BOUNDLESS DATA ANALYTICS THROUGH PROGRESSIVE MINING

by

QIONG HU

A Dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Tomasz Imieliński

And approved by

_____

_____

_____

_____

New Brunswick, New Jersey

October, 2018

ABSTRACT OF THE DISSERTATION

Boundless Data Analytics through Progressive Mining

By QIONG HU

Dissertation Director:
Tomasz Imieliński

Multidimensional distributions in data mining are often represented as plots: scatter plots between two numerical variables; heat maps, bar graphs, histograms, box plots - they either relate two variables together or show frequency distributions of one variable. What makes one distribution more interesting than the other? What if we could generate all possible relationships and rank the most interesting ones at the top - do it all automatically, thus saving days of repetitive human work?

We define an attribute-value pair from a dimension as a descriptor, and a conjunction of k descriptors is used to slice a dataset. The problem of generating all possible large data slices is formalized as the frequent itemset mining problem. Because the number of dimensions may also include derived dimensions so we do not know ahead of time how long the process will take, may even take an unbounded amount of time. We explore solutions which can answer the following questions: 1) Can we provide some progress indicator during this process? 2) Is the best-so-far partial solution available at any time? To this end, we investigate the anytime algorithms and propose a dynamic approach called ALPINE that allows us to achieve flexible trade-offs between efficiency and completeness. ALPINE is to our knowledge the first algorithm to progressively mine frequent itemsets and closed itemsets support-wise. It guarantees that all itemsets with support exceeding the current

checkpoint's support have been found before it proceeds further. ALPINE runs literally forever without a priori decided minimum support value. The ALPINE approach is also generalized to multiple tables based on the Entity-Relationship Modeling without joining the tables to form a single big table.

Finally, we build a boundless analytics system, which can slice a given dataset in all possible ways and generate very large (unbounded) number of plots. The generated plot objects are organized and indexed in a plot base to support the user queries. A search interface with user-friendly search query language is designed to explore all the plots and the query response are sorted nicely based on some interestingness measure. The system is used to analyze the extensive historical NBA Players stats data with promising results.

Acknowledgements

Without the kind support of many people, I would not be where I am today. First and foremost, I would like to thank my advisor, Prof. Tomasz Imieliński, who has spent an unbounded amount of time and has over 6,000 email conversations with me. I truly appreciate his patience in guiding me, his wisdom and experience in research and life. I deeply impressed by his enthusiasm in research and teaching, his unconventional and creative teaching methods in class and real concerns with students. Prof. Imieliński, thank you for teaching me how to always keep a big picture in mind; thank you for teaching me how to explain and present things in a simple and clear way; thank you for teaching me how to solve a problem systematically; thank you for your boundless support and encouragement!

I would also thank my committee members. I owe my sincere thanks to Prof. Alex Borgida and Prof. Shan Muthukrishnan for their insightful comments, helpful suggestions and valuable discussions on my research. I'm honored to have Prof. Wojciech Szpankowski from Purdue University as my outside member. Thanks Prof. Mubbasir Kapadia for providing constructive remarks regarding my early proposal in my qualification exam.

I'm deeply grateful to the Center for Science of Information, NSF Science and Technology Center, which generously supports my study, project and research in spring 2014 and summer 2017, as well as my conference travel in 2016 CSoI Multidisciplinary Research & Data Science workshop and 2017 SIAM International Conference on Data Mining. The precious experience opens up my mind and gets me connected to more researchers in multiple disciplines. I'm fortunate to be one member of the center.

I would like to express my gratitude to my mentor during my internship with EMC, Core Technology Division, CTO Office, Dr. Kevin Xu. Working with you has become a memorable experience.

I also owe much to my friends and colleagues at Rutgers and elsewhere. I would like to thank members from the DataMan Lab and the Computational Biomedicine and Imaging Modeling Center. Thanks Dr. John Robert Yaros for his help in my early research in this

direction. Thanks Arpit Shah and Apoorv Verma for their kind help with the front end of the boundless analytics system.

Finally, I am much indebted to my family for their years of support. Thanks, Xi, my husband, my lover and my world, who has been through all the ups and downs with me, sharing my tears and smiles in life. Thanks, Vincent, my boy, my sunshine, my source of energy, inspiration and joy, your love makes me strong in moving forward. I am also very appreciative of the support of my parents and parents-in-law, you are the staunch backing of mine forever.

Dedication

To my parents, my husband, and my son

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Multidimensional distributions are often used in data mining to describe and summarize different features of large dataset. Skilled data analysts usually use exploratory statistics and data mining techniques to explore the data before they do any formal modeling. As a result, the multidimensional distributions are represented as plots: scatter plots between two numerical variables; heat maps, bar graphs, histograms, box plots - they either relate two variables together or show frequency distributions of one variable.

A typical workflow of analysts is launching an analytic process, waiting for it to complete, inspecting the results, and then re-launching the computation with adjusted parameters. This manual exploration process is labor-intensive and error-prone for many real-world tasks. What makes one distribution more interesting than the other? What if we could generate all possible relationships and rank the most interesting ones at the top - do it all automatically, thus saving days of repetitive human work?

In this dissertation, we propose a boundless analytics framework, which can harness the computational resources to work for us automatically and isolate only plots which have huge potential interest (the interestingness of plots is measured based on their spread). Firstly, pre-aggregations are calculated to derive new dimensions or facts for the multidimensional dataset. Then a global slicing process and a local plotting process are operated on the

supplemented dataset, which can slice the given dataset in all possible ways and generate very large (unbounded) number of plots.

When an attribute-value pair from a dimension is defined as a descriptor, a conjunction of k descriptors denotes a data slice. The problem of generating all possible large data slices is formalized as the frequent itemset mining problem. Because the number of dimensions may also include derived dimensions so we do not know ahead of time how long the process will take, may even take an unbounded amount of time. How to quickly get those interesting plots? Or can we at least partially analyze the data during such a long mining process without waiting until completion, *i.e.*, first analyzing those larger data slices? Could an algorithm guarantee that all slices with support exceeding the current checkpoint's support have been found before it proceeds further?

To solve this problem, we explore some data mining technique, *i.e.*, progressive itemset mining, which slices the data progressively based on the size or support of a slice, *i.e.*, first generate slices with larger support, then gradually output slices with lower support. At any moment in time, it can guarantee the completeness of the set of data slices in the partial answer up to the previous checkpoint.

As to the progressive paradigm, it should be familiar to computer users, for instance, progressively loading of web apps, where the basic components quickly loaded initially, followed by progressive loading of other UIs when required. By investigating such kind of anytime algorithms, we propose a dynamic approach, ALPINE, that allows us to achieve flexible trade-offs between efficiency and completeness.

ALPINE is to our knowledge the first algorithm to progressively mine all and closed frequent itemsets support-wise. It guarantees that all itemsets with support exceeding the current checkpoint's support have been found before it proceeds further. ALPINE runs literally forever without a priori decided minimum support threshold. For multidimensional analysis, we generalized ALPINE to multiple tables based on the foreign-key reference in entity-relationship modeling without joining the tables to form a single big table.

Finally, this dissertation ends with a boundless analytics system. We build a boundless analytics system which can slice a given dataset in all possible ways progressively and generate very large (unbounded) number of plots. The system can provide a progress meter in terms of the minimum size of the data slices analyzed at any time. The Apache Solr search platform and a proposed user-friendly keywords-based search query language powered the searching module of the system, which can support various user queries to explore the plots in the plot base. It is worth mentioning that this system is demonstrated on a large real-world NBA stats data with over 3000 players for 67 NBA seasons.

1.1 Contributions

In this Section we briefly summarize the major contributions of this dissertation. We classify them according to the area they naturally belong to.

1.1.1 Frequent Itemset Mining

- We define the progressive itemset mining paradigm for long mining tasks, which progressively divide the itemset search space into sub-spaces with respect to a set of decreasing minimum supports. Thus, it guarantees the partial completeness and has the so-called anytime properties.

- We define a compact representation, itemset interval, which utilizes the itemset closure operator to greatly reduce the search space and speed up the mining process.

- We propose a dynamic approach, ALPINE, for itemset mining that allows us to achieve flexible trade-offs between efficiency and completeness, whose theoretic correctness and computational effectiveness are presented in the dissertation. ALPINE is to our knowledge the first algorithm to progressively mine all frequent and closed frequent itemsets “support-wise”. It guarantees that all itemsets with support exceed-

ing the current checkpoint's support have been found before it proceeds further. It automatically lowers the minimum support on-the-fly without any a priori decided minimum support threshold.

1.1.2 Automated Data Science

- We propose the boundless analytics engine, which consists of pre-aggregation, a global slice operator and a local plot operator, for large multidimensional data analysis. It slices a given dataset in all possible ways and generates very large number of plots. It may take an unbound amount of time due to the fact that we can keep deriving new dimensions and facts by aggregation.

- We take advantage of the progressive mining paradigm to progressively slice a large multidimensional database based on the attribute-value combinations from its dimensions. It can achieve flexible compromises between the mining time and the completeness of the generated data slices.

- We employ the E-R modeling approach to analyze multidimensional data across tables. The foreign-key reference based processing is adopted instead of joining all the involving tables to form a single big table.

- We build a boundless analytics system, which includes the data module, the generating module, the indexing module and the searching module. The generating module uses the boundless analytics engine, the indexing module employs the Apache Solr search platform which supports full-text search and the searching module is defined based on bags of keywords.

- We show an application on sport data analysis. The boundless analytics system is applied to analyze the historical NBA Players stats data. Some interesting trends among the player's individual attributes are discovered by the system.

1.2 Dissertation Structure

The remainder of this dissertation is organized as follows. In Chapter 2, we propose the progressive itemset mining problem and design three variant algorithms, APLINE, ALPINEclosed and ALPINE+, for frequent itemset, closed frequent itemset and general non-boolean data mining. Extensive computational evaluations are also included. In Chapter 3, the boundless data analytics is introduced with basic notations and prerequisite information, then each step in the framework, *i.e.*, the pre-aggregation, the slicing and the plotting processes, are explained in detail. Sequentially, a boundless analytics system with an application on NBA sport data analysis is built in Chapter 4, where each module of the system, *i.e.*, the data module, the generating module, the indexing module and the searching module, is introduced as well. Finally, we conclude the dissertation and discuss the future directions in Chapter 5.

Chapter 2

Progressive Itemset Mining

Boundless analytics can generate an unbounded number of plots for large multidimensional dataset, the key is to automatically slice the data into all possible large sub-groups. When an attribute-value pair from a dimension is defined as an item, the data slicing problem can be formulated as the frequent itemset mining problem. As datasets grow and computations become more complex, the execution time of itemset mining becomes critical for many large-scale or time-sensitive applications. We propose a dynamic approach, ALPINE, for itemset mining that allows us to achieve flexible trade-offs between efficiency and completeness. ALPINE is to our knowledge the first algorithm to progressively mine itemsets and closed itemsets "support-wise", which can be generalized to handle richer database types like taxonomic or quantitative data. It guarantees that all itemsets with support exceeding the current checkpoint's support have been found before it proceeds further. Thus, it is very attractive for extremely long mining tasks with very high dimensional data because it can offer intermediate meaningful and complete results. This feature is the most important contribution of ALPINE, which is also fast but not necessarily the fastest algorithm around. Another critical advantage of ALPINE is that it does not require the a priori decided minimum support threshold.

2.1 Introduction

Nowadays, automated data science would encompass attempts to automate the cumbersome data science process, the process of extracting knowledge and insight from data, which are usually represented as 1-D and 2-D plots. Boundless data analytics (Figure 2.1) is such an attempt. Boundless analytics is a mining process with the objective to generate all possible univariate and bivariate relationships or plots with sufficient support and isolate only plots which have huge potential interest.

A boundless analytics engine can slice a given large multidimensional dataset in all possible ways based on its dimensions and derived dimensions and generate very large (unbounded) number of plots. As shown in Figure 2.1, it mainly consists of two crucial steps: (a) globally slice the data into different data slices; (b) locally generate all the plots for each data slice. The most challenge part is step (a), how to efficiently discover all the large slices with sufficient support. That's what we'll solve in this chapter.

If we denote each possible attribute-value pair as an item, then a data slice defined by the conjunction of k descriptors in the form of (*attribute*, *value*) can be represented as a k-itemset. In this way, to find all large data slices from a multidimensional database is exactly the problem to mine all frequent itemsets from a transaction set [1]. Specifically, to mine all the frequent combination of attribute-value pairs in its dimensions.

Because the number of dimensions may also include derived dimensions so we do not know ahead of time how long the process will take, may even take an unbounded amount of time. How to quickly get some interesting plots? Or can we at least partially analyze the data during such a long mining process without waiting until completion, *i.e.*, first analyzing the larger data slices? Could an algorithm guarantee that all slices with support exceeding the current checkpoint's support have been found before it proceeds further?

ALPINE is such an algorithm, it progressively mine frequent itemsets (attribute-value combinations) support-wise. It can be stopped at any time and return the frequent itemsets for some minimal support (that is lower if the analyst waits longer). Therefore, it exhibits so-

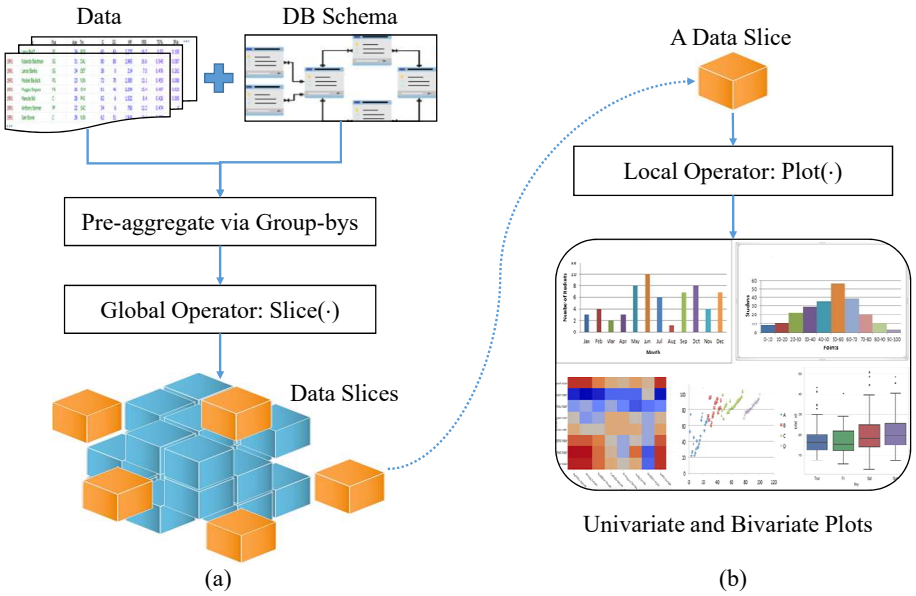


Figure 2.1: Boundless analytics

called *anytime* feature. An *anytime algorithm* uses well-defined quality measures to monitor the progress in problem-solving and is expected to improve the quality of the solution as the computational time increases [75]. Anytime algorithms have been categorized into two types: interruptible and contract algorithms [4]. An interruptible algorithm can be interrupted at any time. A contract algorithm, if interrupted at any point before the termination of the contract time, might not yield any useful results. From this definition, an anytime algorithm is able to return many possible intermediate partial approximate answers to any given input. Thus, it is useful for solving problems where the search space is large and the quality of the results can be compromised [42]. Clearly, the anytime approach is particularly well suited for data mining and more generally for intelligent systems.

In the context of frequent itemset mining, the common framework is to use a *minsup* threshold to ensure the generation of the *correct* and *complete* set of patterns [69]. We require that a progressive mining algorithm reaches partial *completeness* through checkpoints, which define the exploration of well-defined sub-spaces of the entire problem. According to the law of diminishing marginal utility in economics [52], we believe that the additional benefit derived from the completeness of itemsets with a *minsup* diminishes with the

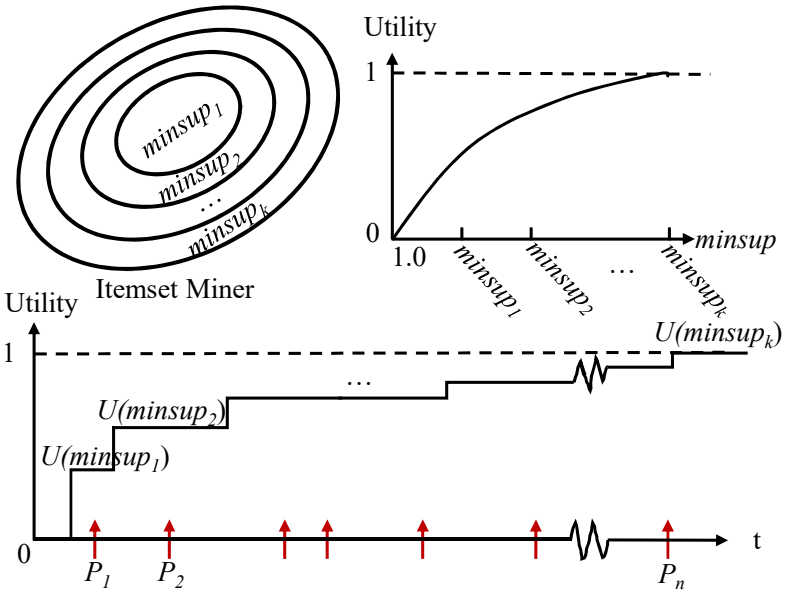


Figure 2.2: Progressive itemset mining. A long mining task, is progressively divided into k sub-search spaces w.r.t. a set of decreasing minimum supports $\{minsup^k_{i=1}\}$. The utility of the current solution is plotted as a function of time, where $\{P^n_{j=1}\}$ is a set of randomly selected probes in time.

decrease of the value of *minsup* (refer to the upper right graph in Figure 2.2). The total utility derived from the outcome of the mining algorithm can be utilized to quantify the usefulness of its intermediate results. The proposed progressive mining framework is illustrated in Figure 2.2, $\{minsup^k_{i=1}\}$ is the set of all distinct itemset supports in a transaction database in decreasing order and $\{P^n_{j=1}\}$ is a set of randomly selected probes in time. Upon the completion of all the itemsets with support greater than or equal to $minsup_i$, the utility associated with the minimum support, $U(minsup_i)$, is obtained instantly. The goal of a progressive miner is to maximize the utility at anytime. Mathematically, we want to maximize the average utility at a set of random probes, i.e., $\max \frac{1}{n} \sum_{j=1}^{n} U(P_j)$.

In this chapter, we propose a dynamic approach for itemset mining and present the ALPINE algorithm [36, 37], namely, Automatic *minsup* Lowering with Progress Indicator in practically "Never-Ending" mining. The ALPINE algorithm proceeds progressively from checkpoint to checkpoint. In ALPINE, the checkpoints correspond to decreasing values of *minsup*s. It might be generalized to other anti-monotone interestingness measures. ALPINE guarantees that all itemsets with support exceeding the current checkpoint's support have

been found before it proceeds further. In this way, we know that we have completed a well-defined subset of the overall, potentially enormous search space.

ALPINE proceeds in this "monotonic" manner with minimal computational overhead as compared to the best existing frequent itemset mining algorithms. In ALPINE, though the mining process is continuous, it does not go totally unchecked. ALPINE can be stopped at any point and we will always be able to offer partial conclusions based on the last checkpoint reached as indicated in Figure 2.2. In contrast, the traditional itemset mining algorithms do not give any intermediate partial completeness guarantees, requiring the user to wait until completion to get any definite result.

There is another very critical advantage of ALPINE. It does not require setting the minimum support a priori. This requirement has always been problematic for all frequent itemset generation algorithms. How do we set the minimum support if we do not know the data? We only learn the data as we continue mining it, but then it is too late to change the value of minimum support. However, it is not the case for ALPINE. ALPINE moves the minimum support as it goes ahead from checkpoint to checkpoint.

ALPINE is, to our knowledge, the first algorithm to progressively mine frequent itemsets and closed frequent itemsets "support-wise". It guarantees that all itemsets with support exceeding the current checkpoint's support have been found before it proceeds further.

Firstly, related work in the literature and some basic definitions and notations will be introduced in Section 2.2 and Section 2.3, respectively. Then we describe the details of different variants of the ALPINE algorithm in Section 2.4. In Section 2.5, extensive experiments are carried out to evaluate this progressive approach and analyze the added value of the anytime feature of the algorithm. We analyze the effectiveness and efficiency of the algorithm in comparison with: 1) the benchmarking approach; 2) sequential top-k mining algorithm, *i.e.*, Seq-Miner [43]; and 3) frequent itemset mining algorithm. Finally, we'll conclude this chapter in Section 2.6.

2.2 Related Work

2.2.1 Frequent itemset mining

Itemset mining is a popular research area with wide applications in domains like bioinformatics [46], text mining [74], product recommendation [44], e-learning [5, 51], web click stream analysis [41], etc. The traditional task of frequent itemset mining is to discover groups of items (more generally, attribute-value pairs) that frequently co-occurring in databases. Finding all frequent patterns from large databases is NP-hard for it's an exhaustive search problem [72].

A lot of algorithms have been proposed to mine itemsets in the past decade [2, 17, 18, 32, 48, 65, 71], the key is how to efficiently reduce the search space. Apriori-like methods utilize the anti-monotone property to prune candidates [2, 71], FP-growth family employs some highly condensed data structure, such as FP-tree [32] or PPC-tree [18], to confine the search space, while PrePost+ [17] introduces the children-parent equivalence pruning strategy. However, the pruning might be incomplete. Thus, the closure operator $I(\mathcal{T}(\cdot))$ is incorporated in other algorithms [48, 65, 66] as we do. In[48], duplicated closed itemset may be generated. The most similar work to ours is the LCM algorithm [65, 66], which also transverses a tree composed of closed itemsets. However, LCM requires to set the *minsup* threshold and no completeness guarantees are given for its intermediate results.

In summary, all existing frequent itemset mining algorithms [12, 23]: 1) require the user to specify a fixed minimum support threshold in advance, 2) do not give any definite feedback to the user while they are running. The high response time and the need to guess the optimal parameter setting limit the performance of current data mining technologies.

2.2.2 Top-k mining

In "concept mining", the top-k mining can gradually raise the *minsup* to mine "the k-most interesting patterns" without specifying a *minsup* threshold in advance to increase the us-

ability of a data mining algorithm. Shen *et al.* [54] first introduced the top-k mining problem to generate an appropriate number of most interesting itemsets. The Itemset-Loop/Itemset-iLoop algorithm [15] based on Apriori [2] and the TFP algorithm [69] extending the FP-growth method [32] are developed to mine the k-most interesting patterns thereafter.

Generally, these algorithms follow the same process: Initially, the *minsup* threshold is set to 0 to ensure no pattern will be missing, then the *minsup* is gradually raised by the algorithm to prune the search space until top-k patterns are found. When k is too large, mining takes an unacceptably long time; on the contrary, when k is too small, it will miss a lot of potential interesting patterns. The problem of setting up the value of *minsup* is now replaced with setting the value of k. Thus, Hirate *et. al.* [35] propose the TF^{2}P-growth algorithm to mine the top-k patterns sequentially without any thresholds.

TF^{2}P-growth outputs every top n_c patterns, where n_c is some user-defined chunk size. For instance, $n_c = 1000$, it sequentially returns exactly the top 1000, 2000, 3000 patterns *etc*. For n_c is a user specified number, it might not return all itemsets having the same support as the last one. Minh *et al.* [43] overcame this shortcoming and proposed an improved algorithm, the Seq-Miner. These methods have a flavor of the contract-type anytime algorithms [75], but they can not be interrupted before the termination of every top n_c patterns. In contrast, ALPINE monotonically explores itemsets with descending values of support and mines continuously from checkpoint to checkpoint, which guarantees the quality of the partial results are checked at any time.

2.2.3 Pattern sampling

Zhang *et al.* [73] used sampling and incremental mining to support multiple-user inquiries at any time. Boley *et al.* [10, 11] proposed to use Metropolis-Hastings sampling for the construction of data mining systems that do not require any user-specified threshold, *i.e.*, $minsup$ or $minconf$. However, all the algorithms generate approximate results and the completeness cannot be guaranteed.

2.3 Preliminaries

Let $\mathcal{I} = \{i_1, i_2, .., i_n\}$ be a set of literals, called *items*, and $\mathcal{T} = \{t_1, t_2, ..., t_m\}$ be a *transaction database*, where each transaction t_k $(k = 1, 2, ..., m)$ in $\mathcal{T}$ is a set of items such that $t_k \subseteq \mathcal{I}$. A unique transaction identifier, tid, is associated with each transaction. Each nonempty subset of $\mathcal{I}$ is called an *itemset* and a transaction is said to contain an itemset if all the items in the itemset are present in the transaction [1, 2]. For an itemset X, the *cover* $\mathcal{T}(X) = \{t \in \mathcal{T} | X \subseteq t\}$ (a $tidset$) be the set of transactions it is contained in and the support of X, denoted as $sup(X)$, is the number of these transactions. Hence, $sup(X) = |\mathcal{T}(X)|$ [12]. If $sup(X)$ exceeds a minimum support threshold *minsup*, then X is called a *frequent itemset* [2, 12]. For a transaction set $\mathcal{S} \subseteq \mathcal{T}$, its intersection is $I(\mathcal{S}) = \cap_{T \in \mathcal{S}} T$. If an itemset X satisfies $I(\mathcal{T}(X)) = X$, then X is called a *closed itemset* [48].

To avoid enumerating itemsets with duplications, it's natural to order the items to structure the search space. We define a total order among the set of items such that item $i <$ item j iff $sup(i) < sup(j)$. The search is confined to extend an itemset only with items greater than all the items inside it. All the items from $\mathcal{I}$ can be recoded to 0, 1, ..., $|\mathcal{I}| - 1$ according to this order. Let $X = \{x_1, ..., x_n\}$ be an itemset as an ordered sequence such that $x_1 < ... < x_n$, the *tail* of X is $tail(X) = x_n$ [65]. Itemset X will only be extended with all items greater than $tail(X)$, resulting in a tree structured lattice.

To further reduce the search tree size, the closure operator $I(\mathcal{T}(\cdot))$ is utilized at each step. Together with the above extension rule, we define the *closure* of itemset X as $X^* = X \cup E$, where $E = \{e \in \mathcal{I} | e \in I(\mathcal{T}(X)) \wedge e > tail(X)\}$, for we know that X union any subset of E is supported exactly by $\mathcal{T}(X)$ (E is a shortcut of support equivalence extension). Thus X does not need to be extended with items belonging to E. If $|E| = k$, this operation will reduce the size of the sub-tree rooted at X by a factor of 2^k. In general, for itemset P and Q, with $P \subseteq Q$ *and* $\mathcal{T}(P) = \mathcal{T}(Q)$, the set of all itemsets Y which is a superset of P and a subset of Q can be compactly represented as an *itemset interval*: $(P, Q) = \{Y | P \subseteq Y \subseteq Q\}$,

for they all share the same supporting transactions as P. In this definition, P and Q specify the minimum itemset and the maximum itemset in an itemset interval, respectively.

We also define a mapping, sup^{-1}, from support to set of itemsets, which is applicable to both frequent and closed itemsets. We name it *support index*, for it is used to index all the itemsets from $\mathcal{T}$ by support. Given a support s, the indexed set of itemsets is $sup^{-1}(s) = \{X | sup(X) = s\}$. The degree of completeness for a specific support s, $D_c(s)$, is defined as the number of itemsets discovered so far with support s divided by the total number of itemsets with support s from the transaction database, i.e., $|sup^{-1}(s)|$.

2.4 The Algorithm

The proposed progressive itemset miner (Figure 2.2) requires to systematically explore the itemset search space. In ALPINE, itemsets are discovered in order of their supports – from higher support to lower support. ALPINE will automatically lower the *minsup* threshold to the next possible, lower value and continue mining. The basic idea of ALPINE is to dynamically build the support index, from the highest possible support gradually to the lowest possible value of support in the given transaction database. It *progressively* partitions itemset intervals into disjoint bins of different supports.

ALPINE starts with the index built from all the itemset intervals (I, I^*) of singleton itemsets I from a transaction database in Figure 2.3a. In this figure, all singleton itemset intervals are sorted in decreasing support from left to right and binned based on their support values. This index is not static, though, it is updated by new itemset intervals generated by extending the minimum itemset of an itemset interval. At any point in time, we are working on the uncompleted bin (there are still some remaining itemset intervals in this bin) with the highest support value s. To enumerate all itemsets with support above or equal to s, we extend the minimum itemset of each itemset interval in this bin. Let (P, Q) be an itemset interval we are currently working on, we extend P with all items j greater than $tail(P)$

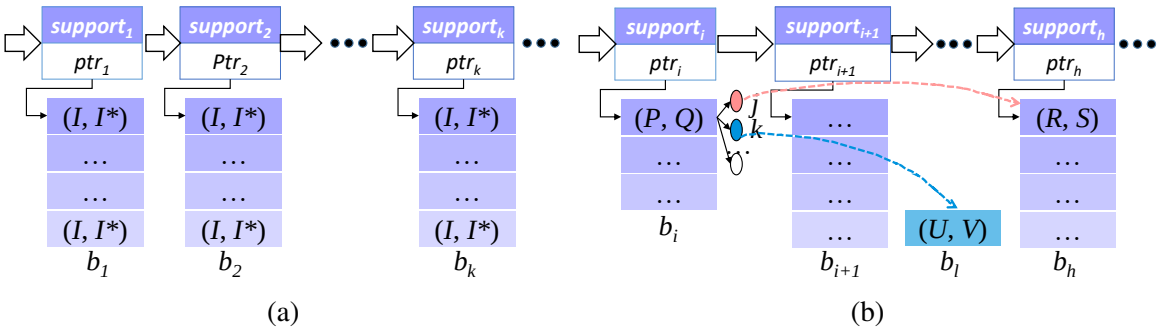


Figure 2.3: Dynamically index itemset intervals by their supports in ALPINE

and not contained in Q. Denote $P \cup \{j\}$ as itemset R, we also find R^* according to the definition of an itemset closure. We can prove Lemma 2.4.1 and get the itemset interval for R accordingly: (R, S), where $S = R^* \cup Q$.

Lemma 2.4.1 *All itemsets in the itemset interval* (R, S)*, where* $S = R^* \cup Q$*, are supported by* $\mathcal{T}(R)$*.*

Proof $\forall$ *itemset* $X \in$ *itemset interval* (R, S)*, we can denote* X *as* $X = R \cup Y$*, where* $Y \subseteq S \setminus R$*. For* $S = R^* \cup Q$*, we can further decompose* Y *into two disjoint subsets* Y_R *and* Y_Q*, with* $Y_R \subseteq R^*$ *and* $Y_Q \subseteq Q \setminus R^*$*. Then, the cover* $\mathcal{T}(X) = \mathcal{T}(R \cup Y_R \cup Y_Q)$*. According to the definition of* R^**, we have* $\mathcal{T}(R \cup Y_R) = \mathcal{T}(R)$*, so* $\mathcal{T}(X) = \mathcal{T}(R \cup Y_Q)$*. Since* $R = P \cup \{j\}$*, then* $\mathcal{T}(X) = \mathcal{T}(P \cup \{j\} \cup Y_Q)$*. The itemset* $P \cup Y_Q$ *is in the itemset interval* (P, Q)*, so it is supported by* $\mathcal{T}(P)$*. Thus,* $\mathcal{T}(X) = \mathcal{T}(P \cup \{j\}) = \mathcal{T}(R)$*.*

All of the newly generated itemset intervals (R, S) are segregated by their support values into different bins of the support index. Admittedly, $sup(R)$ will always be smaller than $sup(P)$ according to the closure operation, otherwise, item j will belong to Q. There are two situations: 1) the support of R is already associated with some existing bin and we only need to add itemset interval (R, S) to that bin; 2) the support of R is a new value, which hasn't been indexed yet and we need to create a new bin to place (R, S) in it. An example is given in Figure 2.3b, the itemset interval under exploration is on top of b_i and $support_i$ corresponds to the highest uncompleted bin. We extend P with items greater than $tail(P)$

and not contained in Q. Suppose item j and item k belong to such set of items. When P is extended with item j, we obtained itemset $R = P \cup \{j\}$ and $S = R^* \cup Q$. It happens that $sup(R)$ is equal to some $support_h$ indexed, so the itemset interval (R, S) will be added to b_h as indicated by the red dashed line. However, when it comes to item k, the support of $U = P \cup \{k\}$ has not been indexed yet, a new bin b_l is created to place the itemset interval (U, V), with $V = U^* \cup Q$.

Only when we finished exploring all itemset intervals in a bin, we are safe to conclude that we have discovered all itemsets with support above or equal to the support associated with the bin. That's when we can declare that support as a new minimum support, *minsup*, and we have reached the subsequent checkpoint. This checkpoint completes a subspace of all itemsets with support above or equal to *minsup*, even though the bins with lower supports are not complete yet. ALPINE always continues to build new bins for new possible supports or extends existing bins. In this way, ALPINE mines patterns with descending supports and outputs partially complete information from checkpoint to checkpoint.

In summary, APLINE initializes the support index from all itemset intervals of the singleton itemsets in a given transaction database, then it repeatedly works on the uncompleted bins of successively lower supports until the support index is *complete*. ALPINE explores all itemset intervals in a bin one by one as exemplified by Figure 2.3b, in which the minimum itemset of each itemset interval is extended with all possible items in descending support order. A checkpoint is reached after exploring all itemset intervals in a bin. The ALPINE algorithm can be interrupted at any moment (that is, it retains the *anytime* feature). Based on the description, we can deduce the following lemma and observations:

Lemma 2.4.2 *Each itemset will be output in exactly one itemset interval.*

Proof *This lemma can be proved by contradiction. Suppose the same itemset X can be output in two different itemset intervals: (P_1, Q_1) and (P_2, Q_2) with $P_1 \neq P_2$. Then we have, $P_1 \subseteq X \subseteq Q_1$ and $P_2 \subseteq X \subseteq Q_2$ and $\mathcal{T}(P_1) = \mathcal{T}(P_2)$ (*). Thus, P_1 and P_2 cannot be inclusion relation with each other, otherwise, $\mathcal{T}(P_1) \neq \mathcal{T}(P_2)$. If we sort the items in P_1*

and P_2 as ordered sequence according to the order defined in preliminaries and denote the first different item between them as x_{1i} and x_{2i} ($x_{1j} = x_{2j}, for\ j = 0, 1, ..., i-1$, and we denote this set of items as P_0 and the associated itemset interval as (P_0, Q_0). Without loss of generality, we can assume $x_{1i} < x_{2i}$. From (*), we have $x_{1i} \in X$ and $x_{2i} \in X$. Consider itemset $R_1 = P_0 \cup \{x_{1i}\}$, since the itemset is only extended with items not in Q_0, we have $x_{1i} \notin Q_0$. Regarding itemset $R_2 = P_0 \cup \{x_{2i}\}$, $S_2 = R_2^* \cup Q_0$. According to the definition of R_2^*, it doesn't include any item less than x_{2i}, so x_{1i} must be in Q_0, which is a contradiction.

Observation 2.4.1 *Every itemset with support above or equal to some minimum support has been output at the related checkpoint.*

With the defined total order among items in Section 2.3 and the exploration procedure in Figure 2.3b, each item greater than $tail(P)$ is either extended explicitly, or it has already been contained in Q, which completes the search space for a specific bin. All itemsets generated from the itemset intervals in the bins with lower supports are less than the minimum support of the current checkpoint according to the anti-monotone property of itemset support.

Observation 2.4.2 *Every distinct support count of an itemset in the transaction database $\mathcal{T}$ will be minsup value of some ALPINE's checkpoint.*

This observation is readily obtained from the monotonic manner the ALPINE algorithm explores successive bins of the support index. Thus, ALPINE will proceed "support-wise" during the mining process. It is not hard to verify the following two properties:

Property 2.4.1 *Given a transaction database $\mathcal{T}$ with m transactions over n items, the number of checkpoints from it is bounded by $min\{2^n, m\}$.*

Property 2.4.2 *The minimum support of the first checkpoint is equal to the highest support of the singleton itemsets in $\mathcal{T}$.*

Remark 2.4.1 *The highest support value which will correspond to the first and highest checkpoint for ALPINE is equal to the support of the most frequent item. If no other*

item shares that support, that item alone constitutes the first checkpoint. In this case, the subspace of itemsets corresponding to the first checkpoint has just one singleton set - that most frequent item.

Remark 2.4.2 *It may be the case that several items share the same, highest support. In such case, ALPINE needs to do more work to reach the first checkpoint. In the extreme case, these top support items may be perfectly correlated (that is, all their combinations also have the same support). This is, of course, unlikely but possible. Suppose there are k such items $i_{\pi_1}, i_{\pi_2}, ..., i_{\pi_k}$ sharing the highest value of support, and the item order among them is $i_{\pi_1} < i_{\pi_2} < ... < i_{\pi_k}$. Then ALPINE needs to explore and output these k itemset intervals before reaching the first checkpoint:* $(\{i_{\pi_1}\}, \{i_{\pi_1} i_{\pi_2} .. i_{\pi_k}\})$, $(\{i_{\pi_2}\}, \{i_{\pi_2} i_{\pi_3} .. i_{\pi_k}\})$, ..., $(\{i_{\pi_{k-1}}\}, \{i_{\pi_{k-1}} i_{\pi_k}\})$, $(\{i_{\pi_k}\}, \{i_{\pi_k}\})$, *even though all the k itemset intervals are contained in ONE closed itemset* $\{i_{\pi_1} i_{\pi_2} .. i_{\pi_k}\}$.

In this situation, it might be beneficial to confine the tree-shaped transversal routes of ALPINE to only closed itemsets, which can reduce the work for the aforementioned extreme case to explore only one itemset interval.

2.4.1 The ALPINE Algorithm for Frequent Itemset Mining

Now let us work through an example in Table 2.1 (a:$\{1,3,5\}$, b:$\{2,3,4,5\}$, c:$\{1,2,3,5\}$,d:$\{1\}$, e:$\{2,3,4,5\}$) to see how the ALPINE algorithm works for frequent itemset mining. The total order among items in $\mathcal{T}$ is defined in Section 2.3, *i.e.*, item $e >$ item $c >$ item $b >$ item $a >$ item d, and the support index $\mathcal{S}$ is initialized from all itemset intervals of the singleton itemsets as shown in Fig. 2.4a. In the support index $\mathcal{S}$, the highest support value is 4, and in the corresponding indexed bin, itemset interval $(\{e\}, \{e\})$ is on the top (highlighted in red), that's where the search starts. In the same figure, we also plot the subset lattice on the right to show the exploration process.

Table 2.1: An example transaction set $\mathcal{T}$. (a) the horizontal representation; (b) the vertical representation.

(a)

tid	*items*
1	{a, c, d}
2	{b, c, e}
3	{a, b, c, e}
4	{b, e}
5	{a, b, c, e}

(b)

item	*tidset*
a	{1, 3, 5}
b	{2, 3, 4, 5}
c	{1, 2, 3, 5}
d	{1}
e	{2, 3, 4, 5}

For item e is the highest ordered item on the total order chain, no further extension is possible after reporting it (indicated by the thick green bounding box in Fig. 2.4b). So the algorithm continues to the next itemset interval $(\{c\}, \{c\})$. The minimum itemset $\{c\}$ in the interval is extended with item e to itemset $\{ce\}$ (indicated by the red arrow), and the corresponding itemset interval is calculated according to Lemma 2.4.1, yielding to $(\{ce\}, \{ce\})$ with support 3. As support 3 has already been indexed in the support index $\mathcal{S}$, highlighted by blue in the figures, we add the itemset interval $(\{ce\}, \{ce\})$ into the indexed bin for support 3 (Figure 2.4c).

Now the algorithm continues with the itemset interval $(\{b\}, \{be\})$, though both item c and item e have higher order than item b, for item e is included in the interval, we only need to extend the minimum itemset $\{b\}$ with item c to get $(\{bc\}, \{bc\boldsymbol{e}\})$ with support 3 in Figure 2.4c. Note that item e here is inherited from the maximum itemset $\{be\}$. We also put the newly generated itemset interval into the bin with support 3. Now, the indexed bin for support = 4 is empty, which means no more itemset intervals having support ≥ 4. The subspace for support ≥ 4 is completed and we are safe to issue this guarantee at this point.

Then the algorithm automatically lowers the minimum support to the next possible value, 3, and continues. It fetches itemset interval $(\{a\}, \{ac\})$ at the top of the bin with support 3 (see Figure 2.4d). Similarly, item c with a higher order than item a is neglected, for it has already been included in $\{ac\}$. So we extend the minimum itemset $\{a\}$ with item e then with item b. When itemset $\{a\}$ is extended with item e, we get the itemset interval $(\{ae\}, \{a\boldsymbol{c}e\})$, where item c is inherited from the original itemset interval. When itemset

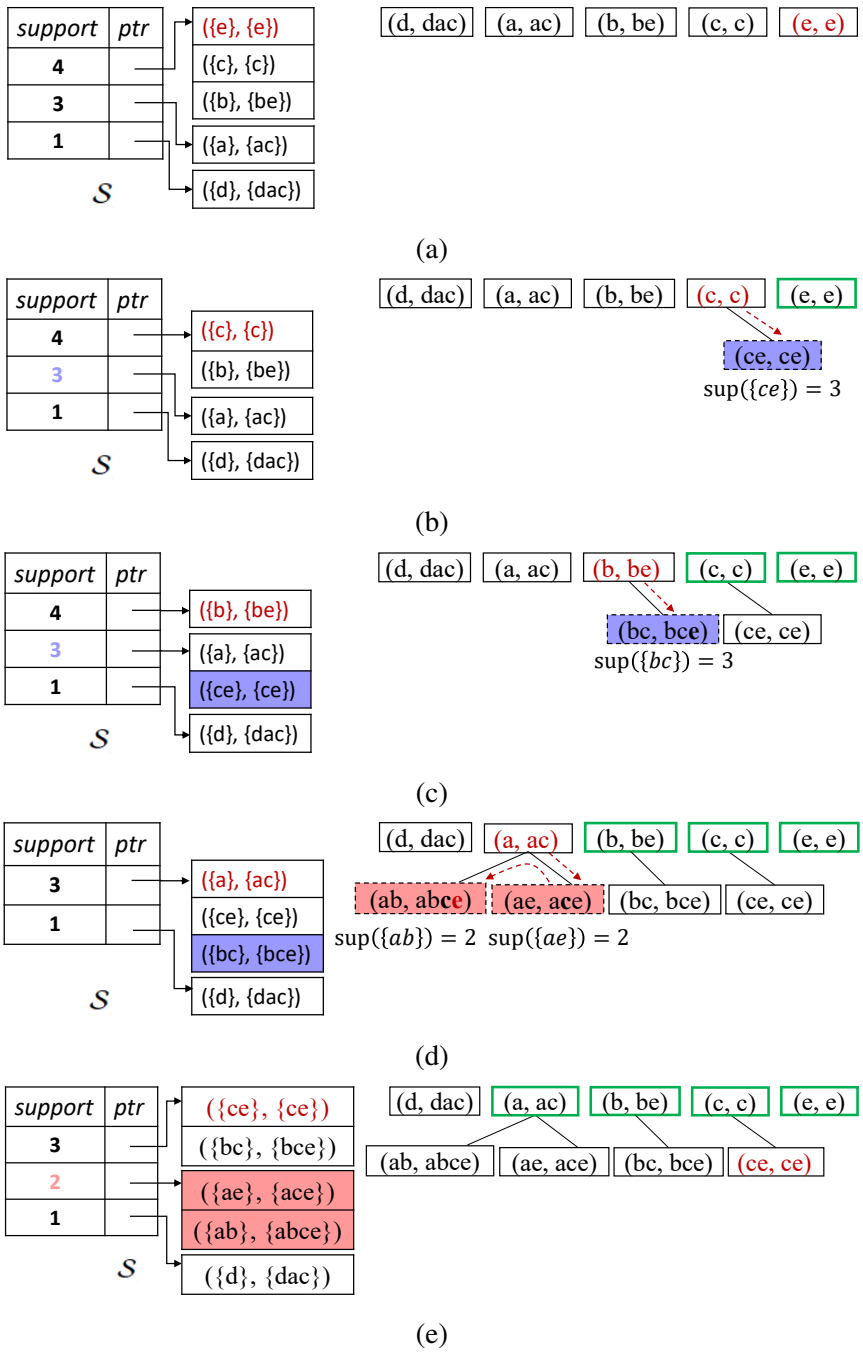


Figure 2.4: The first few steps of the ALPINE algorithm on the example transaction set $\mathcal{T}$ for frequent itemset mining

$\{a\}$ is extended with item b, we get $(\{ab\}, \{abce\})$, where item e is added due to the fact that it is in the closure of itemset $\{ab\}$.

However, both itemset interval $(\{ae\}, \{ace\})$ and $(\{ab\}, \{abce\})$ have support equal to 2, which has not been indexed in the support index yet. We create a new bin with this

Algorithm 1 ALPINE(Transactionset $\mathcal{T}$, Support index $\mathcal{S}$)

```
1:  Init S by all itemset interval (I, I*) of singleton itemset I in T;
2:  repeat
3:     Get the bin b with the highest support s from S;
4:     for all itemset interval (P, Q) in b do
5:        Explorer(P, Q, S);
6:     end for
7:     Declare s to be minsup;
8:     Issue checkpoint: complete subspace of all itemsets ≥ minsup;
9:     if toTerminate == true then
10:       break; // terminate requested by user
11:    end if
12: until the support index is complete
```

Algorithm 2 Explorer(Minimum itemset P, Maximum itemset Q, Support index $\mathcal{S}$)

```
1:  Output itemset interval: (P, Q);
2:  for all item j = |I| − 1; j > tail(P); j−− do
3:     if j ∈ Q then
4:        continue; // no need to extend with item ∈ Q
5:     end if
6:     R ← P ∪ {j};
7:     S ← R* ∪ Q;
8:     if sup(R) is already indexed in S then
9:        Add (R, S) to the indexed bin;
10:    else
11:       Create a new bin with support sup(R) for S and add (R, S) to it;
12:    end if
13: end for
```

support and put them into this bin (see Figure 2.4e). The similar procedure continues until we finished building the full support index for this transaction set.

We can summarize the pseudo-code of our prototypical algorithm - ALPINE (Algorithm 1) from this example. It starts from the bin of the highest support and it explores all itemset intervals in that bin one by one. Then it continues with the bins of successively lower support values. The initialization process in Line 1 initializes the support index from all the itemset intervals of the singleton itemsets in the given transaction database $\mathcal{T}$ in decreasing support order. Each itemset interval in a given bin is explored by the sub-routine Explorer given in Algorithm 2 (Line 5), in which the minimum itemset of each itemset interval is

extended with all possible items in descending support order from the most promising one to the least promising one. Line 8 issues a checkpoint after completing all itemset intervals in a bin (with support above or equal to the support of that bin). The anytime feature makes the ALPINE algorithm can be interrupted at any moment (Line 9 - 11).

2.4.2 The ALPINEclosed Algorithm for Closed Itemset Mining

The ALPINE algorithm elaborated in Section 2.4.1 can build the full support index of all itemsets from a transaction database $\mathcal{T}$ in decreasing support order. If we denote $\mathcal{F}$ and C the sets of all frequent itemsets and all closed frequent itemsets, respectively. According to their definitions, we know that C is a subset of $\mathcal{F}$. Thus, a straightforward way to adapt ALPINE for closed itemset mining is to add the closeness check for the maximum itemset of an itemset interval at each step of the mining process.

Observation 2.4.3 *Let (P, Q) be an itemset interval, then Q is closed if and only if $Q = I(\mathcal{T}(P))$.*

From the definition of itemset interval, we have Q is supported exactly by $\mathcal{T}(P)$, *i.e.* $\mathcal{T}(Q) = \mathcal{T}(P)$. Thus, $Q = I(\mathcal{T}(P)) = I(\mathcal{T}(Q))$. According to the definition of closed itemset given in Section 2.3, Q is a closed itemset.

For singleton itemset I, we know its itemset interval is (I, I^*). The closeness condition in Observation 2.4.3 for I^* is violated if and only if there exists some item $e < tail(I)$ such that e occurs in every transaction in $\mathcal{T}(I)$. For an itemset interval (R, S) generated from some itemset interval (P, Q) in the intermediate stages of ALPINE, where $S = R^* \cup Q$, things are slightly different. To fail the closeness check, there must exist some item $e < tail(R)$ and $e \notin R$ satisfies: (1) e is shared by every transaction of $\mathcal{T}(R)$; (2) $e \notin Q$. In other words, for closed itemset S, item $e < tail(R)$ and $e \notin R$ and $e \in I(\mathcal{T}(R))$ can only be obtained from itemset Q.

Algorithm 3 ALPINEclosed(Transactionset $\mathcal{T}$, Support index $\mathcal{S}$)

```
1: Init S by itemset intervals (I, I*) of singleton itemset I in T s.t. I* = I(T(I)) (closed);
2: repeat
3:    Get the bin b with the highest support s from S;
4:    for all itemset interval (P, Q) in b do
5:       Explorer2(P, Q, S);
6:    end for
7:    Declare s to be minsup;
8:    Issue checkpoint: complete subspace of all closed itemsets ≥ minsup;
9:    if toTerminate == true then
10:      break; // termination requested by user
11:   end if
12: until the support index is complete
```

Algorithm 4 Explorer2(Minimum itemset P, Maximum itemset Q, Support index $\mathcal{S}$)

```
1: Output closed itemset: Q;
2: for all item j = |I| − 1; j > tail(P); j−− do
3:    if j ∈ Q then
4:       continue; // no need to extend with item ∈ Q
5:    end if
6:    R ← P ∪ {j};
7:    S ← R* ∪ Q;
8:    if S = I(T(R)) (closed) then
9:       if sup(R) is already indexed in S then
10:         Add (R, S) to the indexed bin;
11:      else
12:         Create a new bin with support sup(R) for S and add (R, S) to it;
13:      end if
14:   end if
15: end for
```

With this observation, we can modify the ALPINE algorithm given in Section 2.4.1 to the ALPINEclosed algorithm (Algorithm 3). Note that in Line 8 of Algorithm 4, to test the closeness of itemset S, we only need to check all the items $k < tail(R)\ and\ k \notin R\ and\ k \notin Q$. For any nonempty closed itemset $S \in C$, its parent is always defined and belongs to C [65]. This guarantees the completeness of the ALPINEclosed algorithm in mining all closed frequent itemsets. The difference of the ALPINE and ALPINEclosed algorithm lies in:

- Initialization: ALPINE is initialized with all itemset intervals of singleton itemsets from a database $\mathcal{T}$ (Line 1 of Algorithm 1). In contrast, ALPINEclosed is initialized with those itemset intervals passing the closeness test (Line 1 of Algorithm 3).

- Exploration: ALPINE outputs itemset interval and keeps all newly generated itemset intervals (R, S) in the mining process (Algorithm 2). On the other hand, ALPINEclosed only outputs closed itemset and maintains itemset intervals (R, S) meeting the closeness condition (Algorithm 4).

- Status report: ALPINE reports the completion of the subspace of all frequent itemsets above the current *minsup* (Line 8 of Algorithm 1). Instead, ALPINEclosed issues checkpoint about finishing all closed frequent itemsets above the current *minsup* (Line 8 of Algorithm 3).

2.4.3 The Generalized ALPINE+ Algorithm for Non-Boolean Data

Conceptually, the ALPINE and the ALPINEclosed algorithm mine itemsets from relational tables where the attributes are boolean: the value of an attribute for a given record is "1" if the item corresponding to the attribute is present in the transaction corresponding to the record, "0" else. The problem is most data from many real-life applications besides the standard market-basket analysis, like in the business and scientific domains, is non-boolean. It has richer attribute types, which can be taxonomic [57] or quantitative [58]. A pre-processing can be done to make the data into boolean. Or we can generalize the ALPINE algorithm to mine itemsets from those non-boolean datasets, namely, the ALPINE+ algorithm.

All the generic individual items from a transaction set, i.e., taxonomic items are those at the top-most concept level (a set) and quantitative items are those with the largest possible interval (an interval), are sorted in decreasing support. This defines the total order among the generic items. ALPINE+ starts with the index built from all the itemset intervals of

those generic individual items and bins them based on their support values. This index is also dynamically maintained.

At any point in time, we are working on the uncompleted bin with the highest support value s. To enumerate all itemsets with support above or equal to s, we process each itemset interval in the bin in two directions: 1) extending the minimal itemset of the itemset interval with all generic items greater than the tail of this itemset if exist; 2) finding the most frequent divisible item inside the minimum itemset, either going to a lower concept level if it is taxonomic or dividing it into sub-intervals if it is quantitative.

For a quantitative attribute, it is partitioned into intervals (can be a single value when the lower and the upper bound of an interval is the same) and mapped to consecutive integers, such that the order of the intervals is preserved. Suppose the interval of a quantitative attribute consists of n ordered elements ($e_1 < e_2 < ... < e_n$), each time we divide it into two sub-intervals, $[e_1, e_{n-1}]$ and $[e_2, e_n]$ by eliminating one element.

This divide approach is applied recursively until an interval only has one element. To avoid duplication of the generated sub-intervals, we confine to keep both sub-intervals if the smallest element e_1 is inside the original interval, otherwise, we only keep the sub-interval with the larger lower bound. For instance, the interval $[e_1, e_{n-1}]$ is divided into $[e_1, e_{n-2}]$

Algorithm 5 ALPINE+(Transactionset $\mathcal{T}$, Support index $\mathcal{S}$)

```
1: Init S by all itemset interval (I, I*) of singleton itemset I from the generic individual
   items in T;
2: repeat
3:    Get the bin b with the highest support s from S;
4:    for all itemset interval (P, Q) in b do
5:       Explorer3(P, Q, S);
6:    end for
7:    Declare s to be minsup;
8:    Issue checkpoint: complete subspace of all itemsets ≥ minsup;
9:    if toTerminate == true then
10:      break; // terminate requested by user
11:   end if
12: until the support index is complete
```

Algorithm 6 Explorer3(Minimum itemset P, Maximum itemset Q, Support index $\mathcal{S}$)

Output itemset interval: (P, Q);
$N \leftarrow \emptyset$
// Find all possible extensions for P
if P consists of only generic items **then**
  for all generic item $j = |\mathcal{I}| - 1 \wedge j \notin Q; j > tail(P); j--$ **do**
    $R \leftarrow P \cup \{j\}$;
    $S \leftarrow R^* \cup Q$;
    $N \leftarrow N \cup \{(R, S)\}$
  end for
end if
//Divide the most frequent divisible item to the next concept level or sub-intervals
if itemset P is divisible **then**
  Identify the most frequent divisible item i from P
  if item i is taxonomic **then**
    Access the next lower level in the concept hierarchy of item i: $L = \{it_1, it_2, ..., it_k\}$
    for all items it in L **do**
      $N \leftarrow N \cup \{(it \cup (P \setminus i), (it \cup (P \setminus i))^* \cup Q)\}$
    end for
  else if item i is quantitative **then**
    Divide item i into sub-intervals $subI$ (do not keep duplicate sub-intervals)
    for all sub-interval ii in $subI$ **do**
      $N \leftarrow N \cup \{(ii \cup (P \setminus i), (ii \cup (P \setminus i))^* \cup Q)\}$
    end for
  end if
end if
// Index the newly generated itemset intervals
for all itemset interval (P', Q') in N **do**
  if $sup(P')$ is already indexed in $\mathcal{S}$ **then**
    Add (P', Q') to the indexed bin;
  else
    Create a new bin with support $sup(P')$ for $\mathcal{S}$ and add (P', Q') to it;
  end if
end for

and $[e_2, e_{n-1}]$, we keep both intervals for e_1 is included in it; while the interval $[e_2, e_n]$ is divided into $[e_2, e_{n-1}]$ and $[e_3, e_n]$, we only keep the sub-interval $[e_3, e_n]$ to avoid explore sub-interval $[e_2, e_{n-1}]$ twice.

Let (P, Q) be an itemset interval we are currently working on. When P only consists of generic items, we extend P with all possible generic items j greater than $tail(P)$ and not contained in Q. The corresponding itemset intervals for the newly generated itemsets

are obtained according to Lemma 2.4.1. At the same time, we identify one most frequent divisible item i from P (if exists) and go to the next concept level if item i is taxonomic or divide it into sub-intervals if item i is quantitative, then union it with the rest part of P, *i.e.*, $P \setminus i$, to generate the new itemset interval.

All of the newly generated itemset intervals (P', Q') are segregated by their supports into different bins of the support index. Obviously, the support of the newly generated itemset intervals will always be smaller than $sup(P)$. There are two different situations: a) the support is already associated with some existing bin and we only need to add the newly generated interval to that bin; b) the support is a new value, which has not been indexed yet and we need to create a new bin to place the itemset interval. The pseudo-code of ALPINE+ is given in Algorithm 5 with the exploration subroutine is defined in Algorithm 6.

2.5 Computational Experiments

In this section, we extensively explore the ALPINE algorithm and empirically evaluate and analyze the ALPINE algorithm in comparison with related works in both frequent itemset mining and sequential top-k itemset mining. For frequent itemset generation, we choose three of the best existing frequent itemset generation algorithms in literature, *i.e.*, Eclat [71], FPGrowth [31] and LCM [66]. Of which, the LCM algorithm is closely related with our work and we downloaded its implementation - LCM (ver. 3) from the author's website[1]. Eclat and FPGrowth algorithms are implemented in SPMF open-source library [22]. The utility gained at each probe of all algorithms can be used to quantify the usefulness of the intermediate partial solutions. As the measure of utility is usually application-dependent, we don't define the concrete utility function form here, but directly list the *minsup*s reached at each probe. In top-k mining, we select the Seq-Miner [43] that mines the top-k frequent patterns sequentially without any minimum support. The proposed ALPINE algorithm

[1] http://research.nii.ac.jp/~uno/codes.htm

is implemented in JAVA and all the experiments were carried out on our department research cluster[2]. Both the experimental datasets from the FIMI repository[3] and a real gene expression dataset from the Cancer Cell Line Encyclopedia (CCLE) project[4] are used here. The computational overhead of ALPINE in comparison with LCM is also analyzed.

2.5.1 Support-wise Progress Analysis

Let us start with emphasizing the benefits of progressive data mining. Since all conventional frequent itemset generation algorithms [2, 48, 31, 71, 49, 66, 18, 17, 19] require setting up a-priori minimum support by the user, what if the minimum support is set too low and the transaction database is too large? If this happens, such algorithms may run for very long time (practically forever), "hanging" without providing any information to the user, except generating huge numbers of itemsets. However, no guarantees on the *minsup* reached at each point are given.

ALPINE, on the other hand, will provide the user with checkpoints which guarantee the partial completeness. It will provide lower and lower values of minimum support for which the set of frequent itemsets ALPINE produces is ***complete***. These guarantees will offer the user a measure of progress and knowledge about the subspace of the entire itemset search space that has been completely explored. A set of experiments is designed here to check how ALPINE systematically explores the itemset search space and to verify the added value of the ALPINE algorithm.

Experiment on Experimental Datasets

In the first set of experiments, we study how ALPINE and LCM proceed support-wise on experimental datasets from the FIMI repository. To illustrate the benefits of ALPINE, two relatively large transaction databases, i.e., T40I10D100K and Kosarak, which have many

[2] http://research.cs.rutgers.edu/~watrous/research-profile.html
[3] http://fimi.ua.ac.be/data/
[4] http://www.broadinstitute.org/ccle/data/browseData

Table 2.2: The $minsup$s reached at probes (in hour) by LCM and ALPINE on the T40I10D100K (left) and the Kosarak (right) dataset.

T40I10D100K		
Probe	LCM	ALPINE
1	7314	116
2	6390	56
3	5855	34
4	5317	22
5	4873	16
6	4499	13
7	4168	11
8	3882	10
9	3575	9
10	3313	8

Kosarak		
Probe	LCM	ALPINE
1	10178	982
2	9569	926
3	9264	907
4	8955	894
5	8810	885
6	8684	878
7	8645	872
8	8450	867
9	8379	862
10	8158	858

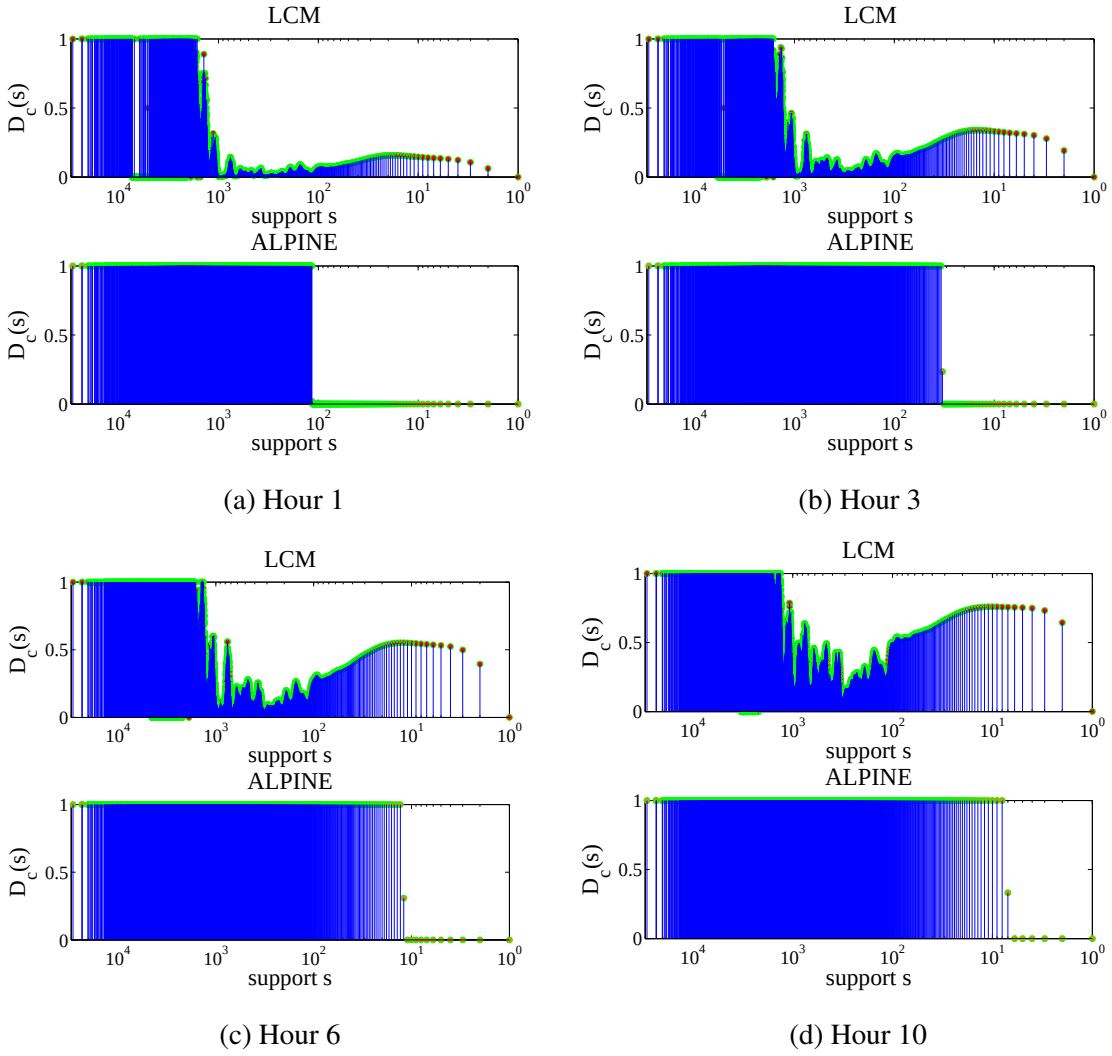
items and many transactions, are selected here. T40I10D100K has 100,000 transactions over 1,000 items generated by the IBM Quest Synthetic Data Generator, while Kosarak has 990,000 transactions over 41,270 items containing the click-stream data of a Hungarian on-line news portal. To reduce the number of mined itemsets, both ALPINE and LCM are confined to mine *closed* frequent itemsets in this experiment.

The experimental setting is as follows: we start both the ALPINE and the LCM algorithm at the same time, and probe every hour since they are started, i.e., Hour 1, Hour 2, ..., to check the status of both algorithms. Since ALPINE is parameter-free, it is not required to set any threshold. It just continuously mines itemsets from checkpoint to checkpoint and tries to build the full support index for a given transaction database $\mathcal{T}$. Different from ALPINE, LCM must be initialized with some user-provided minimum support threshold. In this experiment, we set the minimum support threshold of LCM to 1 to mine all the itemsets from $\mathcal{T}$ in consideration of building the full support index. The minimum support reached, that is, all the itemsets with support greater than or equal to the minimum support are discovered, at each probe t by the ALPINE algorithm is readily obtained from its last checkpoint before t, while this information for the LCM algorithm is obtained by post-processing all its output itemsets up to time t.

The results of both algorithms for the set of probes up to ten hours on T40I10D100K and Kosarak datasets are shown in Table 2.2. For both datasets, the first column is the probe time in hour, and the second and third column list the *minsup* reached at each probe by LCM and ALPINE, respectively. It's clear from the table that ALPINE can quickly reach some lower minimum support value than LCM. For instance, on the T40I10D100K dataset, ALPINE can reach the minimum support of 116 in the first hour while LCM only completes the subspace of all itemsets with support above 7314. The same trend is also observed in the Kosarak dataset. For the Kosarak dataset has more items, it's even harder for the LCM algorithm to move minimum support. We notice that even after twenty hours, the minimum support LCM reached on the Kosarak dataset is 7920, while ALPINE has already finished all itemsets with support greater than or equal to 835.

The underlying reason is ALPINE systematically explore the itemset space in a "monotonic" manner. ALPINE guarantees that all itemsets with support exceeding the current checkpoint's support have been found before it proceeds further, to build the support index for a lower minimum support. In contrast, LCM directly enumerate itemsets in a depth-first-search manner. To understand how these two algorithms behavior differently, we have taken the partial solutions generated for the T40I10D100K dataset at one hour, three hours, six hours and ten hours as slices to look into the algorithms. We analyze these intermediate results and calculate the degree of completeness for all possible support values s from this dataset, *i.e.*, $D_c(s)$.

The results are plotted in Figure 2.5. In each graph, the horizontal axis is the support value in a log scale, and the vertical axis is the normalized degree of completeness for each support s. For example, in Figure 2.5a, it plots the partial answer generated by LCM and ALPINE after one hour. For LCM, in this intermediate solution, we can find itemsets with almost all different possible support values s from this dataset, but the majority of these supports are incomplete (*i.e.*, $D_c(s) < 1$). Different from LCM, in ALPINE's partial output, all supports larger than the support of the current working bin (to the left of it) are



(a) Hour 1

(b) Hour 3

(c) Hour 6

(d) Hour 10

Figure 2.5: Progress of the LCM and the ALPINE algorithm at probes on the T40I10D100K dataset. The horizontal axis is the support in decreasing order in a log scale and the vertical axis is the degree of completeness of each distinct support value.

complete, while none of the itemsets from a lower support bin have been explored. Thus, the computational overhead of ALPINE at each checkpoint is minimum.

By checking all the graphs in Figure 2.5 together, we can intuitively perceive how both algorithms make progress as the computational time increases. The quality of the solution from ALPINE improves as the built support index is more and more complete. For LCM, though the completeness of a specified support value improves, in terms of the moving of minimum support, the progress is not so obvious. Imagine a dataset with even more items, the LCM algorithm might be stuck computing while ALPINE can report useful and

actionable knowledge in time through *checkpoints*. In the next subsection, we'll test both algorithms on a real gene expression dataset.

Experiment on Real Gene Expression Dataset

In the second set of experiments, we use LCM and ALPINE to mine all the co-regulated genes or gene groups from a real gene expression dataset from the Cancer Cell Line Encyclopedia project for the drug sensitivity analysis [8]. The gene-centric RMA-normalized mRNA expression data [30] consists of the expression values of 18,988 genes against 1,037 cell lines (patients).

To make the dataset applicable for both the LCM algorithm and the ALPINE algorithm, each column pertaining to the expression of a single gene is split into several binary columns. Since the data has been properly normalized, we simply adopt the equal-depth (frequency) partitioning [25] method to discretize each gene expression into five bins. The resulting transaction database has 94,940 items and 1,037 transactions, with a density of 20 percent. We name this dataset as the CCLE_Expression dataset. To compress the output from this high-dimensional dataset, only *closed* frequent patterns are mined in this experiment.

We ran both LCM and ALPINE on the gene expression dataset. The minimum support threshold of LCM is set to 80, due to the huge number of resulting closed frequent patterns from this dataset. Similar to the experimental setting for the experimental datasets used in the previous subsection, a series of random probes are selected in time and the minimum support reached by both algorithms are checked at every probe. The results are presented in Table 2.3. We find that ALPINE can continuously make progress in terms of lowering the reached minimum support. However, the LCM algorithm is stuck in this case at the minimum support of 208. Since ALPINE always focuses on building the uncompleted bin with the highest support from the index, while LCM spreads its power to the whole support spectrum, making all bins to be completed almost at the same time.

Table 2.3: The *minsup*s reached at probes (in hour) by the LCM and the ALPINE algorithm on the CCLE_Expression dataset.

Probe	2	3	4	5	6	7	8	9	10	13	15	18	23
LCM	209	208	208	208	208	208	208	208	208	208	208	208	208
ALPINE	136	125	122	120	119	118	117	116	115	114	113	112	111

In practical scenarios like this one, it is generally reasonable to have some time period for most data analysis operations. Though ALPINE might also not be able to finish the whole mining task within the time period, but it is always able to offer a partial solution based on its last checkpoint. Furthermore, the partial solution offered by ALPINE is *complete* in itself and has the definite guarantee with regard to a certain minimum support. Thus, these complete sets of co-regulated genes or gene groups with a higher minimum support returned early by ALPINE can be used to predict the drug response even though the mining process continues. The high support implies high coverage, which might lead to more widely applicable associations in this case. Besides, as the computational time increases, the built support index is more and more complete and ALPINE continues to offer those lower support patterns.

2.5.2 Comparison with The Benchmark Approach

Tušar *et al.* propose an approach for benchmarking budget-dependent algorithms that allows anytime performance assessment of their results [62] based on the Comparing Continuous Optimizers (COCO) platform [34]. The idea is very simple: the algorithm is run with increasing budgets and the resulting runtimes are presented in a single data profile (an empirical cumulative distribution function) [45].

Consider K increasing budgets $b_1, b_2, ..., b_K$ and K budget-dependent algorithm variants $A_{b_1}, A_{b_2}, ..., A_{b_K}$. The algorithm $\tilde{A}$ first works as algorithm A_{b_1} for budgets $b \leq b_1$, then works as algorithm A_{b_2} for budgets b, where $b_1 < b \leq b_2$, and so on, finishing by mimicking algorithm A_{b_K} for budgets b, where $b_{K-1} < b \leq b_K$. Back to the frequent itemset mining problem, the budget-dependent algorithm variants could be the frequent itemset mining

Table 2.4: Properties of datasets

Dataset	Number of Items	Number of Transactions
BMS-WebView-1	497	59,602
BMS-WebView-2	3,340	77,512
Retail	16,470	88,162
T10I4D100K	870	100,000
Chess	75	3,196
Mushroom	119	8,124

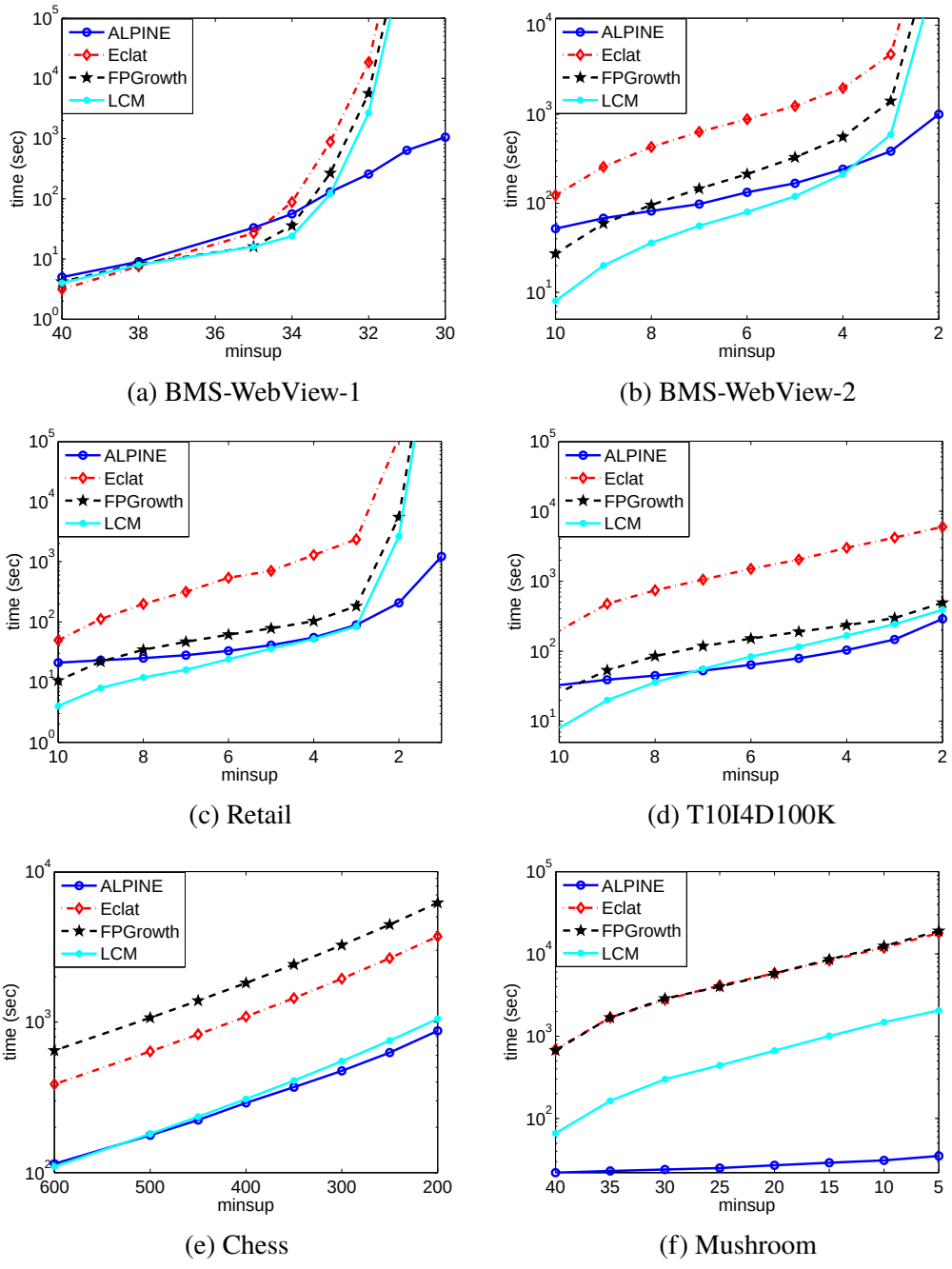
algorithm with different minimum support thresholds. Thus, the benchmarking process is similar to the parameter tuning process in interactive pattern mining [67].

Rather than setting the minimum support a priori, a user may run a mining algorithm sequentially with the successively smaller values of the minimum support threshold. This will of course result in repeated work. Every time a new call to the mining algorithm is made with smaller value of minimum support, the most frequent itemsets are generated again and again. The advantage however is similar to the one provided by ALPINE, checkpoints can now be provided just like they are produced by ALPINE. However, such recursive process, not surprisingly, incurs substantial time penalty as compared with ALPINE. In this subsection, we compare ALPINE with the benchmark approach of recursive evaluation of frequent itemset mining algorithms, Eclat [71], FPGrowth [31] and LCM [66], with decreasing minimum support thresholds.

In the set of experiments, we select BMS-WebView-1, BMS-WebView-2, Retail, T10I4D100K, Chess and Mushroom datasets from the FIMI repository. The transaction instances can be classified into two groups. The first group are *sparse* datasets, composed of BMS-WebView-1/ BMS-WebView-2[5], Retail and T10I4D100K datasets, while the second group are *dense* datasets, *i.e.*, Chess and Mushroom. The properties of these datasets are listed in Table 2.4.

A set of *minsup*s from the checkpoints of ALPINE are applied to Eclat, FPGrowth and LCM recursive for multiple runs. The resulting accumulated runtimes are plotted in a single

[5]click-stream data from a webstore used in KDD-Cup 2000



(a) BMS-WebView-1

(b) BMS-WebView-2

(c) Retail

(d) T10I4D100K

(e) Chess

(f) Mushroom

Figure 2.6: Comparison with the benchmark approach

data profile for each algorithm respectively in Figure 2.6. From the figure, we can verify that the rate of runtime increase of ALPINE is much slower than the rest algorithms for almost all the datasets, eventually ALPINE becomes faster than the three algorithms, although it might take a bit longer than them at the beginning, for ALPINE takes time to initialize the support

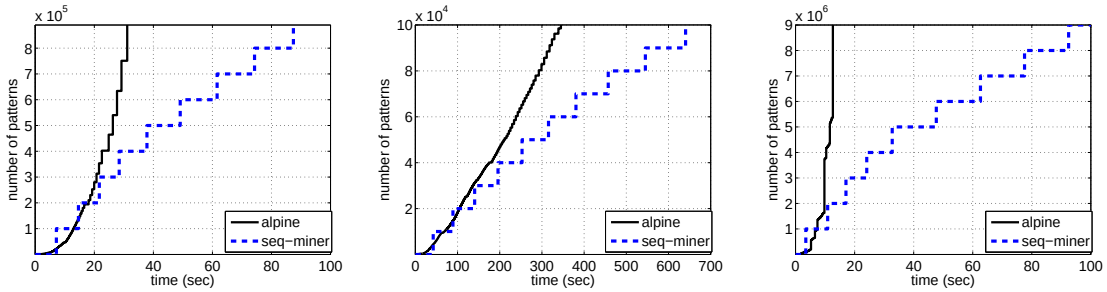
index. The reason is that APLINE mines contiguously from checkpoint to checkpoint in a monotonic manner without any repeated work (never starting from scratch), while the benchmarking process restarts a variant of the algorithm every time. It is not difficult to verify in Figure 2.6, the advantage of ALPINE increases as the number of iterations of frequent pattern mining algorithm increases in this benchmarking process.

2.5.3 Comparison with Sequential Top-k Mining

A sequential top-k miner shares some flavor of the contract-type anytime algorithm [75] and it can also provide definite results at each chunk, i.e, all itemsets with support above or equal to the support of the last one in the chunk. The advantage is similar to the one provided by ALPINE, checkpoints can now be provided just like ALPINE. As shown in [35], the sequential top-k mining algorithm is more efficient than the benchmark approach, running the traditional data mining algorithms recursively with successively smaller values of *minsup*. That's why we also compared our algorithm with a sequential top-k miner - Seq-Miner [43]. Seq-Miner mines the top-k frequent patterns sequentially and outputs every top n_c (a user defined chunk size) patterns.

From the set of datasets used in Section 2.5.2, we selected two sparse datasets and one dense dataset as representatives for this experiment, namely, BMS-WebView-2, T40I10D100K and Mushroom. For Seq-Miner, the chunk size n_c of these datasets are selected to be 10^5, 10^4 and 10^6, respectively, according to the density and output number of frequent patterns of each dataset. The number of patterns generated at each checkpoint and the time to reach that checkpoint is plotted in Figure 2.7 for both algorithms. In this figure, the horizontal axis is the running time since the algorithm starts and the vertical axis is the number of generated patterns.

From these graphs, it's easy to verify the facts: 1) ALPINE produces many more checkpoints than Seq-Miner given the same execution time; 2) the step size (time between two successive checkpoints) of Seq-Miner is much longer than that of ALPINE, and it



(a) BMS-WebView-2 (b) T40I10D100K (c) Mushroom

Figure 2.7: Comparison with sequential top-k mining algorithm

grows as the running time increases. The reason is a new and larger FP-tree has to be rebuilt from scratch whenever a given top-k is changed in the Seq-Miner. Every time a new call to the mining algorithm is made with the smaller value of *minsup* discovered in the VirtualGrowth. Thus, the FP-tree is built many times and the most frequent itemsets are generated again and again.

Different from Seq-Miner, ALPINE monotonically explores itemset intervals with descending values of support and mines continuously from checkpoint to checkpoint without any redundancy (it never starts from scratch). Not surprisingly, the iterative process of Seq-Miner incurs substantial time penalty as compared to that of ALPINE. The step size of Seq-Miner is related with the parameter - chunk size n_c, and we can reduce the step size by reducing its chunk size. In that case, it will result in more iterations in this iterative process and more repeated work in total.

In general, the number of iterations and the total runtime of Seq-Miner might grow dramatically as we generate more and more frequent patterns. This is consistent with the trend displayed in the graphs of Figure 2.7 that the runtime difference of Seq-Miner and ALPINE grows with the increasing of the number of generated patterns. Thus, the superiority of ALPINE increases with the number of iterations of frequent pattern mining in Seq-Miner. Given the same time, ALPINE can always generate more frequent patterns than Seq-Miner. In other words, using ALPINE, users can obtain the complete set of itemsets above a lower *minsup* in the equivalent execution time in comparison with using Seq-Miner.

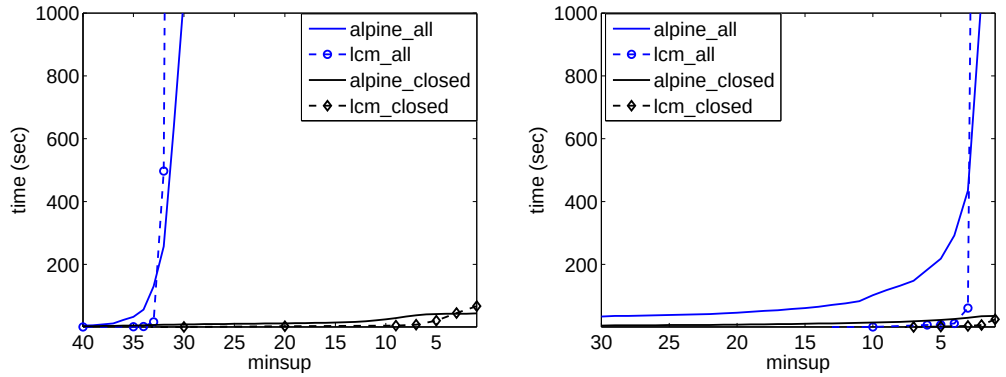
ALPINE turns out to be even more efficient than the contract-type like algorithm, though it is interruptible at any time.

2.5.4 Computational Overhead of ALPINE

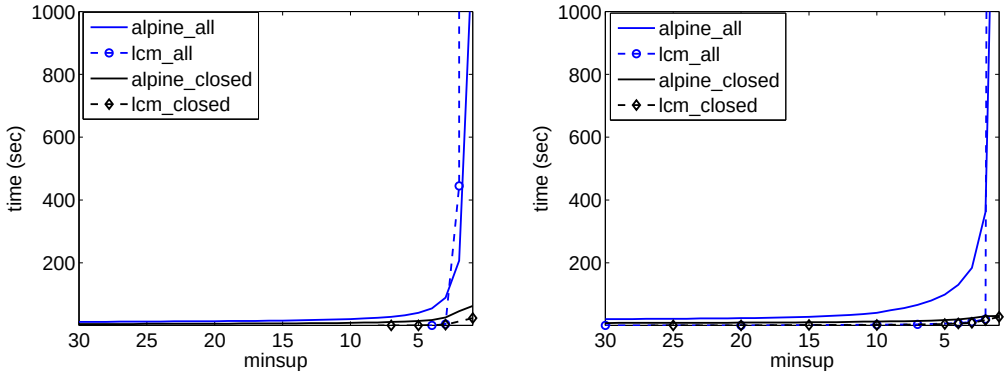
Many frequent pattern discovery algorithms have been developed in literature [2, 48, 31, 71, 49, 66, 18, 17, 19] and it is not our intention to develop yet another efficient algorithm. Instead, our aim here is to show the usefulness of a miner with partial completeness guarantees. To complete the picture, we also conducted experiments to evaluate the performance of ALPINE in mining all frequent or closed itemsets in comparison with LCM (for it performed better than Eclat and FPGrowth in Section 2.5.2).

In this set of experiments, we use the same set of datasets as in Section 2.5.2. ALPINE started without any parameter, while LCM was initialized with some *minsup* from ALPINE's checkpoints. The results are displayed in Figure 2.8. The horizontal axis is the absolute minimum support value, and the vertical axis is the runtime. Note that for every transaction database, ALPINE executes once to mine all frequent or closed itemsets, while LCM runs multiple times for the set of different initial *minsup* values. The curves for alpine_all and alpine_closed in the plots are continuous in the sense that the runtime is known for each distinct *minsup* value, indicated by solid lines. In contrast, the lcm_all and lcm_closed are plotted in dashed lines for only the results at markers are evaluated.

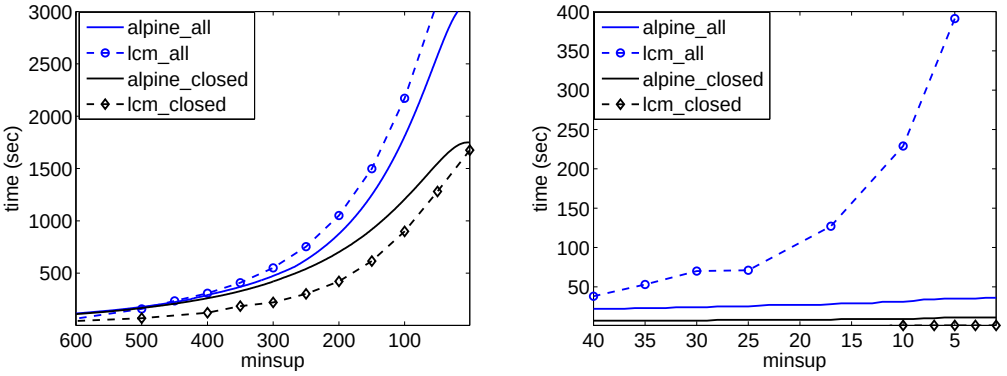
Overall, for all instances and *minsup* values, ALPINE is comparable to LCM, which can be verified from the graphs. The results validate the effectiveness of the itemset closure operator and the compact itemset interval representation. For closed itemset mining, ALPINE is slightly slower than LCM, however, the trend and the order of magnitude of the runtime of both algorithms are similar. For all frequent itemset mining, ALPINE catch up with or even compete with LCM as they get to lower and lower *minsup*. The reason is ALPINE processes and outputs groups of itemsets compactly in itemset intervals instead of enumerating each individual itemset in an interval.



(a) BMS-WebView-1 (b) BMS-WebView-2



(c) Retail (d) T10I4D100K



(e) Chess (f) Mushroom

Figure 2.8: Computational overhead of ALPINE

For sparse datasets like Figure 2.8a - Figure 2.8d, the graphs show similar trends: the curves of LCM and ALPINE are close to each other and ALPINE is slightly slower than LCM at the beginning. The difference between them might further increase in the middle of the curves as they are mining all frequent itemsets, for the overhead in generating and

maintaining a large number of itemset intervals below the current *minsup* might dominate the acceleration of the itemset closure operator and itemset interval compression. However, as the *minsup* gets lower and lower, the previously built partial index for lower minimum supports saves the cost for later stages and the low support itemsets might be more compactly grouped in itemset intervals. That's why we can see ALPINE might compete with LCM at some lower minimum support value for all frequent itemset mining on these datasets.

For dense datasets in Figure 2.8e and Figure 2.8f, the compression ratio of itemset intervals is even higher, so ALPINE becomes faster than LCM for mining all frequent itemsets. This advantage will become more obvious as we get to lower and lower minimum support, as indicated in Figure 2.8f.

2.6 Summary

In this chapter, we defined the progressive itemset mining framework for long mining tasks like slicing large multidimensional dataset in boundless data analytics. A dynamic approach - the ALPINE algorithm is proposed. ALPINE allows us to achieve flexible trade-offs between efficiency and completeness. ALPINE proceeds support-wise from checkpoint to checkpoint and can be interrupted at any time but offer intermediate meaningful and complete results with definite guarantees. ALPINE is evaluated to be superior to the benchmark approach and more efficient than the contract-type like algorithm, Seq-Miner, though it is interruptible at any time. It has minimal computational overhead as compared to the best existing frequent itemset mining algorithms. The progressive mining framework might be generalized to other anti-monotone itemset interestingness measures as well.

Chapter 3

Boundless Data Analytics

Multidimensional distributions in data mining are often represented as plots: scatter plots between two numerical attributes; heat maps, bar graphs, histograms, box plots - they either relate two attributes together or show frequency distributions of one attribute. What makes one plot more interesting than the other? What if we could generate all possible relationships and rank the most interesting ones at the top - do it all automatically, thus saving days of tedious and repetitive human work? The value of this work lies in dramatic reduction of human work needed to analyze data. We would like to harness computational resources to work for us, doing mundane work (at least for a creative data scientist) and isolate only plots which have huge potential interest.

3.1 Introduction

Multidimensional distributions are often used in data mining to describe and summarize different features of large datasets [3]. Multidimensional analysis [60] is an informational analysis on data which takes into account many different relationships, each of which represents a dimension. It is a data analysis process which groups data into two categories: data dimensions and data measurements[1]. For example, a wine retail analyst may want to

[1] https://en.wikipedia.org/wiki/Multidimensional_analysis

understand how the relationship between *price* and *alcohol* level varies by *region*, by wine *type*, by *manufacture*, by *year* of production, *i.e.*, the French chardonnays from 1990s may have quite different joint distribution of price and alcohol from Italian cabernets from the 1980s. Multi-dimensional analysis will yield results for these complex relationships. The challenge is how to handle large data sets in a large number of dimensions [28]. It may calculate an unbounded number of plots and subgroups in two directions:

I. We can keep supplementing the multidimensional database with any arbitrary aggregated dimensions and measurements, *i.e.*, sum, avg (average), max, min, $quartile$, $percentile$, $count$, $rank(n)$ [68], $variance$ [53], $mode$, or other relational aggregate functions of an attribute aggregated over the population of a specific group.

II. We can keep exploring the multidimensional data and selecting subsets of it in all possible ways: 1) by classical navigational operators like $roll{-}up$, $drill{-}down$, $slice$, $dice$ and $pivot$ based on the data dimensions [50] and new derived dimensions; 2) by slicing based on an interval-discretized numeric attribute [70]; 3) by dividing the 2D geometric plane of two numeric attributes into sectors and slicing the data sector-wise etc.

For instance, a typical multidimensional database has five to seven dimensions, an average of three levels hierarchy on each dimension and aggregates more than a million rows [53]. The above form of data analysis and manual exploration process can get tedious and error-prone for large datasets that commonly appear in real-life. What if we could give the work to the computer and automate this data analysis process?

Given a large multidimensional database along with its schema, the *boundless data analytics* is a mining process with the objective to generate all possible univariate and bivariate relationships or plots with sufficient support. Because the number of dimensions may also include derived dimensions so we do not know ahead of time how long the process will take, may even take an unbounded amount of time. This process is briefly mentioned in Chapter 2 and we'll elaborate it in detail in this chapter.

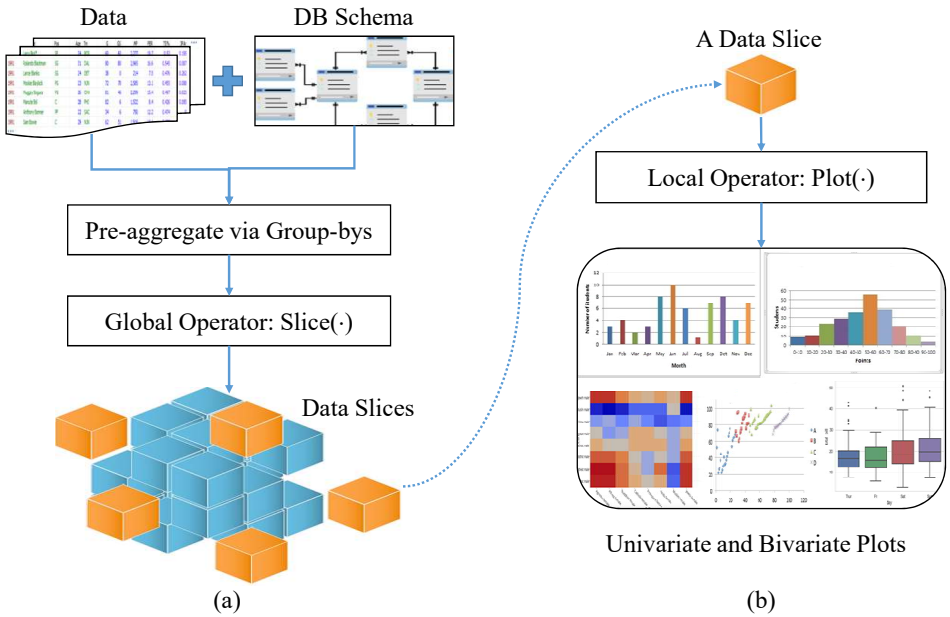


Figure 3.1: The framework of boundless analytics: (a) global slicing process, (b) local plotting process.

The proposed framework of automated boundless analytics is sketched in Figure 3.1. Initially, aggregates are pre-calculated by performing SQL Group by queries on the relationship tables to supplement involving entities with new summarized statistics or measurements. Then the augmented multidimensional database goes through (a) a global slicing process and (b) a local plotting process. The global operator, $Slice(\cdot)$, operates on the supplemented multidimensional database and tries to partition the data into different subgroups or data slices based on its dimensions. We utilize the progressive itemset mining paradigm introduced in Chapter 2 to slice the data progressively in descending support. The local operator, $Plot(\cdot)$, will do the univariate and bivariate analysis for each large data slice output from the global slice operator and generate all the 1-D and 2-D plots in a slice.

The rest of this chapter is organized as follows. Section 3.2 introduces the basic concept and notations used throughout this chapter. Section 3.3 discusses how we score the plots based on their spread. Section 3.4 illustrates how to pre-aggregate the relationship tables by the set of foreign keys. Then Section 3.5 and Section 3.6 explain the global slicing operator and the local plotting operator, respectively. Finally, we conclude this chapter in Section 4.7.

3.2 Basic Notations

3.2.1 Entity-Relationship Model

A multidimensional database is usually organized as a set of decentralized tables forming the star-schema model [7, 39, 47], Franconi *et al.* [24] generalize it in an extended version of the entity-relationship conceptual data model [61], which is able to handle more complex descriptions of aggregated entities. We will also use the E-R model to represent a dataset in this chapter, where *dimension* is a synonym for a domain of an attribute that is structured by a hierarchy or an order.

Conceptually, let $\boldsymbol{S} = \{E, R\}$ represents a database schema, where $\boldsymbol{E} = \{e_i\}$, $i = 1, 2, ..., m$, be the set of entities and $\boldsymbol{R} = \{r_j\}$, $j = 1, 2, ..., n$, be the set of relationships connecting those entities. Each entity e_i contains the information specific to itself, while the relationship r_j connects some entities from $\boldsymbol{E}$ and contains attributes for the relationship.

Let $R[e]$ be a relational table scheme corresponding to the entity e, which has one primary key and no foreign keys; the table may have other fields that are categorical, numerical or boolean. $R[r]$ be a relational table scheme corresponding to the relationship r, which contains *foreign keys* to other entities; in general, a relationship table also contains some other attributes called measurements about the relationship. Given an entity e in $\boldsymbol{E}$, let $FK[e]$ be the set of relationships in R where e is a foreign key.

We'll use the well known bar-beer-drinker database schema [63] as a running example in this chapter. The E-R diagram for the data is displayed in Figure 3.2, in which we can find three entities, *Beers*, *Bars* and *Drinkers* and two relationships, *Sells* and *Frequents*.

Beers(name, manf);

Bars(name, addr, license);

Drinkers(name, addr, phone);

Sells(bar, beer, price);

Frequents(drinker, bar, spending).

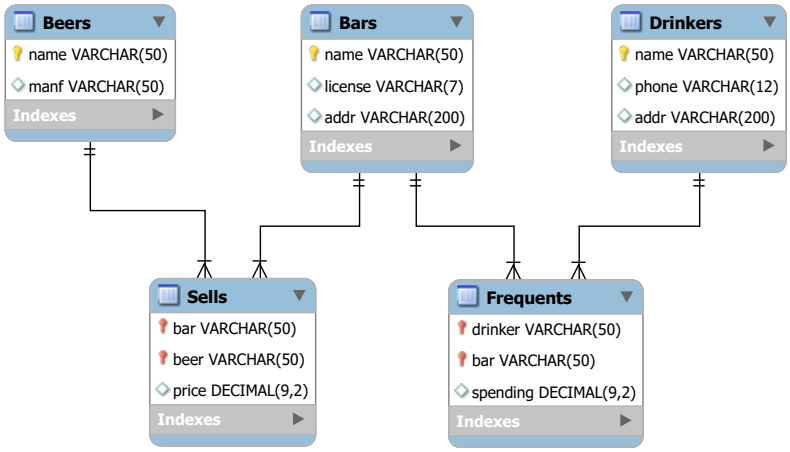


Figure 3.2: Entity-Relationship representation of the bar-beer-drinker database

The three entity tables, *Beers*, *Bars* and *Drinkers*, have a unique name field to record the information of beers, bars and drinkers, respectively. Each entity table has a number of attributes about the entity. To ensure uniqueness, we refer to an attribute by concatenating it with the table name, for instance, *Beers*.name and *Bars*.name.

The relationship table, *Sells*, stores information on all menus in bars. In this table, except the attribute bar references the name in *Bars* and beer references the name in *Beers*, the price information of a beer sold in a bar is also recorded, which is a measurement about this relationship. Similarly, in the relationship table, *Frequents*, drinker references the name in *Drinkers* and bar references the name in *Bars*, and spending tracks the amount a drinker spends in a bar monthly. An instance of the database schema is shown in Table 3.1.

3.2.2 Plots Parameterized by Data Slice

Let $\boldsymbol{I} = \{i_1, i_2, ..., i_m\}$ be a finite set of distinct items. In the case of a relational database, an item corresponds to a *descriptor* of the form (*attribute*, *value*) [56], specifically, *attribute* = *value* if the attribute is a discrete attribute, *i.e.*, categorical or ordinal, or *attribute* $\in$ *interval* if the attribute is a continuous attribute. For instance, for the example given in Table 3.1, an item can be represented as *Beers*.manf = 'Anheuser-Busch' or *Bars*.addr = 'New York'. A set of items is denoted as an *itemset*, which corresponds to a transaction in a database.

Table 3.1: An example instance of bar-beer-drinker schema: entity tables (a) *Beers*, (b) *Bars* and (c) *Drinkers*, and the relationship tables (d) *Sells* and (e) *Frequents*.

(a) Entity: *Beers*

name	manf
Budweiser	Anheuser-Busch
Michelob Ultra	Anheuser-Busch
Blue Moon	Coors Brewing Company
Zima	Coors Brewing Company

(b) Entity: *Bars*

name	addr	license
Cabana	San Francisco	CA4567
Seven Bamboo	New York	NY1234
Hedley Club	New York	NY1235
Blue Angel	San Francisco	CA4578

(c) Entity: *Drinkers*

name	addr	phone
Bob	San Francisco	415-234-6789
Vince	New York	234-456-7890
Jesse	San Francisco	415-234-7642
Rebecca	New York	234-456-4114

(d) Relation: *Sells*

bar	beer	price
Cabana	Budweiser	5
Cabana	Blue Moon	7.5
Cabana	Zima	4.25
Seven Bamboo	Budweiser	5.5
Seven Bamboo	Michelob Ultra	6
Seven Bamboo	Blue Moon	7
Hedley Club	Budweiser	5.75
Hedley Club	Michelob Ultra	6
Blue Angel	Blue Moon	6.75
Blue Angel	Budweiser	5.25

(e) Relation: *Frequents*

drinker	bar	spending
Bob	Cabana	35
Bob	Blue Angel	65
Bob	Seven Bamboo	25
Vince	Seven Bamboo	50
Vince	Hedley Club	60
Jesse	Cabana	30
Jesse	Blue Angel	75
Rebecca	Cabana	30
Rebecca	Seven Bamboo	45
Rebecca	Hedley Club	25

A conjunction of k descriptors is denoted as a *k-conjunct*. For a given *k-conjunct* and a database of objects [38]:

- The set of objects that satisfy the *k-conjunct* define the *slice* for that conjunct. Logically, a slice depicts a multidimensional view of the data.
- The attributes that constitute the *k-conjunct* define the *dimensions* of the slice.
- Distributions of attributes (1-dimensional or 2-dimensional) over objects which satisfy the slice definition define the *plot*s of the slice.

A slice S' is defined to be a specialization or a subslice of another slice S if the set of records in S' is a subset of the set of records in S. As an example, the slice of the sales of beers from Anheuser-Busch in bars in New York from Table 3.1 is denoted by (*Beers*.manf = 'Anheuser-Busch' $\wedge$ *Bars*.addr = 'New York'). The dimensions for this slice are *Beers*.manf and *Bars*.addr. If we look at the relationship between *Sells*.price and *Sells*.beer in this slice, then the side-by-side box plots will be generated for 'Budweiser' and 'Michelob Ultra', respectively. A possible subslice of the slice would be (*Beers*.manf = 'Anheuser-Busch' $\wedge$ *Bars*.addr = 'New York' $\wedge$ *Sells*.price $\in [5, 6]$). Note that if an *m-conjunct* T is a superset of an *n-conjunct* T' ($m \geq n$), then the slice defined with T is a subslice for the slice described by T'. Usually, given a dataset, we treat the categorical attributes as its dimensions (or the independent attributes).

3.2.3 Univariate and Bivariate Plots

Both discrete and continuous attributes are of our concern here. For a data slice with sufficient support, we will do univariate and bivariate analysis. Univariate analysis is the simplest form of data analysis where the data being analyzed contains only one variable, while bivariate analysis is used to find out if there is a relationship between two different variables. Technically, we could extend to multivariate analysis, it just becomes harder to visualize. Thus, we confine to generate all the possible 1-D and 2-D plots for a data slice.

Distribution of a single attribute

Single discrete attribute (like *Beers*.manf): The distribution of a variable shows its pattern of variation, as given by the values of the variable and their frequencies. To get an idea of the pattern of variation of a discrete variable, we can display the information with a *bar graph* as indicated by Figure 3.3a.

Single continuous attribute (like *Sells*.price): For this kind of attributes, we should use a *histogram*. Histograms differ from bar graphs in that they represent frequencies by area and

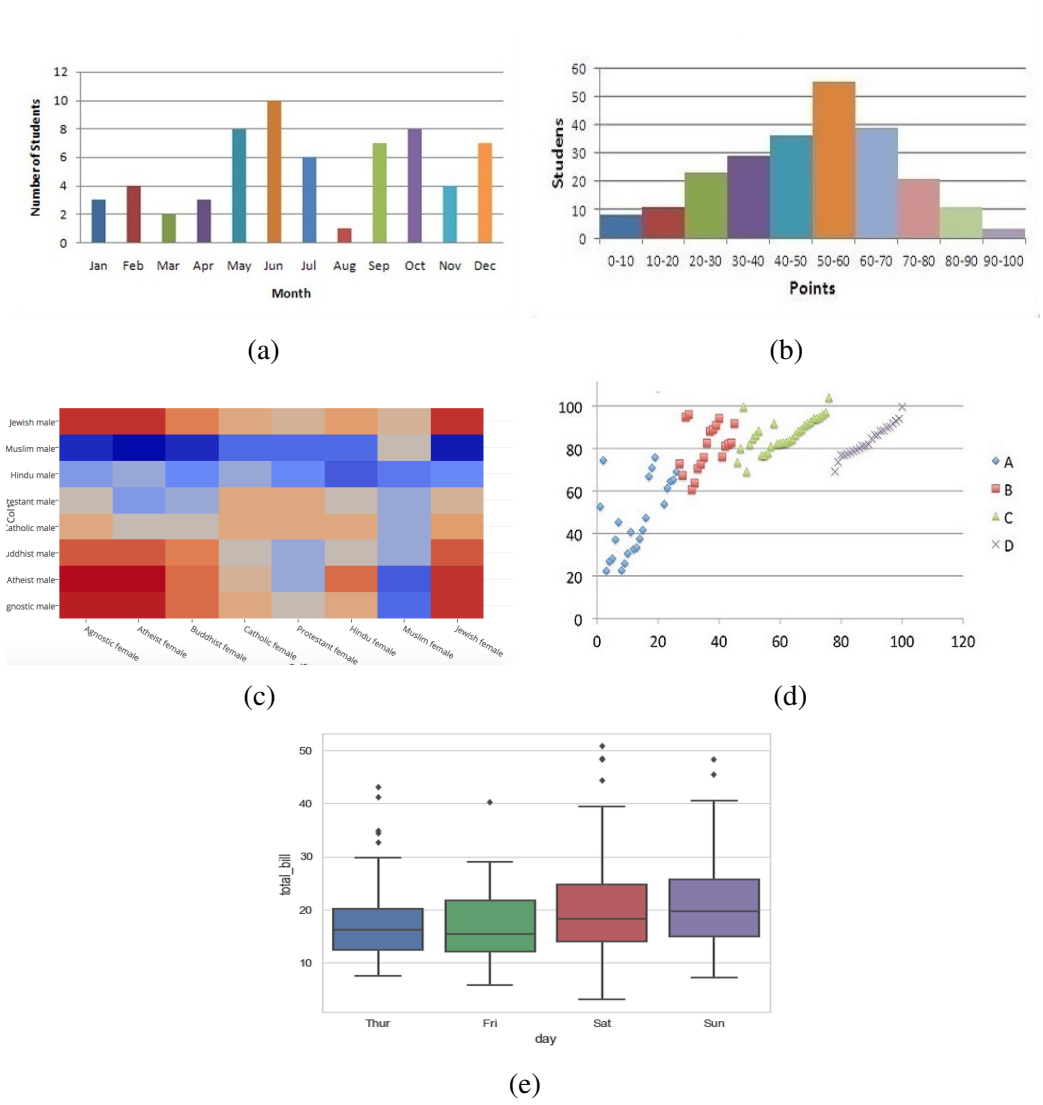


Figure 3.3: Examples of univariate and bivariate plots: (a) bar graph, (b) histogram, (c) heatmap, (d) scatter plot and (e) side-by-side box plots

not by height. A good display will help to summarize a distribution by reporting the center, spread, and shape for that variable. One example is given in Figure 3.3b.

Relationship of two attributes

Two discrete attributes (like *Sells*.bar and *Beers*.manf): We analyze an association through a comparison of conditional probabilities and represent the data using contingency tables. Graphically, a contingency table is shown as a *heatmap* like Figure 3.3c.

Two continuous attributes (like *Frequents*.spending and aggregated attribute *Bars*.avg_price): To analyze this situation we consider how one attribute, called a response attribute, changes in relation to changes in the other attribute, called an explanatory attribute. Graphically, we use a *scatter plot* like Figure 3.3d to display the distribution over two continuous attributes.

One discrete attribute and one continuous variable (like *Sells*.price and *Beers*.manf): These are best compared by using *side-by-side box plots* to display any differences or similarities in the center and variability of the continuous attribute across the discrete attribute. An example is shown in Figure 3.3e.

3.3 Score the Plots

As mentioned in the opening of this chapter, we intend to find those "interesting" plots. How to measure the interestingness of a plot? In this section, we propose to quantify/score the interestingness of a plot based on its spread. The proposed score function favors the inequality or unbalance of a plot. The intuition is that the more unbalanced the distribution is, the more strategies can be taken to promote the response for segmented groups. Otherwise, if the data is evenly distributed across the variable, no group segmentation is necessary, for it is an irrelevant factor for the response. This is just one way to score the plots, not a universal approach, other user-in-the-loop and application-aware measures are yet to develop.

The Gini coefficient [13] (sometimes expressed as a Gini ratio or a normalized Gini index) is a general measure of statistical dispersion and the most commonly used measure of inequality. Thus, we employ it here to score the plots. Another reason the Gini coefficient is chosen over the standard deviation is that it is invariant to scale and is bounded within [0, 1]. A Gini coefficient of 0 expresses perfect equality, where all values are the same. A Gini coefficient of 1 expresses maximal inequality among values.

Mathematically, the Gini coefficient is defined as a ratio of the areas on the Lorenz curve diagram (Figure 3.4). If the area between the line of perfect equality and Lorenz curve is

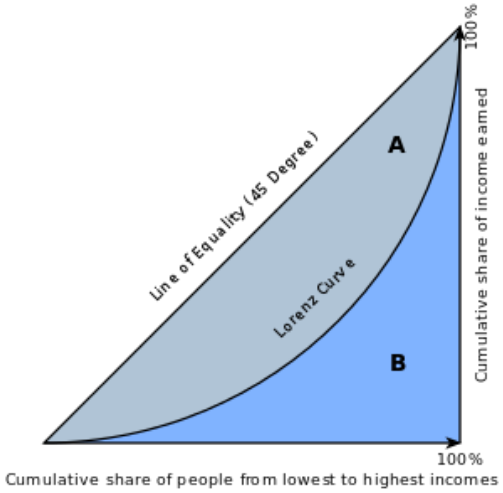


Figure 3.4: Graphical representation of the Gini coefficient[2]

A, and the area under the Lorenz curve is B, then the Gini coefficient is $A/(A + B)$. Since $A + B = 0.5$, the Gini coefficient, $G = 2A = 1 - 2B$. If the Lorenz curve is represented by the function $Y = L(F)$, the value of B can be found with integration and:

$$G = 1 - 2\int_0^1 L(F)dF \tag{3.1}$$

In some cases, this equation can be applied to calculate the Gini coefficient without direct reference to the Lorenz curve. For example: for a population with values y_i, $i = 1$ to n, that are indexed in non-decreasing order ($y_i \leq y_{i+1}$):

$$G = \frac{1}{n}(n + 1 - 2\frac{\sum_{i=1}^{n}(n + 1 - i)y_i}{\sum_{i=1}^{n} y_i}) \tag{3.2}$$

This formula actually applies to any real population, since each sample can be assigned its own y_i. The Gini coefficient's main advantage is that it is a measure of inequality by means of a ratio analysis, rather than a variable unrepresentative of most of the population.

For plots of distributions of single attributes (like bar graphs or histograms), the Gini coefficient can be calculated directly among values of the frequency distribution. For plots of relationships of two attributes, a heat map can be flattened and vectorized as a 1-D bar

[2] https://en.wikipedia.org/wiki/Gini_coefficient

graph and the Gini coefficient is calculated thereafter; for side-by-side box plots, the Gini coefficient can be calculated for the mutability of the five-number summary of each box.

As to the scatter plot, which uses Cartesian coordinates to display values of two variables for a set of data. Strength refers to the degree of "scatter" in the plot. If the dots are widely spread, the relationship between variables is weak. If the dots are concentrated around a line, the relationship is strong. We'll score this kind of plots by measuring the strength of a linear relationship between the two variables. In statistics, the Pearson correlation coefficient [9] is such a measure:

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X \sigma_Y} \tag{3.3}$$

where $cov(X,Y)$ is the covariance between X and Y, σ_X and σ_Y are the standard deviation of X and Y, respectively. μ_X and μ_Y are their means. E is the expectation operation. Standard deviation is a measure of the dispersion of data from its average. Covariance is a measure of how two variables change together, but its magnitude is unbounded so it is difficult to interpret. By dividing covariance by the product of the two standard deviations, a normalized version of the statistic is calculated. Thus, it has a value between +1 and -1, where +1 is total positive linear correlation, 0 is no linear correlation, and -1 is total negative linear correlation. To make the range be [0, 1] and it an indication of the degree of correlation/strength between two attributes, no matter they are positively or negatively correlated, we use $|\rho_{X,Y}|$ as the score function for a scatter plot.

After introducing the basic notations and concepts, now we are ready to discuss each step, *i.e.*, pre-aggregation, slicing and plotting, in boundless data analysis in Figure 3.1.

3.4 Pre-Aggregate the Relationship Tables

Let's first look at several examples. The wine analyst mentioned in Section 3.1 might want to study the general trend of the price of a wine with respect to its year of production, for all

wines, not for a specific type or brand of wine. Or an analyst would like to study the beers' selling price for each manufacturer in Table 3.1. These kinds of analysis requires to change the granularity of the data and can not be conducted directly from the original data, which motivates us to pre-calculate some summary statistics for our boundless analytics.

Kimball *et al.* [40] mentioned that the single most dramatic way to affect performance in a large multidimensional database is to provide a proper set of aggregate (summary) records that coexist with the primary base records. At the simplest form an aggregate is a simple summary table that can be derived by performing a Group by SQL query [26]. Using the same data model and notations introduced in Section 3.2, now we will describe the process of expanding the entity tables $R[e]$ into $R^*[e]$ with additional attribute aggregates resulting from participation of e as foreign key in relationships in $FK[e]$. Each relationship r in $FK[e]$ will provide additional attributes to the supplemented table $R^*[e]$ as follows: for each attribute in r, we will group them by the key of e based on Formula 3.4:

$$agg_e(a_j) = \begin{cases} avg_e(a_j), & \text{if } a_j \text{ is continuous} \\ count_e(distinct\ a_j), & \text{if } a_j \text{ is discrete} \end{cases} \tag{3.4}$$

where $\{a_j\}$ is the set of attributes in r. This process will result in m new aggregated measures for entity e, where m is the number of attributes in r. If e participate in n relationships as foreign key, *i.e.*, $|FK[e]| = n$, then the additional attribute aggregates resulting from participation of e as foreign key in relationships in $FK[e]$ should be in the order of $O(m * n)$.

We'll explain the process in detail by an example. For instance, the entity *Bars* has participated as a foreign key in two relationships: *Sells* and *Frequents*, in the entity-relationship model given in Figure 3.2. In notation, we have $FK[Bars] = \{Sells, Frequents\}$. The relational table schema for these two relationships are: ***Sells***(bar, beer, price) and ***Frequents***(drinker, bar, spending).

Firstly, let's aggregate the relationship table *Sells* based on bar. For the attribute beer, which is a discrete attribute, according to Formula 3.4, we'll count the distinct values of

beer for each bar, whose physical meaning is the number of beers sold at a bar (denoted as num_of_beers_sold). When it comes to the continuous attribute price, we'll calculate its average for each bar, meaning the average price of beers sold at each bar (avg_beer_price). Both num_of_beers_sold and avg_beer_price describe some properties of a bar. Thus, we can supplement the entity table *Bars* with new, unanticipated, aggregated attributes from the aggregates [40], yielding larger entity tables with more columns.

Similarly, we will aggregate the relationship table *Frequents* based on bar, getting $count(distinct$ drinker) and avg(spending) for each bar, which mean the number of drinkers frequenting a bar (num_of_drinkers_frequenting) and the average amount spent by drinkers frequenting the bar (avg_spending). As a result, we can get the supplemented entity table:

Bars*(name, addr, license, num_of_beers_sold, num_of_drinkers_frequenting, avg_beer_price, avg_spending)

When the entity tables are supplemented with pre-calculated aggregates, we can calculate a lot of new interesting plots which are not possible from the original database. For instance, we can plot the relationship between num_of_drinkers_frequenting and avg_beer_price. Is it the cheaper the average beer price sold in a bar, the more drinkers will frequent the bar or vice versa? Or we can look at how the location of a bar influences the average spending of drinkers visiting the bar and so on.

A more common use of aggregates is to take an entity and change the granularity of this entity. For instance, finding the sells by manufacturer for Table 3.1, which changes the granularity in entity *Beers*. When changing the granularity of the entity, the relationship table has to be partially summarized to fit the new grain of the new entity, thus creating new entity and relationship tables, fitting this new level of grain[3]. Having aggregates and atomic data increases the complexity of the data model. The number of possible aggregates is determined by every possible combination of dimension granularities.

[3] https://en.wikipedia.org/wiki/Aggregate_(data_warehouse)

3.5 Slice a Dataset

With the supplemented database from Section 3.4, the decision of how to proceed with a dataset becomes a subjective choice by many data scientists, for they typically choose to perform their ad-hoc querying on manually extracted samples of data in a domain-specific manner [14]. However, domain expertise is required to find the right slices and may not scale if there are many features [16]. Such solutions result in tedious workflows and a lot of human efforts, yet subjective analytic choices influence research results [55]. In this section, we'll address the data slicing problem by applying some data mining techniques, *i.e.*, progressive itemset mining [37] to systematically discover the large slices with sufficient support, where the slicing can be done on one or more dimensions.

Table 3.2: Example data slices for *Sells*

data slice	*support*
(*Beers*.manf, 'Anheuser-Busch')	6
(*Bars*.addr, 'New York')	5
(*Beers*.manf, 'Anheuser-Busch') $\wedge$ (*Bars*.addr, 'New York')	4
(*Beers*.manf, 'Coors Brewing Company') $\wedge$ (*Bars*.addr, 'San Francisco')	3
(*Beers*.manf, 'Anheuser-Busch') $\wedge$ (*Bars*.name, 'Seven Bamboo')	2

For example, a slice of sells of beers from Anheuser-Busch at bars in New York is expressed as a conjunction of descriptors (*Beers*.manf, 'Anheuser-Busch') $\wedge$ (*Bars*.addr, 'New York'). Each discriptor is an attribute-value pair. Some slices for the example *Sells* data are given in Table 3.2. A slice needs to be *large* in the sense that it has sufficient population, which is measured in terms of *support*, as shown in the right column of Table 3.2.

The problem is to automatically divide a given large multidimensional dataset into all possible data slices with sufficient support based on its dimensions. Each data slice provides a distinct perspective to look at the given dataset. The more perspectives we have looked into, the clearer the understanding of a dataset we have. The problem is exacerbated by the exponential nature of the combination of descriptors from all dimensions, especially when the number of dimension of a dataset is large. To solve this problem, we utilize the

progressive miner in Chapter 2 to slice a single entity or relationship table, or multiple tables based on the entity-relationship modeling, which can generate early partial complete results at any time during the long mining process, specifically, it can generate a set of large data slices with a certain time-dependent minimum support.

3.5.1 Data Slicing and Frequent Itemset Mining

Table 3.3: Conceptual representation of example bar-beer-drinker database.

(a) *Beers*

beer	items
b_1	$\{x_1\}$
b_2	$\{x_1\}$
b_3	$\{x_2\}$
b_4	$\{x_2\}$

(b) *Bars*

bar	items
bar_1	$\{y_1, y_3\}$
bar_2	$\{y_2, y_4\}$
bar_3	$\{y_2, y_5\}$
bar_4	$\{y_1, y_6\}$

(c) *Drinkers*

drinker	items
d_1	$\{z_1, z_3\}$
d_2	$\{z_2, z_4\}$
d_3	$\{z_1, z_5\}$
d_4	$\{z_2, z_6\}$

(d) *Sells*

bar	beer	price
bar_1	b_1	5
bar_1	b_3	7.5
bar_1	b_4	4.25
bar_2	b_1	5.5
bar_2	b_2	6
bar_2	b_3	7
bar_3	b_1	5.75
bar_3	b_2	6
bar_4	b_3	6.75
bar_4	b_1	5.25

(e) *Frequents*

drinker	bar	spending
d_1	bar_1	35
d_1	bar_4	65
d_1	bar_2	25
d_2	bar_2	50
d_2	bar_3	60
d_3	bar_1	30
d_3	bar_4	75
d_4	bar_1	30
d_4	bar_2	45
d_4	bar_3	25

In Section 3.2.2, we defined that an item in a relational database corresponds to a *descriptor* of the form (*attribute*, *value*). For the example database given in Table 3.1, we can denote each possible attribute and value pair as an item, *i.e.*, x_i, y_i and z_i represent

Table 3.4: Data slicing and market basket analysis

(a) Slices from *Sales*

slice	*support*
$\{x_1\}$	6
$\{y_2\}$	5
$\{x_1, y_2\}$	4
$\{x_2, y_1\}$	3
$\{x_1, bar_2\}$	2

(b) Market Basket

tid	items
1	{milk, bread}
2	{butter}
3	{beer, diapers}
4	{milk, bread, butter}
5	{bread}

(c) Frequent Itemsets

itemset	support
{bread}	3
{milk, bread}	2
{butter}	2
{milk, bread, butter}	1
{beer, diapers}	1

each possible item from the three entity tables, *Beers*, *Bars* and *Drinkers*, respectively. For instance, x_1 corresponds to (*Beers*.manf, 'Anheuser-Busch'), y_1 corresponds to (*Bars*.addr, 'San Francisco') and z_1 corresponds to (*Drinkers*.addr, 'San Francisco') and so on.

In this way, the conceptual representation of Table 3.1 is displayed in Table 3.3 and the example slices given in Table 3.2 can be reproduced in Table 3.4a. Compared with the frequent itemsets in Table 3.4c for the market basket analysis example given in Table 3.4b, it's easy to verify that the data slices have similar form. Thus, the data slicing problem in a multidimensional database can be formulated as the frequent itemset mining problem [1]: to find all the frequent combination of attribute-value pairs from its dimensions.

As datasets grow and computations become more complex, response time suffers and data exploration is severely hampered [21]. To address this problem, a new computation paradigm has emerged in the last decade: progressive approach [27, 59, 21, 37]. It consists of splitting long computations into a series of approximate results improving with time; in this process, partial or approximate results are then rapidly returned to the user and can be interacted with in a fluent and iterative fashion.

ALPINE [37] is such an progressive algorithm proposed for long mining tasks. It progressively mines itemsets and closed itemsets "support-wise". It can guarantee that all itemsets with support exceeding the current checkpoint's support have been found before it proceeds further. We will utilize the algorithm here to slice large multidimensional database progressively, *i.e.*, first generate slices with higher support, then gradually outputs slices with lower support.

3.5.2 Slice Data in Single Table

When ALPINE is applied to slice the data in a single table, we can code the (*attribute*, *value*) pairs from its dimensions as described in Section 3.2.2. One example is shown in Table 3.3, for each single table in the example database, all the (*attribute*, *value*) pairs have been encoded to distinct items, *i.e.*, x_i, y_i, z_i for table *Beers*, *Bars* and *Drinkers*, respectively.

Based on the coded items, we can directly apply the ALPINE Algorithm (Algorithm 1) to find all frequent (*attribute*, *value*) combinations for each table, respectively. The only variant lies in the Explorer subroutine at Line 5, for the mutual exclusion relation among the (*attribute*, *value*) pairs from the same attribute, namely, *homologous items* (refer to Definition 3.5.1).

Definition 3.5.1 (Homologous Items) *Two coded items from attribute-value pairs,* $I_1 = (a_1, v_1)$ *and* $I_2 = (a_2, v_2)$ *are homologous iff* $a_1 = a_2$ *and* $v_1 \neq v_2$. *Homologous items are said to be homologous to each other.*

For instance, item x_1 (*Beers*.manf = 'Anheuser-Busch') and item x_2 (*Beers*.manf = 'Coors Brewing Company'), both items are from the same attribute, *Beers*.manf. In general, a beer can only be produced by one of those manufacturers, but not multiple manufacturers. In other words, item x_1 won't co-occur with item x_2. Thus, we can omit to extend an itemset

Algorithm 7 SliceExplorer(Minimum itemset P, Maximum itemset Q, Support index $\mathcal{S}$)

```
1:  Output itemset interval: (P, Q);
2:  for all item j = |ℐ| − 1; j > tail(P); j−− do
3:      if j ∈ Q ∨ j is homologous to any item in P then
4:          continue; // no need to extend with item ∈ Q or homologous items
5:      end if
6:      R ← P ∪ {j};
7:      S ← R* ∪ Q;
8:      if sup(R) is already indexed in 𝒮 then
9:          Add (R, S) to the indexed bin;
10:     else
11:         Create a new bin with support sup(R) for 𝒮 and add (R, S) to it;
12:     end if
13: end for
```

with such homologous items in the Explorer subroutine (see Line 3 - 5 in Algorithm 7). The result will be the set of large slices in each single table in the database. However, there might be multiple tables in a multidimensional database as shown in Table 3.1, we'll address the data slicing problem across tables in the next subsection.

3.5.3 Slice Data across Multiple Tables

Available data mining technology usually applies to centrally stored data in one single repository [1, 31, 37, 65, 71]. However, information may be dispersed among different tables like the Enter-Relationship model shown in Figure 3.2. It would be necessary to compute first the join of the relationship table with its involving entity tables to form a single table, *i.e.*, $T = Sells \bowtie Bars \bowtie Beers$.

This approach has two drawbacks: 1) the join is a very cost-expensive operator and a joined table has many more columns and rows which needs a lot of space; 2) it can produce rules which may not reflect accurately the actual relationships existing in data. For instance, with a minimum support threshold set to 0.3, the item for *Beers*.name = 'Budweiser' won't be frequent in the example given in Table 3.1, for its support is $\frac{1}{4}$ in the *Beers* table, while in the joined table, this item will be frequent with support equal to $\frac{4}{10}$ for the beer Budweiser has occurred four times in the *Sells* table. In consideration of both reasons, we should avoid using the join operation.

In contrast to first join all the involving entity tables with the relationship table to form a single table, we exploit the inter-table foreign key reference relationships to adapt the ALPINE algorithm for slicing data across multiple tables. This process will be explained by an example, for instance, we would like to slice the *Sells* data based on the descriptor *Beers*.manf = 'Anheuser-Busch' for the running example in this chapter, where the slicing condition is from a dimension table, but not in the relationship table.

Firstly, we'll compute a projection of the relationship table *Sells* in Table 3.3d onto its foreign key beer referencing the *Beers* table. For each value of b_i, we'll find the set of

records in the relationship table having beer = b_i (An id filed is added to identify each record in the relationship table). This projection will result in k lists of ids. Note that k is equal to the number of rows in the referenced entity table, *Beers* in this example and where $k = 4$. We can find the four projected id lists for beers in Table 3.5.

Table 3.5: Project the relationship table *Sells* to its foreign key beer

id	bar	beer	price
1	bar_1	b_1	5
2	bar_1	b_3	7.5
3	bar_1	b_4	4.25
4	bar_2	b_1	5.5
5	bar_2	b_2	6
6	bar_2	b_3	7
7	bar_3	b_1	5.75
8	bar_3	b_2	6
9	bar_4	b_3	6.75
10	bar_4	b_1	5.25

$\Rightarrow$

(a) beer = b_1

id
1
4
7
10

(b) beer = b_2

id
5
8

(c) beer = b_3

id
2
6
9

(d) beer = b_4

id
3

From the *Beers* table, we can find the set of transactions with *Beers*.manf = 'Anheuser-Busch' (highlighted in red in Table 3.6), that is, $\{b_1, b_2\}$. A union of the projected id lists for those transactions will give us the desired data slice. Specifically, in the example, transaction b_1 and transaction b_2, so we will union the id lists for beer = b_1 and beer = b_2, as indicated in Table 3.6. The resulting data slice for *Sells* satisfying the condition *Beers*.manf = 'Anheuser-Busch' is shown in Table 3.6c.

Table 3.6: Union the projected id lists to get the data slice for *Beers*.manf = 'Anheuser-Busch'

beer	manf
b_1	Anheuser-Busch
b_2	Anheuser-Busch
b_3	...
b_4	...

$\Rightarrow$

(a) beer = b_1

id
1
4
7
10

$\cup$

(b) beer = b_2

id
5
8

$\Rightarrow$

(c) manf = 'Anheuser-Busch'

id
1
4
5
7
8
10

In this way, we have avoided the expensive join between tables for slicing across multiple tables in a multidimensional database. When the slicing condition (*attribute*, *value*) is from the relationship table, the ALPINE algorithm can be applied directly.

3.6 Generate Plots in A Data Slice

The global slice operator can find all the large data slices with sufficient support from a multidimensional dataset as described in Section 3.5, while the goal of a boundless analytics engine is to generate all possible 1-dimensional and 2-dimensional plots among the attributes. The gap is connected by the local plot operator, which operates on each large data slice output from the slicing process and does the univariate and bivariate analysis for all objects in the data slice. In this section, we'll describe how to do the analysis and generate those plots, after that the score of each plot can be calculated.

3.6.1 Univariate Analysis

Univariate analysis is perhaps the simplest form of statistical analysis. The key fact is that only one variable is involved. It doesn't deal with causes or relationships and it's major purpose is to describe [20]. For each data slice, the univariate analysis tries to describe the objects within this group or subgroup.

For discrete variables, the frequency distribution is an organized tabulation/graphical representation of the number of individuals in each category on the scale of measurement [29]. A frequency distribution shows us a summarized grouping of data divided into mutually exclusive classes and the number of occurrences in a class. For the distribution of a discrete variable in a data slice, each entry contains the frequency or count of the occurrences of values within this slice, for instance, the distribution of the manufacturers of beers. Graphically, we represent it as a bar graph, where the horizontal (x) axis represents the values and the vertical (y) axis represents the frequencies or counts for those values.

For continuous variables like *Sells*.price, we'll plot a histogram instead. To construct a histogram, the first step is to "bin" (or "bucket") the range of values, that is, divide the entire range of values into a series of intervals and then count how many values fall into each interval [64]. The bins are usually specified as consecutive, non-overlapping intervals of a variable. Each bin has its area proportional to the frequency of cases in the bin. The last step is to plot the histogram with suitable scales for each axis.

3.6.2 Bivariate Analysis

Bivariate analysis is the simultaneous analysis of two variables (attributes) X and Y, to explore the concept of relationship between the two variables [6]. Graphs that are appropriate for bivariate analysis depend on the type of variables. For two continuous variables, a scatter plot is a common graph. When one variable is discrete and the other is continuous, side-by-side box plots are common and when both variables are discrete a mosaic plot or heat map is common.

The results from bivariate analysis can be stored in a two-column data table. When both variables are from the same table, it can be either an entity table or a relationship table, firstly a SQL selection [26] like operation can get all the objects satisfying the conditions for a data slice, then a SQL projection [26] like operation can project those records to a two-column data table, where each row contributes a data point in the form of (X_i, Y_i) in the two dimensional space. From which, the correspondence and relationship between these two variables can be directly established.

However, when the two variables, X and Y, are from two different tables, they might differ in the number of records. The two-column data table cannot be directly extracted from the original tables, as a result, we couldn't establish the relationship between them. Similar to slice data across multiple tables in Section 3.5.3, we utilize the inter-table foreign key reference relationships to establish the mapping and get the plot of X and Y.

If variable X is from an entity table $R[e]$ and variable Y is from a relationship table $R[r]$, for instance, we would like to establish the relationship between *Beers*.manf and *Sells*.price in Table 3.1. Firstly, we will project the relationship table $R[r]$ onto its foreign key referencing the entity e. In this example, we project the *Sells* table to its foreign key beer as displayed in Table 3.5. Then we augment the relationship table with an attribute X, where each subgroup of the projected id lists will have the same value for attribute X from the entity table. As to the example, the process is shown in Table 3.7 and the resulting manf column is displayed in Table 3.7e. Thereafter the box plots between manf and price can be generated.

Table 3.7: Augment the values for attribute manf with the same value for each projected id list

beer	manf
b_1	Anheuser-Busch
b_2	Anheuser-Busch
b_3	Coors Brewing Company
b_4	Coors Brewing Company

+

(a) beer = b_1

id
1
4
7
10

(b) beer = b_2

id
5
8

(c) beer = b_3

id
2
6
9

(d) beer = b_4

id
3

(e)

$\Rightarrow$

id	bar	beer	manf	price
1	bar_1	b_1	Anheuser-Busch	5
2	bar_1	b_3	Coors Brewing Company	7.5
3	bar_1	b_4	Coors Brewing Company	4.25
4	bar_2	b_1	Anheuser-Busch	5.5
5	bar_2	b_2	Anheuser-Busch	6
6	bar_2	b_3	Coors Brewing Company	7
7	bar_3	b_1	Anheuser-Busch	5.75
8	bar_3	b_2	Anheuser-Busch	6
9	bar_4	b_3	Coors Brewing Company	6.75
10	bar_4	b_1	Anheuser-Busch	5.25

If variable X and variable Y are from two different entity tables $R[e_1]$ and $R[e_2]$, which are connected by some relationship table $R[r]$, *i.e.*, *Beers*.manf and *Bars*.addr connected by the relationship *Sells*. In this case, we need to project $R[r]$ to both key of e_1 and key of e_2, and augment both attributes like the process in Table 3.7. After that, the relationship between the two variables can be established.

3.7 Summary

In this chapter, we described the boundless data analytics process with motivation, goal and procedures. A boundless analytics framework is proposed to automatically discover all possible large data slices and generate an unbounded number of univariate and bivariate plots. This framework consists of three major components, namely, the pre-aggregation process, the global slice operator and the local plot operator. For pre-aggregation, an entity-relationship model based approach is proposed to calculate attribute aggregates, which increase the complexity and expressiveness of the data model, for new dimensions and measurements can be derived. For data slicing, we encode each distinct attribute-value pair as an item and formulate the large data slice discovery problem as the frequent itemset mining problem, where the progressive mining paradigm introduced in Chapter 2 can be employed. Besides, the inter-table foreign-key reference relationship is exploited in the slice and plot operators without joining the involving tables to form a single big table.

Chapter 4

Boundless Analytics System

In this chapter, we build a boundless analytics system and illustrate how our system works by example of the historical NBA stats data from Kaggle[1]. In particular, we present (i) a boundless analytics engine which can slice the data in all possible ways and generate the plots among the attributes (including pre-aggregates) for all large data slices; (ii) an Apache Solr search platform based plotbase to organize and index all the generated plot objects; (iii) a frontend webapp search system to explore the data in various ways and return the plots sorted nicely according to some interestingness measure. The system can provide a progress meter in terms of the minimum size of data slices analyzed at any time and some initial analysis results returned from the system are very promising.

4.1 System Overview

In this section, we give an overview of the boundless analytics system, then we will introduce each module of the system in detail in the following sections: Section 4.2 - Section 4.5. In Figure 4.1, we outline the boundless analytics system into layers, *i.e.*, the data module, the

[1] https://www.kaggle.com/drgilermo/nba-players-stats

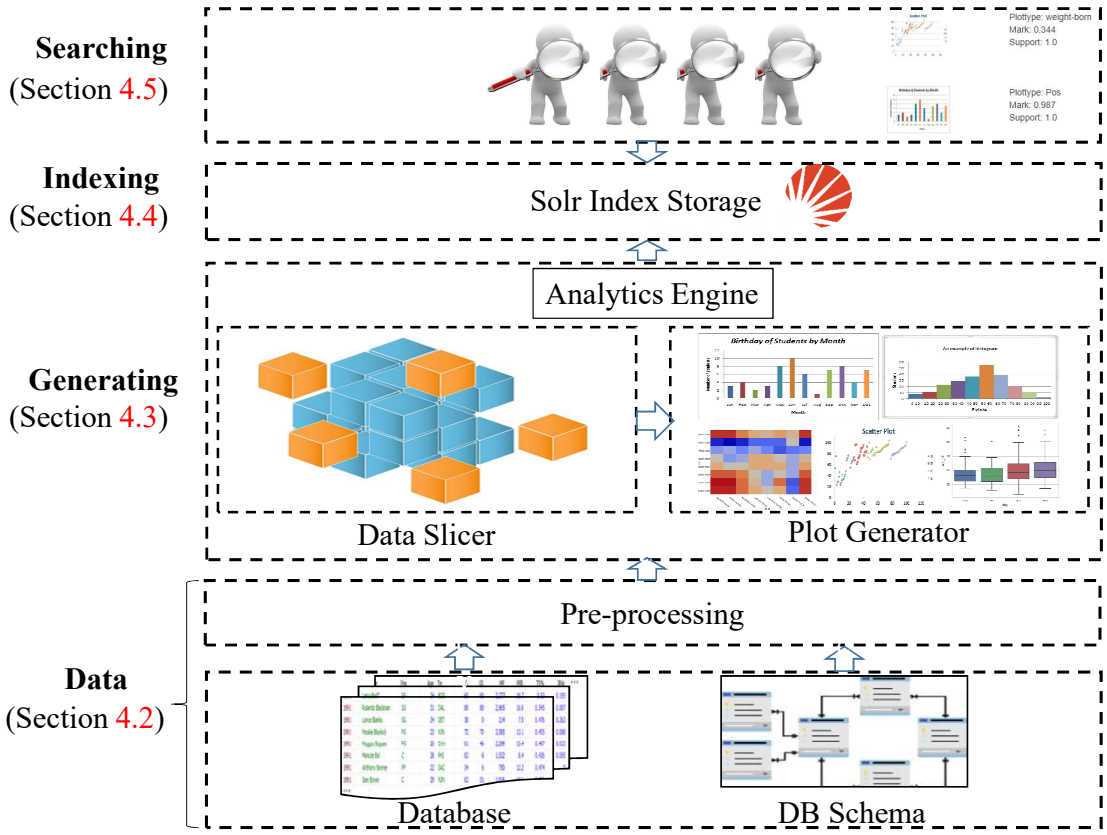


Figure 4.1: Overview of the boundless analytics system. It consists of the data module, the generating module, the indexing module and the searching module.

generating module (the analytics engine), the indexing module (Solr[2] index storage) and the searching module (the webapp).

For the data module, the input is the set of relational tables for each entity and relationship in the database along with its schema specification, which goes through a pre-processing step to (1) filter the noisy data, (2) generate the pre-aggregated measurements by the SQL group by operation on the participating relationships for each entity in the input database, which will be introduced in Section 4.2.

In Section 4.3, we elaborate the generating module (the analytics engine), which consists of two major components, the data slicer and the plot generator. The data slicer will apply the progressive mining algorithm for each single table and across multiple tables to progressively find all possible large data slices in the dataset. Then for each large data

[2] http://lucene.apache.org/solr/

slice output from the slicer, a plot generator will do the univariate and bivariate analysis for the data belonging to the slice and output all the plot objects. Each plot object will be represented as a json object and a plot API can render a plot object into a graphical chart.

The open source enterprise search platform, Solr, from the Apache Lucene project, has a lot of valuable features, including full-text search, faceted search, NoSQL features and rich document handling. In Section 4.4, we utilize the Solr platform to organize and index all the generated plot objects in two layers: (1) the plot type layer, which groups all the plots involving the same set of attributes together with some information about the group, *i.e.*, the number of plots, the score distribution, the gini index and variance of scores of plots within this group, (2) the plot layer, which includes all the information about each plot, like its plot type, score, support, slice information as well as all the data needed to render a graphical chart. Thanks to the full-text search feature of Solr, this module powers the web search interface for various kinds of queries from the end-users.

The last part of this system, is the searching module. The search interface does not require any data science expertise or experience from the end-users, they can search for their interested plots just like doing a web search with keywords. They can query for plot types, or plots with a score range, with a minimum support threshold, for a specific season, or for a specific team or player, or other slicing conditions. Some demonstration queries with the query response will be displayed in Section 4.5.

The system flowchart is depicted in Figure 4.1 (see the arrows). The data module feed the generating module with processed data, where the data slicer mines the large data slices for the plot generator. All the plot objects generated by the analytics engine will be sent to the indexing module, which index every field of them. Finally, the end-users query the plots stored in Solr index storage through the front-end interface in the searching module.

4.2 The Data Module

The data module will feed the boundless analytics system with data. The input data is composed of two parts: database and the db schema. For the database part, each entity or relationship will be represented as a relational table in a csv file, where the header gives the attribute names for each field. In the db schema, we need to specify more information about the input databse, like the database name, number of tables in the database, entity tables, relationship tables, etc. For each table, we need to tell the system the property of each attribute, like whether it is numerical or it is categorical. Besides, the primary key and foreign key reference information among the tables is also entered into the system. All the schema information of a database is written in a json file.

4.2.1 The NBA Players Stats Data

We illustrate how our system works by example of the NBA stats data from Kaggle[3], which is scraped from Basketball-reference[4]. The database contains individual statistics for 67 NBA seasons since 1950 for over 3000 players. It includes over 50 features per player, from basic attributes such as points, assists, rebounds to more advanced features like value over replacement, etc. The database is represented as the E-R diagram in Figure 4.2 with tables:

Players(Player, height, weight, college, born, birth_city, birth_state)

SeasonsStats(Year, Player, Pos, Age, Tm, G, GS, MP, PER, TS%, 3PAr, FTr, ORB%, DRB%, TRB%, AST%, STL%, BLK%, TOV%, USG%, OWS, DWS, WS, WS/48, OBPM, DBPM, BPM, VORP, FG, FGA, FG%, 3P, 3PA, 3P%, 2P, 2PA, 2P%, eFG%, FT, FTA, FT%, ORB, DRB, TRB, AST, STL, BLK, TOV, PF, PTS)

where the meaning of each column can be referred to the original Kaggle page or the basketball glossary[5].

[3] https://www.kaggle.com/drgilermo/nba-players-stats
[4] https://www.basketball-reference.com/
[5] https://www.basketball-reference.com/about/glossary.html

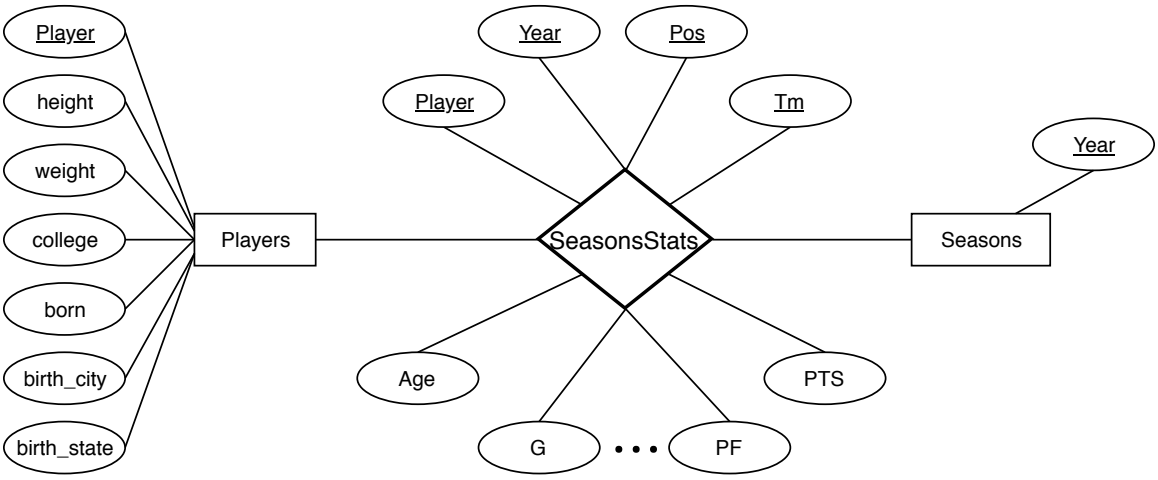


Figure 4.2: The entity-relationship diagram of the NBA players stats data.

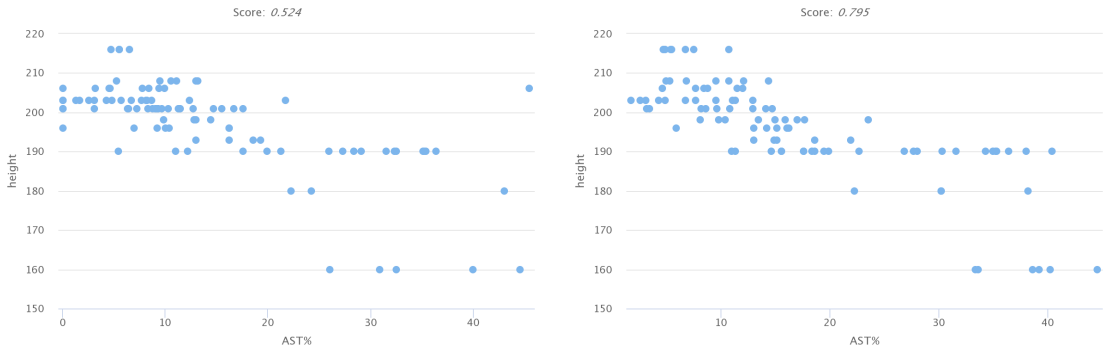
4.2.2 Data Pre-processing

Data Cleaning

According to Basketball-reference, there are some minimum rate statistic requirements[6]. For instance, to be a league leader in a category in the NBA, a player need to play for a minimum of 1500 minutes in a season; to qualify for free throw percentage (FT%), a player need to have 125 free throws. In this dataset, we find that on average there are 75 players who played less than 500 minutes in a season, that is, less than 6 minutes per game. We don't want these players to skew the distribution of an attribute or the relation among the stats, which might hide a useful pattern or trend in the noisy data. To solve this, we filter out players that don't play enough minutes per season, *i.e.*, 500 minutes.

One example is shown in Figure 4.3, we can identify that the trend between assist percentage and the player height is more obvious after the data cleaning, *i.e.*, the correlation coefficient between them increases from 0.524 to 0.795. The higher a player, the less likely the player will assist someone else while he is on the floor.

[6] https://www.basketball-reference.com/about/rate_stat_req.html



(a) (b)

Figure 4.3: The relation between assist percentage and player height (a) before and (b) after the data cleaning.

Algorithm 8 Preaggregate(Entity set E, Relationship set R)

for all Entity $e \in E$ **do**
  $R^*[e] \leftarrow R[e]$
  for all Relationship $r \in$ the participating relationships $FK[e]$ **do**
    Group the relational table $R[r]$ by foreign key e
    for all Attribute a in table $R[r]$ **do**
      if Attribute a is continuous **then**
        Add aggregated attribute $avg_e(a)$ to $R^*[e]$
      else
        Add aggregated attribute $count_e(distinct\ a)$ to $R^*[e]$
      end if
    end for
  end for
  Return the supplemented table $R^*[e]$
end for

Pre-aggregation

As described in Section 3.4, the pre-aggregation process will expand the entity tables $R[e]$ into $R^*[e]$ with additional aggregated attributes resulting from participation of e as foreign key in relationships in $FK[e]$ (Algorithm 8). For the NBA Players Stats Data, the entity table is *Players* and the relationship table is *SeasonsStats*. We'll group the seasons stats by players and supplement the *Players* table with the aggregated measures like number of seasons a player participated in, number of positions played, average age during all participating seasons, number of teams served, average number of games played, average number of games started, average minutes played etc.

4.3 The Generating Module

The generating module is the engine of the whole boundless analytics system, which progressively slices the multidimensional database in decreasing support and generates all the univariate and bivariate plot objects (1) for attribute(s) from each single entity or relationship table and (2) for attributes across multiple tables. It consists of two components: the data slicer and the plot generator. The data slicer continuously supply the plot generator with large data slices in lower and lower support; while the plot generator will do the univariate and bivariate analysis for each slice and generate those 1-D and 2-D plot objects.

To avoid the expensive operation to join each relationship table with all the referenced entities to form a large relational table with more columns (including both attributes from the involving entities as well as the relationship attributes) and to avoid generating superficial relations due to the fact that an entity has occurred in a relationship table multiple times, we abandon the straight-forward way of joining. Instead, to adapt the progressive mining algorithm, ALPINE, to find all the large data slices for a database with multiple relational tables, *i.e.*, both the entity tables and the relationship tables, we suggest a two-phase strategy to solve the problem.

4.3.1 The Two-Phase Approach

Our algorithm mines the large data slices and generates the plot objects in two phases:

Phase I: run the analytics engine on each single entity or relationship table

1. Slice the table using the progressive miner, ALPINE, based on its dimensions. Each attribute-value pair from the dimensions of the table is encoded as an item and the support index structure in ALPINE is initialized from those single items. Then we can directly apply the ALPINE algorithm (Algorithm 1) with the SliceExplorer in Algorithm 7 to identify all the frequent combinations of attribute-value pairs.

2. Generate the univariate and bivariate plot objects in each data slice. For all data belongs to the data slice, we'll generate the five concerned types of plots, *i.e.*, bar graph, histogram, heatmap, scatter plot and side-by-side boxplots, from each single attribute or two-attribute pair from the table.

Phase II: run the analytics engine across multiple tables

1. Compute the projection of each relationship table to its foreign keys. Access a relationship table $r \in R$ to find the occurrences of each value for each foreign key e_i. Store the set of supporting transactions as a *tidlist*. Note that for a foreign key, the number of projected *tidlist*s is equal to the number of records in the referenced entity table. For instance, in the NBA stats data, we project the relationship table *SeasonsStats* to its foreign key *Player* as shown in Figure 4.4, a *tidlist* will be generated for each player.

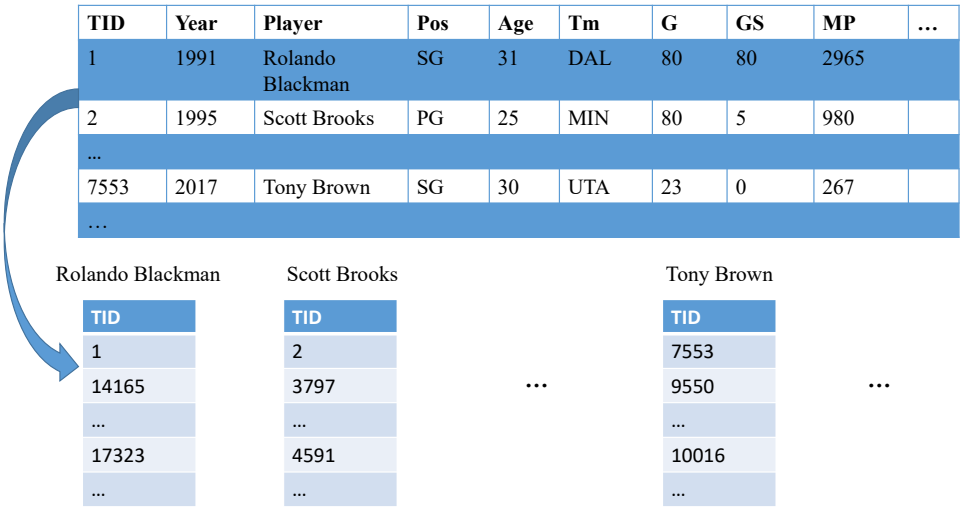


Figure 4.4: Project the relationship table *SeasonsStats* to its foreign key *Player*.

2. Progressively slice each relationship table across multiple tables through the foreign key reference (refer to Section 3.5). If a slicing attribute is from the relationship table, we can slice the data directly by find all records with the attribute equal to some specific value; otherwise, if the slicing attribute is from one referenced entity table, the slicing cannot be done directly. We need to first identify the set of values for the foreign key which satisfy the

slicing condition *attribute* = *value*, then a union of all the *tidlist*s for those values will give us the desired data slice. In Figure 4.5, we show an example to slice *SeasonsStats* based on *birth_state* = "New York".

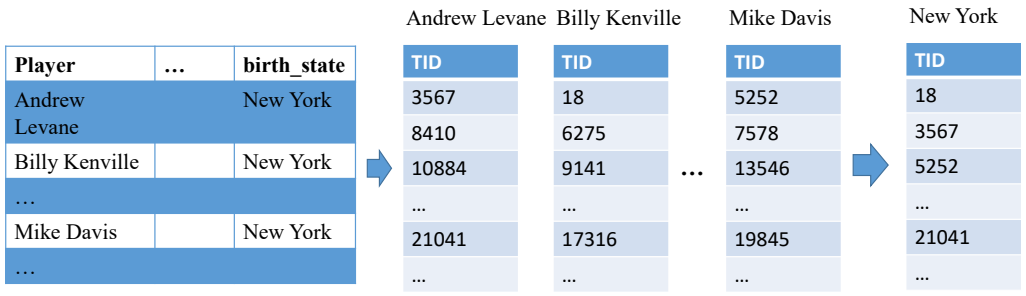


Figure 4.5: Slice the relationship table *SeasonsStats* based on an entity attribute *birth_state* through the foreign key reference.

3. Generate all the bivariate plot objects across relationship table and referenced entity tables (refer to Section 3.6). No plots for attribute(s) from a relationship will be generated, for they have already been generated in Phase I. In this phase, we only focus on the relation between attributes from the relationship table and the referenced entity tables. Since the two attributes are from different tables, they might differ in the number of records, the correspondence between them can not be established directly. Similar to Step 2, firstly we get the projected *tidlist*s for each value of the foreign key, then we augment the attribute for tuples in each *tidlist* with the same value, thus the relation between the two attributes can be plotted. To establish the relation between the player *Height* to the season statistics *field goals* (*FG*) for the slice shown in Step 2, we need to first augment attribute *Height* to have the same number of records as the relationship table as shown in Figure 4.6.

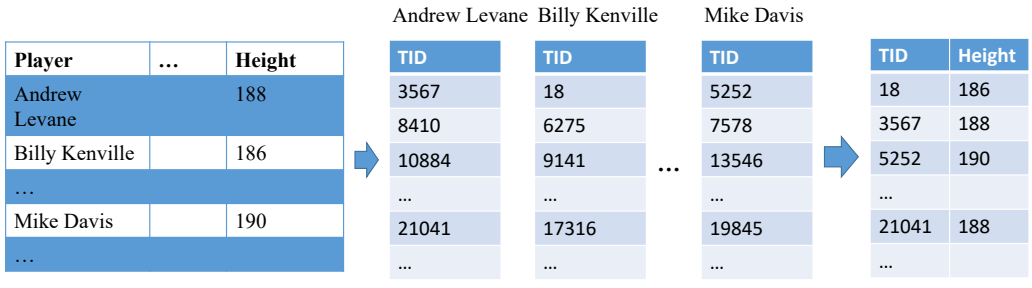


Figure 4.6: Augment the values for an entity attribute *Height* with the same value for tuples in each *tidlist*.

For the NBA Players Stats Data used in this chapter, in Phase I, we generate all the plots about the entity table *Players* or the relationship table *SeasonsStats*. For instance, the geographic distribution of the birth place of all the players or the relation between player height and their average win shares for table *Players*, or the distribution of rebounds or the relation between the minutes played and their points for table *SeasonsStats*. In this phase, 183,122 plot objects are generated. In Phase II, the plots for attributes across the two tables are generated, like the relation between the player height and their assist percentage in games. 434,121 plot objects involving attributes from multiple tables are generated in the second phase for the NBA dataset. In total, the analytics engine generates 617,243 plot objects (with a minimum support threshold of one percent) for the NBA stats data.

4.3.2 The Representation of A Plot Object

Each plot can be represented as a chart, then to explore the plots will be like an image retrieval process. To query those plots by their "content", we need to store a lot of tags for each plot, for instance, the score (refer to Section 3.3) and support of a plot, the attribute(s) involved in the plot, the data slice information for the plot, etc. In regard to the huge number of plots generated, *i.e.*, 617,243 plots for the NBA stats data, the charts (images) along with the tags will take a lot of space, which largely affects the scalability of the boundless analytics system. From this perspective, we make each plot a json object with all the tags and required data to render a chart, one example plot object is displayed below:


```
{
  "dataset": "NBAPlayers",
  "table": "Players",
  "plottype": "height",
  "dimensions": 1,
  "x": "height",
  "type": "histogram",
  "support": 0.088,
  "score": 0.567,
  "slice": [["birth_state", "California"]],
```

```
  "data": [[179, 3], [181, 5], [183, 2], [185, 6], [187, 0],
      [189, 13], [191, 25], [193, 37], [195, 0], [197, 31],
     [199, 33], [201, 48], [203, 27], [205, 0], [207, 43],
     [209, 34], [211, 19], [213, 13], [215, 0], [217, 3],
     [219, 1], [221, 0], [223, 0], [225, 1]],
  "url": "http://foreveranalytics.com/Render-Chart/..."
}
```


A plot object won't be rendered to a chart only when the user would like to explore it. When a user clicked on a plot, it would be plotted to a chart through a visualizer powered by Highcharts[7]. For the example plot object listed above, which is the height distribution of all players born in California, the corresponding graphic chart is displayed in Figure 4.7.

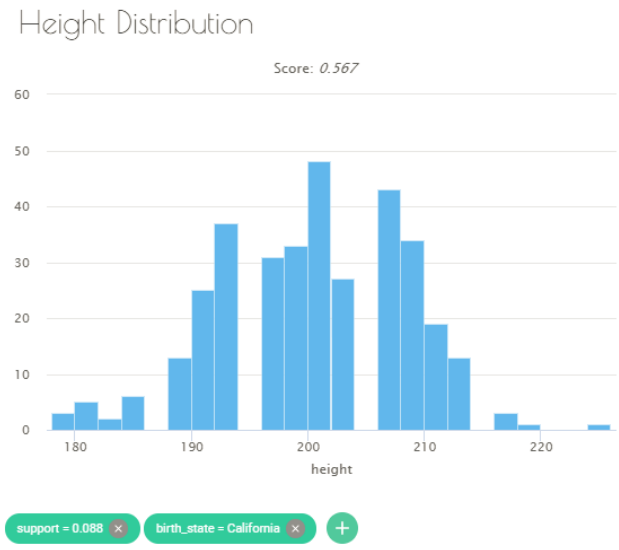


Figure 4.7: An example chart from a plot object.

4.4 The Indexing Module

The indexing module indexes all the plot objects generated by the generating module, providing a plot base for the searching module which directly interacts with the end-users. The Apache Solr is a powerful tool with tremendous search capability, supporting indexing, querying, mapping and ranking [33], which is employed here to power the boundless

[7] https://api.highcharts.com/

analytics system. The basic data structure for data being fed into a Solr index is a document containing multiple fields, each with a name and containing content, which may be empty, like the json plot object shown in Section 4.3.

The boundless analytics engine based on some pre-defined scoring function (Section 3.3) to provide "high-quality" plots might not well satisfy the possible multiple information needs of users, for there is no universal standard for the quality of a visual plot. The proposed scoring function based on their spread favors "inequality" or "correlation", while what the analyst seeks might be something different. As an alternative, the plot diversity can be promoted to comprise various needs, which describes how a distribution or relation varies across various factors. Thus, we group all the plots involving the same set of attributes across the data slices together, which we call it as a plot type. Then the Solr powered plot base organized the plot objects in two layers: the plot type layer and the plot layer.

4.4.1 Definition of a Plot Type

A plot type is a set of plots. It is the relation of the same set of variable(s) over all possible data slices. Say, we take the joint distribution P of two attributes X and Y. P is an example of a plot type. You can think of the remaining attributes, a_1, ..., a_n, as parameters of plot type P. Now we instantiate P over all possible slices made from the attribute-value combinations from a_1, ..., a_n. Each slice defines simply a subset of the original data over which we observe a plot of type P.

Given a univariate or bivariate plot type P in a multidimensional dataset with n dimensions, P can be parameterized by the slice S with n attribute-value pairs like $(a_1, v_1), (a_2, v_2), ..., (a_n, v_n)$, which can be visualized as a plot, namely, $Plot(P, S)$. Each dimension a_i has its own range of values $\{a_{ij} | j = 1, 2, ..., n_i\}$. An attribute a_i can also have the value $*$, where $*$ represents a "don't care" value, meaning this attribute is ignored. All attributes may be ignored as well - that we just have a slice S which includes the whole data set, not sliced by any attribute.

Then a plot type P is such a relation in the space of all possible combinations of values of a_1, ..., a_n. For each plot type, we can keep the information about each group, *i.e.*, the number of plots (nplots), the type of chart (type), the gini index (metagini) and variance of scores of plots within this group, which measure of the diversity of plots from a plot type. One example plot type is listed here:


```
{
  "dataset": "NBAPlayers",
  "nplots": "158",
  "plottype": "3P%-height",
  "x": "3P%",
  "y": "height",
  "type": "scatter",
  "metagini": 0.633,
  "variance": 0.013,
  "stats": [0.01, 0.15, 0.25, 0.3, 0.54]
}
```


It's a plot type about the relation between 3-point field goal percentage (3P%) and player height, which contains 158 plots in total across various data slices with a score variance of 0.013 and the gini index of scores of plots in this group is 0.633. The five number summary of scores, *i.e.*, the minimum, the first quartile, the median, the third quartile, and the maximum, are listed by the key "stats" and the score distribution of this example plot type is displayed in Figure 4.8. This information will guide the user when exploring and searching the plots for a plot type.

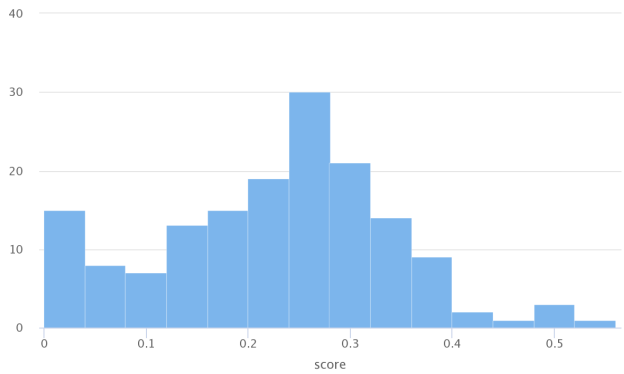


Figure 4.8: The score distribution of plots in the example plot type.

4.4.2 Layered Indexing in the Plot Base

A plot type and a plot object describes a univariate or bivariate relation at different levels of granularities. At the coarse level, the plot types can be ranked based on their diversity (gini index or variance of scores of plots belonging to the plot type); at the fine level, we can sort the plots based on their support or score values. The two layers can be used to serve different needs of users. At the plot type level, it can help the user to quickly locate those interesting relations which varies a lot across different data slices; at the plot level, the users can identify a specific interesting plot with a high score or support. Taking advantage of the search capability of Solr, we index at both levels as shown in Figure 4.9.

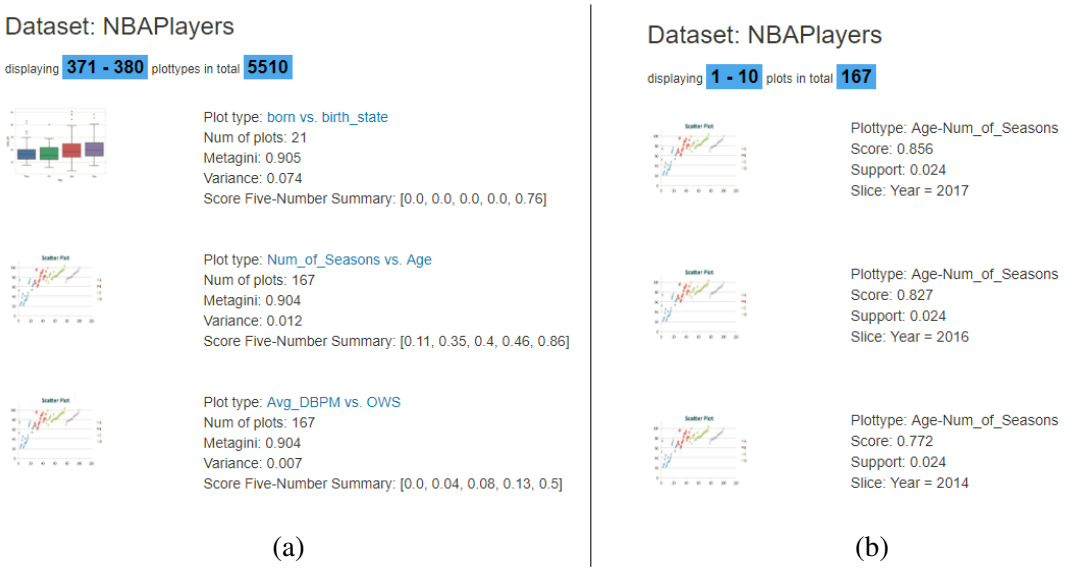


Figure 4.9: Layered indexing of plots in the plot base: (a) by plot type and (b) by plot.

In Figure 4.9a, we can find in total there are 5,510 plot types from the NBA players dataset indexed in the plot base and sorted nicely according to their metagini values, including the original attributes as well as derived features by pre-aggregations, like number of seasons a played participated in (Num_of_Seasons) and average defensive box plus/minus (Avg_DBPM), etc. From the list, we can find relations from single entity or relationship tables like born year vs. birth state, or relations across multiple tables like the average defensive box plus/minus value (Avg_DBPM) vs. the offensive win shares (OWS).

For the plot type of number of seasons and player age, we can find that it contains 167 plot objects with scores in the range of [0.11, 0.86]. When we zoom in to this plot type, we can find all the plots in this group in Figure 4.9b. The plot objects can be ranked by score or by support (in the figure, the plots are ranked by score). The layered organization greatly increases the flexibility of the searching module in the next section.

4.5 The Searching Module

As mentioned in Chapter 3, the data exploration process requires a lot of human efforts in terms of tedious and repetitive programming, especially when the number of dimensions is large. Thus, we would like to build a boundless analytics system to harness the computational resources to automatically find the large data slices and generate all possible univariate and bivariate plot objects. The arisen issue then is that it's impossible to manually exam the numerous plot objects output from the generating module one by one, in consideration of an unbounded number of plots might be produced. Can we reduce the data exploration process to simple search? While the generating module is cranking plot objects, we can query those plots based on variables involved.

In this section we show how indeed we can reduce data analytics to search - by representing data exploration as search of continuously generated (or pre-generated) database of plot objects.

4.5.1 The Search Query Language

A query language in which users can declaratively specify their information needs and the data patterns of interest is needed in the searching module of the boundless analytics system. Our search query language is close to the regular Google query language[8] - basically bags of

[8] https://developers.google.com/issue-tracker/concepts/search-query-language

keywords. Keywords are text strings that you can use to search across the fields of indexed json objects, which will be used to represent:

- Metadata. Metadata will include some but not necessarily all of (a) types of plot, for instance, bar graph, histogram, heat map, scatter plot and box plot etc, and (b) attribute names like 3-point field goals (3P), free throws (FT), 2-point field goals (2P), rebounds, height, weight, player positions (Pos) etc. in the NBA Players dataset. For more attribute names, readers can refer to Section 4.2.

- Data. Data will include data values from the input database. Consider the NBA Players stats data, some possible data values will be like 'Celtics' for team, or 'Lebron James' for player, or 'University of California, Los Angeles' for college, or 'New York' for birth state, etc.

- Numerical conditions on support and score. Those conditions can specify the minimum support, for example, support ≥ 0.05, and range for the interestingness score of a plot, say, $0.50 \leq \text{score} \leq 0.95$.

Searches you perform can contain multiple criteria, including a combination of metadata, data and numerical conditions. Queries will be bags of keywords - some of them are types of plots, others are names of attributes, data values and numerical conditions. The query parser will transform the query to lower level query (like Solr query) if Solr is the underlying storage system. Some example queries for the boundless analytics system are given below:

$\boldsymbol{Q}$ = 'scatter, 3P, 2P, Celtics'

The query would return all scatter plots of 3-point field goals vs. 2-point field goals for all seasons and all Celtics players, but also such plots for each season, each player and each position etc. Each plot will have a score and plots in the answer to this query will be ranked according to some interestingness measure, *i.e.*, score or support.

One can imagine queries which simply ask for a single player, for example

Q = 'Lebron James'

which would return all plots from the plot base about LeBron James. This means for every single season, distributions of NBA stats and relationships between them all limited only to LeBron James.

One may of course limit plots about LeBron James to only plots about number of rebounds and only when he was in Cleveland Cavaliers, this query would look like this:

Q = 'Lebron James, Cleveland Cavaliers, Rebounds'

and will return plots showing rebounds for different seasons he played for Cavaliers etc.

Queries should be as simple as Google queries - and if there is any parsing necessary, it will be done by our system. Thus, we want to limit any syntax which would help the parser but make the learning curve for a user steeper. In particular, we want to avoid using "query languages" with new syntaxes such as Solr language. These languages will play the role of the internal languages which we parse user queries to - before they are executed against our plot object database. But they will never be used directly by the user but be intermediate, executable form of user query in the system.

4.5.2 Query Examples and Discussions

We can address pretty sophisticated data analytics tasks using such simple keyword queries introduced in Section 4.5.1.

Example 1: a user would like to explore all the plots about an attribute, *i.e.*, assist percentage (AST%).

While assists in themselves are kind of useful to look at, assist percentage is a much better statistic to cite when trying to make a case for a player's skill in the passing department, for assist percentage is free from the effects of pace and volume. Assist percentage is an

estimate of the percentage of teammate field goals a player assisted while he was on the floor. The formula is given in Equation 4.1

$$AST\% = \frac{AST}{((MP/(Tm\ MP/5)) * Tm\ FG) - FG} * 100 \tag{4.1}$$

where AST = Assists, MP = Minutes Played, $Tm\ MP$ = Team Minutes Played, $Tm\ FG$ = Team Field Goals, FG = Field Goals. In terms of the search query language introduced in the previous subsection, this query can be written as:

$$\boldsymbol{Q} = \text{‘AST\%’}$$

which will be transformed to the low-level Solr query syntax:

$$\boldsymbol{q} = \text{plottype:AST\%}$$

by our boundless analytics system before actually executing the query against the Solr search platform based plotbase.

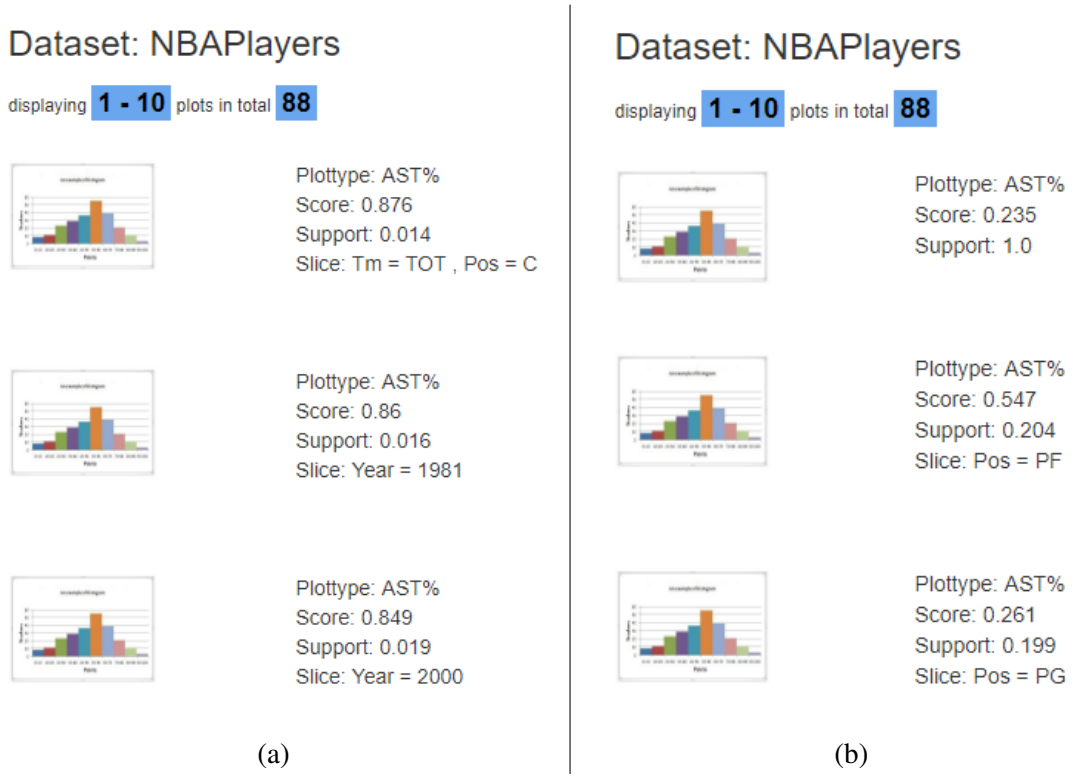
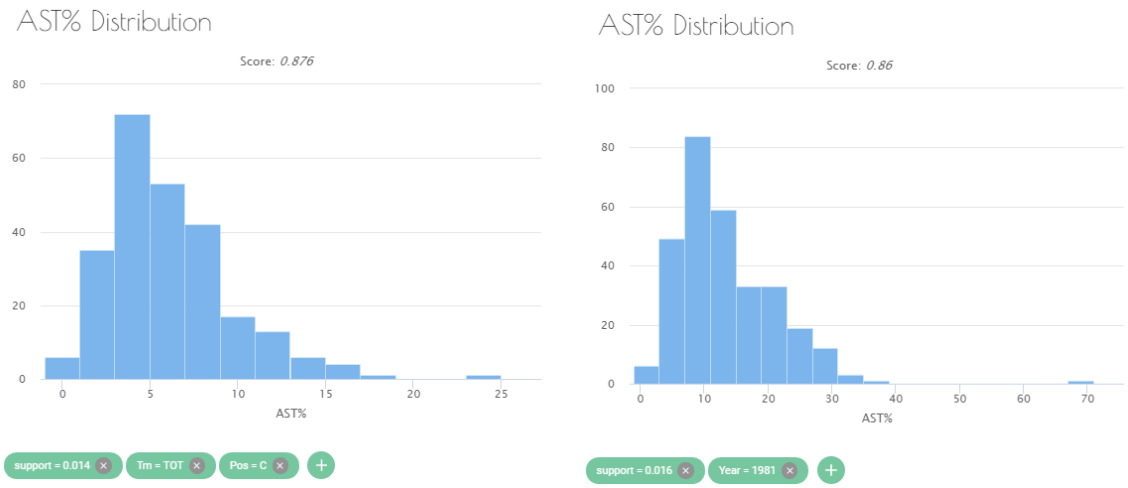


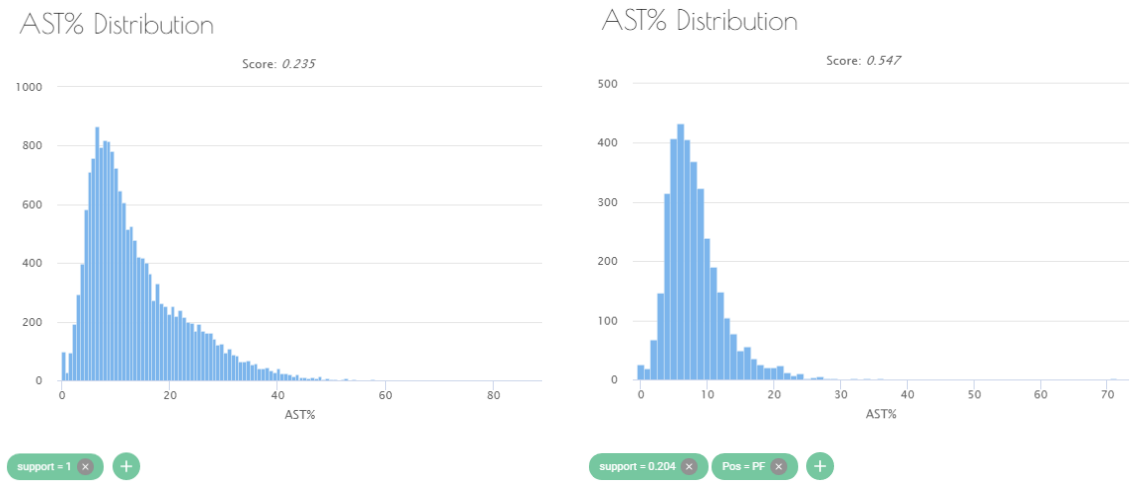
Figure 4.10: Query response for $\boldsymbol{Q}$ = ‘AST%’: (a) ranked by score and (b) ranked by support.

The answer returned is a list of plots about the distribution of assist percentage across all possible data slices, like slice satisfying the conditions Tm = 'TOT' ('TOT' is just the cumulative score from all of the teams a player played for a year) and Pos = 'C' (the center position) or slice with Year = '1981'. Plots in the answer to this query can be ranked based on (a) their score (interestingness) or (b) their support (coverage) as shown in Figure 4.10. In Figure 4.10a, it's easy to verify that the interestingness score is in decreasing order, while the plots are ordered in decreasing support in Figure 4.10b.



(a) (b)

Figure 4.11: Top-2 high-score plots for $\boldsymbol{Q}$ = 'AST%'.



(a) (b)

Figure 4.12: Top-2 high-support plots for $\boldsymbol{Q}$ = 'AST%'.

We can compare the top-2 high-score plots with the top-2 high-support plots for the query, $\boldsymbol{Q}$ = 'AST%', in Figure 4.11 and Figure 4.12, respectively. In Figure 4.11, the top-2 high-score plots with an interestingness score above 0.85 have some "outstanding" bar(s) with a very high frequency, like the assist percentage around 4 on the left and the assist percentage around 9 on the right chart. By contrast, the distribution is more widely spread across the spectrum of assist percentage values in those high-support (high-coverage) plots in Figure 4.12, there are significant occurrences for a range of assist percentage values, *i.e.*, from 4.5 to 11.5 on the left and from 4 to 9 on the right graph.

Based on the slicing conditions for those high-score and high-support plots, we would like to explore how the assist percentage distribution varies among different positions. In terms of the search query language of keywords, the query will be modified to:

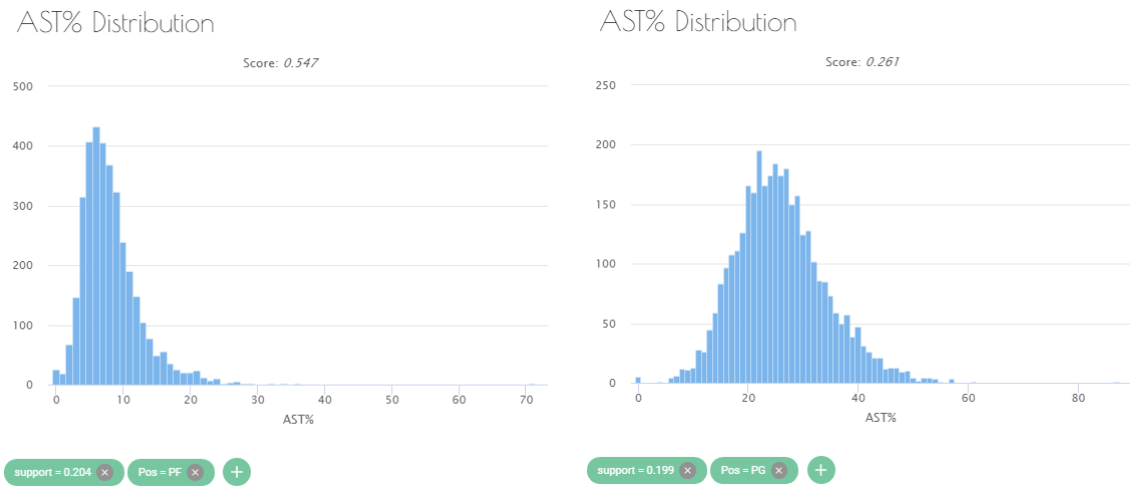
$$\boldsymbol{Q} = \text{'AST\%, Pos'}$$

which will return all plots from the plot base about assist percentage slicing based on the playing position, in addition to plots which show relationships between these two attributes. All plots might also be further segregated based on other attributes, like team, year etc.

(a) (b)

Figure 4.13: (a) The joint distribution of assist percentage and position, (b) The distribution of assist percentage slicing on postions.

The highest score for the relation between assist percentage and position is 0.761, as shown in Figure 4.13a, while the score and support information of the distribution of the assist percentage for the five basketball positions, namely, the power forward (PF), the small forward (SF), the center (C), the shooting guard (SG) and the point guard (PG), are listed in Figure 4.13b. We can compare the top-scored one with the least-scored one among the five positions in Figure 4.14. Both charts are close to the normal curve. For the position power forward (PF) on the left, the standard deviation is small, the curve is tall and narrow; while for the position point guard (PG) on the right, the standard deviation is big, the curve is short and wide.



(a) (b)

Figure 4.14: Comparison of the distribution of assist percentage for position (a) the power forward (PF) and (b) the point guard (PG).

Example 2: a user would like to explore all the plots between two attributes, one is categorical, while the other is numerical. *i.e.*, position (Pos) vs. number of seasons (Num_of_Seasons).

Here the attribute number of seasons a player participated in is calculated during the pre-aggregation process. Does the position played influence the career length of a professional basketball player? Which playing position has the longest career length? To explore answers to this kind of questions, an analytics using our boundless analytics system might query for:

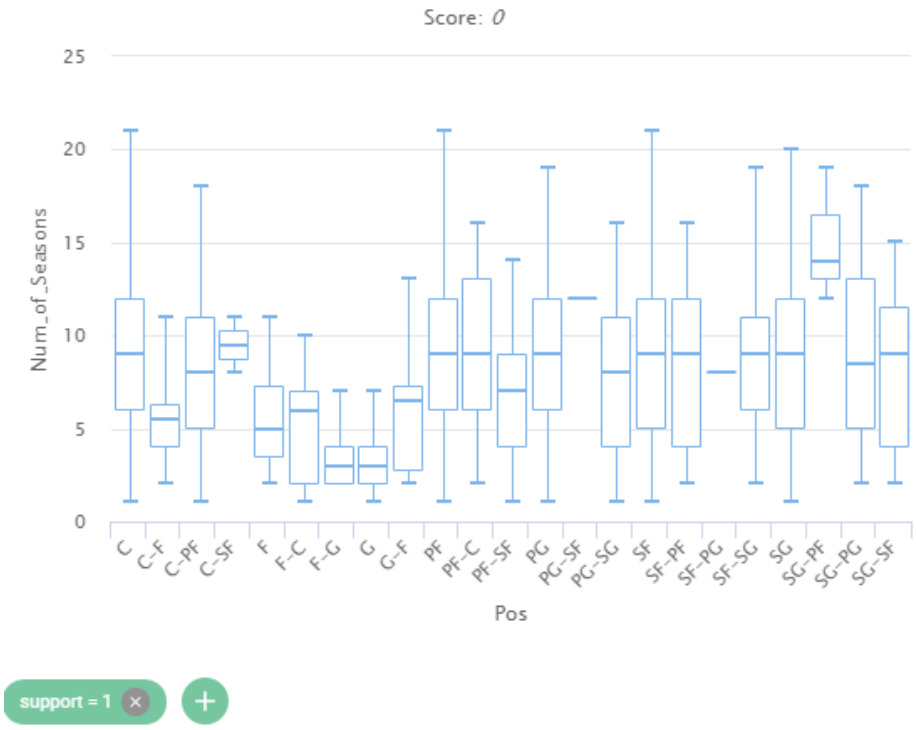


Figure 4.15: The general relationship between attribute Pos and attribute Num_of_Seasons.

$$\boldsymbol{Q} = \text{‘Pos, Num\_of\_Seasons’}$$

The corresponding low-level Solr query is:

$$\boldsymbol{q} = \text{plottype:Pos-Num\_of\_Seasons}$$

The query returns a list of plots satisfying the filtering condition of this query. Firstly, we rank the response by the support values of plots, and find the general trend of the relationship between the two attributes with the largest support in Figure 4.15. In this plot, it displays the side-by-side box plots of how the career length of a player varies by the playing position. We can find that the support of this plot is 1, while the interestingness score is 0, for the majority boxes for different positions have a median value around 9. It seems that position played is not a significant factor to affect the number of seasons a player participated in.

Then we switch to rank the results in decreasing interestingness score, the plots are listed in Figure 4.16, where the top-ranked plots have a pretty high interestingness score, 0.738 for the first plot and 0.654 for the second one. If we further scrutinize the slicing conditions for these high-score plots, it’s easy to verify that the top two are sliced based on

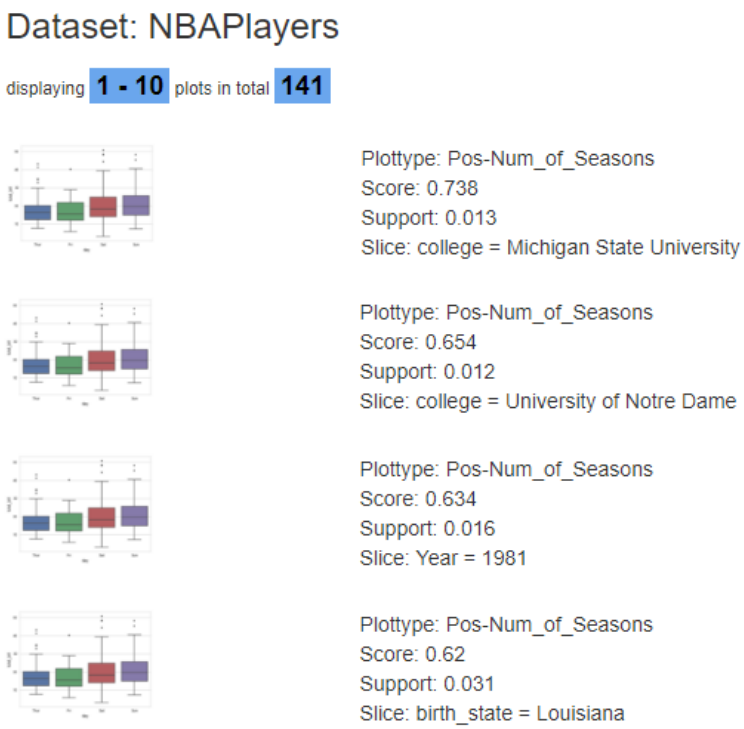


Figure 4.16: Query response for $\boldsymbol{Q}$ = 'Pos, Num_of_Seasons', ranked by score.

the college a player attended. The college might be a more crucial factor in this relation, which guides us to query for:

$$\boldsymbol{Q} = \text{'Pos, Num\_of\_Seasons, college'}$$

and the query is automatically transformed to the Solr query language before executing by the analytics system:

$$\boldsymbol{q} = \text{+plottype:Pos-Num\_of\_Seasons +slice:college}$$

which would return all plots from the plot base about Pos and Num_of_Seasons for each college, might also be sliced based on other dimensions like birth state. The resulting plots are listed in Figure 4.17a. For instance, for Michigan State University in Figure 4.17b, the plot has the highest score in this group, for the median career length of players attended this university changes dramatically, from 14 for the power forward (PF) to 3 for power forward - center (PF-C). As to University of Maryland, the variation in the medium values for different positions is relatively small, as indicated in Figure 4.17c. When it comes to the

Figure 4.17: (a) Query response for $\boldsymbol{Q}$ = 'Pos, Num_of_Seasons, college'; Example plots between Pos and Num_of_Seasons for (b) Michigan State University, (c) University of Maryland and (d) Indiana University.

Indiana University (Figure 4.17d), the medium values tend to stable in the range of 7 to 10, and the difference across positions is not significant enough, thus, the score goes to 0.

Example 3: a user would like to explore how the player height influences rebounding, specifically, the relation between player height and the total rebound percentage (TRB%).

Rebounding is an imperative key to winning in basketball. Offensive rebounding leads to more opportunities to score, while defensive rebounding prevents the opponent from attempting more field goals. The total rebounds will take account of both. Total rebound

percentage is an estimate of the percentage of available rebounds a player grabbed while he was on the floor. It is calculated by Formula 4.2:

$$TRB\% = \frac{TRB * (Tm\ MP/5)}{MP * (Tm\ TRB + Opp\ TRB)} * 100 \tag{4.2}$$

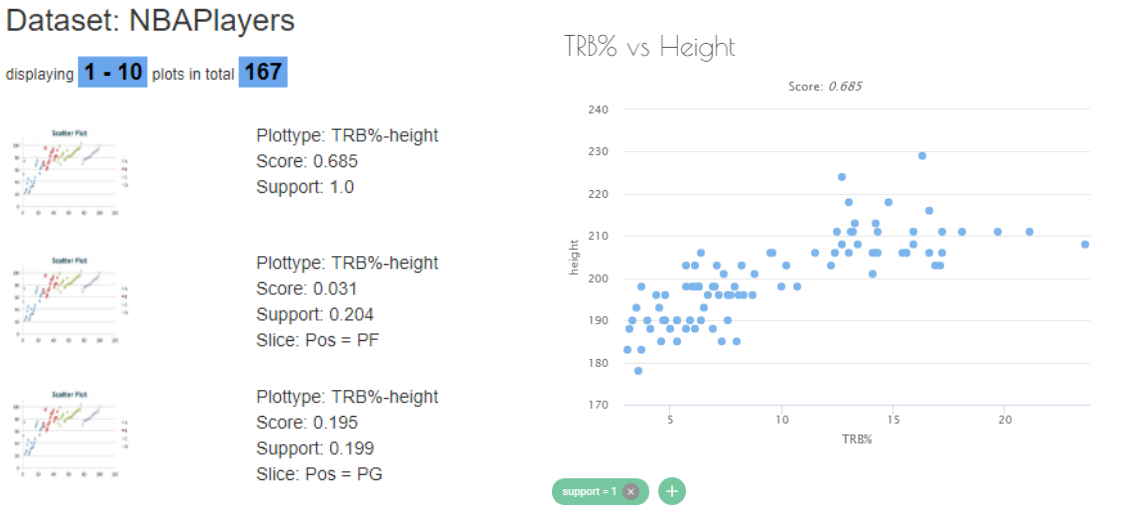
where *TRB* = Total Rebounds, *MP* = Minutes Played, *Tm MP* = Team Minutes Played, *Tm TRB* = Team Total Rebounds and *Opp TRB* = Opponent Total Rebounds. The question is whether or not height truly matters when it comes to rebounding. To find answer to this question, we use the search query language of the system to compose the following query:

$$\boldsymbol{Q} = \text{'TRB\%, height'}$$

with the corresponding low-level Solr query to search the underlying plotbase.

$$\boldsymbol{q} = \text{plottype:TRB\%-height}$$

which will return all plots about the relationship between the total rebound percentage (TRB%) and the player height, possibly sliced by other attributes, like the season, position played in a game, and the birth city, birth state and college of a player. When we explore those returned plots based on their support, we can get a list of plots displayed in Figure 4.18a.

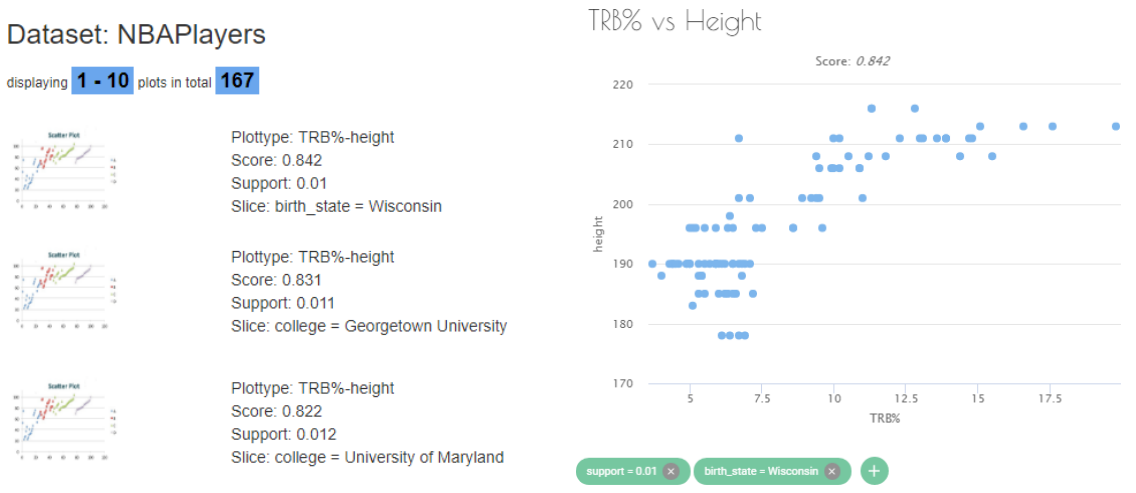


(a) (b)

Figure 4.18: (a) Query response for $\boldsymbol{Q}$ = 'TRB%, height' ranked by support; (b) the relationship between TRB% and height over the whole dataset.

From this list, we can make two remarks: (i) In general, these two variables, TRB% and height, are correlated, for the score for the largest data slice with support equal to 1 is 0.685. The trend can be verified in Figure 4.18b, the data points follow a line, that is, TRB% is positively correlated with player height, the higher a player, the more likely he can grab a rebound when it is available. Height is so import for rebounding, and great rebounders tend to be tall and strong from real-world experience. (ii) The position is not a good factor to further segregate this relation, as the trend has become less obvious in each subgroup, for instance, the interestingness score reduces to 0.031 and 0.195 for the power forward (PF) and the point guard (PG), respectively. The reason might be every position even each player has very clear division of work, not all positions/players have the chance to rebounding.

Alternatively, when we rank the query response by score, the returned list of plots is displayed in Figure 4.19a, in which we can find the scores of the top three plots are above the general one (0.685 in Figure 4.18b). They are corresponding to the three slices: players born in Wisconsin, players attended Georgetown University and players attended University of Maryland, respectively, and their scores are greater than 0.8, which indicate a strong correlation between the two variables. For players born in Wisconsin, we plot the relationship between TRB% and height in Figure 4.19b.



(a) (b)

Figure 4.19: (a) Query response for $\boldsymbol{Q}$ = 'TRB%, height' ranked by score; (b) the relationship between TRB% and height for players born in Wisconsin

In this subsection, we have exemplified three queries using the simple keyword based search query language introduced in Section 4.5.1. During the data exploration process, the intermediate query responses can guide our search and we can take the clues from the responses to refine our query as we are doing a web search. It is indeed possible to replace the tedious data science work just by search.

4.6 System Scalability

Up to this point, the readers should have a clear understanding of the boundless analytics system after introducing each module of the system, *i.e.*, the data module, the generating module, the indexing module and the searching module, from Section 4.2 to Section 4.5 in detail. Now we know how the system works, we also interested in how the system response to changes. To check the capability of the system in handling changes in terms of size and volume, we also conduct the scalability testing of the system. Two sets of experiments are carried out here: one is to vary the minimum support threshold for large data slices; the other is to change the number of seasons (or number of records) in the dataset. In all experiments, we used the NBA players stats data introduced in this chapter.

4.6.1 Variation in Minimum Support Threshold

In this set of experiments, we decrease the minimum support threshold from 3% to 1%, with a step size of 0.5%. As we learned from frequent itemset mining, the decrease of the minimum support threshold leads to an exponential increase in the number of frequent itemsets, which consequently results in an exponential increase in runtime during the mining process. In boundless analytics system, the decrease of minimum support might lead to an exponential increase in the number of generated plots. To check the scalability of the system, we keep track of the number of plots generated and the runtime at Phase I and Phase II (refer to Section 4.3.1 about the two phases) for each specific minimum support value of

Table 4.1: System scalability in the variation of the minimum support threshold (*runtime* in minutes)

minsup(%)	Phase I		Phase II		Total	
	nplots	*runtime*	*nplots*	*runtime*	*nplots*	*runtime*
3	36,745	310	81,918	613	118,663	923
2.5	49,721	315	116,274	679	165,995	994
2	91,673	414	203,162	891	294,835	1,305
1.5	143,929	513	323,127	1,126	467,056	1,639
1	183,122	562	434,121	1,312	617,243	1,874

the large data slices, while the column "Total" calculates the sum from both phases. The results are listed in Table 4.1, where *minsup* is the minimum support threshold, *nplot* and *runtime* are the number of plot objects generated and the computational time of the process, respectively. Note that the runtimes in the table are in minutes.

From this table, we can find that in each phase the number of plots generated increases to about 5 times from the minimum support of 3% to 1%, *i.e.*, from 36,745 to 183,122 in Phase I and from 81,918 to 434,121 in Phase II, while the runtime doubled in this parameter change. In terms of generating those univariate and bivariate plots in the boundless analytics system, the runtime is linear with respect to the number of generated plots.

4.6.2 Variation in Number of Seasons

After the pre-processing by the data module as introduced in Section 4.2.2, the NBA players stats data has 66 seasons, from 1952 to 2017. We vary the number of seasons in the data, *i.e.*, 20%, 40%, ..., 100% of all seasons. For 20% of the seasons, only the most recent 13 seasons ($66 * 20\% \simeq 13$) from 2005 to 2017 are selected, while for 40% of the seasons, the most recent 26 seasons since 1992 are selected, and so on and so forth.

In this set of experiments, we fix the minimum support threshold for large data slices to be 2% and change the number of seasons incorporated in the analysis, from 20% to 100%, with a step size of 20%. This change varies the number of records in the database. In this scenario, we keep track of the number of records in both the *Players* table and *SeasonsStats* table as well as the runtime at Phase I and Phase II for each database instance, for the number

Table 4.2: System scalability in the variation of the number of seasons ($runtime$ in minutes)

$nseasons(\%)$	$num\_of\_records$		$runtime$		
	Players	*SeasonsStats*	Phase I	Phase II	Total
20	956	5,772	80	148	228
40	1,515	10,660	203	410	613
60	2,044	14,961	347	704	1051
80	2,339	17,328	404	852	1256
100	2,569	18,597	414	891	1305

of generated plots is more or less the same through the changes. The results are recorded in Table 4.2, where $nseasons$ is the number of seasons in percentage, $num\_of\_records$ is the number of records in a relational table, and the $runtime$ keeps the computational time in minutes.

As the relationship table *SeasonsStats* has much more number of records (see Table 4.2) and attributes (check the database schema given in Section 4.2.1) than the entity table *Players*, the influence of the number of records on the computational time is dominated by table *SeasonsStats*. From Table 4.2, we can verify that the $runtime$s for both phases increase linearly with the increase of the number of records in table *SeasonsStats*.

4.7 Summary

In this chapter, we built a novel boundless analytics system to automatically analyze large multidimensional dataset and isolate relations which have huge potential interest, which is demonstrated on a real-world NBA player stats data for over 3000 players since 1950. The system consists of four modules: the data module, the generating module, the indexing module and the searching module. We described each module of the system in detail in this chapter. The data module cleans the data and pre-aggregates the data based on the entity-relationship modeling to supplement entities with new, unanticipated, aggregated measures based on their participation as foreign keys in the relationships, which greatly increases the complexity of the data. The generating module is the "engine" of the system,

which systematically explores the attribute-value pair space from all dimensions through the progressive miner and generates all possible univariate and bivariate plot objects for each large data slice. The indexing module indexes those plot objects in two layers, namely, the plot type layer and the plot layer, together with a simple keyword based search query language and the search interface developed in the searching module, to support various full-text search queries on the plots from users. This whole system makes it possible to replace the tedious data exploration process by search. Example queries with query responses on the real NBA stats data validate the effectiveness and usefulness of the system, which can be utilized to analyze other multidimensional datasets as well.

Chapter 5

Conclusion and Future Work

What if data science was as simple as search? Automated data science wound encompass attempts to automate and simplify the cumbersome data science process. Boundless Analytics is such an attempt. In this dissertation, we studied boundless analytics, which is a data mining process with the objective to generate all possible univariate or bivariate plots with sufficient support and isolate potential interesting ones. It can slice a large multidimensional dataset in all possible ways based on its dimensions and derived dimensions and generate an unbounded number of plots. The challenge lies in: (a) how to provide some kind of progress meter in the long mining task to discover all large data slices; (a) how to automatically isolate interesting plots from uninteresting ones from the huge number of generated plots. To solve these problems, we proposed a progressive mining paradigm and designed the ALPINE algorithm (which works in the progressive manner) for large slice (frequent itemset) discovery, which can provide definite guarantees at any time. To rank the generated plots based on their interestingness, we scored the plot based on their spread. Moreover, we built a boundless analytics system with simple keyword based search query language to analyze the large historical NBA player stats data, which showed that it is indeed possible to replace data analytics by search.

5.1 Conclusion

In conclusion, this dissertation has contributed to the following two research areas: frequent itemset mining and automated data science. Let me recap the work in this section.

For frequent itemset mining, especially those long mining tasks, we defined the progressive itemset mining paradigm. A progressive miner progressively divides the itemset search space into sub-spaces with respect to a set of decreasing minimum supports and it guarantees the completeness of the partial solution at any time and has the so-called anytime properties. Thus, it is well suited for solving problems where the search space is large and the quality of the results can be compromised. In addition, we designed a compact representation, namely, *itemset interval*, which utilizes the itemset closure operator to further reduce the itemset search space and speed up the data mining process.

Based on this compact itemset interval representation, we proposed a dynamic approach, ALPINE, for itemset mining that allows us to achieve flexible trade-offs between efficiency and completeness. It works in the progressive manner and proceeds support-wise. The theoretic correctness and the computational effectiveness of ALPINE were presented in the dissertation. In conclusion, ALPINE is to our knowledge the first algorithm to progressively mine all frequent and closed frequent itemsets "support-wise". It guarantees that all itemsets with support exceeding the current checkpoint's support have been found before it proceeds further. It automatically lowers the minimum support on-the-fly without any a priori decided minimum support threshold.

As to automated data science, we proposed the boundless data analytics framework, which automatically slices a given multidimensional dataset in all possible ways and generates very large number of plots. Due to the fact that we can keep deriving new dimensions and facts by SQL group-by aggregation, the boundless analytics process might take an unbounded amount of time. Thus, the progressive mining paradigm is very suitable in this scenario. That's why it was incorporated into boundless analytics.

We denoted an attribute-value pair from the dimensions of a large multidimensional dataset as an item and formulated the problem to find all large data slices as the frequent itemset mining problem, thus, the ALPINE algorithm can be utilized here to slice the data progressively and give us flexible compromises between the mining time and the completeness of the generated data slices. However, the ALPINE algorithm is designed for centrally stored data in one single repository, while a multidimensional dataset might be dispersed across multiple tables. To this end, we employed the foreign-key reference relationship in the entity-relationship model of the multidimensional dataset to slice data across multiple tables and generate the relations or plots for attributes from different tables without the cost-expensive operation of joining tables.

In addition, we built a boundless analytics system, which includes the data module, the generating module, the indexing module and the searching module. The generating module, composed of the progressive data slicer and the plot generator, is the engine of the whole system. The indexing module takes advantage of the full-text search support from the Apache Solr platform to index all generated plot objects. Then the searching module defined based on simple bags of keywords responses all user queries. The system was demonstrated on a large real-world historical NBA player stats dataset of over 3000 players since 1950. The effectiveness and usefulness of the system was validated by real query examples and query responses. With this system, the tedious data science process can be replaced by just simple search.

In summary, the methods explored in this dissertation make steps toward the automation and simplification of the data science process, harnessing the cheaper and cheaper computational resources and reducing the expensive human efforts. We believe these methods should be considered by professional data analysts to enhance or replace current processes that require strong domain expertise and intensive human labor.

5.2 Future Work

Though this dissertation has contributed in the field of frequent itemset mining and automated data science, it also reveals some opportunities for further improvement and extension. Here we just listed a few:

Pattern Sampling. While the progressive mining paradigm has successfully staged a long mining task with minimum support defined milestones or checkpoints to guide the mining process, the exponential computational complexity (in terms of the number of items) of the exhaustive itemset mining problem remains unchanged. One potential improvement is to incorporate highly scalable pattern sampling algorithms into progressive itemset mining, which sacrifices the exactness of the partial solution at any time for further scalability of the algorithm. For instance, the completeness requirement for frequent itemsets at each checkpoint relaxes to be some acceptable range, *i.e.*, greater than or equal to 80%, in exchange for a substantial improvement in the computational efficiency.

Plot Scoring Functions. The Gini coefficient and the Pearson correlation coefficient have been exploited in this thesis to score a univariate or bivariate plot based on its data spread straightforwardly. However, there is no universal interestingness measure of a plot, it's more like a process to fit the needs or expectations of the users. More complicated application-aware or user-in-the-loop measures are worth exploring. Moreover, assume there was some interface for a user to customize his/her scoring function based on the application needs, some entropy or information theory based methods should be studied to find those best-fitting plots from the huge plot base for the end-users.

Dynamic Workflows. Currently, the schema of the input data is kind of pre-defined and static. However, a user might want to modify the data schema after exploring some plots in the response of some initial query. For instance, for the NBA players stats data, a user may discover that the attribute free throws (FT) is perfectly correlated with the attribute free throw attempts (FTA) when exploring those high-score plots returned by the system.

Under this circumstance, he/she would like to delete one of the two attributes. Or after gaining some knowledge about the data through search, an analyst carefully constructed a new schema attribute, the total rebounds dividing by the total number of minutes a player played in a season, and would like to add this feature to the schema. For both cases, of course, the user does not want to modify the schema and then rerun the entire process. A dynamic workflow should be able to handle those dynamic modifications or changes.

Bibliography

[1] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. *ACM SIGMOD Record*, 22(2):207–216, June 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*, pages 487–499, San Francisco, CA, USA, 1994.

[3] David Applegate, Tamraparni Dasu, Shankar Krishnan, and Simon Urbanek. Unsupervised clustering of multidimensional distributions using earth mover distance. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 636–644, New York, NY, USA, 2011.

[4] Benjamin Arai, Gautam Das, Dimitrios Gunopulos, and Nick Koudas. Anytime measures for top-k algorithms on exact and fuzzy data sets. *The VLDB Journal*, 18(2):407–427, April 2009.

[5] Muhammad Gufran Khan Aysha Ashraf. Effectiveness of data mining approaches to e-learning system: A survey. *NFC IEFR Journal of Engineering and Scientific Research*, 4, 2017.

[6] R. Babbie. *The Practice of Social Research*. Cengage learning. Wadsworth Cangage Learning, 2010.

[7] Elena Baralis, Stefano Paraboschi, and Ernest Teniente. Materialized views selection in a multidimensional database. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 156–165, San Francisco, CA, USA, 1997.

[8] Jordi Barretina, Giordano Caponigro, Nicolas Stransky, and et al. The cancer cell line encyclopedia enables predictive modeling of anticancer drug sensitivity. *Nature*, 483(7391):603–607, 2012.

[9] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. In *Noise Reduction in Speech Processing*, pages 1–4. Springer, Berlin, Heidelberg, March 2009.

[10] Mario Boley, Thomas Gärtner, and Henrik Grosskreutz. Formal concept sampling for counting and threshold-free local pattern mining. In *SIAM International Conference on Data Mining*, pages 177–188, 2010.

[11] 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*, pages 582–590, New York, NY, USA, 2011.

[12] Christian Borgelt. Frequent item set mining. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(6):437–456, November 2012.

[13] Lidia Ceriani and Paolo Verme. The origins of the Gini index: extracts from Variabilità e Mutabilità (1912) by Corrado Gini. *The Journal of Economic Inequality*, 10(3):421–443, September 2012.

[14] Badrish Chandramouli, Jonathan Goldstein, and Abdul Quamar. Scalable progressive analytics on big data in the cloud. *Proceedings of the VLDB Endowment*, 6(14):1726–1737, September 2013.

[15] Ada Wai chee Fu, Renfrew Wang wai Kwong, and Jian Tang. Mining N-most interesting itemsets. In *Foundations of Intelligent Systems, ISMIS'00*, Lecture Notes in Computer Science, pages 59–67. Springer, Berlin, Heidelberg, 2000.

[16] Yeounoh Chung, Tim Kraska, Steven Euijong Whang, and Neoklis Polyzotis. Slice Finder: Automated data slicing for model interpretability. In *Proceedings of the SysML Conference*, Stanford, CA, USA, 2018.

[17] Zhi-Hong Deng and Sheng-Long Lv. PrePost$^+$: An efficient n-lists-based algorithm for mining frequent itemsets via children-parent equivalence pruning. *Expert Systems with Applications*, 42(13):5424–5432, 2015.

[18] Zhi-Hong Deng, Zhonghui Wang, and Jia-Jian Jiang. A new algorithm for fast mining frequent itemsets using N-lists. *SCIENCE CHINA Information Sciences*, 55(9):2008–2030, 2012.

[19] Youcef Djenouri, Marco Comuzzi, and Djamel Djenouri. SS-FIM: single scan for frequent itemsets mining in transactional databases. In *Advances in Knowledge Discovery and Data Mining - 21st Pacific-Asia Conference*, pages 644–654, Jeju, South Korea, May 2017.

[20] B S Everitt and A Skrondal. *The Cambridge Dictionary of Statistics; 4th ed.* Cambridge University Press, Leiden, 2010.

[21] Jean-Daniel Fekete and Romain Primet. Progressive analytics: A computation paradigm for exploratory data analysis. *CoRR*, abs/1607.05162, 2016.

[22] Philippe Fournier-Viger, Antonio Gomariz, Ted Gueniche, Azadeh Soltani, Cheng-Wei Wu, and Vincent S. Tseng. SPMF: A java open-source pattern mining library. *Journal of Machine Learning Research*, 15:3389–3393, 2014.

[23] Philippe Fournier-Viger, Jerry Chun-Wei Lin, Bay Vo, Tin Chi Truong, Ji Zhang, and Hoai Bac Le. A survey of itemset mining. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 7(4), 2017.

[24] Enrico Franconi and Ulrike Sattler. A data warehouse conceptual data model for multidimensional aggregation. In *In Proceedings of the Workshop on Design and Management of Data Warehouses*, 1999.

[25] Salvador Garca, Julin Luengo, and Francisco Herrera. *Data Preprocessing in Data Mining*. Springer Publishing Company, Incorporated, 2014.

[26] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2nd edition, 2008.

[27] Michael Glueck, Azam Khan, and Daniel J. Wigdor. Dive in!: Enabling progressive loading for real-time navigation of data visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 561–570, New York, NY, USA, 2014.

[28] Sanjay Goil and Alok Choudhary. High performance multidimensional analysis of large datasets. In *Proceedings of the 1st ACM International Workshop on Data Warehousing and OLAP*, pages 34–39, New York, NY, USA, 1998.

[29] F.J. Gravetter and L.B. Wallnau. *Statistics for the Behavioral Sciences*. Wadsworth, 2009.

[30] Benjamin Haibe-Kains, Nehme El-Hachem, Nicolai Juul Birkbak, Andrew C. Jin, Andrew H. Beck, Hugo J.W.L. Aerts, and John Quackenbush. Inconsistency in large pharmacogenomic studies. *Nature*, 504(7480):389–393, 2013.

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

[32] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 8(1):53–87, January 2004.

[33] Sachin Handiekar and Anshul Johri. *Apache Solr for Indexing Data*. Packt Publishing, 2015.

[34] Nikolaus Hansen, Anne Auger, Olaf Mersmann, Tea Tusar, and Dimo Brockhoff. COCO: A platform for comparing continuous optimizers in a black-box setting. *CoRR*, abs/1603.08785, 2016.

[35] Yu Hirate, Eigo Iwahashi, and Hayato Yamana. TF^{2}P-growth: An efficient algorithm for mining frequent patterns without any thresholds. In *IEEE International Conference on Data Mining*, 2004.

[36] Qiong Hu and Tomasz Imielinski. ALPINE: anytime mining with definite guarantees. *CoRR*, abs/1610.07649, 2016.

[37] Qiong Hu and Tomasz Imielinski. ALPINE: progressive itemset mining with definite guarantees. In *SIAM International Conference on Data Mining*, pages 63–71, 2017.

[38] Tomasz Imieliński, Leonid Khachiyan, and Amin Abdulghani. Cubegrades: Generalizing association rules. *Data Mining and Knowledge Discovery*, 6(3):219–257, July 2002.

[39] Viviane Crestana Jensen and Nandit Soparkar. Frequent itemset counting across multiple tables. In *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 49–61, 2000.

[40] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*. Wiley Publishing, 3rd edition, 2013.

[41] Raymond Kosala and Hendrik Blockeel. Web mining research: A survey. *ACM SIGKDD Explorations Newsletter*, 2(1):1–15, June 2000.

[42] Philipp Kranen and Thomas Seidl. Harnessing the strengths of anytime algorithms for constant data streams. *Data Mining and Knowledge Discovery*, 19(2):245–260, Oct 2009.

[43] Quang Tran Minh, Shigeru Oyanagi, and Katsuhiro Yamazaki. Mining the k-most interesting frequent patterns sequentially. In *Intelligent Data Engineering and Automated Learning*, pages 620–628, 2006.

[44] Bamshad Mobasher, Honghua Dai, Tao Luo, and Miki Nakagawa. Effective personalization based on association rule discovery from web usage data. In *Proceedings of the 3rd International Workshop on Web Information and Data Management*, pages 9–15, New York, NY, USA, 2001.

[45] Jorge J. Moré and Stefan M. Wild. Benchmarking derivative-free optimization algorithms. *SIAM Journal on Optimization*, 20(1):172–191, March 2009.

[46] Stefan Naulaerts, Pieter Meysman, Wout Bittremieux, Trung-Nghia Vu, Wim Vanden Berghe, Bart Goethals, and Kris Laukens. A primer to frequent itemset mining for bioinformatics. *Briefings in Bioinformatics*, 16(2):216–231, 2015.

[47] Eric Ka Ka Ng, Ada Wai-Chee Fu, and Ke Wang. Mining association rules from stars. In *Proceedings of the IEEE International Conference on Data Mining*, pages 322–329, 2002.

[48] Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In *Proceedings of the 7th International Conference on Database Theory*, pages 398–416, London, UK, 1999.

[49] Jian Pei, Jiawei Han, Hongjun Lu, Shojiro Nishio, Shiwei Tang, and Dongqing Yang. H-mine: Hyper-structure mining of frequent patterns in large databases. In *ICDM*, pages 441–448. IEEE Computer Society, 2001.

[50] Paulraj Ponniah. *Data Warehousing Fundamentals: A Comprehensive Guide for IT Professionals*. April 2002.

[51] Cristóbal Romero, José Raúl Romero, José María Luna, and Sebastián Ventura. Mining rare association rules from e-learning data. In *Educational Data Mining*, pages 171–180, 2010.

[52] Murray N. Rothbard. Toward a reconstruction of utility and welfare economics. *On Freedom and Free Enterprise: The Economics of Free Enterprise*, 1956.

[53] Sunita Sarawagi. Explaining differences in multidimensional aggregates. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 42–53, San Francisco, CA, USA, 1999.

[54] Li Shen, Hong Shen, Paul Pritchard, and Rodney Topor. Finding the N largest itemsets. *WIT Transactions on Information and Communication Technologies*, 22:211–222, 1998.

[55] Raphael Silberzahn, Eric Uhlmann, Daniel Martin, Pasquale Anselmi, Frederik Aust, Eli Awtrey, and et al. Many analysts, one dataset: Making transparent how variations in analytical choices affect results. *Advances in Methods and Practices in Psychological Science*, 2018.

[56] Wei Song, Beisi Jiang, and Yangyang Qiao. Mining multi-relational high utility itemsets from star schemas. *Intelligent Data Analysis*, 22(1):143–165, February 2018.

[57] Ramakrishnan Srikant and Rakesh Agrawal. Mining generalized association rules. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 407–419, San Francisco, CA, USA, 1995.

[58] Ramakrishnan Srikant and Rakesh Agrawal. Mining quantitative association rules in large relational tables. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 1–12, New York, NY, USA, 1996.

[59] Charles D. Stolper, Adam Perer, and David Gotz. Progressive visual analytics: User-driven visual exploration of in-progress analytics. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):1653–1662, December 2014.

[60] Fangbo Tao, Kin Hou Lei, Jiawei Han, Chengxiang Zhai, and etc. EventCube: Multi-dimensional search and mining of structured and text data. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1494–1497, New York, NY, USA, 2013. ACM.

[61] Toby J. Teorey. *Database Modeling and Design: The Entity-relationship Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[62] Tea Tušar, Nikolaus Hansen, and Dimo Brockhoff. Anytime benchmarking of budget-dependent algorithms with the coco platform. In *International multiconference Information Society*, pages 1–4, Ljubljana, Slovenia, Oct 2017.

[63] Jeff Ullman and Jennifer Widom. Introduction to SQL. Slides for CS145 - Introduction to Databases, 2007.

[64] Les Underhill and Dave Bradfield. *INTROSTAT*. January 2013.

[65] Takeaki Uno, Tatsuya Asai, Yuzo Uchida, and Hiroki Arimura. LCM: An efficient algorithm for enumerating frequent closed item sets. In *Proceedings of Workshop on Frequent Itemset Mining Implementations*, 2003.

[66] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. LCM ver.3: Collaboration of array, bitmap and prefix tree for frequent itemset mining. In *Proceedings of the 1st International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations*, pages 77–86, New York, NY, USA, 2005. ACM.

[67] Matthijs van Leeuwen. Interactive data exploration using pattern mining. In *Interactive Knowledge Discovery and Data Mining in Biomedical Informatics*, pages 169–182. Springer, Berlin, Heidelberg, 2014.

[68] Panos Vassiliadis. Modeling multidimensional databases, cubes and cube operations. In *Proceedings of the 10th International Conference on Scientific and Statistical Database Management*, pages 53–62, 1998.

[69] Jianyong Wang, Jiawei Han, Ying Lu, and Petre Tzvetkov. TFP: An efficient algorithm for mining top-k frequent closed itemsets. *IEEE Transactions on Knowledge and Data Engineering*, 17(5):652–664, May 2005.

[70] Ying Yang and Geofrey I. Webb. Weighted proportional k-interval discretization for naive-bayes classifiers. In *Proceedings of Advances in Knowledge Discovery and Data Mining: the 7th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 501–512, Seoul, Korea, April 2003.

[71] Mohammed J. Zaki. Scalable algorithms for association mining. *IEEE Transaction on Knowledge and Data Engineer*, 12(3):372–390, May 2000.

[72] Mohammed J. Zaki and Mitsunori Ogihara. Theoretical foundations of association rules. In *3rd ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, Jun 1998.

[73] Shichao Zhang and Chengqi Zhang. Anytime mining for multiuser applications. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 32:515–521, July 2002.

[74] Ning Zhong, Yuefeng Li, and Sheng-Tang Wu. Effective pattern discovery for text mining. *IEEE Transactions on Knowledge and Data Engineering*, 24(1):30–44, January 2012.

[75] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 1996.