Analysis and Comparison of frequent item-set mining algorithms variants

Cyril Bousmar - 63401800
Louvain School of Engineering
cyril.bousmar@uclouvain.be

Group 23

Mohamed-Anass Gallass - 02652201
Louvain School of Engineering
anass.gallass@student.uclouvain.be

***Abstract*—This report presents an analysis and comparison of frequent item-set mining algorithms, focusing on the *Apriori* and *ECLAT* approaches. We implemented three variants: *Apriori Naive*, *Apriori Pruning*, and *ECLAT*, and evaluated their performance across multiple datasets with varying densities. The experiments assess computational efficiency in terms of execution time and memory consumption under different minimum support thresholds. Our findings highlight the strengths and weaknesses of each algorithm, demonstrating that *ECLAT* generally outperforms *Apriori* in dense datasets, while Apriori Pruning offers significant improvements over its naive counterpart. The full implementation and dataset processing scripts are available on the github project repository.**

I. Introduction

In the context of the course LINFO2364: Mining Patterns in Data, we were tasked to implement frequent item-set mining [1] algorithms to extract patterns from transactional datasets given a fixed minimum support threshold. More specifically, variants of a *Breath First Search (BFS)* approach, *Apriori* [2], and of a *Depth First Search (DFS)* one, *ECLAT* [3]. Explored variants are: *Apriori Naive*, *Apriori Pruning*, and *ECLAT*.

This report is structured as follows. Section II describes algorithms implementation details, highlighting key design choices and optimizations. Section III presents the methodology used for performance analysis, detailing the datasets, evaluation metrics, and experimental setup. Section IV showcases the results obtained, including a comparative analysis of execution time and efficiency across different datasets and support values. Section V discusses key findings, and the impact of algorithmic choices on performance. Finally, Section VI summarizes our conclusions and suggests potential improvements for future work.

II. Algorithms Implementation

A. Notations

A dataset $\mathcal{D}$, has transactions $\mathcal{T}$ and a set $\mathcal{I}$ of literals called items. Its minimum support threshold is noted θ.

TABLE I: Notations used for different sets.

Notation	Descriptions
i-item-set	An item-set having i-items.
i-subset	A subset of a $(i+1)$-item-set.
C_i	Set of candidate i-item-sets.
F_i	Set of frequent i-item-sets and their support.
$\mathcal{F}$	Set of all frequent item-sets and their support.

Functions 1, 2, and 3 respectively compute the match, the cover, and the support of an item-set.

$$\mathit{match}(I,T) = \begin{cases} 1, & \text{if } I \subseteq T \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

$$\mathit{cover}_{\mathcal{D}}(I) = \{t \in \mathcal{T} \mid \mathit{match}(I, \mathcal{D}(t)) = 1\} \tag{2}$$

$$\mathit{support}_{\mathcal{D}}(I) = |\mathit{cover}_{\mathcal{D}}(I)| \tag{3}$$

An itemset I of a dataset $\mathcal{D}$ is frequent *iff* $\mathit{support}_{\mathcal{D}}(I) \geq \theta$.

B. Apriori Naive algorithm

The algorithm 1 is the nearly the most basic approach to item-sets mining with a *BFS* method, for which nearly all possible candidates are generated. In fact, only $(i+1)$item-sets for which i-subsets where candidates at level i are considered for level $i+1$. This is already a form of pruning as not all possible combination with permutation of items at each level i are taken into account.

Algorithm 1 Apriori Naive

```
Require: $filepath$, $minFrequency$
Ensure: All frequent item-sets, $\mathcal{F}$, from transactions, $\mathcal{T}$
 1: Load $\mathcal{T}$ from $filepath$ and compute $|\mathcal{T}|$
 2: $\theta \leftarrow minFrequency \times |\mathcal{T}|$
 3: $F_{i=0} \leftarrow \emptyset$, $\mathcal{F} \leftarrow \emptyset$, $i \leftarrow 0$
 4: repeat
 5:     $\mathcal{F} \leftarrow \mathcal{F} \cup \{(I, \mathit{support}_{\mathcal{D}}(I)) \mid I \in F_i\}$
 6:     for all $i$-subsets $s_i$ of $(i+1)$-item-set $I$ do
 7:         if all $s_i \in C_i$ then
 8:             add $I$ to $C_{i+1}$
 9:         end if
10:     end for
11:     for all $T \in \mathcal{T}$ do
12:         for all candidates $I \in C_{i+1}$ do
13:             $I.count \leftarrow I.count + \mathit{match(I, T)}$
14:         end for
15:     end for
16:     $F_{i+1} \leftarrow \{(I, S) \mid S = \mathit{support}_{\mathcal{D}}(I) \geq \theta\ ,\ I \in C_{i+1}\}$
17:     $i \leftarrow i + 1$
18: until $F_i \neq \emptyset$
```

C. Apriori Pruning algorithm

The optimization for algorithm 2 is made from a subtle change at line 7, where are kept only candidates for C_{i+1} for which all of their i-subsets are frequent item-sets. This prevent from generating many candidate for which we already know their frequency is not above the threshold.

Algorithm 2 Apriori Pruning

```
Require: $filepath$, $minFrequency$
Ensure: All frequent item-sets, $\mathcal{F}$, from transactions, $\mathcal{T}$
 1: Load $\mathcal{T}$ from $filepath$ and compute $|\mathcal{T}|$
 2: $\theta \leftarrow minFrequency \times |\mathcal{T}|$
 3: $F_{i=0} \leftarrow \emptyset$, $\mathcal{F} \leftarrow \emptyset$, $i \leftarrow 0$
 4: repeat
 5:     $\mathcal{F} \leftarrow \mathcal{F} \cup \{(I, \text{support}_{\mathcal{D}}(I)) \mid I \in F_i\}$
 6:     for all $i$-subsets $s_i$ of $(i+1)$-item-set $I$ do
 7:         if all $s_i \in F_i$ then
 8:             add $I$ to $C_{i+1}$
 9:         end if
10:     end for
11:     for all $T \in \mathcal{T}$ do
12:         for all candidates $I \in C_{i+1}$ do
13:             $I.count \leftarrow I.count + match(I, T)$
14:         end for
15:     end for
16:     $F_{i+1} \leftarrow \{(I, S) \mid S = support_{\mathcal{D}}(I) \geq \theta \ , I \in C_{i+1}\}$
17:     $i \leftarrow i + 1$
18: until $F_i \neq \emptyset$
```

D. Eclat algorithm

The ECLAT algorithm 3 follows a Depth First Search (DFS) approach, but instead of storing all the items in each transaction (the usual "horizontal" way), it keeps a vertical list of TIDs (Transaction IDs) for each item. To figure out how often two items occur together, the algorithm intersects the TID sets of these items and counts how many TIDs are left. If this count is high enough (above the support threshold), ECLAT then recursively tries to add more items, using further intersections, until no more frequent combinations can be found.

III. Methodology

A. Datasets

This section briefly describes all datasets used to analyze and compare the different implementations of algorithms. All have their own specificity which impact algorithms' performance in different ways. Therefore, this allows us to have better insight on which dataset each algorithm is expected to perform better.

In this context, we define the density of the dataset, $\mathcal{D}$ as:

$$\rho = \frac{n}{|\mathcal{I}| \times |\mathcal{T}|}$$

where n is the total number of recorded entries.

Algorithm 3 Eclat

```
Require: $filepath$, $minFrequency$
Ensure: All frequent item-sets, $\mathcal{F}$, from transactions $\mathcal{T}$
 1: Load $\mathcal{T}$ from $filepath$ and compute $|\mathcal{T}|$
 2: $\theta \leftarrow minFrequency \times |\mathcal{T}|$
 3: $verticalDB \leftarrow \{\ i \mapsto TIDs(i)\ \}$ ▷ Build vertical database:
    for each item $i$, store its TIDs
 4: function ECLAT($P$, $TIDs_P$, $items$)
 5:     for all item $i$ in $items$ do
 6:         $TIDs_i \leftarrow verticalDB[\,i\,]$
 7:         $intersectTIDs \leftarrow TIDs_P \cap TIDs_i$
 8:         if $|intersectTIDs| \geq \theta$ then
 9:             $P' \leftarrow P \cup \{i\}$
10:             $\mathcal{F} \leftarrow \mathcal{F} \cup \{(P', |intersectTIDs|)\}$     ▷ Extend
    search with remaining items
11:             ECLAT($P'$, $intersectTIDs$, $items\ after\ i$)
12:         end if
13:     end for
14: end function
15: $\mathcal{F} \leftarrow \emptyset$     ▷ Initialize DFS with an empty prefix and all
    TIDs
16: $allItems \leftarrow \{$all distinct items in $\mathcal{T}\}$
17: ECLAT( $\emptyset, \{$all TIDs$\}, allItems$ )
```

This is especially useful because this determines the number of considered candidates for the algorithms to consider, which vary by their nature (i.e. *BFS* or *DFS*).

TABLE II: Dataset[1]Statistics: Number of Recorded Entries n, Number of $\mathcal{T}$ Transactions $|\mathcal{T}|$, and Density ρ

$\mathcal{D}$	n	$\lvert\mathcal{T}\rvert$	ρ
accidents [4]	468	340,183	0.0722
chess	75	3,196	0.4933
connect	129	67,557	0.3333
mushroom	119	8,124	0.1933
pumsb	2,113	49,046	0.035
retail [5]	16,470	88,162	0.0006

Accidents will likely consume more memory considering the number of transactions.

Apriori is expected to perform better on sparse datasets that dense ones, while *ECLAT* is expected to do it inversely.

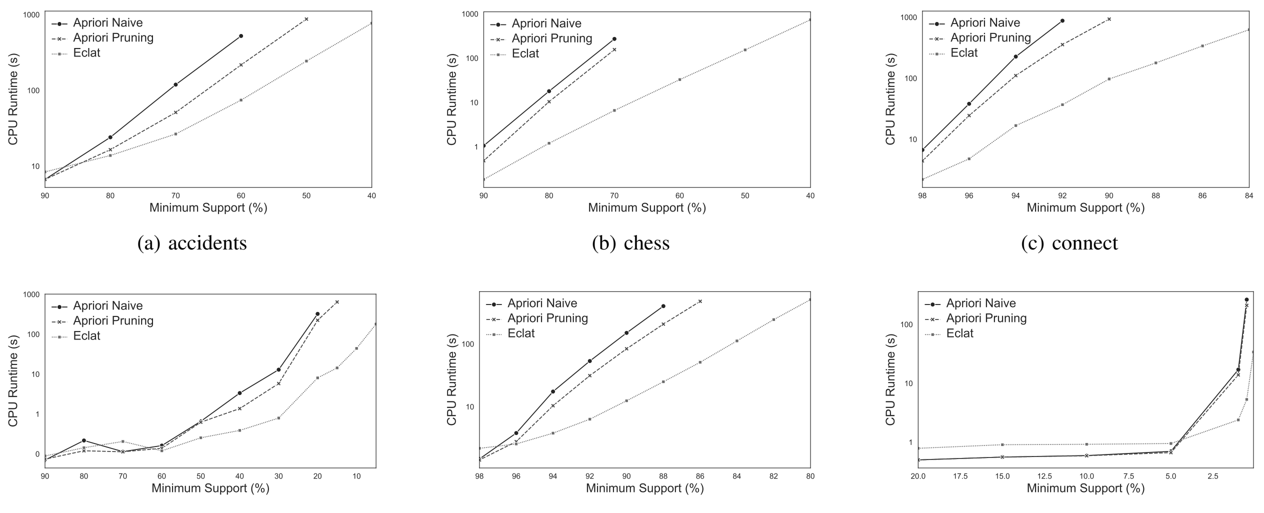
B. Metrics

We compare the performance of algorithms over datasets based on the *average CPU runtime* and the *average maximum memory consumption*. It is studied across a wide variety of *minimum support threshold* values to estimate their break-points.

C. Setup

Experiment was run on an *Intel Core i5-8265U @1.6GHz* with 8 GB of RAM. Each test is run 5 times for outliers

[1]Datasets *chess*, *connect*, *mushroom* and *pumsb* prepared by Roberto Bayardo from the UCI datasets and PUMSB for FIMI'03 and FIMI'04 of IEEE ICDM'03 and IEEE ICDM'04.



(a) accidents (b) chess (c) connect

(d) mushroom (e) pumsb (f) retail

Fig. 1: CPU runtime of *Apriori Naive*, *Apriori Pruning* and *ECLAT* across all datasets for different minimum support threshold. Experiment launched on datasets a) accidents, b) chess, c) connect, d) mushroom, e) pumsb, f) retail.

mitigation, while allowing a maximum CPU runtime of 1000 seconds before signaling a timeout. Exceeding memory allocation times out the test as well.

IV. Results

A. CPU runtime

Figure 1 shows that Apriori algorithms 1 and 2 take longer to complete in almost all cases compare to the Eclat algorithm, 3. The only exception is when the minimum support threshold is very high i.e. in datasets :

- *accidents* (Fig. 1a) with minimum support higher than 80%.
- *mushroom* (Fig. 1d) with minimum support higher than 60%.
- *pumsb* (Fig. 1e) with minimum support higher than 96% and,
- *retail* (Fig. 1f) with minimum support higher than 5%.

We also see that *Apriori Pruning* (2) is generally much faster than *Apriori Naive* (1) with a gap between both that increase as the minimum support goes lower, confirming the benefit of pruning candidates early.

B. Memory Consumption

Figure 2 compares memory usage. From the memory measurements, we expected Eclat to consume more memory than both Apriori algorithms in most cases, but our observations show a few surprises:

- In the *accidents* dataset at a high support threshold (70%), ECLAT actually used around 450MB, whereas both Apriori implementations Naive and Pruning went up to about 700 MB. This outcome was unexpected, since we assumed Eclat's vertical TID storage would naturally increase memory usage.
- In the *retail* dataset at very low support (5%), Eclat's memory consumption grew sharply, while the Apriori algorithms remained relatively stable. A likely reason is the extremely low density (0.0006) of retail.

Overall, apart from these cases, memory usage was not very different across the three algorithms, even though Eclat and Apriori use different data representations.

Note that for too small minimum support threshold values, it was observed that the *DFS* algorithm variant 3 exceeds the maximum depth allowed by python memory and can not conclude.

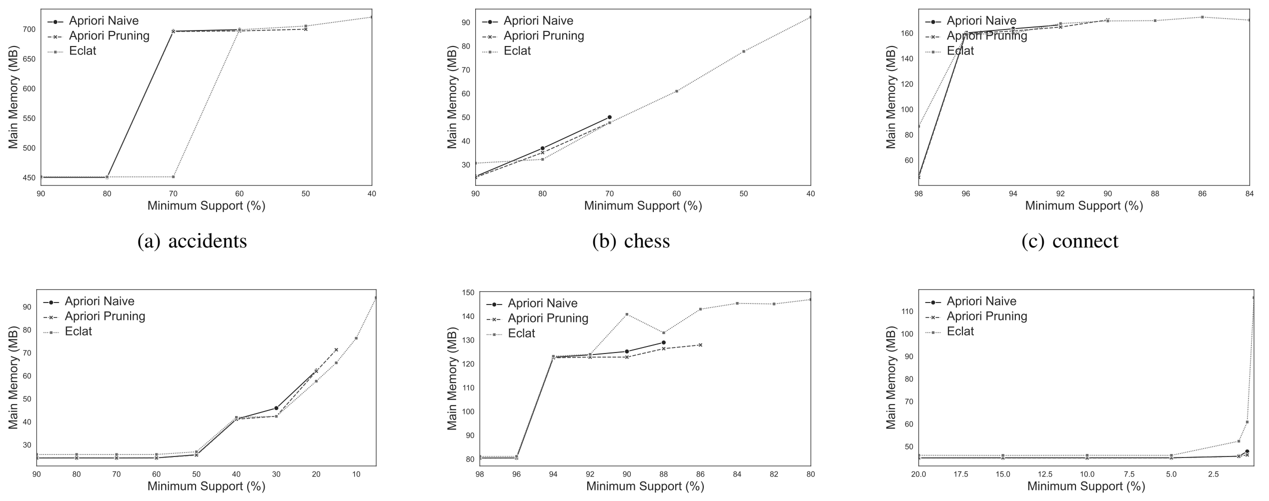
V. Discussion

Surprisingly, the anticipated memory advantage of *Apriori* algorithms over *ECLAT* is rarely observed. This is primarily due to the fact that the implementations 1 and 2 time out before completion, as they do not represent the state-of-the-art in their category. This highlights the potential for improvement, such as leveraging trie structures to optimize memory usage.

The only notable memory advantage of *Apriori* appears in extremely sparse datasets, such as *retail*. Beyond a certain inflection point, the *ECLAT* algorithm requires nearly twice as much memory to compute results, while the *Apriori* implementations continue running without timing out.

Conversely, in transaction-heavy datasets like *accident*, which contains 10 to 100 times more transactions than the others, *Apriori* algorithms consume memory at a much faster rate than ECLAT. This likely stems from their need to iterate over all relevant transactions.

As expected, *Apriori* performs better on sparse datasets, whereas ECLAT excels on denser ones.



(a) accidents (b) chess (c) connect

(d) mushroom (e) pumsb (f) retail

Fig. 2: Maximum main memory consumption of *Apriori Naive*, *Apriori Pruning* and *ECLAT* across all datasets for different minimum support threshold. Experiment launched on datasets a) accidents, b) chess, c) connect, d) mushroom, e) pumsb, f) retail.

VI. Conclusion

In summary, all three algorithms—*Apriori Naive*, *Apriori Pruning*, and *ECLAT*—correctly produce frequent itemsets. However, their performance differs greatly depending on the dataset's density and the minimum support threshold:

- **ECLAT** is typically the fastest, but can encounter large TID lists in certain datasets or deep recursion issues at very low thresholds.
- **Apriori Pruning** consistently outperforms Apriori Naive by removing clearly non-frequent candidates ahead of time.
- **Apriori Naive** performs the slowest in most cases, although it is simpler to implement.

Future work, could integrate a more refined version of Apriori algorithm using tree and also consider alternative algorithms like *FP-Growth* [6], which might further improve execution time and memory usage.

References

[1] R. Agrawal, T. Imielinski and A. Swami. Database mining: a performance perspective. In *IEEE Transactions on Knowledge and Data Engineering*, volume 5, number 6, pages 914-925, 1993. IEEE.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, San Francisco, CA, USA, pages 487-499, 1994. Morgan Kaufmann Publishers Inc.

[3] M. J. Zaki. Scalable algorithms for association mining. In *IEEE Transactions on Knowledge and Data Engineering*, volume 12, number 3, pages 372–390, 2000. IEEE.

[4] K. Geurts, G. Wets, T. Brijs and K. Vanhoof. Profiling high frequency accident locations using association rules. In *Proceedings of the 82nd Annual Transportation Research Board*, Washington, DC, USA, January 12–16, 2003.

[5] T. Brijs, G. Swinnen, K. Vanhoof and G. Wets. Using association rules for product assortment decisions: A case study. In *Proceedings of the Fifth International Conference on Knowledge Discovery and Data Mining*, San Diego, CA, USA, pages 254–260, 1999.

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