iPARAS: Incremental Construction of Parameter Space for Online Association Mining[*]

Xiao Qin XQIN@CS.WPI.EDU
Ramoza Ahsan RAHSAN@WPI.EDU
Xika Lin XIKA@WPI.EDU
Elke A. Rundensteiner RUNDENST@WPI.EDU
Matthew O. Ward MATT@WPI.EDU
Computer Science Department, Worcester Polytechnic Institute
100 Institute Road, Worcester MA, USA.

Editors: Wei Fan, Albert Bifet, Qiang Yang and Philip Yu

Abstract

Association rule mining is known to be computationally intensive, yet real-time decision-making applications are increasingly intolerant to delays. The state-of-the-art *PARAS*[1] solution, a parameter space framework for online association mining, enables efficient rule mining by compactly indexing the final ruleset and providing efficient query-time redundancy resolution. Unfortunately, as many association mining models, *PARAS* was designed for static data. Modern transaction databases undergo regular data updates that quickly invalidating existing rules or introducing new rules for the *PARAS* index. While reloading the *PARAS* index from scratch is impractical, as even upon minor data changes, a complete rule inference and redundancy resolution steps would have to be performed. We now propose to tackle this open problem by designing an incremental parameter space construction approach, called *iPARAS*, that utilizes the previous mining result to minimally adjust the ruleset and associated redundancy relationships. *iPARAS* features two innovative techniques. First, *iPARAS* provides an end-to-end solution, composed of three algorithms, to efficiently update the final ruleset in the parameter space. Second, *iPARAS* designs a compact data structure to maintain the complex redundancy relationships. Overall, *iPARAS* achieves several times speed-up on parameter space construction for transaction databases comparing to the state-of-the-art online association rule mining system *PARAS*.

Keywords: Association Rule Mining, Parameter Space Model, Redundancy Resolution

1. Introduction

With the advent of computers and means for mass digital storage, companies increasingly gather huge data from various sources. Mining useful information and helpful knowledge from large databases has evolved into an important research area in recent years. Mining of associations, correlations and other meaningful patterns within the dataset is critical for applications ranging from market basket analysis and web usage mining to fraud detection and catastrophic weather forecasting. However, association rule mining is known to be com-

* This work was partly supported by the National Science Foundation under grants NSF IIS-1117139, NSF CRI-1305258, NSF IIS-0812027 and NSF CCF-0811510.

1. *PARAS* system can be accessed at http://paras.cs.wpi.edu.

putationally expensive. Mining systems with lagged responses risk losing a user's attention, more importantly, are often unacceptable in mission critical applications. Worse yet, association rule mining algorithms are parametrized, meaning that data analysts often need to perform numerous successive trial and error iterations and compare the results produced by varying parameter configurations until a satisfactory result is found. Poor selection of parameter values may lead to failure in finding genuine association rules.

In an online mining system, an end user should be able to rapidly iterate over possibly a variety of parameter settings and examine the corresponding association rules matching these settings in an interactive session (Lin et al. (2013)). To provide fast response time essentially for a truly interactive experience, the expensive computations must be performed a priori as in a precomputation step. Furthermore, effective data structures and access methods must be designed to support instant retrieval of the rules for any desired setting. Put differently, not only computation overhead for rule computation but also excessive I/O costs for rule look-up must be avoided at run-time.

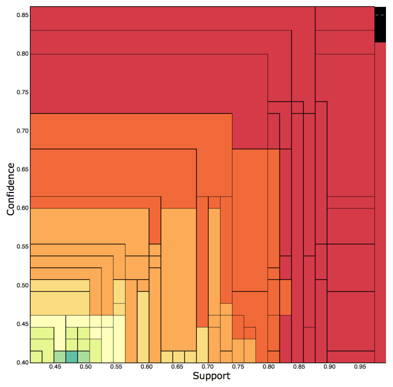

Figure 1: 2-D Parameter Space

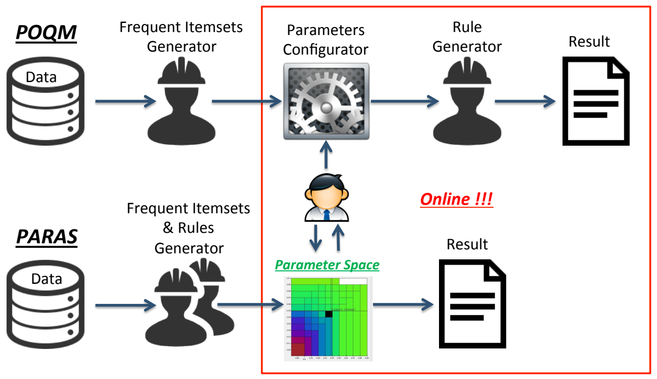

Figure 2: State-of-the-art

In general, association rule mining is a two-step process: in the first step, the frequent itemsets are found, from which association rules are induced in the second step. As shown in Figure 2, *POQM* (Aggarwal and Yu (2001)) precomputes the *primary itemsets* with a low minimum support and stores them in a lattice structure, called *Adjacency Lattice*, in an offline step. At runtime, rules are then generated from the lattice upon the user request for a specified parameter setting. Further, redundant rules can be eliminated by request during this online generation process. Unfortunately, the change of even a single parameter requires user to run rule generation algorithm again from scratch.

PARAS (Lin et al. (2013)) extends this approach by including rule generation into the offline process. Not only are the rules that satisfy a low *primary support* and *primary confidence* precomputed, but also the redundancy relationships among the rules are encoded. A parameter space, in the context of rule mining, consists of an n-dimentional space of parameters such as *support* and $confidence$. The precomputed rules are compactly indexed in this *parameter space* by their parameters for subsequent interactive rule exploration. *PARAS* has been shown to outperform the state-of-the-art online association rule mining systems by several orders of magnitude in terms of the online query processing speed.

However, both frameworks are designed to handle static datasets only, rendering them ineffective for transaction databases. In transaction databases, new transactions are appended as time advances. This may introduce new rules as well as invalidate some existing rules. Worse yet, this might lead to dramatic changes of corresponding redundancy relationships among the rules over time. We observed, and verified in the *retail* dataset (Brijs et al. (1999)) that daily transaction updates may at times affect a small portion of the dataset as well as the precomputed parameter space. Simply re-mining from scratch may waste a huge amount of computational resources and cause tremendous delays. Thus, the maintenance of the parameter space for dynamic datasets is a critical problem to be tackled.

Motivating Example: By analyzing daily online transaction logs, a retail analyst at Amazon may identify products that are frequently purchased together. Such products can be placed together on amazon.com, made into bundled offers or used for recommendations when users search for one of the products. The system can incorporate the updates made by the customers' daily transactions and can seamlessly update the existing patterns as well as identify new patterns in its e-commerce transaction store.

The Proposed *iPARAS* Approach. An interactive parameter space system, capable of not only answering mining queries at near real time but also accommodating data updates efficiently, ensures that insights and results are always delivered based on the most up-to-date knowledge for fresh decision making. To dynamically update the pre-stored information, *POQM* could make use of existing incremental frequent itemset mining (FIM) algorithms (Cheung et al. (1996),Ayan et al. (1999),Cheung and Zaïane (2003),Koh and Shieh (2004), Leung et al. (2005)). We propose to extend the *PARAS* framework with strategies for maintaining the parameter space up-to-date even over dynamic transaction databases. Our proposed model *iPARAS* utilizes the previously computed results indexed in parameter space to update the rules and the redundancy relationships in the parameter space. *iPARAS* features two innovative techniques. First, *iPARAS* provides an end-to-end solution, composed of three algorithms, to efficiently update the rules maintained in the parameter space. Second, *iPARAS* designs a compact data structure to maintain the complex redundancy relationships. Overall, *iPARAS* achieves several times speed-up for the parameter space construction for transaction databases compared to the state-of-the-art online association rule mining system *PARAS*.

2. Preliminaries

2.1. Foundation: The Parameter Space Model

The parameter space model (Lin et al. (2013)) forms the foundation of our proposed *iPARAS* framework. A parameter space as depicted in Figure 1, in the context of rule mining, consists of an n-dimensional space of parameters such as *support* and $confidence$. A parameter setting is represented as a point located within this space. The key observation is that many rules may map to the same location and thus can be compactly indexed by their location. This finite number of points divide the parameter space into several non-overlapping regions, called ***stable regions***. The ruleset valid for any possible location within a stable region remains unchanged, whereas rulesets valid for two locations not in the same stable region are guaranteed to be distinct. The parameter space model supports instant response to user's mining request and reveals the overall distribution of rules within the space of interesting parameters so that trial-and-error selection of parameters can be avoided. In addition, to filter out redundant rules for presenting succinct mining results to users, the redundant relationships among the rules in parameter space are precomputed and encoded using a compact representation, called ***dominating location***. In the online phase, redundant rules can be quickly identified and filtered out by comparing their dominating locations to user specified parameters.

2.2. iPARAS Framework Overview

To efficiently update rules in the parameter space upon changes of the raw data, while preserving instant responsiveness, we design the *iPARAS* framework, as depicted in Figure 3. *iPARAS* consists of two phases: (a) offline parameter space maintenance and (b) online rule exploration. *iPARAS* adapts *PARAS* for the initial parameter space construction from raw data and adopts *PARAS* to support the powerful online interactive rule exploration.

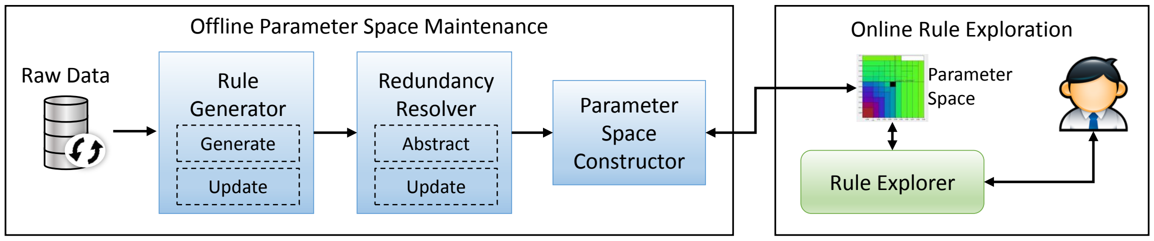

Figure 3: The iPARAS framework overview

The **Rule Generator** operates in two modes, namely, **Generate** and **Update**. The **Generate** mode is designed for processing the full dataset initially. In general, the association rule mining is a two-step process: (a) *Frequent Itemset Mining* and (b) *Rule Induction*. To perform the above tasks for rule generation, we adapt *FP-Growth* (Han et al. (2000)) and *GenerateRules* (Aggarwal and Yu (2001)) respectively. The second mode, **Update**, has been added to process the future batch updates. To efficiently update the parameter space, we design a novel incremental frequent itemset mining algorithm, called *iFIM* and a rule updating technique.

The **Redundancy Resolver** captures and maintains the redundancy relationships among the rules such that, if desired by the user, redundant rules can be efficiently filtered out for any given parameter settings. We optimize the redundancy resolution introduced in *PARAS* by leveraging our proposed *Adjacency Index*. In **Abstract** mode, this improved strategy is used to compute the dominating locations for the rules generated from the initial dataset. The **Update** mode updates the redundancy relationships among the rules in the adjusted parameter space.

The **Parameter Space Constructor** takes the up-to-date ruleset and the corresponding redundancy information to construct the parameter space. After the construction is completed, the user then is able to explore the parameter space with instant responses through the **Rule Explorer** to find interesting rules.

3. Offline Parameter Space Maintenance

3.1. Rule Generation & Updating

We introduce our **Rule Generator** module of *iPARAS* that provides an efficient solution for rule induction and updating. The first task of this module is to generate and update frequent itemsets. Apriori (Agrawal and Srikant (1994)), FP-tree (Han et al. (2000)), Eclat (Zaki et al. (1997)) and their variations could all be used to generate frequent itemsets in a database. However, it has been proven (Hunyadi (2011), Borgelt) that the FP-tree solution outperforms Aprori because it requires fewer scans over the database and also the execution time is not wasted in re-producing candidate items at each step. As FP-tree can be applied to larger datasets and reduces I/O computation costs significantly, *iPARAS* now adopts this method to generate frequent itemsets for the original dataset. Thereafter, with each update it is maintained to accommodate new transactions efficiently. Frequent itemset mining involves two phases. In the first phase the FP-tree is constructed while the second phase, known as FP-Growth (Han et al. (2000)), derives the frequent itemsets from FP-tree.

Table 1: (a) 10 transactions, (b) frequency of each item and (c) sorted transactions by the frequency order.

TID	Items
1	a,c,d,f
2	a,d,f
3	a,d,f
4	b,d,f,g
5	b,c,f,e
6	a,b,d
7	a,e,h
8	b,d,f,i
9	a,e,j
10	f,b

Item	Freq
a	6
b	5
c	2
d	6
e	3
f	7
g	1
h	1
i	1
j	1

TID	Items
1	f,a,d,c
2	f,a,d
3	f,a,d
4	f,d,b
5	f,b,e,c
6	a,d,b
7	a,e
8	f,d,b
9	a,e
10	f,b

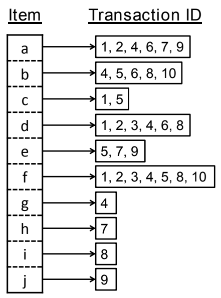

Figure 4: Inverted Index constructed on transactions in Table 1

Consider the example of database given in Table 1 with 10 transactions. The minimum absolute support threshold is 2. In the first pass, the database is scanned to get the frequency counts of each item. Items are sorted in descending order of frequency. The transactions with now sorted frequent items are shown in Table 1 (c). The header table is built which contains the sorted frequent items and their frequency and link to the first node in the FP-tree. FP-tree and header table for the database of Table 1 is shown in Figure 5.

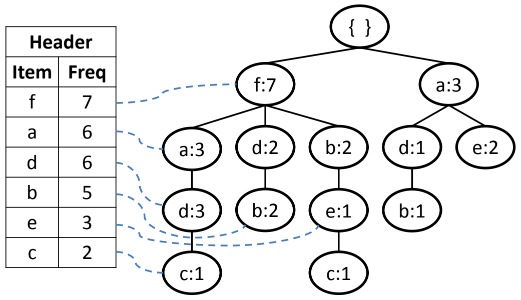

Figure 5: Header Table and FP-Tree

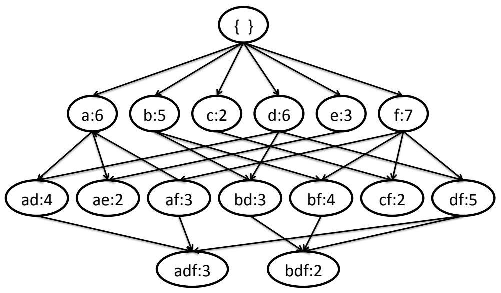

Figure 6: Adjacency Lattice

After the FP-tree has been constructed, the mining process FP-Growth is executed which finds all frequent itemsets directly from the FP-tree. It is a recursive process that works bottom up according to the header table. A conditional FP-tree is generated for each item and frequent itemsets are derived recursively from conditional FP-trees. Once frequent items are generated, they are represented in a *Adjacency Lattice* proposed in (Aggarwal and Yu (2001)) for rule generation process (see Section 3.1.5). The Lattice built on the frequent items for the example database is illustrated in Figure 6.

3.1.1. Incremental Frequent Itemset Mining (*iFIM*)

The FP-tree on the original database must be built as described in the previous section before newly incoming transactions can be processed. During the construction of the FP-tree the inverted index is created which contains the transaction number for each item. When new transactions are added, *iFIM* processes them to update the tree in such a manner that the resulting tree has a compact structure and assures that the frequent itemsets are generated from tree efficiently. The algorithm for updating the FP-tree is introduced in Algorithm 1, with the *Sort Header* procedure adopted from (Koh and Shieh (2004)).

Algorithm 1: iFIM Algorithm

```
Input: $FPTree$, $inverted\_index$, $header$, $suppI$, $D^+$
Output: {$FPTree^+$} // Updated FPTree
begin
    foreach Transaction $T_i$ in $D^+$ do
        foreach Item $a_j$ in $T_i$ do
            if $a_j$ present in header then
                header[$a_j$].count++;
            else
                rescan_vector[$a_j$].count ++;
    $delete\_Item(FPTree, header, suppI)$;
    $rescan\_Item(header, inverted\_index, rescan\_vector, suppI)$;
    $sort\_header(FPtree, header)$;
    foreach Tansaction $T_i$ in $D^+$ do
        process the transaction
```

Algorithm 2: Sort Header

```
Input: $FPTree$, $header$
begin
    Apply bubble sort on header;
    if item X, Y in header are to be exchanged then
        foreach node X, Y in FPtree do
            if node X.count > node Y.count then
                insert new node X' as child of X parent;
                X'.count=node X.count - node Y.count;
                X'.children= X.children- node Y;
            else
                node X.count=node y.count;
            exchange parent & child links of node X and Y;
```

Algorithm 3: Rescan Item

```
Input: $header$, $inverted\_index$, $rescan\_vector$, $suppI$
begin
    foreach Item $a_i$ in rescan_vector do
        if rescan_vector[$a_i$].count >= suppI then
            get transaction number from $inverted\_index$ ;
            insert item $a_i$ in header;
            process the transaction;
```

Algorithm 4: Delete Item

```
Input: $FPTree$, $header$, $suppI$
begin
    foreach Item $a_i$ in header do
        if header[$a_i$].count < suppI then
            foreach node X corresponding to item $a_i$ do
                assign node X children to X parent;
                if parent has two child with same item then
                    merge nodes and add node counts;
                delete item $a_i$ from header;
                delete node X from tree;
```

3.1.2. Incremental Frequent Itemset Mining Example

Table 2: The three new transactions and frequency count of each item in new and updated database.

TID	Items
11	d,f,g
12	d,g,i
13	f,h,j

Item	a	b	c	d	e	f	g	h	i	j
Supp in D'	0	0	0	2	0	2	2	1	1	1
Supp	6	5	2	8	3	9	3	2	2	2

Let D be the database shown in Table 1 and let minimum support threshold be 2. For the original database the *inverted_index* is built (see Figure 4). Sorted transactions according to frequencies is shown in Table 1 and corresponding FP-tree in figure 5. Items f, a, b, d, c and e are frequent in D. Let D^+ as shown in Table 2 contain 3 transactions. Let $Incremental support$ ($suppI$) be 3. The new transactions are scanned to get the counts of each item. The header table is updated and update status of each changed item is set. The updated count of each item is determined. In Figure 7, item c's updated support is below threshold so it is deleted.

Items that have become frequent after the update yet were infrequent before are processed next. Item g's updated support is now above the threshold, but was not present in the original FP-tree. It is inserted in *rescan_vector* and header table. Transactions containing any of the items in *rescan_vector* are obtained from *inverted_index*. Only those selected transactions are indexed rather than scanning the whole database again. This avoids unnecessarily scan of the whole database again. For item g only transaction 4 needs to be rescanned. After deletion of item c and rescanning item g, the resulting tree is depicted in Figure 7.

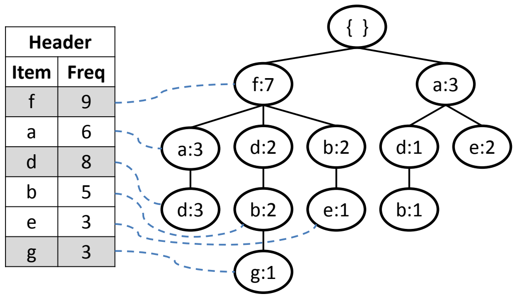

Figure 7: The header table and FP-tree after deleting item c and rescanning transactions.

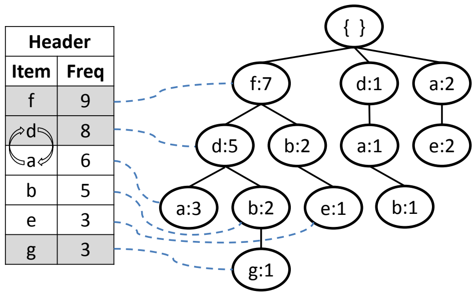

Figure 8: The header table and FP-tree after exchange process.

Next, bubble sort is applied on the header table. In our running example in Figure 8, Item a is exchanged with Item d. The nodes in FP-tree are exchanged and merged accordingly. The items in each new transactions are ordered according to the updated descending order of frequencies of items. They are then processed in same way as FP-tree is constructed originally to get updated FP-tree as shown in figure 9.

3.1.3. Modified Growth Algorithm

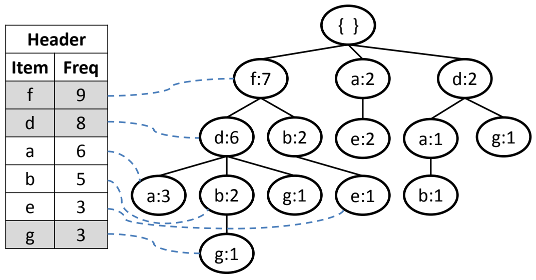

Figure 9: Final header table and FP-tree after all new transactions are processed.

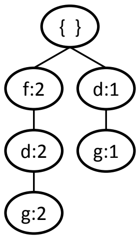

Figure 10: Conditional FP-tree for item g.

We design a modified growth function that generates frequent itemsets from this updated FP-tree avoiding to generate all frequent itemsets from scratch. It recursively processes those items in the header table whose counts have been updated and ignores those that are not modified.

Lemma 1 *If the candidate itemset X is not affected by the current update over the database, then no extension* [2] *$X + I_j$ of X, where $|I_j| > |I_k|$ for any I_k in X must be changed.*

According to Lemma 1, for any item whose count is not changed by current update, any itemset in which it occurs cannot have changed. Thus it can be ignored. A conditional FP-tree is generated for each modified frequent item. From this tree, frequent itemsets are recursively derived. The conditional FP-tree of each item in turn only contains the modified items thus reducing the size of the conditional tree. Consider the updated FP-tree shown in Figure 9. Conditional FP-tree for item g is shown in Figure 10 where conditional FP-tree does not contain item b. The modified growth function then modifies the *Adjacency Lattice* to reflect the changes for each updated frequent itemset. Any frequent itemset that is below the threshold is removed, while new itemsets are added into the lattice. The updated lattice is shown in Figure 12.

3.1.4. Adjacency Index

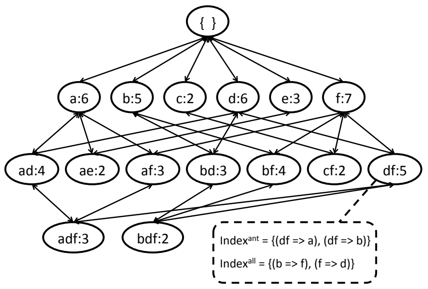
{ }
a:6
b:5
c:2
d:6
e:3
f:7
ad:4
ae:2
af:3
bd:3
bf:4
cf:2
df:5
adf:3
bdf:2
Index^ant = {(df => a), (df => b)}
Index^all = {(b => f), (f => d)}

Figure 11: Adjacency Index L

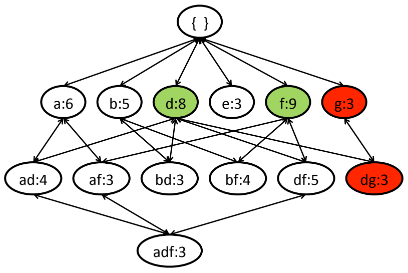

Figure 12: Updated Adjacency Index L'

In this section, we describe our proposed data structure *Adjacency Index* which extends the capabilities of *Adjacency Lattice* (Aggarwal and Yu (2001)) to facilitate rule updating

2. Given a set of items I, an itemset $X + Y$ of items in I is said to be an extension of the itemset X if $X \cap Y = \emptyset$ (Agrawal et al. (1993))

and redundancy resolution in *iPARAS*. *Adjacency Lattice* is a compact data structure to store the frequent itemsets. It is also used for association rule induction. An itemset X is said to be adjacent to an itemset Y, if X can be obtained from Y by adding just one item. In that case, X and Y are connected by a one-directional link from Y to X. Each node in the *Adjacency Lattice* represents a frequent itemset. Our proposed *Adjacency Index* structure extends the capability of the *Adjacency Lattice*. As shown in Figure 11, in each node which represents an itemset X, there are two embedded indices, namely $Index^{ant}$ and $Index^{all}$. $Index^{ant}$ indexes the rules whose $Antecedents \equiv X$, while $Index^{all}$ indexes the rules with $(Antecedents \cup Consequents) \equiv X$. In addition, the one-directional links among the itemsets are replaced by bi-directional links to support both top-down and bottom-up traversal functions.

3.1.5. Rule Induction for Initial Dataset

Table 3: Association Rule Table

Rule	S	C	Rule	S	C	Rule	S	C
$R_1 \equiv (a \Rightarrow d)$	4	4/6	$R_{10} \equiv (d \Rightarrow af)$	3	3/6	$R_{19} \equiv (f \Rightarrow c)$	2	2/7
$R_2 \equiv (a \Rightarrow df)$	3	3/6	$R_{11} \equiv (d \Rightarrow b)$	3	3/6	$R_{20} \equiv (f \Rightarrow d)$	5	5/7
$R_3 \equiv (a \Rightarrow e)$	2	2/6	$R_{12} \equiv (d \Rightarrow bf)$	2	2/6	$R_{21} \equiv (ad \Rightarrow f)$	3	3/4
$R_4 \equiv (a \Rightarrow f)$	3	3/6	$R_{13} \equiv (d \Rightarrow f)$	5	5/6	$R_{22} \equiv (af \Rightarrow d)$	3	3/3
$R_5 \equiv (b \Rightarrow d)$	3	3/6	$R_{14} \equiv (e \Rightarrow a)$	2	2/3	$R_{23} \equiv (bd \Rightarrow f)$	2	2/3
$R_6 \equiv (b \Rightarrow df)$	2	2/6	$R_{15} \equiv (f \Rightarrow a)$	3	3/7	$R_{24} \equiv (bf \Rightarrow d)$	2	2/4
$R_7 \equiv (b \Rightarrow f)$	4	4/6	$R_{16} \equiv (f \Rightarrow ad)$	3	3/7	$R_{25} \equiv (df \Rightarrow a)$	3	3/5
$R_8 \equiv (c \Rightarrow f)$	2	2/2	$R_{17} \equiv (f \Rightarrow b)$	4	4/7	$R_{26} \equiv (df \Rightarrow b)$	2	2/5
$R_9 \equiv (d \Rightarrow a)$	4	4/6	$R_{18} \equiv (f \Rightarrow bd)$	2	2/7			

The second task of **Rule Generator** is to induce and update rules from the generated itemsets. For the **Generate** mode, we adapt the *GenerateRules* (Aggarwal and Yu (2001)) rule induction algorithm and leverage the *Adjacency Index* L to generate the ruleset $\{R\}$ from the initial dataset. The rule generation algorithm forms a rule based on the examination of two itemsets where one is a subset of another. Consider two itemsets X and Y, where $X \subset Y$, if $\frac{Y.support}{X.support}$ is larger than the *primary confidence* threshold, then $R \equiv (X \Rightarrow (Y - X))$ is generated and indexed by $Index^{ant}_{X}$ and $Index^{all}_{Y}$ respectively. To find out all possible pairs to form the rules, for each itemset Y, the algorithm $GenerateRule(Y)$ performs a *Depth-First Search* (DFS) to traverse its ancestors $\{X_i\}$. Any item in $\{X_i\}$ is a subset of Y. Therefore, $\{X_i\}$ is sufficient for Y to form rules where X_i is the *Antecedents*. This process is performed on each itemset of *Adjacency Index* L. We modify this algorithm such that whenever a rule is formed, this rule will be immediately indexed by $Index^{ant}_{Antecedents}$ and $Index^{all}_{(Antecedents \cup Consequents)}$. Consider the itemset "bd" on L. "bdf" is its only descendant. When the modified algorithm $GenerateRule(\text{“}bd\text{”})$ generates the rule $R = (bd \Rightarrow f)$, R will be indexed by $Index^{ant}_{\text{“}bd\text{”}}$ and $Index^{all}_{\text{“}bdf\text{”}}$ in preparation for *iPARAS* rule updating algorithm and redundancy resolution.

3.1.6. Rule Updating for Batch Update

Table 4: Updated Association Rule Table

Rule	S	C	Rule	S	C	Rule	S	C
$R_1 \equiv (a \Rightarrow d)$	4	4/6	$R_{10} \equiv (d \Rightarrow af)$	3	3/8	$R_{20} \equiv (f \Rightarrow d)$	5	5/9
$R_2 \equiv (a \Rightarrow df)$	3	3/6	$R_{11} \equiv (d \Rightarrow b)$	3	3/8	$R_{21} \equiv (ad \Rightarrow f)$	3	3/4
$R_4 \equiv (a \Rightarrow f)$	3	3/6	$R_{13} \equiv (d \Rightarrow f)$	5	5/8	$R_{22} \equiv (af \Rightarrow d)$	3	3/3
$R_5 \equiv (b \Rightarrow d)$	3	3/6	$R_{15} \equiv (f \Rightarrow a)$	3	3/9	$R_{25} \equiv (df \Rightarrow a)$	3	3/5
$R_7 \equiv (b \Rightarrow f)$	4	4/6	$R_{16} \equiv (f \Rightarrow ad)$	3	3/9	$R_{27} \equiv (d \Rightarrow g)$	3	3/8
$R_9 \equiv (d \Rightarrow a)$	4	4/8	$R_{17} \equiv (f \Rightarrow b)$	4	4/9	$R_{28} \equiv (g \Rightarrow d)$	3	3/3

In Section 3.1.1, we demonstrated how to update the *Adjacency Index* L into L' for a given batch update. The operations performed on L are *Delete*, *Insert* and *Update*. Each of these operations will correspondingly trigger certain actions in our proposed rule updating algorithm. In the *Adjacency Index* L, if an itemset X is:
Deleted: All the rules in $Index_X^{all}$ should be deleted. Because all rules indexed by $Index_X^{all}$ consist of X. For instance, as indicated in Figure 12, “cf” has been deleted from L. It is obvious that the two rules indexed by $Index_{“cf”}^{all}$ which are $c \Rightarrow f$ and $f \Rightarrow c$ should be deleted from $\{R\}$.
Inserted: Similar to the rule generation algorithm $GenerateRule(Y)$, $AddRule(Y)$ runs a DFS to traverse Y's ancestors $\{X_i\}$. The newly generated rules conform to the template $R_i \equiv (X_i \Rightarrow (Y - X_i))$ and they will be immediately indexed on L'. For example, for the newly inserted itemset “dg” in L', two new rules $d \Rightarrow g$ and $g \Rightarrow d$ will be added to $\{R\}$.
Updated: The rules originally formed by X and its descendants $\{Y_i\}$ may be updated or expired due to the change of their *support* and $confidence$ values. It is also possible to introduce new rules for some $R \equiv (X \Rightarrow (Y_i - X))$ whose original $confidence$ value was not larger than the *primary confidence* value. Similar to the $GenerateRule(X)$ algorithm, $UpdateRule(X)$ traverses X's descendants to examine the above cases and make changes to $\{R\}$. For instance, the support of “d” being increased will affect the $confidence$ of R_9, R_{10}, R_{11} and R_{13} in Table 3.

3.2. Redundancy Resolution in iPARAS

Aggarwal and Yu (2001) defined redundancy relationships among rules, such that redundant rules, if desired, may be filtered out for presenting succinct result to the user. In particular, redundancy relationships can be of two types , as defined below:

Definition 1 ***Simple Redundancy:*** *Let $R \equiv ((A_1 : A_n) \Rightarrow (C_1 : C_m))$ be a rule. All rules that **simple dominate** it conform to the template $R^{\gg sim} \equiv (((A_1 : A_n) - (A_v : A_w)) \Rightarrow ((A_v : A_w) \cup (C_1 : C_m)))$ where $(A_v : A_w) \subset (A_1 : A_n)$. The rule R is **simple redundant** with respect to the rule $R^{\gg sim}$.*

Definition 2 ***Strict Redundancy:*** *Let $R \equiv ((A_1 : A_n) \Rightarrow (C_1 : C_m))$ be a rule. All rules that **strict dominate** it conform to the template $R^{\gg str} \equiv (((A_1 : A_n) - (A_v : A_w)) \Rightarrow ((A_v : A_w) \cup (C_1 : C_{m+e})))$ where $(A_v : A_w) \subset (A_1 : A_n)$ and $(C_1 : C_m) \subset (C_1 : C_{m+e})$. The rule R is **strict redundant** with respect to the rule $R^{\gg str}$.*

Consider $R_{22} \equiv (ad \Rightarrow f)$ and $R_2 \equiv (a \Rightarrow df)$ in Table 3. By Definition 1 above, R_2 is simple dominating R_{22}. The *support* values of them are guaranteed to be the same because $R^{\ll sim}$ always consists of the same items as $R^{\gg sim}$. Consider $R_5 \equiv (b \Rightarrow d)$ and $R_6 \equiv (b \Rightarrow df)$ in Table 3. By Definition 2 above, R_6 is strict dominating R_5. Note that both *support* and *confident* values of $R^{\ll str}$ are greater or equal to the values of $R^{\gg str}$.

The rules produced in the parameter space may contain redundancies. However, **redundancy relationship is a runtime phenomenon**. In other words, the redundancy among rules depends on runtime input parameters, rendering it impossible to eliminate a rule as redundant at an offline step. For instance, let R_i be the only rule that dominates R_j in the parameter space, R_j is said to be redundant if and only if both of them exist in the ruleset that is generated based on a user specified parameter setting. While eliminating redundant rules can only be performed at runtime, the system must isolate as much as possible the redundancy relationships inside the parameter space in the preprocessing phase. Moreover, database updates may trigger changes of the redundancy information, hence the need for an updating mechanism to identify those changes. In the parameter space model, redundancy information can be abstracted and updated compactly as an offline step by utilizing the certain properties (Lin et al. (2013)) of the redundancy relationships. By leveraging the *Adjacency Index*, our proposed redundancy abstraction method is exponentially faster by a factor of approximately 1.5^n comparing to the state-of-the-art *PARAS* approach, where n is the number of unique items in D.

3.2.1. Redundancy Abstraction in PARAS

Lin et al. (2013) identified a compact representation of the rule redundancies in the parameter space model, namely, the ***simple dominating location*** and the ***strict dominating location***. Given a parameter setting, to determine whether or not a candidate rule R is redundant, it is sufficient to compare it with only two dominating locations instead of an exponential number of dominating rules. The ***simple*** and the ***strict dominating location*** are described below.

Lemma 2 ***Simple Dominating Location****: For each simple dominated rule $R^{\ll sim}$, the set of simple dominating rules $\{R^{\gg sim}\}$ contains a rule $R_i^{\gg sim}$ closest to the dominated rule $R^{\ll sim}$, such that $\forall R_k^{\gg sim} \in \{R^{\gg sim}\}$, where $k \neq i$, $R_i^{\gg sim}.conf \geq R_k^{\gg sim}.conf$. The location of rule $R_i^{\gg sim}$ is called the* ***simple dominating location*** *of $R^{\ll sim}$, denoted by $l^{\gg sim}$.*

For a $R \equiv ((A_1 : A_n) \Rightarrow (C_1 : C_m))$, Lin et al. (2013) revealed that $l^{\gg sim}$ can be obtained from the candidate rules which conform to the template $R^{\gg sim} \equiv (((A_1 : A_n) - A_i) \Rightarrow (A_i \cup (C_1 : C_m)))$. Since R and its simple dominating ruleset $\{R^{\gg sim}\}$ share the same *support* value, the goal here is to find out which candidate has the biggest *confidence* value. We know that the smaller the *antecedents.support*, the larger the *R.confidence*. So the problem has been transferred into the problem to find the smallest *support* value from $\{((A_1 : A_n) - A_i).support\}$. This takes $O(n)$. In the *Adjacency Index*, $\{((A_1 : A_n) - A_i)\}$ are the parents of $(A_1 : A_n)$.

Lemma 3 ***Strict Dominating Location****: For each strict dominated rule $R^{\ll str}$, the set of strict dominating rules $\{R^{\gg str}\}$ contains a rule $R_i^{\gg str}$ closest to the dominated rule $R^{\ll str}$, such that $\forall R_k^{\gg str} \in \{R^{\gg str}\}$, where $k \neq i$, $R_i^{\gg str}.supp \geq R_k^{\gg str}.supp$* ***and*** *$R_i^{\gg str}.conf \geq$*

$R_k^{\gg str}.conf$. *The location of rule* $R_i^{\gg str}$ *is called the* ***strict dominating location*** *of* $R^{\ll str}$*, denoted by* $l^{\gg str}$.

For a $R \equiv ((A_1 : A_n) \Rightarrow (C_1 : C_m))$, Lin et al. (2013) showed that $l^{\gg str}$ can be obtained from the candidate rules which conform to the template $R^{\gg str} \equiv ((A_1 : A_n) \Rightarrow ((C_1 : C_m) \cup C_i))$. Since R and its strict dominating ruleset $\{R^{\gg sim}\}$ share the same *antecedents*, the goal here is to find out which candidate has the biggest *support* value. So the problem has been mapped to the problem of finding the biggest *support* value from $\{((A_1 : A_n) \cup (C_1 : C_m) \cup C_i).support\}$. This takes $O(e)$ where e is the number of children of $((A_1 : A_n) \cup (C_1 : C_m))$ in the *Adjacency Index*.

3.2.2. Optimized Redundancy Abstraction in iPARAS for Initial Dataset

For each rule in the parameter space, *PARAS* takes linear time to obtain its dominating locations. However, how many rules are in the parameter space? Let n be the number of unique items in D, then the maximum number of frequent itemsets is 2^n and the maximum number of rules is $3^n - 2^n + 1$. Even for some small datasets such as *Chess* and *Mushroom* (Newman and Asuncion (2007)), where the number of unique items n are 75 and 119, an immense amount of rules exist in the parameter space, resulting in the high computational costs for abstracting redundancy information. Fortunately, we made the important observation that some rules share exactly the same computation in the process of redundancy abstraction. This provides an optimization opportunity.

Lemma 4 *If* $R_i.antecedents \equiv R_j.antecedents$, $\{R_i^{\gg sim}\} \neq \emptyset$ *and* $\{R_j^{\gg sim}\} \neq \emptyset$, *then there must exist a* $R_i^{\gg sim}$ *located at* $l_i^{\gg sim}$ *and a* $R_j^{\gg sim}$ *located at* $l_j^{\gg sim}$ *sharing the same antecedents.*

Based on Lemma 4, we can conclude that if there are two rules having the same antecedents, then they share the same computation for obtaining their ***simple dominating location***. Consider $R_{25} \equiv (df \Rightarrow a)$ and $R_{26} \equiv (df \Rightarrow b)$ in Table 3. $R_{25}^{\gg sim}$ at $l_{25}^{\gg sim}$ is $R_{10} \equiv (d \Rightarrow af)$. $R_{26}^{\gg sim}$ at $l_{26}^{\gg sim}$ is $R_{12} \equiv (d \Rightarrow bf)$. Observe that R_{10} and R_{12} share the same antecedents.

Lemma 5 *If* $R_i.items \equiv R_j.items$, $\{R_i^{\gg str}\} \neq \emptyset$ *and* $\{R_j^{\gg str}\} \neq \emptyset$, *then there must exist a* $R_i^{\gg str}$ *located at* $l_i^{\gg str}$ *and a* $R_j^{\gg str}$ *located at* $l_j^{\gg str}$ *sharing the same items.*

Based on Lemma 5, we can conclude that if there are two rules consisting of the same set of items, then they share the same computation for obtaining ***strict dominating location***. Consider $R_{13} \equiv (d \Rightarrow f)$ and $R_{20} \equiv (f \Rightarrow d)$ in Table 3. $R_{13}^{\gg str}$ at $l_{13}^{\gg str}$ is $R_{10} \equiv (d \Rightarrow af)$. $R_{20}^{\gg str}$ at $l_{20}^{\gg str}$ is $R_{16} \equiv (f \Rightarrow ad)$. Observe that R_{10} and R_{16} consist of a same set of items. In the *Adjacency Index* L, a node, representing an itemset X where the length of X is n, indexes rules which have the same antecedents as X and rules which consist of X. Instead of obtaining the ***simple dominating location*** for each rule in this first set at a cost of O(n), the computation can be performed only once by leveraging this index. Similarly, O(e) time computation is sufficient to obtain their ***strict dominating locations*** for the second set of rules. Considering the fact that the maximum number of rules is exponentially larger than the maximum number of itemsets. Our optimized redundancy resolution is exponentially faster than *PARAS* by a factor of approximately 1.5^n where n is the number of unique items in the database.

3.2.3. Redundancy Relationships Updating for Batch Update

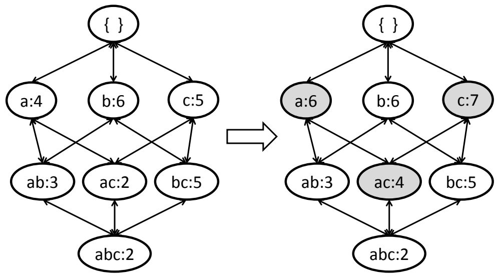

Figure 13: An example of Adjacency Index update

Algorithm 5: Redundancy Update

```
Input: $L'$, $\{Itemset\}^{changed}$, $\{R^+\}$
Output: $\{R^*\}$// Rules with redundancy info
begin
    switch $Itemset_i$ do
        case Updated
            forall the $Itemset$ in $\{Parents\ of\ Itemset_i\}$ do
                $RedundancyAbstraction(Index^{all}_{Itemset})$;
            forall the $Itemset$ in $\{Children\ of\ Itemset_i\}$ do
                $RedundancyAbstraction(Index^{antecedent}_{Itemset})$;
        case Inserted
            $RedundancyAbstraction(Index^{all}_{Itemset_i})$;
            $RedundancyAbstraction(Index^{antecedent}_{Itemset_i})$;
            Update Process in the "Updated" case;
```

Given a batch update to dataset D, new rules may be introduced and old ones expired or updated in terms of their *support* and *confidence* values, resulting in corresponding changes of the redundancy information in the parameter space. As we discussed in Section 3.2.2, the redundancy can be abstracted in the *Adjacency Index*. Therefore, changes of the itemsets can be used to identify the possible changes of redundancy information. The update strategies for redundancy information fall into two different cases:

I. The *support* of an itemset in the *Adjacency Index* is updated.

   In Figure 13, the support of "a", "c" and "ac" has all changed. All corresponding changes of the ruleset have been made by the rule updating mechanism as well. Consider the node "ac" on the left side in Figure 13, it is the least frequent parent of "abc". However, on the right side, "ab" becomes the least frequent parent of "abc" due to increasing frequency of "ac". Therefore, once an itemset is updated, all its child nodes have to be re-examined for possibly having to update the simple dominating locations for the rules in $Index^{ant}$ in each child node. On the left side, "ac" is not the most frequent child of "a". However, on the right side, "ab" is no longer the most frequent children of "a" due to the increasing frequency of "ac". Therefore, once an itemset is updated, all its parent nodes have to be re-examined for possibly having to update the strict dominating locations for the rules in $Index^{all}$ in each parent node.

II. A new itemset is inserted into the *Adjacency Index*.

   Our optimized redundancy abstraction is first executed on this node. Then the above redundancy updating algorithm is performed.

4. Experimental Results

Setup. Experiments are conducted on a OS X machine with 2.4 GHz Intel Core i5 processor and 8 GB of RAM. The system and its competitors are implemented in C++ using Qt Creator with Clang 64-bit compiler.
Datasets. We evaluated the performance of the *iPARAS* system using synthetic and real dataset benchmarks. The *retail* dataset (Brijs et al. (1999)) contains 88,163 transactions collected from a Belgian retail supermarket store in 5 months. To better measure the performance of the systems, we replicated the *retail* dataset 10 times denoted as *10retail*. We also generated 6 datasets by the *IBM Quest data generator* modeling transactions in a retail store. On average, each transaction has 10 items.

Table 5: Datasets

	10retail	*T20000k*	*T5000k*	*T1600k*	*T800k*	*T400k*	*T200k*
Transactions	881,630	19,663,637	4,916,218	1,573,347	786,700	393,210	196,601
Unique Items	16,470	23,870	7,062	7,599	7,596	7,589	7,576
Size	40.8 MB	1.2 GB	248.8 MB	79.6 MB	39.8 MB	19.9 MB	10 MB

4.1. Evaluation of Online Processing Time

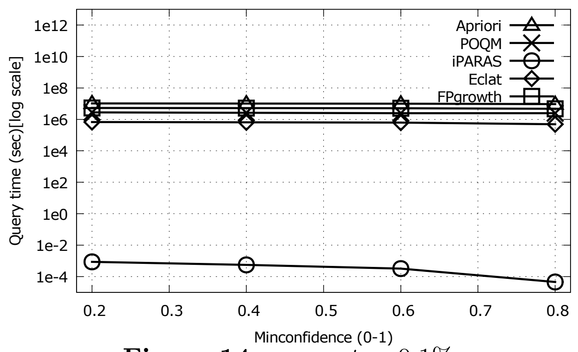

Figure 14: *support* = 0.1%

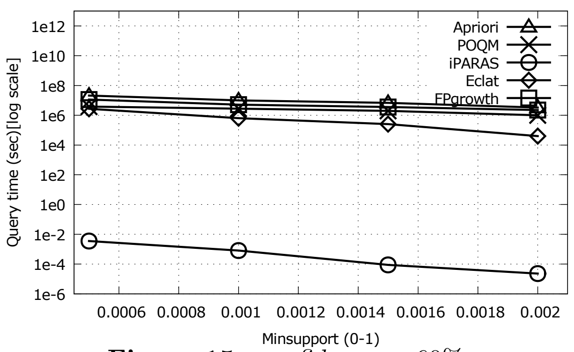

Figure 15: *confidence* = 60%

To evaluate the online processing time, we compared the *iPARAS* system against the original Apriori, Eclat, FP-growth implementations from (Borgelt) and *POQM*. The *primary support* setting for *POQM* and *iPARAS* is 0.03% and the *primary confidence* setting for *iPARAS* is 15%. Figure 14 and 15 illustrate the mining request processing time for T5000K. First, we fixed the *support* to a constant value and measured mining request processing times by varying *confidence* values. In Figure 14, for all of the five alternate algorithms, the mining request processing times decreased with increasing confidence value. As *confidence* increases, more rules get filtered, producing fewer rules in output. Overall, *iPARAS* outperformed four competitors by several orders of magnitude. Second, we fixed the *confidence* value and measured mining request processing times by varying *support* values. Figure 15 shows that the trend of the five alternatives is similar. *iPARAS* performed several orders of magnitude better than the four competitors. Overall, *iPARAS* consistently outperformed Apriori, *POQM*, Eclat and FP-growth by 4, 4, 3 and 4 orders, respectively.

4.2. Evaluation of Offline Construction Time

To evaluate the offline construction time, we compared the the *iPARAS* system against the state-of-the-art rule mining system *PARAS*. Both *iPARAS* and *PARAS* precompute the frequent itemsets and association rules to construct the parameter space. When new data arrives, *iPARAS* can utilize the previous knowledge for construction while *PARAS* has to start from scratch. First, we used each system to process a large initial dataset. Then we measured the precomputing time of each system given a relatively small batch update.

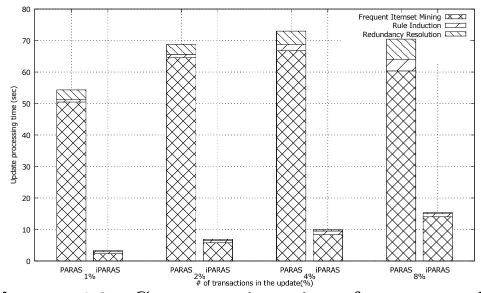

Figure 16: Construction time for one update

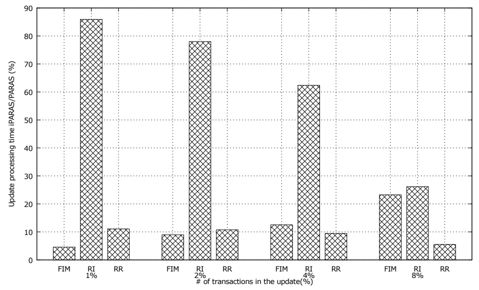

Figure 17: Ratio of construction time

For the *10retail*, we divided the dataset into two sets: one with 90 percent of the dataset, for initial construction, and one with 10 percent of the dataset, for randomly batch update generation. We measured the precomputing times by varying the size of the updates. For both systems, the *primary support* is 0.1% and the *primary confidence* is 10%. In Figure 16, *iPARAS* achieved about 16-fold, 10-fold, 7-fold and 5-fold speed-up of overall offline precomputing time compared to *PARAS* with respect to the update size of 1%, 2%, 4% and 8%. The differences of time consumed decrease with increasing size of the batch updates. Figure 17 shows the ratios of the running time of modules in *iPARAS* to the running time of corresponding modules in *PARAS* by varying the size of the updates. The trend of frequent itemset mining module is similar to the trend of overall performance in Figure 16. The performance of the rule induction (updating) module, however, shows an opposite trend on this dataset. The times used of rule induction module are similar for *iPARAS* and *PARAS* when the size of the updates is small. E.g., the ratio is 86% for 1% update. The reason in this case is that even the size of the update was small, it brought significant changes to previous ruleset. The system is likely to search and make changes of the entire rule collection in the *Adjacency Index* which made this process only a little faster than re-inducing from scratch. Yet the time saved becomes significant when the size of the updates increased. The optimized redundancy resolver in *iPARAS*, as expected, significantly reduced the redundancy information abstracting time as compared to re-abstracting from scratch using the *PARAS* method.

To evaluate *iPARAS* on larger dataset, we used *T20000K* for initial construction and *T200K*, *T400K*, *T800K*, *T1600K* for batch updates. For both *iPARAS* and *PARAS*, we set the *primary support* as 0.01% and the *primary confidence* as 10%. As shown in Figure 18, *iPARAS* achieved about 62-fold, 42-fold, 21-fold and 12-fold speed-up of overall offline precomputing time with respect to the update size of 1%, 2%, 4% and 8%. The time differences decrease with increase of the size of the batch updates. Figure 19 shows the ratios of the running time of modules in *iPARAS* to the running time of corresponding modules in *PARAS* by varying the size of the updates. All three modules show similar trends as the

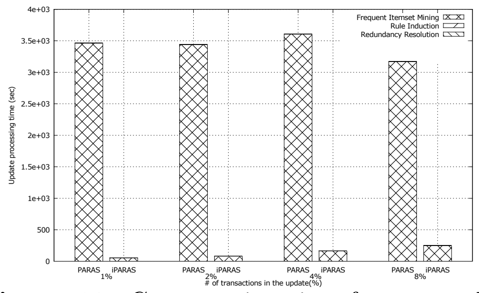

Figure 18: Construction time for one update

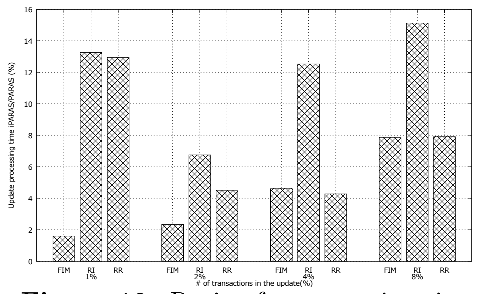

Figure 19: Ratio of construction time

overall performances of *iPARAS* in Figure 18. Each of the modules significantly reduced the precomputing time as compared to corresponding modules in *PARAS*.

5. Conclusion

We present our *iPARAS* framework for incremental parameter space construction to assure online association rule mining. *iPARAS* corresponds to an end-to-end solution, composed of three algorithms, for efficiently updating the precomputed ruleset for transaction databases. In the context of the parameter space, we can achieve fast redundancy abstraction at offline step by leveraging the proposed *Adjacency Index*. In a variety of tested cases, *iPARAS* outperforms the overall performance of the state-of-the-art online association rule mining systems, *POQM* (Aggarwal and Yu (2001)) and *PARAS* (Lin et al. (2013)). Our experimental evaluation using benchmark datasets confirms that *iPARAS* achieves several orders of magnitude improvement over the state-of-the-art approaches in online rule mining, as well as dozen folds speed-up in offline construction in a variety of tested cases.

References

Charu C. Aggarwal and Philip S. Yu. A new approach to online generation of association rules. *IEEE Trans. Knowl. Data Eng.*, 13(4):527–540, 2001.

Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.

Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD Conference*, pages 207–216, 1993.

Necip Fazil Ayan, Abdullah Uz Tansel, and M. Erol Arkun. An efficient algorithm to update large itemsets with early pruning. In *KDD*, pages 287–291, 1999.

Christian Borgelt. Efficient apriori, eclat & fp-growth. `http://www.borgelt.net`.

Tom Brijs, Gilbert Swinnen, Koen Vanhoof, and Geert Wets. Using association rules for product assortment decisions: A case study. In *KDD*, pages 254–260, 1999.

David Wai-Lok Cheung, Jiawei Han, Vincent T. Y. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *ICDE*, pages 106–114, 1996.

William Cheung and Osmar R. Zaïane. Incremental mining of frequent patterns without candidate generation or support constraint. In *IDEAS*, pages 111–116, 2003.

Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *SIGMOD Conference*, pages 1–12, 2000.

Daniel Hunyadi. Performance comparison of apriori and fp-growth algorithms in generating association rules. In *ECC*, pages 376–381, 2011.

Jia-Ling Koh and Shui-Feng Shieh. An efficient approach for maintaining association rules based on adjusting fp-tree structures. In *DASFAA*, volume 2973, pages 417–424. 2004.

Carson Kai-Sang Leung, Quamrul I. Khan, and Tariqul Hoque. Cantree: A tree structure for efficient incremental mining of frequent patterns. In *ICDM*, pages 274–281, 2005.

Xika Lin, Abhishek Mukherji, Elke A. Rundensteiner, Carolina Ruiz, and Matthew O. Ward. Paras: A parameter space framework for online association mining. *PVLDB*, 6 (3):193–204, 2013.

David Newman and Arthur Asuncion. UCI machine learning repository, 2007.

Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. New algorithms for fast discovery of association rules. In *KDD*, pages 283–286, 1997.