Targeted Association Rule Mining in Data Cubes

Shrutendra K. Harsola[†] Prasad M. Deshpande[‡] Jayant R. Haritsa[†]

Technical Report
TR-2012-03

[†]Database Systems Lab
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore 560012, India
`http://dsl.serc.iisc.ernet.in`

[‡]IBM Research India
Bangalore 560045, India

Abstract

We address the problem of mining targeted association rules over multidimensional market-basket data. Here, each transaction has, in addition to the set of purchased items, ancillary dimension attributes associated with it. Based on these dimensions, transactions can be visualized as distributed over cells of an n-dimensional cube. In this framework, a targeted association rule is of the form $\{X \rightarrow Y\}_R$, where R is a convex region in the cube and $X \rightarrow Y$ is a traditional association rule within region R.

We first describe the TOARM algorithm, based on classical techniques, for identifying targeted association rules. Then, we discuss the concepts of bottom-up aggregation and cubing, leading to the CellUnion technique. This approach is further extended, using notions of cube-count interleaving and credit-based pruning, to derive the IceCube algorithm. Our experiments demonstrate that IceCube consistently provides the best execution time performance, especially for large and complex data cubes.

1 Introduction

In this paper, we address the problem of *targeted* association rule mining (TARM) over complex multi-dimensional market-basket data cubes. Our goal here is to extend traditional association rule mining to capture targeted rules that apply to specific customer segments. Thus the minimum support for rules in a particular segment is now a percentage of the number of transactions that map *to that segment*, rather than the total number of transactions over all the segments. To model this scenario, we consider multi-dimensional market-basket data in which each transaction has, in addition to the set of purchased items, ancillary dimension attributes associated with it. Based on these dimension attributes, transactions can be visualized as distributed over cells of a data cube, with customer segments corresponding to convex regions in the multi-dimensional space.

A sample data cube with time and location dimensions is shown in Figure 1. The dimension attributes are grouped into tree-structured containment hierarchies, where cities are grouped into states and then countries, while quarters are grouped into years. Transactions are assigned to cells based on the values of their dimensional attributes, and our goal is to identify association rules that are applicable to localized regions. For example, a rule such as:

$$\{raincoat \rightarrow umbrella\}_{Q3,Washington}(sup = 5\%, conf = 70\%)$$

indicates that umbrellas are often bought in conjunction with raincoats during the fall season in Washington State, USA.

Prior Work. In the literature, there is a substantial body of work on rule mining over data cubes (e.g. [10, 16, 9, 5, 11]). However, most of these prior efforts mine for *inter-dimensional* rules over aggregated data in a cube – for example, they discover rules of the form "2011 → Washington" on the time and location dimensions. While a few do mine for intra-dimensional rules like us, they typically employ a single global support threshold at all levels in the dimension hierarchy. This model enables them to use variants of the classical Apriori mining algorithm [2], but makes it infeasible to discover targeted rules except by taking recourse to the computationally inefficient strategy of relaxing the minimum support to a much lower value. Using different supports at different levels in the dimension hierarchy has been suggested in [7, 16], but they mine each level in a separate pass, leading to repeated counting of the same itemset over multiple levels.

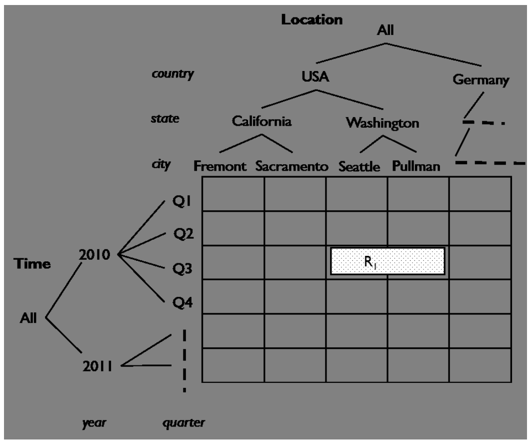


Figure 1: Example 2D Mining Cube

Targeted ARM. Mining for localized rules with region-specific support counts is a difficult problem since the fundamental Apriori monotonicity property – all subsets of a frequent itemset must also be frequent – no longer holds. For example, with reference to Figure 1, the itemsets (Washington, raincoat) and (Q3, raincoat) may lack 1% support in the individual regions corresponding to (Washington) and (Q3), but (Washington, Q3, raincoat) can still retain 1% support in the (Washington, Q3) region (the shaded box in Figure 1).

One obvious mechanism to generate such localized patterns is to include the dimensional attributes as part of the transactional schema (e.g. [13, 7, 16]), and mine with the support threshold reduced to the value required by the most thinly-populated region. However, this approach is computationally impractical due to the large number of spurious patterns generated and processed in the more populous regions. Therefore, we instead take recourse to a bottom-up method similar to the **TOARM** approach [14]. In this method, the frequent itemsets in individual cells are initially computed, and subsequently aggregated to compute the itemsets for the regions. We then exploit the cube operator to avoid counting the same itemset repeatedly over multiple regions, leading to the **CellUnion** algorithm. Limiting the redundant counting of itemsets is further improved through incorporating an interleaved counting-cubing technique and a credit-based pruning mechanism, resulting in the **IceCube** algorithm. An experimental evaluation over a representative set of synthetic datasets shows that IceCube consistently provides superior execution performance, especially for large and complex data cubes.

In the rest of the paper, we focus solely on targeted mining of frequent itemsets, since generating association rules from these frequent itemsets is a straightforward task [1].

2 Problem Definition

We consider market-basket data in which transactions possess ancillary dimensional information. That is, each transaction is of the form $T = \{i_1, \ldots, i_k; d_1, \ldots, d_n\}$, where $i_1, \ldots, i_k$ are the market-basket items, and $d_1, \ldots, d_n$ are the values of the dimensional attributes. For example, {raincoat, umbrella, shoes; Q3, Seattle} is a transaction with three purchased items: raincoat, umbrella and shoes, while Q3 and Seattle are the values of the time and location dimensions, respectively.

Consider a cube constructed over n dimensions, with each dimension j having an associated tree hierarchy, the tree node values comprising its domain, D_j. As a case in point, the domain of *location* in Figure 1 is {USA, California, Fremont, Sacramento, Washington, Seattle, Pullman, Germany, ...}. Let D be the cartesian product of the dimensional domains, that is, $D = D_1 \times D_2 \times \cdots \times D_n$. Then, any $C = (c_1, c_2, \ldots, c_n) \in D$ defines a *cell* in the cube, if $\forall j, c_j$ is a leaf in the j^{th} dimension's hierarchy. Further, any $R = (r_1, r_2, \ldots, r_n) \in D$ defines a *region* in the cube, if $\exists j$ s.t. r_j is not a leaf in the j^{th} dimension's hierarchy. For example, R_1 = (Q2, California) defines a region containing two cells: (Q2, Fremont) and (Q2, Sacramento).

Let *minsup* be the user-specified minimum support threshold, expressed as a percentage of the transaction population over which it is evaluated. Our goal is to identify all the *targeted* frequent patterns of the form $\{I\}_R$, where R is a region in the cube and I is a frequent pattern in the subset of transactions belonging to the region defined by R. For example, if there are 100000 transactions in region (Q3, Washington), items raincoat and umbrella appear together in 5000 of these transactions, and $minsup$ is 1%, then the targeted frequent itemset corresponds to the example presented in the Introduction:

$$\{\text{raincoat, umbrella}\}_{\{\text{time: Q3, location: Washington}\}}(\textit{sup} = 5\%)$$

Note that when R is the entire cube, then we obtain the traditional global mining results.

In the sequel, we will use the notation $L_k(C_i)$ to denote (locally) frequent itemsets of size k in cell C_i.

3 Cube Mining Algorithms

In this section, we describe a suite of three mining algorithms for producing targeted region-specific association rules – these algorithms, titled TOARM, CellUnion and IceCube, cover a broad spectrum of design choices, and are quantitatively compared on a variety of performance metrics in the experimental evaluation of Section 4.

3.1 TOARM Algorithm

A property of the cube that is straightforward to observe is that an itemset can be frequent in a region only if it is frequent in atleast one of the cells in that region. This property has been used previously in the TOARM [14] approach for online association rule mining over regions. To compute the frequent itemsets for a region R, the TOARM algorithm computes the union, $\mathcal{U}_R$, of the frequent itemsets over all individual cells in the region R. Any itemset frequent in the region must necessarily belong to this set. The algorithm then counts the itemsets in $\mathcal{U}_R$ over all the cells in R and prunes the itemsets that do not satisfy the minimum support criterion. A straightforward adaption of the TOARM algorithm for our problem is to repeatedly invoke TOARM for each region R in the cube. The pseudocode for this approach is listed in Figure 2.

3.2 CellUnion Algorithm

TOARM counts an itemset separately for each region, thus leading to repeated counting. Instead, we could take the union, $\mathcal{U}$, of the frequent itemsets over all the cells in the cube. A frequent itemset over any region of the cube must necessarily be present in $\mathcal{U}$. We first count the itemsets in $\mathcal{U}$ over all cells

```
Algorithm TOARM:
For each region R in the cube
    • Let $C_1, C_2, \dots, C_m$ be the cells contained in region R
    • Compute union: Compute union of local frequent itemsets from all cells contained in R.
        – $\mathcal{U}_R = L(C_1) \cup \cdots \cup L(C_m)$
    • Count: Count all the itemsets of $\mathcal{U}_R$ over region R i.e. $\forall I \in \mathcal{U}_R$, compute $count(I, R)$.
    • Filter and generate output: $\forall I \in \mathcal{U}_R$,
        – if $count(I, R) \geq ntrans(R) * minsup$, output targeted frequent itemset $\{I\}_R$
```

Figure 2: Algorithm TOARM

```
Algorithm CellUnion:
    1. Compute union: Compute union of locally frequent itemsets from all cells of the cube.
        • $\mathcal{U} = L(C_1) \cup \cdots \cup L(C_N)$
    2. Count: Count all the itemsets of $\mathcal{U}$ within each cell, i.e. $\forall I \in \mathcal{U}$ and $\forall$ cells C, compute
       $count(I, C)$.
    3. Cube: For all itemsets in $\mathcal{U}$, recursively aggregate cell level counts to obtain region level counts.
        • This provides $\forall I \in \mathcal{U}$ and $\forall$ regions $R$, the value of $count(I, R)$.
    4. Filter and generate output: $\forall I \in \mathcal{U}$, $\forall$ $R$
        • if $count(I, R) \geq ntrans(R) * minsup$, output targeted frequent itemset $\{I\}_R$
```

Figure 3: Algorithm CellUnion

in the cube, after which the counts for any region can be simply computed by aggregating the counts from cells contained in the region.

The pseudocode of this CellUnion algorithm is shown in Figure 3. Here, Step 2 which counts itemsets can be efficiently achieved using a trie data structure. Further, Step 3 can also be computed efficiently using a cubing algorithm. Now, note that only Step 2 is dependent on both the transaction cardinality and the number of frequent itemsets in each cell, whereas all other steps are solely dependent on the frequent itemset population in each cell. However, it is still possible that the initial union sizes may turn out to be extremely large, resulting in inflated counting overheads. We attempt to address this issue in the next algorithm.

3.3 IceCube Algorithm

The IceCube algorithm (Interleaved Credit Elimination Cube Mining Algorithm) is based on two ideas, described below: 1) *credit-based* elimination or pruning of candidate itemsets; and 2) *interleaving* of counting and cubing processes (in contrast to the previous algorithms where cubing and counting were two separate and distinct phases).

Credit-based Pruning. We use the concept of *credit*, as defined recently in [6]. Specifically, for each cell, we have the set of frequent itemsets and their counts. Let $sup(C)$ be the absolute support threshold for cell C i.e. $sup(C) = minsup * ntrans(C)$. We now define the credit of an itemset I in cell C as:

$$credit(I, C) = \begin{cases} count(I, C) - sup(C), \\ \qquad\qquad \text{if I is frequent in cell } C \\ -1, \qquad \text{otherwise} \end{cases}$$

That is, the credit of an itemset I in cell C indicates the extra count of I above the support threshold in that cell. The credit will be zero or positive for frequent itemsets in that cell, and negative for non-frequent itemsets. Since we don't have the actual counts of non-frequent itemsets, we assume the maximum possible count which is $(sup(C) - 1)$. Hence, the credit for non-frequent itemsets is taken as -1. Extending the notion of credit from cells to regions, the credit of an itemset I in region R is the aggregate credit of I over all cells C contained in region R, that is,

$$credit(I, R) = \sum_{C \in R} credit(I, C)$$

Analogous to cells, we can prune from any region all candidate itemsets whose credit is negative in that region. Further, we need to count an infrequent itemset I in a cell C, only if there exists a region R enclosing C in which I can possibly be frequent. Step 1 in the pseudocode of the IceCube algorithm shown in Figure 4 describes how to use credit-based pruning to compute the final set of "survivor itemsets", denoted $S(C)$, to be counted in each cell C. So, instead of counting the whole union $\mathcal{U}$ in each cell, we merely need to count the cell-specific survivor subsets.

Interleaved Counting and Cubing. Here, the two procedures of counting and cubing are carried out in an interleaved fashion, iterating over the itemset length. The idea is that it is necessary to count an itemset of length k in a cell C only if all its subsets of length $(k - 1)$ are frequent in at least one region containing the cell C. More concretely, in each iteration k, we first generate and count the candidate k-itemsets, after which these counts are cubed to generate frequent k-itemsets over all regions. Interleaving the two phases ensures that the number of candidates evaluated during each pass is reduced since frequent itemset information is available at the region level (unlike CellUnion, where frequent itemset information was only available at the cell level).

We now define the notion of *region k-set* of a cell: The region k-set of a cell C, denoted $R_k(C)$, includes all k-itemsets that are frequent in some region enclosing cell C. Armed with this notion, the following lemma is useful in pruning the itemsets to be counted in each cell, as shown in Step 2 of Figure 4.

Lemma 1. *A k-itemset I_k needs counting in cell C only if*

1. *$I_k \in S_k(C)$, where $S_k(C)$ is the subset of $S(C)$ containing itemsets of length k, and*

2. *All $(k - 1)$-subsets of $I_k \in R_{k-1}(C)$*

```
Algorithm IceCube:

  1. Generate survivor sets for each cell of the cube using credit-based pruning mechanism
       • Compute cell union:
         $\mathcal{U} = L(C_1) \cup \cdots \cup L(C_N)$
       • Compute cell level credits: In each cell, compute credits for all itemsets in $\mathcal{U}$ –
         $\forall I \in \mathcal{U}$ and $\forall$ cells C, $credit(I, C)$.
       • Cube to get region credits: Recursively aggregate cell credits to obtain region credits –
         $\forall I \in \mathcal{U}$ and $\forall$ regions $R$, $credit(I, R)$.
       • Identify survivor itemsets: In each cell, only retain candidate itemsets with non-negative
         credits in some enclosing region –
         $\forall I \in \mathcal{U}$ and $\forall$ regions $R$, if $credit(I, R) \geq 0$, then add $I$ to $S(C)$ $\forall$ cells C contained in
         region R, where $S(C)$ is the survivor set for cell C.

  2. Iterate over itemset length. Pass $k$ is as follows:
       • For each cell $C$
           – If $(k = 1)$, then $cand_k(C) = S_k(C)$
             else (use Lemma 1)
               ∗ Compute $cand_k(C)$ from $R_{k-1}(C)$ using Apriori-gen [2].
               ∗ Prune using survivor sets –
                 $cand_k(C) = cand_k(C) \cap S_k(C)$
           – Count all itemsets of $cand_k(C)$ within cell $C$
       • Cube: Recursively aggregate cell counts of itemsets to obtain counts over all regions.
       • Filter and generate output: if $count(I, R) \geq ntrans(R) * minsup$, then
           – Output targeted frequent itemset $\{I\}_R$
           – Add $I$ to the region sets $R_k(C)$ of all cells $C$ contained in region $R$.
```

Figure 4: Algorithm IceCube

Proof. Condition 1 is straightforward from the earlier discussion on credit-based pruning, whereby if $I \notin S_k(C)$, it cannot be frequent in any region R containing C, and hence not worth counting.

For Condition 2, we prove its converse: Let Z be a $(k-1)$-subset of I with $Z \notin R_{k-1}(C)$. Then, by definition of region set, Z is not frequent in any region R containing cell C. So, I cannot be frequent in R because one of its subsets, namely Z, is not frequent in R. Therefore, we do not need to count I in cell C. Conversely, I needs to be counted in cell C if all its $(k-1)$-subsets are elements of $R_{k-1}(C)$. □

3.4 IceCube Example

We now present a simple example to help illustrate the working of the IceCube algorithm. Specifically, consider a cube containing 4 cells, C_1 through C_4, as shown in Figure 5. The cube has 5 associated regions: R_1 containing (C_1, C_2); R_2 containing (C_3, C_4); R_3 containing (C_1, C_3); R_4 containing

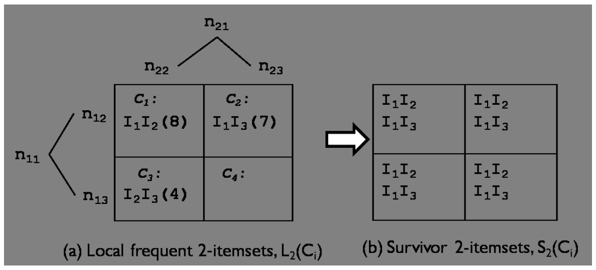
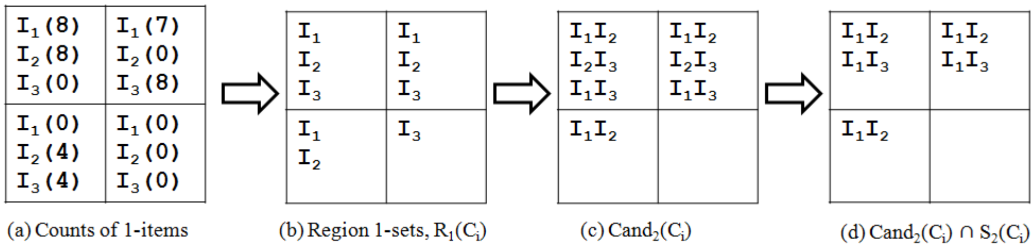


Figure 5: Step 1 of IceCube: Credit-based Survivor Sets



$I_1(8)$ $I_2(8)$ $I_3(0)$
$I_1(7)$ $I_2(0)$ $I_3(8)$
$I_1(0)$ $I_2(4)$ $I_3(4)$
$I_1(0)$ $I_2(0)$ $I_3(0)$
(a) Counts of 1-items
(b) Region 1-sets, $R_1(C_i)$
(c) $Cand_2(C_i)$
(d) $Cand_2(C_i) \cap S_2(C_i)$

Figure 6: Step 2 of IceCube: Region Sets and Cubing

(C_2, C_4); and R_5 containing (C_1, C_2, C_3, C_4). Assume that each individual cell stores 100 transactions and that the mining support threshold is 4%.

For this scenario, the locally frequent 2-itemsets along with their counts are enumerated in Figure 5(a). The credits computed for these itemsets are $credit(I_1I_2, C_1) = 4$; $credit(I_1I_3, C_2) = 3$; $credit(I_2I_3, C_3) = 0$; with all other credits being -1. Aggregating the cell-level credits over regions results in the following non-negative assignments: $credit(I_1I_2, R_1) = 3$; $credit(I_1I_2, R_3) = 3$; $credit(I_1I_2, R_5) = 1$; $credit(I_1I_3, R_1) = 2$; $credit(I_1I_3, R_4) = 2$; and $credit(I_1I_3, R_5) = 0$. Mapping these regions back to the constituent cells produces the survivor itemsets shown in Figure 5(b). Note that itemset I_2I_3, although frequent in cell C_3, is eliminated from being counted in the enclosing regions.

The cell-level counts of the individual items are given in Figure 6(a). Aggregating them generates the following region-level frequent 1-items: $R_1 = \{I_1(15), I_2(8), I_3(8)\}$, $R_2 = \{\}$, $R_3 = \{I_1(8), I_2(12)\}$, $R_4 = \{I_3(8)\}$, $R_5 = \{\}$. Mapping these frequent 1-items back to the cells generates region 1-sets as shown in Figure 6(b). Then, applying the Apriori-gen function generates the cell-level candidates shown in Figure 6(c). Intersecting these candidates with the corresponding survivor sets (Figure 5(b)) delivers the final set of candidates counted by IceCube, shown in Figure 6(d).

Overall, in this example, CellUnion would count a total of 3*4=12 itemsets, whereas IceCube counts only 5.

3.5 Implementation Details

Thus far, we have described the logical construction of the various cube mining algorithms. We now move on to detailing specific aspects related to their implementation of itemset support counting, candidate generation and cubing.

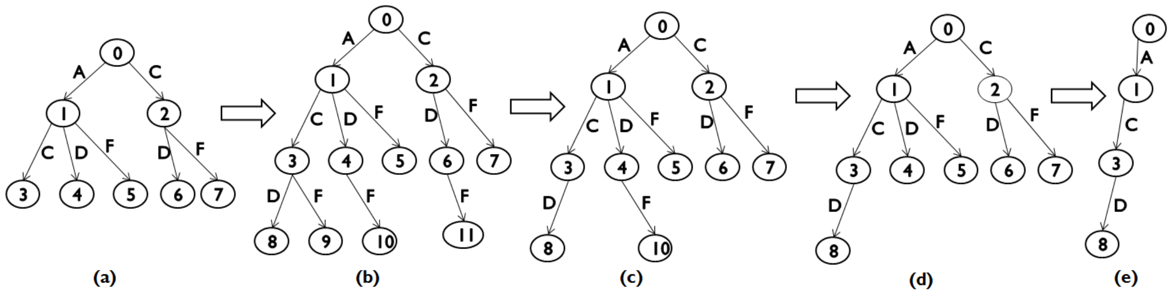


Figure 7: Trie Processing Steps

3.5.1 Itemset Support Counting

As mentioned earlier, we use a memory-resident trie data structure for organizing and counting candidate itemsets, which has become the defacto standard in the ARM literature [4, 3]. Note that, in the case of CellUnion, where candidates of all lengths are counted in one shot, the itemset to be counted could correspond to an *internal* node of the trie. An example trie for candidate 2-itemsets AC, AD, AF, CD, CF is shown in Figure 7(a). In this trie, leaf node 6 represents the itemset CD, node 7 represents itemset CF, and so on.

3.5.2 Generating Candidates in IceCube

The trie structure is used in a standard manner for generating candidate itemsets in the IceCube algorithm. Specifically, for each leaf node N in the trie, all its *right* sibling edges are added as children of N, followed by removing candidates not present in the survivor set, $S_K(C)$. As an example, assume that in pass $k = 3$, a certain cell C has foreign set $F_2(C) = \{AC, AD, AF, CD, CF\}$ and survivor set $S_3(C) = \{ACD, ADF\}$. Initially, a trie is constructed with itemsets from $F_2(C)$ as shown in Figure 7(a). Then, the right sibling edges are added – for instance, two new nodes are added as children of node 3 representing its right siblings, D and F, in Figure 7(b). Finally, nodes 9 and 11 are deleted in Figure 7(c) since their associated itemsets ACF and CDF are not present in $S_3(C)$.

Then, k-candidates whose $(k-1)$-subsets are not all present in the trie are removed. An instance is shown in Figure 7(d) where node 10 corresponding to itemset ADF is removed since DF is not present. This is followed by recursively removing leaf nodes whose depth is less than k, as shown in Figure 7(e) for $k = 3$.

3.5.3 Cubing Algorithm

Given the cell counts of itemsets, the cubing algorithm needs to compute the higher level region counts. A simple approach would be to repeatedly compute these aggregate counts from the constituent atomic cells. However, more efficient cubing algorithms have been proposed ([15, 12]), wherein the region aggregates are computed in a pipelined manner, minimizing the redundant computation.

We adapt these ideas to design an efficient cubing algorithm suited to our objectives. The standard notion of a "view-lattice" is adopted from the data warehousing literature [8], wherein each node in the lattice represents a view constructed from the cross-product of dimension values, and can be computed from its parents in the lattice. An example view-lattice corresponding to the 2D cube of Figure 1 is shown in Figure 8, with the most fine-grained combination, $(quarter, city)$, at the bottom, and the most coarse-grained, $(year, country)$, at the top.

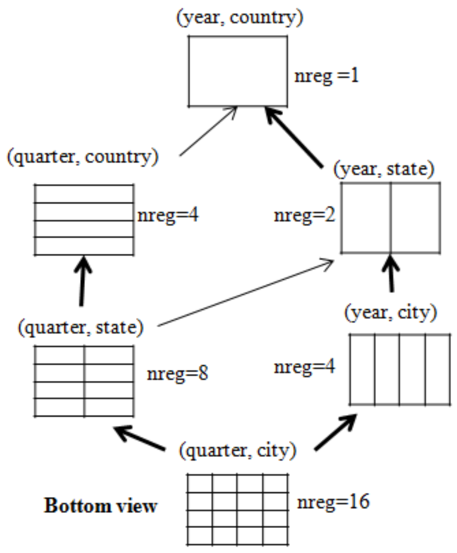


Figure 8: Lattice of views

We convert this lattice into a *tree* by retaining, for each view that can be computed through multiple parents, only the parent with the minimum number of regions. For example, $(year, state)$ can be computed from either $(quarter, state)$ which has 8 regions, or from $(year, city)$ which has 4 regions – we retain only $(year, city)$ due to its lesser number of regions. It is easy to show that our approach is optimal in terms of the number of aggregation (addition) operations required to compute all views. The retained tree edges are shown in boldface in Figure 8.

After converting the lattice to a tree, we read, for each itemset, the cell level counts in the bottom view. Then, we traverse the tree in a DFS manner and calculate each view from its designated parent. The memory allocated to a view can be freed once all its children views have been computed. That means that, at most, the views in an entire path from root to leaf have to be retained simultaneously in memory. So, if we assume that the fanout of any node in the dimension hierarchy is atleast 2, then the number of regions in any view is upper-bounded by half the number of regions in its parent view. This ensures that the maximum memory required by IceCube is within twice the size of the bottom view. Unlike in a traditional OLAP scenario, the bottom view may not be particularly large, since we expect the dimensions to be rolled up to higher levels before we undertake mining for targeted rules. Note that if the bottom view is extremely fine-grained, the rules generated are unlikely to be statistically significant.

If the above expectation is borne out, it is likely that the available memory is far greater than the maximum required for cubing a single itemset. In such situations, instead of cubing a single itemset at a time, we read the counts of multiple itemsets at a time in the bottom view.

4 Experimental Evaluation

In this section, we describe the experimental framework under which the various targeted mining algorithms of Section 3 were evaluated, and sample results obtained under this framework.

Synthetic Data Cube Generation. Our experiments cover both synthetic and real data cubes. For generating synthetic market-basket databases, the IBM Quest Generator [18] is a popular tool. However, it does not include dimension attributes, and we therefore had to augment the generator to provide these additional semantics.

Our cube generator takes the generation parameters listed in Table 1:

Parameter	**Meaning**
$\|D\|$	number of transactions
$\|T\|$	average size of transactions
$\|I\|$	average size of maximal potentially large itemsets
$\|L\|$	number of maximal potentially large itemsets
$items\_cell$	number of unique items to add to each cell
num_reg	number of random regions to consider
$items\_reg$	number of items to add to each random region
n	numDimensions
$l_1, \ldots, l_n$	number of levels in each dimension hierarchy
$f_{i,1}, \ldots, f_{i,l_i}$	fanout at each hierarchy level, $i = 1 \ldots n$

Table 1: Cube Generation Parameters

The synthetic data generation occurs in two phases: In the first phase, the hierarchy in each dimension is created as a tree, starting at the root, with each node $(i : dimension,\ j : level)$ in the tree generating $f_{i,j}$ children. Then, in the second phase, the cell-level data is generated. Here, we have two objectives – each cell should have its own identity and, at the same time, share some similarities with its proximate cells. To achieve this, we use the algorithm shown in Figure 9, wherein Step 1 ensures the distinctness by assigning some items uniquely to individual cells, while Step 2 ensures correlations through sharing of other items across all cells belonging to randomly chosen regions.

We used this augmented generator to generate five synthetic datasets, D1 through D5, each containing ten million transactions, with varying number of dimensions, hierarchies and fanouts. The statistical characteristics of these datasets are shown in Table 2, and cover a wide spectrum of region cardinalities ranging from 87 for D1 to 1458 for D5. While these cube sizes may appear small compared to traditional data warehousing scenarios, we expect that the dimensions will typically be rolled up before targeted mining is initiated. For example, in the *time* dimension, it is unlikely to be useful to mine rules at the lowest granularity such as a *minute*. Instead, rollups to higher levels, such as a *month* or a *quarter*, are more likely to be the norm. Finally, to determine a reasonable support threshold for mining, a thumb rule of $minsup = 0.2 * db\_density$ was used, where $db\_density$ refers to the density of the market-basket data computed over the entire cube.

DBLP Cube Data. Turning our attention to the cube created from real data, here the raw information was sourced from the *inproceedings* records of DBLP's bibliographic database [17]. These records were converted into transactions and a dimension hierarchy was constructed over them. Specifically, the terms in the paper title, authors and author affiliation were taken as the "market-basket" items,

```
Algorithm CubeGen:
Step 1: For $i = 1 \dots N$
        $|D_i|$ = Poisson ($|D|/N$)
        Add $items\_cell$ unique items to set $G_i$
Step 2: For $j = 1 \dots num\_reg$
        Generate a set P of $items\_reg$ unique items.
        Pick a random region R from cube.
        For $i = 1 \dots N$
            If cell $C_i \in R$, then $G_i = G_i \cup P$
Step 3: For $i = 1 \dots N$
    Call IBM Quest data generator to produce $D_i$ transaction database for cell $C_i$ using items in set $G_i$
        with generation parameters $|D_i|, |T|, |I|, |L|, |G_i|$
```

Figure 9: Cube Data Generation

Data Cube	No. trans	Avg length	No. dim	Dim levels	Fan-out	No. cells	No. Regions	Cube Density
D1	10M	8	2	3	3	81	87	0.013
D2	10M	8	2	3	4	256	185	0.023
D3	10M	8	2	3	5	625	336	0.039
D4	10M	8	2	4	3	729	871	0.052
D5	10M	8	3	3	3	729	1468	0.038

Table 2: Synthetic Dataset Characteristics

while the conference and publication year constituted the dimensional attributes. A set of 153 computer science conferences were selected and classified into topic areas such as operating systems, database systems, etc., generating a total of 3366 cells and 1305 regions. The base transactions numbered over 0.2 million, and the effective size of the database was made much larger by replicating this information 50 times. As a result, the final database had in excess of 10 million transactions. A *minsup* value of 5% was used in the mining.

Computational Platform. All our cube mining algorithms were implemented in C++, and their evaluation was conducted on a Sun Ultra 24 quad core machine with 8 GB of RAM, running on the Ubuntu 10 operating system.

4.1 Synthetic Data Results

We present in Figure 10 the log-scale execution times of the various cube mining algorithms, with regard to computing targeted association rules on the five synthetic datasets.

We first observe here that the classical algorithm, TOARM, is extremely inefficient, taking several tens of minutes or even hours to complete its execution. This poor performance is a consequence of the redundancy arising out of computing localized itemsets afresh for each region in the cube, as well as processing a large number of candidates that eventually turn out to be infrequent.

Secondly, we see that the CellUnion algorithm noticeably reduces the computational overheads as compared to TOARM, because it carries out the counting for all regions together, making the subsequent cubing pass very efficient. However, its running time increases sharply with the cube complexity, since the union of frequent itemsets sharply expands with the increased number of cells. This results in an excessive number of candidates being wastefully counted.

Finally, turning to IceCube, we find that it consistently provides the best performance across all the datasets. More importantly, its performance differential with respect to the other algorithms *increases* with cube complexity, with its performance being more than an *order-of-magnitude* better than CellUnion for datasets D3, D4 and D5.

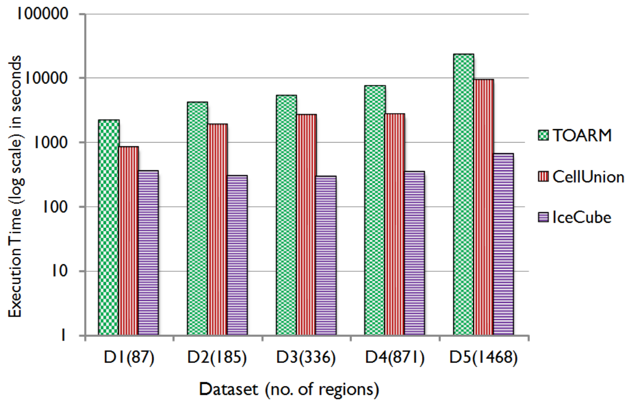


Figure 10: Execution Time for Mining Targeted Rules

Candidates Counted. To better understand the execution time behavior of the various algorithms, we explicitly evaluated the number of candidates that they each counted, normalized to the number counted by TOARM. This statistic is shown on a log-scale in Figure 11, and the interesting observation here is that the number counted by IceCube is *orders of magnitude* less than that of TOARM and, in fact, an order of magnitude less than even that counted by CellUnion for the more complex cubes! This statistic clearly highlights the potent power of the credit-based pruning and count-cube interleaving strategies that form the core of IceCube.

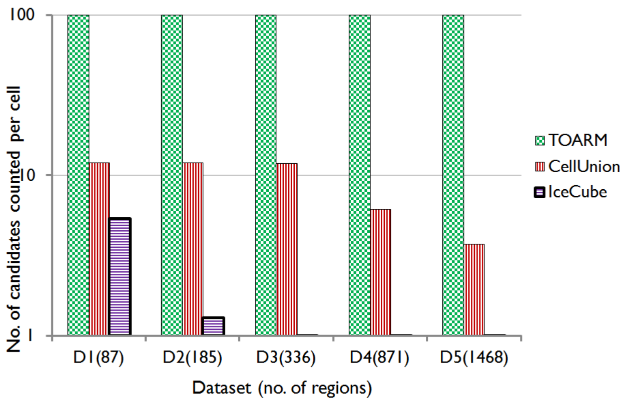


Figure 11: Number of Candidates Counted

Region Processing Time. To assess the practicality of the algorithms in absolute terms, we have shown in Figure 12 the time taken per region across the different datasets. We observe here that for IceCube, the average processing time per region is less than 4 seconds for all datasets and actually goes down to sub-second times for the complex cubes.

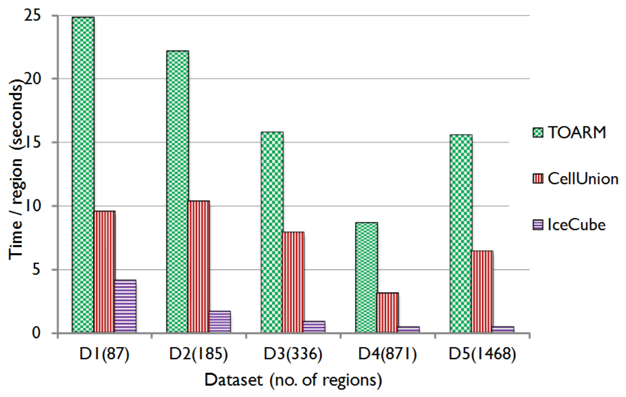


Figure 12: Average Processing Time per Region

Memory Consumption. Finally, while so far only the time aspect was discussed, we also monitored the peak memory consumption overheads of the various algorithms. We found that the memory occupancy of all three algorithms is comparable, IceCube using marginally less than the other algorithms. The consumption was related to the data cube complexity, with D5 requiring a little over 1 GB of space.

4.2 Real Data Results

We now turn our attention to the performance of the targeted mining algorithms on the DBLP-based real data cube. For this scenario, the log-scale execution time profile is shown in Figure 13, and again we observe that there are considerable differences in the performance of the three algorithms. In particular, TOARM takes several hundred seconds, whereas CellUnion brings it down by a factor of about two, and finally, IceCube brings it down even further to well below hundred seconds.

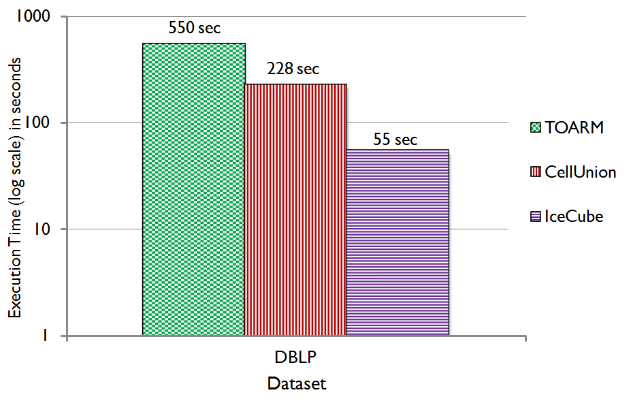


Figure 13: Execution Time (DBLP)

The number of candidates counted by the different algorithms is shown in Table 3, Again, we find a vast gulf in terms of the redundant effort incurred by TOARM and CellUnion as compared to IceCube – specifically, IceCube counts only about a thousand candidates, whereas the other two take one to two orders of magnitude more.

Overall, what the above experiments indicate is that IceCube is able to successfully minimize the amount of redundant work done during cube mining and deliver very good execution time performance,

TOARM	CellUnion	IceCube
122261	11415	1289

Table 3: Candidates Counted (DBLP)

both in absolute and relative terms, while incurring resource overheads commensurate with the other approaches.

Targeted Mining. To quantify the number of targeted patterns that are discovered, we measured the distribution of frequent itemsets over the region space – a sample result for dataset D3 is shown in Figure 14. Here, the rightmost bar (625 cells) corresponds to the entire cube and shows what traditional association rule mining would have produced – in this case, 73 frequent itemsets. But, as we move leftwards in the graph, the region sizes become progressively smaller, and we observe that the number of frequent itemsets increases substantially, reaching an average level of as many as 360 itemsets for regions with five constituent cells. This clearly demonstrates that there are significant data mining patterns that become visible only under localized observation, justifying the motivation for targeted mining.

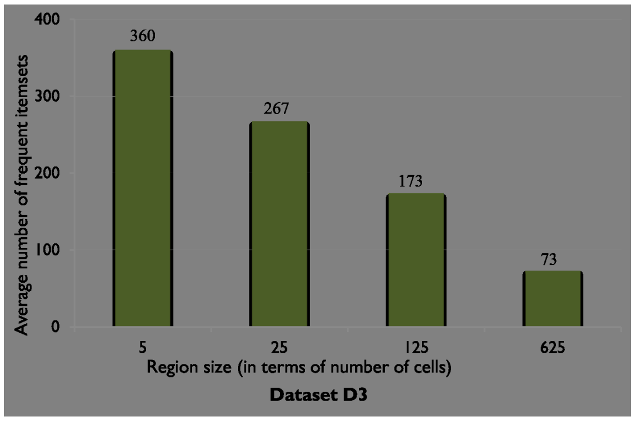


Figure 14: Number of Targeted Itemsets

5 Conclusions

We extended traditional ARM to computing targeted rules in the framework of multi-dimensional data cubes. The primary hurdle here is that since the support thresholds are localized to individual regions, the classical monotonicity properties that are usually leveraged to deliver efficiency no longer hold. We addressed this challenge by developing the IceCube algorithm, which efficiently achieves the new mining objective by bringing together and integrating notions of cubing, count-cube interleaving and credit-based pruning. Our experimental results over a representative range of data cubes indicate that IceCube provides excellent performance, in both absolute and relative terms, as compared to the prior art. The primary underlying reason is that the interleaving and pruning strategies reduce, by an order of magnitude or more, the redundant counting of itemsets, In our future work, we intend to investigate the extension of the ideas proposed here to other data mining patterns.

References

[1] R. Agrawal, T. Imielinski, and A. N. Swami, "Mining association rules between sets of items in large databases," *SIGMOD Conf.*, 1993.

[2] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," *VLDB Conf.*, 1994.

[3] F. Bodon, "A fast apriori implementation," *FIMI Conf.*, 2003.

[4] F. Bodon and L. Ronyai, "Trie: an alternative data structure for data mining algorithms," in *Computers and Mathematics with Applications*, 2003.

[5] B.-C. Chen, L. Chen, Y. Lin, and R. Ramakrishnan, "Prediction cubes," *VLDB Conf.*, 2005.

[6] M. Das, D. P, P. Deshpande, and R. Kannan, "Fast rule mining over multi-dimensional windows," *SDM Conf.*, 2011.

[7] J. Han and Y. Fu, "Discovery of multiple-level association rules from large databases," *VLDB Conf.*, 1995.

[8] V. Harinarayan, A. Rajaraman, and J. D. Ullman, "Implementing data cubes efficiently," in *SIGMOD*, 1996.

[9] T. Imielinski, L. Khachiyan, and A. Abdulghani, "Cubegrades: Generalizing association rules," *Data Mining and Knowledge Discovery*, 6 (3), 2002.

[10] M. Kamber, J. Han, and J. Chiang, "Metarule-guided mining of multi-dimensional association rules using data cubes," *KDD Conf.*, 1997.

[11] R. B. Messaoud, S. L. Rabaséda, O. Boussaid, and R. Missaoui, "Enhanced mining of association rules from data cubes," *DOLAP Conf.*, 2006.

[12] K. A. Ross and D. Srivastava, "Fast computation of sparse datacubes," in *VLDB*, 1997, pp. 116–125.

[13] R. Srikant and R. Agrawal, "Mining generalized association rules," *VLDB Conf.*, 1995.

[14] C.-Y. Wang, S.-S. Tseng, and T.-P. Hong, "Flexible online association rule mining based on multidimensional pattern relations," *Information Sciences*, 176 (12), 2006.

[15] Y. Zhao, P. Deshpande, and J. F. Naughton, "An array-based algorithm for simultaneous multidimensional aggregates," in *SIGMOD Conference*, 1997, pp. 159–170.

[16] H. Zhu, "On-line analytical mining of association rules," Master's Thesis, Simon Fraser University, 1998.

[17] The DBLP computer science bibliography. `http://dblp.uni-trier.de/xml/`

[18] IBM Quest market-basket synthetic data generator. `http://www.cs.nmsu.edu/~cgiannel/assoc\_gen.html`