Hasso Plattner Institute
Information Systems Group

Data Cleansing and Integration Operators for a Parallel Data Analytics Platform

Dissertation
zur Erlangung des akademischen Grades
“Doktor der Ingenieurwissenschaften”
(Dr.-Ing.)
in der Wissenschaftsdisziplin “Informationssysteme”

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

von
Arvid Heise

Potsdam, den 30.09.2014

Published online at the
Institutional Repository of the University of Potsdam:
URN urn:nbn:de:kobv:517-opus4-77100
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-77100

Abstract

The data quality of real-world datasets need to be constantly monitored and maintained to allow organizations and individuals to reliably use their data. Especially, data integration projects suffer from poor initial data quality and as a consequence consume more effort and money. Commercial products and research prototypes for data cleansing and integration help users to improve the quality of individual and combined datasets. They can be divided into either standalone systems or database management system (DBMS) extensions. On the one hand, standalone systems do not interact well with DBMS and require time-consuming data imports and exports. On the other hand, DBMS extensions are often limited by the underlying system and do not cover the full set of data cleansing and integration tasks.

We overcome both limitations by implementing a concise set of five data cleansing and integration operators on the parallel data analytics platform Stratosphere. We define the semantics of the operators, present their parallel implementation, and devise optimization techniques for individual operators and combinations thereof. Users specify declarative queries in our query language METEOR with our new operators to improve the data quality of individual datasets or integrate them to larger datasets. By integrating the data cleansing operators into the higher level language layer of Stratosphere, users can easily combine cleansing operators with operators from other domains, such as information extraction, to complex data flows. Through a generic description of the operators, the Stratosphere optimizer reorders operators even from different domains to find better query plans.

As a case study, we reimplemented a part of the large Open Government Data integration project GovWILD with our new operators and show that our queries run significantly faster than the original GovWILD queries, which rely on relational operators. Evaluation reveals that our operators exhibit good scalability on up to 100 cores, so that even larger inputs can be efficiently processed by scaling out to more machines. Finally, our scripts are considerably shorter than the original GovWILD scripts, which results in better maintainability of the scripts.

Zusammenfassung

Die Datenqualität von Realweltdaten muss ständig überwacht und gewartet werden, damit Organisationen und Individuen ihre Daten verlässlich nutzen können. Besonders Datenintegrationsprojekte leiden unter schlechter Datenqualität in den Quelldaten und benötigen somit mehr Zeit und Geld. Kommerzielle Produkte und Forschungsprototypen helfen Nutzern die Qualität in einzelnen und kombinierten Datensätzen zu verbessern. Die Systeme können in selbständige Systeme und Erweiterungen von bestehenden Datenbankmanagementsystemen (DBMS) unterteilt werden. Auf der einen Seite interagieren selbständige Systeme nicht gut mit DBMS und brauchen zeitaufwändigen Datenimport und -export. Auf der anderen Seite sind die DBMS Erweiterungen häufig durch das unterliegende System limitiert und unterstützen nicht die gesamte Bandbreite an Datenreinigungs- und -integrationsaufgaben.

Wir überwinden beide Limitationen, indem wir eine Menge von häufig benötigten Datenreinigungs- und Datenintegrationsoperatoren direkt in der parallelen Datenanalyseplattform Stratosphere implementieren. Wir definieren die Semantik der Operatoren, präsentieren deren parallele Implementierung und entwickeln Optimierungstechniken für die einzelnen und mehrere Operatoren. Nutzer können deklarative Anfragen in unserer Anfragesprache METEOR mit unseren neuen Operatoren formulieren, um die Datenqualität von einzelnen Datensätzen zu erhöhen, oder um sie zu größeren Datensätzen zu integrieren. Durch die Integration der Operatoren in die Hochsprachenschicht von Stratosphere können Nutzer Datenreinigungsoperatoren einfach mit Operatoren aus anderen Domänen wie Informationsextraktion zu komplexen Datenflüssen kombinieren. Da Stratosphere Operatoren durch generische Beschreibungen in den Optimierer integriert werden, ist es für den Optimierer sogar möglich Operatoren unterschiedlicher Domänen zu vertauschen, um besseren Anfragepläne zu ermitteln.

Für eine Fallstudie haben wir Teile des großen Datenintegrationsprojektes GovWILD auf Stratosphere mit den neuen Operatoren nachimplementiert und zeigen, dass unsere Anfragen signifikant schneller laufen als die originalen GovWILD Anfragen, die sich auf relationale Operatoren verlassen. Die Evaluation zeigt, dass unsere Operatoren gut auf bis zu 100 Kernen skalieren, sodass sogar größere Datensätze effizient verarbeitet werden können, indem die Anfragen auf mehr Maschinen ausgeführt werden. Schließlich sind unsere Skripte erheblich kürzer als die originalen GovWILD Skripte, was in besserer Wartbarkeit unserer Skripte resultiert.

Acknowledgements

I would like to express my deepest appreciation and thanks to my advisor Prof. Dr. Felix Naumann. He supported me all the time with his advices and provided great opportunities to me, such as being part of the Stratosphere project or the visiting stay in Qatar. There could not have been a better advisor for me. However, I am also thankful for the additional guidance of my unofficial co-advisors in the Stratosphere project, especially Prof. Dr. Ulf Leser.

I want to thank all my colleagues at the chair and in the Stratosphere project. Your feedback and ideas helped me immensely towards this thesis. I am grateful for the cooperation with Astrid, Ziawasch, Toni, Jorgé, Thorsten, Tobias, Gjergji, and Dustin. Special thanks to Daniel, Stephan, and Fabian for their dedicated work on the early Stratosphere prototypes – without you there would be no "scalable" in my thesis. And last but not least I very much enjoyed working with Fabian and Tommy – you were a great help.

Most importantly, I am deeply moved by the support of my family and friends. You gave me the strength to complete this thesis. Janine, you are my brightest star – thank you.

Contents

1

The Need for Data Integration Operators

In today's business landscape, data plays an important role – either directly as an asset, most prominently seen in Google, or as the main driver for business decisions. Data is typically collected through several sources and applications, such as customer relations tables, sales reports, or data derived from suppliers. For high tech companies, the acquisition of data is one of the main motivations for buying other companies; for example, when Facebook acquired WhatsApp in 2014 for 19 billion dollars[1].

However, hoarding data does not immediately help an organization. According to the Data Warehouse Institute, poor data quality costs US businesses $600 billion dollars in the early 2000s [Eckerson, 2002]. Therefore, the quality of the data must be constantly monitored and maintained. New datasets must be carefully integrated in the data warehouse of the organization to increase the value of the data and justify the supposedly expensive acquisition.

Despite much research in the area of data cleansing [Bleiholder and Naumann, 2008; Elmagarmid et al., 2007] and integration [Halevy et al., 2006], most of the processes involve a high degree of manual work. The reasons are manifold but can be roughly divided into two categories.

1. The **large variety of data quality constraints** makes it hard to suitably express and enforce them. Current database management systems (DBMS) allow database administrators to express only basic consistency constraints, such as uniqueness, non-null fields, and inclusion dependencies. Consequently, advanced constraints become implicit and their satisfaction needs to be ensured by each application, which makes data quality issues more likely with every new application.

2. The **lack of standards** results in a diverse set of data integration products and often in custom implementations for each cleansing and integration process. Ideally, DBMS would support data cleansing and integration operators out of the box, such

[1] `http://newsroom.fb.com/news/2014/02/facebook-to-acquire-whatsapp/`

that users do not need to use a specialized tool. However, major data management companies may be hesitant to include such functionality for the following reasons:

- Data quality ensuring methods usually decrease the performance and increase the complexity of the DBMS. Consequently, major DBMS provide no proper support for *assertions* and allow only limited *check* constraints, which heavily rely on subqueries.
- The tools need to be flexible enough such that advanced constraints incorporating domain knowledge can be sufficiently expressed. However, vendors have been favoring efficient validation of constraints over the expressiveness in the past and will probably continue in the future.
- Results of data integration are often not well-defined, but commercial products need to guarantee that all relevant information is retained.

Both aforementioned reasons create a hen-egg-problem. A well-accepted, declarative formalization of data quality constraints would surely result in at least partial implementations in DBMS and - vice versa - an implementation in a major DBMS would yield a higher acceptance and usage of such constraints. In the context of assertions, de Haan and Koppelaars [2007, p. 295] believe "that it is not possible for a DBMS vendor to program an algorithm that accepts an arbitrarily complex predicate [...], we should not expect full support for multi-tuple constraints – in a practical, usable and acceptable way – from these vendors in the future. The best we can hope for is that database researchers first come up with more common classes of constraints and develop convenient shorthands for these". Therefore, in contrast to the general purpose assertions, smaller and well-defined constraints should allow vendors to implement them in an easier and more efficient way, as can be seen in referential integrity and domain constraints in SQL.

This thesis aims to overcome the vicious cycle by providing a suite of data cleansing and integration operators with corresponding constraint specifications for the open-source parallel data analytics platform Stratosphere [Alexandrov et al., 2014]. Hereby, we choose simple and scalable operator semantics that improve the data quality in a best effort approach instead of trying to satisfy all constraints with the realistic chance of running into unsatisfiable situations. A full-fledged operator must fulfill several hard and soft requirements: It must be well-defined, useful, efficient and scalable, optimizable, and easily usable. Further, the set of operators must be comprehensive, so that users intuitively choose the combination of operators to solve their tasks.

The remainder of this chapter first presents three general data cleansing and integration scenarios that motivate the scope of this thesis. We then introduce our new data cleansing and integration operators and show how they can be used to solve the introduced scenarios. Afterwards, we present a concrete data integration use case that serves as our running example. We show a query in Stratosphere to address the use case featuring our new operators. Finally, we list the contributions and the structure of this thesis.

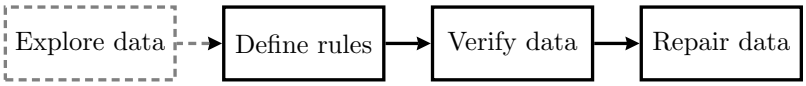


Figure 1.1: Workflow of cleaning one dataset

1.1 Data cleansing and integration scenarios

Because data cleansing and integration can include a wide range of tasks, we first define three common scenarios that we want to address with this thesis.

1.1.1 Cleaning a single dataset

As already stated, current DBMS allow users to enforce only a limited number of constraint types, mostly on schematic level. In an ideal environment, data managers would define all other, advanced constraints in guidelines that are then enforced by applications and/or followed by application users. However, in reality, some constraints are only implicitly specified and applications enforce only a small number of constraints.

Systematic data heterogeneity already exists in one dataset, when several applications and users modify the data. For example, the format of dates, phone numbers, or street name abbreviations may be entered differently by different users. Users from one domain may use abbreviations for common terms in their domain that appear cryptic to other users. *Data conflicts* may arise if records are partially maintained by different departments of a company, such as sales and customer relationship management. Similarly, *duplicate records* may be inserted for the same real world entity if the original record cannot be retrieved properly due to data quality issues.

To address these problems, the data manager may decide to periodically cleanse the dataset in four phases as shown in Figure 1.1. First of all, the data manager needs to explore the data to find data quality issues, possibly with *data profiling* [Naumann, 2013] techniques. Second, he needs to *formalize* all *data quality constraints* that are either known before or inferred from the first step. Third, the quality constraints are applied to the data and *violations* are reported. In the last step, these violations need to be *repaired*.

For this thesis, we aim for an automatic verification and correction process (i.e., for the last two steps). These steps may then be repeated several times without human interaction. In contrast, the first two steps involve a high degree of domain knowledge and are rarely repeated.

In this thesis, we show how the scripting language of Stratosphere offers a high degree of expressiveness to allow users to declaratively specify their *data quality rules*. Through the efficient execution engine, these constraints are efficiently enforced in parallel even on larger datasets. The exploratory phase is out of scope.

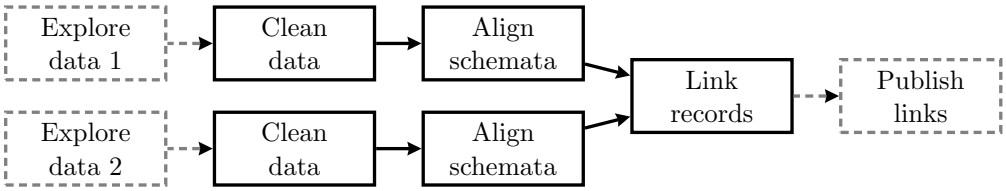


Figure 1.2: Workflow of linking two datasets

1.1.2 Linking several datasets

The wealth of freely available, structured information on the Web is constantly growing. Especially scientists and public administrations are increasingly publishing their data. The former contribute their knowledge to the society while the latter enable organizations and citizens to browse and analyze the data.

However, simply publishing the data on the Web does not guarantee immediate benefits. On the one hand, the large sizes of the datasets render manual inspection of the data futile. On the other hand, the technical, structural, and semantic heterogeneity of the data prevents meaningful automatic processing and analysis. Furthermore, complex data analysis tasks usual need additional background information.

Thus, datasets in the area of *Linked Open Data* benefit immensely from links *across* datasets, namely *owl:sameAs* triples. However, the task of linking two datasets is non-trivial for two reasons. First, data maintainers need to explore the data and find rules that can be *reliably* used to link equivalent concepts. Second, the matching must be performed *efficiently* to avoid a costly, naïve comparison on the Cartesian product of large datasets. Finally, the links must be published.

Figure 1.2 summarizes the process of linking two datasets. With this thesis, we address the matching task, which can be subdivided into preprocessing and optional schema alignment of the individual datasets and the actual record linkage. We tightly integrate our operators into the powerful and extensible query language of Stratosphere and perform the matching of records with parallel variants of well-established techniques that reduce the number of comparisons.

In the context of data integration, we can see this scenario as a virtual integration. The original datasets remain untouched and we create only the links between the records. Any application that wants to use the information of both datasets concerning one linked entity receives potentially conflicting values and needs to resolve these conflicts on its own. Nevertheless, virtual integration has the advantages that the data ownership remains intact and the updates to the data are immediately visible in the integrated view.

1.1.3 Integrating several datasets

The last scenario describes the workflow of a materialized integration. Similar to the previous use case, materialized integration aligns the concepts and the instances of two or more datasets. In contrast to virtual integration, materialized integration produces a

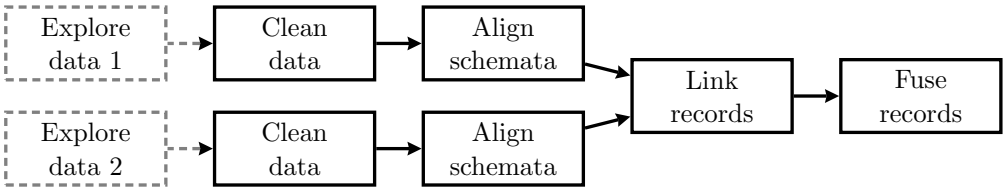


Figure 1.3: Workflow of materialized data integration

consistent dataset as its final result. Schematic heterogeneity and conflicts have already been solved in the process.

Materialized integration ease the consumption of the integrated dataset, because applications may directly use the integrated data as it was a regular dataset. However, the new dataset must be maintained separately to the original sources and thus may become quickly outdated. Hence, materialized data integration usually implies either that the data ownership and data age can be neglected or the integrated dataset replaces the original datasets. For example, in a company acquisition, the data managers perform the integration in a batch process, where they merge the acquired dataset into the existing data warehouse. Afterwards, all applications operate solely on the integrated warehouse.

The workflow of materialized integration in Figure 1.3 strongly resembles the workflow of virtual integration. However, in a final step, the data managers consolidate the matched records and fuse them to a single data entity. Arising data conflicts may be solved by choosing the most plausible value, using master data, or by preferring one dataset over the other. Materialized data integration requires the complete set of our data integration operators. All core tasks are implemented as self-contained and efficient operators in Stratosphere as part of this thesis.

1.2 New data cleansing and integration operators

We briefly introduce our novel data cleansing and integration operators. The set of operators suffices to support common data cleansing and integration projects. All operators can be used in each of the three scenarios, but we can distinguish between optional and mandatory operators with respect to the individual scenarios.

Data Scrubbing The `scrub` operator applies a declaratively specified list of *constraints* and corresponding *repair* methods to a given dataset. Typical constraints include non-null values, pattern conformance, but also functional dependencies. In case of a constraint *violation*, the operator performs the repair if given or filters the records otherwise. The resulting dataset maintains the original schema and confirms to all constraints. An optional second output contains all unresolved violations with the corresponding filtered records. Data maintainers may use `scrub` operator to clean their existing datasets (Scenario 1) or prepare new datasets for virtual and materialized integration (Scenario 2 and 3).

Data Map The `data map` operator transforms n datasets with a list of *schema mappings* into m datasets while maintaining *referential integrity*. The operator is primarily used to align the schemata in integration (Scenario 2 and 3) to overcome schematic heterogeneity. Data maintainers could also use it to restructure their existing datasets (Scenario 1).

Duplicate detection/record Linkage Both operators match multiple representations of the same real-world entity in datasets by using *matching rules* and optionally *candidate selection* techniques. While the `duplicate detection` operator finds duplicate pairs within one dataset, the `record linkage` operator finds matching pairs across datasets with potentially different schema. All three scenarios rely on this operator. While data maintainers may exclude `duplicate detection` from the data cleansing scenario (Scenario 1), the operator can be considered the core activity in virtual and materialized integration.

Duplicate Clustering The set of matching pairs from the `duplicate detection` and `record linkage` operators needs to be transitively closed such that there are no overlapping clusters of records. The `clustering` operator adds and removes edges according to some clustering strategy. The operator is implicitly run with the `duplicate detection` and `record linkage` operator, but users may change the configuration according to their needs. All scenarios implicitly need this operator. For Scenario 2 and Scenario 3 it can be used to guarantee that entities between clean datasets are not being linked to more than one entity of another dataset. On dirty datasets, however, the same operator can detect larger clusters with multiple entries per datasets with a slightly different configuration.

Data Fusion The `fusion` operator ingests one dataset with clusters of records and applies *conflict resolution strategies* to produce a dataset where each cluster of entity representations is consolidated into one record. Similar to the `scrub` operator, data maintainers specify the strategies declaratively. Unresolved conflicts may be filtered and outputted into an optional secondary output. `Fusion` is directly used in the first scenario if duplicates have been detected and in the third scenario after records have been linked. It should also become handy for the consuming application of the virtual integration (Scenario 2).

1.3 Exemplary data integration workflow

In this section, we briefly demonstrate how we perform a materialized integration of Open (Government) Data with our operators. This query serves as our running example throughout the thesis. We want to examine `US Earmarks`[2], personally enacted funds of

[2] `http://earmarks.omb.gov/earmarks-public/`

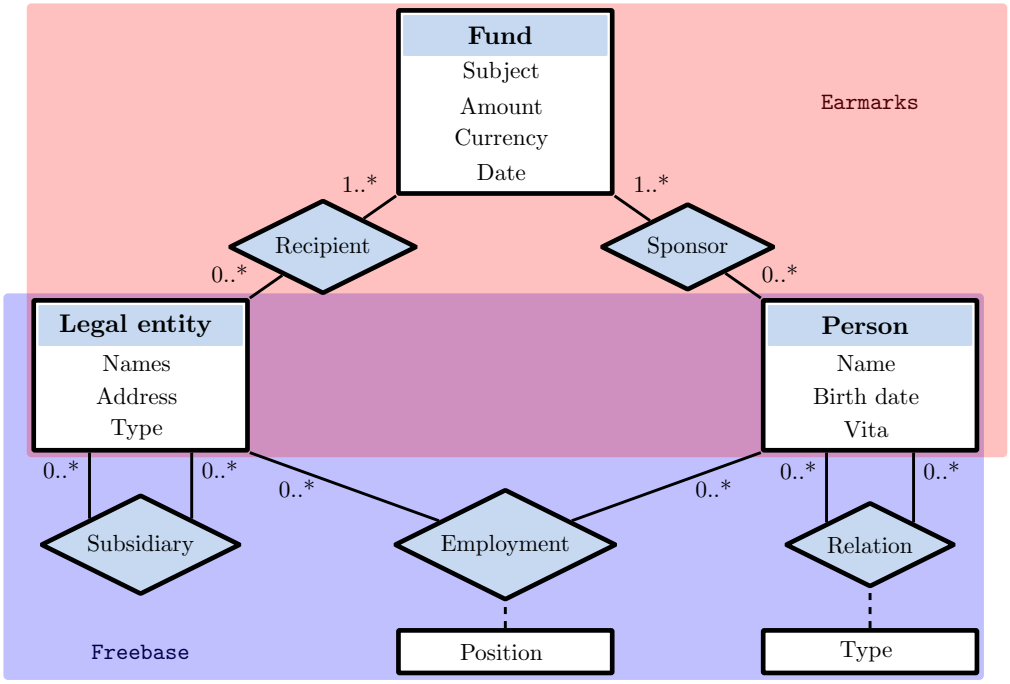


Figure 1.4: Entity-relationship model for the running example.

US Congress members, for potential cases of nepotism. As this dataset contains information only about funds, sponsors, and recipients, we need to add `Freebase`[3] to retrieve family and other relationships between sponsor and recipients, which are indispensable to uncover nepotism.

The brief example already depicts the basic requirements for most integration projects. We have two datasets that *overlap* in at least one type of entity (politician) but each dataset contains unique information about the entities. The overlap may be *intensional* or *extensional* and is usually both. Intensional overlap means that the schemata of both datasets have some attributes in common. Extensional overlap implies that both datasets contain at least some representations of the same real-world entities. Ideally, we have enough intensional overlap to uniquely match all entity representations of the same real-world entities across the datasets, but also many non-overlapping attributes to enrich the matched entities as much as possible.

Figure 1.4 depicts the entity-relationship model for the running example. Both datasets share the concepts person and legal entity. In `US Earmarks` a person is always a politician, whereas `Freebase` contains all kinds of persons and their relationships. A politician in `US Earmarks` may provide funds to a legal entity, which is described through a name and address. `Freebase` contains most of these legal entities and provides additional information, such as its type, employment of top managers, and links to subsidiaries. Together with familiar relationships of the person, we can use these two datasets to find suspicious relationship cycles: A politician enacts a fund to a legal entity or a subsidiary thereof, for which the politician or a family member works.

[3] `http://freebase.com/`

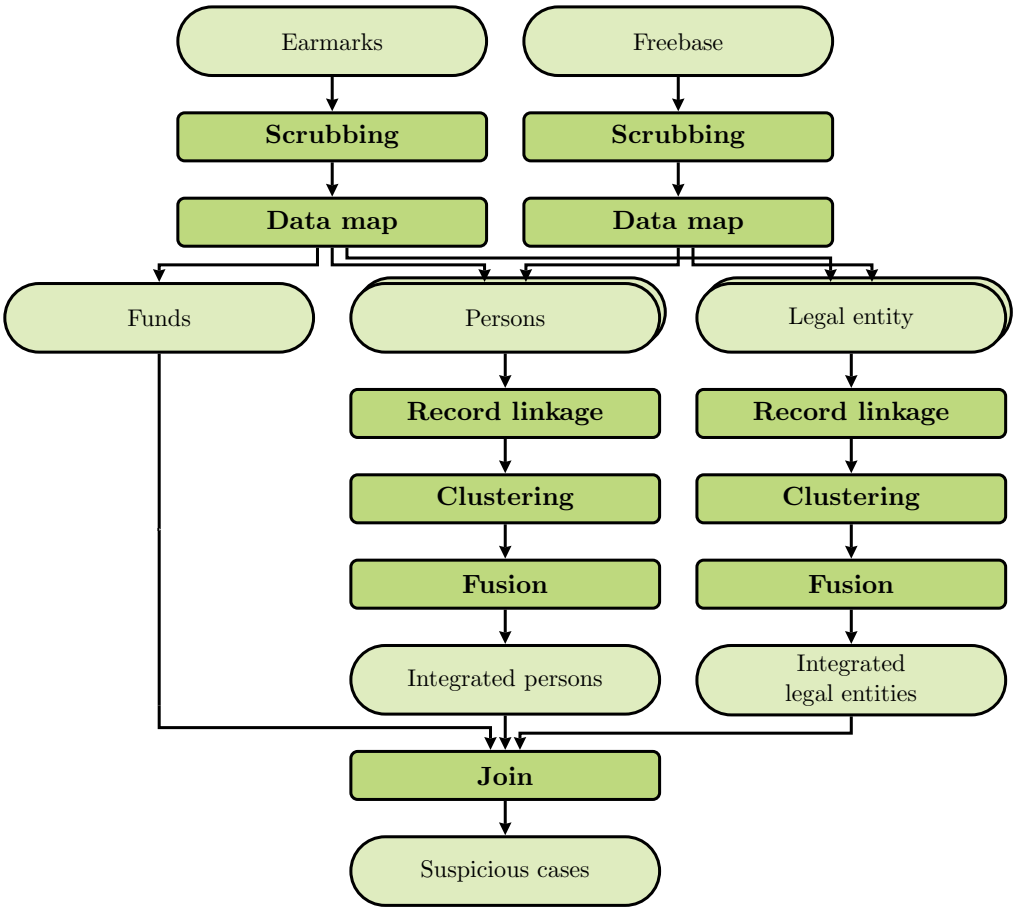


Figure 1.5: Stratosphere data integration query for potential cases of nepotism. The final circular join checks if sponsor and recipient share a relationship.

With the integrated data model in mind, we can now formulate a Stratosphere query using our data integration operators for ad-hoc integration of the datasets with subsequent analysis. At this point of the thesis, we briefly discuss the Stratosphere operator plan of the query Figure 1.5 from top to bottom and defer details to subsequent chapters.

We first address individual data quality issues of the datasets with the `scrub` operator. Then, we map the entities of each dataset to the target schema. We extract funds, persons, and legal entities from `US Earmarks` as well as persons and legal entities from `Freebase`.

In the next steps, we connect both datasets. In particular, we need the respective records about persons and legal entities from both datasets and consolidate the record clusters. Subsequently, we fuse the records that describe the same real-world entity and update the relationships accordingly. As a result, we receive a consistent, homogeneous dataset about persons and legal entities.

Finally, we can perform a (complex) join over the three consistent relations about funds, persons, and legal entities to find potential cases of nepotism. Throughout this

thesis, we reference to this example and closely discuss the individual operators as well as the corresponding Stratosphere query.

Advantages of declarative specification

Obviously, the query of Figure 1.5 corresponds directly to the workflow of data integration. A relational query using only standard operators would be much harder to read and write, because it would necessarily reveal implementation details, such as the blocking strategy of the duplicate detection task. Further, some operators, especially the clustering, require relation functions, which break pipelines and holistic optimizations. Our operators allow users to declaratively describe how they want to integrate their data and require little knowledge about the execution strategies. Apart from increased usability, we also aim for shorter and more concise queries that result in better maintainability.

A declarative description further offers additional optimization potential to the framework. In a relational query, for example, record linkage may be implemented as a join over some blocking key. Further, `data map` usually implies a grouping operator per entity and dataset and can easily result in 20 relational operators for this short example. With our operators, we can leave the choice of the implementation to the Stratosphere optimizer. Because of semantically rich annotation, we have additional optimization possibilities, as we show throughout this thesis. It is, however, important to note that Stratosphere is not tailor towards data integration; the system may be used for standard, relational queries, but also for advanced text and web extraction tasks and many more applications in the future.

1.4 Structure and contributions

In this chapter, we have motivated the need for data cleansing and integration operators. We have defined the scope of the thesis with three scenarios and as a first contribution derived atomic operators that we have combined to an exemplary, complex data integration query in the area of Open Government Data.

Chapter 2 introduces the Stratosphere framework with a strong focus on the high-level operator layer and the query language, which have been contributed to the project as part of this thesis [Heise et al., 2012]. We discuss important design decisions that are necessary to understand the extensibility and optimizability of Stratosphere.

In the next five chapters, we present each of the *five data cleansing operators* in the usual order of execution during a data integration query: `scrub`, `data map`, `duplicate detection/record linkage`, `clustering`, and `fusion`. For each operator, we examine four dimensions needed for a proper database operator. First, we define *semantics* of the operator and derive properties. Second, we propose a *declarative specification* and configuration of the operator in Stratosphere. Third, we describe our *parallel implementations* on Stratosphere. Fourth, we derive *optimization rules* and statistics, with which the Stratosphere optimizer can holistically optimize queries with our operators.

As a case study, we reimplemented a part of the large Open Government Data integration project GovWILD [Böhm et al., 2012b] with our new operators in Chapter 8. We show that our queries run significantly faster than the original GovWILD queries, which rely on relational operators. Further, our operators exhibit good scalability on up to 100 cores, so that even larger inputs can be efficiently processed by scaling out to more machines.

In Chapter 9, we discuss related work, especially declarative data cleansing systems, optimizable data cleansing operators, and parallel data integration workflows on Map/Reduce. We summarize this thesis in Chapter 10 and conclude with an outlook on future work.

2

The Stratosphere Parallel Data Analytics Platform

In this chapter, we briefly introduce the Stratosphere platform for parallel data analysis with a strong focus on the higher level language layer. Because we provide the prototypical implementations of our data integration operators as a Stratosphere package, fundamental knowledge about Stratosphere helps to understand design decisions and implementation details.

Stratosphere is a joint project of the Technical University Berlin, Humboldt University Berlin, and the Hasso-Plattner Institute in Potsdam and aims to marry well-known (parallel) database technologies with the flexibility and scalability of Map/Reduce [Dean and Ghemawat, 2008]. The research prototype is freely available at Github[1], while a stable product is currently forked into an Apache Incubator project[2].

In comparison to state-of-the-art Map/Reduce systems, Stratosphere supports complex data flows of extended Map/Reduce primitives instead of simple cascades of Map/Reduce jobs. Before execution, Stratosphere uses existing and new techniques to logically and physically optimize such data flows. Moreover, cycles in the data flow allow developers to express iterative algorithms, such as machine learning algorithms, making Stratosphere one of the most powerful data analytic platforms to date.

This chapter first gives an overview over the overall architecture of Stratosphere, then discusses the main data flow engine and the higher level language in more detail. It builds upon the Stratosphere overview and the higher level language paper [Alexandrov et al., 2014; Heise et al., 2012] but with the nomenclature and description matching the open source version of Stratosphere (0.4 and upwards).

2.1 The Stratosphere architecture

The Stratosphere framework consists of four layers as shown in Figure 2.1. Lower layer components may be used autonomously, while higher layer components depend on lower

[1] `https://github.com/stratosphere/`

[2] `https://github.com/apache/incubator-flink`

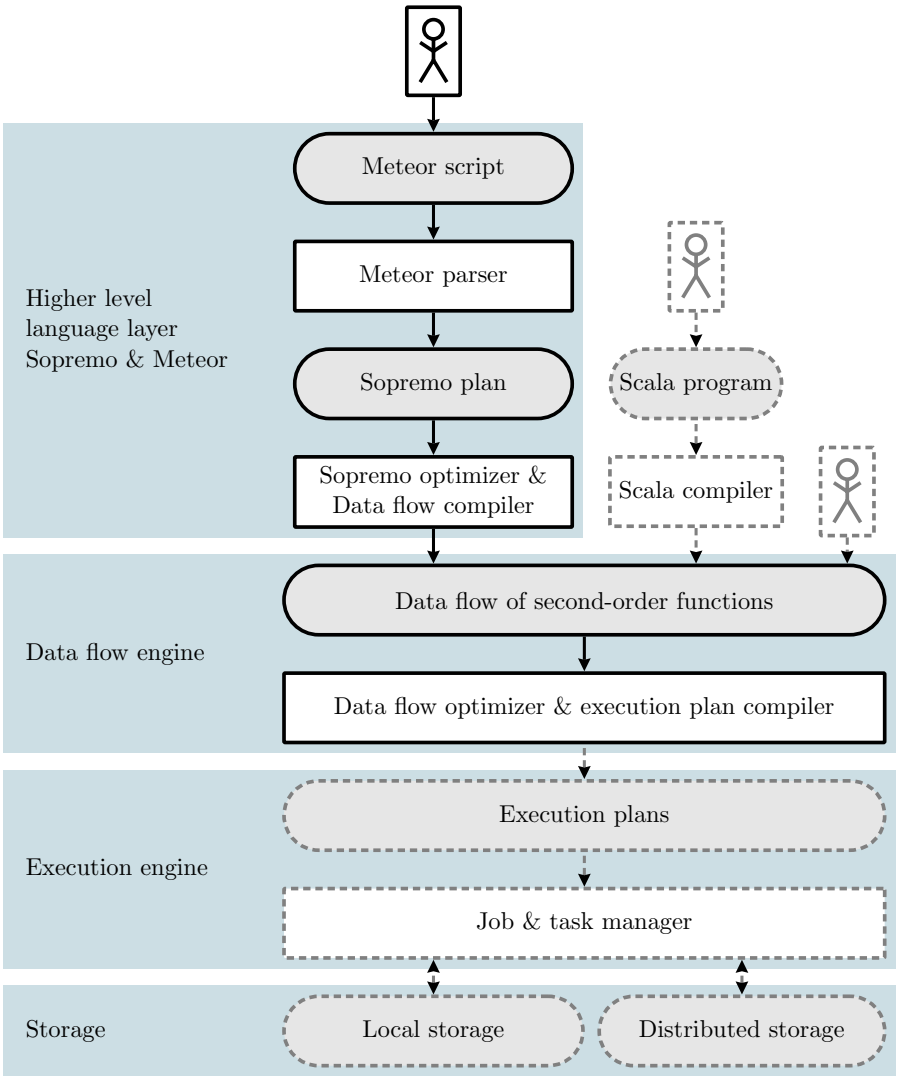


Figure 2.1: High level architecture of Stratosphere. Components in dashed lines are out of the scope of this thesis.

layers. The **storage layer** on the bottom is from the view point of Stratosphere mostly passive. Stratosphere programs may read and write flat files from local and distributed storage. We rely on the *Hadoop Distributed File System* [Shafer et al., 2010] (HDFS), which guarantees a high fail safety through autonomous replication of file blocks.

On the next layer resides the heart of Stratosphere: A flexible **execution engine** that performs the programs in parallel (formerly known as Nephele [Warneke and Kao, 2009]). The execution engine is divided into a *job manager* on a master server and *task managers* on the slaves. The job manager assigns *execution tasks* to each task manager, monitors their states, and restarts or reassigns tasks in cases of failures. The program is defined as an *execution graph*, which describes the intra- and inter-node parallelism of each task and the communication pattern that connects the outputs of one task on one node to the input of another task on the same or a different node.

Stratosphere compiles such execution graphs from **data flows** on the third layer (formerly known as PACT [Alexandrov et al., 2011]). Data flows consist of interconnected *second-order functions*, data sinks, and data sources. Each second-order functions provide a basic data parallelization pattern, which is implemented as one or more *execution tasks*. These tasks partition the input data from one or two data streams by certain criteria and apply a *user-defined, first-order function* (UDF) to the partitioned data. Stratosphere supports the well-known `map` and `reduce` functions of Map/Reduce but provides even more expressive functions for two inputs.

In the Hadoop[3] ecosystem, most jobs are actually written in higher level languages, such as Pig Latin [Olston et al., 2008], Hive [Thusoo et al., 2009], or Jaql [Beyer et al., 2011]. For example, over 70% of the Hadoop jobs at Yahoo are compiled from higher level languages [Elmeleegy, 2013]. Higher level languages allow novice users to formulate their queries without in-depth knowledge and even experts often trade some performance degradation for a more concise, less cluttered scripting code.

Consequently, Stratosphere provides two additional entry points to write data flows. First, users may write Meteor scripts that are compiled into an operator graph in the **higher level language layer**, logically optimized, and finally translated into a data flow. Second, the Scala interface interprets Scala programs, identifies parallelizable and sequential parts, and separates code into first- and second-order functions to automatically generate data flows. In a similar way, the Ferry project translates parts of the application code into SQL queries [Grust et al., 2009].

While the data flow and storage engine has been developed by other project members, I have been the lead developer of the higher level language layer with the Sopremo operator model and the Meteor query language in the context of this thesis [Heise et al., 2012]. Our data cleansing and integration operators are implemented in the Sopremo operator model and accessible in the Meteor query language, whereas the parallel implementation of each operator is a data flow. Hence, we focus on these two layers of the system in subsequent sections. First, we describe the data flow layer in the next section, mostly from a user's perspective with little technical depth. Then, we discuss the higher level language level and the scripting language Meteor in more detail.

2.2 Data flow engine

Stratosphere extends the classic Map/Reduce programming model in three ways. These extensions allow Stratosphere to apply sophisticated *optimization* techniques – including reordering – to the data flows.

1. The data model generalizes from the rigid key/value model to a multikey/value model (see Appendix A.1 on Page 161).
2. Beside `map` and `reduce`, Stratosphere provides three additional, more expressive second-order functions `cross`, `join`, and `cogroup`.
3. These second-order functions can be composed to complex data flow graphs even with cycles.

[3] `www.hadoop.org`

2.2.1 Advanced second-order functions

The original Map/Reduce system was primarily designed for advanced web log analysis. Thus, while scalable, the expressiveness of `map` and `reduce` functions quickly reaches its limits when dealing with more complex data flows and multiple inputs. Advanced data analytics tasks require join-like functions, which need to be simulated either `map`- or `reduce`-side [Blanas et al., 2010]. However, they usually work main memory-based and thus users need to either accept scalability limitations or implement sophisticated partitioning – a task that the framework should actually solve.

Stratosphere provides three additional second-order functions with two inputs to allow users express complex data flow with ease. These functions may be easily used to implement all known join variations over two inputs, but can also be flexibly used for more sophisticated tasks, such as duplicate detection. To understand the semantics of the new second-order functions, we review the definition of the `map` and `reduce` primitives from a slightly different angle. For us, second-order functions define data dependencies between records, such that the framework can identify which records need to be processed together at the same node and which records may be processed independently in parallel.

Definition 2.1 (Second-order function)**.** A *second-order function* f partitions the input data $\mathcal{D}$ into non-overlapping parallelization units $\mathcal{P}_\mathcal{D}$ and applies the user-defined, first-order function u to each partition:

$$f\colon \mathcal{D} \mapsto \bigcup_{p \in \mathcal{P}_\mathcal{D}} u(p)$$

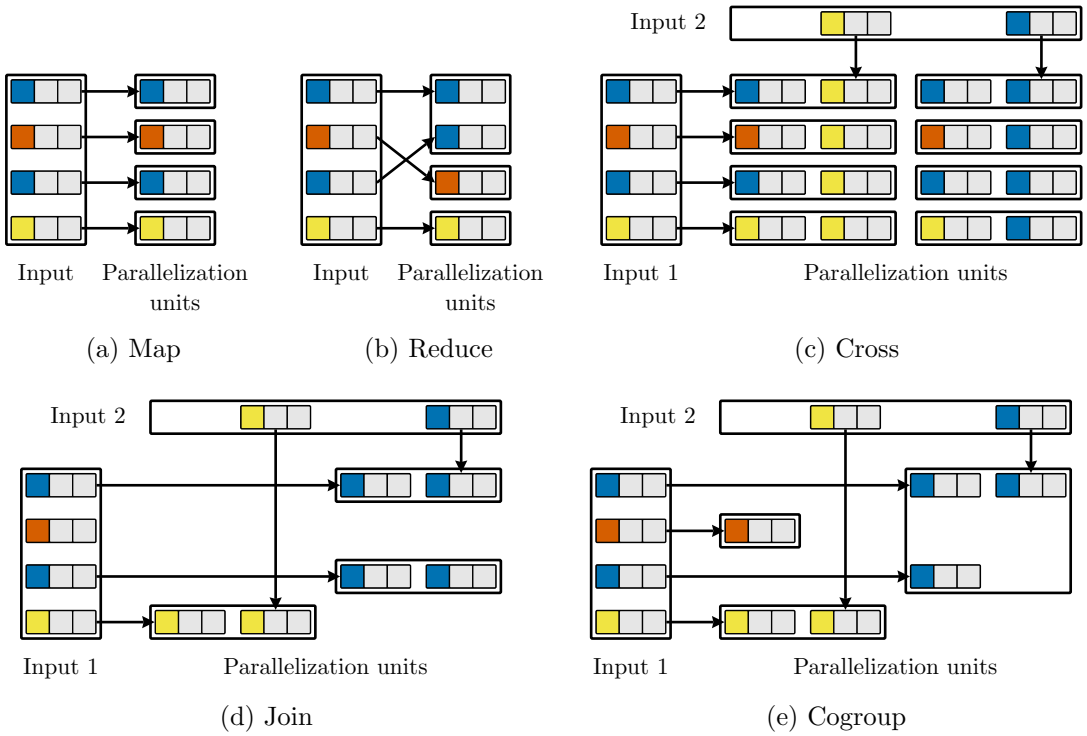
All five second-order functions are depicted in Figure 2.2. `Map` functions constitute the easiest possible case. Each record may be processed independently and thus each record defines its own parallelization unit (see Figure 2.2a). `Reduce` functions guarantee that the UDF is called exactly once for all records with the same key (see Figure 2.2b). The guarantee implies that all records with one key must reside on the same computation node. Therefore, only records with a different key may be processed in parallel.

The new function `cross` creates the Cartesian product of the inputs, where each pair of records is processed independently (see Figure 2.2c). Because of the large number of resulting UDF invocations and the costly data distribution, we recommend using crosses to replicate small intermediate results such as tuple counts or other global aggregations.

For equi-joins, the `join` function should always be preferred to the `cross` (see Figure 2.2d). `Join` is a special case of a `cross` that calls the UDF for each pair of records with equal keys. We recommend using `join` functions whenever possible instead of the other two, more costly two-input functions. Further, physical optimization profits from many different parallel implementations.

Lastly, Stratosphere provides the `cogroup` function (see Figure 2.2e). Cogroup invokes the UDF for all records of both inputs with same key. It generalizes the reduce functions and allows the UDF to distinguish between the input sources.

Developers may add *annotations* to each of the five second-order functions to specify additional constraints or give the system optimization hints. Most importantly, the



(a) Map (b) Reduce (c) Cross

(d) Join (e) Cogroup

Figure 2.2: The five second-order functions currently implemented in Stratosphere. Parallelization units show data dependencies and equal keys are illustrated with same colors. (Source: Adapted from [Alexandrov et al., 2011])

developer may annotate the (composite) keys for each input for `reduce`, `join`, and `cogroup`. The keys determine the parallelization behavior of the three functions, such that records with equal keys form a parallelization unit and are processed on the same node with one invocation of the UDF.

Further, `reduce` and `cogroup` may be annotated to be combinable for associative aggregations. Combinable UDFs produce the correct result when invoked multiple times on the same parallelization unit with partial results. They are primarily used to pre-aggregate results on local sub-partitions and decrease the network traffic. For example, when grouping the records of `US Earmarks` for one person, we can start to collect the earmarks locally for one partition and thus avoid duplicate personal information to be transferred over the network.

Key-less `reducers` constitute a special case, which perform a *global reduce*. Because all records have the same, empty key, they belong to the same parallelization unit and are jointly processed. In that way, developers can collect global statistics, such as the record count or the minimum value of a field. Nevertheless, global reduces should be used only in conjunction with combinable UDFs. Otherwise, all records need to be shipped to a single node. A combinable key-less reduce, however, pre-computes the statistics on the local partitions and transfers only the intermediate result to the node that performs the final aggregation.

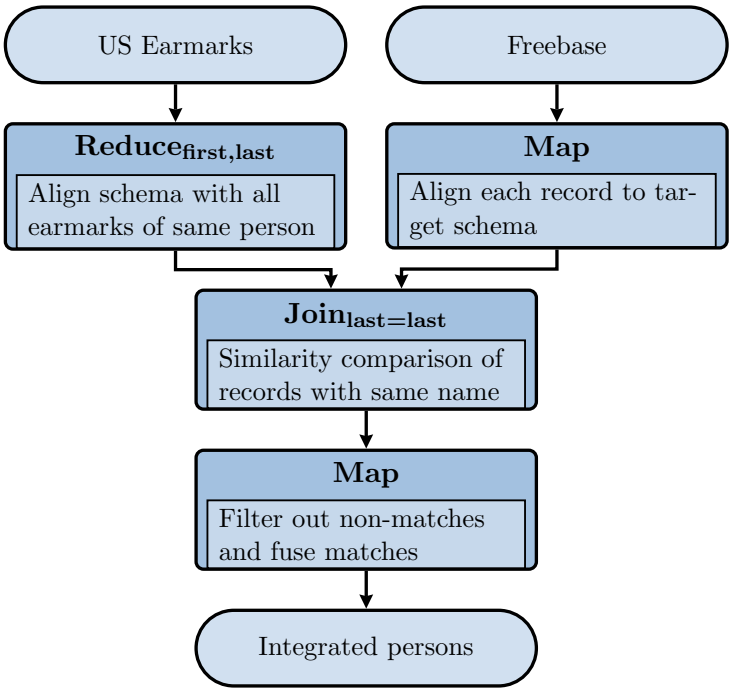


Figure 2.3: Data flow to integrate persons in `US Earmarks` and `Freebase`

2.2.2 Complex data flow

Users may compose second-order functions to large data flows, similar to scientific workflows. In Stratosphere, a data flow is a *directed graph*, where each node is a data source, data sink, or second-order function and each edge defines an output-input relationship. Figure 2.3 illustrates a data flow that is inspired by our running example and performs a hands-on integration of persons in `US Earmarks` and `Freebase`. At the top we see the sources `US Earmarks` and `Freebase`. A `reduce` groups the earmarks by the first and last name of the enacting politician to align the schemata to our global schema. `Freebase` is analogously aligned in a `map`, because there is only one record per politician.

Next, a `join` on the last name invokes a similarity comparison of potential matches. This `join` wraps both tuples in an array and adds the similarity value. A subsequent `map` compares the similarity value with a given threshold, filters out non-matches, and fuses the records. Finally, the output is written in the data sink `Integrated persons`. For this example, we use only a simple data flow tree, which suffices in many cases. Nevertheless, most of our data cleansing and integration operators require directed acyclic graphs, because the output from one function is consumed by several successive functions.

Further, our clustering operator additionally uses *iterations* to compute the clusters. Iterations feed the output of second-order functions back to previous functions to refine the results. Therefore, the directed graph of the data flow contains *limited cycles*, where the cyclic edges have special annotation for limiting the number of iterations or describing the convergence of the result. We defer the detailed discussion of the semantics of cycles to the introduction of the clustering operator in Chapter 6.

2.2.3 Data flow optimizer and compiler

User-defined data flows as well as generated data flows from Sopremo or Scala are all translated into execution graphs in the same way: First, all second-order functions that have not been annotated by the user or by the higher level language compilers are analyzed through static code analysis [Hueske et al., 2013a]. Second, the optimizer processes the plan as further described below. Third, the compiler assembles an execution graph while determining the degree of parallelization for each execution vertex.

The optimizer translates the data flow into an internal graph representation where all data manipulations, such as sorting, hashing, and probing, are explicitly modeled as nodes. For each node, the compiler estimates the cost and denotes the interesting properties to explore possible reorderings [Hueske et al., 2012, 2013b]. The optimizer then chooses the best plan and fixes the data partitioning strategies. Further, it determines the best data shipment strategies, as well as the sort orders on sender and recipient sides of a data flow. Lastly, redundant tasks, such as double sorting, are eliminated.

2.3 Higher level language layer

The higher level language layer eases the use of Stratosphere for common and even advanced tasks. Especially for relational queries, users prefer to not implement the same operators in Java over and over again. In the Hadoop ecosystem, several scripting language alleviate this problem, such as Pig Latin [Olston et al., 2008], Hive [Thusoo et al., 2009], or Jaql [Beyer et al., 2011]. However, each scripting languages implements their own operator model, which isolates them. Consequently, basic operators need to be implemented for each framework and global optimization is nearly impossible. Some scripting languages offer advanced operations that are not available in the other scripting language. For example, Jaql specializes in manipulating semi-structured data, while Pig Latin offers approximate string matching.

With the *Stratosphere operator and data model* (Sopremo), we want to provide a common base for several scripting languages that create exchangeable operator plans. Sopremo offers a *flexible data model* suited for processing unstructured, semi-structured, and structured data. Further, we designed Sopremo to be *extensible* in several ways. Specialized packages provide *domain-specific operators*, functions, data types, and file formats, which can be easily mixed to create advanced data analytic queries.

Users create Sopremo plans through the *Meteor* query language, which is inspired by Jaql [Beyer et al., 2011] but more expressive. Meteor can be seen as a textual interface for Sopremo and Meteor scripts are translated one-to-one into Sopremo plans. Nevertheless, Meteor offers syntactic sugar for common data manipulations.

Both, Sopremo and Meteor, are major contributions of this thesis to the Stratosphere project. Therefore, we discuss the architecture of the higher level language layer, the query language Meteor, and the operator model Sopremo in more detail in the following two sections. Both sections are based on a prior publication [Heise et al., 2012].

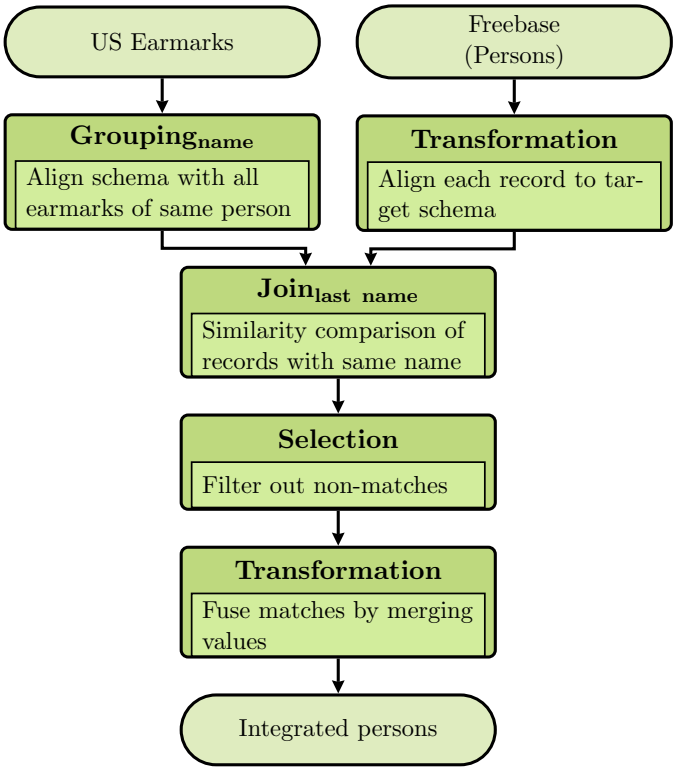


Figure 2.4: Relational query to integrate persons in `US Earmarks` and `Freebase`

For illustration purposes, we use the relational query in Figure 2.4. It performs the same hands-on integration of persons as Figure 2.3 but with the extended relational operators of Sopremo. The schema alignment and data scrubbing of `US Earmarks` requires a `grouping` operator, because the same politician may have enacted several funds and we want to retain all information about one person. In `Freebase`, only one record represents one person and thus a `transformation` (an extended projection) suffices to align the schema.

Next, we perform basic *blocking* with an `equi-join` over the last name. Of course, we could use more sophisticated blocking functions, such as phonetic blocking, but the structure of the query would remain the same. The following `selection` applies any matching rules to the candidate pair and filters out all non-matches. A final `transformation` fuses all matching pairs into one consistent representation. The result is then finally stored in a dataset. At that point, we have retained only the matching persons, which should be sufficient for this example.

2.3.1 Architecture of the higher level language layer

The higher level language layer as depicted in Figure 2.5 currently offers two entry points. End users formulate their analytical queries as Meteor scripts and submit them to the Meteor command line frontend. The command line client invokes the script parser to create a Sopremo plan. After a successful creation of the plan, it is sent to Sopremo. For

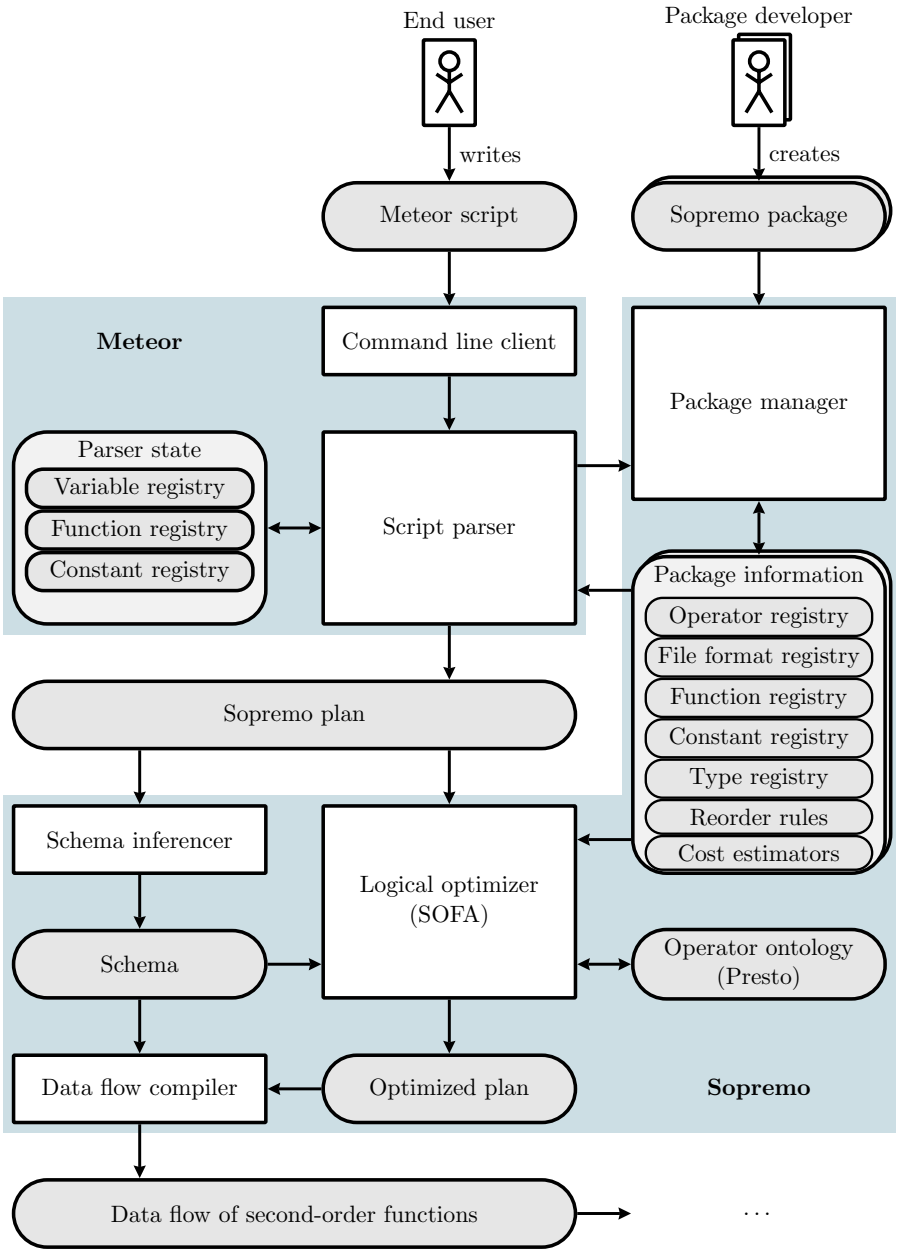


Figure 2.5: Architecture of the higher level language layer.

their queries, users can use basic, relational operators, and advanced operations from domain-specific packages.

Package developers implement some set of operations and provide them as packages for other users. For testing purposes, developers may also programmatically create Sopremo plans (not depicted in the architecture). Through a specific keyword in the Meteor scripts, users can request Sopremo to search for the specified packages and discover the provided functionality. The functionality may include new operators, functions, types, constants, or file formats along with information for the optimizer. The script parser

then dynamically uses the additional registered elements to parse the script and to create a Sopremo plan.

When receiving the plan, Sopremo first analyzes it with a schema inferencer. The inferencer collects all accesses from operators to named attributes to propose a first data layout. It distinguishes between data accesses used for data partitioning and manipulations inside a data partition. At the same time, the Sopremo optimizer extends the built-in operator ontology *Presto* by adding the package-specific optimization rules, operator properties, and registers the domain-specific cost estimators. With the built-in and domain-specific cost models, the optimizer greedily finds its best plan [Rheinländer et al., 2013].

Lastly, the data flow compiler translates the optimized plan to a data flow of second-order functions. At this point, the expressions that generate the keys are annotated on the data flow, to create an optimal data serialization. Sopremo sends the data flow to the data flow engine for further processing. In the following sections, we discuss the extensible syntax of Meteor and demonstrate the translation of Meteor queries into Sopremo plan. We defer in-depth description of the available relational operators and expressions to Section 2.4.

2.3.2 The Meteor query language

The Meteor scripting language treats relational and domain-specific operators as first-class citizens. With the help of the package manager of Sopremo, Meteor allows users to freely combine existing operators and extend the language and runtime functionality with new operators. The syntactical elements resemble Json to easily manipulate Sopremo's semi-structured data model that extends Json. This approach is inspired by Jaql [Beyer et al., 2011], however, we simplified many language features to provide mechanisms for a seamless integration of new operators and to support n-ary input and output operators to unleash the full power of the Sopremo operator model.

In Listing 2.1, we show the Meteor script that generated the relational Sopremo plan (see Figure 2.4). A Meteor script consists of a sequence of operator invocations, where the outputs can be assigned to variables. The script starts by adding all operators, functions, types, and constants from the base- and cleansing-package to the current script scope with the "**using**" keyword. In the script, we are using only operators from the base-package with a similarity functions defined in the cleansing-package. Note that Meteor automatically loads the base package and we included it for demonstration purposes only.

In Line 2, the script reads the `US Earmarks` and `Freebase` from CSV files and stores the result in "$earmarks" and "$freebase" respectively. These variables are conceptually arrays of Json objects and may now be referenced in other operators. In Line 4 a "**group**" operator aggregates all earmark records with the same first and last name. We bind all grouped earmarks to the array variable "$e" and can use this variable inside the operator specification. The grouping key is defined as an array, which is transparently translated into a composite key in Sopremo and later appropriately annotated to the `reduce` function. The "*into*" clause specifies the resulting transformation expression and its syntax simulates a Json object.

```
1  using base, cleansing;

2  $earmarks = read csv from 'earmarks.csv' separator ';';
3  $freebase = read csv from 'freebase.csv';

4  $ePoliticians = group $e in $earmarks by [$e.first, $e.last]
5    into {
6      first: $e[0].first,
7      last: $e[0].last,
8      earmarks: $e
9    };
10 $fPoliticians = transform $f in $freebase
11   into {
12     first: $f.firstName,
13     last: $f.lastName,
14     relationships: [{
15       type: 'spouse',
16       first, last: split($f.spouse, ' ')
17     }]
18   };

19 $candidates = join $e in $ePoliticians, $f in $fPoliticians
20   where $e.last == $f.last
21   into [$e, $f, jaroWinkler(e, f)];
22 $matches = filter $c in $candidates
23   where $c[2] > 0.9;

24 $fused = transform $m in $matches
25   into {
26     $m[0].*,
27     $m[1].relationships
28   };
29 write json $fused to 'integratedPersons.json';
```

Listing 2.1: Meteor query for the relational integration of Figure 2.4.

The "**transform**" operator in Line 10 transforms the records individually. We bind each record in "`$f`" and create a new record where we rename the first and last name and create an array of relationships. The array contains only one object with the type "`'spouse'`" and the corresponding first and last name of the spouse.

The successive "**join**" operator calculates the JaroWinkler similarity on all pairs of records with matching last name and returns an array of the two records with the similarity value. Meteor operators may also return primitive values aside from objects and arrays. Line 22 then filters all pairs with a similarity over 0.9. The following "**transform**" operator fuses both records by copying all values from the first element, and adding all relationships from the second record. Lastly, the "**write**" operator outputs all fused records into a Json file.

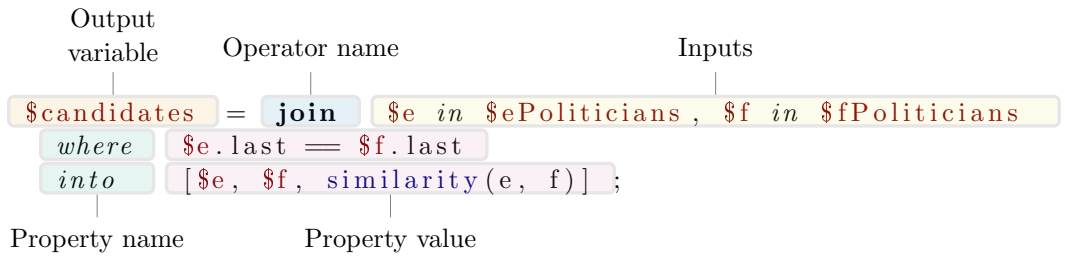


Figure 2.6: Syntax of join broken down to general operator syntax of Meteor.

2.3.3 Operator syntax

Until now, we considered only relational operators. The same query intent can be easily formulated in other query languages, such as Jaql or Pig. The real power of Meteor, however, comes from its extensibility and the generic operator specification syntax.

Figure 2.6 explains the generic syntax with the join operator in Listing 2.1. We assign the output to the variable "$candidates" and start the invocation with the operator name "**join**". In the input list, we bind both inputs to new names. Next, we configure two properties. First, we specify the join condition through the "*where*" property. The property type must be a Boolean expression, which Meteor ensures for us. Second, we specify the output transformation through the "*into*" property.

All operators including the (relational) base operators follow this generic operator syntax `name+ inputs? property+`. Only the read and write operator have a slightly different syntax, because a read operator cannot have any input and the write operator does not generate an output. The generic syntax is the key feature of Meteor to support extensibility. The general Meteor syntax can be found in Appendix A.2 on Page 162. We provide the syntax of specific base operators and expressions in Section 2.4.

2.3.4 Sopremo plan compilation

To conclude this section, we review the translation process of the Meteor script to the Sopremo plan as depicted in Figure 2.7. We truncated the individual lines of the Meteor script by showing only the operator invocation and omit the full specification. The formal algorithm can be found in Appendix A.3 on Page 163.

Every operator invocation in Meteor results in an operator node in the Sopremo plan. Read and write invocations result in data sources and sinks respectively. The parser uses the variable names to resolve the data flow between the operators. Output variables result in an outgoing edge for each successive operator that consumes the variable. There, variables do not specify actual data but define data flow. Consequently, variables that are never consumed are removed. If no output of an operator is consumed, it is also removed. Further, Meteor scripts do not directly specify an execution order. In our example, the `transformation` and the `grouping` are executed in parallel and the `transformation` probably finishes first, although we wrote it after the `grouping` operator in the script. The flexible execution order allows Stratosphere to optimize the queries similar to SQL databases.

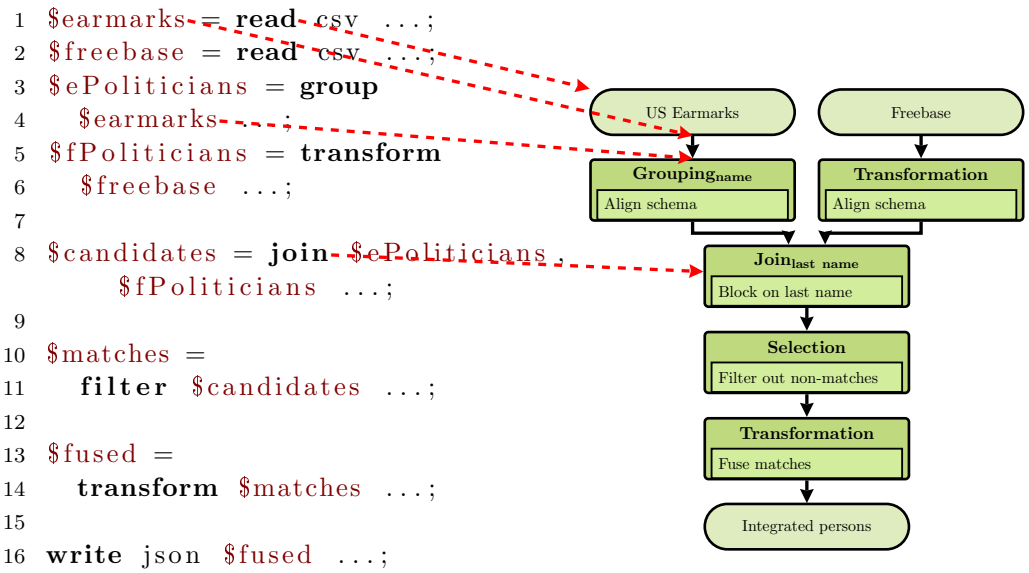


Figure 2.7: Meteor script and resulting Sopremo plan.

2.4 Sopremo operator model

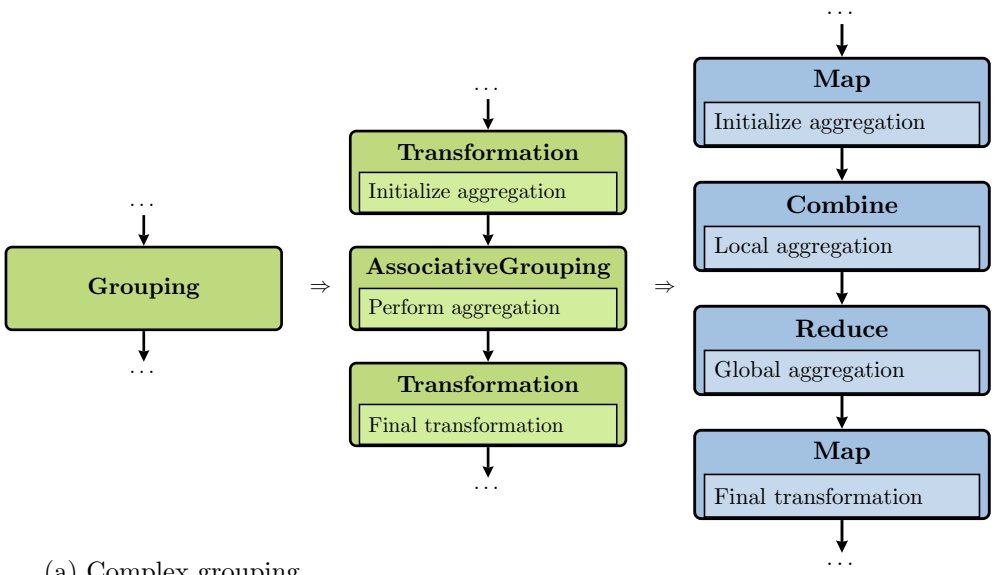
In this section, we present the Sopremo operator model with its novel techniques. The Sopremo components form the largest part of the higher language layer and provide a solid base for implementing advanced operators, such as our new data cleansing and integration operators. In the following, we discuss the four key features of Sopremo:

1. Operators can be *composed* of other operators to facilitate modularization and code reuse as well as reduced complexity when developing new operators.
2. *Extensible* design that allows developers to create new packages of operators, functions, file formats, and data types.
3. A flexible *semi-structured data model* and corresponding manipulation expressions that allow end users to easily describe their data transformations.
4. And a powerful *optimizer* that can optimize complex queries with operators from different packages and derive new reordering on-the-fly.

The Sopremo optimizer is primarily developed by Astrid Rheinländer [Rheinländer et al., 2013, 2014], while I lead the development of the other components.

2.4.1 Operator composition and decomposition

To facilitate extensibility, we introduced the concept of operator composition in Sopremo. Following the good practices of modularization and information hiding in software engineering, developers can define complex operators using simpler ones. Operator composition enables code reuse, and allows complex operators to immediately benefit from more efficient re-implementations of simpler operators.



(a) Complex grouping (b) Elementary grouping (c) Data flow

Figure 2.8: Decomposition of the grouping operator of the Sopremo base package.

A Sopremo operator can either be defined as an *elementary operator* with an implementation as a data flow of second-order functions or as a *composite operator* consisting of other elementary and composite operators. Package developers must ensure that all (recursive) operators can be reduced to a (possibly large) set of interconnected elementary operators.

Figure 2.8 shows the implementation of the composite `grouping` operator, which saves network traffic with pre-aggregations. The operator can be decomposed into three elementary operators as shown in Figure 2.8b. The first `transformation` extracts all values that should be aggregated. These values may now be associatively aggregated in the AssociativeGrouping operator without changing the schema; a logical requirement of associative aggregations. Finally, we use another `transformation` operator to produce the desired schema. These elementary operators can be directly translated into a data flow as depicted in Figure 2.8c. `Transformation` becomes an inexpensive `map` that is always executed in-memory. Further, the AssociativeGrouping operator can now be translated into a combine and a reduce. The combine locally pre-aggregates the data and thus reduces the network load.

Example 2.1. Consider the `grouping` of earmarks from our running example. Earmark records are grouped by the first and last names of the enacting politician and all earmark IDs are collected in an array. The corresponding aggregation functions would choose the first name value and materialize all IDs. Thus, the first `transformation` returns an array `[first, last, [id]]`. The combinable and reduce function takes the names of the first record and merges the arrays of the earmark IDs. Therefore, the first and last names need to be transferred over network at most once for each node.

This example also demonstrates the advantages of operator composition in two other aspects. First, every package developer who wants to perform aggregation as part of another operator may directly use this grouping operator without knowing its non-trivial implementation details. In fact, we replaced a simpler version of the `grouping` operator with this implementation in a recent Sopremo version without breaking any code. Second, composing operators from other known operators immediately allows the optimizer to partially optimize the plan.

Moreover, the `transformation` operator in this `grouping` is the same operator that users invoke when they perform data transformations. Improvements in the `transformation` operator also improve the performance of this `grouping` implementation. However, the internal elementary operator *AssociateGrouping* is not exposed to the user and can be used only by package developers.

2.4.2 Extensible design

To seamlessly integrate domain-specific Sopremo packages, these must satisfy some constraints. Each package and its operators must be self-contained in three ways. First, operators are self-contained in that the Sopremo programmer provides a parallel implementation of new operators directly through data flows or indirectly through operator composition. Second, operators expose their properties through a reflective API. The properties, such as the condition of a join, are transparently managed and validated by the operator itself. Operators may use their properties to choose an appropriate implementation. Thus, no additional knowledge outside of the packages is required to properly configure the operators. Third, the package developer may optionally provide relevant metadata to aid the optimizer in plan transformation and cost estimation. If developers provide more metadata about their operators, optimizers can enumerate more alternative plans and estimate costs more accurately.

To illustrate the current analytical capabilities of the Sopremo packages that are shipped with the system, Table 2.1 lists all operators, the Meteor syntax, and the functionality from the base-package. These operators can be seamlessly mixed with our new data cleansing and integration operators as well as the other currently available packages information extraction and web extraction.

Further, packages can bring domain-specific file formats, functions, and constants. For instance, our data cleansing package contains many similarity functions and conflict resolution functions. Moreover, package developers may also provide custom data types. For example, a machine learning library might introduce an efficient matrix representation to save memory, network bandwidth, and computation time.

2.4.3 Data model and manipulation

To ingest as many different datasets as possible, the *Sopremo data model* is a semi-structured data model that is built upon Json and its notation. We also do not require users to deliver any schema definitions. However, we later see how our data cleansing operators can be used to enforce a specific schema.

Operator	Meteor syntax	Description
Selection	**filter** $in *where* <predicate>	Retains elements satisfying a given predicate.
Transformation	**transform** $in *into* <expression>	Transforms each element of the input according to a given expression.
Join	**join** $in1, $in2 ... *where* <predicate> *into* <expression>	Joins two or more input sets into one result-set according to a join condition. Supports outer, theta, and semi joins.
Grouping	**group** $in *by* $in.*key* *into* <predicate>	Groups the elements of one input on a grouping key to aggregate values.
Bag Union	**union all** $in1, $in2 ...	Concatenates two or more inputs.
Set Difference	**subtract** $in1, $in2 ...	Set-based difference of two or more inputs.
Set Intersection	**intersect** $in1, $in2 ...	Set-based intersection of two or more inputs.
Set Union	**union** $in1, $in2 ...	Set-based union of two or more inputs.
Pivotization	**pivot** $in *around* <path> *into* <expression>	Revolves nested record around a pivot element, such that each unique pivot value results in exactly one record containing the original records.
Enumeration	**enumerate** $in *with key* <path>	Efficiently assigns a unique ID to each record in parallel.
Replace (All)	**replace** (**all**) $in *at* <path> *with* $dictionary *default* <value>	Replaces atomic values with a defined replacement expression.
Sorting	**sort** $in *on* <path> *order* descending	Sorts the input globally.
Array Split	**split array** $in *on* <path>	Splits an array into multiple tuples.
Unique	**unique** $in	Turns a bag of values into a set of values.

Table 2.1: Overview of available Sopremo-Base operators.

Like Json, our data model has three major kinds of data values. First, it has *atomic values* such as string literals, numbers, and Boolean values. Second, values may be contained inside *arrays*. Third, *objects* associate string literals with values. Of course, arrays and objects may be nested as deeply as needed, but it is not possible to express circular data structures. Thus, our data model strictly generates data trees. Custom data types can be implemented as any of three major kinds. In most cases, custom data types should be atomic and can, thus, accessed only atomically in operator properties. However, the matrix from a machine learning library could, for example, be programmatically accessible as an array of arrays and thus allow element values to become join keys.

Sopremo provides a wide range of evaluation expressions that consume a data tree and produce a data tree as shown in Table 2.2. Simple expressions access object or array values, calculate numeric values, or perform Boolean comparisons. Advanced expressions create deeply nested objects or arrays and form expression trees. Further, aggregation expressions consume data streams, modeled as one-time consumable array nodes, and

Type	Expression	Meteor syntax	Description
Path	Input Selection	$in2	Starts a path by selecting an input.
	Object Access	<path>.*key*	Selects the value in an object.
	Array Access	<path>[0]	Selects a single value or a sub-array from an array.
		<path>[1..−1]	
	Array Projection	<path1>[∗].<path2>	Selects path2 for elements in path1.
Creation	Constant	1, 1.0, 1d, **true**, **false**, ' literal ', " literal "	Returns a constant value.
	Object Creation	{*key*: expression, ...}	Creates an object with given key/-values.
		{path: expression, ...}	
	Array Creation	[exp1, exp2, ...]	Creates an array of the values.
	Function Def.	sum = **fn**(a, b){ a + b }	Defines a new function.
Boolean	Comparison	<exp1> == <exp2>	Compares two values.
		or: <, <=, >=, >, !=	
	Membership	<exp1> *in* <exp2>	True if first value element in second.
	Conjunction	<bool1> **and** <bool2>	Logical conjunction.
	Disjunction	<bool1> **and** <bool2>	Logical disjunction.
	Negation	**not** <bool>; **or**: !	Negates the result.
Calculation	Arithmetic	exp1 + exp2	Arithmetic evaluation.
		or: −, /, ∗	
	Ternary	exp1 ? exp2 : exp3	If first value is **true**, returns second value and third value otherwise.
		exp1 ?: exp3	
	Invocation	sum(exp1, ..., expN)	Invokes function with parameters.
	Coercion	(**int**) exp; exp **as int**	Coerces the value to the given type.

Table 2.2: Sopremo expressions and their Meteor syntax for data manipulation.

calculate the values. These expressions are also used to specify the properties of operators. For example, a join condition is a Boolean comparison, a transformation is an object creation, and the `grouping` operator uses object creations mixed with aggregation expressions. Further, *key expressions* are all expressions within a query that define the partitioning of data. For instance, the grouping condition defines which tuples must be grouped together.

The schema inferencer enumerates all relevant key expressions and creates a pseudo tuple layout as shown in Figure 2.9. For each key expression, the physical data layout contains a field that serializes the byte offset of the serialized value. For that purpose, the data serializer hooks inside the serialization of arrays and objects to retrieve the proper index. Calculated values are materialized and appended after the actual value to avoid deserialization of the complete record during the shuffle phase.

2.4.4 Sopremo optimizer

Sopremo provides pay-as-you-go implementation and optimization of new operators to enable developers to rapidly add new operators by implementing basic algorithms, which can be extended and improved over time. Similarly, the Sopremo optimizer allows devel-

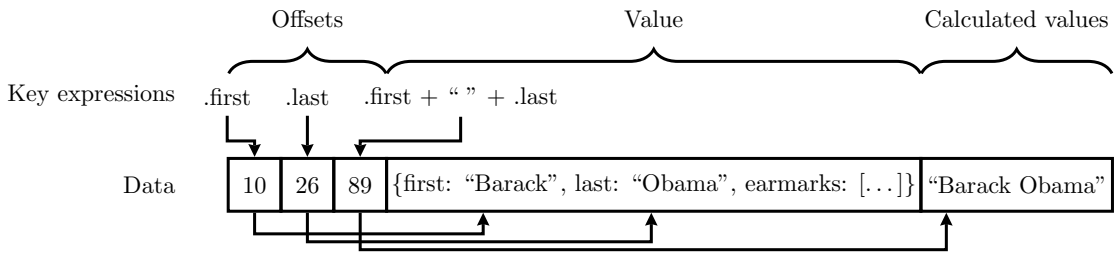


Figure 2.9: Conceptual physical data layout of Sopremo. Values in the actual physical data layout are efficiently binary encoded.

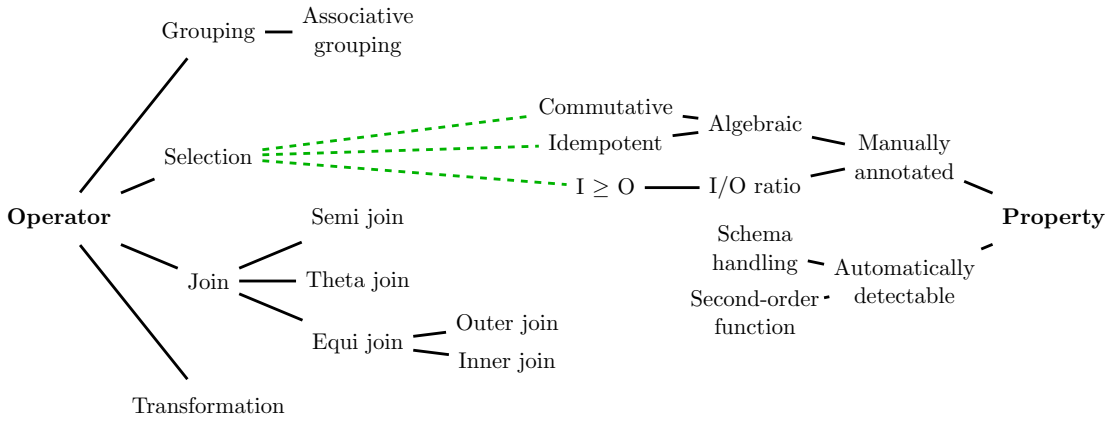


Figure 2.10: Excerpt of the Presto ontology. Black, solid edges show specialization relationships (isA), while green, dashed edges define the hasProperty relationship between operator and property.

opers to incrementally provide and refine semantic information (*metadata*) about their operators, which is used for query optimization (e. g., transformation rules, interesting properties, or cost models). By design, new operators or improved metadata are compatible to existing rewrite rules and inferences.

The Sopremo optimizer relies on the extensible ontology Presto [Rheinländer et al., 2013], which subsumes and defines relationships between operators, operator properties, and cost models based on shared properties and concrete semantics. Figure 2.10 shows an excerpt of the ontology. On the left hand side, we see the operator ontology, where children are specializations of the general operators. On the right hand side, the property specialization tree defines basic properties. Between both parts of the ontology there exist `hasProperty` relationships. In the figure, we can see that a selection produces not more records than it consumes and it is commutative and idempotent.

The optimizer uses this ontology to reason about subsumptions (e. g., an equi-join is a special inner join) and relationships between operators and properties (e. g., a selection is commutative). Most importantly, properties of operators are propagated towards the children. For example, if we annotate the grouping operator with the property that it changes the schema, also the child operators are implicitly annotated. Package developers

may exploit this mechanic to easily add their operators to the ontology if they behave similar to relational operators.

Further, the optimizer rewrites Sopremo data flows using both a set of general reordering axioms and a set of operator-specific reordering rules, which are explicitly modeled in Prolog and annotated to the operator. The general rewrite axioms allow the optimizer to discover novel options for rewriting, which are not explicitly modeled in the operators' metadata.

Definition 2.2 (Reordering rule)**.** A *reordering rule* describes a rewrite of a plan to create an *equivalent* plan, where the operators appear in different order, but both plans result in the same output for any input dataset(s).

For example, consider the first rule of Listing 2.2. With this rule, the optimizer can reorder any operator that has been annotated as commutative, such as the selection operator. Further, the second rule can reorder any two single input, record-at-a-time (RAAT) operators without read write conflicts, effectively performing the black box optimization of Hueske et al. [2012].

```
1 reorder(X, X) :- hasProperty(X, "commutative").
2
3 reorder(X, Y) :- hasProperty(X, "single-in"), hasProperty(X, "RAAT"),
    hasProperty(Y, "single-in"), hasProperty(Y, "RAAT"),
    not readWriteConflicts(X, Y).
```

Listing 2.2: Rewrite templates of the Sopremo compiler

For brevity, we use an algebraic notation of reordering rules in this thesis. The equivalent reordering rules for the rewrite templates are as follows.

$$X_1^{Comm}(X_2^{Comm}(\mathcal{R})) \equiv X_2^{Comm}(X_1^{Comm}(\mathcal{R})) \qquad \text{(Rule I)}$$

$$X^{RAAT}(Y^{RAAT}(\mathcal{R})) \equiv Y^{RAAT}(X^{RAAT}(\mathcal{R})) \qquad \text{(Rule II)}$$

$$\text{if } \neg readWriteConflicts(X, Y)$$

Composition also improves optimization: Reordering rules that cannot be applied to a composite operator might be valid for its building block operators. Additionally, operators may have different implementations that are initially chosen during query compilation through direct or indirect configuration or by the optimizer.

In this thesis, we additionally consider some *fuzzy reordering rules*. Rewritten plans with these rules do not necessarily return the same result for some input datasets. However, data cleansing and integration queries often do not solve all data quality issues with perfectly. Fuzzliy reordered plans might resolve more or less data quality issues, but still correspond approximately to the user's query intent.

Definition 2.3 (Fuzzy reordering rules)**.** A *fuzzy reordering rule* describes a rewrite of a plan to create a *fuzzy equivalent* plan, where the operators appear in different order, but the results are approximately equivalent.

For example, our duplicate detection algorithms typically prune the search space for higher efficiency. When pushing a selection through the duplicate detection algorithm, fewer input datasets need to be processed and the runtime performance increases. However, a Sorted Neighborhood Method [Hernández and Stolfo, 1995] may perform additional comparisons with the same window size and smaller input, so that more duplicates are found (see Section 5.4 for more details). We assume that a user usually benefits from more duplicates in a shorter time, so that we explicitly consider such fuzzy reordering rules.

We omit the detailed description of how the optimizer enumerates and prunes the search space [Rheinländer et al., 2013]. However, we want to emphasize that the compiler operates on directed acyclic graphs and allows package developers to plug-in their own cost estimators for network data volume, disk data volume, and CPU usage.

Summary In this chapter, we presented the parallel data analytics platform Stratosphere. We discussed the overall architecture and highlighted the advanced Map/Reduce functionalities of the data flow engine including more sophisticated second-order functions, complex data flow graphs, and a more flexible data model compared to other state-of-the-art systems.

We then focused on the higher level language layer and emphasized the extensibility. We provided a short overview over the query language Meteor and the translation of queries into Sopremo plans. Further, we explained the concept of operator composition in Sopremo that we heavily utilize for our new data cleansing and integration operators and sketched the Sopremo optimizer with its extensible operator and property ontology. Building on top of the presented system, we introduce our new operators in separate chapters.

3

Scrubbing a Dataset

In this chapter, we introduce our novel `scrub` operator. The operator applies declaratively specified rules to a dataset to enforce validation constraints, such as value types, functional dependencies, or uniqueness.

Database administrators may use this operator periodically to cleanse their relations with constraints that may not be formulated in their DBMS to ensure that even those advanced constraints still hold. Further, in data integration projects, we use this operator as a preprocessing step in our data integration workflow.

In the remainder of this chapter, we introduce a conceptual model for this operator, demonstrate how a Meteor query applies this operator to a dataset, discuss the implementation details, and develop optimization opportunities.

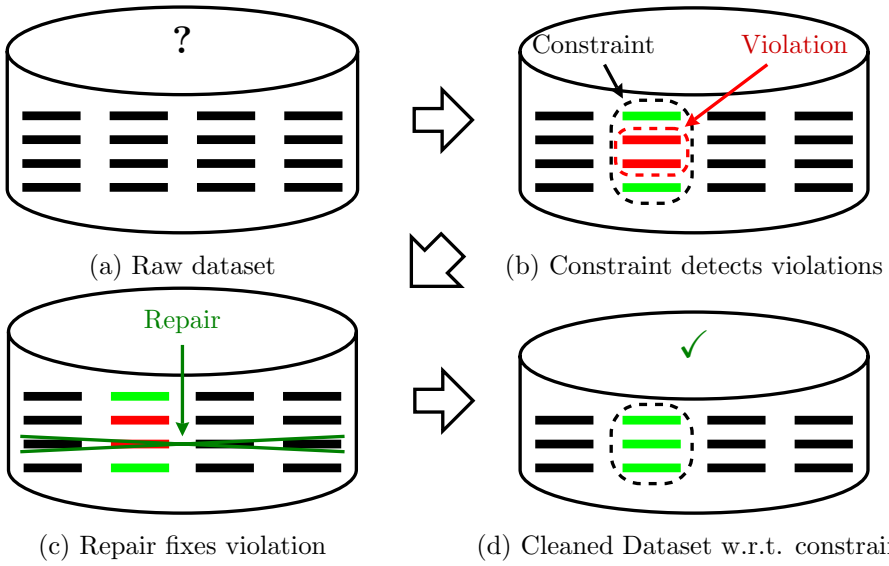
3.1 Constraint and repair model

Data scrubbing denotes the process of enforcing validation *constraints* on a dataset by detecting *violations* and performing *repairs* to create a consistent clean dataset. Validation constraints may range from domain types and range definitions over integrity checks to holistic data dependencies. Repairs modify records to conform to the constraints or even remove some records to guarantee a clean dataset. More formally we define constraints as functions that validate the datasets or parts thereof.

Definition 3.1 (Constraint)**.** A *constraint* is a function $c\colon 2^{\mathcal{R}} \to [true, false]$ that returns $true$ for subsets of the relation $\mathcal{R}$ that satisfy the constraint and $false$ otherwise.

Because constraints can be used to validate subsets of the original data, the validation can be run in parallel. Subsets that do not satisfy a constraints form violations.

Definition 3.2 (Violation)**.** The set of *violations* $\mathcal{V}_c \subseteq 2^{\mathcal{R}}$ with respect to a constraint c is a set of disjoint, minimal subsets of the relation $\mathcal{R}$ such that for each violation $v \in \mathcal{V}_c$ the given constraint does not hold (i.e., $c(v) = false$). Each record $r \in \mathcal{R}$ not contained in any violation is *violation-free* that is they do not contribute to any violation: $\forall \mathcal{R}' \subseteq \mathcal{R} \setminus r\colon c(\mathcal{R}') = c(\{r\} \cup \mathcal{R}')$.



(a) Raw dataset (b) Constraint detects violations

(c) Repair fixes violation (d) Cleaned Dataset w.r.t. constraint

Figure 3.1: Conceptual workflow of applying a uniqueness constraint and corresponding repair to a dirty dataset.

Violations must be minimal but also disjoint. For example, a violation of a uniqueness constraint contains exactly all records that share the same value. Fewer records would result in overlapping sets; more records do not satisfy minimality. To receive a violation-free dataset, we could simply remove all violating records. However, consistent query answering provides techniques to retain at least some of the violating records and their information through repairs [Arenas et al., 1999].

Definition 3.3 (Repair)**.** A *repair* is a function $r\colon \mathcal{V}_c \to 2^{\mathcal{R}}$ that fixes some or all violations, such that the constraint c holds for the repaired dataset consisting of all violation-free and all fixed records (i.e., $c(\bigcup_{v\in\mathcal{V}_c} r(v) \cup (\mathcal{R} \setminus \bigcup_{v\in\mathcal{V}_c} v)) = \mathit{true}$).

Similar to the NADEEF data cleansing system [Ebaid et al., 2013], we do not assume that all violations can be completely fixed. Some violations may require additional domain knowledge for a proper repair, may imply two or more equivalent, but diametrically opposed repairs, or may be impossible to repair even for domain experts. Therefore, we define repairs as *best effort* functions; they may not be able to save all records. For each violation, we apply a series of repair functions on a violation, whereas each repair possibly reduces the set of violating records. If records remain in violation after all repairs, they are removed from the final dataset.

Figure 3.1 depicts the workflow for our scrubbing model with a uniqueness constraint. In Figure 3.1a, the state of the dataset is unknown with respect to the constraint. When we apply the constraint in Figure 3.1b, we find two records that together violate the constraint by sharing the same value. We apply a list of repair functions to each violation in Figure 3.1c, for example by filtering a violating record. Consequently, we receive a violation-free dataset in Figure 3.1d.

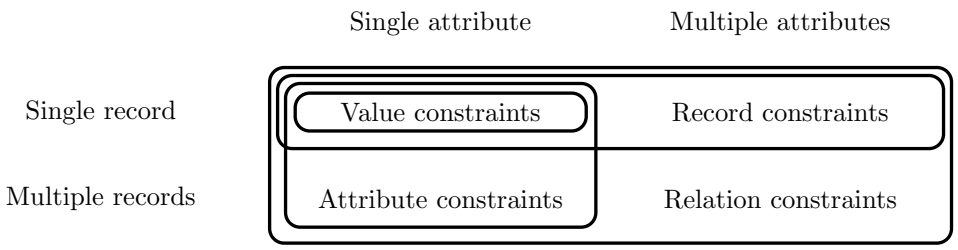


Figure 3.2: Expressiveness of the different constraint types. Inclusion indicates specialization.

From this example, we infer the necessary information for declarative rules. A *scrubbing rule* consists of a path, a constraint, and a list of repairs. We can now define our `scrub` operator.

Definition 3.4 (Scrub operator)**.** The *scrub operator* applies a list of scrubbing rules on a dataset $\mathcal{D}$. For each scrubbing rule, the operator applies the constraint to the respective paths and repairs possible violation in a best-effort manner. The resulting dataset is violation-free according to the scrubbing rules.

3.1.1 Constraint types

The expressiveness of our operators heavily depends on the supported types of constraints and repair functions. We first discuss on the constraints and later elaborate on the repairs. Related work mostly focuses on the origin of data quality issues and seldom addresses the constraints that can be used to detect them. For example, Rahm and Do distinguish between single and multiple sources as well as schema and instance level data quality problems [Rahm and Do, 2000]. They propose a combination of data mining and profiling heuristics to identify the underlying constraints and apply them on the data. For instance, misspellings can be solved by sorting the data and identifying similarly written values, while heterogeneous value representations should be solved with a master data table.

However, because our operators ultimately resolve the data quality issues in parallel, we classify the constraints by data dependencies. Hence, we implicitly model what kind of information needs to be available to be able to detect violations. We distinguish between value, record, attribute, and relation constraints. *Value and record constraints* operate only on single records. A record constraint may access all attributes of the respective record, while a value constraint can access only one attribute, or in general one specific path element. *Attribute constraints* may access all records, but only one attribute. Lastly, a *relation constraint* has complete access to the complete dataset.

Figure 3.2 visualizes the expressiveness of the different of constraints. Every constraint may be expressed as a relation constraint; however, this limits the degree of parallelization as we show later. In general, the more types of violations a type of constraints may detect, the more data dependencies it requires and therefore the less parallel the detection of violation can be performed. We further discuss the types and some of

	Constraint name	Example in Meteor	Violation
Value	Presence	id: required	Value not present or null
	Data type	id: type(**int**)	Not expected data type
	Pattern	zip: hasPattern('[0−9]\{5\}')	Different pattern
	Value range	amount: inRange(0, 10e10)	Outside range
	White list	state: inList($states)	Not in list
	Black list	sponsor: notInList(['?', 'unk.'])	Value in list
	Check value	zip: check(**fn**(x){ (**int**)x > 0 })	Boolean expression is false
	Fail	amount: fail ?: **fn**(x){ x * 1000 }	All values are violations → value always normalized
Record	Any present	{zip, POnumber}: required	All values not present or null
	Check record	{sponsorCode, state}: check(**fn**(c,s){substr(c,0,2)==s})	Boolean expression is false
	Alternative fields	{sponsor, recipient}: alternative	All fields (not) present
	Valid combination	{zip, city}: inList($zipCity)	Combination not in list
	Conditional value constraint	recipCountry: **if**($in.recipient != '')required	Condition met and constraint violated
Attribute	Uniqueness	id: unique	Duplicate values
	Distribution	date: distribution(uniform)	Measures error > threshold
	Aggregation values	amount: aggregate(&sum, 10e10, 10e5)	Aggregate differs from given value more than a threshold
Relation	Functional dependencies	sponsorState: dependsOn(sponsorCode)	Same determinant, different dependent
	Unique column combinations	{id, recipient, sponsor}: unique	Duplicate projected row
	Conditional attribute or relation constraint	recipient: **if**(recipCountry == "USA") dependsOn(recipientCode)	Condition met and constraint violated

Table 3.1: Supported types of constraints

the implemented constraints in the following. All 18 current constraints are summarized by Table 3.1 with their Meteor syntax. Note that Section 3.2 provides a more in-depth discussion on the integration of the `scrub` operator in Meteor.

Value constraints do not need any additional information except the actual datum. Therefore, a value constraint can be checked for each record in parallel. Further, value constraints on different attributes can be executed in arbitrary order. The class of value constraints already covers a broad spectrum of constraints.

For example, the *presence constraint* detects the absence of the value, the *data type constraint* finds mismatching data types, and the *pattern constraint* discovers ill-formed textual values. Further, the *check value constraint* evaluates an arbitrary Boolean expression, and can, for instance, verify a checksum embedded in the value, such as in visa card numbers.

Record constraints additionally need other values from the same records. In contrast to value constraints, record constraints induce a partial order, because values that are repaired in one scrubbing rule may influence the outcome of another rule. Record constraints usually check for consistency across values, caused by redundancy or logic constraints.

For example, *conditional value constraints* enforce any of the previously introduced value constraints on those records that satisfy a given condition. These conditions become necessary when dealing with heterogeneous datasets, such as the `US Earmarks` dataset, which contains dedicated records for sponsors and recipients of an earmark. Hence, valid recipient information can be enforced only on records about recipient. Additionally, the *any present constraint* checks the presence of one of two mandatory fields, for instance, post address must either have a zip code or post office box number. The *valid value combination constraint* verifies the combination of zip code and city with a white list dictionary.

Attribute and relation constraints allow the detection of violations of holistic constraints, such as uniqueness or functional dependencies. Attribute constraints access only one attribute, while relation constraints may operate on the complete table. However, both constraint types define their data dependencies through expressions that can be used to group the relevant entries only and facilitate parallel scrubbing. Still, fewer parallelization opportunities exist in comparison to value and record constraints.

Examples of attribute constraint are the *distribution constraint* and the *aggregation constraint*. The *distribution constraint* compares the error between actual and expected value distribution with a given error threshold. Similarly, the *aggregation constraint* verifies if the aggregation over one attribute yields the expected value (with error). Typical relation constraints are *function dependencies* and multi-column *unique constraints*. Additionally, all attribute constraints becomes relation constraints when conditionalized on another attribute.

3.1.2 Repair methods

A violation of a value and record constraint is a single record, while a violation of an attribute or relation constraint is a group of records. The user specifies one or more repair methods for each constraint to resolve the violations. A repair method may partially solve violations. In contrast to NADEEF [Ebaid et al., 2013] and similar related work, the `scrub` operator does not assess the repair candidates. Choosing between concurrent, interacting repair candidates needs complex reasoning algorithms, which hinder efficiency and scalability. Instead, we focus on user-supplied, certain fixes and incomplete repairs.

Each constraint specifies a default repair that users should change. For most violations, the default repair simply removes all violating records. However, for specific types of constraints, we provide more sophisticated repair methods. For example, when the user expects an atomic data type, such as a number, a lenient parser tries to extract the value from the actual value. Table 3.2 lists more specialized repair functions. Some repairs require configuration as described in the following.

Constraint	Repair	Example in Meteor	Comment
Any	Remove	remove	Removes violating records
Any	Ignore	ignore	Retains violating records (as last repair to keep records)
Any value	Default	default("value")	Sets a default value
Any value	Expression	substr($earmark.code, 0, 2)	Calculates value with Meteor
Any relation	Retain Xth	retain(1)	Retains the Xth record
White list	Choose nearest	nearest(&jaro, .5)	Selects best value from white list with similarity function
Data type	Lenient parser	parse	Tries to extract useful values
Alternative fields	Pick one	pick($earmark.sponsor)	Removes values in fixed order until one left
Distribution	Bias correction	correctBias	Aligns data to expected value

Table 3.2: General (top) and constraint-specific (bottom) repair functions

Most value constraints may be resolved by replacing the violating value with a valid value supplied by the user. The user may also specify a Meteor expression that calculates the correct value either based on the actual value or even using the complete record. Further, the user may decide to ignore all remaining violations to retain the information. In that case, the dataset is cleaned in a best effort manner and still partially incorrect, similar to NADEEF.

For white list constraints, the user already supplied a list of valid values. An appropriate repair can choose the nearest value given a similarity function (Levenshtein by default). He can also give a lower similarity bound, with which the repair function decides whether the proposed value is plausible enough. If no similar value has been found, successive repair functions try to resolve the violation.

When several alternative fields are present, *pick one* repair removes the superfluous attributes in a user-specified order until only one value remains. Similarly, for any attribute or relation constraint that provides an array of violating values, users may choose one record through a fixed index or a Meteor expression. Further, we provide a simple *bias correction* for unexpected distributions, which adjusts all values until the mean equals the given expected value.

Neither the set of constraints nor repairs should be considered as complete. Users may implement their own constraints and repair functions in Java through respective interfaces.

3.1.3 Execution semantics

Before discussing the execution semantics of the `scrub` operator, we quickly review how the state-of-the-art data repairing system work by examining the NADEEF system [Ebaid et al., 2013]. NADEEF incorporates domain knowledge of the user through user-specified *cleaning rules* to increase the repair quality. Each cleaning rule consists of a detection routine and optionally one repair function, which may provide alternative

candidate fixes. The language of NADEEF is expressive enough to cover all currently known cleaning rule types and even allows customized detection routines. To detect violations, NADEEF naïvely compares all record combinations with quadratic complexity. For each violation it collects candidate fixes and then uses complex heuristics to weight the different fixes and chooses the set of fixes that globally causes the fewest value changes. Further, to avoid infinite loops caused by alternating repairs and violations, NADEEF limits the maximum number of value changes to 2 by default. Hence, NADEEF does not guarantee a completely fixed datasets in favor of retaining as much information as possible.

However, while obtaining a good quality, NADEEF performs already poorly on small datasets. The goal of our operators is to provide a scalable implementation and consequently, we focus on a simple yet highly efficient approach, such that our `scrub` operators achieve orders of magnitude higher throughput than NADEEF. NADEEF uses a quadratic routine to detect conflicts and enforces data partitioning only optionally, while our constraints have been streamlined for parallel execution. Similarly, the data repair methods have polynomial or even exponential complexity, while our repairs perform linearly. Therefore, NADEEF needs half an hour to enforce 10 constraints (mostly FDs) on a dataset with 100k rows and 9 attributes [Ebaid et al., 2013], whereas our `scrub` operator processes such data volume in the order of seconds on one machine. Nevertheless, the focus of NADEEF and the `scrub` operator are fundamentally different; NADEEF focuses on data cleaning quality and the `scrub` operator on efficiency and scalability.

In general, the `scrub` operator checks each constraint only once and in the specified order. Obviously, such execution semantics means that a misspecification results in further violations. For example, if a repair function for a functional dependency replaces the determining value in a violation, the new value may now conflict with another record. We leave it to the user or external tools to provide a valid specification to achieve high efficiency for the following reasons:

- To resolve violations in parallel, we need as few data dependencies as possible. If newly introduced values cause new conflicts, the verification and repair process would cause new data dependencies.
- In the worst case, the process may run infinitely. In NADEEF, the maximum number of concurrent value changes is limited to 2 by default to avoid infinite loops, whereas we practically limit the number of repair steps to one.
- We can give accurate performance estimations, since we limit the number of repair steps for each violation to one.
- Side-effects of different repair functions are hard to capture, since we want users to use the full expressive power of Meteor including user-defined functions in Java. NADEEF similarly trades off expressiveness for optimizability.
- Even state-of-the-art research did only study the interaction between multiple types of constraints for very limited settings, for example matching dependencies and functional dependencies [Fan et al., 2014]. Because even sophisticated systems, such as NADEEF, cannot give data quality guarantees for a wide range of constraint types, we instead focus on efficiency and scalability.

In Algorithm 3.1, we formalize the execution semantics of the data `scrub` operator. Given an input dataset and a set of rules, the operator processes the rules sequentially. For each rule, it partitions the dataset on the path of the rule and depending on the type of the constraint. Starting in Line 4, the operator can now check each partition individually for violations. For each violation, it sequentially applies the repair methods of the rule in Line 7. Eventually, the operator returns all fixed records as the primary result and filters unresolved violations, which are collected in a secondary result set.

Input : Dataset $\mathcal{D}$, Rules $\mathcal{R} = \{r | r \in Path \times Constraint \times 2^{Repairs}\}$
Output: Cleansed dataset $\mathcal{D}$, Violations $\mathcal{V}$

```
1  foreach Rule r ∈ R do
2      P ← partition D for r.Path and type of r.Constraint;
3      D ← ∅;
4      foreach Partition p ∈ P do in parallel
5          if p violates r.Constraint then
6              violations ← p;
7              foreach Repair repair ∈ r.Repairs do
8                  perform repair on violations;
9                  D ← D ∪ repaired records;
10                 violations ← unrepaired records;
11             end
12             V ← V ∪ violations;
13         else
14             D ← D ∪ p;
15         end
16     end
17 end
```

Algorithm 3.1: Conceptual data scrubbing algorithm.

Because Stratosphere currently relies completely on HDFS with its horizontal partitioning, we also partition the data only horizontally (i.e., value/record vs. attribute/relation). If Stratosphere at some points also supports other storage techniques with vertical partitioning of data, we can additionally perform vertical partitioning (i.e., value/attribute vs. record/relation). For brevity, we distinguish only record and relation constraints in the remainder of this chapter.

3.2 Declarative rules in Meteor

In this section, we discuss how users specify scrubbing rules in Meteor. Listing 3.1 shows the `scrub` operator specification for the `US Earmarks` dataset. The script starts by loading the earmarks and two white lists for validating country names and state country combinations, which we use later. Line 4 then starts with the specification of the "**scrub**" operator. The operator takes only a single input source and outputs one

```
1  $earmarks = read csv from 'earmarks.csv';
2  $validCountries = read csv from 'validCountries.csv';
3  $validStates = read csv from 'validStates.csv';
4  $scrubbedEarmarks, $violations = scrub $earmarks
5   with rules {
6    earmark_id:          required,
7    amount:              [required, type(numeric), fn(x) { x*1000 }],
8    {earmark_id, recipient, sponsor}: unique,
9    [recipient_state, recipient_country]:
         if($earmarks.recipient != '') required,
10   recipient_country: inList($validCountries) or default('US'),
11   enacted_year:        [required, type(int), inRange(2000, 2010)]
12 };
```

Listing 3.1: Scrub operator for `US Earmarks` in Meteor.

or optionally two outputs. The *primary* output contains the valid dataset. The optional *secondary* output contains all removed records annotated with the violated constraints.

The operator specification then defines the scrubbing rules with the only property "*with rules*" of the `scrub` operator. To specify rules, we reuse the object creation syntax of Meteor to support arbitrary deeply nested path specifications. In Line 6, we assign the *presence* constraints to the `earmark_id` field.

To ease the specification of multiple constraints per path, we also allow users to specify them at once in arrays. Line 7 defines that the `amount` field must be present and must contain a numerical value. Further, it specifies an anonymous Meteor function that normalizes the amount by multiplying it with 1000. Here, the `scrub` operator transparently creates a *fail* constraint, so that the repair function is applied to each record.

So far, we have specified only value constraints. To specify constraints for more than one attribute, we combine the attributes with the object notation (curly braces). The line "{earmark_id, recipient, sponsor}: unique" defines a unique constraint over the three attributes `sponsor`, `recipient`, and `earmark_id` to enforce that there is no earmark where the same sponsor or recipient appears twice. Meteor transparently handles the type of a constraint and thus provides the same syntax for record and relation constraints.

As described before, we allow multiple constraints for one attribute to be written in one array for convenience. Analogously, we can combine multiple attributes in one array, if they share common constraints. In Line 9, we define the conditional constraint for fields related to the recipient that they must be present if the `recipient` field is set.

Line 10 defines a white list for `country`. Further, we use the "default" repair function to replace invalid records with the given value. The last line defines value constraints for the `year`. The year must be present, an integer, and be in the given time frame, in which earmarks were first published until they have been banned. Because we read the earmarks from a structured dataset, our example only referred to the top level attributes

of the records. Nevertheless, the operator fully supports nested object types. Users specify nested paths in the same way they would define a nested `transformation`.

3.3 Parallel implementation in Stratosphere

In this section, we explain how the `scrub` operator interprets the constraint specification expression and compiles it into a series of base operators. The operator validates the scrubbing specification and translates it to a list of rules. Each rule contains the full path, the constraint, and a list of repairs. The operator expands convenience array notation for multiple constraints per path and multiple paths per constraint definition. In the end, we receive a single list of rules that may be plugged into the conceptual scrubbing algorithm (Algorithm 3.1).

3.3.1 Compilation to relational operators

We implement the `scrub` operator as a composite operator. During query optimization, the `scrub` operator provides a partial Sopremo plan that ingests the input of the `scrub` operator and produces one or two outputs according to the Meteor configuration.

The `scrub` operator creates one sub-operator for each rule depending on the type of the rule. `Scrub record` operators handle value and record constraints, while `scrub relation` operators treat attribute and relation constraints. Each sub-operator is a composite operator that eventually gets expanded to two or three base operators for each constraint as shown for two constraints in Figure 3.3. The first operator checks the constraint and performs repairs. The second operator filters all fixed records and streams it to the next scrub operator. The optional third operator filters all violations. These violating records are ultimately collected with a `union all` and emitted as the second output of the scrub operator.

For `scrub record` operators, the first operator is a `transformation` that emits the record if the constraint holds. Otherwise, it successively invokes the repairs until one returns a valid record. If no repair could be found, the operator annotates the record to describe the type of violation. With two `selection`s, the operator filters the records with and without annotation.

The `scrub relation` operator uses a partitioning expression that each attribute and relation constraint provides. It creates a `grouping` operator with this expression as the grouping key, which materializes the respective records, and invokes the constraint validation. The repair functions are applied similar to the `scrub record` operator, but the resulting valid and violating records are kept in separate arrays. Finally, the arrays of fixed and violated records are expanded with two `array split` operators on the respective paths.

3.3.2 Intelligent decomposition

To increase efficiency and optimizability, we extend the straight-forward decomposition scheme in two ways. First, users may specify *safe* repair methods, such as "`default (...)`",

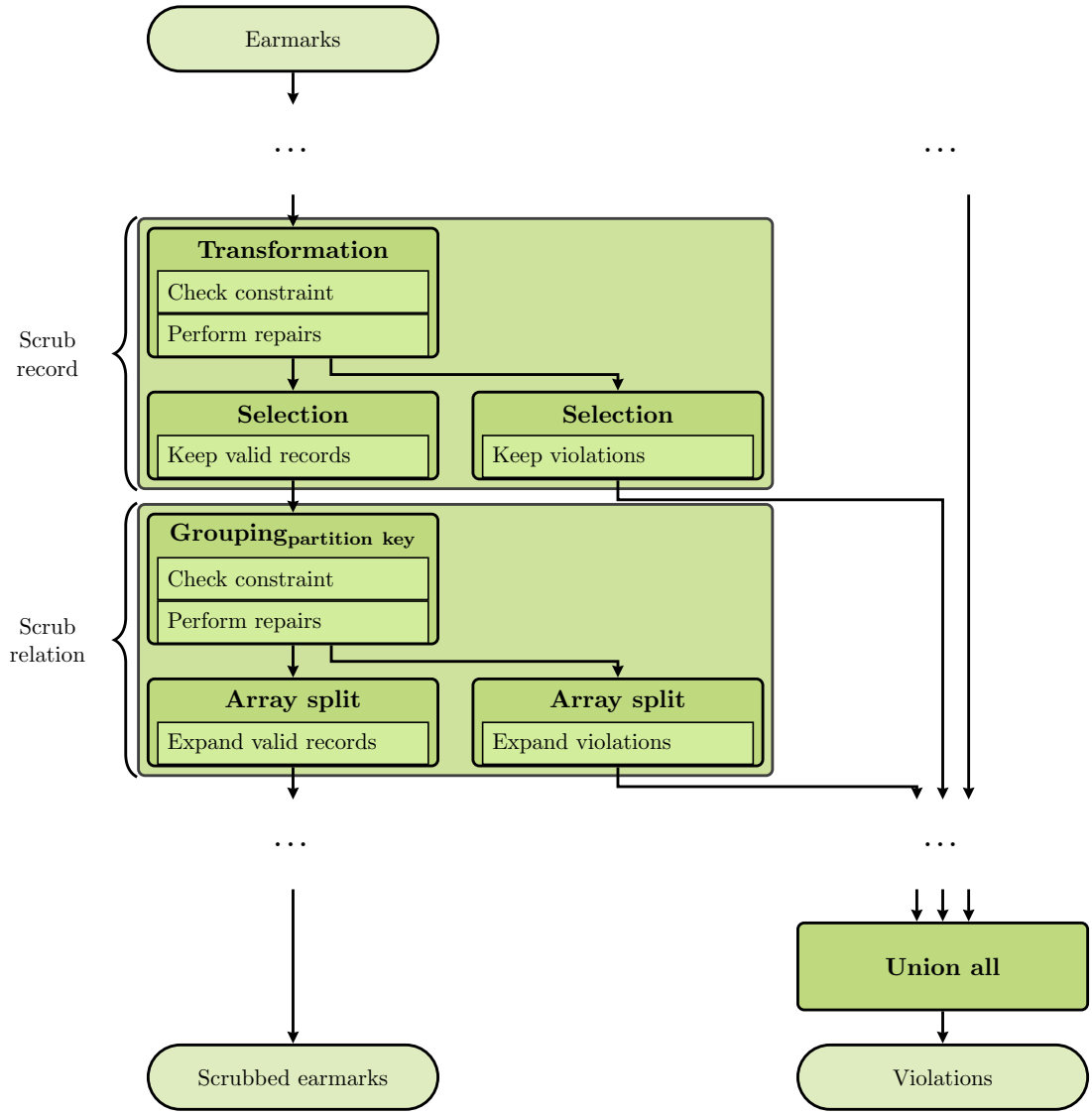


Figure 3.3: Final expansion of composite `scrub` operator with one record constraint (top) and one relation constraint (bottom).

which always fix the values. In that case, the sub-operator does not output any violations and we omit the operator that filters the violations. For `scrub record`, we can also remove the first selection.

Second, we can combine subsequent sub-operators if they have the same partitionings. As soon as one constraint and possible repairs fail for one record, the operator aborts subsequent processing and annotates the record annotated with the first violation. Consequently, the reduced number of operators decreases optimization and start-up time.

Eventually, the three scrub operators may be distinguished as follows: The `scrub record` operator applies a series of value and record constraints with corresponding repairs in one `map` function. A failed repair leads to an early termination. Similarly,

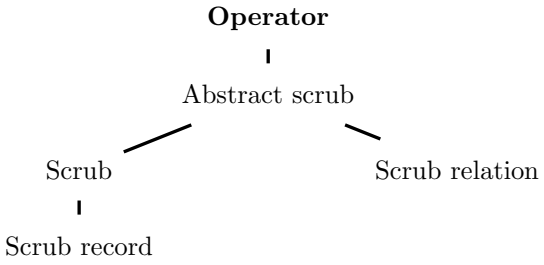


Figure 3.4: Excerpt of the operator taxonomy for scrub operators.

the `scrub relation` operator applies a series of attribute and relation constraints with the same data partitioning and the respective repairs in one `reduce` function. The general `scrub` operator subsumes both operators and can in particular change the data partitioning between constraints. All scrub operators filter records only if they contain at least one unsafe repair function (i.e., the repair may fail for some records).

3.4 Optimization with the scrubbing operator

Because the scrubbing operator decomposes itself completely into base operators, the Sopremo optimizer may already apply a large set of reorderings to these base operators. However, we lose some optimization opportunities if we regard only the elementary pieces and ignore the larger context. We discuss in this section, the general properties of the scrub operator and its sub-operators, develop optimization rules, and integrate the operators into the cost model.

3.4.1 Operator properties

Figure 3.4 shows the operator taxonomy for the scrubbing operator in Presto [Rheinländer et al., 2013]. Intuitively, we would model both sub-operators, `scrub record` and `scrub relation`, as special cases of the general `scrub` operator, because they perform individual subsets of the work. However, not all algebraic properties can be inherited by the sub-operators. Therefore, we have a general abstract scrub operator as the super-operator for all three variants.

All scrub operators are **idempotent**: The output satisfies all constraints and repeated application cannot result in additional violations and subsequent changes (unsound configuration, however, may yield different results with repeated application). Furthermore, `scrub` and `scrub record` are **associative**. The optimizer can merge subsequent operators, because `scrub` conceptually partitions the data for each constraint and `scrub record` operators always share the same record-wise partitioning. In contrast, the `scrub relation` operator can be merged only if the subsequent operators have the same partitioning. `Scrub record` is also **commutative** if no read-write conflicts occur. The other two scrub variants cannot be safely reordered due to the potential filtering of invalid records: Subsequent operators may receive incomplete partitions, which results

in different violation and thus different results. All scrub operators may potentially **filter** the dataset; that is, they return fewer records than ingested. Further, they **keep the schema** on their primary output. The secondary output contains an additional annotation indicating the violation of the filtered record.

Non-filtering scrub

For scrub operators that are safely configured with guaranteed repair functions, we can attribute more specific properties to the operators. First, the record **cardinality does not change** if a scrub operator has no filter repair. Second, all non-filtering scrub operators are **commutative**. Because no records are filtered, the partitionings across different scrub operators remains unchanged as long as no read-write conflicts occur. Third, the `scrub record` operator with safe repairs is actually a special case of a `transformation`. Consequently, all reorder rules that apply to the `transformation` operator can be applied directly to the `scrub record` operator.

3.4.2 Optimization rules

The properties already allow the optimizer to apply any reordering rules that have been defined in the base package. In the following, we define some additional optimization rules for the scrub operator and review optimization opportunities. With the commutativity property, the optimizer can reorder consecutive `scrub record` operators (see second rule of Listing 2.2 on page 29). More general reorderings within the family of scrub operators are possible only through the commutativity property of scrub operators with safe repairs.

Rules with base operators

Because `scrub record` is a record-at-a-time (RAAT) operator with single input, it can also be safely reordered with any other RAAT operator, such as `selection` or `transformation` (see first rule of Listing 2.2 on page 29). With a similar rule, it can be pushed through `joins`, if no read-write conflict occurs. Finally, `scrub record` operators with safe repairs can be reordered with `grouping` operators. For the other two operators, fewer reorder opportunities exist:

$$sc_{rule}(op^{RAAT}(\mathcal{R})) \equiv op^{RAAT}(sc_{rule}(\mathcal{R})) \quad \text{(Rule I)}$$
$$\text{if } |op^{RAAT}(\mathcal{R})| = |\mathcal{R}|$$
$$\sigma_\theta(sc\_rel_{rule}(\mathcal{R})) \equiv sc\_rel_{rule}(\sigma_\theta(\mathcal{R})) \quad \text{(Rule II)}$$
$$\text{if } \forall P \in partition_{rule}(\mathcal{R}) \colon \forall r_1, r_2 \in P \colon \theta(r_1) = \theta(r_2)$$

The first rule shows that we can push any RAAT operator through a `scrub` operator, when it does not change the cardinality (e.g., `transformation` or safe `scrub record`). Reorderings with `selection`, `joins`, and `groupings` are only possible in special cases.

The second rule allows pushing a `selection` through a `scrub relation` operator if partitions are either completely retained or filtered. Both rules can only be performed if no read-write conflict occurs. Putting all rules together, the following example demonstrates how a far we can push a selection towards the data source, which usually benefits the performance.

Example 3.1. Assume that we extend our example query in Listing 3.1 with a subsequent selection for all earmarks enacted by "John Smith" (i.e., recipient = "John Smith").

We now walk through the rules of the listing from bottom to the top. We can easily push the selection through the last three lines of rules up to Line 8, because all the rules are value or record constraints. The "unique" relation constraint requires the application of the second rule. Since the path of the condition is part of the constraints, the rule allows the selection to be pushed even further. The remaining two lines of rules contain again only value or record constraints, so that we can push the selection completely through the scrub operator.

Eager and lazy rule execution

In more complex queries, selections and other operators often cannot be pushed completely through the scrub operators. We thus briefly discuss how such queries can still profit from our reorder rules.

Yan and Larson [1995] coined the term of eager and lazy aggregation in query optimization. Some *eager* calculations of a group-by can be pushed towards the data source through expensive operations, such as joins, to reduce the number of tuples. The remaining *lazy* calculations are executed in the end if they are not needed to reduce the tuples or cannot be pushed through the other operation due to data dependencies.

Through the operator decomposition process, the scrub operator is split into smaller parts, which can be reordered independently. If successive operators reduce the tuple count, the optimizer probably prefers an eager and lazy execution of the scrub rules as demonstrated in the next example.

Example 3.2. Analogously to the previous example, we now select all earmarks given to recipients in Germany. The optimizer can push this selection only through the last rule, because the white list rule for the countries in Line 10 causes a read-write conflict. Nevertheless, it can push some additional rules after the selection for a lazy execution. These are all value and record constraint that do not modify the country (i.e., the recipient black list rule and the conditional rules requiring state and street to be present). Unfortunately, the optimizer cannot reorder the expensive "unique" constraint, since the partitions may potentially change. However, we can push the safe value normalization of the amount after the selection: Through the "unique" constraint, the white list rule, the selection, and even after the rules concerning the country. Altogether, we perform four out of 11 rules after the selection on a probably much smaller dataset.

3.4.3 Cost and cardinality estimation

To assess the performance of reordered plans, the optimizer requires a complete integration of our operators into the cost model. In the following, we describe how to estimate the runtime costs in terms of CPU and network utilization and the resulting record cardinalities of the primary output by reducing the calculation to well-understood relational estimations.

Scrub record

Suppose $scr_C(D)$ denotes the application of a `scrub record` operator with a sequence of record constraints C on the relation D. Further, for each constraint $c \in C$, we have a sequence of repairs R_c. To estimate the output cardinality, we can model scr_C as one generalized selection.

$$scr_C(D) = \sigma_\varphi(D) \tag{3.1}$$

$$\varphi(t) = \bigwedge_{c \in C} (c(t) \vee \bigvee_{r \in R_c} |r(\{t\})| = 1) \tag{3.2}$$

A passing record must thus either directly pass all constraints or be fixed by any associated repair. To measure the selectivity of constraints and repairs, we can use random sampling similar to cardinality estimation of traditional selections in DBMS. Combining these individual selectivity estimates poses the same challenge as conjunctive and disjunctive Boolean filter expressions in traditional DBMS. We can resort to a simple heuristic that is even often found in commercial DBMS: We assume independence of constraints and repairs. The assumption implies that a record does not violate multiple constraints and that repairs address orthogonal issues. Further, for normalization constraint and safe repairs, we can directly specify the selectivity as 0 and 1 respectively to simplify the selectivity estimation.

The runtime costs of a query with the `scrub record` operator can be estimated in a straight-forward manner. First, this operator causes no network traffic, because all sub-operators are resolved to `map` functions. Second, CPU costs depend directly on the cost of checking the constraints, repairing a violation, and the selectivity of previous constraints and repairs. The individual or overall CPU costs can be empirically measured with random sampling.

Scrub relation

We similarly model the `scrub relation` operator as a grouping operator with the grouping keys K and the recursive aggregation function *apply* that applies the constraints on each group $g \subseteq R$. Consider c_1 and r_1 to be the first constraint in C and the first repair in R_c, respectively.

$$scr_C(D) = split_g(\gamma_{K,apply_C(g)}(D)) \tag{3.3}$$

$$apply_C(g) = \begin{cases} g, & \text{if } C = \emptyset \\ apply_{\{C \setminus c_1\}}(fix_{c_1,R_{c_1}}(g)), & \text{otherwise} \end{cases} \tag{3.4}$$

$$fix_{c,R}(g) = \begin{cases} g, & \text{if } c(g) \text{ or } g = \emptyset \\ \emptyset, & \text{if } R = \emptyset \\ r_1(g) \cup fix_{c,R \setminus r_1}(g \setminus r_1(g)), & \text{otherwise} \end{cases} \tag{3.5}$$

It is thus sufficient to estimate the distribution of the partitions and estimate the average runtime of *apply* on the partitions. To estimate the former, we can apply the usual estimation techniques used in the grouping operator, possibly exploiting value distribution statistics in the future. For the latter estimation, we reuse the estimation of the `scrub record` operator. Note that although relation constraints and subsequent repairs operate on groups of records instead of single records, the notion of selectivity still applies.

The cost of the `scrub relation` operator is dominated by the network transfer of the distributed grouping. The additional CPU costs for the constraint validation and repairs can be equivalently calculated to the `scrub record` operator, because our constraints and repairs perform linearly at worst. To empirically estimate the runtime, we could also use focused sampling on the grouping key, and measure the runtime on small, medium, and large representative groups.

Scrub

Accurately estimating a `scrub` operator with many scrubbing rules becomes quickly challenging, because errors in the estimation of one rule propagate throughout the complete estimation. In general, it is worth estimating the cardinality and costs at the same time, because the runtime and cardinality of a rule depends mostly on the number of remaining records. A low selectivity of one rule noticeably reduces the cardinality and thus the runtime of proceeding rules. We can distinguish between the following two cases.

First, the scrubbing rules contain **only record constraints**. Here, the runtime costs are approximately equally distributed among the constraints and depend solely on the number of remaining records. To calculate the selectivity of two or more rules, we treat them as a cascade of selections. Hence, we again assume independence of the constraints and multiply the selectivity of the individual constraints. If we use empiric sampling estimation techniques, we could directly measure the joint selectivity and achieve more accurate results.

Second, the general scrub operator contains **at least one relation constraint**. Then, the runtime is probably dominated by this constraint due to the network costs. Thus, for record constraints, we could estimate the rule costs more approximately, for example by ignoring repair costs. Similarly, for many relation constraints, we can estimate the individual relation constraints more coarse-grain by focusing on network costs and sum them up.

For all scrub variants, we can easily calculate the cardinality of the secondary output from the estimations of the primary output, because the sum of the cardinalities of both outputs must be the cardinality of the input.

Summary This chapter discussed our first data cleansing operator. We defined constraints, violations, and repairs, and presented the currently available constraints and repairs. In comparison to related work, such as NADEEF [Ebaid et al., 2013], we choose simple execution semantics of the operator to achieve good scalability and overall performance.

We showed how users properly specify the constraints and repairs in Meteor. The syntactical elements allow users to group constraints concerning the same attribute and attributes sharing the same constraints for shorter scripts. Through the existing expressiveness of Meteor, users easily specify conditional constraints, custom repair, and normalization functions.

We explained how the Sopremo operator `scrub` can be translated into two sub-operators, `scrub record` and `scrub relation`, and eventually into base operators. We also improved the straight-forward decomposition to a more sophisticated decomposition scheme. Moreover, we discussed properties of the `scrub` operator and resulting optimization opportunities. We developed reordering rules for the different types of constraints and briefly demonstrated the rules in two short examples. We finally sketched how to estimate the runtime cost and cardinality for the `scrub` operator and its two sub-operators.

In a data integration query, the `scrub` operator preprocesses the input sources to remove individual data quality issues and normalize values. In the next chapter, we extract the different types of entities from the cleansed dataset, which can then be used to find duplicates across datasets.

4

Mapping Data to a Target Schema

This chapter presents the `data map` operator, which transforms records from one or more input datasets to one or more output datasets of a different schema using a declarative specification of value correspondences. The operator addresses two main challenges: First, it should retain all information but avoid redundancy. Second, it maintains referential integrity of related input records in the transformed output records.

This operator aligns the schemata of sources to a previously engineered target schema in virtual and materialized data integration scenarios. Further, it helps data maintainers in restructuring their existing datasets, especially for repetitive tasks where the schemata of periodically imported data sources need to be adjusted to the local schema.

The chapter first introduces the concepts from data exchange research and defines the semantics of the operator. We showcase a Meteor query that configures the operator and describe how the `data map` operator uses the second-generation mapping tool ++Spicy [Marnette et al., 2011] to translate the specification into a set of base operators. An overview of optimization and cost estimation techniques conclude this chapter.

4.1 Schema mapping and data exchange

The `data map` operator allows users to easily define *value correspondences* and foreign key relationships, such that users declaratively define their expected results and the operator transparently creates an optimal solution. To find value correspondences, users may use semi-automatic, matured tools from research, such as Clio [Miller et al., 2000], or industry. We build our operator on *data exchange* theory [Fagin et al., 2005a], which we briefly present in this section.

Figure 4.1 depicts the value correspondences between an excerpt of the schemata of three `Freebase` relations and our global target schema. The source side on the left comprises the `politicians` and `parties` datasets as well as the materialized n:m relationship `tenures` between these two entity types. A tenure record contains foreign keys

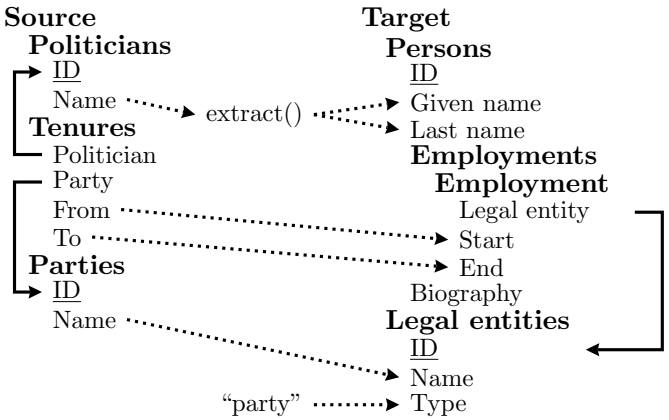


Figure 4.1: Mapping between Freebase and our global target schema.

referring to the corresponding politician and party records. On the right side, we have our two target relations `persons` and `legal entities`. Person records have a nested list of `employments`, which refer to the respective legal entity key.

We can see the following value correspondences between the source and the target side: First, politicians implicitly correspond to persons, because the politicians' names are transformed to persons' first and last names through a specific extraction function. Second, parties should be translated to legal entities with the same name and the constant type "party". Finally, tenures become employments nested within the corresponding politician. The `from` and `to` attributes of a tenure correspond to the `start` and `end` attributes of the employment.

We use this example throughout this chapter to describe the semantics and implementation of our operator. Note that the example maps a plain relational schema into a nested schema with foreign keys. Further, value correspondences may connect more than two attributes and they may use transformation functions and constants. Formally, we define value correspondences as follows.

Definition 4.1 (Value correspondence)**.** Let $\mathcal{S}$ and $\mathcal{T}$ be two relations with schemata $\mathbf{S} = sc(\mathcal{S})$ and $\mathbf{T} = sc(\mathcal{T})$ respectively. A *value correspondence* v is a function $v\colon 2^{\mathbf{S}} \to 2^{\mathbf{T}}$ that maps one or more attributes values of the source relation $\mathcal{S}$ to one or more attribute values of the target relation $\mathcal{T}$.

Tuple-generating dependencies

In data exchange theory [Fagin et al., 2005a], value correspondences and integrity constraints are expressed as tuple-generating dependencies (TGDs). As it is convention in data exchange, we use propositional logic to denote the dependencies. In the following, we first formally define them and then show the dependencies for the example.

Definition 4.2 (Tuple-generating dependency)**.** Let $\mathbf{S}$ and $\mathbf{T}$ be two schemata, $\mathbf{x} \subseteq \mathbf{S}$, $\mathbf{y} \subseteq \mathbf{T}$, and $\varphi_{\mathbf{S}}$ and $\psi_{\mathbf{T}}$ some logic clauses over $\mathbf{S}$ and $\mathbf{T}$ respectively. A *tuple-generating dependency* t is a logic statement $t\colon \forall \mathbf{x}\colon \varphi_{\mathbf{S}}(\mathbf{x}) \to \exists \mathbf{y}\colon \psi_{\mathbf{T}}(\mathbf{y})$, which defines that for

(d_1)	$\forall i, n\colon Parties(i, n)$	$\rightarrow$	$\exists L\colon LegalEntity(L, n, \texttt{"}party\texttt{"})$
(d_2)	$Tenures(i_1, i_2, f, t)$	$\rightarrow$	$\exists L, P\colon Employment(L, f, t, P)$
(d_3)	$Politicians(i, n) \wedge fn(\texttt{"}extract\texttt{"}, n, [g, l])$	$\rightarrow$	$\exists P, B\colon Person(P, g, l, B)$
(d_4)	$Politicians(i, n) \wedge Tenures(i, i_2, f, t) \wedge fn(\texttt{"}extract\texttt{"}, n, [g, l])$	$\rightarrow$	$\exists L, P, B\colon Person(P, g, l, B) \wedge Employment(L, f, t, P)$
(d_5)	$Parties(i, n) \wedge Tenures(i_2, i, f, t)$	$\rightarrow$	$\exists L, P\colon LegalEntity(L, n, \texttt{"}party\texttt{"}) \wedge Employment(L, f, t, P)$
(d_6)	$Politicians(i_1, n_1) \wedge Parties(i_2, n_2) \wedge Tenures(i_1, i_2, f, t) \wedge fn(\texttt{"}extract\texttt{"}, n_1, [g, l])$	$\rightarrow$	$\exists L, B, P\colon Person(P, g, l, B) \wedge LegalEntity(L, n_2, \texttt{"}party\texttt{"}) \wedge Employment(L, f, t, P)$
(d_7)	$Employment(l, s, e, p)$	$\rightarrow$	$\exists G, L, B, N\colon Person(p, G, L, B) \wedge LegalEntity(l, N, \texttt{"}party\texttt{"})$

Table 4.1: Tuple-generating dependency for mapping in Figure 4.1. Top: Source-to-target TGDs. Bottom: Target TGDs.

each tuple on the left-hand side, there must be a tuple on the right hand side with corresponding values.

We mainly distinguish between source-to-target TGDs, where **S** is the source schema and **T** the target schema, and target dependencies, where both **S** and **T** are the target schema. The former type formalizes the notion of value correspondences between source and target schemata, while the latter type defines inclusion dependencies, specifically foreign keys. Table 4.1 lists the seven TGDs for the exemplary value correspondences of our running example (see Figure 4.1). In the following, we briefly describe the TGDs.

The first TGD $\forall i, n\colon Parties(i, n) \rightarrow \exists L\colon LegalEntity(L, n, \texttt{"}party\texttt{"})$ describes the mapping of parties to legal entities. More specifically, it defines that for each party with name n there must be a legal entity with arbitrary ID, the same name, and the type "party". According to a common convention in the data exchange community, we represent free variables on the right hand side of the statement with uppercase letters (e.g., L = legal entity ID) and remove the universal quantifier on the left-hand side in the remainder of the chapter.

In the second TGD $Tenures(i_1, i_2, f, t) \rightarrow \exists L, P\colon Employment(L, f, t, P)$, we translate tenures into employment. For the sake of presentation, we model the employment as a separate relation with a foreign key on the persons. Indeed, we follow the common approach that divides nested structures in separate relations and performs nesting in a post-processing step. For this TGD, the foreign key to the legal entity and person are free variables. The third TGD $Politicians(i, n) \wedge fn(\texttt{"}extract\texttt{"}, n, [g, l]) \rightarrow \exists P, B\colon Person(P, g, l, B)$ transforms politicians into persons by splitting the name n into the three parts with a clause for a function call. The person ID and biography can take arbitrary values.

Until now, the rules operate on individual records and are primarily used for unrelated records, for example a partyless politician. The next three rules use the foreign key relationship of the source schema (or more generally any join relationship). The

fourth TGD $Politicians(i, n) \land Tenures(i, i_2, f, t) \land fn(\texttt{"extract"}, n, [g, l]) \rightarrow \exists L, P, B\colon Person(P, g, l, B) \land Employment(L, f, t, P)$ handles the foreign key relationship between politicians and tenures. In comparison to a simple combination of rule d_2 and d_3, the person ID P and the employment foreign key must have the same value. Similarly, rule d_5 specifies that the foreign key of employment must refer to a valid legal entity. Lastly, the sixth TGD combines all prior rules.

The seventh TGD $d_7\colon Employment(l, s, e, p) \rightarrow \exists G, L, B, N\colon\ Person(p, G, L, B) \land LegalEntity(l, N, \texttt{"party"})$ defines the target dependency between employment, person, and legal entity. The rule states that for every employment, there must be a corresponding person record and legal entity record with matching IDs.

Obviously, the three rules d_2, d_4, and d_5 are less relevant for our running example to find cases of nepotism, because they map records with incomplete relationships to the target schema. Since we want to enrich the individual entities of separate datasets with data integration, extracting only partial records would not improve the final data quality. In contrast, for partyless politicians TGD d_3 is necessary as well as TGD d_1 for memberless parties (for example, all members have been filtered with the `scrub` operator). Nevertheless, for other scenarios and statistical analyses, even incomplete tenure records need to be correctly transformed into the target schema, so that the three rules d_2, d_4, and d_5 are still valid. Lastly, in our setting, we can easily filter incomplete tenure records with a preceding `scrub` operator. In the remainder of this chapter, we continue to discuss the operator with the complete set of TGDs, which is also necessary for optimization.

Data exchange settings and core solutions

Having obtained an intuition on TGDs, we can now formally define the semantics of our `data map` operator. A full configuration of the operator requires a complete *data exchange setting* [Fagin et al., 2005b].

Definition 4.3 (Data exchange setting)**.** We define the *data exchange setting* as a tuple $(\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$ describing the source and target schemata $\mathbf{S}$ and $\mathbf{T}$ as well as the source-to-target and target dependencies Σ_{st} and Σ_t.

We want to find a solution to the *data exchange problem*: Given an actual dataset $\mathcal{S}$, we want to create a dataset $\mathcal{T}$, such that $\langle \mathcal{S}, \mathcal{T} \rangle$ satisfies Σ_{st} and $\mathcal{T}$ satisfies Σ_t. Further, the solution should be *universal*, such that no other solution retains more information than our solution. Ideally, it should also be minimal to avoid redundancy.

Definition 4.4 (Universal solution)**.** The dataset $\mathcal{T}$ is a *universal solution* to a given data exchange setting $(\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$ and input relation $\mathcal{S}$, if there is a homomorphism h that can generate all other solutions $\mathfrak{T}$ from $\mathcal{T}$; $\forall \mathcal{T}' \in \mathfrak{T}\colon \exists h\colon h\colon \mathfrak{T} \rightarrow \mathcal{T}'$.

A *universal* solution obtained with the chase algorithm [Maier et al., 1979] usually results in a high degree of redundancy. Figure 4.2 shows that we can generate up to two redundant tuples in the person relation for the TGDs of our running example (see

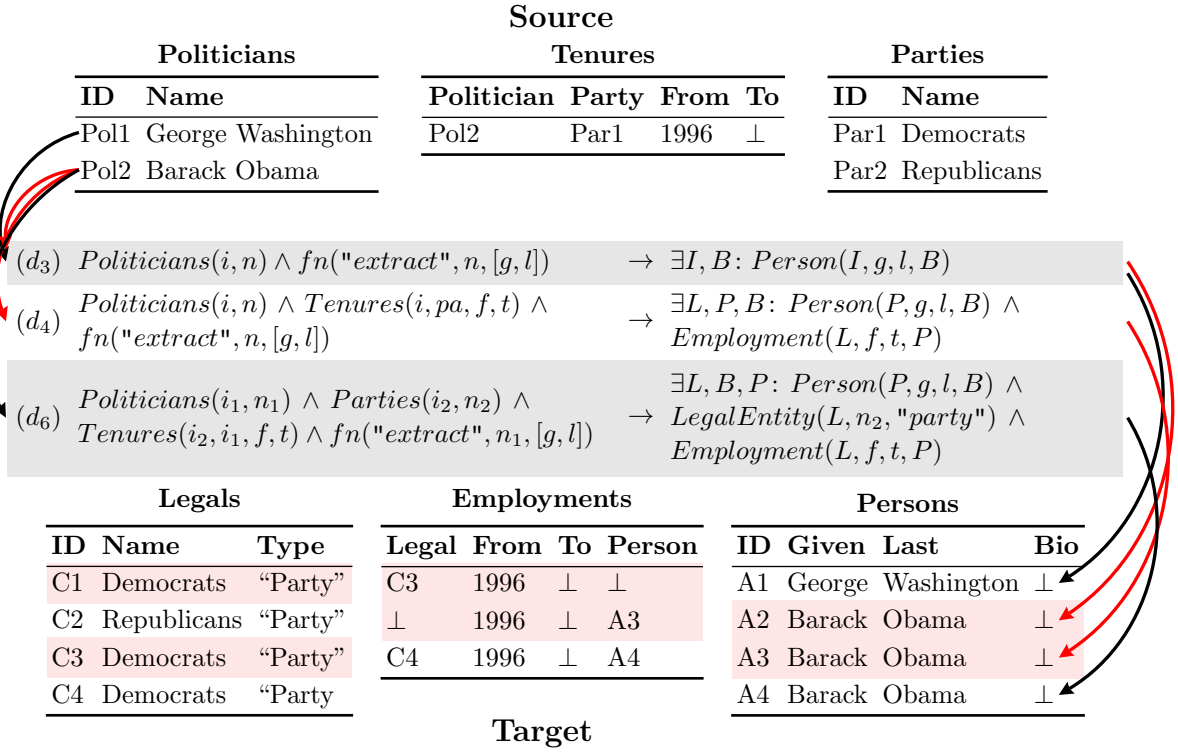


Figure 4.2: Tuples generated from an exemplary source relations (top) with the TGDs from running example (see Table 4.1). Highlighted rows indicate redundancy and arrows show the corresponding TGDs for the generated records.

Table 4.1). A person can be created with d_3, d_4, and d_6 from the same source tuple. In most cases, the most specific rule d_6 generates the desired tuples with the correct foreign keys (e.g., Barack Obama (A4), Democrats (C4), with Employment (A4, C4)). However, exceptions, such as partyless politicians (Pol1), require the first five rules for a universal solution. For fully linked source tuples, these additional rules cause redundancy, because they can be applied to more than one source tuple and generate tuples into the same relation. In general, an overlap on the left-hand sides of TGDs creates systematic redundancy. For our data integration scenarios, we need a better solution that addresses this redundancy. Fagin et al. [2005b] propose *core solutions* that specifically minimize the number of generated tuples. In this thesis, we consider only core solutions that are also universal.

Definition 4.5 (Core solution)**.** The dataset $\mathcal{T}$ is a *core solution* to a given data exchange setting $(\mathbf{S}, \mathbf{T}, \Sigma_{st}, \Sigma_t)$ and input relation $\mathcal{S}$, if it is a universal solution and there is no smaller universal solution.

Traditionally, core solutions are obtained by generating any universal solution (for example with chase) and remove redundancy in a second step. However, redundant tuples often differ in some fields. For example, in Figure 4.2 the redundant employment has a `null` value in the person field, which it typical for TGDs with non-maximal joins

(d'_1)	$Parties(i,n) \;\wedge$ $\neg Tenures(i_2,i,f,t)$	$\rightarrow\ \exists L\colon LegalEntity(L,n,\text{“}party\text{”})$
(d'_2)	$Tenures(i_1,i_2,f,t) \;\wedge$ $\neg(Parties(i_2,n_2) \vee Politicians(i_1,n_1))$	$\rightarrow\ \exists L,P\colon Employment(L,f,t,P)$
(d'_3)	$Politicians(i,n) \wedge fn(\texttt{"}extract\texttt{"},n,[g,l]) \wedge$ $\neg Tenures(i_2,i,f,t)$	$\rightarrow\ \exists I,B\colon Person(I,g,l,B)$
(d'_4)	$Politicians(i,n) \wedge Tenures(i,i_2,f,t) \wedge$ $fn(\texttt{"}extract\texttt{"},n,[g,l]) \wedge \neg Parties(i_2,n_2)$	$\rightarrow$ $\exists L,P,B\colon Person(P,g,l,B) \wedge$ $Employment(L,f,t,P)$
(d'_5)	$Parties(i,n) \wedge Tenures(i_2,i,f,t) \wedge$ $\neg Politicians(i_2,n_2)$	$\rightarrow$ $\exists L,P\colon LegalEntity(L,n,\texttt{"}party\texttt{"}) \wedge$ $Employment(L,f,t,P)$

Table 4.2: Rewritten tuple-generating dependency for Table 4.1.

on the left-hand sides. Therefore, these approaches usually imply a polynomial runtime to the number of tuples [Gottlob and Nash, 2008] and are thus infeasible for our setting where we want to scale to large dataset.

We use the elegant solution to find core solutions introduced by Mecca et al. [2009], which rewrites TGDs to avoid creating systematically redundant records for overlapping TGDs. In particular, they rewrite the left-hand side of TGDs, such that for each record in the source relations only one rule is applicable. Table 4.2 lists the rewritten first rules for our running example. For instance, the TGD d'_1 has an additional negation clause, which excludes all parties that have a tenure. Therefore, TGD d'_1 is applied only to parties without any members (they could have been filtered in a preceding `scrub`). In theory, a third clause would be needed to also exclude all parties that have a tenure and a valid politician record (rule d_6), but the second clause generalizes this clause already, so that the third clause is removed by the rewrite engine.

Rules d_2 – d_5 are rewritten similarly: TGD d'_2 excludes all tenures that have a join partner in either the party or politician relation, TGD d'_3 retains only partyless politicians, TGD d'_4 selects only tenures with politicians and without party, and TGD d'_5 retains those tenures with party but without politicians. It is noteworthy that all rewritten TGDs can be directly expressed as relational algebra to create a core solution completely with database operators. Further, rewriting the TGDs can be done entirely on algebraic properties without using any instance data; ideal for our distributed setting.

Skolem functions

Nevertheless, denormalized input relations constitute a second source of redundant tuples. For example, for each democrat, rule d_6 creates a new democrat party linked only to the new person. Naïvely, such duplicate entries would need to be consolidated into one entry and foreign keys appropriately updated. Both duplicate removal and foreign key consolidation would again result in polynomial runtime. Therefore, Popa et al. [2002] proposed to use skolem functions to generate foreign keys. For example, the legal entity ID for parties can be generated with the name of the party. Hence, all employment records refer to the same legal entity ID for the same party. Further, party records may

```
1  $fbPersons, $fbLegals = map data of
       $pol in $politiciansScrubbed, $t in $tenureScrubbed,
       $par in $partiesScrubbed
2    where $pol.id == $t.politician and $t.party == $par.id
3    into [
4      entity $fbLegals identified by $fbLegals.name with {
5        name: $par.name,
6        type: "party",
7        original: [$par.id],
8      },
9      entity $fbPersons with {
10       firstName, lastName: extractName($pol.name),
11       employments: [{
12         legalEntity: $fbLegals.id,
13         startYear: $t.from,
14         endYear: $t.to,
15         original: [$t.id],
16       }],
17       original: [$pol.id],
18     }
19 ];
```

Listing 4.1: Data map operator for `Freebase` in Meteor.

now be easily consolidated with any set-generating operator (e.g., union), because all records are exact duplicates. However, it is important to give the operator the opportunity to calculate their own IDs to achieve this scalable consolidation. Before showing how the `data map` operator uses these techniques to create a query of Sopremo base operators, we first describe how the user specifies the data exchange settings in Meteor.

4.2 Data exchange settings in Meteor

The `data map` operator ingests an arbitrary number of inputs and creates a variable number of outputs. Hence, we designed a Meteor syntax that helps users to handle the large number of dataset variables. Further, Meteor users should have no problem to perform the actual specification of dependencies.

Listing 4.1 shows the operator invocation for the running example. The first line declares the two output variables and defines the three input datasets. Next the script defines a join graph over the input relations that is later used to generate the left-hand sides of the TGDs. In our example, the join graph models only the foreign keys. The remainder of the Meteor script specifies the value correspondences with two invocations of the sub-operator "**entity**".

The line "**entity** $fbLegals *identified by* $fbLegals.name" starts with the specification of the correspondences for the legal entities. In Meteor, it is possible to reference output variables of operators inside of the operator specification. For the `data map` op-

erator, the output variables unambiguously reference the output dataset to which the correspondences belong. The "*identified by*" keyword optionally defines unique keys over one or more target attributes. The script proceeds with the value correspondence "name: $par.name" and a constant value assignment "type: "party"". Note that, all entities implicitly receive a primary key `ID`, where we generate values with skolem functions using the target values or unique keys if given. Further, in our integration scripts we also store the original IDs "original : [$par.id]" to build lineage information.

The value correspondences for persons in Line 9 demonstrate additional functionalities. A user-defined function "extractName" splits the name of the politician into first and last name. Further, we define the nested `employments` attribute with the usual Meteor syntax. Because the field `legalEntity` references the ID field of the output with "legalEntity : $fbLegalEntities.id", we implicitly specified a foreign key. The array notation around the employment definition indicates a multi-valued attribute.

Summarizing, the Meteor syntax of the `data map` operator should be easily understandable for Meteor users. Value correspondences strongly resemble the specification of the transformation expressions. Foreign key relationships on the source side mimic the syntax of joins, because users actually describe a join graph. Foreign keys on target side are implicitly defined by assigning the primary key `ID` of output variables.

4.3 Executing data exchange transformation

To execute the data exchange transformation, we tightly integrate ++Spicy [Marnette et al., 2011], a second-generation, open-source[1] mapping tool. Figure 4.3 summarizes the interaction of our `data map` operator and ++Spicy (Spicy for the remainder of the thesis). The operator interprets the Meteor specification of the user and extracts Spicy-conform data exchange settings. Additionally, it saves advanced expressions, such as function calls and conditional expressions, which cannot be translated to Spicy.

Spicy compiles source-to-target and target TGDs from the data exchange settings and applies rewriting techniques [Mecca et al., 2009] to reduce redundancy. It then uses the rewritten TGDs to create the core-generating TGDs, which additionally resolve self-joins. The Spicy tool either directly applies the operators to the data sources to generate the resulting datasets or generates SQL or XQuery queries from the core TGD.

For the `data map` operator, we developed our own Sopremo query compiler. Algorithm 4.1 lists the main algorithm for generating a Sopremo query with a given set of core TGDs. The algorithm can be divided into two parts: First, it creates intermediate queries to extract non-overlapping records from the inputs using the left-hand sides of the TGDs. Second, for each target type, it transforms the records from the input records into the target schema and removes redundancy. We discuss both parts in the following.

The first part of the algorithm analyzes the left-hand side of each TGD and generates up to three base operators for each source-to-target TGD. It first joins the relevant sources and retains the important fields in a *positive query* (nomenclature from Spicy). It

[1] `http://www.db.unibas.it/projects/spicy/`

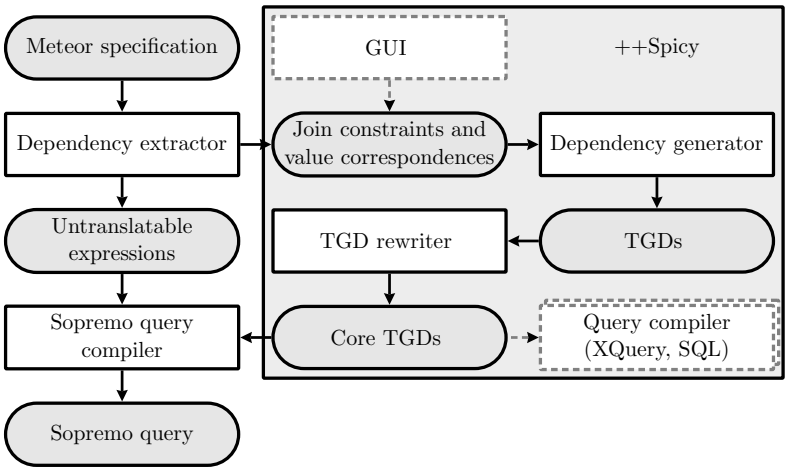


Figure 4.3: Interaction of Sopremo's `data map` operator and ++Spicy.

Input : Inputs $\mathcal{I}$, Core TGDs $\mathcal{D}$
Output: Sopremo query
`/* Extract non-overlapping records from data sources */`
Intermediate queries $\mathcal{Q} \leftarrow \emptyset$;
foreach TGD d $\in \mathcal{D}$ **do**
  Generate positive query with inputs $\mathcal{I}$ from non-negated clause of d;
end
foreach TGD d $\in \mathcal{D}$ **do**
  if d contains negations **then**
    Find most specific positive queries for each negation;
    Generate negative query with anti-join from positive queries;
  end
  $\mathcal{Q} \leftarrow \mathcal{Q} \cup$ Generate transformation of negative or positive query to initialize missing values to satisfy target TGDs;
end
`/* Perform actual transformation */`
$\mathcal{T} \leftarrow$ find all types of target entities;
foreach target entity $t \in \mathcal{T}$ **do**
  foreach TGD $d \in \mathcal{D}$ that generates t **do**
    Find intermediate query $q \in \mathcal{Q}$ corresponding to TGD d;
    Generate extended target transformation to transform values from intermediate query q to target schema;
  end
  Generate union to remove exact duplicates from all transformations;
  Recursively nest sub-target entities into t;
end
Return query with top-level target entities as sinks;

Algorithm 4.1: Sopremo query compiler for a `data map` operator.

then removes all records that are generated by more specific joins in a *negative query* to avoid systematic duplicates. Finally, for all TGDs with a non-maximized join, default values are added with a `transformation` for those attributes that have not been joined, so that all target TGDs (e.g., d_7 in our example) are satisfied and in the end all intermediate results share the same input schema. For example, consider the TGD $Parties(i_a, n_a) \land Tenures(i_o, i_a, f, t) \land \neg Politicians(i_o, n_o) \rightarrow \exists L, P\colon LegalEntity(L, n_a, \texttt{"}party\texttt{"}) \land Employment(L, f, t, P)$. The positive query is a join over `Tenures` and `Parties`, while the negative query is an anti-join over the result of the positive query and the positive query of TGD d_6 to remove all tenures that also link to politicians. The `transformation` then adds default values for all politicians attributes.

The second part of the algorithm extracts the target entities. After all records with their join partners are extracted and optionally enriched with default values, they all share the same intermediate schema and can be easily divided into separate records for the target entities. First, the algorithm determines the target entities from the TGDs. For each target type, it collects the relevant TGDs and creates a `transformation` from the corresponding intermediate queries. The most challenging part is to create and reference `IDs` with skolem functions by choosing the determining fields carefully. Nested relations additionally need the `ID` of the parent record, for example `employments` needs the `ID` of the nesting person. These nested relations are recursively nested into the parent relations by `grouping` all children by the parent `ID` and joining them with the immediate parent entry. The grouping and join operation are repeated from the leaf to the root node of a nested target schema until all records have been merged into a single dataset.

Figure 4.4 shows the Sopremo query generated for our running example. We can see that for the short Meteor script, already 28 base operators are created and properly connected. In general, most of the operators are generated by the first part of the algorithm, where records are joined and prepared for the transformation. Only few operators perform the actual `data map` and the recursive nesting of embedded relations.

4.4 Optimization of the data map operator

Similar to the `scrub` operator, the `data map` operator decomposes into base operators, which have well-known reorder rules. In contrast to the `scrub` operator, we do not have a mingled `selection` and `transformation`, so that reorderings of a `data map` operator or parts thereof are much easier. Nevertheless, we can estimate the cost and cardinality in a simpler yet more accurate way if we also consider the operator as a whole.

4.4.1 Reordering

Because of the large diversity of the involved operations, it is unfortunately not possible to describe the reorder capability of the `data map` operator through Presto [Rheinländer et al., 2013] properties. The operator is not commutative, associative, or idempotent, and changes the schema completely. In fact, the semantics of these properties are not properly defined for an operator with an arbitrary number of inputs and outputs. Nevertheless, we

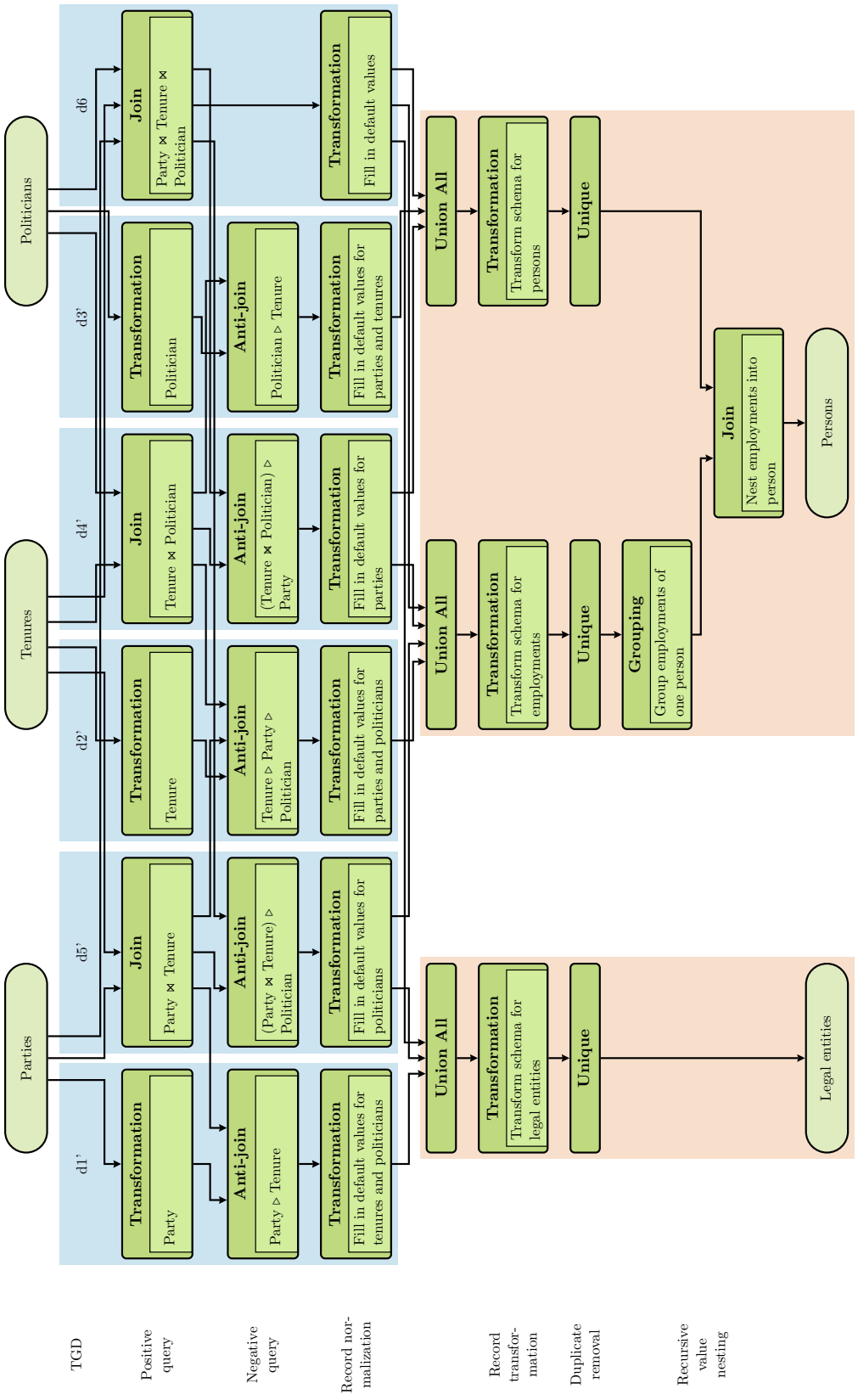


Figure 4.4: Expanded Sopremo query for `data map` operator of running example.

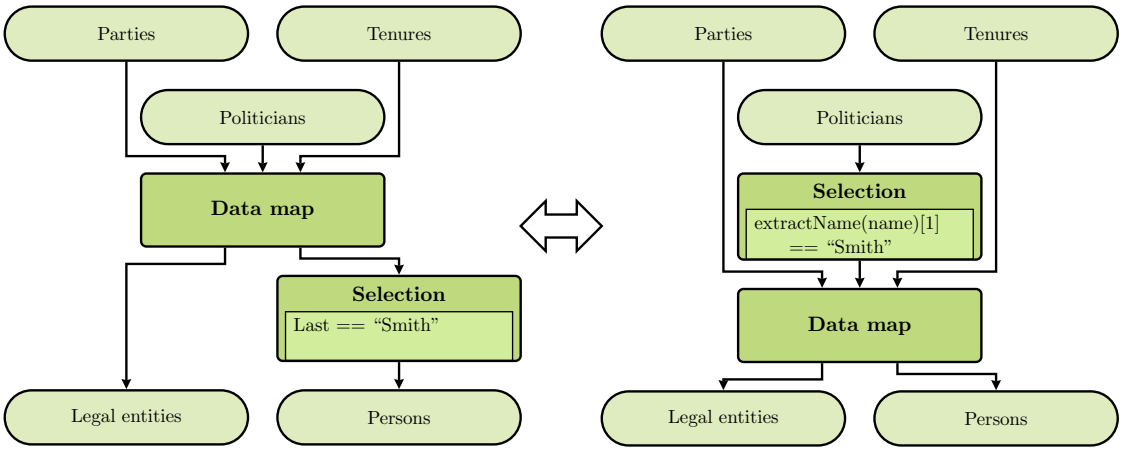


Figure 4.5: Eager-lazy pushing of a selection through the `data map` operator.

can find useful reordering rules with `selection`s and `transformation`s, which greatly help to decrease network traffic and overall runtime.

Pushing a selection

We can push a `selection` operator through the `data map` operator towards the data sources, if its condition does not correlate attributes originating from records of different sources. Consider that a user is interested in persons with last name "Smith" (i.e., last = "Smith") after the `data map` operator of our running example. Then, the `selection` can be pushed through the `data map` operator towards the `politicians` as shown in Figure 4.5 by adjusting the selection condition with the `data map` configuration. This reordering implies the following statements:

- All politicians that do not satisfy "extractName(name)[1] == 'Smith'" are removed from the input of the `data map` operator.
- Hence, no person with a last name other than Smith is created. All the Smiths are correctly created through d'_4, d'_3, and d_6.
- Tenures referring to Smiths are correctly joined in d'_4 and in the end properly nested. Other tenures are transformed to employments with d'_5 and d'_2, but are discarded in the nesting phase, because their join partner is missing.
- All parties are transformed to legal entities. The parties with Smiths through d_6 and those without any Smiths with d'_5. Memberless parties are still correctly transformed with d_1.
- Most importantly: The referential integrity still holds after reordering.

In general, pushing a `selection` through `data map` works similarly to pushing a `selection` through a join. As long as the `selection` does not correlate values from different sources, the `selection` can be pushed through the `data map` operator.

Note, that this rule extends the reordering possibility of the base operators. In particular, applying the basic reordering rules to the completely expanded operator (see Figure 4.4), we can see that the `selection` can be pushed through the nesting `join` and

preceding `unique` to the `transformation`. At this point, we need to adjust the selection condition to compare the result with the transformation function on the original record. The rewritten `selection` can be further pushed through the `union all` towards the partial queries of d'_4, d'_3, and d_6. However, pushing it further upwards is impossible with the basic reordering rules, because the `transformation`s of the partial queries pipe their output into three different `union all` and pushing it through the `transformation`s would indeed change the intermediate results.

Our new reorder rule exploits the property of *universal solutions*, which guarantees that all information is preserved. Therefore, the six partial queries for the TGDs address *all* different join cases, so that even though we remove one join partner, the remaining partners are still correctly translated through other intermediate queries. Therefore, we can push the `selection` through the entire `data map` operator.

Pushing a projection

When attributes generated by a `data map` operator are projected out in a later stage, we can simplify the `data map` transformation itself and the partial queries for the TGDs. In fact, we could simply restrict the right-hand side of the TGDs to the values in the `projection` for the corresponding output relation and rerun the Sopremo query compiler.

Source attributes that become obsolete with a pushed `projection` are automatically removed in positive queries, such that the `projection` has been implicitly pushed through the `data map` operator. For example, projecting the employments on the `start` and `legal` field (leaving out the `end` field) changes TGD d_2 to $Tenures(i_1, i_2, f, t) \rightarrow \exists L, P\colon Employment(L, f, \not{t}, P)$. Therefore, the `to` field becomes irrelevant for the `data map` operator and is removed in the `join` transformation of the positive query of d_2. In contrast, if the `first` name of a person is removed through a `projection`, the `name` attribute of politician is still retained in the positive queries, because it is still needed to extract the last name.

4.4.2 Cost and cardinality estimation

The runtime of the `data map` operator mainly depends on the initial `joins` of the positive queries, the `unique` operators to deduplicate the output, and the nesting operators. The `anti-joins` of the negative queries are relatively cheap, because they can reuse the partitionings of the previous `join` operators and thus avoid costly network traffic. Therefore, for a good runtime estimation it is important to know the join selectivity of the individual components of the join graph over the input relations. Here, we can rely on the `join` estimation of the Stratosphere framework, because such essential operators should be estimated well. The `join` cardinality also directly correlates to the runtime costs of `unique` and nesting.

Nevertheless, we can give an upper bound of the **number of resulting records**. Even with a high join selectivity, we know through the properties of the *core solution* that not more records per output are generated than are originally present. Thus, the theoretic maximum number of records for each output is the sum of the number of tuples for each

input that contributes to it. For example, a legal entity can be generated through a party or a tenure record, but not through a politician: $T(\text{legals}) \leq T(\text{parties}) + T(\text{tenures})$.

Summary This chapter presented the `data map` operator that transforms datasets from a given source schema to a specified target schema. The `data map` operator builds upon the large body of data exchange research, which we formally introduced with tuple-generating dependencies, data exchange settings, and core solutions. We showed how Meteor users can easily define the necessary information and explained how our operator interacts with ++Spicy to rewrite the tuple-generating dependencies. We described an algorithm to translate the rewritten dependencies into a query consisting only of base operators. In the end, we developed some reordering rules by using the theoretic properties of our approach and sketched cost and cardinality estimation.

We use the `data map` operator to transform the records from our input relations to our integrated schema. We can then use the `duplicate detection` operator presented in the next chapter to find different representations of the same-real world entity. Because the `data map` operator maintains referential integrity, such duplicate records help to enrich the representation of entities with more relationships to other entities.

5

Detecting Duplicates in Datasets

Duplicates in a dataset are multiple representations of the same real-world entity and constitute a major data quality problem within one dataset. The consequences range from unsatisfied employees and customers to incorrect analyzes that lead to wrong business decisions.

Detecting duplicates is a core activity of every data cleansing and integration process. At the same time, it typically consumes many resources, so that a good runtime performance mandates an efficient and scalable implementation. This chapter presents the `duplicate detection` and `record linkage` operators, which identify fuzzy duplicates within a single dataset or across two datasets respectively.

In data integration projects, both operators can be used to connect individual datasets by finding extensional overlap and subsequently to create integrated datasets with added value. The `duplicate detection` operator requires the datasets to be schematically aligned, whereas the `record linkage` operator may ingest schematically heterogeneous datasets, but only two datasets at a time. Data maintainers can improve the data quality of one data source with the `duplicate detection` operator by reviewing the duplicate records or automatically fusing them with our `fusion` introduced in Chapter 7.

This chapter defines basic concepts and the semantics of both operators and highlights how the integration in Meteor allows users to express powerful matching rules. Further, we present the parallel implementation of the three most popular variations in Stratosphere. We conclude the chapter by integrating the two operators into the optimizer.

5.1 A model for duplicate detection

In this section, we start with the traditional definition of duplicates and later generalize them. For one dataset, we define duplicate pairs as follows.

Definition 5.1 (Duplicate pair)**.** Let $\mathcal{R}$ be a relation. A *duplicate pair* $\langle r_1, r_2 \rangle$ with $r_1, r_2 \in \mathcal{R}$ denotes two different representations of the *same real-world entity*. We denote the equivalence relation with $r_1 \sim r_2$.

The equivalence relation is necessarily reflexive, symmetric, and transitive, since they associate a record with the same real-world entity. Therefore, the relation partitions the dataset into disjoint clusters.

Definition 5.2 (Duplicate cluster)**.** Let $\mathcal{R}$ be a relation. A *duplicate cluster* $c \subseteq \mathcal{R}$ contains all representations of the same real-world entity. We denote the set of all clusters of $\mathcal{R}$ with $\mathcal{C}_\mathcal{R}$. Formally, $\mathcal{C}_\mathcal{R}$ is a partitioning of a dataset $\mathcal{R}$ over the equivalence relation $\sim$, i.e., the quotient set $\mathcal{R}/\sim$: $\mathcal{C}_\mathcal{R} = \{C \subseteq \mathcal{R} \mid \forall r_i, r_j \in C \colon r_i \sim r_j\}$.

Duplicate detection is the challenging task of finding duplicate clusters in an *efficient* and *effective* way. Naïvely, we need to look at each pair of records, which renders duplicate detection even on medium-sized dataset impractical. Further, for each pair we must decide if they represent the same real-world entity; a non-trivial decision, since we have only limited information with potential data quality issues, such as typographic errors or inconsistent formatting. Before we outline solutions that address efficiency and effectiveness, we generalize from the rather strict interpretations of duplicates.

5.1.1 Generalization of duplicate detection and record linkage

Duplicate detection may also be applied to duplicate-free datasets (e.g., real-world entities have at most one representation). In this case, the user defines an equivalence relation independent of the real-world entity. For example, the user wants to group all recipients of earmarks by their state and country and calculate certain statistics. Without a preceding scrubbing operator, typographic errors result in incomplete groups and misleading statistics when applying a normal grouping operator. However, an equivalence relation $r_1 \sim_{sc} r_2 \Leftrightarrow r_1.state \sim_s r_2.state \wedge r_1.country \sim_c r_2.country$ allows the user to use the duplicate detection operator as a generalized *similarity group-by* operator [Silva et al., 2009]. We can thus generalize duplicate detection for equivalence relations that do not necessarily reflect equivalency of the corresponding real-world entities.

Definition 5.3 (Duplicate detection)**.** Let $\mathcal{R}$ be a relation. *Duplicate detection* partitions $\mathcal{R}$ according to some equivalence relation $\sim$.

A more appropriate, neutral name for duplicate clusters in this thesis would be *equivalence classes* in regard to the used equivalence relation. However, the use of "duplicate" is so wide-spread in related work that we retain the nomenclature to improve the readability. It is important to remember that a duplicate cluster means records with equivalent values in their intensional overlap.

When performing *record linkage* across two datasets, two scenarios are possible. First, the two datasets may be duplicate-free, but mutually contain information about the same real-world entities. In that case, the negatively conotated term duplicate implicates data quality issues, where we actually link two datasets of high quality. Further, the result of linking two duplicate-free datasets can only be disjoint pairs. Second, one or both datasets may contain duplicates. Hence, we receive a cluster of duplicates, where each record in a cluster refers to the same entity. If there is one clean dataset, there can only be one record of that dataset per cluster.

Additionally, similar to the generalized duplicate detection use case, a user may want to link records about different concepts of real-world entities with a specific equivalence relation. For example, a user wants to link `Freebase` politicians to earmarks without prior schema alignment. In that case, the `record linkage` operator acts as a generalized *similarity join* operator [Silva et al., 2010] with an equivalence relation over the politician p and an earmark e: $p \sim_{fl} e \Leftrightarrow p.first \sim_f e.sponsorFirst \wedge p.last \sim_l e.sponsorLast$.

Definition 5.4 (Record linkage)**.** Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be a relation. *Record linkage* returns all pairs of equivalent records from $\mathcal{R}_1 \times \mathcal{R}_2$ according to some equivalence relation $\sim$.

For ease of presentation, we also refer to matched entities or linked records with "duplicates" for the `record linkage`, since `duplicate detection` and `record linkage` share most concepts. In the end, a duplicate pair describes two records that are in a user-specified equivalence relation and a duplicate cluster is a set of records where each pair of records is a duplicate pair.

Batini et al. [1986] present different integration strategies for schema matching of more than two datasets, which can be generalized to data integration workflows in a straight-forward manner. They distinguish between *binary* approaches that add a new dataset in each step to a previously integrated dataset and *n-ary* strategies to integrate several datasets at once in each step. The `record linkage` operator can be used only for binary strategies, while `duplicate detection` operators may also integrate more than two datasets at once, but require prior schema alignment. Appendix A.5 gives detailed guidelines which operator to choose in different integration scenarios.

5.1.2 Sub-tasks of duplicate detection

Both, the `duplicate detection` and `record linkage` operator, perform the same subtasks, which we review primarily for the `record linkage` operator in the following.

Candidate selection

To keep record linkage feasible for large datasets, a *candidate selection* algorithm preselects promising pairs from the Cartesian product of the datasets.

Definition 5.5 (Candidate selection)**.** Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two relations. A *candidate selection cs* is a function $cs\colon \mathcal{R}_1 \times \mathcal{R}_2 \to \mathcal{R}_1 \times \mathcal{R}_2$ that selects candidate pairs for comparison from the Cartesian product of the two relations $\mathcal{R}_1$ and $\mathcal{R}_2$.

The ideal candidate selection would returns exactly the duplicate pairs. However, in reality, a candidate selection rarely works perfectly and causes misclassifications. On the one hand, it might be too strict and miss some duplicates. On the other hand, if it selects the candidates too generously, the overall time may quickly become impractical. Therefore, several *passes* of rather strict candidate selection are usually used. For the `duplicate detection` operator, $\mathcal{R}_1 = \mathcal{R}_2$ and the candidate selection should avoid reflexive and symmetric pairs.

Candidate classification

A set of heuristics carefully classifies each candidate pair as a duplicate or non-duplicate. For our operators, the candidate classification can be arbitrarily complex and use more sophisticated techniques, such as rule-based classification. The classification labels each pair of records returned by the candidate selection either as duplicate or non-duplicate.

Definition 5.6 (Candidate classification)**.** Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two relations. A *candidate classification* cc is a function $cc\colon \mathcal{R}_1 \times \mathcal{R}_2 \to \{dup, nondup\}$ that returns dup iff $r_1 \sim r_2$ for $r_1 \in \mathcal{R}_1, r_2 \in \mathcal{R}_2$ and $nondup$ otherwise.

Typically, the classification uses at least one *similarity measure* composed of several string similarity measures over multiple attributes and compares the result to a given *threshold*. In many cases, the threshold must be carefully chosen, because even small changes may have a large impact on the result.

Definition 5.7 (Similarity measure)**.** Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two relations. A *similarity measure* is a function $sim\colon \mathcal{R}_1 \times \mathcal{R}_2 \to [0, 1]$ that returns the similarity between two records of $\mathcal{R}_1$ and $\mathcal{R}_2$, whereas 1 indicates equal content and 0 completely unequal records.

We encourage users to define several rules that jointly classify the candidates. Sanity checks in the candidate classification help to reduce incorrect classifications and ease the definition of the similarity function, since special cases have already been addressed. Distance functions often allow a more maintainable definition of such sanity checks.

Definition 5.8 (Distance function)**.** Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two relations. A *distance function* $dist$ is a function $dist\colon \mathcal{R}_1 \times \mathcal{R}_2 \to \mathbb{R}_0^+$ that returns the distance between two records of $\mathcal{R}_1$ and $\mathcal{R}_2$, whereas 0 indicates equal content and larger values more dissimilarity.

We can convert any distance function into a similarity function with a maximum distance function $dist_{max}$ that provides an upper bound for the distance for two records. Often, $dist_{max}$ depends on the length of the records. The similarity function sim_{dist} for the distance function $dist$ with maximum bounds $dist_{max}$ can be calculated with $sim_{dist}\colon r_1, r_2 \mapsto 1 - \frac{dist(r_1,r_2)}{dist_{max}(r_1,r_2)}$.

Clustering

With candidate selection and pair classification, we receive a list of pairs that in most cases is not transitively closed. For example, consider the three records r_1, r_2, r_3 with $r_1 \sim r_3$ and $r_2 \sim r_3$, but $\langle r_1, r_2\rangle$ was not declared a duplicate for three possible reasons. First, $\langle r_1, r_2\rangle$ is actually a duplicate but was not preselected by the candidate selection. Second, $\langle r_1, r_2\rangle$ is a duplicate, but the pair classification fails. Third, $r_1 \nsim r_2$ and $\langle r_1, r_3\rangle$ or $\langle r_2, r_3\rangle$ is incorrectly classified and need be removed from the declared duplicates.

The actual clustering algorithm depends on the intention of the user. For example, consider that a user links the records of two clean datasets and the candidate comparison reveals that two records of the first dataset are equivalent to a third record from the

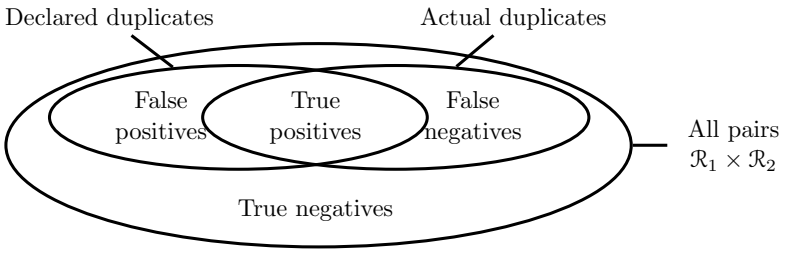


Figure 5.1: Different sets of pairs in respect to declared and actual duplicates.

second dataset for a specific similarity measure. Then, the user probably wants to retain only the pair with the higher similarity. We present two clustering algorithms in Chapter 6 that ideally support our duplicate detection and record linkage use cases. For now we treat the clustering as black box that satisfies the following definition.

Definition 5.9 (Clustering operator)**.** Let $\mathcal{D}$ a set of declared duplicate pairs over the relations $\mathcal{R}_i$. The *clustering operator* ingests $\mathcal{D}$ and returns a partitioning $\mathcal{C}$ over $\bigcup_i \mathcal{R}_i$.

5.1.3 Effectiveness

In this thesis, we primarily focus on improving the efficiency of duplicate detection and record linkage by providing scalable implementations. Nevertheless, we want to empower users to easily define candidate classification expressions that achieve high effectiveness. Therefore, we briefly define common evaluation metrics to unambiguously describe which design decisions are directed towards efficiency and effectiveness respectively.

To evaluate duplicate detection results, a set of *declared duplicate* pairs is compared with a manually labeled *gold standard* that defines the *actual duplicates*. Each pair of records over the relations $\mathcal{R}_1$ and $\mathcal{R}_2$ may lie in one of four possible sets as visualized in Figure 5.1. We refer to correctly classified duplicates as *true positives* and correctly classified non-duplicates as *true negatives*. Incorrectly declared duplicates are named *false positives* and *false negatives* describe missed duplicates. These four sets define the basic evaluation metrics for effectiveness. *Recall* represents the completeness of a duplicate detection run; *precision* measures the quality of the pair classification; and f_1 is the harmonic mean of recall and precision.

$$r = \frac{tp}{tp + fn}, \qquad p = \frac{tp}{tp + fp}, \qquad f_1 = \frac{r * p * 2}{r + p}$$

Any candidate selection algorithm trades recall for efficiency. The fewer pairs it preselects, the faster the duplicate detection run, but also the higher the probability of a false negative and lower recall. Multi-pass candidate selections preselect pairs comparably strict in each pass, but hopefully preselect every true duplicate in one of the passes.

The candidate classification usually trades off precision and recall. A strict candidate classification achieves a high precision but lower recall and a lenient classification finds more duplicates but risks some false positives. In the next section, we demonstrate how Meteor users leverage the expressiveness of Meteor to define powerful candidate classifications for a high effectiveness and especially avoid some false positives.

5.2 Detecting and removing duplicates with Meteor

The Meteor integration of the `duplicate detection` and `record linkage` operator allows users to use any technique for classifying pairs that they see appropriate. Users simply denote a Meteor expression that evaluates to a Boolean value. For example, a user may define a simple comparative expression of a composite similarity function and a threshold, while another user formulates several *matching rules* that address different cases. Additionally, arbitrary Java function calls can be embedded, which in turn could even use learned classifier models. Listing 5.1 shows a rule-based configuration of the `duplicate detection` operator to match persons extracted from `US Earmarks`, `Freebase`, and `US Congress`. We start the operator invocation by specifying the inputs and proceed with a list of conjunctive rules.

```
1 $duplicates = detect duplicates $p in
    $earmarksPersons, $fbPersons, $congressPersons
2   where
3     intDiff($p.birthDate.year) == 0 if $p.birthDate and
4     intDiff($p.birthDate.month) <= 1 if $p.birthDate and
5     ( 5 * jaroWinkler($p.firstName) +
6       5 * jaroWinkler($p.lastName) +
7       3 * jaroWinkler($p.middleName) if $p.middleName +
8       mongeElkan(&jaro, $p.employments[*].position)
9     ) / 14 > 0.8;
```

Listing 5.1: Finding duplicate politicians in three datasets in Meteor.

The first two rules represent sanity checks to reduce false positives and avoid costly string comparisons at the same time. The first rule "`intDiff($p.birthDate.year) == 0`" "`if $p.birthDate`" enforces that if a birth date is present in both records of the candidate pair, the year much be the same. The two data sources `US Congress` and `Freebase` contain the birth date for most politicians and, from our experience, this date is quite accurate. With this rule, we can avoid matching pairs of father/son records with similar names, such as George Bush. Second, in `Freebase` the month is often off by one, which might have been caused by importing data from sources that encoded months differently. Thus, the line "`intDiff($p.birthDate.month) <= 1`" "`if $p.birthDate`" allows the month to differ by one, but dismisses all other pairs.

From Line 5 on we define a weighted average similarity over the first, middle, last name, and the position titles and compare it with a threshold of 0.8. The first and last names are mandatory and we compare them with the *Jaro Winkler* similarity measure [Winkler, 1999]. For middle names, we use again Jaro Winkler if both middle names are present. The `duplicate detection` operator transparently uses the threshold as the default value in cases where the middle name is not given in both records. Lastly, we use the Monge Elkan similarity [Monge and Elkan, 1996] to leniently find the honorific (e.g., "Senator") in the positions of the `Freebase` employments, which also contain congress and senator memberships.

We could easily add more sanity checks and comparisons to the listing. For example, we may want to detect politicians that changed the last name after a marriage. In

that case, we would replicate the last rule, but remove the last name comparison and increase the threshold to avoid false positives. Further, we could add negative rules, such as "jaroWinkler($e.firstName) < 0.5". Finally, we can also incorporate external knowledge, for instance frequency distributions of names to create frequency aware-similarity measures [Lange and Naumann, 2011].

For the `record linkage` operator, we have two inputs with potential schema heterogeneity. Listing 5.2 shows a brief example that performs an extended similarity join of the sponsor of an earmark (without preceding `data map` operator) and persons from `Freebase`. The matching rules have the same syntax as before, but the similarity and distance function also take two arguments to support matching of records with different schemata. Here, we also see that the entire range of Meteor expressions can be used to create the input of similarity and distance functions.

```
1 $simJoined = link records $e in $earmarks, $p in $fbPersons
2   where levenshtein($e.sponsor, $p.firstName+' '+$p.lastName) > .9;
```

Listing 5.2: Extended similarity join with the `record linkage` operator in Meteor.

5.2.1 Candidate selection

Without a specific candidate selection configuration, both operators search *naïvely* for duplicates over the Cartesian product. To reduce the $|\mathcal{R}_1| \cdot |\mathcal{R}_2|$ comparisons especially for larger datasets, the user can configure candidate selections. In this thesis, we implement the two commonly used candidate selections *Sorted Neighborhood Method* (SNM) [Hernández and Stolfo, 1995] and *blocking*. SNM pre-sorts the dataset for a given sorting key and compares records located within a relatively small neighborhood. In Listing 5.3, we configure a `duplicate detection` and a `record linkage` operator to perform a two-pass SNM, respectively.

```
1 $duplicates = detect duplicates $e in $earmarksPersons
2   where jaro($e.lastName)...
3   sort on [$e.firstName, $e.lastName]
4   with window size 20;

5 $simJoined = link records $e in $earmarks, $p in $fbPersons
6   where levenshtein($e.sponsor, $p.firstName+' '+$p.lastName) > .9
7   sort on
8     [{$e.sponsorName: $p.firstName+' '+$p.lastName},
9     {reverse($e.sponsorName): reverse($p.lastName)}]
10  with window size 20;
```

Listing 5.3: Two-pass SNM on duplicate detection (top) and record linkage (bottom)

For the `duplicate detection` operator, we specify two passes with their sorting keys. The first pass sorts on `firstName` and the second pass on `lastName`. Further, we set the window size to 20. We similarly configure the `record linkage` operator with two passes. Because of schema heterogeneity, each pass is defined in an associated list as shown with the first sorting key "{$e.sponsorName: $p.firstName + ' ' + $p.lastName}".

Here, earmarks are sorted by the `sponsorName` and `Freebase` politicians by the concatenated name. The second pass "{reverse($e.sponsorName): reverse($p.lastName)}" causes the operator to sort the entries by the reversed names or last names, respectively.

The *blocking* technique partitions the records into smaller blocks and performs naïve duplicate detection on the smaller blocks. Users configure blocking in the same way as SNM but with the "*partition on*" keyword. Listing 5.4 specifies a calculated blocking key.

```
1 $duplicates = detect duplicates $e in $earmarksPersons
2   where jaro($e.lastName)...
3   partition on substring($e.firstName, 0, 2);

4 $simJoined = link records $e in $earmarks, $p in $fbPersons
5   where levenshtein($e.sponsor, $p.firstName+' '+$p.lastName) > .9
6   partition on
7     {substring($e.firstName, 0, 2): substring($p.name, 0, 2)};
```

Listing 5.4: One-pass blocking on duplicate detection (top) and record linkage (bottom)

5.2.2 Clustering

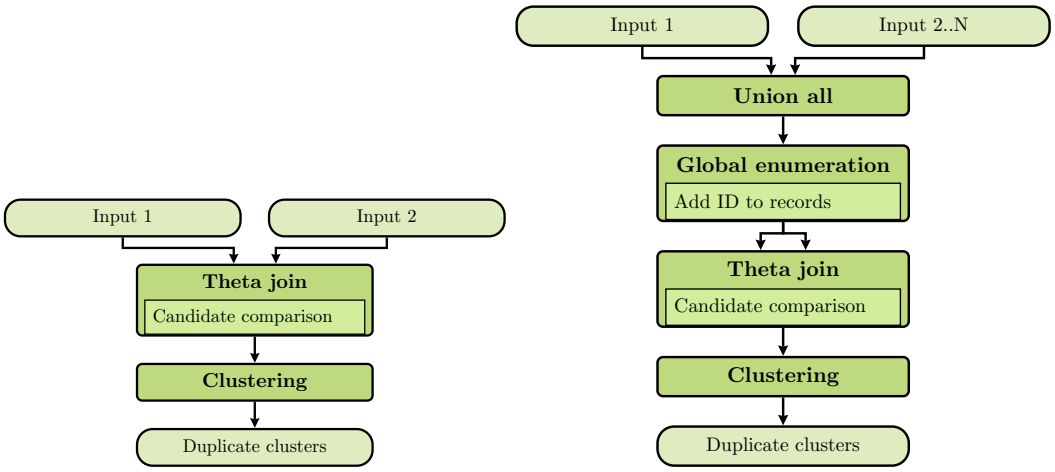
Both operators automatically perform a clustering on the merged output of the individual passes. The final result consists of disjoint clusters. Each operator has a specific default clustering operator that may be changed by the user. The duplicate detection *transitively closes* the clusters, such that a cluster contains all transitively connected records. In contrast, the record linkage applies a *stable matching* algorithm [Gale and Sotomayor, 1985], which retains only stable pairs from connected components. Both algorithms lead to sound results by either merging or removing overlapping pairs to receive disjoint clusters. We discuss the operators in detail in Chapter 6.

A user may change the default cluster with the keyword "*cluster with*" as shown in Listing 5.5. Currently, the user may choose between "`'none'`", "`'transitive closure'`", and "`'stable matching'`". For example, if a user assumes a maximum duplicate cluster size is 2, he prevents some false positives by choosing the stable matching algorithm.

```
1 $duplicates = detect duplicates $e in $earmarksPersons
2   where jaro($e.lastName)...
3   sort on [...]
4   cluster with 'stable matching';
```

Listing 5.5: Changing the default clustering method.

Developers may add new clustering algorithms in their own packages and can register them for future usage. For example, they might implement the algorithms *center* [Haveliwala et al., 2000] or *merge-center* [Hassanzadeh and Miller, 2009], which decide for each overlapping pair whether to merge or split the cluster using coherence heuristics.



(a) Record linkage (b) Duplicate detection

Figure 5.2: Implementation of naïve `duplicate detection` and `record linkage` operator with a theta-join. Duplicate detection may have multiple inputs.

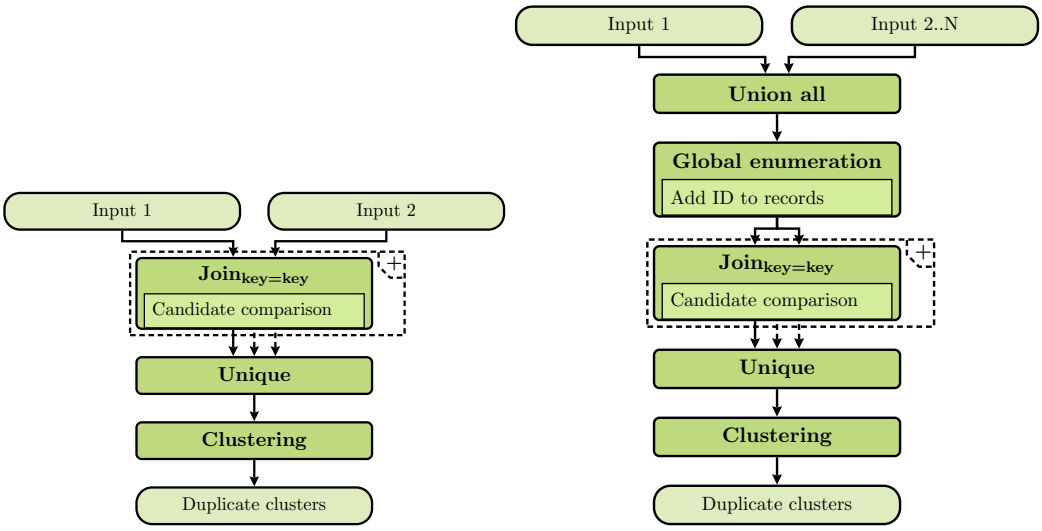
5.3 Parallel algorithms in Stratosphere

In this section, we present the parallel implementation of the three candidate selection algorithms naïve, blocking, and Sorted Neighborhood Method (SNM) for the `duplicate detection` and `record linkage` operators. Each of the candidate selection algorithms performs the candidate comparison as soon as the pair is selected. Therefore, the implementations return a list of declared duplicates. Because of multiple passes, duplicate declarations can re-occur and need to be consolidated. Further, we treat the clustering phase as a black box operator and defer the discussion to Chapter 6.

5.3.1 Naïve algorithm

When the user does not configure any candidate selection, the naïve algorithm is run by default. For both operators, the naïve algorithm basically performs a `theta-join` with the user-specified candidate comparison as shown in Figure 5.2. The implementation of the naïve `record linkage` operator in Figure 5.2a is straight-forward. We can directly use a `theta-join` (implemented as a `cross`) on the inputs to create all pairs of records from both inputs. All pairs that satisfy the join condition are emitted as duplicates and post-processed in a `clustering` operator to form consistent duplicate clusters.

In contrast, the `duplicate detection` operator in Figure 5.2b is more complex. To omit the costly and unnecessary candidate comparisons of reflexive and symmetric pairs, it performs a `global enumeration` to assign globally unique IDs to each record. The `global enumeration` operator is efficiently implemented as a `Map` that uses the task ID and a locally incremented record counter to assign unique, but non-successive numbers to the records. A self-`theta-join` on the enumerated input applies the similarity expression e' that extends the original similarity expression e by an ID comparison: e': $id_{left} < id_{right} \wedge e$. Again, the declared duplicates are post-processed with a `clustering` operator.



(a) Record linkage (b) Duplicate detection

Figure 5.3: Implementation of blocking `duplicate detection` and `record linkage` operator with an `equi-join`. Multiple passes result in multiple concurrent `equi-joins`. Duplicate detection may have multiple inputs.

The naïve algorithm implies heavy network costs, because either the first or second input of a theta-join is completely replicated to each node. Therefore, this algorithm should be used only if no good blocking or sorting key can be calculated. For the `record linkage` operator, the algorithm might be feasible if one input is much smaller than the other input. Even if users need a guarantee that all duplicates according to the similarity measure are selected, there are much better choices than the naïve algorithm. For example, Vernica et al. [2010] provide an efficient set-similarity join for Hadoop, which can be easily applied, when the similarity measure contains Jaccard or Cosine sub-similarities. We describe the Stratosphere implementation in [Heise and Naumann, 2012].

5.3.2 Blocking algorithm

The implementation of the blocking algorithm differs from the implementation of the naïve algorithm in only two points as depicted in Figure 5.3. First, instead of using a `theta-join`, we use an `equi-join` over the respective blocking keys of the inputs. For the `duplicate detection` operator, the `join` is again a self-`join`, where the preceding `global enumeration` operator and expression rewrite save again more than half of the expensive comparisons. Second, both the `duplicate detection` and `record linkage` operator may now perform multiple passes. Therefore, we replicate the `join` operator for each pass with the respective blocking key. The results of the passes deduplicated with the `unique` operator and forwarded to the `clustering` operator.

The performance gain of blocking strongly depends on the data distribution of the blocking key. Baxter et al. [2003] provide a theoretic and empirical evaluation of blocking

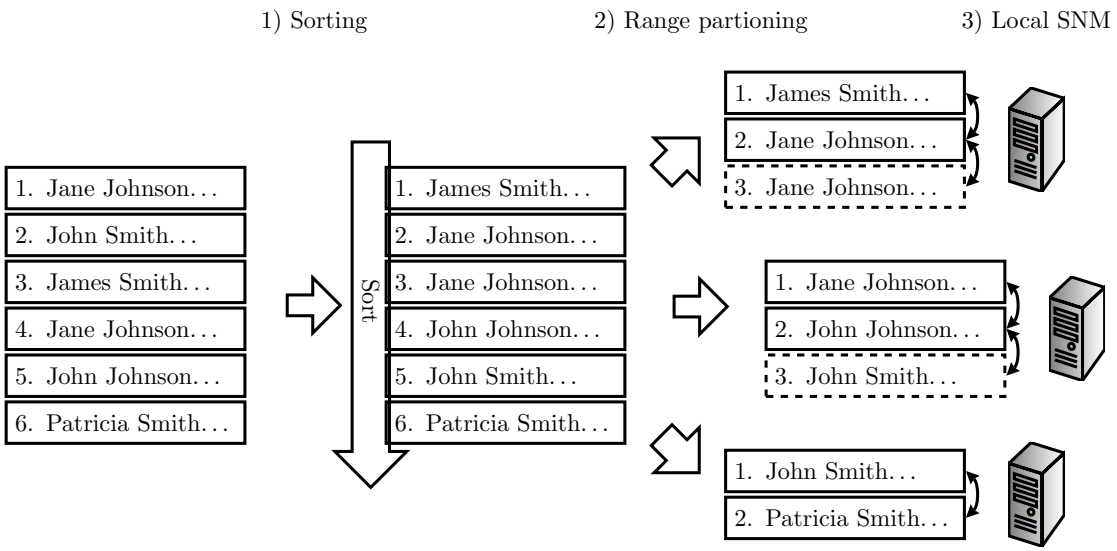


Figure 5.4: Overlapping range partitioning for the parallel sorted neighborhood method with window $w = 2$. Dashed entries indicate replicated records used to guarantee the same number of comparisons as the traditional SNM.

keys and their implication on the *pair completeness* (recall of the candidate selection with perfect similarity measure). For skewed distributions, the largest block dominates the runtime due to the quadratic number of comparisons. In our distributed setting, this effect is even amplified: While a large number of smaller blocks can be easily scaled out, a large block implies a dangling node, for which all the other nodes must wait before they can continue with their next task. Kolb et al. [2011a] demonstrate a parallel and balanced partitioning scheme to mitigate these issues. However, we expect the framework to automatically choose a join strategy for skewed data.

5.3.3 Sorted Neighborhood Method

While the naïve and blocking algorithms naturally translate into the parallelization primitives of Stratosphere, the Sorted Neighborhood Method (SNM) requires a more sophisticated approach. The basic idea is to partition the data in overlapping, balanced blocks of the sorted data, such that we can perform traditional SNM on the blocks while avoiding duplicate comparisons. Figure 5.4 summarizes the three steps of the basic algorithm described by Kolb et al. [2011b] for a table with person records that are sorted by names.

The algorithm first sorts the dataset in parallel and then range-partitions it into blocks of equal size (records with solid line). However, performing a local SNM on these blocks would lead to fewer comparisons than the traditional SNM, because records at the beginning of a block are not compared with the records at the end of the previous block. Therefore, for a window size w, the range partitioning replicates $w-1$ records for each block (records with dotted line), such that each record is compared with the same records as in the traditional SNM. To avoid duplicate comparisons of replicated records, replications are not compared with each other.

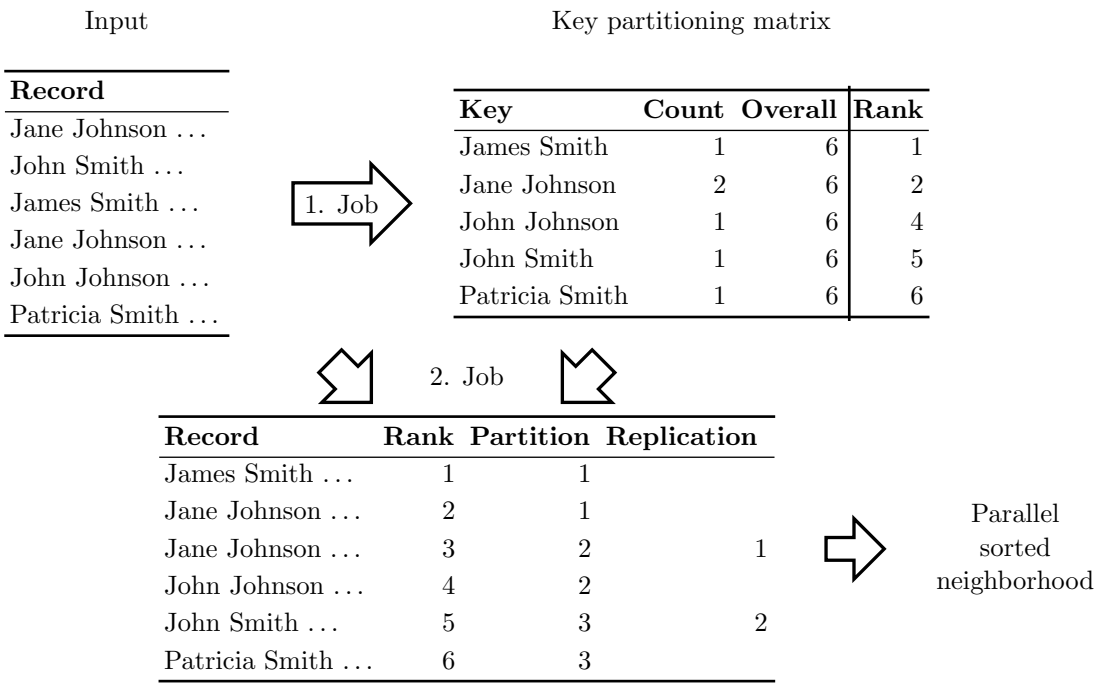


Figure 5.5: Range partitioning with key partitioning matrix in two Hadoop jobs.

The key challenge of this approach is to efficiently partition the data. The overall best implementation of this algorithm on Hadoop by Kolb et al. [2011b] requires two separate jobs. First, a preprocessing job creates a *key partitioning matrix* (KPM), which summarizes the distribution of sorting keys as shown in Figure 5.5. A second job then partitions the data with the KPM in the `map` and performs the SNM in the `reduce`. The main advantage of the KPM is an optimally balanced partitioning (neglecting the $w - 1$ replicated records) even for heavily skewed data, as can be seen in the example: The first "Jane Johnson" is assigned to the first computation node and the second "Jane Johnson" to the second. Nevertheless, all Hadoop approaches require the KPM to be held in main memory for the second job, which limits the scalability for datasets with many different sorting keys.

Together with the two students Jens Hildebrand and Jakob Zwiener, we developed an efficient and scalable implementation of the algorithm in Stratosphere during the seminar "Large-scale Duplicate Detection". The main contribution of our approach is to use ranks instead of value counts in the KPM. Consequently, our algorithm needs only one row of the KPM in main memory to perform the range partitioning of the dataset and thus overcomes one of the main limitations of the original approach. Figure 5.6 depicts the Stratosphere implementation. Note that some parts cannot be expressed with other Sopremo operators and therefore are implemented as elementary operators wrapping a second-order function, which we directly use in the plans.

First, we extract the sorting key of each record with a `transformation` and then count each sorting key with a `grouping`. With a successive non-parallel `sort` (a `reduce` on a dummy key with secondary sort), we sort the unique keys with their counts and

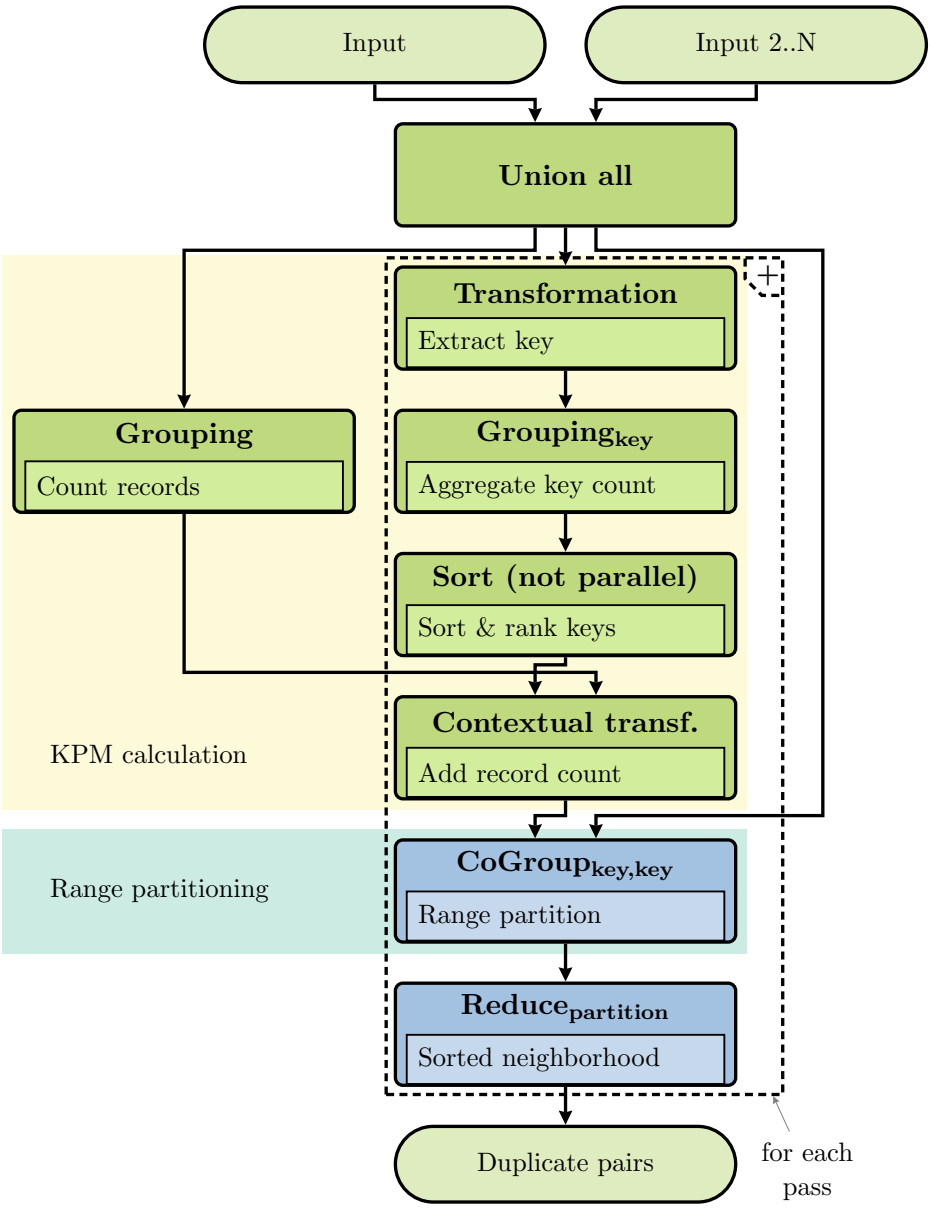


Figure 5.6: Parallel SNM in Stratosphere. The upper part calculates the key partitioning matrix used to create overlapping partitions. Each partition is processed on one node in a final reduce.

calculate their rank. We then add the global count with a `contextual transformation` (a `cross`) to each record and proceed with the range partitioning. The range partitioning uses a `cogroup` to process all records with the same key together with the KPM row of that key. For each record, we can then calculate the minimum partition number $part_{min}$ and the maximum partition number $part_{max}$ for replication using the number of partitions (i.e., the degree of parallelism), the number of all records, the window size w, and the position of each record inside the `cogroup` starting from 0.

$$part_{min} = (rank + position)\frac{\#partitions}{\#records}$$
$$part_{max} = min((rank + position + w - 1)\frac{\#partitions}{\#records}, \#partitions - 1)$$

The range partitioner always emits one record [$\mathbf{part_{min}}, \mathbf{part_{min}}, \mathbf{record}$] and additional replicated records for each p: $part_{min} < p \leq part_{max}$ [$\mathbf{p}, \mathbf{part_{min}}, \mathbf{record}$]. Usually, only few records are replicated once, but for window sizes larger than the partition sizes, records are replicated to more than one node. The actual SNM is then performed in a `reduce` over the partition number with the records being sorted by their sorting key (secondary sort). Replicated records can be easily identified with $p \neq part_{min}$ and are naturally sorted in the back of the partition. The window of a SNM is slid only over the non-replicated records and the SNM terminates as soon as a replicated record is the first element in the window. Different passes replicate all operators except for the `grouping` operator determining the number of records.

Duplicate detection with SNM scales well, because the partitions are equally sized and the number of comparisons depends only on the number of records and the window size and not on the actual data distribution. However, the bottleneck of this approach is the calculation of the KPM, which requires a non-parallel part for the rank calculation. Ideally, the framework efficiently supports range partitioning out of the box. Nonetheless, the scalability of this approach is much higher than the blocking approach and could even be improved in the future, because it is limited only by the expressiveness of second-order functions and not inherently by data dependencies.

Sorted neighborhood method for record linkage

To the best of our knowledge, the semantics of the SNM has not been properly defined for the `record linkage` operator. Therefore, we conceptually compare two plausible solutions in the following.

Figure 5.7 shows the *global sort* approach to a sorted neighborhood method for two inputs. We merge the inputs, sort them, and then slide a window over the resulting dataset. However, we can see that this approach does not adequately transfer the SNM idea. In this example, a normal sliding window would compare many records from the same dataset, which cannot result in duplicates according to our definition of the `record linkage` operator. On the one hand, if we ignore such comparisons completely, we cannot guarantee that records are compared at all. For example, *C. Patricia Jones* from the second input is effectively compared only with the next record and not with any previous record. On the other hand, if we enlarge the window dynamically until we perform $w-1$ actual comparisons in both directions, we lose the straight-forward semantics along with efficient and scalable implementations: In the worst case, the complete dataset must be traversed to find suitable records and every buffer to store the current window in-memory would need to potentially hold the entire dataset.

Therefore, we developed the *merge insert* approach to actively exploit the `record linkage` operator semantics. First, we sort both datasets with the respective sorting

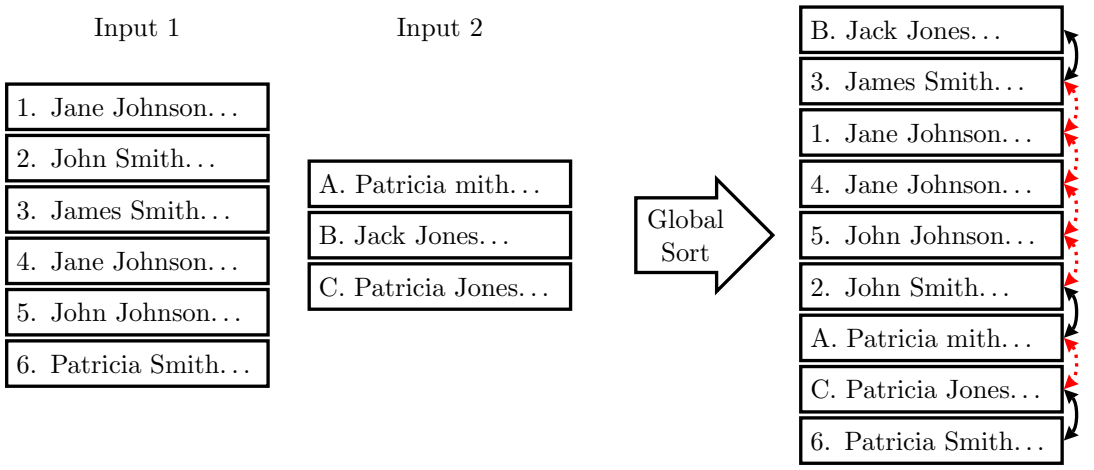


Figure 5.7: The *global sort* approach insufficiently transfers the SNM to the `record linkage` operator with window size 1. Red, dotted arrows indicate superfluous comparisons of records originating from the same dataset.

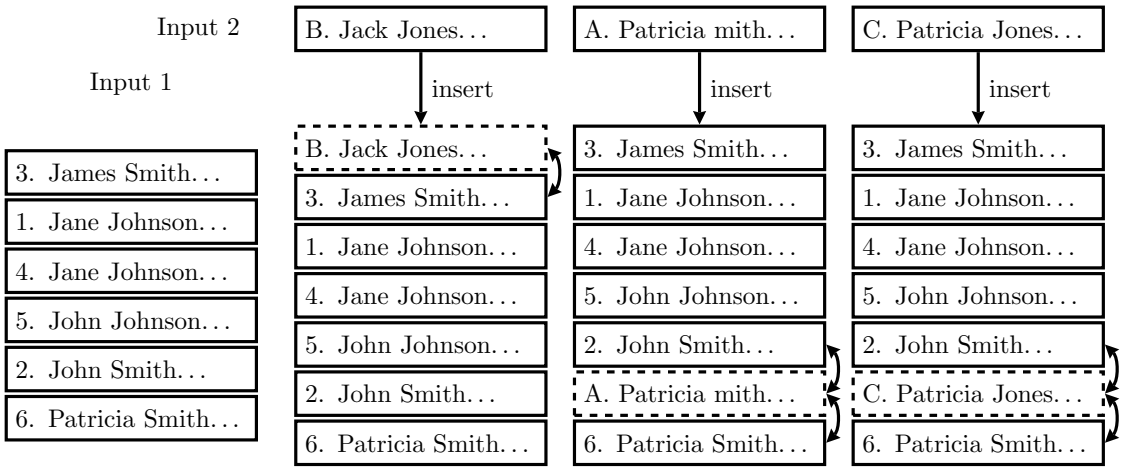


Figure 5.8: The *merge insert* approach finds the insert position for each record of the smaller input in the larger input and compares records within the window.

keys. Second, we find the insertion position of each record of the smaller dataset in the larger dataset. We then compare the record with $w-1$ records before and after the insertion position as shown in Figure 5.8. In comparison to the *global sort* approach, we can guarantee that each record of the smaller dataset is compared to the $2 \cdot w - 2$ nearest records in the large dataset (except near the top or bottom), which corresponds exactly to the number of comparisons of each record in a traditional SNM. Further, the buffer to hold the sliding window in-memory also needs to hold at most $2 \cdot w - 2$ records.

We can efficiently implement the latter approach in Stratosphere. Figure 5.9 depicts the implementation of the parallel sorted neighborhood for the `record linkage` operator. The algorithm differs from the SNM of the `duplicate detection` operator in two

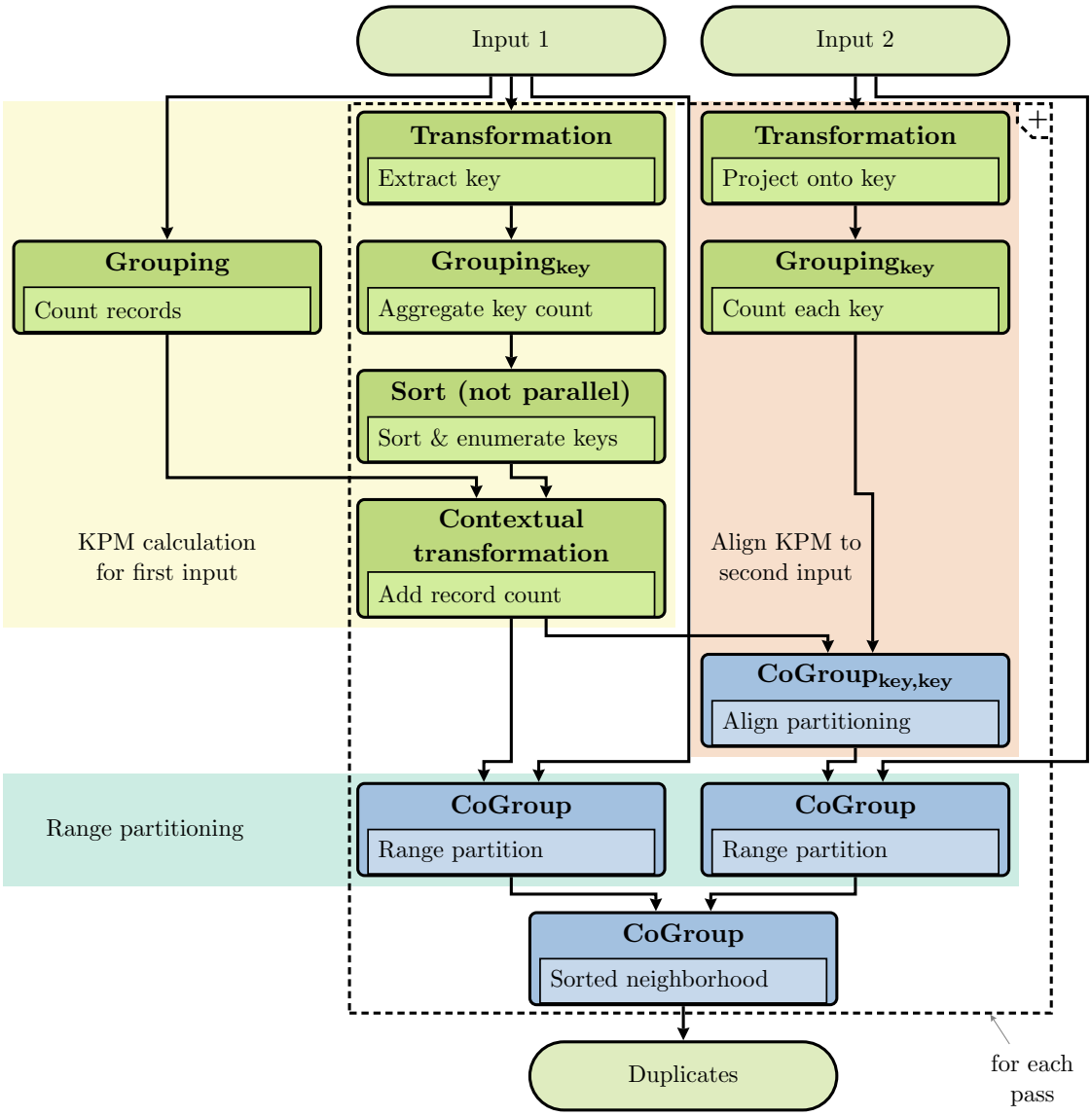


Figure 5.9: Implementation of the parallel sorted neighborhood method for two sources.

ways: Range partitioning of the second input must be aligned from the first input and the local SNM functions must be performed with a `cogroup`.

The calculation of the KPM for the first (smaller) input is identical to the KPM creation of the SNM for the `duplicate detection` operator. The right hand side of the plan uses the KPM to assign the correct partition to the records from the second (larger) input. Note that records need to be replicated from only one input to create overlapping blocks. Because the KPM was calculated using the smaller input, we also replicate records from the smaller input. Therefore, the range partitioning `cogroup` of the first input is identical to the `cogroup` of the `duplicate detection` operator, while the second `cogroup` is a simpler version without record replication.

The actual SNM is performed with a `cogroup`. The previous operators range-partitioned both datasets with the same KPM and for the SNM each partition is additionally sorted with the respective sorting key (secondary sort). The user-defined function of the `cogroup` slides the window over the larger (second) input until the current record of the smaller (first) input would be located in the middle of the window. It then compares the record from the smaller input with all records in the current window and repeats until all records from the smaller input have been processed.

Obviously the SNM for a `record linkage` operator is more complex than the implementation for `duplicate detection` operator. However, this additional complexity quickly amortizes. First, this plan performs the non-parallel KPM ranks calculation only for the smaller input. Second, the KPM alignment to the second input runs efficiently in parallel. Third, we perform significantly fewer sorting comparisons *sc* and fewer than half of the expensive candidate comparisons *cc* for two relations $\mathcal{R}_1, \mathcal{R}_2$ with sizes n_1, n_2:

$$\begin{aligned} sc_{rl} &= n_1 \cdot log(n_1) + n_2 \cdot log(n_2) + n_1 + n_2 \\ &= n_1 \cdot (log(n_1) + 1) + n_2 \cdot (log(n_2) + 1) \\ sc_{dd} &= (n_1 + n_2) \cdot log(n_1 + n_2) \\ &= n_1 \cdot log(n_1) \cdot log(n_2) + n_2 \cdot log(n_2) \cdot log(n_1) \\ &> sc_{rl} \quad \text{(since } log(n_1) > 1 \text{ for non-empty inputs)} \end{aligned}$$

$$\begin{aligned} cc_{rl} &\approx min(n_1, n_2) \cdot w \\ cc_{dd} &\approx (n_1 + n_2) \cdot w \\ &= cc_{rl} + max(n_1, n_2) \cdot w \end{aligned}$$

5.4 Optimizing plans with duplicate detection

In this section, we discuss opportunities to optimize plans with the `duplicate detection` and the `record linkage` operator and present cost and cardinality estimations.

5.4.1 Operator taxonomy

The `duplicate detection` and `record linkage` operators without clustering are basically generalized (self-)joins. In the following, we denote with $dd_{cs,sim,cl}(\mathcal{R}_1, \ldots, \mathcal{R}_n)$ the application of the `duplicate detection` operator with candidate selection *cs*, similarity expression *sim*, and clustering *cl* to the datasets $\mathcal{R}_i$ and with $rl_{cs,sim,cl}(\mathcal{R}_1, \mathcal{R}_2)$ the application of the `record linkage` operator to the datasets $\mathcal{R}_1$ and $\mathcal{R}_2$. We know that:

$$\begin{aligned} rl_{cs,sim,cl}(\mathcal{R}_1, \mathcal{R}_2) &= cl(\sigma_{cs \wedge sim}(\mathcal{R}_1 \times \mathcal{R}_2)) \\ dd_{cs,sim,cl}(\mathcal{R}_1, \ldots, \mathcal{R}_n) &= cl(\sigma_{cs \wedge sim}(\mathcal{R} \times \mathcal{R})), \text{ with } \mathcal{R} = \bigcup_{i \leq n} \mathcal{R}_i \end{aligned}$$

If the user chooses **no clustering** ("none" in Meteor), the operators are (self-) `theta-joins`. The `record linkage` operator returns all pairs of records of the Cartesian

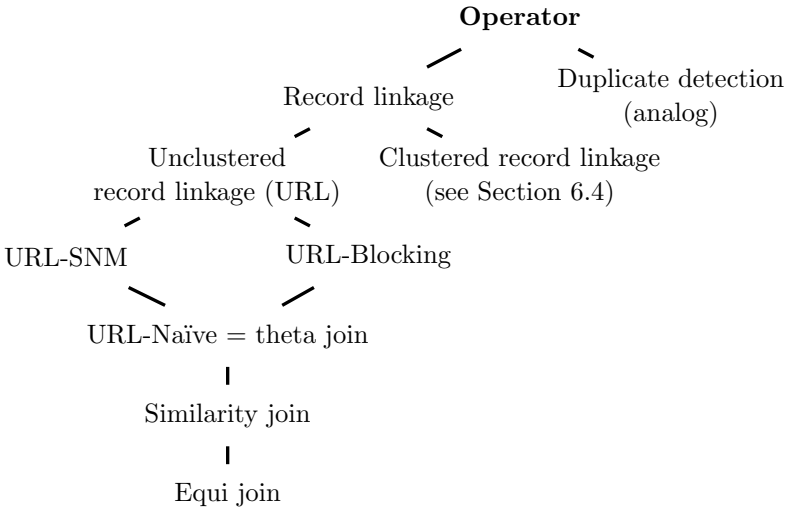


Figure 5.10: Operator taxonomy for `record linkage` operator with clustering.

product satisfying the candidate selection and similarity expression. The `duplicate detection` operator performs a self-join over the bag-unified inputs, where the candidate selection additionally removes reflexive and symmetric pairs.

For the **naïve** implementation, the candidate selection does not remove any pairs, whereas **blocking** can be modeled with an equivalence condition over the blocking keys. The **sorted neighborhood method** retains those records whose ranks in a sorted dataset do not differ more than the given window size. Therefore, blocking and SNM can simulate a naïve strategy: Blocking with arbitrary constant keys or a window size of the total number of records would result in a naïve comparison.

Further, the **naïve** implementation is equivalent to a theta-join as shown in Section 5.3. Hence, all join variations can be expressed as a `record linkage` operator. In particular, the *similarity join operator* with its four variations [Silva et al., 2010] based on theta-joins is a special case of the `record linkage` operator. A **clustering** may completely change the algebraic properties of the `duplicate detection` and the `record linkage` operators as discussed in Section 6.4.

Figure 5.10 summarizes the specialization/generalization relationships between the operators as specified in Presto [Rheinländer et al., 2013]. For each clustering and implementation combination, we have a specific sub-operator of the `record linkage` operator. Blocking without clustering is encoded as URL-Blocking. Similarity join and equi join have been added for completeness.

5.4.2 Reordering rules

For both operators, pushing a `projection` towards the data sources depends only on read-write conflicts. Attributes can be projected out as soon as they are not needed anymore after candidate selection, candidate comparison, or clustering. If none of these steps require the attributes, the `projection` can be safely pushed through the operators.

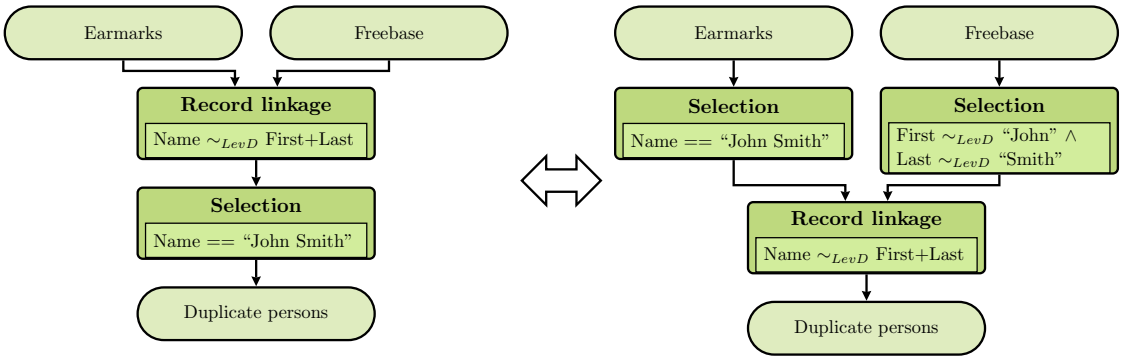


Figure 5.11: Pushing a selection to both inputs of a `record linkage` (Rule III).

In the remainder of this section, we consider the `record linkage` operator without clustering and defer discussion of reordering of clustered results to Section 6.4. We indicate with *id* that no clustering is involved and focus on the `record linkage` operator (rules are trivially applicable to `duplicate detection`). The following reordering rules generalize some of the rules of the similarity join operator [Silva et al., 2010].

Selection

A preceding selection may be pushed through the `record linkage` operator with **blocking** and **naïve** implementations, for **uncorrelated conditions** θ_1 and θ_2, which exclusively use attributes from $\mathcal{R}_1$ and $\mathcal{R}_2$, respectively.

$$\sigma_{\theta_1}(rl_{cs,sim,id}(\mathcal{R}_1, \mathcal{R}_2)) \equiv rl_{cs,sim,id}(\sigma_{\theta_1}(\mathcal{R}_1), \mathcal{R}_2) \quad \text{(Rule I)}$$

$$\sigma_{\theta_1 \wedge \theta_2}(rl_{cs,sim,id}(\mathcal{R}_1, \mathcal{R}_2)) \equiv rl_{cs,sim,id}(\sigma_{\theta_1}(\mathcal{R}_1), \sigma_{\theta_2}(\mathcal{R}_2)) \quad \text{(Rule II)}$$

$$\text{with } cs \in \{naive, blocking\}$$

Cheaper plans can be created by pushing selection conditions in a *relaxed* form to the other input. We denote with θ^{sim} a relaxed selection criterion, such that:

$$\sigma_{\theta_1}(rl_{cs,sim,id}(\mathcal{R}_1, \mathcal{R}_2)) \equiv rl_{cs,sim,id}(\sigma_{\theta_1}(\mathcal{R}_1), \sigma_{\theta_1^{sim}}(\mathcal{R}_2)) \quad \text{(Rule III)}$$

Relaxed selection criteria are derived by propagating the similarity function of calculated expressions toward the path expressions. Obviously, the stricter the similarity expressions are, the more selective the relaxed criteria are. The cost estimator has to assess if the selection with the relaxed condition is selective enough to amortize the additional runtime costs of the selection. Further, for **correlated conditions**, only the uncorrelated parts can be pushed similar to `joins`.

Example 5.1. Consider a `record linkage` over `Earmarks` and `Freebase` politicians, where we allow a maximum Levenshtein distance of sponsor names and politician names of 1. We select from all joined records the sponsors named "John Smith" as shown in Figure 5.11. With the third rule, this selection can be pushed towards the `Earmarks` datasets and in relaxed form towards `Freebase`, where the Levenshtein distance between the first/last name and "John"/"Smith" can be at most 1.

Selection and sorted neighborhood method

For the **SNM** implementation, we have the interesting case that we could potentially identify more duplicates, when pushing the selection towards the data sources with the above rules. Consider a dataset with 100 records. A SNM with window size 10 compares approximately 20% of all non-reflexive and non-symmetric pairs. If we push a selection with 10% selectivity through the `record linkage` operator, the SNM with the same window size compares all pairs. Often, a user would benefit from receiving potentially more duplicates in a shorter time. However, strictly speaking the results are not equivalent.

We see three solutions to this observation with one rule for each. First, we allow *fuzzy reorderings* and can immediately apply the above rules. Second, we additionally reduce the window size, which may miss some duplicates, but compare the desired number of records on average. Third, we incorporate the selection condition in the similarity expression and achieve the same result and save some expensive candidate comparisons, but miss the opportunity to push the selection even more towards the data source. The first two rules are irreversible and may add results $\Rrightarrow^{+}$ or even remove results $\Rrightarrow^{-}$.

$$\begin{aligned}
\sigma_{\theta_1}(rl_{snm,sim,id}(\mathcal{R}_1,\mathcal{R}_2)) &\Rrightarrow^{+} rl_{snm,sim,id}(\sigma_{\theta_1}(\mathcal{R}_1),\mathcal{R}_2)\\
\sigma_{\theta_1}(rl_{snm,sim,id}(\mathcal{R}_1,\mathcal{R}_2)) &\Rrightarrow^{-} rl_{snm',sim,id}(\sigma_{\theta_1}(\mathcal{R}_1),\mathcal{R}_2)\\
\sigma_{\theta_1}(rl_{snm,sim,id}(\mathcal{R}_1,\mathcal{R}_2)) &\equiv \quad rl_{snm,\theta_1 \wedge sim,id}(\mathcal{R}_1,\mathcal{R}_2)
\end{aligned}$$

Commutativity

Both operators are **commutative** for the blocking and naïve variations. The SNM `record linkage` operator performs asymmetric comparisons, but for a *fuzzy optimization*, which also considers *approximate equivalent* plans with slight differences in the results, assuming commutativity can significantly improve the performance for datasets with a large size difference. The SNM `duplicate detection` operator is symmetric, because the initial `union all` is symmetric.

$$rl_{blocking,sim,id}(\mathcal{R}_1,\mathcal{R}_2) \equiv rl_{blocking,sim,id}(\mathcal{R}_2,\mathcal{R}_1) \quad \text{(Rule IV)}$$

$$rl_{naive,sim,id}(\mathcal{R}_1,\mathcal{R}_2) \equiv rl_{naive,sim,id}(\mathcal{R}_2,\mathcal{R}_1) \quad \text{(Rule V)}$$

$$dd_{cs,sim,id}(\mathcal{R}_1,\mathcal{R}_2) \equiv dd_{cs,sim,id}(\mathcal{R}_2,\mathcal{R}_1) \quad \text{(Rule VI)}$$

Associativity

For the **blocking** and **naïve** operators, we can perform subsequent `record linkage` and `duplicate detection` operators in arbitrary order if there is no read-write conflict between the candidate selection and a previous `record linkage` operator. For SNM, the records within a window may change drastically between different orders, so no reasonable rule is possible.

$$rl_{cs_1,sim_1,id}(\mathcal{R}_1, rl_{cs_2,sim_2,id}(\mathcal{R}_2,\mathcal{R}_3)) \equiv rl_{cs_2,sim_2,id}(rl_{cs_1,sim_1,id}(\mathcal{R}_1,\mathcal{R}_2),\mathcal{R}_3), \quad \text{(Rule VII)}$$
$$\text{with } cs \in \{naive, blocking\}$$

Note that through the operator hierarchy these rules additionally allow reordering of `record linkage` and `duplicate detection` operators with (similarity) joins.

In summary, without clustering, a rich set of reordering rules is applicable to the `record linkage` and `duplicate detection` operators. With clustering only few rules remain, but additional opportunities arise in conjunction with the `fusion` operator. Moreover, a large contribution of this thesis is the cost and cardinality estimation.

5.4.3 Cost estimation

The cost estimation in this section has not been implemented yet but should provide a robust method for the future Stratosphere statistics component. Cost estimation requires information about the costs and the cardinalities of the three sub-tasks candidate selection, candidate comparison, and clustering. We examine the clustering phase in Section 6.4 and focus on estimating candidate selection first. We neglect the runtime cost of the global enumeration in the naïve and blocking `duplicate detection` operator, because the resulting `map` task adds no network and little CPU costs. Further, we simply sum up the individual costs and ignore savings from interleaved execution of the steps.

$$cost_{total} = cost_{cs} + cost_{cc} + cost_{cl}$$

Candidate selection

Naïve: The runtime cost for this candidate selection depends completely on the network cost to replicate the (smaller) dataset to each node. The cardinality of the resulting candidates is $|\mathcal{R}_1| \cdot |\mathcal{R}_2|$ for the `record linkage` operator and $\binom{\mathcal{R}}{2}$ for the `duplicate detection` operator.

Blocking: The runtime cost and cardinality is equivalent to the underlying, distributed `equi-joins` on the blocking keys of the passes. We again assume that these key figures are estimated from the core system well enough. For the `duplicate detection` operator, we halve the cardinality estimation to remove symmetric comparisons but ignore the relatively few, reflexive comparisons in our estimate.

SNM: The cardinality of the SNM `record linkage` can be easily estimated with $2 * w * \mathcal{R}_i$ for window size w and smaller dataset $\mathcal{R}_i$, while SNM `duplicate detection` generates $w * \mathcal{R}$ candidate pairs. The runtime cost of the candidate selection is more complex and is dominated by creating the key partitioning matrix. The subsequent range partitioning reuses the same hash partitioning as the KPM creation, so that the overhead can be neglected. Therefore, the two expensive operators are `grouping` all records by the sorting key to create count statistics and the calculation of the rank in a non-distributed `sort`. Both steps depend on the total number of records and the value distribution. The first step can rely on the estimations of the core system for a `grouping` operator. The second step solely depends on the number of unique values and the network costs to

transfer a record for each unique key over the network. The actual computation can be neglected for both steps, because they involve simple aggregations.

$$network\_cost_{snm} = network\_cost_{key\ grouping} + unique\_keys \cdot avg\_network\_cost$$

Candidate comparison

To estimate the CPU cost of the candidate comparison, we measure the average cost of a comparison during the cardinality estimation. Therefore, the runtime cost of the candidate comparison can be directly calculated by extrapolation.

$$cpu\_cost_{cc} = card_{cs} \cdot avg\_cost_{sim}$$

5.4.4 Cardinality estimation

When performing random sampling on the input datasets, the selectivity of the candidate comparison depends on the sample rate. To extrapolate findings on a sample $\mathcal{S} \subseteq \mathcal{R}$ to the original dataset $\mathcal{R} = \bigcup_i \mathcal{R}_i$, it is important to understand how sampling changes the number of duplicates and the cluster size histogram. In this section, we build upon the probabilistic model of the sampling process introduced in [Heise et al., 2014].

Random sampling for duplicate detection

We denote with $n = |\mathcal{S}|$ the number of samples, $N = |\mathcal{R}|$ the size of the original dataset, and with r or r_i single records in $\mathcal{R}$. We start with the naïve `duplicate detection` operator on the input dataset $\mathcal{R}$ and primarily consider random sampling with a sample rate $p(r \in \mathcal{S}) = \frac{n}{N}$. First, we define duplicates as a subset of the *pair set*, such that each pair satisfies the equivalence relation $\sim$.

Definition 5.10 (Pair set)**.** A pair set of $\mathcal{R}$ contains all pairs of different records of $\mathcal{R}$ once: $\mathcal{R}^{<2} = \{(r_i, r_j) \mid (r_i, r_j) \in \mathcal{R} \times \mathcal{R}; i, j \in \{1, \ldots, N\} \wedge i < j\}$.

Definition 5.11 (Duplicate pairs)**.** The set of the duplicate pairs $\mathcal{D}_\mathcal{R}$ contains all pairs of duplicates in $\mathcal{R}$: $\mathcal{D}_\mathcal{R} = \{(r_i, r_j) \mid (r_i, r_j) \in \mathcal{R}^{<2} \wedge r_i \sim r_j\} \subseteq \mathcal{R}^{<2}$.

We now inspect the ratio of duplicate pairs in the sample $\mathcal{S}$ with respect to the original dataset $\mathcal{R}$. First, we estimate how many duplicate pairs $\mathcal{D}_\mathcal{S}$ can be found in the sample. The expected number of duplicate pairs $\mathbf{E}[|\mathcal{D}_\mathcal{S}|]$ depends on the probability $p(d \in \mathcal{D}_\mathcal{S})$ that we sample a duplicate pair $d \in \mathcal{D}_\mathcal{R}$ in the sample $\mathcal{S}$.

$$\begin{aligned}\mathbf{E}[|\mathcal{D}_\mathcal{S}|] &= |\mathcal{R}^{<2}|\ p(d \in \mathcal{D}_\mathcal{S}) \\ &= |\mathcal{R}^{<2}|\ p(d \in \mathcal{D}_\mathcal{R})\ p(d \in \mathcal{D}_\mathcal{S} \mid d \in \mathcal{D}_\mathcal{R})\end{aligned}$$

Because random sampling is independent of the probability that a pair is a duplicate, the second probability is equal to the probability that we draw a specific pair during

sampling $p(d \in \mathcal{S}^{<2})$. We can further factor in that we have $\binom{|\mathcal{X}|}{2}$ possibilities to draw a pair from $\mathcal{X}^{<2}$.

$$\begin{aligned}\mathbf{E}[|\mathcal{D}_\mathcal{S}|] &= |\mathcal{R}^{<2}|\ p(d \in \mathcal{D}_\mathcal{R})\ p(d \in \mathcal{S}^{<2}) \\ &= |\mathcal{D}_\mathcal{R}|\ p(d \in \mathcal{S}^{<2}) = |\mathcal{D}_\mathcal{R}|\ \frac{\binom{n}{2}}{\binom{N}{2}} \\ &= |\mathcal{D}_\mathcal{R}|\ \frac{n\ (n-1)}{N\ (N-1)} \end{aligned} \tag{5.1}$$

Hence, the number of duplicate pairs increases quadratically with the sample size, as is shown in the following example.

Example 5.2. Consider a dataset with 1,000 tuples and 500 duplicate pairs. If we sample 100 records, we would expect only $|\mathcal{D}_\mathcal{S}| = 500 \cdot \frac{100 \cdot 99}{1{,}000 \cdot 999} \approx 4.95$ duplicate pairs on average in the sample. For a sample of 200 records, the expected value would be ≈ 19.92.

Extrapolating from random sampling

From Equation 5.1, we can easily see how to extrapolate the number of duplicates found in the sample $|\mathcal{D}_\mathcal{S}|$ to the original dataset.

$$|\mathcal{D}_\mathcal{R}| = |\mathcal{D}_\mathcal{S}|\ \frac{N\ (N-1)}{n\ (n-1)} \tag{5.2}$$

Clearly, *extrapolating* from such a sample to estimate the original (i.e., actual) number of duplicates is highly sensitive to the sample variance. In the above example, we should receive five duplicates in most samples to obtain a good estimate of $|\mathcal{D}_\mathcal{R}| = 5\ \frac{1{,}000 \cdot 999}{100 \cdot 99} \approx 504.55$. However, even a slight variation of only one sampled duplicate changes the estimation result by approximately 100 duplicate pairs.

Sampling and extrapolation for record linkage

We can easily transfer this sampling and extrapolation model to the naïve `record linkage` operator on the datasets $\mathcal{R}_1$ and $\mathcal{R}_2$ with duplicates $\mathcal{D}_\mathcal{R}$. Because reflexive and symmetric comparisons cannot occur, the expected number of duplicates in the sample $\mathbf{E}[|\mathcal{D}_\mathcal{S}|]$ solely depends on the sample rates of the respective datasets.

$$\begin{aligned}\mathbf{E}[|\mathcal{D}_\mathcal{S}|] &= |\mathcal{R}_1 \times \mathcal{R}_2|\ p(d \in \mathcal{D}_\mathcal{S}) \\ &= |\mathcal{R}_1 \times \mathcal{R}_2|\ p(d \in \mathcal{D}_\mathcal{R})\ p(d \in \mathcal{S}) \\ &= |\mathcal{D}_\mathcal{R}|\ p(d \in \mathcal{D}_\mathcal{R})\ p(d \in \mathcal{S}) \\ &= |\mathcal{D}_\mathcal{R}|\ \frac{n_1\ n_2}{N_1\ N_2} \end{aligned} \tag{5.3}$$

Further, extrapolation requires simple cross-multiplication with the inverse of the sample rates of the respective datasets.

$$|\mathcal{D}_\mathcal{R}| = |\mathcal{D}_\mathcal{S}|\ \frac{N_1\ N_2}{n_1\ n_2} \tag{5.4}$$

Influence of candidate selection

If the actual duplicate detection run encompasses candidate selection techniques, we can see two effects. First, some duplicate pairs are not found. Second, the runtime of the duplicate detection run decreases. Both effects can be used to improve the estimation. The obvious solution is to calculate the sample histogram using the same candidate selection techniques, so that the estimations become more accurate for the configuration and faster at the same time. However, small adjustments to some candidate selection techniques are necessary:

Blocking. Duplicates can be found only within these groups of the same blocking key. Therefore, the estimation for blocking uses the same blocking keys and analogously looks for the duplicates within the blocks. Index structures speed up the insertion of new sample records over iterations.

SNM. The estimation for SNM uses sort-merge to extend the sample using the same sorting keys. The window size w should be linearly adjusted to the sample rate with $\tilde{w} = max(2, \lceil w\frac{n}{N} \rceil)$, so that a comparable amount of neighbors are selected.

Multiple passes should be analogously applied to the sample as well. The estimation algorithm maintains multiple index structures or sorted lists. Further, duplicate comparisons can be avoided with a cache data structure.

Summary In this chapter, we extensively described the two operators `duplicate detection` and `record linkage` as the core operators for data integration. We introduced a general model for both operators and discussed how Meteor users easily configure the candidate selection, candidate comparison, and clustering steps of both operators.

Further, we provided the implementation in Stratosphere for the six operator/candidate selection combinations. Naïve and blocking implementations naturally translate into few second-order functions. The sorted neighborhood method requires a more sophisticated solution. Additionally, we defined the semantics of a SNM for `record linkage` over two datasets and introduced their implementation. Lastly, we found a comprehensive set of reordering rules for the `duplicate detection` and `record linkage` without clustering and provided detailed estimation techniques.

Results obtained by the `record linkage` operator may be immediately used, for instance by publishing additional links between Linked Open Datasets. However, for our running example of materialized integration of Open Government Data, we also want to consolidate the duplicate pairs into disjoint clusters (see Chapter 6), which we can then fuse to concise representations (see Chapter 7).

6

Clustering Duplicate Pairs

The `clustering` operator generates a transitively closed result of the duplicate pairs resulting from the `duplicate detection` and `record linkage` operators. We interpret these pairs as a graph where vertices represent records and edges describe duplicate declarations. We provide two commonly used strategies as sub-operators to create transitively closed clusterings from such graphs.

The `transitive closure` operator adds edges, such that each connected component is maximally connected (i.e., it forms a clique). It is naturally used together with the `duplicate detection` operator and increases the recall. In contrast, the `stable matching` operator retains only pairs of records from connected components that have a maximum similarity. Because it avoids matching records from the same sources, it is primarily designed for a `record linkage` operator between two clean datasets and increases the precision of the result.

This chapter first introduces both strategies in theory and discusses their advantages and disadvantages. It then describes the implementations in Stratosphere after introducing the concept of iterations. Further, it presents reordering rules and advanced cardinality estimation for the `duplicate detection` in conjunction with our clustering strategies. The chapter concludes with an evaluation of the cardinality estimation technique, which demonstrates that relatively small samples suffice to accurately estimate the number and sizes of the resulting cluster.

6.1 Clustering strategies and their implications

In the last chapter, we already briefly defined the `clustering` operator as a post-processing step of the `duplicate detection` and `record linkage` operator. For practical purposes, we extend the definition to allow incomplete partitionings that do not contain all records of the original dataset.

Definition 6.1 (Clustering operator)**.** Let $\mathcal{D}$ be a set of declared duplicate pairs over the relations $\mathcal{R}_i$. The *clustering operator* ingests $\mathcal{D}$ and returns an (incomplete) partitioning $\mathcal{C}$ over $\bigcup_i \mathcal{R}_i$. All records $r \in \bigcup_i \mathcal{R}_i$ not contained in any cluster in $\mathcal{C}$ are implicitly in *singleton* clusters consisting of only one record.

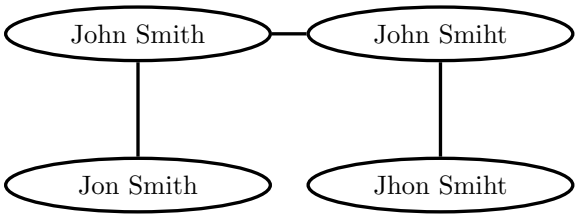


Figure 6.1: Transitively unclosed duplicate pairs. Vertices represent records and edges duplicate pairs.

In some related work, the result of clustering is also represented as duplicate pairs, so that the quality measures of precision and recall remain applicable. However, we explicitly want to obtain clusters for three reasons. First, pairs of a transitively closed result carry redundant information and thus unnecessarily increase the data volume. Second, we think clusters are easier to interpret than pairs, because users need to assess less information. Especially, the important identification of false positives is less daunting as clusters can be inspected independently. Third, the subsequent `fusion` operator can merge duplicate records in parallel, when the input is organized in clusters. In particular, we can assume that there is no data dependency between clusters.

Figure 6.1 shows four records with duplicate declarations from a `duplicate detection` operator invocation. The records form a connected component that is transitively unclosed. There are two different scenarios that lead to such a result. On the one hand, all records represent the same entity according to an equivalence relation $\sim$, but some edges are missing (i.e., false negatives). On the other hand, they might not represent the same entity, which means that we have incorrect edges (i.e., false positives). Of course, there could also be a mix of the two previous cases. Two records may correspond to one entity and the other two to another. Then, we have false positives and false negatives.

We can identify three technical reasons for incorrect classifications. First, false positives result from a lenient candidate classification. Second, the candidate selection prunes true duplicates, which results in false negatives. Third, the candidate classification rejects true positives, because it is too strict. All three cases usually result from dirty and incomplete data as well as sub-optimal configurations. However, for dirty data, it is often even impossible for domain experts to correctly classify candidate pairs. Therefore, data integration queries use clustering operators to correct classification errors.

6.1.1 Transitive closure

The transitive closure may be the most widely used clustering algorithm, because of its simplicity. Basically, the `transitive closure` operator assumes that all inconsistencies arise from pruned true duplicates. All records that are transitively connected correspond to the same entity according to the equivalence relation $\sim$. Therefore, the `transitive closure` operator adds all edges between records within a connected component; that is, it ensures that each connected component is a clique.

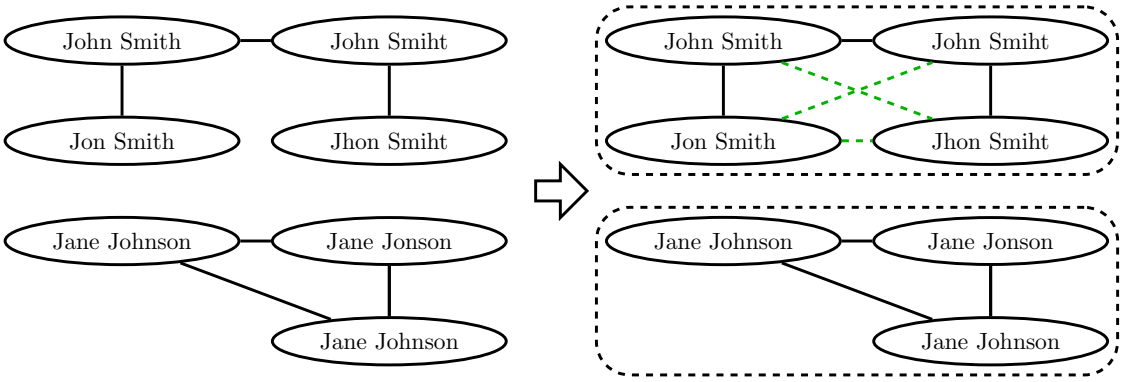


Figure 6.2: The transitive closure transforms connected components to cliques.

Figure 6.2 visualizes the result of applying the strategy to a graph with two connected components. The first component is the transitively unclosed component from before (see Figure 6.1), which becomes a 4-clique. The other component is already a 3-clique and is returned unmodified. Formally, we define transitive closure as follows.

Definition 6.2 (Transitive closure)**.** Let $\mathcal{D}$ be a set of declared duplicate pairs. The `transitive closure` operator returns the partitioning $\mathcal{C}$, such that a) $\mathcal{C}$ *covers* $\mathcal{D}$ and b) $\mathcal{C}$ is *minimal*. The covering property ensures that all duplicate pairs are contained in clusters; that is, $covers(\mathcal{C}, \mathcal{D}) \colon \forall d \in \mathcal{D} \colon \exists C \in \mathcal{C} \colon d \subseteq C$. Minimality ensures that not more duplicate pairs have been added as necessary: $minimal(\mathcal{C}) \colon \nexists \mathcal{C}' \colon covers(\mathcal{C}', \mathcal{D}) \wedge |\mathcal{C}'| < |\mathcal{C}|$.

While the transitive closure strategy does not need any parameter and can be relatively easy used, it has one major drawback, which becomes especially problematic in large-scale use cases. Because duplicate declarations are propagated within the connected components, one false positive edge immediately connects two clusters and causes numerous additional false positives. Therefore, lenient similarity configurations (thresholds or unsuited similarity measures) result in unmanageable large clusters. For example, while finding duplicate CD releases in the FreeDB dataset [Vogel et al., 2014], we experienced that the transitive closures can quickly create huge clusters of all CD releases of one band, if they released many albums and singles that share a certain number of tracks.

Obviously, the unconditional enlargement of clusters to connected components causes this unwanted behavior. The transitive closure discards all additional information, such as the actual similarity values between the records.

6.1.2 Stable matching

For our cleansing package, we provide a second clustering algorithm, which demonstrates the potential of using additional information during the clustering and is tailored for a `record linkage` with two clean datasets. The basic assumptions are that *a*) there are only clusters with a maximum size of 2, but *b*) the declared pairs may not be complete, because of candidate selections. For data integration of two duplicate-free datasets,

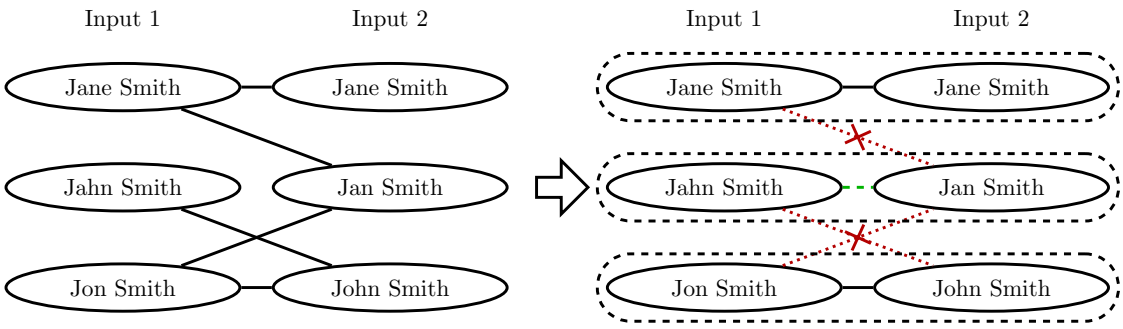


Figure 6.3: Stable matching finds the best matching pairs.

the first assumption helps to avoid matches within one datasets caused by a lenient configuration. Böhm et al. [2012a] use the generalized constraint *unique mapping per data source* in their consistency definition when finding new links between Linked Open Datasets.

Figure 6.3 shows a connected component with declared duplicates from two datasets. A transitive closure would simply declare all records to represent the same entity. However, some entities are more similar than other entities. "Jane Smith" is a perfect match and should be preferred over the match ("Jane Smith", "Jan Smith"). For "John Smith", two records from Input 2 have the same edit distance. Manual inspectors would intuitively choose the probably misspelled "John" over "Jan", because of the phonetic similarity. Further, although ("Jahn Smith", "Jan Smith") is not declared due to candidate selection, a match seems highly likely.

The `stable matching` operator adds and removes edges to find the best matches within a connected component. It basically counteracts the problem of inconsistent pairs in two ways. Like the `transitive closure` operator, it assumes that the candidate selection might lose some true duplicates and therefore searches for pairs in connected components instead of completely relying on the initial pairs. Further, it additionally assumes that the candidate classification is not perfect and might declare false positives, which need to be removed. To that end, the strategy builds upon the assumption that the result contains only pairs of records. Therefore, it ranks the pairs of records with a similarity function and retains the best matches. We can formally define stable matching as follows:

Definition 6.3 (Stable matching)**.** Let C be a transitively connected component over the relations $\mathcal{R}_i$. The *stable matching* clustering uses a similarity expression sim to return a set of stable matches $\mathcal{B}$ for C, such that there is no pair, where one record has a higher mutual similarity with another record: $\forall\langle r_1, r_2\rangle \in \mathcal{B} : \nexists\langle r_3, r_4\rangle \in \mathcal{B} : sim(r_1, r_4) > sim(r_1, r_2) \wedge sim(r_1, r_4) > sim(r_3, r_4)$.

Algorithm 6.1 outlines our stable matching algorithm. The algorithm first partitions the dataset into connected components with the `transitive closure` operator. For each component, it recursively looks for stable pairs; that is, two vertices that have the highest similarity with each other according to the similarity function sim. The algorithm then

removes the matched vertices and repeats the steps with the remaining vertices until no more vertices are left. Because we allow the similarity function to indicate a certain non-match, a vertex may not find any partner at some point. Therefore, we also remove these records from the set of vertices.

```
   Input  : Declared duplicates 𝒟, similarity function sim
   Output: Best pairs 𝒫
 1 𝒞 ← find connected components with transitive closure;
 2 foreach component C ∈ 𝒞 do
 3 |  V ← vertices in C;
 4 |  while V ≠ ∅ do
 5 |  |  ℬ ← new multi set;
   |  |  /* Find best match for each remaining vertex                */
 6 |  |  foreach v ∈ V do
 7 |  |  |  ℬ ← ℬ ∪ max_{v'∈V} sim(v, v');
 8 |  |  end
   |  |  /* If both agree, add them to result and remove them        */
 9 |  |  foreach B ∈ ℬ appearing twice do
10 |  |  |  𝒫 ← 𝒫 ∪ B;
11 |  |  |  𝒱 ← 𝒱 \ B;
12 |  |  end
   |  |  /* Remove all vertices that cannot find a partner           */
13 |  |  foreach v ∈ V do
14 |  |  |  if max_{v'∈V} sim(v, v') = ⊥ then
15 |  |  |  |  𝒱 ← 𝒱 \ {v};
16 |  |  |  end
17 |  |  end
18 |  end
19 end
```

Algorithm 6.1: Stable matching algorithm.

The algorithm obviously terminates if in every iteration at least one pair is chosen or a vertex is removed due to the lack of partners. For trivial clusters of sizes 1 and 2, we can easily see that the condition is fulfilled. For a cluster of size 3 with nodes $[a, b, c]$, there could be two problems. First, a has a higher similarity to b, b to c, and c to a. However, we can safely exclude this case, because the similarity function sim is symmetric by definition. Second, all three vertices have the same similarity to each other. In this case, we use a tie-breaker that primarily uses the natural order of the records corresponding to the vertices and secondarily the vertex position in the cluster.

In comparison to the `transitive closure`, the `stable matching` operator has the advantage that the `duplicate detection` operator may misclassify some pairs and the clustering still produces a relatively sane result for data integration use cases. Additionally, unlike more advanced clustering algorithms, such as center [Haveliwala et al., 2000] or merge-center [Hassanzadeh and Miller, 2009], the parameters are quite intuitive and remain manageable. However, the operator is limited to finding pairs and is less

suited for integrating more than two data sources in a *one-shot* integration strategy (see Appendix A.5).

6.2 Clustering in Meteor

The `duplicate detection` and the `record linkage` operator usually implicitly invoke the `clustering`. However, users may also explicitly invoke it to have more control over the configuration. Listing 6.1 shows the three variants to invoke a `clustering` operator.

```
1 $duplicate_clusters = cluster transitively $duplicate_pairs;

2 $matches = match stably $p in $duplicate_pairs
      with levenshtein($p.firstName + ' ' + $p.lastName) >= .9
      and jaroWinkler($p.firstName) >= .8;

3 $matches = cluster $duplicate_pairs
      using 'stable matching'
      with ...;
```

Listing 6.1: Clustering duplicate pairs in Meteor.

Both strategies are implemented as separate composite operators and may be invoked directly. The first line shows the parameterless transitive closure strategy. In the second line, we invoke the stable matching strategy with two similarity measures and one threshold as discussed in the following. The third line, invokes the `clustering` operator, which offers a property to choose the strategy and all properties from the chosen strategy.

Stable matching

The *similarity expression* of the stable matching strategy consists of at least one similarity measure with optional thresholds. The importance of the similarity measures is defined in descending order. Subsequent similarity measures act only as tie-breakers, when similarities of the previous similarities are equal. In the previous Meteor snippet, the Levenshtein similarity of the concatenated name is the primary similarity measure, while the Jaro-Winkler puts a stronger emphasize on the first name in case of ties. However, thresholds are always applied to ensure valid results. In the example, a pair would be rejected if the first name has a Jaro-Winkler similarity of less than .8 even if the Levenshtein similarity is the highest of the candidate pairs.

We can implement this semantics by calculating *similarity arrays* for each pair. All similarity measures in the conjunction are successively calculated and stored into a Sopremo array. We can then easily remove all pairs that do not satisfy the threshold with an element-wise comparison to a *threshold array* extracted from the similarity expression. For the Meteor example, the threshold array would be "`[.9, .8]`". Further, we can find best matches by comparing the similarity arrays from the first to the last elements, which is Sopremo's standard ordering for arrays.

In 1 \ In 2	Jane Smith	Jan Smith	John Smith
Jane Smith	**[1.00, 1.00]**	[0.90, 0.94]	~~[0.70, 0.70]~~
Jahn Smith	~~[0.80, 0.87]~~	**[0.90, 0.93]**	[0.90, 0.85]
Jon Smith	~~[0.80, 0.75]~~	~~[0.89, 0.80]~~	**[0.90, 0.93]**

In 2 \ In 1	Jane Smith	Jahn Smith	Jon Smith
Jane Smith	**[1.00, 1.00]**	~~[0.80, 0.87]~~	~~[0.80, 0.75]~~
Jan Smith	**[0.90, 0.94]**	[0.90, 0.93]	~~[0.89, 0.80]~~
John Smith	~~[0.70, 0.70]~~	[0.90, 0.85]	**[0.90, 0.93]**

Table 6.1: Similarity arrays for a stable matching. Bold arrays indicate best matches, while strikethrough arrays represent pairs below the given threshold. The top (bottom) table shows the preferences for records of the first (second) input.

Table 6.1 shows the similarity arrays for the stable matching example (see Figure 6.3). Eight combinations are removed with the threshold array (strikethrough). Then, the best pairs are picked. For the two "Jane Smiths" and the two "Jo(h)n Smiths", the respective record in the other source is the best match (bold). Interestingly, "Jan Smith" of Input 2 prefers "Jane Smiths" over "Jahn Smiths", while "Jahn Smiths" favors "Jan Smith".

From the last case, we can see that the stable matching algorithm needs several iterations to successively determine the stable match(es). It first chooses the "Jane Smiths" and "Jo(h)n Smiths" due to mutual preference and removes the four records from the set of vertices. In the second iteration, the "Ja(h)n Smiths" remain, share a mutual preference, and are thus selected as the best match.

Inferring stable matching configurations

By design, the similarity expression of the stable matching strongly resembles the similarity expression of a `duplicate detection` operator (see Section 5.2). Therefore, the `clustering` operator can automatically infer a suitable stable matching configuration from the similarity expression of a `duplicate detection` operator (and `record linkage` analogously).

Although the similarity expression of a `duplicate detection` operator may contain arbitrary Boolean expressions, it usually consists of a conjunction of several rules. Negative rules either enforce equality or a maximum distance of certain attributes, while positive rules require a (composite) similarity to be above a certain threshold. Listing 6.2 shows an exemplary `duplicate detection` operator and the corresponding `stable matching` operator created by the following method.

Intuitively, this algorithm assumes that at least one rule of the `duplicate detection` operator contains an overall similarity measure, which is most suited to have the highest priority in the `stable matching` operator. Negative rules are primarily used to filter non-matches and act as tie-breaker. Therefore, the individual rules are classified as positive (comparison with $\geq \theta$, where θ is a constant real value), negative ($\leq \theta$ or

```
1 $duplicates = detect duplicates $p in $persons
      where intDiff($p.birthDate.year) == 0 if $p.birthDate
            and intDiff($p.birthDate.month) <= 1 if $p.birthDate
            and compositeSim($p) >= .8;
2 $stablePairs = match stably $d in $duplicates
      with compositeSim($d) >= .8
           and -intDiff($d.birthDate.year) >= 0
           and -(intDiff($d.birthDate.month) - 1) >= 0;
```

Listing 6.2: Similarity measure of the `duplicate detection` operator and inferred stable matching configuration in Meteor.

$= \theta$), or unknown (no comparisons). Positive rules can be directly added to the `stable matching` configuration. A negative rule $dist \leq \theta$ must be transformed to a positive rule with $-(dist - \theta) \geq 0$, while unknown rules are discarded. Now it remains to decide the order of importance of the rules. First, expressions originating from positive rules are sorted before all expressions from negative rules. Then, the expressions are ordered decreasingly by the number of sub-similarities. Hence, if we assume that the overall similarity measure contains most sub-similarities, it appears first and other rules only break ties or filter non-matches. In our experience, this method especially finds good stable matching configurations for cases where a non-matched can be easily filtered with one or two negative rules. The result contains significantly fewer false positives on the cost of a lower recall compared to a transitive closure.

6.3 Implementation with iterations in Stratosphere

Both clustering sub-operators rely on the workset iterations of Stratosphere for an efficient implementation. In this section, we briefly introduce the iteration concept and describe the implementation of both sub-operators.

6.3.1 Workset iterations

An iterative algorithm in a data flow engine implies a cyclic data flow, where the output from one second-order function is fed back into an earlier second-order function. However, any cycle usually significantly increases the complexity of the entire data flow engine: The definition of a data flow, the data flow execution, and fault tolerance need to be more sophisticated. Further, arbitrary cycles might run indefinitely and also strongly limit optimization opportunities. Therefore, current data flow engines provide only limited cycles to retain as many optimization opportunities as possible. Stratosphere supports two kinds of iterations as discussed in the following.

Bulk iterations denote the most common and easiest way to repeat parts of the data flow. A certain sub-data flow is simply repeated a certain fixed number of times. The developer has to define the iterative sub-data flow, the cyclic data flow edge, the initial input of the cyclic edge, and the number of repetitions. This simplicity allows

data flow engines, such as Stratosphere and Spark [Zaharia et al., 2010], to simply unroll the data flow to an acyclic graph for a small number of iterations.

However, for our clustering algorithms bulk iterations are unsuited, because the number of iterations is not fixed and can be as high as the number of vertices in the dataset. Additionally, our algorithms usually converge quickly to the result for most parts of the graph and actively process only on a small part of the graph (especially in long tail distributions of cluster sizes). Bulk iterations operate on the entire graph, so that a significant data volume is shipped unnecessarily.

Therefore, modern data flow engines additionally offer **workset iterations** [Ewen et al., 2012; Zaharia et al., 2010]. Additionally to the information needed for bulk iterations, developers define and update a *workset* that describes the active part of the data to be processed. When the workset is empty, the iteration terminates and allows the data flow engine to dynamically decide when to stop. Additionally, Stratosphere maintains a *solution set* that is updated by each iteration. In particular, temporary solution records may be replaced by other solution records using a key expression.

Algorithm 6.2 outlines the workset iteration as developed by Ewen et al. [2012]. Developers provide the iterative plan, the initial work- and solution sets, the number of iterations, and a key expression to identify updates to temporary solution records. The data flow engine first analyzes the iterative plan and separates the second-order functions that depend on the work- and solution set from the second-order functions that are independent and thus yield a constant result in each iteration. The data flow engine then executes the constant functions and feeds the result to the iterative part together with the initial work- and solution set.

Input : Plan p, initial workset w_0, initial solution set s_0,
maximum number of iterations i_{max}, solution key expression key_s
Output: Solution set s

1 $i \leftarrow 0$;
2 $p_{step} \leftarrow$ second-order functions in p **dependant** of work- or solution set;
3 $p_{const} \leftarrow$ second-order functions in p **independant** of work- or solution set;
4 $in_{const} \leftarrow$ execute p_{const};
5 **while** w_i not empty and $i \leq i_{max}$ **do**
6 $\quad s_\delta, w_{i+1} \leftarrow$ execute p_{step} with w_i, in_{const}, and s_i;
7 $\quad s_{i+1} \leftarrow$ update s_i with s_δ and solution key expression key_s;
8 $\quad i \leftarrow i + 1$;
9 **end**
10 $s \leftarrow s_i$;

Algorithm 6.2: Workset iteration framework.

In each iteration, the iterative part ingests the current workset, performs its computations, and creates a new workset. Further, it emits a *solution set delta* a set of resulting records that may have changed in comparison to the last iteration. The iteration engine adds the delta to the solution set, while removing all old records that have the same key as at least one record in the delta. The iterative part may use the current solution

set for its computation, but it can also completely rely on the workset to coordinate the iterations. In the following two sections, we describe the implementation of both clustering operators with workset iterations.

6.3.2 Transitive closure

The iterative, parallel implementation of the `transitive closure` operator can be intuitively best described by observing the vertices of the graph. First, each vertex receives a random cluster ID indicating that the vertex is in a singleton cluster. Then, clusters that are connected through edges are repeatedly merged. Merging clusters occurs implicitly for each vertex by exchanging the ID with the direct neighbors and changing the ID to the smallest ID of the neighborhood. This algorithm terminates if no node changes its ID and thus all nodes from the same cluster share the same ID.

The algorithm fits well into workset iterations: Only vertices that changed their ID in one iteration need to propagate their new ID in the next iteration, because only these nodes may change their neighbors. Consequently, the algorithm needs at most n iterations, where n is the number of edges in the longest acyclic path in the graph, since IDs can be propagated only n times along this path. For data cleansing and integration tasks, this n should be rather small, because either the settings are rather strict and therefore all paths are rather short or they are quite lenient and then many cycles would occur.

Figure 6.4 depicts the implementation of the `transitive closure` operator with the workset iterations of Stratosphere. The first three operators extract the vertices from the pairs, deduplicate them, and assign the initial ID. The iteration uses these vertices as the initial work- and solution set. In the iteration, two `joins` first find neighboring vertices with the duplicate pairs. Because our graph is undirected, we need two joins to find pairs where the workset vertex is the first or the second vertex. Then, a `grouping` operator collects all neighbors and chooses the minimal ID. Vertices with a different ID are first retained with a `selection`, subsequently updated with a `transformation`, and finally assigned to the next workset and solution set delta. After the iteration, a final `grouping` operator collects all vertices with the same ID in an array to form disjoint clusters.

6.3.3 Stable Matching

The `stable matching` operator uses the `transitive closure` operator to find connected components and post-processes them to determine the best matches. There are two implementations depending on whether the `stable matching` operator was invoked by a `record linkage` operator or a `duplicate detection` operator. We discuss only the `record linkage` implementation, because the `duplicate detection` operator can easily be derived from it.

The implementation shown in Figure 6.5 also uses a workset iteration, where the workset contains unmatched records and the solution set the best matching pairs. The Sopremo plan starts by tagging each record in all pairs by its source. Pairs originating

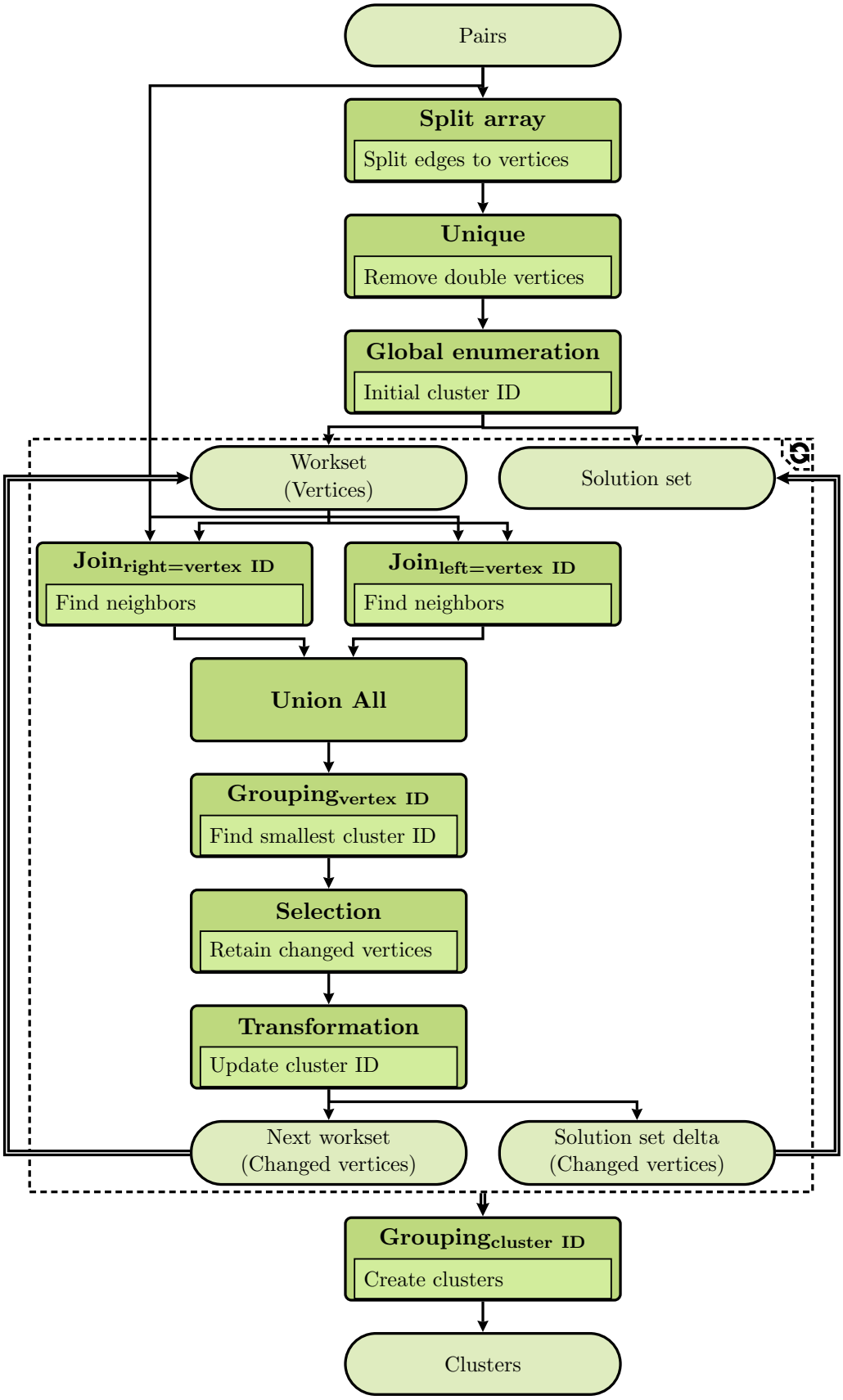


Figure 6.4: `Transitive closure` with Stratopshere's workset iteration.

from the `record linkage` always hold records from the first source in the first position of the array. After tagging the records, the algorithm invokes the `transitive closure` operator. Then, each cluster is assigned an ID and split into records. Each record contains the source ID, cluster ID, and the position in the cluster array.

The workset iteration now repeatedly picks the best pairs and removes the contained records from the workset. Initially, the workset contains the previously extracted records. The iterative sub-plan first separates records from the first and second source with two `selection`s, generates all pairs with a `join` over the cluster ID, and creates their similarity array. A subsequent `selection` removes all pairs that do not satisfy the thresholds. Two `grouping` operators then group all pairs of records from the first and second input and determine the preferred partner. To achieve good scalability, the operators use a secondary sort over the similarity array in descending order and immediately pick the first element. The following `join` then finds mutual preferences and adds them to the solution set. Additionally, separate `array split` and `difference` operators find all records that have no valid join partner anymore. Finally, the iteration creates the next workset by removing all records of the best matches and records without join partners from the workset with the `array split` and `difference` operators. For the `stable matching` operator, the solution set already contains all matched pairs.

6.4 Optimization of the clustering operator

In the remainder of the chapter, we integrate the `clustering` operator into the Sopremo optimizer. Since we primarily design the operator as a sub-operator of the `duplicate detection` and `record linkage` operator, our reordering rules mostly cover the combined operator.

6.4.1 Operator taxonomy

The algebraic properties of the `duplicate detection` and `record linkage` operators change completely when incorporating clustering. For **stable matching** `duplicate detection` (SDD), we can define only the basic sub-operator relationships known from the unclustered `duplicate detection` operators. A *naïve* SDD can be simulated with a *blocking* SDD on a constant blocking key or a Sorted Neighborhood Method (SNM) with a window larger than the number of records $w \geq |\mathcal{R}|$. Other operators cannot be simulated with SDD.

The variants of the transitively closed `duplicate detection` (TDD) share the same hierarchical relationships. Additionally, the TDD-Naïve represents a generalized (similarity) grouping operator. When using a similarity measure over one attribute, it can simulate a *similarity grouping* [Silva et al., 2009], which in turn can perform the normal `grouping` by using a threshold of 1 in the similarity expression. Note that indeed the result sets are not completely the same, when using a `duplicate detection` to simulate a `grouping`: Clusters with one element are not explicitly returned by the `duplicate detection`, while the `grouping` operator performs a complete partitioning. Nevertheless, the operator taxonomy does not necessarily require that the results are the same,

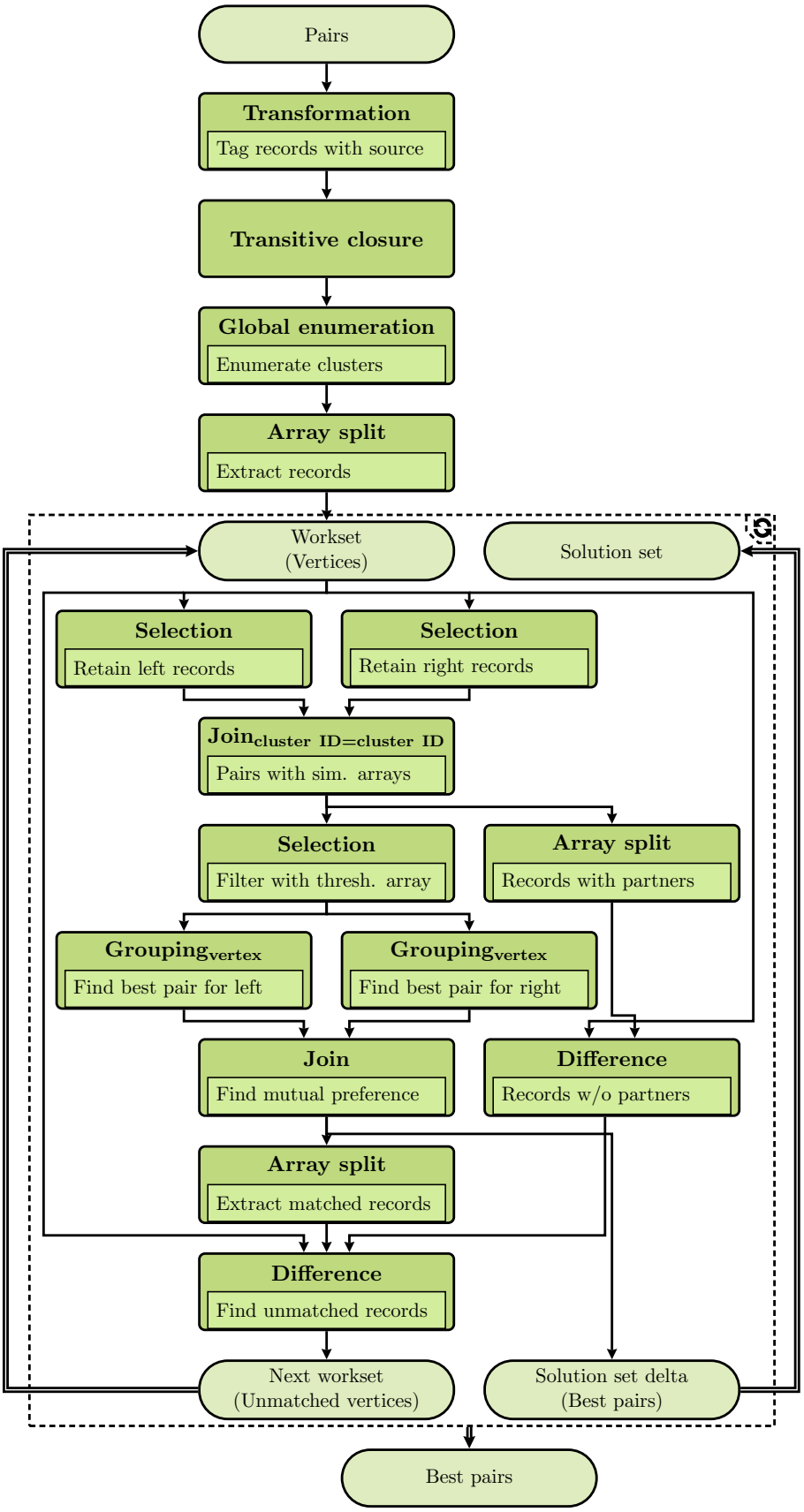


Figure 6.5: `Stable matching` with Stratopshere's workset iteration.

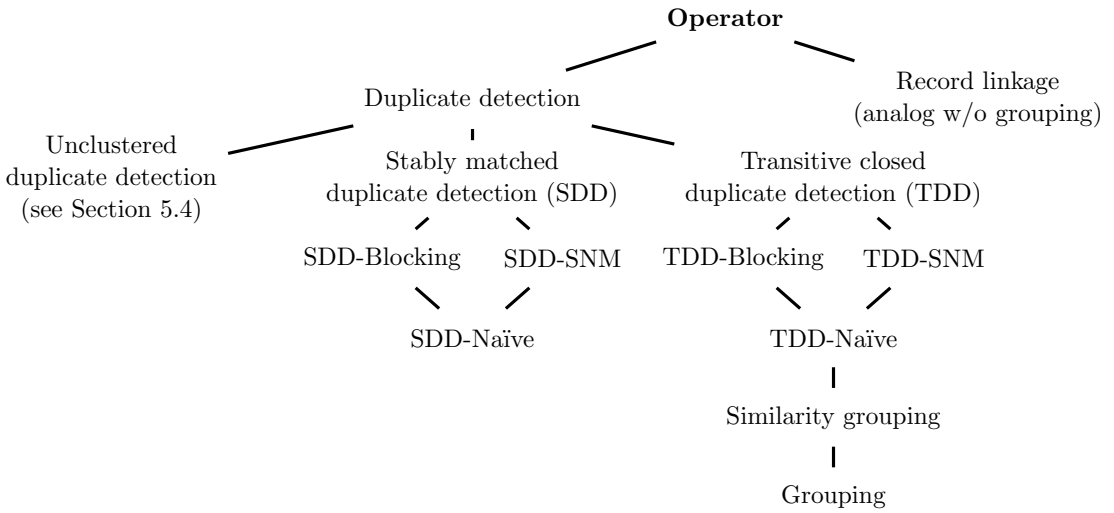


Figure 6.6: Operator taxonomy for `duplicate detection` operator with clustering (`record linkage` analog).

but only the behavior of the operators regarding the reordering rules. The operator hierarchy for the `record linkage` operator variants is the same, except that it does not generalize a `grouping` operator, because it processes two inputs. Figure 6.6 summarizes the operator taxonomy.

Properties

Both `clustering` operators ingest arrays of two records (duplicate pair) and produce arrays of at least two records (duplicate cluster). Hence, the schemata for the `clustering` operators remain the same. Further, the `stable matching` operator is **idempotent**: When the input pairs contain only the best matches, it obviously returns them again. The `transitive closure` operator should be **idempotent** by definition, but the actual implementation would produce different results. We could change the operator to split incoming clusters and thus render it also idempotent, but this change would degrade the performance and not justify the additional algebraic properties for unlikely cases with successive `clustering` operators.

For a `duplicate detection` operator using the `clustering` operator, the schema changes from record to array. Therefore, we can attribute only a **commutativity** property to the `duplicate detection` operator with `clustering` as we show later. More properties become available by considering the `duplicate detection` operator together with the `fusion` operator (see Chapter 7 for more details). The `fusion` operator merges the different records in a duplicate cluster to one consistent record, so that the schema of the resulting record corresponds to the schema of the input records of the `duplicate detection` operator. In particular, the `duplicate removal` operator combines both operators and allows us to elegantly define reordering rules and algebraic properties. In the

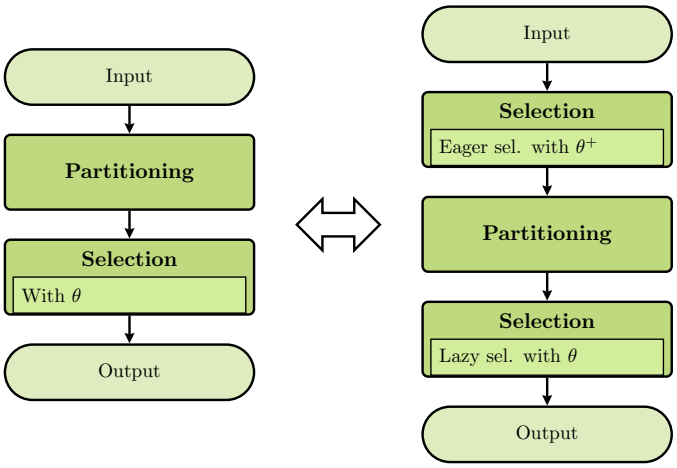


Figure 6.7: Eager-lazy pushing of a selection through the partitioning operator.

following, we review reordering rules and properties of the combination of the `duplicate detection` and `clustering` operators.

6.4.2 Reordering rules

For partitioning operators, eager-lazy optimization ([Yan and Larson, 1995], discussed in Section 3.4) becomes the foundation for reordering rules. We split preceding operators into an eager part that can be pushed towards the data sources and a lazy part that remains after the partitioning operator. Therefore, the eager part must not remove any record, which may contribute to the output.

Eager-lazy optimization of the cluster operator

Hueske et al. [2012] defined the *group preservation* property for general Map/Reduce data flows, which basically guarantees that a pushed operator either retains or removes all records that belong to one group in the end. For our operators, we can relax the condition and require that only those records need to be retained that also belong to a retained group, because we still have the lazy part that filters superfluous groups in the end. In particular, we can use any *eager condition* θ^+ for a selective preceding operator with condition θ to remove pairs of records that surely do not contribute to the final result. Figure 6.7 summarizes the eager-lazy optimization for a selection, on which we focus for the remainder of this section. For a dataset $\mathcal{D}$ of duplicates, we know:

$$\sigma_\theta(cl(\mathcal{D})) \equiv \sigma_\theta(cl(\sigma_{\theta^+}(\mathcal{D})))), \quad \text{(Rule I)}$$
$$\text{with } \theta^+ \colon \forall r_1, r_2 \in C \subseteq \mathcal{D} : \theta(C) \Rightarrow \theta^+(\{r_1, r_2\})$$

The eager condition θ^+ guarantees that every pair of records that is selected by θ, must also be selected by θ^+. Obviously, deriving θ^+ from θ is non-trivial.

Eager conditions

To apply the reordering rules, the optimizer needs to derive eager conditions θ^+ from the preceding selection condition θ. Since θ is applied to clusters $C \in \mathcal{C}$ of a dataset $\mathcal{R}$, we can distinguish two cases for a sub-condition φ.

1. If **all** elements of a cluster need to satisfy φ (i.e., $\theta : \forall r \in C : \varphi(r)$), we can safely set $\theta^+ = \varphi$, since $\theta(C) \Rightarrow \forall r \in C : \varphi(r)$, which corresponds to the definition of θ^+.

2. If only **some** elements need to satisfy φ (i.e., $\theta : \exists r \in C : \varphi(r)$), we need to find a θ^+ that selects all records $\mathcal{R}^{\theta^+}$ that may be transitively connected to all records in $\mathcal{R}$ that satisfy φ.

In the worst case, $\mathcal{R}^{\theta^+} = \mathcal{R}$ and the selection effectively cannot be pushed. For example, if we select the zip code 12345 and allow a Levenshtein distance of 1, all other zip codes are created in a sequence of steps. Nonetheless, there are many cases where we can clearly limit $\mathcal{R}^{\theta^+}$. The following is a non-exhaustive list of possibilities.

Negative rules with 0 distances: To find duplicate politicians in `US Earmarks`, `US Congress`, and `Freebase`, we specified that the birth year must be the same for a match. When selecting politicians born in 1970, all records in a cluster must contain the year 1970 and we can therefore use $\theta^+ = \varphi$.

Limited domains: Relaxation for conditions may also be limited, if the discrete domain contains only few values. For instance, the party attribute contains few different values even with typographic errors. Therefore, not all values can be transitively generated, for example, with a Levenshtein of 1. In these cases, $\mathcal{R}^{\theta^+}$ should be comparably small and we can rewrite $\varphi(r)$ as a disjunction of all relevant values.

Limited cluster size: If an upper bound for a cluster size is given $size_{max}$, distance conditions can be *relaxed* accordingly. The maximum cluster size may be inferred from the `duplicate detection` configuration, from (another) `clustering` algorithm, or for fuzzy rewrites through statistics (see next section). Assume that we allow a difference of 1 in the month of the politicians and the maximum cluster size is 3, we retain only politicians born in April to August, when selecting politicians born in June.

Further, if θ is a conjunction of conditions, we may apply these techniques individually on the conditions. Conditions that may not be transformed to an eager condition cannot be pushed at all.

Duplicate detection with clustering

We can combine the first rule for the `clustering` operator with the rules of the unclustered `duplicate detection` operator (see Section 5.4, Rule I ff). Again, for SNM we

might receive more duplicates when removing records from the input and therefore need separate rules.

$$\sigma_\theta(dd_{cs,sim,cl}(\mathcal{R})) \equiv dd_{cs,sim,cl}(\sigma_{\theta^+}(\mathcal{R})) \quad \text{(Rule II)}$$
$$\text{with} \quad cs \in \{naive, blocking\}$$
$$\theta^+ : \forall r \in C \subseteq \mathcal{R} : \theta(C) \Rightarrow \theta^+(r)$$

$$\sigma_\theta(dd_{snm,sim,cl}(\mathcal{R})) \overset{+}{\Rrightarrow} dd_{snm,sim,cl}(\sigma_{\theta^+}(\mathcal{R})) \quad \text{(Rule III)}$$

$$\sigma_\theta(dd_{snm,sim,cl}(\mathcal{R})) \overset{-}{\Rrightarrow} dd_{snm',sim,cl}(\sigma_{\theta^+}(\mathcal{R})) \quad \text{(Rule IV)}$$

$$\sigma_\theta(dd_{snm,sim,cl}(\mathcal{R})) \equiv dd_{snm,\theta^+ \wedge sim,cl}(\mathcal{R}) \quad \text{(Rule V)}$$

Note that the eager condition θ^+ now needs to ensure that records are retained instead of pairs, since we are processing the input dataset. For `record linkage`, the equivalent rules have up to two eager conditions for the respective inputs.

Commutativity

The stable matching `record linkage` operator has by definition an asymmetric preference and therefore cannot be commutative. However, transitively closed `record linkage` and `duplicate detection` use a bag-unified input and the inputs can therefore be swapped for the symmetric candidate selections naïve and blocking. The `duplicate detection` operator bag-unifies the input in any case and is therefore commutative for every clustering.

$$rl_{blocking,sim,tc}(\mathcal{R}_1, \mathcal{R}_2) \equiv rl_{blocking,sim,tc}(\mathcal{R}_2, \mathcal{R}_1) \quad \text{(Rule VI)}$$
$$rl_{naive,sim,tc}(\mathcal{R}_1, \mathcal{R}_2) \equiv rl_{naive,sim,tc}(\mathcal{R}_2, \mathcal{R}_1) \quad \text{(Rule VII)}$$
$$dd_{cs,sim,cl}(\mathcal{R}_1, \mathcal{R}_2) \equiv dd_{cs,sim,cl}(\mathcal{R}_2, \mathcal{R}_1) \quad \text{(Rule VIII)}$$

In comparison to the unclustered operators, the concept of associativity is not applicable, because clusters cannot easily be further (similarity) joined. The `duplicate removal` operator, however, allows these reorderings again (see Chapter 7), since the intermediate results are fused into records again. Similarly, there cannot be meaningful rules with transformations.

6.4.3 Estimating the number and sizes of clusters

The remainder of the chapter estimates the cardinality and costs of the operators. Because final cluster sizes determine the runtime of the clustering, we start by estimating the outcome of the operators. Similar to the estimation of unclustered `duplicate detection`, we first develop a sample model and then use this model to create an efficient and effective estimation algorithm. This section mainly presents our *Duplicity Estimation* algorithm [Heise et al., 2014]. Primarily, we estimate the *cluster size histogram*.

Definition 6.4 (Cluster Size Histogram)**.** The cluster size histogram $\mathcal{H}_\mathcal{R}$ is the absolute histogram over the number of clusters of a certain size from the set of all clusters $\mathcal{C}_\mathcal{R}$.

We denote the i-th entry with $\mathcal{H}^i_{\mathcal{R}} = |\{C \in \mathcal{C}_{\mathcal{R}} : |C| = i\}|$ corresponding to the number of clusters of size i.

Effects of sampling on the cluster size histogram

We generalize the probabilistic model of Section 5.4 to arbitrary cluster sizes. Again, we first consider naïve `duplicate detection` with `transitive closure` and denote with $n = |\mathcal{S}|$ the number of samples, $N = |\mathcal{R}|$ the size of the original dataset, and with r or r_i single records in $\mathcal{R}$. Similar to estimating the number of duplicate pairs, we use the number of permutations that can occur during random sampling.

Our problem is related to the multivariate hypergeometric distribution; that is, drawing a sample without replacement from a multi-type population. We can map our problem to drawing a number of colored balls without replacement from an urn: Each different duplicate cluster of size i is represented by i balls of the same color. However, we are not interested in how many balls of a specific color are drawn but only in how many balls of the same color would be drawn on average if the random sampling experiment was repeated infinitely many times.

We can thus model our expected value by using the indicator function $[\![|C| = i]\!]$ that is 1 iff the size of a duplicate cluster C is i and 0 otherwise. We iterate over all possible events that can yield a duplicate cluster of size i in a sample of size n. Each outcome is weighted with $\binom{N}{n}^{-1}$, i.e., the reciprocal to the number of possibilities of randomly sampling n out of N elements without replacement:

$$\mathbf{E}[\mathcal{H}^i_{\mathcal{S}}] = \frac{1}{\binom{N}{n}} \sum_{C \in \mathcal{C}_{\mathcal{R}}} \left(\sum_{\mathcal{S} \subseteq \mathcal{R} \wedge |\mathcal{S}| = n} [\![|C \cap \mathcal{S}| = i]\!] \right) \tag{6.1}$$

The first sum iterates over all clusters in $\mathcal{R}$ and the second sum enumerates all possible sample outcomes w.r.t. a specific cluster. Now, we can replace the second sum with the probability that we draw i elements from a cluster in the original dataset, which can be expressed as:

$$\mathbf{E}[\mathcal{H}^i_{\mathcal{S}}] = \frac{1}{\binom{N}{n}} \sum_{C \in \mathcal{C}_{\mathcal{R}}} \binom{|C|}{i} \binom{|\mathcal{R} \setminus C|}{n-i} \tag{6.2}$$

Intuitively, there are $\binom{N}{n}$ possible sample permutations. For a cluster of size i in the sample, we draw i records from the cluster C, add $n - i$ records from the remaining dataset $\mathcal{R} \setminus C$ to fill up the sample, and normalize the result over all sample permutations. Because the probabilities of drawing i elements from two different clusters with the same sizes are the same, we can simplify the calculation by collapsing the cases. For each cluster size k in the original dataset, we calculate the probability once and multiply it by the number of clusters of the given size $\mathcal{H}^k_{\mathcal{R}}$.

$$\mathbf{E}[\mathcal{H}^i_{\mathcal{S}}] = \sum_{k=i}^{|R|} \mathcal{H}^k_{\mathcal{R}} \frac{\binom{k}{i}\binom{N-k}{n-i}}{\binom{N}{n}} \tag{6.3}$$

Example	Original histogram $\mathcal{H}^1_{\mathcal{R}}$	$\mathcal{H}^2_{\mathcal{R}}$	$\mathcal{H}^3_{\mathcal{R}}$	$\mathcal{H}^4_{\mathcal{R}}$	Expected histogram from sampling $\mathcal{H}^1_{\mathcal{S}}$	$\mathcal{H}^2_{\mathcal{S}}$	$\mathcal{H}^3_{\mathcal{S}}$	$\mathcal{H}^4_{\mathcal{S}}$
1	1,000	0	0	0	100.000	0.000	0.000	0.000
2	0	500	0	0	90.090	4.955	0.000	0.000
3	0	0	0	250	73.095	12.088	0.878	0.023
4	600	100	40	20	93.604	3.030	0.109	0.002

Table 6.2: Four different cluster size histograms for a dataset of size 1,000 (left) and the expected histograms from sampling 100 records (right).

For ease of presentation, we assume $\binom{a}{b} = 0$ for $a < b$ in this section. Alternatively, ranges in sum formulas must be adjusted to guarantee $a \geq b$. Now we are interested in understanding which clusters the sampled elements come from. Hence, we define a random variable $X^i_{S,n}$ representing whether an element from a cluster of size i occurs in a sample S of size n. The probability of this event happening is:

$$p(X^i_{S,n} = 1) = \frac{i}{n} \sum_{k=i}^{|R|} \mathcal{H}^k_{\mathcal{R}} \frac{\binom{k}{i}\binom{N-k}{n-i}}{\binom{N}{n}} \tag{6.4}$$

As discussed earlier, the sum in the above formula represents all possibilities to generate a cluster of size i in a sample of size n. Given that we have such a sample of size n at hand (in which there occurs a cluster of size i), the probability of picking an element from an i-cluster is i/n.

In Table 6.2, we assume four different cluster size histograms of a dataset with 1,000 records and calculate the corresponding *expected* histograms with Equation 6.3 for a sample of 100 records. The first row constitutes a sanity check: If all 1,000 records are duplicate-free, all 100 samples also must be duplicate-free. The second row assumes 500 duplicate pairs, from which 4.95 are drawn on average (this result matches the estimation of the number of pairs in Section 5.4). Interestingly, in the third row, even though the dataset consists of 250 clusters with size four, sampling one complete cluster of size four is highly unlikely with a sample size of 100 (one in 43 sample runs). Lastly, the expected sampling histogram of a somewhat realistically distributed dataset in the fourth row shows that sampling clusters of size 3 and 4 is even more unlikely.

Extrapolating the sampled histogram

So far, we deduced Equation 6.3 to calculate the expected histogram from a given histogram. The calculation may be interpreted as multiplication of a matrix with transition probabilities $T_{\mathcal{S}}$ and the cluster size histogram interpreted as a vector $\vec{\mathcal{H}}_{\mathcal{R}}$.

$$\vec{\mathcal{H}_{\mathcal{S}}} = T_{\mathcal{S}}\vec{\mathcal{H}_{\mathcal{R}}} \tag{6.5}$$

$$T_{\mathcal{S}} = \begin{pmatrix} s_{1,1} & s_{1,2} & \cdots & s_{1,k} \\ 0 & s_{2,2} & \cdots & s_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & s_{i,k} \end{pmatrix} \tag{6.6}$$

$$s_{i,k} = \frac{\binom{k}{i}\binom{N-k}{n-i}}{\binom{N}{n}} \tag{6.7}$$

$$\vec{\mathcal{H}_{\mathcal{R}}} = (\mathcal{H}^1_{\mathcal{R}}\mathcal{H}^2_{\mathcal{R}}\cdots\mathcal{H}^i_{\mathcal{R}})^T \tag{6.8}$$

We could now treat the matrix multiplication as a system of linear equations, which can be exactly solved if the sample size is equal to or greater than the maximum cluster in $\mathcal{R}$, i.e., $|\mathcal{S}| \geq max_i(\mathcal{H}^i_{\mathcal{R}} > 0)$, and we would sample the expected numbers $\mathbf{E}[\mathcal{H}_{\mathcal{S}}]$.

However, as seen in the last examples of Table 6.2, it becomes increasingly improbable to randomly sample larger clusters, resulting in two drawbacks of the exact approach. First, we could sample a large cluster despite the low odds and would heavily overestimate the number of large clusters in the original dataset. In fact, we would even receive an impossible high number of clusters with the exact method that needs to be compensated with negative histogram counts of smaller clusters. Second and more probable, we would not sample a large cluster at all and thus underestimate the number of larger clusters. Further, the large variance of the sampling (see Appendix A.4) requires robust methods.

Probabilistic solution with random walk

We present a solution based on a random walk approach to probabilistically estimate from which original cluster a sample cluster has been drawn. The main intuition is to maximize the knowledge that we can obtain from a certain random sample and then successively enlarge the sample until we are sufficiently confident about the estimation.

To extrapolate the sample histogram $\mathcal{H}_{\mathcal{S}}$ to the estimation $\mathcal{H}_{\mathcal{R}}$, we use the matrix $T_{\mathcal{R}}$, whose components are the conditional probabilities $p(X^k_R = 1 \mid X^i_{S,n} = 1)$, where $X^k_R = 1$ is a random variable representing whether a cluster of size k is present in $\mathcal{R}$.

$$\vec{\mathcal{H}_{\mathcal{R}}} = T_{\mathcal{R}}\vec{\mathcal{H}_{\mathcal{S}}} \tag{6.9}$$

$$T_{\mathcal{R}} = \begin{pmatrix} p_{1,1} & 0 & \cdots & 0 \\ p_{2,1} & p_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ p_{k,1} & p_{k,2} & \cdots & p_{k,i} \end{pmatrix} \tag{6.10}$$

$$\begin{aligned} p_{k,i} &= p(X^k_R = 1 \mid X^i_{S,n} = 1) \\ &= \frac{p(X^i_{S,n} = 1 \mid X^k_R = 1) \cdot p(X^k_R = 1)}{\sum_{j=i} p(X^i_{S,n} = 1 \mid X^j_R = 1) \cdot p(X^j_R = 1)} \end{aligned} \tag{6.11}$$

The last equation shows the two pieces of information that we need to calculate $T_{\mathcal{R}}$. The first term is the conditional probability $p(X^k_R = 1 \mid X^i_{S,n} = 1)$ that corresponds to

the mirrored entry of $T_{\mathcal{S}}$, which in turn depends only on $|R|$ and $|S|$. The second term $p(X_R^k = 1)$ corresponds to an entry in $\vec{\mathcal{H}}_{\mathcal{R}}$, which we actually want to estimate. In the denominator, we normalize over all cluster sizes in the sample.

Because we need prior knowledge of $\vec{\mathcal{H}}_{\mathcal{R}}$ to calculate $T_{\mathcal{R}}$, the iterative calculation of $\vec{\mathcal{H}}_{\mathcal{R}}$ is an obvious choice. We combine the transition matrices from Equations 6.5 and 6.9 to a matrix M_R:

$$\vec{\mathcal{H}}_{\mathcal{R}} = T_{\mathcal{R}} T_{\mathcal{S}} \vec{\mathcal{H}}_{\mathcal{R}} = M_R \vec{\mathcal{H}}_{\mathcal{R}} \tag{6.12}$$

We clearly see that $\vec{\mathcal{H}}_{\mathcal{R}}$ is the principal eigenvector of M_R. Therefore, we propose a *random walk* approach to start with a rough initial estimation $\mathcal{H}_{\mathcal{R}(0)}$ and iteratively refine the estimation until a convergence criterion is met. Algorithm 6.3 consists of two nested convergence loops. In the outer loop (Lines 2–13), the algorithm first enlarges the sample and calculates the relative histogram $\mathcal{H}_{\mathcal{S}}$ by performing naïve duplicate detection on the sample. It also calculates the sampling transition matrix $T_{\mathcal{S}}$, which depends on the current size of the sample and $|R|$.

Input : Initial estimate $\vec{\mathcal{H}}_{\mathcal{R}(0)}$,
convergence threshold ε,
minimum number of samples $minS$
Output: Estimated original histogram $\vec{\mathcal{H}}_{\mathcal{R}}$

```
1  i, j ← 0;
2  repeat
3      j ← i;
4      S_(j) ← sample;
5      calculate H⃗_S(j) from S ;
6      calculate T_S for current sample size ;
7      [1] /* Power iterations to find H⃗_R = T_R H⃗_S                    */
8      repeat
9          calculate T_R(i) with H⃗_R(i);
10         H⃗_R(i+1) ← T_R(i) H⃗_S(j);
11         i ← i + 1 ;
12     until ||H⃗_R(i) − H⃗_R(i−1)||_1 < ε or i > j + 100;
       /* Enlarge sample until globally converged                       */
13 until ||H⃗_R(j) − H⃗_R(i)||_1 < ε and |S| ≥ minS;
```

Algorithm 6.3: Two-phase random walk.

The inner loop (Lines 8–12) performs the *Power iterations* [Langville and Meyer, 2003] of the random walk to find a suitable $\mathcal{H}_{\mathcal{R}}$ for the current $\mathcal{H}_{\mathcal{S}}$. In each iteration, the algorithm first calculates the current transition matrix $T_{\mathcal{R}(i)}$ with the previous estimate $\vec{\mathcal{H}}_{\mathcal{R}(i)}$. This transition matrix is then used to calculate a better estimate $\vec{\mathcal{H}}_{\mathcal{R}(i+1)}$. The inner loop terminates when the estimate converges; that is, the L1-distance between two successive estimations is below a given threshold ε (i.e., fewer than ε clusters are

changed), or 100 Power iterations are reached, which avoids overfitting to early samples and speeds up the overall computation.

Having an estimate for a certain, possibly small sample yields a high risk of overfitting to a poor sample. We thus rerun the random walk several times with the outer loop and enlarge the sample until 1) the estimates also converge across samples and 2) we sampled a given minimum number of records. The first criterion causes the algorithm to eventually converge to a well fitting estimate for the current sample and the second criterion avoids the algorithm being stuck in local maximum for poor initial samples.

Properties of the random walk

Algorithm 6.3 iteratively computes the principal eigenvector of the matrix M_R (see Equation 6.12). To show that this computation is well-defined, we have to show that real (non-imaginary) principal eigenvectors of the matrix exist.

We know that every quasi-positive, and thus every positive, matrix is ergodic and therefore has a principal eigenvector. For M_R, it is straight-forward to show that indeed all components have to be positive. All components of $T_{\mathbb{S}}$ and $T_{\mathbb{R}}$ are non-negative, so that no component in the product can be negative. Further, the first row of $T_{\mathbb{S}}$ and the first column of $T_{\mathbb{R}}$ are filled with positive numbers. Because the matrix multiplication each component in M_R involves the multiplication of the first row of $T_{\mathbb{S}}$ and the first column of $T_{\mathbb{R}}$, we can conclude that each component must be positive. Intuitively, from any estimated cluster of size k, we could have sampled exactly one element, which could have been drawn from any other cluster size. Thus, we can change the estimation for one record in each iteration to a completely different cluster size.

Moreover for such matrices, the principal eigenvectors can be effectively approximated through Power iterations [Langville and Meyer, 2003], which are represented also in the inner loop of the algorithm. These facts lay the mathematical foundation for the correctness of Algorithm 6.3.

Theorem 6.1. *Algorithm 6.3 approximates the principal eigenvector of the matrices M_R from Equation 6.12.*

The estimation model reliably estimates the actual duplicate histogram even for poor initial estimates $\vec{\mathcal{H}}_{\mathcal{R}(0)}$ as we experimentally show in Section 6.5. We also show that better initial estimates lead to faster convergence.

Alternative configurations

The presented cluster estimation algorithm assumes a naïve `duplicate detection` with `transitive closure` operator. In the following, we discuss how the algorithm can be applied to the other variations. Because the `stable matching` operator uses the `transitive closure` operator in the implementation, we also rely on the estimation of the `transitive closure` operator to estimate the cardinality of the `stable matching` operator. In particular, the `stable matching` operator splits the clusters given by a

`transitive closure` operator. We can easily give a lower and an upper bound of the number of pairs $\#pairs$ given a cluster size histogram $\mathcal{H}_{\mathcal{R}}$.

$$\sum_k \mathcal{H}^k_{\mathcal{R}} \leq \#pairs \leq \sum_k \mathcal{H}^k_{\mathcal{R}} \cdot \left\lfloor \frac{k}{2} \right\rfloor \tag{6.13}$$

The minimum bound occurs when all clusters are star-shaped; that is, all records prefer only one central record. After this record chooses its partner, all remaining records become singleton clusters. Therefore, the number of pairs is the number of clusters. In contrast, the maximum number of pairs assumes that each record finds a partner (except one in uneven cluster sizes). We assume that in real-world datasets, the latter case is more realistic: Although some record pairs should definitively not be matched, most records in smaller clusters are relevant. Therefore, we would choose upper bound as an estimate for the `stable matching` operator.

Non-naïve candidate selection techniques and multiple passes can be integrated by using the same candidate selections on the samples with adjusted settings in the same way as the estimation of duplicate pairs in Section 5.4. Further, the `record linkage` operator can be interpreted as a `duplicate detection` operator with a secondary candidate selection that avoids matches within the same dataset. Therefore, we can treat `record linkage` operators as `duplicate detection` operator and use the secondary candidate selection as an additional negative similarity rule.

6.4.4 Cost estimation

For the `stable matching` operator, the most compute-intensive part is the `transitive closure` to compute the connected components. Selecting the best pairs from the clusters should have only a noticeable runtime for large clusters, which should be rather rare for the `record linkage` operator use case. Therefore, we primarily focus on estimating the runtime cost of the `transitive closure` operator.

Transitive closure

The cost of iterative operators with an unknown number of iterations are usually hard to estimate. However, in the case of the `transitive closure` operator, we can quite accurately estimate the runtime costs. First, we note that the maximum number of iterations is determined by the longest acyclic path in the graph, which in turn is bound by the maximum cluster size. We receive an estimate for the maximum cluster size by the cardinality estimation. Second, the runtime is determined by the number of vertices in the workset for each iteration.

We know that each iteration assigns the final cluster ID to at least one vertex in each cluster. Therefore, for each cluster of size k and iteration i, there are at most $max(0, k - i)$ vertices in the workset. For instance, a cluster of size 4 needs at most 3 iterations to propagate the minimum cluster IDs completely. Further, we need to know to how many neighbors a vertex propagates his ID. For a cluster of size k, the maximum number of neighbors of any vertex is $k - 1$.

$$|workset_i| \leq \sum_k \mathcal{H}^k_{\mathcal{R}} \cdot \max(0, k-i) \tag{6.14}$$

$$|edges_i| \leq \sum_k \mathcal{H}^k_{\mathcal{R}} \cdot \max(0, (k-i)\cdot(k-1)) \tag{6.15}$$

In Stratosphere, workset iterations build and reuse hash tables in joins using the constant input and probe the (changing) workset. For the `transitive closure` operator, the edges remain in-memory on each computation node and only the vertices are distributed. However, the edges need to be transferred and grouped on separate nodes to determine the new cluster ID per vertex. Hence, the network cost is bound by $|workset_i| + |edges_i|$ for each iteration.

$$\begin{aligned}
network_cost_{tc} &\leq \sum_{i \leq k_{max}} |workset_i| + |edges_i| \\
&= \sum_{i \leq k_{max}} \left(\sum_k \mathcal{H}^k_{\mathcal{R}} \cdot \max(0, k-i) + \sum_k \mathcal{H}^k_{\mathcal{R}} \cdot \max(0, (k-i)\cdot(k-1)) \right) \\
&= \sum_k \mathcal{H}^k_{\mathcal{R}} \sum_{i=0}^{k} \max(0, k\cdot(k-i)) = \sum_k \mathcal{H}^k_{\mathcal{R}} \cdot k \cdot \sum_{i=0}^{k} \max(0, (k-i)) \\
&= \sum_k \mathcal{H}^k_{\mathcal{R}} \cdot k \cdot \sum_{i=0}^{k} i \\
&= \sum_k \mathcal{H}^k_{\mathcal{R}} \cdot k \cdot \frac{k\cdot(k-1)}{2}
\end{aligned} \tag{6.16}$$

The CPU runtime linearly depends on the number of edges. Because the edges dominate the network cost, we can simply use the previous formula to estimate the CPU costs in terms of the number of records to process.

Stable matching

The `stable matching` operator uses the `transitive closure` and adds its own iterative part. In the following, we focus on the cost of the additional iteration. Similar to the `transitive closure` operator, we can argue that in each iteration at least one best pair of records is selected. Because a pair consists of two records, the `stable matching` operator converges twice as fast as the `transitive closure` operator. Therefore, the `stable matching` operator needs at most $\frac{k_{max}}{2}$ iterations. Further, the number of candidate pairs decreases quadratically, because they are generated completely from the workset.

$$|workset_i| \leq \sum_k \mathcal{H}^k_{\mathcal{R}} \cdot \max(0, k-2\cdot i) \tag{6.17}$$

$$|edges_i| \leq \sum_k \mathcal{H}^k_{\mathcal{R}} \cdot \max\left(0, \binom{k-2\cdot i}{2}\right) \tag{6.18}$$

We can safely assume that all records relevant for one cluster are local on one node, because they have been split from one cluster record immediately before the iteration starts. Therefore, the additional network cost for the iteration is negligible (communication overhead for iterations).

$$
\begin{aligned}
cpu\_cost_{sm} &\leq \sum_{i=0}^{\lfloor \frac{k_{max}}{2} \rfloor} |workset_i| + |edges_i| \\
&= \sum_{i=0}^{\lfloor \frac{k_{max}}{2} \rfloor} \left(\sum_k \mathcal{H}^k_{\mathcal{R}} \cdot \max(0, k - 2 \cdot i) + \sum_k \mathcal{H}^k_{\mathcal{R}} \cdot \max\left(0, \binom{k - 2 \cdot i}{2}\right)\right) \\
&= \sum_k \mathcal{H}^k_{\mathcal{R}} \sum_{i=0}^{\lfloor \frac{k}{2} \rfloor} \max\left(0, (k - 2 \cdot i)^2\right) = \sum_k \mathcal{H}^k_{\mathcal{R}} \cdot 4 \cdot \sum_{i=0}^{\lfloor \frac{k}{2} \rfloor} \max\left(0, (\frac{k}{2} - i)^2\right) \\
&\leq \sum_k \mathcal{H}^k_{\mathcal{R}} \cdot 4 \cdot \sum_{i=0}^{\frac{k}{2}} i^2 \\
&= \sum_k \mathcal{H}^k_{\mathcal{R}} \cdot 4 \cdot \frac{(\frac{k}{2} - 1)\frac{k}{2}(k-1)}{6}
\end{aligned} \tag{6.19}
$$

Compared to the total cost of the `transitive closure`, this additional CPU cost becomes only relevant for huge clusters or when CPU resources are much more limited than network resources.

6.5 Evaluation of duplicity estimation

In this section, we evaluate the duplicity estimation algorithm from Section 6.4 (on page 107) on three datasets with different characteristics to demonstrate the generality of our approach. The evaluation appeared in [Heise et al., 2014].

We first describe the datasets and the test setup and then examine how fast our estimates converge to the gold standard with different qualities of initial histograms. The main design goal of the estimation method is to save as many candidate comparisons as possible. Hence, we are mostly interested in the number of iterations of the outer loop in Algorithm 6.3, since this loop enlarges the sample and causes additional comparisons. Further, we observe the variance and discuss the accuracy. Lastly, we measure the number of iterations needed to achieve certain error bounds and measure the runtime of our method.

6.5.1 Datasets

The **CD** dataset consists of 750,000 CD records from FreeDB with a semi-automatic gold standard [Vogel et al., 2014] created by combining automatic labeling for easy-to-classify pairs and manual labeling for hard pairs. The dataset contains 55,323 duplicates

clusters, which are approximately power law-distributed with cluster sizes up to 50. We use it as the main dataset for our evaluation.

A much cleaner dataset is the **Customer** dataset containing 1,039,776 person records including 89,782 artificially polluted duplicate pairs. The dataset was generated by an industry partner to test (their) duplicate detection algorithms and simulates the integration result of three relatively clean, duplicate-free datasets with a small overlap.

On the other side of the spectrum is the **Cora** dataset with only 182 clusters in 1,878 bibliographic records and a maximum cluster size of 238 records. It is a real-world dataset with a published gold standard [Draisbach and Naumann, 2010]. Due to the relatively huge clusters, it tests the boundaries of our method.

6.5.2 Test setups

For each dataset, we used the corresponding gold standard to create a gold histogram. We use an oracle as the candidate comparison, which simulates a perfect similarity measure by looking up the result in the gold standard. Note, that we could have created other gold histograms with a naïve duplicate detection run with other candidate comparisons; the gold histogram would be generated with the result of that particular run. With the oracle, however, we receive the most realistic gold histogram. Further, through preliminary experiments we discovered that $\varepsilon = 10$ and $minS = 5$ show a good trade-off between accuracy and efficiency.

We repeat the estimation 100 times for each dataset with three different initial histograms and measure the estimated cluster size distribution after each iteration. In each iteration, we add $\sqrt{N}$ records from the total N records to the random sample. We choose the following three initial histograms with a maximum cluster size m.

Uniform: Each record has the same probability to be in a cluster of size k. Because the cluster of size k in $\mathcal{H}^k_{\mathcal{R}(0)}$ contains k records, we normalize by $\frac{1}{k}$: $\mathcal{H}^k_{\mathcal{R}(0)} = \frac{1}{k}\frac{N}{m}$

Power law: Half of the records are duplicate-free, a quarter is in pairs, an eights in triples and so on: $\mathcal{H}^k_{\mathcal{R}(0)} = \frac{1}{k}\frac{N}{2^k}\frac{2^m}{2^m-1}$. This is a realistic distribution for most datasets.

Inverse power law: We mirror the power law: $\mathcal{H}^k_{\mathcal{R}(0)} = \frac{1}{k}\frac{N}{2^{m-k+1}}\frac{2^m}{2^m-1}$. This is a highly unrealistic distribution, employed to test the robustness of our approach.

6.5.3 Different initial histogram

First, we measure the convergence of the estimation on the CD dataset in Figure 6.8 with the three different initial vectors. Note that we use square scales on the y-axis to better observe the relatively rare, larger cluster sizes. For further improved readability, we plotted in this and following figures only a few data points; the corresponding line is generated from data points for all 50 cluster sizes.

If we use the power law distribution (top), the estimate converges quickly to the gold histogram within ten iterations. Early iterations already result in good approximations, because the initial vector corresponds well to the gold histogram.

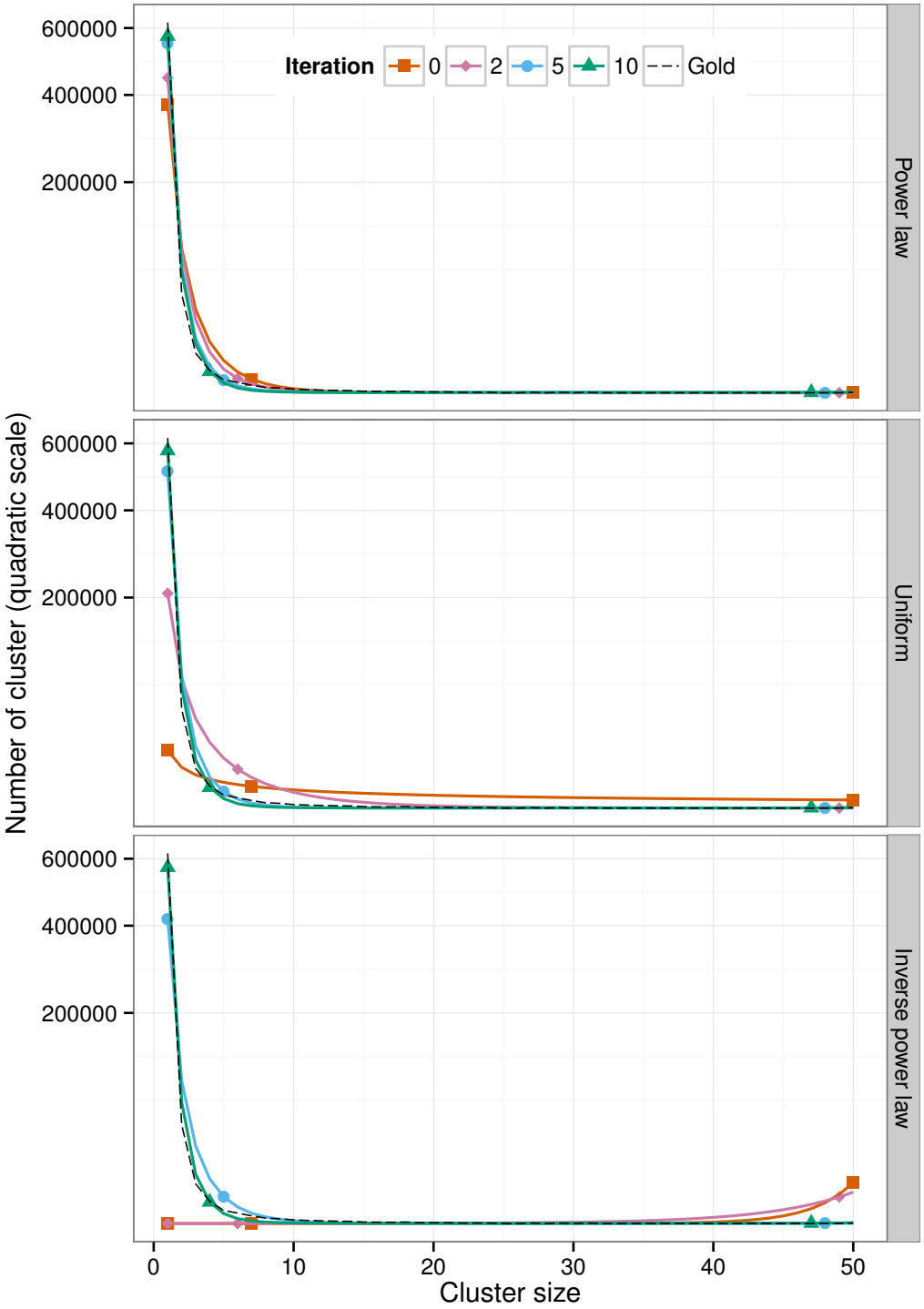


Figure 6.8: Convergence for 10 iterations on the CD dataset with the three different initial histograms (y-axis is square-scaled).

Figure 6.9: Standard deviation of the estimates for duplicates and non-duplicates in the Customer dataset.

The uniform distribution (middle) needs 12 iterations to converge. After two iterations, we can already see a power law distribution, which then quickly approaches the gold histogram.

The estimation algorithm needs five iterations to change the inverse power law (bottom) into a power law and a total of 15 iterations to converge to the gold histogram. Nevertheless, we want to emphasize that the algorithm still converges with a sample of only about 17,000 records (2.3%) to the correct result despite the completely wrong initial estimate.

Altogether, we can see that the algorithm reliably converges. The better the initial estimate, the faster the convergence occurs. For the Cora and Customer dataset, we observe similar convergence after 15 iterations. We also conducted a sanity check and used the gold histogram as the initial histogram and observed that the estimates do not diverge from the gold histogram.

6.5.4 Variance of sampling

In the next experiment, we examine how the estimates vary throughout the 100 aggregated runs on the Customer dataset. This dataset contains only two cluster sizes (namely sizes 1 and 2), which allows a clear visualization of variances. Figure 6.9 shows the standard deviation of the estimates at a specific iteration for the non-duplicates and the duplicate pairs and the three initial histograms. The average estimate quickly converges to the gold histogram within 15 iterations. From that point, the standard deviation decreases with additional iterations and is only $\approx 1\%$ after 50 iterations.

6.5.5 Exactness of results

The first two experiments indicate that our algorithm iteratively fits the estimation to the gold histogram. We now want to closely examine the estimates of the irregular

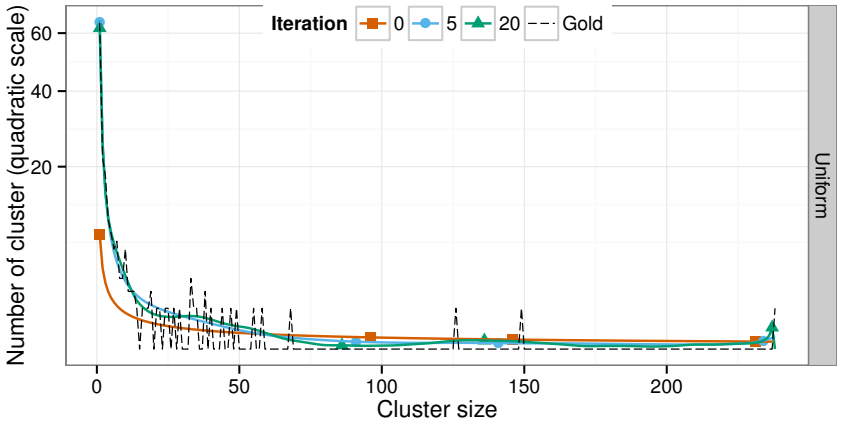


Figure 6.10: Convergence for 20 iterations on the Cora dataset. (y-axis square-scaled to highlight larger cluster sizes)

histogram of the Cora dataset. The histogram in Figure 6.10 exhibits a comparably flat power law-like shape with clear peaks between cluster sizes 30 to 60 as well as peaks (individual clusters) at 127, 148, and 238.

We see that after five iterations, the curve already approaches to the overall form of the distribution. In successive iterations the estimate is further refined until also the individual peaks at cluster sizes 127, 148, and 238 are approximated. Please note that the y-axis is again square-scaled, so that we can examine the low frequencies of 1-5 appropriately.

Obviously, our algorithm fails to exactly determine the cluster sizes even after 20 iterations on this dataset. Nevertheless, for most use cases, the estimate after five iterations may be enough: The power-law distribution is estimated well enough and there is at least one cluster with a size over 100, which may either indicate poor data quality or an ineffective duplicate detection configuration. Further, according to our experience, larger real-world datasets are distributed more evenly, so that the hard-to-estimate peaks are less probable.

6.5.6 Number of iterations

Depending on the use case, the number of iterations for an estimation may vary. In this experiment, we compare the number of iterations needed to meet different error bounds. We use two measures to assess the difference between the current estimation and the gold histogram. The *root mean square error* (RMSE) calculates the normalized squared error between the estimate of each cluster size and the actual value. Because of the power law distributions, this measures heavily favors smaller, more frequent cluster sizes. We define both measures over a gold histogram $\mathcal{H}_\mathcal{G}$ and the current $\mathcal{H}_\mathcal{R}$.

$$rmse(\mathcal{H}_\mathcal{G}, \mathcal{H}_\mathcal{R}) = \sqrt{\sum_i (\mathcal{H}^i_\mathcal{G} - \mathcal{H}^i_\mathcal{R})^2} \tag{6.20}$$

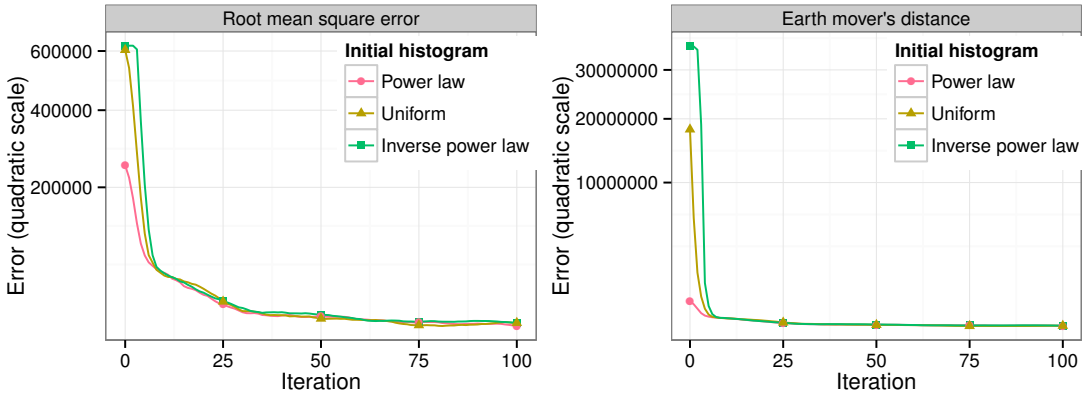


Figure 6.11: Root mean square error and earth mover's distance for up to 100 iterations on the CD dataset.

The *earth mover's distance* (EMD) calculates the number of records that is needed to transform one histogram to another by moving records to adjacent fields. Intuitively, the distance does not penalize slight misestimations as strongly as the RMSE. For example, if a cluster of size 237 is estimated instead of 238, the distance is at most $2 \cdot 237$, while RMSE would quadratically penalize both the present cluster at size 237 and the missing cluster at 238.

$$emd(\mathcal{H}_\mathcal{G}, \mathcal{H}_\mathcal{R}) = \sum_i emd_i(\mathcal{H}_\mathcal{G}, \mathcal{H}_\mathcal{R}) \tag{6.21}$$

$$emd_i(\mathcal{H}_\mathcal{G}, \mathcal{H}_\mathcal{R}) = \begin{cases} 0 & \text{if } i = 0 \\ i \cdot (\mathcal{H}^i_\mathcal{G} - \mathcal{H}^i_\mathcal{R}) & \\ +emd_{i-1}(\mathcal{H}_\mathcal{G}, \mathcal{H}_\mathcal{R}) & \text{else} \end{cases}$$

Figure 6.11 shows the mean errors for up to 100 iterations on the CD dataset. The root mean square error steadily approaches zero after an initial phase that adjusts the general form of the distribution. For a good initial estimate, the error immediately decreases, while the estimate needs up to ten iterations to correct worse initial estimate.

Similarly, the earth mover's distance reflects the correction of the general form for the inverse power law distribution. For the uniform distribution the distance decreases earlier, because the shifting of the elements towards smaller clusters immediately begins. The power law distribution exhibits a low EMD from the start, since only few records need to be adjusted.

If users are not confident in their initial estimate, ten iterations are needed to guarantee a good fit of the distribution – as seen in the EMD plots. The RMSE reveals that fine-grained estimates need up to 50 iterations on the CD dataset. For better initial estimates (e.g., uniform distribution), the general form is found already after five iterations. Fine-grained estimates, however, need almost the same number of iterations for good and poor initial histograms.

Dataset	Number of iterations	Runtime (in ms)	Reduction ratio
CD	10	83	99.987%
	25	190	99.917%
	50	396	99.667%
	100	1,045	98.667%
Customer	10	27	99.990%
	25	53	99.940%
	50	152	99.760%
	100	632	99.040%
Cora	10	797	94.767%
	20	1,104	79.043%
	30	1,666	52.828%

Table 6.3: Runtime and relative number of comparisons for a given number of iterations per dataset.

The RMSE and EMD curves of the difficult Cora dataset (not shown) exhibit a steep decrease of error similar to the EMD curve of the CD dataset. For the Customer dataset, only two kinds of errors are possible and, thus, both RMSE and EMD are much smoother and start to decrease to zero at the first iteration. Both curves (not shown) exhibit the same characteristics, because of the close relation of EMD and RMSE in this case.

6.5.7 Execution time

In the last experiment, we measure the runtime of the estimation. As before, we simulate the candidate comparison with a hash lookup in the gold standard. Hence, we measure the overhead of our random walk without performing any actual candidate comparisons. Further, we measure the *reduction ratio* [Bilenko et al., 2006] that is defined as the ratio of saved comparisons during estimation and the total number of all pairs:

$$reduction\_ratio = 1 - \frac{\binom{n}{2}}{\binom{N}{2}} = 1 - \frac{n(n-1)}{N(N-1)} \tag{6.22}$$

The single-threaded experiment ran on an i5 desktop PC with 16 GB RAM and Java 1.7. The times do not include the initial data loading phase, where all data is loaded to be held in-memory. Table 6.3 lists the runtime in milliseconds and the reduction ratio for the three datasets. For all configurations, the runtime of the random walk with fewer than two seconds is negligible even for high number of iterations.

In general and as expected, the runtime increases with the maximum cluster size and the number of records. For the CD dataset, the three slowest parts of the algorithm took 90% of the runtime, where 40% of the time is spent on the random sampling and calculation of $\mathcal{H}_{\mathbb{S}}$, 10% on the calculation of $T_{\mathbb{S}}$, and 40% on the Power iterations (inner loop of Algorithm 6.3). For later iterations, the calculation of $\mathcal{H}_{\mathbb{S}}$ and $T_{\mathbb{S}}$ take

relatively more time, because the sample becomes larger. In contrast, the number of Power iterations linearly decreases from 100 to 25. Nevertheless, the overall time per iteration increases linearly. Hence, performing rough estimations on small samples is relatively cheap.

The reduction of comparisons for the two larger datasets is enormous and solely depends on the number of iterations. As shown in the last section, coarse-grained estimation requires fewer than ten iterations and needs to perform only every 10,000th comparison. Of course, candidate selection techniques also save comparisons significantly, but as we show in Section 5.4, we can combine both techniques to further improve the efficiency and accuracy.

Finally, the performance can be easily improved. For the Cora dataset, the calculation of T_S dominates the overall time, but can be sped up with pre-calculated tables for the binomial coefficients similar to lookup tables for sinus. For a product, $\mathcal{H}_S$ should be incrementally calculated. Nonetheless, the potential of our method already becomes apparent: If one assumes that one comparison takes 20 microseconds (measured in previous experiments [Vogel et al., 2014]), a full naïve run needs half a year and a single-pass SNM with $w = 100$ about one hour, while a rough estimate in ten iterations take only 45 seconds, and a fine-grained estimate with near-perfect results in 50 iterations takes less than four minutes.

Summary In this chapter, we defined the problem of transitively unclosed duplicate declarations and developed two operators, `transitive closure` and `stable matching`, to produce transitively closed clusters for different data integration use cases. While these clustering operators are usually chosen by the `duplicate detection` and `record linkage` operator automatically, we also showed how Meteor users may invoke them manually.

Moreover, we presented the implementations of the operators, which rely on the workset iterations of Stratosphere. The implementations use the worksets to efficiently reduce the network load by transferring only those records that have may change the result of one iteration.

Lastly, we provided an advanced cardinality estimation technique that accurately and robustly estimates the number and sizes of duplicate clusters. We use these cardinality estimations to estimate the runtime of the operators. Together with some non-trivial reordering rules, these estimation techniques allow a seamless integration of the operator into the optimizer.

For data integration projects, clustering the duplicate pairs represents the last step before the fusion of the duplicates into concise records. In the next chapter, we introduce our final operator `fusion`, which addresses the merge of values.

7

Fusing Duplicate Clusters

The `fusion` operator merges duplicate clusters into integrated records. Primarily, it resolves value conflicts, which naturally occur in fuzzy duplicates. Additionally, it maintains referential integrity of related datasets by updating references on the fused records.

Further, we combine `duplicate detection`, `transitive closure`, and `fusion` into the `duplicate removal` operator. Data maintainers can use this operator to directly create a duplicate-free dataset. In particular, this operator adds the non-duplicates to the result of the `duplicate detection` and `fusion` operator to produce a complete and clean dataset.

This chapter first introduces a conflict resolution model tailored to semi-structured data and provides the Meteor syntax for both operators. It then proceeds with the implementation of the operators and integrates the operator into the optimizer.

7.1 Resolving conflicts in nested data structures

Data fusion denotes the process of merging several duplicate representations of the same entity into one concise representation by resolving *conflicts*.

Definition 7.1 (Conflict)**.** Let $\mathcal{A}$ be a domain and $C \subseteq \mathcal{A}$ a cluster of values. A *conflict* denotes two or more different values in the cluster: $\exists v_1, v_2 \in C : v_1 \neq v_2$.

A conflict can occur only in clusters with more than one value. Further, we assume for our semi-structured value model that a missing attribute value in a record does not contribute to a conflict. Nevertheless, *null* values ($\perp$) contribute to conflicts. For example, the values `[John, ⊥]` and `[John, Jon]` are conflicts, while `[John, John]` and `[John]` are *conflict-free*. The `fusion` operator addresses conflicts with *conflict resolution functions*. Similar to repair functions in the `scrub` operator, we define them as *best effort* functions, which may not resolve the conflict at once, but hopefully remove some less probable values.

Definition 7.2 (Conflict resolution functions)**.** Let $\mathcal{A}$ be a domain and $C \subseteq \mathcal{A}$ a cluster of values. A *conflict resolution function* f_{cr} is a function that processes (conflicting) values

and ideally returns a non-conflicting value: $f_{cr} \colon \mathcal{A}^n \to \mathcal{A}^n$. A *safe* conflict resolution function f_{cr}^{safe} always returns at most one value: $f_{cr}^{safe} \colon \mathcal{A}^n \to \mathcal{A} \cup \{\perp\}$.

To apply the conflict resolution functions, the `fusion` operator additionally needs the (nested) path to identify the portion of the data relevant to the resolution function. We call a pair of conflict resolution function and corresponding path a *conflict resolution*. The `fusion` operator applies conflict resolutions on (conflicting) input values until one value remains. If it uses a safe conflict resolution on the values, the resolution of the conflict on the path can be guaranteed. Otherwise, some values may remain in conflicting state. Before discussing how the `fusion` operator handles conflicting values in the end, we present the range of available conflict resolution functions.

7.1.1 Conflict resolution functions

For the available conflict resolution functions, we mostly adapt the wide range of available conflict resolution functions [Bleiholder and Naumann, 2006; Naumann and Häussler, 2002] to our settings. On the one hand, we leave out functions that rely on a fine-grain data quality model, require external knowledge bases, or introduce data dependencies between the duplicate clusters. On the other hand, we add new conflict resolution functions tailored to semi-structured data and extend the weighted majority voting with unidirectional evidence functions. In contrast to the previous work, we assume that any value is more relevant than *null* values. Therefore, all of our default conflict resolution functions ignore *null* values if non-*null* values exist. However, the user may explicitly request the conflict resolution functions to treat *null* as a regular value.

Table 7.1 shows the available conflict resolution functions and a corresponding Meteor example. The upper part of the table shows the **safe functions**, which always resolve conflicts. With *min/max*, the `fusion` operator chooses the minimum or maximum element from numbers, strings, or even composite values according to the order of Sopremo values. Moreover, the operator can calculate the *mean/sum* for numeric values. If any value suffices, the operator may pick a *random* value. The *first/last* value either corresponds to the first or second source for `record linkage` operators or is random for `duplicate detection` operators. The operator may also ignore any conflict by choosing a *constant* value or by retaining *all* (*distinct*) values. Lastly, the user may provide a *custom* Meteor function that returns the fused value. Currently, Meteor functions cannot signal further conflicts, so that these functions must guarantee safety.

In the middle part of the table, we list the **unsafe functions**. These functions are not guaranteed to resolve a conflict, but in many cases are among the most meaningful conflict resolutions. Taking the *longest* string value may retain most information, for instance for first names. Sometimes the *shortest* value is more appropriate. If two of more strings have the same maximum (or minimum) size, all of them are retained. For numeric values, picking the *median* value might result in one or two values depending if there are an odd or even number of elements. Further, a user may *prefer* one source over the others or choose the *mostFrequent* values. For dependant values, the user may select all values *corresponding* to already chosen value(s) in another attribute. Finally,

	Conflict resolution	Example in Meteor	Comment
Safe atomic	Min/Max	startYear: min	Chooses min/max value
	Mean/Sum	amount: sum	Aggregates values
	Random	famousQuote: random	Picks a random non-null value
	First/Last	lastName: first	Takes first/last non-null value
	Constant	zip: 'unknown'	Sets a constant value
	All	originalID: all	Retains all non-null values
	Distinct	employments: distinct	Retains all distinct values
	Custom	mergeAddr = **fn**(addrs){...} ... addresses: &mergeAddr	Invokes Meteor function that returns one value
Unsafe atomic	Shortest/Longest	firstName: longest	Picks shortest/longest value
	Median	age: median	Chooses middle value(s)
	Prefer source	middle: prefer('earmark')	Retains value(s) from source
	Most frequent	city: mostFrequent	Takes most frequent value(s)
	Corresponding	street: corresponding($p.city)	Picks the value(s) corresponding to another attribute
	Vote	isNick = **fn**(names){...} ... firstName: vote(&isNick)	Choose(s) values with highest belief

Structural	Merge objects	employment: {
		employer: mostFrequent,
		start: min
		}
			Recursive definition

	Merge array	funds: merge	Combines all arrays into one
	Merge distinct	positions: mergeDistinct	Merges all distinct elements
	Array grouping	addresses: resolveGroup($p.addresses.city, {...})	Groups elements and recursively fuses groups

Table 7.1: Conflict resolution functions

the *vote* function selects the values with the highest *belief* using the *Dempster-Shafer theory of confidence* as follows.

Dempster-Shafer theory of confidence

This section is based on our journal article "Integrating open government data with stratosphere for more transparency" [Heise and Naumann, 2012]. With the Dempster-Shafer theory of confidence [Shafer, 1976], we address the question which value v_i of a set of conflicting values $\mathcal{V}$ we believe most in a *vote* conflict resolution. Intuitively, some values may boost our belief in other values, while others are pure contradictions. We capture such boosting relationships with *evidence functions*.

We start with some initial, user-provided *beliefs* in the respective sources. Through evidence functions, a value may share its initial belief with other values to boost them. Then, we combine the respective beliefs of each value to a global belief. Finally, we choose the value with the highest belief. For example, given the three possible values `[J., John, Bill]` with user-provided weights `[.8, .5, .8]` and an evidence function where abbreviated words boost possible original words, the *vote* function selects `John` despite

the low initial weight. In the remainder of this section, we formally define the concepts that are necessary to fully understand the example. First, we define the asymmetric evidence function.

Definition 7.3 (Evidence function)**.** Let $\mathcal{V}$ be a set of values. An *evidence function* e checks whether the first value supports the second value: $e\colon \mathcal{V} \times \mathcal{V} \to \{true, false\}$.

Evidence functions let values boost other values. For a given $v_i \in \mathcal{V}$, we denote with E_i the set of *supported values*: $E_i = \{v_j \in \mathcal{V} \mid e(v_i, v_j)\}$. With the evidence function, we distribute the beliefs in certain values, which are represented through belief functions.

Definition 7.4 (Belief function)**.** Let $\mathcal{V}$ be a set of values. A *belief function* is a function that assigns a belief mass $[0, 1]$ to all value combinations $m_v : 2^{\mathcal{V}} \to [0, 1]$ with $m_v(\varnothing) = 0$ and $\sum_{V \in 2^{\mathcal{V}}} m_v(V) = 1$.

Although the Dempster-Shafer theory of confidence can model the joint belief in value combinations, we assume that the user wants only one value as the final result of the conflict resolution. Therefore, we distribute the initial belief mass to the *supported values* E_i. If the user assigns a weight of less than 1, we also add some mass to the set of all values $\mathcal{V}$ indicating that any other value could also be correct. The mass function for the value v_i with respective user-specified weight w_i is:

$$m_i(V) = \begin{cases} \frac{w_i}{|E_i|} & \text{if } V = \{v_j\} \wedge v_j \in E_i \\ 1 - w_i & \text{if } V = \mathcal{V} \\ 0 & \text{else} \end{cases}$$

The initial mass function $m_i(V)$ distinguishes three cases. First, single values supported by v_i receive a fair share of the mass. Second, the remaining mass is evenly distributed over the set of all values. Third, all other value combinations and unsupported values receive 0 mass. It is relatively easy to see that this function is indeed a mass function, where the empty set receives 0 mass and the sum of all masses equals 1. Because the evidence function is asymmetric and individual weights are used, one value v_1 may boost the belief in another value v_2, while v_2 adds less or no belief mass to v_1. To decide on a specific value, the *vote* function combines the individual mass beliefs of each value with the *Dempster's rule of combination* to an overall belief function.

$$(m_1 \oplus m_2)(V_a) = \frac{1}{1 - K} \sum_{V_b \cap V_c = V_a} m_1(V_b) m_2(V_c), \text{with } V_a \neq \emptyset$$

$$K = \sum_{V_b \cap V_c = \emptyset} m_1(V_b) m_2(V_c)$$

In the nominator, the rule collects shared belief for a value combination V_a. Because we do not allow multi-valued results, $V_b \cap V_c = V_a$ is satisfied when both mass functions m_1 and m_2 directly support one value $V_b = V_a = V_c = \{a\}$ or one mass functions supports all values $V_b = V_a = \{a\}, V_c = \mathcal{V}$ (initial weight below 1). The denominator normalizes small mass sums for conflicting values. In our case, all different values directly contribute to K. With the rule of combination, the *vote* function can successively merge the belief

function until it receives a joint belief function over all values. It then picks the value(s) with the highest belief mass.

Reconsidering the example in the beginning, we have three possible values `[J., John, Bill]` constituting V. The initial belief mass functions m_1 for `J.`, m_2 for `John`, and m_3 for `Bill` are:

$$\begin{aligned} m_1(J.) &= \frac{0.8}{2} = 0.4 \qquad m_2(John) = 0.5 \qquad m_3(Bill) = 0.8 \\ m_1(John) &= \frac{0.8}{2} = 0.4 \\ m_1(\mathcal{V}) &= 0.2 \qquad m_2(\mathcal{V}) = 0.5 \qquad m_3(\mathcal{V}) = 0.2 \end{aligned}$$

All other mass assignments are 0. The combination of the first two masses m_{12} yields

$$\begin{aligned} K =& m_1(J.) \cdot m_2(John) = 0.2 \\ m_{12}(J.) =& \frac{1}{1-K} \cdot (m_1(J.) \cdot m_2(\mathcal{V})) = 0.25 \\ m_{12}(John) =& \frac{1}{1-K} \cdot (m_1(John) \cdot m_2(John) + \\ & m_1(John) \cdot m_2(\mathcal{V}) + m_1(\mathcal{V}) \cdot m_2(John)) = 0.625 \\ m_{12}(\mathcal{V}) =& \frac{1}{1-K} \cdot (m_1(\mathcal{V}) \cdot m_2(\mathcal{V})) = 0.125 \end{aligned}$$

With K, we normalize over the conflicting values, which are $m_1(J.)$ and $m_2(John)$. Other conflicts cannot occur, because m_1 has only additional mass for `John` or $\mathcal{V}$ and m_2 only for $\mathcal{V}$. Further combination of the belief function with m_3 results in

$$\begin{aligned} K &= m_{12}(J.) \cdot m_3(Bill) + m_{12}(John) \cdot m_3(Bill) = 0.3 \\ m_{123}(J.) &= \frac{1}{1-K} \cdot (m_{12}(J.) \cdot m_3(\mathcal{V})) = \frac{2}{12} \\ m_{123}(John) &= \frac{1}{1-K} \cdot (m_{12}(John) \cdot m_3(\mathcal{V})) = \frac{5}{12} \\ m_{123}(Bill) &= \frac{1}{1-K} \cdot (m_{12}(\mathcal{V}) \cdot m_3(Bill)) = \frac{4}{12} \\ m_{123}(\mathcal{V}) &= \frac{1}{1-K} \cdot (m_{12}(\mathcal{V}) \cdot m_3(\mathcal{V})) = \frac{1}{12} \end{aligned}$$

The conflicts are $m_{12}(J.), m_3(Bill)$ and $m_{12}(John), m_3(Bill)$, while all masses m_{123} origin from the same standard case $m_{12}(X), m_3(\mathcal{V})$ and vice versa. Although we initially had little belief in `John`, the final result favors this approach, because of the propagated evidence.

Structural conflict resolution function

The previous conflict resolution functions are primarily used for atomic values, although most of them can be also applied to conflicting values of arrays or objects. In most cases,

the user needs a more fine-grain control over nested data, for example to use separate conflict resolution functions on different attributes of an object.

For **objects**, we provide the conflict resolution function *merge objects* that allows users to specify individual conflict resolution functions for each attribute. When the `fusion` operator processes a (conflicting) set of records with the nested function, the values of each attribute are collected into an array and the individual conflict resolution functions are recursively applied. The Meteor syntax reuses the object notation, such that Meteor users should be able to easily define *nested* conflict resolution functions.

If the resulting values are **arrays**, we provide three functions. First, the *merge* function concatenates conflicting arrays to one array. Second, *mergeDistinct* merges the arrays and then removes all duplicate elements, even if they originate from the same array. Third, *resolveGroup* subdivides the conflicting values into groups with a grouping key and recursively applies another conflict resolution function on the groups. The resulting array contains the resolved values from each group.

Execution semantics of the fusion operator

The `fusion` operator ingests a set of disjoint clusters and fuses all records within a cluster to one consistent record with a list of *conflict resolution rules*. Each rule consists of the path to the value and a conflict resolution function. To ensure scalability, all conflict resolution rules operate on individual cluster and thus each cluster can be processed in parallel.

Because unsafe conflict resolution functions may leave a record in partial conflict, the operator needs to choose one of two strategies to handle incomplete resolutions. First, it may remove such records and return them to the user for manual inspections. Second, it may ignore incomplete resolutions and return the conflicting records with alternative values. Therefore, we designed the `fusion` operator to produce an optional second output. If this output is connected, the first behavior is invoked, else all records are retained with potential conflicts.

Lastly, a major design goal of the data cleansing and integration operators is to support the user to easily and declaratively maintain referential integrity in the integrated datasets. To that end, the `fusion` operator allows users to specify the paths to foreign keys that refer to the entities that are fused. After fusing the entities, the operator automatically updates all foreign keys referring to the original unfused entities to new fused entities.

7.2 Declarative conflict resolution in Meteor

To invoke the `fusion` operator, Meteor users need to specify a (nested) conflict resolution. If they use the *vote* function, they may further define source- and attribute-specific weights, which are used as initial belief masses for the values. Further, users may provide paths describing foreign keys in other datasets referring to the dataset under fusion. Listing 7.1 shows the syntax for the `fusion` operator in Meteor.

```
1  $persons, $unfused = fuse $personClusters
2    with resolution {
3      originalID: all,
4      lastName: mostFrequent,
5      firstName: [vote(&isNick, &isAbbr), longest, min],
6      middleName: [withNull, mostFrequent, withoutNull, min],
7      birth: min,
8      death: max,
9      relatives: merge,
10     addresses: mergeDistinct,
11   }
12   with weights {
13     freebase: .7,
14     uscongress: .9 * {
15       firstName: .8
16     }
17   }
18   update foreign key $persons.relatives[*].person;
```

Listing 7.1: Fusing duplicate clusters in Meteor.

The operator invocation "**fuse**" accepts one input dataset and returns one or two outputs. The optional second output can be used to separate incompletely fused from completely fused records. Users may review the incomplete records and possibly review their definition of the conflict resolution. We assume that users primarily rely on the second output to debug and refine their configuration to eventually use only safe resolution functions.

In the next line, the script specifies one nested resolution function for the `fusion` operator with the keywords "*with resolution*". Further, the script configures individual resolution function for eight attributes (in a similar fashion as scrubbing rules, see Section 3.2). Internally, the operator flattens nested conflict resolution functions to obtain a list of conflict resolution rules consisting of the absolute path and the corresponding conflict resolution functions.

The first sub-resolution function "originalID: all" describes that all original IDs should be retained in an array. In contrast, "lastName: mostFrequent" defines that only the most frequent value(s) should be retained. Note that for such unsafe functions, unless a user explicitly specifies the optional second output to review incomplete fusions, Meteor issues a warning to the user and provides basic hints.

The next resolution specification with the array syntax defines a *cascade* of three resolution functions for the first name. First, the "vote" function is applied to the first name with two evidence functions. Because the "vote" is an unsafe function, more than one value may remain. Therefore, the "longest" value of the remaining values is chosen in a second step. However, that still may result in an incomplete resolution. Therefore, as a last resolution, the lexicographic minimum value is chosen, which guarantees a safe conflict resolution. The last four conflict resolutions are all safe: They select the minimum birth year, the maximum death year, and merge the arrays containing the relatives and addresses.

In the line "middleName: [withNull, mostFrequent, withoutNull, min]", the user explicitly requests *null* values to be considered in the *most frequent* function. In case of ties, the *min* is chosen without *null*. All functions use the *withoutNull* configuration by default. The *withNull* configuration can be used only within a cascade and is automatically reset at the end of the cascade.

For the "vote" function, users may supply evidence functions and initial weights. If no evidence functions are supplied, every value supports only equal values. Else, Meteor combines the evidence functions with a Boolean conjunction to one evidence function. Users may specify the weights in a nested fashion similar to the nested conflict resolution using the keywords "*with weights*". For each source, the user provides an initial weight and then further refines the weights for individual attributes. In this example, the user distrusts the first name of the congress in particular (e.g., it contains many nick names), and therefore assigns a sub-weight of 0.8. The syntax with the multiplication sign already reveals that the sub-weights are multiplied with the parent weights. Thus, the initial belief mass of a first name from congress is 0.72.

The last statement "*update foreign key*" provides foreign keys referring to the dataset under fusion. More than one foreign key can be defined with an array notation. In this example, the foreign key "$persons. relatives [∗]. person" refers to the resulting dataset, which indicates a cyclic reference. Further, the `fusion` operator infers from the array projection that relatives are an array of objects, of which the person attributes should to be updated. In accordance to the `data map` operator, we assume that the foreign key always refers to the `id` of the current record.

Duplicate removal

The second operator `duplicate removal` combines `duplicate detection` and `fusion` and additionally transfers non-duplicates to the result. The `duplicate removal` operator always performs the `transitive closure` operator as the clustering in the `duplicate detection` sub-operator. Accordingly, the operator provides the properties of the two operators except for the cluster parameter as shown in Listing 7.2. This operator is the Swiss knife of data maintainers to remove the duplicates and create a consistent dataset in one operator invocation. It corresponds to the fusion operators of HumMer [Bilke et al., 2005; Bleiholder and Naumann, 2005] and Fusionplex [Motro and Anokhin, 2006].

```
1 $cleanPersons = remove duplicates $dirtyPersons
2   with ...// sim expression
3   sort/partition on ...// blocking key
4   with resolution ...
5   with weights ...
6   update fk ...;
```

Listing 7.2: Properties of the `duplicate removal` operator in Meteor.

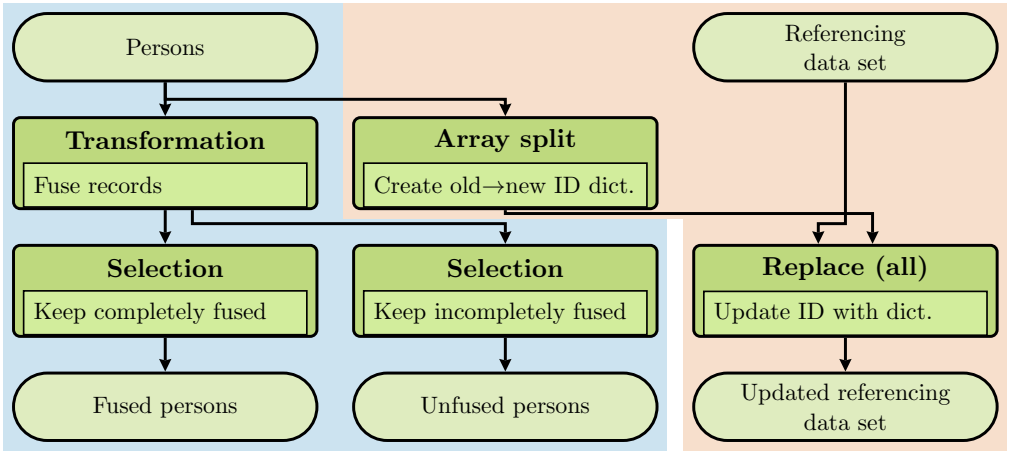


Figure 7.1: The `fusion` operator in Sopremo. Left hand side fuses the records and right hand side updates referencing datasets (one `replace` per referencing dataset).

7.3 Implementation in Stratosphere

In this section, we first discuss the implementation of the `fusion` operator in Sopremo and then provide the implementation of the `duplicate removal` operator. The core activity of the `fusion` operator can be easily performed with a `transformation` operator, because each duplicate cluster can be independently transformed into a fused record. If the user wants to separate fused and partially fused records, two `selection`s need to distinguish the cases in the output of the `transformation` operator. Internally, unfused values are stored in a special array that additionally provides provenance and weight information to the conflict resolution functions. Hence, the `selection`s recursively search for such special arrays in the records and either retain or remove such records. Figure 7.1 visualizes the implementation of the `fusion` operator as a Sopremo plan.

In addition to the core activity, the `fusion` operator also updates foreign keys referring to the fused entities. It first builds a dictionary, where each entry contains the old and the new ID of the records, and second applies a `replace` or `replace all` operator on the referencing dataset. The dictionary can be created relatively straight-forward from the input clusters, by using the conflict resolution function on the ID field (implicitly "min") to determine the new ID and projecting the old ID from the records. Depending on the path specification, the `fusion` operator then invokes a `replace` for non-arrays or a `replace all` for arrays. Note that the `replace` operator basically performs an outer-join to replace the values.

In the running example (see Listing 7.1), the fused dataset referred to itself. Nevertheless, even for this special case, the given method correctly updates the output of the `fusion` operator, so that the plan looks slightly different (`replace` directly after `selection` of the fused elements).

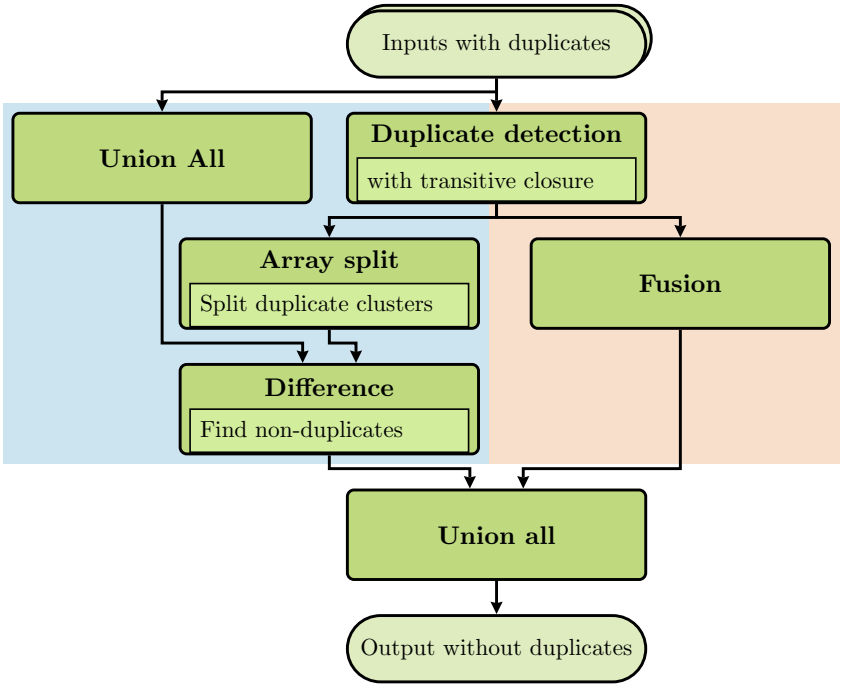


Figure 7.2: Implementation of the `duplicate removal` operator. The left side transfers all non-duplicates, while the right side fuses the duplicates.

Duplicate removal

The `duplicate removal` operator completely forwards its configuration to the inner `duplicate detection` and `fusion` operator as shown in Figure 7.2. On the right hand side of the plan, the detected duplicates are fused and send to the output. On the left hand side, the plan finds all non-duplicates by calculating the difference of the input datasets and all records in duplicate clusters.

7.4 Optimization of the Fusion operator

Because the `fusion` operator translates to base operator, we can directly apply reordering rules for the base operators to the parts. Similarly, the `duplicate removal` operator consists of sub-operators, of which we already derived several reordering rules. Nevertheless, the `fusion` and `duplicate removal` operator exhibit additional, interesting properties useful for optimizations. In this section, we first review the operator taxonomy and then deduce a broad set of reordering rules. We conclude with a brief description of the cost and cardinality estimation. Note that we focus on the core activity of the `fusion`; the update of the foreign keys from other datasets with the `replace` operator is an outer-join with several rewrite rules in the base package.

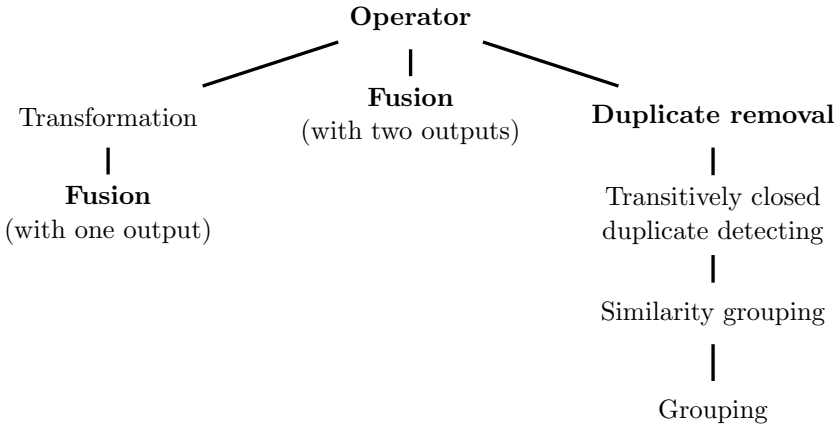


Figure 7.3: Operator taxonomy for the `fusion` and `duplicate removal` operators.

7.4.1 Operator taxonomy

The `fusion` operator ingests one dataset of clusters and fuses each cluster at a time. Unless the operator has unsafe conflict resolution functions and the user wants to examine incompletely fused records, the operator is a specialized `transformation` operator. Otherwise, it performs transformation and selections at the same time and can therefore not be expressed as one relational operator. Figure 7.3 depicts the operator taxonomy in Presto [Rheinländer et al., 2013] for the `fusion` operator.

The `duplicate removal` operator partitions the dataset into disjoint clusters and fuses them. When using the *all* conflict resolution, it performs a transitively closed `duplicate detection`, which in turn is a generalized (similarity) grouping. In contrast to the `duplicate detection` operator, we receive a complete partitioning and not only clusters with two and more records. Nevertheless, as discussed in Section 6.4, the completeness of a partitioning does not change the rewrite rules, as long as groups are either completely present or missing.

With one or two outputs, the `fusion` operator is **associative**: We can split or merge `fusion` operators by dividing or appending the list of rules respectively. Further, the operator changes the schema by fusing an array of records into one record. The `duplicate removal` operator is **commutative**, because all inputs are bag-unified. Further, the operator with safe configuration **keeps the schema** and is in some cases **idempotent**, which we discuss in the following among other reordering rules.

7.4.2 Reordering rules

We first review optimization opportunities of the `fusion` operator before developing holistic optimization rules for the `duplicate removal` operator. As an important consideration, we start by splitting and reordering conflict resolution functions.

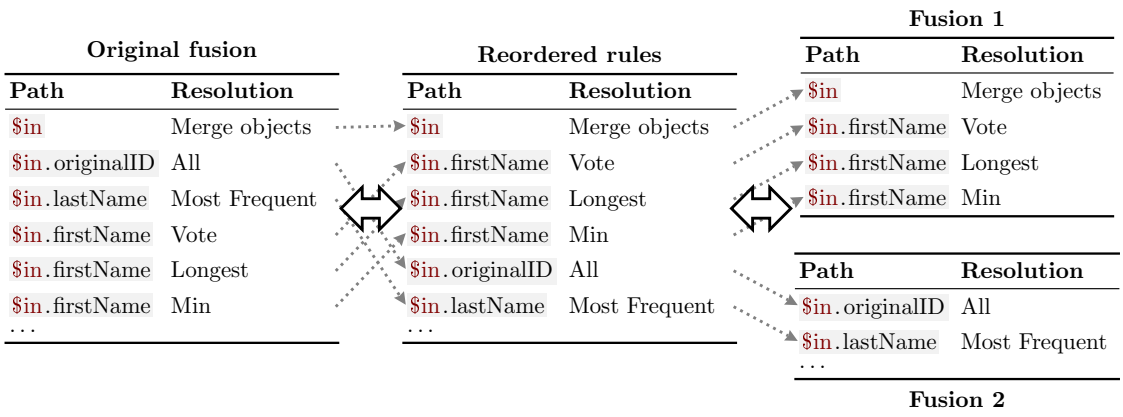


Figure 7.4: Reordering conflict resolution functions and splitting fusion operators. Left side shows the original list of resolution functions (extracted from Listing 7.1). Middle depicts reordering of rules. Right side splits the `fusion` operator into two.

Reordering conflict resolution functions

Similar to the `scrub` operator (see Section 3.4), we internally store conflict resolution functions with their absolute path in a list, which is sequentially processed by a special transformation expression. Therefore, we can split or merge any `fusion` operator.

Figure 7.4 demonstrates reordering of rules and splitting of a `fusion` operator. On the left-hand side, we visualized the rule list of the operator from the `fusion` Meteor script (see Listing 7.1). First, the conflicting objects are merged by collecting the values of all attributes in arrays. Then, the nested resolution functions are executed on the original ID, last name, and first name. In the middle, we push the conflict resolution functions for the first name before the other conflict resolution functions. On the right-hand side, we split the `fusion` operator into two. As with any `transformation` operator, such a reordering is only possible if no read-write conflict occurs. In particular, two conflict resolution functions cannot be reordered:

1. if both conflict resolution functions are applied to the same path,
2. if one path is a sub-path of the other, or
3. if a conflict resolution function depends on the resolution of the other (currently only *corresponding*).

In our example, we cannot reorder the cascading conflict resolutions on the first, because they would change the result (first case). Further, the objects must always be merged before the other resolution functions can be applied (second case).

Rules with the fusion operator

Because the `fusion` operator with one output is a `transformation` operator τ, we can apply those reordering rules to the `fusion` operator. In the following, we provide rules for a `fusion` operator fus with conflict resolution functions cr_i on datasets $\mathcal{C}_i$.

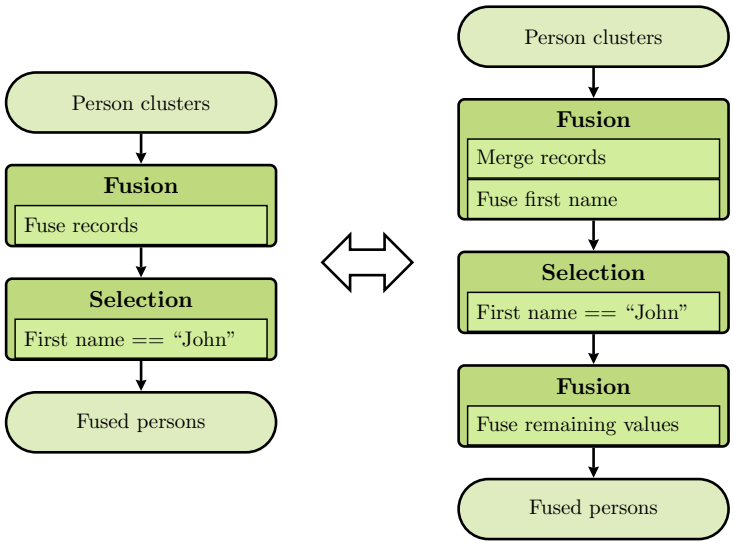


Figure 7.5: Pushing a selection partially through the `fusion` operator (see rule reordering in Figure 7.4).

$$fus_{cr_1,cr_2}(\mathcal{C}) \equiv fus_{cr_2}(fus_{cr_1}(\mathcal{C})) \quad \text{(Rule I)}$$

$$fus_{cr_1,cr_2}(\mathcal{C}) \equiv fus_{cr_2,cr_1}(\mathcal{C}), \text{ if } cr_1 \text{ independent of } cr_2 \quad \text{(Rule II)}$$

$$fus_{cr_1}(fus_{cr_2}(\mathcal{C})) \equiv fus_{cr_2}(fus_{cr_1}(\mathcal{C})), \text{ if } cr_1 \text{ independent of } cr_2 \quad \text{(Rule III)}$$

$$\sigma_\theta(fus_{cr}(\mathcal{C})) \equiv fus_{cr}(\sigma_\theta(\mathcal{C})), \text{ if } cr \text{ independent of } \theta \quad \text{(Rule IV)}$$

$$\pi_\alpha(fus_{cr}(\mathcal{C})) \equiv \pi_\alpha(\mathcal{C}), \text{ if } \alpha \text{ not a sub-path of path in } cr \quad \text{(Rule V)}$$

$$\tau_e(fus_{cr}(\mathcal{C})) \equiv fus_{cr}(\tau_e(\mathcal{C})), \text{ if no read-write conflicts in } e, cr \quad \text{(Rule VI)}$$

$$fus_{cr}(\mathcal{C}_1 \bowtie_\alpha \mathcal{C}_2) \equiv fus_{cr}(\mathcal{C}_1) \bowtie_\alpha \mathcal{C}_2, \text{ if } cr \text{ independent of } \alpha \quad \text{(Rule VII)}$$

The first three rules show the algebraic properties **associativity** and the **commutativity** of rules and operators. The remaining rules can be inferred from the operator taxonomy and the `transformation` rules. Moreover, we can apply the rules to a `fusion` operator with two outputs, which does not inherit the rules directly from the operator taxonomy. Note that in this case, preceding operators can be pushed through the `fusion` operator only if two equivalent operators are applied to both outputs. However, we could also push the operator if applied only to the primary output in *fuzzy optimization*, because we assume that the second output is primarily used for debugging and a pushed selection would result in fewer but more relevant records regarding the user's query intent.

Although the reordering rules appear powerful in the beginning, a close observation reveals that only in special cases operators can be pushed through the entire `fusion` operator. The first conflict resolution function is an *object merge* in most cases, which results in a read-write conflict in all rules. Therefore, the previous rules usually result in partial rewrites as shown in Figure 7.5. We can split the `fusion` operator and reorder the rules, such that only few rules need to be performed before the selection. Consequently, most of the rules can be performed on fewer records.

Advanced rules with rewritten conditions

To push a selective operator entirely through the `fusion` operator towards the data sources, the optimizer needs rules that address cases of read-write conflicts. In these cases, the selection condition must be adjusted accordingly. For a conflict resolution function cr, the *merge object* resolution function mo, a Boolean condition $\phi(\alpha)$ on the attribute α, and a dataset $\mathcal{C}$ of record clusters, we can perform the following reorderings.

$$\sigma_{\phi(\alpha)}(fus_{cr}(\mathcal{C})) \equiv fus_{cr}(\sigma_{\phi(cr(\alpha))}(\mathcal{C})), \text{ if } cr \text{ fuses } \alpha \qquad \text{(Rule VIII)}$$

$$\sigma_{\phi(\alpha)}(fus_{mo}(\mathcal{C})) \equiv fus_{mo}(\sigma_{\phi(\alpha*)}(\mathcal{C})) \qquad \text{(Rule IX)}$$
$$\text{with } \forall C \in \mathcal{C}\colon \alpha* = \{r.\alpha | r \in C\}$$

In the first rule, we push a `selection` through the `fusion` operator, where the conflict resolution function fuses the attribute, over which the selection condition is defined. Therefore, we must change the selection condition, to first perform the conflict resolution before checking the original condition. Because the values have been fused only temporarily for the condition, they still must be permanently fused after the selection. Note that the modified condition $\phi(cr(\alpha))$ is actually another Boolean condition over α. Therefore, the selection can be further pushed through another `fusion` operator with a rule on the same attribute.

The second rule allows the optimizer to push a selection with a condition ϕ over α through the *merge object* resolution function. Among other attributes, *merge object* specifically collects all values in α without performing an actual conflict resolution on α. Consequently, ϕ is defined over an array of alternative values (e.g., by the first rule). Thus, to push the selection through *merge object*, we first must create the set $\alpha*$ containing the values of each record in α, before the Boolean expression ϕ can be evaluated on $\alpha*$. In Sopremo, we can directly use the array projection expression *$input[*].attribute*. The rules for the `join` operator are analogous.

Figure 7.6 summarizes the reordering rules for our running example. We can push the `selection` completely through the `fusion` operator by pushing it through each conflict resolution rule. If the rule implies a read-write conflict, we rewrite the selection condition. Otherwise, we can simply push the `selection` through the rule. The final selection condition is `min(longest(vote($input[*].firstName))) == "John"` for the running example.

7.4.3 Rules with duplicate removal operator

While the `fusion` operator exhibits good optimization opportunities, the previous `clustering` and `duplicate detection` operator provide only limited rules. However, a `selection` that is pushed through these operators probably improves the runtime much more than pushing a `selection` through the `fusion` operator. Therefore, in the following, we review optimization rules for the `duplicate removal` operator.

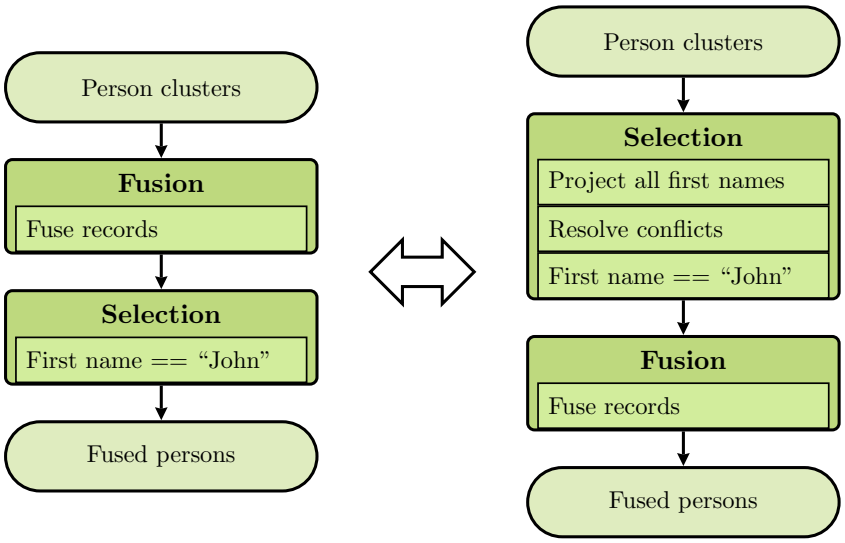


Figure 7.6: Pushing a selection completely through the `fusion` operator (see rule reordering in Figure 7.4).

Eager and lazy optimization

To push operators through the entire `duplicate removal` operator, we need to resort again to eager and lazy optimization ([Yan and Larson, 1995], discussed in Section 3.4). We split preceding operators into an eager part that can be performed before the `duplicate removal` operator and a lazy part that needs to be executed afterwards. For a `selection` with a condition θ^+ and a given relation $\mathcal{R}$ consisting of clusters, we can always find an eager condition θ^+ for the following rules (analogous to Section 6.4):

$$\sigma_\theta(dr_{naive,sim,cr}(\mathcal{R})) \equiv \quad \sigma_\theta(dr_{naive,sim,cr}(\sigma_{\theta^+}(\mathcal{R}))), \qquad \text{(Rule X)}$$

$$\text{with } \theta^+ \colon \theta(r_{clean}) \Rightarrow \quad \theta^+(r_{dirty}), \forall r_{dirty} \in C \subseteq \mathcal{R} \wedge dr_{cs,sim,cr}(C) = \{r_{clean}\}$$

$$\sigma_\theta(dr_{blocking,sim,cr}(\mathcal{R})) \equiv \quad \sigma_\theta(dr_{blocking,sim,cr}(\sigma_{\theta^+}(\mathcal{R}))), \qquad \text{(Rule XI)}$$

$$\sigma_\theta(dr_{snm,sim,cr}(\mathcal{R})) \Rrightarrow^+ dr_{snm,sim,cr}(\sigma_{\theta^+}(\mathcal{R})) \qquad \text{(Rule XII)}$$

$$\sigma_\theta(dr_{snm,sim,cr}(\mathcal{R})) \Rrightarrow^- dr_{snm,sim,cr}(\sigma_{\theta^+}(\mathcal{R})) \qquad \text{(Rule XIII)}$$

$$\sigma_\theta(dr_{snm,sim,cr}(\mathcal{R})) \equiv \quad dr_{snm,\theta^+ \wedge sim,cr}(\mathcal{R}) \qquad \text{(Rule XIV)}$$

The eager condition θ^+ must retain all records that lead to records selected by θ, but it may select additional records. For the SNM, a pre-selection of records changes the windows and lead to additional comparisons. Therefore, we can accept the additional comparisons, which can result in large duplicate clusters; we can adjust the window size, which may result in smaller and larger clusters; or we incorporate it in the similarity selection for an exact, but less powerful reordering.

Finding such an eager condition θ^+ is non-trivial and strongly depends on the conflict resolution functions and θ (similar to the `clustering` operator, see Section 6.4). In fact, we can use equivalent rules to find eager conditions on the `duplicate removal` operator.

Function	θ	θ^+
Min	a) $v = k$; b) $v > k$	a) $v \geq k \vee v = \perp$; b) $v > k \vee v = \perp$
Max	a) $v = k$; b) $v < k$	a) $v \leq k \vee v = \perp$; b) $v < k \vee v = \perp$
Shortest	a) $len_v = k$; b) $len_v > k$	a) $len_v \geq k \vee v = \perp$; b) $len_v > k \vee v = \perp$
Longest	a) $len_v = k$; b) $len_v < k$	a) $len_v \leq k \vee v = \perp$; b) $len_v < k \vee v = \perp$
Prefer source s	θ	$source \neq s \vee \theta$

Table 7.2: Eager Boolean conditions θ^+ for specific conflict resolution functions and a Boolean condition θ over a value v and constant k.

For a given similarity expression, we can review the similarity rules and exploit the following cases.

Negative rules with 0 distances A negative rule might enforce equality of certain attributes, for example the same birth year. When formulating a Boolean condition θ over this attribute, we can set $\theta^+ = \theta$, because fused records that satisfy θ can result only from records that satisfy θ as well.

Limited domains If the domain is limited, we may relax the condition to include all values that may contribute to the final records. For example, for a gender attribute, some records may have typographic errors. Therefore, θ^+ must also cover all transitively reachable typographic variations of selected values.

Limited cluster size For *fuzzy optimization*, we can estimate the maximum cluster size $size_{max}$ and relax the selection criterion according to it. A distance d for a given negative rule, need to be adjusted to $d \cdot (size_{max} - 1)$.

In addition to the similarity measure, we can also infer eager conditions for some conflict resolution functions. Table 7.2 lists the eager conditions for a `selection` with θ. For a *min* resolution function, we know that all values in the cluster have to be large than the selected value or *null*. Therefore, when selecting a fused value larger than a given threshold, we can infer that also all individual records must be larger than the threshold. Similar rules can be inferred for *max*, *shortest*, and *longest*. Further, the *prefer source* resolution function allows us to infer information about records originating from that source. For records of the preferred source, we can create a specific eager condition with the previous rules. Records from other source must always be selected by the eager condition, because we do not know any constraints on them.

Example 7.1. Consider `duplicate removal` on politicians and resolving the death year with *max*. If we are interested only in politicians that died in the last century, then we can remove all politicians who died in this century even prior to deduplication.

Rules with joins

We can transfer the idea of eager and lazy optimization to `joins` with `duplicate removal` operators. For brevity, we do not distinguish between exact and fuzzy optimization rules in the equations and discuss variations only textually.

$$dr_{cs,sim,cr}(\mathcal{R}_1) \bowtie_\theta \mathcal{R}_2 \equiv dr_{cs,sim,cr}(\mathcal{R}_1 \ltimes_{\theta^+} \mathcal{R}_2) \bowtie_\theta \mathcal{R}_2, \quad \text{(Rule XV)}$$

$$dr_{cs,sim_1,cr}(\mathcal{R}_1 \bowtie_\theta \mathcal{R}_2) \equiv dr_{cs,sim_1,cr}(dr_{cs,sim_2,cr}(\mathcal{R}_1) \bowtie_\theta \mathcal{R}_2), \quad \text{(Rule XVI)}$$

$$\text{with } \forall r_1, r_2 \in \mathcal{R}_1 \colon \forall j_1, j_2 \in \{r_1, r_2\} \bowtie_\theta \mathcal{R}_2 \colon sim_2(r_1, r_2) \Rightarrow sim_1(j_1, j_2)$$

Rule XV directly translates the idea of fuzzy selections to joins. We push a `semi-join` before the `duplicate removal` operator to reduce the number of records that need to be deduplicated. For highly selective joins, this rule significantly reduces the runtime of the operator. For example, if we join statistics over birth years with the politicians, we would remove all politicians, for whose birth years no statistics exist. However, if the conflict resolution changes the join key, fewer records would be selected with the original join condition θ and the result would change. In that case, the rewrite rule can adjust the eager condition as described before. Alternatively, we use the original condition for fuzzy optimization. For SNM, the comparison window changes with fewer input records and thus additional duplicates may be detected.

In Rule XVI, we assume that the inner join is rather unselective: The first relation $\mathcal{R}_1$ contains many duplicates that have multiple join partners in $\mathcal{R}_2$. We pre-clean the first dataset with an eager `duplicate removal` operator before joining it with the second relation. However, the similarity expression sim_2 can use only attributes from the first relation $\mathcal{R}_1$ and is more coarse-grain; that is, it finds fewer duplicates than the overall similarity expressions sim_1. Therefore, we apply a second, lazy `duplicate removal` operator with the original configuration over the join result.

Example 7.2. Consider a dirty politician relation with an additional address relation, where each politician may have several addresses. We can first cautiously deduplicate politicians by their names before joining the addresses. Consequently, the lazy `duplicate removal` operator deduplicates fewer records.

Rule XVI is exact if no additional information is discarded; that is, the eager `duplicate removal` operator retains all information with associative conflict resolution functions (e.g., min, sum, random, all, merge) and the similarity measure processes multi-valued attributes. For other cases, the rule is fuzzy and the result may change.

We described the rules for traditional `join` operators only. However, the rules may be extended to incorporate similarity joins and `record linkage` without clustering. Then, inferring the eager condition becomes even more challenging, so that the rules are more relevant to fuzzy optimization. Nevertheless, it is noteworthy, that the rules also apply to mixed query with `record linkage` and (similarity) `grouping` operators due to the operator inheritance.

Rules with partitioning operators

There exist additional reordering rules with partitioning operators (e.g., (similarity) `grouping`, `duplicate removal`) as follows.

$$dr_{cs,sim_1,cr}(dr_{cs,sim_2,cr}(\mathcal{R}))) \equiv dr_{cs,sim_1,cr}(\mathcal{R}) \quad \text{(Rule XVII)}$$

$$\text{with } \forall r_1, r_2 \in \mathcal{R}_1 \colon \forall j_1, j_2 \in \{r_1, r_2\} \bowtie_\theta \mathcal{R}_2 \colon sim_1(r_1, r_2) \Rightarrow sim(j_1, j_2) \quad \text{(Rule XVIII)}$$

If the inner similarity expression sim_2 is more coarse-grain and finds fewer duplicates than the outer expression sim_1, we can remove the inner `duplicate removal` operator. This reordering is exact only in the cases that we discussed for previous rules: The blocking key must be retained and no additional information relevant to the similarity measure must be discarded. For SNM, the rule is always fuzzy as well as for blocking if the key is changed. The rule can also be applied to combinations with (similarity) `grouping` operators. Additionally, for $sim_1 = sim_2$, this rule expresses **idempotence** of the `duplicate removal` operator.

7.4.4 Cardinality and cost estimation

The cardinality estimation of the `fusion` operator heavily depends on the estimation of the number of clusters of the `clustering` operator. If incompletely fused records are not separated for manual inspection, the number of records after fusion is equal to the number of clusters. Otherwise, we can apply selectivity estimation techniques to determine the ratio of incomplete records in a similar fashion to the `scrub` operator (see Section 3.4).

To estimate the cost of the `fusion` operator, we could exploit the following observations. For the core activity of the operator, we know that no network traffic occurs, because only `map` functions are involved. The CPU cost depends on the cluster sizes and the conflict resolution functions. All functions but *vote* have linear or constant complexity to the number of conflicting values. Therefore, if the *vote* function with its quadratic complexity occurs, it usually dominates the runtime for larger clusters. Otherwise, we can simply assume linear complexity. The maximum number of comparison can be therefore estimated with the cluster size histogram $\mathcal{H}_\mathcal{R}$, where $\mathcal{H}^i_\mathcal{R}$ is the number of clusters of size i in dataset $\mathcal{R}$. Further, with random sampling or even with historic data, we can measure the average cost of comparing two values with each other.

$$\#value\_comparison \leq \sum_k {\mathcal{H}^k_\mathcal{R}}^2 \cdot \#vote + \mathcal{H}^k_\mathcal{R} \cdot \#conflict\ resolutions$$

$$cpu\_cost \approx \#value\_comparison \cdot avg\_cost$$

For cost and cardinality estimation of the `duplicate removal` operator, we use the estimation of the building blocks `duplicate detection`, `transitive closure`, and `fusion`. The additional `difference` operator can be executed in parallel to the `fusion` operator and mostly generates network traffic. However, the runtime of both these two operators can be neglected compared to the runtime of the `duplicate detection` and `transitive closure` operators, so that a more sophisticated estimation is not necessary.

Summary We presented the two operators `fusion` and `duplicate removal` in this chapter. The `fusion` operator resolves conflicts within a duplicate cluster and results in a consistent representation, while the `duplicate removal` operator combines `duplicate detection`, `transitive closure`, and `fusion` to one data cleansing operator.

We first introduced a fusion model and discussed our conflict resolution functions. In contrast to previous work, we focused on functions without data dependencies to fuse clusters completely in parallel. For Sopremo's semi-structured data model, we provided four additional conflict resolution functions. Further, we incorporated the Dempster-Shafer theory of confidence in our *vote* function transparently to the user.

Moreover, we presented the integration in Meteor, which allows users to declaratively define conflict resolution functions. A user may define nested resolution and cascades of resolution for the same value, which resolve conflicts in several steps. Further, the Meteor operators provide syntactic sugar to update foreign keys to the fused dataset.

We described the implementations of the operators in Sopremo and integrated both operators into the Sopremo optimizer. The `fusion` operator has a rich set of reordering rules and properties, whereas the `duplicate removal` operator has fewer and more specialized rules. Nevertheless, some conflict resolution functions and similarity expressions allow eager-lazy optimization to reduce the number of records to be deduplicated. Additionally, we briefly outlined the cardinality and cost estimation of both operators.

These operators complete the data cleansing and integration package that we provide for interested users. In the next chapter, we showcase an Open Government Data integration project implemented with our operators.

8

Case Study: Integrating Open Government Data

Together with two student assistants, we ported parts of the large integration project GovWILD [Böhm et al., 2010, 2012b] to Stratosphere to demonstrate the applicability of our data cleansing and integration operators. The project started a bachelor project together with IBM as a sister project of IBM Midas [Balakrishnan et al., 2010], which integrates financial data. GovWILD connects European and American Open Government Data with Linked Open Datasets (Freebase and New York Times), publishes links as *owl:sameAs* RDF triples to the Linked Open Data cloud, and presents an integrated dataset with statistics through its website[1].

In this chapter, we first introduce the datasets and sketch the porting process of the integration scripts. We then briefly describe the evaluation methods, present statistics about the integration result, and evaluate the implementation with scalability experiments on a 1,000-core cluster.

8.1 Datasets

For our case study, we port the GovWILD scripts that integrate the US datasets. For the respective scripts, we acquired more recent datasets. Table 8.1 summarizes the characteristics of the datasets, which we discuss in the following. Note that throughout this thesis, most of the datasets appear as running example.

The `US Earmark` datasets can be downloaded as a CSV from the official websites of the Office of Management and Budget[2]. Earmarks were reformed and effectively banned in 2009 to avoid "waste, and fraud, and abuse"[3]. We chose the last complete and well-documented dataset from 2009 containing approximately 59,000 records with 132 attributes about long-term funds enacted by congress members. Because every year

[1] `http://govwild.hpi-web.de`
[2] `http://earmarks.omb.gov/earmarks-public/`
[3] `http://www.whitehouse.gov/the_press_office/Remarks-by-the-President-on-Earmark-Reform/`

Source	Format	Size (MiB)	Records	Attributes	Person	Legal	Fund
US-Earmark	CSV	98.4	58,751	132	X	X	X
US-Spending	CSV	2,588.8	1,219,800	46	-	X	X
US-Congress	Json	57.3	42,621	37	X	X	-
Freebase							
US-Politician	Json	3.6	2,978	49	X	-	-
Tenure	Json	57.6	72,487	20	-	X	-

Table 8.1: Characteristics of the data sets in the evaluation. Legal entities can be companies, organization, or political parties.

the schema changes drastically (from 32 - 190 attributes) and the integration effort is comparably high, we did not integrate earmarks from other years.

The largest dataset in our data integration scenario `US Spending`[4] reports all spending of the federal US government as required by the Federal Funding Accountability and Transparency Act. We downloaded the CSV from 2009, which consists of over one million records with 46 attributes. Again, the schema varies significantly between datasets from different years, such that the effort to integrate more than one dataset was too high for us.

The `US Congress`[5] dataset contains official information about parties and politicians. Since the original website does not provide a consistent dataset, we use the scrapped information from a government transparency website[6]. We verified the quality of records with over a dozen test samples before including approximately 43,000 records with 37 attributes.

`Freebase`[7] contains several community-curated datasets about important entities of our time. In particular, it contains information about politicians and their relationships to other persons and companies. We acquired two Json dumps with a systematic crawl: The first relation consists of almost 3,000 US-politicians with 49 attributes and the second relation 72,000 records about tenures with 20 attributes.

8.2 Porting the script

To write our integration scripts, we started by studying the GovWILD scripts. GovWILD uses Jaql [Beyer et al., 2011] as the main query language, which is especially suited for semi-structured data and can be executed in parallel on a Hadoop cluster. Because Jaql is too limited to perform efficient and effective duplicate detection, GovWILD invokes an external Java program using the DuDe toolkit [Draisbach and Naumann, 2010] to perform the duplicate detection with transitive closure. For fusion and RDF triple generation, GovWILD runs Jaql scripts.

[4] `http://www.usaspending.gov/data`
[5] `http://www.house.gov`
[6] `https://www.govtrack.us`
[7] `http://freebase.com/`

Hence, the workflow of GovWILD consists of three stages resulting in cluttered code, which is hard to understand. It is non-trivial to realize certain connections between pre-processing steps and duplicate detection when the documentation is not top-notch. Further, Jaql does not provide any cleansing primitives, wherefore a UDFs is needed for each scrubbing and conflict resolution task. While extensibility with UDF usually represents a strength of a system, it becomes a drawback, when half of the code has to rely on UDFs.

Because some schemata of our newer data sources deviated from GovWILD, we additionally used OpenRefine[8] (formerly Google Refine) to better comprehend the structure and the errors of the datasets. In retrospect, understanding the data sources with their individual characteristics, quirks, and errors required much more time than writing the actual integration scripts. Therefore, additional (graphical) tools are necessary to write successful integration scripts.

The final script

In the final integration script, we `scrub` each dataset individually and then use the `data map` to extract the persons, legal entities, and funds. For the Freebase datasets, we used a second `scrub` operator after the `data map` operator to validate the data with additional constraints that can be formulated only after joining both datasets. We deduplicate, cluster, and fuse persons and legal entities, whereas we assume funds to be duplicate-free within and across the two datasets. Figure 8.1 summarizes the query used in the evaluation, which is also openly and freely available at GitHub[9].

Since we focus on efficiency rather than effectiveness in this case study, we mostly transferred the conservative settings of GovWILD to our scripts. The `scrub` operators enforce 10 to 20 constraints on each of the five initial relations. We extract five attributes for funds, ten (nested) attributes for legal entities, and 17 (nested) attributes for persons. To deduplicate legal entities, we use three rules that altogether compare seven attributes in 3-pass sorted neighborhood method (SNM) on name, city, and state with a window size of 200. The duplicate detection of persons uses also three rules on a total number of six attributes with a 3-pass SNM on first name, last name, and birth date with the same window size. In both cases, we clustered with a transitive closure and fused the records with appropriate conflict resolution functions. For a complete integration, we would also need to transfer non-duplicates to the resulting datasets (for example by using the `duplicate removal` operator). However, for our experiments, the presented query allows us to easily examine individual operators.

Effort comparison

Altogether our integration script consists of fewer than 300 lines of code that are subsequently translated into 196 second-order functions and 105 execution vertices. To the best of our knowledge, this is by far the most complex data flow that has been executed

[8] `http://openrefine.org/`

[9] `https://github.com/AHeise/cleansing-example`

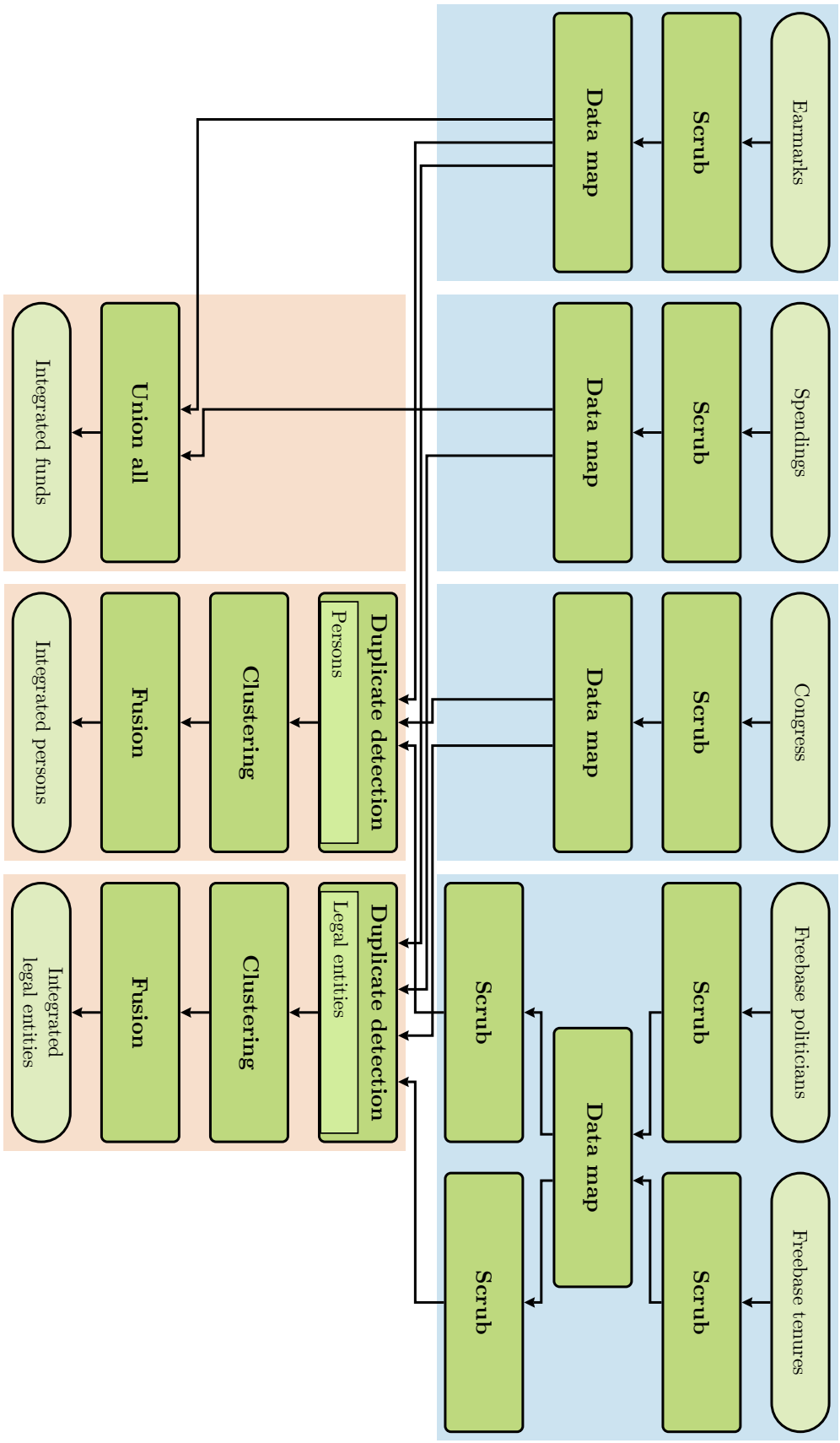


Figure 8.1: Data integration query for scalability evaluation.

on Stratosphere to date. Especially, the number of execution vertices is an order of magnitude higher than in previous jobs. Remember that the execution plan compiler aggregates successive `map` functions to one execution vertex, so that `map` heavy workflows result in few execution vertices (e.g., for information extraction).

We created the integration scripts within 100 man-hours. Because of the iterative nature of data integration, it is non-trivial to identify the work time for individual tasks. However, the acquisition and manual profiling of the datasets clearly took the longest time. Writing the scripts initially required fewer than 10 hours. Nevertheless, it took several iterations to refine the scripts (and debug some underlying functionality) until we received the final version.

The original GovWILD system uses almost 3,300 lines of Jaql plus 2,900 lines of Java code for their integration. Although they integrate more than twice the data sources, it is still apparent that our scripts are much shorter, because we use specific operators for the data integration tasks. The verbosity of the Jaql script also leads to lower maintainability. IBM recognizes the same issues and develops the high-level integration language HIL [Hernández et al., 2013] for their integration project Midas [Balakrishnan et al., 2010], which originally used a similar workflow as GovWILD.

Creating such complex workflows in Java with second-order functions would take much longer in three aspects. First, we simply used the data integration operator of this thesis out of the box, while we would need to write them from scratch in Java. Second, composing such complex Map/Reduce workflows in Java is error-prone and time-consuming as we have experienced in two seminars with altogether eight student groups. Third, debugging such workflows is daunting and can easily consume more time than the previous two aspects. With a high-level query language, such as Meteor, we can easily find erroneous integration tuples with additional queries, while it requires much effort to create such ad-hoc queries in Java or even inspect intermediate results manually.

8.3 Cluster and evaluation methods

We evaluated the scalability of our integration operators on the 1,000 core compute cluster of the Future SOC Lab. The cluster consists of 25 Quanta QSSC-S4R, each having 4 Intel Xeon E7- 4870 @ 2.40GHz and 1 TiB RAM. The nodes are connected through a 2 x Intel Corporation 82599EB 10-Gigabit network. Since local storage was unavailable for us, we simulated it with a 200 GiB Ramdisk.

For HDFS, we installed Hadoop 1.13 and configured it to use the Ramdisk. To execute of the scripts, we used Stratosphere 0.6 (Apache Flink pre-release) and the cleansing package 0.1[10]. We assigned 25 GiB RAM to the dedicated job manager and 100 GiB RAM to each task manager.

We performed two sets of experiments. In the first set, we executed the complete integration scripts without explicitly materializing any intermediate result. The second set of experiment executed each operator individually and materialized the result on

[10] `https://github.com/AHeise/sopremo-cleansing`

HDFS. For both experiments, we changed the degree of parallelism (DOP = number of total cores) and the number of slots per machine (= number of cores per machine) to measure the scalability with inter- and intra-node parallelism (scale out vs. scale up). For each DOP and slot combination, we repeated the experiment 5 times and report the mean values. To reduce caching effects, we restarted the task and job manager for each repetition. For all experiments, we report the total runtime from submitting the Meteor query to writing the final result to HDFS. Note that the Meteor parser and Sopremo compiler need fewer than three seconds to generate the data flow for the complete script.

8.4 Integration results

To put the runtime evaluation in the next section into perspective, we first review the results of the different parts of the script. Table 8.2 shows the number of extract persons, legal entities, and funds per dataset. We briefly discuss each dataset in the following.

Source	Records	Resulting Entities		
		Person	Legal entities	Fund
US-Earmark	58,751	783	9,742	11,577
US-Spending	1,219,800	-	148,647	1,219,793
US-Congress	42,621	11,734	53	-
Freebase				
US-Politician	2,978	1,900	-	-
Tenure	72,487	-	44	-

Table 8.2: Extracted entities per source.

`US Earmarks` are personally sponsored by the 535 congress members in one period. However, the fact that we extracted 783 persons already indicates duplicates within this dataset. Further, we identified almost 10,000 recipient entities in over 11,500 funds. We can conclude that the average congress member enacted almost 22 earmarks, but almost no legal entity received more than one fund if we assume that the number of duplicates in the legal entities is low.

In contrast, the `US Spending` dataset exhibits less obvious data quality problems. Only 7 of the over 1 million records have been filtered through the `scrub` operator. We extracted almost 149,000 legal entities, which corresponds to 8 average funds per entity. The `US-Congress` dataset contains approximately 12,000 historic person records of congress and senate members from 53 parties. `Freebase` contains 1,900, mostly recent US politicians from 44 parties.

The duplicate detection, clustering, and fusion of *persons* resulted in 1,889 final records. In particular, we matched 1,626 politicians from Freebase and 476 persons from `US Earmark` to 1,889 entries from `US Congress`, which corresponds to 213 persons that are contained in all three datasets.

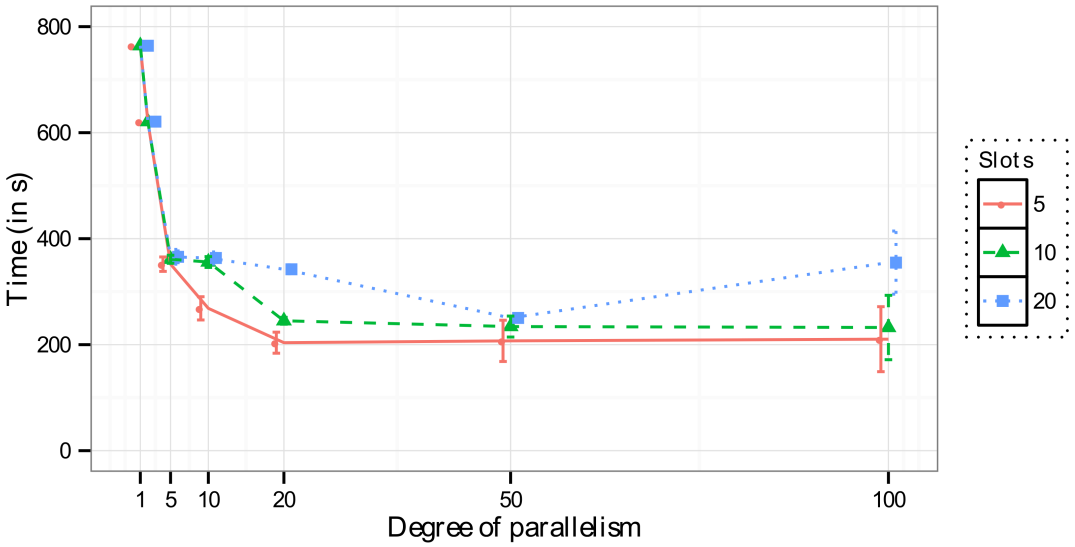


Figure 8.2: Runtime for the complete script executed with different degrees of parallelism and slots per node. The error bars represent the standard error of the measurements. For improved readability, we slightly offset the measurement points.

In comparison, we found only 278 duplicate *legal entities*. For `US-Congress` and `Freebase`, we extracted only parties, which do not appear in `US Earmarks` and `US Spending`. Further, politicians mostly enacted earmarks for non-profit organization, which rarely appear in `US Spending`. Lastly, the conservative settings taken from GovWILD are more precision-oriented, so that we received only few false positives, but probably many false negatives.

8.5 Scalability experiments

In the first experiment, we observe the scale-out properties of the complete script. The overall degree of parallelism varies from 1 to 100 and the number of slots per node from 5 to 20. Figure 8.2 depicts the mean values with standard error bars. On one core, the complete script needs 13 minutes. In the best settings, the script takes 3 minutes to complete. In the following, we discuss the measurements in more details.

We can see that a higher degree of parallelism does not necessarily improve the runtime. For five slots, the runtime does not decrease after a degree of parallelism of 20. For ten and 20 slots, the best runtime is achieved on 50 cores. Surprisingly, the runtime becomes worse for 20 slots and 100 cores. However, Stratosphere has been developed for large-scale data and our dataset sizes are comparably small, such that the overhead of scheduling outweighs the performance gain. Further, higher degrees of parallelism also cause more variance in the execution time as shown by the standard error.

A closer look at a DOP of ten reveals another oddity: Scaling out to two nodes with five slots each is faster than scaling up on one node with ten cores. We would expect that scaling up on one node should yield a better performance than scaling out to two

nodes, because scaling up profits from shared memory access and no network traffic. We have two explanations for this counter-intuitive behavior.

First, the node cannot handle scaling up from five to ten cores and become overloaded. Since we have no physical disk, the memory controller can be the only plausible bottleneck. However, the cluster nodes have been deployed and used for high-performance-computing and other tasks on up to 40 cores. It seems highly unlikely to us that our job can already overload a node with more than five worker threads.

The second – and in our view – more plausible explanation is that the Stratosphere execution engine does not properly exploit scale up. Stratosphere has been primarily developed with scale-out in mind. Indeed, this is the first experiment to our knowledge, which uses more than eight cores per node. Therefore, intra-node communication may currently be sub-optimally implemented. For example, intra-node communication could still use the network socket. If combined with less efficient communication patterns, such as used in the `cross` function, this may cause a quadratically increased load on the network controller when scaling up compared to a linear increased load for scale out settings.

Scalability of different operators

For our second experiment, we set the number of tasks per nodes to five. We executed each operator individually for degrees of parallelisms from one to 100. Figure 8.3 visualizes the average runtime for each task. The total execution time of all tasks on one core with 25 minutes is almost double as high as the execution of the complete script. We attribute the difference to the efficient physical optimization of Stratosphere and less I/O from selective materialization of intermediate data.

On one node, four tasks dominate the runtime: The `scrub` and `data map` of `US Spending` as well as the `duplicate detection` of persons and legal entities. The first two tasks mostly depend on the size of the dataset, so that `US Spending` as the biggest dataset naturally takes the longest to process. `Duplicate detection` is typically among the most expensive parts of a data integration project due to the expensive similarity measures and large search space.

In our case, we use the sorted neighborhood method, which mostly depends on the number of comparison, which in turn depends on the window size and the number of records. Intuitively, we would expect legal entity deduplication to take approximately ten times as long as person deduplication. However, in our experiments person deduplication needs almost 30% more time for three reasons. First, our similarity expression for legal entities has an efficient negative rule over the name to exclude false positives. For persons, we use a significantly less efficient negative rule over the birth date, which is often not set for our sources. Second, the overall similarity measure of person uses twice as many attributes with comparably expensive string similarity measures. Third, our person records contain biographic information and consume more CPU and network resources for serialization.

When scaling out to more nodes, we two effects: Expensive tasks become gradually faster, but cheap tasks require more time when scaling out. Both effects cancel each other

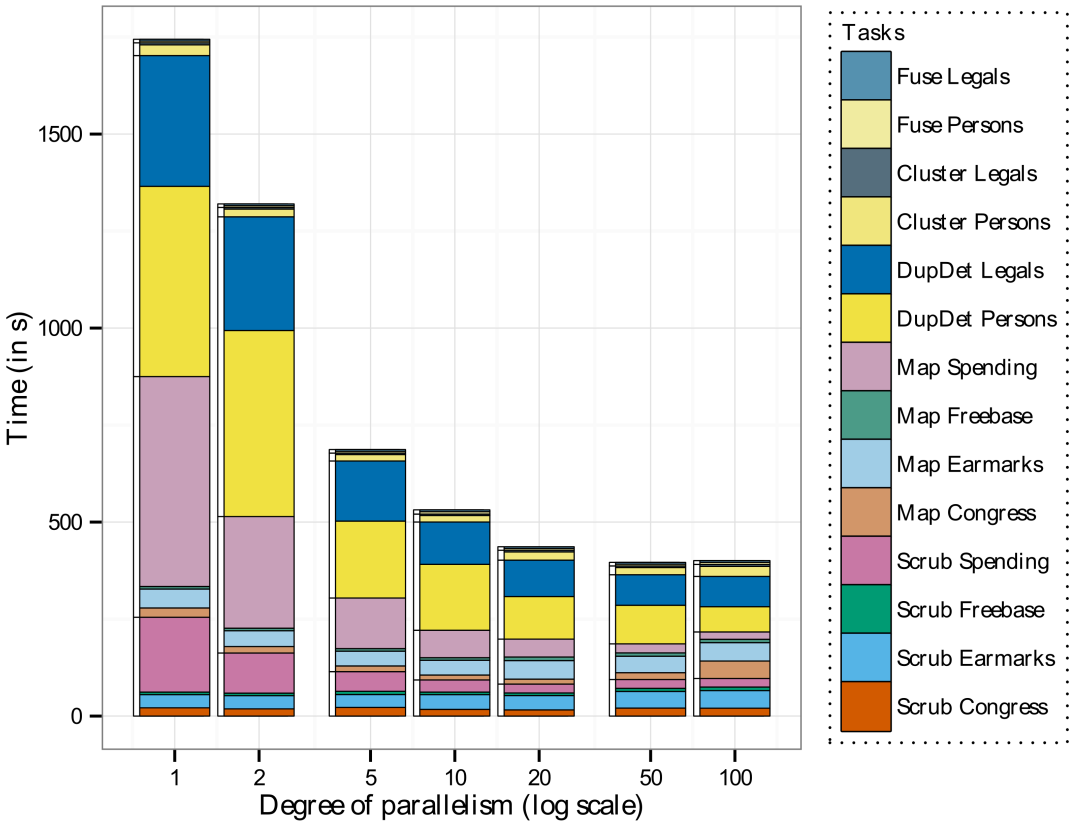


Figure 8.3: Runtime for the individual script executed with different degrees of parallelism. Note the logarithmic x-scale.

after a degree of parallelism of 20, so that we cannot see an overall runtime improvement. For the future, tasks should be more intelligently distributed: If the optimizer estimates that a higher degree of parallelism does not result in a smaller runtime, it should limit the degree for the specific task. Consequently, we achieve a better overall runtime with fewer resources.

Lastly, our operators and the entire Stratosphere project would benefit from a more sophisticated storage layer. With the current HDFS, the `scrub` and the subsequent `data map` tasks depend on the number of input splits of the data. However, in our case all datasets but the `US Spending` fit into one split. Therefore, the Stratosphere scheduler assigns only one execution vertex to the dataset and limits the scalability. A better storage layer may evenly distribute or even pre-index the records for more efficient processing similar to HAIL on Hadoop [Dittrich et al., 2012].

Final considerations

To put these measurements into perspective, the process to create the data for the original GovWILD portal can serve as a baseline. The same set of steps are performed, albeit on around double the sources and twice the total data amount. There, a mixed workflow

of Jaql (on Hadoop) and Java programs have an overall runtime of approximately five hours. We suspect that the intermediate materialization of the data as well as the startup costs of the many small Hadoop jobs contribute most to the comparably higher runtime. Consequently, development cycles to improve the workflow are tremendously shortened with our approach.

In this chapter, we evaluated our Stratosphere operators on a 1000-core cluster with an Open Government Data integration project of four data sources. The evaluation challenged Stratosphere in several ways. It is the first query with more than 100 second-order functions and execution vertices. Never before was the scale up behavior of Stratosphere tested on such large nodes. Further, Stratosphere was developed for rather data-intensive and less compute-intensive tasks. For such parallel data analytics system, our data volume and execution times are often too short to fully unleash their scalability potential.

Nevertheless, the query ran reliably on up to 100 cores and exhibited good overall runtimes. For most queries, we saw a steady decrease of overall runtime for up to 100 cores. Especially, the expensive operators profited heavily from scaling out, such that users can almost interactively tweak the various parameters of the integration queries.

9

Related Work

In this chapter, we review related work and discuss its overlap with our work. We divide the discussion according to the three main contributions of this thesis: A semantic and declarative model for common data cleansing and integration tasks, reordering rules for the operators as well as cost and cardinality estimation techniques for a full-fledged integration into the optimizer, and a scalable implementation on a Map/Reduce platform.

9.1 Declarative data cleansing systems

We first review systems with declarative operators that address data cleansing or integration workflows. Unless noted otherwise, the systems do not exploit optimization opportunities of their new operators and are not designed for a scalable execution. We start with systems that address the complete workflow and continue with those that have a narrower focus.

Galhardas et al. [2001] provide the first declarative data cleansing system **AJAX**. AJAX extends SQL with five new operators to perform data cleansing. Their `mapping` operator performs explicit mappings and enforces integrity constraints. With the `view` operator, they connect several relations from one data source to provide local views. Our `data map` operator provides both functionalities and the `scrub` operator additionally allows user to declaratively specify fine-grained record and relation constraints. The remaining three operators `matching`, `clustering`, and `merging` operators correspond to our `duplicate detection`, `clustering`, and `fusion` operators. Unfortunately, several of their operators require extensions to the query processor, such that the interpreted queries can be executed only on modified SQL execution engines.

Additionally, AJAX offers a powerful debugging interface that is triggered through exceptions and uses a sophisticated method to provide data lineage information for problematic records. Our `scrub` and `fusion` operator also allow users to review unresolved or unexpected tuples through their secondary optional outputs. However, our users have to explicitly provide lineage information to debug their scripts due to the straight-forward storage layer of Stratosphere. Lastly, AJAX performs basic physical optimization on naïve duplicate detection. Through interpretation of the similarity measure, they can

speed up computation by pruning the search space. Nevertheless, holistic optimization and scalability is not a focus of AJAX. Further, the system is limited to relational data and heavily relies on external UDFs to perform matching and fusion.

Herschel and Manolescu [2007] overcome most limitations of AJAX by implementing their declarative data cleansing system **XClean** completely on the standard XML query language XQuery. Therefore, the queries can process semi-structure data, use the full expressiveness of the extensible XQuery, and can still be executed out-of-the-box on any XQuery interpreter. Compared to our operators, XClean sub-divides the `duplicate detection` operator into `duplicate filtering` and `pairwise duplicate detection`. Moreover, it provides two operators to extract relevant entities from an XML document and merge the cleaned results back into an XML document. For our operators, the `data map` operator would handle both cases with comparable brevity.

Nevertheless, the execution of XClean and its syntax is limited by XQuery specification resulting in two drawbacks. First, the configuration of most operators is rather explicit: The scrubbing functions, data transformation, and conflict resolution functions are all explicitly specified, which limits the declarativity of the system. Second, Herschel and Manolescu do not discuss optimization of the operators and rely on the optimization of the building blocks with the underlying XQuery optimizer. Similarly, a parallel execution is not explicitly discussed, but should be possible with a parallel XQuery execution engine.

The **high-level integration language (HIL)** [Hernández et al., 2013] provides two types of declarative rules to perform data integration. First, the *entity population rules* use a SQL-like query to scrub and transform data. Second, the *entity resolution rules* perform a similarity join with optional fusion. HIL scripts are compiled to the Jaql language [Beyer et al., 2011] and can thus be executed on a Hadoop cluster. Some local optimizations are applied to each rule to translate them into efficient Jaql primitives. Further, the user may provide UDFs in Java or Jaql for scrubbing, matching, and fusion. IBM uses HIL for their Midas project [Balakrishnan et al., 2010] to integrate financial data similar to our case study in Chapter 8. The authors have been able to reduce the lengths of the integration scripts by a factor of 5, which is comparable to the saving of our scripts in our case study.

However, HIL is a specialized scripting language that is rather hard to learn and does not easily interact with other tasks. A side-effect is that the language itself does have few built-in functions; the user has to provide UDFs for basically all non-trivial transformations and matching rules. Another drawback is that the candidate selection techniques are still limited: HIL supports only naïve blocking (to allow local optimizations) and single-pass blocking. Consequently, the reported execution time of 80 minutes is not convincing for integrating five small to medium-sized relations (1,000 to 340,000 records with up to 11 attributes) on an 8-core cluster. Further, the user must provide an exact grouping key for the clustering of duplicate pairs, which is non-trivial in our experience.

Bilke et al. [2005] extend SQL with a data fusion operator. The system **HumMer** automatically infers schema mappings, performs necessary data transformations, and detects duplicates. The user provides only resolution functions to merge occurring data

conflicts. For more control over the data integration process, the user may use a graphical interface to change mappings and review duplicate definitions. However, the system does not provide a declarative way to define non-fusion configurations. The fusion operator has been separately improved by Bleiholder and Naumann [2005] as we discuss in the next section and incorporates optimization techniques.

FraQL helps users to implement integrated views on federated databases [Sattler et al., 2000]. The query language extends SQL and allows users to dynamically choose the data source to integrate with metadata similar to SchemaSQL [Lakshmanan et al., 1996]. Like our `data map` operator, FraQL overcomes structural heterogeneity with declarative view definitions. For a good runtime performance, these views can efficiently pre-select data in the federated data sources through inverse functions. While the conflict resolution with Java UDFs allows users to define powerful fusion views, the duplicate detection is limited to matching records with the same primary key (probably due to the federated data sources).

With the same scope as FraQL, **Fusionplex** provides a fine-grain data quality model to fuse data in a federated system [Motro and Anokhin, 2006]. They extend SQL to find entities in the underlying data sources that satisfy certain data quality constraints (e.g., age, transfer costs, or general availability). The data quality model additionally allows users to select the most recent or accurate value in conflict resolution functions. Analogously to FraQL, the focus of the system is data transformation and fusion, whereas the deduplication is limited to primary keys.

NADEEF aims to be first complete data cleansing system [Ebaid et al., 2013] by supporting a large and extensible set of declarative rules including (conditional) functional dependencies [Codd, 1972; Fan et al., 2008], matching dependencies [Fan, 2008], and ETL rules. For each rule, users may incorporate their domain knowledge to provide fixes similar to the repair and conflict resolution functions in our system. NADEEF explicitly models contradicting fixes with conflicting repair candidates for values. With a sophisticated heuristic, NADEEF tries to find the set of minimal changes to the data to satisfy the rules in a best effort manner. However, NADEEF cannot guarantee that all rules are satisfied in the end and different orders of rules produce different results. Further, the runtime performance of NADEEF is already poor for small datasets and the repair algorithm cannot be executed in parallel without additional assumptions.

Vassiliadis et al. [2000] present the ETL tool **Arktos** to declaratively specify data quality rules in data warehouses. It offers a graphical interface as well as SQL-like and XML-based query languages. The system validates the data warehouse periodically with a scheduler and either automatically deletes violations or loads them into special relations for manual inspection. In contrast, most industry ETL tools require explicit transformation steps of the data with composed functions or UDFs. In both cases, optimization can significantly speed up ETL workflows [Simitsis et al., 2005]. However, the provided techniques exploit only data dependencies similar to the UDF optimization in Map/Reduce systems [Hueske et al., 2012] and not semantic reordering rules as shown in this thesis with the SOFA optimizer [Rheinländer et al., 2013].

Table 9.1 summarizes the related data cleansing systems. XClean, Ajax, and HIL are the main competitors to our system and offer similar functionality. AJAX offers

	Ajax	Arktos	FraQL	Fusionplex	HIL	HumMer	NADEEF	XClean	**Stratosphere**
Scrubbing	-	+	-	-	+	$(+)^2$	+	+	+
Data mapping	+	+	+	+	+	$(+)^2$	+	+	+
Duplicate detection	+	-	-	-	+	$(+)^2$	+	+	+
Clustering	+	-	-	-	$(+)^1$	$(+)^2$	-	+	+
Data fusion	+	-	+	+	+	+	$(+)^3$	+	+
Debugging	+	-	-	-	-	+	-		(+)
Quality model	-	-	+	+	-	-	-	-	-
Extensibility	$(+)^4$	-	$(+)^4$	$(+)^4$	$(+)^4$	$(+)^4$	+	$(+)^4$	$(+)^4$
Optimization	-	-	-	-	-	-	-	$(+)^5$	+
Scale-out	-	-	-	-	+	-	-	$(+)^5$	+

Table 9.1: Related declarative data cleansing systems in comparison to Stratosphere with our cleansing operators.

[1]With grouping key
[2]With UI only, not declaratively
[3]Without conflict resolution
[4]With UDFs
[5]Through XQuery engine

the best debugging functionality, while XClean implicitly provides the best optimization of data cleansing queries. Only HIL was directly developed for scaling out, which is a relatively new trend. NADEEF supports the definition of new rule types and is the most extensible system. All other systems mostly allow users to provide UDFs for custom conflict resolution.

9.2 Optimizable data integration operators

Another direction of related work explores individual, optimizable operators in databases for data cleansing and integration. In the following, we review the first papers that introduce such related operators and discuss optimization opportunities.

The **data mapper** operator performs one-to-many data transformations similar to our `data map` operator [Carreira et al., 2007]. They provide a formal model, implementation sketch, reordering rules, and cost estimations for the new operator. Compared to our `data map` operator, the operator does not support a join over the input relations, but has additional algebraic properties, reordering opportunities, and accurate estimation due to this simplification.

Schallehn et al. [2002] introduce an extensible, similarity-based grouping operator **SimGrouping**. It allows to (transitively) group values with a (custom) similarity function. The authors extensively define the semantics, demonstrate the extensibility, and provide several implementation strategies. Further, they outline the physical optimization for special cases. The operator can be best compared to our `duplicate removal` operator with one similarity measure as the grouping key. The same authors later generalize similarity operators and additional describe the similarity join **SimJoin** [Schallehn et al., 2004]. The operator is a special case of our `record linkage` operator with one similarity measure and threshold as its similarity expression.

Silva et al. generalize the work of Schallehn et al. [2004] with their **similarity group-by** [2009] and **similarity join** [2010] operators. They define four variations of the similarity join operator for common queries and provide highly efficient implementations for them. For all operators, they prove reordering rules similar to the reordering rules for our `duplicate removal` and `record linkage` operators. All operators and variations could serve as specific implementations for our operators if they use a single similarity measure as their expression or if they can be expressed as one of their operators with additional post-processing.

The orthogonal work of Jin and Li [2005] estimates the selectivity of similarity predicates. They build histograms over differently clustered strings, which can be used to accurately estimate conditions with edit and set distances. The cardinality estimations are mandatory to integrate the previous four operators into an optimizer to assess reordered plans. Compared to our Duplicity Estimation technique (see Section 6.4), the authors fit their estimations to specific similarity measures and can therefore estimate the selectivity of a similarity `selection` or `join` more accurately. However, their scope is too narrow for us: To detect duplicates composite similarity measures in several sim-

ilarity rules are often needed, which cannot be handled by their estimation technique. Further, we estimate cluster size histograms in addition to the number of duplicate pairs.

The **Fuse By** operator fuses groups of records with common values in certain attributes [Bleiholder and Naumann, 2005]. Because its duplicate detection capabilities are limited, it can be completely translated into standard SQL with conflict resolution UDFs. The authors provide three different variations with different subsumption semantics of duplicate tuples. For all variations, they provide an extensive set of reordering rules and estimation techniques [Bleiholder, 2010], such that the operator can be fully integrated into a DBMS optimizer.

9.3 Scaling out data integration with Map/Reduce

A relatively recent trend scales out analytical queries on larger datasets on computational clusters. Map/Reduce systems allow developers to easily create scalable algorithms that can be executed on clusters with thousands of nodes. In the following, we review how our case study of Chapter 8 could be implemented on other Map/Reduce systems.

Hadoop

The most commonly used Map/Reduce system is Apache Hadoop, which implements the basic Map/Reduce paradigm of Google [Dean and Ghemawat, 2008]. With HIL [Hernández et al., 2013], Hadoop users may directly use a declarative query system for data integration. In the following, we discuss other options in decreasing order of maintainability.

For more runtime efficient queries, Hadoop users may also use **Silk** [Volz et al., 2009], **LIMES** [Ngomo and Auer, 2011], or **Dedoop** [Kolb et al., 2012a]. Silk offers a graphical workbench for users to define data transformation and matching rules to generate new links between datasets in the Linked Open Data (LOD) cloud. In comparison to the previous declarative data cleansing systems, data scrubbing and transformations are explicit, while clustering and fusion are not addressed. LIMES also efficiently detects new links between LOD datasets on Hadoop [Ngomo et al., 2013], but limits candidate pruning on the properties of metric similarities. Dedoop focuses completely on efficient duplicate detection [Kolb et al., 2011b, 2012b], but offers a rich graphical user interface, which can be used to configure advanced similarity measures even with machine learning. For fusion, both systems require additional Hadoop jobs written in one of the following ways.

The Hadoop environment offers a wide range of high level languages for analytical queries. Pig [Olston et al., 2008], Jaql [Beyer et al., 2011], and Hive [Thusoo et al., 2009] are among the most popular scripting languages. All of these languages can be used to implement complete data integration pipelines. However, none of them offer default data cleansing and integration operators. Simple scrubbing, data transformations, and final fusion can be relatively easy implemented with the relational operators that all of the query languages support. Duplicate detection requires more verbose scripts that are hard to maintain and usually take a long time to execute. Therefore, a natural choice

would be to use Dedoop for duplicate detection. Clustering usually needs several passes to be executed and only few of the script languages support loops.

The last option is to implement the complete workflow or parts thereof in Java as Map/Reduce UDFs. There is a wide range of set-similarity implementations on Map/Reduce [Rong et al., 2011; Vernica et al., 2010] that can be used as a blocking strategy for duplicate detection. However, such a complex workflow would also require a verbose driver program that invokes the different Map/Reduce jobs. Such driver programs are usually written in Java or Cascading[1] and support loops, such that the different clustering iterations can be triggered. Nevertheless, we would not recommend such custom programs due to the low maintainability of complex Map/Reduce programs.

ASTERIX/Hyracks

ASTERIX [Behm et al., 2011a] is a scalable data analytic platform similar to Stratosphere with Hyracks [Borkar et al., 2011] being the main execution engine. The ASTERIX query language (AQL) incorporates efficient similarity operators that use distributed index structures [Behm et al., 2011b].

AQL users can formulate integration scripts that perform scrubbing, data transformation, limited duplicate detection, and fusion. The duplicate detection may use a `similarity join` for blocking and a subsequent `selection` for candidate comparison. A `clustering` operator needs to be implemented manually into the system in the ALGEBRICKS layer [Alsubaiee et al., 2012]. ASTERIX supports recursive operators, which can be used to simulate the iterative nature of clustering operators.

The maintainability of AQL scripts should be higher than most Hadoop scripts, because of the similarity join primitive. Basic similarity joins are more efficient on ASTERIX, because of the sophisticated index structures. However, data integration scripts that address the complete integration workflow do not profit from these index structures, because the similarity join would be performed on unmaterialized, intermediate results. Further, the indices do not work well with complex similarity measures needed for data integration. In summary, AQL excels at similarity search, but is less suited for data integration than HIL or Stratosphere.

Spark

Apache Spark [Zaharia et al., 2010] has become increasingly popular in the past few years. It has a strong machine learning community, because it was one of the first platforms to execute iterative programs reliably. Spark offers an SQL dialect Shark [Xin et al., 2013] as its main query language and adapters to Python and Scala. Because data integration requires non-relational operators, a data integration workflow should be implemented in Python or Scala. We focus on duplicate detection and clustering, because scrubbing, data transformation, and fusion can be expressed well albeit less declarative than in our system.

[1] `http://www.cascading.org`

Recently, Zadeh and Goel [2013] contributed a similarity join operator based on matrix multiplication. It is mostly applicable to set-similarity joins, but can be used for overlapping blocking of candidate pairs. Then a subsequent `selection` can be used to perform candidate comparisons. To implement clustering, Spark fortunately supports workset iterations similar to Stratosphere [Zaharia et al., 2010]. The Python and Scala adapters provide primitives to elegantly implement the workset iterations in the script languages, such that a transitive closure can be implemented in less than ten lines of code.

Because the script is implemented in a well-known language, the maintainability is comparably high. Experienced users should directly understand the scripts, which can be modularized with existing programming techniques. For example, the duplicate detection sub-script may be implemented in a function that hides implementation details from users, so that they can focus on the main workflow.

Other scalable solutions

The *Message Passing Interface* (MPI) is another popular way to scale computations and has been successfully used for duplicate detection as well. **Febrl** achieves a near-linear scale-up on four cores with Python and MPI [Christen et al., 2004]. Böhm et al. [2012a] scale up their **LINDA** system on an 80-core server to perform compute-intensive collective entity matching on web-scale data. Since MPI inherently supports scale-out, both systems can also run on clusters.

Manually parallelized systems, such as D-Swoosh [Benjelloun et al., 2007] and P-Swoosh [Kawai et al., 2006] on Java, usually provide a good speed-up, but are naturally limited by the resources of one machine. Kim and Lee [2007] implement their parallel linkage system on MATLAB, which outperforms D-Swoosh on one machine and can even scale out. To achieve a truly parallel workflow, final products should also compute the transitive closures in parallel [Agrawal and Jagadish, 1989; Toptsis, 1991]. Finally, GPU-based duplicate detection [Forchhammer et al., 2013] and transitive closure [Katz and Jr., 2008] promise an even greater scale-up by exploiting the unparalleled processing power of graphic processors.

10

Conclusion and Future Work

In this thesis, we devised a set of data cleansing and integration operators for the parallel data analytics platform Stratosphere. For each operator, we first contributed a general model tailored to scalable execution. Second, we developed a syntax that seamlessly integrates in the Stratosphere query language Meteor. Third, we provided an implementation that directly or indirectly translates to second-order functions for the parallel execution engine. Fourth, we integrated the operators into the Sopremo optimizer by finding algebraic properties, inheritance relationships among operators, reordering rules, and efficient and accurate cost and cardinality estimations.

The first chapter introduced data quality problems and motivated the need for data integration operators. It also provided a running example of Open Government Data integration used throughout the thesis. In Chapter 2, we described Stratosphere with its three layers: Execution engine, data flow engine, and higher level language layer. We focused on the latter layer with the Stratosphere operator model Sopremo and the Meteor query language, which have been also contributed to the project as part of this thesis. Further, we briefly discussed the optimization of Sopremo operators with SOFA [Rheinländer et al., 2013].

In the following five chapters, we formally and extensively defined our operators for five common tasks in data integration projects. The `scrub` operator enforces declarative constraints on datasets and separates valid from invalid records. Next, users declaratively describe schema mappings in the `data map` operator to align the individual datasets to a specifically engineered target schema. The operator uses state-of-the-art data exchange research method to generate complex data transformation queries. Schematically aligned entities can then be further deduplicated with the `duplicate detection` operator, which supports three candidate selection techniques and flexible rule-based similarity expressions. Alternatively, users can invoke the `record linkage` operator to find duplicate entities across two unaligned datasets. Both operators return duplicate pairs, which the `clustering` operator processes with two different strategies for a consistent partitioning of the data. The `fusion` operator subsequently merges the different representations of the matched entity by using a set of conflict resolution functions and updates the foreign keys that refer to the dataset. We also provide the composite `duplicate removal` op-

erator, which combines `duplicate detection`, `clustering`, and `fusion` to a powerful deduplication operator.

We presented a case study for integrating Open Government Data and Linked Open Data with our operators. Because of the declarative specifications, the integration script in Meteor is much shorter than the comparable GovWILD script in Jaql [Beyer et al., 2011]. Our specialized operators performed significantly better on one core and even improved their runtime performance by scaling out. In particular, expensive tasks on large datasets and `duplicate detection` operators strongly benefit from scaling out on up to 100 cores.

Finally, we reviewed related work for the three main contributions: A concise set of declarative operators, integration of the operators into the optimizer, and a scalable implementation. To the best of our knowledge, there is no system apart from ours that addresses all three challenges at once. Nevertheless, we see related research directions that can be pursued building upon our system.

Fuzzy optimization

The current Stratosphere system provides only exact optimization techniques. However, for many of our operators, we can find *fuzzy optimization* rules, which may return a slightly different result, but save a significant amount of time. Especially during fine-tuning of integration script, small development cycles tremendously help the data integration experts to refine their settings and improve the effectiveness of the integration. In this case, a user may explicitly enable inexact reordering rules to achieve a more interactive development workflow.

A related research direction addresses best effort cleansing with *limited budget*. Most data cleansing tasks trade effectiveness with efficiency. For example, the more time the `duplicate detection` operator has, the larger the comparison window or block and the fewer the chance to prune a true duplicate from the search space. If a user has only a limited time or money budget, he would ideally request from the system to clean as much as possible within the budget. We already showed that *progressive duplicate detection* significantly improves the recall within the first few minutes of longer duplicate detection tasks [Papenbrock et al., accepted].

New operators

Our cleansing and integration operators address the five common tasks in data integration projects. However, there are many additional tasks that would heavily benefit from a scalable operator. For example, instance-based *schema matching* methods [Bilke and Naumann, 2005; Li and Clifton, 2000] usually compare a large portion of instance data in all column combinations over two datasets to infer likely matches as a pre-processing step to our `data map` operator. A scalable implementation would compare column combinations in parallel or even sub-partition the data.

Interaction with other operator packages

Beside our data cleansing and base package, currently the only other mature package for Stratosphere is the *information extraction* package. Together with this package, we can additionally ingest textual data for data integration projects. The operators of this package identify entities and relationships between them. Hence, we could automatically add historic and present news articles about politicians to our integrated Open Government Data (described in [Heise et al., 2012]).

For the recently started Stratosphere II project, two new sets of operators are under development. The *data profiling* package helps users to understand structures and characteristics of new large datasets. Possible operators could detect (approximate) functional dependencies, inclusion dependencies, unique column combinations [Heise et al., 2013] etc. A user may use the insights from data profiling to configure our `scrub` and `data map` operators correctly.

The second package provides *machine learning* techniques to the user. Machine learning can be used to (semi-)automatically tweak the similarity measures [Bilenko and Mooney, 2003] in the `duplicate detection` operator, to find approximate patterns useful for the `scrub` operator, or infer schema mappings [Li and Clifton, 2000] for the `data map` operator.

Improved data model

The current semi-structured data model of Stratosphere allows the ingestion of a wide range of data. However, for intermediate and final results, it would be helpful to have a more sophisticated data model that incorporates data quality information. For example, text extraction tasks could annotate a confidence score, lineage information, and their settings. Data sources would provide meta-data about their age or quality.

Our cleansing operators could be improved in the following ways. The `duplicate detection` operator could transparently add similarity scores, which the `clustering` operator uses for an advanced coherence-based clustering [Hassanzadeh and Miller, 2009]. The `fusion` can provide more conflict resolution functions based on the meta-data (e.g., most recent value in Fusionplex [Motro and Anokhin, 2006]). Finally, each record may provide data lineage information for additional debugging similar to AJAX [Galhardas et al., 2000].

Automatic setting inference

As a last future research direction, our operators may infer more settings automatically. Currently, the `record linkage` operator automatically configures the `stable matching clustering` operator with its similarity expression (see Section 6.2). Similarly, the `duplicate detection` operator could infer lossless candidate selection by interpreting rules similar to HIL [Hernández et al., 2013]. For example, negative rules allow us to pick blocking keys; that is, a distance of 0 results in 1 blocking key, a distance of 1 in three

blocking keys. Further, we can use set-similarity measures to configure set-similarity joins [Rong et al., 2011; Vernica et al., 2010].

Additionally, we might also find some settings through heuristics similar to HumMer [Bilke et al., 2005]. With data profiling, we can look for approximate dependencies and configure the `scrub` operator accordingly. Moreover, machine learning can be used to (semi-)automatically specify patterns constraints in the `scrub` operator or the similarity expression in the `duplicate detection` operator. Finally, a data quality capturing data model would allow us to pick the cleanest value according to some heuristics (e.g., actuality, extraction confidences).

The integration of our data cleansing and integration operators in Stratosphere provides users with the opportunity to declaratively formulate their data integration workflows, use functionality developed in other packages, and execute them in parallel on a cluster. In particular, our operators allow users to integrate large-scale datasets in a timely manner.

Appendix

A.1 Multikey/value data model

A flexible multikey/value data model is one of three main extensions of the Stratosphere data flow engine (see Section 2.2). In other Map/Reduce-style systems, the key/value-model is most prevailing. Usually, the value field contains all information, possibly encoded in a custom data type. In one Map/Reduce job, a typical `map` UDF receives the value with an arbitrary key and calculates the key from the value before emitting the key/value record(s). The Map/Reduce system then shuffles the records over the network to groups all records with the same key and it then invokes the `reduce` UDF for each different key once. Often, reducers also emit an arbitrary key along the value.

This design inherently exhibits three flaws. First, keys often contain redundant information as they are extracted from the actual value. Especially during the expensive shuffle phase, users would ideally remove the key information from the value and re-add it in the reducer. Second, a reducer always depends on the `map` to set the correct key. Therefore, the execution order of such Map/Reduce jobs can never be changed. Third, having only two slots for the data in the record poses an additional burden on the user. To implement composite keys, the user must either define new tuple-based key types, use existing libraries, or use quirky techniques such as specially formatted strings.

Stratosphere addresses these issues with its multikey/value data model. Users may define their own data layout and divide their record value across as many slots as they want. Given the correct type, each slot can be used as a key without prior key extraction through an separate `map` UDF. Users annotate at each second-order function which slots to use as keys and the corresponding types. Consequently, the data size is reduced and the tasks are decoupled in regard to the data layout. Further, users may directly specify combined keys by simply specifying several slots as the keys.

Figure A.1 depicts a possible data layout for our exemplary relational query (see Figure 2.3 on page 16). We store the first and the last name separately from the remaining values, because these two are needed as keys for the grouping and the `join` function. The grouping operator uses a composite key of first and last name, while the `join` function requires only the last name to be in a separate field.

Conceptually, the keys do not need to be materialized, but they must only be quickly addressable. The higher level language layer customizes the data model and stores only the offset of keys in the key slots while retaining the nested value in the last slot.

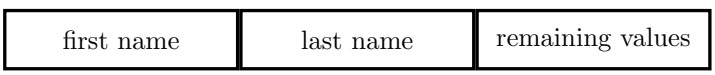


Figure A.1: Data layout of a multikey/value definition for the relational query in Figure 2.3 (Page 16). Slots can be individually or jointly used as keys.

A.2 Meteor grammar

Listing A.1 displays an excerpt of the general Meteor syntax in extended Backus-Naur Form (EBNF). A script consists of a series of semicolon-separated statements. Most statements are operator invocations but other statements define functions and constants, include other packages, and so on.

```
1 script::= (statement ';')*
2 statement::= variable(, variable)* '=' operator | funcDef |
     packageInclusion | ...
3 operator::= name+ inputs? property+
4 inputs::= (variable 'in')? variable (',' inputs)?
5 variable::= '$'name
6 property::= name+ expression
7 expression::= literal | array | object | operator | ...
```

Listing A.1: Excerpt of Meteor's EBNF grammar.

All operator specifications share a common syntax. First, users assign the output of an operator to one or more variables depending on the number of outputs of the operator. Operators may even support different output configurations, as we demonstrate with our `scrub` operator in Chapter 3. Then, users specify the operator name that may be a single verb or a verb phrase in Meteor. The package loader extracts these names from annotations on the operators. Next, the operator receives a list of input variables. Optionally, elements of the input variables can be bound to new variable names, which becomes mandatory when using the same input twice, for example in self-joins.

The operator specification concludes with a list of property configurations in the form of name and value pairs. Package developers annotate the property names to the accessor methods. Properties names usually are prepositions and values may be arbitrary complex expressions. Each operator is responsible for validating the correctness of the expression, although Meteor verifies that the expression types matches.

A.3 Compilation of Meteor scripts to Sopremo plan

Algorithm A.1 translates a given Meteor script to a Sopremo plan. Every operator invocation in Meteor results in an operator node in the Sopremo plan. Read and write invocations result in data sources and sinks respectively. The parser uses the variable names to resolve the data flow between the operators. Output variables result in an outgoing edge for each successive operator that consumes the variable. There, variables do not specify actual data but define data flow.

Consequently, variables that are never consumed are removed. If no output of an operator is consumed, it is also removed. Further, Meteor scripts do not directly specify an execution order. In our example, the `transformation` and the `grouping` are executed in parallel and the `transformation` probably finishes first, although we wrote it after the `grouping` operator in the script. The flexible execution order allows Stratosphere to optimize the queries similar to SQL databases.

```
   Input  : Meteor script
   Output: Sopremo plan
 1 variables ← new hash table;
 2 sinks ← ∅;
 3 foreach statement in script do
 4     if statement is "$output = read from ..." then
 5         newSource ← new source operator;
 6         set specified properties of newSource;
 7         put newSource with key $output into variables;
 8     else if statement is "write $input ..." then
 9         newSink ← new sink operator;
10         find input operator of newSink with $input in variables;
11         set specified properties of newSink;
12         sinks ← sinks ∪ newSink;
13     else // "outputs = name+ inputs? property+"
14         operatorType ← find operator type by name in operator registry;
15         newOperator ← new operator of type operatorType;
16         find input operators of newOperator with input list in variables;
17         set specified properties of newOperator;
18         put newOperator with output keys into variables;
19     end
20 end
21 Determine all reachable operators from sinks;
22 Create Sopremo plan from reachable operators;
```

Algorithm A.1: Parsing Meteor script to Sopremo plan.

A.4 Sample variance of cluster estimation

In Section 6.4 we developed a probabilistic sample model for `duplicate detection` with `transitive closure`. In this section, we develop the variance for the sample of the transitively closed `duplicate detection` operator $\mathcal{H}^i_{\mathcal{S}}$ (see Section 6.4). We define a random variable Y^i_C representing the probability to draw a cluster of size i from a cluster C in the original dataset and use it in Equation 6.3.

$$\mathbf{E}[Y^i_C] = \frac{\binom{|C|}{i}\binom{|\mathcal{R}\setminus C|}{n-i}}{\binom{N}{n}} \tag{A.1}$$

$$\mathbf{E}[\mathcal{H}^i_{\mathcal{S}}] = \sum_{C\in\mathcal{C}_{\mathcal{R}}} \mathbf{E}[Y^i_C] = \mathbf{E}[\sum_{C\in\mathcal{C}_{\mathcal{R}}} Y^i_C] \tag{A.2}$$

We can then define the variance of the sum of random variables by using the usual expansion over the covariance of all variable pairs.

$$\begin{aligned} var\left(\mathcal{H}^i_{\mathcal{S}}\right) &= var\left(\sum_{C\in\mathcal{C}_{\mathcal{R}}} Y^i_C\right) \\ &= \sum_{C_1\in\mathcal{C}_{\mathcal{R}}}\sum_{C_2\in\mathcal{C}_{\mathcal{R}}} cov\left(Y^i_{C_1}, Y^i_{C_2}\right) \end{aligned} \tag{A.3}$$

Thus, we need to develop the covariance over two of our random variables. We first deal with the special case that $C_1 = C_2 = C$, which will result in the variance $var\left(Y^i_C\right)$.

$$\begin{aligned} cov\left(Y^i_C, Y^i_C\right) &= E[Y^i_C \wedge Y^i_C] - E[Y^i_C]E[Y^i_C] \\ &= E[Y^i_C] - E[Y^i_C]^2 = var\left(Y^i_C\right) \end{aligned} \tag{A.4}$$

At that point, we can plug in the definition of $E[Y^i_C]$ to calculate the variance. The more general case of covariance includes the expected value of the joint event that we draw i elements from both clusters C_1 and C_2.

$$cov\left(Y^i_{C_1}, Y^i_{C_2}\right) = E[Y^i_{C_1} \wedge Y^i_{C_2}] - E[Y^i_{C_1}]E[Y^i_{C_2}] \tag{A.5}$$

We can deduce the expected value by extending our draw model. We first draw i from C_1 and then i elements from C_2. Then we fill our sample with $n-2i$ elements from the remaining dataset $\mathcal{R}\setminus C_1\setminus C_2$.

$$E[Y^i_{C_1} \wedge Y^i_{C_2}] = \frac{\binom{|C_1|}{i}\binom{|C_2|}{i}\binom{|\mathcal{R}\setminus C_1\setminus C_2|}{n-i-i}}{\binom{N}{n}} \tag{A.6}$$

At this point, we can already calculate the variance, but we can further collapse the cases for clusters with the same sizes for increased performance. For $\mathcal{H}^j_{\mathcal{R}}$ clusters of the same size j, we can group the variance for $\mathcal{H}^j_{\mathcal{R}}$ clusters, but also need to calculate the covariance for the $\mathcal{H}^j_{\mathcal{R}} - 1$ different clusters.

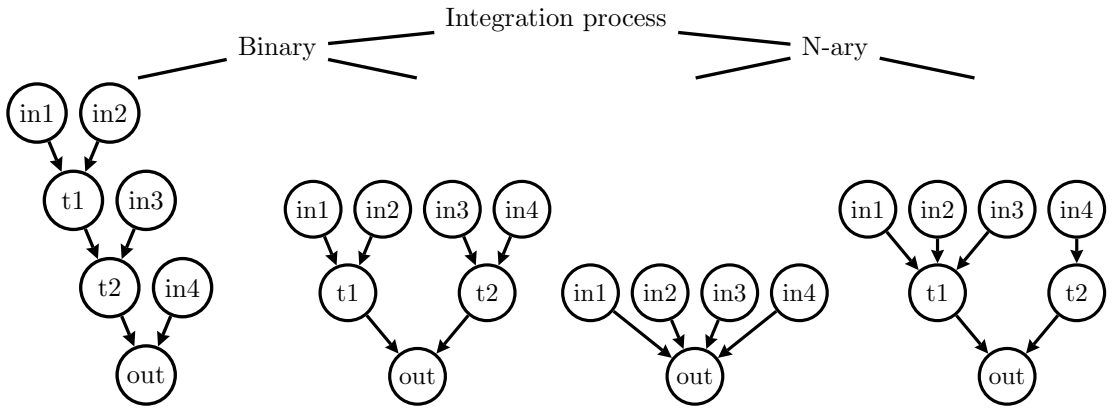
$$var\left(\mathcal{H}_{\mathcal{S}}^{i}\right) = \sum_{j=i}^{|R|} \mathcal{H}_{\mathcal{R}}^{j} var\left(Y_{C^j}^{i}\right) + \sum_{j=i}^{|R|} \mathcal{H}_{\mathcal{R}}^{j}(\mathcal{H}_{\mathcal{R}}^{j}-1) cov\left(Y_{C^j}^{i}, Y_{C^j}^{i}\right) + 2\sum_{j=i}^{|R|}\sum_{k=j+1}^{|R|} \mathcal{H}_{\mathcal{R}}^{j}\mathcal{H}_{\mathcal{R}}^{k} cov\left(Y_{C^j}^{i}, Y_{C^k}^{i}\right) \tag{A.7}$$

Table A.1 shows four different cluster size histograms of a dataset with 1,000 records and corresponding expected histograms and variances for a sample of 100 records calculated with Equation 6.3. The most apparent observation is that the larger the original cluster sizes are, the more variance all cluster sizes in the sample exhibit. For the first row without any duplicates, the variances are obviously 0. If we have 500 duplicate pairs, the variance is 4 for sampled pairs; that is, with 68% probability we sample three to seven duplicate pairs (one standard deviation). In the third and forth row, we see that the variance is as high as the probability, which underlines the difficulty to sample a large cluster reliably. Altogether, we see that randomly sampling larger cluster sizes is quite improbable and probabilistic method necessarily yield better results than exact methods.

	Original histogram				Expected histogram from sampling			
Example	$\mathcal{H}_{\mathcal{R}}^{1}$	$\mathcal{H}_{\mathcal{R}}^{2}$	$\mathcal{H}_{\mathcal{R}}^{3}$	$\mathcal{H}_{\mathcal{R}}^{4}$	$\mathcal{H}_{\mathcal{S}}^{1}$	$\mathcal{H}_{\mathcal{S}}^{2}$	$\mathcal{H}_{\mathcal{S}}^{3}$	$\mathcal{H}_{\mathcal{S}}^{4}$
1	1,000	0	0	0	100.000	0.000	0.000	0.000
2	0	500	0	0	90.090	4.955	0.000	0.000
3	0	0	0	250	73.095	12.088	0.878	0.023
4	600	100	40	20	93.604	3.030	0.109	0.002

	Expected histogram from sampling				Sample variance			
Example	$\mathcal{H}_{\mathcal{S}}^{1}$	$\mathcal{H}_{\mathcal{S}}^{2}$	$\mathcal{H}_{\mathcal{S}}^{3}$	$\mathcal{H}_{\mathcal{S}}^{4}$	$var\left(\mathcal{H}_{\mathcal{S}}^{1}\right)$	$var\left(\mathcal{H}_{\mathcal{S}}^{2}\right)$	$var\left(\mathcal{H}_{\mathcal{S}}^{3}\right)$	$var\left(\mathcal{H}_{\mathcal{S}}^{4}\right)$
1	100.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
2	90.090	4.955	0.000	0.000	16.101	4.025	0.000	0.000
3	73.095	12.088	0.878	0.024	30.449	7.370	0.820	0.024
4	93.604	3.030	0.109	0.002	11.356	2.653	0.108	0.002

Table A.1: The sample variance for four different cluster size histograms for a dataset of size 1,000. Original cluster size histogram with expected histograms from sampling 100 records (top), the expected histograms and the variance from sampling 100 records (bottom).



(a) Ladder (b) Balanced (c) One shot (d) Iterative

Figure A.2: Types of schema integration methodology [Batini et al., 1986].

A.5 Integration strategies

When integrating more than two datasets, the question arises, in which order to integrate them. Batini et al. [1986] categorize different schema integration strategies according to the number of involved steps and the number of schemata integrated in one step as depicted in Figure A.2. They mainly distinguish between binary approaches, which integrate only two schemata at one step, and n-ary solutions, which process several schemata in each step.

These schema integration strategies can be directly generalized to virtual and materialized data integration of more than two datasets. The most popular **binary** approach *ladder* recursively integrates an intermediate dataset with the next input dataset (see Figure A.2a). Hence, datasets can be integrated with decreasing importance and even intermediate results already yield value. The other binary approach *balanced* (see Figure A.2b) is less used and usually implicates that some groups of datasets are more closely related in the beginning and connections between the different groups can be found more effectively if searched between intermediate datasets.

In contrast, **n-ary** approaches provide more flexibility in their integration workflow. The *one-shot* method immediately integrates all datasets to one dataset (see Figure A.2c), while the *iterative* approach combines all other approaches and is the most flexible. While n-ary approaches provide users more expressiveness, they are also more complex. Consider a one-shot integration of n heterogeneous datasets. To find duplicates over all n datasets, the user needs to describe at least $n-1$ equivalence relations if the duplicates can be found transitively or up to $\binom{n}{2}$ equivalence relations if transitivity cannot be exploited. Such a huge number of specifications become quickly intractable on a textual interface, such as Meteor, and would even be a challenge for graphical user interfaces.

Therefore, we limit the supported integration strategies to achieve a good compromise between expressiveness and manageability of Meteor integration scripts. The `record linkage` operator may process only two schematically heterogeneous datasets at a time,

while the `duplicate detection` operator may process an arbitrary number of schematically homogeneous datasets. In the end, the following scenarios are supported with our operators:

Cleaning a single dataset: Duplicates in one dataset are found with the `duplicate detection` operator and removed with subsequent operations. Usually, a preceding `scrub` operator removes systematic value heterogeneity and eases the definition of a good equivalence relation.

Connecting two datasets: Duplicates across two datasets are found with either a `duplicate detection` or a `record linkage` operator. The `duplicate detection` operator requires aligned schemata, whereas the `record linkage` operator finds matches without prior alignment. In both cases, preceding `scrub` operators simplify the specification of the equivalence relation.

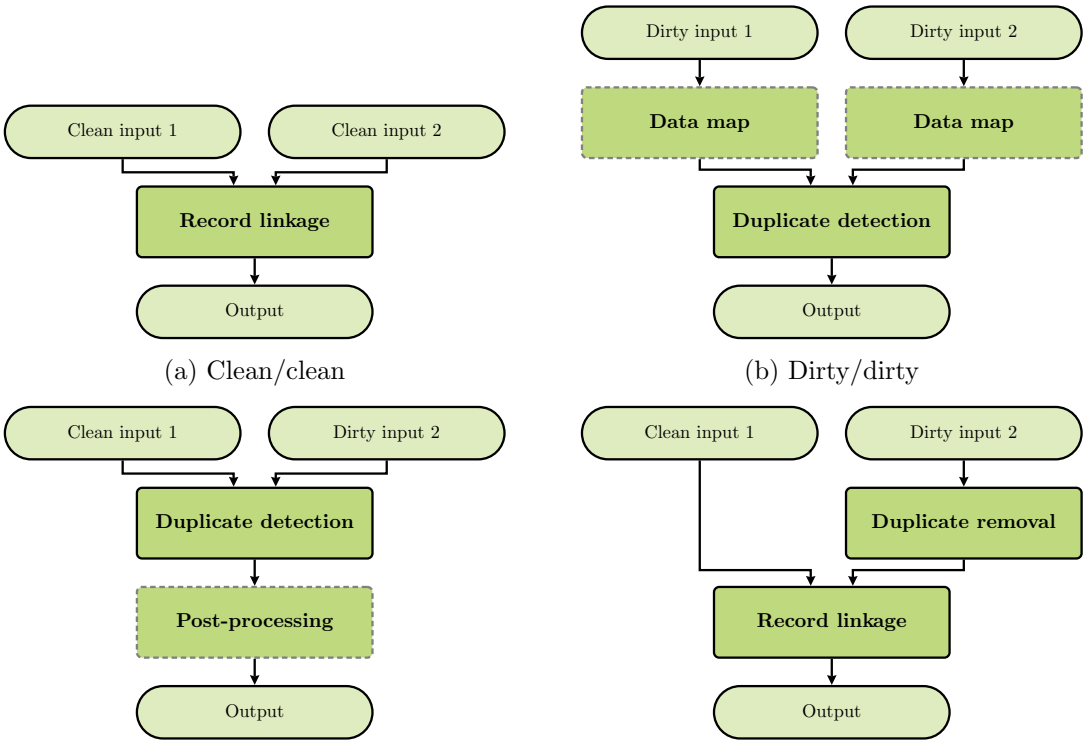
Connecting several datasets: Data integration with more than two datasets can be achieved in two principal ways. First, duplicates are found in a *one-shot* fashion with the `duplicate detection` operator after each dataset is scrubbed and aligned to a global schema. Second, each dataset is scrubbed individually and then integrated with the `record linkage` operator with any binary strategy (*ladder* or *balance*). Of course, both approaches can be mixed to perform an *iterative* integration.

Choosing a strategy for n-ary integration

A `duplicate detection` operator on more than one dataset first concatenates the datasets to a large dataset and then detects duplicates within this large dataset. Therefore, records originating in the same dataset are compared and may be incorrectly declared duplicates, although the datasets are duplicate-free. Users would then remove such unwanted matches in an additional post-processing step. However, the runtime performance of the integration query degrades with this approach, because many unnecessary comparisons are conducted, the clustering must process more intermediate duplicate pairs, and the post-processing also consumes additional resources.

In contrast, a `record linkage` operator on two datasets does not compare records from the same dataset, because a duplicate pair must consists of records originating in different datasets by definition. Especially for two datasets with notable different sizes, the `record linkage` compares significantly fewer candidates. Consider a dataset $\mathcal{R}_1$ with 100 records and a dataset $\mathcal{R}_2$ with 1,000 records. Then, the `duplicate detection` operator needs to perform $\binom{1{,}100}{2} = \frac{1{,}100 \cdot 1{,}099}{2} = 604,450$ comparisons without candidate selection and the `record linkage` operator only $1,000 \cdot 100 = 100,000$.

Therefore, users should choose an integration scenario in respect to the assumed degree of cleanliness within a dataset as shown in Figure A.3. For two datasets we can distinguish three cases. In the best case, the user knows that both datasets are clean (i.e., duplicate-free) and one `record linkage` operator suffices (Figure A.3a). In the opposite case, both datasets are dirty and contain duplicates. Then, we can align both



(a) Clean/clean

(b) Dirty/dirty

(c) Clean/dirty inputs: Post-processing to remove unwanted duplicates

(d) Clean/dirty inputs: Duplicate removal to preclean input 2

Figure A.3: Queries for different types of clean/dirty dataset combinations.

datasets and use a `duplicate detection` operator to find duplicates within each dataset and across datasets simultaneously (Figure A.3b). The third case, involves one dirty and one clean dataset. Using either operator does not result in the desired result: The `duplicate detection` operator would search and potentially declare duplicate within the clean dataset and the `record linkage` operator would not look for duplicates in the dirty dataset. The first solution in Figure A.3c removes unwanted duplicates from the result of a `duplicate detection` operator with a post-processing step. Alternatively, we can also pre-clean the dirty dataset with a `duplicate removal`[1] operator and then use one `record linkage` as shown in Figure A.3c . Latter approach is more complex, but also more efficient if the dirty dataset is significantly smaller than the clean dataset.

Finally, to generalize these cases to more than two datasets, we can differentiate between binary and n-ary strategies when observing the cleanliness of the datasets. On the one hand, if most datasets are clean, then we recommend a *binary ladder* approach starting with the smaller datasets. All intermediate results are clean and therefore a cascade of `record linkage` operators helps to improve the performance by comparing relatively few records. The few dirty datasets can then be preprocessed with the `duplicate removal` operator.

[1] `Duplicate removal` combines duplicate detection with fusion as explained in Chapter 7

On the other hand, if most datasets are dirty, then a *one-shot* integration with a `duplicate detection` operator with additional post-processing should be the better choice. First, the script becomes shorter, because of fewer operators. Second, the number of comparisons is not significantly higher compared to using `record linkage` operator for the few clean datasets. Third, the post-processing to remove unwanted false positives within a clean dataset creates little overhead when only few duplicates are expected to be removed. For data integration scenarios with a similar number of clean and dirty datasets, we recommend an *iterative* strategy to maximize the performance, where first all dirty datasets are integrated with a *one-shot* approach and then the clean datasets are added with a *ladder* strategy. Of course, this approach is only feasible, when the qualities of different integration orders are comparable. Otherwise, the quality naturally dictates the strategy and the choice of the operators.

References

Rakesh Agrawal and H. V. Jagadish. Multiprocessor Transitive Closure Algorithms. *IEEE Data Engineering Bulletin*, 12(1):30–36, 1989.

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 *Proceedings of the Conference Datenbanksysteme in Büro, Technik und Wissenschaft (BTW)*, pages 25–44, 2011.

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, MatthiasJ. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The Stratosphere platform for big data analytics. *VLDB Journal*, pages 1–26, 2014.

Sattam Alsubaiee, Yasser Altowim, Hotham Altwaijry, Alexander Behm, Vinayak R. Borkar, Yingyi Bu, Michael J. Carey, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Nicola Onose, Pouria Pirzadeh, Rares Vernica, and Jian Wen. ASTERIX: An Open Source System for "Big Data" Management and Analysis. *Proceedings of the VLDB Endowment (PVLDB)*, 5(12):1898–1901, 2012.

Marcelo Arenas, Leopoldo E. Bertossi, and Jan Chomicki. Consistent Query Answers in Inconsistent Databases. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 68–79, 1999.

Sreeram Balakrishnan, Vivian Chu, Mauricio A. Hernández, Howard Ho, Rajasekar Krishnamurthy, Shixia Liu, Jan Pieper, Jeffrey S. Pierce, Lucian Popa, Christine Robson, Lei Shi, Ioana Roxana Stanoi, Edison L. Ting, Shivakumar Vaithyanathan, and Huahai Yang. Midas: integrating public financial data. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 1187–1190, 2010.

Carlo Batini, Maurizio Lenzerini, and Shamkant B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18(4): 323–364, 1986.

Rohan Baxter, Peter Christen, and Tim Churches. A comparison of fast blocking methods for record linkage. In *Proceedings of the ACM International Conference on Knowledge discovery and data mining (SIGKDD)*, volume 3, pages 25–27, 2003.

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, 2011a.

Alexander Behm, Chen Li, and Michael J. Carey. Answering approximate string queries on large data sets using external memory. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 888–899, 2011b.

Omar Benjelloun, Hector Garcia-Molina, Heng Gong, Hideki Kawai, Tait Eliott Larson, David Menestrina, and Sutthipong Thavisomboon. D-Swoosh: A Family of Algorithms for Generic, Distributed Entity Resolution. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, page 37, 2007.

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. *Proceedings of the VLDB Endowment (PVLDB)*, 4(12):1272–1283, 2011.

Christoph Böhm, Felix Naumann, Markus Freitag, Stefan George, Norman Höfler, Martin Köppelmann, Claudia Lehmann, Andrina Mascher, and Tobias Schmidt. Linking open government data: what journalists wish they had known. In *Proceedings of the International Conference on Semantic Systems (I-SEMANTICS)*, 2010.

Christoph Böhm, Gerard de Melo, Felix Naumann, and Gerhard Weikum. LINDA: distributed web-of-data-scale entity matching. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 2104–2108, 2012a.

Christoph Böhm, Markus Freitag, Arvid Heise, Claudia Lehmann, Andrina Mascher, Felix Naumann, Mauricio Hernandez, Vuk Ercegovac, and Peter Haase. GovWILD: Integrating Open Government Data for Transparency. In *Proceedings of the International World Wide Web Conference (WWW)*, 2012b. Demo.

Mikhail Bilenko and Raymond J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *Proceedings of the ACM International Conference on Knowledge discovery and data mining (SIGKDD)*, pages 39–48, 2003.

Mikhail Bilenko, Beena Kamath, and Raymond J. Mooney. Adaptive Blocking: Learning to Scale Up Record Linkage. In *Proceedings of the International Conference on Data Mining series (ICDM)*, pages 87–96, 2006.

Alexander Bilke and Felix Naumann. Schema Matching using Duplicates. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 69–80, 2005.

Alexander Bilke, Jens Bleiholder, Christoph Böhm, Karsten Draba, Felix Naumann, and Melanie Weis. Automatic Data Fusion with HumMer. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 1251–1254, 2005.

Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A Comparison of Join Algorithms for Log Processing in MapReduce. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 975–986, 2010.

Jens Bleiholder. Data Fusion and Conflict Resolution in Integrated Information Systems, 2010.

Jens Bleiholder and Felix Naumann. Declarative data fusion – Syntax, semantics, and implementation. In *Advances in Databases and Information Systems (ADBIS)*, 2005.

Jens Bleiholder and Felix Naumann. Conflict Handling Strategies in an Integrated Information System. In *Proceedings of the International Workshop on Information Integration on the Web (IIWeb)*, 2006.

Jens Bleiholder and Felix Naumann. Data Fusion. *ACM Computing Surveys*, 41(1), 2008.

Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1151–1162, 2011.

Paulo Carreira, Helena Galhardas, Antónia Lopes, and João Pereira. One-to-many data transformations through data mappers. *Data and Knowledge Engineering (DKE)*, 62 (3):483–503, 2007.

Peter Christen, Tim Churches, and Markus Hegland. Febrl - A Parallel Open Source Data Linkage System. In *Proceedings of the Advances in Knowledge Discovery and Data Mining (PAKDD)*, pages 638–647, 2004.

E. F. Codd. Relational Completeness of Data Base Sublanguages. *Database Systems*, 1972.

Lex de Haan and Toon Koppelaars. *Applied Mathematics for Database Professionals*. Expert's Voice. Apress, 2007.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Stefan Richter, Stefan Schuh, Alekh Jindal, and Jörg Schad. Only Aggressive Elephants are Fast Elephants. *Proceedings of the VLDB Endowment (PVLDB)*, 5(11):1591–1602, 2012.

Uwe Draisbach and Felix Naumann. DuDe: The Duplicate Detection Toolkit. In *Proceedings of the International Workshop on Quality in Databases (QDB)*, 2010.

Amr Ebaid, Ahmed K. Elmagarmid, Ihab F. Ilyas, Mourad Ouzzani, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. NADEEF: A Generalized Data Cleaning System. *Proceedings of the VLDB Endowment (PVLDB)*, 6(12):1218–1221, 2013.

Wayne Eckerson. Data quality and the bottom line. *TDWI Report, The Data Warehouse Institute*, 2002.

Ahmed Elmagarmid, Panagiotis Ipeirotis, and Vassilios Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 19(1):1–16, 2007.

Khaled Elmeleegy. Piranha: Optimizing Short Jobs in Hadoop. *Proceedings of the VLDB Endowment (PVLDB)*, 6(11):985–996, August 2013.

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

Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theoretical Computer Science*, 336(1):89–124, 2005a.

Ronald Fagin, Phokion G. Kolaitis, and Lucian Popa. Data Exchange: Getting to the core. *ACM Transactions on Database Systems (TODS)*, 30(1):174–210, 2005b.

Wenfei Fan. Dependencies revisited for improving data quality. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 159–170, 2008.

Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Transactions on Database Systems (TODS)*, 33(2), 2008.

Wenfei Fan, Shuai Ma, Nan Tang, and Wenyuan Yu. Interaction between Record Matching and Data Repairing. *Journal of Data and Information Quality*, 4(4):16, 2014.

Benedikt Forchhammer, Thorsten Papenbrock, Thomas Stening, Sven Viehmeier, Uwe Draisbach, and Felix Naumann. Duplicate Detection on GPUs. In *Proceedings of the Conference Datenbanksysteme in Büro, Technik und Wissenschaft (BTW)*, pages 165–184, 2013.

David Gale and Marilda Sotomayor. Ms. Machiavelli and the Stable Matching Problem. *The American Mathematical Monthly*, 92(4):261–268, 1985.

Helena Galhardas, Daniela Florescu, Dennis Shasha, and Eric Simon. AJAX: An Extensible Data Cleaning Tool. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, page 590. ACM, 2000.

Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon, and Cristian-Augustin Saita. Declarative Data Cleaning: Language, Model, and Algorithms. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 371–380, 2001.

Georg Gottlob and Alan Nash. Efficient core computation in data exchange. *Journal of the ACM*, 55(2), 2008.

Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: Database-supported program execution. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 1063–1066. ACM, 2009.

Alon Halevy, Anand Rajaraman, and Joann Ordille. Data integration: The teenage years. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 9–16. VLDB Endowment, 2006.

Oktie Hassanzadeh and Renée J. Miller. Creating probabilistic databases from duplicated data. *VLDB Journal*, 18(5):1141–1166, 2009.

Taher H. Haveliwala, Aristides Gionis, and Piotr Indyk. Scalable Techniques for Clustering the Web. In *Proceedings of the ACM SIGMOD Workshop on the Web and Databases (WebDB)*, pages 129–134, 2000.

Arvid Heise and Felix Naumann. Integrating open government data with stratosphere for more transparency. *Web Semantics: Science, Services and Agents on the World Wide Web*, 14(0):45–56, 2012.

Arvid Heise, Astrid Rheinländer, Marcus Leich, Ulf Leser, and Felix Naumann. Meteor/-Sopremo: An Extensible Query Language and Operator Model. In *BigData workshop at VLDB*, 2012.

Arvid Heise, Jorge-Arnulfo Quiané-Ruiz, Ziawasch Abedjan, Anja Jentzsch, and Felix Naumann. Scalable Discovery of Unique Column Combinations. *Proceedings of the VLDB Endowment (PVLDB)*, 7(4):301–312, 2013.

Arvid Heise, Gjergji Kasneci, and Felix Naumann. Estimating the Number and Sizes of Fuzzy-Duplicate Clusters. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, 2014.

Mauricio A. Hernández, Georgia Koutrika, Rajasekar Krishnamurthy, Lucian Popa, and Ryan Wisnesky. HIL: a high-level scripting language for entity integration. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 549–560, 2013.

Mauricio A. Hernández and Salvatore J. Stolfo. The Merge/Purge Problem for Large Databases. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 127–138, 1995.

Melanie Herschel and Ioana Manolescu. Declarative XML Data Cleaning with XClean. In *Proceedings of the Conference on Advanced Information Systems Engineering (CAiSE)*, volume 4495 of *Lecture Notes in Computer Science*, pages 96–110. Springer Verlag, 2007.

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

Fabian Hueske, Aljoscha Krettek, and Kostas Tzoumas. Enabling Operator Reordering in Data Flow Programs Through Static Code Analysis. *Computing Research Repository (CoRR)*, abs/1301.4200, 2013a.

Fabian Hueske, Mathias Peters, Aljoscha Krettek, Matthias Ringwald, Kostas Tzoumas, Volker Markl, and Johann-Christoph Freytag. Peeking into the Optimization of Data Flow Programs with MapReduce-style UDFs. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2013b. Demo.

Liang Jin and Chen Li. Selectivity Estimation for Fuzzy String Predicates in Large Data Sets. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 397–408, 2005.

Gary J. Katz and Joseph T. Kider Jr. All-Pairs Shortest-Paths for Large Graphs on the GPU. In *Proceedings of the ACM International Conference on Graphics Hardware (SIGGRAPH)*, pages 47–55, 2008.

Hideki Kawai, Hector Garcia-Molina, Omar Benjelloun, David Menestrina, Euijong Whang, and Heng Gong. P-Swoosh: Parallel Algorithm for Generic Entity Resolution. Technical Report 2006-19, Stanford InfoLab, September 2006.

Hung-sik Kim and Dongwon Lee. Parallel linkage. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 283–292, 2007.

Lars Kolb, Andreas Thor, and Erhard Rahm. Block-based load balancing for entity resolution with MapReduce. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 2397–2400, 2011a.

Lars Kolb, Andreas Thor, and Erhard Rahm. Parallel Sorted Neighborhood Blocking with MapReduce. In *Proceedings of the Conference Datenbanksysteme in Büro, Technik und Wissenschaft (BTW)*, pages 45–64, 2011b.

Lars Kolb, Andreas Thor, and Erhard Rahm. Dedoop: Efficient Deduplication with Hadoop. *Proceedings of the VLDB Endowment (PVLDB)*, 5(12):1878–1881, 2012a.

Lars Kolb, Andreas Thor, and Erhard Rahm. Load Balancing for MapReduce-based Entity Resolution. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 618–629, 2012b.

Laks V. S. Lakshmanan, Fereidoon Sadri, and Iyer N. Subramanian. SchemaSQL - A Language for Interoperability in Relational Multi-Database Systems. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 239–250. Morgan Kaufmann, 1996.

Dustin Lange and Felix Naumann. Frequency-aware similarity measures: why Arnold Schwarzenegger is always a duplicate. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 243–248, 2011.

Amy Nicole Langville and Carl Dean Meyer. Survey: Deeper Inside PageRank. *Internet Mathematics*, 1(3):335–380, 2003.

Wen-Syan Li and Chris Clifton. SEMINT: A tool for identifying attribute correspondences in heterogeneous databases using neural networks. *Data and Knowledge Engineering (DKE)*, 33(1):49–84, 2000.

David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. Testing Implications of Data Dependencies. *ACM Transactions on Database Systems (TODS)*, pages 455–469, 1979.

Bruno Marnette, Giansalvatore Mecca, Paolo Papotti, Salvatore Raunich, and Donatello Santoro. ++Spicy: an OpenSource Tool for Second-Generation Schema Mapping and Data Exchange. *Proceedings of the VLDB Endowment (PVLDB)*, 4(12):1438–1441, 2011.

Giansalvatore Mecca, Paolo Papotti, and Salvatore Raunich. Core schema mappings. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 655–668, 2009.

Renée J. Miller, Laura M. Haas, and Mauricio A. Hernández. Schema Mapping as Query Discovery. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 77–88, 2000.

Alvaro E. Monge and Charles Elkan. The Field Matching Problem: Algorithms and Applications. In *Proceedings of the ACM International Conference on Knowledge discovery and data mining (SIGKDD)*, pages 267–270, 1996.

Amihai Motro and Philipp Anokhin. Fusionplex: resolution of data inconsistencies in the integration of heterogeneous information sources. *Information Fusion*, 7(2):176–196, 2006.

Felix Naumann. Data profiling revisited. *SIGMOD Record*, 42(4):40–49, 2013.

Felix Naumann and Matthias Häussler. Declarative Data Merging with Conflict Resolution. In *Proceedings of the International Conference on Information Quality (IQ)*, pages 212–224, 2002.

Axel-Cyrille Ngonga Ngomo and Sören Auer. LIMES - A Time-Efficient Approach for Large-Scale Link Discovery on the Web of Data. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2312–2317, 2011.

Axel-Cyrille Ngonga Ngomo, Lars Kolb, Norman Heino, Michael Hartung, Sören Auer, and Erhard Rahm. When to Reach for the Cloud: Using Parallel Hardware for Link Discovery. In *Proceedings of the International Conference on The Semantic Web: Semantics and Big Data (ESWC)*, pages 275–289, 2013.

Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 1099–1110, 2008.

Thorsten Papenbrock, Arvid Heise, and Felix Naumann. Progressive Duplicate Detection. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, accepted.

Lucian Popa, Yannis Velegrakis, Renée J. Miller, Mauricio A. Hernández, and Ronald Fagin. Translating Web Data. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 598–609, 2002.

Erhard Rahm and Hong Hai Do. Data Cleaning: Problems and Current Approaches. *IEEE Data Engineering Bulletin*, 23(4):3–13, 2000.

Astrid Rheinländer, Arvid Heise, Fabian Hueske, Ulf Leser, and Felix Naumann. SOFA: An Extensible Logical Optimizer for UDF-heavy Dataflows. Technical report, CoRR, 2013.

Astrid Rheinländer, Martin Beckmann, Anja Kunkel, Arvid Heise, Thomas Stoltmann, and Ulf Leser. Versatile optimization of UDF-heavy data flows with Sofa. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2014. Demo.

Chuitian Rong, Wei Lu, Xiaoyong Du, and Xiao Zhang. Efficient Duplicate Detection on Cloud Using a New Signature Scheme. In *Proceedings of the International Web-Age Information Management (WAIM)*, pages 251–263, 2011.

Kai-Uwe Sattler, Stefan Conrad, and Gunter Saake. Adding Conflict Resolution Features to a Query Language for Database Federations. In *Proceedings of the International Workshop on Engineering Federated Information Systems (EFIS)*, pages 41–52, 2000.

Eike Schallehn, Kai-Uwe Sattler, and Gunter Saake. Extensible and Similarity-Based Grouping for Data Integration. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 277–288, 2002.

Eike Schallehn, Kai-Uwe Sattler, and Gunter Saake. Efficient similarity-based operations for data integration. *Data and Knowledge Engineering (DKE)*, 48(3):361–387, 2004.

Glenn Shafer. *A Mathematical Theory of Evidence*. Princeton University Press, Princeton, 1976.

Jeffrey Shafer, Scott Rixner, and Alan L. Cox. The Hadoop distributed filesystem: Balancing portability and performance. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 122–133, 2010.

Yasin N. Silva, Walid G. Aref, and Mohamed H. Ali. Similarity Group-By. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 904–915, 2009.

Yasin N. Silva, Walid G. Aref, and Mohamed H. Ali. The similarity join database operator. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 892–903, 2010.

Alkis Simitsis, Panos Vassiliadis, and Timos K. Sellis. Optimizing ETL Processes in Data Warehouses. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 564–575, 2005.

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. *Proceedings of the VLDB Endowment (PVLDB)*, 2(2):1626–1629, 2009.

Anestis A. Toptsis. Parallel Transitive Closure Computation in Highly Scalable Multiprocessors. In *Proceedings of the International Conference on Computing and Information (ICCI)*, pages 197–206, 1991.

Panos Vassiliadis, Zografoula Vagena, Spiros Skiadopoulos, and Nikos Karayannidis. ARKTOS: A Tool For Data Cleaning and Transformation in Data Warehouse Environments. *IEEE Data Engineering Bulletin*, 23(4):42–47, 2000.

Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using MapReduce. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 495–506, 2010.

Tobias Vogel, Arvid Heise, Uwe Draisbach, Dustin Lange, and Felix Naumann. Reach for Gold: An Annealing Standard to Evaluate Duplicate Detection Results. *Journal of Data and Information Quality*, 5(1–2), 2014.

Julius Volz, Christian Bizer, Martin Gaedke, and Georgi Kobilarov. Discovering and Maintaining Links on the Web of Data. In *International Semantic Web Conference (ISWC)*, pages 650–665, 2009.

Daniel Warneke and Odej Kao. Nephele: Efficient parallel data processing in the cloud. In *Workshop on Many-Task Computing on Grids and Supercomputers (SC-MTAGS)*, 2009.

William E. Winkler. The state of record linkage and current research problems. Statistics of Income Division, Internal Revenue Service Publication R99/04, 1999.

Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: SQL and rich analytics at scale. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 13–24, 2013.

Weipeng P. Yan and Per-Åke Larson. Eager Aggregation and Lazy Aggregation. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 345–357, 1995.

Reza Bosagh Zadeh and Ashish Goel. Dimension independent similarity computation. *Journal of Machine Learning Research*, 14(1):1605–1626, 2013.

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the USENIX conference on Hot topics in cloud computing (HotCloud)*, 2010.

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Doktorarbeit mit dem Thema:

Data Cleansing and Integration Operators for a Parallel Data Analytics Platform

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Potsdam, den 30. September 2014