Mining High Utility Patterns in One Phase without Generating Candidates

Junqiang Liu, *Member, IEEE*, Ke Wang, *Senior Member, IEEE*, and Benjamin C.M. Fung, *Senior Member, IEEE*

Abstract—Utility mining is a new development of data mining technology. Among utility mining problems, utility mining with the itemset share framework is a hard one as no anti-monotonicity property holds with the interestingness measure. Prior works on this problem all employ a two-phase, candidate generation approach with one exception that is however inefficient and not scalable with large databases. The two-phase approach suffers from scalability issue due to the huge number of candidates. This paper proposes a novel algorithm that finds high utility patterns in a single phase without generating candidates. The novelties lie in a high utility pattern growth approach, a lookahead strategy, and a linear data structure. Concretely, our pattern growth approach is to search a reverse set enumeration tree and to prune search space by utility upper bounding. We also look ahead to identify high utility patterns without enumeration by a closure property and a singleton property. Our linear data structure enables us to compute a tight bound for powerful pruning and to directly identify high utility patterns in an efficient and scalable way, which targets the root cause with prior algorithms. Extensive experiments on sparse and dense, synthetic and real world data suggest that our algorithm is up to 1 to 3 orders of magnitude more efficient and is more scalable than the state-of-the-art algorithms.

Index Terms—Data mining, utility mining, high utility patterns, frequent patterns, pattern mining

1 Introduction

Finding interesting patterns has been an important data mining task, and has a variety of applications, for example, genome analysis, condition monitoring, cross marketing, and inventory prediction, where interestingness measures [17], [36], [41] play an important role. With frequent pattern mining [2], [3], [18], [43], a pattern is regarded as interesting if its occurrence frequency exceeds a user-specified threshold. For example, mining frequent patterns from a shopping transaction database refers to the discovery of sets of products that are frequently purchased together by customers. However, a user's interest may relate to many factors that are not necessarily expressed in terms of the occurrence frequency. For example, a supermarket manager may be interested in discovering combinations of products with high profits or revenues, which relates to the unit profits and purchased quantities of products that are not considered in frequent pattern mining.

Utility mining [41] emerged recently to address the limitation of frequent pattern mining by considering the user's expectation or goal as well as the raw data. Utility mining with the itemset share framework [19], [39], [40], for example, discovering combinations of products with high profits or revenues, is much harder than other categories of utility mining problems, for example, weighted itemset mining [10], [25], [30] and objective-oriented utility-based association mining [11], [35]. Concretely, the interestingness measures in the latter categories observe an anti-monotonicity property, that is, a superset of an uninteresting pattern is also uninteresting. Such a property can be employed in pruning search space, which is also the foundation of all frequent pattern mining algorithms [3]. Unfortunately, the anti-monotonicity property does not apply to utility mining with the itemset share framework [39], [40]. Therefore, utility mining with the itemset share framework is more challenging than the other categories of utility mining as well as frequent pattern mining.

Most of the prior utility mining algorithms with the itemset share framework [4], [15], [24], [29], [38], [39] adopt a two-phase, candidate generation approach, that is, first find candidates of high utility patterns in the first phase, and then scan the raw data one more time to identify high utility patterns from the candidates in the second phase.

The challenge is that the number of candidates can be huge, which is the scalability and efficiency bottleneck. Although a lot of effort has been made [4], [15], [24], [38] to reduce the number of candidates generated in the first phase, the challenge still persists when the raw data contains many long transactions or the minimum utility threshold is small. Such a huge number of candidates causes scalability issue not only in the first phase but also in the second phase, and consequently degrades the efficiency. One exception is the HUIMiner algorithm [28], which is however even less efficient than two-phase algorithms when mining large databases due to inefficient join operations, lack of strong pruning, and scalability issue with its vertical data structure.

- *J. Liu is with the School of Information and Electronic Engineering, Zhejiang Gongshang University, Hangzhou 310018, China. E-mail: jjliu@alumni.sfu.ca.*
- *K. Wang is with the School of Computing Science, Simon Fraser University, Burnaby, BC V5A 1S6, Canada. E-mail: wangk@cs.sfu.ca.*
- *B.C.M. Fung is with the School of Information Studies, McGill University, Montreal, QC H3A 1X1, Canada. E-mail: ben.fung@mcgill.ca.*

Manuscript received 23 Mar. 2014; revised 12 Nov. 2015; accepted 10 Dec. 2015. Date of publication 17 Dec. 2015; date of current version 30 Mar. 2016. Recommended for acceptance by B. Goethals.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2015.2510012

To address the challenge, this paper proposes a new algorithm, d^{2}HUP, for utility mining with the itemset share framework, which employs several techniques proposed for mining frequent patterns, including exploring a regular set enumeration in a reverse lexicographic order [43] and heuristics for ordering items [18], [43]. Our contributions are as follows:

- A high utility pattern growth approach is proposed, which we argue is one without candidate generation because while the two-phase, candidate generation approach employed by prior algorithms first generates high TWU patterns (candidates) with TWU being an interim, anti-monotone measure and then identifies high utility patterns from high TWU patterns, our approach directly discovers high utility patterns in a single phase without generating high TWU patterns (candidates). The strength of our approach comes from powerful pruning techniques based on tight upper bounds on utilities.
- A lookahead strategy is incorporated with our approach, which tries to identify high utility patterns earlier without recursive enumeration. Such a strategy is based on a closure property and a singleton property, and enhances the efficiency in dealing with dense data.
- A linear data structure, CAUL, is proposed to represent original utility information in raw data, which targets the root cause with prior algorithms, that is, they all employ a data structure to maintain the utility estimates instead of the original utility information, and thus can only determine the candidacy of a pattern but not the actual utility of the pattern in their first phase.

The rest of the paper is organized as follows. Section 2 defines the utility mining problem. Section 3 surveys related works. Section 4 proposes our pattern growth approach. Section 5 presents our algorithm. Section 6 discusses the data structure and implementation. Section 7 experimentally evaluates our algorithm. Section 8 analyzes individual techniques. Section 9 concludes the paper.

2 Utility Mining Problem

This section defines the utility mining problem with the itemset share framework that we study.

Let I be the universe of items. Let D be a database of transactions $\{t_1, \ldots, t_n\}$, where each transaction $t_i \subseteq I$. Each item in a transaction is assigned a non-zero share. Each distinct item has a weight independent of any transaction, given by an eXternal Utility Table (XUT). The research problem of finding all high utility patterns is formally defined as follows.

Definition 1. *The* internal utility *of an item i in a transaction t, denoted by $iu(i,t)$, is the share of i in t. The* external utility *of an item i, denoted by $eu(i)$, is the weight of i independent of any transaction. The* utility *of an item i in a transaction t, denoted by $u(i,t)$, is the function f of $iu(i,t)$ and $eu(i)$, that is, $u(i,t) = f(iu(i,t), eu(i))$. We assume that the range of f is non-negative, that is, $u(i,t) \geq 0$.*

Although the utility function f may not be non-negative in an application, it is generally agreed that we can transform the utility function f into a non-negative function as discussed by Yao et al. [41].

TABLE 1
Database D and eXternal Utility Table XUT

(a) D: shopping transactions

TID	ITEM						
	a	b	c	d	e	f	g
t_1	1		1		1		
t_2	6	2	2			5	
t_3	1	1	1	2	6		5
t_4	3	1		4	3		
t_5	2	1		2		2	

(b) XUT: prices

ITEM	PRICE
a	1
b	3
c	5
d	2
e	2
f	1
g	1

Running example. Consider the data of a supermarket. Table 1a lists the quantity (share) of each product (item) in each shopping transaction where $I = \{a, b, c, d, e, f, g\}$ and $D = \{t_1, t_2, t_3, t_4, t_5\}$, and Table 1b lists the price (weight) of each product. For transaction t_2 = {a, b, c, f}, we have $iu(a, t_2) = 6$, $iu(b, t_2) = 2$, $iu(c, t_2) = 2$, $iu(f, t_2) = 5$, $eu(a) = 1$, $eu(b) = 3$, $eu(c) = 5$, and $eu(f) = 1$. Here, $u(i,t)$ is the product of $iu(i,t)$ and $eu(i)$. Thus, $u(a, t_2) = 6$, $u(b, t_2) = 6$, $u(c, t_2) = 10$, $u(f, t_2) = 5$, and so on.

Definition 2. *(a) A transaction t* contains *a pattern X if X is a subset of t, that is, $X \subseteq t$, which means that every item i in X has a non-zero share in t, that is, $iu(i,t) \neq 0$. (b) The* transaction set *of a pattern X, denoted by $TS(X)$, is the set of transactions that contain X. The number of transactions in $TS(X)$ is the* support *of X, denoted by $s(X)$.*

Definition 3. *(a) For a pattern X contained in a transaction t, that is, $X \subseteq t$, the utility of X in t, denoted by $u(X,t)$, is the sum of the utility of every constituent item of X in t, that is,*

$$u(X,t) = \sum_{i \in X \subseteq t} u(i,t).$$

(b) The utility of X, denoted by $u(X)$, is the sum of the utility of X in every transaction containing X, that is,

$$u(X) = \sum_{t \in TS(X)} u(X,t) = \sum_{t \in TS(X)} \sum_{i \in X} u(i,t).$$

Definition 4. *A pattern X is a* high utility pattern, *abbreviated as **HUP**, if the utility of X is no less than a user-defined minimum utility threshold, denoted by $minU$. **High utility pattern mining** is to discover all high utility patterns, that is,*

$$HUPset = \{X | X \subseteq I, u(X) \geq minU\}.$$

In the running example, the manager wants to know every combination of products with sales revenue no less than 30, that is, $minU = 30$. Since $TS(\{a, b\}) = \{t_2, t_3, t_4, t_5\}$, we have $u(\{a, b\}) = u(\{a, b\}, t_2) + u(\{a, b\}, t_3) + u(\{a, b\}, t_4) + u(\{a, b\}, t_5) = u(a, t_2) + u(b, t_2) + u(a, t_3) + u(b, t_3) + u(a, t_4) + u(b, t_4) + u(a, t_5) + u(b, t_5) = 27$. Similarly, $u(\{a, c\}) = 28$, $u(\{b, c\}) = 24$, $u(\{a, b, c\}) = 31$, $u(\{a, b, c, d\}) = 13$, and so on. Therefore, $HUPset$ = { {a, b, c}, {a, b, d}, {a, d, e}, {a, b, d, e}, {b, d, e}, {d, e}, {a, b, c, d, e, g} }.

An observation is that the utilities of patterns are neither anti-monotone nor monotone.

3 Related Works

High utility pattern mining problem is closely related to frequent pattern mining, including constraint-based mining. In this section, we briefly review prior works both on frequent pattern mining and on utility mining, and discuss how our work connects to and differs from the prior works.

3.1 Frequent Pattern Mining

Frequent pattern mining was first proposed by Agrawal et al. [2], which is to discover all patterns whose supports are no less than a user-defined minimum support threshold. Frequent pattern mining employs the anti-monotonicity property: the support of a superset of a pattern is no more than the support of the pattern. Algorithms for mining frequent patterns as well as algorithms for mining high utility patterns fall into three categories, breadth-first search, depth-first search, and hybrid search.

Apriori by Agrawal and Srikant [3] is a very famous breadth-first algorithm for mining frequent patterns, which scans the disk-resident database as many times as the maximum length of frequent patterns. FP-growth by Han et al. [18] is a well-known depth-first algorithm, which compresses the database by FP-trees in main memory. Eclat by Zaki [43] is a famous hybrid algorithm. It keeps a database or a database partition [34] in memory by a vertical tid-list layout [21] and can work in either depth-first or breadth-first manner.

This paper adopts a depth-first strategy since breadth-first search is typically more memory-intensive and more likely to exhaust main memory and thus slower. Concretely, our algorithm depth-first searches a reverse set enumeration tree, which can be thought of as exploring a regular set enumeration tree [1], [18], [33] right-to-left in a reverse lexicographic order [43]. While Eclat [43] also explores such an order, our algorithm is the first fully exploiting the benefit in mining high utility patterns.

3.2 Constraint-Based Mining

Constraint-based mining is a milestone in evolving from frequent pattern mining to utility mining. Works on this area mainly focus on how to push constraints into frequent pattern mining algorithms.

Pei et al. [32] discussed constraints that are similar to (normalized) weighted supports [10], and first observed an interesting property, called convertible anti-monotonicity, by arranging the items in weight-descending order. The authors demonstrated how to push them into the FP-growth algorithm [18] .

Bucila et al. [9] considered mining patterns that satisfy a conjunction of anti-monotone and monotone constraints, and proposed an algorithm, DualMiner, that efficiently prunes its search space using both anti-monotone and monotone constraints.

Bonchi et al. [6] introduced the ExAnte property which states that any transaction that does not satisfy the given monotone constraint can be removed from the input database, and integrated the property with Apriori-style algorithms. Bonchi and Goethals [7] applied the ExAnte property with the FP-growth algorithm. Bonchi and Lucchese [8] generalized the data reduction technique to a unified framework.

De Raedt et al. [14] investigated how standard constraint programming techniques can be applied to constraint-based mining problems with constraints that are monotone, anti-monotone, and convertible.

Bayardo and Agrawal [5], and Morishita and Sese [31] proposed techniques of pruning based on upper bounds when the constraint is neither monotone, anti-monotone, nor convertible. This paper also employs such a standard technique. Our contribution is to develop tight upper bounds on the utility.

3.3 Some Categories of Utility Mining

Interestingness measures can be classified as objective measures, subjective measures, and semantic measures [17]. Objective measures [20], [37], such as support or confidence, are based only on data; Subjective measures [13], [36], such as unexpectedness or novelty, take into account the user's domain knowledge; Semantic measures [41], also known as utilities, consider the data as well as the user's expectation. Below, we discuss three categories in detail.

Hilderman et al. [19] proposed the itemset share framework that takes into account the weights both on attributes, for example, the price of a product, and on attribute-value pairs, for example, the quantity of a product in a shopping basket. Then, support and confidence measures can be generalized based on count-shares as well as on amount-shares. Yao et al. [39], [40] proposed a utility measure equivalent to Definition 3 that instantiates this framework. This paper falls into that category.

Cai et al. [10] proposed weighted itemset mining. Lin et al. [25] proposed value added association mining. Both works assigns each item a weight representing its importance, which results in (normalized) weighted supports, also known as horizontal weights. Lu et al. [30] proposed to assign a weight to each transaction representing the significance of the transaction, also known as vertical weights.

Shen et al. [35] and Chan et al. [11] proposed objective-oriented utility-based association mining that explicitly models associations of a specific form "$Pattern \rightarrow Objective$" where $Pattern$ is a set of nonobjective-attribute value pairs, and $Objective$ is a logic expression asserting objective-attributes with each objective-attribute value satisfying (violating) $Objective$ assigned a positive (negative) utility.

3.4 Algorithms with the Itemset Share Framework

As the utility measure with the itemset share framework is neither anti-monotone, monotone, nor convertible, most prior algorithms resort to an interim measure, (TWU), proposed by Liu et al. [29], and adopt a two-phase, candidate generation approach.

Transaction weighted utilization of a pattern is the sum of the transaction utilities of all the transactions containing the pattern. For the running example, $TWU(\{a, b\}) = 88$, the sum of the utilities of transactions t_2, t_3, t_4, and t_5, $TWU(\{a, b, c\}) = 57$, that of t_2 and t_3, and $TWU(\{a, b, c, d\}) = 30$, that of t_3. Clearly, TWU is anti-monotone.

TWU or its variants is employed by most prior algorithms, which first invoke either Apriori [3] or FP-growth [18] to find high TWU patterns (candidates), and then scan the raw data once more to identify high utility patterns from the candidates. An exception is that Yao et al. [40], [41]

presented an upper bound property, that is, the utility of a size-k pattern is no more than the average utility of its size-(k-1) subsets, which is however looser than the TWU property.

Liu et al. [29] proposed the anti-monotonicity property with TWU, based on which they developed the TwoPhase algorithm by adapting Apriori [3].

Li et al. [24] proposed an isolated items discarding strategy (IIDS). An isolated item is one that is not contained in any length-k candidate, and hence it will not occur in any candidate with a length greater than k. Any multi-pass, level-wise algorithm can employ IIDS to reduce the number of redundant candidates.

Lan et al. [23] proposed an projection-based algorithm, based on the TWU model [29], that speeds up the execution by an indexing mechanism.

Erwin et al. [15] proposed the CTU-PROL algorithm for mining high utility patterns that integrates the TWU anti-monotonicity property and pattern growth approach [18] in the first phase, which is facilitated by a compact utility pattern tree structure, CUP-tree.

Ahmed et al. [4] proposed a tree-based algorithm, IHUP$_{TWU}$, for mining high utility patterns, which uses an IHUP$_{TWU}$-tree to maintain the TWU information of transactions, and mines the set of candidates of high utility patterns by adapting FP-growth [18]. Notice that CTU-PROL [15] and IHUP$_{TWU}$ produce the same amount of candidates in the first phase.

Tseng et al. [38] proposed the latest, FP-growth based algorithm, UP-Growth, which uses an UP-tree to maintain the revised TWU information, improves the TWU property based pruning, and thus generates fewer candidates in the first phase.

Yun et al. [42] and Dawar and Goyal [12] improved UP-Growth [38] by pruning more candidates, while the inherent issue of the two-phase approach remains.

Our preliminary work [27] and Liu and Qu [28], simultaneously and independently, proposed to mine high utility patterns without candidate generation. The HUIMiner algorithm by Liu and Qu [28] employs a vertical data structure to represent utility information, which employs inefficient join operations and is also not scalable. HUI-Miner is even less efficient than an improved version of UP-Growth [38] when mining large databases. Therefore, scalability and efficiency remains to be a challenge with HUIMiner [28]. Our work addresses such a challenge with large databases.

Fournier-Viger et al. [16] improved HUIMiner [28] by pre-computing the TWUs of pairs of items to reduce the number of join operations. Krishnamoorthy [22] improved HUIMiner [28] by a partition strategy. Their improvement is within a factor of 2 to 6, while our algorithm is up to 45 times faster than HUIMiner [28] on the same databases.

This paper has enhanced our preliminary work [27] with efficient computation by pseudo projection, and with optimizations by partial materialization and controlled irrelevant item filtering. We have put more thoughts into our algorithm and improved the implementation. Moreover, comparative experiments with state-of-art algorithms and experimental anatomy of our individual techniques have been performed.

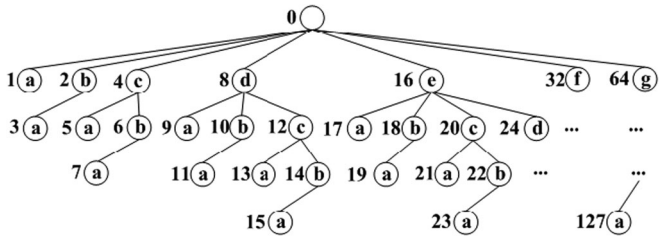

Fig. 1. Reverse set enumeration tree where each node is numbered in the order of depth-first search.

4 High Utility Pattern Growth

The general approach to mining high utility pattern is to enumerate each subset X of I, and test if X has a utility over the threshold. However, an exhaustive enumeration is infeasible due to the huge number of subsets of I, and hence it is critical to employ strong pruning techniques.

This section proposes a new approach to the problem, that is, a high utility pattern growth approach. We first introduce a reverse set enumeration tree as a way to enumerate patterns, and then propose strong pruning techniques that drastically reduces the number of patterns to be enumerated, which lays the theoretical foundation for our algorithm.

4.1 Growing Reverse Set Enumeration Tree

Our pattern growth approach can be thought of as growing or searching a reverse set enumeration tree in a depth-first manner as shown in Fig. 1.

The construction of the reverse set enumeration tree follows an imposed ordering Ω of items. Concretely, the root is labelled by no item, each node N other than the root is labelled by an item, denoted by $item(N)$, the path from N to the root represents a pattern, denoted by $pat(N)$, and the child nodes of N are labelled by items listed before $item(N)$ in Ω. It follows that the sequence of items along the path from N to the root is in accordance with Ω.

Definition 5. *The* imposed ordering of items*, denoted by* Ω*, is a pre-determined, ordered sequence of all the items in* I*. Accordingly, for items* i *and* j*,* $i \prec j$ *denotes that* i *is listed before* j*;* $i \prec X$ *denotes that* $i \prec j$ *for every* $j \in X$*, and* $W \prec X$ *denotes that* $i \prec X$ *for every* $i \in W$*, in accordance with* Ω*.*

The imposed ordering Ω of items can be determined by a heuristic proposed by [43]. Given Ω, a pattern can also be represented as an ordered sequence. For brevity, we use the set notation, for example, {a, b, c}, in place of the sequence notation, for example, $<$a, b, c$>$. For example in Fig. 1, the imposed ordering is the lexicographic order, i.e., Ω = {a, b, c, d, e, f, g}, then a $\prec$ b, a $\prec$ c, a $\prec$ {b, c}, {a, b} $\prec$ {c, d}, and so on.

The reverse set enumeration tree is equivalent to a regular set enumeration tree [1], [18], [33] that is imposed with a reverse lexicographic order [43] and explored right-to-left, which yields a property: a pattern is always enumerated before its supersets [43] in a depth-first search. For example, {a} and {b} are before {a, b}, and {a, b} and {c} before {a, b, c}.

Most importantly, by such a construction, the transaction set supporting the enumerated pattern can be determined

by a pseudo projection, for example, $TS(\{a, b\})$ can be projected from $TS(\{b\})$ without materialization, and thus we can compute the utility of the pattern and a utility upper bound used for pruning in an efficient and scalable way.

4.2 Pruning by Utility Upper Bounding

It is computationally infeasible to enumerate all patterns, and a standard technique is to prune the search space. However, for utility mining with the itemset share framework, no anti-monotonicity property can be employed for pruning. An alternative is pruning based on utility upper bounding [5], [31].

With our pattern growth approach, it is to estimate an upper bound on utilities of all possible patterns represented by nodes in the subtree rooted at the node currently being explored, when growing the reverse set enumeration tree. If such an upper bound is less than $minU$, the subtree can be pruned as all patterns in the subtree are not high utility patterns.

Notice that a pattern Y represented by a node C in the subtree rooted at a node N is a prefix extension of the pattern X represented by N, which leads to a way to estimate an upper bound on the utility of Y.

Definition 6. *Given an ordering Ω, a pattern Y is a* prefix extension *of a pattern X, if X is a suffix of Y, that is, if $Y = W \cup X$ for some W with $W \prec X$ in Ω.*

Definition 7. *Given an ordering Ω, a pattern Y is the* full prefix extension *of a pattern X w.r.t. a transaction t containing X, denoted as $Y = fpe(X, t)$, if Y is a prefix extension of X derived by adding exactly all the items in t that are listed before X in Ω, that is, if $Y = W \cup X$ with $W = \{i | i \in t \wedge i \prec X \wedge X \subseteq t\}$.*

For the running example, the full prefix extensions of {c} w.r.t. t_1 and t_2 are $fpe(\{c\}, t_1) = \{a, c\}$ and $fpe(\{c\}, t_2) = \{a, b, c\}$ respectively.

Theorem 1 (Basic upper bounds). *For a pattern X, the sum of the utility of the full prefix extension of X w.r.t. each transaction in $TS(X)$, denoted by $uB_{fpe}(X)$, is no less than the utility of any prefix extension Y of X, that is,*

$$uB_{fpe}(X) = \sum_{t \in TS(X)} u(fpe(X, t), t) \geq u(Y). \quad (1)$$

Proof. The premise, Y is a prefix extension of X, means $X \subseteq Y$, and thus has two implications. First, $TS(Y) \subseteq TS(X)$. Second, $\forall t \in TS(Y)$, $Y \subseteq fpe(X, t)$. As the utility function is non-negative, we have

$$uB_{fpe}(X) = \sum_{t \in TS(X)} u(fpe(X, t), t)$$

$$\geq \sum_{t \in TS(Y)} u(fpe(X, t), t) \geq \sum_{t \in TS(Y)} u(Y, t) = u(Y). \quad \square$$

For example, when enumerating {} by Node 0 in Fig. 1, we get $TS(\{a\}) = D$ and $uB_{fpe}(\{a\}) = u(\{a\}) = 13 < minU = 30$, $TS(\{b\}) = \{t_2, t_3, t_4, t_5\}$ and $uB_{fpe}(\{b\}) = u(\{a, b\}) = 27 < minU$, $TS(\{c\}) = \{t_1, t_2, t_3\}$ and $uB_{fpe}(\{c\}) = u(\{a, c\}, t_1) + u(\{a, b, c\}, t_2) + u(\{a, b, c\}, t_3) = 37 > minU$, and so on. Thus, Nodes 1 and 2 (with Node 3) are pruned, and Nodes 4, 8, 16, 32, and 64 will be visited.

Clearly, the tighter the upper bound, the stronger the pruning. An observation is that many items never occur in high utility patterns when raw data are sparse [24]. It is possible to exclude them to tighten the upper bound.

Corollary 2 (Relevance of an item). *For a pattern X and an item $i \prec X$, the sum of the utility of the full prefix extension of X w.r.t. every transaction in $TS(\{i\} \cup X)$, denoted by $uB_{item}(i, X)$, is no less than the utility of a prefix extension Y of X that contains i, that is,*

$$uB_{item}(i, X) = \sum_{t \in TS(\{i\} \cup X)} u(fpe(X, t), t) \geq u(Y). \quad (2)$$

Proof. The extra premise in addition to Theorem 1 is that $i \subseteq Y$, which results in that $\{i\} \cup X \subseteq Y$. In the light of Theorem 1, we get this corollary. $\square$

Corollary 2 states that an item $i \prec X$ is *irrelevant* to any high utility pattern that is a prefix extension of X if $uB_{item}(i, X) < minU$, and can be ignored in enumerating prefix extensions of X.

For example, when enumerating {d, e} by Node 24 in Fig. 1, we have $uB_{item}(a, \{d, e\}) = uB_{item}(b, \{d, e\}) = u(\{a, b, c, d, e\}, t_3) + u(\{a, b, d, e\}, t_4) = 45 > minU$, and $uB_{item}(c, \{d, e\}) = u(\{a, b, c, d, e\}, t_3) = 25 < minU$. Thus, items a and b are relevant, and item c is irrelevant in enumerating prefix extensions of {d, e}. Furthermore, we can apply Corollary 2 iteratively as excluding an irrelevant item may decrease uB_{item} and uB_{fpe} of other items.

Corollary 3 (Tighter upper bounds). *For a pattern X and its prefix extension Y that is relevant in growing high utility patterns, a tighter upper bound on the utility of Y is*

$$uB'_{fpe}(X) = \sum_{t \in TS(X)} u(fpe'(X, t), t) \geq u(Y), \quad (3)$$

where $fpe'(X, t)$ is derived from $fpe(X, t)$ by excluding all irrelevant items $i \prec X$ by Corollary 2.

For example, as item c is irrelevant in enumerating prefix extensions of {d, e} represented by Node 24 in Fig. 1, we compute uB_{item} the second time by excluding item c, which yields $uB_{item}(a, \{d, e\}) = uB_{item}(b, \{d, e\}) = u(\{a, b, d, e\}, t_3) + u(\{a, b, d, e\}, t_4) = 40$. The bounds get tighter though the set of relevant items does not shrink.

4.3 Avoiding Enumeration by Lookahead

It is always beneficial to look ahead in a search process if it incurs little extra computation. Inspired by closed frequent pattern mining [44], we observe that when all the prefix extensions of the pattern currently enumerated have the same support, in particular for two cases, it is inexpensive to look ahead.

Case 1: Every prefix extension of a pattern has the same support and has a utility no less than $minU$, which can be tested by Theorem 4 with little overhead.

Theorem 4 (Closure). *For a pattern X and a set W of items with $X \cap W = \emptyset$, if a property denoted as $Closure(X, W, minU)$ is satisfied, i.e., if $s(\{i\} \cup X) = s(X)$ and $u(\{i\} \cup X) \geq minU$*

for all $i \in W$, then

$$u(S \cup X) \geq minU,\ \forall S \subseteq W \wedge S \neq \emptyset. \quad (4)$$

Proof. $\forall i \in W$, it is always true that $TS(\{i\} \cup X) \subseteq TS(X)$ as $X \subset \{i\} \cup X$; by the premise, $s(\{i\} \cup X) = s(X)$, we have $TS(\{i\} \cup X) = TS(X)$, which results in $TS(S \cup X) = TS(X)$ for $S \subseteq W$. Therefore, $\forall S \subseteq W \wedge S \neq \emptyset$,

$$u(S \cup X) = \sum_{t \in TS(S \cup X)} u(S \cup X, t) = \sum_{t \in TS(\{i\} \cup X), i \in S} u(S \cup X, t)$$

$$\geq \sum_{t \in TS(\{i\} \cup X), i \in W} u(\{i\} \cup X, t) = u(\{i\} \cup X) \geq minU. \quad \square$$

For example, when enumerating {d, e} by Node 24 in Fig. 1, we get $s(\{\text{a}\} \cup \{\text{d, e}\}) = s(\{\text{b}\} \cup \{\text{d, e}\}) = s(\{\text{d, e}\}) = 2$, and $u(\{\text{a}\} \cup \{\text{d, e}\}) = 34 > minU$ and $u(\{\text{b}\} \cup \{\text{d, e}\}) = 36 > minU$ while items a and b are relevant items and item c is not. Therefore, we know that all the prefix extensions of {d, e} with relevant items a and b, enumerated by Nodes 24-27, are high utility patterns without searching the rest of the subtree rooted at Node 24.

Case 2: All the prefix extensions of a pattern have the same support, but among which only the longest has a utility no less than $minU$. Such a case can be identified by Theorem 5 with little computation.

Theorem 5 (Singleton). *For a pattern X and a set W of items with $X \cap W = \emptyset$, if a property denoted as $Singleton(X, W, minU)$ holds, that is, if $s(\{i\} \cup X) = s(X)$ for all $i \in W$ and*

$$minU \leq u(W \cup X) < minU + \min_{j \in W} \sum_{t \in TS(X)} u(\{j\}, t)$$

then

$$u(S \cup X) < minU, \forall S \subset W. \quad (5)$$

Proof. In the light of Theorem 4, $TS(S \cup X) = TS(X)$ for $S \subseteq W$. It follows that $\forall S \subset W$,

$$u(S \cup X) = \sum_{t \in TS(S \cup X)} u(S \cup X, t)$$

$$= \sum_{t \in TS(W \cup X)} u(W \cup X, t) - \sum_{t \in TS(W \setminus S \cup X)} u(W \setminus S, t)$$

$$\leq u(W \cup X) - \sum_{t \in TS(\{j\} \cup X) \wedge j \in W \setminus S} u(\{j\}, t)$$

$$\leq u(W \cup X) - \min_{j \in W} \sum_{t \in TS(X)} u(\{j\}, t) < minU. \quad \square$$

For example, when enumerating {g} by Node 64 in Fig. 1, items a, b, c, d, and e are relevant, and $s(\{\text{a}\} \cup \{g\}) = s(\{\text{b}\} \cup \{\text{g}\}) = s(\{\text{c}\} \cup \{\text{g}\}) = s(\{\text{d}\} \cup \{\text{g}\}) = s(\{\text{e}\} \cup \{\text{g}\}) = s(\{\text{g}\}) = 1$, and $u(\{\text{a, b, c, d, e}\} \cup \{\text{g}\}) = 30 = minU$. Thus, we know that {a, b, c, d, e, g} is a high utility pattern and all its proper subsets are not without traversing the rest of the subtree.

5 Mining Patterns in One Phase without Candidate Generation

This section presents our algorithm, d^{2}HUP, namely Direct Discovery of High Utility Patterns, which is an integration of the depth-first search of the reverse set enumeration tree, the pruning techniques that drastically reduces the number of patterns to be enumerated, and a novel data structure that enables efficient computation of utilities and upper bounds which will be detailed in Section 6.1.

Moreover, our algorithm lists items in the descending order of uB_{item} based on a heuristic proposed by [43]. The pseudo code of d^{2}HUP is shown in Algorithm 1, which works as follows.

Algorithm 1. d^{2}HUP$(D, XUT, minU)$

```
1  build TS({}) and Ω from D and XUT
2  N ← root of reverse set enumeration tree
3  DFS(N, TS(pat(N)), minU, Ω)

Subroutine: DFS(N, TS(pat(N)), minU, Ω)
4  if u(pat(N)) ≥ minU then output pat(N)
5  W ← {i | i ≺ pat(N) ∧ uB_item(i, pat(N)) ≥ minU}
6  if Closure(pat(N), W, minU) is satisfied
7  then output nonempty subsets of W ∪ pat(N)
8  else if Singleton(pat(N), W, minU) is satisfied
9  then output W ∪ pat(N) as an HUP
10 else foreach item i ∈ W in Ω do
11   if uB_fpe({i} ∪ pat(N)) ≥ minU
12   then C ← the child node of N for i
13     TS(pat(C)) ← Project(TS(pat(N)), i)
14     DFS(C, TS(pat(C)), minU, Ω)
15   end foreach
```

d^{2}HUP builds $TS(\{\})$ by scanning the database D and the external utility table XUT to compute $s(\{i\})$, $u(\{i\})$, $uB_{item}(i, \{\})$, and $uB_{fpe}(\{i\})$ for each item i by Definitions 2 and 3, Corollary 2, and Theorem 1 or Corollary 3, and makes Ω in the descending order of uB_{item} (at line 1). d^{2}HUP starts searching high utility patterns from the root of reverse set enumeration tree (at lines 2-3) by calling the DFS$(N,\ TS(pat(N)),\ minU,\ \Omega)$ subroutine.

For the node N currently being visited, DFS prints $pat(N))$ as a high utility pattern if its utility is no less than the threshold (at line 4), makes the set W of relevant items (at line 5), and then gets through one of the three branches as follows.

If the closure property holds, DFS outputs every prefix extension of $pat(N)$ with relevant items as a high utility pattern by Theorem 4 (at lines 6-7);

If the singleton property holds, DFS prints the union of all the relevant items and $pat(N)$ as a high utility pattern by Theorem 5 (at lines 8-9);

For each relevant item $i \in W$, if the upper bound on the utilities of prefix extensions of $\{i\} \cup pat(N)$ is no less the threshold, DFS prepares $TS(pat(C))$ for the child node C with $item(C) \leftarrow i$ and $pat(C) \leftarrow \{i\} \cup pat(N)$, and recursively searches the subtree rooted at C (at lines 10–15). Note that DFS computes $TS(pat(C))$ by a pseudo projection from $TS(pat(N))$, which is implemented as Algorithm 2 in Section 6.3.

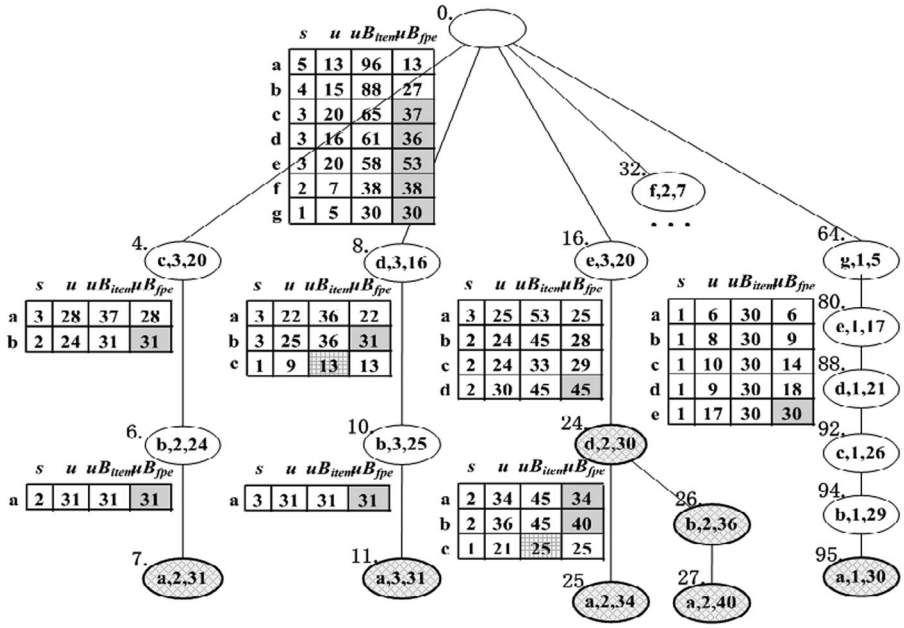

Fig. 2. Pruned version of the reverse set enumeration tree in Fig. 1, showing how d^{2}HUP works given $minU = 30$.

5.1 Revisit the Running Example

The execution process of d^{2}HUP can be thought of as searching a pruned version of a reverse set enumeration tree, which is shown in Fig. 2 for our running example where each node N is labelled with $item(N)$, $s(pat(N))$, and $u(pat(N))$.

d^{2}HUP starts by computing $TS(\{\})$ and by enumerating Node 0 (at lines 1-3). For each node N to be enumerated, d^{2}HUP represents $TS(pat(N))$ by the data structure detailed in Section 6.1, part of which is a summary table indicating $s(\{i\})$, $u(\{i\})$, $uB_{item}(i, \{\})$, and $uB_{fpe}(\{i\})$ for each item i in $TS(pat(N))$. Such a summary table is also attached to N in Fig. 2.

Node 0 represents the empty pattern {}, which is not a high utility pattern (at line 4). According to the summary table of $TS(\{\})$ attached to Node 0 in Fig. 2, all items are relevant, that is, W = {a, b, c, d, e, f, g} (at line 5), and neither the closure property nor the singleton property holds (at lines 6–9).Nodes 1 and 2 will not be enumerated as $uB_{fpe}(\{a\})$ and $uB_{fpe}(\{b\})$ are below the threshold. DFS will search Nodes 4, 8, 16, 32, and 64 (at lines 8-11) as $uB_{fpe}(\{i\}) \geq minU$ for $i \in$ {c, d, e, f, g}.

When visiting Node 4, the utilities and bounds for $i \in$ {a, b} are already maintained in $TS(\{c\})$ which is derived from $TS(\{\})$ by a pseudo projection presented in Section 6.3. It turns out that {c} represented by Node 4 is not a high utility pattern, neither the closure property nor the singleton property holds, and Node 5 is pruned as $uB_{fpe}(\{a, c\}) < minU$. Subsequently, DFS will recursively visits Node 6 where the closure property holds and hence {a, b, c} is output as a high utility pattern without visiting Node 7.

The remaining nodes that will be explored in the order of depth-first search are Node 8, Node 10 where the closure property holds, Node 16, Node 24 where the closure property also holds, Node 32, and Node 64 where the singleton property holds and hence {a, b, c, d, e, g} is identified as the only high utility pattern without searching the subtree under Node 64.

In short, d^{2}HUP only enumerates Nodes 0, 4, 6, 8, 10, 16, 24, 32, and 64, a total of nine nodes, in finding all the high utility patterns, while the entire reverse set enumeration tree consists of $2^7 = 128$ nodes.

6 Efficient Implementation by Representing Transactions Scalably

When growing the reverse set enumeration tree, the d^{2}HUP algorithm needs to determine $TS(pat(N))$ for each node N being visited for computing utilities and utility upper bounds for prefix extensions of $pat(N)$ as shown at line 1 and line 13 in Algorithm 1. How to represent and maintain $TS(pat(N))$ together with related utilities and upper bounds is the key to the scalability and efficiency of the proposed algorithm.

This section introduces a linear data structure, CAUL, namely a Chain of Accurate Utility Lists, which is not tree-based, nor graph-based, but simply consists of linear lists. CAUL maintains the original utility information for each enumerated pattern in a way that enables us to compute the utility and to estimate tight utility upper bounds efficiently.

6.1 Scalable Representation of Utility Information

For the pattern, $pat(N)$, represented by a reverse set enumeration tree node N currently visited by a depth-first search, we use CAUL to maintain the utility information in the transaction set $TS(pat(N))$ of the node N, denoted by $TS_{caul}(pat(N))$, which is necessary for computing the utilities and upper bounds of its prefix extensions. $TS_{caul}(pat(N))$ consists of two parts, utility lists and a summary table.

For each transaction $t \in TS(pat(N))$, there is a utility list holding the utilities of all the items in t relevant in growing prefix extensions of $pat(N)$. That is, $\forall j \in fpe(pat(N), t) \setminus pat(N)$, $u(j, t)$ are stored in the utility list in the imposed ordering Ω. In addition, an extra element is appended to the utility list to maintain $u(pat(N), t)$.

The summary table maintains an entry for each distinct item j relevant in growing prefix extensions of $pat(N)$, which

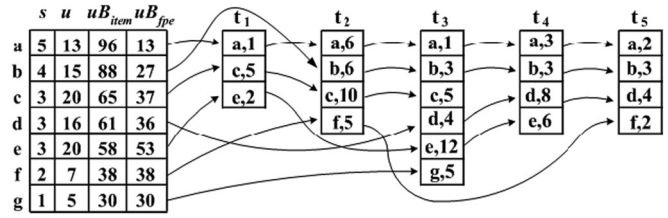

Fig. 3. $TS_{caul}(\{\})$: CAUL representing transaction set $TS(\{\})$, derived from D in Table 1, for the null root of the reverse set enumeration trees in Figs. 1 and 2.

is denoted as a quintuple, $summary[j] = (s[j], u[j], uB_{item}[j], uB_{fpe}[j], link[j])$, as described in the following. Summary entries are also arranged in the imposed ordering Ω.

- $s[j]$ for $s(\{j\} \cup pat(N))$ by Definition 2;
- $u[j]$ for $u(\{j\} \cup pat(N))$ by Definition 3;
- $uB_{item}[j]$ for $uB_{item}(j, pat(N))$ by Corollary 2;
- $uB_{fpe}[j]$ for $uB_{fpe}(\{j\} \cup pat(N))$ by Theorem 1, or Corollary 3;
- $link[j]$ for a chain threading together the occurrences of the same item j in the utility lists.

$TS_{caul}(\{\})$ is built by scanning the database D and the external utility table XUT, filtering out globally irrelevant items, and computing $s[j]$, $u[j]$, $uB_{item}[j]$, and $uB_{fpe}[j]$ for each relevant item j.

For example, Fig. 3 shows $TS_{caul}(\{\})$ for Node 0 in Figs. 1 and 2. The first list represents t_1 with its first element storing item a and $u(a, t_1) = 1$, its second element storing item c and $u(c, t_1) = 5$, and so on. In any of the five lists, there is no extra element to hold the utility of {} in the transaction since it is 0. The occurrences of item a in all the five lists are threaded by $link$[a] of the first summary entry. The other components, s[a], u[a], uB_{item}[a], and uB_{fpe}[a], of the first summary entry keep $s(\{a\})$, $u(\{a\})$, $uB_{item}(a, \{\})$, and $uB_{fpe}(\{a\})$ respectively.

6.2 Approach Generating No Candidates Enabled

One difference between our CAUL and the data structures by prior algorithms [4], [15], [24], [29], [38] is that CAUL keeps the original utility information for each transaction, while the latter keep the utility estimate, TWU, instead. This is the root cause why we are able to mine high utility patterns without generating candidates, while the prior algorithms have to take a two-phase, candidate generation approach.

We characterize our approach as one without candidate generation in the following senses.

- While the two-phase, candidate generation approach first generates high TWU patterns (candidates) and then identifies high utility patterns from high TWU patterns, our approach directly finds high utility patterns without generating any high TWU patterns (candidates).
- For pattern X being enumerated and represented by a reverse set enumeration tree node N, the utility of X is already computed in the CAUL of the parent node P of N, i.e., in $TS_{caul}(pat(P))$. Therefore, our approach can read off the utility of X from the CAUL and determine if X is a high utility pattern before X is enumerated, and thus X is not a candidate.
- Our approach keeps in main memory only the pattern currently being enumerated, i.e., only the path of the reverse set enumeration tree that is being explored, while the prior algorithms materialize all high TWU patterns (candidates) in main memory in order to identify high utility patterns in an additional screening step.

6.3 Efficient Computation by Pseudo Projection

For any node N and its parent node P with $pat(N) = \{i\} \cup pat(P)$ on the reverse set enumeration tree, $TS_{caul}(pat(N))$ can be efficiently computed by a pseudo projection [26], where the pseudo $TS_{caul}(pat(N))$ shares the same memory space with $TS_{caul}(pat(P))$.

The utility lists of the pseudo $TS_{caul}(pat(N))$ are delimited by following $link[i]$ in $TS_{caul}(pat(P))$, and the summary entry for each item $j \prec i$ of the pseudo $TS_{caul}(pat(N))$ is computed by scanning each delimited utility list.

Algorithm 2. PseudoProject($TS_{caul}(pat(P)), i$)

```
1  foreach relevant item j ≺ i do
2    (s[j], u[j], uB_item[j], uB_fpe[j], link[j]) ← 0
3  end foreach
4  foreach utility list t threaded by link[i] do
5    u(pat(N), t) ← u(pat(P), t) + u(i, t)
6    Σ ← u(pat(N), t)
7    foreach relevant item j ∈ t ∧ j ≺ i by Ω do
8      s[j] ← s[j] + 1
9      u[j] ← u[j] + u(j, t) + u(pat(N), t)
10     Σ ← Σ + u(j, t)
11     uB_fpe[j] ← uB_fpe[j] + Σ
12   end foreach
13   foreach relevant item j ∈ t ∧ j ≺ i by Ω do
14     uB_item[j] ← uB_item[j] + Σ
15     thread t into the chain by link[j]
16   end foreach
17 end foreach
```

This computation process is implemented by Algorithm 2, namely PseudoProject, which is called by d^{2}HUP at line 13 in Algorithm 1.

For example, Fig. 4a shows the pseudo $TS_{caul}(\{c\})$ for Node 4 derived from $TS_{caul}(\{\})$ for Node 0 in Figs. 1 and 2. Three utility lists, t_1, t_2, and t_3, are delimited from within $TS_{caul}(\{\})$, and the first two entries of the summary table of $TS_{caul}(\{\})$ are updated, which forms the pseudo $TS_{caul}(\{c\})$. We can do pseudo projection recursively in two senses.

First, $TS_{caul}(pat(P))$ for a reverse set enumeration tree node P can be projected to the **pseudo** $TS_{caul}(pat(N)$ for a child node N of P, which in turn can be projected to the **pseudo** $TS_{caul}(pat(C)$ for a grand child node C of P. For example, Fig. 4b shows the pseudo $TS_{caul}(\{b, c\})$, for Node 6 in Figs. 1 and 2, by calling PseudoProject (the pseudo $TS_{caul}(\{c\})$, b).

Second, although $TS_{caul}(pat(P))$ is partially modified in projecting to the pseudo $TS_{caul}(pat(N))$ and to the pseudo $TS_{caul}(pat(C))$, we can derive the pseudo $TS_{caul}(pat(S))$ for a sibling node S of node N from this modified $TS_{caul}(pat(P))$. For example, we can build the pseudo $TS_{caul}(\{d\})$ when visiting Node 8 since the threading chain starting from $link$[d] in $TS_{caul}(\{\})$ is untouched before visiting Node 8.

6.4 Materialization versus Pseudo Projection

We may only keep $TS_{caul}(\{\})$ in memory and get TS_{caul} for all patterns by recursive pseudo projection, which is the

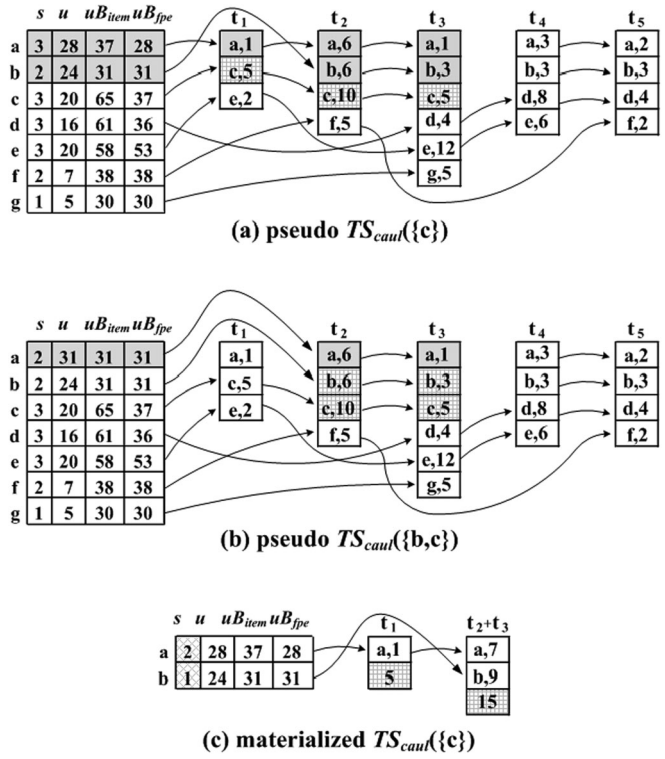

Fig. 4. Projections from $TS_{caul}(\{\})$.

most scalable way to maintain original utility information and is the core of our approach. Furthermore, we can optimize our approach by considering two trade-offs.

Maximum number of rounds γ for irrelevant item filtering. The body of the PseudoProject algorithm iterates multiple rounds to filter out irrelevant items. We introduce a parameter, γ, to make a tradeoff between the benefit of more pruning by tightening upper bounds and the additional computational overhead for iterative irrelevant item filtering. When no more irrelevant items are identified by Corollary 2 or a maximum number of rounds has been reached, the iteration terminates.

Materialization threshold ϕ for space-time tradeoff. We introduce a materialization threshold ϕ to make a tradeoff between the scalability resulted from representing $TS_{caul}(pat(N))$ by pseudo projection and the efficiency resulted from leaving out irrelevant items by materializing $TS_{caul}(pat(N))$. When the percentage of relevant items is below the threshold, a materialized copy will be made by copying the pseudo $TS_{caul}(pat(N))$ to memory space separate from $TS_{caul}(pat(P))$ so that irrelevant items are left out, $u(pat(N),\ t) = u(pat(P),\ t) + u(i,\ t)$ is stored in an extra element for each list t, and lists with an identical set of items are merged.

For example, Fig. 4c shows the materialized $TS_{caul}(\{c\})$ where the summary entries are copied from 4a, the first list has an element for the only relevant item, a, in t_1, and a special element for $u(\{c\},\ t_1)$. As the sets of relevant items are identical, t_2 and t_3 are merged into the second list.

7 Comparative Evaluation

We evaluate our d^{2}HUP algorithm by comparing with the state-of-the-art algorithms, TwoPhase [29], IHUP$_{TWU}$ [4], UP-Growth [38], and HUIMiner [28]. The code of TwoPhase [29] and HUIMiner [28] were provided by the original authors. Due to unavailability, we implemented an improved version of IHUP$_{TWU}$ [4] and an improved version of UP-Growth [38], namely IHUP$^+_{TWU}$ and UP$^+_{UPG}$ respectively. The latter employ a search tree to compactly represent all candidates, facilitate fast matching between candidates and transactions, and improve the efficiency of the second phase greatly. When mining large databases, UP$^+_{UPG}$ is even faster than HUIMiner [28], and UP-Growth [38] simply did not report the running time of the second phase because it is too long [38].

TABLE 2
Characteristics of Six Datasets

Dataset	$\lvert t\rvert$	$\lvert I\rvert$	$\lvert D\rvert$	Type
T10I6D1M	10 : 33	1,000	933,493	mixed
WebView-1	2.5 : 267	497	59,602	sparse
Chess	37 : 37	76	3,197	dense
Chain-store	7.2 : 170	46,086	1,112,949	sparse
T20I6D1M	20 : 49	1,000	999,287	mixed
Foodmart	4.8 : 27	1,559	34,015	dense

Six datasets are used in comparative experiments. T10I6D1M and T20I6D1M with utility information are exactly the same dataset as in [29], and Chain-store is the same as in [4], [28], [29], [38]. WebView-1 and Chess contain no utility information originally. We generate the utility information by following the method in [29]. So do [28], [38]. Thus, Chess with utility information used by [28], [38] and us share the same features, but are not the same datasets. Foodmart is from the Microsoft foodmart database. The datasets are summarized by Table 2 where the first column is the name of a dataset, the second ($|t|$) is the average and maximum length of transactions, the third ($|I|$) is the number of distinct items, the fourth ($|D|$) is the number of transactions, and the fifth (Type) is a rough categorization based on the number of high utility patterns to be mined, partially depending on the minimum utility threshold as in Table 3.

The minimum utility thresholds $minU$(percent) in terms of the percentage of overall utility for each dataset are selected in a way that the results can be verified with [4], [28], [29], [38]. The experiments were performed on a PC with 1.80 GHz CPU and 8 GB memory running CentOS 6.3. The parameter setting of $\gamma = 3$ and $\phi = 0.5$ is used as the default for d^{2}HUP unless specified otherwise, which is discussed in Section 8.1.

7.1 Enumerated Patterns and Candidates

Table 3 shows the number of high utility patterns ($hups$), the maximum length of high utility patterns (ml), the numbers of patterns enumerated by our d^{2}HUP algorithm and HUIMiner respectively, and the numbers of candidates generated in the first phase by UP$^+_{UPG}$, IHUP$^+_{TWU}$, and TwoPhase respectively, with different datasets and varying $minU$.

Our d^{2}HUP algorithm enumerates less patterns than HUIMiner, and the numbers of candidates generated by UP$^+_{UPG}$, IHUP$^+_{TWU}$, and TwoPhase are 1 to 2 orders, 2 to 4 orders, and over 3 orders of magnitude more than the number of patterns enumerated by d^{2}HUP respectively. The reasons are as follows.

TABLE 3
Patterns Enumerated by d^{2}HUP, HUIMiner, and Candidates by UP$^+_{UPG}$, IHUP$^+_{TWU}$, TwoPhase

(a) T10I6D1M

$minU$	$hups(ml)$	d^{2}HUP	HUIM.	UP+	IHUP+	T.P.
0.5%	25 (1)	639	639	673	711	226K
0.1%	389 (8)	1,092	1,871	8,735	32K	611K
.01%	873K (17)	1.06M	1.21M	1.7M	2.1M	-
.005%	1.6M (17)	1.8M	2M	2.4M	2.9M	-

(b) WebView-1

$minU$	$hups(ml)$	d^{2}HUP	HUIM.	UP+	IHUP+	T.P.
3.2%	2 (1)	11	19	159	71K	95K
3%	2 (1)	21	27	171	12M	-
2.1%	6 (1)	203	15.7M	1,809	-	-
1.9%	11K (148)	160K	-	-	-	-

(c) Chess

$minU$	$hups(ml)$	d^{2}HUP	HUIM.	UP+	IHUP+	T.P.
60%	0 (0)	0	0	34	299K	304K
30%	7.4K (16)	13.9K	16.8K	1.34M	43.7M	-
20%	1.1M (20)	1.08M	1.61M	43.7M	-	-
10%	124M (25)	83.6M	158M	-	-	-

(d) Chain-store

$minU$	$hups(ml)$	d^{2}HUP	HUIM.	UP+	IHUP+	T.P.
0.25%	17 (1)	152	135	1,123	1,577	628K
0.1%	80 (2)	1,661	1,629	3,876	7,117	7.3M
0.01%	3,839 (6)	21.4K	28.4K	66.6K	286K	-
.005%	12K (11)	43.1K	71.2K	225K	9.48M	-

(e) T20I6D1M

$minU$	$hups(ml)$	d^{2}HUP	HUIM.	UP+	IHUP+	T.P.
0.5%	25 (1)	786	786	798	3,311	334K
0.1%	669 (11)	2,302	10.3K	66.4K	207K	-
.01%	923K (17)	1.24M	1.54M	2.92M	6.1M	-
.005%	1.8M (17)	2.35M	2.81M	9.55M	23.4M	-

(f) Foodmart

$minU$	$hups(ml)$	d^{2}HUP	HUIM.	UP+	IHUP+	T.P.
0.1%	198 (1)	1,553	1,556	1,559	1,559	1.2M
0.02%	1,467 (27)	1,590	2,097	26.1K	-	-
0.015%	1.72M (27)	1.58M	3.78M	-	-	-
0.01%	74.2M (27)	17.5M	91.5M	-	-	-

- Our utility upper bounds are tighter than the upper bounds by HUIMiner and the TWUs by UP$^+_{UPG}$, IHUP$^+_{TWU}$, and TwoPhase respectively, which results in stronger pruning.
- We propose the lookahead strategy to improve the pruning while HUIMiner does not.
- CAUL enables tightening upper bounds iteratively in the mining process while the data structures by UP$^+_{UPG}$, IHUP$^+_{TWU}$, and TwoPhase cannot.

7.2 Running Time and Memory Usage

Fig. 5 shows the running time by the five algorithms. For example, for T10I6D1M with $minU = 0.01\%$, d^{2}HUP takes 27 seconds, HUIMiner 154, UP$^+_{UPG}$ 101, IHUP$^+_{TWU}$ 109, and TwoPhase runs out of memory. The observations are as follows.

First, d^{2}HUP is up to 1 to 3 orders of magnitude more efficient than UP$^+_{UPG}$, IHUP$^+_{TWU}$, and TwoPhase. In particular,

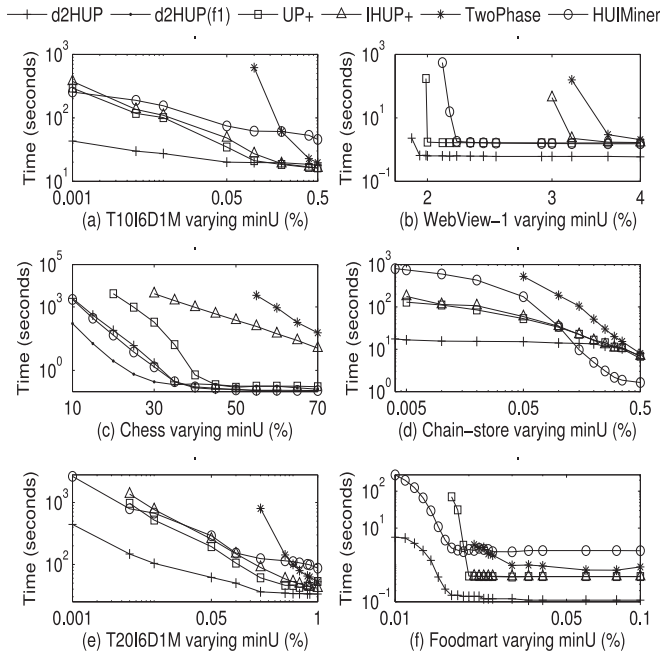

Fig. 5. Running time versus $minU$ (percent).

d^{2}HUP is up to 6.6, 6.8, 7.8, 261, 472, and 1,502 times faster than UP$^+_{UPG}$ on T20I6D1M, T10I6D1M, Chain-store, WebView-1, Foodmart, and Chess respectively. The reasons are as follows:

- The number of patterns enumerated by d^{2}HUP is much smaller than the number of candidates generated by UP$^+_{UPG}$, IHUP$^+_{TWU}$, and TwoPhase. Thus, d^{2}HUP even takes less time than the first phase of the latter algorithms.
- The latter materialize candidates and need a second phase to match each candidate with transactions in the database, which causes scalability issue when the number of candidates is large, and has efficiency issue when the database is large.

Second, d^{2}HUP is up to 1 order of magnitude more efficient than HUIMiner [28]. Concretely, d^{2}HUP is up to 6.3, 6.5, 16.8, 45, 49, and 875 times faster than HUIMiner on T10I6D1M, T20I6D1M, Chess, Chain-store, Foodmart, and WebView-1 respectively. The reasons are:

- d^{2}HUP has stronger pruning and enumerates less patterns than HUIMiner.
- d^{2}HUP proposes the data structure, CAUL, that enables efficient computation, while HUIMiner employs inefficient join operations on a vertical data structure, which is also not scalable.

Third, for a small and dense dataset like Chess, d^{2}HUP with the parameter setting $\phi = 1$, depicted by 'd2HUP(f1)' in Fig. 5c, outperforms HUIMiner by over 1 order of magnitude, but d^{2}HUP with the default setting is a little bit less efficient than HUIMiner. The reason is that although the pseudo CAUL benefits scalability, it entails additional computation due to keeping irrelevant items. In this case, due to a huge number of high utility patterns, the scalability benefit does not excel while the additional computation cost is amplified. A good choice

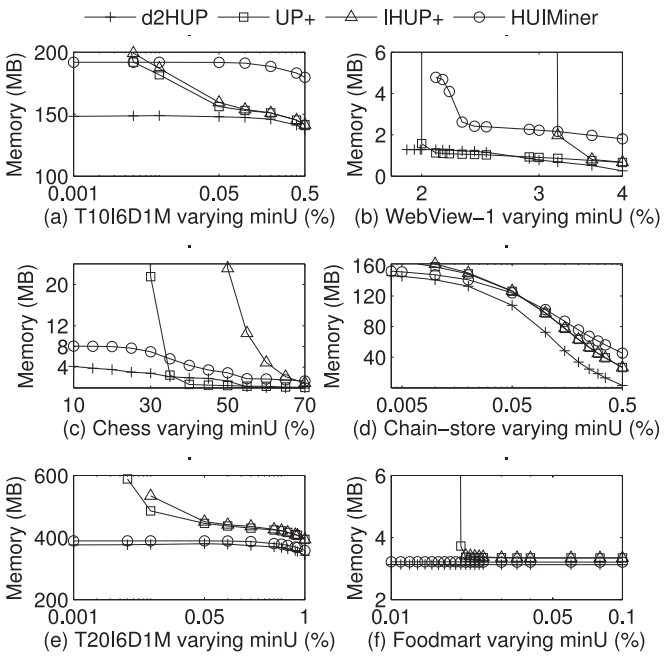

Fig. 6. Peak memory usage versus $minU$ (percent).

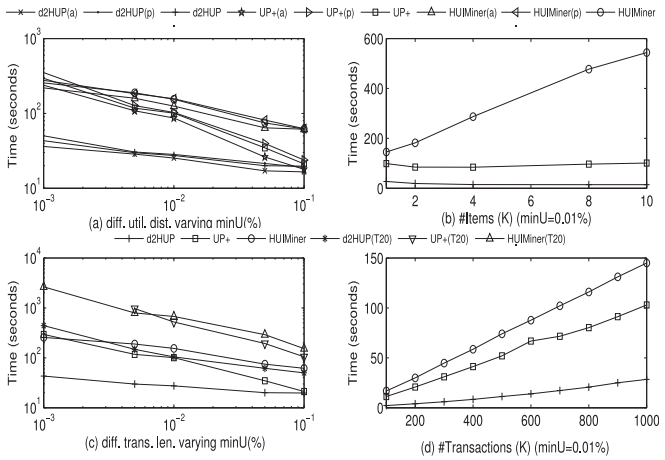

Fig. 7. Running time with varying data characteristic.

for d^{2}HUP is to make a materialized copy of the pseudo CAUL ($\phi = 1$).

Memory usage. We collect the peak memory usage statistics by every algorithm during its execution except TwoPhase as shown in Fig. 6. For example, for T10I6D1M with $minU = 0.1\%$, the peak memory usage by d^{2}HUP is 147 MB, and that by HUIMiner, by UP$^+_{UPG}$, and by IHUP$^+_{TWU}$ are 191, 153, and 154 MB respectively. The following is a summary.

- Our d^{2}HUP algorithm uses the least amount of memory because d^{2}HUP uses CAUL that is more compact than the vertical data structure by HUIMiner, and d^{2}HUP does not materialize candidates in memory while UP$^+_{UPG}$, IHUP$^+_{TWU}$, and TwoPhase do.
- The memory usage by UP$^+_{UPG}$ and IHUP$^+_{TWU}$ are 50 percent to 2 orders, and 90 percent to 2 orders of magnitude more than d^{2}HUP respectively. TwoPhase uses the most, and usually runs out of memory when $minU$ is small.

7.3 Comparison with Varying Data Characteristics

We compare our d^{2}HUP algorithm with the best prior algorithms, HUIMiner and UP$^+_{UPG}$ on varying data characteristics, including different utility distributions, changing number of items, different average length of transactions, and changing data size based on the T10I6D1M dataset as it is large and of a mixed type.

First, we generate external utilities anti-proportional to supports and proportional to supports, in addition to generating external utilities randomly. Fig. 7a shows the running time of the three algorithms on the respective resulting dataset with $minU$ ranging from 0.1 percent down to 0.001 percent. The running time with external utilities anti-proportional to supports, as depicted by '(a)', is less than that proportional to supports, as depicted by '(p)', and the latter is less than that generated randomly. For every utility distribution, d^{2}HUP takes much less time than HUIMiner and UP$^+_{UPG}$.

Second, we conduct a comparative experiment with the number of items ranging from 1K to 10K as shown in Fig. 7b. The running time by d^{2}HUP and by UP$^+_{UPG}$ do not change much because relevant items do not increase much with the increase of items. However, the running time by HUIMiner increases sharply with the increase of items.

Third, we evaluate the effect of the transaction lengths by comparing results both on T10I6D1M and on T20I6D1M. As in Fig. 7c where the results with T20I6D1M are depicted by '(T20)', the running time increases with the average length of transactions since both the average length and the number of high utility patterns also increase, so do the running time gaps among d^{2}HUP, HUIMiner, and UP$^+_{UPG}$.

Finally, Fig. 7d shows the scalability evaluation result with $|D|$ varying from 100K to 1000K. Clearly, d^{2}HUP has better scalability than HUIMiner and UP$^+_{UPG}$ according to the slopes of the curves.

8 Experimental Anatomy of d^{2}HUP

8.1 Analysis of Additional Pruning Techniques

First of all, let us note that our "basic approach" is to depth-first search the reverse set enumeration tree with pruning by basic upper bounding (Theorem 1) which is enabled by the pseudo projection of CAUL. In terms of the maximum number of rounds γ for iterative irrelevant item filtering and the materialization threshold ϕ, our "basic approach" corresponds to the setting of $\gamma = 1$ and $\phi = 0$ without lookahead.

Fig. 8 reports the running time with γ ranging from 1 to 6 and ϕ ranging from 0 to 1 both with and without lookahead. By comparing with Fig. 5 we can find that our "basic approach" already outperforms prior algorithms significantly. For example, for the T10I6D1M dataset ($minU = 0.001\%$), the running time of our "basic approach" is 36 seconds while that of UP$^+_{UPG}$ and IHUP$^+_{TWU}$ is 294 and 374 respectively.

Second, for all datasets and every setting of γ and ϕ, our lookahead strategy (Theorems 4 and 5) is beneficial in terms of decreasing the running time as depicted by 'with LK' in

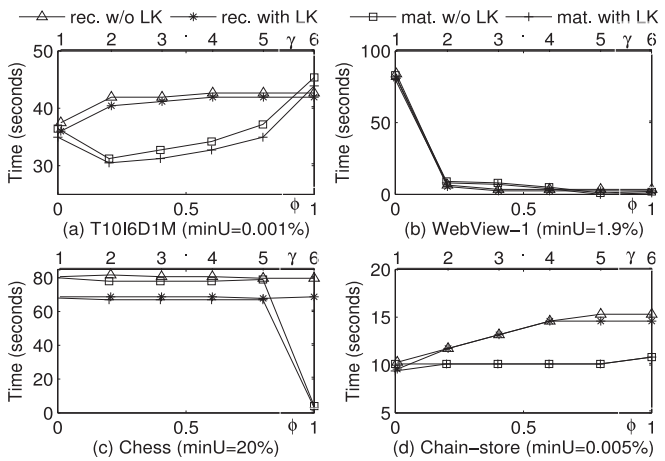

Fig. 8. Running time of d^{2}HUP with varying γ and ϕ.

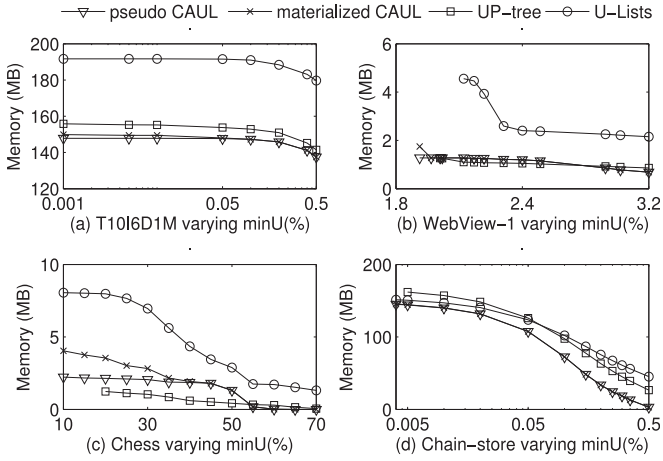

Fig. 9. Memory footprints of trans. representations.

Fig. 8. The lookahead is extremely helpful with dense datasets, for example, for Chess, and large datasets, for example, for T10I6D1M.

Third, in terms of pruning the search space, more irrelevant item filtering (Corollaries 2 and 3) by increasing γ, as depicted by 'rec', and more CAUL materialization by increasing ϕ, as depicted by 'mat', are very helpful as the utility upper bounds become tighter, which also decreases the running time with sparse data, for example, for WebView-1. However, it comes with additional computational overhead and thus the running time does not always decrease with the increase of γ and ϕ, for example, for Chain-store.

In short, while the setting of $\gamma = 1$ and $\phi = 0$ with lookahead is good, we recommend to use the setting of $\gamma = 3$ and $\phi = 0.5$ with lookahead as the default.

8.2 Evaluating the Transaction Representation

We evaluate the memory footprints of the pseudo CAUL and the materialized CAUL by d^{2}HUP, the memory footprint of the tree based structure, UP-tree [38], by UP^+_{UPG}, and the vertical data structure, U-Lists [28], by HUIMiner. The results as shown in Fig. 9 can be summarized as follows.

First, both the pseudo and materialized CAUL have a smaller memory footprint than UP-tree for large datasets, like Chain-store and T10I6D1M. The reason is that transactions in a large dataset are diversified, the memory saved by merging common prefixes of transactions does not offset the additional memory spent on auxiliary fields in each UP-tree node for maintaining the tree structural information.

Second, for small and dense datasets, UP-tree has higher compression ratio, and thus UP-tree has smaller memory footprint than CAUL, which is however not significant as in such a case CAUL also uses little memory, for example, less than 6 MB for Chess.

Third, U-List has the largest memory footprint because U-List has no compression at all.

Finally, materializing CAUL usually increases the memory footprint by a small percentage, but by a large percentage for a dense dataset, like Chess. In the latter case, the overall memory footprint may still be small as a dense dataset is usually not that large.

9 Conclusion and Future Work

This paper proposes a new algorithm, d^{2}HUP, for utility mining with the itemset share framework, which finds high utility patterns without candidate generation. Our contributions include: 1) A linear data structure, CAUL, is proposed, which targets the root cause of the two-phase, candidate generation approach adopted by prior algorithms, that is, their data structures cannot keep the original utility information. 2) A high utility pattern growth approach is presented, which integrates a pattern enumeration strategy, pruning by utility upper bounding, and CAUL. This basic approach outperforms prior algorithms strikingly. 3) Our approach is enhanced significantly by the lookahead strategy that identifies high utility patterns without enumeration.

In the future, we will work on high utility sequential pattern mining, parallel and distributed algorithms, and their application in big data analytics.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (61272306), and the Zhejiang Provincial Natural Science Foundation of China (LY12F02024). The authors would like to express their gratitude to the anonymous reviewers.

References

[1] R. Agarwal, C. Aggarwal, and V. Prasad, "Depth first generation of long patterns," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2000, pp. 108–118.

[2] R. Agrawal, T. Imielinski, and A. Swami, "Mining association rules between sets of items in large databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1993, pp. 207–216.

[3] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proc. 20th Int. Conf. Very Large Databases*, 1994, pp. 487–499.

[4] C. F. Ahmed, S. K. Tanbeer, B.-S. Jeong, and Y.-K. Lee, "Efficient tree structures for high utility pattern mining in incremental databases," *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 12, pp. 1708–1721, Dec. 2009.

[5] R. Bayardo and R. Agrawal, "Mining the most interesting rules," in *Proc. 5th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 1999, pp. 145–154.

[6] F. Bonchi, F. Giannotti, A. Mazzanti, and D. Pedreschi, "ExAnte: A preprocessing method for frequent-pattern mining," *IEEE Intell. Syst.*, vol. 20, no. 3, pp. 25–31, May/Jun. 2005.
[7] F. Bonchi and B. Goethals, "FP-Bonsai: The art of growing and pruning small FP-trees," in *Proc. 8th Pacific-Asia Conf. Adv. Knowl. Discovery Data Mining*, 2004, pp. 155–160.
[8] F. Bonchi and C. Lucchese, "Extending the state-of-the-art of constraint-based pattern discovery," *Data Knowl. Eng.*, vol. 60, no. 2, pp. 377–399, 2007.
[9] C. Bucila, J. Gehrke, D. Kifer, and W. M. White, "Dualminer: A dual-pruning algorithm for itemsets with constraints," *Data Mining Knowl. Discovery*, vol. 7, no. 3, pp. 241–272, 2003.
[10] C. H. Cai, A. W. C. Fu, C. H. Cheng, and W. W. Kwong, "Mining association rules with weighted items," in *Proc. Int. Database Eng. Appl. Symp.*, 1998, pp. 68–77.
[11] R. Chan, Q. Yang, and Y. Shen, "Mining high utility itemsets," in *Proc. Int. Conf. Data Mining*, 2003, pp. 19–26.
[12] S. Dawar and V. Goyal, "UP-Hist tree: An efficient data structure for mining high utility patterns from transaction databases," in *Proc. 19th Int. Database Eng. Appl. Symp.*, 2015, pp. 56–61.
[13] T. De Bie, "Maximum entropy models and subjective interestingness: An application to tiles in binary databases," *Data Mining Knowl. Discovery*, vol. 23, no. 3, pp. 407–446, 2011.
[14] L. De Raedt, T. Guns, and S. Nijssen, "Constraint programming for itemset mining," in *Proc. ACM SIGKDD*, 2008, pp. 204–212.
[15] A. Erwin, R. P. Gopalan, and N. R. Achuthan, "Efficient mining of high utility itemsets from large datasets," in *Proc. 12th Pacific-Asia Conf. Adv. Knowl. Discovery Data Mining*, 2008, pp. 554–561.
[16] P. Fournier-Viger, C.-W. Wu, S. Zida, and V. S. Tseng, "FHM: Faster high-utility itemset mining using estimated utility cooccurrence pruning," in *Proc. 21st Int. Symp. Found. Intell. Syst.*, 2014, pp. 83–92.
[17] L. Geng and H. J. Hamilton, "Interestingness measures for data mining: A survey," *ACM Comput. Surveys*, vol. 38, no. 3, p. 9, 2006.
[18] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2000, pp. 1–12.
[19] R. J. Hilderman, C. L. Carter, H. J. Hamilton, and N. Cercone, "Mining market basket data using share measures and characterized itemsets," in *Proc. PAKDD*, 1998, pp. 72–86.
[20] R. J. Hilderman and H. J. Hamilton, "Measuring the interestingness of discovered knowledge: A principled approach," *Intell. Data Anal.*, vol. 7, no. 4, pp. 347–382, 2003.
[21] M. Holsheimer, M. Kersten, H. Mannila, and H. Toivonen, "A perspective on databases and data mining," in *Proc. 1st Int. Conf. Knowl. Discovery Data Mining*, 1995, pp. 150–155.
[22] S. Krishnamoorthy, "Pruning strategies for mining high utility itemsets," *Expert Syst. Appl.*, vol. 42, no. 5, pp. 2371–2381, 2015.
[23] G.-C. Lan, T.-P. Hong, and V. S. Tseng, "An efficient projection-based indexing approach for mining high utility itemsets," *Knowl. Inf. Syst.*, vol. 38, no. 1, pp. 85–107, 2014.
[24] Y.-C. Li, J.-S. Yeh, and C.-C. Chang, "Isolated items discarding strategy for discovering high utility itemsets," *Data Knowl. Eng.*, vol. 64, no. 1, pp. 198–217, 2008.
[25] T. Y. Lin, Y. Y. Yao, and E. Louie, "Value added association rules," in *Proc. 6th Pacific-Asia Conf. Adv. Knowl. Discovery Data Mining*, 2002, pp. 328–333.
[26] J. Liu, Y. Pan, K. Wang, and J. Han, "Mining frequent item sets by opportunistic projection," in *Proc. 8th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2002, pp. 229–238.
[27] J. Liu, K. Wang, and B. Fung, "Direct discovery of high utility itemsets without candidate generation," in *Proc. IEEE 12th Int. Conf. Data Mining*, 2012, pp. 984–989.
[28] M. Liu and J. Qu, "Mining high utility itemsets without candidate generation," in *Proc. ACM Conf. Inf. Knowl. Manage.*, 2012, pp. 55–64.
[29] Y. Liu, W. Liao, and A. Choudhary, "A fast high utility itemsets mining algorithm," in *Proc. Utility-Based Data Mining Workshop SIGKDD*, 2005, pp. 253–262.
[30] S. Lu, H. Hu, and F. Li, "Mining weighted association rules," *Intell. Data Anal.*, vol. 5, no. 3, pp. 211–225, 2001.
[31] S. Morishita and J. Sese, "Traversing itemset lattice with statistical metric pruning," in *Proc. 19th ACM Symp. Principles Database Syst.*, 2000, pp. 226–236.
[32] J. Pei, J. Han, and V. Lakshmanan, "Pushing convertible constraints in frequent itemset mining," *Data Mining Knowl. Discovery*, vol. 8, no. 3, pp. 227–252, 2004.
[33] J. Pei, J. Han, H. Pinto, Q. Chen, U. Dayal, and M. Hsu, "PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth," in *Proc. 17th Int. Conf. Data Eng.*, 2001, pp. 215–224.
[34] A. Savasere, E. Omiecinski, and S. B. Navathe, "An efficient algorithm for mining association rules in large databases," in *Proc. 21st Int. Conf. Very Large Databases*, 1995, pp. 432–444.
[35] Y. Shen, Q. Yang, and Z. Zhang, "Objective-oriented utility-based association mining," in *Proc. IEEE Int. Conf. Data Mining*, 2002, pp. 426–433.
[36] A. Silberschatz and A. Tuzhilin, "On subjective measures of interestingness in knowledge discovery," in *Proc. ACM 1st Int. Conf. Knowl. Discovery Data Mining*, 1995, pp. 275–281.
[37] P. N. Tan, V. Kumar, and J. Srivastava,, "Selecting the right objective measure for association analysis," *Inf. Syst.*, vol. 29, no. 4, pp. 293–313, 2004.
[38] V. S. Tseng, B.-E. Shie, C.-W. Wu, and P. S. Yu, "Efficient algorithms for mining high utility itemsets from transactional databases," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 8, pp. 1772–1786, Aug. 2013.
[39] H. Yao and H. J. Hamilton, "Mining itemset utilities from transaction databases," *Data Knowl. Eng.*, vol. 59, no. 3, pp. 603–626, 2006.
[40] H. Yao, H. J. Hamilton, and C. J. Butz, "A foundational approach to mining itemset utilities from databases," in *Proc. SIAM Int. Conf. Data Mining*, 2004, pp. 482–486.
[41] H. Yao, H. J. Hamilton, and L. Geng, "A unified framework for utility-based measures for mining itemsets," in *Proc. ACM SIGKDD 2nd Workshop Utility-Based Data Mining*, 2006, pp. 28–37.
[42] U. Yun, H. Ryang, and K. H. Ryu, "High utility itemset mining with techniques for reducing overestimated utilities and pruning candidates," *Expert Syst. Appl.*, vol. 41, no. 8, pp. 3861–3878, 2014.
[43] M. J. Zaki, "Scalable algorithms for association mining," *IEEE Trans. Knowl. Data Eng.*, vol. 12, no. 3, pp. 372–390, May/Jun. 2000.
[44] M. J. Zaki and C. Hsiao, "Efficient algorithms for mining closed itemsets and their lattice structure," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 4, pp. 462–478, Apr. 2005.

Junqiang Liu received the BSc degree from Peking University in China, and the PhD degrees from Simon Fraser University in Canada and Zhejiang University in China, respectively. He is currently a professor at Zhejiang Gongshang University. His research interests include information security, privacy protection, data mining, and software engineering. He has published over 50 papers in prestigious journals and international conferences, including ACM SIGKDD, IEEE ICDE, and IEEE ICDM. He is a member of the IEEE.

Ke Wang received the PhD degree from the Georgia Institute of Technology. He is currently a professor at Simon Fraser University. His research interests include database technology, data mining, and knowledge discovery, with emphasis on massive data sets, graph and network data, and data privacy. He has published more than 100 research papers in database, information retrieval, and data mining conferences. He is currently an associate editor of the *ACM TKDD Journal*. He is a senior member of the IEEE.

Benjamin C.M. Fung is the Canada Research chair in Data Mining for Cybersecurity and an associate professor in the School of Information Studies (SIS), McGill University. He has over 80 refereed publications that span the research forums of data mining, privacy protection, cyber forensics, services computing, and building engineering. His data mining works in crime investigation and authorship analysis have been reported by media worldwide. He is a senior member of the IEEE.