Exploiting Succinct Constraints using FP-trees

Carson Kai-Sang Leung*
The University of British Columbia
Vancouver, BC, Canada
kleung@cs.ubc.ca

Laks V.S. Lakshmanan
The University of British Columbia
Vancouver, BC, Canada
laks@cs.ubc.ca

Raymond T. Ng
The University of British Columbia
Vancouver, BC, Canada
rng@cs.ubc.ca

ABSTRACT

Since its introduction, frequent-set mining has been generalized to many forms, which include constrained data mining. The use of *constraints* permits user focus and guidance, enables user exploration and control, and leads to effective pruning of the search space and efficient mining of frequent itemsets. In this paper, we focus on the use of *succinct constraints*. In particular, we propose a novel algorithm called *FPS* to mine frequent itemsets satisfying succinct constraints. The FPS algorithm avoids the generate-and-test paradigm by exploiting succinctness properties of the constraints in a FP-tree based framework. In terms of functionality, our algorithm is capable of handling not just the succinct aggregate constraint, but any succinct constraint in general. Moreover, it handles multiple succinct constraints. In terms of performance, our algorithm is more efficient and effective than existing FP-tree based constrained frequent-set mining algorithms.

General Terms

Algorithms, design, experimentation, management, measurement, performance, theory

Keywords

Data mining, constraints, succinctness, frequent sets, FP-trees

1. INTRODUCTION

Since its introduction [1], the problem of mining association rules, and the more general problem of finding frequent sets, from large databases has been the subject of numerous studies. These studies can be broadly divided into two "generations". In the first generation, all studies focused either on performance and efficiency issues (e.g., Apriori framework [2; 3], partitioning [15]), or on extending the initial notion of association rules to more general rules (e.g., mining long patterns [4], quantitative and multi-dimensional rules [7; 13], correlations and causal structures [6; 18]). Studies in this generation basically considered the data mining exercise in isolation.

Studies in the second generation explored how data mining can best interact with other key components in the broader picture of knowledge discovery and data mining. One key component is the database management system (DBMS), and some studies (e.g., the integration of association rule mining with relational DBMS [17], query flocks [20]) have explored how association rule mining can handshake with the DBMS most effectively. Another component, which is arguably even more important when it comes to knowledge discovery, is the human user. Studies of this kind allow users to specify the patterns to be mined according to their intention via the use of *constraints*. The algorithms developed exploit properties of these user-specified constraints, and provide support for user guidance and focus. Hence, the computation is more efficient and effective, and is limited to what interests the users.

To handle constraints in the process of mining frequent sets from large databases, many different approaches have been proposed over the past few years. The following are some examples. Srikant et al. [19] considered item constraints in association rule mining. Bayardo et al. [5] developed Dense-Miner to mine association rules with the user-specified consequent meeting "interestingness" constraints (e.g., minimum support, minimum confidence, minimum improvement). Garofalakis et al. [8] developed SPIRIT to mine frequent sequential patterns satisfying regular expression constraints. Ng et al. [14; 11] proposed a constrained frequent-set mining framework within which the user can use a rich set of constraints — including SQL-style aggregate constraints (e.g., Q_1, Q_2 in Figure 1) and non-aggregate constraints (e.g., Q_3, Q_7, Q_8, Q_9) — to guide the mining process to find only those rules satisfying the constraints. In Figure 1, the constraint $Q_1 \equiv min(S.Qty) \geq 500$ says that the minimum Qty value of all items in the set S is at least 500. The constraint $Q_3 \equiv S.Type \supseteq \{snack, soda\}$ says that the set S includes some items whose $Type$ is $snack$ and some items whose $Type$ is $soda$; the constraint $Q_7 \equiv S.Price = 25$ says that all items in the set S are of $Price$ equals 25. Ng et al. also developed the CAP algorithm in the constrained frequent-set mining framework mentioned above. Such an Apriori-based algorithm exploits properties of anti-monotone and/or *succinct constraints* to give as much pruning as possible. Constraints such as $Q_1, ..., Q_9$ in Figure 1 are succinct, because one can directly generate precisely all and only those itemsets satisfying the constraints (e.g., by using a member generating function [14], which does

*Person handling correspondence: Carson K.-S. Leung. Department of Computer Science, The University of British Columbia, 2366 Main Mall, Vancouver, BC, Canada V6T 1Z4. Phone: +1(604)822-4912. Fax: +1(604)822-5485.

Auxiliary information about items:

item	a	b	c	d	e	f	g
Qty	600	200	300	500	700	400	100
$Price$	40	10	25	30	20	35	15
$Type$	$beer$	$snack$	$soda$	$beer$	$beer$	$beer$	$beer$

$$
\begin{array}{rl}
\boldsymbol{Q_1}: & min(S.Qty) \geq 500 \\
\boldsymbol{Q_2}: & max(S.Price) \geq 30 \\
\boldsymbol{Q_3}: & S.Type \supseteq \{snack, soda\} \\
\boldsymbol{Q_4}: & min(S.Qty) \geq 500 \ \wedge \ max(S.Price) \geq 30 \\
\boldsymbol{Q_5}: & max(S.Price) \geq 30 \ \vee \ S.Type \supseteq \{snack, soda\} \\
\boldsymbol{Q_6}: & min(S.Qty) \geq 500 \ \vee \ [max(S.Price) \not\geq 30 \wedge S.Type \supseteq \{snack, soda\}] \\
\boldsymbol{Q_7}: & S.Price = 25 \\
\boldsymbol{Q_8}: & S.Type \subseteq \{beer, snack\} \\
\boldsymbol{Q_9}: & soda \in S.Type \\
\boldsymbol{Q_{10}}: & max(S.Price)/avg(S.Price) \leq 7
\end{array}
$$

Figure 1: Examples of Various Classes of Constraints

not require generation and testing of itemsets not satisfying the constraints). For instance, itemsets satisfying the constraint $Q_2 \equiv max(S.Price) \geq 30$ can be precisely generated by combining at least one item whose $Price \geq 30$ with some possible optional items (whose $Price$s are unimportant), thereby avoiding the substantial overhead of the generate-and-test paradigm. Among the above succinct constraints, $Q_1 \equiv min(S.Qty) \geq 500$ is also anti-monotone, because any superset of an itemset violating the constraint (i.e., containing an item whose $Qty < 500$) also violates the constraint. Grahne et al. [9] also exploited the anti-monotone and/or succinct constraints, but they mined valid correlated itemsets.

Like many Apriori-based algorithms, CAP mines frequent sets by generating candidates and checking their frequencies/support counts against the transaction database. To improve performance and efficiency of the mining process, Han et al. [10] proposed a FP-tree based framework. Specifically, the FP-growth algorithm in this framework constructs an extended prefix-tree structure, called FP-tree, to compress the original transaction database. Rather than employing the generate-and-test strategy of Apriori-based algorithms, it focuses on frequent pattern growth which is a *restricted* test-only approach (i.e., only test for frequency). Pei et al. [16] integrated such a FP-tree based mining framework with constraint pushing. They developed several FP-tree based algorithms such as $\mathcal{FIC}^{\mathcal{A}}$ and $\mathcal{FIC}^{\mathcal{M}}$ mainly to handle the so-called convertible constraints (e.g., Q_{10} in Figure 1). Since a *special class* of succinct constraints — namely succinct aggregate constraints — is convertible, their algorithms handle this class of constraints *indirectly* by first "converting" the constraints into anti-monotone or monotone ones.

Although the $\mathcal{FIC}$ algorithms avoid the generate-and-test (of frequent itemsets), they still require lots of testing for frequency and for constraint satisfaction, even when dealing with succinct constraints. In other words, the algorithms fail to exploit a nice property of succinct constraints: Given a succinct constraint C, one can efficiently enumerate all and only those itemsets that are guaranteed to satisfy C, thereby avoiding the substantial overhead of the generate-and-test paradigm (for frequent itemests satisfying C). Moreover, the $\mathcal{FIC}$ algorithms rely on a strong assumption that a *total order* exists over the set of items, i.e., mining can be done only by ordering items properly. Such an assumption does not hold in many situations. More importantly, the $\mathcal{FIC}$ algorithms cannot handle the "superset" constraint, i.e. a succinct constraint of the form $S.A \supseteq CS$ (e.g., $Q_3 \equiv S.Type \supseteq \{snack, soda\}$ in Figure 1), because there does not exist a total order for such a constraint. Furthermore, *multiple succinct constraints* having different or conflicting item ordering (e.g., $Q_1 \wedge Q_2 \equiv min(S.Qty) \geq 500 \ \wedge \ max(S.Price) \geq 30$, which have one ordering for Qty and another one for $Price$) cannot be handled effectively. Hence, a natural question to ask is: Based on the FP-tree framework, can we do better when we are dealing with succinct constraints? Can the pruning be done *once-and-for-all*? Can multiple succinct constraints be handled effectively?

The contribution of this work is to study how exploiting properties of succinct constraints can help frequent-set mining. More precisely, we develop a FP-tree based algorithm, called **FPS**, for **FP-tree based mining of Succinct constraints**. The algorithm pushes the succinct constraints deep inside the mining process, and thus leads to more effective pruning than the existing algorithms like $\mathcal{FIC}$. Figure 2 summarizes the salient functionalities of our proposed algorithm FPS as compared with the most relevant ones. More specifically, our technical contributions in this paper are as follows:

- *Functionality*: Our proposed algorithm FPS is capable of handling any succinct constraint including succinct non-aggregate constraints, "superset" constraints, and *multiple* succinct constraints. Note from Figure 2 that in terms of classes of constraints handled, CAP [14] looks similar to FPS. However, in terms of performance, CAP is different from FPS, as explained below.

- *Performance*: Although the succinct aggregate constraint can be handled by CAP, $\mathcal{FIC}$, and FPS (see Figure 2), the experimental results in Section 4 show that our proposed algorithm FPS is more efficient than CAP and $\mathcal{FIC}$. Reasons for the performance gain in FPS are that: (i) FPS is a FP-tree based algorithm, which avoids candidate generation, whereas the CAP

Classes of Constraints Handled	CAP [14]	$\mathcal{FIC}^{\mathcal{A}}$ and $\mathcal{FIC}^{\mathcal{M}}$[16]	Our Proposed Algorithm FPS
Succinct aggregate constraint (e.g., Q_1, Q_2)	√	√	√
Succinct "superset" constraint (e.g., Q_3)	√	–	√
Multiple succinct constraints (e.g., Q_4, Q_5, Q_6)	√	–	√
Succinct non-aggregate constraint (e.g., Q_7, Q_8, Q_9)	√	–	√
Non-succinct constraint (e.g., Q_{10})	–	√	–

Figure 2: Our Proposed FPS Algorithm vs. the Most Relevant Algorithms

algorithm is an Apriori-based algorithm, which relies on candidate generation; and (ii) FPS exploits the succinctness properties of the constraint and handles the succinct constraint *directly*, whereas the $\mathcal{FIC}$ algorithms incur overhead by doing so indirectly via the conversion of the succinct constraint into a convertible one.

The paper is organized as follows. In the next section, relevant background material is described. Section 3 presents an overview of our proposed algorithm FPS. Section 4 shows the experimental results. Finally, conclusions are presented in Section 5.

2. BACKGROUND

In this section, we first give definitions to various classes of ("rule"/"semantic") constraints. Then, we give an overview of $\mathcal{FIC}^{\mathcal{A}}$ and $\mathcal{FIC}^{\mathcal{M}}$ [16], which are FP-tree based mining algorithms for handling convertible constraints.

2.1 Constraints

There are several classes of "rule"/"semantic" constraints, which include: (i) succinct constraints, (ii) anti-monotone constraints, (iii) monotone constraints, and (iv) convertible constraints. Note that these classes overlap. For example, the constraint $Q_1 \equiv min(S.Qty) \geq 500$ in Figure 1 is succinct, anti-monotone, and convertible.

DEFINITION 1 (SUCCINCTNESS [14]). *Define* $\text{SAT}_C(\texttt{Item})$ *to be the set of itemsets that satisfy the constraint* C*. With respect to the lattice space consisting of all itemsets,* $\text{SAT}_C(\texttt{Item})$ *represents the pruned space consisting of those itemsets satisfying* C*. We use the notation* 2^I *to mean the powerset of* I*.*

(a) An itemset $I \subseteq \texttt{Item}$ *is a* succinct set *if it can be expressed as* $\sigma_p(\texttt{Item})$ *for some selection predicate* p*, where* σ *is the selection operator.*

(b) $SP \subseteq 2^{\texttt{Item}}$ *is a* succinct powerset *if there is a fixed number of succinct sets* $\texttt{Item}_1, \ldots, \texttt{Item}_k \subseteq \texttt{Item}$ *such that* SP *can be expressed in terms of the powersets of* $\texttt{Item}_1, \ldots, \texttt{Item}_k$ *using union and minus.*

(c) A constraint C *is* **succinct** *provided that* $\text{SAT}_C(\texttt{Item})$ *is a succinct powerset.* □

Consider the constraint $Q_3 \equiv S.Type \supseteq \{snack, soda\}$. Its pruned space consists of all those itemsets that contain at least one *snack* item and at least one *soda* item. Let $\texttt{Item}_1$, $\texttt{Item}_2$ and $\texttt{Item}_3$ respectively be the sets $\sigma_{Type=snack}(\texttt{Item})$, $\sigma_{Type=soda}(\texttt{Item})$ and $\sigma_{Type\neq snack \,\wedge\, Type \neq soda}(\texttt{Item})$. Then, $\texttt{Item}_1$ contains all the *snack* items, $\texttt{Item}_2$ contains all the *soda* items, and $\texttt{Item}_3$ contains neither a *snack* item nor a *soda* item. And, Q_3 is succinct because its pruned space $\text{SAT}_{Q_3}(\texttt{Item})$ can be expressed as:

$$2^{\texttt{Item}} - 2^{\texttt{Item}_1} - 2^{\texttt{Item}_2} - 2^{\texttt{Item}_1 \cup \texttt{Item}_3} - 2^{\texttt{Item}_2 \cup \texttt{Item}_3}.$$

Although $\text{SAT}_{Q_3}(\texttt{Item})$ is a complicated expression involving several unions and minuses, itemsets satisfying the succinct constraint Q_3 can be directly generate precisely (i.e., *without* generating and then excluding those itemset not satisfying Q_3). More specifically, every itemset ν satisfying Q_3 can be efficiently enumerated by combining: (i) a (*snack*) item from $\texttt{Item}_1$, (ii) a (*soda*) item from $\texttt{Item}_2$, and (iii) some possible optional items from any of $\texttt{Item}_1, \texttt{Item}_2$ and $\texttt{Item}_3$.

DEFINITION 2 (CONVERTIBILITY [16]). *A constraint* C *is* **convertible** *if and only if* C *is convertible anti-monotone or* C *is convertible monotone. A constraint* C *is* convertible anti-monotone *provided there is an order* $\mathcal{R}$ *on items such that when an ordered itemset* S *satisfies the constraint* C*, so does any prefix of* S*. A constraint* C *is* convertible monotone *provided there is an order* $\mathcal{R}'$ *on items such that when an ordered itemset* S *violates the constraint* C*, so does any prefix of* S*.* □

In addition to the above two classes of "rule"/"semantic" constraints, others include anti-monotone constraints and monotone constraints. A constraint C is **anti-monotone**[14] if and only if any superset of an itemset violating C also violates C; a constraint C' is **monotone** [16] if and only if any superset of an itemset satisfying C' also satisfies C'.
For example, constraints $Q_1 \equiv min(S.Qty) \geq 500$, $Q_2 \equiv max(S.Price) \geq 30$, and $Q_3 \equiv S.Type \supseteq \{snack, soda\}$ in Figure 1 are succinct. And, $Q_6 \equiv Q_1 \vee (\neg Q_2 \wedge Q_3)$ is also succinct. Among them, the succinct constraint Q_1 is also anti-monotone (and thus convertible anti-monotone), and the succinct constraint Q_2 is also convertible monotone. In addition to constraints Q_1 and Q_2, the constraint $Q_{10} \equiv max(S.Price)/avg(S.Price) \leq 7$ is another example of convertible constraints. For characterization of succinct and of anti-monotone constraints, refer to the works of Ng et al. [14]; for characterization of monotone and of convertible constraints, refer to the works of Pei et al. [16].
As one can observe from the above examples, succinct constraints can be of various forms. In general, succinct constraints can be further categorized into 3 subclasses:

1. **SAM constraints**, i.e. succinct anti-monotone constraints, such as $Q_1 \equiv min(S.Qty) \geq 500$, $Q_7 \equiv S.Price = 25$, $Q_8 \equiv S.Type \subseteq \{beer, snack\}$ in Figure 1;

2. **SUC constraints**, i.e. succinct non-anti-monotone constraints of a form not equivalent to $S.A \supseteq CS$ (where CS is a constant set for the attribute A of the set S), such as $Q_2 \equiv max(S.Price) \geq 30$, $Q_9 \equiv soda \in S.Type$; and

3. **"superset" constraints**, i.e. succinct non-anti-monotone constraints of a form equivalent to $S.A \supseteq CS$, such as $Q_3 \equiv S.Type \supseteq \{snack, soda\}$.

2.2 FP-tree Based Algorithms

In the previous section, we have seen the definitions of various classes of ("rule"/"semantics") constraints. In this section, we turn our attention to the two existing FP-tree based constrained mining algorithms [16]: $\mathcal{FIC}^{\mathcal{A}}$ for handling convertible anti-monotone constraints and $\mathcal{FIC}^{\mathcal{M}}$ for handling convertible monotone constraints.

2.2.1 FP-tree Based Algorithm for Convertible Anti-monotone Constraints: $\mathcal{FIC}^{\mathcal{A}}$

Like many FP-tree based algorithms, $\mathcal{FIC}^{\mathcal{A}}$ [16] consists of two main operations: (i) construction of FP-tree and (ii) growth of frequent patterns. The FP-tree is an extended prefix-tree structure to capture the content of transaction database. Frequent patterns are formed by first finding the singleton itemsets that are *frequent* (and *valid* — i.e. satisfying the constraint — for the constrained mining), and then *recursively* growing them. The entire mining process can be viewed as a divide-and-conquer approach of decomposing both the mining task and the transaction database according to the frequent patterns obtained so far. This leads to a focused search of smaller datasets.

While details of the $\mathcal{FIC}^{\mathcal{A}}$ algorithm can be found in the works of Pei et al. [16], we use the following example to highlight some major steps in the execution run of the algorithm.

Example 1. Consider the following transaction database $\mathcal{T}$:

Transactions in $\mathcal{T}$	Contents
t_1	$\{a, b, c, d\}$
t_2	$\{b, d, f\}$
t_3	$\{a, b, d, e\}$
t_4	$\{a, b, c, e, g\}$
t_5	$\{c, e, g\}$

with the auxiliary information about items as shown in Figure 1:

item	a	b	c	d	e	f	g
Qty	600	200	300	500	700	400	100
Price	40	10	25	30	20	35	15
Type	*beer*	*snack*	*soda*	*beer*	*beer*	*beer*	*beer*

Let constraint C be the SAM constraint $Q_1 \equiv min(S.Qty) \geq 500$ in Figure 1, and the minimum support threshold be 2 (i.e., 40%). Then, the $\mathcal{FIC}^{\mathcal{A}}$ algorithm "converts" C into a convertible anti-monotone constraint and mines valid frequent itemsets as follows. It first scans the database to check frequency/support count of each singleton itemset. As a result, it removes the infrequent itemset $\{f\}$. It obtains frequent singleton itemsets $\{e\}, \{a\}, \{d\}, \{c\}, \{b\}$ and $\{g\}$, which are sorted according to a prefix function order $\mathcal{R}$ such that all supersets of an itemset S having S as prefix violate the constraint C whenever S violates C. These frequent singleton itemsets are then checked for constraint satisfaction. Only frequent itemsets $\{e\}, \{a\}$ and $\{d\}$ are valid. The algorithm then scans the database again to construct a FP-tree. More precisely, for each database transaction, only frequent items — which follow the sorting order imposed by $\mathcal{R}$ — are kept in the tree. Afterwards, for each valid frequent itemset S, the algorithm forms a projected database (i.e., a collection of transactions having S as prefix). The above mining process is then applied recursively to each of the $\{e\}$-, $\{a\}$- and $\{d\}$-projected databases.

Note from the above execution run, items that are known to be *invalid* from the initial constraint checking (e.g, items c, b and g) are not removed, and they are kept in the initial FP-tree (i.e., the FP-tree for built for the transaction database $\mathcal{T}$) as well as FP-trees built for subsequent projected databases. Moreover, the constraint is checked not only at the initial step (i.e., when mining valid frequent singleton itemsets from the initial FP-tree), but also at *all recursive steps* (i.e., when mining valid frequent itemsets from FP-trees built for subsequent projected databases). □

2.2.2 FP-tree Based Algorithm for Convertible Monotone Constraints: $\mathcal{FIC}^{\mathcal{M}}$

In the previous section, we have reviewed how the $\mathcal{FIC}^{\mathcal{A}}$ algorithm handles a convertible anti-monotone constraint. In this section, we turn our attention to how the $\mathcal{FIC}^{\mathcal{M}}$ algorithm handles a convertible monotone constraint. The skeleton of the $\mathcal{FIC}^{\mathcal{M}}$ algorithm [16] is quite similar to that of the $\mathcal{FIC}^{\mathcal{A}}$ algorithm except the following in the $\mathcal{FIC}^{\mathcal{M}}$ algorithm:

- Frequent items are sorted according to a different prefix function order $\mathcal{R}'$ such that all supersets of an itemset S having S as prefix *satisfy* the convertible monotone constraint C whenever S *satisfies* C.

- Projected databases are formed not only for valid frequent itemsets, but also for *invalid* frequent itemsets.

- The constraint is checked at *some* but not necessarily all recursive steps. This is because once a frequent itemset S satisfies the constraint C, every subsequent frequent itemset derived from the corresponding projected database (at subsequent recursive steps) has S as prefix, and thus satisfies C too.

For lack of space, we do not show further details. We leave as an exercise for the reader to apply the $\mathcal{FIC}^{\mathcal{M}}$ algorithm to the same database/setting as in Example 1 for mining valid frequent itemsets satisfying the SUC constraint $Q_2 \equiv max(S.Price) \geq 30$ in Figure 1. The reader may notice from the execution run that: (i) transactions that does not contain any item satisfying the constraint (e.g, transaction t_5, which contains no item whose $Price \geq 30$) are not removed, and they are kept in the initial FP-tree as well as FP-trees built for subsequent projected databases; (ii) projected databases are formed for *all* frequent itemsets, regardless of their constraint satisfiability; and (iii) the constraint is checked not only at the initial step (i.e., when mining valid frequent singleton itemsets from the initial FP-tree built for the transaction database $\mathcal{T}$), but also at *many recursive steps* (i.e., when mining valid frequent itemsets from FP-trees built for subsequent projected databases).

Algorithm FPSam
INPUT: A transaction database, a SAM constraint C, and a minimum support threshold.
OUTPUT: All valid frequent itemsets (i.e., itemsets that are frequent and satisfying C).

1. Generate all valid singleton itemsets (i.e., singleton itemsets satisfying C).
2. Build a FP-tree, which excludes all invalid items (i.e., excludes items not generated in Step 1).
3. Apply the usual FP-tree based mining algorithm (e.g., FP-growth) to the FP-tree built in Step 2.

Figure 3: Skeleton of the FPSam Algorithm

3. EXPLOITING SUCCINCT CONSTRAINTS

Recall from Section 2.1 that there are 3 subclasses of succinct constraints, namely SAM, SUC, and "superset" constraints. In this section, we show how we can exploit each of these 3 subclasses of succinct constraints *directly* to help mining valid frequent itemsets. Specifically, we develop an algorithm called **FPS**, for **FP-tree based mining of Succinct constraints**. The algorithm consists of two main components: One for handling succinct anti-monotone constraints (FPSam in Section 3.1), and another for handling succinct non-anti-monotone constraints (FPSuc in Sections 3.2 and 3.3).

3.1 FPSam for Handling Succinct Anti-monotone Constraints

Recall from Section 2.2.1, the $\mathcal{FIC}^{\mathcal{A}}$ algorithm handles a *special* class of SAM constraints — SAM aggregate constraint — by treating it as a convertible anti-monotone one. By so doing, the algorithm suffers from several problems/weaknesses.

Problem 1 — Redundant items kept in FP-trees. At the initial step of the $\mathcal{FIC}^{\mathcal{A}}$ algorithm, the only pruning is the removal of infrequent items. Even though all frequent items are checked for constraint satisfaction, all *valid* and *invalid* frequent items are kept in the initial FP-tree, and thus in its projected databases and their corresponding FP-trees. But, do we need to keep the invalid items? The answer is no, because any frequent itemset ν satisfying a SAM constraint (whether aggregate or otherwise) is composed of *only* valid items (i.e., items satisfying the constraint individually):

$$\nu \subseteq \text{set of valid items} \tag{1}$$

In other words, all invalid items do not contribute to the final answer set of valid frequent itemsets, thereby can be removed without penalty.

Problem 2 — Unnecessary constraint checking at recursive steps/projected databases. For a convertible anti-monotone constraint C, if $\{x_i\}$ satisfies C but $\{x_j\}$ violates C, then it is possible but not necessary that $\{x_i\}\cup\{x_j\}$ satisfies C, for some item x_j ordered after item x_i with respect to a prefix function order $\mathcal{R}$ over the set of items. Hence, the $\mathcal{FIC}^{\mathcal{A}}$ algorithm requires constraint checking at every recursive step/projected database. However, due to succinctness, if $\{x_j\}$ violates a SAM constraint C', then any superset of $\{x_j\}$ also violates C'. So, the constraint checking on $\{x_i\}\cup\{x_j\}$ becomes unnecessary once $\{x_j\}$ has been identified as invalid.

Solution: Our FPSam algorithm keeps only valid items in the initial FP-tree, i.e. excludes all invalid items. As FP-trees and projected databases formed subsequently at recursive steps depend on the initial FP-tree, such a reduction in the size of the initial tree has a positive effect in reducing sizes of all FP-trees built for subsequent projected databases. It also leads to reduction in computation and space at recursive steps. Moreover, because of succinctness, all frequent itemsets satisfying a SAM constraint can be formed by solely using the valid items (i.e., the only items kept in the trees). Hence, our FPSam algorithm just needs to generate all valid items at the initial step, i.e. pruning for constraint satisfaction can be done *once-and-for-all*, thereby avoiding all unnecessary constraint checking at recursive steps/projected databases. Figure 3 shows the skeleton of the FPSam algorithm. The example below shows an execution run of the algorithm.

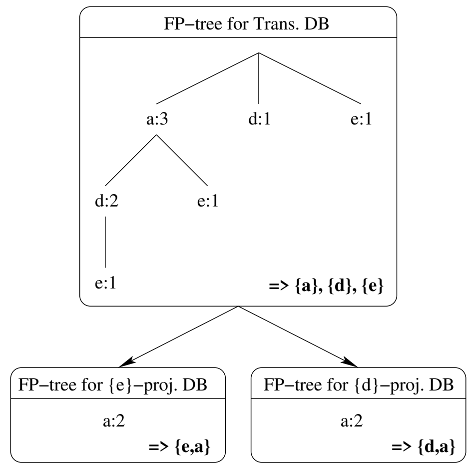


Figure 4: FPSam Mines Frequent Itemsets Satisfying the SAM Constraint Q_1

Example 2. With the same database/setting and the same SAM constraint $Q_1 \equiv min(S.Qty) \geq 500$ as in Example 1, the FPSam algorithm mines valid frequent itemsets as follows. It first generates all valid itemsets; it removes all infrequent items (e.g., item f) and excludes all invalid items (e.g., items b, c and g), because these removed/excluded itemsets do not contribute to the final answer set of valid frequent itemsets. Then, FPSam builds a FP-tree containing only valid items (i.e., items a, d and e) as shown in Figure 4. (The frequency/support count of each item is shown in the Figure, e.g., "a:3" in the initial tree indicates that the frequency of $\{a\}$ is 3.) The usual FP-tree based mining process (with only frequency check) can be applied to the tree. More specifically, the algorithm forms an $\{e\}$-projected database and finds that the itemset $\{e, a\}$ is frequent (and thus valid). Similarly, the algorithm forms a $\{d\}$-projected database and finds that the itemset $\{d, a\}$ is frequent. Hence, FPSam finds all valid frequent itemsets $\{a\}, \{d\}, \{e\}, \{d, a\}$ and $\{e, a\}$.

Note that no constraint checking is required at recursive

Algorithm FPSuc

INPUT: A transaction database, a SUC constraint C, and a minimum support threshold.

OUTPUT: All valid frequent itemsets (i.e., itemsets that are frequent and satisfying C).

1. Generate items that belong to the mandatory group (i.e., items satisfying C) and those that belong to the optional group (i.e., items not satisfying C); and partition the items into the appropriate groups.
2. For each transaction, put mandatory items before optional items.
3. Build a FP-tree, which excludes redundant/"non-contributing" transactions.
4. For each valid frequent singleton itemset (i.e., itemset containing a mandatory item), form a projected database, to which the usual FP-tree based mining algorithm (e.g., FP-growth) is applied.

Figure 5: Skeleton of the FPSuc Algorithm

steps/projected databases. Moreover, the FPSam algorithm does not rely on any prefix function order. Hence, items can be arranged consistently in the FP-trees according to any item-ordering scheme (e.g., arrange items in decreasing frequency order, which helps reduce the tree size). ☐

3.2 FPSuc for Handling Succinct Non-anti-monotone SUC Constraints

Recall from Section 2.2.2, the $\mathcal{FIC}^{\mathcal{M}}$ algorithm handles a *special* class of SUC constraints — SUC aggregate constraint — by treating it as a convertible monotone one. By so doing, the algorithm suffers from several problems/weaknesses, which are similar but not identical to those of the $\mathcal{FIC}^{\mathcal{A}}$ algorithm. Before we discuss these problems and our solution, let us understand the *key difference* between the itemsets satisfying a SUC constraint to those satisfying a SAM constraint: Any frequent itemset ν satisfying a SUC constraint (whether aggregate or otherwise) is composed of one or more **mandatory items** (i.e., items satisfying the constraint) and some **optional items** (i.e., items *not* satisfying the constraint):

$$\nu = \{x\} \cup \gamma \tag{2}$$

where

$x \in$ set of mandatory items, and

$\gamma \subseteq$ (set of mandatory items $\cup$ set of optional items).

Recall from the previous section that the main contribution of FPSam is to set up an appropriate initial FP-tree so that the remaining mining process can be performed as usual using a FP-tree based mining algorithm (e.g., FP-growth). Such a contribution was achieved by: (i) excluding all those items that do not contribute to the final answer set of valid frequent itemsets, and (ii) exploiting succinctness so as to avoid unnecessary constraint checking at recursive steps/projected databases. Below, we discuss the problems/weaknesses of the $\mathcal{FIC}^{\mathcal{M}}$ algorithm and show how we solve these problems in a similar fashion as we did for the SAM constraints.

Problem 1 — Redundant transactions kept in FP-trees. Although all frequent items are checked for constraint satisfaction at the initial step of the $\mathcal{FIC}^{\mathcal{M}}$ algorithm, *all* transactions — including those containing no mandatory items — are kept in the initial FP-tree, and thus in its projected databases and their corresponding FP-trees. But, we do not need to keep all transactions, because any frequent itemset ν satisfying a SUC constraint must contain a mandatory item. In other words, any transaction not containing a mandatory item does not contribute to the final answer set of valid frequent itemsets, thereby can be removed without penalty.

Problem 2 — Excessive formation of projected databases & unnecessary constraint checking at recursive steps/projected databases. For a convertible monotone constraint C, if $\{x_i\}$ violates C, then it is possible but not necessary that $\{x_i\} \cup \{x_j\}$ satisfies C, for some item x_j ordered after item x_i with respect to a prefix function order $\mathcal{R}'$ over the set of items. Hence, the $\mathcal{FIC}^{\mathcal{M}}$ algorithm: (i) needs to form a projected database for every frequent itemset regardless of its constraint satisfiability, and (ii) requires constraint checking at recursive steps/projected databases. But, if we arrange the items in such a way that mandatory items come before optional items, then $\{x_i\}$ violates a SUC constraint C' implies that x_i is an optional item and any item x_j ordered after item x_i is also optional. Thus, $\{x_i\} \cup \{x_j\}$ also violates C'. So, forming projected databases on $\{x_i\}$ does not lead to the finding of valid frequent itemsets, thereby can be skipped without penalty. Moreover, the constraint checking on $\{x_i\} \cup \{x_j\}$ becomes unnecessary once $\{x_i\}$ has been identified as optional.

Solution: Our FPSuc algorithm excludes all transactions containing no mandatory items (i.e., excludes all redundant/"non-contributing" transactions), and keeps only those containing at least one mandatory item. Such a reduction in the size of the initial FP-tree leads to: (i) reduction in sizes of all FP-trees built for subsequent projected databases, and thus saving in computation and space at recursive steps; and (ii) more "accurate" support counts (see Example 3), because items in those transactions that are not contributing to the final answer set are no longer counted. Moreover, instead of sorting items according to the prefix function order $\mathcal{R}'$, the FPSuc algorithm *partitions* items into two groups, namely *"mandatory group"* and *"optional group"*. The algorithm puts items from the mandatory group *before* any item from the optional group. (In other words, mandatory items appear *below* optional items in the FP-tree.) Then, all frequent itemsets satisfying a SUC constraint must be "extensions" of an item from the mandatory group, i.e. all valid frequent itemsets must be grown from an item in the mandatory group. Hence, due to succinctness, our FPSuc algorithm just needs to:

(i) form projected databases *only* for valid frequent itemsets; and

(ii) generate all mandatory and all optional items at the initial step, i.e. pruning for constraint satisfaction can be done *once-and-for-all*, thereby avoiding all unnecessary constraint checking at recursive steps/projected databases.

Figure 5 shows the skeleton of the FPSuc algorithm. The example below shows an execution run of the algorithm.

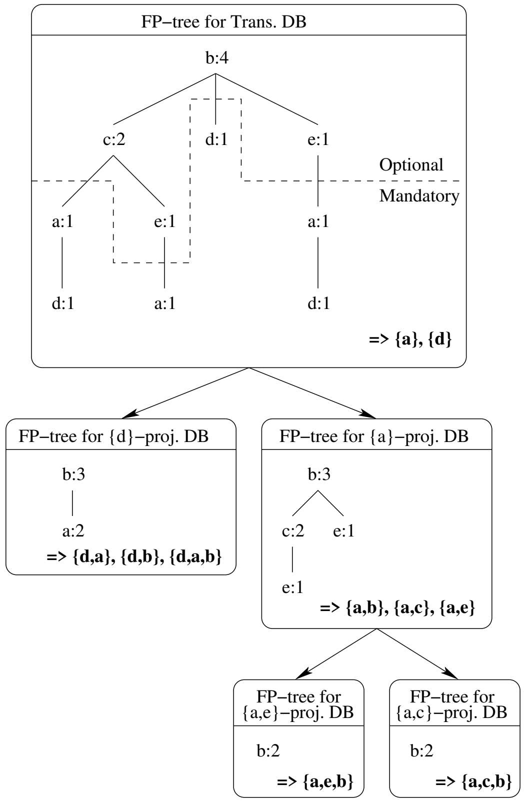


Figure 6: FPSuc Mines Frequent Itemsets Satisfying the SUC Constraint Q_2

Example 3. With the same database/setting as in Example 1 and the SUC constraint $Q_2 \equiv max(S.Price) \geq 30$ as discussed in Section 2.2.2, the FPSuc algorithm mines valid frequent itemsets as follows. It first generates and partitions items into 2 groups: (i) the *mandatory group* containing items a, d and f; and (ii) the *optional group* containing items b, c, e and g. Note that the item f is infrequent and is thus removed. The algorithm also removes transaction t_5, because it contains no mandatory item; such a removal leads to effective pruning and a more "accurate" support count. Specifically, the support count of item g is decremented to 1, which reflects its "accurate" frequency/support count in the set of non-redundant/"contributing" transactions. Since the item g is no longer frequent (i.e., not contributing to the final answer set of valid frequent itemesets), keeping it around is a waste of computation and space, and hence it is pruned. Then, FPSuc builds a FP-tree as shown in Figure 6. In the Figure, the dashed line indicates the *boundary* between mandatory and optional items in the initial FP-tree (i.e., the FP-tree built for transaction database). Here, mandatory items appear below optional items. Within each of the mandatory and the optional groups, items can be arranged consistently according to any item-ordering scheme. Note that for a SUC constraint, the boundary *only* exists in the initial FP-tree but *not* in any FP-tree built for subsequent projected databases. The reason is that once a projected database is formed for each valid frequent singleton itemset (e.g., $\{a\}, \{d\}$), there is no distinction between mandatory and optional items. In other words, once a mandatory item x is found for a valid itemset ν, any other item in ν can be chosen from mandatory or from optional groups (refer to Equation (2)). This explains why once FPSuc forms the $\{a\}$- and $\{d\}$-projected databases, the usual FP-tree based mining process (with only frequency check) can be applied recursively to these projected databases.

Let us complete the execution run. FPSuc finds frequent (and thus valid) itemsets $\{d, a\}, \{d, b\}$ and $\{d, a, b\}$ from the $\{d\}$-projected database; it finds frequent itemsets $\{a, b\}$, $\{a, c\}$ and $\{a, e\}$ from the $\{a\}$-projected database. As the mining process is applied recursively on this $\{a\}$-projected database, $\{a, c\}$- and $\{a, e\}$-projected databases are formed, and (valid) frequent itemsets $\{a, c, b\}$ and $\{a, e, b\}$ are found. Hence, all valid frequent itemsets $\{a\}, \{d\}$, $\{a, b\}$, $\{a, c\}$, $\{a, e\}, \{d, a\}, \{d, b\}, \{a, c, b\}, \{a, e, b\}$ and $\{d, a, b\}$ are found. Note that: (i) projected databases are formed *only* for valid frequent singleton itemsets; and (ii) no constraint checking is required at recursive steps/projected databases. □

3.3 FPSuc for Handling Superset Constraints

So far, we have shown how our proposed algorithms FPSam and FPSuc handle SAM constraints and SUC constraints respectively. Here, we turn our attention to the "superset" constraint, i.e. a succinct non-anti-monotone constraint of the form $S.A \supseteq CS$. Below, we discuss why the existing $\mathcal{FIC}^{\mathcal{M}}$ algorithm [16] cannot handle the superset constraint.

Problem — Reliance on total order over a set of items. Recall that both the $\mathcal{FIC}^{\mathcal{A}}$ and the $\mathcal{FIC}^{\mathcal{M}}$ algorithms rely on a strong assumption about the existence of a total order over a set of items. The algorithms can only exploit a constraint by ordering items properly. Since there does not exist a *total order* for the superset relation, the existing $\mathcal{FIC}^{\mathcal{M}}$ algorithm fails to handle the superset constraint. Given that our proposed FPSuc algorithm does not rely on the total order, can we handle the superset constraint? The answer is yes, because in our framework, the *key difference* between a superset constraint and a SUC constraint is *the number of mandatory groups*. More precisely, any frequent itemset ν satisfying a superset constraint $S.A \supseteq \{cs_1, ..., cs_l\}$ is composed of at least one item from each mandatory group and some optional items (i.e., items that do not belong to any mandatory group):

$$\nu = \{x_1, ..., x_l\} \cup \gamma \tag{3}$$

where

$$x_i \in \text{mandatory group}_i \text{ and}$$
$$\gamma \subseteq \left(\left(\bigcup_{i=1}^{l} \text{mandatory group}_i \right) \cup \text{optional group} \right).$$

Solution: To handle the superset constraint (or any succinct constraint having multiple mandatory groups), we generalize our proposed FPSuc algorithm by making the following modifications:

- Instead of partitioning items into two groups, the FPSuc algorithm is modified to partition items into $(l + 1)$ groups, namely *"mandatory group$_1$"*, ..., *"mandatory group$_l$"* and *"optional group"*, where l is the number of mandatory groups/items in each valid itemset ν. For example, for the constraint $Q_3 \equiv S.Type \supseteq \{snack, soda\}$ in Figure 1, the algorithm partitions items into 3 groups: (i) mandatory group$_1$ containing items whose $Type$ is *snack* (e.g., item b in Example 1), (ii) mandatory group$_2$ containing items whose $Type$ is *soda* (e.g., item c), and (iii) optional group containing items whose $Type$ is neither *snack* nor *soda* (e.g., items a, d, e, f and g).

- Regarding the removal of redundant/"non-contributing" transactions, the FPSuc algorithm is modified to remove transactions not containing an item from *each* mandatory group. For example, for the constraint Q_3 above, any transaction not containing both a *snack* item and a *soda* item (e.g., transactions t_2, t_3 and t_5 in Example 1) can be removed.

- The *key difference* between the original FPSuc and this modified version is at recursive steps. For a SUC constraint, once the initial FP-tree is built and projected databases are formed for mandatory items, the usual FP-tree based mining algorithm can be applied. However, with the superset constraint (having l mandatory groups), the modified FPSuc algorithm needs to ensure that an $\{x_1, x_2, ..., x_j\}$-projected database is formed *only* for each item x_j in mandatory group$_j$. More specifically, the modified FPSuc algorithm forms an $\{x_1\}$-projected database for each item x_1 in mandatory group$_1$ at the initial step. In each $\{x_1\}$-projected database, items are rearranged so that those from mandatory group$_2$ come before those from the other groups; the algorithm then forms an $\{x_1, x_2\}$-projected database for each item x_2 in mandatory group$_2$. Similar approach is carried out until the algorithm forms an $\{x_1, x_2, ..., x_l\}$-projected database (for each item x_l in mandatory group$_l$), to which the usual FP-tree based algorithm (e.g., FP-growth) can be applied afterwards. For example, with the same database/setting as in Example 1 and the constraint Q_3 above, the FPSuc algorithm first forms a $\{b\}$-projected database. Then, the algorithm forms a $\{b, c\}$-projected database, to which the usual FP-tree based mining algorithm like FP-growth is applied. Hence, the modified FPSuc finds all valid frequent itemsets $\{b, c\}$ and $\{b, c, a\}$.

3.4 Handling Multiple Succinct Constraints

Clearly, an algorithm for processing a constrained frequent-set query needs to deal with *multiple* constraints specified in the query. This raises the question of how to handle multiple succinct constraints. Below, we explain why the $\mathcal{FIC}$ algorithms cannot handle multiple constraints effectively.

Problem — Reliance on a proper item ordering. Recall that when using the $\mathcal{FIC}$ algorithms, constraints can be exploited only by ordering items properly. However, different constraints may require different and even conflicting item ordering. So to handle multiple succinct constraints, the best the $\mathcal{FIC}$ algorithms can do is to conduct a cost analysis in determining how to combine multiple order-consistent convertible constraints and how to select the most selective constraint among the order-conflicting ones. In many real-life situations, it is not unusual to get constraints having conflicting item ordering. For example, for constraints $Q_1 \wedge Q_2 \equiv min(S.Qty) \geq 500 \wedge max(S.Price) \geq 30$ in Figure 1, one item ordering exists for $S.Qty$ and a conflicting one for $S.Price$. In this situation, the best the $\mathcal{FIC}$ algorithms can do is to pick the most selective constraint, *ignore* another constraint during the mining process, and perform a *post-processing* step to check the mined itemsets against the previously ignored constraint.

Solution: Our proposed algorithm FPS can exploit simultaneously *all* of the constraints in the multiple succinct constraints, because any Boolean combination of succinct constraints is also succinct [12]. For example, for constraints $Q_1 \wedge Q_2 \equiv min(S.Qty) \geq 500 \ \wedge \ max(S.Price) \geq 30$, any itemset ν satisfying $Q_1 \wedge Q_2$ is composed of: (i) at least one mandatory item (i.e., item whose $Qty \geq 500$ and whose $Price \geq 30$), and (ii) some optional items whose $Qty \geq 500$. When applying $Q_1 \wedge Q_2$ to the same database/setting as in Example 1, valid frequent itemsets $\{a\}, \{d\}, \{a, d\}$ and $\{a, e\}$ can be found. The reason is that any itemset ν satisfying $Q_1 \wedge Q_2$ is composed of: (i) at least one of items a and d, and (ii) possibly the optional item e (cf. Examples 2 and 3).

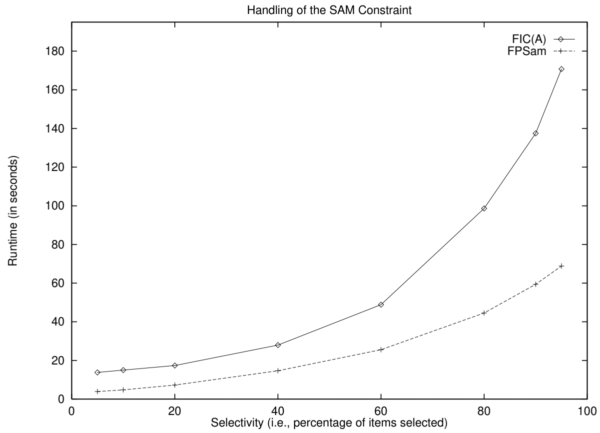
4. EXPERIMENTAL RESULTS

The experimental results cited below are based on a transaction database of 100k records with an average transaction length of 10 items, and a domain of 1000 items. The database was generated by the program developed at IBM Almaden Research Center [3]. (Additional databases, including connect-4 and mushroom from UC Irvine Machine Learning Depository, were used in the experiments. The results produced are consistent with those using the IBM database. So, for lack of space, we only show below the results using the IBM database.) Unless otherwise specified, we used a minimum support threshold of 0.01%. All experiments were run in a time-sharing environment in a 700 MHz machine. The reported figures are based on the average of multiple runs. In the experiment, we mainly compared two sets of algorithms that were implemented in C: (i) $\mathcal{FIC}^{\mathcal{A}}$ vs. FPSam, and (ii) $\mathcal{FIC}^{\mathcal{M}}$ vs. FPSuc.

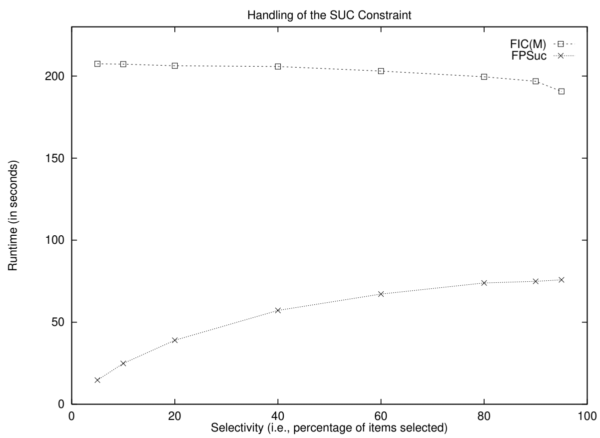
The y-axis of Figure 7 shows the runtime of the algorithms, and the x-axis shows the *selectivity* of the succinct constraint. A constraint with p% selectivity means p% of items are selected. It is observed from Figure 7(a) that as the selectivity of the SAM constraint decreases (i.e., fewer items are selected), the runtimes of both $\mathcal{FIC}^{\mathcal{A}}$ and FPSam decrease. But in terms of speedup, it is more beneficial to use FPSam than $\mathcal{FIC}^{\mathcal{A}}$, especially when the constraint has a lower selectivity. It is observed from Figure 7(b) that as the selectivity of the SUC constraint decreases, the runtime of FPSuc decreases but that of $\mathcal{FIC}^{\mathcal{M}}$ increases. Again, in terms of speedup, it is more beneficial to use FPSuc than $\mathcal{FIC}^{\mathcal{M}}$, especially when the constraint has a lower selectivity. A reason for the gain in performance, when compared FPS with $\mathcal{FIC}$, is that the FPS algorithm exploits succinctness property.

As shown in Figure 8(a), our FPS algorithm requires much smaller number of constraint checking than its counterparts in $\mathcal{FIC}$. The reason is that our algorithm does not require constraint checking at recursive steps/projected databases. In contrast, the $\mathcal{FIC}$ algorithms perform lots of unnecessary constraint checking at recursive steps. Moreover, it is interesting to note that as the selectivity of the succinct constraint decreases (i.e., fewer items are selected), three different trends were observed as follows:

- For FPS, the number of constraint checking is constant, because no constraint checking is required at recursive steps/projected databases. In other words,



(a) Handling of the SAM Constraint



(b) Handling of the SUC Constraint

Figure 7: Runtime

pruning for constraint satisfaction is done once-and-for-all.

- For $\mathcal{FIC}^{\mathcal{A}}$, the number decreases, because $\mathcal{FIC}^{\mathcal{A}}$ only forms projected databases for valid itemsets. So, the lower the selectivity of the constraint, the smaller is the number of valid itemsets.
- For $\mathcal{FIC}^{\mathcal{M}}$, the number increases, because $\mathcal{FIC}^{\mathcal{M}}$ requires constraint checking on those projected databases that have not yet contained a mandatory item. So, the lower the selectivity of the constraint, the smaller is the number of mandatory items, and thus the larger is the number of projected databases not yet containing a mandatory item.

Figure 8(b) shows the size of the initial FP-tree (i.e., the FP-tree built for the transaction database at the initial step). As observed, our FPS algorithm builds a much smaller tree than its counterparts in $\mathcal{FIC}$. The reasons are that: (i) in FPSam, only valid items are kept in the tree, whereas $\mathcal{FIC}^{\mathcal{A}}$ keeps all invalid frequent items (which are redundant) in addition to all valid frequent items in the tree; (ii) in FPSuc, only transactions that are contributing to the final answer set are kept in the tree, whereas $\mathcal{FIC}^{\mathcal{M}}$ keeps all transactions (including redundant ones) in the tree. Moreover, it is

Selectivity	Number of Constraint Checking			
	SAM Constraint		SUC Constraint	
	FPSam	$\mathcal{FIC}^{\mathcal{A}}$	FPSuc	$\mathcal{FIC}^{\mathcal{M}}$
20%	7,451	14,907	7,451	138,897
40%	7,451	36,061	7,451	72,475
60%	7,451	77,575	7,451	34,047
80%	7,451	146,519	7,451	14,223

(a) Number of Constraint Checking

Selectivity	Number of Nodes in the FP-tree			
	SAM Constraint		SUC Constraint	
	FPSam	$\mathcal{FIC}^{\mathcal{A}}$	FPSuc	$\mathcal{FIC}^{\mathcal{M}}$
20%	83,453	806,421	721,644	805,840
40%	246,353	806,421	789,495	805,840
60%	420,562	806,421	797,532	805,840
80%	610,686	806,421	800,883	805,840

(b) Number of Nodes in the Initial FP-tree

Selectivity	No. of "Counters" for "Support Count" Ops			
	SAM Constraint		SUC Constraint	
	FPSam	$\mathcal{FIC}^{\mathcal{A}}$	FPSuc	$\mathcal{FIC}^{\mathcal{M}}$
20%	121,994	1,044,372	1,098,185	3,023,243
40%	478,828	1,888,048	1,942,773	3,023,243
60%	1,054,499	2,490,335	2,509,923	3,023,243
80%	1,912,846	2,890,201	2,857,370	3,023,243

(c) No. of "Counters" for "Support Counts" Operations

Figure 8: Effectiveness of Exploiting Succinct Constraints

observed that the sizes of the initial trees for the $\mathcal{FIC}$ algorithms are constant, regardless of the constraint selectivity. This is an indication that the $\mathcal{FIC}$ algorithms do not exploit the succinct constraint to reduce the tree size (and to save computation and space). In contrast, the tree sizes for our proposed algorithm FPS decrease when the selectivity of the constraint decreases (i.e., when more items are pruned).

Figure 8(c) shows the total number of "counters" required for "support count" operations. This number depends on the total number and the sizes of FP-trees: The smaller the number and the sizes of the trees, the smaller is the number of "counters". It explains why the numbers of "counters" for FPSam, $\mathcal{FIC}^{\mathcal{A}}$, and FPSuc decrease when the selectivity of the constraint decreases (i.e., when fewer items are selected). Among FPSam and $\mathcal{FIC}^{\mathcal{A}}$, the former requires much fewer "counters" because it effectively reduces the number and the sizes of the trees. Moreover, it is interesting to note that $\mathcal{FIC}^{\mathcal{M}}$ requires the same number of "counters", regardless of the constraint selectivity. The reason is that the algorithm forms projected databases for all frequent itemsets, regardless of their constraint satisfiability. Many of these projected databases do not lead to the finding of valid frequent itemsets!

When the minimum support threshold increases, the runtime decreases, and so are the size of FP-tree, the number of constraint checking and the number of "counters" for "support count" operations.

We have also tested scalability with the number of transactions. The results show that our proposed algorithm FPS has a linear scalability.

Last but not the least, we have experimented with multiple

Number of Constraints	Runtime (in seconds)		
	FPSam	$\mathcal{FIC}^{\mathcal{A}}$	CAP
1	14.7s	26.4s	2192.5s
2	6.0s	27.0s	12.8s
3	4.0s	27.1s	5.5s

Figure 9: Effectiveness of Exploiting Multiple Succinct Constraints

succinct constraints. Figure 9 shows the result when multiple *independent* constraints, each with 40% selectivity, were used. Notice that our proposed algorithm FPSam is more efficient than its $\mathcal{FIC}$ counterpart, because our algorithm FPSam exploits all of the constraints in the multiple succinct constraints, whereas $\mathcal{FIC}^{\mathcal{A}}$ exploits only one of them (i.e., generating itemsets satisfying the most selective succinct constraint, testing them against all other constraints, and excluding those not satisfying any of the multiple constraints). When compared $\mathcal{FIC}$ with CAP [14], CAP outperforms $\mathcal{FIC}$ in some situations. The reason is that the CAP algorithm, like FPS, also exploits multiple succinct constraints. Moreover, our proposed algorithm FPSam, as expected, is more efficient than CAP, because our algorithm FPSam is FP-tree based (which avoids candidate generation), whereas CAP is Apriori-based (which relies on candidate generation).

5. CONCLUSIONS

A key contribution of this paper is to optimize the performance of, and to increase functionality of, FP-tree based constrained mining algorithms. To this end, we proposed and studied the novel algorithm of FPS. The algorithm handles succinct constraints directly and exploits succinctness properties, so that the constraints are pushed deep inside the mining process, leading to effective pruning. Moreover, the algorithm does not make any unrealistic assumption about the existence of total order over the set of items. It handles multiple succinct constraints (whether aggregate or otherwise) very efficiently and effectively.

In ongoing work, we are interested in exploring improvements to the FPS algorithm. For example, we are interested in investigating the handling of changes in succinct constraints. Along this direction, an interesting question to explore is how to handle succinct constraints efficiently and effectively when the FP-tree does not fit in memory.

6. REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. SIGMOD 1993*, pp. 207–216.

[2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules. *Advances in Knowledge Discovery and Data Mining*, chapter 12. AAAI/MIT Press, 1996.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. VLDB 1994*, pp. 487–499.

[4] R.J. Bayardo. Efficiently mining long patterns from databases. In *Proc. SIGMOD 1998*, pp. 85–93.

[5] R.J. Bayardo, R. Agrawal, and D. Gunopulos. Constraint-based rule mining in large, dense databases. In *Proc. ICDE 1999*, pp. 188–197. Extended version appears: *Data Mining and Knowledge Discovery*, **4**(2/3), pp. 217–240. July 2000.

[6] S. Brin, R. Motwani, and C. Silverstein. Beyond market basket: Generalizing association rules to correlations. In *Proc. SIGMOD 1997*, pp. 265–276.

[7] T. Fukuda, Y. Morimoto, S. Morishita, and T. Tokuyama. Data mining using two-dimensional optimized association rules: Scheme, algorithms, and visualization. In *Proc. SIGMOD 1996*, pp. 13–23.

[8] M.N. Garofalakis, R. Rastogi, and K. Shim. SPIRIT: Sequential pattern mining with regular expression constraints. In *Proc. VLDB 1999*, pp. 223–234.

[9] G. Grahne, L.V.S. Lakshmanan, and X. Wang. Efficient mining of constrained correlated sets. In *Proc. ICDE 2000*, pp. 512–521.

[10] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. SIGMOD 2000*, pp. 1–12.

[11] L.V.S. Lakshmanan, R. Ng, J. Han, and A. Pang. Optimization of constrained frequent set queries with 2-variable constraints. In *Proc. SIGMOD 1999*, pp. 157–168.

[12] L.V.S. Lakshmanan and R.T. Ng. A theory of succinctness and its application to constrained analysis and mining. In preparation, 2002.

[13] R.J. Miller and Y. Yang. Association rules over interval data. In *Proc. SIGMOD 1997*, pp. 452–461.

[14] R.T. Ng, L.V.S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proc. SIGMOD 1998*, pp. 13–24.

[15] J.S. Park, M.-S. Chen, and P.S. Yu. Using a hash-based method with transaction trimming for mining association rules. *IEEE TKDE*, **9**(5), pp. 813–825, Sept./Oct. 1997.

[16] J. Pei, J. Han, and L.V.S. Lakshmanan. Mining frequent itemsets with convertible constraints. In *Proc. ICDE 2001*, pp. 433–442.

[17] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *Proc. SIGMOD 1998*, pp. 343–354.

[18] C. Silverstein, S. Brin, R. Motwani, and J. Ullman. Scalable techniques for mining causal structures. In *Proc. VLDB 1998*, pp. 594–605.

[19] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In *Proc. KDD 1997*, pp. 67–73.

[20] D. Tsur, J.D. Ullman, S. Abiteboul, C. Clifton, R. Motwani, S. Nestorov, and A. Rosenthal. Query flocks: A generalization of association-rule mining. In *Proc. SIGMOD 1998*, pp. 1–12.