Efficient Relationship Pattern Mining using Multi-relational Data Cubes

Technical Report UCI-DB-04-05
Department of Information and Computer Science
University of California, Irvine

Dawit Yimam Seid, Sharad Mehrotra
Information and Computer Science
University of California, Irvine
{dseid,sharad}@ics.uci.edu

May, 2004

This material is based upon work supported by the National Science Foundation under Grant No. IIS-0083489 and by the Knowledge Discovery and Dissemination (KD-D) Program.
Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).
Efficient Relationship Pattern Mining using Multi-relational Data Cubes

Dawit Yimam Seid, Sharad Mehrotra
Information and Computer Science
University of California, Irvine
dseid,sharad@ics.uci.edu

Abstract

A large class of data mining applications involve datasets that pertain to multiple entities and relationship. This has led to the suggestion of multi-relational data mining (MRDM) that aims to incorporate and exploit the heterogeneous and semantically rich relationships that exist among entity types. Specifically, given a database consisting of multiple tables linked through foreign key joins, a target table (that typically represents a certain real-world entity type) and, optionally, a target attribute (e.g. a class label attribute), MRDM aims to discover patterns and models spanning all the tables and links that either describe or predict the target entity or attribute. In this paper, we study one class of MRDM task, namely multi-relational association rule mining. For this task, we distinguish two types of target tables, namely entity and relationship target tables. For the case of entity target tables, we cast the computation of frequent patterns as multi-relational iceberg cube computation and propose an efficient algorithm for it. Then, we study the application and peculiar requirements of target tables that are relationship tables. This study reveals a new mining task, dubbed linkage mining, where the mere instances of relationships are the objects of mining. We then show how our multi-relational iceberg computation algorithm is extended to do linkage mining. In the end, we present performance studies of our algorithms.

1 Introduction

A number of analysis and data mining tasks (in a wide variety of applications including intelligence analysis, social network analysis, inter-organizational link analysis, web data mining, genomics) are based much on the links among heterogeneous entities and events as the properties of individual entities. Relational databases facilitate such tasks by enabling the storage of data as multiple linked tables which together represent the conceptual entities and relationships. Nonetheless, techniques, like those developed in the Inductive Logic Programming field (ILP) [12], that aim to mine patterns involving relationships on realistically large databases are hampered by scalability problems.

As a result, most widely used data mining algorithms work only on flat tables. In order to apply these algorithms, one is forced to convert multiple tables into one huge joined universal relation. A number of studies have show that this has several disadvantages. First, joining the relations into a huge “universal relation” blows up the data both horizontally and vertically. This restricts the applicability of many mining algorithms that are already resource intensive [16, 7]. Second, not all real-world data can be converted into a flat joined table without losing important semantic information carried by the join links. This is particularly true for domains where reasoning about the structure of objects and relations between them is inherently required [18]. Third, it increases data redundancy (duplication) that may introduce statistical skew [8].

The problem of learning relationships among instances is studied in the field of Inductive Logic Programming [12]. Recently, the task of relational data mining, also referred to as multi-dimensional data mining (MRDM), has been proposed to extend the methods of relational learning to a variety of data mining tasks [5, 6]. The goal of MRDM is that, given a database consisting of multiple tables linked via foreign key joins, a target table (that typically represents a certain real-world entity type) and, optionally, a target attribute (e.g. a class label attribute), discover patterns and models spanning all the tables and links that either describe or predict the target entity or attribute.

In this paper, we focus on one class of MRDM task, namely multi-relational association rule (pattern) mining. The goal here basically is to discover association rules referring to a particular real-world entity (i.e. the target table) but spanning multiple tables in an arbitrary relational database. Association rules [1] are patterns involving two or more items co-occurring in a certain event (e.g. transaction). The rules in our case are conjunctions of atomic predicates over attributes of tuples which have the form $A_i = v_i$.

\[^1\]These are basically what we count support for...
where \( A_i \) is a categorical or numerical attribute and \( v_i \) is a value from a domain of values (intervals in case of numeric attributes). Each pattern (conjunction) has an associated support and confidence. Support of a pattern is the number of tuples in the database that satisfy it while confidence is the probability of satisfying a certain subset of the conjunction given the whole conjunction.

For prolog databases and datalog queries, Deshaspe and Toivonen have developed an ILP extension of the Apriori algorithm called WARMR \[4, 17\] that discovers association rules over a limited set of conjunctive queries. However, even for the limited class of rules it discovers, this algorithm has been found to be very inefficient. Another category of work \[13, 10\] has considered relational association rule mining on star-schema where the joins are of length one. However, the latter body of work is not applicable to discover the general class of multi-relational patterns at which this paper aims.

In this paper, we explore a different approach to the discovery of multi-relational association rules that is based on extending Iceberg-cube algorithms. For the case of mining association rules from a single table having multiple attributes, the sub-task of frequent pattern discovery on a single table is efficiently processed in databases using Iceberg-cube algorithms \[2\]. These algorithms basically treat the generation of the set of conjunctive predicates that meet the user specified support threshold (also called iceberg condition) as a computation of a cube consisting of cells (i.e. conjunctions) that has all the attributes as dimensions. Iceberg-cube algorithms have been shown to be efficient and can be implemented in current databases with relative ease.

We first propose an algorithm to efficiently discover multi-relational Iceberg-cubes (i.e. frequent conjunctive rules) over multiple tables. Focusing on foreign key relations in a physical database, we propose an algorithm that addresses the challenge of combining cube computation with join path traversal while retaining the good I/O and CPU performance of Iceberg-computation algorithms. In addition to discovering rules (patterns) that span the entire database, we demonstrate that our algorithm is also able to discover rules that involve a limited level of recursion over a certain table.

Second, motivated by the traditional distinction between entity tables and relationship tables in database conceptual design, we take a closer look at the target tables and distinguish two types of target tables: entity tables and relationship target tables. We then observe that most MRDM algorithms in general and WARMR in particular fail to discover useful rules for relationship tables. This is due to the common implicit assumption that instances of mining are some real-world entities. However, when we consider relationship tables as target tables, we are essentially making relationships between entities the instances to mine. We give a formal definition of the task of multi-relational association rule mining with relationships as targets (which we refer to as relationship/linkage rules). Then we show that our multi-relational association rule mining algorithm can effectively and correctly mine such rules. To the best of our knowledge, ours is the first work to systematically study and address multi-relational association rules with target relationship tables in the context of relational databases.

1.1 Motivating Examples

Consider a database consisting of information on researchers and their publications. Figure 1 shows the schema of this database. On this schema, consider the following patterns:

1. Researchers from Institute=UCI co-author with researchers from Institute=IBM. This is a multi-relational association rule with the target table \( \text{Researcher} \) and spans the tables of \( \text{Researcher}, \text{MemberOf}, \text{CoAuthorsWith} \) and \( \text{Institute} \).

2. Researchers from Institute=IBM tend to publish their papers in venue type=conference as compared to venue type=journal. This is a pairing rule with the target table \( \text{Writes} \).

3. Researchers who co-author a paper together commonly belong to the same institute. That is, if researchers R1 and R2 are in a co-authorship relationship, then R1.Institute.Name = R2.Institute.Name. This is a pairing rule with the target table \( \text{CoauthorsWith} \). Notice here that the pattern involves the same attribute of paired entities.

4. Researchers from universities that co-author with researchers from companies also tend to write papers that cite papers written by researchers in companies.
5. Researchers who co-author a paper together also tend to cite one another’s paper.

2 The WARMR family of algorithms

The WARMR algorithm is an ILP algorithm that adapts the Apriori algorithm to mining association rules (represented as datalog queries) with in a certain class of conjunctive queries. Like Apriori, WARMR has two phases: candidate predicate set (conjunction) generation that meet a support threshold and then generation of confident rules. To constrain the space of candidates, WARMR specifies a subset of all conjunctive queries (using a declarative language bias as is typically done in ILP). This constraints specify (1) the key (like a primary key of a certain entity table) for mining that need to be available in all datalog predicates (i.e. tables); this key essentially determines the entity to be counted, and (2) a list of allowed patterns in which atomic predicates can be combined to form datalog queries. To determine when a query is a superset (i.e. generalization) of another query in order to exploit Apriori like pruning, it applies an equivalence test similar to query containment in databases (called θ-subsumption in ILP). This is a very expensive process. Subsequent work [14, 15] has tried to reduce the cost of this computation by removing the need for PROLOG and using a different method for equivalence testing.

Example 1 Suppose that the prolog table consists of three types of atomic facts: customer(name), parent(name, name), buys(name, product). To mine multi-relational association rules on a database of such facts, we first have to select a key, say customer, since it fulfills the requirement of occurring in all atomic predicate types. Then we give a list specifying atomic predicates allowed in the pattern that will be generated (using the warmode language). For instance, we can specify parent(+,-) which means a parent need to be preceded by another predicate (in this case predicate on customer). Given this, examples of patterns that can be discovered by WARMR and expressed using datalog are:

\[
\text{parent}(A,B), \text{buys}(B, \text{cola}).
\]

\[
\text{parent}(A,B), \text{buys}(B, \text{cola}), \text{buys}(A, \text{wine}).
\]

Our approach differs from WARMR in the following ways: (1) WARMR is based on a prolog database consisting of a set of predicates (facts) whereas in this paper we deal with a relational database. We do not require the specification of any pattern structure; instead we directly use the links (i.e. foreign key constraints) among tables found in the database. (2) our candidate generation is based on a much more efficient Iceberg-cube computation algorithms. (3) Although we require a target table, we do not require the key of this table to appear in all tables as in WARMR. (3) As will show later, WARMR can not properly handle target tables that are relationship tables (i.e. tables with more than one key).

3 Iceberg-CUBE Algorithms

To apply association rule mining over a table with multiple attributes, one approach is to transform it into a transaction (market-basket) table by making each atomic predicate (Attribute-value pair) an item. Obviously this representation is inefficient since the number of items involved becomes large. An efficient way to generate predicate sets (i.e. “item sets”) and do the required counting on arbitrary attributed tables is to use a data cube computation algorithm. A data cube consists of all possible group-bys (also called cuboids) over all grouping attributes (each of which forms the cube’s dimension). A type of cube called Iceberg-cube is suitable for association rule mining. An Iceberg query is basically an SQL query with a HAVING clause. An Iceberg-cube is the complete cube over all attributes (dimensions) of a table where each cell meets an aggregation constraint specified in the HAVING clause [2].

A popular Iceberg cube computation algorithm is BUC (Bottom-Up Cubing) [2]. BUC builds the cubes from smaller number of dimension combinations to larger ones. It first selects an attribute and groups the data on that attribute producing partitions pertaining to the attribute’s distinct values \(^1\). It then continues recursively partitioning (and aggregating) the database to compute cuboids composed of larger number of dimensions. At each step of recursive partitioning, it evaluates the iceberg condition (minimum support criteria) to remove those which fail to meet this criteria from further partitioning and aggregation. This approach of pushing the iceberg condition test down the cubing process allows BUC to achieve good performance. The order in which BUC performs group-bys is shown in figure 3.

BUC (as well as other Iceberg-cube computation algorithms to be discussed in sec. 7) scales well to large databases since it needs to load only a partition of the database that fits in memory at a time. Moreover, it caches no data across multiple passes. However, being a depth-first approach, BUC is not optimal in terms of pruning. For example, suppose all cells in \(a_1, c_1, a_2, c_2\) and \(a_1, c_3\) in a cuboid AC at step 7 fail to meet the \(\min_{supp}\) condition. Then, it is not necessary to partition cells in the cuboid AB on C. But also notice that for this to happen all values of cells in the cuboid AC should fail the \(\min_{supp}\). Otherwise, it is still necessary to process ABC although we know that some cells in AC do not meet \(\min_{supp}\) criteria. The later observation means that the pruning optimality of a breadth-first approach (e.g. Apriori [1]) may not

\(^1\)or discretized ranges for continuous attributes
result in greater savings in the case of Iceberg-cube computation. This is because unlike the case of Apriori which is based on item set construction through combinations of atomic items, here computation is based on combinations of attributes each of whose distinct values is an “item”. Due to this, we adopt the BUC approach to use as a basis of our multi-relational iceberg cube computation algorithm.

4 Multi-relational Iceberg-cubes

In this section, we explore how to exploit the approach in BUC to develop an algorithm for iceberg-cube computation over multiple relations for use as a basis of multi-relational data mining. Two major challenges are addressed here:

- how to efficiently interleave cubing (predicate set enumeration) with join path traversal. In particular, how to retain the scalability of BUC by ensuring that only a partition of a single table is read to memory at a time.
- how to minimize the number of tuples involved in the join paths from the target table to the linked table, and then to the tables that are two joins apart from that target table, and so on.

In this section, we give an algorithm that addresses these challenges.

4.1 The Multi-Relational Iceberg Cube Trie

Before we go on to the description of our algorithm, we describe the data structure that we employ to enumerate group-by combination orders and to store counts and other information sufficient for mining multi-relational association rules. Iceberg cuboids and support counts for each cell in each cuboid are stored in a prefix trees (trie) referred to as multi-relational iceberg cube trie or simply MICube trie. Each node in an MICube trie corresponds to an attribute. The target table constitutes the root node. A node at depth $k$ represents the cuboid resulting from grouping by the attributes in the path from itself to the root. A particular node stores all information pertaining to cells in the cuboid represented by the path from the root to that node (see sec. ?? for details). In addition to storing counts, the MICube trie makes it easy to enumerate cuboids (predicate sets) as will be illustrated with an example below.

In order to be able to represent group-bys over attributes from multiple tables in the MICube trie, we distinguish between two types of attributes: data attributes and join attributes. Join attributes consist of foreign and primary keys of tables while all the other attributes are considered to be data attributes. We make the reasonable assumption that semantic information about entities is held in data attributes while join attributes serve to store relationship information only. However, we still perform cubing on both types of attributes. Cubing on data attributes results in cells that consist of $A_i = v_i$ pairs while for join attributes the $v_i$ are foreign keys. Minimum support conditions apply to group-bys of join attributes as well. Semantically, aggregation (count) for group-bys on join attributes captures the number of links that exist between the two entities involved. In the MICube trie, we employ two types of nodes, namely data nodes and join nodes, that correspond to the two types of attributes. Hence, the multi-relational cube represented by MICube trie stores patterns over entity attributes as well as patterns over relationships.

Example 2 The MICube trie for the tables in figure 2 is shown in figure 4. You do not need to understand this figure in its entirety at this point; most of it will become clear as we progress. The figure shows the case where the table $T(A,B,C,D)$ is the target table. All the join nodes (in this case those which are found in more than one table) are designated by a boxed node. The non-boxed nodes are data attributes. Notice that for the most part, the ordering of nodes in the MICube trie is similar to the tree shown in figure 3 that illustrates the execution order of BUC. This similarity allows us to populate the MICube trie as we employ a BUC style cubing algorithm. The MICube trie also makes it easy to generate cubing combinations - we just extend the current node with all its right-brother nodes (i.e. nodes with the same parent node). For instance, Once the data is grouped by $AB$, to get the next nodes to group by, we
just append its right-brothers C and D.

Both primary and foreign key attributes from a table have corresponding join nodes in the MICube trie. For primary keys, we identify all the tables that have it as a foreign key based on the DB schema. By doing this, we effectively reverse the direction of join for those tables which are found earlier in the join path traversal. We perform this reversal of join directionality only for tables that should come under the current node in the tree. That is, we avoid the reversal of join directionality for tables that are at the same join-distance from the target table. When two tables are at the same join-distance, we maintain the original link directionality between them. Also, when a single join node (join attribute) has multiple tables as expansion, we get all those expansions and instantiate multiple join nodes for them in the cube tree.

Finally, at each level in the cube tree, we require that all join nodes be placed at the right of all the data nodes as is shown in figure ?? . The reason for this will be clear once we describe our algorithm.

4.2 Algorithm description

We present our overall approach before we turn to the detailed description of our algorithm. The algorithm iterates between two phases, namely cubing phase and expansion phase. The cubing phase starts with the target table and recursively performs cubing on each data attribute using BUC (i.e. applying min_supp pruning) until a join attribute is encountered or no more group-by is possible. As mentioned above, this process is guided by the cube order enumeration based on the MICube trie. Counts produced by the cubing phase are written to the MICube trie as soon as they are computed. When a join attribute is reached, group-by is performed on it like the other data nodes. But, the recursion stops and returns towards the root. As such, join nodes

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4.png}
\caption{Processing of the Multi-relational Iceberg cube computation}
\end{figure}

\begin{algorithm}
\caption{Algorithm for predicate set computation}
\begin{algorithmic}[1]
\Procedure{MICube}{input,CloneJoinNodes}
\State // Cubing Phase
\State CubingAttributes $\leftarrow$ find next attributes to group by using MICube trie
\State \For {each attribute $a$ of the input}
\State \State $j \leftarrow 0$
\State \State \For {each partition $p_j$}
\State \State \If {CellCount $a[p_j] \geq$ min_supp}
\State \State \State insert count to current Leaf Node of $a$ (if absent, create a new one and append to the current leaf node)
\State \State \EndIf
\State \State \EndFor
\State \State \EndFor
\State \State //Stop recursion at join attributes
\State \State \If {$a$ is a data attribute}
\State \State \State for each $CloneJoinNodes[k]$ do
\State \State \State expand($CloneJoinNodes[k]$)
\State \State \EndFor
\State \State \EndIf
\State \State //Expansion Phase
\State \Repeat
\State \State \State $j \leftarrow j + 1$
\State \State \State \EndIf
\State \State \EndFor
\State \State //Expansion Phase
\State \State \State until no more expansion is possible
\State \EndProcedure
\end{algorithmic}
\end{algorithm}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5.png}
\caption{Algorithm for predicate set computation}
\end{figure}
serve as the boundary of the cubing phase. By the time the recursion returns to the root, all the leaves in the cube tree will have either join nodes or a data node with the finest granularity. It is important to notice here that the attribute values in the join nodes are foreign keys.

Once cubing phase is done, the expansion phase takes over. Expansion extends the cubing to joined tables. The goals here are to perform expansion without increasing the memory requirement (as joins are carried out) and to avoid reading a data partition more than once thereby retaining the advantage of the recursive approach of BUC. We achieve both of these goals by the following technique: pick the left most join node that has the smallest depth (the reason for this is given below). For this node, we also identify and store in a queue all other leaf join nodes (called "clones") that also correspond to the same attribute by traversing the MICube trie horizontally backward from right-to-left. We then join only the first node (by joining the foreign keys stored in this join node with the actual referenced table). Notice here that only tuples that meet the min_supp are joined.

Next, we launch the cubing phase again using the joined table and the clone queue. However, this time each data partition that is loaded to memory will be used to concurrently carry out cubing by all the clone nodes as well. This approach results in considerable I/O and CPU efficiency. Also notice that in addition to storing counts and enumeration of cubing order, our algorithm also exploits the MICube trie to effectively integrate cubing with join path traversal.

Our algorithm for multi-relational iceberg cube computation is shown in figure 5. The input to it are a data partition (initially the whole target table) and a queue of join nodes on which the partition is to be joined as expansion (which is initially empty). In line 2 the group-by order to be applied on the current partition is computed from the MICube trie. Line 4 partitions the data after sorting it based on the grouping attribute. Then for each partition, once we test if it passes the iceberg condition (line 7) and also if the current cubing attribute is not a join attribute (line 10), we make a recursive call to *MICube* (line 14). Notice that for join nodes, short of making a recursive call, we perform all the other processing (i.e. partitioning, counting, updating the relevant node in the cube tree). Since at this point *CloneJoinNodes* is empty, the expand function in the loop (lines 11-13) is not executed. But, as described above, when the recursion is being done after the expansion phase, this loop will be run as many times as there are clone join nodes in the MICube trie on which cubing can be performed using the current partition.

Once all the data attributes of the target table are processed (loop in lines 3-19), the recursion returns upwards through all data nodes, at the completion of which the expansion phase is launched. We start, in line 22, by picking the join node that has the least depth and is at the left most of the MICube trie. The reason for picking the join node with the least depth is because it is the least aggregated one. This property of a least-depth join node guarantees that result of joining foreign keys in this node with the referenced table is sufficient to expand all the nodes in the MICube trie pertaining to a join attribute. The following lemma states this concretely:

**Lemma 1** Let $R$ be a relation that is computed by joining foreign keys with attribute name $att_x$ for a least-depth join node with the table to which those keys reference. Then, $R$ is the superset of all relations that can be computed based on any other join node with attribute name $att_x$ in the MICube trie.

**Proof.** Recall that in MICube trie, cubing order is generated by appending right-brothers of a node. Therefore any clone node $N'$ on the left of $N$ has a depth $\geq \text{N}_\text{depth} + 1$. Hence, the size of any cell in the cuboid represented by $N'$ is at most as large as its corresponding cell in the cuboid $N$. Further, recall that $N'$ and $N$ share some prefix in trie. Therefore, there can not exist any tuple $t$ with the following property: $t \in N' \land t \notin N$.

Now we explain the second part of the join node selection criteria, namely picking the left most join node. The rationale for this also relates to the fact that cubing order is enumerated by appending all right-brothers of a MICube trie node. Expanding the left most join node allows this procedure to work correctly since all the right-brothers of that node (if any) can also be appended as the join node grows due to expansion. Notice that these two criteria for join node selection alone are sufficient to guarantee that the join nodes are traversed (expanded) correctly and efficiently without having to keep track of the order in which they were appended to the trie.

Turning back to the description of the *MICube* algorithm, once we pick the node to expand we first expand it by joining the keys in the join node with the referenced table. As noted earlier, since only foreign keys that meet the min_supp condition are found in the join node, no redundant join is performed. At the same time, we traverse all the cube tree leaves from right-to-left to store in a queue (CloneJoinNodes) all "clone" (i.e. same attribute) nodes of the selected join node (line 23-24). The CloneJoinNodes queue is passed in the recursive call to *MICube* so that subsequent cubing based on each data partition of the current expansion is applied on all of these join node instances. Hence, all subtrees of join nodes of the same attribute are simultaneously grown across the MICube tree based only on one read of the join table (or partition there of).

When all instances of selected join node and its clones are completely processed (i.e. cubed until all their leaves
are join nodes or no more cubing is possible ), we pick the next unique join node, and repeat the above process. This continues until no more cubing or expansion is possible (i.e. until all leaf nodes are data nodes with no right-brother). Notice that once a join phase starts, the cubing is encapsulated inside a join node expansion. Also notice that in the MICube algorithm, if a table ( either the target table or any joined table) does not fit in memory, then the algorithm will keep on partitioning it until it does.

4.3 Cubing order for Neighboring Join Nodes

While the elegant strategy of combining a data node with its right-brother to exhaustively enumerate cubing orders works well for data nodes, it presents difficulty for group-by combinations of two brother ( same level) join nodes. We will illustrate this by example. In figure 4, consider the join node D at the right side of the tree represented by a dotted box (as a child of node C). Basically, this group-by order is equivalent to merging two join branches from the ancestor table ( in this case the target table). Hence, a cubing based on these two nodes ( i.e. two foreign keys) is equivalent to pairing nodes from the cuboids formed by two join paths. However, the semantics of such pairings needs to be defined \(^3\). We will do that in section 5. At this point we simply disallow such cubing orders.

Now we have presented all the material need to explain figure ?? completely.

Example 3 We illustrate the processing of our algorithm using figure 4. The order in which the recursive iceberg-cubing part of the algorithm is executed is shown by the labels q1, q2, etc. The two zigzagging lines across the tree show the point where the algorithm entered into the expansion phase. The order in which the join nodes are expanded is shown by the labels e1, e2, etc. We show the order of recursive cubing done at the second and third expansion rounds only for the join nodes at the extreme right. As we explained, the same processing is done simultaneously for all clones of a selected join node. Finally, we show via a broken arrow the cases where the group-by of the join node C with the join node D is disallowed.

4.3.1 MICube Trie Node structure

The MICube trie stores the iceberg cuboids that are computed on the path from the root to each node. Recall that a path in this trie represents a cuboid. Information on the cuboids and their resulting cells is stored in each node of the tree. We give an efficient ( compressed) representation of the node in the box shown in figure 4.

In each child node, all the attribute values are stored in an array and all the counts are in another array. In particular, the counts stored at the current node correspond to every pairing of an attribute value in the current node with every value of the parent node. For example if there are \( n \) attribute values in a node and \( m \) attribute values in its parent node and assuming no pairing is pruning, the current node will store \( mn \) counts. All these counts are stored as a single array and each value in the parent node stores the starting array index for all pairings done with it.

Example 4 Consider the combination AB at left branch of the MICube trie in figure ???. The node A ( corresponding to grouping attribute A) will contain counts for \( a_1,a_2,a_3 \) in the count array. At this point, the pointer array is empty. Then, when cubing by B is done, for each pairing of values of A and B, a count is stored in node B as \( [a_1,b_1],[a_1,b_2],\ldots,[a_2,b_1],[a_1,b_1],\ldots \) Then the array index where counts for \( a_1, a_2, \text{ etc}. \) start is stored in node A. This way, we store counts in a compressed form.

4.4 Join Node Expansion

We can view the tree given in figure 4 as a superimposition of three trees each at different level of granularity. We have already discussed the two levels. At the lowest ( cell) level, we have a set of iceberg-cuboids. Links at this level are represented by the pointers stored in each MICube node. At the medium ( cuboid) level, we represent the processing order of the cubing algorithm ( i.e. group-bys ordering of attributes). The third and highest-level tree is that of the tables linked by join nodes. At this level, the tree - actually a directed acyclic graph ( DAG) - can be interpreted as traversing join paths starting from the target table following join keys ( attributes).

As mentioned in the algorithm description, we have two basic types of expansions on join nodes, namely one that involves join ( done on the least-depth left join node) and another that is performed on clone nodes. For the former one, we use simple foreign key join. The latter can be performed by applying intersection ( e.g. hash set intersection) of the keys in the join node and the corresponding key in a partition. To facilitate this as expansion is simultaneously done on multiple join nodes, all the current leaves with a join node as ancestor also store the keys of tuples in each cell.

4.5 Generation of rules from MICube trie

Once we computed the multi-dimensional Iceberg-cuboids, the cuboids and counts stored in the MICube trie are used to generate multi-relational association rules. This can easily be performed by picking each path in the MICube

\[ \text{Example 2}\text{ } \text{select count(*) from T1, T2, T3 where T1.id=T2.id AND T1.id=T3.id group by T2.id,T3.id} \]
trie. Specifically, from each path, we find all coupling of patterns (say \( P_1 \) and \( P_2 \)) such that \( P_2 \) is contained in \( P_1 \) and this coupling meets the confidence threshold.

5 Cubing on neighboring join nodes

Now consider the problem of interpreting cubing combinations for two join attributes of a table which we simply disallowed in the MICube algorithm (see sec. 4.3). We address a simplified version of the problem where a table contains only join nodes since an approach for this case will also be applicable in the general case where the join attributes occur with data attributes in a table. Tables that contain only join attributes are commonly used to represent relationships between two or more entities in the database. Such tables are called relationship tables in traditional database conceptual design [19].

Now suppose this table is our target table. One way to interpret a multi-relational association rule based on this table is as a rule that captures the pairing of entities in the relationship. Below we give a more concrete semantics of such a pattern:

**Definition 1** Given a relational database and a target table \( T \) composed of relationship tuples (i.e. \( \langle f k_1, f k_2, \ldots \rangle \)), let \( R_i \) be the joined tuple set based on a join path starting from \( f k_i \) values, a pairing or linkage rule has the following form and semantics:

\[
pattern(R_1) \land pattern(R_2) \rightarrow \langle f k_1, f k_2, \ldots \rangle \in T,
\]

where each \( pattern(R_i) \) is a conjunction of predicates on attributes of \( R_i \). Rules of this form are called pairing or linkage rules.

Next we state the multi-relational pairing rules discovery problem:

**Definition 2 (Multi-Relational Pairing Rules)** Given a database, a target table composed of \( k \)-relationship tuples (i.e. \( \langle f k_1, f k_2, \ldots f k_k \rangle \)), and a \( \text{min}_\text{supp} \), generate all pairing patterns (pairs of cuboid cells) that meet the \( \text{min}_\text{supp} \) (iceberg) criteria.

5.1 Mining pairing with MICube

A naive approach to adapt the MICube algorithm for this task is to first apply it on the table corresponding to each entity involved in the relationship to generate the Iceberg cubes and then generate pairings of cells from each iceberg cube to generate candidate pairing rules. Those candidates that are satisfied by sufficient tuples (i.e. \( \text{min}_\text{supp} \)) in the target relationship table are computed. Below we give a considerably more efficient algorithm.

The basic idea of our approach is pushing the pairing \( \text{min}_\text{supp} \) test down to the processing of iceberg cube for each side of the rule. Specifically, we give an algorithm akin to nested loop join. We first select one of the pairing keys and expand it and compute an iceberg cube over it which is represented as a cube tree. Then, as we perform iceberg cube on the second one, we push a condition of checking for pair min support for each partition it processes. That is, after we group by an attribute and before we further partition it by another group by, we first form the pairs and test if they meet the pairing \( \text{min}_\text{supp} \). The correctness of this pruning follows from the observation that pairing rules are monotone (i.e. if a pair fails to meet the \( \text{min}_\text{supp} \), any other pair only consisting further partitioning of either components will also fail).

The algorithm has double advantage: we prune the cubing work for one of the tables, and more importantly we interleave the generation of the pairing rules along with the computation of iceberg cubes. The algorithm is given below:

```plaintext
1: Load the target relationship table (or a partition of it)
2: Pick one of the paired keys (select a smaller side, if possible) and run the MRIC algorithm. In the resulting cube store the tid for each node.
3: Run MRIC on the second table. For each partition being counted by the MRIC algorithm, in addition to checking that partition for \( \text{min}_\text{supp} \), also check if that partition meets pairing \( \text{min}_\text{supp} \). Also apply additional pruning on the pairing as follows: if pairing of e1, g1 fails, then do not continue down checking e1, g1, j1 etc. Output each resulting pairs.
```
6 Performance Analysis

In this section, we present the results of our comprehensive study to validate the efficiency and scalability of our algorithm. In particular we show that:

- Our algorithm scales well with the increase in average data size per table, average dimensionality per table, number of tables involved, average join path length, support threshold.
- Our MICube Trie representation is compact and that it scales well as the leaf branching factor (fan out) increases.

Setup. We implemented our algorithm using Java. The implementation works on tables stored in DB2 running on a Xeon 2.4GHz PC with 2GB RAM. Our program runs as a stored procedure and interacts with DB2 only to load tables. However, we do not always load a whole table. Particularly, as has been described earlier, during an expansion operation, only parts of a new table that join with foreign keys in the current table that also meet the Iceberg criteria are loaded. The join of qualifying foreign keys with the referenced table is executed by the DB2.

As has been done in previous Iceberg papers, we assume that each table (not all tables involved in the computation) fits in memory[20, 2, 9, 21]. However, the results we report below include both IO time needed to load tables and CPU time.

In addition, in our implementation, we assume that all attributes have integer values. This, too, is a common assumption in cube computation in general [2, 21] in order to save space and facilitate sorting which is used for data partitioning. Other attribute types like string and interval values are mapped to integers in a preprocessing step. We do not include the time taken by this step in our experiments.

Figure 8 presents a preliminary experimental result on efficiency of the MICube algorithm as measured by run time. This figure compares MICube’s runtime with the "Join-then-mine" approach that first creates a join of the relations to be mined (using outer joins with the target table when possible) and applies the single-table Iceberg cubing algorithm on it. In addition to the improved performance gain using MICube, the cubes generated by MICube have also effectively retained the relationship structure.

7 Related Work

The problem of multi-relational association rule mining was presented in [3, 4, 17]. This body of work described WARMR summarized in section 2. Subsequent work [14, 15] on a system called FARMR has tried to optimize WARMR by removing the need for PROLOG and using a different method for rule equivalence testing. However, FARMR also assumes a prolog database of facts and a WARMMODE style declarative bias.

There is also a recent body of work that focused on star schema based association rule mining [10, 13]. The problem addressed here is to mine association rules over the fact table without having to perform a join. The approach followed is to apply Apriori [1] on each table first and then merge the discovered frequent item sets to form multi-table frequent item sets. Unlike this body of work, we consider association rules over tables with an arbitrary depth of joins. Besides, use the recursive depth-first based approach of Iceberg-cube computation algorithms that allow us to efficiently generate frequent patterns as we traverse join paths.

In addition to the BUC algorithm [2], there are more optimized algorithms for Iceberg-cube computation [9, 20]. The algorithms suggested in these papers exploit shared computations of partitioning and aggregation to achieve better performance while retaining the possibility of applying Iceberg-conditions. We chose BUC as the basis of our approach because it is the most popular one and because it was simpler. However, the optimizations achieved by the latter works can also be incorporated with our algorithm to achieve even better performance.

Finally, the body of work on association rule mining over graphs [11, 22, 23]. However, this body of work considers only structural relationships and ignores the node and edge attribute. In this paper, we have tried to discover patterns that pertain to both entity attributes and their relationship structures.

8 Conclusion

In this paper our focus has been the development of techniques for frequent pattern discovery for the purpose of
multi-relational association rule mining. To this end, we developed an algorithm that integrates Iceberg-cube computation methods with systematic join path traversal. In particular, our two phased algorithm performs recursive cubing and joined table expansion iteratively in order to compute all Iceberg-cubes that span multiple linked tables. We also provide a compressed data structure that enables to store the discovered Iceberg-cube along with cell-level counts. Then, for the purpose of discovering patterns over branching multi-relational cuboids, we give a semantics for one type of pattern and suggest a method to compute it efficiently based on our MICube algorithm.

There are a number of ways we plan to extend the work presented here. Primarily, we plan to study whether the techniques described in this paper can also be applied to other multi-relational data mining tasks in particular classification methods like decision trees. We also plan to study the various types of patterns (for e.g. pairing rules over same-entity type relationships like co-authorship where the two entities have shared attributes). Another issue to be explored is measures of interestingness for pairing rules other than the conventional support-confidence framework.

References


