Business Intelligence on Scalable Architectures

Csaba István Sidló

Supervisor: András Lukács Ph.D.

Eötvös Loránd University
Faculty of Informatics
Department of Information Systems

Ph.D. School of Computer Science
János Demetrovics D.Sc.

Ph.D. Program of Information Systems
András Benczúr D.Sc.

Budapest, 2011.

Abstract

This thesis considers the overlapping areas of database management and business intelligence. We present new results that aim to narrow the gap between the disciplines of databases and data mining that are still considered rather separate at the time of writing.

We study three selected problems on the boundary of databases and data mining. The first one, frequent itemset mining (FIM), is a classic and central data mining task. The thesis describes new FIM algorithms for relational database environments. We demonstrate that the new algorithms efficiently utilize the facilities provided by the database server, and fit the relational data model and specialties of the environment.

Second, business intelligence architectures are studied. A new architecture type is presented to integrate relational data warehouses and column-oriented storages in a cost-effective way, enabling long-term storage and data mining.

Third, the thesis provides new results on the problem of entity resolution, a central and complex data integration task. We present new models and scalable algorithms for relational databases, for effective index use, as well as for distributed environments. We demonstrate a significant improvement in scalability compared to previously known similar algorithms.

Results are validated over real life data and presented at international conferences. The thesis demonstrates that the results also proved to be useful in practice through industrial applications.

Acknowledgments

First and foremost, I would like to thank András A. Benczúr for valuable feedback, support, and guidance. Many thanks go to all members of the Web Search and Data Mining Research Group of MTA SZTAKI for providing me with an outstanding intellectual environment and support. The Information Systems Department of Eötvös Loránd University, especially Attila Kiss and András Benczúr provided valuable support; it has been a great privilege to work with and learn from them.

I thank Zoltán Hans on the AEGON Hungary side, who contributed with discussions on the problem formulation and clarification of entity resolution applications.

Finally, I thank my parents, my family and my lovely Zita for their unconditional support and love.

Contents

Abstract **i**

Acknowledgments **ii**

Contents **iv**

1 Introduction **1**

1.1 Thesis Outline and Contributions 2

2 Preliminaries **4**

2.1 Business Intelligence 4

2.2 Business Intelligence Applications 6

3 Architectures for Business Intelligence **8**

3.1 Coupling Data Mining and Databases 9

3.2 Our Results 10

3.3 The Two-Phase Architecture 12

3.3.1 Second Phase Reference Implementation 13

3.3.2 A Case Study 16

3.4 Conclusions 20

3.5 My Contribution 21

4 Frequent Itemset Mining **22**

4.1 Problem Formulation 22

4.2 Data Mining inside Relational Databases 23

4.2.1 Related Work 23

4.2.2 SQL-based Apriori Algorithms 24

4.2.3 Pattern-growth Methods 25

4.2.4 Experiments 30

4.3 Conclusions . . . 34
4.4 My Contributions . . . 35

5 Entity Resolution **36**
5.1 Motivation . . . 37
5.1.1 Client Databases . . . 37
5.1.2 Practical Observations and Assumptions . . . 39
5.2 Related Work . . . 41
5.3 ER Models and Formulations . . . 43
5.3.1 Generic ER Model . . . 43
5.3.2 Relational Generic ER Model . . . 44
5.3.3 Record Partitioning Model . . . 46
5.3.4 Matching and Indexability . . . 48
5.3.5 Satisfactory Records . . . 51
5.3.6 Probabilistic Model . . . 51
5.4 Algorithms for Relational Databases . . . 55
5.4.1 Swoosh Variations . . . 56
5.4.2 DB-GER and Variations . . . 58
5.4.3 Experiments . . . 62
5.5 Algorithms with Efficient Indexing . . . 66
5.5.1 Basic Feature Indexing . . . 66
5.5.2 Feature Pre-indexing . . . 68
5.5.3 Feature-based Blocking . . . 69
5.5.4 Index Realizations . . . 70
5.5.5 Experiments . . . 70
5.6 Distributed Algorithms . . . 73
5.6.1 Magnitudes of Input Data . . . 75
5.6.2 A Lower Communication Complexity Bound . . . 76
5.6.3 Algorithms Based on Key-Value Stores . . . 77
5.6.4 Algorithm Based on Map-Reduce . . . 80
5.6.5 A Bulk Synchronous Parallel Algorithm . . . 82
5.6.6 Experiments . . . 86
5.7 Conclusions . . . 90
5.8 My Contribution . . . 92

6 Summary and Conclusions **94**

List of Figures

3.1 Two-phase data warehouse architecture 11
3.2 The snowflake schema of the data warehouse in the case study . 17
3.3 Execution times for the reference queries 19
3.4 Frequent itemset mining on the ACCIDENTS dataset 19
3.5 Comparing average execution times of regular weblog processing jobs 21

4.1 FP-tree for a given database, built with minimum support threshold 2 26
4.2 Constructing the FP-tree for selected databases and minimum support values 32
4.3 Execution times on the RETAIL dataset 32
4.4 Execution times on the T10I4D100K and ACCIDENTS datasets 33
4.5 Execution times on the BMS-WebView datasets 33
4.6 Execution times on weblog datasets 34

5.1 Simplified data model of the insurance client data mart 39
5.2 Example of records belonging to the same entity 47
5.3 Scalability of the algorithms (with linear and log time scale) . . 63
5.4 Percent of execution time needed to eliminate dominated records 64
5.5 Impact of match selectivity 65
5.6 Performance of DB-GER with different indexing strategies . . . 65
5.7 Execution times against input size 72
5.8 Processing speed change for 1.1 M records 73
5.9 Processing speed change for 7.2 M records 74
5.10 Execution time against input size for all algorithms 87
5.11 Execution time against input size for all algorithms, below 20 million records 88

5.12 Execution times plotted against the number of merged entities in 5 million records 88
5.13 Cumulative execution times of the distinct phases in the BSP algorithm 89
5.14 Cumulative execution times of the distinct phases in the Map-Reduce algorithm 90
5.15 Scalability of our entity resolution algorithms. 92
5.16 Scalability of our entity resolution algorithms, below 20 million records. 93

List of Tables

3.1 Space requirements of the page hit data cube (one month) 18
3.2 Reference queries 19

4.1 Dataset properties 31

List of Algorithms

1 FP-tree-based DB FIM 28
2 Dominated record elimination 56
3 R-Swoosh 57
4 DB-G-GER 59
5 DB-GER 60
6 Index-ER 67
7 Pre-Index-ER 68
8 Block-Index-ER 69
9 Union-Find ER by distributed key-value store 78
10 BFS ER by distributed key-value store 79
11 ER by Map-Reduce 80
12 Map-Reduce connected components 81
13 Bulk Synchronous Parallel (BSP) ER 82
14 Data node algorithm for BSP ER 83
15 Master node algorithm for BSP ER 83
16 BSP Connected Component algorithm 85

Chapter 1

Introduction

The first decade of the new millennium was eventful in the IT world. For an everyday person, it was all about broadband internet, new ways of social interaction by social networking, new generation of mobile devices, interfaces and applications, sharing and searching incredible amounts of data and content. For an IT professional, it was about cloud computing, web 2.0, mobile applications, big data, new architectures and new paradigms. New trends changed the way people think about information technologies, how and why these technologies develop.

The promising decision support technologies of the 90's become accepted and widespread. Data warehousing, data mining, data visualization and other analytical procedures and technologies take part in the life of today's organizations. Meanwhile new questions have arisen. More and more data, the demand on real time processing, new emerging technologies are to be considered when developing decision support systems.

This thesis discusses selected topics of decision support technologies, intended to provide answers to some of these newly emerged questions.

A few years ago my main interest was how to adopt data mining algorithms to relational data warehouses. The work on efficient frequent itemset mining and entity resolution algorithms for relational databases were inspired by this question.

Today the main question of my interest is how analytical solutions can benefit from the new emerging architectures and from frameworks capable of handling petabytes of data. The two-phase architecture presented in this thesis gives a possible answer to the weaknesses of traditional relational databases when solving data mining tasks on big data. The work on distributed and indexing-based entity resolution demonstrates how scalability of existing algorithms can be overshadowed by the help of new architectures and tools.

My results range from theory to experimentation and industrial deployment. I present a mixture of theoretical running time bounds, models and heuristics for solving practical problems, and measurements on real life data. Unusual for a doctoral dissertation, some of my results are implemented as part of the daily operation of our partners.

1.1 Thesis Outline and Contributions

The thesis begins with a preliminary chapter with a short introduction on business intelligence, data mining, entity resolution and motivating use cases. The subsequent chapters deal with specific business intelligence topics, each with introduction, motivating examples, related work, detailed description of our results and notes on my contribution.

Chapter 3 gives a detailed discussion of business intelligence architectures, concentrating on how relational databases can be extended to support data mining efficiently.

A new architecture type is developed and described that can integrate relational data warehouses and colum-oriented storages in a cost-effective way, enabling long-term storage and data mining. The results were validated on the massive weblog data of the Hungarian [origo] web portal, building prototype two-phase system.

Chapter 4 gives an example how a data mining algorithm can be tightly coupled to a database: we present efficient frequent itemset mining algorithms for relational databases. These algorithm efficiently utilize the facilities provided by the database server, and fit the relational data model and specialties of the environment. Applicability is demonstrated by experiments and a real-world web analytics application.

In Chapter 5 we contribute to data quality by developing new, efficient methods for entity resolution. We scale up the entity resolution process by tightly coupling algorithms to databases, by indexing efficiently and also by distributing the algorithms, with one Section for each approach.

Applicability is demonstrated by experiments and by case studies, primarily on insurance client data. A large Hungarian insurance company successfully applies the methods and algorithms for years in a client data warehouse application.

Chapter 6 concludes by summing up new results and my contribution.

The work described in this thesis has been partly presented in the following publications listed in chronological order:

- Results of Chapter 4 were published in [Sidl 05b].

- Base ideas of the two-phase architecture of Chapter 3 appeared first in [Benc 05], then [Racz 07] gave a detailed description of the concept.
- Entity resolution methods of Chapter 5 were published in three research papers. Relational ER of Section 5.4 is described in [Sidl 09], algorithms of Section 5.5 with heavy indexing appeared in [Sidl 11a] and distributed solutions of Section 5.6 were presented in [Sidl 11b].

Chapter 2

Preliminaries

The decision support technologies from the '90s, OLAP, data warehousing, data mining, business intelligence found their places. They have become popular and used in everyday practice, not only supporting decisions, but supporting operational activities of today's organizations. Next we briefly introduce the key concepts of business intelligence and some related concepts.

2.1 Business Intelligence

The term **business intelligence (BI)** was coined by Howard Dresner, former analyst of Gartner Inc. He defined BI in 1989 as "a set of concepts and methodologies to improve decision making in business through use of facts and fact-based systems" [Mart 06]. In the 1990s the phrase has become popular, used as an umbrella term, for applications, databases, methods and architectures providing easy access to business data. The main focus of BI are **Decision-Support Systems (DSS)**, offering valuable knowledge to decision makers. The definitions of BI and DSS are rather vague and diverse. DSS for example can be handled as a special BI domain. Besides, BI systems are sometimes described as DSS systems, with the opposite inclusion relation.

In any case, BI (and therefore the fact-based systems in the definition) includes the following activities and areas of our interest ([Moss 03]):

- on-line analytical processing (OLAP),
- data mining,
- click-stream analysis,
- data warehousing,

- master data management.

One of our main BI subjects is **data mining**, a crucial step in knowledge discovery. Data mining, a multidisciplinar area, addresses the discovery of hidden, possibly unexpected, but useful patterns in big data. Data mining methods serve the most complex information needs of organizations, providing useful models of the subject areas (prediction models and classificators, frequent patterns, clusters).

Our main data mining problem is **frequent itemset mining (FIM)**, a classic and central data mining task. The main task of FIM is to find frequent patterns in large databases of transactions (baskets) containing items. An example for FIM is the widely known problem of market basket analysis. The goal is to find sets of items with support (the count of transactions containing all the items in the set) above a given minimum support threshold. FIM is a base to solve several further tasks, including association rule, frequent sequence or subgraph mining. The area has extensive literature, with many algorithms and implementations. Our main interest is the relationship between FIM algorithms and relational databases.

Data integration is the task of combining data of different sources, providing a unified view. Data integration tasks occur frequently when building BI applications, and require massive data quality improvement methods. Data mining methods are also useless without consistent and clean datasets. A central data quality task is **Entity Resolution (ER)** (or deduplication), the process of identifying groups of records that refer to the same real-world entity. Duplicated entity representations raise severe data quality issues leading to corrupted aggregations that may eventually mislead management decisions or operational processes.

Several business intelligence, and also several operative applications need effective and scalable entity resolution algorithms. CRM, marketing, master data management, or even web search engines and other web-based services would profit from identifying duplicates. Without ER even simple questions can be hard to answer, as for example “How many clients we actually have?” or “Is our client already accessed by direct marketing, targeted advertisement?”.

ER is a computationally hard problem with $O(n^2)$ time complexity, and the main difficulty comes from the size of the data sets. No algorithms were published for large data volumes in practical applications: Previous algorithms of the literature were usually designed as in-memory algorithms for small input data set.

BI solutions can be built on several types of IT architectures. Relational data warehouses form a widely accepted platform: they build on traditional relational databases. Scalability has recently imposed new challenges as the volume of the data grows very fast while the increase in the capabilities of a single processor core has slowed down. Distributed systems, MapReduce, Key-Value stores address scalability and have already begun to reshape the BI industry. It is however not yet clear how existing BI methodologies can be applied in these new architectures, or how traditional architectures can be extended to adopt to higher scalability requirements.

2.2 Business Intelligence Applications

The thesis focuses on some selected BI applications. These subject application areas are emerging from practical applications of our algorithms, and from practical needs of R&D partners.

One area of our interest are the storage and analysis of big transactional data. The demand of the telecommunication industry, service and content providers emerges to collect and analyze usage and other transactional data. Some of their questions regarding security, service improvement, marketing or business policy issues can be answered by data mining methods. Even medium sized companies can easily produce log files of extreme sizes (up to hundreds of gigabytes or terabytes per month). Collecting, keeping these log data sets in a storage-efficient and easily accessible way suitable for direct processing by several types of data analysis and mining algorithms is a challenging problem.

Web analytics is an application of transactional log processing aiming to collect and analyze internet data, with the intent to understand and optimize web usage. Here recording activity of website visitors and other web usage events may produce huge amounts of data. Since we first met the problem the area undergone great changes, industry standard methods and applications emerged and become widely accepted and used.

Another possible application of transactional log management methods is the **analysis of IT infrastructure logs**. The IT infrastructure of any organization produces huge amounts of diverse log data: authentication events, hardware failures, resource (disk) shortage as well as auditing database access, people entering doors, printer toners running out. The area is similar to web analytics. In fact, web analytics can be considered as a sub-task of IT-log analytics. However, general IT-log analytics is more complex task and does not have standard methodologies and applications yet.

Client master data management is a third area where we build prototype applications with our new methods. A common task in practice is to collect, aggregate, consolidate, quality-assure and distribute descriptive data of the main subjects, in our case specifically clients. High quality master data is required in all BI applications, including data warehousing, data mining and operational applications such as CRM (customer relationship management) or marketing. We focus on data quality issues of client master data when building a client data warehouse.

Chapter 3

Architectures for Business Intelligence

In this chapter we study how relational databases can be extended to a data mining and business intelligence platform. Efficient solutions already exist for solving data mining tasks, but these are mostly independent of the database management system containing the data they work with. After identifying the limitations of the relational technology and the needs of data mining tasks, an architecture can be designed to efficiently combine the advantages of different storage architectures.

A common practice in relational data warehousing and also in ad-hoc analytical applications is the use of relational databases. Advanced and mature methodologies and tools can be chosen both to design and to implement a relational data warehouse, see [Kimb 11].

Column-oriented database systems such as the *C-store* [Ston 05] extend traditional relational databases with the ability to compress data highly efficient at the cost of slower random access and update. These systems are read-optimized databases designed for relatively few concurrent users asking sophisticated queries by storing relational data in columns rather than in rows as in the conventional "row-store" approach.

A relatively new trend is to integrate row and column-wise storage methods. For example, both SAP with a new storage engine called HANA ([Cha 11]), and Microsoft with SQL Server ([Lars 11]) integrate in-memory column-oriented storage techniques into their databases, blurring the boundaries between column-wise and row-wise storage. This idea is similar to our two-phase architecture following in Section 3.2. Column-oriented storage methods also found their way into MapReduce systems (see [Flor 11] for an example).

Distributing databases is a common way to make our databases scalable.

Hadoop [Whit 10], an open source implementation of the **Map-Reduce** framework [Dean 08] is a common architecture for efficient parallelization.

Emerging distributed **NoSQL** (non-relational) technologies represent another useful option (see [Leav 10] or [Ston 10a] for brief overview), providing high scalability for limited classes of problems. These include hierarchical, graph, object-oriented databases, key-value stores and document databases, each taking a different data storage and access approach. Examples include MemcacheDB, ScalienDB, Dynamo, CouchDB and MongoDB. A great debate is developing if NoSQL solutions can substitute traditional relational databases, are the presumptions they build on useful or not.

A possible answer to the concerns above – as by the column- and row-oriented storage methods – is to bring the NoSQL and SQL worlds closer. For example, Michael Stonebraker, an influential database researcher and architect, founder of several database companies, also a critic of overstating NoSQL skills (see [Ston 10b]), recently founded VoltDB. The distributed main memory-based VoltDB originates from H-Store [Kall 08], and promises excellent scalability, relational model with SQL and ACID compliancy [Hari 08].

In this chapter we present an architecture that we designed for web log mining at the largest Hungarian Web portal [origo] (www.origo.hu). The site [origo] among others provides online news and magazines, community pages, software downloads, free email as well as a search engine. The portal has a size of over 700 thousand pages and receives 7 million page hits on a typical workday, producing 35 GB of raw server logs that remains a size of 4.5 GB per day even after cleansing and preprocessing, thus overrunning the storage space and analysis capabilities of typical commercial systems.

3.1 Coupling Data Mining and Databases

A fundamental question of business intelligence architectures relates to the integration of data mining into traditional databases. The goals of data mining, the extraction of interesting knowledge from large databases complements the goals of the data warehouse and on-line analytical processing technologies. The chasm between the existing *data mining* and the *database* world is rather wide. Most data mining solutions include fully database-independent applications for the data mining tasks. As relational databases are widespread and common tools, coupling data mining with relational databases can remarkably improve knowledge discovery capabilities.

The concept of **inductive databases** [Imie 96, Boul 99] is about integrating data and knowledge. The main goal of an inductive database is to allow the user not only to query the data that resides in the database, but also to query and mine generalizations, patterns of interest. The knowledge discovery process should be supported by an integrated framework, the user should be allowed to perform different operations on both data and patterns.

The interaction takes place through inductive query languages supporting data mining, often extensions of SQL. A good comparison between languages supporting descriptive rule mining can be found in [Bott 04]. Other directions allowing data mining-like queries are data mining APIs [Netz 01]. From the analyst point of view the usability of OLAP systems could also be significantly increased by the integration of data mining methods. This viewpoint of inductive databases is called **on-line analytical mining** (OLAM) [Han 98].

Despite the probable usefulness we are still far away from a general theory and practical realizations of full value of inductive databases. Moreover, mainstream research interest seems to turn to other topics, including new scalable architectures with new ways of knowledge discovery. However, there are promising partial results, including that RDBMS vendors try to integrate more and more knowledge discovery support in their systems, turning them into decision support platforms (see [Li 04] and [MacL 04]).

Data mining enabled database system architectures can be classified by the strength of the coupling between the mining algorithms and the database [Sara 98]:

Tight coupling. Data mining is *integrated* into the DBMS using the existing query processing and storage methods. When used for data mining, this solution has known efficiency limitations [Sara 98, Sidl 05b].

Semi-tight coupling. The existing DBMS system is extended with data mining primitives of different complexity and provides an extension of the SQL query language. Data mining algorithms work separately, but data access is provided through a common interface.

Loose coupling. Data are read from the DBMS and directly loaded into a separate DM system. Both our and typical existing data mining solutions are loosely-coupled and use locally stored copies.

3.2 Our Results

Around 2006 new technologies, column oriented, distributed and NoSQL stores were just in incubatory phase. Some limitations of traditional databases for business intelligence purposes were already clearly visible. At this time, we experimented with a new architecture type, aiming to extend traditional relational data warehouses with data mining and big data handling capabilities.

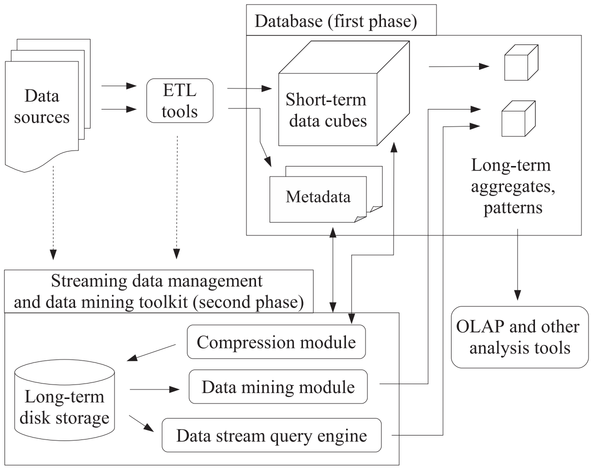


Figure 3.1: Two-phase data warehouse architecture

We proposed a "two-phase data warehouse architecture" by adding a second phase data source to a traditional DBMS. The philosophy of the second phase lays between data streams, nearline databases and column-oriented databases: we provide very fast streaming access for full table scans of long-term fine granularity historical data stored column-wise compressed. While our data mining algorithms remain loosely coupled to the DBMS, they are tightly coupled with the second phase data source and thus overcome the high cost of reading from DBMS. Our architecture promises effective long-term archival, computation and data mining at low costs as demonstrated by a case study of a large scale internet content provider server log warehouse.

The prototype system includes several modules integrated into a complete service for processing and analyzing web logs. After collection and filtering data may enter advanced analysis, optimized high density compression for long-term storage in a form appropriate for off-line data mining, as well as an OLAP-based statistical unit that provides fast on-line service for basic aggregation and detailed short-term queries.

3.3 The Two-Phase Architecture

Figure 3.1 shows the proposal of the two-phase architecture. Data storage and management is divided into two separate components — data is present in two different phases.

The first phase is a traditional DBMS that processes data sources and provides user interface for management and analysis. The first phase can be implemented using standard data warehouse technologies. The source data is extracted, transformed and loaded into the database by the ETL tools. The DBMS contains detailed dimension tables with no restriction on its scheme (snowflake or even more complex). The database is responsible for managing data cubes and metadata, such as dimension tables. First the dimensions and dimension hierarchies are refreshed, and fully detailed data cubes are built. Additional cubes store the derived, subject area-specific data, which are updated after the main cubes. These smaller cubes contain data with no time restrictions. (They can be implemented as independent data marts as well [Desi 01].) By cost considerations the first phase is restricted in its time window or granularity; for historic data only aggregates are preserved. Reporting and analysis is done by tools connecting to the first phase data warehouse. From the usage point of view, the first phase is responsible both for serving OLAP and ad-hoc queries and reports, and for providing predefined data mining patterns generated from the second phase data.

The new element is an additional second phase similar to nearline archive storages but with slightly different goals. The background storage keeps data available through streaming and data mining interfaces and also provides archival. Data is imported either from the first phase or, for real time applications, directly from data sources. Data is stored compressed.

A goal of the second phase is to optimize data mining access to very large data sets. The second phase is restricted to a simple schema with one or more basic fact tables. These are read only except for appending new data, and are optimized for full table scans with possibly only very coarse or no indexes implemented.

We expect a data streaming interface; sequential access to all or certain relatively large blocks of the data fits well with a variety of data mining and machine learning algorithms. Frequent itemset mining as well as partitioning clustering algorithms such as k-means algorithms are designed to use a few passes of full scans over the data [Han 00]. A given model can be applied to classify data; certain models such as Naive Bayes or decision trees can also easily be trained by sequential passes. Finally database sketches or synopses can be efficiently built for further processing [Babc 02].

The second phase data consists of one or more basic fact tables with all frequently used attributes joined. Access is restricted *read-only sequential* with the exception of inserting new data. The second phase does not support random access and arbitrary attribute indexes. Basically it acts both as a nearline solution to store archive data and as a data source for tightly integrated mining algorithms.

The interaction between first and second phase can have multiple forms according to the desired goals. For example we may use the second phase data to generate a new aggregated view in the data warehouse that cannot be deduced from the currently available aggregations. In this case we need to read through all historical data that can be efficiently solved by a single query over the second-phase data to initialize the data cube with accurate "phase back" data as required. The same issue arises when the new system has to be initialized with historical data.

Service providers and companies are usually required to keep detailed data about their customers' actions for a long time. This require access to the most detailed historical data. Telecom sector is a good example of such storage requirement. While basic retrieval functionality can be provided with standard off-line storage solution (eg. tape archival), there are certain circumstances where these methods cannot easily provide adequate answers. One such example is when, instead of a particular record set or time frame, all the historical data has to be searched for a specific pattern (e.g. fraudulent use). Loading hundreds or thousand of tapes for such a query is an extremely costly operation, especially when the pattern may not be formulated perfectly and several passes might be necessary. In contrast, the modest data volumes of the second-phase compressed data allows fast access. Efficient disaster protection by feasible replication of the most detailed data across different sites is also easily achievable.

3.3.1 Second Phase Reference Implementation

In our reference implementation the entire second phase is external to the first phase data warehouse system. The compression/decompression module and the second-phase query engine (including data mining operations) were all developed with keeping the immense data volumes in mind. However, this is not a requirement of the architecture. Handling of compressed tables and streaming operations could be integrated in commercial database or data warehouse systems, sharing much of the code with traditional query execution. Unfortunately the only option not feasible is the golden mean, when only the compression/decompression module is external to the database engine. In this case during the full table scan of second-phase data such a large volume of data has to pass the interface between the DBMS and the external module which makes this option infeasible at hundreds of millions of records.

We primarily use the second phase data over a streaming interface without ever storing large uncompressed data chunks. This is achieved by a compressed storage optimized for high throughput by large compression rates. Compressing the data is relatively slow but rare, while decompression is frequently used but very fast. We employ our semantic compression column by column [Racz 04]. The method is similar to column stores in the use of compression, but differs in other aspects. A main difference is that we only compress data in columns but we store in rows in order to serve data mining algorithms without the need of joins on multiple columns.

The prototype second phase does not prohibit indexes similar to column stores. The only index type we use is a very coarse block index suitable for example to select data of different time periods. Queries hence use full or partial table scans and joins are implemented only for small tables by internal memory indexes. Notice that in our high throughput phase full table scans are relative inexpensive operations.

Similar to our second phase, the column stores enable sophisticated compression methods [Abad 06] and evaluate queries on compressed data as far as it is possible. While we also compress data column-wise, we store them as a row store as an additional benefit over column stores to avoid the use of join indexes. Nevertheless the possibility of keeping the data in column-wise separate files as well as multiple views under different sorting and partitioning is possible in our second phase although our data management philosophy stays closer to data streams than to common database principles.

Modular Design and Interfaces

The second phase is designed and developed for handling and processing very large datasets, according to a data stream approach. It combines an abstraction of data source, run-time modularity and configurability with keeping performance and resource management issues in hand. A versatile platform is built for data mining and data analysis that allows an ordinary desktop or workstation-range computer to perform queries including data mining operations on very large datasets that are difficult to handle by boxed DBMS or statistical systems.

A single modular suite of software is written entirely from scratch in C++ language that spans the tasks of compression, high-throughput data access and data mining. All the second-phase tasks are performed by a set of over 200 independent modules for data manipulation (filters, transforms, aggregation, grouping, sorting etc.) and modeling (clustering, classification, frequent pattern mining). Module configuration can be given in a query language that specifies the required modules, their settings and virtually unrestricted interconnection over a standard interface.

The modules adhere the streaming data source interface both for input and output with a limited number of exceptions that use external memory sorting. By connecting several modules together, one gets a *pipeline* of modules that performs complex operations.

Data streaming allows second-phase data to be decompressed on-the-fly for queries while never stored, not even temporarily. Only the small set of records being currently processed exist in the memory uncompressed.

Compression

Compression is a key factor in the efficiency of the second phase, which our compression module fulfils. In the prototype we used our semantics-based compression scheme [Racz 04] utilizing inter-block data correlation. We optimized for decompression speed as it is a considerably more frequent operation than compression. Instead of a universal compression algorithm we provide a choice of several algorithms specialized for diverse data types. Different compressed parts may share metadata such as compression models, histograms or data distribution information. When querying we utilize the ordering created by the compression module and the natural order of blocks/partitions (usually according to time) as a coarse index. Using compression blocks we can access data according to the filter conditions of the current query.

Data Model

In the second phase storage and query execution engine we use a data model fitting to data mining applications, a generalization of the relational data model and the sparse matrix data model. The model is similar to the model of BigTable [Chan 06].

A relation is a set of rows. The sparse format means following: for each row of the matrix, we take the set of nonzero entries as a list of their column identifiers. In our model we group the attributes of the classic relational model into *header* attributes and *body* attributes. Those records, which have the same values in all the header attributes constitute a *row* and are collected together. The values of the header attributes are stored, processed and transmitted only once for each row, independently of the number of records belonging to that row. A basic relation can be represented easily in our model: all attributes can be header attributes (one row for each record) or there are no header attributes (in which case there will be a single row).

3.3.2 A Case Study

The two-phase data concept was tested in a case study with the largest Hungarian web portal (www.origo.hu). In the time of our experiments the portal had from 7 to 9 million successful page requests per day, resulting in around 35 GB of raw web server log files each day. This amount of data should remain accessible for a long time to meet legal obligation and security liabilities and for analytics. Since the cost of a huge traditional data warehouse was not affordable, they used to build data marts on aggregated data while archive the raw data on tapes that is very difficult to access and analyze.

Our solution is built on top of an Oracle 9i database that we also use for presenting reports to users through an easy-to-use web-based interface. Web server log data enters the DBMS through our ETL tools; in addition these tools collect data from the editorial systems as well. The raw data is filtered, completed (with domain names for IP addresses for example) before sent to the data warehouse component. The main page-hit data cube is partitioned by date: each day has a new partition. We also build hierarchies on top of the dimensions including domains for the IP addresses and page groups based on the structure of the site.

The most granular page hit data cube can store data only for a limited time period, approximately for five months. This data is aggregated into smaller, subject oriented data cubes for the purposes of content optimization, marketing and web-design without capacity problems.

The data warehouse has a refresh period of one day, new data arrives overnight. First we make the database consistent by building the main table. At this point the second phase data is generated: the compression module reads out and compresses the most detailed data. After compression additional data mining tasks are performed, and the results are written back into the database for analysis. Second-phase modules connect to the database through standard Oracle C++ APIs.

To ensure that the second-phase data is always consistent to the database, they share all the data warehouse dimensions and metadata. However, the dimension data is stored also in the second phase as dictionaries, to handle changes of the schema and slowly changing dimensions. Archive, rolled-out detailed data of the star schema is always properly accessible from the second-phase. The second-phase modules are built on the relational-like data model which fits the snowflake schema of Figure 3.2 using the same surrogate keys of dimensions as the database. The schema is designed according to common clickstream data warehouse practice [Swei 02].

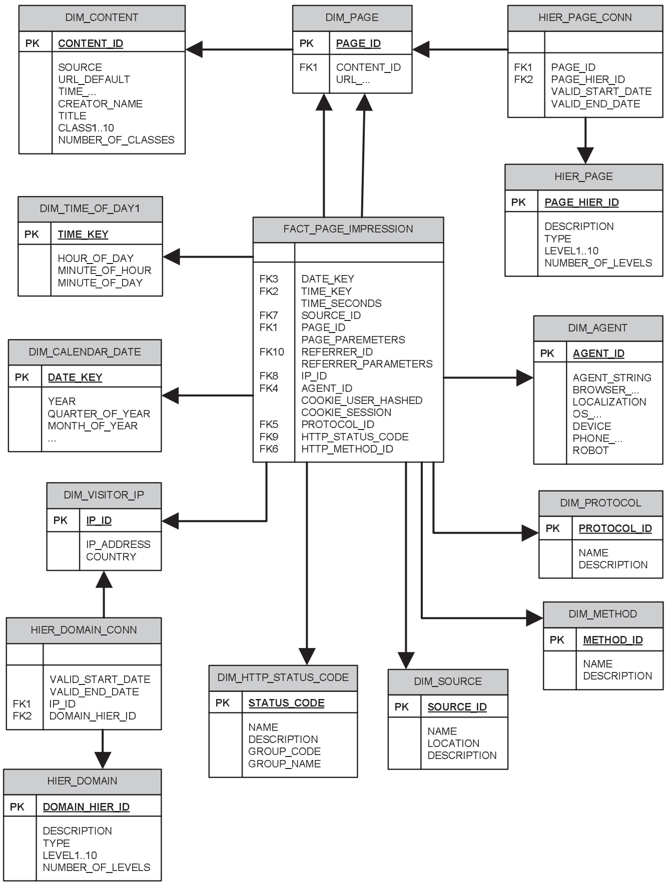


Figure 3.2: The snowflake schema of the data warehouse in the case study

Table 3.1: Space requirements of the page hit data cube (one month)

Storing method	Size on disk
Compressed (bzip) raw log files	180.7 GB
Compressed (bzip), preprocessed log files	17.1 GB
Standard DB table	44.9 GB
Compressed DB table	39.1 GB
Second-phase compressed storage	1.9 GB

Experiments

In the following we demonstrate the usability of the two-phase architecture by selected weblog measurements. We compare the database and the second phase store according to space requirements and query performance. While the TPC-H benchmark [TPC 06] is a good reference, unfortunately it is not applicable due to the specialties of the business area and the second-phase storage. Instead, as in [Josh 03], we measure basic data warehouse implementation-independent properties.

Table 3.1 shows the space requirements of one month (31 days) web log data in the different phases. DB table references to the main fact table of the most granular page hit data, without the dimension and hierarchy tables. The second-phase storage contains the basic dimension attributes as well. Oracle's data segment compression technique [Poss 03] achieves around 13 % storage saving, as a result of the very sparse dataset. Our compression method reduces storage down to 4.2 % of the basic fact table.

Figure 3.3 shows execution times for some basic queries of Table 3.2, observed on the same server. We chose queries that do not use dimension tables and require full table scans (we do not use daily block index that would speed up our execution of Q3) of the fact table that store the required foreign keys. An appropriate tuning of the database requires careful and accurate implementation beyond the scope of the thesis.

Querying second-phase data has near constant performance. The reason of the higher Q3 execution time is the following: the granularity (partition size) of the compressed data is one month. In case of a query for one week of data we have to process a whole month to produce the result, in contrast to the database engine, where the query can be optimized to read only the appropriate fact table partitions for the selected date range. However, the granularity of the compressed data can be chosen arbitrarily, with the possibly increasing penalty of the storage overhead of the dimension tables.

We demonstrate the strength of our architecture for data mining tasks on Figure 3.4 (the experiment extends [Sidl 05b] and is based on the experiments of

Table 3.2: Reference queries

Query	SQL
Q1	select sum(PAGE_ID) from FACT_PAGE_IMPRESSION where DATE_KEY between 20060101 and 20060131
Q2	select count(*) from FACT_PAGE_IMPRESSION where DATE_KEY between 20060101 and 20060131 and HTTP_STATUS_CODE = 200
Q3	select count(distinct USER_ID) from FACT_PAGE_IMPRESSION where DATE_KEY between 20060116 and 20060122

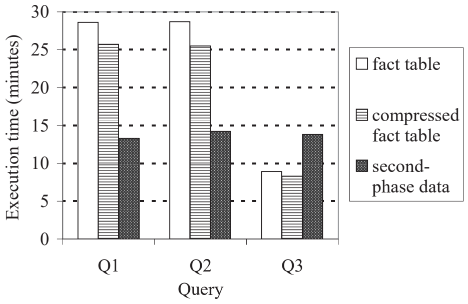


Figure 3.3: Execution times for the reference queries

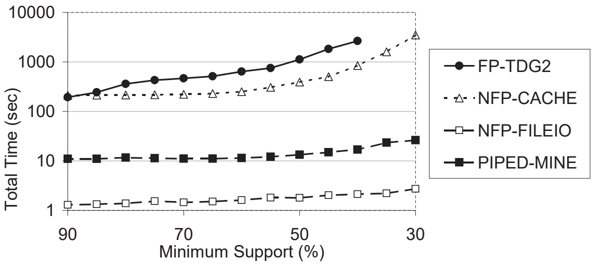


Figure 3.4: Frequent itemset mining on the ACCIDENTS dataset

Section 4.2). The four depicted algorithm incorporate four different philosophy for solving the task of frequent itemset mining in databases.

FP-TDG2 uses only relational database facilities to compute frequent itemsets, namely tables, indexes and SQL operations.

NFP-CACHE is a tiny cache-mine system, reading the input data from the database, caching it on the local filesystem, computing the results by an efficient C++ standalone application (nonord-fp [Freq]), and writing back the results to the database.

NFP-FILEIO serves as a baseline on the figure, representing the nonord-fp algorithm using efficient file I/O libraries, getting the input data from files and writing the output into files too, without any database connection.

PIPED-MINE stores the input data compressed on local disk, computes and writes the result to the database using our second-phase toolkit.

The experiments were were performed on a PC server with a 3 GHz Intel Pentium processor, 2 GB memory, RAID 5 with IDE disks, Debian Linux operating system and Oracle9i Release 2 DBMS (the same as in Section 4.2.4). The dataset is a public dataset, having 340,183 transactions and 468 different items [Freq].

The SQL-based FP-TDG2 performs poor mainly because the structures and operations provided by the database are not suitable enough for implementations of data mining algorithms. The main part of the execution times of NFP-CACHE comes from reading the input data from the database and writing back the results. As the result set growths, the execution times are becoming therefore also larger. PIPED-MINE employs a similar implementation of frequent itemset mining algorithm as NFP algorithms. The time saving comes partly from using the compressed data as input, and partly from using a faster database connection API.

Figure 3.5 illustrates the execution times of the main regular jobs in our experimental system. We see the DBMS daily aggregation as the main bottleneck; another reason for using our architecture is that monthly or weekly aggregates cannot even be computed in tolerable time. We observe that the second phase offers a good basis computing these aggregates as well as for running data mining tasks.

3.4 Conclusions

A new architecture type is developed and described that can integrate relational data warehouses and colum-oriented storages in a cost-effective way, enabling

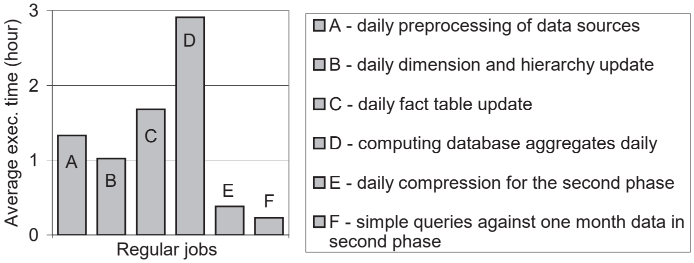


Figure 3.5: Comparing average execution times of regular weblog processing jobs

long-term storage and data mining.

The practical applicability of the new architecture is demonstrated by a prototype system, where a commercial database management system and a data mining framework of our own were integrated. This prototype system is then used for web analytics, where the results were validated on weblog data of the Hungarian [origo] web portal. In 2006, the portal had a size of over 700 thousand pages and received over 7 million page hits on a typical workday, producing 35 GB of raw server logs, thus overrunning the storage space capabilities of typical commercial systems. Despite of large cost allocation, several similar traditional data warehouse technology based web analytics projects failed. Our prototype two-phase web analytics data warehouse solved the same business problem in a cost-effective way, and worked for years with minimal maintenance.

3.5 My Contribution

Two research papers were published as a joint work with András A. Benczúr, Károly Csalogány, András Lukács, Balázs Rácz, Máté Uher and László Végh. The first [Benc 05] concentrates on applying data mining methods on clickstream data. Base ideas of our two-phase architecture are presented. In [Racz 07] we focus on the architecture for the same clickstream analysis task; our data mining framework (the second phase) is also described in detail.

My contribution in both papers consist of designing the general architecture scheme, drawing up and solving the integration problem of the two components, finally implementing and operating the prototype web analytics data warehouse.

Chapter 4

Frequent Itemset Mining

Frequent itemset mining (FIM) is a classic and central data mining task. FIM is a base to solve several further tasks, including association rule, frequent sequence or subgraph mining.

While algorithms for FIM were studied exhaustively (see e.g. [Freq]), much fewer results are known about FIM algorithms implemented inside relational database management systems. Next we study the relationship between FIM algorithms and classic relational databases, and provide efficient coupled tightly coupled SQL-based FIM algorithms.

Comparing the SQL-based implementations to the stand-alone FIM algorithms one can notice that the second class contains the very well performing pattern-growth algorithms [Han 00, Pei 01], while the idea of pattern-growth is poorly represented among the available SQL-based FIM algorithms [Shan 04]. Next we try to fill the gap by suggesting a new pattern-growth FIM algorithm tightly coupled to relational database management systems. The main result is an efficient adoption of the sophisticated FP-growth algorithm to relational databases. We expect that our algorithms process the data inside the database.

4.1 Problem Formulation

Let us consider the set of *items* $I = \{item_1, item_2, \dots item_m\}$. Let a *database* be $D \subseteq P(I)$, and let the *baskets* of items be the elements of D. The *support* of an *itemset* $A \subset I$ is the number of baskets that have all of the items from A. We call an itemset *frequent* if the support of the itemset is greater than a given threshold. **Frequent itemset mining** (FIM) is the task of finding all frequent itemsets.

Association rules are binary relations between itemsets. Let the ordered pair of two disjoint itemset be an **association rule**, denoted as $A \to B$. Let the support of the rule $A \to B$ be the support of $A \cup B$, the number of baskets containing $A \cup B$. The confidence of this rule is defined by the ratio of the support of the set $A \cup B$ to the support of the set A. The aim of association rule mining is to find all the rules that have a support and confidence greater than or equal to some previously given thresholds.

To solve the FIM problem, one can observe that frequent itemsets satisfy the **antimonotonicity property** (or Apriori principle): For a subset of an itemset, the support of the subset is less or equal to the support of the original itemset. This property is the base of the multi-pass algorithm called *Apriori* [Agra 94]. Further algorithms solving the FIM problem are based on pattern-growth [Han 00, Pei 01].

4.2 Data Mining inside Relational Databases

As described in Section 3.1, the level of integration between data mining and databases can vary on a scale from tight to none at all. SQL-based tightly coupled algorithms are considered significantly inferior in terms of running time compared to stand-alone implementations. Nevertheless there are advantages of tightly coupled data mining. Since in practice data appears mostly in data warehouses and other databases, in a tight coupled architecture no additional data mining system is needed. DBMSs are mature technologies that can facilitate data mining to become online, robust, scalable and concurrent.

Next we discuss related work, study SQL-92-based FIM algorithms, both Apriori and pattern-growth versions, and present a new efficient tightly coupled FIM algorithm.

4.2.1 Related Work

The first attempt to the particular problem of integrated frequent itemset mining was the SETM algorithm [Hout 95], expressed as SQL queries working on relational tables. The Apriori algorithm [Agra 94] opened up new prospects for FIM. The database-coupled variations of the Apriori algorithm were carefully examined in [Sara 98]. The SQL-92 based implementations were too slow, but the SQL implementations enhanced with object-relational extensions (SQL-OR) performed acceptable. The so-called Cache-Mine implementation had the best overall performance, where the database-independent mining algorithm cached the relevant data in a local disk cache. The optimization of the key operation, the join queries was studied in [Thom 99], and a new SQL-92-based method, Set-oriented Apriori was introduced. Further performance evaluations on commercial RDBMS can be found in [Yosh 00], evaluations of the SQL-OR option in [Mish 03]. An interesting SQL-92 algorithm based on universal quantification is discussed in [Rant 03] and [Rant 04].

Since the introduction of the FP-growth method [Han 00], a few attempts were made to implement pattern-growth methods inside the RDBMS [Shan 04]. [Bara 05] presents a novel, FP-tree-based indexing method, who provides a complete and compact representation of the dataset for frequent itemset mining, efficiently cooperating with the relational database. [Grah 04] deals with database-independent frequent itemset mining, but supposing that secondary memory have to be used.

Among the problems of data mining in DBMS, FIM is investigated most intensively. Among other classical data mining tasks, we find tightly coupled versions of decision tree classifiers [Satt 01, Bent 02]. A general tightly coupled data mining architecture is introduced in [Meo 98].

4.2.2 SQL-based Apriori Algorithms

The Apriori algorithm is based on the so-called Apriori principle: the n-element candidate sets C_n can be produced from the $(n-1)$-element sets of frequent itemsets F_{n-1}. Since the candidates have to be filtered to produce the frequent itemsets $F_n \subseteq C_n$, the algorithm iterates two basic steps. In the n-th iteration, C_n, the candidates for frequent itemsets having size n are generated. Then the support of the candidates are counted by reading all input data. The process iterates until the candidate itemset becomes empty.

Next we briefly discuss SQL-92 implementations. Typically, the input has the $(transaction\_id, item)$ schema as the number of items per transaction varies. This model fits the star schema design in relational data warehouses too. In the algorithm, except for the trivial C_1, the implementations materialize all C_n and F_n but they may differ in data representation. Two basic variations to represent these sets are the horizontal $(item_1, item_2, ...item_n)$ and the vertical $(set\_id, item)$ approaches. The horizontal approach has the disadvantage that the maximal count of table attributes is limited.

SQL Apriori implementations also differ in the SQL commands for candidate generation and support counting. Since support counting is the most time consuming part, most algorithms use the same efficient candidate generation operation, a k-way join generating C_n from F_{n-1}. The support count commands may be K-Way-Join, Subquery or 2-Way-Join and may utilize join operations or rely on `group by` computations, for example Two-Group-Bys [Sara 98].

The basic support counter operation of the K-Way-Join algorithm joins the data table n times in the nth iteration step.

```
insert into $F_n$ (
   select   $item_1$ ...  $item_n$ $count(*)$
   from     $C_n$, $F_{n-1}$ as $I_1$, ...  $F_{n-1}$ as $I_n$
   where    $I_1.item < C_n.item_1$ and ...  and
            $I_n.item < C_n.item_n$ and
            $I_1.tid = I_2.tid$ and ...  and $I_{n-1}.tid = I_n.tid$
   group by $item_1$, ...  $item_n$
   having   $count(*) \geq minsup$ )
```

Subquery is an optimization of the K-Way-Join, which makes use of the common prefixes between the itemsets in the candidate set. We developed different versions of Subquery to apply the divide-and-conquer idea of [Sava 95]: If we divide the database into distinct partitions, then an itemset can only be frequent, if it is frequent on at least one partition. It is possible therefore to partition the input table, find the frequent itemsets over the partitions, then test all partition-wise valid frequent itemsets over the whole input table. While in our experiments, partitioning remains inefficient compared to our methods, it can be used to mine data stored on multiple databases [Kona 04].

4.2.3 Pattern-growth Methods

Pattern-growth methods, first published in [Han 00], represent the database in a compact data structure called **Frequent-Pattern-tree** (FP-tree). By using the FP-tree structure, we may avoid both repeated database scan and large candidate set generation. An FP-tree stores items having support greater than the minimum support in a tree. Given an ordering of the items, transactions are represented as paths from the root node, and share the same upper path if their first few frequent items are the same. The FP-tree is searched recursively to find the frequent itemsets with the FP-growth method.

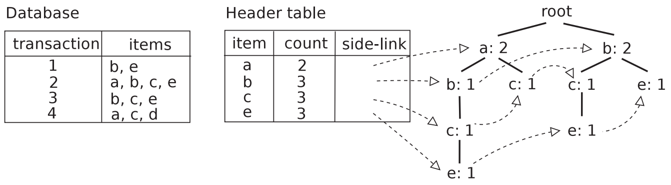
Figure 4.1 shows an FP-tree built for an example dataset with minimum support 2. All nodes are labeled by an item, have a count value and a sidelink to their siblings. The count value refers to the support of the path's itemset from the root to the given node. An additional header table stores the initial sidelink and the total item count for the items.

FP-tree Construction

A table with schema

$$node : (node\_id,\ parent\_id,\ item,\ count,\ sidelink)$$

represents the FP-tree in a natural fashion where sidelink is a Boolean attribute showing whether a node is part of the processed subtree or not in a particular



Database

transaction	items
1	b, e
2	a, b, c, e
3	b, c, e
4	a, c, d

Header table

item	count	side-link
a	2	
b	3	
c	3	
e	3	

Figure 4.1: FP-tree for a given database, built with minimum support threshold 2

state of processing. Parent attributes are NULL on the first level of the tree. An alternative representation can be found in [Shan 04], where, instead of the parent reference, a special and not fully discussed path attribute is used for all nodes.

The FP-tree can be built by reading the database once. For each transaction, we either insert a new path into the tree if the itemset of its frequent items has not been represented yet, or else we increase the counts. This method implemented as a series of SQL queries is, however, not efficient and hard to optimize, since we access the *node* table individually for all items. Instead, we build the FP-tree level by level, inserting all nodes on a particular level of the tree batched in one common SQL command.

The first version we present uses the subset of the original input table containing only the transaction parts having frequent items, along with a table containing (*node_id*, *item*) elements, representing the prefix we have processed. We delete the processed rows from the filtered input table, get the next item for the transactions by a minimum search, and insert new rows in *node*. Supposing that input table is $tdb\_filtered : (tid, item)$, the prefix table is $prefix : (tid, node\_id)$ and $node\_seq.nextval$ is used to generate the unique identifiers, the key step is as follows.

```
insert into node (
       select     node_seq.nextval, min.minitem,
                  prefix.node_id, count(min.tid)
       from       ( select    tid, min(item) minitem
                    from      tdb_filtered
                    group by  tid ) min, prefix
       where      min.tid = prefix.tid
       group by   min.item, prefix.node_id )
```

Our second version uses an analytic function called *dense rank* to produce a sorted and filtered version of the input table. We create groups with the help of this function, according to the tid attribute, and rank the items in the group based on the given ordering (supposing that tdb: $(tid, item)$ is the input table).

```
select tid, item, dense_rank() over ( partition by tid
       order by item ) rank
from   tdb
```

The filtered input table is $tdb\_filtered : (tid, item, rank)$ in this case. Building $node$ is similar to the previous version, but we can eliminate the minimum search and the deletion phase by referring to all levels by the rank value.

Items in the input table are represented by identifiers, therefore a natural ordering exists, but this ordering is not suitable for building the FP-tree. We have to use an additional table for items, in which they are ordered according to exactly one new identifier. The new identifiers are given so that the natural ordering of them will be the same as the descending ordering of the original items based on the count of transactions they appear in. This ordering promises optimal tree structure in the sense of compactness. This step can be solved by a simple sorting query, and the results can be used to initially fill up the header table described below.

FP-tree Evaluation

The basic FP-tree evaluation is not efficient enough with SQL operations. Instead we use a method similar to the top-down FP-growth described in [Wang 02], who enables finding all frequent itemsets without materializing conditional subtrees.

The core of the algorithm is a recursive procedure with SQL operations using auxiliary tables. The $header : (header\_id, item, count)$ table stores count information for items coming up in stages of the recursion, and also serves as a recursion heap. The identifiers in the header table are analogous to the separate header tables in the original FP-growth. All the itemsets ending up with a given $\bar{x}$ item sequence are considered in a recursion step. Another table $header\_postfix : (header\_id, item)$ stores these $\bar{x}$ postfixes for the header identifiers. Using these tables the "mine" algorithm (Algorithm 1) recursively produces all frequent itemsets above a given $minsup$ minimal support value. It is called first after the FP-tree creation phase as mine(0), when $header$ is already filled with frequent items and their counts, and rows referring to the initial 0 $header\_id$.

We implemented the steps of the algorithms as SQL queries, with the help of auxiliary tables. Frequent sets are written to the $result : (set\_id, item)$ table, absolute support values of itemsets to $result\_support : (set\_id, support)$.

Algorithm 1 FP-tree-based DB FIM

input parameter: h_id

```
1:  for $h\_rec$ in ( select $header\_id$, $item$, $count$ from $header$
    where $header\_id$ = $h\_id$ ) do
2:    if $h\_rec.count \geq minsup$ then
3:      output long pattern: ($h\_rec.item$, postfix) using $header\_postfix$ ;
4:      $new\_header \leftarrow$ generate new header id ;
5:      for all $n$ node from $node$ located on paths upwards from $h\_rec.item$-s,
        having $sidelink = Y$ do
6:        $n.count \leftarrow$ sum of counts of leafs ;
7:        $n.sidelink \leftarrow Y$ ;
8:        if ($new\_header$, $n.item$) exists in $header$ then
9:          add $n.count$ to header row identified by ($new\_header$, $n.item$) ;
10:       else
11:         insert $(new\_header, n.item, n.count)$ into $header$;
12:     for all $n$ node from $node$ not located on paths upwards from
        $h\_rec.item$-s, having $sidelink = Y$ and $item < h\_rec.item$ do
13:       $n.sidelink \leftarrow N$ ;
14:     mine($new\_header$) ;
```

The main observation that motivated the top-down FP-growth method is that if we process the tree in a top-down fashion, then the counts of the nodes above the actual leaf are no longer needed, therefore they can be used for counting. We use further temporary tables for the purpose of climbing up the paths and setting sidelinks and counts (rows 5.-12.). Table $path$: $(node\_id, count)$ stores the nodes found on the paths with the actual $count$ value. We climb up the paths level by level, accumulating the counts of the leaves. The required information (original node, actual node, count value of the original node) for these steps are stored in another temporary table. This step can be solved by the use of a recursive query as well (assuming the syntax of Oracle):

```
select            node_id
from              node
start with        node_id = (actual node)
connect by prior  parent_id = node_id.
```

Processing the nodes on the paths leaf by leaf with the use of a recursive query instead of level by level, however, was less effective according to our early test results.

Optimization

It became clear after implementing the first versions of the algorithm that the main costs arise from the $node$ table accesses, especially from updates (steps 6, 7 and 12). These accesses refer to more and more node by the end of the processing, when we process nodes near to the leaves. We can optimize the updates, for example updating only those sidelinks of the nodes which don't have the right value yet, but after all without the use of indexes these steps require full scans of the $node$ table, and this costs mostly lots of block reads and writes. However, one of the main reasons to use a relational databases is their efficient indexing capabilities; next, we briefly enumerate some indexing considerations.

The $node\_id$, $parent\_id$ and $item$ attribute values don't change after building the $node$ table. It is therefore profitable to use standard B-tree indexes on them, like $(item, node\_id)$ for searching $node\_id$-s by $item$, or $(node\_id, parent\_id)$ to find parent nodes efficiently. The $sidelink$ and $count$ values are changed frequently. We don't want to access the table by the $count$ attribute, but the use of some index on the $sidelink$ attribute may be profitable. We can use regular indexes, or, since the $sidelink$ attribute has only two distinct values, we can use bitmap index. We tested both the regular and bitmap version for $sidelink$ along with other index combinations. We refer hereafter the indexed version of the above described algorithm as FP-TDG.

We implemented several alternatives of FP-TDG. In our experiences, denormalizing the $node$ table turned out beneficial: the database schema

$$\begin{aligned} node &: (node\_id, parent\_id, item) \\ sidelink &: (node\_id) \\ count &: (node\_id, count) \end{aligned}$$

enables us to manage the frequently changed information apart from the permanent tree-structure information. In this case we store the binary “header” information as a set of $node\_id$. The “count” values for nodes are stored in a smaller and separate table. Instead of building separate indexes on these two tables we store them as B-trees with the help of the so-called “index-organized table” facility of the database server. We refer this version as FP-TDG2.

4.2.4 Experiments

Our experiments were performed on Oracle 9i Release 2 DBMS, installed on a server with a 3 GHz Intel Pentium processor, 2 GB memory, RAID 5 with IDE disks and Debian Linux operating system. Memory consumption of the database server was limited to 1 GB, because of other background services on the server. Redo logging was reduced for all tables, and parallel processing functions of the database were not enabled.

Since expressive power of SQL is not enough to express our algorithms, we used a programming environment. We generate sequences of SQL operations by PL/SQL, while main data processing and all data remains in the database. PL/SQL could be exchanged to any other programming environment, in which we can connect to the database server through some standard database API. The algorithms can be executed on an arbitrary client, because the main part of the data processing remains inside the database server. The client generates the adequate SQL statements only, which requires little computing and networking capacity.

We used public FIMI [Freq] datasets as test datasets (Table 4.1 contains the properties of the three selected datasets for demonstration).

We have chosen Subquery to compare our algorithm to, because — as suggested in [Sara 98] — Subquery had the best overall performance of the preexisting methods. (That is in fact opposed to [Rant 04], where K-Way-Join is superior in the category). We implemented our version with the so-called second-pass optimization: we don’t materialize the candidates of size two, we replace it with a 2-way join between frequent item tables of size one.

The other algorithm we have chosen for comparison is *nonordfp* [Freq], whith a fully database-independent C++ implementation. *Nonordfp* handles an

Table 4.1: Dataset properties

Dataset	# records (K)	# transactions	# items	Avg. num. of items per transaction
ACCIDENTS	11,500	340,183	468	33.8
BMS-WebView-1	149.6	59,602	497	2.5
BMS-WebView-2	358.3	77,512	3,340	4.6
KOSARAK	8,019	990,002	41,270	8.1
RETAIL	908.6	88,162	16,469	10.3
T10I4D100K	1010	100,000	870	10.1

FP-tree-like structure, and can efficiently evaluate it without materializing subtrees. The algorithm runs on the same machine as the database, but only connects to it to read out the input data and to write back the result through standard JDBC interface. *Nonordfp* caches the data in the filesystem for processing. The memory usage was not limited. We refer hereinafter this implementation as NFP-CACHE.

The main part of the total execution times of *nonordfp* came from reading and writing the database. The response time goes up only below low minimum support values, when the result set becomes large. The algorithm *nonordfp* outperforms the SQL-based methods for low minimum support, however as being in-memory algorithm, the input size is limited by the available memory.

Figure 4.2 shows execution times of our two methods for FP-tree construction. Figure 4.3 (left) shows execution times on different sized samples of the RETAIL database with the minimum support value of 0.5 %. Figures 4.3 (right), 4.4, 4.5 and 4.6 (left) compare the total execution times of our algorithms.

FP-TDG and FP-TDG2 mostly outperform Subquery, but in case of the generated dataset T10I4D100K they do not perform well. This dataset is rather sparse, and most FP-growth methods work less efficiently on sparse datasets. This can be seen here as well. The FP-tree becomes too large, it does not compress the database efficiently, and this causes a leap in the aggregated node-access times. On the other hand the sparsity of the database is advantageous for the join-based Apriori methods, when the size of the candidate sets shrinks fast.

We have also tested Subquery and FP-TDG2 in a real-life environment, over logs of the largest Hungarian web portal (www.origo.hu). The site had 7 million page views on a typical workday in the times of the experiment. That was processed by an experimental weblog mining architecture (see [Benc 05] and Section 3.2 for details). The preprocessed data is stored in an Oracle 9i database component of the architecture. The task is the identification of pages accessed together by a large fraction of the users on a given day. Execution times of the algorithms on a typical workday can be seen on Figure 4.6 (right side), where

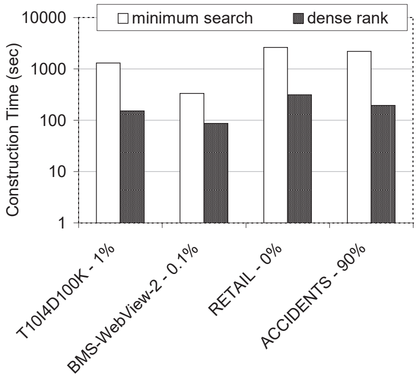


Figure 4.2: Constructing the FP-tree for selected databases and minimum support values

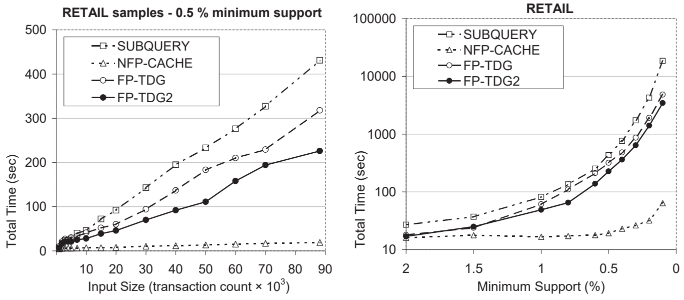


Figure 4.3: Execution times on the RETAIL dataset

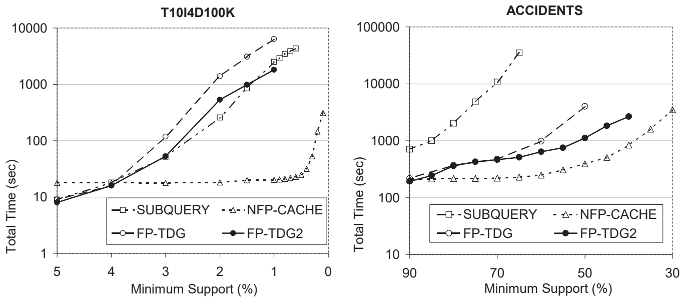


Figure 4.4: Execution times on the T10I4D100K and ACCIDENTS datasets

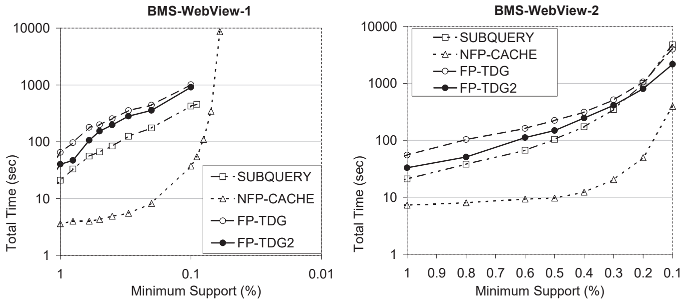


Figure 4.5: Execution times on the BMS-WebView datasets

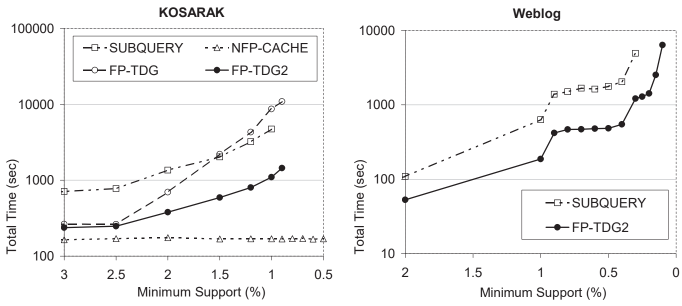


Figure 4.6: Execution times on weblog datasets

767,663 identified user accessed 57,911 different pages during the day, resulting in 2,395,146 page view records. The average number of page views per user was 3.12.

In the weblog mining architecture the trends and statistics can be analyzed by an analytical reporting framework. It can be reached through a webserver with dynamic web pages, connected to the database. Users can discover frequent sets of pages by extending the frequent sets one-by-one, starting with an empty, or with a directly given set of pages. The possible extensions can be chosen from a toplist. This simple method is suitable in our case, where we have 1 to 47 thousand frequent sets with a maximum size of 13 for the different minimum supports measured and appeared on Figure 4.6.

In this real-life application of the SQL-based FP-TDG2 we eliminated the need for a separate FIM system producing duplicated data. Frequent sets are produced by only the use of the common database facilities. The execution times are acceptable, they are comparable to the computation times of some complex statistical aggregations in the database.

Implementations of the algorithms and the used sample datasets can be downloaded from [Sidl 05a].

4.3 Conclusions

We developed and presented a pattern-growth FIM algorithm for relational database environments based on SQL operations. This algorithm efficiently utilizes the facilities provided by the database server, and fits the relational data model and specialties of the environment.

The algorithms represent a major step forward relative to similar existing algorithms. Applicability in real-world scenarios is demonstrated by a prototype system.

4.4 My Contributions

The results were published with András Lukács [Sidl 05b], whose main contributions were writing the introductory and related work sections.

Chapter 5

Entity Resolution

Entity Resolution (ER) is the process of identifying groups of records that refer to the same real-world entity. One of the first description of the record linkage problem appears in 1969, when Fellegi and Sunter [Fell 69] used a probabilistic model to describe this referral relation. Since then the process was described in many different contexts under many different names including duplicate detection, instance identification, heterogeneous join, merge/purge, reference reconciliation or object matching. There are lots of closely related topics too, for example clustering, similarity joins,string similarity, data cleaning, data warehousing, data integration, information integration etc.

In most cases, records are heterogeneous and erroneous and hence the mapping to hidden real-world entities is not straightforward. Structural and syntactic heterogeneity originates mostly from the heterogeneity of source systems, difference in data handling policies, standards, and finally from low data quality due to typos, missing values and other problems. ER can be therefore handled as a data cleaning task, occurring in data integration scenarios often.

Entity resolution is an actively researched area. To demonstrate this, if we look at one of the main database conferences of 2010, the conference on Very Large Databases, then there we can find approximately 5 research papers dealing directly with entity resolution out of approximately 90 papers, and at least other 10 with closely related topics.

The ER problem can be formulated in many different ways. Input and output can be a set of records with attributes, a set of XML documents or a graph. The algorithms can either produce exact results or probabilistic mappings. Matchings can be defined by exact rules, by similarities or by links between records. Results can be represented by record sets, by representative merged elements, or both. Training data or entity activity log can be present. The architecture we use to solve the problem can be distributed, can be a single database server, a standard standalone computer or, for another example, a data mining framework.

The main assumption in any cases is that we examine and group together observations (records) of hidden real-world entities. The main goal is to improve data quality, to give better representation of the real world. The selection of the data model and representation, the match criteria, the architecture, the algorithms and tools used are mostly orthogonal decisions. In the following we examine some of these possible design decisions, while trying to scale up the ER process. We present new algorithms for some of the available configurations, which are able to improve existing solutions.

5.1 Motivation

Entity resolution appears in a wide range of applications. Author name resolution in bibliographic databases forms the first and ever since popular task in ER research where the goal is to group together different occurrences of authors with names written in several ways. Name, institute and a few other personal attributes as well as the co-authorship relations can be used for ER.

Various industrial ER scenarios for clients, products and others share common characteristics. All these areas would profit from an efficient solution of ER problems. Search engines could identify and group together web pages dealing with the same entity, such as a person or a product (see [IGlu 11] for a semantic web example). Web services could identify duplicated registrations. Stores or auction web sites could group together different items of products.

5.1.1 Client Databases

Companies typically face the entity resolution problem when building a client database, or manage client master data. Clients may appear multiple times in multiple source systems, e.g. a record for a contract, another for a purchase. As another example, the same person may appear in several marketing databases obtained by different means. ER is the key step in producing sound and clean client master data.

Client records may consist of attributes, both of persons (birth data, tax and social security numbers, postal address, etc.) and of organizations (client ID, contract number). Attribute values are often missing or erroneous, and some attributes change in time (name, postal address).

By resolving the record set, simple but fundamental as well as more complex questions can be answered: How many clients we actually have? Can a given client be addressed in a marketing campaign, or we just made an offer a few days ago? Does a new client have ever contacted, or had any transaction with our company? Existing ER algorithms are however often not scalable enough, resolving a larger record set may require days on conventional architectures.

Insurance Case Study

Our motivating application is client data integration of several insurance source systems at the AEGON Hungary Insurance Ltd.[1] The ER problem comes into sight during the construction of a client data mart over legacy systems that remained independent of each other for operational reasons during mergers and ownership changes.

The first step of data integration consists of cleaning and integrating data into a unified schema by massive ETL tools. Then a slowly-changing versioned client dimension is built up including all available attributes. The goal of EM is to identify clients with multiple dimension elements and automatically clean the client dimension. Additional fact tables store relations between clients, and between additional dimensions such as contract ID, postal address etc. as seen in Fig. 5.1 (beware, this is not a proper ER diagram, only a sketch of the base ideas). Despite the exhaustive pre-processing, there remained several duplicated dimension elements, caused by differences in attributes of the source systems, different data recording and storage policies as well as time-varying attributes.

Domain experts can define exact rules on customer attributes for constructing match and merge functions of client records. Merging of records is also possible: experts can formulate rules to construct a simple record containing the most valuable information of the underlying matching records. An automated ER process is expected, therefore exact results are preferred. Approximate result based on similarities and links between records can also be produced, but these results have to be reviewed by experts and cannot be used automatically to maintain the dimension.

We can match clients based on attribute similarities (like Mr. Smith and Mrs. Smith), or based on links (for example common address or phone number). Attribute values can change in time. People contained in the data mart have personal attributes and company dependent attributes such as identification numbers in the source portfolios. We deal with simple permanent attributes closely related to the client as name, birth name, birth date, sex, tax and social security number. We call a combination of these attributes an identity. People naturally have multiple identities: they are asked for different set of attributes in different roles (as a client for example compared to an insurance agent or a third-party beneficiary of an insurance). Note that handling identities as separate

[1] The AEGON Hungary has been a member of the AEGON Group since 1992, one of the world's largest life insurance and pension groups, and a strong provider of investment products.

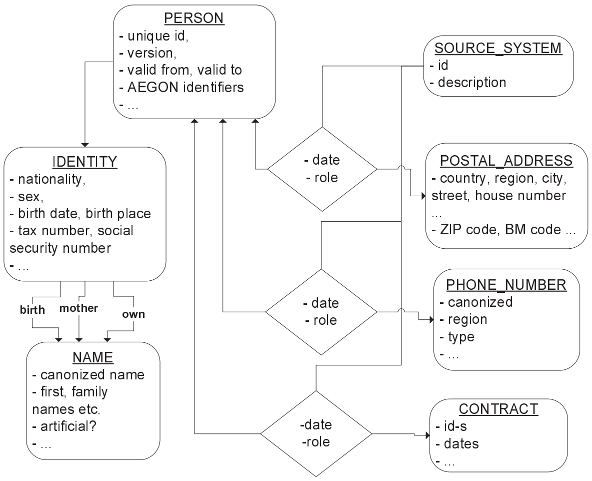


Figure 5.1: Simplified data model of the insurance client data mart

concepts enables the identification of not only the duplicated client records, but also duplicated records of people in arbitrary roles. Agents and clients for example have a lot of distinct attributes, but the essential entity attributes are always present.

5.1.2 Practical Observations and Assumptions

We observed some simple properties of practical ER problems while solving and studying ER scenarios. Next we describe some assumptions that we think are useful for most of the ER tasks in practice.

Uncertain environment. ER tasks are always interpreted in an uncertain environment. The main ER problem arises from the fact that observations of real world entities are erroneous and vague in some sense. Therefore, deterministic entity resolution is not flexible enough; probabilistic models for these uncertain statements are preferred.

Human supervision. Human supervision is hardly avoidable. Exact matching rules can be defined, but the probabilistic nature of the problem and the large amount of expert knowledge to be formulated makes the construction of these rules hard. Therefore, active learning methods are preferred where borderline matching questions are decided by an expert.

Set-based model and matching. Traditional record-record matching oversimplifies real world problems in most cases. The real nature of the problem is closer to matches between sets of records. Representative records can be misleading and hard to construct: for example, it is hard to decide between two birth dates of the same rank. Deciding matches is sometimes also hard without referring to all the feature values, with only a representative value. We think that ER models representing an entity with record sets are favorable.

Type I and II errors. There can be a huge difference between the consequences of match errors. In the client example, finding inappropriate matchings (false negatives) can cause legal troubles with high costs, but not finding a possible matching (false positives) is more tolerable. The model used should adopt to this asymmetry.

A-priori knowledge. Besides the plain database of entity records, a-priori knowledge may be available to improve ER quality. For example, we may have information on distributions of attribute values or cardinalities of entity groups. Use of similarity measures should consider statistical properties of the given entity set: two records having the name "John Smith" match with lower probability than two having "Csaba Sidló". The external information should be incorporated into the ER model or represented as extra attributes.

Entity hierarchies. Real-world entities form natural hierarchies. For example, ER procedures may be applied in a client scenario for the identification of households or company hierarchies as the key entities of interest. In this case we would like to identify entities not explicitly present in the source data. As another example, ER is applicable for postal addresses that are attributes of clients, while clients themselves are subject to resolution.

Overlapping entities. Entities may have vaguely defined boundaries in certain practical cases. For example households may overlap as people may belong to more than one households, or move between households. Producing a definite non-overlapping group of personal records as households is less useful than producing overlapping groups. Overlapping groups of records as entities are therefore preferable in some cases.

5.2 Related Work

One of the first descriptions of record linkage appears in the influential paper of Fellegi and Sunter [Fell 69] in 1969, describing a probabilistic model. Since then, entity resolution problems have been studied in many different disciplines and names. For overview, in [Elma 07] a survey is given on duplicate record detection, describing supervised, unsupervised and active learning, and summarizing statistical and machine learning solutions based on various text similarity and matching measures. Recently the book [Talb 10] introduces key models, methods and new trends from a more practical point of view.

Traditional deduplication approach uses similarity measures for attributes, and learns when two records can be resolved to the same entity. A survey of string similarity functions can be found in [Elma 07], along with a survey of basic duplicate detection algorithms. In [Grav 01] a nice solution is presented for implementing string-similarity joins using q-grams in an RDBMS environment.

ER can be handled as a supervised learning problem, if training data is present. We can apply data mining classification methods, for example Bayes methods [Han 05, Fell 69], decision trees [McCa 95] or SVM [Bile 03, Chri 08]. Unsupervised learning methods such as latent Dirichlet allocation [Bhat 06a] or clustering methods can also be used, if there is no training data. An interesting approach lying between the previous two is called active learning: when a small set of training data is given, the algorithm decides the new elements it could use the best to extend the training set ([Sara 02]). An automated training data selection method is described in [Kopc 08].

ER is formalized many times as generating clusters of linked records. In the citation database scenario, with the goal of identifying authors, we do not really have author attributes other than their names. We can however link these records by joint papers. This way ER can be seen as a special problem of link-mining; a survey containing link based entity resolution can be found in [Geto 05]. The approach is called relational ER, based on the relations between records, or collective ER, because we would like to resolve records based on the link graph as a whole.

Entity resolution as a hypergraph clustering problem can be found in [Bhat 07], under the name of relational clustering. Input data is handled as a reference graph, with nodes as entity records and edges as links between these nodes. The goal of the resolution process is to produce a resolved entity graph, where nodes are entity instances that hold entity records together. Clustering is also suggested [Hern 98]; however, general clustering methods are usually designed for less and larger clusters as records of entitities in ER.

A seminal paper, [Benj 09] (published first in 2005) introduces generic entity resolution with black-box match and merge functions, where resolution means the closure of the original entity set according to these functions. Simple feature indexes are also used. Generic entity resolution forms mostly the base of our following methods and algorithms. The model and the algorithms are extended in [Mene 06] for handling approximate results as records with confidences. [Benj 07] adapts the algorithms to a distributed environment. Our generic ER algorithms for relational databases of Section 5.4.2 was published in [Sidl 09].

Other interesting approaches to ER includes utilizing aggregate constraints [Chau 07], or giving methods for query time ER [Bhat 06b]. In [Bhat 08] a unified model is suggested for entity identification and document categorization. [Wick 08] widens the coreference problem with schema matching and canonicalization, and provides a unified model. The role of cross-field dependencies is described in detail in [Hall 08].

Recently, several new ER results were published. A new approach can be found in [Yako 10]: entity behavior is recorded as transactional log. Common patterns of these transactions are used to identify similar or identical entities. Measuring the quality of entity resolution results is a crucial problem, [Mene 10] deals with possible quality metrics. [Whan 09] enhances core ER algorithms by combining the results of different blocking strategies. [Guo 10] exploits the role of constraints when finding duplicates. [Whan 10] deals with the effect of match/merge rule evolution, and gives methods to preserve results when rules change. [Chri 09b] builds special inverted indexes to speed up ER with blocking. A survey of indexing techniques available for deduplication is provided in [Chri 11], including blocking, sorted neighborhood, Q-grams and canopies.

Entity resolution frameworks are developed, like SERF, MTB, DDUpe and MARLIN (see [Kopc 10a] for a survey). A practical comparison of ER approaches can be found in [Kopc 10b] using the FEVER framework.

The Febrl framework also provides parallelization [Chri 04]. Other parallel algorithms are presented in [Kim 07], tested on a few thousands of records. More recently [Kirs 10] introduces parallel matching and a distributed infrastructure, using similarity-based matchers.

5.3 ER Models and Formulations

The problem of entity resolution can be formulated in several ways, with applicability depending on the particularities of the given practical problem. Next we examine some of the potential models and formulations developed for the algorithms following in Section 5.4, 5.5 and 5.6.

Section 5.3.1 gives an overview on the generic entity resolution model of [Benj 09] as the starting point of our work. This model assumes exact results, pairwise matches between records, representative elements and black-box match and merge functions.

Generic ER model is modified in Section 5.3.2 to enable efficient algorithms on relational databases. Match functions on (record set, record) pairs and merge functions on record sets are used.

A more general model, with entities as sets of records, is described in Section 5.3.3. Here merging is a union of these sets, and no representative elements are used. This model is used for index-based and distributed ER algorithms.

Practical match function classes and useful constructs are defined in Section 5.3.4 and 5.3.5. A probabilistic model is briefly introduced in 5.3.6, capable of extending our algorithms to incorporate probabilistic, similarity-based matching. This model is however not used in this thesis, only demonstrates a possible way to formulate probabilistic matching.

5.3.1 Generic ER Model

The starting point of our work is the Generic ER model of [Benj 09]. They suppose black-box match and merge functions and exact outcome, with representative merged records for entities.

Let us assume a set of records $I = \{r_1, r_2, ...r_n\} \subset R$, called instance ($R$ is a domain of records). Note that records are arbitrary elements, and do not necessary share the same structure, or even have structure.

A **match function** is an $R \times R \rightarrow \{true, false\}$ Boolean function, denoted as $r_1 \sim r_2$ and $r_1 \not\sim r_2$. The $merge : R \times R \rightarrow R$ partial function is defined on matching pairs of records, denoted as $\langle r_1, r_2 \rangle$ (for every $r_1 \sim r_2$).

We also would like to characterize which record describes an entity better. We suppose a partial ordering on records, called **domination**. We use $r_1 \preceq r_2$ (for $r_1 \sim r_2$ records), if r_2 describes the underlying entity “better”; for example contains more information, or newer data.

Given the constructs above, we can define the **generic ER problem** as follows. Given an instance I, let the **merge closure** of I be a set of records that can be reached by recursively adding new elements to I by merging matching records. $ER(I)$ denotes the **resolved entity set**: Let $ER(I)$ be the smallest subset of the merge closure that could only be extended by records that are dominated by other records within $ER(I)$.

Note that up to this point we have no restrictions on match, merge functions and on domination. $ER(I)$ is a well-defined, but not necessary a finite set (consider the example, where the merge function concatenates string records, and $r_1 \preceq r_2$ means r_2 is longer than r_1). However, if we restrict the class of merge and match functions, then we can make $ER(I)$ always finite and independent of the order in which records are processed.

Assume the following (**ICAR**) properties of the match and merge functions:

- idempotence: $\forall\, r : r \sim r$ and $\langle r, r\rangle = r$,
- commutativity:
  $\forall\, r_1, r_2 : r_1 \sim r_2 \Leftrightarrow r_2 \sim r_1$, and $r_1 \sim r_2 \Rightarrow \langle r_1, r_2\rangle = \langle r_2, r_1\rangle$,
- associativity: $\forall\, r_1, r_2, r_3$ where $\langle r_1, \langle r_2, r_3\rangle\rangle$ and $\langle\langle r_1, r_2\rangle, r_3\rangle$ exists:
  $\langle r_1, \langle r_2, r_3\rangle\rangle = \langle\langle r_1, r_2\rangle, r_3\rangle$,
- representativity: $\forall\, r_4, r_1 \sim r_4$: $r_3 = \langle r_1, r_2\rangle \Rightarrow r_3 \sim r_4$.

If we use functions corresponding to the properties above, then we can use a natural domination partial ordering called **merge domination**: r_1 is merge dominated by r_2, if $r_1 \sim r_2$ and $\langle r_1, r_2\rangle = r_2$. Merge domination enables dropping merged elements during processing promptly after merge steps, keeping only merged elements.

ICAR properties and merge domination reduce the computational complexity of computing $ER(I)$. Although these definitions seem natural, we can observe cases, where the expectations do not meet the properties. In such cases we have to decide, if we solve the computationally harder problem, or we define new functions satisfying ICAR properties. By the use of merge domination we also face the same problem, as we will see later. However, in most cases functions according to ICAR and merge domination can be chosen to meet the user requirements.

5.3.2 Relational Generic ER Model

Next we modify the general ER model of Section 5.3.1 and add further requirements to enable efficient RDBMS-based implementations. First of all, the general ER model is too general for RDBMS-based implementations: we would only like to deal with uniform relational instances. We expect instances to meet the relational data model assumptions, and handle them as tables. Let $A_1, A_2 ... A_n$ be attributes, and let a **relational instance** be

$$I_r \subseteq \times_{i=1}^{n} DOM(A_i) = R_r.$$

The result of the resolution process will also be a relational instance. The concept of relational instances is less general than the original, but still practical and flexible. We are going to use such instances, and tuples (records) of these instances, denoted as $t \in I_r$.

RDBMS allows us to carry out batched operations on relations efficiently. Next we re-define match and merge to fit the relational environment better. Let the **relational match function** be

$$match_r : R_r \times 2^{R_r} \to 2^{R_r},$$

where 2^{R_r} is the power set of R_r, the set of I_r instances. The $match_r$ function compares a single record to an instance. Let the **relational merge function** be the

$$merge_r : 2^{R_r} \to R_r$$

partial function that is defined on instances, whose tuples match a single arbitrary tuple. The **relational merge closure** of an I_r relational instance is then defined as the smallest I'_r subset of I_r that satisfies

$$\forall\, S \subseteq I'_r, \forall\, t \in I'_r :\; merge_r(match_r(t, S)) \subseteq I'_r.$$

Applying merges on the closure does not lead us out of the closure. The definition of domination stays the same as by the general model. The **relational entity resolution** of an I_r instance, denoted as $RER(I_r)$, is then defined as the smallest subset of the relational merge closure that does not contain dominated records.

We can derive the semantics of the new functions defined on tuple sets from the pairwise functions: the new match function should produce the set of all matching tuples of I_r. However, pairwise merge semantics cannot always be easily translated to the new form. If we deal with ICAR pairwise functions, the semantics of the corresponding set-styled merge can be understood as applying pairwise merges in some arbitrary order to the original tuple. We can assume that $match_r$ and $merge_r$ are derived from pairwise functions having the ICAR properties the following way:

$$match_r(t, I_r) = \{t' \in I_r | t \sim t'\}$$

$$merge_r(I_r) = t \in R, \text{ where } I_r = \{r_1, \ldots r_n\}, t = \langle \ldots \langle r_1, r_2, \rangle r_3 \rangle \ldots r_n \rangle.$$

We can use the **merge domination** for relational instances if match and merge functions can be derived from ICAR pairwise functions.

Now, instead of derived $merge_r$, we define a more general function class. We consider the relational match and merge functions, only if $match_r$ can be derived from a pairwise function: $match_r(t, I_r) = \{t' \in I_r | t \sim t'\}$, and for all t, t' tuples and $I_1, I_2 \subseteq I_r$ instances, properties

$$\begin{array}{rcl} t \sim t' & \Rightarrow & t' \sim t, \\ t & \sim & t, \\ t' = merge_r(match_r(t,\ I_r)) & \Rightarrow & merge_r(match_r(t,\ I_r \cup \{t'\})) = t', \\ \text{if exists: } merge_r(I_1 \cup I_2) & = & merge_r(merge_r(I_1) \cup merge_r(I_2)) \end{array} \tag{5.1}$$

hold (a sort of idempotency and associativity). The properties reduce the complexity of computing $RER(I_r)$, guarantee that $RER(I_r)$ is finite, and the construction does not depend on the order of operations. In practice most of the useful functions can be formulated to meet these criteria.

Strong Merge Domination

Merge domination is a useful construct for reducing the size of $RER(I)$, while retaining all the information in $RER(I)$. Yet, ICAR properties of pairwise functions are sometimes too strict in practice. Consider the next example: a match function of identities uses conditions based on a tax number equality sub-condition and a combined sub-condition of birth name, current name and birth date attributes. We would like to implement a merge function that collects the more accurate birth date, the longest name and one of the tax numbers if more tax numbers are present. If we collect and merge matching tuples of a given record, the merged tuple can be a new one that does not match the original one: we overwrite the matching attributes. We cannot express the semantics of the example with ICAR match and merge functions.

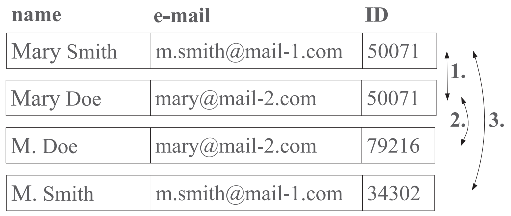
We define a new domination relation called **strong merge domination** that assumes only the properties of (5.1). The goal is to retain source records containing information needed to find merged records. Strong merge domination defines a partial ordering of a given instance I and for tuples t_1 and t_2 in I: t_2 is strong merge dominated by t_1 if

$$t_1 \sim t_2 \text{ and } merge_r(match_r(t_1, I \setminus \{t_2\})) = t_1.$$

Strong merge domination enables dropping source records that are similar to the merged record (but not all source records).

5.3.3 Record Partitioning Model

Next we re-define the ER model to a more practical, record-set based version. The starting point is that entities of the real world are typically hidden and only indirect observations are recorded in a database. The task is basically not to



name	e-mail	ID
Mary Smith	m.smith@mail-1.com	50071
Mary Doe	mary@mail-2.com	50071
M. Doe	mary@mail-2.com	79216
M. Smith	m.smith@mail-1.com	34302

Figure 5.2: Example of records belonging to the same entity

merge, but to group these observations, records together. This intuition is formalized as follows.

Let a set of **records** be $R = \{r_1, r_2, ...r_m\}$, where each r_j is described by its k **attribute values** $a_{r_j1}, \ldots, a_{r_jk}$ such as ID, name, address etc. Some attribute values may be missing, e.g. we may not know the e-mail address of customer j, that we denote by $a_{r_j\ell} = \emptyset$.

The goal is to partition records according to the entities they belong: let $E = \{e_1, e_2, ...e_n\}$ be a set of **entities**, each e_i consisting of a subset of records $e_i \subseteq R$ such that the union of the entities covers all records, and no record belongs to more than one entity:

$$\cup_{i=1}^{n} e_i = R$$

$$r \in e_i \wedge r \notin e_j \Rightarrow i = j.$$

To give an example, consider records and attributes in a client database stating that “the name of the client is Mary Doe”. An example using person records is depicted in Figure 5.2, with Mary Doe already a customer before her marriage when her name changes to Mary Smith. The first two records have identical ID, representing a customer with name Mary Smith and maiden name Mary Doe. The remaining two records are variants of the name and maiden name, respectively. Both of her names will appear in one record as name and maiden name but duplicates for both versions may appear in the data.

An entity can have more than one attribute values $a_1, a_2, \ldots a_i$ of the same type, for example multiple names may exist for a real-world client. We represent such an entity e by a set of records $e = \{r_1, r_2, \ldots\ r_i\}$ such that there are records for each value, $a_{r_1\ell} = a_1$, $a_{r_2\ell} = a_2, \ldots a_{r_i\ell} = a_i$.

Use of merged, representative records as entities is a common practice. Merged records can be however misleading and hard to construct, therefore we omit use of representative merged records in this model. Deciding matches is sometimes impossible without referring to all the feature values, with only a representative value. We think that ER models representing an entity with record sets are favorable and more practical in common scenarios.

Let an **entity match relation** be $e \sim e'$. For such a relation, let the **entity resolution** of an entity set E be another set of entities $ER(E)$, where

$$\forall e_1, e_2 \in E, e_1 \sim e_2 \Rightarrow \exists\, e' \in ER(E) : e_1 \subseteq e' \wedge e_2 \subseteq e',$$
$$e \in E \Rightarrow \exists\, e' \in ER(E) : e \subseteq e'.$$

Thus, the entity resolution is a refined partitioning of the original entity record sets where no more separate but matching entities can be found.

5.3.4 Matching and Indexability

Match is a relation between entities depending on the application. For example, we may require attribute value identity, but the definition may involve distance functions over the attribute space as well.

In case of the generic ER model (Section 5.3.1) we match on record pairs, since entities are represented by a single record. In case of the relational generic ER (Section 5.3.2) the match is defined between a record and a record set, where the semantics may be reduced to record-pair matches on record pairs. In case of the partitioning model (Section 5.3.3) multiple records are put together forming entities, therefore we have to define matchings between record-sets.

Next we define and study some efficient match function classes.

Attribute-based Matching

The simplest way to define entity matching is tracing it back to attribute values: Two entities match, i.e. $e_1 \sim e_2$, if a matching attribute value is found in their records, $a_1 \sim a_2$. Formally, for the more general partitioning model, e_1 and e_2 are matched if there exists $r_1 \in e_1$ and $r_2 \in e_2$ and ℓ with $a_{r_1\ell} \sim a_{r_2\ell}$. For the generic models of Section 5.3.1 and 5.3.1 attribute-based matching becomes simpler, as entities are records.

We formulate a practical requirement for attribute matchings, called **attribute indexability**: A given attribute match relation implements attribute indexability, if attribute values matching to a given a can be enumerated and represented by a single value or a set of values.

Standard indexing methods including hashing or B-trees can be used when we represent attributes with a single value. In the case of multiple values, we may handle multiple representative values in the same way as we normally handle multiple attribute values. Examples of indexing methods working this way include n-grams or fingerprints of shingles [Brod 97], or multidimensional search structures such as R or KD-trees. An exhaustive list of such possibilities can for example be found in [Chri 09a, Table 9].

The requirement of entity match based on indexable attribute matching may seem overly restrictive at the first glance. However, in most of the previous work [Mong 97, Dong 05, Hern 98] (including R-swoosh [Benj 09], our starting point algorithm) an exact match function is assumed, satisfying the indexability requirement. Next we list alternatives from related results and show how they in fact reduce to our simple settings. Then we will primarily describe algorithms with attribute equality as a match function and elaborate on how more complex match functions can be implemented and whether the given algorithm imposes additional constraints.

A potential limitation is that our match function, by strict definition, does not allow fuzzy matching by confidence levels. We may however reach beyond this limitation by shifting the goal of our algorithms towards efficient *candidate generation*. We may rephrase our output as an efficient way to distribute the work of sophisticated match functions and identify potential matches in another, simpler round. Also, we may easily accommodate complex match criteria or even use our solutions to distribute machine learning.

Another limitation is that complex similarity of the values of a single attribute cannot be expressed. We argue that, since pairwise comparison of all attribute values is computationally infeasible, ER can efficiently solved only in the case when similarity indexes can be built for all attributes. This requirement gives no additional restriction if we use similarity search indexes. When using similarity-based or probabilistic features and match conditions, the indexes provide entities (records or sets of records) with similar attribute values beyond a given similarity threshold. Document Q-gram and TF-IDF indexes, min-hash fingerprint [Brod 97] for example have multiple efficient standalone implementations, for distributed environments as well. Similarity-based indexes as described for example in [Chri 09b] are also useful in this case, and are exhaustively investigated topics.

The constraint of indexability above is somewhat orthogonal to the ICAR properties (Section 5.3.1), introduced in [Benj 09]. ICAR is introduced for the same purpose as our indexing requirements, by noting that G-Swoosh, the most general ER algorithm, is inefficient. For a non-ICAR problem we will merge more than allowed and require the split operation as post-processing. Requiring ICAR may also help in somewhat limiting the size of the components that we create by merges.

Feature-based Matching and Indexes

Entity matching can also be defined in a more general way, forming features based on attributes.

In [Benj 09] features are a set of attributes used to decide matchings, used each feature independently. This concept is applicable for the generic models of Section 5.3.1 and Section 5.3.2. Next we define features for the record partitioning model with practical index classes. Feature-based matching and these feature and index classes will be used when we develop efficient indexing algorithms.

Let **features** be a set of functions $F = \{f_1, f_2, ...f_m\}$, each f_i mapping entities to arbitrary set of values. Entities e_1 and e_2 match ($e_1 \sim e_2$), if a matching feature f_i is found, denoted by $f_i(e_1) \sim f_i(e_2)$.

The notion of two entities matching along feature f_i is defined as a function of the two attribute subsets $f_i(e_1)$ and $f_i(e_2)$, depending on the application. For example, we may require $f_i(e_1) \cap f_i(e_2) \neq \emptyset$ but the definition may involve distance functions over the attribute space as well.

Next we define a special set of features, with features preserving similarities after merges. Let a feature f be a **stable feature** for an entity set E, if

$$\forall e_1, e_2 \in E \text{ with } f(e_1) \sim f(e_2) \Rightarrow f(e_1) \sim f(e_1 \cup e_2) \text{ and } f(e_2) \sim f(e_1 \cup e_2).$$

In practice features defined by domain experts are usually stable.

Use of feature indexes can speed up the resolution process when identifying entity matchings. A feature index provides candidates for a given entity: if there exists an entity with matching feature, the feature index must include it in the result.

For feature f and entity $e \in E$, let the **feature index** $index_f(e)$ be the subset of entities E such that

$$\text{if } f(e_1) \sim f(e_2), \text{ then } e_2 \in index_f(e_1) \text{ and } e_1 \in index_f(e_2).$$

Given two entities e_1 and e_2 and features $f_1, f_2, \ldots, f_m$, the match condition is

$$f_1(e_1) \sim f_1(e_2) \ \vee \ f_2(e_1) \sim f_2(e_2) \ \vee \ ... \ \vee \ f_m(e_1) \sim f_m(e_2).$$

It is therefore unlikely that a particular feature index can be used to find all matching candidates: if not all features are indexed, then possible matches on that given feature can be missed. The next completeness property formalizes if a set of feature indexes can be used to find all matching candidates.

A feature index set $\{index_i \mid i = 1, ..., m\}$ is a **complete index set** for f_j ($j = 1, ..., n$) features and for E entities, if

$$\forall e_1, e_2 \in E, k \in [1, n] : f_k(e_1) \sim f_k(e_2) \Rightarrow$$

$$\exists l \in [1, m] : e_2 \in index_l(e_1) \wedge e_1 \in index_l(e_2).$$

The completeness property for given entities states that there exists at least one feature index candidate for all possible match. A trivial complete index set contains only one index, returning the whole E as candidates. The relation between indexes and features is not necessarily one-to-one: common indexes may be used for more than one features.

5.3.5 Satisfactory Records

In practice there may exist records that do not contain enough information to meet the match criteria. We can determine whether none of the attributes allows matching.

For example suppose that we use the birth name, birth date and tax number attributes of a client for matching: two records match, if they share the tax number or both the birth name and date. In this case if both birth date and tax number are unknown, then it is needless to search matching tuples.

We can define $satisfactory$ for the relational ER model (Section 5.3.2) as an $R_r \to \{true, false\}$ function that, if $t \in R_r$, satisfies

$$satisfactory(t) = \begin{cases} true & \text{if } \exists\, t' \in R_r : t \sim t', \\ false & \text{else.} \end{cases}$$

For the partitioning ER model (Section 5.3.3) with features a similar function can be defined for entities (sets of records). Let $satisfactory_f(e)$ be a true or false valued function for entities $e \in E$ and features f such that

$$satisfactory_f(e) = false \text{ if } \nexists\, e' \in E : f(e) \sim f(e').$$

We can use the same domain knowledge as for the match functions to construct $satisfactory$.

5.3.6 Probabilistic Model

Entity resolution deals with hidden and uncertain real world entities. We shortly introduce a probabilistic model that can be used more or less orthogonally to the algorithmic results described later.

We cannot observe our entities directly, but their properties are recorded in a database. These state that “the name of the given client is John Doe” or “the client was born on January 1., 1977”. Records are considered as events of observing some entity properties. They are implicite and uncertain: deviations, changes, heterogeneous formats may occur. Domain experts, as a matter of fact, work with hidden confidences, probabilities when constructing exact match conditions. The rules they construct describe proper entity matching *in most cases* in the given database – they indicate highly probable matching.

A simple way of handling matching confidences is to use thresholds. We keep in mind that we deal with hidden confidences when constructing merge and match functions, and construct them so that records match, if the probability of belonging to the same entity is greater than a given threshold. Preliminary statistics and distributions can help determining thresholds when using record similarities or other fuzzy matching techniques.

Another way of handling probabilities is to use a model incorporating them directly. The influential early work of [Fell 69] also used a probabilistic model. There a linkage rule divides record pairs into three sets: matching pairs, possible matching pairs (undecided) and non-matching pairs. Other models are built with confidences on records (see [Mene 06] for example).

Let $E = \{e_1, e_2, ...e_n\}$ be the set used to represent hidden real world **entities**. Let $R = \{r_1, r_2, ...r_m\}$ be a set of **records**, where $r_j = \{a_1, a_2, ...a_n\}$ is a set of **attributes**. Let a **probabilistic entity resolution** of an R set of records be

$$(1) \quad PER(R) = \{(r, e, p) \mid r \in R, e \in E, p \in [0, 1]\},$$
$$(2) \quad \forall (r_1, e_1, p_1), (r_2, e_2, p_2) \in PER(R) : r_1 = r_2 \wedge e_1 = e_2 \Rightarrow p_1 = p_2,$$
$$(3) \quad \forall r \in R : \exists e \in E, p \in [0, 1] \text{ so that } (r, e, p) \in PER(R).$$

$PER(R)$ is similar to the $ER(E)$ of Section 5.3.3. The triples can be interpreted as statements about the relation between records and entities. In Eq. (1) we simply state that for (r, e, p), the "record r belongs to entity e with the probability of p". According to (2), we cannot have two different probability values for a given record-entity pair. In (3) we state that a record is assigned to at least one entity.

Semantics and Estimation

Without semantics, $PER(R)$ is only a syntactical construct for assigning records to one or more entities – labeling the records with entities and probability values. Next we add useful meaning to the constructs.

We think of records as events, as observations of an entity or of properties of an entity. With the universe of these events, let $P(e_i|r_j)$ be the conditional probability of encountering the hidden entity e_i when observing record r_j. Similarly, $P(e_i|a_k)$ can be used when observing an a_k attribute value.

A Bayesian interpretation that we intend to use for (r_j, e_i, p_k) is the following. Let $p_k = P(e_i|r_j)$ be the conditional probability of the belief that a hidden entity e_i is present when we observe a record r_j. $P(e_i|r_j)$ is not known, but can be estimated with prior probabilities when a new record becomes known. According to the Bayes theorem,

$$P(e_i|r_j) = \frac{P(r_j|e_i)P(e_i)}{P(r_j)}$$

In the Bayes formula, an approximate $P(e_i)$ can be computed: Given a closed entity set, what are the chances of choosing e_i? For example, if we have 1 million clients and assume uniform distribution, then we inspect a given client with probability $1/10^6$.

$P(r_j)$ is the probability of observing a given record, and can be estimated based on the occurrences of attribute values in our database, or in an external database. For example, "John Smith" is a common name with higher observation probability than "Csaba Sidló". Most ER models do not incorporate this kind of information, and rely only on similarities.

$P(r_j|e_i)$ can be estimated using similarities: what are the chances of observing r_j record in connection with e_i entity? For example, when for a client called "John Doe", what is the probability of recording "John Dow"? The estimation is not straightforward, as real properties of a given entity are unknown. However, if some records of the given entity are already available, then the properties of the real entity can be estimated from them.

We can use the Bayes theorem on attributes too to handle attributes as independent evidences:

$$P(e_i|a_k) = \frac{P(a_k|e_i)P(e_i)}{P(a_k)}.$$

Bayesian Inference

The model and interpretation above may be used to represent the relation between records and real-world entities. We give a brief example how to to build a good quality $PER(R)$, how to label records with entities and probabilities.

We apply Bayesian inference, and process every record as a new evidence. For every entity, we calculate the probability of belonging to that entity, and then we assign the record to the most probable one. More formally, for every r record and then for every possible i values, we test the hypothesis that r belongs to e_i. With probabilities above a given threshold we add the record to the given entities, creating a new triple. We add a new e if no matching entities found. Finally if we added a record to an entity, we re-calculate the probabilities of the (r, e, p) triples affected. We may also recalculate probabilities when domain experts give new, external evidences.

With careful probability estimations and threshold tuning a good model of real-world entities can be built. Domain experts can interpret and review this model. We believe that the formalism above is practical and can adopt to the ER scenarios we met and to the algorithms we construct later. As a future work we will apply the model to our algorithms.

5.4 Algorithms for Relational Databases

In practice, input data usually resides in relational databases and can grow to huge volumes. Yet, typical solutions described in the literature employ standalone memory resident algorithms. Next we utilize facilities of standard, unmodified relational database management systems (RDBMS) to enhance the efficiency and scalability of ER algorithms.

The main focus when dealing with ER for relational databases is to develop efficient, industry scale attribute-based methods. We build on the relational data model of Section 5.3.2, aiming to fit the RDBMS environment. Since standard relational databases already offer general, well-tuned algorithms on relations for batch processing, the following algorithms will use tables as main data structures with relational operations expressed as SQL statements. Our methods will hence suit a uniform architecture with efficient storage, memory management, caching, searching and indexing facilities. Our algorithms are tightly coupled to the database (see Section 3.1), a beneficial property in practice since input data usually resides in relational databases.

Existing algorithms for GER are in-memory algorithms and keep the whole closure set in memory, therefore the scalability of these methods is limited. Although the algorithms are optimal in that the number of required match operations is kept minimal, pairwise search for matching pairs would require more efficient data structures than those in pre-existing implementations in order to scale to practical applications.

We demonstrate the advantage of our approach when huge amounts of data has to be handled. We use insurance client data (see Section 5.1.1). In our motivating scenario an experimental client data mart is built, integrating several data sources. Data integration begins with cleaning and loading data into a unified schema by massive ETL tools. Then a slowly-changing, versioned client dimension is built up that includes all available attributes, with additional fact tables providing relations between clients and other dimensions such as contract or postal address. Despite the exhaustive pre-processing, several duplicates remained due to different attribute sets in the source systems, different data recording and storage policies as well as variation of the attributes over time.

The AEGON data mart used in our experiments has tens of millions of source records, which makes the use of in-memory algorithms difficult. However, the Generic ER approaches of Section 5.3.1 and Section 5.3.2 seem adequate for the requirements. Domain experts define exact rules on client attributes for constructing match and merge functions of client records. Merging of records was expected in this case: a simple record had to be produced containing as much information of the underlying matching records as possible. Finally, an automated ER process with exact results has to be produced that can be used for data

Algorithm 2 Dominated record elimination

input: I' closure instance
output: I' input instance without dominated records

```
1: for all $t, t' \in I', t \neq t'$ do
2:     if $t \preceq t'$ then
3:         remove $t$ from $I'$
```

mart updates. We believe that similar tasks and requirements commonly appear in practice and require the revision of existing GER formulations.

5.4.1 Swoosh Variations

Pre-existing G-Swoosh and R-Swoosh ([Benj 09]) are optimal algorithms to compute $ER(I)$. G-Swoosh solves the general ER problem, while R-Swoosh assumes ICAR properties and merge domination. Both algorithms are optimal in sense of the required pairwise match operations. First we adapt these algorithms to the relational environment.

Swoosh algorithms maintain two sets of records, I and I'. I is the set of records to be processed, I' is the set of records forming the closure of the processed elements. G-Swoosh gets an element from I, matches against all elements of I', and adds the merged element to I along with all matching records from I'. If there is no matching, then the selected element is moved to I'. With no assumptions on domination, G-Swoosh can eliminate dominated records only after producing the whole closure. This can be done using a simple algorithm (Algorithm 2).

R-Swoosh (Algorithm 3) enhances the process by dropping source tuples right after merging, making dominated record elimination unnecessary at the end. Eliminating dominated records in every round keeps the size of I' smaller too, reducing the costs. F-Swoosh ([Benj 09]) is the most efficient Swoosh algorithm. F-Swoosh is extension of R-Swoosh, defining features on attributes to decide matches, and maintaining index-like structures to speed up searching for matching pairs.

We can implement Swoosh algorithms in the relational generic ER model by using tables for I and I'. Since data modification languages and APIs – built around standard SQL – do not enable implementing general algorithms, we have to use an embedding language to implement the logic of the algorithm. The implementation itself can be a standalone unit implemented with any programming language able to connect to relational databases, or it can be an embedded stored procedure implemented with a supported programming language. How-

Algorithm 3 R-Swoosh

```
input: $I$
output: $I' = ER(I)$

 1: $I' \leftarrow \emptyset$
 2: while $I \neq \emptyset$ do
 3:     $t \leftarrow$ an element from $I$
 4:     remove $t$ from $I$
 5:     $buddy \leftarrow null$
 6:     for all $t' \in I'$ do
 7:         if $t \sim t'$ then
 8:             $buddy \leftarrow t'$
 9:             exit loop
10:     if $buddy = null$ then
11:         add $t$ to $I'$
12:     else
13:         $merged \leftarrow \langle t, buddy \rangle$
14:         add $merged$ to $I$
15:         remove $buddy$ from $I'$
```

ever, the space- and time-consuming operations can be formalized using SQL, which makes the role of the embedding language insignificant.

Pairwise match functions on relations can be expressed as filtering operations in the `where` clause of SQL queries. Next we will give examples on client data. We are dealing with identities, with match functions such as “two identities cover the same person, if they have the same tax number or social security number, or if the birth date and birth name attributes are both equal”. For example we can find matching pairs in R-Swoosh in the following way (supposing that t is an arbitrary record):

```
select $I'$.* from $I'$
where  ($t.birth\_name\_id = I'.birth\_name\_id$
         and $t.birth\_date = I'.birth\_date$ )
       or $t.tax\_number = I'.tax\_number$
       or $t.ss\_number = I'.ss\_number$
```

Merging two records can be expressed using functions and operators applied to the result set, in the `select` list of a query. The next example depicts a merge of t and t', using functions of the SQL-92 [ISO 92] specification:

```
select coalesce($t.birth\_date$, $t'.birth\_date$) as $birth\_date$,
        ( case when length($t.name$) $\geq$ length($t'.name$)
          then $t.name$ else $t'.name$ end ) as $name$,
        ...
```

Regulations of our current SQL environment give a new set of constraints on expressing match and merge functions, as SQL is not a Turing-complete language (although using UDFs adds some more versatility). These new constraints are orthogonal to the ICAR properties: we can easily implement functions violating ICAR. As a simple example, the SQL merge expression "$t.premium + t'.premium\ as\ premium$" violates ICAR.

SQL implementation of $match_r$, the relational match function is parallel with pairwise match functions. When implementing $merge_r$ functions we would like to formalize the semantics in a single `select` clause. We use grouping selects to collect matching records, and aggregate functions to implement semantics. For example a simple merge function that chooses an arbitrary not-null value can be formalized as follows:

```
select max($birth\_date$) as $birth\_date$,
       max($birth\_name$) as $birth\_name$,
       ...
```

Aggregate functions of our preferred RDBMS can limit the choice of possible set-style merge functions. Windowing analytic aggregate functions of Oracle or other interesting extensions of SQL-92 aggregate functions in other RDBMSs may give us sufficient versatility. We can express complex merge functions such as "the longest name's id" or "the passport id that occurs most often". k

5.4.2 DB-GER and Variations

DB-G-GER algorithm (Algorithm 4) computes $RER(I_r)$ when all the properties of (5.1) without merge domination hold.

DB-G-GER iterates through the input relational instance I, and maintains an instance I' with the previously processed and merged elements. In every iteration step I' is the resolved entity set of the previously processed elements. The main step is line 4, which can be expressed as a single SQL statement using aggregate functions, as the next example shows:

```
select count (*), max($birth\_name$), max($birth\_date$), ...
from   $I'$
where  ( $t.birth\_name\_id = I'.birth\_name\_id$
         and $t.birth\_date = I'.birth\_date$ )
       or $t.tax\_number = I'.tax\_number$
       or $t.ss\_number = I'.ss\_number$
```

Algorithm 4 DB-G-GER

```
input: $I$
output: $I' = RER(I)$

1: $I' \leftarrow \emptyset$
2: for all $t \in I$ do
3:     add $t$ to $I'$
4:     $merged \leftarrow merge_r(match_r(t, I'))$
5:     if $merged \neq t$ then
6:         add $merged$ to $I'$
7: remove dominated elements from $I'$
```

Since t is already in I', we merge at least one tuple. If the merge query groups only one tuple together, we can be sure (in line 5) that the merged element is the same as t: this follows by the properties of (5.1).

We do not presume merge domination, therefore we have to eliminate dominated records in a separate step (line 9). We can build up a batched SQL statement to select dominated records in the following fashion:

```
select $i_2$.*  from $I'$ as $i_1$, $I'$ as $i_2$ or
where  $i_1.rowid \neq i_2.rowid$ and
       -- matching:
       ($i_1.tax\_number = i_2.tax\_number$ or ...)  and
       -- domination:
       ((case when $i_1.birth\_date$ is null then 0 else 1 end)
       + ...)  <
       ((case when $i_2.birth\_date$ is null then 0 else 1 end)
       + ...)
```

Here we formalized a simple domination relation: a tuple dominates another matching tuple if it contains more non-null attributes.

The next algorithm, DB-GER (Algorithm 5) presumes merge domination. It eliminates dominated records right after merging, therefore shrinks I' in every round. Line 6 can be implemented on relations as follows:

```
delete from $I'$ where $i_1.tax\_number = i_2.tax\_number$ or ...
```

Both DB-G-GER and DB-GER produce $RER(I_r)$, and can be implemented using efficient batched database operations.

We can profit from using strong merge domination, we can drop unnecessary tuples right after merging. Not all source records can be dropped, but the ones that match the merged record. The necessary modification affects only one row

Algorithm 5 DB-GER

input: I

output: $I' = RER(I)$

```
1: $I' \leftarrow \emptyset$
2: for all $t \in I$ do
3:     add $t$ to $I'$
4:     $merged \leftarrow merge_r(match_r(t, I'))$
5:     if $merged \neq t$ then
6:         remove $match_r(t, I')$ from $I'$
7:         add $merged$ to $I'$
```

of algorithm DB-GER (Algorithm 5). If we use properties of (5.1) and strong merge domination, line 6 changes to “remove $match_r(merged, I')$ from I'”.

Database Indexes

An advantage of using set-based match functions is that we can efficiently *search* for all matching tuples using indexes instead of going through all elements of a set and making pairwise decisions. In case of relational databases, when DB-GER merges matching records in line 4, the indexes suggest records that satisfy at least one part of the match criteria. If table I' is sparse enough, index-based table accesses can be a lot less costly than a full table scans. The time cost of searching in a regular B-tree index depends on the depth of the search tree, which grows much more slowly than the number of elements (and handled usually as a small constant).

The idea of shaping features on attributes and making feature-level decisions in [Benj 09] has the same motivation as indexing. As described in Section 5.3.4, a feature is a subset of attributes, and the match criteria is a combination of feature-based conditions. Two records match if at least one feature-pair indicates matching. F-Swoosh, the feature-level ER algorithm in [Benj 09] stores positive feature-comparisons in a linear space hash table. Another set is also maintained for storing features that gave only negative matches before. These structures can also be interpreted as indexes.

Available types of indexes are RDBMS-dependent. Besides the basic B-tree variants we may use bitmap, spatial (GIS) and multimedia indexes or indexes for text similarity search. Multidimensional indexes such as general R-trees can also be useful.

We may expect major performance improvement with adequate indexing. However, greedy indexing can harm performance if index updates cost more than the search time improvement. As a basic index selection strategy we can build an index for the feature with the least selectivity. We will examine some observations related to indexing in Section 5.4.3.

Database-independent algorithms with efficient indexing are discussed in Section 5.5.

Pre-filtering

For records not containing enough information for matching we can define the $matchable$ of Section 5.3.5; it is needless to search matching tuples in I' for them. We can sort out unsatisfactory tuples from the input by extending DB-GER with an extra condition in Line 2 on $matchable$.

Uncertainty

DB-GER algorithms produce exact results, but they also enable using fuzzy or similarity-based match conditions (see 5.3.4 for models). We give an example how similarity-based matching with confidence threshold can be implemented.

Common RDBMSs provide us useful attribute types and indexes supporting probability feature matches. For example, in PostgreSQL we can build GIS indexes on geospatial locations. We can then efficiently evaluate match conditions such as “two buildings can be considered the same if the distance of their central point are in a range of 10 meters”. Supposing that b_1 and b_2 are such location attributes, the match condition can be expressed as

$$b_1 \,\&\&\, Expand(b_2, 10) \textbf{ and } distance\_sphere(b_1, b_2) < 10.$$

Here the $\&\&$ operator pre-filters the result based on an efficient GIS index.

Other important examples of uncertain conditions with thresholds are string similarity searches such as matching very similar names. Most of the RDMBSs support string similarity searches with indexes.

Approximate results in the insurance scenario can also be used to identify households or company hierarchies. We would like to find entities not explicitly present in the source data, but GER algorithms can still be applied easily.

Incremental Processing

The agglomerative style of R-Swoosh and DB-GER algorithms fits to the regular data warehouse refreshment policies. We can build an agglomerative delta-load process where only new records are processed in every refreshment cycle. I' always contains $RER(I)$ of the preceding records. This way we do not have to face huge data volumes in every refreshment round. As a special case, on-line, event-driven refresh is also solvable: it is possible to implement real-time updates of resolved entity sets.

Mapping Source and Resolved Records

We would often like to store all input records and define the mapping between source and resolved records. For example after preprocessing we may store all source client records without merging as client versions. We build up $RER(I)$ to compute exact aggregations, or to stream back resolved information to ERP systems.

The $RER(I)$ set contains exactly one matching record for an original source record in case of ICAR and merge domination: we select the single matching record from $RER(I)$ for the original source record. In case of strong merge domination we can have more matching tuples in $RER(I)$ for a given tuple. To find the dominant one we have to use all the information, we have to merge all matching tuples. The merged tuple is guaranteed to be in $RER(I)$.

5.4.3 Experiments

All DB-GER experiments were performed on a commodity PC with Intel Celeron 3.2 GHz CPU, 1 GB RAM and a 7200 RPM disk without RAID. We used Oracle 10g with data warehousing configuration set up to use 400 MB SGA memory.

The algorithms were implemented using PL/SQL, using only regular SQL functionality and regular B-tree indexes. No physical level or other special optimization was done. We implemented F-Swoosh of [Benj 09], using Java 1.5, with hash set and hash table data structures from the standard library. F-Swoosh measurements were performed on a separate but identical hardware with Windows XP. Input data was not stored locally: input records were coming from the separate Oracle database, and results were written back. The execution times do not contain the cost of initial and final data transfer.

Experimental real world dataset was provided by AEGON Hungary, containing approximately 12 million distinct identity records of clients. Identities contain common attributes such as name, birth name, mother's name, sex, birth date and place, external identifiers such as social security number or tax number. Attributes were cleaned and unified using the ETL facilities of an experimental insurance client data mart. Preliminary data cleansing included standardization and correction of attribute values, using external knowledge too, such as first name databases. We have chosen uniform match and merge functions verified by domain experts. We used the properties of (5.1) and strong merge domination. Yet, on our database only a few records conflicted with ICAR.

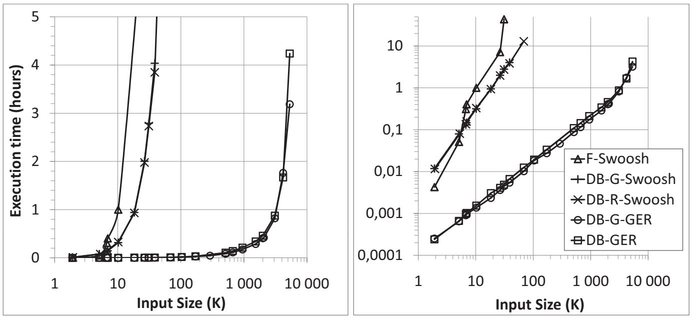


Figure 5.3: Scalability of the algorithms (with linear and log time scale)

We implemented G-Swoosh and R-Swoosh on relations (as DB-G-Swoosh and DB-R-Swoosh), DB-G-GER and DB-GER. Both DB-G-Swoosh and DB-G-GER employed a one-round duplicate elimination step. All algorithms used the same input and output schema. We measured execution times without the operations required to produce input data. The experiments were averaged from multiple executions in different orders to overcome caching and other performance issues beyond our control.

Figure 5.3 shows execution times of the algorithms against the size of input data. Naive database implementations of G-Swoosh and R-Swoosh scale poorly, Java F-Swoosh implementation performed worst. The main cause is that DB-Swoosh variants search for matching records more efficiently than the original linear search, and they use batched set-styled operations.

Interestingly, DB-GER and DB-G-GER, DB-G-Swoosh and DB-R-Swoosh perform similar. This means that the role of the domination is not significant. When using instant dominated record removal, the cost of the required delete operations balances the cost of handling a larger I' when eliminating dominated records at the end. The aggregated costs of duplicate elimination is depicted in Figure 5.4.

We also examined the impact of match selectivity on execution times. We fixed the input size at 50 thousand and measured execution times against merges. We can run experiments with different match functions, but different functions have different evaluation times. Instead, we change the data set, and the match function stays the same: selectivity depends on the match function and both on the data set. With heuristics knowing how the match function works, we can

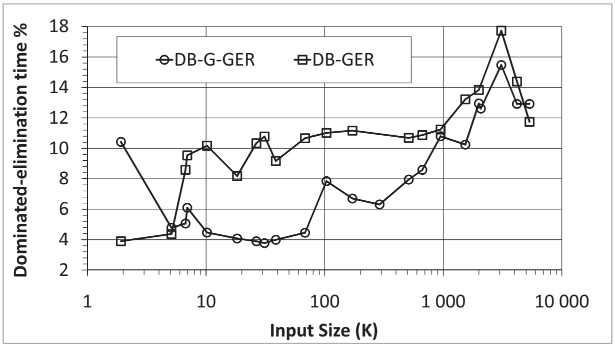


Figure 5.4: Percent of execution time needed to eliminate dominated records

select subsets containing more or less matching pairs. For example, we can increase the number of merges by selecting identities with birth dates of a given year. Figure 5.5 shows the execution times against the count of merged records (the count of distinct records engaged in a merge operation), and against the count of eliminated records (the difference between input and output size). DB-G-GER algorithm performs better for the interval under survey, caused by the high deletion costs of dominated records in every round of DB-GER.

We measured execution times of DB-GER with different indexing strategies (Figure 5.6). Without indexes we do not have to maintain additional structures, but we have to perform full-table scans. The other two variants used a given set of standard indexes over the features, selected and defined by hand. The 'fullindex' version was ordered to always use indexes, a set of indexes. The 'optimizer' version relies on the query optimizer to select an appropriate plan, using indexes of the same set if necessary. Since no stress was laid on the selection of indexes to build, the two versions can only differ in query performance, not when updating the tables

The overall space cost of the indexes (note that some feature indexes could be omitted) were about 1.9 - 2.0 times the size of the table, with a composite index being the largest. This is a significant space cost, yet maintaining these indexes may be a good trade-off. The version without indexes outperforms DB-Swoosh and F-Swoosh variations because of the new set-styled batched operations. There is no significant difference between the 'fullindex' and the 'optimizer' versions.

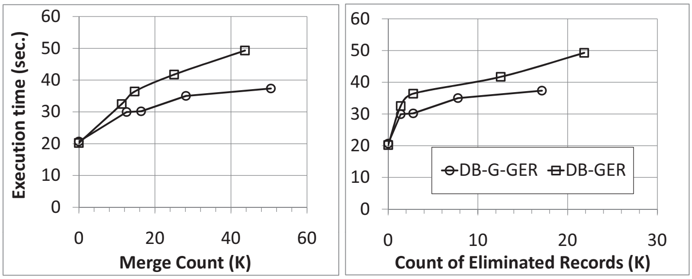


Figure 5.5: Impact of match selectivity

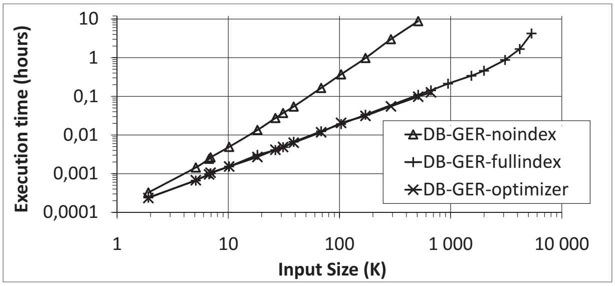


Figure 5.6: Performance of DB-GER with different indexing strategies

5.5 Algorithms with Efficient Indexing

The following algorithms all solve the ER problem of the record-partitioning model of Section 5.3.3; they differ how they build and use indexes. A naive solution would be to iterate through the input entity set and find matching pairs, as long as such pair exists. Instead of comparing all pairs, we reduce search time by indexing our data. However, at the same time indexes increase space complexity and add additional maintenance costs. Indexing solutions are therefore not necessary faster than non-indexing variants: we investigate the efficiency of different index realizations later, and show that index use pays off.

In what follows, we consider ER algorithms solving the problem formulated in Section 5.3.3, where we refine record partitions. The input of the algorithms is therefore a set of records as entities, with each record corresponding to its unique own entity; as the output, some records will be merged to form a smaller size entity set E'. The algorithms may also work with partially merged records as input, as long as only matching records are merged. We assume feature-based matching and use the concepts of feature indexes and complete index sets.

However we can use indexing capabilities of database management systems, as in Section 5.4, these algorithms are loosely-coupled and aimed to build standalone applications.

5.5.1 Basic Feature Indexing

IndexER (Algorithm 6) is our basic indexing solution to the ER problem where feature indexes are handled as search data structures. IndexER maintains a result set E' containing no unexplored matches and extends this set by entities from the input. Feature indexes contain only entities of E', therefore have to be updated when E' changes.

E' always contains the resolution of the processed entities, and while E diminishes, IndexER solves the $ER(E)$ problem. Efficiency of the algorithm depends both on the indexing tools used, and on properties of the input data set (eg. how many matching records it contains), as well as on the match logic (how many features there are, are they similarity-based etc.). We explore some aspects of performance later.

Only satisfactory entities need to be indexed, containing enough information to match. A satisfactory function can be defined based on heuristics of the given feature and match logic (according to Section 5.3.5, for example by filtering incomplete or empty attribute values. When using a $satisfactory$ function, the index lookup and update (Line 9 and 13) become conditional, dealing only with $satisfactory_f(e)$ features.

Algorithm 6 Index-ER

input: Entity set E such that each record corresponds to a unique entity.
output: $E' = ER(E)$

```
1: $E' \leftarrow \emptyset$
2: $merged \leftarrow null$
3: while $E \neq \emptyset \vee merged \neq null$ do
4:     if $merged \neq null$ then
5:         $e \leftarrow merged$
6:     else
7:         $e \leftarrow$ an element from $E$
8:         remove $e$ from $E$
9:     $candidates \leftarrow \cup_f index_f(e)$
10:    $merged \leftarrow null$
11:    if $candidates = \emptyset$ then
12:        $E' \leftarrow E' \cup \{e\}$
13:        for all indexes: add $e$ to $index_f$
14:    else
15:        for all $c \in candidates$ do
16:            if $c \sim e$ then
17:                $merged \leftarrow merged \cup \{c\} \cup \{e\}$
18:                remove $c$ from $E'$
19:                remove $c$ from all $index_f$
```

Algorithm 7 Pre-Index-ER

input: Entity set E such that each record corresponds to a unique entity.
output: $E' = ER(E)$

```
1:  $E' \leftarrow \emptyset$
2:  $merged \leftarrow null$
3:  prepare all $index_f$ with $E$
4:  while $E \neq \emptyset$ or $merged \neq null$ do
5:    if $merged \neq null$ then
6:      $e \leftarrow merged$
7:    else
8:      $e \leftarrow$ an element from $E$
9:      remove $e$ from $E$
10:   $candidates \leftarrow \cup_{f:\text{satisfactory}_f(e)}\{e' \in E' : e'$ originates from an $index_f(e)$ entity $\}$
11:   $merged \leftarrow null$
12:   if $candidates = \emptyset$ then
13:     $E' \leftarrow E' \cup \{e\}$
14:   else
15:     for all $c \in candidates$ do
16:       if $c \sim e$ then
17:         $merged \leftarrow merged \cup \{c\} \cup \{e\}$
18:         remove $c$ from $E'$
```

5.5.2 Feature Pre-indexing

Index updates are usually expensive, next we try to avoid updating the indexes. We can observe that records do not change during the resolution process: Algorithms only re-partition records when merging entities. If we build feature indexes in a batch for all records before the resolution process, we can save index maintenance costs.

We can build the index for a stable feature preliminary. At the beginning each entity consists of a single record, and feature indexes refer to that initial entity. Additional data structures are needed to track entity merges, and to record if an entity belongs to the resolved set. Pre-Index-ER (Algorithm 7) implements this indexing scheme. We also assume the presence of a satisfactory function, used as a pre-filtering condition. While Index-ER (Algorithm 6) feature indexes grow with the number of processed records, Pre-Index-ER indexes always contain all records.

Algorithm 8 Block-Index-ER

input: Entity set E such that each record corresponds to a unique entity.
output: $E' = ER(E)$

```
1: $E' \leftarrow \emptyset$
2: for all $f$ feature do
3:     $B_f \leftarrow$ partition $E'$ according to $f$
4:     for all $B_f^i$ partition in $B_f$ do
5:         update $E'$ with $ER(B_f^i)$
```

5.5.3 Feature-based Blocking

Blocking is a proven method to speed up ER algorithms. Blocking divide records into smaller subsets based on expert heuristics including ZIP code or first letter of family names. ER is then performed on the smaller subsets more easily, especially when they fit in the memory.

With blocking processing speeds up, but potential matching pairs between different blocks may be missed. One solution to the problem of missed pairs is to use multiple blocking criteria, and combine the results. Another potential solution is iterative blocking [Whan 09], where merged entities are delegated to other affected blocks.

Block-Index-ER (Algorithm 8) implements a new partitioning scheme different from both multiple and iterative blocking. We iterate through all features, partition the input set in every round, and solve the sub-problem with an arbitrary ER algorithm, e.g. Index-ER.

In Line 3 a feature-based partitioning is performed. We form up blocks according to feature boundaries. Let $B_f^i (i = 1..n)$ be a **feature-based blocking** of an E entity set with f feature, if

$$\forall i \in [1..n] : e \in B_f^i \Rightarrow \nexists e' \in B_f^j, j \in [1..n], j \neq i : f(e) \sim f(e'),$$

$$\cup_{i \in [1..n]} B_f^i = E$$

Creating feature-based blocking for a given feature and memory limit is not always a straightforward task. However, we do not deal with this sub-problem in the thesis, and suppose that the complexity of constructing the feature-based blocking is negligible.

In Line 5 the algorithm applies all entity merges to E'. If an entity does not exists in E', it is appended. The algorithm iterates through all blocks of a feature-based blocking and applies every merge. Matching entities always fall into a common block for some of the stable features, therefore at the end Block-Index-ER produces $ER(E)$. We have to process all partitions for all features, but the number of these partitions is relatively small for real-world problems.

5.5.4 Index Realizations

Efficiency of Index-ER algorithms depends on properties of the input entity set, the features used and the indexing methods and tools chosen. Next we briefly examine a few alternatives.

The most useful and simplest algorithm variations handle features as attribute value sets, match operators as equality tests. For example, two clients match if they share a birth date, a birth name and a postal address. Conventional search structures are applicable in this scenario. B-trees for example are proven to be optimal and useful general search constructs.

For features based on multiple attributes an index can be built for an arbitrary attribute with good selectivity. Eg. for a complicated birth data feature a birth name B-tree index may be used, if birth name is always known. Another possibility is to use multidimensional indexes, eg. R-trees. With R-trees we can use multiple attributes as search key.

Scalability can be improved by relying on external memory indexes that become slower when running out of cache memory, but keep serving the algorithm. Another possible enhancement is the use of various distributed key-value stores or indexes.

When using similarity-based or probabilistic features and match conditions, the indexes provide entities with similar feature value beyond a given similarity threshold. Examples include finding duplicated web pages, using features based on geographic location and distance, features with name similarities etc. Similarity-based indexes are useful in this case, and are exhaustively investigated topics. Document Q-gram and TF-IDF indexes for example have multiple efficient standalone implementations, for distributed environments as well.

5.5.5 Experiments

Experiments were performed on a Linux server containing an AMD 2 GHz Opteron CPU, 7 GB of main memory and a 7200 RPM disk without RAID. We used PostgreSQL 8.4 using 1GB memory, Berkeley DB Java Edition (BDB) 4.1 using 500 MB cache, Sun JDK 1.6 with 3 GB maximum heap size. A relatively weak hardware architecture was chosen intentionally: behavior of the algorithms and indexing schemes become problematic and therefore interesting when reaching the given constraints (eg. the memory limit).

Algorithms were implemented using Java, input and output were stored in PostgreSQL. We note that our solution works also based on various other indexing tools not covered in this paper, including Project Voldemort, Kyoto Cabinet, ScalienDB, etc. We performed repeated executions and averaged the results. Time needed for input read and output write are not included.

For experiments, a data set of the AEGON Hungary Insurance Ltd. was used, containing approximately 20 million client records. Records consist of both personal attributes (names, birth data, tax number, etc.) and company-dependent identifiers (see Section 5.1.1). According to preliminary estimates and experimental results each client has 1.95 records in average. We used random sampling to obtain smaller subsets. We also used selection heuristics to influence the count of records per user. For example, selecting all records for the family name 'Smith' instead of random sample will increase the match count.

Match logic provided by experts included simple attribute-equality testing, e.g. "two entities match, if they have common tax numbers", and more complicated ones. For example, the birth data feature used 5 attributes, with multiple attribute-value equality testing.

We used two previously known algorithms for comparison. DB-GER is an SQL-based ER algorithm for relational databases (see Section 5.4.2 or [Sidl 09]). Java F-Swoosh is a basic Java in-memory F-Swoosh implementation of [Benj 09], the same as in Section 5.4.2. Both DB-GER and Java F-Swoosh experiments were performed with Oracle 10g database. Index-ER-BDB and Pre-Index-ER-BDB used Berkeley DB B-trees for feature indexes and also to store records. Pre-Index-ER-Pg used standard PostgreSQL indexes. Block-Index-ER-Pg used feature-based blocking and in-memory algorithms for feature indexes. Feature block construction and the update operation with block results is done by PostgreSQL.

Figure 5.7 plots execution times against input size. Java F-Swoosh showed poor performance without proper indexing. Performance of Block-Index-ER-Pg was still inferior: PostgreSQL through JDBC handles batch updates slow on the whole entity set. In-memory processing and Index-ER variation on blocks costs negligible time. So Block-Index-ER stays promising, supposing that a better entity store can be found with faster updates. Interestingly Pre-Index-ER-Pg also performed poor: feature index lookups were much slower than by BDB.

Both Pre-Index-ER-BDB and Index-ER-BDB outperform previous solutions. Figure 5.8 shows how the number of records processed changes for a smaller record set. Index-ER-BDB slows down as the size of the feature indexes increase. With Pre-Index-ER-BDB processing becomes faster as more and more merges were performed.

The reason why Index-ER-BDB outperforms Pre-Index-ER-BDB variation

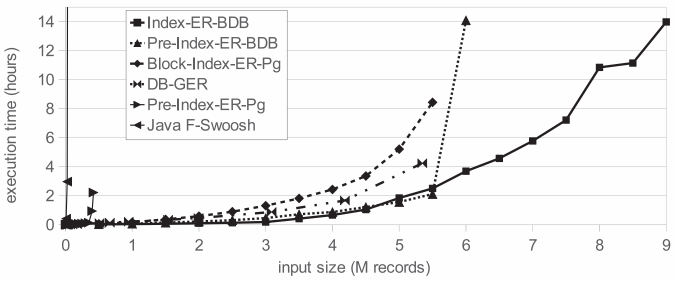


Figure 5.7: Execution times against input size

is depicted in Figure 5.9. At around 3.5 M records, Index-ER-BDB slows down since at this point, BDB runs out of cache memory. However, Pre-Index-ER-BDB builds the whole index at the beginning, runs out of cache memory right away, and runs slow all along.

The additional space cost to store feature indexes depends both on the number of features and on the distribution of feature values. In our experiments 5 features were used, using diverse attributes with more or less values. Index size varied from 2 to 3 times the original database size. For PostgreSQL this realized as around 6.9 GB index size compared to the 2.4 GB full database size.

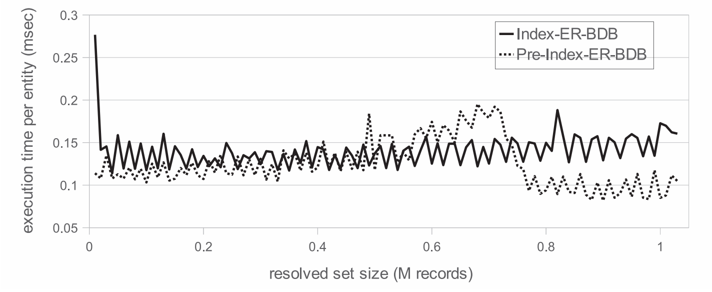


Figure 5.8: Processing speed change for 1.1 M records

5.6 Distributed Algorithms

Previously we built efficient algorithms on relational databases and efficient indexing methods, speeding up the entity resolution process. Another possible way of scalable algorithm construction is distribution. Existing distributed ER algorithms typically rely on shared memory architectures. Next we develop algorithms for shared-nothing architectures, and demonstrate the strength of our algorithms with experiments. The following methods are somewhat orthogonal to the previous results; a good indexing solution can be for example combined with our distributed algorithms.

We demonstrate that ER can be solved using algorithms with three different distributed computing paradigms:

- Distributed key-value stores;
- Map-Reduce;
- Bulk Synchronous Parallel.

At first we show simple reductions to communication complexity and data streaming lower bounds to illustrate the difficulties with a distributed implementation: If the data records are split among servers, then basically all data must be transferred.

The naive solution to solve the ER problem is to iterate through the input entity set and find matching pairs while there are unexplored matchings. Such algorithms, including G-swoosh and even the improved R-swoosh ([Benj 09]), run in quadratic time and are hence inefficient for large data sets.

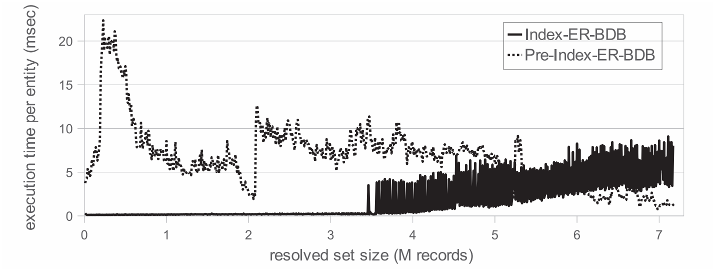


Figure 5.9: Processing speed change for 7.2 M records

We assume the partitioning model of Section 5.3.3, with the indexable attribute properties. In our distributed algorithms we will either use feature indexes combining several attributes or rely solely on attributes and undo false merges in the post-processing phase.

Next we describe two critical issues of a parallel implementation that are normally considered as minor details in the literature since they are naturally resolved by sequential algorithms. Note that D-swoosh [Benj 07] and the older version P-swoosh [Kawa 06] as well as the first results in the area [Hern 98] require shared memory and do not address these problems.

First, using multiple processing nodes, efficient data distribution keeping the majority of candidate duplicates at the same node does not exists. Stated in another way, there is no locality sensitive hashing for the minimum distance. For inputs of length n, the probabilistic (bounded error) communication complexity of set intersection is $\Theta(n)$ [Kaly 92]. We discuss consequent negative results about problem partitioning in Section 5.6.2 by rephrasing data distribution as communication problem.

Second, the graph of entity mergers along matching values of various attributes may form an arbitrary graph. There is a need for connected component identification, which is described first in the iterative blocking algorithm of [Whan 09] that can be considered the sequential version of some of our methods. Parallel connected components [Hirs 79] is not as simple and the general solution may require several passes. For efficiency it is crucial to base our algorithm on the assumption that the graphs are tiny (which is usually the case in practice).

The contributions of this Section are as follows.

- We show that, simply speaking, any distributed ER algorithm must exchange all of its data across different servers, hence significantly hardening the task compared to pre-existing shared memory ER solutions (Section 5.6.2). In particular this means that blocking may result in significant gains for multi-threaded ER but much less for ER over distributed data.

- We demonstrate that ER can be implemented under three major distributed programming paradigms: distributed key-value stores, Map-Reduce, and Bulk Synchronous Parallel.
- We measure our distributed algorithms on very large scale real world data. Our client database consists of 20 million records that we blow up to 600 million for scalability tests.

The goal of our distributed implementations is to evaluate the applicability of the frameworks and show their strength and limitation for complex ER tasks. We emphasize that our codes contain little optimization in their interaction with the corresponding software framework; also the frameworks, in particular the Bulk Synchronous Parallel, is yet in an incubatory phase and we may observe significant speedup by using an improved release.

5.6.1 Magnitudes of Input Data

Next we demonstrate that smart algorithms over emerging distributed infrastructures enable ER for several orders of magnitude larger data sets than in earlier research. Earlier results give experiments for a few 100,000 records (see Section 5.2). For example, similar to the size of our data set, in [Whan 09] 2 million records are processed, but processing took several hours and distribution to more servers was not considered.

For our 20 million record real world customer data set, our fastest distributed algorithm completes its task in less than an hour over 15 low-end servers. Furthermore, we experimented by replicating our data to a 300 times larger data set than in [Whan 09]. Experiments with data closest to our size is reported in [Weis 08], 10 million records. They rely on a simple sorting based algorithm [Hern 98] that also forms the base of our methods. They produce approximate results and deploy commercial database management systems that however impose limitations to their procedure, for example the usability of similarity indexes is unclear.

A distributed ER implementation that we are aware of, D-swoosh [Benj 07] does not rely on shared memory but uses no distributed software infrastructure either. They communicate by sockets between pairs of servers; also the number of records is a mere few 100,000 in their experiment.

5.6.2 A Lower Communication Complexity Bound

Blocking is a proven method to speed up ER algorithms but, as we will show, its power diminishes when data is distributed across multiple servers with no shared memory.

Blocking divide records into smaller subsets based on expert heuristics including ZIP code, first letter of family names, etc., so that ER is performed on the smaller subsets more easily. In this way we can divide the original problem, but may miss potential matching pairs between different blocks. One solution is to use multiple blocking criteria, and then combine the results of the different ER results together. Or, combine all the possible blocking results for exact results, as Algorithm 8 does. Another potential solution is iterative blocking [Whan 09], where merged entities are delegated to other affected blocks.

For efficient blocking, we may use external knowledge, e.g. only compare records with the same zip code, but this approach may fail if a person moves to another location. As another example that is characteristic to our real data set, maiden names have arbitrary connections to names (in some languages, even the first names may change!) for married women representing probably at least a quarter of most customer data sets.

Next we deal with algorithms blocking the original data sets for multiple processing nodes. These nodes compute their ER results locally, while communicate with each other to produce an exact result of the original problem.

The following set of results indicates that no ER algorithm exists that distributes the data and exchanges records among the parts without communicating significantly less than the entire data set among the processes. Common to these results is that they use a reduction to the probabilistic (bounded error) communication complexity of set disjointness. By the result of [Kaly 92], the question whether two n-subsets of a universe intersect cannot be decided by communicating less than $\Theta(n)$ bits.

Our first negative result is a direct consequence of the set disjointness communication complexity bound. Given that the records are partitioned to at least two data servers in some clever way, we may want to test if ER can be solved by local computations. In other words, for a given attribute we need to check whether there is a value that appears at both servers. By the communication bound, this task requires $\Theta(n)$ bits of communication for n records. Although a single bit for an attribute value may be considered a very efficient encoding, still the $\Theta(n)$ bound means that we basically have to communicate all data between the parts regardless of how smart algorithm is used for partitioning.

The above negative result can be reformulated by using another lower bound [Alon 99, Proposition 3.8] stating (among others) that the number of distinct elements cannot be exactly computed in less than $\Omega(n)$ memory bits. In addition, this problem cannot be solved by sampling either [Char 00]. There are however low space relative error approximation results that may give hope of speeding up some ER heuristics both in [Alon 99] and earlier in [Flaj 85]. Note however that for an exact ER algorithm, the negative results apply.

The impossibility of finding a "very smart" partitioning method can be expressed in another way. In the space of attributes as dimensions, two records are similar if they agree in at least one coordinate (as with the basic attribute-equality matching). Hence another way to state the negative results, there is no efficient locality sensitive hashing for entity linkage as similarity. Our argument is reminiscent to the non-existence for extremely nonconvex dissimilarity functions such as the minimum or a variant that simply counts the number of nonzeroes frequently termed Donoho's zero-'norm' [Dono 04].

Common to the three algorithms in Sections 5.6.3–5.6.5 is that they first communicate all data records among different parts of the data set and then perform additional steps to compute connected components. Communication is performed in iterations; in one iteration, the attribute values of a single attribute is communicated. Disregarding the work performed by the connected component algorithms that we consider low in practice, our algorithms are optimal by the above lower bounds.

5.6.3 Algorithms Based on Key-Value Stores

High performance key-value stores are growing in both size and importance; they now are critical parts of major internet services such as Amazon (Dynamo [DeCa 07]), LinkedIn (Project Voldemort [Proj]), and Facebook (memcached [Fitz 04]).

In our first distributed implementation we choose Project Voldemort due to its ease of installation and APIs. We note that our solution works also based on various other indexing tools not covered in this paper, including Kyoto Cabinet, Scalien DB, or even MySQL.

The ER algorithm itself is an adaptation of a sequential connected component algorithm by relying on the distributed key-value store. It is a simplified version of Algorithm 6 (Index-ER), not dealing with features, but only with attribute matchings. The algorithms can be extended to handle more complex match criteria, as they produce candidates that can be verified by arbitrary logic. Efficiency of the algorithm depends both on the indexing tools used, and on properties of the input data set (e.g. how many matching records it contains), as well as on the match logic (how many attributes there are, are they similarity-based etc.).

The implementations manage entities by their entity ID ID(e). We propose two variants:

Algorithm 9 Union-Find ER by distributed key-value store
input: Entity set E; one attribute store for each attribute with attribute value as key and entity ID as value; another auxiliary store with both key and value as entity ID.
output: $E' = ER(E)$

```
1: for all entities $e$ do
2:    for $\ell = 1, \ldots, k$ do
3:       for all $r \in e$ do
4:          for all IDs $i$ with attribute matching $a_{r\ell}$ do
5:             erase $i$ from the attribute store
6:             union$(i, \mathrm{ID}(e))$
```

1. The list of entities is read sequentially from disk. There are k indexes, one for each attribute, and another index that points to the parent of the given entity. This algorithm implements a union-find data structure [Corm 01, Section 21] over this last key-value store.

2. The entities reside in a Voldemort store with ID(e) as key. There are k indexes, one for each attribute, as in the previous solution. Entities are merged by performing a breadth-first search [Corm 01, Section 22] and the entity store is immediately updated during merge.

The union-find based solution (Algorithm 9) simply iterates through all attribute values and unites all pairs that match in this attribute. It requires a data structure in line 6 that can be implemented by another distributed key-value store. As in [Corm 01, Section 21], we need a pointer for all entities to a parent entity; the chain of pointers must lead to the root. We may implement pointers by a key-value store where the key is the entity ID and the value is the parent ID. To save work, optionally in line 5 we may remove those record-attribute pairs that are already considered for merging.

Since the components are assumed to be small, a simple union-find implementation suffices. We apply path collapsing: whenever a pair of entities are united, both have to seek down to the root containing the component IDs. Along these paths, we may replace all pointers directly to the root, thus allowing a near optimal average logarithmic search time.

Supposing small entities, the theoretical complexity of the algorithm is equal to that of union-find, i.e. slightly better than $O(n \log n)$ [Corm 01]. The amount of communication equals to the total size of the data, i.e. this algorithm is optimal with respect to the bounds in Section 5.6.2.

Algorithm 10 BFS ER by distributed key-value store

input: Entity set E; one attribute store for each attribute with attribute value as key and entity ID as value; another entity store with entity ID as key and the complete entity data as value.
output: $E' = ER(E)$

```
 1: while not at the end of the entity store do
 2:   get next entity $e$ from entity store
 3:   erase $e$ from all attribute stores and put its records into $Q$
 4:   while $Q \neq \emptyset$ do
 5:     get next $r$ from $Q$
 6:     for $\ell = 1, \dots, k$ do
 7:       for all entities $e'$ with attribute matching $a_{r\ell}$ do
 8:         erase $e'$ from all attribute stores
 9:         update $e$ by $e \cup e'$ in the entity store.
10:         add all records of $e'$ to $Q$
```

The second implementation (Algorithm 10) performs breadth-first search as in [Corm 01, Section 22]. We use queues storing new records to be merged into the current entity. We iterate through records and, as in Algorithm 9, we obtain all matching entities by using the attribute stores. We immediately delete these records from the attribute stores and update the entity store to avoid infinite loops in a component.

We need to be able to list all attribute values matching $a_{r\ell}$ in line 4 of Algorithm 9 and line 7 of Algorithm 10. While this is trivial if the match function is the equality, our algorithms are suitable for similarity functions by an appropriate distributed similarity index. We may also trivially handle multi-value attributes, or, as in our case, name and maiden name by fetching both values from the corresponding attribute index.

In Algorithm 10 we may apply complex multi-attribute rules or learning based approaches as well, since we have access to the entire entity data via the entity index. For Algorithm 9 for example we handle a feature consisting of name, birth date and mother's name by indexing the concatenation of the three attributes as one single string.

The theoretical complexity of the algorithm is equal to that of BFS, i.e. $O(n \log n)$ [Corm 01]. The amount of communication equals to the total size of the data, i.e. this algorithm is also optimal with respect to the bounds in Section 5.6.2.

Finally we note that Algorithm 10 can take further advantage of parallelism: we may run the algorithm in several copies, each reading entities depending on

Algorithm 11 ER by Map-Reduce

input: Entity set E over a distributed file system.
output: $E' = ER(E)$

```
1: for $\ell = 1, \ldots, k$ do
2:     sort records $r$ by attribute value $a_{r\ell}$
3:     for all attribute values $a$ do
4:         for all pairs of records $r$, $r'$ with $a_{r\ell} = a_{r'\ell} = a$ do
5:             write $(ID(r), ID(r'))$ to graph $G$
6: Map-Reduce connected components$(G)$
7: sort records by component ID and merge groups of identical ID
```

a hash value of their ID. An additional step is needed to resolve the cases when two parallel runs accidentally reach the same merged entity starting out with an initial entity their data portion that we do not discuss here. If we distribute computation to t nodes, we may basically achieve a speedup factor of t if the data is randomly split to both the computing and the data store servers.

5.6.4 Algorithm Based on Map-Reduce

Our next algorithm is based on Hadoop [Whit 10], an open source implementation of the Map-Reduce framework [Dean 08]. It can be considered an extension of the Map-Reduce set similarity join of [Vern 10]; note that we may enhance our algorithm by techniques such as basic or indexed kernels from that result. Compared to a set similarity join, key difference is that we have to merge records over several attributes that will eventually require a connected component algorithm. This latter task can be solved by iterated matrix multiplication [Corm 01, Section 25]; a similar implementation is given in the Hadoop-based Pegasus framework [Kang 09].

Our Map-Reduce algorithm is split into two parts. The first part, Algorithm 11, iterates through all attributes. For each, it sorts attribute values and records all potential matches in a graph file. Then the connected component Algorithm 12 is called that assigns a component ID to all records. Finally, the last line of the main algorithm merges all records with the same ID. In this step, additional split heuristics can be implemented to undo some of the unnecessary merges similar to the algorithms in Section 5.6.3.

In Algorithm 11 we assign IDs to records as follows. If there are entities that consist of more than one record at start, we spit it into two records, both with the same ID. By a slight abuse of notation we may even handle entities with multiple values a and a' for certain attributes: we may place its ID multiple

Algorithm 12 Map-Reduce connected components

input: Graph G of record IDs.
output: Component ID for all record IDs.

```
1:  sort $G$ to form sequences $S_i = \{i, ID_i$, list of edges $(i, j)\}$
2:  change = true
3:  while change = true do
4:     change = false
5:     Map:
6:     for all IDs $i$ do
7:        for all IDs $j$ with $(i, j) \in S_i$ do
8:           emit $ID_i$ to reducer $j$
9:        emit entire $S_i$ to reducer $i$
10:    Reduce:
11:    for all reducers $j$ do
12:       $ID' =$ min of all $ID$ values received
13:       if $ID' < ID_j$ then
14:          change = true
15:          replace $ID_j$ by $ID$ in $S_j$
16:       write $S_j$
```

times, for both a, a' and possibly more, into an array to be sorted in line 2.

Note that placing an entity multiple times in line 2 we may also handle complex features such as the combination of a name and maiden name. In this case we consider name and maiden name as a single attribute with multiple values and proceed as above.

We describe the connected component Algorithm 12 in detail. The algorithm implements the matrix multiplication based all-pairs reachability algorithm of [Corm 01, Section 25] in a way similar to [Kang 09]. Two ingredients are the reduction of the problem to iterated matrix multiplication with a modified associative operation and the implementation of the matrix operation over Hadoop. For the first, let us replace addition by the minimum function and let

$$ID_j = \min(ID_j, \min_{i:(ij) \text{ is an edge}} \{ID_i\}). \tag{5.2}$$

In iteration s, this method selects the minimum value in the s step neighborhood of every record. If we record the fact that some ID_j decrease in an iteration, then we may terminate if there is no change.

Finally we show how to compute the matrix-vector multiplication type step of (5.2) by Map-Reduce. Starting at line 5, mapper i sends its current ID to

Algorithm 13 Bulk Synchronous Parallel (BSP) ER

input: Entity set E.
output: $E' = ER(E)$

```
1: Partition entities $E = \{E_1, \dots, E_d\}$ to data nodes $1 \dots d$.
2: for $i = 1, \dots, d$ do
3:     send $E_i$ to data node $i$
4: start master node and data nodes $1, \dots, d$
5: wait for termination
6: start BSP_connected_components on data nodes $1, \dots, d$
7: wait for termination
8: merge results from data nodes $1, \dots, d$ to $E'$
```

reducer j for all edges ij in the graph to prepare the data needed to compute (5.2). In addition, reducer j starting in line 10 must write data S_j suitable for the next matrix-vector multiplication iteration. In addition to ID_j, this S_j must contain the edges out of record j. For this purpose, mapper i sends its entire data S_i to reducer i, competing the description of the algorithm.

The running time of the algorithm is $O(\ell(n \log n)/t)$ for the ℓ mergesort operations over t servers and $O(sn/t)$ for connected components over t servers where s is the size of the largest component. The implementation transmits all data ℓ times during the sort operations, hence requires ℓ times more communication than the optimum. This algorithm is the least efficient in theory also since Hadoop introduces additional disk I/O operations when storing partial results.

5.6.5 A Bulk Synchronous Parallel Algorithm

In his seminal paper, Valiant [Vali 90] introduced the Bulk Synchronous Parallel framework for distributed algorithms. This framework is reintroduced as evolving infrastructures for distributed processing both proprietary as Google's Pregel [Male 10] and open source such as HAMA [Seo 10]. The idea is to divide the algorithm into phases divided by barriers. In each phase, nodes may produce messages to other nodes that they receive in the next phase.

In the following HAMA implementation, the first step of the main algorithm (Algorithm 13) is to distribute the data to servers (line 1). We may use a strong attribute or set of attributes for distribution that, similar to blocking ER algorithms, may result in significant speedup. Then a merge-sort algorithm is started over a master and d data servers in line 4. After initiating connected component computation in line 6 over the same data nodes, a last step is called that merges all records into the parent entity e at the data server holding e.

Algorithm 14 Data node algorithm for BSP ER

input: Entity set E_d such that each record corresponds to a unique entity.
output: E'_d and list L_e for each entity $e \in E'_d$

```
1:  $E'_d = ER(E_d)$
2:  for $\ell = 1, \ldots, k$ do
3:      start phase $2\ell - 1$
4:      sort records $r \in E'_d$ by attribute value $a_{r\ell}$
5:      for all attribute values $a$ in sorted order do
6:          $m_a \leftarrow \{ID(r_1), ID(r_2), \ldots\}$ for all records $r_i$ with $a_{r_i\ell} = a$
7:          send $m_a$ to master node
8:      send "end" to master node
9:      end phase $2\ell - 1$; start $2\ell$
10:     repeat
11:         receive $a, ID_1, d_1, ID_2, d_2, \ldots$ from master
12:         add $ID_1, d_1, ID_2, d_2, \ldots$ to entity list $L(e)$ holding $r \in e$ with $a_{r\ell} = a$
13:     until master node sends "end"
14:     end of phase $2\ell$
```

Algorithm 15 Master node algorithm for BSP ER

input: list of data nodes $1, \ldots, d$.

```
1:  for $\ell = 1, \ldots, k$ do
2:      start phase $2\ell - 1$
3:      for data nodes $p = 1, \ldots, d$ do
4:          receive $m_a^{(p)} = \{ID_1^{(p,a)}, ID_2^{(p,a)}, \ldots\}$ from node $p$
5:      end phase $2\ell - 1$; start $2\ell$
6:      for all attribute values $a$ in sorted order, merging the message queues
        from nodes $1, \ldots, d$ do
7:          if $s > 1$ for nodes $p_1, \ldots, p_s$ that submitted message $m_a^{(p)}$ for attribute
            value $a$ then
8:              for $i = 1, \ldots, s$ do
9:                  submit $a$ to node $p_i$
10:                 for $j = 1, \ldots, s$ with $j \neq i$ do
11:                     submit $(p_j, ID_t^{(p_j,a)})$ to node $p_i$ for all $t$
12:     end phase $2\ell$
13: terminate master node after phase $k$
```

The algorithm of data nodes (Algorithm 14) assumes, as in the Map-Reduce algorithm that entities with multiple values for certain attributes are split into records and uses the same $ID(r)$ notation for obtaining the entity ID. A farm of d such data nodes perform merge-sort with a master running Algorithm 15. Note that there may be more masters to speed up merge-sort by e.g. hashing the attribute value sets.

We perform merge-sort of the attribute values in iterations over attributes $\ell = 1, \ldots, k$. Algorithms 14 and 15 run in BSP phases $1, \ldots, 2k$. In the odd numbered phases data nodes send the values a along with the list m_a of entity IDs holding value a. The master collects all values shared between different data nodes. In the even numbered phases the master sends out these values a to each data node holding a, followed by the list of entity IDs and corresponding data node ID. Data nodes store these lists at the entity e holding a for the connected component algorithm.

Compared to the current HAMA based implementation, the above algorithm could be more efficient by using primitives for merging sorted lists. In theory the master(s) could balance the amount of data received from data nodes by, for the next attribute value a, immediately discarding if the value appears only once or emitting if matching is found. In this case the master could always requesting the next data from those that hold a. In its current state however we are even less efficient as communication in phases $1 \ldots 2k$ have be broken into smaller blocks so that all messages from data nodes fit into the master's memory.

Finally we turn to the connected component Algorithm 16 implemented as the classical "minimum-over-graph" algorithm. Algorithm 16 finds minimal ID of connected components of the graph in phases until termination. We proceed in BSP phases until there is no change in the entity IDs that we perform by marking entity e if its ID changes in the phase. In a phase, every marked entity sends its ID to its neighbors and unmarks itself. Receiving entities change their ID to minimum and mark themselves if there is a change. It is easy to see that in phase t, each entity gets the minimum ID of its neighbors within at most t steps.

The algorithm can be made more flexible by implementing more complex functions at the master nodes. For example it may collect all attribute values in a similarity index and send response to merge all similar entity values.

The running time of the algorithm is $O(ER(n/t))$ for the resolution of the initial data at each server where $ER(n)$ is the running time of an optimal sequential ER algorithm. Phases $1, \ldots, 2k$ communicate all data exactly once and run linear in the data transmitted, hence they altogether take $O(n)$ time. Finally the connected component algorithm takes $O(sn/t)$ time over t servers where s is the size of the largest component. By the assumption that s is very small, this algorithm is theoretically the best and also uses near optimal amount of commu-

Algorithm 16 BSP Connected Component algorithm

input: E'_d and list L_e for each entity $e \in E'_d$
output: parent entity ID and node $ID(e), P(e)$ for each entity $e \in E'_d$

```
1: sort $E'_d$ by ID
2: for all $(ID, p) \in L(e)$ for entities $e \in E'_d$ with $ID < ID(e)$ do
3:    mark $e$
4:    $ID'(e) \leftarrow ID$
5:    $P(e) = p$
6: while there are marks on entities do
7:    start next phase
8:    for all $(ID, p) \in L(e)$ for marked entities $e$ with $ID > ID(e)$ do
9:       send $ID(e), ID, P(e)$ to data node $p$
10:      unmark $e$
11:   while messages to the current data node $d$ exists do
12:      receive $ID, ID'$, p
13:      find $e$ with $ID(e) = ID$
14:      if $ID'(e) > ID'$ then
15:         mark $e$
16:         $ID'(e) \leftarrow ID$
17:         $P(e) = p$
18: start last phase: move all data to the lowest ID representative that will hold
    the resolved entity
```

nication.

5.6.6 Experiments

Experiments were performed on 15-server Linux farms containing identical dual core 3 GHz Pentium CPUs, 4 GB of main memory. Software versions were Sun Java 1.6, Project Voldemort 0.81, Hadoop 0.20.3, and a patched[2] HAMA 0.3.0. Voldemort was configured to use a replication factor of 1. We configured our systems to use all available internal memory. The largest test data sets do not fit in the memory of one node but still fit in the entire 15×4 GB of the cluster.

The data set is provided by AEGON Hungary Insurance Ltd. containing approximately 20 million client records, the same as in Section 5.5.5. The data set contained several attributes with different types, and 1.95 record per client in average. The size of the record set was also 1.7 GB in a flat CSV file.

We used the same sampling to obtain smaller subsets as in Section 5.5.5. We created larger data by replication and random permutation. In each replica, we added a version tag to all attributes so that the original structure of matches was preserved but no new matches were introduced between replicas.

In the experimental settings we used a real-world match condition composed of 9 attributes in 5 orthogonal matching features, also similar to the condition of Section 5.5.5. All of our algorithms produce the same exact results determined by the match conditions, therefore accuracy is not measured, only verified.

Comparison of Overall Running Times

Figure 5.10 and 5.11 depict the overall execution times including reading the input and finalizing the output. By these figures we observe the superiority of Map-Reduce: It was able to process one order of magnitude larger input sets than the others. Given the fixed duplication factor obtained by our data replication procedure, the running time up to 600 million records appears to be linear in input size, which makes the algorithm very useful in practice. The effect of varying duplication factor is tested later in Fig. 5.12. We did not lay stress on optimization, therefore further performance improvements could be achieved by fine tuning Hadoop and the Java code.

Theoretically, the BSP algorithm requires the least communication between nodes, therefore we expected similar or better behavior compared to the Map-Reduce version. BSP, however, performed poorly. The reasons can be both non-optimal coding, bad initial data distribution to nodes, or the experimental state of the Hama framework. BSP therefore requires further investigation.

[2]We fixed the distributed double barrier implementation.

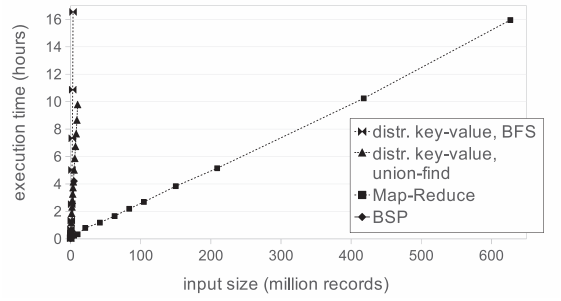


Figure 5.10: Execution time against input size for all algorithms

Re-implementing the algorithm using sockets could clear up the role of the infrastructure.

Our key-value store algorithms give the worst performance. This is no surprise since they are basically sequential in that they process the input one-by-one on a single main node. The main bottleneck turns out neither CPU nor I/O but the communication between the main node and the key-value store nodes. Distributing not just the data but also the computation would be therefore useful, however that requires a careful locking mechanism over the feature indexes and the entity store. Distributed key-value stores are slower than their non-distributed counterpart (in our case, BerkeleyDB) for small data (see Section 5.5.5 or [Sidl 11a] for related experiments). The main reason for using distribution is that the performance of single-node key-value stores falls drastically when reaching memory limits.

Although our BFS distributed key-value store algorithm is the slowest, it has the advantage of applying a general match logic: Attribute indexes provide candidates for a general match function. The main reason of the relatively poor performance of the BFS algorithm is that for larger data sets we get too many candidates for sophisticated matching conditions and indexing a single attribute of a sophisticated feature is simply not efficient enough.

Figure 5.12 shows the impact of match count on the execution times of the Map-Reduce algorithm. As we expected, increasing match count (less hidden entity) results in larger running times: more edges appear in the match graph and more iterations have to be performed when finding connected components.

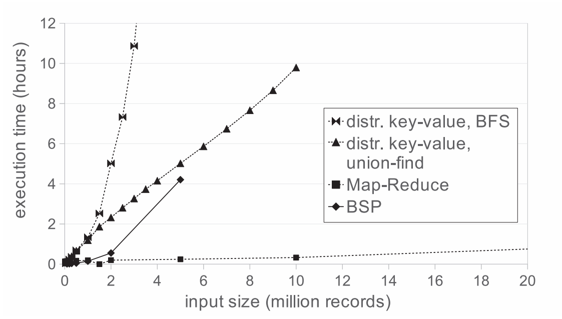


Figure 5.11: Execution time against input size for all algorithms, below 20 million records

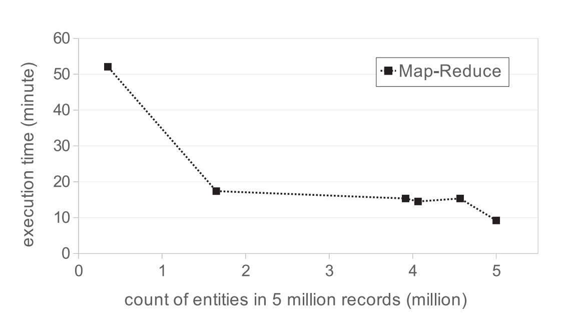


Figure 5.12: Execution times plotted against the number of merged entities in 5 million records

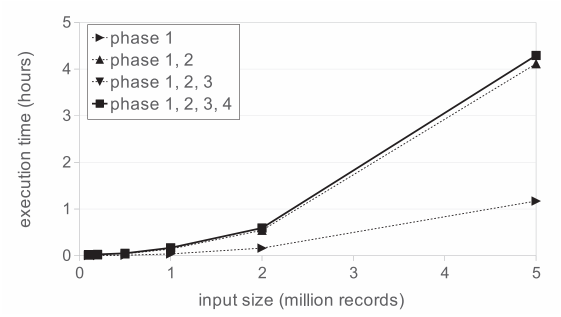


Figure 5.13: Cumulative execution times of the distinct phases in the BSP algorithm

Running Times of Algorithm Phases

Except for the key-value store procedures, all of our algorithms can be separated to different phases. Our first algorithm, the Union-Find key-value store one finds all matchings first. Then in a second round it produces the output based on the union-find store. The time needed for the second round was approximately 20% of the first phase in all settings.

For the BSP algorithm, Figure 5.13 depicts the time required by its phases. In Phase 1, we read the input and resolve the initial datasets per node by an in-memory ER implementation similar to the union-find variation. Phase 2 covers the main process of master and data nodes and Phase 3 the connected component search. The output is finalized in Phase 4. The running time share of the phases seems to be fixed over our range of input sizes with most of the time devoted to Phase 2, the main part of the procedure.

Figure 5.14 shows running times of the Map-Reduce implementation phases. This implementation first sorts the data by feature values and produces the edge matrix in Phase 2. The edge matrix is used to find connected components in Phase 3, taking the majority of execution time. Note the complementarity with the BSP algorithm where the time for connected component computation is negligible. In Phase 4, we produce the output based on the list of the connected components.

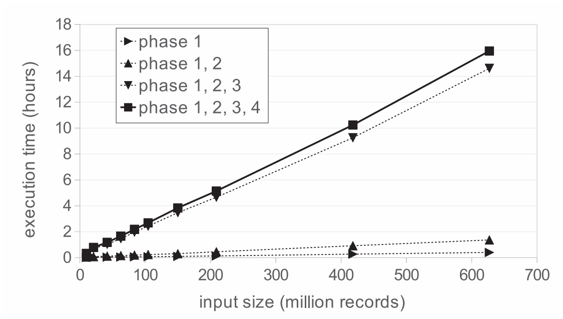


Figure 5.14: Cumulative execution times of the distinct phases in the Map-Reduce algorithm

5.7 Conclusions

ER data models and algorithms were developed and presented for relational databases. These algorithms scale well, are practical and more efficient then earlier similar algorithms.

We developed new, practical postulates for relational databases, based on the generic ER formulation, enabling our DB-GER variations to perform significantly better than previous Swoosh algorithms. DB-GER also proved to be useful in practice in an insurance customer integration scenario. The applicability is demonstrated by experiments and by a case study: AEGON Hungary, a large Hungarian insurance company successfully applies the methods and algorithms for years in a client data warehouse application.

We think that standard SQL is flexible enough to build practical match and merge functions; the formal capture of the SQL match and merge function class remains however unclear.

Efficient entity feature index based algorithms are developed and presented for ER without memory bounds and using external memory if necessary. The constructs used for indexes, features and blocking enable solving a wide range of practical problems. Usefulness of building and maintaining entity indexes for search time reduction is demonstrated by experiments and a case study. The algorithms were tested against a similar insurance client dataset.

We believe that appropriate selection of indexing tools and algorithms can further improve performance. Our algorithms may include arbitrary indexing and search solutions, both for exact and for similarity-based matching. The class of suitable tools depends on the data set, on the features, on the match logic and on the architectural environment as well. Another possible step towards ER on big data is making the algorithms distributed.

The distributed algorithms described in 5.6 all solve the ER problem over three different types of distributed software architectures efficiently. Applicability and efficiency is demonstrated by experiments relying on the insurance client data warehouse environment. In our experiments, distributed ER algorithms also proved to be useful in practice. To our best knowledge, the algorithm based on MapReduce scales the best by the time of writing. We investigated communication complexity bounds and the range of ER tasks that can be described by the given model.

Distributed key-value stores provide the easiest extensions towards similarity search and fuzzy match functions: Instead of lexicographical or numeric indexes, we may simply use distributed similarity search structures. This solution is however the slowest of the alternatives. A further possible way of parallel processing have to be studied: If the resolved entity set resides in the key-value store, then parallel resolving tasks can speed up the entire executions. However, the soundness of this parallelization have to be ensured by locking mechanisms not covered in this paper.

Hadoop is apparently a mature Map-Reduce infrastructure already capable of efficiently implementing ER algorithms. For Bulk Synchronous Parallel algorithms, there is no open source infrastructure of similar level of maturity yet. We demonstrated that HAMA, an incubatory project is capable of supporting ER algorithms and we expect BSP implementations may eventually be superior since they reduce cross-server communication and may completely eliminate slow disk I/O operations. Also, partitioning along a strong feature may result in significant speedup and our solution may also combine well with feature-based blocking.

In future work these algorithms and index alternatives should be tested in other settings, e.g. on conceptually different data sets, or with similarity-based feature matching. Implementing the BSP algorithm using sockets or other basic-level communication could clear up the role of the infrastructure and show the capability of the algorithm for practical ER problems.

The probabilistic model promise good applicability based on our experiments. As a future work, the model and methods of Section 5.3.6 should be applied to our scalable algorithms.

We used slightly different environments to evaluate our ER algorithms, therefore precise comparison is not feasible. Nevertheless, Figure 5.15 and 5.16 provide an inaccurate, but useful overview of the new ER algorithms described in the thesis. Scalability of all algorithms are shown on insurance client data (the

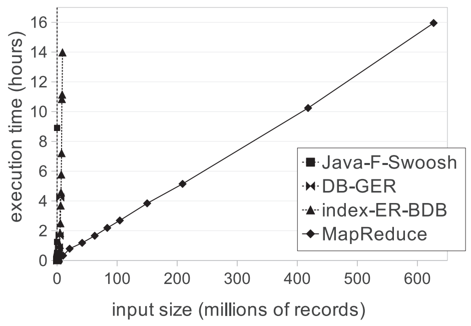


Figure 5.15: Scalability of our entity resolution algorithms.

two sides differ only in the maximum of the horizontal axis).

- Java-F-Swoosh: an own Java implementation of the best previously known generic entity resolution algorithm (F-Swoosh [Benj 09]),
- DB-GER: our best relational entity resolution algorithm, based on a commercial relational database,
- index-ER-BDB: our best efficient indexing algorithm built on Java and Berkeley DB,
- MapReduce: our best distributed ER algorithm using 15 computer nodes and Hadoop implementation.

Previous results assume 10 or 100 thousand records as input. To give an example, in [Kirs 10] a distributed algorithm is described with similarity-based matching, tested on 114 thousand records. Algorithms of [Weis 08] were tested closest to our database size with 10 million records. In comparison, our 600 million input records indicates a significant improvement in scalability.

5.8 My Contribution

Entity resolution methods for relational databases (Section 5.3.2 and 5.4) were presented on an ADBIS conference as a single-author paper that subsequently appeared in an LNCS volume for selected and revised papers of the conference [Sidl 09].

The results on entity resolution with heavy indexing (parts of Section 5.3.3 and Section 5.5) were presented on an ADBIS conference as a single-author work, and appeared in the conference proceedings [Sidl 11a].

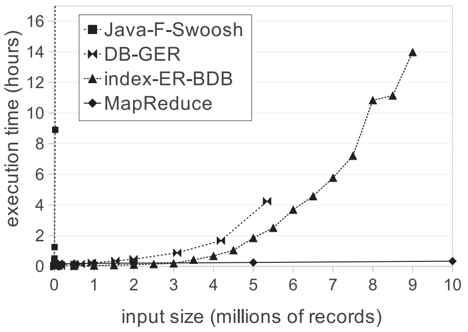


Figure 5.16: Scalability of our entity resolution algorithms, below 20 million records.

Results of Section 5.6 on distributed ER appeared in [Sidl 11b]. This paper represents a joint work with András Garzó, András Molnár and András A. Benczúr. My contributions were formulating the problem, raising the idea of the three different parallelization methods, contributing to detailed algorithm design, implementing the key-value store algorithms, generating the experimental dataset and coordinating the experiments. I also took part in the design of the other types of distributed algorithms. The complexity bounds originate from András A. Benczúr.

Chapter 6

Summary and Conclusions

The thesis described results extending business intelligence applications and methods to meet new needs, discussing selected topics of BI. Results were motivated by real business requirements and needs that cannot, or can be hardly resolved with existing tools. The methods correspond to the common research practice of the area, and were presented to the research community on international conferences and workshops.

The new architecture type of Chapter 3 can integrate relational data warehouses and colum-oriented storages in a cost-effective way, enabling long-term storage and data mining. The practical applicability of the new architecture was demonstrated by a prototype system, where a commercial database management system and a data mining framework of our own were integrated.

A pattern-growth frequent itemset mining algorithm is presented in Chapter 4 for relational database environments based on SQL operations. This algorithm efficiently utilizes the facilities provided by the database server, and fits the relational data model and specialties of the environment.

New entity resolution data models and algorithms were developed in Chapter 5. Algorithms for relational databases scale well, are practical and more efficient then earlier similar algorithms. Efficient entity index-based algorithms were developed and presented for entity resolution without memory bounds and using external memory if necessary. We demonstrated that the problem can be efficiently solved by deploying distributed computing environments. To our best knowledge, the algorithm based on MapReduce scales the best by the time of writing. Applicability of our algorithms are demonstrated by experiments relying on insurance client data data.

Bibliography

[Abad 06] D. J. Abadi, S. R. Madden, and M. C. Ferreira. “Integrating Compression and Execution in Column-Oriented Database Systems”. In: *SIGMOD ’06: Proceedings of the ACM SIGMOD International Conference on Management of Data*, ACM, 2006.

[Agra 94] R. Agrawal and R. Srikant. “Fast Algorithms for Mining Association Rules in Large Databases”. In: *VLDB ’94: Proceedings of the 20th International Conference on Very Large Data Bases*, pp. 487–499, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[Alon 99] N. Alon, Y. Matias, and M. Szegedy. “The space complexity of approximating the frequency moments”. *Journal of Computer and System Sciences*, Vol. 58, No. 1, pp. 137–147, 1999.

[Babc 02] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. “Models and issues in data stream systems”. In: *PODS ’02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 1–16, ACM Press, New York, NY, USA, 2002.

[Bara 05] E. Baralis, T. Cerquitelli, and S. Chiusano. “Index Support for Frequent Itemset Mining in a Relational DBMS”. In: *ICDE ’05: Proceedings of the 21st International Conference on Data Engineering (ICDE’05)*, pp. 754–765, IEEE Computer Society, 2005.

[Benc 05] A. A. Benczúr, K. Csalogány, A. Lukács, B. Rácz, C. Sidló, M. Uher, and L. Végh. “An Architecture for Mining Massive Web Logs with Experiments”. In: *In Proceedings of the HUBUSKA Open Workshop on Generic Issues of Knowledge Technologies*, 2005.

[Benj 07] O. Benjelloun, H. Garcia-Molina, H. Gong, H. Kawai, T. E. Larson, D. Menestrina, and S. Thavisomboon. “D-Swoosh: A Family of Algorithms for Generic, Distributed Entity Resolution”. In: *Proc. of the 27th Int. Conf. on Distributed Computing Systems*, IEEE, 2007.

[Benj 09] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. “Swoosh: a generic approach to entity resolution”. *VLDB J.*, Vol. 18, No. 1, pp. 255–276, 2009.

[Bent 02] F. Bentayeb and J. Darmont. “Decision Tree Modeling with Relational Views”. In: *ISMIS ’02: Proceedings of the 13th International Symposium on Foundations of Intelligent Systems*, pp. 423–431, Springer-Verlag, 2002.

[Bhat 06a] I. Bhattacharya and L. Getoor. “A Latent dirichlet model for unsupervised entity resolution”. *SIAM International Conference on Data Mining*, pp. 47–58, 2006.

[Bhat 06b] I. Bhattacharya, L. Getoor, and L. Licamele. “Query-time entity resolution”. *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 529–534, 2006.

[Bhat 07] I. Bhattacharya and L. Getoor. “Collective entity resolution in relational data”. *ACM Trans. Knowl. Discov. Data*, Vol. 1, No. 1, p. 5, 2007.

[Bhat 08] I. Bhattacharya, S. Godbole, and S. Joshi. “Structured entity identification and document categorization: two tasks with one joint model”. In: *KDD ’08: Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 25–33, ACM, New York, NY, USA, 2008.

[Bile 03] M. Bilenko and R. Mooney. “Adaptive duplicate detection using learnable string similarity measures”. *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 39–48, 2003.

[Bott 04] M. Botta, J.-F. Boulicaut, C. Masson, and R. Meo. “Query Languages Supporting Descriptive Rule Mining: A Comparative Study.”. In: *Database Support for Data Mining Applications*, pp. 24–51, 2004.

[Boul 99] J.-F. Boulicaut, M. Klemettinen, and H. Mannila. “Modeling KDD Processes within the Inductive Database Framework”. In: *DaWaK ’99: Proceedings of the First International Conference on Data Warehousing and Knowledge Discovery*, pp. 293–302, Springer-Verlag, 1999.

[Brod 97] A. Z. Broder. “On the Resemblance and Containment of Documents”. In: *Proceedings of the Compression and Complexity of Sequences (SEQUENCES’97)*, pp. 21–29, IEEE Computer Society, 1997.

[Cha 11] S. K. Cha. “SAP HANA: Breaking Vertical and Horizontal Tiers in Enterprise with High-Performance Distributed In-Memory Database”. http://wwwdb.inf.tu-dresden.de/birte2011/invited.html#invited2, 2011. 5th International Workshop on Business Intelligence for the Real Time Enterprise (BIRTE 2011). Invited Talk. [last accessed: 27 August 2011].

[Chan 06] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. “Bigtable: a distributed storage system for structured data”. In: *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, pp. 15–15, USENIX Association, Berkeley, CA, USA, 2006.

[Char 00] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. “Towards estimation error guarantees for distinct values”. In: *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 268–279, ACM, 2000.

[Chau 07] S. Chaudhuri, A. D. Sarma, V. Ganti, and R. Kaushik. “Leveraging aggregate constraints for deduplication”. In: *SIGMOD ’07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pp. 437–448, ACM, New York, NY, USA, 2007.

[Chri 04] P. Christen, T. Churches, and M. Hegland. “Febrl - A Parallel Open Source Data Linkage System”. In: *PAKDD*, pp. 638–647, Springer, 2004.

[Chri 08] P. Christen. “Automatic record linkage using seeded nearest neighbour and support vector machine classification”. In: *KDD ’08:*

Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 151–159, ACM, New York, NY, USA, 2008.

[Chri 09a] P. Christen. “Development and user experiences of an open source data cleaning, deduplication and record linkage system”. *ACM SIGKDD Explorations Newsletter*, Vol. 11, No. 1, pp. 39–48, 2009.

[Chri 09b] P. Christen, R. Gayler, and D. Hawking. “Similarity-aware indexing for real-time entity resolution”. In: *Proc. of the 18th ACM conference on Information and knowledge management*, pp. 1565–1568, ACM, 2009.

[Chri 11] P. Christen. “A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication”. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 99, No. PrePrints, 2011.

[Corm 01] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT press, 2001.

[Dean 08] J. Dean and S. Ghemawat. “MapReduce: Simplified data processing on large clusters”. *Communications of the ACM*, Vol. 51, No. 1, pp. 107–113, 2008.

[DeCa 07] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. “Dynamo: amazon’s highly available key-value store”. *ACM SIGOPS Operating Systems Review*, Vol. 41, No. 6, pp. 205–220, 2007.

[Desi 01] “Designing data marts for data warehouses”. *ACM Trans. Softw. Eng. Methodol.*, Vol. 10, No. 4, pp. 452–483, 2001.

[Dong 05] X. Dong, A. Halevy, and J. Madhavan. “Reference reconciliation in complex information spaces”. In: *Proc. of the 2005 ACM SIGMOD international conference on Management of data*, ACM, 2005.

[Dono 04] D. Donoho. “Neighborly Polytopes and Sparse Solution of Underdetermined Linear Equations”. Tech. Rep., Stanford University, 2004.

[Elma 07] A. Elmagarmid, P. Ipeirotis, and V. Verykios. “Duplicate Record Detection: A Survey”. *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–16, 2007.

[Fell 69] I. Fellegi and A. Sunter. "A theory for record linkage". *Journal of the American Statistical Association*, Vol. 64, No. 328, pp. 1183–1210, 1969.

[Fitz 04] B. Fitzpatrick. "Distributed caching with memcached". *Linux journal*, Vol. 2004, No. 124, p. 5, 2004.

[Flaj 85] P. Flajolet and G. Nigel Martin. "Probabilistic counting algorithms for data base applications". *Journal of Computer and System Sciences*, Vol. 31, No. 2, pp. 182–209, 1985.

[Flor 11] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata. "Column-oriented storage techniques for MapReduce". *Proc. VLDB Endow.*, Vol. 4, pp. 419–429, April 2011.

[Freq] "Frequent Itemset Mining Implementations Repository". http://fimi.cs.helsinki.fi/. [last accessed: 2 August 2011].

[Geto 05] L. Getoor and C. Diehl. "Link mining: a survey". *ACM SIGKDD Explorations Newsletter*, Vol. 7, No. 2, pp. 3–12, 2005.

[Grah 04] G. Grahne and J. Zhu. "Mining Frequent Itemsets from Secondary Memory". In: *ICDM '04: Proceedings of the Fourth IEEE International Conference on Data Mining (ICDM'04)*, pp. 91–98, IEEE Computer Society, Washington, DC, USA, 2004.

[Grav 01] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. "Approximate string joins in a database (almost) for free". *Proceedings of the 27th International Conference on Very Large Data Bases*, pp. 491–500, 2001.

[Guo 10] S. Guo, X. L. Dong, D. Srivastava, and R. Zajac. "Record linkage with uniqueness constraints and erroneous values". *Proc. VLDB Endow.*, Vol. 3, pp. 417–428, September 2010.

[Hall 08] R. Hall, C. Sutton, and A. McCallum. "Unsupervised deduplication using cross-field dependencies". In: *KDD '08: Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 310–317, ACM, New York, NY, USA, 2008.

[Han 00] J. Han, J. Pei, and Y. Yin. "Mining frequent patterns without candidate generation". In: *Proceedings of the 2000 ACM SIGMOD* *international conference on Management of data*, pp. 1–12, ACM Press, 2000.

[Han 05] H. Han, W. Xu, H. Zha, and C. Giles. "A hierarchical naive Bayes mixture model for name disambiguation in author citations". *Proceedings of the 2005 ACM symposium on Applied computing*, pp. 1065–1069, 2005.

[Han 98] J. Han. "Towards on-line analytical mining in large databases". *SIGMOD Rec.*, Vol. 27, No. 1, pp. 97–107, 1998.

[Hari 08] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. "OLTP through the looking glass, and what we found there". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 981–992, ACM, New York, NY, USA, 2008.

[Hern 98] M. Hernández and S. Stolfo. "Real-world data is dirty: Data cleansing and the merge/purge problem". *Data mining and knowledge discovery*, Vol. 2, No. 1, pp. 9–37, 1998.

[Hirs 79] D. Hirschberg, A. Chandra, and D. Sarwate. "Computing connected components on parallel computers". *Communications of the ACM*, Vol. 22, No. 8, pp. 461–464, 1979.

[Hout 95] M. Houtsma and A. Swami. "Set-oriented data mining in relational databases". *Data Knowl. Eng.*, Vol. 17, No. 3, pp. 245–262, 1995.

[IGlu 11] IGlue. "A Semantic Web Information Dissemination Tool". http://iglue.com/, 2011. [last accessed: 12 Aug 2011].

[Imie 96] T. Imielinski and H. Mannila. "A database perspective on knowledge discovery". *Commun. ACM*, Vol. 39, No. 11, pp. 58–64, 1996.

[ISO 92] "ISO-ANSI SQL-2 Database Language Standard, X3H2-92-154". 1992.

[Josh 03] K. P. Joshi, A. Joshi, and Y. Yesha. "On Using a Warehouse to Analyze Web Logs". *Distrib. Parallel Databases*, Vol. 13, No. 2, pp. 161–180, 2003.

[Kall 08] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and

D. J. Abadi. “H-store: a high-performance, distributed main memory transaction processing system”. *Proc. VLDB Endow.*, Vol. 1, No. 2, pp. 1496–1499, 2008.

[Kaly 92] B. Kalyanasundaram and G. Schintger. “The probabilistic communication complexity of set intersection”. *SIAM Journal on Discrete Mathematics*, Vol. 5, p. 545, 1992.

[Kang 09] U. Kang, C. Tsourakakis, and C. Faloutsos. “PEGASUS: A peta-scale graph mining system implementation and observations”. In: *Ninth IEEE International Conference on Data Mining*, pp. 229–238, IEEE, 2009.

[Kawa 06] H. Kawai, H. Garcia-Molina, O. Benjelloun, D. Menestrina, E. Whang, and H. Gong. “P-Swoosh: Parallel Algorithm for Generic Entity Resolution”. Tech. Rep., Stanford, 2006.

[Kim 07] H.-s. Kim and D. Lee. “Parallel linkage”. In: *Proc. of the sixteenth ACM conference on Conference on information and knowledge management*, ACM, 2007.

[Kimb 11] R. Kimball and M. Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling, Second Edition*. John Wiley & Sons, 2011.

[Kirs 10] T. Kirsten, L. Kolb, M. Hartung, A. Gross, H. Köpcke, and E. Rahm. “Data Partitioning for Parallel Entity Matching”. *Computing Research Repository*, 2010.

[Kona 04] H. Kona and S. Chakravarthy. “Partitioned Approach to Association Rule Mining over Multiple Databases.”. pp. 320–330, 2004.

[Kopc 08] H. Köpcke and E. Rahm. “Training selection for tuning entity matching”. In: *QDB/MUD*, pp. 3–12, 2008.

[Kopc 10a] H. Köpcke and E. Rahm. “Frameworks for entity matching: A comparison”. *Data Knowl. Eng.*, Vol. 69, pp. 197–210, February 2010.

[Kopc 10b] H. Köpcke, A. Thor, and E. Rahm. “Evaluation of entity resolution approaches on real-world match problems”. *Proc. VLDB Endow.*, Vol. 3, pp. 484–493, September 2010.

[Lars 11] P.-A. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou. "SQL server column store indexes". In: *Proceedings of the 2011 international conference on Management of data*, pp. 1177–1184, ACM, New York, NY, USA, 2011.

[Leav 10] N. Leavitt. "Will NoSQL Databases Live Up to Their Promise?". *Computer*, Vol. 43, No. 2, pp. 12 –14, 2010.

[Li 04] W. Li and A. Mozes. "Computing Frequent Itemsets Inside Oracle 10g". In: *VLDB'04*, pp. 1253–1256, 2004.

[MacL 04] J. MacLennan. "SQL Server 2005: Unearth the New Data Mining Features of Analysis Services 2005". *MSDN Magazine*, Vol. 19, No. 9, 2004.

[Male 10] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. "Pregel: a system for large-scale graph processing". In: *Proc. of the 2010 Int. Conf. on Management of Data*, pp. 135–146, ACM, 2010.

[Mart 06] C. Martens. "Business intelligence at age 17". http://www.computerworld.com/s/article/266298/BI_at_age_17, 2006. [last accessed: 7 July 2011].

[McCa 95] J. McCarthy and W. Lehnert. "Using decision trees for coreference resolution". *Proceedings of the Fourteenth International Conference on Artificial Intelligence*, pp. 1050–1055, 1995.

[Mene 06] D. Menestrina, O. Benjelloun, and H. Garcia-Molina. "Generic entity resolution with data confidences". *CleanDB Workshop*, pp. 25–32, 2006.

[Mene 10] D. Menestrina, S. E. Whang, and H. Garcia-Molina. "Evaluating entity resolution results". *Proc. VLDB Endow.*, Vol. 3, pp. 208–219, September 2010.

[Meo 98] R. Meo, G. Psaila, and S. Ceri. "A Tightly-Coupled Architecture for Data Mining". In: *ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering*, pp. 316–323, IEEE Computer Society, Washington, DC, USA, 1998.

[Mish 03] P. Mishra and S. Chakravarthy. "Performance Evaluation of SQL-OR Variants for Association Rule Mining". *Lecture Notes in Computer Science*, Vol. 2737 / 2003, pp. 288 – 298, 2003.

[Mong 97] A. Monge and C. Elkan. “An efficient domain-independent algorithm for detecting approximately duplicate database records”. In: *SIGMOD DMKD*, 1997.

[Moss 03] L. Moss and S. Atre. *Business intelligence roadmap: the complete project lifecycle for decision-support applications*. *Addison-Wesley information technology series*, Addison-Wesley, 2003.

[Netz 01] A. Netz, S. Chaudhuri, U. M. Fayyad, and J. Bernhardt. “Integrating Data Mining with SQL Databases: OLE DB for Data Mining”. In: *Proceedings of the 17th International Conference on Data Engineering*, pp. 379–387, IEEE Computer Society, 2001.

[Pei 01] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang. “H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases”. In: *Proceedings of the 2001 IEEE International Conference on Data Mining*, pp. 441–448, IEEE Computer Society, 2001.

[Poss 03] M. Pöss and D. Potapov. “Data Compression in Oracle.”. In: *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases*, pp. 937–947, Morgan Kaufmann, 2003.

[Proj] Project Voldemort. “A distributed key-value storage system”. `http://project-voldemort.com/`. [last accessed: 14 August 2011].

[Racz 04] B. Rácz and A. Lukács. “High Density Compression of Log Files.”. In: *Data Compression Conference*, p. 557, 2004.

[Racz 07] B. Rácz, C. I. Sidló, A. Lukács, and A. A. Benczúr. “Two-Phase Data Warehouse Optimized for Data Mining”. *Lecture Notes in Computer Science*, Vol. 4365, p. 63, 2007.

[Rant 03] R. Rantzau. “Processing frequent itemset discovery queries by division and set containment join operators”. In: *Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, pp. 20–27, ACM Press, 2003.

[Rant 04] R. Rantzau. “Frequent Itemset Discovery with SQL Using Universal Quantification.”. In: *Database Support for Data Mining Applications*, pp. 194–213, 2004.

[Sara 02] S. Sarawagi and A. Bhamidipaty. “Interactive deduplication using active learning”. *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 269–278, 2002.

[Sara 98] S. Sarawagi, S. Thomas, and R. Agrawal. “Integrating association rule mining with relational database systems: alternatives and implications”. In: *SIGMOD ’98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pp. 343–354, ACM Press, New York, NY, USA, 1998.

[Satt 01] K.-U. Sattler and O. Dunemann. “SQL database primitives for decision tree classifiers”. In: *CIKM ’01: Proceedings of the tenth international conference on Information and knowledge management*, pp. 379–386, ACM Press, 2001.

[Sava 95] A. Savasere, E. Omiecinski, and S. B. Navathe. “An Efficient Algorithm for Mining Association Rules in Large Databases”. In: *Proceedings of the 21th International Conference on Very Large Data Bases*, pp. 432–444, Morgan Kaufmann Publishers Inc., 1995.

[Seo 10] S. Seo, E. Yoon, J. Kim, S. Jin, J. Kim, and S. Maeng. “HAMA: An Efficient Matrix Computation with the MapReduce Framework”. In: *2nd IEEE International Conference on Cloud Computing Technology and Science*, pp. 721–726, IEEE, 2010.

[Shan 04] X. Shang, K.-U. Sattler, and I. Geist. “SQL Based Frequent Pattern Mining with FP-Growth.”. In: *INAP/WLP*, pp. 32–46, 2004.

[Sidl 05a] C. I. Sidló. “SQL-based Frequent Itemset Mining materials”. http://scs.web.elte.hu/sqlfim/, 2005. [last accessed: 10 July 2011].

[Sidl 05b] C. I. Sidló and A. Lukács. “Shaping SQL-Based Frequent Pattern Mining Algorithms”. In: *Knowledge Discovery in Inductive Databases: 4th International Workshop, KDID 2005, Revised Selected and Invited Papers*, pp. 188–201, Springer-Verlag, 2005.

[Sidl 09] C. I. Sidló. “Generic Entity Resolution in Relational Databases”. In: J. Grundspenkis, T. Morzy, and G. Vossen, Eds., *Advances in Databases and Information Systems*, pp. 59–73, Springer, 2009.

[Sidl 11a] C. I. Sidló. “Entity Resolution with Heavy Indexing”. In: *Proceedings of the 2011 International Conference on Advances in Databases and Information Systems*, 2011.

[Sidl 11b] C. I. Sidló, A. Garzó, A. Molnár, and A. A. Benczúr. “Infrastructures and Bounds for Distributed Entity Resolution”. In: *9th International Workshop on Quality in Databases In conjunction with VLDB 2011*, 2011.

[Ston 05] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O’Neil, P. O’Neil, A. Rasin, N. Tran, and S. Zdonik. “C-store: a column-oriented DBMS”. In: *VLDB ’05: Proceedings of the 31st international conference on Very large data bases*, pp. 553–564, VLDB Endowment, 2005.

[Ston 10a] M. Stonebraker. “SQL databases v. NoSQL databases”. *Commun. ACM*, Vol. 53, pp. 10–11, April 2010.

[Ston 10b] M. Stonebraker. “Why Enterprises Are Uninterested in NoSQL”. http://cacm.acm.org/blogs/blog-cacm/99512-why-enterprises-are-uninterested-in-nosql, 2010. [last accessed: 10 Aug 2011].

[Swei 02] M. Sweiger, J. Langston, H. Lombard, and M. R. Madsen. *Clickstream Data Warehousing*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[Talb 10] J. R. Talburt. *Entity Resolution and Information Quality*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st Ed., 2010.

[Thom 99] S. Thomas and S. Chakravarthy. “Performance Evaluation and Optimization of Join Queries for Association Rule Mining”. In: *Proceedings of the First International Conference on Data Warehousing and Knowledge Discovery*, pp. 241–250, Springer-Verlag, 1999.

[TPC 06] “TPC Benchmark H Standard Specification Revision 2.3.0”. http://www.tpc.org/tpch/spec/tpch2.3.0.pdf, 2006. [last accessed: 4 Aug 2011].

[Vali 90] L. Valiant. “A bridging model for parallel computation”. *Communications of the ACM*, Vol. 33, No. 8, pp. 103–111, 1990.

[Vern 10] R. Vernica, M. Carey, and C. Li. “Efficient parallel set-similarity joins using mapreduce”. In: *Proc. of the 2010 Int. Conf. on Management of Data*, pp. 495–506, ACM, 2010.

[Wang 02] K. Wang, L. Tang, J. Han, and J. Liu. “Top Down FP-Growth for Association Rule Mining”. In: *PAKDD ’02: Proceedings of the 6th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*, pp. 334–340, Springer-Verlag, London, UK, 2002.

[Weis 08] M. Weis, F. Naumann, U. Jehle, J. Lufter, and H. Schuster. “Industry-scale duplicate detection”. *Proc. of the VLDB Endow.*, Vol. 1, No. 2, pp. 1253–1264, 2008.

[Whan 09] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. “Entity resolution with iterative blocking”. In: *Proc. of the 35th Int. Conf. on Management of Data*, pp. 219–232, ACM, 2009.

[Whan 10] S. E. Whang and H. Garcia-Molina. “Entity resolution with evolving rules”. *Proc. VLDB Endow.*, Vol. 3, pp. 1326–1337, September 2010.

[Whit 10] T. White. *Hadoop: The Definitive Guide*. Yahoo Press, 2010.

[Wick 08] M. L. Wick, K. Rohanimanesh, K. Schultz, and A. McCallum. “A unified approach for schema matching, coreference and canonicalization”. In: *KDD ’08: Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 722–730, ACM, New York, NY, USA, 2008.

[Yako 10] M. Yakout, A. K. Elmagarmid, H. Elmeleegy, M. Ouzzani, and A. Qi. “Behavior based record linkage”. *Proc. VLDB Endow.*, Vol. 3, pp. 439–448, September 2010.

[Yosh 00] T. Yoshizawa, I. Pramudiono, and M. Kitsuregawa. “SQL Based Association Rule Mining Using Commercial RDBMS (IBM DB2 UBD EEE)”. In: *Proceedings of the Second International Conference on Data Warehousing and Knowledge Discovery*, pp. 301–306, Springer-Verlag, 2000.