Provenance Management for Frequent Itemsets

Javed Siddique
University of Toronto
jsiddique@cs.toronto.edu

Boris Glavic
Illinois Institute of Technology
bglavic@iit.edu

Renée J. Miller
University of Toronto
miller@cs.toronto.edu

ABSTRACT

Provenance has been studied extensively for relational queries and shown to be important in revealing the origin and creation process of data that has been produced by potentially complex relational transformations. Provenance for the results of data mining operators in contrast has not been considered. We argue that provenance offers the same benefits for mining as for relational queries, e.g., it allows us to track errors caused by incorrect input data. We consider the most common mining operator, frequent itemset mining, and introduce two types of provenance (*why-* and *i-provenance*) for this operator. We argue that the concept of *why-provenance* for relational queries can be adapted for frequent itemsets, but that it poses new computational challenges due to the nature of itemset mining and the size of why-provenance. We address these challenges in two ways. First, we propose combining *why-provenance* computation with SQL querying to permit users to select small and more intuitive representations of the provenance, and second by proposing new compression techniques for the why-provenance. Next, we introduce a new provenance type called *i-provenance* (itemset provenance) that succinctly represents the interdependencies between items and transactions that explain how a frequent itemset was derived (intuitively giving insight into the structure of the data that provides the evidence for the itemset). We present techniques for efficient storage and use of both types of provenance information and experimentally evaluate the scalability of our approach. We argue through a set of examples that why- and i-provenance can add significant value to mining results and can be used to analyze the context of the transactions that caused an itemset to be frequent and to understand how combinations of itemsets contribute to a result.

1. INTRODUCTION

Frequent itemset mining (*FIM*) is a popular data mining method that is used for association rule mining [5], correlation analysis, finding functional dependencies [23], classification, clustering [28], and many other application. The input to frequent itemset mining is a set of transactions D. Each transaction is a set of items from some domain $\mathbb{D}$ of items, e.g., modelling the items bought in one customer order. Given a *minimum support* threshold σ, FIM returns all itemsets I (subsets of $\mathbb{D}$) that appear in a fraction larger than σ of the transactions. Many efficient algorithms [5, 17, 22] have been developed for mining frequent itemsets. While some work has considered techniques for visualizing mining results [13] or evaluating their interestingness [25], there are no tools that compute mining provenance, that is, the reasons for why and how a certain result was produced.

A large body of work has studied provenance generation and management for relational data [8] and workflows [12], but to the best of our knowledge, provenance for frequent itemsets has not been explored. In this paper, we define two types of provenance for frequent itemsets and show how they can be generated and managed inside a DBMS.

EXAMPLE 1. *A popular example in the frequent itemset mining literature is a survey [3] studying the behavior of young supermarket shoppers. This survey noted that* {*Diaper, Beer*} *is a frequent itemset, i.e., is often bought together. Though* {*Diaper, Beer*} *may be interesting as it is an unexpected result, to understand this result we may want to understand* why *it is frequent. Adopting a common definition of* why-provenance *for relational queries, we could say it is frequent because it appears in a specific set of perhaps one million transactions. But this set alone does not give us much insight. The actual source of the* {*Diaper, Beer*} *story [3] cites that this frequent itemset is based on analyzing data from only 25 OSCO drug stores, between 5:00PM and 7:00PM. This information is part of the provenance information for the* {*Diaper, Beer*} *itemset as it describes (more intuitively to a human) the set of transactions which contributed to making* {*Diaper, Beer*} *frequent. Being able to compute the why-provenance, and importantly declaratively retrieve contextual information about the transactions in the why-provenance (such as the stores in which the transactions took place), can add tremendous value to the mining results.*

1.1 Why-Provenance

In the example, we made use of a simple but intuitive notion of why-provenance[1] for relational queries which defines provenance as the set of tuples that *contributed* to a query

[1]We use why-provenance as an umbrella term for several related relational provenance types including why-provenance [7], lineage [10], provenance polynomials [24], and Perm-Influence contribution semantics [14].

Proceedings of the VLDB Endowment, Vol. X, No. Y

result, and we have adapted this definition to say that the why-provenance of an itemset is the set of transactions that contribute to it. For relational queries, the definition of *contribution* can be quite subtle (since queries permit tuples to be transformed in complex, non-monotone ways and may use tuples disjunctively) and hence several different semantics have been studied in the literature [8].[2] However, for itemsets we will use a very simple, intuitive definition, a transaction *contributes* to an itemset if it supports (includes) that itemset. Why-provenance has proven to be a very powerful notion in relational DBMS. It has been used to debug data in data warehouses [10], to understand and correct complex data integration transformations [15], to propagate deletions and compute trust in peer-to-peer data exchange [19], and to understand the value of data in curated databases [6]. However, its value in FIM is not as obvious. First, by the very nature of FIM, the why-provenance will be massive. Second, unlike in relational queries, tuples (transactions) are not being combined and transformed in complex ways in FIM. So returning the (large) set of contributing transactions to a user does not provide any "hidden" insight that it might, and often does, for a query. Nonetheless, we argue that why-provenance can still play an important role. First, if we have access to contextual information about transactions (such as information on the store, the time of purchase, the customer, or the credit card), then we can use this information *in lieu* of transaction IDs to represent the why-provenance more compactly and informatively.

EXAMPLE 2. *Our system will let a user ask for the why-provenance of an itemset and also specify how the provenance is described. For example, a user may ask for the set of IDs for the transactions supporting the {Diaper,Beer} itemset. Alternatively, she may ask for the set of store locations for these transactions and request that each store ID be returned with a count of the number of transactions from that store in the why-provenance. In this example, the contextual information (store locations) is a more concise representation of provenance than the actual transactions, because the number of store locations can be expected to be much smaller than the number of transactions in the why-provenance.*

Furthermore, we can ask queries about the itemsets and their provenance together.

EXAMPLE 3. *Continuing our example, suppose we have requested the time a transaction took place in addition to the transaction ID as why-provenance. Running a query over this data (in our system the why-provenance computation could actually be expressed as part of a query) we can retrieve the number of transactions in each 2-hour time-window from the why-provenance of the {Diaper,Beer} itemset. We can continue our investigation to find out how many transactions in the why-provenance per time window contain at most one additional item other than diapers and beer, and if other itemsets containing beer follow the same trend. For example, we may realize that 5pm-7pm is simply a popular shopping time or that {Beer,Diaper} is an isolated phenomenon. Thus, by asking such queries we are able to unearth interesting facts about an itemset and its connection to other itemsets. This type of information is certainly not available in the mining result itself or in the set of transaction IDs constituting the why-provenance.*

We have argued that why-provenance can provide meaningful insight in FIM. However, there are serious computational challenges to be overcome. The why-provenance for a frequent itemset is by definition large. Furthermore, considering all different types of information that may be related to a transaction in each why-provenance computation is prohibitively expensive. Furthermore, a user often knows what information is more relevant to her to give insight into the results. We therefore propose a solution where provenance is computed **on-demand** within a DBMS. The provenance does not have to be materialized, and the user can decide on a case-by-case basis (using a declarative specification) what contextual information should be linked to the transactions in the why-provenance. When a user does want to materialize a specific variant of the provenance (for example, to do complex analytic queries such as the one above that compare the provenance of multiple itemsets), we present compression techniques that permits the efficient storage of provenance and the efficient use of compressed provenance within queries.

1.2 I-Provenance

Why-provenance provides insight into what transactions contribute to making an itemset frequent. A different type of provenance, that has been studied for relational databases and workflows, models which part of a process (query, workflow, ...) generated a specific result. We refer to this type of provenance as *process* provenance. Many provenance-enabled workflow systems [12] track which workflow steps were involved in producing a result. This type of provenance has been adapted for databases under the name *transformation provenance* [15]. A related form of provenance, sometimes called *how provenance*, is provenance polynomials [20].[3] For positive relational algebra queries, these polynomials record disjunctive (union) and conjunctive (join) use of input tuples in the derivation of a result tuple. For itemsets, the inputs (transactions) are not combined in different and complex ways to produce a result frequent itemset. Thus, a straight-forward adaption of relational or workflow process provenance models would not add value over the why-provenance for itemsets. However, the intuition of process provenance, describing *how* a result is derived from the inputs, can still be applied. For itemsets we would like to explain the structure of the data itself, i.e., to give insight into how the interdependencies between items and transactions supporting an itemset (the why-provenance) are actually giving evidence for the itemset. We call this new type of provenance *itemset-provenance* (*i-provenance*).

EXAMPLE 4. *Continuing our example, suppose we wish to better understand the transactions that support the itemset {Diaper,Beer}. One question we may ask is how many other items were purchased with diapers and beer, or to retrieve items that have been purchased together with many different items including diapers and beer. This would help us understand the items that co-occur with diapers and beer. The*

[2]For a subset of relational queries, most of these semantics can be modeled using a generic semiring annotation model [24]. For in-depth discussions see, e.g., [20, 24, 21].

[3]Though as mentioned above, provenance polynomials model the contribution of input data together with how the data has been transformed.

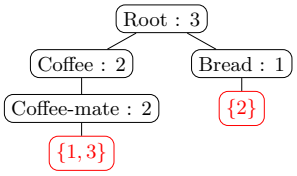

Figure 1: I-Provenance-Tree of {Beer, Diaper}

i-provenance is a representation of how items are grouped together that supports such queries.

We propose a definition of i-provenance which represents provenance using a prefix-tree with itemset counts. Prefix trees, and extensions of prefix trees, like FP-Trees [22], have been used to speed up the computation of frequent itemsets. However, to the best of our knowledge they have been not been used to represent the evidence for how a set of transactions support a frequent itemset.

EXAMPLE 5. *In Figure 1, we give an example of the i-provenance of {Diaper,Beer} for the dataset (`Transaction`) shown in Figure 2. Each path in this prefix tree corresponds to a set of items that co-occur with {Diaper,Beer} in one or more transaction (the number of occurrences is stored as a count in the end node of a path). The leaf nodes store sets of transaction IDs for the transactions witnessing a path. Items on a path are ordered according to how often they occur in combination with {Diaper,Beer}. This representation can be used to answer queries such as the ones outlined above. Assume that we are searching for items that more or less exclusively appear in a large transaction that contains {Diaper,Beer}, because we conjecture that buying such items is likely to cause the other items contained in this transaction to be bought. This information can easily be extracted from the i-provenance and could be used to promote such items.*

To increase the value of our provenance definitions, we propose an implementation of both why- and i-provenance that reuses DBMS technologies for generation and querying.

1.3 Contributions

The key contributions of this paper are the following.

- We introduce i-provenance, a new provenence semantics for frequent itemsets, and show how it can be used to understand the data (the items) that support a frequent itemset. We also provide an appropriate modification of the relational notion of why-provenance for itemsets. To the best of our knowledge, these are the first proposals for FIM provenance.
- We present efficient algorithms to generate the why- and i-provenance for a set of frequent itemsets.
- We present compression techniques to reduce the storage footprint of provenance. Instead of materializing all provenance, we only store partial provenance and use it to generate additional provenance on-demand.
- We enable the user to select what contextual information to use to represent the provenance. By default, transaction IDs are used, but a user can choose instead to use any set of attributes in the database to retrieve a more compact, if desired, and intuitive description of the why-provenance.
- We extend the relational provenance system Perm [14] to support why-provenance and i-provenance for frequent itemsets. Specifically, we extend Perm to support transactions (set of items) and to compute the why-provenance for an itemset. We extend Perm's SQL-PLE language to allow a user to interleave FIM provenance requests within a general SQL query and to declaratively specify the contextual information that should be linked with the generated why-provenance. We show how this permits efficient answering of complex queries that use itemset provenance.
- We evaluate the scalability of our approach in terms of time and space, and the effectiveness of our compression techniques using the real-life frequent itemset mining dataset `retail.dat` [1].

The remainder of the paper is organized as follows. We introduce frequent itemset mining in Section 2. We discuss related work in Section 3. Section 4 introduces the frequent itemset provenance models. We give a prototype implementation of our approach in Section 5, present algorithms for provenance generation in Section 6, and discuss compression techniques in Section 7. Section 8 covers our experimental evaluation. We conclude in Section 9.

2. FREQUENT ITEMSET MINING

Frequent itemset mining (*FIM*) discovers interesting relationships between items in a database. Frequent itemset mining is performed on a transaction database $D = \{t_1, t_2, ..., t_n\}$ where each t_i is a transaction. A transaction t consists of a unique identifier $t.ID$ and a set of items $t.Set$ from a domain $\mathbb{D}$ of items. An itemset $I = \{i_1, i_2, ..., i_m\}$ is said to be contained in a transaction t if each item in I is an element in $t.Set$. We denote the set of transactions in which the itemset I appears as $D_I = \{t | I \subseteq t.Set \wedge t \in D\}$. The set D_I is called the *transaction-base* of itemset I and the size of this set is called the *support* of the itemset. An itemset I is *frequent* in transaction database D if $\frac{|D_I|}{|D|} \geq \sigma$ where σ is a user specified threshold known as the *minimum support*. Given a minimum support σ and transaction database D, frequent itemset mining extracts all itemsets from D that are frequent, i.e., have a support of more than the threshold σ. We use $\mathbb{F}(D, k, \sigma)$ to denote all frequent item sets of size k in transaction database D according to minimal support σ. We omit D and σ if clear from the context. Similarly, $\mathbb{F}(D, \sigma)$ denotes all frequent items of any size ($\mathbb{F}(D, \sigma) = \cup_{i=1}^{\infty} \mathbb{F}(D, i, \sigma)$).

In this work, we consider a generalized type of transaction databases where each transaction t is represented as a tuple in a relational database that may contain additional information in addition to the $t.ID$ and $t.Set$. The additional information will be used by our provenance technique to provide background information about an itemset.

EXAMPLE 6. *Consider the transaction database shown in Figure 2. The relation `Transaction` links each transaction to a customer, credit card, and store. The result of frequent itemset mining for a minimum support of* 50% *is shown in the `FIM` relation. For example, {Beer} appears in transactions* 1 *to* 3 *and, thus, is frequent.*

Transaction

TID	Items	CCID	SID	CID	TS
1	{Coffee-mate, Coffee, Diaper, Beer}	1	1	1	18:23
2	{Diaper, Bread, Beer}	1	1	2	17:39
3	{Coffee-mate, Diaper, Coffee, Beer}	2	1	3	20:13
4	{Coffee-mate, Tea, Coffee, Sweetner, Coffee-cup}	2	2	4	9:23
5	{Coffee-mate, Coffee, Sweetner, Filter, Coffee-cup,Thermal-pot, Coffee-maker}	2	2	4	10:44
6	{Coffee-mate, Sugar}	2	2	4	9:24

FIM

FID	Items	Support
1	{Coffee}	4
2	{Coffee-mate}	5
3	{Diaper}	3
4	{Beer}	3
5	{Diaper, Beer}	3
6	{Coffee, Coffee-mate}	4

Credit Card

CCID	Name	Type	Limit
1	VISA	Gold	4000
2	MCARD	Silver	400

Store

SID	Location	SName	Hours
1	ON	AAA	9AM - 9PM
2	BC	BBB	24 hours

Customer

CID	AgeGroup	Sex
1	20-40	m
2	20-40	m
3	20-40	m
4	50-60	f

Figure 2: Example Schema and Instance

Many algorithms for mining frequent itemsets have been developed since the concept was first introduced [4] and their discussion is beyond the scope of this paper. As we will discuss in Section 5, our approach is oblivious to which algorithm is used to compute frequent itemsets.

3. RELATED WORK

Two lines of work are related to our approach. First, approaches for understanding itemsets or augmenting them with additional data. Second, work in data provenance.

Mei et al. [26] discuss how to generate semantic annotations for frequent itemsets. Here the focus is on finding important objects in the context of a frequent itemset, i.e., objects that co-occur with the itemset. Different types of contexts (e.g., transactions, items, and patterns) and methods of identifying important objects (i.e., weighting functions) are considered. The relation between this work and provenance for frequent itemsets is that the why-provenance is a specific version of context that uses transactions. However, instead of providing a generic framework for modelling contexts of frequent itemsets we focus on how to generate, represent, and query a specific type of context (contributing transactions). Furthermore, we let the user customize this context by deciding what additional information to include in the provenance.

A major challenge of frequent itemset mining is the sheer size of the output. Several approaches study how to compress frequent itemsets [29, 30] by finding representative frequent itemsets. This is different from provenance because using representative frequent itemsets does not help us to find the origin of a frequent itemset or understand how it was produced. However, our approach enables the user to use more concise contextual data to represent the provenance and this data can be used to find meaningful representatives for a set of frequent itemsets. For instance, we could require the representatives to faithfully represent the distribution of shop locations in a set of frequent itemsets.

The provenance of a data item describes its origin or the process by which it was created. We distinguish between *data* provenance, that describes which input data contributed to an output, and *process* provenance that records how the transformations that generated a data item contributed to it. Provenance management systems for relational data include Perm [14], Orchestra [18, 20], Trio [11], DBNotes [9], WHIPS [10, 11], and others. None of these approaches addresses generating itemset provenance or can directly be used to manage this type of provenance.

4. FREQUENT ITEMSET PROVENANCE

In this section, we define two types of provenance for frequent itemsets, why-provenance and i-provenance.

4.1 Why-Provenance

Why-provenance models which input transactions contributed to an output frequent itemset. One approach to model contribution for data provenance is to require that the provenance is a subset of the input that is *sufficient* to produce the result. For instance, the concept of sufficiency is used to define *witnesses* in the definition of relational why-provenance [8]. If t is a tuple in the result ($Q(I)$) of running a query Q over a database instance I, then a set $I' \subset I$ is called a witness for t iff $t \in Q(I')$.

Definition 1 (Why-Provenance). *The why-provenance $\mathcal{W}(I)$ of an itemset I in database D is the transaction-base of I: $\mathcal{W}(I) = \{t | I \subseteq t.Set \wedge t \in D\}$.*

This definition of why-provenance fulfills the desirable properties of *sufficiency* (all transactions of relevance are included in the provenance) and *necessity* (every transaction in the provenance contributes) [27, 16].

Theorem 1 (Sufficiency and Necessity). *We say a set of transactions $T \subseteq D$ is sufficient to compute a frequent itemset I iff $I \in \mathbb{F}(T, \sigma)$ and $|T_I| \geq |D_I|$. We call $T \subseteq D$ necessary for computing I iff removing any of the transactions in T lowers the support of I: $\forall t \in T : |(D - \{t\})_I| < D_I$. The why-provenance $\mathcal{W}(I)$ is the unique sufficient and necessary subset of transactions from D according to I.*

Proof. Follows by substituting the definitions of why-provenance and frequent itemsets in the sufficiency and necessity conditions. □

Different application domains may have different requirements for why-provenance. We use unique transaction IDs to represent the transactions in the why-provenance of a frequent itemset. However, having only the transaction IDs is not sufficient for many application domains. In our prototype system the user can use SQL queries to link additional contextual information with the generated why-provenance and query provenance information. We store materialized

(a) SQL Query

```
SELECT (C.Sex = 'm'
        AND T.TS BETWEEN 17:00 AND 19:00)
        AS MaleEvening,
       count(*) AS NumTrans
FROM (SELECT expand_set(Prov_TID) AS Prov_TID
      FROM Why_Prov
      WHERE FID=5
     ) AS P
     JOIN Transaction T
       ON (P.Prov_TID = T.TID)
     JOIN Customer C
       ON (C.CID = T.CID)
GROUP BY (C.Sex = 'm'
          AND T.TS BETWEEN 17:00 AND 19:00)
```

(b) Why_Prov Relation

FID	Prov_TID
1	{1,3,4,5}
2	{1,3,4,5,6}
3	{1,2,3}
4	{1,2,3}
5	{1,2,3}
6	{1,3,4,5}

(c) Query Result

MaleEvening	NumTrans
true	2
false	1

Figure 3: Why-provenance Example

why-provenance in a relation `Why_Prov(FID,Prov_TID)`. Each tuple in `Why_Prov` maps one frequent itemset (identified by `FID`)[4] to the set `Prov_TID` of identifiers of transactions in the why-provenance of the itemset. For instance, Figure 3(b) shows the why-provenance for the example instance from Figure 2. The following example illustrates how contextual information can be joined with the why-provenance in order to extract interesting facts about frequent itemsets.

Example 7. *Consider the query in Figure 3(a) that uses the tables from the running example in Figure 2 and the why-provenance of $I = \{Diaper, Beer\}$. Assume that the user wants to know whether the high support count of I is based on male customers buying these items in the early evening (5pm-7pm). The query first selects the why-provenance of I from the `Why_Prov` table using the FID of I (FID 5) and then uses function `expand_set` to expand the why-provenance into tuples containing a single transaction ID each. The provenance is then joined with the `Transaction` and `Customer` tables. Finally, the number of transactions supporting the male customer in the early evening hypothesis is computed. The result of this query shows that a significant number of transactions involving beer and diapers are linked with male customers shopping in the early evening.*

4.2 I-Provenance

The i-provenance of a frequent itemset is a compact representation of the interdependencies between the items and transactions that lead to an itemset being frequent. We represent this information in a prefix tree structure - the nodes of the tree represent items and their support count while the structure of the tree models mutual co-occurrence of items. We define the i-provenance of an itemset I as the prefix tree for all transactions in the why-provenance of I projected on the subset of items that appear in the why-provenance of I with a support higher than a threshold η called i-support. We refer to this prefix tree as the *i-provenance tree* for I. An i-provenance tree models the contribution of each transaction to the itemset and illustrates how common prefixes of transactions are merged together to form a frequent itemset. The threshold η enables a user to filter out irrelevant parts (items with low support in the why-provenance) from the i-provenance. Notice that η is independent of the support threshold (set by the mining algorithm), and should be set based on the level of detail required in the provenance.

Definition 2 (I-Provenance). *The i-provenance-tree $\mathcal{I}(I)$ for an itemset I in a database D according to an i-support threshold η is a prefix tree for the transactions in the why-provenance of I ordered and pruned by η. Each node in this tree is labeled with a triple (i, s, T) where i is a item, s is a relative support count, and T is a set of transactions (IDs). The i-provenance tree is constructed as follows. Let $WhyI$ denote the set of items (minus the items contained in I) that appear in the why-provenance of I with a support higher than η:*

$$WhyI = \{i \mid \frac{|\{t \mid t \in \mathcal{W}(I) \wedge i \in t.Set\}|}{|\mathcal{W}(I)|} \geq \eta\} - I$$

Given a transaction t in the why-provenance, let t_{WhyI} denote the list of items that is produced by removing items from $t.Set$ that are not in $WhyI$ and ordering the remaining items descending on their support in $\mathcal{W}(I)$: $|\{t \mid t \in \mathcal{W}(I) \wedge i \in t.Set\}|$. We use $l[i]$ to denote the i^{th} item in list l and $l[i,j]$ to denote the list containing the i^{th} up to the j^{th} item from list l. When applying this notation for tree paths, we ignore the root node. E.g., let $p = [root, n_1, n_2]$ be a path, then $p[2] = n_2$. The i-provenance tree $\mathcal{I}(I)$ for itemset I is the unique tree fulfilling the following properties:

1. *The root node of the tree is labelled with $(root, |\mathcal{W}(I)|, \{t \mid t \in \mathcal{W}(I) \wedge t.Set \cap WhyI = \emptyset\})$.*
2. *For each transaction $t \in \mathcal{W}(I)$ with $|t.Set| = m$ there exists a path p in the tree corresponding to t_{WhyI} so that $t_{WhyI}[i] = p[i].i$ and $t \in p[m].T$. Analogously, if there exists $t \in T$ for an end point of a path p in the tree it follows that $t \in \mathcal{W}(I)$ and $t_{WhyI}[i] = p[i].i$.*
3. *Let p be a path in the tree of length m and let $L = [i_1, \ldots, i_m]$ be the item labels of the nodes on p. The relative support of the last node on the path is $p[m].s = |\{t \mid t_{WhyI}[1,m] = L\}|$.*

The conditions stated in the definition above guarantee that every pruned and sorted transaction is represented as a path in the i-provenance tree and that common prefixes of sorted transactions are merged into single paths in the tree. Similar to why-provenance we define a relational representation of i-provenance. This relation `I_Prov` (`FID`, η, `Prov_Path`, `Prov_TID`) maps each frequent itemset I to all transactions contained in paths of its i-provenance tree according to threshold η. The following example illustrates this representation and the use of i-provenance for understanding a frequent itemset.

[4]These FIDs are generated automatically during frequent itemset mining.

(a) Query

```
SELECT last_element(Prov_Path) AS Item
FROM I_Prov
WHERE FID = (SELECT FID FROM FIM WHERE Items = 'Coffee')
      AND num_items(Prov_Path) = (SELECT max(num_items(Prov_Path)) FROM I_Prov)
```

(b) I_Prov Relation of {Coffee}

FID	η	Prov_Path	Prov_TID	Prov_Freq
1	0.0%	{Coffee-mate, Diaper, Beer}	{1,3}	2
1	0.0%	{Coffee-mate, Sweetner, Coffee-Cup, Tea}	{4}	1
1	0.0%	{Coffee-mate, Sweetner, Coffee-cup Thermal-pot, Filter, Coffee-maker}	{5}	1
...	...	...	...	...

(c) Query Result

Item
{Coffee-maker}

Figure 4: I-provenance Example: Determining Promotional Items

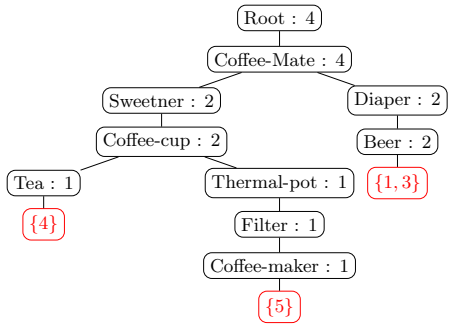

Figure 5: I-provenance Tree of {Coffee}

EXAMPLE 8. *Figure 5 shows the i-provenance tree of the itemset $I = \{Coffee\}$ according to the dataset from the running example ($\eta = 0\%$). Item Coffee-Mate is sold with itemset I in transactions $\{1, 3, 4, 5\}$. The itemsets for transactions $\{1, 3\}$ are the same and, thus, are merged in the tree. Coffee-Mate is also sold with other items in Transaction $\{5\}$ and its node is shared by the all paths i the tree.*

I-provenance can be useful in many domains. For example, a grocery store may want to identify products whose promotion might trigger the shoppers to buy other related items. The store wants to make this decision based on the items that the shopper has in his/her shopping cart. The goal is to increase the sale of as many items as possible by giving discount coupons for an item that is most likely to trigger the sale of many additional items. The following example illustrates the use of i-provenance for such a scenario.

EXAMPLE 9. *Suppose a shopper has Coffee in his/her shopping cart. Consider the i-provenance of $\{Coffee\}$ which is shown in Figure 4(b). The marketing department argues that items which are often sold with a large set of items, but seldom in other combinations may act as triggers for buying the large set of items. Thus, the shop should give coupons for these items. Bob, the shop's data analyst, suggests to use the i-provenance to identify such trigger items and issue coupons accordingly. Large sets of items brought together correspond to long paths in the i-provenance. For example, in the i-provenance of $\{Coffee\}$ we notice that the longest path $\{Coffee\text{-}mate, Sweetner, Filter, Coffee\text{-}cup, Thermal\text{-}pot, Coffee\text{-}maker\}$ contains items which occur less frequently in other paths. Given an frequent itemset I (e.g., contained in the customer's shopping cart) we can use the query shown in Figure 4(a) to return the item on the longest path in the i-provenance of I which occurs most infrequently on other paths. All items on a path occur in the same fixed set of transactions that corresponds to the path and, thus, the item which occurs most infrequently on other path has to have the lowest support in the why-provenance. This item will be the leaf item on the path, because items on the paths in an i-provenance tree are ordered on support. The result of this query for $I = \{Coffee\}$ is shown in Figure 4(c). In the example, we would issue a coupon for a Coffee-maker.*

5. PROTOTYPE IMPLEMENTATION

Our prototype implementation has four main components: the database, the mining algorithms, the provenance manager, and the provenance generator. The system architecture is shown in Figure 6. We use a DBMS (Perm) to store the input transaction database (including contextual information), the results of the frequent itemset mining algorithm, and materialized provenance.

FIM. The provenance generation works solely on the data stored in the database. Thus, provenance generation is independent of which algorithm is chosen for frequent itemset mining. This advantage comes at the small cost of having to implement a wrapper for each mining algorithm. The wrapper retrieves the transactions stored in Perm and sends them to the mining algorithm as input. The generated frequent itemsets with their support counts are then sent back to the wrapper and stored in Perm. An example frequent itemset table for our running example is shown in Figure 2. The `FID` attribute stores a unique identifier for each frequent itemset, the `Items` attribute stores the actual frequent itemset, and the `Support` attribute stores the itemset's support count.

Storage and Querying. We use the Perm system as a storage backend instead of building a new provenance store from scratch. Perm [14] is a provenance system for relational data implemented as an extension of PostgreSQL. Using Perm's SQL-PLE (provenance-language-extension) we can write declarative SQL queries to explore provenance data. The system uses query rewrites to transform provenance queries into standard SQL queries. As a result, it benefits from the query optimization techniques of PostgreSQL. We chose Perm as a data store in this project because the system supports external provenance, i.e., when running a

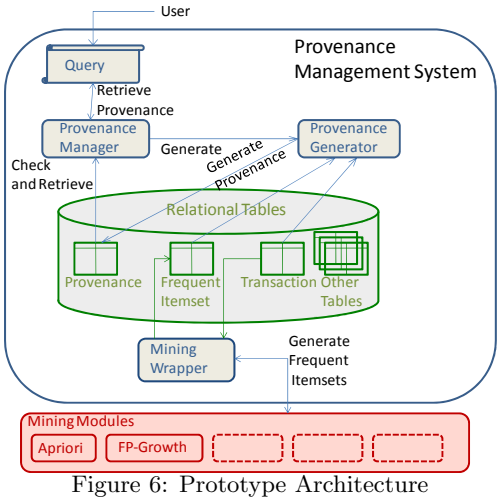

Figure 6: Prototype Architecture

query over data that has provenance attached to it, the system uses this information to compute provenance for the query. In addition to enabling us to run queries over the frequent itemset provenance generated by our system, this feature is used by our prototype to link contextual information and existing provenance of transactions with the itemset provenance. Perm represents data and its provenance using the same relation which fits our model of frequent itemset provenance.

Provenance Generation. Once frequent itemset mining is finished, the provenance generation module can be used to generate the why-provenance or i-provenance for any subset of the frequent itemsets. The details of the provenance generation are discussed in Section 6. The provenance generator and manager modules are implemented as a middleware layer. Using an SQL query, the user selects for which frequent itemsets the provenance should be generated and which additional contextual information should be propagated. Furthermore, the user can indicate whether the raw why- (transaction IDs) or i-provenance should be materialized. Initially, after execution of the mining algorithm the `Why_Prov` and `I_Prov` tables are empty. If the user requests the why-provenance (or i-provenance) for some frequent itemsets by running a query, we compute the requested why-provenance, store it in the `Why_Prov` (respectively, `I_Prov`) relation, and compute the query result. Thus, the `Why_Prov` and `I_Prov` relations may be filled over time. Our prototype uses a naive physical representation of sets of items and sets of transactions as strings. We use the text-search facilities of PostgreSQL to implement containment checks (e.g., is itemset $\{a, b\}$ a subset of itemset $\{a, b, c\}$) and the *ltree* module to store paths in i-provenance trees (a more detailed explanation of the use of this module can be found in Section 6.2).

6. PROVENANCE GENERATION

In this section, we describe the algorithms that are used to generate provenance of frequent itemsets.

6.1 Why-provenance Generation

Recall that we define the why-provenance of a frequent itemset as its *transaction-base*, i.e., the set of transactions that contain all elements of the itemset. Thus, to retrieve

(a) Gen-Why-All

```
SELECT f.FID, concat_agg(t.TID) as Prov_TID
FROM FIM f, Transaction t
WHERE contains(f.Items, t.Items)
GROUP BY FID
```

(b) Gen-Why-List

```
SELECT f.FID, concat_agg(t.TID) as Prov_TID
FROM FIM f, Transaction t
WHERE contains(f.Items, t.Items)
AND f.items = ANY ($1)
GROUP BY FID
```

Figure 7: SQL Queries for Computing Why-provenance

(a) Query over Why-provenance

```
SELECT PROVENANCE count(*) AS numFI
FROM
 (SELECT f.*, w.Prov_TID AS TID
 FROM FIM f, Why_Prov w
 WHERE f.FID = w.FID
 ) PROVENANCE(FID, TID) AS FIM,
WHERE num_items(Items) > 5
      AND f.Support > 100
```

(b) Example Partial Provenance Generation

```
SELECT f.FID, concat_agg(t.TID) as Prov_TID
FROM FIM f, Transaction t
WHERE contains(f.Items, t.Items)
AND f.Items
    = ANY ('{"Beer,Diaper","Coffee"}'::text[])
GROUP BY FID
```

Figure 8: Querying Why-provenance

the why-provenance for a given itemset we can run a query over the transaction table that checks whether the itemset is contained in a transaction's set of items. Figure 7(a) shows the query that is used to generate the why-provenance for all frequent itemsets. Here we assume the existence of a function `contains` that checks whether one set is contained in another. Using the string representation of itemsets we implemented this function using the text search facilities of PostgreSQL.

We also support why-provenance generation for a subset of the frequent itemsets selected by the user. The user writes a query that returns the itemsets of interest and we use this query to generate a query to trace the provenance of these itemsets. This is simply done by replacing the access to relation `FIM` in the query from Figure 7(a) with the user's query. Furthermore, we provide a function that computes the why-provenance for a set of frequent itemsets. The SQL code of this function is shown in Figure 7(b). We use group-by and an aggregation function concatenating strings to combine all transaction IDs in the provenance for a particular FID into a set stored as a single attribute value.

EXAMPLE 10. *The expanded SQL query for generating the why-provenance of itemsets* {*Beer,Diaper*} *and* {*Coffee*} *is given in Figure 8(b).*

The Perm system supports externally generated provenance. Using the `PROVENANCE` keyword in the `FROM` clause we can make Perm aware that certain attributes of a from clause

Algorithm 1 I-Provenance Generation

```
1: procedure GenI-Prov(I, η)
2:     W ← GenWhy(I)              ▷ Compute why-provenance
3:     L ← items(W)               ▷ List of items in w
4:     L ← SortOnSupport(L)       ▷ Decreasing sort
5:     L ← PruneOnSupport(L, η)
6:     T = ∅                      ▷ Initialize i-provenance tree
7:     for all t ∈ w do
8:         L' ← PruneAndSortOnList(t.Set, L)
9:         insertPath(T, L', t)   ▷ Duplicate paths merged
10:    end for
11:    return T
12: end procedure
```

I_Prov

FID	η	Prov_Path	Prov_TID	Prov_Freq
1	50%	{Coffee-mate, Diaper, Beer}	{1,3}	2
1	50%	{Coffee-mate, Sweetner, Coffee-cup}	{4,5}	2
...	...	...	...	...

Figure 9: I-provenance Relation for {Coffee} with $\eta = 50\%$

entry contain provenance information. This additional provenance information will be handled in the same way as provenance generated by Perm, i.e., it will be propagated during query evaluation if provenance is requested by the user. In our system, we make use of this feature to compute the provenance of queries run over the database storing the frequent itemsets and transactions.

Example 11. *Figure 8(a) shows an SQL-PLE query (the SQL dialect with provenance features of Perm) that returns the provenance of a query that counts the number of frequent itemsets with more than 5 elements and a support higher than 100. The user has chosen the combination of fid and tid as contextual information to represent the provenance. The provenance-enabled query returns the support count combined with the tid and fid for all transactions in the provenance of the itemsets fulfilling the* `WHERE` *condition.*

6.2 I-Provenance Generation

Algorithm 1 is used to generate i-provenance. Input to the algorithm is a frequent itemset I and i-support threshold η. In the first step, the algorithm computes the why-provenance $\mathcal{W}(I)$ for itemset I, computes the support of each item in $\mathcal{W}(I)$, prunes infrequent items using threshold η, and then generates a list L of these items sorted decreasingly on support. List L corresponds to $WhyI$ from Section 4.2. For example, the content of list L for itemset {Coffee} and i-support $\eta = 50\%$ according to the running example dataset is [Coffee-mate, Diaper, Beer, Sweetner, Coffee-cup]. Notice that items Thermal-pot, Filter, Coffee-maker, and Tea were pruned from the list because their support in $\mathcal{W}(I)$ is less than 50%. The algorithm then iterates through each transaction in the why-provenance $\mathcal{W}(I)$. Each transaction's itemset is pruned and sorted based on the order of items in list L. The resulting list t_{WhyI} is then inserted into the i-provenance tree. For example, the resulting i-provenance tree for itemset {Coffee} with $\eta = 0\%$ from our running example is given in Figure 5.

```
SELECT last_element(Prov_Path) AS Item
FROM I_Prov
WHERE FID = fid('Coffee')
AND num_items( Prov_Path ) =
 (SELECT max(num_items(Prov_Path))
  FROM I_Prov
  WHERE Prov_Path
        ~ '*.CoffeeCup.*.!CoffeeMaker.*'
  AND FID = fid('Coffee'))
AND Prov_Path ~ '*.CoffeeCup.*.!CoffeeMaker.*'
```

Figure 10: SQL Query over I-provenance using ltree

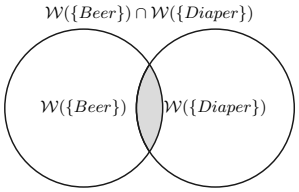

Figure 11: Computing Why-provenance using Intersection

As explained in Section 4.2, we store i-provenance trees in a relation `I_Prov`. Each tuple in this relation represents a path in the i-provenance tree. The `insertPath` method inserts a tuple $(I, L', t.ID)$ into the `I_Prov` table if called with a sorted transaction L' and transaction t. Duplicate paths are merged and their frequency counts are aggregated. For example, Figure 9 shows the `I_Prov` table for itemset {Coffee} from the running example. Notice that in this example, the tuples with `Prov-TID` 4 and 5 are merged together in a single tuple with frequency count 2.

We can execute complex queries over the i-provenance using Perm. However, querying the i-provenance is more challenging than querying why-provenance, because of its tree representation. We use the ltree module [2] of PostgreSQL to query the i-provenance. The ltree module implements common operators for trees and paths such as computing the lowest common ancestor of two paths and regular expression matching for paths.

Example 12. *Continuing with Example 9 where a supermarket has determined that Coffee-maker is the best trigger item for customers buying Coffee (that is, the best item to promote to trigger them to buy the most additional products). Suppose we have a new customer who has both Coffee and Coffee-cups in his basket, but Coffee-makers are out of stock (so we don't want to promote this item). An analyst can write a new query to find an appropriate trigger item. The new query is shown in Figure 10. Notice that using ltree the analyst can search for paths with Coffee-cup that do not contain Coffee-maker. If this query is run on the dataset presented in Example 9 then Tea is reported as the most suitable item to promote.*

7. PROVENANCE COMPRESSION

We now describe how to avoid computing the why-provenance for an itemset from scratch without having to materialize the why-provenance of all frequent itemsets. We exploit the well-known *Apriori*-property of frequent itemsets to compute the why-provenance of a frequent itemset based on materialized why-provenance of its subsets.

Algorithm 2 Why Provenance Compression

```
1: procedure CompressWhyProvenance(I, M)
2:     q ← Evaluate(Π_query(σ_FID=fid(I)(MWhy)))
3:     if q ≠ ∅ then
4:         return Evaluate(q)
5:     end if
6:     q' ← σ_(Items⊂I)(Π_FID,Items(
           FIM ⋈_FID=FID σ_query=NULL(MWhy)))
7:     S ← Evaluate(q')
8:     Cand ← SortOnNumItems(S)
9:     Found ← {}
10:    while Found ≠ I & Cand ≠ {} do
11:        C ← Pop(Cand)
12:        if C ⊄ T then
13:            Found ← Found ∪ C
14:            q' ← Π_ES(Prov_TID)(σ_FID=fid(C)(Why_Prov))
15:            q ← q ∩ q'
16:        end if
17:    end while
18:    Missing ← I − Found
19:    for all i ∈ Missing do
20:        q' ← Π_ES(Prov_TID)(σ_FID=fid({i})(Why_Prov))
21:        q ← q ∩ q'
22:        GenWhyProv(i)
23:        Insert(MWhy, (fid({i}), NULL))
24:    end for
25:    if M then
26:        Insert(MWhy, (fid(I), q))
27:        return Evalutate(q)
28:    else
29:        Insert(MWhy, (fid(I), NULL))
30:        Why_Prov←Why_Prov ∪ Evaluate(q)
31:    end if
32: end procedure
```

Example 13. *Suppose we have materialized the why-provenance of {Beer} and {Diaper} and want to compute the why-provenance of {Diaper, Beer}. Figure 11 shows the why-provenance of {Diaper} and {Beer}. The why-provenance of {Beer} respective {Diaper} contains the set of transactions that contains {Beer} respective {Diaper}. The intersection of these two sets is the set of all transactions containing both {Beer} and {Diaper}, which is the why-provenance of {Diaper, Beer}. Instead of materializing this set we can generate it by executing a query that intersects the materialized why-provenance of {Beer} and {Diaper}.*

The approach exemplified above works in general, because of the so-called *Apriori*-property of frequent itemsets [5]) that we restate for why-provenance.

Theorem 2. *Let I and I' be frequent itemsets. The following holds for $I'' = I \cup I'$: $\mathcal{W}(I'') = \mathcal{W}(I) \cap \mathcal{W}(I')$.*

We use this property as follows. First, we use a table `MWhy(fid,query)` to store pairs of itemsets and queries for generating their provenance. Whenever the why-provenance of an itemset with FID fid is materialized we insert a tuple (fid, NULL) into relation `MWhy`. If the why-provenance of an itemset I is computed using an intersection query q as exemplified above and the user decided to not materialize the resulting why-provenance, then we insert a tuple (fid(I), q) into relation `MWhy`. Given an itemset I and flag M that indicates whether the result should be materialized, Algorithm 2 is used to determine how to compute the why-provenance. We generate a query that generates $\mathcal{W}(I)$ based on the materialized why-provenance of itemsets contained in I. We first check whether the query for generating I is already stored in `MWhy` or if $\mathcal{W}(I)$ has been materialized. If that is the case, we simply execute that query or retrieve the provenance from relation `Why_Prov`. Otherwise, the algorithm greedily generates a query q that intersects the provenance of other itemsets contained in I.

We first extract the list of all itemsets whose provenance is materialized by joining the `FIM` table with the `MWhy` table and filtering out itemsets that are not contained in I. We then sort these itemsets in descending order based on their size (number of items they contain). Our goal is to compute $\mathcal{W}(I)$ by using a small number of intersections with a preference for using large itemsets, because large itemsets tend to have smaller why-provenance. Variable *Found* is used to store the items from I for which we have found a match so far. We loop over the itemsets in *Cand* and in each step choose the largest itemset C which is stored at the beginning of *Cand*. If C is not contained in *Found* then we add an intersection with the query to compute the why-provenance of C to q and update *Found*. Here ES stands for the `expand_set` function introduced Section 4.1. If the materialized why-provenance does not contain a set of itemsets that cover I ($Found \neq I$), then we materialize the why-provenance for all singleton subsets of I that have not been covered. Finally, we evaluate q to generate the why-provenance of I and store query q in table `MWhy` (or materialize $\mathcal{W}(I)$ in `Why_Prov`).

Since i-provenance generation requires generating why-provenance, its performance can benefit from the storage and performance improvements that result from why-provenance compression. Furthermore, if the i-provenance for an itemset I according to an i-support threshold η_1 has been materialized, we can compute the i-provenance of I for any threshold $\eta_2 > \eta_1$ by filtering out items from the i-provenance tree paths that have a support less than η_2.

8. EXPERIMENTAL EVALUATION

We ran three types of experiments to evaluate our system. One related to scalability in terms of execution time of provenance generation, one related to compression, and one to evaluate the effect of provenance materialization on query performance. All experiments were run on a machine with an 8-core 2.93GHz Intel Xeon X3470 processor and 24GB of RAM running Ubuntu Linux 12.04.1. Our provenance generation module is implemented in Java and uses JDBC to connect with Perm. We have used the *retail.dat* [1] dataset in all experiments. The dataset retail.dat is a market basket dataset which contains 88,163 transactions. On average each transaction contains 13 items from a domain of 16,470 items. We have chosen retail.dat because it is an average size frequent itemset mining dataset containing data from real customers.

8.1 Scalability

We first analyze the scalability of why-provenance and i-provenance generation. We created three datasets containing 100%, 50%, and 25% of randomly selected transactions from the retail.dat dataset and used minimum sup-

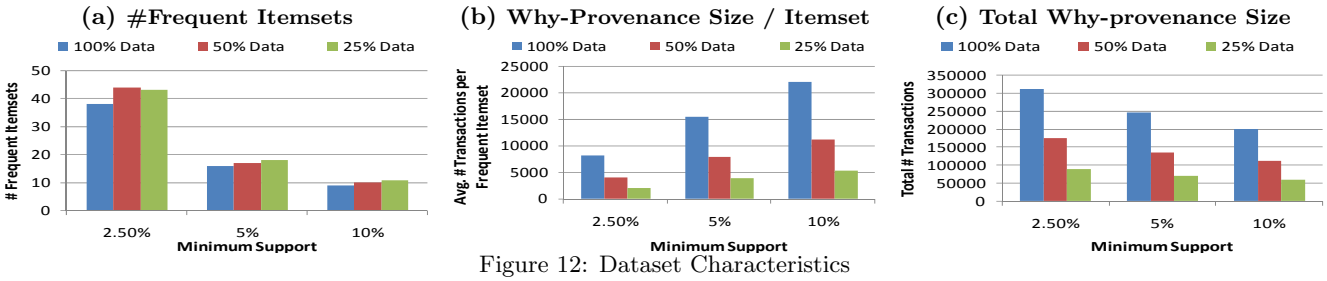

Figure 12: Dataset Characteristics

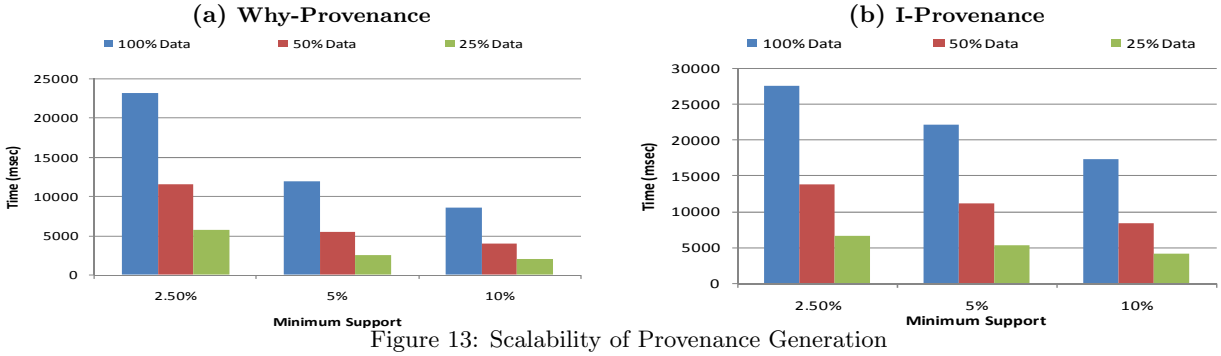

Figure 13: Scalability of Provenance Generation

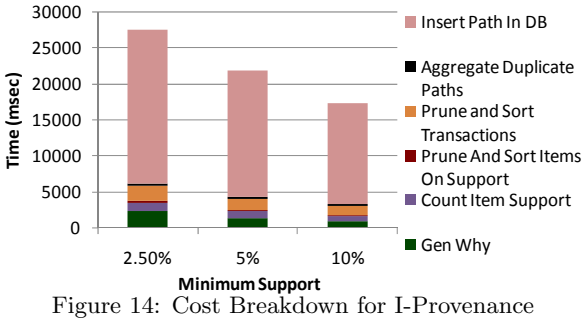

Figure 14: Cost Breakdown for I-Provenance

port thresholds of 2.5%, 5%, and 10% to generate the frequent itemsets for each of the datasets. Figure 12 shows the characteristics of the dataset and why-provenance for each dataset-minimum support combination: the total number of frequent itemsets (Figure 12(a)), the average number of transactions in the why-provenance of a frequent itemset (Figure 12(b)), and the total number of transactions in the why-provenance for all frequent itemsets (Figure 12(c)). We did not apply any compression in this experiment and all results were materialized in the `Why_Prov` relation using the schema introduced in Section 4.

Figure 13(a) shows the runtimes for generating the why-provenance of all frequent itemsets. The why-provenance generation scales linearly in the size of the dataset times the number of frequent itemsets plus the total size of the why-provenance (see Figures 12(a) and 12(c)). This is what is to be expected, because why-provenance generation is implemented as a join between the `Transaction` and `FIM` relations. Since the join condition is set containment, the DBMS cannot apply a hash- or merge-join resulting in performance linear in the number of frequent itemsets times dataset size.

Next we evaluate the i-provenance generation algorithm. We use the setup described above and an i-support threshold η of 0%. In this experiment, we assume that the why-provenance is already materialized, i.e., we do not include the cost of generating why-provenance in the measurement. As is evident from the results shown in Figure 13(b), the i-provenance generation also scales well.

8.2 Cost Breakdown for I-Provenance

Since i-provenance generation is more complex than why-provenance, we measured the run-times of individual steps in the generation to better understand the performance. For this experiment, we have used the 100% data set, minimum support thresholds of 2.5%, 5%, and 10%, and i-support 0%. The results are shown in Figure 14. In our prototype, the i-provenance generation algorithm (Algorithm 1) is implemented in Java. The system first retrieves the why-provenance of an frequent itemset from the database (*Gen-Why*), then calculates the support of all items in the why-provenance (*Count Item Support*), sorts the items based on their support and removes items with a support lower than threshold η (*Prune And Sort Items On Support*), computes t_{Fhow} for each transaction (*Prune and Sort Transactions*), aggregates the supports and transaction sets for the individual paths in the tree (*Aggregate Duplicate Paths*), and finally inserts the i-provenance tree into the database (*Insert Path in DB*). Recall that the why-provenance is materialized in the database in these i-provenance experiments. All operations scale well and none of the operations becomes a major bottleneck as the minimum support is decreased. The main cost driver turned out to be inserting the i-provenance trees into the database.

8.3 Why-Provenance Compression

We now evaluate the why-provenance compression techniques. We have used the 100% dataset with 2.5%, 5%, and 10% minimum support. The number of frequent itemsets for each minimum support is given in Figure 12(a). We materi-

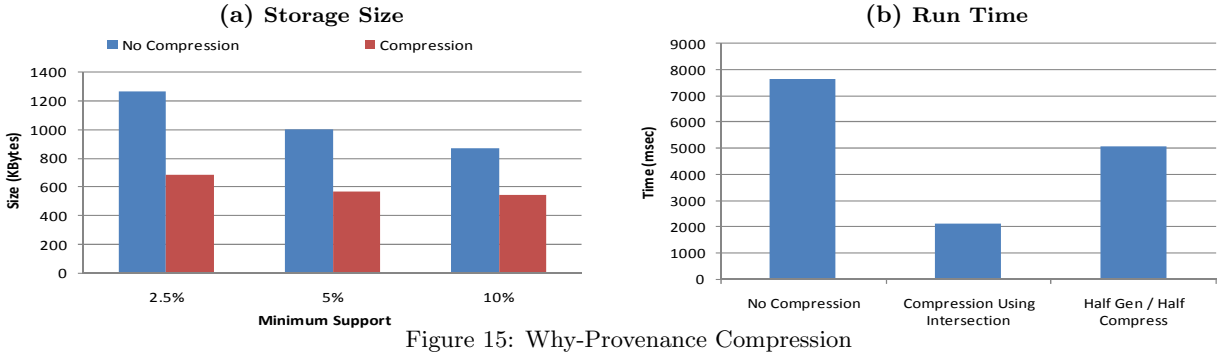

Figure 15: Why-Provenance Compression

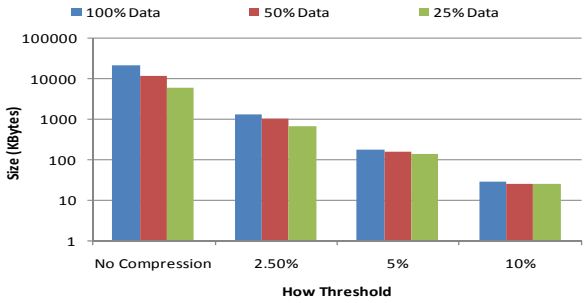

Figure 16: I-provenance Size for $\eta = 0\%, 2.5\%, 5\%, 10\%$

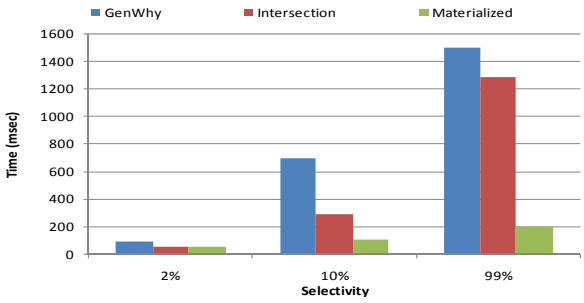

Figure 17: Query Performance

alized the why-provenance of all frequent itemsets of size one (1-itemsets) and expressed the provenance of all remaining frequent itemsets as intersection queries. As shown in Figure 15(a), our compression technique achieves almost 50% compression based on the fact that roughly half of the transactions in the why-provenance belong to singleton itemsets. Compression works well when minimum support is low, because this usually implies that a large number of frequent itemsets of sizes larger than one are generated.

In Figure 15(b) we show the run-times of generating the why-provenance for all itemsets of size larger than one with and with-out compression. We again used the 100% dataset with a minimum support threshold of 2.5%. The mining algorithm generated 38 frequent itemsets out of which 14 are 1-itemsets. The total time required for generating the why-provenance for the remaining 24 itemsets from scratch without using the materialized 1-itemsets (*No Compression*) was about 7.5 seconds. Generating the why-provenance for these itemsets by intersecting the materialized provenance of 1-itemsets took roughly 2 seconds. Finally, we materialized the provenance of 7 1-itemsets to test a scenario where only the provenance of some frequent 1-itemsets has been stored upfront. When possible we used intersection queries to generate the provenance of the remaining itemsets and only fall back to generating the provenance from scratch if the necessary singleton itemset provenance is not available. Using this approach the generation finished in about 5 seconds.

This experiment demonstrates that compression using intersecting queries not only saves space, but also exhibits better performance than generating provenance from scratch.

8.4 Impact of the I-support Threshold

We now analyze the impact of the user-supplied i-support threshold η on the size of the generated i-provenance. Recall that we prune items from the i-provenance tree which have a support in the why-provenance less than the threshold η. The results for all three dataset sizes are shown in Figure 16. Note that the y-axis is in log-scale. Increasing the i-support threshold can result in significant space reduction. This reduction seems to be more sensitive to the i-support threshold than to the size of the dataset. The results indicate that i-support is an effective instrument for drilling-down into important parts of the i-provenance. For example, just removing the most infrequent items using $\eta = 2.5\%$ reduces the size of the provenance by one order of magnitude.

8.5 Query Performance on Why-Provenance

Our final experiment analyzes how query performance is affected by provenance generation and storage. We use a simple query that joins the provenance with the `Transaction` table and vary the selectivity by filtering out a subset of the transactions. Three options for interleaving querying with provenance generation were considered. The *GenWhy* method generates the provenance from scratch using a subquery. In the *Intersection* method, we generate the provenance by using a subquery that intersects the materialized provenance of 1-itemsets as explained in Section 7. Finally, for *Materialized*, all provenance was stored upfront. The results shown in Figure 17 demonstrate that, as expected, materializing the provenance pays off, especially for higher selectivities. For low selectivities, the *Intersection* method shows almost the same performance as materialized provenance. In general, the *Intersection* approach performs better than generating the provenance from scratch (*GenWhy*). These preliminary result indicate that, even with our naive implementation of provenance generation, on-demand computation is a feasible option. For instance, if we would use an index that supports set containment queries, then we con-

jecture that *GenWhy* and *Intersection* would outperform *Materialized* for low selectivities.

9. CONCLUSIONS

In this paper, we introduce *why-* and *i-provenance* for frequent itemsets and present algorithms for generating both types of provenance. Managing provenance naively can be inefficient because the why-provenance tends to be large. Even more importantly, it is hard for users to gain any insight from this overload of information. We address this problem by enabling the user to explore and filter provenance information through declarative querying and we introduce a compression technique for why-provenance. Our definition of i-provenance includes an i-support threshold that allows a user to control the level of detail in the provenance (independently of the support threshold used by a mining algorithm). Last but not least, we argue that using contextual information instead of transactions as a provenance representation often results in compact and more informative provenance. Our experimental evaluation demonstrates that our system scales well with respect to the size of the database and number of frequent itemsets.

An interesting avenue for future work is provenance generation and management for other data mining algorithms like clustering or classification. We would like to explore whether the techniques developed in this paper can enhance applications that use FIM such as association rule mining [4] or detecting functional dependencies [23]. Furthermore, integrating provenance generation with itemset mining opens up new opportunities for performance improvements.

10. REFERENCES

[1] Frequent itemset mining implementations repository. http://fimi.ua.ac.be/.

[2] Postgresql 9.13 documentation. http://www.postgresql.org/docs/9.1/static/ltree.html.

[3] What is the "true story" about data mining, beer and diapers? http://www.dssresources.com/newsletters/66.php.

[4] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. *SIGMOD Record*, 22(2):207–216, 1993.

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

[6] P. Buneman, A. Chapman, and J. Cheney. Provenance Management in Curated Databases. In *SIGMOD*, pages 539–550, 2006.

[7] P. Buneman, S. Khanna, and T. Wang-Chiew. Why and where: A characterization of data provenance. *ICDT*, pages 316–330, 2001.

[8] J. Cheney, L. Chiticariu, and W. Tan. Provenance in databases: Why, how, and where. *FTDB*, 1(4):379–474, 2009.

[9] L. Chiticariu, W. Tan, and G. Vijayvargiya. DBNotes: a post-it system for relational databases based on provenance. In *SIGMOD*, pages 942–944, 2005.

[10] Y. Cui, J. Widom, and J. Wiener. Tracing the lineage of view data in a warehousing environment. *TODS*, 25(2):179–227, 2000.

[11] A. Das Sarma, M. Theobald, and J. Widom. Exploiting lineage for confidence computation in uncertain and probabilistic databases. In *ICDE*, pages 1023–1032, 2008.

[12] J. Freire, D. Koop, E. Santos, and C. T. Silva. Provenance for Computational Tasks: A Survey. *CISE*, 10(3):11–21, 2008.

[13] L. Geng and H. Hamilton. Interestingness measures for data mining: A survey. *CSUR*, 38(3):9, 2006.

[14] B. Glavic and G. Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*, pages 174–185, 2009.

[15] B. Glavic, G. Alonso, R. Miller, and L. Haas. TRAMP: understanding the behavior of schema mappings through provenance. *PVLDB*, 3(1-2):1314–1325, 2010.

[16] B. Glavic and R. J. Miller. Reexamining Some Holy Grails of Data Provenance. In *TaPP*, 2011.

[17] G. Grahne and J. Zhu. Fast algorithms for frequent itemset mining using fp-trees. *TKDE*, 17(10):1347–1362, 2005.

[18] T. Green, G. Karvounarakis, Z. Ives, and V. Tannen. Update exchange with mappings and provenance. In *VLDB*, pages 675–686, 2007.

[19] T. Green, G. Karvounarakis, Z. Ives, and V. Tannen. Provenance in orchestra. *IEEE Data Eng. Bull*, 33(3):9–16, 2010.

[20] T. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

[21] T. J. Green. Containment of Conjunctive Queries on Annotated Relations. In *ICDT*, pages 296–309, 2009.

[22] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *SIGMOD Record*, 29(2):1–12, 2000.

[23] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The Computer Journal*, 42(2):100–111, 1999.

[24] G. Karvounarakis and T. Green. Semiring-annotated data: Queries and provenance. *SIGMOD Record*, 41(3):5–14, 2012.

[25] D. Keim and H. Kriegel. Visualization techniques for mining large databases: A comparison. *TKDE*, 8(6):923–938, 1996.

[26] Q. Mei, D. Xin, H. Cheng, J. Han, and C. Zhai. Generating semantic annotations for frequent patterns with context analysis. In *SIGKDD*, pages 337–346, 2006.

[27] A. Meliou, W. Gatterbauer, K. Moore, and D. Suciu. The Complexity of Causality and Responsibility for Query Answers and non-Answers. *PVLDB*, 4(1):34–45, 2010.

[28] K. Wang, C. Xu, and B. Liu. Clustering transactions using large items. In *CIKM*, pages 483–490, 1999.

[29] D. Xin, J. Han, X. Yan, and H. Cheng. Mining compressed frequent-pattern sets. In *VLDB*, pages 709–720, 2005.

[30] X. Yan, H. Cheng, J. Han, and D. Xin. Summarizing itemset patterns: a profile-based approach. In *SIGKDD*, pages 314–323, 2005.