Semantic Compression and Pattern Extraction with Fascicles

H.V. Jagadish
U. Michigan, Ann Arbor
jag@cs.uiuc.edu

Jason Madar
U. British Columbia
jmadar@cs.ubc.ca

Raymond T. Ng
U. British Columbia
rng@cs.ubc.ca

Abstract

Often many records in a database share similar values for several attributes. If one is able to identify and group together records that share similar values for some – even if not all – attributes, one can both obtain a more parsimonious representation of the data, and gain useful insight into the data from a mining perspective.

In this paper, we introduce the notion of *fascicles*. A fascicle $F(k,t)$ is a subset of records that have k compact attributes. An attribute A of a collection F of records is *compact* if the width of the range of A-values (for numeric attributes) or the number of distinct A-values (for categorical attributes) of all the records in F does not exceed t. We introduce and study two problems related to fascicles. First, we consider how to find fascicles such that the total storage of the relation is minimized. Second, we study how best to extract fascicles whose sizes exceed a given minimum threshold (i.e., support) and that represent patterns of maximal quality, where quality is measured by the pair (k,t). We develop algorithms to attack both of the above problems. We show that these two problems are very hard to solve optimally. But we demonstrate empirically that good solutions can be obtained using our algorithms.

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

Name	Position	Points	Played Mins	Penalty Mins
Blake	defense	43	395	34
Borque	defense	77	430	22
Cullimore	defense	3	30	18
Gretzky	centre	130	458	26
Konstantinov	defense	10	560	120
May	winger	35	290	180
Odjick	winger	9	115	245
Tkachuk	centre	82	530	160
Wotton	defense	5	38	6

Figure 1: A Fragment of the NHL Players' Statistics Table

1 Introduction

Figure 1 shows part of a table of National Hockey League (NHL) players' statistics in 1996. For each player, his record describes the position he played, the number of points he scored, the number of minutes he was on the ice and in the penalty box. While there are players of almost every possible combination, there are numerous *subsets having very similar values for many attributes.* In other words, records in one of these subsets vary considerably only in the other attributes. For example, (i) Cullimore and Wotton belong to a group of defensemen who played, scored and penalized sparingly; and (ii) Borque, Gretzky and Tkachuk belong to another group who played and scored a lot. Identification of such subsets can be the basis for good compression schemes, and can be valuable for data mining purposes.

To do so, the crucial first step is the identification of what we call *fascicles*[1]. A k-dimensional, or k-attributed, *fascicle* $F(k,t)$ of a relation is a subset of records that have k *compact* attributes. An attribute A of a collection F of records is *compact* if the width of the range of A-values (for numeric attributes), or the number of distinct A-values (for categorical attributes) of all the records in F does not exceed t. We refer to t as the compactness tolerance. For instance, in Figure 1, suppose the compactness tolerance imposed on the attributes are: $t_{Position} = 1$ (i.e., exact match is required), $t_{Points} = 10$, $t_{PlayedMins} = 60$ and $t_{PenaltyMins} = 20$.

[1]According to Webster, a fascicle is 1. a small bundle; 2. one of the installments of a book published in parts.

Then Cullimore and Wotton are in a 4-D fascicle with `Position`, `Points`, `Played Mins` and `Penalty Mins` as the compact attributes. Different fascicles may have different numbers and sets of compact attributes.

Now given a fascicle, we can minimize its storage requirement by projecting out all the compact attributes and storing their representative values only once separately. More precisely, suppose we have N records in a relation with n attributes, each requiring b bytes of storage for a total of Nnb bytes. Suppose we are able to find c fascicles, each with k compact attributes projected out, such that they together cover all but N_0 of the records. Then the storage required for a fascicle with cardinality s is $kb + (n-k)bs$. Adding up over all c fascicles, we get $kcb + (n-k)(N-N_0)b$. Thus, in total, we save $(N - N_0 - c)kb$ bytes of storage. If we are able to get small values for N_0 and for c, at least compared to N, then the storage savings would be almost Nkb bytes. Thus, we consider the following problem:

[**Storage Minimization with Fascicles**] *Find fascicles such that the total storage is minimized.*

Note that existing compression techniques are "syntactic" in the sense that they operate at the byte-level. Compression with fascicles are *"semantic"* in nature in that the tolerance t takes into account the meanings and the dynamic ranges of the attributes. The two styles of compression are orthogonal and can work in tandem. In Section 4.2, we will show experimentally that:

- (**lossless compression mode**) First using fascicles only to re-order the tuples, and then applying normal syntactic compression on the re-ordered relation results in substantially greater compression than applying syntactic compression alone.

- (**lossy compression mode**) First performing lossy compression with fascicles by removing compact attributes, and then applying normal syntactic compression on the remaining attributes can reduce the final compressed size a few times more.

Furthermore, because fascicles are semantic in nature, the contents of the fascicles constitute patterns with legitimate semantic meanings. Provided that the size of a k-D fascicle is not small, the fascicle represents a significant sub-population that more or less agrees on the values of the k attributes. For instance, corresponding to a certain 3-D fascicle, amazingly there are about 25% of NHL players who had little impact on the game, i.e., their `Points`, `Played Mins` and `Penalty Mins` all very low (e.g., Cullimore and Wotton in Figure 1). Such observations are clearly of data mining value.

From a data mining perspective, the quality of a fascicle is measured by two components: t, the compactness tolerance, and k, the number of compact attributes. Given the same compactness tolerance, a fascicle with more compact attributes is of higher quality than another with fewer compact attributes. Similarly, given the same number of compact attributes, a fascicle with a smaller compactness tolerance is of higher quality than another with a larger tolerance. Two fascicles are incomparable if one has more compact attributes but the other has a smaller tolerance. This ordering of fascicles, formalized in Section 5.1, is denoted as $\geq_f$. Now given fascicles $F_1, \ldots, F_u$ all containing record R, we define:

$$Fas(R) \quad = \quad \{F_i \mid \not\exists 1 \leq j \leq u : F_j >_f F_i\} \qquad (1)$$

That is, we are only interested in the fascicles with the maximal qualities among $F_1, \ldots, F_u$. Finally, we maximize the set $Fas(R)$ for each record R. Since $Fas(R)$ is a set, maximization is conducted with respect to the well-known Smyth ordering of power sets[20], denoted as $\geq_s$, which will be detailed in Section 5.1. Thus, we have the following problem:

[**Pattern Extraction with Fascicles**] *Given a minimum size m, find fascicles of sizes $\geq m$, such that for all records R, $Fas(R)$ is maximized with respect to the ordering $\geq_s$.*

In this paper, we will develop algorithms to solve both the storage minimization and pattern extraction problems. These algorithms are strongly related to one another. We will show that these two problems are hard to solve optimally. However, we will demonstrate empirically that our algorithms give good solutions. We will also investigate the effect of various parameter choices.

The outline of the paper is as follows. In Section 2, we describe the basic model of fascicles. In Section 3, we develop two algorithms to solve the storage minimization problem. In Section 4, we present experimental results on semantic compression and storage minimization. In Section 5, to solve the pattern extraction problem, we show how to modify previous algorithms and present empirical results. In Section 6, we discuss related work.

2 Model and Assumptions

Altogether there are three parameters in the fascicle framework. First, there is the compactness tolerance t. Strictly speaking, t should be represented as $\vec{t}$ because, as shown in the earlier NHL example, each attribute should have its own compactness tolerance. We simplify our presentation here by using t instead of $\vec{t}$. Moreover, in practice, more elaborate definitions of compactness are possible. For instance, values of a categorical attribute can be organized in a generalization hierarchy and the compactness tolerance defined in terms of the depth of the hierarchy. For numeric attributes, one could specify compactness based on quantiles rather than actual values. The specific definition of compactness is not central to the algorithms developed here. For ease of discussion, we use the definition given in Section 1.

The second parameter in the fascicle framework is the minimum size parameter m. For storage minimization, the smaller the value of $m \geq 2$, the more fascicles could be found and the larger the potential for minimization. Thus, in principle, m is not a key parameter in the storage minimization problem, although in practice very small fascicles, with only a few records, do result only in small incremental savings in storage. For pattern extraction, however, m plays a more critical role. If the value of m is too small, this indicates that the pattern applies to too small a sub-population. In this setting, m plays a role akin to the support threshold in association rule mining.

Finally, there is the dimensionality parameter k, the number of compact attributes required of a fascicle. In both the storage minimization and pattern extraction problems, k plays a central role. Our algorithms to find fascicles treat k as the main independent variable.

3 Solving the Storage Minimization Problem

In this section, we study the storage minimization problem. We first develop algorithms to find candidate fascicles, and show how to minimize storage by selecting from amongst these candidates.

3.1 The Single-k Algorithm

Below we develop the Single-k algorithm, which finds k-D fascicles for a given value k. Later we show how to extend the Single-k algorithm to form the Multi-k algorithm, which finds fascicles with dimensionalities $\geq k$.

3.1.1 The Basic Approach: a Lattice-based Conceptualization

Consider a lattice consisting of all the possible subsets of records in the given relation. As usual, an edge exists between two subsets S, T if $S \supset T$ and $|S| = |T| + 1$. At the top of the lattice is a single point/set, denoted as $\top$, representing the entire relation. At the bottom of the lattice is a single point $\perp$ representing the empty set. Let n be the total number of attributes in the relation, and k $(0 \leq k \leq n)$ be the number of compact attributes desired in a fascicle. Consider any path $\langle \perp, S_1, S_2, \ldots, \top \rangle$ linking the bottom and top points in the lattice. Trivially, $\perp$ and set S_1, which consists of a single record, are n-D fascicles. By the definition of a fascicle, it is clear that if S is a superset of T, and if S is a j-D fascicle, then T must be an i-D fascicle for some $i \geq j$. This monotonicity property guarantees that for any given value $1 \leq k \leq n$, either $\top$ is itself a k-D fascicle[2], or there must exist two successive sets S_t, S_{t+1} on the path such that: (i) S_t is an i-D fascicle, and (ii) S_{t+1} is a j-D fascicle, with $j < k \leq i$. We call S_t a *tip* set.

[2]If this rather unlikely corner case applies, our problem is trivially solved. To keep our exposition clean, we assume that this corner case does not apply in the sequel. That is, $\top$ is a j-D fascicle for some $j < k$. For all practical purposes, $\top$, consisting of the entire relation, is almost never a j-D fascicle for any $j > 0$, and k is always chosen to be at least one.

To put it in another way, S is a (k-D) tip set if S is a k-D fascicle, and there is an (immediate) parent T of S in the lattice such that T is a j-D fascicle with $j < k$. Within the class of tip sets, there is a subclass that we call *maximal* sets. S is a (k-D) maximal set if S is a k-D fascicle, and for *all* parents T of S in the lattice, T is a j-D fascicle with $j < k$.

Our goal is to find maximal sets. Forming the border analyzed in [16], the set of maximal sets completely characterizes all k-D fascicles. However, given the *huge* size of the lattice space, the entire set of maximal sets is so large that its computational cost is prohibitive. Furthermore, as we will see, we cannot in any case afford to select optimally from a given set of candidate fascicles for the problems we seek to address. It may suffice to compute some, but not all, maximal sets. Below we describe how the Single-k algorithm selects "good" initial tip sets, and eventually grows them to maximal sets.

3.1.2 Choosing Good Initial Tip Sets

Conceptually, to find a tip set, we begin by picking a random record as the first member of a tip set. We then add in a second record at random, and then a third, and so on – until the collection of records is no longer a k-D fascicle. In essence, we are constructing a path $\langle \perp, S_1, S_2, \ldots, S_t \rangle$. We can repeat this process as many times as we wish, each time exploring a different path up the lattice.

Given a relation that does not fit in main memory, each record sampled would require one disk access. To minimize the amount of I/O activity performed, a commonly used technique is "block sampling", where an entire disk page is read into memory and all records on the page are used. We propose to adopt this approach as well, but there is an important difference. In traditional sampling, possible correlations between co-located records can be compensated for by increasing the sample size. In our case, the specific ordering of records in the sample is of critical importance. Increasing the sample size would be of no help at all. For example, suppose that the hockey players relation is stored on disk sorted by `Played Mins`. When we consider successive records, they will very likely have `Played Mins` as a compact attribute, to the potential detriment of other possible compact attributes.

To address these concerns, we read into memory some number b of randomly sampled blocks of the relation. Now we can work purely with the sample of the relation in memory, without paying any attention to block boundaries. From amongst the records in memory, we can choose records based on a random permutation of the records (without physically sorting them). When one tip set is complete, the first record that would render it non-compact is used to start a second tip set, and so on. Once all records in the sampled relation have been considered once, a new random permutation of the records is established, and more tip sets are generated.

```
Algorithm Single-k
Input: A dimensionality $k$, number of fascicles $P$, a buffer of $b$ pages, and a relation $R$ of $r$ pages
Output: $P$ $k$-D fascicles

{ 1. Divide $R$ into $q$ disjoint pieces, each comprising upto $b$ randomly chosen pages from $R$, i.e., $q = \lceil r/b \rceil$.
  2. For each piece: /* choosing initial tip sets */
     2.1 Read the piece into main memory.
     2.2 Read the records in main memory to produce a series of tip sets as discussed in Section 3.1.2.
     2.3 Repeat 2.2, each with a different permutation of the records, until $P/q$ tip sets are obtained.
  3. /* growing the tip sets */
     Grow all $P$ tip sets, as discussed in Sections 3.1.3, with one pass over the relation. Output the grown tip sets. }
```

Figure 2: A Skeleton of the Single-k Algorithm

This process is repeated as many times as desired. All operations are in memory and that no (physical) sorting is required.

Because of the oversampling we perform, it is possible not to work with just a single sample of the relation. Instead, we can produce some tip sets working with the sampled relation described above; when this is done, we can obtain a fresh sample of b completely different pages from the relation on disk, and repeat. Proceeding thus, we can consider the entire relation as q disjoint pieces, where each piece is a random collection of pages.

3.1.3 Growing a Tip Set to a Maximal Set

Tip sets obtained thus far leave two aspects to be desired. First, they are confined to records in one piece of the relation. We seek to "grow" a tip set so that it includes all qualified records from the remaining pieces of the relation. Second, with respect to the conceptual lattice discussed earlier, a tip set may be far from maximal. We seek to make a tip set maximal.

This turns out to be quite easy to do. A tip set obtained thus far has k compact attributes, corresponding to k attribute value ranges (for numeric attributes) or sets of values (for categorical attributes). These in effect specify a k-D selection predicate on the relation. And we can simply evaluate a query that returns all records matching the query, such as by scanning the entire relation one more pass. The newly grown tip set is still a k-D fascicle. We refer to this process as the tip set growing phase.

With respect to maximality, the simple scan above is sufficient for most circumstances. Specifically, if before the growing phase the k compact attribute ranges and values of a tip set are already at the maximum "width" allowed by t, then it is easy to see that after the growing phase, the newly grown tip set has indeed become a maximal set, i.e., it is impossible to add any more record to keep it a k-D fascicle. In the less common circumstances, where before the growing phase there is some compact attribute that still has room to reach the maximum allowable width, maximality can be achieved by expanding the compact range or set dynamically until the maximum width is reached. For lack of space, we omit the details here; see [15] for details and an argument why maximality is guaranteed.

In summary, Figure 2 shows a skeleton of the Single-k algorithm. It is a randomized algorithm that computes k-D fascicles, all of which are maximal sets. Note that the algorithm does not compute all k-D fascicles. In Section 4.6, we will present an alternative algorithm that computes all k-D fascicles, but will show that such an alternative is both ineffective and impractical.

3.2 The Multi-k Algorithm

Recall that for both the storage minimization problem and the pattern extraction problem, we may allow fascicles to vary in their dimensionalities. The Single-k algorithm, however, only produces fascicles with one specified value k. The problem here is that the algorithm may miss out on the opportunities offered by the fascicles with higher values of k. For instance, there could be a fascicle of dimensionality 10 that is a subset of a fascicle (of dimensionality 6) found by the algorithm. The higher dimensionality fascicle, if known and exploited, can clearly lead to additional reduction in storage cost.

One simple way to overcome the above problem is to run the Single-k algorithm repeatedly, each with a different value of k. But different runs of the algorithm share many repetitive operations, including the relation scans. The Multi-k algorithm exploits this sharing to produce fascicles all having dimensionalities $\geq k$.

Recall from the Single-k algorithm how a k-D tip set corresponds to a path $\langle \perp, S_1, S_2, \ldots, S_t \rangle$. Observe that as we move from the beginning of the path to the end of the path, the dimensionality of the corresponding fascicle decreases monotonically. Thus, while the Single-k algorithm constructs a path $\langle \perp, S_1, S_2, \ldots, S_t \rangle$ and obtains S_t as a k-D tip set, the Multi-k algorithm uses exactly the same path to obtain the largest set on the path with dimensionality i, for $i \geq k$.

Let us illustrate this by an example. Suppose there are a total of $n = 12$ attributes in the relation. Suppose $k = 6$, and the path corresponding to a particular 6-D tip set computed by the Single-k algorithm is $\langle \perp, S_1, S_2, \ldots, S_{10} \rangle$. The following table shows the dimensionality of each set on the path:

Set	$\perp$	S_1	S_2	S_3	S_4	S_5	S_6
k	12	12	12	11	11	11	9

Set	S_7	S_8	S_9	S_{10}
k	9	8	6	6

The Multi-k algorithm uses the same path to obtain the tip sets S_2, S_5, S_7, S_8 and S_{10} with dimensionalities 12, 11, 9, 8 and 6 respectively.

To complete the description of the Multi-k algorithm, these tip sets with varying dimensionalities can all go through the same growing phase together to become maximal sets, sharing one relation scan. Thus, Figure 2 presents an accurate skeleton of the Multi-k algorithm, provided that Step 2.2 is modified to implement the procedure discussed in the previous paragraph.

No matter what the available amount of buffer space is, both Single-k and Multi-k read each data page from disk at most twice (i.e., once for Step 2 and once for Step 3). Furthermore Step 3 is a sequential scan. Step 2 requires random pages to be read, but since the specific set of random pages is not material, it is conceivable that clever techniques could choose "random" pages while being aware of disk layout to optimize disk access. This assumes that P is small enough to fit in memory, which is easily achievable as will be shown by the results in Section 4. For a relation R, the main memory computational complexity of both algorithms is $O(P|R|)$.

3.3 Greedy Selection for the Single-k Algorithm

To solve the storage minimization problem, the next step is to select from amongst the candidate fascicles produced by either the Single-k or the Multi-k algorithms. The main complication is that the candidate fascicles may overlap. To proceed, let us consider briefly the "unweighted" version of the above task: find the minimum *number* of candidate fascicles that cover the whole data set. This turns out to correspond to the well-known minimum cover problem [12]. That is, given a collection C of subsets of a finite set S and a positive integer K, is there a subset $C' \subseteq C$ with $|C'| \leq K$ such that every element of S belongs to at least one member of C'? The minimum cover problem is NP-complete. Greedy selection is among the best heuristics that exist for solving the minimum cover problem, and is the basis for our selection algorithms. We discuss first greedy selection for the Single-k algorithm.

To represent the storage savings induced by a fascicle F, it is weighted by $wt(F) = k * |F|$, where k is the dimensionality of F.[3] This weight corresponds to the storage savings induced by F, since there are k fewer attributes to store for each record contained in F. In a straightforward implementation of the greedy selection, we select the candidate fascicle with the highest weight, subtract from all the remaining fascicles records that are in the selected fascicle, and adjust the weights of the remaining fascicles accordingly. Specifically, if A is the selected fascicle, then the adjusted weight of each remaining fascicle F is given by:

$$wt(F/A) \quad = \quad k * |F - A| \qquad (2)$$

Then from among the remaining fascicles, we pick the one with the highest adjusted weight, and repeat.

In [15], we give a more optimized implementation of the above greedy selection. The basic idea is not to do weight re-adjustment and re-sorting of the remaining fascicles after each fascicle has been selected. Instead, we pick a "batch" of fascicles $F_1, \ldots, F_u$ so that none of them overlap with each other. At the end of each batch, re-sorting and weight re-adjustment is done only once based on $\cup_{i=1}^{u} F_i$. This saves considerable overhead.

3.4 Greedy Selection for the Multi-k Algorithm

The weight re-adjustment formula is more complicated for the Multi-k algorithm, because the candidate fascicles may have varying dimensionalities. More importantly, as described in Section 3.2, the Multi-k algorithm can generate many pairs of candidates F_1, F_2 such that: (i) $F_1 \subset F_2$, but (ii) $k_1 > k_2$, where k_1 and k_2 are the dimensionalities of the fascicles. If F_2 is selected first, then according to Equation (2), $wt(F_1/F_2)$ becomes 0. However, it is easy to see that even after F_2 is selected, F_1 can still be chosen to provide further storage savings, albeit only to the records contained in F_1. Consider the following example, where $|F_1| = 50$, $k_1 = 8$, $|F_2| = 100$ and $k_2 = 6$. F_2 is selected because its storage savings is 600, as compared with 400 from F_1. However, for the 50 records contained in both F_1 and F_2, they can indeed be compressed further to 8 compact dimensions using F_1. This means an additional savings of $50 * (8 - 6) = 100$ is possible. To reflect this additional savings possibility, we re-adjust the weight of a fascicle in a way more general than in Equation (2):

$$wt(F/A) \quad = \quad \begin{cases} k_F * |F - A| & \text{if } k_F \leq k_A \\ k_F * |F - A| + \\ \quad (k_F - k_A) * |F \cap A| & \text{otherwise} \end{cases} \qquad (3)$$

If the selected fascicle A is of the same or higher dimensionality, then the weight of F is re-adjusted as usual. However, if A is of a lower dimensionality, then the weight of F includes the component $(k_F - k_A) * |F \cap A|$, corresponding to the additional storage reduction of the records in both A and F.

4 Experimental Evaluation for Semantic Compression

4.1 Data Sets used and Experimental Setup

Since we seek to exploit hidden patterns in the data, it is hard to perform any meaningful experiments with synthetic data. We used two very different real data sets for all our experiments:

[3] The value k is the same for all fascicles produced by the Single-k algorithm. As such the multiplication by k can be dropped. However, for easier comparison with Equation (3) later, we show the multiplication explicitly.

	syntactic only	fascicle re-ordering + syntactic	fascicle lossy + syntactic
ascii (gzip)	0.45	0.32	0.14
binary (compress)	0.68	0.51	0.20

Figure 3: Relative Storage: Semantic Compression in Tandem with Syntactic Compression

- **AT&T Data Set**: This data set has 13 attributes describing the behavior of AT&T customers, one per record. Some attributes are categorical (e.g., calling plan subscribed), and others are numeric (e.g., minutes used per month). This is a large data set, of which 500,000 records were extracted randomly and used for the bulk of the experiments.
- **NHL Data Set**: We have already seen a sample of the NHL data set earlier in the paper. For each of about 850 players, there is one record with 12 categorical and numeric attributes describing the position the player played, the minutes played, etc. Among numerous other sites, NHL players statistics can be found in http://nhlstatistics.hypermart.net.

All experiments described in this section were run with both data sets, and produced very similar results. For lack of space, we only show in this section the results from the larger AT&T data set. All experiments were run in a time-sharing environment provided by a 225MHz ultrasparc workstation.

As shown below, the behavior of the algorithms may depend on the choices of values of several parameters: the dimensionality k, the compactness tolerance t, the number P of tip sets, the minimum size m of a fascicle, and the data set size. Unless otherwise stated, the default values are: data set containing 500,000 records, $P = 500$, $m = 8$, [4] and $t = 1/32$, which means that for a numeric attribute A, t is set to be 1/32 the width of the range of A-values in the data set, and that for a categorical attribute A, t is set to be $\lceil w/32 \rceil$, where w is the total number of distinct A-values in the data set. For most of the categorical attributes in our data sets, w is indeed ≤ 32, which effectively means that we are applying *lossless* compression to categorical attributes with this t value.

Results of interest are runtime, relative storage, and coverage. Runtimes are given in seconds of total time (CPU + I/O). *Relative storage* is defined to be the ratio of the size of the compressed data set to the original size. *Coverage* is defined to be the percentage of records that are contained in some fascicle after the greedy selection.

4.2 The Bottom Line: Semantic Compression with Syntactic Compression

In this experiment we show how storage minimization with fascicles can work in tandem with normal syntactic compression algorithms. We used the unix gzip and compress to compress the ascii and binary versions of the data sets respectively. Figure 3 shows the relative storage figures when (i) fascicles are used only to re-order tuples for subsequent syntactic compression; and (ii) fascicles are used to provide lossy compression on compact attributes, followed by syntactic compression on the remaining attributes.

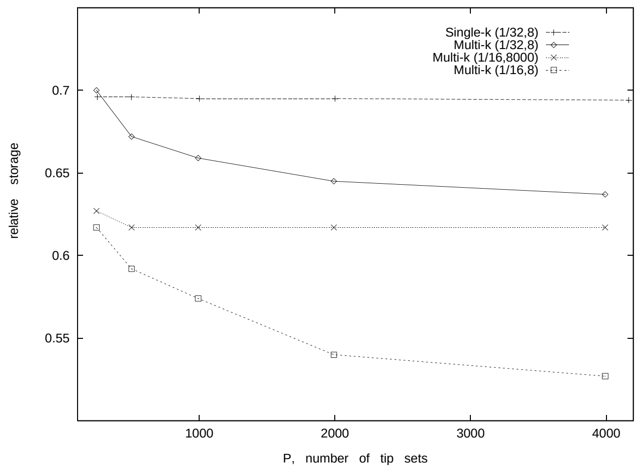
For both the binary and ascii versions, Figure 3 shows that even if we are to use fascicles simply to re-order tuples (i.e., lossless compression here), the additional savings is considerable (e.g., an additional 30% $\approx$ (0.45-0.32)/0.45). And when we allow acceptable lossiness (as governed by $t = 1/32$ here) in the compact attributes, the additional savings is substantial (e.g., 0.45 versus 0.14, a factor of 3).

4.3 The Effect of P, the Number of Tip Sets

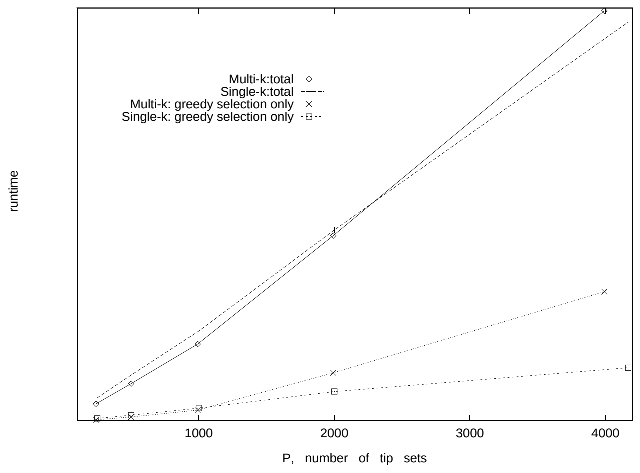
Next we study the effect of the various parameters on the Single-k and Multi-k algorithms. For all the results reported below, we focus on the amount of storage savings produced by fascicles alone. We begin with the internal parameter P, the number of tip sets generated. Figure 4 shows how P affects the relative storage and runtime of the two algorithms. We ran with: $k = 4$, P from 250 to 4000, m from 8 to 8000, and t one of 1/16, 1/32 and 1/48. In terms of storage requirements, we make the following observations:

- Even for as few as $P = 250$ tip sets, for a data set of 500,000 records, both algorithms offer a storage savings of over 30%. This indicates that both algorithms are effective in generating (at least some) good quality candidate fascicles.
- The Single-k algorithm is relatively insensitive to P and gives a relative storage around 0.7 for all values of P with $t = 1/32$ and $m = 8$. With other combinations of t and m, the relative storage changes in a way to be described in Section 4.5. But in all cases, we still get a very flat curve, which is not included in Figure 4(a).
- In contrast, the Multi-k algorithm can be more sensitive to P. For example, with $t = 1/32$ or $1/16$ and $m = 8$, Multi-k becomes more effective in storage reduction with increasing P values. (Here we focus on the shape of the curve; the absolute position of the curve regarding changes in t and m will be discussed in Section 4.5.) This is the case because for the Multi-k algorithm, P corresponds to

[4]Strictly speaking, for storage minimization, the minimum size of a fascicle can be as low as 2. However, to allow some pruning power for the FAP algorithm to be shown in Section 4.6, we arbitrarily set m to a small integer.



(a) Storage Requirement



(b) Runtime

Figure 4: The Effect of P

the total number of j-D tip sets for all $j \geq k$. For example, among a total of 4,000 tip sets found by Multi-k, there may only be around 500 4-D tip sets (there are 13 attributes). Thus, more tip sets are required for the Multi-k algorithm to stablize in relative storage. However, with $m = 8000$, a situation where there are not many tip sets to begin with, even Multi-k becomes insensitive.

In terms of runtime, Figure 4(b) shows the total time and the fraction of the total time spent on greedy selection for both algorithms with $t = 1/32$ and $m = 8$. (The runtime trends for other combinations are the same.) The total time increases linearly with P, and there is little difference between the two algorithms. But Multi-k takes a lot longer in greedy selection than Single-k does. This is the overhead in conducting the more sophisticated weight re-adjustment shown in Section 3.4.

For the remaining experimentation, because the runtimes of both algorithms increase linearly with P, and the improvement of storage from using a larger P value tails off quickly, we consider $P = 500$ as a reasonable choice for our data sets. And all comparisons of the two algorithms are based on the Multi-k algorithm obtaining exactly the same number P of tip sets as the Single-k algorithm. In previous paragraphs, we have explained how Multi-k would compare if a larger value of P were used. We note in passing that the buffer size used, b, has little effect on the above results, so long as it is not too small (say, a few percent of the data set size).

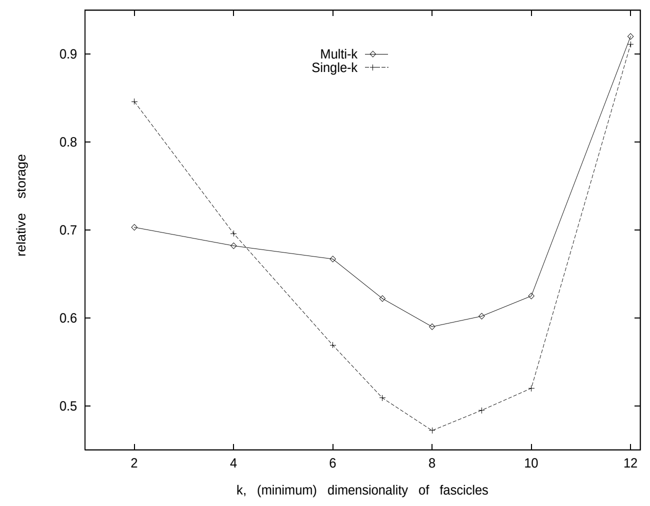
4.4 The Effect of k, the (Minimum) Dimensionality of Fascicles

Figure 5(a) shows the relative storage provided by both algorithms as k varies from 2 to 12. For the Single-k algorithm, we observe that the storage savings is maximized for an intermediate value of k. When k is small, only a small amount of savings in storage is achieved through compaction – precisely because k is small. As k increases, there are two opposing forces: (i) k itself favors further storage reduction, but (ii) coverage decreases, thereby decreasing the number of records that can be compressed. Figure 5(b) shows the coverage figures as k varies. Initially the k factor dominates the coverage factor, giving a minimum relative storage below 0.5. But as k increases past the optimal value, in this case $k_{opt} = 8$, the drop in coverage dominates.

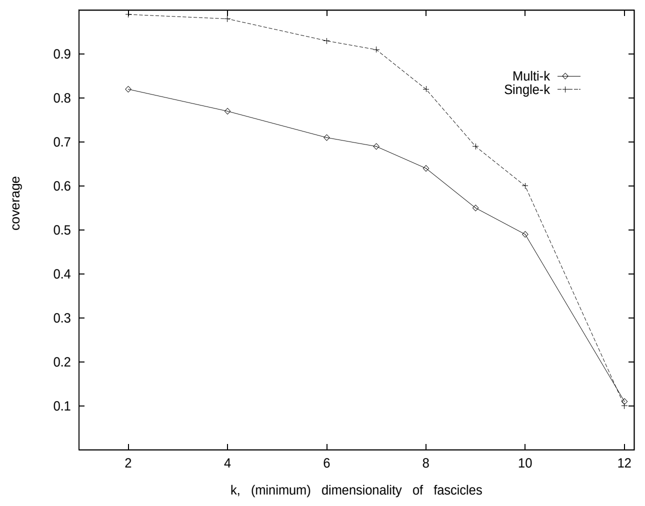
For the Multi-k algorithm, there is also an optimal value of k. This is the case because as explained in Section 3.4, the greedy selection procedure tends to select first fascicles of the minimum dimensionality. So in this sense, it behaves like the Single-k algorithm. But it differs from the Single-k algorithm in the following ways:

- It is less sensitive to k. This is because even if the input k value is smaller than the actual optimal value, Multi-k algorithm is able to produce candidate fascicles of dimensionality $\geq k$.
- When k is small, observe from Figure 5(a) and (b) that despite having a lower coverage, Multi-k provides better compression than Single-k. This is again due to the former's ability to produce candidate fascicles of dimensionality $\geq k$.
- For larger values of k, however, this ability of Multi-k becomes less important because there are fewer fascicles of dimensionality $\geq k$. At that point, coverage becomes more correlated with relative storage, and Multi-k lags behind Single-k in effectiveness. But this is largely a consequence of our forcing both algorithms to generate the same number P of tip sets. The effectiveness of Multi-k can be improved by running with a larger value of P – at the expense of a larger runtime.

As expected, the runtimes of the algorithms were independent of k (results not included here).



(a) Storage Requirement



(b) Coverage

Figure 5: Optimal k for Storage Minimization

4.5 Miscellaneous Effects and Practitioners' Guide

As t, the compactness threshold, increases, more records qualify to participate in a fascicle, and so there are more opportunities for storage reduction. The k_{opt} value that minimizes storage changes slightly as t varies; for both algorithms, as t increases, larger values of k are desirable. In other words, as we become more forgiving in our lossy compression, it is fruitful and possible to compact more dimensions. As for m, the minimum fascicle size, fewer fascicles can assist in storage minimization when m increases. And the optimal k_{opt} value decreases as m increases. Curves are not presented for lack of space.

Figure 6 shows how the two algorithms scale up with respect to increasing data set size. Both algorithms scale up linearly. While the results shown in the figure are based on $k = 4$, the results generalize to other values of k. The reason why Multi-k runs faster than Single-k is strictly a consequence of our forcing both algorithms to produce the same number of tip sets ($P = 500$ as usual).

In sum, we have shown empirically how the two algorithms behave when the various parameters change, among which P is the most critical. In general, increasing values of P result in both greater computational cost and better compression. However, even fairly small values of P give fairly good results. As a "practitioners' guide", we recommend starting with $P = .001$ of the database size. If one can afford more computational resources, one can pick a larger P. Finally, we recommend the Multi-k algorithm for small values of k, but recommend the Single-k algorithm otherwise.

4.6 Comparison with The FAP Algorithm for Finding All k-D Fascicles

Both the Single-k and Multi-k algorithms selectively find P candidate fascicles for storage minimization. In this section we consider a non-randomized alternative that computes all fascicles.

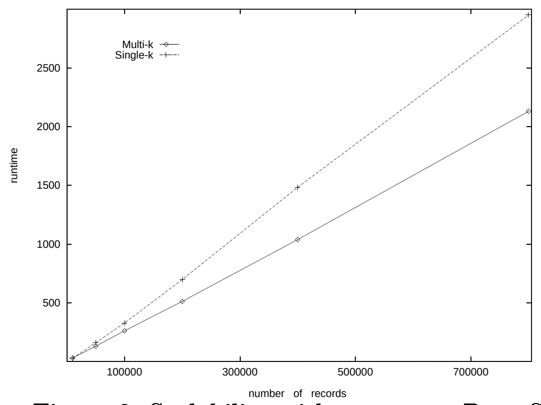


Figure 6: Scalability with respect to Data Set Size

Specifically, we consider an algorithm called *FAP* or *Fascicles through APriori*. As the name suggests, FAP is an adaptation of the well-known Apriori algorithm for association rule mining [2, 3, 13]. Apriori basically performs a bottom-up, level-by-level computation of the underlying lattice space. In the case of FAP for computing all k-D fascicles, the underlying lattice space consists of all possible subsets of attributes. For a given subset of attributes $\{A_1, \ldots, A_u\}$, its "support" is the number of records that form a fascicle with $A_1, \ldots, A_u$ as the compact attributes. This simplified view applies perfectly to categorical attributes. But for numeric attributes, a pre-processing step is necessary to divide the attribute range into bins of width equal to the specified tolerance t. We call this step *pre-binning*. Once pre-binning has been performed, the FAP algorithm computes 1-D fascicles whose support exceed the minimum size parameter m. Then it proceeds to compute 2-D fascicles and so on. The usual pruning strategy applies because the set of records supporting the fascicle monotonically decreases

k	Single-k			FAP		
	Runtime	Relative Storage	Coverage	Runtime	Relative Storage	Coverage
2	3.5	0.846	0.999	13.5	0.848	0.992
3	3.5	0.772	0.988	299.8	0.780	0.967
4	3.4	0.700	0.974	1591.0	0.750	0.846
5	3.4	0.640	0.941	> 2500.0	n/a	n/a

Figure 7: Inappropriateness to Compute All Fascicles for Storage Minimization

in size as the set of compact attributes grows.

Figure 7 compares the FAP algorithm with the Single-k algorithm for computing k-D fascicles, for various values of k. To ensure that FAP terminates in a reasonable amount of time, we used a data set of 1,000 records extracted at random from the AT&T data set.

The difference in runtime performance between Single-k and FAP is truly astonishing. Even for 1,000 records, the time required for FAP for $k \geq 5$ is so large that we did not find it possible to continue running the process on our computer. For $k \leq 4$, Single-k dominates FAP by orders of magnitude. There are two explanations. First, for a given value of k, FAP computes all j-D fascicles for all $1 \leq j \leq k$. This bottom-up strategy is not practical except for very small values of k. Second, particularly for low values of the parameter m, virtually no pruning occurs in the first few passes, resulting in a exponential growth in the number of intermediate candidates. Specifically, if there are u 1-D fascicles, there could be $O(u^2)$ 2-D fascicles, $O(u^3)$ 3-D fascicles, so on, at least for the first several passes.

The only remaining question is whether FAP gives better storage and coverage results, possibly justifying the additional computational cost. Surprisingly, Figure 7 shows that FAP actually did *worse* than Single-k. This is due to the pre-binning step conducted in a fascicle-independent and on a per-attribute basis. In contrast, Single-k (and Multi-k) forms the compact ranges and sets dynamically based on all the attribute values of the current set of tuples under consideration.

5 Solving the Pattern Extraction Problem

Pattern extraction is not as well-defined a problem as storage minimization. There are many different types of patterns one may wish to extract from a data set, and many different possible metrics to assess the quality of the extracted patterns. So we begin this section by defining our pattern extraction task in precise terms. Then we develop an algorithm to carry out the task, which is an extension to the Multi-k algorithm. Finally, we show some empirical results.

5.1 The Pattern Extraction Problem with Fascicles

By definition, a k-D fascicle contains records that more or less agree on the values of the k compact attributes. Given a minimum size m, like the minimum support in association rules, we say that a fascicle is *frequent* if it contains at least m records. Given two (frequent) fascicles F_1, F_2 with dimensionality k_1, k_2 and compactness tolerance t_1, t_2 respectively, we say that:

$$F_1(k_1, t_1) \geq_f F_2(k_2, t_2) \quad \text{iff} \quad k_1 \geq k_2 \text{ and } t_1 \leq t_2 \quad (4)$$

The intuition is that for any fascicle $F_1(k_1, t_1)$, F_1 is automatically a fascicle of the quality pair (k_2, t_2) where $k_2 \leq k_1$ and $t_2 \geq t_1$. For any record R contained in both F_1, F_2 with $F_1 >_f F_2$, we prefer F_1.

For the pattern extraction problem, there is a pre-defined range of dimensionalities $[k_{min}, k_{max}]$ and a series of compactness tolerance $t_1 < \ldots < t_u$. Given these, the ordering in Equation (4) defines a lattice. Figure 8 gives an instance of the quality lattice for pattern extraction from the NHL data set. In this instance, the dimensionality k varies from some minimum (not shown in the figure) to 12, and the compactness tolerance can be 1/8, 1/16 or 1/32 (cf: Section 4.1). In the figure, a directed edge indicates the partial ordering relationship, e.g., (12,1/32) $>_f$ (12,1/16).

The ordering $\geq_f$ defined on fascicles can be extended to give an ordering $\geq_s$ on *sets of fascicles*. To do so, we rely on the well-known Smyth ordering for power sets [20]:

$$\mathcal{S}_1 \geq_s \mathcal{S}_2 \quad \text{iff} \quad \forall F_2 \in \mathcal{S}_2, \exists F_1 \in \mathcal{S}_1 : F_1 \geq_f F_2 \quad (5)$$

For the example in Figure 8, the sets $\mathcal{S}_1 = \{(12, 1/8), (10, 1/16)\}$ and $\mathcal{S}_2 = \{(11, 1/32)\}$ are mutually incomparable. But the set $\mathcal{S}_3 = \{(12, 1/32)\}$ dominates both, i.e., $\mathcal{S}_3 >_s \mathcal{S}_1$, $\mathcal{S}_3 >_s \mathcal{S}_2$.

The Smyth ordering induces equivalence classes of fascicles. Specifically, it is possible to have two sets $\mathcal{S}_1 \geq_s \mathcal{S}_2$ and $\mathcal{S}_2 \geq_s \mathcal{S}_1$ without having $\mathcal{S}_1 = \mathcal{S}_2$. An example are the sets $\{(12, 1/32)\}$ and $\{(12, 1/32), (11, 1/32)\}$. In other words, the equivalence classes induced by $\geq_s$ are not necessarily minimal in the set inclusion sense. This is particularly relevant to our task at hand, because a record R may be contained in many fascicles. To ensure that the final generated fascicles and patterns are of the highest quality, we define $Fas(R)$ given in Equation (1) to ensure that among the set of fascicles all containing R, only the maximal ones with respect to $\geq_f$ can keep R. Hence, putting all the pieces together, the pattern extraction problem we are tackling here is one that given a pre-defined range of dimensionalities $[k_{min}, k_{max}]$ and

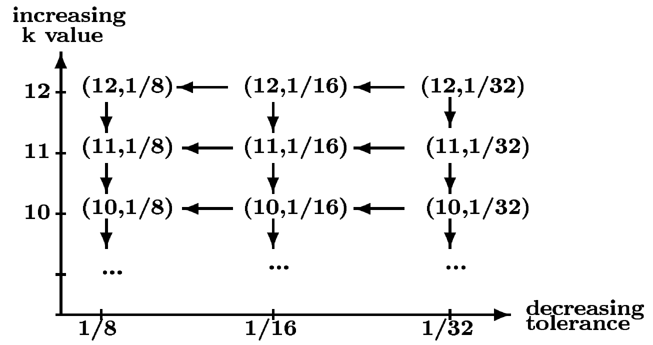


Figure 8: An Example: Pattern Extraction with Fascicles for the NHL data set

a series of compactness tolerance $t_1 < \ldots < t_u$, and a minimum size m, tries to find fascicles of sizes $\geq m$, such that for all records R, $Fas(R)$ is maximized with respect to the ordering $\geq_s$.

There are two important details to note. First, a record R that is contained in both F_1 and F_2 with $F_1 >_f F_2$, does not contribute to the size of F_2. In other words, for F_2 to be frequent, there must be at least m records for which F_2 is a maximal fascicle. Second, one may wonder why bother to maximize $Fas(R)$ for each record R, rather than simply find the fascicles with the maximal qualities. Under this second approach, if we have fascicles of qualities (11,1/16), (10,1/32) and (9,1/32), those of qualities either (11,1/16) or (10,1/32) would be output, but those of quality (9,1/32) would be discarded. Our definition of the problem is strictly more informative than this second approach in that while fascicles of qualities either (11,1/16) or (10,1/32) are output, fascicles F of quality (9,1/32) are also output – provided that there are at least m records for which F is maximal.

5.2 The Multi-Extract Algorithm

5.2.1 Computing Fascicles for all (k, t) Pairs

Like the storage minimization problem, there are two steps in solving the pattern extraction problem. The first step is to find the fascicles. We could simply run the Multi-k algorithm with different values of t. However, it is easy to adapt the Multi-k algorithm to do better. Recall that there is an initial tip set generation phase, followed by a phase to grow tip sets into maximal sets. Iterating the Multi-k algorithm for multiple values of t amounts to running both the generation and the growing phase multiple times. But we can be more efficient by running the generation phase multiple times, but only once for the growing phase. Specifically, we iterate the generation phase (over each piece of the relation read into memory) to produce tip sets for different values of t. This does not require any additional disk I/O. Then we grow all tip sets of varying dimensionalities and compactness tolerance simultaneously in one

```
Algorithm Multi-Extract
Input: relation $R$, minimum dimensionality $k$, minimum size
       $m$, a series of tolerance $t_1 < t_2 < \ldots < t_u$, integer $P$
Output: fascicles $F$ with dimensionality $\geq k$ and containing
        at least $m$ records for which $F$ is maximal

{ 1. ... /* basically the same as in the Single-k algorithm
  2. ...    except for the optimization discussed in
  3. ...    Section 5.2.1 */
  4. /* maximize Fas(R) */
     For all the tip sets $F$ obtained above, process in
     ascending rank:
     4.1 $F_{rem} = F - (\bigcup_{G>F} G)$, where the $G$'s were frequent
         tip sets obtained in previous iterations of this loop.
     4.2 If $|F_{rem}| \geq m$, output $F_{rem}$.
}
```

Figure 9: A Skeleton of the Multi-Extract Algorithm

final scan of the relation. The I/O and main memory complexity remain the same as before.

5.2.2 Maximizing $Fas(R)$

Once candidate fascicles have been generated, the second step for pattern extraction is quite different from that for storage minimization, because for pattern extraction it is fine to have a record put in multiple (frequent) maximal fascicles. To achieve this, we define the *rank* of a fascicle, with respect to the partial ordering given in Equation (4). Given the range of dimensionality $[k_{min}, k_{max}]$ and a series of compactness tolerance $t_1 < \ldots < t_u$, we have the following inductive definition of *rank*:

- (base case) $rank(F(k_{max}, t_1)) = 1$; and
- (inductive case) $rank(F(k, t_i)) = 1 + min\{ rank(F(k+1, t_i)), rank(F(k, t_{i-1}))\}$

For the example given in Figure 8, the series of compactness tolerances is 1/32, 1/16 and 1/8, and the largest dimensionality is 12. Accordingly, all fascicles $F(12, 1/32)$ are assigned the top rank, i.e., $rank = 1$. By the inductive case of the definition, all fascicles $F(11, 1/32)$ and $F(12, 1/16)$ have $rank = 2$. Similarly, all fascicles $F(12, 1/8)$, $F(11, 1/16)$ and $F(10, 1/32)$ have $rank = 3$. From Figure 8, it is easy to see that the rank of $F(k, t)$ corresponds to the length of the shortest path to (k, t) from the root, which is the top pair, e.g., (12,1/32). Also, for any pair of fascicles F_1, F_2 assigned the same rank, it is the case that $F_1 \not\geq_f F_2$ and $F_2 \not\geq_f F_1$.

Having defined the notion of rank, we can now solve the pattern extraction problem by processing fascicles in ascending order of rank. In our example, we first process (12,1/32) fascicles. Then we process all (11,1/32) fascicles and (12,1/16) fascicles, and so on. This order of processing fascicles is captured in Step 4 of the Multi-Extract algorithm shown in Figure 9.

The following formal result ascertains the correctness of the Multi-Extract algorithm. Proof of this lemma can

Fascicles	(12,1/16)	(10,1/12)	(8,1/8)
Population	10%	+ 12.5%	+ 12.5%
Games played [1,82]	[1,5]	([1,79])	([1,82])
Goals Scored [0,50]	[0,0]	[0,3]	[0,3]
Assists [0,80]	[0,2]	[0,5]	[0,5]
Points [0,130]	[0,2]	[0,7]	[0,7]
PlusMinus [-20,40]	[-2,1]	[-2,1]	([-13,8])
Penalty Mins [0,400]	[0,18]	[0,22]	([0,233])
Power Play Goals [0,20]	[0,0]	[0,1]	[0,1]
Short Handed Goals [0,9]	[0,0]	[0,0]	[0,1]
Game winning Goals [0,12]	[0,0]	[0,1]	[0,1]
Game tying Goals [0,8]	[0,0]	[0,0]	[0,1]
Shots [0,400]	[0,8]	[0,23]	[0,25]
Percentage of Goals Scored [0,20]	[0,0]	([0,5])	([0,5])

Figure 10: Descriptions of Sample Patterns in the NHL Data Set. Each column is a fascicle with the attribute ranges given. All attributes are compact except the ones in parentheses.

be found in [15]. The lemma basically shows that Step 4 of Multi-Extract is correct in producing $Fas(R)$ and maximizing $Fas(R)$ for each record R – based on all the candidate fascicles computed in Steps 1 and 3. But the algorithm as a whole is not complete because in Steps 1 and 3 we do not seek to compute all fascicles. The experimental results shown in Section 4.6 convincingly demonstrates that computing all fascicles is computationally prohibitive.

Lemma 1 (Correctness of Multi-Extract) Let $\mathcal{S}$ be the set of candidate fascicles generated in Steps 1 and 3 of Algorithm Multi-Extract. For any record R, let the set of maximal fascicles containing R output by Step 4 be $\mathcal{S}_R = \{F_1, \ldots, F_u\}$. Then: (i) there does not exist a pair F_i, F_j such that $F_i >_f F_j$, $1 \leq i, j \leq u$; and (ii) there does not exist a set $\mathcal{S}' \subseteq \mathcal{S}$ and $\mathcal{S}' \not\supseteq \mathcal{S}_R$ such that $\mathcal{S}' >_s \mathcal{S}_R$. □

5.3 Empirical Results

Below we present the result of applying the Multi-Extract algorithm to the NHL data set. The results from the larger AT&T data set were also interesting. But because we are unable to present these results without divulging proprietary information, we restrict our discussion only to the NHL data set.

Figure 10 shows three fascicles obtained in the NHL data set. Each row describes the range of a particular attribute. The range following the name of the attribute in the first column gives the range of the attribute of the entire data set. The first fascicle has all 12 attributes of the data set as compact attributes and a compactness tolerance of 1/16 of the attribute range. In spite of the stringent requirements, 10% of the players are in this fascicle. These are players who have very limited impact on every aspect of the game. The second fascicle accounts for an additional 12.5% of the population. Compared with the first fascicle, the additional players here could play in a lot more games and have some variations in scoring percentage. Even so, this group still had little impact on the game. The first two fascicles together indicate that almost 1 out of 4 players did next to nothing. Finally, on top of the first two fascicles, there are an additional 12.5% of players whose main contribution appears to be their penalty minutes, as indicated by the dramatic expansion of the range of `Penalty Mins` ([0,22] to [0,233]). The performance of the Multi-Extract algorithm is very similar to that of the Multi-k algorithm, and hence not presented here.

6 Related Work

There are numerous studies on partitioning a data space spanned by a relation or a collection of points, including the studies on clustering (e.g., [17, 21, 6, 1, 10]). Almost all clustering algorithms, with the exception of CLIQUE [1], operate in the original data space, corresponding to row partitioning of a relation. Like CLIQUE, fascicles correspond to both row and column partitioning. CLIQUE is a density based method that finds all clusters in the original data space, as well as all the subspaces. Clusters found by CLIQUE are largest regions of connected dense units, where pre-binning is used to create a grid of basic units. This notion of clusters is fundamentally different from fascicles. Furthermore, the study conducted in [1] does not address the storage minimization problem and the pattern extraction problem analyzed here. Despite all the above differences, the FAP algorithm presented here can be considered a variant of CLIQUE. As shown in Section 4.6, it suffers from the same pitfalls of requiring pre-binning and operating in a bottom-up level-wise computational framework.

One important aspect of fascicles is dimensionality reduction (which leads to compression). In fact, “feature reduction”, is a fundamental problem in machine learning and pattern recognition. There are many well-known statistical techniques for dimensionality reduction, including Singular Value Decomposition (SVD) [9] and Projection Pursuit [8]. The key difference here is that the dimensionality reduction is applied to the entire data set. In contrast, our techniques handle the additional complexity of finding different subsets of the data, all of which may permit a reduction on different subsets of dimensions. The same comment extends to FastMap [7], the SVDD technique [14], and the DataSphere technique [11].

The rearrangement of columns in a relation has been shown to affect the compression achieved in previous studies, such as [19, 18]. With fascicles, this idea is carried further since the columns, as well as the number of columns, could be arranged differently for different subsets of tuples.

7 Conclusions and Future Work

Data sets often have approximately repeated values for many attributes. We have taken a first step towards identifying and exploiting such repetition in this paper. We have introduced the notion of a fascicle, which is a subset of records in a relation that have approximately matching values for some (but not necessarily all) attributes, categorical or otherwise. We have presented a family of algorithms to find fascicles in a given relation. We have evaluated how these algorithms behave under different choices of parameters, and shown how fascicles can be used effectively for the tasks of storage reduction and pattern extraction. Given that optimality is hard to obtain for both tasks, our algorithms produce good results.

We believe that the compression aspect of fascicles can provide good support of approximate query answering in the style championed in [4]. The dimensionality reduction aspect of fascicles can find applications in reducing indexing complexity, reverse engineering of schema decomposition, vocabulary-based document classification, and other operations that are exponential in the number of dimensions. Finally, as shown in Section 5, the pattern extraction aspect of fascicles can find applications in any data set that has (approximately) repeated values for many attributes.

As for future work, we believe that fascicles have opened the door to conducting data reduction (and analysis) not on the coarse granularity of the entire data set, but on the fine granularity of automatically identified subsets. As such, it would be interesting to see how the notion of compactness studied here change to other more sophisticated criteria, such as subsets with strong correlation, subsets amenable to SVD, etc.

References

[1] R. Agrawal, J. Gehrke, D. Gunopulos and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. *Proc. 1998 SIGMOD*, pp 94–105.

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

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. *Proc. 1994 VLDB*, pp 478–499.

[4] D. Barbará, W. DuMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. Ioannidis, H. V. Jagadish, T. Johnson, R. Ng, V. Poosala, K. A. Ross, K. C. Sevcik. The New Jersey Data Reduction Report. *IEEE Data Engineering Bulletin*, 20, 4, Dec. 1997.

[5] S. Brin, R. Motwani, J. Ullman and S. Tsur. Dynamic itemset counting and implication rules for market basket data. *Proc. 1997 SIGMOD*, pp 255–264.

[6] M. Ester, H.P. Kriegel, J. Sander and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noises. *Proc. 1996 KDD*, pp 226–231.

[7] C. Faloutsos and K. Lin. FastMap: a Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Datasets. *Proc. 1995 ACM-SIGMOD*, pp. 163–174.

[8] J.H. Friedman and J.W. Tukey. A Projection Pursuit Algorithm for Exploratory Data Analysis. *IEEE Transactions on Computers*, 23, 9, pp 881–889, 1974.

[9] K. Fukunaga. Introduction to Statistical Pattern Recognition. *Academic Press*, 1990.

[10] S. Guha, R. Rastogi, K. Shim. CURE: an Efficient Clustering Algorithm for Large Databases. *Proc. 1998 ACM-SIGMOD*, pp. 73–84.

[11] T. Johnson and T. Dasu. Comparing massive high-dimensional data sets. *Proc. 1998 KDD*, pp 229–233.

[12] R. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, Plenum Press, 1972, pp 85–103.

[13] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A.I. Verkamo. Finding interesting rules from large sets of discovered association rules. CIKM 94, pp 401–408.

[14] F. Korn, H.V. Jagadish, C. Faloutsos. Efficiently Supporting Ad Hoc Queries in Large Datasets of Time Sequences. *Proc. 1997 ACM-SIGMOD*, pp. 289–300.

[15] J. Madar. *Fascicles for Semantic Compression and Pattern Extraction*, MSc. Thesis, Department of Computer Science, University of British Columbia, 1999.

[16] H. Mannila and H. Toivonen. Level-Wise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1, 3, pp 241–258.

[17] R. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. *Proc. 1994 VLDB*, pp 144–155.

[18] Ng and Ravishankar. Block-Oriented Compression Techniques for Relational Databases. TKDE, April 1997, pp 314–328.

[19] Olken and Rotem. Rearranging Data to Maximize the Efficiency of Compression. *Proc. 1986 PODS*, pp 78-90.

[20] M. Smyth. Power Domains. *Journal of Computer and System Sciences*, 16, 1, pp. 23–36, 1978.

[21] T. Zhang, R. Ramakrishnan and M. Livny. BIRCH: an efficient data clustering method for very large databases. *Proc. 1996 SIGMOD*, pp 103–114.