Efficiently Detecting Inclusion Dependencies

Jana Bauckmann Ulf Leser Felix Naumann Véronique Tietz

Department for Computer Science, Humboldt-Universität zu Berlin
Unter den Linden 6, 10099 Berlin, Germany
{bauckmann,leser,naumann,vtietz}@informatik.hu-berlin.de

Abstract

Data obtained from autonomous data sources often come with only superficial structural information, such as relation names and attribute names. For effective integration and meaningful querying of those data sets, relationships among attributes, such as foreign keys, are equally important. The efficient 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 foreign key attributes. We present our algorithm SPIDER (Single Pass Inclusion DEpendency Recognition) that efficiently finds all INDs in large databases. It uses the efficient sorting facilities of a DBMS but performs the actual comparisons outside the database thus saving many value comparisons. SPIDER analyzes a $2\,GB$ *database in* $\sim 20\,min$ *and a* $21\,GB$ *database in* $\sim 4\,h$*. We generalize SPIDER to find composite INDs covering multiple attributes, and partial INDs that are true 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.*

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 underspecified. For effective integration knowledge beyond attribute names is important. One particularly useful structural information are keys and foreign keys. Very often these constraints are not defined, for instance due to lacking knowledge about the constraints, laziness, lack of a specification language, or simply because the data are dirty and constraints do not entirely hold.

One example is the important Protein Data Bank (PDB). The PDB is distributed as structured flatfile and can be imported into a relational schema using the OpenMMS software[1]. The OpenMSS schema consists of 175 tables with 2705 attributes but without any foreign key constraint.

The goal of the research presented in this paper is to automatically and efficiently detect inclusion dependencies (INDs) 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 in 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. Whether A is in fact a foreign key or not can only be verified by a human expert.

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 of SQL queries, as suggested in previous works [3, 9]. However, as soon as a check fails, the test for that candidate pair can be aborted. A key insight is the inability of relational database systems and SQL to perform this early stop which makes them inefficient for detecting INDs (shown experimentally in [2]). To compensate, our SPIDER algorithm sorts all data in the database, exports them into the file system and then simultaneously checks for INDs in all candidate pairs using a single read of the data. Experiments show that our approach places automatic foreign key discovery even for very large databases with many attributes and many tuples into the realm of the feasible. SPIDER analyzes a 2 GB database with about 1200 attributes in ~ 20 min and a 21 GB database in ~ 4 h.

We generalize our approach to the detection of composite inclusion dependencies to reflect composite keys and foreign keys. That is, we test for all sets of attributes $A_1, \ldots, A_k$ and attributes $B_1, \ldots, B_k$ of two relations whether $(A_1, \ldots, A_k) \subseteq (B_1, \ldots, B_k)$. We also describe how SPIDER can be extended to detect partial inclusion dependencies, i. e., INDs which are satisfied for all but a certain number of values. Partial INDs are impor-

[1] openmms.sdsc.edu

tant to handle dirty data, which quickly occur when foreign keys are not checked. A partial IND cannot immediately be turned into a foreign key constraint but may provide valuable information about the data. We show experiments for all types of INDs.

1.1. Heterogeneous life science data sources

The motivation for our research is rooted in our work on data integration in the life sciences domain where many large databases are openly available. The Aladin[2] project strives to achieve automated integration of such databases in several steps [10]. In the first step an expert chooses the new database that is to be included in the system. Most life sciences databases are provided in a relational form or parsers are openly available to generate that form. But, as stated before, this relational form often lacks detailed schema information, such as specific data types, keys, or foreign keys. From this step on 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. For details and a discussion of this approach see [10].

The second step of Aladin checks uniqueness for each attribute to discover key candidates. The third step is the focus of this paper: the discovery of intra-source relationships among attributes. We check for set inclusion of each attribute in each key candidate. Knowledge about INDs also lets us determine the *primary relation* of the schema, i. e., the relation that stores the main class of the database, as is typical in life sciences data sources. This relation also defines the external IDs of the database' objects.

The next, fourth step aims at discovering relationships between different data sources. This step also profits from the results of this paper, as links between database objects are usually reflected in the form that one database stores the external IDs of objects of another database in its schema. Finally, the fifth step discovers relationships not between schema elements but between data items in the form of duplicates, i. e., information about a certain real-world object stored in several databases.

1.2. Data sets

We test the algorithms described in this paper on real life databases from our application domain: *UniProt*[3] is a database of annotated protein sequences [1] available in several formats. We used the BioSQL[4] schema and parser, creating a database of 16 tables with 85 attributes. The total size of the database is 667 MB, with the largest attribute having approximately 1 million different values.

PDB is a large database of protein structures [4]. We used the OpenMMS software for parsing PDB files into a relational database. The PDB populates 115 tables over $1,711$ attributes in the OpenMMS schema. There are no specified foreign keys. The total database size is 21 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 2.7 GB fraction of the PDB obtained by removing some extremely large tables.

Finally, we also verify our results by using a generated *TPC-H* database (scale factor 1.0).

1.3. Structure and contributions

The following section describes several approaches to the discovery of unary inclusion dependencies, and in particular the SPIDER algorithm which requires only a single scan of the data. Section 3 presents several filter techniques, which efficiently exclude pairs of attributes before running the IND detection algorithm. In Section 4 we present experimental results and compare SPIDER with other approaches from the literature. We show that our algorithm either surpasses those in runtime or makes less assumptions. Sections 5 and 6 present extensions of SPIDER to find composite INDs and partial INDs, respectively. Related work is discussed in Sec. 7, and we conclude in Sec. 8.

Note that some preliminary results from our project were presented already in [2]. We extend these results in several ways with the present paper. SPIDER improves on the SinglePass algorithm described in [2] by a factor of approx. 10 for large databases. Further, all material from Section 3 on is new.

2. Detecting Unary INDs

An inclusion dependency $A \subseteq B$ requires that the set of values of the *dependent* attribute A is included in the set of values of the *referenced* attribute B. We call a pair of attributes A and B an IND candidate prior to any tests. 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.

2.1. SQL approaches

One possible way to implement the IND tests is to utilize SQL statements. In this case each IND candidate is tested

[2] ALADIN – ALmost hAnds-off Data INtegration
[3] `www.pir.uniprot.org`
[4] `obda.open-bio.org`

independently of all other tests. Therefore, the number of comparisons is $O(n^2 t \log t)$ assuming a sort merge join on both attributes after removing duplicate values [8].

In [2] we tested the performance of several possible SQL statements for IND tests, 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 attributes cardinality. However, we showed that all SQL approaches are much slower for large databases than the algorithms we describe in the following. The reason is twofold: First, the independent test of each IND candidate prevents reusing intermediate results, in particular sorting. Thus, each attribute is sorted as often as it is part of an IND candidate. Second, one cannot formulate in SQL that query execution should stop immediately after a counter-example for the IND is found. Thus, each SQL statement we analyzed computed more than necessary. For instance, the join variants essentially computes the number of counter-examples.

2.2. Two efficient algorithms

To compute all unary INDs we must test all pairs of attributes. The test can be performed using the following procedure: First, sort the value sets of all attributes using an arbitrary but fixed sort order. From this point on it is sufficient to regard only distinct values. Second, iterate over the sorted value sets of each pair starting from the smallest item using cursors. Let dep be the current dependent value and ref 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 stop immediately, because the tested IND is unsatisfied. An IND candidate is shown to be a satisfied IND if all dependent values were found in the referenced value set.

The two following algorithms apply this approach. Both use the database to sort and "distinct" the values of all attributes, and then write the sorted lists to disk. However, the order in which those sets are read is quite different. 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 different ways. In Section 4 we shall analyze in detail which of these components dominate the runtime of our algorithms.

Brute Force This algorithm creates and tests all IND candidates sequentially, i. e., one by one. Compared to a SQL join, the main advantage is the implemented early stop for unsatisfied INDs as, of course, most IND candidates are unsatisified and the test can often stop after comparing only a few or even only a single value pair. The brute force algorithm has the disadvantage that each attribute's values are read as often as the attribute is part of an IND candidate.

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 $(n-1)^2$ IND candidates. Thus, the complexity of this approach is $O(nt \log t + n^2 t)$, which is already considerably better than the SQL approach. Furthermore, this worst case complexity is grossly misleading on average, as for most IND candidates only very few values must be read and compared. The number of values read from disk in the client is $O(n^2 \cdot 2t)$, because for each of the candidates we must read at most once all of the two attribute's values.

Single Pass Inclusion DEpendency Recognition (SPIDER) This algorithm eliminates the need to read data multiple times. All IND candidates are created and tested in parallel. SPIDER improves on the algorithm in [2] by using an entirely different data structure for managing cursors, thus reducing the run time by a factor of about 10.

The algorithm first opens all attribute files on disk and starts reading values through one cursor per file. The challenge is to decide when the cursor for each file can be moved. All 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. Consider the following dependent attributes with their sorted value sets $\mathsf{Dep1} = \{1, 2\}$ and $\mathsf{Dep2} = \{2, 3\}$, and the referenced attributes $\mathsf{Ref1} = \{1, 3\}$ and $\mathsf{Ref2} = \{1, 2\}$. The IND candidates are $\mathsf{Dep1} \subseteq \mathsf{Ref1}$, $\mathsf{Dep1} \subseteq \mathsf{Ref2}$, $\mathsf{Dep2} \subseteq \mathsf{Ref1}$, and $\mathsf{Dep2} \subseteq \mathsf{Ref2}$. Let all cursors point to the first item of each attribute. To test $\mathsf{Dep2} \subseteq \mathsf{Ref1}$ the cursor in Ref1 has to be moved, because $2 > 1$. But before this movement the current value of Ref1 is needed to test $\mathsf{Dep1} \subseteq \mathsf{Ref1}$. Vice versa, the cursor in Dep1 can be moved only after the comparison with Ref1. But before, Dep1 has to be compared with Ref2 too.

Despite this mutual dependency, it is possible to synchronize the cursor movements without running into deadlocks or missing some IND candidate tests, because we use *sorted* data sets. We represent each attribute as an *attribute object* providing the attribute's sorted values and a cursor to the current value. An attribute can be covered in multiple IND candidates as referenced attribute and / or as dependent attribute. Thus, each attribute object potentially has two roles.

IND candidates can be divided into distinct sets by their dependent attribute, i. e., all IND candidates covering a dependent attribute define a set. IND candidates are represented in the dependent role of attribute objects as set of

referenced attributes. These referenced attributes are held in two sets distinguishing referenced attributes that are known to contain the currently viewed dependent value (satisfiedRefs) and referenced attributes that are not (yet) known to contain this value (unsatisfiedRefs).

Given these attribute objects we can apply the following algorithm (see Alg. 1). Hold all attribute objects in a min-heap sorted by their currently viewed value. Note that this heap contains one value from each attribute. The following procedure has to be repeated until the heap is empty: Remove all attribute objects with minimal but equal values from the heap and store them in a set Min. Notify each dependent attribute object in Min about each referenced attribute object in Min. This way, the dependent attribute object tracks which referenced attribute includes its currently viewed value and which does not, using the lists satisfiedRefs and unsatisfiedRefs.

After this step, test for all attribute objects in Min if a next value exists. If not, output all INDs that have the attribute object as dependent attribute and all referenced attribute objects in satisfiedRefs as referenced attribute. Otherwise read the next value and update the lists satisfiedRefs and unsatisfiedRefs: Discard all attribute objects in unsatisfiedRefs, because they did not contain the previous dependent value; and move all attribute objects from satisfiedRefs to unsatisfiedRefs as we have not yet seen the currently viewed dependent value in them.

The complexity of the SPIDER algorithm in terms of comparisons is as follows: 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 Min). Thus, the complexity of SPIDER is $O(nt \log n)$ without sorting each attribute's values and $O(nt \log t)$ with sorting, assuming $t > n$. The number of values read from disk is $O(n \cdot t)$, i. e., each value is read at most once. The considerably reduced number of necessary comparisons and the decreased number of read operations are the main improvements over the Brute Force algorithm.

Note that the quadratic number of IND candidates causes no quadratic number of comparisons in n in SPIDER. But it may cause a quadratic number of (inexpensive) movements of attribute objects from unsatisfiedRefs to satisfiedRefs and back. We cannot eliminate this quadratic operations, because in some way we need to represent all IND candidates – potentially, they are all satisfied, which means that the result set must have $(n-1)^2$ elements. This complexity is hidden in all possible algorithms. But on average only a minor fraction of IND candidates is satisfied, and SPIDER find those with only a small number of comparisons.

```
Input: attributes: set of attribute objects; sets
       unsatisfiedRefs in the dependent role of each
       attribute object represent the IND candidates
Output: Set of satisfied INDs.
Min-Heap heap := new Min-Heap( attributes ) ;
while heap != ∅ do
    /* get attributes with minimum value                */
    min := heap.removeMinAttributes() ;
    /* notify dependent attribute                       */
    foreach dep ∈ min & dep in dependent role do
        foreach ref ∈ min & ref in referenced role do
            /* move ref from unsatisfiedRefs to
               satisfiedRefs                            */
            if ref ∈ dep.unsatisfiedRefs then
                dep.unsatisfiedRefs :=
                    dep.unsatisfiedRefs \ {ref} ;
                dep.satisfiedRefs :=
                    dep.satisfiedRefs ∪ {ref} ;
    /* process next value                               */
    foreach att ∈ min do
        if att has next value then
            /* Only sets still satisfied remain         */
            if att in dependent role then
                att.unsatisfiedRefs := att.satisfiedRefs ;
                att.satisfiedRefs := ∅ ;
            att.movePointer ;
            heap.add(att) ;
        else
            if att in dependent role then
                foreach ref ∈ att.satisfiedRefs do
                    INDs := INDs ∪ { att ⊆ ref } ;
    return INDs
```

Algorithm 1: **Algorithm SPIDER.**

The experiments in Section 4 will back 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.

3. Heuristics for Pruning IND candidates

As the complexity of SPIDER depends on the number of attributes, we want to prune IND candidates or, even better, exclude entire attributes. This section describes several pruning heuristics and evaluates their selectivity on the test data sets. These pruning strategies are also applicable to other algorithms for IND detection in the literature (see Sections 4 and 7).

3.1. Simple properties

A simple pruning strategy is to compare the number of distinct values of each IND candidate (called distinct in Table 1): Let $v(A)$ be the set of (distinct) values of attribute A. If $v(\text{dependent}) > v(\text{referenced})$ then there is at least one value in the dependent attribute that is not included in the referenced attribute. Thus, this IND candidate can be excluded.

An equally simple test 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 (called max) or (ii) if the minimum dependent value is lower than the minimum referenced value (called min).

The selectivity of these filters on our test databases is given in Table 1. The top section of the table gives the number of attributes in the schemas, the number of IND candidates, the actual number of satisfied INDs, and the number of attributes covered by at least one satisfied IND. These last two numbers provide a lower bound for the filters. The combination of all filters provides – as expected – the best selectivity on both, IND candidates and the number of attributes. For the full PDB data set the number of relevant attributes is reduced from $1,290$ to 949 and the number of IND candidates is reduced by $\sim 75\%$. In all cases, the number of IND candidates is reduced by at least a factor of 4.

	UniProt	TPC-H	PDB	
			2.7 GB	21 GB
# attributes	68	61	1,208	1,290
# IND cand.	1,393	877	216,659	242,970
# INDs	36	33	30,753	34,988
# attr. in INDs	31	20	593	661
distinct				
# IND cand.	910	477	139,356	158,432
# attr. in IND cand.	68	58	1,208	1,290
distinct & max				
# IND cand.	541	295	74,588	85,901
# attr. in IND cand.	59	54	1,003	1,073
distinct & min				
# IND cand.	345	275	91,998	101,640
# attr. in IND cand.	54	57	1,082	1,149
dist. & max & min				
# IND cand.	174	137	45,957	51,393
# attr. in IND cand.	49	52	891	949

Table 1. Number of remaining IND candidates and attributes after pruning using distinct**,** max**, and** min**.**

3.2. Bloom filter

The simple filters described 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. We used Bloom filter to better adapt the filter to the data [5]. Therefore, we hash each attribute's values into a bit array, such that each bit represents several values. To filter IND candidates we compare the bit arrays of an IND candidate looking for bits that are set in the dependent bit array, but not set in the referenced bit 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 $\text{dep} \wedge \neg\text{ref}$ operation, such that in the resulting bit array a bit is set iff the IND candidate can be excluded.

To achieve an optimal impact on filtering complete attributes, we experimented with the size of the bit array and the hash function. Further, we examined how hashing only prefixes of certain length instead of hashing complete attribute values affects filtering, because thus the hash value can be computed much faster and we already expected high selectivity in the first few characters. The results of our experiments are shown in Figure 1.

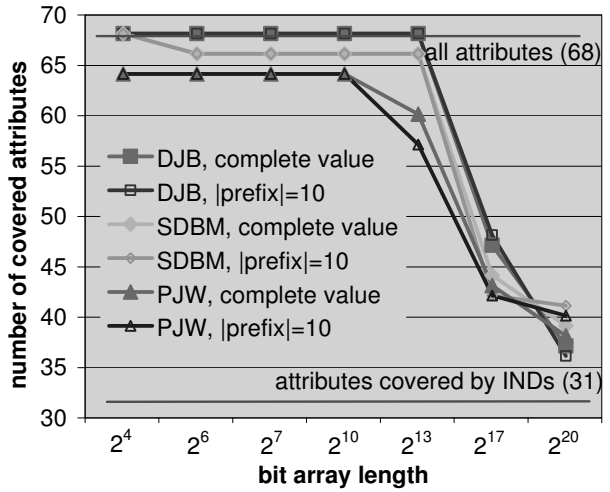
The results confirm the intuition that increasing the size of the bit array leads to a higher amount of pruned IND candidates and attributes due to improved spreading of the values over the longer bit array. On the other hand, large arrays require large amount of memory and more time for their comparison. The experiments show that a length of 2^{17} bit (which requires about 20MB memory for 1000 attributes) already is optimal for the PDB with respect to pruned attributes and also has very good effects on UniProt. Longer arrays improve pruning on UniProt only marginally. Although these figures are highly data set dependent, they show that reasonable reductions can be achieved with modest memory consumption.

We tested the PJW, DJB, and SDBM hash functions[5]. 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.

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. We tested on prefix lengths between 1 and 10. Interestingly, hashing prefixes of fixed length already leads to impressive results in pruning on even 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

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

(a) UniProt



(b) PDB 2.7 GB

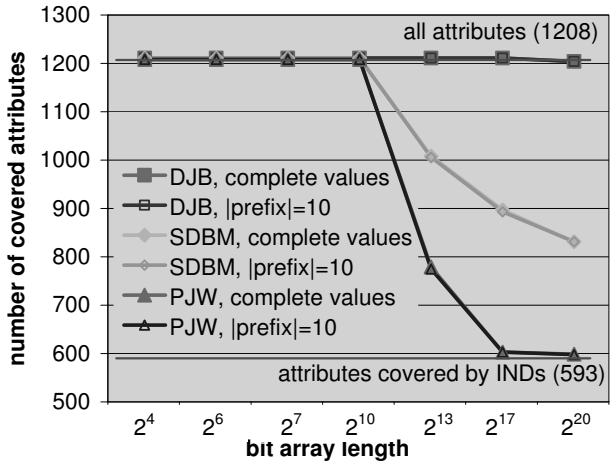


Figure 1. Impact of varying Bloom filter parameters on number of filtered attributes for (a) UniProt and (b) PDB.

is even slightly better than hashing complete values. Additionally, hashing only prefixes is more efficient.

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. Because we chose these databases representatively for the life sciences domain, we expect good performance at least on other biological data sources.

The effects of combining Bloom filter (parameters as above) with the simple pruning strategies are shown in Table 2. The Bloom filter is in almost all cases more selective than the filter on number of distinct items, maximum, and minimum. But the simple filters exclude IND candidates that are not pruned by the Bloom filter. Therefore, all filters together reach the best performance.

	UniProt	TPC-H	PDB	
			2.7 GB	21 GB
# attributes	68	61	1,208	1,290
# IND cand.	1,393	877	216,659	242,970
# INDs	36	33	30,753	34,988
# attr. in INDs	31	20	593	661
distinct & bloom				
# IND cand.	245	43	35,044	40,307
# attr. in IND cand.	42	26	600	798
dist., max, min, bloom				
# IND cand.	125	40	34,673	39,399
# attr. in IND cand.	35	25	595	665

Table 2. Number of remaining IND candidates and attributes using Bloom filter and simple pruning.

As previously mentioned, the overall runtime of our algorithms 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 in advance in the database is also costly, because the database must perform additional sort operations. For instance, to obtain the minimum and maximum value and the number of different values for an attribute the database must sort the entire bag of values of this attribute – a sort that will be repeated later for all attributes that cannot be excluded completely. For Bloom filters, expensive computations must take place that might overweight 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, but 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 the next section.

4. Experimental Results for Unary INDs

We tested all algorithms on the databases presented in Sec. 1.2 on a Linux system with 2 Intel Xeon processors (2.60 GHz) and 12 GB RAM. We use a commercial object-relational database management system.

We only create IND candidates with an unique referenced attribute, such that we can infer foreign keys. Second, we exclude intra-table references. Third, we always used the distinct filter. The runtime effects of the other filters were tested individually.

Results can be seen in Table 3. Both algorithms outperform the fastest SQL approach (join). The difference

mostly depends on the number of attributes and thus IND candidates and not so much on the size of the database, as expected from our complexity analysis in Section 2. For large schemas all SQL approaches are inapplicable.

For low numbers of IND candidates, i. e., UniProt and TPC-H, there is just a small difference between the Brute Force and the SPIDER algorithm. For large schemas with high numbers of IND candidates the improvement of SPIDER over Brute Force is considerable.

	UniProt	TPC-H	PDB	
DB size	667 MB	1.3 GB	2.7 GB	21 GB
# IND cand.	910	477	139,356	158,432
# INDs	36	33	30,753	34,988
`join`	15 m 03 s	29 m 22 s	> 7 days	–
Brute Force	2 m 01 s	6 m 56 s	3 h 13 m	20 h 21 m
SPIDER	1 m 38 s	6 m 40 s	19 m 06 s	3 h 52 m

Table 3. Run-time performance of the different algorithms.

4.1. Comparison to other approaches

We also compared our algorithms with the approaches of Bell and Brockhausen [3] and Marchi et al. [11] using our own implementation. Those approaches only test candidates of same data type. In our life sciences setting, we often find schemas with only `string` attributes and thus must test all pairs of attributes (note that most parsers are written in Perl). For a fair comparison we assigned the same data types to all attributes. The algorithm of Bell and Brockhausen leverages filters on maxima and minima to prune the IND candidates and utilizes already finished IND tests for further pruning exploiting transitivity of IND. IND candidates are tested by SQL join statements. It runs 4 m 39 s on UniProt data which is three times slower than SPIDER. On the 2.7 GB fraction of PDB it did not finish within 24 hours.

The approach of Marchi et al. preprocesses all data assigning to each value in the database all attribute's names that include this value. The results are stored in tables. Afterwards all IND candidates are tested in parallel exploiting the sets of attribute names. However, the preprocessing on UniProt already takes 9 h 45 m (the actual IND test only 2 m 10 s).

4.2. Effects of pruning

Table 4 shows run times when we apply filtering 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 SPIDER to the remaining IND candidates. In Table 4, we differentiate between two components: (i) the time to sort and read (inside the RDBMS), ship (to the client) and write the data (*SRSW*), and (ii) the time to read and test the data at the client (*test*). The third component of the overall runtime consists of writing the satisified INDs to the DBMS. As these runtimes are equal for both versions they are not given in Tab. 4.

	UniProt	TPC-H	PDB	
DB size	667 MB	1.3 GB	2.7 GB	21 GB
distinct	1 m 38 s	6 m 40 s	19 m 06 s	3 h 52 m
SRSW	1 m 22 s	6 m 31 s	14 m 29 s	3 h 41 m
test	15 s	8 s	2 m 48 s	9 m 30 s
distinct & max & min & bloom	1 m 40 s	6 m 50 s	19 m 23 s	3 h 58 m
SRSW	1 m 26 s	6 m 41 s	14 m 44 s	3 h 47 m
test	12 s	8 s	2 m 45 s	8 m 56 s

Table 4. SPIDER with and without filtering of IND candidates.

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. On the other side, the runtime for testing decreases only slightly. This shows that the runtime is almost independent on the number of IND candidates, i. e., representing and tracking the IND candidates incurs almost no cost as stated by the complexity estimation in Section 2. It also shows that the exclusion of entire attributes (not just candidates) also did not yield much speed up. The reason is that the number of satisfied INDs is the same in all cases, which means the number of processed items does not decrease as much as the number of attributes.

Thus, saving unsatisfied INDs by filtering in the end does not save much time for testing – but costs time for computing the filters.

4.3. Further ideas

To find other ways of further improving SPIDER we analyzed the time consumption in more detail. Of the 19 minutes to test the 2.7 GB part of the PDB, approx. 15 minutes are spent in the SRSW phase and less than 3 minutes in the test phase (see Tab. 4). As the SRSW phase dominates the overall time, it is a natural idea to reduce the amount of data to be shipped to the client by applying filtering already inside the RDBMS.

However, this idea is not as straight-forward as one might think. The 15 minutes of the SRSW phase split up into 10 minutes for reading and sorting the data inside the database, 4 minutes to read and ship them out of the database, and 1 minute to write them to disk. Thus, any filtering which requires to scan or sort the data in first place (such as min, max or distinct) will very likely not improve the overall performance, because the time saved during shipping and testing will probably be dominated by the

time required to scan/sort the data. Reading those values from the database catalogue opens the door to wrong results due to outdated statistics. We also experimented with a database-internal implementation of the Bloom filter but could not obtain any improvements for the same reasons.

5. Detecting Composite INDs

Composite INDs can be identified by creating and testing IND candidates levelwise based on satisfied INDs on the previous level, because satisfied INDs of lower levels are a precondition for a composite IND. For instance, if $A \not\subseteq C$, then there exists no B, D such that $AB \subseteq CD$ or $BA \subseteq DC$.

All IND candidates of higher levels can be created by an adapted AprioriGen algorithm using an order on attributes [11]. The IND candidates of level l are generated by sorting all satisfied INDs of level $l-1$ by the first $l-2$ attribute pairs. An IND candidate is generated from two level $l-1$ INDs with the same first $l-2$ corresponding attributes and different attributes at position $l-1$. The generated IND candidate is the combination of these two INDs. Only if all INDs of level $l-1$ that are implied by the generated IND candidate are satisfied the IND candidate must be tested. For example, if we verified the INDs $AB \subseteq DE$ and $AC \subseteq DF$, we will generate the IND candidate $ABC \subseteq DEF$ on the next level. We must test this IND candidate if the implied INDs $AB \subseteq DE$, $AC \subseteq DF$, and $BC \subseteq EF$ are satisfied. In [11] the authors show that this algorithm enumerates all possible composite INDs.

We apply our SPIDER algorithm with minor modifications to test the IND candidates of each level. The entire algorithm leverages the advantage of SPIDER – the independence of the number of IND candidates. Therefore, the possibly large number of created IND candidates is acceptable.

To obtain all necessary satisfied unary INDs we cannot restrict the IND candidates to unique referenced attributes as we did before, because we would miss attributes that might be part of a larger IND. The results of testing exhaustively all IND candidates using SPIDER are given in Table 5. In comparison to Table 3 the runtime increases slightly, because more INDs are tested and satisfied. This implies that more attribute 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 much larger number of INDs has to be saved.

To test composite IND candidates we modified the SPIDER algorithm in several aspects. We hold attribute tuples and their values in the min-heap, instead of single attributes. For each level, we query the sorted composite data sets from the database and write them to disk. This is necessary, because we need the correct associations of the attribute values in tuples, which cannot be recovered from the single attribute value files.

	UniProt	TPC-H	PDB	
DB size	667 MB	1.3 GB	2.7 GB	21 GB
# IND cand.	2,235	1,616	729,674	831,984
# INDs	116	86	84,232	99,669
exhaustive SPIDER	1 m 45 s	6 m 55 s	23 m 27 s	4 h 05 m

Table 5. Experimental results of testing unrestricted IND candidates using SPIDER.

The results of our experiments are shown in Table 6. No composite IND with more than two attributes was found. The runtime increases mostly because of the requirement to read and write tuples for each level.

	UniProt	TPC-H
composite SPIDER	5 m 25 s	27 m 29 s
level 1		
# IND cand.	2,235	1,616
# attribute tuples	68	61
# INDs	116	86
SRSW	1 m 24 s	6 m 31 s
test	19 s	23 s
level 2		
# IND cand.	20	59
# attribute tuples	14	36
# INDs	13	21
SRSW	3 m 20 s	19 m 33 s
test	19 s	2 m 00 s

Table 6. Experimental results for detecting composite INDs.

We did not test this algorithm on PDB, because of the large number of satisfied unary INDs. These INDs are mostly based on inclusions of surrogate keys among each other and are therefore unreasonable candidates for being part of an actual composite foreign key. We currently work on methods to filter these INDs before using them for higher-level INDs.

6. Detecting Partial INDs on Dirty Data

In most real world databases one finds dirty data. Often foreign keys are not defined or not checked for performance reasons. Another reason are faulty parsers for importing data into a RDBMS that do not maintain existing constraints correctly. We call such "foreign keys" with exceptions "dirty foreign keys".

We cannot find all dirty foreign keys by testing the IND candidates as before. The SPIDER algorithm – with some minor modifications – is able to detect partial INDs, i. e., INDs that allow a certain number of dependent values that

are not included in the referenced attribute's values. There are two possible ways of counting these dependent values: (i) the number of all *distinct*, not included values expressed as a percentage or (ii) the *absolute* number of not included values. Both values are helpful to rate a partial IND.

To collect the number of all distinct, not included values, we add a counter to the references in each dependent attribute object in the lists satisfiedRefs and unsatisfiedRefs. These counters represent the number of values of this dependent attribute object that are not included in the referenced attribute. This modification applies only to the part of processing the next value (see Algorithm 2): Instead of discarding all referenced attribute objects in unsatisfiedRefs we increase the counter of these objects. Only if a given threshold of allowed violating values is exceeded, the referenced attribute object will be removed. The INDs result from all referenced attributes in satisfiedRefs and unsatisfiedRefs.

```
/* process next value                                    */
foreach att ∈ min do
    if att in dependent role then
        /* Only sets still satisfied remain.             */
        foreach ref ∈ att.unsatisfiedRefs do
            ref.counter++ ;
            if ref.counter > threshold then
                att.unsatisfiedRefs :=
                    att.unsatisfiedRefs \ {ref} ;
    if att has next value then
        if att in dependent role then
            att.unsatisfiedRefs :=
                att.unsatisfiedRefs ∪ att.satisfiedRefs ;
            att.satisfiedRefs := ∅ ;
        att.movePointer ;
        heap.add(att) ;
    else
        if att in dependent role then
            foreach ref ∈ (att.satisfiedRefs ∪
                    att.unsatisfiedRefs) do
                INDs := INDs ∪ { att ⊆ ref } ;
```

Algorithm 2: Modification in algorithm SPIDER to test partial IND candidates.

To obtain the absolute number of not included values we need the number of occurrences for each value in the dependent attributes, which can be extracted from the database. Each not included dependent value has then not only to be counted, but to be multiplied by its number of occurrence.

The results of our experiments are shown in Table 7. We allowed 5% violating items in the dependent value. Interestingly there are indeed a considerable amount of dirty foreign keys in three of the four data sets. Again, one can see in comparison to the results of exact tests (see Table 3) that the runtime increases only minimally. The difference is based on the additional costs for counting the number of not included dependent values and 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 than in the non-partial version.

	UniProt	TPC-H	PDB	
DB size	667 MB	1.3 GB	2.7 GB	21 GB
# IND cand.	1,393	877	216,659	242,970
# INDs	36	40	35,245	40,106
partial SPIDER	1 m 42 s	6 m 45 s	23 m 34 s	4 h 02 m

Table 7. Results for finding partial INDs. 5 % of not included dependent values were allowed.

7. Related Work

Bell and Brockhausen propose to use SQL join statements to evaluate unary IND candidates [3] after min/max filtering (see Section 4.1). Known foreign keys and already tested IND candidates (satisfied and unsatisfied INDs) are used for further pruning. SPIDER clearly outperforms this approach even without any knowledge of existing foreign keys as shown in Sec. 4.1.

Marchi et al. detect unary INDs among same data types by preprocessing all data and then testing all IND candidates in parallel [11]. The preprocessing assigns to each value in the database a 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. 4.1 that SPIDER outperforms this approach by orders of magnitude. Further, the authors give a levelwise approach for detecting composite INDs. We employ their approach for IND candidate creation but test the candidates with our SPIDER algorithm.

Marchi et al. extend the levelwise approach for detecting composite INDs in [12]. The 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 propose to create composite IND candidates by finding cliques in k-uniform hypergraphs [9]. These hypergraphs are built of satisfied INDs of lower levels. Unary and binary INDs are tested by an approach similar to [3]. [12] and [9] imply tests for single IND candidates on diverse levels. The strength of SPIDER is – opposite to this – its independence of the number of IND candidates.

Dasu et al. apply data summaries to detect join paths approximately, i. e., to detect INDs [7]. They use set resem-

blance and multiset resemblance to find a join path, its size and direction. The authors use this approach as a first step in schema discovery to help a human expert. In our scenario we want to be able to surely detect satisfied INDs and therefore need an exact algorithm.

Brown and Haas study algebraic constraints between pairs of attributes to utilize them in query optimization [6]. They create IND candidates by heuristics and data samples and might therefore miss some INDs. SQL join statements are utilized 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 [13]. These joins are then tested against the database and rated by a human expert.

8. Conclusion

We described the SPIDER algorithm for detecting inclusion dependencies in an RDBMS with no previously known schema information. It is divided into two phases. The first phase leverages optimized sort operations of the DBMS but avoids the constraints of SQL. The second phase tests all IND candidates in parallel such that all data values are read only once and tests are stopped early. We showed its superiority to other approaches in terms of complexity and by experiments on different data sets. SPIDER is the only method that allows a feasible detection of INDs in databases with large numbers of attributes and data values.

Furthermore, we presented and analyzed pruning strategies on IND candidates. We showed and explained in detail that database-external pruning does not speed up the computation. We also discussed possibilities to perform database-internal pruning and showed why this is harder than one might expect. However, we will further investigate this option.

We extended SPIDER to also find composite and partial INDs. Both tasks can be solved by minor modifications of the algorithm. The test of partial INDs is very fast, again, due to the efficient test and the independence of the number of IND candidates. Testing composite INDs implies additional sorts on the composite attributes for each level, which increases the runtime heavily when many IND candidates are satisfied. Thus, we believe that prior to higher level one should apply methods to discern INDs from real foreign key constraints using heuristics. This is the second line of research we are following.

Finally, we are working to integrate SPIDER into the Aladin framework for identifying intra-source and inter-source relationships. This step also requires heuristics to discern INDs from foreign keys, and a framework to compute the sensitivity and specificity of foreign key detection using gold standards.

Acknowledgments. This research was supported by the German Ministry of Research (BMBF grant no. 0312705B) and by the German Research Society (DFG grant no. NA 432).

References

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

[2] J. Bauckmann, U. Leser, and F. Naumann. Efficiently computing inclusion dependencies for schema discovery. In *Second International Workshop on Database Interoperability. In Workshop-Proceedings of the ICDE 06*, 2006.

[3] S. Bell and P. Brockhausen. Discovery of data dependencies in relational databases. In Y. Kodratoff, G. Nakhaeizadeh, and C. Taylor, editors, *Statistics, Machine Learning and Knowledge Discovery in Databases, ML–Net Familiarization Workshop*, pages 53–58, 1995.

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

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

[6] P. Brown and P. J. Haas. BHUNT: Automatic discovery of fuzzy algebraic constraints in relational data. In *29th International Conference on Very Large Data Bases (VLDB '03)*, pages 668–679, 2003.

[7] T. Dasu, T. Johnson, S. Muthukrishnan, and V. 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.

[8] M. Kantola, H. Mannila, K.-J. Rih, and H. Siirtola. Discovering functional and inclusion dependencies in relational databases. *International Journal of Intelligent Systems*, 7:591–607, 1992.

[9] A. Koeller and E. A. Rundensteiner. Discovery of high-dimensional inclusion dependencies. In *19th International Conference on Data Engineering*, pages 683–685, 2003.

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

[11] F. D. Marchi, S. Lopes, and J.-M. Petit. Efficient algorithms for mining inclusion dependencies. In *8th International Conference on Extending Database Technology (EDBT '02)*, pages 464–476. Springer-Verlag, 2002.

[12] F. D. Marchi and J.-M. Petit. Zigzag: a new algorithm for mining large inclusion dependencies in databases. In *Third IEEE International Conference on Data Mining*, pages 27–34, 2003.

[13] J.-M. Petit, F. Toumani, J.-F. Boulicaut, and J. Kouloumdjian. Towards the reverse engineering of denormalized relational databases. In *12th International Conference on Data Engineering (ICDE '96)*, pages 218–227, 1996.