Efficiently Computing Inclusion Dependencies for Schema Discovery

Jana Bauckmann Ulf Leser Felix Naumann

Department for Computer Science, Humboldt-Universität zu Berlin
Rudower Chaussee 25, 12489 Berlin, Germany
{bauckmann,leser,naumann}@informatik.hu-berlin.de

Abstract

Large data integration projects must often cope with undocumented data sources. Schema discovery aims at automatically finding structures in such cases. An important class of relationships between attributes that can be detected automatically are inclusion dependencies (IND), which provide an excellent basis for guessing foreign key constraints. INDs can be discovered by comparing the sets of distinct values of pairs of attributes.

In this paper we present efficient algorithms for finding unary INDs. We first show that (and why) SQL is not suitable for this task. We then develop two algorithms that compute inclusion dependencies outside of the database. Both are much faster than the SQL-based methods; in fact, for larger schemas they are the only feasible solution. Our experiments show that we can compute all unary INDs in a schema of $1,680$ *attributes with a total database size of* 3.2 *GB in approximately* 2.5 *hours.*

1. Problem Description

In large integration projects one often copes with undocumented data sources, i. e., sets of relations with attributes, without any knowledge about how these schema elements refer to each other. Recovering their structure, which is a prerequisite to semantic integration, is an interesting challenge in which users must be supported as much as possible by automated analysis. Two approaches can be distinguished [16]. Schema-driven approaches analyze only the schema, i. e., try to find relationships between schema elements by analyzing their names. In contrast, instance-driven (or data-driven) approaches directly analyze the data of a given database instance. Since names of schema elements very often carry little meaning, over the last years research in this area has mostly concentrated on the instance-driven approach [4, 6].

In relational databases, the most important means to structure data are foreign key constraints indicating a 1:n relationship between two relations. Foreign key constraints are essentially inclusion dependencies, i. e., the assertion that the set of values of the referring attribute is completely contained in the set of values of the referenced attribute. Thus, if no foreign key constraints are known, one can derive strong guesses about their existence by finding pairs of attributes A and B such that all values of A also exist as values of B. Clearly, these guesses have to be confirmed by a user using background knowledge, as foreign key constraints imply set inclusion but not vice versa. However, in our experience cases of set inclusion without a corresponding foreign key constraint only happen in certain settings which can be filtered by additional heuristics (see Sec. 5). Note that by finding all INDs in a given database, algorithms can produce only false positives, but no false negative foreign key constraints.

1.1. Background and Motivation

The presented study is part of the Aladin project for **Al**most h**a**nds-off **d**ata **in**tegration[1] in the life sciences [9]. The architecture of Aladin can be seen in Figure 1. Integration is performed in five steps: In the first step, data sources are downloaded in whatever format and imported into the Aladin database. If source databases are not available in a relational format, this step (and only this step) requires manual intervention for writing a parser and loading the database. However, there is a large degree of freedom in the design of the target schema, as Aladin makes no assumption about the generated schema; in particular, it does not expect any constraints to be in place.

In the second step candidates for primary keys are computed using the uniqueness constraint for keys. In the third step, intra-source relationships are computed using set inclusion and several heuristics. In the life science domain databases typically contain one major class of data with

[1] http://www.informatik.hu-berlin.de/wbi/research/aladin

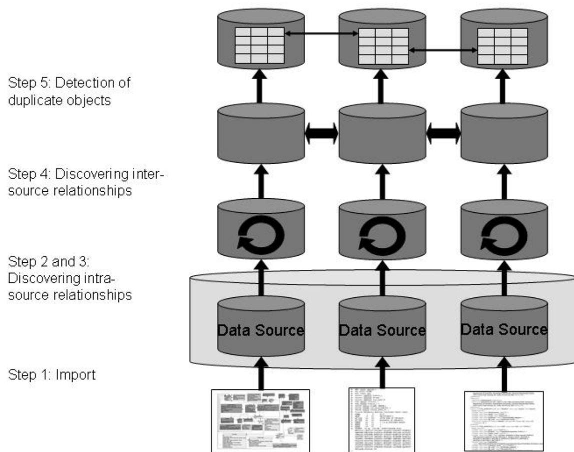

Figure 1. Architecture of Aladin.

several annotations. So the relation representing this major class of data is chosen as "primary relation". This is only possible by using domain knowledge, i. e., exploiting typical properties of biological databases (see Sec. 5). In the fourth step relationships between data sources are inferred by again using set inclusion and domain-specific heuristics. This step only considers primary relations as targets, thus drastically reducing the search space. In the fifth step duplicate objects are detected and flagged. More details on Aladin can be found in [9].

The present paper describes methods for implementing essential parts of the third and forth step. Efficiency is crucial, especially during the development phase of Aladin where different heuristics for improving the relationship guesses have to be tested. Using our first implementation of set inclusion in SQL on the Protein Data Bank (PDB) database [3]—using only a fraction of the actual database comprising 2.7 GB—did not finish within seven days, which considerably impeded further development.

The problem of schema discovery also appears in other applications. For instance, we recently compared four different public microarray databases [13]. Three of them came with virtually no documentation, and two had no predefined integrity constraints. One reason for omitting constraints in biological RDBMS is intended compatibility with MySQL, which until recently did not support foreign key constraints. Further applications of schema discovery are described in [15] and [12].

1.2. A High-level Solution

Consider two attributes a and b. For an attribute a, let $v(a)$ be the bag of all its values and $s(a)$ be the sorted set of its different values using an arbitrary but fixed sorting criteria. We call $a \subseteq b$ an *IND candidate*, which is *satisfied* iff $s(a) \subseteq s(b)$.

To test if the IND candidate $a \subseteq b$ is satisfied, a natural method is to retrieve both $v(a)$ and $v(b)$, sort them with duplicate removal to compute $s(a)$ and $s(b)$, and then scan linearly through both sets. Starting from the smallest items we can decide for each item in $s(a)$ if it is contained in $s(b)$ while iterating once over both sets. The exact procedure is described in Sec. 3.1. It requires $O(|v(b)| \times log(|v(b)|) + |v(a)| \times log(|v(a)|))$ comparisons for sorting and duplicate removal and, for $s(b) > s(a)$, $O(|s(b)|)$ comparisons to test for set inclusion. How many INDs do we have to test, given a database with n attributes? For a pair of attributes, we test for $a \subseteq b$ if $|s(a)| < |s(b)|$ and for $b \subseteq a$ if $|s(a)| > |s(b)|$; if both sets are of equal size, we also need to perform only one test (which is then a test for set equivalence). Thus, we need to perform $\frac{n^2-n}{2}$ set inclusion tests to find all INDs. Given a large database with many attributes and large numbers of tuples in each relation, the entire procedure might take considerable time.

There are two obvious optimizations rendering our worst case analysis as too pessimistic in most cases. First, if we can store sorted sets we have to sort each attribute only once and not for each IND test. Second, assume we have computed $s(a)$ and $s(b)$ and let $|s(a)| < |s(b)|$. When we compare those sets, we can immediately stop as soon as we find one element of $s(a)$ that is not included in $s(b)$. Frequently this happens after only a few comparisons. Note that each satisfied IND candidate requires a complete scan through at least the smaller set.

1.3. Implementation

There are two ways to implement a procedure as the one just described. Assuming that the database to analyze is actually installed on a relational database management system (RDBMS), it is natural to *perform all tests directly inside the RDBMS using SQL*. This leverages the highly efficient methods for sorting in RDBMS and avoids any data having to be shipped to clients. However, we shall see that we cannot force SQL to implement the two optimizations described above. The second possibility is a client program. We can retrieve sorted sets from the RDBMS and then *perform the inclusion tests on the client*. The drawback is that data is shipped out of the database, and that temporary storage, incurring additional I/O, is necessary if sets are too large to fit into main memory. Thus, it is not immediately clear which method is better. Our paper is a contribution to answer this question.

1.4. Test Data

We tested our algorithms at three different data sets from our application domain, i.e., the life sciences.

- UniProt[2] is a database of annotated protein sequences [1]. It is available in several formats. We used the BioSQL[3] schema and parser. It consists of 85 attributes in 16 tables. The total size of the database is 667 MB, with the largest attribute having approximately 1 million different values.
- SCOP[4] is a database of protein classification available as a set of files [14]. We wrote our own parser, populating 4 tables with 22 attributes. The total size of the database is 17 MB, with the largest attribute having 94,441 different values.
- PDB is a large database of protein structures [3]. We used the OpenMMS[5] software for parsing PDB files into a relational database. The OpenMMS schema consists of 1,711 attributes in 115 non-empty tables, with a total size of 21 GB, with the largest attribute having approximately 152 million different values. The fraction of the PDB that we use for our tests is of size 2.7 GB, with the largest attribute having approximately 16 million different values.

We defined indexes on the data as given by the used schemas; for SCOP we did not define any indexes. We ran all tests on a Linux system with 2 processors and 4 GB RAM using a commercial object-relational database management system.

1.5. Contributions

In the important field of schema discovery we make several contributions toward the problem of efficiently detecting unary inclusion dependencies among attributes:

- We present several SQL queries for IND discovery and show that and why they usually perform badly despite advanced optimization techniques.
- We develop two algorithms for discovery, analyze their efficiency, and show that they are superior to the in-database approach.
- Finally, we bed our results into a life sciences data integration scenario where automatic schema discovery is particularly important, due to abundant heterogeneity and poor documentation.

1.6. Structure of This Paper

In the following two sections we present the two types solutions, each with experimental results: In Sec. 2 we show three variants for inclusion dependency detection using SQL. In Sec. 3 we present our new and more efficient approaches, which let the database engine perform sorting but perform the inclusion tests outside of the database. Section 4 covers scalability issues at schema and at system level. In Sec. 5 we turn back to our application scenario of discovering foreign keys in life sciences data and analyze the effectiveness of this approach. Finally, Sec. 6 reviews other work in this area and Sec. 7 concludes with an outlook on future work.

2. Approaches Using SQL

The most obvious approach for solving a problem of sorting and set comparisons where the data is stored inside an RDBMS is using the internal capabilities of the database management system. So our first approaches use SQL for performing set inclusion tests. We propose three alternative statements to describe the problem: using `join`, `minus`, and `not in`. In each case, only one query is necessary to perform the actual test for a pair of attributes.

We build IND candidates by choosing pairs of potentially dependent attributes and potentially referenced attributes. We define potentially dependent attributes (or dependent attributes for short) as non-empty columns of any type except LOB and potentially referenced attributes (or referenced attributes) as non-empty unique columns. Note that each referenced attribute is also in the set of dependent attributes, but not vice versa.

The satisfiedness of these IND candidates is tested in two phases: The first phase is a pretest on the cardinality of the distinct values of both attributes as described in Sec. 1.2, as the IND candidate cannot be satisfied if the number of distinct values of the dependent attribute is greater than the number of distinct values of the referenced attribute. The second phase executes an SQL statement to verify the IND candidates. In all SQL statements dependent tables and columns are prefixed with 'dep', and referenced tables and columns are prefixed with 'ref'.

2.1. Three SQL Statements

Utilizing `Join` The first statement utilizes a `join` as proposed by [2]. We perform a join on the assumed inclusion dependency and test the number of returned tuples against the number of non-null values in the dependent attribute. The IND is satisfied if both values are equal. The statement can be seen in Figure 2.

This statement simply computes a join over two sets, which is a different problem than IND test. We can use a join to verify INDs, but we cannot tell the RDBMS engine that the procedure can be stopped as soon as any tuple is detected for which no join partner exists. Furthermore, there is no way to give the database this hint using SQL and a join. Therefore, we formulate two other statements that aim to express this point. The idea is to find a statement that

[2] http://www.pir.uniprot.org
[3] http://obda.open-bio.org
[4] http://scop.mrc-lmb.cam.ac.uk/scop
[5] http://openmms.sdsc.edu

```
select  count(*)  as  matchedDeps
from    (depTable  JOIN refTable
on      depTable.depColumn = refTable.refColumn)
```

IND candidate is satisfied $\Leftrightarrow$
$|\text{matchedDeps}| = |\text{non}-\text{null dependent values}|$

Figure 2. Statement utilizing `join`.

returns no tuples if the IND candidate is satisfied and one or more tuples otherwise. This way, we can stop the computation after the first tuple in the result set, hoping that first tuples can be produced without computing the entire result (i.e., without sorting or grouping).

Utilizing `Minus` The idea of utilizing `minus` is to subtract values of the referenced attribute from values of the dependent attribute; if there are tuples in the result set then the IND candidate is not satisfied. The statement can be seen in Figure 3. Note that this requires product-specific, non-standard SQL elements to enable the query engine to stop execution early.

```
select  count(*)  as  unmatchedDeps  from
  ( select  /*+  first  rows (1)  */  *
    from
      ( select   to_char (depColumn)
        from    depTable
        where   depColumn is not null
        MINUS
        select   to_char (refColumn)
        from    refTable )
    where   rownum < 2)
```

IND cand. is satisfied $\Leftrightarrow$ $|\text{unmatchedDeps}| = 0$

Figure 3. Statement utilizing `minus`.

Utilizing `Not In` Another possibility to obtain an empty result set if the tested IND candidate is satisfied is to utilize `not in`. The idea is to ask for values in the dependent attribute that are not included in the referenced attribute. Again, we can restrict the result set to a single tuple using product-specific tricks, as can be seen Figure 4.

2.2. Experimental Results

We measured the required time for computing all INDs for each of the three methods and for each of the three data sets. The measured times together with the number of satisfied inclusion dependencies are given in Tab. 1. We observe that the join approach is the fastest alternative, despite its obvious inability to stop execution after the first mismatch. We believe the reason lies in the extensive optimization of

```
select  count(*)  as  unmatchedDeps  from
  ( select  /*+  first  rows (1)  */  depColumn
    from    depTable
    where   depColumn NOT IN
              ( select   refColumn
                from    refTable )
            and rownum < 2 )
```

IND cand. is satisfied $\Leftrightarrow$ $|\text{unmatchedDeps}| = 0$

Figure 4. Statement utilizing `not in`.

join operations in RDBMSs. Surprisingly, our attempts to enable early stops using `minus` or `not in` failed, which is probably caused by the special implementation of the `rownum` function that obviously is not merged with the inner queries during query rewriting. However, since even the join times are much too slow for our purposes (see column three in Table 1), we didn't research deeper into the reasons for this behavior.

All three measured are not applicable to discover all INDs in a database of the size of the PDB. We first ran tests on the entire PDB, but stopped after two days because the RDBMS estimated a particular table-scan to last several more days. We then eliminated the biggest PDB tables, containing atom coordinates for each atom in each protein, and thus reduced the database size from 21 GB to 2.7 GB. The discovery procedure did not finish within seven days even for this reduced data set.

The problems with using SQL for set inclusion are twofold. First, one cannot tell the optimizer what the real question is, and that, as a consequence, there are optimization strategies that are clearly better than those used for "ordinary" SQL statements. The optimizer therefore fails to perform the early stop after a first mismatch, described in Sec. 1.2. Second, we have to run a single SQL statement for each pair of attributes to be tested. Thus, the engine cannot exploit a strategy where sorts of value sets are performed only once for each attribute and later re-used in each test. As relational databases by design do not store sorted sets[6], there is no way to implement such strategies using SQL alone.

	UniProt	SCOP	PDB
# IND candidates	910	43	139,356
# satisfied INDs	36	11	30,753
`join`	15 min 03 s	7.3 s	> 7 days
`minus`	29 min 16 s	14.3 s	-
`not in`	1 h 53 min	46 min	-

Table 1. Experimental results utilizing SQL. We used only a 2,7 GB fraction of PDB.

[6]Note that product-specific feature such as index-organized tables are no solution, as one would have to build a new table for each attribute.

3. Using Order on Data

As the previous section showed, there is no possibility to describe the problem of verifying IND candidates in SQL in an efficient way. We can elegantly describe the task in various ways in SQL, all leading to the same correct result. But all tested SQL approaches could not take advantage of the simple ideas described in the introduction. In the following, we present two algorithms that follow exactly these ideas, using the RDBMS only for tasks it is good at: We first extract from the database the sorted sets of distinct values of each attribute using SQL. Second, we test the satisfiedness of the IND candidates with a Java program.

The first approach is called *brute force* (see 3.1), because it tests one IND candidate at a time and therefore has to read value sets multiple times. The second approach is called *single-pass* (see 3.2), because it reads all value sets only once and tests all IND candidates in parallel.

3.1. Brute Force

The brute force approach creates all IND candidates while iterating over all dependent and referenced attributes. Each created IND candidate is tested directly after its creation. Therefore, the algorithm iterates $n \times m$ times over the data of attributes where n is the number of all referenced attributes and m is the number of all dependent attributes. We implement this as follows: First, we extract sorted sets of distinct attribute values from the database and store them in sorted files. For each pair of one dependent and one referenced attribute, we open and traverse through both files. Obviously, we could reduce I/O by first loading an entire dependent attribute set and then test it against each referenced set. To cope with limited memory, an approach similar to a block-nested-loop join would help. We did not test such an algorithm, but instead developed a more sophisticated optimization described in Sec. 3.2.

The test of a single IND candidate is described in Algorithm 1. We iterate over both data sets starting from the smallest item. For each dependent item we look for an equal item in the referenced items list by stepping to the next referenced item if the referenced item is less than or equal to the dependent item. We can stop execution (i) if all dependent values were positively tested or (ii) if at some point the next referenced value is greater than the current dependent value, because then we know that this dependent value is not included in the set of referenced values.

3.2. A Single-Pass Algorithm

Our single-pass algorithm minimizes the amount of I/O over the sets of attribute values. Each value is read only once and all INDs are tested in parallel.

```
Input: refValues, depValues: ordered sets of distinct
       attribute values
Output: Is dep ⊆ ref satisfied?
while depValues has next value do
    currentDep := depValues.next() ;
    if refValues is empty then
        return false;
    while true do
        currentRef := refValues.next() ;
        // test next item in depValues
        if currentDep = currentRef then
            break;
        // currentDep ∉ refValues
        else if currentDep < currentRef then
            return false;
        // test next item in refValues
        else if refValues has no next value then
            return false;
return true;
```

Algorithm 1: **Algorithm to test a single IND candidate.**

The key idea is as follows: All value sets are extracted from the database and stored in sorted files. We can use lexicographic sorting for all values including numeric values, because the actual order of values is irrelevant as long as it is consistent over all sets. In the brute-force approach, we enumerate all IND candidates and test them by iteratively skipping through two files. Now, we open all files at once and move a cursor through each file. Recall that we distinguish (potentially) dependent and (potentially) referenced attributes. For brevity, we call the corresponding files dependent files or referenced files, respectively.

Intuitively, we move a cursor r on a referenced file R one step further, i.e., read the next tuple, when *all* cursors to dependent files point to values that are greater than the current value pointed to by r in R. Conversely, we move a cursor d on a dependent file D one step further, when the value that d points at in D is smaller than *all* values currently pointed at in referenced files. While moving cursors, the algorithm keeps track of all IND candidates. Consider a pair D and R. Whenever the cursor on D is moved without haven seen in R the old value in D, the IND candidate $D \subseteq R$ is refuted and removed from the list of all candidates.

Our current implementation uses the subject-observer-model pattern from [7]. We view each attribute as a self-acting object that manages a list of its values and decides when it reads the next value in this list. There are two types of objects—objects representing referenced files (or referenced objects for short) and objects representing dependent files (or dependent objects). Each referenced object manages a list of all dependent objects with which the IND can-

didate was not yet refuted, as these have to be considered in future moves. Vice versa, each dependent object manages a list of all referenced objects with which an IND candidate could still be satisfied. As IND candidates are identified as unsatisfied, both sets are updated.

The challenge is to synchronize the read operations of all objects. We implemented it such that dependent objects take control: Thus, all IND candidates are verified in parallel by the dependent objects. A referenced object delivers its next value only when each of its dependent objects has issued a request for a move. A dependent object compares its current value with all current values of its referenced objects, decides when it needs the next value of a referenced object, decides when IND candidates are removed, and decides when it reads its own next value.

Each dependent object has three lists with referenced objects, called *currentWaiting*, *nextWaiting*, and *next*. These lists result from the idea to request a referenced object's next value as soon as possible, i. e., when a dependent object decides to compare (its current or next value) with a next referenced value. So *currentWaiting* contains all referenced objects whose next value still has to be compared with the current dependent value. *nextWaiting* and *next* contain all referenced objects whose next value has to be compared with the next dependent value. The difference between them is that *nextWaiting* contains all referenced objects that did not yet deliver their next value, and *next* contains all referenced objects that already delivered their next value—together with this value.

The algorithm of comparing the dependent value and a received referenced value is given in Algorithm 2. The decisions about which values should be compared next are essentially equal to the decisions in Algorithm 1, but here many more attributes have to be considered. This action is controlled using the lists *currentWaiting*, *next* and *nextWaiting*.

The procedure when a referenced value is delivered is given in Algorithm 3. This algorithm saves the given referenced value if it has to be compared with the next dependent value. Otherwise it compares first the current dependent value with the received referenced value and tests if all comparisons with the current dependent value have been done. In this case the next dependent value is read and possible comparisons (with values of referenced objects in list *next*) are performed. These comparisons fill the lists *currentWaiting* and *nextWaiting*, which are updated afterwards—if *currentWaiting* is empty—by fetching the next dependent value.

The overall process is controlled by a monitor, which collects requests and possible deliveries of all referenced objects and activates delivery using a first-in-first-out queue.

In the following we show that the single-pass algorithm indeed tests *all* IND candidates. A set of dependent objects

```
Input: referencedObject, referencedValue,
       dependentValue
Output: satisfied IND or null
if dependentValue = referencedValue then
    if ∃ next dependent value then
        if referencedObject.wantNextValue(this) then
            nextWaiting :=
                nextWaiting ∪ referencedObject ;
        else
            // exclude IND candidate
            referencedObject.detach(this) ;
    else
        // IND candidate satisfied
        referencedObject.detach(this) ;
        return this ⊆ referencedObject ;
else if dependentValue > referencedValue then
    if referencedObject.wantNextValue(this) then
        currentWaiting :=
            currentWaiting ∪ referencedObject ;
    else
        // exclude IND candidate
        referencedObject.detach(this) ;
// current dep. value ∉ values of referencedObject
else
    // exclude IND candidate
    referencedObject.detach(this) ;
return null ;
```

Algorithm 2: **Alg. processComparison; comparison of current dependent value and received referenced value for single-pass approach.**

influences when a referenced object's next value is read. Vice versa, a set of referenced objects influences when a dependent object's next value is read. Despite this mutual dependency, the algorithm does not run into deadlocks. The proof is based on the fact that we use *sorted* data sets.

Theorem 3.1 *The single-pass algorithm is deadlock-free and tests all IND candidates.*

Proof: The algorithm could only fail if it runs into a deadlock. Assume without loss of generality a deadlock between the dependent and referenced objects dep_1, dep_2, dep_3, and ref_1, ref_2, ref_3, i. e.,

- dep_1 waits for next value of ref_1
  $\rightarrow \mathrm{ref}_1 \in \mathrm{dep}_1.\mathrm{currentWaiting}$
  $\rightarrow \mathrm{ref}_1.\mathrm{currentValue} < \mathrm{dep}_1.\mathrm{currentValue}$
- ref_1 waits with delivering its next value until dep_2 requests it
  $\rightarrow \mathrm{ref}_1 \in \mathrm{dep}_2.\mathrm{next}$
  $\rightarrow \mathrm{ref}_1.\mathrm{currentValue} \geq \mathrm{dep}_2.\mathrm{currentValue}$
- dep_2 waits for next value of ref_2

```
Input: referencedObject, referencedValue
// compare with next dependent value
if referencedObject ∈ nextWaiting then
    nextWaiting := nextWaiting \ referencedObject ;
    next := next ∪
        {(referencedObject, referencedValue)} ;
    return ;
// compare with current dependent value
currentWaiting := currentWaiting \ referencedObject ;
processComparison(
    referencedObject, referencedValue) ;
// Do we need current value any longer?
if currentWaiting = ∅ and
    (next != ∅ or nextWaiting != ∅) then
    dependentValue := next dependent value ;
    // update waiting lists
    currentWaiting := nextWaiting ;
    nextWaiting := ∅ ;
    // test corresponding inclusion dependencies
    foreach otherReferencedObject in next do
        processComparison(
            otherReferencedObject,
            value of otherReferencedObject) ;
    next := ∅ ;
    // Do we need current value any longer?
    if currentWaiting = ∅ and nextWaiting != ∅ then
        dependentValue := next dependent value ;
        currentWaiting := nextWaiting ;
        nextWaiting := ∅ ;
```

Algorithm 3: **Perform an update in dependent object, i. e., procedure after delivery of a referenced value.**

$\rightarrow \mathrm{ref}_2 \in \mathrm{dep}_2.\mathrm{currentWaiting}$
$\rightarrow \mathrm{ref}_2.\mathrm{currentValue} < \mathrm{dep}_2.\mathrm{currentValue}$

- ref_2 waits with delivering its next value until dep_3 requests it
  $\rightarrow \mathrm{ref}_2 \in \mathrm{dep}_3.\mathrm{next}$
  $\rightarrow \mathrm{ref}_2.\mathrm{currentValue} \geq \mathrm{dep}_3.\mathrm{currentValue}$
- dep_3 waits for next value of ref_3
  $\rightarrow \mathrm{ref}_3 \in \mathrm{dep}_3.\mathrm{currentWaiting}$
  $\rightarrow \mathrm{ref}_3.\mathrm{currentValue} < \mathrm{dep}_3.\mathrm{currentValue}$
- ref_3 waits with delivering its next value until dep_1 requests it
  $\rightarrow \mathrm{ref}_3 \in \mathrm{dep}_1.\mathrm{next}$
  $\rightarrow \mathrm{ref}_3.\mathrm{currentValue} \geq \mathrm{dep}_1.\mathrm{currentValue}$

This means we have
$\mathrm{dep}_1.\mathrm{currentValue} \leq \mathrm{ref}_3.\mathrm{currentValue}$
$< \mathrm{dep}_3.\mathrm{currentValue} \leq \mathrm{ref}_2.\mathrm{currentValue}$
$< \mathrm{dep}_2.\mathrm{currentValue} \leq \mathrm{ref}_1.\mathrm{currentValue}$
$< \mathrm{dep}_1.\mathrm{currentValue}$, which is a contradiction. □

3.3. Experimental Results

We compared the SQL approach against both order-approaches. All tests were performed on the three databases described in Sec. 1.4. The algorithms were implemented in Java 1.5. The running times are given in Tab. 2, together with the fastest SQL approach (utilizing `join`). Note that the time measurements summarize all costs—inclusively shipping the data outside the database. We run the tests on two different fractions of the PDB, because of scaling issues related to the single pass algorithm (number of dependent and referenced attributes) that are addressed in Sec. 4. The first fraction (with 139, 356 IND candidates) covers 2560 attributes in 167 tables and 2.7 GB and the second fraction (with 4, 268 IND candidates) covers 541 attributes in 39 tables over 2.6 GB.

	UniProt	SCOP	PDB	
# IND cand.	910	43	139, 356	18, 230
# sat. INDs	36	11	30, 753	4, 268
DB size	667 MB	17 MB	2.7 GB	2.6 GB
`join`	15 m 03 s	7.3 s	> 7 days	-
`brute-force`	2 m 38 s	10.7 s	3 h 13 m	1 h 29 m
`single-pass`	3 m 08 s	13.0 s	see Sec. 4	3 h 06 m

Table 2. Experimental results of approaches using order on data compared to the SQL approach. Only fractions of the PDB were used.

Clearly, the algorithms using order on data are significantly faster than the `join` approach for large databases, especially for the PDB. Using the brute force algorithm, we could extend feasible databases sizes further and compute all INDs on a 3.2 GB fraction of the PDB, which lasted 3 h 44 min and identified 33, 989 INDs.

Surprisingly, the brute force approach is faster than the single-pass approach. We currently believe that the reason is the synchronization overhead of our particular implementation of the single-pass approach using a strictly object-oriented approach. To compare I/O efficiency of both algorithms we counted the total number of tuples read using growing subsets of attribute sets from the UniProt database. The results are given in Figure 5 and clearly show that the single-pass algorithm is much more I/O efficient than the brute-force algorithm. Note that also the amount of I/O of the brute-force algorithm seems to grow only linearly with the number of attributes, although the number of IND candidates grows quadratic. This situation can be explained by the fact that most candidates are refuted after reading only few tuples, e.g., when one attribute contains strings (names, sequences, descriptions, ...) and the other numbers (surrogate keys, numeric values, ...).

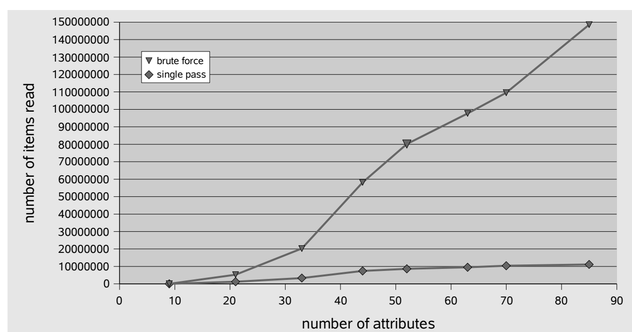

Figure 5. I/O comparison for brute force and single pass approach.

4. Scaling Up

In this section we describe scalability issues of the different approaches. First, we regard heuristics to reduce the number of IND candidates that have to be explicitly tested. These apply to all described approaches. Second, we describe scalability issues at system level for the brute force and the single pass approaches.

4.1. Pruning IND candidates

An important potential improvement is to reduce the number of IND candidates before actually testing their satisfiedness. Such improvements apply to all described approaches, but the performance differences—caused by the cost for a single IND test—remain.

We observed that in many cases, after sorting attribute value sets, it is beneficial to first compare the respective maximum values of the two sets. If the maximum of the (potentially) dependent set is larger than the maximum of the (potentially) referenced set, we can stop the test immediately. We implemented this improvement in all described approaches. For UniProt the reduction was from 910 to 541 candidates. There was a benefit of $15\,\%$ for the `join` approach, $39\,\%$ using `minus`, and $14\,\%$ using `not in`. The brute force and single pass approach run about $20\,\%$ faster. Thus, the approaches using order still outperform the SQL approaches. For PDB we cannot compare the SQL approaches as they did not terminate within reasonable time. We ran the brute force and single pass approach on the $2.6\,$GB fraction of the PDB. The number of IND candidates decreased from $18,230$ to $7,354$ and both implementations ran about $40\,\%$ faster. There is no benefit for SCOP concerning all approaches, as we believe due to the small size of SCOP.

Furthermore, there is a possibility to use the transitivity property of inclusion dependencies to reduce the number of IND candidates, as described in [2]. There, IND candidates are excluded using already identified (satisfied and unsatisfied) INDs. Another idea is to pretest the IND candidates using random samples of the dependent data. We believe that this should exclude a large number of IND candidates. We leave this point as further work.

Note that using data types as a heuristic to prune IND candidates is not applicable in the life science domain, because often even attributes containing solely integers are represented as string.

4.2. Scalability at System Level

For scaling the approaches at system level we must consider two facts: main memory consumption and number of open files.

The brute force algorithm opens one referenced file and one dependent file to test an induced IND candidate. Furthermore, it needs one dependent and one referenced value at a time to decide the satisfiedness of the IND candidates. That way the brute force algorithm scales up to test IND candidates in very large databases.

The single-pass algorithm on the other hand opens all referenced and dependent files in parallel. It needs one value of each dependent attribute and at most two values of each referenced attribute (one actually delivered and one temporary stored in dependent attributes, waiting until it can be used). Thus, the memory consumption is not the limiting factor for the single-pass algorithm, but the number of open files. That is the reason why we could not compute the satisfied INDs of the PDB fraction covering $2.7\,$GB; we had to open 2560 files, which is not feasible for our system. To scale the single-pass algorithm to such numbers of dependent and referenced attributes we must implement a blockwise approach—comparing blocks of dependent attributes against (all or blocks of) referenced attributes. As our current implementation of the single-pass algorithm is slower than the brute force implementation we will leave the blockwise implementation as further work—following the speed up of the single-pass implementation.

5. Schema Discovery using INDs

The final goal of computing all INDs in the Aladin project is schema discovery. Therefore, we analyzed to what extent the discovered INDs are in fact foreign key constraints. Furthermore, the goal of the project is also to find links between different databases. In our domain, databases typically contain one major class of data (gene, protein, sequence, etc.) with structured annotations. The identifier of the major objects are called accession numbers, and these are used as link targets when databases refer to each other. To find such links, one must first identify the primary relation of a database, i.e., the relation containing the accession number of the primary objects. We also verified to what extent our computed INDs are useful for this task.

We verified the utility of INDs for finding foreign keys using UniProt and PDB. The BioSQL schema, which we used for UniProt as explained in Sec. 1.4, defines foreign key constraints, which we use as gold standard. The results are promising: Our algorithm found *all* defined foreign keys as INDs, with the exception of two foreign keys that are defined on empty tables and obviously cannot be found when regarding the data. Additionally, we found 11 INDs that are in the transitive closure of the foreign key definitions, i. e., if there are foreign keys $A \subseteq B$ and $B \subseteq C$ we find the satisfied INDs $A \subseteq B$, $B \subseteq C$, and $A \subseteq C$. Finally, no false positives were produced.

The OpenMMS schema—into which we imported the PDB data—does not define any foreign keys. On the one hand this is a good example for the necessity of identifying foreign keys, it is on the other hand difficult to verify the identified satisfied INDs. As the OpenMMS schema is very large, we could not perform a systematic test. However, we observed that the OpenMMS schema often utilizes surrogate IDs, i.e., semantic-free integers whose ranges all begin at 1, as primary keys. This is a case where INDs fail to identify foreign keys. There are INDs between almost all of these ID attributes, leading to the observed $30,000$ satisfied INDs. In future work we will look into heuristics for removing such false positives. One idea is to analyze the ranges of attributes.

For identifying the primary relation of a database, we use the following heuristics:

1. One of the attributes of a primary relation must be an accession number candidate, which is a domain specific criterion and means that all values of this attribute are at least four characters long, contain at least one character, and must not differ in length more than $20\,\%$.
2. The number of INDs referencing any attribute in a relation containing an accession number candidate is maximal for the primary relation.

Applying these heuristics to BioSQL we identified three accession number candidates (sg_bioentry.accession, sg_reference.crc and sg_ontology.name). Out of them, Heuristic 2 identifies unambiguously the correct primary relation, namely sg_bioentry.

For the OpenMMS schema we find nine accession number candidates, and 19 accession number candidates when softening the rules such that only $99.98\,\%$ of a columns values must fulfill the first criteria. Heuristic 2 leads to three primary relation candidates (exptl, struct, struct_keywords). Of these, struct is the correct solution, whereas struct_keywords could be considered as a second primary relation, as it is a table containing controlled vocabulary. Even though this selection is not perfect, it is a very effective pre-selection (three tables out of 115 tables), which helps a human expert to manually choose the primary relation. Furthermore, we believe that a filter on the satisfied INDs—concerning the described problem using natural numbers as primary key—will yield a clearer decision for the primary relation.

6. Related Work

Bell and Brockhausen propose to create all unary IND candidates and test them sequentially by utilizing an SQL `join` statement [2]. The tested (satisfied and not satisfied) INDs are used to exclude further tests and therefore to reduce the number of IND candidates to test. Furthermore, the number of IND candidates is reduced by constraints on the datatypes and maximal and minimal values. The `join` statement performs a join on the attributes A and B of the IND candidate and compares the number of returned tuples to the number of distinct values in A and B therefore verifying $A \subseteq B$ and $B \subseteq A$. We use a similar `join` statement in our join approach in Sec. 2. The further criteria to exclude IND candidates could be used for any of our algorithms. Thus, we expect that the difference in performance will remain for larger schemas.

De Marchi et al. propose another way to identify unary INDs [10]. They use a preprocessing on the data to create a table for each datatype with tuples for each value contained in the database and all attributes which contain this value. After this they test all IND candidates using this tables by iterating over all values and excluding IND candidates, which are violated by the current value and its containing attributes. A major drawback of this method is its huge preprocessing requirement. Furthermore, the authors describe a levelwise approach to deduce multivalued IND candidates, which is improved in [11]. The idea is to combine the levelwise approach of [10] with an approach for higher-level INDs reducing the number of IND candidates by switching between a top-down and a bottom-up approach using a optimistic positive border.

Koeller and Rundensteiner identify INDs in relations with up to 100 attributes [8]. They utilize an exhaustive approach for identifying unary and binary INDs similar to [2] and identify multivalued IND candidates by finding cliques in k-uniform hypergraphs created of lowervalued satisfied INDs. We believe that our algorithms for finding unary INDs more efficiently than with pure SQL will also be beneficial for finding multivalued INDs.

Dasu et al. use data summaries to approximately identify join path, i. e., to identify approximately inclusion dependencies [5]. They use set resemblance and multiset resemblance to identify the join path and its size and direction. Although we want to compute exact satisfied INDs, we could use this procedure to reduce the number of IND candidates.

7. Conclusion

We described in this paper five approaches to test IND candidates—three of them using variants of SQL `joins` on the data stored in an RDBMS and two database external approaches that use linear scans through sorted sets.

The SQL approaches lack the ability to express the problem efficiently. All three approaches provide correct results, but the database computes more than necessary. Although all operations on the data occur inside the database engine, i. e., no data is shipped out of the database, all SQL approaches are significantly slower than the database-external approaches.

The database-external approaches—using ordered, distinct sets of attribute values delivered from the RDBMS—preclude an IND candidate at the first dependent value that is not included in the set of referenced values. We described a brute force approach (testing all IND candidates sequentially) and a single-pass approach (testing all IND candidates in parallel). Although the latter is much more I/O efficient, as we also showed by experiments, our current brute force implementation is faster. We believe that the reason for this behavior is the synchronization overhead partly incurred by our object-oriented implementation of the single-pass algorithm. Thus, in our current work we concentrate on improving the performance of the single-pass algorithm. Furthermore, we showed how a simple heuristic already greatly reduces the number of IND candidates that need to be tested. We will, in future work, research more such heuristics, for instance using random sampling. Also, we will consider using the transitivity property of inclusion dependencies to further reduce the number of comparisons.

We evaluated the identified INDs in our application scenario of discovering foreign keys in the life sciences and for identifying a domain-specific primary relation that describes the main objects in the database. Our current heuristics produced very good results in one database, i.e., UniProt, but need further improvements for another database, i.e., PDB. For the PDB, the identified INDs contained many false positive foreign keys but where still very helpful for identifying the primary relation. Clearly, we need to develop more (probably domain-specific) filters to reduce the number of false positives.

In future work we plan use this procedure to identify inclusion dependencies between attributes of different databases to identify object links and between concatenated values, e. g., attributes containing PDB codes as "144f" or as "PDB-144f". Furthermore we plan to extend our procedure to identify partial INDs on dirty data.

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). We thank Véronique Tietz for her extensive work in data preparation.

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 Research*, 33(Database issue):D154–9, 2005.

[2] 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.

[3] 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.

[4] A. Bilke and F. Naumann. Schema matching using duplicates. In *Int. Conf. on Data Engineering (ICDE)*, 2005.

[5] T. Dasu, T. Johnson, S. Muthukrishnan, and V. Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 240–251, 2002.

[6] A. Doan, P. Domingos, and A. Y. Halevy. Reconciling schemas of disparate data sources: a machine-learning approach. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 509–520. ACM Press, 2001.

[7] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] A. Koeller and E. A. Rundensteiner. Discovery of high-dimensional inclusion dependencies. In *Int. Conf. on Data Engineering (ICDE)*, pages 683–685, 2003.

[9] U. Leser and F. Naumann. (Almost) hands-off information integration for the life sciences. In *Conf. on Innovative Database Research (CIDR)*, 2005.

[10] F. D. Marchi, S. Lopes, and J.-M. Petit. Efficient algorithms for mining inclusion dependencies. In *Int. Conf. on Extending Database Technology (EDBT)*, pages 464–476. Springer-Verlag, 2002.

[11] F. D. Marchi and J.-M. Petit. Zigzag: a new algorithm for mining large inclusion dependencies in databases. In *IEEE Int. Conf. on Data Mining (ICDM)*, pages 27–34, 2003.

[12] R. J. Miller, M. A. Hernandez, L. M. Haas, L. Yan, H. Ho, R. Fagin, and L. Popa. The clio project: Managing heterogeneity. *SIGMOD Record*, 30(1):78–83, 2001.

[13] G. Muhammad. Performanzvergleich von Genexpressionsdatenbanken. Master's thesis, Humboldt-Universität zu Berlin, 2004.

[14] A. G. Murzin, S. E. Brenner, T. Hubbard, and C. Chothia. Scop: a structural classification of proteins database for the investigation of sequences and structures. *J Mol Biol*, 247(4):536–40, 1995.

[15] R. A. Pottinger and P. A. Bernstein. Merging models based on given correspondences. In *Int. Conf. on Very Large Data Bases (VLDB)*, pages 862–873, 2003.

[16] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.