Cluster-Driven Navigation of the Query Space

Thibault Sellam *Member, IEEE* and Martin Kersten

Abstract—How can users who know neither programming nor statistics explore large databases? We present a novel interface, designed to guide explorers through their data: Blaeu. Blaeu is a database front-end, “boosted” with unsupervised learning primitives. Thanks to these primitives, it can *summarize* and *recommend queries*. Our first contribution is Blaeu’s interaction model. With Blaeu, users explore the data through *data maps*. A data map is an interactive set of clusters, which users navigate with zooms and projections. Our second contribution is Blaeu’s engine. We present three mapping algorithms, for three different settings. The first algorithm deals with small to medium databases, the second one targets high dimensional spaces and the last one focuses on speed and interaction. We then present an optimization strategy based on sampling. Our experiments reveal that Blaeu can cluster millions of tuples with hundreds of columns in a few seconds on commodity hardware.

Index Terms—Interactive data exploration and discovery, query languages, clustering

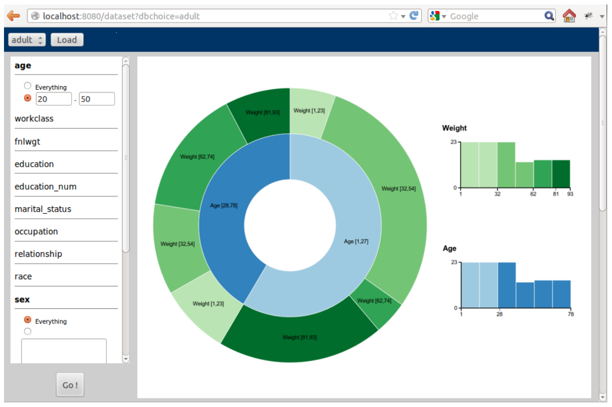
1 Introduction

THE volume of data available to businesses and scientists has exploded. Unfortunately, so have the skills required to exploit this data. The question we are looking into is the following: *How can users who know neither programming nor statistics explore large databases ?*

Query By Example [27] and Tableau [25] are popular examples of database interfaces for non-programmers. With these tools, users can quickly write queries such as “give me the age of customer X” or “show me which products sell better in the Netherlands”. When the users know exactly what they want, these systems can hardly be beaten. Yet, what if they are *exploring* the data? In exploration, the questions are vague (“who are my young customers?”), or general (“what is in my data?”). By definition, explorers do not know what to ask because they do not know their data. Typically, they must resort to trial and error. They start with a candidate query, and tweak it until something interesting pops up [1]. With complex databases, manual trial and error is a tedious process. The data mining community has developed much faster methods for decades. Unfortunately, most data mining packages (R, SPSS, Weka) target statistics-savvy users. Their first-class citizens are control and accuracy. Simplicity is far behind.

This paper presents an attempt to reconcile both worlds. Blaeu is a database interface “boosted” with cluster analysis primitives. With them, the interface can recommend queries and create visual summaries. As a result, *Blaeu lets users explore their data with little prior knowledge and high speed*. Figure 1 displays a screenshot of Blaeu’s interface. A screencast of the working system is available online[1].

Our first contribution is Blaeu’s novel exploration method. Instead of writing queries, the users proceed in a step-by-step, interactive manner. At each step, the system generates a set of clusters called a *data map*. The users can *zoom* into one of these clusters, *project* it on another subspace



localhost:8080/dataset?dbchoice=adult

adult | Load

age: Everything / [✓ 20 - 50]

workclass, fnlwgt, education, education_num, marital_status, occupation, relationship, race

sex: [✓ Everything]

Fig. 1. Screenshot of Blaeu’ s interface

or *rollback*. Thus, they *navigate* the data, without worrying about the nuts and bolts of processing.

Our second contribution is Blaeu’s mapping engine. To create data maps, Blaeu must find clusters in the database. Yet, it must be fast enough to support interaction, and it must rely on few parameters. We present three clustering algorithms which address these requirements. The first algorithm, *SimpleMap*, targets small to medium data sets. The second algorithm, *MultiMap*, supports high dimensional datasets with a multi-view approach. The last algorithm, *LightMap*, focuses on speed and interaction. Additionally, we present an aggressive sampling strategy to accelerate mapping. Thus, Blaeu can cluster millions of tuples in dozens of subspaces within seconds on a commodity laptop. To summarize, here are our contributions:

- We expose the link between query recommendation and cluster analysis
- We present a novel interactive data exploration system based on these results
- We introduce three algorithms to create data maps
- We report an extensive range of use cases and experiments

Thibault Selam and Martin Kersten are with CWI, 123 Science Park, 1098XG, the Netherlands. E-mail: {thibault.sellam, martin.kersten}@cwi.nl

1. http://homepages.cwi.nl/~sellam/blaeu.html

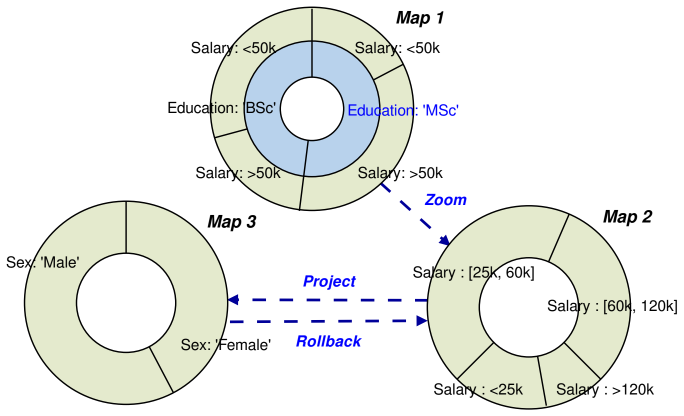


Fig. 2. “Surfing” the data maps

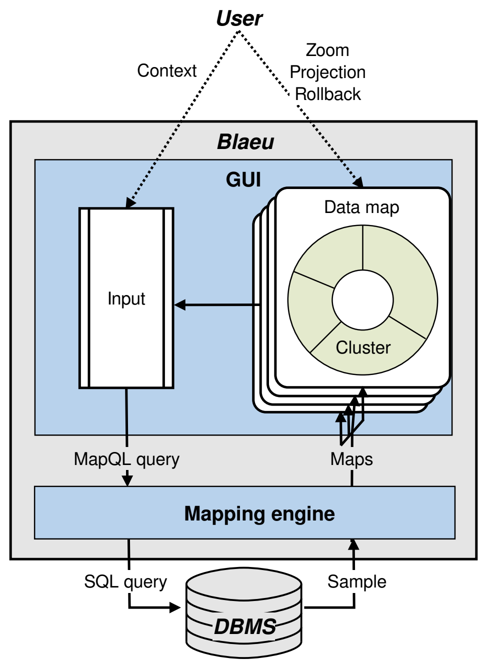


Fig. 3. High level view of Blaeu's architecture.

The rest of this paper is organized as follows. In Sections 3, 4 and 5 present our three algorithms. Section 6 discusses how sampling and recycling accelerate Blaeu's mapping engine. We present use cases and a experiments in Sections 7 and 8. Finally we compare Blaeu to related works in Section 9 and conclude in Section 10.

2 Data Cartography

In this section, we present the most important concept behind Blaeu: the data map. Through this simple abstraction, users can visualize and query their data. We introduce our terminology and our system.

2.1 Overview

Our users are confronted with an unknown database. This database contains a few special, interesting tuples. However, these tuples are hidden among thousands of others. How can we help?

With Blaeu, users write queries in a top-down fashion. They start with a wide Select-Project-Join query. Then, they refine it, step by step. At each step, they visualize the current selection, identify the interesting tuples, and narrow their query. Thus, they drill in iteratively, as they discover the database. In principle, users could carry out top-down exploration through any database front-end. However, raw tables are often difficult to read and the number of potential refinements can be huge. Our idea is to guide the users with cluster analysis. At each step, Blaeu decomposes the current selection into clusters. If the users are interested in one of them, they can click on it. Then, Blaeu updates the selection and creates new clusters. The process is repeated until the users reach the interesting tuples.

Cluster analysis is a classic data mining technique. Its aim is to partition a data set, such that similar items are grouped, and distinct objects are separated [12]. The literature contains dozens of different ways to define a cluster, we will instantiate the definition later in the paper. In our case, this method is helpful in two ways. First, it lets Blaeu *summarize* the users' selection: instead of showing long list of tuples, it displays a few clusters. Second, our system uses clustering to *suggest refinements*. If a tuple is interesting, then its neighborhood probably contains other interesting tuples. Oppositely, if a tuple is irrelevant, its neighbors are likely to be irrelevant too. Therefore, grouping similar tuples helps users separate the interesting tuples from the noise. We will refine these arguments in the following section. First, let's show how Blaeu works in practice.

With Blaeu, users explore their data through **data maps**. A data map is an interactive representation of the clusters in a data set. Figure 2 illustrates how to work with data maps. Suppose that we explore a database which describes the alumni of a fictional university. To seed the process, we provide a wide query. We select the whole database, projected on the columns `Salary` and `Education`. Blaeu clusters the data, and it returns a data map, titled Map 1. The map describes four types of individuals: alumni with a Bachelor's and a low salary, alumni with a Bachelor's and high salary, alumni with a Master's and a low salary, alumni with a Master's and a high salary. The size of the slices represent the count of the clusters. We see that the map summarizes our selection. Furthermore, it gives us options: to refine our selection, we can simply click on one of the regions. Then, Blaeu returns a new map. For instance, if we select the region `MSc`, we obtain Map 2. We call this operation a **zoom**.

Blaeu offers two additional primitives: the **projection** and the **rollback**. The projection does not affect the selection of tuples, but it changes the columns used by the map. If we project Map 2 on `Sex`, we obtain Map 3. With a rollback, we go back to a previous state of the system. Thanks to zooms, projections and rollbacks, we can “surf” from one map to another. Figure 3 gives an overview of the system's components. Users interact with a graphical interface. They specify the seed query with an input menu. They zoom with clicks, and project with drags. Internally, all the user actions are translated into a proprietary language, MapQL.

2.2 Formalization

We introduced the data map, and presented how to interact with it. We now define these concepts formally.

Definition 1. Let $Q \subset DB$ represent a set of tuples in a database DB. We obtain a partitioning $\mathcal{C} = \{C_1, \ldots, C_n\}$ over Q with cluster analysis. A *data map* is a representation of $\mathcal{C}$ which supports zooms, projections and rollbacks.

Our definition contains two degrees of freedom: it is oblivious to our choice of clustering method, and it does not specify how to represent the clusters. These choices depend on practical considerations, discussed in further sections.

Each data map is based on a set of tuples Q. We call this set the **context** of the map. To start an exploration session, the user provides a first context Q_0 explicitly. We name it the **seed context**. The subsequent contexts are obtained iteratively, with zooms, rollbacks and projections. At each step i, these operations transform a context Q_i into a new, refined context Q_{i+1}. Let us define the possible transformations with relational algebra:

Definition 2. Let Q_i describe a set of tuples with primary key k, $i \geq 0$. A data map over Q_i supports three types of operations:

- Zoom in cluster C_j: $Q_{i+1} = \sigma_{t \in C_j}(Q_i)$ pour
- Projection on a set of variables V_p: $Q_{i+1} = \pi_{V_p \cup \{k\}}(Q_i \bowtie Q_0)$
- Rollback: $Q_{i+1} = Q_{i-1}$, undefined if $i < 1$

As we defined the operators in a declarative way, implementation details may vary. For instance, we described the projection with an inner join $Q_i \bowtie Q_0$, to fetch the columns missing from Q_i. In practice, we can bypass this join: we maintain *all* the columns of Q_0 during the whole session, and perform the projections lazily, when we build the maps. This method speeds up the exploration and lets Blaeu operate without any primary key.

Thanks to these definitions, we can formalize Blaeu's expressivity.

Lemma 1. Let $\mathbb{Q}_{SPJ}$ describe the set of all possible Select-Project-Join queries over a database DB. If $clus$ describes all the clusters in the database and col describes all the columns in the database, the set of queries that users can express with zooms, projections and rollbacks is the following:

$$\mathbb{Q}_{Blaeu} = \{\pi_P(\sigma_R(Q)) \mid P \subset col, R \in clus, Q \in \mathbb{Q}_{SPJ}\}$$

Proof: The first query of the session Q_0 is a SPJ query. The property can be derived recursively with definition 2. □

This lemma leads to $\mathbb{Q}_{Blaeu} \subset \mathbb{Q}_{SPJ}$: Blaeu's primitives yield SPJ queries. Yet, not all SPJ queries may be expressed. Blaeu *quantizes* $\mathbb{Q}_{SPJ}$, transforming this large, continuous space into a small and finite set of options.

Blaeu trades control for accessibility. This approach is useful in two cases. First, it helps users with fuzzy, high level queries. Suppose that our users are seeking "young, well paid individuals" in the alumni database. A classic query language requires that they set precise thresholds on the age and the salary; for instance, less than 35 years old and more than \$100,000 per year. Thus, they need some preliminary knowledge, or some investigation time. In contrast, Blaeu automatically partitions the columns age and salary into semantic groups, such that the users can reach the target tuples with a few zooms. Second, Blaeu supports users with *no* query. These users are browsing; they seek interesting tuples but they ignore where to find them. With a classic SQL-based system, they must make guesses. These guesses can lead to empty or overwhelmingly large result sets. With Blaeu, the users obtain a few query suggestions. Thus, they can assess each option in turn, and pick one according to their preferences.

2.3 Properties

According to Definition 1, a data map serves both as output and input. It serves as output because it summarizes the user's current selection. It serves as input because users can refine their queries by clicking on the clusters. The idea to summarize data with clusters has been studied for decades [12]. However, it is less clear how clustering can generate interesting query refinements. In this section, we create a user model, and show under which conditions this property holds.

We model a user by an *utility function* $u : DB \to \{-1, 1\}$. The function takes a tuple t as input, and returns $u(t) = +1$ if t is interesting, $u(t) = -1$ otherwise. To deal with sets, we sum the utility of each tuple. Formally, for a set of tuples $Q \subset DB$, the aggregated utility is $U(Q) = \sum_{t \in Q} u(t)$. Our aim is to suggest interesting subsets of the data. For a given partitioning, if at least one of the partitions is interesting, then we reached our goal. We formalize this objective as follows:

Definition 3. Let set $Q \subset DB$ describe a set of tuples, and let $\mathcal{C}$ represent a partitioning $\{C_1, \ldots, C_n\}$ of Q. We say that $\mathcal{C}$ is **informative** if and only if there exists a $C_j \in \mathcal{C}$ such that $U(C_j) > U(Q)$.

Generating informative partitions would be trivial if we knew our user's utility function. This is not the case, we know almost nothing. However, we can prove the following property: if the interesting tuples are sufficiently close to each other, then recommending *clusters* is a safe bet.

Consider a partitioning of $\mathcal{C}$ over Q, obtained by clustering. From $\mathcal{C}$ we infer a *separation threshold* $\theta(\mathcal{C})$. This function measures how well separated the clusters are. It returns a high value if $\mathcal{C}$'s clusters are tight and far apart. It returns a low value if $\mathcal{C}$'s clusters have a large overlap. We define this threshold as follows:

Definition 4. The **separation threshold** $\theta(\mathcal{C})$ of a partitioning $\mathcal{C}$ is the largest distance such that for any tuple t, for any partition $C_j \in \mathcal{C}$: if at least half of C_j's tuples are within a distance $\theta(\mathcal{C})$ of t, then $t \in C_j$.

We illustrate this property with Fig. 4. Thanks to this notion, we can identify scenarios in which clusters always form interesting recommendations.

Lemma 2. Consider a set of tuples Q with at least one interesting tuple. Let ϕ represent the largest possible distance between two interesting tuples of Q ($\phi = 0$ if there is only one interesting tuple). Any partitioning $\mathcal{C}$ of Q such that $\theta(\mathcal{C}) > \phi$ is informative.

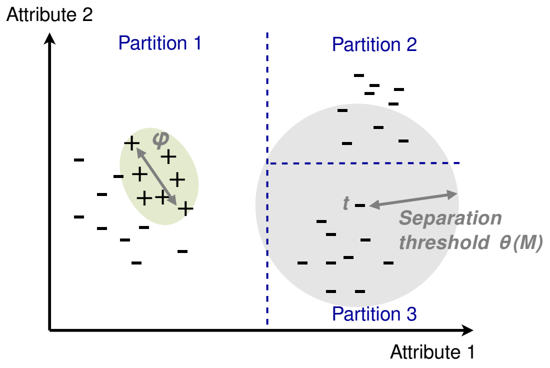


Fig. 4. Example of informative map. Each point represents a tuple in a two dimensional space, the symbols + and - represent the utility, the blue dashed lines represent clusters. The grey area around tuple t represents the separation threshold $\theta(\mathcal{C})$. Most of Partition 3's tuples are within $\theta(\mathcal{C})$ of tuple t. Therefore, t belongs to Partition 3. Conversely, as t does not belong to Partition 2, the majority Partition 2's tuples are beyond $\theta(\mathcal{C})$.

Proof: Let T represent the set of all the interesting tuples. There is at least one tuple t^* in the database such that $u(t^*) = +1$. We note C^* its cluster in $\mathcal{C}$. Now consider a cluster $C \neq C^*$. For the majority of C's tuples t we have $d(t, t^*) > \theta(\mathcal{C})$. If $\theta(\mathcal{C}) > \phi$, then the majority of C's tuples are outside T. Therefore, $U(C) < 0$. Note that $U(Q) = U(C^*) + \sum_{C \in \mathcal{C} \setminus \{C^*\}} U(C)$. We derive the following: $U(C^*) = U(Q) - \sum_{C \in \mathcal{C} \setminus \{C^*\}} U(C) > U(Q)$. In conclusion, the map $\mathcal{C}$ is informative. $\square$

Intuitively, this lemma states that data maps work when the user is consistent (i.e., the interesting tuples are similar to each other) and the data is clustered. If these conditions are met, then the interesting tuples end up in the same cluster, as in Figure 4. Therefore, at least one partition in the map is interesting. Oppositely, if the interesting tuples are far from each other, they may spread over several clusters. In this case, we have no guarantee about the map's interestingness.

To avoid confusion, we insist that Blaeu also works when the users are seeking outliers. According to Blaeu, a subset is interesting if its tuples are close *to each other*. This does not mean that the tuples must be close *to the rest of the data*. Therefore, outliers can very well appear in data maps. For Blaeu, an outlier is simply a small, isolated cluster.

2.4 Representation

We now discuss how to represent clusters. Which information about the clusters should we convey? And which visualization method should we use?

A data map should at least provide one identifier for each cluster. Additionally, it should reflect the structure of the partitions: if we use hierarchical clustering, the map should show the inclusions; if we use flat partitions, it should present the clusters side-by-side. Our implementation describes the clusters with bounding boxes (e.g., `Education: 'MSc', Age < 30`), and it organizes them in a tree (we justify these choices in Section 3). Thus, we instantiate the maps with sunburst charts, known to be efficient in this case [26]. Note that several other visualization methods could qualify, based on trees or even raw text.

In practice, identifiers such as bounding boxes may not provide enough information to assess the content of the clusters. For each partition, Blaeu's front-end provides additional statistics and a few sample tuples. In this paper, we annotate each cluster with a count, conveyed by the size of the slices. Our prototype provides more options: users can request an aggregate or a histogram over any variable in the database. As an illustration, Figure 5 shows how Blaeu displays the distribution of `Salary` for different generations of alumni. In effect, this feature enriches Blaeu's expressivity: it allows grouping and aggregating on top of the queries in $\mathbb{Q}_{Blaeu}$. Thus, users can quickly perform side-by-side comparisons. More generally, this function lets Blaeu emulate traditional OLAP front-ends: the highlighted variable is equivalent to a measure, and the context variables are equivalent to dimensions. In the interface, users request aggregates with mouse-overs, as demonstrated in our screencast[2].

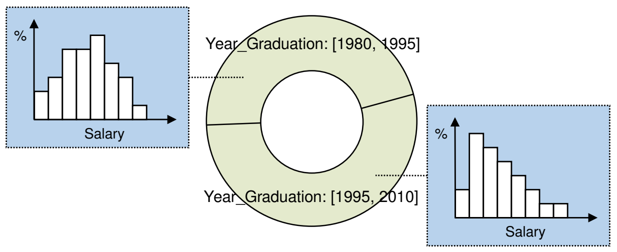


Fig. 5. Example of map, augmented with additional statistics. For each region, we show a histogram of the variable `Salary`.

3 Algorithm 1: Building Maps

In the previous sections, we discussed cluster analysis as an abstract task, we were indifferent to how it was implemented. In this section, we discuss how to make it work in practice. We present a first algorithm, **SimpleMap**, based on existing machine learning methods. We will use this procedure as building block for the following sections.

Blaeu is subject to two contradictory requirements. On one hand, the mapping engine should be accurate, and it should be flexible enough to handle any kind of data, including missing values and categorical attributes. On the other hand, the output should be simple. The descriptions of the clusters must be interpretable by non technicians, and they must be translatable into SQL to express zooms. To avoid setting a fixed number of clusters, Blaeu should return a hierarchy of clusters rather than a flat partitioning. When the clusters are organized in nested sets, users can see several levels of resolution simultaneously. They can experiment with different settings without running the algorithm several times.

The idea behind SimpleMap is to separate *cluster detection* from *cluster description*. We pipeline two algorithms: one to detect the clusters, and another one to generate simpler, hierarchical descriptions. For both steps, we exploit existing work from the data mining literature. First, we cluster the data with a well established algorithm, such as k-means or PAM (Partitioning Around Medoids) [12]. Then we build a classification tree, using the cluster assignments as class labels. We illustrate the procedure with Figure 6.

2. http://homepages.cwi.nl/~sellam/blaeu.html

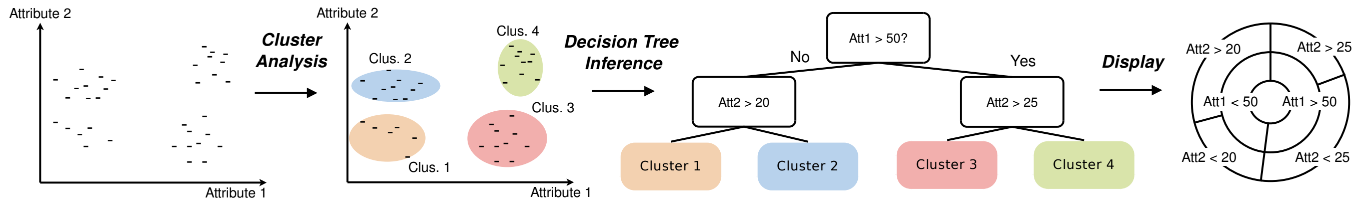


Fig. 6. The SimpleMap algorithm combines cluster analysis and decision tree inference.

```
1: function SimpleMap($DB$, $MAX_c$, $MAX_t$)
2:     $DB' \leftarrow$ preprocess($DB$)
3:     $labels \leftarrow$ PAM($DB', MAX_c$)
4:     $tree \leftarrow$ CART($DB, labels, MAX_t$)
5:     return $tree$
6: end function
```

Fig. 7. Detail of SimpleMap for mixed datasets. SimpleMap relies on a preprocessor (preprocess), an unsupervised learning algorithm (PAM), and a tree inference algorithm (CART). The preprocessor transforms categorical variables into dummy binary variables. PAM takes a dataset DB and a number of clusters MAX_c as input, and returns a cluster assignment for each tuple. CART's input is a dataset DB, an array of class labels $clusters$ and a maximum number of leaves MAX_t; it returns a classification tree.

Thanks to SimpleMap, we can use sophisticated clustering methods without sacrificing interpretability of the results. Another advantage is that we can run the clustering step and the decision tree step on two different versions of the data. This is useful when the data contains categorical variables. During the cluster analysis, we transform these variables into dummy binary variables, such that each binary variable represents one category. Then, we build the decision tree on the original dataset. The cost of SimpleMap is that of chaining two heuristic algorithms: each procedure induces a latency and some inaccuracies.

We present the detail of Simple Map in Figure 7. In our implementation, we use k-means if the data contains only numerical data, and PAM otherwise. PAM is a k-medoid algorithm: for each cluster, it seeks to minimize the distance between all the data points and a central, representative point called medoid. PAM can use any metric, therefore it can cope with a wide range of input data [12]. For the decision tree, we used CART (Classification and Regression Trees). CART operates by splitting the database recursively, minimizing an impurity criterion. This method is well established, it copes with both numerical and categorical data [6].

Our algorithm relies on two parameters: the initial number of clusters to generate MAX_c, and the maximum number of leaves in the decision tree MAX_t. The first parameter can take any value, as long as we generate more clusters than leaves in the tree ($MAX_c > MAX_t$). A higher value leads to more precision, but also longer runtimes. By default, we set $MAX_c = 2MAX_t$. For the second parameter MAX_t, we generate as many leaves as the users' screen can display (by default, 8). If the users wish to work with less partitions, they can use intermediate nodes from the tree.

The total time complexity of SimpleMap depends on the choice of underlying algorithms. Let N represent the number of tuples in the database, and D the number of columns. The complexity of k-means is $\mathcal{O}(DNMAX_c)$. The complexity of the original PAM algorithm is quadratic in N, but we used CLARA [12], a randomized variant which also runs in $\mathcal{O}(NDMAX_c)$. The CART algorithm runs in $\mathcal{O}(DNMAX_t)$. Therefore, the total time complexity of SimpleMap is $\mathcal{O}(DN(MAX_c + MAX_t))$.

4 Algorithm 2: Mapping High-Dimension Data

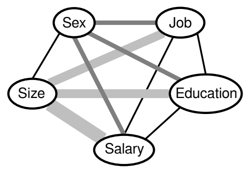
Previously, we presented a first mapping algorithm, SimpleMap. In this section, we extend our model to support datasets with many columns. We show that such large tables cannot be summarized with one map. We introduce **Multimap**, an algorithm to generate *sets* of maps, in which each map uses a different subset of columns.

4.1 Problem Formulation

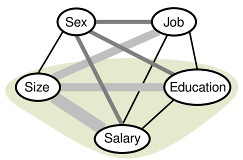
Mapping large tables is challenging for two reasons. First, we must deal with the curse of dimensionality. In high dimensional spaces, all items tend to be equidistant. Therefore, clustering does not make sense. Experimental studies showed that problems start to appear with as little as 10-15 columns [4]. Second, we must consider the diversity of the data. Nothing stops a user from using completely unrelated variables. Clustering a survey on `eye color`, `job` and `tupleID` is irrelevant. Yet, this could very well happen in an exploration scenario. Our solution is to use a multi-view approach: we generate several maps instead of a large one. We operate in two steps. First, we partition the data vertically. Then, we create one map for each partition.

The aim of the vertical partitioning is to detect groups of columns which describe the same aspect of the data. To form the groups, we use statistical dependency. If two variables are perfectly independent, there is little chance that they are related in real life. Oppositely, a perfect correlation means that they measure the same property. To partition the data, we identify sets of columns which are mutually dependent.

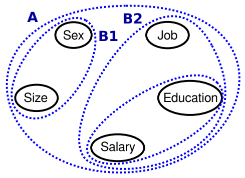
A flexible and robust way to quantify the dependency between two variables is the Variation of Information (VI) [16]. The VI measures the distance between two variables - it is in fact a true metric. It has a low value if the variables are similar, and a high value if the variables are independent. We compute it as follows: if X and Y are two variables, H denotes the entropy and I the mutual information, $VI(X,Y) = H(X) + H(Y) - 2I(X,Y)$. As the VI can only cope with two variables, we introduce the *diameter*. The diameter of a set of variables is the largest VI observed among every pair:



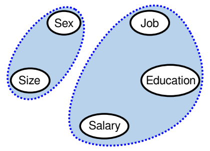
(a) VI graph



(b) Sub-optimal set



(c) Comprehensive sets



(d) Max-simple sets

Fig. 8. The VI graph describes the dependencies between the variables. The vertices represent the variables, the thickness of the edges depicts their weight, i.e., VI.

Definition 5. For a set of variables $V = \{v_1, \ldots, v_d\}$, we define the **diameter** as follows:

$$\text{diameter}(V) = \begin{cases} \max\limits_{v_i, v_j \in V} \text{VI}(v_i, v_j) \text{ if } d \geq 2 \\ 0 \text{ otherwise} \end{cases}$$

To make coherent maps, we need to identify sets of variables with low diameters. We represent this problem with a graph in Figure 8a. In this weighted, undirected graph, the vertices represent the variables, and the edges represent the distances. We want to find partitions inside which the heaviest edge is as light as possible.

Unfortunately, the diameter alone cannot tell us which partitions are the best. If we create one partition per variable, we obtain low diameters but the maps are not satisfying. We want a few sets with many columns but minimal diameters. *We must strike a balance between dependency and cardinality.* Formally, we can express this statement as a multi-objective optimization problem. Let the set V_{DB} contain all the variables of the database, and the set of sets $\mathcal{V}$ describe a partitioning of V_{DB}. We want to solve the following system:

$$\begin{aligned} \text{minimize} \quad & \sum_{V \in \mathcal{V}} \mathit{diameter}(V) \\ \text{minimize} \quad & |\mathcal{V}| \\ \text{subject to} \quad & \bigcup_{V \in \mathcal{V}} V = V_{DB} \\ & V \cap V' = \emptyset \text{ for every distinct } V, V' \text{ in } \mathcal{V} \end{aligned}$$

These two objectives are conflicting: less partitions lead to higher diameters, and lower diameters lead to more partitions.

4.2 Enumerating Candidates

We defined variable grouping as a multi-objective optimization problem: we want a large groups of strongly related variables. As our objectives are conflicting, there is no unique partitioning which satisfies them both. However, there exists a set of solutions for which we cannot improve one objective without degrading the other, the *Pareto front* [9]. The solutions in this set are called Pareto-efficient, we depict them in Figure 9. In this section, we discuss how to compute this Pareto front. We show that the problem can be solved by finding cliques in the VI graph.

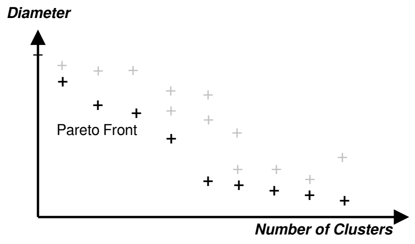


Fig. 9. The Pareto front contains the set of acceptable partitionings.

First, we introduce a useful property. A set of variables is **comprehensive** if we cannot add any variable without increasing its diameter. To illustrate this notion, we use a counter-example. Consider the partition highlighted in Figure 8b. We could add the vertex `Sex` without degrading its diameter, because no edge from `Sex` is heavier than `Size-Salary`. Therefore, the set is not comprehensive.

Definition 6. Let V_{DB} describe the columns of the database and $V_C \subset V_{DB}$ a set of columns.
V_C is *comprehensive* iff.

$$\forall v_i \in V_{DB} \setminus V_C, \text{diameter}(V_C \cup \{v_i\}) > \text{diameter}(V_C)$$

The relation between comprehensive sets and our optimization problem is straightforward.

Lemma 3. The set of partitions $\mathcal{V} = \{V_1, \ldots, V_M\}$ is Pareto-efficient if and only if its elements $V_1, \ldots, V_M$ are comprehensive.

Proof: Suppose that we wish to decrease the total diameter. To achieve this, we must split a partition, which increases the count. Suppose we wish to decrease the count. To achieve this, we must merge partitions. If the sets are comprehensive, this will increase the sum of diameters. Thus, we cannot ameliorate one objective without degrading the other, the set $\mathcal{V}$ is Pareto-efficient. □

Figure 8c shows the comprehensive sets in the example graph, and Figure 8d gives an example of Pareto-efficient partitioning based on this set. Note that any other partitioning based on the comprehensive sets would be valid. We postpone the discussion about the final choice to the end of this Section.

How do we detect comprehensive sets? In fact, there is a tight connection between this problem and *maximal cliques*. Recall that a clique is a tight subset of vertices. In a clique, every pair of vertices is connected with an edge. The clique is maximal if there is no larger clique which contains it. To understand the relationship between comprehensive sets and cliques, we introduce the threshold function t_σ. This function takes a weighted graph as input, and eliminates all the edges with a weight larger than σ.

Lemma 4. Consider a weighted, undirected graph $G = (V, E)$. A set of vertices $U \subset V$ is comprehensive iff. there is a σ such that U is a maximal clique in $t_\sigma(G)$.

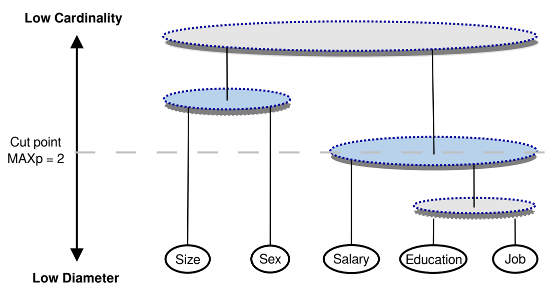


Fig. 10. Dendrogram produced by the complete link algorithm. A node at height σ represents a clique in the graph $t_\sigma(G)$.

```
1: function COMPREHENSIVESETS($columns$, $MAX_p$)
2:     $G \leftarrow$ generateVIGraph($columns$)
3:     $dendrogram \leftarrow \{\}$
4:     while nVertices($G$) $> 1$ do
5:         $(v_i, v_j) \leftarrow \text{Argmin}_{v_i,v_j}\text{edgeWeight}(v_i, v_j)$
6:         $dendrogram \leftarrow dendrogram \cup \{(v_i, v_j)\}$
7:         $G \leftarrow$ mergeNodes($v_i, v_j, G$)
8:     end while
9:     return $dendrogram$
10: end function
```

Fig. 11. Detecting cliques in subsets of dissimilarity graph.

Proof: Let U describe a comprehensive set of vertices with diameter p. Consider the graph $t_p(G)$. The edges between the vertices of U weigh at most p. Therefore U is a clique in $t_p(G)$. Also, one cannot add a vertex of $t_p(G) \setminus U$ without degrading the diameter. Thus, U is a maximal clique. Oppositely, consider a graph $t_\sigma(G)$, and the maximal clique W. Every edge in $t_\sigma(G)$ weighs at most σ. Also, for every vertex in $t_\sigma(G) \setminus W$, there is an edge to W which weighs more than σ. W's diameter is at most σ, and it is comprehensive. □

Thanks to this connection, we can obtain the complexity of our problem.

Lemma 5. Enumerating all the comprehensive sets of a weighted undirected graph is NP-hard.

Proof: Enumerating maximal cliques in a non-weighted, undirected graph is NP-hard [12]. We can reduce this problem to enumerating comprehensive sets. Therefore, enumerating comprehensive sets is NP-hard. To perform the reduction, we map a non weighted graph H to a weighted graph G. For any pair of vertices, we create an edge of weight 0 in G if there is an edge in H. Otherwise, we create an edge with weight 1. If we know the comprehensive sets of G, we can infer the maximal cliques of H in polynomial time. □

Enumerating comprehensive sets is hard. Our solution is to relax our requirements. Consider a set of vertices U in a graph G. If we can find a threshold graph $t_\sigma(G)$ in which U is a clique (not necessarily maximal), then U is "good enough". Finding cliques in threshold graphs is precisely what the Complete Link algorithm does [12]. The Complete Link algorithm is a standard hierarchical clustering algorithm, which operates as follows. First, it creates one partition for each vertex. Then, it finds the two "closest" partitions, and it merges them. The distance between two partitions is the diameter of their union. The procedure is repeated until all the partitions are merged. We present the algorithm in Figure 11.

```
1: function MULTIMAP($DB$, $MAX_c$, $MAX_t$, $MAX_p$)
2:     $columns \leftarrow$ extractColumns($DB$)
3:     $colGroups \leftarrow$ comprehensiveSets($columns, MAX_p$)
4:     $atlas \leftarrow \{\}$
5:     for $colGroup \in colGroups$ do
6:         $db \leftarrow pasteColumns(colGroup)$
7:         $map \leftarrow$ SimpleMap($db, MAX_c, MAX_t$)
8:         $atlas \leftarrow atlas \cup \{map\}$
9:     end for
10:    return $atlas$
11: end function
```

Fig. 12. Detail of MultiMap. The function extractColumns takes a table as input and decomposes it into a set of arrays, where each array represents a column. The function pasteColumn does the opposite: it forms a table from a set of arrays.

The output of the Complete Link algorithm is a tree of nested partitions called dendrogram. Figure 10 displays the dendrogram of our example. A node is drawn at height σ if the partitions it represents forms a clique in the graph $t_\sigma(G)$. To obtain near-Pareto efficient partitionings, we "cut" the tree: we let the user choose a maximum number of partitions MAX_p, and keep the solution which is just under this threshold. The threshold encodes the user's preference for one objective function over the other: a low MAX_p will lead to large partitions, a high value will yield many tight sets. In practice, setting this parameter can be difficult for novice users. We help them with two mechanisms. First, we present the dendrogram explicitly in the interface. This provides visual feedback. Second, we propose default values. Several dendrogram-cutting heuristics were introduced in the clustering literature, such as the Silhouette method [12] or DynamicTreeCut [15]. Although they provide no accuracy guarantee, they form excellent starting points for exploration. We used DynamicTreeCut in our implementation.

4.3 Summary and Complexity

Previously, we presented the algorithm SimpleMap. We can now enrich it to deal with high dimensional spaces. Here is our new framework, MultiMap:

1) Partition the columns of the context into comprehensive sets
2) Create one map for each partition with SimpleMap

In essence we process the columns first, then the rows: we decouple subspace search and clustering [19]. Figure 12 details the algorithm. Recall that N and D describe the number of rows and columns respectively. The complexity of the column partitioning is $\mathcal{O}(ND^2)$, because we need to compute the dependency between every couple of columns and run the Complete Link algorithm. If the procedure generates P partitions, the second step will run in $\mathcal{O}\big(PND(MAX_c+MAX_t)/P\big) = \mathcal{O}\big(ND(MAX_c+MAX_t)\big)$, using the results from Section 3. Thus the total time complexity of MultiMap is $\mathcal{O}\big(ND^2 + ND(MAX_c + MAX_t)\big) \approx \mathcal{O}(ND^2)$. The quadratic term in D indicates a potential

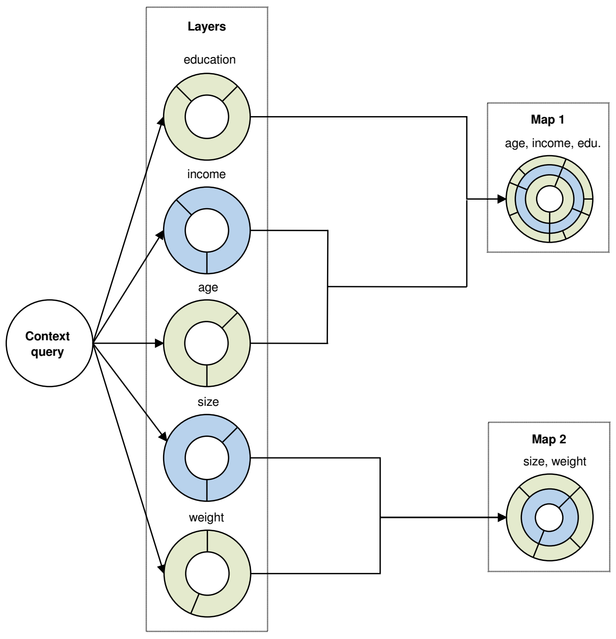


Fig. 13. Overview of LightMap

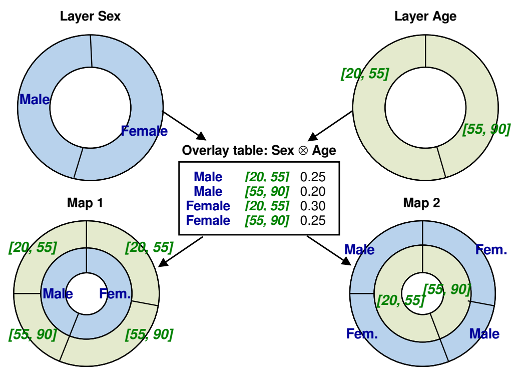


Fig. 14. Two layers, their combination

bottleneck. To bypass it, we compute the VI graph offline, and reuse the vertical partitions across zooms (cf. recycling, discussed in Section 6). Also, our experiments nuance the complexity analysis: as long the the data does not contain more than several hundred columns, the execution time is completely dominated by the map creation step, linear in N and D (as shown in Section 8).

5 Algorithm 3: Lightweight Data Mapping

We now present our third algorithm **LightMap**. LightMap is more flexible than its predecessors: the maps it generates can be modified by the users at little cost. Also, it is very fast. We will see that this comes at a price: the algorithm is less accurate than MultiMap. LightMap operates in three phases. First it creates *layers*. A layer is a map, based on one column only. LightMap creates one layer for each column of the database. Then, it forms groups of similar layers, e.g., layers which describe the same aspect of the data. Finally, it combines the layers of each group into larger maps. This process is illustrated in Figure 13.

The first two steps of LightMap use the primitives we presented earlier. We create the layers with SimpleMap.

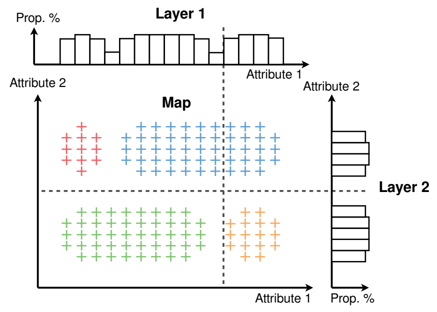


Fig. 15. Limit case of the LightMap algorithm. The data contains four clusters, which full two-dimension structure is lost in the projections.

To group similar layers, we exploit the notion of comprehensive sets discussed in Section 4. However, this time we cluster maps, not columns - in fact, we form clusters of clusterings. During the last step, LightMap forms complex maps by combining layers. To combine two layers, LightMap intersects every partition of one map with the partitions of the other. We call the resulting structure *overlay table*. From this table, LightMap can generate hierarchical partitionings for any permutation of the variables. Consider for instance Figure 14. We present two layers, one based on `Sex`, the other on `Age`. The Figure presents the overlay table in the center. This table lets us compute two maps: one based on `Sex` then `Age`, the other on `Age` then `Sex`. If we keep the overlay table in memory, the users can switch from one representation to the other at little cost, in order to chose the variable ordering which suits them most. By default, we order the layers by clustering quality (obtained during the layer creation phase).

We now summarize the full LightMap procedure:

1) Create one layer for each column in the context
2) Create comprehensive groups of *layers*
3) Overlay the layers in each group

The detail of the algorithm is given in Figure 16. Note that LightMap's partitions are not real clusters. They approximate what the full MultiMap algorithm would have found. We suppose that this approximation is good enough for exploration. Formally, we can justify our scheme with the downward closure property of density. If an area has a high density in d dimensions, its projections on any $d-1$ dimensional subspace is dense. In other words, the clusters do not "disappear" when they are projected to single dimensional spaces, and therefore they appear on the layers [13]. However, the projection incurs a loss of information, because the clusters may be stacked onto each other. Therefore, LightMap can incur mispredictions, as shown in Figure 15. This accuracy penalty is the counterpart for significant speedups.

The respective time complexities of map creation, layer grouping and layer combination are $\mathcal{O}(ND(MAX_c + MAX_t))$, $\mathcal{O}(ND^2)$ and $\mathcal{O}(ND)$, using the results from Sections 3 and 4. Here again, we bypass the layer grouping step by recycling (presented in Section 6). Furthermore, our experiments nuance the complexity analysis: in our implementation, the first phase totally dominates the total runtime, even with several hundred columns (cf. Section 8.2).

```
1: function LightMap($DB$, $MAX_c$, $MAX_t$, $MAX_p$)
2:     /* Generate Layers */
3:     $columns \leftarrow$ extractColumns($DB$)
4:     $layers \leftarrow \{\}$
5:     for $column \in columns$ do
6:         $lay \leftarrow$ SimpleMap($column, MAX_c, MAX_t$)
7:         $layers \leftarrow layers \cup \{lay\}$
8:     end for
9:     /* Form groups of layers */
10:    $layerGroups \leftarrow$ comprehensiveSets($layers, MAX_p$)
11:    /* Aggregate the layers of each group */
12:    $atlas \leftarrow \{\}$
13:    for $layerGroup \in layerGroups$ do
14:        $oTable \leftarrow$ overlayTable($layerGroup$)
15:        $map \leftarrow$ makeMapFromTable($oTable, MAX_t$)
16:        $atlas \leftarrow atlas \cup \{map\}$
17:    end for
18:    return $atlas$
19: end function
```

Fig. 16. Detail of LightMap. The function extractColumns splits a table into a set of arrays, one for each column. The function pasteColumn forms a table from a set of arrays. The function makeMapFromTable generates a map from an overlay table, with at most MAX_t leaves (if necessary, it truncates the overlay table).

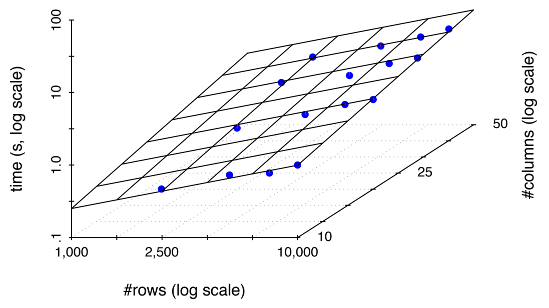


Fig. 17. To calibrate our system, we run Blaeu on synthetic data, measure the runtime and fit a multiplicative model. The points represent the observations, the plane our fitted model. We obtain a correlation coefficient of 0.98, which indicates an almost perfect correlation.

6 Optimization

We improve the latency of Blaeu with two optimizations. First, **we recycle**. We compute the VI graph offline. During the exploration, we do not recluster the variables after each zoom. We run the full procedure for the first query, and reuse the comprehensive sets until the user changes the variables in the context.

Second, **we sample**. For each user action, we only take a few tuples from the context. As we have no preliminary knowledge about the data and we wish to estimate the relative proportions of each group, we use use random sampling with replacement. To pick an optimal sample size, we run a *calibration phase* when Blaeu starts up. During this phase, we create synthetic data sets with different sizes, run Blaeu and measure the time spent. From these observations, we generate a cost model. Thanks to this model, we can pick a maximum sample size given a time objective and a number of columns.

To build our cost model, we fit a multiplicative model on the observed runtime. If $\#rows$ represents the number of tuples, and $\#columns$ the number of columns, our model can be expressed as follows:

$$runtime \approx \alpha \times \#rows^{\beta} \times \#columns^{\gamma}$$

If we compute the logarithm of each operand, the model becomes linear: $log(runtime) \approx log(\alpha) + \beta.log(\#rows) + \gamma.log(\#columns)$. Thanks to this transformation, we can obtain the coefficients with linear regression. Figure 17 pictures an example of calibration session. We used the same hardware as in our Experiments section. We observe that the model fits the observed values almost perfectly.

7 Sample Sessions

We are currently developing Blaeu as a commercial product. A screencast of a working prototype is available online[3].

We now illustrate the system with two "real-life" scenarios. The datasets are available on our website. First, we use a sample of the On-Time database, provided by the US Bureau of Transportation Statistics. The dataset describes delays of US internal flights during January 2010. It contains about 521,000 rows and 91 columns. Our users want to understand the causes of delays. For now, we assume that they know where to begin: they seed the exploration with the columns `Distance` and `ArrDelay`. The first variable `Distance` refers to the distance covered by the flight (in Miles). `ArrDelay` is the delay of the flight (in minutes). We reproduce Blaeu's map in the top-left corner of the figure. Four categories of flights appear: short flights (up to 1167 miles), or longer ones, with or without long delays. Our users decide to zoom in the short delays on short flights, and project to highlight the causes. About a quarter of delays come from the carrier. Another quarter is caused by a late aircraft. They rollback to the first map and zoom into the long delays on short flights. A slightly higher proportion of delays come from weather conditions. Also, the carriers are responsible for more than a quarter of the late flights. They zoom in the longest carrier delays, and project on the name of the companies. They observe that roughly half the delays come from only five carriers. Do they have more delays because they operate more flights? Or should they be suspicious next time they book a flight?

Our second example describes a somewhat more glamorous domain: Hollywood films. Our database describes a few economic indicators for 785 movies released between 2007 and 2012. Our users' focus is abstract and subjective: they are looking for disappointing movies, also called "flops". This time, they do not even know where to begin. Therefore, they run Blaeu on the whole database. Blaeu replies with a set of maps, listed in the top part of Figure 19. Two maps seem like good candidates: the first one, which shows the number of theater for each movie, and the third one, which describes the public and critic's response. Our users pick the first map, and zoom on the movies shown in many cinemas. Then, they switch map, and select those for which the critiques were bad. After three clicks, they already have a list of candidates. With a few more, they could dig further: did these movies generate any income? Did they fail similarly everywhere? Were they expensive? We see Blaeu lets our users "dive" in the data with only a few clicks.

3. http://homepages.cwi.nl/~sellam/blaeu.html

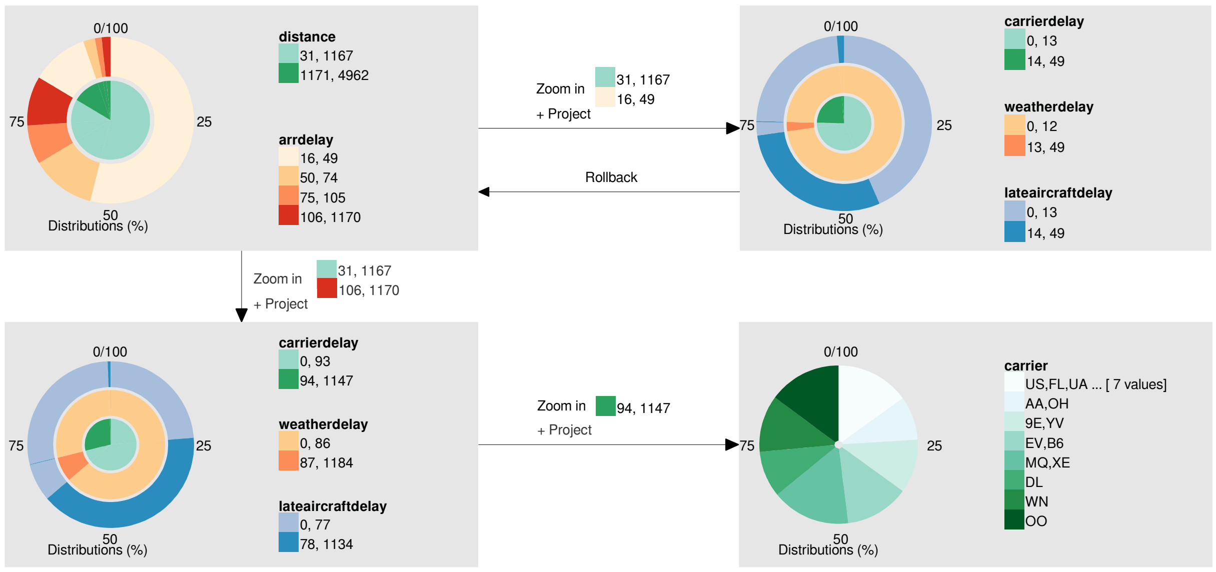
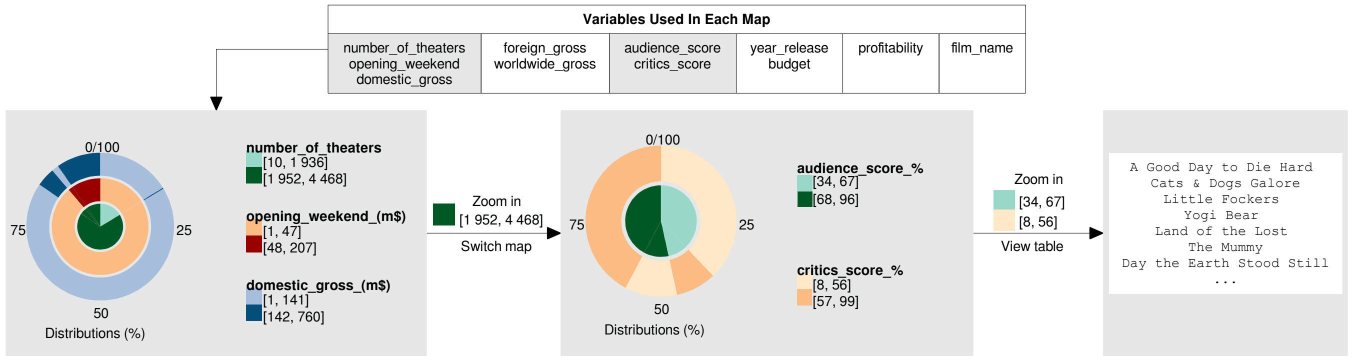


Fig. 18. Running Blaeu on the Ontime database



Variables Used In Each Map					
number_of_theaters opening_weekend domestic_gross	foreign_gross worldwide_gross	audience_score critics_score	year_release budget	profitability	film_name

```
A Good Day to Die Hard
Cats & Dogs Galore
Little Fockers
Yogi Bear
Land of the Lost
The Mummy
Day the Earth Stood Still
...
```

Fig. 19. Running Blaeu on the Hollywood database

8 Validation and Evaluation

Blaeu's suggestions only make sense if they reflect the structure of the data, and it they are computed at interaction time. In this section evaluate the runtime and accuracy of our algorithms. In what follows, we will abbreviate SimpleMap, MultiMap and LightMap as S-Map, M-Map and L-Map respectively.

We confront Blaeu to state-of-the-art algorithms from the clustering literature. We used two families of algorithms: *subspace search* and *subspace clustering*. Methods from the first family return only subspaces, the user is responsible for the actual clustering. Typically, they seek "high contrast" subspaces, that is, subspaces with clusters and outliers. To do so, they enumerate different combinations of variables with level-wise search, and return the top k best candidates. Algorithms from the second family seek clusters and subspaces simultaneously. For each cluster, they return a set of tuples and a set of variables.

To represent the subspace search family, we used CMI [20]. CMI is theoretically well founded, and it resembles Blaeu: like DM, CMI maximizes a mutual information criterion (the Cumulative Mutual Information). To get the actual clusters, we use SimpleMap. For the subspace clustering family, we chose PROCLUS [2] and FIRES [13]. These algorithms have been shown to be fast and accurate [18]. Also, they are close to our approach: like LightMap, FIRES combines one-dimension clusterings.

Contrary to our system, CMI, PROCLUS and FIRES target data miners. They value exhaustivity, while we seek simplicity and speed. Consequently, they rely on a broad range of parameters (up to 9 for FIRES), while we need just one (the maximum tree size). Also, their search space is larger. With these algorithms, a variable can appear in several subspaces, or not at all. With Blaeu, each variable appears exactly once in the whole result set. Our approach may miss some interesting subspaces, but it has two advantages. First, it completely eliminates redundancy in the result set (Blaeu's competitors often return dozens, sometimes hundreds of highly redundant subspaces on small datasets). Second, it reduces latency.

Our code, settings and datasets are available online[4]. We used Open Subspace's implementation of PROCLUS and FIRES [18] (written in Java). For CMI, we used the author's Java implementation[5]. We implemented S-Map, M-Map and L-Map in R 3.0.1, but some of the critical parts are either native C primitives or our own C function. We use

4. `http://homepages.cwi.nl/~sellam/blaeu.html`
5. `http://www.ipd.kit.edu/~muellere/CMI/`

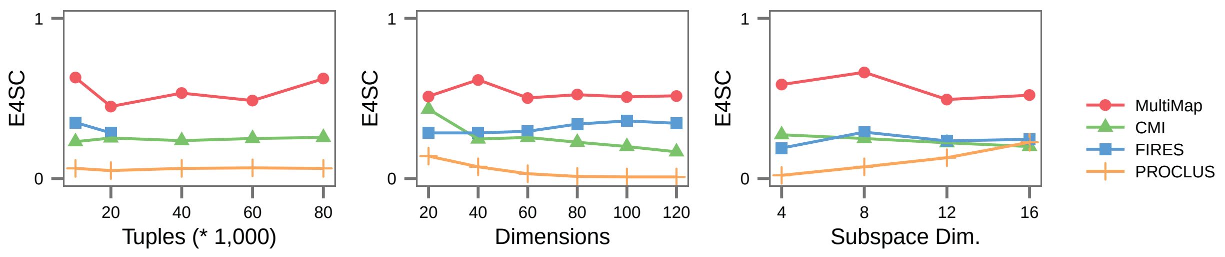


Fig. 20. Accuracy of MultiMap, CMI, PROCLUS and FIRES on synthetic datasets.

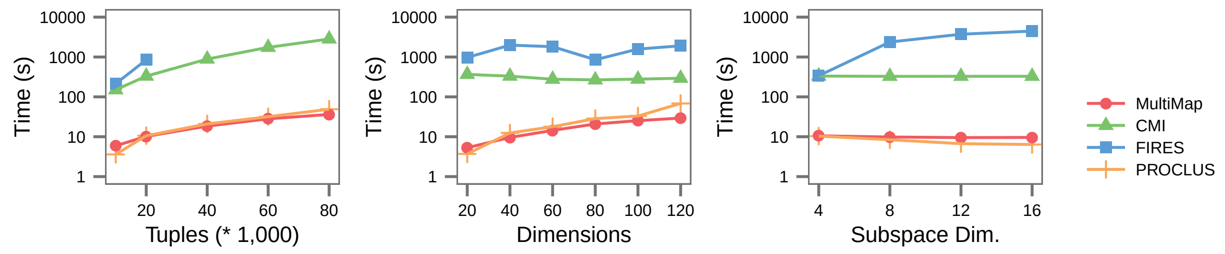


Fig. 21. Execution time of MultiMap, CMI, PROCLUS and FIRES on synthetic datasets.

TABLE 1
Parameters of the Data Generator

Parameter	Distribution	Value
Tuples	Constant	20,000
Columns	Constant	40
Columns per subspace	Uniform	[2,10]
Clusters per subspace	Constant	5
Centroid position	Uniform	[1,100]
Cluster radius	Uniform	[2,10]

TABLE 2
Characteristics of the datasets

Name	Rows	Col.
breast	198	33
diabetes	768	8
communities	1,994	127
internet	7,390	70
pendigits	7,494	16
adult	32,561	14
covertype	581,012	54
gisette	7,000	2,500
mutant1	16,592	4,000

three functions from the R repository: `CLARA` for the cluster analysis, `DynamicTreeCut` to chose a number of subspaces and `rpart` for detection tree inference. The function `CLARA` is a randomized variant of PAM, and `rpart` is an implementation of CART. Our test system is based on a 3.40 GHz Intel(R) Core(TM) i7-2600 processor. It is equipped with 16 GB RAM, but the Java heap space is limited to 12 GB. The operating system is Fedora 16. Unless written otherwise, we disable sampling. The data is stored in MonetDB (release Feb2013), managed by the MonetDB.R Connector [5][17].

8.1 Synthetic Data

In this section, we report the results of experiments on synthetic data. We create datasets for which we know the "truth", and evaluate Blaeu's output. We generate columns by groups, such that *the data is clustered differently on each group of columns*. Therefore, each dataset contains several clustered subspaces. Note that the groups are not overlapping. The clusters are based on multinomial Gaussian distributions, with random parameters. The default options of the generator are reported in Table 1. The subspaces are isolated, and the clusters are well separated: this is an "easy" scenario for Blaeu, but also for its competitors.

We made significant efforts to tune FIRES, PROCLUS and CMI correctly. As the data is synthetic, we can calculate the "ideal" parameters. We report them on our website. Also, recall that S-Map, M-Map and L-Map are hierarchical. For a meaningful comparison, we altered them so that they return a fixed number of clusters. For each setup, we run each algorithm with five different datasets, and report the average performance. For practical reasons, we interrupt the experiments which take more more than 10,000 seconds (2 hours and 46 minutes).

As in the recent subspace clustering literature, we measure clustering quality with a variant of the F1 measure called E4SC [18][10]. The traditional F1 score focuses on tuples only: two clusters are similar if they contain the same data points. With the E4SC, two clusters are similar if they contain the same tuples and they appear on the same dimensions. Therefore, the E4SC considers both the clusters and the subspaces in which they were found. The measure varies between 0 and 1; a high value indicates that the found clusters are similar to the true clusters, a low value shows discrepancies.

Figure 20 compares MultiMap to CMI, PROCLUS and FIRES. We observe that MultiMap is accurate. The results are stable with regards to the number of tuples and columns. In most cases, FIRES comes second, closely followed by CMI, and PROCLUS comes last. Figure 21 shows the runtime of the algorithms. MultiMap and PROCLUS are orders of magnitudes faster than CMI and FIRES. M-Map is almost always faster that PROCLUS. Our approach is validated.

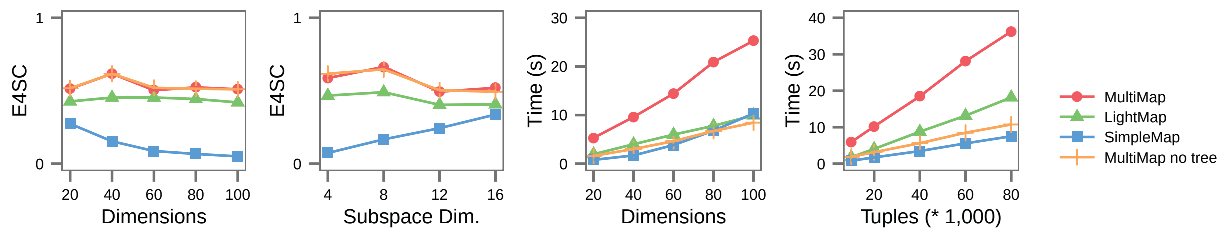
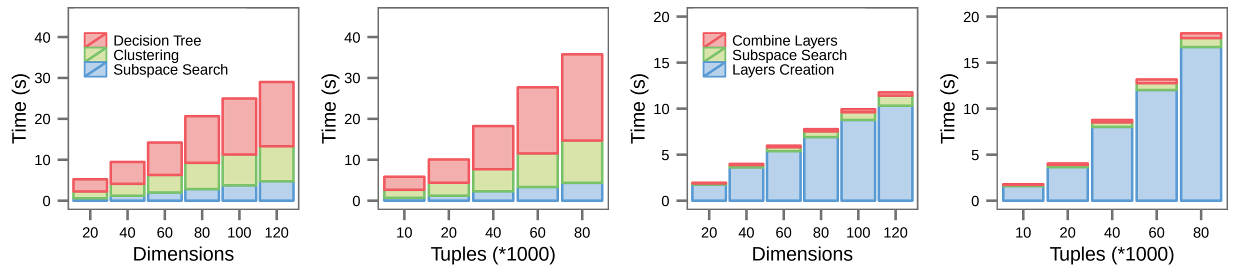


Fig. 22. Comparison of SimpleMap, MultiMap, and LightMap on synthetic datasets.



(a) MultiMap

(b) LightMap

Fig. 23. Execution time breakdown of MultiMap and LightMap.

We compare S-Map, M-Map and L-Map in Figure 22. As SmallMap cannot detect subspaces, it generates poor E4SC scores. Its performance is particularly low when the subspaces have a low dimensionality, and it gets better when the data contains a few wide subspaces. MultiMap is the most accurate algorithm, but it is also the slowest. Recall that MultiMap is based on a combination of cluster analysis an decision tree classification. To separate the errors of both components, we introduced a variant in which we removed the decision tree. We observe very little difference in E4SC, which indicates that the accuracy penalty incurred by the decision tree is acceptable. However, the runtime increases more than twofold. Finally, LightMap is at least twice fast as MultiMap, but this comes at the price of inaccuracies.

Figure 23 details the time consumption during the execution of M-Map and L-Map. In MultiMap, more than half the time is spent building the decision tree. Blaeu spends comparatively very little time clustering the rows and the columns. Fortunately, this step can easily be accelerated by sampling (cf. 8.3). In LightMap, almost all the time is spent creating layers. The subsequent steps are very fast, except when the data contains many columns.

8.2 Real Data

We showed that S-Map, M-Map and L-Map perform well on synthetic data sets. In this section, we run them on nine databases from the UCI repository [3], described in Table 2. As we do not have any base truth, we use a "trick" from the subspace clustering literature: we exploit the class labels originally designed for classification and regression, in the hope that they reflect the structure of the data [18]. We report the F1 between these labels and our predictions. The F1 varies between 0 and 1, higher is better. To tune the algorithms, we generate different sets of parameters and report the best results. For instance, we try 50 combinations for FIRES and 56 for CMI. The details of the experiments are

TABLE 3
Accuracy (F1) for SimpleMap, MultiMap, LightMap, CMI, FIRES and PROCLUS with real datasets

	S.Map	M.Map	L.Map	CMI	FIRES	PROC
breast	*0.20*	0.60	0.53	0.69	0.39	0.58
diabetes	*0.12*	0.55	0.40	0.54	0.42	0.37
comm.	0.62	0.60	*0.56*	0.61	0.89	0.57
internet	*0.27*	0.39	0.44	0.40	*	0.38
pendigits	0.91	0.47	0.37	0.36	*0.22*	0.59
adult	0.61	0.59	0.54	0.56	*	*0.52*
covertype	0.48	0.47	*0.44*	0.48	*	0.50
gisette	*0.34*	0.51	0.42	*	*	*
mutant1	*0.43*	0.64	0.51	*	*	*

TABLE 4
Runtimes (seconds) for SimpleMap, MultiMap, LightMap, CMI, FIRES and PROCLUS with real datasets

	S.Map	M.Map	L.Map	CMI	FIRES	PROC
breast	0.18	0.72	0.08	0.87	*1.01*	0.25
diabetes	0.16	0.35	0.05	0.45	*1.98*	0.26
comm.	0.57	4.53	1.22	3.97	*150.8*	1.65
internet	0.43	2.31	0.54	*16.70*	*	3.17
pendig.	0.46	5.69	0.90	40.82	*94.25*	1.00
adult	0.83	6.00	1.32	*683.81*	*	3.31
covert.	17.88	79.77	47.08	*79,069*	*	12,139
gisette	154.4	*688.2*	177.75	*	*	*
mutant1	960.9	*2,155*	829.21	*	*	*

published on our website. We do not enforce any time limit, but we discard the algorithms which saturate the 12GB heap space limit.

Table 3 reports the accuracy of the algorithms. The highest and lowest values are highlighted. There is no clear winner: each algorithm can perform very well with a dataset, and underperform with an other. For instance, FIRES performs very well with `Communities`, but comes last for `PenDigits`. Similarly, SimpleMap excels with `PenDigits` and `Adult`, but fails with `Breast`, `Diabetes` and `Internet`. CMI and MultiMap are more consistent, and they appear to have similar scores. We observe that MultiMap does not always outperform SimpleMap: when the data contains few columns (less than 16 in our case), MultiMap's vertical partitioning strategy may be too aggressive. LightMap is slightly over-performed by its competitors. In conclusion, Blaeu does not outperform subspace search and clustering algorithms, but its performance is comparable.

Table 4 shows the execution times. Clearly, S-Map and L-Map are the fastest algorithms. They are up to three orders of magnitude faster than CMI and FIRES, and up to five times faster than PROCLUS. In most cases, MultiMap comes third. Also, all the competitors exceed the memory limit for the sets `gisette` and `mutant1`. Remarkably, LightMap overperforms SimpleMap for `mutant1`. We conclude that out algorithms are more efficient than CMI, FIRES and PROCLUS.

8.3 Scaling and Sampling

We now show how sampling helps us deal with very large data sets. For each dataset, we train Blaeu with a small sample, extract the resulting decision tree and apply it to the full dataset. We then compare the result to a base truth: if the labels are similar, then our strategy works. We sampled the data uniformly, with replacement. For each experiment, we took 5 different samples and averaged the results.

Figure 24 shows the results of our experiments with MultiMap on synthetic data. We observe that Blaeu makes considerable progress with the first thousand tuples. Then, the sample size has almost no influence on the accuracy of the results. We conclude that small samples are good enough: Blaeu can very well infer the structure of the data from small subsets of the data. Two factors explain this result. First, Blaeu operates at a very coarse level. It seeks at most a dozen large clusters. Therefore, it is not sensitive to noise and micro-clusters. Second, it works mostly with low dimensional subspaces. Even if the data contains hundreds of columns, a map rarely contains more than a dozen of them. This also explains why the number of variables has such little impact on our results (the top-left and top-right charts are almost similar).

Figure 25 presents a similar experiment with real data. We used two public datasets, available through our Website. Here again, we observe an increase of accuracy with the first few hundred samples. Then, the F1 stays almost constant : Blaeu makes little to no progress. However, we do observe that the F1's variability decreases as the sample size increases.

Considering Blaeu's execution times in both experiments, we conclude that Blaeu can produce high precision maps of million-tuples datasets within two seconds. Note however that our measurements do not include the actual sampling: we assume that Blaeu's back-end can produce samples quickly enough (our back-end MonetDB generates samples in less than a second when the data is "hot")

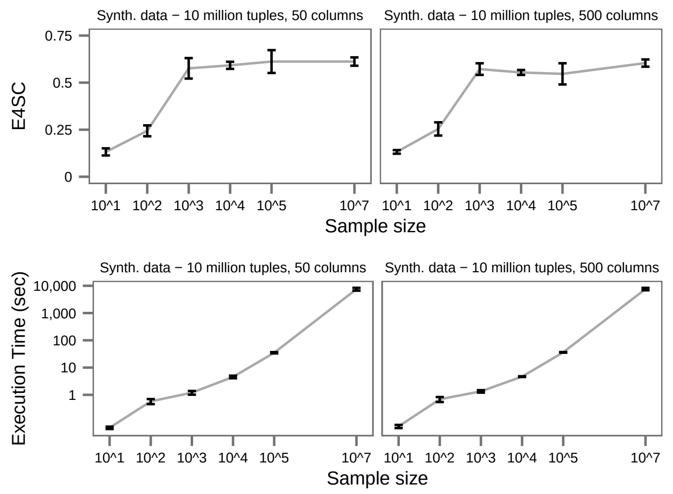


Fig. 24. Accuracy and runtime of MultiMap with sampling on synthetic data. The error bars represent Normal-based 99% confidence intervals.

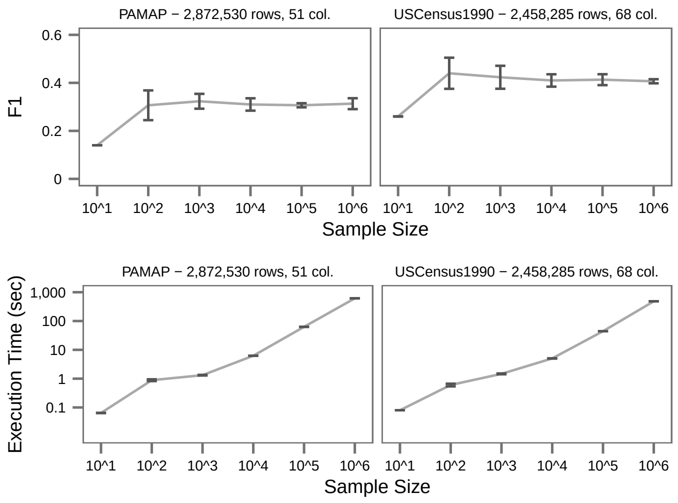


Fig. 25. Accuracy and runtime of MultiMap with sampling on real data. The error bars represent Normal-based 99% confidence intervals.

9 Related work

Researchers have recently shown much interest in data exploration. The research effort is multidisciplinary: proposed studies range from core database technology [24] to visualization [21]. A comprehensive overview of the field is provided by Idreos et al. [11].

OLAP cubes. Drilling and slicing through graphical tools have been common practice for years in the OLAP world. The aim of such tools is to expose how a *measure* varies along several *dimensions*. Currently, Tableau is one of the most popular solutions (based on Polaris [25]). With Tableau, users pick the dimensions and measures with drag-and-drops. Then, the system chooses the optimal representation according to best practices of visualization. Like many OLAP front-ends, Blaeu relies on a graphical query languages. However, it differs on two point. First, it makes recommendations, while most OLAP tools are passive. Second, data maps make no distinction between dimensions and measures. Blaeu can however emulate this behavior, as shown in Section 2.4.

Query recommendation. Several authors have proposed query recommendation algorithms. A common approach is to use the query log of the database [7] Our work does not use it. Therefore, we can drop the assumption that several experts have been using the same database for the exact same purpose. The work of Sarawagi et al. [22] is closer to ours. It recommends queries with statistics. Nevertheless, it is restricted to data cubes, and it uses a very specific notion of "interestingness" based on surprise. Our model is unsupervised, and we assume that *any* subset can be interesting, as long as the tuples are similar to each other. Other systems resembling Blaeu were suggested recently. Our system is the direct descendant of the vision system Charles [23]. Like Blaeu, Charles is based on top-down exploration. However, it does not use any clustering, it only considers partitionings based on medians. Query Steering is related [8], but it models the user rather than the data. Our method is orthogonal, both approaches could be combined. Finally, the vision system SeeDB [21] is close to Blaeu because it choses visualizations automatically. Nevertheless, it focuses on the presentation of the results, not on the user input: it relies on traditional SQL queries.

Clustering. Our work relies on subspace clustering. An exhaustive review of the field was written by Kriegel et al. [14]. We describe the most relevant approaches in Section 8. All the algorithms mentioned in this paper are based on axis-parallel subspaces. We chose to discard more general techniques, e.g., based on affine projections or kernel methods [14], because their results are much harder to interpret, especially for non statisticians.

10 Conclusion

Too often, data exploration tools assume that users know the data and know what they are after. In this paper, we challenge this assumption. We introduce a new mode of interaction, based on data maps. We formalize the problem of generating maps, study its complexity and present our algorithms. Experiments on real and synthetic data reveal that our framework is fast and accurate.

The road for more intelligent interfaces still lies wide open. In future work, we will study how to use other machine learning techniques. We will incorporate data from external sources (e.g. the Web) to improve the queries. Also, we will attempt to personalize the sessions to suit the needs of each user.

References

[1] A. Abouzied, J. M. Hellerstein, and A. Silberschatz. Dataplay: Interactive tweaking and example-driven correction of graphical database queries. In *UIST*, 2012.

[2] C. C. Aggarwal, J. L. Wolf, P. S. Yu, C. Procopiuc, and J. S. Park. Fast algorithms for projected clustering. In *SIGMOD*, 1999.

[3] K. Bache and M. Lichman. UCI machine learning repository, 2013.

[4] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *ICDT*. 1999.

[5] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *Proc. VLDB*, 2009.

[6] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and regression trees*. CRC press, 1984.

[7] G. Chatzopoulou, M. Eirinaki, and N. Polyzotis. Query recommendations for interactive database exploration. In *SSDBM*, 2009.

[8] K. Dimitriadou, O. Papaemmanouil, and Y. Diao. Explore-by-example: An automatic query steering framework for interactive data exploration. In *Proc. SIGMOD*, 2014.

[9] P. Godfrey, R. Shipley, and J. Gryz. Algorithms and analyses for maximal vector computation. *Proc. VLDB*, 2007.

[10] S. Günnemann, I. Färber, E. Müller, I. Assent, and T. Seidl. External evaluation measures for subspace clustering. In *CIKM*, 2011.

[11] S. Idreos, O. Papaemmanouil, and S. Chaudhuri. Overview of data exploration techniques. In *Proc. SIGMOD*, pages 277–281, 2015.

[12] L. Kaufman and P. J. Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 1990.

[13] H.-P. Kriegel, P. Kroger, M. Renz, and S. Wurst. A generic framework for efficient subspace clustering of high-dimensional data. In *ICDM*, 2005.

[14] H.-P. Kriegel, P. Kröger, and A. Zimek. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *TKDD*, 2009.

[15] P. Langfelder, B. Zhang, and S. Horvath. Defining clusters from a hierarchical cluster tree: the dynamic tree cut package for r. *Bioinformatics*, 2008.

[16] M. Meilă. Comparing clusterings – an information based distance. *Journal of Multivariate Analysis*, 2007.

[17] H. Mühleisen. *MonetDB.R - MonetDB to R Connector*, 2013. R package version 0.7.

[18] E. Müller, S. Günnemann, I. Assent, and T. Seidl. Evaluating clustering in subspace projections of high dimensional data. *Proc. VLDB*, 2009.

[19] E. Muller, S. Gunnemann, I. Farber, and T. Seidl. Discovering multiple clustering solutions: Grouping objects in different views of the data. In *ICDE*, 2012.

[20] E. Müller, H. V. Nguyen, F. Keller, K. Böhm, and J. Vreeken. Cmi: An information-theoretic contrast measure for enhancing subspace cluster and outlier detection. In *ICDM*, 2013.

[21] A. Parameswaran, N. Polyzotis, and H. Garcia-Molina. Seedb: Visualizing database queries efficiently. *Proc. VLDB*, 2013.

[22] S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-driven exploration of olap data cubes. *EDBT*, 1998.

[23] T. Sellam and M. Kersten. Meet charles, big data query advisor. *CIDR*, 2013.

[24] L. Sidirourgos, M. L. Kersten, and P. A. Boncz. Sciborq: Scientific data management with bounds on runtime and quality. In *CIDR*, 2011.

[25] C. Stolte, D. Tang, and P. Hanrahan. Polaris: a system for query, analysis, and visualization of multidimensional relational databases. *TVCG*, 2002.

[26] J. Yang, M. Ward, and E. A. Rundensteiner. Interring: An interactive tool for visually navigating and manipulating hierarchical structures. In *InfoVis*, 2002.

[27] M. M. Zloof. QBE/OBE: a language for office and business automation. *Computer*, 1981.

Thibault Sellam received a MSc in computer science and artificial intelligence from Paris Dauphine University, France, in 2010. He has been studying for his PhD at CWI, the Netherlands, since 2011. His research interests include data exploration, data mining and visualization.

Martin Kersten is a full professsor at the University of Amsterdam (the Netherlands) since 1994, and a research fellow at CWI (the Netherlands) since 2011. He is the architect of the MonetDB system, and the founder of several CWI spin-offs. He has more than 140 publications to date, and won a number of awards, including th VLDB 10-year Best Paper Award in 2009 and the ACM SIGMOD Edgar F. Codd Innovations Award in 2014.