Efficient and exact computation of inclusion dependencies for data integration

Jana Bauckmann
Hasso-Plattner-Institut
Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany
jana.bauckmann@hpi.uni-potsdam.de

Ulf Leser
Humboldt-Universität zu Berlin,
Unter den Linden 6, 10099 Berlin, Germany
leser@informatik.hu-berlin.de

Felix Naumann
Hasso-Plattner-Institut
Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany
naumann@hpi.uni-potsdam.de

Abstract

Data obtained from foreign data sources often come with only superficial structural information, such as relation names and attribute names. Other types of metadata that are important for effective integration and meaningful querying of such data sets are missing. In particular, relationships among attributes, such as foreign keys, are crucial metadata for understanding the structure of an unknown database. The discovery of such relationships is difficult, because in principle for each pair of attributes in the database each pair of data values must be compared.

A precondition for a foreign key is an inclusion dependency (IND) between the key and the foreign key attributes. We present with SPIDER an algorithm that efficiently finds all INDs in a given relational database. It leverages the sorting facilities of DBMS but performs the actual comparisons outside of the database to save computation. SPIDER analyzes very large databases up to an order of magnitude faster than previous approaches. We also evaluate in detail the effectiveness of several heuristics to reduce the number of necessary comparisons. Furthermore, we generalize SPIDER to find composite INDs covering multiple attributes, and partial INDs, which are true INDs for all but a certain number of values. This last type is particularly relevant when integrating dirty data as is often the case in the life sciences domain – our driving motivation.

Chapter 1

Schema Discovery for Data Integration

In large integration projects one is often confronted with poorly documented data sources, i. e., data sources for which the schema is under-specified. However, effective integration requires knowledge of metadata beyond mere attribute names. In particular, keys and foreign keys are useful structural information. The use of such constraints is standard in commercial applications, but not in scientific applications. Even in projects that use relational databases, foreign keys are often not defined. The reasons include lacking knowledge of foreign keys, laziness of the developer, technical limitations of the particular DBMS, or simply because the data are dirty and constraints do not entirely hold. For instance, many life science databases run on MySQL, which until very recently had only very limited support for foreign keys. Also, in many scientific and business domains there is a continuing belief that checking constraints has a considerable negative impact on database operations. Thus, even extremely successful systems, such as the Ensembl database with hundreds of tables, are delivered without foreign keys [HBB+02].

The example we mostly use in this article is the important Protein Data Bank (PDB), a publicly available database containing essentially all known protein 3D-structures. PDB is distributed as a structured flat-file and can be imported into a relational schema using the OpenMMS software[1]. The OpenMMS schema consists of 176 tables with 2,713 attributes but does not specify a single foreign key constraint. This under-specification hinders the integration of this database considerably [TRM+05].

The goal of the research presented in this article is to automatically and efficiently detect inclusion dependencies (IND) as a precondition for foreign key constraints. An inclusion dependency $A \subseteq B$ between two attributes A and B (a candidate pair) is satisfied iff the set of values of A is a subset of the set of values of attribute B. Assuming that B contains only unique data values, this relationship constitutes the syntactically testable part of a foreign key. To test whether A is in fact a foreign key or merely a spurious IND requires additional knowledge.

An IND $A \subseteq B$ between a candidate pair (A, B) can be tested by checking for each value of A whether it appears as a value of B. This test can be performed in various ways by programs or SQL queries, as suggested in our previous work [BLN06] and in related

[1] `openmms.sdsc.edu`

work [BB95, Koe01]. However, as soon as a check fails for some value, the test for that candidate pair can be aborted. One key insight of our research is the inability of relational database systems and SQL to perform this early stop efficiently (or at all), which renders them inefficient for detecting INDs [BLN06]. To compensate, our SPIDER algorithm sorts all columns in the database, exports them into the file system and then checks for INDs in all candidate pairs *simultaneously*. The salient characteristics of SPIDER are:

- Concurrent tests of all candidate pairs, synchronized in a specialized data structure.
- The test of a candidate pair stops after the first counter-example is found.
- The number of value comparisons is greatly reduced, compared to approaches testing all candidate pairs sequentially.
- After exporting the data from the database, each value is read at most once to test all candidate pairs.

We prove that the complexity of our method is superior to previous approaches and give experimental results showing that our approach places automatic foreign key discovery into the realm of the feasible, even for very large databases with many attributes and many tuples. For instance, SPIDER analyzes a 2.8 GB database with about 1,200 attributes in $\sim$ 24 min, a 32 GB database in $\sim$ 6 h, and a $\sim$ 40 GB excerpt from an SAP/R3 installation covering about 23,000 attributes in $\sim$ 2 h.

Because the detection of unary INDs is only one step in discovering structure in unknown databases, we also describe several extensions of SPIDER:

- **Composite INDs:** Foreign keys are often defined on sets of attributes. We generalize SPIDER to detect composite inclusion dependencies, reflecting composite keys and composite foreign keys. This modification requires to test for all lists of attributes $A_1, \ldots, A_k$ and attributes $B_1, \ldots, B_k$ of two relations whether $(A_1, \ldots, A_k) \subseteq (B_1, \ldots, B_k)$. Obviously, there exists an exponential number of such candidates. We show how the level-wise approach to candidate enumeration from [MLP09] (where a level is the number of dependent attributes in a candidate) can be combined with SPIDER, thus leveraging its speed for candidate testing. We also discuss heuristics that reduce the number of IND candidates enormously without excluding too many potential foreign keys.
- **Partial INDs:** We extend SPIDER to detect partial inclusion dependencies, i. e., INDs that are satisfied for all but a certain number of values. Partial INDs are important to handle dirty data, which frequently occur when foreign key constraints are not defined or disabled. A partial IND cannot be immediately turned into a foreign key constraint but may still provide valuable information about the data. Experiments show that the modified SPIDER algorithm detects partial INDs at almost no additional cost compared to exact IND detection.

Finally, we evaluate the identified INDs in respect to real foreign keys.

This article significantly improves and extends previous work in [BLN06] and [BLNT07]. In [BLN06] we discussed several ways of solving the IND detection problem using SQL and presented a first, yet suboptimal database-external algorithm. We very briefly recap those results in Section 2.1. SPIDER is a new algorithm, which tests all IND candidates in parallel and improves on [BLN06] (and on other related work) by an order of magnitude for large databases. Some experimental results for SPIDER were also presented in a poster [BLNT07]. In this article we describe the algorithm in detail, analyze its complexity, discuss several new pruning strategies, provide a comprehensive experimental evaluation and comparison with previous methods, and describe and evaluate its extensions towards composite INDs and partial INDs.

1.1 Heterogeneous life science data sources

The motivation for our research is rooted in the data integration problem in the life sciences domain. The Aladin project strives to achieve semi-automatic integration of such databases in several steps [LN05]. In the first step an expert chooses the new database that is to be included in the system. Most life sciences databases are either provided in a relational form or parsers are freely available to generate that form. But, as stated before, this relational form often lacks detailed metadata, such as data types, keys, or foreign keys. Following this manual choice of a data source we have devised a fully automatic procedure to integrate life sciences databases. Clearly, one cannot expect error-free integration, but for many scenarios manual integration and curation is too expensive making an explorative approach attractive.

In this setting, we utilize IND detection in two ways: (i) Identification of intra-source relationships to extract implicitly given structural information; and (ii) identification of inter-source relationships to find links between databases. These are usually stored as "foreign object IDs" in the referencing database and could be leveraged as life science databases heavily cross-reference each other [HK04]. Note that we need to find exact INDs in our scenario to limit errors in the entire process at this low level. We discuss several approaches to the approximate detection of INDs and approaches pruning the search space (e. g. by data types) in Chapter 8, but want to re-emphasize that these solve a different problem and they are not the optimal solution for our problem setting.

1.2 Data sets

We test the algorithms described in this article on real and synthetic databases from different domains (see Table 1.1). First we use databases of the life science domain: *UniProt* is a database of annotated protein sequences available in several formats [BAW+05]. We chose the BioSQL[2] schema and parser for UniProt, creating a database of 16 non-empty tables with 68 non-empty attributes. The total size of the database is 900 MB, with the

[2] obda.open-bio.org

largest attribute having approximately 1 million distinct values. Next, *PDB* is a large database of protein structures [BWF+00]. We used the OpenMMS software to parse PDB files into a relational database. PDB populates 116 tables with 1,297 non-empty attributes in the OpenMMS schema, i. e., only a part of the complete OpenMMS schema with 176 tables is used. No foreign keys are specified. The total database size is 32 GB, with the largest attribute having approximately 152 million different values. To achieve a better idea of the scalability of our approach, we also used a 2.8 GB fraction of the PDB, obtained by removing some extremely large tables.

	UniProt	TPC-H	PDB		SAP/R3
DB size	900 MB	1.3 GB	2.8 GB	32 GB	145 GB
# tables	16	8	110	116	25,002
# attributes	68	61	1,215	1,297	237,836

Table 1.1: Used data sets and their characteristics.

To test our algorithm on very large databases with enormous schemata, we use an SAP/R3 database instance[3], which populates 25,002 non-empty tables with 237,836 attributes. The total size is 145 GB. Finally, we verify our results by using a generated *TPC-H* database (scale factor 1.0)[4].

1.3 Structure and contributions

The following chapter describes several approaches to the discovery of unary inclusion dependencies, and in particular the SPIDER algorithm, which requires only sorting, exporting, and a single scan of the data to test *all* IND candidates while additionally decreasing the number of necessary comparisons. Chapter 3 presents filter techniques to exclude IND candidates. In Chapter 4 we present experimental results and compare SPIDER to other approaches from the literature. Chapters 5 and 6 present extensions of SPIDER to find composite INDs and partial INDs, respectively. In Chapter 7 we discuss the quality of the detected INDs concerning foreign keys. Related work is discussed in Chapter 8 and we conclude in Chapter 9.

[3]We thank the SAP HCC at Otto-von-Guericke-Universität Magdeburg for providing this database instance.

[4]Database sizes differ slightly from previous work in [BLN06] and [BLNT07] due to new versions of the DBMS and OpenMMS parser.

Chapter 2

Detecting Unary INDs

A unary inclusion dependency (IND) $A \subseteq B$ requires inclusion of the set of values of the *dependent* attribute A in the set of values of the *referenced* attribute B. Obviously, the IND relation is not symmetric. We call any pair of attributes A and B an IND candidate. An IND is *satisfied* if the IND requirements are met and *unsatisfied* otherwise. An attribute is *covered by an IND (candidate)* if it is part of that IND (candidate) as dependent or as referenced attribute. For complexity analysis, we consider a schema with n attributes and maximally t values in each attribute.

Strictly speaking, INDs cannot be discovered: One can merely verify whether they are satisfied for a given data set. For sake of clarity we ignore this fine distinction throughout the article.

2.1 Using SQL

One way to implement IND tests is to use SQL [BLN06, BB95, Koe01]. In this case each IND candidate is tested independently of all other tests using a specific query. We tested the performance of several possible SQL statements for IND tests in [BLN06], i. e., `join`, `minus`, and `not in`. The fastest approach was to join the IND candidate's attributes and to compare the join cardinality with the dependent attribute's cardinality. We repeat the complexity analysis of [KMRS92] and provide a proof assuming a sort merge join on both attributes:

Theorem 1. *The number of comparisons to test all IND candidates using one join query per IND candidate is $O(n^2 t \log t)$ assuming a sort merge join on both attributes.*

Proof. We need $t \log t$ comparisons to sort an attribute's values and $O(t)$ comparisons to merge two attributes. Thus, for the $n(n-1)$ IND candidates we need $O(n^2 t \log t)$ comparisons. □

In our experiments all SQL approaches were much slower than the class of algorithms presented in the next section. Even worse, for large databases all SQL approaches were infeasible, because of their immense runtime. We also tested sorted indexes on all attributes

to reduce the amount of sorting. Thus, the complexity in terms of comparisons reduces to $O(nt \log t + n^2 t)$. Also with indexes, join queries are the fastest SQL approach and – as expected – the test itself speeds up. But (i) the time for index creation almost uses up the time saved during IND testing, and (ii) for large databases this approach is still infeasible (note that all attributes must be indexed). Experimental results are presented in Chapter 4.

2.2 Single Pass Inclusion DEpendency Recognition (SPIDER)

To test all unary IND candidates we must test all pairs of attributes. First, we describe the basic idea for testing single IND candidates. Then we introduce the SPIDER algorithm, which extends the basic approach two-attributes-at-a-time by synchronizing all tests of IND candidates to save computation, i. e., reducing the number of value comparisons and I/O operations.

2.2.1 Basic test for single IND candidates

The basic test can be performed using the following procedure, which is a variation of a sort merge join: First, sort the value sets of all attributes using any common sort order. From this point on it is sufficient to regard only distinct values. Second, iterate over the sorted value sets of each candidate $A \subseteq B$, starting from the smallest items using cursors. Let $\mathsf{dep} \in A$ be the current dependent value and $\mathsf{ref} \in B$ be the current referenced value of an IND candidate. There are three possible cases: (i) If $\mathsf{dep} = \mathsf{ref}$ move both cursors one position further, because the dependent value was found in the set of referenced values. (ii) If $\mathsf{dep} > \mathsf{ref}$ move only the referenced cursor, i. e., look for the current dependent value in the remaining referenced values. (iii) Otherwise, if $\mathsf{dep} < \mathsf{ref}$, dep is not included in the referenced value set and we can immediately stop the test for this candidate. A candidate satisfies an IND if all dependent values were found in the referenced value set.

The brute force algorithm of [BLN06] directly implements this procedure and creates and tests all IND candidates sequentially, i. e., one by one. The sorted attribute value lists are stored as files on disk. Compared to a SQL join, the main advantages are (i) the single sort of all attribute's values (instead of once per IND) and (ii) the early stop for unsatisfied INDs, as usually most IND candidates are unsatisfied. Accordingly, most tests stop after comparing only a few or even only a single value pair, while a SQL join always computes all unmatched values (or all matching values).

Theorem 2. *The complexity in terms of comparisons of the brute force algorithm is* $O(nt \log t + n^2 t)$.

Proof. We need $O(nt \log t)$ comparisons to sort all n attributes inside the database. Furthermore, we need $O(n^2 t)$ comparisons to test all $n(n-1)$ IND candidates. □

This complexity equals the SQL approaches complexity using indexes. But this worst case is rarely necessary on average – as opposed to the SQL approach –, assuming that for most IND candidates only very few values must be compared before an invalid value in the dependent attribute is found. But there are yet two disadvantages: First, each attribute's values are read as often as the attribute is part of an IND candidate, i. e., $2(n-1)$ times.

Theorem 3. *The number of values read from disk to test IND candidates is* $O(n^2 \cdot 2t)$ *for the brute force algorithm.*

Proof. For each of the $n(n-1)$ candidates we must read at most once all of the two attribute's values. □

Second, the values of all attributes are compared pairwise for the IND tests. When testing all IND candidates in parallel we are able to combine all these tests. Thus, we reduce the quadratic number of comparisons to a logarithmic number of comparisons, as we show next.

2.2.2 Parallel test for all IND candidates with SPIDER

We now show how the simple brute force approach can be improved considerably. We retain the idea of using sorted data sets allowing an early stop of execution. However, the new algorithm, SPIDER, eliminates the need to read data multiple times from disk. Instead, it creates and tests all IND candidates in parallel with a single read over the data. Further, it eliminates the need to compare attribute values pairwise by sorting all attribute's values as needed using a special data structure. This procedure greatly reduces the complexity in terms of I/O and of comparisons and reduces the run time (by a factor up to 10).

SPIDER first opens all sorted attribute's lists (stored as files on disk) and starts reading values using one cursor per attribute. The challenge is to decide when the cursor for each attribute can be moved: All potentially dependent attributes affect the point in time when the cursor of a referenced attribute can be moved. But also all referenced attributes control when the cursor of a dependent attribute can be moved. Finally, any particular attribute can be a referenced and dependent attribute simultaneously.

Consider the following attributes and their values: $A = \{1, 2, 3, 4\}$, $B = \{2, 4\}$, $C = \{1, 2, 4, 5\}$ (see Figure 2.1). In all, $n(n-1) = 6$ IND candidates have to be tested: $A \subseteq B, A \subseteq C, B \subseteq A, B \subseteq C, C \subseteq A$, and $C \subseteq B$. Only two out of these six IND candidates are in fact satisfied: $B \subseteq A$ and $B \subseteq C$. Let all cursors initially point to the first item of each attribute.

To test $B \subseteq C$ the cursor in C has to be moved, because $2 > 1$. But before this movement the current value of C is needed to test $A \subseteq C$. Vice versa, the cursor in C is needed to test $C \subseteq A$. Despite this mutual dependency, it is possible to synchronize the cursor movements without running into deadlocks or missing IND candidate tests, because we use *sorted* data sets. The main idea is to process attributes blockwise sorted by their values while maintaining all IND candidates covered by these attributes. E. g.,

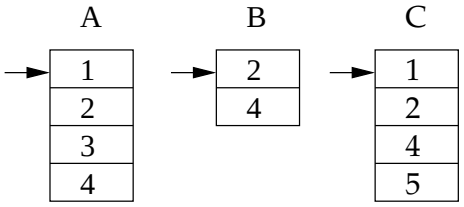


Figure 2.1: Three attributes with sorted data sets.

in our example we first process all attributes (and their IND candidates) that include the value 1, then all attributes that include the value 2, and so on. Before formally describing the algorithm we give an intuitive idea of SPIDER using the example above.

SPIDER is based on a data structure that sorts all attribute's values and represents equal values grouped together. In Figure 2.2 the attributes are given as columns and same values are grouped in rows.

attributes A,B,C

A	B	C
1		1
2	2	2
3		
4	4	4
		5

att and refs in SPIDER iteration steps

	att	A.refs	B.refs	C.refs
Initialization:		B,C	A,C	A,B
Step 1:	A,C	C	A,C	A
Step 2:	A,B,C	C	A,C	A
Step 3:	A	∅	A,C	A
Step 4:	A,B,C	∅	A,C	A
Step 5:	C	∅	A,C	∅

Figure 2.2: Attributes of Figure 2.1 represented in SPIDER's data structure: Attribute values are shown as columns; equal values are grouped in rows. IND candidates are given as lists 'refs' for each attribute.

To represent the IND candidates we use the following observation: IND candidates can be grouped into disjoint sets by their dependent attribute, i. e., all IND candidates covering a given dependent attribute define a set. Thus, each attribute A maintains in SPIDER's data structure a list $A.\mathsf{refs}$ of attributes that build a (still satisfied) IND candidate $A \subseteq B, B \in A.\mathsf{refs}$. In our running example these lists are initially set to $A.\mathsf{refs} = \{B, C\}$, $B.\mathsf{refs} = \{A, C\}$, and $C.\mathsf{refs} = \{A, B\}$.

SPIDER iterates row by row over the data structure shown in Figure 2.2 performing the following procedure:

1. Get the set att of all attributes that contain this row's value.

2. For all attributes in list att: Update their lists refs by intersecting refs and att.

In our running example this procedure yields the following results as visualized in Figure 2.2:

1. In the first iteration step for the data value '1' we have $\mathsf{att} = \{A, C\}$. Thus, the lists refs of A and C are updated as follows:

$$\begin{aligned} A.\mathsf{refs} &:= A.\mathsf{refs} \cap \mathsf{att} = \{B, C\} \cap \{A, C\} = \{C\} \\ C.\mathsf{refs} &:= C.\mathsf{refs} \cap \mathsf{att} = \{A, B\} \cap \{A, C\} = \{A\} \end{aligned}$$

   Thus we now know the IND candidates $A \subseteq B$ and $C \subseteq B$ are unsatisfied, which is correct, because the value 1 of A and C is not contained in B.

2. In the second iteration step $\mathsf{att} = \{A, B, C\}$. This means the intersections with their lists refs do not change these lists, which is correct, because value 2 is contained in all attributes.

3. The next iteration step defines $\mathsf{att} = \{A\}$, because the value '3' appears only in A. Thus, we update $A.\mathsf{refs} := A.\mathsf{refs} \cap \{A\} = \emptyset$. This – correctly – means A is not contained in any other attribute. Note that the values of A are still needed in the data structure, because A is still a potentially referenced attribute of attributes B and C.

4. The fourth step is analogous to step two, because all attributes are contained in att.

5. The last iteration step sets $C.\mathsf{refs} := C.\mathsf{refs} \cap \{C\} = \emptyset$, because $\mathsf{att} = \{C\}$.

After processing all values the data structure provides only the satisfied INDs given by all attribute's lists refs. In our running example $B \subseteq A$ and $B \subseteq C$ are satisfied INDs. There are no INDs with A or C as referenced attribute, because their lists refs are empty.

2.2.3 The SPIDER algorithm

The data structure of SPIDER represents all attributes and their values as well as the IND candidates. Further, the attributes are sorted by all values in all attributes. This structure can be achieved by the following two representations.

Each attribute is represented as an *attribute object* providing the attribute's sorted values and a cursor to the current value. As IND candidates can be divided into distinct sets by their dependent attribute, the IND candidates are represented as the list $\mathsf{A.refs}$ in each attribute object A, i. e., if $A \subseteq B$ then $B \in \mathsf{A.refs}$. Initially these sets are filled with all IND candidates that are to be tested. During a SPIDER run the attributes in $\mathsf{A.refs}$ are known to contain all previously viewed values of A and the currently viewed value of A, i.e., the IND yet holds.

Note that an attribute can be covered in multiple IND candidates as referenced attribute and / or as dependent attribute. The dependent role is represented as attribute object, the referenced role is represented in another attribute object's list refs.

The main idea of SPIDER is to process all attributes with equal values as a block. Thus, the data structure provides all attributes sorted by all values in all attributes. This sorting can be efficiently achieved, because the values in each individual attribute are

already sorted. Thus, we hold all attribute objects in a min-heap sorted by their current values.

The SPIDER algorithm is given in Algorithm 2.3. SPIDER iterates blockwise over the heap data structure by (1) receiving the set att of attribute objects with the currently minimal but equal value, and (2) updating the lists refs of attribute objects in att by intersecting refs and att. This way, all attributes B not containing the current value, i. e., $B \notin$ att, are discarded from the set of referenced attributes for attributes $A \in$ att. (3) If an attribute object A in att has a next value then the cursor is moved on and the attribute object is re-inserted into the min-heap. Otherwise, all INDs given by A and its list A.refs are proven satisfied.

Input: attributes: set of attribute objects with their sorted values and their respective refs sets (the IND candidates)
Output: Set of satisfied INDs.

```
Min-Heap heap := new Min-Heap( attributes )
while heap != ∅ do
    // get attributes with equal min. value
    att := heap.removeMinAttributes()
    foreach A ∈ att do
        // update list A.refs
        A.refs := A.refs ∩ att
        // process next value
        if A has next value then
            A.moveCursor()
            heap.add(A)
        else
            foreach B ∈ A.refs do
                INDs := INDs ∪ { A ⊆ B }
    return INDs
```

Figure 2.3: Algorithm SPIDER.

Theorem 4. *The number of comparisons of* SPIDER *is* $O(nt \log t)$ *with* n *attributes and maximally* t *values in each attribute, assuming* $t > n$.

Proof. To sort all data we need $O(nt \log t)$ comparisons. We need $O(\log n)$ comparisons to insert one attribute object into the heap depending on its currently viewed value, and thus $O(nt \log n)$ to insert all attributes. To pop attributes from the heap we need $O(nt \log n)$ comparisons for the heap operations and $O(nt)$ comparisons for identifying the attributes in the minimum value set (att). The list intersection of A.refs and att is $O(1)$ assuming both lists are represented as bit vectors of fixed size[1], i. e., we need $O(nt)$ operations for all needed intersections. Thus, the complexity of SPIDER to test the IND candidates (i. e.,

[1]Note that the complexity of operations at bit vectors is O(1) if the maximum number of items is constant, and O(MaxItems) if the bit vector must be implemented with variable size. As we don't need

without sorting) is $O(nt \log n)$. Assuming $t > n$, we need $O(nt \log t)$ comparisons for the complete execution of SPIDER, i. e., for sorting and testing. □

This analysis shows that the complexity to test the IND candidates is lower than the complexity to sort all attribute's values (if $t > n$). This is a considerable improvement over previously described approaches [BB95, MLP09], which require more effort for testing than for sorting (see Chapters 4 and 8).

Theorem 5. *The number of values read from disk for* SPIDER *is* $O(n \cdot t)$.

Proof. Each value is read at most once, because each attribute's values are represented in a single attribute object and each item in the list of its values is read at most once. □

The experiments in Chapter 4 shall validate the statement that the complexity of SPIDER depends only on the number of attributes and the number of their values, but not on the number of tested IND candidates.

SPIDER uses a database to sort and "distinct" the values of all attributes, and then writes the sorted lists to disk. Thus, the total time of a run consists of sorting inside the database, shipping the sets to a client, writing them to disk, and reading them in parallel for the tests. In Chapter 4 we shall analyze in detail which of these components dominate the runtime of the algorithm. Before that we evaluate in Chapter 3 several strategies to prune IND candidates.

to vary the size during a SPIDER run, the complexity in our case is O(1). (see e.g. Nell Dale and Henry M. Walker: Abstract Data Types. Specifications, Implementations and Applications. Jones and Bartlett Publishers, Inc., 1996)

Chapter 3

Pruning IND candidates

In this chapter we discuss various strategies to prune IND candidates. The strategies are safe in that they do not prune candidates unless it is guaranteed that they cannot be INDs. For each strategy we evaluate the impact on various tests sets. The pruning strategies are also applicable to other algorithms for IND detection (see Chapters 4 and 8).

We also and particularly examine the exclusion of entire attributes from consideration. This is possible when all IND candidates that are covered by an attribute are excluded. Excluding attributes is particularly interesting for SPIDER since its complexity directly depends on the number of involved attributes.

We note already here that we found that the efficiency improvement of pruning to be less favorable as one might expect on first glance. This effect was not mentioned by related work and also came as a surprise to us and we thus examine it intensively.

3.1 Simple strategies

A simple pruning strategy (called distinct in Table 3.1) is to compare the number of distinct values of each IND candidate: Let $v(A)$ denote the set of distinct values of attribute A. If $|v(\mathrm{A})| > |v(\mathrm{B})|$ then there is at least one value in the attribute A that is not included in attribute B. Thus, the IND candidate $A \subseteq B$ can be excluded (but not $B \subseteq A$).

An equally simple test – suggested by [BB95] – is to compare the maximum and minimum values of attributes. An IND candidate can be excluded (i) if the maximum dependent value is greater than the maximum referenced value (we call this strategy max) or (ii) if the minimum dependent value is lower than the minimum referenced value (called min).

All three tests are inexpensive, because we can piggy-back the computation of distinct, max, and min to the procedure of sorting the attributes in the database and writing them to disk.

The selectivity of these filters on some test databases is shown in Tab. 3.1. In all cases, the number of IND candidates is reduced by at least a factor of 6. While these savings seem impressive we analyze in Sec. 4.3 how the actual runtime is affected.

	UniProt	TPC-H	PDB	
	900 MB	1.3 GB	2.8 GB	32 GB
# attributes	68	61	1,215	1,297
# IND cand.	1,393	877	219,106	245,562
# INDs	36	33	4,972	5,431
# attr. in INDs	31	20	448	478
distinct				
# IND cand.	910	477	139,807	157,818
# attr. in IND cand.	68	58	1,208	1,297
distinct & max				
# IND cand.	541	295	72,016	83,321
# attr. in IND cand.	59	54	997	1,080
distinct & min				
# IND cand.	345	275	61,920	68,664
# attr. in IND cand.	54	57	990	1,042
dist. & max & min				
# IND cand.	174	137	22,655	25,821
# attr. in IND cand.	49	52	670	709

Table 3.1: Number of remaining IND candidates and attributes after pruning using distinct, max, and min.

3.2 Bloom filter

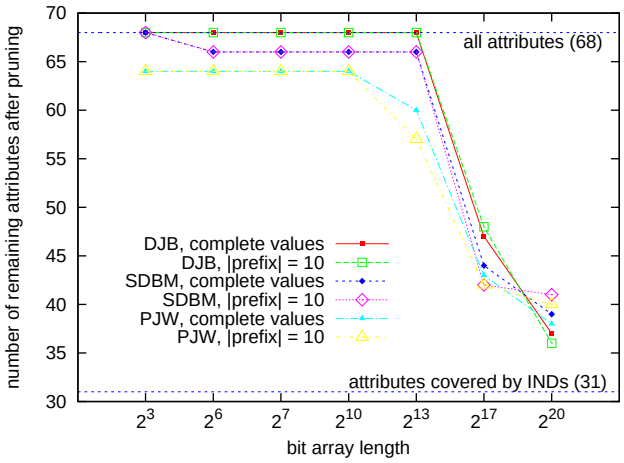
The simple filters described above use only very little information about the attributes. In particular, these filters are insensitive to the distribution of values between the minimum and maximum values. Bloom filters offer a better adaptation of the filter to the data [Blo70]: We hash each attribute's values into a bit-array such that each bit represents several values. To prune IND candidates we compare the bit arrays of the two attributes looking for bits that are 1 in the dependent array, but 0 in the referenced array. If one such bit is found, the candidate is not satisfied; otherwise, we still need to test all values. The test can be performed efficiently by a bitwise $\mathsf{dep} \wedge \neg\mathsf{ref}$ operation, such that in the resulting array a bit is 1 iff the IND candidate can be excluded.

To achieve an optimal impact on filtering entire attributes and not just candidate pairs, we experimented with the size of the bit array and the hash function. Further, we examined whether hashing only prefixes of certain length instead of hashing complete attribute values affects filtering. We tested the PJW, DJB, and SDBM hash functions[1]. The results of our experiments are shown in Figure 3.1.

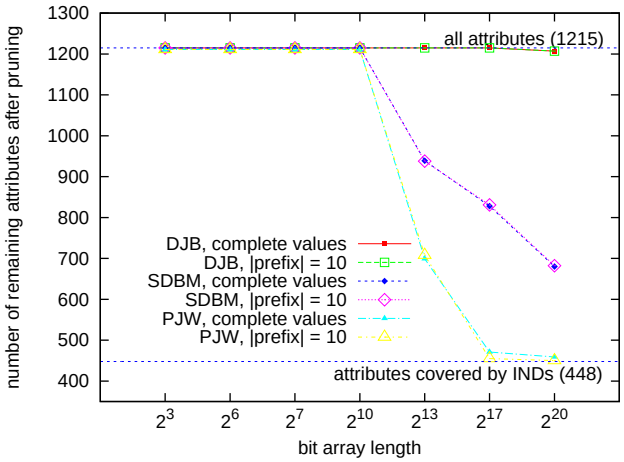
The DJB shows worst filtering impact on all tested databases. SDBM and PJW are comparable on UniProt, but PJW prunes clearly better on PDB data.

The results confirm the intuition that increasing the size of the bit array leads to a higher amount of pruned IND candidates and attributes due to an improved spread of

[1] See General Purpose Hash Function Algorithms, `www.partow.net`



(a) UniProt



(b) PDB 2.8 GB

Figure 3.1: Impact of varying Bloom filter parameters on number of remaining attributes for (a) UniProt and (b) PDB.

values. On the other hand, large arrays require more memory and more time for their comparison. The experiments show that a length of 2^{17} bit (which requires 20 MB memory for 1000 attributes) already is very effective for the PDB with respect to pruned attributes and also has very good effects on UniProt (see Fig. 3.1). Longer arrays improve pruning on UniProt only marginally. Although these results highly depend on the data sets, they show that reasonable reductions can be achieved with modest memory consumption.

To reduce filter creation costs we tested the idea not to hash entire values into the bit array but just prefixes of a certain length. The hash value of a small prefix can be computed much faster and we already expect high selectivity in the first few characters. We tested on prefix lengths between 1 and 10. Interestingly, hashing prefixes of fixed length already leads to impressive results on very small bit arrays. Larger hashed prefixes result in larger numbers of pruned IND candidates and attributes – as one would expect.

However, we found that hashing prefixes of length 10 already behaves nearly identical to hashing the complete values with regard to filtering attributes; in UniProt data it is even slightly better than hashing complete values due to the different distribution of the shorter values to hash buckets (see Fig. 3.1).

In the context of information integration of unknown data sources one cannot determine "the" optimal setting of parameters without a detailed (and costly) analysis. Nevertheless, hashing prefixes of length 10 into a 2^{17} bit array with the PJW hash function seems to be the combination that covers all tested databases best. The filter results for TPC-H and the entire PDB confirm this statement (see Table 3.2).

The effects of combining Bloom filters (parameters as above) with the simple pruning strategies are shown in Table 3.2. The Bloom filter is in almost all cases more selective than the filter on number of distinct items, maximum, and minimum (compare to Tab. 3.1, last line). But the simple filters catch some IND candidates that are not pruned by the Bloom filter. Therefore, all filters together reach the best performance.

	UniProt	TPC-H	PDB	
	900 MB	1.3 GB	2.8 GB	32 GB
# attributes	68	61	1,215	1,297
# IND cand.	1,393	877	219,106	245,562
# INDs	36	33	4,972	5,431
# attr. in INDs	31	20	448	478
distinct & bloom				
# IND cand.	245	43	10,462	12,130
# attr. in IND cand.	42	26	454	587
dist., max, min, bloom				
# IND cand.	125	40	9,006	10,149
# attr. in IND cand.	35	25	450	518

Table 3.2: Number of remaining IND candidates and attributes using Bloom filter and simple pruning.

3.3 Filtering and Performance

As previously mentioned, the overall runtime of SPIDER is composed of the costs of sorting data, reading it from the database, writing it to disk, and reading it again for the IND tests. Therefore, filters are most useful if they can be applied within the database, thus reducing the amount of data to be shipped to and considered by the client. However, applying the filters inside the database also costs time. For instance, to obtain the minimum and maximum value and the number of different values for an attribute the database must read the entire bag of values of this attribute – a read that is repeated later for all attributes that cannot be excluded completely. Note that in general it is impossible to force a SQL database to save the previously "distinct" attribute's values

for reuse. For Bloom filters, expensive computations, for which database programming languages, such as Transact-SQL or PL/SQL, are not well prepared (array manipulation, XOR operations), must take place that might outweigh the simple read-and-compare style of the SPIDER algorithm. On the other hand, applying the filters while we read the sorted columns from the database comes at almost no additional cost, as by then data shipping has already taken place. Thus, it is not at all clear where filters should be applied. We analyze this trade-off in Sections 4.3 and 4.4.

Chapter 4

Experimental Results for Unary INDs

We tested all algorithms on the databases presented in Section 1.2 on a Linux system with 2 Intel Xeon processors (2.80 GHz) and 12 GB RAM. We used a commercial object-relational database management system.

Given the ultimate goal of detecting foreign key relationships (through INDs), we applied two obvious simplifications. First, we create only IND candidates that have a referenced attribute with only unique non-null values (a potential primary key). Second, we exclude candidates with both attributes from the same table (intra-table references). Note that in general SPIDER does not depend on these restrictions. All tests were performed including the distinct filter. The runtime effects of the other filters were tested individually in Section 4.3.

4.1 Runtime results

Experimental runtime results are shown in Table 4.1. We directly compare SPIDER to the fastest SQL approach (join), to the brute force algorithm, and to two re-implemented approaches from related work – Bell and Brockhausen [BB95] and Marchi et al. [MLP09] (see Section 4.2). Both brute force and SPIDER greatly outperform the join approach, which is infeasible for large schemata.

For low numbers of IND candidates, i. e., UniProt and TPC-H, there is only a small difference between the brute force and the SPIDER algorithm. For large schemata with high numbers of IND candidates the improvement of SPIDER over brute force is considerable.

In our example databases the number of satisfied INDs depends on the number of attributes and therefore in the number of IND candidates. It does not depend on the database size as can be seen when comparing the number of satisfied INDs in TPC-H and the 2.8 GB part of PDB. We expect this behavior for most databases.

Brute force and SPIDER share the same cost (and time) to sort and "distinct" the data inside the database, to ship them to the client and to write them to the file system. The real difference is the IND candidate test. The shared overhead for UniProt and TPC-H is 1 m 32 s and 6 m 15 s, respectively. Thus, the IND candidate test speeds up for these small schemata by factor of 2 for UniProt and by factor 1.5 for TPC-H. For PDB the overhead

	UniProt	TPC-H	PDB	
DB size	900 MB	1.3 GB	2.8 GB	32 GB
# attributes	68	61	1,215	1,297
# IND cand.	910	477	139,807	157,818
# INDs	36	33	4,972	5,431
`join`	9 m 04 s	25 m 02 s	16 h 14 m	–
Brute Force	2 m 11 s	6 m 30 s	3 h 29 m	19 h 51 m
Bell&Brockhausen	4 m 39 s	–	1 h 32 m	–
Marchi et al.	9 h 58 m	–	–	–
SPIDER	1 m 51 s	6 m 25 s	23 m 36 s	6 h 07 m

Table 4.1: Run-time performance of the algorithms. IND candidates are restricted to cover unique referenced attributes.

is 21 m for the 2.8 GB part and 5 h 56 m for the complete PDB. Thus, in both cases the speed-up for testing IND candidates is 75-fold. These results confirm that the difference between brute force and SPIDER mostly depends on the number of attributes and not so much on the size of the database, as suggested by our complexity analysis in Chapter 2.

We also ran SPIDER on the SAP data set with initially almost $\frac{1}{4}$ million attributes. However, we had to reduce the number of used attributes due to (i) the allowed number of open files on our system and (ii) main memory constraints of the Java virtual machine, which made it impossible to open more than $\sim 25,000$ attributes at once. These problems could be circumvented by using partitioning techniques on the candidate sets, but we have not explored this direction yet. Instead, we applied a filter to the original 237,972 attributes in the SAP database: We considered only attributes with at least 50 distinct values. This restriction resulted in about 230,000 IND candidates covering 22,887 attributes. The size of this excerpt is about 40 GB. We identified 3,927 INDs within 2 h 38 m. These measurements show that SPIDER is well capable of handling very large databases with a large number of IND candidates. Further, together with our results on the complete PDB (32 GB, tested within ~ 6 hours) it shows that the runtime depends not directly on the size of the database, but mostly on the number and size of distinct values in the database.

4.2 Comparison to other approaches

We compared our algorithms with the approaches of Bell and Brockhausen [BB95] and Marchi et al. [MLP09] using our own, careful re-implementation (see Table 4.1). We made one adaptation to both algorithms: As a simple filtering, both methods test only candidates of the same data type. In our life sciences setting, we usually find schemata with only `string` attributes or no type definition at all, and thus must test all pairs of attributes. For a fair comparison we assigned the same data types to all attributes; obviously, it would be very simple to extend SPIDER to also filter for data types.

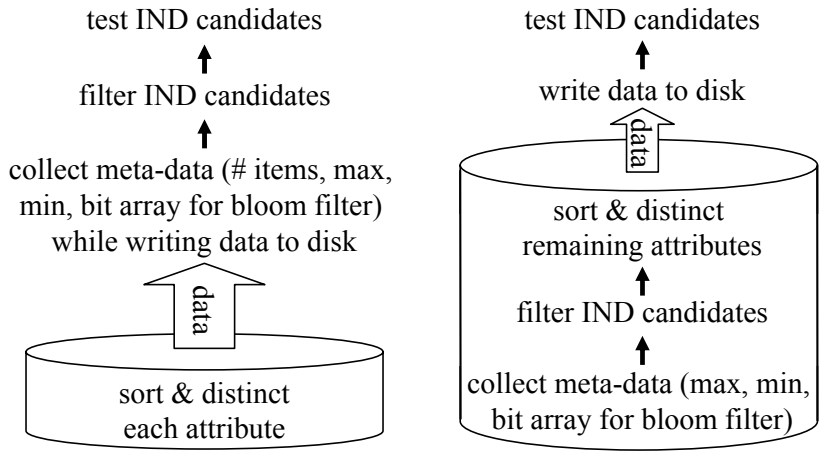
The algorithm of Bell and Brockhausen leverages filters on maxima and minima to

prune IND candidates and utilizes already finished IND tests for further pruning, thus exploiting the transitivity of the IND relationship. IND candidates are tested by SQL join statements. It runs 4 m 39 s on UniProt data and 1 h 32 m on the smaller part of the PDB, which is three times slower than SPIDER. Furthermore, the runtime of this approach strongly depends on the number of satisfied INDs – as opposed to SPIDER. To show this dependency, we carried out an experiment in which we removed the restrictions to IND candidates covering unique referenced attributes, thus increasing the number of candidates. The analysis of the UniProt data set (2,235 IND candidates) did not stop within 5 hours and the analysis of the 2.8 GB part of PDB (734,851 IND candidates) did not finish within 19 hours. SPIDER tests these sets within $\sim$ 2 m and $\sim$ 25 m, respectively (see also Tab. 5.1).

The approach of Marchi et al. preprocesses all data by assigning to each value in the database all attribute's names that contain this value. The results are stored in a table. Afterwards, all IND candidates are tested in parallel exploiting the sets of attribute names. We implemented the preprocessing as a PL/SQL script and the test as a Java program. The preprocessing on UniProt already consumes 9 h 58 m, the actual IND test only $\sim$ 4 m, compared to the total run time of SPIDER of 1 m 51 s. Note that Marchi et al. tested their approach only on small databases; their largest database comes in a 77 MB dump file, compared to UniProt as our smallest database with 900 MB.

4.3 Effects of pruning

Table 4.2 shows run times when we apply filters before running SPIDER. For the experiment, we read all data out of the database and wrote it to disk. After this step we filtered the IND candidates and applied the SPIDER test to the remaining IND candidates (see Fig. 4.1 a). In the next section we examine the alternative of filtering within the database (Fig. 4.1 b).



(a) Database-external pruning. (b) Database-internal pruning.

Figure 4.1: Steps required for database-internal and database-external pruning.

In Table 4.2, we distinguish two components: (i) the time to sort and read (inside the RDBMS), ship (to the client), and write the data to files ("*SRSW*" in the table), and (ii) the time to read and test the data at the client ("*test*").

	UniProt	TPC-H	PDB	
DB size	900 MB	1.3 GB	2.8 GB	32 GB
distinct	1 m 51 s	6 m 25 s	23 m 36 s	6 h 07 m
SRSW	1 m 32 s	6 m 15 s	20 m 59 s	5 h 56 m
test	17 s	8 s	2 m 23 s	11 m 21 s
distinct & max & min & bloom	1 m 55 s	6 m 53 s	24 m 06 s	6 h 18 m
SRSW	1 m 34 s	6 m 33 s	21 m 42 s	6 h 08 m
test	20 s	7 s	2 m 15 s	9 m 24 s

Table 4.2: SPIDER with and without filtering of IND candidates. IND candidates are restricted to cover unique referenced attributes.

As expected, the runtime for reading the data and writing them to disk increases slightly when applying the filters, because the bit array for the Bloom filter has to be computed. However, it is on first sight very surprising that, despite the great reduction in IND candidates to be tested, the runtime for testing decreases only minimally. This observation reconfirms our claim from Chapter 2 that the runtime of SPIDER is independent in the number of IND candidates, i. e., representing and tracking the IND candidates incurs almost no cost.

It is more surprising that even the exclusion of entire attributes does not yield much speed-up – even though the complexity of SPIDER depends on the number of attributes. This observation can be explained as follows: Most of the time for testing is required by satisfied INDs, because in those cases the entire value sets need to be read and compared until the end – there is no early stopping. However, the filtering removes only candidates that are certainly not satisfied and which would require only a few comparisons by SPIDER anyway. Thus, filtering on average removes only candidates that would not incur much effort.

Together, excluding unsatisfied INDs by filtering outside the database does not save much time but costs time for computing the filters. The next section evaluates possible approaches of filtering IND candidates (and with it attributes) inside the RDBMS. This way, data of attributes not covered by IND candidates do not have to be shipped out of the database incurring high I/O cost.

4.4 Database internal pruning

Given the observations described in the previous section, it is clear that further optimization of SPIDER should concentrate on the SRSW phase (sorting, reading, shipping, and writing into filesystem). Of the 24 minutes to test the 2.8 GB part of PDB, approx. 21 minutes are spent in the SRSW phase and less than 3 minutes in the test phase

(see Tab. 4.2). Thus, we strived to reduce the number and complexity of queries inside the database and/or reduce the amount of data to be shipped to the client by applying filtering already inside the RDBMS.

Reaching this goal is not as straight-forward as one might think: The 21 minutes of the SRSW phase split down to $\frac{2}{3}$ for reading and sorting the data inside the database, and $\frac{1}{3}$ for reading and shipping them out of the database and writing them to disk. Thus, any filtering that requires to scan or sort all values, such as min, max, or distinct, very likely does not improve the overall performance, because the time saved during shipping and testing is dominated by the time required to scan/sort the data twice.

Another procedure for efficient filtering inside the database is to read all tables once and collect the needed metadata (maximum, minimum, and the Bloom filter bit array) for all their attributes on the fly (instead of reading them attribute-by-attribute). The reading operations can be done efficiently by a full table scan. After applying the filter only the remaining attributes must be sorted and further processed. Note that the number of distinct values cannot be collected by this approach (see Fig. 4.1 b).

There are two problems with this idea: (i) *All* values have to be processed to collect the needed metadata instead of only distinct values. The ratio of distinct values to all values of the complete database varies for our test databases from 4 % to 24 %, i. e., at least 4 times as many items have to be processed for collecting metadata. (ii) The interface between the DBMS and the metadata collector must be fast enough to process this amount of data. We tested two setups: a Java interface and a dynamic PL/SQL interface. Both implementations showed that already reading the data over the interface (without any further processing) takes more time than sorting the attributes, shipping the sorted "distinct" values out of the database, and writing them to disk. If this overhead were eliminated by a faster interface (using for instance a natively implemented method inside the DB kernel), the overall performance of SPIDER should speed up greatly because of the large number of pruned attributes (as shown in Chapter 3).

Note that we consciously decided not to use information on maxima, minima, and the number of distinct values supplied by the database statistics. Reading those values from the database catalogue opens the door to incorrect results due to outdated statistics. We could demand up-to-date statistics, but this requirement would just mask the necessary costs.

Chapter 5

Detecting Composite INDs

In the following chapters, we show how SPIDER can be extended in two ways: to find composite INDs (this chapter) and to find partial INDs (Chapter 6). Composite INDs are INDs that include groups of attributes, e. g., $AB \subseteq CD$. We call the number of dependent attributes the *level* of a composite IND (which of course is identical to the number of referenced attributes). Composite INDs can be identified by creating and testing IND candidates level-wise. Unsatisfied INDs at lower levels can be used to prune the space of potential candidates at higher levels, because satisfied INDs of lower levels are a precondition for a composite IND: If $A \not\subseteq C$, then there exist no B, D such that $AB \subseteq CD$.

We can use SPIDER with only minor modifications to test IND candidates of each level. SPIDER's test procedure is especially suitable for the detection of composite INDs, because its runtime is independent of the number of IND candidates. Therefore, the possibly large number of created IND candidates are analyzed efficiently.

5.1 Exhaustive run of SPIDER

To use SPIDER for composite INDs, we remove the restriction that only candidates covering a unique attribute are tested, because otherwise we could miss INDs that might be part of a larger IND. The results of exhaustively testing all IND candidates using SPIDER on our various test data sets are given in Table 5.1. In comparison to the restricted test (Table 4.1; repeated for readability in Tab. 5.1), the runtime increases slightly and more satisfied INDs are obtained. This increase implies that more values have to be handled, because attributes cannot be excluded from the SPIDER heap as early. For PDB there is the additional effect that the larger number of discovered INDs has to be saved to disk.

5.2 Composite SPIDER

The detection of INDs of level $l \geq 2$ is divided into two phases: (i) enumeration of all IND candidates that could be satisfied regarding the satisfied INDs of level $l - 1$ and (ii) test

	UniProt	TPC-H	PDB	
DB size	900 MB	1.3 GB	2.8 GB	32 GB
restricted IND candidates				
# IND cand.	910	477	139,807	157,818
# INDs	36	33	4,972	5,431
SPIDER	1 m 51 s	6 m 25 s	23 m 36 s	6 h 07 m
unrestricted IND candidates				
# IND cand.	2,235	1,616	734,851	837,358
# INDs	116	86	35,752	40,415
SPIDER	2 m 07 s	6 m 35 s	24 m 34 s	6 h 10 m

Table 5.1: Experimental results of testing IND candidates restricted to cover unique referenced attributes (repeated from Tab. 4.1) and unrestricted IND candidates using SPIDER.

of those IND candidates.

We use the GenNext algorithm presented in [MLP02] to create the IND candidates level-wise. This algorithm is an adapted AprioriGen algorithm using an order on attributes: The IND candidates of level l are generated by sorting all satisfied INDs of level $l-1$ by the first $l-2$ attribute pairs. In [MLP02] the authors show that this algorithm indeed enumerates all possible composite INDs.

We modified SPIDER in the following aspects to test composite IND candidates, preserving the advantage of testing all IND candidates (of a given level) in parallel: We hold attribute tuples and their values in the min-heap, instead of single attribute values. For each level, we query the sorted composite data sets from the database and write them to disk. This step is necessary, because we need the correct associations of the attribute values to tuples, which cannot be recovered from the single attribute value files.

The results of our experiments are shown in Table 5.2. No composite IND with more than two attributes was found for UniProt and TPC-H, i. e., no IND candidate of level 3 was generated. The results show that the total runtime over all levels is dominated by the requirement to read and write tuples for each level. This observation shows again that a database internal filter on IND candidates could heavily speed up the run times.

When analyzing the 2.8 GB part of the PDB we had to resort to a heuristic to constrain the set of satisfied unary INDs. When all unary INDs are used to create composite INDs of level 2, then 368,997 IND candidates are created covering 15,798 attribute tuples. The run took $\sim$ 33 hours and resulted in 227,028 satisfied INDs. These INDs implied about $5 * 10^6$ possible IND candidates at level 3, which cannot be tested in reasonable time.

However, recall that in the end we are hunting for foreign keys. Since we are confident that the number of actual composite foreign keys at level 2 or 3 in the data is rather small, we applied two heuristics to prune probably uninteresting, yet satisfied unary INDs: (i) The referenced attribute must have more than one distinct value; and (ii) At least 1 % of all distinct values of the referenced attribute must be covered by values of the dependent attribute. Both restrictions are very weak and should not exclude interesting foreign keys,

but they reduce the number of unary INDs to 8,830 and the number of IND candidates in level 2 to 12,535 covering 2,576 attribute tuples. Thus, we identified 7,442 binary INDs which implied 13,647 IND candidates of level 3 covering 4,336 attribute tuples (see Table 5.2).

These experiments confirm that the exponential explosion in the number of candidates is real, even when pruning with satisfied INDs from lower levels is fully exploited. There are further approaches [MP03, KR03] to find composite INDs in the literature. These approaches aim to find composite INDs at larger levels (up to level 41) and use procedures as described in Section 4.2 to find unary and binary INDs. Using SPIDER for lower levels and the approaches of [MP03, KR03] for larger levels would improve composite IND detection at all.

	UniProt	TPC-H	PDB 2.8GB[a]
composite SPIDER	5 m 25 s	27 m 29 s	6 h 40 m
level 1			
# IND cand.	2,235	1,616	734,851
# attribute tuples	68	61	1,215
# INDs	116	86	35,752
SRSW	1 m 43 s	6 m 13 s	20 m 59 s
test	37 s	17 s	2 m 41 s
level 2			
# IND cand.	20	59	12,535
# attribute tuples	14	36	2,576
# INDs	13	21	7,442
SRSW	8 m 24 s	25 m 31 s	6 h 00 m
test	44 s	54 s	16 m

[a] We filtered unary INDs before running level 2 and stopped execution after this level.

Table 5.2: Experimental results for detecting composite INDs.

Chapter 6

Detecting Partial INDs on Dirty Data

In most real-world databases one finds dirty data whenever foreign key constraints are not enforced by the system. Potential reasons are faulty parsers for importing data or simply errors in the data. We call such "foreign keys" with exceptions "dirty foreign keys", and we call such violated inclusion dependencies "partial INDs". Thus, a partial IND is an IND candidate where all but a certain (small) number of dependent values are included in the referenced attribute's values. There are two possible ways of specifying the allowable number of such values: (i) the number of all distinct, not included values expressed as a percentage of distinct values or (ii) the absolute number of not included values (as suggested by [LPT02]). Both values are helpful to rate a partial IND.

The SPIDER algorithm – with some minor modifications – is able to detect partial INDs very efficiently. Furthermore, SPIDER is able to give the exact number of not included values, which distinguishes it from other algorithms for the detection of approximate INDs [KR06, DJMS02, BH03]. The following modifications are necessary to collect for each IND candidate the number of all distinct, not included values during the test: We add a counter to the references in each attribute object in the list refs. These counters represent the number of violating values of this dependent attribute object. When updating the list refs of an attribute object we do not discard attributes from refs directly. Instead we buffer them in a list unsatRefs and increase the counter of these objects. If the given threshold is not exceeded, the attribute is re-added to the list refs. Only once a given threshold of violating values is exceeded, is the referenced attribute object discarded. The modified algorithm is shown in Figure 6.1.

To obtain the absolute number of not included values, i.e., also counting duplicates, we need the number of occurrences for each value in the dependent attributes. These can be extracted from the database using a SQL `group by` statement on the dependent attribute with `count` as aggregation function, which does not incur more work for the database compared to a single sort and `distinct`. For counting absolute numbers, every not-included dependent value needs to be multiplied by its number of occurrences.

We give experimental results only for the first alternative for specifying the tolerated level of dirtyness, i.e., allowing a certain percentage of distinct values to be not contained. The results are shown in Table 6.1. We allowed 5% violating items in the dependent value, which we believe is a very high error rate. Thus, the results should be considered

```
Input: attributes: set of attribute objects with their sorted values and their respective
  refs sets (the IND candidates); threshold
Output: Set of satisfied partial INDs.
  Min-Heap heap := new Min-Heap( attributes )
  while heap != ∅ do
      // get attributes with equal min. value
      att := heap.removeMinAttributes()
      foreach A ∈ att do
          // update list A.refs
          A.unsatRefs := A.refs \ att
          A.refs := A.refs ∩ att
          // Refs with counter ≤ threshold
          // remain.
          foreach ref ∈ A.unsatRefs do
              ref.counter++
              if ref.counter ≤ threshold then
                  A.refs := A.refs ∪ {ref}
          // process next value
          if A has next value then
              A.moveCursor()
              heap.add(A)
          else
              foreach ref ∈ A.refs do
                  INDs := INDs ∪ { A ⊆ ref }
      return INDs
```

Figure 6.1: Modified algorithm SPIDER to test partial IND candidates.

as rather conservative runtime estimations.

At this level there are indeed a considerable amount of dirty foreign keys in three of the four data sets. However, compared to the results of exact tests (Table 4.1; repeated in Tab. 6.1), the runtime increases only minimally. The difference stems from the additional costs for counting and for comparing this number to the given threshold. Furthermore, more values have to be processed, because attributes are excluded later from all IND candidates and thus from the SPIDER heap. But the experiments show that SPIDER is very efficient also for detecting partial INDs. Note that for a lower error rate the difference to a run without allowed errors would be even smaller.

	UniProt	TPC-H	PDB	
DB size	900 MB	1.5 GB	2.8 GB	32 GB
# IND cand.	1,393	877	219,106	245,562
# INDs	36	33	4,972	5,431
# partial INDs	36	40	10,737	12,081
SPIDER	1 m 51 s	6 m 25 s	23 m 36 s	6 h 07 m
partial SPIDER	1 m 57 s	6 m 26 s	27 m 25 s	6 h 27 m

Table 6.1: Results for finding partial INDs. 5 % of not included dependent values were allowed. IND candidates are restricted to cover unique referenced attributes.

Chapter 7

Evaluating INDs

We now evaluate the quality of detected INDs for their ultimate purpose, their indication of foreign keys. We evaluate UniProt and TPC-H, because their schema definitions come along with foreign keys and thus provide a gold standard.

The BioSQL schema, into which we parsed UniProt, defines 21 foreign keys. Of those we find 19 as INDs. The remaining two constraints are defined on empty tables and thus cannot be discovered by any instance-based approach. Another 11 INDs found by SPIDER are not defined as foreign keys in BioSQL but provide an interesting insight: They result from situations where there are two foreign key attributes A, B referencing a primary key attribute C. In addition to the defined FKs $A \subseteq C$ and $B \subseteq C$, SPIDER also detects the IND $A \subseteq B$ and sometimes additionally $B \subseteq A$ (i. e., $A = B$). SPIDER found three INDs in 1:1 relationships where only one direction was defined as foreign key constraint. Note that all these "false positive" INDs actually are semantically correct and constitute helpful metadata to understand unknown schemata. They are omitted from the database definitions only for technical reasons. For instance, if a 1:1 relationship is defined by two foreign keys, then systems without the ability to defer constraint checking cannot allow the insertion of tuples.

Further, SPIDER detected three false positives that each relate a dependent attribute with only a single distinct value to a referenced attribute with about 10,000 distinct values. Altogether, these results imply a recall of 90% and a precision of $\sim 92\%$ for the detection of unary foreign key constraints in this particular example. Filtering already at the unary level is very important when it comes to composite INDs, as shown in Section 5.2. For instance, the BioSQL schema defines no composite keys. All the 13 detected composite INDs derive from the three false unary INDs.

TPC-H defines seven unary foreign keys and one binary foreign key, which were all found (recall 100%). We further detected 24 INDs that are false positives (precision $\sim 23\%$). They all derive from the fact that the database uses only surrogate keys generated using a counter starting from the same initial number '1'. Thus, we often detect INDs either between two surrogate keys or between a numeric attribute and a surrogate key. Such cases, i. e., artificial keys generated by sequences ranging over the same namespace in all tables, are an apparent problem for all instance-based metadata discovery methods.

Overall, this evaluation shows that by far not all detected INDs represent foreign

keys. The precision of simply taking every IND as foreign key would be $\sim$ 92% for UniProt and $\sim$ 23% for TPC-H. In [RAB+09] we described an advised method for filtering foreign keys from INDs using machine learning. We define 10 features based on coverage and distribution of attribute values, on attribute cardinality, and on attribute names. We evaluate four different classifiers on six data sets from three domains (including the test databases given in this paper). Using the J48 classification algorithm our approach consistently reaches F-measures above 80 % and often close to 100 %.

Chapter 8

Related Work

We already mentioned the approaches of Bell and Brockhausen [BB95] and of Marchi et al. [MLP09] in Section 4.2 and compared them to SPIDER. We summarize our statements in this Chapter and elaborate on other related work.

Bell and Brockhausen proposed SQL join statements to evaluate unary IND candidates after min/max filtering. Known foreign keys and already tested IND candidates (satisfied and unsatisfied INDs) are used for further pruning. SPIDER considerably outperforms this approach as shown in Sec. 4.2.

Marchi et al. detect unary INDs among same data types by preprocessing all data and then testing all IND candidates in parallel [MLP09]. The preprocessing assigns to each value in the database the list of attributes that include this value. This step is very costly, because all values in all attributes must be combined into one data structure. We showed in Sec. 4.2 that SPIDER outperforms this approach by orders of magnitude. Further, the authors extend their approach to find partial INDs. As the preprocessing step is reused the disadvantage of this approach remains. The authors also give a level-wise approach for detecting composite INDs. We employ their approach for composite IND candidate creation but test the candidates with our SPIDER algorithm.

Marchi et al. extended the level-wise approach for detecting composite INDs in [MP03]. Their main idea is to reduce the number of IND candidates by switching between a top-down and a bottom-up approach using an optimistic positive border. Koeller and Rundensteiner proposed to create composite IND candidates by finding cliques in k-uniform hypergraphs [KR03]. These hypergraphs are made of satisfied INDs of lower levels. Unary and binary INDs are tested by an approach similar to [BB95]. Further, the authors extend their approach in [KR06] by defining heuristics to reduce the search space. [MP03], [KR03], and [KR06] use tests for single IND candidates on diverse levels. The strength of SPIDER is – in contrast – its independence in runtime of the number of IND candidates. Note that [MP03], [KR03], and [KR06] aim at finding composite INDs of higher levels and run tests either on real world databases with INDs of maximally level three or on artificial databases with higher level INDs (up to level 41). In our test databases we did not observe such INDs and are not aware of real world databases with these characteristics. However, SPIDER could be leveraged to find composite INDs of lower levels followed by [MP03], [KR03], or [KR06] to find INDs of higher levels. Furthermore,

note that both projects provide experimental data only for rather small schemata with only few tables and attributes. A detailed run-time comparison of different algorithms for composite IND detection remains for future work.

Dasu et al. apply data summaries to detect join paths approximately, i. e., to detect INDs [DJMS02]. They use set resemblance and multiset resemblance to find a join path, its size, and its direction. The authors use this approach as a first approximate step in schema discovery to help a human expert. In our scenario we want to avoid false positive and false negative INDs. Our re-implementation of this approach did not produce such exact results.

Brown and Haas study algebraic constraints between pairs of attributes to use them in query optimization [BH03]. They create IND candidates by heuristics and data samples and might therefore miss some INDs. Our aim is explicitly to find all INDs. SQL join statements are used to test the IND candidates. Finally, Petit et al. extract IND candidates from existing applications on a database by analyzing a workload searching for frequently used equi-joins [PTBK96]. These joins are then tested against the database and rated by a human expert. Again, this is a heuristic approach that might miss some INDs.

Bravo et al. [BFM07] define conditional inclusion dependencies (CIND) as INDs restricted on data values that satisfy a given condition but do not propose a detection method. They show their potential use for data cleaning and schema matching and analyse them in terms of consistency and implication. SPIDER could be used to identify CINDs.

Chapter 9

Conclusion and Future Work

Efficient metadata discovery in foreign schemata is an important step towards data integration. We described the SPIDER algorithm for detecting inclusion dependencies in a database with insufficient schema information. The algorithm is divided into two phases: The first phase leverages the highly optimized sort operations of RDBMS. The second phase tests all IND candidates in parallel such that value comparisons are saved, all data values are read only once, and tests are stopped early. We showed the superiority of SPIDER over other approaches in terms of complexity and by experiments on a range of different data sets. SPIDER is the only known method that allows a feasible detection of INDs in databases with large numbers of attributes and data values. Even very large databases are analyzed within hours. However, as our experiments with SAP/R3 show, there is room for further research for extremely large schemata ($> 25,000$ attributes).

Furthermore, we presented and analyzed pruning strategies on IND candidates. We showed experimentally and explained in detail that database-external pruning does not speed up computation. We discussed possibilities to perform database-internal pruning and showed why this task is harder than one might expect at first glance.

We extended SPIDER to find composite and partial INDs. Testing partial INDs is almost as fast as testing exact INDs (depending on the allowed error rate). This performance makes SPIDER a suitable algorithm on dirty data sets. Testing composite INDs is more difficult and implies additional sort-runs on the composite attributes for each level, which may increase the run-time to a prohibitive degree. We showed some ideas how the effect can be alleviated by applying heuristics to discern spurious INDs from real foreign key constraints before higher-level INDs are searched. Finding more good such heuristics is also vital for the ultimate goal of our research, i.e., semi-automatic data integration in "data spaces" [HFM06].

While examining the life sciences data sets we came upon a further type of reference between different data sources: foreign keys that are prefixed or suffixed by a string identifying the referenced source. For instance, the data value `pdb|1v3a` in a protein classification source (CATH) points to the PDB protein with the accession number `1V3A`. Discovering such relationships is a daunting task, because different prefixes and suffixes of different length can be used within the same table and sometimes values even have prefixes and suffixes simultaneously.

Finally, the entire research area of schema discovery and in particular foreign key discovery lacks a benchmark or at least common data sets to evaluate and compare algorithms. While the TPC benchmarks are generally available and suitable for efficiency tests, their data does not reflect typical IND situations. Developing a schema discovery / data profiling benchmark would be a laudable task.

Acknowledgments. This research was supported by the German Ministry of Research (BMBF grant no. 0312705B). We thank Véronique Tietz and Tobias Flach for implementing bits and pieces of the Aladin system.

Bibliography

[BAW+05] A. Bairoch, R. Apweiler, C. H. Wu, W. C. Barker, B. Boeckmann, S. Ferro, E. Gasteiger, H. Huang, R. Lopez, M. Magrane, M.J. Martin, D.A. Natale, C. O'Donovan, N. Redaschi, and L.S. Yeh. The universal protein resource (UniProt). *Nucleic Acids Res*, 33(Database issue):D154–9, 2005.

[BB95] Siegfried Bell and Peter Brockhausen. Discovery of data dependencies in relational databases. In Yves Kodratoff, Gholamreza Nakhaeizadeh, and Charles Taylor, editors, *Statistics, Machine Learning and Knowledge Discovery in Databases, ML–Net Familiarization Workshop*, pages 53–58, 1995.

[BFM07] Loreto Bravo, Wenfei Fan, and Shuai Ma. Extending dependencies with conditions. In *33rd International Conference on Very Large Data Bases (VLDB)*, pages 243–254, 2007.

[BH03] Paul Brown and Peter J. Haas. BHUNT: Automatic discovery of fuzzy algebraic constraints in relational data. In *29th International Conference on Very Large Data Bases (VLDB)*, pages 668–679, 2003.

[BLN06] Jana Bauckmann, Ulf Leser, and Felix Naumann. Efficiently computing inclusion dependencies for schema discovery. In *Second International Workshop on Database Interoperability. In Workshop-Proceedings of the ICDE 06*, 2006.

[BLNT07] Jana Bauckmann, Ulf Leser, Felix Naumann, and Véronique Tietz. Efficiently detecting inclusion dependencies. In *23rd International Conference on Data Engineering (ICDE)*, 2007. Poster.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[BWF+00] H.M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T.N. Bhat, H. Weissig, I.N. Shindyalov, and P.E. Bourne. The Protein Data Bank. *Nucleic Acids Research*, 28:235–242, 2000.

[DJMS02] Tamraparni Dasu, Theodore Johnson, S. Muthukrishnan, and Vladislav Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *2002 ACM SIGMOD International Conference on Management of Data*, pages 240–251, 2002.

[HBB+02] T. Hubbard, D. Barker, E. Birney, G. Cameron, Y. Chen, L. Clark, T. Cox, J. Cuff, V. Curwen, T. Down, and et al. The Ensembl genome database project. *Nucleic Acids Res*, 30(1):38–41, 2002.

[HFM06] Alon Halevy, Michael Franklin, and David Maier. Principles of dataspace systems. In *25th ACM SIGMOD-SIGACT- SIGART symposium on Principles ofdatabase systems (PODS)*, pages 1–9. ACM Press, 2006.

[HK04] Thomas Hernandez and Subbarao Kambhampati. Integration of biological sources: current systems and challenges ahead. *SIGMOD Rec.*, 33(3):51–60, 2004.

[KMRS92] Martti Kantola, Heikki Mannila, Kari-Jouko Räihä, and Harri Siirtola. Discovering functional and inclusion dependencies in relational databases. *International Journal of Intelligent Systems*, 7:591–607, 1992.

[Koe01] Andreas Koeller. *Integration of Heterogeneous Databases: Discovery of Meta-Information and Maintenance of Schema-Restructuring Views*. PhD thesis, Worcester Polytechnic Institute, 2001.

[KR03] Andreas Koeller and Elke A. Rundensteiner. Discovery of high-dimensional inclusion dependencies. In *19th International Conference on Data Engineering (ICDE)*, pages 683–685, 2003.

[KR06] Andreas Koeller and Elke A. Rundensteiner. Heuristic strategies for the discovery of inclusion dependencies and other patterns. *Journal on Data Semantics*, 5:185–210, 2006.

[LN05] Ulf Leser and Felix Naumann. (Almost) hands-off information integration for the life sciences. In *Conference on Innovative Data Systems (CIDR 2005)*, 2005.

[LPT02] Stéphane Lopes, Jean-Marc Petit, and Farouk Toumani. Discovering interesting inclusion dependencies: application to logical database tuning. *Information Systems*, 27(1):1–19, 2002.

[MLP02] Fabien De Marchi, Stéphane Lopes, and Jean-Marc Petit. Efficient algorithms for mining inclusion dependencies. In *8th International Conference on Extending Database Technology (EDBT)*, pages 464–476. Springer-Verlag, 2002.

[MLP09] Fabien De Marchi, Stéphane Lopes, and Jean-Marc Petit. Unary and n-ary inclusion dependency discovery in relational databases. *J. Intell. Inf. Syst.*, 32(1):53–73, 2009.

[MP03] Fabien De Marchi and Jean-Marc Petit. Zigzag: a new algorithm for mining large inclusion dependencies in databases. In *Third IEEE International Conference on Data Mining*, pages 27–34, 2003.

[PTBK96] Jean-Marc Petit, Farouk Toumani, Jean-Francois Boulicaut, and Jacques Kouloumdjian. Towards the reverse engineering of denormalized relational databases. In *12th International Conference on Data Engineering (ICDE)*, pages 218–227, 1996.

[RAB+09] Alexandra Rostin, Oliver Albrecht, Jana Bauckmann, Felix Naumann, and Ulf Leser. A machine learning approach to foreign key discovery. In *12th International Workshop on the Web and Databases (WebDB)*, Providence, Rhode Island, 2009.

[TRM+05] Silke Trissl, Kristian Rother, Heiko Müller, Thomas Steinke, Ina Koch, Robert Preissner, Cornelius Frömmel, and Ulf Leser. Columba: An integrated database of proteins, structures, and annotations. *BMC Bioinformatics*, 6:81, 2005.