Efficiently Identifying Inclusion Dependencies in RDBMS

Jana Bauckmann

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

Abstract

In large integration projects one is often confronted with poorly documented databases. One possibility to gather information on database schemas is to search for inclusion dependencies (IND). These provide a solid basis for deducing foreign key constraints—as they are pre-condition for potential (semantically valid but missing) foreign key constraints.

In this paper we present and compare several algorithms to identify unary INDs. The obvious way is to utilize an appropriate SQL statement on each potential IND to test its satisfiedness. We show that this approach is not efficient enough for large databases. Therefore, we developed database-external approaches that are up to several magnitudes faster than a SQL based approach. We tested our algorithms on databases of up to 3 GB with about 1200 attributes, which can be analyzed by our software in approximately 25 minutes.

1 Motivation

In large integration projects one often copes with undocumented databases. One important information about their semantic structure can be provided by foreign keys. Often these definitions are incomplete, or simply not existing.

Especially in life science databases this is a huge problem. One example is the Protein Data Bank[1] (PDB), which can be imported into a relational database system using the OpenMMS schema[2]: This schema defines 175 tables over 2705 attributes. It has primary key constraints—but not a single foreign key constraint.

We want to identify inclusion dependencies (IND) as a pre-condition for foreign keys. In this paper we describe algorithms for identifying unary INDs $A \subseteq B$, meaning all values of attribute A are included in the bag of values of attribute B.

We use the term IND candidate for pairs of attributes previous to any tests. If an IND candidate fulfills the IND definition we call it a satisfied IND, otherwise an unsatisfied IND. The left hand side of an IND $A \subseteq B$ is called dependent attribute, the right hand side referenced attribute.

Another property of life science databases is that one cannot trust the defined data types. For that reason we have to test all pairs of attributes as IND candidates, i. e., $O(n^2)$ tests if n is the number of attributes. The number of IND candidates can be reduced by two simple restrictions: The referenced attribute must be unique and the number of distinct dependent values must be smaller or equal to the number of distinct referenced values. But both restrictions do not decrease the problem complexity.

In our project Aladin we aim at **Al**most h**a**nds-off **d**ata **in**tegration of life science databases [3]. In a first step, data sources are imported into a relational database system. This step requires manual support to transfer the data from whatever format into a relational schema. In a second and third step we want to identify intra-schema relationships, such as primary and foreign

[1]http://www.rcsb.org
[2]http://openmms.sdsc.edu

keys. Furthermore we want to apply domain specific knowledge: Life science databases provide usually information on one concept, e. g., proteines or genes. That is why the general structure is based on one "primary relation" representing the objects generally and several "secondary relations" providing additional information on these objects. The fourth step intends to identify intra-source relationships between attributes or even objects of different databases. The fifth step will be to find duplicate objects. Identified INDs help us in the third step to derive foreign keys, which help to identify the primary relation, and to derive intra-source relationships in the fourth step.

The efficiency of identifying INDs is crucial especially in the development phase of Aladin, where several heuristics have to be applied and tested for their usefulness in the given context. We tested a simple approach utilizing a `join` statement for each IND candidate on a 2.7 GB fraction of the PDB consisting of 109 tables with 1576 attributes. This test did not finish within seven days—which considerably impeded further development.

In this paper we present new approaches for testing IND candidates together with experimental results. In Section 2 we describe sql solutions, and in Section 3 two high-level algorithms. Section 4 concludes and gives an outlook on future work. This paper is a summary of our previous work [1] extended by an improved high-level algorithm.

2 Approaches using SQL

The most obvious way to identify INDs in a RDBMS is to utilize an appropriate sql statement leveraging all optimizations implemented in the database system. In the following we provide three sql statements—all computing the correct results—together with their experimental results.

2.1 Utilizing SQL Statements

The first statement utilizes a `join` statement (see Fig. 1). The idea is to join the attributes and compare the number of joined tuples with the number of non-null tuples in the dependent attribute. The IND candidate is satisfied iff both numbers are equal.

When looking closer at this statement one realizes that the statement actually computes too much. The only necessary information is, is there one dependent value that is not included in the referenced attribute's values. Therefore, we formulated two further statements: The result of the statement should only return tuples iff the IND candidate is not satisfied. Thus, we may stop the computation after the first tuple in the result set using a TOP K sql construct.

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

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

Figure 1: Statement utilizing `join`.

The `minus` statement given in Fig. 2 (a) substracts all referenced values from all dependent values. Note that we had to use vendor specific constructs to stop the execution after the first tuple in the result set. The idea of the `not in` statement in Fig. 2 (b) is to ask for all dependent values that are not contained in the referenced attributes values.

```
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)
```

(a) Utilizing `minus`.

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

(b) Utilizing `not in`.

Figure 2: Statements intended to stop after first unmatched dependent item. Therefore, for both statements holds: IND candidate is satisfied $\Leftrightarrow$ $|\text{unmatchedDeps}| = 0$

2.2 Experiments

We tested all three statements on three life science databases: SCOP[3] provides classified proteins. It is available in four files containing table structured data, which we imported into relations. UniProt[4] is a database of annotated protein sequences. We used the BioSQL[5] schema and parser to import the data. The PDB is a large database of protein structures, which we imported using the OpenMMS schema and parser (see Sec. 1 Motivation).

We run all tests on a Linux machine with 2 processors and 12 GB RAM. The results are given in Table 1. The `join` statement is the fastest alternative, which is surprising because it does not perform the early stop of computation. We believe the reason lies in the extensive optimization of `join` operations in RDBMS. Furthermore, the implementation of the TOP K constructs seems not to be merged with the inner query during query rewriting.

Despite this, all three statements where not applicable to test all IND candidates on a database of the size of PDB: The `join` approach did not finish within seven days. The reason for this runtime performance is twofold. First, all IND candidates are tested sequentially and independently, such that necessary sort operations have to be computed several times. Second, we cannot describe our problem exactly—all statements present just workarounds that cause the database to compute too much.

	SCOP	UniProt	fraction of PDB
# tables	4	16	109
# attributes	22	85	1576
DB size	17 MB	667 MB	2.7 GB
`join`	7.3 s	15 min 03 s	> 7 days
`minus`	14.3 s	29 min 16 s	-
`not in`	46 min	1 h 53 min	-

Table 1: Experimental results utilizing SQL.

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

3 Approaches Using Order On Data

When looking at the problem at a higher level one realizes a rather simple approach: First, sort all distinct values of each attribute using an arbitrary but fixed order. Second, scan linearly through the ordered value sets of an IND candidate starting from smallest one while comparing the values. Let dep be the current dependent value and ref be the current referenced value. We have to distinguish three cases. (i) If dep = ref then move the scan pointer in both sets one position further, because we found the current dependent value in the set of referenced values. (ii) Otherwise, if dep > ref we can move the referenced pointer one position further—that way, we look for the current dependent value in the following referenced values. (iii) In the third case, dep < ref we can stop the computation, because the current dependent value is surely not included in the set of referenced values. The IND is unsatisfied. To identify a satisfied IND we have to scan all dependent values and find an equal referenced value.

To implement this approach we use the database to sort the data. Afterwards we ship the data out of the database and write it to disk. The second step utilizes a database external java program. It is not obvious if this approach outperforms the sql approaches, because of the trade-off between the early stop of computation and shipping the data out of the database.

3.1 Brute Force Approach

A Brute Force Approach following the above algorithm tests all IND candidates sequentially. The advantage over the sql approaches is first the early stop of computation if the IND candidate is unsatisfied. This is a severe argument, because most IND candidates are unsatisfied and therefore their tests stop after few comparisons. The second advantage is that the values of each attribute are sorted only once.

A major disadvantage is the need to read several times over each attributes sorted values—once for each occurence in an IND candidate.

3.2 Single Pass++

The Single Pass++ approach uses the advantages of the Brute Force Approach and eliminates disadvantage of reading values multiple times. Therefore, all IND candidates are tested in parallel. The challenge is to decide when the pointer for each attribute can be moved: A set of current dependent values influences when a current referenced pointer is moved on. Vice versa, a set of current referenced values influences when a current dependent pointer can be moved. Despite this mutual dependency, it is possible to synchronize the pointer movements without running into deadlocks or missing some IND candidate tests. This is founded on the fact that we use *sorted* data sets.

We represent each attribute's sorted values as an *attribute object* providing the values and a pointer at the current value. Furthermore, it provides two roles—as dependent attribute and as referenced attribute. The IND candidates are represented and handled by the role as dependent attribute. Therefore, each attribute object stores, which referenced attributes form IND candidates with this attribute object as dependent attribute. They are devided into two lists depending on the fact if this current dependent value was already found in the referenced attribute's values (satisfiedRefs) or not (unsatisfiedRefs).

Given these attribute objects we can apply the following algorithm. Hold all attribute objects in a sorted min-heap depending on their current value. 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. Then inform each dependent attribute object in Min of each referenced attribute object in Min. This way, the dependent attribute objects can track, which referenced attribute includes its current value and which not.

After this, test for all attribute objects in Min if there is a next value. If there is no next value, output all INDs build of the attribute object as dependent attribute and all referenced attribute objects in satisfiedRefs as referenced object. Otherwise, if there is such a next value, then read it 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 current dependent value in them.

3.3 Experiments

We tested both approaches with the same test sets as the sql approaches on the same machine. The results are given in Table 2 in comparison to the join approach results as it is the fastest sql approach.

	SCOP	UniProt	fraction of PDB
# tables	4	16	109
# attributes	22	85	1576
DB size	17 MB	667 MB	2.7 GB
`join`	7.3 s	15 m 03 s	> 7 days
`Brute Force`	10.7 s	2 m 38 s	3 h 13 m
`Single Pass++`	10.2 s	2 m 14 s	22 m 30 s

Table 2: Experimental results of approaches using order on data compared to the fastest SQL approach (utilizing `join`).

We are able to test all IND candidates on the used fraction of the PDB with both approaches. Thus, we achived a performance improvement of several magnitudes. In fact, the 22 minutes for testing all IND candidates on the PDB fraction contain 18 minutes to sort the data, ship them out of the database and store them to disk.

4 Conclusion and Outlook

We showed two general approaches for testing IND candidates. The database external approaches based on the order of data outperform the presented sql approaches. This is caused by the fact that the problem of identifying INDs cannot be expressed exactly and therefore not efficiently using sql. We implemented the approaches given by Bell and Brockhausen [2] and De Marchi et al. [4] as both identify unary INDs exactly. Our high-level approaches outperform both.

In future work we want to look at partial INDs for use on dirty data, multi-valued INDs and complex INDs with concatenated attributes or value substrings.

References

[1] Bauckmann, J., Leser, U. und Naumann, F. (2006): Efficiently Computing Inclusion Dependencies for Schema Discovery. In *Workshop Proceedings of the ICDE 06: 2nd Int. Workshop on Database Interoperability.*

[2] Bell, S. und Brockhausen, P. (1995): Discovery of Data Dependencies in Relational Databases. In Y. Kodratoff, G. Nakhaeizadeh und C. Taylor (Herausgeber), *Statistics, Machine Learning and Knowledge Discovery in Databases, ML–Net Familiarization Workshop*, Seiten 53–58.

[3] Leser, U. und Naumann, F. (2005): (Almost) Hands-Off Information Integration for the Life Sciences. In *Conf. on Innovative Database Research (CIDR)*, Seiten 131–143.

[4] Marchi, F. D., Lopes, S. und Petit, J.-M. (2002): Efficient Algorithms for Mining Inclusion Dependencies. In *Int. Conf. on Extending Database Technology (EDBT)*, Seiten 464–476. Springer-Verlag.