with outgoing links to pages with incoming links), or the computation of word/term histograms to identify topics and term relevance for web sites.

MapReduce implements a simple form of the data parallelism concept outlined in the previous section. It uses a simple key/value data model: All elements processed by MapReduce are represented as a pair of values, where one is designated the *key* and is used to define relationships between elements. The other element is the *value*, which can be thought of as payload data that is not interpreted by the MapReduce framework, but only by the specific user-defined functions of an analysis program. MapReduce programs implement two functions, *Map* and *Reduce*, consuming and producing key/value pairs as described by the following signatures[2]:

$$Map : (K, V) \rightarrow (K', V')^*$$

$$Reduce : (K', \{V'\}) \rightarrow (\tilde{K}, \tilde{V})^*$$

The Map function takes one key/value pair (K, V) at a time and transforms it, producing zero or more key/value pairs (K', V') with possibly different key and value data types. Between the Map and the Reduce function, the framework groups together all (K', V') pairs that have the same key. The Reduce function is then applied to each such group with key K' and the set of all values V' that appeared with that key. The Reduce function in turn produces zero, one, or more key/value pairs $(\tilde{K}, \tilde{V})$, again with potentially different data types.

Example Let us illustrate the MapReduce abstraction by the example of a simple program that computes an occurrence histogram of words. Input to the program is a sequence of sentences in the form of key/value pairs where the key is irrelevant and the value is the sentence. The output is a sequence of pairs where the key is the word and the

[2] We follow the common notation to use the Kleene Star (*) do denote sequences with of zero or more elements, and the plus (+) to denote sequences of one or more elements.

value the number of times the word has occurred. Pseudo code for the Map and Reduce functions is given in Listing 2.1. The Map function takes each sentence and tokenizes it into words. It returns each of these words as a pair $(word, 1)$. Between the application of the Map and Reduce functions, the MapReduce framework groups all pairs with the same word together. The reduce function then sums up the counts (1) for each word to produce the final count.

```
1  function map(key: Int, value: String) : Sequence[(String, Int)]
2  {
3    words: Array[String] = value.splitAtNonWordCharacters();  // sentence to words
4
5    for (word : String in words) {
6      return (word, 1);                  // return each word with its initial count
7    }
8  }
9
10 function reduce(key: String, values: Sequence[Int]) : (String, Int)
11 {
12   count: Int = 0;
13   for (num: Int in values) {         // take all values and sum up
14     count += cum;                    // the occurence count
15   }
16   return (key, count);
17 }
```

Listing 2.1: Sample Map and Reduce functions for the Word Histogram example.

To define a complete MapReduce analysis program, the programmer must specify two additional functions. The first one (the *reader*) produces the initial input data as sequence of (K, V) pairs, while the second one (the *writer*) consumes the final pairs $(\tilde{K}, \tilde{V})$. In most cases, the reader simply reads and parses files into key/value pairs and the writer key/value pairs to files. These reader and writer functions are often omitted from program specifications because they are highly reusable and typically created as part of a library once per data format. A common example is a reader that produces as key/value pairs the sentences from text files, setting the key to the offset of the sentence in the file, and the sentence itself as the value.

Figure 2.2 illustrates the processing of a MapReduce program on a logical level. The input is partitioned into several chunks - in the example here one chunk per sentence. Each chuck is independently supplied to an instance of the Map function. The result of the mappers is grouped by the key (the word), before the Reduce function computes the sum of the individual counts. This example illustrates nicely the core idea behind data-parallel programming: Different parts of the data are processed independently by different instances of the program code. The two instances of the Map function can easily process the input sentences in parallel (on different computing cores of a machine or on different machines), because they have no interdependency. After the grouping, the various instances of the Reduce functions can work independently on different sets of groups. The Map and Reduce steps are connected by the intermediate grouping step that is handled by the MapReduce framework. This grouping step reorganizes the data

across the individual sets produced by the different instances of the Map function. As a consequence, the task of grouping the data needs to address the challenges of parallel systems, while the implementation of the Map and Reduce functions are shielded from that complexity.

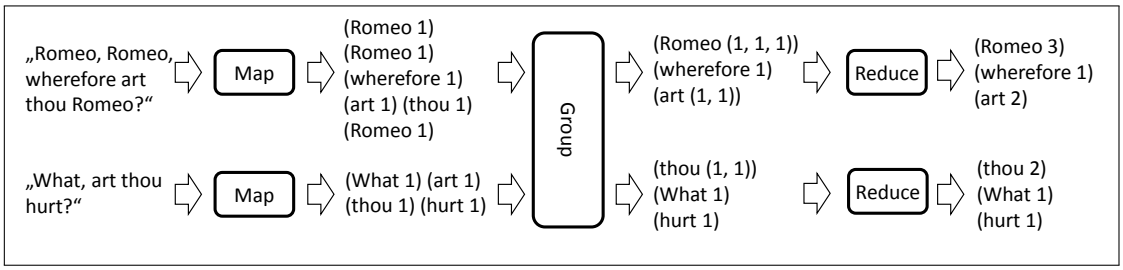

Figure 2.2.: The logical execution of the sample MapReduce program.

Reduce functions are often used to aggregate data elements. Many of the commonly used aggregation functions (e. g., *sum*, *minimum*, *maximum, average*) are *distributive* or *algebraic* [59]. Such functions can compute the aggregate value by computing *partial aggregates* over parts of the group's elements, and incrementally combine these partial aggregates until the final aggregate value is computed, effectively forming an *aggregation tree*. The Reduce function in the above pseudo code fulfills that condition because computing the sum of all single occurrences is an associative operation. In the context of MapReduce, such functions are often called "*combinable reduce functions*".

In the above example, the Reduce function performs a simple aggregation, consuming multiple key/value pairs and producing a single pair. However, Reduce functions are not restricted to aggregation functions. The signature of the Reduce function allows it to produce multiple key/value pairs, making it a generic *group-at-a-time* operation that consumes a group of elements and produces a group of elements (aggregation functions are a special case of such a group-at-a-time operation, where the returned group is of size one). By allowing groups of larger size, Reduce functions can also represent operations that are beyond the expressiveness of aggregation functions, such as filters based on characteristics of the entire group, rather then individual elements.

2.2.1. Parallel Execution of MapReduce Programs

MapReduce is a simple example of a data-parallel system and implements representative architecture followed by many other systems. In this section, we briefly discuss the execution of MapReduce programs as described by Dean et al. in the original paper [44] and implemented by the Hadoop system [106]. The execution of the MapReduce programs is sketched in Figure 2.3. At the beginning of each execution, the input is divided into

several splits. These splits are each processed by an individual instance of the Map function. The reader function turns the data into a sequence of key/value pairs and the system evaluates the Map function over each pair. The resulting key/value pairs are partitioned on the key (typically using a hash function) into r partitions, where r is the number of parallel Reduce instances that will be executed. The partitioning has the effect that all values for one specific key are in the same partition, and through sorting the pairs on the key, all elements with the same key are grouped together. The system performs these operations independently for each instance of the m Map functions, producing in total $m * r$ partitions.

The r reducers each process one of the r partitions. For each of them, the system fetches the m parts that together constitute the respective partition, transferring them across machine boundaries over the network. Since each part is already sorted after the key, the system can create a fully sorted partition by merging the parts the same way as in a multi-way merge join. Having the key/value pairs grouped, the Reduce function is applied once per consecutive sequence of key/value pairs with the same key.

The storage of the input data is transparent to the MapReduce execution system. However, the by far most common case is storing the input as files in a distributed file system. In such a setup, files are chunked into blocks of a certain size (typically 100s of megabytes) that are spread across the machines of a cluster. Each block acts as an input split for a parallel instance of the Map function. By spreading the blocks across different machines, the aggregate I/O bandwidth of all involved machines is available for data reading. Furthermore, the blocks of each file are typically replicated to multiple machines, such that no data is lost when a machine fails.

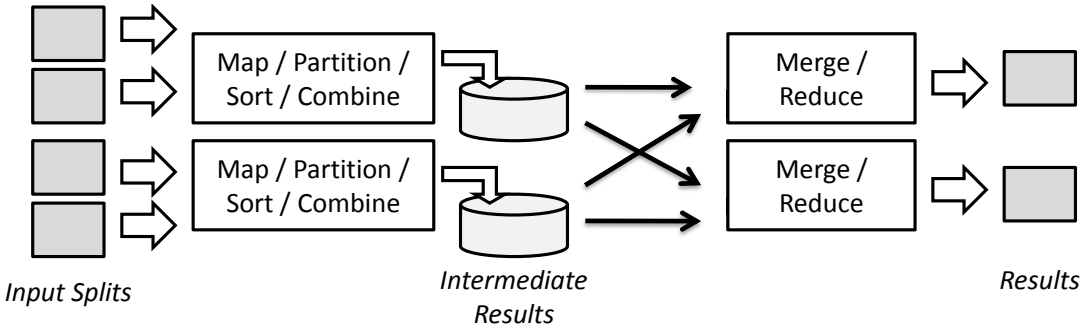

Figure 2.3.: The execution of a MapReduce program.

It is in principle arbitrary on which machines the parallel instances of the Map and Reduce functions execute. From a performance perspective, it is beneficial to execute the Map functions at the same locations where the input data is stored. If a Map function runs on the same machine as the input split that it processes, it can access that input

split locally and does not need to fetch its data over the network. Because data is moved across the network anyways for the Reduce functions, their placement has rather little restrictions.

2.2.2. A Functional Programming View on MapReduce

The MapReduce paradigm is heavily inspired by functional programming. *Map* and *Reduce* are *second-order functions* that apply a first-order function to a sequence of elements. Map applies the first-order function to each element individually and Reduce collapses a number of elements to a single value by always combining two values with the first-order function until all values are combined to one element. In functional programming all functions are side effect free, i. e., they do not depend on state other than their parameters and do not modify any state. The result of a function is solely represented by its return value.

The MapReduce programming model builds on top of that idea. The Map and Reduce functions implemented as part of the program can be thought of as the first-order functions which are applied to the data. The MapReduce framework assumes that the parallel instances of the Map and Reduce functions do not influence each other through side effects. The functions consequently define a scope of data elements that a function may see at a time, and MapReduce uses this limited scope to isolate parallel instances against each other. By choosing between Map or Reduce as the function type, the programmer declares the scope that is required for the particular function is either *group-at-a-time* or *element-at-a-time*. Figure 2.4 shows how Map defines every key/value to be independent from all other key/value pairs, and Reduce to define all pairs with the same key to depend on each other.

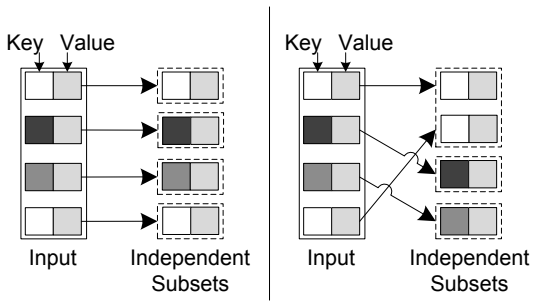

Figure 2.4.: Illustration of partitioning records into independent sets in Map (left) and Reduce (right). Figure is taken from our previous publication [9].

2.2.3. Limits of MapReduce

MapReduce is a straightforward abstraction for simple use cases. However, its highly restrictive programming model forces programmers to use many workarounds for any but the simplest programs. MapReduce offers only two primitives Map and Reduce, which are both unary functions. Any operation that works on multiple inputs must be rewritten as one or more unary operations, which is often very hard and frequently inefficient. MapReduce programs must always consist of a Map function, followed by a Reduce function. Many analytical tasks do not map naturally to an element-at-a-time operation (Map), followed by a group-at-a-time operation (Reduce). Such tasks must be rewritten to become one or more MapReduce programs.

The problem that many analytical programs need to be written as a sequence of MapReduce jobs has been somewhat mitigated by programming- and workflow frameworks on top of MapReduce. Examples for such systems are Cascading [37] or Apache Oozie [15]. They offer an abstraction where more than two functions may be composed into a program. These programs are then broken down into a sequence of multiple MapReduce jobs. After each job, data is written to distributed files, which form the input to the next job. Not only is the repeated result writing a costly operation, it also voids any potential of cross-operator optimization. Consider for example the case of a program that contains two Reduce functions $R_1 : (A, (B, C)) \rightarrow ((A, B), D)$ and $R_2 : ((A, B), D) \rightarrow ((A, B), E)$. We denote the key by the first element of the data type (A respectively (A, B)) and the value by the second. Because the first Reduce function operates on data partitioned by A, the second Reduce function could re-use that partitioning, as any partitioning on A is also a valid partitioning on (A, B). The second Reduce function would only have to group the key/value pairs differently inside the partitions, which is a much cheaper operation than repartitioning the data. However, partitioning cannot be preserved between MapReduce jobs, because these two jobs are treated independently by the MapReduce framework and exchange data only through the distributed filesystem.

The second problem, the limited set of primitives, becomes obvious when considering typical operations on two inputs, for example something like a relational join. We illustrate the typical MapReduce implementation of a join between two large data sets (T and R) in Figure 2.5. In MapReduce, the only way to combine two large data sets is through by defining the input to be the union of their files. The Map function reads splits from both inputs and assigns to each element from the input a lineage tag that marks from which of the two logical inputs the element came. The result of the mapper is a *tagged union* of elements, which are partitioned on the key. Each partition contains for each key the elements from both inputs. The Reduce function takes the set of key/value pairs for each key, separates the pairs by the lineage tag into two sets corresponding to the two inputs. The join result is formed by building the Cartesian product of between

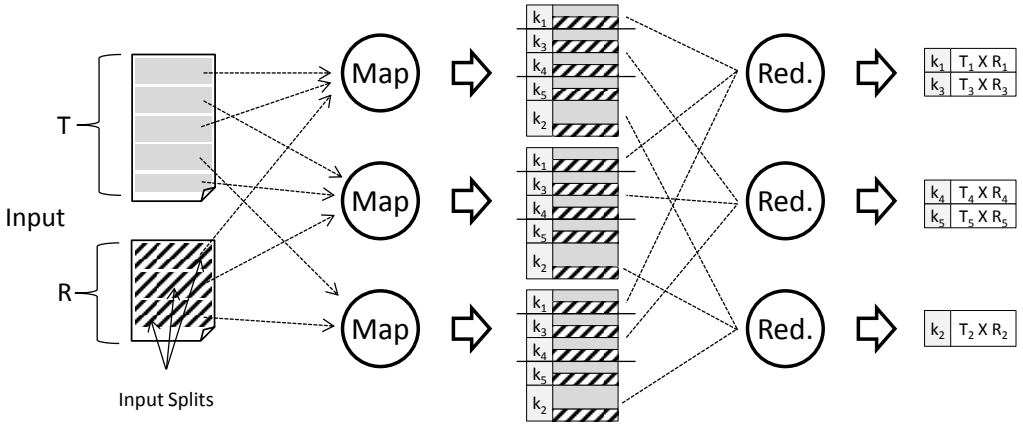

Figure 2.5.: Realizing a partitioned join in MapReduce. Mappers consume a union of the inputs and add lineage information to each key/value pair. The partitioned result holds per key values from both inputs. The Reducer separates those based on the lineage tag and forms the cross-product of the two sets of values per key.

these two sets.

As seen above, the restriction to unary functions requires clumsy workarounds for certain operations. In addition, the MapReduce framework has no notion that the executed operation works on two separate data sets. MapReduce can consequently not treat the input data sets differently, thereby loosing efficiency. A very common situation in the query plans of relational database management systems is that one input is already partitioned on the join key, and the other one needs partitioning. This type of join, often called *directed join*, leaves the already partitioned data at its locations and partitions the second set towards the already partitioned data. It is naturally a lot more efficient than the repartitioning of both inputs. Furthermore, not treating the inputs differently prevents the usage of many of the algorithms that adjust the partitioning to improve performance in the presence of data skew. A well known representative of such an algorithms is PRPD (Partial Redistribution & Partial Duplication) [109], as used for example in the Teradata Distributed Relational Database Management System.

2.3. The Stream Programming Model

A well-known and powerful alternative to the MapReduce is the *Stream* model, implemented in some form or another by the systems *Dryad* [64, 111], *Nephele* [103],

Hyracks [31], and *Storm* [79]. While all of these systems have different capabilities, they share the stream model as a programming abstraction. Similar to MapReduce, the stream model composes programs from sequential code blocks (henceforth called tasks), each of which is executed in multiple parallel instances that work independently of each other. While in MapReduce these tasks are a Map or a Reduce function, the stream model is even more generic: Tasks consume and produce streams of data elements. Other than that, the computation inside the tasks is not restricted by any constraints. The produced streams of one task are connected with the consumed streams of different tasks forming a *directed acyclic data flow*, or *DAG data flow*. A task that produces data elements can choose to which streams (and thus to which consumers) the elements are sent; this flexibility supports the creation of complex data routing logic. In all of the above mentioned systems, the programmer specifies a compact version of the parallel data flow. Program parameters include how many parallel instances to create of each task, the stream connections between the tasks, and how the data produced by a tasks is distributed among the outgoing streams.

Example Consider an analysis program that takes image files of text articles and produces a text document for each image (say a PDF file) and simultaneously an inverted index to search within these files. The example has first been presented in the context of the Nephele system [103], but works in a similar way for the other systems mentioned in this section as well. Figure 2.6 shows the unparallelized data flow. The program starts with a data source, reading images and forwarding them to a text recognition task. That task in turn converts each image into formatted text. The converted texts are then sent to two different tasks: The first task converts the formatted texts to PDF documents and passes these documents to a writer that stores the PDF documents as files. The second task tokenizes each text into the list of words. The document names and word lists are sent to the successor task that creates an inverted index from these words (occurrence lists).

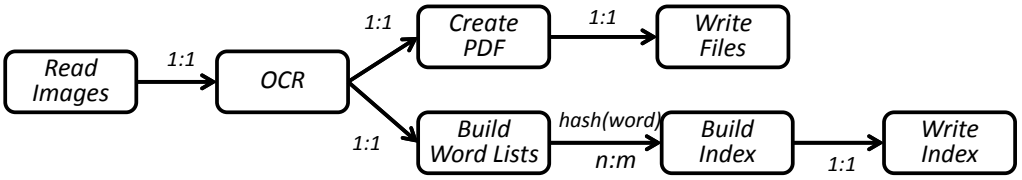

Figure 2.6.: The example program represented as a data flow of tasks. The annotations to the edges $(x : y)$ describe the behavior of the channels when their source and target tasks are parallelized.

The connection between most tasks have a $1 : 1$ pattern, meaning that in the parallelized version, the n-th parallel instance of a task streams data only to the n-th parallel instance

of the respective successor task. The connection between the "Build Word List" and the "Build Index" task have a $n : m$ pattern, meaning that each of the parallel "Build Word List" task instances streams data to all of the parallel "Build Index" task instances. That way, the "Build Word List" task can choose which task to send a word to and realize the partitioning (all equal words are sent to the same task) that is necessary for the parallel index building.

The parallelized version of the example data flow is shown in Figure 2.7. Here, each task is executed in two parallel instances. As outlined in the previous paragraph, the figure shows how most streams connect corresponding instances of source and target task, while the "Build Word List" and the "Build Index" tasks connect their parallel instances pairwise.

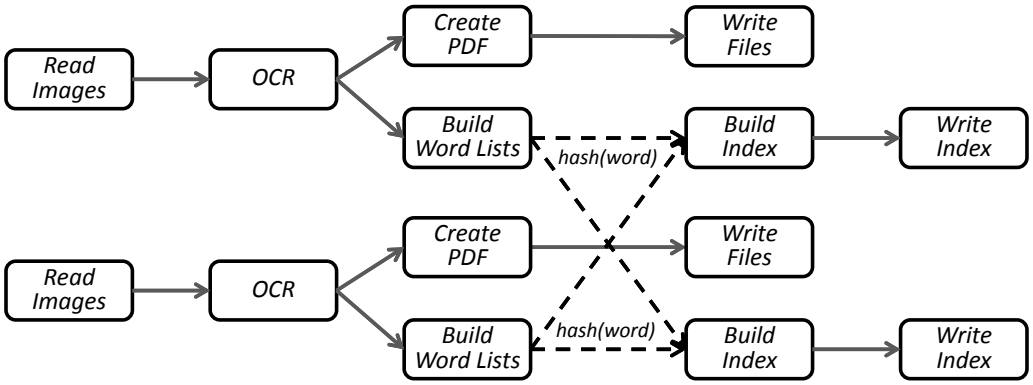

Figure 2.7.: The example program as a parallelized data flow of tasks. Solid edges forward data elements in a piped fashion, dashed edges denote network data exchange.

2.3.1. Parallel Execution and Relationship to MapReduce

The stream programming model explicitly describes the parallelization of its programs and defines the patterns with which the parallel instances of the tasks exchange data with one another. Given that these programs are restricted to DAGs, there is always a partial order which describes what tasks have to be started before what other tasks, thereby giving explicit scheduling information. In most cases one can co-locate the execution of parallel tasks connected via $1 : 1$ patterns, such that their data exchange does not cross machine boundaries. The $n : m$ data exchanges require generally that data is shipped across machine boundaries.

The stream data flow model is much more powerful than MapReduce. It is possible to view MapReduce as a special case of a DAG data flow, as illustrated in Figure 2.8.

The stream programming model addresses several of the limitations in the MapReduce paradigm: One can explicitly model tasks operating on multiple inputs, compose programs that consist of many tasks, and realize operations inside the tasks that are not easily representable as a Map or Reduce function. At the same time, there are also major drawbacks compared to MapReduce: An abstraction as generic as the stream abstraction requires programmers to implement a significant amount of functionality to achieve even simple tasks. For example, MapReduce offers the frequently needed parallel grouping in the form of the Reducer out of the box. The same functionality in the streams abstraction entails explicitly specifying the data redistribution (partitioning) and the local grouping (for example by sorting and splitting the sorted stream into groups). In addition, the generality of the stream abstraction makes it very hard for the system to reason about the semantics of the individual tasks. In contrast, the MapReduce framework knows that the Map function is a pipelined record-at-a-time operation[3], i. e., resulting key/value pairs are produced before all input pairs are consumed. Similarly, the Reduce function is known to be a blocking group-at-a-time operation. Such knowledge about pipelining and blocking operator behavior is essential for efficient execution and necessary to avoid stream deadlocks (which may occur in non-tree DAGs under various combinations of blocking and pipelining tasks). If the system lacks such knowledge, it cannot automatically detect and resolve such situations. It hence becomes the burden of the programmer to reason about the behavior of his distributed program, to detect and resolve such situations.

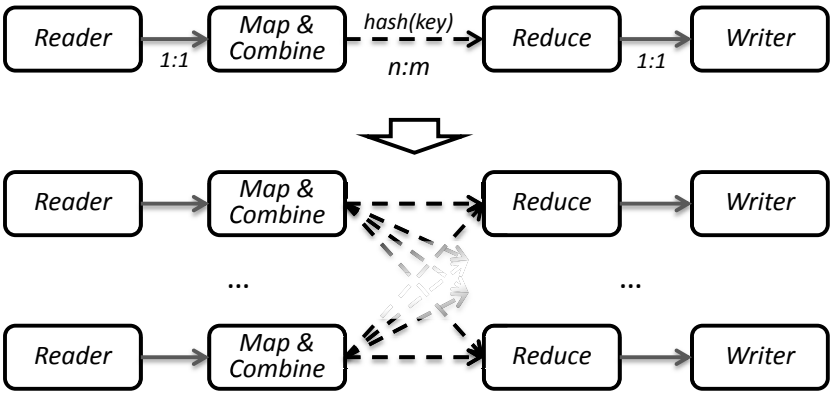

Figure 2.8.: The MapReduce pipeline as a special case of a DAG data flow. Unparallelized stream program (top) and parallelized version (bottom).

[3]Popular MapReduce implementations like Hadoop [106] allow programmers to break out of the record-at-a-time mode and access many records together. This increases expressiveness while further reducing the ability for the system to reason about the function behavior. We refer to the original MapReduce programming abstraction here.

2.3.2. Limits of the Stream Model

As discussed in the previous section, the Stream Model is very expressive and powerful, but comes at a price of comparatively high program development effort (which can be somewhat mitigated by libraries). In addition, the individual tasks appear even more like black boxes than MapReduce functions do. These problems are however not relevant if the programs executed by a stream data flow system are created from a more semantically rich abstraction: The stream abstraction acts here only as a lower level interface against which higher-level abstractions create their execution plans. In that case, the semantically richer abstraction can reason about the behavior of the tasks and set up the data flow program accordingly. Examples for this approach include compiling SCOPE [39] to Dryad [64] flows, PACT [22] to Nephele [103] schedules, and compiling AQL [25] to Hyracks programs [31].

The stream abstraction reaches the limits of its expressibility when used to express programs that have complex control flow constraints across tasks. This is especially true for iterative- or recursive algorithms. If the latter are to be expressed as a single data flow program, this program naturally needs to feed back intermediate results from later tasks to earlier tasks, thus forming a cycle in the data flow. Arbitrary cyclic data flows break the clear predecessor/successor relationship between tasks; this relationship is however essential for scheduling. In addition, many iterative algorithms need to make the control flow decisions coordinated across all parallel task instances, which goes beyond the expressiveness of the stream abstraction. We will describe in Chapter 4 how to model generic iterative algorithms and how to extend a stream-based data flow engine for their execution.

3. The Parallelization Contract Model

The previous chapter explained data-parallel programming as a generic mechanism to implement parallelizable programs at the example of two well known abstractions: The *MapReduce* model (Section 2.2) and the *Stream* model (Section 2.3). We discussed the principles behind the abstractions, as well as their limitations with respect to expressing advanced algorithms and with respect to separation of programming model and execution, which prevents MapReduce programs to be automatically optimized.

In the following, we describe the *Parallelization Contract (PACT)* programming model that we, together with our co-authors, conceived and published in [7, 9, 22]. PACT is a powerful and expressive programming abstraction that lends itself to optimizable data flow specification. The PACT programming model has a similar level of abstraction as MapReduce and can be thought of as a generalization thereof, because MapReduce programs are specific cases of PACT programs. In comparison to MapReduce, PACT offers additional *second-order functions*, as well as lightweight methods to let the system know about the behavior of the functions, which greatly increases the optimization potential.

In addition to what we published about PACT in previous papers [7,9,22], we discuss how it resembles a small orthogonal set of efficient data-parallel primitives and corresponds to the set operations of the extended relational algebra without record manipulations. One aspect in this chapter is the question what declarative specification means in the context of data-parallel UDFs, and what implications different data models have on the power of the declarative abstraction. The chapter concludes with a discussion of the limits of expressibility and optimizability of programs written against the PACT abstraction.

3.1. Problem Statement

As discussed in the previous chapter, both MapReduce (Section 2.2) and the Stream model (Section 2.3) have significant drawbacks when it comes to expressing analysis programs for more complex problems. MapReduce's was originally designed for tasks like web-index building, inverting the direction of directed web graphs, or building word histograms to model the relevance of words in documents [44]. Implementing an operation as common as a join in MapReduce requires the programmer to bend the programming

model by creating a tagged union of the inputs to realize the join in the *reduce* function (cf. Section 2.2.3). Not only is this a sign that the programming model is somehow unsuitable for the operation, but it also hides from the system the fact that there are two distinct inputs and prevents the system from applying different operations to the two inputs. Apart from requiring awkward programming, this shortcoming is often a cause of low performance [88].

The stream model is more generic, but at the same time complex to program, because the framework itself only routes data elements between different parallel instances of code blocks (tasks). Systems based on the stream model abstraction (e. g., *Dryad* or *Nephele*) typically exhibiting better performance for analysis tasks than MapReduce [64, 103], but are also more complex to program. Both MapReduce and the stream programming model have in common that they often require the programmer to make decisions that are dependent on dynamic parameters, like the degree of parallelism, or the sizes of input data sets, or the distribution of certain values in the input data. As a result, the programs are tailored towards a specific scenario and need to be manually adjusted if the parameters change. In contrast, the field of relational databases has long recognized that the automatic optimization of relational analysis programs (stated as SQL queries) is crucial. Different execution plans for the same query can outperform each other by orders of magnitude in different situations [58, 94].

In the following, we list what we believe are the major shortcomings of MapReduce and the stream model with respect to ease of programming and potential for optimization. The list defines a series of desiderata for a programming model that is easy to program and lends itself to optimization.

- In MapReduce, the primitives *Map* and *Reduce* alone are too constrained to express many data processing tasks both naturally and efficiently. The single primitive of the stream abstraction (the sequential code block that consumes and produces streams) is too generic for easy program composition and exposes no characteristics of the executed code. A suitable abstraction must expose similar characteristics of the tasks as the functional view on MapReduce (for example pipelined record-at-a-time versus group-at-a-time operations), while allowing flexible program composition as in the stream model.
- Both programming models tie their programs to fixed execution strategies. They imperatively specify exactly how to execute the program by hard-coding steps or explicitly describing how data is routed between parallel instances. Optimizable programs need a declarative specification of *"what"* a specific operation should do with its data, not *"how"* exactly to execute this operation [94]. The later allows the system to pick an efficient implementation from a predefined set of execution strategies. In order to achieve that, the system considers characteristics of the

input data (e. g. size, distributions), the operation itself, and the interaction with previous and succeeding operators. The later part is crucial for the re-usability of functions in different scenarios.

- Both abstractions make no assumptions about the behavior of the functions that are executed inside the tasks. In MapReduce, the type of the function (Map or Reduce) may give a certain indication about characteristics of the functions, which are not available in the stream model. The dependencies between the data produced by an operation with respect to the consumed data is crucial to exploit optimization potential.

- The key/value data model of MapReduce, and the arbitrary data model of the stream abstraction complicate the writing of programs and prevent reasoning about relationships between fields or attributes across different functions or tasks. A data model with more fields being transparently visible to the system (rather than only a single key fields in the Key/value data model) increases the optimization potential.

3.2. The PACT Programming Model

The *"Parallelization Contract"* programming model (PACT) [7, 22] can be thought of as an extension, or a generalization, of MapReduce. Its core is the idea to retain the level of abstraction of MapReduce, but to separate the specification of the program's logical operations from the details of the execution. The PACT model offers a larger set of primitives (in the form of further second-order functions) and high flexibility in composing them into programs. At the same time, it adopts the functional interpretation of MapReduce: Programs consist of blocks of code (user-defined functions) that process independent partitions of the data in parallel. Each user-defined function is used together with a specific *second-order function* (such as Map or Reduce), which specifies rules how to build the independent partitions. In the context of PACT, these rules are called *contracts*, hence the name *Parallelization Contract* model. We commonly use the term *function* to refer to a user-defined function with its according second-order function. We call functions with a *map* second-order function "Map function", and so on, because the type of second-order function is the most important property of a function[1].

[1] The terminology in this chapter is slightly different from the terminology in the original paper that introduced PACT [22]. The changes reflect the feedback we got with respect to making the concepts easier to understand. The most notable difference between the terminologies is that in this thesis, we use the term *PACT* only as the name of the overall programming model. We use the terms *user-defined function* (UDF) to describe sequential units of user-defined code, and the term *second-order function* for the rules (contracts) to build the independent partitions of the data and to apply the functions on the data.

The functions are assembled into programs by connecting them to a DAG of functions. PACT uses a compiler and optimizer to take the specified program and devise an efficient execution plan that builds correct independent partitions of each function. The principle follows the idea of database query compilers that translate SQL queries to query execution plans. Introducing a compiler and optimizer has the advantage that the compiler can choose execution algorithms that create independent partitions that are suitable for multiple functions at the same time. Through this reuse of independent partitions, the program execution can become much more efficient. Furthermore, the compiler can examine the data characteristics to find an efficient way to build the partitions the specific environment.

PACT programs are not fully declarative (unlike SQL queries), because they contain imperative implementations of the user-defined functions. However, the contracts of the second-order functions are declarative: They state requirements for the independent partitions, rather than defining by which algorithm and data movement strategy the partitions are formed. In a large number of use cases, the user-defined functions perform rather simple operations on the partitions of data. Consequently, the contracts that describe requirements on how form the data partitions provide the most important information with respect to finding an efficient way to execute the program. The definition of PACT's set of second-order functions follows the principle to expose maximal information with a small set of simple primitives. We use the following rational to arrive at a small, orthogonal, and automatically optimizable set of second-order functions, which form the programming model's primitives:

A complete set of second-order functions must contain at least a record-at-a-time function, a group-at-a-time function, and operators to combine different inputs. To combine multiple inputs, the programming model requires in the most generic case simply a *bag union* operator. If we assume that records retain their lineage (the information from which original data set they came), we can express any further binary operators as group-at-a-time function over the unioned data set. We argue that a Cartesian product operator is crucial as well, because the only way to produce Cartesian product using a union is with a group-at-a-time operation over a single set that containing the entire result of the union - a technique that is by definition non parallelizable.

The *Map* and *Reduce* functions, as suggested by Dean et al. in the original MapReduce paper [44] represent generic record-at-a-time, respectively group-at-a-time functions. With *Union* and *Cross* (Cartesian Product), we have the core set of orthogonal second-order functions. In fact, applications on top of MapReduce implement all their operations through combinations of these for functions[2].

[2]Cf. the implementation of a join in MapReduce in Section 2.2.3. While MapReduce has no explicit Cartesian product operator, one can emulate it using utilities like the *distributed cache* [106].

These four primitives constitute a minimal set of parallelizable functions that allow programmers to express most operations. However, following PACT's central requirement to specify constrains transparently, it is imperative to express binary operations as such to allow the system to treat the two inputs differently:

- We add a binary group-at-a-time operation (*CoGroup*) that groups each input according to a key (similar as Reduce) and joins groups with equivalent keys. CoGroup can be thought of as the transparent version of the reducer on a tagged union input.
- Following the same line of arguments, we introduce a binary record-at-a-time operation (*Match*) that does not associate all elements pairwise, but only those with matching keys.

Figure 2.4 and Figure 3.1 illustrate how the second-order functions Map, Reduce, Cross, Match, and CoGroup define their data partitions from the input data set(s). A formal definition of these rules can be found in our previous publication [22]. Note that both Reduce and CoGroup can work as a unary-, respectively binary-, operation over the entire input, if the key is specified to be null.

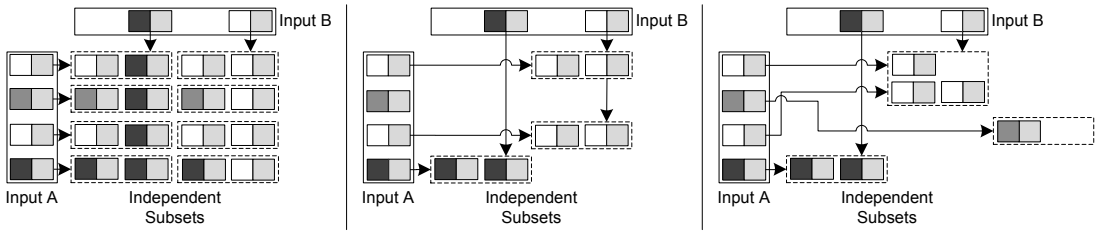

Figure 3.1.: Illustration of partitioning records into independent sets in Cross (left), Match (middle) and CoGroup (right). Figure is taken from our previous publication [9].

Compared to the operators in the relational algebra, the *Map* function subsumes *selection* (σ) and projection without duplicate elimination. Since the Map function can produce multiple elements from a single record, it also subsumes the *unnesting* operation of the Non-First-Normal-Form (NF^2) algebra. *Projection* with duplicate elimination (π) is expressible through *Reduce*. Both sets of primitives contain *Cartesian product* ($\times$) and *union* ($\cup$). Using the PACT primitives, relational *set intersection* ($\cap$) and *set difference* ($\setminus$) can be represented through *CoGroup*. One can express *aggregations* (as defined in the extended relational algebra) through a *Reduce* function. Joins ($\bowtie$) and semi joins ($\ltimes, \rtimes$) are a natural fit for the Match operation - in fact, Match can be represented through a join followed by a Map over each joining pair of records. Anti-joins ($\triangleright, \triangleleft$) can be represented through *CoGroup* by checking if the joining set is empty.

These apparent analogies between PACT and the relational algebra lead to a different view on the PACT primitives: They resemble relaxed versions of the relational algebra operators. The PACT primitives only specify the actual set operations (e. g., grouping, associating) without describing the exact record manipulations (e. g., concatenation, elimination, aggregation, field manipulations). This naturally allows the PACT compiler and optimizer to treat the PACT primitives in a very similar way as the relational database optimizer treats relational operators. In that sense, PACT programs can be thought of as small sequential building blocks that are assembled into a program using declarative rules (expressed through the second-order functions) on how to pass data between the building blocks. In [22], we have outlined the implementation of a System R style optimizer [94] for PACT programs. The paper describes which interesting properties each second-order function generates, and which candidate algorithms and operations (such as partitioning exchange, replication, sorting, hashing) each second-order function instantiates. We distinguish between *ship strategies* that reorganize partitions across machines or compute cores, and *local strategies* that organize data within a partition to form the final independent subsets. We adopted and extended the set of possible operations and algorithms that are known from parallel databases systems [45, 109]:

- *Map* is a trivial second-order function, treating every record independently. It requires no special runtime strategies.

- *Reduce* processes data grouped on a key. It requires the data to be partitioned on that key across machines. By default, a hash partitioning scheme is used [52], but range partitioning may be chosen in cases where the result is consumed by an operator that requires range partitioned data. Within each machine, the data is grouped though sorting. If the Reduce function is combinable (i. e., it implements an associative aggregation function), a running hash grouping and aggregation may be chosen as well (cf. [53]).

- *Cross* has as candidate ship strategies the different variants of the *"(a)symmetric fragment and replicate"* algorithm, of which fully replicating (broadcasting) one input data set is a special case. Possible local strategies include *nested-loops join* and *block-nested-loops join*. For both, the optimizer decides which input to make the outer- and which the inner side.

- The ship strategy for *Match* is chosen from partitioning the inputs (*partitioned join*), or replicating one of them (*broadcast join*). Local strategies include *hybrid hash join* (with choice of build side) and *sort-merge join*.

- *CoGroup* requires a partitioning similar to Reduce (hash- or range based). Locally, CoGroup uses a variant or the sort-merge join strategy that merges groups instead of records.

The first version of the Stratosphere system [97] implemented the PACT programming model as described here and in [22]. As stated before, the System used a classical System-R style optimizer [94] with partitioning and order as interesting properties. The distributed runtime that executed the query plans produced by the optimizer was the Nephele streaming system [103] (cf. Figure 3.2). The latest version of the Stratosphere system (at the time of writing this thesis) uses an optimizer that works with a generalization of the concept of *interesting properties* [58]. Very similar techniques have been published in the context of the SCOPE system's optimizer [114, 115]. In Chapter 4, we discuss extensions of these techniques.

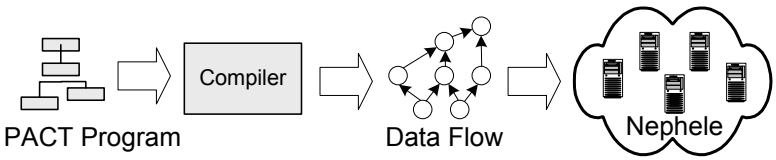

Figure 3.2.: High level view of a PACT program being translated to a parallel data flow, which is then turned into a schedule for the Nephele streaming system.

3.2.1. Implications of the Data Model

We have seen that the PACT model uses declarative rules (defined by the second-order functions) to define how the data should be organized in order to correctly apply the user-defined functions on the individual partitions in parallel. These declarative rules leave degrees of freedom for the optimizer, because there are often several alternative strategies and algorithms that fulfill them. This sections discusses the influence that the data model has on the optimization across second-odder functions.

The data model does not constrain the degrees of freedom that the optimizer has when choosing between strategies for each individual function, but does influences the potential for optimization across individual functions. MapReduce, as well as the original version of the PACT programming model build on top of a key/value data model. The key has the special role to describe which records belong together into one group, or which ones should be associated between inputs. The value is to the MapReduce- or PACT framework simply a payload and is only interpreted by the user-defined function. The key that is relevant to the next function is set by its preceding function - this design effectively moves some fragment of the logic of each function into its predecessor function(s), namely the logic that selects the part of the data record that is relevant for the data preparation of the successor.

For optimization across functions, the relationship between the rules of succeeding functions is of great importance. It allows the optimizer to extend its search space and consider solutions where one data reorganization builds partitions that are suitable for multiple functions at the same time. Recall the example from Section 2.2.3, where the second reducer can reuse the partitioning of the first reducer. The relationship between the functions' partitioning rules depends on the relationship between the keys of the two functions. However, the information about the relationship between the keys is hidden in the code of the first reduce function, in the logic that produces the result key from the input key/value pairs. To expose that information to the system, we suggested to add lightweight annotations called *"Output Contracts"* to the functions that declare relationships between the key passed to a function invocation and the keys produced by this invocation. Examples for such output contracts are *"Same-Key"*, *"Super-Key"*, or *"Sub-Key"*, depending on whether the produced key is exactly the same, has a super-set of fields, or a subset of fields.

Even with such explicit knowledge, the key/value data model's characteristic of having a single key field to expose the relevant parts of the records are fundamentally limiting the optimization potential across operators. We illustrate that at the example of a simple program that processes the data in the manner of the following query, inspired by the TPC-H decision support benchmark [100]:

```
1   SELECT l_orderkey, o_shippriority, sum(l_extendedprice)
2     FROM orders, lineitem
3    WHERE l_orderkey = o_orderkey
4      AND o_custkey IN [X]
5      AND o_orderdate > [Y]
6 GROUP BY l_orderkey, o_shippriority;
```

Listing 3.1: A relational select/project/join/aggregate query in the style of the TPC-H Query 3 [100], expressed in SQL.

A PACT program resembling this query would typically use a Map over each input to filter and/or project the tuples. A Match function implements the join and a Reduce function the successive aggregation. The Match function is annotated with a *Super-Key* output contract, because it is invoked with the key *o_orderkey* and produces a the composite key $(o\_orderkey, o\_shippriority)$. A typical execution plan is for this query, as created by the original implementation of the Stratosphere system [22] is shown in Figure 3.3. To build the pairs of matching elements for the Match, the system partitions the inputs on the key (similar to a partitioned join in a parallel database [52]) and sorts the inputs. The matching itself happens by merging the two sorted inputs. Note that the data is not repartitioned after the Match function. Instead, the Reduce function may reuse the partitioning (because it operates on a super key of the partitioning key), but it must still re-establish the necessary grouping within partitions, here realized through a

sorting operation.

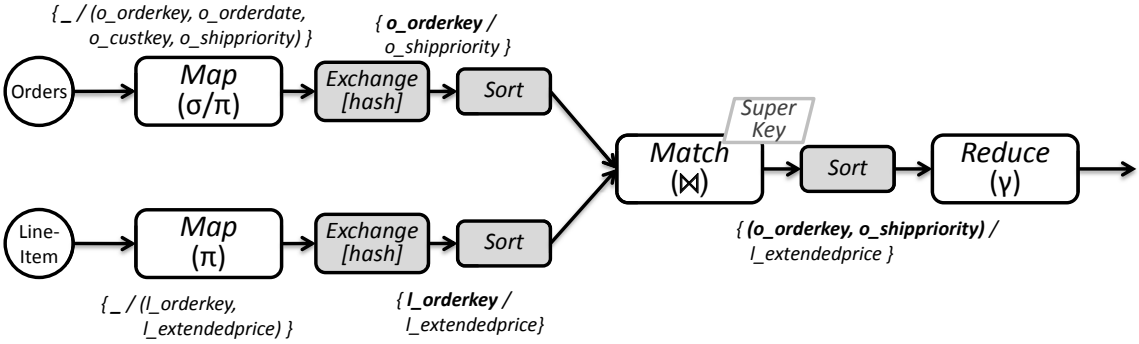

Figure 3.3.: The execution plan for a PACT program imitating the query from Listing 3.1, using the key/value data model. The *Match* function has a *Super-Key* output contract, allowing the system to infer that the partitioning may be reused for the Reduce.

The execution plan can be further improved if the data model is not key/value pairs, but tuples, like in relational databases. In the context of such a tuple data model, Match and Reduce functions are parametrized to interpret one or more fields of the tuple as the key. Figure 3.4 shows the above example rewritten for that data model. Match takes the first field from both its inputs as the key, and the Reduce function takes the first two fields as the key. The relationship between keys is directly expressed in these parametrization. Rather than needing an output contract that provides information about how the user code changes the key, all the system needs to know is whether fields are changed by the function. With this information, the optimizer can improve the plan and eliminate an additional operation. Because the fields that the Reducer uses for grouping are already visible to the system before the Match function, the optimizer can sort the records after the first two fields and use this order for both the Match on the first field and the grouping on the first two fields. This simple example illustrates the effect of the data model on cross-operator optimization potential. Note that the data model described here does not imply that the system is restricted to flat data types. It simply means that there are more addressable fields in one element than a key and a value. The fields that are not referred to by functions as key contributing fields may still have arbitrary data types that are only interpreted by the user-defined function implementation. The key/value data model is in fact a special case of this data model with exactly two fields. To distinguish this data model from the transparent and flat tuple data model used by relational database systems, we refer to it as the *record data model*.

A fitting analogy to how the data model affects the optimization potential is the opti-

mization of code blocks in programming language compilers. These compilers typically perform a data flow analysis of the variables within and between code blocks, for example to eliminate unnecessary accesses, detect constants, or reorder statements and blocks. We mentioned earlier how PACT programs can be thought of as sequential blocks of code that are connected together. In the key value model, the system has notion of a single variable (the key) across these blocks, whose contents is altered inside the blocks in a non transparent way. In the record model, the system has notion of multiple variables (the fields) which are transparently accessed. If they remain constant, the system can perform cross-block optimization. In fact, the highest optimization potential exists, if all fields are never overwritten, but every modification creates a new (logical) field in the record. This resembles the program representation of *static-single-assignment* [10], which has been widely recognized as a highly effective representation for program analysis and optimization. This treatment of record fields in a static-single-assignment fashion is equivalent to the *global schema* discussed by Hueske et al. for optimization of UDF data flows [63].

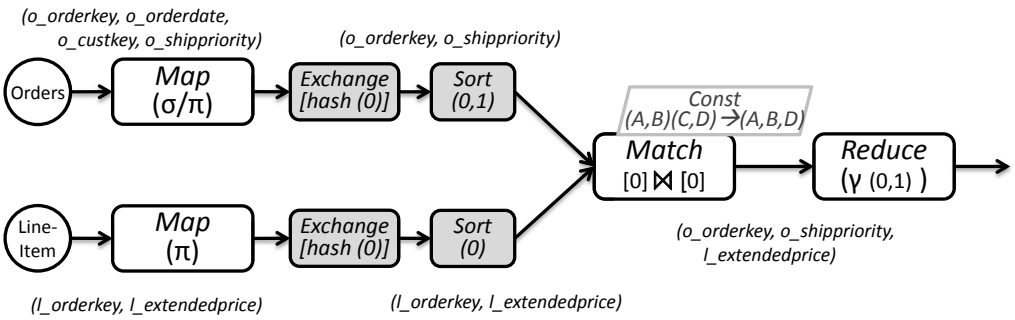

Figure 3.4.: The execution plan for a PACT program imitating the query from Listing 3.1, using a record data model. The *Match* is here shown with additional schema data that describes how input and output fields relate.

In addition to inhibiting optimization potential, the key/value data model is also detrimental for the re-usability of functions, for examples as part of libraries. Each function contains part of the logic of its successor hardwired in its implementation. This ties a function strongly to its context in the program. In contrast, the record model makes each function independent from its predecessor and successor. The specification of the fields that act as the key is part of the program specific parametrization of a function. Different function contexts in different usages of the function have a different record schema, which would require a polymorphic function signature to make the same function implementation work in these various contexts. Accordingly, Stratosphere's PACT abstraction adopted a model of variable-length and dynamically-typed sparse tuples. This allows for an increased re-usability of the function code, but is paid for by runtime schema interpretation and runtime reflection on the specific data types - both of which result

often in significant overhead during program execution. From the system design point, there is a conflict between the two goals to not strongly bind to the function's context for code re usability and optimizability, and to strongly bind to it for runtime efficiency. The conflict is solved in relational databases through the separation of query specification and the execution by means of a compiler and code generation. In Chapter 5, we outline an equivalent solution for an object-functional programming language.

3.2.2. Mixing PACT and the Relational Model

Even though the original version of PACT described a UDF-only programming model, it is naturally possible to combine the second-order functions and relational operators. The second-order functions can be used to realize parallelizable UDFs inside relational database management systems, in a similar fashion as AsterData uses MapReduce-style UDFs in their database [51]. Similarly, one could add relational operations to the PACT model as special cases and specific implementations of the user-defined functions. The same way as the here described set of PACT primitives relates to the relational algebra, one can devise new types of second-order functions from extended database operators. Examples of such additional functions are *top-k* operators and a sliding window version of Reduce that works similar as sliding-window aggregates in SQL.

The biggest problem when mixing relational operations with PACT's functions is to bridge between the data models. While relational operations are defined on flat tuples, the records produced by the PACT functions may have arbitrary types representing nested and recursive structures. This is a classical example of the impedance mismatch between databases and user-defined logic (here in UDFs rather than in the database client). One could place a restriction on the inputs and output data types of the user-defined functions, allowing them only to use a compatible flat data model. In fact, this is how interactions between UDFs and operations happen in the "Language-on-top-of-MapReduce" solutions Hive [99] and Pig [85]. However, this requires UDFs to translate their data structures, if possible, and it voids the opportunity to use relational operations in the settings where the data types have complex structures. Current data-parallel systems with high-level operators are consequently used in two separate modes: Running queries on a simple data model with high-level operators, or implementing low level programs that may use complex data structures. We will present an elegant solution that mitigates this problem almost entirely in Chapter 5.

Given that all relational operations are expressible as variants of the here discussed second-order functions, we will use the names of the second-order functions in the example throughout this thesis. That way, we show how the described techniques are applicable to generic data parallel programming. Naturally, all techniques described in

this thesis work therefore especially in the context of relational database management systems.

3.2.3. Limits of PACT's Expressiveness

The PACT abstraction is very expressive. In a previous publication [9], we have discussed its applicability to several domains. Beyond being applicable to all problems that MapReduce can handle, we have shown that it can elegantly express programs from many domains without workarounds that make the implementations more difficult and that reduce optimization potential. Examples are parallel programs realizing relational queries and XQueries (including text data types), clustering functions, finding triangles and squares in community graphs, and step functions for graph analysis.

However, there are some crucial classes of algorithms that the here presented version of PACT is not applicable to. Examples are algorithms based on parallel graph traversal (graph path problems like distances or centrality), multi-pass clustering algorithms (like K-Means), optimization algorithms based on gradient descend, regression algorithms, or the low-rank factorization of matrices. A common characteristic of all of these algorithms is that they are iterative or recursive in nature and make multiple passes over the input data and/or a partial intermediate solution. The common workaround for these algorithms is to write a PACT (or MapReduce) program that evaluates the iterative or recursive step function once. A driver program invokes that program multiple times, each time setting the output of the previous program as the input of the next invocation. This solution has several major problems, ranging from performance to optimizability and fragmentation of program logic. We discuss these problems in detail in Chapter 4 and offer an elegant and novel solution.

Many operations that repeatedly occur as part of data analytics problems are expressible through PACTs, but require multiple functions used together. For example the multiplication of two matrices, represented either cell-wise (typical for sparse matrices) or as row or column vectors, consists of a Match- and a Reduce function together. Re-usability of such operations, for example in libraries, goes beyond the re-usability of individual functions. Rather than implementing dedicated primitives for these operations on the programming model level, we abstract the into composite functions, using an elegant way described in Chapter 5.

3.3. Related Work

The PACT programming model incorporates concepts and ideas from *MapReduce*, the *Stream Programming Model*, and *Relational Databases*. In the following, we will discuss

similarities and differences between PACT and these techniques, as well as other data-parallel programming models.

PACT is a generalization of MapReduce [44]. It builds on second-order functions that can be automatically parallelized. There are however crucial differences between MapReduce and PACT: First, we add additional second-order functions that fit many problems which are not naturally expressible as a *Map* or *Reduce* function. Second, in current MapReduce implementations [44, 106], the programming model and the execution model are tightly coupled – each job is executed with a static plan that follows the steps map/combine/shuffle/sort/reduce. In contrast, PACT separates the programming model and the execution and uses a compiler to generate the execution plan from the program. For several of the new second-order functions, multiple parallelization strategies are available. Third, MapReduce loses all semantic information from the application, except the information that a function is either a *Map* or a *Reduce*. PACT preserves more semantic information through both a larger set of second-order functions and through the lightweight mechanism of code annotations. Last, PACT builds on a record data model that better supports optimization across operators than MapReduce's key/value data model.

An extension to MapReduce with an additional function called *Merge* has been suggested in [110]. *Merge* is similar to our *CoGroup* function, but closely tied to the execution model. The programmer must define explicitly which partitions are processed at which instances. The work focuses only on joins and how to realize different local join algorithms (Hash-Join/Merge-Join) rather than different parallel join strategies (re-partitioning, asymmetric-fragment-and-replicate. etc.).

PACT programs resemble DAG data flows, similar to programs in Dryad [64] and Nephele [103]. While the DAG data flows from the stream abstraction in Dryad and Nephele do not expose any semantics of the executed functions or tasks, the PACT model exposes the set operations through the different second-order functions. Systems like Dryad and Nephele are suitable execution engines for the execution plan created from compiling and optimizing PACT programs. Stratosphere uses Nephele to execute compiled PACT programs.

The Cascading framework [37] offers a DAG data flow abstraction with more primitives than MapReduce. Some of the additional primitives are similar to those in PACT, for example CoGroup. There are, however, two fundamental differences: First, the primitives in Cascading are for programming convenience (like special templates of Map or Reduce functions), while in PACT the primitives are an orthogonal set of data-parallel operations. Second, Cascading breaks its programs down into a series of MapReduce program, following a fix mapping of primitives to Map or Reduce. In contrast, PACT uses an optimizer to determine the execution plan for a program. In different programs, the same primitive may result in different runtime operators.

Similar to Cascading, Spark defines flexible programs with a larger set of function primitives. Spark does not perform any optimization on the program data flow, but executes each primitive with a specific runtime strategy. Similar to PACT (and in contrast to Cascading), Spark does not use MapReduce to execute its program, but comes with its own distributed execution engine.

PACT is largely inspired by relational database techniques. PACT primitives are similar to relational set operations, the runtime strategies are well known parallel database systems like *Gamma* [45] and *Grace* [52], and the optimizer inherits its basic principal from the System-R [94] and the Volcano [58] optimizers. PACT can be thought of as a data-model free variant of relational systems - the primitives describe set operations without record manipulations. This idea allows the PACT abstraction to be used in a larger variety of use cases. The PACT optimizer can also deal with various types of higher level languages, as long as its primitives map to similar set operations.

SCOPE [39] and DryadLINQ [111] are similar to our approach, as they compile declarative or more abstract programs to a parallel data flow for Dryad. The first one generates the parallel data flow from an SQL query, the second one from a .NET program. Both restrict the programmer to their specific data model and set of operations (relational, resp. LINQ). They optimize the parallelization in a domain specific way during the compilation. Our system differentiates itself from them by being more generic. It can parallelize and optimize the execution independently of a particular domain by the virtue of the PACT abstraction.

Related techniques for the optimization of parallelism beyond parallel databases are found in the languages that are put on top of MapReduce. Pig [84,85], JAQL [26], and Hive [99] generate MapReduce jobs for Hadoop from declarative queries. Some perform domain specific optimization and all of them apply simple heuristics to improve performance, such as chaining of *Map* operations. Both aspects are orthogonal to our optimization of the parallelization. Hive optimizes join parallelism by choosing between (still inefficient) *Map*-side joins and *Reduce*-side joins. These domain specific optimization techniques are tailored specifically towards equi-joins, similar to how parallel databases perform them. The alternative execution strategies for our *Match* contract encapsulate that optimization in a more generic fashion.

3.4. Conclusions

In this chapter, we have presented a generic programming abstraction for data parallel analysis programs. At the abstraction's core are *Parallelization Contracts* (PACTs). PACT generalizes MapReduce by adding additional primitives and relaxing constraints on program composition. It is a more generic form of the relational algebra, defining only set

operations and leaving the exact record manipulations to user-defined functions. PACT programs are optimizable with relational-style techniques. With respect to parallelization, PACT uses second-order functions to abstract one or more distributed communication patterns. The runtime is independent of the programming abstraction. In our system, we use the Nephele engine that is able to execute arbitrary acyclic data flows.

We believe that PACT is a sweet spot between the relational model and MapReduce. Similar to MapReduce, it has a rather generic level of abstraction that fits complex data processing tasks. At the same time, it captures a reasonable amount of semantics, which can be used to compile and optimize the program, producing an efficient data flow. We have successfully demonstrated that PACT programs, compiled to Nephele data flows, exhibit very good performance, compared to MapReduce and other systems. A summary of these performance experiments can be found in [8].

4. An Abstraction for Iterative Algorithms

The previous chapter presented the *Parallelization Contract* (PACT) programming model as an abstraction for parallelizable analysis programs. This chapter presents techniques to incorporate iterative algorithms in an efficient way. We present solutions to the specification of iterative data flows which maintain state across iterations, as well as optimization techniques for these iterative data flows.

The works presented in this chapter have been published in [48] (VLDB 2012) and presented as a demonstration [47] (SIGMOD 2013).

4.1. Introduction

Parallel data flow systems are an increasingly popular solution for analyzing large data volumes. They offer a simple programming abstraction based on directed acyclic graphs, and relieve the programmer from dealing with the complicated tasks of scheduling computation, transferring intermediate results, and dealing with failures. Most importantly, they allow data flow programs to be distributed across large numbers of machines, which is imperative when dealing with today's data volumes. Besides parallel databases [45, 52], MapReduce [44] is the best known representative, popular for its applicability beyond relational data and relational queries. Several other systems, like Dryad [64], Hyracks [31], and Stratosphere [22], follow that trend and push the paradigm further, eliminating many shortcomings of MapReduce.

While data flow systems were originally built for tasks like indexing, filtering, transforming, or aggregating data, their simple interface and powerful abstraction have made them popular for other kinds of applications, like machine learning [14] or graph analysis [70]. Many of these algorithms are of *iterative* or *recursive* nature, repeating some computation until a condition is fulfilled. Naturally, these tasks pose a challenge to data flow systems, as the flow of data is no longer acyclic.

During the last years, a number of solutions to specify and execute iterative algorithms as data flows have appeared. MapReduce extensions like *Twister* [46] or *HaLoop* [34], and frameworks like *Spark* [112] are able to efficiently execute a certain class of iterative algorithms. However, many machine learning and graph algorithms still perform poorly, due to those systems' inability to exploit the (sparse) computational dependencies present

in these tasks [77]. We refer to the recomputed state as the partial solution of the iteration, and henceforth distinguish between two different kinds of iterations:

- *Bulk Iterations*: Each iteration computes a completely new partial solution from the previous iteration's result, optionally using additional data sets that remain constant in the course of the iteration. Prominent examples are machine learning algorithms like Batch Gradient Descend [104] and Distributed Stochastic Gradient Descent [116], many clustering algorithms (such as K-Means), and the well known PageRank algorithm[1].

- *Incremental Iterations*: Each iteration's result differs only partially from the result of the previous iteration. Sparse computational dependencies exist between the elements in the partial solution: an update on one element has a direct impact only on a small number of other elements, such that different parts of the solution may converge at different speeds. An example is the Connected Components algorithm, where a change in a vertex's component membership directly influences only the membership of its neighbors. Further algorithms in this category are many graph algorithms where nodes propagate changes to neighbors, such as shortest paths, belief propagation, and finding densely connected sub-components. For certain algorithms, the updates can be applied asynchronously, eliminating the synchronization barrier between iterations.

Existing iterative data flow systems support bulk iterations, because those iterations resemble the systems' batch processing mode: the algorithms fully consume the previous iteration's result and compute a completely new result. In contrast, incrementally iterative algorithms evolve the result by changing or adding some data points, instead of fully recomputing it in a batch. This implies updating a mutable state that is carried to the next iteration. Since existing data flow systems execute incremental iterations as if they were bulk iterative, they are drastically outperformed by specialized systems [77, 78].

Existing data flow systems are therefore practically inefficient for many iterative algorithms. The systems are, however, still required for other typical analysis and transformation tasks. Hence, many data processing pipelines span multiple different systems, using workflow frameworks to orchestrate the various steps. Training a model over a large data corpus frequently requires a data flow (like MapReduce) for preprocessing the data (e. g., for joining different sources and normalization), a specialized system for the training algorithm, followed by another data flow for postprocessing (such as applying the model to assess its quality) [104].

[1] We refer to the original batch version of the PageRank algorithm. An incremental version of the algorithm exists [69].

We argue that the integration of iterations with data flows, rather than the creation of specialized systems, is important for several reasons: first, an integrated approach enables many analytical pipelines to be expressed in a unified fashion, eliminating the need for an orchestration framework. Second, data flows have been long known to lend themselves well to optimization, not only in database systems, but also when using more flexible programming models [22, 63]. Third, data flows seem to be a well adopted abstraction for distributed algorithms, as shown by their increased popularity in the database and machine learning community [14, 104].

The contributions of this chapter are the following:

- We discuss how to integrate bulk iterations in a parallel data flow system, as well as the consequences for the optimizer and execution engine (Section 4.3).
- We discuss an *incremental iteration* abstraction using *worksets*. The abstraction integrates well with the data flow programming paradigm, can exploit the inherent computational dependencies between data elements, allowing for very efficient execution of many graph and machine learning algorithms (Section 4.4).
- We implement bulk and incremental iterations in the Stratosphere system, and integrate iterative processing with Stratosphere's optimizer and execution engine.
- We present a case study, comparing the performance of graph algorithms in a state-of-the-art batch processing system, a dedicated graph analysis system, and our own Stratosphere data flow system that supports both bulk and incremental iterations. Our experimental results indicate that incremental iterations are competitive with the specialized system, outperforming both the batch system and Stratosphere's own bulk iterations by up to two orders of magnitude. At the same time, Stratosphere outperforms the specialized system for bulk iterations (Section 4.5).

4.2. Iterative Computations

This section recapitulates the fundamentals of iterations and different representations that lend themselves to optimized execution.

4.2.1. Fixpoint Iterations

An iteration is, in its most general form, a computation that repeatedly evaluates a function f on a partial solution s until a certain termination criterion t is met:

while $\neg t(s, f(s))$ **do**
    $s = f(s)$

A specific class of iterations are fixpoint computations, which apply the step function until the partial solution no longer changes:

while $s \neq f(s)$ **do**
    $s = f(s)$

For continuous domains, the termination criterion typically checks whether a certain error threshold has been achieved, rather than exact equality: $t(s, f(s)) \equiv (|s - f(s)| \leq \epsilon)$.

Fixpoint iterations compute the Kleene chain of partial solutions $(s, f(s), f^2(s), \ldots, f^i(s))$, and terminate when $f^k(s) = f^{k+1}(s)$ for some $k > 0$. The value k is the number of iterations needed to reach the fixpoint $f^k(s)$. Denote $s_i = f^i(s)$. Fixpoint iterations are guaranteed to converge if it is possible to define a *complete partial order* (CPO) $\preceq$ for the data type of s, with a bottom element $\perp$. Furthermore, the step function f must guarantee the production of a successor to s when applied: $\forall s : f(s) \succeq s$. The existence of a supremum and the guaranteed progress towards the supremum result in eventual termination.

Example: Connected Components. Assume an undirected graph $G = (V, E)$. We wish to partition the vertex set V into maximal subsets $V_i \subseteq V$ such that all vertices in the same subset are mutually reachable. We compute a solution as a mapping $s : V \to \mathbb{N}$, which assigns to each vertex a unique number (called component ID) representing the connected component the vertex belongs to: $\forall v \in V_i, w \in V_j : s(v) = s(w) \Leftrightarrow i = j$. Algorithm FIXPOINT-CC in Table 4.1 shows pseudocode of the pure fixpoint implementation of the Connected Components algorithm. The algorithm takes as input the set of vertices V and a neighborhood mapping $N : V \to V^*$, which assigns to each vertex the set of its immediate neighbors: $\forall x, v \in V : x \in N(v) \Leftrightarrow (v, x) \in E \vee (x, v) \in E$. The mapping

Iteration Template	Connected Components
1: **function** FIXPOINT(f, s) 2: **while** $s \prec f(s)$ **do** 3: $s = f(s)$	1: **function** FIXPOINT-CC(V,N) 2: **while** $(\exists v, x \in V \mid x \in N(v) \wedge s(x) < s(v))$ **do** 3: **for** $(v \in V)$ **do** 4: $m = \min\{s(x) \mid x \in N(v)\}$ 5: $s(v) = \min\{m, s(v)\}$
1: **function** INCR(δ,u,s,w) 2: **while** $w \neq \emptyset$ **do** 3: $w' = \delta(s, w)$ 4: $s = u(s, w)$ 5: $w = w'$	1: **function** INCR-CC(V,N) 2: **while** $w \neq \emptyset$ **do** 3: $w' = \emptyset$ 4: **for** $(x, c) \in w$ **do** 5: **if** $c < s(x)$ **then** 6: **for** $z \in N(x)$ **do** 7: $w' = w' \cup \{(z, c)\}$ 8: **for** $(x, c) \in w$ **do** 9: **if** $c < s(x)$ **then** 10: $s(x) = c$ 11: $w = w'$
1: **function** MICRO(δ,u,s,w) 2: **while** $w \neq \emptyset$ **do** 3: $d = \mathbf{arb}(w)$ 4: $s = u(s, d)$ 5: $w = w \cup \delta(s, d)$	1: **function** MICRO-CC(V,N) 2: **while** $w \neq \emptyset$ **do** 3: $(d, c) = \mathbf{arb}(w)$ 4: **if** $(c < s(d))$ **then** 5: $s(d) = c$ 6: **for** $(z \in N(d))$ **do** 7: $w = w \cup \{(z, c)\}$

Table 4.1.: Classes of iterations and the corresponding implementations of the Connected Components algorithm. The **arb** function selects and removes an arbitrary element from a set.

s is the partial solution and is iteratively improved. Initially, $s(v)$ is a unique natural number for each vertex v (we can simply number the vertices from 1 to $|V|$ in any order). Line 2 of the algorithm corresponds to the termination condition $s \prec f(s)$, and lines 3-5 correspond to the partial solution update $s \leftarrow f(s)$: For each vertex, its component ID is set to the minimal component ID of itself and all its neighbors. Like all algorithms in the second column of Table 4.1, FIXPOINT-CC returns s as the result of the iteration.

The CPO over s is defined by comparing the component IDs assigned to vertices: $s \succeq s' \Leftrightarrow \forall v \in V : s(v) \leq s'(v)$. A simple supremum is the mapping that assigns zero to all vertices.

4.2.2. Incremental Iterations & Microsteps

For many fixpoint algorithms, the partial solution s is a set of data points and the algorithms do not fully recompute s_{i+1} from s_i, but rather update s_i by adding or updating some of its data points. Frequently, the change to a data point in one iteration affects only few other data points in the next iteration. For example, in most algorithms that operate on graphs, changing a vertex immediately affects its neighbors only. This pattern is often referred to as *sparse computational dependencies* [77, 78].

To support such algorithms efficiently, we can express an iteration using two distinct functions u and δ, instead of a single step function f. Algorithm Incr of Table 4.1 provides pseudocode for this iteration scheme. The δ function computes the *working set* $w = \delta(s, f(s))$, which is conceptually the set of (candidate) updates that, when applied to s, produce the next partial solution. The function u combines w with s to build the next partial solution: $f(s) = u(s, w)$. Because the evaluation of $f(s)$ is what we seek to avoid, we vary this pattern to evaluate δ on s_i and w_i to compute the next working set w_{i+1}.

The function u is typically efficient when w contains only candidate updates relevant to the current iteration. Consequently, this form of *incremental* iterations is of particular interest, if a δ function exists that is both effective, in that it adds only relevant candidate updates to w, and efficient, in that it does not require the evaluation of $f(s)$. Incremental iterations are similar to workset algorithms used in optimizing compilers when performing data flow analysis [68]. To the best of our knowledge, no formal characterization of the class of functions whose fixpoint computation is amenable to such an optimization exists. It is known that minimal fixpoints of all distributive functions $f : T \to T : f(A \vee B) = f(A) \vee f(B)$, where T is a semilattice with a meet operator $\vee$, can be executed in an incremental fashion [36].

Example: Connected Components. Algorithm Incr-CC in Table 4.1 shows pseudocode for the incremental implementation of the Connected Components algorithm. The working set w contains in each step the new candidate component IDs for a set of vertices. Initially, w consists of all pairs (v, c) where c is the component ID of a neighbor of v. For each vertex that gets a new component ID, δ adds this ID as a candidate for all of the vertex's neighbors (lines 4-7). The function u updates the partial solution in lines 8-10 of the algorithm. For each element of the working set, it replaces the vertex's component ID by a candidate component ID, if the latter is lower. This representation implicitly exploits the computational dependencies present in the algorithm: a vertex's component can only change if one of its neighbors' components changed in the previous iteration. In the algorithm, a new candidate component is only in the working set for exactly those vertices.

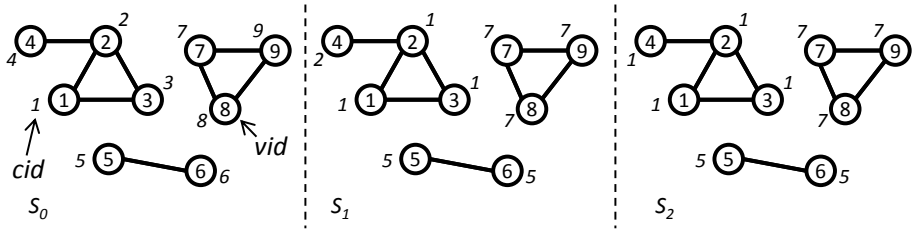

Figure 4.1.: Sample graph for the Connected Components.

In practice, one can frequently obtain an effective and efficient δ function by decomposing the iterations into a series of *microsteps*, and eliminating the *supersteps*. A microstep removes a single element d from w and uses it to update s and w, effectively interleaving the updates of the partial solution and the working set. Microstep iterations lead to a modified chain of solutions $(s \preceq p_{0,1} \preceq \ldots \preceq p_{0,n} \preceq f(s) \preceq p_{1,1} \preceq \ldots \preceq p_{1,n} \preceq f^2(s), \ldots)$, where $p_{i,j}$ is the partial solution in iteration i after combining the j-th element from w. The changes introduced by the element d are directly reflected in the partial solution after the microstep. Algorithm Micro of Table 4.1 shows the structure of a microstep iteration. The iteration state s and the working set w are both incrementally updated by looking at one element $d \in w$ at a time. Similar to superstep iterations, microstep iterations are guaranteed to converge, if each individual update to the partial solution leads to a successor state in the CPO. Note that this is a stricter condition than for incremental iterations, where all updates together need to produce a successor state.

Example: Connected Components. Consider the pseudocode for the Connected Components algorithm shown in Algorithm Micro-CC of Table 4.1. Inside each iteration, instead of performing two loops to update the state and the working set, these are simultaneously updated in one loop over the elements of the working set.

Note that in parallel setups, this last form of fixpoint iterations is amenable to asynchronous execution. The conformance of microsteps to the CPO can therefore enable fine-grained parallelism where individual element updates take effect in parallel, and no synchronization is required to coordinate the iterations/supersteps across parallel instances.

4.2.3. Performance Implications

The efficiency of bulk and incremental iterations may differ significantly. We illustrate this using the example of the Connected Components algorithm. In the bulk iteration algorithm, each vertex takes in each iteration the minimum component ID (*cid*) of itself and all its neighbors. Consequently, the number of accesses to the vertex state and the

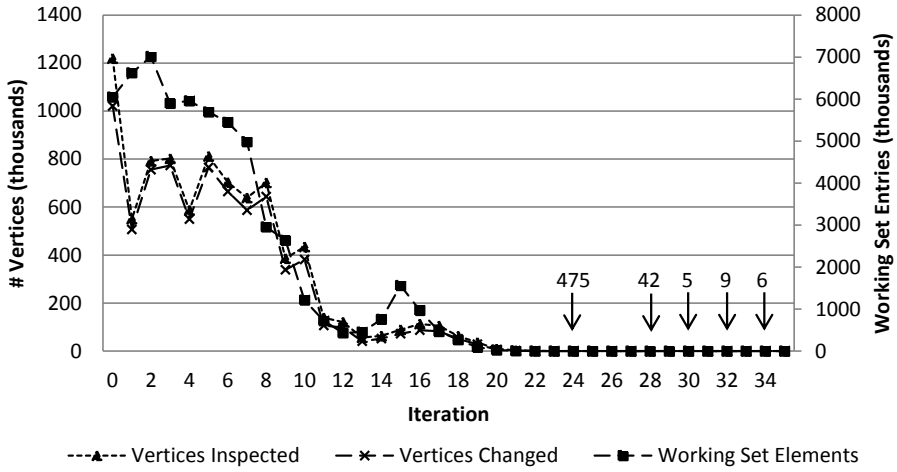

Figure 4.2.: The effective work the Connected Components algorithm performs on the FOAF subgraph.

neighbor set is constant across all iterations. For the incremental (and microstep) variant of the algorithm, the cost of an iteration depends on the size of its working set.

Consider the sample graph from Figure 4.1. The numbers inside the vertices denote the vertex ID (*vid*), and the number next to a vertex denotes its current component ID (*cid*). The figure shows how the assigned *cid* values change for each vertex over the three iterations. We can see that all except the vertex with $vid = 4$ reach their final *cid* in one step. For most vertices, all their neighbors reach their final *cid* in the first step as well. Those vertices need not be re-inspected. Their state cannot possibly change, since none of their neighbors' state did.

The incremental variant of the Connected Components algorithm reflects that by accessing only vertices, for which the working set contains new candidate *cid*s. That way, the algorithm focuses on "hot" portions of the graph, which still undergo changes, while "cold" portions are not accessed. The magnitude of this effect for a larger graph is illustrated in Figure 4.2. The graph is a small subset of the *Friend-of-a-Friend* network derived from a Billion-Triple-Challenge Web-Crawl and contains 1.2 million vertices and 7 million edges. The figure shows how the number of accesses and modifications to elements of the vertex states s (left y axis), as well as the number of records added to the working set (right y axis) vary across iterations. We see that the work performed in later iterations is significantly lower than the work in the first iterations, and that the actual progress (number of changed vertices) closely follows the size of the working set.

4.3. Bulk Iterations

This section describes the integration of bulk iterations into parallel data flows. Bulk iterations recompute the entire partial solution in each iteration.

4.3.1. Data Flow Embedding

We represent this general form of iterations as a special construct that is embedded in the data flow like a regular operator. An iteration is a higher-order operator defined as a tuple (G, I, O, T). G is a data flow that represents the step function $f : S \rightarrow S$, S being the data type of the partial solution. The partial solution corresponds to the pair (I, O), where I is an edge that provides the latest partial solution as input to G. O is the output of G, representing the next partial solution.[2] In each but the first iteration, the previous iteration's O becomes the next iteration's I. The iteration is embedded into a data flow by connecting the operator providing the initial version of the partial solution I to the operator consuming the result of the last iteration to O. T, finally, denotes the *termination criterion* for the iteration. T is an operator integrated into G and is similar to a data sink in that it has only a single input and no output. It contains a Boolean function that consumes the input and returns a flag indicating whether to trigger a successive iteration. Instead of a termination criterion, the number of iterations n may be statically defined. The iteration then is represented as a tuple (G, I, O, n).

Example: PageRank. The PageRank algorithm [86] finds the fixpoint $p = A \times p$, where p is the rank vector and A is the left stochastic matrix describing the probabilities $p_{i,j}$ of going from page j to page i. We represent the rank vector as a set of tuples (pid, r), where pid is the row index (and a unique identifier for the page), and r is the rank. The sparse stochastic matrix A is represented as a set of tuples (tid, pid, p), where tid is the row index (the unique identifier of the target page), pid is the column index (the unique identifier of the source page), and p is the transition probability. This representation does not include the tuples with $p = 0.0$. In each iteration the algorithm first joins the vector and the matrix tuple sets on pid, returning for each match $(tid, k = r * p)$. Second, all values are grouped by tid, which becomes the pid for the result vector, and all k are summed up to form the new rank r. Figure 4.3 shows the algorithm as an iterative data flow. The big dashed box represents the iteration construct. Here, the data flow G comprises the rightmost Match operator (which joins the vector and the matrix), the Reduce operator (which groups and recomputes the rank), the data source A, and their connecting edges. The termination criterion T uses another Match operator between the new and old rank data sets. The operator emits a record if a page's rank changed more

[2] The concept extends straightforwardly to multiple data sets and hence multiple pairs $(I, O)_i$. For ease of exposition, we present the unary case in our examples.

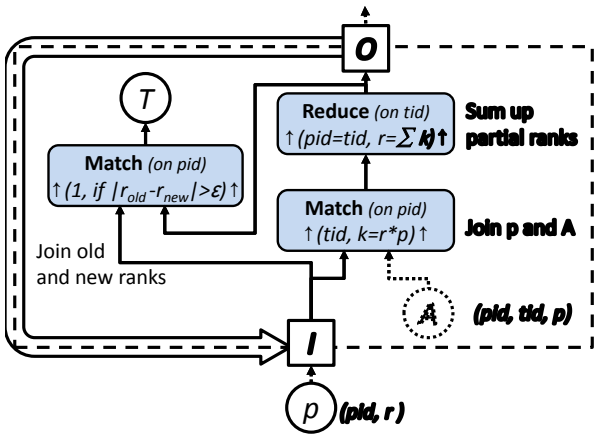

Figure 4.3.: PageRank as an iterative data flow.

than a threshold ϵ.

All nodes and edges in G on the path from I to O and from I to T process different data during each iteration. We call that path the *dynamic data path*. In contrast, all nodes and edges that belong to the *constant data path* process data that is constant across all iterations. In Figure 4.3, the dynamic data path consists of the solid edges and the Match and Reduce operators. The constant data path includes data source A and the edge connecting A to the Match operator. Note that data sources defined in G are always on the constant data path.

4.3.2. Execution

Executing an iterative data flow is possible by *"unrolling the loop"* or via *feedback channels*. When executing by unrolling, we roll out the iterations lazily: a new instance of G is created whenever O receives the first record and T has signaled that another iteration is needed. The new instance of G is concatenated to the current data flow by connecting the existing unconnected O to the new I. During an iteration, O may receive records before T has consumed all records, depending on which operators in G materialize their intermediate result. The PageRank example in Figure 4.3 describes such a case, as the Reduce operator emits records simultaneously to O and the Match that is input to T. Here, an extra pipeline breaker (a materializing "dam") must be added to O, preventing the next iteration from receiving records before the decision whether to have that iteration was made. In some cases, the operator succeeding I materializes its corresponding input (e.g. in a sort buffer or hash table). In that case, this specific materialization point serves as the dam and no extra dam is needed.

The feedback-channel based execution reuses G in each iteration. Each operator in G is reset after producing its last record. Similar as in the "unrolling" execution, the feedback edge materializes an iteration's result if O receives records before T decides upon a new iteration. Furthermore, if the dynamic data path contains less than two materializing operators, the feedback channel must also dam the data flow to prevent the operators from participating in two iterations simultaneously. For PageRank in Figure 4.3, the feedback channel needs an additional dam if either the Reduce is pipelined or the right hand side Match pipelines its input I.

For all massively parallel systems, resilience to machine failures is necessary to ensure progress and completion of complex data flows spanning many machines. Iterative data flows may log intermediate results for recovery just as non-iterative data flows, following their normal materialization strategies. In Stratosphere, for example, the Nephele execution engine judiciously picks operators whose output is materialized for recovery, trading the cost of creating the log against the potential cost of having to recompute the data. When executing with feedback channels, a new version of the log needs to be created for every logged iteration.

4.3.3. Optimization

Many data flow systems optimize data flow programs before executing them [22, 25, 45]. The optimization process typically creates an *execution plan* that describes the actual execution strategy for the degrees-of-freedom in the program. Such degrees-of-freedom comprise operator order, shipping strategies (partitioning, broadcasting) and local strategies (e. g., hashing vs. sorting operator implementations, as well as inner and outer role).

The optimizer may choose a plan for the iteration's data flow G following the techniques for non-iterative data flows [58, 94]. Note that in the general case, a different plan may be optimal for every iteration, if the size of the partial solution varies. The number of iterations is often not known a priori, or it is hard to get a good estimate for the size of the partial solution in later iteration steps. It is therefore hard for cost-based optimizers to estimate the cost of the entire iterative program. In our implementation in Stratosphere, we resort to a simple heuristic and let the optimizer pick the plan that has the least cost for the first iteration. For programs where the result size is rather constant across all iterations, that plan should be close to the best plan. To avoid re-executing the constant path's operators during every iteration, we include a heuristic that caches the intermediate result at the operator where the constant path meets the dynamic path. The caches are in-memory and gradually spilled in the presence of memory pressure. The cache stores the records not necessarily as an unordered collection, but possibly as a hash table, or B^+-Tree, depending on the execution strategy of the operator at the data flow

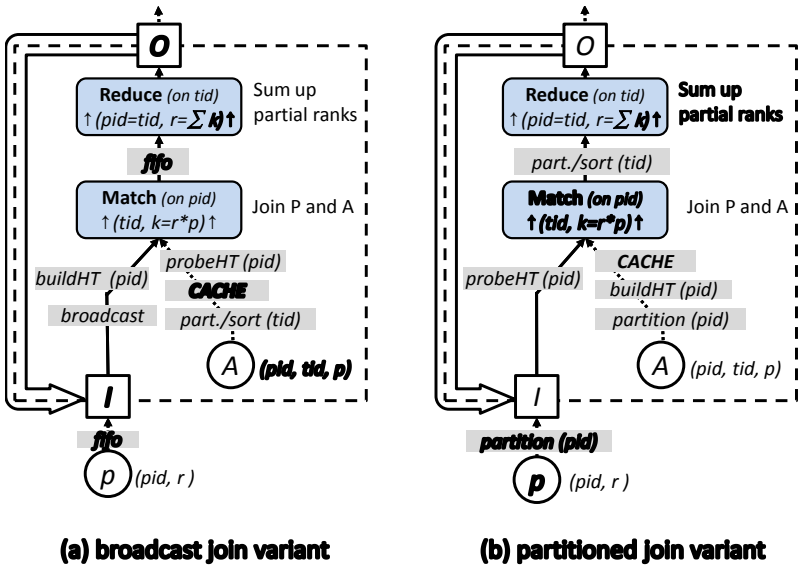

Figure 4.4.: Different execution plans for the PageRank iterative data flow. The gray boxes show the optimizer's strategy choices.

position where the constant and dynamic data paths meet. Finally, when comparing different plans, we weigh the cost of the dynamic data path by a factor proportional to expected number of iterations, since it is repeatedly executed. Plans that place costly operations on the constant data path are consequently cheaper than plans that place those operations on the dynamic data path.

Figure Figure 4.4 shows two different execution plans for the PageRank algorithm, as chosen by Stratosphere's optimizer depending on the sizes of p and A. The gray shaded boxes describe the optimizer's choices for execution strategies. It is noteworthy that the two plans resemble two prominent implementations of PageRank in Hadoop MapReduce [106]. The left variant, as implemented by Mahout [14] and optimized for smaller models, replicates the rank vector and creates a hash table for it. This variant avoids to repeatedly ship the transition matrix by caching it in partitioned and sorted form, such that grouping happens directly on the join result without additional partitioning/sorting. The right hand side variant, which is close to the PageRank implementation in Pegasus [70], joins a partitioned vector with the transition matrix and re-partitions for the aggregation. The transition matrix is here cached as the join's hash table. While in MapReduce, different implementations exist to efficiently handle different problem sizes, a data flow system with an optimizer, such as Stratosphere, can derive the efficient execution strategy automatically, allowing one implementation to fit both cases.

Many optimizers in data flow systems follow the Volcano approach [58], generating Interesting Properties (IPs) while they enumerate execution plan alternatives. Interesting Properties are used during plan pruning to recognize plans in which early operators can be more expensive, but produce data in a format that helps later operators to be executed more efficiently. For finding the optimal plan for G across multiple iterations, the IPs propagated down from O depend through the feedback on the IPs created for I, which themselves depend on those from O. In general, for an edge e that is input to operator P, its interesting properties are $\mathrm{IP}_e = \mathrm{IP}_{P,e} \cup \mathrm{AP}_e$, where $\mathrm{IP}_{P,e}$ are the IPs that P creates for that edge and $\mathrm{AP}_e \subseteq \bigcup_{f \succ e} \mathrm{IP}_f$ are the inherited properties, where $f \succ e$, if edge f is a successor to edge e in the DAG G. Note that $\mathrm{IP}_{P,e}$ only depends on the possible execution strategies for P. A Match creates for example "partitioning" or "replication" as IPs for both edges. Which IPs are inherited depends on which properties could be preserved through the possible execution strategies of P and the user code executed inside the operators[3]. The formula can be expanded to $\mathrm{IP}_e \subseteq \bigcup_{f \succ e, P \in G} \mathrm{IP}_{P,f}$. In the iterative setting, all edges on the dynamic data path are successors to all other edges, so an edge's interesting properties depend on all operators on the dynamic data path. To gather all relevant IPs for each edge, the optimization performs two top down traversals over G, feeding the IPs from the first traversal back from I to O for the second traversal.

In contrast to the methods originally described in [58], we interpret the interesting properties of an edge additionally as a hint to create a plan candidate that establishes those properties at that edge. The left hand plan in Figure Figure 4.4 is the result of that procedure: the edge connecting A and the Match operator has an interesting property for partitioning and sorting, generated by the Reduce operator. The plan candidate that applies partitioning and sorting at that point is actually very cheap in the end, because the expensive partitioning and sorting occur early on the constant data path.

4.4. Incremental Iterations

In this section we discuss how to integrate incremental iterations, as described in Section 4.2, into data flows.

4.4.1. Data Flow Embedding

An incremental iteration can be expressed using the bulk iterations introduced in Section 4.3, where the partial solution consists of two data sets: The solution set (S) and

[3] Reference [22] describes *OutputContracts* to determine how the user code behaves with respect to data properties.

the workset (W). The a step functions combines u and δ, reading both data sets and computing a new version of S and W. However, recall that the primary motivation for incremental iterations is to avoid creating a completely new version of the partial solution, but to apply point updates instead. The updated partial solution should be implicitly carried to the next iteration.

In imperative programming, updating the partial solution is achievable by modifying the statement $S = u(S, W)$ to $u(\&S, W)$, i. e., passing a reference to the state ($\&S$) of the partial solution and modifying that shared state. Data flow programs (like functional programs) require that the operators/functions are side effect free[4]. We work around this obstacle by modifying the update function from $S = u(S, W)$ to $D = u(S, W)$. The *delta set* D contains all records that will be added to the partial solution and the new versions of the records that should be replaced in the partial solution. The solution set S is treated as a set of records s, where each $s \in S$ is uniquely identified by a key $k(s)$. The delta set is combined with the solution set as $S = S \dot{\cup} D$. The $\dot{\cup}$ operator denotes a set union that, when finding two records from S and D with the same key, chooses the record from D: $S \dot{\cup} D = D \cup \{s \in S : \neg\exists d \in D | k(d) = k(s)\}$

We hence express an update of a record in the partial solution through the replacement of that record. The incremental iterations algorithm becomes

function INCR(δ,u,S,W)
  while $W \neq \emptyset$ **do**
    $D \leftarrow u(S, W)$
    $W \leftarrow \delta(D, S, W)$
    $S = S \dot{\cup} D$

Because the update function u and the working set function δ frequently share operations, we combine them both to a single function Δ, for ease of programmability: $(D_{i+1}, W_{i+1}) = \Delta(S_i, W_i)$

Example: Connected Components. The example follows the algorithm INCR-CC in Table 4.1. The solution set S is a set of pairs (vid, cid), which represents the mapping from vertex (vid) to component ID (cid). The vid acts as the key that uniquely identifies a record in S. The working set W contains pairs (vid, cid). Each pair in W is a candidate component ID cid for vertex vid. Figure 4.5 shows the algorithm as an incrementally iterative data flow. The box as a whole represents the iteration operator, containing the data flow for Δ that computes W_{i+1} and D_{i+1} from W_i and S_i. We compute D through an InnerCoGroup[5] operator that uses vid as key. The InnerCoGroup calls its UDF for

[4] An intransparent side effect would void the possibility of automatic parallelization, which is one of the main reasons to use data flow programming for large scale analytics.

[5] The InnerCoGroup is like a CoGroup, except that, much like an inner join, it drops groups where the key does not exist on both sides.

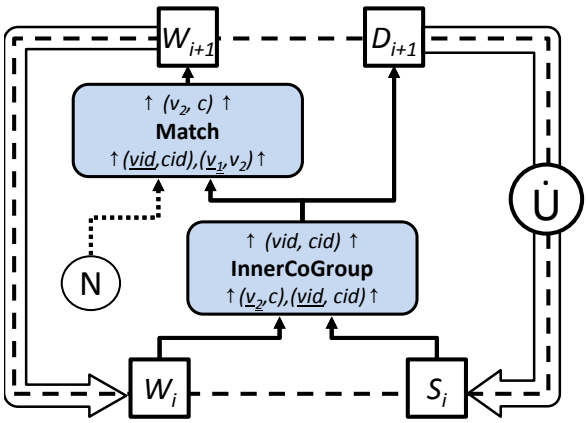

Figure 4.5.: The Connected Components algorithm as an Incremental Iteration.

each *vid* individually, providing the current *cid* for the vertex from the input S, and all candidate *cid*s from W. Among the candidates, it selects the minimal *cid* and, if that *cid* is smaller than the operators current *cid*, it returns a pair (vid, cid) with that new *cid*. When D is combined with S, each record in D replaces the record with the same *vid* in S, thereby effectively updating the associated *cid* for a *vid*.

The next working set W_{i+1} is computed though a single Match operator that takes the delta set D and joins it with the data source N that represents the neighborhood mapping. N contains the graph's edges as (vid_1, vid_2) pairs[6]. The Match joins D and N via $vid = vid_1$ and returns for each match a pair (vid_2, cid). Each returned pair represents the changed vertex's new *cid* as a candidate *cid* for its neighboring vertex vid_2.

In fact, the iterative data flow of Figure 4.5 can serve as a template for implementing a wide range of graph algorithms as incremental iterations. Every algorithm that can be expressed via a message-passing interface [13, 78] can also be expressed as an incremental iteration. $S(vid, state)$ represents the graph states, and $W(tid, vid, msg)$ represents the messages sent from vertex *vid* to vertex *tid*. In each superstep, Δ combines the current state S and messages W, it produces the changed states D, and creates the new set of messages using D and possibly the graph topology table N.

By computing a delta set instead of the next partial solution, we can achieve that the iteration returns fewer records when fewer changes need to be made to the partial solution (cf. later iterations in Figure 4.2). The solution set S is here a persistent state across the iterations, saving the cost of copying the unchanged records to the next iteration.

[6] We assume an undirected graph here, such that N contains for every edge (vid_1, vid_2) also the symmetric (vid_2, vid_1) pair.

Merging the small D with S is highly efficient if S is indexed by the key that identifies the records.

To generalize the aforementioned example, we represent an *Incremental Iteration* as a complex data flow operator, defined as a tuple (Δ, S_0, W_0). Let S denote the *solution set*. S is a set of records s that are identified by a key $k(s)$. In iteration i, S holds the i-th partial solution S_i. The initial version of S is S_0, which is input to the iteration. After the incremental iteration has converged, S holds the iteration's result. Let W_i denote the working set for iteration i. The initial working set W_0 is input to the iteration.

The step function Δ computes in iteration i the delta set D_{i+1} with all new and changed records for S_{i+1}, as well as the working set W_{i+1} for the next iteration: $(D_{i+1}, W_{i+1}) = \Delta(S_i, W_i)$. Δ is expressed as a data flow in the same way as G expresses the step function f for bulk iterations (Section 4.3). Since Δ must return two data sets, it is necessarily a non-tree DAG. After Δ is evaluated for iteration i, D_{i+1} is combined with S_i using the modified union operator $\dot{\cup}$, producing the next partial solution S_{i+1}. That implies that any accesses to S during the computation of D_{i+1} and W_{i+1} read the state of S_i. The iteration terminates once the computed working set is empty.

Because the delta set D is the result of a data flow operator, it is naturally an unordered bag of records, rather than a set. D may hence contain multiple different records that are identified by the same key, and would replace the same record in the current partial solution. The exact result of $S \dot{\cup} D$ is then undefined. In practice, many update functions create records such that only one record with the same key can exist in D. That is, for example, the case in the Connected Components algorithm, because the InnerCoGroup operator joins the each record in the partial solution exactly once on the key that indexes it (the *vid*), and the UDF does not modify that field. Hence, each *vid* appears at most once in the operators result. But since that is not necessarily the case in general, we allow the optional definition of a comparator for the data type of S. Whenever a record in S is to be replaced by a record in D, the comparator establishes an order among the two records. The larger one will be reflected in S, and the smaller one is discarded. The usage of a comparator naturally reflects the strategy to establish an order among two states before and after a point update, as done in the definition of the CPO for S. Selecting the larger element represents the record leading to a successor state. Because the record from D is dropped if it is the smaller one, D reflects only the records that contributed to the new partial solution.

4.4.2. Microstep Iterations

Section 4.2 discussed microstep iterations as a special case of incremental iterations. Recall that a microstep iteration is characterized by the fact that it takes a single element

d from the working set, and updates both the partial solution and the working set. Note that this implies that the partial solution already reflects the modification when the next d is processed.

We represent microsteps iterations through the same abstraction as incremental iterations. In our implementation, an incremental iteration may be executed in microsteps rather than supersteps, if it meets the following constraints: first, the step function Δ must consist solely of record-at-a-time operations (e. g., Map, Match, Cross, ...), such that each record is processed individually[7]. For the same reason, binary operators in Δ may have at most one input on the dynamic data path, otherwise their semantics are ill defined. Consequently, the dynamic data path may not have branches, i. e. each operator may have only one immediate successor, with the exception of the output that connects to D. Note that this implies that W_{i+1} may depend on W_i only through d, which is consistent with the definition of microstep iterations in Table 4.1, line 5.

Finally, for microstep iterations, we need to assure that each time Δ is invoked, the partial solution reflects a consistent state, reflecting all updates made by prior invocations of Δ. In a parallel system with multiple instances of Δ active at the same time, we encounter the classic transactional consistency problem, where guaranteeing a consistent up-to-date partial solution requires in the general case fine grained distributed locking of the elements in S (or their keys). We can avoid the locking, when an update of a record in S affects only the same partition of S that created the updated record. This is true, when the data flow between S and D does not cross partition boundaries, which is easily inferable from Δ. The sufficient conditions are that the key field containing $k(s)$ is constant across the path between S and D, and that all operations on that path are either key-less (e. g. Map or Cross) or use $k(s)$ as the key (for example in the case of the Match).

The Connected Components example above becomes amenable to microstep execution, if we replace InnerCoGroup operator by the record-at-a-time Match operator. Since its UDF keeps the key field constant, all records in D will replace records in S in the local partition.

4.4.3. Runtime & Optimization

The principal execution scheme for incremental iterations follows the bulk iteration scheme using feedback channels (Section 4.3.2). The techniques described for the partial solution in bulk iterations are used for the working set in the context of incremental

[7] For group- or set-at-a-time operations, supersteps are required to define the scope of the sets. Cf. systems for stream analytics, where windows or other reference frames are required to define set operations.

iterations. The dynamic and constant data path distinction applies directly to the Δ data flow in incremental iterations. The optimization, including caching static data and the modified handling of interesting properties happens the same way.

We extend those techniques to efficiently handle the persistent partial solution. To facilitate efficient access and updates to the partial solution, we store S partitioned by its key across all nodes. Each node stores its partition of S in a primary index structure, organized by the key. It is a typical pattern that the records from S are joined (or otherwise associated) by an operator o using the identifying key as the join key. In this case, we merge the S index into o, creating a stateful operator that uses the S index for its operation. The exact type of index is in this case determined by the execution strategy of o. For example, if o is a join operator and the optimizer chooses to execute that join with a hash strategy, then S will be stored in an updateable hash table. In contrast, if the optimizer picks a sort-based join strategy, S is stored in a sorted index (B^+-Tree). That way, both accesses to S by the o and record additions to S happen through the same index.

In the general case, we cache the records in the delta set D until the end of the superstep and afterwards merge them with S, to guarantee a consistent view of S during the superstep. Under certain conditions, the records can be directly merged with S, because we can guarantee that they will not be accessed in the same superstep again. Those conditions are equivalent to the conditions that guarantee updates to the partial solution to be local.

Figure 4.6 illustrates the resulting execution strategy for the Connected Components algorithm, as derived by the optimizer. The working set W is cached in queues, partitioned by *vid* for the subsequent Match with S. S is stored in a partitioned hash table using *vid* as the key. When the Match between W and S returns a record, the plan writes the record back to the hash table. Simultaneously, it passes the record to the second Match function, which creates the new (vid, cid) pairs for W. Note that the second Match uses the same key as the first Match, so it is not necessary to repartition the records by the join key. The contents of the data source N is cached in a partitioned way as a hash table, such that Δ becomes pipelined and local. The plan partitions Δ's result (the new working set) and adds the records to the corresponding queues.

If an iteration is amenable to execution in microsteps, the difference between superstep and microstep execution manifests itself only in the behavior of the queues that store W: for an asynchronous microstep iteration, they behave like regular non-blocking queues, passing records through in a FIFO fashion. In the presence of supersteps, the queues buffer all records that are added to them, but do not yet return them, as those added records are destined for the next iteration's working set. When all queues have returned all records for the current iteration, the superstep ends. The queues are then signaled to switch their buffers in a synchronized step, making the records added in the last

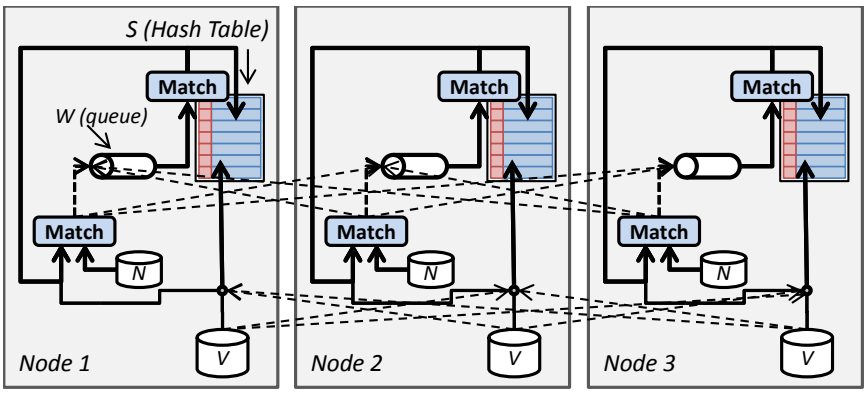

Figure 4.6.: The execution of the incrementally iterative Connected Components algorithm.

iteration available to the current and buffering the records added in the current iteration for the next one. In our implementation in Stratosphere, we currently make use of the channel events offered by the Nephele data flow engine to coordinate superstep transitions. Special channel events are sent by each node to signal the end of its superstep to all other nodes. Upon reception of an according number of events, each node switches to the next superstep.

In the synchronous case, the execution has converged if the working set W_i is empty at the end of iteration i. A simple voting scheme can here decide upon termination or progression. For the asynchronous execution, however, no end-of-superstep point exists. The distributed systems community has explored a variety of algorithms for termination detection in processor networks. Many of these algorithms apply also to the asynchronous execution of the parallel data flow. For example, [74] works by requesting and sending acknowledgments for the records along the data channels.

4.5. Evaluation

To evaluate the practical benefit of incremental iterations, we compare three systems that support iterations in different ways: *Stratosphere* [22], *Spark* [112], and *Giraph* [13].

Stratosphere supports both bulk and incremental iterations, which were implemented as described in Sections 4.3 and 4.4. The implementation uses the feedback-channel based execution strategy.

Spark is a parallel data flow system implemented in Scala and centered around the concept of Resilient Distributed Data Sets (RDDs). RDDs are partitioned intermediate results cached in memory. Spark queries may contain operations like Join, Group, or

CoGroup and apply user defined functions on the operators results. The system accepts iterative programs, which create and consume RDDs in a loop. Because RDDs are cached in memory and the data flow is created lazily when an RDDS is needed, Spark's model is well suited for bulk iterative algorithms.

Giraph is an implementation of Google's Pregel [78] and hence a variant of Bulk Synchronous Parallel processing adopted for graph analysis. The model is explicitly designed to exploit sparse computational dependencies in graphs. A program consists in its core of a vertex update function. The function computes its update based on messages it receives from other vertices, and sends out messages to other vertices in turn. Because the function has only a local view of one vertex, the algorithms have to be expressed by means of localizing the updates. Pregel is thus a special case of incremental iterations - the vertices represent the partial solution state and the messages form the working set.

All of the above systems run in a Java Virtual Machine (JVM), making their runtimes easy to compare. We ran our experiments on a small cluster of four machines. Each machine was equipped each with 2 Intel Xeon E5530 CPUs (4 cores, 8 hardware contexts) and 48 GB RAM. The machines' disk arrays read 500 MB/sec, according to `hdparm`. We started the JVMs with 38 GB heap size, leaving 10 GB to operating system, distributed filesystem caches and other JVM memory pools, such as for native buffers for network I/O. The cluster has consequently 32 cores, 64 threads and an aggregate Java heap size of 152 GB.

We use four different graphs as data sets, which we obtained from the University of Milan's Web Data Set Repository [87]: the link graph from the English Wikipedia and the Webbase web crawl from 2001 are typical web graphs. The Hollywood graph, linking Actors that appeared in the same movie, and the Twitter follower graph are representatives of social network graphs. The latter are typically more densely connected. Table 4.2 shows the graphs' basic properties.

DataSet	Vertices	Edges	Avg. Degree
Wikipedia-EN	16,513,969	219,505,928	13.29
Webbase	115,657,290	1,736,677,821	15.02
Hollywood	1,985,306	228,985,632	115.34
Twitter	41,652,230	1,468,365,182	35.25

Table 4.2.: Data Set Properties

4.5.1. Full Iterations

We first establish a base line among the three systems using the bulk iterative PageRank algorithm, as described in Section 4.3. For Giraph and Spark, we used the the algorithm implementations that were included with the system as example algorithms. Giraph's algorithm follows the example in Pregel [78], and Spark's implementation follows Pegasus [70]. For Stratosphere, we executed both strategies from Figure Figure 4.4. The partitioning plan (Figure 4.4 (b)) is equivalent to the Spark implementation. The broadcasting plan (Figure 4.4 (a)) is cheaper by network cost, because it computes the new ranks locally.

We run PageRank for 20 iterations. Even though computational dependencies do exist in the graphs, the algorithm operates in batches, updating each vertex every time. Consequently, we expect all systems to have roughly equal runtime for PageRank because all iterations do the same work: they create records for each edge propagating the current rank of a vertex to its neighbors. These records are pre-aggregated (cf. Combiners in MapReduce and Pregel) and are then sent over the network to the node containing the neighbor vertex. An exception is, as mentioned, Stratosphere's broadcasting strategy.

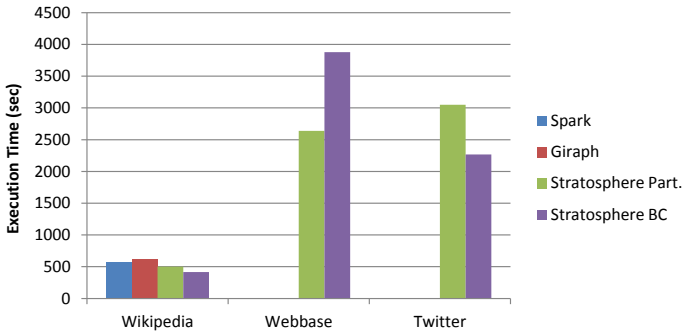

Figure 4.7.: Total execution times for the PageRank algorithm.

Figure 4.7 shows the results for the PageRank algorithm on the three systems on different data sets. As expected, the runtime of the algorithm is similar in Spark, Giraph, and Stratosphere for the small Wikipedia data set. We were unable to use Spark and Giraph with the large data sets, because the number of messages created exceeds the heap size on each node. At the time of conducting the experiments, Spark and Giraph lacked the feature to spill messages in the presence of memory pressure. For the large Webbase graph, we see that Stratosphere's broadcasting strategy degrades. Due to a limitation in the hash join algorithm, the hash table containing the broadcasted rank vector is currently built by a single thread on each machine, introducing a critical serial code path.

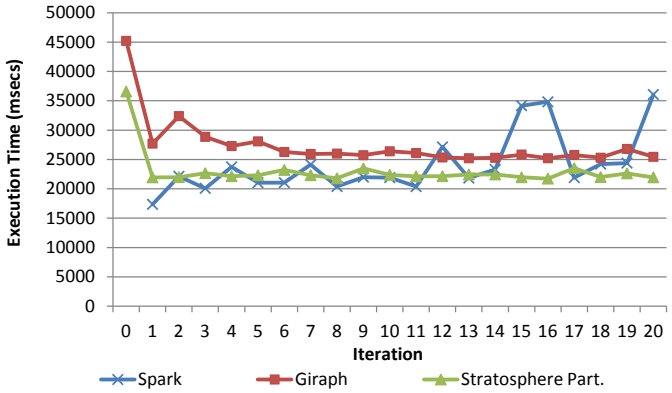

Figure 4.8.: Execution times of the individual iterations for the PageRank algorithm on the Wikipedia data set.

It is interesting to examine the time each iteration takes in the different systems. Figure 4.8 breaks down the execution time for the PageRank algorithm on the Wikipedia data set. For Stratosphere, we examine the partitioning strategy, because it performs the same work as the two other systems. The iteration times are rather constant in Stratosphere and Giraph. The first iteration is longer, because it includes for Stratosphere the execution of the constant data path, and for Giraph the partitioning and setup of the vertex states. The iteration times in Spark vary quite heavily due to garbage collection problems[8]. Spark uses new objects for all messages, creating a substantial garbage collection overhead. In contrast, Stratosphere's PACT runtime stores records in serialized form to reduce memory consumption and object allocation overhead. The runtime algorithms are optimized for operation on serialized data. The Giraph implementation is also hand tuned to avoid creating objects where possible, removing pressure from the garbage collector.

4.5.2. Incremental Iterations

While we expected the systems to perform similarly for bulk iterative algorithms, we expect a big performance difference for incrementally iterative algorithms. The magnitude of the difference should depend on the sparsity of the computational dependencies: the sparser the dependencies are, the more speedup we expect through incremental iterations, as compared to bulk iterations.

To verify our expectations, we ran the different versions of the Connected Components algorithm on the three systems. For directed graphs, we interpreted the links as undirected,

[8]The iteration times for Spark were averaged from multiple runs of the algorithm. Inside a single run, the variance is even greater.

thus finding weakly Connected Components. We ran the bulk iterative algorithm from Section 4.2 on Spark and Stratosphere and an the incrementally iterative version on Stratosphere and Giraph.

We have two Stratosphere implementations for the incrementally iterative algorithm, resembling the examples in Section 4.4: one uses a Match operator for the update function, the other one a CoGroup. Recall that the Match variant corresponds to a microstep iteration and takes each element from the working set individually, potentially updating the partial solution and producing new working set elements. The CoGroup variant represents a batch incremental iteration. It groups all working set elements relevant to one entry of the partial solution, such that each such entry is updated only once. The grouping of the working set does, however, incur some additional cost. We consequently expect the microstep variant to be faster on the sparse graphs, where the number of redundant candidates in the working set is smaller. For denser graphs, we expect the batch incremental algorithm to eventually be the faster variant, as the cost of grouping the working set elements is amortized by the saved cost due to fewer accesses to the partial solution.

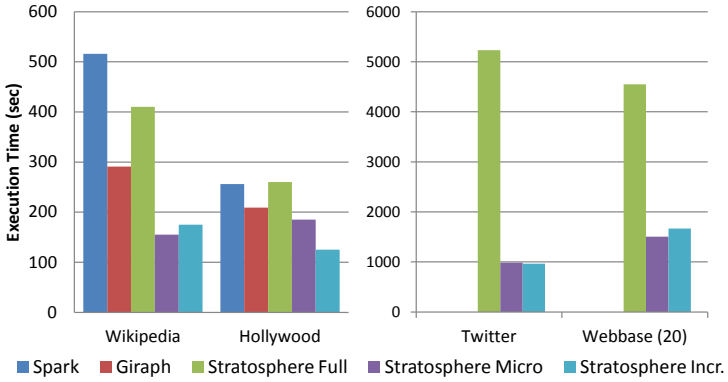

Figure 4.9.: Total execution times for the Connected Components algorithm.

Figure 4.9 shows the execution times for the Connected Components algorithm. We were again unable to run the algorithm on Giraph and Spark for the Twitter and Webbase data set, as the systems ran out of memory.

For the Wikipedia graph, the algorithms took 14 iterations to converge. The speedup of incremental iterations compared to bulk iterations in Stratosphere is roughly a factor of 2. Giraph also clearly outperforms the bulk iterations in Spark and Stratosphere.

For the Hollywood data set, the gain though incremental iterations is less, because the vertex connections, and consequently the computational dependencies, are more dense. In contrast to the runtimes on the Wikipedia graph, the batch incremental algorithm is

here roughly 30% faster than the microstep algorithm. As discussed before, this is due to the graph's density, which results for the microstep algorithm in many more accesses to the partial solution, while the batch incremental algorithm only groups a larger set.

For the Twitter data set, where the algorithms takes 14 iterations to converge, we see a tremendous gain through incremental iterations. They are in Stratosphere roughly 5.3 times faster than the bulk iterations. The performance gain is high, because a large subset of the vertices finds its final component ID within the first four iterations. The remaining 10 iterations change less than 5% of the elements in the partial solution.

For the Webbase graph, the figure shows the algorithm runtimes for the first 20 iterations. Because the largest component in the graph has a huge diameter, the algorithms require 744 iterations to fully converge. Already in the first 20 iterations, in which the majority of the actual component ID changes happen, the incrementally iterative algorithms are 3 times as fast as their bulk iterative counterparts. For the entire 744 iterations, the incremental algorithms take 37 minutes, which is only a few minutes longer than for the first 20 iterations. In contrast, the extrapolated runtime for the bulk iterative algorithm to converge on the whole graph is roughly 47 hours. The comparison yields a speedup factor of 75. By the time the incremental algorithms converge on the entire graph, the bulk iterative algorithm has just finished its 10th iteration. Figure 4.10 shows the execution time and the number of working set elements per iteration for the Webbase graph. The execution time does not drop below 1 second, which is a lower bound currently imposed by the distributed synchronization which works with heartbeat messages that have a certain delay.

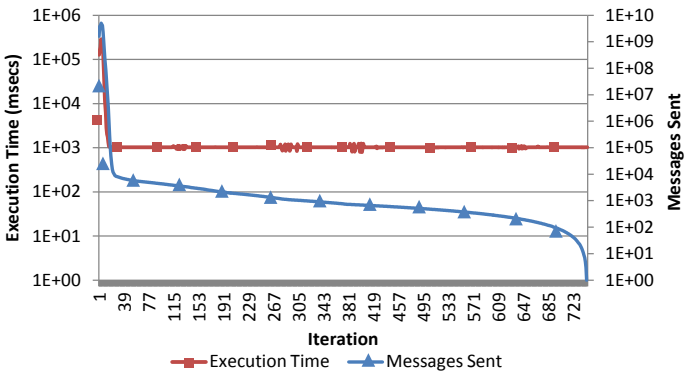

Figure 4.10.: Execution time and working set elements for Connected Components in the Webbase graph on Stratosphere.

In Figure 4.11 we examine the iteration runtimes for the various algorithms processing the Wikipedia graph. To assess the benefit from sharing the partial solution state across iterations, we added an additional iteration variant that simulates an incremental iteration

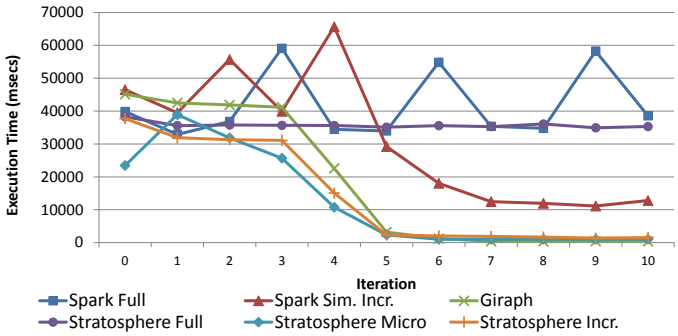

Figure 4.11.: Execution times of the individual iterations for the Connected Components algorithm.

on top of Spark's non-incremental runtime. This simulated variant adds a Boolean flag to the each entry in the partial solution. The flag indicates whether the component id decreased in the last iteration. If it did not, then no messages are sent to the neighbors, but only to the vertex itself, in order to carry the current component id mapping to the next iteration. This simulated variant exploits computational dependencies, but needs to explicitly copy unchanged state.

We see that the bulk iterations in Stratosphere and Spark have a constant iteration time (modulo the variance introduced by garbage collection in Spark). In contrast, the incrementally iterative algorithms in Stratosphere and Giraph converge towards a very low iteration time after 4 iterations. They exploit the sparse computational dependencies and do not touch the already converged parts of the partial solution in later iterations. Giraph's iteration times are the lowest after 5 iterations, because it currently uses a more efficient way to synchronize iterations than Stratosphere. The simulated incremental algorithm on Spark ("Spark Sim. Incr.") decreases in iteration time as well, but sustains at a much higher level, because of the cost for copying the unchanged records in the partial solution.

We finally look at how the number of created candidate component identifiers (messages, respectively working set elements), influence the runtime of bulk iterative and incrementally iterative algorithms. Figure 4.12 shows for the Wikipedia data set the time per iteration and the number of messages (working set elements). By the overlapping curves, we see that for the bulk iterative algorithm and the batch incremental algorithm (CoGroup), the runtime is almost a linear function of number of candidate component identifiers, in both cases with the same slope. For the microstep iterative algorithm (Match), a similar relationship exists but with a much lower slope. Because in the microstep variant, the update function is much cheaper than for the batch incremental variant, it can process in equal time much larger working sets containing many more

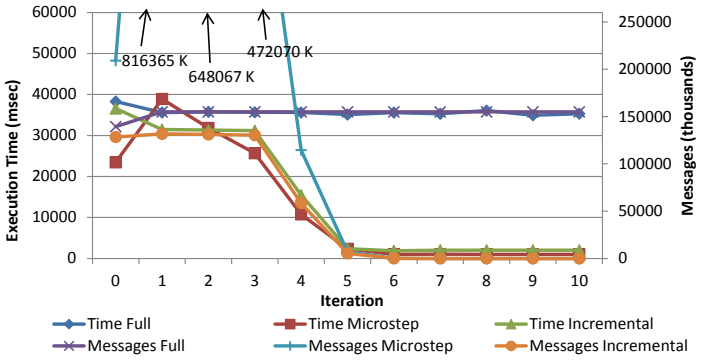

Figure 4.12.: Correlation between the execution time and the exchanged messages for the Wikipedia graph on Stratosphere.

redundant candidate component identifiers.

We have seen that Stratosphere, as a general iterative data flow system, can outperform specialized analysis systems for bulk iterative algorithms. At the same time, the incremental iterations speed up algorithms that exploit sparse computational dependencies. They touch in each iteration only the elements in the partial solution that potentially require modification, thereby greatly outperforming full iterations.

4.6. Related Work

In this section, we first relate our work to recursive query evaluation, which is the most prominent alternative to iterative queries. Subsequently, we relate our work to other systems for parallel data analysis.

4.6.1. Relationship to Recursive Queries

In programming languages, the equally expressive alternative to iteration is recursion. There has been a wealth of research addressing recursion in the context of Datalog and the relational model [21]. The query processing line of that research focused largely on top-down versus bottom-up evaluation techniques and suitable optimizations, like predicate pushdown [24]. The incremental iteration techniques presented here bear similarity to the *semi-naïve* evaluation technique. More recently, Afrati et al. [5] and Bu et al. [33] discuss how to compile recursive Datalog queries to data flows. Both evaluate the recursion bottom up. The approach of [5] uses stateful operators for the datalog rules, while [33] discusses a more comprehensive approach including optimization of the

created data flow. Our incremental iterations offer a similarly expressive abstraction. All examples from [5, 33] are expressible as incremental iterations. In fact, following the arguments from Afanasiev et al. [4] and Bu et al. [33], it becomes clear that one can express stratified (or XY-stratified) Datalog programs as incremental iterations with supersteps. For programs without negation and aggregation, the supersteps can be omitted. Since recursion in relational databases has much stricter conditions than in Datalog, the proposed incremental evaluation is applicable to relational DBMSs. Early work in that direction includes *delta iterations* [60]. We build on a similar abstraction, but additionally focus on aspects relevant to parallel execution.

Both Datalog queries (and recursive relational queries) may only add rule derivations (respectively tuples) to the partial solution. There is no synchronization barrier defined, so this model naturally lends itself to asynchronous execution. For algorithms that require supersteps, Bu et al. [33] express synchronization barriers by introducing a temporal variable that tracks the supersteps and using aggregation predicates to access the latest superstep's state. Because this leads easily to a complicated set of rules, they use this technique to specify domain specific programming models rather than to specify queries directly. A distinguishing aspect in our approach is the fact that incremental iterations can actually update the state of the partial solution, allowing the iterations to compute non-inflationary fixpoints (cf. [4]). This obliterates the need to filter the state for the latest version and naturally leads to a semi-naïve flavor of evaluation, resulting in simple queries.

In contrast to the work of [33], we do not distinguish between global- and local model training, but leave the decision to replicate or broadcast the model to the data flow optimizer. Instead, we distinguish between iterations that recompute the partial solution or incrementally evolve it, because that has a direct impact on how the programmer has to specify the algorithm.

4.6.2. Other Iterative Analytical Systems

HaLoop [34] and Twister [46] are MapReduce [44] extensions for iterative queries. They provide looping constructs and support caching of loop-invariant data for the special case of a MapReduce data flow. In contrast, our technique addresses more general data flows. The optimization techniques presented in Section 4.3.3 subsume their special-case optimization. The Spark data flow system [112] supports resilient distributed data sets (RDD). With loop constructs in its Scala front-end, it supports full iterations. Neither of the systems addresses incremental iterations or considers data flow optimization.

Pregel [78] is a graph processing adoption of bulk synchronous parallel processing [101]. Programs directly model a graph, where vertices hold state and send messages to other

vertices along the edges. By receiving messages, vertices update their state. Pregel is able to exploit sparse computational dependencies by having a mutable state and the choice whether to update it and propagate changes in an iteration. The incremental iterations proposed in this chapter permit equally efficient programs, with respect to the number of times the state of a vertex is inspected, and the number of messages (in our case records) sent. It is straightforward to implement Pregel on top of Stratosphere's iterative abstraction: the partial solution holds the state of the vertices, the workset holds the messages. The step function Δ updates the vertex state from the messages in the working set and creates new update messages. Since Δ may be a complex parallel data flow in itself, it allows for writing even more complex algorithms easier. The adaptive version of PageRank [69] for example, can be expressed as an incremental iteration, while it is hard to express it on top of Pregel. The reason for that is that Pregel combines vertex activation with messaging, while incremental iterations support programs that separate these aspects. In addition, incremental iterations have well defined conditions when they allow for (asynchronous) microstep execution, and offer a data flow abstraction, thus integrating naturally with parallel data flow systems.

GraphLab [77] is a framework for parallel machine learning, where programs model a graph expressing the computational dependencies. The abstraction is distributed shared memory: nodes exchange data by reading neighboring nodes' state. GraphLab has configurable consistency levels and update schedulers, making it a powerful, but low level abstraction. Most GraphLab algorithms can be implemented as incremental iterations by using the working set to hold the IDs of scheduled vertices and accessing neighboring states though a join with the partial solution.

ScalOps [104] is a domain specific language for big data analytics and machine learning. It provides a simple language interface with integrated support for iterations. ScalOps programs are compiled to optimized Hyracks [31] data flows. At the time of writing this chapter, the system aspects of executing ScalOps on top of Hyracks have not been published.

Ciel [81] is a data flow execution engine that supports very fine grained task dependencies and runtime task scheduling suitable for iterations and recursions. Ciel does not offer a direct data flow abstraction to iterations or recursions; instead, the task dependencies are generated from queries in a specialized domain specific language. Ciel cannot share state across iterations, but the runtime task scheduling allows to somewhat mimic that, although at the cost of creating very many tasks.

4.7. Conclusions and Outlook

We have shown a general approach to iterations in data flows. For algorithms with sparse computational dependencies, we argued that their exploitation is crucial, as it leads to a huge speedup. To exploit computational dependencies in data flow systems, we suggested an abstraction for incremental iterations. Incremental iterations are general and suit many iterative algorithms. We presented optimization and execution strategies that allow a data flow system to execute the incremental iterations efficiently, touching only the state that needs to be modified to arrive at the iteration's result.

Incremental iterations subsume several specialized analysis models, such as the Pregel model, both in expressibility and efficiency. In fact, all algorithms that are naturally expressible in Pregel's programming model, can be expressed as incremental iterations without a performance hit. Furthermore, incremental iterations allow for (asynchronous) microstep execution under well defined conditions.

In the case study with our prototypical implementation of incremental iterations in the Stratosphere system, we have shown that an implementation of incremental iterations allows a parallel data flow system to compete with specialized systems for algorithms that are a sweet spot of those systems. At the same time, Stratosphere maintains a general data flow abstraction, thus reducing the number of systems required for large scale analytics.

In future work we want to explore how far compiler techniques can automatically transform algorithms stated as fully iterative algorithms into incremental algorithms. A powerful method to exploit algorithmic properties for fault tolerance has been demonstrated with Stratosphere in [93]. As future work, we plan to investigate these optimistic techniques can work in symbiosis with traditional techniques (checkpointing).

5. A Functional Language Frontend

The previous chapters described a principled programming model for data parallel programs based on second-order functions. They discussed the basic programming abstraction, the data model, the set of primitives, and how they are amendable to automatic optimization. It is possible to expose this programming abstraction directly in an API; in fact, the PACT programming abstraction in early versions of the Stratosphere system embeds this model into a Java library in a straightforward fashion.

However, we have seen that it is very challenging to write concise and reusable programs against such an API. There is a rather tight coupling between the algorithm's logic and the contextual aspects of its implementation. Programmers must be very disciplined to create components that are both efficient and reusable; in some cases the two goals are contradicting. Programs tend to become cluttered with what we call *"formal noise"*: code that is necessary for the execution but has little to do with the algorithm's intent. Examples of formal noise are code that wraps variables and data structures into special system supported data types, or mapping variables to the framework's data model (records or key/value pairs). Formal noise severely impedes readability and conciseness. Finally, we observed that a minimal set of primitives in the API is helpful for understanding the programming abstraction, but for efficient execution, the runtime that executes the program needs a larger set of more specialized primitives.

Some of these problems can clearly be mitigated by a good design of the API, which defines how the primitives are instantiated and composed into programs. By choosing an expressive language and designing the API appropriately, one delegates some of the problems to the language compiler, using for example mechanisms like operator overloading. Other problems require additional techniques, like schema generation and code generation.

This chapter discusses techniques to solve the above mentioned problems. We discuss the design of a functional language frontend called *Sky*, realized as a domain-specific language in Scala. Sky extends the PACT abstraction and consists of a library and a plug-in to the Scala compiler. The compiler plug-in realizes the mapping between an algebraic subset of Scala's data model and the record-oriented data model in PACT, and generates the glue code required for the distributed runtime to handle the user data types. This allows programs to work with Scala's regular data types, thus significantly reducing the formal noise of the programs. At the same time, the PACT compiler

and optimizer still get a record data model view of the programs and can optimize them. The Sky library defines an API based on the PACT primitives, using functional constructs to simplify their expression. The functional constructs decouple the function implementation from contextual aspects, making it concise and reusable. To ensure efficient execution, the library infers the program-wide schema information, giving the optimizer maximum latitude in exploiting the space of possible execution plans. The result is a language frontend, where programs are concise like high-level queries, but retain the full expressiveness and flexibility of general-purpose data parallel programs.

5.1. Problem Statement

The concepts and techniques described in the previous chapters have been embedded in a Java API as the *PACT Programming Model* for the Stratosphere system. Programs written against the PACT API can be optimized extensively and have been shown to yield high performance parallel execution [8, 22, 48, 63]. However, these programs are at the same time very verbose and tend to be hard to write. An algorithm has to be broken up into several classes (typically one per function) to facilitate code-, data-, and parameter shipping as required for distributed execution. Furthermore, the programmer has to map the function's individual input variables and data structures to record fields in a consistent manner across all user-defined functions: If a function expects an input variable at a certain position in the record, the predecessor function must make sure to write the variable to that specific position.

This results in a disconnect between the serial and parallel formulation of an algorithm. Consider a sample program that computes a simple word histogram (the number of occurrences per distinct word) over a textual data source. The given task is very simple: The program first splits its input lines into lower case words, then groups the result by word and counts the elements in each group. The program is shown in Listing 5.1 using a functional pseudo code syntax resembling the Scala language. The sample code defines the program using second-order functions, which makes it very comparable to the conceptual structure of a corresponding PACT program. However, the actual PACT program implementing the same functionality (Listing 5.2) requires 44 non-empty lines of code. It also appears fragmented and code lines that implement actual parts of the algorithm are interrupted by *glue code* that is required for the interaction between the user code and the parallel system. We refer to the sum of these additional code lines and programmatic constructs as the *formal noise*.

The effect is even more apparent in more complex programs. Listings B.3 and B.4 (in the Appendix) give the implementation of a relational analysis program, in functional code, respectively the PACT Java API. In addition to the type of formal noise in the word

```
val input = Datasource(...)
val words = input flatMap { _ toLowerCase split("\\W+") } map { w => (w,1) }
val counts = words groupBy { _._1 } aggregate ( sum, _._2 )
```

Listing 5.1: The actual work performed by the WordCount program, in functional pseudo-code. The expression $\backslash\backslash W+$ denotes the regular expression pattern for whitespace characters, defining to split words at spaces, tabulators, and line breaks. For an overview of the Scala syntax, please refer to Appendix A.

```
public class TokenizeLine extends MapUDF {

  public void map(PactRecord record, Collector<PactRecord> collector) {
    PactString pLine = record.getField(0, PactString.class);
    String line = pLine.getValue();

    for (String word : line.split(" ")) {
      PactString s = new PactString(word);
      PactInteger one = new PactInteger(1);
      collector.collect(new PactRecord(s, one));
    }
  }
}

@Combinable
public class CountWords extends ReduceUDF {

  public void reduce(Iterator<PactRecord> r, Collector<PactRecord> c) {
    PactRecord element = null;
    int sum = 0;
    while (r.hasNext()) {
      element = r.next();
      PactInteger i = element.getField(1, PactInteger.class);
      sum += i.getValue();
    }
    element.setField(1, new PactInteger(sum));
    c.collect(element);
  }
}

public class WordCount implements PlanAssembler {

  public Plan getPlan(String... args) {
    FileSource source = new FileSource(TextInput.class, "file://...");

    MapContract mapper = MapContract.builder(TokenizeLine.class)
      .input(source).build();

    ReduceContract reducer = ReduceContract.builder(CountWords.class)
      .key(PactString.class, 0)
      .input(mapper).build();

    FileSink out = new FileSink(RecordOutput.class, output);
    out.setInput(reducer);
    RecordOutput.configureRecordFormat(out)
      .field(PactString.class, 0)
      .field(PactInteger.class, 1);

    return new Plan(out);
  }
}
```

Listing 5.2: The word histogram program expressed through the Java PACT API.

histogram program, this example has an explicit mapping of each UDF's tuple schema to the physical fields of the records exchanged by the UDFs. Programmers effectively need

to design the physical runtime schema when implementing the programs and need tailor functions specific to that physical schema. We refer to this characteristic as *physical schema dependence* of a program implementation.

A part of the verbosity and clutter that constitutes the formal noise is due to the fundamental mismatch between Java's object-oriented paradigm and the functional nature of the underlying PACT programming model. This has been recognized by the authors of *Flume Java* [41], a fluent Java API for MapReduce programs. A functional language, or a language supporting certain functional programming concepts (lambdas/function literals, functions as data types), alleviates most of those aspects. Other problems, like the necessity for glue code and the physical schema dependence are more fundamental. In particular, the later significantly impedes the creation of reusable components and the fluidity of the development of programs.

Similar observations have let to a plethora of higher-level languages created on top of data-parallel programming systems. Prominent examples are Hive [99], Pig [85], JAQL [26], Meteor [62], and AQL [25]. All of these languages build on the relational algebra (or extensions thereof) and give up the generality of generic data-parallel programming model: Hive is essentially a subset of SQL, Pig adds sets and maps as opaque data types, JAQL, Meteor, and AQL implement a version of the relational model with nesting, thereby being comparable to XQuery, or a subset thereof[1]. For programs that require operations beyond the language's closed set of operators, the programmer must resort back to the lower-level parallel programming API (MapReduce, PACT, ...), or at least use UDFs to realize functionality beyond the built-in operators.

The types of UDFs supported by the above mentioned languages range from scalar functions or custom aggregation functions to generic parallel functions that implement complex behavior. The latter are built using the underlying system's parallel programming building blocks (Map and Reduce in the case of Hive, Pig, JAQL, and the PACT primitives in the case of Meteor). Since the parallel UDFs contain imperative code in a second-order functional corset, there is a grave impedance mismatch between the higher-level language and its UDFs. This becomes obvious when examining the way that the user-defined code is embedded into these higher level languages and the way user code accesses data produced by a query. Listings 5.3 and 5.4 show the syntax for embedding a MapReduce program into a Pig query, respectively Hive query. This problem is, however, the same across all higher level languages: Mixing and matching higher-level languages and user-defined functions causes fragmentation, paradigm switches within a program, and introduces glue code to bridge between the data model of the higher-level language and that of the programming language.

[1] In fact, Hive positions itself as a SQL engine, whereas the design of AQL and JAQL is based on XQuery without some of the confusing semantics introduced by XQuery, such as document order.

```
1  A = load 'WordcountInput.txt'
2  B = MAPREDUCE wordcount.jar store A into 'inputDir'
3                               load 'outputDir' as (word:chararray, count: int)
4                               'org.myorg.WordCount inputDir outputDir';
5  C = sort B by count;
```

Listing 5.3: Example for embedding a user-defined code into the Pig language [85].

```
1  FROM (
2    FROM pv_users
3    MAP pv_users.userid, pv_users.date
4    USING 'map_script'
5    AS dt, uid
6   CLUSTER BY dt) map_output
7  INSERT OVERWRITE TABLE pv_users_reduced
8    REDUCE map_output.dt, map_output.uid
9    USING 'reduce_script'
10   AS date, count;
```

Listing 5.4: Example for embedding a user-defined code into the Hive query language [99].

An exception are programming languages designed as an extension of an algebraic language for the specific purpose of implementing UDFs for that language. An example is SQL with the PL/SQL extension, which results in smoother mixing and matching of higher-level language operations with user-defined functions. The drawback here is that the UDF language is a custom language, which naturally restricts the usage of libraries and other components from other general-purpose programming languages.

As a consequence of this discussion, we suggest that a language frontend that supports concise parallel programs and seamlessly integrates with a higher level language must meet the following requirements:

- The composition of data analysis programs or queries should follow the same paradigm as the underlying parallel programming model to avoid mismatches between the programming paradigms.
- The parallel programs should state their essential functionality and contain little formal noise. This noise is measured as extra statements or lines of code that have nothing to do with the algorithm's core intent.
- There must be an abstraction between the logical data layout in the user-defined functions and the physical layout in the runtime. That way, functions can be implemented independently of the context set by their predecessor functions and the context expected by their successor functions.
- Built-in operations and user-defined functions should be treated the same way. When built-in operations appear as a special case of user-code (with specialized execution strategies and rich semantics), they mix seamlessly.

- The previous point implies that the higher level language and the user-defined functions should share a common data model.

In this chapter, we will outline an approach for a language frontend that satisfies the above mentioned requirements. We show how it leads in fact to concise and fluent programs that retain the generality of a data parallel programming API, while being optimizable and compact. The language uses the same mechanism to compose algebraic operations with user-defined functions, allowing to mix and match them seamlessly. The remainder of this section dissects the PACT programs and analyzes the causes of the above-mentioned problems and the following sections outline the design and implementation of the language frontend.

5.2. Analyzing the Sources of Programming Overheads

This section examines the sources of the above-mentioned problems by dissecting a PACT program and analyzing the compilation process. We first give a discussion of the sources of *formal noise*, followed by a discussion about the mechanisms causing the *physical data model dependence*. For each problem source, we discuss techniques to alleviate the problem.

5.2.1. Sources of Formal Noise

Several factors contribute to the formal noise present in PACT programs. They occur in each of the individual parts of a PACT program: Data type definitions, function definition, function parametrization, and data flow composition. In the following, we will examine formal noise in each of the parts and make suggestions for how to eliminate it.

PACT programs have special requirements for the ***data types*** that are used. Data types are moved between machines or memory pools frequently in the course of the distributed execution. This requires the types to support highly efficient serialization and deserialization, with low computational cost and a compact serialized representation. The built-in serialization mechanism in Java does not fulfill these requirements, forcing PACT programs to use special data types that implement their own serialization and deserialization routines. The PACT API offers such wrapping data types for the basic data types typically found in languages like Java, C# or the C/C++ language family, plus template classes for the implementation of strongly typed list, map, and set collection types. Using rich wrapper types instead of the languages built-in primitives requires as the bare minimum the extra instantiation of these types, or the setting of their values (cf. Listing 5.2, lines 8 and 9). In addition, it frequently comprises the definition of these types.

While the record data model allows multiple primitive- and collection types to be passed between functions, it is often desirable to create composite types as encapsulation units. These composite types inherit from the same abstract data type as all primitive types and must supply extra functions for serialization and deserialization. Data types acting as keys in functions like Match or Reduce must additionally support efficient and effective ordered comparisons and hash code computation. Efficient runtime algorithms often require even further utility functions, for example the creation of normalized key prefixes [57]. Listing 5.5 shows the code for a two dimensional point data type, implemented to be usable as a key and supporting efficient sorting operations. It is obvious that this definition comes with an enormous amount of formal noise, compared to the simple encapsulation of two integers that the data type does.

```
1  public class Point implements NormalizableKey {
2
3    public int x, y;
4
5    public void write(DataOutput out) throws IOException {
6      out.writeInt(x);
7      out.writeInt(y);
8    }
9
10   public void read(DataInput in) throws IOException {
11     x = in.readInt();
12     y = in.readInt();
13   }
14
15   public int compareTo(Point o) {
16     return x - o.x == 0 ? y - o.y : x - o.x;
17   }
18
19   public int hashCode() {
20     return x ^ y;
21   }
22
23   public boolean equals(Object obj) {
24     if (obj.getClass() == Point.class) {
25       Point o = (Point) obj;
26       return o.x == x & o.y == y;
27     } else
28       return false;
29   }
30
31   public int getMaxNormalizedKeyLen() {
32     return 8;
33   }
34
35   public void writeNormalizedKey(byte[] target, int offset, int len) {
36     target[offset++] = (byte) ((x >>> 24) - Byte.MIN_VALUE);
37     for (int i = 2; len > 1 && i >= 0; i--, len--) {
38       target[offset++] = (byte) ((x >>> (i << 3)) & 0xff);
39     }
40     if (len > 1) {
41       target[offset++] = (byte) ((y >>> 24) - Byte.MIN_VALUE);
42       for (int i = 2; len > 2 && i >= 0; i--, len--) {
43         target[offset++] = (byte) ((y >>> (i << 3)) & 0xff);
44       }
45     }
46   }
47 }
```

Listing 5.5: A data type for a two dimensional point, as implemented for the Java PACT API.

The formal noise introduced by custom data types is fortunately relatively easy to alleviate. It is possible to generate all serialization code, as well as code for further utility functions. For primitive types and arrays of primitive types, this code readily exists. Composite types, such as classes (possibly nested) recursively invoke the serialization functions of their individual components. The same is possible for the utility functions for key data types, such as hashing and the creation of normalized key byte strings. A richer type system in the programming language or API library helps to reduce the need for composite types in the first place.

The ***definition of a user-function*** in the Java PACT API requires implementing a dedicated class that inherits the specific abstract function signature. When shipping and loading external code into a program (such as a user-defined function into the distributed runtime), the entry point for the code must be given as a function pointer. The only way to provide a function pointer in Java is implicitly through the method look-up table of classes, requiring programmers to implement a dedicated class for a user-defined function. The syntax overhead induced by this mechanism is solely attributed to the particular programming language. It disappears in programming languages that support compact references to functions, such as raw function pointers, or functions as data types.

Additional formal noise in the implementation of user-defined functions is introduced by the mapping between the API's data model and the local variables. The PACT API features a data model of schema free records (tuples) with an arbitrary number of fields. The first part of a UDF typically assigns the contents of the relevant record fields to local variables (possibly unwrapping the actual types from the rich wrapping data types), while the last part frequently builds the result record(s) from the modified variables. This is visible for example in Listing 5.2 in lines 4, 23, and 26, and in the example Listing B.4, lines 15-17.

The necessity for such a mapping in the UDF stems from a mismatch between the programming languages type system and the data model of the parallel programming API. The mismatch can theoretically be mitigated by using an unrestricted data model with arbitrary data types. The key/value pair model implemented by MapReduce does essentially that: The value is an arbitrary data type, and the key is an explicit field for grouping. However, as Section 3.2.1 showed, such an overly simplified data model severely limits the potential for optimization. There is consequently a conflict between the need for a somewhat constrained data model for the optimization, and a very free data model for a concise function implementation. A way to overcome that problem is to define a very rich data model that contains enough constructs to have a very generic "feel"[2], and let the system map it to the constrained data model before the optimization.

[2]There are no hard measures for how suitable a specific data model is. We argue here that it is more suitable, if it leads to fewer user-defined data types and less extra code for wrapping function variables with special system-supported types. This typically correlates with how many constructs of the

This approach would combine the benefits of a very rich data model with the potential to optimize the programs. However, it leaves several non-obvious questions, like the design of the rich data model, and, most importantly, how to specify subsets of fields to be used as keys.

The ***function parametrization*** introduces extra syntactic overhead. Functions may only be defined as methods of "stateless" classes: The class must declare a default constructor and, if they are nested classes, cannot access instance state provided by their lexical environment. Any parameters must be explicitly read from a configuration object that is passed to the class after the instantiation. This effect can be observed in the program in Listing B.4: Parameters are read from the configuration (lines 4-11) into which they are written in lines 68-69.

The described mechanism for function parametrization ensures that the size of the shipped code and parameters is minimal: Built-in serialization mechanisms for objects (Java) or advanced mechanisms to serialize functions including the variables in the function closures (Scala) often serialize a lot more than actually required (such as transient fields that were not marked as such). In fact, the size of a serialized parametrized function can easily be in the order of many megabytes or more, depending on the scope of the function's closure. Given that using built-in serialization of stateful objects or closures eliminates all of the overhead code required for parametrization, these mechanisms are a reasonable choice, in combination with a "safety net" that catch cases where the size of a closure has grown beyond what can be efficiently handled.

The final factor contributing to the formal noise is the ***data flow composition***. It defines the data sources and sinks, and describes which function results are passed as inputs to other functions. Defining the data flow composition separately from the function implementation causes program fragmentation. In the word histogram example in Listing 5.2, the first 29 lines state the function definition, whereas the following 21 lines define the data flow. A more fluent design of the PACT API could allow for a more concise data flow definition, but does not solve the fragmentation problem: Even the simplest functions that are not intended as reusable units, are not defined where they are used. The general limitation of imperative languages that do not support functions as literals and a paradigm mismatch with the functional nature of the PACT programming API. Using a language that supports function literals solves this fragmentation.

Minor formal noise is often introduced by assigning an explicit name for the second-order functions in the data flow, in order to identify it during monitoring or debugging. Outside of specialized APIs, such naming happens implicitly by the choice of variable names. Reusing the variable names rather than explicit names eliminates additional overhead code from the data flow composition.

programming languages type system the data model is able to represent directly.

5.2.2. Sources of Physical Data Model Dependence

Physical data model dependence refers to the fact that a program implementation, specifically the implementation of its user-defined functions, must be conscious of the physical data layout used by the runtime system during the execution of the algorithm. In MapReduce for example, the physical data model is key/value pairs and a function is specialized to its specific key and value data types. It is cumbersome and often impossible to write generic reusable units against that abstraction. In non-trivial programs, a large portion of the functions require a dedicated extra data type. PACT's record data model was designed to allow for optimization and reuse of functional units[3]. Its record data model can be thought of as dynamically typed variable-length tuples. However, a function is still dependent on the physical model as it is responsible for knowing the data type and physical position of each field it accesses.

Listing B.4 illustrates the problem of physical data model dependence prominently. The implementer must design the physical execution schema during the implementation. As a bare minimum, this requires to set up the field positions consistently across all functions, meaning that all functions must agree to which position in the tuple what variable goes. While logical addressing with names is theoretically possible, it has shown to have a very high overhead at runtime to resolve these logical look-ups. For optimizations that change the structure of the program (reordering operations), the functions have to be implemented with a physical schema that works in multiple contexts [63]. For that, a function must not only coordinate its field positions with its predecessors, but also with a large portion (possibly all) of the other functions as well, in order to avoid collisions on the physical fields. This requirement makes the creation or reusable components that still allow for automatic optimization virtually impossible in practice.

The root cause for the problem of physical data model dependence lies naturally in the fact that there is no distinction between logical and physical access to the data model. In typical higher-level languages with a fixed data model and a closed set of operators, all accesses to the data are logical. The compiler has a holistic view of all accesses and logical schemata and devises a physical layout for the data that is passed between operators/functions. Figure 5.1 shows a simplified version of a compile chain in a typical database management system. After the queries are parsed and checked (for valid semantics), they are optimized and finally translated into executable code. Before the code generation happens, the optimizer has a holistic view of the entire query and devises a physical schemata for each operator (or pipeline) that is specific to its context.

In contrast to that, the compile chain of a typical UDF data flow system (such as Stratosphere [22, 63], Cascading [37] for Hadoop [106], or Dryad [64, 111]) is given in

[3] Optimizability of a program and re-usability of functional units is often closely related, as the re-ordering of the functional units implies that they can be used in various contexts or positions in the program.

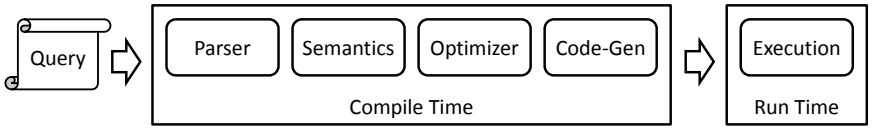

Figure 5.1.: Typical compile chain of a Relational Database Management System.

Figure 5.2. The most obvious distinction is that there are two different compilers involved: The compiler of the UDF programming language and the data flow compiler. The UDF language compiler translates each UDF individually into executable code. The successive data flow compiler optimizes the data flow with a holistic view of the program and translates it into an executable parallel schedule. The code generation happens here before the data flow optimization phase and can consequently not exploit the knowledge from a holistic view of the data flow program. Systems that apply heavy optimizations to the data flow are more affected by this than systems that perform only minor optimizations.

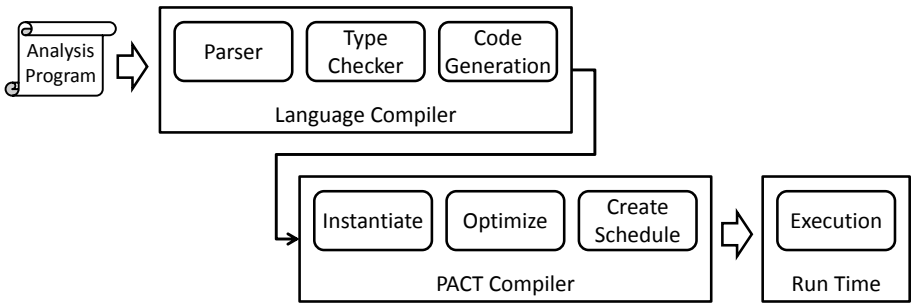

Figure 5.2.: Compile chain of for PACT programs in Stratosphere.

Moving the entire UDF code generation phase after the optimizer is infeasible for several practical reasons (e. g., libraries, incremental compilers), but it is also not necessary. It suffices to move the generation of certain the parts of the code after the optimizer; specifically the parts that require information about the physical runtime data layout, such as field accesses and compositions of new records. In this approach, the data flow optimization can use holistic logical information to devise specialized physical data layouts and create or adjust the UDF code accordingly. There are several ways to effectively do that, ranging from parameterizable indirection layers to pre-compiling and rewriting parts of the executable function code. We will discuss these approaches in the later sections of this chapter.

5.3. Choosing a Host Language

The support for the above described features constitutes the minimal requirements for a host language. Further requirements originate from technical challenges to implement the compile chain and host the use cases, as well as from social factors that influence adoptability. From a technical perspective, a strong JVM support is highly desirable; it enables easier integration with the existing Stratosphere stack and allows seamless integration with the wealth of existing libraries available for that platform. In addition, our design requires a "hook" into the language compiler to integrate the mapping from the language's data model to the logical tuple model, together with the translation for field selector functions to numeric indexes, and the generation of serialization code. Fortunately, many modern compilers feature extensible designs and provide an officially supported method of integration. Our approach requires a compiler with a modular architecture where the hook can be inserted between the well-defined phases. In particular, our technique requires access to a valid abstract syntax tree (AST) of the program, after type-checking is complete, but before code optimizations have been performed.

There are various methods to hook into the compiler. Some compilers offer ways to statically register plug-ins, to be executed between specific phases. More advanced compilers have the ability to lift parts of the compile-time AST to runtime data structures, allowing a program to manipulate some portion of itself and trigger code generation at runtime on demand. Microsoft's implementation of .NET expression trees follows this technique and is used to power the Language Integrated Query (LINQ) feature. Some compilers expose their functionality through a compile-time meta-programming mechanism. Scala's macros [35] define a mechanism that transparently loads and executes special functions at compile time, passing them the AST of the functions at whose definition site the macro is registered. All these technique represent working hooks into the compile time to analyze the user-defined functions.

Besides these technical requirements, various social factors make some languages better candidates than others. While this quality is less tangible than concrete technical benefits, some reasonable assumptions can be made. It is always desirable to choose a language with an active user community in the target audience. Furthermore, the skill set of the target audience, most importantly the programming paradigms that they are familiar with, is a crucial factor. Choosing a pure functional language yields a low entry barrier for functional programmers. However, for the class of developers that are primarily accustomed to object-oriented programming, a pure functional language may be a serious obstacle. The relatively constrained syntax for the data flow composition with function literals is most likely easy to adopt. Still, eliminating object-oriented and imperative concepts from the implementation of the user-defined functions may be a stopper for many programmers. We argue that a hybrid language, supporting both functional and

imperative concepts, reaches a broader group of developers. It supports the requirements to host the programming API and gives the programmer a free choice of concepts and paradigms for the implementation of the user-defined functions. The downside is that hybrid languages are more difficult to analyze than pure functional languages, to a large extend due to their support for mutable state. For a proof-of-concept implementation of a parallel programming frontend, a pure functional language would be a good choice. An implementation that targets programmers beyond programming language- and system experts may be more successful in reaching its target audience when adopting a hybrid language. We argue that the target audience adoption is the more important factor to consider at this point, because as a social factor, is is inherently hard (if at all possible) to influence. The technical problem of a more complicated code analysis, on the other hand, can be mitigated by a more elaborate analysis algorithm [50].

We choose *Scala* as a host language for our language frontend, as it meets the above mentioned requirements very well. While the design and implementation of our language frontend uses a number of Scala's advanced features extensively (for example implicit parameters and conversions, pattern matching, mix-in-composition, and serialization of closures), the presented techniques are not specific to Scala and could be implemented in other expressive languages as well. An additional advantage of the Scala language is its compact syntax and powerful type inference. These mechanisms allow programs to appear extremely concise and reminiscent of higher-level languages (like SQL or Pig), rather than flexible data parallel programs. A possible drawback of Scala is that its flexibility sometimes overwhelms new programmers with choices. The ability to state most constructs both in a functional fashion as well as in an object-oriented way frequently confuses Scala newcomers. However, we believe that the advantages outweigh the drawbacks by far. We use the Scala language syntax for all further code examples. Appendix A gives a brief introduction to all Scala language constructs necessary to understand the examples used in this chapter.

A possible alternative to Scala would be the *F#* language from Microsoft's .NET framework. F# has powerful language features that are competitive to those of Scala. The choice towards Scala was ultimately based on its ability to seamlessly inter-operate with Java, which is important because both the Stratosphere framework is written in Java, and the ecosystem of analytics libraries in Java is, to the extend of our assessment, more mature.

5.4. The Sky Language Architecture

We build the Sky language following the hybrid functional-relational paradigm [55]. This paradigm composes queries as functional programs whose logical data flow is described via

monadic comprehension. It has been widely recognized that this technique leads to natural query language embeddings [98] and it is used in some form or another by frameworks like LINQ [89], Ferry [54], Data-parallel Haskell [40], and other data-parallel languages like Spark [112] or Scalding [92]. The different frameworks span a wide spectrum, from strictly algebraic frameworks (such as Ferry) to non-algebraic frameworks (for example Spark). The totally algebraic implementations expose strong semantics of a query and can consequently be optimized automatically, but they require programs to strictly adhere to their paradigm. While they are excellent candidates for optimizer engines, they are somewhat hard to program against and do not interact well with imperative code and libraries, thus being somewhat limited in their applicability as data-parallel programming APIs. The non algebraic frameworks use the idea of monadic comprehension for a fluent data flow composition, but feature a very loosely defined data model. They accept almost arbitrary first-order functions, which they treat as black boxes, and consequently expose little semantic information about the program. This approach makes them flexible and easy to program, but wastes much optimization potential. We present our design as a sweet spot between these two extremes: Our data model is highly flexible, but algebraic. Functions are treated as black boxes, except for their accesses to the data model's types. Combined with a shallow code analysis of the functions, our approach can perform heavy optimization of the data flow, while being as flexible and generic as the non-algebraic frameworks.

Sky's design follows from the requirements and basic approaches stated in the previous sections. The cornerstones of that design are as follows:

- A fluent and natural embedding of the program definition into a functional language, using Monad comprehensions.
- A very flexible, yet well-defined algebraic data model. The model is internally mapped to logical tuples with a mix of transparent and back box types.
- All field addressing happens though *field selector functions*. These functions appear as natural constructs of the programming language, but are analyzed and translated into logical field references. Because functional languages allow for the specification of selector functions through concise literals, they appear lightweight like logical field references in higher-level declarative languages.
- A compile chain that generates the glue code, such as code for serialization/deserialization. This glue code is required to use the data structures in user-defined functions with the distributed execution engine. The code generation is split between phases before and after the data flow optimizer, to allow for the generation of efficient physical record layouts.
- A mechanism that uses operator overloading and place holder functions to model built-in operators (e. g., joins or aggregations) as special cases of UDFs. This allows

the system to exploit the rich compile time semantics and the runtime efficiency of built-in operators in a seamless mix with user-defined functions.

While several of the individual features have been presented before, we believe that this work uses them in a unique combination to achieve a powerful programming abstraction. Some parts of this design, specifically the API design, the data model, and parts of the compile chain, have been published in a master thesis supervised by the author [61]. In this section, we summarize the results of that work and elaborate on further aspects in later sections.

```
1  class ConnectedComponents(v: String, e: String, out: String) extends PactProgram {
2
3    val vertices = new CsvFileSource[Int](v)
4    val edges = new CsvFileSource[(Int, Int)](e)
5    val output = CsvFileSource[(Int, Int)](out)
6
7    val undirectedEdges = edges flatMap {case (from, to) => Seq(from -> to, to -> from)}
8    val components = stepFunct iterate (s0 = vertices distinctBy {_._1}, ws0 = vertices)
9
10   override def outputs = output <~ components
11
12   def stepFunct = (s: DataStream[(Int, Int)], ws: DataStream[(Int, Int)]) => {
13
14     val neighbors = ws join undirectedEdges on { _._1 } isEqualTo { _._1 }
15                       map { (w, e) => e._2 -> w._2 }
16     val min = neighbors groupBy {case (to, _) => to} combine {cs => cs minBy { _._2 }}
17
18     // updated solution elements == new workset
19     val s1 = min join s on { _._1 } isEqualTo { _._1 } flatMap { (n, s) =>
20         if (n._2 < s._2) then Some(n) else None
21       }
22     (s1, s1)
23   }
24 }
```

Listing 5.6: A sample program in Sky, computing the Connected Components of a graph using a workset iteration. The algorithm propagates the component membership of a vertex iteratively to its neighbors [70].

5.4.1. Defining the Data Flow

As stated in the previous section, we construct the data flow of a program in Sky following the paradigm of Monad comprehension. An abstract distributed collection type (here called `DataStream`) represents in-flight data. The in-flight data is initially created from data sources and is transformed using second-order functions. The result is similar to a data flow graph, where nodes are intermediate results and edges are transformations on the data[4]. Line 7 of Listing 5.6 gives an example for the application of a second-order function: The data set "edges" (loaded in Line 4 from a file) is transformed using

[4]This reverses the way the PACT API or relational algebra expressions describe data flows, where the nodes are operators and the edges represent communication between the operators (i. e., intermediate results).

Operator	Syntax	Comment
map	x map f	1 : 1 record-wise transformation
	x flatMap f	1 : 0..n record-wise transformation
reduce	x aggregate f	global distributive aggregation
	x groupBy k aggregate f	distributive aggregation
	x groupBy k reduce f	generic set-wise operation
	x groupBy k combine f_1 reduce f_2	generic combinable set-wise operation
cross	x cross y map f	1 : 1 operation over cross product
	x cross y flatMap f	1 : 0..n operation over cross product
join	x join y on k_x isEqualTo k_y map f	1 : 1 operation over join result
	x join y on k_x isEqualTo k_y flatMap f	1 : 0..n operation over join result
cogroup	x cogroup y on k_x isEqualTo k_y map f	(m,n) : 1 binary set operation
	x cogroup y on k_x isEqualTo k_y flatMap f	(m,n) : 0..p binary set operation
union	x union y	bag union
iteration	h iterate (s = x, n = int)	bulk iteration
	h iterate (s = x, n = int, conv = ...)	bulk iteration with convergence check
	h iterate (s = x distinctBy k, ws = y)	incremental iteration

Table 5.1.: Operator Syntax Reference. Variables x and y represent inputs, k represents a key selector, f a user-defined function.

a "flatMap" operation. The actual transformation (the first-order function) applied by the second-order function "flatMap" is stated in the curly braces. The result of that transformation is assigned to the variable "undirectedEdges". We interpret the `DataStream` distributed collection type as a bag data type, i. e., there are no properties such as order or uniqueness explicitly represented. This choice is reasonable, because it gives the optimizer the degrees of freedom to make context-dependent choices to create, exploit, and preserve such properties.

User-defined functions appear as parameters to these second-order functions. They can be passed as function literals (closures) or as a references to a function definition. UDFs receive data as regular types from the language's type system for record-at-a-time operations, or as traversable sequences (iterators) of these for set-at-a-time operations. Closures close around the scope of all variables that they reference outside their own definition, allowing them to use these variables for parametrization of the code (e. g., values for predicates in Listing B.3, Lines 7-8). For cases where higher-order functions have more parameters than just the UDF (like match or set operations that require a specification of the keys) we use function chaining to create a fluent interface, as shown in Lines 14 or 19 of Listing 5.6. The complete grammar of API is shown in Table 5.1.

5.4.2. The Data Model

During its life cycle, a program will have three different data models. Programs are written against the *language data model*. This data model is designed to match the programming paradigm of the language in which user-defined functions are implemented. It supports primitive types, tuples, list and arrays, classes, and the (recursive) nesting of these. UDFs can follow a natural programming style, free of extra code and constructs to map the algorithm's data structures to the constrained data model of the optimizer or runtime. At the same time, the language data model is algebraic, which allows the compiler to analyze it and map it to a *logical tuple data model*, which is used by the optimizer. This tuple data model is flat, i. e., there are no nested structures. In the process of mapping the language data model to the logical tuple model, certain fields of the data structures in the language data model become individual fields in the logical tuple. We refer to those as *addressable fields*. Other fields in the tuples may contain entire nested structures as uninterpreted binary objects. The later is for example applied to structures with a variable nesting depth (such as recursive types). We will discuss the mechanism for flattening and the rules for addressability in the following paragraphs. Addressable fields may act as keys for certain operators, like Reduce, Join, or CoGroup. They are consequently the fields for which the optimizer tracks the data properties. After determining the best execution plan, the optimizer computes for each operator a *physical record layout* from the logical tuple model. The physical record layout is optimized for runtime efficiency and corresponds to a sequence of bytes in a buffer, where fields are accessed either through a fix offset in the byte sequence, or through an offset read from the record header. We choose to operate on buffers containing binary representations of the records, rather than operating directly on the language data model's objects at runtime for several reasons: Records are frequently exchanged between nodes over the network, consequently requiring them to be serialized anyways in many cases. In addition, buffer oriented execution allows to transparently move data between disk and memory or between memory from different processes, such as for storing or checkpointing data using shared memory. Finally, the binary representation is frequently a lot more compact than the object representation[5].

The separation between the logical tuple model in the optimizer and physical record model in the runtime follows the classical relational database design [2]. By means

[5] A typical 64bit JVM implementation adds to each object a header of 2 pointers (of which one is compressed, 12 bytes total) and pads the object to have a size which is a multiple of 8 bytes [65]. Consider the example of a tuple containing 4 fields (integers or floating point numbers with 4 bytes each), having a net memory requirement of 16 bytes. A generic object oriented representation of that tuple has would consequently require up to 64 bytes for the 4 objects representing the fields, plus 32 bytes for an array object holding the pointers to those objects. The memory overhead is hence more than 80%. A single custom code-generated object for the record still consumes 32 bytes - an overhead of 50%.

of programming against the algebraic language data model, which is mapped by the compiler to the logical tuple data model, we alleviate the impedance mismatch problem, which otherwise occurs when general purpose programming languages interact with declarative languages, such as in UDF embedding (cf. Listing 5.3). Because UDFs and language operators are defined in the same manner against the language data model, they mix seamlessly. In the remainder of this section, we summarize the type system of the language data model and how the individual types are mapped to the logical tuple model. A discussion with more examples can be found in [61].

The elementary data types of the language data model are the same **primitive types** that occur in common imperative programming languages, including 1-,2-,4-, and 8 byte signed integers, single- and double precision floating point numbers, boolean values, characters and character strings. All primitive types are mapped to a single field in the tuple model and can be addressed and used as keys for operators. The data model also supports **arrays** and **lists** of primitive types. Because arrays and lists may contain a variable number of elements, the entire array (or list) is mapped to a single field in the tuple model. As such, the array (or list) is addressable as a whole in the logical tuple model, but its contained elements are not. Consequently, elements of arrays or list may cannot be used in a key.

Composite types are supported in the form of classes. We differentiate two ways of supporting classes and inheritance hierarchies, which correspond to the functional notion of *product types* and *summation types*.

Product types encapsulate one or more data types in one type, similar to a struct in C, or a class in C++ or Java. In our case, the product type corresponds simply to regular classes. This includes **tuples** as a special case. Each member variable of the class is mapped to a logical tuple recursively (if they are composite types themselves) and then all these logical tuples are concatenated to create the flattened version of the class. Because there is a fixed and a priori known number of fields, each field is transparently addressable in the logical tuple. An exception are recursive types, where a type contains itself as a member. Since recursion leads to a variable number of fields in the flattened model, we store the flattened recursive member in a black box record in a single field of the logical tuple. No fields in the recursion are addressable, however, all fields in the logical tuple representation of the top-most (non recursed) type instance are. For a similar reason, the mapping will only include members defined in the declared type class. If a subclass is supplied at runtime, our data model will ignore all but the fields from the declared type. In the current implementation, we restricted the product types to Scala's case classes, because the compiler generates constructors, getter-, and setter methods automatically. There is no principle reason that prevents us from supporting other classes as well and dynamically add the necessary constructor, getter-, and setter methods.

Summation types are the functional counterpart of object-oriented class hierarchies. If a program uses types that relate to each other in an inheritance hierarchy, the complete type hierarchy must be known at compile time[6]. We flatten each class individually, similar to the product types. Where classes have a common ancestor, the commonly inherited members are mapped to the same fields in the logical tuple. The total number of fields occupied in the logical tuple is the maximum number of fields required by any type at the bottom of the inheritance hierarchy. As such, the handling of these types is similar to the *union* type structure in C. An extra tag field is always used to identify which concrete type is currently stored in the logical tuple. Even though all fields in each subtype are addressable in the logical tuple, we can safely address only the ones defined in the super-most type, as all other fields are not guaranteed to be present in every record at runtime.

Note that it is easily possible to relax the summation type condition that all classes of the hierarchy must be known in advance. It is possible to track the different encountered classes and assign tags dynamically. On the first encounter of a new type, the serialization would need to include the class name and the new type tag. Later encounters treat this type as described for the summation types above. The serialization library Kryo [73] uses this mechanism. In addition to defining new tags, we need to reserve additional fields at the end of the logical tuple for the additional fields of the subtype. The fields of a single type are here in general not laid out consecutively in the logical tuple.

Our model defines also a way to include arbitrary data types. The programmer must here explicitly specify the mapping to a flat tuple, the code for serialization/deserialization, and optionally operations like hashing/comparisons, etc.

5.4.3. Field Selector Functions

The field selector functions are Sky's means of referencing fields in the data structures. Such references occur for example when defining the key for a set operation like Reduce, or specifying the field for an aggregation. A field selector is essentially a function that returns a subset of the fields from a data type. During compile time, these functions are analyzed and transformed into the references to corresponding fields in the logical tuple. The function may return a single value (primitive type or composite type), or a tuple of types. Examples of field selector functions are in Lines 14 and 16 of Listing 5.6 (Page 87), where they are used to describe the join keys respectively the grouping key: The statement "_._1" accesses the first element (_1) of the supplied argument (the data type to be joined). Similarly, the statement "$case(to, \_) => to$" refers to the first field of a 2-tuple, using Scala's concise pattern matching syntax. In the simple example of

[6] Scala supports the concept of *abstract sealed types*, declaring that a type may only be extended in the same compilation unit. That way, all possible subtypes are known at compile time.

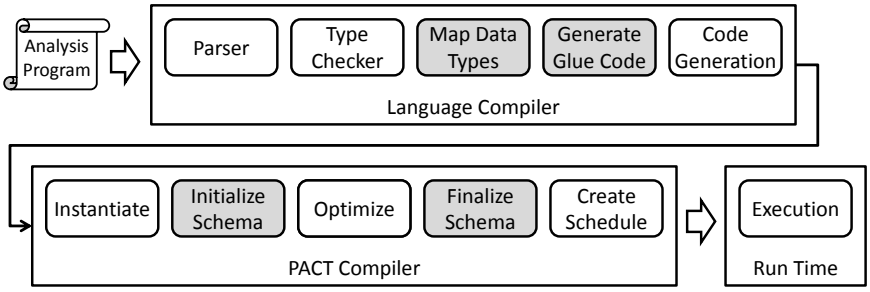

Figure 5.3.: The extended compile chain of the PACT API. The grayed boxed depict the additional steps.

Listing 5.6, all field selector functions select the first element of a tuple using equivalent and interchangeable Scala syntax (anonymous parameters in Line 14, pattern matching in line 16).

While field selectors are regular Scala functions[7], there are certain restrictions that apply, such that they must be function literals and can close around variables in an enclosing scope (i. e., they may not be parametrized). They may select only fields that are addressable in the logical tuple model, as defined in the previous section. If the function returns a composite type, all fields in the logical tuple representation of this type must by addressable. Finally, to guarantee that the functions are deterministic to analyze, they may not be recursive. In fact, our current implementation restricts them to cascading member field selections and pattern matching. A formal grammar is given in [61]. Key selectors are a special case of field selector functions. In addition to the above described conditions, all fields in the selection must be usable as keys. For composite types that means that they contain either no members, or all of there members are usable as keys. This definition is applied recursively.

5.4.4. The Compile Chain

The separation between the algebraic language data model, the optimizer's logical tuple model, and the physical runtime data layout requires additional steps during the program compilation. There are two data model transformations, the first from the language data model to the flat logical tuple model, and the second from the logical tuple model to the physical runtime layout. Together, these two transformations are reflected in the generated code that, at runtime, turns the physical records into the language data

[7]Note that Scala supports the *uniform access pattern*, where read access to public class member and calling parameter-less functions appear equivalent.

model's objects. The compile chain with the additional steps is illustrated in Figure 5.3. In summary, the compile chain consists of the following major steps:

1. Parsing and type, performed by the language compiler.
2. Mapping the UDF parameter and return data types to the logical tuple model and analyzing the UDF code. This step has a local view of one UDF at a time.
3. Generate the glue code for serialization/deserialization/comparison of the data types. Taking the information from the first step, the glue code is composed from templates, but information about final positions in the runtime layout is filled with placeholders.
4. Global schema generation. Before the data flow is submitted to the PACT compiler a logical global schema is created from all the local (per UDF) logical tuple mappings as input to the PACT optimizer. This schema generation ensures that maximum reordering potential exists.
5. Data flow optimization. The PACT optimizer picks physical execution strategies and potentially reorders operators.
6. Schema and glue code finalization. After the final data flow is known, this phase prunes away unnecessary fields for each operator and computes a compact local schema for each UDF's input(s). This schema information is used to finalize the serialization/deserialization code, replacing the placeholders with the actual fields offsets in the binary data layout.

The first transformation, which maps the language data model to the logical tuple model, happens in the Scala compiler (we discuss various ways to hook into the compiler in Section 5.5). Our mechanism operates on the program's abstract syntax tree (AST) and must consequently run after the type checking phase and before the (byte) code generation. The first additional step, the **data type mapping**, analyzes the data types of the UDFs' input (the parameters) and the return types and creates a flattened tuple layout for them, following the mapping rules described in Section 5.4.2. Because each UDF is analyzed individually, the result of this phase is a set of logical tuple layouts with a *local mapping* of types in the language data model to fields in the tuple, i. e., each individual flattened schema starts numbering its fields at zero. The data type mapping step also analyzes the field selectors. Given the mapping for the UDF input data types, all accesses to the data types' fields described by the field selectors are replaced with the indexes from this mapping.

Consider the example program given in Listing 5.7. The data types consumed and produced by the UDFs are the three case classes defined in Lines 20-22. In this simple example

```
1  class Relational(ordersInput: String, lineItemsInput: String, ordersOutput: String) {
2
3    val lineItems: DataStream[LineItem] = null
4    val orders: DataStream[Order] = null
5    val output: DataSink[OrderWithLineitem] = CSVDataSink(_.order.id,
6                                                          _.order.shipPrio,
7                                                          _.lineitem.price)
8
9    val joined = orders join lineItems on { _.id } isEqualTo { _.id } map {
10     (o, li) => OrderWithLineitem(o, li) }
11
12   val filtered = joined filter { o => o.order.status == 'F' &&
13                                       o.order.date.substring(0, 4).toInt > 1993 &&
14                                       o.order.prio.startsWith("5") }
15
16   val result = filtered groupBy { o => (o.order.id, o.order.shipPrio) }
17                        aggregate (sum, _.lineitem.price);
18 }
19
20 case class Order(id: Long, status: String, date: String, prio: String, shipPrio: String)
21 case class LineItem(id: Long, price: Double)
22 case class OrderWithLineitem(order: Order, lineitem: LineItem)
```

Listing 5.7: A sample Scala program using the Sky API, realizing a simple relational query (join, filter, aggregation).

here, we can think of case classes tuples with named fields[8]. The `Order` and `LineItem` data types are simply flattened into a 5- and 2-tuple, while the `OrderWithLineitem` class is flattened into a 7-tuple (first the fields from the nested `Order` type, then the fields from the nested `LineItem` type. The field selectors used to describe the join keys in Line 9 are both translated to value 0, as they access the first field in the class. The field selector in Line 16, which defines the grouping key for the aggregation, is translated to values $0, 1$, and the selector in Line 17, defining the field to aggregate, is translated to the position 6.

The analysis of the UDFs for the additional properties (accessed and modified fields) is a complex topic in itself and is discussed later in Section 5.5. For now, we assume that after this first step, we know for each function which fields in its parameters are accessed and which ones are returned unmodified [9]. In the example in Listing 5.7, the UDF in line 10 accesses all fields (as it creates a new object containing the entire objects passed as parameters), while the UDF in lines 13-15 accesses only fields 1, 2, and 3. Essentially all UDFs preserve all fields of their parameters (the filter returns the entire object unmodified, if it passes the filter, the join only concatenates the fields), except for the aggregation UDF, which changes the `extendedPrice` field.

[8] In fact, Scala's tuples are case classes where the fields are simply named _1 .. _n.

[9] For testing and simulation purposes, we support manual semantic assertions declaring the accessed and modified fields through statements like:
`filtered observes { o => (o.order.status, o.order.year, o.order.orderPriority) }`
`joined.right preserves { li => li.extendedPrice } as { pi =>`
`pi.lineitem.extendedPrice }`.

As a second step in the language compiler, we **generate the glue code** that transforms the binary records into the Scala objects and vice versa. This generated code wraps the Scala UDF and is invoked by the system whenever the UDF should be invoked. Recall that the data type mapping and the glue code generation happen independently for one UDF at a time. Because the final binary data layout cannot be known at this time (for example due to later reordering in the optimizer), the generated glue code is missing some information, like final offsets and whether fields are actually present (or pruned). Listing 5.8 shows conceptual code for the de-serializer of the `Order` data type as consumed by the `join` UDF in Listing 5.7. The class parameters `offX` denote the offsets where the records start in the buffers, the flags `pX` whether the fields are present. Variable length fields (like Strings) are addressed indirectly: their offsets refer to the position in the header where the actual offset is found (cf. [53], Chapter 13).

```
1  class Deserializer(off1: Int, off2: Int, off3: Int, off4: Int, off5: Int,
2                     p1: Boolean, p2: Boolean, p3: Boolean, p4: Boolean, p5: Boolean) {
3
4    def deserialize(b: Buffer, offset: Int, len: Int) : Order = {
5      val id = if (p1) b.readInt(offset + off1) else null
6      val status = if (p2) b.readStringIndirect(offset + off2) else null
7      val date = if (p3) b.readStringIndirect(offset + off3) else null
8      val prio = if (p4) b.readStringIndirect(offset + off4) else null
9      val shipPrio = if (p5) b.readStringIndirect(offset + off5) else null
10     return new Order(id, status, date, prio, shipPrio)
11   }
12 }
```

Listing 5.8: Sample code of a parametrized deserializer.

The remaining steps take place in the PACT compiler. Once the program is submitted to the system for execution, the PACT compiler first instantiates the program assembler function to obtain a description of the data flow. After the instantiation of the data flow, we traverse the data flow in source-to-sink order to create the **global schema**. This global schema contains a field for every field produced by a data source and thus is similar to a de-normalized relational schema, resulting from joining all inputs (base relations). In addition, it contains a field for every additional field and every new version of a field created by a UDF or operator, following the idea of *static single assignment* code [10]. An example for a global schema for the program in Listing 5.7 is shown in Figure 5.4 (a): Fields 0 and 1 represent the input fields from the `LineItems` source, and fields 2 to 6 those from the `Orders` source. The result of the join is fields 0 to 6, because all fields are kept constant; the same is true for the filter function. The aggregation operation adds field 7 for the aggregated `price`. One can think of the global schema to assign each field at every position in the data flow a unique name, except for the cases where we know that the field is a copy from a predecessor in the data flow.

The global schema generated by this step provides the input to the later analysis for pruning unused fields. In addition, the *static single assignment* (SSA) [10] nature of

the schema meets the exact conditions described by Hueske et al. [63] to detect cases where UDFs may be reordered: To optimally discover read/write conflicts on fields, the fields consumed/produced by different functions must have different (logical) positions, unless they are copies of each other. While Hueske et al. describe an API that allows to manually write functions and data flows that meet these requirements, their approach does not separate logical and physical schema. This forces each function to be aware of the full context and global schema: UDFs are consequently not specialized to a specific position in the data flow, but to the data flow as a whole. We believe that our approach complements the operator reordering techniques from Hueske et al. optimally, by automatically generating a suitable intermediate representation on which to operate. In the example in Listing 5.7, assume that the PACT optimizer decided to reorder the data flow by pushing the filter below the join. This is possible because the join preserves all fields that the filter accesses (fields 3, 4, and 5 in the global schema). As a result, the "filter" UDF follows the "orders" data source (Figure 5.4 (b)).

The final step in the optimization is the **schema finalization**, consisting of pruning unnecessary fields and compacting the schema. This phase takes place after the PACT optimizer has determined the execution strategies for the program and has potentially reordered the user-defined functions and operators. The pruning analysis is similar to data flow analysis in language compilers to eliminate unused variables [6]: We traverse the data flow sink to source, adding at each UDF/operator the variables that are accessed to a read-set, and pruning those that are produced but not in the successors' read-sets. Finally, create the compact physical runtime schema by taking the remaining fields and mapping them to a dense tuple, starting with the fix-length fields, then adding the variable-length fields. The UDF serializers/deserializers/comparators are then parametrized with information whether a field has been pruned and, if not, what its physical offset in the input record is.

Figure 5.4 (c) shows finalized schema. The data sink writes only a subset of its input fields (2, 6, and 7 in the global schema) - all other fields my be pruned in the predecessors. Similarly, the aggregation accesses only its grouping fields and the `price` field (1, 2, 6). The join forwards fields without accessing them otherwise. Following this, the pruning analysis will discard the fields for `status`, `date`, and `prio` (3, 4, 5) after the filter UDF, leaving these fields unset the data types produced by the join and the aggregation. In a similar way, the fields produced by the "lineitem" data source (0 and 1) are contained in the filter UDF's declared input data type, but are not accessed by the filer function; the respective deserializers will consequently not set them. As a result, the filter UDF, whose parameter has the `OrderWithLineitem` type, may follow the "orders" data source that produces records of a different data type (`Order`). Similarly, even though the join operator's UDF "`(o, li) => OrderWithLineitem(o, li)`" logically creates a concatenation of the entire order and lineitem type, it only joins a subset of the fields. This example illustrates how both reordering and pruning depend on the same interactions

of UDFs with fields in the global schema and mirror the operation of language compiler data flow analysis on a coarser level (fields instead of variables, side-effect free functions instead of individual instructions). The example illustrates nicely why this abstraction blurs the boundary between UDF programming and specifying algebraic operations.

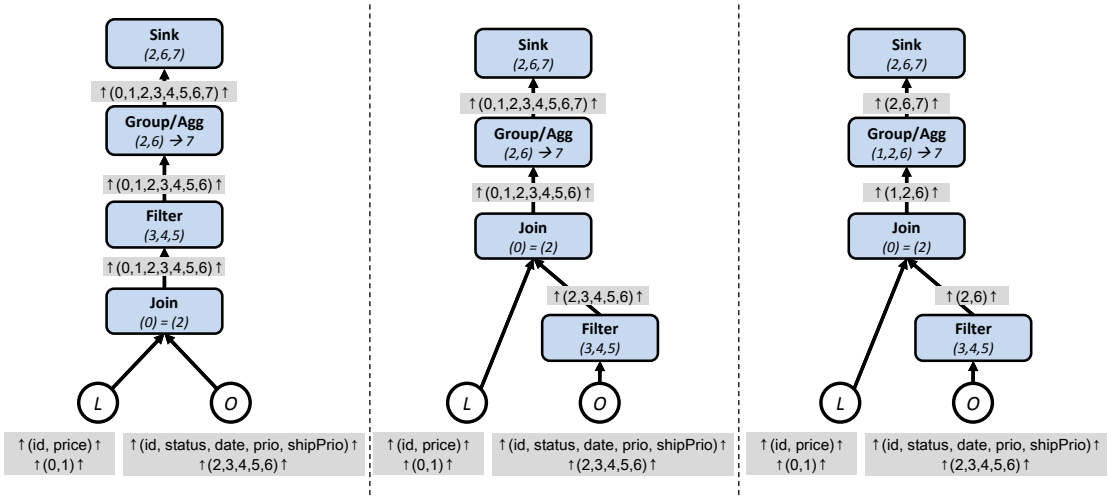

Figure 5.4.: The different schemas created by the compiler for the example program from Listing 5.7. The gray shaded boxed describe the fields forwarded at each point. The number inside the operator boxed denote the fields accessed and created by this operator.

5.5. Staging User-defined Functions, Code Analysis, and Code Generation

Our approach generates serialization/deserialization code for user-defined data types (as they appear in the signatures of user-defined functions) and performs a shallow analysis of the user code to determine which fields are accessed and which fields are preserved unmodified. This requires to obtain some form of representation of the UDF code which can be analyzed. Hueske et al. suggested to use a simple form of byte code analysis on the compiled UDFs to obtain the information about accessed fields and fields that are forwarded without modification [63]. This approach is a pragmatic solution for the analysis of simple Java UDFs: The Java compiler offers no flexible hooks to inject additional logic, and the PACT compiler has frequently only access to the byte code

of a UDF. Since the here presented techniques need to modify and augment the user code with generated glue code, we chose to make use of the Scala compiler's flexible architecture and to operate on the function's abstract syntax tree (AST) directly.

In the recent years, there has been active research around two different classes of techniques that are particularly suitable to lift user-defined code for analysis, rewriting, and code generation: *AST lifting/compile-time meta programming* and *lightweight modular staging*. The first class of techniques essentially allows programmers to hook into the language compiler at certain phases and get hold of the AST of the compilation unit or a local subunit (like an individual function) to modify and/or augment it, potentially using tools that abstract from the AST representation. It is useful if the majority of the code may remain unmodified and only parts need to be changed or augmented. One way to realize this technique is to use the Scala compiler's modular architecture and insert custom compiler plug-ins between some of the compiler's transformation phases. Such a compiler plug-in has access to the entire AST of the compilation unit, as well as to various information concerning the compilation context/environment. The plug-in implementer is responsible for finding the relevant part of the compilation unit's AST that he wants to analyze or modify. For the architecture of our language, a compiler plug-in is a viable solution, as it allows us to get access to the AST, analyze it, augment it with the glue code, and have the later phases of the compiler perform their optimization and the final byte code translation.

Another technique supporting compile-time meta programming, called *Compiler Macros* [35], has only recently been added to the Scala compiler. In essence, it allows the programmer to declare abstract functions with special signatures, where the compiler invokes a handler function whenever it translates an invocation of this function. This handler gets passed the AST of its parameters, allowing it to analyze user-defined functions in the case where a parameter is a function or closure. Because the handlers are specialized for macro definitions and directly invoked with the macro function parameters, they need no mechanism to locate the part of the compilation unit's AST that describes the user-defined function or -data types. The macro infrastructure also supports tools to get an AST of a pre-defined piece of code and to merge such (modified) pre-defined ASTs with ASTs from UDFs, making it a powerful tool to analyze, rewrite, and augment user-defined code.

An entirely different approach is implemented by the *lightweight modular staging* (LMS) [90] technique. LMS follows the idea of *generative programming*, where instead of creating a program, one creates a program generator that can then create highly specialized code for a certain target environment. It replaces the types that the program normally works on by special `Rep` types. These `Rep` types provide the same functions that the normal data types support, but rather than implementing the actual logic/behavior, these functions create a symbolic description of themselves. As a consequence, the programs written

against the LMS abstraction look like regular Scala programs, but, when invoked, create a representation of themselves in a form comparable to a simple AST. This AST is amendable for analysis and can be used to generate code. The goal of LMS is to allow for simple definition of code lifting libraries, in order to gain a high level of abstraction, but retain efficiency through specialized code generation. It is possible to realize the compile chain described in this chapter using LMS. Instead of a plug-in to the Scala compiler, the LMS library would lift the entire UDF code. Similarly, the key selectors could be lifted with LMS. The analysis for the data type mapping could happen in a similar way as described. It would however take place outside the Scala compiler, but rather directly before the global schema generation. At the point where we currently finalize the generated glue code, LMS would do its entire code generation.

While LMS is a good match for compiled DSLs that attempt to create highly specialized code, it not the perfect match for our requirements. The gravest problem is that LMS's method of staging code does not support all language constructs. Even with specialized versions of the Scala compiler (Scala Virtualized [80]), several constructs, like class definitions or variable declarations, are not lifted. Furthermore, lifting user defined or third party classes requires manual effort (defining `Rep` types). While this may present only a minor inconvenience to experienced DSL authors (as they can be expected to understand the difference between staged and non-staged code), it likely exceeds the abilities of many end users, who may not be professional software developers. Our UDFs desire full coverage of all language constructs, seamless interaction with third party libraries, and that our UDF code needs no rewriting, but only augmentation with glue code. We therefore implemented our code generation for the first version of the Sky language using a Scala compiler plug-in, set to run after the type checker phase and before the lambda lifting phase. To detect sites that require data type analysis and code generation, we use Scala's mechanism of *implicit parameters*. These implicit parameters are parameters that are automatically provided by the compiler, if found within a certain scope. In the signatures of the second-order functions, we define the utilities (e. g., serializers/deserializers) for the function's data types as an implicit parameter. The compiler plug-in detects an unsuccessful search for an implicit utility and generates the sought utility. This has the advantage that custom utilities can be provided by the programmer through simply declaring them as an implicit value in scope.

In the most current version, we realize the data type mapping and glue code serialization using the compiler macro feature. Because macros are handlers that run in the Scala compiler when compiling a UDF or data flow description, they work very similar to the code in the compiler plug-in. Because the macro handlers are invoked whenever the compiler translates a call to a macro function and passed the local context of this invocation, we can omit the previously mentioned mechanism that uses implicit parameters to detect the call sites where parameter types need to be analyzed.

5.6. A Performance Overhead Comparison

In this section, we discuss benefits and overheads of the Sky API compared to the classical Java-based API. Assessing the effectiveness of an API or language is difficult, as there are no hard measures to refer to. Regarding the effort for the programmer to write programs, the lines of code may give a rough tendency. However, across different programming languages (here Java and Scala), the number of code lines allows only for a limited comparison. From our personal experience with programmers that we taught both APIs, we can say that the Scala-based API leads in the vast majority of cases to better productivity, meaning faster time-to-result and fewer errors.

The techniques presented above allow the Sky API to raise the level of abstraction compared to the PACT API, while maintaining good performance. In the following, we give an evaluation of the performance impact of the individual techniques used in the compile chain, in comparison to a naïve implementation of the Scala API.

We choose three example programs with different schema characteristics: The *Word Count* program from Section 5.1 (Listings 5.1 and 5.2) has a very simple schema, consisting of a primitive type string in the first step, and String/Integer tuples in the second step. The second program is the *relation query* used as an example in Section 5.4.4. We slightly modify it here to apply the filter before the join (as in Listing Appendix B.3). The query has a wide schema, out of which few fields are needed such that it is possible to apply aggressive field pruning. The final test program is the step function of the *K-Means* clustering algorithm, as given in Appendix B. We use the non-iterative variant here to factor out the influence of iteration specific mechanisms (such as data caching). The function is highly compute intensive through the comparison of every data point with every centroid. During this computation, however, many fields remain unchanged and are simply forwarded by the functions.

We test each of the above described programs with various configurations of the compile chain, as shown in Table 5.2. The first configuration (*NoOpts*) corresponds to a naïve implementation of the Sky API on top of the PACT API. No special optimization is performed and the key selector functions are executed for every tuple like a regular function to extract the key. The next variant (K) analyzes the key selectors at compile time and saves the overhead of executing the key selector functions at runtime as well as the extra space otherwise occupied by storing the designated key. As a next step, we activate schema compaction (KC). This reduces the number of fields that the physical schema has. However, without knowing what fields a function will access and which ones are forwarded unmodified, the schema compaction cannot yet prune these fields. It mostly eliminates fields reserved in the global schema but are not set at runtime anyways (and remain null), reducing the sparsity of the schema. Effective pruning becomes possible when activating the next optimization (KCA), which uses function analysis,

respectively assertions, to deduce which fields of the input data types are accessed by the functions, which are discarded, and which are returned unmodified. This allows pruning of discarded fields as well as circumventing the (de)serializers for fields that are only forwarded unmodified. The final optimization (*KCAM*) uses mutable objects for the data types passed to the functions, rather than immutable types. This allows the deserializers to reuse the objects and takes some pressure off the garbage collector.

NoOpts	The program is executed without any schema optimization and key selector functions are executed.
K	Key Selector functions are analyzed at compile time and not executed at runtime.
KC	*(K)* + Schema is compacted after the PACT compiler's optimization, resulting in smaller schemata.
KCA	*(KC)* + Code analysis or assertions give knowledge about which fields are accessed in a function and which are returned unmodified. This results in smaller schemata.
KCAM	*(KCA)* + Mutable object are used and re-used when the user code functions are invoked.

Table 5.2.: Configurations for the experiments enabling and disabling selected features of the compile chain.

The experiments were run on a four node cluster. Each machine contained two quad core Intel Xeon E5530 processors running at 2.40 GHz, disk array capable of supporting sequential reads at up to 400 MB/s. Worker processes used 36 GB of available RAM, where 26 GB were reserved by the PACT runtime for sort- and hash algorithms. The remainder was available for utilization by user code functions. The HDFS replication factor was set to two and data files used 128 MB block sizes. The default degree of parallelism set for each job was 32. The Figures 5.5, 5.6, and 5.7 show the experimental results. The blue bars (PACT4S) show the runtime of the Scala programs going through the here described compile chain. We add as a baseline the performance of the same programs written against the PACT API (Chapter 3), once implemented in Java (dark gray line) and once in Scala (red line), to assess the effects introduced by the language compilers.

The results of the Word Count experiments are shown in Figure 5.5. The algorithm processes roughly 10 GB of text. The unoptimized version has about 40% overhead when compared to the PACT implementation. Compile time analysis of the key selector functions shows the most significant performance gain in this experiment. The schema compaction adds marginal benefit, which is expected, given the very simple schemas of the Word Count program. With information about which fields are accessed (KCA), the reduce function does not deserialize the word strings any more, as the function only

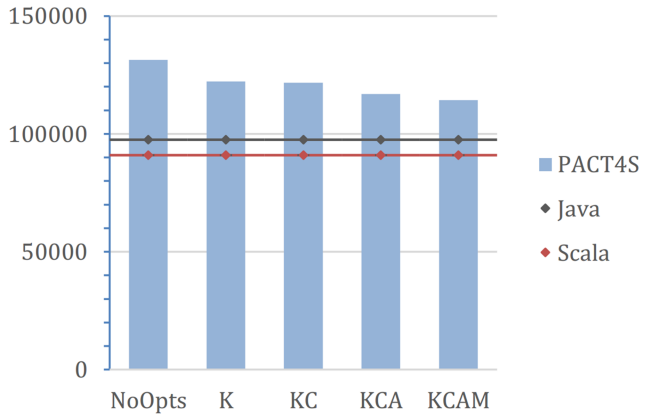

Figure 5.5.: Performance of the *Word Count* program with different features of the compile chain enabled. See Table 5.2 for a description of the different experiment configurations.

counts the elements per group without looking at the elements themselves. Using mutable objects gives another minor performance gain.

Figure 5.6 illustrates the results of the experiments with the relational query, which processes the "Orders" and the "Lineitem" table from the TPC-H benchmark [100] with scaling factor 100, encoded as CSV files. The data size is roughly 16.5 GB for the order table and 74 GB for the line item table. With all optimizations disabled, the query takes almost three times as long. The key selectors contribute a relative small performance gain. The schema compaction alone only cannot do much, but with additional semantic knowledge about accessed and modified fields (KCA), it prune unused fields early, reducing the size of shipped data, the size of the join hash tables, and even the parsing overhead for the CSV files. With this optimization, the performance of the Sky program is en par with the hand optimized PACT programs. Using mutable objects instead of immutable ones does not contribute visibly to the performance.

The performance of the unoptimized K-Means step function is more than four times slower than that of the PACT implementations. Analyzing the key selector functions brings this overhead down a bit, while schema compaction yielded an additional 70% gain, relative to the baseline. As in the previous experiment, the greatest improvement is achieved by the analysis/assertions about field accesses and copies. In contrast to the relational query experiments, where knowledge about accessed and written fields allowed to prune unneeded fields, the program can here forward many fields directly from the input to the output in binary form, reducing the (de)serialization overhead. This is especially important in the Cross operation that computes the distances between the

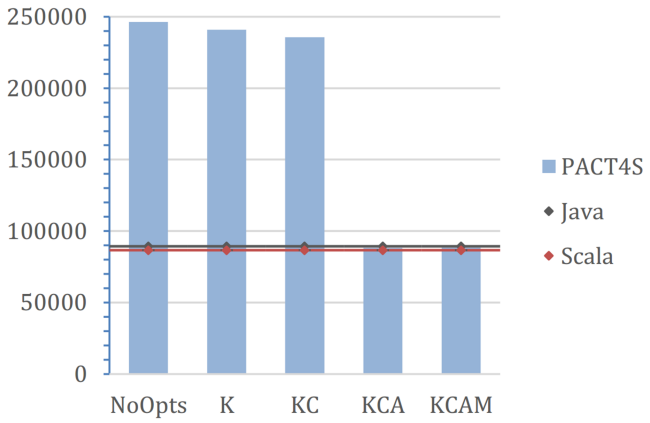

Figure 5.6.: Performance of a *Relational Query* (Appendix B.3) with different features of the compile chain enabled. See Table 5.2 for a description of the different experiment configurations.

centroids and data points, because this function is invokes very many times. Mutable types accounted for another 15% performance improvement.

In conclusion, the experiments show the value of the optimizations performed by the Sky compile chain. The programs from the still have a noticeable overhead, compared to their native PACT cousins. Possible reasons for this are plentiful: While the PACT API allows to write programs which work with no object allocation at runtime, the generated serialization code still creates objects, thus putting more pressure on the JVM's garbage collector. Furthermore, the generated code goes through additional virtual method calls in the serializers and deserializers. These problems are, however, ultimately attributed to the specific implementation of the code generation, and not fundamental. The Java versus Scala baselines shows that the performance of Scala programs per se is highly competitive with that of Java programs.

5.7. Related Work

The related work to our approach can be divided into two broad categories: The first one deals with alternative designs for parallel programming language frontends and APIs. We will discuss alternative languages highlight which of the here motivated problems they address and what their techniques are. In the second part, we elaborate on alternative approaches to lift the user-defined function code, to analyze it, and to generate the glue code between the UDF and the runtime system.

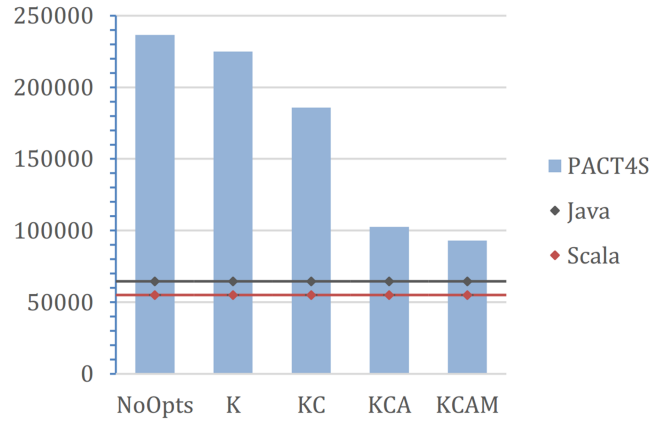

Figure 5.7.: Performance of the *K-Means* program with different features of the compile chain enabled. See Table 5.2 for a description of the different experiment configurations.

5.7.1. Other Language Frontend Designs

Flume Java [41] implements a fluent API for MapReduce in Java. Its design is very much determined by the effort to lessen the overhead induced by the inherent mismatch between Java's imperative programming paradigm and the functional nature of the MapReduce model. The authors recognized that problem and motivate their approach as a pragmatic solution to the problem that fits into the predominant Java ecosystem. Flume Java performs some limited optimization (for example map function chaining), but its optimization potential is fundamentally limited by the key/value pair data model and the small set of primitives it adopts from MapReduce.

The *Spark* system [112] from the University of California, Berkeley, is the result of implementing a fluent syntax in a functional language. Similar to our solution, they use Scala as the host language and make extensive use of its rich data model and its ability to serialize closures. Spark gets rid of serialization glue code through a generic serialization library (Kryo [73]). In contrast, our approach features a well-defined data model based on the concept of algebraic types. This model is a central tenet of its design and serves as the foundation for several of its core features, such as field selectors and the transparent mapping to the logical record model, which in turn enables deep optimization. While simple programs look compellingly concise in Spark (and in fact bear great resemblance to those in Sky), its overly simple data model (key/value pairs) prohibits advanced optimization as supported by our system, like push-down of operations into the constant data path. For operations that require a key, programs must transform

the data into a key/value pair. Sky's richer and well-defined data model, with its concept of field selectors, allows it to apply its built-in operators beyond simple 2-tuple data types. Spark's runtime does not support iterations per se, but it is feasible for Spark to write an iteration loop. The Spark client repeatedly pushes work requests to the runtime with comparatively low latency, caching intermediate data sets in main memory. Incremental iterations can only be emulated by explicitly copying unmodified elements of the solution set, which often adds significant overhead in later supersteps, where few elements change. Microstep iterations are not possible in this model.

Scalding [92] implements a functional language frontend on top of the Cascading library [37] for Hadoop [106]. Like the solution presented in this chapter, it uses the Scala language to host the abstraction. Scalding provides a fluent syntax and uses standard mechanisms and serialization libraries to avoid much formal noise. Of particular interest is the tuple-oriented API, which provides some form of logical abstraction from the physical data model. All fields in the tuple data stream are always named, starting with the data sources that assign initial names to the fields of their produced tuples. UDFs state the names of the fields they consume and produce, allowing functions to operate on tuples irrespective of the number of fields present in the tuple. Each produced field is appended to the tuple unless the result name exists already and unneeded fields may be projected early. Because Scalding does not lift the code for analysis (at least of the parameter types and the return type), it is limited to the tuple data model and cannot perform compile-time type checks. In comparison, Sky's data model is much more expressive and is mapped by the compiler to a flat model. Fields are implicitly named by the optimizer during the global schema generation, and unused field pruning happens automatically during schema compaction. Sky also avoids explicit stating of the read and written fields by performing code analysis to determine them. Furthermore, Sky supports iterations and performs much deeper holistic data flow optimization. The later is partly due to the fact that Scalding builds on Cascading and consequently the Hadoop stack, which limits the set of possible execution strategies.

ScalOps [104] is an embedded Scala DSL designed for machine learning algorithms. Like all the other languages discussed here, it provides data parallel primitives such as map, reduce, CoGroup. Unlike the previously mentioned languages, but much like our approach, it features a lifted looping construct for iteration. The design of ScalOps' user-facing API is relatively similar to that of our Sky language. However, both frameworks differs significantly in their underpinnings. ScalOps targets the Hyracks platform. Its compiler turns programs into relational query plans via an intermediate Datalog representation [33]. Loops are translated to recursive query plans, thus avoiding a client-side driver. While the authors published the specification and optimization via recursive Datalog programs, there is no published material available to date that describes ScalOps' approach to language embedding and the translation of lifted expressions to Datalog rules.

5.7.2. Code Analysis and Code Generation

Serialization frameworks like Avro [12], Thrift [16] and the Protocol Buffers project [56], realize some form of code generation (specifically data types and data type utilities like serializers) without explicit support from the host language's compiler. In their approach, the programmer declares types in an auxiliary interface definition language. Prior to the compilation of the actual user program that uses these types, the auxiliary schema files must be processed by some compiler to generate JVM byte code consumable by the user's program. This approach not only complicates the compile chain and introduces an additional tool-set for the programmer to learn, it also severely complicates the analysis of program code interacting with these types. A good example for that is the analysis of the field selector functions, which would need to work against generated types and schema defined in an external file and language. These hassles can be avoided by working directly with the program's abstract syntax tree.

The *Jet* Framework [3] defines a programming model for distributed data-parallel processing and embeds it into the Scala language. While the user-facing part of the framework is similar to Sky or Spark, its implementation is based on the concept of Lightweight Modular Staging (LMS), which was briefly discussed in Section 5.5. The essential differences between our approach and Jet boil down to a large extend to implication of choosing LMS to host the DSL, or using a compiler hook analyze the user-defined code. Jet performs aggressive optimization of the actual user function code, while our design incorporates an optimizer for holistic data flow optimization. Jet exploits LMS' inherent side effect tracking to detect and prune unused fields from the data flow. Our global schema generation and compaction achieves a similar result. Both approaches require explicit knowledge about the function behavior. LMS assumes by default that operations are pure and provides a set of annotations that can be applied inline to a function definition to indicate its local side effects, such as object allocation or mutation. This is a suitable approach for languages that target an expert audience, or even DSL authors. Since our approach aims at being a general-purpose data analytics language, we take a conservative view and assume that functions are impure, guaranteeing correctness in the absence of either user-defined-, or analysis-derived specifications. Our semantic assertions are at a higher level of abstraction than low-level side effect annotations. Because Jet stages the user code, it can apply aggressive optimization, like chaining of operations together to reduce overhead. Through LMS, Jet also features an extensible code generation layer that allows it to target multiple different backends, currently Spark and Crunch. The price is that Jet restricts the language to what is supported by LMS, which currently excludes several constructs of the Scala Language, or requires manual wrapping into LMS `Rep` types. Current Jet attempts to lessen the burden on users by providing a script to parse Scala classes and generate the corresponding `Rep` types. Since the script only does an incomplete job, it would be interesting to see if a compiler plug-in could provide an

automatic lifting facility robust enough to alleviate these concerns.

In essence, the design goals of the project are different. If you aim at generating efficient code for multiple target systems, lifting the user-defined code is a prerequisite. Jet with LMS provides a suitable architecture for that, but requires that concessions be made with respect to the richness of the languages type system. In contrast, our approach based on compile time code analysis and -generation places no restrictions on the language used in the user functions and still achieves very deep optimization, but cannot generate code for different platforms.

The *Delite* Compiler Framework [32,91] aims at decoupling parallel programming language frontends from the execution engines. It allows for the creation of DSLs that compile to different systems and hardware platforms (currently Scala, CUDA, and C++). In contrast to Jet, Delite is not a DSL but framework for building DSLs, although the boundaries in this space can become somewhat fluid when the domain encompasses an entire programming paradigm. Similar to other language compiler frameworks (like for example LLVM [75]), it transforms all languages into an common intermediate representation of primitives, from which it generates code for the specific target system or platform. This intermediate representation contains rich primitives for many data parallel operations, such as the second-order functions map, reduce, and zip, as well as a divide-and-conquer primitive used for looping/recursing. Delite uses LMS to define DSLs and translate them into the intermediate representation. Currently, its code generation layer focuses on multi-core parallelism and GPU offloading. However, since this layer is extensible, it is not difficult to imagine DSLs like Jet being built on top of Delite to target both in-memory and distributed data processing scenarios simultaneously. The discussion comparing our approach to Jet applies to Delite as well, given their common LMS underpinnings.

It is theoretically possible to move the code generation of the entire UDFs until after the data flow optimizer, or into to the runtime system. There are some projects that investigate technology related to that approach, all of them investigating techniques for just-in-time (JIT) compilation and code generation. The *Truffle* project [107] created a virtual machine with self optimizing abstract syntax trees. The runtime takes code that is written in a very generic fashion (such as a context agnostic record model) and dynamically generates optimized low-level code for the hot paths (specific to the final position of the UDF in the plan). Truffle targets more dynamic environments. Given that the context of a UDF remains stable after the data flow optimization, compile time code generation techniques produce better results. First, they can fully exploit all schema information without a ramp-up phase to determine the hot path. Second, they need no safe-guards in the generated code to detect changes in the code paths and to "de-optimize" and start a new round of detecting hot paths.

The *QVM* (Query Virtual Machine) project at Oracle Inc. investigates techniques to run user-defined code deeply linked with the database runtime. They make extensive use of just-in-time compilation techniques to generate code that merges the database operator implementation and interacting UDFs, having no runtime separation between database kernel and user code. Unfortunately, we do not know of any publications, yet, detailing the approach.

5.8. Conclusion

We have presented the design of the Sky domain-specific functional language for data analysis, embedded in the Scala programming language. The language is designed for fluent composition of programs with user-defined operations. The language design reduces program fragmentation and frees programmers from writing code to bridge between the algorithm and the executing system, thus severely reducing the programs formal noise. The compiler, consisting of a Scala compiler plug-in and a standalone Java step, creates efficient programs to be executed with the Stratosphere runtime.

The compilation process allows the Sky language to separate the surfaced data model (a large subset of Scala's data model) from the data model used by the distributed runtime. Internally, the compiler uses data type analysis, code analysis, code generation, and schema generation and compaction to efficiently map the user-defined functions to Stratosphere's runtime. The result is that functions and programs are very concise and reusable and at the same time efficient in the execution.

The here presented language is a major step forward for parallel programming APIs towards an abstraction level that is similar to that of relational query languages. At the same time some open issues remain. One big distinction to relational queries is that Sky programs need to always explicitly construct the result types of operations like joins (somewhat necessary for languages with such a flexible data model), where relational queries simply use the union logical attributes for the result schema. While it is possible to add default result types for some operations and data types (for example for the join of tuples), the result types would not retain the logical names of their parts. The realization of such a feature would need language support such as *computed types*, which are a complex feature and not available in Scala or any major programming language. An implementation of a similar mechanism inside the schema generator is possible, but forces the API to give up type-safety. The deep integration of the relational data model with the programming language's data model *inside* the language's type system is an open question for programming language research. Given the available techniques of today's programming languages, we believe that the approach presented in this chapter is an elegant and strong solution.

6. Related Systems

To the best of our knowledge, the techniques presented here are novel, especially in the way they are combined. We have discussed work related to the individual contributions of the thesis in the respective chapters. In this chapter, we would like to specifically discuss two related research projects and the systems they have built: The *Asterix* project and the *Spark* project. The discussion here focuses on the overall architecture of the systems, rather than on relationships between individual techniques. We will in how far these systems achieve similar goals and how the designs differ.

6.1. The Asterix Project

The *Asterix Project* [25] is research and development effort headed by the University of California, Irvine (UCI). The current state is a first release of the system "Asterix DB", which offers a complete data processing stack, from query language, optimizer, runtime, and storage. The Asterix architecture combines both classical parallel database techniques with techniques that more recently sprouted in the context of parallel runtimes for non-relational data.

Asterix DB supports the Hive and Pig query languages from the Hadoop ecosystem, as well as its own language "Asterix Query Language" (AQL) for semi-structured data. AQL builds on a data model that is similar to JSON, but with the possibility to define strict schemata and extensible schemata. The system uses a common optimizer framework *"Algebrix"* [30] for all its supported languages. Algebrix is an extensible rule based query optimizer that works data model independent, but data model specific rules may exist. The data models are represented to the runtime through a set of utility classes that encapsulate serialization and field accesses, record manipulations, and operations like hashing and comparisons.

Our concepts of PACTs is related to this design: PACT acts as the intermediate representation of an optimizer that supports multiple query languages and programming APIs on top: Sky, Meteor [62], Pig [66], and a proof-of-concept adoption of the JAQL language [26]. At the same time, we maintain PACT as a data-parallel UDF programming abstraction. The second-order functions define through their constraints the core set of logical-to-physical transformation rules that are enumerated by the optimizer. Additional

transformation rules come from two different sources: First, the latest version of the Stratosphere system uses the original PACTs as the basis of an operator hierarchy that specializes them to joins, aggregations, etc., for which logical transformation rules exist. Second, Hueske et al. have shown that elementary rules for reordering exist and that the conditions are deducible from read/write conflicts on attributes. These rules represent additional logical transformation rules. Overall, the Algebrix optimizer framework can perform more transformations, but requires in general more semantic knowledge; the approaches followed in Stratosphere aim to perform as much optimization as possible without explicit specification of higher semantics. Data model independence in the optimizer happens in an analogous way in both Algebrix and PACT. The schema finalization and code generation as described in Chapter 5, Section 5.4.4 realize the same gluing between the optimizer's result and the data model dependent runtime operations.

The Asterix project runs its programs on the parallel query engine *Hyracks* [31], while we run programs written in Sky and PACT on the Nephele streaming system. Hyracks jobs are in general DAGs or operators (or stages of operators) while the Nephele system implements a DAG of data streams, with additional information about materializing/pipelining behavior of the task that consume and produce the streams. Both abstractions support controlled cyclic data flow graphs as needed for iterative/recursive computations. Because the execution engines are orthogonal to the query abstraction and query compilation techniques presented here in this thesis, the execution engines are to a large extent interchangeable.

Asterix DB offers a separate abstraction for vertex-centric computing (Pregel-style [78]) called *"Pregelix"*. Under its hood, the query execution is similar as the optimizer plan created for Pregel-style programs by the techniques described in Chapter 4.

The *"ScalOps"* machine learning language [104] is not a direct part of Asterix DB, but part of the research ecosystem of the Asterix project. It follows many similar goals as the Sky language presented in this thesis. From a language design point, many aspects are shared, such as using functions to extract the relevant keys. In contrast to the approach taken in Sky, there is no analysis of the data model and the key selector functions in ScalOps - the functions are executed at runtime, rather than translated into field references. Similarly, the current publications leave it open how the flexible data model in ScalOps, which is essentially that of the Scala language, is integrated with the runtime or the Algebrix optimizer. In Chapter 5, we present exactly these techniques using code analysis and code generation. A related publication by the same authors [33] describes a way to optimize machine learning programs when specified via recursive datalog rules. This optimization arrives at similar results as the techniques presented in Chapter 4. The published techniques are however more suited to design language abstractions than to expose them directly as a query interface. Additional discussions can be found in Section 4.6.2 and Section 5.7.

Among all projects we know of, the Asterix project and system are closest to the techniques presented here. The system stacks of Stratosphere and Asterix are very similar in their structure. In many fields the Asterix project has solved similar problems in a different way and arrived at a comparable solution.

6.2. The Spark Project

The Spark project is part of the *"Berkeley Data Analytics Stack"* (BDAS), developed at the University of California, Berkeley (UCB). At the project's core is the Spark system [112], which builds on second-order functions over distributed collections. On top of Spark sit various extensions, such a SQL compiler (Shark [108]), an interface for streaming-like programs (Spark Streaming [113]), and libraries with various templates (for example vertex-centric graph processing). In contrast to the Asterix project and the works presented here, the Spark Project is more a collection of interfaces and compilers that have in common that they all execute on the Spark system, rather then sharing a common language or algebra core.

The Spark system [112] forms the core of the project and the base of the analytics stack. Spark's abstraction centers around so called *"Resilient Distributed Data Sets"* (RDDs), which represent the intermediate results and are transformed through operations that are similar to the primitives of PACT. Spark programs are regular Scala programs that use second-order functions on the RDD data type. These programs are not deployed into the parallel runtime as a query data flow, but executed in batched fashion, one operation after the other, as the program invokes them. Data types are serialized for network transfer through either Java's built-in serialization mechanism, or through the Kryo framework [73]. Through that abstraction, Spark programs have out-of-the-box a powerful control flow mechanism, which can implement loops and programs that are composed incrementally, statement by statement in a shell-like fashion.

Spark's goal is to provide a simple but powerful abstraction for distributed operations. The API to define programs uses a key/value data model and translates set transformations directly into a distributed operation. In contrast to that, our Sky language offers an algebraic data model and field selectors functions in order to apply the available operations to complex data models beyond key/value pairs. Sky maps these Scala programs to PACT program specifications through generating a flat schema with logical global fields. That way, programs can use more complex data models than in Spark and are still amendable to the relational-style optimizations of PACT.

Shark [108] implements a SQL interface on top of Spark, translating SQL into a series of transformation on the input data sets. The example of Shark shows nicely the differences between the approach taken by Spark and the approach of our work and Asterix: While

Shark comes with a domain specific optimizer for SQL, both Asterix and the PACT have a common optimizer that is abstracted below generic set primitives. The result is that the programs and higher level language front-ends implemented on top of PACT (Chapter 3) or Asterix's Algebrix abstraction [30] benefit directly form the optimizer. Sharing the optimizer between multiple font-end languages also implies that it is possible to holistically optimize programs consisting of parts written in different front-end languages.

In Spark, iterative programs are simply repeated invocations of the same functions over the previous result. This resembles the mechanism for iterative programs in Hadoop (loop outside the system), but the API makes them much more elegant. In addition, the intermediate results are kept in memory, which circumvents the overhead of repeated (distributed) filesystem I/O, when compared to Hadoop. In contrast, our approach in Stratosphere runs the loop inside the system, which gives it the ability to offer the workset iteration abstraction that does not need to touch the entire intermediate result in every iteration. Furthermore, the optimizer abstracted below the PACT model can automatically optimize iterations with respect to identifying loop invariant data and re-using data properties across supersteps.

In comparison to Stratosphere and Asterix, Spark resembles more a collection of efficient distributed tools, while the other two systems are centered around algebraic models that can deeply integrate the different front-ends and share system components like optimizers.

7. Conclusions

Data analytics is becoming increasingly important for many companies and scientific institutions. In addition to simply scaling to larger data volumes, the fast developing field of *advanced analytics* requires to evaluate complex algorithms in parallel over data in various formats. Machine learning, data mining, and graph analysis algorithms are at the heart of this new direction of data analytics that aims at modeling the data with its hidden patterns and relationships, instead of only describing it with statistical moments. At the same time, the classical operations still play a major role. They are needed as part of traditional descriptive analytics, and in data exploration, transformation, cleansing, normalization. The later are often executed as part of data preparation phases such as ETL (extract, transform, load) for warehousing or to preprocess data before the training of statistical models.

The recent research efforts have suggested a series of systems which address the problems of bringing parallel analytics to different data structures, new types of operations, or complex iterative algorithms. In most cases, these systems specialize on certain types of operations and are only able to cover a very specific subset of operations efficiently. Parallel relational databases are highly efficient at SQL-style analytics, MapReduce related systems are good at handling unstructured data and non relational operations, graph analysis systems are efficient for the analysis of graph structures data and for certain machine-learning algorithms. Few systems have taken on the challenge to create abstractions and runtimes that are able to efficiently represent and execute problems across these domains. The techniques presented in this thesis support the creation of an abstraction for efficient analytical algorithms across the here states use cases.

The relationship between technologies based on the MapReduce idea and the technological fields of relational databases (and their offspring) has been rather controversial. While MapReduce opened up the field of parallel analytics to many new applications, the approach gave up many of the well-researched and time-tested benefits of relational databases - most importantly the benefits of schema and a declarative language that abstracts from execution-related details and can be automatically optimized. These benefits are widely considered important in making analytical applications testable, maintainable, and efficient. Because of giving up these features, MapReduce has often been labeled “a major step backwards”. Efforts to bring relational database techniques to MapReduce resulted in either giving up the generality of MapReduce, or in wiring

relational operations and MapReduce style operations together in a work flow fashion rather than integrating them.

We believe that the concepts presented in this thesis allow to define a sweet spot between analytical relational databases and MapReduce techniques. They combine many of the advantages of both fields, offering many of the important benefits of the relational database approach without giving up the generality of MapReduce style systems. We employ techniques from the fields of databases and compiler construction to bring relational techniques to non-relational domain specific programs. Data type analysis and mapping and code generation bridge between the more general programs and the constrained algebraic database abstractions. At the same time, we present techniques that push the abilities of our abstraction beyond the ability of both MapReduce and relational databases, allowing programmers to write complex iterative algorithms efficiently. Our approach is based on a principled data flow abstraction for iterative algorithms. For algorithms that are typically stateful and incrementally improve the state, we suggested an abstraction based on worksets and deltas. We provided optimization techniques for iterative programs that deal with cross-loop optimization and loop-invariant data. The suggested abstraction is powerful, subsuming dedicated systems while offering competitive performance.

The here presented research results lead to a series of interesting follow-up questions:

- The techniques used to create the Scala-based functional programming front-end (Chapter 5) are not specific to Scala, or functional programming languages. It would be interesting to see in how far the here presented abstractions can be applied to a language like SQL. SQL could be extended with a complex data model and an adoption of the selector functions (for example in the form of simple path expressions). Using a lambda syntax for inline user-defined functions could result in a similarly general and concise abstraction as the one presented here. Beyond SQL, also other languages are interesting to consider. In the field of statistical analysis and machine learning, the R language is very popular. While not all R language features are easy to integrate with parallelizable functions (for example generic loops), some look like a good match for a data-parallel API.

- The abstraction presented here for iterative algorithms, especially for stateful iterative algorithms is very powerful. Yet, experience has shown that it is not the simplest to use for programmers that implement these algorithms. While it remains a powerful abstraction for the optimizer to reason about iterative algorithms, we plan to build various different APIs (more specialized an easier to use) on top of it. Examples a Pregel-style API, and an API for generic sparse dependency problems. By sitting on top of the workset abstraction, they fit into the optimization framework and can be embedded in PACT programs.

- The distinction between bulk iterations and incremental iterations makes sense for many algorithms and from a system perspective. Ideally, however, the programmer would not have to worry about that distinction. In cases where the program exposes sufficient semantics about the computation, some techniques to automatically incrementalize iterative computations can be applied. It remains an open question in how for a system can automatically exploit computational dependencies for more generic UDF program with less semantic knowledge about the computations.

A. A Brief Introduction to Scala

This appendix gives a very brief introduction to the Scala language. It it not designed to teach the reader Scala, but rather help to bridge the Scala syntax towards other well known programming languages, such as Java or C++, in order to make the Scala code samples used throughout this thesis understandable. We assume that the reader is familiar with Java or C++. For a comprehensive introduction to Scala, please refer to the book "Programming in Scala" [83].

The introduction will cover:

- Variables and assignments
- Generics
- Control flow
- Functions, function literals
- Higher-order functions, lambdas
- Object orientation, classes, case classes

Basics

Scala is a *hybrid language*, offering constructs from both *functional programming* and *object oriented programming*. As in other pure object oriented languages, every data type is an object. As in functional programming, functions are objects, can be passed as parameters and be assigned to variables.

Scala's basic type system is similar to that of Java and C++, offering the types *Byte*, *Short*, *Int*, *Long* (8-, 16-, 32, and 64 bit integer numbers), *Float* and *Double* (for 32 bit resp. 64 bit floating point numbers), as well as *Char* and *String* (for single characters resp. character strings) and *Boolean*.

Expressions (e. g., `x+y`, `sum += t`) appear the same way as in Java and C++. Scala allows operator overloading as a special case of functions that have are named like the operator. In fact, the basic operators are all implemented as functions on the built-in types. Scala determines operator precedence and associativity through a set of basic

rules on function names. Expressions need not be terminated with a semicolon, as in Java or C++, as long as no other expression follows in the same line.

Variables and Assignments

Variables are declared following the pattern "`(val|var) <name> :<type> = <value>`". Scala supports two types of variables: Variables with immutable assignments (*val*) and variables with mutable assignments (*var*). In comparison to Java and C++, "var" is like a regular variable, while "val" is like a "const" or "final" variable. The Scala compiler has powerful type inference, such that the variable type can often be omitted.

```
1  val half : Double = 0.5  // type explicitly set to Double
2  val full : Float = 1     // type explicitly set as Float
3
4  val product = half * full  // type inferred as Double
5
6  val x = 1   // type inferred to be Int
7  x += 2      // error, immutable assignment
8
9  var line = "Home sweet "   // type inferred as string
10 line += "Home"             // line is now "Home sweet Home"
11
12 val array: Array[String] = new Array(1)  // create string array of size 1
13 array(0) = "some string"    // contents changed, but not variable assignment
```

Generic Data Types & Tuples

Scala supports *generic data types*. These data types are specialized towards a type when they are instantiated. A simple example is the `Array` type, which is specialized towards the specific type of element that it holds. Generic parameters are given in square brackets [ ]. Generic types can also be inferred by the compiler in many cases. If types are mixed, the compiler uses the most specific common ancestor in the type hierarchy, which may be the root of the type hierarchy, the `Any` type (comparable to `Object` in Java, but also the super-class of value types like `Int` and `Boolean`).

```
1  val a : Array[String] = new Array("abc", "def", "ghi")
2  val b : = new Array(2, 3, 4, 5, 6)  // type inferred as Array[Int]
3  val c : = new Array(1, 0.2, 0.3f)   // type inferred as Array[Double]
4
5  val list = List("abc", 1, true)     // type is List[Any]
```

A special generic type in Scala are *tuples*. Tuples are n-ary data types with possibly different types at each position. This contrasts them to arrays, which are sequences of the same data type. Tuples are first class language citizens and are defined in round

```
val t = (1, "blue", true)       // type is tuple (Int, String, Boolean)
val e : (Double, Long) = (5, 1)   // explicitly typed tuple

val s : String = t._2      // get the second field of tuple t
val v = e._1               // value of field 1 in tuple e

val list : List[(Int, Double)] = List( (1, 0.5), (2, 3) ) // list of tuples
```

parentheses: `(a, b, c, ...)`. The fields tuples are named _1, _2, _3, ... until _n, where n is the arity of the tuple.

Similar to C++, Scala's generics support annotations of *covariance* and *contravariance*, to tell the compiler whether super-classes or subclasses are valid substitutes for a generic type. For details, we refer the reader to [83].

Control Flow Constructs

Scala supports the usual control flow constructs, like `if ...then ...else ...`, `while` loops, and `for` loops. Their syntax is given below by example. In Scala, all code blocks return a value, including those in conditional expressions and loops. The returned value is always the last statement in the block; if the last statement is a function, it is the function's return type. That way, the if-then-else expressions can be used to assign values to variables (similar to the ternary conditional operator `<condition> ? <true> : <false>` in Java and C++). The return type may be `Unit` (which is the type returned by functions with no return type, similar to `void` in Java and C++).

```
val configFile = if (configFile.exists()) {
  configFile.getAbsPath()
} else {
  configFile.createNewFile()
}

if (x > 0) println("positive")
  else println("negative")     // return type if statement is Unit

while (it.hasNext) foo(it.next)

for (i <- 1 to 4) println(i)                // prints 1, 2, 3, 4
for (j <- List.range(1, 10)) println(j * 2) // prints 2, 4, 6, ...
```

Functions

Functions in Scala are defined following the pattern **`def`** `<name> (<parameters>) : <return-type> = <body>`. The parameters are a list of variable declarations. Because of Scala's sophisticated type inference, the return type may be omitted in many cases. Functions may return values, or `Unit()`, which is returned when the function

has no actual return type. As such, `Unit()` is used in situations where Java and C++ would use the `void` keyword.

```
def square(x: Double) : Double = x * x
square(3)    // returns 9

// this function has an inferred return type
def sumOfSquares(x: Double, y: Double) = square(x) + square(y)
sumOfSquares(2, 3)    // returns 13

def abs(x: Double) = if (x >= 0) x else -x

// a function with return type Unit
def echo(msg: String) = println("Hello " + msg)

def timesTwo(i: Int): Int = {
  println("hello world")
  i * 2
}
```

In Scala, functions defined in classes need not be invoked with the dot notation `type.function()`. In most cases, the dot can be omitted. Similarly, if a function takes no parameters, the parentheses can be omitted. Functions without parameters hence appear like immutable variables. This is called the *uniform access pattern*.

```
// both statements are equivalent
println(System.currentTimeMillis())

println(System currentTimeMillis)
```

Functions are objects in Scala and can be assigned to variables. The type of the variable is `<parameter types> => <return type>`, which denotes a function that turns the given parameters into the return type. This is illustrated by the following code snippet:

```
def square(x: Double) = x * x

def cube(x: Double) = x * x * x

// define variable of function from Double to Double
var op : Double => Double = null

// assign the op function variable
op = square
op(4)       // returns 16

// reassign the op variable
op = cube
op(4)       // returns 64
```

In some cases, it is not necessary to define a function explicitly with a name, because its only use is that it will be assigned to a variable. This is analogous to declaring an anonymous subclass in Java, which happens when the only purpose of the class is to be instantiated once and being assigned to a variable. The declaration of anonymous

functions follows the function literal syntax `(<parameters>) => <body>`. Note that the parameter list may be empty.

```
// variable with explicit type function Int to Int and anonymous
// function body
val times2 : Int => Int = (x: Int) => 2*x
times2(5)                          // returns 10

// variable with inferred type "Int => Int"
val times3 = (x: Int) => 3*x
times3(5)                          // returns 15

// function with explicitly type (no parameters) to Long.
val time : () => Long = () => System.currentTimeMillis
```

Higher-Order Functions

Higher-order functions are functions that take other functions as arguments. Functions whose arguments are not functions themselves are called first-order functions. Functions whose arguments include first-order functions are called second-order functions, and so on. The functions passed as arguments may be regular named functions, variables with functions assigned to them, or directly inline defined anonymous functions. The latter are often called *lambdas*. The following code snippet illustrates that at the example of the foreach function, defined on lists. This second-order function invokes the given function once for every element and supplies the element to that function. It expects a function with one argument which is of the type of the list's elements. If a function is the only parameter, it may be wrapped in curly braces (like a code block) instead of round parentheses.

```
// standalone function
def printElement(x: Int) = println("element " + x)

// anonymous function
val printer = (x: Int) => println("element " + x)

val lst = List(1, 2, 3)

// all three statements do the same
lst.foreach(printElement)
lst.foreach(printer)
lst.foreach(x => println("element " + x))
lst.foreach { x => println("element " + x) }
// prints:
// element 1
// element 2
// element 3

// this does not compile, as the foreach function expects a
// function with one parameter
lst.foreach((x,y) => println("element " + x))

// this does not compile, as the foreach function expects a
// function from Int to Unit
lst.foreach((x: String) => println("element " + x))
```

Second-order functions are particularly common on the processing of collections. Among the most common second order functions are:

- *filter*: Each element is tested whether it passes a boolean condition.
- *map*: Each element of the collection is individually transformed.
- *flatMap*: Each element of the collection is individually transformed. If the result is multiple elements, the resulting collection is flattened.
- *reduceLeft*/*reduceRight*: Incrementally combines two elements of the list until the list has only a single element.

```
1  val lst = List(1, 2, 3, 4, 5)
2
3  lst.filter(x => x < 3)    // result is List(1, 2)
4  lst.map(x => x + 1)       // result is List(2, 3, 4, 5, 6)
5  lst.reduceLeft((x,y) => x + y)  // result is 15
6
7  val strings = List("abc", "def")
8
9  strings.map(x => x.toCharArray())
10 // --> returns List([a,b,c],[d,e,f])
11
12 strings.flatMap(x => x.toCharArray())
13 // --> returns List(a,b,c,d,e,f)
```

For function literals, scala has a shortcut notation for the parameters passed to the functions. The underscore character '_' acts as a placeholder for the parameters. The first underscore takes the place of the first parameter, the second underscore the place of the second parameter, and so on. Using the underscore notation, the function literal needs not specify a parameter list any more. The statements from the previous example can be rewritten as the following:

```
1  val lst = List(1, 2, 3, 4, 5)
2
3  lst.filter(_ < 3)    // result is List(1, 2)
4  lst.map(_ + 1)       // result is List(2, 3, 4, 5, 6)
5  lst.reduceLeft(_ + _)  // result is 15
6
7  val strings = List("abc", "def")
8
9  strings.map(_.toCharArray())
10 // --> returns List([a,b,c],[d,e,f])
11
12 strings.flatMap(_.toCharArray())
13 // --> returns List(a,b,c,d,e,f)
```

As mentioned earlier, Scala's functions come with syntactic flavors that allow programmers to omit dots and replacing parentheses. When chaining function calls, this gives the impression of a scripting language, more than a functional or object oriented language.

```scala
val lst = List((1, "abc"),(2, "def"),(3, "ghi"))

// both statements are equivalent
lst.filter(x => x._1 < 3).map(x => x._2)

lst filter { _._1 < 3 } map { _._2 }
```

Object Orientation & Classes

The basic mechanics of *object orientation* in Scala are very similar to those in Java. For simplicity, we will not discuss Scala's concepts of *Traits* and *mix-in inheritance* here. As far as the examples in this thesis are concerned, the most important difference between Scala and Java is that Scala's syntax is much more concise. The following examples illustrates that. In addition Scala features so called *case classes*. These work very similar to regular classes, but come with extra utility methods and can be used for *pattern matching* (not covered here).

```scala
// simple class with two immutable fields in SCALA
class Point2D (val x: Int, val y: Int)

// simple class with two immutable fields in JAVA
public class Point2D {
  private int x, y;

  public Point2D(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public void x() {
    return x;
  }

  public void y() {
    return y;
  }
}
```

B. Additional Code Samples

This chapter of the appendix lists additional code samples in different languages.

Word Histogram Building

```
1  public class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
2
3    private final IntWritable one = new IntWritable(1);
4    private Text word = new Text();
5
6    public void map(LongWritable key, Text value, Context context) {
7      String line = value.toString();
8      StringTokenizer tokenizer = new StringTokenizer(line);
9      while (tokenizer.hasMoreTokens()) {
10       word.set(tokenizer.nextToken());
11       context.write(word, one);
12     }
13   }
14 }
15
16 public class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
17
18   public void reduce(Text key, Iterable<IntWritable> values, Context context) {
19     int sum = 0;
20     for (IntWritable val : values) {
21       sum += val.get();
22     }
23     context.write(key, new IntWritable(sum));
24   }
25 }
26
27 public class WordCount {
28   public static void main(String[] args) {
29     Configuration conf = new Configuration();
30     Job job = new Job(conf, "wordcount");
31
32     job.setOutputKeyClass(Text.class);
33     job.setOutputValueClass(IntWritable.class);
34
35     job.setMapperClass(Map.class);
36     job.setReducerClass(Reduce.class);
37
38     job.setInputFormatClass(TextInputFormat.class);
39     job.setOutputFormatClass(TextOutputFormat.class);
40
41     FileInputFormat.addInputPath(job, ...);
42     FileOutputFormat.setOutputPath(job, ...);
43     job.waitForCompletion(true);
44   }
45 }
```

Listing B.1: A program computing a simple word histogram, implemented in the Hadoop MapReduce API [106].

Relational Query - Style of TPC-H [100] Query 3

```
1    SELECT l_orderkey, o_shippriority, sum(l_extendedprice)
2      FROM orders, lineitem
3     WHERE l_orderkey = o_orderkey
4       AND o_orderstatus = "..."
5       AND YEAR(o_orderdate) > ...
6       AND o_orderpriority = "..."
7  GROUP BY l_orderkey, o_shippriority;
```

Listing B.2: A relational select/project/join/aggregate query in the style of the TPC-H Query 3 [100], expressed in SQL.

```
1  class TPCHQuery3(status: String, year: Int, prio: String) extends PactProgram {
2
3    val orders = new DataSource[Order]("hdfs://...")
4    val lineitem = new DataSource[LineItem]("hdfs://...")
5    val output = new DataSink[PrioritizedOrder]("hdfs://...")
6
7    val filtered = orders filter { o => o.status == status &&
8      o.date.substring(0, 4).toInt > year && o.orderPriority.startsWith(prio) }
9
10   val joined = filtered join lineitem on { _.id } isEqualTo { _.id }
11     using { (o, li) => PrioritizedOrder(o.id, o.shipPrio, li.extendedPrice) }
12
13   val result = joined groupBy { o => (o.id, o.shipPrio) } combine { sum, _.revenue }
14
15   override def outputs = output <~ result
16 }
17
18 case class Order(id: Long, status: String, date: String, prio: String, shipPrio: String)
19 case class LineItem(id: Long, extendedPrice: Double)
20 case class PrioritizedOrder(id: Long, shipPrio: String, revenue: Double)
```

Listing B.3: A relational select/project/join/aggregate query expressed using the functional API of Chapter 5. The query corresponds to the SQL statement in Listing B.2.

```
@ConstantFieldsAll
public class FilterO extends MapStub {

  private String prioFilter;  // filter literal for the order priority
  private int yearFilter;     // filter literal for the year

  public void open(Configuration param) {
    this.yearFilter = param.getInteger("YEAR_FILTER", 1990);
    this.prioFilter = param.getString("PRIO_FILTER", "0");
  }

  public void map(PactRecord record, Collector<PactRecord> out) {
    PactString status = record.getField(1, PactString.class);
    PactString prio = record.getField(2, PactString.class);
    PactString date = record.getField(3, PactString.class);

    if ( status.getValue().equals("F") &&
         prio.getValue().startsWith(prioFilter) &&
         parseInt(date.getValue().substring(0,4)) > yearFilter)
    {
      out.collect(record);
    }
  }
}

@ConstantFieldsFirstAll
public class JoinLiO extends MatchStub {

  public void match(PactRecord o, PactRecord l, Collector<PactRecord> c) {
    o.setField(5, l.getField(1, PactDouble.class));
    c.collect(o);
  }
}

@Combinable
public class AggLiO extends ReduceStub {

  public void reduce(Iterator<PactRecord> values, Collector<PactRecord> c) {
    PactRecord rec = null;
    double partExtendedPriceSum = 0;
      while (values.hasNext()) {
      rec = values.next();
      partExtendedPriceSum += rec.getField(5, PactDouble.class).getValue();
    }
    rec.setField(5, new PactDouble(partExtendedPriceSum));
    out.collect(rec);
  }
}

public class TPCHQuery3 implements PlanAssembler {

  public Plan getPlan(final String... args) {
    FileSource orders = new FileSource(RecordInput.class, "hdfs://...");
    orders.schema(...); // read relevant attributes into fields 0 to 4

    FileSource lineitem = new FileSource(RecordInput.class, "hdfs://...");
    lineitem.schema(...); // read relevant attributes into fields 0 to 1

    MapContract filterO = new MapContract(FilterO.class, orders);
    filterO.setParameter("YEAR_FILTER", 1993);
    filterO.setParameter("PRIO_FILTER", "5");

    MatchContract joinLiO = new MatchContract(JoinLiO.class,
        PactLong.class, 0, 0, filterO, lineitem);

    ReduceContract aggLiO = ReduceContract.builder(AggLiO.class)
      .keyField(PactLong.class, 0).keyField(PactString.class, 4)
      .input(joinLiO).build();

    FileSink result = new FileSink(RecordOutput.class, "hdfs://...", aggLiO);
    result.schema(...); // write fields 0, 4, 5

    return new Plan(result, "TPCH Q3");
  }
}
```

Listing B.4: A relational select/project/join/aggregate query, expressed using the Java PACT API. The query corresponds to the SQL statement in Listing B.2.

K-Means Clustering

```
1  // case class for the type of a point
2  case class Point(x: Double, y: Double, z: Double) {
3
4    def this() = this(0, 0, 0);  // auxiliary constructor
5
6    def computeEuclidianDistance(other: Point) = {
7      val Point(x2, y2, z2) = other;
8      math.sqrt(math.pow(x - x2, 2) + math.pow(y - y2, 2) + math.pow(z - z2, 2))
9    }
10
11   def +(that: Point) = Point(x + that.x, y + that.x, z + that.z)
12
13   def /(div: Int) = Point(x/div, y/div, z/div)
14 }
15
16 // case class for the type of a point with a distance to a centroid
17 case class Distance(dataPoint: Point, clusterId: Int, distance: Double)
18
19 // program composition
20 def getPlan(pointPath: String, clusterPath: String, output: String, iters: Int) = {
21   val dataPoints = DataSource(pointsPath, CsvInputFormat[(Int, Point)]())
22   val clusterPoints = DataSource(clusterInput, CsvInputFormat[(Int, Point)]())
23
24   // define the step function
25   def computeNewCenters(centers: DataSet[(Int, Point)]) = {
26
27     val distances = dataPoints cross centers map { case ((pid, point), (cid, cent)) =>
28                     (pid, Distance(point, cid, point.computeEuclidianDistance(cent))) }
29
30     val nearestCenters = distances groupBy { case (pid, _) => pid }
31                                    reduceGroup { ds => ds.minBy(_._2.distance) }
32                                    map { case (_, Distance(p, cid, _)) => (cid, p, 1) }
33
34     val newCenters = nearestCenters groupBy { _._1 } reduceGroup { x =>
35         val (id, point, count) = x.reduce( (a,b) => (a._1, a._2+b._2, a._3+b._3) )
36         (id, point / count)
37     }
38
39     newCenters
40   }
41
42   // invoke the iteration
43   val finalCenters = clusterPoints.iterate(numIterations, computeNewCenters)
44   val result = finalCenters.write(clusterOutput, CsvOutputFormat())
45
46   override def outputs = output <~ result
47 }
```

Listing B.5: A *K-Means Clustering Algorithm* implemented in the functional API of Chapter 5.

Bibliography

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR*, pages 277–289, 2005.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] Stefan Ackermann, Vojin Jovanovic, Tiark Rompf, and Martin Odersky. Jet: An Embedded DSL for High Performance Big Data Processing. In *International Workshop on End-to-end Management of Big Data*, 2012.

[4] Loredana Afanasiev, Torsten Grust, Maarten Marx, Jan Rittinger, and Jens Teubner. Recursion in XQuery: put your distributivity safety belt on. In *EDBT*, volume 360 of *ACM International Conference Proceeding Series*, pages 345–356. ACM, 2009.

[5] Foto N. Afrati, Vinayak R. Borkar, Michael J. Carey, Neoklis Polyzotis, and Jeffrey D. Ullman. Map-reduce Extensions and Recursive Queries. In *EDBT*, pages 1–8, 2011.

[6] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Princiles, Techniques, and Tools*. Addison-Wesley, 1986.

[7] Alexander Alexandrov, Dominic Battré, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. Massively Parallel Data Analysis with PACTs on Nephele. *PVLDB*, 3(2):1625–1628, 2010.

[8] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The Stratosphere Platform for Big Data Analytics. In *VLDB Journal (to appear)*. VLDB Endowment, 2014.

[9] Alexander Alexandrov, Stephan Ewen, Max Heimel, Fabian Hueske, Odej Kao, Volker Markl, Erik Nijkamp, and Daniel Warneke. MapReduce and PACT - Comparing Data Parallel Programming Models. In *BTW*, volume 180 of *LNI*, pages 25–44. GI, 2011.

[10] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *POPL*, pages 1–11. ACM Press, 1988.

[11] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[12] Apache Avro. http://avro.apache.org/.

[13] Apache Giraph. http://incubator.apache.org/giraph/.

[14] Apache Mahout. http://mahout.apache.org.

[15] Apache Oozie. http://oozie.apache.org/.

[16] Apache Thrift. http://thrift.apache.org/.

[17] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas, Rohit Varma, and Jennifer Widom. STREAM: The Stanford Stream Data Manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.

[18] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N. Gray, Patricia P. Griffiths, W. Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. System R: Relational Approach to Database Management. *ACM Transactions on Database Systems (TODS)*, 1(2):97–137, 1976.

[19] Monya Baker. Next-generation Sequencing: Adjusting to Data Overload. *Nature Methods*, 7(7):495–499, 2010.

[20] Maarten Ballintijn, Rene Brun, Fons Rademakers, and Gunther Roland. The proof aistributed parallel analysis framework based on root. *arXiv preprint physics/0306110*, 2003.

[21] François Bancilhon and Raghu Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *SIGMOD Conference*, pages 16–52. ACM Press, 1986.

[22] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In *Symposium on Cloud Computing*, 2010.

[23] Jacek Becla, Andrew Hanushevsky, Sergei Nikolaev, Ghaleb Abdulla, Alexander S. Szalay, María A. Nieto-Santisteban, Ani Thakar, and Jim Gray. Designing a Multi-petabyte Database for LSST. *CoRR*, abs/cs/0604112, 2006.

[24] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. In *PODS*, pages 269–284, 1987.

[25] Alexander Behm, Vinayak R. Borkar, Michael J. Carey, Raman Grover, Chen Li, Nicola Onose, Rares Vernica, Alin Deutsch, Yannis Papakonstantinou, and Vassilis J. Tsotras. ASTERIX: Towards a Scalable, Semistructured Data Platform for Evolving-world Models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[26] Kevin S. Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Y. Eltabakh, Carl-Christian Kanne, Fatma Özcan, and Eugene J. Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. *PVLDB*, 4(12):1272–1283, 2011.

[27] Mark Beyer. Gartner Says Solving 'Big Data' Challenge Involves More Than Just Managing Volumes of Data. `http://www.gartner.com/newsroom/id/1731916`, 2011.

[28] Richard J. Bolton and David J. Hand. Statistical Fraud Detection: A Review. *Statistical Science*, pages 235–249, 2002.

[29] Charles Bontempo and George Zagelow. The IBM Data Warehouse Architecture. *Communications of the ACM*, 41(9):38–48, 1998.

[30] Vinayak R. Borkar and Michael J. Carey. A Common Compiler Framework for Big Data Languages: Motivation, Opportunities, and Benefits. *IEEE Data Eng. Bull.*, 36(1):56–64, 2013.

[31] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*, pages 1151–1162, 2011.

[32] Kevin J. Brown, Arvind K. Sujeeth, HyoukJoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A Heterogeneous Parallel Framework for Domain-Specific Languages. In *PACT*, pages 89–100. IEEE Computer Society, 2011.

[33] Yingyi Bu, Vinayak R. Borkar, Michael J. Carey, Joshua Rosen, Neoklis Polyzotis, Tyson Condie, Markus Weimer, and Raghu Ramakrishnan. Scaling Datalog for Machine Learning on Big Data. *CoRR*, abs/1203.0160, 2012.

[34] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters. *PVLDB*, 3(1):285–296, 2010.

[35] Eugene Burmako. Scala macros: Let our powers combine! Technical report, EPFL, 2013.

[36] Jiazhen Cai and Robert Paige. Program derivation by fixed point computation. *Sci. Comput. Program.*, 11(3):197–261, 1989.

[37] Cascading. URL: http://www.cascading.org/.

[38] CERN Computing Report. `http://public.web.cern.ch/public/en/LHC/Computing-en.html`.

[39] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. *PVLDB*, 1(2):1265–1276, 2008.

[40] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel Haskell: A Status Report. In *DAMP*, pages 10–18. ACM, 2007.

[41] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 363–375. ACM, 2010.

[42] Edgar F Codd. A Relational Model of Data for Large Shared Data Banks. In *Pioneers and Their Contributions to Software Engineering*, pages 61–98. Springer, 2001.

[43] Michael J. Corey and Michael Abbey. *Oracle Data Warehousing*. Osborne/McGraw-Hill, 1996.

[44] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[45] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. GAMMA - A High Performance Dataflow Database Machine. In *VLDB*, pages 228–237, 1986.

[46] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: A Runtime for Iterative MapReduce. In *HPDC*, pages 810–818, 2010.

[47] Stephan Ewen, Sebastian Schelter, Kostas Tzoumas, Daniel Warneke, and Volker Markl. Iterative parallel data processing with stratosphere: an inside look. In *SIGMOD Conference*, pages 1053–1056. ACM, 2013.

[48] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. Spinning Fast Iterative Data Flows. *PVLDB*, 5(11):1268–1279, 2012.

[49] Leonidas Fegaras, Chengkai Li, and Upa Gupta. An Optimization Framework for Map-Reduce Queries. In *Proceedings of the 15th International Conference on Extending Database Technology*, pages 26–37. ACM, 2012.

[50] Andrzej Filinski. Monads in Action. *ACM Sigplan Notices*, 45(1):483–494, 2010.

[51] Eric Friedman, Peter M. Pawlowski, and John Cieslewicz. SQL/MapReduce: A Practical Approach to Self-describing, Polymorphic, and Parallelizable User-defined Functions. *Proceedings of the VLDB Endowment*, 2(2):1402–1413, 2009.

[52] Shinya Fushimi, Masaru Kitsuregawa, and Hidehiko Tanaka. An Overview of the System Software of a Parallel Relational Database Machine GRACE. In *VLDB*, pages 209–219, 1986.

[53] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[54] George Giorgidze, Torsten Grust, Tom Schreiber, and Jeroen Weijers. Haskell Boards the Ferry - Database-Supported Program Execution for Haskell. In *IFL*, volume 6647 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2010.

[55] George Giorgidze, Torsten Grust, Nils Schweinsberg, and Jeroen Weijers. Bringing back Monad Comprehensions. In *Haskell*, pages 13–22. ACM, 2011.

[56] Google Protocol Buffers. https://code.google.com/p/protobuf/.

[57] Goetz Graefe. Implementing Sorting in Database Systems. *ACM Comput. Surv.*, 38(3), 2006.

[58] Goetz Graefe and William J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*, pages 209–218. IEEE Computer Society, 1993.

[59] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *CoRR*, abs/cs/0701155, 2007.

[60] Ulrich Güntzer, Werner Kießling, and Rudolf Bayer. On the Evaluation of Recursion in (Deductive) Database Systems by Efficient Differential Fixpoint Iteration. In *ICDE*, pages 120–129, 1987.

[61] Joseph J. Harjung. *Reducing Formal Noise in PACT Programs.* Thesis for the degree of Diploma in Computer Science, TU Berlin, 2013.

[62] Arvid Heise, Astrid Rheinländer, Marcus Leich, Ulf Leser, and Felix Naumann. Meteor/Sopremo: An Extensible Query Language and Operator Model. In *Workshop on End-to-end Management of Big Data (BigData)*, 2012.

[63] Fabian Hueske, Mathias Peters, Matthias Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. Opening the Black Boxes in Data Flow Optimization. *PVLDB*, 5(11):1256–1267, 2012.

[64] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, pages 59–72, 2007.

[65] Java HotSpot VM Whitepaper. http://www.oracle.com/technetwork/java/whitepaper-135217.html.

[66] Vasiliki Kalavri, Vladimir Vlassov, and Per Brand. PonIC: Using Stratosphere to Speed Up Pig Analytics. In *Euro-Par*, volume 8097 of *Lecture Notes in Computer Science*, pages 279–290. Springer, 2013.

[67] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: A High-performance, Distributed Main Memory Transaction Processing System. *PVLDB*, 1(2):1496–1499, 2008.

[68] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7:305–317, 1977.

[69] Sepandar Kamvar, Taher Haveliwala, and Gene Golub. Adaptive Methods for the Computation of PageRank. Technical Report 2003-26, Stanford InfoLab, April 2003.

[70] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system. In *ICDM*, pages 229–238, 2009.

[71] Lars Kolb, Andreas Thor, and Erhard Rahm. Multi-pass Sorted Neighborhood Blocking with MapReduce. *Computer Science - R&D*, 27(1):45–63, 2012.

[72] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix Factorization Techniques for Recommender Systems. *Computer*, 42(8):30–37, 2009.

[73] Kryo Serialization Library. URL: https://code.google.com/p/kryo/.

[74] T.-H. Lai, Y.-C. Tseng, and X. Dong. A more efficient message-optimal algorithm for distributed termination detection. In *Parallel Processing Symposium, 1992. Proceedings., Sixth International*, pages 646 –649, 1992.

[75] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[76] Jacob Leverich and Christos Kozyrakis. On the Energy (in)efficiency of Hadoop Clusters. *Operating Systems Review*, 44(1):61–65, 2010.

[77] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. *CoRR*, abs/1006.4990, 2010.

[78] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, 2010.

[79] Nathan Marz. Storm: Distributed and Fault-tolerant Realtime Computation. http://storm-project.net/.

[80] Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. Scala-virtualized. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, pages 117–120. ACM, 2012.

[81] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: A Universal Execution Engine for Distributed Data-flow Computing. *NSDI, Boston, MA*, 2011.

[82] Solomon Negash. Business Intelligence. *Communications of the Association for Information Systems*, 13(1):177–195, 2004.

[83] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2008.

[84] Christopher Olston, Benjamin Reed, Adam Silberstein, and Utkarsh Srivastava. Automatic Optimization of Parallel Dataflow Programs. In *USENIX Annual Technical Conference*, pages 267–273, 2008.

[85] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD Conference*, pages 1099–1110, 2008.

[86] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[87] Paolo Boldi and Sebastiano Vigna. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[88] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *SIGMOD Conference*, pages 165–178, 2009.

[89] Paolo Pialorsi and Marco Russo. *Introducing Microsoft® LINQ*. Microsoft Press, 2007.

[90] Tiark Rompf and Martin Odersky. Lightweight Modular Staging: A pragmatic Approach to Runtime Code Generation and Compiled DSLs. *Commun. ACM*, 55(6):121–130, 2012.

[91] Tiark Rompf, Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Building-Blocks for Performance Oriented DSLs. In *DSL*, volume 66, pages 93–117, 2011.

[92] Scalding, a Scala-based Analysis Language on top of Hadoop. `http://scalding.org`, 2012.

[93] Sebastian Schelter, Stephan Ewen, Kostas Tzoumas, and Volker Markl. "All roads lead to Rome": Optimistic Recovery for Distributed Iterative Data Processing. In *CIKM*, pages 1919–1928. ACM, 2013.

[94] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD Conference*, pages 23–34, 1979.

[95] Michael Stonebraker and Judy Robertson. Big Data is 'Buzzword du Dour;' CS Academics 'Have the Best Job'. *Commun. ACM*, 56(9):10–11, 2013.

[96] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, and Others. C-Store: A Column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 553–564. VLDB Endowment, 2005.

[97] The Stratosphere Project. http://stratosphere.eu/.

[98] Val Tannen, Peter Buneman, and Limsoon Wong. Naturally Embedded Query Languages. In *ICDT*, volume 646 of *Lecture Notes in Computer Science*, pages 140–154. Springer, 1992.

[99] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB*, 2(2):1626–1629, 2009.

[100] TPC-H. URL: http://www.tpc.org/tpch/.

[101] Leslie G. Valiant. General purpose parallel architectures. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 943–972, 1990.

[102] Rares Vernica, Michael J. Carey, and Chen Li. Efficient Parallel Set-similarity Joins using MapReduce. In *SIGMOD '10: Proceedings of the 2010 International Conference on Management of Data*, pages 495–506, New York, NY, USA, 2010. ACM.

[103] Daniel Warneke and Odej Kao. Nephele: Efficient Parallel Data Processing in the Cloud. In *SC-MTAGS*, 2009.

[104] Markus Weimer, Tyson Condie, and Raghu Ramakrishnan. Machine learning in ScalOps, a higher order cloud computing language. In *NIPS 2011 Workshop on parallel and large-scale machine learning (BigLearn)*, December 2011.

[105] Ryen W. White, Nicholas P. Tatonetti, Nigam H. Shah, Russ B. Altman, and Eric Horvitz. Web-Scale Pharmacovigilance: Listening to Signals from the Crowd. *Journal of the American Medical Informatics Association (JAMIA)*, 2013.

[106] Tom White. *Hadoop - The Definitive Guide: Storage and Analysis at Internet Scale (3. ed., revised and updated).* O'Reilly, 2012.

[107] Christian Wimmer and Thomas Würthinger. Truffle: A Self-Optimizing Runtime System. In *SPLASH*, pages 13–14. ACM, 2012.

[108] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *SIGMOD Conference*, pages 13–24. ACM, 2013.

[109] Yu Xu, Pekka Kostamaa, Xin Zhou, and Liang Chen. Handling data skew in parallel joins in shared-nothing systems. In *SIGMOD Conference*, pages 1043–1052. ACM, 2008.

[110] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and Douglas Stott Parker. Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In *SIGMOD Conference*, pages 1029–1040, 2007.

[111] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI*, pages 1–14, 2008.

[112] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[113] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP*, pages 423–438. ACM, 2013.

[114] Jingren Zhou, Nicolas Bruno, Ming-Chuan Wu, Per-Ake Larson, Ronnie Chaiken, and Darren Shakib. SCOPE: Parallel Databases meet MapReduce. *The VLDB JournalThe International Journal on Very Large Data Bases*, 21(5):611–636, 2012.

[115] Jingren Zhou, Per-Åke Larson, and Ronnie Chaiken. Incorporating partitioning and parallel plans into the scope optimizer. In *ICDE*, pages 1060–1071. IEEE, 2010.

[116] Martin Zinkevich, Markus Weimer, Alexander J. Smola, and Lihong Li. Parallelized stochastic gradient descent. In *NIPS*, pages 2595–2603, 2010.