Pattern Set Mining

Final essay

Athanasios Tsiamis

5223652

Graduate School of Natural Sciences
Utrecht University
The Netherlands
June 2022

1 Introduction

Patterns are so simple yet so complicated concepts. Their existence is everywhere dubbed with different names. In nature a pattern may be referred to as symmetry, in mathematics as fractals while in zoology as stripes. However, patterns are not always visible or easily discoverable. There are many cases in which patterns are well hidden either due to the nature of the problem (e.g. turbulence dissipation in partial differential equations) or the amount of data is so big, that they are easily lost in it.

The data mining community has made systematic attempts to formalize and solve the problem of finding patterns or, as Fischer and Vreeken stated [11], "easily interpretable symbolic statements about data - that give non trivial insights in the process that generated it".

As it will be shown in the following sections, researchers from all over the world strive to find efficient algorithms and frameworks which, when applied to a database, would fetch "interesting" patterns.

However, can this "interestingness" of a pattern somehow be measured and is there any straightforward way to mine patterns? In other words, can we find a process or a framework, which, when applied to a database, will fetch the most "important" set of patterns?

This essay aims to present some noteworthy papers that outline the field of pattern set mining. It is by no means a complete list; rather an introduction to different approaches to the general problem. In section 3, an in depth view of a specific subpart of pattern set mining is given, or more accurately, how a relational database management system (RDBMS) can be combined with some of these papers to produce notable results.

2 Frequent Pattern Mining Approaches

2.1 Frequent item set mining

Before we embark into the journey of pattern set mining, it would be wise to conceptualise somewhat the problem with the help of an example [25].

Consider an online movie streaming platform that has various movies from different genres. Their data analytics team, as one might expect, wants to learn more about which movies are watched consecutively. This offers them great insight on which movie projects to fund and continue hosting on their native platform.

Given a set of items (i.e. an *itemset*) $I{=}i_1, ..., i_n$, and in our case movies, a transaction t is a subset of I and a database D is a set of those transactions. In our example, a transaction t would be a sequence of movies a viewer watched (e.g. "Rocky I" then "Rocky II" and then the "Million Dollar Baby") while a database D would contain various of these transactions alongside a unique id to make them apart.

Although the streaming platform surely does contain a lot of different kinds of movies, it would make sense if a viewer has some preference to a certain genre. For example, there is one person who likes comedies but is clearly not into autobiography movies. In this case, even though "Groundhog Day" and "Bohemian Rhapsody" may be the two most watched films in the platform, it wouldn't make sense to recommend the latter film to them -even though it certainly could, as there are not a lot people who actually follow through with that choice. Therefore, a framework is needed that can somehow mine the patterns that occur more frequently than a desired threshold. This occurrence in the database will be denoted from now on as *support*.

Formally, this type of problem is known as *Frequent Itemset Mining* (*FI Mining*); Given a transaction database D over a set of items $\mathcal{I}$, find all itemsets that are frequent in D given the minimal support threshold θ.

Agrawal, Imilienski and Swami can be considered among the forefathers of Pattern Set Mining. In their paper [1], a gentle introduction towards the problem of mining large databases for association rules is made. In doing so, the problem is broken down into two subproblems; generate all combinations of itemsets that have a transactional support above a certain threshold (i.e. per the authors "large itemsets") and for those itemsets generate rules that use items from this "large" itemset. By definition the second subproblem is far easier than the first. Their algorithm (which would be later called *AIS*) makes use of the frontier set, a predecessor to the Apriori algorithm (which will be described later on). AIS makes numerous passes over the database, which reduces significantly the algorithm performance. Due to this, the authors attempt to introduce memory management techniques to mitigate this issue.

Having set those foundations, Agrawal alongside Srikant greatly improve the previous process by introducing in their paper the Apriori algorithm [2]. The basic intuition of the Apriori algorithm is based on the idea that any subset of a large itemset must be large. Thus, the candidate sets with k items can be generated by joining large itemsets with $k-1$ items and deleting those that contain any subset that is not large.

Mannila et al. [20] extend the notion of patterns into the domain of sequence of events. More specifically, they analyse data which could be viewed as a sequence of events where each event has an associated time of occurrence. In particular, they present efficient algorithms for the discovery of all frequent episodes, from a given class of episodes. For reference, episodes are nothing more than partially ordered sets

of events or more informally a collection of events occurring together.

However, since the notion of "closeness" is pointless without some time bounds, the authors define the *window*; a sequence of events with a specific time span. Based on this, the *winepi* algorithm is presented which uses a sliding window to go through the event sequence. As an output, it finds episode rules within a sequence with respect to a certain time span. Additionally, the problem is also tackled from another approach, that of minimal occurrence of episodes. The algorithm called *minepi* can be also used to solve *winepi* since a window contains an occurrence of an episode exactly when it contains a minimal occurrence.

By applying these algorithms in the telecommunication alarm management not only did the authors show that their algorithm works correctly but also that the pattern mining set field can be extended in other fields where it may not seem relevant from a first glance.

2.2 Limiting the results

Even though the Apriori algorithm described before would be a groundbreaking paper that would set as a reference point for future algorithms in the field, it has two major drawbacks. First, it is extremely costly to handle a huge number of candidate sets. For example, to discover a frequent pattern of size 100 it must generate 2^{100}-2 candidates in total. This problem is also known as pattern explosion in which an algorithm produces an extreme amount of candidate patterns where only some of them may be of relevance. Second, Apriori algorithm imposes a significant overhead in the computation by having to repeatedly scan the database and check a large set of candidates by pattern matching.

Thus, based on those, it seems fairly natural to set some constraints on the number of candidate sets fetched by the Apriori algorithm.

Han et al. [14] created a method to mitigate the aforementioned problems. This was achieved through *FP-trees*, a compact data structure which avoids repeatedly scanning the transaction database. A FP-tree consists of one root labeled as "null", a set of item-prefix subtrees as the children of the root, and a frequent-item-header table. The frequent item header table is used so that each item points to its first occurrence in the tree via a node-link. The authors suggest that the construction of a compact FP-tree can be done through an algorithm called *FP-growth*. This algorithm, by keeping track of the increasingly bigger frequent itemsets through this aformentioned structure, can construct the complete set of frequent items. The authors admit that, since it's a main memory structure, it is unrealistic to create an FP-tree in a big data context. Therefore, they propose splitting the database into sub databases and for each of them finding its corresponding FP-tree. The problem of FP-growth in big data will be addressed in more detail in section 3.0.3.

2.3 Constraint pushing

In the constraints that have been mentioned thus far, in order to decide whether something is a pattern or not, it had to do with the frequency of an itemset. However, while the support is an example of a constraint, it is certainly not the only one.

Reverting back to the movie example, the data analytics team in order to maximize ad revenue may be also interested in the average duration a user is spending on the platform (apart from the amount of movies they watched). Such constraint, by its definition, is considered hard for an Apriori algorithm since in step $k+1$ a new lengthy or short movie may change completely the end result.

Therefore, the important thing is not so much the constraint itself but whether it holds for a subset or a superset of the given set. *Monotone* constraints hold for supersets of X if the constraint holds for X while anti-monotone constraints hold for subsets of X.

Pei et al. [22] study these kind of constraints and come up with some concepts in order to implement certain types of them in some mining algorithms. They introduce *convertible* constraints which fall into three classes: convertible anti-monotone, convertible monotone and strongly convertible. As per the name suggests, a constraint can be converted to monotone (or anti-monotone) by imposing an order to it. Strongly convertible are those constraints that with a particular order $\mathcal{R}$ are convertible anti-monotone, while with respect to its inverse $\mathcal{R}^{-1}$ are convertible monotone.

Based on those, two algorithms are developed: $\mathcal{FIC}^{\mathcal{A}}$ for mining FI with convertible anti-monotone constraints and $\mathcal{FIC}^{\mathcal{M}}$ for convertible monotone constraints. The main idea for both of them is based on the notion of FP-trees that was discussed earlier.

2.3.0.1 Looking ahead id est ExAnte

Based on the previous papers, constraint pushing techniques have been proven to be effective in reducing the search space in frequent pattern mining. However, while pushing anti-monotone constraints has been proven to actually be profitable, this doesn't stand for monotone ones.

Bonchi et al. [10] show that this claim is unjustified and introduce ExAnte, a preprocessing data algorithm which greatly reduces both the search space and the input dataset in constraint frequent pattern mining. ExAnte can be used with any constraint that has a monotone component and thus convertible

monotone constraints. Not only that, but being a preprocessing algorithm ExAnte can be coupled with any constrained pattern mining algorithm.

2.4 Missing the forest for the trees

Those previous papers were strong advocates of constraint pushing. Indeed, it greatly reduces on the fly the amount of candidates that exist and thus effectively overcomes the limitations of Apriori. However, constraint pushing cannot be considered the panacea of all pattern set mining frameworks.

Jochen and Güntzer [15] express their incredulity to this aforementioned technique. They state that recklessly applying constraints during the mining runs diverges from the point of an Knowledge Data Discovery process (KDD process). Instead, what they propose, is to do a single and expensive mining run once and apply the filtering afterwards. This approach, albeit slower than constraint pushing - especially in the initial run, does not narrow a priori the result set based on some potentially ill founded assumption by the user.

This concept, although it makes some very overoptimistic assumptions, such as that the association rules can fit in main memory, it raises some very valid points on how researchers should address the problems of pattern set mining. Sometimes, even though computer scientists develop algorithms that are optimal in the sense that they can solve a problem really fast, they can forget the bigger picture; that of pattern set mining as a process from start to end.

2.5 Condensed representation

Up until now, the general approaches for finding frequent itemsets were either looking over all possible candidates (e.g. Apriori) or applying some sort of pre or post filtering to the complete set results (e.g. ExAnte). In the following section, a different approach will be presented; that of condensed representation, which in its most general form, as the name suggests, tries to find patterns that offer the most succinct representation of the whole set of patterns.

Pasquier et al. [21] propose a new and efficient algorithm, named *Close*. Close is loosely related to Apriori, in the sense that it is based on *closed itemset lattice* which is a sub-order of Apriori's *subset lattice*. For reference, a subset lattice is an abstract mathematical structure which imposes a partial order on the sets.

Delving into the details of the paper, Close algorithm is strongly based on *closed itemsets*. An itemset I is considered closed if there exists no superset that has the same support as I. Reverting back to the movie example, if the viewers of Rocky I ($R1$) always compulsively also watched Rocky II ($R2$), then $R1$ is not closed since $R1R2$ has the same support as $R1$.

Therefore, using this concept and the notion of subset lattice, close Algorithm is able to prune a large number of candidate itemsets reducing the number of passes in the database as well as the CPU overhead.

Even though Close produces in general few candidates, it is important to notice that it does not necessarily outperform Apriori. Some particular datasets may force Close to perform exceptionally bad rendering its functionalities useless.

Calders and Goethals [7] exploit this concept of condensed representations to introduce the *Non Derivable Itemsets* (NDI).

More specifically, they present some rules (called deduction rules) to deduct tight upper and lower bounds on the support of a candidate set. When those tight upper and lower bounds coincide, then the itemset is called *derivable* as its support can be derived exactly from its bound. Understandably, those itemsets pose no interest to the researcher and can be effectively discarded. On the other hand, NDIs are especially intriguing since they are a cornerstone in the minimal representation of all frequent itemsets. Their algorithm called *NDI-algorithm* is based on the Apriori algorithm with some pruning for the non frequent and also derivable itemsets.

Unfortunately, these condensed representations have no guarantees on how many they will exactly be. However, a nice property of the NDIs is that their size is bounded by the logarithm of the database which indicates that in general the NDI set size will not be very large. Empirical evaluation verifies this by showing that NDI is among the best (if not the best) condensed representations of FIs.

Knobbe and Ho [16] proposed a new approach which does not focus on finding patterns by themselves but rather based on the interestingness of them when comparing them to other patterns. They present a method of selecting a small subset of patterns, called a *pattern team*, that optimises some given quality measure for a set of patterns.

Based on six intuitions about the essence of patterns, four quality measures are thus defined: *joint entropy*, *exclusive coverage*, *Decision Table Majority Accuracy* (DTM) and *Area Under Curve*(AUC). Joint entropy is the amount by which the uncertainty of one random variable is reduced due to the knowledge of another. Exclusive coverage is a measure which favours pattern sets that have less overlap between other patterns. DTM accuracy determines how predictive a pattern set is by computing the accuracy of a classifier while, lastly, AUC measures the area of the convex hull of the patterns in the ROC-space.

Experiments show that Joint Entropy and DTM perform well in general with the others performing adequately in only some cases.

The same authors therefore continue on the same topic of choosing a few among the many [17]. In their

paper, their main interest is selecting an itemset from the total set of items such that the database is partitioned with as uniform of a distribution over the parts as possible.

Still, as many itemsets could fit into that selection process, an (optimal) measure has to be used to find the best. For that, the notion of *entropy* from the field of Information Theory is employed. In short, entropy measures, in this particular case, distinction; items that appear in either most of the transactions or none at all, convey little to no information as they can't be distinguished in a database.

Based on that concept, the authors are interested in the itemsets of k-size that maximize the joint entropy among all other itemsets of k-size. Such itemsets are called maximally informative k-itemsets (mikis).

Four exact and one greedy algorithms are presented to compute mikis. The central principle of the exact algorithms is to consider all subsets of size k in lexicographic order (i.e. generalised alphabetical order) and compute the joint entropy of each in order to find the maximum.

The almost exhaustive search in the first algorithm is, needless to say, computationally expensive and thus, tweaks are made in the other exact algorithms to mitigate this issue. These adjustments are reflected in the experiments, where significant improvements are made over the baseline solution, but even then with conflicting results in some particular database sets.

On the same note, Bringmann and Zimmermann [6] introduce a heuristic approach to select a few patterns, amongst the many, as best as possible. To elaborate further on this, let S be the set of patterns p_i present in a database $\mathcal{T}$. Their goal is to select a subset $S^* \subset S$ that has three main characteristics: (a) S^* should be of relatively small size, so a user can inspect it, (b) elements of S^* should accurately describe the main aspects of $\mathcal{T}$ and (c) it should contain only the essential; that is patterns, that could be described by another pattern, should be discarded. This redundancy, with respect to $\mathcal{T}$ is quantified, according to a measure Φ.

Given those, the heuristic approach considers three different alternatives based on (three) different measures. Experiments done on various datasets show no clear winner among them with one measure (Φ_C) being computationally most expensive but also showing very good reduction effects, while the others recover more of the original partitioning.

Geerts et al. [12] introduce a new and objective interesting measure to extract knowledge from binary databases using the concept of *tiles*. A tile is a region of databases consisting only of ones. A collection of possibly overlapping tiles constitutes a tiling.

In that paper, they address the following problems: "the *maximum k-tiling* problem which asks for a tiling consisting of at most k tiles having the largest possible area; the *minimum tiling* problem which asks for a tiling of which the area equals to the total number of ones in the database and consists of the minimum number of tiles; the *large tile mining* problem which asks for all tiles in the database each having at least some minimum area; and the *top-k tiles* problem which asks for the k tiles that have the largest area". All of them are proven to have the NP-hard property.

As for the algorithm, the *Large Tile Mining* (LTM) algorithm uses a branch and bound strategy with several pruning techniques to solve the large tile mining problem. Meanwhile, k-LTM, a greedy algorithm, which is built on top of LTM finds the top-k tiles. The other problems are addressed through NP-reduction techniques from other NP-hard problems.

Experimental evaluation is done on a sparse as well as a dense dataset. For context, a sparse dataset is a dataset which contains a relatively high percentage of zeroes. The experiments verify the efficacy and efficiency of the algorithms. However, as noted by the authors, this work is preliminary and many improvements can be made regarding the experimental evaluation and upper bounds of the approximation.

2.6 Selecting few to describe the whole with mathematics

Up until this point, all of the approaches described in the papers were either based on exhaustive search within vast pattern spaces or on some sort of post or pre filtering of the results. However, from a pure theoretical statistical perspective this is redundant; representative sampling guarantees that inferences and conclusions can reasonably extend from the sample to the population as a whole.

This is what motivated Boley et al. [5] when they presented local pattern sampling algorithms which are by definition non-enumerative. Surely, as noted also by them, they weren't the first breaking new ground in combining stochastic processes with pattern set mining [4]. They are, though, the first to sample patterns directly, avoiding the quite time consuming stochastic process named Markov Chain Monte Carlo (MCMC).

Four ways of sampling are proposed which all have a two step process as their backbone. In the first step, one element from a properly constructed set of objects is drawn and in the second step a sub-object from those preconstructed objects is chosen. Using those drawn objects, one can build a classifier which will predict, as verified by some experiments, well enough the set of frequent itemsets.

Sampling is not the only way to estimate the number of frequent patterns (for a given threshold). Leeuwen and Ukkonen [18] present an algorithm to estimate the frequent pattern set quickly, hence the name *FastEst*. Using FastEst, they also develop another algorithm (called *SPECTRA*) which uses iso-

tonic regression to estimate the number of frequent patterns for all possible thresholds.

The main idea behind FastEst relies on Knuth's algorithm. In short, Knuth estimates the size of a search tree without exhaustive traversal by constructing a tree where every node corresponds to a frequent itemset. FastEst traverses the tree until it reaches a maximal frequent itemset and then uses Knuth's process to compute the size of the set fast.

2.6.0.1 From rules to itemsets

Turning the focus from finding "interesting" association rules to finding "interesting" itemsets may seem redundant at a first glance, since the latter is already contained in the former. However, Webb [27] having a different view introduces *self-sufficient* itemsets. Per the author, "self-sufficient itemsets are those whose frequency cannot be explained solely by their frequency of either their subsets or supersets". Itemsets that are not self sufficient are unlikely to be interesting in many contexts. An important note is though that this does not imply that all self-sufficient itemsets are of great interest.

To make this more concrete, consider the following example. Let DiS denote a distant star while s denote the shape of that star. If it were not known a priori that all stars are round, the fact that they are could be an interesting discovery. But once this is revealed, it is expected that every superset of $\{DiS,s\}\cup X$ will have the same support as $\{DiS\}\cup X$. Hence, $\{DiS,s\}$ poses little to no interest.

To test whether those interesting patterns are self-sufficient or not, Webb employs statistical testing. There are three main approaches to tackle this problem which are intended to be applied as a post-processing step to filter the itemsets from the general set of itemsets.

Experiments show that these approaches perform well for collections of itemsets that are few hundred but this does not apply when hundreds of thousands are to be computed.

In a somewhat similar statistical context, Lijffijt et al. [19] constructed a novel approach for finding the smallest set of results that accurately describes the data by using statistical significance testing.

The approach consists of three major components: (i) a null hypothesis, (ii) a test statistic and (iii) some constraints. The null hypothesis expresses the background knowledge the user has when dealing with the data. The test statistic quantifies in a single number all the properties about the data a user wants to have explained, while the constraints are, for example, the patterns that the algorithm is allowed to give as an output.

Formalising this concept a bit more, it is nothing more than a maximization problem; for a given k, find a set of constraints I of size k such that the p-value for that set of constraints is maximized.

The most obvious solution to this problem (denoted by I^*) would be to perform an exhaustive search over all sets of constraints (where $|I|=k$) and select the subset with the maximal global p-value. However, this requires exponential time and is NP-hard. Therefore, other options have to be considered. The $GREEDY$ algorithm selects every time the best constraint that maximizes the objective function and terminates when $|I|=k$. $GREEDY$, albeit not optimal, can be considered as such in some cases, as proven by the authors.

2.7 Quantifying subjectiveness

In the previous section, a notable point was made which was deliberately overlooked; a user has to select a null hypothesis for their model which would be used as a baseline for measuring the interestingness of a pattern. This concept, as it is, is a bit vague; subjective interestingness is not defined yet in a formal way which would make use of a user's background information. Tijl de Bie [8] aims to bridge that gap by using the maximum entropy (*MaxEnt*) distribution subject to some constraints that represent the user's knowledge.

The MaxEnt is nothing more than a way to formalise surprise through entropy. The probability distribution which best represents the current state of user's background knowledge about the data is the one with the largest entropy (i.e. surprise).

Various measures of unexpectedness are considered in the paper such as *self-information* (the smaller the probability the more surprising it is), information compression ratio or even the p-value from hypothesis testing.

Continuing on the same topic, Tijl de Bie [3] moves up one level of abstraction and introduces a framework to formalise interestingness in a subjective manner. Thus, he manages to sketch a process in which a user with a given initial belief state about the data, is able to update their beliefs based on the most surprising data. This process, albeit it may seem at a first glance as only a high level concept, is actually properly defined through various mathematical tools. In fact, it is somewhat similar to Bayesian statistics, but adjusted to take advantage of the notions defined in the previous paper.

2.8 Pattern Explosion: Revisited

The notions of entropy and tiling (section 2.7 and 2.5 respectively) give a fresh new perspective at things. Various concepts can be revisited now where new solutions with these tools can be found.

Pattern explosion, as briefly mentioned before, is an important issue of pattern set mining. Loose constraints or ill-defined algorithms lead to an extreme

amount of candidate frequent itemsets. To battle this, Vreeken et al. [26] propose a heuristic algorithm called *KRIMP* using the Minimal Description Length (MDL) principle. In short, the MDL is a model selection principle where the shortest description of the data is the best model.

To compress the data, the concept of a code table is used. More formally, let I be a set of items and C a set of code words. A code table CT is a two column table such that the first column contains itemsets and the second column contains elements from C, such that each element of C occurs at most once.

KRIMP constitutes of a greedy search algorithm that starts from the singleton patterns in a code table and adds to it patterns sequentially from a preordered list of patterns. If a pattern improves the overall compression (score) of the database, then it is added in the final list of patterns. Otherwise it is discarded and the algorithm moves on to the next pattern.

In order to test whether *KRIMP* is actually a viable solution to the pattern explosion problem, it is tested against 27 datasets of various sizes. The algorithm performs fairly well with some cases achieving a reduction of seven orders of magnitude.

3 RDBMS AND PATTERN SET MINING: AN INTERESTING PAIR

Pattern Set Mining as it has been done exceedingly clear by now, extracts, as per the name suggests, patterns from a database. Up until this point, the focus of this paper has been on the different approaches at tackling the problem from a more theoretical perspective. That is, while most research papers that were mentioned had algorithms which were implemented and tested against other algorithms, the problem of applying this knowledge to a relational database with queries was neglected.

This section aims to address this issue by presenting an SQL-like language (section 3.0.1) and how these languages were used with two well known frequent pattern mining algorithms to extract results from relational databases (section 3.0.2 &3.0.3).

The widespread use of the internet, especially in the 21st century, led to an explosion of data created by mankind. To tackle this problem, all of the companies by now have chosen to implement either a more modern solution (NO-SQL approach) or a standard relational database model. Relational databases consist of an excellent tool to store, filter and access data. These operations are usually done through a Structured Query Language (SQL) with the help of highly specialised cluster computing frameworks. However, as advanced this field may seem now, just 25 years ago, this concept was still very new. The majority of pattern mining systems were developed largely on file systems and specialised data structures. Combining pattern set mining with database systems was at best loose and access was done through an interface or some sort of SQL-like language.

3.0.1 DMQL: A first attempt at formalising the language

Han et al. [13] take advantage of the ongoing attempt to standardize the evergrowing SQL-1999[1], Object Management Group (OMF) and Object Data Management Group (ODBG) languages to create a Data Mining Query Language (DMQL). Although, this language cannot be considered complete in any way, it would act as a great influence for other papers (see section 3.0.2).

The authors, thankfully, before arbitrarily designing this language, decided to set some guidelines based on the general philosophies of the field of data mining. Those philosophies revolve around the following five basic points.

(I) The set of data relevant to a data mining task should be specified in a data mining request.

(II) The kinds of knowledge to be discovered should be specified in a data mining request.

(III) Background knowledge could be generally available for data mining process.

(IV) Data mining results should be able to be expressed in terms of generalized or multiple-level concepts.

(V) Various kinds of thresholds should be able to be specified flexibly to filter out less interesting knowledge.

Based on these, a DMQL can now be designed. This language would consist of four cornerstones: (i) the set of data in relevance to the data mining process, (ii) the kind of knowledge to be discovered, (iii) the background knowledge and (iv) the justification of interestingness of the knowledge.

The set of relevant data can be thought as a query (e.g. an SQL-query) which asks the database for a specific result. The second point can be considered as the association rule per se; the concept of searching relations and patterns in the data. Background knowledge is nothing more than a high level concept of the query language design. It should not be confused, though, with the background knowledge described in (ADD REF). Finally, the fourth cornerstone pertains to the well known concept of threshold in a database such as the aforementioned support of an itemset.

DMQL supports a SQL-like syntax with an extended Backus-Naur grammar. Figure 1 shows an example of a query in DMQL while Figure 2 shows how to mine association rules in it.

Finally, as mentioned before, the goal of their paper is to construct a robust language for data and pattern mining processes. However, it would be fully

[1] Also called SQL-3

```
⟨DMQL⟩ ::=
  use database ⟨database_name⟩
  {use hierarchy ⟨hierarchy_name⟩ for ⟨attribute⟩}
  ⟨rule_spec⟩
  related to ⟨attr_or_agg_list⟩
  from ⟨relation(s)⟩
  [where ⟨condition⟩]
  [order by ⟨order_list⟩]
  {with [⟨kinds_of⟩] threshold = ⟨threshold_value⟩
    [for ⟨attribute(s)⟩]}
```

Figure 1: DMQL query

```
⟨rule_spec⟩ ::=
  find association rules [as ⟨rule_name⟩]
```

Figure 2: Mining association rules in DMQL

reasonable for a user to expect a graphical user interface (GUI) which displays the results in a nice and comprehensible way while in the back it applies those "core" five points. Therefore, a GUI is designed in the DBMiner System but due to a huge variety of data mining GUI systems across the scientific community, a standard for it, is difficult to be set.

3.0.2 Piecing the puzzle together

Sarawagi et al. [23] paved the way in trying to unify the various data mining techniques with database systems. In their paper, a task force of researchers with deep expertise on mining methods and the IBM Database Management System (i.e. DB2), tried to implement efficiently the Apriori algorithm, described in section 2.1, by exploring several implementation alternatives scrapped from various papers. This implementation would be then tested against various other architectures to see whether it is viable or not. Varying results were achieved.

Before moving forward into the details of the paper, one might question whether SQL can be considered a feasible option in the first place. Indeed, a mining computation expressed on SQL can seriously leverage the program by taking advantage of the underlying SQL-parallelization, portability and scalability it offers.

Figure 3 indicates the architectures the authors had

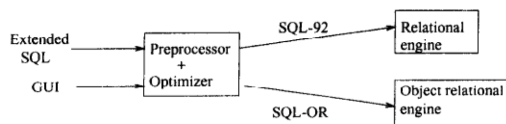

Figure 3: Architecture Sarawagi et al. considered

in their mind when thinking this process. Briefly, the mining operation is done on some extension of the SQL or a graphical interface before being translated into either SQL-92[2] or to an object-relational SQL (SQL-OR) by a preprocessor.

This architecture is compared with three other alternatives : i) *Read directly from DBMS*, ii) *Cache-mine* and iii) *User Defined Functions* (UDFs).

In the first case, as per its name, data is read directly from DBMS tuple by tuple. There are two variants in this category: *loose coupling* and *stored procedure*. In the loose coupling approach, the DBMS and the mining process run in a different address space, whereas in the stored procedure they run in the same. Cache mine is a slight variation of (i) where the entire database is read once and the algorithm stores the relevant data in a side buffer. Finally, UDFs are a collection of User Defined Functions which are placed appropriately in the SQL data scan queries. Most of the processing happens in the UDFs and thus DBMS's main goal is to provide tuples to the UDFs.

As for the candidate generation algorithm in SQL, it resembles the original Apriori implementation. Each pass k of the algorithm generates a candidate set C_k from frequent itemset F_{k-1} of the previous pass. This is a two step process, namely the *join step* and the *prune step*. In the join step, a superset of the candidate itemsets C_k is generated by joining F_{k-1} with itself (Figure 4), while in the *prune step*, all itemsets $c \in C_k$, where some subset of c is not in F_{k-1} are deleted. Due to the nature of the SQL, these two steps can be performed simultaneously as a k-way join (Figure 5).

```
insert into C_k select I_1.item_1, ..., I_1.item_{k-1}, I_2.item_{k-1}
from    F_{k-1} I_1, F_{k-1} I_2
where   I_1.item_1 = I_2.item_1 and
          ⋮
        I_1.item_{k-2} = I_2.item_{k-2} and
        I_1.item_{k-1} < I_2.item_{k-1}
```

Figure 4: Join Step SQL code

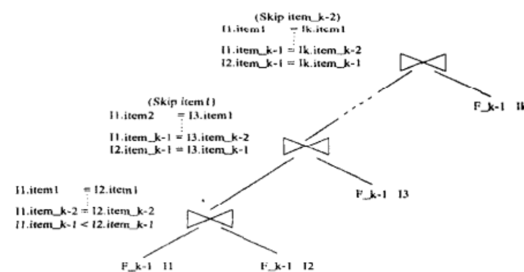

Figure 5: Candidate Generation for any k

As was the case with many other algorithms, the part that takes the most time is counting the support to find frequent itemsets. Thus, the authors consider

[2] SQL-92 was the third revision of SQL.

two different categories of SQL-implementations: (A) Based on SQL-92 and (B) based on SQL-OR.

3.0.2.1 SQL-92 approaches

The approaches based on SQL-92 consist of "k-way joins" and "subquery based approaches".

K-way joins:
In each pass k, the candidate itemsets C_k are joined with k transaction tables T and then a group by is executed.Figure 6 shows the SQL query.

```
insert into F_k select item_1, ... item_k, count(*)
from      C_k, T t_1, ... T t_k
where     t_1.item = C_k.item_1 and
                 ⋮
          t_k.item = C_k.item_k and
          t_1.tid = t_2.tid and
                 ⋮
          t_{k-1}.tid = t_k.tid
group by  item_1,item_2 ... item_k
having    count(*) > :minsup
```

Figure 6: K-way SQL query

Subquery based approaches:
A subquery based approach is somewhat more complex than the K-way one. It takes advantage of the common prefixes between the itemsets in C_k to reduce the amount of work done during support counting. This support counting phase is split into a cascade of k subqueries. The l-th subquery Q_l finds all tids that match the distinct itemsets formed by the first l columns of C_k (d_l). The output of Q_l is joined with T and d_{l+1} to get Q_{l+1}. Finally, as a last step the output is obtained by a group-by on the k items to count support as before. Figure 7 shows the SQL commands.

```
insert into F_k select item_1, ..., item_k, count(*)
    from (Subquery Q_k) t
    group by item_1,item_2 ... item_k
    having count(*) > :minsup

Subquery Q_l (for any l between 1 and k):
    select item_1, ... item_l, tid
    from T t_l, (Subquery Q_{l-1}) as r_{l-1},
      (select distinct item_1 ... item_l from C_k) as d_l
    where r_{l-1}.item_1 = d_l.item_1 and ... and
        r_{l-1}.item_{l-1} = d_l.item_{l-1} and
        r_{l-1}.tid = t_l.tid and
        t_l.item = d_l.item_l
```

Figure 7: Subquery based approach

3.0.2.2 SQL-OR based approaches

SQL-OR based approaches consist of one called *GatherJoin* with its 3 variants and another called *Vertical*.

GatherJoin generates all possible k-item combinations of items contained in a transaction, joins them with the candidate table C_k and counts the support of the itemsets by grouping the join result. This concept is shown in Figure 8. Its three variants *GatherCount*, *GatherPrune* and *Horizontal* aim to address some of the drawbacks that it has, with some tradeoffs though, in space or performance.

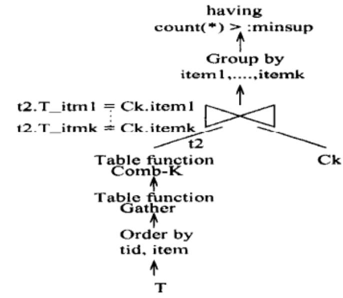

Figure 8: GatherJoin SQL-OR approach

Vertical, as described before, is another approach of the SQL-OR. First, the data table is transformed into a vertical format by creating for each item a Binary Large Object (BLOB) containing all the *tids* that contain the item. Then, the support of the itemsets is computed by merging together these tid-lists. For reference, a BLOB is nothing more than a collection of binary data stored as a single entity. The advantage of this, is that it can be written as a single SQL query for any k.

An overall performance comparison of the SQL-OR approaches declares *Vertical* as the best option for higher passes. This, however, does not hold when the size of the candidate itemsets is too large. Therefore, a hybrid scheme is adopted which chooses the best of *GatherJoin*, *GatherCount* and *Vertical* based on some computational costs incurred by the problem a user is facing and the algorithms themselves.

Having said all that, the comparison of the three architectures described before and the ones from Figure 3 can be made. Timewise Cache-Mine has the best or almost the best performance in all the cases, while the Stored-procedure approach has the worst. The SQL-approach comes second due to the fact that it takes more time in the second passes compared to Cache-Mine even if it is significantly faster in the first pass. But still, it is 1.8 to 3 times better than the Loose-coupling approach. Spacewise the UDF and the Stored-procedure require the least amount of space. Unfortunately, the SQL-approaches require roughly as much extra storage as the data. No clear winner can be drawn, as every approach has some trade-offs. The SQL approach offers some auxiliary advantages like easier development and maintenance but it might not be as portable as the Cache-Mine approach across different database management systems.

3.0.3 Refining the idea with FP trees

The SQL-based approach made by Sarawagi et al. (section 3.0.2) was a truly fascinating idea that tried to implement a theoretical algorithm in a relational database. However, as it was based on the Apriori algorithm, it too suffered the problems induced by it described in section 2.2.

This motivated Shang et al. [24] to use different approaches for the SQL-based frequent pattern mining problem. Therefore, they present an evaluation of SQL based frequent pattern mining with a novel frequent pattern growth (FP-growth) method as described in section 2.2. This approach is according to the authors “highly efficient and scalable for short and long patterns”, a claim highly accurate by their results.

However, one might wonder if the concepts of a FP-tree and a relational table, holding millions of data, can even be combined together in the first place. Even though a FP-tree is a compact data structure, it would be unrealistic to construct it in main memory in a big-data context. No main memory would be able to hold such amounts of (compressed) data or, even if that was the case (e.g. in a highly thought clustered solution) , it would lose its purpose. However, using RDBMS's buffer management systems, this memory limitation obstacle can be successfully surpassed. A buffer will only load the necessary parts of the table required from the algorithm, while leaving the rest unchanged, saving up precious space.

The authors propose two different approaches for combining FP-trees with SQL based frequent pattern mining; namely *FP* and *EFP*. In short, FP looks at each frequent item individually to determine whether it should be added into the table FP, while EFP generalises the previous concept by introducing a bigger but faster structure.

As for the input, the transaction data is transformed into a table T with two column attributes: transaction identifier (*tid*) and item identifier (*item*). For a given *tid*, there may be multiple rows corresponding to different items in the transaction. The number of items per transaction is a variable and is unknown during table creation time.

For the sake of emphasis, the properties of a FP-tree are restated once more; the node-link property (i.e. all possible frequent patterns can be obtained by following each frequent's node link) and prefix path property (i.e. to compute the frequent patterns for a node a_i in a path, only the prefix sub-path of a_i in P need to be accumulated). Since the data stored inside most likely won't be unique, a data structure, called flat table, is used which is highly efficient for this kind of job.

Having those two properties as a guide, an FP-tree can be represented by a table FP with three column attributes; item identifier (*item*), number of transactions that contain this item in a subtree (*count*) and item prefix subtree (*path*). Notice that the second node-link property is reflected in the *path* attribute.

The creation of a FP-tree is a two step process: Create table T' and from that create table FP.

In the first step, the transaction table T is transferred into table T' from which infrequent items are removed. Since, as mentioned before, the amount of entries are unknown, size plays a major role in the cost of joins that include T. However, this can be optimised by pruning the non-frequent items from the transactions after the first pass and inserting them into table T'. Then, in the following passes, instead of joining with T, a join with T' is done which is a lot less computationally expensive.

In the second step, frequent items in T are sorted in descending order by frequency. Afterwards, for each item if it does not have the same (attribute) *item* and (attribute) *path* as those in the FP, it is inserted into the FP as a new item with the count being 1. Otherwise, the FP is updated by increasing the count by 1.

The approach described previously, even though it is undeniably a fair attempt to tackle this issue, suffers from computational issues since the construction of an FP-table is an extremely time expensive process. Each item must be tested one by one to construct the table $ConFP$ which is rather inefficient. Therefore, the second approach called *EFP* aims to alleviate this issue by introducing an extended *FP-table* called *EFP-table*. *EFP-table* has the same attributes as *FP-table* but performs much better than the latter. Moreover, a FP-table can be easily created by an EFP-table with admittedly some minor space issues.

EFP is obtained by directly transforming frequent items in transaction table T'. The path attribute in EFP-table is set as follows; the path attribute of the first frequent item i_1 is set as *null*, while the path of the second frequent item i_2 is set as $null : i_1$ and so on. Figure 9 (b) shows an example of such a EFP-table. Combining items with identical paths would fetch table FP. Figure 9 also shows the aforementioned difference in size between FP and EFP. However, this space compromise is clearly unimportant in favour of increased performance, as EFP, in contrast to FP, avoids checking each transaction one by one.

Apart from the implementation, the authors put the algorithms to the test, comparing both FP and EFP with various other SQL-implementations such as the *Loose* and the *k-way join* approach described previously. As expected, the EFP-algorithm managed to outperform the Apriori algorithm when the support threshold is low, but it did not manage to achieve this with higher values. Additionally, as verified by experiments, EFP has superior performance

Item	*Count*	*Path*
3	1	*null*
1	1	*null* : 3
2	3	*null*
3	2	*null* : 2
5	2	*null* : 2 : 3
1	1	*null* : 2 : 3 : 5
5	1	*null* : 2

(a) An FP table for Table 1

Item	*Count*	*Path*
3	1	*null*
1	1	*null* : 3
2	1	*null*
3	1	*null* : 2
5	1	*null* : 2 : 3
2	1	*null*
3	1	*null* : 2
5	1	*null* : 2 : 3
1	1	*null* : 2 : 3 : 5
2	1	*null*
5	1	*null* : 2

(b) An EFP table for Table 1

Figure 9: table FP and EFP

over FP for reasons mentioned previously. Finally, based on the investigations, it seems that EFP and *Path approach* can get better performance than *K-way* join on large data sets or long patterns.

4 Conclusions

Having said all this lengthy narrative, a natural question would be if there is more to that. The answer to this, is that there is always more. Matters pertaining to pattern set mining are, and will be, on the rise especially if one considers the amount of data humanity is producing. Finding patterns in a database, albeit sounds extremely simple, is an extremely intricate problem that has generated dozens of approaches towards finding the perfect solution. Meanwhile, as section 3 says the field of pattern set mining is much more broad and overlapping across different fields such as the one of database management. "*And to make an end is to make a beginning. The end is where we start from...*" as T.S. Eliot wrote [9].

References

[1] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. *SIGMOD Rec.*, 22(2):207–216, jun 1993.

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

[3] Tijl De Bie. Subjective interestingness in exploratory data mining. In *International Symposium on Intelligent Data Analysis*, pages 19–31. Springer, 2013.

[4] Mario Boley, Thomas Gärtner, and Henrik Grosskreutz. Formal concept sampling for counting and threshold-free local pattern mining. pages 177–188, 04 2010.

[5] Mario Boley, Claudio Lucchese, Daniel Paurat, and Thomas Gärtner. Direct local pattern sampling by efficient two-step random procedures. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, page 582–590, New York, NY, USA, 2011. Association for Computing Machinery.

[6] B. Bringmann and A. Zimmermann. The chosen few: On identifying valuable patterns. In *2007 7th IEEE International Conference on Data Mining (ICDM '07)*, pages 63–72, Los Alamitos, CA, USA, oct 2007. IEEE Computer Society.

[7] Toon Calders and Bart Goethals. Non-derivable itemset mining. *Data Mining and Knowledge Discovery*, 14:171–206, 02 2007.

[8] Tijl De Bie. Maximum entropy models and subjective interestingness: an application to tiles in binary databases, 2010.

[9] TS Eliot. Little gidding. four quartets. *Selected poems*, 1943.

[10] Bonchi F., Giannotti F., Mazzanti A., and Pedreschi D. Exante: Anticipated data reduction in constrained pattern mining. In *PKDD 2003 - 7th European Conference on Principles of Data Mining and Knowledge Discovery, pp. 59–70, Cavtat-Dubrovnik, Croatia, September 22-26, 2003.* Springer, Berlin , Germania, 2003.

[11] Jonas Fischer and Jilles Vreeken. Differentiable pattern set mining. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery Data Mining*, KDD '21, page 383–392, New York, NY, USA, 2021. Association for Computing Machinery.

[12] Floris Geerts, Bart Goethals, and Taneli Mielikäinen. Tiling databases. pages 278–289, 10 2004.

[13] Jiawei Han, Yongjian Fu, Wei Wang, Krzysztof Koperski, and Osmar Zaïane. Dmql: A data mining query language for relational databases. 09 1999.

[14] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29(2):1–12, may 2000.

[15] Jochen Hipp and Ulrich Güntzer. Is pushing constraints deeply into the mining algorithms really what we want? an alternative approach for association rule mining. *SIGKDD Explor. Newsl.*, 4(1):50–55, jun 2002.

[16] Arno J. Knobbe and Eric K. Y. Ho. Maximally informative k-itemsets and their efficient discovery. KDD '06, page 237–244, New York, NY, USA, 2006. Association for Computing Machinery.

[17] Arno J Knobbe and Eric KY Ho. Maximally informative k-itemsets and their efficient discovery. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 237–244, 2006.

[18] Matthijs Leeuwen and Antti Ukkonen. Fast estimation of the pattern frequency spectrum. pages 114–129, 09 2014.

[19] Jefrey Lijffijt, Panagiotis Papapetrou, and Kai Puolamäki. A statistical significance testing approach to mining the most informative set of patterns. *Data Mining and Knowledge Discovery*, 28, 12 2012.

[20] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Discovering frequent episodes in sequences. In Usama M. Fayyad and Ramasamy Uthurusamy, editors, *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD'95)*, pages 210–215, United States, August 1995. AAAI Press.

[21] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Efficient mining of association rules using closed itemset lattices. *Information Systems*, 24(1):25–46, 1999.

[22] Jian Pei, Jiawei Han, and L.V.S. Lakshmanan. Mining frequent itemsets with convertible constraints. In *Proceedings 17th International Conference on Data Engineering*, pages 433–442, 2001.

[23] Sunita Sarawagi, Shiby Thomas, and Rakesh Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. SIGMOD '98, page 343–354, New York, NY, USA, 1998. Association for Computing Machinery.

[24] Xuequn Shang, Kai-Uwe Sattler, and Ingolf Geist. Sql based frequent pattern mining with fp-growth. pages 32–46, 01 2004.

[25] Athanasios Tsiamis. Big data essay [unpublished manuscript], thanostsiamis.github.io/assets/pdfs/ Athanasios_Tsiamis_5223652_BigDataEssay.pdf, April 2022.

[26] Jilles Vreeken, Matthijs Leeuwen, and Arno Siebes. Krimp: Mining itemsets that compress. *Data Min. Knowl. Discov.*, 23:169–214, 07 2011.

[27] Geoffrey I. Webb. Self-sufficient itemsets: An approach to screening potentially interesting associations between items. *ACM Trans. Knowl. Discov. Data*, 4(1), jan 2010.