San Jose State University

SJSU ScholarWorks

2006

A Fast Algorithm For Data Mining

Aarathi Raghu
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the Computer Sciences Commons

A Fast Algorithm For Data Mining

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment of the Requirement for the Degree

Master of Science

By

Aarathi Raghu

December 2006

Table of Contents

Table of Figures

Abstract

In the past few years, there has been a keen interest in mining frequent itemsets in large data repositories. Frequent itemsets correspond to the set of items that occur frequently in transactions in a database. Several novel algorithms have been developed recently to mine closed frequent itemsets - these itemsets are a subset of the frequent itemsets. These algorithms are of practical value: they can be applied to real-world applications to extract patterns of interest in data repositories. However, prior to using an algorithm in practice, it is necessary to know its performance as well implementation issues. In this project, we address such a need for the algorithm "Using Attribute Value Lattice to Find Frequent Itemsets" that was developed by Lin et. al. We clarify some aspects of the algorithm, develop an implementation of the algorithm, and present the results of a performance study. In our experiments we find that the running time of the algorithm for certain input datasets grows exponentially. To address this problem, we develop a novel procedure for binning the data. Our results show that with binned data, the running time of the algorithm grows linearly. This allows one to obtain trends for the dataset.

1. Introduction

Mining large data repositories to find frequent itemsets has been studied for over fifteen years [Agarwal1994]. During the past five years, there has been a renewed interest in mining frequent itemsets [Burdick2001, Lin2003, Pei2000, Zaki2002]. Frequent itemsets correspond to the set of items that occur frequently in transactions in a database. Several novel algorithms have been developed recently to mine closed frequent itemsets---these itemsets are a subset of the frequent itemsets. These algorithms are of practical value: they can be applied to real-world applications to extract patterns of interest in data repositories. However, prior to using an algorithm in practice, it is necessary to know its performance as well implementation issues. The goal of this project is to address such a need for the algorithm "Using Attribute Value Lattice to Find Frequent Itemsets" that was developed by Lin et al [Lin2003]. Before we describe our contributions, we provide a brief overview of the problem space.

Mining large data repositories to identify interesting patterns is a challenging problem. The volume of data to be processed is large (several hundred GB to a few TB in size) and hence, requires designing efficient algorithms to

identify patterns that occur frequently. To illustrate, suppose a user is

buying a book at Amazon.com's web-site. When the user chooses a book,

the Amazon.com site also shows related books that would be of potential

interest to the user. By doing so, this has the effect of increasing the

revenue.   Given the large number of book titles, it is non-trivial to manually

generate the list of related books. However, such related books are inferred

from the buying habits of Amazon.com's customers. That is, determining

the set of related books that are bought frequently.

To illustrate the problem of data mining of frequent occurring patterns,

consider a sample database of transactions shown in This example **has been**

adapted from Dr. Lin's Lattice paper [Lin 2003].

Figure 1. This example is adapted from [Zaki2002]. The set of items for a

given transaction could be the buying habits of users, such as, books written

by Jane Austen, Agatha Christie, Sir Arthur Conan Doyle, etc.

Transaction	Items
1	ACTW
2	CDW
3	ACTWHG

4	ACDWHF
5	ACDTWHGK
6	CDTHFK
7	HFKQR
8	HGKQR
9	QRS
10	QRS

This example has been adapted from Dr. Lin's Lattice paper [Lin 2003].
Figure 1 Sample Database

The database consists of seven transactions with twelve different items. Let ϑ denote the set of items in the database. A set $N \subseteq \vartheta$ consisting of items from the database is called an *itemset*. For example, $N = \{A, C, D\}$ is an itemset. For notational convenience, we will write ACD to denote the itemset N consisting of items A, C, and D. Suppose that one is interested in identifying the itemsets that occur in at least 2 transactions (i.e., the set of authors whose books are commonly bought). Given the sample database, the itemsets are A, C, D, H, F, K, Q, R. A commonly used terminology in the data mining literature to denote the number of transactions in which an itemset occurs as a subset is *support*. The problem of finding patterns in the

database can be restated as: identify the itemsets that have at least the user-specified level of *support*. The user-specified level of support is known as *minimum support* (or *minsup* for short) and itemsets that satisfy minsup are known as *frequent itemsets*.

Devising algorithms for mining frequently occurring patterns in large databases is an area of active research [Survey]. Some of the challenges common to algorithms for mining frequently occurring patterns in large data repositories are [Survey]:

1. Identifying the set (possibly, complete) of patterns that satisfy user-specified thresholds, such as, minsup

2. Minimize the number of scans over the database

3. Be computationally efficient

An algorithm that satisfies the above requirements is "Using Attribute Value Lattice to Find Closed Frequent Itemsets" [Lin2003]. This thesis builds on their algorithm. In particular, we make the following contributions:

1. We identify correctness issues with the algorithm's pseudo-code and rewrote the algorithm for clarity.

2. We developed an implementation of their algorithm. As part of the implementation, we identify issues with algorithm and propose solutions.

3. We use our implementation to analyze the performance of the algorithm using synthetically generated data-sets.

4. We use data binning mechanisms to improve the run-time performance of the algorithm for certain data-sets.

The remainder of this document is organized as follows. In Chapter 2, we provide an overview of algorithms for mining frequent itemsets. In Chapter 3, we describe the algorithm "Using Attribute Value Lattice to Find Closed Frequent Itemsets" which is the basis for our work. In Chapter 4, we describe our implementation and present the results of our experiments. Finally, Chapter 5 concludes.

2. Background and Related Work

The classical algorithm for mining frequent itemsets is the Apriori algorithm [Agarwal1994]. Given a database of itemsets and a user specified minsup value, the algorithm finds frequent itemsets using a "bottom up" approach. That is, the algorithm starts with set of frequent itemsets of length 1 (i.e., the cardinality of the number of items in a frequent itemset is 1) and it attempts to find frequent itemsets of length 2. It does so by extending the frequent itemsets of length 1 with one item at a time. This step of extending a frequent itemset with one item is known as *candidate generation*. A candidate is tested to see if it satisfies the minsup threshold before it is added to the set of frequent itemsets. This process is repeated for increasing values on the length of frequent itemsets. During each iteration, candidate itemsets of length k are generated by combining two frequent itemsets of length k-1. The algorithm terminates when no further extensions of the frequent itemset are possible.

For computational efficiency, the Apriori algorithm prunes the set of candidates using a *downward closure lemma* [Agarwal1994] .Given an itemset sequence N , if N is not frequent, then any itemset that contains N is

also not frequent. We illustrate the effectiveness of this lemma using an

example. This example has been adapted from [Survey].

	<m>	<n>	<o>	<p>	<q>	<r>
<m>	<mm>	<mn>	<mo>	<mp>	<mq>	<mr>
<n>	<nm>	<nn>	<no>	<np>	<nq>	<nr>
<o>	<om>	<on>	<oo>	<op>	<oq>	<or>
<p>	<pm>	<pn>	<po>	<pp>	<pq>	<pr>
<q>	<qm>	<qn>	<qo>	<qp>	<qq>	<qr>
<r>	<rm>	<rn>	<ro>	<rp>	<rq>	<rr>

Figure 2 Pre-Apriori

	<m>	<n>	<o>	<p>	<q>	<r>
<m>		<mn>	<mo>	<mp>	<mq>	<mr>
<n>			<no>	<np>	<nq>	<nr>
<o>				<op>	<oq>	<or>
<p>					<pq>	<pr>
<q>						<qr>
<r>						

Figure 3 Post-Apriori

As shown in Figure 2 , the possible number of candidates of length-2 is 36.

With the optimization used by Apriori, as Figure 3 shows, the number of

candidates of length-2 is 15. For this example, Apriori prunes 58% of the exploration space.

Several frequent itemset mining algorithms based on Apriori have been developed [Bastide2000, Brin1997, Sarasere1995]. These papers also show that Apriori provides good run-time performance when the length of frequent itemset is small. However, the performance of Apriori is impacted by two factors:

1. **Pruning efficiency:** If the database consists of datasets with many frequently occurring patterns, then pruning becomes less efficient. For instance, it has been observed that if S consists of frequent itemset of length k, there could be upto $2^{|S|} - 2$ candidates of length $k+1$ [Zaki2002]. This is because the set of candidates consists of the subsets of S. As a result, the computation can become CPU bound.

2. **Number of database scans:** The number of database scans is proportional to the length of the longest frequent itemset. As the length increases, the number of scans also increases. As noted in [Bayardo1998] for real world problems such as patterns in biosequences, itemsets of length 30 or higher is typical.

To address the limitations of Apriori for mining long patterns, alternate approaches have been considered in the literature [Lin2002, Lin2003]. One approach is to mine the database for *closed frequent itemsets*. A frequent itemset N is said to be closed if and only if there does not exist another frequent itemset of which N is a subset. If F denotes the set of frequent itemsets and C denotes the set of closed frequent itemsets, then $C \subseteq F$. It is generally believed that the cardinality of C is much less than F [Zaki2002]. Therefore, if closed frequent itemsets can be efficiently determined, then identifying frequent itemsets is straightforward: for instance, given C, then F consists of all possible subsets of the itemsets in C. Alternately, given C, we can determine if an itemset N is frequent by checking if N is a subset of an itemset in C. Recently, several algorithms for mining closed frequent itemsets have been developed [Zaki2002, Bastide2000, Pei2000, Burdick2001, Lin2003]. In our work, we study one of the algorithms ,*Using Attribute Value Lattice to Find Frequent Itemsets*,[Lin2003] in depth. In the next chapter, we describe the algorithm in detail.

3. Attribute Value Lattice For Mining Closed Frequent Itemsets

In this chapter, we describe the algorithm of Lin, Hu, and Louie [Lin2003] that we have implemented for our work. We begin by describing some preliminaries and then discuss the algorithm.

3.1 Data Representation

The transactions database can be viewed as two-dimensional matrix: the rows represent individual transactions and the columns represent items. For designing data mining algorithms, the data can be represented as either *horizontal* view or a *vertical* view [Lin2003]:

- **Horizontal view** consists of representing each row with a unique transaction identifier and a bitmap to represent the items involved in the transaction. For example, if there could be 10 items involved in a transaction, then the bit-string 1000100010 means that items 1, 5, and 9 were involved.

- **Vertical view** consists of assigning a unique identifier to each column (i.e., item) and a bitmap that represents the transactions in which that particular item is involved. For example, if there are

10 transactions that involve a particular item, then the bitstring

1000100010 means that transactions 1, 5, and 9 are involved.

In their paper, Lin et. al [Lin2003] suggest that a vertical representation is a

natural choice for mining frequent itemsets. This is because the vertical

representation allows operating only those itemsets that are frequent.

Furthermore, for itemsets that are not frequent, their associated bitmap

representation can be discarded, thereby leading to reduced memory

footprint. Consequently, Lin et. al use a vertical representation in their

algorithm.

In the literature the vertical representation of an item in the database is

known as a *granule* [Lin2000, Lin2002, Lin2003-2, Louie2000, Louie2000-

2]. The granule is implemented as a bitmap since it allows fast bit-

manipulation operations.

3.2 Frequent Itemsets and Lattice

A binary relation $\oplus$ that satisfies reflexive, symmetric, and transitive

relationships on a set P is said to be a *partial order* (cite a book on

mathematical logic). That is, $\forall$ *a, b, c* $\in$ P,

- **Reflexive:** $a \oplus a$

- **Symmetric:** $a \oplus b \wedge b \oplus a \Rightarrow a = b$

- **Transitive:** $a \oplus b \wedge b \oplus c \Rightarrow a \oplus c$

The set P under the relation $\oplus$ is a partially ordered set (commonly referred to as *poset*). It is also well known that a poset can be represented as a directed acyclic graph in which the nodes are elements from the set and a path exists from *a* to *b* if and only if $a \oplus b$. A poset is as a *lattice* if all non-empty finite subsets have a *greatest lower bound* and a *least upper bound.* Let $S \subseteq P$ and $u, l \in P$. Then,

- u is the least upper bound if and only if, $\forall s \in S, s \oplus u$

- l is the greatest upper bound if and only if, $\forall s \in S, l \oplus s$

In terms of frequent itemset mining algorithms, the set consisting of granules from the database with the $\subseteq$ relationship defined on the bitmaps is a partial order. To illustrate, if *a, b, c* are granules from the database, then it is easy to see that,

- **Reflexive:** $a \subseteq a$

- **Symmetric:** $a \subseteq b \wedge b \subseteq a \Rightarrow a = b$

- **Transitive:** $a \subseteq b \wedge b \subseteq c \Rightarrow a \subseteq c$

If we restrict the set of granules to those corresponding to frequent itemsets, then that set under the $\subseteq$ relation is a lattice. If we represent the lattice as a directed acyclic graph, then a path in the graph from a node that is a least upper bound to a node that is a greatest upper bound identifies a closed frequent itemset: the nodes (i.e., items) in the path are the members of a closed frequent itemset.

3.3 The Algorithm

Briefly, in designing their algorithm, Lin et. al first construct a lattice of attribute values with the granules that correspond to frequent itemsets. Subsequently, they use the lattice to identify closed frequent itemsets. They do so by generating candidate itemsets from the lattice in a bottom-up breadth-first approach. During candidate generation, the algorithm uses the transitive property of the lattice to prune redundant frequent itemsets that do not result in new closed frequent itemsets. The algorithm, therefore, has two phases:

1. Phase 1 consists of constructing the attribute value lattice

2. Phase 2 consists of exploring the lattice to determine closed frequent itemsets.

3.3.1 Constructing the Lattice

The procedure for constructing an attribute value lattice for items in a database D is shown in Figure 4 below.

The main idea behind this phase of the algorithm is as follows. The database is parsed to get a bitmap for each frequent itemset in the database. Initially the level of each of the itemsets is set at 1. Nodes are constructed, such that each node stores the level and its corresponding bitmap. The nodes we are interested in are only those whose cardinality is greater than minsup. The nodes are sorted based on the bitcount in descending order. These are placed in a priority queue where the priority is set as $(2^L)*B$, where L is the level and B is the bitcount.

The nodes constructed above are then traversed to obtain the attribute value lattice. For traversal, the set of nodes are ordered based on the bitcount. Every node is compared with each node following it and this leads to the generation of the attribute value lattice. The bitmap of the node (I) is intersected with the bitmap of the nodes (J) following it. If such an intersection yields a bitmap whose cardinality is greater than the minsup, then the node (I) is compared with the node (J) in one of three ways.

As outlined in the paper, the three cases are:

1. if $B(I_i) = B(I_j)$, then $B(I_i \cup I_j) = B(I_i) \cap B(I_j) = B(I_i) = B(I_j)$

   [Lin2003]. Consequently, I_i can be replaced by $I_i \cup I_j$. I_j is no longer

   used for the algorithm as it has the same closure as $I_i \cup I_j$.

2. if $B(I_i) \subset B(I_j)$, then $B(I_i \cup I_j) = B(I_i)$; however, $B(I_i) \neq B(I_j)$

   This implies that an edge can be drawn from I_i to I_j because they

   always occur together. However, since the bitmaps $B(I_i) \neq B(I_j)$

   differ from each other, unlike the previous case, I_j would have a

   different closure and removing I_j will cause the algorithm to lose some

   closed frequent itemsets.

3. if $B(I_i) \supset B(I_j)$, then $B(I_i \cup I_j) = B(I_i)$; however, $B(I_i) \neq B(I_j)$. This is

   similar to the previous case, except that an edge is created from I_j to I_i.

Phase One()

1. Construct the bitmap B(I) for each frequent itemset (I) in the

    database.

2. Set level number L of each I as 1

3. Construct the set of nodes, N, that contains I, L and B(I) where

    B(I) > minSup

4. Sort the nodes based on level and bitcount . Have these in a priority

    queue where the priority is set as $(2^L)*B(I)$.

5. For each node I_i in Nodes

        5.1 For each sibling I_j after I_i in Nodes

                5.1.1 $I = I_i \cup I_j$ and $B_{comb} = B(I_i) \cap B(I_j)$

                5.1.2 If B_{comb} > minSup

                        5.1.2.1 If $B(I_i) = B(I_j)$

                                5.1.2.1.1 Remove I_j from Nodes

                                5.1.2.1.2 Replace all I_j with I (i.e $I_i \cup I_j$)

                        5.1.2.2 Else, if $B(I_i) \subset B(I_j)$

                                5.1.2.2.1 Create an edge from I_i to I_j

                                5.1.2.2.2 $L_j = Max (L_j, L_i + 1)$

                        5.1.2.3 Else, if $B(I_j) \subset B(I_i)$

This pseudo code has been taken verbatim from Dr. Lin's Lattice paper [Lin 2003].
Figure 4 Constructing Attribute Value Lattice

We illustrate the steps in the algorithm using the example from This example **has been adapted from Dr. Lin's Lattice paper [Lin 2003].**

Figure 1. With a minsup of 3, we have:

- $B(A) = 1011100000$

- $B(D) = 0101110000$

- $B(T) = 1010110000$

- $B(K) = 0000111100$

- $B(Q) = 0000001111$

N = {}

C = {}

Iteration 1:

I = {AD}

$B_{comb} = 0001100000$

$| B_{comb}| = 2 < 3.$

//N contains A's parents

N = {WC}

Iteration 2:

I = {AT}

$B_{comb} = 101010000$

$| B_{comb}| = 3$

N = {AD}

Figure 4 shows resulting the lattice that corresponds to the sample database

from This example **has been adapted from Dr. Lin's Lattice paper [Lin 2003].**

Figure 1. For this lattice, we used a minsup of 3 (i.e., an item appears in 30%

of the transactions).
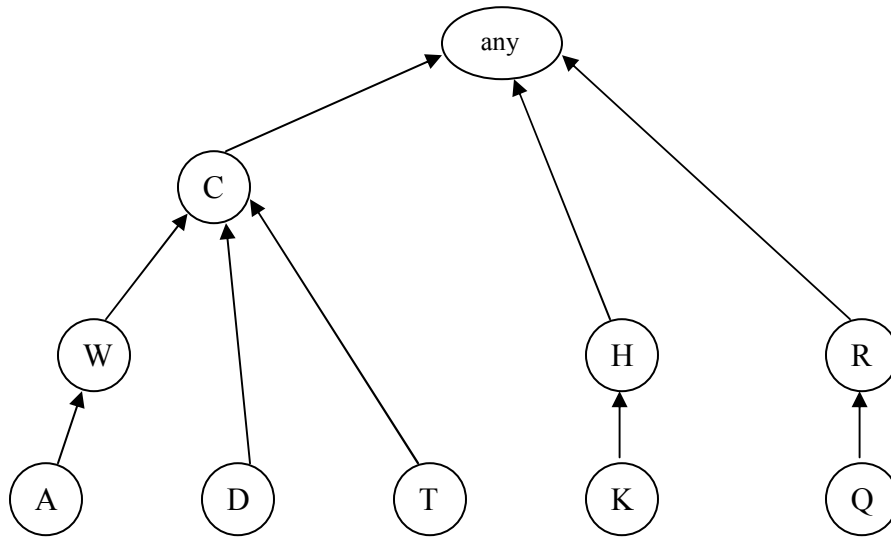
Figure 5 Attribute Value Lattice

3.3.2 Identifying Closed Frequent Itemsets

The procedure for identifying closed frequent itemsets from the lattice is

shown in Figure 6 Revised Algorithm below. In this phase of the algorithm,

we build on the lattice by using the set of nodes at the same level for

candidate generation: the nodes are sorted in decreasing order of bitcount;

each node is combined with its siblings in a breadth-first manner. Then,

expansion is performed on the set of candidates in increasing order of levels in a bottom-up approach. That is, starting with the level-1 leaf nodes of the lattice corresponding to the set of frequent itemsets.

We describe the workings of the algorithm using the attribute value lattice from Figure 5. The algorithm starts with nodes in the order A, D, T, K, Q. Next, when AD is combined, we find that AD is not-frequent. Since WD could be frequent, the algorithm adds W to the set of nodes for next round expansion. The algorithm then considers AT, AK, AQ in that order. After the siblings of A are exhausted, the algorithm then considers DT, DK, DQ in that order and so on. Since A, D, T, K, Q are frequent itemsets, they are added to F. After level-1 nodes are exhausted, the algorithm then uses the level-2 nodes for next round of expansion. This process continues until there are no further nodes for expansion.

This expansion phase of the algorithm could be viewed as augmenting the lattice with additional frequent itemsets constructed using the nodes of the lattice itself. At the end of this phase, we have the lattice setup for finding the closed frequent itemsets: To illustrate, as pointed out earlier, let us use the directed acyclic graph view of the lattice. Then, a path in the graph from the leaf node to the root represents a closed frequent itemset. The overall

procedure that combines the various phases and returns the set of closed

frequent itemsets is shown in Figure 7.

Procedure ExpandFreqItemSet(Nodes, minsup)

1. For every node $I_i \in$ Nodes

    1.1 NewNodes = Ø, I = I_i

    1.2 For each sibling I_j after I_i in Nodes

        1.2.1 I = $I_i \cup I_j$ and B_{comb} = B(I_i) $\cap$ B(I_j)

        1.2.2 If B_{comb} > minSup

            1.2.2.1 Add I x B_{comb} to the NewNode

        1.2.3 Else add I_i's parents to the NewNode

    1.3 F = F $\cup$ I

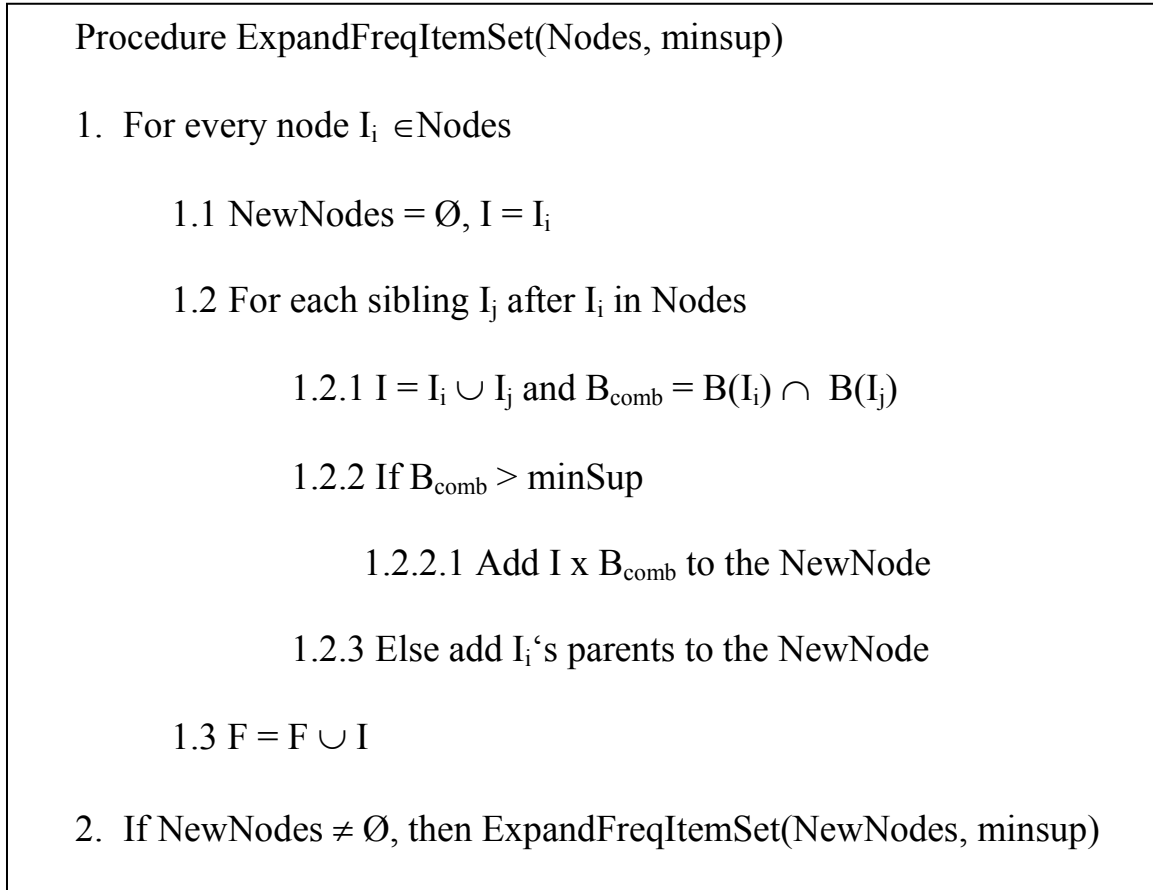2. If NewNodes ≠ Ø, then ExpandFreqItemSet(NewNodes, minsup)

Figure 6 Revised Algorithm

```
Main()

   1.  C = { } // set of closed frequent itemsets

   2.  F = { } // set of frequent itemsets

   3.  Construct attribute value lattice (i.e., Phase one)

   4.  Expand frequent itemsets (i.e., Phase 2)

   5.  For every node $I_i \in F$, add the ancestor set of $I_i$ to C
```

Figure 7 Procedure for Finding Closed Frequent Itemsets

3.4 Issues In Implementing the Algorithm

We faced the following issues in implementing the algorithm:

1. In Phase 1, the sorting of the nodes was just based on the cardinality
   of the bitmaps as defined in the paper. We modified this to include
   both the level and the cardinality and set this to (2^L) * B. This will
   improve the speed of the algorithm because the new nodes are not
   added to the end of the list. Instead, it is inserted based on a priority
   and therefore can be fetched faster.

2. The indentation of the algorithm for Phase 2 was incorrect.   In
   particular, line 8 should be in the loop of statement 2; in the pseudo-
   code in the paper, it is outside the loop (see Figure 8).

3. The specification for Phase 2 of the algorithm in the paper by Lin et. al is imprecise. For instance, in the original specification, line 3 says "continue the expanding", when it actually means a recursive call to a procedure. The specification presented in Figure 6 addresses such issues.

---

1. Nodes = all the greatest lower bounds items of the lattice

2. For every node I_i in Nodes

    2.1 NewNodes = Ø, $I = I_i$

    2.2 For each sibling I_j after I_i in the nodes

        2.2.1 $I = I_i \cup I_j$ and $B_{comb} = B(I_i) \cap B(I_j)$

        2.2.2   If $|B_{comb}| > minsup$

            2.2.2.1   Add $I* B_{comb}$ to the NewNode

        2.2.3   Else

            2.2.3.1   Add I's parents to the NewNode

        2.2.4   If NewNodes ≠ Ø, then continue the expanding

3. /* expand the frequent nodes */

4. $C = C \cup I$

5. For every node $C_i \in C$, replace it by its ancestor set

---

This pseudo code has been taken verbatim from Dr. Lin's Lattice paper [Lin 2003].
Figure 8 Original Algorithm

4. In line 1.2.2.1, we add IxBcomb to the set of nodes for expansion. However, the specification does not define what the parents of the newly combined node should be. This is noteworthy because the parents of a node are used to identify additional frequent itemsets. We addressed this issue by setting the parent of a combined node I to be $P(I_i) \cup P(I_j)$.

5. In line 1.2.2.1, we add I_i's parents to the set of nodes for expansion. Observe that, the specification does not include I_j in that set. This could have the effect of not generating some closed frequent itemsets from the algorithm. For instance, for the lattice in Figure 5, if AD is not frequent, then only W is added to the new node set, but D is not. As a result, the algorithm does generate WD as a candidate frequent itemset (note that, it is possible that WD is a frequent itemset). In our implementation, we considered I_j to the new node set. For some datasets explored in our work, adding I_j significantly increased the running time of the algorithm to the point that the algorithm continued to execute for several hours without terminating. Hence, we did not change this line of the algorithm in our implementation. That is, we implemented this line of the algorithm as specified in the paper.

4. Experimental Evaluation

4.1 Implementation

We implemented the algorithm described in the previous chapter using the Java programming language. In addition to Java classes for implementing the algorithm, we implemented helper classes for doing buffered I/O and fast bit-manipulation operations. The Java classes used for this algorithm are the following: lattice.java, NodeInfo.java, ItemSetInfo.java, and AttrValueLattice.java. The helper classes for this include BitClass.java, DiskReader.java, and Timer.java. Details of the Java classes are as follows:

BitClass.java:

For constructing granules using bitmaps, we had two choices: use Java's BitSet class or develop a custom implementation given the characteristics of our dataset. For common bit manipulation operations such as "and", "or", "cardinality", "set", and "clear", we timed the native implementation and our implementation and for the most part, our implementation was faster than Java's BitSet class.

DiskReader.java:

This program simulates disk-reads by reading in data from a file into memory in 4K chunks. The 4K chunk of data in memory is used to build the Granular model directly. Once this is built, the next 4K chunk is fetched from disk. This ensures that we use memory judiciously, especially when we are dealing with large datasets.

ItemSetInfo.java:

This program implements the data structure for holding the bitmaps corresponding to each unique value in a column.

Algorithm.java:

In this program, we set a variable maxValsPerColumn that keeps track of the maximum number of $(n-1)$ large itemsets before we move on to n – large itemsets. Limiting the number of (n-1) large itemsets is beneficial because we can index into an array to generate the n large array by intersecting the (n-1) large and 1-large itemsets. This array is a two dimensional array in which the first dimension keeps track of which large itemset we are building and the second dimension keeps track of the values obtained by intersecting 2 columns. This dimension has a maximum index which limits how many

values we generate. Though limiting the number of values hinders

completeness of results, it ensures better scalability by reducing memory

usage.

Lattice.java:

This is the driver class and reads from an input file. An object of the class

AttrValueLattice is instantiated here which then makes the bitmap, makes

the nodes based on the minSup, combines nodes and finds the closed

frequent itemsets.

AttrValueLattice.java:

This class implements both phases of the lattice algorithm as elicited by the

pseudocode shown in Figure 4, Figure 5 and Figure 6. The main functions of

this class is to MakeNodes (to create nodes with their bitcount and parent

information), CombineNodes (to combine a pair of nodes by intersecting

their bitmaps and taking a union of their set of parents), ExpandItemSets (for

generating candidate frequent itemsets), and FindClosedFrequentIemSets.

4.2 Experimental Setup

The experiments were done on an Apple Powerbook laptop with 1GB RAM.
The input file for each experiment was stored on the laptop's disk (i.e., local
disk). Data was read in from disk for building the bitmaps when
constructing the lattice and then discarded. This helped reduce the memory
footprint for our implementation.

4.3 Data Characteristics

In this thesis, we study the performance of the algorithm using synthetic
data. We model the occurrence of an item in a transaction based on
mathematical distributions. For each distribution, we generate a dataset that
consists of numbers to represent items, where the numbers are based on the
distribution. That is, when items in the database are modeled using a
particular distribution, this means that the probability of an item being in a
transaction depends on the characteristics of that distribution. Since the size
of the dataset could have an impact on the running time of the algorithm, we
also study the performance of the algorithm for datasets of varying sizes.
The distributions we considered in our work are:

- **Normal distribution**: The probability density function (Figure 9) for the normal distribution is : $F(x; \mu, \sigma) = 1/\sigma \sqrt{(2\Pi)} \, e^{(-(x-\mu)^2)/((2\sigma^2))}$ [Wikipedia]

- **Exponential distribution**: The probability density function for this (Figure 10)distribution is: $F(x ; \lambda) = \lambda e^{-\lambda x}$, when x >=0 and 0 when x < 0 [Wikipedia]

- **Zipf distribution**: The Zipf law was proposed by a Harvard University linguist George Zipf. This law was put forth as it applied to language, i.e., the frequency of some words in any language is much higher than the frequency of others. When such a frequency is plotted against the rank of such a parameter, the rank and frequency become inversely proportional. Another observation typical of such a dataset is that, when drawn to logarithmic scales, the most frequently occurring and the least frequently occurring data lie close to the axes of the graph. Zipf's law (Figure 11) is given by the following:

  $F(k; s, N) = (1 / k^s) / (\Sigma^N_{n=1} 1 /n^s)$ where,

  N is the number of elements, k is the rank, and s is the exponent characterizing the distribution [Wikipedia].
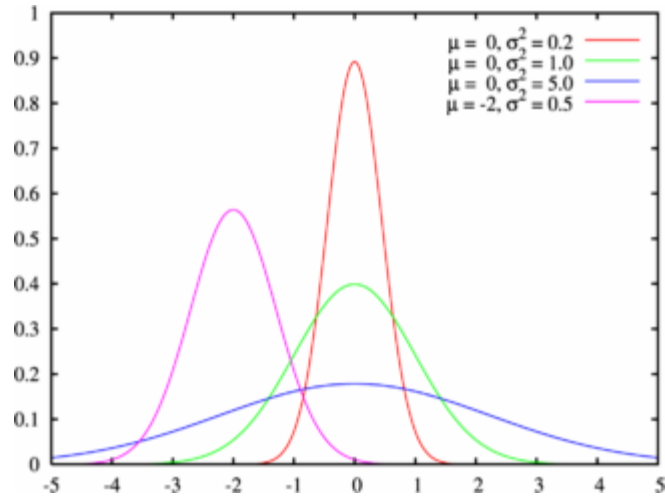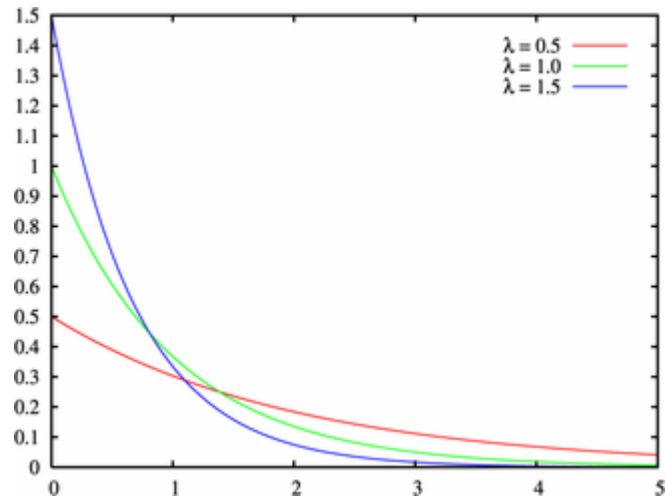
Figure 9 Normal Distribution [Wkipedia]

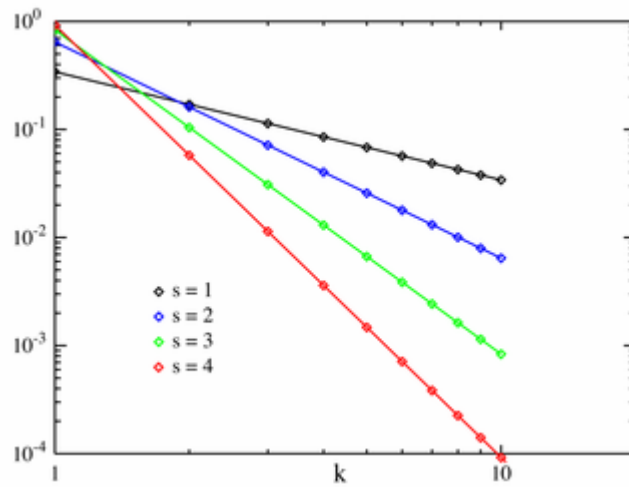**Figure 10 Exponential Distribution [Wikipedia]**

Figure 11 Zipf Distribution [Wikipedia]

4.4 Results

For each distribution, there are two parameters that impact the running time of the algorithm:

1. **Input data size:** What is the impact of increasing the dataset size

2. **Minsup value:** How does changing the minsup affect the running time

In our results, we present the running times and also show the line of best fit for the data. Also, we present the number of closed frequent itemsets identified by the algorithm.

Non-linear regression was used for fitting the curves in Figures 12 - 19. We used GraphPad Prism Software version 4.03 [Trial], February 02, 2005. GraphPad Software is located at San Diego USA, www.graphpad.com.

4.4.1 Normal Distribution
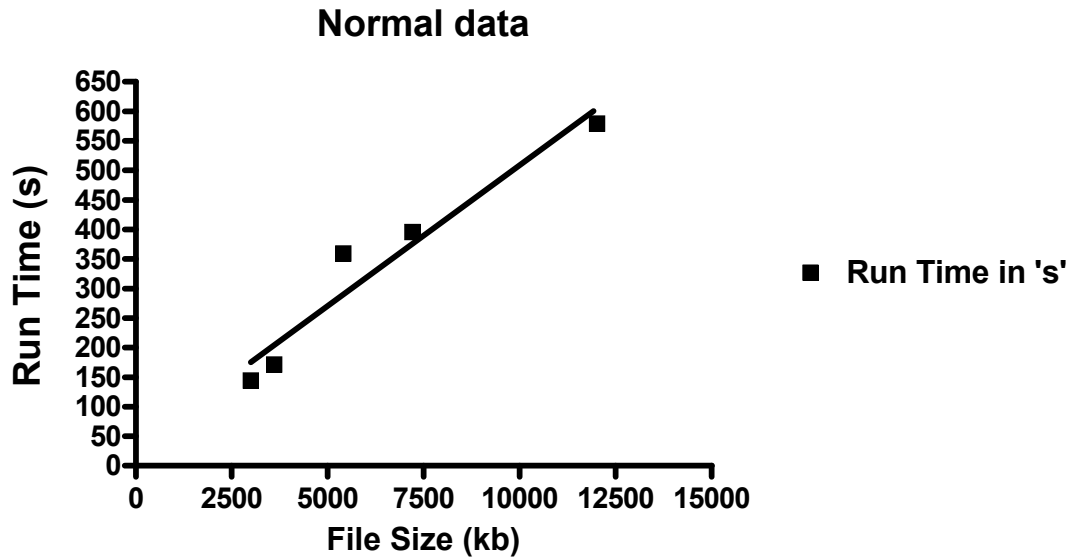
Normal data

File Size	run Time in 's'	closed frequent itemsets	minSup
2996.495	144.357	2124	12
3603	171.415	2101	14
5404.5	359.627	2111	73
7206	395.822	2207	97
12010.05	579.319	2119	100

Figure 12 Run Time Vs File Size – Normal Data

$$F(x;\mu,\sigma) = 1/\sigma\sqrt{(2\Pi)}\; e^{(-(x-\mu)^2)/((2\sigma)^2))} \quad \text{[Wikipedia]}$$

Best-fit values
    Slope   0.04767 ± 0.007099
    Y-intercept when X=0.0   32.47 ± 49.93
    X-intercept when Y=0.0   -681.2
    1/slope   20.98
95% Confidence Intervals
    Slope   0.02508 to 0.07026
    Y-intercept when X=0.0   -126.4 to 191.3
    X-intercept when Y=0.0   -7263 to 1890
Goodness of Fit
    r²   0.9376
    Sy.x   51.38

normal-samesize (file size = 7206 kB)

minSup	run Time 's'
100	120.145
98	224.437
97	301.506
96	395.822
95	527.39

Figure 13 Run time Vs Minsup – Normal Data

Best-fit values
   Slope   0.04767 ± 0.007099
   Y-intercept when X=0.0   32.47 ± 49.93
   X-intercept when Y=0.0   -681.2
   1/slope   20.98
95% Confidence Intervals
   Slope   0.02508 to 0.07026
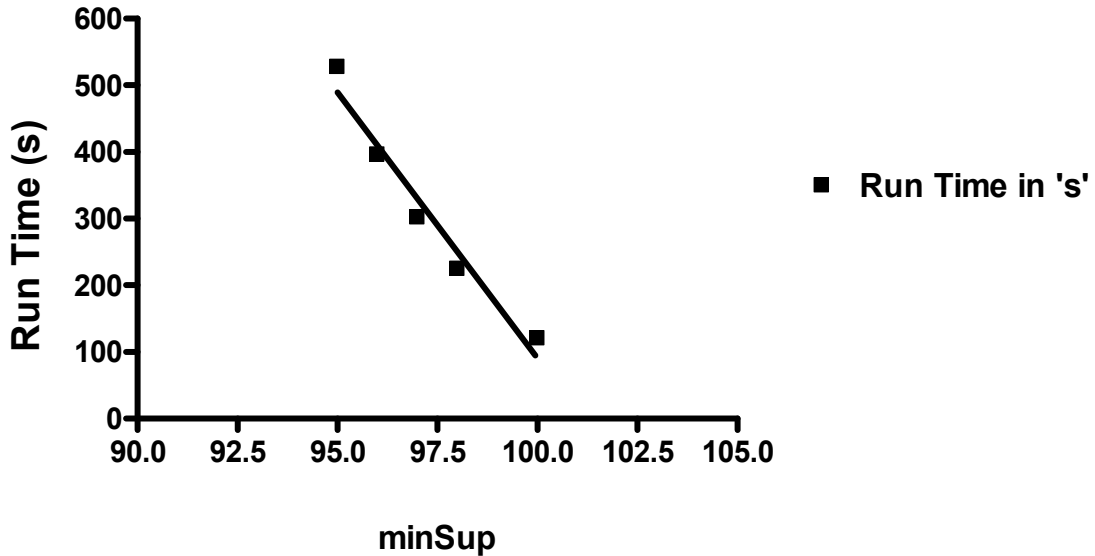   Y-intercept when X=0.0   -126.4 to 191.3
   X-intercept when Y=0.0   -7263 to 1890
Goodness of Fit
   r^2 0.9376
   Sy.x   51.38

The graphs are shown in Figure 12 and Figure 13. From the figures, we

make the following observations:

1. As we increase the size of the input dataset, the running time increases almost linearly.

2. As we increase the minsup value, for a given dataset, running time decreases. This is to be expected because as minsup is increased, the number of frequent itemsets decreases. Conversely, for a given minsup, as we increase the size of the dataset, the number of closed frequent itemsets increases. This is also expected----as the size of dataset increases, there are more transactions, and hence, the number of frequent itemsets increases.

4.4.2 Exponential Distribution
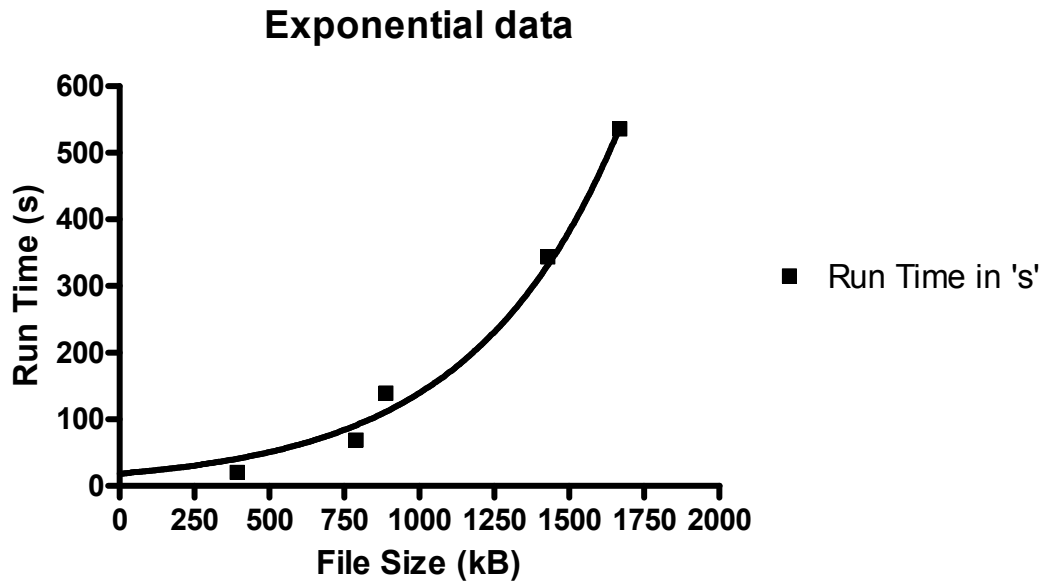
Exponential data

File Size 'kB'	run Time 's'	closed frequent itemsets	minSup
395.317	19.294	1267	20
790.526	67.599	1900	20
890.821	138.392	1978	20
1430.603	342.775	3198	20
1670.086	534.528	3968	20

Figure 14 Run Time Vs File Size – Exponential Data

Exponential growth
Best-fit values
    START   18.54
    K   0.002019
    Doubling Time   343.3
Std. Error
    START   5.400
    K   0.0001848
95% Confidence Intervals
    START   1.360 to 35.73
    K   0.001431 to 0.002607
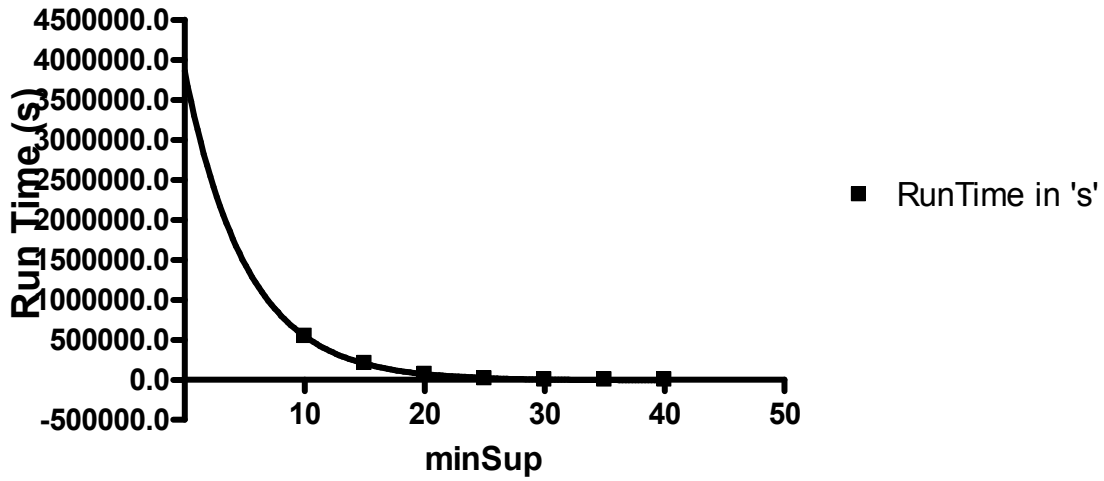   Doubling Time   265.9 to 484.4
Goodness of Fit
    Degrees of Freedom   3
    R² 0.9898
    Absolute Sum of Squares   1871
    Sy.x   24.98

expData-samefile (file-size = 890.821 kB)

minSup	runTime 's'
40	5
30	90
35	1530
30	13603
25	67599
20	207478
15	543597

Figure 15 Run Time Vs Minsup – Exponential Data

$F(x ; \lambda) = \lambda e^{-\lambda x}$ [Wikipedia]

One phase exponential decay
Best-fit values
   SPAN   3.870e+006
   K   0.1946
   PLATEAU -7627
   HalfLife   3.561
Std. Error
   SPAN   266957
   K   0.006876
   PLATEAU 4133
95% Confidence Intervals
   SPAN   3.129e+006 to 4.611e+006
   K   0.1755 to 0.2137
   PLATEAU -19100 to 3845
   HalfLife   3.243 to 3.948
Goodness of Fit
   Degrees of Freedom   4

R²   0.9993
Absolute Sum of Squares   1.814e+008
Sy.x   6735

The graphs are shown in Figure 14 and Figure 15. From the figures, we

make the following observations:

1. As we increase the size of the input dataset, the running time increases

    exponentially.

2. As we increase the minsup value, for a given dataset, running time

    decreases with an exponential decay. The reasons for this are similar

    to the behavior of normal distribution dataset:

    a. For a given dataset, as we increase minsup, the number of

        closed frequent itemsets decreases.

    b. For a given minsup, as we increase the size of input data, the

        number of closed frequent itemsets increases.

4.4.3 Zipf Distribution
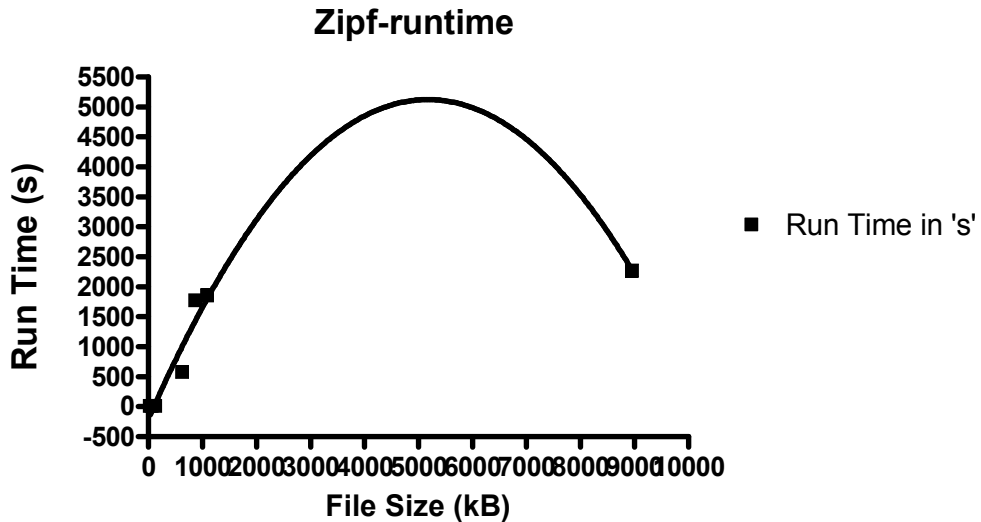
Zipf-runtime

file size in kB	time in s	closed freq-itemsets	minSup
33.925	0.071	1	10
67.979	0.105	6	20
136.163	0.12	9	30
635.486	568.612	50	200
879.23	1760.606	60	400
1099.067	1847.93	70	500
8971.779	2257.052	152	750

Figure 16 Run Time Vs File Size – Zipf Data

Polynomial: Second Order ($Y=A + B*X + C*X^2$)
Best-fit values
   A   -202.4
   B   2.058
   C   -0.0001989
Std. Error
   A   180.0
   B   0.3157
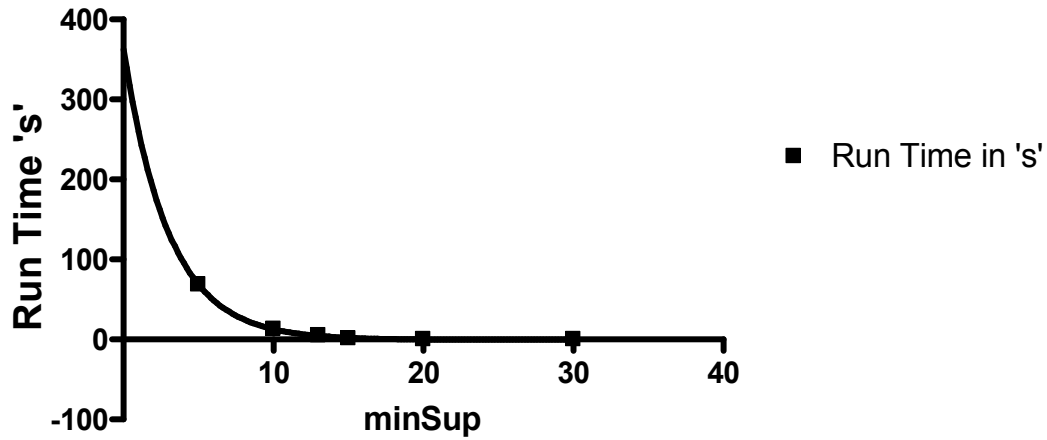   C   3.378e-005
95% Confidence Intervals
   A   -702.2 to 297.3
   B   1.182 to 2.935
   C   -0.0002926 to -0.0001051
Goodness of Fit
   Degrees of Freedom   4
   R² 0.9450
   Absolute Sum of Squares 330808

Sy.x   287.6

Zipf-same file (file-size=136.163 kB)

minsup	Run Time
5	68.686
10	13.032
13	4.648
15	0.866
20	0.21
30	0.12

Figure 17 Run Time Vs Minsup – Zipf Data

$$F(k; s, N) = (1 / k^s) / (\Sigma^N_{n=1} 1 / n^s) \text{ [Wikipedia]}$$

One phase exponential decay
Best-fit values
   SPAN   363.3
   K   0.3318
   PLATEAU -0.4363
   HalfLife   2.089
Std. Error
   SPAN   25.01
   K   0.01425
   PLATEAU 0.5399
95% Confidence Intervals
   SPAN   283.8 to 442.9
   K   0.2865 to 0.3771
   PLATEAU -2.154 to 1.282
   HalfLife   1.838 to 2.420

Goodness of Fit
    Degrees of Freedom   3
    R²   0.9995
    Absolute Sum of Squares   1.909
    Sy.x   0.7978

The graphs are shown in Figure 16 and Figure 17. From the figures, we make the following observations:

1. As we increase the size of the input dataset, the running time increases and then stabilizes. This is because of the characteristics of Zipf data: there are very few unique values in a Zipf distribution; as we increase the dataset the number of itemsets for a given minsup stabilize and hence, there is not a noticeable increase in running time.

2. As we increase the minsup value, for a given dataset, running time decreases as an exponential decay. This is again due to the characteristics of the Zipf distribution.

4.5 Discussion

Of the three distributions we studied, our results showed that Zipf data performs better compared to the other two. This is because with Zipf-distributed data, the numbers are clustered around a few values (i.e., very few items in the database appear in most of the transactions). On the other hand, with the remaining distributions, the data is unlikely to be clustered.

For instance, with normal distribution, every item can appear in every transaction with uniform probability. As a result, the number of frequent itemsets for such distributions can be large.

While the procedure for finding closed frequent itemsets tries to provide accurate answers, there are datasets for which the running time is exponential. Rather than obtain accurate answers, it may be worthwhile to obtain an approximate answer and then refine the search. For example, suppose there is a merchant who sells millions of items. To answer a query such as, find the top hundred frequently bought items, we need to determine closed frequent itemsets over the data with a minsup of 100. If such a set is large, we could instead represent the data into categories and then try to find the top n-categories. From such a frequent category set, we could find the desired closed frequent itemsets. Note that this procedure is lossy: since we are restricting the search to the top categories, we may miss closed frequent itemsets that are not in the top categories. Procedures such as the one outlined in this example are data binning techniques.

Of the distributions studied, Zipf distribution has polynomial run-time and is faster than the other two. Hence, we develop a method to bin data such that

the resulting binned data resembles a Zipf distribution. We illustrate our ideas using an example. Consider data from a normal distribution that is binned into bins of equal width. We construct a histogram from the data for each bin. We then place histogram buckets with the same frequency into the same bin. The resulting distribution is like Zipfian.

To apply our idea to input data, we use Chi-square test [Press] to see which distribution the data matches closely. That is, we evaluate column-wise (i.e., granule) the characteristics of the input data. For each column, we compute a chi-square for the distribution for that column using non-linear least squares method of Levenberg-Marquardt. The recipe for this procedure is defined in pages 683 - 687 of the Numerical Analysis text [Press]. The resulting chi-square value is compared to the chi-square of known distributions such as Normal, Exponential, and Zipf to identify degree of similarity. Then, if the data resembles exponential or normal distribution, binning is required. For Zipf data, binning is not required---as our results showed, the running time of the algorithm for Zipf distribution is polynomial.

The procedure for binning the data is as follows. From the input data, we construct a histogram for each granule. For the histogram, we divide the data into uniform sized bins of a given bin width. We then make another pass over the data and for each input value, we compute the logarithm of the frequency of its bin. Now, if this computed value is above a threshold, this computed value is used to represent the data; otherwise, the original value is used as is. This has the effect of transforming the data from a large set of values to a few values and thereby mimics Zipf data. As a result of binning in this manner, the number of level-1 nodes in the lattice is significantly reduced.

As proof of concept, we performed experiments using two used sets of data for binning. First, we use data from exponential distribution as input to the binning procedure. The procedure identifies the data as being exponentially distributed (as expected) and we then bin it. This experiment serves to validate our binning procedure i.e., provide input from known distribution and it should be mapped to the same distribution. Second, we then apply the procedure to a "mixed" data set---one that contains data from both exponential and Zipf. As expected, the procedure only bins columns that

belong to the exponential distribution. The results are explained in the next section.

4.6 Results

4.6.1 Exponential binned data

From the figures, we make the following observations:

1. As we increase the size of the input dataset, the running time is nearly constant: This is very similar to the results of the Zipf distribution.

2. As we change the minsup value, for a given dataset, running time decreases rapidly (again, very similar to that of the Zipf distribution).

This experiment serves to verify our methodology: our idea was transform the input data to something that mimics Zipf distribution and thereby reduce running time of the algorithm. These graphs validate our ideas. We now consider mixed data sets: data sets that contain a mix of Zipf, normal, and exponentially distributed data. We apply our methodology and bin only the columns in the input dataset that closely resemble either normal or exponentially distributed data based on the procedure outlined in the previous section. Our results follow.

4.6.1.1 Binned Exponential Data

Binned Exponential
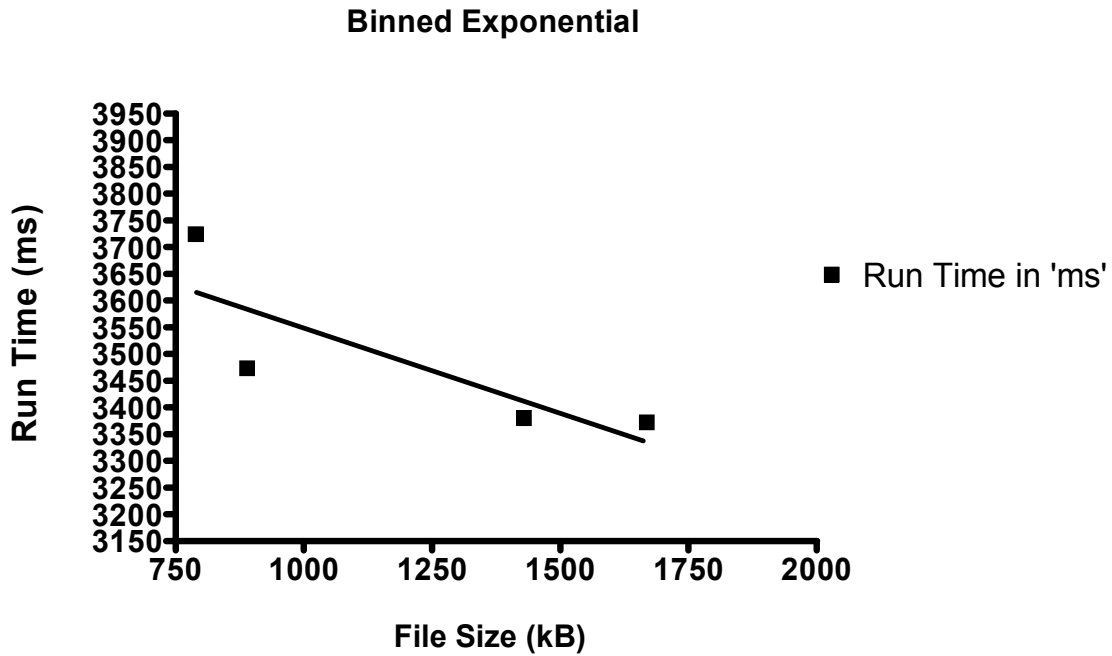
File Size 'kB'	run Time 'ms'
790.526	3723
890.821	3472
1430.603	3379
1670.086	3371

Figure 18 Run Time Vs File Size – Binned Exponential Data

Best-fit values
    Slope   -0.3191 ± 0.1565
    Y-intercept when X=0.0   3868 ± 195.7
    X-intercept when Y=0.0   12120
    1/slope   -3.134
95% Confidence Intervals
    Slope   -0.9927 to 0.3545
    Y-intercept when X=0.0   3025 to 4710
    X-intercept when Y=0.0
Goodness of Fit
    r^2 0.6750
    Sy.x   114.8

4.6.2 Mixed Data

The results for the mixed data sets *without binning* are shown in Figure

From the graphs in Figure 19, Figure 20, Figure 23 and Figure 24, we make

the following observations:

1. As we increase minsup, running time decreases. This is to be

   expected as there is a decrease in the number of closed frequent

   itemsets.

2. As we increase the size of the input dataset, the running time of the

   algorithm increases. The rate of increase depends on the input

   distribution: For instance, if the input data is a mix of Zipf and

   Normal distributions, the rate of decrease is similar to that of the

   normal distribution. A similar result holds for a mix of Zipf and

   exponentially distributed dataset.

The results for three mixed data sets *with binning* are shown in Figure 21,

Figure 22, Figure 25 and Figure 26. From the graphs, we make the following

observations:

1. As we increase minsup, running time decreases. This is to be

   expected as there is a decrease in the number of closed frequent

   itemsets.

2. As we increase the size of the input dataset, the running time of the

   algorithm is nearly constant. That is, the results are very similar to

   that of the Zipf distribution. Since our procedure only bins data in the

   columns corresponding to either Exponential or Normal distribution,

   the transforms the input dataset to a dataset that closely resembles

   Zipf distribution.

4.6.2.1 Unbinned Zipf Exponential Data
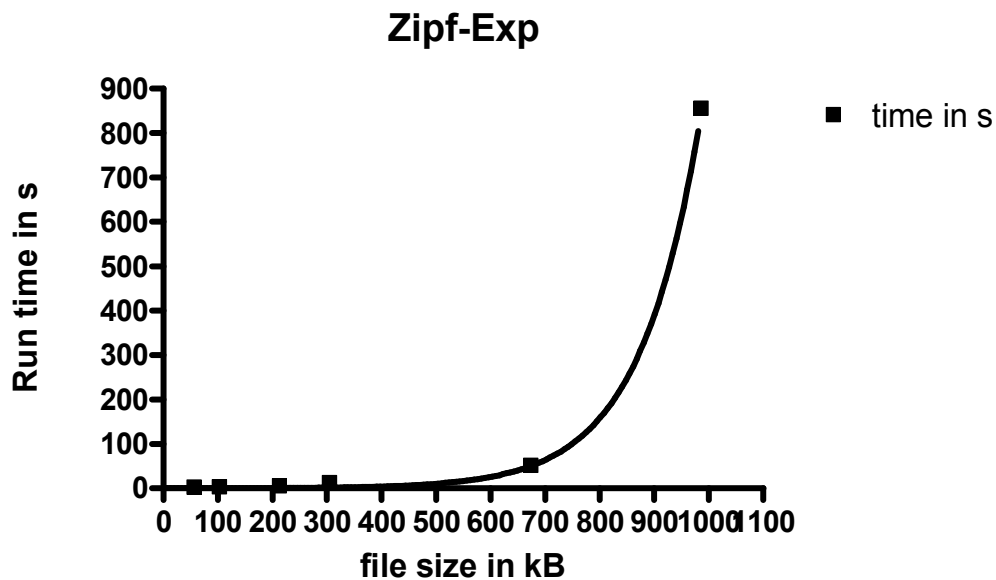
Zipf-Exp

file size in kB	time in s	closed freq.itemsets	minSup
57.552	1	1	7
104.127	2.22	155	10
214.33	4.346	138	18
306.008	11.609	991	30
675.213	50.213	1100	75
987.08	854.138	95	250

Figure 19 Run Time Vs File Size – Zipf Exponential Data (Unbinned)

Exponential growth
Best-fit values
   START   0.1153

K   0.009027
Doubling Time   76.79
Std. Error
    START   0.03789
    K   0.0003332
95% Confidence Intervals
    START   0.01012 to 0.2205
    K   0.008102 to 0.009952
    Doubling Time   69.65 to 85.55
Goodness of Fit
    Degrees of Freedom   4
    R²   0.9998
    Absolute Sum of Squares   113.5
    Sy.x   5.328

zipf-exp (file-size = 214.440kB)
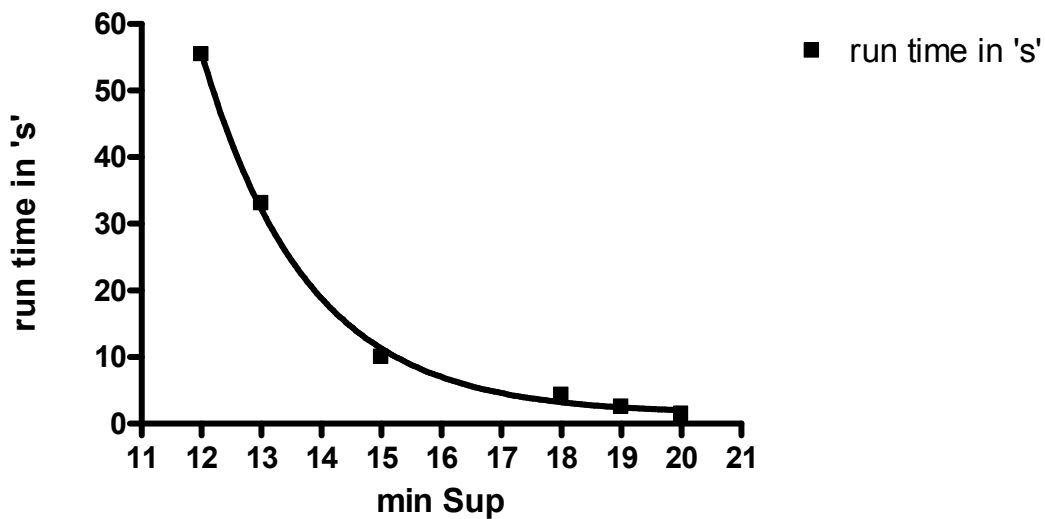
min Sup	run time in 's'
12	55.415
13	33.085
15	10.009
18	4.346
19	2.547
20	1.482

Figure 20 Run Time Vs Minsup – Zipf Exponential Data (Unbinned)

One phase exponential decay
Best-fit values
    SPAN   49833
    K   0.5685

PLATEAU 1.433
HalfLife   1.219
Std. Error
    SPAN   23793
    K   0.03986
    PLATEAU 0.8443
95% Confidence Intervals
    SPAN   -25880 to 125542
    K   0.4417 to 0.6953
    PLATEAU -1.254 to 4.119
    HalfLife   0.9968 to 1.569
Goodness of Fit
    Degrees of Freedom   3
    R^2 0.9983
    Absolute Sum of Squares 4.112
    Sy.x   1.171

4.6.2.2 Binned Zipf Exponential Data
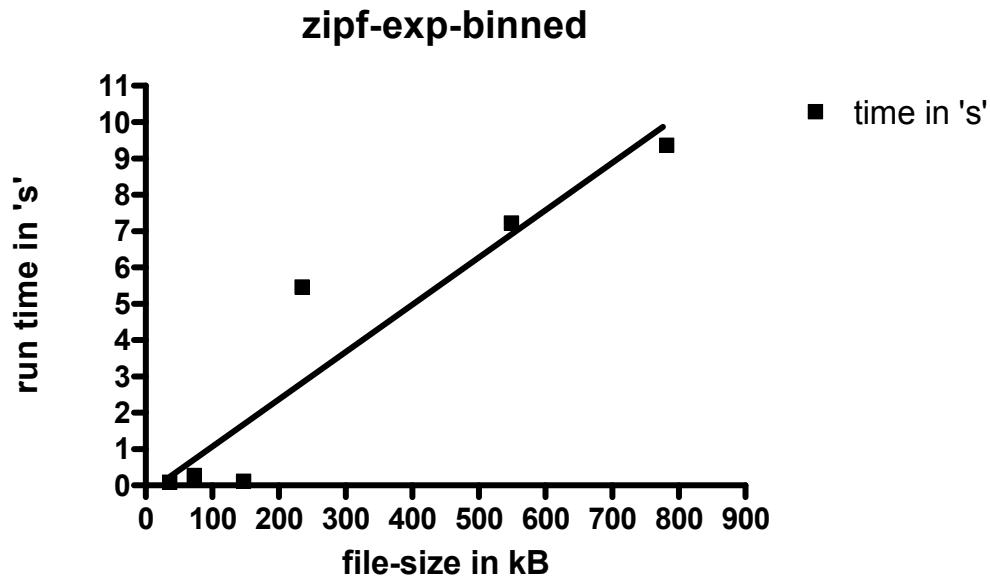
zipf-exp-binned

file-size	time in 's'	closed freq.itemsets	minSup
37.468	0.065	1	7
74.169	0.2497	1	10
148.366	0.1	1	18
236.316	5.44	1	30
550.213	7.2	1	75
783.48	9.34	1	250

Figure 21 Run Time Vs File Size – Zipf Exponential Data (Binned)

Best-fit values
    Slope   0.01302 ± 0.002379
    Y-intercept when X=0.0   -0.2386 ± 0.9718
    X-intercept when Y=0.0   18.33
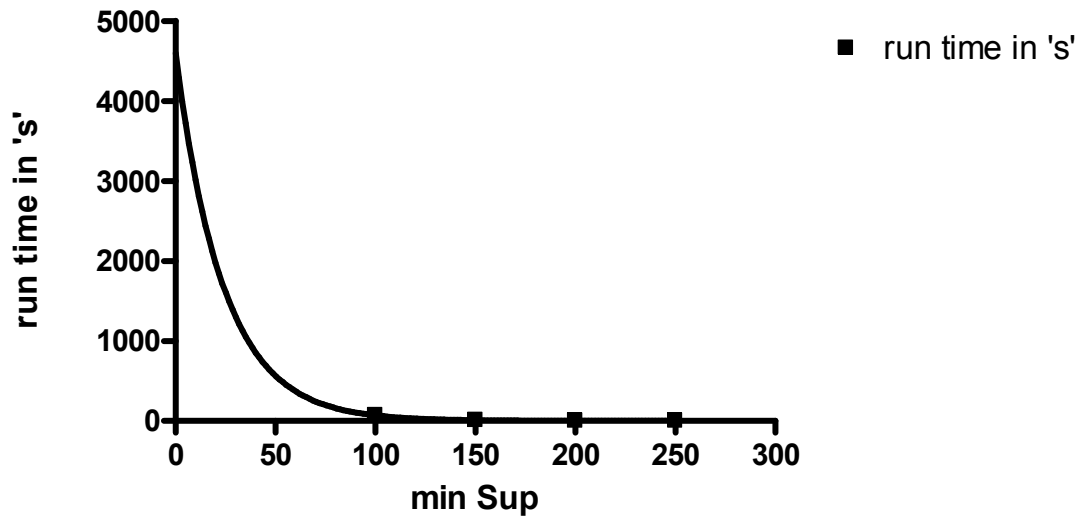   1/slope   76.81
95% Confidence Intervals
    Slope   0.006416 to 0.01962
    Y-intercept when X=0.0   -2.936 to 2.459
    X-intercept when Y=0.0   -333.8 to 171.8
Goodness of Fit
    r²   0.8822
    Sy.x   1.584

zipf-exp-binned (file-size=783.480 kB)

min Sup	run time in 's'
100	70.358
150	9.346
200	2.245
250	0.74

Figure 22 Run Time Vs Minsup – Zipf Exponential Data (Binned)

One phase exponential decay
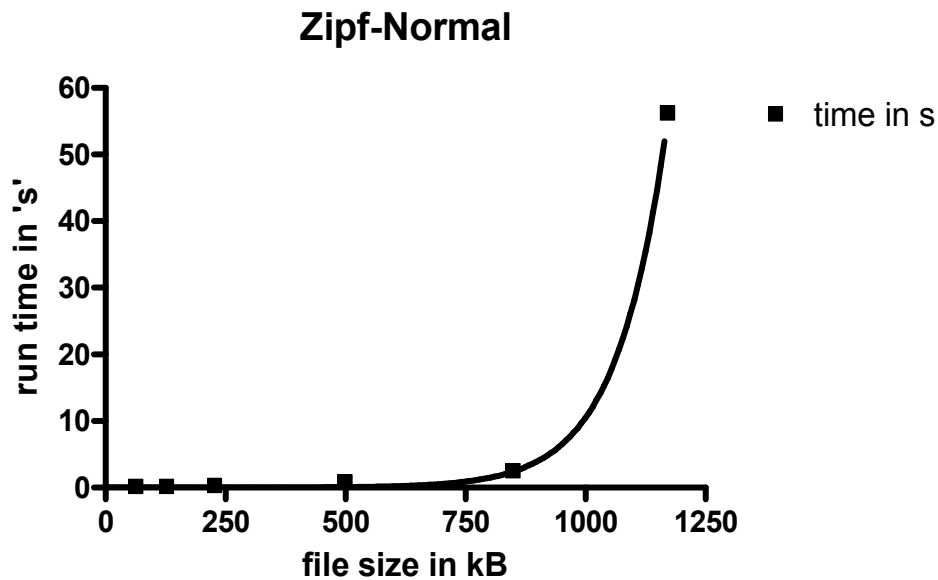Best-fit values
    SPAN   4596
    K   0.04192
    PLATEAU 0.8708
    HalfLife   16.54
Std. Error

SPAN   592.2
K   0.001315
PLATEAU 0.3484
95% Confidence Intervals
SPAN   -2929 to 12121
K   0.02521 to 0.05862
PLATEAU -3.555 to 5.297
HalfLife   11.82 to 27.49
Goodness of Fit
Degrees of Freedom   1
R²   0.9999
Absolute Sum of Squares 0.1771
Sy.x   0.4208

4.6.2.3 Unbinned Zipf Normal Data

file size in kB	time in s	closed freq.itemsets	minSup
64.516	0.05	1	10
128.934	0.083	10	20
228.947	0.235	14	30
500.23	0.75	20	70
850.34	2.4	35	120
1172.338	56.128	51	300

Figure 23 Run Time Vs File Size – Zipf Normal Data (Unbinned)
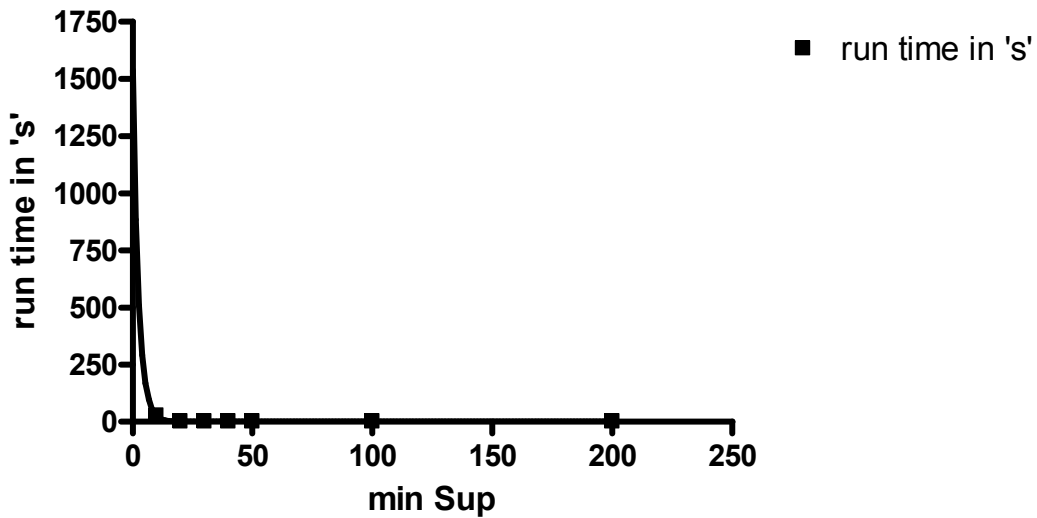
Exponential growth
Best-fit values
START   0.0006259
K   0.009728

Doubling Time   71.26
Std. Error
   START   0.0003320
   K   0.0004527
95% Confidence Intervals
   START   -0.0002959 to 0.001548
   K   0.008471 to 0.01098
   Doubling Time   63.10 to 81.83
Goodness of Fit
   Degrees of Freedom   4
   R²   0.9998
   Absolute Sum of Squares 0.5110
   Sy.x   0.3574

Zipf-normal (file-size=228.947 kB)

min Sup	run time in 's'
10	24.99
20	0.482
30	0.245
40	0.096
50	0.067
100	0.019
200	0.003

Figure 24 Run Time Vs Minsup – Zipf Normal Data (Unbinned)

One phase exponential decay
Best-fit values
   SPAN   1538
   K   0.4123

PLATEAU 0.08368
HalfLife   1.681
Std. Error
    SPAN   391.7
    K   0.02553
    PLATEAU 0.04217
95% Confidence Intervals
    SPAN   450.6 to 2625
    K   0.3414 to 0.4832
    PLATEAU -0.03340 to 0.2008
    HalfLife   1.435 to 2.030
Goodness of Fit
    Degrees of Freedom   4
    R²   0.9999
    Absolute Sum of Squares 0.03510
    Sy.x   0.09368

4.6.2.4 Binned Zipf Normal Data
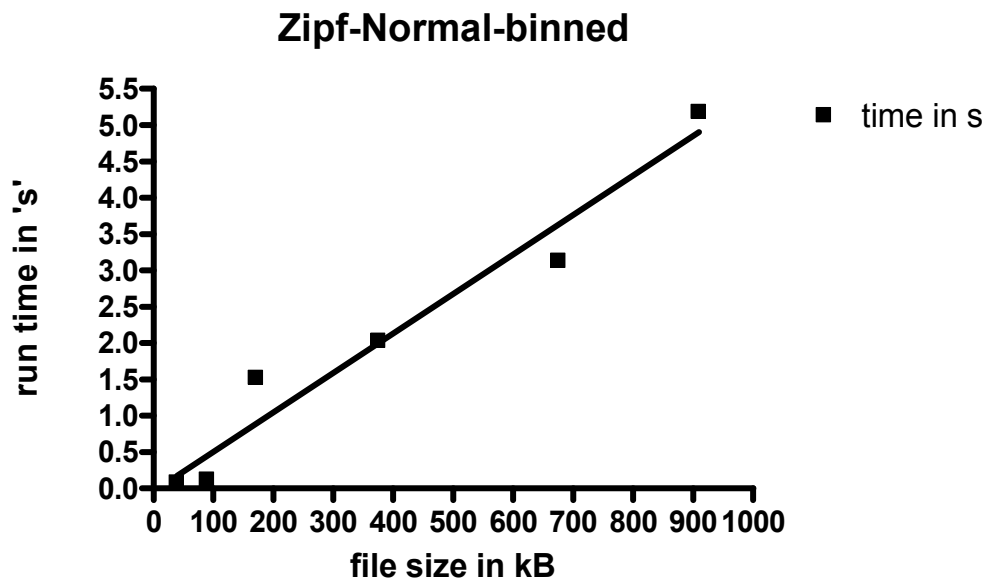
file size in kB	time in s	Closed freq.itemsets	minSup
39.287	0.075	1	10
89.863	0.118	1	20
171.176	1.519	1	20
375.213	2.03	1	30
675.256	3.13	1	80
909.882	5.177	1	200

Figure 25 Run Time Vs File Size – Zipf Normal Data (Binned)

Best-fit values

Slope   0.005426 ± 0.0005855
Y-intercept when X=0.0   -0.03618 ± 0.2892
X-intercept when Y=0.0   6.668
1/slope   184.3
95% Confidence Intervals
Slope   0.003801 to 0.007051
Y-intercept when X=0.0   -0.8389 to 0.7665
X-intercept when Y=0.0   -187.6 to 127.9
Goodness of Fit
  r²   0.9555
  Sy.x   0.4579

Zipf-Normal binned (file-size = 89.863 kB)
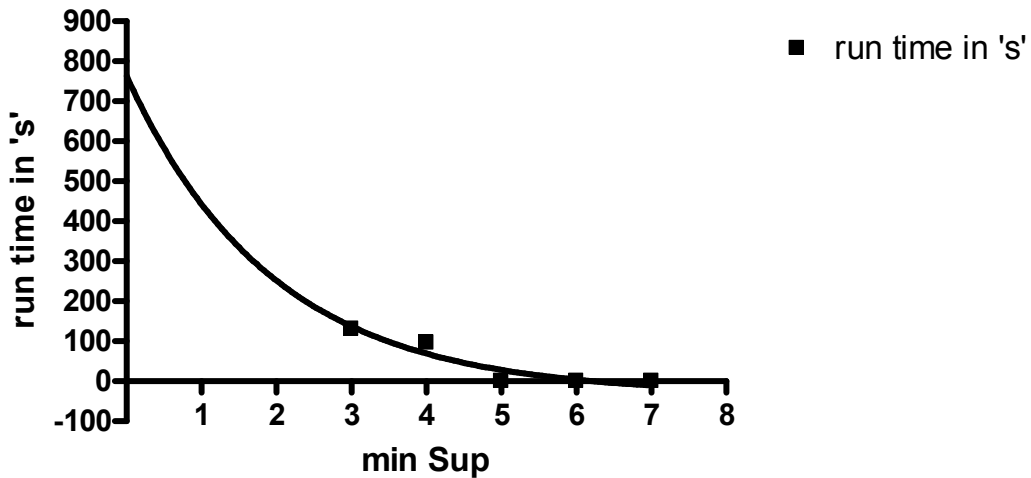
min Sup	run time in 's'
3	130.005
4	96.243
5	0.788
6	0.085
7	0.048

Figure 26 Run Time Vs Minsup – Zipf Normal Data (Binned)

One phase exponential decay
Best-fit values
  SPAN   796.1
  K   0.5156
  PLATEAU -32.34
  HalfLife   1.344
Std. Error
  SPAN   937.4

K   0.4900
PLATEAU 69.42
95% Confidence Intervals
    SPAN   -3238 to 4830
    K   0.0 to 2.624
    PLATEAU -331.0 to 266.4
    HalfLife
Goodness of Fit
    Degrees of Freedom   2
    R²   0.8942
    Absolute Sum of Squares   1677
    Sy.x   28.96

4.7 Discussion

The experiments with binning show significant improvements in the running time of the algorithm. For instance, without binning and with mixed data, the running time of the algorithm increases at a rapid rate (either polynomial or exponential); with binning, the running time is nearly constant (i.e., it is very similar to the results of a Zipf distributed data). Note that binning only provides an approximation to the number of closed frequent itemsets in the input data. Using the results of binning, further analysis maybe performed on a restricted set of the input data.

Without binning, exponential data has an exponential run time growth. With binning, the run time becomes polynomial. So, we increase our chances of arriving at the solutions of the lattice with binning. For mixed data, we found

that we were not able to get the program to complete in less than an hours time for unbinned data and hence had to terminate the run.

As pointed out earlier, it helps in determining trends in data. Since binning is lossy, based on the results further analysis may be performed.

5. Conclusion

In this thesis, we studied the problem of mining closed frequent itemsets in large data repositories. We used the algorithm of Lin[Lin2003] as the basis for our implementation. As part of the implementation, we identified several issues with the algorithm and proposed solutions for them. We then implemented the algorithm and used it to a performance study. Our results showed that for certain datasets (such as, dataset that is derived from an exponential distribution), the running time of the algorithm grows exponentially. To improve the running time of the algorithm, we developed a novel mechanism for binning data. Our binning procedure transforms data from exponential/normal distributions to Zipf distributed data. Our experiments with the binned data showed significant performance improvement: The running time of exponentially distribute data grows exponentially; in contrast, the running time of the binned data is nearly constant in the size of input.

Some possible future effort can build upon our work are:

- **Suggestion server**: For instance, consider the example we have used in this thesis related to buying books. We can mine the set of

transactions to identify the set of closed frequent itemsets

corresponding to authors whose books are frequently bought. This set

can be used as the basis for constructing a recommendation list.

Furthermore, whenever one of these authors writes a new book, that

book could be a candidate for inclusion in this recommendation list.

Other characteristics such as the quality of reviews can also be used as

candidate signals. Similar suggestions servers can be constructed for

other domains such as video rentals as well.

- **Performance comparison**: Compare the performance of the

  algorithm we implemented with others published in the literature such

  as Charm [Zaki2002], Closet[Pei2000], Mafia [Burdick2001],

  Pascal[Bastide2000].

6. References

[Ramakrishnan] R.Ramakrishnan and J.Gehkre. Fundamentals of Database Systems.McGraw-Hill, 2002

[Molina] H.Garcia-Molina, J.Ullman, and J.Widom.Database System Implementation.Prentice-Hall, 2000

[Agarwal1994] R.Agarwal, and R. Srikant. *Fast Algorithms for Mining Association Rules*. Proc. Intl. Conf. on Very Large Databases. pp1522-1534.

[Lin2003] T.Y.Lin, X.T.Hu, and E.Louie.*Using Attribute Value Lattice to Find Frequent Itemsets*. Data Mining and Knowledge Discovery: Theory,Tools and Technology. 2003.pp 28-36.

[Lin2002] T.Y.Lin, and Eric Louie. *Finding Asscoiation Rules by Granular Computing: Fast Algorithm for finding association rules*. Data Mining, Rough Sets and Granular Computing. 2002. pp 23 - 42.

[Zaki2002] M.J.Zaki, and C.Hsiao. CHARM: *An Efficient Algorithm for Closed Itemset Mining*. Siam International Conference on Data Mining. 2002. pp 457 - 473.

[Bastide2000] Y.Bastide, R.Taouil, N.Pasquier,G.Stumme, and L.Lakhal. *Mining frequent patterns with counting inference*. SIGKDD Explorations. 2000. pp 66 – 74.

[Press] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. Numerical Recipes in C. Cambridge University Press, 2002. pp 683 - 688.

[Brin1997] S.Brin, R.Motwani, J.Ullman, and S.Tsur. *Dynamic itemset counting and implication rules for market basket data*. ACM SIGMOD Conf. Management of Data. 1997. pp.255—264.

[Sarasere1995] A. Sarasere, E. Omiecinsky, and S. Navathe. *An efficient algorithm for mining association rules in large databases*. In Proc. 21st VLDB.1995. pp 432-443.

[Burdick2001] D.Burdick, M.Calimlim, and J.Gehrke. *MAFIA: A maximal frequent itemset algorithm for transactional databases*. In ICDE, 2001. pp 443—452.

[Pei2000] J. Pei, J. Han, and R. Mao. *CLOSET an efficient algorithm for mining frequent closed itemsets*. In Proceedings of the ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery DMKD'00, Dallas, USA, May 2000. pp21—30.

[Bayardo1998] R. J. Bayardo. Efficiently mining long patterns from databases. In ACM SIGFMOD Conf. Management of Data, June 1998.

[Wikipedia]Wikipedia http://en.wikipedia.org

[Survey]Association Rule Mining.
www.cs.unc.edu/Courses/comp290-90f04/lecturenotes/**associationrule**1.pdf

Graphpad Software