# Integration of Heterogeneous Databases: Discovery of Meta-Information and Maintenance of Schema-Restructuring Views

by

Andreas Koeller

A Dissertation

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

in

Computer Science

by

December 14, 2001

**APPROVED:**

| | |
|---|---|
| Prof. Elke A. Rundensteiner<br>Advisor | Prof. Nabil I. Hachem<br>Committee Member |
| Prof. Carolina Ruiz<br>Committee Member | Prof. David C. Brown<br>Committee Member |
| Prof. Dr. rer. nat. habil. Gunter Saake<br>External Committee Member<br>University of Magdeburg | Prof. Micha Hofri<br>Head of Department |

# Abstract

In today's networked world, information is widely distributed across many independent databases in heterogeneous formats. Integrating such information is a difficult task and has been adressed by several projects. However, previous integration solutions, such as the *EVE*-Project, have several shortcomings. Database contents and structure change frequently, and users often have incomplete information about the data content and structure of the databases they use. When information from several such insufficiently described sources is to be extracted and integrated, two problems have to be solved: How can we discover the structure and contents of and interrelationships among unknown databases, and how can we provide durable integration views over several such databases? In this dissertation, we have developed solutions for those key problems in information integration.

The first part of the dissertation addresses the fact that knowledge about the interrelationships between databases is essential for any attempt at solving the information integration problem. We are presenting an algorithm called $\mathsf{FIND}_2$ based on the clique-finding problem in graphs and $k$-uniform hypergraphs to discover redundancy relationships between two relations.

Furthermore, the algorithm is enhanced by heuristics that significantly reduce the search space when necessary. Extensive experimental studies on the algorithm both with and without heuristics illustrate its effectiveness on a variety of real-world data sets.

The second part of the dissertation addresses the durable view problem and presents the first algorithm for incremental view maintenance in schema-restructuring views. Such views are essential for the integration of heterogeneous databases. They are typically defined in schema-restructuring query languages like *SchemaSQL*, which can transform schema into data and vice versa, making traditional view maintenance based on differential queries impossible. Based on an existing algebra for *SchemaSQL*, we present an update propagation algorithm that propagates updates along the query algebra tree and prove its correctness. We also propose optimizations on our algorithm and present experimental results showing its benefits over view recomputation.

# Acknowledgements

I would like to thank my advisor, Prof. Elke A. Rundensteiner, for her guidance, advice and support during my time at WPI. Thanks also to the other members of my dissertation committee, Prof. Carolina Ruiz, Prof. David Brown, Prof. Nabil Hachem, and Prof. Gunter Saake, who provided valuable feedback and suggestions during my research that were useful in guiding my work.

My colleagues in the DSRG group, in particular Kajal Claypool, Li Chen, and Xin Zhang, with whom I have shared an office for the past four years, contributed to my work and progress of this dissertation in many ways.

Prof. Micha Hofri gave me the opportunity to teach at WPI, which provided valuable experience to me and also a welcome occasional distraction from the research that I was doing. I also would like to thank the other faculty in the CS department at WPI, in particular Profs. Stanley Selkow, George Heineman, and Kathi Fisler, for valuable discussions and help with many issues over the past few years. Michael Voorhis and Jesse Banning provided technical support and (unfortunately necessary) backups of my machine.

Finally, I want to thank my fiancée Hajira Begum, for her love and support and the great amount of understanding she showed whenever I cancelled yet another weekend plan in the final phase of dissertation writing. I can't promise that I'll stop going to school, but I promise that I won't write another dissertation anytime soon.

# Contents

# List of Figures

# Part I

# Information Integration

# Chapter 1

# Introduction

## 1.1 Information Integration—Background

One important use of computer technology has always been the storage, processing, retrieval and presentation of data and information. While several different definitions of information exist, we will refer to *data* as discrete objects stored in a computer system, while *information* is data that has meaning to a human user. Information is typically stored as data in databases, while database management systems provide services to access databases [EN94].

Due to changing requirements, advances in database theory and computer technology, and the tendency of users to retain legacy databases, a large number of different principles for the organization of data in database systems are in existence. Examples include the historic hierarchical and network data models [EN94], the commercially most successful relational model [Cod70], and the later approaches of object-oriented (e.g., [LAC$^+$93]), object-relational [Ram97], and semistructured data models [Abi97, BDT98, BPSM97].

While all these data models have advantages and are used to different extents, there is a common inherent difficulty to all: modelling information for a database (i.e., creating a *schema* for a given set of information) is not trivial. There are typically many ways of finding an appropriate schema for a given application, and the range of possible schemas varies with the expressiveness of the underlying data model [Hul86, MIR94]. Many modelling languages (e.g., [Che76, EWH85]) and methodologies [HK87, Eic91] have been proposed to obtain a schema for a given application. However, the process remains difficult for any data model, and an "algorithm" (i.e., an automatic procedure free from human interference) for modelling has not been found [MIR93, LSS96].

Advances over the last decade in networks and general computing technology have made it possible to access data from different, possibly remote, data sources and view and/or combine them in a single location. The increasing desire to access data from different sources in a uniform way thus led to a large field of research: information integration. This term refers to the process of building database systems that access data from multiple sources that may differ in data, schema, and even data model. In addition to the difficulties in using *one* data model or schema, there are now problems of *integrating* different databases with one another, see for example [AMM97, BRU97, GRVB98]. Thus, a widely studied problem in the area of database systems is the schema integration problem, i.e., the question how related databases can conceptually be used together in an application [BLN86, SL90, PBE95, HZ96, RB01].

## 1.2 Some Issues in Information Integration

Figure 1.1 gives an overview over some of the important issues in information integration and introduces some terminology used in this dissertation. We are not concerned with physical integration but exclusively with logical integration, as defined in the figure.

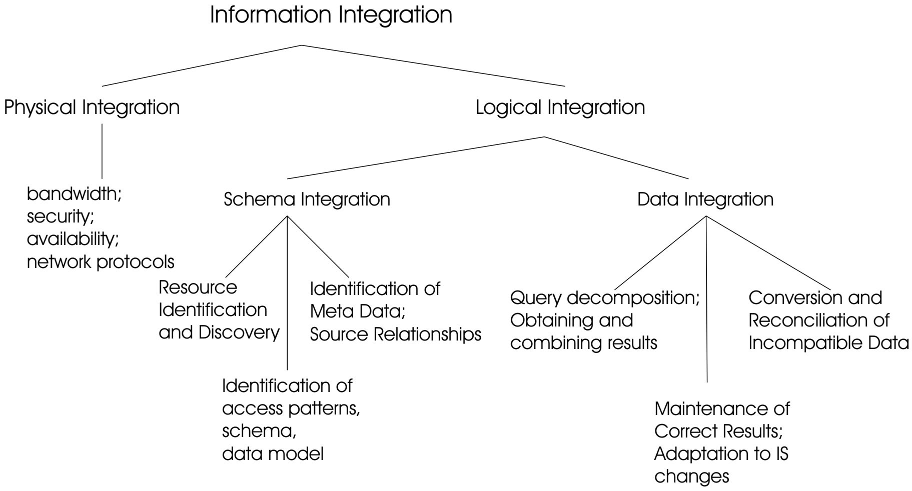

Information Integration
Physical Integration
Logical Integration
bandwidth; security; availability; network protocols
Schema Integration
Data Integration
Resource Identification and Discovery
Identification of Meta Data; Source Relationships
Identification of access patterns, schema, data model
Query decomposition; Obtaining and combining results
Conversion and Reconciliation of Incompatible Data
Maintenance of Correct Results; Adaptation to IS changes

Figure 1.1: Tasks in Information Integration

The process of logical information integration (i.e., building a database system integrating other source databases) involves two essential phases (Fig. 1.1): (1) setting up an organizational structure (model, schema, data objects) for the integrated database ("schema integration"), and (2) actually providing *data* to a user ("data integration").

In the first phase, two tasks arise:

- Obtaining (by human input or automatic discovery) as much schema information as necessary about the sources, for example through manual input [BLN86, RB01], inference from existing meta-information [PSU98, LNE89], or data-driven discovery [DP95, LC00]. This is necessary to find a good global structure for the integrated database.

- Performing the actual integration, i.e., identification of ways to translate data models [GMHI$^+$95, TRV96, RS97, GRVB98] and schemas [SL90, LSS96] of each source to fit into the global data view that is presented to the user.

For the second phase of data integration, important steps include breaking down global queries for the information sources, converting data, and maintaining consistency of the results. Different approaches to data integration have been taken, most notably mediators [Wie92, HGMN$^+$97, LYV$^+$98, TRV96, BRU97] and (materialized) views [EN94].

In the latter approach, data is provided as a *view*, i.e., a result of a query, called a *view definition*. View-based information integration is characterized by the following features:

- **security:** Views make it possible to give users access only to certain data elements (e.g., through export schemas [SL90]).

- **performance**: Views can be materialized (i.e., their view extents can be replicated at the location of the view) to minimize access to the sources [EN94]. Materialized views are closely related to *Data Warehouses* [Rou98].

- **durability**: Views can be incrementally updated when data sources change [BLT86, ZGMW96, NLR98]. This process is referred to as *view maintenance*. Even *view definitions* (as opposed to only the view extent) can be adapted to changing source, for example when sources change their schema or become unavailable [NLR98] ("view synchronization"). The *EVE*-System, described in Sec. 2, provides for view maintenance under both data and schema changes.

- **uniformity**: Depending on the view definition language, differences in schema and model of underlying sources can be made transparent to some extent. Differences that can be made transparent range from simple translation of data types to more complex issues of schema heterogeneity [SL90, LSS96].

However, there are important unsolved problems in both phases of information integration.

One problem that has recently become more and more important arises from the fact that data sources that are to be integrated have often been developed independently from one another [LSS96]. While such sources may be storing related data, they are often *schematically heterogeneous*, which is the situation in which "one database's data (values) correspond to metadata (schema elements) in others" [KLK91].

Such databases are also usually maintained by independent entities, meaning that changes in their data or even schema can occur at any time and cannot be controlled by operators or users of the integrated database system. Clearly, there is a need for integration methodologies that can (1) transform schemas [LSS96], (2) "survive" schema changes in the sources (i.e., adapt to such changes without becoming undefined) [LNR01], and (3) incrementally update views defined over such sources [AESY97, NR98, NLR98]. While some solutions have been proposed for each phase, there is so far no comprehensive solution for the entire process of maintaining a view over heterogeneous databases.

Another problem is that the *schemas* of databases are often not fully known or understood when databases are to be integrated. Reasons for the absence of schema information include, for example, a lack of cooperation by providers (e.g., when Web sites are used as information sources) or a lack of documentation of the sources (e.g., when companies merge and in the process need to integrate legacy databases with one another). One field of research in which solutions for the gathering of additional information about the structure of a database have been developed is known as *knowledge discovery in databases (KDD)* [FPSS96, FPSSU96, Fay97, PF91].

A related issue is *redundancy* across data sources. Different data sources may be partly or completely redundant, for example if independent databases contain data about the same real-world entities or if partial or complete backup data is available for some database [LNR01, Dus97, KLSS95]. Redundancies in databases, if known to a user, can be helpful in providing higher availability of integrated databases [NR98]. Knowledge about redundancies can also help with integrating databases in general, for example when different entities collecting similar data are combined, as is the case with many company-mergers. As such knowledge about the relationships between databases is very useful, it is important to find strategies to *discover* such knowledge if it is not available.

In summary, database technology in a networked world would benefit greatly from a solution to the following *information integration problem*:

> *Given a set of databases that may be semantically related but may differ in data model and schema, provide a view over those databases such that the greatest amount of information relevant for the user is extracted from the available data. To find the best possible views, it will generally be necessary to exploit the interrelationships and redundancies between data sources.*

A general automated solution to this problem does not seem within reach, however many interesting proposals have been made [EW94, KLSS95, GMHI$^{+}$95, TRV96, GKD97, RS97, AKS96, LNR01]. Previous work in which the author of this dissertation participated includes a solution to some of the problems mentioned above. The *EVE*-Project ([NLR98] and others, Sec. 2) defines a materialized view management system that, among other features, is able to maintain views under schema changes of underlying information sources. While this is an important contribution to the field of information integration from independent sources, there are shortcomings of the *EVE*-approach. Two important shortcomings are the fact that (1) only data sources that can be queried together in one SQL-query can be integrated, and (2) that in order to maintain views under schema changes, the system need meta-information about the relationship between sources that is not generally available. In this way, the *EVE*-Project provided some of the motivation for the problems tackled in this dissertation. A brief overview of the *EVE*-System and its significance in our context can be found in the next chapter.

## 1.3 Problem Definition

### 1.3.1 Discovery of Inclusion Dependencies

Usually, meta information about sources, such as the semantics of schema objects, functional dependencies, or relationships between different data sources, is not explicitly available for an integration effort. Often, only the schema is known (or can be queried) and data can be queried through some kind of interface (e.g., using a query language such as SQL). However, many other kinds of meta information about sources would be beneficial to perform meaningful information integration.

One important class of meta information is the class of *constraints* that restrict the possible states of an information source. Such constraints are useful in the determination of relationships between sources [LNE89] and thus for information integration. Some integration systems perform a semi-automatic integration using such constraints (e.g., [GKD97, KLSS95]). Such approaches, as well as manual or (hypothetical) fully automatic integration systems, would benefit greatly from the availability of as much meta-information as possible about the sources to be integrated. The manual search for such constraints is tedious and often not possible. This is true in particular when many related information sources are available or when large relations (with many attributes) are to be compared for interrelationships. Therefore, the question of whether it is possible to automatically *discover* meta-information in otherwise unknown data sources is important and has been approach by a number of authors,e.g., [SF93, LNE89, KMRS92].

Clearly, the existence of a *constraint* cannot be inferred from an inspection of the data, as any suspected constraint may only hold temporarily or accidentally. However, it is possible to gather evidence for the existence of a suspected constraint (a process usually referred to as *discovery*) by the determination of *patterns* in source data. The assumption underlying this discovery is that if a large part of a database supports a hypothesis about a constraint and there is no evidence to refute the hypothesis, then this constraint is likely to exist in the database.

While some types of constraint discovery have been studied to some extent (for example, functional dependencies [SF93] and various key constraints [LNE89]), one important class of constraints, namely *inclusion dependencies*, has received little attention in the literature thus far. Inclusion dependencies express subset-relationships between databases and are thus important indicators for redundancies between data sources. Therefore, the discovery of inclusion dependencies is important in the context of information integration.

While inference rules on such dependencies have been derived in the literature [CFP82, Mit83], the *discovery* of inclusion dependencies has not yet been treated thoroughly. A paper by Kantola *et al.* [KMRS92] provides some initial ideas and a rough complexity bound. It has also been argued in the literature [LV00] that a new database normal form based on inclusion dependencies (IDNF) would be beneficial for some applications. Furthermore, there is some work on the discovery of relationships between web sites, in which web sites with their hyperlinks are modelled as graphs [CSGM00].

### 1.3.2 Incremental Maintenance of Schema-Restructuring Views

It is known [MIR94, GLS96, GLS$^+$97] that, in any data model, there is no unique or even "best" database schema for a given application. As a consequence, there are many databases in the "real world" that can store the same data (i.e., whose databases states can be mapped by an isomorphism) but that may use *incompatible* schemas. Such databases are referred to as *semantically equivalent* [MIR93], but schematically (or syntactically [LSS96]) heterogeneous.

Incompatibility (or *schematic heterogeneity*) is generally defined with respect to some query language, i.e., two databases are incompatible (or *schematically heterogeneous*) if the query language used to query them cannot produce identical query results even if the two databases contain identical information. With respect to SQL (as well as OQL and its variants), semantically equivalent databases are often incompatible [LSS96, LSS99]. Incompatibility with respect to SQL is due to the limited query capabilities of SQL, such as the requirement that elements of the SELECT-clause be constants, that aggregation can only occur over single attributes, that attribute and relation names ("schema") are treated in a fundamentally different way from values in tuples ("data"), or that in SQL there is no support for any kind of loop through schema elements. Lakshmanan *et al.* [LSS96] have proposed a query language called *SchemaSQL* that overcomes many of those restrictions. Data and schema of relational tables are treated in a uniform way (i.e., there is no distinction between schema values such as attribute names and data values), with the effect that a larger number of databases can be restructured into one another, removing much of the incompatibility.

However, *SchemaSQL* (and other such proposals, such as MSQL [LAZ$^+$89] or XSQL [KKS92]) have only been defined as *query languages*, not *view definition languages.* So far, no incremental view maintenance schemes have been developed in the literature. While there is much work in incremental view maintenance for the standard relational data model, such as [BLT86, ZGMW96, AESY97], it is not possible to simply adapt that work to schema-restructuring views. One problem is that traditional SQL view maintenance assumes that data updates have a particular schema and that any change to the view can be expressed as a *delta-relation* (i.e., a set of tuples describing the difference between old and updated view). These assumptions do not hold for *SchemaSQL* and other schema-restructuring languages. Furthermore, view maintenance traditionally only takes data updates into account, whereas in schema-restructuring languages data updates may be transformed into schema changes, and vice versa.

## 1.4 Approach and Contributions

> *In this dissertation, we propose solutions for the two problems in information integration described in Sec. 1.3: a comprehensive, fully automatic, methodology for the discovery of inclusion dependencies in databases and a framework including an algorithm for the incremental maintenance of schema-restructuring views.*

Fig. 1.2 gives a general overview of related solutions in the field of information integration, based on the classification given in Fig. 1.1. It includes the author's previous contributions to the *EVE* project (Sec. 2) and places this dissertation's solutions in the overall information integration context.

In the following sections, we give a brief overview over the assumptions made for the two major information integration solutions provided in this dissertation and describe the scope of the solutions provided.

### 1.4.1 Discovery of Inclusion Dependencies

Inclusion dependencies are rules between two tables $R$ and $S$ of the form $R[X] \subseteq S[Y]$, with $X$ and $Y$ attribute sets. The discovery of inclusion dependencies is an NP-complete problem, as Kantola *et al.* [KMRS92] have shown. The number of potential inclusion dependencies between two tables is exponential in the number of attributes in the two relations. However, the number of *interesting* inclusion dependencies (i.e., such dependencies that are not subsumed by others or that actually express some semantic relationship between tables) is often quite small. Therefore, the problem of finding those "interesting" dependencies is difficult, but solvable as we will show.

In this dissertation, we propose an automated process for the discovery of inclusion dependencies in databases under three assumptions:

- The data model of the databases in question must include the concept of *attributes* ("columns" of data), cannot have complex objects (such as nested relations), and cannot have pointers or cross-references between data objects. The relational data model, currently the most widely used, satisfies this requirement. For other data models, such

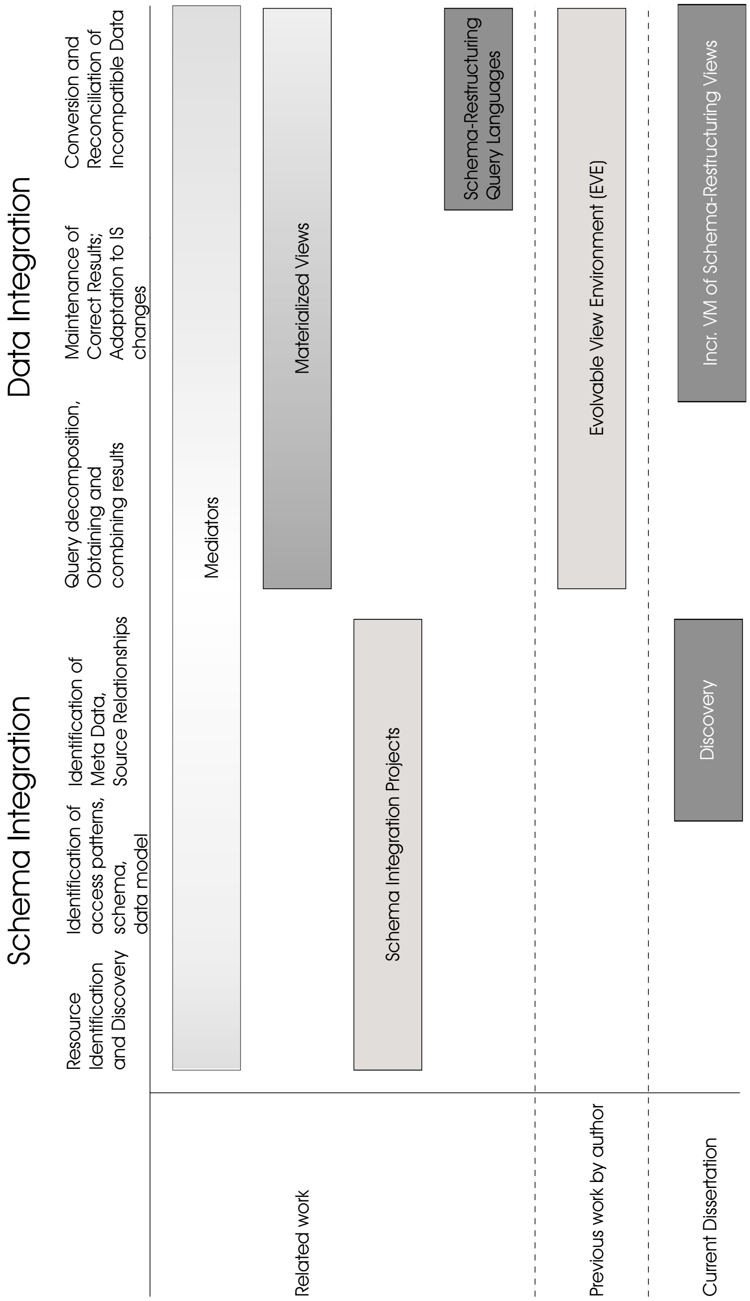

Schema Integration
Data Integration
Resource Identification and Discovery
Identification of access patterns, schema, data model
Identification of Meta Data, Source Relationships
Query decomposition, Obtaining and combining results
Maintenance of Correct Results; Adaptation to IS changes
Conversion and Reconciliation of Incompatible Data
Related work
Mediators
Materialized Views
Schema Integration Projects
Schema-Restructuring Query Languages
Previous work by author
Evolvable View Environment (EVE)
Current Dissertation
Discovery
Incr. VM of Schema-Restructuring Views

Figure 1.2: Previous and Current Solutions in Information Integration. Darker Shading in a Box Represents a More Comprehensive Solution.

as object-oriented models or semi-structured models (XML), slight restrictions must be imposed on possible data sources.

- The data sources must provide the names and perhaps the types of their schema elements (i.e., the schema must be available).
- The data sources must support the *testing* of a given inclusion dependency, for example through some query language.

No further requirements are made of the data sources.

Under these assumptions, our algorithm, named $\mathsf{FIND}_2$, will discover inclusion dependencies between two given data sources (see box labeled "Discovery" in Fig. 1.2). Since the general problem is NP-complete, a full solution cannot always be found. Therefore, the $\mathsf{FIND}_2$ algorithm will first attempt to discover *all* such dependencies, if the problem size is small enough. For problems that exceed a certain size it will apply heuristics to either discover the *largest* inclusion dependency between the data sources or at least find *some large* dependency for very large problems. The algorithm will report whether it has found the complete or only a partial solution.

Our approach uses a mapping of the inclusion dependency discovery problem to the *clique-finding* problem in graphs and $k$-uniform hypergraphs. As the clique finding problem itself is also NP-complete [GGL95], additional heuristics are used when the problem size exceeds certain limits to restrict the size of the graphs involved and find a partial or complete solution to the problem.

The contributions of this work are as follows:

1. We give a complete theory of the discovery of inclusion dependencies.

2. We present an algorithm for an exact solution of the problem for smaller problem sizes (relations with fewer than 30–40 attributes).
3. We propose heuristics that can be applied for large problem sizes (up to about 100 attributes).
4. We have implemented all algorithms and heuristics.
5. We have performed extensive experiments on our implementation of the $\mathsf{FIND}_2$ algorithm over relational databases. The experiments show the feasibility of the approach and prove that even for relations of 80 attributes and $100,000$ tuples, the discovery of inclusion dependencies is possible within reasonable time (on the order of magnitude of a few hours on a standard 400-MHz-Pentium-PC).

### 1.4.2 Incremental Maintenance of Schema-Restructuring Views

A schema restructuring view is a view defined in a schema-restructuring query language like *SchemaSQL* [LSS96]. Incremental view maintenance in such views is fundamentally different from traditional incremental view maintenance. One important reason is that it is necessary to handle schema changes in addition to data updates. Furthermore, due to the conversion between data and schema elements that is a main feature of schema-restructuring query languages, the computation of updates to a view based on source updates becomes much more complicated compared to standard relational view maintenance.

In this dissertation, we give an algebra-based solution for incremental view maintenance in schema-restructuring views, under the following assumptions:

- The data sources must be relational (this is a requirement of the class of query language used, in particular *SchemaSQL*).
- We must have access to the data sources through a query language (e.g., SQL)
- Data sources must notify the view maintenance system about their updates, i.e., we do not study the problem of change discovery itself.

We solve the issues that arise in incremental view maintenance for schema-restructuring views, using *SchemaSQL* as an example for the view definition language (see the box labeled "Incr. VM of Schema-Restructuring Views" in Fig. 1.2). We observe that, due to the possible transformation of "schema" into "data" and vice-versa, we must not only consider data updates (DUs) for *SchemaSQL*, but also schema changes (SCs). A consequence is that, as shown in this work, using the standard approach of generating query expressions that compute some kind of "delta" relation $\Delta$ between the old and the new view after an update is not sufficient. Our algorithm thus transforms an incoming (schema or data) update into a sequence of schema changes and/or data updates on the view extent.

The contributions of this part of our work are as follows:

1. We identify the new problem of schema-restructuring view maintenance.
2. We give an algebra-based solution of this view maintenance problem.

3. We prove this approach correct.

4. We develop a prototype implementation of a query engine and incremental view maintenance system for *SchemaSQL*.

5. We describe performance experiments showing the improvements of this approach over recomputation.

## 1.5 Organization of this Dissertation

This dissertation is organized into four parts. Part I includes this introduction (Chapter 1) and reviews problems and solutions in information integration, in particular the *EVE*-Project (Chapter 2) to which the author has made contributions and which has provided some of the motivation for the work in this dissertation.

Part II describes the discovery of inclusion dependencies. Chapter 3 introduces the problem and reviews background. Chapter 4 describes algorithm $\mathsf{FIND}_2$ which finds the exact solution to the problem for smaller-size problems. Chapter 5 introduces heuristics and supplements algorithm $\mathsf{FIND}_2$ by algorithm $\mathsf{CHECK}_\mathsf{H}$, adding heuristics to find partial or complete solutions for larger problems. Chapter 6 shows our experimental results and Chapter 7 reviews related work.

Part III deals with the incremental maintenance of schema-restructuring views. Chapter 9 introduces the topic and reviews background. Chapter 10 introduces our incremental view maintenance strategy for schema-restructuring views, including detailed rules for update propagation for both single and batched updates. Chapter 11 gives summaries of our performance experiments and Chapter 12 reviews related work.

Finally, Part IV concludes the dissertation. Chapter 14 summarizes our results and lists starting points for future work. Two additional algorithms that were used in the discovery work are presented in Appendices A and B, respectively.

# Chapter 2

# The Evolvable View Environment (*EVE*)

## 2.1 Maintenance of Views Under Schema Changes

In this chapter, we will give a brief overview of the *EVE* project, as it provides additional motivation for the importance of the issues approached in this dissertation. Shortcomings of the *EVE* work led us to tackle the problems solved in this dissertation.

The focus of this dissertation lies on information integration. While this has been an important field of research for a long time, newer developments, such as the World Wide Web, have increased the importance of integration. An important feature of the WWW is the inherent independence of data producers from data consumers. Independent data producers or providers have control over the capabilities (schema) of their information sources which raises the question of the influence of *schema changes* (deletions, renames, and additions of attributes or relations in underlying databases) on a view. In traditional views (as introduced in the literature, e.g., [BLT86, ZGMW96, AESY97]), schema changes can render a view definition undefined.

Two general approaches that can address this problem have been presented in the literature. One approach taken by Levy et al. [LSK95], as well as Arens et al. [AKS96] is to create a *global domain model*, i.e., an a-priori defined schema fixed in time that defines all possible attributes and relations in a given domain ("world view"). Over such a domain model, information providers define views that specify which part of the world's data they provide. Consumers also query the domain model. An algorithm then rewrites a consumer's query in terms of the providers' views currently available and thus provides the consumer with whatever data happens to be available at the moment. Changes would then only be possible in the views, while the domain model never changes, and could be accommodated by query rewriting algorithms that rewrite queries using views.

The inverse approach, explored by the author and others in a number of publications in the context of the *EVE* project [RLN97, NLR98, NR98, KRH98, KR99, LKNR99b, KR00] neither relies on a globally fixed domain nor on an ontology of permitted classes of data—both strong assumptions that are often not realistic. Instead, views are built in the traditional way over a number of base schemas and those views are adapted to base schema changes by rewriting them using information space redundancy and relaxable view queries [RLN97, RLN97, NLR98, NR98, KRH98, Nic99]. The benefit of this approach is that no pre-defined domain (which is hard to establish and to maintain) is necessary, and a view can adapt to changes in the underlying data by automatically rewriting user queries (without human intervention).

In the *EVE*-Project, the author of this dissertation and others have defined algorithms that can rewrite a view definition under schema changes (in particular *deletions*) of underlying sources and retain all or a part of the view extent in the new rewritten view. The notion of non-equivalence of view rewritings has been defined [NLR98], and a model for a numeric assessment of the quality and cost of such rewritings has been presented [LKNR99b].

However, the *EVE*-Project has several important shortcomings that provided the motivation for some of the work presented in this dissertation. We will give a brief overview over the work done in the EVE-project and its significance in general and for this dissertation in particular.

## 2.2 The *EVE*-System for Synchronization of Materialized Views

The *Evolvable View Environment* (*EVE*), to which the author of this dissertation made several contributions [KRH98, KR99, LKNR99b, KR00, KR01], is a materialized view maintenance framework that is able to maintain views over dynamic distributed data sources. Source updates include both the commonly studied data updates and the previously unexplored update class of schema changes (deleting, renaming, and adding relations and attributes). *EVE* consists of several modules (Fig. 2.1) that accomplish the tasks described below.

The *EVE*-Middleware integrates data sources through wrappers and supplies data specified through view queries to a user. Its major components are a *Multidatabase Query Engine* that collects data and handles the propagation of updates and maintenance queries, as well as a *Materialized View Evolver* that tracks schema changes in underlying sources and keeps the MKB synchronized with source schemas. To support those components, *EVE* keeps two meta-data stores. The *View Knowledge Base* contains the definitions of user views in an SQL-like language. The *Meta Knowledge Base*, similar to the University of Michigan Digital Library system [NR97a, NR97b], stores information about source schemas and source relationships. The Meta Knowledge Base (MKB) is a resource that can be exploited when searching for an appropriate substitution for the affected components of a view in the global environment. The structure of VKB and MKB data is discussed below.

Several modules fulfill subtasks in the general *EVE*-System. The *View Synchronizer* rewrites views if otherwise a view would become undefined after a source schema change. The *View Maintainer* adapts view extents incrementally after view rewritings, reducing the amount of data requested from the sources. The *QC-Computation and View Selection* module compares different possibilities for view rewritings and chooses a desirable one. The *Concurrency Control* module resolves concurrency issues between data updates and/or schema changes in the information sources.

### 2.2.1 A Model for Information Source Description

The purpose of view synchronization in *EVE* is to preserve useful view information in terms of the view interface as well as the view extent, to

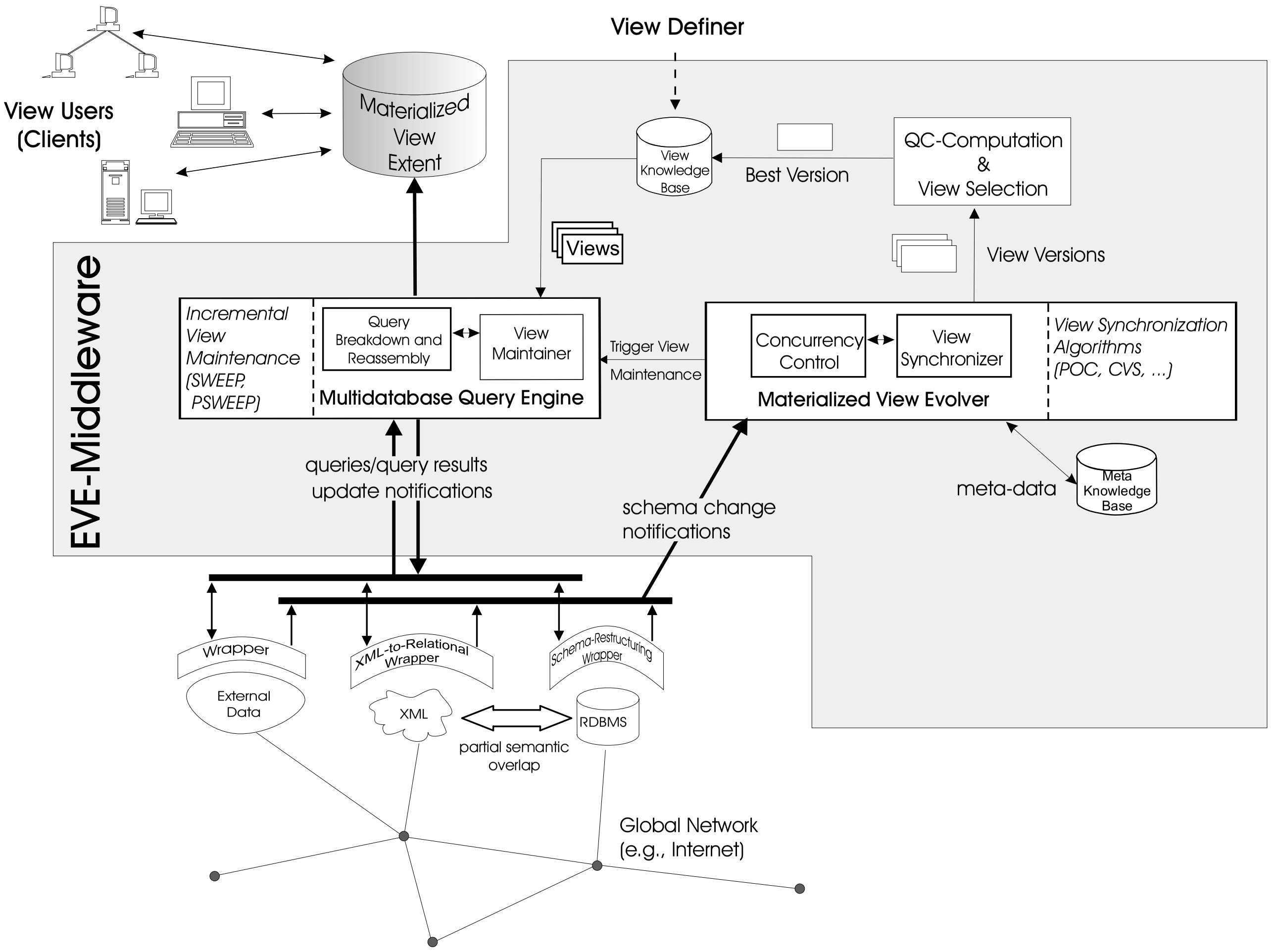


Figure 2.1: The Framework of the Evolvable View Environment (*EVE*).

the largest degree possible. This requires us to be able to find alternative, ideally semantically equivalent, replacements for components of a view definition that may be no longer available from one of the information sources (ISs). To accomplish this task, *EVE* contains a model for the description of both the capabilities of each IS as well as interrelationships between ISs. The availability of such type of knowledge is a critical resource that can be exploited when searching for appropriate substitutions for the affected components of a view in the global environment. While several different types of meta-information are used in EVE, the most important form of useful information is knowledge about containment between relations or their projections. However, such information is not always explicitly provided by information providers, in particular when the sources to be compared belong to independent or competing providers.

Here, an automated way of "mining" some containment information about information sources would be very helpful in filling the *EVE* MKB with useful data. In fact, this requirement of the *EVE* system provided some of the motivation for the work discussed in Part II of this dissertation. Our solution of the inclusion dependency discovery problem presented in this dissertation can be used to discover containment information in databases.

We will briefly review *containment constraints*, which serve as the tool for modeling containment in *EVE*, to point out their similarity to the inclusion dependencies discussed in Part II of this dissertation.

A containment constraint between two relations $R_1$ and $R_2$ states that a (horizontal and/or vertical) fragment of $R_1$ is semantically contained or equivalent to a (horizontal and/or vertical) fragment of $R_2$.

Consider the containment constraint shown in Figure 2.3 which is defined over the example information space in Fig. 2.2. This containment constraint shows that the projection on the Holder and Age attributes of relation Insurance forms a superset of the projection on attributes Name, Age of relation BackBay for all tuples in Insurance whose Amount is over $1,000,000$ and whose Age is under 50.

IS 1: **Flight Information**
Customer(Name,Address,PhoneNo,Age)
FlightRes(PName,Airline,FlightNo,Source,
Dest,Date)

IS 2: **Insurance Information**
Insurance(Holder,Type,Amount,Age)
PreferredCust(PrefName,PrefAddress,
PrefPhone)

IS 3: **Tour Participant Information**
Participant(Name,TourID,StartDate,Location)
Tour(TourID,TourName,Type,NoDays)

Figure 2.2: Example Information Source Content Descriptions

$$\mathcal{CC}_{\text{Customer,Insurance}} = \\ (\pi_{\text{Insurance.Holder, Insurance.Age}}(\sigma_{(Insurance.Amount > 1{,}000{,}000)}\text{Insurance})) \subseteq \\ \pi_{\text{Customer.Name,Customer.Age}}\text{Customer}$$

Figure 2.3: A Containment Constraint in the Example Information Space

It is clear that containment constraints are closely related to inclusion dependencies (INDs) which are the focus of Part II of this dissertation. In fact, INDs are simply a special case of containment constraints, with a set relationship of "⊆" or "⊇" and no selection conditions. Therefore, the discovery of INDs between relations would provide very valuable information that could be used to form containment constraints for the *EVE*-System.

### 2.2.2 A Preference Model for View Evolution

View definers themselves need to be able to control the view evolution process, as they are knowledgeable about the extent to which the different components of a view are critical or dispensable. For example, a view definer may know that one attribute (say the attribute Name) is indispensable to the view, whereas another attribute (say the attribute Address) is desirable yet can be omitted from the original view definition, if keeping it becomes impossible, without jeopardizing the utility of the view.

While on the one hand we do desire user input to inform our system of preferences about the most desirable view synchronization, we must face the problem that the original view definer may no longer be around when a view becomes affected by changes of its underlying sources. In addition, it may not be practical to disable the view and to stop all applications dependent on it until the original view definer (or other knowledgeable users of the view) are available to help with this process. For this purpose we developed a view definition language, called Evolvable-SQL (E-SQL)[RLN97], that incorporates *evolution parameters* into the SQL view definition language. E-SQL allows the view definer to specify criteria based on which the view will be evolved by the system. A typical E-SQL view is given in Figure 2.4.

The *view-extent* parameter $\mathcal{VE} \in \{\subseteq, \supseteq, \equiv, \approx\}$ expresses the relationship between the original and rewritten query as required by the view definer. For instance, $\mathcal{VE} =$"$\supseteq$" requires any query rewriting $V_i$ for the current view $V$ to compute a superset of the original view extent (i.e., $V_i \supseteq V$). The value

```
CREATE VIEW Asia-Customer ($\mathcal{VE}$ = “$\subseteq$”) AS
SELECT      Name ($\mathcal{AR} = true$), Address ($\mathcal{AD} = true$)
FROM        IS1.Customer C ($\mathcal{RR} = true$), IS1.FlightRes F
WHERE       C.Name = F.PName ($\mathcal{CR} = true$) AND (F.Dest = ‘Asia’)
```

Figure 2.4: A Typical E-SQL View

$\mathcal{VE}$ =“$\approx$” means no restrictions for the extent are given. Also, for each element in the view definition's SELECT-, FROM- and WHERE-clause, respectively, two boolean values determine whether that view element is (1) *dispensable* from the view definition and/or (2) *replaceable* with a meaningful alternative from the information space. Those parameters are named *attribute-dispensable (AD), attribute-replaceable (AR), relation-dispensable (RD), relation-replaceable (RR), condition-dispensable (CD), and condition-replaceable (CR).* For a full description of the E-SQL language, the reader is referred to [LNR97].

The semantics of the query in Fig. 2.4 are as follows: Any rewriting of the view query is acceptable as long as the new view extent is a subset of the old one (expressed by $\mathcal{VE}$ = “$\subseteq$”); the attribute Address is dispensable (expressed by $\mathcal{AD} = true$) and attribute Name can be replaced from another source ($\mathcal{AR} = true$); the relation Customer (but not FlightRes) can be replaced with another relation ($\mathcal{RR} = true$) and the user will still have use for the view even if the first WHERE-condition has to be replaced with a similar one ($\mathcal{CR} = true$).

### 2.2.3 View Synchronization Strategies.

One of the primary objectives of *EVE* is to design alternate strategies (algorithms) for evolving views transparently according to users' preferences as well as available information in the environment. Based on this solution framework of E-SQL and the information source descriptions, we introduced strategies for the transparent evolution of views. Our proposed view rewriting process, which we call *view synchronization*, finds a new view definition that meets all view preservation constraints specified in the original view definition, i.e., the preferences noted in the E-SQL definition. Furthermore, it identifies and extracts appropriate information from other ISs as replacement of the affected components of the view definition and produces an alternative view definition. Important algorithms for view synchronization are POC/SPOC [NR98], CVS [NLR98], as well as GRASP [KRH98] and *History-Driven View Synchronization (HD-VS)* [KR00]. The later two are by the author of this dissertation.

#### The POC Algorithm

Due to its simplicity, the most widely used algorithm in the *EVE* project is the *Project-Containment (POC) Algorithm.* POC uses information from *containment constraints* to rewrite views after source schema changes. Depending on the view evolution parameter specified by a user (see Fig. 2.4) and the available containment information, POC replaces deleted attributes or relations by alternatives found through containment constraints. The constraint given in the view evolution parameter is kept valid by choosing only replacements that satisfying that parameter. In that process, it might be necessary to introduce additional constraints in the WHERE-clause of the view query.

**Example 2.1** *We define an information space (Meta Knowledge Base) according to Figures 2.2 and 2.3. We consider the view* Customer-Passengers *in Figure 2.4 and show how to apply the POC algorithm and find a replacement under the schema change* `delete_relation(Customer)`*.*

*The POC algorithm uses containment constraints in the MKB that connect the new relation to the remaining relations in the existing query. Here, we can replace the attribute* Customer.Age *by the similar attribute* Insurance.Age *in relation* Insurance *and join the new table with* FlightRes *using the containment constraint from Figure 2.3. Then all view elements (i.e., attributes and WHERE-clauses) that depend on the old relation are replaced by view elements using the new relation. A possible rewriting of the query in Figure 2.4 using this substitution is given by the query in Figure 2.5.*

```
CREATE VIEW Asia-Customer' (𝒱ℰ = "⊆") AS
SELECT      I.Holder (𝒜ℛ = true)
FROM        IS2.Insurance I (ℛℛ = true), IS1.FlightRes F
WHERE       I.Holder = F.PName (𝒞ℛ = true)
            AND (F.Dest = 'Asia')
            AND (I.Amount>1,000,000)
```

Figure 2.5: A Possible Rewriting for a View.

*This view rewriting will have no* Address*-attribute, as no address information is available after the deletion of relation* Customer*. However, names of customers are still available, as long as those customers had a large enough* *insurance policy, and will now come from information source IS2, from relation Insurance.*

### History-Driven View Synchronization

All earlier view synchronization algorithms (i.e., SVS, POC/SPOC, CVS, GRASP) are *single-step* algorithms and perform synchronization only after *delete*-schema changes. They react to a single schema change in the underlying relations with a single view synchronization step. The synchronized view definition is then used as the basis for any further synchronization steps.

In particular, after the deletion of an underlying relation that has been used by a view, the view is rewritten to not refer to that relation any more. Even if the same relation is later added back to the information space (for instance, after a temporary unavailability due to a network problem), it will never be used by the view again since without a global domain model the view synchronization algorithm cannot determine in what relationship a new data element stands to other previously available elements.

As a solution to this problem, the author of this dissertation proposed *history-driven view synchronization (HD-VS)* [KR00, KR01]. This is a process capable of handling a more comprehensive set of information source schema changes, namely *adds*, *renames*, and *deletes* of attributes and relations. Also, it is capable of rewriting views as necessary under changes of *constraints* across the source databases (such as a *containment relationship* defining that IS1.Hotels contains IS2.BostonHotels). The main contribution of the HD-VS work is the use of additional available meta data to keep views as close to their original definition as possible, under a sequence of meta data changes that occurs over time. We will give a brief overview of the HD-VS algorithm for History-Driven View Synchronization.

- As in one-step synchronization (e.g., [NR98]), a synchronization occurs after each meta data change for an affected view. However, now input data are not only the current but also all previous states of MKB and VKB as well as the meta data changes that occurred, i.e., the complete history.

- If a view can be rewritten, the algorithm rewrites a valid view on the old information space into a valid view on the new information space. In certain cases, the algorithm falls back on the one-step view synchronization algorithms (e.g., POC).

- Now, not only *deletes* and *renames* but also all *adds* of meta data are considered. Meta data includes both *schema* such as relations and attributes, as well as other *constraints* such as containment constraints.

- If a view is rewritten, its quality (usefulness to a user, Sec. 2.2.4) may increase over the previous version (whereas the one-step algorithms could never improve on quality), depending on the meta data change that caused the synchronization.

- If a meta data item is deleted, the algorithm tries to compensate the deletion with a previous add and vice-versa (“cancellation”). That means that *temporary unavailability* of data can be accounted for. The algorithm is capable of *returning* to a previous view definition if appropriate.

HD-VS uses three main concepts: backtracking in the history of a view, re-applying a part of a meta data update sequence from that history, and reconstructing part of the view's history graph in the process of re-application of meta data changes.

### 2.2.4 Cost Model for Evolved View Definitions

Unlike in traditional query processing research, we may need to construct alternate view definitions as solutions that no longer are *equivalent* to the original view, i.e., the view interface (set of attributes) may be reduced or the number of tuples returned may not correspond to the original extent. *EVE* thus contains a *formal model of correctness* for view synchronization that characterizes what are "correct" view rewritings for a given view definition, as well as what are measures that allow us to evaluate the *quality* of alternate solutions. It is important to note here that existing cost measures of query rewritings are only partially applicable to our problem domain, as we must take *relative quality* of the amount of view preservation into account. It is not sufficient to assess only the *cost* of synchronizing or maintaining the view.

For this purpose the author of dissertation and others have developed the QC-Model [LKNR99b] to estimate the *quality* and *cost* of the rewritings. Each legal query rewriting will in general preserve a different amount (extent) and different types (interface) of information, which we refer to as the *quality* of the view. Also, each new view query will cause different *view maintenance costs*, since in general data will have to be collected from a different set of ISs. With these two dimensions, the QC-Model can *compare* different view queries with each other, even if they are not equivalent. This comparison is accomplished by assessing five different factors as outlined below (full algorithm in [LKNR99b]).

- **Quality Factors:** Quality refers to the *similarity* (vs. divergence) between an original view $V$ and its $n$-th rewriting $V^{(n)}$ and is expressed for an original view $V$ by $Q(V^{(n)})$.
  - The *Degree of Divergence in Terms of the View Interface* determines how different the view interfaces of the two queries are. This can be expressed numerically by counting the common and non-common attributes in both queries and computing a percentage. For the purpose of this current work, we will abstract from further details and instead refer to [LKNR98].
  - The *Degree of Divergence in Terms of the View Extent* is determined by the relative numbers of missing and additional tuples in the extent of a view rewriting (as compared to the extent of the original view). In order to estimate the overlap between old and new views, containment constraints are used, since they make statements about relationships between relations. Again, the actual formulas can be found in [LKNR98].
- **Cost Factors:** Cost factors measure the (long-term) cost associated with future *incremental view maintenance* after the view has been rewritten and the extent has been updated. The factors are *Number of Messages* between data warehouse and information sources, *Number of Bytes Transferred* through the network, and *Number of I/Os* at

the ISs. Cost factors are combined into a single value using tradeoff factors, and the combined cost value of a view rewriting is denoted by $C(V^{(n)})$.

Normalizing and then combining these factors yields the *QC-Value* for a given rewriting $QC(V^{(n)})$, a real number between 0 (bad) and 1 (good) that can be used to assess the value of a given query rewriting for a particular user (in terms of that user's E-SQL query evolution preferences).

This Quality-and-Cost Model (QC-Model) [LKNR99b, LKNR99a] has been developed in part by the author of this dissertation.

### 2.2.5 Maintenance of Materialized Views after Synchronization

To assure minimal impact of view evolution on users of a view, we have explored incremental techniques for updating the view extents after the definition of a view has been synchronized. Our goal is to achieve the level of efficiency needed to make the overhead for view synchronization minimal from a user's point of view. Previous work on view maintenance assumed that only data updates were done on the underlying ISs instead of schema updates [NR99]. This problem provides another motivation for this dissertation. In Part III, we propose the first incremental view maintenance scheme that can maintain views under both data updates and schema changes. We demonstrate the principle of view synchronization under schema changes at the example of *SchemaSQL*, but our concept can be used in the context of the *EVE*-System as well.

### 2.2.6 View Maintenance Under Concurrent Schema and Data Updates

In general, ISs are not aware of and do not cooperate with the integrating data warehouse in handling schema changes and data updates. Due to the independence of the ISs, concurrent schema changes and data updates may occur at any time in any order. The *SDCC-System* [ZR99] and the *Dynamic Data Warehouse (DyDa)* system [CZC$^+$01] are extensions of the *EVE*-System and address those issues.

### 2.2.7 *EVE*-Implementation

To verify the *feasibility* of our proposed approach, we have implemented a prototype of the *EVE* system and have demonstrated it at major conferences [KLZ$^+$97, RKL$^+$98, RKZ$^+$99, CZC$^+$01]. The *EVE* system as depicted in Figure 2.1 is implemented using Java. ISs in *EVE* are either stored in an Oracle or an MS Access database, and communication between *EVE* modules and the ISs occurs through JDBC. The *EVE* graphical user interface, written in Java, communicates with the underlying system via RMI.

# Part II

# Discovery of Inclusion Dependencies

# Chapter 3

# Introduction and Background

## 3.1 Introduction

In this work, we are concerned with the discovery of meta-information (i.e., information about the syntax and semantics of data) in databases. In this work, we use the relational data model, in which data (atomic simple values such as numbers or strings) is stored in attributes (columns) that are grouped in relations (tables), but other data models also support our concepts. Rows in relational tables are referred to as tuples. In the relational model, a number of constraints have been defined [EN94] that impose several useful restrictions on a table. Examples are keys (uniqueness constraints on attributes), functional dependencies (dependencies of the values in one attribute on the values in the same tuple in other attributes), and inclusion dependencies, explained below.

### 3.1.1 Significance of Inclusion Relationships

Fagin [Fag81] suggests that inclusion dependencies (INDs) express an important relationship between relations. For example, he shows that relations whose only intra-relational dependencies are functional dependencies can be restructured into relations that have only INDs as inter-relational dependencies. More specifically, a relation that has as its only constraints a set of functional dependencies can be equivalently composed into a set of relations that have only key constraints and inter-relational inclusion dependencies. Fagin also presents a normal form for relational databases (DK/NF for *domain-key normal form*) that attempts to use *only* INDs as inter-relational dependencies.

In the context of data integration, INDs can help to solve a very common and difficult problem: discovering redundancies across data sources. Due to the nature of data and its generation, information is often stored in multiple places, with large amounts of redundancy. When trying to integrate data sources that are likely to be (even partly) redundant, a method to *discover* such redundancies would be very beneficial. One example in which redundancy discovery would be helpful is the Evolvable View Environment (EVE) [NLR98, KRH98] which is concerned with maintaining views under schema changes by replacing deleted information sources with partly redundant alternative sources. In general, the discovery of INDs will be beneficial in any effort to integrate unknown databases. A reliable algorithm to discover INDs will enable an integration system to incorporate new data sources that would previously would not have been used since their relationships with existing data was not known. Examples of systems in which our technology could be used to improve efficiency include data warehouses [Gar98], multidatabase systems such as Infomaster [GKD97], schema integration systems such as ARTEMIS [BBC$^+$00] or SemInt [LC95, LC00], or other schema matching approaches like Clio [MHH00]. Many other uses are conceivable in the field of data and schema integration, in which our tool can be useful to a human integrator as a decision support tool.

In summary, whenever sufficient meta-information about data is not available (i.e., whenever the constraints that exist in a database are not known to a user), an algorithm that discovers *candidates* for INDs (since dependencies as such cannot be "discovered" from a single state of the database) would be very helpful in extracting such meta information. A tool that solves this problem does not exist to date. Also, a manual extraction of inclusion dependencies by domain experts does not seem feasible due to the large number of information sources in the world, the potential high number of attributes in real-world relations (50–100 attributes are common), and a widespread lack of reliable meta-information about legacy databases. This work deals with the efficient extraction of candidate sets for INDs from a set of relations. We will show in the following that the discovery of such redundancies is possible and feasible by way of establishing and testing inclusion dependency candidates between given data sources.

As shown in the literature [KMRS92], the problem of finding even one maximal inclusion dependency for two given relations is inherently NP-hard. In the worst case, there is an exponential number of such dependencies for a given set of attributes in two relations. We will show that despite this inherent complexity, the discovery of inclusion dependencies is tractable for real-world databases. The limits of applicability for our algorithms are relations of approximately 100–150 attributes, while the extent size (number of tuples) of each relation is only a linear factor for performance. The size of tractable problems depends on a number of properties, such as the number of distinct values in each attribute, and the solution found by our algorithms is not always complete for large problems. However, the algorithm will always will first attempt to find *all* INDs, then fall back to finding *the* largest inclusion dependency, and if that is not possible either, will find large inclusion dependencies that are not necessarily maximal. The algorithm can be adapted to the available amount of computing time and will find better solutions given more time. It also provides a measure of quality of the solution.

## 3.2 Background

As explained above, we are concerned with discovering meta-information about interrelationships between separate databases. We will focus on relational databases, but our methodology is applicable to all data models in which inclusion dependencies can be defined (i.e., in which simple values are organized in attributes and some higher level such as relations).

The problem of discovering information and meta information from large amounts of data is widely studied, in particular in the fields of KDD (Knowledge Discovery in Databases) [FPSS96] and AI (Artificial Intelligence) [RN95]. Most efforts in information discovery are concerned with the derivation of patterns from the data available. Often, algorithms look for patterns which suggest some kind of constraint between database objects. Naturally, constraints cannot be "discovered". However one can detect data patterns that would be allowed (or not allowed) under an *assumed* constraint and thus accept or reject certain hypotheses about data patterns. Sometimes, this process is called "constraint discovery" [LH97]. The term "dependency discovery" seems to be more correct and is more widely used [KMRS92, BB95a].

We will first discuss inclusion dependencies and their theory and then review additional dependencies whose theory is related to our problem in Section 3.2.3.

### 3.2.1 Notation

For clarity, we review the notation used in this work. The notation is similar to [CFP82], from which the following section has been adapted.

Throughout this work, we will denote set variables by capital letters and variables that denote elements of a set by small letters. By "$k$-subset of $X$" we mean a subset of $X$ with cardinality $k$, while a "$k$-set" is simply a set with cardinality $k$.

A *value* is an atomic element of data that is stored in a relation's extent. Examples include "Stanley Kubrick", 1984, or 04/19/1972. A *domain* $D$ is a finite set of *values*.

An *attribute* is a bag (multiset) of values. A *relation schema* is a pair $(Rel, U)$ where $Rel$ is the relation name and $U = (a_1, \ldots, a_m)$ is a finite ordered $m$-tuple of labels, which are called *attribute names*.

A *relation* is a 3-tuple $R = (Rel, U, E)$ with $Rel$ and $U$ as above and

$E \subseteq D_1 \times D_2 \times \ldots \times D_n$ the relation extent. The sets $D_1, \ldots, D_n$ are called the *domains* of $R$'s attributes. A *tuple* in relation $R = (\mathit{Rel}, U, E)$ is an element of $E$. An operator $t[a_1, a_2, \ldots, a_k]$ returns the projection of $t$ on the attributes named $a_1, a_2, \ldots, a_k$. To be more specific about our definition of attribute, we can define an *attribute* $A_i$ as a bag constructed as follows: $A_i = \{t[a_i] | t \in E\}$ or, in short, $A_i = E[a_i]$.

We write $\mathit{Rel}[U]$ or $\mathit{Rel}[a_1, \ldots, a_m]$ when referring to the projection of a relation on a set of attributes. Note that the construct *Films[Title,Director]*, according to this definition means "the relation whose name is 'Films', with two attributes whose names are 'Title' and 'Director'". In this case, "Films", "Title", and "Director" are constants (values), not variables.

For the remainder of this work, and if not stated otherwise, we will use the generic constants $R$ and $S$ for relation names and $X$ and $Y$ as symbols for sets of attribute names.

### 3.2.2 Inclusion Dependencies

There are interesting data patterns to discover when given a *set* of relations rather than a single relation. The most common and useful pattern that can be derived across two relations are inclusion dependencies, introduced by Fagin [Fag81]. Inclusion dependencies are formally defined below.

**Definition 3.1 (IND)** *Let $R[a_1, a_2, \ldots, a_n]$ and $S[b_1, b_2, \ldots, b_m]$ be (projections on) two relations. Let $X$ be a sequence of $k$ distinct attribute names from $R$ and $Y$ a sequence of $k$ distinct attribute names from $S$, with $1 \leq k \leq \min(n, m)$. Then, an* ***inclusion dependency (IND)*** *is an asser-*

*tion of the form* $R[X] \subseteq S[Y]$.

Note that so-called "referential integrity constraints", asserting an implication between values across two attributes in two different relations, are simply a special case of an IND where both sides are projections on a single attribute each. Also, one can define *equivalence dependencies* [Fag81] in the following way: $R[X] \equiv S[Y] \iff (R[X] \subseteq S[Y]) \wedge (S[Y] \subseteq R[X])$.

**Example 3.1** *We are introducing a running example. Consider the relations defined in Fig. 3.1. An IND would be, for example,* $MyMovies[Title, Style] \subseteq Movies[Genre, Title]$*. Note that "IND" does not imply "valid in the database". Rather, validity is a feature of INDs that will be defined shortly. Some other INDs are listed in the figure.*

**Definition 3.2 (valid)** *An IND* $\sigma = (R[a_{i_1}, \ldots, a_{i_k}] \subseteq S[b_{i_1}, \ldots, b_{i_k}])$ *is* ***valid*** *between two relations* $R = (r, (a_1, \ldots, a_n), E_R)$ *and* $S = (s, (b_1, \ldots, b_m), E_S)$ *if the sets of tuples in* $E_R$ *and* $E_S$ *satisfy the assertion given by* $\sigma$*. Otherwise, the IND is called* ***invalid*** *for* $R$ *and* $S$*.*

**Example 3.2** *All INDs listed in Fig. 3.1 are valid INDs.* $MyMovies[Title, Style] \subseteq Movies[Genre, Title]$*, as mentioned in Example 3.1, is an invalid IND. Note that in order for this IND to be valid,* MyMovies.Title *would have to be a subset of* Movies.Genre *and* MyMovies.Style *would have to be a subset of* Movies.Title

An inclusion dependency is merely a statement about two relations which may be true or false. A **valid** IND describes the fact that a projection of

Movies

| Title | Genre | Director | Year |
|---|---|---|---|
| Dune | Sci-Fi | David Lynch | 1984 |
| Titanic | Drama | James Cameron | 1997 |
| Titanic | Drama | Jean Negulesco | 1953 |
| Dr. Strangelove | Satire | Stanley Kubrick | 1963 |
| A.I. | Sci-Fi | Steven Spielberg | 2001 |
| Shrek | Animation | Andrew Adamson | 2001 |
| 2001–A Space Odyssey | Sci-Fi | Stanley Kubrick | 1968 |

MyMovies

| Title | Style |
|---|---|
| Dune | Sci-Fi |
| Titanic | Drama |
| Dr. Strangelove | Satire |
| A.I. | Sci-Fi |
| Shrek | Animation |

Movies2001

| Title | Director |
|---|---|
| A.I. | Steven Spielberg |
| Shrek | Andrew Adamson |

**Some Functional Dependencies:**
Movies[Title,Year⟶Director]
Movies[Title,Director⟶Year]
Movies[Title⟶Genre]
**Valid Inclusion Dependencies (INDs):**
MyMovies[Title,Style] $\subseteq$ Movies[Title,Genre]
Movies2001[Title,Director] $\subseteq$ Movies[Title,Director]
Movies2001[Title] $\subseteq$ MyMovies[Title]
The data supports a possible
**Referential Integrity Constraint**
between MyMovies.Title and Movies.Title

Figure 3.1: Functional and Inclusion Dependencies in a Database

one relation $R$ forms a subset of another projection (of the same number of attributes) of a relation $S$. Note that INDs are defined over *sequences* of attributes, not sets, since the order of attributes is important (INDs are not invariant under permutation of the attributes of only one side).

We will also define an important feature of INDs that we will call *arity*.

**Definition 3.3 (arity of an IND)** *Let $X$ and $Y$ be sequences of $k$ attributes, respectively and $\sigma = R[X] \subseteq S[Y]$ be an IND. Then $k$ is the **arity** of $\sigma$, denoted by $|\sigma|$, and $\sigma$ is called a **k-ary IND**.*

**Example 3.3** *Genres[Title,Genre] $\subseteq$ Movies[Title,Genre] is a 2-ary or* binary *IND. Likewise, Genres[Title] $\subseteq$ Movies[Title] is a* unary *IND.*

Casanova *et al.* [CFP82] have provided some important insights into the IND problem. They have described a complete set of inference rules for INDs, in the sense that repeated application of their rules will generate all valid INDs that can be derived from a given set of valid INDs (i.e., those rules form an *axiomatization* for INDs). The rules are given below.

**Axiom 3.1 (reflexivity)** *$R[X] \subseteq R[X]$, if $X$ is a sequence of distinct attributes from $R$.*

**Axiom 3.2 (projection and permutation)** *If $R[A_1, \ldots, A_m] \subseteq S[B_1, \ldots, B_m]$ is valid (by Def. 3.2), then $R[A_{i_1}, \ldots, A_{i_k}] \subseteq S[B_{i_1}, \ldots, B_{i_k}]$ is valid for any sequence $(i_1, \ldots, i_k)$ of distinct integers from $\{1, \ldots, m\}$.*

Note that permutation refers to “synchronous” reordering of attributes on both sides, i.e., $R[X,Y] \subseteq S[X,Y] \implies R[Y,X] \subseteq S[Y,X]$, but $\neg(R[X,Y] \subseteq S[X,Y] \implies R[Y,X] \subseteq S[X,Y])$

**Axiom 3.3 (transitivity)** *If* $R[X] \subseteq S[Y]$ *and* $S[Y] \subseteq T[Z]$ *are both valid (by Def. 3.2), then* $R[X] \subseteq T[Z]$ *is valid.*

**Definition 3.4 (derived INDs)** *A valid IND* $\sigma$ *can be* ***derived*** *from a set* $\Sigma$ *of valid INDs, denoted by* $\Sigma \models \sigma$*, if* $\sigma$ *can be obtained by repeatedly applying the above axioms on some set of INDs taken from* $\Sigma$.

**Example 3.4** *Some valid INDs that can be derived from the valid INDs stated in Fig. 3.1 are* $Movies[Title] \subseteq Genres[Title]$, $Movies[Genre] \subseteq Genres[Genre]$, *and* $Movies2001[Director] \subseteq Movies[Director]$.

Casanova *et al.* [CFP82] consider the following decision problem:

> "Decide whether $\Sigma \models \sigma$, i.e., decide whether a particular IND $\sigma$ can be derived from a given set $\Sigma$ of INDs".

They show that this decision problem is decidable for finite databases but PSPACE-complete. The reason for this complexity is the exponential number of potential inclusion dependencies that can be derived from a set of INDs.

Since INDs are invariant under synchronous permutation of both sides (by Axiom 3.2), we will now define equality of INDs (which applies to both valid and invalid INDs).

**Definition 3.5 (equality of INDs)** *Two INDs* $R[a_1, \ldots, a_m] \subseteq S[b_1, \ldots, b_m]$ *and* $R[c_{i_1}, \ldots, c_{i_m}] \subseteq S[d_{i_1}, \ldots, d_{i_m}]$ *are* ***equal*** *if and only if there is a sequence* $(i_1, \ldots, i_m)$ *of distinct integers* $1, \ldots, m$ *such that* $a_1 = c_{i_1} \wedge a_2 = c_{i_2} \wedge \ldots \wedge a_m = c_{i_m} \wedge b_1 = d_{i_1} \wedge b_2 = d_{i_2} \wedge \ldots \wedge b_m = d_{i_m}$.

Note that equality according to this definition is an equivalence relation on INDs. It is also clear that equivalence preserves validity, i.e., in a set of equal INDs, the elements are either all valid or all invalid.

**Example 3.5** *In Fig. 3.1, a valid IND Genres[Title, Genre] ⊆ Movies[Title, Genre] is listed. This IND would be equal to Genres[Genre, Title] ⊆ Movies[Genre, Title] but not to an IND Genres[Title, Genre] ⊆ Movies[Genre, Title].*

One very important observation on INDs is that a $k$-ary IND with $k > 1$ naturally implies a set of unary INDs. Let $\sigma = R[X] \subseteq S[Y]$ be a $k$-ary IND. Let $\Sigma_1$ be the set of all unary INDs $R[x] \subseteq S[y]$ with $x \in X$ and $y \in Y$. Then, there clearly is a close relationship between $\sigma$ and $\Sigma_1$, as formalized in Corollary 3.1.

**Corollary 3.1** *Let $\Sigma_k$ be the set of all possible $k$-ary INDs between two given relations $R$ and $S$. Let $\Sigma_1^k$ be the set whose elements are all $k$-sets of unary INDs between $R$ and $S$. Then, there is an isomorphism between $\Sigma_k$ and $\Sigma_1^k$. We say that $\Sigma_1^k$ is **implied by** $\Sigma_k$.*

This isomorphic mapping is possible since INDs are invariant under permutations of their attribute pairs (such that there are exactly as many $k$-ary INDs as there are $k$-subsets of unary INDs), and each pair of single attributes in a $k$-ary IND $\sigma$ corresponds to one unary IND implied by $\sigma$. Note that the isomorphism does not hold for *valid* INDs since clearly the existence of $k$ unary valid IND does not imply the existence of any higher-arity valid INDs (i.e., only the direction $\Sigma_k \implies \Sigma_1^k$ holds for valid INDs, not the converse, see Sec. 4.2.2).

Validity of INDs is preserved under projections and permutation, by Axiom 3.2. In order to describe all inclusion information between two relations it is therefore not necessary to list *all* INDs between two relations. Rather, a small set of INDs from which all others can be generated will suffice, as formalized with the following definition.

**Definition 3.6 (generating set of INDs)** *Consider a set of valid inclusion dependencies:* $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$*. A* ***generating set*** *of* $\Sigma$*, denoted by* $\mathcal{G}(\Sigma)$*, is a set of valid INDs with the following properties*[1]*:*

*1.* $\forall \sigma \in \Sigma : \mathcal{G}(\Sigma) \models \sigma$

*2.* $\forall \sigma \in \mathcal{G}(\Sigma) : \neg((\mathcal{G}(\Sigma) \setminus \sigma) \models \sigma)$

In words, the generating set $\mathcal{G}(\Sigma)$ contains exactly those valid INDs from which *all* valid INDs in $\Sigma$ can be derived. The set is not empty for any $\Sigma$, since it can be constructed by first including all $\sigma \in \Sigma$ into $\mathcal{G}(\Sigma)$ and then removing all $\sigma$ for which property 2 does not hold. The set is minimal since removing any IND $\sigma$ from a $\mathcal{G}(\Sigma)$ for which property 2 holds would by definition violate property 1. Therefore, generating sets contain all information about inclusion dependencies between relations in a minimal number of INDs.

If all INDs in $\Sigma$ are defined between exactly two different relations, i.e., $\forall \sigma \in \Sigma : \sigma = (R[a_1, a_2, \ldots, a_k] \subseteq S[b_1, b_2, \ldots, b_k])$, the transitivity rule (Axiom 3.3) does not apply. Then, the set $\mathcal{G}(\Sigma)$ is also unique for a given $\Sigma$. If there were two distinct generating sets $\mathcal{G}_1(\Sigma)$ and $\mathcal{G}_2(\Sigma)$ for a $\Sigma$, at least

[1] The symbol \ stands for "set-difference".

one IND in $\mathcal{G}_2(\Sigma)$ that does not exist in $\mathcal{G}_1(\Sigma)$ would have to be derivable from $\mathcal{G}_1(\Sigma)$, which contradicts property 2. If the transitivity rule is used (Axiom 3.3) the generating set as defined above may not be unique.

### 3.2.3 Related Work on Other Dependencies

We have now defined inclusion dependencies, whose discovery is the focus of this chapter. In this section, we are looking at related work in knowledge discovery in databases that may be useful as starting points for a solution to the IND discovery problem.

One important notion which has many real-world applications and is studied widely is the pattern of association rules [AS94], giving information about approximate dependencies between values in so called transactional data. Association rules are often used in commercial "market basket" research, where a database is mined for information of the form: "Which articles do customers buy together in one transaction?" Discovering such rules is usually accomplished by generating *rule candidates* based on value frequencies in a database and then *growing* candidate sets attribute by attribute, as long as some minimum *support* for the rule is maintained. The concepts used in association rule mining (in particular the *apriori*-Strategy described above) are related to our problem but not directly applicable since association rules are probabilistic in nature [AS94], whereas we are generally looking for *constraints*, i.e., exact dependencies. Even when we weaken our requirements and no longer look for *exact* dependencies, the *apriori* mechanism used in association rule mining does not provide a feasible solution for our problem, as its computational complexity is too high for our purposes.

Another important class of related work is the class of functional dependencies. A functional dependency is a constraint on a set of attributes $(A_1, A_2, \ldots, A_k, X)$ in a relation $R$, specifying that for any two tuples $t_1$ and $t_2$ from $R$, the following conditions holds:

$$t_1[A_1, A_2, \ldots, A_k] = t_2[A_1, A_2, \ldots, A_k] \implies t_1[X] = t_2[X].$$

The derivation of functional dependencies through inference rules has been treated extensively [Mit83, KMRS92, CFP82, MG90]. The problem of finding evidence for functional dependencies from the *extent* of relations has also been considered. Several projects deal with the question how to efficiently find candidates for functional dependencies from among the attributes of a relation [BB95a, SF93].

Functional dependencies and inclusion dependencies are related but have some important differences. In particular, functional dependencies generally are defined only *within* one relation, whereas the natural purpose of inclusion dependencies is to define relationships across two different relations. Mitchell [Mit83] also considers inclusion dependencies within one relation. Functional and inclusion dependencies are related in the sense that they both constrain possible valid database states and are thus helpful in database design. However, for our purpose of discovering information about relationships across unknown databases the case of inclusion dependencies is more useful.

Commercial database systems also define *referential integrity constraints.* Such constraints ensure that any value in an attribute $A$ in relation $R$ exists in an attribute $B$ in relation $S$. Clearly, those are unary inclusion dependencies. The practical use of such constraints is to ensure that after database normalization, dependent relations will contain in their (foreign) keys all values necessary for a join (Fig. 3.1). Referential integrity constraints are typically defined on those attributes between which a functional dependency held before normalization. So in turn, discovering inclusion dependencies between relations might provide some information about a database that suggests the existence of a referential integrity constraint.

However, the existing functional dependency algorithms (e.g., [BB95a, SF93]) cannot be used to derive inclusion dependencies (or to deduce inclusion dependencies from functional dependencies), as the complexity of our problem is much higher than the complexity of functional dependency discovery. A few rules for the deduction of INDs from functional dependencies *and* other INDs are given in [Mit83], but the are only applicable in very specific cases. Functional dependencies that are defined within one relation (which is true for most such dependencies) cannot directly help in the detection of general (inter-relation) inclusion dependencies.

# Chapter 4

# Algorithm **FIND**$_2$ for the Discovery of Inclusion Dependencies

## 4.1 Finding Inclusion Relationships across Databases

We can now state our problem in a concise manner:

> *Given a set of relations $R^* = \{R_1, \ldots, R_n\}$ stored in one or more DBMS, find the generating set of inclusion dependencies (INDs) between any two relations in $R^*$.*

Note that we are not looking for *all* INDs but for a (minimal) generating set. Since all other INDs can be derived from the generating set by projection and permutation, it is sufficient to consider only this set. As we will see shortly, trying to generate *all* INDs is impractical due to their large number while a generating set can be found more efficiently.

### 4.1.1 Assumptions

For the discovery of INDs, we will assume that INDs can be defined in the underlying data model. That is certainly true for the relational model, but object-oriented models and to a certain extent semi-structured models also have a notion of data inclusion. We furthermore require that the data model must include the concept of *attributes* ("columns" of data), cannot have complex objects (such as nested relations) and cannot have pointers or cross-references between data objects.

Throughout this dissertation, we will assume that equality between tuples is a binary function, i.e., two tuples (and thus each matching pair of their values) are either equal or they are not equal. There is some work on the value-matching problem, which asks for "approximate" equality between values across two attributes [Coh98]. In this work, we will not focus on that problem and rather assume that we can compare two tuples and decide whether they are equal or not.

Furthermore, we will restrict ourselves to databases that are queryable with some kind of query language, for example SQL. The language should support the following types of queries (for relational databases):

1. find the set of names of all relational tables in the database

2. find the names and types of all attributes in a table

3. decide if a given IND holds in the database (form a set difference between two projections of relations and decide whether the set is empty).

The first two types of queries access the data dictionary of a database. Those queries are supported by all commercial SQL-database systems (as well as by wrappers like JDBC). Deciding whether the tuples in a set of attributes in one relation form a subset of the tuples in another set of attributes in another relation is supported by the standard SQL-`minus`-statement. A query to that effect can be formulated in SQL (see Sec. 5.1)

### 4.1.2 A Three-Staged Solution to the IND-Finding Problem

The problem as given above asks for complex relationships among databases. Relational databases have an inherent three-layer hierarchy: databases consist of relations, which in turn consist of attributes. Analogous layers exist for other data models as well, such as database-object-attribute in object-oriented databases. Some additional restrictions may be necessary for such more expressive data models to support the concept of inclusion dependencies. For example, the semi-structured data model as used in XML allows for different *multiplicities* of attributes, which is difficult to map to the concept of INDs. In order to discover INDs in such databases, our approach requires that no attribute occurs more than once in any one data object.

The existence of such layers suggests a three-layered strategy to discover relationships between databases: compare attributes, compare relations, and, finally, compare databases. It is clear that two relations whose attributes are not related cannot in turn be related, and likewise a relationship between databases requires relationships between their relations. Thus, there are three necessary stages in any algorithm that could solve the above problem:

1. $\mathsf{FIND}_n$: finding valid INDs between *a set* of given relations in a set of databases (the general problem)[1],

2. $\mathsf{FIND}_2$: finding valid INDs between *a pair* of given relations,

3. CHECK: determining whether a given IND is valid.

A general overview over our approach is shown in Fig. 4.1. Those stages do not necessarily need to be executed separately, but treating them separately helps to gain insights into the nature of the problem.

A simple algorithm (i.e., pairwise comparison) for the first stage could express the general problem for $n$ relations as $\binom{n}{2}$ problems on pairs of relations. Improvements are possible for example by using the transitivity property of INDs (Section 4.1.3). Some ideas are given as future work (Sec. 14.2).

The second stage needs to find maximal valid INDs (i.e., a generating set for each pair of relations considered) with a minimal number of single IND checks. The focus in this stage is not how to check the validity of INDs against a database state, but in finding a generating set of INDs with a minimal number of checks. Each check is assumed to take unit time, and we will consider ways to reduce that time in the third stage below. We will

[1] FIND stands for **F**ind **In**clusion **D**ependencies.

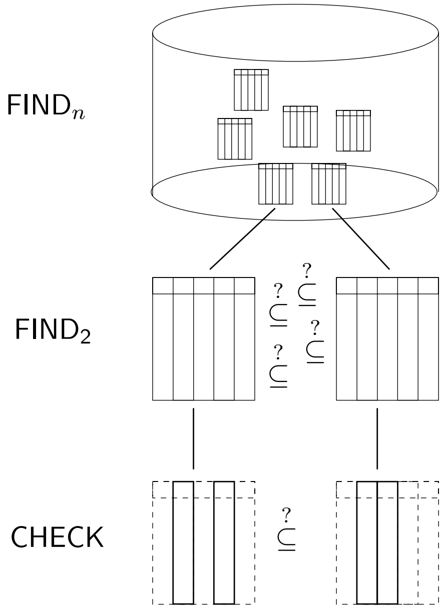


Figure 4.1: The Three Stages of Inclusion Dependency Discovery

show in Sec. 4.2 that the problem of finding a generating set of INDs is NP-hard for the general case (based on well known complexity results for related problems). In fact, the number of possible (valid or invalid) INDs between two relations of $k$ attributes is very high so that an exhaustive search of all INDs is intractable even for relations with 7 or 8 attributes. For example, there are $1,441,728$ possible INDs between two relations with 8 attributes each, without considering permutations, see Sec. 4.2. However, it is not likely that many distinct valid INDs exist between two real-world relations. Therefore, the number of *maximal* INDs (i.e., the size of the generating set of existing INDs) is likely to be small for real-world data, and giving a solution to the general problem remains possible.

The third stage (checking a particular IND), which has to be executed for every IND generated in Step 2, involves querying one or two database systems in order to determine an inclusion between two sets of attributes across two relations. When the relations exist in the same database, they can be queried with an SQL-`minus`-query. Relational database systems using external sorting may effectively answer such queries in linear time in the size of input databases (Chapter 6). There are many approaches for improvements here that we will discuss in Section 5.1. If the relations are in two different database systems, a single SQL query is not sufficient. In that case, other techniques can be applied. Some ideas are given in Section 5.1 as well.

Overall, the stage with the highest complexity is the stage of finding valid INDs between two relations. Therefore, we expect the greatest improvements in runtime for a general algorithm at this stage. This is therefore the problem upon which we focus our attention.

It is possible to make use of additional information about the databases in question in order to reduce the amount of necessary work. Some concepts that we will explore further in this text are heuristics on the *meaning* of attribute and relation names (ontologies) and other properties of data in relations, such as data types and value distribution histograms.

### 4.1.3 Comparing two Databases

As stated in Section 4.1, we are looking for a generating set of INDs among a set of relations, meaning that the output of our algorithm should be only those INDs that cannot be derived from other INDs. In particular, this excludes INDs that can be obtained from other INDs by transitivity (Sec. 3.3).

However, note that transitivity cannot be universally applied to our problem. For example, consider an IND finding problem in which two valid maximal INDs have been found: $R[A, B] \subseteq S[C, D]$ and $S[C, D] \subseteq T[E, F]$. Those two INDs together are not sufficient to reason about the validity of a third IND $R[A, B, X] \subseteq T[E, F, X]$, even though it may also be valid and maximal. That means that even if a new IND is found by transitivity, there is no guarantee that this IND is maximal and thus part of the generating set. Therefore, it seems necessary for the time being to exhaustively search all pairs of relations in a database for inclusion dependencies. Transitivity can help with reducing the search space somewhat, but all relations under consideration will have to be accessed. For a set of $n$ relations, that means that $\binom{n}{2}$ pairs of relations have to be considered. All INDs found must then be tested for mutual dependencies, and those INDs that can be derived from others must be removed from the solution.

Thus, a simple algorithm to solve the general IND-finding problem is as follows [2]:

```
$\mathsf{FIND}_n$(Database $\mathcal{D}_1$, Database $\mathcal{D}_2$)
    Set $INDs \leftarrow \emptyset$
    **forall** $(R \in \mathcal{D}_1)$
        **forall** $(S \in \mathcal{D}_2)$
            **if** $(R \neq S)$
                $INDs \leftarrow INDs \cup \mathsf{FIND}_2(R, S)$    //defined in Sec. 4.2
    removeDerivableINDs($INDs$)
```

[2] For all pseudocode in this dissertation, we use a $C$-like syntax, i.e., variables are defined by writing their type before their name. The symbol $\leftarrow$ is used to denote assignment. The scope of complex statements is marked by indentation.

If algorithm $\mathsf{FIND}_2$ is assumed to run at unit cost, the algorithm $\mathsf{FIND}_n$ runs in $O(n^2)$ in the number of relations in the database. Thus, we did not pursue optimizations on this algorithm further since the potential savings in runtime at this stage are small compared to the possible optimizations in the discovery of INDs between two *given* relations. Missaoui and Godin [MG90] presented an algorithm for the efficient computation of the closure of INDs under transitivity (for relation schemes with many relations). The authors give simple algorithms for polynomial-time computation of closure and computation of a minimal cover of INDs for a multi-relation IND inference problem using transitivity only.

## 4.2 Finding Inclusion Dependencies between Two Relations

In this section, we consider the problem of finding inclusion dependencies between two given relations:

> *Consider two relations $R$ and $S$. Find a generating set $\mathcal{G}(\Sigma)$ of valid inclusion dependencies $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$ of the form $R[A_R] \subseteq S[A_S]$ with $A_R$ a subsequence of attributes from $R$ and $A_S$ a subsequence of attributes from $S$.*

### 4.2.1 Complexity

We will consider the worst-case complexity of the problem as a function of the number of attributes in both relations. The maximum number of distinct

INDs between two relations can be computed as follows. For two relations $R$ and $S$ with $k_R$ and $k_S$ attributes, respectively, one can form $k_R \cdot k_S$ *unary* INDs. It is possible for all such INDs to be valid at the same time, namely when all attributes in both relations have exactly the same data (which is not likely to occur in practice). The number of $k$-ary INDs in general is determined by the number of pairs of $k$-ary subsets of the attributes in each relation. For each such pair, there are $k!$ INDs, since each permutation of *one side* of an IND, while keeping the other side unchanged, gives a new IND (i.e., an IND not equal to any previously generated IND). Permutations of *both* sides do not lead to new INDs (see Def. 3.5), as this process would generate INDs *equal* to previously generated ones. Therefore, the number of $k$-ary INDs, denoted by $I_k$, is

$$I_k(k_R, k_S) = \binom{k_R}{k} \cdot \binom{k_S}{k} \cdot k! \tag{4.1}$$

Assuming without loss of generality that $k_S < k_R$, the total number of INDs between $R$ and $S$, denoted by $I$, is

$$\begin{aligned} I(k_R, k_S) &= \sum_{i=1}^{k_S} \binom{k_R}{i} \cdot \binom{k_S}{i} \cdot i! \qquad (4.2) \\ &= \sum_{i=1}^{k_S} \frac{k_R! \cdot k_S!}{(k_R - i)! \cdot (k_S - i)! \cdot i!} \end{aligned}$$

A naïve brute-force IND-finding algorithm for a pair of relations could first generate all possible INDs and then test each of them for validity. It would thus have a complexity in at least $O(I(k_R, k_S))$, even if IND-testing could be done in constant time. Clearly, since this number grows extremely fast, it is not meaningful to test for or even generate all IND candidates for a pair of relations, even if those relations have a small number (say $< 10$) of attributes. Also, as $I(k_R, k_S) \notin O({k_R}^n \cdot {k_S}^m)$ for any finite $n, m$, the problem cannot be solved in polynomial time since its solution cannot be enumerated in polynomial time. It is possible to find a *generating set* of valid INDs without enumerating all INDs, but even that problem is NP-hard, as we show in Sec. 4.2.8.

### 4.2.2 Solution Approach for Finding INDs Between Two Relations

In order to improve the runtime of any algorithm for this problem, it is necessary to reduce the problem space. In this section, we will concentrate on how to reduce the *number* of individual IND-checks, rather than on ways to speed up such checks. Databases carry additional information about their data (such as data domains, database statistics, indices, attribute name ontologies), and such information can be used to avoid testing all INDs for any non-trivial problem. That is, not all possible INDs that our system considers will have to be checked against the database, but rather many IND checks can be answered in very short time. We will give details on the possibilities of such search-space reductions in Section 5.1.

We now describe a general framework for the solution of the IND discovery problem. Recall from Axiom 3.2, p. 46 that a $k$-ary valid IND implies certain $i$-ary valid INDs, for $i < k$. We observe that the Axiom 3.2 does not change if we allow only *ordered* sequences of integers to serve as indices for generated sub-INDs (due to the definition of equality, Def. 3.5).

Note that since we are concerned with INDs between two given relations only, Axiom 3.2 in the form of the above observation also states the only derivation rule (out of the complete axiomatization system given by the three Axioms in Section 3.2.2) that can be used to *derive* new INDs (see Def. 3.4). Thus, the set of INDs obtained from an IND $\sigma_0$ through projection only is equivalent to the set $\{\sigma | \ \sigma_0 \models \sigma\}$.

We also make an observation about the *number* of INDs that can be derived as projections,i.e.,, subsets of other INDs.

**Lemma 4.1** *A $k$-ary valid IND implies $\binom{k}{m}$ $m$-ary valid INDs, for any $1 \leq m \leq k$.*

Let us assume a hypothetical IND-finding strategy that generates all possible INDs, and then tests each IND and marks it as either valid or invalid. In such a strategy, one would define a data structure that holds INDs plus a *state* from the set $\{\mathsf{unknown}, \mathsf{valid}, \mathsf{invalid}\}$ for each IND.

1. Check high-arity IND candidates. If a *valid* $k$-ary IND $\sigma_0$ is found, mark all INDs derivable from $\sigma_0$ (i.e., all elements of the set $\{\sigma | \sigma_0 \models \sigma\}$) as valid. Note that all those INDs have arities $< k$.

2. Check low-arity IND candidates. If an *invalid* $k$-ary IND $\sigma_0$ is found, mark all those INDs $\sigma$ as invalid from which $\sigma_0$ would be derivable if it were valid (i.e., all elements of the set $\{\sigma | \sigma \models \sigma_0\}$).

This strategy would still require to generate explicitly or implicitly all INDs, which is not feasible. However, it suggests algorithm $\mathsf{simpleFIND}_2$ in

Fig. 4.2. It solves the IND discovery problem for two relations $R$ and $S$ with $k$ attributes, respectively.

ALGORITHM simpleFIND$_2$ for two relations with $k$ attributes each

1. Set (a local variable) $m = 1$. Generate all unary INDs. There are exactly $k^2$ such INDs. After testing each of them, retain only the valid INDs (generally much fewer than $k^2$) and store them in a set $\Sigma_1$.
2. Increase $m$ by $1$.
3. Generate only those $m$-ary INDs whose implied $(m-1)$-ary INDs are *all* members of $\Sigma_{m-1}$ (i.e., from which *no* IND *not* in $\Sigma_{m-1}$ can be derived).
4. Test these $m$-ary INDs and retain only the valid ones in a set $\Sigma_m$.
5. Repeat from step 2 until $m = k$.

Figure 4.2: A Simple Algorithm simpleFIND$_2$ for the Two-Relation IND-Finding Problem.

Clearly, this algorithm still has very high complexity as the number of INDs potentially to consider is prohibitively high. We will describe an optimization on this algorithm in the next section. In order to assess the correctness of algorithm simpleFIND$_2$, consider the following observation.

**Observation 4.1** *The validity of all $(m-1)$-ary INDs that are derivable from an $m$-ary IND $\sigma$ is a necessary but not a sufficient condition for the validity of $\sigma$.*

Counterexamples can easily be obtained (e.g., Fig. 4.3). However, note that in order for this situation to occur, a total of $\binom{m}{m-1} = m$ tuples in a relation $S$ (in an IND-finding problem between relations $R$ and $S$) have to be related to each other in a certain way. More specifically, for each tuple in an $m$-set of tuples from $S$, $m-1$ of that tuple's attributes have to have the same value as the corresponding attributes in one fixed tuple in $R$. In Fig. 4.3, this condition is satisfied by the tuple $(3, 6, 9) \in R$ and the set of tuples $\{(3, 6, -1), (-1, 6, 9), (3, -1, 9)\} \subset S$. This situation seems unlikely in practice. Of course, any algorithm for finding INDs must nonetheless take this "degenerate" case into account. That means that the testing step (Step 4) in algorithm simpleFIND$_2$ is necessary to ensure the correct solution.

R

| $A_1$ | $A_2$ | $A_3$ |
|---|---|---|
| 1 | 4 | 7 |
| 2 | 5 | 8 |
| 3 | 6 | 9 |

S

| $B_1$ | $B_2$ | $B_3$ |
|---|---|---|
| 1 | 4 | 7 |
| 2 | 5 | 8 |
| 3 | 6 | -1 |
| -1 | 6 | 9 |
| 3 | -1 | 9 |

$R[A_1, A_2] \subseteq S[B_1, B_2]$ is valid.
$R[A_2, A_3] \subseteq S[B_2, B_3]$ is valid.
$R[A_1, A_3] \subseteq S[B_1, B_3]$ is valid.
$R[A_1, A_2, A_3] \subseteq S[B_1, B_2, B_3]$ is not valid.

Figure 4.3: Validity of All Derived INDs Is Not a Sufficient Validity Test.

### 4.2.3 Mapping to a Graph Problem

With the ideas from the previous section as a starting point, we propose a mapping of our problem into a more tractable graph problem. This will also give us additional insights into the inherent complexity of the problem. For clarity, we will give brief definitions of the concepts used.

**Definition 4.1 ($k$-hypergraph)** *A $k$-**uniform hypergraph** (or $k$-**hypergraph**) is a pair $G = (V, E)$ of the set $V$ of nodes and the set $E$ of edges. An element $e \in E$ is a set with cardinality $k$ of pairwise distinct elements from*

*$V$, denoted by $\{v_1, \ldots, v_k\}$. An element $e \in E$ is called a $k$-*hyperedge*. $k$ is called the* rank *of graph $G$.*

**Example 4.1** *An example for a 3-hypergraph is given in Figure 4.4. This graph has 6 edges: $\{1, 2, 3\}$, $\{1, 2, 4\}$, $\{1, 2, 5\}$, $\{1, 3, 4\}$, $\{2, 3, 4\}$, and $\{3, 4, 5\}$. Each edge is drawn as three lines, connected by a full circle •.*

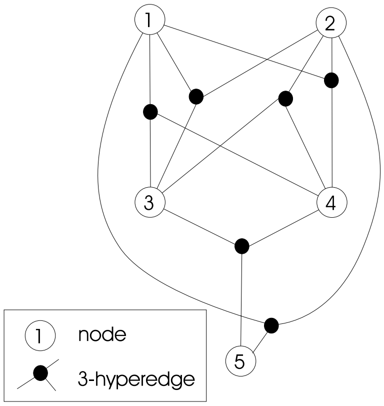


Figure 4.4: A 3-hypergraph with 6 edges.

Clearly, an undirected graph is a special case of $k$-hypergraph with $k = 2$. Note that the requirements that elements of edges are pairwise distinct disallows self-loops. The above definition extends the concept of undirected graphs to $k$-hypergraphs. However note that we treat edges as *sets* of nodes rather than a sequence (tuple) as the edges are not "directed".

As $k$-hypergraphs are undirected, they can be stored in memory efficiently using an extension of the concept of adjacency matrix in graphs. We impose an order on the nodes in $V$. For a (regular) graph, only the upper triangle of the adjacency matrix is stored (i.e., those edges $\{v_1, v_2\}$ for which

$v_1 < v_2$), while for $k$-hypergraphs analogously, only those edges are stored whose nodes are listed in increasing order.

**Definition 4.2 (Clique)** *Let $G = (V, E)$ be a graph. A* ***clique*** *$C$ is a set $C \subseteq V$ such that $\forall v_1, v_2 \in C : \{v_1, v_2\} \in E$. A single node with no adjacent edges is a clique of cardinality 1.*

Thus, a clique is a set of nodes, not a graph. This is a common definition as there is no need to speak of the set of edges in a clique. The edges of a clique $C$ are thus trivially defined as the set of edges in the complete graph induced by the set of nodes $C$.

**Definition 4.3 (hyperclique)** *Let $G = (V, E)$ be a $k$-hypergraph. A* ***hyperclique*** *is a set $C \subseteq V$ such that for each $k$-subset $S$ of distinct nodes from $C$, the edge implied by $S$ exists in $E$. The cardinality of a hyperclique $C$ is the number of nodes in $C$. A single node with no adjacent edges is a hyperclique of cardinality 1.*

Note that a $k$-hypergraph (with $k > 2$) cannot have hypercliques with cardinalities $2 \ldots k - 1$. A (hyper)clique is either a node with no edges or it must have at least as many nodes as there are nodes in an edge, i.e., $k$. Since a hyperclique is simply a set of nodes, there is no need to speak of $k$-hypercliques as long as it is unambiguous in the context. Thus, we sometimes speak of cliques when both cliques and hypercliques are meant.

Analogous to the case for cliques, if $k$ is known, the edges of a hyperclique $C$ are defined as the set of all possible undirected edges ($k$-sets of nodes) in the complete $k$-hypergraph induced by $C$.

**Corollary 4.1** *Consider a $k$-hypergraph $G = (V, E)$. A set $C \subseteq V$ is a hyperclique if the $k$-hypergraph $G_C$ induced by $C$ has $\binom{|C|}{k}$ edges.*

**Proof:** This property follows from the observation that the nodes of a $k$-hypergraph $G_C$ form a hyperclique $C$ if all possible edges between these nodes exist in $G$ (by Def. 4.3). There are exactly as many possible distinct edges in such a $k$-hypergraph as there are $k$-subsets of $|C|$ elements, such that the number of possible edges is $\binom{|C|}{k}$. **q.e.d.** □

**Example 4.2** *Consider the* 3*-hypergraph in Fig. 4.4. The set of all its nodes does not form a hyperclique, since for example the edge* $\{1, 3, 5\}$ *is missing. However, the set of nodes* $\{1, 2, 3, 4\}$ *forms a hyperclique (to be exact: a* 3*-hyperclique with cardinality 4). Further hypercliques are the sets* $\{1, 2, 5\}$ *and* $\{3, 4, 5\}$

Again, a clique is a special case of a hyperclique in a 2-hypergraph. We now define the concept of *degree* of a node in a $k$-hypergraph.

**Definition 4.4 (degree of a node)** *The **degree** of a node $v \in V$ in a $k$-hypergraph $G = (V, E)$ is the number of edges that have $v$ as element. More formally,* $\deg(v) = |\{e \in E | v \in e\}|$.

This definition applies to both graphs and $k$-hypergraphs.

**Example 4.3** *In Fig. 4.4, nodes* $1, \ldots, 4$ *each have degree 4, while node* 5 *has degree 2. Note that node* 5 *is "adjacent" to all nodes* 1–4*, such that the number of "adjacent" nodes is not its degree.*

**Mapping the Set of Inclusion Dependencies to a Graph.** Consider the problem of finding inclusion dependencies between two relations $R$ and $S$. Let $R$ have $k_R$ attributes and $S$ have $k_S$ attributes and assume without loss of generality that $k_S < k_R$. The mapping is given in Fig. 4.5.

1. Create a set $V$ whose elements are all *unary valid INDs* between $R$ and $S$ (i.e., all INDs of the form $R[a_i] \subseteq S[b_i]$).
2. Create a graph (2-hypergraph) as follows:
   (a) Create a set $E_2$ whose elements are all *binary valid INDs* between $R$ and $S$. Note that each binary valid IND $\sigma$ can be seen as a set of exactly two unary valid INDs $\sigma_1$ and $\sigma_2$, by Corollary 3.1.
   (b) Create a graph $G_2 = (V, E_2)$. Note that its *nodes* are all unary valid INDs between $R$ and $S$, while its *edges* are all binary valid INDs between those relations.
3. Create hypergraphs as follows:
   (a) For $i = 3, \ldots, k_S$, create the set $E_i$ to be composed of all $i$-ary valid INDs, respectively. As per the assumption stated in the text, $k_S$ is the maximal arity for an IND in this problem.
   (b) For each $i = 3, \ldots, k_S$, create one $i$-Hypergraph $G_i = (V, E_i)$.

Figure 4.5: Mapping a Set of INDs to a Graph

The IND-finding problem can now be expressed as the problem of constructing the above graphs. Note that this mapping does not yet support the notion of *generating set* (Def. 3.6) but rather expresses *all* valid INDs between two relations. However, a slight change in the mapping (by simply keeping only valid INDs *not implied* by higher-arity valid INDs) would correspond to this refined notion.

Now recall that, by Lemma 4.1, a $k$-ary valid IND implies $\binom{k}{m}$ $m$-ary INDs (with $m \leq k$). We observe the following theorem.

**Theorem 4.1** *Given the relations $R$ and $S$, with $k_R$ and $k_S$ attributes, respectively, consider a collection of $k$-Hypergraphs $\{G_2, G_3, \ldots, G_{k_S}\}$ representing the INDs between $R$ and $S$ (as defined in Fig. 4.5). Furthermore, let $\sigma_k$ be a $k$-ary valid IND between $R$ and $S$. For a number $m$, with $m < k$, construct a set $E_m$ of all the $m$-ary INDs implied by $\sigma_k$, which are all $m$-hyperedges in $G_m$. Then the set of all nodes that are elements of any edge in $E_m$ forms an $m$-Hyperclique in $G_m$, or alternatively, $E_m$ is the set of edges of an $m$-Hyperclique in $G_m$.*

**Proof:** By Definition 4.3, the set of edges of a hyperclique $C$ in an $m$-hypergraph $G_m = (V, E)$ correspond to exactly all $m$-subsets of nodes from $C$. Also, the $m$-ary INDs implied by an IND $\sigma_k$ (with $m < k$) are exactly all $m$-subsets of the set of nodes implied by $\sigma_k$. If $C$ is the set of nodes implied by $\sigma_k$, clearly there is a trivial isomorphic mapping between the edges of $C$ and the $m$-ary INDs implied by $\sigma_k$. **q.e.d.** □

We have now reduced the problem of finding INDs to the problem of finding hypercliques in a collection of $k$-hypergraphs.

### 4.2.4 The Clique-Finding Problem

The **Clique-Finding Problem** [GGL95] (also called the Maximum Clique Problem) is a well known graph problem. It is usually defined as follows:

> ***Clique-Finding Problem**: For a graph $G = (V, E)$, find all subsets of $V$ that form complete subgraphs in $G$ and are maximal with respect to that property.*

We observe the following properties:

1. The Clique-Finding Problem is NP-complete in the number of nodes $|V|$. A proof is given, for example, in [GJ79].

2. The Clique-Finding Problem extends naturally to $k$-hypergraphs, and must be at least as hard as the Clique-Finding Problem for graphs, such that it is also NP-hard.

3. The Clique-Finding Problem is closely related to the IND-finding problem discussed in this work, when mapped as described in Theorem 4.1. We will elaborate on this point in the next section.

There is substantial research on the Clique-Finding Problem. Important results include that finding a maximum clique is NP-hard [GJ79] and that even the approximation of the problem (finding a clique whose size is larger than a constant fraction of the size of the largest clique) is NP-hard [ALM$^+$92]. This complexity is mainly due to an exponential number of possible cliques in a graph. The worst case for the number of cliques is given by so called Moon-Moser-Graphs [MM65], which are graphs with $3k$ nodes (for $k \in \mathbb{N}$) and are constructed as the complement of $k$ disjoint 3-cliques [BK73]. They contain $3^k$ cliques.

However, there are algorithms that will take polynomial time *for the generation of each clique*, such that the complexity can be expressed by a polynomial in the number of cliques.

**Finding all Cliques of an Undirected Graph.** In order to find cliques in the special case of a 2-hypergraph (i.e., in a regular graph), an algorithm by Bron and Kerbosch [BK73] can be used. The algorithm finds all maximal cliques in a given graph $G$ and uses a backtracking strategy and several heuristics. The complete Bron/Kerbosch algorithm is given in Appendix A.

Essentially, the Bron/Kerbosch algorithm performs a depth-first tree traversal of a tree of all clique candidates, backtracking when one of several conditions for cliques does no longer hold. The nodes of that search tree are taken from the nodes of the input graph, and the tree is then constructed by ensuring that higher-degree nodes are closer to the root. In particular, the algorithm makes use of the property that in a fully connected graph with $k$ nodes, all nodes must have degree $k-1$, which is a necessary and sufficient condition for full connectivity (proven below for the general case of $k$-hypergraphs, Thm. 4.2).

The algorithm tends to find larger cliques first and achieves a runtime behavior that is dependent on the number of cliques in a graph, but not on the number of nodes. We give a complexity analysis in the context of our hyperclique finding algorithm in Section 4.2.8.

### 4.2.5 Finding Hypercliques

While the Bron/Kerbosch-algorithm is efficient for graphs, it makes use of a property of such graphs that does not extend to $k$-hypergraphs. The algorithm relies on the following fact.

**Corollary 4.2** *Let $C$ be a clique in a graph $G = (V, E)$. Let $v$ be a node with $v \notin C$ and $\forall v_i \in C : \exists \{v_i, v\} \in E$. Then, $C \cup \{v\}$ is a clique.*

The low complexity of the Bron/Kerbosch algorithm is due to the fact that cliques can be "grown" very efficiently by simply checking the *connected*-property of a new node to each node of a previously found sub-clique. That is, given a clique, adding a node to this clique that is connected to each clique element will create a larger clique. Given a candidate node, this property can be tested in a time linear in the number of nodes in the graph.

On the other hand, this is not true for $k$-hypergraphs with $k > 2$ as the concept of "connected" is more complex for $k$-hypergraphs. Growing hypercliques in the same way requires to test for the existence of an edge between the new node and any $k-1$-*subset* of the nodes of the existing clique, which is exponentially more expensive. A further complication arises from the fact that the Bron/Kerbosch algorithm keeps a counter for each node $v$ tracking the number of nodes *not connected* to $v$ (variable *nod* in the algorithm in Appendix A). Clearly, this "non-connectedness" property is more complex for $k$-hypergraphs. Therefore, a direct extension of the Bron/Kerbosch algorithm would lead to an algorithm with intractable complexity.

A naïve hyperclique-finding algorithm, which would enumerate all possible subsets of nodes in a graph and test each for the clique property, is also clearly not feasible, as it is exponential in the number of nodes in the graph.

However, it is possible to find a better algorithm. The following clique-criterion, which is a generalization of the same criterion for graphs, can be used to find cliques in $k$-hypergraphs.

**Theorem 4.2** *A $k$-hypergraph $G_k$ with $n$ nodes is a hyperclique if and only if each of its nodes has a degree of $\binom{n-1}{k-1}$.*

**Proof:** We have to show that the condition is necessary and sufficient. First, we show necessity.

Recall that we define an edge in a $k$-hypergraph as a set of nodes. By Cor. 4.1, a fully connected $k$-hypergraph with $n$ nodes has $\binom{n}{k}$ edges. (To see this, set $C = V$ in Cor. 4.1.) Each edge is a set of $k$ nodes, and edges in a $k$-hypergraph are pairwise distinct (Def. 4.1). Since there are exactly $\binom{n}{k}$ subsets of size $k$ over a set of size $n$, there is one edge for each possible combination of $k$ nodes from $n$ nodes, such that all nodes must be members of an equal number of edges, i.e., must have the same degree.

Now, note that all edges together contain a total of $\binom{n}{k} \cdot k$ (not necessarily distinct) nodes. Since there are only $n$ different nodes, and all nodes have the same degree, each node must have a degree of $\binom{n}{k} \cdot k/n = \binom{n-1}{k-1}$.

For sufficiency, note that the existence of $n$ nodes each with degree $d = \binom{n-1}{k-1}$ implies the existence of $e = n \cdot d$ edges. Each edge connects exactly $k$ nodes. Since all edges are distinct, there must be at least $e/k$ edges. Now $e/k = \binom{n-1}{k-1} \cdot n/k = \binom{n}{k}$, which is the condition for a hyperclique (by Corollary 4.1) **q.e.d.** □

**Example 4.4** *Consider the induced subgraph of the hypergraph in Fig. 4.4 which has only the nodes $1, 2, 3, 4$. Each node of that subgraph has degree $\binom{4-1}{3-1} = 3$. This subgraph is fully connected (since the nodes in each 3-subset of its four nodes are connected by a 3-hyperedge).*

The above proof subsumes the proof for the analogous clique-criterion on traditional cliques (i.e., 2-Hypercliques). We have now established a simple criterion for a hyperclique, in that a minimal degree is required for a node to be part of a hyperclique of a certain size (number of nodes). Thus, the degree of a node gives us an upper bound on the size of any clique in which this node is involved.

*Note that it is not sufficient for a* subset *of nodes in a graph to have a certain minimum degree in order to be a clique.* Theorem 4.2 applies only if the entire set of nodes of a graph forms a clique.

When storing a $k$-hypergraph $G_k = (V, E)$ simply as a list of its edges, determining the degree of a node takes time in $O(|E|)$ (if set membership can be tested in constant time), as opposed to the more efficient $O(|V|)$ for regular graphs stored as adjacency matrices. The number of edges in a $k$-hypergraph is in $O(|V|^k)$ (since $\binom{n}{k} \in O(n^k)$), such that storing the graph or determining its node degrees may not be feasible for large graphs. However, we will see that for our purpose, the size of graphs will usually be manageable, and in fact, in our implementation we used a list of edges as the storage principle.

### Algorithm HYPERCLIQUE to Find Cliques in $k$-Uniform Hypergraphs

With the background given, we can now describe an algorithm to find hypercliques in $k$-hypergraphs. Algorithm HYPERCLIQUE uses several branch-and-bound strategies to reduce a graph to smaller subgraphs, while finding cliques. It quickly finds some cliques in relatively large graphs by removing edges from a graph that can no longer contribute to cliques. In some cases, this strategy finds all cliques, especially if the graph is sparse. However, the heuristics may fail on certain graphs, as explained below. Then, a recursive strategy is used that reduces the graph by removing certain edges and working on subgraphs. The algorithm will always find all cliques in a given hypergraph. The algorithm has exponential worst-time complexity but will finish quickly on a much larger class of graphs than the naïve algorithm. For comparison, such a naïve algorithm that can also be used for this problem is given in Appendix B

Fig. 4.6 gives a summary of algorithm HYPERCLIQUE in pseudo code. In the following section, we explain the details of this algorithm. We proceed from bottom to top, first explaining the algorithm's main functionality, then introducing two search-space reducing steps to reduce the size of the hypergraph, and finally concluding with a description of the entire algorithm.

**Overall Strategy of Algorithm HYPERCLIQUE** The HYPERCLIQUE algorithm applies a number of branch-and-bound strategies to find cliques in less time than brute-force enumeration would allow. The basic algorithm will find some hypercliques very fast, and for relatively sparse graphs will find all hypercliques in a graph. Augmented by branch-and-bound strategies, algorithm HYPERCLIQUE then applies a recursive procedure that reduces the size of the input graph. In all cases, algorithm HYPERCLIQUE finds all hypercliques in a given $k$-hypergraph.

In order to explain the algorithm, we first make a number of observations about $k$-hypergraphs.

```
ALGORITHM HYPERCLIQUE

INPUT:
$k$-Uniform Hypergraph $G_k = (V, E)$

OUTPUT:
Set $\mathit{result}$, containing all cliques in $G_k$

function findHypercliques(Graph $G_k$)
$\mathit{result} \leftarrow \emptyset$    //result is a variable global to this function
Set $S$    //a clique candidate
$\mathit{reducible} \leftarrow$ false
do
    $E^* \leftarrow \emptyset$    //edges that are element of more than one clique
    forall $(e \in E)$    //$E$ is set of edges in $G_k$
        $S \leftarrow$ generateCliqueCandidate($G_k, e$)
        if ($S$ is clique )    //Thm. 4.2
            if ($S$ is not subset of any element of $\mathit{result}$)
                $\mathit{result} \leftarrow \mathit{result} \cup \{S\}$
            $\mathit{reducible} \leftarrow$ true
        else
            $E^* \leftarrow E^* \cup \{e\}$
    if ($\neg\mathit{reducible}$)
        do
            $e \leftarrow$ a random edge from $E$
            $S \leftarrow$ generateCliqueCandidate($G_k, e$)
            $G_1(V_1, E_1) \leftarrow$ subgraph of $G_k$ induced by $S$
            $G_2(V_2, E_2) \leftarrow (V, E \backslash e)$
        while $(|V_1| \neq |V|)$    //end of do-while-loop
        $\mathit{result} \leftarrow \mathit{result} \cup$ findHypercliques($G_1$)
        $\mathit{result} \leftarrow \mathit{result} \cup$ findHypercliques($G_2$)
    $G_k \leftarrow (V, E^*)$    //reduced graph $G_k$
while $(G_k \neq \emptyset \wedge \mathit{reducible})$    //end of do-while-loop
```

Figure 4.6: Algorithm HYPERCLIQUE for Finding Cliques in a $k$-Uniform Hypergraph

**Lemma 4.2** *Consider a $k$-hypergraph $G_k = (V, E)$ and a set $V_k$ of $k$ nodes $v_1, \ldots, v_k$ from $G_k$. Consider another node $v \in V \backslash V_k$. If there is* any *$(k-1)$-subset $V_{k-1}$ of $V_k$ such that $V_{k-1} \cup \{v\}$ does not form an edge in $E$, then the set $V_k \cup \{v\}$ does not form a clique.*

This formalizes the concept of a "missing edge" between a new node and any existing node in the clique candidate for $k$-hypergraphs. See Corollary 4.2, p. 72, for the analogous property on regular graphs.

We define an auxiliary procedure on graphs that we will call *clique candidate generation.* Starting from an edge in a $k$-hypergraph, this procedure will generate one candidate set for a clique. For any given edge $e_0$, **generateCliqueCandidate** will return the union of the set of nodes in $e_0$ and the set of nodes that are connected by a $k$-hyperedge each with $k-1$ nodes in $e_0$ (this is a concept analogous to the concept of *connected* in regular graphs). The intuition behind this procedure is that it either finds the (only) clique that contains edge $e_0$ or the union of all nodes of all cliques that contain edge $e_0$. The algorithm in Fig. 4.7 gives pseudo code for this procedure.

**Theorem 4.3** *Let $G_k = (V, E)$ be a $k$-hypergraph and $e_0 \in E$ an edge in $G_k$. Construct a set $S$ as the output of procedure* ***generateCliqueCandidate(**$G_k, e_0$**)**, *Fig. 4.7. Denote by $C_{max}(G_k)$ the set of maximal cliques in graph $G_k$. If $S$ is a clique, then*

$$C_{max}(G_k^*) \supseteq C_{max}(G_k) \backslash \{S\},$$

*with $G_k^* = (V, E \backslash \{e_0\})$. That is, the set of maximal cliques in the graph obtained by removing edge $e$ from $G_k$ is a superset of the set of maximal cliques in $G_k$ without clique $S$. In other words, if $S$ is a clique, then it is*

```
function generateCliqueCandidate(Graph $G_k(V, E)$, Edge $e_0$)
$S \leftarrow$ nodes of $e_0$    //an edge is a set of nodes, thus $S$ is a set
forall ($v_1 \in V \backslash S$)
    $f \leftarrow$ true    //a flag
    forall ($v_2 \in e_0$)
        if ($(\{v_1\} \cup (e_0 \backslash \{v_2\})) \notin E$)
            $f \leftarrow$ false
    if ($f$)
        $S \leftarrow S \cup \{v_1\}$
return $S$
```

Figure 4.7: Clique Candidate Generation for $k$-Hypergraphs

*the* only *maximal clique in* $G_k$ *that contains edge* $e_0$*. The removal of edge* $e_0$ *may introduce smaller cliques in graph* $G^*_K$ *that are sub-cliques of* $S$ *and have to be removed by algorithm HYPERCLIQUE.*

**Proof:** The theorem states that edge $e_0$ is an edge *only* in clique $S$ and not in any other clique from $C_{max}(G_k)$. Consider procedure **generateClique-Candidate**$(G_k, e_0)$. It finds all nodes "connected" to all nodes in edge $e_0$. No node that is not connected to all nodes in $e_0$ can be a member of a clique that contains edge $e_0$. Therefore, the set $S$ this procedure returns must be a superset of the union of nodes in *all* cliques of which edge $e_0$ is a member. Now if $S$ is a clique, which is a premise in our theorem, and $S$ contains all nodes that could possibly be members of cliques containing $e_0$, $S$ must also be the *only* clique that contains edge $e_0$. □

With this theorem, we can state that any set of nodes returned by procedure **generateCliqueCandidate(**$G_k, e$**)** that induces a complete graph is a maximal clique. Furthermore, retaining only the edges $E^*$ for a graph

$G_k^*$ (see Fig. 4.6) will produce a correct result for the clique finding problem in graph $G_k$, since all cliques that contain edges *not* in $E^*$ have already been found.

Procedure **generateCliqueCandidate**$(G_k, e)$ is used to check which edges of $G_k$ can be safely removed (since they only contribute to one clique each, which is returned by the corresponding call to this function). For our algorithm HYPERCLIQUE (Fig. 4.6), removing edges in this way is the major strategy to achieve better performance. With the brute-force procedure and the main heuristic (Thm. 4.3) in place, we can now explain the algorithm itself.

The algorithm HYPERCLIQUE starts by generating one clique candidate for each edge in the input graph $G_k$ and collects all those clique candidates that pass a test for the clique property in a set *result* (**forall**-loop in Fig. 4.6). The following example shows this procedure.

**Example 4.5** *In order to illustrate this procedure, consider Figure 4.8, which is broken down into four stages. Stage 1 shows a* 3*-hypergraph* $G_k$ *(this is the same graph as in Fig. 4.4). Procedure* ***generateCliqueCandidate*** *is now called on all* $e \in E$*. Stage 2 shows edge* $e = \{1, 2, 3\}$ *being selected. Initially, set* $S$ *in function* ***generateCliqueCandidate*** *(Fig. 4.7) contains nodes 1,2, and 3, which are the nodes in* $e$*. Node 4 is added to* $S$ *since edges* $\{1, 2, 4\}$*,* $\{2, 3, 4\}$*, and* $\{1, 3, 4\}$ *all exist. Node 5 is not added, since, for example, there is no edge* $\{1, 3, 5\}$*. That is, function* ***generateCliqueCandidate*** *returns the set* $S = \{1, 2, 3, 4\}$ *to algorithm HYPERCLIQUE (Fig. 4.6). This set is now tested for the clique property, and tests*

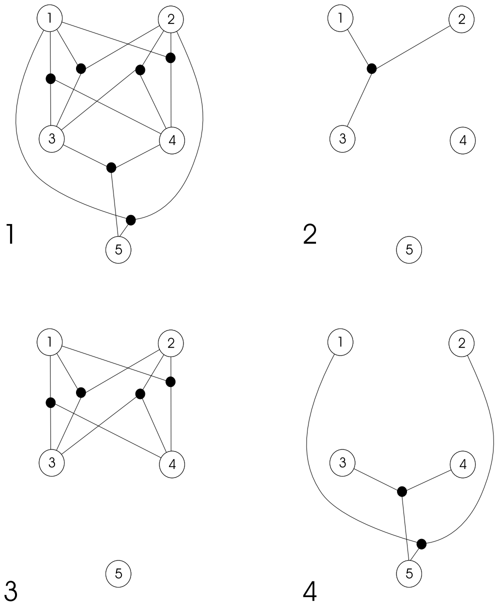


Figure 4.8: Phase 1 of Algorithm HYPERCLIQUES: Growing hypercliques

*true (Stage 3). That means that S is a maximal clique (by the definition of function* ***generateCliqueCandidates****) and furthermore that edge* $\{1, 2, 3\}$ *cannot be part of another clique that includes nodes other than 1,2,3, and 4 (by Thm. 4.3). Thus, S is included in set result. This process is now repeated for all other edges in E. Clique* $\{1, 2, 3, 4\}$ *is found three more times (and thus not added to the final result). Subsequent calls to procedure* ***generateCliqueCandidate*** *on edges* $\{1, 2, 5\}$ *and* $\{3, 4, 5\}$*, respectively (Stage 4), find cliques* $\{1, 2, 5\}$ *and* $\{3, 4, 5\}$*, respectively. Since cliques have been found for* all *edges in E, and each edge is member of exactly one clique (by Thm. 4.3), all cliques must have been found. Note that in algorithm* ***HYPERCLIQUE****, set* $E^*$ *is now empty and the graph* $G_k$ *is reduced to an empty graph, terminating the algorithm.*

However, this procedure might fail. It may occur that the function **generateCliqueCandidate**$(G_k, e)$ returns a set that is *not* a clique. In this case, edge $e$ must be a member of two or more maximal cliques that cannot be merged into one clique. Therefore, a new graph $G_k = (V, E^*)$ is constructed that contains all edges for which no cliques were generated before. In most cases, repeating our algorithm on $G_k$, we will find more cliques, and eventually reduce our graph to an empty set. We give an example of that case.

**Example 4.6** *For an example, refer to Fig. 4.9. The graph shown is a* 2-*hypergraph, as it is difficult to visualize the problem for higher-rank graphs. Cliques in that graph are* $\{0, 1, 2\}$, $\{1, 2, 4\}$, $\{1, 3, 4\}$, *and* $\{2, 4, 5\}$. *The three edges shown by bold lines (*$\{1, 2\}$, $\{2, 4\}$, $\{1, 4\}$*) are all members of more than one maximal clique, while all other edges are members of only one clique each. Function* ***generateCliqueCandidate****, used on any of those three edges, would return set of nodes that does not have a clique property, such that we would miss clique* $\{1, 2, 4\}$ *in the solution. We do find each of the other cliques when considering one of the thinly drawn edges in the figure. Thus, we have to reduce this graph to include only those edges* $E^*$ *for which no clique was found in the first iteration (those are exactly the edges drawn in bold in the figure). Then, we call our algorithm again with only this subgraph* $G_k = (V, E^*)$. *Again, by Thm. 4.3, we will not miss any cliques when using this procedure. Some sub-cliques of previously discovered cliques may be found, which have to be removed from the solution (however not in this example). In this example, we would find all cliques in this graph within* *two loops of algorithm* HYPERCLIQUE*: cliques* $\{0, 1, 2\}$*,* $\{1, 3, 4\}$*,* $\{2, 4, 5\}$ *in the first run and clique* $\{1, 2, 4\}$ *in the second.*

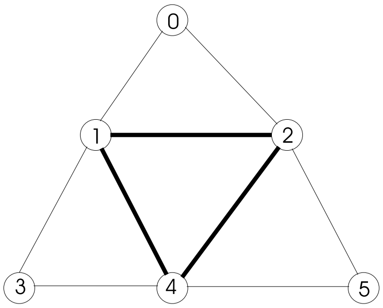


Figure 4.9: Phase 2 of Algorithm HYPERCLIQUES: Reducible Graphs

In certain cases, this procedure might fail, namely when set $E^* = E$ (we will call such a graph *irreducible*). Algorithm HYPERCLIQUE in Fig. 4.6 uses a flag *reducible* to keep track of that property. That situation occurs when *each edge* in a graph is part of more than one maximal clique, i.e., when function **generateCliqueCandidate** never returned a clique for any of the edges in $E$. For example, the 9-node MOON-MOSER-Graph (see Sec. 4.2.4) has this property. Each edge in that graph is a member of three cliques (Fig. 4.10).

In this case, we apply the following strategy. From the irreducible graph $G = (V, E)$, we take a random edge $e \in E$ and call procedure **generateCliqueCandidate** on $e$. This will return a set $S$ that is not a clique, but the nodes of all cliques that $e$ is a member of will be elements of $S$. Now, we *split* $G$ into two graphs: graph $G_e$ which is the subgraph of $G$ induced by $S$, and graph $G_{\bar{e}}$, which is identical to $G$, except that it does not contain

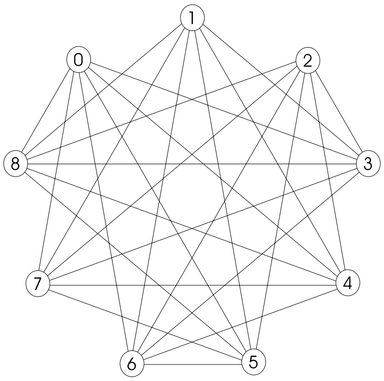


Figure 4.10: An Irreducible Graph.

edge $e$.[3] We then call algorithm HYPERCLIQUES (Fig. 4.6) on $G_e$ and $G_{\bar{e}}$, separately. The intuition behind this strategy is that removing edge $e$ from the graph $G$ will make the graph reducible. In order to still generate all cliques, we need to consider graph $G_e$ separately. This heuristic leads to an increase in complexity that can be severe for dense graphs. However, since the input graphs in our IND-finding problem are quite sparse (as we find in our Experiments in Chapter 6), the algorithm's performance is sufficient for our purpose.

**Example 4.7** *Refer to Fig. 4.11 for an example of the split procedure. The graph from Fig. 4.10 is being split at edge* $\{0, 8\}$ *into two graphs* $G_e$ *and* $G_{\bar{e}}$*. Note that graph* $G_{\bar{e}}$ *in Fig. 4.11 is identical to the graph in Fig. 4.10, except that edge* $\{0, 8\}$ *is missing.*

In very large hypergraphs, a final problem may occur that we call *strong irreducibility.* A graph $G$ may not be reducible such that it can be split,

[3] This is not a partition as some edges exist in both $G_e$ and $G_{\bar{e}}$.

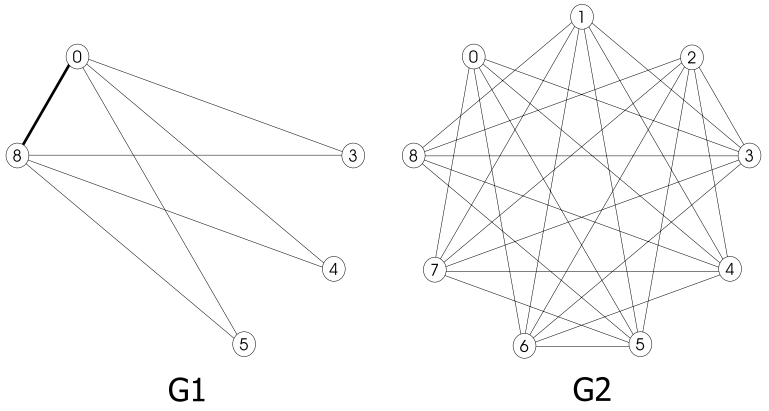


Figure 4.11: Splitting an Irreducible Graph.

*and* the graph $G_e$ split off from $G$ may actually be identical to $G$. In this case, infinite recursion occurs. We detect such cases and repeat the split with a different edge from $G$ (inner do-while-loop in Fig. 4.6). This loop will eventually terminate since the graph cannot be strongly irreducible for every one of its edges. If this were the case, the graph itself would have to correspond to a hyperclique, and the inner do-while-loop would not have been reached.

It is clear that for most graphs, there will be a significant performance benefit in applying algorithm **HYPERCLIQUE** rather than the simple brute-force algorithm, since a smaller number of cliques (compared to the brute-force algorithm) actually has to be considered. In most cases, the size of the graph is quickly reduced by removing edges, which reduces the problem size. Our experiments (Chapter 6) have shown that algorithm HYPERCLIQUE finishes in a short time in all cases, even when algorithm FIND_HYPERCLIQUES_BRUTE_FORCE takes a prohibitively long time or fails to finish altogether.

### 4.2.6 An Algorithm to Find Inclusion Dependencies

After describing algorithms to find cliques, we now present the algorithm $\mathsf{FIND}_2$ which uses those clique-finding algorithms (Bron/Kerbosch and HYPERCLIQUE) to find inclusion dependencies. $\mathsf{FIND}_2$ takes as input two relations $R$ and $S$, with $k_R$ and $k_S$ attributes, respectively and returns a generating set of inclusion dependencies between attributes from $R$ and $S$. The schema of both relations must be known, and it must be possible to perform a test for validity for any inclusion dependency between two given sets of attributes from $R$ and $S$. $R$ and $S$ do not necessarily have to have the same number of attributes.

#### Overview

We first establish the relationship between hypercliques and the generating set of IND that we are trying to find. Intuitively, Thm. 4.4 shows that a clique-finding algorithm is a sensible approach to finding maximal INDs between relations. More specifically, we show that the INDs generated through a clique-finding algorithm are a (relatively small) superset of the generating set $\mathcal{G}(\Sigma)$ and are thus a starting point for a complete and fast solution of the IND-finding problem.

**Theorem 4.4** *Consider the IND finding problem between relations* $R[A]$ *and* $S[B]$ *with solution* $\mathcal{G}(\Sigma)$ *(i.e., generating set of valid INDs). Let* $V$ *be the set of unary valid INDs between* $R$ *and* $S$*. Let* $E_k$*, with* $1 < k \leq$

$\min(|A|, |B|)$, *be the set of $k$-ary valid INDs between $R$ and $S$. Recall that the elements of $E_k$ can then be seen as edges in a $k$-Hypergraph $G_k(V, E_k)$, by Thm. 4.1.*

*Now consider the set $C$ of all maximal cliques in the $k$-hypergraph $G_k$, obtained by the above clique-finding algorithms (*Bron/Kerbosch *and HYPERCLIQUE). The following properties hold for any $c \in C$:*

1. *If the IND $\sigma_c$ corresponding to $c$ is valid, it is part of the generating set $\mathcal{G}(\Sigma)$ of INDs between $R$ and $S$.*
2. *If $\sigma_c$ is* not *valid, some of its subsets are part of $\mathcal{G}(\Sigma)$.*
3. *Furthermore,* all *elements $\sigma \in \mathcal{G}(\Sigma)$ are subsets of or equal to some $\sigma_c$ as above.*

**Proof:** By Thm. 4.1, a valid IND implies a $k$-hyperclique in a $k$-hypergraph $G_k$ constructed for this IND-finding problem. Also, a correct clique finding algorithm returns a set of maximal cliques. Property (1) must hold since if there was an IND larger than $\sigma_c$, a clique corresponding to that larger IND would have been found. Property (2) is true since we assumed valid unary and $k$-ary INDs to make up graph $G_k$. If $\sigma_c$ is *not* valid but its unary and binary sub-INDs are, then some sub-INDs of $\sigma_c$ must be part of $\mathcal{G}(\Sigma)$. Property (3) holds since *any* IND implies some (not necessarily maximal) complete subgraph of $G_k$, and by the definition of a clique, all such complete subgraphs are subsumed by the set of cliques found in $G_k$. □

Due to the high complexity of the problem, enumerating INDs is not feasible, as we have seen in Sec. 4.2.1. By way of the graph-mapping of the IND-finding problem (Sec. 4.2.3) and using clique-finding algorithms, we avoid generating most INDs and rather concentrate only on those that are generated by a clique-finding approach. Since the clique property is only necessary but not sufficient for an IND to hold, we also need to deal with invalid INDs thus found, in a way that will be explained in this section.

**Example 4.8** *We introduce a running example that we will refer to throughout this section. Fig. 4.12 shows two relations R and S. The following maximal INDs hold between R and S:* $R[A, B, C, D, E] \subseteq S[A, B, C, D, E]$, $R[D, F, G] \subseteq S[D, F, G]$, $R[E, F] \subseteq S[E, F]$, $R[E, G] \subseteq S[E, G]$. *Our algorithm has to discover this generating set of INDs for R and S.*

R

| $A$ | $B$ | $C$ | $D$ | $E$ | $F$ | $G$ |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 |

S

| $A$ | $B$ | $C$ | $D$ | $E$ | $F$ | $G$ |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 0 | 0 |
| 0 | 0 | 0 | 4 | 0 | 6 | 7 |
| 0 | 0 | 0 | 0 | 5 | 6 | 0 |
| 0 | 0 | 0 | 0 | 5 | 0 | 7 |
| 0 | 0 | 0 | 0 | 0 | 6 | 7 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 |

Figure 4.12: The Running Example for the IND-Finding Problem.

Algorithm $\mathsf{FIND}_2$ in pseudo code is given in Fig. 4.13. It proceeds in three steps: finding unary and binary INDs, finding higher-arity IND candidates from those, finding further IND candidates. In the following sections, we discuss these three phases of the $\mathsf{FIND}_2$ algorithm and explain them, as well as the overall algorithm, in detail.

```
ALGORITHM $\mathsf{FIND}_2$

INPUT:
Relations $R, S$ with $k_R, k_S$ attributes ($k_S \leq k_R$)

OUTPUT:
Set $\mathit{result}$, containing a generating set of INDs for $R$ and $S$

Set $V \leftarrow$ generateValidUnaryINDs$(R, S)$
Set $E \leftarrow$ generateValidBinaryINDs$(R, S, V)$
Graph $G_2 \leftarrow (V, E)$
Set $I \leftarrow$ generateCliquesAndCheckAsINDs$(G_2)$
Set $\mathit{result} \leftarrow \{c \in I \wedge |c| = 1\}$
for $m \leftarrow 3 \ldots k_S$
    KHypergraph $G_m \leftarrow (V, \emptyset)$
    Set $C_{\mathit{tmp}} \leftarrow \emptyset$
    forall $(c \in I)$
        if $(c$ is valid $\wedge |c| \geq (m-1))$ $\mathit{result} \leftarrow \mathit{result} \cup c$
        if $(c$ is invalid $\wedge |c| \geq m)$ $C_{\mathit{tmp}} \leftarrow C_{\mathit{tmp}} \cup c$
    $E_m \leftarrow$ generateKAryINDsFromCliques$(m, C_{\mathit{tmp}})$
    if $(E_m = \emptyset)$ return $\mathit{result}$
    $\mathit{result} \leftarrow \mathit{result} \cup$ generateSubINDs$(m, E_m, \mathit{result})$
    Graph $G_m \leftarrow (V,$ validINDs$(E_m))$
    $I \leftarrow$ generateCliquesAndCheckAsINDs$(G_m)$
return $\mathit{result}$
```

Figure 4.13: Algorithm $\mathsf{FIND}_2$ for Finding a Generating Set of INDs Between Two Relations

### Finding Unary and Binary INDs

In Fig. 4.13, this phase corresponds to lines number 01 and 02. As explained in Sec. 4.2.1, there is a total of $I_1(k_R, k_S) = k_R \cdot k_S$ unary (not necessarily valid) INDs between $R$ and $S$. To establish the set of nodes upon which all subsequent $k$-hypergraphs will be built, all those INDs have to be validated. This requires a generation of all $k_R \cdot k_S$ unary INDs and their subsequent testing. Methods for efficient testing are covered in Section 5.1. Bell and Brockhausen [BB95b] present a slightly improved algorithm for the discovery of unary INDs which relies mainly on finding maximal and minimal attributes values for numeric domains, as well as on the availability of foreign key constraints in a database. We consider those assumptions to be quite strong given the performance benefits of the algorithm, such that we will not discuss this approach here further. If foreign keys are available, a number of unary database queries can be saved by their approach. Much more widely applicable heuristics that do no rely on outside constraints but solely on the values in an attribute (as well as its name and domain) are discussed in Section 5.1.

**Example 4.9** *In our running example (Fig. 4.12), the valid unary INDs found are $R[A] \subseteq S[A]$, $R[B] \subseteq S[B]$, $R[C] \subseteq S[C]$, $R[D] \subseteq S[D]$, $R[E] \subseteq S[E]$, $R[F] \subseteq S[F]$, and $R[G] \subseteq S[G]$.*

Once the valid unary INDs are established, we create from them a set of nodes $V$ which will be used for all subsequent graph mappings (by simply creating a node for each valid unary IND). We can now proceed to generate candidates for binary INDs. In principle, there are $\binom{k_R}{2} \cdot \binom{k_S}{2} \cdot 2$ such INDs (by Eqn. 4.1), or when multiplied out :

$$I_2(k_R, k_S) = \frac{k_R(k_R - 1) \cdot k_S(k_S - 1)}{2} \text{ INDs.}$$

While this number can be quite large for relations with dozens or hundreds of attributes, many of those INDs do not have to be generated or tested since they would imply invalid unary INDs. In general, the number of binary INDs is bounded by a function quadratic in the number of unary INDs.

Also, by our definition of INDs (Def. 3.1), we do not allow duplicate attribute names on either side of an IND. Thus, when several unary INDs hold for the same attribute (e.g., $R[a, d] \subseteq S[b, e]$ and $R[a, d] \subseteq S[c, e]$ could be valid at the same time), not all combinations of their attributes have to be tested as binary INDs.

After finding unary and binary INDs, we can then construct graph $G_2$ as defined in Section 4.2.3, p. 69 (Line 03 in $\mathsf{FIND}_2$).

**Example 4.10** *The binary INDs valid between relations R and S in Fig. 4.12 are:*

$R[A, B] \subseteq S[A, B]$ $R[A, C] \subseteq S[A, C]$ $R[A, D] \subseteq S[A, D]$
$R[A, E] \subseteq S[A, E]$ $R[B, C] \subseteq S[B, C]$ $R[B, D] \subseteq S[B, D]$
$R[B, E] \subseteq S[B, E]$ $R[C, D] \subseteq S[C, D]$ $R[C, E] \subseteq S[C, E]$
$R[D, E] \subseteq S[D, E]$ $R[D, F] \subseteq S[D, F]$ $R[D, G] \subseteq S[D, G]$
$R[E, F] \subseteq S[E, F]$ $R[E, G] \subseteq S[E, G]$ $R[F, G] \subseteq S[F, G]$

*Graph $G_2$ is depicted in Fig. 4.14. The nodes represent unary INDs (labeled with index numbers) and the edges represent the binary INDs listed above.*

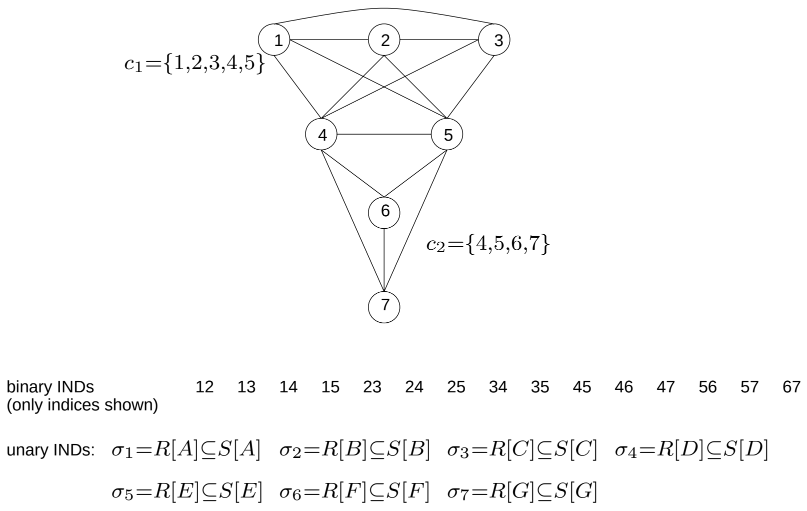

1
2
3
4
5
6
7
$c_1$={1,2,3,4,5}
$c_2$={4,5,6,7}
binary INDs (only indices shown) 12 13 14 15 23 24 25 34 35 45 46 47 56 57 67
unary INDs: $\sigma_1$=$R[A] \subseteq S[A]$ $\sigma_2$=$R[B] \subseteq S[B]$ $\sigma_3$=$R[C] \subseteq S[C]$ $\sigma_4$=$R[D] \subseteq S[D]$
$\sigma_5$=$R[E] \subseteq S[E]$ $\sigma_6$=$R[F] \subseteq S[F]$ $\sigma_7$=$R[G] \subseteq S[G]$

Figure 4.14: A Graph $G_2$ constructed by Algorithm FIND$_2$.

### Generating Candidates for Higher-Arity INDs

We now run the Bron/Kerbosch algorithm on the graph $G_2$ (Line 04 in FIND$_2$). The algorithm generates a set $I$ (see Fig. 4.13), containing all maximal cliques in $G_2$. We denote the number of nodes in each clique $c \in I$ by $k_c$. Any clique with $k_c = 1$ (single unconnected nodes in $G_2$) will be part of the solution and is thus added to the variable *result* of FIND$_2$ (Line 05).

To understand the remainder of the algorithm, consider a collection of $k$-hypergraphs with $k = 3, \ldots, \max(k_c)$ (i.e., $G_3, \ldots, G_{\max(k_c)}$) obtained by storing each $k$-subset of each clique $c \in I$ (with $k \leq k_c$) as an edge in $G_k$. Together with the mapping from INDs to $k$-hypergraphs from Section 4.2.3, we observe the following property.

**Theorem 4.5** *Let $E_k$ be a $k$-set of nodes from $V$. If the edge $e_k$ whose set of nodes is exactly $E_k$ does* not *exist in $G_k$, the IND $\sigma_k$ corresponding to $e_k$*

*cannot be valid.*

**Proof:** We prove by contradiction. Assume a valid $k$-ary INDs that does *not* correspond to an edge in the $k$-hypergraph $G_k$. The validity of $\sigma_k$ implies the (existence and) validity of $\binom{k}{2}$ binary INDs that are edges in $G_2$. By Theorem 4.1 (with $m = 2$) those binary INDs form a clique $c$.

However, the Bron/Kerbosch clique finding algorithm guarantees that *all* cliques in $G_2$ are found. Furthermore, any subset of a clique is also a clique. Thus, we would have found a subset $c_k$ of some clique $c$ that corresponds to $\sigma_k$ and stored it as an edge in the appropriate $k$-hypergraph. This contradicts the above assumption. □

However, the converse of the above proposition is not true. The existence of an edge in $G_k$ does *not* imply the existence of a $k$-ary IND in the database. To see why, refer to Fig. 4.14. In that example, a clique-finding algorithm would find a clique $c_2 = \{4, 5, 6, 7\}$, but the corresponding IND $R[D, E, F, G] \subseteq S[D, E, F, G]$ is not valid in the database from Fig. 4.12. Therefore generating graph $G_2$ is not enough. Rather, we must also test INDs that are implied by cliques that do *not* correspond to valid INDs (see Property (2) in Thm. 4.4).

Making use of the property described in Theorem 4.5, the maximum number of INDs to consider for testing is determined by the number of cliques generated by Bron/Kerbosch. This is much smaller than the total number of INDs. For a real-world problem, the number of cliques is likely to be manageable (see Chapter 6). Hence, this results in significant performance benefits of the $\mathsf{FIND}_2$ algorithm over the naïve algorithm. We also use heuristics to limit the number of cliques when too many (spurious) cliques do occur, as explained later in Sec. 5.1.

### Determining Higher-Arity INDs

So far, only unary and binary members of the generating set $\mathcal{G}(\Sigma)$ have been generated. We now show how to write an algorithm using the idea of Thm. 4.4. A first step in finding higher arity-INDs is the checking of all IND candidates (cliques) generated by the previous steps. INDs that are valid will become part of the solution. However, if an IND generated by the clique-finding algorithm tests invalid, nothing can be said about the validity of any other INDs. IND-finding then proceeds as shown in Fig. 4.15.

Main loop of algorithm FIND$_2$

1. Set $m = 3$. Let $I$ be the set of cliques in $G_2$ created earlier.
2. Test all cliques in the set $I$ as INDs for their validity and collect *invalid* INDs in a set $C_{tmp}$. All *valid* INDs are added to the set of solutions. Now, for each $\sigma \in C_{tmp}$, add to a set $E_m$ all $m$-ary INDs implied by $\sigma$. (Lines 09–12 in FIND$_2$, Fig. 4.13)
3. Test all INDs $\sigma \in C_{tmp}$ against the database as they are generated and store the valid ones as edges in an $m$-Hypergraph $G_m$ (Line 15).
4. Run algorithm HYPERCLIQUE (Fig. 4.6) to find $m$-hypercliques in $G_m$ (Line 16) and store the result in the set $I$ to be used in the next iteration of the loop. If no cliques with more than $m$ nodes (i.e., nontrivial cliques) are found, FIND$_2$ will terminate in the next loop and return the solution.
5. Increment $m$ and repeat from step 2.

Figure 4.15: Generating Higher-Arity Member of the Generating Set $\mathcal{G}(\Sigma)$ in Algorithm FIND$_2$

After iterating through the main loop and terminating, the solution set contains all valid INDs that have been found as cliques or hypercliques of lower-arity INDs. However, some maximal INDs may have been missed in this process. Consider set $I$ as generated by algorithm $\mathsf{FIND}_2$ (Line 16). For a given $m$, the set $I$ (Figure 4.13) contains cliques that were found from a $(m-1)$-hypergraph. All those cliques correspond to INDs in the database, but not all those INDs are necessarily valid. In Fig. 4.16 this situation is depicted for $m = 3$, where clique $c_2 = \{4, 5, 6, 7\}$ is not valid.

If an IND $\sigma$ thus found is invalid, it is still possible that some of its implied sub-INDs are valid. We will find most of those in subsequent hypergraphs (for higher $m$). However, some sub-INDs of $\sigma$ with arity $m-1$ could be maximal, i.e., part of the solution. The $\mathsf{FIND}_2$ algorithm as described so far would not find those. In Fig. 4.16, three implied 3-ary INDs of clique $c_2$ (labeled 456, 457, and 567) are invalid. Therefore, some of their implied 2-ary INDs (which we know to be valid) are maximal and thus part of the generating set. Therefore, we need to generate all maximal sub-INDs $\sigma_m$ of the invalid INDs in $I$.

For the given $m$, we thus form all implied $m$-ary INDs of all invalid INDs in $I$ (i.e., all invalid INDs implied by any clique) and test each one for validity. We observe the following properties of any such implied IND $\sigma_m$:

1. If $\sigma_m$ itself is valid, it is a (not necessarily proper) subset of an IND not yet discovered (and vice-versa it cannot be a subset of any undiscovered IND if it is invalid)

2. If $\sigma_m$ is invalid, some or all of its $(m-1)$-ary sub-INDs may be maximal

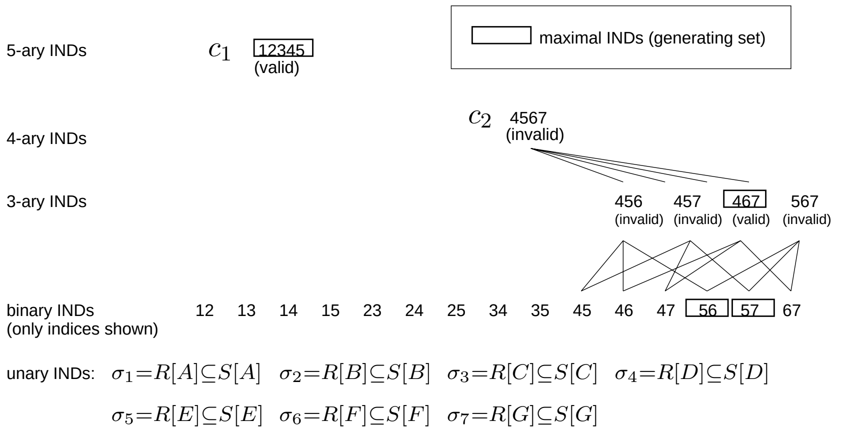

maximal INDs (generating set)
5-ary INDs
$c_1$
12345
(valid)
4-ary INDs
$c_2$
4567
(invalid)
3-ary INDs
456 (invalid)
457 (invalid)
467 (valid)
567 (invalid)
binary INDs
(only indices shown)
12 13 14 15 23 24 25 34 35 45 46 47 56 57 67
unary INDs: $\sigma_1{=}R[A]{\subseteq}S[A]$ $\sigma_2{=}R[B]{\subseteq}S[B]$ $\sigma_3{=}R[C]{\subseteq}S[C]$ $\sigma_4{=}R[D]{\subseteq}S[D]$
$\sigma_5{=}R[E]{\subseteq}S[E]$ $\sigma_6{=}R[F]{\subseteq}S[F]$ $\sigma_7{=}R[G]{\subseteq}S[G]$

Figure 4.16: Invalid INDs Generated by the Clique-Finding Algorithm in Fig. 4.15.

valid INDs and thus part of the generating set of INDs.

We further observe the following properties of any $(m-1)$-ary sub-IND $\sigma_{m-1}$ generated from any $\sigma_m$:

1. $\sigma_{m-1}$ is valid in the database (since it is a hyperedge in the $(m-1)$-Hypergraph $G_{m-1}$ which has previously been formed)

2. if $\sigma_m$ is valid, $\sigma_{m-1}$ is not maximal

3. if $\sigma_m$ is invalid and $\sigma_{m-1}$ is *not* a subset of any other valid $\sigma$ of arity $m$ as well as not a subset of any larger valid IND already part of the result, $\sigma_{m-1}$ must be a maximal IND.

4. as $m$ will grow during subsequent executions of the outer for-loop in $\mathsf{FIND}_2$, and no $k$-hypergraph can contain cliques of sizes $2\dots(k-1)$ (p. 67), no further cliques of size $m-1$, and thus INDs of size $m-1$, can

be discovered after this step. That means that the procedure described here will finally find all maximal INDs of arity $m$ and smaller.

These properties suggest an algorithm to generate all maximal INDs that have not been discovered by clique-finding, which is stated in the next section (function **generateSubINDs**). We now give a concrete example of this procedure for our running example problem.

**Example 4.11** *In Fig. 4.16, algorithm* $\mathsf{FIND}_2$ *found a clique* $\{4, 5, 6, 7\}$ *in graph* $G_2$*. For* $k = 3$*, this clique is part of set* $I$ *(see Fig. 4.13). As the IND implied by this clique (*$R[D, E, F, G] \subseteq S[D, E, F, G]$*) is invalid in the database (Fig. 4.12), the four implied* 3*-ary INDs have to be checked. One of them (*$R[D, F, G] \subseteq S[D, F, G]$*, labeled* 467*) is valid in the database, the other three are not. Thus, INDs* $R[D, F] \subseteq S[D, F]$*,* $R[D, G] \subseteq S[D, G]$*, and* $R[F, G] \subseteq S[F, G]$ *(labeled* 46*,* 47*, and* 67*, respectively) are not maximal, while the remaining binary INDs implied by the invalid INDs may be valid (those are* $R[D, E] \subseteq S[D, E]$*,* $R[E, F] \subseteq S[E, F]$*,* $R[E, G] \subseteq S[E, G]$*, labeled* 45*,* 56*,* 57*, respectively).* $R[D, E] \subseteq S[D, E]$ *(labeled* 45*) is implied by the valid* $R[A, B, C, D, E] \subseteq S[A, B, C, D, E]$ *(labeled* 12345*), which is already in the solution. Thus it is not maximal. The remaining two INDs (*$R[E, F] \subseteq S[E, F]$*,* $R[E, G] \subseteq S[E, G]$*, labeled* 56*,* 57*) are maximal and are thus included in the solution. Note that there is no need to check any binary IND for validity anymore, since all binary INDs that are considered in this step are edges in* $G_2$ *(and must thus be valid).*

With these observations, we can state that algorithm $\mathsf{FIND}_2$ generates all maximal INDs that hold between two given relations. We will give a proof in Section 4.2.7.

### Subroutines of $\mathsf{FIND}_2$

The following subroutines are used in algorithm $\mathsf{FIND}_2$.

**generateValidUnaryINDs(R,S).** This function generates all $k_R \cdot k_S$ unary INDs and checks them against the database. It returns the set $V$ of the INDs valid in the database, which are the nodes for all subsequent graphs and hypergraphs.

**generateValidBinaryINDs(R,S,V).** This function generates all those INDs whose implied binary INDs are elements of $E$, as explained in Sec. 4.2.6. The function then checks all those INDs against the database and returns the set $E$ of all those valid binary INDs.

**generateCliquesAndCheckAsINDs($G_k$).** This function accepts a $k$-hypergraph. It returns a set of all hypercliques in $G_k$, together with a Boolean value for each element in the set in the following manner:

For $k > 2$, this function calls algorithm HYPERCLIQUE on $G_k$. If $k = 2$, the Bron/Kerbosch-algorithm is run. The function then tests each generated (hyper)clique's implied IND against the database. It returns a set of *all those INDs with more than k nodes* (i.e., at most one IND for each clique discovered), regardless of their validity, but *marks* each IND as valid or invalid according to its state in the database.

**generateKAryINDsFromCliques(k,E).** This function accepts a number $k$ and a set of INDs $E$. Input set $E$ is assumed to be composed of invalid INDs and to correspond to cliques found by a clique finding algorithm on a $(k-1)$-Hypergraph. This function now generates all $k$-ary INDs implied by each IND in $E$, tests each one of them, and returns the union of those INDs for all elements of $E$. Similarly to function **generateCliquesAndCheckAsINDs()**, both valid and invalid INDs are returned, and each IND is marked as valid or invalid according to its state in the database.

**generateSubINDs(k,E,*result*).** This function accepts a number $k$, a set of (valid or invalid) INDs $E$, and set *result*, which is the set of INDs already included in the solution earlier (lines 05 and 10 in Fig. 4.13). The function now generates all INDs $\sigma$ that satisfy the following three conditions (derived from the conditions on p. 96):

1. $\sigma$ is a $k$-subset of an *invalid* IND from $E$
2. $\sigma$ is *not* a subset of *any valid* IND from $E$
3. $\sigma$ is *not* a subset of any IND already in *result*.

The function returns the set of all $\sigma$ that meet the above conditions. Note that subset testing can be done very efficiently when sets are stored as bit fields (by simple bit operations).

**validINDs(E).** Given a set of INDs marked as valid or invalid, this function simply returns the valid INDs from $E$.

Also note that the set *result* of solutions is built through set-union operations (Lines 10 and 14 in Fig. 4.13). That will prevent INDs that have been found multiple times from being added to the solution more than once.

The first four functions in the above list (**generateValidUnaryINDs**, **generateValidBinaryINDs**, **generateCliquesAndCheckAsINDs**, and **generateKAryINDsFromCliques** all test INDs against the database. In the $\mathsf{FIND}_2$ algorithm as described in this section, we accomplish this by database queries, as explained in Sec. 5.1.1. Faster and more comprehensive methods of checking INDs are explored in Sec. 5.2.2.

**Example 4.12** *We demonstrate algorithm* $\mathsf{FIND}_2$ *by way of our running example.* $\mathsf{FIND}_2$ *first finds unary and binary inclusion dependencies (Fig. 4.17). For INDs of arity higher than 1, we only show the indices of unary INDs that they imply. The algorithm then proceeds to find cliques from the discovered valid binary (and unary) INDs, from which we have formed a graph (Thm. 4.1). The cliques found in this graph are* $c_1 = (1, 2, 3, 4, 5)$ *and* $c_2 = (4, 5, 6, 7)$*. A test against the database of those two cliques yields that the IND induced by* $c_1$ *is valid while the IND induced by* $c_2$ *is false (invalid). Now, algorithm* $\mathsf{FIND}_2$ *proceeds to generate four 3-ary INDs from the invalid IND induced by* $c_2$*. Those INDs (*$(4, 5, 6)$*,* $(4, 5, 7)$*,* $(4, 6, 7)$*, and* $(5, 6, 7)$*) are tested against the database and only* $(4, 6, 7)$ *is found valid. That implies that any 2-ary IND induced by* $(4, 6, 7)$ *is not maximal while any 2-ary IND implied by the other three INDs may be maximal. In fact, two of them are (*$(5, 6)$ *and* $(5, 7)$*), while* $(4, 5)$ *is subsumed by* $(1, 2, 3, 4, 5)$ *and is therefore not maximal. The only edge in the newly formed* 3*-Hypergraph* $G_3$ *is now* $(4, 6, 7)$*, such that the only* clique *in that graph is also* $(4, 6, 7)$*, which is then the last remaining maximal IND. This implies that the generating set for all INDs between* $R$ *and* $S$ *is* $\{(1, 2, 3, 4, 5),\ (4, 6, 7),\ (5, 6),\ (5, 7)\}$.

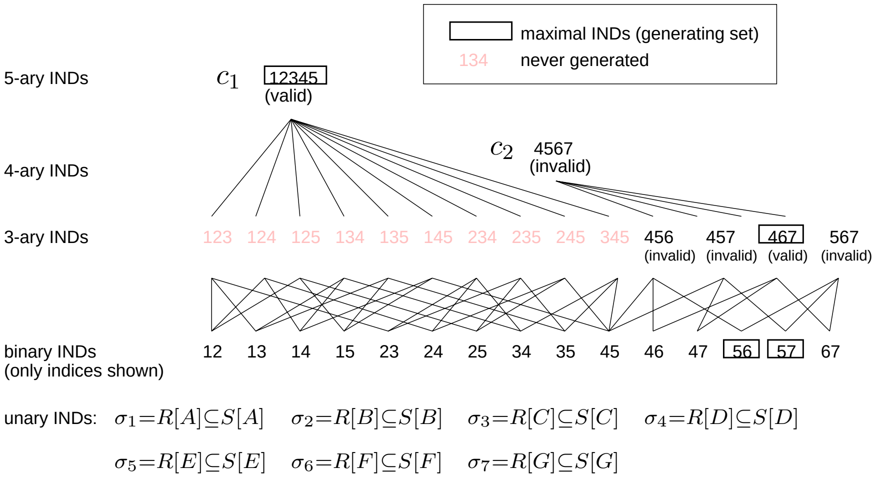

maximal INDs (generating set)
134 never generated
5-ary INDs
$c_1$
12345
(valid)
4-ary INDs
$c_2$
4567
(invalid)
3-ary INDs
123 124 125 134 135 145 234 235 245 345
456 (invalid)
457 (invalid)
467 (valid)
567 (invalid)
binary INDs
(only indices shown)
12 13 14 15 23 24 25 34 35 45 46 47 56 57 67
unary INDs: $\sigma_1{=}R[A]{\subseteq}S[A]$ $\sigma_2{=}R[B]{\subseteq}S[B]$ $\sigma_3{=}R[C]{\subseteq}S[C]$ $\sigma_4{=}R[D]{\subseteq}S[D]$
$\sigma_5{=}R[E]{\subseteq}S[E]$ $\sigma_6{=}R[F]{\subseteq}S[F]$ $\sigma_7{=}R[G]{\subseteq}S[G]$

Figure 4.17: An Example for the Complete Algorithm FIND$_2$.

## 4.2.7 Correctness of Algorithm FIND$_2$

### Correctness of Functions Called by FIND$_2$

To show the correctness of algorithm FIND$_2$, we first assume that the functions called by it are correct. All functions called, except for *generateSubINDs* whose soundness we have motivated beginning on page 95, are fairly straightforward and easy to prove so that their correctness will not be shown here.

The clique-generating function (*generateCliquesAndCheckAsINDs*) uses the Bron/Kerbosch-algorithm for $k = 2$ and a similar hyperclique algorithm for $k > 2$. Both algorithms are also assumed to run correctly (i.e., find all maximal cliques in the given graph or $k$-hypergraph).

### Correctness of the Overall Algorithm

It remains to show that the algorithm $\mathsf{FIND}_2$ finds the generating set (Def. 3.6) of INDs between two relations $R$ and $S$. We have to show that the result set is complete, i.e., that all INDs between $R$ and $S$ can in fact be derived from the INDs in the set *result* and that the set *result* is minimal with respect to this property.

First, we show that all INDs are in fact found. Consider the graph $G_2$ whose nodes are valid unary INDs and whose edges are valid binary INDs between the relations in question. By Thm. 4.4, all elements $\sigma \in \mathcal{G}(\Sigma)$ of the generating set, i.e., all solutions to the IND-finding problem, are either elements or subsets of the set of cliques found by the clique algorithm for $k = 2$. We consider the INDs that have *not* been found directly as cliques in $G_2$ (property (2) in Thm. 4.4). Let us denote this set of not-yet-discovered INDs by $\Sigma_2 \subseteq \mathcal{G}(\Sigma)$. Now there are two cases:

1. INDs $\sigma_2 \in \Sigma_2$ with $|\sigma_2| = 2$ are found by the function *generateSubINDs*, line (14) in algorithm $\mathsf{FIND}_2$ (Fig. 4.13)

2. INDs $\sigma_2 \in \Sigma_2$ with $|\sigma_2| > 2$ are not found by function *generateSubINDs*. Rather, 3-ary INDs are created as subsets of those cliques in $G_2$ that correspond to invalid INDs, and used as edges in a 3-hypergraph $G_3$. This set of 3-ary INDs is necessarily equal to or a superset of the set of 3-ary sub-INDs of elements of $\Sigma_2$ (i.e., the set of as-yet undiscovered INDs, by Thm. 4.4).

We now show that all INDs are found. For $3 \leq i \leq \min(|R|, |S|)$, the clique-finding algorithm is called for $G_i$. By Thm. 4.4, some new INDs may be found from (hyper)cliques. However, function *generateSubINDs* does generate *all $i$-ary INDs*, analogous to the case of $i = 2$ described above. For each $i$, a new set $\Sigma_i$ with the not-yet-discovered INDs is formed as above. Since all INDs must have arity lower than or equal to $\min(|R|, |S|)$, and all (missing) $i$-ary INDs are found at step $i$, $\Sigma_i$ must be empty after calling function *generateSubINDs* for $i = \min(|R|, |S|)$. That means, all INDs must have been found.

It remains to show minimality. There are three places in the algorithm $\mathsf{FIND}_2$ where discovered INDs are added to the result, in lines 5, 10, and 14 (Fig. 4.13). First, the result is initialized with the single-node cliques. Those correspond to nodes in the initial graph $G_2$ which have degree 0, such that their implied INDs cannot be part of any higher-arity IND (i.e., they are maximal). In lines 5 and 10, we add INDs that have been generated by a clique algorithm, which must be maximal (since clique algorithms find *maximal* cliques). INDs added in line 14 are maximal as shown in the previous section. Since only *maximal* INDs are added to the result, the result set is a generating set, as long as all INDs are included. We have proved that all INDs are included. Thus the set is indeed a generating set, and therefore minimal, by the definition of generating sets in Sec. 3.2.2. **q.e.d.**

### 4.2.8 Complexity of Algorithm $\mathsf{FIND}_2$

In this section, we will consider the complexity of algorithm $\mathsf{FIND}_2$. We will show the complexity of each function called by it, and the complexity of the complete algorithm. Throughout the section, we will assume that

IND-testing can be done in constant time. We will also assume the variable names from algorithm $\mathsf{FIND}_2$, Fig. 4.13.

### Generating Unary and Binary INDs

The runtime of function *generateValidUnaryINDs* is in $O(k_R \cdot k_S)$ and returns a set $V$. Function *generateValidBinaryINDs* runs in $O(|V|^2)$ time, which, in the worst case is bounded by $(k_R \cdot k_S)^2$. However, for real-world data, the value is normally much smaller than that number (see our experiments in Chapter 6). The maximum is only reached for the special case of two identical relations with only attributes with identical values.

### Generating Cliques in Graphs

The function *generateCliquesAndCheckAsINDs* finds cliques in $k$-hypergraphs. As mentioned earlier, the decision problem of whether there is a clique of a certain minimum size in a graph is NP-complete [GJ79]. Thus, finding *all* cliques in a graph is NP-hard. The maximal number of cliques in a graph is in $O(3^{\frac{n}{3}})$ with $n$ the number of nodes [MM65]. The problem of finding all hypercliques in a $k$-hypergraph is a generalization of the clique-finding problem and thus also NP-hard. However, the problem is tractable in practice, since we expect a small number of cliques only, and graphs with few cliques can be searched for cliques efficiently.

We now show that the Bron/Kerbosch-algorithm (Appendix A) is capable of finding the set of cliques $C$ in a graph $G = (V, E)$ in a time polynomial in the *number of cliques*.[4] The algorithm is a backtracking algorithm

[4] This complexity results is suggested but not proven in [BK73].

that keeps a stack of candidate nodes for a clique. It essentially traverses a tree $T$ of clique candidates, cutting off a branch when the addition of another node to the contents of the current stack cannot lead to a clique. Assuming that no clique is discovered more than once (for which the algorithm takes special precautions, see below), each path in this tree $T$ corresponds to a clique, while the tree has exactly as many leaves as there are cliques in the graph $G$. The number of recursive calls in Bron/Kerbosch is equal to the number of edges $e$ in this clique tree $T$. There is an obvious (loose) upper bound for $e$, related to the size of the largest clique $c_{\max} = \max(|c| : c \in C)$, namely $e < c_{\max} \cdot |C|$. The number of different large cliques is normally very small, such that the actual number for $e$ is much smaller than that. $c_{\max}$ is bounded by the number of vertices in G, which is a (small) constant that does not depend on $|C|$. That is, the number of recursive calls is in $O(|V| \cdot |C|)$.

For each recursive call, Bron/Kerbosch executes two steps:

1. finding the degrees of all nodes in a graph $G'(V', E)$ (constructed as the graph induced by $G(V, E)$ in some set $V' \subset V$), which is done in $O(|V'|^2)$ time and

2. finding a set of nodes in $G'(V', E)$ connected to a special node (the *fixpoint*, variable *fixp* in Appendix A) which is done in $O(|V'|)$ time.

No graph $G'$ can have more nodes than $G$. Therefore, combining those two steps, the runtime of Bron/Kerbosch has an upper bound of $O(|V|^3 \cdot |C|)$ where $|V|$ is the number of nodes in $G$ and $|C|$ the number of cliques.

A more careful analysis shows that the algorithm sometimes starts finding the same clique several times, but since it keeps a history of nodes that have already served as starting points for cliques, it never re-discovers more than one node of an already discovered clique. This will not slow down the algorithm by more than a factor of $c_{\max} \cdot |C|$ (our upper bound for the number of edges in search tree $T$), which gives the whole algorithm a complexity of $O(|V|^4 \cdot |C|^2)$ as a very loose upper bound.

To complete the analysis, it remains to evaluate the number of cliques generated in this step. In our experiments, we have found that depending on the number of attributes in the underlying tables, the number of cliques at any step in the algorithm does not exceed a few thousand. This number poses no problem for our algorithm. For problems in which the number of cliques becomes too large, we apply heuristics explained in Sec. 5.1.

### Generating Cliques in Hypergraphs

Algorithm HYPERCLIQUE (Sec. 4.2.5) generates all cliques in a $k$-hypergraph $G_k$. HYPERCLIQUE is more difficult to analyze than the BRON/KERBOSCH-algorithm. The worst case complexity is clearly exponential, which occurs in the case when the entire graph is strongly irreducible, such that brute-force clique finding has to be used. The complexity of the brute-force algorithm is in the order of the number of possible subsets of the nodes in the input graph, which is $O(2^n)$ in the number of nodes $n$.

In the best case, the algorithm will find one clique for every edge in the graph during the first loop through all edges and then terminate, such that its complexity is determined by the number of cliques and the time it takes to generate clique candidates and test them. An analysis of the average runtime complexity did not seem feasible, but that we will discuss some complexity issues of certain steps in the algorithm.

Two important functions that algorithm HYPERCLIQUE needs to support are generating clique candidates and checking a set for the clique property.

In order to generate the clique candidate for an edge $e$ (Function *generateCliqueCandidate*$(G_k, e)$ in algorithm $\mathsf{FIND}_2$), the algorithm needs to cycle through (almost) all nodes in $G_k$, and for each node $\nu$ determine whether for all possible sets $S$ of $k-1$ nodes not containing $\nu$, there is an edge in $e$ that connects $S$ with $\nu$. This involves searching for the correct edge once for each node in $e$, i.e., $k$ times. Since the edges can be stored in a sorted list, searching is done in logarithmic time. Thus, the complexity of function *generateCliqueCandidate*$(G_k, e)$ is in $O(|V| \cdot k \cdot \log(|E|))$.

Testing clique candidates in $k$-hypergraphs is accomplished by checking node degrees, using Thm. 4.2. In $k$-hypergraphs, the finding of node degrees is more complex than in regular graphs. Instead of simply iterating through one row of the adjacency matrix, one now has to go through all edges (which are sets) and test for set membership of a node in that edge. For graphs with few nodes (less than the word size of the processor), the time for testing set membership is constant, such that a node degree can be determined in $O(|E|)$ time. Thus, testing whether a graph is reducible involves generating the clique candidate and testing it for each edge in $G_k$, such that reducibility testing runs in $O(|E| \cdot (|V| \cdot k \cdot |E| + |E|)) = O(|V| \cdot k \cdot |E|^2)$ time. For completely reducible hypergraphs (i.e., hypergraphs in which no hyperedge is a member of more than one maximal clique), the algorithm HYPERCLIQUE terminates after one iteration through all edges, such that it also runs in $O(|V| \cdot k \cdot |E|^2)$, which is small for the sparse hypergraphs likely to occur in our environment. However, the runtime is mainly dependent on the number of edges in $G_k$, which is an important observation for our heuristics in Sec. 5.1.

As irreducible graphs are being split and treated separately, a complexity blowup can occur. To solve this problem, we will introduce in Chapter 5 some heuristics that restrict the size of the graphs involved.

### Generating IND Subsets

Generating subsets (functions *generateKAryINDsFromClique* and *generateSubINDs* in algorithm $\mathsf{FIND}_2$, Fig. 4.13) is simple to analyze. The runtime of the generating algorithm is bounded by the number of subsets to generate, i.e., by $\binom{|c|}{k}$ for a clique $c$ in the case of *generateKAryINDsFromClique* and $|E_k| \cdot \binom{k}{k-1}$ in the case of *generateSubINDs*. The first number can become very large when $k$ is not very small and is not close to $|c|$.

Therefore, we need to look at likely values that $k$ can assume. In algorithm $\mathsf{FIND}_2$, $k$ will grow as long as there are new non-empty $k$-hypergraphs $G_k$ constructed. A hypergraph $G_k$ will be non-empty if and only if there were cliques in the $(k-1)$-Hypergraph $G_{k-1}$ that corresponded to invalid INDs. Let us assume the IND-finding problem between relations $R$ and $S$. Assume that algorithm $\mathsf{FIND}_2$ has found a clique $c$ with $n$ nodes in a $k$-hypergraph $G_k$ and that $c$ corresponds to an invalid IND $\sigma_c = R[A_c] \subseteq S[B_c]$. This implies that there is a set $I_k$ of valid $k$-ary INDs with $|I_k| = \binom{n}{k}$, namely the ones that were used to construct $c$ to begin with.

For those conditions to hold, each tuple $t_R \in R$ must either exist in $S$ or the following condition must hold for $t_R$ (and there must be at least one such tuple):

$$\forall t \in S[B_c] : t_R[A_c] \neq t \wedge$$
$$\forall (\sigma_i = \underbrace{R[A_i] \subseteq S[B_i]}_{\text{a sub-IND of } \sigma_c}) \in I_k : \exists t \in S : t_R[A_i] = t[B_i]$$

In words, while tuple $t_R[A_c]$ itself must not exist in $S[B_c]$, $S[B_c]$ must contain at least one tuple for each possible projection of $t_R[A_c]$ on any $k$ attributes. Note that this puts a constraint on the values of $r(k+1)$ tuples in $S$, for some number $1 \leq r \leq |R|$. While it is simple to construct an example where these conditions hold (Fig. 4.3), it seems unlikely in practice that such conditions occur accidentally or by design. Thus, we do not expect the value of $k$ to grow very large in our algorithm. In our experiments, $k$ did not grow beyond 5 or 6.

### Complexity of the Entire Algorithm

A useful general bound for the complexity of algorithm $\mathsf{FIND}_2$ is difficult to give. Furthermore, giving *tight* bounds for the general case seems impossible due to the complexity of the problem. Therefore, and due to the fact that a subproblem of $\mathsf{FIND}_2$ is NP-hard, we will not give a complexity bound but rather refer to our experiments. In cases when our original algorithm as shown in Fig. 4.13 runs too slowly, we restrict the search space by applying heuristics (Sec. 5.1). Those heuristics allow us to adjust the runtime of

$\mathsf{FIND}_2$ to a user's needs, while still being able to control the errors that the algorithm produces.

# Chapter 5

# Heuristic Strategies to Find Inclusion Dependencies

## 5.1 Comparing two Attribute Sets

In Chapter 4, we discussed algorithm $\mathsf{FIND}_2$ which finds the generating set of inclusion dependencies between two given relational tables. That algorithm relies on being able to test any given inclusion dependency against the database. In this chapter, we will discuss how such inclusion dependencies are tested, what the issues and problems are and how the testing can be optimized.

Thus, we consider the following problem:

> *Consider a relation $R$, a list of attributes $A_1, \ldots, A_k$ from $R$, a relation $S$, and a list of attributes $B_1, \ldots, B_m$ from $S$. Decide whether the IND $\sigma = R[A_1, \ldots, A_k] \subseteq S[B_1, \ldots, B_m]$ is valid in*

*the database, i.e., whether the projected extents of relations $R$ and $S$ stand in the set relationship asserted by $\sigma$.*

An answer to this question can simply be found by formulating a database query, as suggested in the following section, but such a procedure will not necessarily be optimal for algorithm $\mathsf{FIND}_2$, as we show in this chapter. Rather, we introduce a more sophisticated way of testing inclusion dependencies, by making use of heuristics that extract additional information from attribute sets beyond simple inclusion.

### 5.1.1 Simple IND-Testing

In the previous section, we treated the testing of INDs as an atomic operation whose runtime is constant. We will now look more closely at the complexity of this operation.

There are at least two different ways to determine the validity of an IND using SQL: the direct computation via a MINUS-query, or comparing the results of two COUNT-queries. For example given an IND $R[A_1, A_2] \subseteq S[B_1, B_2]$, those SQL queries would be formed as shown in Figs. 5.1 and 5.2.

```
select count(*)
from ( select A1,A2
       from R
       minus
       select B1,B2
       from S)
```

Figure 5.1: Determining the Validity of an IND by a MINUS-Query

In the first case of MINUS-queries (Fig. 5.1), the IND is valid if and only

```
select count(distinct R.A1, R.A2)
from   R,S
where  R.A1=S.B1 and R.A2=S.B2;

select count(distinct A1,A2)
       from R;
```

Figure 5.2: Determining the Validity of an IND by Two COUNT-Queries

if the number returned by the query is 0. In the second case of count-queries (Fig. 5.2), the IND is valid iff the numbers returned by the two queries are equal [1]. It is also possible to use an outer join, which is slightly faster than a count of a natural join, but still significantly slower than computing the set difference (in our implementation, Chapter 6).

Computing the set difference between two projections involves sorting both relations by the attributes in the projection and then making an additional run through all tuples to compare them. Assuming two equal-size relations $R$ and $S$ with $n$ tuples, the complexity of the MINUS-query (Fig. 5.1) is in $O(n \cdot \log(n) + n) = O(n \cdot \log(n))$ in main memory or in $O(n)$ in terms of I/O-cost for relational databases, given sufficient buffer sizes (since external merge-sort can be used to sort data [EN94]). Since comparing two tuples may not be trivial, the runtime may depend on the number of attributes in the IND and their data types, depending on the implementation of the database system used.

The second method (Fig. 5.2) involves joining two tables and thus runs in $O(n^2)$; clearly slower than the first query. This query becomes extremely slow when two large relations with many duplicates in the join attribute

[1] This method was suggested by Bell and Brockhausen [BB95b].

are considered, since the join result before duplicate removal can have up to $|R||S|$ tuples.

If relations $R$ and $S$ do not reside in the same database, those queries are not applicable. In that case, the queries have to be broken down for the two databases and the results have to be combined, for example by using a local database. For example, such an algorithm could proceed as in Fig. 5.3 (in JDBC-like syntax).

```
function CHECK(Relation $R$, AttribList $A$, Relation $S$, AttribList $B$)
    RelationExtent $E_R, E_S$
    $E_R \leftarrow$ executeQuery(select distinct A from DB1.R)
    insert $E_R$ into localDB
    $E_S \leftarrow$ executeQuery(select distinct B from DB2.S)
    insert $E_S$ into localDB
     $c \leftarrow$ executeQuery( select * from localDB.E_R
                         minus select * from localDB.E_S)
    if ($c = 0$) return true    //IND is valid
    else return false
```

Figure 5.3: A Simple Algorithm to Check for IND Validity

A possibly improved method for IND-testing involves first computing $E_R$, and then iterating through tuples from $E_R$ and searching these tuples in $E_S$. At best, one could iterate through tuples in $E_R$ and find the IND in question to be invalid as soon as a tuple in $E_R$ is not found in $E_S$. However, the worst-case complexity is not better than in function CHECK when using this method. In any case, a large amount of tuples has to be transferred through the network, such that the complete computation of the validity of INDs should be avoided when possible.

As IND testing is expensive, we propose in the remainder of this chapter several heuristics for an improvement on this part of IND discovery. We focus on two strategies: reducing the runtime for a single IND check, and reducing the number of IND checks algorithm $\mathsf{FIND}_2$ has to perform against the database.

## 5.2 Finding Inclusion Dependencies Using Heuristics

Algorithm $\mathsf{FIND}_2$ as described so far finds the complete and correct solution to the IND-finding problem for two given data sets. However, since several of the algorithms involved have high complexity, $\mathsf{FIND}_2$ does not finish for large problems. The problem size for which complete solutions can be found depends on the number of distinct values in each attribute of the data sets tested, as we explain in Section 5.2.1. However, we have established a number of heuristics that help to reduce the sizes of the graphs involved at the expense of the completeness and/or correctness of the solution. The large increase in the size of problems that are tractable using heuristics justifies the errors introduced, as we argue in the section below.

### 5.2.1 Accidental INDs

Algorithm $\mathsf{FIND}_2$ (Section 4.2) works by generating and checking low-arity INDs and then deducing higher-arity INDs based on the results. However, this process is very sensitive to the number of low-arity INDs in the problem, as all graph algorithms used are NP-hard. We now define a notion of “specious” or “accidental” INDs that are valid in the database but do not contribute to a fast solution found by $\mathsf{FIND}_2$.

**Definition 5.1 (Accidental IND)** *Consider the IND-finding problem between two given relations $R$ and $S$. Let $\Sigma$ be the set of valid INDs between $R$ and $S$ and $\mathcal{G}(\Sigma)$ be the generating set of $\Sigma$. Furthermore, let $\sigma \in \Sigma$ be a valid IND. Let $\sigma_L \in \mathcal{G}(\Sigma)$ be the largest element of $\mathcal{G}(\Sigma)$ that implies $\sigma$, and $\sigma_{max} \in \mathcal{G}(\Sigma)$ be the largest IND in $\mathcal{G}(\Sigma)$. With $p = \frac{|\sigma_L|}{|\sigma_{max}|}$, $\sigma$ is then called* $p$-confident*. An IND that is $p$-confident with a $p$ lower than some threshold is called* accidental*.*

The intuition behind this definition is that for a pair of relations in which the largest IND has arity $k$, most INDs of arity less than $k \cdot p$ (i.e., of *relatively* small arity) are not very “interesting”, as we are primarily concerned with finding large overlaps between relations. Since smaller INDs that are implied *only* by such “uninteresting” INDs do not contribute to finding large INDs, there are deemed noise for our algorithm, at the risk of losing some of the completeness of the result.

For large IND-finding problems, we consider INDs accidental if their confidence is lower than about 0.1, an empirically found value. An IND that is not accidental does most likely express some kind of semantic relationship between two tables, whereas an accidental IND is likely to be valid by statistical coincidence, as explained in the remainder of this section.

Often, accidental INDs occur when information is encoded as small integers; a procedure often used in real-world database design, as the following example shows.

**Example 5.1** *Consider Fig. 5.4 for an example. Using our example database from Fig. 3.1, we assume that the string-valued attributes* Genre *and* Director *have been encoded using numeric values.*

**GenreIndex**

| **Genre** | Sci-Fi | Drama | Satire | TV-Feature |
|---|---|---|---|---|
| **Index** | 1 | 2 | 3 | 4 |

**DirectorIndex**

| **Director** | D.Lynch | J. Negulesco | J.Cameron | S.Kubrick |
|---|---|---|---|---|
| **Index** | 1 | 2 | 3 | 4 |

**Movies**

| **Title** | **Genre** | **Director** |
|---|---|---|
| Dune | 1 | 1 |
| Aliens | 2 | 3 |
| Under My Skin | 2 | 2 |
| Titanic | 2 | 3 |
| Titanic | 4 | 3 |
| Dr. Strangelove | 3 | 4 |

**MyMovies**

| **Title** | **Genre** | **Director** |
|---|---|---|
| Dune | 1 | 1 |
| Titanic | 2 | 3 |
| Dr. Strangelove | 3 | 4 |

$$\textbf{MyMovies}[\textbf{Genre}] \subseteq \textbf{Movies}[\textbf{Director}]$$
$$\textbf{MyMovies}[\textbf{Director}] \subseteq \textbf{Movies}[\textbf{Genre}]$$
$$\textbf{MyMovies}[\textbf{Genre,Director}] \subseteq \textbf{Movies}[\textbf{Director,Genre}]$$

Figure 5.4: Accidental INDs Introduced by Encoding Data

*In this example, the actual information content is the same as in the database in Fig. 3.1 (i.e., the two databases are semantically equivalent [MIR94]). The four INDs shown there also hold in this database. However, in the encoded version of the database, three additional low-arity*

*INDs are introduced (shown in bold font in Fig. 5.4). The third IND (**MyMovies**[**Genre,Director**] ⊆ **Movies**[**Director,Genre**]) is valid and maximal and is thus part of the generating set of INDs between **MyMovies** and **Movies**. However, in some sense, this IND is intuitively "wrong". Note that it is not implied by any other valid IND otherwise found (in Fig. 3.1), meaning that it will be discarded (in this case) from the search space when INDs of arity higher than 2 are generated. Therefore, we will not need this IND for a correct solution of INDs with arities larger than 2. Also, having to consider these additional unary and binary INDs enlarges the graphs used by our algorithm and increases its runtime. The confidence of all three new INDs as defined in Def. 5.1 is* 0.67*. This value is relatively high as this small example relation has very few attributes. For larger relations, we frequently encounter INDs with very low confidence.*

Our definition of "accidental INDs" states that even though an IND may be valid in the database, it may not actually be "good" data, that is, it may not help in the discovery of larger INDs. In addition, we hold the view that such INDs are also less useful to a user as part of the generating set of INDs. To see this, consider two relations that have a large overlap of $k$ attributes, expressing some semantic relationship between the relations. An additional overlap of, for example, $k/10$ attributes that is disjunct from the large overlap will most likely not carry any useful information.

In order to assess the probability for such accidental INDs to occur we look at a statistical model. Assume two relations $R$ and $S$ and the problem of assessing whether a valid IND $\sigma = R[A] \subseteq S[B]$ actually expresses a semantic relationship between $R$ and $S$ or whether $\sigma$ holds “by accident”. Furthermore assume that both attributes have equivalent domains (such as `int` or `char`) with $k$ elements. Let $A$ have $r$ distinct values and $B$ have $n$ distinct values. Now consider the statistical probability that $r$ particular objects are included in a random sample of $n$ objects from $k$ different objects. This probability $P(n, r, k)$ in some sense gives the likelihood that two attributes form an “accidental IND”. $P(n, r, k)$ can be computed by the following formula.

**Theorem 5.1** *Consider a set $R = \{e_1, \ldots, e_r\}$ of $r$ elements from a universe $K$ of $k$ distinct elements. The probability that a random sample (obtained by sampling with replacement) of $n$ elements from $K$ contains set $R$ is*

$$P(n, r, k) = 1 - \frac{\sum_{i=1}^{r} (-1)^{i+1} \cdot \binom{r}{i} \cdot (k-i)^n}{k^n} \tag{5.1}$$

**Proof:** There are $k^n$ different samples of size $n$ from $k$ distinct elements (sampling with replacement). We compute how many of those do *not* contain $R$. A sample that does not contain $R$ is a sample in which at least one element from $R$ is missing. Let us denote by $A_e$ the set of all samples that are missing element $e$. Then, the number of samples that do not contain at least one element from $R$ is $r_0 = |A_{e_1} \cup A_{e_2} \cup \cdots \cup A_{e_r}|$. The size of each $A_e$ is easy to compute, however we need to determine the size of the union of all those sets. For this purposes, we use the *inclusion-exclusion rule* of combinatorics.

This rule is a generalization of the rule $|A \cup B| = |A| + |B| - |A \cap B|$ (“addition law”, [Ric95]) and states that:

$$\left|\bigcup_{i=1\ldots n} A_i\right| = \sum_{i=1}^{n}\left[(-1)^{i+1}\cdot \sum_{J:\ |J|=i}\left|\bigcap_{j\in J} A_j\right|\right] \tag{5.2}$$

In our problem, the number of distinct samples in a set $\bigcap_{j\in J} A_j$ (i.e., the number of samples that contain none of the elements in $J$), is equal to the number of distinct samples of $n$ elements from $(k-|J|)$ elements, i.e., $(k-|J|)^n$. With our definition of set $R$ from above, the number of such subsets $J \subset R$ of a certain size $i$ is $\binom{r}{i}$. Therefore, the number of samples that do not contain at least one element from $R$ is:

$$\begin{aligned} r_0 = |A_{e_1} \cup A_{e_2} \cup \cdots \cup A_{e_r}| &= \sum_{i=1}^{r}\left[(-1)^{i+1}\cdot \sum_{J\subset R:\ |J|=i}\left|\bigcap_{j\in J} A_j\right|\right] \\ &= \sum_{i=1}^{r}(-1)^{i+1}\cdot\binom{r}{i}\cdot(k-i)^n \end{aligned}$$

By dividing this number $r_0$ of samples that do not contain $R$ by the number $k^n$ of possible distinct samples, we get the probability $P' = \frac{r_0}{k^n}$ that a sample does *not* contain $R$, such that $P(n,r,k) = 1 - P'$, **q.e.d.** □

For the purpose of determining the probability of "accidental INDs", we can use this formula in the following sense: Recall from above that in the valid $R[A] \subseteq S[B]$, $A$ has $r$ distinct values and $B$ has $n$ distinct values. One can argue that since the values in $A$ are a subset of the values in $B$, the values in both attributes are from a domain $B$ (then, $A \subseteq B$ and $B \subseteq B$). We are interested in the "chance" that attribute $A$ just "happens" to be included in attribute $B$ as a statistical property. This "chance" can be assessed by the probability that a superset of $A$ is obtained by a random sampling of $n$ values from $k = |D|$ values by setting $n = k$ in Equation 5.1. For the case $r = 1$, this probability, which we will call $p_1$, tends towards $p_1 = 1 - \frac{1}{e} = 0.632$ (i.e., the probability that $n$ samples from $n$ distinct elements contain *one* particular element is $p_1$ for $n = k$ and $n \rightarrow \infty$). What this probability states is that "if attribute $A$ is from the domain given by $B$ and has only one value, the chance that the IND $R[A] \subseteq S[B]$ does not express any semantic relation between $A$ and $B$ is $p_1$". In Table 5.1 we have listed the minimum value for $r$ for which $p_r$ remains lower than 0.05, for different $n = k$. That is, we have computed how many distinct values an attribute $A$ must contain, given an attribute $B$ with $n$ values, in order to have 95% confidence that a *valid* IND $R[A] \subseteq S[B]$ does express a semantic relationship between $A$ and $B$. Note that for $n \leq 4$, $p$ never drops below 5% for any $r$, which means that in this model, a valid IND $R[A] \subseteq S[B]$ for a $B$ with 4 or less distinct values can never be seen as an indicator for a semantic relationship between $A$ and $B$.

| $n = k$ | 95% confidence for $r \geq$ |
|---:|---:|
| 1000 | 7 |
| 100 | 7 |
| 10 | 6 |
| 8 | 6 |
| 6 | 5 |
| 5 | 5 |
| 4 | N/A |

Table 5.1: Minimum Number of Distinct Values to Avoid Accidental INDs

**Example 5.2** *It is possible to compute confidence levels for the number of distinct values for other cases. For example, if expert knowledge is available stating that the domains of both $A$ and $B$ are integer percentages (i.e., integers between* 0 *and* 100*,* 101 *distinct values), one obtains that a valid IND* $R[A] \subseteq S[B]$ *with* 50 *distinct tuples in* $B$ *and* 3 *distinct tuples in* $A$ *has a* $P(50, 3, 101) = 0.058$ *probability to be "accidental", i.e., expresses a semantic relationship between* $A$ *and* $B$ *with* 94% *confidence.*

From Table 5.1 it can be concluded that inclusion dependencies where the included attribute has less than 6 or 7 distinct values have a high chance of being valid by statistical coincidence, rather than by semantic relationships between the attributes. This is a very important result that we use to restrict the search space of our algorithm.

### 5.2.2 Heuristics for IND-Validity Testing

We have argued in Sec. 5.1.1 that IND validity testing is expensive, and that simple SQL-queries to test an IND are computationally and I/O-expensive. Also, we have seen that INDs that are valid in a database may actually not express a semantic relationship (with the consequence that they are not likely to be helpful as nodes or edges in our clique-finding algorithms). These observations suggest a number of heuristics that we can use to cut down the search space for algorithm $\mathsf{FIND}_2$.

In the following sections, we propose four heuristics for validity testing of INDs. These heuristics make use of information about attributes that is easier to obtain and compare than the complete list of values in each attribute that would be necessary for IND-validity testing. This additional information can be grouped as follows:

- meta-information (data types, domains, attribute names),
- simple statistical properties (minimum, maximum for continuous domains, number of distinct values for discrete domains), and
- attribute value distributions or histograms.

In the literature, other information is sometimes used. For example, in a framework for attribute equivalence in databases, Larson, Navathe, and ElMasri [LNE89] include additional meta-information, such as semantic integrity constraints available from the database, security constraints (permissions), allowed operations and scale. We hold the view that such meta information is hard to obtain in a general way for heterogeneous databases and restrict ourselves to information that is easily available from an otherwise unknown database by querying the database in SQL.

In the Artificial Intelligence (AI) community [KLN00, MWJ99, DDH00], the semantics of attributes are often used to assess the similarity of attributes. AI solutions can complement and enhance our results if appropriate meta data is available. We will not concentrate on these approaches here as we are looking for a solution that is as automated as possible. Attribute semantics are expert knowledge and often must be entered into a system by human intervention.

There is substantial work on *ontologies*, i.e., usually task-independent knowledge bases used to describe the "world" [Gua95, UG96, GPB99, SBF98], with the goal (in our case) to automatically determine the semantics of an attribute by its name. We are not discussing this work here either, but state that ontologies could be used in our algorithm when available. Available ontologies can be used to simply determine a given IND to be valid or invalid, without having to execute a database query against the source data.

Given the types of information we propose to use in our algorithm, we will now discuss four heuristics.

1. **Domains:** If $A$ and $B$ have incompatible domains, an IND between them cannot be valid.

2. **Names:** If the *names* of $A$ and $B$ are similar, the attributes may be related.

3. **Number of Distinct Values (DV):** If the number of distinct values in $A$ is much smaller than the number of distinct values in $B$, or if both numbers are very small, the attributes may not be related (but rather form an *accidental IND*, as discussed in Section 5.2.1)

4. **Attribute Value Distribution (AVD):** If the distribution of values in $A$ and $B$, respectively, is different, the attributes may not be related.

We use the first heuristic to reduce the number of database queries sent to the source databases, while the latter three heuristics help to distinguish "accidental" INDs from likely semantically meaningful INDs. The Domains heuristic will not affect the quality of our algorithm. The other three heuristics may lead to false negatives (IND that are falsely classified as accidental, i.e., invalid) and thus potentially introduce errors in the solution provided by $\mathsf{FIND}_2$. Solutions to the IND-discovery problem that are found using these heuristics will contain correct but not necessarily maximal INDs.

### 5.2.3 Heuristic: Domains

A very simple way to avoid database queries in order to test INDs is the use of domain information. In the simplest case of relational databases, domain information is available in the form of SQL-data types, under the assumption that reasonable data types are used for each attribute. For example, since an SQL-query-engine cannot compute a MINUS-query between an attribute of type `int` and an attribute of type `char`, a query to test an IND between those attributes does not have to be formed. If more comprehensive domain information is available, it can be used in an appropriate way to avoid forming database queries for incompatible attributes.

This rule does not produce false negative results, in the sense that an IND that is rejected by it is definitely not valid. Of course, if the rule does not reject the IND, the IND may still be invalid.

The runtime of the overall algorithm $\mathsf{FIND}_2$ is improved according to the number of different domains in the database and the distribution of attributes among those domains.

**Example 5.3** *Consider the number $i_1$ of unary INDs between two relations $R$ and $S$ with $k_R = 6$ and $k_S = 4$ attributes, respectively. The upper bound for $i_1$ is $i_1 < k_R \cdot k_S = 24$. Assuming that both $R$ and $S$ have half of their attributes in domain `int` and half of their attributes in domain `char`, the number of unary INDs to be rejected by this heuristic is $3 \cdot 2 + 3 \cdot 2 = 12$,* *such that only* 12 *instead of* 24 *unary INDs have to be tested against the database.*

### 5.2.4 Heuristic: Attribute Names

In addition to attribute *domains*, attribute *names* may also provide information that is useful in determining which INDs to test for validity. Under the assumption that attribute names are "sensible", i.e., that terms are used that describe the contents of the attribute in some way, the attribute names provide valuable meta-information that can help to avoid database queries.

However, attribute names may not always be "meaningful". Attributes may have generic names like `A0,A1,A2` or names that are acronyms or combined from acronyms, such as `A_MON_SAL`. Clearly, it is difficult to automatically recognize a similarity between an attribute name such as `A_MON_SAL` and an attribute name `MonthlySalary`.

Therefore, the attribute name heuristic can only be used in a positive sense, i.e., if two attribute names match, the attributes in question are likely related. Otherwise, nothing can be said about the attributes. Clearly, some additional false positives may be introduced by this heuristic, i.e., attributes may not be related even if the heuristic declares them similar or equal. An additional check for actual validity of the IND question (through a database query) may need to be performed.

#### Computing Attribute Name Matches

A number of name matching algorithms have been proposed in the literature. Most of those algorithms are based on some kind of distance metric [WM97].

Sometimes, ontologies are used, in the form of dictionaries or hierarchical structures of related terms [DDH00]. For our experiments, we used the *Trigram*-Distance metric proposed by Ukkonen [Ukk92].

Ukkonen defines the general case of $q$-gram metric as follows, with $\Sigma$ a finite alphabet, $\Sigma^*$ the set of all strings over $\Sigma$, and $\Sigma^q$ all strings of length $q$ over $\Sigma$, for $q \in \mathbb{N}^+$. A *q-gram* is any string $v = a_1 a_2 \ldots a_q$ in $\Sigma^q$.

**Definition 5.2** *(from [Ukk92]) Let* $x = a_1 a_2 \ldots a_n$ *be a string in* $\Sigma^*$*, and let* $v$ *in* $\Sigma^q$ *be a q-gram. If* $a_i a_{i+1} \ldots a_{i+q-1} = v$ *for some* $i$*, then* $x$ *has an* occurrence *of* $v$*. Let* $G(x)[v]$ *denote the total number of the occurrences of* $v$ *in* $x$*. The* q-gram profile *of* $x$ *is the vector* $G_q(x) = (G(x)[v]), v \in \Sigma^q$.

**Definition 5.3** *(from [Ukk92]) Let* $x, y$ *be strings in* $\Sigma^*$*, and let* $q > 0$ *be an integer. The* q-gram distance *between* $x$ *and* $y$ *is*

$$D_q(x, y) = \sum_{v \in \Sigma^q} |G(x)[v] - G(y)[v]|. \tag{5.3}$$

**Example 5.4** *(from [Ukk92]) Let* $x = 01000$ *and* $y = 001111$ *be strings in the binary alphabet. Their 2-gram profiles are, listed in the lexicographical order of the 2-grams,* $(2, 1, 1, 0)$ *and* $(1, 1, 0, 3)$*, and their 2-gram distance is 5.*

By this definition, a low distance means similar attribute names. The distance is sensitive to the length of the input strings, such that it is useful to normalize the distance by dividing it by the number of $q$-grams across both strings.

For $q = 3$, we have *trigrams* and the *trigram distance metric.* Trigrams are used widely in data mining, which led us to adopt this metric to assess attribute name similarity. As an example of typical trigram distances, consider Table 5.2, with all possible distances between the two sets of strings $\{$*Title, Genre, Director*$\}$ and $\{$*Titel, Genre, Regisseur*$\}$.

| | **"TITEL"** | **"GENRE"** | **"REGISSEUR"** |
|---|---|---|---|
| **"TITLE"** | 4 | 6 | 10 |
| **"GENRE"** | 6 | 0 | 10 |
| **"DIRECTOR"** | 9 | 9 | 13 |

Table 5.2: Trigram-Distances for Two Sets of Strings

It is clear that semantically equivalent names (like **Regisseur** and **Director**) do not necessarily produce low distance values, while low distances generally mean similar strings (attribute names) and thus potentially related attributes.

**Example 5.5** *We give an example of how the trigram metric can be used to make decisions. In our experiments, we used an empirical formula to accept or reject the hypothesis "attribute names are related" for two attribute names. We considered two attribute names $A$ and $B$ related if the trigram-distance $D_3(A, B) < 0.6 \cdot (|A| + |B| - 4)$. Note that $|A| + |B| - 4$ is the sum of the number of (not necessarily distinct) trigrams across both strings.* [2] *For the example in Table 5.2, this method would consider only one attribute name pair to be related, namely (**Genre,Genre**). Thus, two positive matches (**Title,Titel** and **Regisseur,Director**) are not recognized. No false positives*

[2] A string of $n$ characters has $n - 2$ trigrams.

*are found, such that the heuristic can be considered successful (but not "optimal") in this case.*

### 5.2.5 Heuristic: Number of Distinct Values

We have motivated in Section 5.2.1 that even though an IND may be valid in the database, it may not express a semantic relationship between attributes or attribute sets. We stated that the likelihood that two attribute sets in a valid IND are related depends on the probability that the exact values in those attributes are obtained by random sampling. Thus, we showed that attributes with a low number of distinct values have a high probability to be accidental even if their data stands in an inclusion relationship.

Based on this assessment, our heuristic claims that an IND $R[A] \subseteq S[B]$ should *not* be used as a node or edge in a hypergraph in algorithm $\mathsf{FIND}_2$ if the attribute (or attribute set) $A$ has few distinct values (tuples). Based on the statistical analysis in Section 5.2.1, we experimented with different empirical formulae to assess this property.

One simple implementation of this heuristic is to simply discard all inclusion dependencies in which the included attribute has less than $n$ distinct values. In our experiment, we found that $n = 6$ is a good empirical choice. This value is also supported by our theoretical results in Sec. 5.2.1. When this heuristic is used by itself, it may discard valid INDs that are not accidental (especially when the domains of the attributes involved have very few elements). In this case, algorithm $\mathsf{FIND}_2$ will no longer find the set of maximal INDs.

We also experimented with more complex heuristics, taking the numbers of distinct values in both sides of the IND into account. One heuristic was based on the *ratio* between the number of distinct values in both attributes. According to this heuristic, an IND $R[A] \subseteq S[B]$ is considered invalid if the following condition holds ($|A|_D$ is the number of distinct values in attribute set $A$).

$$\frac{|A|_D}{|B|_D} < \begin{cases} 0.5 & \text{if } |A|_D > 40 \\ 0.9 - \dfrac{|A|_D}{100} & \text{otherwise} \end{cases}$$

The intuition behind this function is as follows: By this condition, a valid IND is considered non-accidental if the number of distinct values in the left attribute (set) $|A|_D$ exceeds a certain percentage of the number of values in the right attribute (set) $|B|_D$. This percentage varies between 50% (for $|A|_D > 40$) and just under 90% (for very small $|A|_D$) and is a linear function of $|A|_D$ between those two limits. The cutover point of $|A|_D = 40$ was chosen to achieve a continuous function over the entire range of $|A|_D$.

Our experiments (Chapter 6) showed that the second heuristic works relatively well on its own, while the first (simple) heuristic is superior when used in combination with the attribute-value-distribution (AVD) heuristic described in the next section.

The number-of-distinct-values heuristic can only be used to test for *valid* INDs, i.e., an IND that is already considered invalid (for example because it has been rejected by the domain heuristic) will not be affected. It also may produce false negatives, since it may reject INDs as invalid that are members of the generating set that represents the correct solution for this problem. Since false negatives have adverse effects on the quality of our algorithm's results (Sec. 5.2.8), this heuristic cannot be used on its own, but rather should be used only in combination with the other heuristics that filter out INDs that potentially lead to false negatives in this heuristic.

On the other hand, using this heuristic can have significant effects in reducing the runtime of algorithm $\mathsf{FIND}_2$. Since in tables with many numerically encoded attributes (Sec. 5.2.1) numerous INDs may be "accidentally" valid, this heuristic may reduce the search space by a significant amount. Since the $\mathsf{FIND}_2$ algorithm's complexity is up to exponential in the number of nodes (unary INDs) in the problem, a reduction in the number of unary INDs as achieved by this heuristic may be required to make the algorithm finish at all. The heuristic may lead to some missing unary INDs which may prevent the largest INDs from being found (Chapter 6). However, this will occur only in cases in which the non-heuristic algorithm finds no results at all, consequently this heuristic is important for our algorithm.

### 5.2.6 Heuristic: Attribute Value Distribution (AVD)

The Attribute Value Distribution (AVD) heuristic is the most complex to compute but has very strong predictive power for certain data sets. It is based on the following hypothesis:

> *Two attributes $A$ and $B$ that are semantically related (i.e., form a non-accidental IND) are random subsets of a common domain of values.*

Obviously, this is not always true, for example when one or both of the attributes $A$ and $B$ are selections obtained by applying some predicate (e.g., a SQL-WHERE-condition) on them. However, related attributes *must* be subsets of a common domain, such that the additional assumption that they are *random* (as opposed to selected) subsets seems reasonable at least for some cases. One can test efficiently for this property, by way of statistical hypothesis testing, explained below.

The heuristic then states that if the values of attributes $A$ and $B$, respectively, are not random samples of the same domain (universe) of values, the attributes are not related.

This heuristic can theoretically be used to test for either invalid or valid INDs, but due to the nature of statistical hypothesis testing is only safe to use for testing valid INDs. That is, an IND $\sigma$ that is valid in the database (i.e., may be accidentally valid) can be tested with this heuristic. If the validity hypothesis for $\sigma$ is rejected, $\sigma$ can be considered invalid. If it is not rejected, no new information is gained about $\sigma$. This heuristic can produce false negatives when attributes that are actually semantically related are rejected due to the fact that they are actually not random samples from the same domain. This may occur when the two relations compared have been obtained by applying two different predicates (WHERE-clauses) on a larger data sets, such that the subsets obtained are not random. The statistical hypothesis testing itself, which is probabilistic in nature, may also produce an erratic result. In those cases, a non-accidental IND may not be found and may be missing from the result reported by $\mathsf{FIND}_2$, similarly to the distinct-value heuristic.

Therefore, we use the latter two heuristics in combination. The distinct-value-heuristic is used to pre-screen valid (possibly accidental) INDs, while the AVD-heuristic is used to further support the claim for non-relatedness. If the latter heuristic failed to provide evidence for non-relatedness, the IND is assumed valid. Since we are thus restricting the use of the AVD-heuristic to attributes with few distinct values, the "randomness of subsets" assumption appears more justified. Our experiments support this claim (see Experiment 4 in Chapter 6).

### Performing Statistical Hypothesis Testing for AVD

For the hypothesis test, we use the widely applicable $\chi^2$-Test [Lin76, Ric95], in particular a $\chi^2$-Test for independence. This test is designed to assess the independence of categorical variables. Random sampling data is given as $n$ pairs $(x, y)$. One then constructs a two-dimensional matrix $M$, a so called "contingency table". The rows of $M$ correspond to all distinct values of $x$ and the columns of $M$ correspond to all distinct values of $y$. The element $M_{a,b}$ of $M$ is then the number of pairs $(a, b)$ in the data.

The $\chi^2$-Test then tests under the null hypothesis that the two variables $x$ and $y$ are independent, i.e., that the value of variable $x$ does not influence the value of variable $y$.

For our purpose we perform the following mapping: given an IND $R[A] \subseteq S[B]$ and the question "is this IND valid accidentally or because $A$ and $B$ are semantically related?", we set $x = \{R, S\}$ and, if $<A>$ denotes the set of distinct values in $A$, $y = <A> \cup <B>$. That is, we are testing for the null hypothesis: "the distribution of values in an attribute does not depend on the relation (out of $\{R, S\}$) from which this attribute is taken". In other words, both attributes $A$ and $B$ are **related**, since the value distribution in them is the same (i.e., independent of the relation the attribute is coming from). This method was suggested by Minka [Min01]. Detailed information on the $\chi^2$-Test can be found in [Ric95]. The attribute value distribution in a single attribute can be obtained easily through an SQL-query such as:

```
select a,count(a) from r group by a
```

and is no more complex than testing a single IND. The value distributions of all attributes have to be computed only once for a relation, and since we use the heuristic only for attributes with small numbers of distinct values, only small amounts of data are additionally transferred through the network.

**Example 5.6** *Consider two relations $R$ and $S$ and two attributes $R.A$ and $S.B$. Let $R.A$ have the following values: "0" (5774 times), "1" (40 times), "2" (4 times), "3" (3 times), "4" (once) and $S.B$ the following values: "0" (4648 times), "1" (40 times), "2" (once). Clearly, there is a valid IND $S[B] \subseteq R[A]$. We are now trying to determine whether this IND suggests a semantic relationship between $A$ and $B$ or whether the IND is accidental. A semantic relationship is likely to exist if the value distributions of the attributes are similar, while the IND is likely to be accidental if the value distributions are different.*

*We build a so called contingency table as follows:*

| | *# of tuples* | "0" | "1" | "2" | "3" | "4" |
|---|---|---|---|---|---|---|
| Relation R (Attribute A) | 5822 | 5774 | 40 | 4 | 3 | 1 |
| Relation S (Attribute B) | 4689 | 4648 | 40 | 1 | 0 | 0 |

*The null-hypothesis here is that "the value distribution is equal in both* *attributes". Performing a $\chi^2$-Test on this data gives a $\chi^2$-value of* 5.388*. At confidence-level* 95%*, the percentile for the rejection of the null hypothesis in this case (with 4 degrees of freedom) is* 9.49*. Therefore, in this case we cannot reject the null hypothesis, i.e., we cannot say with significant statistical confidence that the two attributes have different value distributions. In other words, in this case the $\chi^2$-test does not help to decide whether the two attributes form an IND by accident. Thus, no additional information is gained that could help us to answer the question stated above.*

*However, as a variation on this example, if attribute S.B would have only* 10 *occurrences of the value "1", with all other data being equal, we would obtain a $\chi^2$-value of* 16.61 *and could reject the null-hypothesis. We would then obtain the result that $S[B] \subseteq R[A]$ is very likely valid by accident.*

### 5.2.7 Summary

Table 5.3 gives an overview over the four heuristics discussed. For each heuristics, it lists the conditions for application, the error possibilities (false positives and/or negatives) and the impact on the runtime of the algorithm. As mentioned before, several heuristics produce false negatives. False negatives translate into missing edges in the graphs $G_i$ in algorithm $\mathsf{FIND}_2$ and thus lead to incomplete solutions for the algorithm. The effect on the solution returned by algorithm $\mathsf{FIND}_2$ is that maximal INDs may no longer be found, but rather several (large) sub-INDs of those maximal INDs are included in the reported solution. In the next section, we show how this effect can be alleviated.

| | **Domain** | **Attribute Name** | **Distinct Value** | **AVD** |
|---|---|---|---|---|
| Class | Meta-Data | Meta-Data | Data | Data |
| Used to accept INDs | no | yes | no | no |
| False positives | N/A | few | N/A | N/A |
| Used to reject INDs | yes | no | yes | yes |
| False negatives | none | N/A | many | few to many |
| Computational complexity | constant | small | one-time, linear in number of attributes | one-time, linear in number of attributes |
| Impact on run-time by... | avoiding DB queries | avoiding DB queries | reducing size of graphs | reducing size of graphs |
| Overall benefit | small | potentially large | large | large |

Table 5.3: Four Heuristics Used to Improve Algorithm $\mathsf{FIND}_2$

### 5.2.8 False Negatives and the Clique-Merging Heuristic

Consider a complete graph (i.e., a graph with all possible edges) $G = (V, E)$. The set of nodes $V$ in $G$ forms a clique. Now remove a single edge from $E$. Clearly, the clique property does no longer hold, but rather $G$ will now contain at least two distinct maximal cliques. Those cliques are likely to have a substantial overlap (i.e., common set of nodes).

When using heuristics to test the validity of INDs in algorithm $\mathsf{FIND}_2$, some INDs may test as invalid even though they should be considered valid for our algorithm. Thus, some edges (or even nodes) of any graph or hypergraph considered by $\mathsf{FIND}_2$ may be missing. The clique finding algorithms used by $\mathsf{FIND}_2$ will then no longer find cliques that correspond to the maximal INDs in the given problem, but rather find only smaller subsets of those cliques. A simulation (Fig. 5.5) shows that the removal of as little as 5 random edges from a complete graph (i.e., a clique) of 40 or 50 nodes will generally produce a graph with around 20 distinct maximal cliques. Fig. 5.5 shows the number of different cliques in almost complete graphs of 20, 30, 40, and 50 nodes, averaged over 500 experiments each.

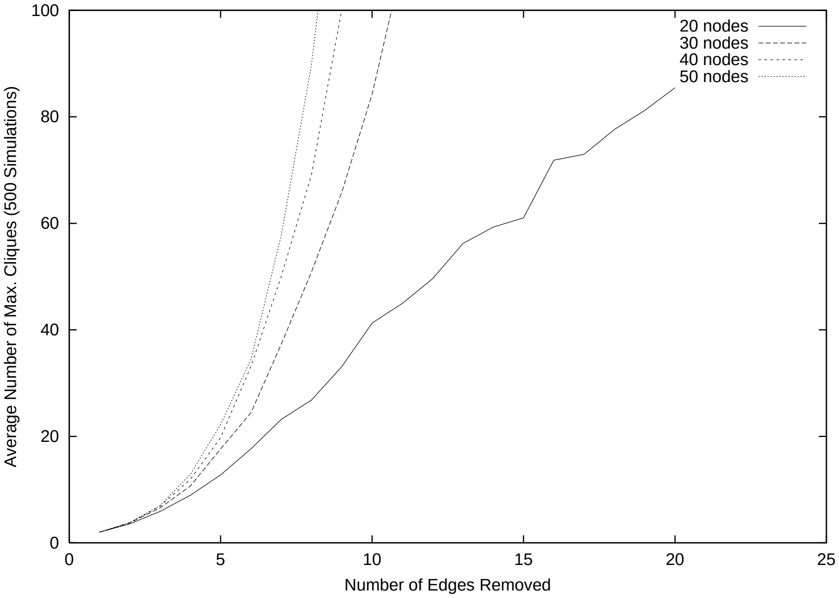


Figure 5.5: Number of Maximal Cliques in Almost Complete Graphs

However, those sub-cliques will often show substantial overlaps. Therefore, we use the following strategy:

> *When heuristics are used in* $\mathsf{FIND}_2$ *that may produce false negative results (i.e., reporting non-accidental INDs as invalid), and*

*we obtain several large, overlapping INDs from algorithm* $\mathsf{FIND}_2$*, merge those INDs by finding the union of their nodes.*

Naturally, merging *all* INDs found by algorithm $\mathsf{FIND}_2$ will in general not lead to a valid INDs, unless the (true) generating set of INDs actually contains only one IND. Therefore, we propose to only merge INDs of decreasing size, starting from the largest, until adding another IND to the result will no longer produce a valid IND.

Our experiments show that this heuristic is powerful enough to find large or maximal valid INDs even in cases when many underlying edges (small-arity INDs) are not found (Chapter 6). Of course, as only the *distinct value heuristic* and potentially the AVD heuristic produces such negative results, IND merging is not necessary when these two heuristics are either not used or when they did not reject any otherwise valid IND.

A similar merging strategy is used by ZAKI in [Zak00] for the problem of association rule mining. More information on ZAKI'S work is given in the related work section (Chapter 7).

## 5.3 Incorporating Heuristics into the IND-Testing Algorithm

Recall the $\mathsf{FIND}_2$ algorithm (Fig. 4.13, p. 89). Four of the subroutines it calls, namely **generateValidUnaryINDs**, **generateValidBinaryINDs**, **generateCliquesAndCheckAsINDs**, and **generateKAryINDsFromCliques** test inclusion dependencies against the database. As we have shown at the beginning of this section, this is a time-consuming procedure that our heuristics can help to improve. Thus, we have integrated the four heuristics discussed above in an algorithm to find INDs for large relations, to be applied when the exhaustive algorithm fails due to the inherent complexity of the problem. Instead of using simple database queries to assess the validity of an IND, we now use a separate algorithm that assess an IND's validity. The goal of this "heuristic" algorithm is to find large maximal INDs with the smallest number of actual DB queries and the smallest possible intermediate graphs and hypergraphs constructed. In order to minimize the impact on the correctness of the algorithm (i.e., on the number and size of INDs found), we apply the heuristics in order of their potential impact on the feasibility of the algorithm and the correctness of the solution (i.e., first heuristics that generate no false results, then heuristics that generate only false negatives, then heuristics that generate false positives).

This heuristic-based algorithm, called $\mathsf{CHECK_H}$, is shown in Fig. 5.6. It expands on the simple IND-checking algorithm $\mathsf{CHECK}$ (Fig. 5.3) by (1) avoiding some DB queries and (2) improving the confidence that INDs reported as valid by the algorithm are not in fact accidental.

The flowchart in Figure 5.7 shows how the four heuristics and the exact algorithm interplay. An IND with unknown validity is first subjected to the domain heuristic. If both attributes sets have matching domains, the IND is tested against the database. If it tests valid, the distinct-value-heuristic and the AVD-heuristic are applied. Only if those heuristics determine that the attributes in question have few distinct values and their value distributions are similar, the attribute name heuristic is used additionally in order to distinguish between accidental and non-accidental INDs.

```
function $\mathsf{CHECK_H}$(Relation $R$, AttribList $A$, Relation $S$, AttribList $B$)
    if (domains of $R[A]$ and $S[B]$ do not match)
        return false
    else if ($\mathsf{CHECK}(R, A, S, B) =$ false)
            return false
        else if ($\neg$(number of distinct values in $A$ is small))
                return true
            else if ($\neg$(AVDs of $A$ and $B$ are similar))
                    return false
                else if (Attribute names of $A$ and $B$ are related)
                        return true
                    else return false
```

Figure 5.6: The Heuristic IND-Checking Algorithm $\mathsf{CHECK_H}$

The $\mathsf{CHECK_H}$ algorithm will mark an IND as either valid or invalid. However, some INDs marked as "invalid" will now actually be valid in the database. The reason for this procedure is that INDs may be valid "by accident" and has been discussed in Section 5.2.1. The four functions in algorithm $\mathsf{FIND_2}$ mentioned above now simply call algorithm $\mathsf{CHECK_H}$ instead of $\mathsf{CHECK}$. In Chapter 6, we report on correlations between the accuracy of IND validity determination and the accuracy of the complete result. Naturally, the overall accuracy is reduced when IND-testing becomes less accurate. On the other hand, for relations with many attributes the runtime of $\mathsf{FIND_2}$ may be reduced so dramatically that the fact that *some* large INDs are found outweighs the drawback that not the *largest* or not *all* INDs are discovered.

Fig. 5.8 gives a flowchart of the entire algorithm $\mathsf{FIND_2}$ with $\mathsf{CHECK_H}$.

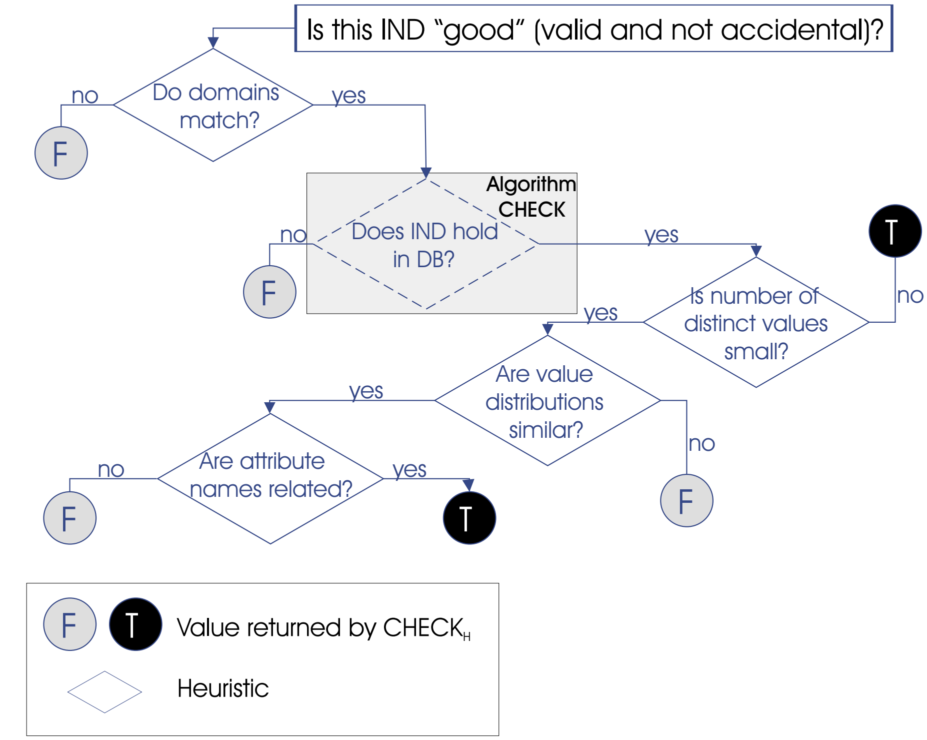

Is this IND "good" (valid and not accidental)?
Do domains match?
no
yes
F
Algorithm CHECK
Does IND hold in DB?
no
yes
F
T
Is number of distinct values small?
no
yes
Are value distributions similar?
no
yes
F
Are attribute names related?
no
yes
F
T
F T Value returned by CHECK_H
Heuristic

Figure 5.7: Flowchart for the $\mathsf{CHECK_H}$ Algorithm.

The non-heuristic algorithm is used for smaller sized problems. The cutover point between small and large problems is set to 200 unary INDs (UINDs) and/or 3000 binary INDs. This point was empirically found for the implementation used but, due to the exponential nature of the problem, it should not change much in other implementations. Given the 200 UIND-limit, we can ideally use the non-heuristic algorithm on relations with up to 200 attributes. However, our experiments showed that some relations contain a large percentage of accidental UINDs, such that the heuristics sometimes start to become effective for relations with as few as 20 attributes. When heuristics are used, the completeness of the solution is affected. IND-merging is used when several large similar INDs are found.

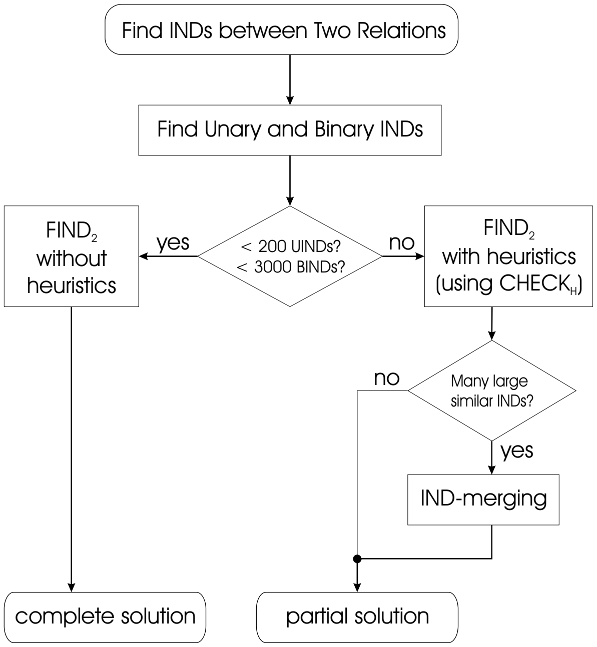


Figure 5.8: Flow of the Overall Discovery Algorithm

## 5.4 Further Runtime Reductions

Even using all four heuristics as above, some large problems cause algorithm FIND$_2$ using CHECK$_H$ to have unacceptably high runtime. Therefore, we introduce further adjustments to the algorithm that significantly reduce the algorithm's runtime, while in some cases reducing the correctness of the result further. We consider two approaches at runtime reductions: restricting the sizes of graphs simply based on the number of nodes and edges, rather than by some data-driven criteria as in the case of heuristics, and reducing the time taken by the database to test a given IND for validity by using tuple-level sampling. While our prototype software used for the experiments does use the first strategy (size restrictions), the second strategy (sampling) is only presented here as a feasibility study.

### 5.4.1 Restricting the Size of Graphs in Algorithm $\mathsf{FIND}_2$

Some real-world relations have a very small number of distinct values in many of their attributes. One test relation (`INSURANCE`) used in our experiments (Chapter 6) had less than 10 distinct values in 84 out of its 86 attributes, with a total size of over 5,000 tuples. Even using the heuristics described above will detect too many valid non-accidental INDs at the unary and binary level and the algorithm will not finish. For these cases, we propose additional restrictions. Our experiments showed that good results are still obtained when such restrictions are used, as reported in the experiments (Chapter 6). We refer to algorithm $\mathsf{FIND}_2$ (Fig.4.13, p. 89) and mark the lines in the algorithm that are affected by these restrictions.

- We introduce a measure of *support* for an IND. The support is a number composed of the number of distinct tuples in the two attributes and the actual $\chi^2$-value computed by the AVD-heuristic. A high support value means that the number of distinct tuples in the attributes is high and/or that the statistical distribution of the values in both attributes is similar.

- We sort unary and binary INDs by that support value and then only retain a certain number of those INDs for subsequent steps of the algorithm. Naturally, depending on the quality of our heuristics, “good” INDs may be removed from the problem space and the full solution will no longer be found. This is a trade-off with the runtime of the algorithm. (Lines 01 and 02)

- We also restrict the number of cliques tested against the DB (if a clique algorithm generates more cliques, the smaller ones are discarded, function **generateCliquesAndCheckAsINDs**, lines 04 and 16), and furthermore we only retain a certain number of the cliques that tested valid for the next step (i.e., we restrict the size of set $I$ in lines 04 and 16). The actual values for these restrictions were found empirically and are reported below.

- Furthermore, we restrict the number of sub-INDs generated from falsified cliques (see p. 94) by only generating sub-INDs from the largest falsified cliques found at any step (function **generateKAryINDs-FromClique**, line 12).

- Finally, if during sub-IND-generation a clique implies more than a certain number of INDs, not all INDs are generated (function **generateSubINDs**, line 14).

By restricting the sizes of internal data structures, the data structures involved in the algorithm never become too big to incur a prohibitive overall runtime of the algorithm $\mathsf{FIND}_2$. Of course, some of these restrictions, if they are used, discard valid INDs quite arbitrarily. Sorting INDs by some “support” value as explained above somewhat alleviates this problem. However, as our restrictions will ensure termination of the algorithm within a predictable time, the algorithm can be "tuned", i.e., adapted to the available time, producing the highest possible accuracy within a given time. Practical values for restrictions in our implementation (Chapter 6) are:

- 200 unary INDs (nodes in graphs/hypergraphs)
- 3000 binary INDs (edges in graph $G_2$ in Fig. 4.13)
- 2000 cliques tested against the DB
- 1000 cliques retained for subsequent steps of the algorithm, including cliques corresponding to both valid and invalid INDs
- 2000 INDs maximally generated from any single clique (line 12 in algorithm $\mathsf{FIND}_2$, Fig. 4.13).

We did not restrict the number of edges in hypergraphs. This did not seem necessary as the restrictions of the first graph $G_2$ usually prevented the construction of very large hypergraphs.

A possible extension of the algorithm $\mathsf{FIND}_2$ with $\mathsf{CHECK}_\mathsf{H}$ heuristics would be to produce a partial (incomplete) solution *quickly*, and reuse the results obtained when determining more complete solutions given more runtime. This is currently not implemented, but no fundamental obstacles to such an extension exist.

### 5.4.2 Determining INDs Using Sampling

So far, we have discussed only a simple query-based method to determine the validity of an IND in a database. However, if we relax the requirement that IND-checking be an exact procedure, *probabilistic* algorithms for this task can be used. Of course, less accurate IND-checking will adversely affect the quality of our algorithm's results. It is clear that the errors introduced by probabilistic IND checking need to be kept small. In particular, false negatives (reporting non-accidental valid INDs as invalid) pose a problem for the quality of algorithm $\mathsf{FIND}_2$.

We now give a brief overview of sampling strategies that could be used in IND discovery. Sampling in general means the determination of parameters of data sets by considering only random subsets of those data sets. By the nature of sampling, exact results are impossible to obtain. Different statistical models are used to keep errors low, but in general it is much harder to make *decisions* based on sampling than it is to *estimate* parameters within some bounds based on sampling. Clearly, it is impossible to decide whether an IND holds in the database without testing each tuple of the included attribute for inclusion.

A compromise would be to estimate some query size and decide through a statistical test whether a valid IND is likely to exist or not. There is substantial work on the estimation of the size of query results in databases. Work by Hou and Özsoyoğlu [HÖ91] developed statistical estimators for several aggregate queries, such as COUNT-queries. The authors suggest simple estimators for the values of COUNT-queries. Those can be used to assess IND validity by estimating the number of tuples in an INTERSECT query through sampling.

**Example 5.7** *Consider the problem of testing IND $R[A] \subseteq S[B]$. If an es-*

*timator is available that determines the approximate size of an INTERSECT query, we can estimate the size for the following SQL-query:*

```
select count (*) from (
  select A from R
    intersect
  select B from S
)
```

*If the size of the result of this query is close to $|R|$, the IND is likely to be valid. If the result size is much less than $|R|$, the IND is invalid.*

Results reported by Hou and Özsoyoğlu suggest that the error in estimating counts increases significantly as the sampling fraction is reduced, but *decreases* with less selective predicates (larger query results) in the query. For a sampling rate of 5% over two relations of 20,000 tuples each, they report relative errors of 10%–50% for the estimation of the size of an intersection query, depending on the selectivity of the query. They also report estimates for errors in simple random sampling. For two relations $R$ and $S$ and a sample of size $s$ from each relation, the relative error is estimated as a function of the size of the intersection $|R \cap S|$ as

$$e = \frac{1}{s} \cdot \sqrt{\frac{|R||S|}{|R \cap S|}} \tag{5.4}$$

Thus, for two large relations $R$ and $S$ with $|R| \approx |S|$, $|R \cap S| \approx |R|$, the relative error of the count estimate of an intersection query is $e \approx \frac{1}{f} \cdot \frac{1}{\sqrt{|R|}}$, with $f$ the sample fraction. If the actual size of the intersection is smaller, the relative error increases. Clearly, the error also increases with smaller samples, and smaller relations. This result is slightly simplified relative to what is described in [HÖ91], as we are only giving an intuition about the usefulness of sampling in our algorithm.

A large body of work exists on sampling from relational tables, such as [OR94, Toi96, JL96, CMN98, MTL00], such that techniques for sampling are well established theoretically and practically. Simple random sampling is very sensitive to skewed data. The results above hold only for uniformly distributed data. More complex sampling schemes have been proposed by several authors, such as Lipton *et al.* [LNS90], Haas *et al.* [HS95, HNSS95, HNSS93], Poosala and Ioannidis [PI97], and, in particular Ganguly *et al.* [GGMS96], who proposed *bifocal sampling*. This method overcomes the limitation of previous methods that either assumed uniformly distributed or skewed data, and essentially samples a data set twice, once under the assumption of uniform distribution and once under the assumption of skewed distribution. The results presented by Ganguly *et al.* suggest that in order to achieve "good" estimations for the size of a join (one can see intersection queries as joins over all attributes), a sample of size at most $O(\sqrt{n}\log n)$ is sufficient as long as the join size is $\Omega(n \lg n)$ for relations of $n$ tuples.

We can use sampling schemes such as the above as a method to reduce the runtime of our algorithm. However, the influence of the naturally higher error rate on the accuracy of the results of algorithm $\mathsf{FIND}_2$ remains problematic. This could be a topic for future work. For the current work, we did not feel that incorporating sampling would substantially contribute to either the correctness or runtime reductions of the algorithm. However, some work exists in the literature [KM95] suggesting that for certain universally quantified statements (assertions that make a statement about *all* tuples in a database), sampling can help to gather statistical evidence about their validity in a database. Those ideas could be used in incorporating sampling into our scheme.

# Chapter 6

# Experiments and Evaluation

## 6.1 Implementation

We implemented algorithm $\mathsf{FIND}_2$ in Java over Oracle relational databases (using JDBC). The implementation is modular, such that different aspects of the algorithm can be tested separately. INDs are modeled as objects, which simplifies the code. Some implementation improvements could be made by representing INDs as simpler data structures, however that was not the focus of the prototype implementation. All heuristics and strategies described in this dissertation, with the exception of tuple-level sampling, are implemented and can be turned on or off in the prototype. The schemas of the test databases can either be queried or explicitly specified (if only subsets of the attributes in the source relations are to be tested).

Figure 6.1 gives an overview over the system architecture. The implementation has two major modules: the *IND-Generator* which generates INDs to be tested, and the *IND-Tester* which tests the IND in question through a database query and/or through applying the four heuristics from Fig. 5.7. Those two modules roughly correspond to the $\mathsf{FIND}_2$ and $\mathsf{CHECK}/\mathsf{CHECK}_\mathsf{H}$ algorithms, respectively. At the end of the discovery process, the IND-Generator also assumes the task of reporting all INDs that tested as maximal and valid.

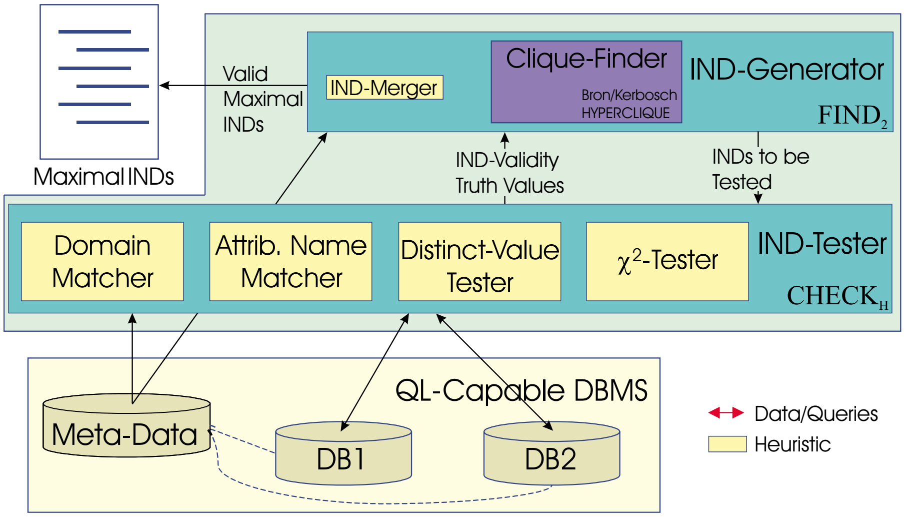


Figure 6.1: Overview of System Architecture

The implementation has a total of 6 parameters that can be "tuned". Five of those parameters restrict the sizes of certain data structures, as described in Sec. 5.4.1. In particular, those are: the maximal number of nodes in any graph (i.e., unary INDs), the maximal number of edges in graph $G_2$ (i.e., binary INDs), the maximal number of cliques tested as INDs at any one step, the maximal number of cliques *retained* for the next step, and the maximal number of INDs generated from any clique by function **generateINDsFromCliques**. The sixth parameter that can be set in the implementation affects the *distinct-values* heuristic and sets the minimum number of distinct values in an attribute or attribute set that is necessary for the distinct-value heuristic to *not* reject the IND as accidental. For Experiment 4, we additionally used "fuzzy" IND checking, as explained in the detailed description of that experiment.

## 6.2 Experimental Setup

The testing environment for all our experiments consisted of two Linux-machines—one 400-MHz-Pentium II (Linux-Kernel 2.2.14) running Blackdown Java V1.3.0 for the system and one 800-MHz-Pentium III (Linux-Kernel 2.2.16) running Oracle 8i for the databases. For simplicity, we only tested the algorithm on tables within the same database instance, as the added slowdown of having to test INDs across different databases would lead to no interesting new insights into our implementation.

We ran experiments on four data sets obtained from the UC Irvine KDD Archive [1]. The data sets (converted into relational tables in Oracle) are

- **INSURANCE:** Data from the "CoIL 2000 Insurance Company benchmark" from the KDD archive. A table with 86 attributes and approximately 6000 tuples. A significant feature of this data set is that all attributes are *encoded* by integers. The effect is that nearly all domains in this table are the same, and also have very low distinct-value counts (all but 2 attributes have fewer than 10 distinct values). Many accidental INDs are thus created. In this data set, our heuristics have

[1] URL as of September 2001: `http://kdd.ics.uci.edu`

the most beneficial effect.

- **INTERNET:** The Internet Usage Data set containing 72 attributes and approximately 10,000 tuples. This data set also has many accidental INDs. It is in many ways similar to the **INSURANCE** data set but has clearly defined semantics (its attributes have clear meaning) so that we used it to test the AVD heuristic by applying predicates to certain attributes.

- **CENSUS:** The Census Income Database containing 41 attributes and approximately 200,000 tuples. As most attributes have unencoded domains (i.e., the domains are largely distinct sets of strings), our algorithm can find maximal INDs in this data set without heuristics and thus this set is mainly used for performance (rather than quality) testing.

- **CUP98:** A data set of direct-mail data used for the KDD-Cup 1998. A relation with 481 attributes and about 95,000 tuples. This relation has too many attributes for our purpose, so we used two different projections of this relation onto 80 attributes each, with an overlap of 60 attributes. That is, the maximal IND between the two subsets is 60-ary.

From each of these data sets, we generated overlapping subsets, partly through random sampling (using Oracle's SAMPLE clause and row-level sampling), and partly through selection (predicates, Experiment 4). The size of the overlaps varied from about 20% to 100% of the size of the smaller relation.

Throughout our experiments we did not use the attribute name heuristic as this would make it more difficult to assess the general properties of our algorithm. Especially the fact that the data sets we compared were generated from the same original data set makes it difficult to assess the exact benefit of the attribute-names heuristic.

## 6.3 Experiments

Our experiments are measuring the following factors and dependencies:

- The runtime behavior of the exact algorithm $\mathsf{FIND}_2$ (using $\mathsf{CHECK}$) versus the heuristic algorithm ($\mathsf{FIND}_2$ using $\mathsf{CHECK}_\mathsf{H}$),
- the influence of the number of attributes and the relation size on the runtime of both the heuristic and non-heuristic algorithm,
- the influence of different domains, and in particular different numbers of distinct values, on the runtime of the algorithm,
- the applicability and influence on quality of the attribute-value-distribution (AVD) heuristic, as the applicability of the AVD heuristic for data sets obtained through selection by some condition is not obvious,
- the influence of small amounts of noise on the data, since noise will prevent exact inclusion dependencies from holding.

The size-restricting strategies from Sec. 5.4.1 were applied whenever the size of the problem would have prevented algorithm $\mathsf{FIND}_2$ from finding a solution otherwise.

As a simple quality measure for the output of the algorithm, we use the largest inclusion dependency found by it (in relationship to the largest IND that *exists* in the database). Other quality measures are conceivable, such as measures that take user preferences for some attributes into account (along the lines of the preference model in the *EVE*-Project, Sec. 2.2.2).

### 6.3.1 Experiment 1: Number of Unary and Binary INDs

In a “nice” data set, there will be few accidental INDs. That is, the number of unary INDs will not be much larger than the number of attributes. We performed an experiment on data set INSURANCE to assess the number of unary INDs (UINDs) and binary INDs (BINDs) in a “bad” case with many accidental INDs. Figures 6.2 and 6.3 show the results. For a number of different projections of $k$ attributes of relation INSURANCE, Fig. 6.2 shows the number of possible UINDs ($k^2$) and the number of UINDs that tested *valid* in the database. Fig. 6.3 shows the number of possible BINDs (which depends on the number of valid UINDs) and the number of BINDs that tested *valid*. Clearly, there are many more than $k$ UINDs in the first case and the number of BINDs is prohibitively high even for relations of as few as 35 attributes. In fact, the basic (non-heuristic) algorithm $\mathsf{FIND}_2$ does not finish for this data set for projections with more than 10 attributes.

Table 6.1 shows the number of unary and binary INDs found in each of our data sets, which form the nodes and edges, respectively, in the graph $G_2$ used for clique finding in algorithm $\mathsf{FIND}_2$. In each case, we listed the largest projection of a table for which these values could be determined

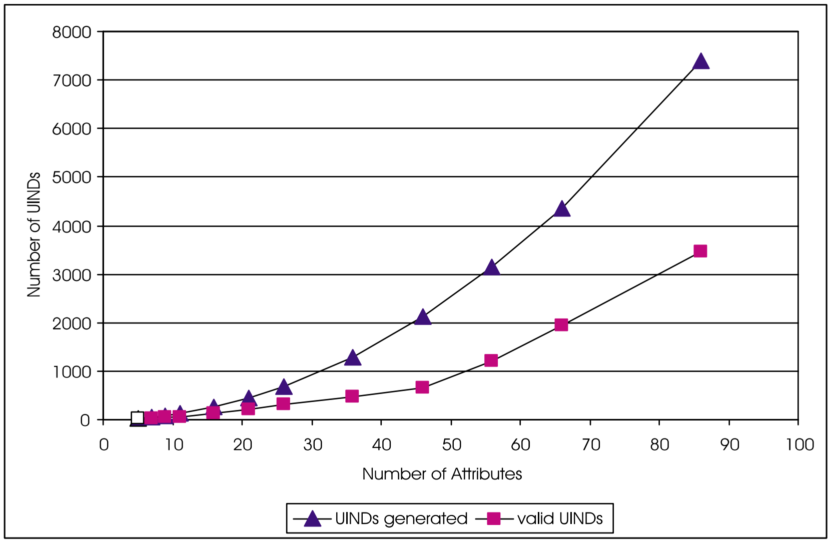


Figure 6.2: Number of Unary INDs in Data Set INSURANCE

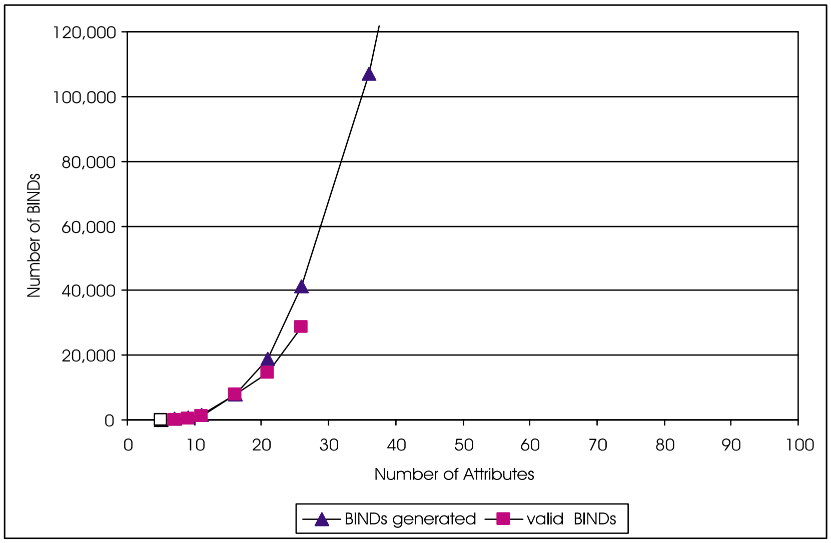


Figure 6.3: Number of Binary INDs in Data Set INSURANCE

within reasonable time.

| **Data set** | **Attrib.** | **UINDs** | **% of possible INDs** | **BINDs** | **% of possible INDs** |
|---|---:|---:|---|---:|---|
| CUP98 | 80 | 278 | 4.3% | 14136 | 38% |
| CENSUS | 41 | 69 | 4.1% | 994 | 45% |
| INTERNET | 20 | 173 | 43% | 8398 | 65% |
| INSURANCE | 26 | 304 | 45% | 28793 | 70% |

Table 6.1: Number of Valid Unary and Binary INDs in Different Data Sets

Given that graphs with more than about 200 nodes and 3000 edges constitute problems for the clique-finding algorithm (Sec. 5.4.1), the results reported in this experiment clearly indicate that for many real-world problems, the non-heuristic algorithm will not work. In the case of the data set INSURANCE, algorithm $\mathsf{FIND}_2$ without heuristics will not work for even 20 out of the 86 attributes in the base relations. This motivates the need for heuristics in our algorithm. On the other hand, for some data sets (e.g., our test data set CENSUS), the basic algorithm will work quite well and produce the correct and complete result in short time.

From Table 6.1, in connection with our further experiments, it can be seen that the percentage of UINDs that is valid between two relations $R$ and $S$ (as a fraction of the maximal number $|R||S|$) is a good indicator of whether the data set has few accidental INDs or many, i.e., whether the INDs in the data set can be easily discovered by the $\mathsf{FIND}_2$ algorithm. This number can also be used to assess at an early stage whether the algorithm $\mathsf{FIND}_2$ without using heuristics and/or without restricting the sizes of internal data structures will finish within reasonable time or not.

### 6.3.2 Experiment 2: Performance and Quality Effects of Heuristics

This experiment was conducted to assess the runtime of the algorithm and the quality of its output for a given data set, with and without the use of heuristics. For this experiment, we used a 5000-tuple random subset CENSUS1 of data set CENSUS and a further random subset of 4500 tuples (90%) of CENSUS1. We compared the performance and quality of algorithm $\mathsf{FIND}_2$ with and without heuristics. We used different projections on the set CENSUS, which has 41 attributes. Figure 6.4 shows the runtime of algorithm $\mathsf{FIND}_2$ with and without heuristics, for different size projections. Figure 6.5 shows the size of the largest single IND found by the heuristic algorithm in each case.

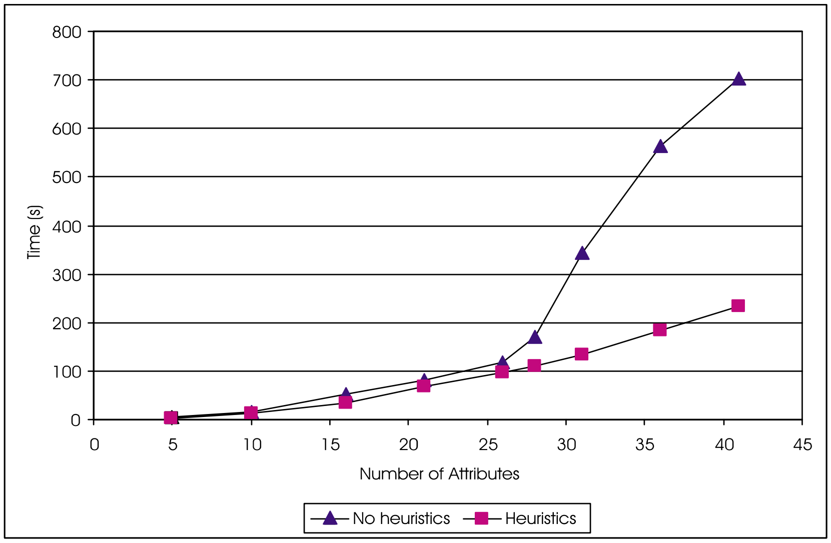


Figure 6.4: Performance of Algorithm $\mathsf{FIND}_2$ Using CHECK and $\mathsf{CHECK}_\mathsf{H}$, Respectively for Data Set CENSUS.

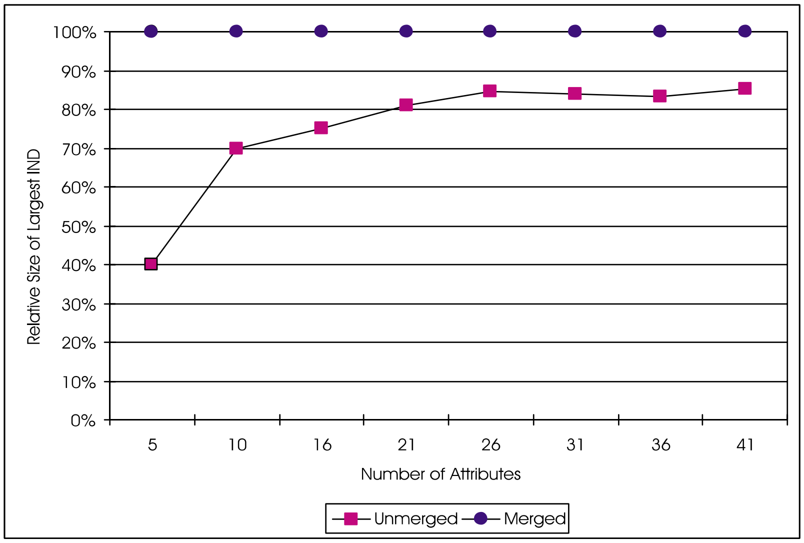


Figure 6.5: Quality of Algorithm FIND$_2$ Using CHECK$_H$ for Data Set CENSUS.

The experiment shows the large performance benefits of the heuristic approach. For projections of data set CENSUS (Fig. 6.4) of an increasing number of attributes, the runtime for the heuristic version of algorithm FIND$_2$ increases much slower that the runtime for the non-heuristic version, yielding a reduction in runtime by two thirds for the full 41-attribute relation.

Or course, there is a penalty in accuracy as a tradeoff for the lower runtime. The full generating set of INDs is not found by the heuristic algorithm, rather FIND$_2$ reports a maximum IND whose arity is about 70%-85% of the largest valid IND between the test data sets. However, through IND merging (Sec. 5.2.8), we still correctly find the largest IND in all cases, for this data set. In other cases, the results of clique merging (Sec. 5.2.8)) are not perfect as here, but still large INDs are found, as shown in the next experiment.

### 6.3.3 Experiment 3: Effect of Low Numbers of Distinct Values in Data Set

In this experiment, we wanted to assess the quality of the heuristic algorithm in a data set with many accidental INDs. Table INSURANCE is such a data set, as nearly 50 percent of its UINDs are valid. For the full data set of 86 attributes, graph $G_2$ would have 4000 nodes, such that a clique-finding algorithm would not finish.

Figure 6.6 shows the quality achieved by the heuristic algorithm for this case, for different size projections of table INSURANCE. Both the size of the largest IND found directly by algorithm $\mathsf{FIND_2}$ and the size of the largest *merged* IND are reported. The quality is high for small relations (less than 30 attributes), since fewer than 200 unary INDs and fewer than 3000 binary INDs are found for those relations (using algorithm $\mathsf{CHECK_H}$). No INDs are therefore excluded from the input to the clique-finding algorithm by the restrictions listed in Sec. 5.4.1. For larger relations, an increasing number of UINDs and BINDs are discarded, such that the quality of the final result becomes lower. The power of the IND-merging strategy (merging large INDs as long as a valid new IND is formed) becomes clear for large relations, as the size of the largest discovered IND (relative to the size of the largest *existing* IND) actually increases with larger relations. The algorithm $\mathsf{FIND_2}$ *without* heuristics fails for this data set for all cases with more than 10 attributes, so no results for the non-heuristic algorithm can be reported for comparison.

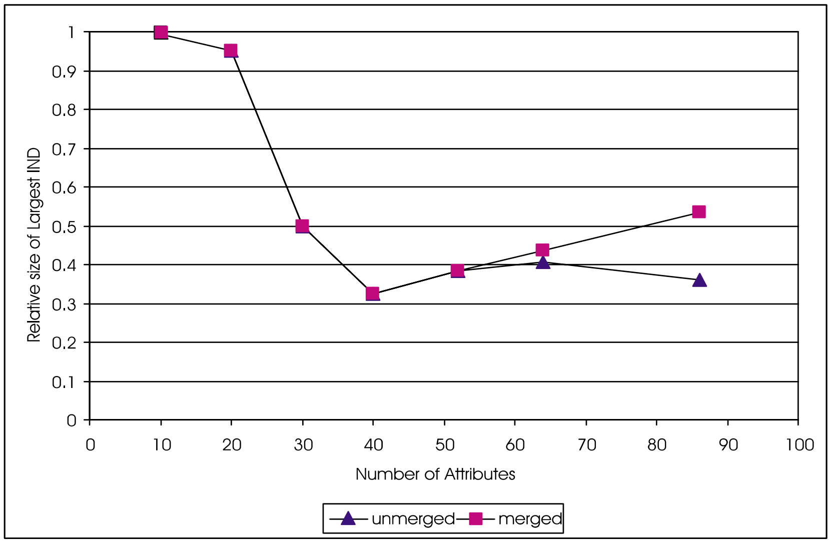


Figure 6.6: Relative Size of Largest IND Discovered, Data Set INSURANCE

Given that the non-heuristic algorithm is practically worthless for this data set, the performance of the heuristics is quite good. Its major advantage is that is produces results for any size data set, not only for a data set with 10 or less attributes. In the worst case for this experiment (the projection on 40 attributes, with a valid IND of also 40 attributes), an IND of 13 attributes (32.5 percent) is found. The increase in quality for relations with over 40 attributes can be attributed to the particular choices for the projections of the originally 86-attribute data set INSURANCE. As the data set has many attributes with very few distinct values, it is likely that some of the few attributes with higher numbers of distinct values were projected out in the smaller-size problem, such that the algorithm encountered more noise in the unary and binary INDs and discarded a higher percentage of good (valid and non-accidental) INDs.

A possible extension of our algorithm here is to *repeat* the experiment with the attributes that are not members of the IND found (in this case we would have a second IND-finding problem with 26 instead of 40 attributes, and so on), and try to merge the INDs found. We tried this strategy in this case and found that an IND of 38 attributes was found after three runs of algorithm $\mathsf{FIND}_2$ with heuristics.

### 6.3.4 Experiment 4: Accuracy of the $\chi^2$-Test and the Attribute Value Heuristic

The *attribute value heuristic (AVD)* relies on the assumption that attributes that stand in an inclusion relationship to one another are semantically related and thus show a similar distribution of their values. This will certainly be true if the two relations in question are actually random samples of some larger real-world data set. However, the value distribution might not be preserved if a subset of a relation is formed by some *predicate.* That is, if algorithm $\mathsf{FIND}_2$ is run on two relations $R$ and $S$, with $R = \sigma_{\overline{A}}(S)$, the value distribution in some attributes in $R$ might be different from the value distribution in some attributes in $S$. That is obvious for all attributes in $\overline{A}$, but not necessarily true for other attributes. For example, if a data set containing information about individual people is selected on the GENDER attribute, it is clear that an attribute HEIGHT in the subset will have a different value distribution than the same attribute in the whole relation, while the distributions will likely be equal for an attribute IQ. Thus, we performed a number of experiments in which we generated subsets of our data sets using *predicates* rather than random sampling. The expectation was that whenever heuristics are used, the AVD heuristic will falsely discard INDs that are actually expressing semantic relationships between attributes, such that the maximum IND will no longer be found.

Figure 6.7 shows the quality (ratio of size of largest IND found to size of largest existing IND) of the result in data set INTERNET for four different predicates. The data set represents a survey in Internet usage data, and we selected the following four attributes for predicates: *gender*, *household income*, *country of origin*, *major occupation*, with conditions that had selectivities between 0.45 and 0.8. Table 6.2 shows the exact predicates ($\sigma$=selectivity of predicate). In all cases, there existed a 72-ary IND between the two relations tested, and heuristics as well as IND-merging were used to obtain the result.

| Attribute | Predicate | $\sigma$ | Decoded |
|---|---|---|---|
| GENDER | >= 1 | 0.6 | `<>'female'` |
| HOUSEHOLD_INCOME | <= 6 | 0.8 | `<75,000` |
| COUNTRY | <= 30 | 0.46 | `='US' AND state <= 'North Carolina'` |
| MAJOR_OCCUPATION | <> 99 | 0.77 | `<>'other'` |

Table 6.2: Predicates Used on Dataset INTERNET to Evaluate AVD Heuristic

We performed similar experiments with our other data sets and found that the AVD heuristic helps to find between 50 percent (data set CUP98_80) and 10 percent (data set INSURANCE) larger INDs than the algorithm without this heuristic, averaged over several different predicates. The algorithm without the AVD heuristic never produced a larger IND and also showed significantly lower performance in many cases.

This experiment shows that using the AVD heuristic gives better results

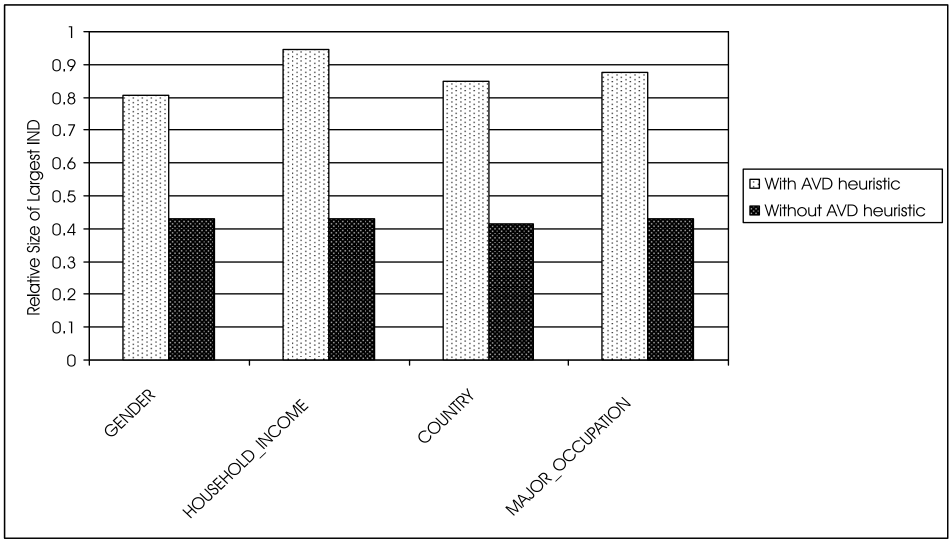


Figure 6.7: Quality of Heuristic Algorithm for Subsets Generated Through Predicates

(i.e., larger INDs) in most of our experimental cases in which it was actually applied. The heuristic does not affect the result adversely, since it is only used when (1) the number of valid UINDs and BINDs is too large for the exact algorithm to run, and (2) the distinct-value heuristic has already rejected the IND (see Figs. 5.6 and 5.7). It is possible that some attribute pairs that would be found without using *any* heuristic may not be found when the AVD heuristic is used, but as explained above, the large improvements in the problem size feasible for the algorithm outweigh this drawback.

### 6.3.5 Experiment 5: Effect of Data Set Size on Runtime

This experiment was performed to assess the relationship between the sizes of the underlying relations and the performance of algorithm $\mathsf{FIND}_2$. We let

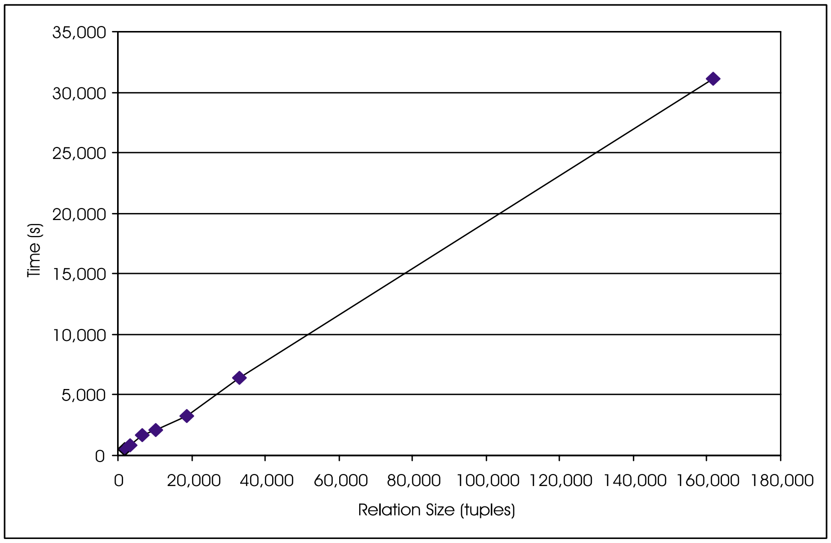


Figure 6.8: Effect of the Size of Data Sources on Performance of $\mathsf{FIND}_2$

the algorithm discover INDs in different (random) subsets of the same data set (CUP98) and recorded the runtime of the algorithm versus the sum of the numbers of tuples in both input relations. Fig. 6.8 shows the results. There is a near linear correlation between the two variables, suggesting that the query execution time for the SQL INTERSECT queries used (together) grows linearly with the relation size. There is also a linear correlation between the execution time of *one* such query and the size of the relations involved, which is in agreement with theory if external merge sort is used in the database to compute the queries.

This experiment suggests that, as long as our basic strategy of a separation of the discovery and testing modules is used (Fig. 4.1), the possible reductions on runtime achieved by an improvement of the IND-testing algorithm are small (since a linear algorithm for IND-testing cannot be improved much in comparison to the order-of-magnitude improvements of algorithm $\mathsf{FIND}_2$ vs. the naïve algorithm).

### 6.3.6 Experiment 6: Effect of Noise on the Correctness of Algorithm $\mathsf{FIND}_2$

In this experiment, we assessed the influence of small amounts of noise (tuples violating the exact inclusion property) on the $\mathsf{FIND}_2$ algorithm. We experimented on the INTERNET dataset, by creating a 90% random sample and a 99% random sample from the original data set, such that roughly 1% of tuples in the smaller relation violated the target maximal inclusion dependency. We then relaxed the condition on IND-testing (algorithm CHECK) by declaring an IND valid if it is violated by less than 1% of the tuples in the smaller relation.

Fig. 6.9 shows the ratio of the largest IND found by the algorithm $\mathsf{FIND}_2$ modified as above, versus the largest IND that existed (with 1% noise) in the database. IND merging was used in obtaining the result, and results are reported with and without IND merging. The figure shows that there is some influence of noise on the completeness of the result, but that a large sub-IND of the largest IND is still found in the cases tested. The largest single (unmerged) INDs that were discovered by our algorithm in this experiment had sizes of roughly 50% of the target size, but could be successfully merged to reach at more than 90% of the target size, as shown in the Figure.

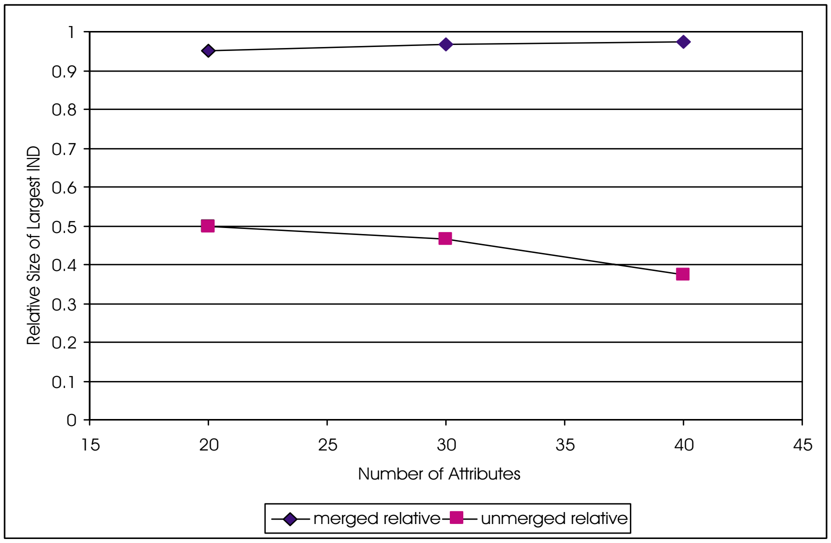


Figure 6.9: Effect of Noise on the Quality of $\mathsf{FIND}_2$

# Chapter 7

# Related Work

The work presented here uses results from a variety of areas of research. We give an overview over related work in each area.

**Inclusion Dependencies.** Inclusion dependencies have been widely studied on a theoretical level. Early work on inclusion dependencies deals mostly with the implication problem, i.e., with the question of how to derive inclusion dependencies from other inclusion dependencies and/or functional dependencies (FDs). Fundamental work is done by Casanova, Fagin and Papadimitriou [CFP82]. They present the simple axiomatization for INDs used in our work and prove that the decision problem for INDs is PSPACE-complete. They also show further theorems on the interaction between INDs and functional dependencies (FDs). In particular they show the absence of a complete axiomatization for a system of both INDs and FDs. Later, Mitchell [Mit83] developed inference rules for INDs. Among other rules, Mitchell shows an inference rule called *Collection* that can derive new

INDs from existing INDs and FDs. We did not use those rules as their potential benefit (i.e., the number of new INDs derived through those rules) is small and using those rules would require comprehensive knowledge about FDs in the relations in question. Cosmadakis *et al.* [CK84] give results very closely related to [Mit83], but using a graph-based approach. Missaoui and Godin [MG90] give a graph-based approach for the use of transitivity for the IND inference problem, but restrict themselves to so-called *typed* INDs. No discovery on the data-level is mentioned. Cosmadakis *et al.* [CKV90] also survey implication problems for *unary* INDs only.

A recent paper by Levene and Vincent [LV00] argues for the definition of a new "inclusion dependency normal form" which could remove some important redundancies in databases. Earlier, Fagin [Fag81] had argued that knowledge about inclusion dependencies is important in database design. Work like this supports our claim that the discovery of inclusion dependencies is an important topic.

**Graphs and Hypergraphs.** Information on graphs and cliques in graphs is available widely in textbooks (e.g., [Ski97, GJ79]). Even though the clique-finding problem is NP-complete, many algorithms have been proposed that solve this problem for small input sets. Examples include the Bron/Kerbosch-algorithm used in our work [BK73], an early algorithm by Bierstone and its corrections [MC72], and more recent algorithms (a survey is given in [PX94], see also [Woo97]). All this work is restricted to graphs (i.e., 2-uniform hypergraphs).

Work on ($k$-uniform) hypergraphs is somewhat more limited. Some information on hypergraphs is available in textbooks [Ber89, GGL95]. Several authors have investigated maximal independent sets in hypergraphs, which is a problem related to the clique-problem. Garrido *et al.* [GKL96] give an algorithm that assumes a parallel architecture and computes the related problem of *maximum independent set (MIS)*, finding one MIS in a hypergraph, while putting certain restrictions (low arboricity) on the graph. Their algorithm does not compute all MIS in a graph. LaPadula and Picollelli [LP01] give some basic mathematical results on hypergraphs concerning maximal independent sets and other properties. Boros *et al.* [BEGK00] give additional properties of hypergraphs and MIS with respect to parallel algorithms.

**Discovery of Patterns in Databases.** There is substantial work on the discovery of patterns in databases. Much work is concentrated on functional dependencies (FDs), such as Lim and Harrison [LH97], Bell and Brockhausen [BB95a], Huhtala *et al.* [HKPT98], Knobbe and Adriaans [KA96] and Savnik and Flach[SF93]. The last two authors also use a machine-learning approach to find FDs [FS99].

An important related paper is by Kantola, Mannila *et al.* [KMRS92]. The authors describe an algorithm for discovering functional dependencies and also mention inclusion dependencies. However, no algorithm for IND discovery is given, and only a very rough upper bound for the complexity of the IND-finding problem is presented (in addition to a proof of NP-completeness of the problem).

Cohen investigates the problem of non-matching, semantically-equivalent values in databases, which we have not covered in this work [Coh98]. The author presents an algorithm to find related data values based on statistical properties in a database. The focus of this work is very different from ours, but could be complementary in future applications. Cai *et al.* [CCH91] discover classification rules from a database. They present a machine-learning technique that infers classification rules from patterns in the data of a database. Again, their focus is different from ours and could be complementary to our work.

Another important body of work originates in data mining, namely the problem of the discovery of association rules. The key idea of the Apriori algorithm used in association rule mining (presented by Aggarwal and Yu [AY98]) is also used in our solution as information about lower-arity INDs is used in generating higher-arity INDs. Much work in the literature deals with implementation issues on Aggarwal's algorithm (as one example, [STA98]) or generalizations of the problem [TUA$^+$98].

Zaki [Zak00] presents algorithms for association rule mining. In fact, the author presents an algorithm that also uses the concepts of finding cliques in graphs and clique-merging in a graph modeled for the association-rule mining problem. However, there are important differences between Zaki's work and ours:

1. Association rules have the inherent concept of *support* as a real number, whereas we are looking for a binary property (inclusion) in our database. Therefore, our algorithm cannot use the simple sorting strategy that Zaki proposes for an optimization of his association rule

algorithm. As less information about each basic unit (association rule for Zaki, IND for us) is available, our algorithm must find solutions with less information as input.

2. Association rules are very unlikely to be as large (in "arity") as our inclusion dependencies. In fact, it can be said that in many applications a small association rule is more useful than a larger one (with lower support), whereas in our domain, finding the largest correct IND is the optimal goal for the algorithm. Zaki reports item sets with 22 items or less in his largest experiments. In contrast, a relation with a 22-ary IND poses few problem for even our non-heuristic algorithm. In his approach, the author does not use heuristics to restrict the problem size. In contrast, heuristics are very important for us since many real-world problems are too large for the exhaustive algorithm to produce good results.

3. In association rule discovery, techniques for "clique-merging" and item-set-merging are used. While we employ some of those techniques as a last step in our algorithm for very large problems, merging is the main step for finding larger association rules in the work by Zaki. In the INDs-finding problem, such simple merging cannot be used as the success rate (i.e., the probability that merged INDs are valid) is very small. This again is a consequence of the lack of the concepts of "support" or "confidence" in our problem. Furthermore, Zaki's algorithm employs clique-finding only for traditional graphs (i.e., 2-hypergraphs) and not for higher-dimensional graphs. In our case, cliques also have

to be found in higher-dimensional graphs (i.e., $k$-hypergraphs with $k > 2$) and clique-merging is only attempted as a last step for certain problems for which only low-quality solutions would be found otherwise.

In summary, while the two approaches use some of the same mathematical foundations (such as the mapping of the respective discovery problem to the clique-finding problem), the application of the algorithm in [Zak00] to solve our problem is not possible. Rather, we have made substantial contributions beyond the initial idea of mapping the IND-finding problem to a clique-finding problem in order to achieve a practically feasible solution to the IND-finding problem in high-dimensional data sets.

Other work co-authored by Zaki on association rules includes a parallelization of association rule mining algorithms [ZOPL96].

Hypergraphs have been used in other areas of databases and data mining. For example, Mannila and Raiha [MR94] give an algorithm for the discovery of functional dependencies that maps the problem to a hypergraph traversal. Catarci and Tarantino [CT92] use hypergraphs for modeling the structure of databases. A similar approach is presented by Wang and Wong [WW96], who introduce *attributed hypergraphs* to represent order patterns in databases.

**Heuristics.** There is substantial related work on some of the heuristics that we have used to restrict problem spaces in our algorithm. Ontologies for attribute names are treated in more general ontology work such as [MWJ99, Gua95, UG96, GPB99] and also more specifically for databases [YJSD91].

Crow and Shadbolt [CS01] describe methods to generate ontologies by using information on the Web. Distance metrics as a simple replacement for ontologies (as mentioned in this work) can be found for example in [WM97, Ukk92, Hyl96]. Work on the theory of attribute value distributions can be found in [MCS88] or [HZZ93]. The statistical $\chi^2$-test itself is described in statistics textbooks such as [Ric95].

**Sampling.** In order to further reduce the runtime of our algorithm, we have proposed the use of sampling (rather than using SQL queries) to test INDs against a database. A large number of authors have investigated the theory of sampling, in particular the question of how and to what extent database properties can be determined by examining a random sample of a database rather than the entire database. Since sampling naturally introduces errors into the conclusions one makes based on it, the accuracy of algorithms that rely on database statistics is lower than the accuracy of exhaustive strategies. Work on using sampling for association rules has been done by Toivonen [Toi96] and Zaki *et al.* [ZPLO97]. Kivinen and Mannila [KM94] also present theoretical results on the minimal error bounds of sampling. Their work supports the conclusion that the errors introduced by sampling are too large to make sampling very beneficial to our algorithm. In most real-world cases, the error for the size of the intersection between two relations that can be obtained from a sample of a useful size exceeds 100%, which supports no useful hypothesis testing on the validity of an IND. Work on determining quantitative database parameters (which we can indirectly use for determining IND validity) is extensive and includes [CMN98, MTL00, LNS90, GGMS96, HNSS95, PI97, PSC84, HNSS93, PIHS96, CR94, HÖ91, JL96]. Most of these authors are concerned with efficient ways to estimate query results (mainly aggregation queries) from samples of a database. In principle, such work could be used to establish statistical tests for the acceptance or refusal of hypotheses about INDs, based on data samples. However, the errors introduced by all these statistical models make an application for our problem difficult.

**Schema Integration.** Schema integration is not limited to the discovery of INDs. Beyond the approach used in this work, which discovers relationships between data objects based on data and very simple meta-data (i.e., domains and attribute names only), there is a massive amount of work on schema integration in general—from manual approaches using domain experts to fully automatic systems using meta-information structured in a variety of ways. Those projects often aim for the larger goal of automatically or semi-automatically integrating entire databases, but do not give algorithmic solutions on how such an automatic integration can be done on the data level.

LARSON *et al.* [LNE89] give a theory in which they infer attribute equivalence by a variety of indicators, such as domains, maximal and minimal values, and any constraints imposed by the (relational) database system. Their work is complementary to ours in some sense but ignores the actual data inside the attributes. Therefore, it is very sensitive to the availability and correctness of their assumed constraints. We are using some of the same indicators as LARSON *et al.* , in particular the domains of attributes. The integration of further indicators suggested by their paper did not appear helpful in our context, due to a lack of availability and/or significance of those factors for the case of IND discovery.

Semi-automatic approaches include [PSU98] (described in detail in a dissertation by Ursino [Urs99]), Sheth, Larson *et al.* [SLCN88], the MOMIS project [BBC$^+$00], and others [MWJ99, LC95, LC00]. Machine-learning, neural networks and other artificial intelligence concepts are often used for schema integration [LC94, DDH00, Klu95, DP95]. Rahm and Bernstein [RB01] give an overview over some recent schema-integration projects; an earlier survey is [BLN86]. Finally, Roddick *et al.* [RCR96] reviews the implications of using discovered meta-information rather than explicitly known meta-information.

**Discovery of Resources for Data Integration** A paper that in its goal of finding database relationships is related to ours is by Cho *et al.* [CSGM00]. In their work, the authors investigate the discovery of replicated web sites. They make use of the set of hyperlinks in a HTML-file (and compare the graph of links) in order to assess the relationship between web sites. Their approach could solve some of the same problems as ours, however the reliance on the structure of links rather than the actual data content as their first strategy to reduce the problem space is different from our focus. Also, our emphasis is on data sources that have identifiable attributes with data values in a particular domain, whereas Cho *et al.* focus on semi-structured data. Their solution is not applicable to databases in models other than the semi-structured model.

Liu *et al.* [LLY98] introduce a concept of *relevance* for databases in multi-database mining. They select "interesting" *databases* from a set of available ones, which is not our focus.

# Chapter 8

# Conclusions

In this part of the dissertation, we have proposed an algorithm called $\mathsf{FIND}_2$ for the problem of discovery of inclusion dependencies in databases. Inclusion dependencies are an important form of database interrelationships, expressing redundancies between databases. With our solution, it is possible to automatically compare two databases with known schema, but unknown interrelationships, and identify inclusion dependencies between their attributes.

This is the first solution of the inclusion dependency discovery problem. Previously, only implication rules for inclusion dependencies were known [CFP82]. There is some work on discovering other types of dependencies, especially functional dependencies [KMRS92, FS99], but no algorithm for inclusion dependencies had been proposed.

The discovery of inclusion dependencies is a hard problem, with an inherent NP-complexity [KMRS92]. By reducing the problem to a graph problem, we achieved a significant improvement in performance over the naïve algorithm. Our algorithm uses an NP-complete graph algorithm (clique-finding), but a test implementation showed that most real-world problems (relations with up to about 100 attributes each) can be solved with our approach.

For larger-size problems, we additionally identified heuristics that help to reduce the search space in our algorithm, achieving good results even for very large problem sizes. This approach also helps to overcome the problem of "accidental INDs", which are inclusion dependencies that are valid in a database even though they do not express any semantic interrelationship between data objects.

Uses of this technology lie for example in schema integration, an important phase of information integration (Fig. 1.1). As our algorithm discovers database interrelationships, it helps in the identification of data resources that are useful for a variety of purposes, such as the identification of database duplicates, system integration support, or the purpose of identifiying backup data for a view synchronization system like EVE (Chapter 2).

# Part III

# Incremental Maintenance of Schema-Restructuring Views

# Chapter 9

# Introduction and Background

## 9.1 Introduction

Information sources, especially on the Web, are increasingly independent from each other, being designed, administered and maintained by a multitude of autonomous data providers. Nevertheless, it becomes more and more important to integrate data from such sources. Issues in data integration include the heterogeneity of data and query models across different sources, called model heterogeneity [TRV96, HGMN$^{+}$97] and incompatibilities in schematic representations of different sources even when using the same data model, called schema heterogeneity [MIR93, LSS96]. Much work on these problems has dealt with the integration of schematically different sources under the assumption that all "data" is stored in tuples and all "schema" is stored in attribute and relation names. We now relax this assumption and focus on the integration of heterogeneous sources under the assumption that schema elements may express data and vice versa.

One recent promising approach at overcoming such schematic heterogeneity are *schema-restructuring query languages*, such as *SchemaSQL*, an extension of SQL devised by LAKSHMANAN et al. [LSS96, LSS99]. Other proposals include IDL by KRISHNAMURTHY et al. [KLK91], HiLog [CKW89] and MSQL [LAZ+89]. Schema-restructuring query languages, and *SchemaSQL* in particular, support queries over not only data but also schema elements (such as lists of attribute or relation names) in SQL-like queries. Also, they allow sets of values obtained from *data tuples* to be used as *schema* in the output relation. This extension leads to more powerful query languages, effectively achieving a transformation of semantically equivalent but syntactically different schemas [LSS96] into each other.

Previous work on integration used either SQL-views, if the underlying schema agreed with what was needed in the view schema [QW91, NLR98, etc.], or translation programs written in a programming language to reorganize source data [BRU97, HGMN+97]. We propose to use *views* defined in schema-restructuring languages in a way analogous to SQL-views. This makes it possible to include a larger class of information sources into an information system using a query language as the integration mechanism. This concept is much simpler and more flexible than ad-hoc "wrappers" that would have to be written for each data source. It is also possible to use or adapt query optimization techniques for such an architecture.

However, such an integration strategy raises the issue of maintaining schema-restructuring views, which is an open problem. View maintenance in a restructuring view is different from SQL view maintenance, due to the disappearance of the distinction between data and schema, leading to new classes of updates and update transformations. In this part of the dissertation, we present the first incremental maintenance strategy for a schema-restructuring view language, using *SchemaSQL* as an example.

### 9.1.1 Motivating Example

Consider the two relational schemas in Fig. 9.1 that are able to hold the same information and can be mapped into each other using *SchemaSQL* queries. The view query restructures the input relations on the top of the figure representing airlines into attributes of the output relations on the bottom representing destinations. The *arrow*-operator (->) attached to an element in the FROM-clause of a *SchemaSQL*-query allows to query schema elements, giving *SchemaSQL* its meta-data restructuring power. Standing by itself, it refers to "all relation names in that database", while attached to a relation name it means "all attribute names in that relation".

*SchemaSQL* is also able to transform data into schema. For example, *data* from the attribute Destination in the input schema is transformed into *relation names* in the output schema, and vice versa *attribute names* in the input (Business and Economy) are restructured into *data*.

Now consider an update to one of the base relations in our example. Let a tuple $t[\mathsf{Destination}, \mathsf{Business}, \mathsf{Economy}] = (\mathsf{Berlin}, 1400, 610)$ be added to the base table LH (a **data update**). The change to the output would be the addition of a new relation Berlin (a **schema change**) with the same schema as the other two relations. This new relation would contain one

BA

| Destination | Business | Economy |
|---|---|---|
| Paris | 1200 | 600 |
| London | 1100 | 475 |

LH

| Destination | Business | Economy |
|---|---|---|
| Paris | 1220 | 700 |
| London | 1180 | 500 |

⇓

```
create view CITY(Type, AIRLINE) AS
 select PRICETYPE, FLIGHT.PRICETYPE
 from
   -> AIRLINE,
   AIRLINE FLIGHT,
   AIRLINE-> PRICETYPE,
   FLIGHT.Destination CITY
 where PRICETYPE <> 'Destination'
 and FLIGHT.PRICETYPE <= 1100;
```

⇓

LONDON

| Type | BA | LH |
|---|---|---|
| Business | 1100 | *null* |
| Economy | 475 | 500 |

PARIS

| Type | BA | LH |
|---|---|---|
| Economy | 600 | 700 |

Figure 9.1: A Schema-Restructuring Query in *SchemaSQL*.

tuple $t[\mathsf{Type}, \mathsf{BA}, \mathsf{LH}] = (\mathsf{Economy}, null, \mathsf{610})$.

In this example, a data update is transformed into a schema change, but all other combinations are also possible. The effect of the propagation of an update in such a query depends on numerous factors, such as the input schema, the view definition, the set of unique values in the attribute Destination across *all* input relations (city names), and the set of input relations (airline codes). For example, the propagation would also depend on whether other airlines offer a flight to Berlin in the Economy-class, since in that case the desired view relation already exists.

### 9.1.2 Contributions

We propose to use schema-restructuring *query* languages to define *views* over relational sources and we solve several new problems that arise, using *SchemaSQL* as an example. We observe that, due to the possible transformation of "schema" into "data" and vice-versa, we must not only consider data updates (DUs) for *SchemaSQL*, but also schema changes (SCs). A consequence is that, as shown in this work, using the standard approach of generating query expressions that compute some kind of "delta" relation $\Delta$ between the old and the new view after an update is not sufficient, since the schema of $\Delta$ would not be defined. Our algorithm in fact transforms an incoming (schema or data) update into a sequence of schema changes and/or data updates on the view extent.

The contributions of this part are as follows: (1) we identify the new problem of schema-restructuring view maintenance, (2) we give an solution to the problem that is based on a query algebra, (3) we prove this approach correct, (4) we develop a prototype implementation of a query engine and incremental view maintenance system for *SchemaSQL*, and (5) we describe performance showing the improvements of this approach over recomputation.

## 9.2 Background

### 9.2.1 Notation

A *value* is an element of data that is stored in a relation. Examples include strings, numbers, and dates. A *domain* $D$ is a set of *values*.[1] $D_N$ is the special domain of "attribute- and relation names". We implicitly assume that there is a bijective mapping from some domains to $D_N$. This means that the values of some, but not necessarily all, domains can be converted to names and vice versa. For example, the values from a numeric domain cannot be converted easily into attribute names in most database systems.

A *relation* is a 3-tuple $R = (n, S, E)$ with $n \in D_N$ (the relation name), $S = (a_1, a_2, \ldots, a_n) \in (D_N)^n$ (the schema—a tuple of $n$ attribute names) and $E \subseteq \{D_1 \times D_2 \times \ldots \times D_n\}$ (the relation extent, which is a subset of the cross-product of the domains $D_1 \times D_2 \times \ldots \times D_n$). Note that this definition associates exactly one value in $S$ with each domain from which $E$ is constructed (the name of an "attribute").

A *relational tuple* $t \in E$ is an $n$-tuple and is an element of a relation's extent. An operator $t[a_{l_1}, a_{l_2}, \ldots, a_{l_k}]$ returns the projection of $t$ on the attributes named $a_{l_1}, a_{l_2}, \ldots, a_{l_k}$. We also define $t[* \backslash \{a_1, \ldots, a_n\}]$ to be the projection of $t$ onto all its attributes except the ones named $a_1, \ldots, a_n$.

An *attribute* $A_i \subseteq D_i$ is a multiset that is constructed as follows: $A_i = \{t[a_i] \mid t \in E\}$, or short $A_i = E[a_i]$. Then *attribute* $A_i$ has *attribute name* $a_i$. Note that we denote *attributes* by capital letters (as they are sets) and

[1] Throughout this work, we will use capital letters $R$ to denote *(multi)sets* and small letters $a$ to denote elements of sets.

*attribute names* by small letters. We extend this notation for $E[a_1, \ldots, a_k]$ to mean the *projection* of extent $E$ on the attributes $A_1, \ldots, A_k$. For readability, if we refer to attribute $A$ of $R = (n, S, E)$, we actually mean the pair $A = (a, E[a])$, with $a \in S$. The term *prime attribute* refers to an attribute that is a member of any key of $R$ and the term *non-prime attribute* refers to an attribute that is not a member of any key of $R$ (see [Ull89]). The *distinct*-operator $\langle a_i \rangle$ on an attribute $A_i$ in extent $E$ returns the set of distinct values in $A_i$ by removing all duplicates from the multiset $E[a_i]$.

*Functional dependencies* in $R$ are defined as usual (see [Ull89, Chapter 7]), with $X \to A$ defining the attribute $A$ to be functionally dependent on the set of attributes $X$ (i.e., for any $t \in E$, the value of $t[a]$ depends only on $t[x_1, \ldots, x_k]$) . Likewise, we assume the usual definitions of *natural join* $\bowtie$ and *cross product* $\times$.

### 9.2.2 *SchemaSQL*

In relational databases it is possible to store equivalent data in different schemas [Hul86, MIR93] that are incompatible when queried in SQL. However, for information integration purposes it is desirable to combine data from such heterogeneous schemas. *SchemaSQL* is an SQL derivative designed by Lakshmanan et al. [LSS96] which can be used to restructure the schema of a relational database. In [LSS99], Lakshmanan et al. describe an extended algebra and algebra execution strategies to implement a *SchemaSQL* query evaluation system. It extends the standard SQL algebra which uses operators such as $\sigma(R)$, $\pi(R)$, and $R \bowtie S$ by adding four operators named Unite, Fold, Unfold, and Split originally introduced by

Gyssens *et al.* [GLS96] as part of their "Tabular Algebra". Lakshmanan et al. show that any *SchemaSQL* query can be translated into this extended algebra.

### *SchemaSQL* Algebra Operators

As the definitions of the operators in [LSS99] are very brief and could be misinterpreted, we will now give more precise operator definitions in more detail. While we followed Lakshmanan's definitions closely, we have slightly changed the semantics of the Fold/Unfold-operator pair [LSS99] to include an explicit key constraint, which avoids an ambiguity explained below. The original SchemaSQL proposal can be supported as well, with slight changes in the update propagation scheme. Examples for the four operators defined in this section can be found in Fig. 9.2. We will refer to the input relation of each operator as $R$ and to the output relation as $Q$.

**The** Unite**-Operator** is defined on a set of $k$ relations $R^* = \{R_1, \ldots, R_k\}$ with $a_p$ as an argument. We define a set $N^* = \{n_{R_1}, \ldots, n_{R_k}\}$, which is the set of all relation names in $R^*$, in the new domain $D_p$. We further denote by $N_k$ the relation $R(n_k, E, S)$. Then, for each $R_i$, we assume the schema $S_{R_i} = (a_1, \ldots, a_n)$ and the extent $E_{R_i} \subseteq \{D_1 \times \ldots \times D_n\}$. Note that this implies that all $R_i$ have the same schema. The output of the Unite operator is then one relation $Q = \text{UNITE}_{a_p}(R^*)$ with $E_Q \subseteq \{D_1 \times \ldots \times D_n \times D_p\}$ and $S_Q = (a_1, \ldots, a_n, a_p)$ with $E_Q = \bigcup\limits_{n_k \in N^*} (N_k \times \{n_k\})$. Note that $N_k$ denotes the *relation* $N_k$ and $n_k$ its *relation name.* In words, a new relation is constructed by taking the union of all input relations and adding a new attribute $A_p$ whose values are the *relation names* of the input relations. In

BA

| Destination | Business | Economy |
|---|---|---|
| Paris | 1200 | 600 |
| London | 1100 | 475 |

LH

| Destination | Business | Economy |
|---|---|---|
| Paris | 1220 | 700 |
| London | 1180 | 500 |

⇓ UNITE Airline

TMP_REL_0001

| Airline | Destination | Business | Economy |
|---|---|---|---|
| BA | Paris | 1200 | 600 |
| BA | London | 1100 | 475 |
| LH | Paris | 1220 | 700 |
| LH | London | 1180 | 500 |

⇓ FOLD Type, Price,{Business,Economy}

TMP_REL_0002

| Airline | Type | Destination | Price |
|---|---|---|---|
| BA | Business | Paris | 1200 |
| BA | Business | London | 1100 |
| BA | Economy | Paris | 600 |
| BA | Economy | London | 475 |
| LH | Business | Paris | 1220 |
| ... | ... | ... | ... |

⇓ STANDARD-SQL

```
select * from tmp_rel_0002
where price <= 1100;
```

TMP_REL_0003

| Airline | Type | Destination | Price |
|---|---|---|---|
| BA | Business | London | 1100 |
| BA | Economy | Paris | 600 |
| BA | Economy | London | 475 |
| ... | ... | ... | ... |

⇓ UNFOLD Airline, Price

TMP_REL_0004

| Type | Destination | BA | LH |
|---|---|---|---|
| Business | London | 1100 | *null* |
| Economy | Paris | 600 | 700 |
| Economy | London | 475 | 500 |

⇓ SPLIT Destination

LONDON

| Type | BA | LH |
|---|---|---|
| Business | 1100 | *null* |
| Economy | 475 | 500 |

PARIS

| Type | BA | LH |
|---|---|---|
| Economy | 600 | 700 |

Figure 9.2: The Four *SchemaSQL* Operators UNITE, FOLD, UNFOLD, SPLIT.

Fig. 9.2, the UNITE-operator is defined over the set of relations BA, LH and has the attribute name Airline as its argument.

**The** FOLD**-Operator** works on a relation $R = (n_R, E_R, S_R)$ with $E_R \subseteq \{D_1 \times \ldots \times D_n \times \underbrace{D_d \times \ldots \times D_d}_{k \text{ times}}\}$ and $S_R = (a_1, \ldots, a_n, a_{n+1}, \ldots a_{n+k})$ and takes as arguments the *names* of the pivot and data attributes $a_p$ and $a_d$ in its output relation. Note that this definition requires that $k$ attributes of $R$ have to be of the same domain. Furthermore, we require the attribute set $\{a_1, \ldots, a_n\}$ to satisfy a uniqueness constraint in order to avoid ambiguities in the operator (this requirement is not explicit in [LSS99]). With $R$ having $n + k$ attributes, we then define $Q = \text{FOLD}_{a_p,a_d}(R) = (n_Q, E_Q, S_Q)$ with $n_Q = n_R$, $E_Q \subseteq \{D_1 \times \ldots \times D_n \times D_d \times D_p\}$ and $S_Q = (a_1, \ldots, a_n, a_d, a_p)$. We define $A^* = \{a_{n+1}, \ldots a_{n+k}\}$ as a set of values in a new domain $D_p$ where the values are obtained by the above-mentioned conversion of attribute names into data values. Finally, $E = \bigcup_{a_k \in A^*} (R[a_1, \ldots, a_n, a_k] \times \{a_k\})$. In words, the operator takes all data values from the set of related attributes, and sorts them into *one* new attribute $a_d$, introducing another new attribute $a_p$ that holds the former attribute names. Note that, since $a_p$ becomes part of a key for $a_d$, it has to be included in the set $X$ for any functional dependency $X \rightarrow A_d$. To motivate the above uniqueness constraint, note that its violation would require us to introduce multiple tuples in the output relation that differ only in their attribute $a_d$. The semantics of such tuples are not clear in a real-world application. In Fig. 9.2, the FOLD-operator is defined on relation TMP_REL_0001 and has the arguments $a_p = \text{Type}, a_d = \text{Price}, A^* = \{\text{Business,Economy}\}$.

**The** Unfold**-Operator** is the inverse of Fold. Defined on a relation $R = (n_R, E_R, S_R)$ with the extent $E_R \subseteq \{D_1 \times \ldots \times D_n \times D_p \times D_d\}$ and the schema $S_R = (a_1, \ldots, a_n, a_p, a_d)$, it takes two arguments $a_p, a_d$ which are attribute names from $S_R$. To simplify the notation and without loss of generality, we reorder the attributes in $R$ (by exchanging the indices on both $S_R$ and $E_R$ accordingly), such that $A_p$ and $A_d$ become the last two attributes in $R$. We call $A_p$ the *pivot attribute* and $A_d$ the *data attribute*. Let $R$ have $n+2$ attributes. In addition to the original *SchemaSQL*, we impose two conditions on functional dependencies in $R$: $(X \rightarrow Y) \Rightarrow A_d \notin X$ and $\exists(X \rightarrow A_d)$ with $A_p \in X$. That is, $A_d$ must be non-prime, $A_p$ must be prime and $A_d$ must depend on a key containing $A_p$ (this is not explicity stated in [LSS99]). We also set $A^* = R\langle A_p \rangle$ (the set of distinct values in $A_p$), $k = |A^*|$ and impose a total order on $A^*$ to assign an index $1 \leq i \leq k$ to each of its elements ($A^* = \{a_1^*, \ldots, a_k^*\}$).

Then we have $Q = \text{Unfold}_{a_p,a_d}(R) = (n_Q, S_Q, E_Q)$ with $N_Q = n_R$, $E_Q \subseteq \{D_1 \times \ldots \times D_n \times \underbrace{D_d \times \ldots \times D_d}_{k \text{ times}}\}$ and $S_Q = (a_1, \ldots, a_n, a_1^*, \ldots, a_k^*)$. The extent is constructed by $E_Q = E_R[a_1, \ldots, a_n] \bowtie E_1 \bowtie \ldots \bowtie E_k$ with $E_i = \{t[a_1, \ldots, a_n, a_d] \mid t \in E_R \ \wedge \ t[a_p] = a_i^*\}$.

In words, the schema of $Q$ consists of all attributes in $R$ except the data and pivot attribute, plus one attribute for each distinct data value in the pivot attribute. Each tuple $t'$ in $Q$ is constructed by taking a tuple $t$ in $R$ and filling each new attribute $A_i$ with the value from attribute $A_d$ in a tuple from $R$ that has the *name* $a_i$ as *value* in $A_p$ (assuming an implicit conversion between names and values as required above). The new attributes all have the domain $D_d$ of the old attribute $A_d$.

In Fig. 9.2, the UNFOLD-operator with arguments $a_p$ = Airline and $a_d$ = Price is defined over relation TMP_REL_0003. The operator produces output by taking tuples from TMP_REL_0003, and filling the attributes representing airlines with values from the data attribute $a_d$ = Price in the relation TMP_REL_0003, matching attribute names in the output relation with the values of the pivot attribute Airline in the input relation.

**The SPLIT-Operator** is the inverse of the UNITE-operator. It transforms a single relation $R = (n_R, E_R, S_R)$ with $E_R \subseteq \{D_1 \times \ldots \times D_n \times D_p\}$ and $S_R = (a_1, \ldots, a_n, a_p)$ into a *set* of $k$ relations with the same schema. It takes as argument the name of the pivot attribute $a_p$ which we assume to be the last in $R$. We require that $A_p$ does not have NULL-values, i.e., $\forall x \in A_p : x \neq \perp$, to avoid having attributes without names. One could allow NULL-values with slight changes in the semantics (and update propagation algorithm), however the language appears cleanest when including this requirement. The output of SPLIT is a set of relations $Q^* = \text{SPLIT}_{a_p}(R) = \{Q_1, \ldots, Q_k\}$ with $A^* = R\langle A_p \rangle$ and $k = |A^*|$. We will refer to the ordered elements of $A^*$ as in the UNFOLD-case, i.e., $A^* = \{a_1^*, \ldots, a_k^*\}$. For each output relation $Q_i$, we have: $n_{Q_i} = a_i^*$, $E_{Q_i} \subseteq \{D_1 \times \ldots \times D_n\}$, $S_{Q_i} = (a_1, \ldots, a_n)$, and $E_{Q_i} = \{t[a_1, \ldots, a_n] \mid t \in R \wedge t[a_p] = a_i^*\}$. In words, we break $R$ down into $k$ relations of the same schema, with the new relation names the $k$ distinct values from $R$'s attribute $A_p$. In Fig. 9.2, the SPLIT-operator is defined over relation TMP_REL_0004, takes as its only argument $a_p$ = Destination, and produces 2 tables names LONDON and PARIS.

### *SchemaSQL* Query Evaluation

Similar to traditional SQL evaluation, [LSS99] proposes a strategy for query evaluation in *SchemaSQL* that first constructs and then processes an algebra query tree, leading to an efficient implementation of *SchemaSQL* query evaluation over an SQL database system. In order to evaluate a *SchemaSQL* query, an algebra expression using standard relational algebra plus the four operators introduced above is constructed. This expression is of the following standard form [LSS99] :

$$\begin{aligned} V &= \textsc{Split}_a(\textsc{Unfold}_{b,c}(\pi_D(\sigma_{cond}(\textsc{Fold}_{e_1,f_1,G_1}(\textsc{Unite}_h(R_1)) \times \ldots \\ &\quad \ldots \times \textsc{Fold}_{e_m,f_m,G_m}(\textsc{Unite}_h(R_m)))))) \end{aligned} \tag{9.1}$$

with attribute names $a, b, c, e_i, f_i, h_i$, the sets of attribute names $D$ and $G_i$, and selection predicates *cond* determined by the query. Any of the four *SchemaSQL* operators may not be needed for a particular query and would then be omitted from the expression. $R_1 \ldots R_m$ are base relations, or, in the case that the expression contains a UNITE-operator, sets of relations with equal schema.

The algebraic expression for our running example (Fig. 9.1) is:

$$\begin{aligned} V &= \textsc{Split}_{\mathsf{Destination}}(\textsc{Unfold}_{\mathsf{Airline,Price}}(\sigma_{\mathsf{Price}<1100} \\ &\quad (\textsc{Fold}_{\mathsf{Type,\ Price,\ \{Business,Economy\}}}(\textsc{Unite}_{\mathsf{Airline}}(\mathsf{BA,LH}))))) \end{aligned} \tag{9.2}$$

This algebraic expression is then used to construct an algebra tree (Fig. 9.3) whose nodes are any of the four *SchemaSQL* operators or a “Standard-SQL”-

operator (including the $\pi$, $\sigma$, and $\times$-operators of the algebra expression) with standard relations "traveling" along its edges. The query is then evaluated by traversing the algebra tree and executing a query processing strategy for each operator, analogous to traditional SQL query evaluation.

Note that the query tree could include $\times$-operators (which do not exist in our example), but that the order of UNITE, FOLD, UNFOLD, SPLIT (if they exist) is fixed by the template in Equation 9.1. The UNITE operator takes a number of relations of the same schema as an input, while the SPLIT-operator produces as output a set of relations of the same schema. Note that the algebra tree in Fig. 9.3 is very simple. In more complex queries, the tree could "fork" at the *Standard-SQL-node*, and several smaller "flattening" trees using UNITE- and FOLD-operators could occur. In that case, and also in the case of standard relational joins, the *Standard-SQL-node* would itself contain a more complex algebra tree containing simple SQL algebra nodes.

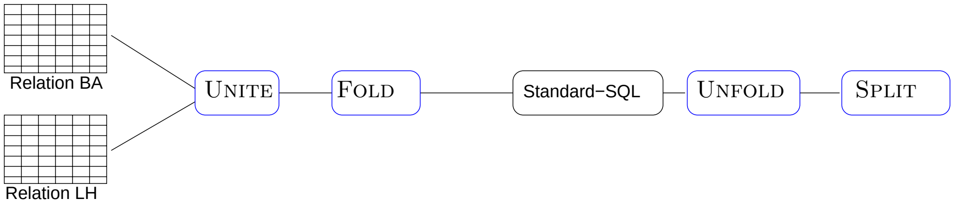


Figure 9.3: The Algebra Tree for the Example in Fig.9.1

# Chapter 10

# Propagation of Updates in a *SchemaSQL* View

## 10.1 The *SchemaSQL* Update Propagation Strategy

In this section, we introduce the update propagation strategy for *SchemaSQL* by defining updates and then describing the propagation of updates through the operators of the *SchemaSQL* algebra tree.

### 10.1.1 Classes of Updates and Transformations

The updates that can be propagated through views in *SchemaSQL* can be grouped into two categories: *Schema Changes (SC)* and *Data Updates (DU)*. Schema changes that we consider are: *add-relation*$(n, S)$, *delete-relation*$(n)$, *rename-relation*$(n, n')$ with relation names $n, n'$ and schema $S$ as introduced in Section 9.2.1 as well as *add-attribute*$(r, a)$, *delete-attribute*$(r, a)$, *rename-attribute*$(r, a, a')$ with $r$ the name of the relation $R$ that the attribute named $a$ belongs to, $a'$ the new attribute name in the rename-case, and the notation otherwise as above. Data updates are any changes affecting a tuple (and not the schema of the relation), i.e., *add-tuple(r,t), delete-tuple(r,t), update-tuple(r,t,t'))*, with $t$ and $t'$ tuples in relation $R$ with name $r$. Note that we consider *update-tuple* as a basic update type, instead of breaking it down into a *delete-tuple* and an *add-tuple*. An *update-tuple* update consists of two tuples, one representing an existing tuple in $R$ and the other representing the values of that tuple after the update. This allows to keep relational integrity constraints valid that would otherwise be violated temporarily.

### 10.1.2 *SchemaSQL* Update Propagation vs. Relational View Maintenance

Update propagation in *SchemaSQL*-views, as in any other view environment, consists in recording updates that occur in the input data and translating them into updates to the view extent. In incremental view maintenance of SQL views [QW91, GL95, MKK97], many update propagation mechanisms have been proposed. Their common feature is that the new view extent is obtained by first computing *extent differences* between the old view $V$ and the new view $V'$ and then adding them to or subtracting them from the view, i.e., $V' = (V \backslash \nabla V) \cup \Delta V$, with $\nabla V$ denoting some set of tuples computed from the base relations that needs to be deleted from the view and $\Delta V$ some set that needs to be added to the view [QW91].

In *SchemaSQL*, this mechanism leads to difficulties. If *SchemaSQL* views must propagate both schema *and* data updates, the schema of $\Delta V$ or $\nabla V$ does not necessarily agree with the schema of the output relation $V$. But even when considering only data updates to the base relations, the new view $V'$ may have a different schema than $V$. That means the concept of set difference between the tuples of $V'$ and $V$ is not even meaningful. Thus, we must find a way to incorporate the concept of schema changes. For this purpose, we now introduce a data structure $\partial$ which represents an update sequence of $n$ data updates DU and schema changes SC.

**Definition 10.1 (defined update)** *Assume two sets DU and SC which represent all possible data updates and schema changes, respectively. A change $c \in DU \cup SC$ is **defined on a given relation** $R$ if one of the following conditions holds:*

- *if $c \in DU$, the schema of the tuple added or deleted must be equal to the schema of R.*
- *if $c \in SC$, the object c is applied to (an attribute or relation) must exist (for delete- and update-changes) or must not exist (for add-changes) in R.*

**Definition 10.2 (valid update sequence)** *A sequence of updates $(c_1, \ldots, c_n)$ with $c_i \in DU \cup SC$, denoted by $\partial R$, is called **valid for** $R$ if for all $i$ $(1 < i \leq n)$, $c_i$ is defined on the relation $R^{(i-1)}$ that was obtained by applying $c_1, \ldots, c_{i-1}$ to R.*

For simplicity, we will also use the notation $\partial\omega$ to refer to a valid update sequence to the output table of an algebra operator $\omega$. Note that these definitions naturally extend to views, since views can also be seen as relational schemas. For an example, consider propagation of the update `add-tuple('Berlin',1400,610) to` LH in Fig. 10.7 (p. 210). Having the value Berlin in the update tuple will lead to the addition of a new relation BERLIN in the output schema of the view—forming a sequence $\partial V$ which contains both a schema change and a data update:

$$\begin{aligned} \partial V \quad = ( \quad & \textit{add-relation}(\mathsf{BERLIN}, (\mathsf{Type,Destination,BA,LH})), \\ & \textit{add-tuple}(\mathsf{BERLIN}, (\text{'}\mathsf{Economy}\text{'},\mathsf{null},\mathsf{610})) \\ ) & \end{aligned}$$

The *add-relation*-update is valid since the relation BERLIN did not exist in the output schema before, and the *add-tuple*-update is valid since its schema agrees with the schema of relation BERLIN defined by the previous update.

### 10.1.3 Overall Propagation Strategy

Given an *update sequence* $\partial V$ implemented by a `List` data structure, our update propagation strategy works according to the algorithm in Fig. 10.1. Each node in the algebra tree has knowledge about the operator it represents. This operator is able to accept *one* input update and generate a sequence of updates as output. Each (leaf node) operator can also recognize whether it is affected by an update (by comparing the relation(s) on which the update is defined with its own input relation(s)). If it is not affected, it simply returns an empty update sequence.

After all the updates for the children of a node $n$ are computed and collected in a list (variable $s$ in the algorithm in Fig. 10.1), they are propagated one-by-one through $n$. Each output update generated by the operator of $n$ when processing an input update will be placed into one update sequence, all of which are concatenated into the final return sequence $r$ (see Fig. 10.1, $\leftarrow$ is the assignment operator).

```
function $propagateUpdate$(Node $n$, Update $u$)
    List $r \leftarrow \emptyset,\ s \leftarrow \emptyset$
    if ($n$ is leaf)
        if ($n.operator$ is affected by $u$)
            $r.append(n.operator.operatorPropagate(u))$
    else
        for($all\ children\ c_i\ of\ n$)
                //s will change exactly once, see text
            $s.append(propagateUpdate(c_i, u))$
        for($all\ updates\ u_i\ in\ s$)
            $r.append(n.operator.operatorPropagate(u_i))$
    return $r$
```

Figure 10.1: The *SchemaSQL* View Maintenance Algorithm

The algorithm performs a postorder traversal of the algebra tree. This ensures that each operator processes input updates after all its children have already computed their output[1]. At each node $n$, an incoming update is translated into an output sequence $\partial n$ of length greater than or equal to 0 which is then propagated to $n$'s parent node. Since the algebra tree is connected and cycle-free (not considering joins of relations with themselves) all nodes will be visited exactly once. Also note that since updates occur only in one leaf at a time, only exactly one child of any node will have a non-empty update sequence to be propagated. That is, the first **for**-loop will

[1] We are not considering concurrent updates in this work.

find a non-empty addition to $s$ only once per function call. After all nodes have been visited, the output of the algorithm will be an update sequence $\partial$V to the view $V$ that we will prove to have an effect on $V$ equivalent to recomputation.

### 10.1.4 Propagation of Updates through Individual *SchemaSQL* Operators

Since update propagation in our algorithm occurs at each operator in the algebra tree, we have to design a propagation strategy for each type of operator.

#### Propagation of Schema Changes through SQL Algebra Operators

The propagation of updates through standard SQL algebra nodes is simple. Deriving the update propagation for data updates is discussed in the literature on view maintenance [QW91, GL95]. It remains to define update propagation for selection, projection, and cross-product operators under schema changes [2].

The propagation of updates through standard SQL algebra nodes is simple. Deriving the update propagation for data updates is discussed in the literature on view maintenance [BLT86, QW91, GL95]. It remains to define update propagation for selection, projection, and cross-product operators under schema changes. As with all the other operators in this work, we

[2] these are the only operators necessary for the types of queries discussed in this dissertation

denote the input relation by $R$ (and its name by $r$) and the output relation by $Q$ (named $q$).

**Relation-Updates:** For all standard SQL algebra operators, updates of type *add-relation(n,S)* is not propagated since it will not affect existing output. *Delete-relation*$(r)$ makes the operator's output invalid (i.e., is propagated into a *delete-relation*$(q)$ of its output relation), and *rename-relation*$(n, n')$ is not propagated since the name of an input relation is not relevant to a standard SQL-operator.

**Attribute-Updates:**

1. Selection operator $\sigma_{cond}$ with *cond* a list of conditions on the attributes in this operator's relation $R$.

   All attribute updates are propagated by changing the parameter $r$ (which represents the name of the input relation) to the name of the output relation $q$. If a predicate in *cond* is defined over a renamed attribute, it is changed accordingly. If it is defined over a deleted attribute, the view becomes invalid.

2. Projection operator $\pi_{\bar{A}}$ with a set of attributes $\bar{A}$ as projection list

   *add-attribute*$(r, a)$ will not be propagated ($\partial\omega = \emptyset$) as $a$ will not be in $\bar{A}$. *Delete-attribute*$(r, a)$ will be propagated as *delete-attribute*$(q, a)$ if $a \in \bar{A}$. *Rename-attribute*$(r, a, a')$ will be propagated as *rename-attribute*$(q, a, a')$ if $a \in \bar{A}$, in which case the projection list changes: $\bar{A}' \leftarrow \bar{A} \backslash a \cup a'$. Thus, the changed attribute name will be available on the output.

3. Cross-Product Operator ×

   The cross-product operator simply propagates any incoming update (on an attribute) into an output update by changing the relation name parameter $r$ to $q$. Incremental view maintenance is performed as necessary, in a manner described in the literature [ZGMW96, AESY97, ZR99].

### ***SchemaSQL*** **Operators**

In Figs. 10.2–10.5, we give the update propagation tables for the four *SchemaSQL* operators. For the notation and meaning of variables and constants, please refer to Section 9.2.1. In order to avoid repetitions in the notation, the cases for each update type are to be read in an "if-else"-manner, i.e., the first case that matches a given update will be used for the update generation (and no other). Also, NULL-values are like other data values, except where stated otherwise.

The tables show the propagation of each possible input change, broken down into several cases as necessary (in the second column). The second column also shows any auxiliary variables needed for propagation and their computation. For example, if an attribute is deleted from the input of a Split-operator, the propagation depends on whether the deleted attribute is the pivot attribute $A_p$ or another attribute. The last column in all tables shows the output of the operator, i.e., the update sequence $\partial\omega$ generated as the propagation of an input update. This output is shown as an SQL `insert`-, `delete`-, or `update`-statement (in `typewriter font`) or as a schema change (in *italics*). Note that whenever our propagation tables give an SQL-statement as the output of an operator $\omega$, the exact list of updates $\partial\omega$ can be derived simply by applying this SQL-statement to the output relation $R_\omega$. In the case of most SQL-`update` and `delete`-statements, the output relation of an operator $\omega$ is needed to translate a SQL-statement into an update sequence $\partial\omega$. This situation is similar to projection-operators in SQL, where the removal of duplicate output tuples can only be accomplished if the output of the operator is known. Otherwise, each operator $\omega$ in the tree requires as input for the propagation of any update only the input update itself, its own parameters, and the set $A^*$ determined according to Section 9.2.2.

Inspection of the update propagation tables shows several properties of our algorithm. For example, the view becomes invalid under some schema changes or data updates, mainly if an attribute or relation that was necessary to determine the output schema of the operator is deleted (e.g., when deleting the pivot or data attribute in UNFOLD). In the case of *rename*-schema changes (e.g., under *rename-relation* in FOLD), some operators change their parameters. Those are simple renames that do not affect operators otherwise. In those cases we denote renaming by $\Rightarrow$. The operator will produce a zero-element output sequence.

### Formalization of the Propagation of Updates

A formalization of the propagation of updates is extensive and lacks the conciseness of the propagation tables given in this section. Therefore, we will only give an example of how such a definition could be accomplished. We will consider the propagation of *add-tuple* through UNFOLD (Fig. 10.6):

| Input Change | Conditions | Propagation |
|---|---|---|
| *add-tuple* $(r, t)$ | $t[a_1, \ldots, a_n, a_p] \in R$ | invalid view (key violation) |
| | $t[a_p] \in A^*$<br>$t[a_1, \ldots, a_n] \in R$ | **update** Q **set** [$t[a_p]$] = $t[a_d]$<br>**where** $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$ |
| | $t[a_p] \in A^*$<br>$t[a_1, \ldots, a_n] \notin R$ | **insert into** Q ($a_1, \ldots, a_n, a_p$)<br>**values** ($a_1, \ldots, a_n, a_d$) |
| | $t[a_p] \notin A^*$<br>$t[a_1, \ldots, a_n] \in R$ | *add-attribute($q$,$t[a_p]$)*,<br>**update** Q **set** [$t[a_p]$] = $t[a_d]$<br>**where** $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$ |
| | $t[a_p] \notin A^*$<br>$t[a_1, \ldots, a_n] \notin R$ | *add-attribute($q$,$t[a_p]$)*,<br>**insert into** Q ($a_1, \ldots, a_n, a_p$)<br>**values** ($a_1, \ldots, a_n, a_d$) |
| *delete-tuple* $(r, t)$ | $t[a_p]$ exists in $R[a_p]$ exactly once | *delete-attribute($q$,$t[a_p]$)* |
| | $t[a_p]$ exists in $R[a_p]$ more than once | **update** Q **set** [$t[a_p]$] = NULL<br>**where** $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$ [3] |
| *update-tuple* $(r, t, t')$ | $t[a_1, \ldots, a_n, a_p] = t'[a_1, \ldots, a_n, a_p]$ | **update** Q **set** [$t[a_p]$] = $t[a_d]$<br>**where** $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$ |
| | $t[a_1, \ldots, a_n, a_p] \neq t'[a_1, \ldots, a_n, a_p]$ | break down into (*delete-tuple*, *add-tuple*) |
| *add-attribute*$(r, a)$ | | *add-attribute*$(q, a)$ |
| *delete-attribute*$(r, a)$ | $a \in \{A_d, A_p\}$ | invalid view |
| | $a \notin \{A_d, A_p\}$ | *delete-attribute*$(q, a)$ |
| *rename-attribute*$(r, a, a')$ | $a = A_d$ | $\text{UNFOLD}_{a_p,a} \Longrightarrow \text{UNFOLD}_{a_p,a'}$ |
| | $a = A_p$ | $\text{UNFOLD}_{a,a_d}(R) \Longrightarrow \text{UNFOLD}_{a',a_d}(R)$ |
| | $a \notin \{A_d, A_p\}$ | *rename-attribute*$(q, a, a')$ |
| *delete-relation*$(r)$ | | *delete-relation*$(q)$ |
| *rename-relation*$(n, n')$ | | $\text{UNFOLD}_{a_p,a_d}(N) \Longrightarrow \text{UNFOLD}_{a_p,a_d}(N')$<br>(renaming the input relation) |

[3] if this update leads to a tuple with all NULL-values, the tuple must be deleted.

Figure 10.2: Propagation Rules for Q=$\text{UNFOLD}_{a_p,a_d}(R)$

| Input Change | Conditions and Variable Binding | Propagation |
|---|---|---|
| *add-tuple* $(r, t)$ | $(A^* = \{a_1^*, \ldots, a_k^*\}$,$k \leftarrow \lvert A^* \rvert)$ | for $i$ := $1$..$k$<br>insert into Q<br>values $(a_1, \ldots, a_n$,$a_i^*$,$t[a_i^*])$ |
| *delete-tuple* $(r, t)$ | | delete from Q<br>where $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$ |
| *update-tuple* $(r, t, t')$ | $A \in A^*$; set $t[a]$ to a value $c$ | update Q set $a_d = c$<br>where $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$ and $a_p = a$ |
| | $A \notin A^*$; set $t[a]$ from a value $b$ to a value $c$ | update Q set $a = c$ where $a = b$ |
| *add-attribute*$(r, a)$ | $A \in A^{*4}$ | foreach tuple $u \in R$<br>insert into Q $(a_1, \ldots, a_n, a_p, a_d)$<br>values $(u[a_1, \ldots, a_n], a$,NULL) |
| | $A \notin A^*$ | *add-attribute*$(q, a)$ |
| *delete-attribute*$(r, a)$ | $A \in A^*$ | delete from Q where $a_p = a$ |
| | $A \notin A^*$ | *delete-attribute*$(q, a)$ |
| *rename-attribute*$(r, a, a')$ | $A \in A^*$ | update Q set $a_p = a'$ where $a_p = a$ |
| | $A \notin A^*$ | *rename-attribute*$(q, a, a')$ |
| *delete-relation*$(r)$ | | *delete-relation*$(q)$ |
| *rename-relation*$(n, n')$ | | $\textsc{Fold}_{a_p,a_d}(N) \Longrightarrow \textsc{Fold}_{a_p,a_d}(N')$ |

[4] Note that the decision whether a *new* attribute should be a member of $a_1, \ldots, a_n$ can only be made by evaluating the view query.

Figure 10.3: Propagation Rules for Q=$\textsc{Fold}_{a_p,a_d,A^*}(R)$

| Input Change | Conditions | Propagation |
|---|---|---|
| *add-tuple* $(r, t)$ | $t[a_p] \notin A^*$ | `add-relation` $[t[a_p]]$<br>`with schema` $(S_R \backslash R.A_p)$`;`<br>`insert into` $[t[a_p]]$<br>`values` $(t[a_1, \ldots, a_n])$ |
| | $t[a_p] \in A^*$ | `insert into` $[t[a_p]]$<br>`values` $(t[a_1, \ldots, a_n])$ |
| *delete-tuple* $(r, t)$ | $t[a_p]$ exists in $R[a_p]$ exactly once | `delete-relation` $[t[a_p]]$ |
| | $t[a_p]$ exists in $R[a_p]$ more than once | `delete from` $[t[a_p]]$<br>`where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$ |
| *update-tuple* $(r, t, t')$ | $t[a_1, \ldots, a_n, a_p] = t'[a_1, \ldots, a_n, a_p]$ | `update` $[t[a_p]]$ `set` $[a_d]$ `=` $t[a_d]$<br>`where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$ |
| | $t[a_1, \ldots, a_n, a_p] \neq t'[a_1, \ldots, a_n, a_p]$ | break down into (*delete-tuple*,*add-tuple*) |
| *add-attribute*$(r, a)$ | | $\forall q \in \{q_1 \ldots q_n\}$ : *add-attribute*$(q, a)$ |
| *delete-attribute*$(r, a)$ | $a = A_p$ | invalid view |
| | $a \neq A_p$ | $\forall q \in \{q_1 \ldots q_n\}$ : *delete-attribute*$(q, a)$[5] |
| *rename-attribute*$(r, a, a')$ | $a = A_p$ | $\textsc{Split}_a(R) \Longrightarrow \textsc{Split}_{a'}(R)$ |
| | $a \neq A_p$ | $\forall q \in \{q_1 \ldots q_n\}$ : *rename-attribute*$(q, a, a')$ |
| *delete-relation*$(r)$ | | $\forall q \in \{q_1 \ldots q_n\}$ : *delete-relation*$(q)$ |
| *rename-relation*$(n, n')$ | | $\textsc{Split}_{a_p}(N) \Longrightarrow \textsc{Split}_{a_p}(N')$ |

[5] If this update leads to a tuple with all NULL-values in an output relation, the tuple must be deleted.

Figure 10.4: Propagation Rules for Q=$\textsc{Split}_{a_p}(R)$

| Input Change | Conditions and Variable Bindings | Propagation |
|---|---|---|
| *add-tuple* $(r_x, t)$ | | `insert into Q` $(a_1, \ldots, a_n, a_p)$<br>`values` $(t[a_1, \ldots, a_n], r_x)$ |
| *delete-tuple* $(r_x, t)$ | | `delete from Q`<br>`where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$ `and` $a_p = r_x$ |
| *update-tuple* $(r_x, t, t')$ | $A = A_d$; set $t[a]$ to a value $c$ | `update Q set` $a = c$<br>`where` $a_1, \ldots, a_n = t[a_1, \ldots, a_n]$ `and` $a_p = r_x$ |
| | $A \neq A_d$; set $t[a]$ from a value $b$ to a value $c$ | `update Q set` $a = c$<br>`where` $a = b$ `and` $a_p = r_x$ |
| *add-attribute*$(r, a)$ | add simultaneously to all $R_i$ | *add-attribute*$(q, a)$ |
| | otherwise | invalid view |
| *delete-attribute*$(r, a)$ | delete simultaneously from all $R_i$ | *delete-attribute*$(q, a)$ |
| | otherwise | invalid view |
| *rename-attribute*$(r, a, a')$ | rename simultaneously in all $R_i$ | *rename-attribute*$(q, a, a')$ |
| | otherwise | invalid view |
| *add-relation*$(r_x, S)$ | | no change (until first *add-tuple* to $R_x$) |
| *delete-relation*$(r_x)$ | | `delete from Q where` $a_p = r_x$ |
| *rename-relation*$(n, n')$ | | $\text{UNITE}_{a_p}(\{R_1, \ldots, N, \ldots, R_n\}) \Longrightarrow$<br>$\text{UNITE}_{a_p}(\{R_1, \ldots, N', \ldots, R_n\})$<br>`update Q set` $a_p = n'$<br>`where` $a_p = n$ |

Figure 10.5: Propagation Rules for Q=$\text{UNITE}_{a_p}(R_1, R_2, \ldots, R_n)$[6]

[6] Note that $r_x$ is the name of Relation $R_x$, which is one of the $n$ relations of equal schema that are united by the UNITE-operator.

Using the notation from Section 9.2.2, assume a relation $R = (N_R, S_R, E_R)$ with $n$ attributes that is the input for an operator $Q = \textsc{Unfold}_{a_p,a_d}(R)$ producing an output relation $Q = (N_Q, S_Q, E_Q)$ with $n - 2 + k$ attributes and an update to $R$, denoted by $\Delta_R = t[a_1, \ldots, a_n, a_p, a_d] = (x_1, \ldots x_n, x_p, x_d)$. Let $A^*$ be a set of the $k$ distinct values in $A_p$ (the pivot attribute, see the definition of $\textsc{Unfold}$ in Sec. 9.2.2).

The propagation of this update is shown in Fig. 10.6.

The structure $(E_Q \setminus T_1)^{\oplus}$ in the figure[7] is constructed by *adding an attribute* to $E_Q \setminus T_1$, i.e., $(E_Q \setminus T_1 \subseteq D_1 \times \ldots \times D_{n+k}) \Rightarrow ((E_Q \setminus T_1)^{\oplus} \subseteq D_1 \times \ldots \times D_{n+k} \times D_d)$ with all data values in this new attribute set to NULL ($\perp$). Note that the output relation becomes invalid iff an update is inserted into the input relation that agrees in $a_1, \ldots, a_n, a_p$ with an existing tuple (similar to a key violation).

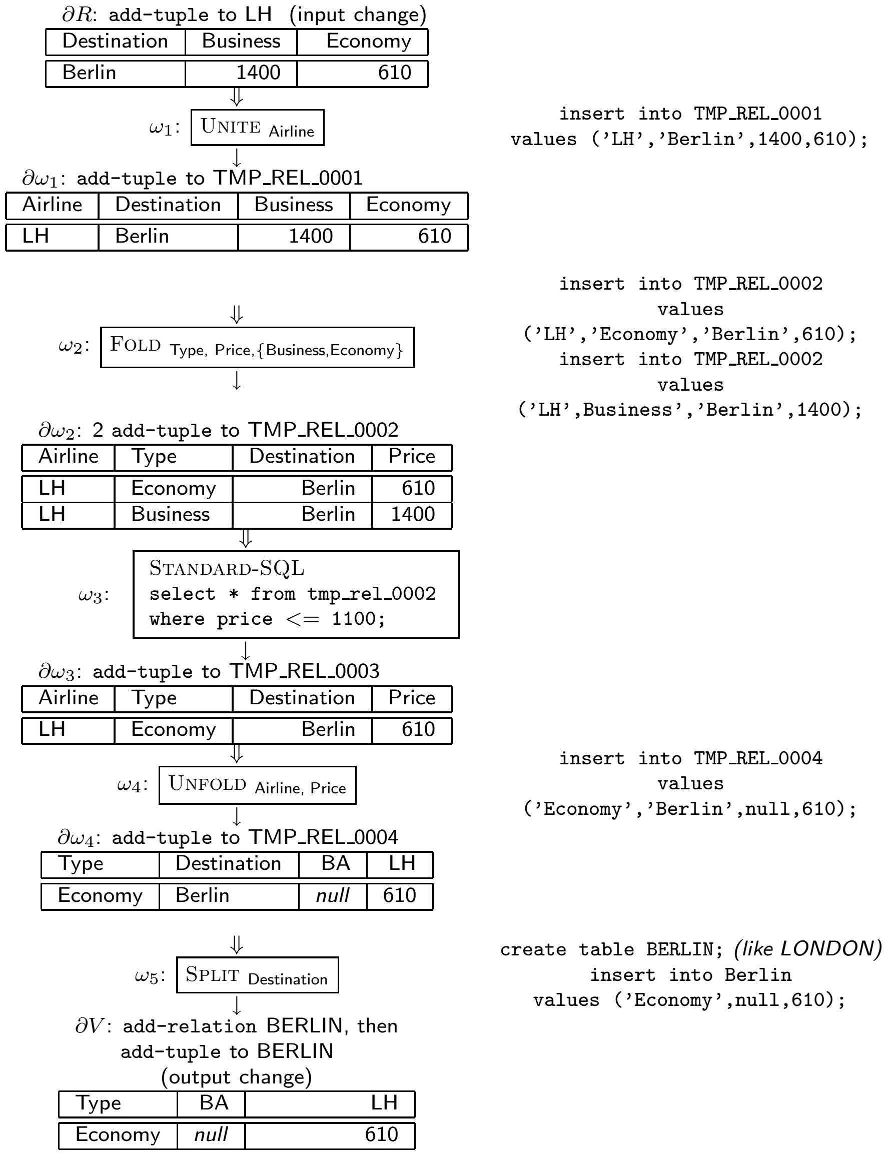
### 10.1.5 Update Propagation Example

Fig. 10.7 gives an example for an update that is propagated through the *SchemaSQL*-algebra-tree in Fig. 9.2 (see also Fig. 10.12). All updates are computed by means of the propagation tables in the previous section. The operators appear in boxes with their output attached to the rightof each box (SQL-statements according to our update tables, Figs. 10.2–10.5). The actual tuples added by these SQL-statements are shown in tabular form. The sending of updates to another operator is denoted by double arrows ($\Downarrow$), while single arrows ($\downarrow$) symbolize the transformation of SQL-statements into updates. We are propagating an *add-tuple*-update to base relation $\mathsf{LH}$.

[7] We use the symbol \ to denote set-difference.

$$
\begin{array}{rcl}
E'_R & \leftarrow & E_R \cup \Delta_R \qquad \textit{a tuple-add} \\
 & \Downarrow & \\
\nu & \leftarrow & \Delta_R[a_p] \qquad \textit{the pivot-value of the new tuple} \\
T_0 & \leftarrow & \{t \in E_R \mid t[a_1,\ldots,a_n,a_p] = \Delta_R[a_1,\ldots,a_n,a_p]\} \qquad \textit{find out if added tuple exists in } E_R \\
S'_Q & \leftarrow & \begin{cases} \emptyset & \text{if } T_0 \neq \emptyset \qquad \textit{key violation} \\ S_Q & \text{if } \nu \in A^* \qquad \textit{note that } k = |A^*| \\ (a_1,\ldots,a_n,a^*_1,\ldots,a^*_k,a_p) & \text{otherwise} \qquad \textit{schema change if necessary} \end{cases} \\
T_1 & \leftarrow & \{t \in E_Q \mid t[a_1,\ldots,a_n] = \Delta_R[a_1,\ldots,a_n]\} \qquad \textit{the matching tuples in the output relation} \\
T_2 & \leftarrow & \{t \in T_1 \mid t[\nu] \leftarrow \Delta_R[a_d]\} \qquad \textit{set pivot attribute to value in data attribute} \\
T_3 & \leftarrow & \begin{cases} \{t[a_1,\ldots,a_n,a^*_1,\ldots,a^*_k,a_p] \mid t[a_1,\ldots,a_n,a_p] \leftarrow \Delta_R[a_1,\ldots,a_n,a_d];\ t[a^*_i] \leftarrow \bot\} \\ \hfill \text{if } T_1 = \emptyset \\ \qquad \textit{if new row in output table, construct new tuple, fill unused attributes with NULL} \\ \\ \{t[a_1,\ldots,a_n,a^*_1,\ldots,a^*_k,a_p] \mid t[a_1,\ldots,a_n,a^*_1,\ldots,a^*_k] \in T_1;\ t[a_p] \leftarrow \Delta_R[a_d]\} \\ \hfill \text{otherwise} \\ \hfill \textit{just set appropriate value} \end{cases} \\
E'_Q & \leftarrow & \begin{cases} \emptyset & \text{if } T_0 \neq \emptyset \\ (E_Q \setminus T_1) \cup T_2 & \text{if } \nu \in A^* \qquad \textit{no schema change} \\ (E_Q \setminus T_1)^{\oplus} \cup T_3 & \text{otherwise} \qquad \textit{an output schema change} \end{cases}
\end{array}
$$

Figure 10.6: Propagation of *add-tuple($\Delta_R$)* through an UNFOLD-Operator



$\partial R$: add-tuple to LH (input change)

| Destination | Business | Economy |
|---|---|---|
| Berlin | 1400 | 610 |

⇓
$\omega_1$: UNITE Airline
↓

```
insert into TMP_REL_0001
values (’LH’,’Berlin’,1400,610);
```

$\partial\omega_1$: add-tuple to TMP_REL_0001

| Airline | Destination | Business | Economy |
|---|---|---|---|
| LH | Berlin | 1400 | 610 |

⇓
$\omega_2$: FOLD Type, Price, {Business,Economy}
↓

```
insert into TMP_REL_0002
values
(’LH’,’Economy’,’Berlin’,610);
insert into TMP_REL_0002
values
(’LH’,Business’,’Berlin’,1400);
```

$\partial\omega_2$: 2 add-tuple to TMP_REL_0002

| Airline | Type | Destination | Price |
|---|---|---|---|
| LH | Economy | Berlin | 610 |
| LH | Business | Berlin | 1400 |

```
select * from tmp_rel_0002
where price <= 1100;
```

$\partial\omega_3$: add-tuple to TMP_REL_0003

| Airline | Type | Destination | Price |
|---|---|---|---|
| LH | Economy | Berlin | 610 |

```
insert into TMP_REL_0004
values
(’Economy’,’Berlin’,null,610);
```

$\partial\omega_4$: add-tuple to TMP_REL_0004

| Type | Destination | BA | LH |
|---|---|---|---|
| Economy | Berlin | *null* | 610 |

```
create table BERLIN; (like LONDON)
insert into Berlin
values (’Economy’,null,610);
```

$\partial V$: add-relation BERLIN, then add-tuple to BERLIN (output change)

| Type | BA | LH |
|---|---|---|
| Economy | *null* | 610 |

**Legend**

| | | | | | |
|---|---|---|---|---|---|
| ⇑ | updates | UNITE | operators | *tables* | data updates generated |
| ↑ | SQL-statements applied to output relation | insert... | queries generated by operator | | |

Figure 10.7: Update Propagation in the View from Figure 9.2. See Section 10.1.5 for explanation.

Algorithm *propagateUpdate* will perform a postorder tree traversal, i.e., process the deepest node (UNITE) first, and the root node (SPLIT) last. The operators are denoted by $\omega_1$ through $\omega_5$, in order of their processing. First, the UNITE operator propagates the incoming update into a one-element sequence $\partial\omega_1$ of updates which is then used as input to the FOLD-operator. The FOLD-operator propagates its input into a two-element sequence $\partial\omega_2$, sent to the StandardSQL-operator. This operator then propagates each of the two updates separately, creating two sequences $\partial\omega_{3_1}$ and $\partial\omega_{3_2}$, with 1 and 0 elements, respectively. Recall from Section 10.1.3 that in the case of more than one update sequence being created by an operator, those sequences can simply be concatenated before the next operator's propagation is executed, yielding $\partial\omega_3$. Since one update is not propagated due to the WHERE-condition in the StandardSQL-node, we have $\partial\omega_3 = \partial\omega_{3_1}$. UNFOLD now transforms its incoming one-element update sequence $\partial\omega_3$ into another one-element sequence $\partial\omega_4$ which becomes the input for the SPLIT-operator. This operator finally creates a two-element sequence, consisting of an *add-relation* schema change followed by an *add-tuple* data update. This sequence is the final update sequence $\partial V$ which is applied to view $V$, leading to the new view $V'$ equivalent to the view obtained by recomputation.

### 10.1.6 Grouping Similar *SchemaSQL* Updates in Batches

Certain updates in our strategy are transformed by some operators into update sequences $\partial$ in which all the updates are similar. This gives an opportunity for optimization on our update propagation strategy.

For example, a FOLD-node can transform a single schema change (such as a *attribute-delete*) into a sequence of data updates (such as a sequence of *tuple-deletes*). An inspection of the update propagation tables in this section shows that, typically, such a sequence consists of similar updates. Consider the example in Fig. 10.7, where a deletion of attribute Business in relation TMP_REL_0001 would lead to a sequence of *delete-tuple* updates of all tuples in TMP_REL_0002 that have the value Business in the attribute Type. A simple way of executing all those updates efficiently using SQL would be to issue a query such as `delete from TMP_REL_0002 where type='Business'`. Thus, instead of propagating all individual tuple updates using some delta relation, as done in traditional view maintenance, we instead propose to abstract this sequence of updates into an SQL update statement and push the complete statement through the algebra tree.

We have identified two classes of such *batched updates* that occur frequently as outputs of our propagation strategy, as described below.

**Definition 10.3 (Batched Update)** *A* batched update *is a sequence of* SchemaSQL *updates, denoted by $\partial$, which adheres to one of the following structures:*

- *$\partial$ consists entirely of* delete-tuple-*updates to the same relation $R$, with equal schema and a set of attributes $a_1 \dots a_k$ whose values are a unique identifier for each tuple in $\partial$ (i.e., form a key). We denote such a sequence by*

  $$\text{delete-tuple-batch}(r, \mathit{cond}(a_1, c_1), \dots, \mathit{cond}(a_k, c_k))$$

  *with $\mathit{cond}(a_i, c_i)$ a condition selecting tuples $t \in R$ that have value $c_i$ in* *attribute $a_i$ ($t[a_i] = c_i$). This represents a set of delete-tuple statements on the output relation $R$ that could be generated by an SQL-***delete** *statement with the WHERE-conditions* $\mathtt{a_1} = \mathtt{c_1}, \ldots, \mathtt{a_k} = \mathtt{c_k}$.

- $\partial$ *consists entirely of* update-tuple-*updates to relation $R$. All update-tuples have equal schema. Parameters are a single attribute $b$ with (unique or duplicate) values, and a function $f$ between the old and new values of another attribute $a$ in each tuple. We denote such a sequence by*

  $$\text{update-tuple-batch}(r, a, f, b, c)$$

  *with $a, b$ denoting attribute names, $f$ denoting a function over the domain of the attribute with name $a$ (that is, $f : D_a \rightarrow D_a$), and $c$ denoting a constant. This represents a set of update-tuple updates affecting every tuple $t$ for which the value of attribute $b$ is $c$, by changing the value $t[a]$ to $f(t[a])$, i.e., $\forall t \in R$ s.t. $t[b] = c :\ t[a] \leftarrow f(t[a])$. In words, the update* update-tuple-batch$(r, a, f, b, c)$ *means "in relation $r$, set $a = f(a)$ where $b = c$". Note that for simplicity, we are restricting batched updates to a single WHERE-condition.*

With this definition of batched update, the above example can now be represented as *delete-tuple-batch* (TMP_REL_0002, *cond*(type, 'Business')).

We do not define insert-tuple batches since we consider only single data updates or schema changes entering our algebra tree, and such updates will never be transformed into larger "batches". In particular, adding an attribute or a relation in a base table means adding an *empty* structure containing no data. As only structures with matching schemas (i.e., attribute of a matching data type or relations with a matching set of attributes) can be added to the information space, the only new information to the system is the *name* of the new attribute or relation, respectively. Thus, such updates do not lead to batches of updates, and in fact often do not lead to any updates on the view extent at all.

Batches of schema-changes are also not useful because meaningful schema-change batches do not occur in our context. Inspection of the update tables in this section shows that, with the exception of the Split node, propagation of schema changes always leads to a *single* schema change, not sequences of related changes. In the case of the Split-node, any resulting "batch" of schema changes will lead to changes across several relations, an operation that cannot be optimized using our batched-approach and SQL-statements.

As mentioned above, the main benefit of batched updates lies in a possible optimization of the implementation of our update propagation strategy. Since some operators generate batches of related updates, considering batches as types of updates and propagating those through the algebra tree just like single updates could lead to performance improvements of the system. For an example, consider again Fig. 10.7 and an update *delete-attribute*$(\mathsf{TMP\_REL\_0001}, \mathsf{Business})$ as input to the Fold-operator. Setting $n = |\mathsf{TMP\_REL\_0001}|$, this update in the current strategy would lead to a propagation of $n$ single *delete-tuple* updates, whereas a treatment of all those updates as a batch would require the propagation of only one update, namely *delete-tuple-batch*(TMP_REL_0002,*cond*(type,'Business')). Figures 10.8–10.11 show the propagation tables. As before, the input table is denoted by $R$ (with name $r$) and the output table by $Q$ (with name $q$). The remaining syntax follows Def. 10.3.

| Input Change | Parameters | Conditions | Propagation |
|---|---|---|---|
| *delete-tuple-batch* | $(r, cond(a, c))$ | $a = a_p$ | *delete-attribute(q,c)* |
| | | $a = a_d$,<br>$A^*$ unchanged | `foreach` $a \in A^*$<br>`update Q`<br>`set` $a =$ `NULL`<br>`where` $a = c$ |
| | | other | *delete-tuple-batch*<br>$(q, cond(a, c))$ |
| | $(r, cond(a_1, c_1), \ldots, cond(a_n, c_n))$ | | *delete-tuple-batch*<br>$(q, cond(a_1, c_1), \ldots, cond(a_n, c_n))$ |
| *update-tuple-batch* | $(r, a, f, b, c)$ | $f(v) = c_{\text{new}}$ (a constant function),<br>$a = b = a_p$ | *rename-attribute(q,c,$c_{\text{new}}$)* |
| | | $a = a_d, b = a_p$ | `update Q set` $[c] = f([c])$<br>or<br>*update-tuple-batch*<br>$(q, c, f, null, null)$ |
| | | $a = a_d, b \neq a_p$ | `foreach` $\nu \in A^*$<br>*update-tuple-batch*$(q, \nu, f, b, c)$ |

Figure 10.8: Batched Update Propagation Rules for Q=UNFOLD $_{a_p,a_d}(R)$

## 10.2 Correctness

Our update propagation strategy is equivalent to a stepwise evaluation of the algebraic expression constructed for a query. Each operator transforms its input changes into a set of semantically equivalent output changes, eventually leading to a set of changes that must be applied to the view to synchronize it with the base relation change. In this section, we will show that

| Input Change | Parameters | Conditions | Propagation |
|---|---|---|---|
| *delete-tuple-batch* | $(r, cond(a, c))$ | $a \in A^*$ | `delete from Q`<br>`where` $a_p = a$ `and` $a_d = c$ |
| | | other | *delete-tuple-batch*<br>$(q, cond(a, c))$ |
| | $(q, cond(a_1, c_1), \ldots, cond(a_n, c_n))$ | | *delete-tuple-batch*<br>$(q, cond(a_1, c_1), \ldots, cond(a_n, c_n))$ |
| *update-tuple-batch* | $(r, a, f, b, c)$ | $a \in A^*, b \in A^*$ | `update Q`<br>`set` $a_d = f(a_d)$<br>`where` $a_p = a$ `and` $b = c$ |
| | | other | *update-tuple-batch*$(q, a, f, b, c)$ |

Figure 10.9: Batched Update Propagation Rules for Q=FOLD $_{a_p,a_d,A^*}(R)$

| Input Change | Parameters | Addtl. Conditions | Propagation |
|---|---|---|---|
| *delete-tuple-batch* | $(r, cond(a, c))$ | $a = a_p$ | *del-relation(q,c)* |
| | | $a = a_d$,<br>$A^*$ unchanged | `foreach` $q \in A^*$<br>`update` $q$<br>`set` $q.a_d =$ NULL<br>`where` $q.a_d = c$ |
| | | other | `foreach` $q \in A^*$<br>*delete-tuple-batch*$(q, cond(a, c))$ |
| | $(r, cond(a_1, c_1), \ldots, cond(a_n, c_n))$ | | `foreach` $q \in A^*$<br>*delete-tuple-batch*<br>$(q, cond(a_1, c_1), \ldots, cond(a_n, c_n))$ |
| *update-tuple-batch* | $(r, a_p, f, b, c_{\text{old}})$ with $f(v) = c_{\text{new}}$ (a constant function) | $b = a_p$ | *rename-relation(*$c_{\text{old}}$*,*$c_{\text{new}}$*)* |
| | | $b \neq a_p,\ a \in A^*$ | `foreach` $q \in A^*$<br>*update-tuple-batch*$(q, a, f, b, c)$ |

Figure 10.10: Batched Update Propagation Rules for Q=SPLIT $_{a_p}(R)$. Note that $A^* = \{q_1, q_2, \ldots, q_k\}$ is the set of output relation names.

| Input Change | Parameters | Addtl. Conditions | Propagation |
|---|---|---|---|
| *delete-tuple-batch* | $(r_x, cond(a, c))$ | | `delete from Q`<br>`where` $a_p = r_x$ `and` $a = c$ |
| *update-tuple-batch* | $(r_x, a, f, b, c)$ | $a \in A^*$ | `update Q`<br>`set` $a = f(a)$<br>`where` $a_p = r_x$ `and` $b = c$ |

Figure 10.11: Batched Update Propagation Rules for Q=Unite $_{a_p}(R_1, R_2, \ldots, R_n)$

this strategy leads to correct update propagation. Recall that we denote an update sequence applied to relation $R$ by $\partial R$. We will use the notation $R$ for the input relation and $Q$ for the output relation throughout this section.

Before we prove the correctness of the algorithm, we state some observations: The structure of the algebra tree for a view depends only on the query, not on the base data [LSS99]. The only changes to operators under base relation updates are possible changes of parameters (schema element names) inside the operators; an algebra operator can not disappear or appear as the result of a base update. However, the entire view query may be rendered invalid, for example under some *delete-relation*-updates.

Furthermore, an inspection of the update propagation algorithm (see Fig. 10.1, p. 199) shows that the propagation of any single base relation update occurs strictly along a path in the algebra tree, strictly from a leaf to the root. That is, only *SchemaSQL* algebra operators along the single path from the updated base relation to the root are affected by an update. This is in contrast to SQL view maintenance, where maintenance queries to related sources are necessary for some operators. The four *SchemaSQL* operators do not combine input relations in a way similar to an SQL-join,

Figure 10.12: A *SchemaSQL* Algebra Tree.

so that maintenance queries to other branches of the algebra tree are not generated by the *SchemaSQL* operators. For correctness of standard SQL maintenance queries (which only occur for some operators such as *join*), we rely on well-known related work [GL95, AESY97].

Let us label the output relations of each operator along the path of update propagation with $X_1, \ldots, X_n$, in ascending order from the operator closest to the leaf to the operator closest to the root of the algebra tree. In Fig. 10.12, we have labeled the output relations of each operator $(X_1, \ldots, X_4)$, as well as the base relations $(R_1, R_2)$ and the view $(V)$.

We first prove correctness of operators and then show the overall propagation scheme to be correct.

**Theorem 10.1 (Correctness for Single Operators)**
*Let $\omega \in \{\textsc{Unite}, \textsc{Split}, \textsc{Unfold}, \textsc{Fold}, \pi, \sigma, \times\}$ be a node in a* SchemaSQL *algebra tree. Let $R$ be the input relation(s) for $\omega$ and $Q = \omega(R)$ be its output relation(s). Furthermore, let $\Delta R$ be a data update or schema change to $R$, transforming $R \rightarrow R'$ and $Q_{\textsf{REC}}$ the output relation of $\omega$ after recomputation. Applying the rules from the update propagation tables Figs. 10.2–10.5 and Section 10.1.4 for $\omega$ and $\Delta R$ will generate a sequence of updates defined on the node's output relation (denoted by $\partial Q$, see Def. 10.1) that transforms $Q \rightarrow Q_{\textsf{INC}}$, with $Q_{\textsf{INC}} = Q_{\textsf{REC}}$.*

**Proof:** The proof is given by inspecting the update propagation tables, Figs. 10.2–10.5, and comparing their output with the expected output after recomputation for each case. We will only give two examples for such comparisons as a proof idea. Consider the propagation of a *delete-tuple* data update in the FOLD-operator (Fig. 10.3). Let a relation $R$ be folded by $Q = \textsc{Fold}_{a_p,a_d,A^*}(R)$. Now consider the relation $R' = R\backslash\{t\}$, with tuple $t$ deleted. Note that $t$ has up to $|A^*|$ non-null values in its data attributes (i.e., in attributes whose names are in $A^*$). For each of those non-null values, the pre-update output relation $Q$ contained a separate tuple which now has to be deleted. Therefore, after recomputation, the FOLD-operator produces an output relation $Q'$ that differs from $Q$ in that it has up to $|A^*|$ tuples less. All those missing tuples have as a common feature that they agree in the values of their attributes $a_1, \ldots, a_n$ (i.e., all attributes except the ones in $A^*$) with the deleted tuple. This is precisely what the update propagation rule (line 2 of Fig. 10.3) accomplishes by deleting all tuples with that condition.

As another example, let us also consider the propagation of the schema change *delete-attribute*$(r, a)$ in FOLD (line 5 of Fig. 10.3). Recomputation of the operator yields a $Q'$ that differs from $Q$ in one of two ways: if a data attribute $A$ ($a \in A^*$) in $R$ is deleted, all tuples whose values in $A_p$ correspond to the name of $A$ are missing from $Q'$. If a non-data attribute is deleted from $R$, the attribute in $Q'$ that has the same name as the deleted attribute in $R$ is deleted. In both cases, the update propagation rules change $Q$ in exactly that way.

The remaining operators and cases can be verified in a similar fashion. □

The following corollary is immediate since if an update sequence correctly transforms a relation, it must also be valid (Def. 10.2) on that relation.

**Corollary 10.1** *The propagation of any update defined on input relation $R$ through an operator $\omega$ will produce a valid update sequence for output relation $Q$.*

**Theorem 10.2 (Correctness of *SchemaSQL* View Maintenance)** *Let $V$ be a view defined over the set of base relations $R_1, \ldots, R_p$, and $\Delta R_u \in \{DU, SC\}$ an update applied to one relation $R_u$ $(1 \leq u \leq p)$. Let $R'_u$ be the relation $R_u$ after the application of $\Delta R_u$ and $V'_{\mathsf{REC}}$ be the view after recomputation. Furthermore, let the* SchemaSQL *View Maintenance Algorithm as defined in Section 10.1.3 produce a change sequence $\partial V$ that transforms view $V$ into view $V'_{\mathsf{INC}}$. Then, $V'_{\mathsf{REC}} = V'_{\mathsf{INC}}$.*

**Proof:** Let $n$ be the number of intermediate relations $X_i$ affected by an update (along the path from $R_u$ to $V$). We want to prove that recomputation generates the same intermediate relations (and therefore the same view relation) as incremental updating, i.e, $\forall i$ $(1 \leq i \leq n) : (X'_i)_{\mathsf{REC}} = (X'_i)_{\mathsf{INC}}$ and thus $V'_{\mathsf{REC}} = V'_{\mathsf{INC}}$. The proof is by induction over $X_i$ for $i = 0 \ldots n+1$. Set $X_0 = R_u$, $(X'_{n+1})_{\mathsf{REC}} = V'_{\mathsf{REC}}$, $(X'_{n+1})_{\mathsf{INC}} = V'_{\mathsf{INC}}$.

**Base Case:** The base case for $i = 0$ is trivial. $R'_u$ is the same relation, whether the algebra tree is recomputed or incrementally updated, i.e., $(X'_0)_{\mathsf{REC}} = (X'_0)_{\mathsf{INC}} = R'_u$.

**Induction Hypothesis:** $(X'_k)_{\mathsf{REC}} = (X'_k)_{\mathsf{INC}}$ $(k \geq 0)$.

**Induction Step:** It is to show that $(X'_{k+1})_{\mathsf{REC}} = (X'_{k+1})_{\mathsf{INC}}$.

Since by hypothesis, $(X'_k)_{\mathsf{REC}} = (X'_k)_{\mathsf{INC}}$, there must exist an update sequence $\partial X_k$ that correctly transforms $X_k$ to $X'_k$ (and must therefore be valid on $X_k$). Let us denote the operator whose input table is $X_k$ by $\omega_k$ and let $m = |\partial X_k|$. By Thm. 10.1, any single valid update to any relation is correctly propagated through any one operator, in the sense that recomputation of the operator will yield the same result as incremental propagation. If $m = 1$, the induction step is thus proven. For $m > 1$, a valid sequence of $m$ updates on $X_k$ will trigger a sequence of incremental propagation steps in $\omega_k$. This will cause $\omega_k$ to transform $X_{k+1}$ into a sequence of $m$ intermediate (temporary) relations $(X^{(1)}_{k+1})_{\mathsf{INC}}, \ldots, (X^{(m)}_{k+1})_{\mathsf{INC}}$, each of which is equivalent to the corresponding state $(X^{(1)}_{k+1})_{\mathsf{REC}}, \ldots, (X^{(m)}_{k+1})_{\mathsf{REC}}$ that could be reached by recomputing $\omega_k$ after each update. Note that $X^{(m)}_{k+1} \equiv X'_{k+1}$. After application of all $m$ updates to $X_{k+1}$ we have $X'_{k+1} = (X^{(m)}_{k+1})_{\mathsf{INC}} = (X^{(m)}_{k+1})_{\mathsf{REC}}$, or $(X'_{k+1})_{\mathsf{INC}} = (X'_{k+1})_{\mathsf{REC}}$. If any valid sequence of updates gets propagated correctly, the sequence $\partial X_k$ (valid by Corollary 10.1) in particular must also be correctly propagated, i.e, produce a relation $(X'_{k+1})_{\mathsf{INC}}$ with $(X'_{k+1})_{\mathsf{INC}} = (X'_{k+1})_{\mathsf{REC}}$. **q.e.d.** □

# Chapter 11

# Implementation and Evaluation

## 11.1 Implementation

### 11.1.1 *SchemaSQL* Query Engine

The update propagation strategy described in this dissertation has been implemented in Java on top of a *SchemaSQL* query evaluation module also written by the author. This query engine was built based on the description in [LSS99]. Our code first parses *SchemaSQL* queries (using JavaCC), then builds an algebra tree out of the parsed query, and finally evaluates the query result through a postorder traversal of that tree, computing the output of each algebra node as it is visited (see Fig. 11.1). The next node then reads its child node's temporary relation to compute its output. For this prototype implementation, each node temporarily stores its output through JDBC in the query engine's "local" relational database (Oracle 8) as keeping relations in memory only incurs limitations on the size of input relations and also would have required us to reimplement significant parts of relational query technology. For performance reasons, all intermediate nodes in the algebra tree share one JDBC-connection to the local database.

The implementation of the query engine uses pure Java and JDBC-connections to several instances of Oracle 8. We use standard SQL DDL and DML statements (sequences of `select`, `insert`, `delete`, `update` and statements for schema change operations like `alter table`) for all queries—thus making full use of the source database's SQL query evaluation capabilities. We do not use any system specific functions other than simple schema changes. A wrapper class (which we have also successfully implemented for Microsoft's Access) makes differences in the syntax of schema change operations transparent. Therefore, the implementation is independent of the database used.

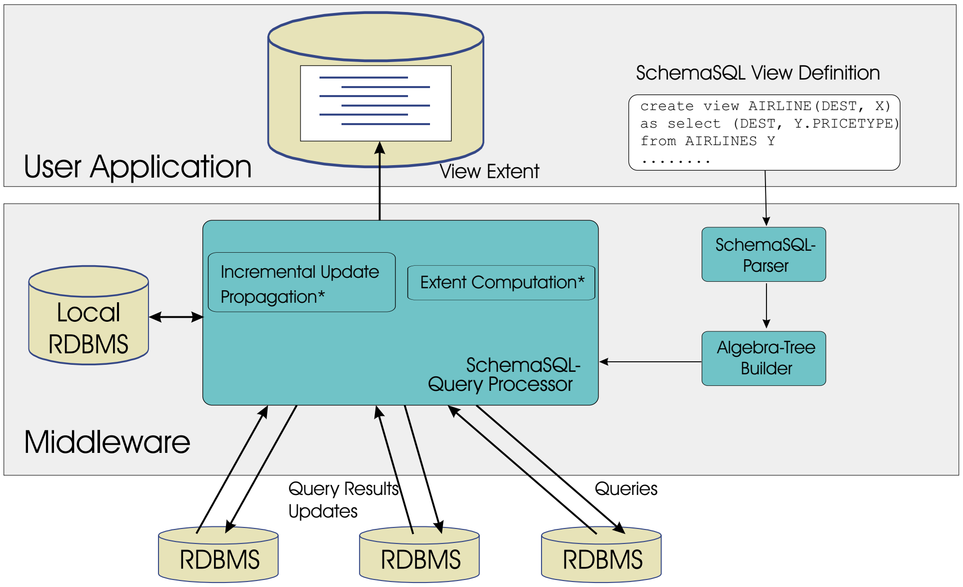
To improve performance and simplify our code, our implementation attempts to use as much of the SQL query capability as possible, in particular by extracting standard SQL out of the *SchemaSQL* query and evaluating it in a single `StandardSQL`-node, which simply executes its stored SQL-statement against the local SQL-database. The *SchemaSQL* query engine currently does not perform or utilize any query optimization strategies other than those provided by the underlying SQL query engine when executing queries against the local database.

### 11.1.2 Incremental Update Propagation

To perform update propagation in the manner described in this dissertation, we added update propagation capabilities to each algebra operator class (that is, Unite, Fold, Unfold, Split, StandardSQL). A method in each node (named `propagateUpdate()`) accepts one update and returns a list of (data and/or schema) updates which represent the result of the update propagation. Then, we added code for the propagation of the output updates of each operator to its parent. Thus, the same code that performs the postorder traversal of the operator tree for the initial materialization of the view can now also perform the incremental update propagation, by simply calling the update propagation method instead of the materialization method on each node and recursively using each operator's output as the input for the parent operator.

Updates were modeled into a small class hierarchy consisting mainly of the classes `SchemaUpdate` and `DataUpdate`. Each node in the algebra tree is now extended by the ability to propagate all such updates. The current code does not support batched updates.

Fig. 11.1 shows the architecture of our system. The main difference to a traditional query engine implementation is the fact that, instead of being a module separate from the query evaluation engine, the update propagation module is part of the engine and an instance of this module is active in every algebra tree operator. This enables the system to propagate updates incrementally through the algebra tree.



```
create view AIRLINE(DEST, X)
as select (DEST, Y.PRICETYPE)
from AIRLINES Y
........
```

* One instance of this module for each node in algebra tree

Figure 11.1: The architecture of the *SchemaSQL* View Maintenance System

## 11.2 Performance Evaluation

### 11.2.1 Experimental Setup

#### Factors Relevant to Performance

As already explained, *SchemaSQL* update propagation is significantly different from traditional incremental view maintenance. Major differences are the transformation of data updates into schema changes and vice-versa, the need for the propagation of base schema changes, and the propagation strategy based on propagating an update through an algebra tree rather than computing delta-queries against the base relations.

To assess the influence of those issues on performance, we executed a number of experiments using our prototype. We are reporting some of the results in this section. For the experiments described here, we focused on the following factors contributing to *SchemaSQL* update propagation performance:

- the type of update (data update or schema change) at the base relations;
- the transformation type of the update (i.e., the type, data update or schema change, into which a base update is propagated);
- the selectivity of conditions in the view query that determines the size of the view relative to the sizes of the base relations;
- the size of base relations.

### Schema and View Queries

If not stated otherwise, all our experiments use the following view query, over the same base schema as in our running example (Fig. 9.1):

```
create view CITY(Type, AIRLINE) AS
  select PRICETYPE, FLIGHT.PRICETYPE
  from   -> AIRLINE,
         AIRLINE FLIGHT,
         AIRLINE-> PRICETYPE,
         FLIGHT.Destination CITY
  where PRICETYPE <> 'Destination'
  and AIRLINE like 'AIRLINE%'
  and FLIGHT.PRICETYPE <= '1101';
```

The output schema of this query, considering the input schemas from Fig. 9.1, consists of two relations Business and Economy which both have the schema (Destination,AIRLINE_1,...,AIRLINE_K), with one attribute named AIRLINE_$X$ for each relation named AIRLINE_$X$ in the input schema. The output schema may change during an experiment.

The base data was generated from a list of strings (representing names of cities), augmented by random numbers representing "flight prices". Those numbers were generated using uniformly distributed random numbers in a certain range. The base relation sizes and distribution of updates are described with each experiment.

Since we have multiple output relations, we need to extend the concept of view size to multiple relations. We thus define the *view size* to be the *sum* of the sizes of all output relations.

### Measurements and System Parameters

The test system was a Pentium II/400 running Linux and Blackdown Java 1.2.2. Database connectivity was achieved through JDBC. The databases used for our tests (local database and information source) were two installations of Oracle 8i, running on a Pentium 233 under Windows NT and a 4-processor 300MHz DEC Alpha under DEC OSF1, respectively.

For our experiments, we measured the total execution time of the initial materialization of the view (for control purposes, not shown in the chart), then the time for a number of updates, depending on the experiment, and finally the time for a recomputation of the view extent. The times were measured by comparing the system time before and after executing update propagation or recomputation, i.e., they include system, user, and I/O time.

### 11.2.2 Deleting Base Relations of Different Sizes

In our classification of updates (Section 10.1), inserting a table implies inserting an *empty* table. The data would have to be added in subsequent data updates. Therefore, inserting (and also renaming) schema elements leads to relatively simple propagation results, as the information content of the database does not change much under such updates. Therefore the experiments reported here concentrate mainly on the deletion of schema elements (attributes, relations), as well as on the insertion and deletion of data.

We ran the above query over a schema containing four base relations (representing four different airlines) with approximately 100, 200, 300, and 400 tuples, respectively. We then deleted each of those base relations and compared the time for incremental update propagation with the time for recomputation after a base relation was deleted. After each deletion and measurement, the original information space was restored. With the above query, the original view extent had two relations Business and Economy with 378 and 444 tuples, respectively and decreased roughly proportionally after deletions of base relations. Fig. 11.2 shows the times measured.

In our current implementation, deleting a base relation $R$ in a query such as the one above will result in the creation of approximately $|R|$ updates inside the operator tree. Therefore, deleting larger relations takes longer than deleting smaller relations. On the other hand, recomputation time will decrease with larger sizes of the deleted relation, as the resulting view extent has less tuples. The crossover point between view maintenance and recomputation is at about a base size of 225, i.e., deleting a relation of more than 225 tuples (about 25% of the view size) will take longer than recomputing the view. This means that our update propagation strategy will perform better than recomputation for tuple-wise deletions of up to 25% of the entire information space. Propagating 225 delete-tuple updates will have a smaller total execution time than a single recomputation of the view. On the other hand, this experiment shows that a single input update (in this case a *delete-relation* update) can be very expensive as it may lead to many updates in the view extent.

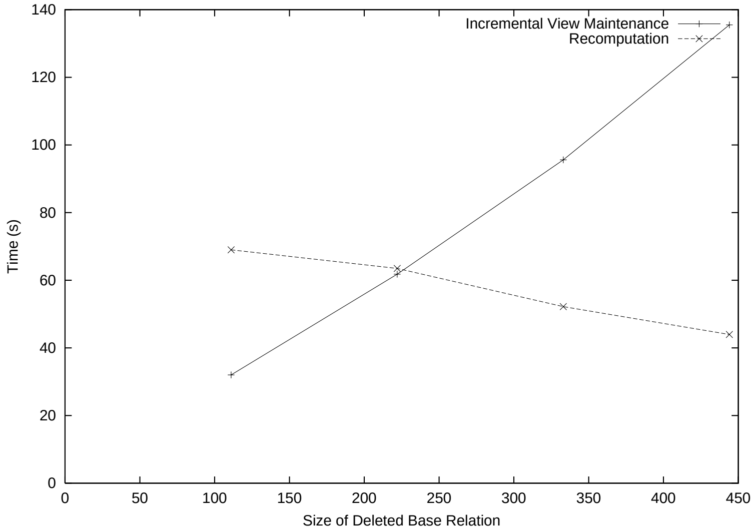


Figure 11.2: View Maintenance and Recomputation Times vs. Size of Deleted Base Relations

We also ran the same set of updates under the view query from our running example (Fig. 9.1), which creates one output relation for each unique destination in any of the input relations. A first experiment showed that, during the materialization of the view from large base relations, the Oracle database we were using rejected the creation of new tables after about 620 `create table`-statements. So we repeated the experiment with the above schema having 100,200,300, and 400 tuples and deleted each relation. This time, the query computed properly. Some of the running times in seconds are given in Table 11.1.

| Base Table Deleted | Relation Size | Incremental VM | Recomputation |
|---|---:|---:|---:|
| AIRLINE_LH | 333 | 395s | 125s |
| AIRLINE_KL | 111 | 318s | 145s |

Table 11.1: Propagation and Recomputation in the Query from Fig. 9.1

Note that in this case, the deletion of a base relation eventually leads to *one delete-attribute schema change per output relation*, so deleting a base relation with 222 tuples will first lead to the propagation of 222 *delete-tuple* updates, propagated to one output relation each, and then to up to 444 *delete-attribute* schema changes—exactly one for each output relation. The experiment shows that in this case, which incurs many schema changes in the output relations, a recomputation has a performance advantage over the unoptimized incremental update propagation. An obvious optimization for the processing of this particular query would be to ignore the *delete-tuple*-updates and apply only the *delete-attribute* changes.

### 11.2.3 Deleting Tuples from Base Relations

In this experiment (Fig. 11.3), we delete a sequence of random tuples from the base schema and measured the cumulative propagation time. For our chart, we numbered the updates by consecutive numbers $i$. The cumulative propagation time for update $i$ is the sum of the propagation times for all updates numbered $0 \dots i$. We also measured the time to recompute the view after the entire update sequence was executed. This experiment shows again that in our schema, the crossover point between incremental maintenance and recomputation is at roughly 200 tuples, i.e., after 200 updates, recomputation of our view (with view size 888) would become more efficient. The view size is the major factor determining the recomputation time for a view, whereas the time for incremental propagation of a single data-update mainly depends on the system implementation, i.e., is roughly constant for our implementation and test environment. The average time to propagate a single update can be estimated from the slope of the curve in Fig. 11.3 to be about 285ms, which is a value depending mainly on the size of the tuples propagated.

From those facts, it can be concluded that the *ratio* between the number of propagated updates at the crossover point and the view size is a system constant depending only on the implementation, i.e., we expect that recomputation of a view will take roughly the same time as the incremental propagation of the deletion of a certain percentage of the view's tuples. In our experiment, this ratio was about 1/4 (a crossover point of 200 for a view with roughly 800 tuples).

Note the jump in the incremental maintenance time at the end of the curve. This time represents a data update that led to a schema change in the output relation. The reason is that the last tuple from a base relation was deleted which led to a *delete-attribute* change in an output relation.

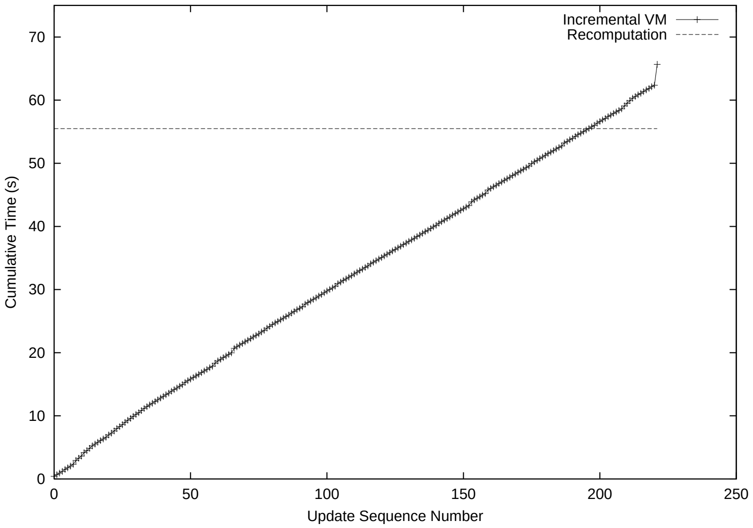


Figure 11.3: Deleting Tuples from the Input Schema

### 11.2.4 Deleting Tuples Leading to Schema Changes

In this experiment, we wanted to assess the difference in propagation time of the same updates, depending on whether these updates lead to data updates or schema changes in the view. Schema changes actually executed on a relation database are slow operations. Therefore, the expectation is that an update propagation (including the application of those updates against a database) that leads to a schema change will be slower than an update propagation leading to only a data update in the view.

Thus, in this experiment we are deleting tuples from the base schema. This time we have four base relations $R_1, R_2, R_3, R_4$ of sizes 1, 10, 100, and 1000, respectively, but make sure that some of the updates incur schema changes in the output schema. First, we inserted 10 tuples into relation

$R_1$, then removed tuples from relation $R_2$ one-by-one, until $R_2$ was empty. This leads to a schema change in the output schema since the corresponding attribute is removed from each output relation. Then, we removed all 11 tuples from relation $R_1$ (incurring another schema change), followed by a removal of 10 random tuples from relation $R_3$. Note that in this sequence, updates #19 and #30 lead to schema changes in the output.

We then plotted the time each update took to propagate. Note the relatively even distribution of update propagation times around 250 ms in Fig. 11.4, except for two updates, which take over 2 seconds to propagate. Those are clearly those updates that led to schema changes in the output relations. The measured update time (if no schema change is incurred) is similar to the average time measured in Fig. 11.3. Again, the propagation time for the data updates depends mainly on the tuple size, and the underlying database, whereas the time for schema changes depends on the underlying database only (as practically no data has to be transported).

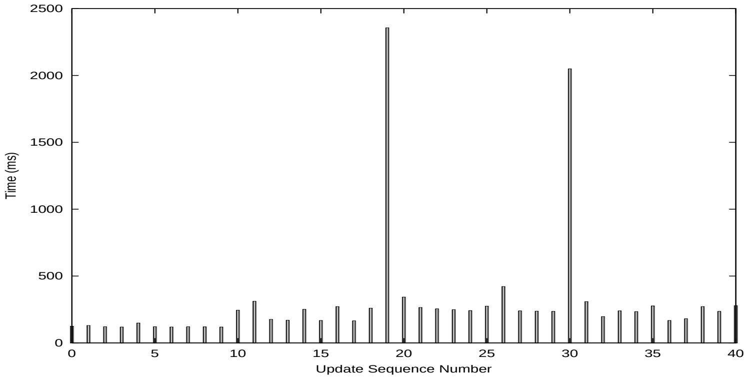


Figure 11.4: Base Updates lead to Data Updates or Schema Changes

### 11.2.5 View Selectivity

In this experiment, we measure how the performance advantage of incremental view maintenance over recomputation is affected by the view selectivity, i.e., by the probability that a base tuple's data will actually be reflected in a view.

To assess the effect of different view selectivities on view maintenance times, we ran an experiment over different selectivities in the view query. We adjusted selectivity in the range $[0.02 \ldots 1]$ by using different constant values for local conditions in the WHERE-clause of our query (i.e., conditions of the type `FLIGHT.PRICETYPE<=1100`). We define view selectivity over our multiple-relation output schema in analogy to view size as the ratio of the view size of the current query and the view size of a query without WHERE-clause. For each selectivity setting, we deleted a relation with 100 tuples (10% of the input tuples) from the base schema and measured incremental view maintenance time and view recomputation time. Fig. 11.5 shows the result of the experiment. The graph shows that both incremental view maintenance time and recomputation time increase with the view selectivity, which is not surprising, since in both cases more tuples have to be processed when the view selectivity (and thus the view) becomes larger. However, the relative increase in the incremental update propagation time is similar to the relative increase in recomputation time, meaning that our propagation strategy will keep its performance benefits under changes of the view's selectivity.

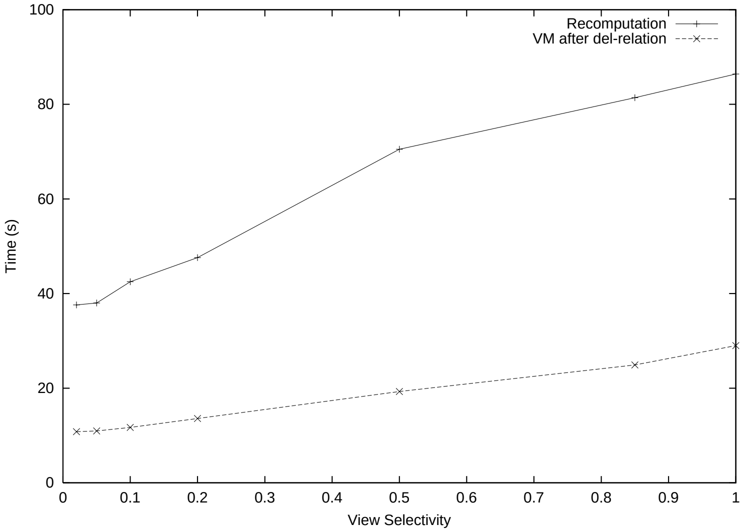


Figure 11.5: Update Propagation under Views of Different Selectivities

# Chapter 12

# Related Work

The integration of data stored in heterogeneous schemas has long been an object of intensive studies. The problem of schematic heterogeneity or different source capabilities is repeatedly encountered when attempting to integrate data. Some examples are Garlic [TAH$^+$96], TSIMMIS [HGMN$^+$97], DISCO [TRV96], and GenCompact [GMLY99].

A number of logic-based languages have been developed to integrate heterogeneous data sources,e.g., HiLog [CKW89] or SchemaLog [GLS$^+$97]. Some SQL-extensions have also been proposed, such as MSQL [LAZ$^+$89] which has capabilities for basic querying of schema elements, XSQL [KKS92] which allows schema-querying in object-oriented databases, and, in particular, *SchemaSQL* [LSS96] (see below).

Those approaches overcome different classes of heterogeneities in relational schemas. However, the important class of schematic heterogeneities in semantically equivalent relational databases is often excluded from integration language proposals. Krishnamurthy *et al.* [KLK91] were the first to recognize the importance of schematic discrepancies and developed a logic-based language called IDL to deal with such problems. Miller *et al.* [MIR93] show that relational databases may contain equivalent information in different schemas and give a formal model (Schema Intension Graphs) to study such "semantic equivalence" of heterogeneous schemas.

The predominant approach at integrating such semantically equivalent schemas has been done by Gyssens *et al.* [GLS96] and later by Lakshmanan, Sadri, and Subramanian [LSS96, LSS99]. In [LSS96], the authors present *SchemaSQL*, which is used as the basis for our work. *SchemaSQL* builds upon earlier work in SchemaLog [GLS$^+$97]. It is a direct extension of SQL, with the added capability of querying and restructuring not only data, but also schema in relational databases, and transforming data into schema and vice-versa. Thus, using *SchemaSQL* as a query language makes it possible to overcome schematic heterogeneities between relational data sources.

A second foundation of our work is the large body of work on incremental view maintenance. Many algorithms for efficient and correct view maintenance for SQL-type queries have been proposed. Prominent results, often taking concurrency into account, include ECA [ZGMHW95], SWEEP [AESY97], Mohania *et al.* [MKK97], and parallel view maintenance [ZRD01]. Those approaches follow an *algorithmic* approach in that they propose algorithms to compute changes to a view.

Griffin and Libkin [GL95] consider views with duplicates, and, more importantly, follow an *algebraic* approach which defines a complete and minimal set of relational algebra operators. They achieve a rigorous proof of the correctness of view maintenance by proving the correctness of those operators and their nesting. Their proof mechanism is similar to ours. Griffin and Libkin's work is partly based on the algebraic approach by Qian and Wiederhold [QW91].

Related to our work are also performance studies on incremental view maintenance algorithms. An early paper on measuring the performance of incremental view maintenance strategies is Hanson [Han87]. The ECA paper [ZGMHW95] contains a study on the performance of its algorithm, but only in an analytical manner rather than actual performance studies. More recently, there are studies on some view maintenance algorithms for example by Kuno *et al.* [KR98] and O'Gorman *et al.* [OAE00]. Griffin and Libkin [GL95] give an analytical complexity study of their algorithm but do not evaluate system performance.

# Chapter 13

# Conclusions

In this work, we have proposed the first incremental view maintenance algorithm for schema-restructuring views. We have expanded upon the work by Lakshmanan *et al.* [LSS96], which allowed *queries* to be defined over schematically heterogeneous sources, by introducing an algorithm to incrementally maintain view extents and schemas. We have shown that the traditional approach to incremental view maintenance—rewriting view queries and executing them against the source data—is not easy to adapt for such views, and that it is also necessary to include schema changes. We have solved this problem by defining an algebra-based update propagation scheme in which updates are propagated from the leaves to the root of the algebra tree corresponding to the query. As some updates (especially schema changes) are translated into long sequences of similar updates, we have also investigated the possibility of optimizations introducing *batches* of updates. Furthermore, we have formally proved the correctness of the algorithm. Performance experiments on a prototype of a view-maintenance-capable query engine have shown that update propagation has the expected large benefits over recomputation of views.

In summary, we believe our work is a significant step towards supporting the integration of large yet schematically heterogeneous data sources into integrated environments such as data warehouses or information gathering applications, while allowing for incremental propagation of updates. One application of this work is in a larger data integration environment such as EVE [NLR98] (Chapter 2), in which the *SchemaSQL* wrapper would help to integrate a new class of information sources into a view.

# Part IV

# Conclusions and Future Work

# Chapter 14

# Conclusions and Future Work

Information integration has been an important and lively topic of research for two decades. Despite all the efforts, a truly comprehensive solution for the information integration problem is not in sight. Several proposals have been made that approach the problem of integration under the assumption that information sources are well known and understood, and have equal data models and similar schemas. There has also been substantial work on semi-automatic methods to identify database contents, structures, relationships, and capabilities. However, there has been much less work in the area of a fully automatic discovery of database properties.

The goal of this dissertation is to provide solutions in information integration. We have concentrated on two key issues that have gained importance through newer developments in database technology: the discovery of relationships, and the integration of schematically heterogeneous sources.

## 14.1 Results and Contributions of this Dissertation

### Discovery of Inclusion Dependencies in Databases

Inclusion dependencies express redundancies between databases. Such redundancies are more and more common as more data is being collected by independent information providers and stored separately. Any effort to combine such data must rely on knowledge about inclusion relationships between databases. The inclusion dependency discovery problem has been proven NP-complete [KMRS92], and the naïve algorithm has a prohibitively high runtime even for very small problems (relations with as few as 10 attributes).

In Part II of this dissertation, we have shown a comprehensive, fully automated solution of this problem. The key to the solution is the reduction of the exponential-complexity problem (whose naïve solution consists in enumerating all possible inclusion dependencies and testing them) to a graph-problem, namely the maximal-clique problem. Even though the general clique problem in itself is also NP-complete, we used an algorithm that finds cliques in a time polynomial in the number of cliques in a graph. That enabled us to dramatically reduce the algorithm's complexity in the average case. We defined algorithm $\mathsf{FIND}_2$ which uses a clique-finding algorithm to discover inclusion dependencies. Algorithm $\mathsf{FIND}_2$ also incorporates a new algorithm HYPERCLIQUE defined by us that finds cliques in $k$-uniform hypergraphs, which is a necessary step in the discovery algorithm. The main advantage of the FIND$_2$ algorithm is its ability to reduce the number of inclusion dependency candidates *tested* against a database by many orders of magnitude, while still finding the complete and correct solution to the problem.

With algorithm FIND$_2$, we are able to solve the inclusion dependency discovery problem for relations of about 40 attributes. In order to extend the applicability of the algorithm, we defined heuristics that help to reduce the search space further. The heuristics are powerful enough to increase the feasible problem size for this exponential problem to relations with at least 100 attributes (up to 200 attributes, depending on the source data). By using heuristics, we sacrifice some of the completeness of the solutions, but will in most cases still discover the largest existing inclusion dependency between two relations (which is the most interesting result among all possible dependencies), or at least a large fragment of the largest dependency.

Extensive experiments on a prototype implementation show that the solution is feasible and works well on real-world data. We found that (1) the runtimes of both a naïve algorithm and a strategy along the lines of the Apriori-Algorithm for association rules [AS94] are prohibitively high for relations with as few as ten attributes, (2) our algorithm without heuristics finds the complete set of inclusion dependencies for relations with typically 50 attributes and 10,000 tuples in around one hour on a typical PC, and (3) using heuristics, the feasible problem size of this exponential-complexity problem can be extended to 100 attributes in difficult cases and 200 attributes in well-behaved cases. The number of tuples of the relations involved is only a linear factor for the runtime. The algorithm needs no manual input and can adapt to the nature of the problem, reporting less complete results for more difficult problems.

### Incremental Maintenance of Schema-Restructuring Views

Schema-restructuring views (for example defined in *SchemaSQL* [LSS96]) are an extension of traditional SQL-style views in that the rigorous distinction between schema (attribute names, relation names, domains) and data (values) in a relational database is softened. Schema-restructuring query languages can query across a list of related attributes, even using the attribute names themselves as data, such that predicates formulated over attribute names become possible. Conversely, values in the input relation can be transformed into attribute names in the view. A major advantage of such query languages is the ability to restructure a set of schematically heterogeneous but related databases into a common relational view. Schematic heterogeneity [MIR94] is a condition in which two databases are able to assume isomorphic sets of states (i.e., are able to hold the same data) but do so in schemas that can not be queried by a common SQL query.

Schema-restructuring query languages have been defined in the literature, but no view maintenance algorithm had been found so far. In Part III of this dissertation, we have provided the first such algorithm. View maintenance in schema-restructuring views is a challenging task. The transformation between schema and data which schema-restructuring languages achieve leads to the transformation of *data updates* into *schema changes* and vice versa as well. This, and the fact that due to the restructuring of schemas a data update cannot be assumed to have a predictable schema, means that the traditional view maintenance approach of computing differential queries and sending them to data sources cannot be used.

We have proposed an algorithm that instead performs maintenance at each operator in the query evaluation tree. Updates in our scheme occur in a leaf of that tree and are propagated through a depth-first (postorder) tree traversal. We have identified update propagation algorithms for each operator in a schema-restructuring algebra given by Lakshmanan *et al.* [LSS99] and defined an overall scheme that provides correct and efficient view maintenance of schema-restructuring views.

We implemented the solution and ran experiments testing correctness and performance. The experiments show a significant performance advantage over view recomputation, after both data updates and schema changes at the sources. We also identified optimizations for our view maintenance algorithm by combining sequences of similar updates in *batches* and propagating them as a unit. This update batching can achieve a further significant improvement in update propagation time. Furthermore, we provided a formal proof of the correctness of the algorithm.

## 14.2 Ideas for Future Work

### 14.2.1 Discovery Across Multiple Databases

The solution for inclusion dependency discovery that we provided in this dissertation is primarily optimized for the case of two relations whose interrelationships are to be found. However, the discovery problem can also be extended towards multiple relations. An idea for a solution given in this work is to simply find redundancies between any pair of relations and compute the union of the results. However, it seems likely that significant improvements can be made to this strategy. One approach would be to make use of the transitivity of inclusion dependencies. Computing the transitive closure of a subset of inclusion dependencies identified among a set of relations could lead to new inclusion dependencies whose existence does not have to be discovered by the $\mathsf{FIND}_2$ algorithm.

Global optimization for discovery across multiple relations seem possible. One idea would be to replace the current two-relation storage structure for inclusion dependencies in the $\mathsf{FIND}_2$ algorithm with a structure that can hold dependencies and their attributes among multiple relations. That may make it possible to find a new algorithm that uses global knowledge about sets of single attribute pairs, rather than complete inclusion dependencies. Exploiting knowledge about single attribute pairs across multiple relations may lead to a significant reduction of the search space for each pair of relations considered, especially in combination with the transitivity property of inclusion dependencies.

### 14.2.2 Interactivity in the Discovery Process

Our current algorithm runs as a stand-alone solution and works without human input or intervention. While that behavior seems appropriate for an “agent-like” use of our technology, it is desirable for the algorithm to be able to use any existing human input about the problem it is to solve. For example, a human expert could provide knowledge about known relationships between certain attributes, which would restrict the remaining search space. Another possibility that would be feasible to incorporate in algorithm $\mathsf{FIND}_2$ is a prioritization of attributes. Currently, all attributes in a given relation are considered equally important for the solution of the problem. However, if for large problem sizes a complete solution cannot be found, it would be possible for our algorithm to explore more closely a given subset of attributes and spend more time discovering relationships between such *interesting* attributes. Such use of human input could eventually lead to an interactive form of discovery, in which the algorithm presents intermediate results to a human user and incorporates user feedback to direct its further attention to certain subsets of the search space. An interactive solution along those lines could be integrated with the *EVE*-System (Chapter 2) for a more comprehensive view synchronization solution.

Naturally, a simple form of incorporating expert domain knowledge would be to simply exclude any IND known to be valid or invalid from testing by the algorithm. A list of known INDs could be kept and used similar to the Domain heuristic in avoiding some database queries.

### 14.2.3 Adaptive Discovery

A further direction of future work is the level of automation that the algorithm can achieve. Currently, the $\mathsf{FIND}_2$ algorithm works independent from user input, but may fail to find a complete result, or in same cases even any result, for large problem sizes. However, it is possible to improve on this behavior by (1) finding more heuristics that reduce the problem space further or achieve a higher-quality distinction between information and noise and (2) define a more comprehensive algorithmic flow, by incorporating "fall-back" mechanism that allow the algorithm to pursue alternative strategies when a particular step in the discovery process fails or times out. Currently, no such time-out is in place, the algorithm rather relies on empirical data that determine which strategy (heuristic or non-heuristic) is to be used.

### 14.2.4 Schema-restructuring Views

There are many interesting future work directions in the area of schema-restructuring views as well.

For example, a new class of wrapper generators could help in establishing such views in the first place. Currently, schema-restructuring query languages lack the understandability that SQL has. An interactive tool that helps a user to define schema-restructuring queries simply by pointing out schema elements to be transformed seems to be feasible and useful. The utility of such a tool could be greatly enhanced by incorporating some kind of *discovery* of relationships between sources with conflicting schemas. Our work on discovery of database interrelationships, presented in this dissertation, could be a step in this direction. Extending our work towards the discovery of redundancies among truly heterogeneous databases would be a major step towards an automatic integration of information sources.

### 14.2.5 Query Optimization and Implementation of View Maintenance

New query optimization strategies for our maintenance algorithm can be explored as well. Currently, the only query optimization that takes place is the batching of updates. While that strategy is quite successful, the question of a global query optimization for data updates or schema changes has not been explored. Lakshmanan *et al.* , who are the original authors of the important schema-restructuring language *SchemaSQL* [LSS99], have identified some simple optimization strategies upon which a more comprehensive theory could be built.

Finally, a more immediate next step would be a review of the implementation with the goal of removing some of the performance obstacles. Currently, all algebra operators require to store some data in a local database. It would be interesting to investigate whether this is a necessary feature of schema-restructuring query evaluation or whether it is possible to eliminate the need for intermediate storage.

### 14.2.6 Discovery and Maintenance in Non-relational Data

Last but not least, our work on both interrelationship discovery and schema-restructuring view maintenance could be extended towards other, non-relational, databases. While the discovery algorithm does not rely on relational data, it is not suitable for semi-structured databases, in which the concept of *attribute* is not as strictly defined as in the relational or object-oriented case. On the other hand, schema-restructuring views naturally occur in such semi-structured databases, for example in XML files. An interesting topic of research would be to explore to which extent existing query language proposals for such data models (such as XQuery [W3C01]) support views and to which extent existing update propagation strategies can be adapted to the semi-structured paradigm. We think that it is likely that the results obtained in this dissertation are useful in this context, since the idea of mapping data relationship discovery problems to graph (or hypergraph) problems has been proven to be feasible in a number of cases.

# Appendix A

# The Bron/Kerbosch-Algorithm

This algorithm has been published in [BK73] and finds the set of maximum cliques in a graph. While the general problem is NP-complete, the runtime of the algorithm is polynomial in the number of cliques, such that the algorithm finishes quickly for graphs with few cliques. The algorithm is given below in pseudocode (with zero-based arrays).

```
ALGORITHM BRON-KERBOSCH

INPUT:
A graph $G = (V, E)$

OUTPUT:
The set of maximal cliques in $G$

Stack compsub
Set cliques
int $nodes[0 \ldots |V| - 1]$

for $i \leftarrow 0 \ldots |V| - 1$
  $nodes[i] \leftarrow i$
extend($nodes, 0, |V|$)     //overwrites set cliques
return cliques
```

FUNCTION EXTEND

GLOBAL VARIABLES:
Graph $G$ as adjacency matrix $connected[0 \dots |V|-1][0 \dots |V|-1]$
Stack *compsub*
Set *cliques*

INPUT:
A set of nodes

```
function extend(int $oldSet[], ne, ce$)
  int $newSet[0 \dots ce-1]$
  $nod_{\mathrm{min}} \leftarrow ce; nod \leftarrow 0$   //number of disconnections
  for $i \leftarrow 0 \dots ce-1$
    $p \leftarrow old[i]; count \leftarrow 0$
    for $j \leftarrow ne \dots ce-1$
      if $(\neg connected[p][old[j]])$
        $count \leftarrow count+1; pos \leftarrow j$
        if $(count \geq nod_{min})$ break
    if $(count < nod_{min})$
      $fixp \leftarrow p; nod_{min} \leftarrow count$
      if $(i < ne)$ $s \leftarrow pos$;  $nod \leftarrow 0$
      else $s \leftarrow i$;  $nod \leftarrow 1$
    if $(nod_{min} = 0)$ break
  for $k \leftarrow (nod_{min} + nod) \dots 1$
    $old[s] \leftrightarrow old[ne]$;  $sel \leftarrow old[ne]; ne_{new} \leftarrow 0$
    for $i \leftarrow 0 \dots ne-1$
      if $(connected[sel][old[i]])$  $newSets[ne_{new}] \leftarrow old[i]$;  $ne_{new} \leftarrow ne_{new}+1$
    $ce_{new} \leftarrow ne_{new}+1$
    for $i \leftarrow ne+1 \dots ce-1$
      if $(connected[sel][old[i]])$  $newSets[ce_{new}] \leftarrow old[i]$;  $ce_{new} \leftarrow ce_{new}+1$
    $compsub$.push($sel$)
    if $(ce_{new} = 0)$
      $cliques$.addElement($compsub$)   //add stack as new clique
    else if $(ne_{new} < ce_{new})$ extend($newSet, ne_{new}, ce_{new}$)
    $compsub$.pop($sel$);  $ne \leftarrow ne+1$
    if $(k > 1)$
      $s \leftarrow ne$
      while $(connected[fixp][old[s]])$ $s \leftarrow s+1$
          //find node disconnected from $fixp$
  return
```

# Appendix B

# A Brute-Force Algorithm to Find Hypercliques

Figure B.1 gives the algorithm FIND_HYPERCLIQUES_BRUTE_FORCE in pseudo code. This exhaustive algorithm simply enumerates possible clique candidates, starting with the entire $k$-hypergraph and continuing towards smaller and smaller cliques, and tests them. As there is generally a large number of candidates, only promising candidates are generated. Promising candidates for a clique are sets whose elements (nodes) all have the minimum degree necessary for a clique (Thm. 4.2). Also, the highest node degree in the graph determines the largest possible clique size, such that larger clique candidates do not have to be generated. The algorithm stops when all cliques have been found (i.e., when the size of clique candidates enumerated reaches a number lower than the rank $k$ of the input graph).

It is clear that the complexity of this algorithm is very high such that it is not feasible for larger graphs (more than about 30 nodes), and will even take excessive time on dense $k$-hypergraphs with fewer nodes. For example, for a very dense 3-hypergraph with 30 nodes, the algorithm would have to check $\sum_{i=3}^{30} \binom{30}{i} \approx 1.1 \cdot 10^9$ sets for the clique property, where each check involves testing up to $\binom{30}{3} = 4060$ edges. The algorithm is feasible on sparse graphs since low node degrees mean that many larger cliques do not have to be generated.

```
ALGORITHM FIND_HYPERCLIQUES_BRUTE_FORCE

INPUT:
$k$-Uniform Hypergraph $G_k = (V, E)$

OUTPUT:
a set $result$, containing cliques in $G_k$

function findHypercliquesBruteForce(Graph $G_k$)
$result \leftarrow \emptyset$
$d \leftarrow$ maximum node degree in $G_k$
$s_{max} \leftarrow$ largest $s$ for which $\binom{s-1}{k-1} \leq d$    //maximum clique size
do
  $d_{min} \leftarrow \binom{s_{max}-1}{k-1}$    //min. node degree for clique with $s_{max}$ nodes
  if (fewer than $s_{max}$ nodes with degree $d_{min}$ in $G$)
    $s_{max} \leftarrow s_{max} - 1$
  else break
while ($s_{max} \geq k$)

for $s \leftarrow s_{max} \ldots k$   //for all clique sizes $\leq$ max.
  $d_{min} \leftarrow \binom{s-1}{k-1}$   //min. degree for this clique size
  $G^* \leftarrow$ induced subgraph of $G_k$ with all nodes with degree $\geq d_{min}$
  forall ($s$-subsets $G_s$ of $G^*$)
    if ($G_s$ is a clique in $G^*$)
      $result \leftarrow result \cup G_s$
return $result$
```

Figure B.1: Brute-Force Algorithm to Find Cliques in a $k$-Uniform Hypergraph

# Bibliography

[Abi97] S. Abiteboul. Querying semistructured data. In *Proceedings of the International Conference on Database Theory*, Delphi, Greece, January 1997.

[AESY97] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient View Maintenance at Data Warehouses. In *Proceedings of SIGMOD*, pages 417–427, 1997.

[AKS96] Y. Arens, C. A. Knoblock, and W.-M. Shen. Query Reformulation for Dynamic Information Integration. *Journal of Intelligent Information Systems*, 6 (2/3):99–130, 1996.

[ALM$^+$92] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof Verification and Intractability of Approximation Problems. In *Proc. 33rd IEEE Symp. on Foundations of Computer Science*, 1992.

[AMM97] P. Atzeni, G. Mecca, and P. Merialdo. Semistructured and Structured Data in the Web: Going Back and Forth. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(4):16 ff., 1997.

[AS94] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 12–15 September 1994.

[AY98] C. C. Aggarwal and P. S. Yu. Online Generation of Association Rules. In *Proceedings of IEEE International Conference on Data Engineering*, pages 402–411, 1998.

[BB95a] S. Bell and P. Brockhausen. Discovery of Data Dependencies in Relational Databases. In Y. Kodratoff, G. Nakhaeizadeh, and C. Taylor, editors, *ML-Net Familiarization Workshop*, 1995.

[BB95b] S. Bell and P. Brockhausen. Discovery of Data Dependencies in Relational Databases. Technical report, University of Dortmund, 1995.

[BBC$^+$00] D. Beneventano, S. Bergamaschi, S. Castano, et al. Information Integration: The MOMIS Project Demonstration. In *International Conference on Very Large Data Bases*, pages 611–614, 2000.

[BDT98] P. Buneman, A. Deutsch, and W. Tan. A deterministic model for semistructured data. In *Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, 1998.

[BEGK00] E. Boros, K. Elbassioni, V. Gurvich, and L. Khachiyan. An Incremental RNC Algorithm for Generating All Maximal Independent Sets in Hypergraphs of Bounded Dimension. Technical Report 47-2000, Rutgers University, 2000.

[Ber89] C. Berge. *Hypergraphs.* North-Holland, 1989.

[BK73] C. Bron and J. Kerbosch. Finding All Cliques of an Undirected Graph. *Communications of the ACM*, 16(9):575–577, September 1973.

[BLN86] C. Batini, M. Lenzerini, and S. Navathe. A Comparative Analysis of Methodologies of Database Schema Integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[BLT86] J. A. Blakeley, P.-E. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. *Proceedings of SIGMOD*, pages 61–71, 1986.

[BPSM97] E. T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible Markup Language (XML), 1997. http://www.w3.org/TR/PR-xml-971208.

[BRU97] P. Buneman, L. Raschid, and J. Ullman. Mediator Languages—a Proposal for a Standard. *SIGMOD Record*, March 1997.

[CCH91] Y. Cai, N. Cercone, and J. Han. Attribute-Oriented Induction in Relational Databases. In *Knowledge Discovery in Databases*, pages 213–228. AAAI Press, Menlo Park, California, 1991.

[CFP82] M. A. Casanova, R. Fagin, and C. H. Papadimitriou. Inclusion Dependencies and their Interaction with Functional Dependencies. In *Proceedings of ACM Conference on Principles of Database Systems (PODS)*, pages 171–176, 1982.

[Che76] P.-S. Chen. The Entity-Relationship Model—Towards a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[CK84] S. S. Cosmadakis and P. C. Kanellakis. Functional and Inclusion Dependencies: A graph-theoretic Approach. In ACM, editor, *Proceedings of ACM Symposium on Principles of Database Systems*, pages 29–37. ACM Press, 1984.

[CKV90] S. S. Cosmadakis, P. C. Kanellakis, and M. Y. Vardi. Polynomial-Time Implication Problems for Unary Inclusion Dependencies. *Journal of the ACM*, 37(1):15–46, January 1990.

[CKW89] W. Chen, M. Kifer, and D. Warren. HiLog as a Platform for Database Languages. *IEEE Data Eng. Bull.*, 12(3):37, September 1989.

[CMN98] S. Chaudhuri, R. Motwani, and V. Narasayya. Random Sampling for Histogram Construction: How much is enough? *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 27(2):436–447, 1998.

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *CACM*, 13(6):377–387, 1970.

[Coh98] W. W. Cohen. Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual/ Similarity. *SIGMOD Record*, 27(2):201–213, 1998.

[CR94] C. M. Chen and N. Roussopoulos. Adaptive Selectivity Estimation Using Query Feedback. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(2):161–172, June 1994.

[CS01] L. Crow and N. Shadbolt. Extracting Focused Knowledge from the Semantic Web. *International Journal of Human-Computer Studies*, 54:155–184, 2001.

[CSGM00] J. Cho, N. Shivakumar, and H. Garcia-Molina. Finding Replicated Web Collections. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(2):355–366, 2000.

[CT92] T. Catarci and L. Tarantino. Structure Modeling Hypergraphs: a Complete Representation for Databases. In *Database and Expert Systems Applications (DEXA)*, pages 314–319, 1992.

[CZC$^+$01] J. Chen, X. Zhang, S. Chen, A. Koeller, and E. A. Rundensteiner. DyDa: Data Warehouse Maintenance in Fully Concurrent Environments. In *Proceedings of SIGMOD'01. Demo Session*, page 619, Santa Barbara, California, May 2001.

[DDH00] A. Doan, P. Domingos, and A. Halevy. Learning Source Description for Data Integration. In *Proceedings of the Third International Workshop on the Web and Databases (WebDB)*, pages 81–86, Dallas, 2000.

[DP95] S. Dao and B. Perry. Applying a Data Miner To Heterogeneous Schema Integration. In *Knowledge Discovery and Data Mining*, pages 63–68, 1995.

[Dus97] O. M. Duschka. *Query Planning and Optimization in Information Integration*. PhD thesis, Stanford University, Stanford, California, December 1997.

[Eic91] C. F. Eick. A Methodology for the Design and Transformation of Conceptual Schemas. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *17th International Conference on Very Large Data Bases*, pages 25–34, Barcelona, Catalonia, Spain, 3–6 September 1991. Morgan Kaufmann.

[EN94] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[EW94] O. Etzioni and D. Weld. A Softbot-Based Interface to the Internet. *Communications of the ACM*, 37(7):72–76, July 1994.

[EWH85] R. Elmasri, J. Weeldreyer, and A. Hevner. The category concept: An extension to the entity-relationship model. *Data & Knowledge Engineering*, 1:75–116, 1985.

[Fag81] R. Fagin. A Normal Form for Relational Databases that is Based on Domains and Keys. *ACM Transactions on Database Systems (TODS)*, 6(3):387–415, 1981.

[Fay97] U. Fayyad. Knowledge Discovery in Databases: An Overview. In N. Lavrač and S. Džeroski, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *LNAI*, pages 3–16, Berlin, September 17–20 1997. Springer.

[FPSS96] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery in databases. *AI Magazine*, 17:37–54, 1996.

[FPSSU96] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors. *Advances in Knowledge Discovery and Data Mining*. AIII Press/MIT Press, March 1996.

[FS99] P. A. Flach and I. Savnik. Database dependency discovery: a machine learning approach. *AI Communications*, 12(3):139–160, November 1999.

[Gar98] S. R. Gardner. Building the Data Warehouse. *Communications of the ACM*, 41(9):52–60, 1998.

[GGL95] R. L. Graham, M. Grötschel, and L. Lovász, editors. *Handbook of Combinatorics*. Elsevier/MIT Press, 1995.

[GGMS96] S. Ganguly, P. B. Gibbons, Y. Matias, and A. Silberschatz. Bifocal sampling for skew-resistant join size estimation. *SIGMOD Record*, 25(2):271–281, June 1996.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.

[GKD97] M. R. Genesereth, A. M. Keller, and O. M. Duschka. Infomaster: An Information Integration System. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(2):539ff., 1997.

[GKL96] O. Garrido, P. Kelsen, and A. Lingas. A Simple NC-Algorithm for a Maximal Independent Set in a Hypergraph of Poly-Log Arboricity. *Information Processing Letters*, 58:55–58, 1996.

[GL95] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *Proceedings of SIGMOD*, pages 328–339, 1995.

[GLS96] M. Gyssens, L. V. S. Lakshmanan, and I. N. Subramanian. Tables as a Paradigm for Querying and Restructuring (extended abstract). In ACM, editor, *Proceedings of ACM Symposium on Principles of Database Systems*, volume 15, pages 93–103, New York, NY 10036, USA, 1996. ACM Press.

[GLS$^+$97] F. Gingras, L. Lakshmanan, I. N. Subramanian, D. Papoulis, and N. Shiri. Languages for Multi-Database Interoperability. In J. M. Peckman, editor, *Proceedings of SIGMOD*, volume 26(2) of *SIGMOD Record*, pages 536–538, 1997.

[GMHI$^+$95] H. García-Molina, J. Hammer, K. Ireland, et al. Integrating and Accessing Heterogeneous Information Sources in TSIMMIS. In *AAAI Spring Symposium on Information Gathering*, 1995.

[GMLY99] H. García-Molina, W. Labio, and R. Yerneni. Capability Sensitive Query Processing on Internet Sources. In *Proceedings of the 15th International Conference on Data Engineering*, Sydney, Australia, March 1999. Accessible at `http://www-db.-stanford.edu/`.

[GPB99] A. Gómez-Pérez and V. Benjamins. Applications of Ontologies and Problem-Solving Methods. *AI Magazine*, 20(1):119–122, 1999.

[GRVB98] J.-R. Gruser, L. Raschid, M. E. Vidal, and L. Bright. Wrapper Generation for Web-Accessible Data Sources. In *6th Int. Conf. on Cooperative Information Systems*, pages 14–23, New York, 1998.

[Gua95] N. Guarino. Formal Ontology, Conceptual Analysis and Knowledge Representation. *International Journal of Human-Computer Studies*, 43(5/6):625–640, 1995.

[Han87] E. N. Hanson. A Performance Analysis of View Materialization Strategies. In *Proceedings of SIGMOD*, pages 440–453, 1987.

[HGMN$^+$97] J. Hammer, H. García-Molina, S. Nestorov, R. Yerneni, M. Breunig, and V. Vassalos. Template-Based Wrappers in the TSIMMIS System. In *Proceedings of SIGMOD*, pages 532–535, 1997.

[HK87] R. Hull and R. King. Semantic Database Modelling: Survey, Applications and Research Issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[HKPT98] Y. Huhtala, J. Karkkainen, P. Porkka, and H. Toivonen. Efficient discovery of functional and approximate dependencies using partitions. In *Proceedings of IEEE International Conference on Data Engineering*, pages 392–401, 1998.

[HNSS93] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami. Fixed-Precision Estimation of Join Selectivity. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 190–201. ACM Press, May 1993.

[HNSS95] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes. Sampling-Based Estimation of the Number of Distinct Values of an Attribute. In *International Conference on Very Large Data Bases*, pages 311–322, 1995.

[HÖ91] W.-C. Hou and G. Özsoyoğlu. Statistical Estimators for Aggregate Relational Algebra Queries. *ACM Transactions on Database Systems*, 16(4):600–654, December 1991.

[HS95] P. J. Haas and A. N. Swami. Sampling-based Selectivity Estimation for Joins Using Augmented Frequent Value Statistics. In *Proceedings of IEEE International Conference on Data Engineering*, pages 522–531, 1995.

[Hul86] R. Hull. Relative information capacity of simple relational database schemata. *SIAM Journal on Computing*, 15(3):856–886, 1986.

[Hyl96] J. A. Hylton. Identifying and Merging Related Bibliographical Records. Master's thesis, MIT, Dept. of EECS, 1996. MIT ME-EECS.

[HZ96] R. Hull and G. Zhou. A framework for supporting data integration using the materialized and virtual approaches. In *SIGMOD*, pages 481–492, Montreal, Canada, 1996. ACM Press.

[HZZ93] W.-C. Hon, Z. Zhang, and N. Zhou. Statistical Inference of Unknown Attribute Values in Databases. In B. Bhargava, T. Finin, and Y. Yesha, editors, *Proceedings of International Conference on Information and Knowledge Management*, pages 21–30, New York, NY, USA, November 1993. ACM Press.

[JL96] G. H. John and P. Langley. Static Versus Dynamic Sampling for Data Mining. In E. Simoudis, J. Han, and U. M. Fayyad, editors, *Proc. 2nd Int. Conf. Knowledge Discovery and Data Mining, KDD*, pages 367–370. AAAI Press, 2–4 August 1996.

[KA96] A. J. Knobbe and P. W. Adriaans. Discovering Foreign Key Relations in Relational Databases. In R. Trappl, editor, *Proceedings of the Thirteenth European Meeting on Cybernetics and Systems Research*, volume 2, pages 961–966, Vienna, Austria, 1996. Austrian Soc. Cybernetic Studies, Vienna, Austria.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In M. Stonebraker, editor, *Proceedings of SIGMOD*, volume 21(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 393–402, New York, NY 10036, USA, 1992. ACM Press.

[KLK91] R. Krishnamurthy, W. Litwin, and W. Kent. Language features for interoperability of databases with schematic discrepancies. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 20(2):40–49, June 1991.

[KLN00] J. Kang, M. L. Lee, and J. F. Naughton. IDB: Toward the Scalable Integration of Queryable Internet Data Sources. University of Wisconsin—Madison, 2000.

[KLSS95] T. Kirk, A. Y. Levy, Y. Sagiv, and D. Srivastava. The Information Manifold. *Proceedings of the AAAI Spring Symposium on Information Gathering in Distributed Heterogeneous Environments, Stanford, California*, March 1995.

[Klu95] M. Klusch. Cooperative Recognition of Interdatabase Dependencies. In *Proceedings of the Second International Workshop on Advances in Databases and Information Systems — ADBIS'95*, pages 135–139, Moscow, June 27–30 1995.

[KLZ$^+$97] A. Koeller, Y. Li, X. Zhang, A. Lee, A.Nica, and E. Rundensteiner. *Evolvable View Environment (EVE): Maintaining Views over Dynamic Distributed Information Sources*. Centre for Advanced Studies Conference, November 1997.

[KM94] J. Kivinen and H. Mannila. The Power of Sampling in Knowledge Discovery. In *Proceedings of ACM Symposium on Principles of Database Systems*, volume 13, pages 77–85, 1994.

[KM95] J. Kivinen and H. Mannila. Approximate inference of functional dependencies from relations. *Theoretical Computer Science*, 149(1):129–149, 18 September 1995.

[KMRS92] M. Kantola, H. Mannila, K. J. Räihä, and H. Siirtola. Discovering functional and inclusion dependencies in relational databases. *International J. Of Intelligent Systems*, 7:591–607, 1992.

[KR98] H. A. Kuno and E. A. Rundensteiner. Incremental maintenance of materialized object-oriented views in *MultiView*: Strategies and performance evaluation. *IEEE Transaction on Data and Knowledge Engineering*, 10(5):768–792, Sept/Oct. 1998.

[KR99] A. Koeller and E. A. Rundensteiner. View Synchronization: Using an Integrated Approach of Rewriting and Ranking View Queries. *Journal of Computer Science and Information Management*, 2(1), 1999.

[KR00] A. Koeller and E. A. Rundensteiner. History-Driven View Synchronization. In *Proceedings of 2nd International Conference on Data Warehousing and Knowledge Discovery (DaWaK)*, Lecture Notes in Computer Science 1874, pages 168–177, Greenwich, UK, September 2000. Springer Verlag.

[KR01] A. Koeller and E. A. Rundensteiner. A History-Driven Approach at Evolving Views Under Meta Data Changes. *submitted to Journal for Information Systems*, 2001.

[KRH98] A. Koeller, E. A. Rundensteiner, and N. Hachem. Integrating the Rewriting and Ranking Phases of View Synchronization. In *Proceedings of the ACM First International Workshop on Data Warehousing and OLAP (DOLAP'98)*, pages 60–65, November 1998.

[LAC$^+$93] M. Loomis, T. Atwood, R. Cattell, J. Duhl, G. Ferran, and D. Wade. The ODMG object model. *Journal of Object Oriented Programming*, pages 64–69, June 1993.

[LAZ$^+$89] W. Litwin, A. Abdellatif, A. Zeroual, B. Nicolas, and P. Vigier. MSQL: A Multidatabase Language. *Information Sciences*, 49(1), 1989.

[LC94] W.-S. Li and C. Clifton. Semantic Integration in Heterogeneous Databases Using Neural Networks. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *International Conference on Very Large Data Bases*, pages 1–12, 1994.

[LC95] W.-S. Li and C. Clifton. SemInt: a system prototype for semantic integration in heterogeneous databases. In *Proceedings of SIGMOD*, volume 24(2) of *SIGMOD Record*, pages 484–484, 1995.

[LC00] W. Li and C. Clifton. SemInt: A tool for identifying attribute correspondences in heterogeneous databases using neural networks. *Data and Knowledge Engineering*, 33(1):49–84, 2000.

[LH97] W. Lim and J. Harrison. Discovery of Constraints from Data for Information System Reverse Engineering. In *Proc. of Australian Software Engineering Conference (ASWEC '97)*, Sydney, Australia, Sep 28–Oct 2 1997.

[Lin76] B. W. Lindgren. *Statistical Theory.* Macmillan Publishing Co., Inc., New York, 3rd edition, 1976.

[LKNR98] A. J. Lee, A. Koeller, A. Nica, and E. A. Rundensteiner. Data Warehouse Evolution: Trade-offs between Quality and Cost of Query Rewritings. Technical Report WPI-CS-TR-98-2, revised in 1999., Worcester Polytechnic Institute, Dept. of Computer Science, 1998.

[LKNR99a] A. J. Lee, A. Koeller, A. Nica, and E. A. Rundensteiner. Data Warehouse Evolution: Trade-offs between Quality and Cost of Query Rewritings. In *Proceedings of IEEE International Conference on Data Engineering*, Special Poster Session, page 255, March, Sydney, Australia 1999.

[LKNR99b] A. J. Lee, A. Koeller, A. Nica, and E. A. Rundensteiner. Non-Equivalent Query Rewritings. In *Proceedings of the 9th International Databases Conference*, pages 248–262. City University of Hong Kong Press, Hong Kong, July 1999.

[LLY98] H. Liu, H. Lu, and J. Yao. Identifying Relevant Databases for Multidatabase Mining. *Lecture Notes in Computer Science*, 1394:210–221, 1998.

[LNE89] J. A. Larson, S. B. Navathe, and R. Elmasri. A Theory of Attribute Equivalence in Databases with Applications to Schema Integration. *IEEE Transactions on Software Engineering*, 15(4):449–463, April 1989.

[LNR97] A. J. Lee, A. Nica, and E. A. Rundensteiner. The EVE Framework: View Synchronization in Evolving Environments. Technical Report WPI-CS-TR-97-4, Worcester Polytechnic Institute, Dept. of Computer Science, 1997.

[LNR01] A. J. Lee, A. Nica, and E. A. Rundensteiner. The *EVE* Approach: View Synchronization In Dynamic Distributed Environments. *IEEE Transaction on Knowledge and Data Engineering*, Accepted 2001. To Appear.

[LNS90] R. J. Lipton, J. F. Naughton, and D. A. Schneider. Practical selectivity estimation through adaptive sampling. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 19(2):1–11, June 1990.

[LP01] M. J. LaPadula and M. E. Picollelli. On Redundant and Non-Semi-Intersecting Hypergraphs. DIMACS—Center for Discrete Mathematics and Theoretical Computer Science, 2001.

[LSK95] A. Y. Levy, D. Srivastava, and T. Kirk. Data model and query evaluation in global information systems. *Journal of Intelligent Information Systems - Special Issue on Networked Information Discovery and Retrieval*, 5(2):121–143, 1995.

[LSS96] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. SchemaSQL — A Language for Interoperability in Relational Multi-database Systems. In T. M. Vijayaraman et al., editors, *International Conference on Very Large Data Bases*, pages 239–250, Mumbai, India, Sept. 1996.

[LSS99] L. V. S. Lakshmanan, F. Sadri, and S. N. Subramanian. On Efficiently Implementing SchemaSQL on an SQL Database System. In *International Conference on Very Large Data Bases*, pages 471–482, 1999.

[LV00] M. Levene and M. W. Vincent. Justification for Inclusion Dependency Normal Form. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 12, 2000.

[LYV$^+$98] C. Li, R. Yerneni, V. Vassalos, H. García-Molina, Y. Papakonstantinou, J. Ullman, and M. Valiveti. Capability Based Mediation in TSIMMIS. In *Proceedings of SIGMOD*, pages 564–566, 1998.

[MC72] G. D. Mulligan and D. G. Corneil. Corrections to Bierstone's Algorithm for Generating Cliques. *Journal of the ACM*, 19(2):244–247, April 1972.

[MCS88] M. V. Mannino, P. Chu, and T. Sager. Statistical Profile Estimation in Database Systems. *ACM Computing, Springer Verlag (Heidelberg, FRG and NewYork NY, USA)-Verlag Surveys*, 20(3), September 1988.

[MG90] R. Missaoui and R. Godin. The Implication Problem for Inclusion Dependences: A Graph Approach. *SIGMOD Record*, 19(1):36–40, March 1990.

[MHH00] R. J. Miller, L. M. Haas, and M. A. Hernández. Schema Mapping as Query Discovery. In *International Conference on Very Large Data Bases*, pages 77–88, 2000.

[Min01] T. P. Minka. *Bayesian Inference, Entropy, and the Multinomial Distribution.* MIT, http://www-white.media.mit.edu/~tpminka/papers/learning.html, February 2001. Tutorial.

[MIR93] R. J. Miller, Y. Ioannidis, and R. Ramakrishnan. The Use of Information Capacity in Schema Integration and Translation. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases (VLDB)*, pages 120–133, Dublin, Ireland, August 1993.

[MIR94] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. Schema Intension Graphs: A Formal Model for the Study of Schema Equivalence. Technical Report CS-TR-1994-1185, University of Wisconsin, Madison, January 1994.

[Mit83] J. C. Mitchell. Inference Rules for Functional and Inclusion Dependencies. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 58–69, Atlanta, Georgia, 21–23 March 1983.

[MKK97] M. K. Mohania, S. Konomi, and Y. Kambayashi. Incremental Maintenance of Materialized Views. In *Database and Expert Systems Applications (DEXA)*, pages 551–560, 1997.

[MM65] J. W. Moon and L. Moser. On Cliques in Graphs. *Israel Journal of Mathematics*, 3:23–28, 1965.

[MR94] H. Mannila and K. J. Raiha. Algorithms for inferring functional-dependencies from relations. *Data & Knowledge Engineering*, 12:83–99, 1994.

[MTL00] D. P. Makawita, K.-L. Tan, and H. Liu. Sampling from Databases Using $B^+$-Trees. In A. Agah, J. Callan, and E. Rundensteiner, editors, *Proceedings of International Conference on Information and Knowledge Management*, pages 158–164, November 6–11 2000.

[MWJ99] P. Mitra, G. Wiederhold, and J. Jannink. Semi-automatic integration of knowledge sources. In *Proc. of the 2nd Int. Conf. On Information Fusion (FUSION'99)*, Sunnyvale, California, 1999.

[Nic99] A. Nica. *View Evolution Support for Information Integration Systems over Dynamic Distributed Information Spaces.* PhD thesis, University of Michigan in Ann Arbor, in progress 1999.

[NLR98] A. Nica, A. J. Lee, and E. A. Rundensteiner. The CVS Algorithm for View Synchronization in Evolvable Large-Scale Information Systems. In *Proceedings of International Conference on Extending Database Technology (EDBT'98)*, pages 359–373, Valencia, Spain, March 1998.

[NR97a] A. Nica and E. A. Rundensteiner. Loosely-Specified Query Processing in Large-Scale Information Systems. *International Journal of Cooperative Information Systems*, 1997.

[NR97b] A. Nica and E. A. Rundensteiner. On Translating Loosely-Specified Queries into Executable Plans in Large-Scale Information Systems. In *Proceedings of Second IFCIS International Conference on Cooperative Information Systems CoopIS'97*, pages 213–222, June 1997.

[NR98] A. Nica and E. A. Rundensteiner. Using Containment Information for View Evolution in Dynamic Distributed Environments. In *Proceedings of International Workshop on Data Warehouse Design and OLAP Technology (DWDOT'98)*, Vienna, Austria, August 1998.

[NR99] A. Nica and E. A. Rundensteiner. View Maintenance after View Synchronization. In *International Database Engineering and Applications Symposium (IDEAS'99)*, pages 213–215, August, Montreal, Canada 1999.

[OAE00] K. O'Gorman, D. Agrawal, and A. El Abbadi. On the Importance of Tuning in Incremental View Maintenance: An Experience Case Study (Extended Abstract). In *Proceedings of the Second International Conference on Data Warehousing and Knowledge Discovery (DaWaK)*, pages 77–82, September 2000.

[OR94] F. Olken and D. Rotem. Random Sampling from Databases — A Survey. Technical report, Information and Computing Sciences Division, Lawrence Berkeley National Laboratory, Berkeley, California, 22 March 1994.

[PBE95] E. Pitoura, O. Bukhres, and A. Elmagarmid. Object Orientation in Multidatabase Systems. *ACM Computing Surveys*, 27(2):141–195, June 1995.

[PF91] G. Piatetsky-Shapiro and W. J. Frawley, editors. *Knowledge Discovery in Databases.* American Association for Artificial Intelligence, Menlo Park, California, U.S.A., 1991.

[PI97] V. Poosala and Y. E. Ioannidis. Selectivity Estimation Without the Attribute Value Independence Assumption. In *International Conference on Very Large Data Bases*, pages 486–495, 1997.

[PIHS96] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. *SIGMOD Record*, 25(2):294–305, June 1996.

[PSC84] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. *SIGMOD Record*, 14(2):256–276, 1984.

[PSU98] L. Palopoli, D. Saccà, and D. Ursino. Semi-automatic, semantic discovery of properties from database schemes. In *International Database Engineering and Application Symposium*, pages 244–253, 1998.

[PX94] P. Pardalos and J. Xue. The Maximum-Clique Problem. *Journal of Global Optimization*, 4:301–328, 1994.

[QW91] X. Qian and G. Wiederhold. Incremental Recomputation of Active Relational Expressions. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 3(3):337–341, September 1991.

[Ram97] R. Ramakrishnan. *Database Management Systems.* WCB/McGraw-Hill, 1997.

[RB01] E. Rahm and P. A. Bernstein. On Matching Schemas Automatically. Technical Report MSR-TR-2001-17, Microsoft Research, `http://www.research.microsoft.com`, feb 2001.

[RCR96] J. F. Roddick, N. G. Craske, and T. J. Richards. Handling discovered structure in database-systems. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 8:227–240, 1996.

[Ric95] J. A. Rice. *Mathematical Statistics and Data Analysis.* Duxbury Press, 2nd edition, 1995.

[RKL$^+$98] E. A. Rundensteiner, A. Koeller, A. Lee, Y. Li, A. Nica, and X. Zhang. Evolvable View Environment (*EVE*) Project: Synchronizing Views over Dynamic Distributed Information Sources. In *Demo Session Proceedings of International Conference on Extending Database Technology (EDBT'98)*, pages 41–42, Valencia, Spain, March 1998.

[RKZ$^+$99] E. A. Rundensteiner, A. Koeller, X. Zhang, A. Lee, A. Nica, A. VanWyk, and Y. Li. Evolvable View Environment. In *Proceedings of SIGMOD'99 Demo Session*, pages 553–555, May 1999.

[RLN97] E. A. Rundensteiner, A. J. Lee, and A. Nica. On Preserving Views in Evolving Environments. In *Proceedings of 4th Int. Workshop on Knowledge Representation Meets Databases (KRDB'97): Intelligent Access to Heterogeneous Information*, pages 13.1–13.11, Athens, Greece, August 1997.

[RN95] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1995.

[Rou98] N. Roussopoulos. Materialized Views and Data Warehouses. *SIGMOD Record*, 27(1):21–26, 1998.

[RS97] M. T. Roth and P. M. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *International Conference on Very Large Data Bases*, pages 266–275, 1997.

[SBF98] R. Studer, R. Benjamins, and D. Fensel. Knowledge Engineering: Principles and Methods. *Data and Knowledge Engineering*, 25:161–197, 1998.

[SF93] I. Savnik and P. A. Flach. Bottom-up induction of functional dependencies from relations. In G. Piatetsky-Shapiro, editor, *Proc. of AAAI-93 Workshop: Knowledge Discovery in Databases*, pages 174–185, July 1993.

[Ski97] S. S. Skiena. *The Algorithm Design Manual.* Springer-Verlag, 1997.

[SL90] A. P. Sheth and J. A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous

Databases. *ACM Computing Surveys*, 22(3):183–236, September 1990. Also published in/as: Bellcore, TM-STS-016302, Jun.1990.

[SLCN88] A. P. Sheth, J. A. Larson, A. Cornelio, and S. B. Navathe. A Tool for Integrating Conceptual Schemas and User Views. In *Proceedings of IEEE International Conference on Data Engineering*. IEEE, 1988.

[STA98] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 27(2):343–354, 1998.

[TAH+96] M. Tork Roth, M. Arya, L. M. Haas, M. J. Carey, W. Cody, R. Fagin, P. M. Schwarz, J. Thomas, and E. L. Wimmers. The Garlic Project. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(2):557 ff., 1996.

[Toi96] H. Toivonen. Sampling Large Databases for Association Rules. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases*, pages 134–145, Mumbai (Bombay), India, 3–6 September 1996. Morgan Kaufmann.

[TRV96] A. Tomasic, L. Raschid, and P. Valduriez. Scaling Heterogeneous Databases and the Design of Disco. In *International Conference Distributed Computing Systems*, pages 449–457, May 1996.

[TUA+98] D. Tsur, J. D. Ullman, S. Abiteboul, C. Clifton, R. Motwani, S. Nestorov, and A. Rosenthal. Query flocks: a generalization of association-rule mining. In *Proceedings of SIGMOD*, volume 27(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 1–12, New York, NY 10036, USA, 1998. ACM Press.

[UG96] M. Uschold and M. Grüninger. Ontologies: Principles, Methods and Applications. *Knowledge Engineering Review*, 1(2):93–155, June 1996.

[Ukk92] E. Ukkonen. Approximate String-Matching with $q$-grams and Maximal Matches. *Journal of Theoretical Computer Science*, 92:191–211, 1992.

[Ull89] J. Ullman. *Principle of Database and Knowledge-Base Systems.* Computer Science Press, 1989.

[Urs99] D. Ursino. *Semi-Automatic Approaches and Tools for the Extraction and the Exploitation of Intensional Knowledge from Heterogeneous Information Sources.* PhD thesis, Università degli Studi della Calabria, Cosenza, Italy, 1999.

[W3C01] W3C. XQuery: A Query Language for XML. http://www.w3.org/TR/xquery/, February 2001.

[Wie92] G. Wiederhold. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(2):38–49, 1992.

[WM97] D. Wilson and T. R. Martinez. Improved Heterogeneous Distance Functions. *Journal of Artificial Intelligence Research*, 6:1–34, 1997.

[Woo97] D. R. Wood. An algorithm for finding a maximum clique in a graph. *Operations Research Letters*, 21:211–217, 1997.

[WW96] Y. Wang and A. K. C. Wong. Representing Discovered Patterns Using Attributed Hypergraph. In *International Conference on Knowledge Discovery and Data Mining*, pages 283–286, 1996.

[YJSD91] C. Yu, B. Jia, W. Sun, and S. Dao. Determining relationships among names in heterogeneous databases. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 20(4):79–80, December 1991.

[Zak00] M. J. Zaki. Scalable Algorithms for Association Mining. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 12(3):372–390, May/June 2000.

[ZGMHW95] Y. Zhuge, H. García-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *Proceedings of SIGMOD*, pages 316–327, May 1995.

[ZGMW96] Y. Zhuge, H. García-Molina, and J. L. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. In *International Conference on Parallel and Distributed Information Systems*, pages 146–157, December 1996.

[ZOPL96] M. J. Zaki, M. Ogihara, S. Parthasarathy, and W. Li. Parallel Data Mining for Association Rules on Shared-memory Multiprocessors. In *Supercomputing '96 Conference Proceedings: November 17–22, Pittsburgh, PA*, 1996.

[ZPLO97] M. J. Zaki, S. Parthasarathy, W. Li, and M. Ogihara. Evaluation of Sampling for Data Mining of Association Rules. In *Proceedings of the Seventh International Workshop on Research Issues in Data Engineering (RIDE'97)*, pages 42–50. IEEE, April 1997.

[ZR99] X. Zhang and E. A. Rundensteiner. Flexible Data Warehouse Maintenance Under Concurrent Schema and Data Updates. In *Proceedings of IEEE International Conference on Data Engineering*, Special Poster Session, page 253, March, Sydney, Australia 1999.

[ZRD01] X. Zhang, E. A. Rundensteiner, and L. Ding. PVM: Parallel View Maintenance Under Concurrent Data Updates of Distributed Sources. In *Data Warehousing and Knowledge Discovery, Proceedings*, Munich, Germany, September 2001. 230–239.