WPI-CS-TR-02-15 May 2002

Discovery of High-Dimensional Inclusion Dependencies

by

Andreas Koeller
Elke A. Rundensteiner

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

Abstract

Determining relationships such as functional or inclusion dependencies within and across databases is important for many applications in information integration. When such information is not available as explicit meta data, it is possible to discover candidates for dependencies from the source database extents. However, the complexity of such discovery problems is typically exponential in the number of attributes.

We present algorithm FIND_2 for the discovery of inclusion dependencies in high-dimensional relations in the order of 100 attributes. Our algorithm solves the inclusion-dependency discovery problem by mapping it into a progressive series of clique-finding problems in k-uniform hypergraphs. For even larger problems, FIND_2 is enhanced by several heuristics that significantly reduce the search space in practice. Extensive experimental studies both with and without heuristics illustrate the algorithm's efficiency on a variety of real-world data sets.

We also present algorithm HYPERCLIQUE, a new efficient algorithm to find cliques in k-uniform hypergraphs.

1 Introduction

In this work, we are concerned with the discovery of meta-information (dependencies and interrelationships) in databases, and in particular with the discovery of inclusion dependencies (INDs). In the context of data integration, INDs can help to solve a very common and difficult problem: discovering redundancies across data sources. Due to the nature of data and its generation, information is often stored in multiple places, with large amounts of redundancy. When trying to integrate data sources that are likely to be (even partly) redundant, a method to *discover* such redundancies is needed.

One example in which redundancy discovery would be helpful is the Evolvable View Environment (EVE) [NLR98] which is concerned with maintaining views under schema changes by replacing deleted information sources with redundant data from alternative sources. Other systems in which IND discovery could be used to indentify potential data sources include data warehouses [Gar98], multidatabase systems such as Infomaster [GKD97], schema integration systems such as SemInt [LC00], or other schema matching approaches like Clio [MHH00]. A reliable algorithm to discover INDs will enable such integration systems to incorporate new data sources that previously would not have been used since their relationships with existing data was not known.

In general, the discovery of INDs will be beneficial in any effort to integrate unknown databases. A manual extraction of INDs by domain experts does not seem feasible in such projects due to the large number of information sources in the world, the potential high number of attributes in real-world relations (50–100 attributes are common), and a widespread lack of reliable meta-information about legacy databases. Our tool can be useful to a human integrator as a decision support tool for such integration projects.

As shown in the literature [KMRS92], the problem of finding even one maximal inclusion dependency for two given relations is inherently NP-hard (as a function of the number of attributes in the relations). In the worst case, there is an exponential number of such dependencies for a given set of attributes in two relations. We will show that despite this inherent complexity, the discovery of inclusion dependencies is tractable for real-world databases. The limits of applicability for our algorithm are relations of approximately 100–150 attributes. We have also developed heuristics to make the algorithm applicable for even larger problems. The algorithm's performance is linear in the *extent size* (number of tuples) of each relation.

To the best of our knowledge, our tool is the first that solves the IND-finding problem for large numbers of attributes. A recent Apriori-like solution [dLP02] that shows good performance for relations with few attributes does not scale well to numbers of attributes (and in particular maximal IND sizes) beyond 10.

The remainder of the paper is organized as follows: Section 2 reviews background on INDs. Section 3 gives an overview of our solution approach. Section 4 presents algorithm FIND_2 as well as the underlying hyperclique-finding algorithm and shows its complexity and correctness. Section 5 reviews experimental results, and Sections 6 and 7 give related work and conclusions, respectively.

2 Background

Inclusion dependencies are formally defined below.

Definition 1 (IND) *Let $R[a_1, a_2, \ldots, a_n]$ and $S[b_1, b_2, \ldots, b_m]$ be (projections on) two relations. Let X be a sequence of k distinct attribute names from R and Y a sequence of k distinct attribute names from S, with $1 \leq k \leq \min(n, m)$. Then, an* ***inclusion dependency (IND)*** *σ is an assertion of the form $\sigma = R[X] \subseteq S[Y]$. k is called the* ***arity*** *of σ.*

Definition 2 (valid) *An IND $\sigma = (R[a_{i_1}, \ldots, a_{i_k}] \subseteq S[b_{i_1}, \ldots, b_{i_k}])$ is* ***valid*** *between two relations R and S if the sets of tuples in R and S satisfy the assertion given by σ. Otherwise, the IND is called* ***invalid*** *for R and S.*

INDs are defined over *sequences* of attributes, not sets, since the order of attributes is important. That is, INDs are not invariant under permutation of the attributes of only one relation.

Casanova *et al.* [CFP82] describe a complete set of inference rules for INDs. Repeated application of their rules will generate all valid INDs that can be derived from a given set of valid INDs (i.e., those rules form an *axiomatization* for INDs). The most important axiom for our purpose is stated below. The other two axioms show reflexivity and transitivity of valid INDs.

Axiom 1 (projection and permutation) *(from [CFP82]) If $R[A_1, \ldots, A_m] \subseteq S[B_1, \ldots, B_m]$ is valid (by Def. 2), then $R[A_{i_1}, \ldots, A_{i_k}] \subseteq S[B_{i_1}, \ldots, B_{i_k}]$ is valid for any sequence $(i_1, \ldots, i_k)$ of distinct integers from $\{1, \ldots, m\}$.*

Note that permutation refers to "synchronous" reordering of attributes on both sides, i.e., $R[X, Y] \subseteq S[X, Y] \implies R[Y, X] \subseteq S[Y, X]$, but $\neg(R[X, Y] \subseteq S[X, Y] \implies R[Y, X] \subseteq S[X, Y])$.

Definition 3 (derived INDs) *A valid IND σ can be* ***derived*** *from a set Σ of valid INDs, denoted by $\Sigma \models \sigma$, if σ can be obtained by repeatedly applying the above axioms on some set of INDs taken from Σ.*

One very important observation on INDs is that a k-ary IND with $k > 1$ naturally implies a set of m-ary INDs, with $1 \leq m \leq k$.

Corollary 1 *Let σ_k be a k-ary IND between two given relations R and S. Let Σ_m^k be the set of all m-ary INDs ($1 \leq m < k$) between R and S that can be derived from σ_k by Axiom 1. Then, $|\Sigma_m^k| = \binom{k}{m}$.*

We say that Σ_m^k is **implied by** σ_k. Validity of INDs is preserved under projections and permutation, by Axiom 1. In order to describe all inclusion information between two relations it is therefore not necessary to list *all* INDs between two relations, but rather only INDs that are not implied by any other IND. A small set of such **maximal** INDs, from which all others can be generated, will suffice to describe all INDs.

Example 1 *Fig. 1 introduces a running example of two relations R and S. The following maximal INDs hold between R and S: $R[A, B, C, D, E] \subseteq S[A, B, C, D, E]$, $R[D, F, G] \subseteq S[D, F, G]$, $R[E, F] \subseteq S[E, F]$, $R[E, G] \subseteq S[E, G]$. Our algorithm has to discover this generating set of INDs for R and S.*

Definition 4 (equality of INDs) *Two INDs $R[a_1, \ldots, a_m] \subseteq S[b_1, \ldots, b_m]$ and $R[c_1, \ldots, c_m] \subseteq S[d_1, \ldots, d_m]$ are* ***equal*** *if and only if there is a sequence $(i_1, \ldots, i_m)$ of distinct integers $1, \ldots, m$ such that $(a_1 = c_{i_1}) \wedge (a_2 = c_{i_2}) \wedge \ldots \wedge (a_m = c_{i_m}) \wedge (b_1 = d_{i_1}) \wedge (b_2 = d_{i_2}) \wedge \ldots \wedge (b_m = d_{i_m})$.*

For example, $R[D, F, G] \subseteq S[D, F, G] = R[D, G, F] \subseteq S[D, G, F] \neq R[D, F, G] \subseteq S[D, G, F]$.

Definition 5 (generating set of INDs) *Consider a set of valid inclusion dependencies: $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$. A* ***generating set*** *of Σ, denoted by $\mathcal{G}(\Sigma)$, is a set of valid INDs with the following properties[1]:*

1. $\forall \sigma \in \Sigma : \mathcal{G}(\Sigma) \models \sigma$

[1] The symbol $\setminus$ stands for "set-difference".

R

A	B	C	D	E	F	G
1	2	3	4	5	6	7
8	9	10	11	12	13	14

S

A	B	C	D	E	F	G
1	2	3	4	5	0	0
0	0	0	4	0	6	7
0	0	0	0	5	6	0
0	0	0	0	5	0	7
0	0	0	0	0	6	7
8	9	10	11	12	13	14

Figure 1: Running Example for IND-Finding Problem.

2. $\forall \sigma \in \mathcal{G}(\Sigma) : \neg((\mathcal{G}(\Sigma) \setminus \sigma) \models \sigma)$

In words, the generating set $\mathcal{G}(\Sigma)$ contains exactly those valid INDs from which *all* valid INDs in Σ can be derived. The set can be constructed by first including all $\sigma \in \Sigma$ into $\mathcal{G}(\Sigma)$ and then removing all σ for which property 2 does not hold. The set is minimal since removing any IND σ from a $\mathcal{G}(\Sigma)$ for which property 2 holds would by definition violate property 1. Therefore, generating sets contain all information about inclusion dependencies between relations in a minimal number of INDs.

Another important observation used by our algorithm is stated below.

Theorem 1 *The validity of all $(m-1)$-ary INDs that are derivable from an m-ary IND σ is a necessary but not a sufficient condition for the validity of σ.*

Proof: Necessity follows immediately, and insufficiency can easily be shown by a counterexample. In our running example (Fig. 1), such a counterexample would be $\neg(R[E,F] \subseteq S[E,F] \wedge R[E,G] \subseteq S[E,G] \wedge R[F,G] \subseteq S[F,G] \implies R[E,F,G] \subseteq S[E,F,G])$ $\square$

3 Solution Approach: Mapping into a Graph Problem

Given two relations R_1 and R_2 stored in one or more DBMS, our problem can now be stated as the problem of finding the generating set of inclusion dependencies (INDs) between R_1 and R_2. In general, an algorithm to solve this problem must perform two tasks: generate a (ideally small) number of INDs to actually test against a database, and then test individual INDs for validity (Def. 2).

Note that we are not looking for *all* INDs but for a generating set. Since all other INDs can be derived from the generating set by projection and permutation, it is sufficient to consider only this set. As we will see shortly, trying to generate *all* valid INDs is impractical due to their large number while a generating set can be found more efficiently.

3.1 Assumptions

We assume that INDs can be defined in the underlying data model. That is certainly true for the relational model, but object-oriented models and to a certain extent semi-structured models also have a notion of data inclusion. We furthermore require that the data model must include the concept of *attributes* ("columns" of data). We do not consider data models with complex objects (such as nested relations) or cross-references between data objects.

We furthermore assume that equality between tuples is a binary function, i.e., two tuples (and thus each matching pair of their values) are either equal or they are not equal.

Also, we restrict ourselves to databases that are capable of testing an arbitrary IND for validity (for example through some query language, like SQL), and that the attribute and relation names of a database can be queried (supported by all commercial SQL-database systems as well as by wrappers like JDBC).

3.2 Worst-Case Complexity of the Problem

The worst-case complexity of the problem is a function of the number of attributes in both relations. The maximum number of distinct INDs between two relations can be computed as follows. For two relations R and S with k_R and k_S attributes, respectively, one can form $k_R \cdot k_S$ *unary* INDs. It is possible for all such INDs to be valid at the same time, namely when all attributes in both relations have exactly the same data (though this is not likely to occur in practice). The number of k-ary INDs in general is determined by the number of pairs of k-ary subsets of the attributes in each relation. For each such pair, there are $k!$ INDs, since each permutation of *one side* of an IND, while keeping the other side unchanged, gives a new IND (i.e., an IND not equal to any previously generated IND). Permutations of *both* sides do not lead to new INDs (see Def. 4), as this process would generate INDs *equal* to previously generated ones. Assuming without loss of generality that $k_S < k_R$, the total number of INDs between R and S, denoted by I, is therefore

$$I(k_R, k_S) = \sum_{i=1}^{k_S} \binom{k_R}{i} \cdot \binom{k_S}{i} \cdot i!$$

A naïve brute-force IND-finding algorithm for a pair of relations could first generate all possible INDs and then test each of them for validity by an SQL query. It would thus have a complexity in at least $O(I(k_R, k_S))$, even if IND-testing could be done in constant time. Clearly, since this number grows extremely fast, it is not meaningful to test for or even generate all IND candidates for a pair of relations, even if those relations have a small number (say < 10) of attributes. Also, as $I(k_R, k_S) \notin O({k_R}^n \cdot {k_S}^m)$ for any finite n, m, the problem cannot be solved in polynomial time since its solution cannot be enumerated in polynomial time. A very recent paper [dLP02] demonstrates that for small problem sizes (few attributes in either relation *and* very small number of actual valid INDs), efficient enumeration along the line of the Apriori-algorithm [AR94] seems feasible. In this paper, we instead make use of the fact that it is possible to find a *generating set* of valid INDs without even enumerating all valid INDs, reducing the complexity significantly.

3.3 Mapping to a Graph Problem

We now propose a mapping of our problem into a more tractable graph problem. This will also give us additional insights into the inherent complexity of the problem and lead to our solution. For clarity, we will give brief definitions of the concepts used.

Definition 6 (k-hypergraph) *A k-**uniform hypergraph** (or k-**hypergraph**) is a pair $G = (V, E)$ with the set V of nodes and the set E of edges. An element $e \in E$ is a set with cardinality k of pairwise distinct elements from V, denoted by $\{v_1, \ldots, v_k\}$. An element $e \in E$ is called a k-*hyperedge*. k is called the* rank *of graph G.*

Example 2 *An example for a 3-hypergraph is given in Figure 2. This graph has 5 nodes and 6 edges: $\{1, 2, 3\}$, $\{1, 2, 4\}$, $\{1, 2, 5\}$, $\{1, 3, 4\}$, $\{2, 3, 4\}$, and $\{3, 4, 5\}$. Each edge is drawn as three lines, connected by a filled circle $\bullet$.*

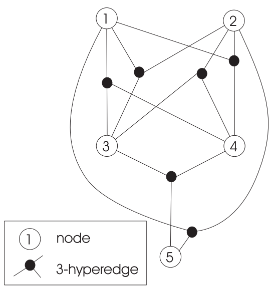

Figure 2: A 3-hypergraph with 6 Edges.

Clearly, an undirected graph is a special case of k-hypergraph with $k = 2$. The requirement that the nodes inside each edge are pairwise distinct disallows self-loops. The above definition extends the concept of undirected graphs to k-hypergraphs. We treat edges as *sets* of nodes rather than a sequence (tuple) of nodes as the edges are not "directed".

Definition 7 (hyperclique) *Let $G = (V, E)$ be a k-hypergraph. A* ***hyperclique*** *is a set $C \subseteq V$ such that for each k-subset S of distinct nodes from C, the edge corresponding to S exists in E. The cardinality of a hyperclique C, denoted by $|C|$, is the number of nodes in C. A single node with no adjacent edges is a hyperclique of cardinality 1.*

In analogy to above, a clique is a 2-hyperclique. Note that a k-hypergraph (with $k > 2$) cannot have hypercliques with cardinalities $2 \ldots k-1$. A (hyper)clique is either a node with no edges or it must have at least as many nodes as there are nodes in an edge, i.e., k. As an example, the hypergraph in Fig. 2 has hypercliques $\{1, 2, 3, 4\}$, $\{1, 2, 5\}$, and $\{3, 4, 5\}$.

Corollary 2 *Consider a k-hypergraph $G = (V, E)$. A set $C \subseteq V$ is a hyperclique if the k-hypergraph G_C induced by C has $\binom{|C|}{k}$ edges.*

Proof: This property follows from the observation that the nodes of a k-hypergraph G_C form a hyperclique C if all possible edges between these nodes exist in G (by Def. 7). There are exactly as many possible distinct edges in such a k-hypergraph as there are k-subsets of $|C|$ elements, such that the number of possible edges is $\binom{|C|}{k}$. **q.e.d.** $\square$

We now show how to map the problem of finding INDs between relations R and S to the problem of finding (hyper)cliques in k-uniform hypergraphs. We first map the set of valid IND to a set of hypergraphs G_m, $(2 \le m < k)$ according to Fig. 3 and then show that, for $k > m \ge 2$, any set Σ_m^k of INDs implied by a valid σ_k maps to a hyperclique in the corresponding hypergraph G_m (Thm. 2). The mapping of valid INDs between R and S to a set of hypergraphs is given in Fig. 3.

1. Create a set V whose elements are all *unary valid INDs* between R and S. Let R and S have k_R and k_S attributes, respectively and assume without loss of generality that $k_S < k_R$.
2. For $k = 2, \ldots, k_S$, create a k-hypergraph as follows:
   (a) Create a set E_k whose elements are all *k-ary valid INDs* between R and S. Note that each k-ary valid IND σ can be seen as a set of exactly k unary valid INDs, by Corollary 1.
   (b) Create a graph $G_k = (V, E_k)$. Note that its *nodes* are all unary valid INDs between R and S, while its *edges* are all k-ary valid INDs between those relations.

Figure 3: Mapping a Set of INDs to a Set of Graphs.

Note that this mapping does not yet support the notion of *generating set* (Def. 5) but rather expresses *all* valid INDs between two relations. However, a slight change in the mapping (by simply keeping only valid INDs *not implied* by higher-arity valid INDs) would correspond to this refined notion.

Now recall that, by Corollary 1, a k-ary valid IND implies $\binom{k}{m}$ m-ary INDs (with $m \le k$). We observe the following theorem.

Theorem 2 *Given the relations R and S, with k_R and k_S attributes, respectively and $k_S < k_R$, consider a collection of (i-uniform) hypergraphs $\{G_2, G_3, \ldots, G_{k_S}\}$ representing the INDs between R and S (as defined in Fig. 3). Furthermore, let σ_k be a k-ary valid IND between R and S. For a number m, with $m < k$, construct a set Σ_m of all the m-ary INDs implied by σ_k. Note that each element of Σ_m corresponds to one m-hyperedge in G_m. Then, the set E_m of those edges forms a m-Hyperclique in G_m.*

Proof: By Definition 7, the set of edges of a hyperclique C in an m-hypergraph $G_m = (V, E)$ correspond to exactly all m-subsets of nodes from C. Also, the m-ary INDs implied by an IND σ_k (with $m < k$) are exactly all m-subsets of the set of nodes implied by σ_k. If C is the set of nodes implied by σ_k, clearly there is an isomorphism between the m-subsets of C and the m-ary INDs implied by σ_k. **q.e.d.** □

Note that Thm. 2 gives a way of finding at least all valid INDs, but may find also INDs that are not actually valid (because of Thm. 1).

3.4 The Clique-Finding Problem

The **Clique-Finding Problem** (also called the Maximum Clique Problem) is a well known NP-complete graph problem. Efficient algorithms for reasonably sized graphs (i.e., 2-hypergraphs, with fewer than about 100 nodes) are given in the literature, e.g., [BK73]. The Clique-Finding Problem extends naturally to k-hypergraphs.

The NP-complexity of the clique-problem is mainly due to the exponential number of possible cliques in a graph. However, there are algorithms that will take polynomial time *for the generation of each clique*, such that the complexity can be expressed by a polynomial in the number of cliques.

3.5 Finding Hypercliques

We have developed and implemented an algorithm called HYPERCLIQUE that finds cliques in k-uniform hypergraphs and, while NP-complete, shows satisfactory performance for relatively sparse graphs with few cliques. A direct extension of the Bron/Kerbosch-algorithm is not possible, since it makes use of a property of such graphs that does not extend to k-hypergraphs. In contrast, the HYPERCLIQUE algorithm mainly relies on the fact stated in Corollary 3 to find cliques.

Corollary 3 *Let C be a clique in a graph $G = (V, E)$. Let v be a node with $v \notin C$ and $\forall v_i \in C : \exists \{v_i, v\} \in E$. Then, $C \cup \{v\}$ is a clique.*

The low complexity of the Bron/Kerbosch algorithm is due to the fact that cliques can be "grown" very efficiently by simply checking the *connected*-property of a new node to each node of a previously found sub-clique. That is, given a clique, adding a node to this clique that is connected to each clique element will create a larger clique. Given a candidate node, this property can be tested in a time linear in the number of nodes in the graph.

On the other hand, this is not true for k-hypergraphs with $k > 2$ as the concept of "connected" is more complex for k-hypergraphs. Growing hypercliques in the same way requires to test for the existence of an edge between the new node and any $k-1$-*subset* of the nodes of the existing clique, which is exponentially more expensive. A further complication arises from the fact that the Bron/Kerbosch algorithm keeps a counter for each node v tracking the number of nodes *not connected* to v (variable *nod* in the algorithm in Appendix A). Clearly, this "non-connectedness" property is more complex for k-hypergraphs. Therefore, a direct extension of the Bron/Kerbosch algorithm would lead to an algorithm with intractable complexity.

A naïve hyperclique-finding algorithm, which would enumerate all possible subsets of nodes in a graph and test each for the clique property, is also clearly not feasible, as it is exponential in the number of nodes in the graph.

However, it is possible to find a better algorithm. The following clique-criterion, which is a generalization of the same criterion for graphs, can be used to find cliques in k-hypergraphs.

Theorem 3 *A k-hypergraph G_k with n nodes is a hyperclique if and only if each of its nodes has a degree of $\binom{n-1}{k-1}$.*

Proof: We have to show that the condition is necessary and sufficient. First, we show necessity.

Recall that we define an edge in a k-hypergraph as a set of nodes. By Cor. 2, a fully connected k-hypergraph with n nodes has $\binom{n}{k}$ edges. (To see this, set $C = V$ in Cor. 2.) Each edge is a set of k nodes, and edges in a k-hypergraph are pairwise distinct (Def. 6). Since there are exactly $\binom{n}{k}$ subsets of size k over a set of size n, there is one edge for each possible combination of k nodes from n nodes, such that all nodes must be members of an equal number of edges, i.e., must have the same degree.

Now, note that all edges together contain a total of $\binom{n}{k} \cdot k$ (not necessarily distinct) nodes. Since there are only n different nodes, and all nodes have the same degree, each node must have a degree of $\binom{n}{k} \cdot k/n = \binom{n-1}{k-1}$.

For sufficiency, note that the existence of n nodes each with degree $d = \binom{n-1}{k-1}$ implies the existence of $e = n \cdot d$ edges. Each edge connects exactly k nodes. Since all edges are distinct, there must be at least e/k edges. Now $e/k = \binom{n-1}{k-1} \cdot n/k = \binom{n}{k}$, which is the condition for a hyperclique (by Corollary 2) **q.e.d.** $\square$

Example 3 *Consider the induced subgraph of the hypergraph in Fig. 2 which has only the nodes* $1, 2, 3, 4$. *Each node of that subgraph has degree* $\binom{4-1}{3-1} = 3$. *This subgraph is fully connected (since the nodes in each 3-subset of its four nodes are connected by a* 3*-hyperedge).*

The above proof subsumes the proof for the analogous clique-criterion on traditional cliques (i.e., 2-Hypercliques). We have now established a simple criterion for a hyperclique, in that a minimal degree is required for a node to be part of a hyperclique of a certain size (number of nodes). Thus, the degree of a node gives us an upper bound on the size of any clique in which this node is involved.

Note that it is not sufficient for a subset *of nodes in a graph to have a certain minimum degree in order to be a clique.* Theorem 3 applies only if the entire set of nodes of a graph forms a clique.

When storing a k-hypergraph $G_k = (V, E)$ simply as a list of its edges, determining the degree of a node takes time in $O(|E|)$ (if set membership can be tested in constant time), as opposed to the more efficient $O(|V|)$ for regular graphs stored as adjacency matrices. The number of edges in a k-hypergraph is in $O(|V|^k)$ (since $\binom{n}{k} \in O(n^k)$), such that storing the graph or determining its node degrees may not be feasible for large graphs. However, we will see that for our purpose, the size of graphs will usually be manageable, and in fact, in our implementation we used a list of edges as the storage principle.

3.5.1 Algorithm HYPERCLIQUE to Find Cliques in k-Uniform Hypergraphs

With the background given, we can now describe an algorithm to find hypercliques in k-hypergraphs. Algorithm HYPERCLIQUE uses several branch-and-bound strategies to reduce a graph to smaller subgraphs, while finding cliques. It quickly finds some cliques in relatively large graphs by removing edges from a graph that can no longer contribute to cliques. In some cases, this strategy finds all cliques, especially if the graph is sparse. However, the heuristics may fail on certain graphs, as explained below. Then, a recursive strategy is used that reduces the graph by removing certain edges and working on subgraphs. The algorithm will always find all cliques in a given hypergraph. The algorithm has exponential worst-time complexity but will finish quickly on a much larger class of graphs than the naïve algorithm.

Fig. 4 gives a summary of algorithm HYPERCLIQUE in pseudo code. In the following section, we explain the details of this algorithm. We proceed from bottom to top, first explaining the algorithm's main functionality, then introducing two search-space reducing steps to reduce the size of the hypergraph, and finally concluding with a description of the entire algorithm.

Overall Strategy of Algorithm HYPERCLIQUE The HYPERCLIQUE algorithm applies a number of branch-and-bound strategies to find cliques in less time than brute-force enumeration would allow. The basic algorithm will find some hypercliques very fast, and for relatively sparse graphs will find all hypercliques in a graph. Augmented by branch-and-bound strategies, algorithm HYPERCLIQUE then applies a recursive procedure that reduces the size of the input graph. In all cases, algorithm HYPERCLIQUE finds all hypercliques in a given k-hypergraph.

In order to explain the algorithm, we first make a number of observations about k-hypergraphs.

Lemma 1 *Consider a* k*-hypergraph* $G_k = (V, E)$ *and a set* V_k *of* k *nodes* $v_1, \ldots, v_k$ *from* G_k. *Consider another node* $v \in V \backslash V_k$. *If there is* any $(k-1)$*-subset* V_{k-1} *of* V_k *such that* $V_{k-1} \cup \{v\}$ *does not form an edge in* E*, then the set* $V_k \cup \{v\}$ *does not form a clique.*

This formalizes the concept of a "missing edge" between a new node and any existing node in the clique candidate for k-hypergraphs. See Corollary 3, p. 6, for the analogous property on regular graphs.

We define an auxiliary procedure on graphs that we will call *clique candidate generation.* Starting from an edge in a k-hypergraph, this procedure will generate one candidate set for a clique. For any given edge e_0, **generateCliqueCandidate** will return the union of the set of nodes in e_0 and the set of nodes that are connected by a k-hyperedge each with $k-1$ nodes in e_0 (this is a concept analogous to the concept of

ALGORITHM HYPERCLIQUE

INPUT:
k-Uniform Hypergraph $G_k = (V, E)$

OUTPUT:
Set *result*, containing all cliques in G_k

```
function findHypercliques(Graph G_k)
result ← ∅    //result is a variable global to this function
Set S    //a clique candidate
reducible ← false
do
    E* ← ∅    //edges that are element of more than one clique
    forall (e ∈ E)    //E is set of edges in G_k
        S ← generateCliqueCandidate(G_k, e)
        if (S is clique )    //Thm. 3
            if (S is not subset of any element of result)
                result ← result ∪ {S}
            reducible ← true
        else
            E* ← E* ∪ {e}
    if (¬reducible)
        do
            e ← a random edge from E
            S ← generateCliqueCandidate(G_k, e)
            G_1(V_1, E_1) ← subgraph of G_k induced by S
            G_2(V_2, E_2) ← (V, E\e)
        while (|V_1| ≠ |V|)    //end of do-while-loop
        result ← result ∪ findHypercliques(G_1)
        result ← result ∪ findHypercliques(G_2)
    G_k ← (V, E*)    //reduced graph G_k
while (G_k ≠ ∅ ∧ reducible)    //end of do-while-loop
```

Figure 4: Algorithm HYPERCLIQUE for Finding Cliques in a k-Uniform Hypergraph

connected in regular graphs). The intuition behind this procedure is that it either finds the (only) clique that contains edge e_0 or the union of all nodes of all cliques that contain edge e_0. The algorithm in Fig. 5 gives pseudo code for this procedure.

function generateCliqueCandidate(Graph $G_k(V,E)$, Edge e_0)
$S \leftarrow$ nodes of e_0 *//an edge is a set of nodes, thus S is a set*
forall $(v_1 \in V \backslash S)$
  $f \leftarrow$ **true** *//a flag*
  forall $(v_2 \in e_0)$
    if $((\{v_1\} \cup (e_0 \backslash \{v_2\})) \notin E)$
      $f \leftarrow$ **false**
  if (f)
    $S \leftarrow S \cup \{v_1\}$
return S

Figure 5: Clique Candidate Generation for k-Hypergraphs

Theorem 4 *Let $G_k = (V, E)$ be a k-hypergraph and $e_0 \in E$ an edge in G_k. Construct a set S as the output of procedure **generateCliqueCandidate(**G_k, e_0**)**, Fig. 5. Denote by $C_{max}(G_k)$ the set of maximal cliques in graph G_k. If S is a clique, then*

$$C_{max}(G_k^*) \supseteq C_{max}(G_k) \backslash \{S\},$$

with $G_k^* = (V, E\backslash\{e_0\})$. That is, the set of maximal cliques in the graph obtained by removing edge e from G_k is a superset of the set of maximal cliques in G_k without clique S. In other words, if S is a clique, then it is the only *maximal clique in G_k that contains edge e_0. The removal of edge e_0 may introduce smaller cliques in graph G_K^* that are sub-cliques of S and have to be removed by algorithm* HYPERCLIQUE.

Proof: The theorem states that edge e_0 is an edge *only* in clique S and not in any other clique from $C_{max}(G_k)$. Consider procedure **generateCliqueCandidate**(G_k, e_0). It finds all nodes "connected" to all nodes in edge e_0. No node that is not connected to all nodes in e_0 can be a member of a clique that contains edge e_0. Therefore, the set S this procedure returns must be a superset of the union of nodes in *all* cliques of which edge e_0 is a member. Now if S is a clique, which is a premise in our theorem, and S contains all nodes that could possibly be members of cliques containing e_0, S must also be the *only* clique that contains edge e_0. □

With this theorem, we can state that any set of nodes returned by procedure **generateCliqueCandidate(**G_k, e**)** that induces a complete graph is a maximal clique. Furthermore, retaining only the edges E^* for a graph G_k^* (see Fig. 4) will produce a correct result for the clique finding problem in graph G_k, since all cliques that contain edges *not* in E^* have already been found.

Procedure **generateCliqueCandidate(**G_k, e**)** is used to check which edges of G_k can be safely removed (since they only contribute to one clique each, which is returned by the corresponding call to this function). For our algorithm HYPERCLIQUE (Fig. 4), removing edges in this way is the major strategy to achieve better performance. With the brute-force procedure and the main heuristic (Thm. 4) in place, we can now explain the algorithm itself.

The algorithm HYPERCLIQUE starts by generating one clique candidate for each edge in the input graph G_k and collects all those clique candidates that pass a test for the clique property in a set *result* (**forall**-loop in Fig. 4). The following example shows this procedure.

Example 4 *In order to illustrate this procedure, consider Figure 6, which is broken down into four stages. Stage 1 shows a 3-hypergraph G_k (this is the same graph as in Fig. 2). Procedure **generateCliqueCandidate** is now called on all $e \in E$. Stage 2 shows edge $e = \{1, 2, 3\}$ being selected. Initially, set S in function **generateCliqueCandidate** (Fig. 5) contains nodes 1,2, and 3, which are the nodes in e. Node 4 is added to S since edges $\{1, 2, 4\}$, $\{2, 3, 4\}$, and $\{1, 3, 4\}$ all exist. Node 5 is not added, since, for example,*

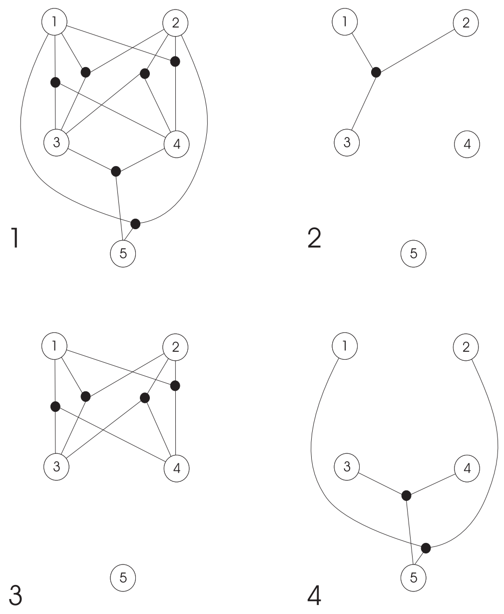

Figure 6: Phase 1 of Algorithm HYPERCLIQUES: Growing hypercliques

there is no edge $\{1, 3, 5\}$. *That is, function* ***generateCliqueCandidate*** *returns the set* $S = \{1, 2, 3, 4\}$ *to algorithm HYPERCLIQUE (Fig. 4). This set is now tested for the clique property, and tests true (Stage 3). That means that* S *is a maximal clique (by the definition of function* ***generateCliqueCandidates****) and furthermore that edge* $\{1, 2, 3\}$ *cannot be part of another clique that includes nodes other than 1,2,3, and 4 (by Thm. 4). Thus,* S *is included in set result. This process is now repeated for all other edges in* E*. Clique* $\{1, 2, 3, 4\}$ *is found three more times (and thus not added to the final result). Subsequent calls to procedure* ***generateCliqueCandidate*** *on edges* $\{1, 2, 5\}$ *and* $\{3, 4, 5\}$*, respectively (Stage 4), find cliques* $\{1, 2, 5\}$ *and* $\{3, 4, 5\}$*, respectively. Since cliques have been found for* all *edges in* E*, and each edge is member of exactly one clique (by Thm. 4), all cliques must have been found. Note that in algorithm HYPERCLIQUE, set* E^* *is now empty and the graph* G_k *is reduced to an empty graph, terminating the algorithm.*

However, this procedure might fail. It may occur that the function **generateCliqueCandidate**(G_k, e) returns a set that is *not* a clique. In this case, edge e must be a member of two or more maximal cliques that cannot be merged into one clique. Therefore, a new graph $G_k = (V, E^*)$ is constructed that contains all edges for which no cliques were generated before. In most cases, repeating our algorithm on G_k, we will find more cliques, and eventually reduce our graph to an empty set. We give an example of that case.

Example 5 *For an example, refer to Fig. 7. The graph shown is a 2-hypergraph, as it is difficult to visualize the problem for higher-rank graphs. Cliques in that graph are* $\{0, 1, 2\}$, $\{1, 2, 4\}$, $\{1, 3, 4\}$, *and* $\{2, 4, 5\}$*. The three edges shown by bold lines (*$\{1, 2\}$, $\{2, 4\}$, $\{1, 4\}$*) are all members of more than one maximal clique, while all other edges are members of only one clique each. Function* ***generateCliqueCandidate****, used on any of those three edges, would return set of nodes that does not have a clique property, such that we would miss clique* $\{1, 2, 4\}$ *in the solution. We do find each of the other cliques when considering one of the thinly drawn edges in the figure. Thus, we have to reduce this graph to include only those edges* E^* *for which no clique was found in the first iteration (those are exactly the edges drawn in bold in the figure). Then, we call our algorithm again with only this subgraph* $G_k = (V, E^*)$*. Again, by Thm. 4, we will not miss any cliques when using this procedure. Some sub-cliques of previously discovered cliques may be found, which have to be removed from the solution (however not in this example). In this example, we would find all cliques in this graph within two loops of algorithm HYPERCLIQUE: cliques* $\{0, 1, 2\}$, $\{1, 3, 4\}$, $\{2, 4, 5\}$ *in the first run and clique* $\{1, 2, 4\}$ *in the second.*

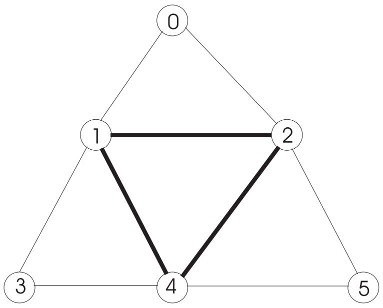

Figure 7: Phase 2 of Algorithm HYPERCLIQUES: Reducible Graphs

In certain cases, this procedure might fail, namely when set $E^* = E$ (we will call such a graph *irreducible*). Algorithm HYPERCLIQUE in Fig. 4 uses a flag *reducible* to keep track of that property. That situation occurs when *each edge* in a graph is part of more than one maximal clique, i.e., when function **generateCliqueCandidate** never returned a clique for any of the edges in E. For example, the 9-node Moon-Moser-Graph (see Sec. 3.4) has this property. Each edge in that graph is a member of three cliques (Fig. 8).

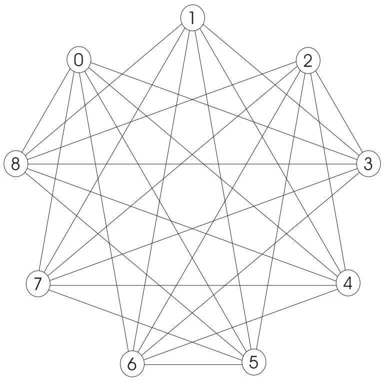

Figure 8: An Irreducible Graph.

In this case, we apply the following strategy. From the irreducible graph $G = (V, E)$, we take a random edge $e \in E$ and call procedure **generateCliqueCandidate** on e. This will return a set S that is not a clique, but the nodes of all cliques that e is a member of will be elements of S. Now, we *split* G into two graphs: graph G_e which is the subgraph of G induced by S, and graph $G_{\bar{e}}$, which is identical to G, except that it does not contain edge e.[2] We then call algorithm HYPERCLIQUES (Fig. 4) on G_e and $G_{\bar{e}}$, separately. The intuition behind this strategy is that removing edge e from the graph G will make the graph reducible. In order to still generate all cliques, we need to consider graph G_e separately. This heuristic leads to an increase in complexity that can be severe for dense graphs. However, since the input graphs in our IND-finding problem are quite sparse (as we find in our Experiments in Section 5), the algorithm's performance is sufficient for our purpose.

Example 6 *Refer to Fig. 9 for an example of the split procedure. The graph from Fig. 8 is being split at edge* $\{0, 8\}$ *into two graphs* G_e *and* $G_{\bar{e}}$. *Note that graph* $G_{\bar{e}}$ *in Fig. 9 is identical to the graph in Fig. 8, except that edge* $\{0, 8\}$ *is missing.*

In very large hypergraphs, a final problem may occur that we call *strong irreducibility*. A graph G may not be reducible such that it can be split, *and* the graph G_e split off from G may actually be identical to G.

[2]This is not a partition as some edges exist in both G_e and $G_{\bar{e}}$.

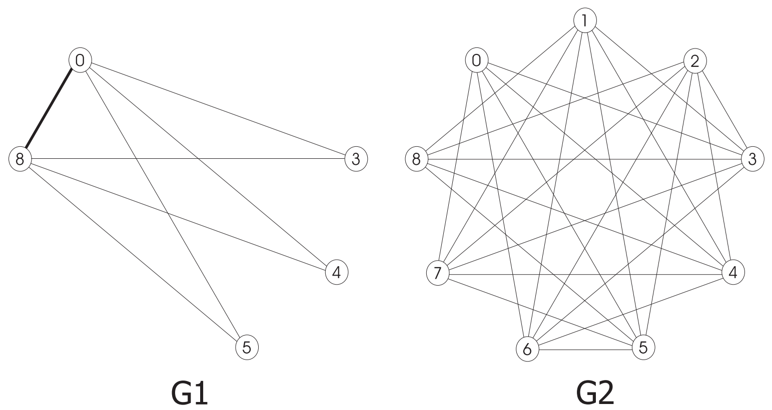

Figure 9: Splitting an Irreducible Graph.

In this case, infinite recursion occurs. We detect such cases and repeat the split with a different edge from G (inner do-while-loop in Fig. 4). This loop will eventually terminate since the graph cannot be strongly irreducible for every one of its edges. If this were the case, the graph itself would have to correspond to a hyperclique, and the inner do-while-loop would not have been reached.

It is clear that for most graphs, there will be a significant performance benefit in applying algorithm HYPERCLIQUE rather than the simple brute-force algorithm, since a smaller number of cliques (compared to the brute-force algorithm) actually has to be considered. In most cases, the size of the graph is quickly reduced by removing edges, which reduces the problem size. Our experiments (Chapter 5) have shown that algorithm HYPERCLIQUE finishes in a short time in all cases, even when algorithm FIND_HYPERCLIQUES_BRUTE_FORCE takes a prohibitively long time or fails to finish altogether.

4 Algorithm FIND_2 to Find Inclusion Dependencies

We now present the algorithm FIND_2 which applies clique-finding techniques to find inclusion dependencies. FIND_2 takes as input two relations R and S, with k_R and k_S attributes, respectively and returns a generating set of inclusion dependencies between attributes from R and S.

4.1 Overview

We first establish the relationship between hypercliques and the generating set of IND that we are trying to find. Intuitively, Thm. 5 shows that a clique-finding algorithm is a sensible approach to finding maximal INDs between relations. More specifically, the theorem states that the INDs generated through a clique-finding algorithm are a superset of the generating set $\mathcal{G}(\Sigma)$ and are thus a starting point for a complete and fast solution of the IND-finding problem.

Theorem 5 *Consider the IND finding problem between relations R, with attribute set A, and S, with attribute set B, with solution $\mathcal{G}(\Sigma)$ (i.e., generating set of valid INDs). Let V be the set of unary valid INDs between R and S. Let E_k, with $1 < k \leq \min(|A|, |B|)$, be the set of k-ary valid INDs between R and S. Recall that the elements of E_k can then be seen as edges in a k-Hypergraph $G_k(V, E_k)$, by Thm. 2.*

Now consider the set C of all maximal cliques in the k-hypergraph G_k, obtained by a (hyper)clique-finding algorithm. The following properties hold for any $c \in C$:

1. *If the IND σ_c corresponding to c is valid, it is part of the generating set $\mathcal{G}(\Sigma)$ of INDs between R and S.*
2. *If σ_c is* not *valid, some of its subsets are part of $\mathcal{G}(\Sigma)$.*
3. *Furthermore,* all *elements $\sigma \in \mathcal{G}(\Sigma)$ are subsets of or equal to some σ_c as above.*

Proof: By Thm. 2, a valid IND implies a k-hyperclique in a k-hypergraph G_k constructed for this IND-finding problem. Also, a correct clique finding algorithm returns a set of maximal cliques. Property (1) must hold since if there was an IND larger than σ_c, a clique corresponding to that larger IND would have been found. Property (2) is true since we assumed valid unary and k-ary INDs to make up graph G_k. If σ_c is *not* valid but its unary and binary sub-INDs are, then some sub-INDs of σ_c must be part of $\mathcal{G}(\Sigma)$. Property (3) holds since *any* IND implies some (not necessarily maximal) complete subgraph of G_k, and by the definition of a clique, all such complete subgraphs are subsumed by the set of cliques found in G_k. □

Algorithm FIND_2 in pseudo code is given in Fig. 10. It proceeds in three steps: finding unary and binary INDs, finding higher-arity IND candidates from those, finding additional IND candidates. In the following sections, we discuss these three phases of the FIND_2 algorithm and explain them, as well as the overall algorithm, in detail.

01 : Set $V \leftarrow$ generateValidUnaryINDs(R, S)
02 : Set $E \leftarrow$ generateValidBinaryINDs(R, S, V)
03 : Graph $G_2 \leftarrow (V, E)$
04 : Set $I \leftarrow$ generateCliquesAndCheckAsINDs(G_2)
05 : Set $result \leftarrow \{c \in I \wedge |c| = 1\}$
06 : **for** $m \leftarrow 3 \ldots k_S$
07 : KHypergraph $G_m \leftarrow (V, \emptyset)$
08 : Set $C_{tmp} \leftarrow \emptyset$
09 : **forall** $(c \in I)$
10 : **if** $(c \text{ is valid} \wedge |c| \geq (m-1))$
$result \leftarrow result \cup c$
11 : **if** $(c \text{ is invalid} \wedge |c| \geq m)$
$C_{tmp} \leftarrow C_{tmp} \cup c$
12 : $E_m \leftarrow$ genKAryINDsFromCliques(m, C_{tmp})
13 : **if** $(E_m = \emptyset)$ **return** *result*
14 : $result \leftarrow result \cup$ genSubINDs$(m, E_m, result)$
15 : Graph $G_m \leftarrow (V, \text{validINDs}(E_m))$
16 : $I \leftarrow$ generateCliquesAndCheckAsINDs(G_m)
17 : **return** *result*

Figure 10: Algorithm FIND_2. Functions used are explained in Sec. 4.5.

4.2 Finding Unary and Binary INDs

In Fig. 10, this phase corresponds to lines number 01 and 02. As explained in Sec. 3.2, there is a total of $I_1(k_R, k_S) = k_R \cdot k_S$ unary (not necessarily valid) INDs between R and S. To establish the set of nodes upon which all subsequent k-hypergraphs will be built, all those INDs have to be validated. This requires a generation of all $k_R \cdot k_S$ unary INDs and their subsequent testing. Methods for efficient testing are covered in [Koe01]. Bell and Brockhausen [BB95] present a slightly improved algorithm for the discovery of unary INDs which relies mainly on finding maximal and minimal attributes values for numeric domains, as well as on the availability of foreign key constraints in a database. If foreign keys are available, a number of unary database queries can be saved by their approach. We consider those assumptions to be quite strong given the performance benefits of the algorithm, such that we will not discuss this approach here further. Much more widely applicable heuristics that do no rely on outside constraints but solely on the values in an attribute (as well as its name and domain) are briefly mentioned in Section 4.8 and detailed out in [Koe01].

Example 7 *In our running example (Fig. 1), the valid unary INDs found are* $\sigma_1 = R[A] \subseteq S[A]$, $\sigma_2 = R[B] \subseteq S[B]$, $\sigma_3 = R[C] \subseteq S[C]$, $\sigma_4 = R[D] \subseteq S[D]$, $\sigma_5 = R[E] \subseteq S[E]$, $\sigma_6 = R[F] \subseteq S[F]$, $\sigma_7 = R[G] \subseteq S[G]$.

Once the valid unary INDs are established, we create from them a set of nodes V which will be used for all graphs G_k (by simply creating a node for each valid unary IND). We can now proceed to generate candidates for binary INDs. In principle, there are $\binom{k_R}{2} \cdot \binom{k_S}{2} \cdot 2$ such INDs. While this number can be quite large for relations with dozens or hundreds of attributes, many of those INDs do not have to be generated or tested since they would imply invalid unary INDs. In general, the number of binary INDs is bounded by a function quadratic in the number of unary INDs.

Also, by our definition of INDs (Def. 1), we do not allow duplicate attribute names on either side of an IND. Thus, when several unary INDs hold for the same attribute, not all combinations of their attributes have to be tested as binary INDs (e.g., $R[A] \subseteq S[B]$ and $R[A] \subseteq S[C]$ could be valid at the same time, but $R[A, A] \subseteq S[B, C]$ would not be tested).

After finding unary and binary INDs, we can then construct graph G_2 as defined in Fig. 3 (Line 03 in FIND_2).

Example 8 *The binary INDs valid between relations R and S in Fig. 1 are: $R[A, B] \subseteq S[A, B]$, $R[A, C] \subseteq S[A, C]$, $R[A, D] \subseteq S[A, D]$, $R[A, E] \subseteq S[A, E]$, $R[B, C] \subseteq S[B, C]$, $R[B, D] \subseteq S[B, D]$, $R[B, E] \subseteq S[B, E]$, $R[C, D] \subseteq S[C, D]$, $R[C, E] \subseteq S[C, E]$, $R[D, E] \subseteq S[D, E]$, $R[D, F] \subseteq S[D, F]$, $R[D, G] \subseteq S[D, G]$, $R[E, F] \subseteq S[E, F]$, $R[E, G] \subseteq S[E, G]$, $R[F, G] \subseteq S[F, G]$*

Graph G_2 is depicted in Fig. 11. The nodes represent unary INDs (labeled with index numbers in the order of the unary IND from Example 7) and the edges represent the binary INDs listed above.G_2 has two cliques, labeled c_1 and c_2 in the Figure.

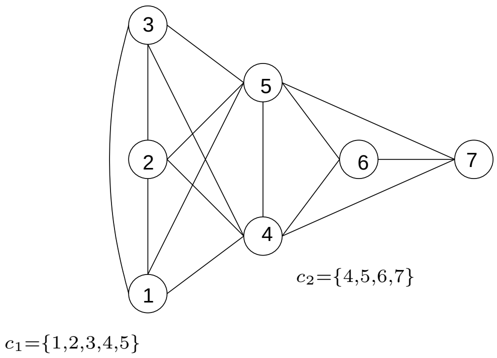

Figure 11: A Graph G_2 constructed by Algorithm FIND_2.

4.3 Generating Candidates for Higher-Arity INDs

We now run a clique-finding algorithm on the graph G_2 (Line 04 in FIND_2). The algorithm generates a set I (see Fig. 10), containing all maximal cliques in G_2. We denote the number of nodes in each clique $c \in I$ by k_c. Any clique with $k_c = 1$ (single unconnected nodes in G_2) will be part of the solution and is thus added to the variable *result* of FIND_2 (Line 05).

To understand the remainder of the algorithm, consider a collection of k-hypergraphs with $k = 3, \ldots, \max(k_c)$ (i.e., $G_3, \ldots, G_{\max(k_c)}$) obtained by storing each k-subset of each clique $c \in I$ (with $k \leq k_c$) as an edge in G_k. Together with the mapping from INDs to k-hypergraphs from Section 3.3, we observe the following property.

Theorem 6 *Let E_k be a k-set of nodes from V. If the edge e_k whose set of nodes is exactly E_k does* not *exist in G_k, the IND σ_k corresponding to e_k cannot be valid.*

Proof: (by contradiction) Assume a valid k-ary INDs that does *not* correspond to an edge in the k-hypergraph G_k. The validity of σ_k implies the (existence and) validity of $\binom{k}{2}$ binary INDs that are edges in G_2. By Theorem 2 (with $m = 2$) those binary INDs form a clique c.

However, any correct clique finding algorithm guarantees that *all* cliques in G_2 are found. Furthermore, any subset of a clique is also a clique. Thus, we would have found a subset c_k of some clique c that corresponds to σ_k and stored it as an edge in the appropriate k-hypergraph. This contradicts the above assumption. □

However, the converse of the above proposition is not true. The existence of an edge in G_k does *not* imply the existence of a k-ary IND in the database. To see why, refer to Fig. 11. In that example, a clique-finding algorithm would find a clique $c_2 = \{4, 5, 6, 7\}$, but the corresponding IND $R[D, E, F, G] \subseteq S[D, E, F, G]$ is not valid in the database in Fig. 1. Therefore generating graph G_2 is not enough. Rather, we must also test INDs that are implied by cliques, to verify whether they are indeed valid INDs or not (see Property (2) in Thm. 5).

Making use of the property described in Theorem 6, the maximum number of INDs to consider for testing is bounded by the *number of cliques* generated. This is much smaller than the total number of INDs. For a real-world problem, the number of cliques is likely to be manageable (see our experiments in Sec. 5 and [Koe01]). Hence, this results in significant performance benefits of the FIND_2 algorithm over the naïve algorithm. We also use heuristics to limit the number of cliques when too many (spurious) cliques do occur, as mentioned in Sec. 4.8.

4.4 Determining Higher-Arity INDs

So far, only unary and binary members of the generating set $\mathcal{G}(\Sigma)$ have been generated. We now develop an algorithm based on the idea of Thm. 5. A first step in finding higher arity-INDs is the checking of all IND candidates (cliques) generated by the previous steps. INDs that are valid will become part of the solution. However, if an IND generated by the clique-finding algorithm tests invalid, nothing can be said about the validity of any other INDs. IND-finding then proceeds as shown in Fig. 12, which is a less formal representation of steps 06–16 in Fig. 10.

1. Set $m = 3$. Let I be the set of cliques in G_2 created in lines 01–05 in FIND_2.
2. Test all cliques in the set I as INDs for their validity and collect *invalid* INDs in a set C_{tmp}. All *valid* INDs are added to the set of solutions. Now, for each $\sigma \in C_{tmp}$, add to a set E_m all m-ary INDs implied by σ. (Lines 09–12 in FIND_2, Fig. 10)
3. Test all INDs $\sigma \in C_{tmp}$ against the database as they are generated and store the valid ones as edges in an m-Hypergraph G_m (Line 15).
4. Run algorithm $\mathsf{HYPERCLIQUE}$ to find m-hypercliques in G_m (Line 16) and store the result in the set I to be used in the next iteration of the loop. If no cliques with more than m nodes (i.e., nontrivial cliques) are found, FIND_2 will terminate in the next loop and return the solution.
5. Increment m and repeat from step 2.

Figure 12: Generating Higher-Arity Members of the Generating Set $\mathcal{G}(\Sigma)$ in Algorithm FIND_2.

After iterating through the main loop and terminating, the solution set contains all valid INDs that have been found as cliques or hypercliques of lower-arity INDs. However, some maximal INDs may have been missed in this process. Consider set I as generated by algorithm FIND_2 (Line 16). For a given m, the set I (Figure 10) contains cliques that were found from a $(m-1)$-hypergraph. All those cliques correspond to INDs in the database, but not all those INDs are necessarily valid. In Fig. 13 this situation is depicted for $m = 3$, where clique $c_2 = \{4, 5, 6, 7\}$ is not valid.

If an IND σ thus found is invalid, it is still possible that some of its implied sub-INDs are valid. We will find most of those in subsequent hypergraphs (for higher m). However, some sub-INDs of σ with arity $m-1$ could be maximal, i.e., part of the solution. The FIND_2 algorithm as described so far would not find those. For example, in Fig. 13, three implied 3-ary INDs of clique c_2 (labeled 456, 457, and 567) are invalid. Therefore, some of their implied 2-ary INDs (which we know to be valid) are maximal and thus part of the generating set. Therefore, we need to generate all maximal sub-INDs σ_m of the invalid INDs in I.

For the given m, we thus form all implied m-ary INDs of all invalid INDs in I (i.e., all invalid INDs implied by any clique) and test each one for validity. We observe the following properties of any such implied IND σ_m:

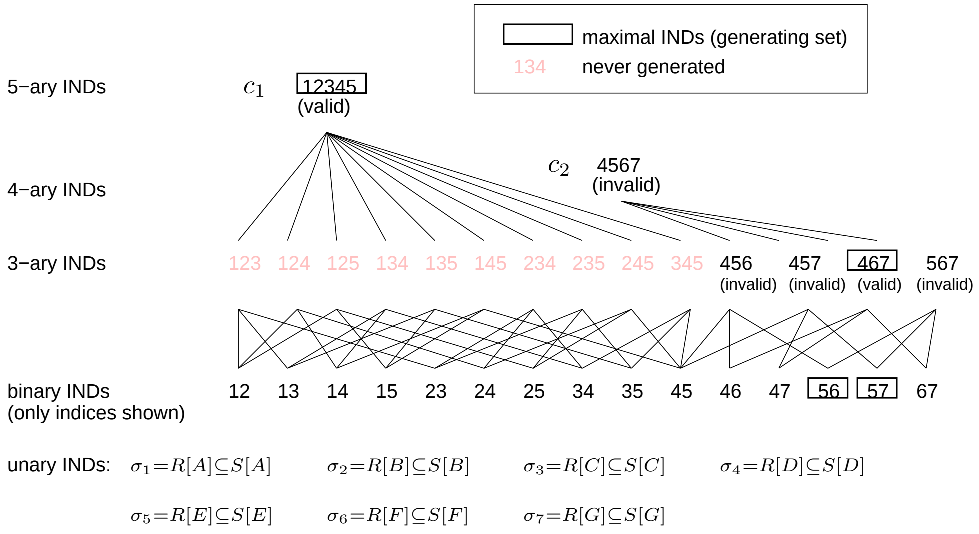

Figure 13: An Example for the Complete Algorithm FIND_2.

1. If σ_m itself is valid, it is a (not necessarily proper) subset of an IND not yet discovered (and vice-versa it cannot be a subset of any undiscovered IND if it is invalid).

2. If σ_m is invalid, some or all of its $(m-1)$-ary sub-INDs may be maximal valid INDs and thus part of the generating set of INDs.

We further observe the following properties of any $(m-1)$-ary sub-IND σ_{m-1} generated from any σ_m:

1. σ_{m-1} is valid in the database (since it is a hyperedge in the $(m-1)$-Hypergraph G_{m-1} which has previously been formed).

2. If σ_m is valid, σ_{m-1} is not maximal.

3. If σ_m is invalid and σ_{m-1} is *not* a subset of any other valid σ of arity m as well as not a subset of any larger valid IND already part of the result, σ_{m-1} must be a maximal IND.

4. As noted above (after Def. 7), no k-hypergraph can contain cliques of sizes $2\ldots(k-1)$. Therefore, as m will grow during subsequent executions of the outer for-loop in FIND_2, no further cliques of size $m-1$, and thus INDs of size $m-1$, can be discovered after this step. That means that the procedure described here will finally find all maximal INDs of arity m and smaller.

We now give a concrete example of the procedure just explained for our running example problem.

Example 9 *In Fig. 13, algorithm* FIND_2 *found a clique* $\{4,5,6,7\}$ *in graph* G_2*. For* $k=3$*, this clique is part of set I (see Fig. 10). As the IND implied by this clique (*$R[D,E,F,G] \subseteq S[D,E,F,G]$*) is invalid in the database (Fig. 1), the four implied* 3*-ary INDs have to be checked. One of them (*$R[D,F,G] \subseteq S[D,F,G]$*, labeled* 467*) is valid in the database, the other three are not. Thus, INDs* $R[D,F] \subseteq S[D,F]$*,* $R[D,G] \subseteq S[D,G]$*, and* $R[F,G] \subseteq S[F,G]$ *(labeled* 46*,* 47*, and* 67*, respectively) are not maximal, while the remaining binary INDs implied by the invalid INDs may be valid (those are* $R[D,E] \subseteq S[D,E]$*,* $R[E,F] \subseteq S[E,F]$*,* $R[E,G] \subseteq S[E,G]$*, labeled* 45*,* 56*,* 57*, respectively).* $R[D,E] \subseteq S[D,E]$ *(labeled* 45*) is implied by the valid* $R[A,B,C,D,E] \subseteq S[A,B,C,D,E]$ *(labeled* 12345*), which is already in the solution. Thus it is not maximal. The remaining two INDs (*$R[E,F] \subseteq S[E,F]$*,* $R[E,G] \subseteq S[E,G]$*, labeled* 56*,* 57*) are maximal and are thus included in the solution. Note that there is no need to check any binary IND for validity anymore, since all binary INDs that are considered in this step are edges in* G_2 *(and must thus be valid).*

These properties suggest a method to generate all maximal INDs that have not been discovered by clique-finding. This method is stated in the next section, in function **genSubINDs**.

4.5 Subroutines of FIND_2

The following subroutines are used in algorithm FIND_2.

generateValidUnaryINDs(R,S) and generateValidBinaryINDs(R,S,V). These functions generate all unary and binary INDs, respectively, and check them against the database. They return the sets V and E of the unary and binary INDs valid in the database.

generateCliquesAndCheckAsINDs(G_k). This function accepts a k-hypergraph G_k and then calls a (hyper)clique finding algorithm for G_k. The function then tests each generated (hyper)clique's implied IND against the database. It returns a set of *all those INDs with more than k nodes* (i.e., at most one IND for each clique discovered), regardless of their validity, but *marks* each IND as valid or invalid according to its state in the database.

genKAryINDsFromCliques(k,E). This function accepts a number k and a set of INDs E. Input set E is assumed to be composed of invalid INDs and to correspond to cliques found by a clique finding algorithm on a $(k-1)$-Hypergraph. This function now generates all k-ary INDs implied by each IND in E, tests each one of them, and returns the union of those INDs for all elements of E. Similarly to function **generateCliquesAndCheckAsINDs()**, both valid and invalid INDs are returned, and each IND is marked as valid or invalid according to its state in the database.

genSubINDs(k,E,*result*). This function accepts a number k, a set of (valid or invalid) INDs E, and set *result*, which is the set of INDs already included in the solution earlier (lines 05 and 10 in Fig. 10). The function now generates all INDs σ that satisfy the following three conditions (derived from the conditions in Sec. 4.4):

1. σ is a k-subset of an *invalid* IND from E
2. σ is *not* a subset of *any valid* IND from E
3. σ is *not* a subset of any IND already in *result*.

The function returns the set of all σ that meet the above conditions. Note that subset testing can be done very efficiently when sets are stored as bit fields (by simple bit operations).

validINDs(E). Given a set of INDs marked as valid or invalid, this function simply returns the valid INDs from E.

Also note that the set *result* of solutions is built through set-union operations (Lines 10 and 14 in Fig. 10). That will prevent INDs that have been found multiple times from being added to the solution more than once.

With these observations, we can state that algorithm FIND_2 generates all maximal INDs that hold between two given relations. We will give a correctness argument in Section 4.6. We now give an example for the complete algorithm FIND_2 by way of our running example.

Example 10 FIND_2 *first finds unary and binary inclusion dependencies (Fig. 13). For INDs of arity higher than 1, we only show the indices of unary INDs that they imply. The algorithm then proceeds to find cliques from the discovered valid binary (and unary) INDs, from which we have formed a graph (Thm. 2). The cliques found in this graph are* $c_1 = (1,2,3,4,5)$ *and* $c_2 = (4,5,6,7)$. *A test against the database of those two cliques yields that the IND induced by* c_1 *is valid while the IND induced by* c_2 *is false (invalid). Now, algorithm* FIND_2 *proceeds to generate four 3-ary INDs from the invalid IND induced by* c_2. *Those INDs (*$(4,5,6)$, $(4,5,7)$, $(4,6,7)$, *and* $(5,6,7)$*) are tested against the database and only* $(4,6,7)$ *is found valid. That implies that any 2-ary IND induced by* $(4,6,7)$ *is not maximal while any 2-ary IND implied by the other three INDs may be maximal. In fact, two of them are (*$(5,6)$ *and* $(5,7)$*), while* $(4,5)$ *is subsumed by* $(1,2,3,4,5)$ *and is therefore not maximal. The only edge in the newly formed* 3*-Hypergraph* G_3 *is now* $(4,6,7)$*, such that the only* clique *in that graph is also* $(4,6,7)$*, which is then the last remaining maximal IND. This implies that the generating set for all INDs between R and S is* $\{(1,2,3,4,5),\ (4,6,7),\ (5,6),\ (5,7)\}$.

4.6 Correctness of Algorithm FIND_2

To show the correctness of algorithm FIND_2, we first assume that the functions called by it are correct. All functions called, except for *genSubINDs* whose soundness we have motivated earlier, are fairly straightforward and easy to prove so that their correctness will not be shown here.

The clique-generating function (*generateCliquesAndCheckAsINDs*) uses the BRON/KERBOSCH-algorithm [BK73] for $k = 2$ and algorithm HYPERCLIQUE for $k > 2$. Both algorithms are also assumed to run correctly (i.e., find all maximal cliques in the given graph or k-hypergraph).

It remains to show that the algorithm FIND_2 finds the generating set (Def. 5) of INDs between two relations R and S. We have to show that the result set is complete, i.e., that all INDs between R and S can in fact be derived from the INDs in the set *result* and that the set *result* is minimal with respect to this property.

First, we show that all INDs are in fact found. Consider the graph G_2 whose nodes are valid unary INDs and whose edges are valid binary INDs between the relations in question. By Thm. 5, all elements $\sigma \in \mathcal{G}(\Sigma)$ of the generating set, i.e., all solutions to the IND-finding problem, are either elements or subsets of the set of cliques found by the clique algorithm for $k = 2$. We consider the INDs that have *not* been found directly as cliques in G_2 (property (2) in Thm. 5). Let us denote this set of not-yet-discovered INDs by $\Sigma_2 \subseteq \mathcal{G}(\Sigma)$. Now there are two cases:

1. INDs $\sigma_2 \in \Sigma_2$ with $|\sigma_2| = 2$ are found by the function *genSubINDs*, line (14) in algorithm FIND_2 (Fig. 10)

2. INDs $\sigma_2 \in \Sigma_2$ with $|\sigma_2| > 2$ are not found by function *genSubINDs*. Rather, 3-ary INDs are created as subsets of those cliques in G_2 that correspond to invalid INDs, and used as edges in a 3-hypergraph G_3. This set of 3-ary INDs is necessarily equal to or a superset of the set of 3-ary sub-INDs of elements of Σ_2 (i.e., the set of as-yet undiscovered INDs, by Thm. 5).

We now show that all INDs are found. For $3 \leq i \leq \min(|R|, |S|)$, the clique-finding algorithm is called for G_i. By Thm. 5, some new INDs may be found from (hyper)cliques. However, function *genSubINDs* does generate *all i-ary INDs*, analogous to the case of $i = 2$ described above. For each i, a new set Σ_i with the not-yet-discovered INDs is formed as above. Since all INDs must have arity lower than or equal to $\min(|R|, |S|)$, and all (missing) i-ary INDs are found at step i, Σ_i must be empty after calling function *genSubINDs* for $i = \min(|R|, |S|)$. That means, all INDs must have been found.

It remains to show minimality. There are three places in the algorithm FIND_2 where discovered INDs are added to the result, in lines 5, 10, and 14 (Fig. 10). First, the result is initialized with the single-node cliques. Those correspond to nodes in the initial graph G_2 which have degree 0, such that their implied INDs cannot be part of any higher-arity IND (i.e., they are maximal). In lines 5 and 10, we add INDs that have been generated by a clique algorithm, which must be maximal (since clique algorithms find *maximal* cliques). INDs added in line 14 are maximal as shown in the previous section. Since only *maximal* INDs are added to the result, the result set is a generating set, as long as all INDs are included. We have proved that all INDs are included. Thus the set is indeed a generating set, and therefore minimal, by the definition of generating sets in Sec. 2. **q.e.d.**

4.7 Runtime of Algorithm FIND_2

In this section, we will briefly explore the complexity of algorithm FIND_2. We will show the complexity of each function called by it, and the complexity of the complete algorithm. Throughout the section, we will assume that IND-testing can be done in constant time. We will assume the variable names from algorithm FIND_2, Fig. 10.

The runtime of function *generateValidUnaryINDs* is in $O(k_R \cdot k_S)$ and returns a set V. Function *generateValidBinaryINDs* runs in $O(|V|^2)$ time, which in the worst case is bounded by $(k_R \cdot k_S)^2$.

Generating cliques and hypercliques is a more expensive process. The problem of finding all hypercliques in a k-hypergraph is a generalization of the clique-finding problem and thus also NP-hard. However, the problem is tractable in practice, since we expect a small number of cliques only, and graphs with few cliques can be searched for cliques efficiently.

As shown in [Koe01], the runtime of BRON/KERBOSCH has a (loose) upper bound of $O(|V|^4 \cdot |C|^2)$ where $|V|$ is the number of nodes in G and $|C|$ the number of cliques. Finding cliques in a hypergraph is not as cheap but can be done in $O(|V| \cdot k \cdot |E|^2)$ for a k-uniform hypergraph, as long as no hyperedge from $|E|$ is a member of more than one maximal clique. If hypergraphs become more dense and k increases, this assumption does not necessarily hold, but we found in our experiments that the algorithm always terminated for $k \leq 6$ and also used relatively sparse graphs only. For problems in which the number of cliques becomes too large, we apply the heuristics mentioned in Sec. 4.8.

Generating subsets (functions *genKAryINDsFromClique* and *genSubINDs* in algorithm FIND_2, Fig. 10) is simple to analyze. The runtime of the generating algorithm is bounded by the number of subsets to generate, i.e., by $\binom{|c|}{k}$ for a clique c in the case of *genKAryINDsFromClique* and $|E_k| \cdot \binom{k}{k-1}$ in the case of *genSubINDs*. The first number can become very large when k is not either very small or close to $|c|$. However, as mentioned above, we do not expect the value of k to grow very large. In our experiments, k did not grow beyond 5 or 6.

A useful general bound for the complexity of algorithm FIND_2 is difficult to give. Furthermore, giving *tight* bounds for the general case seems impossible due to the complexity of the problem. Therefore, and due to the fact that a subproblem of FIND_2 is NP-hard, we will not give a complexity bound but rather refer to our experiments. In cases when our original algorithm as shown in Fig. 10 runs too slowly, we restrict the search space by applying heuristics (Sec. 4.8).

4.8 Exact and Heuristic Testing of IND-Validity

The first four functions in the list in Sec. 4.5 all test INDs against the database. In the FIND_2 algorithm as described in this paper, we accomplish this by database queries. Faster and more comprehensive methods of checking INDs are explored in [Koe01]. In this paper, we concentrate on how to reduce the *number* of individual IND-checks, rather than on ways to speed up such checks, since the the possible speedups of IND testing (which is linear in the number of attributes and tuples in the database) cannot contribute much to the speedup of high-dimensional IND discovery. However, databases carry additional information about their data (such as data domains, database statistics, indices, attribute name ontologies), and such information can be used to avoid testing some INDs for any non-trivial real problem. That is, not all possible INDs that our system considers would have to be checked against the database, but rather many IND checks can be answered in very short time, for example by comparing the domains of corresponding attributes.

If meta-information is not sufficent to decide the validity of an IND, IND-testing can be accomplished in SQL through a simple `MINUS`-query. However, there is much room for improvement, in particular between attributes with many duplicate values. In [Koe01], we developed a number of heuristics that can improve the runtime of the FIND_2 algorithm significantly, without sacrificing too much of its accuracy. Those heuristics mainly rely on exploiting statistical properties of attributes, such as the number of distinct values and the attribute value distribution.

4.9 False Negatives and the Clique-Merging Heuristic

Consider a complete graph (i.e., a graph with all possible edges) $G = (V, E)$. The set of nodes V in G forms a clique. Now remove a single edge from E. Clearly, the clique property does no longer hold, but rather G will now contain at least two distinct maximal cliques. Those cliques are likely to have a substantial overlap (i.e., common set of nodes).

When using heuristics to test the validity of INDs in algorithm FIND_2, some INDs may test as invalid even though they should be considered valid for our algorithm. Thus, some edges (or even nodes) of any graph or hypergraph considered by FIND_2 may be missing. The clique finding algorithms used by FIND_2 will then no longer find cliques that correspond to the maximal INDs in the given problem, but rather find only smaller subsets of those cliques. However, those sub-cliques will often show substantial overlaps. Therefore, we use the following strategy:

> *When heuristics are used in* FIND_2 *that may produce false negative results (i.e., reporting non-accidental INDs as invalid), and we obtain several large, overlapping INDs from algorithm* FIND_2*, merge those INDs by finding the union of their nodes.*

Naturally, merging *all* INDs found by algorithm FIND_2 will in general not lead to a valid INDs, unless the (true) generating set of INDs actually contains only one IND. Therefore, we propose to only merge INDs of decreasing size, starting from the largest, until adding another IND to the result will no longer produce a valid IND.

Our experiments show that this heuristic is powerful enough to find large or maximal valid INDs even in cases when many underlying edges (small-arity INDs) are not found (Chapter 5). Of course, as only the *distinct value heuristic* and potentially the AVD heuristic produces such negative results, IND merging is not necessary when these two heuristics are either not used or when they did not reject any otherwise valid IND.

A similar merging strategy is used by Zaki in [Zak00] for the problem of association rule mining. More information on Zaki's work is given in the related work section (Chapter 6).

Ways to speed up the discovery of *unary* INDs have been described in the literature [dLP02, BB95]

5 Experimental Evaluation

5.1 Implementation

We implemented algorithm FIND_2 in Java over Oracle relational databases (using JDBC). The implementation is modular, such that different aspects of the algorithm can be tested separately. INDs are modeled as objects, which simplifies the code. Some implementation improvements could be made by representing INDs as simpler data structures, however that was not the focus of the prototype implementation. All heuristics and strategies described in this dissertation, with the exception of tuple-level sampling, are implemented and can be turned on or off in the prototype. The schemas of the test databases can either be queried or explicitly specified (if only subsets of the attributes in the source relations are to be tested).

Figure 14 gives an overview over the system architecture. The implementation has two major modules: the *IND-Generator* which generates INDs to be tested, and the *IND-Tester* which tests the IND in question through a database query and/or through applying heuristics. Those two modules roughly correspond to the FIND_2 and $\mathsf{CHECK}/\mathsf{CHECK_H}$ algorithms, respectively. At the end of the discovery process, the IND-Generator also assumes the task of reporting all INDs that tested as maximal and valid.

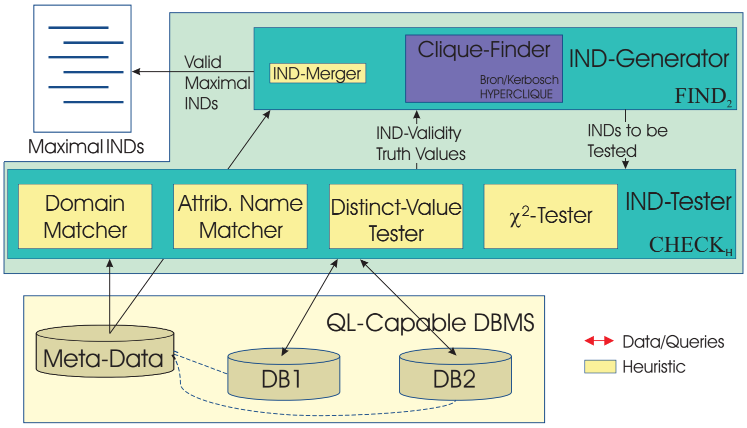

Figure 14: Overview of System Architecture

The implementation has a total of 6 parameters that can be "tuned". Five of those parameters restrict the sizes of certain data structures, as described in Sec. 4. In particular, those are: the maximal number of nodes in any graph (i.e., unary INDs), the maximal number of edges in graph G_2 (i.e., binary INDs), the maximal number of cliques tested as INDs at any one step, the maximal number of cliques *retained* for the next step, and the maximal number of INDs generated from any clique by function **generateINDsFromCliques**. The sixth parameter that can be set in the implementation affects the *distinct-values* heuristic and sets the minimum number of distinct values in an attribute or attribute set that is necessary for the distinct-value heuristic to *not* reject the IND as accidental. For Experiment 4, we additionally used "fuzzy" IND checking, as explained in the detailed description of that experiment.

We ran experiments on data sets obtained from the UC Irvine KDD Archive[3]. We report only on a small

[3]URL as of September 2001: `http://kdd.ics.uci.edu`

subset of our experiments here, while an extensive study can be found in [Koe01]. In order to "discover" INDs, we generated different subsets of existing relations from the KDD archive (relations with 5,000–100,000 tuples and 40–200 attributes each) and let our algorithm discover the inclusion between those tables.

5.2 Experiment 1: Comparison with Apriori-Based Strategies

In any Apriori-based algorithm, INDs would be found by discovering every valid IND from smallest to largest. One might prune and not generate *all* INDs, but all *valid* INDs will have to be generated and tested against the database. [dLP02] applies this strategy. As stated earlier, a valid k-ary IND implies $\binom{k}{m}$ valid INDs for all $2 \leq m < k$, such that the total number of implied valid INDs is 2^k. In Fig. 15 we plotted that number for different sized INDs against the number of DB queries actually performed by algorithm FIND_2 for data set CENSUS. Actual runtimes for that dataset are reported in Experiment 2. Note that the Y-axis is logarithmic, indicating that the Apriori-strategy is not feasible for INDs with arities higher than about 15, whereas our algorithm (not using heuristics to prune the search space) needs less than $6,000$ queries to discover the largest IND in our experiment.

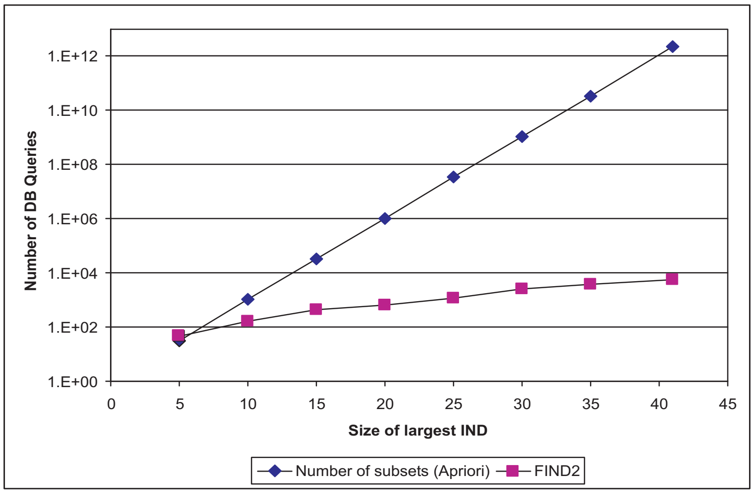

Figure 15: Number of DB Queries Needed to Discover a Large IND.

5.3 Experiment 2: Performance of Algorithm FIND_2

This experiment was conducted to assess the runtime of the algorithm and the quality of its output for a given data set, with and without the use of heuristics. For this experiment, we used a 5000-tuple random subset $\mathsf{CENSUS1}$ of data set CENSUS and a further random subset $\mathsf{CENSUS2}$ of 4500 tuples (90%) of $\mathsf{CENSUS1}$. We measured the performance of algorithm FIND_2 when discovering INDs between $\mathsf{CENSUS2}$ and $\mathsf{CENSUS1}$, both with and without heuristics. We used different arbitrary projections on the set CENSUS from KDD, which has 41 attributes. Figure 16 shows the runtime of algorithm FIND_2 with and without heuristics, for different size projections.

The experiment shows that algorithm FIND_2 finds large INDs in reasonable time, and furthermore demonstrates the large performance benefits of the heuristic approach. Or course, there is a penalty in accuracy as a tradeoff for the lower runtime. The full generating set of INDs is not found by the heuristic algorithm, rather FIND_2 reports a maximum IND whose arity is about 70%-85% of the largest valid IND between the test data sets. However, through a simple technique which attempts to merge large INDs and checks whether the merged INDs still hold in the database [Koe01], we still correctly find the largest IND in all cases, for this data set. IND merging incurs an insignificant number of additional database queries.

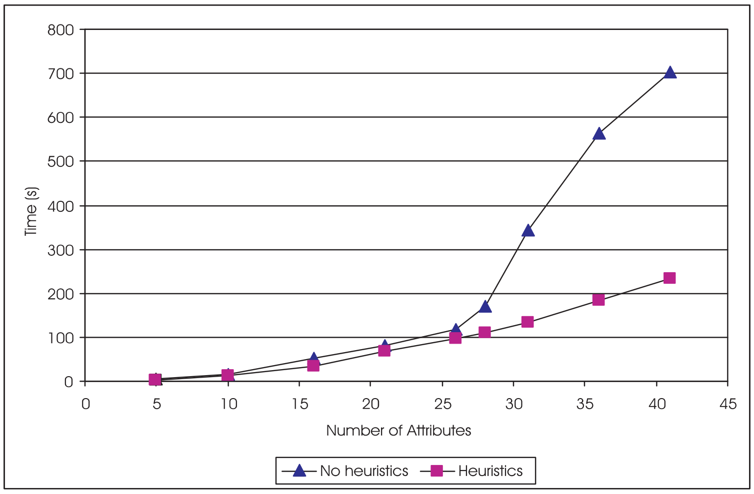

Figure 16: Performance for Data Set CENSUS.

5.4 Experiment 3: Effect of Data Set Size on Runtime

This experiment was performed to demonstrate that, as long as our basic strategy of a separation of the discovery and testing steps of the algorithm is used, the possible reductions on runtime achieved by an improvement of the IND-testing algorithm are small. We let the algorithm discover INDs in different (random) subsets of the same data set (CUP98 from KDD-Archive) and recorded the runtime of the algorithm versus the sum of the numbers of tuples in both input relations. Fig. 17 shows the results. There is a near linear correlation between the two variables, suggesting that the query execution time for the SQL INTERSECT queries used (together) grows linearly with the relation size. We also found a linear correlation between the execution time of *one* such query and the size of the relations involved, which is in agreement with theory if an efficient disk-based sorting algorithm is used to compute the queries. Since a linear algorithm for IND-testing cannot be improved much in comparison to the order-of-magnitude improvements of algorithm FIND_2 vs. the naïve algorithm, this experiment confirms that improving the performance of *IND-testing* is not a priority for our problem.

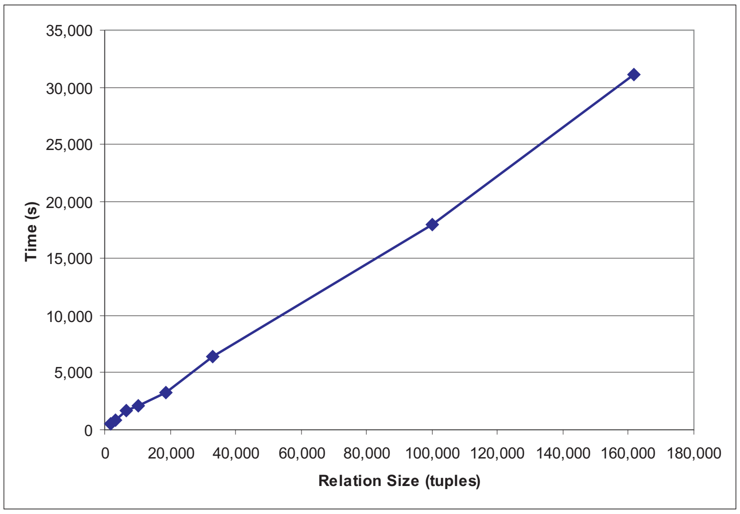

Figure 17: Size of Data Sources vs. Performance of FIND_2

5.5 Experiment 4: Effect of Noise on the Correctness of Algorithm FIND_2

In this experiment, we assessed the influence of small amounts of noise (tuples violating the exact inclusion property) on the FIND_2 algorithm. We experimented on the INTERNET dataset, by creating a 90% random sample and a 99% random sample from the original data set, such that roughly 1% of tuples in the smaller relation violated the target maximal inclusion dependency. We then relaxed the condition on IND-testing (algorithm CHECK) by declaring an IND valid if it is violated by less than 1% of the tuples in the smaller relation.

Fig. 18 shows the ratio of the largest IND found by the algorithm FIND_2 modified as above, versus the largest IND that existed (with 1% noise) in the database. IND merging was used in obtaining the result, and results are reported with and without IND merging. The figure shows that there is some influence of noise on the completeness of the result, but that a large sub-IND of the largest IND is still found in the cases tested. The largest single (unmerged) INDs that were discovered by our algorithm in this experiment had sizes of roughly 50% of the target size, but could be successfully merged to reach at more than 90% of the target size, as shown in the Figure.

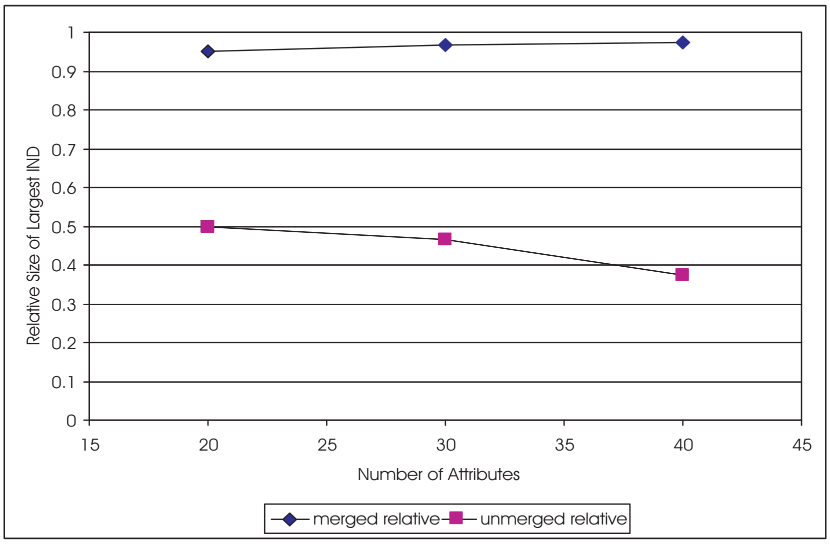

Figure 18: Effect of Noise on the Quality of FIND_2

6 Related Work

Inclusion dependencies have been widely studied on a theoretical level. Fundamental work is done by Casanova, Fagin and Papadimitriou [CFP82]. They present the simple axiomatization for INDs used in our work and prove that the decision problem for INDs is PSPACE-complete. Later, Mitchell [Mit83] developed inference rules for INDs.

Algorithms on cliques in graphs are surveyed in [PX94]. Work on (k-uniform) hypergraphs is somewhat more limited. Several authors (e.g., [GKL96]) have investigated maximal independent sets in hypergraphs, which is a problem related to the clique-problem.

An important related paper is by Kantola, Mannila *et al.* [KMRS92]. The authors describe an algorithm for discovering functional dependencies and also briefly mention inclusion dependencies. However, no algorithm for IND discovery is given, and only a very rough upper bound for the complexity of the IND-finding problem is presented (in addition to a proof of NP-completeness of the problem).

Cohen investigates the problem of non-matching, semantically-equivalent values in databases, which we have not covered in this work [Coh98]. There is substantial work on the discovery of patterns in databases. Much work is concentrated on functional dependencies (FDs), such as Bell and Brockhausen [BB95], or

Savnik and Flach[SF93].

Another important body of work originates in data mining, namely the problem of the discovery of association rules [AR94]. Much work in the literature deals with implementation issues on Agrawal's algorithm or generalizations of the problem [MT97]. Zaki [Zak00] presents efficient algorithms for association rule mining. One of his algorithms also uses the concepts of finding cliques in graphs and clique-merging in a graph. However, there are important differences between Zaki's work and ours: the concept of *support* is defined for association rules but not for our problem, association rules are generally smaller (in "arity") than our inclusion dependencies (and smaller association rules are often more useful than larger ones due to their higher support). Zaki's algorithm also relies largely on the effectiveness of "clique merging"—a technique which we use only as a last resort for our algorithm [Koe01]. In summary, while the two approaches use some of the same mathematical foundations (such as the mapping of the respective discovery problem to the clique-finding problem), Zaki's algorithm uses properties of association rules that are not true for inclusion dependencies. We have made substantial contributions beyond the initial idea of mapping a discovery problem to a clique-finding problem that allowed us to achieve a practically feasible solution to the IND-finding problem in high-dimensional data sets.

There is another body of work in association rule mining that considers improvements on Apriori-mining that is *not* based on finding cliques. Notable examples are Bayardo's *Max-Miner* algorithm, which uses a variety of pruning techniques that rely on the association-rule-specific concept of support [Bay98], and *Pincer-Search*[LK98], by Lin and Kedem, which achieves an improvement on Apriori through a two-way (bottom-up and top-down) search in the lattice of possible association rules. The latter concept is also explored by Flach and Savnik [FS99] in the context of functional dependency discovery. One more example for an improved association rule mining algorithm is by Han *et al.* [HPY00] who propose a tree-based pruning approach that makes use of some association-rule specific problem features. All those algorithms are optimized for their domain and cannot be directly used for our approach, mostly due to their reliance on association-rule specific features. Our algorithm does incorporate the basic idea of Lin and Kedem (and others) of a combined bottom-up and top-down approach.

Other work co-authored by Zaki on association rules includes a parallelization of association rule mining algorithms [ZOPL96].

Hypergraphs have been used in other areas of databases and data mining. For example, Mannila and Raiha [MR94] give an algorithm for the discovery of functional dependencies that maps the problem to a hypergraph traversal.

A paper that in its goal of finding database relationships is related to ours is by Cho *et al.* [CSGM00]. In their work, the authors investigate the discovery of replicated web sites. They make use of the set of hyperlinks in a HTML-file (and compare the graph of links) in order to assess the relationship between web sites. Their approach could solve similar problems to ours, but their approach of comparing the graphs of hyperlinks between two data sources rather than the actual data content differs from our focus.

Also, our emphasis is on data sources that have identifiable attributes with data values in a particular domain, whereas Cho *et al.* focus on semi-structured data. Their solution is not applicable to databases in models other than the semi-structured model.

7 Conclusion

In this paper, we have proposed an algorithm called FIND_2 for the problem of discovering inclusion dependencies over high-dimensional databases.

With our solution, it is possible to automatically compare two databases with known schema, but unknown interrelationships, and identify inclusion dependencies between their attributes. As our algorithm discovers database interrelationships, it is useful for a variety of purposes, such as the identification of database duplicates, system integration support, or the purpose of identifiying backup data for a view synchronization system like EVE [NLR98].

The discovery of inclusion dependencies is a hard problem, with an inherent NP-complexity [KMRS92]. By reducing the problem to a graph problem, we achieved a significant improvement in performance over the naïve algorithm. For larger-size problems, we additionally identified heuristics that help to reduce the search space in our algorithm, achieving good results even for very large problem sizes. A test implementation showed over real-world data sets (from the UC Irvine KDD Archive) that the algorithm is feasible on large datasets with 100 or more attributes.

References

[AR94] R. Agrawal and S. Ramakrishnan. Fast algorithms for mining association rules. In *Proc. Intl. Conf. on Very Large Databases (VLDB)*, pages 487–499, 1994.

[Bay98] R. J. Bayardo. Efficiently mining long patterns from databases. In L. Haas and A. Tiwary, editors, *Proceedings of SIGMOD*, volume 27(2), pages 85–93, 1998.

[BB95] S. Bell and P. Brockhausen. Discovery of data dependencies in relational databases. In Y. Kodratoff, G. Nakhaeizadeh, and C. Taylor, editors, *ML-Net Familiarization Workshop*, 1995.

[BK73] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, September 1973.

[CFP82] M. A. Casanova, R. Fagin, and C. H. Papadimitriou. Inclusion dependencies and their interaction with functional dependencies. In *Proceedings of ACM Conference on Principles of Database Systems (PODS)*, pages 171–176, 1982.

[Coh98] W. W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual/similarity. *SIGMOD Record*, 27(2):201–213, 1998.

[CSGM00] J. Cho, N. Shivakumar, and H. Garcia-Molina. Finding replicated Web collections. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(2):355–366, 2000.

[dLP02] F. de Marchi, S. Lopes, and J.-M. Petit. Efficient algorithms for mining inclusion dependencies. In *Proceedings of International Conference on Extending Database Technology (EDBT)*, 2002. to appear.

[FS99] P. A. Flach and I. Savnik. Database dependency discovery: a machine learning approach. *AI Communications*, 12(3):139–160, November 1999.

[Gar98] S. R. Gardner. Building the Data Warehouse. *Communications of the ACM*, 41(9):52–60, 1998.

[GKD97] M. R. Genesereth, A. M. Keller, and O. M. Duschka. Infomaster: An information integration system. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(2):539ff., 1997.

[GKL96] O. Garrido, P. Kelsen, and A. Lingas. A simple NC-algorithm for a maximal independent set in a hypergraph of poly-log arboricity. *Information Processing Letters*, 58:55–58, 1996.

[HPY00] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In W. Chen, J. Naughton, and P. A. Bernstein, editors, *Proceedings of SIGMOD*, volume 29(2), pages 1–12, 2000.

[KMRS92] M. Kantola, H. Mannila, K. J. Räihä, and H. Siirtola. Discovering functional and inclusion dependencies in relational databases. *International J. Of Intelligent Systems*, 7:591–607, 1992.

[Koe01] A. Koeller. *Integration of Heterogeneous Databases: Discovery of Meta-Information and Maintenance of Schema-Restructuring Views.* PhD thesis, Worcester Polytechnic Institute, Worcester, MA, USA, December 2001.

[LC00] W. Li and C. Clifton. SemInt: A tool for identifying attribute correspondences in heterogeneous databases using neural networks. *Data and Knowledge Engineering*, 33(1):49–84, 2000.

[LK98] D.-I. Lin and Z. M. Kedem. Pincer search: A new algorithm for discovering the maximum frequent set. In *International Conference on Extending Database Technology, EDBT'98*, pages 385–392, Valencia, Spain, 1998.

[MHH00] R. J. Miller, L. M. Haas, and M. A. Hernández. Schema mapping as query discovery. In *International Conference on Very Large Data Bases*, pages 77–88, 2000.

[Mit83] J. C. Mitchell. Inference rules for functional and inclusion dependencies. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 58–69, Atlanta, Georgia, 21–23 March 1983.

[MR94] H. Mannila and K. J. Räihä. Algorithms for inferring functional-dependencies from relations. *Data & Knowledge Engineering*, 12:83–99, 1994.

[MT97] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3):241–258, 1997.

[NLR98] A. Nica, A. J. Lee, and E. A. Rundensteiner. The CVS Algorithm for View Synchronization in Evolvable Large-Scale Information Systems. In *Proceedings of International Conference on Extending Database Technology (EDBT'98)*, pages 359–373, Valencia, Spain, March 1998.

[PX94] P. Pardalos and J. Xue. The maximum-clique problem. *Journal of Global Optimization*, 4:301–328, 1994.

[SF93] I. Savnik and P. A. Flach. Bottom-up induction of functional dependencies from relations. In G. Piatetsky-Shapiro, editor, *Proc. of AAAI-93 Workshop: Knowledge Discovery in Databases*, pages 174–185, July 1993.

[Zak00] M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 12(3):372–390, May/June 2000.

[ZOPL96] M. J. Zaki, M. Ogihara, S. Parthasarathy, and W. Li. Parallel data mining for association rules on shared-memory multi-processors. In *Supercomputing '96 Conference Proceedings: November 17–22, Pittsburgh, PA*, 1996.

A The Bron/Kerbosch-Algorithm

This algorithm has been published in [BK73] and finds the set of maximum cliques in a graph. While the general problem is NP-complete, the runtime of the algorithm is polynomial in the number of cliques, such that the algorithm finishes quickly for graphs with few cliques. The algorithm is given below in pseudocode (with zero-based arrays).

ALGORITHM BRON-KERBOSCH

INPUT:
A graph $G = (V, E)$

OUTPUT:
The set of maximal cliques in G

Stack *compsub*
Set *cliques*
int $nodes[0 \ldots |V| - 1]$

for $i \leftarrow 0 \ldots |V| - 1$
  $nodes[i] \leftarrow i$
extend($nodes, 0, |V|$) *//overwrites set cliques*
return *cliques*

FUNCTION EXTEND

GLOBAL VARIABLES:
Graph G as adjacency matrix $connected[0 \ldots |V|-1][0 \ldots |V|-1]$
Stack *compsub*
Set *cliques*

INPUT:
A set of nodes

function extend(int $oldSet[], ne, ce$)
  int $newSet[0 \ldots ce-1]$
  $nod_{\min} \leftarrow ce; nod \leftarrow 0$ //*number of disconnections*
  for $i \leftarrow 0 \ldots ce-1$
    $p \leftarrow old[i]; count \leftarrow 0$
    for $j \leftarrow ne \ldots ce-1$
      if $(\neg connected[p][old[j]])$
        $count \leftarrow count+1; pos \leftarrow j$
        if $(count \geq nod_{min})$ **break**
    if $(count < nod_{min})$
      $fixp \leftarrow p; nod_{min} \leftarrow count$
      if $(i < ne)$ $s \leftarrow pos;$ $nod \leftarrow 0$
      else $s \leftarrow i;$ $nod \leftarrow 1$
    if $(nod_{min} = 0)$ **break**
  for $k \leftarrow (nod_{min} + nod) \ldots 1$
    $old[s] \leftrightarrow old[ne];$ $sel \leftarrow old[ne]; ne_{new} \leftarrow 0$
    for $i \leftarrow 0 \ldots ne-1$
      if $(connected[sel][old[i]])$ $newSets[ne_{new}] \leftarrow old[i];$ $ne_{new} \leftarrow ne_{new}+1$
    $ce_{new} \leftarrow ne_{new}+1$
    for $i \leftarrow ne+1 \ldots ce-1$
      if $(connected[sel][old[i]])$ $newSets[ce_{new}] \leftarrow old[i];$ $ce_{new} \leftarrow ce_{new}+1$
    compsub.push(sel)
    if $(ce_{new} = 0)$
      cliques.addElement(*compsub*)   //*add stack as new clique*
    else if $(ne_{new} < ce_{new})$ extend($newSet, ne_{new}, ce_{new}$)
    compsub.pop(sel); $ne \leftarrow ne+1$
    if $(k > 1)$
      $s \leftarrow ne$
      while $(connected[fixp][old[s]])$ $s \leftarrow s+1$
          //*find node disconnected from* $fixp$
  return