An Audit Environment for Outsourcing of Frequent Itemset Mining

W. K. Wong
The University of Hong Kong
wkwong2@cs.hku.hk

David W. Cheung
The University of Hong Kong
dcheung@cs.hku.hk

Edward Hung
The Hong Kong Polytechnic University
csehung@comp.polyu.edu.hk

Ben Kao
The University of Hong Kong
kao@cs.hku.hk

Nikos Mamoulis
The University of Hong Kong
nikos@cs.hku.hk

ABSTRACT

Finding frequent itemsets is the most costly task in association rule mining. Outsourcing this task to a service provider brings several benefits to the data owner such as cost relief and a less commitment to storage and computational resources. Mining results, however, can be corrupted if the service provider (i) is honest but makes mistakes in the mining process, or (ii) is lazy and reduces costly computation, returning incomplete results, or (iii) is malicious and contaminates the mining results. We address the integrity issue in the outsourcing process, i.e., how the data owner verifies the correctness of the mining results. For this purpose, we propose and develop an *audit environment*, which consists of a database transformation method and a result verification method. The main component of our audit environment is an *artificial itemset planting* (AIP) technique. We provide a theoretical foundation on our technique by proving its appropriateness and showing probabilistic guarantees about the correctness of the verification process. Through analytical and experimental studies, we show that our technique is both effective and efficient.

1. INTRODUCTION

Association rule mining discovers correlated itemsets that occur frequently in a transactional database. A variety of efficient algorithms for mining association rules have been proposed [1, 2, 4]. The problem can be divided into two subproblems: (i) computing the set of frequent itemsets, and (ii) computing the set of association rules based on the mined frequent itemsets. While the latter problem (rule generation) is computationally inexpensive, the problem of mining frequent itemsets has an exponential time complexity and is thus very costly. This motivates businesses to outsource the task of mining frequent itemsets to service providers. With outsourcing, a data owner exports its data to a service provider, who returns the set of frequent itemsets together with their support counts. Apart from cost relief, outsourcing also brings a number of benefits. For example, if data is transient and only a statistical summary (as captured by frequent itemsets and association rules) is desired, the data owner can ship its data to a service provider without archiving them locally.[1] As another benefit, transactional data collected at different sources (such as those generated at different stores of a chain supermarket) can be consolidated and processed at the service provider. The service provider can find the frequent itemsets that are local to each individual source, or the global frequent itemsets for the whole organization. The cost of transferring transactions among the sources and performing the global mining in a distributed manner can be saved. Finally, with outsourcing, the data owner does not need to maintain an IT team for the data mining task.

For outsourcing to be practical, the issues of *security* and *integrity* have to be addressed satisfactorily. Regarding security, the data owner has to ensure that neither the content of its data nor the mining result is disclosed to the service provider. This security problem has been addressed in [16], in which an encryption scheme was devised to protect data content and mining results. In this paper we focus on the integrity problem, that is, how the data owner can ensure the correctness of the mining results. The results of this paper, combined with the techniques we proposed in [16] for enforcing security, constitute a complete solution to the outsourcing problem.

The first step towards solving the integrity problem is to understand the behavior of a (potentially malicious) service provider that can undermine the integrity of the mining results. A service provider may return inaccurate results if (i) it is honest but sloppy, e.g., there are bugs in its mining programs; (ii) it is lazy and tries to reduce costly computation, e.g., it mines only a small portion of the dataset; (iii) it is malicious and purposely returns wrong results, e.g., a busi-

[1]This is an alternative approach to applying a data mining algorithm for streaming data [9]. The advantage is that with outsourcing the data owner receives the complete and exact set of frequent itemsets from the service provider, while applying a streaming data mining method only computes an approximate solution to the problem.

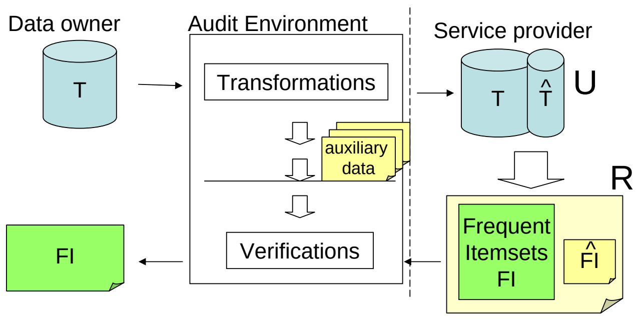

Figure 1: The architecture of the scheme

ness competitor has paid the service provider for providing incorrect results so as to affect the business decisions of the data owner. The concept of *result integrity* should thus be defined on two criteria:

- *Correctness*: All returned frequent itemsets are actually frequent and their returned support counts are correct.
- *Completeness*: All actual frequent itemsets are included in the result.

A straightforward attempt to solving the integrity problem is to verify the mining results against the database — we scan the database once to count the support of each frequent itemset reported in the result. These support counts are then compared against those returned by the service provider. Though simple, this approach has a number of shortcomings. First, it verifies the correctness criterion but not the completeness criterion. It fails to detect frequent itemsets that are missing in the result. Second, it is somewhat costly. The verification requires scanning the complete database once and counting the supports of a (potentially) large set of itemsets. Third, it requires the original database to be available. If the content of the database is continuously updated, an image dump has to be taken and archived (for later verification). This adds to the cost of the mining exercise, particularly when the database is large. It is thus not suitable for applications such as those related to data streams.

Our approach to solve the integrity problem is to construct an *audit environment*. Essentially, an audit environment consists of (i) a set of transformation methods that transform a database T to another database U, based on which the service provider will mine and return a mining result R; (ii) a set of verification methods that take R as an input and return a deduction of whether R is correct and complete; (iii) auxiliary data that assist the verification methods. An interesting property of our approach is that the audit environment forms a standalone system. It is self-contained in the sense that the verification process can be done entirely by using only the auxiliary data that are included in the environment. In other words, the original database need not be accessed during verification. Figure 1 shows the architecture of our scheme.

The core component of our audit environment is a technique of database transformation and verification called *artificial itemset planting* (AIP). AIP provides probabilistic guarantees that incorrect or incomplete mining results returned by the service provider will be identified by the owner with a controllably high confidence. To give the intuition behind AIP, we briefly describe it here (more details will be given in Section 4.1). Given a set of itemsets $\widehat{FI}$, AIP generates a (small) artificial database $\hat{T}$ such that all itemsets in $\widehat{FI}$ are guaranteed to be frequent and their exact support counts are known. Also, the original database T and $\hat{T}$ contain disjoint sets of items. T is then transformed to U by merging transactions in T with those in $\hat{T}$ (i.e., a transaction in $\hat{T}$ is appended to the end of some transaction in T). The idea is that when the service provider mines U, the set $\widehat{FI}$ (and the associated support counts) will be part of the mining result R. Since the service provider cannot distinguish itemsets of T from those of $\hat{T}$, if the result R is incorrect or incomplete, there are high chances that the returned $\widehat{FI}$ is also incorrect or incomplete. So, by verifying $\widehat{FI}$, we are able to obtain a probabilistic guarantee on whether the result integrity is enforced. Essentially, $\widehat{FI}$ serves as a *fragile watermark* of the mining result — perturbation of the result will very likely destroy the integrity of $\widehat{FI}$.

Our Contributions. The contributions of this work include: (i) a formal definition of a model of malicious actions that a service provider might perform to undermine result integrity; (ii) a novel artificial itemset planting (AIP) technique for constructing an audit environment; (iii) a theoretical study on the cost and effectiveness of AIP technique; and (iv) an empirical study to evaluate the performance of the proposed methods.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 defines our model of malicious service providers and an audit environment. Section 4 describes the AIP technique for constructing an audit environment. We propose efficient algorithms for implementing AIP and give an analytical study on the algorithms. Section 5 empirically evaluates the performance of AIP, both in terms of its effectiveness in detecting malicious actions performed by a service provider and the efficiency of our algorithms. Finally, Section 6 concludes the paper.

2. RELATED WORK

The problem of outsourcing the task of data mining with accurate result was first introduced in our previous work [16]. There, we address the security issues in outsourcing association rule mining. An item mapping and transaction transformation approach was proposed to encrypt a transactional database and to decrypt the mined association rules returned from a service provider. This paper focuses on the integrity issues and thus complements the study in [16]. A data owner can apply both techniques to protect sensitive information and at the same time verify the result returned from the service provider. To the best of our knowledge, integrity issues in outsourcing data mining have not been studied before.

The most similar model to outsourcing data mining is the outsourced database model [5]. A data owner exports its database to a service provider who processes queries by the owner and reports results. A number of papers have been published on the integrity problem of the outsourced database model [7, 12, 8, 15, 17, 11]. For example, in [7, 12, 8], Merkle hash trees are built on both the owner side and the service provider side to achieve authentication of query results. As another example, in [11], each record in a database is digitally signed. The proposed signature scheme has an interesting property that missing tuples in query results can be detected. In the above examples, queries are limited to those that look for sets of tuples as answers (such as point and range queries). Aggregate queries are not supported. In [15], an alternative strategy, called challenge token, was proposed. The scheme allows general queries (point, range, aggregate) to be verified; challenge tokens (queries whose answers are known) are submitted to the service provider together with regular queries. In addition to the query answers, the service provider finds and returns the tokens, which are then used as proof of integrity. The scheme, however, can only guard against "sloppy" and "lazy" providers, who do not intentionally return incorrect or incomplete results. Malicious providers may selectively answer challenge tokens correctly but provide wrong answers for other queries. They can thus work around the scheme. In [17], fake tuples are injected into a database. By tracking the fake tuples, query results are probabilistically verified. The advantage of this scheme is that it works conveniently on off-the-shelf database systems. The method is thus unintrusive (unlike, e.g., the Merkle-hash-tree-based methods). The drawback of the fake-tuple scheme is that it does not support aggregate queries. In the outsourced data mining model, query results are composed of statistical aggregations (e.g., itemset counts in association rule mining, centroid computation in clustering). The above technique is thus not applicable. The integrity problem in outsourced frequent itemset mining has not been addressed.

A major difference between the outsourced database model and the outsourced mining model is that for the former, a service provider is expected to answer numerous (small) queries on the same database, while for the latter, one or only a few mining exercises are performed for each instance of the database. A larger amount of resources, such as storage and preparation cost can be invested for the outsourced database model, since the cost can be amortized over a large number of owner queries. On the other hand, an outsourced mining model should avoid high preparation cost, as it is not expected to pay-off.

In the brief description of our artificial itemset planting (AIP) technique (Section 1), we mentioned about generating an artificial database $\hat{T}$ so that its (known) set of frequent itemsets $\widehat{FI}$ can be used to verify the mining results. The generation of the database $\hat{T}$ is a core part of AIP. Given a set of frequent itemsets and the corresponding support counts, the problem of generating a database that satisfies the support constraints is proved to be an NP-hard problem [10]. In [3], an iterative approach that uses a greedy heuristic is proposed to generate such a database. As we have argued, the preparation cost of the outsourced mining model should be small, the cost of the heuristic algorithm put forward in [3] is still too high to be practical (e.g., the algorithm requires multiple database scans). There are other database generation algorithms previously proposed in the literature, e.g., [13, 14]. Since many of the properties of the generated databases (such as database size and the set of frequent itemsets) cannot be precisely controlled, they are not suitable for AIP. In this paper we propose a method for efficiently generating an artificial database $\hat{T}$ for AIP. Our database generation method does not contradict the NP-hardness result proved in [10] because the set of frequent itemsets $\widehat{FI}$ and the associated support counts are not rigidly fixed. Instead, the constraints are dynamically adjusted so that an efficient method for generating $\hat{T}$ is possible. Details about this database generation approach will be discussed in Section 4.1.

3. MODEL

In this section we formally define the integrity problem in outsourcing frequent itemset mining. We define notation, state the properties of an audit environment, define the set of malicious actions that a service provider might perform to alter the mining results, and formulate the concept of "malicious service provider gain" which captures the incentive and penalty to a service provider for his malicious actions.

Let I be a set of items. A transaction t_i is a subset of I. A transaction t_i *contains* an itemset x if and only if $x \subseteq t_i$. Given a database T that contains a number of transactions, the support count of an itemset x is the number of transactions in T that contain the itemset x. Let σ be a function such that $\sigma(x)$ gives the support count for any itemset $x \subseteq I$. Given a support threshold $s\%$, an itemset x is *frequent* if and only if $\sigma(x) \geq |T| \times s\%$, where $|T|$ is the number of transactions in T. The objective of frequent itemset mining is to find all frequent itemsets and their support counts in T with respect to a given support threshold.

3.1 Malicious Actions

Assume a party p_{owner} owns a set of transactions T. Another party (service provider) p_{miner} helps p_{owner} to compute the set of frequent itemsets L in T. The service provider p_{miner} is not trusted and it is possible that p_{miner} performs malicious actions and purposely modifies the mining results. Let $R = (L, \sigma)$ be the true result of mining (i.e., L is the complete set of frequent itemsets and $\sigma(x)$ gives the correct support count for any $x \in L$). Let $R' = (L', \sigma')$ be the result returned by p_{miner}. R' may not equal R and p_{miner} may have performed a series of the following malicious actions:

Insertion. p_{miner} includes an infrequent itemset in the returned set of frequent itemset claiming that the itemset is frequent. More specifically, p_{miner} picks an itemset $y \notin L$, sets $L' = L \bigcup \{y\}$, and sets $\sigma'(y) = r$ where r is an artificially generated value that is greater than the support threshold. $\sigma'(x) = \sigma(x)$ for all $x \in L$.

Deletion. p_{miner} excludes a frequent itemset from the returned result. p_{miner} picks an itemset $y \in L$ and sets $L' = L - \{y\}$. $\sigma'(x) = \sigma(x)$ for all $x \in L'$.

Replacement. p_{miner} returns a modified (incorrect) support count of a frequent itemset. p_{miner} picks an itemset $y \in L$, sets $L' = L$, and sets $\sigma'(y) = r \neq \sigma(y)$ where r is an artificially generated value that is greater than the support threshold. $\sigma'(x) = \sigma(x)$ for all $x \in L' - \{y\}$.

Every possible returned result given by the miner can be simulated by a series of the above malicious actions. Insertions and modifications contaminate the correctness of the result while deletions affect the completeness of the result. If it can be proved that the miner has not performed any of the three malicious actions, the returned result will be both *correct* and *complete*. We remark that a malicious miner can be easily caught if it performs the malicious actions randomly since the returned set L' may not satisfy the monotonicity property [1] (which states that any subset of a frequent itemset must be frequent). For example, let $I = \{A, B, C\}$. Suppose p_{miner} computes $L = \{A, B, AB\}$. If p_{miner} inserts AC to this result, the returned result to the owner is

$L' = \{A, B, AB, AC\}$. Note that L' does not satisfy the monotonicity property (C is a subset of AC, however, AC is frequent and C is infrequent). Similarly, if p_{miner} deletes B, but not AB, there will be an integrity violation due to monotonicity. This observation leads us to the definition of a *valid return*.

Definition 1. *(Valid Return) A returned result $R' = (L', \sigma')$ is valid if $\forall x \in L', \forall y \subset x$, $y \neq \emptyset \Rightarrow y \in L'$ and $\sigma'(y) \geq \sigma'(x)$.*

A *smart* but malicious miner should always give a valid return, since violation of integrity in invalid returns can easily be detected. For example, if p_{miner} decides to insert an itemset $x \notin L$ to L', he should also insert all the subsets of x that are not in L. In the following discussion, we assume that R' is always valid.

3.2 Expected Gain

When a malicious service provider performs a malicious action, the mining result is contaminated and he is rewarded, for example, from a business competitor of p_{owner}. The more malicious actions are performed, the more rewards are earned. On the other hand, if a malicious action is detected, the service provider not only loses the reward he would be paid for performing the mining task, but should also compensate p_{owner} for returning incorrect results. In addition, if the service provider is caught changing the results, he loses its reputation in the industry, which is a big penalty. The aim of the malicious service provider is to perturb the mining result as much as possible without being noticed. We model p_{miner}'s gain and loss of perturbing mining results by a measure called *expected gain* (EG).

Definition 2. *(Expected Gain) Let $R = (L, \sigma)$ be the true result and $R' = (L', \sigma')$ be the returned result. Let n be the minimum number of malicious actions taken to obtain R' from R and A_1, A_2, ..., A_n be the corresponding n malicious actions. Let ϕ be a scoring function such that $\phi(A_i)$ returns the score gained by performing A_i. Let ρ be the penalty the miner suffers if any of its malicious actions is detected by p_{owner}. Let p be the probability of such a detection. The expected gain (EG) is given by, $EG(R') = (1-p)\sum_{i=1}^{n}\phi(A_i) - p\rho$.*

Note that $EG(R) = 0$ if the miner returns the true result R. The objective of a malicious miner is to find an R' such that $EG(R')$ is maximized. If $EG(R') < 0$ for all $R' \neq R$, p_{miner} should be forced to return the true result R, as he will suffer a certain penalty for doing otherwise. The goal of our audit environment is to transform the data prior to outsourcing in order to force the service provider to return the correct result.

3.3 Audit Environment

An audit environment consists of a set of transformation methods, a set of verification methods, and auxiliary data for verification. An audit environment is self-contained such that the verification process can be carried out without accessing the original database. Moreover, it should satisfy the following properties:

- Its preparation cost should be low. The resources put in this process should be much less than the resources required by the mining process.
- The audit environment should not induce a large overhead to the service provider. In particular, mining the transformed database U should not cost much more than mining the original database T.
- The audit environment should be robust. In particular, the expected gain of a malicious miner should be controllably small or even negative.

4. PREPARING THE AUDIT ENVIRONMENT

In this section we discuss how an audit environment can be prepared efficiently. We first prove a theorem that allows us to detect malicious insertions and deletions by examining the positive and negative borders of L'. We then discuss a straightforward method for detecting malicious replacements. We point out the drawbacks of the straightforward method and propose our novel technique AIP. We start by defining the terms *negative border* and *positive border*.

Definition 3. *(Negative Border) Given an item domain I, let S be a set of frequent itemsets that satisfy the monotonicity property. The negative border of S, denoted by $B^-(S)$, is the set of all minimal infrequent itemsets w.r.t. to S, i.e., $B^-(S) = \{x \mid x \subseteq I$ and $x \notin S$ and $\forall y \subset x$ where $y \neq \emptyset$, $y \in S\}$.*

Definition 4. *(Positive Border) Given a set of frequent itemsets S that satisfies the monotonicity property, the positive border of S, denoted by $B^+(S)$, is the set of all maximal frequent itemsets w.r.t. to S, i.e., $B^+(S) = \{x \mid x \in S$ and $\forall y \supset x, y \notin S\}$.*

For example, if $I = \{A, B, C, D\}$, $S = \{A, B, C, AB, BC\}$, then $B^-(S) = \{D, AC\}$ and $B^+(S) = \{AB, BC\}$.

Given a result $R' = (L', \sigma')$ returned by p_{miner}, we need to verify that no malicious insertions, deletions, or replacements have been applied. The following theorem shows that insertions and deletions can be detected by examining the borders of L'.

Theorem 1. *Suppose p_{miner} returns a valid return $R' = (L', \sigma')$ to p_{owner}. No insertions are performed to the actual set L if and only if all itemsets in $B^+(L')$ are frequent in p_{owner}'s database and no deletions are performed if and only if all itemsets in $B^-(L')$ are infrequent in p_{owner}'s database.*

Proof. *Insertion-if.* We prove the transposition of the statement. If the miner has inserted an itemset x, then $x \in L'$ and $x \notin L$. Since R' is a valid return, there must exist an itemset $y \in B^+(L')$ such that $x \subseteq y$. By the monotonicity property, $x \notin L \Rightarrow y \notin L$. Hence, there exists y in the positive border that is not frequent.

Insertion-only if. If no insertions are performed, the miner must have only performed deletions and/or replacements. So, $L' \subseteq L$. Since $B^+(L') \subseteq L'$, all itemsets in $B^+(L')$ are frequent.

Deletion-if. We prove the transposition of the statement. If the miner has deleted an itemset x, then $x \in L$ and $x \notin L'$. Since R' is a valid return, there must exist an itemset $y \in B^-(L')$ such that $y \subseteq x$. By the monotonicity property, $x \in L \Rightarrow y \in L$. Hence, there exists y in the negative border that is frequent.

Deletion-only if. If no deletions are performed, the miner would have only performed insertions and/or replacements. So, $L \subseteq L'$. Since $B^-(L') \bigcap L' = \emptyset$, we have $B^-(L') \bigcap L = \emptyset$. So, all itemsets in $B^-(L')$ are infrequent. $\Box$

From Theorem 1, we know that it is *necessary* that all support counts of itemsets in the borders $B^-(L')$ and $B^+(L')$ are verified. Also, to detect replacement, we need to verify support counts of itemsets in L'. Therefore, an ideal audit environment should include all the support counts of itemsets in $L' \bigcup B^+(L') \bigcup B^-(L') = L' \bigcup B^-(L')$ for verification.

As we have argued, it is desirable that an audit environment be prepared as the database is exported to a miner. The audit environment should also be self-contained so that subsequent verification does not require accesses to the original database (which might have already been updated or unavailable during verification). Therefore, preparing such an audit environment with support counts of all the itemsets in $L' \bigcup B^-(L')$ is impractical because the set L' is not known when the environment is being prepared. Also, finding all these supports is equivalent to mining the database, which defeats the purpose of outsourcing.

One possible approach to reduce verification cost is sampling. For example, we select a set of itemsets Z and count their supports. An audit environment includes all these counts. Given a result $R' = (L', \sigma')$, we verify the support counts of itemsets in $Z \bigcap (L' \bigcup B^-(L'))$, effectively examining only a sample of $L' \bigcup B^-(L')$. A major problem with the simple sampling strategy is that the universe of itemsets is very large and thus most of the elements in Z may not be in $L' \bigcup B^-(L')$. Therefore, the set Z has to be sufficiently large in order for the verification process to be statistically reliable, making the method inefficient.

To make the approach more effective, we wisely set up an *artificial* sample Z and inject it to the original database so that most of Z's elements are in $L' \bigcup B^-(L')$. This leads to the AIP method which we describe next.

4.1 Overview of artificial itemset planting

The idea of AIP is to insert artificial items in the database such that the support counts of certain itemsets are known by the owner, who uses them to verify the correctness and completeness of the mining result. More specifically, let I_A be a set of artificial items (we assume $I_A \bigcap I = \emptyset$). We select two sets of artificial itemsets: *AFI* (Artificial Frequent Itemsets) and *AII* (Artificial Infrequent Itemsets). We then generate an artificial database $\hat{T}$ with n transactions $\widehat{t_1}, \ldots, \widehat{t_n}$, where n is the size of the original database T, such that (1) $\widehat{t_i} \subseteq I_A$ for $1 \leq i \leq n$; (2) each itemset in *AFI* is frequent in $\hat{T}$ (with respect to the mining support threshold s); and (3) each itemset in *AII* is infrequent in $\hat{T}$. (Note that *AFI* (*AII*) does not have to contain *all* frequent (infrequent) itemsets in $\hat{T}$.) During the database generation process, we register the support counts of all itemsets in *AFI* and *AII*. The original database T is then transformed into a database $U = \{u_1, ..., u_n\}$ such that $u_i = t_i \cup \widehat{t_i}$. We are thus extending T horizontally by merging transactions in T with those in $\hat{T}$. The database U is then submitted to p_{miner}.

The sets *AFI* and *AII* together serve as the set Z for result verification and they are included in the audit environment (with the corresponding support counts). To illustrate the idea, let $I = \{A, B, C, D\}$, $L = \{A, B, AB\}$ and $I_A = \{\alpha, \beta, \gamma\}$. Suppose we select $AFI = \{\alpha, \beta, \alpha\beta\}$ and $AII = \{\gamma\}$, then the itemsets in $Z = \{\alpha, \beta, \gamma, \alpha\beta\}$ and their support counts will be included in the audit environment. Suppose p_{miner} returns $L' = \{A, B, AB, \alpha, \beta, \gamma\}$, we verify the itemsets in $Z \bigcap (L' \bigcup B^-(L')) = \{\alpha, \beta, \gamma, \alpha\beta\}$. With the help of Theorem 1, we detect an insertion since $\gamma \in B^+(L')$ belongs to L', however, we know that γ is infrequent ($\gamma \in AII$), and we detect a deletion since itemset $\alpha\beta \in B^-(L')$ does not belong to L', but we know that it is frequent ($\alpha\beta \in AFI$). We also attempt to detect replacement actions by comparing the counts returned by the miner to those recorded in the environment for all the itemsets in $Z \bigcap L'$.

The crux of AIP is the selection of *AFI* and *AII*, and the generation of the artificial database $\hat{T}$. We remark that the sets and the database have to satisfy a number of stringent restrictions. For example, *AFI* and *AII* must not violate the monotonicity property — a (frequent) itemset in *AFI* must not contain an (infrequent) itemset in *AII*; itemsets in *AFI* must be frequent in $\hat{T}$; and itemsets in *AII* must be infrequent in $\hat{T}$.

An efficient and automatic method for determining *AFI*, *AII* and $\hat{T}$ is a challenging problem. In the following subsections, we first provide the theoretical foundation for checking whether a choice of *AFI* and *AII* can be used as a basis for AIP. Then, we describe an algorithm for constructing a pair of *AFI* and *AII*, based on this theory. Next, the process that generates the artificial database is outlined. A security and cost analysis follows. Finally, we propose some optimizations that reduce the cost of generating the artificial database $\hat{T}$ to be outsourced.

4.2 Checking the validity of an itemset pattern

We first consider the selection of an *AFI* and an *AII*. We call an (*AFI*, *AII*) pair an *itemset pattern*. An itemset pattern is valid if it is possible to generate a database that satisfies the support requirements of the pattern.

DEFINITION 5. *(Valid pattern) We say that an itemset pattern is an s-valid pattern if there exists a database $\hat{T}$ such that all itemsets in AFI are frequent in $\hat{T}$ and all itemsets in AII are infrequent in $\hat{T}$, with respect to a given support threshold s%.*

It is obvious that a valid pattern must not violate the monotonicity property, which can be checked and enforced easily. Satisfying the monotonicity property, however, is not sufficient. For example, suppose the support threshold is 100%, the pattern: ($AFI = \{A, B\}, AII = \{AB\}$) satisfies the monotonicity property. Since $s = 100\%$, every transaction generated for the pattern must contain both A and B, and so AB is frequent and cannot be in *AII*. This shows that the pattern is not a valid pattern with respect to $s = 100\%$.

A simple way to satisfy *AII* is to include no itemsets in *AII* in the generated transactions. To satisfy *AFI*, in the generated database, for each itemset $x \in AFI$, at least $n \times s\%$ transactions should contain x, where n is the total number of transactions generated. If $|AFI| > 1/s\%$, then some transactions must contain at least 2 itemsets from *AFI*. Doing so may accidentally cause some itemsets in *AII* to be included in the generated transactions, jeopardizing correctness.

As an example, if $AFI = \{AX, BY\}$ and $AII = \{AB\}$, then a transaction that includes both AX and BY includes

AB as well. Intuitively, two itemsets x_i and x_j in AFI *conflict* if a transaction that includes both x_i and x_j has the potential of including some itemsets in AII. We now formally define the concept of "conflict" and prove that if conflicting itemsets are never included in the same transaction, then we can generate a database with no itemsets in AII included in any transactions.

DEFINITION 6. *(Conflicts in AFI) Let x_i, x_j be two distinct itemsets in AFI. x_i conflicts with x_j if and only if $\exists z \in AII$ such that $(z - x_i) \bigcap x_j \neq \emptyset$ and $(z - x_j) \bigcap x_i \neq \emptyset$.*

For example, consider $AFI = \{AX, AY, BY, CZ, ABZ\}$, $AII = \{ABC\}$. AX conflicts with BY, AX conflicts with CZ, while AX does not conflict with AY, and AX does not conflict with ABZ. Conflict is a symmetric relationship; if x conflicts with y then y conflicts with x.

THEOREM 2. *Assume AFI and AII satisfy the monotonicity property (i.e., no itemset in AFI contains an itemset in AII). Suppose we pick k itemsets $(x_1, x_2, \ldots, x_k)$ in AFI and construct $t_i = \bigcup_{i=1}^{k} x_i$. If an AII itemset y is contained in t_i, i.e., $y \subseteq t_i$, then $\exists p, q \in [1, k]$ such that $p \neq q$ and x_p conflicts with x_q.*

PROOF. Since $y \subseteq t_i$ and $t_i = \bigcup_{i=1}^{k} x_i$, $\exists p$ such that $y \bigcap x_p \neq \emptyset$. Without loss of generality, we assume there does not exist another x_i $(i \in [1, k], i \neq p)$ such that $x_p \bigcap y \subset x_i \bigcap y$. (If such an x_i exists, we take x_i in place of x_p and repeat the argument.) Since $x_p \in AFI$ and $y \in AII$, y cannot be a subset of x_p (recall that AFI and AII satisfy the monotonicity property). So, $y - x_p \neq \emptyset$. In other words, some items in y must come from another itemset in AFI, i.e., $\exists q$, $q \neq p$ and $(y - x_p) \bigcap x_q \neq \emptyset$. Also, since $x_p \bigcap y \not\subset x_q \bigcap y$, there exists an item $m \in y \bigcap x_p$ such that $m \notin x_q$ (and thus $m \notin y \bigcap x_q$). It follows that $(y - x_q) \bigcap x_p \neq \emptyset$. By definition, x_p conflicts with x_q. □

Theorem 2 gives us a guideline of generating an artificial database. More specifically, if we never put conflicting AFI itemsets in the same transaction, then no transactions will contain any AII itemsets. We thus guarantee that all AII itemsets have zero support and thus are never frequent with respect to any non-zero support threshold.

We represent the conflict relationship among AFI itemsets in a *conflict graph* $G = (V, E)$. Each itemset in AFI is represented by a node in G, i.e., $V = AFI$. An edge (v_1, v_2) is in E if and only if v_1 *conflicts* with v_2. The number of neighbors of a node v in the conflict graph thus represents the number of itemsets that conflict with v.

DEFINITION 7. *(Conflict index) Given a conflict graph $G = (V, E)$, for $x \in V$, let $N(x)$ be the set of neighbors of x, i.e., $N(x) = \{y \mid (x, y) \in E\}$. The conflict index c_x of x equals the number of neighbors (degree) of x, i.e., $c_x = |N(x)|$. The conflict index of G, $CI(G) = max_{x \in V} c_x$.*

THEOREM 3. *An itemset pattern (AFI, AII) is an s-valid pattern if both of the following conditions hold:*

1. *AFI and AII satisfy the monotonicity property.*

2. *$CI(G) \leq \frac{1}{s\%} - 1$ where s is the support threshold and G is the conflict graph representing the itemset pattern.*

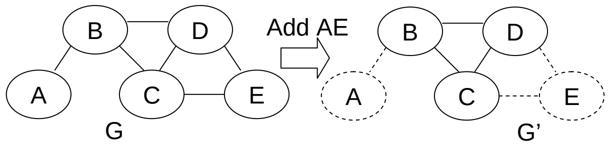

Figure 2: Updating a conflict graph after itemsets A and E are used to compose a transaction

PROOF. We prove the theorem by constructing a database that matches the requirements[2]. Without loss of generality, assume we have to generate $1/s\%$ transactions. (To generate an artificial database of n transactions, we replicate the database $ns\%$ times.) Thus, an itemset that is contained in at least one transaction is frequent.

A transaction is generated by adding AFI itemsets to it. Intuitively, we want to add as many AFI itemsets without bringing in any AII itemsets to the transaction. By Theorem 2, this can be achieved by ensuring that no conflicting AFI itemsets are added to the transaction. To do so, we maintain two sets Q^+ and Q^-, which are initially empty. Q^+ keeps track of the itemsets that have been added to the transaction, and Q^- keeps track of the itemsets that conflict with any itemsets in Q^+. We randomly pick an itemset v in AFI, put v in Q^+ and all its neighbors $N(v)$ to Q^-. We repeat this process until AFI is partitioned into:

- Q^+: Every itemset in Q^+ does not conflict with any other itemsets in Q^+.

- Q^-: Every itemset in Q^- conflicts with at least one itemset in Q^+.

The first transaction is given by $\bigcup_{x \in Q^+} x$. Since all itemsets in Q^+ are now frequent (recall that we only need a support count of 1 to make an itemset frequent), subsequent transactions need not contain them. We remove all itemsets in Q^+ from AFI and update the conflict graph removing the corresponding nodes and their associated edges. Let $G' = (V', E')$ be the updated conflict graph and c'_x be the conflict index of any node x in G'.

For any node x in G', we know that $x \in Q^-$. Hence, there must exist a neighbor of v in the original conflict graph G that has been removed in G'. So, $c'_x \leq c_x - 1$. This implies $CI(G') \leq CI(G) - 1$. The conflict index of the conflict graph is reduced by at least 1.

To generate another transaction, we repeat the above procedure. Finally, the conflict index of the conflict graph will be reduced to 0. This implies that the itemsets remaining in AFI are conflict-free. We then generate a transaction that includes all these remaining itemsets. Note that in the whole process, we have generated at most $CI(G) + 1$ transactions.

Recall that we have to generate a database of $1/s\%$ transactions. So if $1/s\% \geq CI(G) + 1$, all the transactions generated in the above procedure can be accommodated. We can replicate some of the generated transactions so that the total number of them is $1/s\%$. As a result, if $CI(G) \leq \frac{1}{s\%} - 1$, the procedure correctly generates a database with the desired property. Hence, the theorem. □

[2]We refer to this construction method "baseline construction" in the rest of the paper.

Let us use an example to illustrate this baseline construction procedure. Consider the itemset pattern ($AFI = \{A, B, C, D, E\}$, $AII = \{AB, BC, BD, CD, DE, CE\}$), whose conflict graph G is shown in Figure 2 (left). $CI(G) = 3$. To generate the first transaction, we pick an AFI itemset, say A, and get $Q^+ = \{A\}$, $Q^- = \{B\}$. Since $\{C, D, E\}$ in AFI are not yet partitioned, we repeat the process and pick, say, E, resulting in $Q^+ = \{A, E\}$, $Q^- = \{B, C, D\}$. Now, AFI is partitioned into Q^+ and Q^-, therefore transaction AE is generated. The conflict graph is updated by removing A and E (see Figure 2 (right)), which has a smaller conflict index ($CI(G') = 2$). The process is repeated for the remaining AFI itemsets ($\{B, C, D\}$) and eventually the baseline construction method generates three more transactions: B, C, and D.

Theorem 3 states a sufficient condition under which an itemset pattern is valid. We call this condition *valid-pattern condition* or *vp-condition* for short. Our next task is to generate such a valid pattern.

4.3 Valid itemset pattern generation

Recall that itemsets in AFI and AII are included in an audit environment and are used to verify the mining result. So, a larger $AFI \cup AII$ leads to a higher verification confidence, but also a longer verification time.

We assume that p_{owner} has some rough information of the number of frequent itemsets and the number of itemsets in the negative border of his database (i.e., $|L|$ and $|B^-(L)|$). For example, these figures could be obtained from a previous mining exercise, or from the mining result of a small sample of the database. The owner then selects a fraction $0 < f \leq 1$ and set the *target sizes* of AFI and AII to $f_{AFI} = f \cdot |L|$ and $f_{AII} = f \cdot |B^-(L)|$, respectively. (Here, $|L|$ and $|B^-(L)|$ are rough estimates.) The owner thus controls the tradeoff between verification accuracy and speed through f.[3] We now briefly describe the high-level idea of a procedure for generating a valid itemset pattern (AFI, AII) such that $|AFI| \geq f_{AFI}$ and $|AII| \geq f_{AII}$. A pseudocode showing the details of the procedure is listed in the Appendix.

First, we create a set of artificial items, I_A.[4] The procedure attempts to add itemsets to AII until both AFI and AII are "big enough". We randomly generate an itemset $J \subseteq I_A$ that is not already in AII and add J to AII. Since itemsets in AII are used to verify the negative border of the mining result, we add all immediate subsets of J to AFI (so as to make sure that J is in the negative border of the returned result if p_{miner} is not malicious). For example, if $J = ABC$, we add AB, BC, AC to AFI. If the resulting (AFI, AII) does not satisfy the vp-condition, we roll back the insertion of J. Otherwise, we compute the negative border of (the updated) AFI. Itemsets in this negative border can be added to AII (if not already there). We check the vp-condition while adding each of them. If that is not satisfied, we roll back that insertion.

[3]If p_{owner} wishes to perform mining with various support thresholds (which would result in various numbers of frequent itemsets), he should generate the AFI and AII using the minimum of these support thresholds, as the AFI and AII generated for lower thresholds include the corresponding sets generated for a higher threshold.

[4]The initial size of I_A is not critical, since our procedure will dynamically adjust it. A reasonable initial size would be the size of the largest itemset in the estimated $B^-(L)$.

In the above procedure, if J is successfully added to AII, we generate another J from I_A and repeat the steps. On the other hand, if the insertion of J is rolled back, we create a new artificial item α, put α in I_A, replace an "old" item in J by α, and attempt to insert J into AII again using the above procedure. The reason for such a replacement strategy is that if $|J| = k$, then in the worst case, after k attempts of inserting J into AII, J will be composed of purely new items. In that case, inserting J into AII will not violate the vp-condition and the insertion is guaranteed to be successful. The replacement strategy thus ensures that the construction procedure terminates within a finite amount of time.

4.4 Database generation

Given a valid itemset pattern (AFI, AII), the next step is to generate an artificial database $\hat{T}$ such that all itemsets in AFI are frequent and all itemsets in AII are infrequent. A simple approach to generate such a database is to follow the baseline construction method described in the proof of Theorem 3. However, such a database has the special property that all itemsets in AII have 0 supports and the supports of the itemsets in AFI are very close to the support threshold. This is undesirable because a malicious miner might deduce the artificial items and eliminate the chance of being detected, by avoiding to change their supports.

To improve the robustness of the audit environment, we add more randomness in the generation of an artificial database. In particular, itemsets in AII could be given small, but non-zero supports. The supports of itemsets in AFI are also given more variation. In this subsection, we describe one such artificial database generation method.

We start with a few definitions. Each itemset $x \in AFI$ is associated with a weight, denoted by $w(x)$. Intuitively, $w(x)$ indicates the minimum number of transactions in $\hat{T}$ that should contain x. So, $w(x) = |\hat{T}| \cdot s\%$ because there have to be at least $|\hat{T}| \cdot s\%$ transactions in $\hat{T}$ that contain x for x to be frequent.

Definition 8. *(Weighted conflict index) Given a conflict graph $G = (V, E)$ and a weight function $w()$, let $N(x)$ denote the set of neighbors of vertex x. The weighted conflict index wc_x of x is the sum of the weight of x and the total weights of its neighbors, i.e., $wc_x = w(x) + \sum_{y \in N(x)} w(y)$. The weighted conflict index of G, denoted by $WCI(G)$, is $max_{x \in V} wc_x$.*

Theorem 4. *Given an itemset pattern (AFI, AII), a weight function $w()$, and an integer n, there exists an artificial database $\hat{T}$ of n transactions such that (1) for each $x \in AFI$, the support of $x \geq w(x)$ and (2) all itemsets in AII have 0 supports, if both of the following conditions hold:*

1. *AFI and AII satisfy the monotonicity property.*
2. *$WCI(G) \leq n$.*

Proof. We give a sketch of a proof that is very similar to the construction proof we described in Theorem 3. Similar to the baseline construction method, we partition AFI into Q^+ and Q^-. A transaction $\bigcup_{x \in Q^+} x$ is generated. The weight function is updated to $w'()$ as follows: $w'(x) = w(x) - 1, \forall x \in Q^+$; $w'(x) = w(x), \forall x \in Q^-$. That is, the weight of each itemset included in the generated transaction is reduced by 1. We update the conflict graph $G = (V, E)$ to $G' = (V', E')$ such that all vertices x with $w'(x) = 0$ are removed from G together with all their associated edges. Also, denote the weighted conflict index of any $x \in G'$ by wc'_x. We note that for each $x \in V'$, if $x \in Q^+$, then all its neighbors must be in Q^-. Since $w'(x) = w(x) - 1$ and the weights of all x's neighbors are unchanged, we have $wc'_x = wc_x - 1$. Moreover, if $x \in Q^-$, then there must exist at least one neighbor y of x such that $y \in Q^+$. Since $w'(y) = w(y) - 1$, we have $wc'_x \leq wc_x - 1$. As a result, $\mathit{WCI}(G') \leq \mathit{WCI}(G) - 1$.

We repeat this process of transaction generation. For each transaction generated, the weighted conflict index of the graph is reduced by at least 1. Eventually, the conflict graph is reduced to the null graph, after at most $\mathit{WCI}(G)$ transactions have been generated. Since each itemset $x \in \mathit{AFI}$ has its weight reduced from $w(x)$ to 0 in the process, $w(x)$ transactions that contain x must have been generated. If $\mathit{WCI}(G) \leq n$, an artificial database of n transactions that satisfies the minimum support requirement can be obtained by taking all the generated transactions and replicate some of them until we get n transactions. $\square$

We now briefly describe an algorithm for generating an artificial database $\hat{T}$ such that itemsets in AII could have nonzero (but infrequent) supports, and the itemsets in AFI are frequent with a wider variation of support counts. We highlight the important steps; a detailed pseudo code is listed in the Appendix. We assume that AFI and AII satisfy the monotonicity property.

First, for each $x \in \mathit{AFI}$, we set $w(x) = n \cdot s\%$ where n is the number of artificial transactions to be generated. Also, for each $y \in \mathit{AII}$, we set a quota, $q_y < n \cdot s\%$. Intuitively, q_y specifies how many generated transactions can contain y at most. We randomly pick an itemset $z_1 \in \mathit{AFI}$ and randomly pick a number of other items in I_A, say $z_2 \subset I_A$, to form a transaction $\hat{t} = z_1 \cup z_2$. For each $x \in \mathit{AFI}$, if $x \subseteq \hat{t}$, we reduce its weight, $w(x)$, by 1. For each $y \in \mathit{AII}$, if $y \subseteq \hat{t}$, we reduce its quota, q_y, by 1. If $q_y < 0$, we know that taking $\hat{t}$ will cause some AII itemset to be frequent, so transaction $\hat{t}$ is discarded. Otherwise, we check the condition ($\mathit{WCI}(G) \leq n - 1$) with respect to ($\mathit{AFI}$, AII, (updated) $w()$, $n-1$). If the condition is satisfied, then by Theorem 4, we know that it is possible to generate a database that, together with $\hat{t}$, satisfies all the support constraints. We thus include $\hat{t}$ in $\hat{T}$ and repeat the above process. On the other hand, if the condition is not satisfied, we discard $\hat{t}$ and generate another transaction. When a generated transaction $\hat{t}$ is inserted to $\hat{T}$, we increment the support count of each subset u of $\hat{t}$ if $u \in \mathit{AFI}$ or u is a subset of an itemset that is in AFI.

To ensure that the procedure terminates in a finite amount of time, we use a control parameter b. If we have discarded transactions b consecutive times without successfully generating one, we fall back to the baseline construction method to generate the next transaction.

After the database generation concludes, our audit environment consists of (i) AII, (ii) AFI, and (iii) the support counts of all itemsets in AFI and their subsets. The latter set is used to verify whether the supports of returned itemsets are not modified by a malicious action of p_{miner}.

4.5 Security and cost analysis

In this section, we analyze the effectiveness of AIP in guarding against malicious actions by p_{miner} and the computational cost of applying AIP at p_{owner}.

Due to the random generation of transactions in the artificial database, the supports of artificial itemsets vary and follow a similar distribution as the supports of the original itemsets. Therefore, p_{miner} is expected not to be able to distinguish between original itemsets and artificial ones in the outsourced database. As a result, the malicious actions performed by p_{miner} (described in Section 3.1) may apply to artificial and/or actual itemsets.

Suppose p_{miner} performs a malicious action on an itemset x; x may be (i) an itemset in the original database; or (ii) an itemset in AFI or AII; or (iii) an itemset that is neither from the original database nor in $\mathit{AFI} \cup \mathit{AII}$ (e.g., x contains both original as well as artificial items). Our audit environment will fail to detect actions on type-(i) itemsets. In addition, p_{miner}'s gain on such actions will be positive, since they will affect the mining result of the original database. On the other hand, p_{miner}'s actions on type-(ii) and type-(iii) itemsets do not affect the actual results and bring no gain to him. Moreover, if x is of type (ii), the action can be detected by our audit environment and p_{miner} may be caught and penalized. Let the gain $\phi(A_i)$ by a malicious action A_i be $h > 0$ if A_i is performed on a type-(i) itemset. Note that $\phi(A_i) = 0$ for actions on any itemset of another type. For simplicity, we assume no malicious actions are performed on type-(iii) itemsets, since p_{miner} does not gain from such actions and the actions cannot be detected. Let $m = |L \bigcup B^-(L)|$, where L is the true set of frequent itemsets in the original database (i.e., type-(i) itemsets). Let n be the number of type-(ii) itemsets. If p_{miner} performs j malicious actions and returns R', the probability p of being caught is equal to the probability that he picks at least one of the n balls in a set of $m + n$ balls. So, $p = 1 - \Pi_{i=0}^{j-1} \frac{m-i}{m+n-i} = 1 - \frac{m!}{(m+n)!} \times \frac{(m+n-j)!}{(m-j)!}$. If p_{miner} is not caught (by not picking any of the n balls), the expected gain is jh. So, $EG(R') = jh(1-p) - p\rho$. If $EG(R')$ is negative for all values of j and R', the malicious miner is expected to lose. Therefore, p_{miner} is forced to act honestly and returns the correct and complete results. Using this analysis as a guideline, we can derive the required number of artificial itemsets to be planted in order to protect the mining result. In Section 5, we perform an experimental security analysis and demonstrate that AIP is very effective in practice.

The cost of AIP at p_{owner} consists of three parts:

a. Itemset pattern generation. The dominating cost factor in itemset pattern generation is the maintenance of the conflict graph. When an AII itemset is added, we also add its immediate subsets to AFI (those that are not already there). Then, for every pair of itemsets in the updated AFI, which are not already in conflict, we need to check whether they are now in conflict due to the insertion of the new AII itemset. There are $|\mathit{AFI}|^2$ such pairs in the worst case. Therefore each AII itemset insertion costs $O(|\mathit{AFI}|^2)$ and the total cost of the itemset pattern generation phase is $O(|\mathit{AII}| \times |\mathit{AFI}|^2)$. Despite this seemingly large complexity, the generation process is independent of database size and it is expected to be cheap compared to database scans for small AII and AFI. Our experiments (see Section 5) show that this cost is indeed insignificant.

b. Database generation. When a transaction $\hat{t}$ is generated, we have to update the quotas (weights) of all AII (AFI) itemsets that are included in $\hat{t}$. This requires $O(|\mathit{AFI}| + |\mathit{AII}|)$ time. In addition, for each such AFI itemset y, we need to decrement the weighted conflict index wc_x for each neighbor x of y in the conflict graph. In the worst case, there are $1/s\%$ such neighbors. Therefore the cost of generating $\hat{t}$ is $O(\frac{|AFI|}{s\%} + |AII|)$. In the worst case, b unsuccessful trials could be attempted before a transaction $\hat{t}$ is successfully generated. Hence, the maximum number of transactions tested is $b \times |\hat{T}|$. Overall, the cost of generating $\hat{T}$ is $O(b \times (\frac{|AFI|}{s\%} + |AII|)|\hat{T}|)$. We remark that the bounds mentioned about in our worst-case analysis are very loose. Also, we will discuss an optimization method in Section 4.6 that greatly reduces the database generation time. As we will see later in our experimental results, the database generation time is much smaller than the mining time in practice.

c. Detection of malicious actions. The owner detects malicious actions by (i) checking whether any *AII* itemsets are returned by the miner as frequent and (ii) for all itemsets in *AFI* and the subsets thereof, comparing the support counts given by p_{miner} with the stored counts prepared in the audit environment, during the database generation phase. The total cost of this phase is $O(k)$, where k is equal to the number of *AII* itemsets plus the number of support counts recorded in the audit environment. Again, our experimental results show that this verification cost is small.

4.6 Reducing the cost of database generation

In Section 4.4 we discussed how to generate an artificial database $\hat{T}$. The number of transactions generated $|\hat{T}|$ equals the size of the original database T. We remark that it is not necessary to generate such a large number of artificial transactions. Recall that the requirement of $\hat{T}$ is to ensure that all AFI itemsets are frequent while all AII itemsets are infrequent. A more efficient way to generate $\hat{T}$ is to generate a smaller database $\widehat{T_D}$ that satisfies the AFI and AII constraints and replicate $\widehat{T_D}$ to obtain $|\hat{T}|$ artificial transactions. For example, we can generate a $\widehat{T_D}$ of 1,000 transactions, replicate it 100 times to obtain a $\hat{T}$ of 100,000 transactions. A minor problem of this method is that the support counts of artificial itemsets would all be multiple of $|\hat{T}|/|\widehat{T_D}|$. To avoid frequency attack, we add variability to the support counts. This can be achieved by generating another small database $\widehat{T_V}$ that satisfies the AFI and AII constraints. Database $\hat{T}$ is then obtained by replicating $\widehat{T_D}$ a number of times followed by adding the transactions in $\widehat{T_V}$. With this approach, we are generating two *small* databases $\widehat{T_D}$ and $\widehat{T_V}$ instead of a large one $\hat{T}$. The database generation process is thus much faster.

An interesting issue is how to pick the sizes of $\widehat{T_D}$ and $\widehat{T_V}$. Let r be the number of times $\widehat{T_D}$ is replicated. We have

$$|\widehat{T_D}| \times r + |\widehat{T_V}| = |T| \qquad (1)$$

Since the purpose of $\widehat{T_V}$ is to inject variations to the support counts (which are originally all multiples of r), ideally, we want the support counts of the itemsets found in $\widehat{T_V}$ to cover at least the range $[1 \dots r]$. An easy way to ensure that is to make r smaller than the support count threshold of $\widehat{T_V}$. So if we consider the itemsets in $\widehat{T_V}$ (which include those frequent ones), the support counts can cover the range $[1 \dots r]$. Hence, we set

$$|\widehat{T_V}| \times s\% \geq r \qquad (2)$$

Substituting Eq. 2 into Eq. 1, we get $|\widehat{T_V}|(1 + s\%|\widehat{T_D}|) \geq$

length of itemset	$s = 1$		$s = 2$		$s = 3$	
	$\|L_i\|$	$\|B_i^-(L)\|$	$\|L_i\|$	$\|B_i^-(L)\|$	$\|L_i\|$	$\|B_i^-(L)\|$
1	310.0	690.0	179.8	820.2	99.0	901.0
2	590.6	47305.8	136.6	15937.6	38.8	4812.2
3	664.6	807.4	110.8	44.2	19.0	8.0
4	383.2	127.0	60.6	2.4	8.0	0.0
5	156.0	3.4	14.0	6.0	5.6	0.0
6	44.2	3.4	0.8	0.2	0.0	0.0
7	4.2	0.2	0.0	0.0	0.0	0.0
Total	2152.8	48937.2	502.6	16810.2	170.0	5721.2

Table 1: Average values of $|L_i|$ and $|B_i^-(L)|$ under different support threshold (s%)

$|T|$. Therefore, determining $|\widehat{T_D}|$ and $|\widehat{T_V}|$ becomes a constraint optimization problem with the objective of minimizing $|\widehat{T_D}| + |\widehat{T_V}|$ (i.e., the total number of transactions to be generated). For example, if $|T| = 1M$ and $s = 5$, the optimal solution is $|\widehat{T_D}| = 5000$ and $|\widehat{T_V}| = 5000$ for an integer r.

5. EXPERIMENTAL EVALUATION

In this section we evaluate AIP empirically. We study its effectiveness in detecting malicious actions and the cost they induce to both the data owner and the data miner. We implemented all the programs for AIP using C++. Experiments were performed on an Intel Core 2 Duo 2.66GHz computer with 2 GB RAM running Windows.

5.1 Settings

In the experiments, we generated 5 transactional databases using the IBM data generator [6] with the same set of parameters ($|I| = 1000$, average transaction length $|t| = 10$). The databases differ in size, from 100k transactions to 500k transactions. Since the same set of parameters are used in generating the databases, the different databases have similar numbers of frequent itemsets ($|L|$) and similar sizes of their negative borders ($|B^-(L)|$). Table 1 shows the average number of length-i frequent itemsets, denoted by $|L_i|$ and the average number of length-i itemsets that are in the negative border, denoted by $|B_i^-(L)|$, for the 5 databases under 3 different support thresholds ($s = 1\%, 2\%, 3\%$).

As we have discussed, in AIP, we need to provide a rough estimate of the sizes of *AFI* and *AII* (in order to generate *AFI* and *AII*). In our experiment, we set $|AFI| = v \cdot |L|$ and $|AII| = v \cdot |B^-(L)|$, for some fractional value v.

5.2 Effectiveness in detecting malicious actions

We first study the probability that a malicious miner is detected/caught by AIP. If the miner returns an accurate result L, a perfect verifier will have to check the support counts of all itemsets in $L \cup B^-(L)$ (see Section 4). So, if the miner performs $e \cdot (|L| + |B^-(L)|)$ malicious actions, loosely speaking, the miner is perturbing a fraction e of the result. In our first experiment, the miner randomly performs $e \cdot (|L| + |B^-(L)|)$ malicious actions. We apply AIP to verify the result and take note of whether a malicious act is detected. We repeat this experiment 5,000 times and record the probability (p) that the malicious miner is caught by AIP over the 5,000 sample runs. Figure 3 plots this probability against e for v ranges from 0.5% to 3%. In this experiment, we set $s = 1$ and $|T| = 100$k.

From the figure, we see that p increases with e — the more perturbation done, the more likely a malicious miner

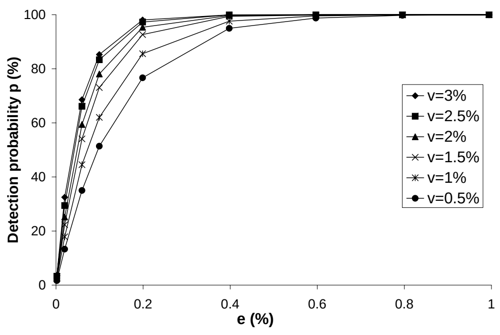

Figure 3: Probability that a malicious miner is caught (p) vs. e

is caught. Also, a larger v (i.e., more AFI and AII itemsets are used for verification) gives a larger p. Moreover, the detection probability p is almost 100% for all v values even when the miner has perturbed as little as $e = 0.6\%$ of the result. The following 1%-1% rule: "By verifying 1% of the result ($v = 1\%$), a malicious miner that has perturbed more than 1% of the result ($e > 1\%$) is almost always caught," can be seen as a conservative statement on the effectiveness of AIP in this experiment.

Recall that in Section 2 we define the expected gain (EG) of a malicious miner. An interesting question is what Figure 3 can tell us about such expected gains. Let g be the gain obtained by the miner for each malicious action performed and ρ be the penalty suffered by the miner if it gets caught. If the miner performs N malicious actions, we have $EG = (1-p)Ng - p\rho$. In order for such malicious acts to be profitable, we need $EG > 0$, which implies $\frac{\rho}{g} < N \cdot \frac{1-p}{p}$. Now consider Figure 3. Given e, we get $N = e \cdot (|L| + |B^-(L)|)$. For a given v, the corresponding curve in Figure 3 gives us a p value. For example, in our experiment, with $e = 0.4\%$ and $v = 1\%$, we get $N = 200$ and $p = 0.976$. Hence, $N \cdot \frac{1-p}{p} = 4.92$. In other words, the gain per each malicious act has to be at least $\frac{1}{4.92}$ of the penalty suffered in order for $EG > 0$. However, as we have argued, ρ should be much much larger than g in practice. Therefore, under AIP, malicious actions are simply non-profitable. Result integrity can thus be strongly enforced.

5.3 Cost analysis

We study the efficiency of AIP. In particular, we study the cost of generating itemset patterns, the cost of generating an artificial database, the cost of verification, and the cost of the miner in mining a transformed (and larger) database. First, Table 2 shows the execution time of the classic Apriori algorithm when applied to our databases under different support thresholds[5]. We remark that any practical verification scheme should not cost the data owner more time than those listed in the table.

Generation of a valid pattern Section 4.3 described our algorithm for generating a valid pattern (AFI, AII). Figure 4 shows the execution time of the algorithm as v changes from 0.5% to 3%. Three lines are shown corresponding to three support thresholds.

From the figure, we see that as v increases, the time taken to generate a valid pattern becomes longer. This is because a larger v implies a larger AFI and a larger AII. More itemsets have to be generated and that takes longer. Also, generating itemsets when AFI and AII are already big is harder. This causes more rollbacks and retries during the generation process. In any case, the pattern generation time is very small compared with the mining time (Table 2). For example, when $s = 1\%$ and $v = 3$, pattern generation takes about 2 seconds. The execution time is negligible for higher support thresholds.

Generation of an artificial database Given a valid pattern (AFI, AII) we generate an artificial database. Section 4.4 described our basic algorithm for generating artificial transactions and Section 4.6 described an optimization that generates two small databases instead of a big one. Figure 5 shows the database generation time using the optimized method under different combinations of v and database sizes $|T|$. In this experiment, the support threshold is 2%.

From the figure, we observe that a larger v causes the

[5]We use Apriori here just to illustrate the typical mining times if the data owner chooses to perform mining itself using off-the-shelf packages instead of outsourcing the task. Other more efficient mining algorithms can also be applied. For the latter case, the numbers shown in Table 2 will be smaller, although we expect that the numbers will be of similar magnitude.

support threshold	Database size 100k	200k	300k	400k	500k
1%	186.6s	383.8s	569.1s	761.9s	944.3s
2%	67.3s	135.7s	203.5s	271.5s	339.3s
3%	24.8s	49.5s	74.2s	98.9s	123.6s

Table 2: Execution time of Apriori

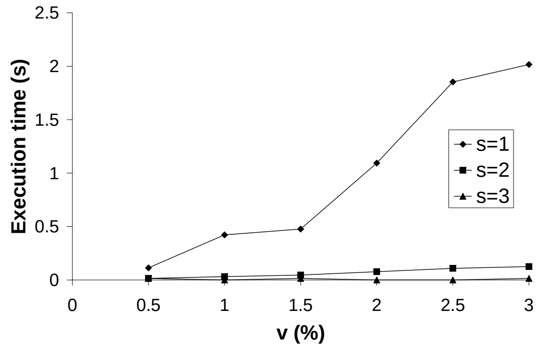

Figure 4: Time taken to generate a valid pattern

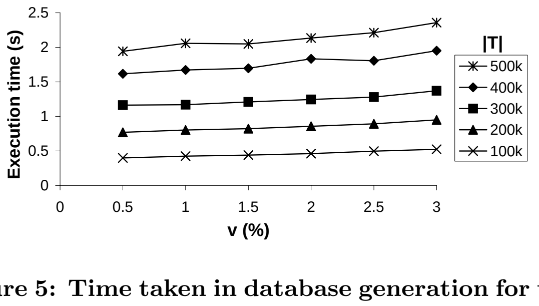

Figure 5: Time taken in database generation for various v and database sizes; $s = 2$

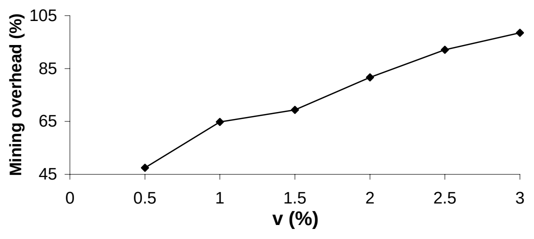

Figure 6: Mining overhead for various v; $s = 2$ and $|T| = 500k$

database generation process to take slightly more time. This is because a larger v means larger AFI and AII. The weighted conflict graph G is thus larger. The cost of updating G and checking the condition $WCI(G) \leq n$ (see Section 4.4) is slightly higher.

Also, the larger the database is, the higher is the generation time. This is because a larger $|T|$ implies a larger $|T_D|$ and a larger $|T_V|$. So the two small databases we generate are bigger, leading to a longer generation time.

Again, we have argued that $v = 1\%$ is generally good enough to achieve a high detection probability. In that case, Figure 5 shows that database generation takes about 2 seconds to complete (even for a database of 500k transactions). Compare that to the numbers shown in the second row of Table 2 (for support threshold = 2%), the cost of generating a database is relatively insignificant.

Verification Given a returned result L', the verification process verifies the support counts of itemsets in $(AFI \cup AII) \cap (L' \cup B^{-}(L'))$ by comparing the stored count values against those returned in the result. In our experiment, it takes less than 1 ms.

Mining overheads at service provider Next, we study how much additional mining time the miner has to pay. Note that the miner has to mine a larger (horizontally extended) database with additional artificial items. We compute the ratio of the *additional* mining time with AIP and the mining time without AIP[6]. Figure 6 shows this ratio for the case $|T|$ = 500k and $s = 2$. For example, the figure shows that when $v = 2\%$, the additional mining overhead is about 80% of the original mining time; for $v = 1\%$, the overhead is reduced to about 65%. In the latter case, the average transaction size increases from 10.08 items per transaction in the original database to 12.26 items in the database submitted to the miner; the database size increases from 24.6MB to 28.7MB; and the number of frequent itemsets increases from 501 to 639. In general, a larger v leads to a higher mining overhead at p_{miner}. This is because a larger v causes more artificial items and itemsets to be included in the database. The mining time is thus higher. We remark that this overhead should be manageable by the service provider.

6. CONCLUSIONS AND FUTURE WORK

In this paper we put forward the integrity problem in outsourcing the task of frequent itemset mining. We established a formal framework of the problem with a definition of a set of malicious actions that a malicious service provider might perform. We have shown theoretically that a full verification of the correctness and completeness of a mined result requires examination of the support counts of all itemsets in the result plus those in the negative border of the result. The high cost of such verifications leads to the development of a sampling approach under the concept of an audit environment. We proposed the *artificial itemset planting* technique for preparing an audit environment. We explained how AIP works through a set of theorems. A malicious miner cannot benefit from performing any of the malicious actions and thus the returned mining result is both correct and complete with a high confidence. We evaluated our algorithms through a series of experiments. Our technique is shown to be both effective and efficient at p_{owner}. In the future, we will work on reducing the mining overhead and make it a controllable factor by integrating AIP and sampling techniques on original database, which do not inject additional data and hence do not bring in additional mining effort at p_{miner}.

[6]We assume p_{miner} uses the Apriori algorithm in this experiment. We remark that AIP is independent of the mining algorithm the miner uses. We expect that the overhead induced by AIP is mostly dependent on the additional itemsets and artificial transactions introduced by AIP, instead of on the mining algorithm employed.

7. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB*, 1994.

[2] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *SIGMOD*, 1997.

[3] X. Chen and M. Orlowska. A further study on inverse frequent set mining. In *ADMA*, 2005.

[4] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *FIMI*, 2003.

[5] H. Hacigumus, B. Iyer, and S. Mehrotra. Providing database as a service. In *ICDE*, 2002.

[6] IBM Almaden Research Center. Synthetic data generation code for association and sequential patterns.

[7] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD*, 2007.

[8] F. Li, K. Yi, M. Hadjieleftheriou, and G. Kollios. Proof-infused streams: Enabling authentication of sliding window queries on streams. In *VLDB*, 2007.

[9] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, 2002.

[10] T. Mielikainen. On inverse frequent set mining. In *PPDM*, 2003.

[11] H. Pang, A. Jain, K. Ramamritham, and K.-L. Tan. Verifying completeness of relational query results in data publishing. In *SIGMOD*, 2005.

[12] S. Papadopoulos, Y. Yang, and D. Papadias. CADS: Continuous authentication on data streams. In *VLDB*, 2007.

[13] G. Ramesh, W. A. Maniatty, and M. J. Zaki. Feasible itemset distributions in data mining: theory and application. In *PODS*, 2003.

[14] G. Ramesh, M. J. Zaki, and W. A. Maniatty. Distribution-based synthetic database generation techniques for itemset mining. In *IDEAS*, 2005.

[15] R. Sion. Query execution assurance for outsourced databases. In *VLDB*, 2006.

[16] W. K. Wong, D. W. Cheung, E. Hung, B. Kao, and N. Mamoulis. Security in outsourcing of association rule mining. In *VLDB*, 2007.

[17] M. Xie, H. Wang, J. Yin, and X. Meng. Integrity auditing of outsourced data. In *VLDB*, 2007.

APPENDIX

Algorithm 1 Pseudo code of database generation

```
Require: $AFI$, $AII$ : valid itemset pattern
Require: $I_A$ : set of items in the artificial database
Require: $b$ : the maximum number of random generation
Require: $n$ : number of transactions to generate
Require: $s\%$ : support threshold
  {Initialization}
  for each itemset $x$ in $AFI$ do
    $w(x) = n \cdot s\%$
  end for
  for each itemset $x$ in $AII$ do
    $q_x = r$ where $r < n \cdot s\%$
  end for
  {Generate transaction}
  for $i = 1$ to $n$ do
    $j = 1$
    repeat
      if $j > 1$ then
        roll back updates of $w(x)$ and $q_x$
      end if
      if $j > b$ then
        generate $\hat{t}$ by baseline construction
      else
        $r$ = random integer in $[0, |AFI|]$
        if $r = 0$ then
          $z_1 = \emptyset$
        else
          $z_1 = AFI[r]$ {$AFI[r]$ is the $r$-th itemset in $AFI$}
        end if
        $z_2$ = random subset of $I_A$
        $\hat{t} = z_1 \cup z_2$
      end if
      {Update weights and quotas}
      $j++$
      for each itemset $x \in AII$ and $x \subseteq \hat{t}$ do
        if $q_x = 0$ then
          go to next iteration {$x$ is made frequent}
        else
          $q_x --$
        end if
      end for
      for each itemset $x \in AFI$ and $x \subseteq \hat{t}$ do
        $q_x = max(0, q_x - 1)$
      end for
    until $WCI(G) < n - i$
  end for
```

Algorithm 2 Pseudo code of valid pattern generation

```
Require: $f_{AFI}[i]$ : a size $m$ array specifying the target number of size $i$ itemset in $AFI$
Require: $f_{AII}[i]$ : a size $n$ array specifying the target number of size $i$ itemset in $AII$
Require: $s\%$ : support threshold
  {Initialization}
  $AII = AFI = \emptyset$, $G = (\emptyset, \emptyset)$
  $I_A$ = a set of $n$ items
  {Generate $AII$}
  for $i = n$ to 1 do
    $count = |\{x \mid x \in AII$ and $|x| = i\}|$
    while $count < f_{AII}[i]$ do
      generate an itemset $x$ by randomly pick $i$ items in $I_A$ such that $x \notin AII$
      $AII = AII \bigcup \{x\}$
      $j = 1$
      $Y = \{y \mid y \subset x$ and $|y| = i - 1$ and $\forall z \in AFI, y \not\subseteq z\}$
      $AFI = AFI \bigcup Y$
      update $G$ accordingly
      while vp-condition is not satisfied do
        roll back adding of $x$ to $AII$ and $Y$ to $AFI$
        generate a new item $p$
        $x[j] = p$ {$x[j]$ is the $j$-th item in $x$}
        $AII = AII \bigcup \{x\}$
        $I_A = I_A \bigcup \{p\}$
        $Y = \{y \mid y \subset x$ and $|y| = i - 1$ and $\forall z \in AFI, y \not\subseteq z\}$
        $AFI = AFI \bigcup Y$
        update $G$ accordingly, $j++$
      end while
      $count++$
      {We include the new members in $B^-(AFI)$ to $AII$}
      $\Delta B^-(AFI) = \{z \mid \forall s \subset z, \exists w \in AFI, s \subseteq w$ and $\exists s \subset z, \forall w \in AFI - Y, s \not\subseteq w\}$
      for each itemset $z \in \Delta B^-(AFI)$ do
        $AII = AII \bigcup \{z\}$
        if vp-condition is not satisfied then
          roll back adding of $z$
        else
          if $|z| = i$ then
            $count++$
          end if
        end if
      end for
    end while
  end for
  {Generate $AFI$}
  for $i = m$ to 1 do
    $count = |\{x \mid \exists y \in AFI, x \subseteq y$ and $|x| = i\}|$
    while $count < f_{AFI}[i]$ do
      generate an itemset $x$ by randomly pick $i$ items in $I_A$ such that $\forall y \in AFI, x \not\subseteq y$
      $AFI = AFI \bigcup \{x\}$
      $j = 1$
      update $G$ accordingly
      while vp-condition is not satisfied do
        roll back adding of $x$ to $AFI$
        generate a new item $p$
        $x[j] = p$
        $AFI = AFI \bigcup \{x\}$
        $I_A = I_A \bigcup \{p\}$
        update $G$ accordingly, $j++$
      end while
      $count++$
    end while
  end for
```