

nonordfp: An FP-Growth Variation without Rebuilding the FP-Tree

Balázs RÁCZ

Computer and Automation Research Institute
of the Hungarian Academy of Sciences
H-1111 Budapest, Lágymányosi u. 11.
bracz+fm4@math.bme.hu

Abstract

We describe a frequent itemset mining algorithm and implementation based on the well-known algorithm FP-growth. The theoretical difference is the main data structure (tree), which is more compact and which we do not need to rebuild for each conditional step. We thoroughly deal with implementation issues, data structures, memory layout, I/O and library functions we use to achieve comparable performance as the best implementations of the 1st Frequent Itemset Mining Implementations (FIMI) Workshop.

1. Introduction

Frequent Itemset Mining (FIM) is one of the first and thus most well studied problems of data mining. From the many published algorithms for this task, pattern growth approaches (FP-growth and its variations) were among the best performing ones.

This paper describes an implementation of a pattern growth algorithm. We assume the reader is familiar with the problem of frequent itemset mining[2] and pattern growth algorithms[5], like FP-growth, and hence we will omit their description here. For the reasons and goals for analyzing implementation issues of the FIM problem, see the introduction to the 1st FIMI workshop [3].

Our implementation is based on a variation of FP-tree, a similar data structure than used by FP-growth, but with a more compact representation that allows faster allocation, traversal, and optionally projection. It maintains less administrative information (the nodes do not need to store their labels (item identifiers), no header lists and children mappings are required, only counters and parent pointers), and allows more recursive steps to be carried out on the same data structure, without the need to rebuild it. There are also drawbacks of never rebuilding the tree: though projection is possible to filter conditionally infrequent items, the order of items cannot be changed to adapt to the con-

ditional frequencies. Hence the acronym of the algorithm: nonordfp.

We describe implementational details, data structure traversal routines, memory allocation scheme, library functions and I/O acceleration, among with the algorithmic parameters of our implementation that control different traversal functions and projection. The implementation is freely available for research purposes, aimed not only for performance comparison but for further tuning of these theoretical parameters.

2. Overview of the algorithm and data structures

As a preprocessing the database is scanned and the global frequencies of the items are counted. Using the minimum support infrequent items are erased and frequent items are renumbered in frequency-descending order. During a second scan of the database all transactions are pre-processed: infrequent items are erased, frequent items are translated and sorted according to the new numbering. Then the itemset is inserted into a temporary trie.

This trie is similar to the classic FP-tree: each node contains an item identifier, a counter, a parent pointer and a children map. The children map is an unordered array of pairs (child item identifier, child node index). Lookup is done with linear scan. Though this is asymptotically not an optimal structure, the number of elements in a single children map is expected to be very small, linear scan has the least overhead compared to ordered arrays with binary search, or search trees/hash maps. The implementation uses syntactics that are equivalent to the Standard Template Library (STL) interface *pair-associative container* thus it is easy to exchange this to the RB-tree based STL `map` or `hash_map`. It results in a slight performance decrease due to data structure overhead.

As a final step in the preprocessing phase this trie is copied into the data structure that the core of the algorithm will use, which we will describe later.

The core algorithm consists of a recursion. In each step the input is a condition (an itemset), a trie structure and an array of counters that describe the conditional frequencies of the trie nodes. In the body we iterate through the remaining items, calculate the conditional counters for the input condition extended with that single item, and call the recursion with the new counters and with the original or a new, projected structure, depending on the projection configuration and the percentage of empty nodes. The core recursion is shown as Algorithm 1.

Algorithm 1 Core algorithm

Recursion(*condition*, *nextitem*, *structure*, *counters*):

```

for citem=nextitem-1 downto 0 do
  if support of citem < min_supp then
    continue at next citem
  end if
  newcounters=aggregate conditional pattern base for
  condition  $\cup$  citem
  if projection is beneficial then
    newstructure=projection of structure to newcounters
    Recursion(condition $\cup$ citem, citem, newstructure,
    newcounters)
  else
    Recursion(condition $\cup$ citem, citem, structure, newcounters)
  end if
end for

```

The recursion has four different implementations, that suit differently sized FP trees:

- Very large FP trees that contain millions of nodes are treated by *simultaneous projection*: the tree is traversed once and a projection to each item is calculated simultaneously. This phase is applied only at the first level of recursion; very large trees are expected to arise from sparse databases, like real market basket data; conditional trees projected to a single item are already small in this case.
- *Sparse aggregate* is an aggregation and projection algorithm that does not traverse those part of the tree that will not exist in the next projection. To achieve this, a linked list is built dynamically that contains the indices to non-zero counters. This is similar to the header lists of FP-trees. This aggregation algorithm is used typically near the top of the recursion, where the tree is large and many zeroes are expected. The exact choice is tunable with parameters.
- *Dense aggregate* is the default aggregation algorithm. Each node of the tree is visited exactly once and its

conditional counter is added to the counter of the parent. This is the default aggregation algorithm and it is very fast due to the memory layout of the data structure, described later.

- *Single node optimization* is used near the last levels of recursion, when there is at most one node for each item left in the tree. (This is a slight generalization of the tree being a single chain.) In this case no aggregation and calculation of new counters is needed, so a specialized very simple recursive procedure starts that outputs all subsets of the paths in the tree as a frequent itemset.

The core data structure is a trie. Each node contains a counter and a pointer to the parent. As the trie is never searched, only traversed from the bottom to the top, child maps are not required. The nodes are stored in an array, node pointers are indices to this array.

Nodes that are labelled with the same item occupy a consecutive part of this array, this way we do not need to store the item identifiers in the nodes. Furthermore, we do not need the header lists, as processing all nodes of a specified item requires traversing an interval of this array. This also allows faster execution as only contiguous memory reads are executed. We only need one memory cell per frequent item to store the starting points of these intervals (the *item-starts* array).

The parent pointers (indices) and the counters are stored in separate arrays (*parents* and *counters* resp.) to fit the core algorithm's flexibility: if projection is not beneficial, then the recursion proceeds with the same structural information (parent pointers) but a new set of counters.

The item intervals of the trie are allocated in the array ascending, in topological order. This way the bottom-up and top-down traversal of the trie is possible with a descending resp. ascending iteration through the array of the trie, still only using contiguous memory reads and writes. This order also allows the truncation of the tree to a particular level/item: if the structure is not rebuilt but only a set of conditional counters is calculated for an item, then the recursion can proceed with a smaller sized *newcounters* array and the original *parents* and *itemstarts* array.

The pseudocode for conditional aggregation and projection is shown as Algorithm 2 and 3. Some details are not shown, for example during the aggregation phase we calculate the expected size of the projected structure to allow decision about the projection benefits and to allocate arrays for the projected structure.

Algorithm 2 Aggregation on the compact trie data structure
`cpb-aggregate(item, parents, itemstarts, counters, newcounters, condfreqs)`:

Input: *item* is the identifier of the item to add to the current condition; *parents* and *itemstarts* describe the current structure of the tree; *counters* and *newcounters* hold the current and new conditional counters of the nodes: *counters* is an *itemstarts*[*item*+1] sized array, *newcounters* is an *itemstarts*[*item*] sized array; *condfreqs* will hold the new conditional frequencies of the items. This is the default (dense) aggregation algorithm.

```
fill newcounters and condfreqs with zeroes
for n=itemstarts[item] to itemstarts[item+1]-1 do
    newcounters[parents[n]]=counters[n]
end for
for citem=item-1 downto 0 do
    for n=itemstarts[citem] to itemstarts[citem+1]-1 do
        newcounters[parents[n]]+=newcounters[n]
        condfreqs[citem]+=newcounters[n]
    end for
end for
```

3. Auxiliary routines and optimization: what counts and what doesn't

A very important observation is, that in a first and straightforward implementation of most FIM algorithms the library and auxiliary routines take 70-90% of the running time. Therefore it is essential that these tasks and routines get extra attention, especially in a FIM contest, like the FIMI workshop, where actual running times are measured and every millisecond counts.¹

These auxiliary routines include all C/C++ library calls, memory allocation, input/output implementation, data structure management (including initialization, copy constructors, etc.). Instead of reciting some general “rule of thumbs” we describe our implementation about these issues. The most important issues are posed by those auxiliary routines that appear in the inner recursion, and thus are called proportionally to the core running time.

3.1. Input/Output

The input parsing code released for FIMI'03 is a very well written, low-level implementation, the only relevant change we made to it is that we added a buffer of several megabytes to the input file to avoid OS overhead.

The output routine was completely rewritten. The very slow `fprintf` calls are eliminated, and replaced by proce-

¹This leads to an unfortunate bias: a very good low level programmer implementing a fairly good algorithm can spectacularly defeat a FIM expert implementing the best algorithm on a higher level.

Algorithm 3 Projection on the compact trie data structure
`project(item, parents, itemstarts, newcounters, condfreqs, newparents, newitemstarts, newnewcounters)`:

Input: *newcounters* and *condfreqs* as computed by the aggregation algorithm; *newparents* and *newitemstarts* will hold the projected structure; *newnewcounters* will hold the values of *newcounters* reordered accordingly. The array *newcounters* is reused during the algorithm to store the old position to new position mapping.

```
newcounters[0]=0 /*node 0 is reserved for the root*/
nn=1 /*the next free node index*/
for citem=0 to item-1 do
    newitemstarts[citem]=nn
    for n=itemstarts[citem] to itemstarts[citem+1]-1 do
        if condfreqs[citem]<min_supp OR newcounters[n]==0 then
            newcounters[n]=newcounters[parents[n]] /*skip this node, the new position will be the same as the parent's*/
        else
            newnewcounters[nn]=newcounters[n]
            newcounters[n]=nn /*save the position mapping*/
            newparents[nn]=newcounters[parents[n]] /*retrieve new position of parent from the saved mapping*/
            nn++
        end if
    end for
end for
newitemstarts[item]=nn
```

dures customized for the simple format of FIMI. The most important optimization is that the output routine follows the recursive traversal of the itemset search space, and the text format of the previously outputted itemset is reused for the next output line. The library calls are eliminated or simplified as much as possible (for example outputting a zero-terminated string is approximately 20% slower than outputting a character sequence of a known length due to the additional `strlen`). This optimization is essential for the very low support test cases, where up to gigabytes of output strings are rendered.

The performance comparison of the different output routines is shown on Figure 1. Output was directed to `/dev/null`, the core running time is shown on line *no output*, where no other optimization is employed but to avoid rendering the frequent itemsets to text.

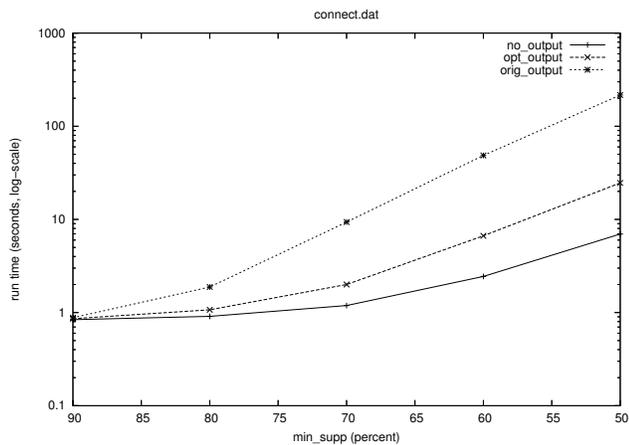


Figure 1. Performance of output routines

3.2. Memory allocation scheme

Another, similarly important point is memory allocation. In each recursive step (the number of which is equal to the total number of frequent itemsets written to output, up to tens of millions) several arrays are allocated for the conditional counters and possibly the projected structure. Calling `malloc` and `free`, or `new` and `delete` incurs a considerable overhead of these library functions due to their general memory management possibilities. Thus it is essential to reuse allocated memory.

The best approach would be to allocate these arrays for each level of the recursion in advance, but as we do not know the required size, and it can be upper bounded only by the size of the full tree, we would run out of main memory.

There are two main observations that lead to our solution to this issue. First, an allocated array can be reused for the same array in any later recursive call. Second, as the recursion proceeds into deeper levels, the required size of arrays decreases monotonically. (This is due to projections and the layout of the trie in the arrays, as discussed earlier.) Thus upon exit from a recursive step we can push the memory arrays on a stack of free blocks, this way the stack will contain decreasingly sized blocks. When entering a new level of recursion we check if the block on top of the stack is large enough. If yes, we can pop the stack and proceed. Otherwise we allocate a new memory block and proceed with an unmodified stack to the next level. Thus the monotonicity remains intact, only the free blocks are shifted one level downwards into the recursion.

In each recursive call we enter the next level several times (depending on the number of remaining items), these require differently sized arrays for the next level of recursion. It is important, that we go from the largest to the smallest when iterating through these possibilities. This way a

larger memory block can be used for the smaller array, otherwise expensive reallocation would be necessary. This was already taken into consideration in Algorithm 1.

Another important factor is when and how to zero the allocated/reused memory blocks. In our first implementation all allocated memory was filled with zero before use; this resulted in up to three times more time spent in the memory fill procedure than the core recursion. This was eliminated by carefully analyzing which arrays need to be filled with zero before use. In some cases it was faster to clean up the array after use than to fill with zero before the next use (in the case when the array is sparsely filled and we have a list of non-zero elements). This way the total amount of memory zeroed was reduced (database `connect.dat`, `min_supp` 50%, 88 million frequent itemsets) from 54 gigabytes (!) to 915 megabytes.

This scheme is implemented as and supplemented by several block-oriented memory allocators. An important side effect of these allocators is, that the initial memory allocated by the program is high (up to 100 MB) even on very small datasets where it is not used completely. This is not a performance issue, the OS should be able to satisfy allocated but otherwise unused memory from swap space, and this parameter may be tuned if the program has to be run on computers with very limited amount of memory.

The effect of memory allocation scheme on running time is shown on Figure 2.

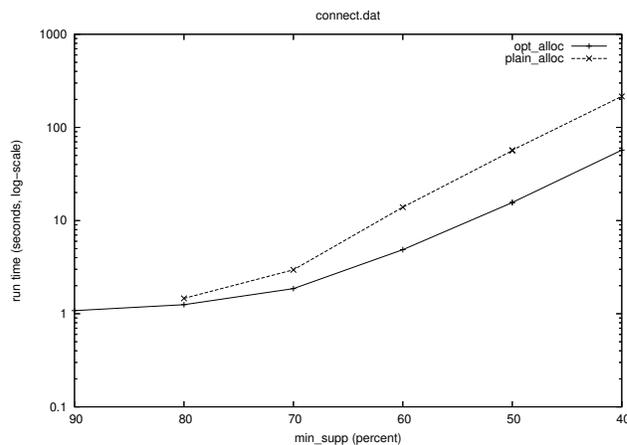


Figure 2. Performance of memory allocation schemes

3.3. Projection configuration

During the conditional pattern base aggregation phase we can easily calculate the expected size of the projected dataset, which will be the number of nodes visited with non-zero `newcounters` value.

Based on this information and the current recursion level we do projection iff the recursion level is smaller than *projlevel*, or the percentage of empty nodes exceeds *projpercent*, where *projlevel* and *projpercent* are tunable parameters of the algorithm. *projlevel* = 0 and *projpercent* = 100 means do not do projection, *projpercent* = 0 means do every projection.

We must note that “projection” here means Algorithm 3, which differs from the original concept of projected database/projected tree as premised in the introduction. Projection here does mean to compact the tree by eliminating infrequent items and nodes that have zero conditional frequency, but it does not reorder the items to continue further recursions with the conditionally most infrequent items, nor does it combine those nodes of the tree that have the same label and the same path to the root (e.g. their respective transactions differ only in conditionally infrequent items).

The effect of some projection configurations on running time is shown on Figure 3. In the line captions the first number is *projlevel*, while the second is *projpercent*. The figure shows that on many databases the projection benefits and projection costs are surprisingly well balanced, thus projection adds little enhancement to the core algorithm.

4. Performance comparison

In this section we evaluate the performance of our implementation. We use publicly available databases *accidents*, *connect*, *pumsb* and *retail* to compare the running time of our implementation to a few competitors (including the best performing ones) of the 1st Frequent Itemset Mining Implementations Workshop, *fpgrowth** [4], *patricia* [6], and *eclat.goethals*.

All of the following tests were run on a 3.0 GHz (FSB800) Intel Pentium 4 processor (hyperthreading disabled) with 2 gigabytes of dual-channel DDR400 main memory and Linux OS. Output was redirected to */dev/null*. The running times of different implementation on the test datasets are displayed on Figure 4.

The figures show that on dense datasets the fast traversal routines take advantage, while on sparse datasets the performance is still competitive.

On sparse datasets the first level of recursion dominates the running time. To achieve better performance for these cases a specialized data structure could be employed in the simultaneous projection phase that adapts better to the skewedness of sparse datasets.

The final submitted version of the implementation (as available in the FIMI repository [1]) uses the following tuning parameters:

- *projlevel* is set to 0 (do not do any projection automatically based on the level of recursion),

- *projpercent* is set to 90% (project the tree if it will shrink to at most tenth of its size),
- *densepercent* is set to 70%, *densesize* is set to 5000 (switch from sparse to dense aggregation algorithm if there is less than 70% empty nodes, or less than 5000 nodes),
- *simultprojthres* to 1 million (do simultaneous projection on the first level of the recursion if the tree size exceeds 1 million nodes).

These values can be set in the beginning of the main program or a run-time configuration file, and should be subject to further, extensive tuning over the parameter space, which was beyond the possibilities and the time-frame available to the author. Also, on different datasets different tuning parameters may give the best performance or memory usage.

5. Conclusion and further work

We described an implementation of a pattern growth-based frequent itemset mining algorithm. We showed a compact, memory efficient representation of an FP-tree that supports the most important requirements of the core algorithm, with a memory layout that allows fast traversal.

The implementation based on this data structure and several further optimizations in the auxiliary routines performs well against some of the best competitors of the 1st Frequent Itemset Mining Implementations Workshop.

The data structure presented here can accommodate the top-down recursion approach, thereby further reducing memory need and computation time.

References

- [1] FIMI repository. <http://fimi.cs.helsinki.fi>.
- [2] B. Goethals. Survey on frequent pattern mining. Technical report, Helsinki Institute for Information Technology, 2003.
- [3] B. Goethals and M. J. Zaki. Advances in frequent itemset mining implementations: Introduction to fimi03. In B. Goethals and M. J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, volume 90 of *CEUR Workshop Proceedings*, Melbourne, Florida, USA, 19. November 2003.
- [4] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, 2003.
- [5] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 1–12. ACM Press, 2000.
- [6] A. Pietracaprina and D. Zandolin. Mining frequent itemsets using patricia tries. In B. Goethals and M. J. Zaki, editors, *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, Melbourne, FL, USA, November 2003.

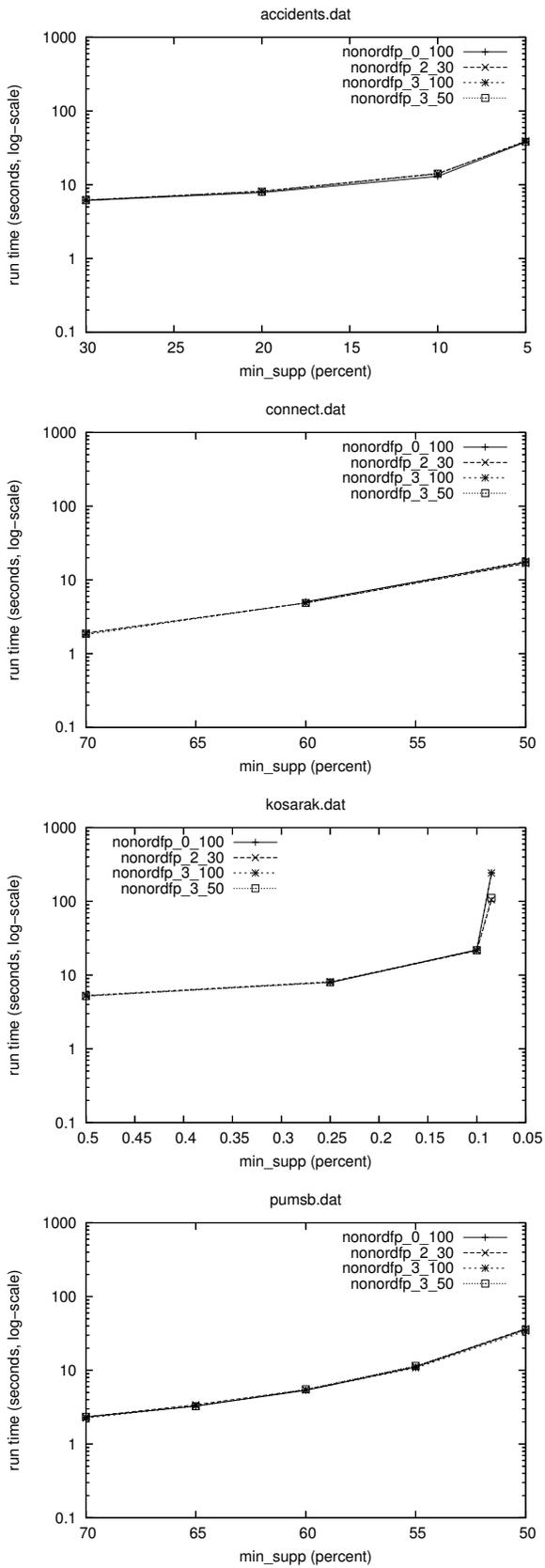


Figure 3. Performance of projection configs

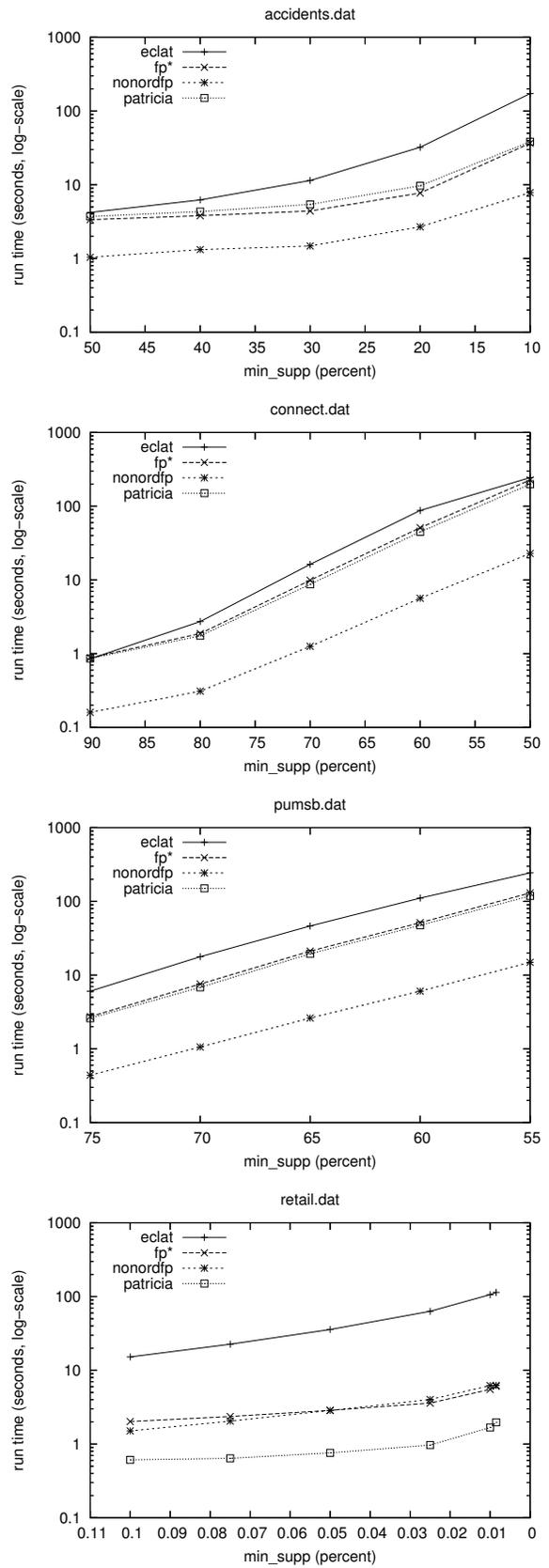


Figure 4. Performance comparison charts