Efficient Dynamic Mining of Constrained Frequent Sets

LAKS V. S. LAKSHMANAN
The University of British Columbia, Vancouver, BC, Canada
CARSON KAI-SANG LEUNG
The University of Manitoba, Winnipeg, MB, Canada
and
RAYMOND T. NG
The University of British Columbia, Vancouver, BC, Canada

Data mining is supposed to be an iterative and exploratory process. In this context, we are working on a project with the overall objective of developing a practical computing environment for the human-centered exploratory mining of frequent sets. One critical component of such an environment is the support for the dynamic mining of constrained frequent sets of items. Constraints enable users to impose a certain focus on the mining process; dynamic means that, in the middle of the computation, users are able to (i) change (such as tighten or relax) the constraints and/or (ii) change the minimum support threshold, thus having a decisive influence on subsequent computations. In a real-life situation, the available buffer space may be limited, thus adding another complication to the problem.

In this article, we develop an algorithm, called DCF, for *Dynamic Constrained Frequent-set computation*. This algorithm is enhanced with a few optimizations, exploiting a lightweight structure called a *segment support map*. It enables DCF to (i) obtain sharper bounds on the support of sets of items, and to (ii) better exploit properties of constraints. Furthermore, when handling dynamic changes to constraints, DCF relies on the concept of a *delta member generating function*, which generates precisely the sets of items that satisfy the new but not the old constraints. Our experimental results show the effectiveness of these enhancements.

Categories and Subject Descriptors: H.2.8 [**Database Management**]: Database Applications—*data mining*

General Terms: Algorithms, Design, Experimentation, Human factors, Management, Performance, Theory

Additional Key Words and Phrases: Association rules, constraints, data mining, dynamic changes, frequent sets, limited buffer space

Authors' addresses: L. V. S. Lakshmanan, Department of Computer Science, The University of British Columbia, Vancouver, BC, Canada V6T 1Z4; email: laks@cs.ubc.ca; C. K.-S. Leung, Department of Computer Science, The University of Manitoba, Winnipeg, MB, Canada R3T 2N2; email: kleung@cs.umanitoba.ca; R. T. Ng, Department of Computer Science, The University of British Columbia, Vancouver, BC, Canada V6T 1Z4; email: rng@cs.ubc.ca.

1. INTRODUCTION

Since its introduction [Agrawal et al. 1993], the problem of mining association rules, as well as 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., the Apriori framework [Agrawal and Srikant 1994; Agrawal et al. 1996], partitioning [Park et al. 1997], sampling [Toivonen 1996; Gibbons and Matias 1998], the tree-based framework [Agarwal et al. 2000, 2001; Liu et al. 2002]), or on extending the initial notion of association rules to more general rules (e.g., multi-level rules [Han and Fu 1995], quantitative and multi-dimensional rules [Fukuda et al. 1996; Miller and Yang 1997], correlations and causal structures [Brin et al. 1997; Silverstein et al. 1998], mining long patterns [Bayardo 1998], ratio rules [Korn et al. 1998]). *Studies in this generation basically considered the data mining exercise in isolation*.

Studies in the second generation have explored how data mining can best interact with other key components in the broader picture of knowledge discovery. One key component is the database management system (DBMS). Some studies (e.g., the integration of association rule mining with relational DBMS [Sarawagi et al. 1998], query flocks [Tsur et al. 1998]) 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*. From this standpoint, studies in the first generation rely on a computational model where the computer does almost everything, and the user is un-engaged during the process. This model provides little support for (i) *user guidance and focus* (e.g., limiting the computation to what interests the user), and (ii) *user interaction* (e.g., dynamically changing the parameters midstream).

Note that it is not uncommon for the user to have certain broad phenomena in mind, on which to focus the mining. Without user focus, the mining process is treated as an impenetrable black-box—only allowing the user to set the thresholds at the beginning, showing the user all the frequent sets (or rules) at the end, but nothing in between. Moreover, the user cannot specify his interest via the use of constraints. As a result, the user often needs to wait for a long period of time for numerous frequent sets (or rules), out of which only a tiny fraction may be interesting to the user. This motivates the call for a computing environment where the user can express his focus. To this end, Ng, Lakshmanan, and their colleagues [Ng et al. 1998; Lakshmanan et al. 1999] proposed the following: (i) a constrained frequent-set mining framework within which the user can use a rich set of constraints to guide the mining process to find only those frequent sets satisfying the constraints; and (ii) a mining algorithm, called CAP, that exploits the constraints to ensure that the computational effort is proportional to the selectivity of the constraints.

Moreover, it is also important to note that the user usually does not know in advance appropriate parameters (e.g., an appropriate support threshold) to be used for the mining algorithms. An inappropriate choice yields, after possibly a long wait, either too many or too few frequent sets (or rules). This problem is worsened when lacking continuous feedback from the mining algorithms and when lacking user control during processing. This motivates the call for a computing environment where the user is continually engaged in the mining process, and can monitor the progress and dynamically make changes during processing. To this end, Hidber [1999] proposed a "continuous/online" mining algorithm, called Carma, that provides the user with continuous feedback on the numerous frequent sets being computed, and permits the user to change the support threshold dynamically.

The work presented in this article is motivated by the above observations and the observation that data mining in general (and frequent-set mining in particular) is a human-centered and exploratory process—not a one-shot exercise. This calls for an environment where: (i) the user can express his focus, (ii) the user is continually engaged in the process, and (iii) the user can monitor the progress and dynamically make changes—not only to the numerical parameters (e.g., the support threshold), but also to the constraints—thus having a decisive influence on subsequent computations. Such an environment needs to support both user focus and user interaction equally well. Moreover, it has to be extremely efficient, so as not to lose the user's attention. Last but not least, it has to be practical in the sense of not making unrealistic assumptions about resources (e.g., buffer space). It is important to note that, although advances in technology have improved today's hardware (e.g., cheap abundant memory), these advances also enable the user to easily collect a tremendous amount of data. For instance, while the size of available buffer space has doubled over a short period of time, the amount of data collected and used for mining may have grown more than double over the same period! Hence, having the capability of handling limited buffer space is always beneficial.

With respect to the above call for a practical environment for the human-centered exploratory mining of constrained frequent sets, the two bodies of work mentioned earlier fall short in different respects. Specifically, while the use of constraints is very effective in capturing user focus, CAP does not handle dynamic changes to the support threshold or to the constraints. While Carma can handle dynamic changes to the support threshold, it does not handle constraints, let alone deal with dynamic changes to the constraints. Moreover, Carma fails to work when the available buffer space is limited. The key general contribution of this article is the development of an algorithmic framework, called *DCF*, for *Dynamic Constrained Frequent-set computation*. Table I summarizes the salient functionalities of DCF as compared with CAP and Carma. Our technical contributions in this article are as follows:

—First, in Section 3, we introduce an algorithm called *DCF (Dynamic Constrained Frequent-set computation)* as an extension of Carma—but enhanced with a structure called a *segment support map (SSM)*. As a lightweight and easy-to-compute structure, the SSM improves the efficiency of DCF by obtaining sharper upper bounds on the support of sets of items (i.e., itemsets), which, in turn, helps to reduce the number of candidate itemsets that need to be counted for support. Experimental results presented in Section 8 show

Table I. DCF vs. the Most Relevant Algorithms

Salient Features	Carma	CAP	Proposed DCF
Handling dynamic changes to the support threshold	√	—	√
Handling constrained frequent-set queries	—	√	√
Performance enhancements with a segment support map	—	—	√
Handling dynamic changes to constraints	—	—	√
Capability of operating in limited buffer space	—	—	√

that the SSM can significantly improve the efficiency of our proposed DCF algorithm.

—Second, in Section 4, we extend the DCF algorithm introduced in Section 3 to handle constraints. DCF is optimized in the sense that it exploits properties of constraints to ensure that the search space is appropriately pruned as early as possible. One such optimization by DCF is achieved by using the SSM in a way different from that discussed in Section 3. This reinforces the usefulness of the SSM.

—Third, in Section 5, we show how DCF can handle dynamic changes to constraints. Constraints can be tightened or relaxed. The latter case is particularly computationally intensive. We show how the notion of *delta member generating functions* (*delta MGFs*) can be used to optimize the computation. Note that delta MGFs are nontrivial—and original—extensions of member generating functions because we are not dealing with just a solution space of the (conjunctions of) constraints, but are dealing with the *set difference* between the solution spaces of the relaxed and the original constraints. Experimental results presented in Section 8 show the effectiveness of this optimization.

—Fourth, in Section 6, we show how DCF can operate in limited buffer space. For the unconstrained case, Carma assumes that there is enough buffer space available for counting the support of *all* potentially frequent sets simultaneously. We contend that this assumption can be unrealistic for many scenarios. Hence, in Section 6, we show how DCF can handle realistic situations where the buffer space is limited, and show that DCF is designed to use the limited space as judiciously as possible to reduce I/O. Analytical results presented in Section 6.4 show that when there is enough buffer space, DCF is then just as efficient as Carma.

In sum, our goal is to develop an algorithmic framework for the efficient dynamic mining of constrained frequent sets. Our framework provides the user with (i) the opportunity to express his focus on the mining process via the use of constraints, and (ii) the flexibility to dynamically change the constraints. In addition to the capabilities of handling constraints and of handling dynamic changes to constraints, our framework also allows the user to operate in situations where the available buffer space is limited. In terms of performance, our proposed SSM structure reduces the number of candidates that need to be counted for support. Our proposed delta MGF generates precisely those itemsets satisfying the relaxed but not the original constraints (when handling dynamic relaxing changes to a constraint), thereby avoiding the generation and exclusion of those itemsets that do not satisfy the relaxed constraints. This, in turn, helps to speed up the computation.

This article is organized as follows: Section 2 provides related work and background material relevant to the rest of this article. We start presenting our current contributions in Section 3, where we introduce the DCF algorithm and the SSM structure. While Section 4 focuses on handling constraints, Section 5 focuses on handling dynamic changes to the constraints. Section 6 shows how DCF can operate in limited buffer space. Since the benefits of the SSM are not confined to DCF, Section 7 discusses other useful applications of the SSM. Section 8 gives experimental results. Finally, Section 9 presents conclusions and future work.

2. RELATED WORK AND BACKGROUND

Here, we first discuss the related work; we then give a brief summary of Carma, properties of constraints, and other background material relevant to the rest of this article. Readers familiar with the work of Ng et al. [1998] can skip Section 2.2, and those familiar with the work of Hidber [1999] can skip Section 2.3.

2.1 Related Work

Among numerous studies on frequent-set mining, the most relevant algorithms are CAP [Ng et al. 1998; Lakshmanan et al. 1999] and Carma [Hidber 1999]. They have been compared in Table I. More specifically, while the use of constraints is very effective in capturing user focus, CAP does not handle dynamic changes to the support threshold or to the constraints. While Carma can handle dynamic changes to the support threshold, it does not handle constraints, let alone deal with dynamic changes to the constraints. Moreover, Carma fails to work when the available buffer space is limited. In contrast, our proposed DCF algorithm is capable of (i) handling constraints, (ii) handling dynamic changes to constraints, (iii) handling dynamic changes to the support threshold, and (iv) operating in limited buffer space. In terms of performance, DCF is enhanced with a segment support map.

In addition, there are other relevant studies as well. One of them is the DHP algorithm [Park et al. 1997], which was proposed to speed up the Apriori algorithm by using a hash table. On the surface, the hash-based partioning done by this DHP algorithm looks similar to our SSM structure proposed in Section 3. However, there are several key differences between the DHP work and our proposal. First, in the DHP algorithm, the hash table is used to partition the itemsets of the same cardinalities (e.g., 2) into buckets; whereas, in our DCF algorithm, the SSM partitions *transactions*, not itemsets. Second, the SSM is intended to be a *static* data structure, whereas the hash table is created dynamically for each run of the DHP algorithm. Third, the DHP algorithm does not handle dynamic changes to the support threshold or to constraints, and it does not deal with buffer space concerns. Fourth, the SSM delivers additional benefits (e.g., more effective pruning of candidate itemsets) for skewed data, when compared with the DHP.

Another relevant algorithm is the Partition algorithm [Savasere et al. 1995], which divides transactions into partitions, and computes all itemsets that are frequent locally within the partition. Such an algorithm does not set up any structure like the SSM to reduce the number of candidates. Moreover, the algorithm does not handle dynamic changes to the support threshold or to constraints, and it does not deal with buffer space concerns.

In our previous paper [Lakshmanan et al. 2000], we presented a preliminary study on the SSM. Specifically, we focused on the effectiveness of the SSM in facilitating *online mining with Carma*. Due to its effectiveness, we incorporate the SSM into the DCF algorithm proposed in the current article. In addition to this performance enhancement with the SSM, the two other key technical focus areas of the current article are (i) the handling of dynamic changes to constraints and (ii) the handling of limited buffer space. These two areas have not been considered in our previous paper.

In another previous paper [Leung et al. 2002b], we presented methods to optimize the SSM. Specifically, we only focused on methods (or criteria to be used) for partitioning database transactions into segments so that the performance of the SSM can be further improved. We did not consider handling dynamic changes to constraints or handling limited buffer space in that paper. In contrast, two key technical focus areas of the current article are (i) the handling of dynamic changes to constraints and (ii) the handling of limited buffer space. Moreover, in the current article, we build an SSM by *arbitrarily* partitioning the transactions (i.e., not by partitioning the transactions in accordance with some segmentation criteria). As a preview, such an SSM can prune a large number of itemsets, and can bring a speedup that is several times better than without using the SSM. We expect to obtain a higher speedup if we were to use the "optimized" SSM presented in our previous paper [Leung et al. 2002b].

The next group of related work focuses on the FP-tree based (Frequent-Pattern tree based) mining algorithms [Han et al. 2000; Pei et al. 2001; Leung et al. 2002a]. While most algorithms find frequent sets by generating candidates and checking their support against the transaction database, the FP-tree based algorithms do not rely on candidate generation. One example of this group of algorithms is the FP-growth algorithm [Han et al. 2000], which was developed—based on an extended prefix-tree structure called an FP-tree—to improve the performance and efficiency of frequent-set mining. Specifically, the algorithm first constructs an FP-tree using the set of frequent singleton itemsets; it then maps each database transaction into a tree path. By successively concatenating those frequent singleton itemsets found in the tree path, frequent itemsets (of various cardinalities) can be generated. Our proposed technique is different from the FP-growth algorithm in several key aspects. First, FP-tree is query-dependent, because it is constructed based on the transaction database and the support threshold *minsup*. When *minsup* is relaxed, the FP-tree may need to be reconstructed. Whereas, the SSM is *query-independent*. Any change of *minsup* value does not affect the SSM. Second, FP-tree is constructed main memory-based. If the FP-tree representing the lattice of a large database does not fit into memory, recursive projections and partitioning are required to break such a database into smaller pieces. As a result, there will be a corresponding performance overhead. Whereas, in our proposed technique, the size of the SSM is independent of the size of the lattice. Third, the FP-growth algorithm does not handle constraints, let alone deal with dynamic changes to the constraints.

Recently, extensions of the FP-growth algorithm—called $\mathcal{FIC}$ [Pei et al. 2001], FPS [Leung et al. 2002a], and OpportuneProject [Liu et al. 2002]—were proposed to exploit constraints for mining constrained frequent sets. However, these algorithms still do not handle dynamic changes to the support threshold or to the constraints.

So far, we have discussed those studies that improve performance. Next, we turn our attention to studies that support user interaction. Toivonen [1996] proposed random sampling algorithms to find frequent itemsets from a random sample of a large transaction database. While this technique reduces I/O and computation costs, one potential weakness is that it may lead to inaccurate answers (caused by the presence of skewed data). Hellerstein and his colleagues [Hellerstein et al. 1997; Haas and Hellerstein 2001; Raman and Hellerstein 2002] proposed algorithms for online query processing (e.g., online aggregation), but not for mining. They used sampling techniques to give approximate query answers (though these answers can be refined continuously during the query processing). Gibbons and Matias [1998] proposed sampling-based summary statistics to improve approximate query answers. In other words, these answers are approximate, and only hold within a certain error bound. Moreover, these sampling techniques work reasonably well for numerical parameters like the support threshold, but do not necessarily work for constraints (e.g., $S.Type \supseteq \{snack, soda\}$). In contrast, the itemsets computed by our DCF algorithm are guaranteed to satisfy both numerical parameters (e.g., the support threshold) and the constraints.

2.2 The CFQ Language and Properties of Constraints

A *constrained frequent-set query* (*CFQ*) [Ng et al. 1998] is a query of the form $\{(S, T) \mid \mathcal{C}\}$, where S and T are set variables, and $\mathcal{C}$ is a conjunction of domain, class, and aggregate constraints. The aggregate operations allowed are the basic ones supported by SQL, that is, *min*(), *max*(), *sum*(), and *avg*(). For our examples below, we assume to have the transaction database `trans(TID, Itemset)` with auxiliary information contained in `itemInfo(Item, Type, Price)`. The CFQ $\{(S, T) \mid S.Type \cap T.Type = \emptyset\}$ asks for frequent itemset pairs whose associated type sets are disjointed. Similarly, the CFQ $\{(S, T) \mid S.Type = snack \wedge T.Type = beer \wedge max(S.Price) < min(T.Price)\}$ finds pairs of frequent itemsets of cheaper snack items and of more expensive beer items.

Constraints such as $S.Type = snack$ are called *1-var constraints* (i.e., constraints with one variable); constraints such as $max(S.Price) < min(T.Price)$ are called *2-var constraints* (i.e., constraints with two variables). A 1-var constraint can be *antimonotone*, and/or *succinct*, or neither. See Definitions 2.1 and 2.2.

Table II. Characterization of 1-var Constraints: Antimonotonicity and Succinctness

1-var Constraint	Antimonotone	Succinct
$support(S) \geq minsup$	yes	no
$S.A\ \theta\ cn$, where $\theta \in \{=, \leq, \geq\}$	yes	yes
$cn \in S.A$	no	yes
$S.A \supseteq CS$	no	yes
$S.A \subseteq CS$	yes	yes
$min(S.A) \leq cn$	no	yes
$min(S.A) \geq cn$	yes	yes
$max(S.A) \leq cn$	yes	yes
$max(S.A) \geq cn$	no	yes
$sum(S.A) \leq cn$	yes	no
$sum(S.A) \geq cn$	no	no
$avg(S.A)\ \theta\ cn$, where $\theta \in \{=, \leq, \geq\}$	no	no

Note: S is a set variable, A is an attribute of a set S, cn is a constant for A, and CS is a set of constants for A.

Definition 2.1 *Antimonotonicity* [*Ng et al.* 1998]. A 1-var constraint C is *antimonotone* if and only if for all itemsets S, S':

$$\text{if } S' \supseteq S \text{ and } S' \text{ satisfies } C, \text{ then } S \text{ satisfies } C.$$

Although antimonotonicity is not a new concept, one of the main contributions in Ng et al.'s paper [1998]—with respect to this notion—is a detailed analysis and a complete characterization of the class of 1-var constraints that are antimonotone. Table II shows such a characterization. The second column of the table identifies the constraints that are antimonotone, while the third column identifies the constraints that are succinct. (The concept of succinctness is discussed below.) See Ng et al.'s paper [1998] for more details.

Like in the classical Apriori algorithm, antimonotonicity helps to prune candidates when evaluating a CFQ. However, at each iteration of the algorithm, we still need to generate-and-test these candidates for satisfaction of the antimonotone constraints under consideration. Thus, the question we ask is whether there are classes of constraints for which pruning can be done once-and-for-all before any iteration takes place, thereby avoiding the generate-and-test paradigm. This raises two key questions: (i) Can we succinctly characterize the set of all itemsets that satisfy a given constraint, and when? (ii) How can we generate all and only those itemsets that satisfy the given constraint, and hence avoid the generate-and-test paradigm completely? Toward question (i), Ng et al. [1998] formalized the notion of *succinctness* below (see Definition 2.2). Toward question (ii), they proposed the notion of a *member generating function* (see Definition 2.3).

In the following, for concreteness, but without loss of generality, we assume that S is a set variable ranging over the domain of attribute `Item` (i.e., $S \subset$ `Item`). Let C be a 1-var constraint. Define $\text{SAT}_C(\texttt{Item})$ to be the set of itemsets that satisfy 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. For example, if $C \equiv S.Price \geq 50$, then the pruned space for C contains precisely those itemsets such that each item in the itemset has a price equal to 50 or above.

Definition 2.2 *Succinctness* [*Ng et al.* 1998]. Define $\text{SAT}_C(\texttt{Item})$ to be the set of itemsets that satisfy 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. The notation 2^I means the powerset of I.

(a) $I \subseteq \texttt{Item}$ 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) Finally, a 1-var constraint C is *succinct* provided that $\text{SAT}_C(\texttt{Item})$ is a succinct powerset.

For example, the constraint $max(S.Price) \geq 100$ is succinct because such an itemset must contain at least one item with $Price \geq 100$. Since there is a precise "formula" (namely, a member generating function) to generate all the itemsets satisfying such a succinct constraint, there is no need to iteratively check the constraint during the mining process.

Let us consider a more complicated example as follows. Given the constraint $C_2 \equiv S.Type \supseteq \{snack, soda\}$, the pruned space consists of all those itemsets that contain at least one item of type *snack* and at least one item of type *soda*. Let $\texttt{Item}_2$, $\texttt{Item}_3$, and $\texttt{Item}_4$ be the sets $\sigma_{Type=snack}(\texttt{Item})$, $\sigma_{Type=soda}(\texttt{Item})$, and $\sigma_{(Type \neq snack) \wedge (Type \neq soda)}(\texttt{Item})$, respectively. Then, C_2 is succinct because the pruned space $\text{SAT}_{C_2}(\texttt{Item})$ can be expressed as $2^{\texttt{Item}} - 2^{\texttt{Item}_2} - 2^{\texttt{Item}_3} - 2^{\texttt{Item}_4} - 2^{\texttt{Item}_2 \cup \texttt{Item}_4} - 2^{\texttt{Item}_3 \cup \texttt{Item}_4}$.

Definition 2.3 *Member Generating Functions* [*Ng et al.* 1998]. There are two types of member generating functions:

(a) A succinct powerset $SP \subseteq 2^{\texttt{Item}}$ is said to have a *basic member generating function* (*basic MGF*) provided there is a function that can enumerate all and only elements of SP, and that is of the form $MGF \equiv \{X_1 \cup \cdots \cup X_{k+1} \mid X_i \subseteq \sigma_{p_i}(\texttt{Item})\ \&\ X_i \neq \emptyset$, for $1 \leq i \leq k$; $X_{k+1} \subseteq \sigma_{p_{k+1}}(\texttt{Item})\}$ for some selection predicates $p_1, \ldots, p_{k+1}$. In this definition, the X_i's (for $1 \leq i \leq k$) that are required to be nonempty are called *mandatory item-subsets*; whereas X_{k+1} is called the *optional item-subset*.
(b) A succinct powerset $SP \subseteq 2^{\texttt{Item}}$ is said to have a *general member generating function* (*general MGF*) provided there is a function that can enumerate all and only elements of SP, and that is of the form $\bigcup_{j=1}^{l} MGF_j$, for some basic member generating functions $MGF_1, \ldots, MGF_l$.

Whenever no confusion arises, we do not explicitly distinguish between a basic MGF and a general MGF. Let us consider the following examples to gain a better understanding of the above definition.

—For the constraint $S.Price \geq 50$, its MGF is $\{X_1 \mid X_1 \subseteq \sigma_{Price \geq 50}(\texttt{Item})\ \&\ X_1 \neq \emptyset\}$. In this case, there is only one mandatory item-subset X_1, but there is no optional item-subset. As shown above, this corresponds to the pruned space being $2^{\texttt{Item}_1}$, where $\texttt{Item}_1 = \sigma_{Price \geq 50}(\texttt{Item})$.

—For $max(S.Price) \geq 100$, its MGF is $\{X_1 \cup X_2 \mid X_1 \subseteq \sigma_{Price \geq 100}(\texttt{Item})$ & $X_1 \neq \emptyset$ & $X_2 \subseteq \sigma_{Price<100}(\texttt{Item})\}$. This is different from the previous example (for $S.Price \geq 50$) because once there is at least one item with $Price \geq 100$, then it is acceptable to include any item with $Price < 100$. This is what the optional item-subset X_2 represents.

—For the constraint $S.Type \supseteq \{snack, soda\}$, its MGF is $\{X_1 \cup X_2 \cup X_3 \mid X_1 \subseteq \sigma_{Type=snack}(\texttt{Item})$ & $X_1 \neq \emptyset$ & $X_2 \subseteq \sigma_{Type=soda}(\texttt{Item})$ & $X_2 \neq \emptyset$ & $X_3 \subseteq \sigma_{(Type \neq snack) \wedge (Type \neq soda)}(\texttt{Item})\}$. In this case, the optional item-subset X_3 contains any item that is not of type *snack* or *soda*. These items may be included so long as an itemset contains at least one *snack* item and one *soda* item.

So far, we have seen situations in which basic MGFs are used. However, there are situations in which a general MGF is needed; the example below shows one such situation.

—For the constraint $S.Type \supseteq \{snack\} \wedge S.Type \not\supseteq \{dairy, beer\}$, its MGF is as follows:

$$\begin{aligned}\{X_1 \cup X_2 \mid X_1 &\subseteq \sigma_{Type=snack}(\texttt{Item}) \ \& \ X_1 \neq \emptyset \ \& \\ X_2 &\subseteq \sigma_{(Type \neq snack) \wedge (Type \neq dairy)}(\texttt{Item})\} \ \cup \\ \{X_1 \cup X_3 \mid X_1 &\subseteq \sigma_{Type=snack}(\texttt{Item}) \ \& \ X_1 \neq \emptyset \ \& \\ X_3 &\subseteq \sigma_{(Type \neq snack) \wedge (Type \neq beer)}(\texttt{Item})\}.\end{aligned}$$

In this case, we want to find itemsets that satisfy the following "restrictions": (i) itemsets containing at least one item of type *snack*, and (ii) itemsets not *simultaneously* containing both *dairy* and *beer* items (say, it may cause an allergic reaction if we intake them together). Notice that for this constraint, a general MGF is needed. This is not due to the conjunctive nature of the constraint, but due to its "restrictions." To elaborate, on the one hand, we should not suppress any *dairy* or *beer* item from being included; on the other hand, we should not *simultaneously* include both *dairy* and *beer* items. Here, we use a general MGF consisting of two basic MGFs. The first basic MGF enumerates the itemsets containing at least one *snack* item and some optional items that are not of type *snack* or *dairy*; the second basic MGF enumerates the itemsets containing at least one *snack* item and some optional items that are not of type *snack* or *beer*. By so doing, we do not suppress any *dairy* or *beer* item from being included because optional *beer* items can be enumerated by the first basic MGF, while optional *dairy* items can be enumerated by the second basic MGF. In addition, we also prevent both *dairy* and *beer* items from being simultaneously included because none of the basic MGFs enumerates both *dairy* and *beer* items.

While more details and discussions on constraints and MGFs can be found in Ng et al.'s paper [1998], it suffices—for the material contained in this article—to know the following two lemmas concerning succinct constraints.

LEMMA 2.4 ([NG ET AL. 1998]). *Succinct constraints have the following properties:*

(*a*) *For every succinct constraint C, there is a member generating function* MGF_C *that can generate precisely all those itemsets satisfying C.*

```
Procedure Carma-Phase I ($TDB = \langle t_1, \ldots, t_n \rangle$, support sequence $\langle \sigma_1, \ldots, \sigma_n \rangle$) {
(0)  $V = \emptyset$; /* initialization */
(1)  for $i$ from 1 to $n$ { /* start scanning $TDB$ */
(2)     for all $v \in V$ with $v \subseteq t_i$ { increment $count(v)$; }
(3)     for all $v \subseteq t_i$ with $v \notin V$ { /* insert $v$ if appropriate */
(4)        if (for all $w \subset v$: $w \in V$ and $firstTrans(w) < i$ and $maxSupport(w) \geq \sigma_i$) {
(5)           insert $v$ into $V$;
(6)           $firstTrans(v) = i$; $count(v) = 1$;
(7)           if ($v$ is a singleton itemset) $maxMissed(v) = 0$;
(8)           else $maxMissed(v) = min\{\, base,\ (maxMissed(w) + count(w) - 1) \mid w \subset v\}$;
           } /* end if */
        } /* end for-all */
(9)     /* pruning step, details omitted here: Retain $v$ if $maxSupport(v) \geq \sigma_i$; */
     } /* end for */
(10) return $V$;
}
```

Fig. 1. Phase I of Algorithm Carma.

(*b*) *For a set $\mathcal{SC}$ of succinct constraints, there is a member generating function $MGF_{\mathcal{SC}}$ that can generate precisely all those itemsets satisfying all the constraints in $\mathcal{SC}$.*

LEMMA 2.5 ([NG ET AL. 1998]). *All 1-var domain, class, and aggregate constraints involving only min() and max() are succinct. However, all 1-var constraints involving sum() and avg() are not.*

Hence, unlike antimonotone constraints, a succinct constraint can simply operate in a generate-only environment (by using an MGF), rather than in a generate-and-test environment. Moreover, it is important to note that *a majority of constraints are succinct*. (For constraints that are not succinct, many of them can be induced into weaker constraints that are succinct; see Example 4.2.)

All the results to be presented in this article can be generalized from 1-var succinct constraints to a class of 2-var constraints called *quasi-succinct* [Lakshmanan et al. 1999]. For lack of space, we do not pursue 2-var constraints further.

2.3 Overview of Carma

To allow the user to dynamically adjust the support threshold, Carma [Hidber 1999] is divided into Phase I and Phase II. During Phase I, Carma constructs a lattice V called the *support lattice*. For each itemset $v \in V$, Carma maintains three integer counters: (i) $firstTrans(v)$, storing the transaction index at which v was inserted into the lattice V; (ii) $count(v)$, storing the support of v *since* v was inserted; and (iii) $maxMissed(v)$, storing an upper bound on the support of v *before* v was inserted.

Suppose the ith transaction t_i has just been read. Then, $maxSupport(v)$, defined to be $(maxMissed(v) + count(v))/i$, gives an upper bound on the support of v in the first i transactions. At this point after reading t_i, two main operations (i.e., increment and insert) can take place. On the one hand, as shown in Step (2) of Figure 1, Carma increments $count(v)$ for each itemset v that is currently in

V and is a subset of transaction t_i. On the other hand, as shown in Step (3), if v—that is a subset of t_i—is not currently in the lattice V, then v is inserted provided that the following *Insertion Condition* (in Step (4)) is met:

$$\text{for all } w \subset v\colon w \in V \quad \text{and} \quad \mathit{firstTrans}(w) < i \quad \text{and} \quad \mathit{maxSupport}(w) \geq \sigma_i$$

where σ_i is the support threshold at the point after t_i has just been read. If the above Insertion Condition is met and v is inserted, then the three counters associated with v are initialized in Steps (6), (7), and (8). Of the three counters, the initialization of *maxMissed*(v) requires the most attention. If v is a singleton itemset, the above Insertion Condition guarantees that this transaction is the first transaction containing v, in which case *maxMissed*(v) should be initialized to 0. If v is not a singleton itemset, then *maxMissed*(v) is initialized based on the counters associated with all subsets w of v. In addition, such an initialization is also based on a quantity called *base*, which is an estimate defined according to the support threshold sequence: $\mathit{base} = (i-1) \times g(\langle \sigma_1, \ldots, \sigma_n \rangle) + |v| - 1$. Here, we omit the precise definition of the function $g()$, because it is immaterial to the rest of the article.

THEOREM 2.6 ([HIDBER 1999]). *Let $\langle \sigma_1, \ldots, \sigma_n \rangle$ be the sequence such that σ_i is the support threshold after transaction t_i is read. The support lattice V produced at the end of Phase I of Carma is typically a superset of all frequent itemsets relative to the support threshold given by $g(\langle \sigma_1, \ldots, \sigma_n \rangle)$.*

The above theorem establishes some notion of correctness for Carma: The support lattice V produced at the end of Phase I of Carma typically contains all the frequent itemsets satisfying the specific threshold condition given by $g(\langle \sigma_1, \ldots, \sigma_n \rangle)$. Here, we note that $g(\langle \sigma_1, \ldots, \sigma_n \rangle) \geq \sigma_n$ in general. To deal with the situation where $g(\langle \sigma_1, \ldots, \sigma_n \rangle) > \sigma_n$, a heuristic was proposed to make $g(\langle \sigma_1, \ldots, \sigma_n \rangle) \approx \sigma_n$ (or to make it equal to σ_n), and was shown to behave very well experimentally. Refer to Hidber [1999] for more details.

From time to time, Carma may invoke the pruning step (i.e., Step (9)) to prune the lattice V during Phase I; details of the pruning step are omitted. Finally, to complete the description of Carma, Phase II rescans the transaction database *TDB* to get precise counts for all itemsets $v \in V$. Those itemsets whose supports are below the last support threshold σ_n are pruned. The support threshold is allowed to vary only during Phase I, but not during Phase II.

3. SEGMENT SUPPORT MAP

In the previous section, we provided related work and background material. In this section, we start presenting our development of an algorithmic framework for the efficient dynamic mining of constrained frequent sets. Our focus of this section is on how we make our algorithmic framework *efficient*. More specifically, to provide a human-centered and exploratory environment for data mining, it is imperative that the system be efficient, and be able to deliver responses to the user in as "real time" as possible, so as not to lose the attention of the user. Towards this objective, we introduce in this section a structure called a *segment support map* (*SSM*). We first give an overview of the SSM. We then introduce the DCF algorithm (as an extension of Carma), and show how DCF can use the SSM to effect more pruning than Carma.

3.1 Motivation and Overview

Recall from Section 2.3 and from Steps (4) and (9) of Figure 1 that the only source of pruning in Carma is based on the condition $maxSupport(w) < \sigma_i$. The term $maxSupport(v)$ is defined as $(maxMissed(v) + count(v))/i$. In turn, $maxMissed(v)$ is mainly initialized as in Step (8):

$$maxMissed(v) = min\{base,\ (maxMissed(w) + count(w) - 1) \mid w \subset v\}. \qquad (1)$$

However, there are two main weaknesses with this pruning strategy of Carma. The first problem is that the right-hand-side of the above equation is too loose an upper bound. In addition, because of the recursive nature of the equation, a loose initialization of $maxMissed(v)$ has a compound effect that causes the bound for $maxMissed(u)$, for all supersets u of v, to be quite loose as well.

The second problem is with the division by i in the term $(maxMissed(v) + count(v))/i$. Basically, this is a uniform distribution assumption—assuming that the transactions supporting v are uniformly distributed, that is, whatever happens in the first i transactions will continue to hold for the remaining transactions. In practice, this assumption is hardly true. For example, if the transaction database consists of supermarket transactions over a few months, items sold during the summer can be very different from those sold in the fall. Thus, pruning based on this assumption can be highly inaccurate. One consequence is that an itemset might be pruned too early, and needs to be reinserted afterwards. Another consequence is that an itemset might be kept for too long, while it should have been pruned earlier.

The SSM can record whatever variations that may exist in the support of an itemset from different parts of the transaction database. In this section, we show how it can be used to tighten the $maxMissed(v)$ bound, as well as to effect pruning while discarding the uniform distribution assumption.

Let the transaction database *TDB* be arbitrarily divided into m nonoverlapping partitions, called segments. A *segment support map* is a structure consisting of $support_i(\{a\})$ for all *singleton* itemsets $\{a\}$, where $support_i(\{a\})$ denotes the support of $\{a\}$ in the ith segment for $1 \leq i \leq m$. The support of $\{a\}$, by definition, is then $\sum_{i=1}^{m} support_i(\{a\})$. While the SSM only contains the segment supports of singleton itemsets, it can be used to give an upper bound on the support of an arbitrary itemset v:

$$estsup(v) = \sum_{i=1}^{m} min\{support_i(\{a\}) \mid a \in v\}, \qquad (2)$$

where $estsup(v)$ denotes the estimated segment support of v.

Example 3.1. Suppose there are four segments in an SSM, as shown in Figure 2. The SSM stores the actual segment supports of items a, b, c for each of the four segments. By Eq. (2), $estsup(\{a, b\})$ is $min\{20, 5\} + min\{10, 20\} + min\{5, 20\} + min\{20, 20\}$, for a total of 40. Similarly, by Eq. (2), the support of itemset $\{a, b, c\}$ is bounded from above by 30. On the other hand, if we did not

SSM:

	seg 1	seg 2	seg 3	seg 4	*TDB*
$\{a\}$	20	10	5	20	55
$\{b\}$	5	20	20	20	65
$\{c\}$	10	20	10	10	50

Fig. 2. Example of an SSM.

use the SSM (i.e., the number of segments is one), then the estimated support for $\{a, b\}$ would have been $min\{55, 65\} = 55$, while that for $\{a, b, c\}$ would have been $min\{55, 65, 50\} = 50$.

The above example shows how the SSM can provide valuable filtering by reducing the number of candidate itemsets that need to be counted for their supports (e.g., when the support threshold is less than 50). For many frequent-set mining algorithms, one of the performance bottlenecks is the number of candidates that have to be considered; even in cases where the true number of frequent itemsets is small, there can be a huge number of candidate itemsets. Another important thing to note is that for many algorithms (e.g., those based on hashing), skewed data is problematic. In contrast, the more skewed the data, the more effective the SSM is.

Clearly, the upper bound *estsup*(v) provided by the SSM can be made tighter in two ways. The first way is to increase the number of segments m. The amount of storage space required is then increased linearly. The second way to generalize the SSM is to store not only the actual segment supports of singleton itemsets, but also the actual segment supports of the itemsets of sizes at least two. For example, for the support of itemset $\{a, b, c\}$, the actual segment supports based on $\{a, b\}$, $\{a, c\}$, and $\{b, c\}$ provide a tighter upper bound than those based on $\{a\}$, $\{b\}$, and $\{c\}$. The price is that the amount of storage space required is then increased exponentially with respect to the sizes of the stored itemsets. Thus, in most parts of this article, we restrict our consideration to segment supports of singleton itemsets. (We only discuss segment supports of 2-itemsets in Section 8.5.) In this way, we keep the SSM as a very lightweight structure. For instance, for a domain of 10,000 items, even 50 segments require the storage of only 500,000 integers. We also note that the SSM is a fixed structure that can be computed *once* at "compile-time," and can be used regardless of how the support threshold is changed dynamically during "exploration-time." Finally, there is no searching involved when the SSM is used. Once the singleton itemsets are enumerated based on some canonical ordering, the itemsets themselves need not be stored (i.e., the first column in Figure 2), and direct addressing into the SSM makes the computation of Eq. (2) very efficient.

3.2 Tightening the *maxMissed*(*v*) Bound

So far, we have given a general description of the SSM. Next, we introduce the DCF algorithm. While DCF have other important features, which we will present in later sections, we focus narrowly here on how the SSM is used in DCF.

Figure 3 depicts the various events during a transaction scan. Here, we assume that the kth segment contains h transactions. Suppose a certain itemset

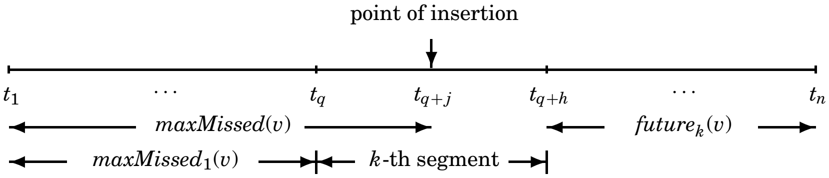


Fig. 3. Using the SSM for *maxMissed*(v) and *future*$_k(v)$.

v is inserted after the jth transaction of this segment (i.e., transaction t_{q+j} where $j < h$) has been read. Exactly as in Carma, our *DCF algorithm* initializes *maxMissed*(v) as per Eq. (1). However, in the presence of the SSM, the DCF algorithm can divide *maxMissed*(v) into two components: (i) the part from the first $k - 1$ segments, which we refer to as *maxMissed*$_1(v)$; and (ii) the part from within the first j transactions in the kth segment, which we refer to as *maxMissed*$_2(v)$.

Let us examine *estsup*(v) in Eq. (2) in the finer granularity of individual segments. Specifically, let *estsup*$_k(v)$ denote the estimated segment support of v (via the SSM) from the kth segment, that is,

$$estsup_k(v) = min\{support_k(\{a\}) \mid a \in v\}. \tag{3}$$

Then, the first component *maxMissed*$_1(v)$ is simply

$$maxMissed_1(v) = \sum_{u=1}^{k-1} estsup_u(v), \tag{4}$$

which is the sum of the estimated segment supports of v from the first $k - 1$ segments.

The second component *maxMissed*$_2(v)$ is harder to estimate, because this corresponds to only part of a segment. Clearly, *maxMissed*$_2(v)$ is bounded from above by j, the number of transactions in the kth segment read so far. Depending on how far the current segment has been read, such a bound may be too loose. In this case, the estimated segment support of v, *just* from the kth segment (i.e., *estsup*$_k(v)$) may give a tighter bound. Thus, the second component of *maxMissed*(v) can be computed as follows:

$$maxMissed_2(v) = min\{j, estsup_k(v)\}. \tag{5}$$

Hence, the DCF algorithm can combine all of Eqs. (1), (3), (4), and (5) to give a better bound for *maxMissed*(v). Note that this is a bound obtained *at the point of insertion* of v (i.e., after the transaction t_{q+j} has been read).

However, at a later point in time, when the *entire kth segment has been read*, a better bound may be possible. This is because in Eq. (5) above, *estsup*$_k(v)$ is used to estimate the support of v from the first j transactions in the kth segment, when in fact the same bound applies to the *entire* segment, implying that we should be able to do better. Let *count*$_k(v)$ denote the *actual* support of v in the kth segment *after v was inserted*. Then, by the definition of the SSM, it must be the case that

$$maxMissed_2(v) + count_k(v) \leq estsup_k(v).$$

Hence, while Eq. (5) is appropriate for initializing $maxMissed_2(v)$ *at* the point of insertion, the following equation can be used to *update*—and possibly *tighten*—$maxMissed_2(v)$ *after* the entire kth segment has been processed:

$$maxMissed_2(v) = min\{j, (estsup_k(v) - count_k(v))\}. \tag{6}$$

The value of $maxMissed(v)$ can be tightened accordingly.

3.3 Creating the *future*$_k$(*v*) Bound

Recall from Section 3.1 that there are two weaknesses associated with the pruning based on $maxSupport(v) = (maxMissed(v) + count(v))/i$ in Carma. So far, our DCF algorithm has addressed the first problem by tightening the $maxMissed(v)$ bound. Next, we turn our attention to the second problem of making the (strong) assumption of uniform distribution.

Suppose the transaction scan is at the point when the kth segment has just been processed. Given an itemset v that has been inserted and is being processed, let $future_k(v)$ denote an upper bound on the support of v from all the future/remaining segments, that is, $(k+1)$th, ..., mth segments. With the SSM, this is defined as follows:

$$future_k(v) = \sum_{u=k+1}^{m} estsup_u(v). \tag{7}$$

By putting all the pieces together, the old pruning condition (which is based on $maxSupport(v) = (maxMissed(v) + count(v))/i$) can now be replaced by a new condition (which is based on the value of $(maxMissed(v) + count(v) + future_k(v))/n$, where n is the total number of transactions) at the point when the kth segment has just been processed. The new condition can be used in Step (9) of Figure 1. Note that both $maxMissed(v)$ and $future_k(v)$ are upper bounds, and $count(v)$ is the actual count up to the point of pruning. Thus, unlike the old condition, the new pruning condition guarantees that a pruned itemset will never need to be reinserted, unless the support threshold is reduced.

In sum, our DCF algorithm overcomes Carma's problems of (i) a very loose $maxMissed(v)$ bound and (ii) a strong assumption of uniform distribution, respectively, by:

- tightening the $maxMissed(v)$ bound, and
- creating the $future_k(v)$ bound to effect pruning and to discard the uniform distribution assumption.

It is important to note that the SSM provides direct information about variability of support counts in different segments of the transaction database. As a result, such a generic structure—the SSM—can bring benefits not only to constrained online/dynamic mining algorithms like DCF, but also to a general class of offline/nondynamic mining algorithms (constrained or otherwise). To avoid distraction from our focus on the development of an algorithmic framework for the efficient mining of constrained frequent sets, we defer discussions on other useful applications of the SSM to Section 7.

4. HANDLING NONFREQUENCY CONSTRAINTS

In the previous section, we considered the performance aspect of DCF. Specifically, we focused on how the SSM enhances the performance of DCF. In this section, we turn our attention to the functionality aspect of DCF. More specifically, we show *how DCF can handle nonfrequency constraints*.

4.1 Succinctness-Based Optimization

Given a CFQ $\{(S, T) \mid \mathcal{C}\}$, $\mathcal{C}$ may contain frequency as well as nonfrequency constraints. Throughout this section, for simplicity, we use $\mathcal{C}$ to denote just the set of nonfrequency constraints. A naïve way to handle a CFQ is to first run Carma as shown in Figure 1, and then to check every $v \in V$ whether v satisfies $\mathcal{C}$ (denoted as $v \models \mathcal{C}$). We call this naïve strategy *Carma+*. While simple, Carma+ suffers from the fact that constraints $\mathcal{C}$ are not pushed inside Carma to effect pruning as early as possible. As a result, the computational effort is *not* proportional to the selectivity of constraints $\mathcal{C}$. To overcome this problem, DCF—which follows the skeleton of Carma—pushes the constraints $\mathcal{C}$ deep "inside" the computation by modifying the Insertion Step of Carma (i.e., Step(3) in Figure 1):

(3) for all $v \subseteq t_i$ with $v \notin V$ { /* insert v if appropriate */ }

to become the Insertion Step of DCF, as follows:

(3a) for all $v \subseteq t_i$ with ($v \notin V$ and $v \models \mathcal{C}$) { /* insert v if appropriate */ }

Recall that a constraint $C \in \mathcal{C}$ can be succinct, and/or antimonotone, or neither. The modified Step (3a) above works for *any* of these classes of constraints, but DCF can do a lot better with succinct constraints. Recall from Lemma 2.4 that every succinct constraint C corresponds to a member generating function MGF_C that can precisely enumerate all itemsets satisfying C. Thus, while Step (3a) above adopts a generate-and-test strategy, for a succinct constraint C, we can use MGF_C directly to avoid any testing whatsoever. Let us consider the example below.

Example 4.1. Suppose $C \equiv max(S.Price) \geq 10$, and there are 12 items in a transaction t_i (say, $t_i = \{a, b, \ldots, k, l\}$). In a "blind" generate-and-test, there are $\binom{12}{4} = 495$ subsets of t_i of size 4. Suppose only items a, b, and c have $Price \geq 10$, while the remaining items have $Price < 10$. Then, $MGF_C(t_i)$ corresponds to

$$\{S_1 \cup S_2 \mid S_1 \subseteq \{a, b, c\} \ \&\ S_1 \neq \emptyset \ \&\ S_2 \subseteq \{d, \ldots, l\}\}.$$

That is, every subset of t_i satisfying $max(S.Price) \geq 10$ must be amongst those generated above. We can restrict $MGF_C(t_i)$ to generate only subsets of size 4. In this case, there are $\binom{3}{1} \times \binom{9}{3} + \binom{3}{2} \times \binom{9}{2} + \binom{3}{3} \times \binom{9}{1}$ subsets of size 4 *satisfying* C, for a total of 369 subsets. So, for subsets of size 4 alone, making use of succinctness generates $495 - 369 = 126$ fewer subsets, *and no testing is required for all the 369 subsets*. This gives considerable savings.

So far, we have focused on one succinct constraint. By Lemma 2.4, we know that given a set of succinct constraints $\mathcal{SC}$, there is a member generating function $MGF_{\mathcal{SC}}(t_i)$ that can be exploited in the manner shown above. Thus, if $\mathcal{SC}$ contains all the succinct constraints in $\mathcal{C}$ in the CFQ, our DCF algorithm modifies Step (3a) to become the following:

```
(3b) for each $v$ generated by $MGF_{\mathcal{SC}}(t_i)$ where ($v \notin V$ and $v \models (\mathcal{C} - \mathcal{SC})$)
         { /* insert $v$ if appropriate */ }
```

The set $(\mathcal{C} - \mathcal{SC})$ consists of the nonsuccinct constraints, if any. These constraints are not covered by $MGF_{\mathcal{SC}}(t_i)$ and need to be verified explicitly.

Furthermore, the optimization based on $MGF_{\mathcal{SC}}(t_i)$ is not confined to just the succinct 1-var constraints in $\mathcal{C}$. Its application can be broadened. For a nonsuccinct 1-var constraint C, it may induce a weaker constraint that is succinct, as shown in the following example.

Example 4.2. Suppose $\mathcal{C}$ consists of the single constraint $C_1 \equiv avg(S.Price) \geq 10$. C_1 induces the weaker constraint $C_2 \equiv max(S.Price) \geq 10$, in the sense that every itemset satisfying C_1 must also satisfy C_2. Then, for this example, Step (3b) instantiates to the following:

```
for each $v$ generated by $MGF_{C_2}(t_i)$ where ($v \notin V$ and $v \models C_1$)
    { /* insert $v$ if appropriate */ }
```

4.2 Antimonotonicity-Based Optimization with the SSM

So far, we have discussed how DCF exploits succinct constraints. Next, we turn our attention to the other important class of constraints, namely the antimonotone ones. A constraint C is antimonotone if for any itemset S, $S \not\models C$ implies $S' \not\models C$ for any superset S' of S. The frequency constraint $C_1 \equiv support(S) \geq minsup$ is one example, and $C_2 \equiv sum(S.Price) \leq 10$ is another (assuming that the price of any item is nonnegative). Without the SSM, Step (3b) is all DCF can do.

However, with the SSM, the antimonotonic nature of $C_1 \equiv support(S) \geq minsup$ can be exploited as follows. After transaction t_i has just been read, DCF only needs to consider as candidates those itemsets that do not include any *confirmed infrequent* single items. For example, suppose $t_i = \{a, b, \ldots, k, l\}$, and knowing from the SSM that only three of the 12 items (namely, items a, b, and c) have a support below σ_i. Then, for all subsequent computations regarding t_i, we only need to consider $t_i - \{a, b, c\}$ because no itemset containing a, b, or c can be frequent. Specifically, for subsets of size j of t_i, we need to process only $\binom{9}{j}$ itemsets, as opposed to $\binom{12}{j}$ itemsets. Accumulating for all $j \geq 2$, this optimization step can bring about considerable savings.

The SSM we have seen so far can be easily extended to help exploit antimonotone constraints other than the frequency constraint. Using $C_2 \equiv sum(S.Price) \leq 10$ as an example, all DCF needs to do is to add one column to the SSM, recording the *Price* value of each singleton itemset. For transaction t_i, suppose only items a, d, and e have $Price > 10$. Then, it is sufficient to consider only subsets of $t_i - \{a, d, e\}$ because no itemset containing a, d, or e can satisfy C_2. Moreover, for $C_1 \equiv support(S) \geq minsup$ and $C_2 \equiv sum(S.Price) \leq 10$ together, only subsets of $t_i - \{a, b, c, d, e\}$ deserve consideration. Thus, in general, DCF can further enhance Step (3b) to become the following:

```
(3c) for each $v$ generated by $MGF_{\mathcal{SC}}(t_i^{SSM})$ where ($v \notin V$ and $v \models (\mathcal{C} - \mathcal{SC})$)
         {/* insert $v$ if appropriate */ }
```

where t_i^{SSM} is the subset of t_i that remains after the above antimonotonicity-based optimization with the SSM has been applied.

5. HANDLING DYNAMIC CHANGES TO NONFREQUENCY CONSTRAINTS

In Section 3, we discussed how the SSM enhances the performance of DCF; in Section 4, we discussed how DCF handles nonfrequency constraints. Hence, we have developed an algorithmic framework for the efficient mining of constrained frequent sets. In this section, we show how DCF can handle *dynamic changes* to constraints (i.e., leading to the efficient *dynamic* mining of constrained frequent sets). Because DCF follows the skeleton of Carma, DCF handles dynamic changes to the support threshold *minsup* in exactly the same fashion as Carma does. Thus, below, we focus on dynamic changes to nonfrequency constraints. There are two cases: a tightening change and a relaxing change. It is important to note that handling dynamic changes to nonfrequency constraints is *not* a straightforward variation of handling dynamic changes to *minsup* (the frequency constraint), because nonfrequency constraints are not necessarily numerical. Constraints $S.Type = meat$ and $S.Type \supseteq \{snack, soda\}$ are some examples. Hence, handling nonfrequency constraints takes on a flavor different from handling *minsup*. Handling dynamic changes to nonfrequency constraints is totally different from handling dynamic changes to *minsup*.

5.1 Tightening a Constraint Dynamically

Our discussion here assumes that at any point in time, there is at most one constraint that is being modified. (During the entire process, many different constraints can, of course, be changed.) We begin with a 1-var constraint $C(S, \theta, agg, cn)$ with a set variable S, a relational operator θ, an (optional) aggregate or set operator *agg*, and a constant *cn*. The base case of our discussion below is on how to deal with changes to the constant *cn*. When *cn* is modified by the user in the direction of restricting the new solution space to be a *subset* of the old space, we call this a *tightening change*. Otherwise, whenever the change to *cn* corresponds to the situation where the new solution space contains the old space, we call this a *relaxing change*. For example, if the original constraint is $max(S.Price) \geq 10$, then changing from 10 to 12 and to 8 corresponds to a tightening change and a relaxing change, respectively. Similar remarks apply to the whole family of constraints introduced in Table II (and in Ng et al.'s work [1998]). Clearly, inserting a new constraint is a special case of a tightening change, and deleting an old constraint is an extreme case of a relaxing change. Thus, while our discussion below is confined to changing the constant *cn*, any other modification to $C(S, \theta, agg, cn)$ such as modifying $max(S.Price) \geq 10$ to $max(S.Price) \leq 10$ (or modifying $S.Type = meat$ to $S.Type \supseteq \{snack, soda\}$) can be dealt with as a pair of constraint deletion and insertion.

By definition, a tightening change from C_{old} to C_{new} corresponds to a restriction of the old solution space. To accommodate C_{new} dynamically, DCF takes two steps:

—Replace C_{old} with C_{new} for future insertions to lattice V. That is to say, Step (3c) mentioned in the previous section now becomes the following:

```
(3d) for each $v$ generated by $MGF_{\mathcal{SC}_{new}}(t_i^{SSM})$
         where ($v \notin V$ and $v \models (\mathcal{C}_{new} - \mathcal{SC}_{new})$) { /* insert $v$ if appropriate */ }
```

—The step above deals with future operations on V; what remains to be done is to "fix" up the current V. All DCF needs to do is to check for each current itemset $v \in V$ whether it still satisfies C_{new} (i.e., check if $v \models C_{new}$).

5.2 Relaxing a Constraint Dynamically: A Generate-and-Test Approach

In general, a relaxing change has different, and tougher, computational requirements than a tightening change. This is because for a tightening change, the new solution space is contained in the old space, and all that is needed is to verify whether $v \models C_{new}$. In contrast, for a relaxing change, this verification is unnecessary. What is needed, however, is to insert into V all the itemsets that were not generated before the constraint was relaxed. The question here is whether DCF needs to do this at once, or whether—following the skeleton in Figure 1—these itemsets will be inserted into V in due course.

The following lemma states that if there does not exist a future transaction t_j containing itemset v, then v will not be added into the lattice V. This problem is particularly serious when the change from C_{old} to C_{new} appears towards the end of the transaction sequence. Note that even if there is no future transaction after t_i supporting v, there could *possibly* be sufficiently many transactions supporting v prior to t_i so as to make v frequent. Hence, to ensure correctness, it is imperative that *right after the relaxing change*, DCF must insert into the lattice V all those v's satisfying C_{new} but not C_{old}.

LEMMA 5.1. *Let TDB* $= \langle t_1, \ldots, t_n \rangle$ *be the transaction sequence, and let a constraint* C_{old} *be relaxed to* C_{new} *after transaction* t_i (*where* $1 \leq i \leq n$) *has been read. If (i) any itemset v satisfies the relaxed constraint* C_{new} *but does not satisfy the original constraint* C_{old}*, and (ii) there does not exist another future transaction* $t_j \supseteq v$ *(where* $i + 1 \leq j \leq n$*), then v cannot be in the lattice V unless we insert it into V right after the relaxing change.*

PROOF. According to the modifications to the Insertion Step (i.e., Step (3)) described so far, if v was added to V between transaction t_1 to t_i, then v must satisfy the constraint C_{old}, which is not the case here. Thus, in order for v to be in V eventually, v must be added after transaction t_{i+1}. However, in accordance with Figure 1, this is possible only if $v \subseteq t_j$ for some $i + 1 \leq j \leq n$, which is not the case here. Thus, v cannot be in V (unless we insert it right after the relaxing change). □

Therefore, to ensure correctness, it is imperative that *right after the relaxing change*, DCF must insert into lattice V all those v's satisfying C_{new} but not C_{old}. Hence, the next issue is how to do this efficiently.

The simplest approach no doubt is to do a blind generate-and-test. That is, for every previously read transaction t_k (where $1 \leq k \leq i$), check for each itemset $v \subseteq t_k^{SSM}$ to see if v satisfies $\mathcal{C}_{new} \equiv (\mathcal{C}_{old} - \{C_{old}\} \cup \{C_{new}\})$, and insert v into V if $v \notin V$ currently. This approach can be optimized in three ways, as follows:

—Recall that DCF (and Carma) inserts v into V only if all $w \subset v$ are already in V. Thus, we can reduce the number of v's that are generated-and-tested by not considering all v's (in particular, we do not need to consider those v's with large cardinality). Let *maxCard* be the cardinality of the largest v currently in V. Then, DCF can restrict the generate-and-test process to each $v \subseteq t_k^{SSM}$ (where $1 \leq k \leq i$) with $|v| \leq maxCard$. This is particularly effective if the relaxing change occurs early in the transaction scan (i.e., i is closer to 1 than to n).

—The generate-and-test process can be further optimized by using the succinct constraints $\mathcal{SC}_{new}$ in $\mathcal{C}_{new}$, if any. Based on the argument in Section 4.1, the member generating function $MGF_{\mathcal{SC}_{new}}(t_k^{SSM})$ can be used to reduce the effort for generate-and-test.

—In practice, DCF avoids using $MGF_{\mathcal{SC}_{new}}(t_k^{SSM})$ for each transaction t_k (for $1 \leq k \leq i$), which would require a rescanning of the transaction database. Instead, it uses $MGF_{\mathcal{SC}_{new}}(\cup_{k=1}^{i} t_k^{SSM})$.

Putting these three optimizations together, DCF modifies the Insertion Step to become the following:

```
(3e) for each $v$ generated by $MGF_{\mathcal{SC}_{new}}(\cup_{k=1}^{i} t_k^{SSM})$
     where ($v \notin V$ and $v \models (\mathcal{C}_{new} - \mathcal{SC}_{new})$ and $|v| \leq maxCard$)
         { /* insert $v$ if appropriate */ }
```

5.3 Relaxing a Constraint Dynamically: An Optimized Approach with a Delta Member Generating Function

As it turns out, a relaxing change can be further optimized when the constraint being relaxed is succinct. (Recall from Section 2.2 that a majority of the constraints are succinct.) The trick is to use a *delta member generating function (delta MGF)*, which generates *only* the new solutions. Note that the situation here is more complex than what is guaranteed in Lemma 2.4. The reason is that the MGF that we have discussed so far only deals with the solution space of (conjunctions of) constraints; however, here we are dealing with the *set difference* between the solution space of C_{new} and that of C_{old}. The complication is that it is very inefficient to obtain results by first generating all itemsets satisfying C_{new} and then excluding those satisfying C_{old}, because such a generate-and-test approach would waste a lot of (unnecessary) computation. What we want is a generate-only approach. In other words, we want to use a delta MGF to generate precisely those itemsets in the set difference between the two solution spaces. In the following, we show that succinct constraints enjoy the additional desirable property that there is a delta MGF that exactly enumerates those itemsets that satisfy the relaxed constraint C_{new} but not the original constraint C_{old}.

Throughout this article, we use the notation $MGF_{C_{new}/C_{old}}$ to denote the delta MGF that generates precisely those itemsets that are solutions to C_{new} but not to C_{old}. Below, we give an example to illustrate the idea.

Example 5.2. Let us return to Example 4.1. Let $C_{old} \equiv max(S.Price) \geq 10$ be relaxed to $C_{new} \equiv max(S.Price) \geq 6$. Suppose for the items contained in the transaction $t_i = \{a, b, \ldots, k, l\}$, (i) items a, b, c, d, e have their *Price* values at 6 or above, and (ii) only items a, b, c have their *Price* values at 10 or above. Then, the MGF for C_{old} *alone* is

$$\{X_1 \cup X_2 \mid X_1 \subseteq \{a, b, c\} \ \&\ X_1 \neq \emptyset \ \&\ X_2 \subseteq \{d, \ldots, l\}\}.$$

Similarly, the MGF for C_{new} *alone* is

$$\{X_1 \cup X_2 \mid X_1 \subseteq \{a, b, c, d, e\} \ \&\ X_1 \neq \emptyset \ \&\ X_2 \subseteq \{f, \ldots, l\}\}.$$

Capturing the "delta" between the MGFs for C_{old} and C_{new} is precisely the delta member generating function $MGF_{C_{new}/C_{old}}(t_i)$, which is the following:

$$\{X_1 \cup X_2 \ \mid\ X_1 \subseteq \{d, e\} \ \&\ X_1 \neq \emptyset \ \&\ X_2 \subseteq \{f, \ldots, l\}\}.$$

In the remainder of this section, we prove that a relaxing change to a succinct constraint C_{old} admits a delta MGF. We give our proof in two stages. In the first stage, we refine Lemma 2.4(a) to give the precise form of an MGF of a constraint, depending on the nature of the constraint. The following lemma is for that purpose.

LEMMA 5.3. *Let C be a constraint equivalent to one of the succinct constraints listed in Table* II. *Then, one of the following holds.*

(*a*) *Suppose C is of a form equivalent to $S.A \supseteq CS$, where A is an attribute of a set S, and $CS = \{cn_1, \ldots, cn_k\}$ is a set of constants for attribute A. Then, the corresponding MGF is of the form $\{X_1 \cup \cdots \cup X_{k+1} \mid X_i \subseteq \sigma_{p_i}(\texttt{Item}) \ \&\ X_i \neq \emptyset$, for $1 \leq i \leq k$; $X_{k+1} \subseteq \sigma_{p_{k+1}}(\texttt{Item})\}$, where the selection predicate p_i is of the form $(A = cn_i)$, for $1 \leq i \leq k$, and p_{k+1} is of the form $((A \neq cn_1) \wedge \cdots \wedge (A \neq cn_k))$.*

(*b*) *Suppose C is of a form not equivalent to that in part (a), and C is not antimonotone. Then, the corresponding MGF is of the form $\{X_1 \cup X_2 \mid X_1 \subseteq \sigma_{p_1}(\texttt{Item}) \ \&\ X_1 \neq \emptyset \ \&\ X_2 \subseteq \sigma_{p_2}(\texttt{Item})\}$, for some selection predicates p_1 and p_2.*

(*c*) *Suppose C is of a form not equivalent to that in part (a), but C is also antimonotone. Then, the corresponding MGF is of the form $\{X_1 \mid X_1 \subseteq \sigma_{p_1}(\texttt{Item}) \ \&\ X_1 \neq \emptyset\}$, for some selection predicate p_1.*

PROOF. This proof is divided into the following three cases, depending on the nature of the constraint.

(a) Let C be a succinct constraint of a form equivalent to $S.A \supseteq CS$. It is easy to verify that any itemset satisfying $C \equiv S.A \supseteq CS$ must contain at least k items (where k is the cardinality of CS). In particular, for $CS = \{cn_1, \ldots, cn_k\}$, the itemset satisfying C requires at least one item chosen from each of the selections $\sigma_{A=cn_i}(\texttt{Item})$ for $1 \le i \le k$. Hence, for $S.A \supseteq \{cn_1, \ldots, cn_k\}$, the MGF is of the form $\{X_1 \cup \cdots \cup X_k \cup X_{k+1} \mid X_i \subseteq \sigma_{A=cn_i}(\texttt{Item})$ & $X_i \neq \emptyset$, for $1 \le i \le k; X_{k+1} \subseteq \sigma_{(A \neq cn_1) \wedge \cdots \wedge (A \neq cn_k)}(\texttt{Item})\}$.

(b) Let C be a succinct constraint of a form not equivalent to that in part (a), and C is not antimonotone. From Table II, it is easy to observe that C is equivalent to one of the following: $cn \in S.A$, $min(S.A) \le cn$, or $max(S.A) \ge cn$, as summarized in the table below.

1-var Constraint	p_1	p_2
$cn \in S.A$	$A = cn$	$A \neq cn$
$min(S.A) \le cn$	$A \le cn$	$A > cn$
$max(S.A) \ge cn$	$A \ge cn$	$A < cn$

The table above specifies the exact definitions of p_1 and p_2, depending on the exact form of C. Such a constraint C can be effectively reduced to the constraint $S.Type \supseteq \{1\}$, where

$$\begin{cases} Type = 1 & \text{if } p_1 \\ Type = 0 & \text{if } p_2. \end{cases}$$

Here, $\sigma_{p_1}(\texttt{Item})$ selects the mandatory items, and $\sigma_{p_2}(\texttt{Item})$ selects the optional items. Hence, the corresponding MGF is of the form $\{X_1 \cup X_2 \mid X_1 \subseteq \sigma_{p_1}(\texttt{Item})$ & $X_1 \neq \emptyset$ & $X_2 \subseteq \sigma_{p_2}(\texttt{Item})\}$.

(c) Let C be a succinct constraint of a form not equivalent to that in part (a), but C is also antimonotone. From Table II, it is easy to observe that C is equivalent to one of the forms summarized in the table below.

1-var Constraint	p_1
$S.A\ \theta\ cn$ (where $\theta \in \{=, \le, \ge\}$)	$A\ \theta\ cn$
$S.A \subseteq \{cn_1, \ldots, cn_k\}$	$(A = cn_1) \vee \cdots \vee (A = cn_k)$
$min(S.A) \ge cn$	$A \ge cn$
$max(S.A) \le cn$	$A \le cn$

Similar to the proof for part (b), the constraint C can be effectively reduced to the constraint $S.Type \supseteq \{1\}$, where

$$\begin{cases} Type = 1 & \text{if } p_1 \\ Type = 0 & \text{if } p_2 \text{ (where } p_2 = \neg p_1). \end{cases}$$

Here, $\sigma_{p_1}(\texttt{Item})$ selects the mandatory items and $\sigma_{p_2}(\texttt{Item})$ selects the optional items. Accordingly, the corresponding MGF is of the form $\{X_1 \cup X_2 \mid X_1 \subseteq \sigma_{p_1}(\texttt{Item})$ & $X_1 \neq \emptyset$ & $X_2 \subseteq \sigma_{p_2}(\texttt{Item})\}$. However, the key difference between this part and part (b) is that X_2 *must be empty* (i.e., there is no optional items). Let us show this by contradiction. Suppose X_2 is not empty. Then, for any $a \in \sigma_{p_2}(\texttt{Item})$, it is easy to verify that $X_1 \cup \{a\}$ does not satisfy C. Due to the antimonotonicity of C, any superset of $X_1 \cup \{a\}$ also does not satisfy C. In particular, for all $X_2 \subseteq \sigma_{p_2}(\texttt{Item})$, $X_1 \cup X_2$ does not satisfy C, unless X_2 is an empty set. Hence, the corresponding MGF is of the form $\{X_1 \mid X_1 \subseteq \sigma_{p_1}(\texttt{Item})$ & $X_1 \neq \emptyset\}$. □

We illustrate the use of the above lemma with the following example.

Example 5.4. Let us consider the three succinct constraints below. Depending on the nature of each constraint, the precise form of the corresponding MGF is different.

- For $C_1 \equiv S.Type \supseteq \{snack, soda\}$, C_1 is succinct and of a form equivalent to $S.A \supseteq CS$. Hence, its MGF is $\{X_1 \cup X_2 \cup X_3 \mid X_1 \subseteq \sigma_{Type=snack}(\texttt{Item})\ \&\ X_1 \neq \emptyset\ \&\ X_2 \subseteq \sigma_{Type=soda}(\texttt{Item})\ \&\ X_2 \neq \emptyset\ \&\ X_3 \subseteq \sigma_{(Type \neq snack) \wedge (Type \neq soda)}(\texttt{Item})\}$. In this case, the optional item-subset X_3 contains any item that is not of type *snack* or *soda*. These items may be included so long as an itemset contains some mandatory items (i.e., some items from X_1 and some items from X_2).
- For $C_2 \equiv max(S.Price) \geq 100$, C_2 is succinct but not antimonotone, and is not of a form equivalent to $S.A \supseteq CS$. Hence, its MGF is $\{X_1 \cup X_2 \mid X_1 \subseteq \sigma_{Price \geq 100}(\texttt{Item})\ \&\ X_1 \neq \emptyset\ \&\ X_2 \subseteq \sigma_{Price < 100}(\texttt{Item})\}$. This is different from that for constraint C_1 above because once there is at least one item with $Price \geq 100$, then it is acceptable to include any item whose $Price < 100$. This is what the optional item-subset X_2 represents.
- For $C_3 \equiv S.Price \geq 50$, C_3 is succinct and antimonotone, and is not of a form equivalent to $S.A \supseteq CS$. Hence, its MGF is $\{X_1 \mid X_1 \subseteq \sigma_{Price \geq 50}(\texttt{Item})\ \&\ X_1 \neq \emptyset\}$. In this case, there is only one mandatory item-subset X_1, but there is no optional item-subset.

Given the above lemma, we can now complete our proof that a relaxing change to a succinct constraint C_{old} admits a delta MGF. Our proof is divided into two cases, depending on whether or not the constraint is of the form $S.A \supseteq CS$. In Lemma 5.5, we first deal with the simpler case where the constraint is not of the form $S.A \supseteq CS$. Then, in Lemma 5.8, we deal with the more complicated case where the constraint is of this form.

LEMMA 5.5. *Let C_{old} be a succinct constraint not of a form equivalent to $S.A \supseteq CS$, where (i) A is an attribute of a set S and (ii) $CS = \{cn_1, \ldots, cn_k\}$ is a set of constants for attribute A. Let C_{new} be a relaxing change to C_{old}. Then, there exists a delta member generating function $MGF_{C_{new}/C_{old}}$ that generates precisely those itemsets that satisfy C_{new} but not C_{old}.*

PROOF. This proof is divided into two cases: one for succinct but non-antimonotone constraints, and another for succinct and antimonotone constraints. First, let us consider the case in which C is succinct but non-antimonotone. According to Lemma 5.3(b), the MGF of C_{old} is of the form $\{X_1 \cup X_2 \mid X_1 \subseteq \sigma_{p_1}(\texttt{Item})\ \&\ X_1 \neq \emptyset\ \&\ X_2 \subseteq \sigma_{p_2}(\texttt{Item})\}$, where $p_2 = \neg p_1$. By definition of a relaxing change, the MGF of the C_{new} must be of the form $\{X_1 \cup X_2 \mid X_1 \subseteq \sigma_{p'_1}(\texttt{Item})\ \&\ X_1 \neq \emptyset\ \&\ X_2 \subseteq \sigma_{p'_2}(\texttt{Item})\}$, where $p'_2 = \neg p'_1$. Then, consider the delta member generating function $MGF_{C_{new}/C_{old}} \equiv \{X_1 \cup X_2 \mid X_1 \subseteq \sigma_{p'_1 \wedge \neg p_1}(\texttt{Item})\ \&\ X_1 \neq \emptyset\ \&\ X_2 \subseteq \sigma_{p'_2}(\texttt{Item})\}$. It is easy to verify that this delta MGF precisely enumerates those itemsets that satisfy C_{new} but not C_{old}.

Then, let us consider the case in which C is both succinct and antimonotone. In accordance with Lemma 5.3(c), the MGF of C_{old} is of the form $\{X_1 \mid X_1 \subseteq \sigma_{p_1}(\texttt{Item})\ \&\ X_1 \neq \emptyset\}$. By definition of a relaxing change, the MGF of the C_{new} must be of the form $\{X_1 \mid X_1 \subseteq \sigma_{p'_1}(\texttt{Item})\ \&\ X_1 \neq \emptyset\}$. Then, consider $MGF_{C_{new}/C_{old}} \equiv \{X_1 \cup X_2 \mid X_1 \subseteq \sigma_{p'_1 \wedge \neg p_1}(\texttt{Item})\ \&\ X_1 \neq \emptyset\ \&\ X_2 \subseteq \sigma_{p_1}(\texttt{Item})\}$. Again, it is easy to verify that this delta MGF precisely enumerates those itemsets that satisfy C_{new} but not C_{old}. □

The example below illustrates the use of Lemma 5.5.

Example 5.6. Let us consider the following relaxing changes to the succinct constraints that are not of the form $S.A \supseteq CS$.

—Suppose a succinct but non-antimonotone constraint $C_{old} \equiv max(S.Price) \geq 100$ is relaxed to $C_{new} \equiv max(S.Price) \geq 60$. Then, $MGF_{C_{new}/C_{old}}$ is

$$\{X_1 \cup X_2 \mid X_1 \subseteq \sigma_{60 \leq Price < 100}(\texttt{Item})\ \&\ X_1 \neq \emptyset\ \&\ X_2 \subseteq \sigma_{Price < 60}(\texttt{Item})\}.$$

It is easy to verify that this delta MGF precisely enumerates those itemsets that satisfy C_{new} but not C_{old}.

—Suppose a succinct and antimonotone constraint $C_{old} \equiv S.Price \geq 50$ is relaxed to $C_{new} \equiv S.Price \geq 30$. Then, $MGF_{C_{new}/C_{old}}$ is

$$\{X_1 \cup X_2 \mid X_1 \subseteq \sigma_{30 \leq Price < 50}(\texttt{Item})\ \&\ X_1 \neq \emptyset\ \&\ X_2 \subseteq \sigma_{Price \geq 50}(\texttt{Item})\}.$$

It is easy to verify that this delta MGF precisely enumerates those itemsets that satisfy C_{new} but not C_{old}.

Note that Lemma 5.5 (as well as Example 5.6) above deals with the case where there is only one mandatory item-subset (i.e., $X_1 \neq \emptyset$). The situation is more complicated when there are more mandatory item-subsets, as shown in the following example.

Example 5.7. Suppose a constraint $C_{old} \equiv S.Type \supseteq \{snack, soda, meat\}$ is relaxed to $C_{new} \equiv S.Type \supseteq \{meat\}$. We cannot follow the approach taken in Lemma 5.5. The reason is that, on the one hand, we should not suppress any *soda* or *snack* item from being included; on the other hand, we should not *simultaneously* include both *soda* and *snack* items. In other words, it is not correct to have the following MGF:

$$\begin{aligned}\{X_1 \cup X_2 \mid X_1 &\subseteq \sigma_{Type=meat}(\texttt{Item})\ \&\ X_1 \neq \emptyset\ \&\\ X_2 &\subseteq \sigma_{(Type \neq meat) \wedge (Type \neq snack) \wedge (Type \neq soda)}(\texttt{Item})\}\end{aligned}$$

which excludes too many itemsets. On the other hand, it is also incorrect to have the following MGF:

$$\begin{aligned}\{X_1 \cup X_2 \mid X_1 &\subseteq \sigma_{Type=meat}(\texttt{Item})\ \&\ X_1 \neq \emptyset\ \&\\ X_2 &\subseteq \sigma_{(Type=snack) \vee (Type=soda)}(\texttt{Item})\}\end{aligned}$$

which includes itemsets that simultaneously contain both *soda* and *snack* items. Hence, the correct $MGF_{C_{new}/C_{old}}$ is as follows:

$$\begin{aligned}\{X_1 \cup X_2 \mid X_1 &\subseteq \sigma_{Type=meat}(\texttt{Item})\ \&\ X_1 \neq \emptyset\ \&\\ X_2 &\subseteq \sigma_{(Type \neq meat) \wedge (Type \neq snack)}(\texttt{Item})\} \cup\\ \{X_1 \cup X_3 \mid X_1 &\subseteq \sigma_{Type=meat}(\texttt{Item})\ \&\ X_1 \neq \emptyset\ \&\\ X_3 &\subseteq \sigma_{(Type \neq meat) \wedge (Type \neq soda)}(\texttt{Item})\}.\end{aligned}$$

This is a case in which a general MGF is needed.

The lemma below shows that a relaxing change to a succinct constraint C_{old} of the form $S.A \supseteq CS$ also admits a delta MGF.

LEMMA 5.8. *Let C_{old} be a succinct constraint of a form equivalent to $S.A \supseteq CS$, where A is an attribute of a set S, and $CS = \{cn_1, \ldots, cn_k\}$ is a set of constants for attribute A. Let C_{new} be a relaxing change to C_{old}. Then, there exists a delta member generating function $MGF_{C_{new}/C_{old}}$ that generates precisely those itemsets that satisfy C_{new} but not C_{old}.*

PROOF. According to Lemma 5.3(a), where $CS = \{cn_1, \ldots, cn_k\}$, the MGF of C_{old} is of the form $\{X_1 \cup \cdots \cup X_k \cup X_{k+1} \mid X_i \subseteq \sigma_{A=cn_i}(\texttt{Item})\ \&\ X_i \neq \emptyset$, for $1 \leq i \leq k$; $X_{k+1} \subseteq \sigma_{(A \neq cn_1) \wedge \cdots \wedge (A \neq cn_k)}(\texttt{Item})\}$. By definition of a relaxing change, C_{new} must be of the form $S.A \supseteq CS'$, where $CS' \subset CS$. Without loss of generality, let $CS - CS' = \{cn_{m+1}, \ldots, cn_k\}$. The MGF of C_{new} must be of the form $\{X_1 \cup \cdots \cup X_m \cup X_{k+1} \mid X_i \subseteq \sigma_{A=cn_i}(\texttt{Item})\ \&\ X_i \neq \emptyset$, for $1 \leq i \leq m; X_{k+1} \subseteq \sigma_{(A \neq cn_1) \wedge \cdots \wedge (A \neq cn_m)}(\texttt{Item})\}$. Now, consider $MGF_{C_{new}/C_{old}} \equiv \bigcup_{j=1}^{k-m} MGF_j$, where

$$\begin{aligned} MGF_j \equiv \{&X_1 \cup \cdots \cup X_m \cup X_{k+1} \mid X_i \subseteq \sigma_{A=cn_i}(\texttt{Item})\ \&\ X_i \neq \emptyset, \text{ for } 1 \leq i \leq m; \\ &X_{k+1} \subseteq \sigma_{(A \neq cn_1) \wedge \cdots \wedge (A \neq cn_m) \wedge (A \neq cn_{m+j})}(\texttt{Item})\}. \end{aligned}$$

Each MGF_j guarantees that no item with $A = cn_{m+j}$ is included in the generation. It is easy to verify that the above delta MGF precisely enumerates those itemsets that satisfy C_{new} but not C_{old}. □

Lemmas 5.5 and 5.8 above complete our proof that when C_{old} is a succinct constraint, its relaxation can be efficiently handled with a delta MGF.

5.4 Discussion: Relaxing One of Multiple Constraints Dynamically

So far, we have considered handling dynamic changes to a constraint in isolation. In practice, more than one constraint may be imposed on frequent-set mining and any of them may be changed dynamically. If the change is a tightening one, it can be handled easily. What if it is a relaxing one? Can we exactly generate those itemsets that satisfy all constraints—including the changed one but not the old version of the changed constraint? To answer these questions, let us first present Lemma 5.9 showing how to combine multiple MGFs into one MGF, and then present Theorem 5.11 showing how we can exactly generate those itemsets.

LEMMA 5.9. *Given multiple succinct constraints, their corresponding MGFs can be combined into one general MGF representing the conjunction of the succinct constraints.*

PROOF. First, let us consider the combination of two basic member generating functions MGF_1 and MGF_2 corresponding to the two succinct constraints. The proof is divided into two cases, depending on whether one of the succinct constraints is also antimonotone.

We start with the simpler case where at least one of the constraints is succinct and antimonotone. Recall from Definition 2.3 that the itemsets generated using the MGF contain some mandatory items and may contain some optional items. The mandatory items are generated using the *mandatory selection predicates*, and there may be more than one mandatory selection predicate in an MGF (e.g., for a constraint of the form $S.A \supseteq CS$); the optional items are generated using the *optional selection predicate*. From Lemma 5.3(c), we know that the MGF corresponding to a succinct and antimonotone constraint contains no optional selection predicate. Hence, when combining two MGFs in this simpler case, the mandatory items for the combined MGF can be enumerated using the conjunction of the mandatory selection predicates from both MGFs. More precisely, we are given MGF_1 and MGF_2. Without loss of generality, suppose MGF_2 is the MGF for the succinct and antimonotone constraint. Then, MGF_2 is of the form $\{Y_1 \mid Y_1 \subseteq \sigma_{q_1}(\texttt{Item})\ \&\ Y_1 \neq \emptyset\}$ for some selection predicate q_1. If the constraint corresponding to MGF_1 is also antimonotone, then MGF_1 is of the form $\{X_1 \mid X_1 \subseteq \sigma_{p_1}(\texttt{Item})\ \&\ X_1 \neq \emptyset\}$ for some selection predicate p_1; hence, the combined MGF corresponding to the conjunction can be constructed as $\{Z_1 \mid Z_1 \subseteq \sigma_{p_1 \wedge q_1}(\texttt{Item})\ \&\ Z_1 \neq \emptyset\}$. Otherwise, MGF_1 is of the form $\{X_1 \cup \cdots \cup X_k \cup X_{k+1} \mid X_i \subseteq \sigma_{p_i}(\texttt{Item})\ \&\ X_i \neq \emptyset$, for $1 \leq i \leq k; X_{k+1} \subseteq \sigma_{p_{k+1}}(\texttt{Item})\}$ for some selection predicates $p_1, \ldots, p_{k+1}$; hence, the combined MGF corresponding to the conjunction can be constructed as $\{Z_1 \cup \cdots \cup Z_k \cup Z_{k+1} \mid Z_i \subseteq \sigma_{p_i \wedge q_1}(\texttt{Item})\ \&\ Z_i \neq \emptyset$, for $1 \leq i \leq k; Z_{k+1} \subseteq \sigma_{p_{k+1} \wedge q_1}(\texttt{Item})\}$.

We then consider the more complicated case where none of the succinct constraints is antimonotone. Given that each MGF has two types of selection predicates (mandatory and optional), there are four possible types of conjunctions of these predicates (e.g., conjunctions of mandatory predicates of MGF_1 with mandatory predicates of MGF_2, conjunctions of mandatory predicates of MGF_1 with the optional predicate of MGF_2). Among these types of conjunctions, the mandatory items for the combined MGF can be enumerated using either of the following:

(a) the conjunction of the mandatory selection predicates from both MGFs; or

(b) the conjunction of the mandatory selection predicates from MGF_1 and the optional selection predicate from MGF_2, together with the conjunction of the optional selection predicate from MGF_1 and the mandatory selection predicates from MGF_2.

The resulting MGF is a union of the MGFs using part (a) and the MGFs using part (b). More precisely, we are given MGF_1 and MGF_2, where MGF_1 is of the form $\{X_1 \cup \cdots \cup X_k \cup X_{k+1} \mid X_i \subseteq \sigma_{p_i}(\texttt{Item})\ \&\ X_i \neq \emptyset$, for $1 \leq i \leq k; X_{k+1} \subseteq \sigma_{p_{k+1}}(\texttt{Item})\}$ for some selection predicates $p_1, \ldots, p_{k+1}$, and MGF_2 is of the form $\{Y_1 \cup \cdots \cup Y_h \cup Y_{h+1} \mid Y_j \subseteq \sigma_{q_j}(\texttt{Item})\ \&\ Y_j \neq \emptyset$, for $1 \leq j \leq h; Y_{h+1} \subseteq \sigma_{q_{h+1}}(\texttt{Item})\}$ for some selection predicates $q_1, \ldots, q_{h+1}$. We define the notion of a *(k, h)-cover* for the product $\{1, \ldots, k+1\} \times \{1, \ldots, h+1\}$ as any subset $S \subseteq \{1, \ldots, k+1\} \times \{1, \ldots, h+1\}$ satisfying the following conditions:

(1) for all $i \in \{1, \ldots, k\}$, there exists j such that $(i, j) \in S$; and

(2) for all $j \in \{1, \ldots, h\}$, there exists i such that $(i, j) \in S$.

Such a cover is *minimal* if none of its proper subsets is a (k, h)-cover. Therefore, the MGF corresponding to the conjunction can be constructed as

$$\bigcup_{\substack{S \text{ is any minimal } (k,h)\text{-cover of} \\ \{1,\ldots,k+1\}\times\{1,\ldots,h+1\}}} MGF_{1,2}$$

where $MGF_{1,2}$ is of the form $\{Z_1 \cup Z_2 \cup Z_3 \cup \cdots \cup W \mid Z_{ij} \subseteq \sigma_{p_i \wedge q_j}(\texttt{Item})\ \&\ Z_{ij} \neq \emptyset$, for all $(i, j) \in S; W \subseteq \sigma_{\bigvee_{ij}(p_i \wedge q_j)}(\texttt{Item})$, for all $(i, j) \notin S\}$.

The above construction shows the combination of two basic MGFs. This technique can be easily generalized to handle the combination of two general MGFs (i.e., combining $MGF_1 \equiv \bigcup_{u=1}^{l_1} MGF_{1u}$ with $MGF_2 \equiv \bigcup_{v=1}^{l_2} MGF_{2v}$ into one MGF), by applying the technique to each (u, v)-pair and taking the union of the results. In other words, the MGF for the conjunction is of the following form:

$$\bigcup_{1\le u\le l_1;\ 1\le v\le l_2} \left(\bigcup_{\substack{S \text{ is any minimal } (k,h)\text{-cover of} \\ \{1,\ldots,k+1\}\times\{1,\ldots,h+1\}}} MGF_{1u,2v} \right),$$

where $MGF_{1u,2v}$ is of the form $\{Z_1 \cup Z_2 \cup Z_3 \cup \cdots \cup W \mid Z_{ij} \subseteq \sigma_{p_i \wedge q_j}(\texttt{Item})\ \&\ Z_{ij} \neq \emptyset$, for all $(i, j) \in S; W \subseteq \sigma_{\bigvee_{ij}(p_i \wedge q_j)}(\texttt{Item})$, for all $(i, j) \notin S\}$ for each (u, v)-pair. This completes the proof for combining two MGFs. Next, we consider the combination of more than two MGFs.

Given that the result of combining any two succinct constraints is an MGF of the form we have described above, combining the MGFs of more than two succinct constraints can be done by repeated applications of the above construction. This completes the proof of combining multiple MGFs for succinct constraints into one MGF. □

The example below illustrates the use of the above lemma.

Example 5.10. Suppose we are given the following three succinct constraints: $C_1 \equiv max(S.Price) \leq 200$, $C_2 \equiv max(S.Price) \geq 100$, and $C_3 \equiv S.Type \supseteq \{snack, soda\}$. Then, their corresponding MGFs are as follows:

$$\begin{aligned}
MGF_1 &\equiv \{X_1 \mid X_1 \subseteq \sigma_{Price \leq 200}(\texttt{Item})\ \&\ X_1 \neq \emptyset\}, \\
MGF_2 &\equiv \{Y_1 \cup Y_2 \mid Y_1 \subseteq \sigma_{Price \geq 100}(\texttt{Item})\ \&\ Y_1 \neq \emptyset; Y_2 \subseteq \sigma_{Price<100}(\texttt{Item})\}, \quad \text{and} \\
MGF_3 &\equiv \{U_1 \cup U_2 \cup U_3 \mid U_1 \subseteq \sigma_{Type=snack}(\texttt{Item})\ \&\ U_1 \neq \emptyset; U_2 \subseteq \sigma_{Type=soda}(\texttt{Item}) \\
&\qquad \&\ U_2 \neq \emptyset; U_3 \subseteq \sigma_{(Type \neq snack)\wedge(Type \neq soda)}(\texttt{Item})\},
\end{aligned}$$

respectively. Here, C_1 is succinct antimonotone, while both C_2 and C_3 are succinct non-antimonotone. Then, the combined MGF for the first two succinct constraints is as follows:

$$\{Z_1 \cup W \ \mid\ Z_1 \subseteq \sigma_{100 \leq Price \leq 200}(\texttt{Item})\ \&\ Z_1 \neq \emptyset; W \subseteq \sigma_{Price<100}(\texttt{Item})\}.$$

The combined MGF for all three succinct constraints can be constructed by combining the above result with MGF_3. To simplify our representation, we denote the conjunctions of the selection predicates as follows:

$$\begin{aligned}
pred_{11} &\equiv (100 \le Price \le 200) \wedge (Type = snack),\\
pred_{12} &\equiv (100 \le Price \le 200) \wedge (Type = soda),\\
pred_{13} &\equiv (100 \le Price \le 200) \wedge (Type \ne snack) \wedge (Type \ne soda),\\
pred_{21} &\equiv (Price < 100) \wedge (Type = snack),\\
pred_{22} &\equiv (Price < 100) \wedge (Type = soda), \text{ and}\\
pred_{23} &\equiv (Price < 100) \wedge (Type \ne snack) \wedge (Type \ne soda).
\end{aligned}$$

Then, the resulting MGF is a general MGF consisting of four basic MGFs:

$$MGF_5 \cup MGF_6 \cup MGF_7 \cup MGF_8.$$

Here, MGF_5 is

$$\{Z_1 \cup Z_2 \cup W_1 \mid Z_1 \subseteq \sigma_{pred_{11}}(\mathtt{Item}) \;\&\; Z_1 \ne \emptyset; Z_2 \subseteq \sigma_{pred_{12}}(\mathtt{Item}) \;\&\; Z_2 \ne \emptyset; W_1 \subseteq \sigma_{pred_{13} \vee pred_{21} \vee pred_{22} \vee pred_{23}}(\mathtt{Item})\}$$

which generates itemsets with mandatory items (i.e., itemsets containing at least one *snack* item and one *soda* item, both with prices between 100 and 200 inclusive) enumerated using the conjunction of the mandatory selection predicates from the given MGFs (i.e., MGF_1, MGF_2 and MGF_3). Then, the remaining three basic MGFs (i.e., MGF_6, MGF_7 and MGF_8) generate itemsets with mandatory items enumerated using the conjunction of appropriate mandatory selection predicates from some MGFs with appropriate optional selection predicates from other MGFs. Specifically, MGF_6 is

$$\{Z_3 \cup Z_4 \cup W_2 \mid Z_3 \subseteq \sigma_{pred_{11}}(\mathtt{Item}) \;\&\; Z_3 \ne \emptyset; Z_4 \subseteq \sigma_{pred_{22}}(\mathtt{Item}) \;\&\; Z_4 \ne \emptyset; W_2 \subseteq \sigma_{pred_{12} \vee pred_{13} \vee pred_{21} \vee pred_{23}}(\mathtt{Item})\}$$

which generates itemsets with (i) at least one *snack* item within the mandatory price range (i.e., $100 \le Price \le 200$), and (ii) at least one *soda* item within the optional price range (i.e., $Price < 100$). MGF_7 is

$$\{Z_5 \cup Z_6 \cup W_3 \mid Z_5 \subseteq \sigma_{pred_{12}}(\mathtt{Item}) \;\&\; Z_5 \ne \emptyset; Z_6 \subseteq \sigma_{pred_{21}}(\mathtt{Item}) \;\&\; Z_6 \ne \emptyset; W_3 \subseteq \sigma_{pred_{11} \vee pred_{13} \vee pred_{22} \vee pred_{23}}(\mathtt{Item})\}$$

which generates (i) itemsets with at least one *soda* item within the mandatory price range, and (ii) at least one *snack* item within the optional price range. Finally, MGF_8 is

$$\{Z_7 \cup Z_8 \cup Z_9 \cup W_4 \mid Z_7 \subseteq \sigma_{pred_{21}}(\mathtt{Item}) \;\&\; Z_7 \ne \emptyset; Z_8 \subseteq \sigma_{pred_{22}}(\mathtt{Item}) \;\&\; Z_8 \ne \emptyset; Z_9 \subseteq \sigma_{pred_{13}}(\mathtt{Item}) \;\&\; Z_9 \ne \emptyset; W_4 \subseteq \sigma_{pred_{11} \vee pred_{12} \vee pred_{23}}(\mathtt{Item})\}$$

which generates itemsets with (i) at least one *snack* item and one *soda* item, both within the optional price range, and (ii) at least one item of optional types (i.e., non-*snack* and non-*soda*) but within the mandatory price range.

Given the above lemma showing how to combine multiple MGFs into a general MGF, we now present the following theorem showing how we can exactly generating those itemsets that satisfy all constraints including the changed one but not the old version of the changed constraints.

THEOREM 5.11. *Let $C_1, \ldots, C_n$ be the succinct constraints imposed on frequent-set mining using DCF. Suppose we relax any of these constraints, say C_i. Then, there is an MGF which generates precisely those itemsets that satisfy $C_1 \wedge \cdots \wedge C_{i-1} \wedge C_i' \wedge C_{i+1} \wedge \cdots \wedge C_n$ (where C_i' is the relaxed version of C_i) but do not satisfy C_i.*

PROOF. From Lemmas 5.5 and 5.8, we know there is a delta member generating function MGF_{C_i'/C_i} which generates exactly those itemsets that satisfy C_i' but not C_i. From Lemma 5.9, we know that there exists a general MGF corresponding to $C_1 \wedge \cdots \wedge C_{i-1} \wedge C_{i+1} \wedge \cdots \wedge C_n$. Consequently, the desired MGF can simply be obtained by constructing the MGF corresponding to the conjunction of $(C_1 \wedge \cdots \wedge C_{i-1} \wedge C_{i+1} \wedge \cdots \wedge C_n)$ and $(C_i' \wedge \neg C_i)$, as discussed in Lemma 5.9. □

We illustrate the use of the above theorem with the following example.

Example 5.12. Let constraints $C_1 \equiv S.Type \supseteq \{snack, soda, meat\}$ and $C_2 \equiv max(S.Price) \geq 100$ be the succinct constraints imposed on frequent-set mining. Suppose C_1 is relaxed to $C_1' \equiv S.Type \supseteq \{meat\}$. Then, recall from Example 5.7 that $MGF_{C_1'/C_1} = MGF_{11} \cup MGF_{12}$, which is equivalent to the following:

$$\begin{aligned}\{X_1 \cup X_2 \mid X_1 &\subseteq \sigma_{Type=meat}(\texttt{Item}) \;\&\; X_1 \neq \emptyset \;\&\\ X_2 &\subseteq \sigma_{(Type\neq meat)\wedge(Type\neq snack)}(\texttt{Item})\} \;\cup\\ \{X_1 \cup X_3 \mid X_1 &\subseteq \sigma_{Type=meat}(\texttt{Item}) \;\&\; X_1 \neq \emptyset \;\&\\ X_3 &\subseteq \sigma_{(Type\neq meat)\wedge(Type\neq soda)}(\texttt{Item})\}.\end{aligned}$$

And, recall from Example 5.10 that the MGF for C_2 is as follows:

$$MGF_{21} \equiv \{Y_1 \cup Y_2 \mid Y_1 \subseteq \sigma_{Price\geq 100}(\texttt{Item}) \;\&\; Y_1 \neq \emptyset \;\&\; Y_2 \subseteq \sigma_{Price<100}(\texttt{Item})\}.$$

Hence, the resulting MGF which generates precisely those itemsets satisfying $C_1' \wedge C_2$ but not satisfying C_1 is $MGF_{11,21} \cup MGF_{12,21}$, where $MGF_{11,21}$ is

$$\begin{aligned}\{Z_{11} \cup W_1 \mid Z_{11} &\subseteq \sigma_{(Type=meat)\wedge(Price\geq 100)}(\texttt{Item}) \;\&\; Z_{11} \neq \emptyset\\ W_1 &\subseteq \sigma_{(Type=meat\wedge Price<100)\vee(Type\neq meat\wedge Type\neq snack)}(\texttt{Item})\} \;\cup\\ \{Z_{12} \cup Z_{21} \cup W_2 \mid Z_{12} &\subseteq \sigma_{(Type=meat)\wedge(Price<100)}(\texttt{Item}) \;\&\; Z_{12} \neq \emptyset;\\ Z_{21} &\subseteq \sigma_{(Type\neq meat\wedge Type\neq snack)\wedge(Price\geq 100)}(\texttt{Item}) \;\&\; Z_{21} \neq \emptyset;\\ W_2 &\subseteq \sigma_{(Type=meat\wedge Price\geq 100)\vee(Type\neq meat\wedge Type\neq snack\wedge Price<100)}(\texttt{Item})\}\end{aligned}$$

which generates *meat* with optional non-*snack* items, and $MGF_{12,21}$ is

$$\begin{aligned}\{Z_{11} \cup W_3 \mid Z_{11} &\subseteq \sigma_{(Type=meat)\wedge(Price\geq 100)}(\texttt{Item}) \;\&\; Z_{11} \neq \emptyset;\\ W_3 &\subseteq \sigma_{(Type=meat\wedge Price<100)\vee(Type\neq meat\wedge Type\neq soda)}(\texttt{Item})\} \;\cup\\ \{Z_{12} \cup Z_{31} \cup W_4 \mid Z_{12} &\subseteq \sigma_{(Type=meat)\wedge(Price<100)}(\texttt{Item}) \;\&\; Z_{12} \neq \emptyset;\\ Z_{31} &\subseteq \sigma_{(Type\neq meat\wedge Type\neq soda)\wedge(Price\geq 100)}(\texttt{Item}) \;\&\; Z_{31} \neq \emptyset;\\ W_4 &\subseteq \sigma_{(Type=meat\wedge Price\geq 100)\vee(Type\neq meat\wedge Type\neq soda\wedge Price<100)}(\texttt{Item})\}\end{aligned}$$

which generates *meat* with optional non-*soda* items. Note that all the itemsets generated by $MGF_{11,21}$ and $MGF_{12,21}$ contain at least one item with $Price \geq 100$.

In sum, we showed in this section how DCF handles dynamic changes to nonfrequency constraints. In Section 8, we present experimental results evaluating the effectiveness of the various optimizations proposed so far.

6. HANDLING LIMITED BUFFER SPACE

So far, we have discussed how DCF performs the efficient dynamic mining of constrained frequent sets (with a liberal amount of buffer space). In this section, we discuss how we can extend DCF to perform the efficient dynamic mining of constrained frequent sets when *the available buffer space is limited*.

As described in Hidber's paper [1999], Carma only works when there is enough buffer space to contain the entire lattice V, which is a superset of *all* frequent itemsets. For many practical situations, this could be too strong an assumption. Take the synthetic dataset used in Hidber's paper as an example, which considers a domain of 10^4 items. For this domain, the total number of itemsets of size 4 (i.e., 4-itemsets) is about 4×10^{14}. Even if a small fraction of those (say 0.1%) are potentially frequent, just the number of 4-itemsets in V is about 4×10^{11}. Assuming a byte for each of the three counters (e.g., *maxMissed*(v), etc.), this alone calls for about 1.2 terabytes of buffer space for just 4-itemsets! If the number of items in the domain is doubled (i.e., 2×10^4 domain items), the amount of buffer space required for just 4-itemsets jumps to 20 terabytes! Clearly, Carma's requirement to simultaneously keep (at least) all the frequent itemsets in the buffer pool can be hard to satisfy. Towards the development of a realistic and practical environment for mining (constrained) frequent sets, we show in this section how DCF can operate in limited buffer space. We first sketch the key aspects of how DCF handles limited buffer space by presenting a skeleton of DCF. To simplify our presentation, the pseudocode given in Figure 4 omits some implementation and optimization details, which will be discussed in Section 6.2. We then analyze the behavior of DCF more rigorously.

6.1 General Ideas

Recall from Section 2.3 that on reading a transaction t_i, Carma inserts appropriate subsets v of t_i into (the lattice V contained in) the buffer. A simple way to handle limited buffer space is to conduct the insertion of v as usual if there is space in the buffer, and to put v into a *waiting list WL*—maintained in a file—if the buffer is full. When the current batch of itemsets in the buffer have been processed, these itemsets are moved to the "done list" *DL*; a new batch is loaded into the buffer from *WL* and counted in a next iteration. However, a serious drawback is that a prohibitive number of I/O is required to enforce the following Insertion Condition (as stated in Section 2.3), which is critical for the success of Carma:

$$\text{for all } w \subset v\text{: } w \in V \text{ and } \mathit{firstTrans}(w) < i \text{ and } \mathit{maxSupport}(w) \geq \sigma_i,$$

where $v \subseteq t_i$.

A careful examination of the above Insertion Condition reveals that insertions of itemsets are done in the order of subsets first, followed by supersets. However, the complication in Carma is that insertions are also based on the appearance of items in the transactions. Thus, there can be itemsets of widely

```
Procedure DCF-Phase I (transaction database $TDB = \{t_1, \ldots, t_n\}$,
support sequence $\langle\sigma_1, \ldots, \sigma_n\rangle$, set of constraints $\mathcal{C}$, buffer pool $B$) {
      /* Step (0) Initialization */
(0)   $minCard = 0$;
      openfile($WL$); /* waiting list */
      openfile($DL$); /* "done list" (which contains processed itemsets) */
(1)   repeat {
(2)      for $i$ from 1 to $n$ { /* start scanning $TDB$ */
            /* Step (3) Increment: Exactly as in Carma */
(3)         for all $v$ in $B$ with $v \subseteq t_i$ { increment $count(v)$; }
            /* Steps (4) to (14) Insert: Several key differences from Carma, note the "else" clause */
(4)         if(($B$ is not full) and (the insertions as in Carma do not overflow $B$)) {
(5)            for each $v$ generated by $MGF_{\mathcal{SC}}(t_i^{SSM})$ where $v \models (\mathcal{C} - \mathcal{SC})$ {
(6)               perform insertions of $v$ as in Carma, and update $maxCard$ if necessary;
               }
(7)         } else { /* $B$ is full */
               /* Step (8): Intelligently generate $v$ that has not been processed, using an MGF */
(8)            for each $v$ generated by $MGF_{\mathcal{SC}}(t_i^{SSM})$ where ($v \models (\mathcal{C} - \mathcal{SC})$ & $v \notin (B \cup WL)$ &
               $(minCard + 1) \leq |v| \leq (maxCard + 1)$) {
(9)               if (($|v| \geq maxCard$) and ($\forall w \subset v$ such that $w \in B$: $maxSupport(w) \geq \sigma_i$)) {
                     /* If all subsets $w$ (of $v$) that are in $B$ satisfy the $maxSupport$ condition,
                     then $v$ (of cardinality $\geq maxCard$) will be processed in a future batch. */
(10)                 insert $v$ into $WL$;
(11)              } else if (($|v| < maxCard$) & ($\forall w \subset v$ such that $w \in B$: $maxSupport(w) \geq \sigma_i$)) {
                     /* If all subsets $w$ (of $v$) that are in $B$ satisfy the $maxSupport$ condition,
                     then $v$ (of cardinality $< maxCard$) is being processed in the current batch. */
                     /* The replacement strategy replaces an itemset $u$ (of higher cardinality) by
                     an itemset $v$ (having lower cardinality). */
(12)                 pick an itemset $u$ in $B$ such that $|u| = maxCard$;
(13)                 insert $u$ into $WL$; decrement $maxCard$ if necessary;
(14)                 insert $v$ into $B$;
                     initialize $count(v)$, $maxMissed(v)$, and $firstTrans(v)$ as in Carma;
                  } /* end else-if */
               } /* end for-each */
            } /* end else */
(15)        prune exactly as in Carma;
         } /* end for; complete one scan of $TDB$ */
         /* Steps (16) to (21): Set up buffer $B$ for the next batch */
(16)     for all $v$ in $B$ {
            /* move results from the buffer $B$ to the "done list" $DL$ */
(17)        copy $count(v)$, $maxMissed(v)$, and $firstTrans(v)$ to $DL$;
         }
(18)     if ($WL$ is empty) { break out of the repeat-loop; }
(19)     else { /* reload a new batch from the waiting list $WL$ into the buffer $B$ */
(20)        fill $B$ with as many $v$'s as possible from $WL$ in ascending order of size;
(21)        update $minCard$ and $maxCard$ accordingly for the next iteration;
         }
      } /* end repeat-loop */
(22)  return $DL$;
}
```

Fig. 4. Phase I of algorithm DCF.

varied cardinalities in the buffer simultaneously. When there is only limited buffer space available, this works against the efficient verification of the above Insertion Condition. To this end, DCF adopts the following two strategies:

- DCF enforces a levelwise counting strategy. Specifically, when the buffer is full, if an itemset v is to be inserted, then v "replaces" an itemset u in the buffer if u is of a strictly higher cardinality (i.e., u, instead of v, is moved to the waiting list *WL*). The rationale is that v, being of a lower cardinality than u, is expected to be required more often than u for the verification of the above Insertion Condition. Thus, when both cannot fit in the buffer, preferring the smaller itemset over the larger itemset should help reduce I/O.
- DCF modifies the above Insertion Condition to consider only those subsets that are in the buffer B. The rationale is that to decide whether v is to be inserted, it is not necessary to check all subsets w of v. When there is only a limited amount of buffer space, checking all subsets requires an excessive amount of I/O. The strategy used by DCF is to restrict the checking to *"immediate" subsets* (of v) that are *in the buffer*. In this way, no I/O is involved here. This represents a tradeoff situation because more itemsets may be added to the waiting list *WL*.

So far, we have dealt with the issue of which itemsets are to be kept in the buffer and which are to be moved to the waiting list *WL*. An equally important issue is the maintenance of *WL*. The insert operation of Carma (see the Insertion Condition in Section 2.3) generates numerous subsets v of transaction t_i, many of which may need to be inserted. This could create a huge waiting list. DCF intelligently deals with this issue as follows. For the current batch of itemsets in the buffer, it restricts the generation of subsets to a maximum cardinality of ($maxCard + 1$), where *maxCard* is the maximum cardinality of the itemsets that are currently in the buffer. Note that this restricted generation does *not increase* the number of itemsets to be processed, *nor* does it affect the number of scans of the transaction database *TDB*. This is because subsets that are not generated for the current batch are of cardinalities at least ($maxCard + 2$), and are to be put in the *waiting list*. In due course, their generation can be triggered by itemsets of size ($maxCard + 1$), when the latter eventually get into the buffer pool. See the example below.

Example 6.1. Suppose *maxCard* is 3. Then, for the current batch of itemsets in the buffer, DCF only generates subsets of t_i to a maximum size of 4. Those subsets are put in the waiting list *WL*, if they are not there already. Subsequently, when those subsets of size 4 are loaded into the buffer for their counting, then the subsets of t_i of size at least 5 will be considered and added to *WL* (if necessary) at that time.

In short, when only limited buffer space is available, the main design objective of DCF is to use the buffer space judiciously to reduce I/O as much as possible. It operates in a highly structured, levelwise fashion. Figure 4 gives the pseudocode for DCF. Let us use the simple example below to illustrate how DCF works.

Example 6.2. Suppose the item domain is $\{a, b, c, d, e, f, \ldots\}$, and the first few transactions in *TDB* are $t_1 = \{a, b, c, d\}$, $t_2 = \{a, b, c\}$, $t_3 = \{a, b, c, d\}$, and $t_4 = \{a, b, e, f\}$. To simplify our example, let us assume that the buffer B can only accommodate 8 itemsets, and that the itemsets discussed below *all* satisfy the given constraints $\mathcal{C}$ (i.e., all are generated in Step (5) or (8)).

After transaction t_1 is read, Steps (5) and (6) are executed, and B now contains itemsets $\{a\}$, $\{b\}$, $\{c\}$, and $\{d\}$, with *maxCard* set to 1.

After t_2 is read, Steps (5) and (6) are executed again, and this time $\{a, b\}$, $\{a, c\}$, and $\{b, c\}$ are added to B, with *maxCard* incremented to 2.

So far, the actions taken are exactly as in Carma. However, after t_3 is read, DCF deviates from Carma because B is now full. Step (6) adds $\{a, d\}$ to B to fill up B, and Step (10) then inserts itemsets $\{b, d\}$, $\{c, d\}$, and $\{a, b, c\}$ into the waiting list *WL*.

Next, after t_4 is read, because items e and f appear for the first time, Step (12) is executed. Suppose itemsets $\{b, c\}$ and $\{a, d\}$ that are originally in B are picked to be moved to *WL*. Then, Step (14) inserts $\{e\}$ and $\{f\}$ into B, displacing $\{b, c\}$ and $\{a, d\}$.

Eventually, the first scan of *TDB* is completed, and all the itemsets and their counted values (i.e., *count*(v), *maxMissed*(v), etc.) are moved to the file *DL* (i.e., the *"done list"*). Then, in Step (20), the buffer pool is reloaded with itemsets from the *WL*. Suppose they are all of size 2 (e.g., $\{b, c\}$, $\{a, d\}$, etc.), in which case both *minCard* and *maxCard* are set to 2. Thus, in the next iteration of the repeat-loop, only subsets of size 3 of t_i are generated in Step (8). In this case, all those subsets that meet the condition in Step (9) are inserted into *WL* in Step (10).

6.2 Optimizations of DCF

The pseudocode given in Figure 4 omits many details for simplicity; here, we discuss some of the more important ones. In Figure 4, whenever an itemset is inserted into either the buffer pool or the waiting list, we need to check for duplication. This could be inefficient if duplication is checked every time against the itemsets contained in the file *WL*. A more efficient way is to delay the duplication check until the reloading conducted in Step (20). In this way, the file *WL* can operate completely in an *append mode*, as the ordering of itemsets of the same size in *WL* is immaterial to DCF.

In Step (13), after the itemset u has been identified and replaced, *maxCard* may need to be decremented if u happens to be the last itemset of size *maxCard*. An efficient implementation of this step is to keep track of the numbers of itemsets in the buffer pool, corresponding to sizes *minCard*, *minCard* $+ 1, \ldots,$ *maxCard*. Step (13) then decrements the number for *maxCard*; when this number drops to 0, the value of *maxCard* is decremented.

So far, we have focused our attention only on Phase I of DCF. For the sake of completeness, we briefly consider the situation for Phase II. Note that Phase II of Carma is a straightforward rescanning of the transaction database *TDB* to obtain exact supports for all the itemsets in the support lattice V. It is somewhat optimized by using *firstTrans*(v); this is because as far as v is concerned, it is

```
/* Invoking condition: When there is a tightening change */
Procedure DCF-tighten (new constraint $C_{new}$, buffer pool $B$, waiting list $WL$) {
    /* No immediate rechecking on itemsets $v$ in $DL$; delay until Phase II */
    /* Immediate rechecking on itemsets $v$ in $B$ */
    for all $v \in B$ {
        if ($v \not\models C_{new}$) {
            delete $v$ from $B$;
            replace $v$ with a new itemset $u \in WL$ where $u \models C_{new}$;
        } /* end if */
    } /* end for-all */
    /* No immediate rechecking on $v$ in $WL$; delay until $v$ is loaded into buffer */
}
```

Fig. 5. Procedure DCF-tighten (for a tightening change).

sufficient to rescan *TDB* up to the transaction prior to *firstTrans*(v). In DCF, Phase II is identical to that of Carma, except that when the buffer space is limited, it is worth ignoring *firstTrans*(v), and simply applying the full rescan for all v. What we gain back is that we no longer need to keep the counter *firstTrans*(v) for each itemset. This would reduce the number of counters from 3 to 2, for each itemset. In other words, for a fixed size buffer pool, this would translate to about 50% increase in the number of itemsets being processed in each batch in both Phase I and Phase II, which, in turn, translates to a reduced number of scans.

6.3 Handling Dynamic Changes to Nonfrequency Constraints with Limited Buffer Space

In the remainder of this section, we turn our attention to how DCF handles dynamic constraint changes when buffer space is limited. Recall from our earlier discussion that DCF constructs the solution space as it scans the transaction database *TDB* batch-by-batch. When a tightening change is requested by the user, the main algorithm of DCF is suspended, and the *procedure DCF-tighten* is invoked. DCF-tighten needs to (i) recheck results obtained from previous batches, and to (ii) enforce the new constraint for subsequent computations. Figure 5 gives the pseudocode to show how DCF-tighten handles a tightening change. The highlights of DCF-tighten are as follows:

—For each itemset v in the file *DL* (i.e., itemsets already processed), if v no longer satisfies the new constraint C_{new} (i.e., $v \models C_{old}$ but $v \not\models C_{new}$), then v can be deleted. However, there may be numerous changes to the constraints during the mining process, DCF-tighten delays this verification step until v is loaded for Phase II, so as to reduce I/O.

—For each itemset v in the buffer pool B, DCF-tighten checks whether $v \models C_{new}$. This immediate enforcement can stop any future effort spent on itemsets that violate the new constraint.

—For itemsets that have not been processed, they are divided into two groups: (i) those that are in file *WL*, and (ii) those that have not even been generated and put into *WL*. For the former group, again to reduce I/O, DCF-tighten delays the verification step until v is loaded into buffer. For the latter group,

```
/* Invoking condition: When there is a relaxing change */
/* Prerequisite: Buffer pool $B$ empty on entry */
Procedure DCF-relax (new/relaxed constraint $C_{new}$, old/original constraint $C_{old}$, set of new
constraint $\mathcal{C}_{new}$, maximum cardinality $maxCard$, transaction database $TDB$) {
(1)   if ($C_{old}$ not succinct) { /* $C_{new}$ not succinct as well */
(2)     for all $k$ from 1 to $i$ { /* start scanning $TDB$ */
(3)       for all $v \in B$ with $v \subseteq t_k^{SSM}$ { increment $count(v)$; }
(4)       for all $v$ generated by $MGF_{\mathcal{SC}_{new}}(t_k^{SSM})$
          where ($v \models (\mathcal{C}_{new} - \mathcal{SC}_{new})$ and $1 \leq |v| \leq maxCard$) {
(5)         insert $v$ into $B$ as in Carma;
          }
(6)       for all $v$ generated by $MGF_{\mathcal{SC}_{new}}(t_k^{SSM})$
          where ($v \models (\mathcal{C}_{new} - \mathcal{SC}_{new})$ and $|v| = maxCard + 1$) {
(7)         put $v$ in $WL$;
          }
(8)       prune the itemsets in $B$ exactly as in Carma;
        } /* end for-all: scanning completed */
(9)     for all $v \in B$ { move $v$ and its counters to $DL$; }
(10)  } else { /* $C_{old}$ and $C_{new}$ succinct */
(11)    for all $k$ from 1 to $i$ { /* start scanning $TDB$ */
(12)      for all $v \in B$ with $v \subseteq t_k^{SSM}$ { increment $count(v)$; }
(13)      for all $v$ generated by $MGF_{C_{new}/C_{old}}(t_k^{SSM})$
          where ($v \models (\mathcal{C}_{new} - \mathcal{SC}_{new})$ and $1 \leq |v| \leq maxCard$) {
(14)        insert $v$ into $B$ as in Carma;
          }
(15)      for all $v$ generated by $MGF_{C_{new}/C_{old}}(t_k^{SSM})$
          where ($v \models (\mathcal{C}_{new} - \mathcal{SC}_{new})$ and $|v| = maxCard + 1$) {
(16)        put $v$ in $WL$;
          }
(17)      prune the itemsets in $B$ exactly as in Carma;
        } /* end for-all: scanning completed */
(18)    for all $v \in B$ { move $v$ and its counters to $DL$; }
      } /* end else */
}
```

Fig. 6. Procedure DCF-relax (for a relaxing change).

all DCF needs to do is to enforce the new constraint in Step (8), when the itemset is being generated.

Next, we turn our attention to a relaxing change. Recall that a relaxing change has different, and tougher, computational requirements than a tightening change. When a relaxing change to a constraint is requested while the main algorithm of DCF is processing a certain batch of itemsets with a specific *maxCard* value, the *procedure DCF-relax* is invoked. Figure 6 gives the pseudocode on how DCF-relax handles a relaxing change. The highlights of DCF-relax are as follows:

—To ensure that the *DL* file is updated with respect to the new constraint C_{new}, DCF-relax must insert all those v's satisfying C_{new} but not C_{old} into *DL*. To do this efficiently, the procedure DCF-relax restricts the generate-and-test process to those v's such that $v \subseteq t_k^{SSM}$ and $|v| \leq maxCard$. Similarly, the *WL* file is updated in accordance with $\mathcal{C}_{new}$, as carried out in Steps (6) and (7) of

Figure 6. Notice that, to simplify our presentation in Figure 6, we assume that all these new itemsets to be counted (i.e., cardinalities between 1 and *maxCard* inclusive) can all fit into the buffer pool. If this is not true, this can be easily dealt with as shown in Figure 4; we omit the details here.

—As in DCF, a relaxing change can be further optimized by using a delta MGF, provided that the constraint being relaxed is succinct. This explains Steps (13) and (15) in Figure 6.

6.4 Analysis of DCF

As discussed so far, DCF is designed to handle situations where the buffer space is limited, and is optimized for its efficiency. To show the efficiency and correctness of DCF, we establish some formal assurances on how DCF operates in limited buffer space.

We begin with performance issues. Unlike Carma, our DCF algorithm is capable of operating in situations where there is limited buffer space. In those situations, it is obvious from the repeat-loop in Figure 4 that multiple scans of the transaction database *TDB* may be required. The exact number of scans is inversely proportional to the size of the buffer pool B. However, the following lemma guarantees that when B is large enough to accommodate the support lattice V constructed by Carma, then DCF—just like Carma—can complete Phase I in just one scan. In the sequel, we use *VS* to denote the set of all itemsets generated and counted by DCF. Hence, $|V|$ and $|VS|$ denote the number of itemsets in V and *VS*, respectively. Moreover, we use $|B|$ to denote the size of the buffer pool, in terms of the number of itemsets (and their associated counters) the buffer pool can accommodate.

LEMMA 6.3. *Let V denote the support lattice constructed by Carma, and let VS denote the set of all itemsets generated and counted by DCF. Then, $|B|$, $|V|$, and $|VS|$ are the sizes of B (buffer pool), V, and VS, respectively.*

(a) *When $|B| < |V|$, Phase I of DCF requires no more than $s = \lceil |VS|/|B| \rceil$ scans of TDB.*

(b) *When $|B| \geq |V|$, we have $VS = V$, and Phase I of DCF requires exactly one scan of TDB.*

PROOF. For part (a), when the buffer pool is full, Steps (8) to (14) of Figure 4 may be executed. Combined with the reloading conducted in Steps (19) to (21), it is clear that each itemset $v \in VS$ is counted exactly once. Thus, the number of scans required is $s = \lceil |VS|/|B| \rceil$. On the other hand, for part (b), when the buffer pool is large enough, Step (6) is executed repeatedly, which is basically running Carma. Thus, *VS* is identical to V, and the number of scan required is reduced to 1. □

In general, depending on the size of the buffer pool, *VS* can be a strict superset of V. This is a consequence of the tradeoff discussed in Section 6.1, when the Insertion Condition in Section 2.3 is modified in DCF by adding $w \in B$, as in Steps (9) and (11) of Figure 4. In the event that the buffer pool is large enough, part (b) of the above lemma guarantees equality between V and *VS*.

Next, we consider correctness issues of DCF. The first issue is to formalize the structured-ness of DCF. Lemma 6.4 establishes the nondecreasing nature of *minCard* and *maxCard* values at the end of each iteration. Lemma 6.4 states that DCF generates and counts itemsets $v \in VS$ in a levelwise fashion. It guarantees that if k is the least size of any itemset processed in the current batch, then no itemset of size smaller than k will be processed in future iterations.

LEMMA 6.4. *Let minCard and maxCard, respectively, denote the minimum and the maximum cardinalities of the itemsets that are in the buffer pool. From one iteration of DCF to the next, the minCard and maxCard values at the end of each iteration are nondecreasing.*

PROOF. We can show this lemma by induction. Let $minCard_j$ and $maxCard_j$ denote the *minCard* and *maxCard* values at the end of the jth iteration. For the base case, $minCard_1 = 0$ and $maxCard_1$ gets stabilized to some value $c > 0$ at the end of the first iteration. In preparation for the second iteration, v's are loaded into buffer B in Step (20) of Figure 4. The cardinalities of these v's are at least c, that is, $minCard_2 \geq c$. Otherwise, they would have been processed in the previous batch because of Step (11). Hence, we have $minCard_1 = 0 < c \leq minCard_2$ and $maxCard_1 = c \leq minCard_2 \leq maxCard_2$. This completes the base case.

For the inductive case, suppose $minCard_j \leq minCard_{j+1}$ and $maxCard_j \leq maxCard_{j+1}$. Then, in preparation for the $(j+2)$th iteration, v's are loaded into buffer B in Step (20). The cardinalities of these v's are at least the value of $maxCard_{j+1}$, that is, $minCard_{j+2} \geq maxCard_{j+1}$. Otherwise, they would have been processed in the previous batch because of Step (11). Hence, we have $minCard_{j+1} \leq maxCard_{j+1} \leq minCard_{j+2}$ and $maxCard_{j+1} \leq minCard_{j+2} \leq maxCard_{j+2}$. This completes the inductive case. □

LEMMA 6.5. *Let VS denote the set of all itemsets generated and counted by DCF. Suppose $|B| < |V|$ (i.e., the size of buffer pool B is smaller than the size of the support lattice V constructed by Carma). Then, at the end of each batch of itemsets in B (as constructed by DCF), we have the following, with respect to the corresponding minCard and maxCard values (i.e., the minimum and the maximum cardinalities of the itemsets that are in B):*

(*a*) *For any $v \in VS$ such that $|v| <$ minCard, v was processed (i.e., count(v) and maxMissed(v) computed) in a previous batch.*

(*b*) *For any $v \in VS$ such that minCard $\leq |v| <$ maxCard, v either was processed in a previous batch, or is in the current batch.*

PROOF. Having established in Lemma 6.4 the nondecreasing nature of *minCard* and *maxCard* values at the end of each iteration, we can show Lemma 6.5 by induction. For the base case, part (a) is trivially true. For part (b), because the buffer pool is full, Steps (8) to (14) of Figure 4 are executed at some point during the first iteration. For any to-be-inserted itemset v whose cardinality is strictly less than *maxCard*, there is always room for v—at the expense of another itemset u with cardinality exactly *maxCard*. Thus, when the first scan of *TDB* is completed, all itemsets with cardinality strictly less than *maxCard* are in the current batch. This completes the base case.

For the inductive case, we assume that parts (a) and (b) are true for the j-iteration. Then, for the $(j + 1)$th iteration, we know from Lemma 6.4 that *minCard* and *maxCard* are nondecreasing, that is, $minCard_j \le minCard_{j+1}$ and $maxCard_j \le maxCard_{j+1}$. We also know that $maxCard_j \le minCard_{j+1}$. If $minCard_j = minCard_{j+1}$, then part (a) is true because of the induction assumption in part (a). On the other hand, if $minCard_j < minCard_{j+1}$, then there are two cases. Under the first case where $minCard_{j+1} = maxCard_j$, then part (a) is proved because of the induction assumptions in parts (a) and (b). Under the second case where $minCard_{j+1} > maxCard_j$, this is only possible if all the itemsets v of cardinality less than $minCard_{j+1}$ has been processed. This completes the proof of part (a).

As for part (b), if $maxCard_j = maxCard_{j+1}$, then part (b) is trivially true. On the other hand, if $maxCard_j < maxCard_{j+1}$, then there are two cases. Under the first case where $minCard_{j+1} = maxCard_j$, then an itemset v with $minCard_{j+1} \le |v| < maxCard_{j+1}$ was processed in a previous batch or is in the current batch, because of Steps (9), (10), and (20). Under the second case where $minCard_{j+1} > maxCard_j$, v is in the current batch, because of Steps (11) to (14). This completes the proof of part (b). □

Recall that the main purpose of Carma is that it handles dynamic changes to the support threshold during Phase I. As formalized in Theorem 2.6, its correctness is mainly guaranteed by the fact that the constructed lattice V is typically a superset of all the frequent sets with respect to the support threshold $g(\langle\sigma_1, \ldots, \sigma_n\rangle)$. For DCF, a similar goal is to show that DCF can handle dynamic changes to the support threshold—even in the absence of sufficient memory to contain the lattice V. However, to do so, there is a slight complication due to the possibility of multiple scans of *TDB* (cf. Lemma 6.3) when there is not enough buffer space. For Carma, the sequence of the support threshold is denoted as $\langle\sigma_1, \ldots, \sigma_n\rangle$, where σ_i is the support threshold after transaction t_i is read. For DCF, because there could be $s \ge 1$ scans, the corresponding sequence is extended to $\langle\sigma_1, \ldots, \sigma_n, \sigma_{n+1}, \ldots, \sigma_{2n}, \ldots, \sigma_{sn}\rangle$. The following theorem shows the correctness of DCF.

THEOREM 6.6. *At the end of Phase I of the DCF algorithm, VS (i.e., the set of all itemsets generated and counted by DCF) is typically a superset of all frequent itemsets relative to the support threshold given by max* $\{g(\langle\sigma_{(j-1)n+1}, \ldots, \sigma_{jn}\rangle) \mid 1 \le j \le s\}$, *for the same function* $g()$ *as in Theorem* 2.6.

PROOF. For the jth batch of itemsets processed by DCF (denoted as VS_j), the previous lemma guarantees that all itemsets $v \in VS$ with cardinality strictly less than $minCard_j$ have been processed. Then, by Theorem 2.6, we can conclude that the set $\{v \mid v \in VS \; \& \; |v| < minCard_j\}$ is a superset of the set of all frequent itemsets with cardinality less than $minCard_j$ relative to the support threshold $g(\langle\sigma_{(j-1)n+1}, \ldots, \sigma_{jn}\rangle)$ and the maximum threshold $max\{g(\langle\sigma_{(j-1)n+1}, \ldots, \sigma_{jn}\rangle) \mid 1 \le j \le s\}$. By taking the union of all the s batches (i.e., $VS = VS_1 \cup \cdots \cup VS_s$), VS typically contains all the frequent itemsets relative to the maximum support threshold. □

Recall from Section 2.3 that, in the presence of changes in the support threshold, we would finally like to find all frequent sets with respect to the last support threshold of the last scan (i.e., σ_{sn}). Just like $g(\langle\sigma_1, \ldots, \sigma_n\rangle) \geq \sigma_n$ in Carma's case, a similar condition holds here: In general, $max\ \{g(\langle\sigma_{(j-1)n+1}, \ldots, \sigma_{jn}\rangle) \mid 1 \leq j \leq s\} \geq \sigma_{sn}$. In the event that the last support threshold σ_{sn} is strictly smaller, the heuristic proposed by Hidber [1999] can be used—in exactly the same way as was done to overcome Carma's situation when $g(\langle\sigma_1, \ldots, \sigma_n\rangle) > \sigma_n$.

7. DISCUSSION: OTHER APPLICATIONS OF THE SSM

Recall from Sections 3 and 4.2 that the segment support map (SSM) enhances the performance of our proposed DCF algorithm by (i) tightening the *maxSupport*(v) and *maxMissed*(v) bounds for pruning, and (ii) better exploiting the frequency constraint as well as other antimonotone constraints. However, it is important to note that, being a generic structure, the SSM can bring benefits not only to constrained online/dynamic mining algorithms like DCF, but also to a general class of offline/nondynamic mining algorithms (constrained or otherwise). Below, we present some examples of these useful applications. Experimental results in Section 8.4 show the effectiveness of the SSM in one of these applications.

7.1 Using the SSM in Hash-Based Frequent-Set Mining

Since its introduction, the association rule mining problem (and the frequent-set finding problem) has been the subject of numerous studies. As a result, many algorithms for finding association rules or frequent sets have been developed. One of them is the *DHP algorithm* [Park et al. 1997], which uses a hash-based partitioning technique to speed up the performance of frequent-set mining. Recall from Section 2.1 that the DHP algorithm hashes candidate k-itemsets (e.g., $k = 2$) into different buckets. If an insufficient number of itemsets is hashed into a bucket (i.e., the bucket count is below the user-defined support threshold), all candidate itemsets in this bucket are pruned before counting their support.

In the presence of the SSM, the performance of the DHP algorithm can be enhanced as follows. When an SSM is used simultaneously with hash tables built by DHP, known infrequent k-itemsets do not need to be generated at all. Itemsets that pass through the pruning by the SSM can now be further pruned by the DHP algorithm. See Figure 7 for a skeleton of the *SSM-enhanced DHP algorithm*. To simplify our presentation, this skeleton omits some details that are immaterial to our discussion.

As shown in Step (1) of Figure 7, the SSM-enhanced DHP algorithm can easily obtain from the SSM those 1-itemsets whose support values satisfy the user support threshold, thereby avoiding the generate-and-test process of candidate 1-itemsets. Regarding the building of hash tables for k-itemsets (i.e., Steps (2) and (9)), the SSM-enhanced DHP algorithm avoids hashing any k-itemset v that is known to be infrequent. This helps to reduce the number of "redundant" (i.e., known infrequent) itemsets being hashed into a bucket. Consequently, this effectively lowers the number of candidates to be counted. Moreover, in Steps (3)

```
Algorithm SSM-enhanced DHP (transaction database $TDB = \{t_1, ..., t_n\}$){
(1)   obtain the set of frequent 1-itemsets from the SSM
      by picking those 1-itemsets $v$ whose $support(v) = estsup(v) \geq$ threshold;
      /* Step (2): Build a hash table for 2-itemsets $v \subseteq t_i^{SSM}$; use $t_i^{SSM}$ (instead of $t_i$) */
(2)   for all 2-itemsets $v \subseteq t_i^{SSM}$ {
        if $v$ not known to be infrequent (from the SSM),
        then hash $v$ into an appropriate bucket, and increment the corresponding bucket count;
      }
      /* Step (3): Use the SSM in candidate generation to ensure that known infrequent itemsets
      are not generated */
(3)   generate candidate 2-itemsets using the SSM and the set of frequent 1-itemsets;
(4)   remove those candidate 2-itemsets in a bucket whose count < threshold;
(5)   $k = 2$;
(6)   while (there exists a candidate $k$-itemset) {
(7)     for all $t_i^{SSM}$ {
(8)       count candidate $k$-itemsets $v \subseteq t_i^{SSM}$;
(9)       build a hash table for $(k + 1)$-itemsets $v \subseteq t_i^{SSM}$;
        }
(10)    obtain the set of frequent $k$-itemsets by picking those candidate $k$-itemsets whose
        counts $\geq$ threshold;
(11)    generate candidate $(k + 1)$-itemsets using the SSM and the set of frequent $k$-itemsets;
(12)    remove those candidate $(k + 1)$-itemsets in a bucket whose count < threshold;
(13)    $k = k + 1$;
      } /* end while */
}
```

Fig. 7. The SSM-enhanced DHP algorithm.

and (11), when the SSM-enhanced DHP algorithm generates candidate $(k + 1)$-itemsets, it uses both the SSM and the set of frequent k-itemsets to ensure that known infrequent $(k + 1)$-itemsets are not generated. Consequently, fewer candidate itemsets are hashed into buckets, thereby reducing the number of candidates that need to be counted and speeding up the computation.

7.2 Using the SSM in Depth-First Search Based Frequent-Set Mining

In the previous section, we showed how the SSM brings additional benefits to a hash-based frequent-set mining algorithm (namely, DHP). In this section, we show how the SSM can bring additional benefits to another frequent-set mining algorithm, namely a depth-first search based algorithm called *DepthProject* [Agarwal et al. 2000].

While many existing frequent-set mining algorithms use a breadth-first search approach (i.e., a levelwise bottom-up approach), DepthProject uses a depth-first approach. More precisely, DepthProject generates frequent itemsets by using a depth-first search on a lexicographic tree of itemsets. At each recursion, the algorithm generates possible frequent lexicographic extensions (i.e., candidates) of a tree node v and tests for frequency.

Like its applications within the DHP algorithm in Section 7.1, the SSM can enhance the performance of the DepthProject algorithm. When an SSM is used simultaneously with DepthProject, known infrequent candidates can be pruned before the frequency counting. See Figure 8 for a skeleton of the *SSM-enhanced*

```
Algorithm SSM-enhanced DepthProject (non-root itemset node $v$, transaction database $TDB$) {
(1)  generate the set of candidates—which contains all possible frequent lexicographic
     extensions of $v$—with the SSM so that known infrequent candidates are not generated in
     the first place;
(2)  obtain frequent lexicographic extensions of $v$ (i.e., $E(v) = \{item_1, ..., item_{|E(v)|}\}$) by
     counting the support of candidates in $TDB$;
(3)  for $k$ from 1 to $|E(v)|$ {
(4)      call SSM-enhanced DepthProject ($v \cup \{item_k\}$, projected $TDB$);
     }
}
```

Fig. 8. The SSM-enhanced DepthProject algorithm.

DepthProject algorithm. Again, to simplify our presentation, this skeleton omits some details that are immaterial to our discussion.

The SSM-enhanced DepthProject algorithm can directly obtain the set of frequent 1-itemsets from the SSM, thereby avoiding the generate-and-test process of candidate 1-itemsets (i.e., avoiding the support counting for all domain items). Once these frequent 1-itemsets (i.e., frequent lexicographic extensions of the root node) are found, their frequent lexicographic extensions can be generated by calling the algorithm recursively. In each recursive call, the algorithm generates candidates (i.e., possible frequent lexicographic extensions of an itemset v) with the SSM so as to ensure that known infrequent lexicographic extensions of v are not generated (refer to Step (1) of Figure 8). This helps to reduce the number of candidates that need to be counted in Step (2), and helps to prevent unnecessary computation on known infrequent itemsets.

7.3 Using the SSM in Sequential Mining

To show that the benefits of the SSM are not confined to the mining of frequent sets, we show in this section how the SSM can be used to enhance the performance of *sequential mining algorithms*. The problem of sequential mining [Agrawal and Srikant 1995] can be divided into the following five phases:

I. *Sort Phase*, in which the original transaction database is converted into a database of customer sequences. In other words, transactions in the original database are sorted by customer ID and transaction time.

II. *Frequent Itemset Phase*, in which classical association rule mining algorithms (e.g., Apriori [Agrawal and Srikant 1994]) are applied to find (i) all frequent itemsets, and (ii) all frequent sequences of length 1 (i.e., *frequent 1-sequences*).

III. *Transformation Phase*, in which each transaction t_i is replaced by the set of all frequent itemsets (found in the Frequent Itemset Phase) that are contained in t_i.

IV. *Sequence Phase*, in which algorithms such as AprioriAll [Agrawal and Srikant 1995] are applied to find the desired sequences by using the set of frequent itemsets found in the Frequent Itemset Phase.

V. *Maximal Phase*, which is an optional phase for finding maximal sequences. Among the frequent sequences found in the Sequence Phase, it removes those that are not maximal.

```
Algorithm SSM-enhanced Apriori (database of customer sequence TDB) {
(1)  obtain the set of frequent 1-itemsets from the SSM
     by picking those 1-itemsets $v$ whose $support(v) = estsup(v) \geq$ threshold;
(2)  $k$ = 1;
(3)  while (there exists a frequent $k$-itemset) {
       /* Step (4): Use the SSM in candidate generation to ensure that known infrequent itemsets
       are not generated */
(4)    generate candidate $(k + 1)$-itemsets using the SSM and the set of frequent $k$-itemsets;
(5)    for each customer sequence $cs$ in $TDB$ {
(6)      increment the count of all candidate $(k + 1)$-itemsets that are contained in $cs$;
       }
(7)    set of frequent $(k + 1)$-itemsets
       = { candidate $(k + 1)$-itemsets whose support $\geq$ threshold };
(8)    $k = k + 1$;
     } /* end while */
(9)  return $\bigcup_k$ set of frequent $k$-itemsets;
}
```

Fig. 9. The SSM-enhanced Apriori algorithm.

```
Algorithm SSM-enhanced AprioriAll (collection of transformed database $TDB'$) {
(1)  obtain the set of frequent 1-sequences from the SSM
     by picking those 1-sequences $v'$ whose $support(v') = estsup(v') \geq$ threshold;
(2)  $k$ = 1;
(3)  while (there exists a frequent $k$-sequence) {
       /* Step (4): Use the SSM in candidate generation to ensure that known infrequent sequences
       are not generated */
(4)    generate candidate $(k + 1)$-sequences using the SSM and the set of frequent $k$-sequences;
(5)    for each transformed customer sequence $tcs$ in $TDB'$ {
(6)      increment the count of all candidate $(k + 1)$-sequences that are contained in $tcs$;
       }
(7)    set of frequent $(k + 1)$-sequences
       = {candidate $(k + 1)$-sequences with support $\geq$ threshold};
(8)    $k = k + 1$;
     } /* end while */
(9)  return $\bigcup_k$ set of frequent $k$-sequences;
}
```

Fig. 10. The SSM-enhanced AprioriAll algorithm.

In addition to enhancing the performance of DHP and DepthProject, the SSM can also help to enhance the performance of sequential mining algorithms—especially in the Frequent Itemset Phase and the Sequence Phase—by reducing the number of candidate itemsets and the number of candidate sequences. More concretely, in the Frequent Itemset Phase, when generating candidate $(k+1)$-itemsets from the set of frequent k-itemsets, the SSM-enhanced Apriori algorithm uses the SSM to ensure that those candidate $(k + 1)$-itemsets that are known to be infrequent are not generated (see Step (4) of Figure 9). This reduces the number of candidates that require support counting in Step (6), and thereby speeding up the computation.

Similar comments apply to the Sequence Phase. When the SSM-enhanced AprioriAll algorithm generates candidate $(k + 1)$-sequences in Step (4) of Figure 10, it uses the SSM to ensure that those candidate $(k + 1)$-sequences that are known to be infrequent are not generated.

8. EXPERIMENTAL RESULTS

The experimental results cited below are based on a transaction database *TDB* of 100k records, and a domain of 10k items. *TDB* was generated by the program developed at the IBM Almaden Research Center [Agrawal and Srikant 1994]. The average transaction length is 10 items, and the average cardinality of a frequent itemset is 4. Unless otherwise specified, we used a support threshold of 0.1%. All experiments were run in a time-sharing environment using a 700 MHz machine. The speedup shown is with respect to the total CPU and I/O time.

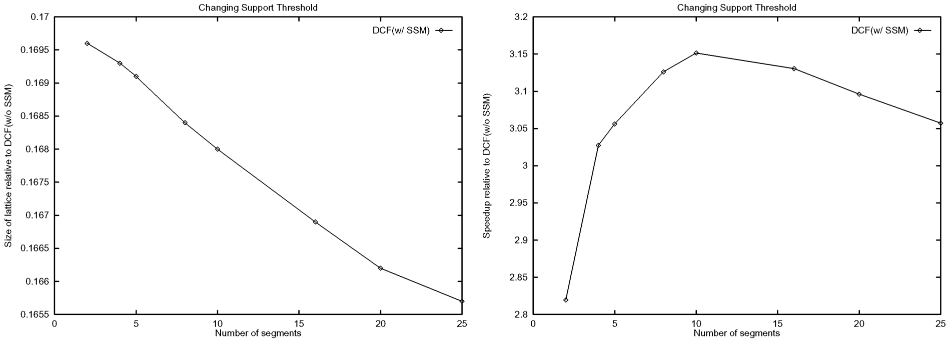
8.1 SSM-Based Pruning within DCF

In this experiment, we compared the results for two algorithms (implemented in C) that support the kinds of succinctness-based pruning shown in Section 4: *DCF(w/o SSM)* and *DCF(w/SSM)*. The former does not include the SSM, whereas the latter does. The difference is to highlight the effectiveness of SSM-based pruning.

In this section, we first showed the results of the SSM-based pruning based on a constant and a changing support thresholds. More specifically, we evaluated how the number of segments can benefit DCF(w/SSM), with a CFQ consisting of the succinct constraint $C \equiv max(S.Price) \leq 10$. We conducted two sets of experiments where 40% of items with $Price \leq 10$: (i) one experiment with a fixed support threshold of 0.1%, and (ii) another with support threshold varied from 0.075% to 0.125% then to 0.1%. These values were chosen to correspond to a similar set of experiments shown for Carma [Hidber 1999]. The results of these two sets of experiments turned out to be almost the same. For lack of space, we only show the results based on the changing support threshold.

The x-axis in Figure 11 shows the number of segments varying from 2 to 25. The y-axis in Figure 11(a) shows the size of the lattice computed by DCF(w/SSM) relative to that by DCF(w/o SSM). The size of the lattice corresponds to the number of itemsets that were counted. The smaller the size, the more effective the pruning was. As expected, the larger the number of segments in the SSM, the larger was the number of itemsets that were pruned. For instance, with 10 segments, the number of itemsets counted was about 1/6 of that required without the SSM.

In Figure 11(b), the y-axis gives the speedup of DCF(w/SSM) relative to DCF(w/o SSM) in terms of total runtime. With increasing number of segments, while the relative size of the lattice decreases monotonically in Figure 11(a), the relative speedup shows a peak when the number of segments is 10 in Figure 11(b). The peak occurs when the reduction in the lattice size is no longer significant enough to offset the cost of processing an extra segment. The result shows that while the SSM is a lightweight structure (e.g., 10 segments requiring 100,000 integers for 10k items, for a total space of 0.2 megabytes using 2 bytes per integer), the pruning effect is spectacular. In absolute terms, DCF(w/o SSM) took about 10 seconds total time, whereas DCF(w/SSM) took less than 3 seconds with 10 segments. This is very encouraging because this shows that we are making significant progress towards the eventual goal of providing a real-time response (or a real-time completion of CFQ evaluation).



(a) Relative Size of Lattice

(b) Relative Speedup

Fig. 11. SSM-based pruning: Changing support threshold.

Next, we showed the results of the SSM-based pruning based on a "seasonal" transaction database. More specifically, in Section 3, we mentioned that the SSM is well suited to handle transaction databases that do not follow the uniform distribution assumption, and there are many databases of this kind. An example is the supermarket database consisting of "seasonal" transactions. By using the IBM Almaden program, we generated a transaction database in which the effect of the "seasonal" nature has been simulated. Specifically, 50% of the items have a higher probability of appearing in the first half of the database, and the other 50% have a higher probability of appearing in the second half. The following table shows the speedup of DCF(w/SSM) relative to that of DCF(w/o SSM).

Number of Segments	2	4	5	8	10	16	20	25
Relative Speedup	5.81	5.96	5.99	6.04	6.06	6.03	6.00	5.96

For the "seasonal" data, the average speedup is around 6 times, as opposed to around 3 times as shown in Figure 11(b). This shows that the SSM, mainly via the $future_k(v)$ bound, delivers additional benefits when the transaction database is not uniformly distributed and is "seasonal" in nature.

8.2 Succinctness-Based Optimization

Next, we turned our attention to succinctness-based optimization. This experiment was a continuation of the one in Section 8.1, having the CFQ consisting of the succinct constraint $C \equiv max(S.Price) \leq 10$. The exceptions are that we set the number of segments to 10, and we varied the percentage *pct* of items whose *Price* is at 10 or below. In the experiment, we compared the results for *DCF(w/o SSM)* and *DCF(w/SSM)* with that for *Carma+*. In Carma+, we first ran Carma to deal with the support threshold, and then checked all the constraints at the end. The difference between Carma+ and the two DCF algorithms is to highlight the effectiveness of the succinctness-based optimization described in Section 4.1.

The x-axis in Figure 12 shows *pct* varying from 40% to 100%. The y-axis shows, in logarithmic scale, the speedup of DCF(w/SSM) and DCF(w/o SSM)

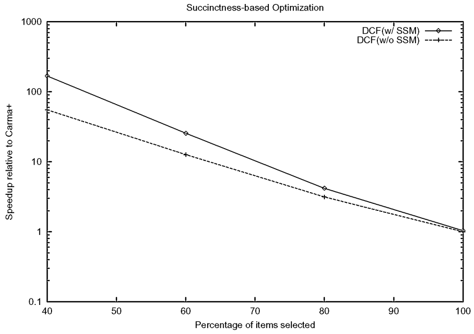


Fig. 12. Succinctness-based optimization: Relative speedup.

against Carma+. Take $pct = 40\%$ as an example. The two DCF algorithms, by exploiting the succinct constraint, achieve a speedup of about two orders of magnitude. The results here convincingly show that succinctness-based optimization is effective.

8.3 Effectiveness of Dynamic Constraint Changes

Then, we turned our attention to dynamic changes of constraints. In particular, we focused on relaxing changes, which are usually computationally intensive. In this experiment, we compared the results for three algorithms that were implemented in C:

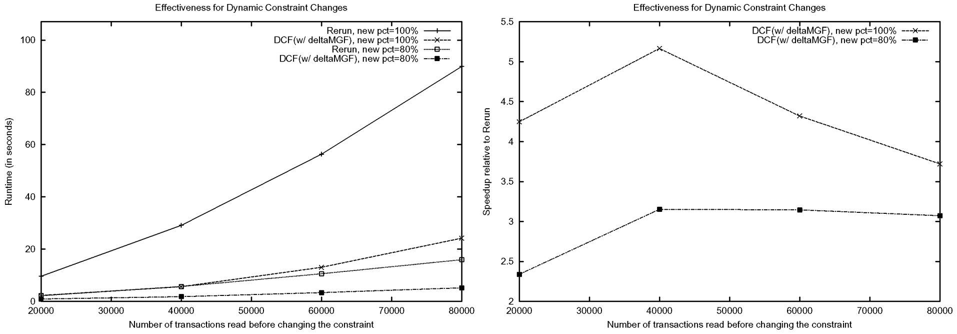
—*DCF(w/deltaMGF)*, which deploys the delta member generation functions in generating those itemsets v's satisfying C_{new} but not C_{old}. Note that to evaluate directly the efficiency of handling the dynamic changes, we only consider here the runtime for "fixing" the current lattice V. More specifically, suppose C_{old} is relaxed to C_{new} after i transactions have been read. Then, we are only concerned here with the runtime to change the $V(C_{old}, i)$ to $V(C_{new}, i)$, where $V(C, i)$ denotes the state of the lattice for constraint C after the first i transactions have been processed (from scratch).

—*DCF(w/o deltaMGF)*, which adopts the generate-and-test approach described in Section 5.2 (i.e., without using the delta MGFs).

—*Rerun*, which directly computes $V(C_{new}, i)$ from scratch, instead of "evolving" it from $V(C_{old}, i)$ like the two algorithms above.

In this experiment, the CFQ consists of $C_{old} \equiv max(S.Price) \leq 8$, which was relaxed to $C_{new} \equiv max(S.Price) \leq 10$. Note that both C_{old} and C_{new} are succinct constraints. The percentage pct_{new} of items having $Price \leq 10$ and the percentage pct_{old} of items having $Price \leq 8$ are set in such a way that $pct_{old} = pct_{new} - 20\%$. We varied pct_{new} from 80% to 100%.

In all experiments, DCF(w/deltaMGF) far dominates DCF(w/o deltaMGF); the former is often four orders of magnitude faster! This shows the importance of the delta member generating functions in dealing with dynamic changes to constraints.



(a) Runtime (b) Relative Speedup

Fig. 13. Effectiveness of dynamic constraint changes: DCF(w/deltaMGF) vs. Rerun.

To deal with dynamic changes to nonfrequency constraints, the approaches outlined in Section 5 are essentially incremental in nature. That is, based on the current state $V(C_{old}, i)$, DCF "evolves" it into the appropriate state $V(C_{new}, i)$. As with any incremental approaches, they may not always win against a simple rerun from scratch. The following experiment is to examine this issue in detail.

The comparison between Rerun and DCF(w/deltaMGF) depends on the time when the constraint C_{old} is relaxed to C_{new}. The x-axis in Figure 13 shows the number i of transactions read before relaxing the constraint, and i varied from 20k to 80k (for a 100k transaction database). The y-axis in Figure 13(a) shows the total runtime (in seconds) of both DCF(w/deltaMGF) and Rerun. From the graph, it is clear that DCF(w/deltaMGF) always beats Rerun, but the extent varies under different situations. When C_{new} selects more items (e.g., pct_{new} = 100%, when compared with pct_{new} = 80%), Rerun is more affected by it. Thus, the relative speedup between Rerun and DCF (w/deltaMGF) is higher for $pct_{new} = 100\%$. This is shown in Figure 13(b), which essentially translates the absolute runtime in Figure 13(a) to give the relative speedup in Figure 13(b). It is clear from Figure 13(b) that the incremental algorithm with delta MGFs is always a few times faster than Rerun.

8.4 SSM-Based Pruning within Other Applications

In Section 8.1, we reported experimental results that convincingly show the effectiveness of SSM-based pruning within the DCF algorithm. In this section, we turned our attention to SSM-based pruning within other applications. Recall from Section 7 that the benefits of the SSM are not confined to the DCF algorithm for the efficient dynamic mining of constrained frequent sets. The SSM can bring additional benefits to many other applications (e.g., the hash-based mining of frequent sets, the depth-first search based mining of frequent sets, and the mining of sequential patterns).

We evaluated the effectiveness of the SSM in providing additional benefits to an instance of these applications—namely, the DHP algorithm. Although we only evaluated one instance (due to the lack of space), we expect that the

SSM is also effective in providing pruning within other applications too. In the experiment, we compared the results of two algorithms (implemented in C):

—the original DHP algorithm, denoted as *DHP(w/o SSM)*, and
—the SSM-enhanced DHP algorithm, denoted as *DHP(w/SSM)*.

Among the two algorithms, the former does not include the SSM, whereas the latter does. The difference is to highlight the effectiveness of SSM-based pruning within DHP. In both algorithms, hash tables having 32,768 buckets are built for candidate 2-itemsets. In the experiment, we used a transaction database of 100k records and 10k domain items (where average transaction length is 4 items). We varied the number of segments from 2 to 25. The following table shows the results of DHP(w/SSM) relative to those of DHP(w/o SSM).

No. of Segments	Relative Speedup	No. of Candidate 2-itemsets
2	3.38	20,984
4	3.32	20,080
5	3.27	19,660
8	3.20	18,392
10	3.05	17,737
16	2.84	15,951
20	2.67	14,875
25	2.47	13,782

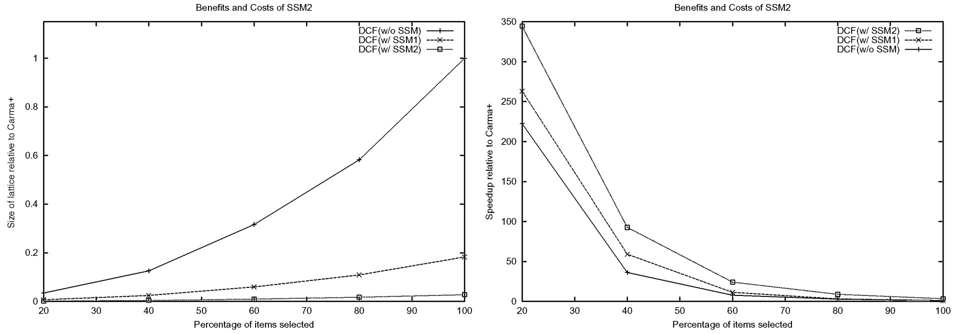
The above table shows that, when the DHP algorithm is used in conjunction with the SSM, the average speedup is around 3 times (when comparing the DHP algorithm with the SSM to that without the SSM). This indicates that, by reducing the number of candidate 2-itemsets, the SSM brings additional benefits to DHP. For example, when using 10 segments for the SSM, DHP(w/SSM) generates 17,737 candidate 2-itemsets, which are (i) about 83% of candidate 2-itemsets generated by DHP(w/o SSM), and (ii) about 1.7% of candidate 2-itemsets generated by Apriori.

It is interesting to note from the above table that, while the number of candidate 2-itemsets decreases when more segments are used in the SSM (as expected), the relative speedup decreases. The reason is that when the number of segments increases, the reduction in the number of candidate 2-itemsets is no longer sufficient to offset the cost of processing an extra segment.

In sum, this experimental result further reinforces the usefulness of the SSM—namely, the SSM brings additional benefits to other applications, including the DHP algorithm.

8.5 Benefits and Costs of the SSM of Higher Cardinalities

Finally, we evaluated the benefits and costs of using an SSM of higher cardinalities. So far in the experiments, we used SSMs that store the actual segment supports of singleton itemsets. However, recall from Section 3.1 that the upper bound *estsup*(v) provided by the SSM can be made tighter in two ways. The



(a) Relative Size of Lattice

(b) Relative Speedup

Fig. 14. Effectiveness of SSM2-based pruning.

first way is to increase the number of segments m. The amount of storage space required is then increased linearly. We experimented with varying the number of segments in Section 8.1.

The second way to generalize the SSM is to store not only the actual segment supports of singleton itemsets, but also the actual segment supports of k-itemsets where $k \geq 2$. A benefit of so doing is that the upper bound on the support of an itemset v can be tightened by using the actual segment supports based on 2-itemsets than using those based on singleton itemsets. However, the price is that the amount of storage space required is then increased *exponentially* with respect to the sizes of the itemsets. For instance, for a domain of 10k items, an SSM storing the actual segment supports of all singleton itemsets costs 0.2 megabytes when there are 10 segments in the SSM. However, the amount of space required jumps to 2 gigabytes just for storing the actual segment supports of all 2-itemsets! Even for a domain of 1k items, the amount of space required for 10 segments jumps from 20 kilobytes (for storing singleton itemsets) to 20 megabytes (for storing 2-itemsets).

The experiment was conducted on a transaction database of 10k records and a domain of 1k items, with average transaction length of 10 items. Like the experimental setup in Section 8.2, we used a CFQ consisting of the succinct constraint $C \equiv max(S.Price) \leq 10$, set the number of segments to 10, and varied the percentage *pct* of items whose price is at 10 or below. We compared the results for the following algorithms that were implemented in C:

- —*Carma*+, which first runs Carma to deal with the support threshold, and then checks all the constraints at the end (refer to Sections 4.1 and 8.2);
- —*DCF(w/o SSM)*, which does not include the SSM;
- —*DCF(w/SSM1)*, which includes the (usual) SSM storing the actual segment supports of singleton itemsets; and
- —*DCF(w/SSM2)*, which includes the SSM storing the actual segment supports of singleton itemsets and of 2-itemsets.

The x-axis in Figure 14 shows *pct* varying from 20% to 100%. The y-axis in Figure 14(a) shows the sizes of the lattice computed by the three DCF algorithms relative to that by Carma+; the y-axis in Figure 14(b) shows the speedups of DCF(w/o SSM), DCF(w/SSM1), and DCF(w/SSM2) against Carma+.

The experimental results served at least three purposes. First, the difference between DCF(w/SSM1) and DCF(w/SSM2) highlights the effectiveness of pruning based on an SSM that stores the actual segment supports of 2-itemsets (in addition to the supports of singleton itemsets). As shown in Figure 14(a), DCF(w/SSM2) further reduces the lattice size. Due to the reduction in lattice size, the speedup of DCF(w/SSM2) is higher than those of DCF(w/SSM1) as indicated in Figure 14(b). However, this comes with the price of an extra $20 - 0.02 = 19.98$ megabytes of storage space.

Second, the difference between DCF(w/o SSM) and the other two DCF algorithms—namely, DCF(w/SSM1) and DCF(w/SSM2)—shows the benefits brought by the SSM. As shown in Figure 14, both DCF(w/SSM1) and DCF(w/SSM2) lead to a smaller lattice size and higher speedup, when compared to DCF(w/o SSM).

Third, the difference between Carma+ and the three DCF algorithms once again highlights the effectiveness of the succinctness-based optimization described in Section 4.1. To elaborate, the lower the percentage of items selected (i.e., the fewer the items being selected), the smaller the relative lattice size and the higher the relative speedup.

9. CONCLUSIONS

Towards the development of a practical environment for the human-centered exploratory mining of constrained frequent sets, we considered in this article an important component—namely, how to support efficient dynamic mining. The DCF algorithm developed here has the capabilities of handling constrained frequent-set queries, and handling dynamic changes to the constraints (both relaxing and tightening) and/or the support threshold. It is also capable of operating in situations where there is not enough buffer space to simultaneously accommodate all frequent itemsets.

Functionality aside, a key contribution of this article is to optimize the performance of the DCF algorithm. To this end, we proposed and studied the novel structure of SSM. While very lightweight, the SSM provides direct information about the variability of support counts in different segments of the transaction database. Consequently, it helps to tighten the $maxSupport(v)$ and $maxMissed(v)$ bounds for pruning, and to better exploit the support constraint as well as other antimonotone constraints. Experimental results reported convincingly shows the benefits of the SSM. With a very small price to pay (e.g., 0.2 megabytes of space for 10,000 items and 10 segments), the SSM can prune a much larger number of itemsets, and can bring about a speedup that is (i) several times better than without using the SSM and (ii) about two orders of magnitude better than Carma+. Furthermore, it is important to note that all the results on the SSM reported here have serious implications not only to constrained dynamic/online mining, but also to a general class of offline mining algorithms (constrained or otherwise). This is because the notions of segment supports and SSM can apply even to the classical Apriori algorithm (for offline mining of unconstrained association rules), and hence to many of its variants for several related data mining tasks.

Central to the subject matter of this article is the efficiency of DCF in handling dynamic changes to nonfrequency constraints. To this end, we proposed and studied the notion of delta MGFs for a relaxing change, which is more computationally expensive than a tightening change. The experimental results reported here show that the delta MGFs are very effective in handling the changes.

A key overall objective of our project is to develop a practical environment for the human-centered exploratory mining of constrained frequent sets. For this environment to be truly exploratory, it is imperative that the response time of a system should be as little as possible. By that, we require the system not only to give continuous feedback, but also to allow the user to make dynamic changes; this aspect we have dealt with successfully in this article. Moreover, we also require the system to "complete" all processing for a CFQ in a short time. To this end, we are very encouraged to see that with the optimizations described here, it takes less than 3 seconds total time to complete the CFQ in Section 8.1. In ongoing work, we are interested in exploring how much sampling and parallel processing [Park et al. 1995; Agrawal and Shafer 1996] can help. We are also interested in extending the dynamic mining framework to handle other patterns, such as correlations [Brin et al. 1997; Grahne et al. 2000], quantitative rules [Miller and Yang 1997], and sequential patterns [Agrawal and Srikant 1995; Garofalakis et al. 1999].

REFERENCES

Agarwal, R. C., Aggarwal, C. C., and Prasad, V. V. V. 2000. Depth first generation of long patterns. In *Proceedings of the KDD 2000*. ACM, New York, 108–118.

Agarwal, R. C., Aggarwal, C. C., and Prasad, V. V. V. 2001. A tree projection algorithm for generation of frequent item sets. *J. Parall. Distrib. Comput. 61*, 3 (Mar.), 350–371.

Agrawal, R., Imielinski, T., and Swami, A. 1993. Mining association rules between sets of items in large databases. In *Proceedings of the SIGMOD 1993*. ACM, New York, 207–216.

Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., and Verkamo, A. I. 1996. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, Menlo Park, Calif., Chapter 12.

Agrawal, R. and Shafer, J. C. 1996. Parallel mining of association rules. *IEEE Trans. Knowledge and Data Engineering 8*, 6 (Dec.), 962–969.

Agrawal, R. and Srikant, R. 1994. Fast algorithms for mining association rules. In *Proceedings of the VLDB 1994*. Morgan-Kaufmann, Los Altos, Calif., 487–499.

Agrawal, R. and Srikant, R. 1995. Mining sequential patterns. In *Proceedings of the ICDE 1995*. IEEE Computer Society Press, Los Angeles, Calif., USA, 3–14.

Bayardo, Jr., R. J. 1998. Efficiently mining long patterns from databases. In *Proceedings of the SIGMOD 1998*. ACM, New York, 85–93.

Brin, S., Motwani, R., and Silverstein, C. 1997. Beyond market baskets: Generalizing association rules to correlations. In *Proceedings of the SIGMOD 1997*. ACM, New York, 265–276.

Fukuda, T., Morimoto, Y., Morishita, S., and Tokuyama, T. 1996. Data mining using two-dimensional optimized association rules: Scheme, algorithms, and visualization. In *Proceedings of the SIGMOD 1996*. ACM, New York, 13–23.

Garofalakis, M. N., Rastogi, R., and Shim, K. 1999. SPIRIT: Sequential pattern mining with regular expression constraints. In *Proceedings of the VLDB 1999*. Morgan-Kaufmann, Los Altos, Calif., 223–234.

Gibbons, P. B. and Matias, Y. 1998. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of the SIGMOD 1998*. ACM, New York, 331–342.

Grahne, G., Lakshmanan, L. V. S., and Wang, X. 2000. Efficient mining of constrained correlated sets. In *Proceedings of the ICDE 2000*. IEEE Computer Society Press, Los Angeles, Calif., 512–521.

Haas, P. J. and Hellerstein, J. M. 2001. Online query processing. In *Proceedings of the SIGMOD 2001*. ACM, New York, 623.

Han, J. and Fu, Y. 1995. Discovery of multiple-level association rules from large databases. In *Proceedings of the VLDB 1995*. Morgan-Kaufmann, Los Altos, Calif., 420–431.

Han, J., Pei, J., and Yin, Y. 2000. Mining frequent patterns without candidate generation. In *Proceedings of the SIGMOD 2000*. ACM, New York, 1–12.

Hellerstein, J. M., Haas, P. J., and Wang, H. J. 1997. Online aggregation. In *Proceedings of the SIGMOD 1997*. ACM, New York, 171–182.

Hidber, C. 1999. Online association rule mining. In *Proceedings of the SIGMOD 1999*. ACM, New York, 145–156.

Korn, F., Labrinidis, A., Kotidis, Y., and Faloutsos, C. 1998. Ratio rules: A new paradigm for fast, quantifiable data mining. In *Proceedings of the VLDB 1998*. Morgan-Kaufmann, Los Altos, Calif., 582–593.

Lakshmanan, L. V. S., Leung, C. K.-S., and Ng, R. T. 2000. The segment support map: Scalable mining of frequent itemsets. *SIGKDD Expl. 2*, 2 (Dec.), 21–27.

Lakshmanan, L. V. S., Ng, R., Han, J., and Pang, A. 1999. Optimization of constrained frequent set queries with 2-variable constraints. In *Proceedings of the SIGMOD 1999*. ACM, New York, 157–168.

Leung, C. K.-S., Lakshmanan, L. V. S., and Ng, R. T. 2002a. Exploiting succinct constraints using FP-trees. *SIGKDD Expl. 4*, 1 (June), 40–49.

Leung, C. K.-S., Ng, R. T., and Mannila, H. 2002b. OSSM: A segmentation approach to optimize frequency counting. In *Proceedings of the ICDE 2002*. IEEE Computer Society Press, Los Angeles, Calif., 583–592.

Liu, J., Pan, Y., Wang, K., and Han, J. 2002. Mining frequent item sets by opportunistic projection. In *Proceedings of the KDD 2002*. ACM, New York, 229–238.

Miller, R. J. and Yang, Y. 1997. Association rules over interval data. In *Proceedings of the SIGMOD 1997*. ACM, New York, 452–461.

Ng, R. T., Lakshmanan, L. V. S., Han, J., and Pang, A. 1998. Exploratory mining and pruning optimizations of constrained associations rules. In *Proceedings of the SIGMOD 1998*. ACM, New York, 13–24.

Park, J. S., Chen, M.-S., and Yu, P. S. 1995. Efficient parallel data mining for association rules. In *Proceedings of the CIKM 1995*. ACM, New York, 31–36.

Park, J. S., Chen, M.-S., and Yu, P. S. 1997. Using a hash-based method with transaction trimming for mining association rules. *IEEE Trans. Knowl. Data Eng. 9*, 5 (Sep./Oct.), 813–825.

Pei, J., Han, J., and Lakshmanan, L. V. S. 2001. Mining frequent itemsets with convertible constraints. In *Proceedings of the ICDE 2001*. IEEE Computer Society Press, Los Angeles, Calif., USA, 433–442.

Raman, V. and Hellerstein, J. M. 2002. Partial results for online query processing. In *Proceedings of the SIGMOD 2002*. ACM, New York, 275–286.

Sarawagi, S., Thomas, S., and Agrawal, R. 1998. Integrating association rule mining with relational database systems: Alternatives and implications. In *Proceedings of the SIGMOD 1998*. ACM, New York, 343–354.

Savasere, A., Omiecinski, E., and Navathe, S. 1995. An efficient algorithm for mining association rules in large databases. In *Proceedings of the VLDB 1995*. Morgan-Kaufmann, Los Altos, Calif., 432–444.

Silverstein, C., Brin, S., Motwani, R., and Ullman, J. 1998. Scalable techniques for mining causal structures. In *Proceedings of the VLDB 1998*. Morgan-Kaufmann, Los Altos, Calif., 594–605.

Toivonen, H. 1996. Sampling large databases for association rules. In *Proceedings of the VLDB 1996*. Morgan-Kaufmann, Los Altos, Calif., 134–145.

Tsur, D., Ullman, J. D., Abiteboul, S., Clifton, C., Motwani, R., Nestorov, S., and Rosenthal, A. 1998. Query flocks: A generalization of association-rule mining. In *Proceedings of the SIGMOD 1998*. ACM, New York, 1–12.

Received February 2002; revised February 2003; accepted July 2003