Programming Patterns for Architecture-Level Software Optimizations on Frequent Pattern Mining

Mingliang Wei Changhao Jiang Marc Snir
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin, Urbana, IL 61801-2302, USA
{mwei1, cjiang, snir}@uiuc.edu

Abstract

One very important application in the data mining domain is frequent pattern mining. Various authors have worked on improving the efficiency of this computation, mostly focusing on algorithm-level improvement. More recent work has explored architecture specific optimizations of this computation. Our goal in this paper is to provide a systematic approach to architecture-level software optimizations by identifying applicable tuning patterns. We show the generality and effectiveness of these patterns by tuning several frequent pattern mining algorithms and showing significant performance improvements.

1. Introduction and motivation

Frequent pattern mining, also known as frequent itemset mining, aims to discover groups of items that co-occur frequently in a database. This is a fundamental data mining problem with many applications. Since the introduction of this problem by Agrawal et al. [2], a large number of algorithms [2, 3, 14, 12, 7, 30, 34, 15, 21, 25, 33] have been proposed. No one algorithm dominates: previous research has shown that the performance of these algorithms is very dependent on input characteristics [13, 19]; we have also found that the performance is very dependent on platform specific optimizations.

We study in this paper the issue of adapting an algorithm to platform characteristics. We use the term *Architecture-Level Software Optimizations (ALSO)* to denote such architecture specific optimizations; by ALSO we mean optimizations that are beyond the capabilities of current compilers, because they require high level transformations that are often application specific, and often require information that is not available to compilers.

Pattern, in software engineering terminology, is a general repeatable solution to a commonly-occurring problem in software design. A pattern is not a finished design that can be transformed directly into code; it is a description or template for how to solve a problem that can be used in many different situations. In this paper we study ALSO tuning patterns: general tuning techniques that can solve performance issues that recur in many codes; and can be easily applied by algorithm implementors.

To our knowledge, our paper provides the first systematic study of architecture-level software optimizations for frequent pattern mining. ALSO techniques such as cache-conscious data access, prefetch and SIMDization have been applied in scientific computing, multimedia and database, but have had few applications to pattern mining. Ghoting et al. [11] have proposed optimizations for some tree based implementations. Adaptive data structures have been used in [21, 20, 24, 29]. These papers have studied algorithms in isolation and little work has been done to develop optimizations that generalize to multiple implementations. We study tuning patterns that have broad applicability. This includes changes in in-memory database layout to improve the spatial locality; cache-conscious and optimization-friendly data structure design; and data accessing and processing patterns that improve temporal locality, reduce memory access latency and improve computation. Some of the tuning patterns, such as *lexicographic ordering* and *wave-front prefetch* are, to our knowledge, new. The *aggregation*, *compaction*, *software prefetch* and *SIMDization* patterns are for the first time used in frequent pattern mining. We demonstrate the general applicability and effectiveness of these tuning patterns by selectively applying them to three efficient and very different pattern mining algorithms, LCM, Eclat and FP-growth, and showing significant improvements.

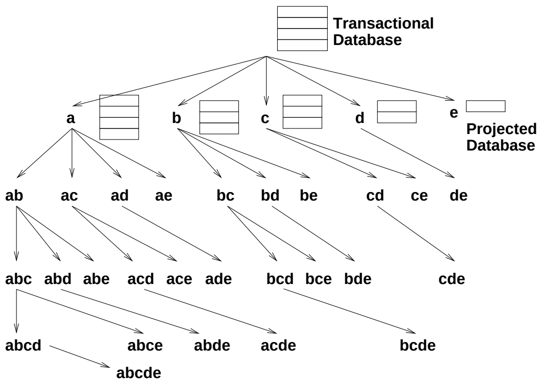


Figure 1. The traversal space of itemsets

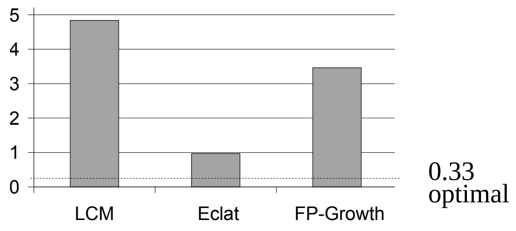


Figure 2. CPI for most time consuming functions

2. Frequent pattern mining

2.1. Frequent pattern mining algorithms

Frequent pattern mining was introduced by Agrawal et al. [2] in the study of association rule mining. Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of m items, and let a *database* $\mathcal{T} = \{t_1, t_2, \ldots, t_n\}$ be a set of n transactions, where each *transaction* t_i is a subset of $\mathcal{I}$. Any subset of $\mathcal{I}$ is called an *itemset*. The *support* for an itemset is defined as the number of transactions that subsume the itemset. The task of frequent pattern mining is, given a transactional database $\mathcal{T}$ and a support threshold s, to output all itemsets with support greater than or equal to s.

Given a database with m items, there are potentially 2^m itemsets, which form a *lattice of subsets* over $\mathcal{I}$. Figure 1 shows an example of itemset traversal space for a database with $\mathcal{I} = \{a, b, c, d, e\}$. A typical depth-first algorithm, starts with the initial database, and recursively creates projected databases that consist of the transactions containing a particular item.

2.2. Optimization potentials

Figure 2 shows the CPI (Cycle Per Instruction) of the most time consuming functions in three leading frequent pattern mining codes. The *LCM* implementation got best implementation award at the FIMI'04 workshop [19]; the *FP-Growth* got the award at the FIMI'03 workshop [13]; the *Eclat* implementation is taken from the repository of FIMI'04. These three kernels cover most common data structures and data access patterns. The CPI data is collected on the Pentium D system described in column M1 in Table 5. Each core of the Pentium D processor is able to retire 3 μops per cycle, with an optimum CPI of 0.33.

As we can see from Figure 2, there is plenty of room for performance improvements. Our general approach is to optimize memory accesses for those codes with a high CPI and cache miss rate; and to optimize the arithmetic operations for those with a low CPI and cache miss rate. The LCM and FP-Growth algorithms are clearly memory bound, as they have a high CPI, and further studies reveal that they also have high cache miss rates. As to Eclat, it has a low CPI and is computation bound. We provide details on optimization patterns that improve the performance of these codes in Section 3.

3. ALSO patterns for frequent pattern mining

We have identified several optimization problems that occur frequently. We document the solutions to these problems as patterns in this section. Requiring application specific knowledge to apply, these patterns are high-level optimization techniques, complementary to compiler optimizations such as loop unrolling and software pipelining. These optimizations can be roughly generalized to three categories of patterns: patterns to optimize the database layout, patterns to optimize the internal data representations and patterns to optimize data accesses.

3.1. Common optimization opportunities

The first optimization modifies the layout of the in-memory (projected) databases. The ordering of transactions in these databases is not significant, and transactions can be permuted. We can improve the locality of accesses to the database by choosing a suitable permutation. This can have a significant impact as database transactions are often repeatedly accessed during computation. In addition, the locality property is partly inherited by the lower-level, projected databases.

The second optimization concerns the data structure used for the database. We focus on representations that are *cache-friendly*, i.e., reduce cache misses; and are *optimization-friendly*, e.g., inserting prefetch pointers for software prefetch.

Finally, memory latency hiding and arithmetic acceleration techniques can be used for memory bound and computation bound applications respectively.

We describe the ALSO patterns in detail in the following sections; we use the symbol ***Pi*** to mark the ith tuning pattern.

3.2. Database Layout

This optimization is used when unordered transactions are frequently accessed in a particular order. It moves the transactions that are often successively accessed to consecutive memory locations to improve the spatial locality, reducing both cache and TLB misses.

The reordering process may require additional memory and it is especially expensive when the database is large and the transactions are long. Among all of the databases created during the mining process, the initial database is the largest and is accessed most frequently. Furthermore, the layout of the initial database is preserved to some extent in the projected databases. Therefore, we focus on improving locality in the initial database.

P1: **Lexicographic ordering.** The *frequency* of an item is the number of transactions that contain that item. We order the items in each transaction in decreasing frequency order. We then order the transactions in lexicographic order, based on the decreasing frequency order of the items, i.e., the alphabet are items in decreasing frequency order. The transformation is illustrated in Table 1. Lexicographic ordering can be used in various algorithms, we give an example of its use in an array based horizontal database setting, which is used in LCM.

tid	transaction
0	{a, c, f}
1	{b, c, f}
2	{a, c, f}
3	{d, e}
4	{a, b, c, d, e, f}

$\Rightarrow$

Alphabet: c, f, a, b, d, e

tid	transaction
0	{c, f a}
1	{c, f, a}
2	{c, f, a, b, d, e}
3	{c, f, b}
4	{d, e}

Table 1. Lexicographic ordering

As described in section 2.1, an operation common to frequent pattern mining algorithms is to walk through the (projected) databases and construct lower-level projected databases. All transactions that contain a particular item are accessed in this process. The lexicographic ordering moves transactions containing the same item close to each other, so that spatial locality is improved; cache and TLB misses are reduced. This reduction in cache misses will be most significant when the transactions are short, as in long transactions, most of the spatial locality is already captured by storing items in each transaction in consecutive memory locations.

In the lexicographic layout all transactions on the most frequent item are contiguous; transactions on the second most frequent item have at most one discontinuity; and so on. This ordering will tend to reduce the total number of discontinuities, and especially reduce discontinuities for frequent items, thus improving locality.

If a bit vector is used to represent transactions occurrences in a vertical database then the lexicographic ordering enables another optimization, 0-escaping (see section 4.2 for details). For the tree-based horizontal database, we lexicographically reorder transactions before tree construction. This improves the temporal locality for insertion and places nodes that are adjacent in a path in consecutive memory locations, thus improving the spatial locality for later traversal. For tree based algorithms, the difference between the lexicographic ordering and depth-first order storage [11] is that the lexicographic ordering is performed as a preprocessing before the tree is built and it optimizes both insertion and traversal operations, whereas the depth-first ordering is a reorganization of the tree structure, only to optimize the traversal.

3.3. Data structures

P2: **Data structure adaptation.** The data structure used to represent in-memory databases can be adapted to the input characteristics.

We can think of a database with n transactions and m items as of a $m \times n$ table A; $A_{ij} = 1$ if transaction i contains item j, $A_{ij} = 0$, otherwise. There are several choices on how to represent this table.

Feature 1: The table can be stored *horizontally* in transaction-major order; or *vertically*, in an item-major order.

Feature 2: Assume a transaction-major order (some similar choices exist for item-major order). (1) One can store each row as a bit vector, so that the table is represented as a *dense* $m \times n$ boolean matrix; (2) alternatively, one can use a *sparse* representation that stores, for each row, the indices of the non-zero entries; (3) finally, one can use a *prefix tree* representation where shared nodes are used to represent a common prefix of several rows. These three representations are illustrated in Figure 3, for the database shown in Table 1.

Another example of the data structure adaptation pattern is to use a compression scheme whereby fewer bytes are used to represent the common cases.

P3: **Aggregation.** It is used to improve performance for the traversal of linked data structures, which are common in frequent pattern mining. There are two problems with such traversal. The first is that the traversal is memory latency bound, as successive memory accesses cannot be overlapped. The second is poor spatial locality, as nodes may occupy less than a cache line and successive nodes are not necessarily stored in consecutive locations.

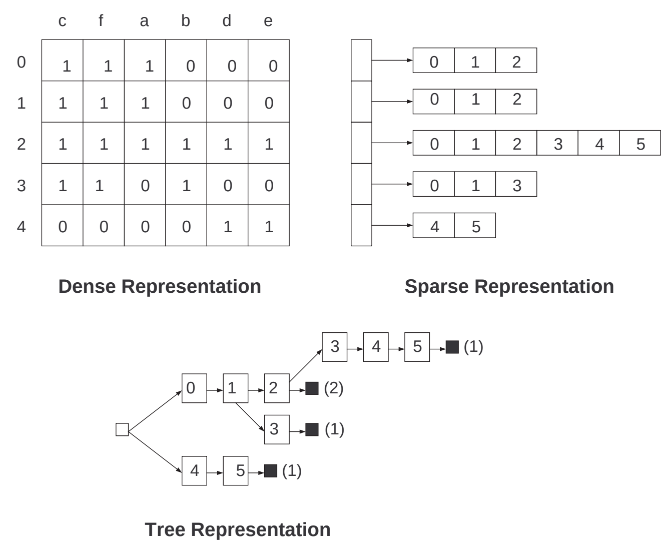


Figure 3. Database Representations

Performance is improved by aggregating multiple consecutive nodes on a traversal path into one *supernode*. Making each supernode the size of a cache line seems to be optimal.

When this optimization is applied to trees, then nodes can be replicated, as they are shared by multiple path; this partially offsets the compression achieved by using a prefix tree representation. Figure 4 shows the aggregation of a tree structure. We compress four consecutive tree levels into one *superlevel*, aggregating all paths in the superlevel into one node.

The aggregation is efficient only when the data structure is seldom updated, as an insertion to the middle of an aggregated linked list is expensive.

P4: **Compaction.** *Compaction* copies data that are scattered in memory into consecutive memory locations, to improve spatial locality. Compaction is worthwhile if the cost of copying is amortized over a large number of subsequent accesses.

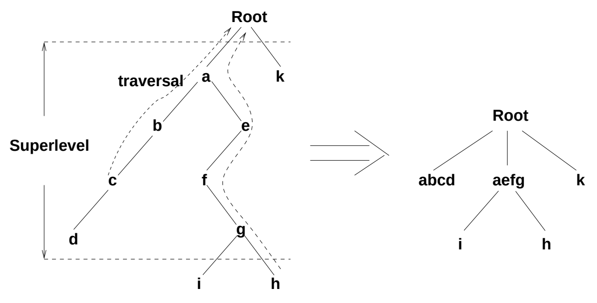


Figure 4. Aggregation for tree

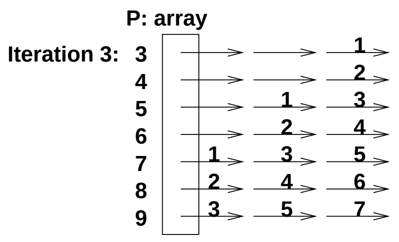


Figure 5. Wave-front prefetch

P5: **Prefetch Pointers.** Some ALSOs are implemented by creating additional data structures. An example is the use of prefetch pointers to improve the traversal of linked data structures. *Prefetch pointers* [28] are inserted in a preprocessing stage, pointing from each node to other nodes that are likely to be accessed soon after the access to that node. Prefetch pointers allow a better overlap of memory accesses, at the expense of extra storage and preprocessing time.

3.4. Data access

Optimizations in this category focus on reducing memory bottlenecks.

P6: **Tiling.** Tiling is used when large data structures are accessed frequently. Tiling for dense matrix operations is well-known and can be applied to variants of frequent pattern mining that use such matrices. Tiling for trees is proposed in [11].

P6.1: **Tiling for sparse representations.** Sparse matrices are commonly used to represent the database. Temporal locality is poor in the common case when a large database is repeatedly traversed. Researchers have proposed tiling for sparse matrix vector multiplications [16, 18, 17]. This work is, however, tied to sparse matrix and dense vector operations and does not directly apply to frequent pattern mining. Our basic idea for tiling is to slice the sparse matrices into horizontal tiles according to the row range and then to process one tile at a time, with an outer loop that walks through tiles and an inner loop that traverses entries within a tile. See section 4.1 for an example. The disadvantage of tiling is the overhead for the added level of loop nesting.

P7: **Software prefetching.** Prefetching [23] is an effective way to hide memory latencies. Software prefetching can be used for linked data structure, where hardware prefetching does not work well. Software prefetching can be performed by following the pre-inserted prefetch pointers [28]. Mispredicted prefetches, however, may impair the performance.

Pattern	Spatial locality	Temporal locality	Memory latency	Compu-tation
Lexicographic ordering	√	√		√
Data structure adaptation	√			
Aggregation	√		√	
Compaction	√	√		
Software prefetch			√	
Tiling		√		
SIMDization				√

Table 2. ALSO patterns

P7.1: **Wave-front prefetching.** Arrays of short linked lists (see Figure 5) are commonly used in frequent pattern mining. The common access pattern is to traverse each linked list. Existing linked list prefetch algorithms have good performance only when the linked lists are long and do not apply to our case. Instead, we propose to use *wave-front prefetching*. The basic idea is that we can prefetch entries from different linked lists in the same iteration. In Figure 5, the numbers over the arrows are the iteration numbers when the correspondent entries are prefetched; the indices on the left indicate the iteration when the linked list is traversed. Suppose the memory latency is less than the time to traverse two short linked lists; then we can prefetch three links in each iteration as shown in Figure 5. At the time when entries need to be prefetched, their addresses have already been loaded by previous prefetches.

3.5. Instruction parallelism

Optimizations in this category focus on improving instruction parallelism, for computation bound kernels.

P8:* *SIMDization. SIMD instructions are available on most of the commodity processors; they can accelerate computation bound applications. Memory prefetch instructions are also available in the SIMD instruction set. The SIMDization optimization, however, requires sufficient data-level parallelism in the algorithm and needs to handle memory alignment problems.

Table 2 summarizes the ALSO patterns and shows what improvements these optimizations can provide.

4. Case studies: LCM, Eclat and FP-Growth

We selected three highly optimized frequent pattern mining kernels to evaluate the applicability and effectiveness of our ALSO patterns. They cover most efficient algorithm space and data structure design choices. The *LCM* implementation got best implementation award at the FIMI'04 workshop [19]; the *FP-Growth* is an efficient implementation of the FP-Growth algorithm; the *Eclat* implementation is an optimized version of that taken from the repository of FIMI'04 [6]. The Eclat implementation that we studied uses a bit vector data structure for the transactional database. Table 3 shows the characteristics of the three kernels evaluated. We did not cover breadth-first search algorithms, such as *Apriori* [3], because the depth-first search algorithms are generally considered to be more efficient and our study is focusing on kernels with different data representations, rather than a study on different algorithms. We applied several locality and memory optimization patterns on LCM and FP-Growth, and mainly used computation optimization patterns on Eclat. Table 4 shows the patterns that we have studied for these three kernels. The "√" marks those patterns that we have applied in the case studies. The "◯" marks the optimizations that have already been proposed in the literature, which we did not incorporate in the evaluation. "—" are the patterns that we have not studied.

Kernel	Database type	Data structure	Bound
LCM	horizontal	array	memory
Eclat	vertical	bit vector (array)	computation
FP-Growth	horizontal	tree	memory

Table 3. Characteristics of LCM, Eclat and FP-Growth

Patterns	LCM	Eclat	FP-Growth
Lexicographic ordering	√	√	√
Data structure adaptation	—	◯	√
Aggregation	√	—	√
Compaction	√	—	√
Pointer prefetching	—	—	√
Tiling	√	—	◯
Software prefetch	√	—	√
SIMDization	—	√	—

Table 4. Optimization patterns for LCM, Eclat and FP-Growth

4.1. LCM

Since LCM (Linear time Closed itemset Miner) [32] is memory bound, we focus on patterns that could improve memory performance.

Figure 6 shows the main data structure that is traversed by the CALCFREQ function which takes 54.43% of the total execution time. The data structure consists of a transaction-major sparse array that represents the database, augmented by an item-major sparse array *OccArray* that is used for speeding up the construction of projected databases. Each column (called *occ*, shown as shaded column) stores point-

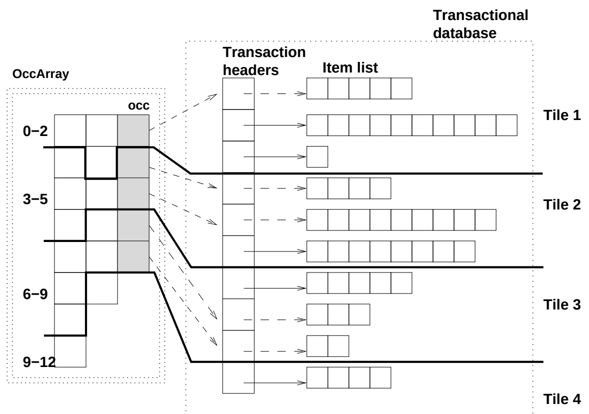


Figure 6. Main data structure used in CALCFREQ

ers to the headers of transactions containing the corresponding item. For each call of CALCFREQ, the execution traverses one of these columns, follows the pointers to transaction headers and accesses all the items in these transactions. Pointers dereferenced in this process are shown as dashed arrows in Figure 6.

We use *lexicographic ordering* to improve the spacial locality of the initial database.

Another function that takes 25.5% of the total execution time is RMDUPTRANS. It removes identical transactions in the database. In the original implementation, bucket (radix) sorting is used to find duplicated transactions. A linked list is used to link all the transactions that fall into the same bucket. As the linked list is mostly read only, we use *aggregation* to reduce dereferences and improve spatial locality.

The *frequency counters* that are frequently used in CALCFREQ are not in contiguous locations. They are structured with the *OccArray*. By *compaction* the frequency counters are moved to contiguous memory locations, thus improving the locality and reducing the cache and TLB misses.

As CALCFREQ involves the traversal of a list of short linked lists, we use *wave-front prefetch* for pointers in *occ* array and pointers in transaction headers.

The function CACLFREQ is called from a loop that invokes CALCFREQ for columns of *OccArray*. For each run of CALCFREQ, in the worst case, the whole database is scanned, with little cache reuse. *Tiling for sparse representation* could be done for invocation of CALCFREQ for the columns of the OccArray in the following way: The array *OccArray* is split into tiles (separated by dark lines in Figure 6). Each tile contains the transactions within a particular offset range. The internal loop performs all the CALCFREQ computations for one tile; the external loop iterates over all tiles. We choose the tile size to fit in the L1 cache.

4.2. Eclat

The Eclat algorithm [6] uses a vertical, dense bit matrix representation. The columns represent initially the occurrence of items in transactions; as the algorithm proceeds, the columns represent the occurrence of itemsets in transactions. The "and" of the bit vectors for two itemsets computes the bit vector for the union of the two itemsets. 98% of the total execution time is spent in these vector and's and in counting the number of ones in the resulting vectors (frequency counting).

By *lexicographic ordering* the initial transactions, the 1s in the bit vectors for the most frequent items are clustered. In particular, the 1s for the most frequent item are consecutively stored at the beginning of the vector. The lexicographic ordering enables the 0-*escaping*. The idea of 0-escaping is to skip intersecting and frequency counting on the bit vector sections where either operand vectors are all 0s. This is achieved by storing, for each vector, the start and end position of a 1-*range*, which includes all the 1s in the bit vector. The ranges are initialized by computing the first and last 1 in each item bit-vector and updated by intersecting the corresponding 1-ranges when two bit vectors are and'ed. Then the intersection and frequency counting are performed only within the computed range, skipping 0s at the beginning and the end of the intersecting vectors. The reordering improves the performance of 0-escaping, as the 1s are moved together and the 1-range for the correspondent bit vectors becomes shorter; fewer operations need to be performed. Note that the 1-ranges thus computed are conservative, but not necessarily optimal.

There is plenty of data-level parallelism in Eclat. Clearly, the bit vector intersection can be *SIMDized*. In the original implementation, table lookups are used to count the number of 1s in the bit vector. The table lookup is an indirect load, which cannot be SIMDized. We use computations to count the frequency of ones, which can be easily SIMDized.

4.3. FP-Growth

FP-Growth [15] uses an augmented prefix tree known as the FP-tree (see Figure 7) to represent the database. The most common access pattern is to follow pointers in *head of node links* to access the nodes labeled by the same item (shown as dashed arrows in Figure 7). For each node accessed, the path from that node to the root is then traversed.

The FP-Growth algorithm has a high CPI and cache miss rate; it is a memory bound computation. Several optimizations have been proposed in [11], which include initial database reorganization, tiling, etc. We propose to use lexicographic ordering, data structure adaptation, aggregation and software prefetch to improve the performance. These new techniques are complementary to the optimizations that have been previously studied.

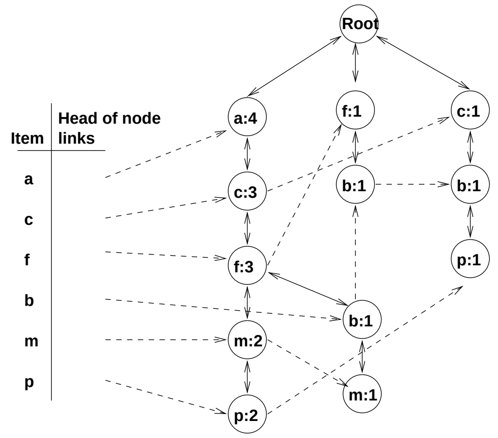


Figure 7. An FP-tree / prefix tree

A *lexicographically ordering* of the transactions for FP-Growth provides two benefits: First, the tree construction is more cache efficient. The tree building process inserts transactions one by one. After the reordering, as each transaction shares many items with the previous one, most of the nodes accessed during an insertion are already in the cache. Second, pairs of parent node and child node, which are often accessed together during later traversals, are likely to be stored next to each other.

A useful *data structure adaptation* is to represent the *item ID* of a node with fewer bytes, using differential encoding: one stores the difference between the local item ID and the ID of the parent node; this can usually be stored in a single byte, with an escape code to handle the exception cases. This reduces the node size and memory requirements dramatically.

Aggregation can be used in FP-Growth to improve the spatial locality of tree traversal. As nodes that are shared between paths need to be copied, the aggregated tree requires more memory. We find that, when combined with data structure adaptation, the memory requirements are moderate.

Prefetch pointers can also be inserted to help *software prefetch*.

4.4. Optimization results

We evaluate our ALSO patterns by applying them to frequent pattern mining kernels and benchmark them on two different platforms. Table 5 shows the configuration of the two systems. We use two synthetic data sets generated by the IBM Quest Dataset Generator, one real data set called WebDocs [22], and another real data set called AP from the Text Research Collection [1]. Table 6 shows the data sets and the support that we use in the evaluation. We choose WebDocs and AP, because other available real world data sets are too small.

Parameters	M1	M2
Processor type	Intel Pentium D 830 dual core 3GHz	AMD Athlon 64 X2 dual core 4200+
L1 cache per core	16KB D-cache 12KB trace cache	64KB D-cache 64KB I-cache
L2 cache per core	1MB	512KB
Memory	4GB	4GB

Table 5. Experimental platforms

The baselines of our speedup are the best implementation of FIMI'04: LCM, Eclat from FIMI'04 and an efficient implementation of FP-Growth. The baseline running times are listed in Figure 8. The speedup is based on overall execution time.

Figure 8 shows the speedup of the optimized LCM, Eclat and FP-Growth on systems M1 and M2. In these figures, *Lex* means the speedup we get after we lexicographically reorder the initial database; *Reorg* refers to the data structure optimizations such as aggregation and compaction; *Pref* refers to software prefetching; *Tile* and *SIMD* are the tiling and SIMDization pattern respectively. We first apply each ALSO pattern to each algorithm to see the benefit of a single pattern. Then we test the performance for code that incorporates all applicable patterns. For each cluster of columns, the second column from the right, labeled *all*, is the performance after we apply all applicable patterns; the rightmost column, labeled *best*, is the best performance that we can get by selectively applying the patterns. Most of the time *best* and *all* are the same, indicating that each of the optimizations provides some benefit, when combined with all others. In some cases, for example in Figure 8(a) data set DS4, the best optimization is not *all*. Instead, it is the combination of prefetch and data structure patterns. The texts above the best bars show the combination of patterns that yields the *best* performance.

We can immediately see that there is no single best algorithm. For the baselines, the Eclat algorithm performs the best on DS3, while for other data sets, LCM is the fastest algorithm. The FP-Growth also has a competitive performance, and in some cases is close to optimal.

We see an overall performance improvement for the *best* combination of patterns, ranging from 1.05 to 2.1. We also see a significant performance improvement for the application of each individual pattern. To be specific, the lexicographic ordering provides up to 1.5 speedup. Prefetch gives

Parameters	DS1	DS2	DS3	DS4
Name	T60I10D300K	T70I10D300K	WebDocs	AP
# transactions	300K	300K	500K	1.8M
Support used	3000	3000	50000	2000

Table 6. Data sets and support in the evaluation

up to 1.3 speedup. The SIMDization provides a speedup between 1.25 and 1.45 on M1. In FP-Growth, data structuring technique, particularly, data structure adaptation and tree aggregation gives a speedup of 1.6. Tiling in LCM gives a speedup of up to 1.75. The tiling for FP-Growth has been studied elsewhere [11], and yields a speedup of about 2.

The effectiveness of optimizations is input dependent. For the inputs shown in Figure 8(a), tiling in most cases provides the most significant speedup to LCM, in particular, for DS1 and DS2, tiling produces a speedup of over 1.5. In DS4, tiling, however, produces almost no speedup. Each software optimization have some associated cost, which can negate its benefit. DS4 is a very sparse data set, where transactions containing one item are scattered over memory. In this sparse data set, tiling does not introduce much data reuse. The lexicographic ordering is not performing well in FP-Growth for DS4, because the data set contains too many transactions, so that lexicographic ordering is very time consuming.

In general, software prefetch and aggregation work better for long linked data structure, as there is more potential for latency reduction. E.g. in FP-Growth, a greater average transaction length would be an indication of deeper FP-tree. Lexicographic ordering would work better if the order of transactions in input database are random. One could define a metric that capture the clustering of the input transactions. Tiling would work better when the transactions are clustered, as it tends to have more cache reuse in this case.

The optimization results are also platform dependent. Figure 8(b) shows the same experiments as in Figure 8(a) but on a different platform M2. Although optimizations have similar impact on the performance, the magnitude is different. In particular, in Figure 8(c) and Figure 8(d), the SIMD performance of M2 is not so significant as that of M1, providing less than 1.2 speedup for the best case.

Finally, the optimizations seem not to be independent. Several optimizations may have the same objective (e.g., improving spatial locality). If one optimization is sufficiently effective, then the other optimization may add little value, while still incurring an overhead.

Our results show that for Eclat and for FP-Growth, there is on each platform one code that is best for all inputs, while LCM requires different codes for different inputs. However, due to the small number of experiments one cannot attach too much significance to this conclusion.

5. Related work

Since the introduction of frequent pattern mining, a large number of of algorithms and implementations [2, 3, 14, 12, 7, 30, 34, 15, 21, 25, 33] have been proposed. Different algorithms and implementations use significantly different data representations and access them differently. Some algorithms adapts algorithm's data structures and traversing order according to input features [21, 20, 24, 29].

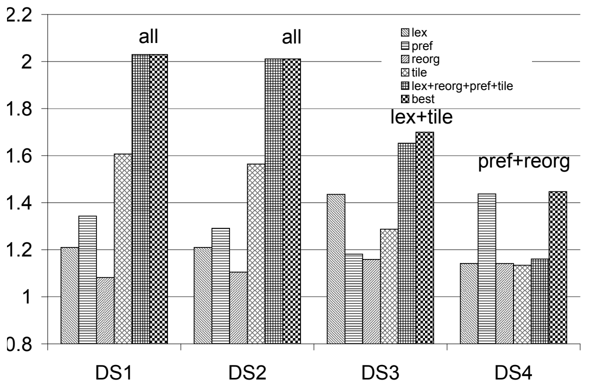
Ghoting et al. [11] have studied the problem of ALSO for some tree-based frequent pattern mining implementations. They proposed cache conscious prefix-tree to improve spatial locality and also enhance the benefits from hardware cache line prefetch. Tiling is used to improve the temporal locality. Targeting SMT processors, a thread-based decomposition is used to ensure cache reuse between threads that are co-scheduled at a fine granularity. We have included some of these optimizations as patterns for completeness, knowing that many of these optimizations are tied to tree based implementations. However, we did not apply them in our evaluation because we wanted to study the impact of the newly proposed patterns. We believe that the new optimizations are complementary to existing ones.

In the database domain, optimizations have been proposed for core database algorithms to improve cache performance [5, 31]. Rao and Ross [26, 27] proposed two new types of data structures: Cache-Sensitive Search Trees and Cache-Sensitive B+ Trees. Studies [9, 10, 8] have shown software prefetch could improve searches on B+ trees and Hash-Join operations. Software jump-pointer prefetch has been proposed and evaluated on intensive pointer benchmarks [28], which yields an average speedup of 15%. Ailamaki et al. [4] examined DBMS performance on modern architectures. They concluded that poor cache utilization is the primary cause of extended query execution time.

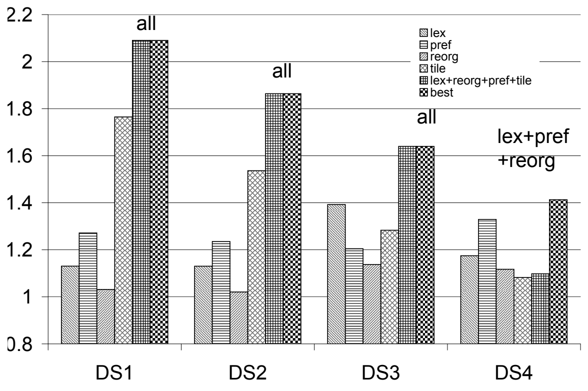
6. Conclusion and future work

In this paper, we have proposed various ALSO patterns for frequent pattern mining. These patterns are effective and generally applicable to various implementations of frequent pattern mining algorithms. The patterns are not tied to particular implementations or applications and can be used in other domains.

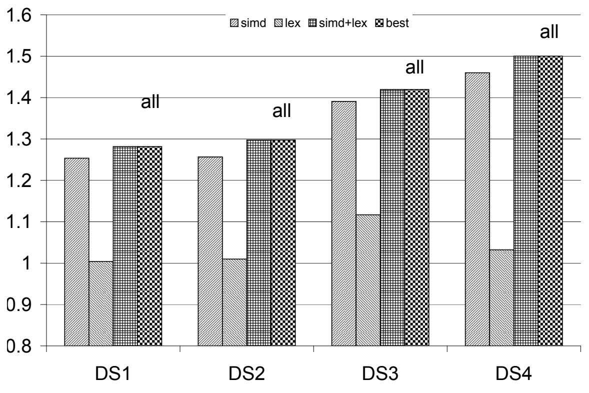
We have verified the applicability and effectiveness of these patterns in three highly optimized frequent pattern mining algorithms. Experimental results show that each of the patterns that we used is beneficial, and there is a good overall speedup of up to 2.1. Combined with previously proposed optimization strategies [11], the overall speedup could be even greater. This is quite impressive, given that we started with implementations that had already been carefully tuned. Surprisingly, the software prefetch does not give us as much as we have expected, providing a speedup of 1.3 for the best case. Although this is consistent with some of the previous research on prefetching [28], it is far from the speedup up to 2.9 in some existing work [9, 10, 8]. There are two main reasons: First, in some previous work, the speedup is evaluated for a particular execution phase, rather than the whole application run time. Second, previous research on prefetching used simulators or non-commodity processors. We believe the moderate



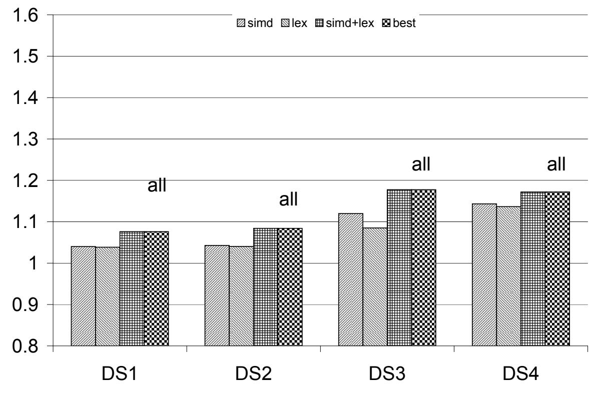
(a) LCM on M1,baseline in seconds(77,169,90,36)



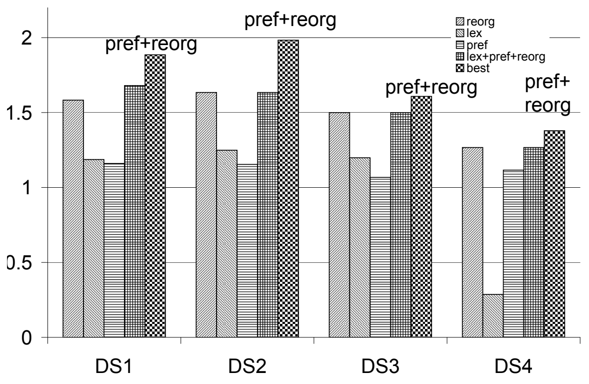
(b) LCM on M2, baseline (74,159,93,35)



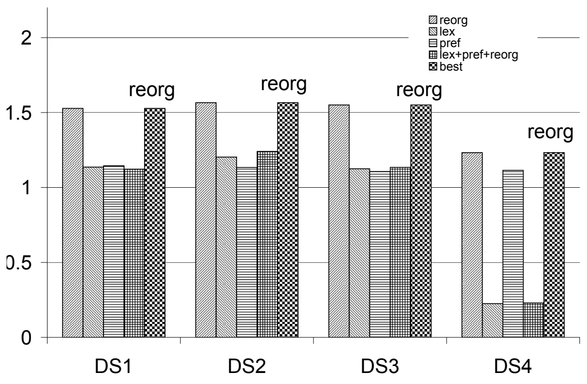
(c) Eclat on M1, baseline(137,270,50,751)



(d) Eclat on M2, baseline (142,285,50,887)



(e) FP-Growth on M1, baseline (157,345,94,50)



(f) FP-Growth on M2, baseline (135,293,89,46)

Figure 8. Speedup of LCM, Eclat and FP-Growth on M1 and M2

speedup for software prefetching is normal for commodity processors.

For completeness, we mentioned some other optimizations proposed in the literature; however, we did not include them in our evaluation, as these optimizations have shown effectiveness in previous work and we wanted to focus on the new patterns and those patterns that have never been applied in this domain. We believe the patterns that we have applied in the evaluation are complementary to those that have already been studied.

Our work shows that it is not only the case that one algorithm is not always best, but also it is not always the same set of transformations that most benefit a code. The right set of transformation depends both on the input and on the system architecture. We expect to explore in future work the problem of selecting an optimal set of transformations, given the input and machine parameters.

References

[1] Tipster information-retrieval text research collection on cd-rom. Gaithersburg, Maryland, March 1994. National Institute of Standards and Technology.

[2] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD'93*, pages 207–216, Washington, D.C., 1993.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB'94*, pages 487–499, 1994.

[4] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *The VLDB Journal*, pages 266–277, 1999.

[5] M. A. Bender et al. Cache-oblivious b-trees. In *FOCS'00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 399, Washington, DC, USA, 2000. IEEE Computer Society.

[6] C. Borgelt. Efficient implementations of apriori and eclat. In *FIMI*, 2004.

[7] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *ICDE'01*, pages 443–452.

[8] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In *ICDE'04*, page 116, 2004.

[9] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *SIGMOD'01*, pages 235–246, Santa Barbara, California, United States, 2001.

[10] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal prefetching b+-trees: optimizing both cache and disk performance. In *SIGMOD'02*, pages 157–168, Madison, Wisconsin, 2002. ACM Press.

[11] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y.-K. Chen, and P. Dubey. Cache-conscious frequent pattern mining on a modern processor. In *VLDB'05*, pages 577–588, Trondheim, Norway, 2005.

[12] B. Goethals. *Efficient Frequent Pattern Mining*. PhD thesis, University of Limburg, Belgium, 2002.

[13] B. Goethals and M. J. Zaki, editors. *FIMI'03: Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations*, Melbourne, Florida, USA, 2003.

[14] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *ICDM*, pages 163–170, 2001.

[15] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD'00*, pages 1–12, 05 2000.

[16] E.-J. Im. *Optimization the Performance of Sparse Matrix–Vector Multiplication*. PhD thesis, University of California, Berkeley, May 2000.

[17] E.-J. Im and K. A. Yelick. Optimizing sparse matrix kernels for data mining. In *SIAM International Conference on Data Mining*, Chicago, IL, April 2001.

[18] E.-J. Im, K. A. Yelick, and R. Vuduc. SPARSITY: Framework for optimizing sparse matrix-vector multiply. *International Journal of High Performance Computing Applications*, 18(1):135–158, February 2004.

[19] R. J. B. Jr., B. Goethals, and M. J. Zaki, editors. *FIMI'04: Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, Brighton, UK, 2004.

[20] G. Liu et al. Afopt: An efficient implementation of pattern growth approach. In *FIMI*, 2003.

[21] J. Liu, Y. Pan, K. Wang, and J. Han. Mining frequent item sets by opportunistic projection. In *KDD'02*, pages 229–238, Edmonton, Alberta, Canada, 2002. ACM Press.

[22] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri. Web-Docs: a real-life huge transactional dataset.

[23] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *ASPLOS-V*, pages 62–73, Boston, Massachusetts, United States, 1992.

[24] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive and resource-aware mining of frequent sets, 2002.

[25] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang. H-mine: Hyper-structure mining of frequent patterns in large databases. In *ICDM*, pages 441–448, 2001.

[26] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB'99*, pages 78–89, San Francisco, CA, USA, 1999.

[27] J. Rao and K. A. Ross. Making b+- trees cache conscious in main memory. In *SIGMOD'00*, pages 475–486, Dallas, Texas, United States, 2000.

[28] A. Roth and G. S. Sohi. Effective jump-pointer prefetching for linked data structures. In *ISCA'99*, pages 111–121, Atlanta, Georgia, United States, 1999.

[29] C. L. Salvatore. kdci: a multi-strategy algorithm for mining frequent sets.

[30] A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In *The VLDB Journal*, pages 432–444, 1995.

[31] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB'94*, pages 510–521, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[32] T. Uno et al. Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *FIMI*, 2004.

[33] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *KDD'03*, pages 326–335, New York, NY, USA, 2003.

[34] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In D. Heckerman, H. Mannila, D. Pregibon, R. Uthurusamy, and M. Park, editors, *In 3rd Intl. Conf. on Knowledge Discovery and Data Mining*, pages 283–296, 12–15 1997.