Automatically Integrating Life Science Data Sources

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

ABSTRACT

Data integration in the life sciences is currently implemented by very costly manually curated projects or by schema-driven approaches that require knowledge of the data sources and knowledge about schema integration techniques. We propose ALADIN – an entirely different, almost automatic approach based on the database instances and domain knowledge on life science data sources.

We focus on two main tasks in ALADIN – detecting intra-schema relationships and detecting inter-schema relationships. For the first task, we present our algorithm SPIDER for detecting inclusion dependencies (INDs) as a precondition for foreign keys. SPIDER analyzes a 2.8 GB database in $\sim$ 24 min and a 32 GB database in $\sim$ 6 h, up to an order of magnitude faster than previous approaches. We use INDs in two ways: (i) as hint for semantically correct foreign keys and (ii) for identification of a "primary relation" – a domain-specific schema construct intended to help in further integration. For the second task, we describe ideas and first results on detecting cross-references between life science data sources and on detecting duplicate objects.

1. PROJECT IDEA

The project ALADIN – ALmost Automatic Data INtegration – aims at integrating life science data sources almost automatically. The most special characteristic is the complete renunciation of human interaction. We want to leverage domain specific characteristics of data source structures and data itself. This procedure aims at providing an alternative to manually curated integration projects that are costly but accepted by biologist and to schema-driven integration projects that require knowledge on schema matching and schema mapping [17].

On the one hand, life science data sources deliver an excellent application area for data integration. First, there is a large amount of life science data sources and even domain experts know – and use – only a part of them. Automatic data integration provides the ability to use information given in data sources with unknown structure and interfaces. Second, the usage of information spread over several life science sources is dominated by manual interaction: Searching one data source, finding a keyword or a cross-reference to another data source, and finally searching the other data source with this information. Thus, automatic integration would provide a better accessibility, usage, and access to information over several known data sources. Third, life science data sources are heavily overlapping. Thus, there is a real need to integrate them. Further, life science data sources cross-reference each other heavily [10]. Finding those references could enable automatic data integration.

On the other hand there are several domain-specific problems in this application area. Schemas of life science data sources are most often modeled insufficiently. Data are provided in structured flat files or in relational schemas. Data types are often defined as string although the attribute represents numeric data. Referential integrity constraints are often undefined, because they are believed to badly influence database operation and because until recently they were not provided by MySQL, which is widely used in life science projects. Thus, even extremely successful systems, such as the Ensembl database, with hundreds of tables, is delivered without foreign key definitions [11].

Further, the terminology is different over several data sources. There are huge problems related to synonyms and homonyms in schema names. That is why we decide not to rely upon schema information for data integration, but the data itself.

There are several domain-specific characteristics on life science data sources that we want to leverage for integration. A data source typically covers information on a single type of real world objects, e. g., proteins, or genes, or DNA sequences. These objects are represented in one relation – called *primary relation*. All other relations provide additional information on these objects. We call those relations *secondary relations*.

Life science data sources often cross-reference each other using so called *accession numbers*. An accession number is a key that uniquely represents an object of the primary relation (e. g., a protein) over different data source versions and over different data sources. The structure of an accession number has typical characteristics so that we can automatically find it.

*Supervised by Felix Naumann from the Hasso-Plattner-Institut, University of Potsdam and by Ulf Leser from the Department for Computer Science, Humboldt-Universität zu Berlin.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '07, September 23-28, 2007, Vienna, Austria.
Copyright 2007 VLDB Endowment, ACM 978-1-59593-649-3/07/09.

1.1 Steps in the ALADIN project

We divide the integration procedure of ALADIN into four steps. At first, the data source must be imported into a relational database. For many life science data sources a parser exists to convert the data into a relational form – with all the problems on missing constraints as mentioned above. To be able to process as many life science data sources as possible we decided to make no assumptions on schema structure or data. This step is the only task requiring human action.

The second step detects *intra-schema relationships*. We detect foreign keys or more exactly we detect inclusion dependencies, i. e., pairs of attributes A, B with all values of A being included in all values of B. This definition relates to the automatically testable part of a foreign key. Further, we use this knowledge of structural information to identify the primary relation.

The third step is to find *inter-schema relationships*. Data lineage is very important in life science projects. Thus, we do not aim to define an integrated schema, but to define schema and object links, which can be browsed by domain experts. Therefore, we want to find cross-references between the data sources – at schema and at object level. The fourth and last step aims at duplicate detection, i. e., detecting representations of the same real world object in several data sources – probably using different or even conflicting data. Using the information on cross-references we want to identify duplicates at object level.

1.2 Structure of this paper

In this paper we show our results on detecting intra-schema relationships and line out ideas on detecting inter-schema relationships and duplicates. Section 2 gives an overview on data sources used so far in the project and on further data sources we plan to use for future work. In Section 3, we give an algorithm to detect inclusion dependencies efficiently and use its results to identify the primary relation. In Section 4 we show our ideas on detecting inter-schema relationships and in Sec. 5 our plans to detect duplicate objects. We conclude in Section 6.

2. IMPORTING DATA SOURCES

The first step of our work is to import data sources in a relational database. At the current state we have imported the following three data sources on proteins, which we use to test our ideas on detecting intra-schema relationships.

UniProt[1] is a database of annotated protein sequences available in several formats [2]. We chose the BioSQL[2] schema and parser, creating a database of 16 tables with 85 attributes. The total size of the database is 900 MB, with the largest attribute having approximately 1 million different values.

SCOP[3] is a database of protein classification available as a set of files [20]. 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 [6]. We used the OpenMMS software for parsing PDB files into a relational database. PDB populates 116 tables with 1,297 non-empty attributes in the OpenMMS schema. No foreign keys are specified. The total database size is 32 GB, with the largest attribute having approximately 152 million different values.[4]

For future work on detecting intra-schema relationships we plan to import data sources on protein-protein-interaction such as IntAct[5] or Reactome[6]. We know that these data sources (i) reference each other and (ii) reference proteins in the already imported data sources.

3. DETECTING INTRA-SCHEMA RELATIONSHIPS

The first automatic step is the detection of intra-schema relationships, i. e., finding foreign keys, identifying the primary relation and the accession number attribute. This structural information is necessary for understanding the data and for meaningful querying. Further, we need *intra*-schema relationships for detecting *inter*-schema relationships, i. e., the cross-references from one data source to another using accession numbers.

3.1 Inclusion Dependencies

The automatically testable part of a foreign key definition is an inclusion dependency (IND). An IND requires that the set of values of attribute A is included in the set of values of attribute B, i. e., $A \subseteq B$. We call each attribute pair A, B an IND candidate.

3.1.1 Limitations of Previous Approaches

Detecting INDs efficiently is not as easy as it seems, because $O(n^2)$ attribute pairs must be tested with n being the number of attributes. Previous approaches test IND candidates by using SQL statements (using one join per IND candidate). But all approaches reduce the number of candidates by constraining pairs to equal data types [5, 15] or by using samples to create IND candidates [7]. Dasu et al. [8] reduce the test complexity of each IND candidate test by using data summaries.

We decided not to use samples or data summaries, because we want to avoid errors in this step of the ALADIN framework. Errors at this level would surely propagate through the entire integration process. Thus, working as exact as possible is a basic requirement for detecting INDs in ALADIN.

Further, attributes in the life science domain are commonly defined as data type string. This means that we have to test all $O(n^2)$ pairs of attributes. We tested several SQL statements to perform this task. We used one statement per IND candidate, but found that this approach is infeasible for large schemas such as PDB. The reason is two-fold: First, the independent test of each IND candidate by one query 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,

[1] www.pir.uniprot.org
[2] obda.open-bio.org
[3] http://scop.mrc-lmb.cam.ac.uk/scop
[4] Database sizes differ slightly from previous work due to new versions of the DBMS and OpenMMS parser.
[5] www.ebi.ac.uk/intact/
[6] www.reactome.org

the join variant essentially computes the number of counter-examples [3].

Marchi et al. [19] propose a completely different approach requiring a preprocessing on all data. This preprocessing assigns to each value in the database a list of attributes that include this value. The test itself uses one pass on this structure. We tested this approach and showed its infeasibility to large schemas such as PDB [4].

3.1.2 Detecting INDs with SPIDER

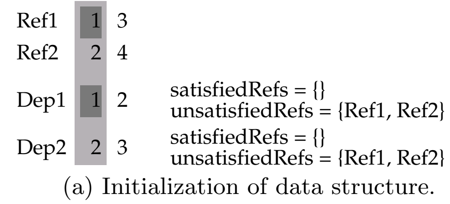
The evaluation of previous approaches led us to developing the SPIDER algorithm [4]. It is capable to exactly detect all INDs in databases within large schemas. SPIDER is based on two ideas: (i) Stop the test of a single IND candidate as soon as the first counter-example is known. A sort-merge-like procedure is used to find this first dependent value not included in the referenced attribute's values. (ii) All IND candidates are tested in parallel while reading all attribute's values at most once. Both characteristics and a special data-structure to synchronize all IND tests build the main advantage of SPIDER: A greatly reduced complexity in terms of comparisons that enables IND tests on huge schemas. This feasibility led to a co-operation with FUZZY! Informatik AG, a German company providing the data profiling tool FUZZY! DIME. The next release of this tool will implement SPIDER.

We describe SPIDER in the following in detail, because some of our ideas for detecting inter-schema relationships are based on SPIDER or on modifications of SPIDER. In a preprocessing step all attribute's value sets are sorted inside the DBMS and these distinct sorted values are saved to disk.

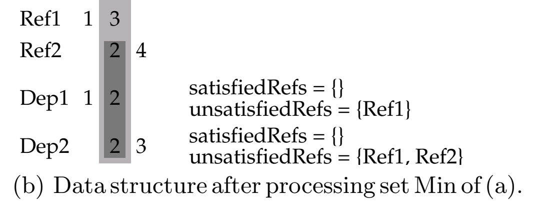
The basic procedure for testing *one* IND candidate $A \subseteq B$ is described in the following. Handle cursors on the sorted value sets of both attributes starting from the smallest value. We call A the dependent attribute and B the referenced attribute. Let dep and ref be the current value in the dependent and referenced attribute, respectively. Compare dep and ref and move the cursors as follows: (i) If dep $>$ ref move the referenced attribute's cursor one position further, i. e., look for the current dependent value in the remaining referenced values. (ii) If dep $=$ ref move both cursors on, because we found dep in the referenced attribute and want to test the next dependent value. (iii) Otherwise, if dep $<$ ref stop the execution, because we found the counter-example dep being not included in the set of values of B. An IND candidate is tested to be satisfying the IND definition if all values of the dependent attribute are tested and found.

SPIDER performs the described test on all IND candidates in parallel while saving comparisons. The used data structure and one step of the SPIDER test are illustrated in Figure 1. It shows two referenced attributes Ref1, Ref2 and two dependent attributes Dep1, Dep2.

All attributes are represented as attribute object with a cursor pointing at its current value and two roles: as dependent object and as referenced object. In Figure 1 we show attribute objects with only dependent or only referenced role for better traceability. The IND candidates are represented and maintained in the dependent object role using two sets of referenced attributes – namely satisfiedRefs and unsatisfiedRefs. The referenced attributes in both lists form IND candidates with the dependent attribute that are to be tested, i. e., in Fig. 1 a the IND candidates Dep1 $\subseteq$ Ref1 and Dep1 $\subseteq$ Ref2 are represented in attribute object



(a) Initialization of data structure.



(b) Data structure after processing set Min of (a).

Figure 1: Illustration of Algorithm SPIDER. Attribute objects with their values and additional helper structures, are given horizontally. Light gray areas mark the current values, dark gray areas mark minimal but equal values.

Dep1. For referenced objects in satisfiedRefs we know that the current dependent value is included in this referenced attribute; for referenced objects in unsatisfiedRefs we have not (yet) seen the current dependent value in this referenced attribute.

To synchronize the cursor movements we hold all attribute objects in a min-heap sorted by their current value. A min-heap is a data structure that provides peeking its smallest item in constant time, and inserting any item as well as deleting the minimal item in $O(\log n)$.

The following procedure is repeated until the min-heap is empty: Remove all attribute objects with equal, currently minimal value in set Min. Inform all dependent objects in Min of all referenced objects in Min. This way, each dependent object can track the referenced objects including – until now – all dependent values by updating the lists satisfiedRefs and unsatisfiedRefs. That is, if the delivered referenced objects are included in list unsatisfiedRefs they are moved to satisfiedRefs. In our example, we move in attribute object Dep1 the referenced attribute object Ref1 from unsatisfiedRefs to satisfiedRefs. Afterwards, read the next values. If there is no next value return all INDs with referenced attributes in satisfiedRefs. Otherwise, if there is a next value read it and remove all referenced objects from unsatisfiedRefs as the current dependent value is a counter-example for the implied IND. That is, in our example discard Ref2 in Dep1, because it does not include the value 1. Further, move all referenced objects from satisfiedRefs to unsatisfiedRefs as we did not yet see the next dependent value in them and re-add the attribute object to the min-heap (see Fig. 1 b).

Let n be the number of attributes and t the maximum number of values in an attribute. The number of comparisons of SPIDER is $O(nt \log t)$, assuming $t > n$: 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 min-

imum value set (Min). Thus, the complexity of SPIDER to test the IND candidates (i. e., without sorting) is $O(nt \log n)$. Assuming $t > n$, we need $O(nt \log t)$ comparisons for the complete execution of SPIDER, i. e., for sorting and testing. This analysis shows that the complexity to test the IND candidates is lower than the complexity to sort all attribute's values. Further, the complexity depends on the number of attributes, but not in the quadratic number of IND candidates. This is a considerable improvement over previously described approaches.

3.1.3 Results on IND detection

Using SPIDER we were able to detect all INDs on the entire PDB within $\sim$ 6 hours. With previous approaches [5, 19] this task could not be solved. On our smaller databases SPIDER outperformed these approaches by at least a factor of 3 [4].

The BioSQL schema, in which we parsed UniProt, defines 21 foreign keys. Among those we find 19 as INDs. The remaining two foreign key constraints are defined on empty tables and cannot be found by any approach based on analyzing the actual content of a database. Another 11 INDs found by SPIDER that are not defined as foreign keys in BioSQL provide an interesting insight: They result from situations where there are two foreign key attributes A, B referencing a primary key attribute C. In addition to the defined foreign keys $A \subseteq C$ and $B \subseteq C$, SPIDER also detects the IND $A \subseteq B$ and sometimes even $B \subseteq A$. We found this structure between several tables, e. g., a relationship between the tables sequence feature and biosequence over bioentry. In this case, our additional IND is semantically correct, as for each entry in UniProt there must exist exactly one sequence (biosequence), which implies that always biofeature⊆biosequence (but not vice versa in this example). Furthermore, SPIDER found three INDs in 1:1 relationships where only one direction was defined as foreign key constraint, i. e., we detected a semantically correct IND $A \subseteq B$ where only the constraint $B \subseteq A$ was contained in the schema. One example is the relationship between the tables bioentry and biosequence. SPIDER detected three false positives that relate a dependent attribute with a single distinct value to a referenced attribute with about 10,000 distinct values.

In the PDB, SPIDER detected 5,431 INDs with a unique referenced attribute. We applied the two following simple heuristics to prune probably uninteresting, yet satisfied INDs: (i) The referenced attribute must have more than one value; and (ii) At least 1 % of all distinct values of the referenced attribute must be covered by values of the dependent attribute. Both restrictions are very weak and should not exclude interesting possible foreign keys, but they reduced the number of INDs to 2,480. If we roughly estimate that each of the 116 tables in this schema defines one foreign key constraint, we see that further heuristics are necessary to support the step from INDs to foreign keys.

For SCOP we found 11 INDs of which 2 are semantically correct. The other INDs base on a key built from numerical values only. Using the heuristics described above we could exclude one IND. Therefore, finding and testing further filters is an important task for future work. But nevertheless, we aimed at using the INDs for detecting the primary relation of a data source. This task is very well supported by the detected INDs as we will see in the following section.

3.2 Primary Relation

A "primary relation" is a domain specific structural characteristic of life science data sources. This relation represents the objects described in the data source such as proteins or genes. All other relations provide secondary information on these objects.

We use two classes of information to choose the primary relation: the inclusion dependencies and accession number candidates, i. e., attributes with a structure of an accession number. An accession number is in our experience characterized by the following properties: The attribute containing the accession number is unique, the values are typically build of at least one non-digit character, are at least four characters long, and the length of all accession numbers in one data source differs by at most 20 %.

To identify the primary relation of a database, we use the following heuristics:

1. One of the attributes of a primary relation has to be an accession number candidate.
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 the BioSQL schema, we identified three accession number candidates (sg_bioentry.accession, sg_reference.crc and sg_ontology.name). Out of these, Heuristic 2 unambiguously identifies 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 an attribute's values have to fulfill the criteria of minimum length and containment of at least one non-digit character. 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. Furthermore, the accession number candidates in these relations contain exactly the same values, which means they also relate to equivalent INDs. Thus, an automatic procedure cannot distinguish these relations. But a distinction is not necessary for the following steps on detecting inter-schema relationships, because the chosen accession numbers are equal.

For SCOP, we identified one accession number candidate classification.scop_id, which is the old accession number used in SCOP. It is still correct, but the newer version of SCOP uses an accession number build from numbers only [18]. Thus, we cannot detect it as an accession number with our current heuristic. Consequently, we are able to detect cross-references to SCOP using the old accession number when restricting the detection to the primary relation, but unable to detect cross-references using the new accession number. We will need efficient algorithms for detecting inter-schema relationships allowing wider search.

4. DETECTING INTER-SCHEMA RELATIONSHIPS

Life sciences databases heavily cross-reference each other using the data source's accession numbers [10]. We want to detect these references to define links at schema level and at object level. There are several methods used to link objects, i. e., to store and represent the referenced data source

and the accession number. So far, we know the following methods:

1. A data source *references only one data source* using a single attribute, e. g., SCOP only references PDB using the attribute classification.pdb_id. In this case, the referenced data source is given in the attribute name. Another way is the concatenation of a "referenced database code" and the accession number. For instance, CATH – a protein classification data source – uses a single attribute to reference the PDB concatenating the string "pdb|", the accession number and two characters representing the protein domain.
2. A data source *references several data sources.* If the data source uses *one attribute per referenced data source* we can apply SPIDER to find these INDs.
3. The more usual way to reference several data sources is to use a *combination of a "referenced database code" and an accession number.* This combination could be represented in two attributes or concatenated in one attribute. For instance, BioSQL uses the attributes dbxref.dbname and dbxref.accessionnr to reference 67 life science data sources.
4. A completely different method is the reference in a description attribute using natural language.

We cover these kinds of references in the following sections and represent our ideas on detecting each of them.

4.1 References to a Single Data Source

In this section we cover the cases 1 and 2, i. e., one attribute referencing one data source.

4.1.1 Using accession number

If the referencing attribute contains only accession numbers, i. e., no concatenated extra-string, we can apply SPIDER. But we assume that there will never be an exact IND between the referencing attribute and the referenced accession number attribute. E. g., for SCOP there are 91 references out of $25,972$ that are not included in the accession number attribute in PDB. Thus, we need a modification of SPIDER that allows a certain percentage of dependent values not included in the referenced attribute.

A minor modification of SPIDER gives this functionality: The original SPIDER discards a referenced attribute from the dependent attribute's unsatisfiedRefs list after the first counter-example was found, i. e., a dependent value that is not included in the referenced attribute. To allow a certain number of counter-examples we just have to add a counter to each referenced attribute object representing the number of found counter-examples. When the lists unsatisfiedRefs and satisfiedRefs are updated we raise the counter for each attribute object in unsatisfiedRefs. Only if the counter exceeds a given threshold the referenced attribute object – and with it the implied IND – is discarded.

The complexity in comparisons remains $O(n \log n)$, because the synchronization procedure is exactly the same. In fact, the runtime increases slightly, because INDs are discarded later such that attributes are discarded later from the min-heap. Using this modified SPIDER we are able to detect the described cross-reference from SCOP to PDB.

4.1.2 Using prefix or suffix and accession number

To detect cross-references of type "pdb|<accessionNr>" we develop algorithms to detect INDs with prefixes and suffixes, i. e., INDs after removing a prefix and / or suffix from the referencing attribute's values. We plan to modify SPIDER to detect such kind of INDs, because we want to leverage its excellent complexity and its independence in the number of tested IND candidates.

There are several classes of prefix-suffix-INDs: (i) Prefixes and suffixes of constant length or (ii) variable length; (iii) Accession numbers of constant length or (iv) variable length.

The detection of INDs with a suffix at all referencing values (suffix-IND) seems affordable with SPIDER, because the sorted value sets and the basic read-and-compare procedure can be retained. Detecting INDs with a prefix at all referencing values (prefix-IND) seems even more interesting when looking at our known data source cross-references. Solving this problem could be reduced to the problem of suffix-INDs by working on inversed values. This procedure adds large cost for preprocessing the data, but the advantage of parallel tests with good test complexity remains.

Modifying the read-and-compare procedure of SPIDER is a non-trivial task: When can a cursor be moved? And which information do we have to store to get the maximum information at the end? We briefly illustrate this task for detecting Suffix-INDs with variable accession number length and constant suffixes. Assume the dependent value list $\{abcd, abce\}$ and the referenced value list $\{ab, abc, abd\}$. The intended solution here is an suffix-IND with suffix length 1, i. e., we see two references to *abc*. Starting with cursors at the smallest items we compare *abcd* and *ab* and find a match of length 2 with a suffix *cd* of length 2. We move only the referenced cursor to look for longer matches with *abcd*. Thus, we compare *abcd* and *abc* and find a match of length 3 with a suffix *d* of length 1. We move the referenced cursor again and see that there is no match between *abcd* and *abd*. We can now move the dependent cursor on to compare the next value. The cursors point now at *abce* and *abd*. If we only compare these values we miss the matches *ab* and *abc*. Thus, we have to store and transfer the knowledge from the previous value to this value.

In future work we will check the feasibility to integrate this read-and-compare procedure into the parallel test architecture of SPIDER.

4.2 References to Multiple Data Sources

We now look at the third class of cross-references, i. e., references to several data sources. The general idea is to find a horizontal partition on the referencing relation that divides the cross-references by the referenced data sources. Afterwards, each partition can be tested separately.

If the information on the referenced data source and the accession number are represented in two attributes we plan to apply approaches for detecting functional dependencies. The basic idea is, that different data sources use different structures for accession numbers. That is why we look for accession number candidates and normalize their values into patterns of numbers and letters. We expect to find functional dependencies of style "attribute db_name dominates attribute accession_number". Thus, the problem of finding a horizontal partition is reduced to the (still complex) problem of finding functional dependencies [8, 12, 13].

If otherwise, the information on the referenced data source and the accession number are concatenated in values of one attribute we plan to use keyword trees to detect the horizontal partitions. In a keyword tree each path from root to

a leaf represents one word and each branch from a node is unambiguously defined [1]. When each value is inserted as word into a keyword tree, we expect a small branching from the root node into the prefixes representing the referenced data sources followed by a large branching for the following accession numbers.

4.3 References at Object Level

The last, and most delicate class of cross-references are references in natural language text in description fields. We expect these references to be detectable only at object level. We plan to apply text mining approaches in attributes with very long values. That is, we want to identify textual references to accession numbers in other data sources [16].

Another idea for references at object level is based on fields representing DNA or protein sequences. These attributes can be found by their restriction to a small alphabet. Thus, we could use sequence alignment algorithms to detect sequence inclusion and therefore infer a reference. This task is very complex, because even sequence alignment is a complex problem.

5. DUPLICATE DETECTION

For duplicate detection we want to examine the links found in the previous step of detecting inter-schema relationships. While comparing the objects themselves we plan to distinguish cross-references and duplicates.

We plan to use the structural information on intra-schema relationships to consider the entire information of the objects given in both data sources. Thus, this task relates to schema matching techniques and duplicate detection for semi-structured data [9, 14, 21].

6. CONCLUSION

We presented our life science integration project Aladin, which intends to integrate data sources automatically. Its basic idea is twofold: (i) Use the data instance for integration instead of schema information to avoid problems with mostly underspecified schemas. (ii) Use domain knowledge on life science data sources, i. e., identify and use accession numbers, primary relations, and data source cross-references.

Aladin is divided into four steps: (i) data source import into a relational database – the only manual interaction, (ii) detecting intra-schema relationships, (iii) detecting inter-schema relationships, and (iv) duplicate detection.

We presented our results on the first two steps. The major contribution is our algorithm Spider, which identifies inclusion dependencies efficiently. We use these INDs as hint for foreign key constraints and as input to identify the primary relation.

Furthermore, we proposed and discussed several ideas on detecting data source cross-references. We gave a classification of reference representations and discussed approaches to detect them. The results of this step will be used as input for duplicate detection.

7. REFERENCES

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[2] 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.

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

[4] J. Bauckmann, U. Leser, F. Naumann, and V. Tietz. Efficiently detecting inclusion dependencies. In *Int. Conf. on Data Engineering (ICDE 07)*, Istanbul, Turkey, 2007. Poster.

[5] S. Bell and P. Brockhausen. Discovery of data dependencies in relational databases. In *Statistics, Machine Learning and Knowledge Discovery in Databases, ML–Net Familiarization Workshop*, pages 53–58, 1995.

[6] 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(1):235–242, 2000.

[7] P. Brown and P. J. Haas. BHUNT: Automatic discovery of fuzzy algebraic constraints in relational data. In *Int. Conf. on Very Large Databases (VLDB 03)*, pages 668–679, 2003.

[8] 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.

[9] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Transanctions on Knowledge and Data Engineering*, 19(1), 2007.

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

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

[12] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *Computer Journal*, 42(2):100–111, 1999.

[13] I. F. Ilyas, V. Markl, P. J. Haas, P. Brown, and A. Aboulnaga. CORDS: Automatic discovery of correlations and soft functional dependencies. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 647–658, 2004.

[14] D. V. Kalashnikov and S. Mehrotra. Domain-independent data cleaning via analysis of entity-relationship graph. *ACM Transactions on Database Systems (TODS)*, 31(2):716–767, 2006.

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

[16] U. Leser and J. Hakenberg. What makes a gene name? named entity recognition in the biomedical literature. *Briefings in Bioinformatics*, 6(4):357–369, 2005.

[17] U. Leser and F. Naumann. (Almost) hands-off information integration for the life sciences. In *Conf. on Innovative Data Systems (CIDR 05)*, 2005.

[18] L. Lo Conte, S. E. Brenner, T. J. Hubbard, C. Chothia, and A. G. Murzin. SCOP database in 2002: refinements accommodate structural genomics. *Nucleic Acids Research*, 30(1):264–267, 2002.

[19] F. D. Marchi, S. Lopes, and J.-M. Petit. Efficient algorithms for mining inclusion dependencies. In *Int. Conf. on Extending Database Technology (EDBT 02)*, pages 464–476, 2002.

[20] 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.

[21] M. Weis and F. Naumann. DogmatiX tracks down duplicates in XML. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 431–442, Baltimore, MD, 2005.