Algorithmic Features of Eclat

Lars Schmidt-Thieme
Computer-based New Media Group (CGNM),
Institute for Computer Science, University of Freiburg, Germany
lst@informatik.uni-freiburg.de

Abstract

Nowadays basic algorithms such as Apriori and Eclat often are conceived as mere textbook examples without much practical applicability: in practice more sophisticated algorithms with better performance have to be used. We would like to challenge that point of view by showing that a carefully assembled implementation of Eclat outperforms the best algorithms known in the field, at least for dense datasets. For that we view Eclat as a basic algorithm and a bundle of optional algorithmic features that are taken partly from other algorithms like lcm and Apriori, partly new ones. We evaluate the performance impact of these different features and report about results of experiments that support our claim of the competitiveness of Eclat.

1. Introduction

Algorithms for mining frequent itemsets often are presented in a monolithic way and labeled with a fancy name for marketing. Careful inspection often reveals similarities with other mining algorithms that allow the transfer from a smart solution of a specific (detail) problem in one algorithm to another one. We would like to go one step further and view such mining algorithms as a basic algorithm and a bundle of algorithmic features.

Basically, there are only two large families of mining algorithms, Apriori [1] and Eclat [10] (counting fpgrowth [5] in the Eclat family what might be arguable). As the basic computation schemes of both of these algorithms are quite simple, one might get the impression, that nowadays they are good only as examples how mining algorithms work in principle for textbooks, but in practice more sophisticated algorithms have to be applied to get good performance results: for example, the four best-performing algorithms of the FIMI-03 workshop, patricia, kdci, lcm, and fpgrowth* (see [7, 6, 9, 8], for the implementations and [3] for a performance evaluation of these algorithms, respectively) do use candidate generation procedures and data structures quite different from those usually associated with the basic algorithms.

In this paper, we would like to challenge that point of view by presenting an Eclat algorithm that for dense datasets outperforms all its more sophisticated competitors.

We will start with a formal outline of Eclat algorithm in section 2. In section 3 we investigate several algorithmic features of Eclat, partly gathered from other algorithms as lcm, fpgrowth, and Apriori, partly new ones, review their usefulness in Eclat and shortly discuss their possible performance impact along with possible reasons thereof. In section 4 we present an empirical evaluation of that impact as well as a comparison with the competitor algorithms from FIMI 03 mentioned above. – We will stick to Eclat. See [2] for an excellent discussion and evaluation of different features of Apriori.

Let us fix notations for the frequent itemset mining problem in the rest of this section. Let A be a set, called **set of items** or **alphabet**. Any subset $X \in \mathcal{P}(A)$ of A is called an **itemset**. Let $\mathcal{T} \subseteq \mathcal{P}(A)$ be a multiset of itemsets, called **transaction database**, and its elements $T \in \mathcal{T}$ called **transactions**. For a given itemset $X \in \mathcal{P}(A)$, the set of transactions that contain X

$$\mathcal{T}(X) := \{T \in \mathcal{T} \mid X \subseteq T\}$$

is called **(transaction) cover of X in $\mathcal{T}$** and its cardinality

$$\sup_{\mathcal{T}}(X) := |\mathcal{T}(X)|$$

(absolute) support of X in $\mathcal{T}$. An **(all) frequent itemset mining task** is specified by a dataset $\mathcal{T}$ and a lower bound $\mathrm{minsup} \in \mathbb{N}$ on support, called **minimum support**, and asks for enumerating all itemsets with support at least minsup, called **frequent** or **(frequent) patterns**.

An itemset X is called **closed**, if

$$X = \bigcap \mathcal{T}(X)$$

i.e., if any super-itemset of X has lower support.

2. Basic Eclat Algorithm

Most frequent itemset mining algorithms as Apriori [1] and Eclat [10] use a total order on the items A of the alphabet and the itemsets $\mathcal{P}(A)$ to prevent that the same itemset, called **candidate**, is checked twice for frequency. Items orderings $\leq$ are in one-to-one-correspondence with **item codings**, i.e., bijective maps $o : A \to \{1, \ldots, n\}$ via natural ordering on $\mathbb{N}$. – For itemsets $X, Y \in \mathcal{P}(A)$ one defines their **prefix** as

$$\text{prefix}(X,Y) := \{\{x \in X \mid x \leq z\} \mid \text{maximal } z \in X \cap Y : \{x \in X \mid x \leq z\} = \{y \in Y \mid y \leq z\}\}$$

Any order on A uniquely determines a total order on $\mathcal{P}(A)$, called **lexicographic order**, by

$$X < Y :\Leftrightarrow \min(X \backslash \text{prefix}(X,Y)) < \min(Y \backslash \text{prefix}(X,Y))$$

For an itemset $X \in \mathcal{P}(A)$ an itemset $Y \in \mathcal{P}(A)$ with $X \subset Y$ and $X < Y$ is called an **extension of** X. An extension Y of X with $Y = X \cup \{y\}$ (and thus $y > \max X$) is called an **1-item-extension of** X. The extension relation organizes all itemsets in a tree, called **extension tree** or **search tree**.

Eclat starts with the empty prefix and the item-transaction incidence matrix $C_\emptyset$, shortly called **incidence matrix** in the following, and stored sparsely as list of item covers: $C_\emptyset := \{(x, \mathcal{T}(\{x\})) \mid x \in A\}$. The incidence matrix is filtered to only contain frequent items by

$$\text{freq}(C) := \{(x, \mathcal{T}_x) \mid (x, \mathcal{T}_x) \in C, |\mathcal{T}_x| \geq \text{minsup}\}.$$

that represent frequent 1-item-extensions of the prefix. For any prefix $p \in \mathcal{P}(A)$ and incidence matrix C of frequent 1-item-extensions of p one can compute the incidence matrix C_x of 1-item-extensions of $p \cup \{x\}$ by intersection rows:

$$C_x := \{(y, \mathcal{T}_x \cap \mathcal{T}_y) \mid (y, \mathcal{T}_y) \in C, y > x\}$$

where $(x, \mathcal{T}_x) \in C$ is the row representing $p \cup \{x\}$. C_x has to be filtered to get all frequent 1-item-extensions of $p \cup \{x\}$ and then this procedure is recursively iterated until the resulting incidence matrix C_x is empty, signaling that there are no further frequent 1-item-extensions of the prefix. See alg. 1 for an exact description of the Eclat algorithm.

Algorithm 1 Basic Eclat algorithm.

input: alphabet A with ordering $\leq$,
multiset $\mathcal{T} \subseteq \mathcal{P}(A)$ of sets of items,
minimum support value $\text{minsup} \in \mathbb{N}$.
output: set F of frequent itemsets and their support counts.
$F := \{(\emptyset, |\mathcal{T}|)\}$.
$C_\emptyset := \{(x, \mathcal{T}(\{x\})) \mid x \in A\}$.
$C'_\emptyset := \text{freq}(C_\emptyset) := \{(x, \mathcal{T}_x) \mid (x, \mathcal{T}_x) \in C_\emptyset, |\mathcal{T}_x| \geq \text{minsup}\}$.
$F := \{\emptyset\}$.
addFrequentSupersets$(\emptyset, C'_\emptyset)$.

function addFrequentSupersets():
input: frequent itemset $p \in \mathcal{P}(A)$ called prefix,
incidence matrix C of frequent 1-item-extensions of p.
output: add all frequent extensions of p to global variable F.
for $(x, \mathcal{T}_x) \in C$ **do**
$q := p \cup \{x\}$.
$C_q := \{(y, \mathcal{T}_x \cap \mathcal{T}_y) \mid (y, \mathcal{T}_y) \in C, y > x\}$.
$C'_q := \text{freq}(C_q) := \{(y, \mathcal{T}_y) \mid (y, \mathcal{T}_y) \in C_q, |\mathcal{T}_y| \geq \text{minsup}\}$.
if $C'_q \neq \emptyset$ **then**
addFrequentSupersets(q, C'_q).
end if
$F := F \cup \{(q, |\mathcal{T}_x|)\}$.
end for

3. Features of Eclat

The formal description of the Eclat algorithm in the last section allows us to point to several algorithmic features that this algorithm may have. These sometimes are described as implementation details, sometimes as extensions of Eclat, and sometimes as new algorithms.

3.1. Transaction Recoding

Before the first incidence matrix $C_\emptyset$ is built, it is usually beneficial 1) to remove infrequent items from the transactions, 2) to recode the items in the transaction database s.t. they are sorted in a specific order, and 3) to sort the transaction in that order. As implementations usually use the natural order on item codes, item recoding affects the order in which candidates are checked. There are several recodings used in the literature and in existing implementations of Eclat and other algorithms as Apriori (see e.g., [2]). The most common codings are coding by increasing frequency and coding by decreasing frequency. For Eclat in most cases recoding items by increasing frequency turns out to give better performance. Increasing frequency means that the length of the rows of the (initial) incidence matrix $C_\emptyset$ grows with increasing index. Let there be f_1 frequent items. As a row at index i is used $f_1 - i$ times at left side (x in the formulas above) and $i - 1$ times at right side (y in the formulas above) of the intersection operator, the order of rows is not important from the point of view of total usage in intersections. But assume the data is gray, i.e., the mining task does not contain any surprising associative patterns, where surprisingness of an itemset X is defined in

terms of lift:

$$\text{lift}(X) := \frac{\sup(X)}{|\mathcal{T}|} / \prod_{x \in X} \frac{\sup(\{x\})}{|\mathcal{T}|} /$$

$\text{lift}(X) = 1$ means that X is found in the data exactly as often as expected from the frequencies of its items, $\text{lift}(X) > 1$ or $\text{lift}(X) < 1$ means that there is an associative or dissociative effect, i.e., it is observed more often or less often than expected. Now, if $\text{lift} \approx 1$ for all or most patterns, as it is typically for benchmark datasets, then the best chances we have to identify a pattern X as infrequent before we actually have counted its support, is to check its subpattern made up from its least frequent items. And that is exactly what recoding by increasing frequency does.

3.2. Types of Incidence Structures: Covers vs. Diffsets

One of the major early improvements of Eclat algorithms has been the replacement of item covers in incidence matrices by their relative complement in its superpattern, so called **diffsets**, see [11]. Instead of keeping track of $\mathcal{T}(q)$ for a pattern q, we keep track of $\mathcal{T}(p) \setminus \mathcal{T}(q)$ for its superpattern p, i.e., $q := p \cup \{x\}$ for an item $x > \max(p)$. $\mathcal{T}(p) \setminus \mathcal{T}(q)$ are those transactions we loose if we extend p to q, i.e., its additional **defect** relative to p. From an incidence matrix C of item covers and one of the 1-item-extensions $(x, \mathcal{T}_x) \in C$ of its prefix we can derive the incidence matrix D of item defects of this extension by

$$D_x := \{(y, T_x \setminus T_y) \,|\, (y, T_y) \in C, y > x\}$$

From an incidence matrix D of item defects and one of its 1-item-extensions $(x, \mathcal{T}_x) \in D$ of its prefix we can derive the incidence matrix D_x of item defects of this extension by

$$D_x := \{(y, T_y \setminus T_x) \,|\, (y, T_y) \in D, y > x\}$$

If we expand first by x and then by y in the second step, we loose transactions that not contain y unless we have lost them before as they did not contain x.

Defects computed from covers may have at most size

$$\text{maxdef}_p := |\mathcal{T}(p)| - \text{minsup},$$

those computed recursively from other defects at most size

$$\text{maxdef}_{p \cup \{x\}} := \text{maxdef}_p - |\mathcal{T}_x|$$

1-item-extensions exceeding that maximal defect are removed by a filter step:

$$\text{freq}(D) := \{(x, \mathcal{T}_x) \,|\, (x, \mathcal{T}_x) \in C, |\mathcal{T}_x| \leq \text{maxdef}\}.$$

Computing intersections of covers or set differences for defects are computationally equivalent complex tasks. Thus, the usage of defects can improve performance only by leading to smaller incidence matrices. For dense datasets where covers overlap considerably, intersection reduces the size of the incidence matrix only slowly, while defects cut down considerably. On the other side, for sparse data using defects may deteriorate the performance. – Common items in covers also can be removed by omitting equisupport extensions (see section 3.5).

While there is an efficient transition from covers to defects as given by the formula above, the reverse transition from defects to covers seems hard to perform efficiently as all defects on the path to the root of the search tree would have to be accumulated.

Regardless which type of incidence matrix is used, it can be stored as sparse matrix (i.e., as list of lists as discussed so far) or as dense (bit)matrix (used e.g, by [2]).

A third alternative for keeping track of item-transaction incidences is not to store item covers as a set of incident transaction IDs per 1-item-extension, but to store all transactions $\mathcal{T}(p)$ that contain a given prefix p in a trie (plus some index structure, known as frequent pattern tree and first used in fp-growth; see [5]). Due to time restrictions, we will not pursue this alternative further here.

3.3. Incidence Matrix Derivation

For both incidence matrices, covers and defects, two different ways of computing the operator that derives an incidence matrix from a given incidence matrix recursively, i.e., intersection and set difference, respectively, can be chosen. The straightforward way is to implement both operators as set operators operating on the sets of transaction IDs.

Alternatively, intersection and difference of several sets $\mathcal{T}_y, y > x$ of transactions by another set $\mathcal{T}_x$ of transactions also can be computed in parallel using the original transaction database by counting in IDs of matching transactions (called occurrence deliver in [9]). To compute $\mathcal{T}'_y := \mathcal{T}_y \cap \mathcal{T}_x$ for several $y > x$ one computes

$$\forall T \in \mathcal{T}_x \forall y \in T : \mathcal{T}'_y := \mathcal{T}'_y \cup \{T\}.$$

Similar, to compute $\mathcal{T}'_y := \mathcal{T}_x \setminus \mathcal{T}_y$ for several $y > x$ one computes

$$\forall T \in \mathcal{T}_x \forall y \notin T : \mathcal{T}'_y := \mathcal{T}'_y \cup \{T\}.$$

3.4. Initial Incidence Matrix

Basic Eclat first builds the incidence matrix $C_\emptyset$ of single item covers as initial incidence matrix and then recursively derives incidence matrices C_p of covers of increasing prefixes p or D_p of defects.

Obviously, one also can start with $D_\emptyset$, the matrix of item cover complements. This seems only useful for very dense

datasets as it basically inverts the encoding of item occurrence and non-occurrence (dualization).

It seems more interesting to start already with incidence matrices for 1-item-prefixes, i.e., not to use Eclat computation schemes for the computation of frequent pairs, but count them directly from the transaction data. For Apriori this is a standard procedure. The cover incidence matrix $C_x = \{(y, \mathcal{T}_y)\}$ for an frequent item x, i.e., $\mathcal{T}_y = \mathcal{T}(\{x\}) \cap \mathcal{T}(\{y\})$, is computed as follows:

$$\forall T \in \mathcal{T} : \text{if } x \in T : \forall y \in T, y > x : \mathcal{T}_y := \mathcal{T}_y \cup \{T\}.$$

The test for $x \in T$ looks worse than it is in practice: if transactions are sorted, items x are processed in increasing order, and deleted from the transaction database after computation of C_x, then if x is contained in a transaction T it has to be its first item.

Similarly, a defect incidence matrix $D_x = \{(y, \mathcal{T}_y)\}$ for a frequent item x, i.e., $\mathcal{T}_y = \mathcal{T}(\{x\}) \setminus \mathcal{T}(\{y\})$, can be computed directly from the transaction database by

$$\forall T \in \mathcal{T} : \text{if } x \in T : \forall y \notin T, y > x : \mathcal{T}_y := \mathcal{T}_y \cup \{T\}.$$

If C_x or D_x is computed directly from the transaction database, then it has to be filtered afterwards to remove infrequent extensions. An additional pass over $\mathcal{T}$ in advance can count pair frequencies for all x, y in parallel, so that unnecessary creation of covers or defects of infrequent extensions can be avoided.

3.5. Omission of Equisupport Extensions

Whenever an extension x has the same support as its prefix p, it is contained in the closure $\bigcap \mathcal{T}(p)$ of the prefix. That means that one can add any such equisupport extension to any extension of p without changing its support; thus, one can omit to explicitly check its extensions. Equisupport extensions can be filtered out and kept in a separate list E for the active branch: whenever an itemset X is output, all its $2^{|E|}$ supersets $X' \subseteq X \cup E$ are also output.

Omission of equisupport extensions is extremely cheap to implement as it can be included in the filtering step that has to check support values anyway. For dense datasets with many equisupport extensions, the number of candidates that have to be checked and accordingly the runtime can be reduced drastically.

3.6. Interleaving Incidence Matrix Computation and Filtering

When the intersection $\mathcal{T}_x \cap T_y$ of two sets of transaction IDs is computed, we are interested in the result of this computation only if it is at least of size minsup, as otherwise it is filtered out in the next step. As the sets of transactions are sorted, intersections are computed by iterating over the lists of transaction IDs and comparing items. Once one of the tails of the lists to intersect is shorter than minsup minus the length of the intersection so far, we can stop and drop that candidate, as it never can become frequent. – For set difference of maximal length maxdef a completely analogous procedure can be used.

3.7. Omission of Final Incidence Matrix Derivation

Finally, once the incidence matrix has only two rows, the result of the next incidence matrix derivation will be an incidence matrix with a single row. As this is only checked for frequency, but its items are not used any further, we can omit to generate the list of transaction IDs and just count its length.

3.8. IO

So far we have investigated features that are specific to Eclat and the frequent itemset mining problem. Though these specific algorithmic features are what should be of primary interest, we noticed in our experiments, that often different IO mechanism dominate runtime behavior. At least three output schemes are implemented in several of the algorithms available: IO using C++ streams, IO using `printf`, and IO using handcrafted rendering of integer itemsets to a char buffer and writing that buffer to files using low-level `fwrite` (for the latter see e.g., the implementation of lcm, [9]). Handcrafted rendering of itemsets to char buffers is by far the fastest method; especially for low support values, when huge numbers of patterns are output, the runtime penalty from slower output mechanisms cannot be compensated by better mining mechanisms whatsoever.

4. Evaluation

By evaluating different features of Eclat we wanted to answer two questions:

1. What features will make Eclat run fastest? Especially, what is its marginal runtime improvement of each feature in a sophisticated Eclat implementation?

2. Is Eclat competitive compared with more complex algorithms?

To answer the question about the runtime improvement of the different features, we implemented a modular version of Eclat in C++ (basically mostly plain C) that allows the flexible inclusion or exclusion of different algorithmic features. At the time of writing the following features are implemented: the incidence structure types covers and diffsets (COV, DIFF), transaction recoding (none, decreasing,

increasing; NREC, RECDEC, RECINC), omission of equisupport extensions (NEE), interleaving incidence matrix computation and filtering (IFILT), and omission of final incidence matrix (NFIN). As initial incidence matrix alway covers of frequent 1-itemsets ($C_\emptyset$) was used.

To measure the marginal runtime improvement of a feature we configured a sophisticated Eclat algorithm with all features turned on (SOPH:= DIFF, RECINC, NEE+, IFILT+, NFIN+) and additionally for each feature an Eclat algorithm derived from SOPH by omitting this feature (SOPH-DIFF, SOPH-RECINC (decreasing encoding), SOPH-REC (no recoding at all), SOPH-NEE+, SOPH-IFILT+, SOPH-NFIN+).

We used several of the data sets and mining tasks that have been used in the FIMI-03 workshop ([4]): accidents, chess, connect, kosarak, mushroom, pumsb, pumsbstar, retail, T10I5N1KP5KC0.25D200K, T20I10N1KP5KC0.25D200K, and T30I15N1KP5KC0.25-D200K. All experiments are ran on a standard Linux box (P4/2MHz, 1.5GB RAM, SuSE 9.0). Jobs were killed if they run more than 1000 seconds and the corresponding datapoint is missing in the charts.

A sample from the results of these experiments can be seen in fig. 1 (the remaining charts can be found at http://www.informatik.uni-freiburg.de/cgnm/papers/-fimi04). One can see some common behavior across datasets and mining tasks:

- For dense mining tasks like accidents, chess, etc. SOPH is the best configuration.
- For sparse mining tasks like retail, T20I10N1KP5KC0-25D200K etc. SOPH-diff is the best configuration, i.e., using defects harms performance here – both effects are rather distinct.
- Recoding is important and shows a huge variety w.r.t. runtime: compare e.g., decreasing and no encoding for connect: the natural encoding is not much worse than decreasing encoding, but the curve for increasing encoding shows what harm the wrong encoding can do: note that the natural encoding is close to optimal only by mere chance and could be anywhere between increasing and decreasing!
- Omitting equisupport extensions also shows a clear benefit for most mining tasks, with exception for mushroom.
- Compared with other features, the impact of the features IFILT and NFIN is neglectible.

To answer the second question about competitiveness of Eclat compared with more advanced frequent pattern mining algorithms we have chosen the four best-performing algorithms from the FIMI-03 workshop: patricia, kdci, lcm, and fpgrowth* (see [7, 6, 9, 8], for the implementations and [3] for a performance evaluation of these algorithms, respectively).

Again, a sample from the results of these experiments can be seen in fig. 2 (the remaining charts also can be found at http://www.informatik.uni-freiburg.de/cgnm/-papers/fimi04). For several datasets (chess, connect, mushroom, pumsb, and – not shown – pumsbstar), Eclat-SOPH is faster than all other algorithms. For some datasets it is faster for high minimum support values, but beaten by fpgrowth* when support values get smaller (accidents, T30I15N1KP5KC0-25D200K) and for some datasets its performance is really poor (retail, T20I10N1KP5KC0-25D200K, and – not shown – kosarak and T10I5N1KP5KC0.25D200K). We can draw two conclusions from this observations: 1) at least for dense datasets, Eclat-SOPH is faster than all its competitors, 2) for sparse datasets, Eclat-SOPH is not suitable. Recalling our discussion on the potential of using defects instead of covers and on starting with frequent 2-itemsets instead of with frequent 1-itemsets, the latter conclusion is not very surprising.

5. Outlook

There are at least four more features we do not have investigated yet: using tries to store the transaction covers, the method to compute the initial incidence matrix, pruning, and memory management. Our further research will try to address questions about the impact of these features.

This update of optimization for dense datasets has to be complemented with research in performance drivers for sparse datasets. As can be seen from our results, Eclat seems not suited well for that task. Though using covers instead of defects improves performance, it still is not competitive with other algorithms in the field.

Furthermore, results for dense datasets will have to be compared with that of the next generation of mining algorithms we expect as outcome of FIMI'04 and eventually new features of these algorithms have to be integrated in Eclat. We expect both, that Eclat is clearly beaten at FIMI'04 as well that it will be not too hard to identify the relevant features and integrate them in Eclat.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94), Santiago de Chile, September 12-15*, pages 487–499. Morgan Kaufmann, 1994.

[2] C. Borgelt. Efficient implementations of apriori and eclat. In Goethals and Zaki [4].

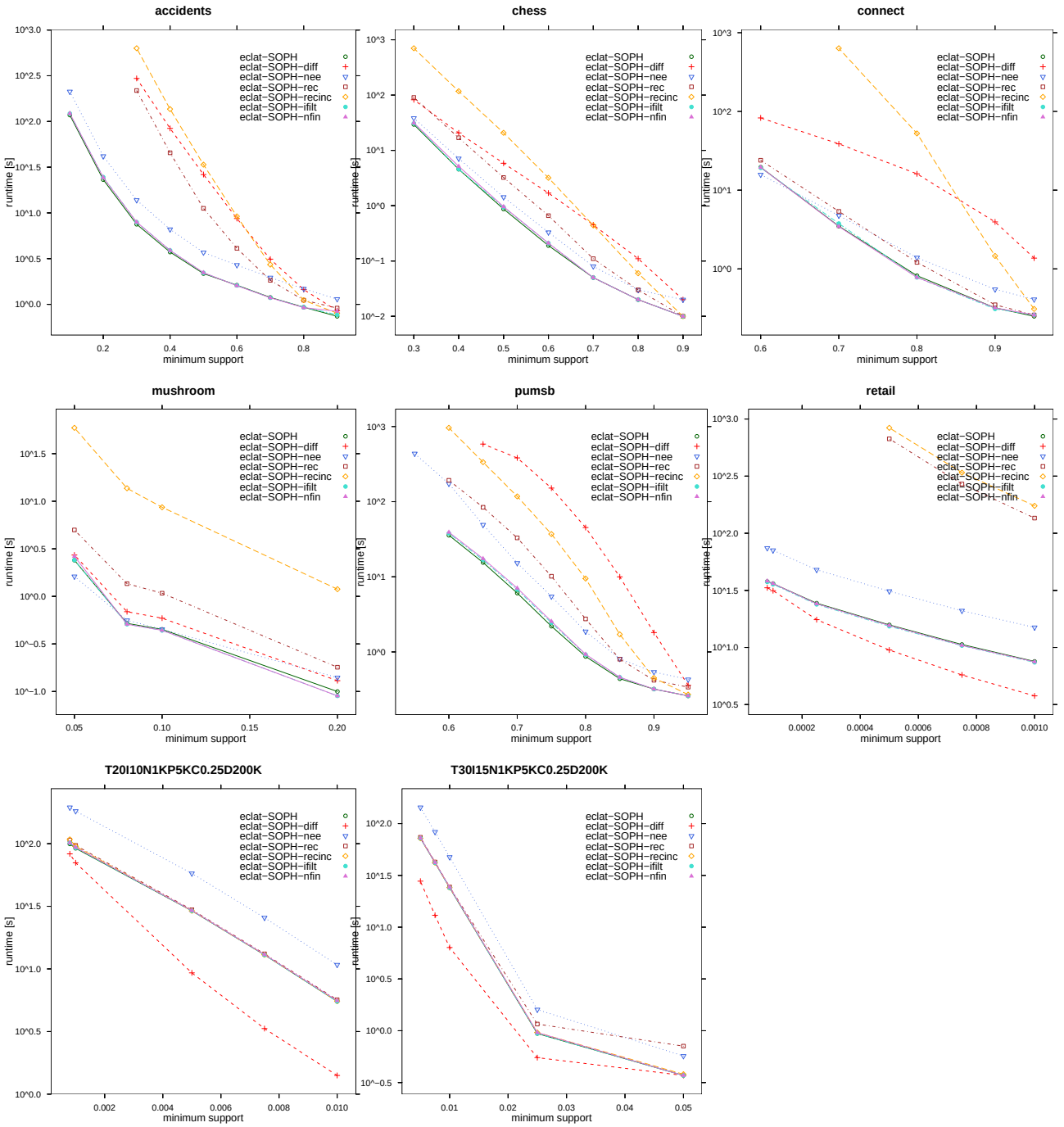

Figure 1. Evaluation of the marginal effect of different features of Eclat on runtime.

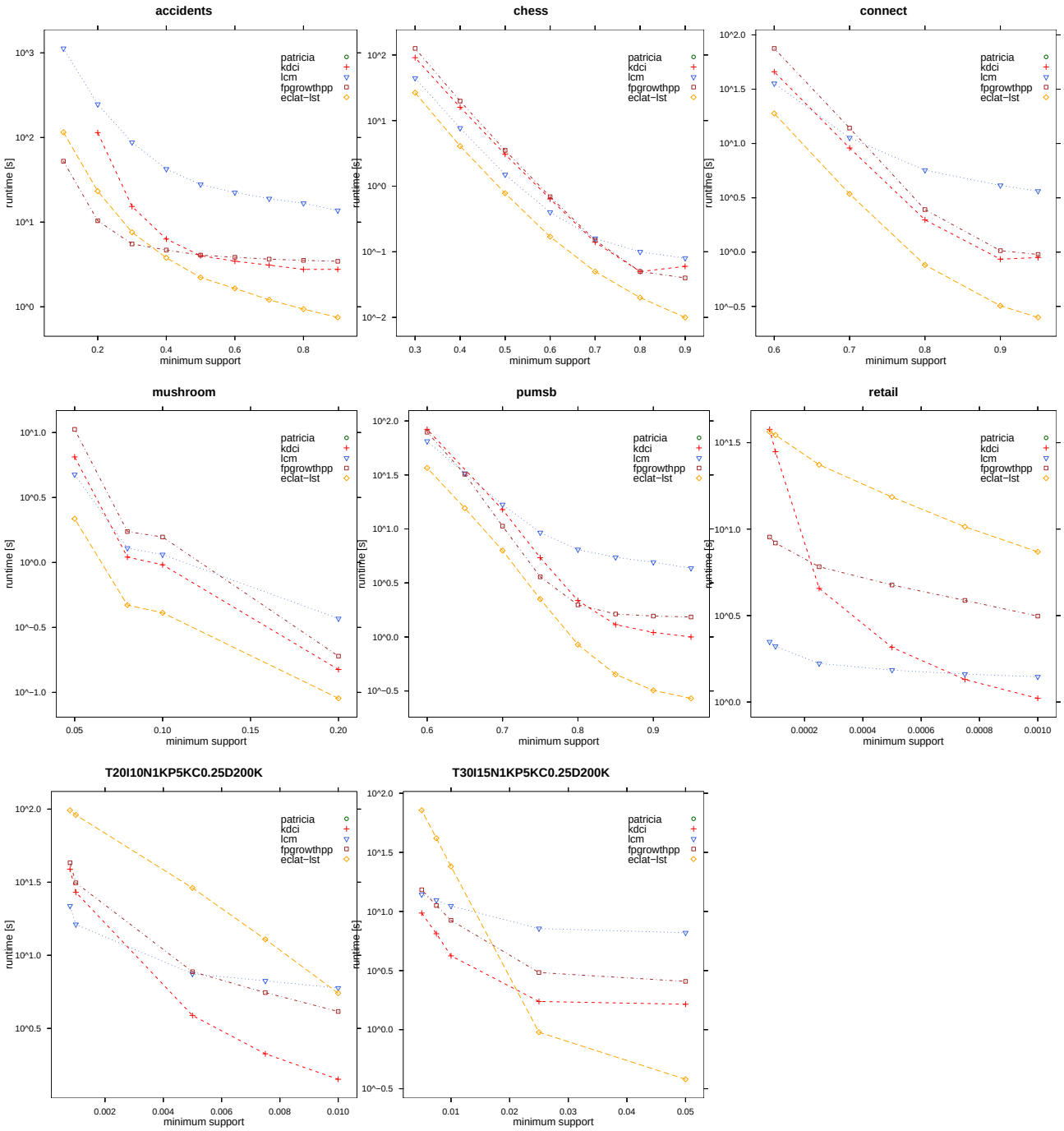

Figure 2. Evaluation of Eclat-SOPH (= eclat-lst) vs. fastest algorithms of the FIMI-03 workshop.

[3] B. Goethals and M. J. Zaki. Advances in frequent itemset mining implementations: Introduction to fimi03. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, Melbourne, Florida, USA, November 19* [4].

[4] B. Goethals and M. J. Zaki, editors. *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, Melbourne, Florida, USA, November 19, 2003*. 2003.

[5] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In W. Chen, J. F. Naughton, and P. A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM Press, 2000.

[6] S. Orlando, C. Lucchese, P. Palmerini, R. Perego, and F. Silvestri. kdci: a multi-strategy algorithm for mining frequent sets. In Goethals and Zaki [4].

[7] A. Pietracaprina and D. Zandolin. Mining frequent itemsets using patricia tries. In Goethals and Zaki [4].

[8] G. sta Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In Goethals and Zaki [4].

[9] T. Uno, T. Asai, Y. Uchida, and H. Arimura. Lcm: An efficient algorithm for enumerating frequent closed item sets. In Goethals and Zaki [4].

[10] M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372–390, 2000.

[11] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. Technical report, RPI, 2001. Tech. Report. 01-1.