Scaling Out the Discovery of Inclusion Dependencies

Sebastian Kruse, Thorsten Papenbrock, Felix Naumann

Hasso Plattner Institute
Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam
firstname.lastname@hpi.de

Abstract: Inclusion dependencies are among the most important database dependencies. In addition to their most prominent application – foreign key discovery – inclusion dependencies are an important input to data integration, query optimization, and schema redesign. With their discovery being a recurring data profiling task, previous research has proposed different algorithms to discover all inclusion dependencies within a given dataset. However, none of the proposed algorithms is designed to scale out, i.e., none can be distributed across multiple nodes in a computer cluster to increase the performance. So on large datasets with many inclusion dependencies, these algorithms can take days to complete, even on high-performance computers.

We introduce SINDY, an algorithm that efficiently discovers all unary inclusion dependencies of a given relational dataset in a distributed fashion and that is not tied to main memory requirements. We give a practical implementation of SINDY that builds upon the map-reduce-style framework Stratosphere and conduct several experiments showing that SINDY can process huge datasets by several factors faster than its competitors while scaling with the number of cluster nodes.

1 Discovering Inclusion Dependencies

Given a relational dataset, an inclusion dependency (or short IND) is a constraint expressing that all values of some column (combination) A are also found in some other column (combination) B. This is formally expressed as $A \subseteq B$ and read as "A is included in B". In the context of this IND, A is called the *dependent* column and B the *referenced* column.

This rather simple type of dependency has a broad range of applications, the most prominent one being foreign key discovery. In a dataset that consists of multiple relations, foreign keys describe how their records relate to each other, thus, their knowledge is crucial to any further work with this dataset. However, for various reasons this knowledge can be lacking: datasets are obtained as dumps in some export format that does not capture the schema of the dataset but merely its content; not all databases are capable of enforcing foreign key constraints; and even if so, capturing these constraints is sometimes avoided for performance reasons as the foreign key enforcement involves some computational overhead. Besides foreign keys, INDs are valuable knowledge to many other applications [CTF88, Gry98, LV00], like schema redesign (e.g., detect duplicate information) and data integration (e.g., joining tables from multiple data sources).

As opposed to the detection of other data dependencies, such as functional dependen-

cies [HKPT99, ASN14] or uniqueness constraints [SBHR06, HQRA+13], the IND detection problem suffers the general deficiency of not being able to prune candidates early on: If one wants to know whether $A \subseteq B$ holds for some columns A and B, one usually has to read both columns completely: To confirm this dependency, *each value of* A must be shown to be in B, and disproving it requires to show for a value of A that *each value of* B is different to this value. To this end, previous IND discovery algorithms mostly comprise a *data reorganization phase* to make columns better comparable, which takes the major share of the runtime. Pruning techniques are only applied in the subsequent *comparison phase*, where it does not pay off much.

Given this problem, a promising and new approach to push the envelope of efficient detection is to scale out. We propose SINDY (scalable inclusion dependency discovery), an algorithm that distributes the discovery of INDs among multiple nodes in a computing cluster. In particular, the usually expensive, I/O-bound data reorganization benefits from high I/O throughput rates that can be achieved thereby.

As SINDY reuses some ideas and concepts of previous algorithms, we first present related work in Sec. 2. Then, in Sec. 3, we explain the distributed setting of our approach and explain the discovery of INDs in distributed computing environments in detail in Sec. 4. We conducted various experiments with SINDY on mostly real-world datasets, the results of which we present in Sec. 5. Eventually, we conclude in Sec. 6.

2 Related Work

The discovery of all inclusion dependencies is an important and well studied profiling task [CFP82]. A good part of the previous work deals with the more specific problem of foreign key detection [RAB+09, ZHO+10]. However, since in general only a few INDs correspond to actual foreign keys, these approaches do not aim at finding all the INDs of a dataset and rather optimize for precision and recall than for runtime. In consequence, they are inapplicable to other IND use cases, such as query optimization [Gry98], integrity checking [CTF88], or schema matching [LV00] that require complete results.

Bell and Brockhausen devised a system that starts by collecting data statistics from which it derives a set of IND candidates [BB95]. These IND candidates are then successively checked against the dataset using SQL `join`-statements. This checking procedure also uses intermediate results to prune yet unchecked candidates. Despite the pruning, the proposed algorithm is much slower than more recent works in this area, mostly due to the exhaustive use of SQL `join`-statements for the IND validations.

De Marchi et al. identified the validation of INDs as the bottleneck of the discovery process and developed an algorithm that greatly optimizes it [DLP09]. The algorithm consists of two phases: In the first phase, it transforms the database into an inverted index pointing each value to the set of attributes containing the value. In the second phase, the algorithm then validates IND candidates against the inverted index by intersecting their referenced attribute groups with all attribute sets of the inverted index that also contain the dependent attribute group. The set intersections are an efficient way to validate many IND candidates

simultaneously without querying the dataset for each check. Related works still outperform De Marchi's algorithm, because it needs multiple passes over the inverted index and waives any pruning [BLNT07]. In our case, where the computation is distributed among computers, we can perform only minor pruning, but the heavy use of parallelization compensates this disadvantage. The need to pre-calculate the whole inverted index also makes the De Marchi algorithm inapplicable for datasets that do not fit into main memory. Our distributed algorithm, which targets in particular such large datasets, solves this issue with data distribution and dynamic memory handling.

Bauckmann et al. presented the SPIDER algorithm [BLN06], which builds upon an adapted sort-merge-join. Like the join-algorithm, SPIDER executes in two phases: In the first phase, it sorts the values of each attribute, removes duplicate values, and writes the resulting sorted lists to disk. In the second phase, SPIDER then simultaneously iterates the sorted lists and can thereby determine for each value the set of attributes that contain this value. Like De Marchi's algorithm, SPIDER intersects these attribute groups with all IND candidates whose dependent attributes occur in the attribute group in order to exclude invalid IND candidates. The dynamic construction of attribute groups from the sorted value lists allows SPIDER to omit all those attributes from the validation procedure that do not appear in any valid IND candidate any more. Although this pruning makes SPIDER an efficient IND discovery algorithm on single compute nodes, it prevents the algorithm from being efficiently parallelized. SPIDER also requires an open file for each attribute; this is expensive and limited by many operating systems. Our algorithm, in contrast, is not subject to such restrictions.

3 Distributed Setting

Consider the example database depicted in Tab. 1 with discographic data over two relations. Its six columns yield $6 \cdot (6-1) = 30$ IND candidates, one for each column pair.[1] A closer look reveals that this database comprises six INDs, e.g., $album \subseteq track\_alb$, $position \subseteq track\_pos$, and $album \subseteq title$. To extract these six INDs from the set of IND candidates, SINDY applies a special procedure that requires only a single pass over the data rather than comparing each candidate separately. In particular, this procedure can be scaled out over many computers as we detail in Sec. 4.

Hence, SINDY is designed to run on clusters. In such a distributed scenario, multiple independent computers (*workers*) are available for the execution of the algorithm. These workers have no shared state or control, which renders the design of distributed algorithms particularly challenging. However, the workers are interconnected via a network and can exchange messages and data. Furthermore, the input datasets are (redundantly) distributed over these nodes, e.g., as CSV files in an HDFS[2]. Facing these circumstances, our main goal is to distribute the incurring workload among the workers in a manner that utilizes all their available resources like CPU cycles, main memory, and especially I/O bandwidth

[1] In practical scenarios, this number can be sometimes reduced by incorporating already known metadata such as datatypes [DLP09] or minimum and maximum values.

[2] `http://hadoop.apache.org/`

album	position	title
Thriller	4	Thriller
Thriller	5	Beat it
Back in Black	1	Hells Bells
Back in Black	6	Back in Black

(a) Example table *Tracks*.

track_alb	track_pos	name
Thriller	4	Michael Jackson
Thriller	4	Vincent Price
Thriller	5	Michael Jackson
Thriller	5	Eddie van Halen
Back in Black	1	AC/DC
Back in Black	6	AC/DC

(b) Example table *Artists*.

Table 1: An example database with two relations.

simultaneously. Hereby, we want to achieve increased performance as more workers – and thus more resources – are available for the computation.

Map/Reduce frameworks like Apache Hadoop that support this goal have gained remarkable attention over the last years and are nowadays widely used. By means of a simple programming model, they allow to create distributed applications, that can be executed in a fault-tolerant and scalable manner (wrt. available main memory as well as the number of workers). Following this trend, generalizations of the map-reduce paradigm have been researched, especially by the Stratosphere research project. Amongst others, in Stratosphere complex processing pipelines replace the static map-reduce structure [ABE+14].

We particularly took care that SINDY can be easily implemented within such frameworks, to benefit from their aforementioned advantages. However, using such frameworks comes with the drawback that they mostly offer high-level interfaces that only give control over the data processing to a certain level of detail. This is similar to formulating a SQL query, where one also cannot specify the algorithms to be used for joins, sortations etc. Nevertheless, we believe that this trade-off is worthwhile for IND discovery, because frameworks like Stratosphere can leverage this abstract specification for optimization purposes and are capable of intelligently picking suitable data processing strategies.

4 Distributing the discovery of INDs

The goal of IND discovery is to state for every irreflexive column pair (A, B) in the given relational dataset if $A \subseteq B$ holds. However, instead of checking each column pair individually, SINDY efficiently checks all pairs in a single pass over the data. We provide the foundation for our approach with an alternative formalization of INDs: Let $O = A \bowtie B \bowtie \ldots$ be the full outer join of all columns in a database, then

$$A \subseteq B \Leftrightarrow \forall t \in O : (t[A] \neq \bot \Rightarrow t[B] \neq \bot) \tag{1}$$

Existing algorithms already use this formalization implicitly. For instance, SPIDER [BLN06] performs an optimized sort-merge join to find INDs and MIND [DLP09] pivots the database for this reason. Then, they iterate the join result once or more often to find the inclusion dependencies as described above. SINDY consists of a join and a checking phase, too, but

performs both phases in a distributed manner. In the following, we explain these phases in detail and afterwards describe how they are implemented in Stratosphere.

4.1 Join columns to attribute sets

In the first phase, the full outer join $O = A \rtimes B \rtimes \ldots$ of all columns in the database is computed. Since each tuple in the join product is of the form $t_{O,v} = (v_A, v_B, \ldots)$ with $v_X \in \{v, \perp\}$ (i.e., each tuple can only contain null values and one distinct value v), these tuples can be simply represented as $r(t_{O,v}) = (v, \{X | t_{O,v} \neq \perp\})$. Additionally, Equation 1 neither makes use of the actual values in O nor profits from tuples where the exact same attributes are null; therefore, we can discard the value and opportunely remove duplicates. As the result of this process, SINDY yields *attribute sets* that represent O.

Figure 1 exemplifies the dataflow of the outer join procedure on two workers with some tuples of the example from Sec. 3. To find the attribute sets, each worker initially reads its local share of the input data and splits each read record into *cells*, i.e., tuples with a value and a singleton set containing the corresponding attribute. For instance, the record (“Thriller”, 4, “Thriller”) from the example table $tracks$ is split into the three cells (“Thriller”, {album}) and (4, {position}), and (“Thriller”, {title}). SINDY now re-orders all cells among the workers of the cluster: For n workers with IDs $0, \ldots, n-1$ we define the target worker for each cell from its value by means of a partition function $p\colon \mathbb{V} \to \{0, 1, \ldots, n-1\}$. It is thereby guaranteed that cells with the same value are placed on the same worker. Ideally, the partition function distributes the values evenly among the workers. A suitable choice for p is $p(v) \stackrel{def}{=} \mathit{hash}(v) \bmod n$.

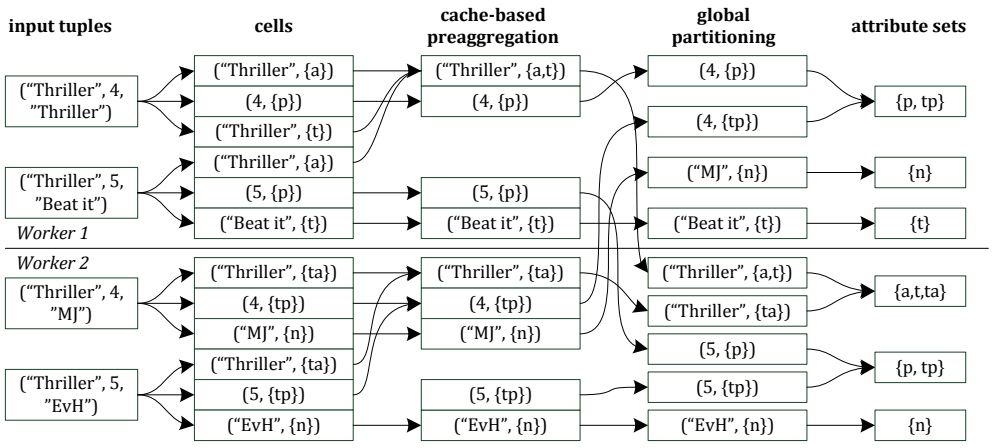


Figure 1: Dataflow for a distributed full outer join on all columns of a dataset. The attribute names and some values are abbreviated for the purpose of lucidity.

Finally, each worker groups all its cells by their values and aggregates their attribute sets

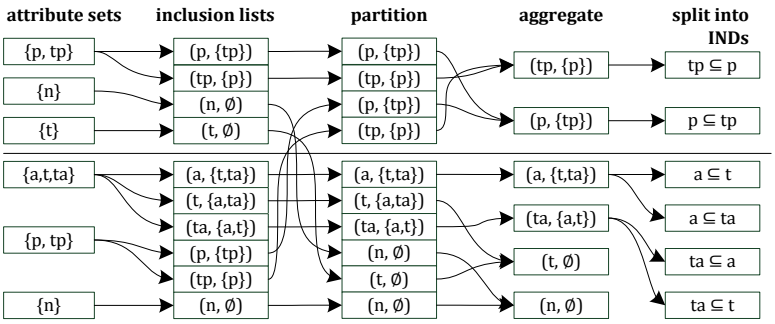


Figure 2: Dataflow for deriving INDs from the outer join product.

using the union operator. The value of each aggregated cell can be discarded, leaving only the attribute sets. Note that the aggregation with the union operator is a commutative, associative operation – it is therefore possible to pre-aggregate the cells on each worker before they are sent to the target worker to reduce network load. However, experiments showed that this almost doubles the runtime. The rationale behind this is that the aggregation for cells requires disk-based execution, which dominates the runtime. Performing an additional pre-aggregation potentially requires each cell to be written and read twice as many times as when performing only the aggregation. Nevertheless, oftentimes columns of a dataset contain only a few distinct values that are then frequently read. It makes good sense to cache cells with values on the reading worker that occur repeatedly and send only cells to the target worker that are not found in this cache yet.

4.2 Derive INDs from the join product

The join product from the previous phase is present as a distributed set of attribute sets. To identify INDs from this product we reformulate the criterion from Equation 1 as it is similarly applied in [DLP09] and [BLN06]:

$$A \subseteq B \quad \Leftrightarrow \quad \forall \text{ attribute set } S\colon (A \in S \Rightarrow B \in S) \quad \Leftrightarrow \quad B \in \bigcap_{S\colon A \in S} S \qquad (2)$$

SINDY performs this check by creating *inclusion lists* for each attribute set as depicted in Fig. 2 for our example. Let S be an attribute set with n attributes, then n inclusion lists of the form $(A, S \setminus A)$ with $A \in S$ are created. These inclusion lists are globally grouped by the first attribute and the attribute sets are intersected. Similar to the union aggregation in the previous phase, the intersection aggregation is associative and commutative and thus allows pre-aggregation as well; the same mechanisms can be reused for this aggregation. Furthermore, the number of attributes in a dataset is usually much lower than the number of values. Therefore, cache-based pre-aggregation seems to be especially suitable here; and once an inclusion list associates an attribute with the empty set, any further aggregation

with an inclusion list for the same attribute is moot and can be omitted. Nevertheless, this approach also allows to handle huge schemata, as we can perform the aggregation of the inclusion lists with disk-based execution if necessary. Finally, the aggregated inclusion lists can be disassembled into INDs: For every inclusion list (A, S), each $A \subseteq B$ $(B \in S)$ is a valid IND, because of Equation 2. Hereby, every inclusion list yields a set of unique INDs, as the attribute A is distinct in every inclusion list.

The logic of SINDY can be directly expressed in terms of map and reduce operators and is therefore well-suited for the implementation on frameworks like Stratosphere. In Fig. 3, we show the execution plan for the unary IND detection. Note that this exact plan can also be implemented with Spark – and in addition, it can be split into two Map/Reduce jobs to be deployed on Hadoop.

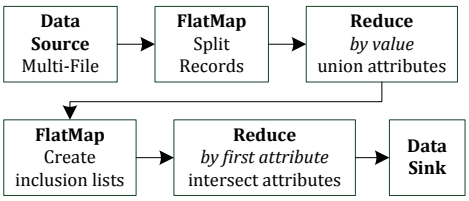


Figure 3: Stratosphere plan for unary IND detection.

5 Evaluation

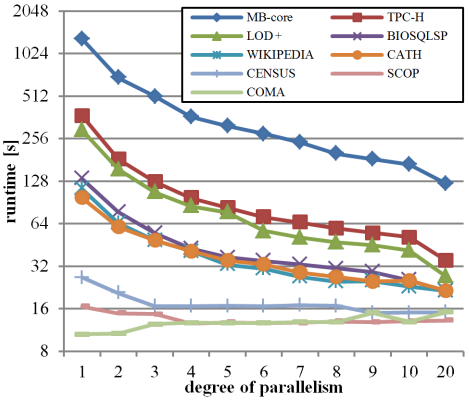
To evaluate SINDY, we performed several experiments with different datasets[3]. Most importantly, we tested its scale-out behavior and compared the runtimes with the state-of-the-art algorithm SPIDER. For our experiments, we used a cluster with a dedicated master node (2.67 GHz Intel Xeon processor and 8 GiB of RAM) and 10 workers with 2.6 GHz Intel Core 2 Duo processor and 8 GiB of RAM of which we dedicated 6 GiB for Stratosphere. All computers were interconnected with a switch router using Gigabit ethernet. We used Hadoop Filesystem 2.2 and Flink 0.6.2[4], which is the most recent version of the Stratosphere software. SPIDER was executed on a machine with two 8-core 2 GHz Intel Xeon processors, 128 GiB of RAM, and a RAID-1 storage.

5.1 Scale-out and comparison

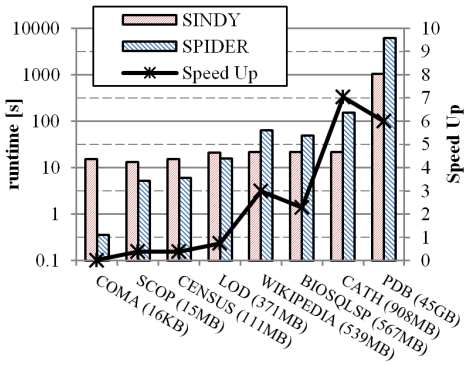
We measured the scale-out behavior of SINDY with small (starting from 16 KiB, COMA) and large datasets (up to 5.8 GiB, MB-core) and present the results in Fig. 4a. We used one to ten workers for the execution and finally added intra-worker parallelization of factor 2 (last measurement). For small datasets that have a short runtime of only a few seconds, the scale-out is not appropriate and can even slow down the execution. For large datasets, though, scaling-out pays off. Adding a second worker can already halve the execution time. For all ten workers, the scale-out is not completely linear, though. We suspect that this is the typical load balancing issue: oftentimes we could see that some workers took

[3] http://hpi.de/naumann/projects/repeatability.html
[4] https://flink.incubator.apache.org/

longer than others to complete their work share. The runtime is determined by the slowest worker. The more workers are involved, the higher is the chance that one of these workers has to cope with an higher-than-average workload.



(a) Scale-out of SINDY on different datasets.



(b) Performance comparison of SINDY and SPIDER.

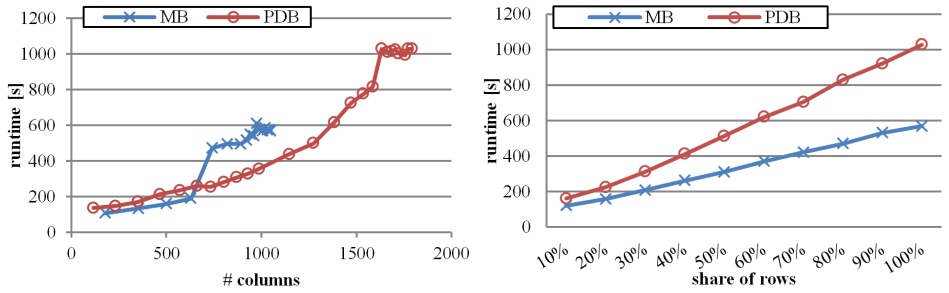
Figure 4: Scale-out behavior and comparison with state-of-the-art.

Next, we compared the runtimes of SINDY with full scale-out to the runtimes of SPIDER (Fig. 4b). As seen before, SINDY does not adapt well to small datasets and is outperformed by SPIDER there. However, for the larger datasets, starting with WIKIPEDIA, the advantages of the scale-out take effect. For our setup with ten workers, the full scale-out potential is apparently reached at around 900 MB. On the largest dataset, SINDY's runtime of 17 minutes in fact saves 85 minutes in comparison to SPIDER. This speedup can be presumably further improved by adding new workers to the cluster, while for the I/O-bound SPIDER the hardware improvements are limited and costly; one could switch to SSDs or extend the RAM to fit datasets in main memory.

5.2 Row and column scalability

Furthermore, we investigated how SINDY behaves as its input data grows. Relational data can grow in two dimensions, namely the number of columns and the number of rows. To obtain meaningful measurements, we based our experiments on the two larger real-world datasets PDB (44.9 GiB) and MB (26.8 GiB).

Figure 5a shows the column-scale behavior of SINDY. To vary the number of columns for both datasets, the original input files remained unchanged, but for each test execution, we used only the first k columns of each file. The idea behind this is to let the amount of processed data grow approximately proportionally with the number of columns on the face of an uneven distribution of the data volume among the input files. For the MB dataset, the processing time increases irregularly with the number of processed columns. In particular, there is a large hop at the area of around 750 columns. We explain this jump, because



(a) Behavior of SINDY on increasing numbers of columns.

(b) Behavior of SINDY on increasing numbers of rows.

Figure 5: Behavior of SINDY on increasing numbers of columns and rows.

from there on a large column with JSON data is included that makes up an estimated 50 % of the dataset's overall data volume. For the PDB dataset, the runtime grows seemingly quadratically with the number of columns but settles after around 1,600 columns. The former observation can be explained with the quadratic growth of IND candidates and the latter with the fact that at this point 31 columns per table are used – exactly the number of columns of a large table that contains over 90 % of PDB's data volume.

For the experiments with row scalability, we also read the complete source files and sampled the input rows for SINDY of each file on the fly. Apparently, the algorithm scales linearly with the input data, presumably because using x % of the rows also yields approximately x % of the data volume. In summary, these observations suggest that on typical datasets, the data volume is the main runtime driver for SINDY.

6 Conclusion and Future Work

We presented SINDY, an efficient IND discovery algorithm that can be scaled out on clusters to process large datasets within reasonable time. This scalability is achieved by partitioning the dataset into cells and merging them in a distributed fashion. The result of this merge is then split up into IND candidates that are again merged in a distributed way. We have shown that this algorithm scales well with the volume of a dataset and allows for almost linear scale-out for large datasets. We did not encounter a dataset that could break the algorithm due to its size, even for datasets with almost 2,000 columns. Moreover, SINDY can be quickly adapted to many distributed environments as its logic can be seamlessly implemented on map-reduce-style frameworks like Stratosphere.

Recently, we have extended SINDY to find not only unary but also n-ary INDs using an apriori-based approach like MIND [DLP09]. First experiments have shown that this - even in combination with scale out - is only a limited remedy for the exponential growth of the IND search space. Therefore, our future effort will deal with the question on how to efficiently discover n-ary INDs in scale-out scenarios.

Acknowledgements. This research was funded by the German Research Society (DFG grant no. FOR 1306).

References

[ABE+14] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal*, to appear 2014.

[ASN14] Ziawasch Abedjan, Patrick Schulze, and Felix Naumann. DFD: Efficient Discovery of Functional Dependencies. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, 2014.

[BB95] Siegfried Bell and Peter Brockhausen. Discovery of Data Dependencies in Relational Databases. Technical report, Universität Dortmund, 1995.

[BLN06] Jana Bauckmann, Ulf Leser, and Felix Naumann. Efficiently Computing Inclusion Dependencies for Schema Discovery. In *ICDE Workshops*, 2006.

[BLNT07] Jana Bauckmann, Ulf Leser, Felix Naumann, and Veronique Tietz. Efficiently detecting inclusion dependencies. In *ICDE Posters*, 2007.

[CFP82] Marco A. Casanova, Ronald Fagin, and Christos H. Papadimitriou. Inclusion Dependencies and Their Interaction with Functional Dependencies. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, 1982.

[CTF88] Marco A. Casanova, Luiz Tucherman, and Antonio L. Furtado. Enforcing Inclusion Dependencies and Referencial Integrity. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1988.

[DLP09] Fabien De Marchi, Stéphan Lopes, and Jean-Marc Petit. Unary and n-ary inclusion dependency discovery in relational databases. *Journal of Intelligent Information Systems*, 32:53–73, 2009.

[Gry98] Jarek Gryz. Query folding with inclusion dependencies. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 1998.

[HKPT99] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. TANE: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100–111, 1999.

[HQRA+13] Arvid Heise, Jorge-Arnulfo Quiane-Ruiz, Ziawasch Abedjan, Anja Jentzsch, and Felix Naumann. Scalable Discovery of Unique Column Combinations. In *Proceedings of the VLDB Endowment*, 2013.

[LV00] Mark Levene and Millist W. Vincent. Justification for inclusion dependency normal form. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):281–291, 2000.

[RAB+09] Alexandra Rostin, Oliver Albrecht, Jana Bauckmann, Felix Naumann, and Ulf Leser. A Machine Learning Approach to Foreign Key Discovery. In *Proceedings of the ACM Workshop on the Web and Databases (WebDB)*, 2009.

[SBHR06] Yannis Sismanis, Paul Brown, Peter J. Haas, and Berthold Reinwald. Gordian: efficient and scalable discovery of composite keys. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2006.

[ZHO+10] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M. Procopiuc, and Divesh Srivastava. On Multi-column Foreign Key Discovery. *Proceedings of the VLDB Endowment*, 3:805–814, 2010.